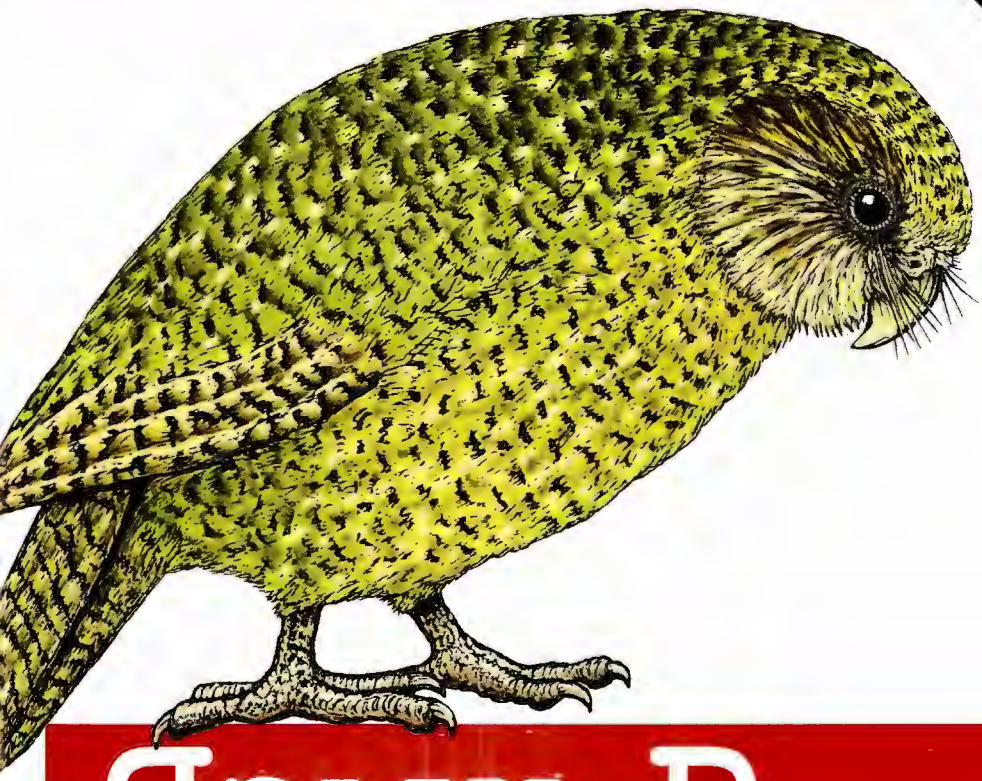
O'REILLY®

ПОЛНОЦВЕТНОЕ ИЗДАНИЕ



Язык R в задачах науки о данных

ИМПОРТ, ПОДГОТОВКА, ОБРАБОТКА, ВИЗУАЛИЗАЦИЯ И МОДЕЛИРОВАНИЕ ДАННЫХ

Хэдли Уикем
Гарретт Гроулмунд

Язык R в задачах науки о данных

ИМПОРТ, ПОДГОТОВКА, ОБРАБОТКА, ВИЗУАЛИЗАЦИЯ И МОДЕЛИРОВАНИЕ ДАННЫХ

R for Data Science

IMPORT, TIDY, TRANSFORM, VISUALIZE, AND MODEL DATA

Hadley Wickham
and Garrett Grolemund

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

Язык R в задачах науки о данных

ИМПОРТ, ПОДГОТОВКА, ОБРАБОТКА, ВИЗУАЛИЗАЦИЯ И МОДЕЛИРОВАНИЕ ДАННЫХ

Хэдли Уикем
Гарретт Гроулмунд

ДИАЛЕКТИКА

Москва · Санкт-Петербург
2018

ББК 32.973.26-018.2.75
У35
УДК 681.3.07

Компьютерное издательство "Диалектика"

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Уикем, Хэдли, **Гроулмунд**, Гарретт.

У35 Язык R в задачах науки о данных: импорт, подготовка, обработка, визуализация и моделирование данных. : Пер. с англ. — СПб. : ООО "Диалектика", 2018. — 592 с. : ил. — Парал. тит. англ.

ISBN 978-5-9909446-8-8 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly Media, Inc.

Authorized Russian translation of the English edition of *R for Data Science* (ISBN 978-1-491-91039-9), © 2017 Garrett Grolemund, Hadley Wickham.

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

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

Научно-популярное издание

Хэдли Уикем, Гарретт Гроулмунд

Язык R в задачах науки о данных: импорт, подготовка, обработка, визуализация и моделирование данных

ООО "Диалектика", 195027, г. Санкт-Петербург, ул. Магнитогорская, д. 30, лит. А, пом. 848

ISBN 978-5-9909446-8-8 (рус.)

ISBN 978-1-491-91039-9 (англ.)

© 2018, Компьютерное изд-во "Диалектика", перевод, оформление, макетирование
© 2017 Garrett Grolemund, Hadley Wickham

Оглавление

Предисловие 21

Часть I. Предварительный анализ 39
Глава 1. Визуализация данных с помощью ggplot2 41
Глава 2. Рабочий процесс: основы 83
Глава 3. Преобразование данных с помощью пакета dplyr 89
Глава 4. Организация рабочего процесса: скрипты 127
Глава 5. Предварительный анализ данных 131
Глава 6. Организация рабочего процесса: проекты 165

Часть II. Подготовка данных 173
Глава 7. Создание tibble-фреймов с помощью пакета tibble 175
Глава 8. Импорт данных с помощью пакета readr 183
Глава 9. Аккуратизация данных с помощью пакета tidyr 207
Глава 10. Работа с реляционными данными с помощью пакета dplyr 233
Глава 11. Работа со строками с помощью пакета stringr 259
Глава 12. Работа с факторами с помощью пакета forcats 291
Глава 13. Работа с датами и временем с помощью пакета lubridate 307

Часть III. Программирование 329
Глава 14. Работа с каналами с помощью пакета magrittr 333
Глава 15. Функции 343
Глава 16. Векторы 367
Глава 17. Итерирование с помощью пакета purrr 391

Часть IV. Моделирование 421
Глава 18. Базовое моделирование с помощью пакета modelr 425
Глава 19. Создание модели 459
Глава 20. Обработка множества моделей с помощью пакетов purrr и broom 487

Часть V. Коммуникация 513
Глава 21. Язык разметки R Markdown 515
Глава 22. Обмен графикой с помощью пакета ggplot2 533
Глава 23. Выходные форматы R Markdown 567
Глава 24. Организация работы с R Markdown 579
Предметный указатель 583

Содержание

Об авторах 19
Об изображении на обложке 19

Предисловие 21

О чем эта книга 21
Как организована книга 23
О чем не будет говориться в книге 24
Большие данные 25
Python, Julia и прочее 26
Непрямоугольные данные 26
Подтверждение гипотез 27
Что необходимо для работы с книгой 28
R 28
RStudio 28
Библиотека `tidyverse` 29
Другие пакеты 30
Выполнение R-кода 31
Получение справочной и дополнительной информации 32
Электронная версия книги 34
Соглашения, принятые в книге 35
Использование примеров кода 36
Ждем ваших отзывов! 37

Часть I. Предварительный анализ 39

Глава 1. Визуализация данных с помощью `ggplot2` 41

Введение 41
Используемые ресурсы 41
Первые шаги 42
Фрейм данных `mpg` 43
Создание графика 43

Шаблон команды создания графика 45
Упражнения 45
Эстетика визуализации 46
Упражнения 52
Распространенные ошибки 53
Панели 54
Упражнения 56
Геометрические объекты 57
Упражнения 62
Статистические преобразования 64
Упражнения 69
Позиционные настройки 70
Упражнения 76
Системы координат 76
Упражнения 78
Многослойная грамматика графики 79

Глава 2. Рабочий процесс: основы 83

Основы написания кода 83
Что представляют собой имена 84
Вызов функций 85
Упражнения 87

Глава 3. Преобразование данных с помощью пакета `dplyr` 89

Введение 89
Используемые ресурсы 89
Пакет `nycflights13` 90
Основные сведения о пакете `dplyr` 91
Фильтрация строк с помощью функции `filter()` 92
Сравнения 93
Логические операторы 93
Отсутствующие значения 95
Упражнения 96
Перестановка строк с помощью функции `arrange()` 97
Упражнения 98
Выбор столбцов с помощью функции `select()` 99
Упражнения 101
Добавление новых переменных с помощью функции `mutate()` 102

Полезные функции создания объектов 103
Упражнения 106
Получение групповых итогов с помощью функции `summarize()` 107
Объединение нескольких операций с помощью канала 108
Отсутствующие значения 110
Счетчики 111
Полезные итоговые функции 116
Группирование по нескольким переменным 121
Разгруппирование 122
Упражнения 123
Групповое видоизменение (и фильтрация) 124
Упражнения 125

Глава 4. Организация рабочего процесса: скрипты 127

Выполнение кода 128
Диагностические средства RStudio 129
Упражнения 130

Глава 5. Предварительный анализ данных 131

Введение 131
Необходимые ресурсы 131
Вопросы 132
Вариация 133
Визуализация распределений 134
Типичные значения 137
Необычные значения 140
Упражнения 142
Отсутствующие значения 143
Упражнения 145
Ковариация 145
Категориальные и непрерывные переменные 146
Упражнения 152
Две категориальные переменные 153
Упражнения 155
Две непрерывные переменные 155
Упражнения 159
Шаблоны поведения переменных и модели 159
Вызовы функций `ggplot2` 163
Дополнительная информация 164

Глава 6. Организация рабочего процесса: проекты 165
Что является вашим “достоянием” 165
Где находится ваш анализ 167
Пути и каталоги 168
Проекты RStudio 168
Резюме 171

Часть II. Подготовка данных 173

Глава 7. Создание tibble-фреймов с помощью пакета `tibble` 175
Введение 175
Необходимые ресурсы 175
Создание tibble-фреймов 176
Сравнение tibble-фреймов с фреймами `data.frame` 178
Вывод на печать 178
Извлечение поднаборов 179
Взаимодействие с разработанным ранее кодом 180
Упражнения 180

Глава 8. Импорт данных с помощью пакета `readr` 183
Введение 183
Необходимые ресурсы 183
Приступаем к работе 183
Сравнение с базовыми возможностями R 186
Упражнения 187
Синтаксический анализ векторов 188
Числа 190
Строки 192
Факторы 193
Календарные даты, календарные дата и время, время суток 194
Упражнения 196
Синтаксический анализ файлов 197
Стратегия 197
Проблемы 198
Другие стратегии 201
Запись в файл 203
Другие типы данных 205

Глава 9. Аккуратизация данных с помощью пакета tidyr 207
Введение 207
Необходимые ресурсы 208
Аккуратные данные 208
Упражнения 211
Рассредоточение и сведение столбцов 212
Сведение столбцов 213
Рассредоточение столбцов 215
Упражнения 216
Разделение и объединение столбцов 217
Разделение столбцов 218
Объединение столбцов 220
Упражнения 222
Отсутствующие значения 222
Упражнения 224
Учебный пример 225
Упражнения 230
Неаккуратные данные 230

Глава 10. Работа с реляционными данными с помощью пакета dplyr 233
Введение 233
Необходимые ресурсы 234
Пакет nycflights13 234
Упражнения 237
Ключи 237
Упражнения 240
Мутирующие соединения 240
Что представляют собой соединения 242
Внутреннее соединение 243
Внешние соединения 244
Неуникальные ключи 246
Определение ключевых столбцов 248
Упражнения 250
Другие реализации 251
Фильтрующие соединения 252
Упражнения 255
Проблемы соединений 255
Операции над множествами 257

Глава 11. Работа со строками с помощью пакета stringr 259
Введение 259
Необходимые ресурсы 259
Основы работы со строками 259
Длина строки 261
Объединение строк 261
Извлечение подстрок 262
Локали 263
Упражнения 264
Поиск соответствия шаблонам с помощью регулярных выражений 265
Базовые соответствия 265
Упражнения 267
Якоря 267
Упражнения 269
Символьные классы и чередования 269
Упражнения 270
Повторения 270
Упражнения 272
Группирование и обратные ссылки 273
Упражнения 273
Инструментарий 274
Обнаружение совпадений 276
Упражнения 278
Извлечение совпадений 279
Упражнения 281
Разбиение совпадений на группы 281
Упражнения 282
Замена совпадений 283
Упражнения 283
Разбиение строк 283
Упражнения 285
Поиск совпадений 285
Другие типы шаблонов 286
Упражнения 289
Другие применения регулярных выражений 289
Пакет stringi 290
Упражнения 290

Глава 12. Работа с факторами с помощью пакета forcats 291
Введение 291
Необходимые ресурсы 291
Создание факторов 292
Опрос GSS 293
Упражнения 296
Изменение порядка следования факторов 296
Упражнения 302
Изменение уровней факторов 302
Упражнения 305

Глава 13. Работа с датами и временем с помощью пакета lubridate 307
Введение 307
Необходимые ресурсы 308
Создание переменных, описывающих дату и время 308
Создание значений даты/времени на основе строк 309
Создание значений даты/времени на основе отдельных компонентов 310
Создание значений даты/времени на основе других типов 313
Упражнения 314
Компоненты даты/времени 314
Получение компонентов 314
Округление 318
Задание значений компонентов 318
Упражнения 319
Временные промежутки 320
Длительности 321
Периоды 322
Интервалы 324
Резюме 325
Упражнения 326
Часовые пояса 326

Часть III. Программирование 329

Глава 14. Работа с каналами с помощью пакета magrittr 333
Введение 333
Необходимые ресурсы 333
Альтернативы каналам 333
Промежуточные объекты 334

Замена исходного объекта 336
Композиция функций 336
Использование канала 337
Когда не следует использовать канал 338
Другие инструменты, доступные в пакете `magrittr` 339

Глава 15. Функции 343

Введение 343
Необходимые ресурсы 344
Когда следует писать функции 344
Упражнения 347
Функции создаются для компьютеров и людей 348
Упражнения 351
Условное выполнение 351
Условия 352
Множественные условия 353
Стиль кода 354
Упражнение 355
Аргументы функций 356
Выбор имен 358
Проверка значений 358
Точка-точка-точка (...) 360
Ленивые вычисления 361
Упражнения 361
Возвращаемые значения 362
Явные команды возврата 362
Написание функций, пригодных для включения в канал 363
Окружение 364

Глава 16. Векторы 367

Введение 367
Необходимые ресурсы 367
Основные сведения о векторах 368
Важные типы атомарных векторов 370
Логический тип 370
Числовой тип 370
Символьный тип 371
Отсутствующие значения 372
Упражнения 372

Использование атомарных векторов 373
Приведение типов 373
Функции проверки типов 375
Скаляры и правила зацикливания 375
Именование векторов 377
Извлечение элементов 377
Упражнения 379
Рекурсивные векторы (списки) 380
Визуализация списков 381
Извлечение элементов 382
Список специй 383
Упражнения 385
Атрибуты 385
Расширенные векторы 387
Факторы 388
Дата и дата/время 388
Tibble-фреймы 389
Упражнения 390

Глава 17. Итерирование с помощью пакета purrr 391

Введение 391
Необходимые ресурсы 392
Циклы for 392
Упражнения 394
Варианты цикла for 395
Изменение существующего объекта 396
Шаблоны формирования циклов 397
Выходные данные неизвестной длины 398
Последовательность неизвестной длины 399
Упражнения 400
Циклы for и функционалы 401
Упражнения 404
Функции семейства map 404
Сокращения 406
Базовые возможности R 407
Упражнения 408
Обработка ошибок 409
Функции семейства map с несколькими входными переменными 412
Вызов различных функций 415

Функции семейства walk 416
Другие шаблоны циклов for 417
Функции-предикаты 417
Функции reduce и accumulate 418
Упражнения 420

Часть IV. Моделирование 421
Глава 18. Базовое моделирование с помощью пакета modelr 425
Введение 425
Необходимые ресурсы 426
Простая модель 427
Упражнения 435
Визуализация моделей 436
Предсказания 437
Остатки 439
Упражнения 441
Формулы и семейства моделей 441
Категориальные переменные 442
Взаимодействия (непрерывная и категориальная переменные) 445
Взаимодействия (две непрерывные переменные) 449
Преобразования 453
Упражнения 456
Отсутствующие значения 457
Другие семейства моделей 457
Глава 19. Создание модели 459
Введение 459
Необходимые ресурсы 460
Почему бриллианты низкого качества оказались более дорогими 460
Цена и вес 462
Более сложная модель 467
Упражнения 470
Что влияет на количество ежедневных авиарейсов 470
День недели 471
Сезонный субботний эффект 475
Вычисляемые переменные 481
Времена года: альтернативный подход 482
Упражнения 483
Дополнительная информация о моделях 484

Глава 20. Обработка множества моделей с помощью пакетов purrr и broom 487
Введение 487
Необходимые ресурсы 488
Набор данных gapminder 489
Вложенные данные 491
Столбец-список 493
Отмена вложения данных 494
Качество модели 497
Упражнения 499
Столбцы-списки 500
Создание столбцов-списков 502
Создание столбцов-списков на основе вложений 503
Создание столбцов-списков на основе векторизованных функций 504
Создание столбцов-списков на основе многозначных итогов 505
Создание столбцов-списков на основе именованных списков 506
Упражнения 507
Упрощение столбцов-списков 508
Преобразование списка в вектор 509
Отмена вложения 510
Упражнения 511
Приведение данных к аккуратной форме с помощью пакета broom 511

Часть V. Коммуникация 513

Глава 21. Язык разметки R Markdown 515
Введение 515
Необходимые ресурсы 516
Основы R Markdown 516
Упражнения 519
Форматирование текста с помощью Markdown 519
Упражнение 521
Фрагменты кода 521
Имя фрагмента 522
Опции фрагмента 523
Таблица 524
Кеширование 525
Глобальные опции 526

Встроенный код 527
Упражнения 528
Устранение неполадок 528
Заголовок YAML 529
Параметры 529
Библиографические ссылки и цитаты 531
Дополнительная информация 532

Глава 22. Обмен графикой с помощью пакета ggplot2 533

Введение 533
Необходимые ресурсы 534
Подписи 534
Упражнения 537
Аннотации 538
Упражнения 545
Шкалы 545
Метки на осях и ключи легенды 546
Компоновка легенды 549
Замена шкалы 550
Упражнения 556
Изменение масштаба изображения 556
Темы 558
Сохранение графиков 560
Изменение размера рисунка 561
Другие важные опции 564
Дополнительная информация 565

Глава 23. Выходные форматы R Markdown 567

Введение 567
Опции выходных форматов 568
Документы 569
Блокноты 570
Презентации 571
Демонстрационные панели 571
Интерактивность 573
Виджеты 573
Пакет Shiny 575

Веб-сайты 576
Другие форматы 577
Дополнительная информация 577

Глава 24. Организация работы с R Markdown 579

Предметный указатель 583

Об авторах

Хэдли Уикем — старший научный сотрудник компании RStudio и член организации R Foundation. Создаваемые им инструменты (как вычислительные, так и когнитивные) не только упрощают анализ данных и сокращают сроки его выполнения, но и превращают этот вид деятельности в увлекательное занятие. К числу его разработок относятся пакеты для науки о данных (библиотека *tidyverse*: **ggplot2**, **dplyr**, **tidyr**, **purrr**, **readr** и др.) и инструментальные средства разработки пакетов (**roxygen2**, **testthat**, **devtools**). Хэдли написал несколько книг, ведет обучающие курсы и часто выступает с лекциями, пропагандируя использование R в науке о данных. Более подробную информацию о нем можно найти на сайте `http://hadley.nz`.

Гарретт Гроулмунд — специалист в области статистики, преподаватель и разработчик в компании RStudio. Он автор хорошо известного пакета **lubridate**, а также книги *Hands-On Programming with R*.

Учебные курсы Гарретта по языку R на сайтах `DataCamp.com` и `oreilly.com/safari` пользуются большой популярностью, и его часто приглашают выступать с лекциями, посвященными R и науке о данных, в таких известных компаниях, как Google, eBay, Roche и многие другие. В RStudio Гарретт готовит материалы для вебинаров, семинаров и серии шпаргалок по теме R.

Об изображении на обложке

Животное на обложке книги — это *какапо*, или *совиный попугай* (лат. *Strigops habroptila*). Какапо — крупная нелетающая птица, обитающая в Новой Зеландии. Рост взрослых особей может достигать 64 см, вес — до 4 кг. Как правило, цвет оперения — зеленовато-желтый, но может значительно варьироваться у разных особей. Какапо ведут ночной образ жизни, ориентируясь в темноте за счет развитого обоняния. Утрату способности к активному полету компенсируют сильные ноги, благодаря которым какапо способны быстро бегать и взбираться по деревьям гораздо лучше, чем большинство птиц.

Какапо — название этого попугая на языке маори, коренных жителей Новой Зеландии. Какапо является важной частью культуры и мифологии маори, которые употребляли мясо этих птиц в пищу, тогда как перья и кожа шли на изготовление накидок и головных уборов.

Вследствие завоза хищников (в особенности крыс и кошек) в ходе колонизации Новой Зеландии европейцами какапо как вид в настоящее время

находится на грани исчезновения, насчитывая менее 200 особей. Правительство Новой Зеландии предпринимает активные меры для сохранения популяции, создавая специальные охранные зоны на островах, свободных от хищников.

Многие из животных, изображаемых на обложках книг издательства O'Reilly, находятся под угрозой вымирания, и все они представляют ценность для нашего мира. Чтобы узнать о том, каким может быть ваш личный вклад в их спасение, посетите сайт `animals.oreilly.com`.

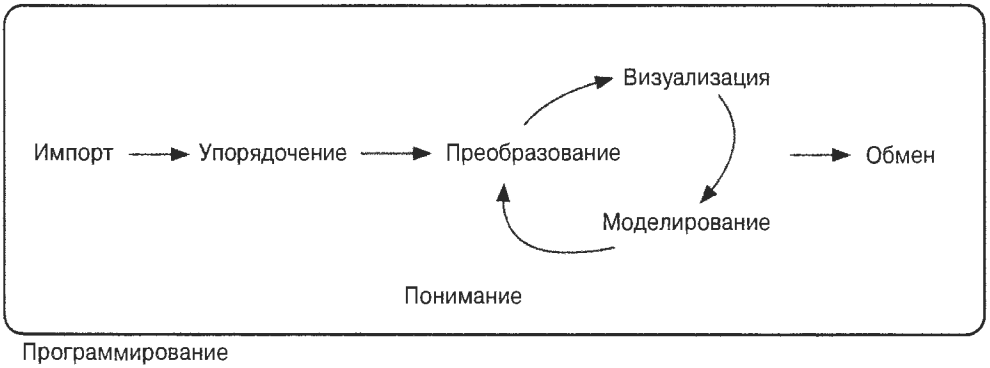
Изображение на обложке взято из книги Вуда *Animate Creation*.

Предисловие

Наука о данных — увлекательная дисциплина, позволяющая превращать необработанные первичные данные в гипотезы и новые знания. Цель книги — помочь вам в изучении наиболее важных средств R, освоив которые вы сможете приступить к профессиональной деятельности в этой области. Прочитав книгу, вы научитесь работать с инструментами, обеспечивающими эффективное применение языка R для решения широкого круга задач науки о данных.

О чем эта книга

Наука о данных — обширная область, для освоения которой недостаточно прочтения только одной книги. Задача этой книги — помочь вам как можно быстрее овладеть наиболее важными инструментами. Наше видение того, какие инструментальные средства могут понадобиться вам в типичном проекте науки о данных, можно проиллюстрировать следующей схемой.



Прежде всего вам потребуется *импортировать* данные в вычислительную среду R. В типичных случаях данные берутся из файла, базы данных или сайта и загружаются во фрейм данных R. Не имея возможности получить свои данные в R, вы не сможете их исследовать!

Как правило, импортированные данные целесообразно предварительно *упорядочить*, т.е. привести к определенному аккуратному виду. Аккуратно организованные данные — это данные, сохраненные в таком виде, который обеспечивает согласованность семантики набора данных со способом их хранения. Другими словами, если ваши данные аккуратно организованы, или, как говорят, *опрятны* (tidy), то каждый столбец представляет переменную, а каждая строка — наблюдение. Понимаемая в этом смысле опрятность данных имеет большое значение, поскольку следование единому принципу их структуризации позволяет вам не тратить каждый раз усилия на приведение их к виду, необходимому для использования совместно с той или иной функцией, а сосредоточиться на задачах, касающихся данных как таковых.

Обычно, после того как данные приведены к аккуратному виду, первое, что необходимо сделать, — это *преобразовать* их. Преобразование данных включает сужение их до набора интересующих вас наблюдений (например, путем отбора только данных, относящихся к населению одного города, или данных за последний год), создание новых переменных, являющихся функциями существующих переменных (например, вычисление скорости по известным расстоянию и времени), и вычисление вспомогательных статистик (например, количества элементов или их среднего). Объединенный процесс приведения данных к аккуратному виду и их преобразования называют *подготовкой* данных.

Для генерации знаний на основе надлежащим образом подготовленных данных существуют два основных механизма: визуализация и моделирование. У каждого из них есть свои сильные и слабые стороны, но они взаимно дополняют друг друга, поэтому при выполнении практического анализа вы будете не один раз попеременно применять каждый из них.

Визуализация информации — важный вид человеческой деятельности. Хорошая визуализация обнажит те стороны явления, которые вы даже не ожидали увидеть, или заставит поставить новые вопросы в отношении данных. Кроме того, хорошая визуализация может подсказать вам, что вы ставите не те вопросы или что требуется сбор других данных. Визуализация способна удивить вас своими результатами, но соответствующие решения масштабируются не слишком хорошо, поскольку интерпретация результатов требует вмешательства человека.

Модели дополняют визуализацию. Если вы достаточно точно сформулируете свои вопросы, то модель поможет вам найти ответы на них. В своей основе модели служат математическим или вычислительным инструментом,

поэтому, как правило, проблем с масштабированием не возникает. Даже если проблемы и появляются, то обычно они решаются покупкой дополнительных вычислительных мощностей. (Мозги ведь все равно не купишь!) Однако любая модель основана на определенных допущениях, которые она сама не может подвергать сомнению. Это означает, что в основном модель ничем не сможет вас удивить.

Завершающий этап науки о данных — это *обмен информацией*, являющийся неотъемлемой частью любого аналитического проекта. Какого бы глубокого понимания данных вы ни достигли с помощью своих моделей и визуализации, это не будет иметь ровным счетом никакого значения, если вы не сможете поделиться результатами с другими людьми.

Программирование играет роль объединяющего начала по отношению ко всем описанным инструментам. Программирование — это всепроникающий инструмент, который можно использовать в любой части проекта. Чтобы заниматься наукой о данных, вовсе не обязательно быть специалистом по программированию, но знание программирования никогда не помешает, поскольку это позволит вам автоматизировать рутинные виды работ и решать новые задачи с гораздо меньшими усилиями.

Описанные инструментальные средства будут использоваться вами в любом проекте, связанном с анализом данных, но чаще всего вы не сможете обойтись только ими. Здесь работает так называемое правило “80/20”: для выполнения примерно 80% объема работ любого проекта вы будете использовать инструменты, которые изучите в этой книге, но для выполнения оставшихся 20% работ вам потребуются другие инструменты. В необходимых случаях в книге даются ссылки на источники, в которых вы сможете узнать больше о дополнительных инструментах.

Как организована книга

Используемые в науке о данных инструменты были описаны выше примерно в том порядке, в каком они будут применяться по мере выполнения анализа (хотя, разумеется, вы будете неоднократно переходить от одного из них к другому). Однако, как показывает наш опыт, изучение инструментов в такой очередности не является оптимальным, что обусловлено следующими причинами.

- Начинать с изучения инструментов, предназначенных для получения данных и приведения их к нужному виду, не совсем целесообразно,

поскольку 80% этой работы — сплошная рутина и скука, тогда как при выполнении остальных 20% вас не будет покидать чувство неопределенности и раздражения. Совершенно неподходящее начало для изучения нового предмета! Вместо этого мы начнем с визуализации и преобразования данных, которые уже импортированы и приведены к надлежащему виду. Благодаря этому самостоятельное выполнение операций импорта и приведения данных к аккуратной форме в качестве подготовки к анализу в будущем не сможет убить ваш энтузиазм, поскольку вы будете знать, что игра стоит свеч.

- Некоторые темы лучше всего объяснять с использованием других инструментальных средств. Например, мы считаем, что понять, как работают модели, вам будет легче, если перед этим вы познакомитесь с визуализацией, приведением данных к аккуратному виду и программированием.
- Средства программирования могут не представлять для вас самостоятельного интереса, но с их помощью вы сможете решать гораздо более широкий круг трудных задач. В середине книги мы познакомим вас с некоторыми программными инструментами, а затем покажем, как их сочетание с традиционными инструментами науки о данных позволяет справиться со многими интересными задачами моделирования.

В каждой главе мы старались следовать одному и тому же подходу: начинать с показательных примеров, чтобы вы могли увидеть общую картину, а затем углубляться в рассмотрение деталей. Каждый раздел книги сопровождается контрольными упражнениями, которые помогут вам закрепить на практике пройденный материал. Не поддавайтесь соблазну пропускать упражнения, поскольку нет лучшего способа обучения, чем решение реальных задач.

О чем не будет говориться в книге

Существует ряд важных тем, которые не рассматриваются в этой книге. Мы считаем исключительно важным сфокусировать ваше внимание исключительно на тех вещах, знание которых позволит вам приступить к самостоятельной работе в кратчайшие сроки. Это означает, что из поля нашего зрения выпадут некоторые важные темы, перечисленные ниже.

Большие данные

В этой книге мы ограничиваемся рассмотрением небольших, умещающихся в памяти компьютера наборов данных. Это неплохо для начала, поскольку справиться с данными большого объема, или, в соответствии с современной терминологией, с "большими данными" (Big Data), без предварительно приобретенного опыта работы с данными небольшого объема вы просто не сможете. Инструменты, которые вы изучите в книге, позволяют легко обрабатывать сотни мегабайт данных, а приложив дополнительные усилия, вы сможете увеличить этот объем до 1-2 Гбайт. Если обычно вам приходится работать с данными большего объема (скажем, 10-100 Гбайт), воспользуйтесь пакетом **data.table** (`https://github.com/Rdatatable/data.table`). В данной книге он не рассматривается ввиду предельной лаконичности его интерфейса, который предлагает слишком мало лингвистических подсказок, из-за чего его изучение представляет определенные трудности. Но если вы часто работаете с большими данными, то выигрыш в производительности стоит того, чтобы затратить усилия на изучение этого пакета.

В случае еще более крупных данных тщательно проанализируйте, не может ли быть так, что проблема больших данных на самом деле является замаскированной проблемой небольших данных. В то время как полный набор данных может быть большим, зачастую для получения ответа на конкретный вопрос достаточно использовать лишь небольшую порцию данных. Возможно, вам удастся найти поднабор меньших размеров, который умещается в памяти и при этом все еще позволяет получить ответ на интересующий вас вопрос. В таком случае наибольшую трудность представляет нахождение подходящего небольшого набора, на что часто требуется не одна попытка.

Другая возможность заключается в том, чтобы попытаться представить задачу с большими данными в виде множества задач меньшего масштаба. Каждая отдельная задача может легко помещаться в памяти, хотя их общее число может исчисляться миллионами. Например, первоначально вы строите модель для каждого объекта в своем наборе данных. Если объектов всего 10 или 100, то это тривиальная задача, но все значительно усложняется при наличии нескольких миллионов объектов. К счастью, каждая задача решается независимо от других (иногда такие ситуации описывают несколько двусмысленным термином *параллелизм*), поэтому вам достаточно воспользоваться одной из подходящих систем распределенных вычислений (например, Hadoop или Spark), позволяющих обрабатывать разные наборы данных

на разных компьютерах. Выяснив с помощью описанных в этой книге инструментов, каким может быть ответ на вопрос, относящийся к одиночному набору данных, вы можете изучить новые инструменты, такие как **sparklyr**, **rhipe** и **dd**, и использовать их для решения задачи, относящейся к полному набору данных.

Python, Julia и прочее

В книге ничего не говорится о Python, Julia или любом другом языке программирования из тех, которые применяются в науке о данных. И не потому, что каждый из этих языков плох в каком-либо отношении. Вовсе нет! На практике в большинстве случаев исследования проводятся с использованием одновременно нескольких языков, включая, как правило, R и Python.

Вместе с тем мы твердо убеждены, что лучше всего осваивать инструменты поочередно. Гораздо эффективнее глубоко изучить что-то одно, чем распылять силы на изучение сразу нескольких тем. Отсюда не следует, что вы должны знать только какой-то один предмет. Это лишь означает, что ваше обучение пройдет значительно быстрее, если вы будете каждый раз концентрироваться только на одном предмете. Вы должны стремиться узнавать новое на протяжении всей своей карьеры, но, прежде чем приступать к изучению очередной интересной темы, вы должны быть уверены в том, что твердо усвоили предыдущий материал.

Мы считаем изучение языка R отличной отправной точкой для увлекательного путешествия в мир науки о данных, поскольку он изначально проектировался для применения в данной области. R — не только язык программирования, но и интерактивная вычислительная среда для исследования данных. В отношении поддержки интерактивности R обладает значительно большей гибкостью по сравнению со многими из его аналогов. Эта гибкость сопряжена и с определенными недостатками, но они компенсируются легкостью введения грамматик, специально приспособленных для выполнения отдельных этапов исследовательского процесса. Эти мини-языки помогают вам мыслить категориями специалиста в области науки о данных и вместе с тем чрезвычайно облегчают ваше взаимодействие с компьютером.

Непрямоугольные данные

В книге рассматриваются исключительно прямоугольные данные: коллекции значений, каждое из которых связано с одной переменной и одним

наблюдением. Существует много наборов данных, не вписывающихся естественным образом в эту парадигму, к числу которых относятся изображения, звуковые данные, двоичные деревья и текст. Но прямоугольные фреймы данных широко распространены в науке и промышленности, и мы считаем, что ваше путешествие в мир науки о данных целесообразно начать именно с них.

Подтверждение гипотез

Процедуры анализа данных можно разделить на два класса: выдвижение гипотез и их подтверждение (также употребляются термины *разведочный анализ* и *подтверждающий анализ*). В этой книге рассматривается исключительно процедура выдвижения гипотез, т.е. разведочный анализ. В ходе анализа вы тщательно изучаете данные и, используя свои знания предметной области, выдвигаете ряд гипотез, предлагающих возможные объяснения того, почему данные ведут себя так, а не иначе. При этом вы оцениваете гипотезы неформально, с позиций здравого скептицизма, основанного на рассмотрении данных под разными углами.

Выдвижение гипотез должно дополняться попытками их подтверждения. Подтверждение гипотез затруднено следующими двумя обстоятельствами.

- Вы должны иметь в своем распоряжении точную математическую модель, позволяющую генерировать опровергаемые гипотезы. Зачастую это требует углубления в дебри статистики.
- Любое наблюдение можно использовать в целях подтверждения гипотезы только один раз. Использование наблюдения более одного раза немедленно возвращает вас обратно к разведочному анализу. Это означает, что еще до того, как вы приступите к процедуре подтверждения гипотезы, вы должны "зарегистрировать" (заранее подготовить в явном виде) свой план анализа и не отклоняться от него, когда ознакомитесь с данными. О выборе стратегий, которые можно использовать для упрощения этой задачи, речь пойдет в части IV.

Общепринято считать моделирование инструментом выдвижения гипотез, а визуализацию — инструментом их подтверждения. Но эта дихотомия не совсем верна: модели часто применяют на этапе подтверждающего анализа, а визуализацию — в разведочном анализе. Ключевым фактором их дифференциации является частота использования каждого наблюдения: если наблюдение рассматривается только один раз, то вы имеете дело с подтверждением

гипотезы; если же наблюдение рассматривается более одного раза, то это означает, что выполняется разведочный анализ.

Что необходимо для работы с книгой

Чтобы чтение книги принесло вам максимальную пользу, установите на своем компьютере описанные ниже ресурсы. Мы рассчитываем на то, что вы математически грамотны; наличие предварительного опыта программирования будет только плюсом. Если до этого вы никогда не занимались программированием, то, возможно, вас заинтересует книга Гарретта *Hands-On Programming with R*, которая отлично дополнит данную книгу.

Для выполнения приведенных в книге примеров вам понадобятся следующие средства: R, RStudio, коллекция пакетов R в виде библиотеки *tidyverse* и довольно большое количество других пакетов, включающих готовые функции, документацию с их описанием и образцы данных.

R

Загрузите R из сети CRAN (от англ. *comprehensive R archive network*). Эта сеть состоит из множества серверов-зеркал, разбросанных по всему миру, и используется для распространения R и пакетов R. Не пытайтесь определить, какой из серверов является для вас ближайшим: просто воспользуйтесь зеркальным облаком `https://cloud.r-project.org`, которое сделает это автоматически.

Каждый год выпускаются одна главная версия R и 2-3 вспомогательные. Целесообразно регулярно обновлять установленную у вас версию. Обновление, особенно в случае главной версии, может доставить определенные неудобства, поскольку потребует переустановки всех пакетов, но отказываться не стоит, ведь лучше идти в ногу с прогрессом.

RStudio

RStudio — это интегрированная среда разработки (IDE) для программирования на R. Чтобы загрузить и установить среду, посетите сайт `http://www.rstudio.com/download`. RStudio обновляется пару раз в год. Если доступна очередная версия, RStudio известит вас об этом. Имеет смысл регулярно обновлять программу, поскольку в таком случае вы сможете воспользоваться

преимуществами новых средств. Для работы с книгой вам потребуется версия RStudio не ниже 1.0.0.

Запустив RStudio, вы увидите окно, разделенное на две основные области.

Пока что вам достаточно знать, что код следует вводить в панели консоли и запускать нажатием клавиши <Enter>. Остальные подробности будут сообщены позже.

Библиотека `tidyverse`

Вам также потребуется установить некоторые из пакетов R. Пакет R — это коллекция функций, данных и документации, расширяющая возможности базовой комплектации R. Использование пакетов является залогом успешной работы с R. Большая часть пакетов, которые вы изучите в книге, входит в состав библиотеки *tidyverse*. Пакеты *tidyverse* разделяют общую философию обработки данных и программирования на языке R и были спроектированы так, чтобы обеспечить их естественное взаимодействие.

Для установки полной библиотеки *tidyverse* понадобится всего одна строка кода:

```
install.packages("tidyverse")
```

Введите этот код в окне консоли и выполните его, нажав клавишу <Enter>. R загрузит пакеты из CRAN и установит их на вашем компьютере. Если в процессе установки возникнут проблемы, убедитесь в том, что вы подключены к Интернету, а адрес `https://cloud.r-project.org/` не заблокирован вашим брандмауэром или прокси-сервером.

Вы не сможете использовать функции, объекты и файлы справки пакета, если не загрузите его с помощью функции `library()`. Это можно сделать в любой момент после установки пакета.

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: tidyr
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages ----------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

Сообщения уведомляют вас о загрузке пакетов **ggplot2**, **tibble**, **tidyr**, **readr**, **purrr** и **dplyr**. Эти пакеты считаются ядром библиотеки *tidyverse*, поскольку вы будете использовать их при выполнении практически любого вида анализа.

Пакеты библиотеки *tidyverse* обновляются довольно часто. Чтобы узнать о наличии обновлений и, по желанию, установить их, воспользуйтесь командой `tidyverse_update()`.

Другие пакеты

Существует множество великолепных пакетов, не являющихся частью библиотеки *tidyverse*, поскольку они предназначены для решения задач, относящихся к другим предметным областям, или проектировались на основе других базовых принципов. Это не делает их лучше или хуже — они просто другие. По мере того как количество ваших проектов, выполняемых в R, будет расти, вы будете знакомиться с новыми пакетами и новыми способами исследования данных.

В этой книге используются три пакета, не входящих в состав библиотеки *tidyverse*:

```
install.packages(c("nycflights13", "gapminder", "Lahman"))
```

Указанные пакеты предоставляют данные об авиарейсах, уровне развития различных стран и бейсбольной статистике. Мы будем использовать для иллюстрации основных идей науки о данных.

Выполнение R-кода

В предыдущем разделе уже было приведены некоторые примеры выполнения R-кода. В книге код выглядит примерно так.

```
1 + 2
#> [1] 3
```

Если вы выполните этот же код в окне консоли, то он будет выглядеть несколько иначе:

```
> 1 + 2 [1] 3
```

Здесь можно выделить два основных отличия. В окне консоли ваш ввод начинается после символа >, так называемого *приглашения* (мы не показываем его в книге). В книжном варианте вывод превращен в комментарии с помощью символов #>; в окне консоли вывод появляется непосредственно вслед за кодом. Эти два отличия означают, что если вы работаете с электронной версией книги, то можете свободно копировать код из книги и вставлять его на консоль.

На протяжении всей книги мы придерживаемся следующих соглашений при ссылках на код:

- для имен функций используется моноширинный шрифт, и за каждым таким именем следует пара скобок, например `sum()` или `mean()`;
- для других объектов R (таких, как данные или аргументы функций) также используется моноширинный шрифт, но их имена указываются без круглых скобок, например `flights` или `x`;
- если мы хотим уточнить, к какому пакету относится объект, то перед именем объекта указывается имя пакета, за которым следует двоеточие (например, `dplyr::mutate()` или `nycflights13::flights`); это тоже действительный R-код.

Получение справочной и дополнительной информации

Эта книга не является исчерпывающим пособием, да и не существует такой книги, прочитав которую вы смогли бы сказать, что полностью освоили R. По мере того как вы начнете применять описанные в данной книге методы в своих проектах, вы очень быстро столкнетесь с вопросами, на которые мы не даем ответа. Ниже вы найдете некоторые советы относительно того, где можно получить необходимые справочные сведения и как организовать свое дальнейшее обучение.

Зайдя в тупик, попытайтесь выбраться из него с помощью Google. Обычно добавления "R" в конце запроса достаточно для того, чтобы ограничить круг выводимых результатов поиска до нужного вам; если такой поиск окажется бесполезным, то это будет означать, что специфические для R результаты поиска отсутствуют. Если вы получаете сообщение об ошибке и не можете понять, что оно означает, попытайтесь "погуглить" его! Существует вероятность того, что кто-то другой ранее уже интересовался этим вопросом и соответствующая информация доступна где-то в Интернете. (Если сообщение об ошибке выведено не на английском языке, выполните команду `Sys.setenv(LANGUAGE = "en")` и перезапустите код; скорее всего, вам удастся получить справку об этом сообщении на английском языке.)

Если вам не поможет Google, попытайтесь найти помощь на сайте Stackoverflow (`http://stackoverflow.com/`). Для начала потратьте некоторое время на поиск ответа, который кто-то уже мог предложить; включение *[R]* в запрос позволяет сузить круг искомых вопросов и ответов до тех, которые имеют отношение к R. Если ничего полезного найти не удается, подготовьте воспроизводимый пример минимального объема. Увидев этот пример, другим людям будет легче оказать вам помощь, а нередко в процессе подготовки такого примера решение проблемы удается найти самому.

Чтобы сделать пример воспроизводимым, включите в него требуемые пакеты, данные и код.

- *Пакеты* должны загружаться в самом начале скрипта, чтобы можно было сразу же увидеть, какие из них используются в примере. Это наиболее подходящий момент убедиться в том, что вы используете самые последние версии каждого из пакетов. Вполне возможно, что вы столкнулись с ошибкой, которая была устранена с того момента, когда вы в последний раз устанавливали этот пакет на своем компьютере. В случае

пакетов библиотеки *tidyverse* самым простым способом проверки является выполнение функции `tidyverse_update()`.

- Самый простой способ включения в пример используемых данных — это вызов функции `dput()` для генерации кода, воспроизводящего данные. Например, чтобы воссоздать в R набор данных `mtcars`, поступите следующим образом:

1) вызовите в R функцию `dput(mtcars)`;
2) скопируйте вывод;
3) введите в своем воспроизводимом скрипте строку `mtcars <-`, а затем вставьте содержимое буфера обмена.

Попытайтесь найти минимальный набор данных, при котором проблема все еще проявляется.

- Не пожалейте времени на то, чтобы привести код к виду, в котором его будет легко читать другим людям:
  — обязательно используйте пробелы, облегчающие чтение текста, и короткие, но информативные имена переменных;
  — используйте комментарии для указания проблемных мест в коде;
  — исключите из примера все то, что не имеет отношения к возникшей проблеме.

  Чем короче код, тем он понятнее и тем легче его исправить.

Завершите этот процесс проверкой того, что вы действительно подготовили воспроизводимый пример, начав в R новый сеанс и скопировав в него свой скрипт.

Кроме того, постоянно готовьтесь к самостоятельному устранению проблем, прежде чем они возникнут. Ежедневное изучение R небольшими порциями окупится сторицей в долгосрочной перспективе. Следите за сообщениями Хэдли, Гарретта и других сотрудников компании RStudio, публикуемыми в блоге (`https://blog.rstudio.org/`). Именно там мы размещаем объявления о новых пакетах, новых возможностях IDE и персональных курсах. Возможно, вы захотите следить в Твиттере за сообщениями Хэдли (`@hadleywickham`) или Гарретта (`@statgarrett`) или быть в курсе всех новинок IDE (`@rstudiotips`).

Для поддержания контакта с более обширным R-сообществом рекомендуем посещать сайт `http://www.r-bloggers.com`: он агрегирует более 500

блогов со всего света по тематике R. Если вы активный пользователь Твиттера, следите за хеш-тегом #rstats. Твиттер — один из основных инструментов, используемых Хэдли для того, чтобы постоянно быть в курсе новых разработок, появляющихся в сообществе.

Электронная версия книги

Электронная версия книги на английском языке доступна на сайте http://r4ds.had.co.nz. Она будет непрерывно обновляться в промежутках между выпусками дополнительных тиражей печатного издания. Исходный код книги доступен по адресу https://github.com/hadley/r4ds. Книга поддерживается сайтом https://bookdown.org, который упрощает преобразование файлов в формате R Markdown в файлы формата HTML, PDF и EPUB.

Эта книга создавалась с использованием следующих ресурсов.

```
devtools::session_info(c("tidyverse"))
#> Session info ------------------------------------------------
#>  setting  value
#>  version  R version 3.3.1 (2016-06-21)
#>  system   x86_64, darwin13.4.0
#>  ui       X11
#>  language (EN)
#>  collate  en_US.UTF-8
#>  tz       America/Los_Angeles
#>  date     2016-10-10
#> Packages ----------------------------------------------------
#>  package      * version    date       source
#>  assertthat     0.1        2013-12-06 CRAN (R 3.3.0)
#>  BH             1.60.0-2   2016-05-07 CRAN (R 3.3.0)
#>  broom          0.4.1      2016-06-24 CRAN (R 3.3.0)
#>  colorspace     1.2-6      2015-03-11 CRAN (R 3.3.0)
#>  curl           2.1        2016-09-22 CRAN (R 3.3.0)
#>  DBI            0.5-1      2016-09-10 CRAN (R 3.3.0)
#>  dichromat      2.0-0      2013-01-24 CRAN (R 3.3.0)
#>  digest         0.6.10     2016-08-02 CRAN (R 3.3.0)
#>  dplyr        * 0.5.0      2016-06-24 CRAN (R 3.3.0)
#>  forcats        0.1.1      2016-09-16 CRAN (R 3.3.0)
#>  foreign        0.8-67     2016-09-13 CRAN (R 3.3.0)
#>  ggplot2      * 2.1.0.9001 2016-10-06 local
#>  gtable         0.2.0      2016-02-26 CRAN (R 3.3.0)
#>  haven          1.0.0      2016-09-30 local
#>  hms            0.2-1      2016-07-28 CRAN (R 3.3.1)
#>  httr           1.2.1      2016-07-03 cran (@1.2.1)
#>  jsonlite       1.1        2016-09-14 CRAN (R 3.3.0)
```

```
#>  labeling       0.3        2014-08-23 CRAN (R 3.3.0)
#>  lattice        0.20-34    2016-09-06 CRAN (R 3.3.0)
#>  lazyeval       0.2.0      2016-06-12 CRAN (R 3.3.0)
#>  lubridate      1.6.0      2016-09-13 CRAN (R 3.3.0)
#>  magrittr       1.5        2014-11-22 CRAN (R 3.3.0)
#>  MASS           7.3-45     2016-04-21 CRAN (R 3.3.1)
#>  mime           0.5        2016-07-07 cran (@0.5)
#>  mnormt         1.5-4      2016-03-09 CRAN (R 3.3.0)
#>  modelr         0.1.0      2016-08-31 CRAN (R 3.3.0)
#>  munsell        0.4.3      2016-02-13 CRAN (R 3.3.0)
#>  nlme           3.1-128    2016-05-10 CRAN (R 3.3.1)
#>  openssl        0.9.4      2016-05-25 cran (@0.9.4)
#>  plyr           1.8.4      2016-06-08 cran (@1.8.4)
#>  psych          1.6.9      2016-09-17 CRAN (R 3.3.0)
#>  purrr        * 0.2.2      2016-06-18 CRAN (R 3.3.0)
#>  R6             2.1.3      2016-08-19 CRAN (R 3.3.0)
#>  RColorBrewer   1.1-2      2014-12-07 CRAN (R 3.3.0)
#>  Rcpp           0.12.7     2016-09-05 CRAN (R 3.3.0)
#>  readr        * 1.0.0      2016-08-03 CRAN (R 3.3.0)
#>  readxl         0.1.1      2016-03-28 CRAN (R 3.3.0)
#>  reshape2       1.4.1      2014-12-06 CRAN (R 3.3.0)
#>  rvest          0.3.2      2016-06-17 CRAN (R 3.3.0)
#>  scales         0.4.0.9003 2016-10-06 local
#>  selectr        0.3-0      2016-08-30 CRAN (R 3.3.0)
#>  stringi        1.1.2      2016-10-01 CRAN (R 3.3.1)
#>  stringr        1.1.0      2016-08-19 cran (@1.1.0)
#>  tibble       * 1.2        2016-08-26 CRAN (R 3.3.0)
#>  tidyr        * 0.6.0      2016-08-12 CRAN (R 3.3.0)
#>  tidyverse    * 1.0.0      2016-09-09 CRAN (R 3.3.0)
#>  xml2           1.0.0.9001 2016-09-30 local
```

Соглашения, принятые в книге

В книге приняты следующие типографские соглашения.

- *Курсивом* выделяются новые термины, имена и расширения имен файлов, названия библиотек.
- **Полужирным** шрифтом выделяются названия пакетов R.
- `Моноширинный` шрифт используется в листингах программ, а также в основном тексте для представления таких программных элементов, как названия функций и переменных, баз данных, типов данных, инструкций, ключевых слов, а также URL-адресов, адресов электронной почты и других подобных элементов.

- **Полужирный моноширинный** шрифт используется для представления команд и другого текста, который должен вводиться пользователем.
- *Курсивный моноширинный* шрифт используется для представления текста, который должен быть заменен вводимыми пользователем значениями, определяемыми контекстом.

Этой пиктограммой обозначены абзацы, содержащие советы и рекомендации.

Использование примеров кода

Исходный код всех примеров доступен по адресу `https://github.com/hadley/r4ds`.

Книга была написана для того, чтобы облегчить вам работу. Вообще говоря, вы можете свободно использовать приведенный в книге код в своих программах и документации. Получения какого-либо специального разрешения от нас, если только речь не идет о значительных объемах кода, не требуется. Например, использование в вашей программе нескольких фрагментов кода, взятых из этой книги, не требует разрешения. Однако продажа или распространение компакт-диска, содержащего примеры из книг, выпущенных издательством O'Reilly, без предварительного получения разрешения запрещена. Цитирование данной книги и использование примеров кода в ответах на вопросы не требует разрешения. Вместе с тем, если вы включаете в документацию своего продукта значительные объемы кода из приведенных в книге примеров, то получение соответствующего разрешения является обязательным условием.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

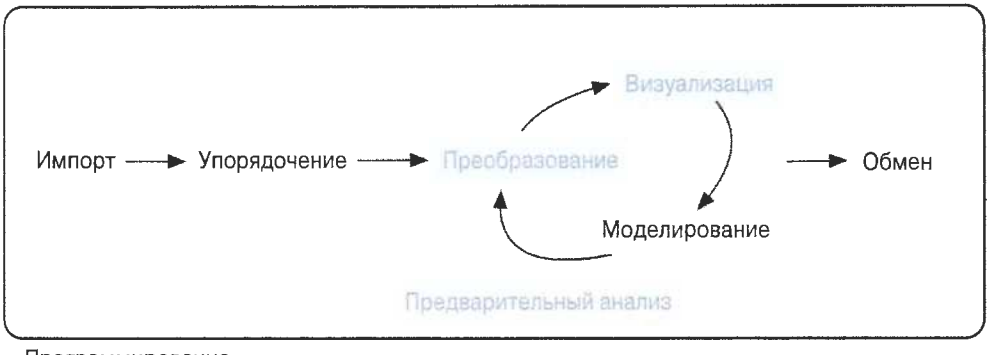
E-mail: info@dialektika.com

WWW: http://www.dialektika.com

ЧАСТЬ I

Предварительный анализ

Часть I поможет вам в сжатые сроки освоить базовые инструменты предварительного (разведочного) анализа данных. В ходе такого анализа выявляются наиболее общие закономерности, свойственные анализируемым данным, на основании которых выдвигают и проверяют различные рабочие гипотезы, при необходимости многократно повторяя описанный цикл действий. Главной задачей этого этапа является выработка нескольких перспективных предположений, которые впоследствии можно будет подвергнуть более глубокому анализу.



В этой части вы познакомитесь с некоторыми полезными инструментами, отдачу от использования которых почувствуете незамедлительно.

- Визуализация — отличная отправная точка для того, чтобы начать программировать на языке R, поскольку преимущества очевидны: вы получаете возможность строить аккуратные информативные графики, облегчающие понимание данных. В главе 1 вы узнаете о возможностях пакета **ggplot2** и изучите мощные методы, позволяющие превращать данные в графики.
- В типичных случаях одной лишь визуализации данных еще недостаточно, поэтому в главе 3 вы изучите основные команды, позволяющие

выбирать важные переменные, фильтровать ключевые наблюдения, создавать новые переменные и вычислять итоги.

- Наконец, в главе 5 вы научитесь формулировать вопросы в отношении интересующих вас данных и получать на них ответы.

Важной частью процесса предварительного анализа является моделирование, но на данном этапе для его эффективного изучения или применения вам недостает знаний. К этой теме мы вернемся в части IV, когда вы уже будете хорошо владеть инструментами, предназначенными для подготовки данных, и приобретете необходимые навыки программирования.

Вышеперечисленные три главы, посвященные инструментам предварительного анализа данных, перемежаются тремя другими главами, в которых внимание фокусируется на технических аспектах работы в среде R. Главы 2, 4 и 6 познакомят вас с установившейся практикой написания и организации кода на языке R. Это послужит залогом успешного функционирования ваших программ в долгосрочной перспективе, поскольку вы овладеете инструментами, позволяющими эффективно управлять реальными проектами на протяжении всего времени их жизни.

ГЛАВА 1

Визуализация данных с помощью ggplot2

Введение

"Обычный график дает аналитику гораздо больше пищи для размышлений, чем любая другая форма представления информации".

Джон Тьюки

В этой главе вы научитесь визуализировать данные с помощью расширения **ggplot2**. Среда R предлагает несколько систем для работы с графиками, но одной из наиболее элегантных и универсальных среди них является **ggplot2**. Расширение **ggplot2** реализует грамматику графики — стройную систему, обеспечивающую описание и построение графиков. Вместо того чтобы изучать все системы, новичку целесообразно сэкономить время и изучить одно лишь расширение **ggplot2**, которого будет вполне достаточно для подавляющего круга задач.

Тем читателям, которые хотели бы узнать больше о теоретической подоплеке расширения **ggplot2**, рекомендуем прочитать статью "A Layered Grammar of Graphics" (`http://vita.had.co.nz/papers/layered-grammar.pdf`).

Используемые ресурсы

В этой главе рассматривается пакет **ggplot2** — один из центральных элементов библиотеки *tidyverse*. Чтобы получить доступ к наборам данных, справочным материалам и функциям, которые будут использоваться в этой главе, загрузите *tidyverse* с помощью следующей команды.

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: tidyr
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages ----------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

Эта единственная строка кода загружает ядро библиотеки *tidyverse* — пакеты, которые будут использоваться практически в любом анализе данных. Попутно она информирует вас о наличии конфликтов между функциями *tidyverse* и функциями базовой среды R (или функциями, входящими в состав других уже загруженных вами пакетов).

Если после выполнения этой команды вы получите сообщение об ошибке "there is no package called 'tidyverse'", то вам потребуется сначала установить библиотеку, а затем вновь выполнить команду `library()`.

```
install.packages("tidyverse")
library(tidyverse)
```

Требуется лишь однократная установка пакета, но его необходимо заново загружать при каждом запуске нового рабочего сеанса.

В тех случаях, когда требуется явно указать принадлежность функции (или набора данных) определенному пакету, применяется специальная синтаксическая конструкция `пакет::функция()`. Например, выражение `ggplot2::ggplot()` явно указывает на то, что мы используем функцию `ggplot()` из пакета **ggplot2**.

Первые шаги

Воспользуемся нашим первым графиком, чтобы получить ответ на следующий вопрос: действительно ли автомобили с большим объемом двигателя потребляют больше топлива, чем автомобили, объем двигателей которых меньше? Вероятно, ответ вам уже известен, но мы попытаемся обосновать его. Что мы можем сказать о характере зависимости расхода топлива от объема двигателя? Что это за связь — положительная, отрицательная, линейная, нелинейная?

Фрейм данных mpg

Для проверки своего ответа можно воспользоваться фреймом данных mpg из пакета **ggplot2** (полное название ggplot2::mpg). *Фрейм (кадр) данных* — это прямоугольная таблица, столбцами которой служат переменные, а строками — наблюдения (значения соответствующих переменных). Фрейм mpg включает результаты наблюдений, подготовленных Управлением по защите окружающей среды США для 38 моделей автомобилей.

```
mpg
#> # A tibble: 234 × 11
#>   manufacturer model displ   year    cyl        trans   drv
#>          <chr> <chr> <dbl>  <int>  <int>        <chr> <chr>
#> 1         audi    a4   1.8   1999      4     auto(l5)     f
#> 2         audi    a4   1.8   1999      4   manual(m5)     f
#> 3         audi    a4   2.0   2008      4   manual(m6)     f
#> 4         audi    a4   2.0   2008      4     auto(av)     f
#> 5         audi    a4   2.8   1999      6     auto(l5)     f
#> 6         audi    a4   2.8   1999      6   manual(m5)     f
#> # ... with 228 more rows, and 4 more variables:
#> # cty <int>, hwy <int>, fl <chr>, class <chr>
```

В mpg среди прочих других фигурируют следующие переменные:

- displ — объем двигателя автомобиля, выражаемый в литрах;
- hwy — показатель эффективности расхода топлива, выражаемый в милях пробега на галлон топлива (mpg). При прохождении одного и того же расстояния автомобиль с более низким значением показателя эффективности потребляет больше топлива, чем автомобиль с более высоким значением этого показателя.

Чтобы получить более подробные сведения о фрейме данных mpg, откройте соответствующий раздел справки, выполнив команду ?mpg.

Создание графика

Чтобы построить график mpg, в котором значения переменной displ располагаются вдоль оси *x*, а значения переменной hwy — вдоль оси *y* (рис. 1.1), выполните следующую команду.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))
```

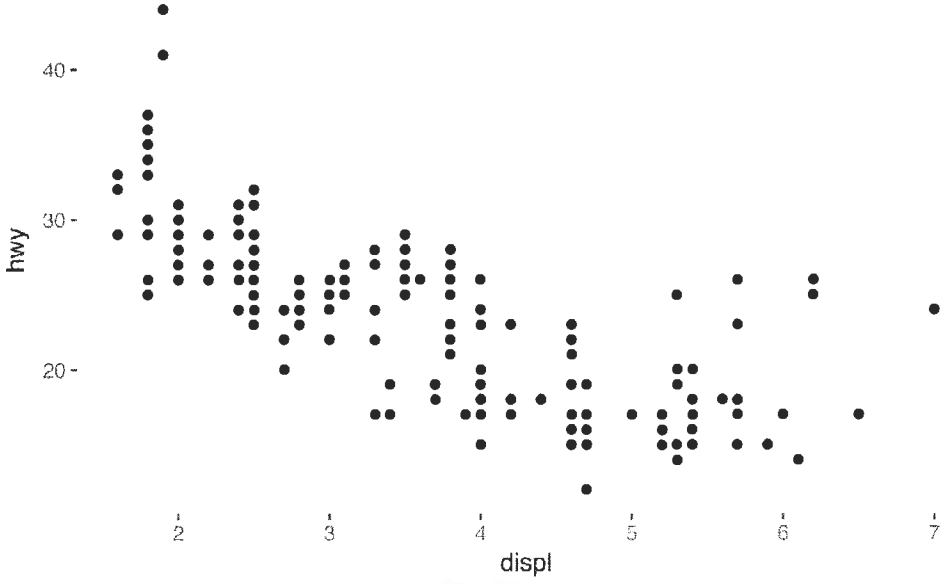


Рис. 1.1

График отчетливо демонстрирует отрицательный характер зависимости показателя эффективности расхода топлива (`hwy`) от объема двигателя (`displ`). Другими словами, автомобили с большими двигателями расходуют больше топлива. Подтверждает это или опровергает вашу гипотезу о соотношении между показателем эффективности расхода топлива автомобилем и объемом его двигателя?

При работе с пакетом **ggplot2** создание графика начинается с вызова функции `ggplot()`. Этот вызов создает систему координат, к которой можно добавлять слои. При этом функции передается значение аргумента `data`, представляющего набор данных, который далее будет использован для построения графика. Таким образом, строка кода `ggplot(data = mpg)` лишь создает пустой график, который не представляет для нас никакого интереса и поэтому не показан в виде отдельного рисунка.

Построение графика завершается добавлением в `ggplot()` одного или нескольких слоев. Функция `geom_point()` добавляет в график слой точек, создающий точечную диаграмму. В состав пакета **ggplot2** включено большое количество различных функций `geom_xxx()`, каждая из которых предназначена для добавления в график слоя определенного типа. Вы будете постепенно знакомиться с ними в процессе чтения главы.

Каждая функция `geom_xxx()` из пакета **ggplot2** принимает аргумент `mapping`, задающий сопоставление переменных из набора данных с определенными визуальными средствами. Значение этого аргумента всегда определяется посредством вызова функции `aes()`, аргументы `x` и `y` которой указывают, значения каких переменных располагаются вдоль осей *x* и *y*. Пакет **ggplot2** ищет сопоставляемую переменную в наборе данных, задаваемом аргументом `data` (в данном случае — в наборе данных `mpg`).

Шаблон команды создания графика

Запишем приведенный выше код в виде повторно используемого шаблона команды для создания графиков с помощью расширения **ggplot2**. Чтобы создать график, замените элементы, выделенные в следующем коде угловыми скобками, конкретными именами набора данных, функции `geom_xxx()` и коллекции сопоставлений.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>))
```

В оставшейся части главы будут продемонстрированы различные варианты заполнения и расширения этого шаблона для создания графиков различного типа. Начнем с компонента `<MAPPINGS>`.

Упражнения

1. Выполните команду `ggplot(data = mpg)`. Что вы видите?
2. Сколько строк имеет таблица `mpg`? Сколько столбцов?
3. Что описывает переменная `drv`? Выясните это, выведя справочную информацию командой `?mpg`.
4. Создайте точечную диаграмму, описывающую зависимость переменной `hwy` от переменной `cyl`.
5. Что произойдет, если попытаться создать график зависимости переменной `class` от переменной `drv`? Почему невозможно использовать этот график?

Эстетика визуализации

"Величайшее достоинство графики в том, что она заставляет нас замечать то, чего мы никогда не ожидали увидеть".
Джон Тьюки

На приведенном ниже графике (рис. 1.2) одна группа точек (выделенная красным цветом) явно выпадает из линейного тренда. Эти автомобили демонстрируют более высокую эффективность расхода топлива по сравнению с тем, чего можно было бы ожидать. Чем это можно объяснить?

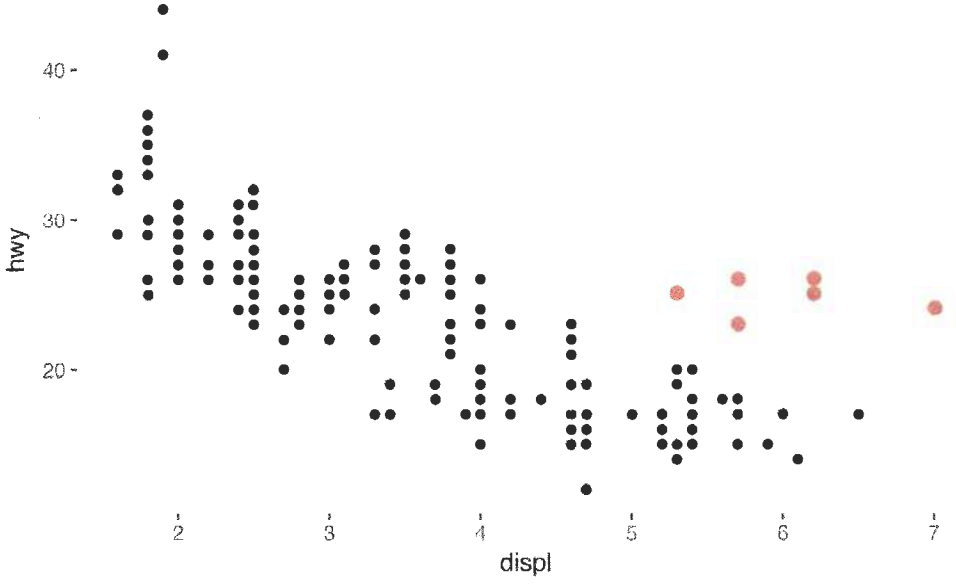


Рис. 1.2

Можно было бы предположить, что эти автомобили относятся к гибридному типу. Один из способов проверить данную гипотезу — проанализировать значения класса каждого автомобиля. Переменная `class` набора данных `mpg` разбивает автомобили на группы, такие как *compact*, *midsize* или *SUV*[1]. Если выпадающие из общего ряда точки соответствуют гибридным типам, то, судя по расходу топлива, их следует классифицировать как компактные или, возможно, субкомпактные автомобили (имейте в виду, что в тот период

[1] В примере используется принятая в США система классификации автомобилей, основанная на длине колесной базы и объеме полезного пространства салона. В этой системе классу SUV (Sport-Utility Vehicles) соответствуют внедорожники. — *Примеч. ред.*

времени, к которому относятся эти данные, гибридные автомобили и внедорожники еще не пользовались такой популярностью, как в наши дни).

Мы можем добавить в двухмерную точечную диаграмму третью переменную, `class`, связав ее с каким-либо *визуальным атрибутом*[2]. Визуальный атрибут — это свойство объектов графика, определяющее их внешний вид. К визуальным атрибутам относятся такие, например, свойства, как размер, форма и цвет точек данных. Вы можете управлять внешним видом точек данных, изменяя значения их атрибутов визуализации. Поскольку слово "значение" уже используется нами для описания данных, то применительно к атрибутам визуализации мы будем использовать слово "уровень". Ниже приведен пример того, как изменяется внешний вид точки данных при изменении ее размера, формы и цвета (рис. 1.3).

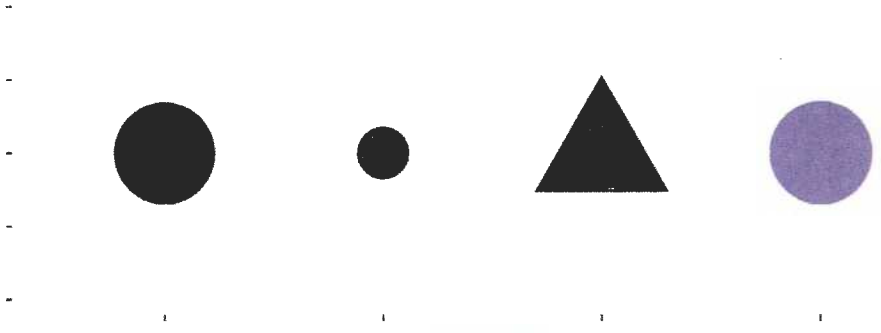


Рис. 1.3

Можно сообщать информацию о данных, связывая атрибуты визуализации графика с переменными набора данных. Например, установив соответствие между переменной `class` и цветом точек данных, можно сообщить информацию о классе каждого автомобиля (рис. 1.4).

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, color = class))
```

Соответствие визуальных атрибутов переменным устанавливается путем связывания имени атрибута с именем переменной при вызове функции `aes()`. Расширение **ggplot2** автоматически присвоит уникальный уровень этого атрибута (в данном случае `color`) каждому уникальному значению переменной — этот процесс называется *градацией* (scaling). Кроме того, **ggplot2**

[2] В оригинале используется термин *aesthetic* (эстетическое свойство), введенный одним из авторов этой книги и разработчиком пакета **ggplot2** Хэдли Уикемом. — *Примеч. ред.*

автоматически добавит легенду, описывающую соответствие между уровнями и значениями.

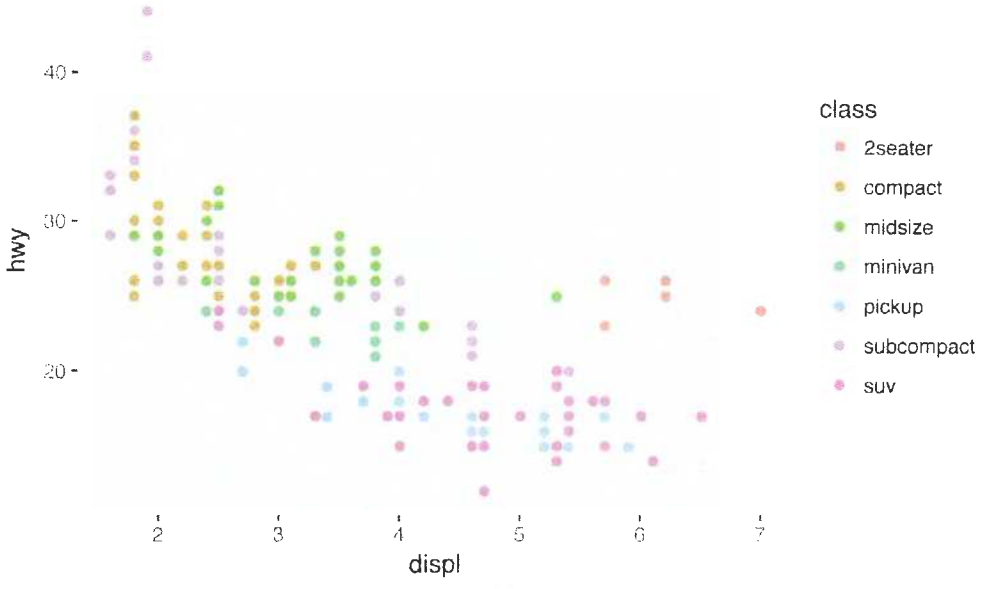


Рис. 1.4

Цветовое выделение показывает, что многие из необычных точек соответствуют 2-местным автомобилям. Вряд ли ими могут быть гибриды. Скорее всего, это спортивные автомобили! Объем двигателей спортивных автомобилей сравним с объемом двигателя внедорожников и пикапов, но благодаря небольшому кузову, сравнимому с кузовами среднеразмерных и компактных моделей автомобилей, они характеризуются высокими показателями эффективности расхода топлива. Возвращаясь к высказанной ранее первоначальной гипотезе, можно сказать, что с большой долей вероятности эти автомобили не могут быть гибридами ввиду больших объемов их двигателей.

В предыдущем примере переменная `class` связывалась с визуальным атрибутом цвета, но с таким же успехом мы могли бы связать ее с атрибутом размера. В данном случае на принадлежность точки к тому или иному классу будет указывать ее размер. Однако при этом будет выдано *предупреждающее сообщение*, поскольку сопоставление неупорядоченной переменной (`class`) с упорядоченным визуальным атрибутом (`size`) — не очень удачная идея (рис. 1.5).

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, size = class))
#> Warning: Using size for a discrete variable is not advised.
```

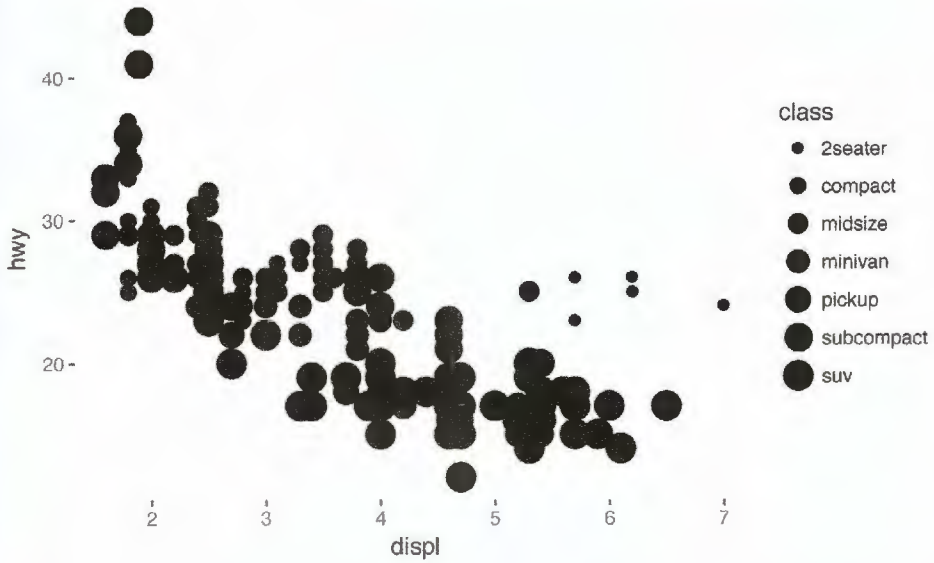


Рис. 1.5

Мы также могли бы сопоставить классу атрибут `alpha`, управляющий прозрачностью точек, или атрибут `shape`, управляющий формой точек (рис. 1.6 и 1.7).

```
# Рис. 1.6
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, alpha = class))

# Рис. 1.7
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy, shape = class))
```

А что случилось с внедорожниками (SUV)? Расширение **ggplot2** использует одновременно лишь шесть различных фигур. При использовании атрибута формы дополнительные группы по умолчанию не отображаются на графике.

Функция `aes()` связывает имя каждого из используемых атрибутов с отображаемой переменной. Она собирает все сопоставления переменных с визуальными атрибутами, используемые слоем, и передает их аргументу `mapping` слоя. Данный синтаксис подчеркивает полезную особенность параметров `x` и `y`: переменные `x` и `y`, описывающие положение точки, сами являются

визуальными атрибутами — свойствами, которые можно сопоставлять переменным для отображения информации, касающейся данных.

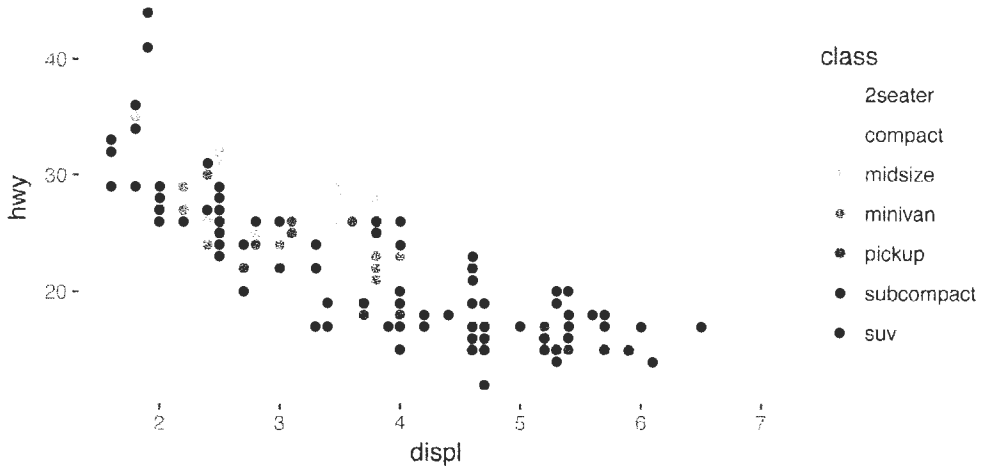


Рис. 1.6

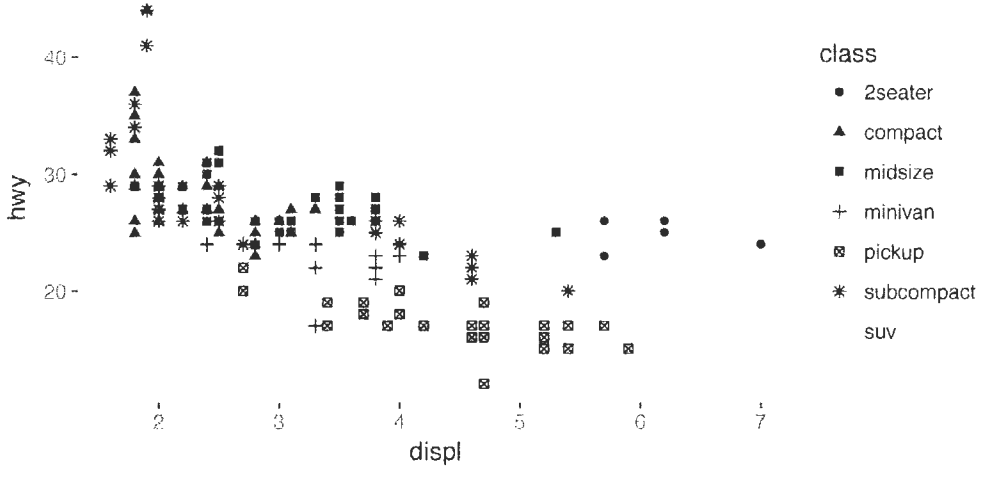


Рис. 1.7

Как только вы задали соответствия между переменными и визуальными атрибутами, **ggplot2** самостоятельно позаботится обо всем остальном: выберет подходящую шкалу для визуального атрибута и создаст легенду, описывающую взаимное соответствие уровней и значений. Для атрибутов x и y вместо элементов легенды создаются линии осей с метками и подписями. Линия оси выполняет функции легенды; она описывает соответствие между положениями точек и значениями.

Вы также можете *установить* визуальные атрибуты вручную. Например, можно окрасить все точки графика в синий цвет (рис. 1.8).

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy), color = "blue")
```

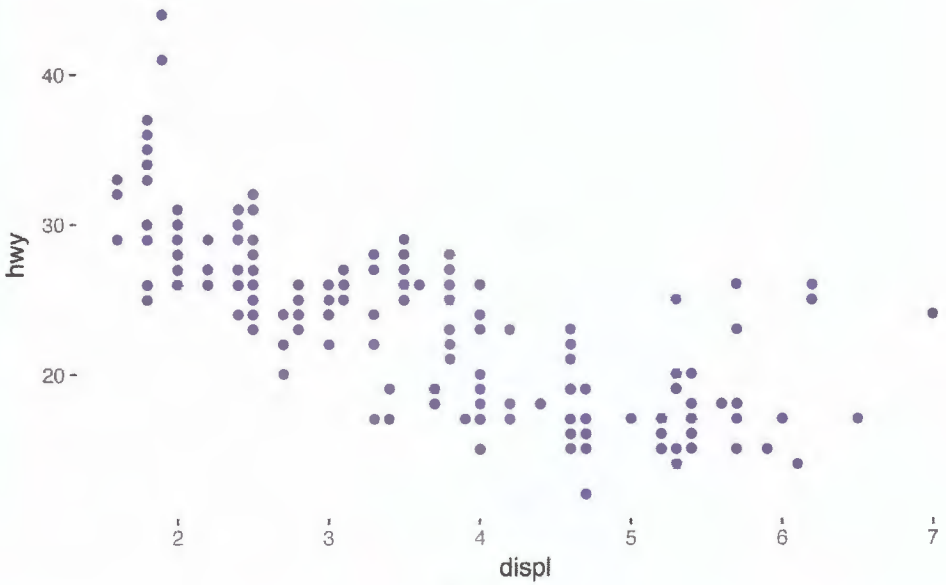


Рис. 1.8

В данном случае атрибут цвета используется лишь для изменения внешнего вида графика, а не для передачи определенной информации о переменной. Чтобы задать визуальный атрибут вручную, следует включить его в список аргументов функции `geom_xxx()`, т.е. вынести его за пределы функции `aes()`. Выбираемые вами значения должны соответствовать типу атрибута:

- название цвета задается в виде символьной строки;
- размер точки задается в миллиметрах;
- форма точки задается в виде числа (рис. 1.9). При этом наблюдается кажущееся дублирование значений: например, каждому из значений 0, 15 и 22 соответствует квадрат. Различия вносятся сочетанием атрибутов цвета и заливки. Цвет обводки полых фигур (0–14) определяется атрибутом `color`; цвет сплошных фигур (15–18) — атрибутом `color`; для фигур с обводкой (21–24) цвет границы определяется атрибутом `color`, а цвет заливки — атрибутом `fill`.

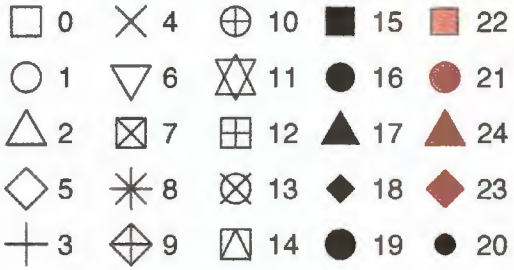


Рис. 1.9

Упражнения

1. Что не так с приведенным ниже кодом? Почему точки не окрашиваются в синий цвет (рис. 1.10)?

```
ggplot(data = mpg) +
  geom_point(
    mapping = aes(x = displ, y = hwy, color = "blue")
  )
```

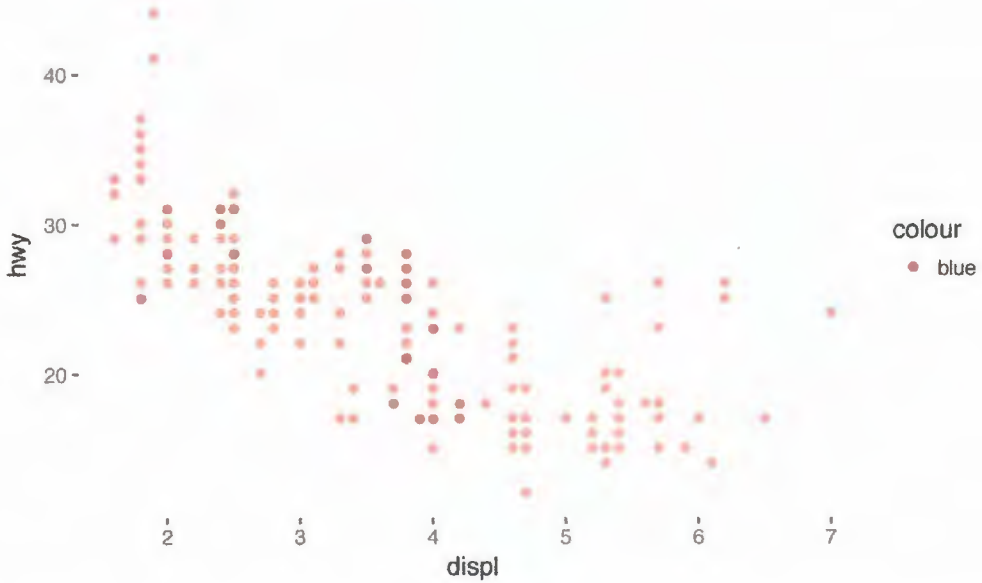


Рис. 1.10

2. Какие переменные в наборе данных `mpg` являются категориальными? Какие переменные являются непрерывными? (Подсказка: команда `?mpg`

открывает доступ к документации по данному набору.) Как можно увидеть эту информацию, если выполнить команду `mpg`?

3. Сопоставьте непрерывной переменной цвет, размер и форму точек. В чем заключаются различия в поведении этих атрибутов по отношению к категориальным и непрерывным переменным?
4. Что произойдет, если вы свяжете одну и ту же переменную с несколькими визуальными атрибутами?
5. Каково назначение визуального атрибута `stroke`? С какими фигурами он работает? (Подсказка: воспользуйтесь командой `?geom_point`.)
6. Что произойдет, если вы свяжете визуальный атрибут не с именем переменной, а с чем-то другим, например `aes(color=displ < 5)`?

Распространенные ошибки

Всегда существует вероятность того, что при запуске своего R-кода вы столкнетесь с проблемами. Не огорчайтесь — это случается с каждым. Я программирую на языке R уже многие коды, но это не мешает мне ежедневно сталкиваться с тем, что написанный только что код не работает!

Начните с тщательного сравнения запускаемого вами кода с тем, который приведен в книге. Язык R чрезвычайно придирчив, и виновником проблемы может оказаться единственный символ, стоящий не на своем месте. Убедитесь в том, что все фрагменты кода записаны правильно и взаимосвязаны. Иногда бывает так, что при запуске кода ничего не происходит. Проверьте начало каждой строки содержимого консоли: если оно начинается с символа "+", то это означает, что R не считает введенное вами выражение полным и ожидает завершения ввода. Как правило, в таких случаях проще повторить ввод с самого начала, нажав клавишу <Esc> для сброса текущей команды.

Одна из распространенных ошибок при создании графиков с помощью **ggplot2** — расположение оператора "+" в неподходящем месте: он должен находиться в конце строки, а не в начале. Иными словами, убедитесь в том, что код не был случайно записан примерно в следующем виде.

```
ggplot(data = mpg)
+ geom_point(mapping =  es(x = displ, y = hwy))
```

Если проблему не удается разрешить, обратитесь к справочной системе. Для получения справки относительно любой функции R достаточно

выполнить на консоли команду `?имя_функции` или выделить в RStudio имя функции и нажать клавишу <F1>. Не расстраивайтесь, если справка показалась вам не очень полезной; просмотрите примеры и постарайтесь подыскать код, реализующий то, что вы хотите сделать.

Если и это не помогло, внимательно прочитайте сообщение об ошибке. Иногда там может быть скрыт ответ! Но если вы новичок в R, то даже когда ответ содержится в сообщении об ошибке, вы можете этого не понять. "Палочкой-выручалочкой" для вас может оказаться Google: попробуйте "погуглить" текст сообщения об ошибке, поскольку не исключено, что кто-то уже сталкивался с аналогичной проблемой и ему удалось получить помощь в Интернете.

Панели

Визуальные атрибуты — один из возможных способов введения дополнительных переменных. Другой способ, особенно полезный для категориальных переменных, заключается в разбиении графика на отдельные *панели* (facets), каждая из которых представляет собой срез полного графика, отображающий отдельное подмножество данных (рис. 1.11).

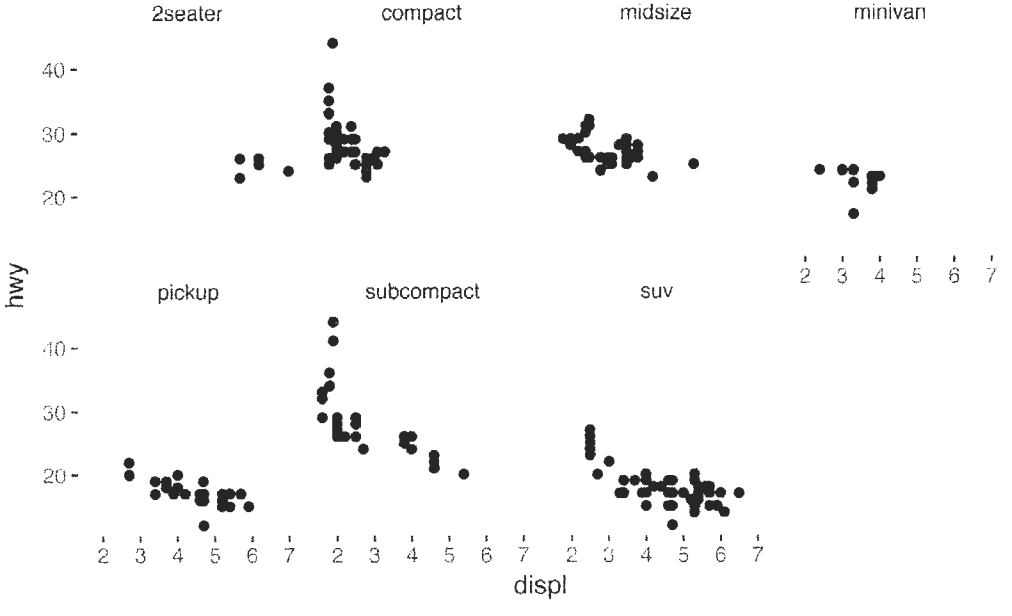


Рис. 1.11

Для получения среза графика по одной переменной используется функция `facet_wrap()`. В качестве первого аргумента этой функции должна задаваться формула, начинающаяся с символа тильды ("~"), за которым следует имя переменной (в этом контексте слово "формула" не является синонимом слова "уравнение" и означает имя структуры данных R). Передаваемая функции `facet_wrap()` переменная должна быть дискретной.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

Для получения среза графика по комбинации двух переменных следует включить в команду построения графика функцию `facet_grid()`. Первым аргументом этой функции также является формула. На этот раз формула должна содержать имена двух переменных, разделенные символом "~" (рис. 1.12).

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ cyl)
```

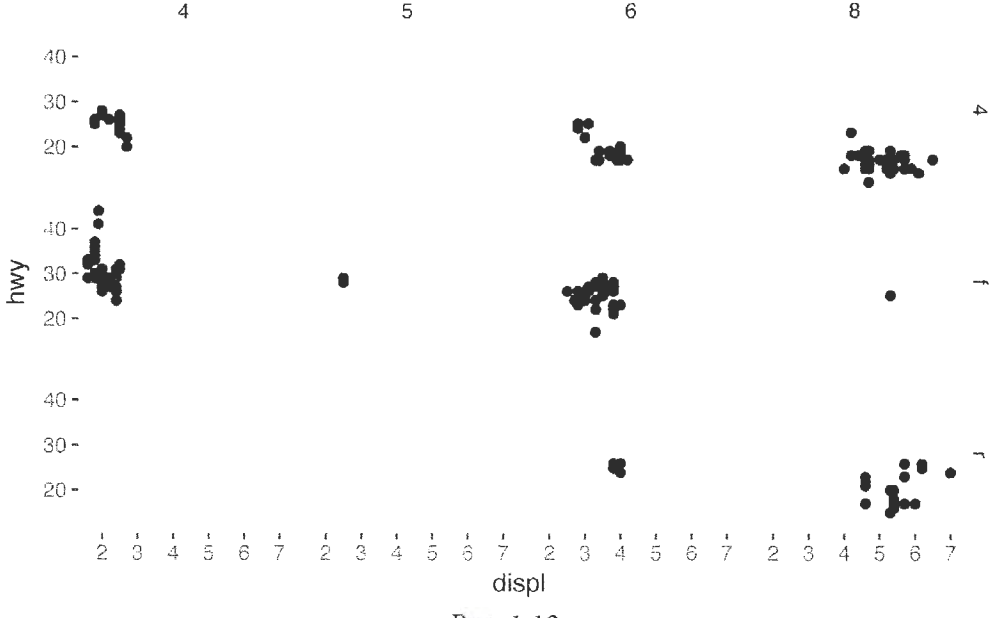


Рис. 1.12

Если вы хотите отказаться от выделения срезов по строкам или столбцам, используйте символ точки “.” вместо имени соответствующей переменной, например `+ facet_grid(. ~ cyl)`.

Упражнения

1. Что произойдет, если вы попытаетесь получить срез графика по непрерывной переменной?
2. Что означают пустые ячейки на графике, созданном с помощью команды `facet_grid(drv ~ cyl)`? Как они соотносятся с графиком, созданном с помощью приведенной ниже команды?

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = drv, y = cyl))
```

3. Какие графики создает приведенный ниже код? Какую функцию выполняет в нем символ точки “.”?

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(drv ~ .)

ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_grid(. ~ cyl)
```

4. Обратимся к первому из приведенных в этом разделе примеров создания срезов графика на отдельных панелях.

```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

   В чем заключаются преимущества использования панелей (срезов) по сравнению с визуальными атрибутами? В чем заключаются недостатки этого способа? Какого характера изменений в соотношении преимуществ и недостатков можно ожидать при увеличении размера набора данных?
5. Прочитайте текст, выводимый по команде `?facet_wrap`. Каковы функции параметра `nrow`? Для чего предназначен параметр `ncol`? Какие еще параметры позволяют управлять компоновкой отдельных панелей? Почему функция `facet_grid()` не имеет параметров `nrow` и `ncol`?
6. Как правило, когда используется функция `facet_grid()`, в столбцы следует помещать переменную с более уникальными уровнями. Почему?

Геометрические объекты

Насколько близки между собой следующие два графика (рис. 1.13)?

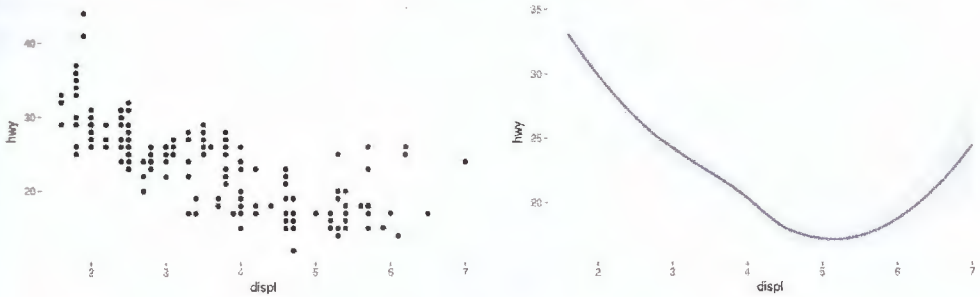


Рис. 1.13

Переменные `x` и `y` в них одни и те же, и оба графика отображают одни и те же данные. И тем не менее разница очевидна. Дело в том, что для представления данных в этих графиках используются, следуя терминологии **ggplot2**, разные *геометрии*[3].

Геометрия графика определяется *геометрическим объектом*, используемым для представления данных. Графики (диаграммы) часто описываются типом используемого геометрического объекта. Например, в столбчатых диаграммах используется геометрия `bar` (полоса), в графиках "ящик с усами" (ящичковая диаграмма) — геометрия `box` (ящик) и т.п. Диаграммы рассеяния выпадают из этого ряда — в них используется геометрия `point` (точка). Как следует из предыдущего примера, графики одних и тех же данных могут создаваться с использованием различных геометрических объектов. На графике слева используются объекты `point`, на графике справа — объекты `smooth`, которым соответствует гладкая кривая, подогнанная под данные.

Чтобы изменить геометрию графика, измените функцию `geom_xxx()`, которую вы добавляете к функции `ggplot()`. Например, для создания предыдущих графиков можно использовать следующий код.

```
# слева
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

# справа
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))
```

[3] См. статью Хэдли Уикема "A Layered Grammar of Graphics" (`https://www.uvm.edu/~ngotelli/Rscripts/wickham.2009.pdf`). — *Примеч. ред.*

Каждая из функций `geom_xxx()` в **ggplot2** принимает аргумент `mapping`. Однако не все визуальные атрибуты подходят для любой геометрии. Можно задать форму точки (маркера данных), но нельзя задать "форму" линии. С другой стороны, можно задать тип линии. Функция `geom_smooth()` вычерчивает линии разного типа для каждого уникального значения переменной, сопоставляемой с типом линии (рис. 1.14).

```
gp    (data = mpg) +
  om_  pot  (mapping = aes(x = displ, y = hwy, linetype = drv))
```

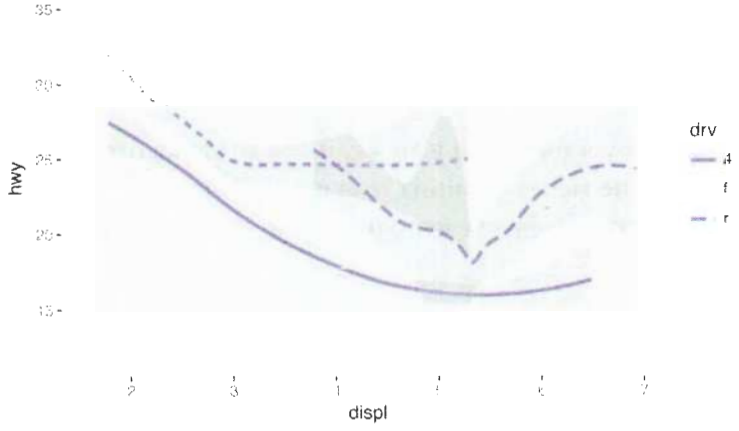


Рис. 1.14

Здесь функция `geom_smooth()` распределяет автомобили по трем линиям на основе значений переменной `drv`, описывающей тип привода автомобиля. Одна линия соответствует всем точкам, для которых это значение равно *4*, другая — точкам, для которых это значение равно *f*, а третья — точкам, для которых это значение равно *r*. Здесь значение *4* соответствует полноприводным, *f* — переднеприводным, а *r* — заднеприводным автомобилям.

Вышесказанное становится еще более понятным, если наложить линии графиков на исходные точки данных и использовать цвет для выделения различных значений `drv` (рис. 1.15).

Обратите внимание на то, что эта диаграмма совмещает графики, в которых используются две различные геометрии! Если вы горите желанием узнать, как это делается, немного потерпите. В следующем разделе будет показано, как совместить на одной и той же диаграмме графики, основанные на различных геометриях.

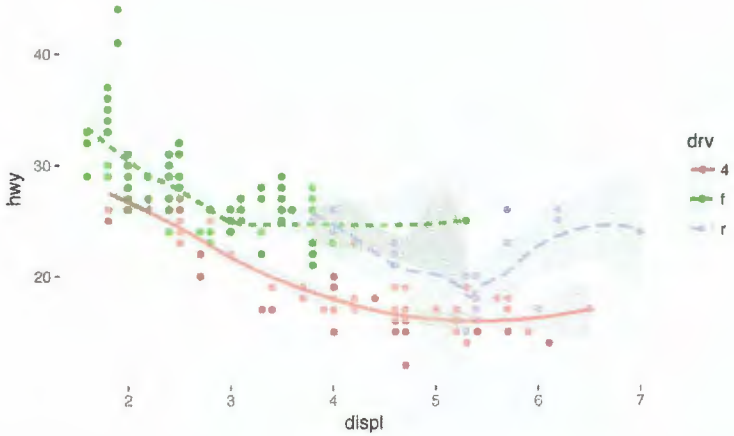


Рис. 1.15

Пакет **ggplot2** предоставляет свыше 30 объектов `geom`, и еще большее их количество доступно в пакетах расширений (список этих расширений можно найти на сайте `https://www.ggplot2-exts.org`). Для получения наиболее полных сведений по этому вопросу лучше всего обратиться к "шпаргалке" по пакету **ggplot2**, которую можно найти по адресу `http://rstudio.com/cheatsheets`. Более подробные сведения о любом объекте `geom` можно получить в справочной системе, используя команды наподобие следующей: `?geom_smooth`.

Многие функции, такие как `geom_smooth()`, отображают несколько рядов данных с использованием одного геометрического объекта. В подобных случаях можно назначать категориальной переменной один групповой визуальный атрибут для вычерчивания нескольких объектов.

Пакет **ggplot2** будет вычерчивать отдельный объект для каждого уникального значения групповой переменной. На практике **ggplot2** автоматически группирует данные для таких геометрий всякий раз, когда вы связываете визуальный атрибут с дискретной переменной (как в примере с различными типами линий). Это очень удобно, поскольку групповые визуальные атрибуты не создают дополнительные элементы легенды и не предоставляют никаких других средств визуального различения геометрических объектов (рис. 1.16).

```
ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy))

ggplot(data = mpg) +
  geom_smooth(mapping = aes(x = displ, y = hwy, group = drv))
```

```
ggplot(data = mpg) +
  geom_smo    (
    mapping = ae (x = displ, y = hwy, color = drv),
    show.legend = FALSE
  )
```

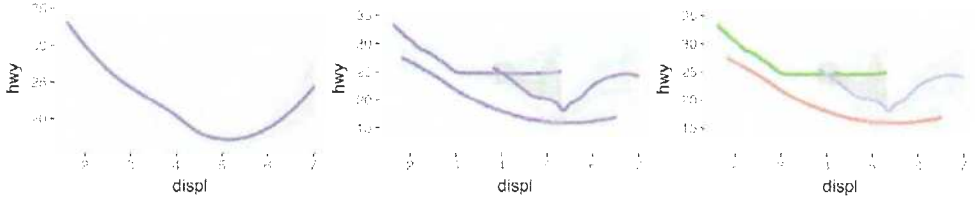


Рис. 1.16

Чтобы отобразить на одной диаграмме графики с разными геометриями, добавьте к `ggplot()` соответствующие функции `geom_xxx()` (рис. 1.17).

```
  (data = mpg) +
  geom_poi  (mapping = aes(x = displ, y = hwy)) +
  geom_sm   (mapping = ae (x = displ, y = hwy))
```

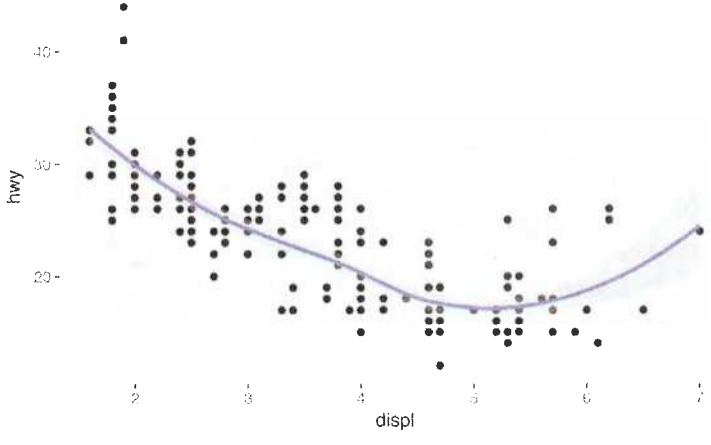


Рис. 1.17

Однако это приводит к частичному дублированию кода. Что если вы захотите отображать вдоль оси не переменную `hwy`, а переменную `cty`? В этом случае вам нужно будет заменить переменную в двух местах, и вполне возможно, что об одном из них вы забудете. Можно избежать такого рода повторений кода, передавая функции `ggplot()` набор сопоставлений в качестве параметра. Пакет **ggplot2** будет трактовать эти сопоставления как глобальные, т.е. применяемые к каждой геометрии, используемой на диаграмме.

Иными словами, приведенный ниже код создает ту же диаграмму, что и предыдущий.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth()
```

Если же поместить сопоставления в функцию `geom_xxx()`, то **ggplot2** будет трактовать их как локальные для каждого слоя. Они будут использоваться лишь для расширения или отмены глобальных сопоставлений, относящихся *только к данному слою*. Это делает возможным отображение разных визуальных атрибутов в разных слоях (рис. 1.18).

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth()
```

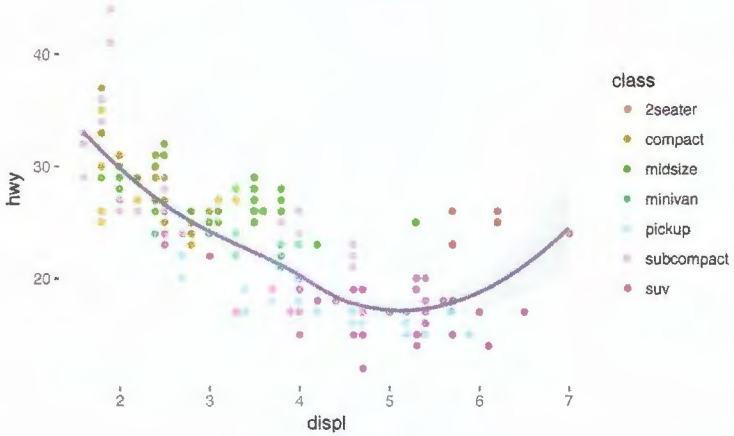


Рис. 1.18

Ту же идею можно использовать для указания различных данных для каждого слоя. В приведенном ниже примере гладкая линия отображает лишь подмножество набора данных `mpg`, соответствующее субкомпактным автомобилям. Локальный аргумент `data`, передаваемый функции `geom_smooth()`, перекрывает глобальный аргумент `data`, переданный функции `ggplot()`, только для данного слоя (рис. 1.19).

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point(mapping = aes(color = class)) +
  geom_smooth(
    data = filter(mpg, class == "subcompact"),
    se = FALSE
  )
```

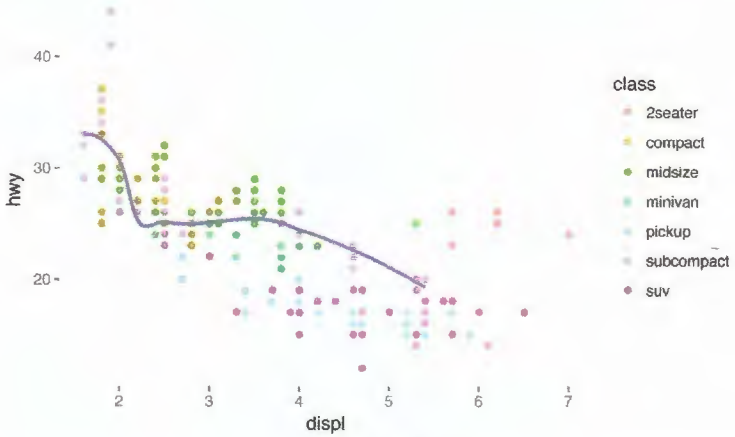


Рис. 1.19

С работой функции `filter()` вы ознакомитесь в следующей главе, а пока что вам достаточно знать, что эта команда выбирает лишь субкомпактные автомобили.

Упражнения

1. Каким бы объектом `geom` вы воспользовались для создания линейного графика? Полосчатого графика? Гистограммы? Комбинированной гистограммы?
2. Мысленно выполните приведенный ниже код и постарайтесь предсказать, как будет выглядеть окончательный результат. После этого выполните код в среде R, чтобы увидеть, что получается на самом деле.

```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy, color = drv)
) +
  geom_point() +
  geom_smooth(se = FALSE)
```

3. Что делает инструкция `show.legend = FALSE`? Что произойдет, если ее удалить? Как вы считаете, почему я использовал ее в одном из примеров?
4. Каково назначение аргумента `se` функции `geom_smooth()`?
5. Будут ли выглядеть одинаково создаваемые ниже графики? Обоснуйте свой ответ.

```
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) +
  geom_point() +
```

```
  geom_smooth()

ggplot() +
  geom_point(
    data = mpg,
    mapping = aes(x = displ, y = hwy)
  ) +
  geom_smooth(
    data = mpg,
    mapping = aes(x = displ, y = hwy)
  )
```

6. Воспроизведите R-код, необходимый для создания следующих графиков (рис. 1.20–1.22).

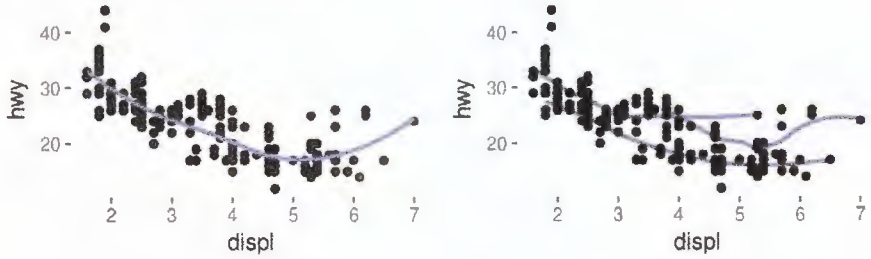


Рис. 1.20

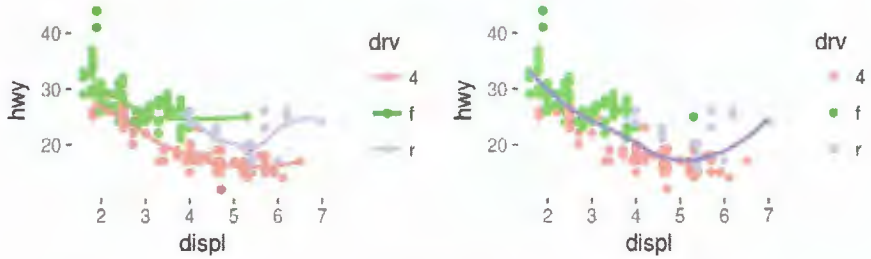


Рис. 1.21

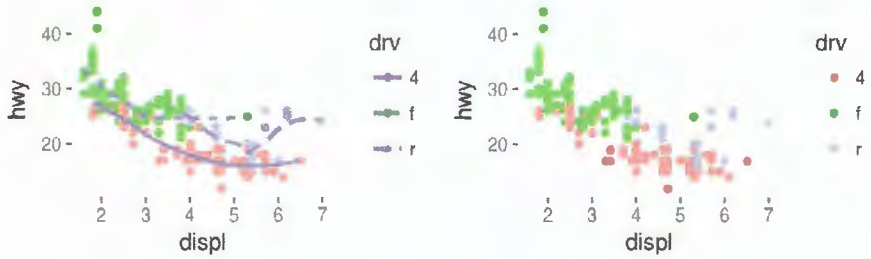


Рис. 1.22

Статистические преобразования

Рассмотрим столбчатую диаграмму. Несмотря на кажущуюся простоту, столбчатые диаграммы заслуживают внимания, поскольку позволяют выявить некоторые тонкие закономерности поведения графиков. Обратимся к примеру столбчатой диаграммы, созданной с помощью функции `geom_bar()`. Приведенная ниже диаграмма показывает общее количество алмазов в наборе данных `diamonds`, сгруппированных по качеству обработки. Передаваемый **ggplot2** набор данных `diamonds` содержит информацию примерно о 54000 алмазов, включая цену, вес, цвет, чистоту и качество огранки каждого из них. Из диаграммы следует, что в представленном наборе данных количество высококачественных алмазов превышает количество низкокачественных (рис. 1.23).

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```

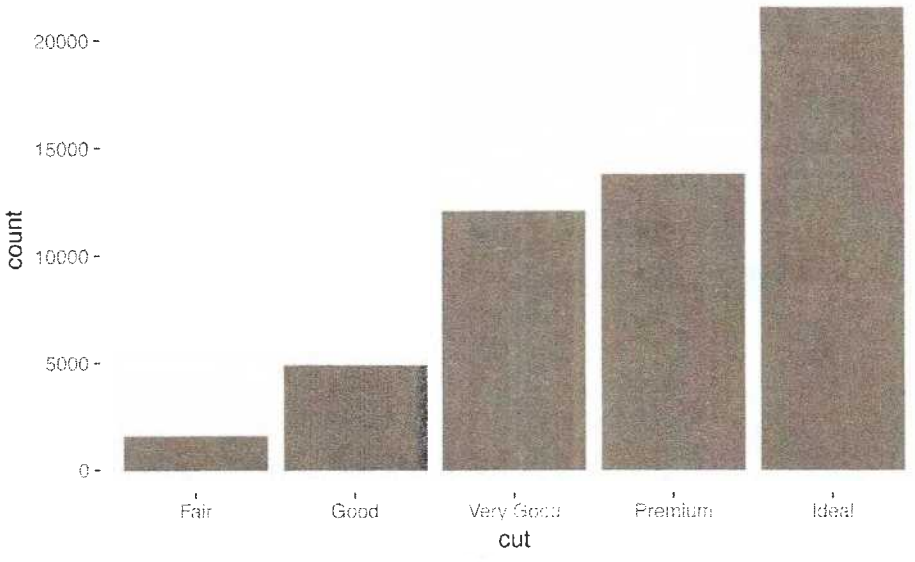


Рис. 1.23

Вдоль оси *x* отображается качество обработки алмазов, `cut` — переменная из набора данных `diamonds`. Вдоль оси *y* откладывается количество алмазов, `count`, но эта переменная не входит в набор данных `diamonds`! Откуда берутся значения `count`? Многие графики, такие как диаграммы рассеяния,

отображают исходные значения переменных, входящих в набор данных. Другие же графики, такие как столбчатые диаграммы, требуют расчета значений.

- Столбчатые диаграммы, гистограммы и частотные многоугольники разбивают данные на определенные интервалы (карманы), а затем отображают на графике количество точек данных, попадающих в каждый из интервалов.
- В графиках на основе гладких кривых сначала выбирается модель аппроксимации данных кривой, а затем на основании прогнозных значений модели строится график.
- При построении ящичковых диаграмм вычисляются надежные итоги распределения и отображается специальным образом отформатированный прямоугольник.

Алгоритм, по которому вычисляются новые значения для графика, называется *статистикой* (`stat`). Следующий рисунок поясняет, как работает этот процесс в случае функции `geom_bar()` (рис. 1.24).

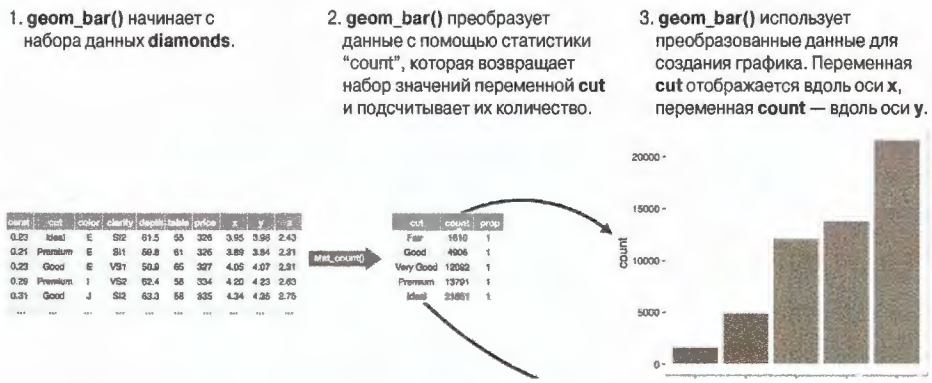


Рис. 1.24

Вы можете узнать, какую статистику использует интересующая вас функция `geom_xxx()`, проверив значение по умолчанию ее аргумента `stat`. Например, команда `?geom_bar` отображает для аргумента `stat` значение "count" (счет), означающее, что функция `geom_bar()` использует статистику, вычисляемую с помощью функции `stat_count()`. Документация к функции `stat_count()` располагается на той же странице, что и документация к функции `geom_bar()`, и если вы прокрутите страницу вперед, то найдете раздел "Computed variables" ("Вычисляемые переменные"), в котором указаны две вычисляемые переменные: `count` и `prop`.

Обычно функции `geom_xxx()` и `stat_xxx()` являются взаимозаменяемыми. Например, предыдущий график можно создать, используя функцию `stat_count()` вместо функции `geom_bar()` (рис. 1.25).

```
ggplot(data = diamonds) +
  stat_count(mapping = aes(x = cut))
```

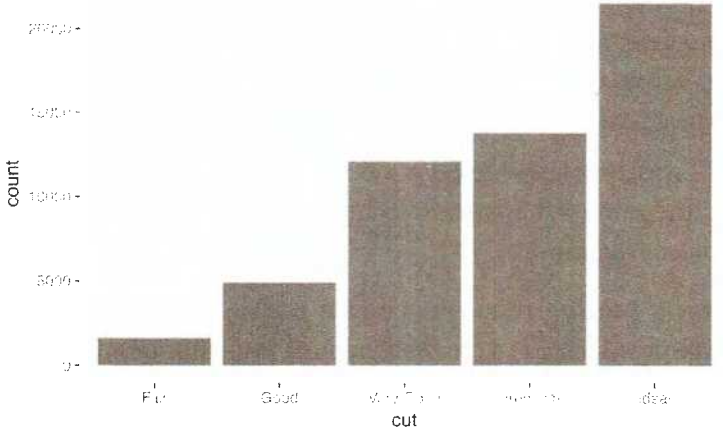


Рис. 1.25

Это возможно, поскольку с каждой геометрией по умолчанию связана определенная статистика, а с каждой статистикой — определенная геометрия. Отсюда следует, что, как правило, можно использовать геометрию, не заботясь о связывании с ней статистического преобразования. Необходимость в явном задании статистики может возникать по следующим трем причинам.

- Вы хотите использовать другую статистику вместо той, которая задана по умолчанию. В приведенном ниже коде я использую для функции `geom_bar()` статистику `identity` вместо `count` (задана по умолчанию). Это позволило мне сопоставить исходным значениям переменной `y` высоту столбцов диаграммы. К сожалению, говоря о столбчатых диаграммах, часто не уточняют, что именно имеется в виду: столбчатая диаграмма этого типа, в которой высота столбцов уже имеется в составе данных, или столбчатая диаграмма предыдущего типа, в которой высота столбцов генерируется путем подсчета строк (рис. 1.26).

```
demo <- tibble(
  ~a,      ~b,
  "bar_1", 20,
  "bar_2", 30,
```

```
  "bar_3", 40
)

ggplot(data = demo) +
  geom_bar(
    mapping = aes(x = a, y = b), stat = "identity"
  )
```

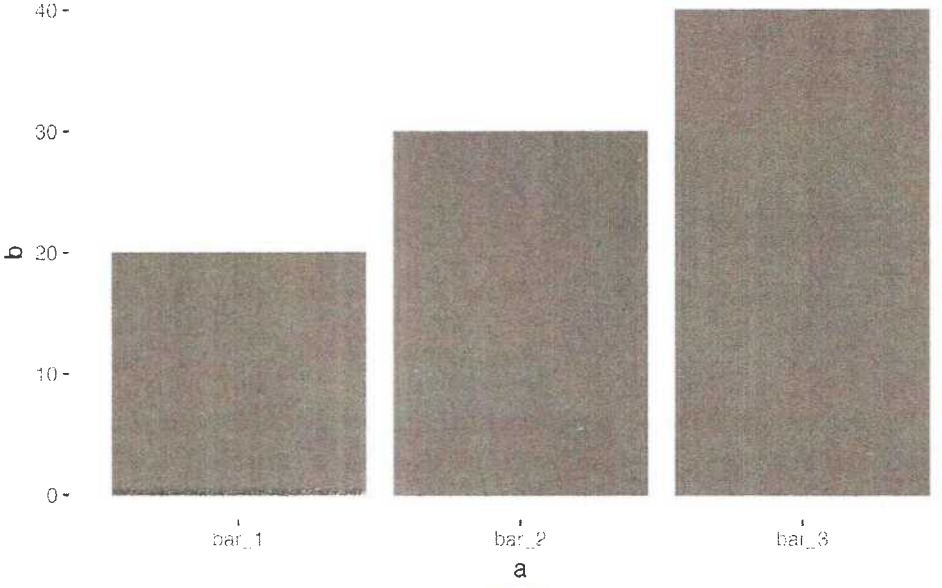


Рис. 1.26

(Не задумывайтесь пока что о фигурирующих в приведенной выше инструкции элементах `<-` или `tribble()`, с которыми вы еще не сталкивались. Вы можете догадаться об их назначении из контекста, но вскоре они будут рассмотрены более подробно.)

- Возможно, вам понадобится установить соответствия между преобразованными переменными и визуальными атрибутами, отличные от тех, которые заданы по умолчанию. Например, вы хотите отобразить столбчатую диаграмму пропорций (`..prop..`), а не количеств (`count`), как показано на рис. 1.27.

```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, y = ..prop.., group = 1)
  )
```

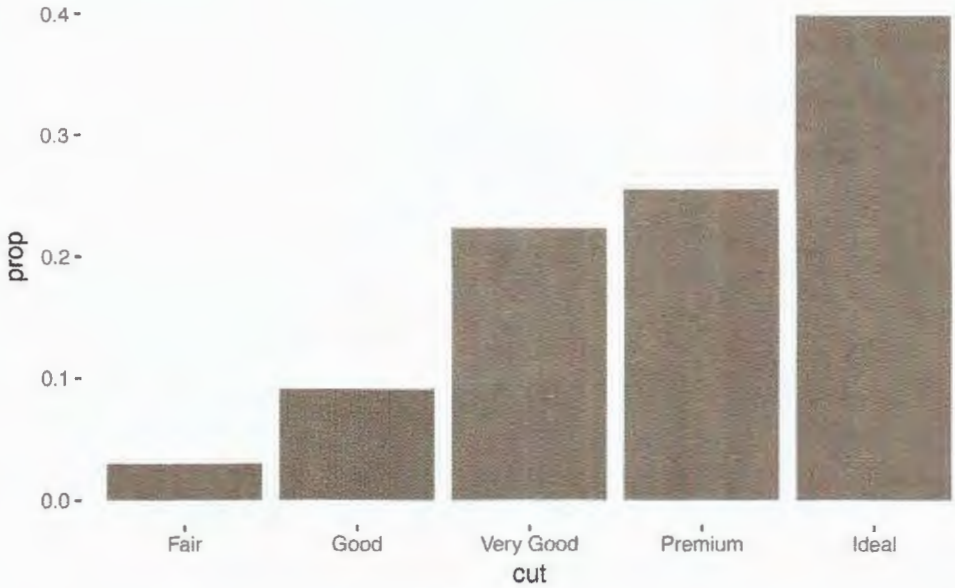


Рис. 1.27

Чтобы узнать больше о переменных, вычисляемых этой статистикой, обратитесь к справочному разделу "Computed variables".

- Возможно, вы захотите привлечь больше внимания к статистическому преобразованию, используемому в коде. Например, вы можете использовать функцию `stat_summary()`, суммирующую значения `y` для каждого уникального значения `x`, чтобы сделать акцент на вычисляемой итоговой характеристике (рис. 1.28).

```
ggplot(data = diamonds) +
  stat_summary(
    mapping = aes(x = cut, y = depth),
    fun.ymin = min,
    fun.ymax = max,
    fun.y = median
  )
```

Пакет **ggplot2** предоставляет около 20 готовых к использованию статистик. Каждая статистика (`stat`) — это функция, справку о которой можно получить обычным путем, например: `?stat_bin`. Полный список статистик можно посмотреть в "шпаргалке" к **ggplot2**.

Рис. 1.28

Упражнения

1. Какая геометрия связана со статистикой `stat_summary()` по умолчанию? Как бы вы переписали код предыдущего графика, чтобы использовать функцию `geom_xxx()` вместо функции `stat_xxx()`?
2. Что делает функция `geom_col()`? В чем ее отличие от функции `geom_bar()`?
3. Большинство геометрий и статистик группируются попарно, и элементы такой пары почти всегда используются совместно. Прочитайте документацию и составьте список таких пар. Что у них имеется общего?
4. Какие переменные вычисляет статистика `stat_smooth()`? Какими параметрами регулируется это поведение?
5. При построении диаграммы пропорций мы установили для параметра `group` значение `1`. Почему? Иными словами, какая проблема возникнет при попытке построения следующих двух графиков?

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, y = ..prop..))
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = color, y = ..prop..)
  )
```

Позиционные настройки

Со столбчатыми диаграммами связан еще один удивительный элемент. Для цветового оформления столбчатой диаграммы можно использовать визуальный атрибут `color` или, в большинстве случаев, атрибут `fill` (рис. 1.29).

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, color = cut))
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = cut))
```

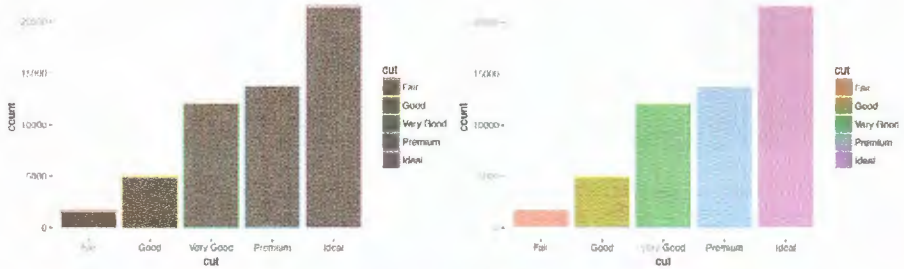


Рис. 1.29

Обратите внимание на то, что происходит, когда мы связываем визуальный атрибут `fill` с другой переменной, например `clarity`: столбцы автоматически преобразуются в стеки. Каждый цветной прямоугольник представляет комбинацию значений переменных `cut` и `clarity` (рис. 1.30).

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut, fill = clarity))
```

Автоматическое создание стеков обеспечивается позиционной настройкой, задаваемой с помощью аргумента `position`. Если вас не устраивает стековая столбчатая диаграмма, воспользуйтесь одной из трех других опций: `"identity"`, `"dodge"` или `"fill"`.

- Настройка `position = "identity"` помещает каждый объект в точности туда, где он и должен был бы находиться в контексте диаграммы. Этот вариант не очень хорош для столбцов, поскольку приводит к их перекрыванию. Чтобы заметить это перекрывание, необходимо сделать

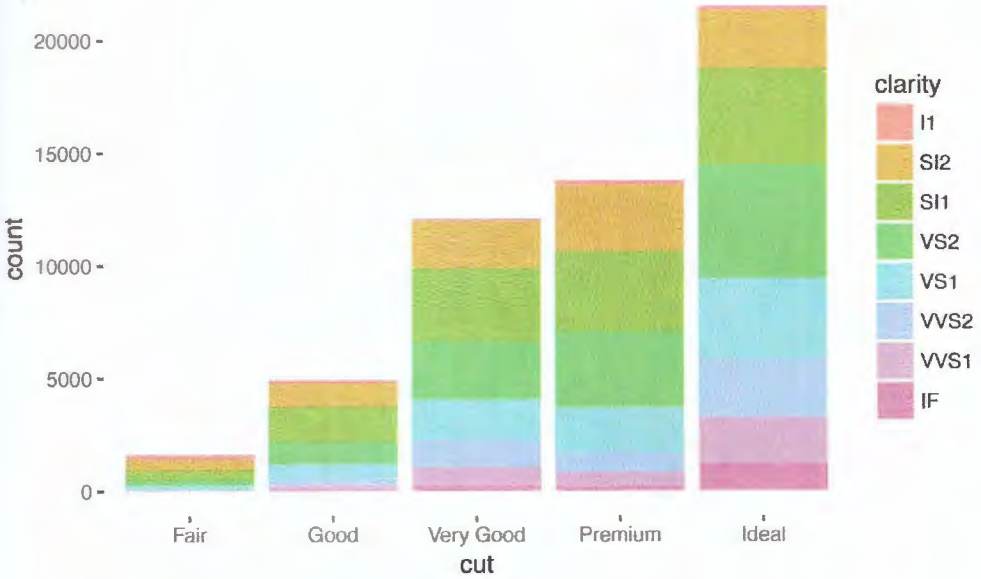


Рис. 1.30

столбцы либо частично прозрачными, установив для параметра `alpha` небольшое значение, либо полностью прозрачными, опустив указание соответствия для визуального атрибута `fill` (рис. 1.31 и 1.32).

```
ggplot(
  data = diamonds,
  mapping = aes(x = cut, fill = clarity)
) +
  geom_bar(alpha = 1/5, position = "identity")
ggplot(
  data = diamonds,
  mapping = aes(x = cut, color = clarity)
) +
  geom_bar(fill = NA, position = "identity")
```

Вариант позиционной настройки `identity` больше всего подходит для двухмерных геометрических объектов, таких как точки, для которых он используется по умолчанию.

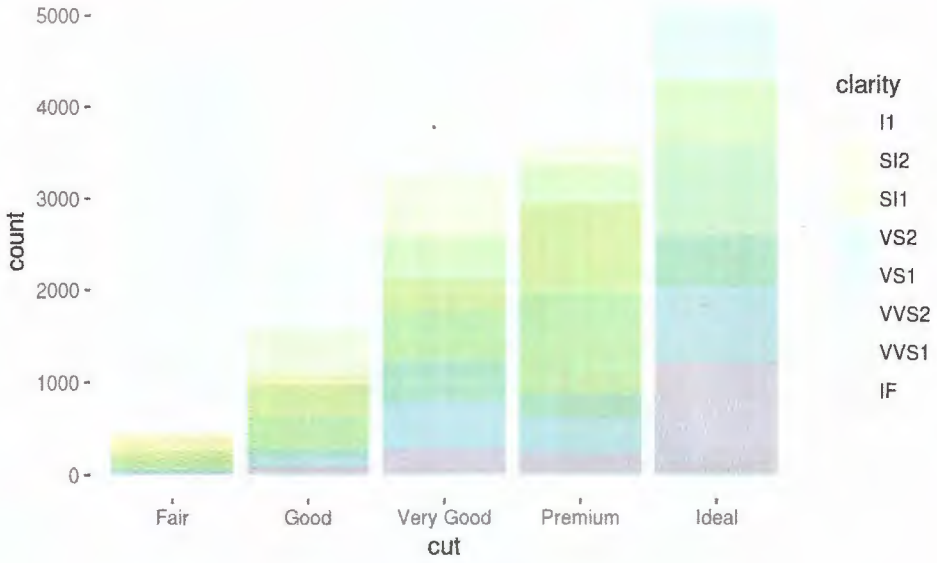


Рис. 1.31

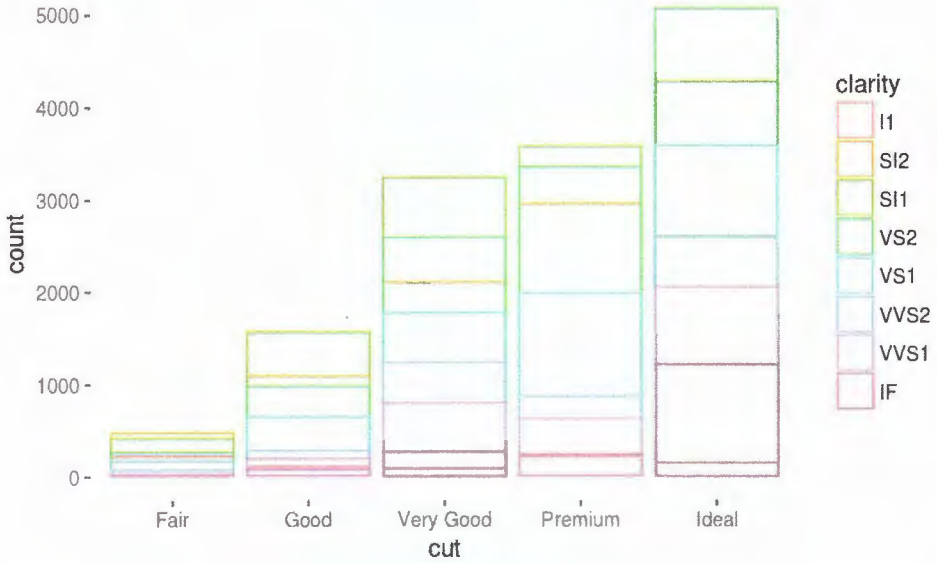


Рис. 1.32

- Настройка `position = "fill"` работает подобно предыдущей, но создает стековые столбцы одинаковой высоты. Это облегчает сравнение пропорций между группами (рис. 1.33).

```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = clarity),
    position = "fill"
)
```

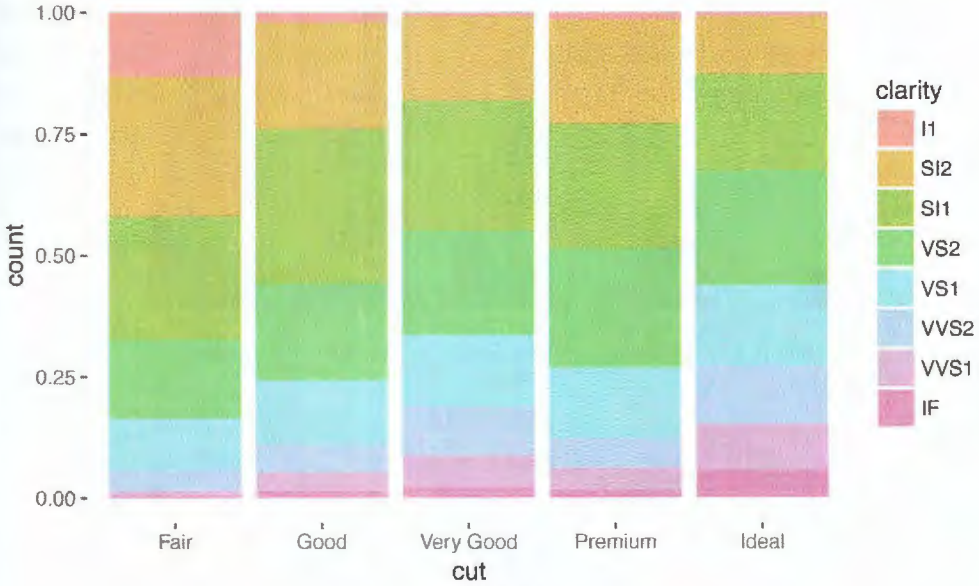


Рис. 1.33

- Настройка `position = "dodge"` располагает перекрывающиеся объекты непосредственно рядом друг с другом. Это упрощает сравнение индивидуальных значений (рис. 1.34).

```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = clarity),
    position = "dodge"
  )
```

Существует еще один тип настроек, которые не подходят столбчатым диаграммам, но могут быть весьма полезными в случае диаграмм рассеяния. Вспомните наш первый пример диаграммы рассеяния. Вы обратили внимание на то, что на диаграмме отображены только 126 точек, хотя набор содержит данные о 234 наблюдениях (рис. 1.35)?

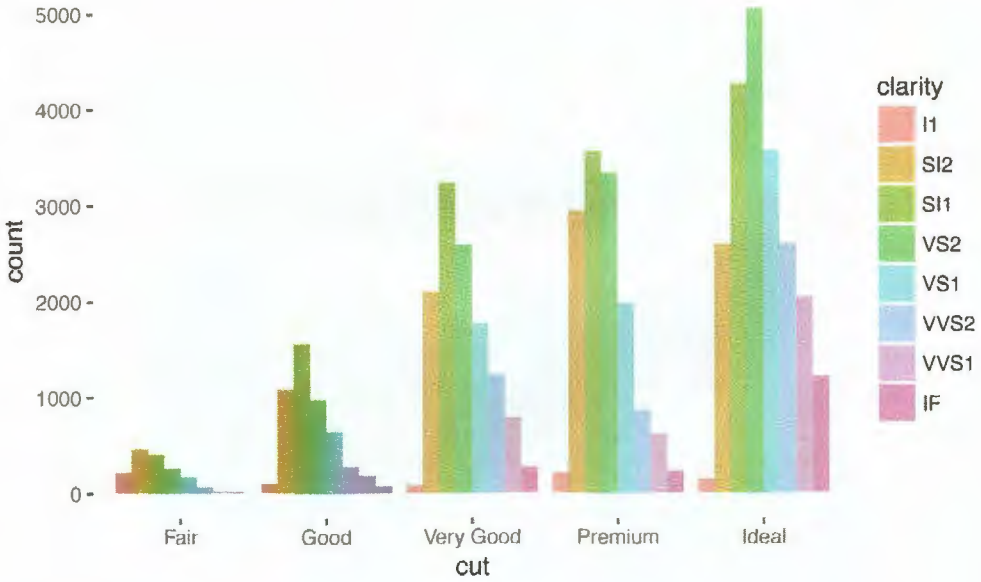


Рис. 1.34

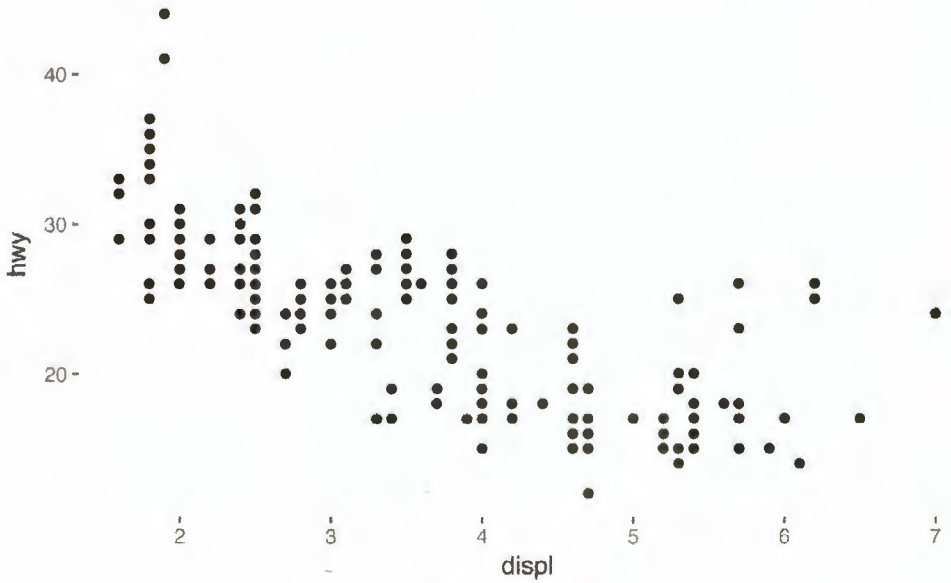


Рис. 1.35

Значения переменных `hwy` и `displ` округлены, чтобы соответствующие точки располагались в узлах сетки, вследствие чего многие точки располагаются одна поверх другой. Эта проблема носит название *наложение графиков*.

Когда она возникает, определить места графика, соответствующие наибольшему скоплению точек, затруднительно. Распределены ли точки равномерно по всему графику, или существует одна особая комбинация значений переменных hwy и displ, которая содержит всего лишь 109 значений?

Этой проблемы можно избежать, установив для позиционной настройки position значение jitter. Настройка position = "jitter" добавляет к каждой точке небольшой случайный шум. Это несколько разносит позиции точек, поскольку маловероятно, чтобы две точки получили одно и то же случайное смещение (рис. 1.36).

```
ggplot(data = mpg) +
  geom_point(
    mapping = aes(x = displ, y = hwy),
    position = "jitter"
  )
```

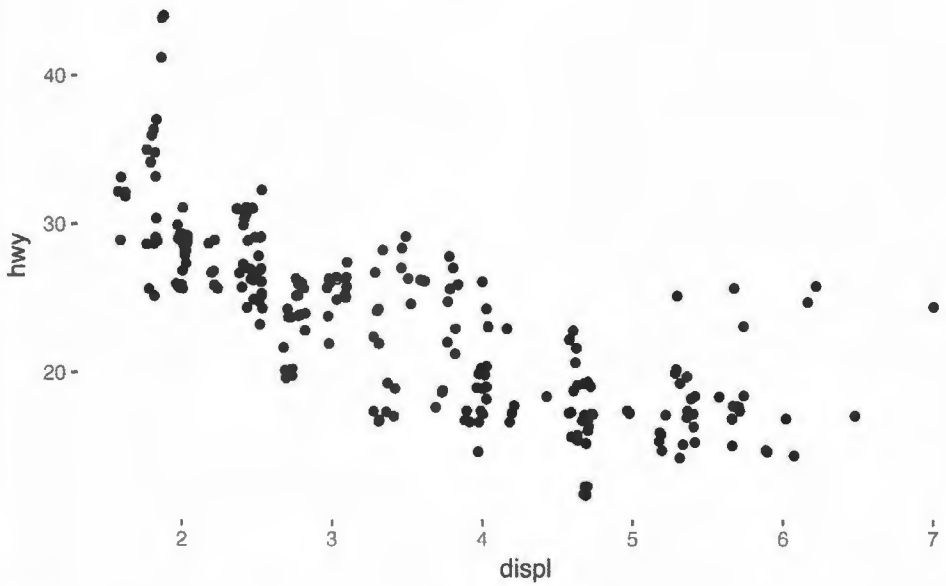


Рис. 1.36

Добавление элемента случайности в график может показаться довольно странным способом его улучшения, однако график в целом становится при этом более информативным, тогда как вносимая неточность сказывается лишь в мелких масштабах. Учитывая полезность данной операции, в **ggplot2** была введена сокращенная запись вызова функции geom_point(position = "jitter"): geom_jitter().

Чтобы узнать больше о настройке position, обратитесь к страницам справки, содержащим информацию о каждой из следующих настроек: ?position_dodge, ?position_fill, ?position_identity, ?position_jitter и ?position_stack.

Упражнения

1. Какая проблема связана с приведенным ниже графиком (рис. 1.37)? Как это можно исправить?

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point()
```

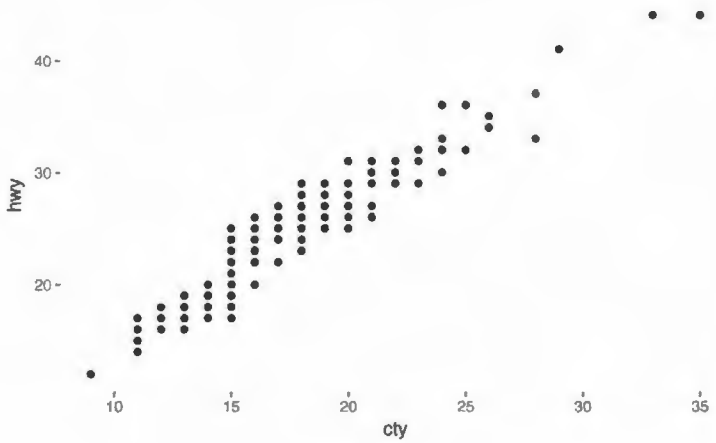


Рис. 1.37

2. Какие параметры функции geom_jitter() управляют величиной случайных флуктуаций данных?
3. Сравните и противопоставьте функцию geom_jitter() функции geom_count().
4. Укажите позиционную настройку по умолчанию для функции geom_boxplot()? Создайте визуализацию набора данных mpg, с помощью которой это можно продемонстрировать.

Системы координат

Пожалуй, наиболее сложной частью модуля **ggplot2** являются системы координат. По умолчанию используется декартова система координат, в

которой положение каждой точки задается ее координатами вдоль ортогональных осей x и y. Существует ряд других систем координат, которые будут полезными в тех или иных ситуациях.

- Функция coord_flip() меняет местами оси x и y. Это, например, может пригодиться, если вы хотите построить полосчатую диаграмму вместо столбчатой. Кроме того, это может понадобиться в случае длинных подписей, которые будут перекрываться, если расположить их вдоль оси x (рис. 1.38).

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot() +
  coord_flip()
```

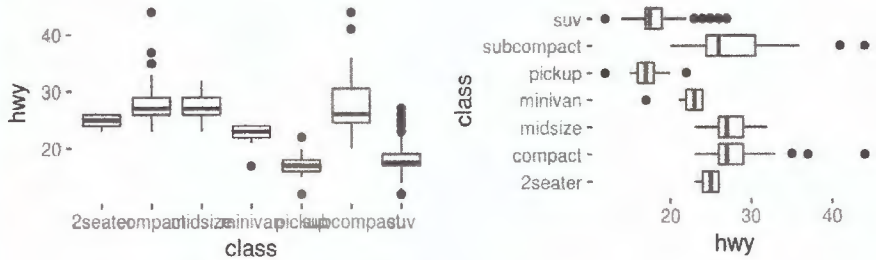


Рис. 1.38

- Функция coord_quickmap() корректно устанавливает форматное соотношение для карт. Это очень важно, если вы отображаете географические данные с помощью **ggplot2** (рис. 1.39). (К сожалению, ограниченность объема данной книги не позволяет обсудить этот вопрос более подробно.)

```
nz <- map_data("nz")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", color = "black")

ggplot(nz, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", color = "black") +
  coord_quickmap()
```

- Функция coord_polar() использует полярные координаты. Они проявляют интересную связь между столбчатой диаграммой и кокскомб-диаграммой (рис. 1.40).

Рис. 1.39

```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

bar + coord_flip()
bar + coord_polar()
```

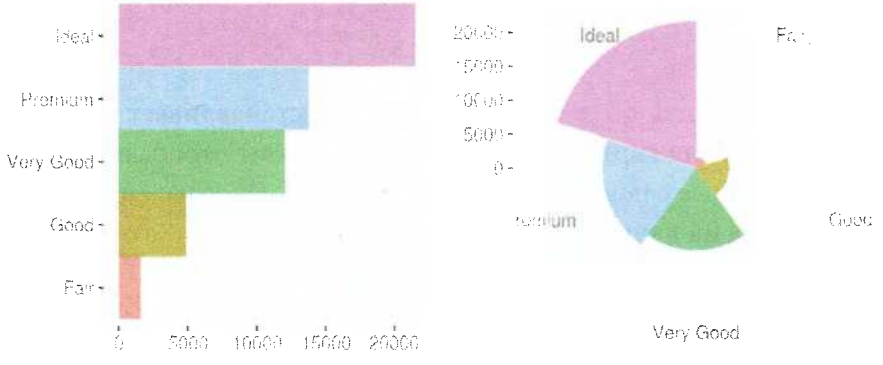


Рис. 1.40

Упражнения

1. Преобразуйте стековую диаграмму в круговую, используя функцию `coord_polar()`.
2. Что делает функция `labs()`? Прочтите документацию.

3. Чем отличаются функции `coord_quickmap()` и `coord_map()`?

4. Какую информацию о соотношении между показателями расхода топлива (`mpg`) при движении автомобиля в городских условиях (`city`) и на трассе (`highway`) предоставляет вам следующий график? Почему важен вызов функции `coord_fixed()`? Какова роль функции `geom_abline()` (рис. 1.41)?

```
ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) +
  geom_point() +
  geom_abline() +
  coord_fixed()
```

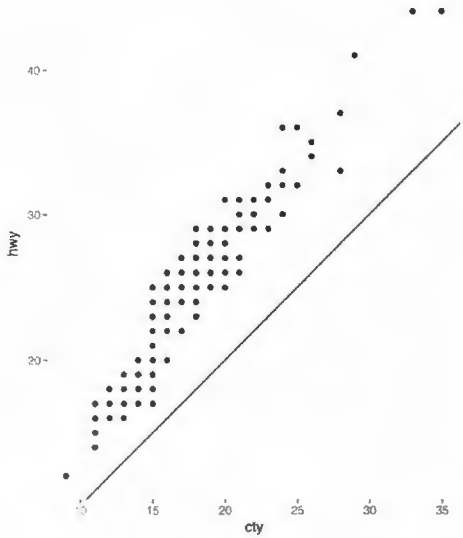


Рис. 1.41

Многослойная грамматика графики

В предыдущих разделах вы научились гораздо большему, чем просто создание диаграмм рассеяния, столбчатых и ящичковых диаграмм. Вы освоили базовые методы, которые сможете использовать для создания диаграмм любого типа с помощью расширения **ggplot2**. Чтобы убедиться в этом, добавим позиционные настройки (`POSITION`), статистики (`STAT`), системы координат (`COORDINATE_FUNCTION`) и панели (`FACET_FUNCTION`) в наш шаблон кода.

```
ggplot(data = <DATA>) +
  <GEOM_FUNCTION>(
```

```
    mapping = aes(<MAPPINGS>),
    stat = <STAT>,
    position = <POSITION>
  ) +
  <COORDINATE_FUNCTION> +
  <FACET_FUNCTION>
```

Наш новый шаблон принимает семь параметров (выделены угловыми скобками). На практике использовать для построения графика все семь параметров вам придется очень редко, поскольку ggplot2 предоставляет полезные значения по умолчанию для всех параметров, за исключением данных, сопоставлений и геометрической функции.

Перечисленные семь параметров шаблона составляют грамматику графики — формальную систему, предназначенную для построения диаграмм. Грамматика графики основана на понимании того, что любую диаграмму можно однозначно описать, задав комбинацию набора данных, геометрии, набора сопоставлений, статистических преобразований, координатной системы и схемы аспектов.

Чтобы понять, как это работает, представьте, что вам нужно создать базовый график с "чистого листа": вы можете начать с набора данных, а затем преобразовать его (используя статистические преобразования) в информацию, которую хотите отобразить (рис. 1.42).

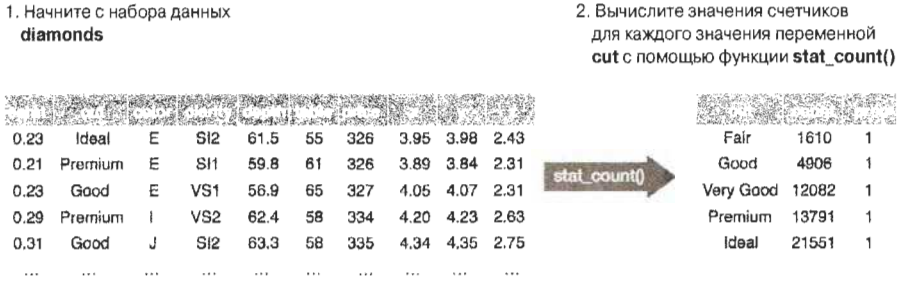


Рис. 1.42

Затем вы можете выбрать геометрический объект для представления каждого наблюдения в преобразованных данных, а далее использовать визуальные атрибуты (эстетические свойства) геометрий для представления переменных, имеющихся в данных. Для этого вы сопоставите значениям каждой переменной различные уровни визуального атрибута (рис. 1.43).

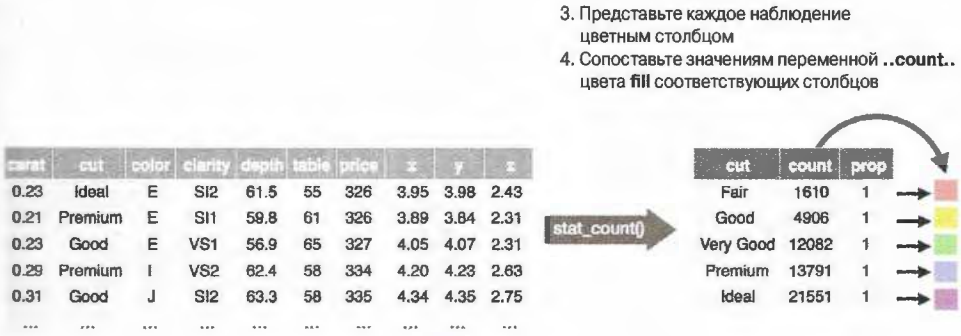


Рис. 1.43

После этого вы выберете систему координат, в которую должны быть помещены геометрические объекты. Для отображения значений переменных `x` и `y` вы используете расположение объектов (которое само является визуальным атрибутом). К этому времени вы уже имеете готовый график, но вы также могли бы дополнительно настроить позиционные параметры (`position`) геометрических объектов в системе координат или разбить график на отдельные вспомогательные графики (аспекты, срезы). Кроме того, вы могли бы добавить в график один или несколько дополнительных слоев, в каждом из которых используется дополнительный набор данных, геометрический объект, набор сопоставлений, статистика и позиционная настройка (рис. 1.44).

Рис. 1.44

Этот метод можно использовать для создания любого графика, который вам подскажет ваше воображение. Иными словами, изученный вами в этой главе шаблон кода можно использовать для создания сотен тысяч уникальных графиков.

ГЛАВА 2

Рабочий процесс: основы

К этому моменту вы уже приобрели кое-какой опыт работы с R-кодом. Мы не рассказали о многих деталях, но вы несомненно и сами могли о многом догадаться, иначе отложили бы книгу с раздражением в сторону. Появление чувства раздражения у тех, кто только начинает программировать на языке R, вполне естественно, поскольку этот язык необычайно придирчив в отношении пунктуации, и даже один неправильно расположенный символ может стать причиной вывода сообщения об ошибке. Успокаивайте себя тем, что это чувство временное: его испытывают все новички, и единственный способ преодолеть его — не падать духом!

Прежде чем двигаться дальше, мы должны более подробно рассмотреть ряд базовых понятий программирования на языке R и обсудить наиболее полезные возможности интерактивной среды разработки RStudio.

Основы написания кода

Сейчас нам предстоит обсудить важные понятия, рассмотрению которых ранее не было уделено должного внимания исключительно ради того, чтобы как можно быстрее дать вам возможность увидеть практические результаты применения R-кода для построения графиков. Начнем с примера, демонстрирующего использование программы на языке R в качестве обычного калькулятора.

```
1 / 200 * 30
#> [1] 0.15
(59 + 73 + 2) / 3
#> [1] 44.7
sin(pi / 2)
#> [1] 1
```

Новые объекты можно создавать с помощью оператора `<-`:

```
x <- 3 * 4
```

Все команды R, с помощью которых создаются объекты (команды присваивания), имеют одинаковый синтаксис:

```
имя_объекта <- значение
```

Мысленно этот код следует читать так: “Объект с указанным именем получает указанное значение”.

Вам придется делать множество присваиваний, и постоянный ввод оператора `<-` вскоре может показаться неудобным. Однако не поддавайтесь искушению использовать вместо него оператор `=`: он будет работать, но впоследствии может возникнуть путаница. Гораздо лучше воспользоваться комбинацией клавиш <Alt+-> (знак “минус”). Обратите внимание на то, что RStudio автоматически окружает оператор `<-` пробелами, что является правилом хорошего тона в отношении форматирования кода. Пробелы улучшают читаемость листингов, в результате чего при чтении кода глаза будут меньше уставать.

Что представляют собой имена

Имена объектов должны начинаться с буквы и могут содержать лишь буквы и цифры, а также символы `_` (подчеркивание) и `.` (точка). Лучше выбирать описательные имена, и в этом случае целесообразно следовать определенным соглашениям. Мы рекомендуем использовать *змеиный_регистр* (snake_case) — стиль написания, в котором слова разделяются символом подчеркивания и начинаются с маленькой буквы.

```
я_использую_змеиный_регистр
другиеЛюдиИспользуютВерблюжийРегистр
некоторые.люди.используют.точки
А_некоторые.Вообще_нЕ_СЛЕДУЮТ_никаким.Соглашениям
```

К обсуждению стилевого оформления кода мы еще вернемся в главе 15.

Можно проверить содержимое любого объекта, введя его имя.

```
x
#> [1] 12
```

Вот еще один пример операции присваивания:

```
this_is_a_really_long_name <- 2.5
```

Проверьте содержимое этого объекта, воспользовавшись средством завершения ввода RStudio: введите “this”, нажмите клавишу <Tab> и добавляйте символы до тех пор, пока не получите уникальный префикс, после чего нажмите клавишу <Enter> или <Return>.

Вот досада — вы ошиблись! Объекту `this_is_a_really_long_name` надо было присвоить значение 3,5, а не 2,5. Чтобы исправить ошибку, воспользуйтесь другой комбинацией клавиш. Введите “this”, а затем нажмите комбинацию клавиш <Ctrl+↑>. Вы получите список всех введенных вами команд, начинающихся с этого текста. Для перемещения по списку используйте клавиши со стрелками, а затем нажмите клавишу <Enter> для повторного набора нужной команды. Введите `3.5` вместо `2.5` и нажмите клавишу <Enter>.

Выполните другую операцию присваивания:

```
r_rocks <- 2 ^ 3
```

Проверим результат.

```
r_rock
#> Error: object 'r_rock' not found
R_rocks
#> Error: object 'R_rocks' not found
```

Между вами и средой R действует негласное соглашение: среда выполнит за вас любой трудоемкий расчет, но взамен ваши команды должны быть предельно точными. Опечатки недопустимы. Регистр букв имеет значение.

Вызов функций

В языке R имеется большая коллекция встроенных функций, которые можно вызывать следующим образом:

```
имя_функции(arg1 = val1, arg2 = val2, ...)
```

Испытаем функцию `seq()`, которая создает последовательности чисел, попутно изучив другие полезные средства RStudio. Начните ввод команды с букв `se` и нажмите клавишу <Tab>. Откроется список с возможными вариантами автозавершения ввода. Укажите функцию `seq()`, устранив неоднозначность путем ввода дополнительной буквы (“q”), или выберите ее, пользуясь клавишами со стрелками (<↑/↓>). Обратите внимание на всплывающую подсказку, содержащую краткую информацию о назначении функции и ее аргументах. Если этой информации вам недостаточно, нажмите клавишу <F1>

для получения более подробных сведений, которые отобразятся в нижней части панели справа.

Выбрав нужную функцию, нажмите клавишу <Tab> еще один раз. RStudio автоматически добавит открывающую (() и закрывающую ()) скобки после имени функции. Введите аргументы 1, 10 и нажмите клавишу <Enter> (<Return>).

```
seq(1, 10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
```

Введите следующий код, обратив внимание на аналогичную помощь со стороны RStudio при вводе парных кавычек:

```
x <- "hello world"
```

Кавычки и скобки всегда должны быть парными. RStudio пытается оказать вам максимальную помощь при их вводе, но даже при этом возможно появление скобок и кавычек, не имеющих пары. Признаком того, что это произошло, служит знак +.

```
> x <- "hello
+
```

Появление знака + сигнализирует о том, что RStudio ожидает дополнительного ввода и считает, что он не завершен. Как правило, это означает, что вы забыли ввести один из символов " или). В таком случае либо введите соответствующий парный символ, либо нажмите клавишу <Esc>, чтобы сбросить выражение и ввести его заново.

Выполнение операции присваивания не сопровождается отображением результата, однако вы можете перепроверить себя.

```
y <- seq(1, 10, length.out = 5)
y
#> [1]  1.00  3.25  5.50  7.75 10.00
```

Можно ускорить выполнение этих действий, заключив команду в круглые скобки, что приведет к немедленному выводу результата на экран после выполнения команды.

```
(y <- seq(1, 10, length.out = 5))
#> [1]  1.00  3.25  5.50  7.75 10.00
```

На рис. 2.1 показан текущий вид среды, отображаемый на правой панели.

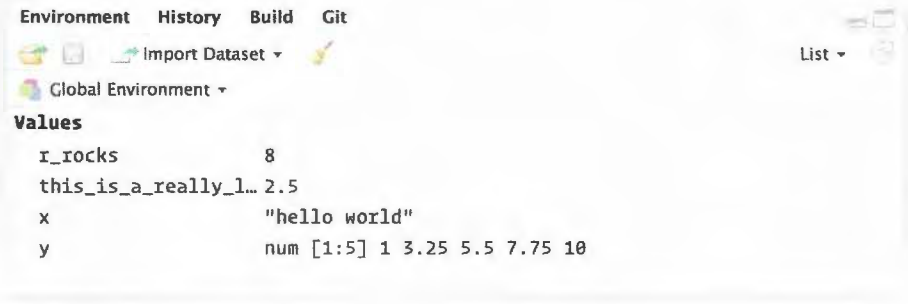


Рис. 2.1

Здесь представлены все объекты, созданные вами к данному моменту.

Упражнения

1. Почему не работает этот код?

```
my_variable <- 10
my_variable
#> Error in eval(expr, envir, enclos):
#> object 'my_variable' not found
```

   Внимательно присмотритесь к коду! (Возможно, это упражнение кому-то покажется бессмысленным, но затраты времени на тренировку способности замечать даже малейшие отличия в коде с лихвой окупится, когда вы приступите к программированию.)

2. Исправьте каждую из приведенных ниже R-команд, чтобы обеспечить их корректное выполнение.

```
library(tidyverse)

ggplot(dota = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy))

fliter(mpg, cyl = 8)
filter(diamond, carat > 3)
```

3. Нажмите комбинацию клавиш <Alt+Shift+K>. К чему это привело? Как переместиться в то же место в коде, используя меню?

ГЛАВА 3

Преобразование данных с помощью пакета dplyr

Введение

Визуализация — важный инструмент, позволяющий глубже вникать в смысл данных, но последние редко поступают к вам именно в том виде, который нужен. Вам часто потребуется создавать новые переменные или итоговые данные, а также переименовывать переменные или реорганизовывать наблюдения, чтобы немного упростить их обработку. Об этом (и многом другом) вы узнаете в данной главе, в которой научитесь выполнять преобразования с помощью пакета **dplyr** на примере нового набора данных, содержащего информацию об авиарейсах из аэропортов Нью-Йорка за 2013 год.

Используемые ресурсы

В этой главе рассматривается использование пакета **dplyr** — еще одного центрального компонента библиотеки *tidyverse*. Основные идеи демонстрируются с использованием данных из пакета **nycflights13**, а для облегчения анализа данных задействуется пакет **ggplot2**.

```
library(nycflights13)
library(tidyverse)
```

Внимательно отнеситесь к сообщению о возникновении конфликтной ситуации, которое выводится после загрузки библиотеки *tidyverse*. Оно информирует вас о том, что пакет **dplyr** перезаписывает некоторые функции R. Если хотите использовать базовые версии этих функций после загрузки пакета **dplyr**, используйте их полные имена, например `stats::filter()` и `stats::lag()`.

Пакет nycflights13

Чтобы ознакомить вас с базовыми глаголами **dplyr**, обеспечивающими манипулирование данными, мы используем набор данных `nycflights13::flights`. В этом фрейме данных содержится информация о всех 336776 авиарейсах из аэропортов Нью-Йорка, совершенных в 2013 году. Источником данных является база Бюро статистики перевозок США, а их описание вы сможете прочитать, выполнив команду `?flights`.

```
flights
#> # A tibble: 336,776 × 19
#>    year month    day dep_time sched_dep_time dep_delay
#>   <int> <int>  <int>    <int>          <int>     <dbl>
#> 1  2013     1      1      517            515         2
#> 2  2013     1      1      533            529         4
#> 3  2013     1      1      542            540         2
#> 4  2013     1      1      544            545        -1
#> 5  2013     1      1      554            600        -6
#> 6  2013     1      1      554            558        -4
#> # ... with 336,776 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

Возможно, вы обратили внимание на несколько непривычный способ вывода этого фрейма данных: отображаются лишь несколько первых строк и все столбцы, которые умещаются на одном экране. (Если хотите увидеть полный набор данных — отобразите его в окне просмотра RStudio, выполнив команду `View(flights)`.) Дело в том, что это не обычный фрейм данных, а так называемый tibble-фрейм (tibble). *Tibble-фреймы* — это те же фреймы данных, но несколько доработанные для оптимизации их взаимодействия со средствами библиотеки *tidyverse*. На данном этапе различия для вас несущественны, так что вам не о чем беспокоиться. К более подробному рассмотрению tibble-фреймов мы еще вернемся в части II.

Наверное, ваше внимание также привлекла строка с трехбуквенными обозначениями, расположенная под строкой с подписями столбцов. Эти обозначения указывают на тип переменных. В рассматриваемом наборе данных используются следующие типы переменных:

- `int` — целые числа;
- `dbl` — числа с плавающей точкой (вещественные числа);

- chr — символьные векторы, или строки;
- dttm — значения даты и времени (дата + время).

Существуют также три других распространенных типа переменных, которые не используются в этом наборе данных, но будут встречаться в других примерах книги:

- lgl — логические векторы, которые могут содержать только значение TRUE или FALSE;
- fctr — факторы, которые используются в R для представления категориальных переменных с фиксированными возможными значениями;
- date — даты.

Основные сведения о пакете dplyr

В этой главе вы изучите пять основных функций, входящих в пакет **dplyr**, с помощью которых сможете решать ряд сложных задач, связанных с манипулированием данными:

- выбор наблюдений по их значениям — filter();
- перестановка строк — arrange();
- выбор переменных по их именам — select();
- создание новых переменных с использованием функций существующих переменных — mutate();
- сведение нескольких значений в одно итоговое — summarize().

Любая из этих функций может быть использована совместно с функцией group_by(), которая способна изменять область действия функции таким образом, чтобы она применялась как к целому набору данных, так и к отдельным его группам. Эти шесть функций представляют глаголы языка манипулирования данными.

Все глаголы подчиняются одному и тому же синтаксису.

1. Первый аргумент — это фрейм данных.
2. Последующие аргументы, в качестве которых используются (без заключения их в кавычки) имена переменных, описывают действия, которые должны быть выполнены по отношению к фрейму данных.
3. Результатом является новый фрейм данных.

Совокупность этих свойств упрощает создание цепочек простых команд, обеспечивающих получение сложного результата. Рассмотрим работу этих глаголов более подробно.

Фильтрация строк с помощью функции filter()

Функция filter() позволяет извлекать поднаборы наблюдений на основе их значений. Первый ее аргумент — имя фрейма данных. Второй и последующие аргументы — это выражения, обеспечивающие фильтрацию фрейма. Например, можно отобрать все авиарейсы за 1 января.

```
filter(flights, month == 1, day == 1)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 836 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>,origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

При выполнении этой строки кода пакет **dplyr** осуществляет фильтрацию данных и возвращает новый фрейм. Функции **dplyr** никогда не изменяют свои входные данные, и если вы хотите сохранить результат, то вам придется использовать оператор присваивания <-:

```
jan1 <- filter(flights, month == 1, day == 1)
```

R либо выводит результаты, либо сохраняет их в переменной. Если требуется выполнить оба действия сразу, заключите команду присваивания в круглые скобки.

```
(dec25 <- filter(flights, month == 12, day == 25))
#> # A tibble: 719 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013    12    25      456            500        -4
#> 2  2013    12    25      524            515         9
#> 3  2013    12    25      542            540         2
```

```
#> 4  2013    12    25      546            550        -4
#> 5  2013    12    25      556            600        -4
#> 6  2013    12    25      557            600        -3
#> # ... with 713 more rows, and 13 more variables:
#>   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#>   carrier <chr>, flight <int>, tailnum <chr>,origin <chr>,
#>   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#>   minute <dbl>, time_hour <dttm>
```

Сравнения

Чтобы эффективно использовать фильтрацию, вы должны уметь отбирать нужные наблюдения с помощью операторов сравнения. R предоставляет стандартный набор таких операторов: >, >=, <, <=, != (не равно) и == (равно)

Когда кто-то только учится работать с R, он может легко ошибиться и использовать для проверки равенства оператор = вместо ==. В подобных случаях выводится содержательное сообщение об ошибке.

```
filter(flights, month = 1)
#> Error: filter() takes unnamed arguments. Do you need `==`?
```

Существует еще одна распространенная проблема, с которой могут столкнуться новички, применяя оператор == к вещественным числам. Приведенные ниже результаты могут вас удивить!

```
sqrt(2) ^ 2 == 2
#> [1] FALSE
1/49 * 49 == 1
#> [1] FALSE
```

Компьютеры используют арифметику конечной точности (не могут же они хранить бесконечное количество десятичных разрядов!), поэтому не забывайте о том, что любое число, которое вы видите, является приближением. В подобных случаях вместо использования оператора == следует обращаться к функции `near()`.

```
near(sqrt(2) ^ 2, 2)
#> [1] TRUE
near(1 / 49 * 49, 1)
#> [1] TRUE
```

Логические операторы

Если функции `filter()` передаются несколько аргументов, они комбинируются с использованием логических операторов И: строка будет включена

в вывод только в том случае, когда истинно каждое из выражений. Для образования комбинаций других типов вы должны самостоятельно использовать булевы операторы: `&` — И, `|` — ИЛИ и `!` — НЕ, а также функцию `xor()` — исключающее ИЛИ. Полный набор булевых операций представлен на рис. 3.1.

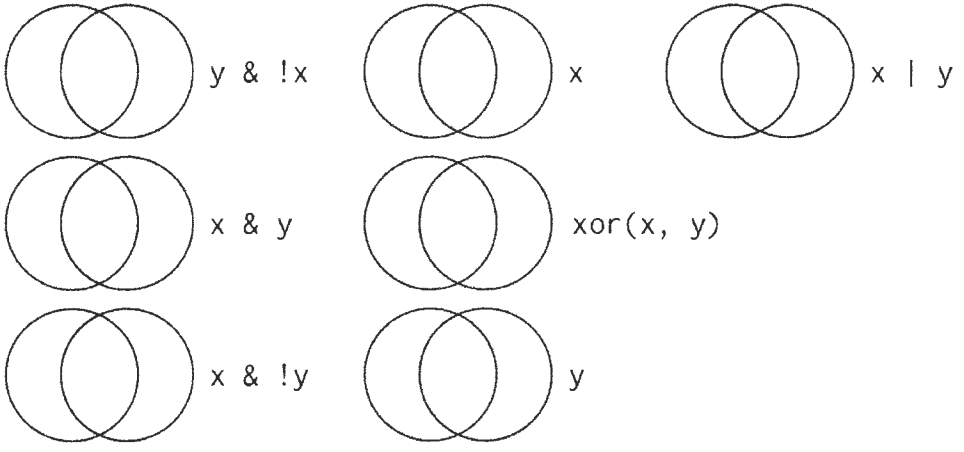


Рис. 3.1

Следующий код находит все авиарейсы, отправленные в ноябре или декабре:

```
filter(flights, month == 11 | month == 12)
```

Порядок выполнения операций отличается от того, который можно было бы ожидать при словесном описании условий поиска. Было бы ошибкой записать эту команду в виде `filter(flights, month == 11 | 12)`, основываясь на ее буквальной интерпретации: "найти все авиарейсы, отправленные в ноябре или декабре". Такая команда нашла бы все месяцы, определяемые результатом вычисления выражения `11 | 12`, который равен TRUE. В числовом контексте (как в данном случае) значением TRUE является 1, и поэтому были бы найдены все вылеты, совершенные в январе, а не в ноябре или декабре. Это довольно тонкий момент!

Удобный способ решения этой проблемы — использовать выражение `x %in% y`. Оно выбирает все строки, в которых `x` является одним из значений, содержащихся в `y`. Перепишем предыдущий код с помощью такого выражения:

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

Иногда сложную задачу выделения поднаборов данных можно упростить, воспользовавшись законами де Моргана: `!(x & y)` — это то же самое, что `!x | !y`, а `!(x | y)` — то же самое, что `!x & !y`. Например, если бы требовалось найти все авиарейсы с задержкой (как по прибытию, так и по отправке) не более двух часов, то для этого можно было бы воспользоваться любым из следующих двух фильтров.

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

Наряду с операторами `&` и `|` в R имеются операторы `&&` и `||`. Не пытайтесь их использовать в данном случае! О применении этих операторов вы узнаете в разделе "Условное выполнение" главы 15.

Всякий раз, когда вы собираетесь использовать сложные составные выражения в функции `filter()`, пытайтесь находить возможность превращать их в явные переменные. Это значительно упростит проверку результатов вашей работы. Вскоре мы поговорим о том, как создавать новые переменные.

Отсутствующие значения

Одной из важных особенностей R, которая может сделать сравнение запутанным, являются отсутствующие (пропущенные) значения, обозначаемые как `NA` (от англ. *not available* — недоступно). `NA` представляет неизвестное значение, поэтому отсутствующие значения "заразны"; результат почти любой операции, включающей неизвестное значение, также будет неизвестен.

```
NA > 5
#> [1] NA
10 == NA
#> [1] NA
NA + 10
#> [1] NA
NA / 2
#> [1] NA
```

Вот результат, который может больше всего сбивать с толку.

```
NA == NA
#> [1] NA
```

Разобраться в том, почему это действительно так, будет проще всего, если немного уточнить контекст.

```
# Пусть x - возраст Мэри. Он нам неизвестен.
x <- NA
```

```
# Пусть y - возраст Джона. Он нам также неизвестен.
y <- NA

# Имеют ли Джон и Мэри одинаковый возраст?
x == y
#> [1] NA
# Мы не знаем!
```

Если вы хотите проверить, является ли значение отсутствующим, используйте функцию `is.na()`.

```
is.na(x)
#> [1] TRUE
```

Функция `filter()` включает в вывод лишь те строки, для которых условие имеет значение `TRUE`; значения `FALSE` и `NA` исключаются. Если вы хотите сохранить недостающие значения, запросите их в явном виде.

```
df <- tibble(x = c(1, NA, 3))
filter(df, x > 1)
#> # A tibble: 1 × 1
#>       x
#>   <dbl>
#> 1     3
filter(df, is.na(x) | x > 1)
#> # A tibble: 2 × 1
#>       x
#>   <dbl>
#> 1    NA
#> 2     3
```

Упражнения

1. Найдите все авиарейсы, которые:
   а) задержались с прилетом на два и более часа;
   б) вылетали в Хьюстон (аэропорты IAH или HOU);
   в) обслуживались авиакомпаниями United, American или Delta;
   г) вылетали в летнее время (июль, август и сентябрь);
   д) прибывали с опозданием более чем на два часа, но не задерживались с вылетом;
   е) вылетали с задержкой не менее чем на час, но наверстывали более 30 минут во время полета;
   ж) вылетали между полуночью и 6 часами утра (включительно).

2. В dplyr имеется еще одно полезное средство фильтрации — функция `between()`. Что она делает? Можете ли вы использовать ее для упрощения кода, с помощью которого решаются задачи, предложенные в п. 1?

3. Каково количество авиарейсов с отсутствующими данными о времени вылета (`dep_time`)? Назовите другие переменные с отсутствующими значениями. Что могли бы представлять эти строки?

4. Почему `NA ^ 0` не является отсутствующим значением? Почему `NA | TRUE` не является отсутствующим значением? Почему `FALSE & NA` не является отсутствующим значением? Можете ли вы сформулировать общее правило? (Весьма изящный контрпример — `NA * 0`!)

Перестановка строк с помощью функции `arrange()`

Функция `arrange()` работает аналогично функции `filter()`, за исключением того, что вместо отбора строк она изменяет их порядок следования. Аргументами функции служат фрейм данных и набор имен столбцов (или более сложных выражений), по которым осуществляется сортировка. Если вы предоставляете более одного столбца, каждый дополнительный столбец будет использоваться для разрыва связей в значениях предыдущих столбцов.

```
arrange(flights, year, month, day)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
#> # ... with 3.368e+05 more rows, and 13 more variables:
#>   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#>   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#>   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#>   minute <dbl>, time_hour <dttm>
```

Используем функцию `desc()` для сортировки столбца по убыванию.

```
arrange(flights, desc(arr_delay))
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     9      641            900      1301
```

```
#> 2  2013     6    15     1432           1935      1137
#> 3  2013     1    10     1121           1635      1126
#> 4  2013     9    20     1139           1845      1014
#> 5  2013     7    22      845           1600      1005
#> 6  2013     4    10     1100           1900       960
#> # ... with 3.368e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>,
```

В отсортированном наборе отсутствующие значения всегда оказываются последними.

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
#> # A tibble: 3 × 1
#>       x
#>   <dbl>
#> 1     2
#> 2     5
#> 3    NA
arrange(df, desc(x))
#> # A tibble: 3 × 1
#>       x
#>   <dbl>
#> 1     5
#> 2     2
#> 3    NA
```

Упражнения

1. Как бы вы использовали функцию `arrange()`, чтобы поместить отсутствующие значения в начало отсортированного набора? (Подсказка: используйте функцию `is.na()`.)
2. Выполните сортировку авиарейсов для нахождения тех из них, которые характеризуются наибольшей задержкой. Найдите авиарейсы с наиболее ранним временем вылета.
3. Выполните сортировку авиарейсов для нахождения наиболее быстрых полетов.
4. Длительность каких авиарейсов была наибольшей? Длительность каких авиарейсов была наибольшей?

Выбор столбцов с помощью функции select()

Наборы данных, насчитывающие сотни и даже тысячи переменных, не являются редкостью. В подобных случаях первоочередной задачей часто является сужение набора по переменным, которые вас действительно интересуют. Функция select() позволяет быстро сконцентрировать внимание на полезном поднаборе с помощью операций, базирующихся на именах переменных.

Функция select() не будет особенно полезной при анализе набора данных об авиарейсах, насчитывающего всего лишь 19 переменных, но это не помешает вам ухватить общую идею.

```
# Выбор столбцов по имени
select(flights, year, month, day)
#> # A tibble: 336,776 × 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # ... with 3.368e+05 more rows

# Выбор всех столбцов, которые находятся между
# столбцами year и day (включая последние)
select(flights, year:day)
#> # A tibble: 336,776 × 3
#>    year month   day
#>   <int> <int> <int>
#> 1  2013     1     1
#> 2  2013     1     1
#> 3  2013     1     1
#> 4  2013     1     1
#> 5  2013     1     1
#> 6  2013     1     1
#> # ... with 3.368e+05 more rows

# Выбор всех столбцов, за исключением тех, которые
# находятся между столбцами year и day (включая последние)
select(flights, -(year:day))
#> # A tibble: 336,776 × 16
#>   dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>      <int>          <int>     <dbl>    <int>          <int>
#> 1      517            515         2      830            819
```

```
#> 2      533            529         4      850            830
#> 3      542            540         2      923            850
#> 4      544            545        -1     1004           1022
#> 5      554            600        -6      812            837
#> 6      554            558        -4      740            728
#> # ... with 3.368e+05 more rows, and 12 more variables:
#> #   arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>,
#> #   distance <dbl>,  hour <dbl>, minute <dbl>,
#> #   time_hour <dttm>
```

Существует ряд вспомогательных функций, которые можно использовать вместе с функцией `select()`:

- `starts_with("abc")` — соответствует именам, начинающимся с последовательности символов “abc”;
- `ends_with("xyz")` — соответствует именам, заканчивающимся последовательностью символов“xyz”;
- `contains("ijk")` — соответствует именам, содержащим последовательность символов “ijk”;
- `matches("(.)\\1")` — выбирает переменные, соответствующие регулярному выражению. Данному регулярному выражению соответствуют любые переменные, содержащие повторяющиеся символы. Более подробно о регулярных выражениях будет рассказано в главе 11.
- `num_range("x", 1:3)` — соответствует `x1`, `x2` и `x3`.

Для получения более подробной информации выполните команду `?select`.

Несмотря на существование принципиальной возможности переименовывать переменные с помощью функции `select()`, в таком амплуа она используется крайне редко, поскольку этот процесс приводит к выпадению всех переменных, имена которых не были указаны в явном виде. Для этих целей лучше использовать функцию `rename()` — модифицированный вариант функции `select()`, обеспечивающий сохранение всех переменных, не указанных в явном виде.

```
rename(flights, tail_num = tailnum)
#> # A tibble: 336,776 × 19
#>   year month   day dep_time sched_dep_time dep_delay
#>  <int> <int> <int>    <int>          <int>     <dbl>
#> 1 2013     1     1      517            515         2
```

```
#> 2 2013     1     1      533            529         4
#> 3 2013     1     1      542            540         2
#> 4 2013     1     1      544            545        -1
#> 5 2013     1     1      554            600        -6
#> 6 2013     1     1      554            558        -4
#> # ... with 3.368e+05 more rows, and 13 more variables:
#>   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#>   carrier <chr>, flight <int>, tail_num <chr>,
#>   origin <chr>, dest <chr>, air_time <dbl>,
#>   distance <dbl>, hour <dbl>, minute <dbl>,
#>   time_hour <dttm>
```

Для этого есть еще одна возможность — использовать функцию `select()` в сочетании со вспомогательной функцией `everything()`. Это может быть полезным при наличии большого количества переменных, которые вы хотели бы переместить в начало фрейма данных.

```
select(flights, time_hour, air_time, everything())
#> # A tibble: 336,776 × 19
#>              time_hour air_time  year month   day dep_time
#>                 <dttm>    <dbl> <int> <int> <int>    <int>
#> 1 2013-01-01 05:00:00      227  2013     1     1      517
#> 2 2013-01-01 05:00:00      227  2013     1     1      533
#> 3 2013-01-01 05:00:00      160  2013     1     1      542
#> 4 2013-01-01 05:00:00      183  2013     1     1      544
#> 5 2013-01-01 06:00:00      116  2013     1     1      554
#> 6 2013-01-01 05:00:00      150  2013     1     1      554
#> # ... with 3.368e+05 more rows, and 13 more variables:
#>   sched_dep_time <int>, dep_delay <dbl>, arr_time <int>,
#>   sched_arr_time <int>, arr_delay <dbl>, carrier <chr>,
#>   flight <int>, tailnum <chr>, origin <chr>, dest <chr>,
#>   distance <dbl>, hour <dbl>, minute <dbl>
```

Упражнения

1. Укажите как можно больше способов выбора столбцов `dep_time`, `dep_delay`, `arr_time` и `arr_delay` из фрейма данных `flights`.
2. Что произойдет, если при вызове функции `select()` имя какой-либо переменной будет указано несколько раз?
3. Что делает функция `one_of()`? Чем она может быть полезна в отношении приведенного ниже вектора?

```
vars <- c(
   "year", "month", "day", "dep_delay", "arr_delay"
)
```

4. Удивляет ли вас результат выполнения приведенного ниже кода? Как работают вспомогательные функции выбора по умолчанию? Как вы можете изменить это умолчание?

```
select(flights, contains("TIME"))
```

Добавление новых переменных с помощью функции mutate()

Помимо выбора наборов существующих столбцов часто возникает необходимость в добавлении новых столбцов, являющихся функциями существующих. Это обеспечивает функция `mutate()`.

Функция `mutate()` всегда добавляет новые столбцы в конец набора данных, поэтому мы начнем с создания более узкого набора, позволяющего увидеть новые переменные. Не забывайте о том, что при работе в RStudio самый простой способ, обеспечивающий отображение всех столбцов, — это выполнение команды `View()`.

```
flights_sml <- select(flights,
  year:day,
  ends_with("delay"),
  distance,
  air_time
)
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  speed = distance / air_time * 60
)
#> # A tibble: 336,776 × 9
#>    year month   day dep_delay arr_delay distance air_time
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
#> 1  2013     1     1         2        11     1400      227
#> 2  2013     1     1         4        20     1416      227
#> 3  2013     1     1         2        33     1089      160
#> 4  2013     1     1        -1       -18     1576      183
#> 5  2013     1     1        -6       -25      762      116
#> 6  2013     1     1        -4        12      719      150
#> # ... with 3.368e+05 more rows, and 2 more variables:
#> #  gain <dbl>, speed <dbl>
```

Заметьте, что на только что созданные столбцы можно ссылаться.

```
mutate(flights_sml,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
```

```
  gain_per_hour = gain / hours
)
#> # A tibble: 336,776 × 10
#>    year month   day dep_delay arr_delay distance air_time
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>    <dbl>
#> 1  2013     1     1         2        11     1400      227
#> 2  2013     1     1         4        20     1416      227
#> 3  2013     1     1         2        33     1089      160
#> 4  2013     1     1        -1       -18     1576      183
#> 5  2013     1     1        -6       -25      762      116
#> 6  2013     1     1        -4        12      719      150
#> # ... with 3.368e+05 more rows, and 3 more variables:
#>   gain <dbl>, hours <dbl>, gain_per_hour <dbl>
```

Если вам нужно лишь сохранить новые переменные, воспользуйтесь функцией `transmute()`.

```
transmute(flights,
  gain = arr_delay - dep_delay,
  hours = air_time / 60,
  gain_per_hour = gain / hours
)
#> # A tibble: 336,776 × 3
#>    gain hours gain_per_hour
#>   <dbl> <dbl>         <dbl>
#> 1     9  3.78          2.38
#> 2    16  3.78          4.23
#> 3    31  2.67         11.62
#> 4   -17  3.05         -5.57
#> 5   -19  1.93         -9.83
#> 6    16  2.50          6.40
#> # ... with 3.368e+05 more rows
```

Полезные функции создания объектов

Существует много функций, которые предназначены для создания новых переменных, используемых с функцией `mutate()`. Важным моментом является то, что любая такая функция должна быть векторизована: она должна принимать вектор значений и возвращать вектор с тем же количеством значений. Перечислить все подобные функции не представляется возможным, но те из них, которые используются наиболее часто, рассмотрены ниже.

Арифметические операторы +, -, *, /, ^

Все эти операции векторизованы и используют так называемые "правила переписывания" (recycling rules). Если длина одного параметра меньше длины

другого, то он автоматически расширяется, чтобы параметры имели равную длину. Это наиболее полезно в тех случаях, когда один из параметров представляет собой одиночное число: `air_time / 60`, `hours * 60 + minute` и т.п.

Кроме того, арифметические операторы используются в сочетании с агрегирующими функциями, о которых вскоре пойдет речь. Например, выражение `x / sum(x)` вычисляет долю величины относительно общей суммы величин, а `y - mean(y)` — отклонение от среднего.

Модулярная арифметика (`%/%` и `%%`)

К модульным операциям относятся, в частности, операции `%/%` (целочисленное деление) и `%%` (взятие остатка).

```
x == y * (x %/% y) + (x %% y)
```

Модулярная арифметика является весьма удобным средством, поскольку обеспечивает разделение целых чисел на части. Например, в случае набора данных `flights` можно извлечь часы и минуты из столбца `dep_time` с помощью следующей команды.

```
transmute(flights,
  dep_time,
  hour = dep_time %/% 100,
  minute = dep_time %% 100
)
#> # A tibble: 336,776 × 3
#>   dep_time  hour minute
#>      <int> <dbl>  <dbl>
#> 1      517     5     17
#> 2      533     5     33
#> 3      542     5     42
#> 4      544     5     44
#> 5      554     5     54
#> 6      554     5     54
#> # ... with 3.368e+05 more rows
```

Логарифмические функции `log()`, `log2()`, `log10()`

Логарифмические преобразования оказываются чрезвычайно полезными при работе с данными, охватывающими диапазон в несколько порядков по величине. Кроме того, они преобразуют мультипликативные отношения в аддитивные — особенность, к рассмотрению которой мы вернемся в части IV.

При прочих равных условиях я рекомендую использовать функцию `log2()`, поскольку ее результаты легко интерпретировать: разнице значений

в 1 по логарифмической шкале соответствует удвоение значений по исходной шкале, а разнице в -1 — деление пополам.

Смещения

Функции `lead()` и `lag()` позволяют ссылаться на значения, отстоящие от заданного на указанное число позиций в прямом и обратном направлении соответственно. Это позволяет вычислять текущие разности (например, `x - lag(x)`) или определять изменение значений (`x != lag(x)`). Эти функции оказываются наиболее полезными в сочетании с функцией `group_by()`, которая вскоре будет обсуждаться.

```
(x <- 1:10)
#>  [1]  1  2  3  4  5  6  7  8  9 10
lag(x)
#>  [1] NA  1  2  3  4  5  6  7  8  9
lead(x)
#>  [1]  2  3  4  5  6  7  8  9 10 NA
```

Кумулятивные и скользящие агрегаты

Язык R предоставляет функции, предназначенные для вычисления скользящих сумм, произведений, минимальных и максимальных значений: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; кроме того, пакет **dplyr** предоставляет функцию `cummean()` для вычисления скользящих средних. Если вам нужны скользящие агрегаты (сумма, вычисляемая в пределах окна скольжения), обратитесь к пакету **RcppRoll**.

```
x
#>  [1]  1  2  3  4  5  6  7  8  9 10
cumsum(x)
#>  [1]  1  3  6 10 15 21 28 36 45 55
cummean(x)
#>  [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5
```

Логические операторы сравнения <, <=, >, >=, !=

Во многих случаях при выполнении сложных последовательностей логических операций целесообразно сохранять промежуточные значения в новых переменных, что позволяет контролировать правильность выполнения отдельных шагов.

Ранжирование

Существует не одна функция ранжирования, однако вам следует начать с функции `min_rank()`. Она позволяет решать наиболее распространенные задачи ранжирования (например, находить первый, второй, третий, четвертый по величине элементы). По умолчанию наименьшим элементам присваиваются наименьшие ранги; для предоставления наименьших рангов наибольшим значениям используйте функцию `desc(x)`.

```
y <- c(1, 2, 2, NA, 3, 4)
min_rank(y)
#> [1]  1  2  2 NA  4  5
min_rank(desc(y))
#> [1]  5  3  3 NA  2  1
```

Если функция `min_rank()` не позволяет сделать то, что нужно, воспользуйтесь другими аналогичными функциями: `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()` и `ntile()`. Более полную информацию обо всех этих функциях можно получить на страницах справки.

```
row_number(y)
#> [1]  1  2  3 NA  4  5
dense_rank(y)
#> [1]  1  2  2 NA  3  4
percent_rank(y)
#> [1] 0.00 0.25 0.25   NA 0.75 1.00
cume_dist(y)
#> [1] 0.2 0.6 0.6  NA 0.8 1.0
```

Упражнения

1. Значения переменных `dep_time` и `sched_dep_time` в том виде, в котором мы с ними до сих пор работали, удобны для просмотра, но не для вычислений, поскольку они не представляют собой непрерывные числа. Преобразуйте их в более удобное представление в виде количества минут, истекших после полуночи.
2. Сравните переменную `air_time` с результатом вычитания `arr_time - dep_time`. Что вы ожидаете увидеть? Что вы увидели? Что необходимо сделать, чтобы получить правильный результат?
3. Сравните между собой переменные `dep_time`, `sched_dep_time` и `dep_delay`. Как, по вашему мнению, должны соотноситься между собой величины этих переменных?

4. Найдите 10 авиарейсов, характеризующихся наибольшей задержкой, используя функцию ранжирования. Как вы хотите обрабатывать связи? Внимательно прочитайте документацию по функции `min_rank()`.
5. Какой результат возвращает выражение `1:3 + 1:10`? Почему результат именно такой?
6. Какие тригонометрические функции предоставляет язык R?

Получение групповых итогов с помощью функции `summarize()`

Последний из глаголов, который мы рассмотрим, — `summarize()`. Он свертывает фрейм данных в одну строку.

```
summarize(flights, delay = mean(dep_delay, na.rm = TRUE))
#> # A tibble: 1 × 1
#>   delay
#>   <dbl>
#> 1  12.6
```

(К обсуждению смысла параметра `na.rm = TRUE` мы вскоре вернемся.)

Функция `summarize()` не особенно полезна, если используется не в сочетании с функцией `group_by()`. Эта функция позволяет переходить от полного набора данных в качестве единицы анализа к отдельным группам. После этого, если вы используете глаголы пакета **dplyr** в отношении сгруппированного фрейма данных, они автоматически применяются "погруппно". Например, если бы мы применили точно тот же код к фрейму данных, сгруппированному по датам, то получили бы средние времена задержки на каждую дату.

```
by_day <- group_by(flights, year, month, day)
summarize(by_day, delay = mean(dep_delay, na.rm = TRUE))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day delay
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.55
#> 2  2013     1     2 13.86
#> 3  2013     1     3 10.99
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
```

```
#> 6  2013     1     6  7.15
#> # ... with 359 more rows
```

Функции `group_by()` и `summarize()`, взятые вместе, предоставляют один из инструментов, который вы будете использовать сплошь и рядом, работая с пакетом **dplyr**: сгруппированные итоги. Но прежде чем продолжить эту тему, вы должны познакомиться с еще одной мощной идеей: каналами.

Объединение нескольких операций с помощью канала

Предположим, мы хотим исследовать связь между расстоянием и средним временем задержки рейса для всех пунктов приема и вылета (рис. 3.2). Используя все то, что вы знаете о возможностях пакета **dplyr**, вы могли бы написать примерно следующий код.

```
by_dest <- group_by(flights, dest)
delay <- summarize(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE)
  delay = mean(arr_delay, na.rm = TRUE)
)
delay <- filter(delay, count > 20, dest != "HNL")

# Как следует из графика, длительность задержек увеличивается с
# ростом расстояния примерно до 750 миль, а затем уменьшается.
# Не объясняется ли это тем, что с увеличением длительности полета
# появляется больше возможностей для того, чтобы наверстать
# упущенное время?
ggplot(data = delay, mapping = aes(x = dist, y = delay)) +
  geom_point(aes(size = count), alpha = 1/3) +
  geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'loess'
```

Процесс подготовки этих данных можно разбить на три этапа.

1. Группирование авиарейсов по пунктам назначения.
2. Получение сводных данных, касающихся расстояний, среднего времени задержки и количества авиарейсов.
3. Фильтрация данных с целью исключения точек данных, создающих информационный шум, и аэропорта Гонолулу, который располагается на расстоянии, почти вдвое превышающем расстояние до следующего ближайшего аэропорта.

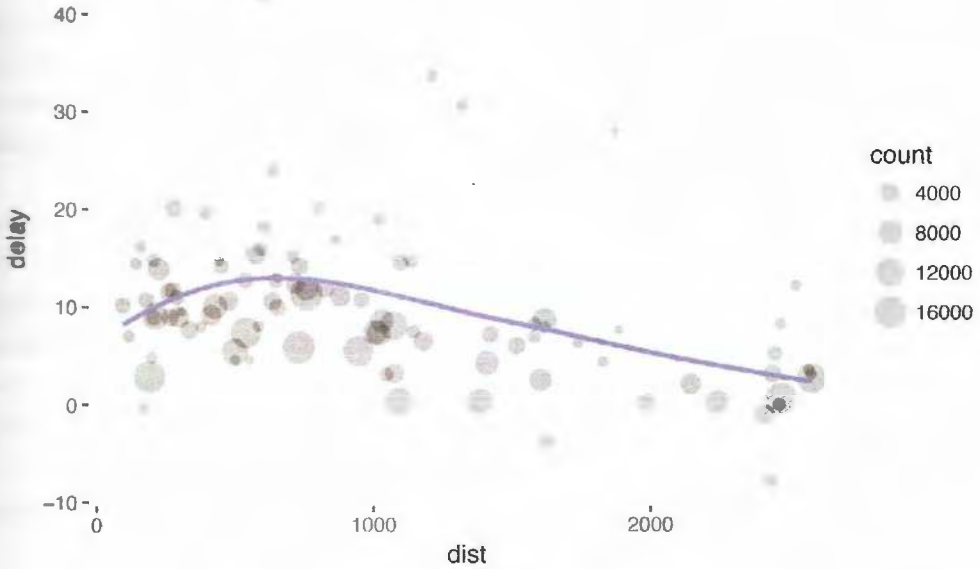


Рис. 3.2

Писать такой код не особенно приятно, поскольку каждому промежуточному фрейму данных необходимо присвоить отдельное имя, даже если этот фрейм для нас не представляет интереса. Процедура именования довольно трудоемкая, и это может замедлить анализ.

Однако существует другая возможность справиться с обозначенной проблемой — использовать канал `%>%`.

```
delays <- flights %>%
  group_by(dest) %>%
  summarize(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  %>%
filter(count > 20, dest != "HNL")
```

Здесь внимание фокусируется на преобразованиях, а не на том, что преобразовывается, благодаря чему читать такой код становится легче. Можете читать его как последовательность повелительных предложений: сгруппируй, затем просуммируй, после чего отфильтруй. Как подсказывает только что приведенная формулировка, смысл выражения `%>%` неплохо представлять себе как “затем”.

За кулисами выражение `x %>% f(y)` превращается в `f(x, y)`, а выражение `x %>% f(y) %>% g(z)` — в `g(f(x, y), z)` и т.д. Канал можно использовать для переписывания последовательности нескольких операторов таким образом, чтобы их можно было читать слева направо и сверху вниз. С этого момента мы будем часто использовать каналы, поскольку они значительно улучшают читаемость кода, и мы продолжим их обсуждение в главе 14.

Использование каналов — один из ключевых аспектов работы с библиотекой *tidyverse*. Единственным исключением является пакет **ggplot2**: он был написан еще до того, как появились каналы. К сожалению, время для следующего поколения **ggplot2**, пакета **ggvis**, позволяющего использовать каналы, еще не наступило.

Отсутствующие значения

Возможно, вы обратили внимание на аргумент `na.rm`, который мы перед этим использовали. Что случится, если его не установить?

```
flights %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1    NA
#> 2  2013     1     2    NA
#> 3  2013     1     3    NA
#> 4  2013     1     4    NA
#> 5  2013     1     5    NA
#> 6  2013     1     6    NA
#> # ... with 359 more rows
```

Мы получили множество отсутствующих значений! Это произошло потому, что агрегирующие функции подчиняются обычному правилу, касающемуся отсутствующих значений: если среди входных данных есть отсутствующие значения, выходной результат также будет отсутствующим значением. К счастью, все агрегирующие функции могут принимать аргумент `na.rm`, который удаляет отсутствующие значения еще до выполнения вычислений.

```
flights %>%
group_by(year, month, day) %>%
summarize(mean = mean(dep_delay, na.rm = TRUE))
#> Source: local data frame [365 x 4]
```

```
#> Groups: year, month [?]
#>
#>    year month   day  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.55
#> 2  2013     1     2 13.86
#> 3  2013     1     3 10.99
#> 4  2013     1     4  8.95
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> # ... with 359 more rows
```

В данном случае, когда пропущенные значения представляют отмененные авиарейсы, мы могли бы справиться с проблемой, предварительно удалив такие значения. Мы сохраним этот набор данных для того, чтобы его можно было повторно использовать в последующих примерах.

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day  mean
#>   <int> <int> <int> <dbl>
#> 1  2013     1     1 11.44
#> 2  2013     1     2 13.68
#> 3  2013     1     3 10.91
#> 4  2013     1     4  8.97
#> 5  2013     1     5  5.73
#> 6  2013     1     6  7.15
#> #   ...  with   359  more
```

Счетчики

Каждый раз, когда вы агрегируете данные, целесообразно включать в них счетчик наблюдений в текущей группе (`n()`) или счетчик отсутствующих значений (`sum(!is.na(x))`). Это хороший способ контроля того, что ваши заключения базируются на достаточно большом количестве данных. Предположим, например, что нас интересуют самолеты (идентифицируемые по их бортовым номерам) с наибольшим средним временем задержки (рис. 3.3).

```
delays <- not_cancelled %>%
  group_by(tailnum)%>%
  summarize(
    delay = mean(arr_delay)
  )

ggplot(data = delays, mapping = aes(x = delay)) +
  geom_freqpoly(binwidth = 10)
```

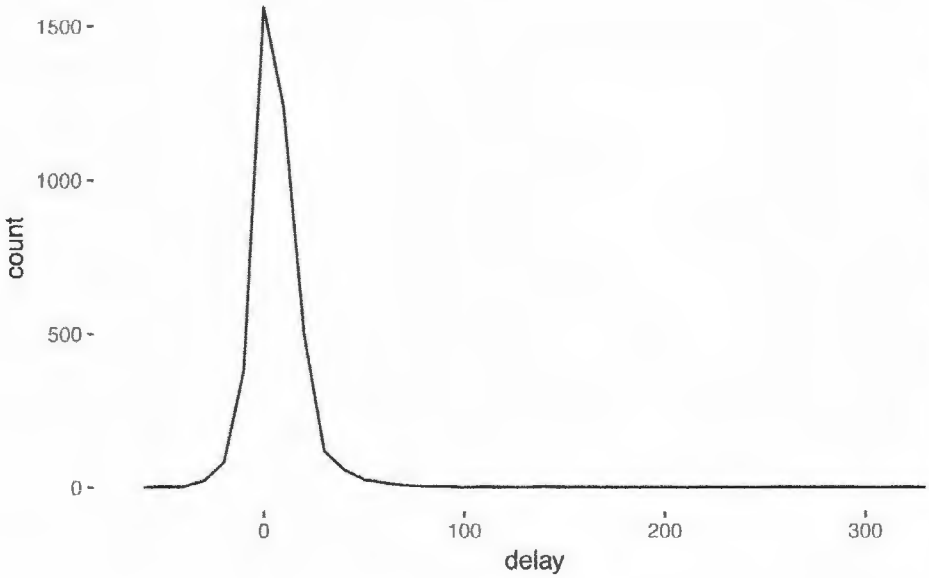


Рис. 3.3

Ничего себе! Оказывается, для некоторых самолетов среднее время задержки составило целых 5 часов (300 минут)!

На самом деле в этой истории есть свои нюансы. Мы можем рассмотреть эту ситуацию под другим ракурсом, построив диаграмму рассеяния (точечную диаграмму), по осям которой откладываются количество авиарейсов и среднее время задержки (рис. 3.4).

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )
```

```
ggplot(data = delays, mapping = aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```

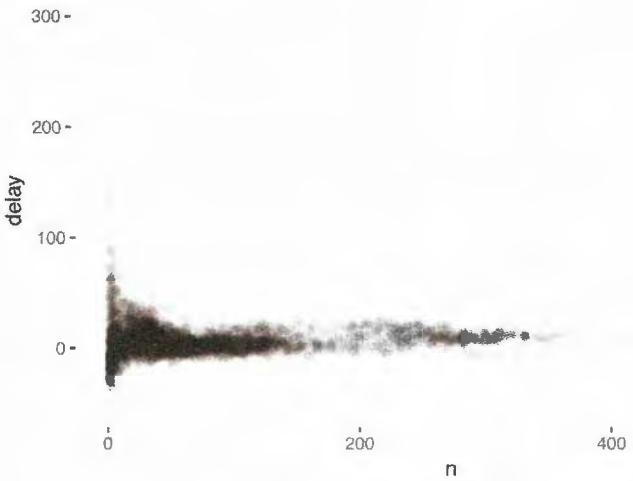


Рис. 3.4

В том, что намного больший разброс среднего времени задержки наблюдается для меньшего количества авиарейсов, нет ничего удивительного. Вид этой диаграммы весьма характерен: всякий раз, когда вы будете отображать на графике зависимость среднего (или другой итоговой характеристики) от размера группы, разброс данных уменьшается с увеличением размера выборки.

При анализе диаграмм подобного рода часто имеет смысл отфильтровать группы с наименьшим количеством наблюдений. Исключение предельных вариаций в наименьших группах обеспечивает более эффективное выявление характерных закономерностей. Именно такой подход реализует приведенный ниже код, который также демонстрирует удобный образец интеграции пакета **ggplot2** в потоки операций **dplyr** (рис. 3.5). Несколько неудобно то, что это заставляет переключаться с оператора `%>%` на оператор `+`, но, поработав некоторое время, вы к нему привыкнете.

```
delays %>%
  filter(n > 25) %>%
  ggplot(mapping = aes(x = n, y = delay)) +
    geom_point(alpha = 1/10)
```

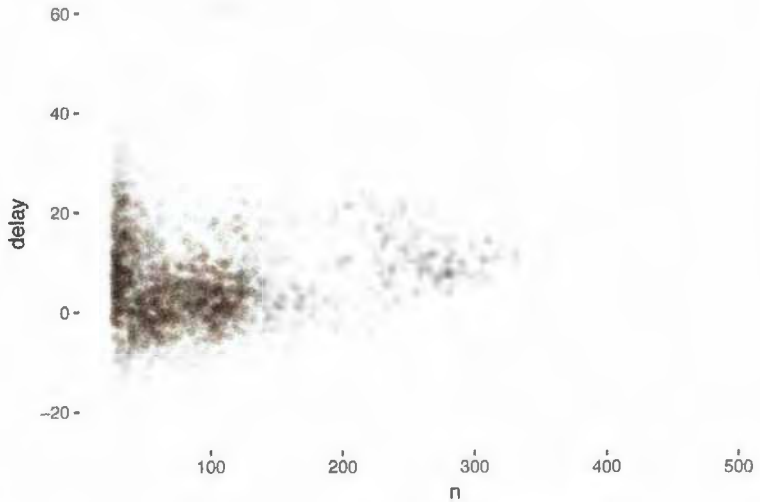


Рис. 3.5

В RStudio предусмотрена полезная комбинация клавиш <Ctrl+Shift+P>. Она пересылает ранее переданный фрагмент кода из окна редактора в окно консоли. Это очень удобно при исследовании влияния мелких изменений в коде. Например, в данном случае вы можете передать в окно консоли весь блок кода с помощью комбинации клавиш <Ctrl+Enter>, а затем изменить значение `n` и нажать комбинацию клавиш <Ctrl+Shift+P> для повторной пересылки всего блока.

Существует другая часто встречающаяся разновидность закономерности этого типа. Рассмотрим связь между средней результативностью отбивающих в бейсболе и количеством их выходов на удар. Для вычисления среднего коэффициента результативности отбиваний (количество успешных ударов / количество попыток) каждого бейсболиста Главной лиги я использую данные из пакета **Lahman**.

Если представить на графике зависимость уровня мастерства отбивающего (измеряемого средним коэффициентом результативности, `ba`) от количества возможностей выполнения удара по мячу (измеряемого количеством выходов на удар, `ab`), то можно заметить следующие две закономерности (рис. 3.6).

- Как и прежде, разброс значений агрегированного показателя уменьшается по мере увеличения количества точек данных.

- Между уровнем мастерства бейсболиста (ba) и количеством возможностей выполнить удар (ab) существует положительная корреляция. Это происходит потому, что команда контролирует, кто должен вступать в игру, и вполне естественно, что обычно выбирают лучших игроков.

```
# Преобразование фрейма в tibble-фрейм для получения
# красивого вывода
batting <- as_tibble(Lahman::Batting)

batters <- batting %>%
  group_by(playerID) %>%
  summarize(
    ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
    ab = sum(AB, na.rm = TRUE)
  )

batters %>%
  filter(ab > 100) %>%
  ggplot(mapping = aes(x = ab, y = ba)) +
    geom_point() +
    geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'gam'
```

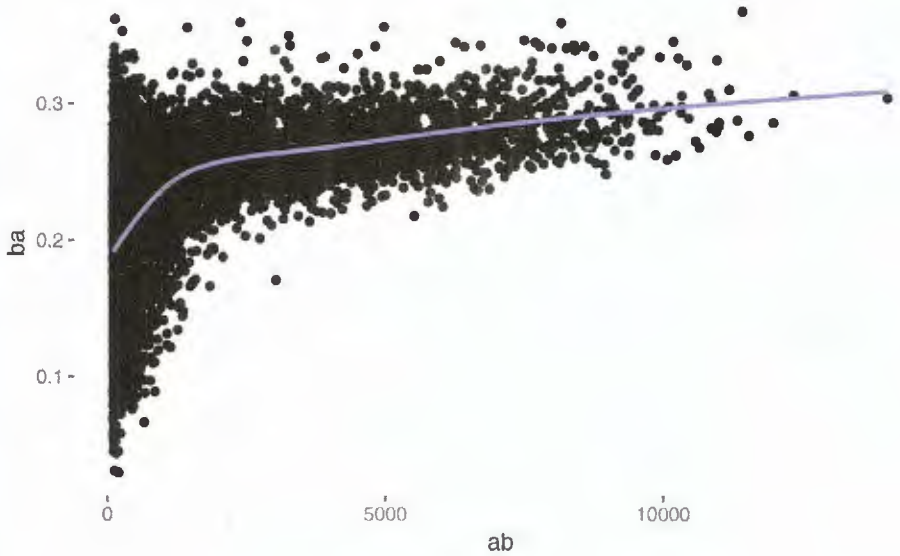


Рис. 3.6

Это также имеет важные последствия для ранжирования. Если вы выполните сортировку ba по убыванию с помощью функции desc(), то увидите,

что список игроков, имеющих наилучший средний коэффициент результативности, возглавляют вовсе не мастера, а игроки, которым просто повезло в том небольшом количестве попыток отбивания мяча, которое им было предоставлено.

```
batters %>%
arrange(desc(ba))
#> # A tibble: 18,659 × 3
#>    playerID    ba    ab
#>       <chr> <dbl> <int>
#> 1 abramge01     1     1
#> 2 banisje01     1     1
#> 3 bartocl01     1     1
#> 4  bassdo01     1     1
#> 5 birasst01     1     2
#> 6 bruneju01     1     1
#> # ... with 1.865e+04 more rows
```

Неплохое обсуждение этой проблемы вы найдете на следующих сайтах: http://bit.ly/Bayesbbal и http://bit.ly/notsortavg.

Полезные итоговые функции

Уже одно только использование средних, счетчиков и сумм дает довольно много информации, но R предоставляет множество других полезных итоговых (агрегатных) функций.

Меры положения

Мы уже использовали функцию `mean(x)`, однако функция `median(x)` также полезна. Среднее — это сумма, деленная на количество наблюдений; медиана — это такое значение, что 50% значений x расположено ниже его, а другие 50% — выше него.

Иногда агрегирование полезно сочетать с логическим выделением поднаборов. Мы пока что не обсуждали эту разновидность выделения поднаборов, однако данный вопрос подробно обсуждается в разделе “Выделение поднаборов” главы 16.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    # средняя задержка
    avg_delay1 = mean(arr_delay),
    # средняя положительная задержка
    avg_delay2 = mean(arr_delay[arr_delay > 0])
```

```
#> Source: local data frame [365 x 5]
#> Groups: year, month [?]
#>
#>    year month   day avg_delay1 avg_delay2
#>   <int> <int> <int>      <dbl>      <dbl>
#> 1  2013     1     1      12.65       32.5
#> 2  2013     1     2      12.69       32.0
#> 3  2013     1     3       5.73       27.7
#> 4  2013     1     4      -1.93       28.3
#> 5  2013     1     5      -1.53       22.6
#> 6  2013     1     6       4.24       24.4
#> # ... with 359 more rows
```

Меры разброса: `sd(x)`, `IQR(x)`, `mad(x)`

Стандартной мерой разброса значений является среднеквадратическое отклонение, вычисляемое с помощью функции `sd()`. Его надежные эквиваленты — межквартильный размах (функция `IQR()`) и медианное абсолютное отклонение (функция `mad()`) — могут быть более полезны при наличии выбросов.

```
# Почему одни расстояния до пунктов назначения
# более изменчивы, чем другие?
not_cancelled %>%
  group_by(dest) %>%
  summarize(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))
#> # A tibble: 104 × 2
#>    dest distance_sd
#>   <chr>       <dbl>
#> 1   EGE       10.54
#> 2   SAN       10.35
#> 3   SFO       10.22
#> 4   HNL       10.00
#> 5   SEA        9.98
#> 6   LAS        9.91
#> # ... with 98 more rows
```

Меры ранжирования: `min(x)`, `quantile(x, 0.25)`, `max(x)`

Квантили — это обобщение медианы. Например, функция `quantile(x, 0.25)` найдет значение *x*, которое превышает 25% значений и меньше оставшихся 75% значений.

```
# Найти время вылета первого и последнего авиарейсов
# для каждого дня в году
```

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    first = min(dep_time),
    last = max(dep_time)
  )
#> Source: local data frame [365 x 5]
#> Groups: year, month [?]
#>
#>    year month   day first  last
#>   <int> <int> <int> <int> <int>
#> 1  2013     1     1   517  2356
#> 2  2013     1     2    42  2354
#> 3  2013     1     3    32  2349
#> 4  2013     1     4    25  2358
#> 5  2013     1     5    14  2357
#> 6  2013     1     6    16  2355
#> # ... with 359 more rows
```

Порядковые меры: `first(x)`, `nth(x, 2)`, `last(x)`

Эти функции работают аналогично выражениям `x[1]`, `x[2]` и `x[length(x)]`, но позволяют задать значение по умолчанию, если указанной позиции не существует (скажем, если вы пытаетесь получить третий элемент группы, насчитывающей только два элемента). Например, мы можем найти время вылета первого и последнего авиарейсов для каждого дня в году.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    first_dep = first(dep_time),
    last_dep = last(dep_time)
  )
#> Source: local data frame [365 x 5]
#> Groups: year, month [?]
#>
#>    year month    day first_dep last_dep
#>   <int> <int>  <int>     <int>    <int>
#> 1  2013     1      1       517     2356
#> 2  2013     1      2        42     2354
#> 3  2013     1      3        32     2349
#> 4  2013     1      4        25     2358
#> 5  2013     1      5        14     2357
#> 6  2013     1      6        16     2355
#> # ... with 359 more rows
```

Эти функции являются комплементарными по отношению к фильтрации по рангу. С помощью фильтра вы получаете все переменные, причем каждое наблюдение располагается в отдельной строке.

```
not_cancelled %>%
  group_by(year, month, day) %>%
  mutate(r = min_rank(desc(dep_time))) %>%
  filter(r %in% range(r))
#> Source: local data frame [770 x 20]
#> Groups: year, month, day [365]
#>
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1     2356           2359        -3
#> 3  2013     1     2       42           2359        43
#> 4  2013     1     2     2354           2359        -5
#> 5  2013     1     3       32           2359        33
#> 6  2013     1     3     2349           2359       -10
#> # ... with 764 more rows, and 13 more variables:
#>   arr_time <int>, sched_arr_time <int>,
#>   arr_delay <dbl>, carrier <chr>, flight <int>,
#>   tailnum <chr>, origin <chr>, dest <chr>,
#>   air_time <dbl>, distance <dbl>, hour <dbl>,
#>   minute <dbl>, time_hour <dttm>, r <int>
```

Счетчики

Ранее вы уже познакомились с функцией `n()`, которая не принимает аргументов и возвращает размер текущей группы. Для подсчета количества значений, не являющихся отсутствующими, используйте функцию `sum(!is.na(x))`. Для подсчета количества различных (уникальных) значений используйте функцию `n_distinct(x)`.

```
# Какие пункты назначения имеют наибольшее количество
# перевозчиков?
not_cancelled %>%
  group_by(dest) %>%
  summarize(carriers = n_distinct(carrier)) %>%
  arrange(desc(carriers))
#> # A tibble: 104 × 2
#>    dest carriers
#>   <chr>    <int>
#> 1   ATL        7
#> 2   BOS        7
#> 3   CLT        7
#> 4   ORD        7
```

```
#> 5   TPA        7
#> 6   AUS        6
#> # ... with 98 more rows
```

Счетчики настолько полезны, что пакет dplyr даже предоставляет простую вспомогательную функцию на тот случай, если все, что вам нужно, — только подсчитать значения.

```
not_cancelled %>%
  count(dest)
#> # A tibble: 104 × 2
#>    dest     n
#>   <chr> <int>
#> 1   ABQ   254
#> 2   ACK   264
#> 3   ALB   418
#> 4   ANC     8
#> 5   ATL 16837
#> 6   AUS  2411
#> # ... with 98 more rows
```

Вы можете предоставить функции `count()` необязательный весовой коэффициент. Например, можно использовать эту возможность для "подсчета" (суммирования) общего количества миль, которое налетало воздушное судно.

```
not_cancelled %>%
  count(tailnum, wt = distance)
#> # A tibble: 4,037 × 2
#>   tailnum      n
#>     <chr>  <dbl>
#> 1  D942DN   3418
#> 2  N0EGMQ 239143
#> 3  N10156 109664
#> 4  N102UW  25722
#> 5  N103US  24619
#> 6  N104UW  24616
#> # ... with 4,031 more rows
```

Подсчет количества и долей логических значений: `sum(x > 10)`, `mean(y == 0)`

Когда они используются с числовыми функциями, значение `TRUE` преобразуется в 1, а значение `FALSE` — в 0. Это делает функции `sum()` и `mean()` весьма полезными: `sum(x)` дает количество значений `TRUE` в `x`, а `mean(x)` — их относительную долю.

```
# Сколько авиарейсов было отправлено до 5 часов утра?
# (Обычно это задержанные авиарейсы с прошлого дня.)
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(n_early = sum(dep_time < 500))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day n_early
#>   <int> <int> <int>   <int>
#> 1  2013     1     1       0
#> 2  2013     1     2       3
#> 3  2013     1     3       4
#> 4  2013     1     4       3
#> 5  2013     1     5       3
#> 6  2013     1     6       2
#> # ... with 359 more rows

# Какова доля авиарейсов, задержанных более чем на один час?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(hour_perc = mean(arr_delay > 60))
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month   day hour_perc
#>   <int> <int> <int>     <dbl>
#> 1  2013     1     1    0.0722
#> 2  2013     1     2    0.0851
#> 3  2013     1     3    0.0567
#> 4  2013     1     4    0.0396
#> 5  2013     1     5    0.0349
#> 6  2013     1     6    0.0470
#> # ... with 359 more rows
```

Группирование по нескольким переменным

Когда вы группируете набор данных по нескольким переменным, каждый сформированный итог уменьшает количество уровней группировки на единицу. Это упрощает последовательное свертывание набора данных.

```
daily <- group_by(flights, year, month, day)
per_day   <- summarize(daily, flights = n())
#> Source: local data frame [365 x 4]
#> Groups: year, month [?]
#>
#>    year month    day flights
```

```
#>   <int> <int>  <int>   <int>
#> 1  2013     1      1     842
#> 2  2013     1      2     943
#> 3  2013     1      3     914
#> 4  2013     1      4     915
#> 5  2013     1      5     720
#> 6  2013     1      6     832
#> # ... with 359 more rows

(per_month <- summarize(per_day, flights = sum(flights)))
#> Source: local data frame [12 x 3]
#> Groups: year [?]
#>
#>    year month flights
#>   <int> <int>   <int>
#> 1  2013     1   27004
#> 2  2013     2   24951
#> 3  2013     3   28834
#> 4  2013     4   28330
#> 5  2013     5   28796
#> 6  2013     6   28243
#> # ... with 6 more rows

(per_year <- summarize(per_month, flights = sum(flights)))
#> # A tibble: 1 × 2
#>    year flights
#>   <int>   <int>
#> 1  2013  336776
```

Будьте особенно внимательны при последовательном свертывании итогов: эта операция безопасна в случае суммирования и подсчета, но вам следует продумывать все, что связано с вычислением взвешенных средних и дисперсий, а кроме того, это невозможно точно сделать в случае таких основанных на ранжировании статистик, как медиана. Другими словами, сумма групповых сумм равна общей сумме, но медиана групповых медиан не равна общей медиане.

Разгруппирование

Если вам необходимо устранить группирование и вернуться к операциям над несгруппированными данными, используйте функцию `ungroup()`.

```
daily %>%
  ungroup() %>%              # больше не группировать по дате
  summarize(flights = n())   # все рейсы
#> # A tibble: 1 × 1
```

```
#>   flights
#>     <int>
#> 1  336776
```

Упражнения

1. Предложите не менее пяти способов доступа к типичным характеристикам времени задержки для групп авиарейсов. Рассмотрите следующие сценарии:
   - авиарейсы, опережающие график на 15 минут в 50% случаев, и авиарейсы, отстающие от графика на 15 минут в 50% случаев;
   - авиарейсы, которые постоянно опаздывают на 10 минут;
   - авиарейсы, опережающие график на 30 минут в 50% случаев, и авиарейсы, отстающие от графика на 30 минут в 50% случаев;
   - авиарейсы, которые в 99% случаев укладывались в график и в 1% случаев опаздывали на 2 часа.

   Какой фактор является более важным: задержка прибытия или задержка отправки авиарейса?
2. Предложите другой подход, обеспечивающий тот же вывод, что и следующий код: `not_cancelled %>% count(dest)` и `not_cancelled %>% count(tailnum, wt = distance)` (без использования функции `count()`).
3. Наше определение отмененных авиарейсов (`is.na(dep_delay) | is.na(arr_delay)`) лишь близко к оптимальному. Почему? Какой столбец является самым важным?
4. Найдите количество ежедневно отменяемых авиарейсов. Удалось ли вам заметить какие-либо закономерности? Связана ли доля отмененных авиарейсов со средним временем задержки?
5. Какой перевозчик чаще других допускал задержки? Задача: можете ли вы разделить эффекты плохих аэропортов и плохих перевозчиков? Почему это возможно (или невозможно)? Подсказка: поразмышляйте над следующим кодом:

   ```
   flights %>% group_by(carrier, dest) %>% summarize(n())
   ```
6. Подсчитайте для каждого воздушного судна количество авиарейсов, совершенных до первой задержки более чем на 1 час.

7. Каково назначение аргумента `sort` функции `count()`? Когда вам может понадобиться использовать его?

Групповое видоизменение (и фильтрация)

Группирование наиболее полезно применять в сочетании с функцией `summarize()`, но также существуют операции, в которых его удобно применять совместно с функциями `mutate()` и `filter()`.

- Найти члены каждой группы, имеющие наихудшие показатели.

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
#> Source: local data frame [3,306 x 7]
#> Groups: year, month, day [365]
#>
#>    year month   day dep_delay arr_delay distance
#>   <int> <int> <int>     <dbl>     <dbl>    <dbl>
#> 1  2013     1     1       853       851      184
#> 2  2013     1     1       290       338     1134
#> 3  2013     1     1       260       263      266
#> 4  2013     1     1       157       174      213
#> 5  2013     1     1       216       222      708
#> 6  2013     1     1       255       250      589
#> # ... with 3,300 more rows, and 1 more variables:
#> #  air_time <dbl>
```

- Найти все группы, которым соответствует превышение порогового значения.

```
popular_dests <- flights %>%
  group_by(dest) %>%
  filter(n() > 365)
popular_dests
#> Source: local data frame [332,577 x 19]
#> Groups: dest [77]
#>
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      517            515         2
#> 2  2013     1     1      533            529         4
#> 3  2013     1     1      542            540         2
#> 4  2013     1     1      544            545        -1
#> 5  2013     1     1      554            600        -6
#> 6  2013     1     1      554            558        -4
```

```
#> # ... with 3.326e+05 more rows, and 13 more variables:
#> #   arr_time <int>, sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>,
#> #   tailnum <chr>, origin <chr>, dest <chr>,
#> #   air_time <dbl>, distance <dbl>, hour <dbl>,
#> #   minute <dbl>, time_hour <dttm>
```

- Стандартизовать для вычисления метрик групп.

```
popular_dests %>%
  filter(arr_delay > 0) %>%
  mutate(prop_delay = arr_delay / sum(arr_delay)) %>%
  select(year:day, dest, arr_delay, prop_delay)
#> Source: local data frame [131,106 x 6]
#> Groups: dest [77]
#>
#>    year month   day  dest arr_delay prop_delay
#>   <int> <int> <int> <chr>     <dbl>      <dbl>
#> 1  2013     1     1   IAH        11   1.11e-04
#> 2  2013     1     1   IAH        20   2.01e-04
#> 3  2013     1     1   MIA        33   2.35e-04
#> 4  2013     1     1   ORD        12   4.24e-05
#> 5  2013     1     1   FLL        19   9.38e-05
#> 6  2013     1     1   ORD         8   2.83e-05
#> # ... with 1.311e+05 more rows
```

Групповая фильтрация — это то же самое, что групповое изменение, вслед за которым выполняется обычная фильтрация. Обычно я избегаю использовать данную операцию, за исключением случаев, когда требуется выполнить на скорую руку какие-либо манипуляции: иначе трудно проверить, что манипуляции были выполнены правильно.

Для совместного использования с групповыми видоизменениями и фильтрами наиболее естественным образом подходят так называемые оконные функции (в отличие от агрегирующих функций, используемых для получения итоговых данных). Для получения более подробной информации о полезных оконных функциях выполните команду `vignette("window-functions")`.

Упражнения

1. Просмотрите предыдущий материал, посвященный функциям, используемым совместно с функциями `mutate()` и `filter()`. Опишите, как изменяется каждая операция, если она выполняется в сочетании с группированием.

2. Какое воздушное судно (`tailnum`) имеет наихудшие показатели соблюдения графика?
3. Какое время вы выбрали бы для вылета как наиболее благоприятное для того, чтобы избежать задержки?
4. Рассчитайте суммарную длительность задержек в минутах для каждого пункта назначения. Рассчитайте для каждого авиарейса относительную долю его суммарного времени задержек с прибытием в пункт назначения.
5. Для задержек характерно наличие корреляций во времени: даже после устранения проблемы, вызвавшей первоначальную задержку, более поздние рейсы задерживаются для того, чтобы предоставить возможность вылететь более ранним. Функция `lag()` позволяет исследовать связь между задержкой конкретного авиарейса и задержкой авиарейса, который был отправлен непосредственно перед ним.
6. Можете ли вы определить для каждого пункта назначения авиарейсы с подозрительно малым временем полета? (Это может указывать на потенциальные ошибки, допущенные при вводе данных.) Рассчитайте время пребывания в воздухе авиарейсов и выясните, какие из них характеризуются наибольшей задержкой во время полета?
7. Найдите все пункты назначения, которые обслуживаются по крайней мере двумя авиаперевозчиками. Используйте эту информацию для ранжирования перевозчиков.

ГЛАВА 4

Организация рабочего процесса: скрипты

В предыдущих главах вы выполняли код в окне консоли. Для начала это было удобно, но как только вы приступите к работе с более сложными диаграммами ggplot2 и каналами dplyr, то вскоре убедитесь, что это доставляет большие неудобства. Использование окна редактора скриптов (сценариев) позволяет значительно расширить размеры доступного рабочего пространства. Чтобы открыть редактор скриптов, щелкните на пункте меню File (Файл), а затем последовательно выберите пункты New File (Создать файл) и R script (R-скрипт), либо воспользуйтесь комбинацией клавиш <Ctrl+Shift+N>. Теперь в вашем распоряжении будут четыре панели (рис. 4.1).

Редактор скриптов предоставляет великолепные возможности для работы с интересующим вас кодом. Продолжайте использовать консоль для того, чтобы экспериментировать с кодом, но как только вы получите вариант кода, который делает то, что вам нужно, переместите его в редактор скриптов. RStudio автоматически сохраняет его содержимое при завершении рабочего сеанса и автоматически загружает его при повторном открытии RStudio. Тем не менее будет неплохо, если вы выработаете привычку регулярно сохранять свои скрипты и создавать их резервные копии.

Рис. 4.1

Выполнение кода

Редактор скриптов отлично подходит для создания сложных диаграмм пакета **ggplot2** или длинных последовательностей команд пакета **dplyr**. Для эффективного использования редактора скриптов вам нужно запомнить одну из наиболее важных комбинаций клавиш: <Ctrl+Enter>. Она обеспечивает выполнение в окне консоли текущего R-выражения. Возьмем, например, приведенный ниже код. Если курсор находится в позиции ▌, нажатие клавиш <Ctrl+Enter> приведет к запуску полной команды `not_cancelled`. При этом курсор переместится к следующей инструкции (начинающейся с выражения `not_cancelled %>%`). Эта особенность упрощает выполнение всего сценария путем повторного нажатия комбинации клавиш <Ctrl+Enter>.

```
library(dplyr)
library(nycflights13)
```

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(mean = mean(dep_delay))
```

Вместо того чтобы последовательно выполнять одну инструкцию за другой, можно выполнить весь сценарий за один раз. Для этого следует нажать комбинацию клавиш <Ctrl+Shift+S>. Это отличный способ контроля того, что в сценарий включены все важные компоненты вашего кода.

Мы рекомендуем всегда начинать свой сценарий с указания необходимых пакетов. Благодаря этому люди, с которыми вы поделитесь своим кодом, сразу же увидят, какие пакеты им необходимо установить. Однако никогда не включайте в код совместно используемого скрипта инструкции наподобие `install.packages()` или `setwd()`. Принудительное изменение настроек на чужом компьютере недопустимо!

Я настоятельно рекомендую, чтобы работу с кодом, приведенным в последующих главах, вы всегда начинали в окне редактора скриптов и практиковались в использовании клавиатурных сокращений. Со временем отправка кода на консоль описанным способом станет для вас настолько естественной, что вы даже не будете об этом задумываться.

Диагностические средства RStudio

Редактор скриптов выделяет синтаксические ошибки волнистой красной линией и помечает соответствующие строки крестиком слева (рис. 4.2).

```
3
4  x y <- 10
5
```

Рис. 4.2

Наведя указатель мыши на этот крестик, вы получите подсказку относительно возможных причин возникновения ошибки (рис. 4.3).

```
4  x y <- 10
5
unexpected token 'y'
unexpected token '<-'
```

Рис. 4.3

RStudio также информирует вас о наличии потенциальных проблем (рис. 4.4).

```
! 17   3 == NA
  1 use 'is.na' to check whether expression evaluates to
  1  NA
  20
```

Рис. 4.4

Упражнения

1. Посетите микроблог RStudio Tips в Твиттере (`@rstudiotips`), содержащий советы по использованию RStudio, и выберите тот из них, который вас больше всего заинтересовал. Попрактикуйтесь в его использовании!
2. О каких других распространенных видах ошибок информирует RStudio в своих диагностических сообщениях? Чтобы выяснить это, перейдите по адресу `http://bit.ly/RStudiocodediag`.

ГЛАВА 5

Предварительный анализ данных

Введение

В этой главе будет показано, как организовать предварительный анализ данных (ПАД) путем их визуализации и преобразования. По своей сути ПАД — итеративный процесс, включающий следующие стадии:

1. формулирование вопросов относительно данных;
2. поиск ответов на поставленные вопросы путем визуализации, преобразования и моделирования данных;
3. использование полученных знаний для уточнения вопросов и (или) формулировки новых вопросов.

ПАД не относится к числу процессов с набором строго установленных правил. Скорее всего, ПАД — это способ мышления. На начальных стадиях ПАД вы можете свободно исследовать любую идею, которая приходит вам на ум. Одни из этих идей будут удачными, другие — тупиковыми. По мере продвижения исследования вы, отшлифовав наиболее плодотворные идеи, выпишете их и поделитесь с другими людьми.

ПАД — важная часть любого анализа данных, даже если вопросы будут преподнесены вам на блюдечке, поскольку вы всегда должны убедиться в качестве данных. Очистка данных является всего лишь одним из аспектов ПАД: вы ставите вопросы, касающиеся того, соответствуют или не соответствуют данные вашим ожиданиям. Для выполнения очистки данных вы должны развернуть все средства ПАД: визуализацию, преобразования и моделирование.

Необходимые ресурсы

В этой главе мы объединим все то, что вы узнали о возможностях пакетов **dplyr** и **ggplot2**, и используем эти знания для интерактивной постановки

вопросов, получения на них ответов на основе имеющихся данных и дальнейшей постановки новых вопросов.

```
library(tidyverse)
```

Вопросы

"В области статистики нет стандартных вопросов, есть лишь стандарты, вызывающие вопросы".

Сэр Дэвид Кокс

"Приблизительный ответ на правильный вопрос имеет гораздо большую ценность, чем точный ответ на неправильный вопрос".

Джон Тьюки

Ваша главная цель при выполнении ПАД — добиться понимания имеющихся данных. Простейшим способом достижения этой цели является постановка вопросов, получение ответов на которые помогает определить направление исследования. В процессе формулировки вопроса ваше внимание фокусируется на конкретной части набора данных, что облегчает принятие решений относительно того, какие диаграммы, модели или преобразования должны быть привлечены в первую очередь.

В своей основе ПАД — творческий процесс. Как и в любом творческом процессе, важное значение при выработке *качественных* вопросов имеет количество самих вопросов. Сформулировать ключевые вопросы, когда вы только приступаете к анализу, трудно, поскольку вам еще не известно, какие закономерности скрыты в данных. С другой стороны, каждый новый вопрос позволяет высветлять новые аспекты данных и увеличивает ваши шансы обнаружения в них новых свойств. Вы сможете быстро исследовать наиболее интересные части данных подробным образом (и сформулировать при этом ряд вопросов, пробуждающих в вас свежие мысли), если за каждым вопросом будете ставить новые, исходя из того, что вы уже успели узнать.

Относительно того, какие вопросы вы должны ставить при проведении такого исследования, не существует общих правил. Однако есть два типа вопросов, ответы на которые всегда будут полезными для выявления закономерностей, скрытых в данных. В широкой формулировке они звучат так.

1. Вариация какого типа существует среди переменных?
2. Ковариация какого типа существует между переменными?

Эти два вопроса будут в центре нашего внимания в оставшейся части главы. Мы объясним, что означают термины *вариация* и *ковариация*, и покажем несколько способов получения ответов на поставленные выше вопросы. Чтобы упростить обсуждение, определимся с некоторыми терминами.

- *Переменная* — это количество, качество или свойство, которые можно измерить.
- *Значение* — это состояние переменной в момент ее измерения. Значение переменной может изменяться от измерения к измерению.
- *Наблюдение* или *случай* — это набор измерений, выполненный в одинаковых условиях (обычно все измерения, входящие в наблюдение, делаются в одно и то же время для одного и того же объекта). Наблюдение содержит несколько значений, каждое из которых связано с отдельной переменной. Иногда вместо термина *наблюдение* мы будем пользоваться термином *точка данных*.
- *Табличные данные* — это набор значений, каждое из которых связано с переменной и наблюдением. Табличные данные считаются аккуратно организованными, если каждое значение находится в собственной ячейке, каждая переменная — в собственном столбце, а каждое наблюдение — в собственной строке.

До сих пор все данные, с которыми мы имели дело, были аккуратно организованными. На практике большинство данных таковыми не являются, и мы еще вернемся к этому в главе 9.

Вариация

Вариация — это свойство значений переменной изменяться от измерения к измерению. Вы легко найдете примеры вариации в реальной жизни: если вы дважды измерите значение любой непрерывной переменной, то получите два разных результата. Это относится даже к измерению таких постоянных величин, как скорость света. Результат каждого измерения всегда включает небольшую ошибку, величина которой различна от измерения к измерению. Значения категориальных переменных также могут варьироваться, если измерения выполняются для разных объектов (например, цвет глаз у разных людей) или в разное время (например, уровни энергии электрона в разные моменты времени). Вариация каждой переменной имеет свои закономерности, что позволяет собирать интересную информацию. Наилучший способ

понять эти закономерности — визуализировать распределение значений переменной.

Визуализация распределений

Как именно следует визуализировать распределение значений переменной, зависит от того, является ли переменная категориальной или непрерывной. *Категориальными* называют переменные, которые имеют небольшой набор возможных значений. В языке R категориальные переменные обычно сохраняются в виде факторов или символьных векторов. Для изучения распределения категориальной переменной используйте столбчатую диаграмму (рис. 5.1).

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```

Рис. 5.1

Высота столбцов соответствует количеству наблюдений для каждого значения `x`. Вы можете вычислять соответствующие данные вручную с помощью функции `dplyr::count()`.

```
diamonds %>%
  count(cut)
#> # A tibble: 5 × 2
```

```
#>         cut     n
#>       <ord> <int>
#> 1      Fair  1610
#> 2      Good  4906
#> 3 Very Good 12082
#> 4   Premium 13791
#> 5     Ideal 21551
```

Переменная является *непрерывной*, если она может принимать любое из бесконечного множества упорядоченных значений. Например, непрерывными переменными являются числа, а также переменные даты и времени. Для изучения распределения непрерывной переменной используйте гистограмму (рис. 5.2).

```
ggplot(data = diamonds) +
  geom_histogram(mapping = aes(x = carat), binwidth = 0.5)
```

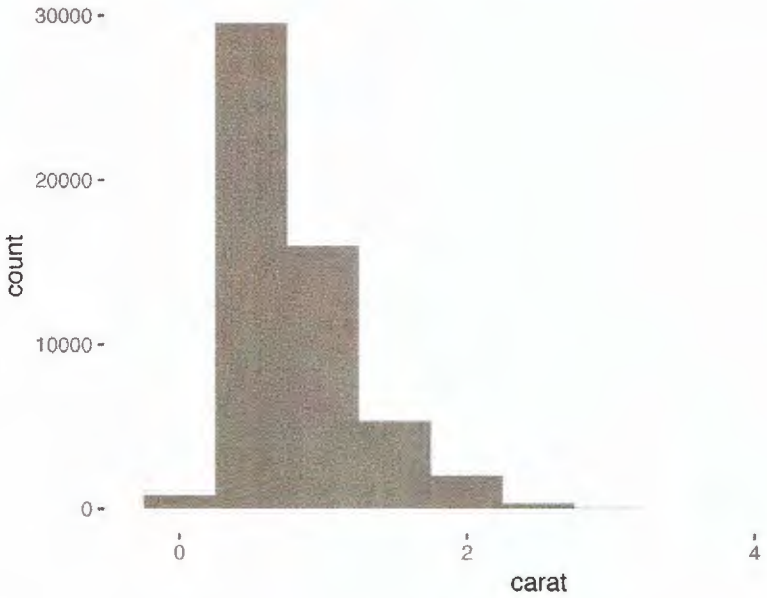


Рис. 5.2

Вы можете вычислять соответствующие данные вручную с помощью сочетания функций `dplyr::count()` и `ggplot2::cut_width()`.

```
diamonds %>%
  count(cut_width(carat, 0.5))
#> # A tibble: 11 × 2
#>   `cut_width(carat, 0.5)`     n
#>                    <fctr> <int>
```

```
#> 1            [-0.25,0.25]   785
#> 2             (0.25,0.75] 29498
#> 3             (0.75,1.25] 15977
#> 4             (1.25,1.75]  5313
#> 5             (1.75,2.25]  2002
#> 6             (2.25,2.75]   322
#> # ... with 5 more rows
```

Гистограмма разбивает ось x на равные интервалы (карманы), а затем определяет высоту каждого столбца по количеству наблюдений, попадающих в каждый интервал. Как следует из предыдущей диаграммы, самый высокий столбец указывает на то, что почти в 30000 наблюдений переменная `carat` попадает в интервал значений от 0,25 до 0,75, границы которого являются левым и правым краями столбца.

Размер интервалов гистограммы можно задать с помощью аргумента `binwidth`, измеряемого в единицах переменной `x`. В процессе работы с гистограммами вы всегда должны исследовать результаты с использованием различных значений ширины карманов, поскольку это позволяет проявлять различные закономерности. В качестве примера ниже показано, как изменится вид предыдущей гистограммы, если ограничиться бриллиантами весом менее трех каратов и выбрать меньшую ширину кармана (рис. 5.3).

```
smaller <- diamonds %>%
  filter(carat < 3)

ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.1)
```

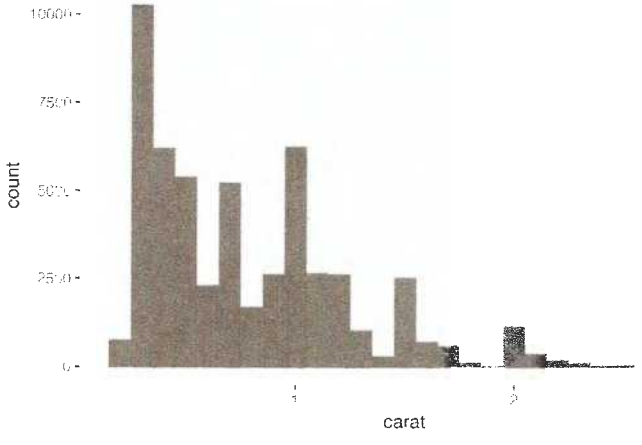


Рис. 5.3

В тех случаях, когда вы хотите отобразить на одной и той же диаграмме несколько гистограмм, наложив их одна на другую, мы рекомендуем использовать вместо функции `geom_histogram()` функцию `geom_freqpoly()`. Последняя выполняет те же вычисления, что и `geom_histogram()`, но вместо отображения количества наблюдений с помощью столбцов она использует линии. Различать перекрывающиеся линии гораздо легче, чем перекрывающиеся столбцы (рис. 5.4).

```
ggplot(data = smaller, mapping = aes(x = carat, color = cut)) +
  geom_freqpoly(binwidth = 0.1)
```

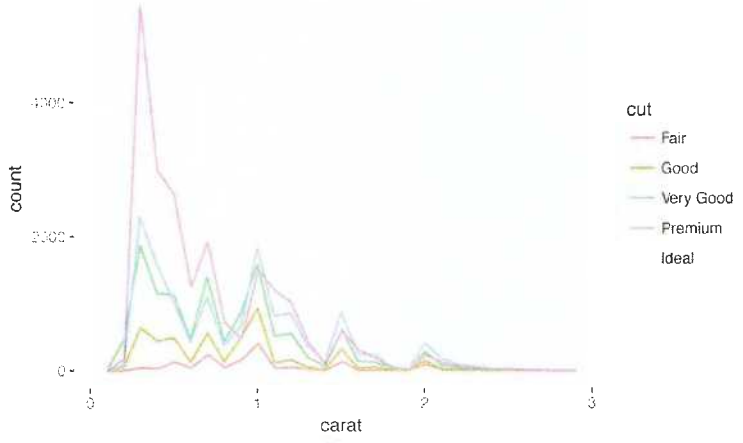


Рис. 5.4

При работе с графиками этого типа могут возникать определенные трудности, к обсуждению которых мы вернемся в разделе “Категориальные и непрерывные переменные”.

Типичные значения

Как в случае столбчатых диаграмм, так и гистограмм высокие столбцы указывают на часто встречающиеся значения переменной, а короткие — на те, которые встречаются редко. Участки, на которых столбцы не видны, указывают на то, что эти значения не встречаются среди данных. Чтобы превратить подобную информацию в полезные вопросы, ищите нечто необычное.

- Какие значения встречаются наиболее часто? Почему?
- Какие значения являются редкими? Почему? Соответствует ли это вашим ожиданиям?

- Замечаете ли вы некоторые необычные закономерности? Чем их можно объяснить?

В качестве примера рассмотрим некоторые интересные вопросы, на которые наталкивает приведенная ниже диаграмма (рис. 5.5).

- Почему в большем количестве встречаются бриллианты, вес которых в каратах выражается целочисленными значениями и распространенными дробными значениями?
- Почему больше бриллиантов встречается в областях значений непосредственно справа от пиков, чем непосредственно слева от них?
- Почему на диаграмме не отображены данные бриллиантов весом более 3 каратов?

```
ggplot(data = smaller, mapping = aes(x = carat)) +
  geom_histogram(binwidth = 0.01)
```

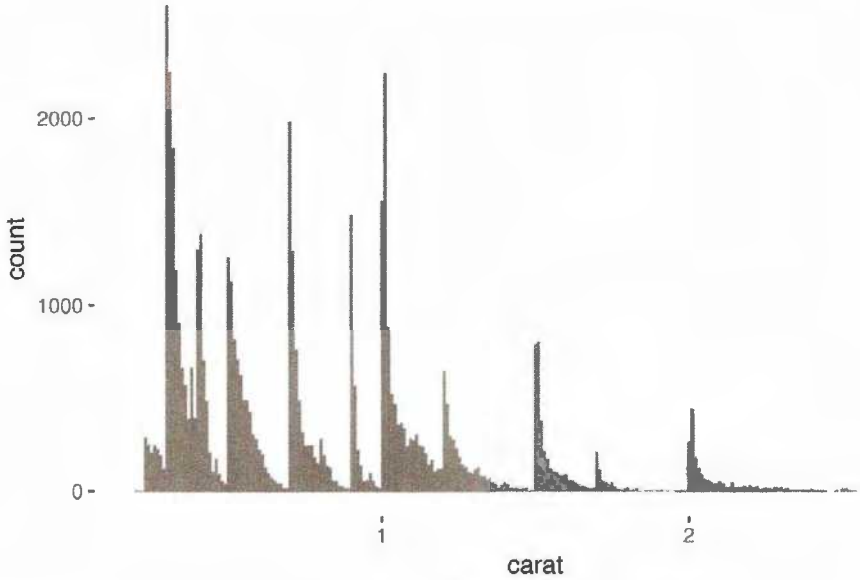


Рис. 5.5

Вообще говоря, образование кластеров близких значений указывает на существование подгрупп среди данных. Чтобы понять природу подгрупп, задайте следующие вопросы.

- Насколько близки друг к другу наблюдения в пределах каждого кластера?
- Насколько различаются между собой наблюдения, принадлежащие к разным кластерам?
- Как можно объяснить или описать кластеры?
- Почему внешний вид кластеров может вводить в заблуждение?

На следующей гистограмме (рис. 5.6) отображена продолжительность в минутах) 272 извержений гейзера "Старый служака", который находится в Йелоустоунском парке. Наблюдения заметно кластеризуются в две группы: кратковременные (продолжительностью около 2 минут) и длительные (4–5 минут) извержения, промежуток между которыми заполнен незначительным количеством извержений другой длительности.

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +
geom_histogram(binwidth = 0.25)
```

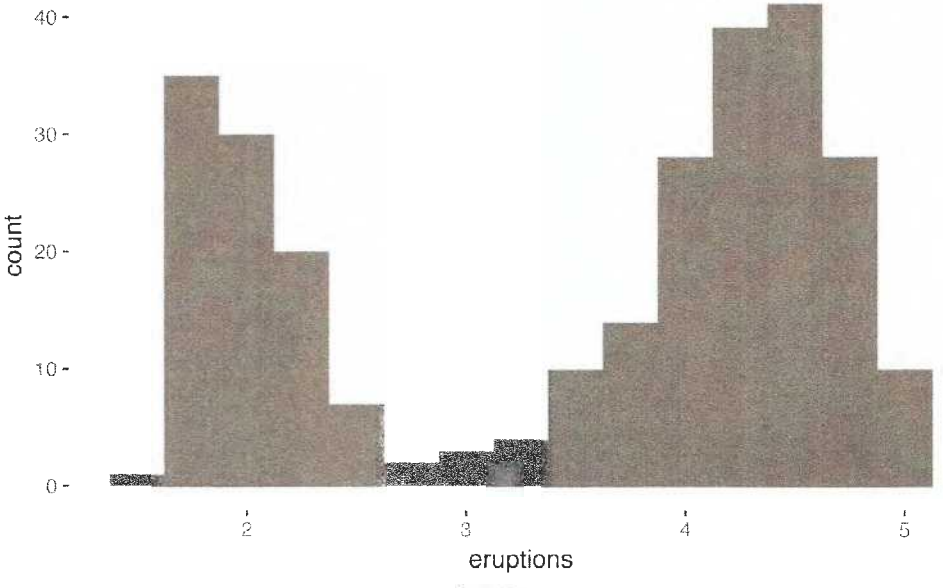


Рис. 5.6

Многие из предыдущих вопросов могут побудить вас к исследованию соотношения между переменными, например, для того, чтобы проверить, могут ли значения одной переменной объяснить поведение другой переменной. Вскоре мы обсудим этот вопрос.

Необычные значения

Выбросами называют необычные наблюдения — точки данных, выпадающие из общего ряда. В одних случаях выбросы обусловлены ошибками в точках данных, тогда как в других они могут указывать путь к новым научным открытиям. Иногда, когда данных очень много, заметить выбросы на гистограмме бывает нелегко.

В качестве примера рассмотрим распределение переменной `y` из набора данных о бриллиантах (рис. 5.7). Единственным свидетельством наличия выбросов являются необычайно широкие пределы изменения значений `y`.

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5)
```

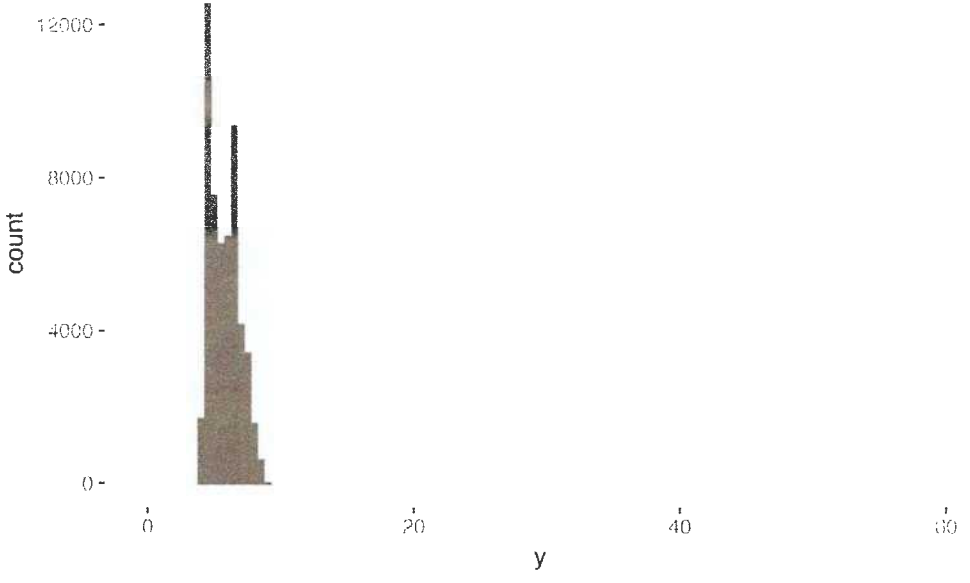


Рис. 5.7

Количество наблюдений в плотно заполненных интервалах настолько велико, что в разреженных интервалах их невозможно заметить (разве что, если вы пристально вглядитесь в окрестность 0, то сможете там что-то увидеть). Чтобы было легче заметить необычные значения, необходимо увеличить масштаб до малых значений вдоль оси *y* с помощью функции `coord_cartesian()` (рис. 5.8).

```
#> 7  0.71      Good     F     SI2  64.1    60  2130  0.00
#> 8  0.51     Ideal     E     VS1  61.8    55  2075  5.15
#> 9  2.00   Premium     H     SI2  58.9    57 12210  8.09
#> # ... with 2 more variables:
#> #  y <dbl>, z <dbl>
```

Переменная `y` измеряет (в миллиметрах) один из трех размеров бриллиантов. Мы знаем, что ширина бриллианта не может быть равной 0, так что эти значения, должно быть, некорректные. Кроме того, мы можем предположить, что размеры 32 и 59 мм также невозможны: эти бриллианты имеют длину свыше одного дюйма, но не стоят сотен тысяч долларов!

Хорошая практика заключается в повторном выполнении анализа как с учетом, так и без учета выбросов. Если выбросы оказывают лишь минимальное влияние на результаты и установить причину их появления не удается, то имеет смысл заменить их отсутствующими значениями и двигаться дальше. Однако если влияние выбросов на результаты велико, их нельзя отбрасывать без всяких на то оснований. Вам нужно будет попытаться установить причину появления выбросов (например, это может быть следствием ошибки при вводе данных) и сопроводить анализ примечанием о том, что вы их удалили.

Упражнения

1. Исследуйте распределение значений каждой из переменных `x`, `y` и `z` для набора данных `diamonds`. Что вы из этого узнали? Коль скоро речь идет о бриллиантах, как вы могли бы определить, какие именно из этих переменных представляют длину, ширину и глубину?
2. Исследуйте распределение значений переменой `price` (цена). Замечаете ли вы что-то необычное или удивительное? (Подсказка: тщательно обдумайте выбор ширины кармана (`binwidth`) и используйте широкий интервал значений.)
3. Сколько бриллиантов имеют вес 0,99 карата? Сколько бриллиантов имеют вес 1 карат? Чем, по вашему мнению, обусловлена разница в этих количествах?
4. Чем отличается работа функции `coord_cartesian()` от работы функций `xlim()` и `ylim()` при увеличении масштаба изображения гистограммы? Что произойдет, если не задать аргумент `binwidth`? Что произойдет, если вы попытаетесь увеличить масштаб изображения гистограммы таким образом, чтобы отображалась лишь половина столбца?

Отсутствующие значения

Если вы заметили в наборе данных необычные значения и просто хотите перейти к оставшейся части анализа, то это можно сделать одним из следующих двух способов.

- Отбросьте всю строку, содержащую необычные значения.

  ```
  diamonds2 <- diamonds %>%
    filter(between(y, 3, 20))
  ```

  Я не рекомендую использовать эту возможность, поскольку наличие одного недействительного измерения еще не означает, что все остальные измерения также недействительны. Кроме того, если данные, с которыми вы работаете, имеют низкое качество, то после того, как вы примените такой подход ко всем переменным, в вашем распоряжении может вообще не оказаться данных!

- Вместо этого рекомендуем заменить необычные значения отсутствующими значениями. Проще всего это можно сделать, заменив переменную ее видоизмененной копией с помощью функции `mutate()`. Для замены необычных значений значениями `NA` можно использовать функцию `ifelse()`.

  ```
  diamonds2 <- diamonds %>%
    mutate(y = ifelse(y < 3 | y > 20, NA, y))
  ```

Функция `ifelse()` имеет три аргумента. Первый аргумент тестирует логический вектор. В случае положительного результата тестирования (`TRUE`) функция возвращает второй аргумент; если же результат тестирования отрицательный (`FALSE`), то функция возвращает третий аргумент.

Как и R, пакет **ggplot2** поддерживает философию, в соответствии с которой отсутствующие значения никогда не должны молчаливо исчезать. Поскольку абсолютно не очевидно, где на графике должны откладываться отсутствующие значения, **ggplot2** не включает их в график, но при этом выводит предупреждающее сообщение о том, что эти значения были исключены (рис. 5.9).

```
ggplot(data = diamonds2, mapping = aes(x = x, y = y)) +
  geom_point()
#> Warning: Removed 9 rows containing missing values
#> (geom_point).
```

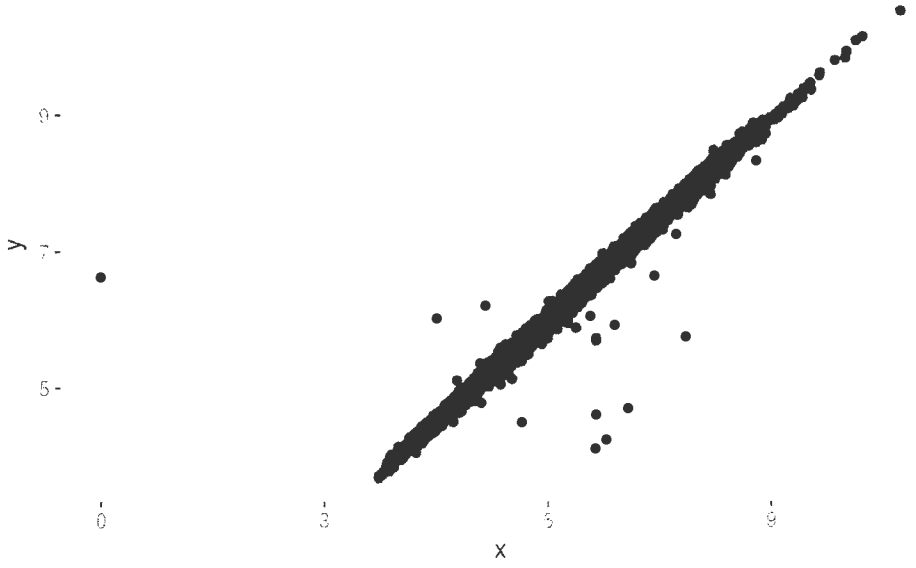


Рис. 5.9

Чтобы отменить вывод таких сообщений, задайте параметр `na.rm = TRUE`.

```
ggplot(data = diamonds2, mapping =     (x = x, y = y)) +
  geom_point(na.rm = TRUE)
```

Иногда у вас может возникнуть желание выяснить, что именно отличает наблюдения с отсутствующими значениями от наблюдений, содержащих все значения. Например, в случае набора данных `nycflights13::flights` отсутствующие значения в переменной `dep_time` указывают на то, что данный авиарейс был отменен. Возможно, вы захотите сравнить количество отмененных и неотмененных авиарейсов для всех времен отправки, содержащихся в расписании. Это можно сделать, создав новую переменную с помощью функции `is.na()` (рис. 5.10).

```
nycflights13::flights %>%
  mutate(
    cancelled = is.na(dep_time),
    sched_hour = sched_dep_time %/% 100,
    sched_min = sched_dep_time %% 100,
    sched_dep_time = sched_hour + sched_min / 60
  ) %>%
  ggplot(mapping = aes(sched_dep_time)) +
    geom_freqpoly(
```

```
  mapping = aes(color = cancelled),
  binwidth = 1/4
)
```

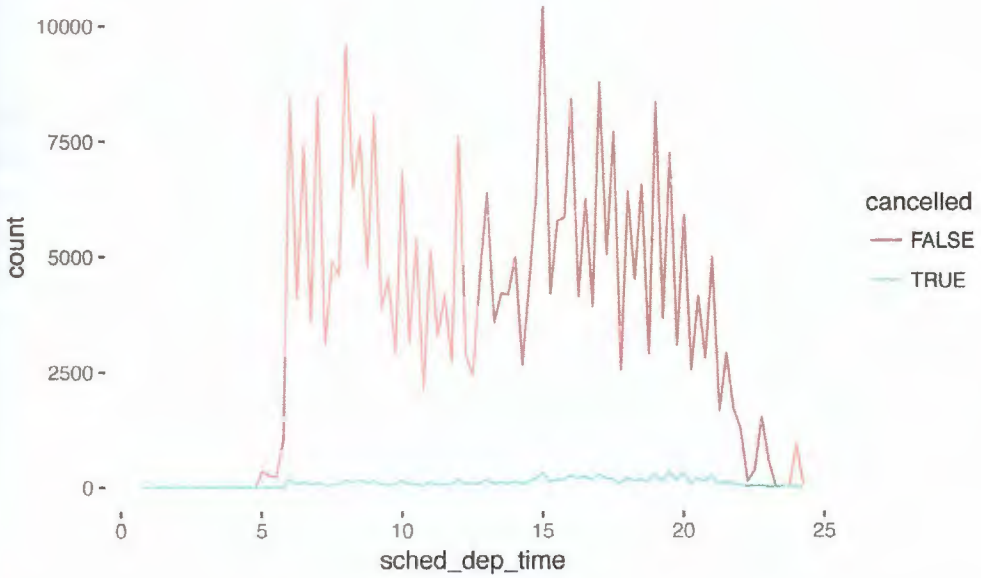


Рис. 5.10

Однако такое сравнение не может считаться вполне удовлетворительным, поскольку количество неотмененных авиарейсов значительно превышает количество отмененных. В следующем разделе вы познакомитесь с методиками, позволяющими улучшить это сравнение.

Упражнения

1. Что происходит с отсутствующими значениями в случае гистограммы? Что происходит с отсутствующими значениями в случае столбчатой диаграммы? В чем причина различий?
2. Какова роль аргумента `na.rm = TRUE` в случае функций `mean()` и `sum()`?

Ковариация

Если вариация описывает поведение значений в пределах *одной* переменной, то ковариация описывает совместное поведение значений в пределах *нескольких* переменных. *Ковариация* — это тенденция к взаимосвязанному

изменению значений двух и более переменных. Лучший способ отследить наличие ковариации — это визуализировать соотношения между двумя или несколькими переменными. Как именно это следует делать, зависит от типа интересующих вас переменных.

Категориальные и непрерывные переменные

Довольно часто вы будете исследовать распределение непрерывной переменной, диапазон значений которой разбит на интервалы, соответствующие значениям категориальной переменной, как на предыдущем сравнительном графике. Используемый по умолчанию внешний вид графика, созданного с помощью функции `geom_freqpoly()`, не очень подходит для такого рода сравнений, поскольку высота графика определяется отсчетами (`count`), т.е. количеством значений, попадающих в данный интервал. Это означает, что в тех случаях, когда размер одной группы значительно превышает размер другой, заметить различия в форме кривых нелегко. Рассмотрим в качестве примера, как изменяется цена бриллиантов в зависимости от их качества (рис. 5.11).

```
ggplot(data = diamonds, mapping = aes(x = price)) +
  geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```

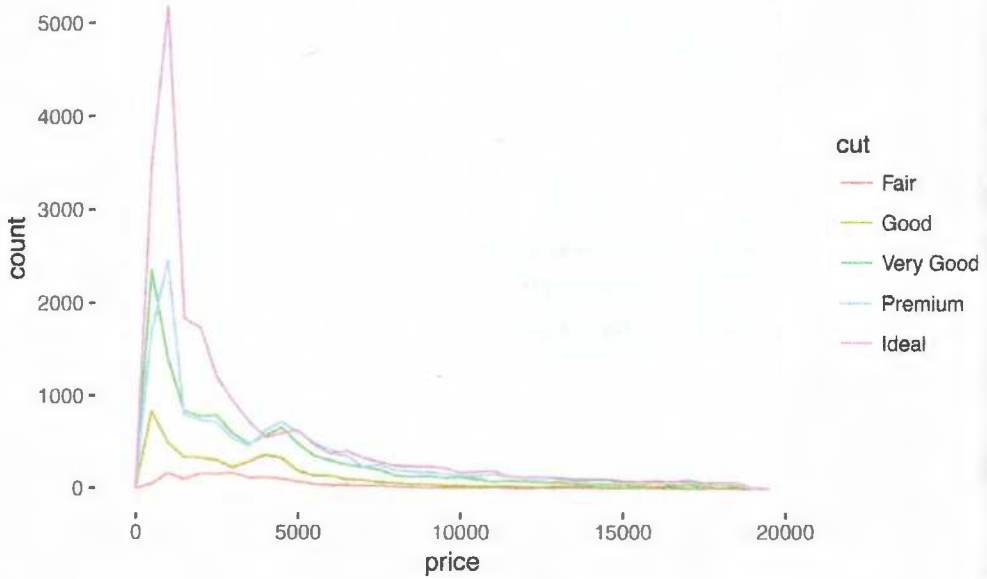


Рис. 5.11

Заметить различия в распределениях трудно ввиду больших различий в размерах групп (рис. 5.12).

```
ggplot(diamonds) +
  geom_bar(mapping = aes(x = cut))
```

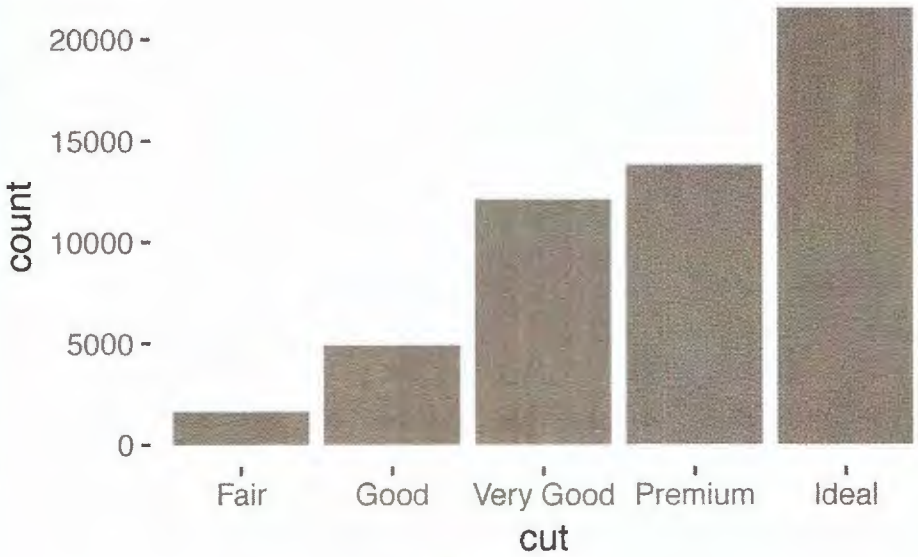


Рис. 5.12

Чтобы облегчить сравнение распределений, необходимо поменять переменную, отображаемую вдоль оси *y*. Давайте отобразим вместо отсчетов (`count`) *плотность* (`density`), являющуюся стандартизированной количественной характеристикой, т.е. такой, что площадь под каждой кривой равна 1 (рис. 5.13).

```
ggplot(
  data = diamonds,
    mapping = aes(x = price, y = ..density..)
  ) +
    geom_freqpoly(mapping = aes(color = cut), binwidth = 500)
```

С этими графиками связана одна удивительная особенность: из них следует, что наиболее высокая средняя цена характерна для бриллиантов наихудшего качества (`fair`)! Но не исключено, что этот вывод стал возможен лишь потому, что интерпретация полигонов частот требует более внимательного отношения — значительная часть содержащейся в них информации остается скрытой за графическими пластами.

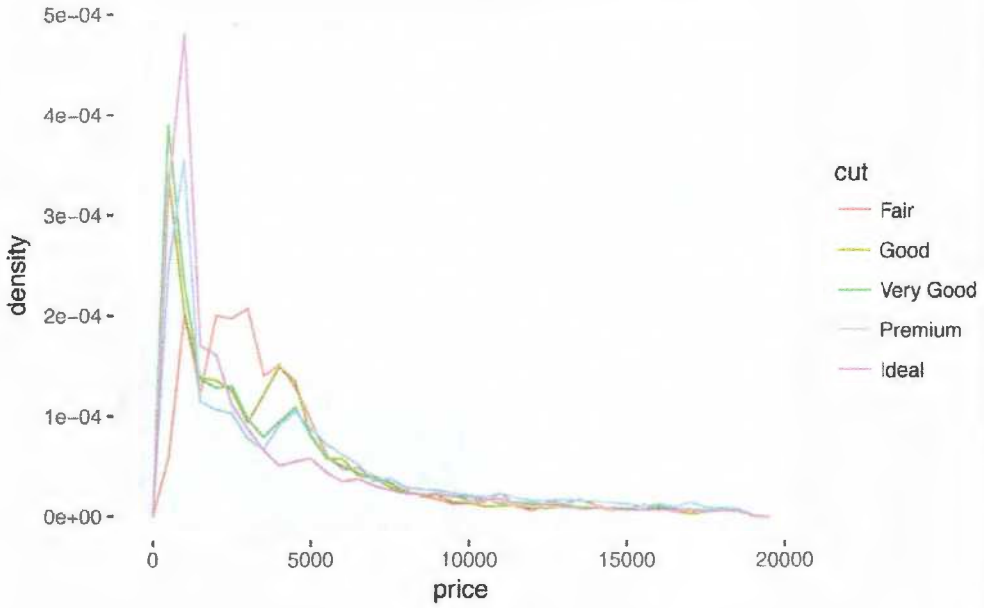


Рис. 5.13

Другим возможным вариантом отображения распределения непрерывной переменной, диапазон значений которой разбит на частотные интервалы с помощью категориальной переменной, является *диаграмма размаха* (boxplot), другим распространенным названием которой является "ящик с усами". Диаграммы размаха удобно использовать для визуальной экспресс-оценки распределения значений, и они пользуются широкой популярностью среди статистиков. Диаграмма размаха состоит из следующих элементов (рис. 5.14).

- Прямоугольник ("ящик"), ограниченный 25-м и 75-м процентами, расстояние между которыми носит название *межквартильный размах* (IQR). Внутри прямоугольника проходит линия, отмечающая *медиану*, т.е. 50-й процентиль распределения. Эти три линии дают представление о размахе распределения и позволяют судить о том, симметрично ли распределение относительно медианы или скошено в одну сторону.
- Визуальные точки, отстоящие от границ прямоугольника на расстояние, превышающее 1,5 IQR. Эти экстремальные точки, или выбросы, являются необычными и поэтому отображаются по отдельности.
- Две линии (или "усы"), проходящие от границ прямоугольника до самой дальней точки распределения с каждой стороны, не являющейся выбросом.

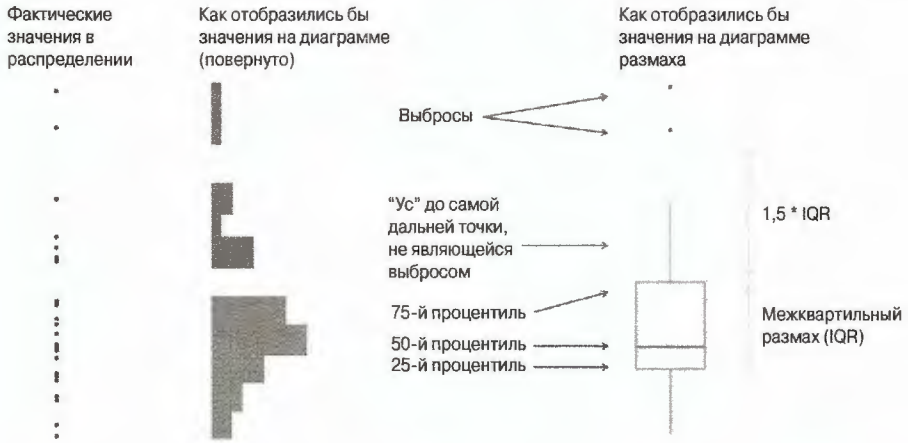


Рис. 5.14

Рассмотрим распределение цены (`price`) в зависимости от качества огранки (`cut`), используя функцию `geom_boxplot()` (рис. 5.15).

```
ggplot(data = diamonds, mapping = aes(x = cut, y = price)) +
  geom_boxplot()
```

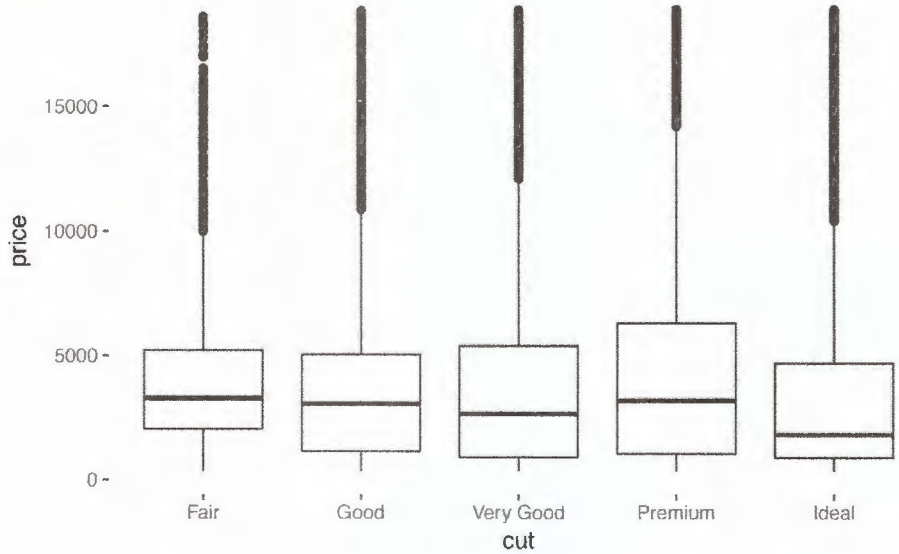


Рис. 5.15

На диаграмме размаха отображается гораздо меньше информации о распределении, но компактность диаграмм этого типа облегчает их сравнение между собой (поскольку их можно расположить рядом одна с другой). Эта

диаграмма также свидетельствует в пользу того противоречащего интуиции вывода, что бриллианты наилучшего качества имеют более низкую среднюю цену! В упражнении вам будет предложено дать этому объяснение.

Переменная `cut` выступает в качестве порядкового фактора: класс качества `fair` (удовлетворительный) ниже класса `good` (хороший), который, в свою очередь, ниже класса `very good` (очень хороший) и т.д. Многие категориальные переменные не имеют такого же внутренне присущего им фактора упорядочивания, и в таком случае вы можете переставлять их, чтобы сделать диаграмму более информативной. В частности, это можно сделать с помощью функции `reorder()`.

Рассмотрим, например, переменную `class` из набора данных `mpg`. Предположим, вы хотите выяснить, как меняется показатель расхода топлива (`hwy`) в зависимости от класса автомобиля (`class`), как показано на рис. 5.16.

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) +
  geom_boxplot()
```

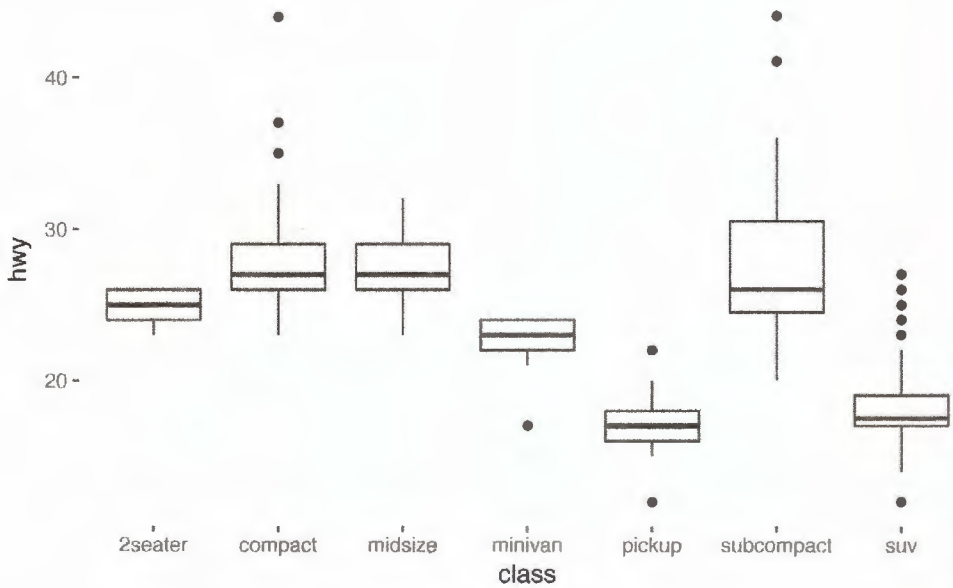


Рис. 5.16

Чтобы сделать тренд более наглядным, переставим значения класса, основываясь на медиане значений `hwy` (рис. 5.17).

```
ggplot(data = mpg) +
  geom_boxplot(
    mapping = aes(
      x = reorder(class, hwy, FUN = median),
      y = hwy
    )
```

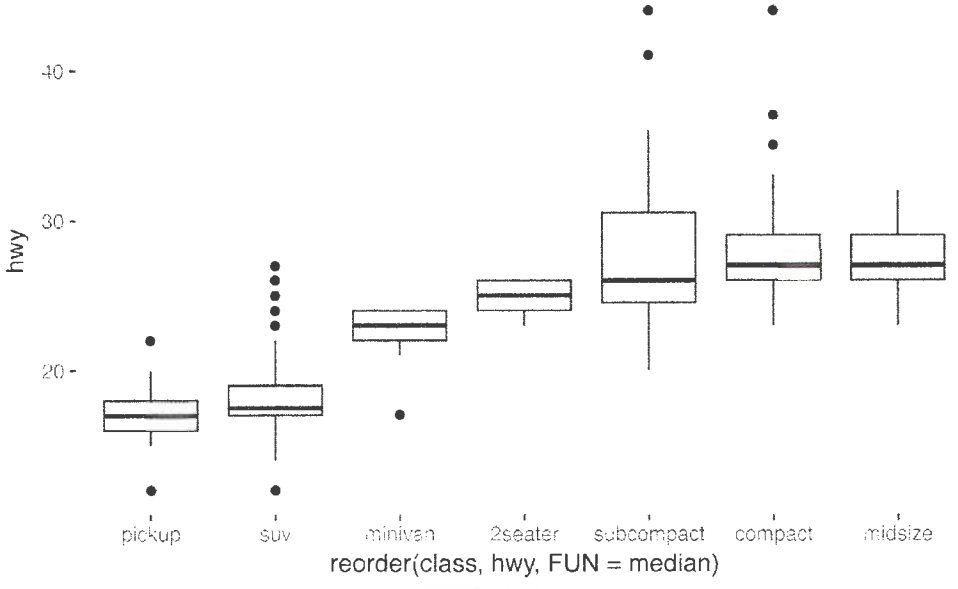


Рис. 5.17

При наличии переменных с длинными именами функция `geom_boxplot()` будет работать лучше, если повернуть диаграмму на 90°. Это можно сделать с помощью функции `coord_flip()` (рис. 5.18).

```
ggplot(data = mpg) +
  geom_boxplot(
    mapping = aes(
      x = reorder(class, hwy, FUN = median),
      y = hwy
    )
  ) +
  coord_flip()
```

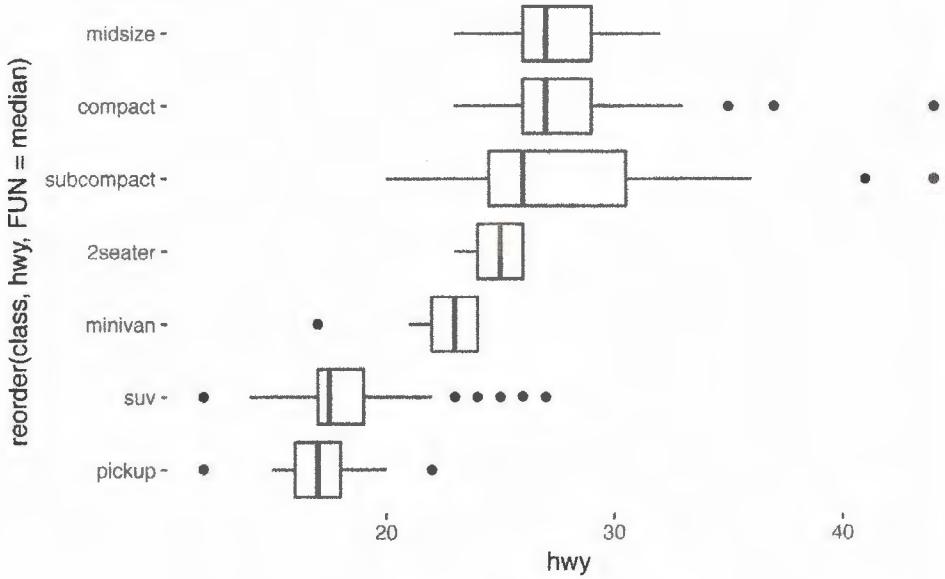


Рис. 5.18

Упражнения

1. Используйте то, о чем вы узнали, для улучшения визуализации времени отбытия отмененных авиарейсов в сравнении со временем отбытия неотмененных.
2. Какая из переменных набора данных `diamonds` наиболее важна для прогнозирования цены бриллианта? Как эта переменная коррелирует с переменной `cut`? Почему комбинация этих двух соотношений приводит к тому, что бриллианты худшего качества оказываются более дорогими?
3. Установите пакет **ggstance** и создайте горизонтальную диаграмму размаха. Как это соотносится с использованием функции `coord_flip()`?
4. Одной из проблем, свойственных диаграммам размаха, является то, что они разрабатывались в те времена, когда размеры наборов данных были сравнительно небольшими, и потому отображают недопустимо большое количество выбросов. Одним из подходов, позволяющих решить эту проблему, является использование *lv-диаграмм*. Установите пакет **lvplot** и попытайтесь использовать функцию `geom_lv()` для отображения распределения цены бриллиантов в зависимости от качества огранки. Что вам удалось выяснить? Как вы интерпретируете эти графики?

5. Сравните работу функции contrast geom_violin() с работой функции geom_histogram(), использующей срезы (facets), и работой функции geom_freqpoly(), использующей цвет. Каковы преимущества и недостатки каждого из этих методов?
6. В случае небольших наборов данных для анализа связи между непрерывной и категориальной переменными иногда удобно использовать функцию geom_jitter(). Пакет **ggbeeswarm** предоставляет ряд методов, аналогичных функции geom_jitter(). Перечислите их и кратко опишите работу каждого из них.

Две категориальные переменные

Чтобы визуализировать вариацию между категориальными переменными, необходимо подсчитать количество наблюдений для каждой комбинации значений. Один из способов сделать это заключается в использовании встроенной функции geom_count() (рис. 5.19).

```
ggplot(data = diamonds) +
  geom_count(mapping = aes(x = cut, y = color))
```

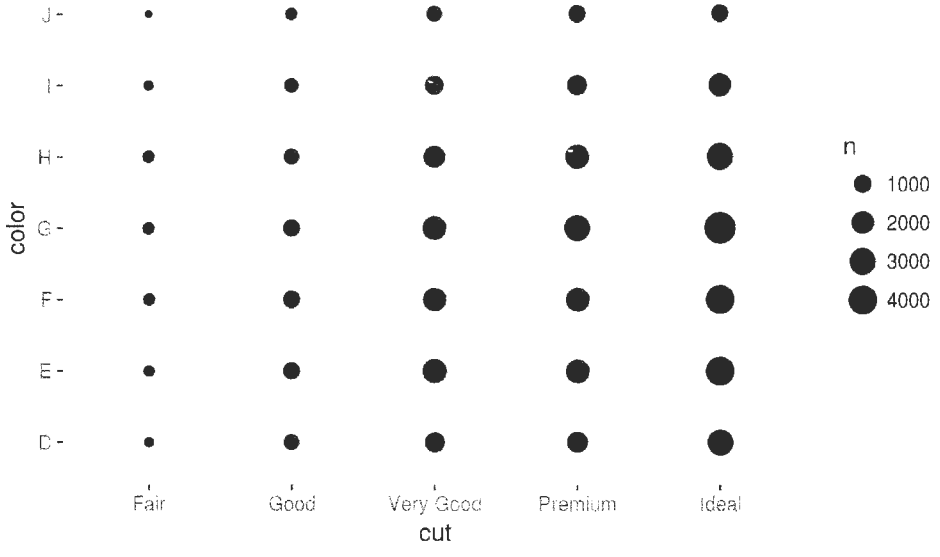


Рис. 5.19

Размер каждого кружка отражает количество наблюдений, встречающихся в каждой комбинации значений. Ковариация проявляется в сильной корреляции между конкретными значениями x и y.

Другой подход заключается в выполнении подсчета с помощью пакета **dplyr**.

```
diamonds %>%
  count(color, cut)
#> Source: local data frame [35 x 3]
#> Groups: color [?]
#>
#>   color       cut     n
#>   <ord>     <ord> <int>
#> 1     D      Fair   163
#> 2     D      Good   662
#> 3     D Very Good  1513
#> 4     D   Premium  1603
#> 5     D     Ideal  2834
#> 6     E      Fair   224
#> # ... with 29 more rows
```

После этого остается визуализировать данные, используя функцию `geom_tile()` и визуальный атрибут `fill` (рис. 5.20).

```
diamonds %>%
  count(color, cut) %>%
  ggplot(mapping = aes(x = color, y = cut)) +
    geom_tile(mapping = aes(fill = n))
```

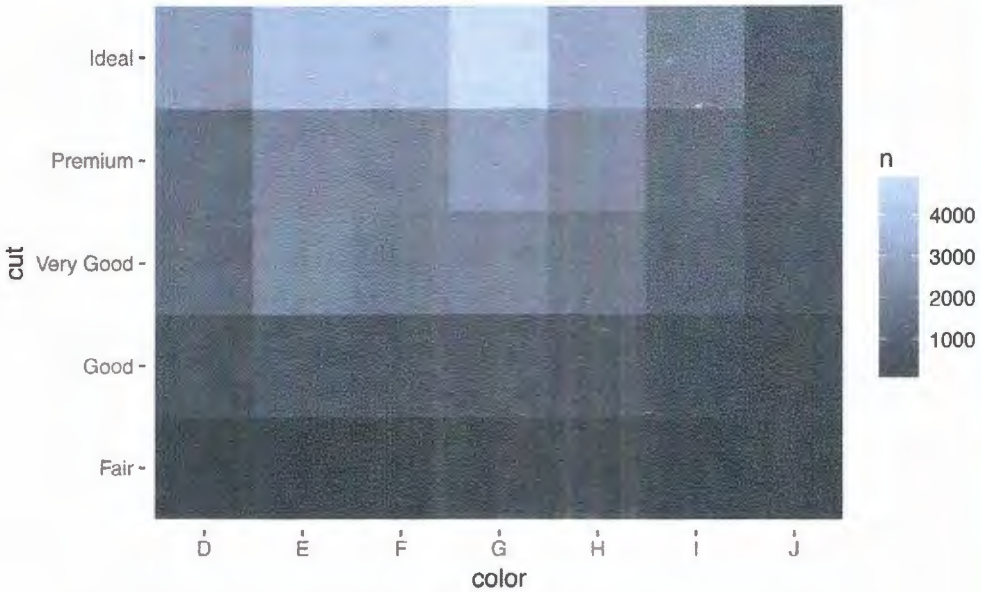


Рис. 5.20

Если категориальные переменные не упорядочены, используйте пакет **seriation** для одновременной перестановки строк и столбцов, чтобы более отчетливо проявить интересные закономерности. В случае крупных диаграмм попытайтесь использовать один из пакетов **d3heatmap** или **heatmaply**, позволяющих создавать интерактивные диаграммы.

Упражнения

1. Как бы вы изменили масштаб набора данных `count`, чтобы более отчетливо проявить распределение переменной `cut` с использованием `color` в качестве группирующей переменной или распределение переменной `color` с использованием `cut` в качестве группирующей переменной?
2. Используйте функцию `geom_tile()` вместе с пакетом **dplyr** для исследования того, как меняется среднее время задержки авиарейсов в зависимости от пункта назначения и месяца в году. Что затрудняет чтение диаграмм? Как бы вы могли это улучшить?
3. Почему в предыдущем примере лучше использовать атрибуты `aes(x = color, y = cut)`, чем атрибуты `aes(x = cut, y = color)`?

Две непрерывные переменные

Вы уже знакомы с одним из замечательных способов визуализации ковариации между двумя непрерывными переменными: создание диаграммы рассеяния (точечной диаграммы) с помощью функции `geom_point()`. Ковариация проявляет себя в виде закономерности в расположении точек.

Например, легко просматривается экспоненциальный характер связи между ценой и весом бриллиантов в каратах (рис. 5.21).

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = carat, y = price))
```

По мере увеличения набора данных диаграммы рассеяния становятся все менее полезными, поскольку точки налагаются одна на другую с образованием полностью закрашенных областей (как на предыдущей диаграмме). С одним из способов решения проблемы вы уже знакомы — это использование визуального атрибута `alpha` для добавления прозрачности (рис. 5.22).

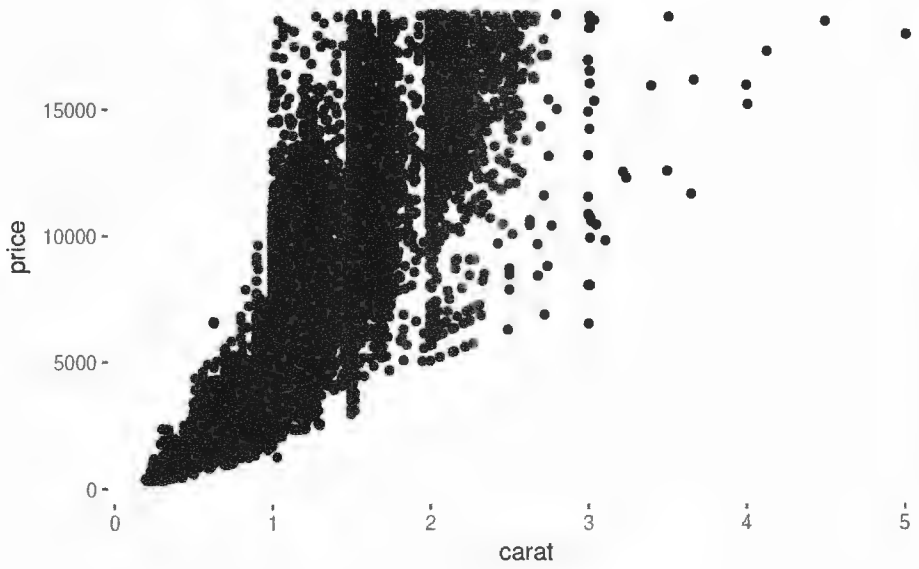


Рис. 5.21

```
ggplot(data = diamonds) +
  geom_point(
    mapping = aes(x = carat, y = price),
    alpha = 1 / 100
  )
```

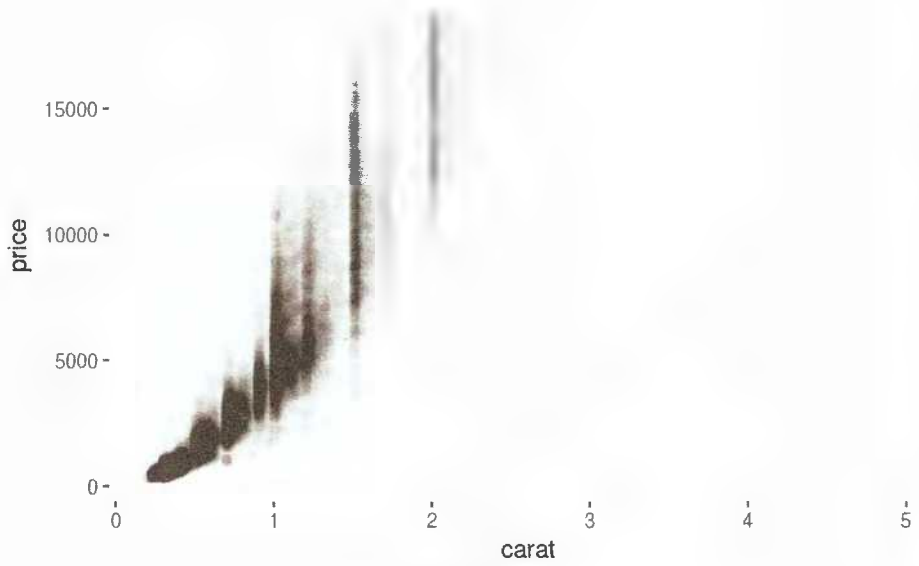


Рис. 5.22

Но для очень больших наборов данных введение прозрачности не является выходом. Другое возможное решение — использование карманов (интервалов значений). Ранее мы использовали функции `geom_histogram()` и `geom_freqpoly()` для создания карманов в одном измерении. Сейчас вы узнаете о том, как создавать карманы в двух измерениях с помощью функций `geom_bin2d()` и `geom_hex()`.

Функции `geom_bin2d()` и `geom_hex()` делят координатную плоскость на двухмерные карманы, а затем используют цвет заливки для представления количества точек, попадающих в каждый карман. Функция `geom_bin2d()` создает прямоугольные карманы, а функция `geom_hex()` — шестиугольные. Чтобы использовать функцию `geom_hex()`, нужно установить пакет **hexbin**.

```
ggplot(data = smaller) +
  geom_bin2d(mapping = aes(x = carat, y = price))

# install.packages("hexbin")
ggplot(data = smaller) +
  geom_hex(mapping = aes(x = carat, y = price))
#> Loading required package: methods
```

Другая возможность заключается в создании карманов для одной непрерывной переменной с той целью, чтобы она выступала в качестве категориальной переменной. После этого можете использовать один из методов визуализации комбинации категориальной и непрерывной переменной, с которым вы уже знакомы. Например, можно создать карманы для переменной `carat` и отобразить диаграммы размаха для каждой группы (рис. 5.23).

```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_width(carat, 0.1)))
```

Используемая здесь функция `cut_width(x, width)` делит диапазон значений `x` на интервалы шириной `width`. По умолчанию "ящики с усами" имеют примерно один и тот же внешний вид (за исключением количества выбросов), независимо от количества наблюдений, которым они соответствуют, и поэтому судить о том, какое количество наблюдений просуммировано в каждом кармане, очень трудно. Один из способов сделать это возможным заключается в использовании карманов, ширина которых пропорциональна количеству точек данных: `varwidth = TRUE`.

Другой подход заключается в том, чтобы отображать примерно одно и то же количество точек в каждом кармане. Это делается с помощью функции `cut_number()` (рис. 5.24).

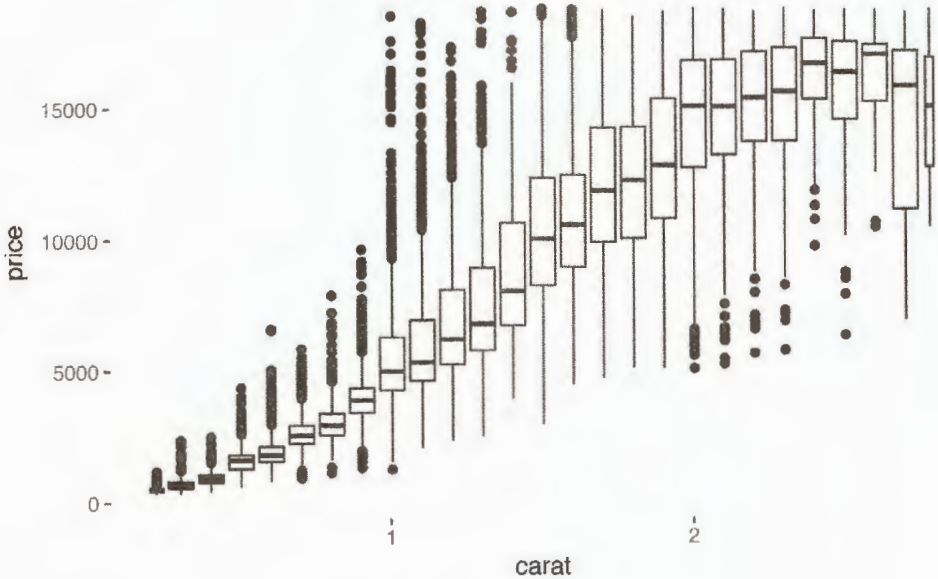


Рис. 5.23

```
ggplot(data = smaller, mapping = aes(x = carat, y = price)) +
  geom_boxplot(mapping = aes(group = cut_number(carat, 20)))
```

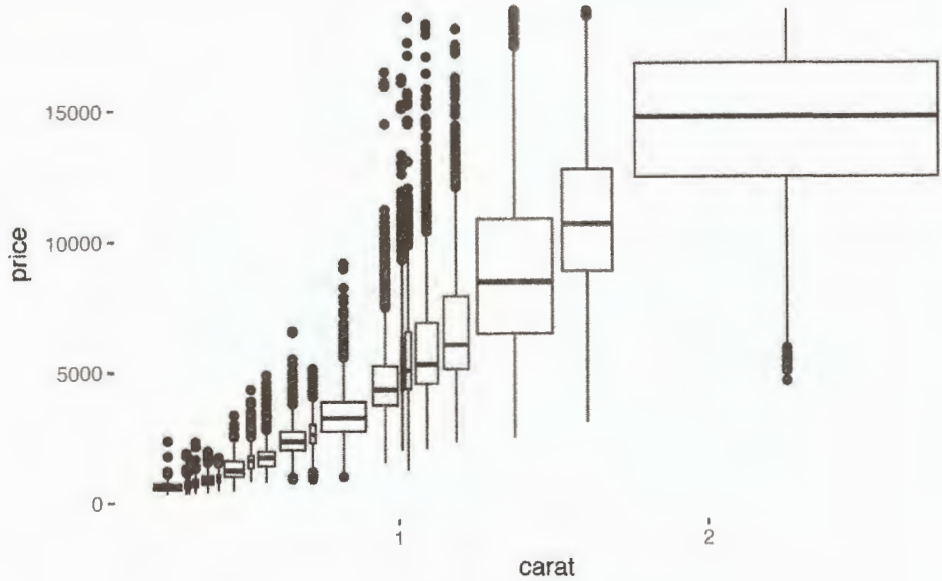


Рис. 5.24

Упражнения

1. Вместо суммирования условных распределений с помощью диаграмм размаха можно использовать полигоны частот. Какие факторы вы должны учитывать, принимая решение о том, какую из функций использовать: `cut_width()` или `cut_number()`? Как этот выбор влияет на визуализацию двухмерного распределения переменных `carat` и `price`?
2. Визуализируйте распределение переменной `carat`, используя `price` в качестве группирующей переменной.
3. Сравните распределение переменной `price` для крупных и небольших бриллиантов. Соответствуют ли результаты этого сравнения вашим ожиданиям или что-то в них вызывает у вас удивление?
4. Объедините две изученные вами методики для визуализации комбинированного распределения переменных `cut`, `carat` и `price`.
5. Двухмерные графики проявляют выбросы, которые на одномерных графиках остаются невидимыми. Например, на приведенной ниже диаграмме (рис. 5.25) некоторые точки имеют необычные сочетания значений `x` и `y`, что заставляет считать их выбросами, хотя взятые по отдельности значения `x` и `y` вполне обычны.

```
ggplot(data = diamonds) +
  geom_point(mapping = aes(x = x, y = y)) +
  coord_cartesian(xlim = c(4, 11), ylim = c(4, 11))
```

Почему в данном случае это легче заметить на диаграмме рассеяния, чем на диаграмме, использующей карманы?

Шаблоны поведения переменных и модели

Установление закономерностей в поведении данных имеет решающее значение для определения связей между ними. Если в поведении двух переменных наблюдается систематическая закономерность, то это может свидетельствовать о наличии устойчивой связи между данными. Как только вы замечаете какую-либо закономерность, задайте себе следующие вопросы.

- Может ли эта закономерность быть лишь кажущейся (т.е. не обусловлена ли она случайным совпадением)?

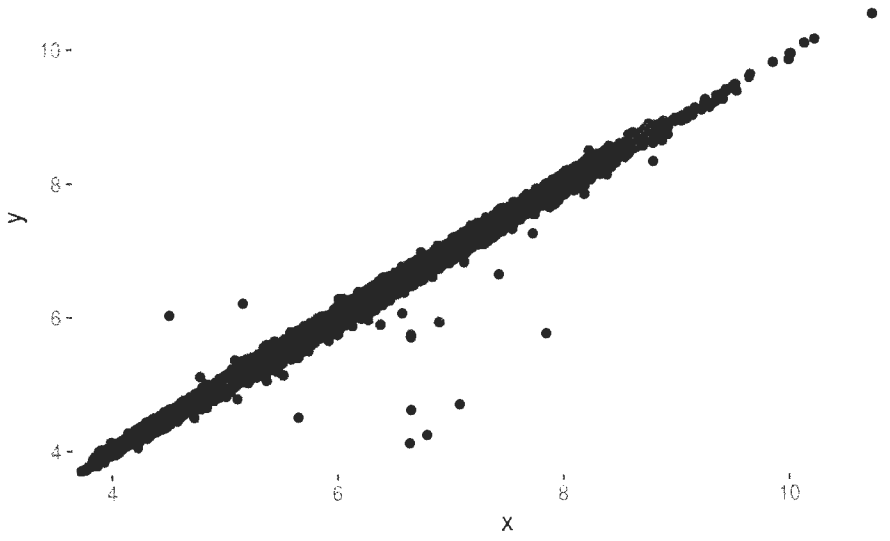


Рис. 5.25

- Как можно описать отношение между переменными, которое следует из наблюдаемой закономерности?
- Насколько сильна связь между переменными, которую можно предполагать, исходя из наблюдаемой закономерности?
- Какие другие переменные могут оказывать влияние на отношение между данными переменными?
- Изменяется ли это отношение при переходе к рассмотрению отдельных подгрупп данных?

На диаграмме рассеяния, по осям которой отложены длительность извержений гейзера "Старый служака" и длительность пауз между извержениями, можно заметить следующую закономерность: более продолжительные паузы связаны с более длительными извержениями. Кроме того, на диаграмме рассеяния отчетливо выделяются два кластера, на которые ранее уже обращалось ваше внимание (рис. 5.26).

```
ggplot(data = faithful) +
  geom_point(mapping = aes(x = eruptions, y = waiting))
```

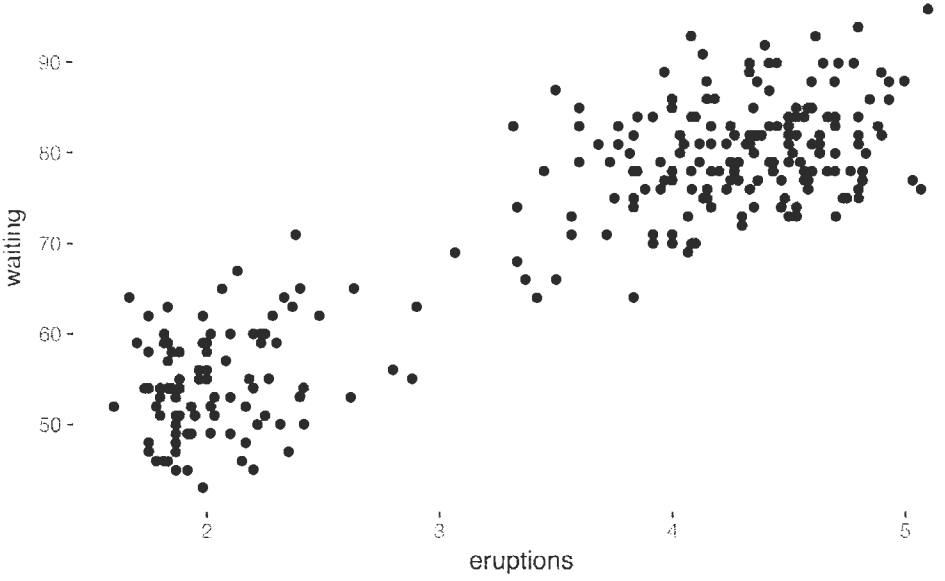


Рис. 5.26

Выявление закономерностей в поведении переменных является одним из наиболее полезных инструментов в науке о данных, поскольку это позволяет обнаружить ковариацию. Если вариация — это явление, создающее неопределенность, то ковариация — это явление, которое уменьшает неопределенность. Если между переменными существует ковариация, то значения одной переменной могут быть использованы для улучшения предсказания значений другой. Если ковариация обусловлена причинно-следственной связью (особый случай), то значения одной переменной могут быть использованы для управления значениями другой.

Модели — это инструмент, используемый для выявления закономерностей в поведении данных. Рассмотрим, например, данные, касающиеся бриллиантов. Понять природу отношения между переменными `cut` (качество огранки) и `price` (цена) нелегко, поскольку переменные `cut` и `carat` (вес в каратах), а также `carat` и `price` тесно связаны между собой. Можно использовать модель, которая исключает влияние очень сильной связи между переменными `price` и `carat` и тем самым предоставляет возможность исследовать более тонкие детали. Приведенный ниже код соответствует модели, которая предсказывает цену по весу, а затем вычисляет остатки (разница между предсказанным и фактическим значением). Остатки дают представление о цене бриллиантов после исключения из нее влияния веса (рис. 5.27).

```
library(modelr)

mod <- lm(log(price) ~ log(carat), data = diamonds)

diamonds2 <- diamonds %>%
  add_residuals(mod) %>%
  mutate(resid = exp(resid))

ggplot(data = diamonds2) +
  geom_point(mapping = aes(x = carat, y = resid))
```

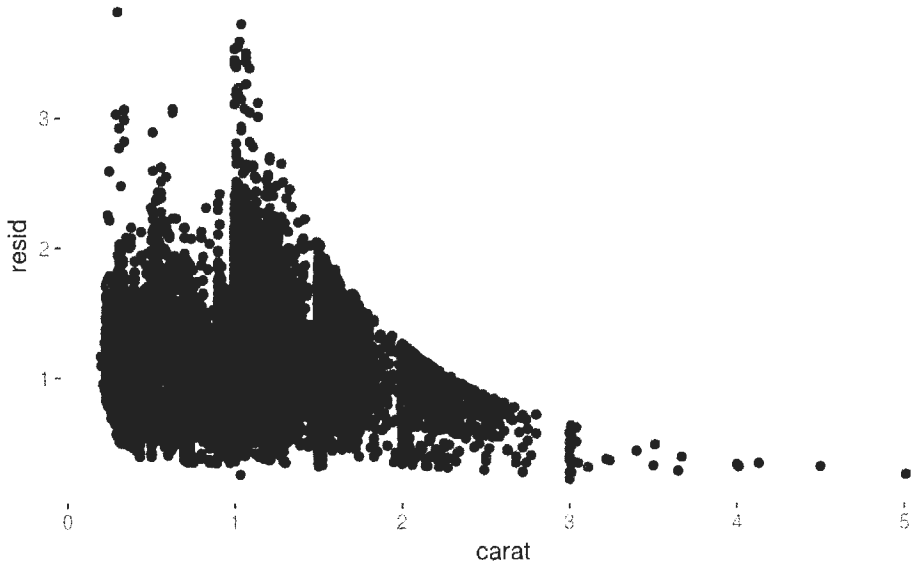


Рис. 5.27

Исключив влияние сильной взаимосвязи переменных carat и price, вы увидите, что соотношение между переменными cut и price соответствует ожидаемому — для алмазов сравнимого размера более высокому качеству соответствует более высокая цена (рис. 5.28).

```
ggplot(data = diamonds2) +
  geom_boxplot(mapping = aes(x = cut, y = resid))
```

О том, как работают модели и пакет **modelr**, вы узнаете в части IV. Рассмотрение моделей отложено до последней части книги, потому что вам будет легче понять, что такое модели и как они работают, уже после того, как вы овладеете инструментами подготовки данных и приобретете навыки программирования.

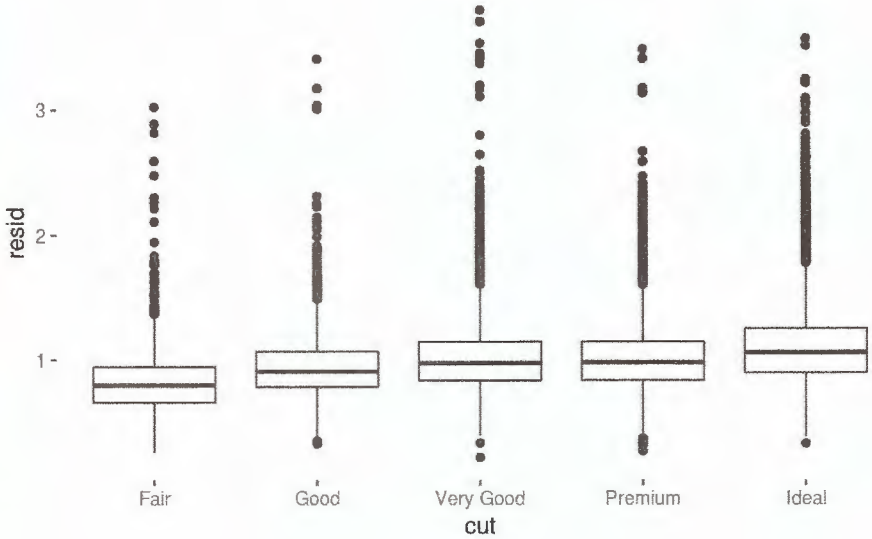


Рис. 5.28

Вызовы функций ggplot2

Как только закончатся вводные главы, мы начнем использовать более компактную запись кода выражений **ggplot2**. До сих пор мы все записывали в явном виде, что было оправдано, пока вы только знакомились с этим пакетом.

```
ggplot(data = faithful, mapping = aes(x = eruptions)) +
  geom_freqpoly(binwidth = 0.25)
```

Как правило, первые один или два аргумента любой функции являются самыми важными, поэтому вы должны знать их на память. Первые два аргумента функции `ggplot()` — это `data` и `mapping`, а первые два аргумента функции `aes()` — это `x` и `y`. В оставшейся части книги мы не будем указывать перечисленные имена. Это избавит нас от лишнего ввода и благодаря уменьшению объема шаблонного текста облегчит распознавание различий в коде графиков. Это очень важный технический аспект программирования, к рассмотрению которого мы еще вернемся в главе 15.

Придадим предыдущему коду более компактный вид.

```
ggplot(faithful, aes(eruptions)) +
  geom_freqpoly(binwidth = 0.25)
```

Иногда мы будем ставить в конец канала преобразования данных команду построение графика. Внимательно следите за переходом от знака `%>%` к знаку `+`. Мне бы хотелось, чтобы это не было необходимым, но, к сожалению, пакет **ggplot2** создавался до того, как появились каналы.

```
diamonds %>%
  count(cut, clarity) %>%
  ggplot(aes(clarity, cut, fill = n)) +
    geom_tile()
```

Дополнительная информация

Если вы захотите больше узнать о работе пакета **ggplot2**, рекомендуем обратиться к книге *ggplot2* (`http://ggplot2.org/book/`). Недавно она была обновлена, так что теперь в нее включен код **dplyr** и **tidyr**, что дает большой простор для исследования всех аспектов визуализации. К сожалению, эта книга не предоставляется бесплатно, но если вы связаны с каким-либо университетом, то, вероятно, сможете бесплатно получить ее электронную версию через платформу SpringerLink (`https://link.springer.com`).

Еще один полезный ресурс — книга *R Graphics Cookbook* (Winston Chang). Большая часть ее содержимого доступна по адресу `http://www.cookbook-r.com/Graphs/`.

Кроме того, можем порекомендовать также книгу *Graphical Data Analysis with R* (Antony Unwin). Содержащийся в ней материал аналогичен тому, который рассмотрен в этой главе, но объем книги позволил ее автору изложить материал более подробно.

ГЛАВА 6

Организация рабочего процесса: проекты

В один прекрасный день у вас может возникнуть необходимость временно прервать работу с R и вернуться к ней только на следующий день, чтобы заняться чем-то другим. Вы также можете быть заняты параллельным выполнением сразу нескольких анализов данных, в каждом из которых используется R, и вы хотели бы хранить все результаты отдельно. В другой раз вам может понадобиться выполнить импорт данных в R из внешнего источника, а затем переслать полученные численные результаты и графики своим коллегам. В подобных ситуациях, которые часто возникают на практике, вам придется принимать следующие два типа решений.

1. Что в вашем анализе является "достоянием" , т.е. что именно вы хотите сохранить в качестве очередной порции информации, касающейся вашего исследования?
2. Где следует хранить текущие результаты анализа?

Что является вашим "достоянием"

Начинающий пользователь R вполне может считать своим "достоянием" среду (т.е. объекты, список которых отображается на панели Environment). Однако в долгосрочной перспективе будет намного лучше, если вы будете считать своим "достоянием" написанные вами R-скрипты.

Располагая своими R-скриптами (и файлами данных), вы всегда сможете восстановить среду. Восстановить же R-скрипты из своей среды гораздо труднее! Вы должны будете либо заново ввести большой объем кода, набрав его по памяти (и допустив при этом множество опечаток), либо тщательно покопаться в записях истории команд R.

Чтобы воспитать в вас правильное отношение к скриптам, настоятельно рекомендуем запретить сохранение рабочего пространства RStudio в промежутках между рабочими сеансами, используя соответствующие настройки (рис. 6.1).

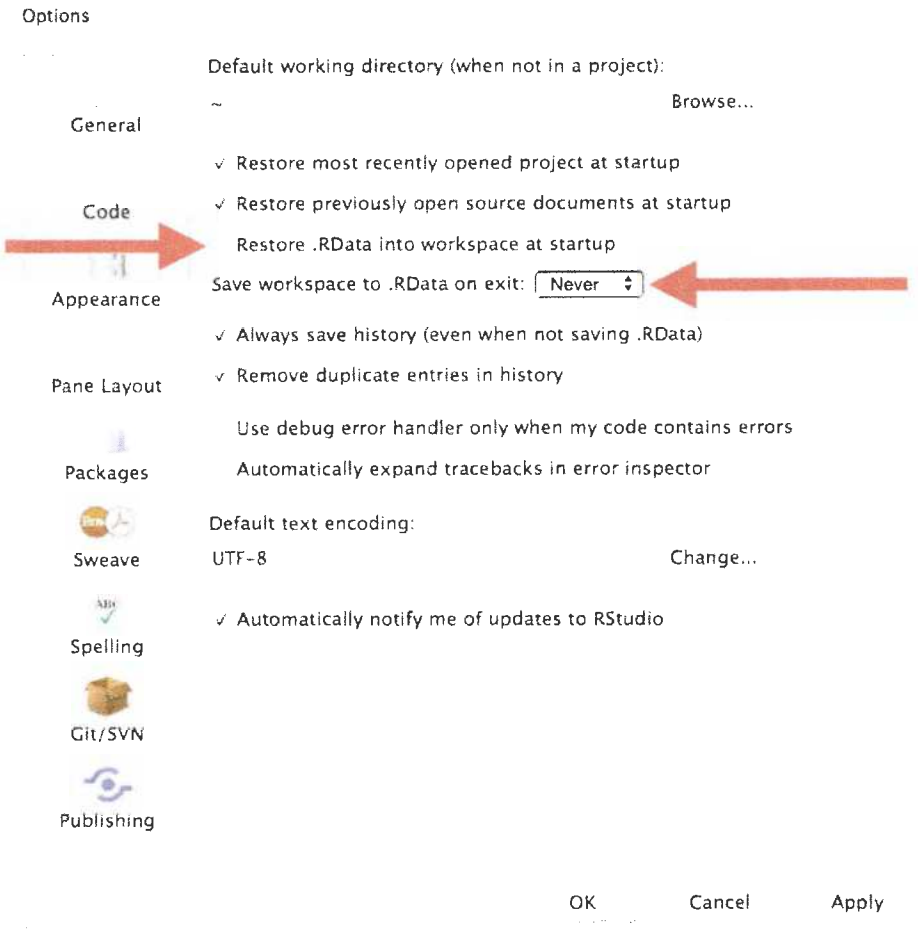


Рис. 6.1

Это доставит вам кратковременные неудобства, поскольку теперь при повторном запуске RStudio не сможет вспомнить результаты выполнения кода, который вы запускали в последний раз. Но эти неудобства сэкономят вам в будущем массу времени, поскольку приучат вас фиксировать все важные взаимодействия в своем коде. Нет ничего хуже, чем обнаружить три месяца спустя, что вы сохранили лишь важные результаты в своем рабочем пространстве, но не сам код, с помощью которого они были получены.

В RStudio предусмотрена замечательная пара комбинаций клавиш, совместное использование которых служит залогом того, что вы зафиксировали важные части своего кода, отображаемого в окне редактора:

- нажмите комбинацию клавиш <Ctrl+Shift+F10> для перезапуска RStudio;
- нажмите комбинацию клавиш <Ctrl+Shift+S> для перезапуска текущего сценария.

Мы следуем этому шаблону поведения сотни раз в неделю.

Где находится ваш анализ

В R существует удобное понятие *рабочего каталога*. Именно в нем R ищет файлы, которые вы хотите загрузить, и в него же помещает файлы, которые вы хотите сохранить. RStudio отображает текущий рабочий каталог в верхней части окна консоли (рис. 6.2).

Console Find in Files R Markdown

Рис. 6.2

Вы сможете узнать имя рабочего каталога с помощью функции `getwd()`.

```
getwd()
#> [1] "/Users/hadley/Documents/r4ds/r4ds"
```

На начальном этапе можете оставить в качестве рабочего каталога R свой домашний каталог (личный каталог пользователя, которым, например, в Windows служит папка `C:/Пользователи/имя_пользователя/Документы`). Но у вас за плечами уже шесть глав книги, и вы уже далеко не новичок. Вскоре вы начнете организовывать свои аналитические проекты в каталогах, а в процессе работы с проектом будете устанавливать соответствующий каталог в качестве рабочего.

Существует также возможность задавать рабочий каталог прямо из R:

```
setwd("/path/to/my/CoolProject")
```

Однако вы *никогда не должны так поступать*, поскольку для этого существует лучший способ, применяемый профессионалами.

Пути и каталоги

Работа с путями доступа и каталогами немного осложнена тем, что существуют два базовых стиля указания путей: Mac/Linux и Windows. Они отличаются в трех разных аспектах.

- Наиболее важное отличие состоит в том, какой символ используется в качестве разделителя между составляющими пути. В операционных системах Mac и Linux таким символом служит косая черта (например, `plots/diamonds.pdf`), тогда как в Windows — обратная косая черта (например, `plots\diamonds.pdf`). R может работать с любым из этих разделителей (независимо от того, какую платформу вы используете), но, к сожалению, обратная косая черта (`\`) имеет в R специальный смысл, и для того, чтобы указать в пути одну обратную косую черту, вам потребуется ввести ее дважды! Это немного усложняет жизнь, поэтому рекомендуем всегда придерживаться стиля Linux/Mac, т.е. использовать косую черту (/).
- Абсолютные пути (т.е. пути, ведущие к одному и тому же расположению, независимо от местоположения рабочего каталога) выглядят поразному в разных операционных системах. В Windows они начинаются с буквы диска (например, `C:`) или с двух символов обратной косой черты (например, `\\servername`), а в Mac/Linux они начинаются с косой черты (например, `/users/hadley`). Никогда не используйте абсолютные пути в своих сценариях, поскольку это затрудняет совместное использование скриптов: ни на одном компьютере не будет точно такой же конфигурации каталогов, как и на вашем.
- Последнее (незначительное) отличие связано с тем, на какой каталог указывает символ "тильда" (~). Символ ~ — удобное сокращенное обозначение домашнего каталога. Но в Windows, где понятие домашнего каталога по сути отсутствует, этот символ указывает на папку `Документы`.

Проекты RStudio

Специалисты по языку R хранят вместе все файлы, связанные с проектом: входные данные, R-скрипты, аналитические результаты, рисунки. Это

настолько разумная и распространенная практика, что в RStudio для нее предусмотрена встроенная поддержка посредством *проектов*.

Давайте создадим для вас проект, который вы будете использовать на протяжении оставшейся части книги. Щелкните на пунктах меню File⇨New Project (Файл⇨Создать проект), а затем выполните действия, показанные на рис. 6.3–6.5.

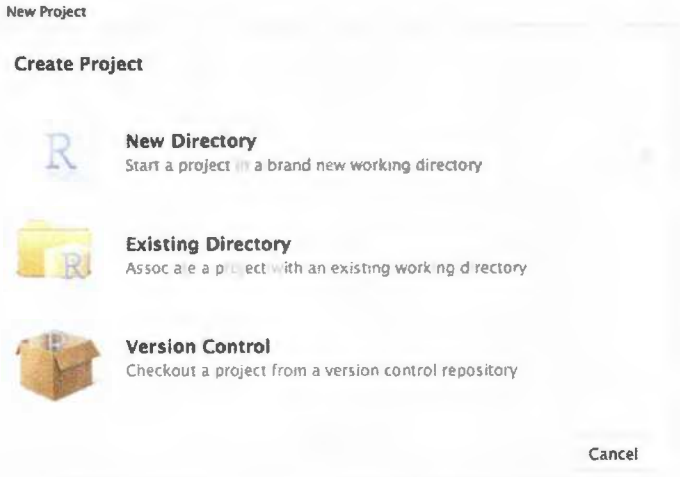


Рис. 6.3

Рис. 6.4

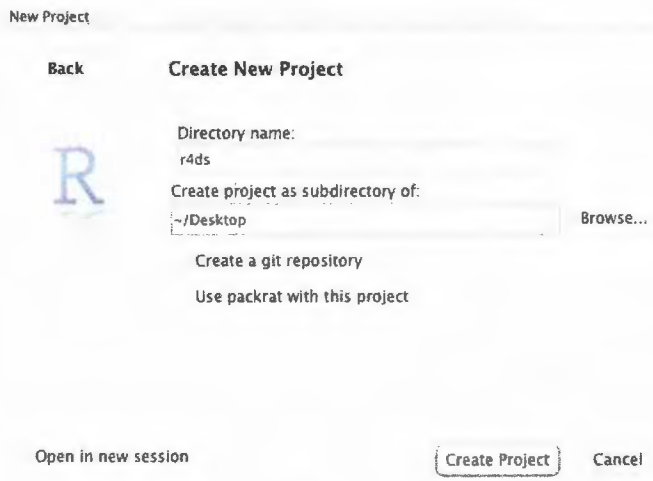


Рис. 6.5

Назовите свой проект `r4ds` и тщательно обдумайте, в какой подкаталог вы его поместите. Если вы не сохраните его в каталоге, который сможете запомнить, то в будущем вам будет трудно отыскать свой проект!

Завершив описанный процесс, вы получите новый проект RStudio, пригодный для этой книги. Убедитесь в том, что текущим рабочим каталогом является ваш "домашний" каталог.

```
getwd()
#> [1] /Users/hadley/Documents/r4ds/r4ds
```

Всякий раз, когда вы будете ссылаться на файл с помощью относительного пути, RStudio будет искать его здесь.

А теперь введите следующие команды в редакторе скриптов и сохраните файл, назвав его *diamonds.R*. Затем выполните весь сценарий, который сохранит файлы с расширениями PDF и CSV в каталоге вашего проекта. Пусть вас не волнуют детали — вы узнаете о них чуть позже.

```
library(tidyverse)

ggplot(diamonds, aes(carat, price)) +
  geom_hex()
ggsave("diamonds.pdf")

write_csv(diamonds, "diamonds.csv")
```

Выйдите из RStudio. Проверьте содержимое папки, связанной с вашим проектом, и найдите в ней файл с расширением *.Rproj*. Дважды щелкните на

нем для открытия проекта. Заметьте, что вы вернулись туда же, откуда вышли: тот же рабочий каталог и та же история команд, а все файлы, с которыми вы работали, по-прежнему открыты. Однако, коль скоро вы следовали нашим инструкциям, среда проекта будет совершенно пустой, тем самым гарантируя, что вы начинаете с чистого листа.

Выполните поиск файла *diamonds.pdf* способом, специфичным для установленной на вашем компьютере операционной системы, и вы найдете не только PDF-файл (в чем нет ничего удивительного), но и файл создавшего его скрипта (*diamonds.r*). Это огромное достижение! В один прекрасный день вы захотите сделать еще одну копию рисунка или просто понять, откуда он взялся. Если вы будете сохранять рисунки строго посредством R-кода, никогда не используя для этой цели мышь или буфер обмена, то воспроизвести результаты прежней работы для вас не составит большого труда.

Резюме

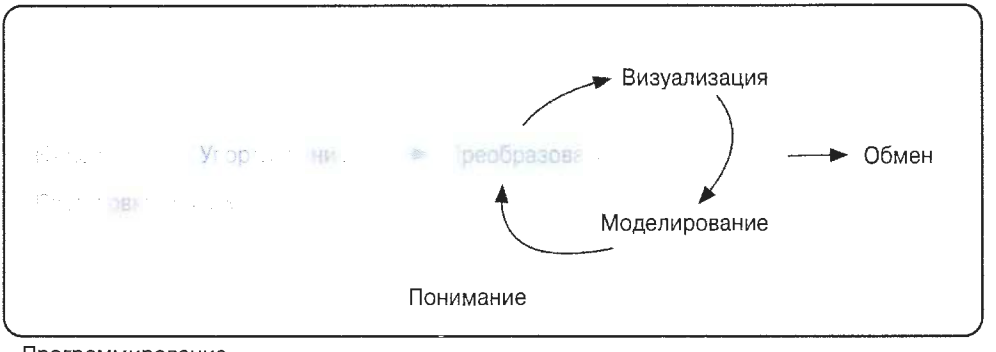
Ниже приведен краткий перечень тех возможностей работы с R-кодом, который предлагает RStudio. Итак, вы сможете:

- создавать проект RStudio для каждого проекта анализа данных;
- хранить в проекте файлы данных (о загрузке файлов в R речь пойдет в главе 8);
- хранить в проекте скрипты, редактировать их и выполнять по частям или целиком;
- сохранять в проекте свои выходные результаты (графики и очищенные данные);
- использовать только относительные пути, а не абсолютные.

Все необходимое будет собрано в одном месте и отделено от других проектов, с которыми вы работаете.

Подготовка данных

В этой части вы узнаете о подготовке данных — искусстве приведения исходных данных к виду, обеспечивающему возможность их визуализации и моделирования в среде R. Эта процедура очень важна: не выполнив ее, вы не сможете работать с собственными данными! Подготовка данных включает три этапа, представленных на приведенной ниже иллюстрации.



Программирование

Данная часть состоит из следующих глав.

- В главе 7 речь пойдет о видоизмененных фреймах данных, используемых в этой книге: *tibble-фреймах*. В ней рассказано о том, что отличает tibble-фреймы от обычных фреймов данных, и показано, как их можно создавать "вручную".
- В главе 8 вы узнаете о том, как перенести данные с диска в среду R. Основное внимание будет уделено прямоугольным форматам простых текстовых данных, но при этом будут указаны пакеты, позволяющие работать с другими типами данных.
- В главе 9 обсуждается аккуратизация данных — последовательный подход к сохранению данных, упрощающий их преобразование, визуализацию и моделирование. В ней изложены как базовые принципы, так и конкретные методы приведения данных к аккуратной форме.

Подготовка данных включает их преобразование, с чем вы уже успели частично ознакомиться в предыдущих главах. Теперь же мы сосредоточим внимание на особенностях работы с тремя специфическими типами данных, с которыми вы будете часто сталкиваться в своей практике.

- Глава 10 предоставит в ваше распоряжение инструменты для работы с несколькими взаимосвязанными наборами данных.
- В главе 11 вводятся регулярные выражения — мощный инструмент, используемый для манипулирования строками.
- В главе 12 показано, как сохранять категориальные данные в R. Этот тип данных используется в тех случаях, когда переменная имеет фиксированный набор возможных значений, а также тогда, когда требуется упорядочивать строки в порядке, отличном от алфавитного.
- В главе 13 вас ждет знакомство с основными инструментами, предназначенными для работы с датами и временем.

ГЛАВА 7

Создание tibble-фреймов с помощью пакета tibble

Введение

На протяжении всей книги мы работаем не с традиционными фреймами данных R (тип `data.frame`), а с так называемыми tibble-фреймами. Это те же фреймы данных, часть аспектов поведения которых была оптимизирована для упрощения обработки данных. R — язык с солидным стажем, и кое-что из того, что было в нем полезно десять-двадцать лет тому назад, сейчас устарело. Изменить базовые компоненты R, не ставя под угрозу работоспособность существующего кода, очень трудно, поэтому самые передовые инновации реализуются в пакетах. В этой главе будет рассмотрен пакет **tibble**, который предоставляет видоизмененные фреймы данных, немного упрощающие взаимодействие с библиотекой *tidyverse*. В большинстве случаев мы используем термины "tibble-фрейм" и "фрейм данных" как взаимозаменяемые; если же мы хотим подчеркнуть, что имеем в виду встроенные фреймы данных R, то будем ссылаться на тип `data.frame`.

Если после прочтения этой главы вы захотите получить более подробную информацию о tibble-фреймах, воспользуйтесь командой `vignette("tibble")`.

Необходимые ресурсы

В этой главе мы будем работать с пакетом **tibble**, являющимся частью ядра библиотеки *tidyverse*.

```
library(tidyverse)
```

Создание tibble-фреймов

Почти все функции, которые вы будете использовать в книге, создают tibble-фреймы, так как они являются одним из унифицирующих элементов библиотеки *tidyverse*. В большинстве других пакетов R используются обычные фреймы данных, поэтому иногда вам может потребоваться привести обычный фрейм данных к типу tibble-фрейма. Это можно сделать с помощью функции `as_tibble()`.

```
as_tibble(iris)
#> # A tibble: 150 × 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>          <dbl>       <dbl>        <dbl>       <dbl>  <fctr>
#> 1          5.1         3.5          1.4         0.2  setosa
#> 2          4.9         3.0          1.4         0.2  setosa
#> 3          4.7         3.2          1.3         0.2  setosa
#> 4          4.6         3.1          1.5         0.2  setosa
#> 5          5.0         3.6          1.4         0.2  setosa
#> 6          5.4         3.9          1.7         0.4  setosa
#> # ... with 144 more rows
```

С помощью функции `tibble()` можно создать новый tibble-фрейм из отдельных векторов. В необходимых случаях функция `tibble()` использует циклическое повторение (recycling) входных данных длиной 1 до достижения размеров tibble-фрейма и позволяет ссылаться на только что созданные переменные, как показано ниже.

```
tibble(
  x = 1:5,
  y = 1,
  z = x ^ 2 + y
)
#> # A tibble: 5 × 3
#> ?     x     y     z
#> ? <int> <dbl> <dbl>
#> 1     1     1     2
#> 2     2     1     5
#> 3     3     1    10
#> 4     4     1    17
#> 5     5     1    26
```

Если вы уже знакомы с функцией `data.frame()`, то обратите внимание на то, что функция `tibble()` делает намного меньше: она никогда не изменяет тип входных данных (например, никогда не преобразует строки в факторы!), никогда не изменяет имена переменных и не создает имена строк.

В случае tibble-фреймов вполне возможно использовать имена столбцов, являющиеся недопустимыми с точки зрения R, — так называемые *несинтаксические* имена. Например, они могут начинаться не с буквы или могут содержать необычные символы, в частности, пробелы. При обращении к таким именам их следует заключать в обратные апострофы (`).

```
tb <- tibble(
  `:)` = "smile",
  ` ` = "space",
  `2000` = "number"
)

tb
#> # A tibble: 1 × 3
#>   `:)`   ` ` `2000`
#>   <chr> <chr>  <chr>
#> 1 smile space number
```

Обратные апострофы понадобятся вам и при работе с аналогичными переменными в других пакетах, таких как **ggplot2**, **dplyr** и **tidyr**.

Еще один способ создания tibble-фреймов заключается в использовании функции `tribble()` (от англ. *transposed tibble* — транспонированный tibble-фрейм). Функция `tribble()` приспособлена для ввода данных в коде: заголовки столбцов определяются формулами (начинаются с символа ~), а элементы данных разделены запятыми. Это дает возможность располагать данные небольшого объема в удобочитаемой форме.

```
tribble(
  ~x, ~y, ~z,
  #--|--|----
  "a", 2, 3.6,
  "b", 1, 8.5
)
#> # A tibble: 2 × 3
#>       x     y     z
#>   <chr> <dbl> <dbl>
#> 1     a     2   3.6
#> 2     b     1   8.5
```

В подобных случаях часто добавляют комментарий (строка, начинающаяся с символа #), чтобы сделать очевидными заголовки.

Сравнение tibble-фреймов с фреймами data.frame

Различия в использовании tibble-фреймов и классических фреймов данных типа data.frame в основном касаются двух видов операций: вывода на печать и извлечения поднаборов.

Вывод на печать

Для tibble-фреймов предусмотрен более рациональный способ вывода на печать, при котором отображаются лишь первые 10 строк и все столбцы, умещающиеся в пределах экрана. Это намного упрощает работу с большими наборами данных. Для каждого столбца в дополнение к его имени выводится также тип содержащихся в нем данных — замечательная особенность, заимствованная из функции str().

```
tibble(
  a = lubridate::now() + runif(1e3) * 86400,
  b = lubridate::today() + runif(1e3) * 30,
  c = 1:1e3,
  d = runif(1e3),
  e = sample(letters, 1e3, replace = TRUE)
)
#> # A tibble: 1,000 × 5
#>                     a          b     c     d     e
#>                 <dttm>     <date> <int> <dbl> <chr>
#> 1 2016-10-10 17:14:14 2016-10-17     1 0.368     h
#> 2 2016-10-11 11:19:24 2016-10-22     2 0.612     n
#> 3 2016-10-11 05:43:03 2016-11-01     3 0.415     l
#> 4 2016-10-10 19:04:20 2016-10-31     4 0.212     x
#> 5 2016-10-10 15:28:37 2016-10-28     5 0.733     a
#> 6 2016-10-11 02:29:34 2016-10-24     6 0.460     v
#> # ... with 994 more rows
```

Tibble-фреймы спроектированы таким образом, чтобы вы не могли случайно переполнить консоль при выводе на печать больших фреймов данных. Но иногда вам может потребоваться расширенный вывод, а не тот, который предусмотрен по умолчанию. Для этого есть несколько возможностей.

Во-первых, вы можете вывести на печать фрейм данных путем явного вызова функции print() с передачей ей количества выводимых строк (n) и ширины (width) печати. При width = Inf будут отображаться все столбцы.

```
nycflights13::flights %>%
  print(n = 10, width = Inf)
```

Кроме того, можно управлять поведением при печати, заданным по умолчанию, с помощью группы настроек `options`.

- `options(tibble.print_max = n, tibble.print_min = m)`: если количество строк больше `m`, вывести на печать лишь `n` строк. Чтобы всегда отображались все строки, используйте вариант `options(dplyr.print_min = Inf)`.
- Чтобы всегда выводить на печать все столбцы, независимо от ширины экрана, используйте вариант `options(tibble.width = Inf)`.

Чтобы увидеть полный список возможных опций, выполните команду `package?tibble`.

Наконец, можно воспользоваться встроенным средством просмотра RStudio для получения прокручиваемого представления полного набора данных. Часто эту возможность удобно использовать в конце длинной цепочки манипуляций.

```
nycflights13::flights %>%
  View()
```

Извлечение поднаборов

Все инструменты, которые вы изучали до сих пор, работали с полными фреймами данных. Если требуется извлечь одну переменную, необходимо использовать другие средства — операторы `$` и `[[`. Оператор `[[` обеспечивает доступ к элементам по имени и по индексу; оператор `$` обеспечивает доступ только по имени, но требует ввода меньшего количества символов.

```
df <- tibble(
  x = runif(5),
  y = rnorm(5)
)

# Извлечение элемента по имени
df$x
#> [1] 0.434 0.395 0.548 0.762 0.254
df[["x"]]
#> [1] 0.434 0.395 0.548 0.762 0.254

# Извлечение элемента по индексу
df[[1]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

Применение этих операций в канале требует использования специального заместителя (.).

```
df %>% .$x
#> [1] 0.434 0.395 0.548 0.762 0.254
df %>% .[["x"]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

По сравнению с объектами `data.frame` tibble-фреймы отличаются большей строгостью: они никогда не удовлетворяются частичным соответствием и генерируют сообщение об ошибке, если столбец, к которому вы пытаетесь получить доступ, не существует.

Взаимодействие с разработанным ранее кодом

Некоторые из функций прежних версий не работают с tibble-фреймами. Столкнувшись с такой ситуацией, используйте функцию `as.data.frame()` для обратного преобразования tibble-фрейма в `data.frame`.

```
class(as.data.frame(tb))
#> [1] "data.frame"
```

Основной причиной того, что некоторые из прежних функций не работают с tibble-фреймами, является оператор `[`. В этой книге мы не используем его особенно часто, поскольку функции `dplyr::filter()` и `dplyr::select()` позволяют делать то же самое с помощью более прозрачного кода (тем не менее дальнейшее обсуждение этого оператора можно найти в разделе “Извлечение элементов” главы 16). При работе с базовыми фреймами данных R оператор `[` иногда возвращает фрейм данных, а иногда вектор. В случае tibble-фреймов он всегда возвращает другой tibble-фрейм.

Упражнения

1. Как определить, что объект является tibble-фреймом? (Подсказка: попытайтесь вывести на печать набор данных `mtcars`, являющийся обычным фреймом данных.)
2. Сопоставьте результаты применения приведенных ниже операций к фрейму данных типа `data.frame` и эквивалентному tibble-фрейму. Чем они различаются? Какие особенности поведения фрейма данных, заданные по умолчанию, могут вас раздражать?

```
df <- data.frame(abc = 1, xyz = "a")
df$x
df[, "xyz"]
df[, c("abc", "xyz")]
```

3. Если имя переменной сохранено в объекте (например, `var <- "mpg"`), то как бы вы могли извлечь переменную с этим именем из tibble-фрейма?

4. Попрактикуйтесь в использовании несинтаксических имен, выполнив следующие операции применительно к представленному ниже фрейму данных:

   а) извлеките переменную `1`;

   б) создайте точечный график для переменных `1` и `2`;

   в) создайте новый столбец с именем `3`, значения которого равны значениям столбца `2`, деленным на значения столбца `1`;

   г) измените названия столбцов на `one`, `two` и `three`.

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

5. Что делает функция `tibble::enframe()`? В каких случаях вы могли бы ее использовать?

6. Какая опция управляет количеством дополнительных имен столбцов, выводимых в нижнем колонтитуле tibble-фрейма?

ГЛАВА 8

Импорт данных с помощью пакета readr

Введение

Работа с наборами данных, предоставляемыми пакетами R, — отличный способ ознакомления с инструментами, используемыми в науке о данных, но в какой-то момент вам захочется сделать паузу в обучении и приступить к работе с собственными данными. В этой главе вы узнаете о том, как прочитать данные из простого текстового файла в R. Здесь мы лишь слегка коснемся вопросов, связанных с импортом данных, но многие из принципов, которые мы обсудим, распространяются на данные, существующие в других формах. Глава завершается ссылками на ряд пакетов, которые могут быть полезными при работе с другими типами данных.

Необходимые ресурсы

В этой главе вы научитесь загружать файлы в R с помощью пакета **readr**, входящего в состав ядра библиотеки *tidyverse*.

```
library(tidyverse)
```

Приступаем к работе

Большинство функций пакета **readr** связано с перемещением данных из простых текстовых файлов во фреймы данных.

- Функция `read_csv()` читает текстовые файлы с запятой в качестве разделителя, функция `read_csv2()` — файлы с двоеточием в качестве разделителя (типичны для стран, в которых запятая используется

для группирования цифр по разрядам), функция `read_tsv()` — файлы с символом табуляции в качестве разделителя, а функция `read_delim()` — файлы с любым разделителем.

- Функция `read_fwf()` читает текстовые файлы с фиксированной шириной полей. Вы можете определить поля либо по их ширине с помощью функции `fwf_widths()`, либо по их позиции с помощью функции `fwf_positions()`. Функция `read_table()` читает популярную разновидность файлов с фиксированной шириной полей, в которых столбцы разделены символом пробела.
- Функция `read_log()` читает файлы журналов в стиле Apache. (Вам также стоит обратить внимание на пакет **webreadr**, который создан поверх функции `read_log()` и предоставляет намного больше полезных инструментов.)

Все эти функции имеют схожий синтаксис: стоит научиться работать с одной из них, и вы легко сможете использовать любую другую. В оставшейся части главы наше внимание будет сосредоточено на функции `read_csv()`. С одной стороны, это обусловлено тем, что CSV-файлы являются наиболее популярной формой хранения данных, а с другой — тем, что, освоив функцию `read_csv()`, вы легко примените свои знания для работы с другими функциями пакета **readr**.

Первый аргумент функции `read_csv()` играет наиболее важную роль: это путь к файлу, который вы хотите прочитать.

```
heights <- read_csv("data/heights.csv")
#> Parsed with column specification:
#> cols(
#>   earn = col_double(),
#>   height = col_double(),
#>   sex = col_character(),
#>   ed = col_integer(),
#>   age = col_integer(),
#>   race = col_character()
#> )
```

Когда вы выполните функцию `read_csv()`, она выведет спецификацию столбцов, содержащую название и тип данных каждого столбца. Это важная особенность пакета **readr**, к рассмотрению которой мы еще вернемся в разделе “Синтаксический анализ файлов”.

Кроме того, у вас имеется возможность предоставить встроенный CSV-файл. Это полезно как для экспериментирования с пакетом readr, так и для создания воспроизводимых примеров, которыми вы сможете поделиться с другими людьми.

```
read_csv("a,b,c
1,2,3
4,5,6")
#> # A tibble: 2 × 3
#>       a     b     c
#>   <int> <int> <int>
#>       1     1     2
#>       2     4     5
```

В обоих случаях функция read_csv() использует для названий столбцов первую строку данных — весьма распространенное соглашение. Существуют две ситуации, в которых у вас может возникнуть потребность в изменении такого поведения.

- Некоторые файлы начинаются с нескольких строк метаданных. Вы можете использовать аргумент skip = n, чтобы пропустить первые n строк, или аргумент comment = "#", чтобы пропустить все строки, начинающиеся, например, с символа #.

```
read_csv("The first line of metadata
  The second line of metadata
  x,y,z
  1,2,3", skip = 2)
#> # A tibble: 1 × 3
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     2     3

read_csv("# A comment I want to skip
  x,y,z
  1,2,3", comment = "#")
#> # A tibble: 1 × 3
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     2     3
```

- Данные могут не иметь названий столбцов. Чтобы не обрабатывать первую строку как строку заголовков, используйте аргумент col_names = FALSE; в этом случае в заголовки столбцов будут автоматически помещены последовательные метки от X1 до Xn.

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
#> # A tibble: 2 × 3
#>      X1    X2    X3
#>   <int> <int> <int>
#> 1     1     2     3
#> 2     4     5     6
```

("\n" — удобное сокращение для добавления символа перехода на новую строку. Об этом и других экранированных символах вы узнаете больше в разделе "Основы работы со строками" главы 11.)

Еще один возможный вариант — передать с помощью аргумента col_names символьный вектор, из которого будут взяты названия столбцов.

```
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
#> # A tibble: 2 × 3
#>       x     y     z
#>   <int> <int> <int>
#> 1     1     2     3
#> 2     4     5     6
```

Другой параметр, который обычно приходится настраивать, — это na. Он определяет значение (или значения), используемое для представления отсутствующих значений в файле.

```
read_csv("a,b,c\n1,2,.", na = ".")
#> # A tibble: 1 × 3
#>       a     b     c
#>   <int> <int> <chr>
#> 1     1     2  <NA>
```

Это все, что вам требуется знать для того, чтобы прочитать примерно 75% CSV-файлов, с которыми вы столкнетесь на практике. Вы легко сможете адаптировать приобретенные навыки для чтения файлов с разделителями — символами табуляции с помощью функции read_tsv() и файлов с полями фиксированной ширины с помощью функции read_fwf(). Чтобы научиться читать файлы более сложных форматов, вам следует более подробно ознакомиться с тем, каким образом пакет **readr** выполняет синтаксический анализ (парсинг) столбцов, превращая их в R-векторы.

Сравнение с базовыми возможностями R

Если до этого вы уже работали с R, то, возможно, вас удивляет, почему мы не используем функцию read.csv(). В пользу того, что функции **readr** более

предпочтительны по сравнению с их базовыми эквивалентами, можно привести следующие веские доводы.

- Обычно эти функции работают гораздо быстрее (примерно в 10 раз), чем их базовые эквиваленты. Для длительно выполняющихся заданий предусмотрен индикатор выполнения, позволяющий вам быть в курсе того, что происходит. Если вас интересует исключительно скорость, попробуйте использовать функцию `data.table::fread()`. Она не очень хорошо сочетается с библиотекой *tidyverse*, но работает заметно быстрее.
- Эти функции создают tibble-фреймы, а кроме того, не преобразуют символьные векторы в факторы, не используют имена строк и не изменяют имена столбцов, что обычно служит источником раздражения при использовании базовых функций R.
- Их поведение более воспроизводимо. Базовые функции R наследуют некоторые особенности поведения из установленной на компьютере операционной системы и переменных окружения, поэтому импортируемый код, который работает на вашем компьютере, на другом компьютере может не работать.

Упражнения

1. Какую функцию вы использовали бы для чтения файла, в котором поля разделены символами `|`?
2. Какие еще общие аргументы, кроме `file`, `skip` и `comment`, имеют функции `read_csv()` и `read_tsv()`?
3. Какие аргументы функции `read_fwf()` являются наиболее важными?
4. Иногда строки в CSV-файле содержат запятые. Чтобы они не стали источником проблем, их необходимо заключать в двойные (`"`) или одинарные (`'`) кавычки. В соответствии с общепринятым соглашением функция `read_csv()` предполагает использование кавычек, и если вы хотите использовать для этой цели другой символ, используйте функцию `read_delim()`. Какие аргументы необходимо указать, чтобы прочитать следующий текст во фрейм данных?

```
"x,y\n1,'a,b'"
```

5. Укажите, что не так в каждом из следующих встроенных CSV-файлов. Что произойдет, если вы попытаетесь выполнить этот код?

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a,b\n\"1")
read_csv("a,b\n1,2\na,b")
read_csv("a;b\n1;3")
```

Синтаксический анализ векторов

Прежде чем мы углубимся в детали того, каким образом пакет **readr** читает файлы с диска, мы должны несколько отвлечься и уделить внимание функциям семейства `parse_*()`, обеспечивающим синтаксический анализ (парсинг) входных данных. Эти функции принимают символьный вектор и возвращают вектор специализированного типа, например логический, целочисленный или вектор дат.

```
str(parse_logical(c("TRUE", "FALSE", "NA")))
#>  logi [1:3] TRUE FALSE NA
str(parse_integer(c("1", "2", "3")))
#>  int [1:3] 1 2 3
str(parse_date(c("2010-01-01", "1979-10-14")))
#>  Date[1:2], format: "2010-01-01" "1979-10-14"
```

Функции этого семейства полезны уже сами по себе, но одновременно они выступают в качестве строительных кирпичиков для пакета **readr**. После того как вы ознакомитесь с работой отдельных синтаксических анализаторов (парсеров) в этом разделе, мы вернемся на исходные позиции в следующем разделе, где будет показано, как все они вместе обеспечивают парсинг полного файла.

Как и все функции, входящие в состав библиотеки *tidyverse*, функции `parse_*()` унифицированы. Их первый аргумент — это символьный вектор, подлежащий анализу, а аргумент `na` указывает, какие строки должны обрабатываться как отсутствующие значения.

```
parse_integer(c("1", "231", ".", "456"), na = ".")
#> [1]   1 231  NA 456
```

В случае неуспешного парсинга какого-либо элемента выводится соответствующее сообщение.

```
x <- parse_integer(c("123", "345", "abc", "123.45"))
#> Warning: 2 parsing failures.
```

```
#> row col               expected actual
#>   3  -- an integer                abc
#>   4  -- no trailing characters    .45
```

Такие элементы помещаются в вывод как отсутствующие (NA).

```
x
#> [1] 123 345  NA  NA
#> attr(,"problems")
#> # A tibble: 2 × 4
#>     row   col               expected actual
#>   <int> <int>                  <chr>  <chr>
#> 1     3    NA             an integer    abc
#> 2     4    NA no trailing characters    .45
```

Если ошибок парсинга много, вы можете получить полный их набор с помощью функции problems(). Она возвращает tibble-фрейм, которым можно манипулировать с помощью пакета **dplyr**.

```
problems(x)
#> # A tibble: 2 × 4
#>     row   col               expected actual
#>   <int> <int>                  <chr>  <chr>
#> 1     3    NA             an?integer    abc
#> 2     4    NA no trailing characters    .45
```

Эффективное выполнение парсинга требует знания того, какие парсеры имеются в вашем распоряжении и как они обрабатывают различные типы входных данных. Чаще всего используются девять парсеров, перечень которых приводится ниже.

- parse_logical() и parse_integer() — парсинг логических и целочисленных значений соответственно. Никаких затруднений при работе с ними в основном не возникает, поэтому далее мы на них останавливаться не будем.
- parse_double() — строго числовой парсер, тогда как parse_number() — гибкий числовой парсер. Работать с ними сложнее, чем вы могли бы подумать, поскольку в разных странах числа записывают по-разному.
- parse_character() — символьный парсер, который кажется настолько простым, что могли бы возникнуть сомнения в том, что он нужен, если бы не одно осложняющее обстоятельство: существование различных кодировок символов.

- `parse_factor()` — создает факторы, или структуры данных, которые R использует для представления категориальных переменных, т.е. переменных, имеющих известный фиксированный набор возможных значений.
- `parse_datetime()`, `parse_date()` и `parse_time()` — позволяют анализировать значения даты и времени, заданные в соответствии с различными спецификациями. Эти парсеры отличаются наибольшей сложностью ввиду существования множества различных форматов записи дат и времени.

В следующих разделах эти парсеры рассмотрены более подробно.

Числа

Парсинг чисел был бы совсем простой задачей, если бы не три осложняющих обстоятельства.

- В разных частях света люди записывают числа по-разному. Так, для разделения дробной и целой частей вещественных чисел в одних странах используют точку (`.`), в других — запятую (`,`).
- Числа часто окружены другими символами, обеспечивающими определенный контекст, например `"$1000"` или `"10%"`.
- Числа часто содержат разделители групп разрядов (группирующие символы), облегчающие чтение больших чисел, например `"1,000,000"`, причем в разных странах для этой цели могут использоваться разные символы.

Чтобы справиться с первой проблемой, в пакет **readr** введено понятие "локали" — объекта, определяющего правила парсинга, которые соответствуют различным региональным стандартам. При парсинге чисел наиболее важным параметром является символ, используемый в качестве десятичного разделителя. Вы можете использовать вместо него другой символ, создав новую локаль и задав соответствующий символ с помощью аргумента `decimal_mark`.

```
parse_double("1.23")
#> [1] 1.23
parse_double("1,23", locale = locale(decimal_mark = ","))
#> [1] 1.23
```

Используемая в пакете **readr** локаль ориентирована на региональные стандарты США, поскольку на это ориентирована среда R в целом (например, базовая документация R написана с использованием американского варианта

английского языка). В качестве альтернативы можно было бы попытаться определить параметры, используемые по умолчанию установленной на компьютере операционной системой. Однако хорошая реализация этой идеи довольно затруднительна, и к тому же, что более важно, это сделало бы ваш код хрупким: даже если бы он работал на вашем компьютере, ваш коллега в другой стране, получив код по электронной почте, мог бы столкнуться с тем, что он не работает.

Вторая проблема решается с помощью функции `parse_number()`: она игнорирует символы, не являющиеся цифрами, которые располагаются непосредственно перед числом и после него. Это особенно удобно в случае денежных и процентных значений, но также может быть использовано для извлечения чисел из текста.

```
parse_number("$100")
#> [1] 100
parse_number("20%")
#> [1] 20
parse_number("It cost $123.45")
#> [1] 123
```

Последняя проблема решается путем совместного использования функции `parse_number()` и локали, которая задает символ, группирующий цифры по разрядам.

```
# Используется в США
parse_number("$123,456,789")
#> [1] 1.23e+08

# Используется во многих европейских странах
parse_number(
  "123.456.789",
  locale = locale(grouping_mark = ".")
)
#> [1] 1.23e+08

# Используется в Швейцарии
parse_number(
  "123'456'789",
  locale = locale(grouping_mark = "'")
)
#> [1] 1.23e+08
```

Строки

Можно было бы подумать, что функция `parse_character()` должна быть действительно простой — ведь от нее требуется только вернуть входную информацию. К сожалению, жизнь не так проста, поскольку одну и ту же строку можно представить множеством способов. Чтобы разобраться в этом, необходимо углубиться в детали того, как строки представляются в компьютере. В R, чтобы получить строку в том виде, в каком она хранится в памяти компьютера, следует использовать функцию `charToRaw()`.

```
charToRaw("Hadley")
#> [1] 48 61 64 6c 65 79
```

Каждое шестнадцатеричное число представляет байт информации: `48` — это `H`, `61` — это `a` и т.п. Отображение шестнадцатеричного числа в символ называется кодированием, и используемый в данном случае код называется *ASCII* (от англ. *American Standard Code for Information Interchange* — американский стандартный код для обмена информацией). Кодировка ASCII отлично подходит для представления символов английского алфавита, о чем нетрудно догадаться по ее полному названию.

В случае языков, отличных от английского, не все так просто. На заре компьютерных технологий существовало множество конкурирующих стандартов кодирования символов, отсутствующих в английском языке, и для корректной интерпретации строки требовалось знать не только значения, но и способ кодировки. Например, двумя распространенными кодировками являются Latin1 (стандарт ISO-8859-1; используется в западноевропейских языках) и Latin2 (стандарт ISO-8859-2; используется в восточноевропейских языках). В кодировке Latin1 байту `b1` соответствует символ “±”, но в кодировке Latin2 ему соответствует символ “ą”! К счастью, в наши дни существует универсальный стандарт UTF-8, обеспечивающий кодирование практически любого символа, используемого в национальных алфавитах, а также множество дополнительных символов (например, эмодзи). Пакет **readr** использует UTF-8 во всех случаях: предполагается, что считываемые символы имеют кодировку UTF-8, и она же используется при записи символов. Это неплохой вариант для использования по умолчанию, но если данные были получены в старых системах, которым кодировка UTF-8 была неизвестна, он не сработает. Столкнувшись с подобным случаем, вы увидите, что выведенные на печать строки выглядят очень странно. Иногда странными могут быть лишь один-два символа, но иногда весь текст будет представлять собой сплошную абракадабру.

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
```

С этой проблемой можно справиться, указав кодировку в функции `parse_character()`.

```
parse_character(x1, locale = locale(encoding = "Latin1"))
#> [1] "El Niño was particularly bad this year"
parse_character(x2, locale = locale(encoding = "Shift-JIS"))
#> [1] "こんにちは"
```

Как определить, какая кодировка правильная? Если вам повезет, вы найдете эту информацию в документации к данным. К сожалению, такое случается редко, поэтому пакет **readr** предоставляет функцию `guess_encoding()`. Эта функция не является панацеей, и она работает лучше, если объем текста большой (в отличие от приведенных выше примеров), но начинать с нее будет вполне разумно. Будьте готовы к тому, что вам придется перепробовать несколько кодировок, прежде чем удастся найти правильную.

```
guess_encoding(charToRaw(x1))
#> encoding confidence
#> 1 ISO-8859-1       0.46
#> 2 ISO-8859-9       0.23
guess_encoding(charToRaw(x2))
#>   encoding confidence
#> 1   KOI8-R       0.42
```

Первым аргументом функции `guess_encoding()` может быть либо путь к файлу, либо, как в данном случае, “сырой” вектор (что может быть полезным, если строка уже перемещена в R).

Кодирование символов — обширная и сложная тема, и здесь мы лишь слегка коснулись ее. Подробное разъяснение по этому вопросу вы сможете найти по адресу `http://kunststube.net/encoding/`.

Факторы

Факторы используются в R для представления категориальных значений, т.е. значений, образующих известный набор. Стоит вам предоставить функции `parse_factor()` вектор с неизвестным значением, и вы получите предупреждающее сообщение об этом.

```
fruit <- c("apple", "banana")
parse_factor(c("apple", "banana", "bananana"), levels = fruit)
#> Warning: 1 parsing failure.
#> row col           expected   actual
```

```
#>   3  -- value in level set bananana
#> [1] apple  banana <NA>
#> attr(,"problems")
#> # A tibble: 1 × 4
#>     row   col           expected   actual
#>   <int> <int>              <chr>    <chr>
#> 1     3    NA value in level set bananana
#> Levels: apple banana
```

Но если проблемных записей много, то зачастую проще оставить их в виде символьных векторов, а затем очистить данные с помощью инструментов, о которых вы узнаете в главах 11 и 12.

Календарные даты, календарные дата и время, время суток

Выбор одного из трех парсеров зависит от того, какой тип значений вам необходим: *дата* (количество дней, истекших с 01-01-1970), *дата и время* (количество секунд, истекших после полуночи 01-01-1970) или *время* (количество секунд, истекших после полуночи). При вызове соответствующих функций без указания дополнительных аргументов предполагается следующее.

- Функция `parse_datetime()` ожидает значение, определяющее дату и время в соответствии со стандартом ISO8601 — это международный стандарт, предписывающий задание даты в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС (год, месяц, день, часы, минуты, секунды).

  ```
  parse_datetime("2010-10-01T2010")
  #> [1] "2010-10-01 20:10:00 UTC"

  # Если время опущено, автоматически устанавливается полночь
  parse_datetime("20101010")
  #> [1] "2010-10-10 UTC"
  ```

  Это самый важный стандарт указания даты/времени, и если вы часто работаете с датами и временем, обратитесь к статье по адресу `https://en.wikipedia.org/wiki/ISO_8601`.

- Функция `parse_date()` ожидает значение в одном из форматов ГГГГ-ММ-ДД или ГГГГ/ММ/ДД.

  ```
  parse_date("2010-10-01")
  #> [1] "2010-10-01"
  ```

- Функция `parse_time()` ожидает значение в формате ЧЧ:ММ:СС (последний компонент необязателен) с необязательным спецификатором a.m./p.m (до полудня/после полудня).

  ```
  library(hms)
  parse_time("01:10 am")
  #> 01:10:00
  parse_time("20:10:01")
  #> 20:10:01
  ```

  Ввиду отсутствия в базовых пакетах R встроенного специализированного класса, предназначенного для работы со временем суток, мы используем для этого класс, предоставляемый пакетом **hms**.

Если эти предусмотренные по умолчанию форматы не подходят для ваших данных, определите собственный формат, включающий перечисленные ниже элементы.

Год

`%Y` (4 цифры).

`%y` (2 цифры; 00–69 → 2000–2069, 70–99 → 1970–1999).

Месяц

`%m` (2 цифры).

`%b` (сокращенное название, например “Jan”).

`%B` (полное название, например “January”).

День

`%d` (2 цифры).

`%e` (необязательный ведущий пробел).

Время

`%H` (часовой формат 0–23).

`%I` (часовой формат 0–12; следует использовать вместе с `%p`).

`%p` (индикатор a.m./p.m. — до полудня/после полудня).

`%M` (минуты).

`%S` (целочисленные секунды).

`%OS` (вещественные секунды).

`%Z` (часовой пояс, например `America/Chicago`). Примечание: внимательно следите за использованием сокращений. Если вы находитесь

в Америке, учтите, что "EST" — это часовой пояс Канады, для которого не предусмотрен переход на летнее время. К рассмотрению часовых поясов мы еще вернемся в главе 13.

`%z` (смещение от всемирного скоординированного времени UTC, например +0800).

Нецифровые символы

`%.` (пропуск одного нецифрового символа).

`%*` (пропуск любого количества нецифровых символов).

Наилучший способ определить корректный формат — создать несколько примеров в символьном векторе и протестировать их с помощью одной из функций-анализаторов.

```
parse_date("01/02/15", "%m/%d/%y")
#> [1] "2015-01-02"
parse_date("01/02/15", "%d/%m/%y")
#> [1] "2015-02-01"
parse_date("01/02/15", "%y/%m/%d")
#> [1] "2001-02-15"
```

В случае использования элементов формата `%b` или `%B` с названиями месяцев не на английском языке необходимо задать аргумент `lang` функции `locale()`. Просмотрите список встроенных языков, вызвав функцию `date_names_langs()`, и вызовите функцию `date_names_lang()` с соответствующим аргументом или, если нужный язык отсутствует, создайте собственные названия с помощью функции `date_names()`.

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
#> [1] "2015-01-01"
```

Упражнения

1. Какие из аргументов функции `locale()` наиболее важные?
2. Что произойдет, если вы попытаетесь установить с помощью аргументов `decimal_mark` и `grouping_mark` один и тот же символ для десятичного и группирующего разделителей? Что произойдет со значением по умолчанию аргумента `grouping_mark`, если вы установите для аргумента `decimal_mark` значение `","`? Что произойдет со значением по умолчанию аргумента `decimal_mark`, если вы установите для аргумента `grouping_mark` значение `"."`?

3. Мы не обсуждали опции `date_format` и `time_format` функции `locale()`. Что они делают? Придумайте пример, в котором они были бы полезны.
4. Если вы живете за пределами США, то создайте новый объект локали, инкапсулирующий настройки для типов файлов, которые вам чаще всего приходится читать.
5. В чем различие между функциями `read_csv()` и `read_csv2()`?
6. Какие кодировки наиболее распространены в Европе? Какие кодировки чаще всего используются в Азии? Выясните это, воспользовавшись поисковиком Google.
7. Сгенерируйте подходящую строку формата для парсинга каждого из следующих значений даты и времени.

```
d1 <- "January 1, 2010" d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14"    # 30 декабря 2014 года
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

Синтаксический анализ файлов

Теперь, когда вы узнали о том, как выполняется синтаксический анализ векторов, мы хотим вернуться немного назад и рассказать вам о средствах пакета **readr**, предназначенных для синтаксического анализа файлов. Новыми для вас будут следующие два момента, которые мы рассмотрим в этом разделе:

- как пакет **readr** автоматически определяет тип каждого столбца;
- как изменить параметры, заданные по умолчанию.

Стратегия

Для определения типа каждого из столбцов в пакете **readr** используется эвристический подход: читаются первые 1000 строк, и для определения типов столбцов используются (умеренно-консервативные) эвристики (эвристические алгоритмы). Этот процесс можно эмулировать с помощью символьного вектора, используя функцию `guess_parser()`, которая возвращает наилучшую догадку, и функцию `parse_guess()`, которая использует эту догадку для анализа столбца.

```
guess_parser("2010-10-01")
#> [1] "date"
guess_parser("15:01")
#> [1] "time"
guess_parser(c("TRUE", "FALSE"))
#> [1] "logical"
guess_parser(c("1", "5", "9"))
#> [1] "integer"
guess_parser(c("12,352,561"))
#> [1] "number"

str(parse_guess("2010-10-10"))
#>  Date[1:1], format: "2010-10-10"
```

Эвристика испытывает каждый из перечисленных ниже типов, останавливая этот процесс в случае совпадения.

logical

Содержит только значения “F”, “T”, “FALSE” и “TRUE”.

integer

Содержит только цифровые символы (и -)

double

Содержит только допустимые вещественные числа (включая числа наподобие `4.5e-5`).

number

Содержит допустимые вещественные числа с разделителями групп разрядов.

time

Соответствует формату `time_format`, заданному по умолчанию.

date

Соответствует формату `date_format`, заданному по умолчанию.

date-time

Любая дата в формате ISO8601.

Если ни одно из этих правил не применимо, столбец остается в виде вектора строк.

Проблемы

Перечисленные ниже предположения, используемые по умолчанию, не всегда работают для крупных файлов. В таком случае возможны следующие проблемы.

- Первая тысяча строк может быть не показательной, вследствие чего догадки пакета **readr** окажутся недостаточно общими. Например, у вас может быть столбец вещественных значений, в первых 1000 строках которого содержатся только целые числа.
- Столбец может содержать множество отсутствующих значений. Если первая тысяча строк содержит лишь значения NA, то пакет **readr** сочтет, что это символьный вектор, тогда как вам, вероятно, хотелось бы выбрать более специфический способ их обработки.

Пакет **readr** содержит тестовый CSV-файл, который иллюстрирует обе указанные проблемы.

```
challenge <- read_csv(readr_example("challenge.csv"))
=> Parsed with column specification:
=> cols(
=>   x = col_integer(),
=>   y = col_character()
=> )
=> Warning: 1000 parsing failures.
=>  row col               expected             actual
=> 1001   x no trailing characters .23837975086644292
=> 1002   x no trailing characters .41167997173033655
=> 1003   x no trailing characters .7460716762579978
=> 1004   x no trailing characters .723450553836301
=> 1005   x no trailing characters .614524137461558
=> .... ... ..................... .................
=> See problems(...) for more details.
```

(Обратите внимание на использование функции `readr_example()`, которая находит путь к одному из файлов, включенных в пакет.)

Вывод состоит из двух частей: столбца спецификаций, сгенерированного при просмотре первых 1000 строк, и первых пяти неудачно разобранных строк. В подобных случаях целесообразно явно вызывать функцию `problems()` для получения дополнительной информации о причинах того, почему анализ файла не удался.

```
problems(challenge)
=> # A tibble: 1,000 × 4
=>     row   col               expected             actual
=>   <int> <chr>                  <chr>              <chr>
=> 1  1001     x no trailing characters .23837975086644292
=> 2  1002     x no trailing characters .41167997173033655
=> 3  1003     x no trailing characters  .7460716762579978
=> 4  1004     x no trailing characters   .723450553836301
```

```
#> 5  1005     x no trailing characters   .614524137461558
#> 6  1006     x no trailing characters   .473980569280684
#> # ... with 994 more rows
```

Неплохая стратегия — последовательно обрабатывать столбец за столбцом, пока проблемы не исчезнут. В данном случае видно, что проблемы связаны со столбцом `x` — в нем за целочисленными значения следуют хвостовые символы. Это наводит на мысль, что необходимо использовать парсер вещественных чисел.

Начнем с того, что скопируем и вставим спецификацию столбца в первоначальный вызов.

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_integer(),
    y = col_character()
  )
)
```

После этого можно исправить тип столбца `x`.

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_character()
  )
)
```

Это устраняет первую проблему, но если мы присмотримся к нескольким последним строкам, то заметим, что они содержат даты, сохраненные в символьном векторе.

```
tail(challenge)
#> # A tibble: 6 × 2
#>       x          y
#>   <dbl>      <chr>
#> 1 0.805 2019-11-21
#> 2 0.164 2018-03-29
#> 3 0.472 2014-08-04
#> 4 0.718 2015-08-16
#> 5 0.270 2020-02-04
#> 6 0.608 2019-01-06
```

Это можно исправить, указав, что `y` является столбцом дат.

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_date()
```

```
tail(challenge)
=> # A tibble: 6 × 2
=>       x          y
=>   <dbl>     <date>
=> 1 0.805 2019-11-21
=> 2 0.164 2018-03-29
=> 3 0.472 2014-08-04
=> 4 0.718 2015-08-16
=> 5 0.270 2020-02-04
=> 6 0.608 2019-01-06
```

Для каждой функции `parse_xyz()` имеется соответствующая функция `col_xyz()`. Функция `parse_xyz()` используется в тех случаях, когда данные уже находятся в R в виде символьного вектора, а функция `col_xyz()` — когда нужно сообщить пакету **readr**, как загружать данные.

Настоятельно рекомендуем всегда предоставлять типы столбцов, отталкиваясь от вывода, обеспечиваемого пакетом **readr**. Это является гарантией того, что ваш сценарий импорта данных всегда будет согласованным и воспроизводимым. Если вы будете полагаться на используемые по умолчанию эвристические типы и ваши данные изменятся, пакет **readr** все равно будет их читать. Если вам нужен действительно строгий контроль типов, используйте функцию `stop_for_problems()`: в случае возникновения проблем с разбором данных она сгенерирует ошибку и прекратит выполнение сценария.

Другие стратегии

Существует несколько других общих стратегий, облегчающих синтаксический анализ файлов.

- В предыдущем примере нам просто не повезло: стоило нам просмотреть файл всего на одну строку больше, и мы могли бы корректно выполнить анализ одним махом.

```
challenge2 <- read_csv(
                readr_example("challenge.csv"),
                guess_max = 1001
              )
```

```
#> Parsed with column specification:
#> cols(
#>   x = col_double(),
#>   y = col_date(format = "")
#> )
challenge2
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
```

- Иногда проще всего диагностировать проблему, прочитав все столбцы как символьные векторы.

```
challenge2 <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character())
)
```

Такой подход особенно полезно использовать в сочетании с функцией `type_convert()`, которая применяет эвристики парсинга к символьным столбцам фрейма данных.

```
df <- tribble(
  ~x, ~y,
  "1", "1.21",
  "2", "2.32",
  "3", "4.56"
)
df
#> # A tibble: 3 × 2
#>       x     y
#>   <chr> <chr>
#> 1     1  1.21
#> 2     2  2.32
#> 3     3  4.56

# Обратите внимание на типы столбцов
type_convert(df)
#> Parsed with column specification:
#> cols(
#>   x = col_integer(),
```

```
#>   y = col_double()
#> )
#> # A tibble: 3 × 2
#>       x     y
#>   <int> <dbl>
#> 1     1  1.21
#> 2     2  2.32
#> 3     3  4.56
```

- При чтении очень больших файлов имеет смысл установить для параметра `n_max` меньшее значение, например 10000 или 100000. Это ускорит ваши итерации по устранению распространенных проблем.
- В случае более сложных проблем иногда проще прочитать данные в символьный вектор строк с помощью функции `read_lines()` или даже в символьный вектор длиной 1 с помощью функции `read_file()`. После этого вы сможете применить методы синтаксического анализа строк, с которыми ознакомитесь позже, для обработки более сложных форматов.

Запись в файл

Обычно пакет **readr** поставляется с двумя полезными функциями, обеспечивающими запись данных на диск: `write_csv()` и `write_tsv()`. Обе функции увеличивают вероятность того, что впоследствии данные будут читаться с диска корректно, по следующим причинам:

- они всегда используют кодировку UTF-8;
- всегда сохраняют календарные даты и время в формате ISO8601, и поэтому их последующий парсинг на других компьютерах не будет вызывать проблем.

Если вы хотите экспортировать CSV-файл в Excel, используйте функцию `write_excel_csv()` — она записывает в начале файла специальную метку (маркер последовательности байтов), информирующую Excel о том, что вы используете кодировку UTF-8.

Самые важные аргументы — `x` (сохраняемый фрейм данных) и `path` (расположение, в котором должен быть сохранен файл). Кроме того, вы можете сообщить с помощью аргумента `na`, как должны записываться отсутствующие значения, и указать, следует ли добавить запись в конец уже существующего файла:

```
write_csv(challenge, "challenge.csv")
```

Заметьте, что при сохранении фрейма в формате CSV информация о типах теряется.

```
challenge
#> # A tibble: 2,000 × 2
#>       x      y
#>   <dbl> <date>
#> 1   404   <NA>
#> 2  4172   <NA>
#> 3  3004   <NA>
#> 4   787   <NA>
#> 5    37   <NA>
#> 6  2332   <NA>
#> # ... with 1,994 more rows
write_csv(challenge, "challenge-2.csv")
read_csv("challenge-2.csv")
#> Parsed with column specification:
#> cols(
#>   x = col_double(),
#>   y = col_character()
#> )
#> # A tibble: 2,000 × 2
#>       x     y
#>   <dbl> <chr>
#> 1   404  <NA>
#> 2  4172  <NA>
#> 3  3004  <NA>
#> 4   787  <NA>
#> 5    37  <NA>
#> 6  2332  <NA>
#> # ... with 1,994 more rows
```

Это делает CSV-файлы несколько ненадежными в отношении кеширования промежуточных результатов — вы должны заново создавать спецификации столбцов при каждой их загрузке. В подобных ситуациях можно поступать двояким образом.

- Можете воспользоваться функциями `write_rds()` и `read_rds()`, которые служат оболочками для базовых функций `saveRDS()` и `readRDS()`. Они сохраняют данные в настраиваемом двоичном формате R, носящем название RDS.

  ```
  write_rds(challenge, "challenge.rds")
  read_rds("challenge.rds")
  #> # A tibble: 2,000 × 2
  ```

```
#>        x     y
#>    <dbl><date>
#> 1    404  <NA>
#> 2   4172  <NA>
#> 3   3004  <NA>
#> 4    787  <NA>
#> 5     37  <NA>
#> 6   2332  <NA>
#> # ... with 1,994 more rows
```

- Пакет **feather** реализует быстрый двоичный формат файлов, совместимый с другими языками программирования.

```
library(feather)
write_feather(challenge, "challenge.feather")
read_feather("challenge.feather")
#> # A tibble: 2,000 x 2
#>        x     y
#>    <dbl><date>
#> 1    404  <NA>
#> 2   4172  <NA>
#> 3   3004  <NA>
#> 4    787  <NA>
#> 5     37  <NA>
#> 6   2332  <NA>
#> # ... with 1,994 more rows
```

Обычно пакет **feather** работает быстрее RDS и применим вне R. RDS поддерживает столбцы-списки (о которых речь пойдет в главе 20), чего нельзя сказать о текущей версии **feather**.

Другие типы данных

Для перемещения данных других типов в R мы в первую очередь рекомендуем перечисленные ниже пакеты библиотеки *tidyverse*. Несомненно, они не совершенны, но вам стоит начать с них.

- Пакет **haven** читает файлы приложений SPSS, Stata и SAS.
- Пакет **readxl** читает файлы Excel (как *.xls*, так и *.xlsx*).
- Пакет **DBI** совместно с серверами баз данных (например, RMySQL, RSQLite, RPostgreSQL и др.) обеспечивает выполнение SQL-запросов к базам данных и возвращение фреймов данных.

Для иерархических данных вам подойдут пакеты **jsonlite** (написан Джероеном Умсом) для работы с форматом JSON и **xml2** для работы с форматом XML. Дженни Брайан разработала несколько отличных примеров, которые можно найти по адресу `https://jennybc.github.io/purrr-tutorial/`.

Для работы с другими типами файлов обратитесь к руководству по импорту/экспорту данных в R и воспользуйтесь пакетом **rio**.

ГЛАВА 9

Аккуратизация данных с помощью пакета tidyr

Введение

"Все счастливые семьи похожи друг на друга, каждая несчастливая семья несчастлива по-своему".

Лев Толстой

"Все опрятные наборы данных похожи друг на друга, каждый неряшливый набор данных неряшлив по-своему".

Хэдли Уикем

В этой главе вы познакомитесь с последовательным подходом к организации своих данных в R, реализация которого обеспечивает поддержание данных в так называемой *аккуратной* форме. Аккуратизация данных, т.е. приведение их к такой форме, требует выполнения определенной предварительной работы, но в долгосрочной перспективе это себя окупает. Имея в своем распоряжении аккуратные данные и набор инструментов для их аккуратизации, предоставляемый пакетами библиотеки *tidyverse*, вы сможете уделять гораздо больше внимания аналитике интересующих вас данных.

В этой главе вы узнаете о том, что такое аккуратные данные, и научитесь работать с ними с помощью инструментов, входящих в состав пакета **tidyr**. Для более подробного ознакомления с лежащей в основе аккуратизации данных теорией можем порекомендовать статью *Tidy Data*, опубликованную в *Journal of Statistical Software* (`http://www.jstatsoft.org/v59/i10/paper`).

Необходимые ресурсы

В этой главе мы будем использовать пакет **tidyr**, предоставляющий группу инструментов, которые облегчают приведение в порядок неряшливых наборов данных. Пакет **tidyr** входит в состав ядра библиотеки *tidyverse*.

```
library(tidyverse)
```

Аккуратные данные

Одни и те же базовые данные можно представить множеством способов. Приведенный ниже пример иллюстрирует четыре разных способа представления данных. Каждый набор данных содержит одни и те же значения переменных `country`, `year`, `population` и `cases`, но значения в них организованы по-разному.

```
table1
#> # A tibble: 6 × 4
#>       country  year  cases population
#>         <chr> <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3      Brazil  1999  37737  172006362
#> 4      Brazil  2000  80488  174504898
#> 5       China  1999 212258 1272915272
#> 6       China  2000 213766 1280428583
table2
#> # A tibble: 12 × 4
#>       country  year       type     count
#>         <chr> <int>      <chr>     <int>
#> 1 Afghanistan  1999      cases       745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000      cases      2666
#> 4 Afghanistan  2000 population  20595360
#> 5      Brazil  1999      cases     37737
#> 6      Brazil  1999 population 172006362
#> # ... with 6 more rows
table3
#> # A tibble: 6 × 3
#>       country  year              rate
#> *       <chr> <int>             <chr>
#> 1 Afghanistan  1999      745/19987071
#> 2 Afghanistan  2000     2666/20595360
#> 3      Brazil  1999   37737/172006362
#> 4      Brazil  2000   80488/174504898
```

```
#> 5       China  1999 212258/1272915272
#> 6       China  2000 213766/1280428583

# Данные распределены по двум tibble-фреймам
table4a  # cases
#> # A tibble: 3 × 3
#>       country `1999` `2000`
#> *       <chr>  <int>  <int>
#> 1 Afghanistan    745   2666
#> 2      Brazil  37737  80488
#> 3       China 212258 213766

table4b  # population
#> # A tibble: 3 × 3
#>       country     `1999`     `2000`
#> *       <chr>      <int>      <int>
#> 1 Afghanistan   19987071   20595360
#> 2      Brazil  172006362  174504898
#> 3       China 1272915272 1280428583
```

Здесь все представления относятся к одним и тем же базовым данным, но не со всеми ими работать одинаково легко. Средства библиотеки *tidyverse* проще всего использовать по отношению к аккуратному набору данных.

Существуют три правила, соблюдение которых делает набор данных аккуратным:

каждая переменная должна иметь собственный столбец;

каждое наблюдение должно иметь собственную строку;

каждое значение должно иметь собственную ячейку.

Эти правила проиллюстрированы на рис. 9.1.

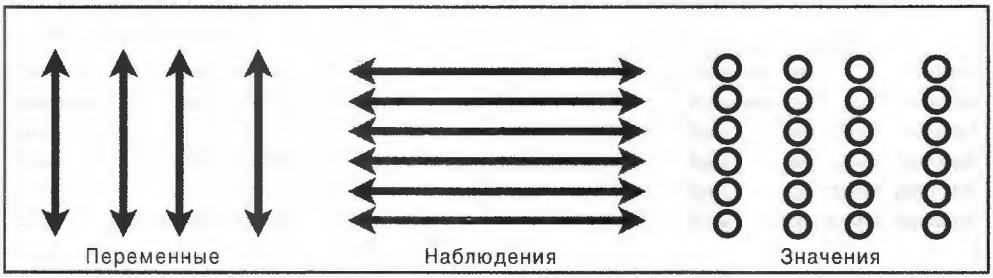


Рис. 9.1

Все три правила взаимосвязаны, поскольку соблюдение только двух из них невозможно. Эта взаимосвязь приводит к еще более простому набору практических инструкций.

1. Помещайте каждый набор данных в tibble-фрейм.
2. Помещайте каждую переменную в столбец.

В данном примере аккуратным является только набор данных `table1`. Это единственное представление, в котором каждый столбец является переменной.

Почему так важно, чтобы ваши данные были аккуратными? Соблюдение этого требования обеспечивает следующие два преимущества.

- Преимущество общего характера заключается в самом факте применения единообразного способа хранения данных. Последовательно используя согласованную структуру данных, легче изучить инструменты, с которыми при этом приходится работать, ввиду однородности базовых предположений относительно структуры данных.
- Преимущество узкоспециального характера обеспечивается размещением переменных в столбцах, поскольку это позволяет в полной мере использовать возможности векторизованной природы R. Как вам уже известно из главы 3, большинство встроенных функций R работает с векторными значениями. Поэтому преобразование аккуратных данных выполняется наиболее естественным образом.

Пакеты **dplyr**, **ggplot2** и другие, входящие в библиотеку *tidyverse*, проектировались с ориентацией на работу с аккуратными данными. Ниже приведено несколько примеров, демонстрирующих, как можно было бы работать с набором данных `table1` (рис. 9.2).

```
# Вычислить коэффициент заболеваемости на 10000 человек
table1 %>%
  mutate(rate = cases / population * 10000)
#> # A tibble: 6 × 5
#>       country  year  cases population  rate
#>         <chr> <int>  <int>      <int> <dbl>
#> 1 Afghanistan  1999    745   19987071 0.373
#> 2 Afghanistan  2000   2666   20595360 1.294
#> 3      Brazil  1999  37737  172006362 2.194
#> 4      Brazil  2000  80488  174504898 4.612
#> 5       China  1999 212258 1272915272 1.667
#> 6       China  2000 213766 1280428583 1.669

# Вычислить количество случаев заболевания в году
table1 %>%
  count(year, wt = cases)
```

```
#> # A tibble: 2 × 2
#>    year      n
#>   <int>  <int>
#> 1  1999 250740
#> 2  2000 296920

# Визуализировать изменения во времени
library(ggplot2)
ggplot(table1, aes(year, cases)) +
  geom_line(aes(group = country), color = "grey50") +
  geom_point(aes(color = country))
```

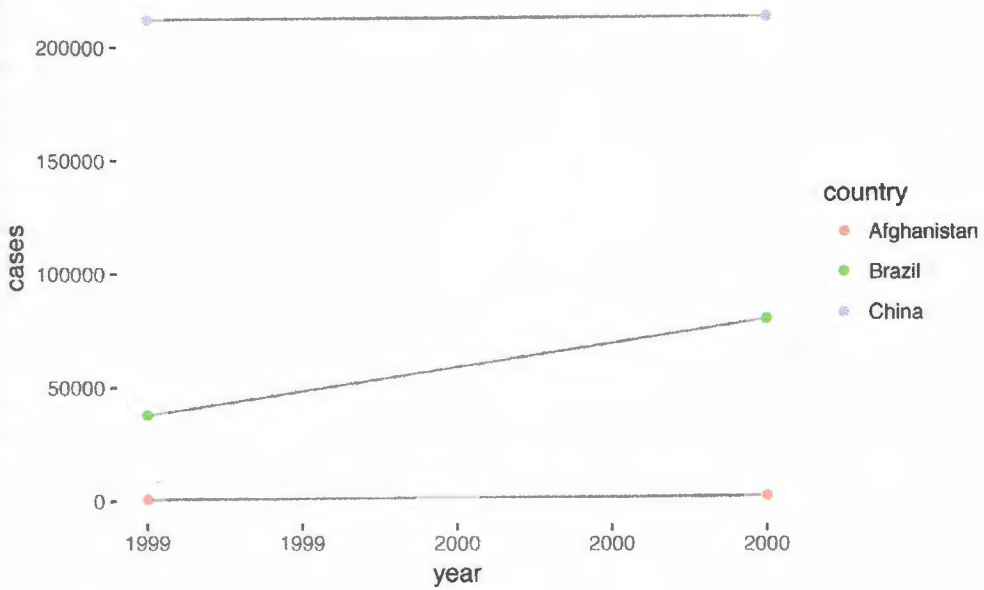


Рис. 9.2

Упражнения

1. Опишите простыми словами, как организованы переменные и наблюдения в каждой таблице-примере.
2. Вычислите значения коэффициента заболеваемости туберкулезом (`rate`) для наборов данных `table2` и `table4a` + `table4b`. Для этого вам придется выполнить четыре операции:
   а) извлеките количество случаев заболевания туберкулезом для каждого года и каждой страны;

б) извлеките соответствующие значения численности населения для каждого года и каждой страны;

в) разделите количество случаев заболевания на численность населения и умножьте полученный результат на 10000;

г) сохраните данные в соответствующем месте.

С каким представлением проще всего работать, а с каким труднее всего? Почему?

3. Постройте график, отображающий изменение количества случаев заболевания со временем, используя набор данных `table2` вместо набора данных `table1`. Что вам нужно сделать в первую очередь?

Рассредоточение и сведение столбцов

Принципы аккуратных данных кажутся настолько очевидными, что у вас могут возникнуть сомнения относительно того, что вам вообще когда-либо встретятся неаккуратные данные. К сожалению, в большинстве случаев вы будете сталкиваться с именно ними. На то существуют две основные причины.

- Большинство людей не знакомы с принципами аккуратных данных, и те, кто не посвящает большую часть времени работе с данными, вряд ли придут к ним самостоятельно.
- Данные часто организуют таким образом, чтобы облегчить какой-то способ их использования, например максимально упростить ввод, а не анализ.

Для большинства видов практического анализа вам придется что-либо делать в плане аккуратизации данных. Прежде всего следует выяснить, какие переменные и наблюдения имеются. Иногда это делается просто, но бывают случаи, когда вам потребуется консультация с людьми, занимавшимися первоначальным сбором данных. На следующем этапе обычно приходится решать одну из двух распространенных проблем:

- одна переменная может быть разнесена по нескольким столбцам;
- одно наблюдение может быть рассредоточено по нескольким строкам.

Типичные наборы данных страдают лишь одним из этих недостатков; столкнувшись одновременно с двумя указанными проблемами, можете

считать, что вам крупно не повезло! Для разрешения этих двух проблем вам понадобятся две очень важные функции пакета **tidyr**: `gather()` и `spread()`.

Сведение столбцов

Часто встречаются наборы данных, в которых некоторые из названий столбцов являются *значениями* переменных, а не их именами. Возьмем, к примеру, набор данных `table4a`; в нем названия столбцов `1999` и `2000` — это значения переменной `year`, а каждая строка представляет два наблюдения, а не одно.

```
table4a
#> # A tibble: 3 × 3
#>       country `1999` `2000`
#> *       <chr>  <int>  <int>
#> 1 Afghanistan    745   2666
#> 2      Brazil  37737  80488
#> 3       China 212258 213766
```

Чтобы привести набор данных, подобный этому, к аккуратному виду, мы должны свести столбцы в новую пару переменных. Для описания этой операции нам понадобятся следующие три параметра.

- Набор столбцов, которые представляют значения, а не переменные. В данном примере таковыми являются столбцы `1999` и `2000`.
- Имя переменной, значения которой образуют имена столбцов. Я назову этот параметр `key`, а переменной в данном примере является `year`.
- Имя переменной, значения которой рассредоточены по ячейкам. Я назову этот параметр `value`, а переменной в данном примере является количество случаев заболевания.

Все эти параметры используются при вызове функции `gather()` (рис. 9.3).

```
table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
#> # A tibble: 6 × 3
#>       country  year cases
#>         <chr> <chr> <int>
#> 1 Afghanistan  1999   745
#> 2      Brazil  1999 37737
#> 3       China  1999212258
#> 4 Afghanistan  2000  2666
#> 5      Brazil  2000 80488
#> 6       China  2000213766
```

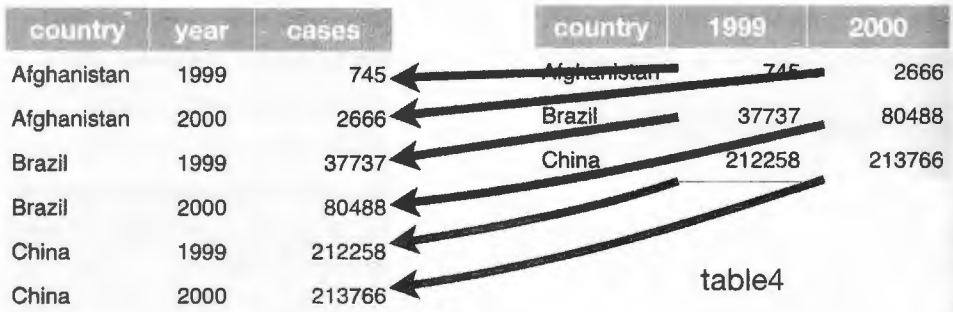


Рис. 9.3

Столбцы, которые необходимо свести, указываются с использованием нотации в стиле `dplyr::select()`. В данном случае таких столбцов только два, поэтому мы задаем их по отдельности. Обратите внимание на то, что "1999" и "2000" — несинтаксические имена (поскольку они начинаются не с буквы), в связи с чем их следует заключить в обратные апострофы. Чтобы восстановить в памяти другие способы выбора столбцов с помощью функции `select()`, см. главу 3.

Сведенные столбцы выпадают из конечного результата, в котором мы получаем два новых столбца: `key` и `value`. Все остальные соотношения между исходными переменными сохраняются. Аналогичным образом мы можем использовать функцию `gather()` для приведения к аккуратному виду набора данных. Единственным отличием будут другие значения, хранящиеся в ячейках.

```
table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
#> # A tibble: 6 × 3
#>       country  year population
#>         <chr> <chr>      <int>
#> 1 Afghanistan  1999   19987071
#> 2      Brazil  1999  172006362
#> 3       China  1999 1272915272
#> 4 Afghanistan  2000   20595360
#> 5      Brazil  2000  174504898
#> 6       China  2000 1280428583
```

Чтобы объединить приведенные к аккуратному виду наборы данных `table4a` и `table4b` в один tibble-фрейм, необходимо использовать функцию `dplyr::left_join()`, о которой вы узнаете больше в главе 10.

```
tidy4a <- table4a %>%
  gather(`1999`, `2000`, key = "year", value = "cases")
```

```
tidy4b <- table4b %>%
  gather(`1999`, `2000`, key = "year", value = "population")
left_join(tidy4a, tidy4b)
#> Joining, by = c("country", "year")
#> # A tibble: 6 × 4
#>       country  year  cases population
#>         <chr> <chr>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2      Brazil  1999  37737  172006362
#> 3       China  1999 212258 1272915272
#> 4 Afghanistan  2000   2666   20595360
#> 5      Brazil  2000  80488  174504898
#> 6       China  2000 213766 1280428583
```

Рассредоточение столбцов

Операция рассредоточения столбцов (*spreading*) противоположна их сбору. Ее используют в тех случаях, когда наблюдение распределено по нескольким строкам. Возьмем, например, набор данных `table2` — наблюдением является страна в определенном году, но каждое наблюдение распределено по двум строкам.

```
table2
#> # A tibble: 12 × 4
#>       country  year       type     count
#>         <chr> <int>      <chr>     <int>
#> 1 Afghanistan  1999      cases       745
#> 2 Afghanistan  1999 population  19987071
#> 3 Afghanistan  2000      cases      2666
#> 4 Afghanistan  2000 population  20595360
#> 5      Brazil  1999      cases     37737
#> 6      Brazil  1999 population 172006362
#> # ... with 6 more rows
```

Чтобы привести этот набор данных к аккуратному виду, прежде всего необходимо проанализировать представление так, как это делалось перед применением функции `gather()`. Однако на этот раз нам нужны только два приведенных ниже параметра.

- Столбец `key`, который содержит имена переменных. В данном примере это `type`.
- Столбец `value`, который содержит значения, формирует переменные. В данном примере это `count`.

Как только мы это определим, можно использовать функцию `spread()`, как показано в приведенном ниже коде и на рис. 9.4.

```
spread(table2, key = type, value = count)
#> # A tibble: 6 × 4
#>       country  year  cases population
#> *       <chr> <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
#> 3      Brazil  1999  37737  172006362
#> 4      Brazil  2000  80488  174504898
#> 5       China  1999 212258 1272915272
#> 6       China  2000 213766 1280428583
```

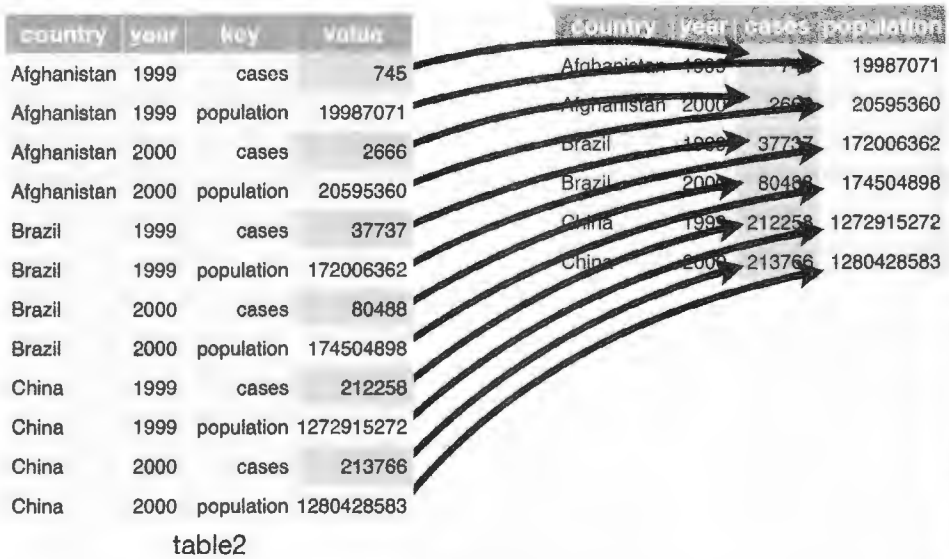


Рис. 9.4

Как вы, возможно, и сами догадались по наличию общих аргументов `key` и `value`, функции `spread()` и `gather()` взаимно дополняют друг друга. Функция `gather()` делает таблицы *уже* и длиннее, функция `spread()` делает длинные таблицы короче и шире.

Упражнения

1. Почему функции `gather()` и `spread()` не приводят к идеально симметричным результатам? Тщательно проанализируйте следующий пример.

```
stocks <- tibble(
  year  = c(2015, 2015, 2016, 2016),
```

```
  half = c(  1,   2,    1,   2),
  return = c(1.88, 0.59, 0.92, 0.17)
)
stocks %>%
  spread(year, return) %>%
  gather("year", "return", `2015`:`2016`)
```

(Подсказка: просмотрите типы переменных и продумайте имена столбцов.)

Обе функции, `spread()` и `gather()`, имеют аргумент `convert`. Что он делает?

2. Почему не работает этот код?

```
table4a %>%
  gather(1999, 2000, key = "year", value = "cases")
#> Error in eval(expr, envir, enclos):
#> Position must be between 0 and n
```

3. Почему не удается выполнить рассредоточение этого tibble-фрейма? Как можно было бы устранить проблему посредством добавления нового столбца?

```
people <- tribble(
  ~name,              ~key,    ~value,
  #-----------------|--------|------
  "Phillip Woods",   "age",       45,
  "Phillip Woods",   "height",   186,
  "Phillip Woods",   "age",       50,
  "Jessica Cordero", "age",       37,
  "Jessica Cordero", "height",   156
)
```

4. Приведите следующий простой tibble-фрейм к аккуратному виду. Нужно ли вам сводить его или рассредоточивать? Какие здесь переменные?

```
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
  "no",      20,    12
)
```

Разделение и объединение столбцов

К этому моменту вы знаете, как привести к аккуратному виду наборы данных `table2` и `table4`, но не `table3`. С набором данных `table3` связана

проблема другого рода: в нем имеется один столбец (rate), который содержит две переменные (cases и population). Для исправления этого недостатка нам нужна функция separate(). Вы также узнаете о дополняющей ее функции unite(), которая используется в том случае, если одна переменная распределена по нескольким столбцам.

Разделение столбцов

Функция separate() разносит один столбец по нескольким столбцам, выделяя значение всякий раз, когда встречается символ-разделитель. Рассмотрим набор данных table3.

```
table3
#> # A tibble: 6 × 3
#>       country  year              rate
#> *       <chr> <int>             <chr>
#> 1 Afghanistan  1999      745/19987071
#> 2 Afghanistan  2000     2666/20595360
#> 3      Brazil  1999   37737/172006362
#> 4      Brazil  2000   80488/174504898
#> 5       China  1999 212258/1272915272
#> 6       China  2000 213766/1280428583
```

Столбец rate одновременно содержит переменные cases и population, которые мы должны разделить. Функция separate() принимает имя столбца, переменные в котором подлежат разделению, и имена столбцов, в которые эти переменные должны быть вынесены, как показано в приведенном ниже коде и на рис. 9.5.

```
table3 %>%
  separate(rate, into = c("cases", "population"))
#> # A tibble: 6 × 4
#>       country  year   cases population
#> *       <chr> <int>   <chr>      <chr>
#> 1 Afghanistan 1999     745   19987071
#> 2 Afghanistan 2000    2666   20595360
#> 3      Brazil 1999   37737  172006362
#> 4      Brazil 2000   80488  174504898
#> 5       China 1999  212258 1272915272
#> 6       China 2000  213766 1280428583
```

По умолчанию функция separate() разделяет значения всякий раз, когда встречается символ, не являющийся алфавитно-цифровым (например, символ, не являющийся цифрой или буквой). Например, в предыдущем коде функция separate() разделит значения rate в тех местах, где встречаются

country	year	rate
Afghanistan	1999	**745** / 19987071
Afghanistan	2000	**2666** / 20595360
Brazil	1999	**37737** / 172006362
Brazil	2000	**80488** / 174504898
China	1999	**212258** / 1272915272
China	2000	**213766** / 1280428583

table3

country	year	cases	population
Afghanistan	1999	**745**	19987071
Afghanistan	2000	**2666**	20595360
Brazil	1999	**37737**	172006362
Brazil	2000	**80488**	174504898
China	1999	**212258**	1272915272
China	2000	**213766**	1280428583

Рис. 9.5

символы косой черты. Если вы хотите использовать какой-либо другой символ для разбиения столбца, его можно передать функции `separate()` с помощью аргумента `sep`. Например, можно переписать предыдущий код в следующем виде.

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

(С формальной точки зрения `sep` является регулярным выражением. Более подробно регулярные выражения рассматриваются в главе 11.)

Внимательно просмотрите типы столбцов: вы увидите, что столбцы `case` и `population` имеют символьный тип. Это поведение задано в функции `separate()` по умолчанию: оно оставляет тип столбца таким, каким он есть. Однако в данном случае это нам не очень подходит, поскольку в действительности интересующими нас значениями являются числа. Используя аргумент `convert = TRUE`, мы можем сообщить функции `separate()`, что желательно попытаться преобразовать данные в более подходящий тип.

```
table3 %>%
  separate(
    rate,
    into = c("cases", "population"),
    convert = TRUE
  )
#> # A tibble: 6 × 4
#>       country  year  cases population
#> *       <chr> <int>  <int>      <int>
#> 1 Afghanistan  1999    745   19987071
#> 2 Afghanistan  2000   2666   20595360
```

```
#> 3       Brazil  1999  37737  172006362
#> 4       Brazil  2000  80488  174504898
#> 5        China  1999 212258 1272915272
#> 6        China  2000 213766 1280428583
```

Кроме того, параметру `sep` можно передать целочисленный вектор. Функция `separate()` будет интерпретировать целые числа как номера позиций, по которым следует разбивать значения. Положительные значения начинаются с 1 и отсчитываются от крайней слева позиции в строках; отрицательные значения начинаются с –1 и отсчитываются от крайней справа позиции. При разбиении строк с использованием целых чисел длина `sep` должна быть на единицу меньше количества имен в `into`.

Можно использовать этот подход для отделения последних двух цифр каждого года. При этом данные становятся менее аккуратными, но в некоторых случаях это оказывается полезным, как вы вскоре увидите.

```
table3 %>%
  separate(year, into = c("century", "year"), sep = 2)
#> # A tibble: 6 × 4
#>       country century  year              rate
#> *       <chr>   <chr> <chr>             <chr>
#> 1 Afghanistan      19    99      745/19987071
#> 2 Afghanistan      20    00     2666/20595360
#> 3      Brazil      19    99   37737/172006362
#> 4      Brazil      20    00   80488/174504898
#> 5       China      19    99 212258/1272915272
#> 6       China      20    00 213766/1280428583
```

Объединение столбцов

Функция `unite()` выполняет действие, обратное функции `separate()`: она объединяет несколько столбцов в один столбец. Вы будете использовать ее гораздо реже, чем функцию `separate()`, и все же ее полезно иметь в своем арсенале.

Функцию `unite()` можно использовать для обратного объединения столбцов `century` и `year`, которые мы создали в последнем примере. Эти данные сохранены в виде набора `tidyr::table5`. Функция `unite()` принимает в качестве аргументов фрейм данных, имя новой переменной, которую необходимо создать, и набор объединяемых столбцов, определенных в `dplyr::select()`. Результат представлен на рис. 9.6 и в приведенном ниже коде.

```
table5 %>%
  unite(new, century, year)
#> # A tibble: 6 × 3
#>      country   new              rate
#> *      <chr> <chr>             <chr>
#> 1 Afghanistan 19_99      745/19987071
#> 2 Afghanistan 20_00     2666/20595360
#> 3      Brazil 19_99   37737/172006362
#> 4      Brazil 20_00   80488/174504898
#> 5       China 19_99 212258/1272915272
#> 6       China 20_00 213766/1280428583
```

В данном случае мы также должны использовать аргумент `sep`. По умолчанию объединяемые значения из различных столбцов разделяются символами подчеркивания (`_`). В этом примере нам не нужны разделители, и потому мы задаем в качестве разделителя пустую строку (`""`).

```
table5 %>%
  unite(new, century, year, sep = "")
#> # A tibble: 6 × 3
#>      country   new              rate
#> *      <chr> <chr>             <chr>
#> 1 Afghanistan  1999      745/19987071
#> 2 Afghanistan  2000     2666/20595360
#> 3      Brazil  1999   37737/172006362
#> 4      Brazil  2000   80488/174504898
#> 5       China  1999 212258/1272915272
#> 6       China  2000 213766/1280428583
```

Рис. 9.6

Упражнения

1. Для чего предназначены аргументы `extra` и `fill` функции `separate()`? Поэкспериментируйте с различными опциями, используя два набора данных, приведенных ниже.

```
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"))

tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
```

2. Обе функции, `unite()` и `separate()`, имеют аргумент `remove`. Что он делает? В каких случаях вы устанавливали бы для него значение `FALSE`?

3. Сравните между собой функции `separate()` и `extract()`. Почему существуют три варианта разбиения на столбцы (по позиции, с помощью разделителя, по группам), но только один вариант объединения столбцов?

Отсутствующие значения

В связи с изменением представления набора данных возникает один важный нюанс, касающийся отсутствующих значений. Как это ни парадоксально, отсутствующие значения могут существовать в двух формах:

- *явной*, т.е. отмеченной маркером `NA`;
- *неявной*, т.е. буквальное отсутствие значения среди данных. Проиллюстрируем эту идею на очень простом наборе данных.

```
stocks <- tibble(
  year     = c(2015, 2015, 2015, 2015, 2016, 2016, 2016),
  qtr      = c(   1,    2,    3,    4,    2,    3,    4),
  return   = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

В этом наборе данных есть два отсутствующих значения:

- значение для IV квартала 2015 года явно обозначено как отсутствующее, поскольку в той ячейке, в которой оно должно было находиться, вместо него содержится `NA`;
- значение для I квартала 2016 года отсутствует неявно, поскольку оно просто не появляется в наборе данных.

Одно из объяснений различия между этими двумя случаями сродни коану в дзен-буддизме: явное отсутствие значения — это наличие отсутствия, неявное — отсутствие наличия.

Применяемый к набору данных способ представления может сделать неявное отсутствующее значение явным. Например, мы можем это сделать, поместив годы в столбцы.

```
stocks %>%
  spread(year, return)
#> # A tibble: 4 × 3
#>     qtr `2015` `2016`
#> * <dbl>  <dbl>  <dbl>
#> 1     1   1.88     NA
#> 2     2   0.59   0.92
#> 3     3   0.35   0.17
#> 4     4     NA   2.66
```

Поскольку эти отсутствующие значения могут не играть существенной роли для других представлений данных, можете превратить явное отсутствие в неявное, задав для функции `gather()` аргумент `na.rm = TRUE`.

```
stocks %>%
  spread(year, return) %>%
  gather(year, return, `2015`:`2016`, na.rm = TRUE)
#> # A tibble: 6 × 3
#>     qtr  year return
#> * <dbl> <chr>  <dbl>
#> 1     1  2015   1.88
#> 2     2  2015   0.59
#> 3     3  2015   0.35
#> 4     2  2016   0.92
#> 5     3  2016   0.17
#> 6     4  2016   2.66
```

Другое важное средство, позволяющее сделать отсутствие значений явным в аккуратных данных, — функция `complete()`.

```
stocks %>%
  complete(year, qtr)
#> # A tibble: 8 × 3
#>    year   qtr return
#>   <dbl> <dbl>  <dbl>
#> 1  2015     1   1.88
#> 2  2015     2   0.59
#> 3  2015     3   0.35
#> 4  2015     4     NA
```

```
#> 5   2016     1     NA
#> 6   2016     2   0.92
#> # ... with 2 more rows
```

Функция `complete()` принимает набор столбцов и находит все уникальные комбинации. Далее она убеждается в том, что исходный набор данных содержит все эти значения, вставляя маркеры явного отсутствия значений (`NA`) там, где это необходимо.

Существует еще один важный инструмент для работы с отсутствующими значениями, о котором вам следует знать. Иногда, когда источник данных используется главным образом для ввода данных, отсутствующее значение указывает на то, что предыдущее значение должно быть перенесено в следующую запись.

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,            9,
  "Katherine Burke",  1,            4
)
```

Эти отсутствующие значения можно заполнить с помощью функции `fill()`. Она принимает набор столбцов, отсутствующие значения в которых вы хотите заменить ближайшим предыдущим значением, не являющимся отсутствующим (которое иногда называют последним наблюдением, переносимым вперед).

```
treatment %>%
  fill(person)
#> # A tibble: 4 × 3
#>              person treatment response
#>               <chr>     <dbl>    <dbl>
#> 1 Derrick Whitmore        1         7
#> 2 Derrick Whitmore        2        10
#> 3 Derrick Whitmore        3         9
#> 4  Katherine Burke        1         4
```

Упражнения

1. Сравните аргументы `fill` функций `spread()` и `complete()`.
2. Что делает аргумент `direction` функции `fill()`?

Учебный пример

В завершение рассмотрим пример реалистичных данных, для приведения которых к аккуратному виду вам придется применить на практике все то, о чем вы узнали в этой главе. Набор `tidyr::who` содержит данные о случаях заболевания туберкулезом, разбитые по годам, странам, возрасту, полу и методам диагностики. Данные взяты из отчета Всемирной организации здравоохранения по заболеваемости туберкулезом в различных странах за 2014 год и доступны по следующему адресу: http://www.who.int/tb/country/data/download/en/.

В этом наборе данных содержится огромный объем эпидемиологической информации, но мы будем работать с данными в той форме, в которой они представлены ниже.

```
who
#> # A tibble: 7,240 × 60
#>       country  iso2  iso3  year new_sp_m014 new_sp_m1524
#>         <chr> <chr> <chr> <int>       <int>        <int>
#> 1 Afghanistan    AF   AFG  1980          NA           NA
#> 2 Afghanistan    AF   AFG  1981          NA           NA
#> 3 Afghanistan    AF   AFG  1982          NA           NA
#> 4 Afghanistan    AF   AFG  1983          NA           NA
#> 5 Afghanistan    AF   AFG  1984          NA           NA
#> 6 Afghanistan    AF   AFG  1985          NA           NA
#> # ... with 7,234 more rows, and 54 more variables:
#>   new_sp_m2534 <int>, new_sp_m3544 <int>,
#>   new_sp_m4554 <int>, new_sp_m5564 <int>,
#>   new_sp_m65   <int>,  new_sp_f014 <int>,
#>   new_sp_f1524 <int>, new_sp_f2534 <int>,
#>   new_sp_f3544 <int>, new_sp_f4554 <int>,
#>   new_sp_f5564 <int>,   new_sp_f65 <int>,
#>   new_sn_m014  <int>, new_sn_m1524 <int>,
#>   new_sn_m2534 <int>, new_sn_m3544 <int>,
#>   new_sn_m4554 <int>, new_sn_m5564 <int>,
#>   new_sn_m65   <int>,  new_sn_f014 <int>,
#>   new_sn_f1524 <int>, new_sn_f2534 <int>,
#>   new_sn_f3544 <int>, new_sn_f4554 <int>,
#>   new_sn_f5564 <int>,   new_sn_f65 <int>,
#>   new_ep_m014  <int>, new_ep_m1524 <int>,
#>   new_ep_m2534 <int>, new_ep_m3544 <int>,
#>   new_ep_m4554 <int>, new_ep_m5564 <int>,
#>   new_ep_m65   <int>,  new_ep_f014 <int>,
#>   new_ep_f1524 <int>, new_ep_f2534 <int>,
#>   new_ep_f3544 <int>, new_ep_f4554 <int>,
```

```
#> #   new_ep_f5564 <int>,   new_ep_f65 <int>,
#> #   newrel_m014  <int>, newrel_m1524 <int>,
#> #   newrel_m2534 <int>, newrel_m3544 <int>,
#> #   newrel_m4554 <int>, newrel_m5564 <int>,
#> #   newrel_m65   <int>,  newrel_f014 <int>,
#> #   newrel_f1524 <int>, newrel_f2534 <int>,
#> #   newrel_f3544 <int>, newrel_f4554 <int>,
#> #   newrel_f5564 <int>,   newrel_f65 <int>
```

Это типичный пример реального набора данных. Он содержит избыточные столбцы, странные коды переменных и много отсутствующих значений. Короче говоря, набор данных `who` подготовлен неряшливо, и мы должны привести его к аккуратному виду. Как и **dplyr**, пакет **tidyr** спроектирован таким образом, чтобы каждая функция хорошо справлялась с какой-то одной операцией. Это означает, что в реальных ситуациях вам, как правило, придется объединять несколько глаголов в один канал.

Почти всегда лучше начать с того, чтобы собрать вместе все столбцы, не являющиеся переменными. Посмотрим, что мы имеем.

- Похоже на то, что для определения страны используется избыточное количество переменных: `country`, `iso2` и `iso3`.
- Очевидно, что столбец `year` также является переменной.
- Пока что мы не знаем, что собой представляют остальные столбцы, однако, учитывая структуру имен переменных (например, `new_sp_m014`, `new_ep_m014`, `new_ep_f014`), можно предположить, что они должны быть значениями, а не переменными.

Итак, мы должны собрать вместе все столбцы от `new_sp_m014` до `newrel_f65`. Поскольку нам все еще неизвестно, что именно представляют эти значения, присвоим им обобщенное имя `"key"`. Мы знаем, что ячейки представляют количество случаев, и потому используем переменную `cases`. В текущем представлении имеется множество отсутствующих значений, соответственно, на данном этапе мы используем параметр `na.rm` для того, чтобы сфокусировать внимание на имеющихся значениях.

```
who1 <- who %>%
  gather(
    new_sp_m014:newrel_f65, key = "key",
    value = "cases",
    na.rm = TRUE
)
who1
```

```
#> # A tibble: 76,046 × 6
#>      country  iso2  iso3  year         key cases
#> *      <chr> <chr> <chr> <int>       <chr> <int>
#> 1 Afghanistan    AF   AFG  1997 new_sp_m014     0
#> 2 Afghanistan    AF   AFG  1998 new_sp_m014    30
#> 3 Afghanistan    AF   AFG  1999 new_sp_m014     8
#> 4 Afghanistan    AF   AFG  2000 new_sp_m014    52
#> 5 Afghanistan    AF   AFG  2001 new_sp_m014   129
#> 6 Afghanistan    AF   AFG  2002 new_sp_m014    90
#> # ... with 7.604e+04 more rows
```

Мы можем получить некоторое представление о структуре значений в новом столбце `key`, подсчитав их.

```
who1 %>%
  count(key)
#> # A tibble: 56 × 2
#>            key    n
#>          <chr><int>
#> 1  new_ep_f014 1032
#> 2 new_ep_f1524 1021
#> 3 new_ep_f2534 1021
#> 4 new_ep_f3544 1021
#> 5 new_ep_f4554 1017
#> 6 new_ep_f5564 1017
#> # ... with 50 more rows
```

Немного поразмыслив и поэкспериментировав, вы могли бы и сами разобрать эти данные, но, к счастью, у нас под рукой есть словарь данных, который сообщает нам следующее.

1. Первые три буквы названия каждого столбца говорят о том, информацию о каких случаях заболевания он содержит: новых (`new`) или старых (`ul`). В этом наборе данных каждый столбец содержит данные о новых случаях заболевания.
2. Следующие две буквы описывают тип туберкулеза:
   - `reel` — рецидивный;
   - `up` — внелегочный;
   - `sun` — легочный, не диагностируемый с помощью мазка (отрицательный результат мазка);
   - `spa` — легочный, диагностируемый с помощью мазка (положительный результат мазка).

3. Шестая буква обозначает пол пациента. В наборе данных случаи заболеваний сгруппированы по мужчинам (`m`) и женщинам (`f`).
4. Оставшиеся цифры дают возрастную группу. Данные сгруппированы по семи возрастным группам:
   - 014 = 0–14 лет;
   - 1524 = 15–24 лет;
   - 2534 = 25–34 лет;
   - 3544 = 35–44 лет;
   - 4554 = 45–54 лет;
   - 5564 = 55–64 лет;
   - 65 = 65 и старше.

Нам нужно ввести одну незначительную правку в формат названий столбцов: к сожалению, имена присвоены немного непоследовательно, поскольку вместо `new_rel` мы имеем `newrel` (здесь вы этого не можете заметить, но если данный нюанс не исправить, то на последующих этапах возникнут проблемы). Более подробную информацию о функции `str_replace()` вы получите в главе 11, но основная идея довольно проста: с помощью этой функции мы заменим символы `"newrel"` на `"new_rel"`, что придаст именам согласованную форму.

```
who2 <- who1 %>%
  mutate(key = stringr::str_replace(key, "newrel", "new_rel"))
who2
#> # A tibble: 76,046 × 6
#>       country  iso2  iso3  year         key cases
#>         <chr> <chr> <chr> <int>       <chr> <int>
#> 1 Afghanistan    AF   AFG  1997 new_sp_m014     0
#> 2 Afghanistan    AF   AFG  1998 new_sp_m014    30
#> 3 Afghanistan    AF   AFG  1999 new_sp_m014     8
#> 4 Afghanistan    AF   AFG  2000 new_sp_m014    52
#> 5 Afghanistan    AF   AFG  2001 new_sp_m014   129
#> 6 Afghanistan    AF   AFG  2002 new_sp_m014    90
#> # ... with 7.604e+04 more rows
```

Мы можем разделить значения в каждом коде за два прохода с помощью функции `separate()`. Первый проход разделит коды в тех местах, где встречается символ подчеркивания.

```
who3 <- who2 %>%
  separate(key, c("new", "type", "sexage"), sep = "_")
who3
=> # A tibble: 76,046 × 8
=>       country  iso2  iso3  year   new  type sexage cases
=> *       <chr> <chr> <chr> <int> <chr> <chr>  <chr> <int>
=> 1 Afghanistan    AF   AFG  1997   new    sp   m014     0
=> 2 Afghanistan    AF   AFG  1998   new    sp   m014    30
=> 3 Afghanistan    AF   AFG  1999   new    sp   m014     8
=> 4 Afghanistan    AF   AFG  2000   new    sp   m014    52
=> 5 Afghanistan    AF   AFG  2001   new    sp   m014   129
=> 6 Afghanistan    AF   AFG  2002   new    sp   m014    90
=> # ... with 7.604e+04 more rows
```

После этого можно опустить новый столбец, поскольку он является постоянным в данном наборе. Опуская его, одновременно опустим столбцы `iso2` и `iso3`, так как они избыточны.

```
who3 %>%
  count(new)
=> # A tibble: 1 × 2
=>     new     n
=>   <chr> <int>
=> 1   new 76046
who4 <- who3 %>%
  select(-new, -iso2, -iso3)
```

Далее мы разделим столбец `sexage` на столбцы `sex` и `age`, выполнив разбивку после первого символа.

```
who5 <- who4 %>%
  separate(sexage, c("sex", "age"), sep = 1)
who5
#> # A tibble: 76,046 × 6
#>       country  year  type   sex   age cases
#> *       <chr> <int> <chr> <chr> <chr> <int>
#> 1 Afghanistan  1997    sp     m   014     0
#> 2 Afghanistan  1998    sp     m   014    30
#> 3 Afghanistan  1999    sp     m   014     8
#> 4 Afghanistan  2000    sp     m   014    52
#> 5 Afghanistan  2001    sp     m   014   129
#> 6 Afghanistan  2002    sp     m   014    90
#> # ... with 7.604e+04 more rows
```

Набор данных `who` приведен к аккуратному виду!

Мы продемонстрировали вам работу кода по частям, назначая каждый промежуточный результат новой переменной. Как правило, вы не будете так работать в интерактивном режиме и вместо этого будете использовать канал.

```
who %>%
  gather(code, value, new_sp_m014:newrel_f65, na.rm = TRUE) %>%
  mutate(
    code = stringr::str_replace(code, "newrel", "new_rel")
  ) %>%
  separate(code, c("new", "var", "sexage")) %>%
  select(-new, -iso2, -iso3) %>%
  separate(sexage, c("sex", "age"), sep = 1)
```

Упражнения

1. В этом учебном примере мы установили `na.rm = TRUE`, чтобы упростить проверку того, получены ли корректные значения. Обоснованно ли это? Подумайте над тем, как представлены отсутствующие значения в этом наборе данных. Являются ли они неявными отсутствующими значениями? В чем разница между `NA` и нулем?
2. Что произойдет, если вы пренебрежете этапом `mutate()`? (`mutate(key = stringr::str_replace(key, "newrel", "new_rel"))`).
3. Мы заявляли, что столбцы `iso2` и `iso3` избыточны при наличии столбца `country`. Подтвердите это заявление.
4. Вычислите общее количество случаев заболевания туберкулезом для каждой страны (`country`), года (`year`) и пола (`sex`). Визуализируйте эти данные информативным способом.

Неаккуратные данные

Прежде чем переходить к рассмотрению других тем, следует кратко обсудить неаккуратные данные. Ранее мы использовали в отношении неаккуратных данных уничижительный термин “неряшливые”. Это было чрезмерным упрощением ситуации: существует множество полезных и хорошо устроенных структур данных, которые нельзя отнести к категории аккуратных. Использование таких структур обусловлено двумя основными причинами:

- альтернативные представления могут обеспечить существенный выигрыш в производительности и объеме используемой памяти;

- для специализированных полей могут существовать собственные соглашения относительно хранения данных, которые могут заметно отличаться от соглашений, принятых для аккуратных данных.

Любая из этих причин означает, что вам требуется нечто отличное от tibble-фрейма (или фрейма данных). Если ваши данные естественным образом вписываются в прямоугольную структуру, состоящую из наблюдений и переменных, то, по нашему мнению, аккуратные данные должны быть вашим выбором по умолчанию. Но для использования других структур данных также есть веские причины; аккуратные данные не являются единственно возможным выбором. Если хотите больше узнать об использовании данных, не приведенных к аккуратному виду, прочитайте содержательную статью на эту тему в блоге Джеффа Лика (http://simplystatistics.org/2016/02/17/non-tidy-data/).

ГЛАВА 10

Работа с реляционными данными с помощью пакета dplyr

Введение

Анализ данных лишь в редких случаях включает только одну таблицу. В типичных случаях вы будете работать со многими таблицами данных и должны будете комбинировать их для того, чтобы иметь возможность получить ответы на интересующие вас вопросы. Совокупность нескольких таблиц называют *реляционными данными*, поскольку они не разрознены, а взаимосвязаны.

Реляционные отношения всегда определяются между парой таблиц. Все другие отношения строятся на основе простой идеи: отношения между тремя и более таблицами всегда являются свойством отношений между каждой их парой. Иногда оба элемента пары могут быть одной и той же таблицей! Это требуется, например, если в таблице содержатся данные о людях и среди данных о каждом лице имеются ссылки на его родителей.

Для работы с реляционными таблицами необходимы глаголы, которые работают с парами таблиц. Есть три семейства глаголов, предназначенных для выполнения следующих операций над реляционными данными:

- *видоизменение соединений*, добавляющее в один фрейм данных новые переменные, которые берутся из соответствующих наблюдений другого фрейма данных;
- *фильтрация соединений*, осуществляющая фильтрацию наблюдений одного фрейма данных на основании того, соответствуют ли они наблюдению из другой таблицы;

- *операции над множествами*, которые обрабатывают наблюдения так, как если бы они были элементами множества.

Чаще всего реляционные данные хранятся в системах управления базами данных (СУБД) — термин, охватывающий почти все современные базы данных. Если до этого вы уже пользовались базами данных, то почти наверняка применяли SQL. В таком случае понятия, которые обсуждаются в этой главе, будут вам знакомы, хотя соответствующие выражения в **dplyr** выглядят немного иначе. Вообще говоря, использовать пакет **dplyr** немного легче, чем SQL, поскольку **dplyr** специализируется на анализе данных: он упрощает выполнение обычных операций анализа данных за счет усложнения других операций, которые редко применяются при анализе данных.

Необходимые ресурсы

Мы будем работать с реляционной базой данных из пакета **nycflights13**, используя двухтабличные глаголы из **dplyr**.

```
library(tidyverse)
library(nycflights13)
```

Пакет nycflights13

Для знакомства с реляционными данными мы применяем пакет **nycflights13**. Он содержит четыре tibble-фрейма, связанных с таблицей `flights`, которую мы использовали в главе 3.

- Фрейм `airlines` позволяет найти полное название перевозчика по его сокращенному коду.

```
airlines
#> # A tibble: 16 × 2
#>   carrier                    name
#>     <chr>                   <chr>
#> 1      9E        Endeavor Air Inc.
#> 2      AA   American Airlines Inc.
#> 3      AS     Alaska Airlines Inc.
#> 4      B6          JetBlue Airways
#> 5      DL     Delta Air Lines Inc.
#> 6      EV ExpressJet Airlines Inc.
#> # ... with 10 more rows
```

- Фрейм `airports` предоставляет информацию о каждом аэропорте, идентифицируемом с помощью кода `faa`.

```
airports
#> # A tibble: 1,396 × 7
#>     faa                           name   lat   lon
#>   <chr>                          <chr> <dbl> <dbl>
#> 1   04G              Lansdowne Airport  41.1 -80.6
#> 2   06A  Moton Field Municipal Airport  32.5 -85.7
#> 3   06C            Schaumburg Regional  42.0 -88.1
#> 4   06N                Randall Airport  41.4 -74.4
#> 5   09J          Jekyll Island Airport  31.1 -81.4
#> 6   0A9 Elizabethton Municipal Airport  36.4 -82.2
#> # ... with 1,390 more rows, and 3 more variables:
#> #   alt <int>, tz <dbl>, dst <chr>
```

- Фрейм `planes` предоставляет информацию о каждом самолете, идентифицируемом по его бортовому номеру.

```
planes
#> # A tibble: 3,322 × 9
#>   tailnum  year                    type
#>     <chr> <int>                   <chr>
#> 1  N10156  2004 Fixed wing multi engine
#> 2  N102UW  1998 Fixed wing multi engine
#> 3  N103US  1999 Fixed wing multi engine
#> 4  N104UW  1999 Fixed wing multi engine
#> 5  N10575  2002 Fixed wing multi engine
#> 6  N105UW  1999 Fixed wing multi engine
#> # ... with 3,316 more rows, and 6 more variables:
#> #  manufacturer <chr>, model <chr>, engines <int>,
#> #  seats <int>, speed <int>, engine <chr>
```

- Фрейм `weather` предоставляет почасовые сведения о погодных условиях в каждом из аэропортов Нью-Йорка.

```
weather
#> # A tibble: 26,130 × 15
#>   origin  year month   day  hour  temp  dewp humid
#>    <chr> <dbl> <dbl> <int> <int> <dbl> <dbl> <dbl>
#> 1    EWR  2013     1     1     0  37.0  21.9  54.0
#> 2    EWR  2013     1     1     1  37.0  21.9  54.0
#> 3    EWR  2013     1     1     2  37.9  21.9  52.1
#> 4    EWR  2013     1     1     3  37.9  23.0  54.5
#> 5    EWR  2013     1     1     4  37.9  24.1  57.0
#> 6    EWR  2013     1     1     6  39.0  26.1  59.4
#> # ... with 2.612e+04 more rows, and 7 more variables:
```

```
#> #  wind_dir <dbl>, wind_speed <dbl>, wind_gust <dbl>,
#> #  precip <dbl>, pressure <dbl>, visib <dbl>,
#> #  time_hour <dttm>
```

Одним из способов отображения отношений между различными таблицами является построение диаграмм (рис. 10.1).

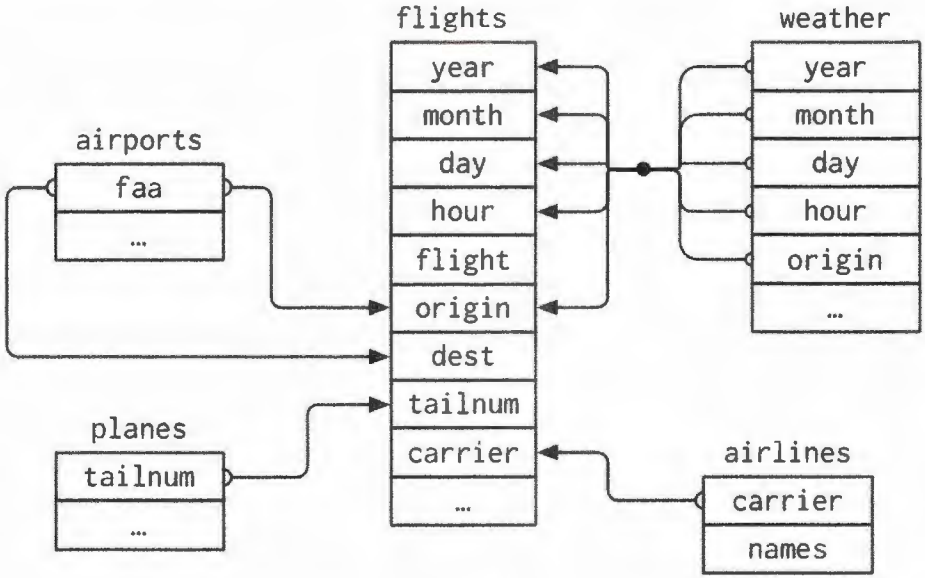


Рис. 10.1

Эта диаграмма может показаться чрезмерно сложной, но она очень проста по сравнению с теми диаграммами, с которыми вам придется столкнуться на практике! Вам будет легче понимать диаграммы наподобие этой, если вы запомните, что каждое отношение всегда касается пары таблиц. Вам не нужно понимать сразу все — достаточно понимать цепочку отношений между интересующими вас таблицами.

Для пакета **nycflights13**:

- `flights` соединяется с `planes` посредством переменной `tailnum`;
- `flights` соединяется с `airlines` посредством переменной `carrier`;
- `flights` соединяется с `airports` посредством двух переменных, `origin` и `dest`;
- `flights` соединяется с `weather` посредством переменной `origin` (местоположение), а также переменных `year`, `month`, `day` и `hour` (время).

Упражнения

1. Предположим, вы хотите построить (приблизительно) маршрут полета каждого самолета из исходного пункта в пункт назначения. Какие переменные вам потребуются? Какие таблицы вам нужно будет совместно использовать?
2. Мы забыли указать связь между таблицами `weather` и `airports`. О каком отношении идет речь и как оно должно выглядеть на диаграмме?
3. В таблице `weather` содержатся данные о погоде, касающиеся только аэропортов Нью-Йорка. Если бы она содержала сведения о погодных условиях во всех аэропортах США, то какое дополнительное отношение с таблицей `flights` нужно было бы определить?
4. Мы знаем, что некоторые дни в году являются "особыми", и некоторые люди предпочитают не летать в эти дни. Как бы вы могли представить эти данные в виде фрейма? Что служило бы первичными ключами в этой таблице? Как бы вы связали ее с другими таблицами?

Ключи

Переменные, используемые для установления связи между каждой парой таблиц, называются *ключами*. Ключ — это переменная (или набор переменных), которая однозначно идентифицирует наблюдение. В простых случаях для идентификации одного наблюдения достаточно одного ключа. Например, каждый самолет можно однозначно идентифицировать по его бортовому номеру (`tailnum`). В других случаях может потребоваться несколько переменных. Например, чтобы идентифицировать наблюдение в таблице погодных условий (`weather`), требуются пять переменных: `year`, `month`, `day`, `hour` и `origin` (год, месяц, день, час и аэропорт вылета).

Ключи бывают двух видов.

- *Первичный ключ* идентифицирует наблюдение в собственной таблице. Например, `planes$tailnum` — первичный ключ, поскольку он однозначно идентифицирует каждый самолет в таблице `planes`.
- *Внешний ключ* однозначно идентифицирует наблюдение в другой таблице. Например, `flights$tailnum` — внешний ключ, поскольку он появляется в таблице `flights`, в которой он устанавливает для каждого авиарейса однозначно определяемый самолет.

Переменная может служить *одновременно* первичным и внешним ключом. Например, переменная `origin` является частью первичного ключа для таблицы `weather` и одновременно внешним ключом для таблицы `airports`.

Как только вы определите первичные ключи в своих таблицах, целесообразно проверить, что они действительно однозначно определяют каждое наблюдение. Один из способов сделать это заключается в подсчете первичных ключей с помощью функции `count()` и выводе значений, для которых `n` больше единицы.

```
planes %>%
  count(tailnum) %>%
  filter(n > 1)
#> # A tibble: 0 × 2
#> # ... with 2 variables: tailnum <chr>, n <int>

weather %>%
  count(year, month, day, hour, origin) %>%
  filter(n > 1)
#> Source: local data frame [0 x 6]
#> Groups: year, month, day, hour [0]
#>
#> # ... with 6 variables: year <dbl>, month <dbl>, day <int>,
#> #    hour <int>, origin <chr>, n <int>
```

Иногда таблица не имеет явного первичного ключа: каждая строка является наблюдением, но ни одна комбинация переменных не обеспечивает ее надежную идентификацию. Например, что является первичным ключом в таблице `flights`? Вы могли бы подумать, что в качестве первичного ключа можно использовать комбинацию даты и номера авиарейса или бортового номера, но ни одна из таких комбинаций не является уникальной.

```
flights %>%
  count(year, month, day, flight) %>%
  filter(n > 1)
#> Source: local data frame [29,768 x 5]
#> Groups: year, month, day [365]
#>
#>    year month   day flight     n
#>   <int> <int> <int> <int> <int>
#> 1  2013     1     1     1     2
#> 2  2013     1     1     3     2
#> 3  2013     1     1     4     2
#> 4  2013     1     1    11     3
```

```
=> 5  2013     1     1     15     2
=> 6  2013     1     1     21     2
=> # ... with 2.976e+04 more rows

flights %>%
  count(year, month, day, tailnum) %>%
  filter(n > 1)
=> Source: local data frame [64,928 x 5]
=> Groups: year, month, day [365]
=>
=> ?  year month   day tailnum     n
=> ? <int> <int> <int>   <chr> <int>
=> 1  2013     1     1  N0EGMQ     2
=> 2  2013     1     1  N11189     2
=> 3  2013     1     1  N11536     2
=> 4  2013     1     1  N11544     3
=> 5  2013     1     1  N11551     2
=> 6  2013     1     1  N12540     2
=> # ... with 6.492e+04 more rows
```

Приступая к работе с этими данными, мы наивно предполагали, что каждый номер авиарейса будет использоваться только один раз на протяжении одного дня: это намного упростило бы вывод сообщений о возможных проблемах с конкретным авиарейсом. К сожалению, это не так! Если в таблице отсутствует первичный ключ, его иногда удобно добавить с помощью функций `mutate()` и `row_number()`. Это упрощает нахождение совпадений с наблюдениями, если вы осуществили какую-то фильтрацию данных и хотите выполнить обратное сопоставление результата с исходными данными. Такой ключ называют *суррогатным*.

Первичный ключ и соответствующий внешний ключ в другой таблице образуют отношение. Как правило, такие отношения относятся к типу "один ко многим". Например, каждому авиарейсу соответствует один самолет, но каждому самолету соответствует множество авиарейсов. В других случаях встречаются отношения "один к одному". Такие отношения можно представлять себе как частный случай отношений "один ко многим". Отношения "многие ко многим" можно моделировать с помощью отношения "многие к одному" плюс отношение "один ко многим". Например, в наших данных отношение "многие ко многим" существует между таблицами `airlines` и `airports`: каждая авиакомпания совершает полеты во многие аэропорты, и каждый аэропорт принимает самолеты многих компаний.

Упражнения

1. Добавьте суррогатный ключ в таблицу `flights`.
2. Идентифицируйте ключи в следующих наборах данных:

   а) `Lahman::Batting`

   б) `babynames::babynames`

   в) `nasaweather::atmos`

   г) `fueleconomy::vehicles`

   д) `ggplot2::diamonds`

   (Возможно, вам придется установить некоторые пакеты и прочитать соответствующую документацию.)
3. Нарисуйте диаграмму, иллюстрирующую отношения между таблицами `Batting`, `Master` и `Salaries` пакета `Lahman`. Нарисуйте другую диаграмму, отображающую отношения между таблицами `Master`, `Managers` и `AwardsManagers`. Как бы вы охарактеризовали отношения между таблицами `Batting`, `Pitching` и `Fielding`?

Мутирующие соединения

Первое из предназначенных для соединения пары таблиц средств, которое мы рассмотрим, — *мутирующие соединения*, которые также называют *изменяющимися*. Мутирующее соединение позволяет объединять переменные из двух таблиц. Сначала оно находит соответствующие наблюдения по их ключам, а затем копирует переменные из одной таблицы в другую.

Как и функция `mutate()`, функции, соединяющие таблицы между собой, добавляют переменные в правый конец таблицы, и если в таблице уже содержится большое количество переменных, то вы не увидите новые переменные в выводе. Для приведенных ниже примеров был специально создан более узкий набор данных, чтобы вам было легче наблюдать за тем, что происходит.

```
flights2 <- flights %>%
  select(year:day, hour, origin, dest, tailnum, carrier)
flights2
#> # A tibble: 336,776 × 8
#>    year month   day  hour origin  dest tailnum carrier
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228      UA
#> 2  2013     1     1     5    LGA   IAH  N24211      UA
```

```
#> 3  2013     1     1     5    JFK   MIA  N619AA      AA
#> 4  2013     1     1     5    JFK   BQN  N804JB      B6
#> 5  2013     1     1     6    LGA   ATL  N668DN      DL
#> 6  2013     1     1     5    EWR   ORD  N39463      UA
#> # ... with 3.368e+05 more rows
```

(Не забывайте о том, что при работе в RStudio эту проблему можно обойти, используя функцию `View()`.)

Предположим, вы хотите добавить полное название авиакомпании в данные `flights2`. Фреймы данных `airlines` и `flights2` можно соединить с помощью функции `left_join()`.

```
flights2 %>%
  select(-origin, -dest) %>%
  left_join(airlines, by = "carrier")
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier
#>   <int> <int> <int> <dbl>   <chr>   <chr>
#> 1  2013     1     1     5  N14228      UA
#> 2  2013     1     1     5  N24211      UA
#> 3  2013     1     1     5  N619AA      AA
#> 4  2013     1     1     5  N804JB      B6
#> 5  2013     1     1     6  N668DN      DL
#> 6  2013     1     1     5  N39463      UA
#> # ... with 3.368e+05 more rows, and 1 more variable:
#>  name <chr>
```

Результатом соединения таблицы `airlines` с таблицей `flights2` является создание дополнительной переменной: `name`. Вот почему мы называем этот тип соединений мутирующим. В данном случае можно было бы получить тот же результат, используя функцию `mutate()` и базовое средство извлечения поднаборов R.

```
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier
#>   <int> <int> <int> <dbl>   <chr>   <chr>
#> 1  2013     1     1     5  N14228      UA
#> 2  2013     1     1     5  N24211      UA
#> 3  2013     1     1     5  N619AA      AA
#> 4  2013     1     1     5  N804JB      B6
#> 5  2013     1     1     6  N668DN      DL
#> 6  2013     1     1     5  N39463      UA
#> # ... with 3.368e+05 more rows, and 1 more variable:
#>  name <chr>
```

Но такой подход трудно обобщить на те случаи, когда необходимо устанавливать соответствие для нескольких переменных, и он требует очень внимательного чтения кода для того, чтобы понять общие намерения того, кто его написал.

В последующих разделах подробно объясняется, как работают мутирующие соединения. Мы начнем с изучения полезного визуального способа представления соединений. Затем будут объяснены принципы работы четырех разновидностей мутирующих соединений: одного внутреннего и трех внешних. В процессе работы с реальными данными ключи не всегда однозначно идентифицируют наблюдения, в связи с чем мы также обсудим, что происходит в случае отсутствия уникальных совпадений. Наконец, вы узнаете о том, как сообщить пакету **dplyr**, какие переменные являются ключами для данного соединения.

Что представляют собой соединения

Чтобы вам было легче понять, как работают соединения, мы будем использовать их визуальное представление (рис. 10.2).

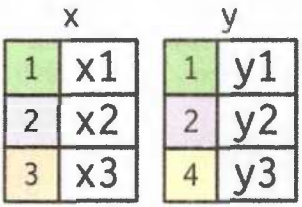


Рис. 10.2

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

Цветные столбцы представляют переменную “key”: они используются для установления соответствия между строками обеих таблиц. Серый столбец представляет столбец “значение”. В этих примерах мы будем отображать одну

переменную-ключ и одну переменную-значение, но сама идея легко обобщается на случай нескольких ключей и нескольких значений.

Соединение — это способ связывания каждой строки x с нулем, одной или несколькими строками y. На рис. 10.3 каждое потенциальное совпадение отображается в виде пересечения пары линий (рис. 10.3).

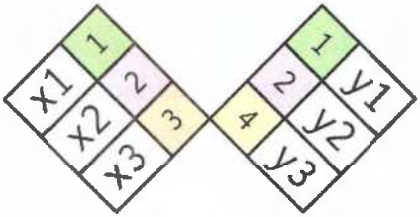


Рис. 10.3

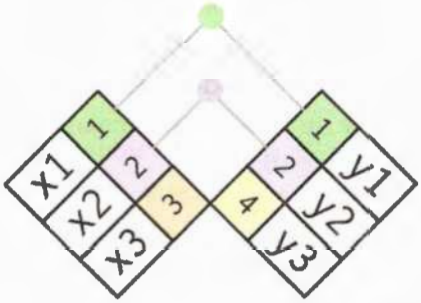
(Если вы внимательно присмотритесь к этой диаграмме, то заметите изменение в порядке следования столбцов ключей и значений в x. Это было сделано для того, чтобы подчеркнуть, что соединение устанавливает соответствия на основе ключа, тогда как значения лишь следуют за ключом.)

В фактическом соединении совпадения будут обозначаться точками. Количество точек = количеству совпадений = количеству строк в результате (рис. 10.4).

Рис. 10.4

Внутреннее соединение

Простейшим типом соединений является *внутреннее соединение*, которое устанавливает соответствие между парами наблюдений, если их ключи совпадают (рис. 10.5).



key	val_x	val_y
1	x1	y1
2	x2	y2

Рис. 10.5

(Строго говоря, этот тип соединения является *внутренним соединением по эквивалентности*, так как для сопоставления ключей используется оператор равенства. Поскольку большинство ключей относится к этому типу, мы, как правило, будем опускать это уточнение.)

Результатом внутреннего соединения является новый фрейм данных, который содержит ключ, значения `x` и значения `y`. Для того чтобы сообщить пакету **dplyr**, какая из переменных является ключом, мы используем параметр `by`.

```
x %>%
  inner_join(y, by = "key")
#> # A tibble: 2 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
```

Самым важным свойством внутреннего соединения является то, что несовпадающие строки не включаются в результат. Это означает, что, как правило, обычные внутренние соединения не подходят для использования в анализе, поскольку можно очень легко потерять наблюдения.

Внешние соединения

Внутреннее соединение сохраняет наблюдения, которые встречаются в обеих таблицах. Внешнее соединение сохраняет наблюдения, встречающиеся хотя бы в одной из таблиц. Различают три типа внешних соединений:

- *левое соединение* сохраняет все наблюдения в `x`;
- *правое соединение* сохраняет все наблюдения в `y`;
- *полное соединение* сохраняет все наблюдения в `x` и `y`.

Эти соединения работают, добавляя дополнительное "виртуальное" наблюдение в каждую таблицу. Данное наблюдение имеет ключ, который совпадает с любым ключом (если никакой другой ключ не совпадает), и значение NA.

Графически это можно представить так (рис. 10.6).

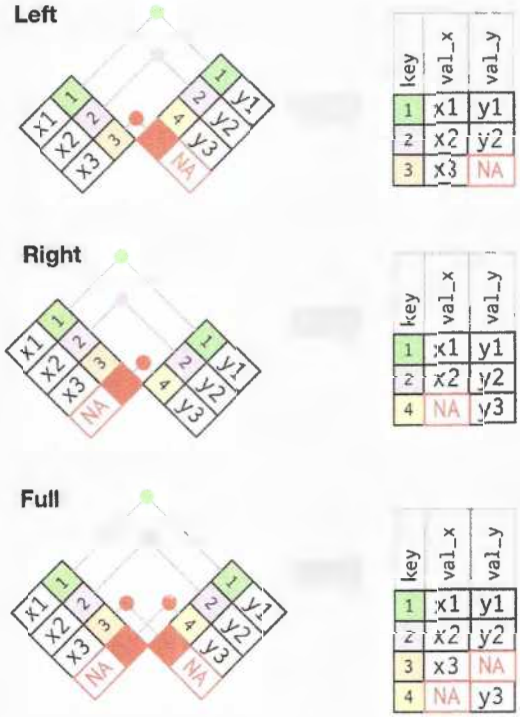


Рис. 10.6

Чаще всего применяется левое соединение: вы используете его всякий раз, когда просматриваете данные из другой таблицы, поскольку оно сохраняет исходные наблюдения даже в случае отсутствия совпадений. Левое соединение должно быть вашим выбором по умолчанию: используйте его всегда, кроме тех случаев, когда имеются веские основания для использования других типов соединений.

Для графического изображения различных типов соединений существует другой способ — диаграммы Венна (рис. 10.7).

Однако этот вариант нельзя назвать удовлетворительным. Он может освежить в вашей памяти, в какой из таблиц то или иное соединение сохраняет наблюдения, но страдает одним серьезным недостатком: диаграммы Венна

ничего не сообщают о том, что происходит в тех случаях, когда ключи не обеспечивают однозначную идентификацию наблюдений.

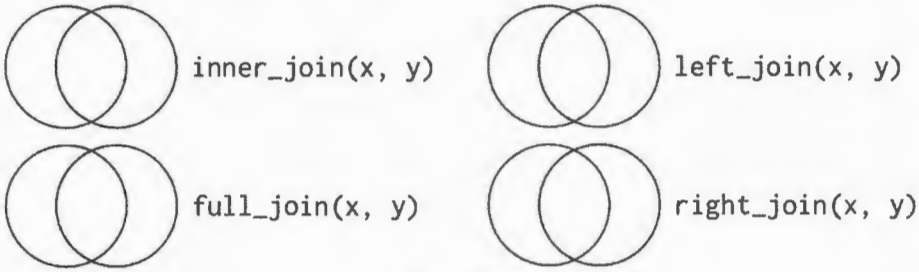


Рис. 10.7

Неуникальные ключи

До сих пор мы во всех случаях предполагали, что ключи уникальны. Однако так бывает не всегда. В этом разделе объясняется, что происходит в тех случаях, когда ключи не являются уникальными. Возможны две ситуации подобного рода.

- Неуникальные ключи имеются в одной таблице. Это может быть полезным, если вы хотите добавить дополнительную информацию в тех случаях, когда имеется типичное отношение "один ко многим" (рис. 10.8).

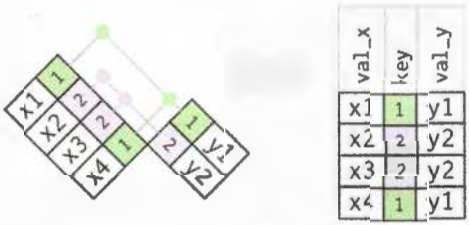


Рис. 10.8

Обратите внимание на некоторое отличие положения ключевого столбца (`key`) в результирующей таблице. Эта деталь отражает тот факт, что данный ключ является первичным ключом в `y`, но внешним в `x`.

```
x <- tribble(
  ~key, ~val_x,
    1, "x1",
    2, "x2",
    2, "x3",
    1, "x4"
```

```
)
y <- tribble(
  ~key, ~val_y,
    1, "y1",
    2, "y2"
)
left_join(x, y, by = "key")
#> # A tibble: 4 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
#> 3     2    x3    y2
#> 4     1    x4    y1
```

- Неуникальные ключи имеются в обеих таблицах. Обычно это является ошибкой, поскольку ни в одной таблице ключи не идентифицируют наблюдения однозначным образом. При соединении неуникальных ключей вы получаете все возможные комбинации — так называемое декартово произведение (рис. 10.9).

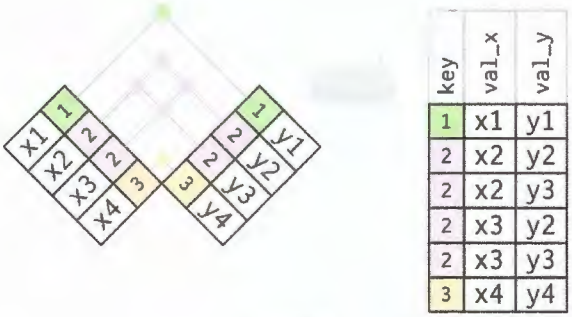


Рис. 10.9

```
x <- tribble(
  ~key, ~val_x,
    1, "x1",
    2, "x2",
    2, "x3",
    3, "x4"
)
y <- tribble(
  ~key, ~val_y,
    1, "y1",
    2, "y2",
    2, "y3",
```

```
    3, "y4"
)
left_join(x, y, by = "key")
#> # A tibble: 6 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1    x1    y1
#> 2     2    x2    y2
#> 3     2    x2    y3
#> 4     2    x3    y2
#> 5     2    x3    y3
#> 6     3    x4    y4
```

Определение ключевых столбцов

До сих пор во всех примерах пары таблиц соединялись посредством одной переменной, которая называлась одинаково в обеих таблицах. Это ограничение задавалось параметром `by = "key"`. Таблицы можно соединять другими способами, используя другие значения параметра `by`.

- Заданному по умолчанию значению `by = NULL` соответствует использование всех переменных, встречающихся в обеих таблицах. Это так называемое *естественное* соединение. Например, таблицы `flights` и `weather` согласуются по своим общим переменным: `year`, `month`, `day`, `hour` и `origin`.

```
flights2 %>%
  left_join(weather)
#> Joining, by = c("year", "month", "day", "hour",
#>   "origin")
#> # A tibble: 336,776 × 18
#>    year month   day  hour origin  dest tailnum
#>   <dbl> <dbl> <int> <dbl>  <chr> <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228
#> 2  2013     1     1     5    LGA   IAH  N24211
#> 3  2013     1     1     5    JFK   MIA  N619AA
#> 4  2013     1     1     5    JFK   BQN  N804JB
#> 5  2013     1     1     6    LGA   ATL  N668DN
#> 6  2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 11 more variables:
#> #  carrier <chr>, temp <dbl>, dewp <dbl>,
#> #  humid <dbl>, wind_dir <dbl>, wind_speed <dbl>,
#> #  wind_gust <dbl>, precip <dbl>, pressure <dbl>,
#> #  visib <dbl>, time_hour <dttm>
```

- Символьный вектор, `by = "x"`. Соединения этого типа похожи на естественное соединение, но используют некоторые из общих переменных. Например, переменная `year` содержится в обеих таблицах, `flights` и `planes`, но имеет в них разный смысл, и потому для соединения таблиц мы используем только переменную `tailnum`.

```
flights2 %>%
  left_join(planes, by = "tailnum")
#> # A tibble: 336,776 × 16
#>   year.x month   day  hour origin  dest tailnum
#>    <int> <int> <int> <dbl>  <chr> <chr>   <chr>
#> 1   2013     1     1     5    EWR   IAH  N14228
#> 2   2013     1     1     5    LGA   IAH  N24211
#> 3   2013     1     1     5    JFK   MIA  N619AA
#> 4   2013     1     1     5    JFK   BQN  N804JB
#> 5   2013     1     1     6    LGA   ATL  N668DN
#> 6   2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 9 more variables:
#> #  carrier <chr>, year.y <int>, type <chr>,
#> #  manufacturer <chr>, model <chr>, engines <int>,
#> #  seats <int>, speed <int>, engine <chr>
```

Обратите внимание на то, что неоднозначность переменных `year` (которые встречаются в обоих входных фреймах данных, но не обязаны быть равными) устранена с помощью суффикса.

- Именованный символьный вектор, `by = c("a" = "b")`. Соединения этого типа сопоставляют переменную `a` в таблице `x` с переменной `b` в таблице `y`. В результате будут использоваться переменные из `x`.

Например, если мы хотим нарисовать карту, необходимо соединить данные таблицы `flights` с данными таблицы `airports`, которая содержит широту и долготу (`lat` и `long`) каждого аэропорта. Каждый авиарейс характеризуется аэропортом вылета и аэропортом назначения, поэтому мы должны конкретизировать, какой из вариантов соединения нам необходим.

```
flights2 %>%
  left_join(airports, c("dest" = "faa"))
#> # A tibble: 336,776 × 14
#>    year month   day  hour origin  dest tailnum
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228
#> 2  2013     1     1     5    LGA   IAH  N24211
#> 3  2013     1     1     5    JFK   MIA  N619AA
```

```
#> 4  2013     1     1     5    JFK   BQN  N804JB
#> 5  2013     1     1     6    LGA   ATL  N668DN
#> 6  2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 7 more variables:
#> #  carrier <chr>, name <chr>, lat <dbl>, lon <dbl>,
#> #  alt <int>, tz <dbl>, dst <chr>

flights2 %>%
  left_join(airports, c("origin" = "faa"))
#> # A tibble: 336,776 × 14
#>    year month   day  hour origin  dest tailnum
#>   <int> <int> <int> <dbl>  <chr> <chr>   <chr>
#> 1  2013     1     1     5    EWR   IAH  N14228
#> 2  2013     1     1     5    LGA   IAH  N24211
#> 3  2013     1     1     5    JFK   MIA  N619AA
#> 4  2013     1     1     5    JFK   BQN  N804JB
#> 5  2013     1     1     6    LGA   ATL  N668DN
#> 6  2013     1     1     5    EWR   ORD  N39463
#> # ... with 3.368e+05 more rows, and 7 more variables:
#> #  carrier <chr>, name <chr>, lat <dbl>, lon <dbl>,
#> #  alt <int>, tz <dbl>, dst <chr>
```

Упражнения

1. Вычислите среднее время задержки по пунктам назначения, а затем выполните соединение для фрейма данных `airports`, чтобы можно было продемонстрировать пространственное распределение времени задержки. Вот один из простых способов нарисовать карту США.

```
airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
    borders("state") +
    geom_point() +
    coord_quickmap()
```

   (Не беспокойтесь, если вам непонятно, что делает функция `semi_join()`, — вскоре вы об этом узнаете.)

   Возможно, вы захотите использовать размер или цвет точек для отображения среднего времени задержки для каждого аэропорта.

2. Добавьте координаты местоположения аэропорта вылета и аэропорта назначения (т.е. переменные `lat` и `lon`) в таблицу `flights`.

3. Существует ли связь между сроком эксплуатации самолета и его задержками?
4. При каких погодных условиях возрастает вероятность задержек?
5. Что случилось 13 июня 2013 года? Отобразите пространственную картину задержек, а затем используйте Google для установления их связи с погодой.

Другие реализации

Функция `base::merge()` способна выполнять все четыре типа мутирующих соединений.

dplyr	merge
`inner_join(x, y)`	`merge(x, y)`
`left_join(x, y)`	`merge(x, y, all.x = TRUE)`
`right_join(x, y)`	`merge(x, y, all.y = TRUE)`
`full_join(x, y)`	`merge(x, y, all.x = TRUE, all.y = TRUE)`

Преимуществом специфических глаголов **dplyr** является то, что они значительно лучше передают назначение вашего кода: различия между типами соединения играют очень важную роль, но они скрываются за аргументами функции `merge()`. Соединения **dplyr** работают значительно быстрее и не могут привести к путанице с порядком строк.

Многие из соглашений **dplyr** навеяны языком SQL, поэтому трансляция команд в SQL-запросы не составляет труда.

dplyr	SQL
`inner_join(x, y, by = "z")`	`SELECT * FROM x INNER JOIN y USING (z)`
`left_join(x, y, by = "z")`	`SELECT * FROM x LEFT OUTER JOIN y USING (z)`
`right_join(x, y, by = "z")`	`SELECT * FROM x RIGHT OUTER JOIN y USING (z)`
`full_join(x, y, by = "z")`	`SELECT * FROM x FULL OUTER JOIN y USING (z)`

Заметьте, что ключевые слова `INNER` и `OUTER` не являются обязательными, и их часто опускают. Для соединения разных переменных двух таблиц, например `inner_join(x,y, by = c("a' = "b"))`, в SQL используется несколько иной синтаксис:

```
SELECT * FROM x INNER JOIN y ON x.a = y.b
```

Синтаксис SQL обеспечивает поддержку более широкого ряда типов соединений, чем в пакете **dplyr**, поскольку допускает использование ограничений, отличных от ограничения равенством.

Фильтрующие соединения

Фильтрующие соединения (полусоединения) сопоставляют наблюдения точно так же, как и мутирующие, но влияют на наблюдения, а не на переменные. Такие соединения бывают двух типов:

- `semi_join(x, y)` — сохраняет все наблюдения в `x`, для которых имеется совпадение в `y`;
- `anti_join(x, y)` — опускает все наблюдения в `x`, для которых имеется совпадение в `y`.

Полусоединения могут быть полезны при обратном сопоставлении отфильтрованных итоговых таблиц с исходными строками. Предположим, например, что вы нашли первую десятку наиболее популярных пунктов назначения.

```
top_dest <- flights %>%
  count(dest, sort = TRUE) %>%
  head(10)
top_dest
#> # A tibble: 10 × 2
#>    dest     n
#>   <chr> <int>
#> 1   ORD 17283
#> 2   ATL 17215
#> 3   LAX 16174
#> 4   BOS 15508
#> 5   MCO 14082
#> 6   CLT 14064
#> # ... with 4 more rows
```

А теперь вы хотите найти все авиарейсы, совершенные в один из этих аэропортов назначения. Вы могли бы самостоятельно спроектировать подходящий фильтр.

```
flights %>%
  filter(dest %in% top_dest$dest)
#> # A tibble: 141,145 × 19
#>    year month   day dep_time sched_dep_time dep_delay
```

```
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      542            540         2
#> 2  2013     1     1      554            600        -6
#> 3  2013     1     1      554            558        -4
#> 4  2013     1     1      555            600        -5
#> 5  2013     1     1      557            600        -3
#> 6  2013     1     1      558            600        -2
#> # ... with 1.411e+05 more rows, and 12 more variables:
#> #  arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #  carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #  dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #  minute <dbl>, time_hour <dttm>
```

Однако такой подход трудно распространить на случай нескольких переменных. Предположим, например, что вы нашли 10 дней с наибольшим временем задержки. Как бы вы спроектировали фильтр, использующий переменные `year`, `month` и `day` для обратного сопоставления с таблицей `flights`?

Вместо этого можно использовать полусоединение, которое соединяет две таблицы подобно мутирующему соединению, но вместо добавления новых столбцов оставляет `x` только в тех строках, для которых находится соответствие в `y`.

```
flights %>%
  semi_join(top_dest)
#> Joining, by = "dest"
#> # A tibble: 141,145 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      554            558        -4
#> 2  2013     1     1      558            600        -2
#> 3  2013     1     1      608            600         8
#> 4  2013     1     1      629            630        -1
#> 5  2013     1     1      656            700        -4
#> 6  2013     1     1      709            700         9
#> # ... with 1.411e+05 more rows, and 13 more variables:
#> #  arr_time <int>, sched_arr_time <int>, arr_delay <dbl>,
#> #  carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#> #  dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#> #  minute <dbl>, time_hour <dttm>
```

В графическом представлении полусоединение выглядит так (рис. 10.10).

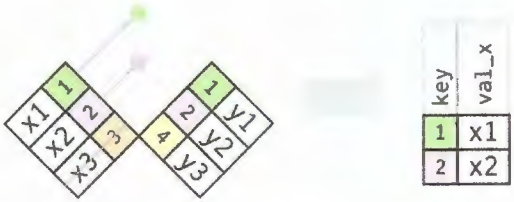


Рис. 10.10

Важен лишь сам факт наличия соответствия; не имеет значения, для какого элемента находится соответствие. Это означает, что фильтрующее соединение никогда не дублирует строки, как это делают мутирующие соединения (рис. 10.11).

Рис. 10.11

Прямой противоположностью полусоединения является *антисоединение*. Оно сохраняет строки, для которых отсутствует соответствие (рис. 10.12).

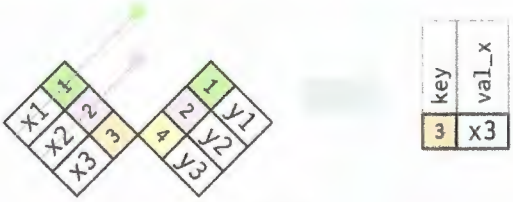


Рис. 10.12

Антисоединения полезны в качестве средства диагностики несоответствий соединения. Например, при соединении таблиц `flights` и `planes` вас могло бы интересовать, много ли найдется авиарейсов, для которых не находится соответствия по самолетам.

```
flights %>%
  anti_join(planes, by = "tailnum") %>%
  count(tailnum, sort = TRUE)
```

```
=  = A tibble: 722 × 2
=    tailnum     n
=      <chr> <int>
=  1    <NA>  2512
=> 2  N725MQ   575
=  3  N722MQ   513
=> 4  N723MQ   507
=> 5  N713MQ   483
=> 6  N735MQ   396
=> # ... with 716 more rows
```

Упражнения

1. Что означает отсутствие бортового номера (`tailnum`) для авиарейса (`flight`)? Что общего имеют бортовые номера, для которых отсутствует соответствующая запись в таблице `planes`? (Подсказка: одна переменная объясняет 90% всех проблем.)
2. Отфильтруйте таблицу `flights`, чтобы отобразить только авиарейсы, которые выполняются самолетами, совершившими не менее 100 полетов.
3. Объедините `fueleconomy::vehicles` и `fueleconomy::common`, чтобы найти лишь записи, соответствующие самым обычным моделям.
4. Найдите 48 часов (на протяжении всего года), характеризующиеся наихудшим показателем задержки рейса. Найдите перекрестные данные в таблице `weather`. Замечаете ли вы какую-либо закономерность?
5. Что делает вызов функции `anti_join(flights, airports, by = c("dest" = "faa"))` tell you? Что делает вызов функции `anti_join(airports, flights, by = c("faa" = "dest"))`?
6. Можно было бы ожидать, что между самолетами и компаниями существуют неявные отношения, поскольку полет каждого самолета совершается одной авиакомпанией. Подтвердите или опровергните эту гипотезу, используя инструменты, которые вы изучили в предыдущем разделе.

Проблемы соединений

Данные, с которыми мы работаем в этой главе, были очищены таким образом, чтобы у вас возникало как можно меньше проблем. Маловероятно, чтобы ваши собственные данные были такими же, и поэтому, чтобы ваши

соединения работали гладко, вы должны предпринимать в отношении данных определенные меры.

1. Начните с идентификации переменных, образующих первичный ключ в каждой таблице. Обычно вы должны это делать на основании понимания сути данных, а не эмпирическим путем, подбирая комбинацию переменных, которая могла бы служить уникальным идентификатором. Если вы всего лишь просматриваете переменные, не задумываясь над тем, что они означают, то вам, возможно, (не) повезет, и вы найдете комбинацию, уникальную для текущих данных, но в общем случае это отношение может не быть истинным.

   Например, высота над уровнем моря и долгота однозначно идентифицируют каждый аэропорт, но такую комбинацию вряд ли можно считать удачным идентификатором!

```
airports %>% count(alt, lon) %>% filter(n > 1)
#> Source: local data frame [0 x 3]
#> Groups: alt [0]
#>
#> # ... with 3 variables: alt <int>, lon <dbl>, n <int>
```

2. Убедитесь в том, что ни одна из переменных в первичном ключе не содержит отсутствующих значений. Отсутствующее значение не может идентифицировать наблюдение!

3. Убедитесь в том, что внешним ключам соответствуют первичные ключи в другой таблице. Лучше всего выполнять такую проверку с помощью функции `anti_join()`. Обычно причиной несоответствия ключей являются ошибки при вводе данных. Устранение подобных ошибок часто стоит больших трудов.

   Если у вас имеются отсутствующие ключи, то вы должны быть особенно внимательны при выборе между внутренними и внешними соединениями, тщательно продумывая, стоит или не стоит опускать строки, для которых не находится соответствия.

Вы должны осознавать, что простой проверки количества строк до и после соединения таблиц еще не достаточно для того, чтобы утверждать, что соединение прошло гладко. Если у вас имеется внутреннее соединение с неуникальными ключами в обеих таблицах, то вам может не повезти в том смысле, что количество опущенных строк может в точности совпасть с количеством продублированных строк!

Операции над множествами

Последний тип двухтабличных глаголов предназначен для выполнения операций над множествами. Вообще говоря, они используются реже других, но иногда они могут быть удобными, если вы хотите разбить один сложный фильтр на более простые части. Все эти операции работают с целыми строками, сравнивая значения каждой переменной. Приведенные ниже функции ожидают, что входные таблицы `x` и `y` имеют одни и те же переменные, и обрабатывают наблюдения как множества.

`intersect(x, y)`

Возвращает только наблюдения, содержащиеся одновременно в `x` и `y`.

`union(x, y)`

Возвращает уникальные наблюдения, содержащиеся в `x` и `y`.

`setdiff(x, y)`

Возвращает наблюдения, содержащиеся в `x`, но не в `y`.

Пусть имеются следующие простые данные.

```
df1 <- tribble(
  ~x, ~y,
   1,  1,
   2,  1
)
df2 <- tribble(
  ~x, ~y,
   1,  1,
   1,  2
)
```

Тогда возможны следующие четыре операции.

```
intersect(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     1

# Заметьте, что мы получаем 3 строки, а не 4
union(df1, df2)
#> # A tibble: 3 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
#> 2     2     1
```

```
#> 3     1     1

setdiff(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     2     1

setdiff(df2, df1)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
```

ГЛАВА 11

Работа со строками с помощью пакета stringr

Введение

В этой главе вы познакомитесь с операциями, позволяющими осуществлять всевозможное манипулирование строками в языке R. Вы узнаете о том, что собой представляют строки и как их можно создавать вручную, но основное внимание будет уделено регулярным выражениям. Как правило, содержащаяся в строках информация не структурирована или частично структурирована, и регулярные выражения предлагают компактный язык для описания строковых шаблонов. Впервые увидев регулярное выражение, вы можете подумать, что это работа вашего кота, прогулявшегося по клавиатуре, но по мере работы с ними их смысл прояснится для вас.

Необходимые ресурсы

В этой главе мы сфокусируем свое внимание на пакете **stringr**, предназначенном для манипулирования строками. Пакет **stringr** не входит в состав ядра библиотеки *tidyverse*, но поскольку работать с текстовыми данными приходится не всегда, мы должны явно загружать его.

```
library(tidyverse)
library(stringr)
```

Основы работы со строками

Строки можно создавать, используя апострофы или кавычки. В языке R, в отличие от других языков, это никак не влияет на поведение строк.

екомендую всегда использовать кавычки ("), если только вы не хотите создавать строку, которая содержит кавычки.

```
string1 <- "Это строка"
string2 <- 'Чтобы включить "кавычки" в строку, используйте апострофы'
```

Если вы забудете ввести закрывающие кавычки, то увидите символ +, означающий ожидание продолжения ввода.

```
> "Это строка без закрывающей кавычки
+
+
+ ПОМОГИТЕ! Я ЗАСТРЯЛ
```

Очутившись в подобной ситуации, нажмите клавишу <Esc> и попытайтесь повторить ввод!

Чтобы включить в строку литеральный апостроф или кавычки, их можно "экранировать" символом обратной косой черты (\).

```
double_quote <- "\"" # или '"'
single_quote <- '\'' # или "'"
```

Это означает, что если вы хотите включить в строку литеральный символ обратной косой черты, то должны продублировать его: "\\".

Имейте в виду, что печатное представление строки — это не то же самое, что сама строка, поскольку печатное представление отображает результаты экранирования символов. Чтобы увидеть "сырое" содержимое строки, используйте функцию writeLines().

```
x <- c("\"", "\\")
x
#> [1] "\"" "\\"
writeLines(x)
#> "
#> \
```

Существует множество других специальных символов. Наиболее распространенными из них являются символы "\n" (перевод строки) и "\t" (символ табуляции); вы сможете увидеть их полный список, вызвав справку командой ?'"' или ?"'". Кроме того, иногда вам встретятся строки наподобие "\u00b5", используемые в качестве способа записи символов, не включенных в латинский алфавит.

```
x <- "\u00b5"
x
#> [1] "µ"
```

Для сохранения сразу нескольких строк часто используют символьные векторы, которые создаются с помощью функции `c()`.

```
c("one", "two", "three")
#> [1] "one" "two" "three"
```

Длина строки

В базовый комплект R входит много функций для работы со строками, но мы будем стараться избегать их использования ввиду недостаточной продуманности их названий, что затрудняет их запоминание. Вместо этого мы будем использовать функции из пакета **stringr**. Их имена интуитивно понятны, и все они начинаются с префикса `str_`. Например, функция `str_length()` возвращает количество символов в строке.

```
str_length(c("a", "R for data science", NA))
#> [1]  1 18 NA
```

Общий префикс `str_` оказывается особенно полезным, если вы используете RStudio, поскольку ввод символов `str_` запускает средство автозавершения ввода, позволяющее увидеть все функции **stringr** (рис. 11.1).

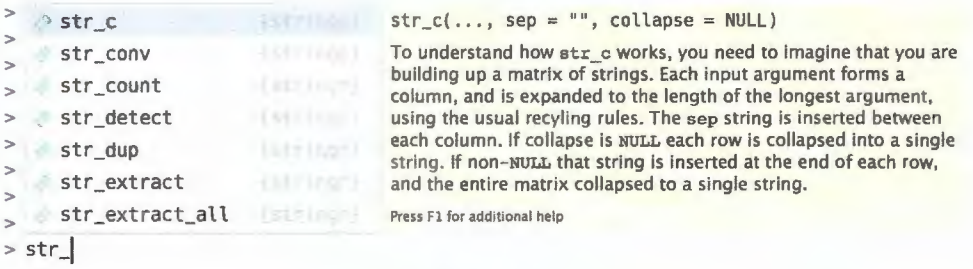


Рис. 11.1

Объединение строк

Чтобы объединить две или большее число строк, используйте функцию `str_c()`.

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
```

Аргумент `sep` позволяет задать символ, который будет отделять строки одну от другой после их объединения.

```
str_c("x", "y", sep = ", ")
#> [1] "x, y"
```

Как и в случае большинства других функций R, отсутствующие значения считаются "инфекцией". Если вы хотите, чтобы они выводились как "NA", используйте функцию str_replace_na().

```
x <- c("abc", NA)
str_c("|-", x, "-|")
#> [1] "|-abc-|" NA
str_c("|-", str_replace_na(x), "-|")
#> [1] "|-abc-|" "|-NA-|"
```

Как следует из предыдущего кода, функция str_c() векторизована и автоматически "зацикливает" короткие векторы по длине самого длинного вектора.

```
str_c("prefix-", c("a", "b", "c"), "-suffix")
#> [1] "prefix-a-suffix" "prefix-b-suffix" "prefix-c-suffix"
```

Объекты длиной 0 опускаются. Этой возможностью особенно удобно пользоваться в сочетании с оператором if.

```
name <- "Hadley"
time_of_day <- "morning"
birthday <- FALSE

str_c(
  "Good ", time_of_day, " ", name,
  if (birthday) " and HAPPY BIRTHDAY",
  "."
)
#> [1] "Good morning Hadley."
```

Для свертывания строкового вектора в одну строку используйте функцию collapse().

```
str_c(c("x", "y", "z"), collapse = ", ")
#> [1] "x, y, z"
```

Извлечение подстрок

Для извлечения подстрок используйте функцию str_sub(). Наряду с самой строкой функция str_sub() принимает аргументы start и end, предоставляющие начальную и конечную позиции подстроки (границы включаются в подстроку).

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"

# отрицательным числам соответствует отсчет с конца строки
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

Обратите внимание на то, что функция успешно выполняется даже для слишком коротких строк; в этом случае она возвращает подстроку максимально возможной длины.

```
str_sub("a", 1, 5)
#> [1] "a"
```

Функцию `str_sub()` можно использовать в форме присваивания для изменения строк.

```
str_sub(x, 1, 1) <- str_to_lower(str_sub(x, 1, 1))
x
#> [1] "apple"  "banana" "pear"
```

Локали

Ранее мы уже использовали функцию `str_to_lower()` для преобразования текста в нижний регистр. В R также предусмотрены функции `str_to_upper()` и `str_to_title()`; первая из них преобразует текст в верхний регистр, а вторая делает первую букву каждого слова прописной. Однако изменение регистра букв — не такая простая операция, как может показаться, поскольку в различных языках для этого существуют разные правила. Выбор нужного набора правил осуществляется с помощью локалей.

```
# В турецком языке имеются две буквы i (с точкой и без точки),
# и к ним применяются разные правила преобразования
# строчных букв в прописные
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

Данная локаль определена в соответствии со стандартом языковых кодов ISO 639, которые представляют собой двух- или трехбуквенные сокращения. Если код нужного языка вам неизвестен, можете найти его в исчерпывающем списке, приведенном в Википедии:

https://en.wikipedia.org/wiki/List_of_ISO_639-1_codes

Если локаль не задана, функция использует текущую локаль, предоставляемую вашей операционной системой.

Другой важной операцией, на которую влияет локаль, является сортировка. Базовые функции R `order()` и `sort()` сортируют строки, используя текущую локаль. Если вы хотите обеспечить устойчивое поведение кода, не зависящее от компьютера, используйте функции `str_sort()` и `str_order()`,принимающие дополнительный аргумент `locale`.

```
x <- c("apple", "eggplant", "banana")

str_sort(x, locale = "en")  # English
#> [1] "apple"    "banana"   "eggplant"

str_sort(x, locale = "haw") # Hawaiian
#> [1] "apple"    "eggplant" "banana"
```

Упражнения

1. В коде, в котором не используются средства пакета **stringr**, вам будут часто встречаться функции `paste()` и `paste0()`. В чем различие между ними? Каким функциям **stringr** они эквивалентны? Каковы различия в обработке этими функциями значений `NA`?
2. Опишите своими словами различия между аргументами `sep` и `collapse` функции `str_c()`.
3. Используйте функции `str_length()` и `str_sub()` для извлечения среднего символа строки. Как вы поступите, если строка содержит четное количество символов?
4. Что делает функция `str_wrap()`? В каких случаях она может вам потребоваться?
5. Что делает функция `str_trim()`? Какая функция является прямой противоположностью функции `str_trim()`?
6. Напишите функцию, которая превращает, например, вектор `c("a", "b", "c")` в строку `a, b, and c`. Тщательно обдумайте, что она должна делать в случае передачи ей вектора длиной 0, 1 или 2.

Поиск соответствия шаблонам с помощью регулярных выражений

Регулярные выражения — это чрезвычайно лаконичный язык, позволяющий описывать строковые шаблоны. С непривычки их вид может довести вас до головокружения, но поняв, как они работают, вы обнаружите, что они очень полезны.

Для первоначального знакомства с регулярными выражениями мы используем функции `str_view()` и `str_view_all()`, которые принимают в качестве аргументов символьный вектор и регулярное выражение и демонстрируют, чему они соответствуют. Начнем с очень простых регулярных выражений, а затем будем постепенно их усложнять. После того как вы освоите основные способы поиска соответствий заданному шаблону, мы перейдем к применению этих идей с помощью различных функций.

Базовые соответствия

Простейшие шаблоны в точности соответствуют строкам (рис. 11.2).

```
x <- c("apple", "banana", "pear")
str_view(x, "an")
```

apple

banana

pear

Рис. 11.2

Следующий по мере возрастания сложности шаблона элемент — точка (`.`), которой соответствует любой символ (кроме символа перевода строки) (рис. 11.3).

```
str_view(x, ".a.")
```

apple

banana

pear

Рис. 11.3

Но если точке соответствует любой символ, то какому шаблону будет соответствовать сам символ точки (.)? В таком случае необходимо экранировать символ точки, который информирует подсистему регулярных выражений о том, что в данном случае он используется не в качестве специального символа, а в качестве литерала, требующего точного совпадения. В регулярных выражениях, как и в строках, для экранирования (отмены) специального поведения используется символ обратной косой черты (\). Следовательно, в качестве шаблона для нахождения символа . вы должны использовать регулярное выражение \.. К сожалению, это создает некоторые неудобства. Для представления регулярных выражений мы используем строки, в которых для экранирования символов также используется символ \, поэтому для создания регулярного выражения \. мы должны использовать строку "\\." (рис. 11.4).

```
# Для создания регулярного выражения следует
# использовать символы \\
dot <- "\\."

# Но само выражение содержит только один такой символ
writeLines(dot)
#> \.

# А в этом выражении задается соответствие
# литеральному символу "."
str_view(c("abc", "a.c", "bef"), "a\\.c")
```

abc

a.c

bef

Рис. 11.4

Если символ \ используется в регулярных выражениях в качестве экранирующего символа, то какому регулярному выражению будет соответствовать литерал \? Конечно же, создавая для него регулярное выражение, его следует экранировать: \\. Но при создании искомого регулярного выражения мы должны использовать строку, которая также нуждается в экранировании символа \. Это означает, что в регулярном выражении, которому будет соответствовать единственный литеральный символ \, нам придется записать четыре таких символа: "\\\\" (рис. 11.5).

```
x <- "a\\b"
writeLines(x)
#> a\b

str_view(x, "\\\\")
```

a\b

Рис. 11.5

В этой книге мы будем записывать регулярные выражения как `\.`, а строки, представляющие эти выражения, — как `"\\."`.

Упражнения

1. Объясните, почему ни одна из этих строк не соответствует регулярному выражению `\`: `"\"`, `"\\"`, `"\\\"`.
2. Какому регулярному выражению соответствует последовательность символов `"'\`?
3. Какие шаблоны будут соответствовать регулярному выражению `\..\..\..`? Каково строковое представление этого регулярного выражения?

Якоря

По умолчанию регулярные выражения будут искать совпадение с любой частью строки. Во многих случаях в регулярных выражениях удобно использовать позиционные привязки, так называемые *якоря*. В частности, для поиска совпадений с текстом, расположенным в начале или в конце строки, используются следующие символы (рис. 11.6 и 11.7):

- `^` (крышка, циркумфлекс) — совпадает с началом строки;
- `$` (доллар) — совпадает с концом строки.

```
x <- c("apple", "banana", "pear")
str_view(x, "^a")
```

apple

banana

pear

Рис. 11.6

```
str_view(x, "a$")
```

apple

banana

pear

Рис. 11.7

Чтобы запомнить, за что отвечает каждый из этих символов, воспользуйтесь следующим мнемоническим правилом: высокое положение в иерархии (^) всегда приводит к деньгам ($).

Чтобы принудить регулярное выражение к совпадениям только с целой строкой, используйте оба якоря, ^ и $ (рис. 11.8 и 11.9).

```
x <- c("apple pie", "apple", "apple cake")
str_view(x, "apple")
```

apple pie

apple

apple cake

Рис. 11.8

```
str_view(x, "^apple$")
```

apple pie

apple

apple cake

Рис. 11.9

Для нахождения совпадений на границах слов можно использовать якорь \b. Мы редко пользуемся им в R, но иногда используем в процессе работы в RStudio, когда хотим найти функцию, имя которой входит в имена других функций. Например, мы будем использовать регулярное выражение \bsum\b, чтобы избежать совпадения с именами функций summarize, summary, rowsum и т.п.

Упражнения

1. Какое регулярное выражение вы использовали бы для поиска литеральной строки `"$^$"`?
2. Обратившись к списку часто встречающихся слов `stringr::words`, создайте регулярное выражение, позволяющее находить слова, которые:
   а) начинаются с буквы "y";
   б) оканчиваются на букву "x";
   в) состоят ровно из трех букв (не пытайтесь схитрить, используя функцию `str_length()`);
   г) состоят из семи и более букв.

   Поскольку этот список очень длинный, вам, возможно, будет удобно использовать аргумент `match` функции `str_view()`, чтобы отображать только слова, совпадающие или не совпадающие с регулярным выражением.

Символьные классы и чередования

Существует ряд специальных шаблонов, которым соответствует не только какой-нибудь один символ. С одним из них вы уже знакомы — это символ "точка" (`.`), который совпадает с любым символом, кроме символа перевода строки. Существуют также четыре других полезных символа:

- `\d` — совпадает с любой цифрой;
- `\s` — совпадает с любым пробельным символом (пробел, символ табуляции, перевод строки);
- `[abc]` — совпадает с `a`, `b` или `c`;
- `[^abc]` — совпадает с любым символом, кроме `a`, `b` или `c`.

Вспомните, что при создании регулярного выражения, содержащего подвыражение `\d` или `\s`, требуется экранировать символ `\` в строке, которая должна выглядеть так: `"\\d"` или `"\\s"`.

Чередование (alternation) обеспечивает возможность выбора альтернативных вариантов шаблона. Например, чередованию `abc|d..f` будет соответствовать строка `"abc"` или `"deaf"`. Учтите, что специальный символ `|` имеет низкий приоритет, так что выражение `abc|xyz` совпадет с `abc` или `xyz`, но не с `abcyz` или `abxyz`. Как и в случае математических выражений, если у

вас есть сомнения относительно приоритета элементов, используйте круглые скобки для того, чтобы прояснить свои намерения (рис. 11.10).

```
str_view(c("grey", "gray"), "gr(e|a)y")
```

grey

gray

Рис. 11.10

Упражнения

1. Создайте регулярное выражение для нахождения всех слов, которые:
   а) начинаются с гласной;
   б) содержат только согласные (подсказка: подумайте над сопоставлением с "не гласными");
   в) оканчиваются на `ed`, но не на `eed`;
   г) оканчиваются на `ing` или `ize`;
2. Проверьте опытным путем соблюдение правила "*i* перед *e*, кроме случаев, когда *i* после *c*";
3. Всегда ли за "q" следует "u"?
4. Напишите регулярное выражение, которое совпадает со словом, если оно, вероятно, написано на британском английском, а не на американском английском.
5. Создайте регулярное выражение, которое будет совпадать с телефонными номерами, записанными в принятом в вашей стране формате.

Повторения

Следующий уровень сложности регулярных выражений включает использование специальных символов (квантификаторов), позволяющих управлять тем, сколько раз может встречаться предшествующий символ или выражение, чтобы совпадение считалось достигнутым (рис. 11.11, 11.12 и 11.13):

- `?` — 0 или 1 раз;
- `+` — 1 или более раз;
- `*` — 0 или более раз.

```
x <- "1888 is the longest year in Roman numerals: MDCCCLXXXVIII"
str_view(x, "CC?")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.11

```
str_view(0, "CC+")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.12

```
str_view(x, 'C[LX]+')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.13

Обратите внимание на то, что эти операторы имеют высокий приоритет, поэтому вы можете записать `colou?r` для поиска совпадений с написанием слова в соответствии с правилами американского или британского английского языка. Отсюда следует, что в большинстве случаев вам понадобится использовать круглые скобки, как в выражении `bana(na)+`.

Кроме того, можно точно определить количество совпадений (рис. 11.14, 11.15 и 11.16):

- `{n}` — ровно *n* раз;
- `{n,}` — *n* или более раз;
- `{,m}` — самое большее *m* раз;
- `{n,m}` — от *n* до *m* раз.

```
str_view(x, "C{2}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.14

```
str_view(x, "C{2,}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.15

```
str_view(x, "C{2,3}")
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.16

По умолчанию выполняется "жадный" поиск, т.е. эти выражения стремятся совпасть с как можно более длинной строкой. Вы можете сделать их "ленивыми", чтобы они стремились совпасть с как можно более короткой строкой, поставив после выражения символ `?`. Эти возможности регулярных выражений предназначены для применения опытными пользователями, но вам будет полезно знать о том, что они существуют (рис. 11.17 и 11.18).

```
str_view(x, 'C{2,3}?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.17

```
str_view(x, 'C[LX]+?')
```

```
1888 is the longest year in Roman numerals: MDCCCLXXXVIII
```

Рис. 11.18

Упражнения

1. Опишите эквиваленты `?`, `+` и `*` в форме `{m,n}`.
2. Опишите словами, какие строки соответствуют следующим выражениям (будьте внимательны и постарайтесь определить, используем ли мы регулярное выражение или строку, определяющую регулярное выражение):
   а) `^.*$`
   б) `"\\{.+\\}"`
   в) `\d{4}-\d{2}-\d{2}`
   г) `"\\\\{4}"`
3. Создайте регулярное выражение для нахождения всех слов, которые:
   а) начинаются с согласных;
   б) содержат три или более гласные подряд;
   в) содержат две или более пары "гласная — согласная" подряд.

4. Решите кроссворды на основе регулярных выражений для начинающих, приведенные по адресу `https://regexcrossword.com/challenges/beginner`.

Группирование и обратные ссылки

Ранее вы узнали о том, что круглые скобки служат для устранения неоднозначности в сложных выражениях. Кроме того, круглые скобки определяют *группы*, на которые можно ссылаться с помощью обратных ссылок наподобие `\1`, `\2` и т.д. Например, следующее регулярное выражение находит все фрукты, названия которых содержат повторяющиеся пары букв (рис. 11.19):

```
str_view(fruit, "(..)\\1", match = TRUE)
```

```
banana
coconut
cucumber
jujube
papaya
salal berry
```

Рис. 11.19

Вскоре вы убедитесь в том, насколько полезны группы в сочетании с функцией `str_match()`.

Упражнения

1. Опишите на словах, с чем будут совпадать эти выражения:
   а) `(.)\1\1`
   б) `"(.)(.)\\2\\1"`
   в) `(..)\1`
   г) `"(.).\\1.\\1"`
   д) `"(.)(.)(.).*\\3\\2\\1"`
2. Сконструируйте регулярные выражения, совпадающие со словами, которые:

а) начинаются и оканчиваются одной и той же буквой;

б) содержат повторяющуюся пару букв (например, в слове “church” сочетание “ch” встречается дважды);

в) содержат одну букву, повторенную по крайней мере в трех позициях (например, в слове “eleven” буква “e” встречается три раза).

Инструментарий

Теперь, когда вы познакомились с основами регулярных выражений, мы можем заняться изучением способов их применения для решения практических задач. В этом разделе рассматривается широкий спектр функций пакета **stringr**, которые позволяют:

- определять, какие строки соответствуют шаблону;
- находить позиции совпадений;
- извлекать содержимое совпадений;
- заменять совпадения новыми значениями;
- разбивать строку на части на основе совпадений.

Прежде чем мы продолжим, хочу вас предостеречь: учитывая мощь регулярных выражений, нетрудно поддаться соблазну и пытаться решать любую задачу с помощью единственного регулярного выражения. Здесь будет уместно процитировать Джейми Завински:

> Некоторые люди, столкнувшись с проблемой, думают: “О, а использую-ка я регулярные выражения. Теперь у них есть две проблемы.

В качестве подтверждения этого взгляните на приведенный ниже пример регулярного выражения, предназначенного для проверки действительности адресов электронной почты (рис. 11.20).

В некотором смысле это патологический пример (что обусловлено действительной сложностью адресов электронной почты), но он используется в реальном коде. Более подробное его обсуждение можно найти на сайте Stackoverflow по следующему адресу:

```
http://stackoverflow.com/questions/201323/
using-a-regular-expression-to-validate-an-email-address/201378#201378)
```

```
(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\
[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\
".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])
+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:
[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\]
(?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\
[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\
".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:
\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\
["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\
".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]))*\>(?:(?:\r\n)?
[ \t])*)|(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:
[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*:(?:(?:\r\n)?[ \t])*(?:(?:(?:[^()<>@,;:\\".\[\]
\000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*))*@(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t]))*)|(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.
)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.
)*\](?:(?:\r\n)?[ \t])*))*(?:,\s*(?:(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:
(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?
[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\
\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-
\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)|
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|
(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:
(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:
\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^
[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:
\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*|(?:[^()<>@,;:
\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?
[ \t]))*"(?:(?:\r\n)?[ \t])*)*\<(?:(?:\r\n)?[ \t])*(?:@(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?[ \t])*
(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.
)*\](?:(?:\r\n)?[ \t])*))*(?:,@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*)(?:\.(?:(?:\r\n)?
[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\
\]|\\.)*\](?:(?:\r\n)?[ \t])*))*:(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?
[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|"(?:[^\"\r\\]|\\.|(?:(?:\r\n)?[ \t]))*"(?:(?:\r\n)?[ \t])*)(?:\.(?:
(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\
[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*@(?:(?:\r\n)?[ \t])*(?:[^()<>@,;:\\".\[\] \000-\031]+(?:(?:
(?:\r\n)?[ \t])+|\Z|(?=[\["()<>@,;:\\".\[\]]))|\[([^\[\]\r\\]|\\.)*\](?:(?:\r\n)?[ \t])*))*\>(?:(?:\r\n)?
[ \t])*)?;\s*)
```

Рис. 11.20

Не забывайте о том, что речь идет о языке программирования, и в вашем распоряжении имеются другие инструменты. Часто вместо того, чтобы создавать одно сложное регулярное выражение, легче создать несколько более

простых. Если вы зашли в тупик, безуспешно пытаясь найти выражение, которое решит все ваши проблемы, приостановите попытки и подумайте, нельзя ли разбить задачу на более мелкие части, которые можно решать последовательно, одну за другой.

Обнаружение совпадений

Чтобы определить, соответствует ли символьный вектор шаблону, используйте функцию `str_detect()`. Она возвращает логический вектор той же длины, что и входной.

```
x <- c("apple", "banana", "pear")
str_detect(x, "e")
#> [1]  TRUE FALSE  TRUE
```

Вспомните, что при использовании логического вектора в числовом контексте `FALSE` превращается в 0, а `TRUE` — в 1. В силу этого, если вы заинтересованы в получении ответов на вопросы, касающиеся совпадений в большом векторе, вам пригодятся функции `sum()` и `mean()`.

```
# Сколько слов начинаются с буквы t?
sum(str_detect(words, "^t"))
#> [1] 65
# Какова доля слов, оканчивающихся на гласную?
mean(str_detect(words, "[aeiou]$"))
#> [1] 0.277
```

В случае сложных логических условий зачастую проще объединить несколько вызовов функции `str_detect()` с логическими операторами, чем пытаться создать одно сложное регулярное выражение. Например, вот два способа поиска всех слов, которые вообще не содержат гласных.

```
# Найти все слова, содержащие хотя бы одну гласную,
# и инвертировать эту операцию
no_vowels_1 <- !str_detect(words, "[aeiou]")
# Найти все слова, состоящие только из согласных
# (символов, не являющихся гласными)
no_vowels_2 <- str_detect(words, "^[^aeiou]+$")
identical(no_vowels_1, no_vowels_2)
#> [1] TRUE
```

Оба способа приводят к одному и тому же результату, но, с нашей точки зрения, первый подход более понятен. Если ваше регулярное выражение чрезмерно усложняется, попробуйте разбить его на меньшие части, присвоив каждой из них отдельное имя, а затем объедините их логическими операциями.

Как правило, функция `str_detect()` используется для выбора элементов, соответствующих шаблону. То же самое можно сделать с помощью удобной функции-оболочки `str_subset()`.

```
words[str_detect(words, "x$")]
#> [1] "box" "sex" "six" "tax"
str_subset(words, "x$")
#> [1] "box" "sex" "six" "tax"
```

Однако обычно вашей строкой будет один из столбцов фрейма данных, и вместо этого вы будете использовать фильтр.

```
df <- tibble(
  word = words,
  i = seq_along(word)
)
df %>%
  filter(str_detect(words, "x$"))
#> # A tibble: 4 × 2
#>    word     i
#>   <chr> <int>
#> 1   box   108
#> 2   sex   747
#> 3   six   772
#> 4   tax   841
```

Видоизменением функции `str_detect()` является функция `str_count()`: вместо простого “да” или “нет” она сообщает, сколько совпадений обнаружено в строке.

```
x <- c("apple", "banana", "pear")
str_count(x, "a")
#> [1] 1 3 1

# Сколько гласных приходится в среднем на одно слово?
mean(str_count(words, "[aeiou]"))
#> [1] 1.99
```

Функцию `str_count()` вполне естественно использовать совместно с функцией `mutate()` (рис. 11.21).

```
df %>%
  mutate(
    vowels = str_count(word, "[aeiou]"),
    consonants = str_count(word, "[^aeiou]")
  )
#> # A tibble: 980 × 4
#>       word     i vowels consonants
```

```
#>       <chr> <int>  <int>      <int>
#> 1         a     1      1          0
#> 2      able     2      2          2
#> 3     about     3      3          2
#> 4 absolute     4      4          4
#> 5    accept     5      2          4
#> 6   account     6      3          4
#> # ... with 974 more rows
```

abababa

Рис. 11.21

Заметьте, что совпадения никогда не перекрываются. Например, сколько раз в строке `"abababa"` встречается шаблон `"aba"`? Регулярные выражения скажут вам, что он содержится два раза, а не три (рис. 11.22).

```
str_count("abababa", "aba")
#> [1] 2
str_view_all("abababa", "aba")
```

abababa

Рис. 11.22

Обратите внимание на использование функции `str_view_all()`. Как вы вскоре узнаете, многие функции пакета **stringr** идут парами: одна функция работает с одиночными совпадениями, другая — со всеми. Вторая функция всегда имеет суффикс `suffix_all`.

Упражнения

1. Попытайтесь решить каждую из приведенных ниже задач, используя как одиночное регулярное выражение, так и сочетание нескольких вызовов функции `str_detect()`.

   А. Найдите все слова, которые начинаются или оканчиваются на букву `x`.

   Б. Найдите все слова, которые начинаются на гласную, а оканчиваются согласной.

   В. Существуют ли слова, в которых каждая гласная содержится по крайней мере один раз?

   Г. Какое слово содержит наибольшее количество гласных? В каком слове наибольшая доля гласных? (Подсказка: что служит знаменателем?)

Извлечение совпадений

Для извлечения фактического текста совпадения используйте функцию `str_extract()`. Чтобы продемонстрировать это, нам нужен более сложный пример. Мы воспользуемся текстом *Harvard sentences* (https://en.wikipedia.org/wiki/Harvard_sentences), который предназначен для тестирования систем голосовой связи, работающих через Интернет, но его также удобно использовать для тестирования регулярных выражений. Предложения, составляющие этот текст, предоставляются пакетом `stringr::sentences`.

```
length(sentences)
#> [1] 720
head(sentences)
#> [1] "The birch canoe slid on the smooth planks."
#> [2] "Glue the sheet to the dark blue background."
#> [3] "It's easy to tell the depth of a well."
#> [4] "These days a chicken leg is a rare dish."
#> [5] "Rice is often served in round bowls."
#> [6] "The juice of lemons makes fine punch."
```

Предположим, вы хотите найти все предложения, в которых упоминается цвет. Сначала мы создаем вектор названий цветов, а затем превращаем его в одиночное регулярное выражение.

```
colors <- c(
  "red", "orange", "yellow", "green", "blue", "purple"
)
color_match <- str_c(colors, collapse = "|")
color_match
#> [1] "red|orange|yellow|green|blue|purple"
```

Теперь мы можем выбрать все предложения, в которых упоминается какой-либо цвет, а затем извлечь каждый цвет, чтобы увидеть, что это за цвета.

```
has_color <- str_subset(sentences, color_match)
matches <- str_extract(has_color, color_match)
head(matches)
#> [1] "blue" "blue" "red"  "red"  "red"  "blue"
```

Заметьте, что функция `str_extract()` извлекает лишь первое совпадение. Это легче всего увидеть, если выбрать сначала все предложения, в которых имеется более одного совпадения (рис. 11.23).

```
more <- sentences[str_count(sentences, color_match) > 1]
str_view_all(more, color_match)
```

Упражнения

1. Возможно, вы заметили, что регулярное выражение совпало со словом "flickered", которое не обозначает цвет. Измените регулярное выражение таким образом, чтобы устранить этот недостаток.
2. Извлеките из текста *Harvard sentences*:
   а) первое слово каждого предложения;
   б) все слова, оканчивающиеся на `ing`;
   в) все слова во множественном числе.

Разбиение совпадений на группы

Ранее в этой главе мы обсуждали использование круглых скобок для прояснения приоритетов и обеспечения возможности обратных ссылок в процессе обнаружения совпадений. Кроме того, круглые скобки можно использовать для извлечения частей сложных совпадений. Предположим, вы хотите извлечь существительные из предложений. Нашей эвристикой будет поиск любого слова, следующего за артиклем "a" или "the". Определение "слова" в регулярном выражении — не совсем простая штука, поэтому мы используем простое приближение: слово — это последовательность, состоящая по крайней мере из одного символа, не являющегося пробелом.

```
noun <- "(a|the) ([^ ]+)"

has_noun <- sentences %>%
  str_subset(noun) %>%
  head(10)
has_noun %>%
  str_extract(noun)
#>  [1] "the smooth" "the sheet"  "the depth"  "a chicken"
#>  [5] "the parked" "the sun"    "the huge"   "the ball"
#>  [9] "the woman"  "a helps"
```

Функция `str_extract()` дает нам полное совпадение, функция `str_match()` — каждый отдельный его компонент. Вместо символьного вектора она возвращает матрицу с одним столбцом для полного совпадения, за которым следуют столбцы, по одному для каждой группы.

```
has_noun %>%
  str_match(noun)
#>       [,1]         [,2]  [,3]
#>  [1,] "the smooth" "the" "smooth"
```

```
#>  [2,] "the sheet"  "the" "sheet"
#>  [3,] "the depth"  "the" "depth"
#>  [4,] "a chicken"  "a" "chicken"
#>  [5,] "the parked" "the" "parked"
#>  [6,] "the sun" "the" "sun"
#>  [7,] "the huge" "the" "huge"
#>  [8,] "the ball" "the" "ball"
#>  [9,] "the woman"  "the" "woman"
#> [10,] "a helps" "a" "helps"
```

(Неудивительно, что наша эвристика, которую мы используем для обнаружения существительных, работает плохо, выбирая заодно такие прилагательные, как smooth и parked.)

Если ваши данные содержатся в tibble-фрейме, зачастую проще использовать функцию tidyr::extract(). Она работает подобно функции str_match(), но требует, чтобы вы именовали совпадения, которые затем помещаются в новые столбцы.

```
tibble(sentence = sentences) %>%
  tidyr::extract(
    sentence, c("article", "noun"), "(a|the) ([^ ]+)",
    remove = FALSE
  )
#> # A tibble: 720 × 3
#>                                   sentence  article    noun
#> *                                    <chr>    <chr>   <chr>
#> 1  The birch canoe slid on the smooth planks.     the  smooth
#> 2 Glue the sheet to the dark blue background.     the   sheet
#> 3      It's easy to tell the depth of a well.     the   depth
#> 4    These days a chicken leg is a rare dish.       a chicken
#> 5        Rice is often served in round bowls.    <NA>    <NA>
#> 6       The juice of lemons makes fine punch.    <NA>    <NA>
#> # ... with 714 more rows
```

Как и в случае функции str_extract(), если вы хотите получить все совпадения, следует использовать функцию str_match_all().

Упражнения

1. Найдите все слова, которым предшествуют “числа”, такие как “one”, “two”, “three” и т.д. Извлеките как число, так и слово.
2. Найдите все сокращения. Отделите части, располагающиеся до и после апострофа.

Замена совпадений

Функции str_replace() и str_replace_all() позволяют заменять совпадения новыми строками. Проще всего заменить шаблон фиксированной строкой.

```
x <- c("apple", "pear", "banana")
str_replace(x, "[aeiou]", "-")
#> [1] "-pple"  "p-ar"   "b-nana"
str_replace_all(x, "[aeiou]", "-")
#> [1] "-ppl-"  "p--r"   "b-n-n-"
```

С помощью функции str_replace_all() можно выполнять множественную замену, предоставляя именованный вектор.

```
x <- c("1 house", "2 cars", "3 people")
str_replace_all(x, c("1" = "one", "2" = "two", "3" = "three"))
#> [1] "one house"    "two cars"     "three people"
```

Вместо замены совпадений фиксированной строкой можно использовать обратные ссылки для вставки компонентов совпадений. В следующем коде мы меняем порядок следования второго и третьего слов.

```
sentences %>%
  str_replace("([^ ]+) ([^ ]+) ([^ ]+)", "\\1 \\3 \\2") %>%
  head(5)
#> [1] "The canoe birch slid on the smooth planks."
#> [2] "Glue sheet the to the dark blue background."
#> [3] "It's to easy tell the depth of a well."
#> [4] "These a days chicken leg is a rare dish."
#> [5] "Rice often is served in round bowls."
```

Упражнения

1. Замените в строке все символы косой черты символами обратной косой черты.
2. Реализуйте простую версию функции str_to_lower(), используя функцию replace_all().
3. Поменяйте местами первую и последнюю буквы слов. Какие из строк все еще остаются словами?

Разбиение строк

Используйте функцию str_split() для разбиения строки на части. Например, мы можем разбить предложения на слова.

```
sentences %>%
  head(5) %>%
  str_split(" ")
#> [[1]]
#> [1] "The"     "birch"   "canoe"   "slid"    "on"      "the"
#> [7] "smooth"  "planks."
#>
#> [[2]]
#> [1] "Glue"        "the"         "sheet"       "to"
#> [5] "the"         "dark"        "blue"        "background."
#>
#> [[3]]
#> [1] "It's"  "easy"  "to"    "tell"  "the"   "depth" "of"
#> [8] "a"   "well."
#>
#> [[4]]
#> [1] "These"   "days"    "a"       "chicken" "leg"     "is"
#> [7] "a"       "rare"    "dish."
#>
#> [[5]]
#> [1] "Rice"   "is"     "often"  "served" "in"     "round"
#> [7] "bowls."
```

Поскольку каждый компонент может состоять из различного количества частей, возвращается список. Если вы работаете с вектором длиной 1, то самый простой способ состоит в том, чтобы извлечь первый элемент списка.

```
"a|b|c|d" %>%
str_split("\\|") %>%
.[[1]]
#> [1] "a" "b" "c" "d"
```

В противном случае, как и при работе с другими функциями, возвращающими список, можно использовать параметр `simplify = TRUE`, чтобы вернуть матрицу.

```
sentences %>%
  head(5) %>%
  str_split(" ", simplify = TRUE)
#>      [,1]    [,2]    [,3]    [,4]      [,5]  [,6]    [,7]
#> [1,] "The"   "birch" "canoe" "slid"    "on"  "the"   "smooth"
#> [2,] "Glue"  "the"   "sheet" "to"      "the" "dark"  "blue"
#> [3,] "It's"  "easy"  "to"    "tell"    "the" "depth" "of"
#> [4,] "These" "days"  "a"     "chicken" "leg" "is"    "a"
#> [5,] "Rice"  "is"    "often" "served"  "in"  "round" "bowls."
#>      [,8]          [,9]
#> [1,] "planks."     ""
#> [2,] "background." ""
```

```
#> [3,] "a"          "well."
#> [4,] "rare"       "dish."
#> [5,] ""           ""
```

Вы также можете запросить максимальное количество частей.

```
fields <- c("Name: Hadley", "Country: NZ", "Age: 35")
fields %>% str_split(": ", n = 2, simplify = TRUE)
#>      [,1]      [,2]
#> [1,] "Name"    "Hadley"
#> [2,] "Country" "NZ"
#> [3,] "Age"     "35"
```

Вместо разбивки строк по шаблонам можно разбивать их по границам символов, строк, предложений и слов (рис. 11.24).

```
x <- "This is a sentence.  This is another sentence."
str_view_all(x, boundary("word"))
```

This is a sentence. This is another sentence.

Рис. 11.24

```
str_split(x, " ")[[1]]
#> [1] "This"      "is"        "a"         "sentence." ""
#> [6]         "This"
#> [7] "is"        "another"   "sentence."
str_split(x, boundary("word"))[[1]]
#> [1] "This"     "is"       "a"        "sentence" "This"
#> [6] "is"
#> [7] "another"  "sentence"
```

Упражнения

1. Разбейте строку `"apples, pears, and bananas"` на отдельные компоненты.
2. Почему разбивать строки на слова лучше по границам слов (`boundary("word")`), чем по пробелам (`" "`)?
3. Что произойдет при попытке разбить строку по пустой строке (`""`)? Сначала проведите самостоятельные эксперименты, а затем прочитайте документацию.

Поиск совпадений

Функции `str_locate()` и `str_locate_all()` предоставляют начальную и конечную позиции каждого совпадения. Это особенно полезно в тех случаях, когда ни одна из других функций не делает в точности то, что вам нужно.

Функцию `str_locate()` можно использовать для поиска совпадающих шаблонов, а функцию `str_sub()` — для их извлечения и (или) изменения.

Другие типы шаблонов

Когда вы используете шаблон, являющийся строкой, она автоматически помещается в вызов функции `regex()`.

```
# Обычный вызов:
str_view(fruit, "nana")
# является сокращенной записью вызова
str_view(fruit, regex("nana"))
```

Передавая функции `regex()` дополнительные аргументы, можно управлять деталями сопоставления строк с шаблоном.

- `ignore_case = TRUE` — позволяет игнорировать регистр символов при сопоставлении с шаблоном. В первом из приведенных ниже вызовов функции `str_view()` используется текущая локаль (рис. 11.25).

  ```
  bananas <- c("banana", "Banana", "BANANA")
  str_view(bananas, "banana")
  ```

  banana

  Banana

  BANANA

  Рис. 11.25

  ```
  str_view(bananas, regex("banana", ignore_case = TRUE))
  ```

- `multiline = TRUE` — разрешает специальным символам `^` и `$` совпадать с началом и с концом каждой логической строки, а не полной строки.

  ```
  x <- "Line 1\nLine 2\nLine 3"
  str_extract_all(x, "^Line")[[1]]
  #> [1] "Line"
  str_extract_all(x, regex("^Line", multiline = TRUE))[[1]]
  #> [1] "Line" "Line" "Line"
  ```

- `comments = TRUE` — разрешает использование комментариев и пробельных символов для повышения удобочитаемости сложных регулярных выражений. Пробелы, как и весь остальной текст после символа `#`, игнорируются. Чтобы литеральный символ пробела учитывался при сопоставлении, его необходимо экранировать: `"\\"`.

```
phone <- regex("
  \\(?     # optional opening parens
  (\\d{3}) # area code
  [)- ]?   # optional closing parens, dash, or space
  (\\d{3}) # another three numbers
  [ -]?    # optional space or dash
  (\\d{3}) # three more numbers
  ", comments = TRUE)

str_match("514-791-8141", phone)
#>      [,1]          [,2]  [,3]  [,4]
#> [1,] "514-791-814" "514" "791" "814"
```

- `dotall = TRUE` — разрешает символу `.` совпадать с любым символом, включая `\n`.

Существуют три другие функции, которые можно использовать вместо функции `regex()`.

- `fixed()` — ищет точные совпадения с указанной последовательностью байтов. Эта функция игнорирует все специальные регулярные выражения и работает на очень низком уровне, что позволяет избежать сложного экранирования символов и значительно повысить быстродействие. В соответствии с результатами приведенного ниже теста быстродействие в этом простом примере повысилось примерно в три раза.

```
microbenchmark::microbenchmark(
  fixed = str_detect(sentences, fixed("the")),
  regex = str_detect(sentences, "the"),
  times = 20
)
#> Unit: microseconds
#>   expr min  lq mean median  uq max neval cld
#>  fixed 116 117  136    120 125 389    20   a
#>  regex 333 337  346    338 342 467    20   b
```

Проявляйте осторожность, используя функцию `fixed()` для работы с данными, подготовленными не на английском языке. Это весьма проблематично, поскольку для представления одного и того же символа существует множество способов. Например, символ `á` можно представить двумя способами: как одиночный символ или как символ `a` плюс акцент.

```
a1 <- "\u00e1"
a2 <- "a\u0301"
c(a1, a2)
```

```
#> [1] "á" "á"
a1 == a2
#> [1] FALSE
```

Визуализированные символы выглядят одинаково в обоих случаях, но поскольку они определялись двумя разными способами, функция `fixed()` не обнаруживает совпадения. Вместо этого можно использовать функцию `coll()`, определенную далее, которая уважает принятые среди людей правила сравнения символов.

```
str_detect(a1, fixed(a2))
#> [1] FALSE
str_detect(a1, coll(a2))
#> [1] TRUE
```

- `coll()` — сравнивает строки, используя стандартные правила сравнения. Это полезно при выполнении поиска без учета регистра букв. Обратите внимание на то, что функция `coll()` принимает параметр `locale`, позволяющий определять, какие правила следует использовать при сравнении символов. К сожалению, в различных частях мира используются различные правила!

```
# Это означает, что при выполнении поиска, не зависящего от
# регистра букв, вы должны учитывать все отличия.
i <- c("I", "İ", "i", "ı")
i
#> [1] "I" "İ" "i" "ı"

str_subset(i, coll("i", ignore_case = TRUE))
#> [1] "I" "i"
str_subset(
  i,
  coll("i", ignore_case = TRUE, locale = "tr")
)
#> [1] "İ" "i"
```

Функции `fixed()` и `regex()` имеют аргумент `ignore_case`, но они не дают возможности выбирать локаль: они всегда используют локаль, заданную по умолчанию. Можете убедиться в этом на примере следующего кода (более подробно о пакете **stringi** будет говориться далее).

```
stringi::stri_locale_info()
#> $Language
#> [1] "en"
#>
#> $Country
```

```
#> [1] "US"
#>
#> $Variant
#> [1] ""
#>
#> $Name
#> [1] "en_US"
```

Слабым местом функции `coll()` является скорость работы. Поскольку правила определения эквивалентности символов чрезвычайно сложны, функция `coll()` работает медленнее по сравнению с функциями `regex()` и `fixed()`.

- В примере с функцией `str_split()` вы уже видели применение функции `boundary()` для сопоставления с границами. То же самое можно делать и с другими функциями (рис. 11.26).

```
x <- "This is a sentence."
str_view_all(x, boundary("word"))
```

This is a sentence.

Рис. 11.26

```
str_extract_all(x, boundary("word"))
#> [[1]]
#> [1] "This"     "is"       "a"        "sentence"
```

Упражнения

1. Как бы вы организовали поиск всех строк, содержащих символ \, с помощью функции `regex()` и с помощью функции `fixed()`?
2. Какие пять слов встречаются в предложениях чаще других?

Другие применения регулярных выражений

В базовый состав R входят еще две функции, которые также используют регулярные выражения.

- `apropos()` — выполняет поиск всех объектов, доступных в глобальном окружении. Эту функцию удобно использовать в тех случаях, когда вы не можете вспомнить точное имя какой-либо функции.

```
apropos("replace")
#> [1] "%+replace%"  "replace"         "replace_na"
```

```
#> [4] "str_replace" "str_replace_all" "str_replace_na"
#> [7] "theme_replace"
```

- `dir()` — перечисляет все файлы, хранящиеся в каталоге. Аргумент `pattern` принимает регулярное выражение и возвращает имена лишь тех файлов, которые соответствуют шаблону. Например, можно найти все файлы R Markdown в текущем каталоге с помощью следующего кода.

```
head(dir(pattern = "\\.Rmd$"))
#> [1] "communicate-plots.Rmd" "communicate.Rmd"
#> [3] "datetimes.Rmd" "EDA.Rmd"
#> [5] "explore.Rmd" "factors.Rmd"
```

(Если вы привыкли работать с шаблонами поиска, в которых используются метасимволы, например `*.Rmd`, преобразуйте их в регулярное выражение с помощью функции `glob2rx()`).

Пакет stringi

Пакет **stringr** построен поверх пакета **stringi**. Его удобно использовать в процессе обучения, поскольку он предоставляет минимальный набор функций, тщательно подобранных для выполнения наиболее часто используемых строковых операций. С другой стороны, пакет **stringi** проектировался так, чтобы охватить как можно более широкий круг возможностей. В нем вы найдете почти любую функцию, которая может вам понадобиться: он содержит 234 функции, в то время как пакет **stringr** — всего лишь 42.

Если вы столкнетесь с трудностями, пытаясь сделать что-то с помощью пакета **stringr**, вам стоит вспомнить о пакете **stringi**. Эти пакеты очень похожи, поэтому все, что вы знаете об одном пакете, самым естественным образом переносится на другой. Основное различие между названиями функций заключается в префиксах: `str_` и `stri_`.

Упражнения

1. Найдите функции пакета **stringi**, которые:
   а) подсчитывают количество слов;
   б) находят повторяющиеся строки;
   в) генерируют случайный текст.
2. Как управлять тем, какой язык функция `stri_sort()` будет использовать в процессе сортировки текста?

ГЛАВА 12

Работа с факторами с помощью пакета forcats

Введение

Факторы используются в R для работы с категориальными переменными, т.е. переменными, которые имеют фиксированный ряд известных возможных значений. Их также можно использовать, если вы хотите отобразить символьные векторы в порядке следования, отличающемся от алфавитного.

Исторически сложилось так, что с факторами было легче работать, чем с символами. В результате многие функции, входящие в базовый состав R, автоматически преобразуют символы в факторы. Это означает, что факторы часто появляются в тех местах, где от них фактически нет никакой пользы. К счастью, работая с библиотекой *tidyverse*, вы не должны об этом беспокоиться и можете сосредоточиться на ситуациях, в которых факторы действительно нужны.

Чтобы получить общее представление о роли факторов в историческом контексте, рекомендуем прочитать статьи Роджера Пенга *stringsAsFactors: An unauthorized biography* (http://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/) и Томаса Ламли *stringsAsFactors = <sigh>* (http://notstatschat.tumblr.com/post/124987394001/stringsasfactors-sigh).

Необходимые ресурсы

Для работы с факторами мы будем использовать пакет **forcats**, который предоставляет инструменты для обработки категориальных переменных. (Кстати, само название пакета является анаграммой на основе слова *factors*!) Этот пакет предоставляет широкий ассортимент вспомогательных функций,

предназначенных для работы с факторами. Пакет **forcats** не является частью ядра библиотеки *tidyverse*, поэтому его следует загружать явным образом.

```
library(tidyverse)
library(forcats)
```

Создание факторов

Предположим, у вас имеется переменная, в которую записываются названия месяцев:

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

С записью строки в эту переменную связаны две проблемы.

1. Существуют всего двенадцать возможных значений названий месяцев, но и это не гарантирует отсутствия опечаток:

   ```
   x2 <- c("Dec", "Apr", "Jam", "Mar")
   ```

2. Этот список не сортируется так, как нам хотелось бы.

   ```
   sort(x1)
   #> [1] "Apr" "Dec" "Jan" "Mar"
   ```

Оба этих недостатка можно легко исправить, используя фактор. Чтобы создать фактор, вы должны начать с создания списка допустимых уровней.

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

Теперь можно создать фактор.

```
y1 <- factor(x1, levels = month_levels)
y1
#> [1] Dec Apr Jan Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
sort(y1)
#> [1] Jan Mar Apr Dec
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Любые значения, не входящие в этот набор, будут автоматически преобразовываться в NA.

```
y2 <- factor(x2, levels = month_levels)
y2
#> [1] Dec  Apr  <NA> Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Если хотите получать сообщения об ошибках, можете использовать функцию `readr::parse_factor()`.

```
y2 <- parse_factor(x2, levels = month_levels)
#> Warning: 1 parsing failure.
#> row col           expected actual
#>   3  -- value in level setJam
```

Если опустить уровни, они будут браться из данных в алфавитном порядке.

```
factor(x1)
#> [1] Dec Apr Jan Mar
#> Levels: Apr Dec Jan Mar
```

Иногда предпочтительнее, чтобы порядок следования уровней совпадал с порядком первого появления в данных. Можно сделать это в процессе создания фактора путем задания уровней для функции `unique(x)` или уже постфактум с помощью функции `fct_inorder()`.

```
f1 <- factor(x1, levels = unique(x1))
f1
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar

f2 <- x1 %>% factor() %>% fct_inorder()
f2
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
```

Если вам когда-либо потребуется непосредственный доступ к набору допустимых уровней, это можно сделать с помощью функции `levels()`.

```
levels(f2)
#> [1] "Dec" "Apr" "Jan" "Mar"
```

Опрос GSS

В оставшейся части главы наше внимание будет приковано к набору данных `forcats::gss_cat`. Он представляет собой выборку данных общенационального социального опроса (General Social Survey — GSS), проводимого в США два раза в год независимой исследовательской организацией NORC, штаб-квартира которой находится в Чикагском университете. Опросный лист насчитывает несколько тысяч вопросов, поэтому в набор данных `gss_cat` включена лишь достаточно представительная выборка, иллюстрирующая ряд общих проблем, с которыми вы можете столкнуться, работая с факторами.

```
gss_cat
#> # A tibble: 21,483 × 9
#>    year       marital   age   race        rincome
#>   <int>        <fctr> <int> <fctr>         <fctr>
#> 1  2000 Never married    26  White  $8000 to 9999
#> 2  2000      Divorced    48  White  $8000 to 9999
#> 3  2000       Widowed    67  White Not applicable
#> 4  2000 Never married    39  White Not applicable
#> 5  2000      Divorced    25  White Not applicable
#> 6  2000       Married    25  White $20000 - 24999
#> # ... with 2.148e+04 more rows, and 4 more variables:
#> #   partyid <fctr>, relig <fctr>, denom <fctr>, tvhours <int>
```

(Не забывайте о том, что, поскольку этот набор данных предоставляется пакетом, более подробную информацию о переменных можно получить с помощью команды `?gss_cat`.)

Если факторы хранятся в tibble-фрейме, вы не сможете увидеть их, не предпринимая никаких мер. Один из способов увидеть их — использовать функцию `count()` (рис. 12.1).

```
gss_cat %>%
  count(race)
#> # A tibble: 3 × 2
#>     race     n
#>   <fctr> <int>
#> 1  Other  1959
#> 2  Black  3129
#> 3  White 16395
```

Или же, если отобразить это на диаграмме:

```
ggplot(gss_cat, aes(race)) +
  geom_bar()
```

По умолчанию пакет **ggplot2** опускает уровни, которые не содержат какие-либо значения. Можете принудительно задать их отображение с помощью следующей команды (рис. 12.2).

```
ggplot(gss_cat, aes(race)) +
  geom_bar() +
  scale_x_discrete(drop = FALSE)
```

Эти уровни представляют допустимые значения, которые просто не встречаются в используемом наборе данных. К сожалению, в пакете **dplyr** пока еще не предусмотрена опция `drop`, но это будет сделано в будущем.

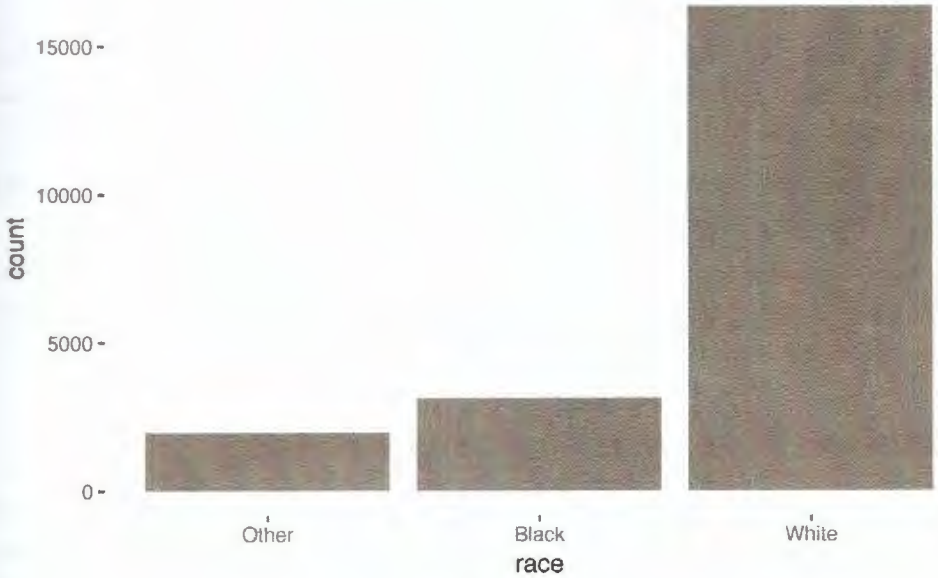


Рис. 12.1

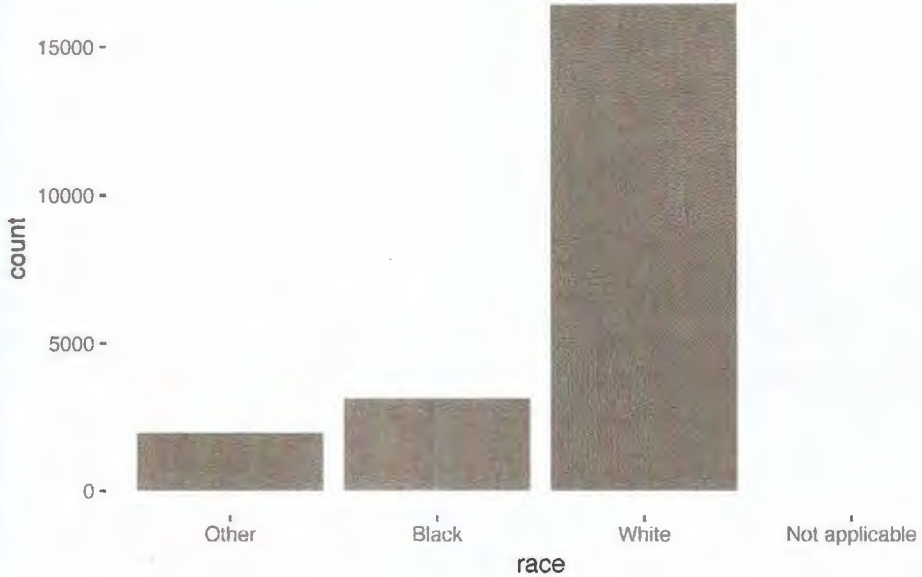


Рис. 12.2

Две чаще всего встречающиеся операции при работе с факторами — изменение порядка следования уровней и изменение их значений. Эти операции описаны в следующих разделах.

Упражнения

1. Исследуйте распределение переменной `rincome` (reported income — заявленный доход). Почему так трудно понять смысл столбчатой диаграммы, заданной по умолчанию? Как можно улучшить этот график?
2. Какое значение переменной `relig` встречается в этом опросе чаще других? Какое значение переменной `partyid` встречается чаще других?
3. К каким значениям переменной `relig` применимо понятие `denom` (denomination — конфессия)? Как это выяснить с помощью таблицы? Как это выяснить с помощью визуализации?

Изменение порядка следования факторов

Часто оказывается полезным изменить порядок следования уровней факторов в визуализации. Предположим, вы хотите исследовать среднее количество часов, посвящаемых ежедневно просмотру телевизионных передач представителями различных религий (рис. 12.3).

```
relig <- gss_cat %>%
  group_by(relig) %>%
  summarize(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
)

ggplot(relig, aes(tvhours, relig)) + geom_point()
```

Этот график трудно интерпретировать, поскольку на нем нельзя увидеть никакой общей закономерности. Ситуацию можно улучшить за счет изменения порядка следования уровней `relig`, используя функцию `fct_reorder()`, которая принимает три аргумента (рис. 12.4):

- `f` — фактор, уровни которого вы хотите изменить;
- `x` — числовой вектор, который вы хотите использовать для изменения порядка следования уровней;
- необязательный аргумент `fun` — функция, которая используется, если для каждого значения `f` существует несколько значений `x` (значение по умолчанию — `median`).

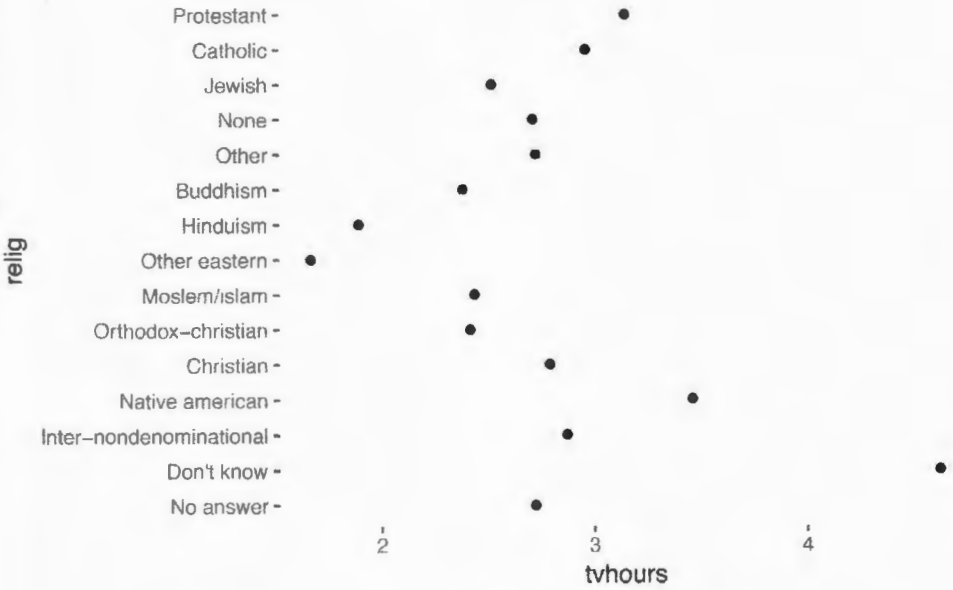


Рис. 12.3

```
ggplot(relig, aes(tvhours, fct_reorder(relig, tvhours))) +
  geom_point()
```

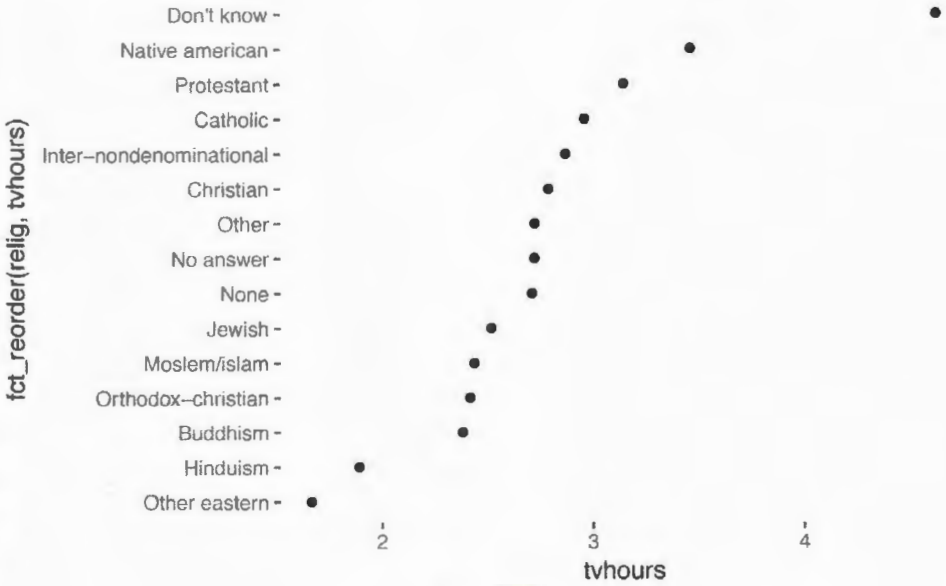


Рис. 12.4

Перестановка значений `religion` делает более очевидным тот факт, что люди категории "Don't know" ("Не знаю") чаще смотрят телевизор, тогда как приверженцы индуизма и других восточных религий смотрят телевизор реже.

Когда вы перейдете к использованию более сложных преобразований, рекомендуем переместить их из функции `aes()` в отдельный шаг вызова функции `mutate()`. Например, можно переписать код построения предыдущего графика в следующем виде.

```
relig %>%
  mutate(relig = fct_reorder(relig, tvhours)) %>%
  ggplot(aes(tvhours, relig)) +
    geom_point()
```

Что если мы создадим аналогичный график (рис. 12.5), чтобы посмотреть, как меняется средний возраст (`age`) в зависимости от заявленного уровня дохода (`rincome`)?

```
rincome <- gss_cat %>%
  group_by(rincome) %>%
  summarize(
    age = mean(age, na.rm = TRUE),
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
)

ggplot(
  rincome,
  aes(age, fct_reorder(rincome, age))
) + geom_point()
```

В данном случае произвольная перестановка порядка следования уровней — не очень хорошая идея! Причина заключается в том, что переменная `rincome` уже имеет твердый порядок следования значений, который мы не должны портить. Оставьте функцию `fct_reorder()` для факторов, уровни которых упорядочиваются произвольно.

В то же время имеет смысл переместить уровень "Not applicable" ("Неприменимо") в начало вместе с другими специальными уровнями. Для этого можно использовать функцию `fct_relevel()`. В качестве аргументов она принимает фактор, `f`, и любое количество уровней, которые вы хотите переместить в начало линии (рис. 12.6).

```
ggplot(
  rincome,
```

```
aes(age, fct_relevel(rincome, "Not applicable"))
+
geom_point()
```

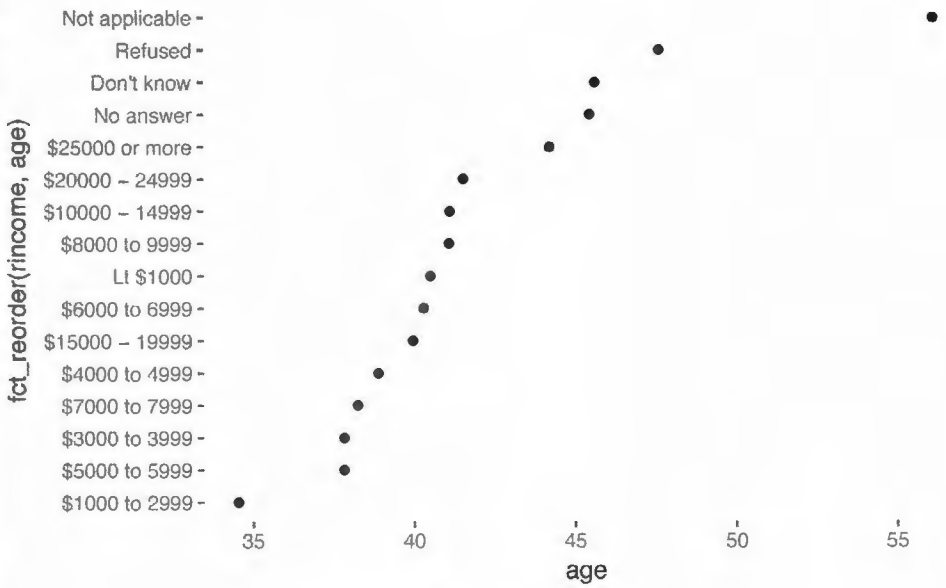


Рис. 12.5

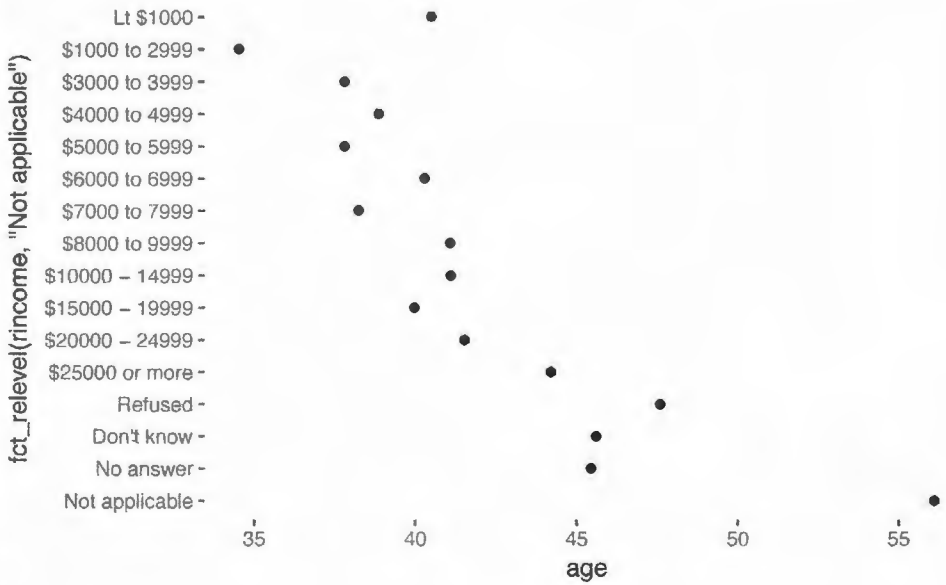


Рис. 12.6

Как вы думаете, чем объясняется столь высокое значение для уровня "Not applicable"?

Другой тип переупорядочения уровней полезен в тех случаях, когда вы выделяете цветом линии на графике. Функция `fct_reorder2()` переупорядочивает факторы по значениям `y`, связанным с наибольшими значениями `x`. Это облегчает чтение графика, поскольку цвета линий выравниваются с легендой (рис. 12.7 и 12.8).

```
by_age <- gss_cat %>%
  filter(!is.na(age)) %>%
  group_by(age, marital) %>%
  count() %>%
  mutate(prop = n / sum(n))

ggplot(by_age, aes(age, prop, color = marital)) +
  geom_line(na.rm = TRUE)

ggplot(
  by_age,
  aes(age, prop, color = fct_reorder2(marital, age, prop))
) +
  geom_line() +
  labs(color = "marital")
```

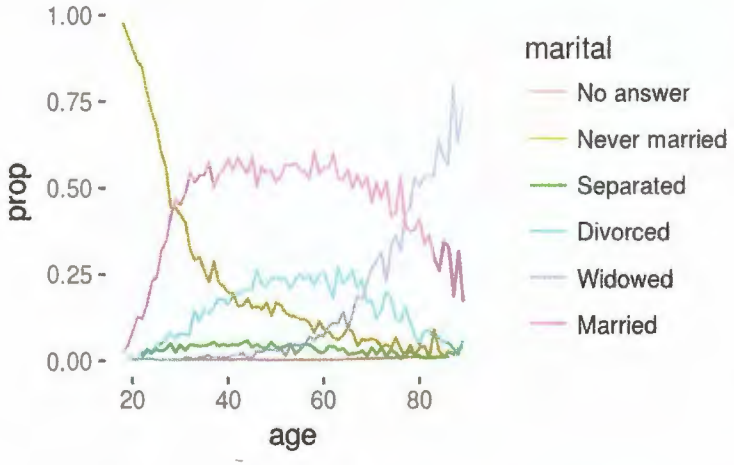


Рис. 12.7

Наконец, в случае столбчатых диаграмм можно использовать функцию `fct_infreq()` для упорядочивания уровней по возрастанию: это простейший тип переупорядочения, поскольку он не требует никаких дополнительных

переменных (рис. 12.9). Возможно, вы захотите использовать ее в сочетании с функцией `fct_rev()`.

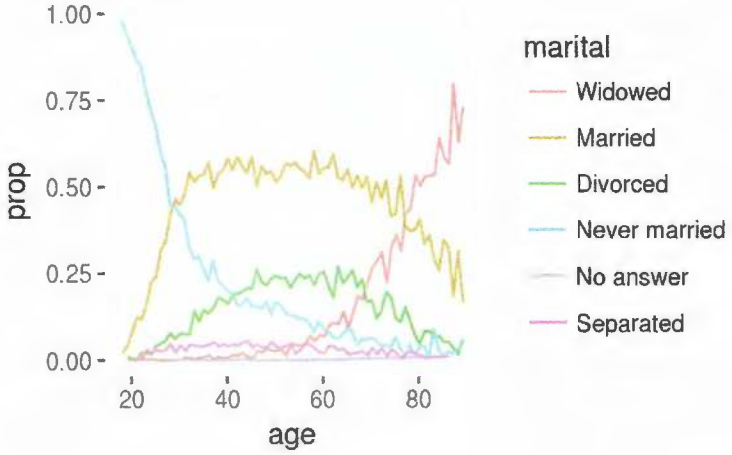


Рис. 12.8

```
gss_cat %>%
  mutate(marital = marital %>% fct_infreq() %>% fct_rev()) %>%
  ggplot(aes(marital)) +
    geom_bar()
```

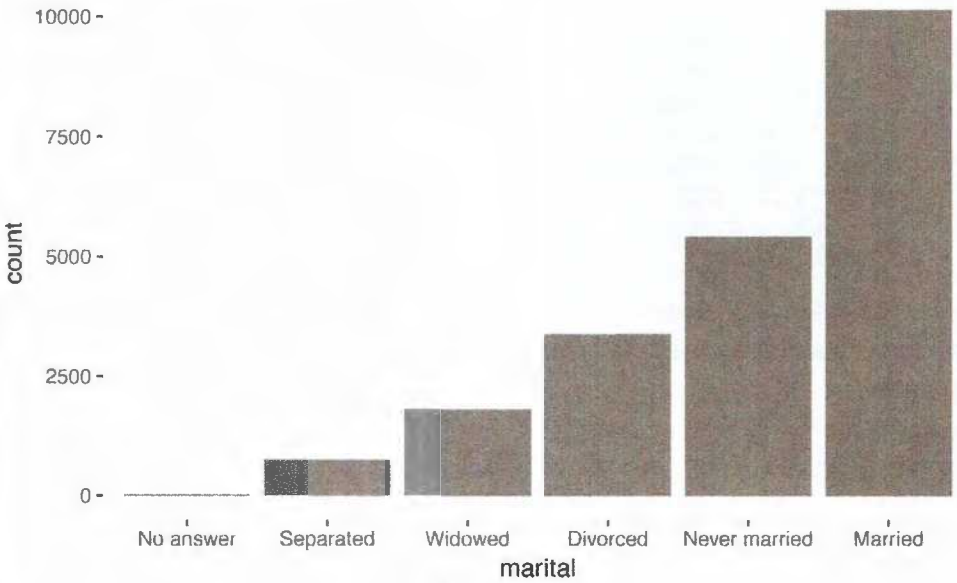


Рис. 12.9

Упражнения

1. Среди данных, касающихся времени, проводимого за телевизором, имеются подозрительно высокие цифры. Является ли среднее время хорошей итоговой характеристикой?
2. Укажите для каждого фактора в наборе данных `gss_cat`, является ли порядок следования его уровней твердым или произвольным.
3. Почему перемещение "Not applicable" в начало списка уровней приводит к его перемещению в нижнюю часть графика?

Изменение уровней факторов

Более мощным по сравнению с изменением порядка следования уровней приемом является изменение их значений. Это позволяет сделать более понятными подписи для публикаций и свертывать уровни, создавая уровни более высокого порядка для отображения. Наиболее общим и располагающим наибольшими возможностями инструментом является функция `fct_recode()`. Она позволяет перекодировать, т.е. изменять, значения каждого уровня. Возьмем, например, `gss_cat$partyid`.

```
gss_cat %>% count(partyid)
#> # A tibble: 10 × 2
#>              partyid     n
#>               <fctr> <int>
#> 1          No answer   154
#> 2         Don't know     1
#> 3        Other party   393
#> 4  Strong republican  2314
#> 5 Not str republican  3032
#> 6       Ind,near rep  1791
#> # ... with 4 more rows
```

Содержащаяся в названиях уровней информация сжата и непоследовательна. Давайте немного расширим их и используем параллельную конструкцию.

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
```

```
    "Democrat, strong"      = "Strong democrat"
  )) %>%
  count(partyid)
=> # A tibble: 10 × 2
=>                 partyid     n
=>                  <fctr> <int>
=> 1             No answer   154
=> 2            Don't know     1
=> 3           Other party   393
=> 4    Republican, strong  2314
=> 5      Republican, weak  3032
=> 6 Independent, near rep  1791
=> # ... with 4 more rows
```

Функция `fct_recode()` оставляет все уровни, которые не были явно упомянуты, в том виде, как они есть, и предупредит вас, если вы случайно сошлетесь на несуществующий уровень.

Можно создавать группы, приписывая несколько прежних уровней одному и тому же новому уровню.

```
gss_cat %>%
  mutate(partyid = fct_recode(partyid,
    "Republican, strong"    = "Strong republican",
    "Republican, weak"      = "Not str republican",
    "Independent, near rep" = "Ind,near rep",
    "Independent, near dem" = "Ind,near dem",
    "Democrat, weak"        = "Not str democrat",
    "Democrat, strong"      = "Strong democrat",
    "Other"                 = "No answer",
    "Other"                 = "Don't know",
    "Other"                 = "Other party"
  )) %>%
  count(partyid)
=> # A tibble: 8 × 2
=>                partyid     n
=>                 <fctr> <int>
=> 1                Other   548
=> 2   Republican, strong  2314
=> 3     Republican, weak  3032
=> 4Independent, near rep  1791
=> 5          Independent  4119
=> 6Independent, near dem  2499
=> # ... with 2 more rows
```

Эту методику следует применять с осторожностью: группируя вместе категории, которые в действительности являются различными, вы получите результаты, которые будут вводить в заблуждение.

Если вы планируете свернуть значительное количество уровней, функция `fct_collapse()` — полезный видоизмененный вариант функции `fct_recode()`. Для каждой новой переменной можно предоставить вектор прежних уровней.

```
gss_cat %>%
  mutate(partyid = fct_collapse(partyid,
    other = c("No answer", "Don't know", "Other party"),
    rep = c("Strong republican", "Not str republican"),
    ind = c("Ind,near rep", "Independent", "Ind,near dem"),
    dem = c("Not str democrat", "Strong democrat")
  )) %>%
  count(partyid)
#> # A tibble: 4 × 2
#>   partyid     n
#>    <fctr> <int>
#> 1   other   548
#> 2     rep  5346
#> 3     ind  8409
#> 4     dem  7180
```

Иногда вы захотите просто объединить все небольшие группы для того, чтобы упростить график или таблицу. Это работа функции `fct_lump()`.

```
gss_cat %>%
  mutate(relig = fct_lump(relig)) %>%
  count(relig)
#> # A tibble: 2 × 2
#>        relig     n
#>       <fctr> <int>
#> 1 Protestant 10846
#> 2      Other 10637
```

Поведением по умолчанию является последовательное объединение наименьших групп, гарантирующее, что агрегат все еще является наименьшей группой. В данном случае это нецелесообразно: несмотря на то, что в этом опросе большинство американцев действительно являются протестантами, мы, вероятно, перестарались со свертыванием уровней.

Вместо этого мы можем использовать параметр `n`, чтобы указать, сколько групп (исключая другие) мы хотим сохранить.

```
gss_cat %>%
  mutate(relig = fct_lump(relig, n = 10)) %>%
  count(relig, sort = TRUE) %>%
  print(n = Inf)
#> # A tibble: 10 × 2
```

```
=>                      relig     n
=>                     <fctr> <int>
=> 1               Protestant 10846
=> 2                 Catholic  5124
=> 3                     None  3523
=> 4                Christian   689
=> 5                    Other   458
=> 6                   Jewish   388
=> 7                 Buddhism   147
=> 8  Inter-nondenominational   109
=> 9             Moslem/islam   104
=> 10      Orthodox-christian    95
```

Упражнения

1. Как относительные доли людей, идентифицирующих себя как демократы, республиканцы и независимые, изменяются со временем?
2. Как бы вы свернули `rincome` в небольшой набор категорий?

ГЛАВА 13

Работа с датами и временем с помощью пакета lubridate

Введение

В этой главе продемонстрировано, как работать со значениями даты и времени в R. На первый взгляд, работа с датами и временем не должна вызывать затруднений. Мы регулярно пользуемся их значениями в повседневной жизни, и здесь, казалось бы, не должно возникать никакой путаницы. Однако, чем больше вы узнаете о переменных, описывающих дату и время, тем более сложными они вам будут казаться. В качестве разминки попробуйте ответить на следующие три вопроса, которые выглядят довольно простыми.

- В каждом ли году содержится 365 дней?
- В каждых ли сутках содержится 24 часа?
- В каждой ли минуте содержится 60 секунд?

Я уверен, вам известно о том, что не каждый год состоит из 365 дней, но знаете ли вы полный набор правил, позволяющих проверить, является ли данный год високосным? (Это правило включает три части.) Наверное, вы знаете, что во многих странах используется переход на летнее время, поэтому в них одни сутки могут содержать 23 часа, другие — 25. Возможно, вы не знаете, что некоторые минуты содержат 61 секунду, поскольку для учета постепенного замедления скорости вращения Земли иногда к минутам приходится добавлять дополнительные секунды.

Работать со значениями даты и времени непросто, поскольку они должны согласовывать два физических явления (вращение Земли вокруг собственной оси и обращение Земли вокруг Солнца) с множеством явлений

геополитического характера, включая определения месяцев, часовые пояса и переход на летнее время. В этой главе мы не будем разбирать работу с датами и временем до мельчайших подробностей, но вы получите солидную основу для развития необходимых практических навыков, которые облегчат вам решение различных задач, связанных с анализом временных характеристик.

Необходимые ресурсы

В этой главе мы сфокусируем внимание на пакете **lubridate**, упрощающем работу с датами и временем в R. Пакет **lubridate** не является частью ядра библиотеки *tidyverse*, поскольку необходимость в нем возникает лишь тогда, когда вы работаете с датами и временем. В качестве источника данных для работы нам также понадобится пакет **nycflights13**.

```
library(tidyverse)

library(lubridate)
library(nycflights13)
```

Создание переменных, описывающих дату и время

Существуют три типа данных ссылающихся на момент времени:

- *дата* — календарная дата, tibble-фреймы выводят этот тип как `<date>`;
- *время* — время суток, tibble-фреймы выводят этот тип как `<time>`;
- *дата/время* — календарная дата плюс время суток: однозначно идентифицирует момент времени (обычно с точностью до ближайшей секунды); tibble-фреймы выводят этот тип как `<dttm>` (в R он фигурирует также под названием POSIXct, но мы не считаем это название особенно полезным).

В этой главе мы сосредоточимся лишь на двух типах: дата и дата/время, поскольку в R не предусмотрен собственный класс, предназначенный для хранения значений времени суток. Если у вас возникнет необходимость в таком классе, воспользуйтесь пакетом **hms**.

Вы всегда должны стараться использовать простейший из возможных классов, удовлетворяющих ваши потребности. Это означает, что если вы можете использовать значения даты вместо значений даты/времени, то так и следует поступать. Значения даты/времени отличаются значительно большей

сложностью ввиду необходимости обработки часовых поясов, к обсуждению которых мы вернемся в конце главы.

Для получения значения текущей даты или даты/времени можно использовать функцию `today()` или `now()`.

```
today()
=> [1] "2016-10-10"
now()
=> [1] "2016-10-10 15:19:39 PDT"
```

В других случаях можно воспользоваться одним из трех способов создания переменных даты/времени;

- на основе строки;
- на основе индивидуальных компонентов даты/времени;
- на основе существующего объекта даты/времени.

Рассмотрим, как они работают.

Создание значений даты/времени на основе строк

Данные, описывающие дату и время, часто поступают в виде строк. С одним из подходов к синтаксическому разбору строк для извлечения значений даты/времени вы познакомились в главе 8. Другой подход основан на использовании вспомогательных функций, предоставляемых пакетом **lubridate**. Они автоматически устанавливают нужный формат, коль скоро вы указываете порядок следования компонентов. Чтобы использовать эти функции, достаточно определить порядок следования значений года, месяца и дня в ваших данных, а затем расставить буквы “y”, “m” и “d” в том же порядке. Последовательность этих букв определяет имя функции пакета **lubridate**, которая проанализирует вашу дату.

```
ymd("2017-01-31")
=> [1] "2017-01-31"
mdy("January 31st, 2017")
=> [1] "2017-01-31"
dmy("31-Jan-2017")
=> [1] "2017-01-31"
```

Эти функции также принимают числа, не заключенные в кавычки. Это наиболее компактный способ создания одиночного объекта даты/времени, который может вам понадобиться при фильтрации таких данных. Название функции `ymd()` краткое и исключает неоднозначное толкование.

```
ymd(20170131)
#> [1] "2017-01-31"
```

Функция `ymd()` и родственные ей функции создают даты. Чтобы создать дату/время, добавьте в имя функции-парсера символ подчеркивания и одну или несколько дополнительных букв “h”, “m”, and “s”, которым соответствуют часы, минуты и секунды.

```
ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"
```

Вы также можете принудительно создать дату/время на основе даты, предоставив значение часового пояса.

```
ymd(20170131, tz = "UTC")
#> [1] "2017-01-31 UTC"
```

Создание значений даты/времени на основе отдельных компонентов

Иногда вместо отдельной строки вы будете располагать отдельными компонентами даты/времени, предоставленными в нескольких столбцах. Именно такой вариант реализован в наборе данных `flights`.

```
flights %>%
  select(year, month, day, hour, minute)
#> # A tibble: 336,776 × 5
#>    year month   day  hour minute
#>   <int> <int> <int> <dbl>  <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # ... with 3.368e+05 more rows
```

Чтобы создать значения даты/времени в случае входных данных такого рода, используйте функцию `make_date()` для даты и `make_datetime()` для даты/времени.

```
flights %>%
  select(year, month, day, hour, minute) %>%
  mutate(
    departure = make_datetime(year, month, day, hour, minute)
```

```
)
#> # A tibble: 336,776 × 6
#>    year month   day  hour minute           departure
#>   <int> <int> <int> <dbl>  <dbl>              <dttm>
#> 1  2013     1     1     5     15 2013-01-01 05:15:00
#> 2  2013     1     1     5     29 2013-01-01 05:29:00
#> 3  2013     1     1     5     40 2013-01-01 05:40:00
#> 4  2013     1     1     5     45 2013-01-01 05:45:00
#> 5  2013     1     1     6      0 2013-01-01 06:00:00
#> 6  2013     1     1     5     58 2013-01-01 05:58:00
#> # ... with 3.368e+05 more rows
```

Сделаем то же самое для каждого из четырех столбцов таблицы `flights`, связанных со временем. В этой таблице время представлено в несколько необычном формате, поэтому для извлечения часов и минут мы обратимся к модульной арифметике. Создав переменные даты/времени, мы сфокусируемся на переменных, которые будем исследовать в остальной части главы.

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights %>%
  filter(!is.na(dep_time), !is.na(arr_time)) %>%
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(
      year, month, day, sched_dep_time
    ),
    sched_arr_time = make_datetime_100(
      year, month, day, sched_arr_time
    )
  ) %>%
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt
#> # A tibble: 328,063 × 9
#>   origin  dest dep_delay arr_delay            dep_time
#>    <chr> <chr>     <dbl>     <dbl>              <dttm>
#> 1    EWR   IAH         2        11 2013-01-01 05:17:00
#> 2    LGA   IAH         4        20 2013-01-01 05:33:00
#> 3    JFK   MIA         2        33 2013-01-01 05:42:00
#> 4    JFK   BQN        -1       -18 2013-01-01 05:44:00
#> 5    LGA   ATL        -6       -25 2013-01-01 05:54:00
#> 6    EWR   ORD        -4        12 2013-01-01 05:54:00
#> # ... with 3.281e+05 more rows, and 4 more variables:
```

```
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

Используя эти данные, мы можем визуализировать распределение времени вылета на протяжении года (рис. 13.1).

```
flights_dt %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 секунд = 1 день
```

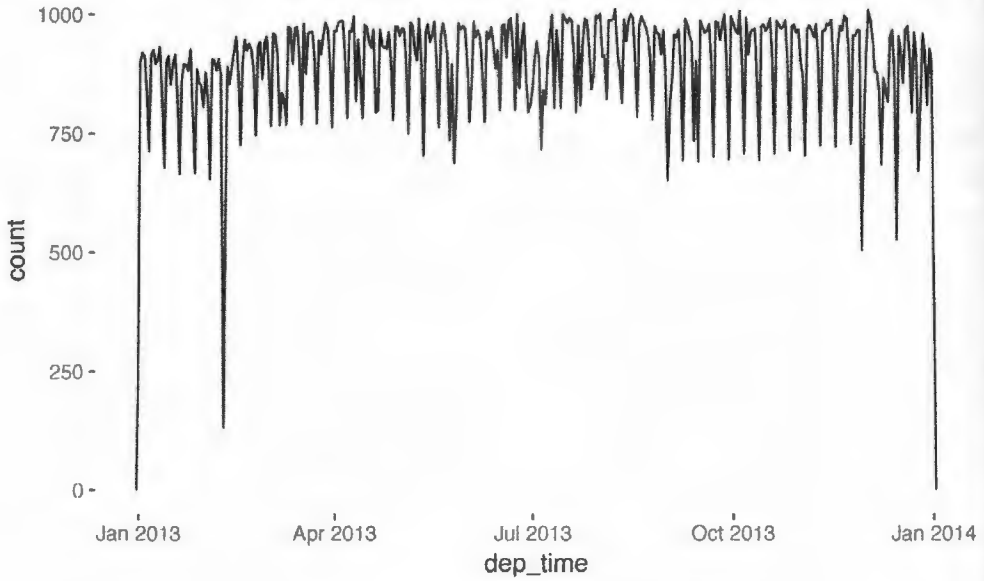


Рис. 13.1

Или на протяжении одного дня (рис. 13.2).

```
flights_dt %>%
  filter(dep_time < ymd(20130102)) %>%
  ggplot(aes(dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 секунд = 10 минут
```

Учтите, что при использовании значений даты/времени в числовом контексте (как в случае гистограммы) 1 означает 1 секунду, поэтому значению параметра `binwidth`, равному 86400, соответствует один день. Для дат значению 1 соответствует 1 день.

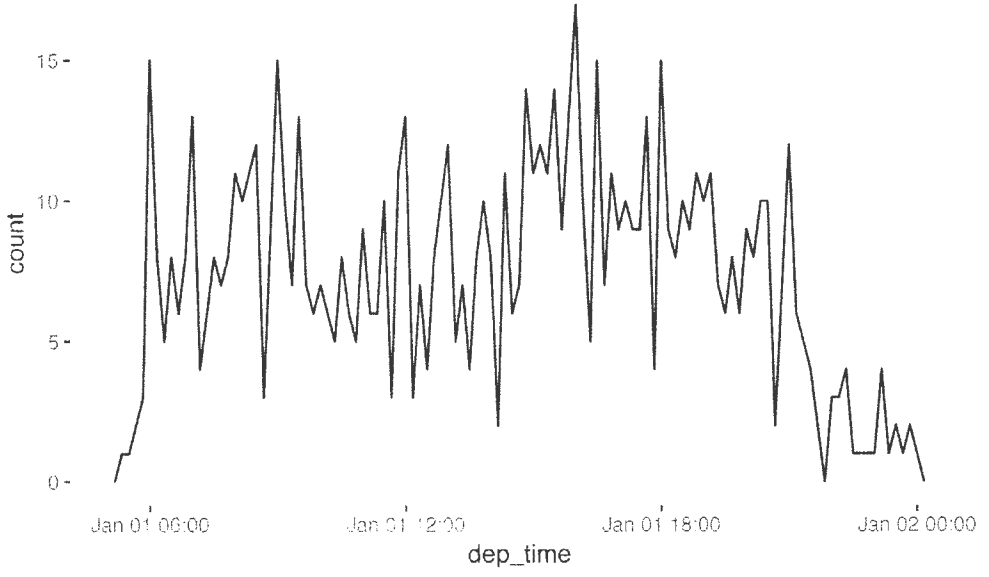


Рис. 13.2

Создание значений даты/времени на основе других типов

У вас может возникнуть необходимость в выполнении переходов между значениями даты/времени и даты. Это работа функций `as_datetime()` и `as_date()`.

```
as_datetime(today())
=> [1] "2016-10-10 UTC"
as_date(now())
=> [1] "2016-10-10"
```

Иногда ваши исходные значения даты и времени будут заданы в виде числового смещения от *эпохи Unix* (1970-01-01). Если это смещение задано в секундах, используйте функцию `as_datetime()`; если оно задано в днях, используйте функцию `as_date()`.

```
as_datetime(60 * 60 * 10)
=> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
=> [1] "1980-01-01"
```

Упражнения

1. Что произойдет при синтаксическом разборе строки, содержащей недопустимые значения даты?

```
ymd(c("2010-10-10", "bananas"))
```

2. Каково назначение аргумента `tzone` функции `today()`? Почему он важен?
3. Используйте подходящую функцию пакета **lubridate** для разбора каждой из приведенных ниже дат.

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # 30 декабря 2014 года
```

Компоненты даты/времени

Теперь, когда вы уже знаете, как включить данные о дате и времени в соответствующие структуры данных в R, рассмотрим, что можно делать с ними. В этом разделе наше внимание будет сосредоточено на функциях доступа, позволяющих получать и устанавливать значения отдельных компонентов. В следующем разделе мы обсудим выполнение арифметических операций по отношению к значениям даты/времени.

Получение компонентов

Для извлечения отдельных компонентов даты и времени предназначены функции `year()`, `month()`, `mday()` (день месяца), `yday()` (день года), `wday()` (день недели), `hour()`, `minute()` и `second()`.

```
datetime <- ymd_hms("2016-07-08 12:34:56")

year(datetime)
#> [1] 2016
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

yday(datetime)
#> [1] 190
wday(datetime)
#> [1] 6
```

Передав функциям `month()` и `wday()` аргумент `label = TRUE`, можно получить сокращенное название месяца или дня недели. Чтобы получить полное название, задайте аргумент `abbr = FALSE`.

```
month(datetime, label = TRUE)
#> [1] Jul
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < ... < Dec
wday(datetime, label = TRUE, abbr = FALSE)
#> [1] Friday
#> 7 Levels: Sunday < Monday < Tuesday < ... < Saturday
```

Мы можем использовать функцию `wday()` для того, чтобы убедиться в том, что количество вылетов на протяжении недели больше, чем в выходные дни (рис. 13.3).

```
flights_dt %>%
  mutate(wday = wday(dep_time, label = TRUE)) %>%
  ggplot(aes(x = wday)) +
    geom_bar()
```

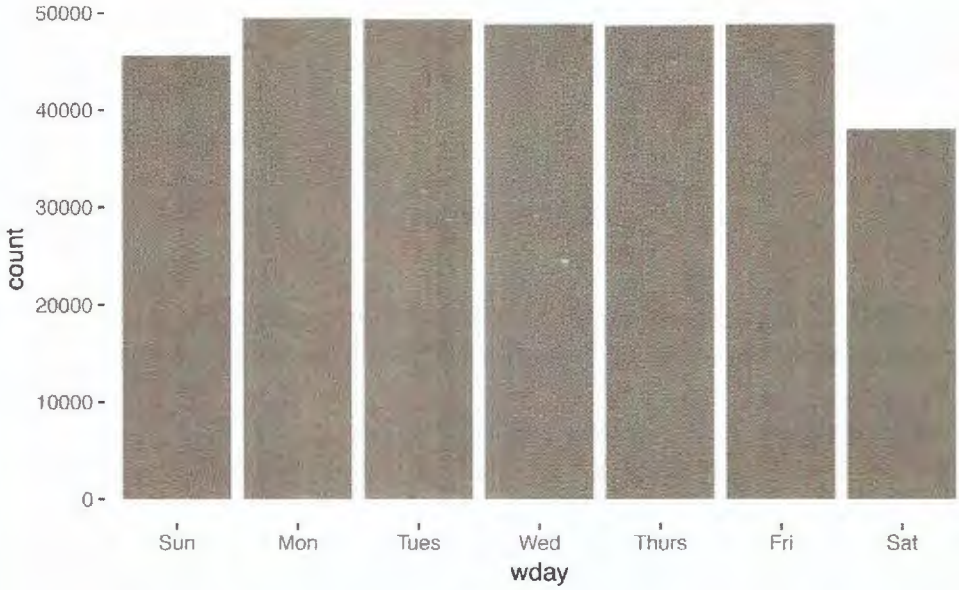


Рис. 13.3

В поведении среднего времени задержки вылета в зависимости от минуты часа наблюдается любопытная закономерность (рис. 13.4). Похоже, что для авиарейсов, вылетающих в минутные интервалы часа 20–30 и 50–60, задержка намного меньше, чем для остальных временных промежутков!

```
flights_dt %>%
  mutate(minute = minute(dep_time)) %>%
  group_by(minute) %>%
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n()) %>%
  ggplot(aes(minute, avg_delay)) +
    geom_line()
```

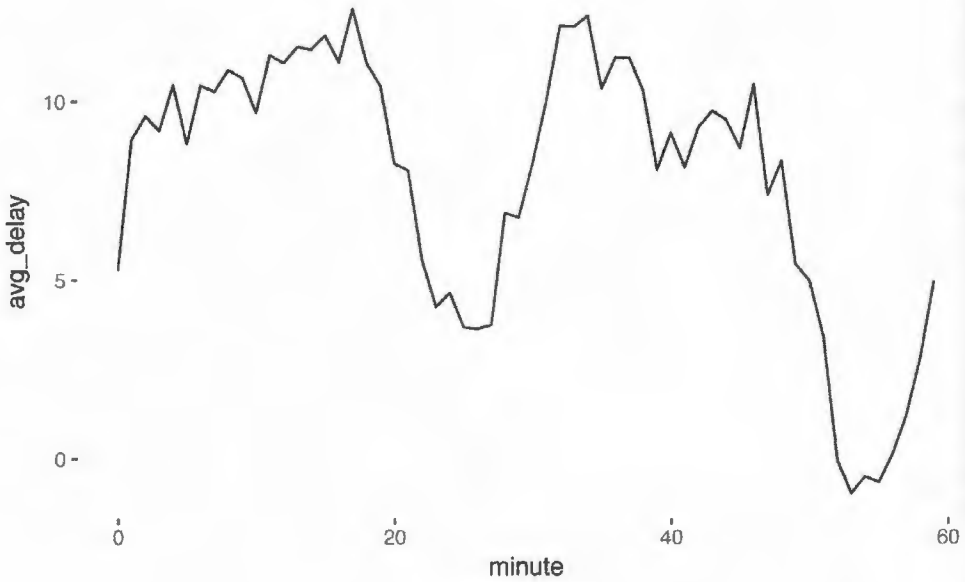


Рис. 13.4

Интересно отметить, что если взглянуть на время вылета по расписанию, то никакой ярко выраженной закономерности не наблюдается (рис. 13.5).

```
sched_dep <- flights_dt %>%
  mutate(minute = minute(sched_dep_time)) %>%
  group_by(minute) %>%
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n())

ggplot(sched_dep, aes(minute, avg_delay)) +
  geom_line()
```

Так почему же нам удается наблюдать закономерность в фактическом времени вылета? Подобно многим другим данным, подготовленным людьми, здесь наблюдается отчетливое смещение в сторону авиарейсов, отправляю-

щихся в “круглое” время (рис. 13.6). Работая с данными, подверженными влиянию человеческих суждений, всегда надо быть настороже!

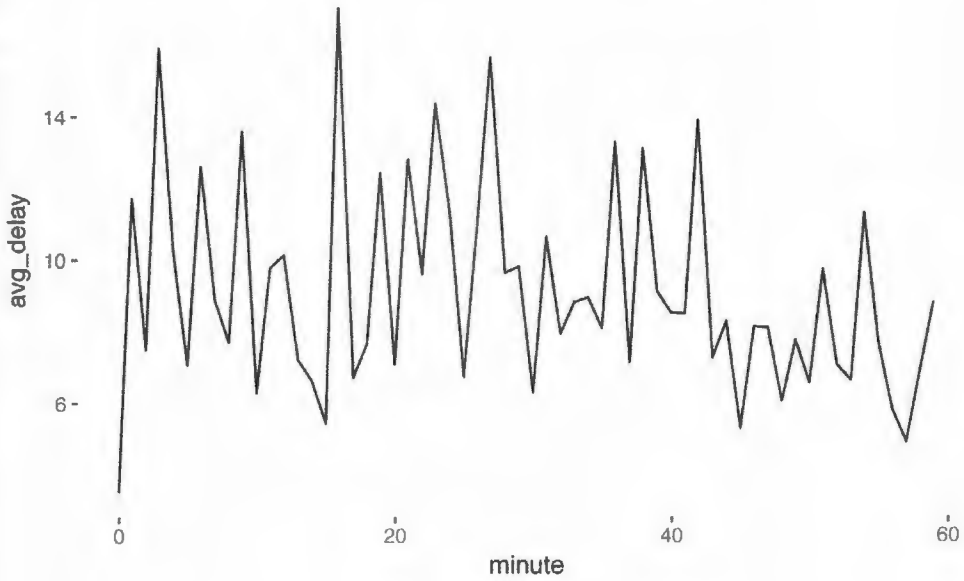


Рис. 13.5

```
ggplot(sched_dep, aes(minute, n)) +
  geom_line()
```

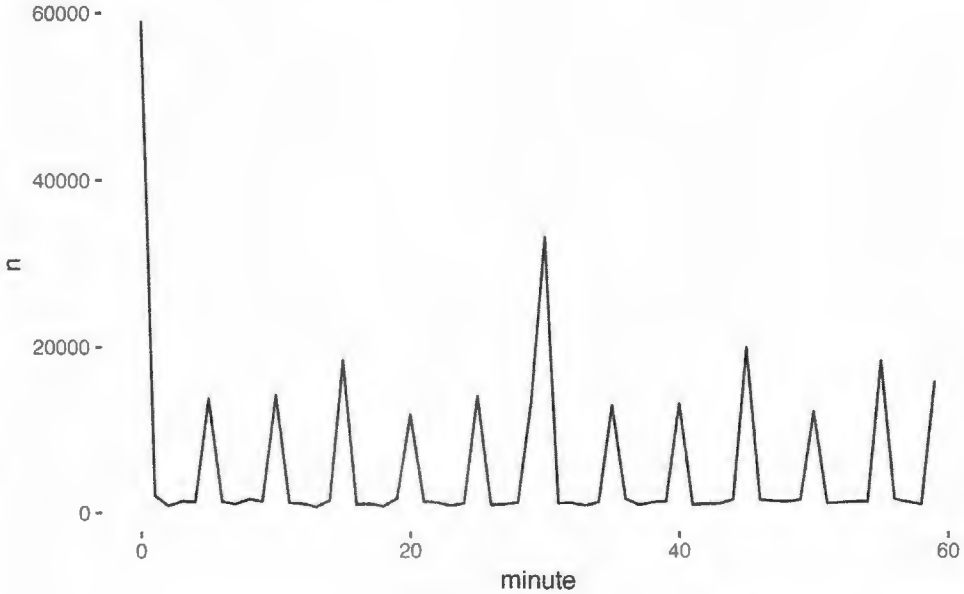


Рис. 13.6

Округление

Альтернативный подход к построению графиков отдельных компонентов — округление даты до ближайшей единицы времени с помощью функций `floor_date()`, `round_date()` и `ceiling_date()`. Каждая из этих функций принимает вектор дат, подлежащих округлению, и название единицы, до которой должно выполняться округление вниз (`floor`), округление вверх (`ceiling`) или округление до ближайшего значения (`round`). Это, например, позволяет построить график количества авиарейсов в неделю (рис. 13.7).

```
flights_dt %>%
  count(week = floor_date(dep_time, "week")) %>%
  ggplot(aes(week, n)) +
    geom_line()
```

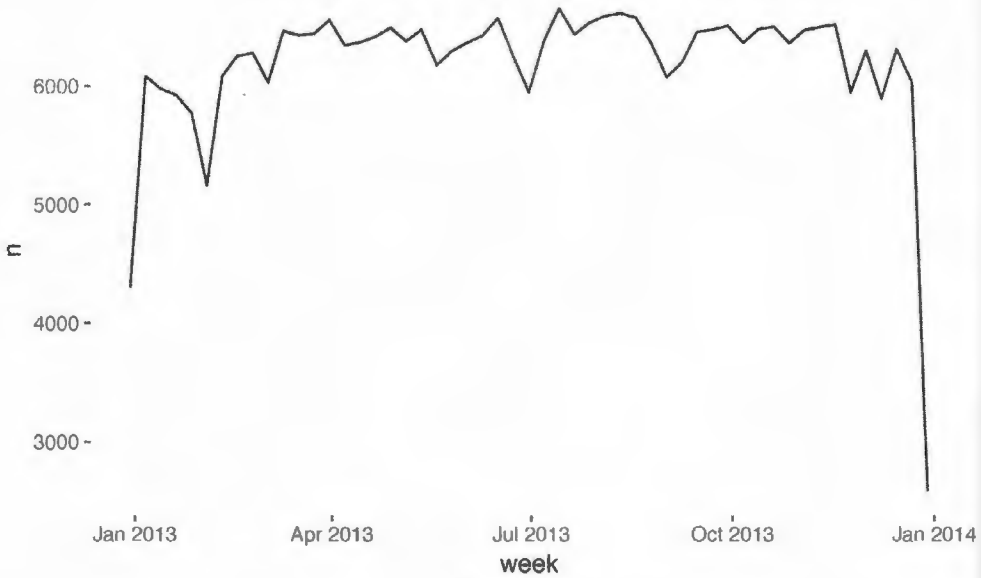


Рис. 13.7

Особенно полезным может быть вычисление разности между округленной и неокругленной датами.

Задание значений компонентов

Функции доступа также позволяют устанавливать значения компонентов даты/времени.

```
(datetime <- ymd_hms("2016-07-08 12:34:56"))
#> [1] "2016-07-08 12:34:56 UTC"

year(datetime) <- 2020
datetime
#> [1] "2020-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2020-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
```

Альтернативный вариант позволяет не изменять значения на месте, а создавать новые значения с помощью функции `update()`. Кроме того, можно устанавливать одновременно несколько значений.

```
update(datetime, year = 2020, month = 2, mday = 2, hour = 2)
#> [1] "2020-02-02 02:34:56 UTC"
```

Если значения слишком велики, они "закольцовываются".

```
ymd("2015-02-01") %>%
  update(mday = 30)
#> [1] "2015-03-02"
ymd("2015-02-01") %>%
  update(hour = 400)
#> [1] "2015-02-17 16:00:00 UTC"
```

Функцию `update()` можно использовать для того, чтобы отобразить распределение авиарейсов на протяжении дня для каждого дня в году (рис. 13.8).

```
flights_dt %>%
  mutate(dep_hour = update(dep_time, yday = 1)) %>%
  ggplot(aes(dep_hour)) +
    geom_freqpoly(binwidth = 300)
```

Задание для более крупных компонентов даты постоянного значения — удобная методика, позволяющая исследовать закономерности, связанные с меньшими компонентами.

Упражнения

1. Как распределение авиарейсов по времени на протяжении дня изменяется в течение года?
2. Сравните между собой значения переменных `dep_time`, `sched_dep_time` и `dep_delay`. Согласуются ли они между собой? Объясните обнаруженные вами результаты.

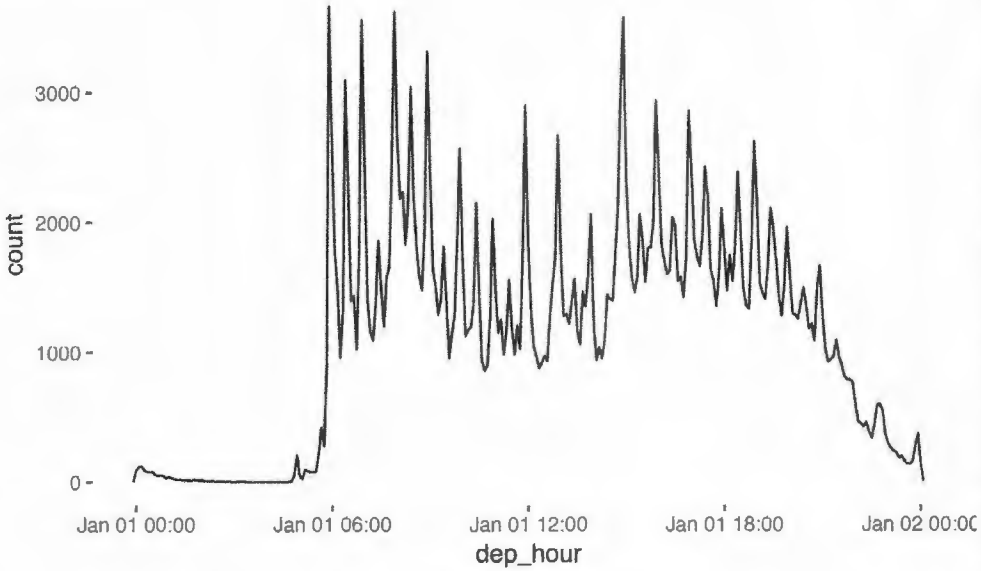


Рис. 13.8

3. Сравните значения переменной `air_time` с длительностью промежутка времени между вылетом и прилетом самолета. Объясните полученные результаты. (Подсказка: учитывайте местоположение аэропорта.)
4. Как среднее время задержки изменяется в течение дня? Что вы должны использовать: `dep_time` или `sched_dep_time`? Почему?
5. Какой день недели выбрали бы вы для вылета, чтобы минимизировать время задержки?
6. Чем обусловлено сходство распределения переменных `diamonds$carat` и `flights$sched_dep_time`?
7. Подтвердите справедливость моей гипотезы о том, что своевременная отправка авиарейсов в минутные промежутки часа 20–30 и 50–60 обусловлена своевременной отправкой запланированных авиарейсов. (Подсказка: создайте бинарную переменную, которая сообщает о том, задержался или не задержался авиарейс.)

Временные промежутки

Далее вы узнаете о том, как выполнять над датами арифметические операции, такие как вычитание, сложение и деление. Параллельно вы изучите три важных класса, определяющие временные промежутки:

- *длительности*, представляющие количество секунд;
- *периоды*, представляющие используемые людьми промежутки времени, такие как недели и месяцы;
- *интервалы*, представляющие начальный и конечный моменты времени.

Длительности

В R при вычитании двух дат вы получаете объект `difftime`.

```
# Сколько лет Хэдли?
h_age <- today() - ymd(19791014)
h_age
#> Time difference of 13511 days
```

Объект класса `difftime` записывает временн*ы*е промежутки, выраженные в секундах, минутах, часах, днях или неделях. Эта неоднозначность может несколько затруднять работу с объектами `difftime`, поэтому пакет **lubridate** предоставляет альтернативу — *длительность* (`duration`), которая всегда выражается в секундах.

```
as.duration(h_age)
#> [1] "1167350400s (~36.99 years)"
```

Для длительностей предусмотрен целый ряд удобных конструкторов.

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                 "86400s (~1 days)"
#> [3] "172800s (~2 days)" "259200s (~3 days)"
#> [5] "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31536000s (~52.14 weeks)"
```

Длительности всегда выражают временн*ы*е промежутки в секундах. Более крупные единицы создаются путем перевода минут, часов, дней, недель и лет в секунды с использованием стандартных коэффициентов (60 минут в часе, 24 часа в сутках, 7 дней в неделе и 365 дней в году).

Длительности можно складывать и умножать на числа.

```
2 * dyears(1)
#> [1] "63072000s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38847600s (~1.23 years)"
```

Длительности также могут участвовать в операциях сложения и вычитания совместно с днями.

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

Вместе с тем, поскольку длительности представляют точное количество секунд, иногда возможно получение неожиданного результата.

```
one_pm <- ymd_hms(
  "2016-03-12 13:00:00",
  tz = "America/New_York"
)

one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
```

Почему момент времени спустя 1 сутки после 13:00 12 марта получился равным 14:00 13 марта?! Если вы внимательно присмотритесь к датам, то, возможно, обратите внимание на изменение часового пояса. Ввиду перехода на летнее время 12 марта содержит лишь 23 часа, но после добавления полных суток, выраженных в секундах, мы получаем другое время.

Периоды

Для разрешения этой проблемы пакет **lubridate** предоставляет *периоды*. Это временны*е* промежутки, но они не работают с фиксированной длительностью, выраженной в секундах, а используют такие единицы, используемые людьми, как дни и месяцы, благодаря чему работают более интуитивным образом.

```
one_pm
#> [1] "2016-03-12 13:00:00 EST"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Подобно длительностям, периоды можно создавать с помощью целого ряда родственных функций-конструкторов.

```
seconds(15)
#> [1] "15S"
minutes(10)
#> [1] "10M 0S"
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S"
#> [4] "4m 0d 0H 0M 0S" "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
weeks(3)
#> [1] "21d 0H 0M 0S"
years(1)
#> [1] "1y 0m 0d 0H 0M 0S"
```

Периоды можно складывать и умножать на числа.

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

Разумеется, периоды также можно складывать с датами. В отличие от длительностей, работая с периодами, вы будете получать результаты в соответствии со своими ожиданиями.

```
# Високосный год
ymd("2016-01-01") + dyears(1)
#> [1] "2016-12-31"
ymd("2016-01-01") + years(1)
#> [1] "2017-01-01"

# Переход на летнее время
one_pm + ddays(1)
#> [1] "2016-03-13 14:00:00 EDT"
one_pm + days(1)
#> [1] "2016-03-13 13:00:00 EDT"
```

Давайте используем периоды для устранения кажущихся странностей, связанных с датами авиарейсов. Некоторые самолеты, если верить данным, прилетают в пункт назначения еще до того, как вылетели из Нью-Йорка!

```
flights_dt %>%
  filter(arr_time < dep_time)
#> # A tibble: 10,633 × 9
#>   origin  dest dep_delay arr_delay            dep_time
#>    <chr> <chr>     <dbl>     <dbl>              <dttm>
```

```
#> 1    EWR   BQN         9        -4 2013-01-01 19:29:00
#> 2    JFK   DFW        59        NA 2013-01-01 19:39:00
#> 3    EWR   TPA        -2         9 2013-01-01 20:58:00
#> 4    EWR   SJU        -6       -12 2013-01-01 21:02:00
#> 5    EWR   SFO        11       -14 2013-01-01 21:08:00
#> 6    LGA   FLL       -10        -2 2013-01-01 21:20:00
#> # ... with 1.063e+04 more rows, and 4 more variables:
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

Это авиарейсы, в которых при переходе через полночь самолет находится в воздухе. Мы использовали одну и ту же информацию о дате как для времени вылета, так и для времени прилета, но эти ночные авиарейсы прибывают в аэропорт назначения на следующий день. Мы можем это учесть, добавив `days(1)` ко времени прибытия каждого ночного авиарейса.

```
flights_dt <- flights_dt %>%
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight * 1),
    sched_arr_time = sched_arr_time + days(overnight * 1)
)
```

Теперь все авиарейсы подчиняются законам физики.

```
flights_dt %>%
  filter(overnight, arr_time < dep_time)
#> # A tibble: 0 × 10
#> # ... with 10 variables: origin <chr>, dest <chr>,
#> #   dep_delay <dbl>, arr_delay <dbl>, dep_time <dttm>,
#> #   sched_dep_time <dttm>, arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>, overnight <lgl>
```

Интервалы

Результат деления `dyears(1) / ddays(365)` очевиден — 1, поскольку длительности всегда представляются количеством секунд, а длительность года определяется как количество секунд в 365 днях.

А какой результат даст деление `years(1) / days(1)`? Если имеется в виду 2015 год, то результат будет равен 365, а если 2016 год — 366!

В данном случае пакет **lubridate** не располагает достаточной информацией для того, чтобы дать однозначный ответ. Вместо этого он даст лишь оценку результата, сопроводив ее предупреждающим сообщением.

```
years(1) / days(1)
#> estimate only: convert to intervals for accuracy
#> [1] 365
```

Для получения точного ответа следует использовать *интервал*. Интервал — это длительность, для которой указан начальный момент времени, что позволяет точно определить, сколько времени она занимает.

```
next_year <- today() + years(1)
(today() %--% next_year) / ddays(1)
#> [1] 365
```

Чтобы выяснить, сколько периодов приходится на интервал, необходимо использовать целочисленное деление.

```
(today() %--% next_year) %/% days(1)
#> [1] 365
```

Резюме

Каковы критерии выбора между длительностями, периодами и интервалами? Как обычно, всегда выбирайте самую простую структуру, которая позволяет решить вашу задачу. Если вас интересует лишь физическое время, используйте длительность; если вам необходимо выразить время в терминах, используемых людьми, используйте период; если вам необходимо выразить длительность временного промежутка в терминах, используемых людьми, используйте интервал.

На рис. 13.9 приведена сводка арифметических операций, разрешенных для различных типов даты.

	date				date time				duration				period				interval				number			
date	-								-	+			-	+							-	+		
date time					-				-	+			-	+							-	+		
duration	-	+			-	+			-	+		/									-	+	×	/
period	-	+			-	+							-	+							-	+	×	/
interval			-									/				/								
number	-	+			-	+			-	+	×		-	+	×		-	+	×		-	+	×	/

Рис. 13.9

Упражнения

1. Почему есть функция `months()`, но нет функции `dmonths()`?
2. Как объяснить смысл вызова функции `days(overnight * 1)` человеку, только приступившему к изучению R? Как это работает?
3. Создайте вектор дат, представляющий первый день каждого месяца 2015 года. Создайте вектор дат, представляющий первый день каждого месяца текущего года.
4. Напишите функцию, которая, получив данные о вашем дне рождения (в виде даты), возвратит ваш возраст в годах.
5. Почему не сработает следующее выражение?

```
(today() %--% (today() + years(1)) /months(1)
```

Часовые пояса

Часовые пояса — невероятно сложная тема из-за их зависимости от ряда геополитических факторов. К счастью, нам нет нужды углубляться во все детали, поскольку не все они важны для анализа данных, однако есть некоторые трудности, с которыми вы должны научиться справляться.

Первая из них — это то, что обычно используемые названия часовых поясов страдают неоднозначностью. Например, если вы американец, то, вероятно, знаете, что такое EST (Eastern Standard Time — стандартное восточное время). Но в Австралии и Канаде используются свои варианты EST! Во избежание путаницы в R используется международный стандарт часовых поясов IANA. В нем применяется согласованная схема именования поясов с символом "/" в качестве разделителя, обычно в формате "*<континент>/<город>*" (есть некоторые исключения, поскольку не все страны располагаются на континенте). Вот некоторые примеры: "America/New_York", "Europe/Paris" и "Pacific/Auckland".

Возможно, вас удивляет, что часовой пояс указывается с использованием названия города, тогда как обычно часовые пояса связывают со странами или регионами в пределах страны. Это обусловлено тем, что в базе данных IANA хранится история смены правил определения часовых поясов. На протяжении десятилетий страны довольно часто меняют свои названия (или распадаются), но названия городов, как правило, не меняются. Другая проблема заключается в том, что названия поясов должны отражать не только

их текущее поведение, но и всю их историю. Например, существуют часовые пояса как для "America/New_York", так и для "America/Detroit". В настоящее время оба города относятся к часовому поясу EST, но в 1969–1972 годах Мичиган (штат, в котором находится Детройт) не переходил на летнее время, и для этого периода времени требуется другое название. Вам стоит просмотреть исходную базу данных часовых поясов (доступную по адресу `http://www.iana.org/time-zones`), где вы сможете прочитать о нескольких историях подобного рода.

Вы сможете выяснить, что известно R о вашем текущем часовом поясе, вызвав функцию `Sys.timezone()`.

```
Sys.timezone()
#> [1] "America/Los_Angeles"
```

(Если R ничего об этом не известно, вы получите `NA` в качестве ответа.)

А если вы хотите ознакомиться с полным списком названий часовых поясов, то воспользуйтесь функцией `OlsonNames()`.

```
length(OlsonNames())
#> [1] 589
head(OlsonNames())
#> [1] "Africa/Abidjan"      "Africa/Accra"
#> [3] "Africa/Addis_Ababa"  "Africa/Algiers"
#> [5] "Africa/Asmara"       "Africa/Asmera"
```

В R часовой пояс — это атрибут даты/времени, который лишь управляет выводом на печать. Например, следующие три объекта представляют один и тот же момент времени.

```
(x1 <- ymd_hms("2015-06-01 12:00:00", tz = "America/New_York"))
#> [1] "2015-06-01 12:00:00 EDT"
(x2 <- ymd_hms("2015-06-01 18:00:00", tz = "Europe/Copenhagen"))
#> [1] "2015-06-01 18:00:00 CEST"
(x3 <- ymd_hms("2015-06-02 04:00:00", tz = "Pacific/Auckland"))
#> [1] "2015-06-02 04:00:00 NZST"
```

Можете проверить это путем вычитания.

```
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

Если не указано иное, пакет **lubridate** всегда использует UTC (Coordinated Universal Time — всемирное координированное время). Это стандартный часовой пояс, используемый научным сообществом, который примерно

эквивалентен своему предшественнику GMT (Greenwich Mean Time — среднее время по Гринвичу). В нем отсутствует переход на летнее время, что удобно для выполнения вычислений. В операциях, объединяющих дату и время, наподобие `c()`, часовой пояс часто опускается. В этом случае дата и время будут отображаться в соответствии с вашим текущим часовым поясом.

```
x4 <- c(x1, x2, x3)
x4
#> [1] „2015-06-01 09:00:00 PDT" „2015-06-01 09:00:00 PDT"
#> [3] „2015-06-01 09:00:00 PDT"
```

Часовой пояс можно изменить двумя способами.

- Можно оставить то же время, но изменить его отображение. Используйте этот метод, если хотите отобразить время в более естественном формате.

  ```
  x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
  x4a
  #> [1] "2015-06-02 02:30:00 LHST"
  #> [2] "2015-06-02 02:30:00 LHST"
  #> [3] "2015-06-02 02:30:00 LHST"
  x4a - x4
  #> Time differences in secs
  #> [1] 0 0 0
  ```

  (Этот пример также иллюстрирует суть еще одной проблемы часовых поясов: разница между ними не всегда выражается целым числом часов!)

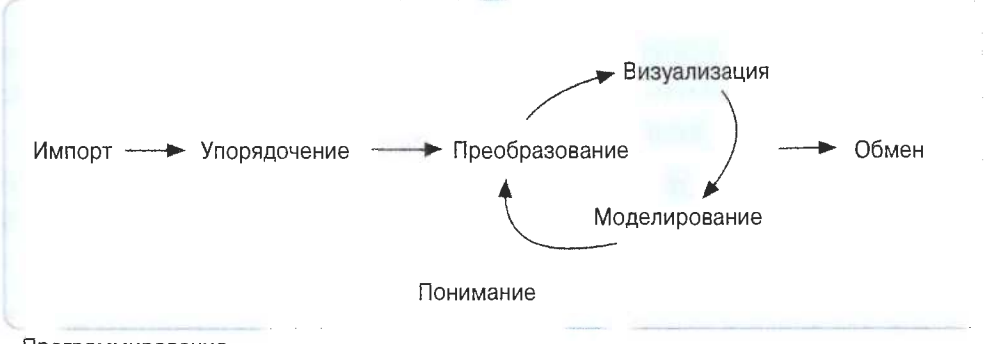
- Можно изменить само время. Используйте данный метод, если время было выведено с некорректно указанным часовым поясом и вы хотите исправить это.

  ```
  x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
  x4b
  #> [1] "2015-06-01 09:00:00 LHST"
  #> [2] "2015-06-01 09:00:00 LHST"
  #> [3] "2015-06-01 09:00:00 LHST"
  x4b - x4
  #> Time differences in hours
  #> [1] -17.5 -17.5 -17.5
  ```

ЧАСТЬ III

Программирование

В этой части вы усовершенствуете свои навыки программирования. В науке о данных программирование играет важнейшую роль: невозможно исследовать данные, выполняя необходимые вычисления в голове или пользуясь карандашом и бумагой, — для этого необходимо использовать компьютер.



Программирование

Результатом программирования является код, а код — это средство коммуникации. Разумеется, в первую очередь код необходим для того, чтобы сообщить компьютеру, что он должен сделать. Но кроме этого он сообщает о смысле выполняемых действий другим людям. Очень важно, чтобы вы относились к коду как к средству коммуникации, поскольку каждый проект, выполняемый вами, по сути является коллективной работой. Даже если вы не сотрудничаете с другими людьми, вы несомненно сотрудничаете со своим будущим "я"! Ясность кода — ключевая предпосылка того, что другие люди (в том числе и вы сами впоследствии) смогут понять, почему вы анализировали данные именно так, а не иначе. Отсюда следует, что развитие навыков программирования одновременно развивает ваши навыки общения. Со временем вам не только будет легче писать код — его будет проще читать другим людям.

Написание кода во многом напоминает написание прозы. Одна из параллелей, которую мы считаем наиболее плодотворной, — это то, что в обоих

случаях не нужно лениться переписывать текст. Маловероятно, чтобы ваши идеи были ясно изложены уже с первой попытки, и вам потребуется многократно переписывать текст программы. После того как задача анализа данных решена, часто полезно вернуться к коду и посмотреть, достаточно ли очевидно в нем изложены ваши намерения. Посвятив немного времени переписыванию кода, пока идеи еще свежи в голове, вы впоследствии сэкономите массу времени, которое в противном случае пришлось бы потратить лишь на то, чтобы вспомнить, как именно работает код. Это вовсе не означает, что переписывания требует каждая функция: вы должны стремиться к достижению разумного компромисса между тем, чего вы хотите добиться сейчас, и возможностью сэкономить время в долгосрочной перспективе. (Однако чем больше функций вы перепишете, тем вероятнее, что ваш код приобретет бо́льшую ясность.)

В следующих четырех главах вы приобретете навыки, с помощью которых вам будет не только проще писать новые программы с понятным кодом, но и легче справляться с существующими задачами.

- В главе 14 мы займемся углубленным изучением возможностей *канала*, `%>%`, того, как он работает, какие существуют альтернативы и как не следует его использовать.
- Копирование и вставка кода — удобный прием, но следует избегать делать это дважды. Подобное дублирование фрагментов опасно тем, что может легко привести к ошибкам и несогласованности кода. В главе 15 будет показано, что вместо этого лучше писать *функции*, которые позволяют обращаться к одному и тому же фрагменту кода столько раз, сколько нужно, обеспечивая простоту его многократного использования.
- Написание более мощных функций требует уверенного владения *структурами данных* R, чему посвящена глава 16. Вы изучите четыре часто используемых атомарных вектора и три построенных поверх них важных класса S3 и узнаете обо всех тонкостях работы со списками и фреймами данных.
- Функции содержат повторяющийся код, но часто требуется повторение одних и тех же действий по отношению к различным входным данным. Для этого вам потребуются средства *итерирования*, позволяющие вновь и вновь выполнять однотипные действия. К числу этих инструментов

относятся циклы `for` и функциональное программирование, о которых вы прочитаете в главе 17.

Узнайте больше

Цель перечисленных выше глав — помочь вам приобрести тот минимум навыков программирования, который потребуется вам для осуществления профессиональной деятельности в области науки о данных и объема которого на первых порах будет вполне достаточно. Мы настоятельно рекомендуем, чтобы после проработки материала, изложенного в этой книге, вы приложили дополнительные усилия к овладению программированием. Развитие умения программировать — это долгосрочная инвестиция: она не окупится немедленно, но в перспективе это позволит вам быстрее решать вновь возникающие задачи и повторно использовать накопленный опыт в новых сценариях.

Для расширения своих знаний вам потребуется изучить R как язык программирования, а не только как интерактивную среду, обеспечивающую анализ данных. Мы написали две книги, которые помогут вам в этом.

- *Hands-On Programming with R* (Гарретт Гроулмунд). Это введение в язык программирования R, которое отлично подойдет для начала, если R — ваш первый язык программирования. Она охватывает тот же материал, что и эти главы, но написана в другом стиле и с другими мотивирующими примерами (в центре которых находится работа казино). Книга послужит вам хорошим дополнением, если вы сочтете, что в четырех перечисленных главах материал изложен слишком бегло.
- *Advanced R* (Хэдли Уикем). Эта книга посвящена подробному рассмотрению языка программирования R. С нее неплохо начать, если у вас уже есть опыт программирования. Она также отлично подойдет вам в качестве следующего шага после освоения материала этих четырех глав. Кроме того, можете воспользоваться электронным вариантом книги, доступным в Интернете на сайте `http://adv-r.had.co.nz`.

ГЛАВА 14

Работа с каналами с помощью пакета magrittr

Введение

Каналы — мощный инструмент, позволяющий ясно выражать последовательность выполнения нескольких операций. До этого времени вы использовали их, не зная, как они работают и какие возможны альтернативы. Эта глава посвящена более подробному рассмотрению каналов. Вы познакомитесь с альтернативами каналам, узнаете о том, когда не следует использовать каналы, и изучите ряд полезных родственных инструментов.

Необходимые ресурсы

Канал, `%>%`, происходит из пакета **magrittr**, разработанного Стефаном Мильтоном Башем. Пакеты библиотеки *tidyverse* загружают канал `%>%` автоматически, так что обычно это не приходится делать самостоятельно. Однако в данном случае мы фокусируем внимание на работе с каналами и не загружаем другие пакеты, в связи с чем нам понадобится явная загрузка этого пакета.

```
library(magrittr)
```

Альтернативы каналам

Смысл использования каналов заключается в том, чтобы помочь вам писать код, который легко читать и понимать. Чтобы продемонстрировать, в чем именно состоит польза каналов, мы исследуем ряд способов написания

одного и того же кода. В качестве примера используем код, излагающий историю о маленьком кролике, которого звали Foo Foo.

Little bunny Foo Foo
Went hopping through the forest
Scooping up the field mice
And bopping them on the head

Это текст популярной детской песенки, исполнение которой сопровождается движениями рук.

Начнем с определения объекта, представляющего кролика Foo Foo:

```
foo_foo <- little_bunny()
```

Для каждого ключевого глагола используем соответствующую функцию: `hop()` (скакать), `scoop()` (ловить) и `bop()` (стучать). Используя этот объект и глаголы, можно предложить (по крайней мере) четыре способа изложения этой истории в виде кода:

- сохранение каждого промежуточного действия в новом объекте;
- многократное изменение исходного объекта;
- композиция функций;
- использование канала.

Мы проработаем каждый из этих подходов, демонстрируя соответствующий код и обсуждая его преимущества и недостатки.

Промежуточные объекты

Простейший подход состоит в том, чтобы сохранить каждое действие в виде нового объекта.

```
foo_foo_1 <- hop(foo_foo, through = forest)
foo_foo_2 <- scoop(foo_foo_1, up = field_mice)
foo_foo_3 <- bop(foo_foo_2, on = head)
```

Основной недостаток этого способа записи кода заключается в том, что он заставляет именовать каждый промежуточный элемент. Если для этого подходят естественные имена, то это неплохая идея, и вы должны следовать ей. Но во многих случаях, как в нашем примере, естественные имена отсутствуют, и для того, чтобы сделать имена уникальными, мы добавляем числовые суффиксы. Это порождает две проблемы:

- код засоряется несущественными именами;
- необходимо тщательно следить за приращением суффикса в каждой строке.

Всякий раз, когда приходится писать код наподобие этого, неизменно где-то пропускаешь неправильный номер, а потом сидишь и минут десять ломаешь голову над тем, почему не работает код.

Вас также может беспокоить тот факт, что при такой записи кода создается несколько копий данных, на что тратится дополнительная память. К счастью, это не так. Во-первых, не стоит заранее беспокоиться по поводу расхода памяти: начинайте беспокоиться только тогда, когда это действительно становится проблемой (т.е. когда вы сталкиваетесь с нехваткой памяти). Во-вторых, среда R вовсе не лишена интеллекта и организует совместное использование столбцов фреймами данных, когда это возможно. Рассмотрим канал манипулирования фактическими данными, в котором мы добавляем новый столбец во фрейм данных `ggplot2::diamonds`.

```
diamonds <- ggplot2::diamonds
diamonds2 <- diamonds %>%
  dplyr::mutate(price_per_carat = price / carat)

pryr::object_size(diamonds)
#> 3.46 MB
pryr::object_size(diamonds2)
#> 3.89 MB
pryr::object_size(diamonds, diamonds2)
#> 3.89 MB
```

Функция `pryr::object_size()` возвращает объем памяти, занимаемой всеми ее аргументами. Поначалу может показаться, что результаты противоречат логике:

- `diamonds` занимает 3,46 Мбайт;
- `diamonds2` занимает 3,89 Мбайт;
- `diamonds` и `diamonds2` вместе занимают 3,89 Мбайт!

Почему так получается? Фрейм данных `diamonds2` имеет 10 общих столбцов с фреймом `diamonds`: необходимость дублирования всех данных отсутствует, поэтому оба фрейма имеют общие переменные. Эти переменные будут копироваться только в том случае, если вы измените одну из них. В следующем примере мы изменяем одно значение в `diamonds$carat`. Это означает,

что переменная carat уже не может использоваться одновременно двумя фреймами данных, а потому должна быть создана ее копия. Размер каждого фрейма данных остался неизменным, но их совокупный размер увеличился.

```
diamonds$carat[1] <- NA
pryr::object_size(diamonds)
#> 3.46 MB
pryr::object_size(diamonds2)
#> 3.89 MB
pryr::object_size(diamonds, diamonds2)
#> 4.32 MB
```

(Обратите внимание на то, что здесь мы используем функцию pryr::object_size(), а не встроенную функцию object.size(). Последняя принимает только один объект, что не позволяет вычислить объем памяти, разделяемой несколькими объектами.)

Замена исходного объекта

Вместо того чтобы создавать промежуточные объекты для каждого действия, мы можем изменять исходный объект.

```
foo_foo <- hop(foo_foo, through = forest)
foo_foo <- scoop(foo_foo, up = field_mice)
foo_foo <- bop(foo_foo, on = head)
```

Такой код требует меньшего объема ввода (и размышлений), что снижает вероятность опечаток. Однако и с этим подходом связаны две проблемы:

- осложняется отладка (если вы допустите ошибку, то вам придется заново выполнять весь канал с самого начала);
- повторение преобразуемого объекта (имя foo_foo повторяется шесть раз!) затрудняет понимание смысла изменений в каждой строке.

Композиция функций

Другой подход заключается в том, чтобы вообще избавиться от присваиваний и обойтись всего одной строкой объединенных вызовов функций.

```
bop(
  scoop(
    hop(foo_foo, through = forest),
    up = field_mice
  ),
  on = head
)
```

Недостатком этого подхода является то, что код приходится читать снизу вверх и слева направо, а аргументы и соответствующие им функции оказываются разнесенными. В общем, работать с таким кодом человеку очень нелегко.

Использование канала

Наконец, мы можем использовать канал.

```
foo_foo %>%
  hop(through = forest) %>%
  scoop(up = field_mouse) %>%
  bop(on = head)
```

Это наш излюбленный способ записи, поскольку он фокусирует внимание на глаголах, а не на существительных. Эту последовательность можно читать как набор императивных действий. Кролик Foo Foo прыгает (hop), затем ловит (scoop), затем стучит (bop). Разумеется, недостатком этого подхода является то, что он требует знакомства с каналами. Если вы до этого никогда не сталкивались с оператором %>%, то не будете иметь ни малейшего представления о том, что делает такой код. К счастью, большинство людей схватывают идею канала очень быстро, так что если вы поделитесь своим кодом с людьми, незнакомыми с понятием канала, то сможете очень быстро объяснить им, что к чему.

Канал работает, выполняя "лексическое преобразование": **magrittr** за кулисами переассемблирует код в канале, преобразуя его в форму, которая работает путем замены промежуточного объекта. Когда вы выполняете канал наподобие предыдущего, **magrittr** делает примерно следующее.

```
my_pipe <- function (.) {
  . <- hop(., through = forest)
  . <- scoop(., up = field_mice)
  bop(., on = head)
}

my_pipe(foo_foo)
```

Отсюда следует, что канал не работает для следующих двух классов функций.

- **Функции**, использующие текущее окружение. Например, функция assign() создает новую переменную с заданным именем в текущем окружении.

  ```
  assign("x", 10)
  x
  ```

```
#> [1] 10

"x" %>% assign(100)
x
#> [1] 10
```

Функция `assign()` не может работать с каналом, поскольку новая переменная создается во временном окружении, используемом каналом `%>%`. Если вы все же хотите использовать ее совместно с каналом, необходимо явно указать окружение.

```
env <- environment()
"x" %>% assign(100, envir = env)
x
#> [1] 100
```

К числу других функций, на которые распространяется вышесказанное, относятся `get()` и `load()`.

- Функции, использующие "ленивые" вычисления. В R аргументы функций вычисляются лишь тогда, когда в этом возникает необходимость, а не до вызова функции. Канал вычисляет каждый элемент по очереди, поэтому вы не можете полагаться на такое поведение.

  В качестве примера того, где может возникнуть данная проблема, рассмотрим функцию `tryCatch()`, которая позволяет перехватывать и обрабатывать ошибки.

```
tryCatch(stop("!"), error = function(e) "Ошибка")
#> [1] "Ошибка"

stop("!") %>%
  tryCatch(error = function(e) "Ошибка")
#> Error in eval(expr, envir, enclos): !
```

  Существует довольно широкий класс функций, демонстрирующих подобное поведение, в том числе функции `try()`, `suppressMessages()` и `suppressWarnings()`, включенные в базовый комплект R.

Когда не следует использовать канал

Канал — мощный инструмент, но не он один имеется в вашем распоряжении, и не все задачи можно решить с его помощью! Наиболее целесообразно использовать каналы для переписывания сравнительно коротких линейных

последовательностей операций. Мы считаем, что вы должны привлекать другие возможности в следующих ситуациях.

- Длина канала превышает, к примеру, 10 шагов. В таком случае создайте промежуточные объекты с описательными именами. Это облегчит отладку, поскольку проверять промежуточные результаты будет проще, а код станет более понятным, так как имена переменных могут описывать их назначение.
- Имеется несколько входных или выходных значений. Если преобразуемых объектов не один, а два или более, и все они взаимосвязаны, не используйте канал.
- Вы начинаете задумываться об использовании направленного графа со сложной структурой зависимостей. Каналы в силу своей природы линейны, и выражение сложных отношений с их помощью обычно приводит к запутанному коду.

Другие инструменты, доступные в пакете magrittr

Все пакеты библиотеки *tidyverse* автоматически делают оператор `%>%` доступным для вас, поэтому, как правило, явная загрузка пакета **magrittr** не требуется. Однако пакет **magrittr** предлагает некоторые другие полезные функции, которые могут пригодиться вам в следующих случаях.

- В процессе работы с более сложными каналами иногда удобно вызывать функцию ради ее побочных эффектов. Это может быть вывод текущего объекта на печать, построение его графика или сохранение на диске. Во многих случаях такие функции ничего не возвращают, фактически закрывая канал.

  В качестве обходного пути для решения этой проблемы можно использовать Т-канал. Оператор `%T>%` работает подобно оператору `%>%`, за исключением того, что возвращает не правую часть, а левую (рис. 14.1 и 14.2).

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
#>  NULL
```

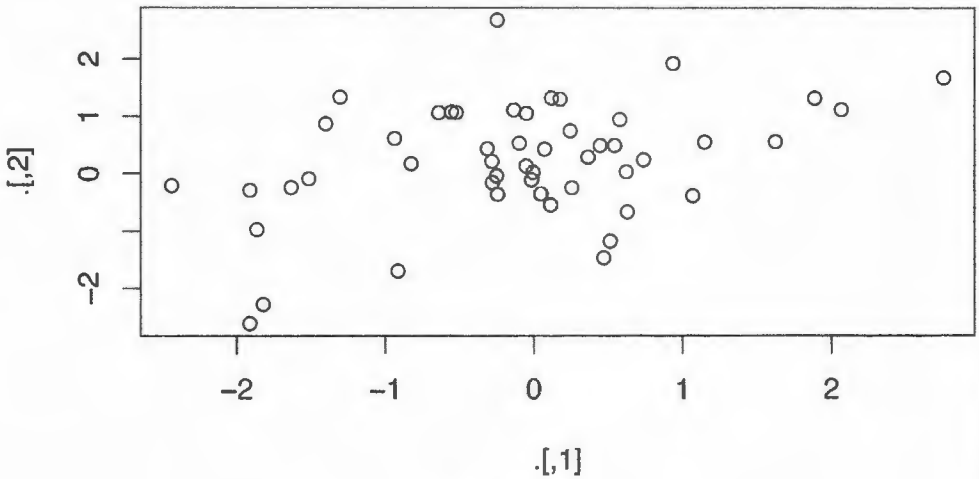


Рис. 14.1

```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
#>  num [1:50, 1:2] -0.387 -0.785 -1.057 -0.796 -1.756 ...
```

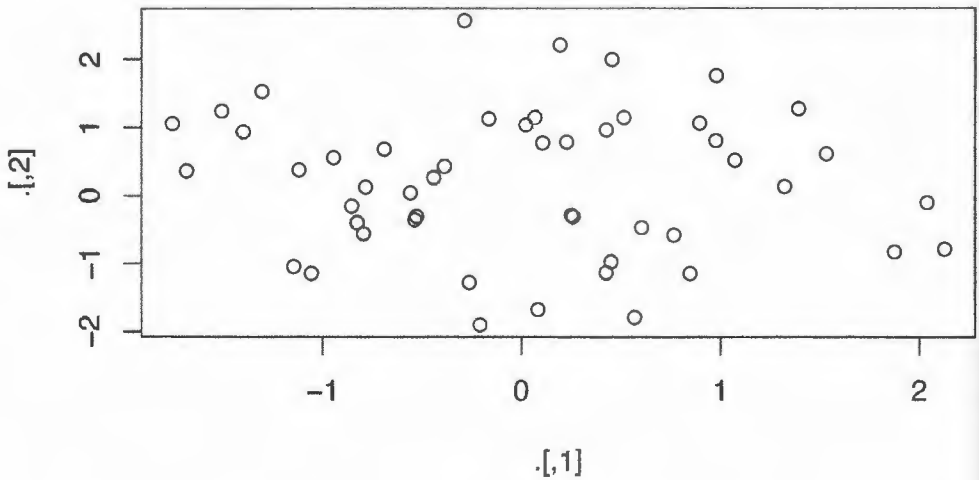


Рис. 14.2

- Если вы работаете с функциями, не имеющими программного интерфейса для работы с фреймами данных (т.е. передаете им индивидуальные векторы, а не фрейм данных и вычисляемые в его контексте выражения), вам может быть полезен оператор `%$%`. Он "вырывает

переменные из фрейма данных, чтобы на них можно было явно ссылаться. Это может пригодиться при работе со многими функциями, входящими в базовый комплект R.

```
mtcars %$%
    cor(disp, mpg)
#> [1] -0.848
```

- Для операций присваивания пакет **magrittr** предлагает оператор `%<>%`, с помощью которого, например, следующий код:

```
mtcars <- mtcars %>%
    transform(cyl = cyl * 2)
```

может быть заменен таким:

```
mtcars %<>% transform(cyl = cyl * 2)
```

Мы не являемся сторонниками использования этого оператора, поскольку считаем присваивание настолько специфической операцией, что она всегда должна быть прозрачной. По нашему мнению, незначительное дублирование кода (использование имени объекта два раза вместо одного) — вполне нормальная плата за большую ясность операции присваивания.

ГЛАВА 15

Функции

Введение

Один из лучших способов повышения вашей квалификации как специалиста в области науки о данных — написание функций. Функции позволяют автоматизировать простые задачи гораздо эффективнее, чем путем копирования и вставки кода. Написание функций обладает следующими тремя преимуществами по сравнению с методом копирования и вставки:

- функции можно присвоить запоминающееся имя, которое облегчит понимание вашего кода;
- при изменении требований вам будет достаточно изменить код не во многих местах, а только в одном;
- устраняется вероятность внесения случайных ошибок, возможных при использовании метода копирования и вставки (например, вы обновили имя переменной в одном месте кода, но забыли сделать это в других местах).

Написание хороших функций — это путешествие длиною в жизнь. Даже после использования R в течение многих лет мы все еще изучаем новые методики и более оптимальные способы решения прежних задач. Цель этой главы не в том, чтобы изложить вам все тонкости работы с функциями, а в том, чтобы дать практические советы, которые вы сможете немедленно применить.

Кроме советов по написанию функций, в этой главе также даются рекомендации, касающиеся стиля написания кода. Хороший стиль кода можно уподобить корректной пунктуации. Соблюдение определенного стиля не является обязательным требованием, но это безусловно улучшает читаемость

кода. Как и в случае с литературными стилями, в этом отношении возможны самые разные вариации. Ниже представлен стиль, который мы используем в своем коде, но самое главное — быть последовательным.

Необходимые ресурсы

В этой главе акцент сделан на написании функций с использованием базовых возможностей R, поэтому вам не потребуются дополнительные пакеты.

Когда следует писать функции

О написании функции следует задумываться всякий раз, когда приходится копировать и вставлять фрагмент кода более чем два раза (получая при этом три экземпляра одного и того же кода). Взгляните, например, на приведенный ниже код. Что он делает?

```
df <- tibble::tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df$a <- (df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$b <- (df$b - min(df$b, na.rm = TRUE)) /
  (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
df$c <- (df$c - min(df$c, na.rm = TRUE)) /
  (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
df$d <- (df$d - min(df$d, na.rm = TRUE)) /
  (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
```

Возможно, вам удалось догадаться, что этот код масштабирует значения в каждом столбце таким образом, чтобы они изменялись в пределах от 0 до 1. Но заметили ли вы ошибку? Мы допустили ее, когда копировали и вставляли код для `df$b`, забыв заменить `a` на `b`. Вынесение повторяющегося кода в функцию — неплохая идея, поскольку это избавит вас от ошибок подобного рода.

Чтобы написать функцию, нужно сначала проанализировать код. Сколько в нем имеется входных переменных?

```
(df$a - min(df$a, na.rm = TRUE)) /
  (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
```

В этом коде имеется лишь одна входная переменная: `df$a`. (Если вы удивлены тем, что `TRUE` не является входным значением, можете исследовать, почему этот так, с помощью следующего упражнения.) Чтобы входные переменные выделялись более отчетливо, имеет смысл переписать код, используя временные переменные с обычными именами. В этом коде требуется всего лишь один числовой вектор, который мы обозначим через `x`.

```
x <- df$a
(x - min(x, na.rm = TRUE)) /
(max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

Мы видим, что код дублируется: диапазон данных вычисляется три раза, хотя это имеет смысл сделать только один раз.

```
rng <- range(x, na.rm = TRUE)
(x - rng[1]) / (rng[2] - rng[1])
#>  [1] 0.289 0.751 0.000 0.678 0.853 1.000 0.172 0.611 0.612
#>  [10] 0.601
```

Вынесение промежуточных вычислений в именованные переменные — хорошая практика, поскольку это проясняет, что именно делает ваш код. Теперь, когда код упрощен и мы убедились в том, что он по-прежнему работает, можно превратить его в функцию.

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(c(0, 5, 10))
#> [1] 0.0 0.5 1.0
```

Создание новой функции включает три ключевых момента.

1. Вы должны выбрать *имя* для функции. Здесь мы используем имя `rescale01`, поскольку функция масштабирует вектор таким образом, чтобы его значения изменялись в пределах от 0 до 1.
2. Вы должны перечислить входные значения, или *аргументы* функции, после ключевого слова `function`, заключив их в круглые скобки. В данном случае имеется всего один аргумент. Если бы их было больше, вызов функции выглядел бы примерно так: `function(x, y, z)`.

3. Разработанный код помещается в *тело* функции — заключенный в фигурные скобки блок кода, следующий непосредственно за названием функции со списком аргументов.

Взгляните на процесс в целом: мы создали функцию лишь после того, как убедились в том, что она правильно работает с простыми входными данными. Проще начать с работающего кода, а затем превратить его в функцию; намного труднее сначала создать функцию и только потом пытаться заставить ее работать.

На этой стадии целесообразно проверить работу функции с различными вариантами входных данных.

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0
rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50 NA 1.00
```

По мере написания все большего количества функций вы в конечном счете захотите превратить эти неформальные интерактивные проверки в формальные автоматизированные тесты. Этот процесс называют *блочным тестированием*. К сожалению, рассмотрение данной темы выходит за рамки книги, но вы сможете узнать больше, посетив следующий сайт: `http://r-pkgs.had.co.nz/tests.html`.

Теперь, когда у нас имеется функция, мы можем упростить первоначальный пример.

```
df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

По сравнению с первоначальным вариантом этот код более понятен, и к тому же мы устранили возможность внесения ошибок в процессе копирования и вставки. Однако код все еще дублируется, поскольку мы выполняем одни и те же операции для нескольких столбцов. О том, как устранить дублирование такого рода, вы узнаете в главе 17, познакомившись предварительно со структурами данных R в главе 16.

Другим преимуществом функций является то, что при изменении требований к коду нам необходимо вносить исправления только в одном месте. Например, мы могли бы обнаружить, что некоторые из наших переменных включают бесконечные значения, приводящие к аварийному завершению функции `rescale01()`.

```
x <- c(1:10, Inf)
rescale01(x)
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

Поскольку мы вынесли код в функцию, правку нужно сделать только в одном месте.

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
rescale01(x)
#>  [1] 0.000 0.111 0.222 0.333 0.444 0.556 0.667 0.778 0.889
#>  [10] 1.000 Inf
```

Это важная составляющая принципа “не повторяйтесь” (“do not repeat yourself”, *сокр.* DRY). Чем больше повторений в коде, тем труднее будет удерживать в уме все места в нем, которые будут нуждаться в обновлении в случае необходимости (а она всегда возникает!), и тем вероятнее внесение ошибок в код с течением времени.

Упражнения

1. Почему `TRUE` не является параметром функции `rescale01()`? Что произойдет, если `x` содержит одно отсутствующее значение, а для `na.rm` установлено значение `FALSE`?
2. Во втором варианте функции `rescale01()` бесконечные значения остаются неизменными. Перепишите функцию таким образом, чтобы значение `-Inf` транслировалось в 0, а `Inf` — в 1.
3. Попрактикуйтесь в преобразовании приведенных ниже фрагментов кода в функции. Подумайте над тем, что делает каждая из этих функций. Какие имена вы присвоили бы им? В скольких аргументах они нуждаются? Можете ли вы переписать их так, чтобы они стали более выразительными или содержали меньше повторяющегося кода?

   ```
   mean(is.na(x))

   x / sum(x, na.rm = TRUE)

   sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)
   ```

4. Ознакомьтесь с рекомендациями, приведенными по адресу `http://nicercode.github.io/intro/writing-functions.html`, и напишите

существительным. Руководствуйтесь здравым смыслом и не бойтесь переименовать функцию, если впоследствии подберете для нее лучшее имя.

```
= Слишком короткое
f()

= Не глагол и не описательное
my_awesome_function()

= Длинное, но понятное
impute_missing()
collapse_years()
```

Если имя функции состоит из нескольких слов, рекомендуем использовать "змеиный_регистр", в котором слова записываются в нижнем регистре и разделяются символами подчеркивания. Популярным альтернативным вариантом является "верблюжийРегистр". В действительности не имеет значения, какой вариант вы выберете. Важно быть последовательным: остановившись на конкретном варианте, следуйте ему. В этом отношении сам язык R не очень последователен, но тут уж ничего не поделаешь. Не попадитесь в тот же капкан и будьте последовательны, насколько это возможно.

```
= Никогда так не делайте!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

Если у вас имеется семейство аналогичных функций, проследите за тем, чтобы их имена и аргументы согласовывались между собой. Используйте в их именах общий префикс, указывающий на то, что функции взаимосвязаны. Это лучше, чем общий суффикс, поскольку средство автозавершения ввода позволит вам увидеть сразу все семейство, как только вы введете префикс.

```
= Хорошо
input_select()
input_checkbox()
input_text()

= Не очень хорошо
select_input()
checkbox_input()
text_input()
```

Хорошим примером такого дизайна может служить пакет **stringr**: если не можете вспомнить точное имя нужной вам функции, не напрягайте свою память и просто введите `str_`.

Старайтесь избегать перекрывания имен функций и переменных. Вообще говоря, этого риска невозможно избежать ввиду многочисленности имен, уже используемых другими пакетами, но если вы избежите использования большинства наиболее распространенных имен из базовых пакетов R, то это позволит снизить риск возникновения путаницы.

```
# Не делайте так!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

Используйте комментарии, т.е. строки, начинающиеся с символа #, для ответов на вопросы "зачем?", касающиеся вашего кода. Обычно следует избегать комментариев, отвечающих на вопросы "что?" или "как?" Если одного только чтения кода недостаточно для того, чтобы понять, что он делает, то следует подумать над тем, как его следует переписать, чтобы он стал более понятным. Не стоит ли добавить промежуточные переменные с информативными именами? Или вынести подчиненный компонент большой функции в отдельный компонент, чтобы ему можно было присвоить имя? Однако ваш код никогда не сможет пояснить причины, побудившие вас к принятию тех или иных решений. Почему вы выбрали именно этот подход, а не другой? Что еще вы пытались сделать, но это не сработало? Комментарии прекрасно справятся со своим назначением, если будут давать ответы на вопросы такого рода.

Другой важной функцией комментариев является разбиение файла на легко читаемые разделы. Чтобы сделать такие комментарии более заметными, используйте строки заголовков, включающие длинные последовательности символов - или =.

```
# Загрузка данных ---------------------------------------

# Отображение данных в виде графика --------------------
```

RStudio предоставляет комбинацию клавиш, позволяющую создавать такие заголовки (<Ctrl+Shift+R>), и отображает заголовки в раскрывающемся списке навигации, расположенном в левом нижнем углу окна редактора кода (рис. 15.1).

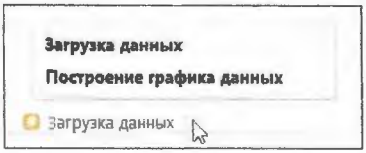


Рис. 15.1

Упражнения

1. Прочитайте исходный код каждой из приведенных ниже функций, определите, что они делают, и придумайте для них лучшие имена.

```
f1 <- function(string, prefix) {
  substr(string, 1, nchar(prefix)) == prefix
}
f2 <- function(x) {
  if (length(x) <= 1) return(NULL)
  x[-length(x)]
}
f3 <- function(x, y) {
  rep(y, length.out = length(x))
}
```

2. Возьмите одну из написанных вами в последнее время функций и потратьте пять минут на то, чтобы придумать лучшие имена для нее самой и ее аргументов.
3. Сравните функции `rnorm()` и `MASS::mvrnorm()`. Что бы вы предложили для того, чтобы сделать их имена более согласованными?
4. Придумайте пример, для которого использовать имена `norm_r()`, `norm_d()` и т.п. было бы лучше, чем имена `rnorm()`, `dnorm()` и т.п. Придумайте противоположный пример.

Условное выполнение

Инструкция `if` позволяет организовать условное выполнение кода. Она имеет следующий синтаксис.

```
if (условие) {
  # код, выполняемый, если условие равно TRUE (ИСТИНА)
} else {
  # код, выполняемый, если условие равно FALSE (ЛОЖЬ)
}
```

При получении справки относительно инструкции `if` ее необходимо заключать в обратные апострофы: `?`if``. Приведенные в справочной системе сведения мало что дадут неопытным программистам, но теперь вы, по крайней мере, знаете, как их получить!

Ниже приведен пример простой функции, в которой используется инструкция `if`. Эта функция возвращает логический вектор, указывающий на то, именован или не именован каждый из элементов вектора.

```
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
  } else {
    !is.na(nms) & nms != ""
  }
}
```

В этой функции используется стандартное правило: функция возвращает последнее вычисленное значение. В данном случае им может быть результат вычисления любой из двух ветвей оператора `if`.

Условия

Результатом вычисления условия должно быть значение `TRUE` или `FALSE`. Если это вектор, вы получите предупреждающее сообщение; если это `NA`, вы получите сообщение об ошибке. Следите за этими сообщениями в своем коде.

```
if (c(TRUE, FALSE)) {}
#> Warning in if (c(TRUE, FALSE)) {:
#> the condition has length > 1 and only the
#> first element will be used
#> NULL

if (NA) {}
#> Error in if (NA) {: missing value where TRUE/FALSE needed
```

Несколько логических выражений можно объединить с помощью операторов `||` (ИЛИ) и `&&` (И), которые используют так называемый *короткий цикл вычислений*. Как только для оператора `||` обнаруживается, что первое условие равно `TRUE`, остальные условия не обрабатываются. Аналогично, если оператор `&&` обнаруживает, что первое условие равно `FALSE`, он не обрабатывает остальные условия. Никогда не используйте операторы `|` и `&` в инструкции `if`: эти операции векторизованы и применяются к нескольким значениям (именно поэтому они используются в функции `filter()`). Если у вас имеется логический вектор, можете свернуть его в одиночное значение с помощью функции `any()` или `all()`.

Будьте внимательны при проверке на равенство. Оператор `==` векторизован, а это означает, что может быть получено более чем одно выходное

значение. В таком случае можете проверить, что длина переменной уже равна 1, свернуть вектор с помощью функции `all()` или `any()` или использовать не векторизованную функцию `identical()`. Функция `identical()` очень строгая: она всегда возвращает либо одиночное значение `TRUE`, либо одиночное значение `FALSE` и не выполняет приведение типов. Это означает, что вы должны быть очень внимательны, сравнивая целые числа с числами двойной точности.

```
identical(0L, 0)
=> [1] FALSE
```

Также следует проявлять осторожность при работе с вещественными числами.

```
x <- sqrt(2) ^ 2
x
=> [1] 2
x == 2
=> [1] FALSE
x - 2
=> [1] 4.44e-16
```

Вместо этого используйте для сравнения функцию `dplyr::near()`, которая обсуждалась в главе 3.

Кроме того, запомните, что сравнение `x == NA` не принесет вам никакой пользы!

Множественные условия

Несколько инструкций `if` можно объединять в цепочки.

```
if (this) {
= сделай это
  else if (that) {
= сделай что-то еще
  else {
=
```

Но если такая цепочка получается очень длинной, подумайте над тем, нельзя ли переписать код по-другому. Одна полезная методика предполагает использование функции `switch()`. Эта функция позволяет выбирать ветвь кода на основании заданной позиции или имени.

```
=> function(x, y, op) {
=>   switch(op,
```

```
#>     plus = x + y,
#>     minus = x - y,
#>     times = x * y,
#>     divide = x / y,
#>     stop("Unknown op!")
#>   )
#> }
```

Другой полезной функцией, с помощью которой часто удается избавиться от длинных цепочек инструкций `if`, является функция `cut()`, которая используется для дискретизации непрерывных переменных.

Стиль кода

Как за инструкцией `if`, так и за любой функцией (почти) всегда должны следовать фигурные скобки (`{}`), содержимое внутри которых должно отделяться двумя пробелами. Так легче отслеживать иерархию в коде, скользя взглядом вдоль левого поля.

Открывающая фигурная скобка никогда не должна располагаться на отдельной строке, и за ней всегда должен следовать переход на новую строку. Закрывающая фигурная скобка всегда должна располагаться на отдельной строке, если за ней не следует блок `else`. Код в фигурных скобках следует всегда отделять пробелами.

```
# Хорошо
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}

# Плохо
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

В случае коротких инструкций, умещающихся в одной строке, фигурные скобки можно опустить.

```
y <- 10
x <- if (y < 20) "Too low" else "Too high"
```

Мы рекомендуем использовать эту возможность только для очень коротких инструкций. В противном случае полный синтаксис значительно облегчит чтение кода.

```
if (y < 20) {
  x <- "Too low"
} else {
  x <- "Too high"
}
```

Упражнение

1. В чем разница между инструкциями `if` и `ifelse()`? Тщательно изучите справочную документацию и составьте три примера, иллюстрирующие ключевые различия между этими инструкциями.
2. Напишите функцию, выводящую одно из приветствий "Доброе утро!", "Добрый день!" или "Добрый вечер!", в зависимости от времени суток. (Подсказка: используйте аргумент, задающий время, который по умолчанию принимает значение `lubridate::now()`. Это упростит тестирование функции.)
3. Реализуйте функцию `fizzbuzz()`. Ее единственным аргументом является число. Если число делится на три, функция возвращает строку "fizz". Если число делится на пять, функция возвращает строку "buzz". Если же число делится и на три, и на пять, функция возвращает строку "fizzbuzz". В любом другом случае возвращается само число. Обязательно сначала напишите работающий код и лишь после этого создавайте функцию.
4. Как использовать функцию `cut()` для упрощения следующего набора вложенных инструкций `if/else`?

```
if (temp <= 0) {
  "freezing"
} else if (temp <= 10) {
  "cold"
} else if (temp <= 20) {
  "cool"
} else if (temp <= 30) {
```

```
  "warm"
} else {
  "hot"
}
```

Как следует изменить вызов функции `cut()`, если использовать оператор `<` вместо оператора `<=`? Каково другое главное преимущество функции `cut()` в связи с этой проблемой? (Подсказка: что произойдет, если `temp` содержит несколько значений?)

5. Что произойдет, если использовать функцию `switch()` с числовыми значениями?

6. Что делает приведенный ниже вызов функции `switch()`? Что произойдет, если `x` равно "e"?

```
switch(x,
  a = ,
  b = "ab",
  c = ,
  d = "cd"
)
```

Сначала проведите самостоятельные эксперименты и лишь затем прочитайте документацию.

Аргументы функций

Как правило, аргументы функций можно разделить на две широкие категории: одни из них предоставляют данные для вычислений, а другие — параметры, управляющие деталями вычислений. Рассмотрим примеры.

- Функция `log()`: данные — `x`, детали вычислений — основание логарифма.
- Функция `mean()`: данные — `x`, детали вычислений — сколько данных отсекается с конца (`trim`) и как обрабатываются отсутствующие значения (`na.rm`).
- Функция `t.test()`: данные — `x` и `y`, детали тестирования — `alternative`, `mu`, `paired`, `var.equal` и `conf.level`.
- Функция `str_c()`: вы можете предоставить любое количество строк для конкатенации, а детали конкатенации контролируются параметрами `sep` и `collapse`.

Как правило, первыми идут аргументы данных. Аргументы, управляющие деталями вычислений, должны идти в конце, и обычно для них должны предоставляться значения по умолчанию, которые указываются точно так же, как и при вызове функций с именованными аргументами.

```
# Вычисление доверительного интервала вокруг среднего
# с использованием нормального распределения
mean_ci <- function(x, conf = 0.95) {
  se <- sd(x) / sqrt(length(x))
  alpha <- 1 - conf
  mean(x) + se * qnorm(c(alpha / 2, 1 - alpha / 2))
}

x <- runif(100)
mean_ci(x)
#> [1] 0.498 0.610
mean_ci(x, conf = 0.99)
#> [1] 0.480 0.628
```

В качестве значения по умолчанию почти всегда следует использовать наиболее часто используемое значение. Исключения из этого правила связаны с соображениями безопасности. Например, в качестве значения по умолчанию для `na.rm` разумно принять `FALSE`, поскольку отсутствующие значения играют важную роль. Даже если вы привыкли вставлять в свой код `na.rm = TRUE`, по умолчанию игнорировать отсутствующие значения — неудачная идея.

Когда вы вызываете функцию, то обычно опускаете имена аргументов данных, поскольку они используются повсеместно. Если вы перекрываете значение по умолчанию аргумента, управляющего вычислениями, то должны использовать полное имя.

```
# Хорошо
mean(1:10, na.rm = TRUE)

# Плохо
mean(x = 1:10, , FALSE)
mean(, TRUE, x = c(1:10, NA))
```

На аргумент можно ссылаться по его уникальному префиксу (например, `mean(x, n = TRUE)`), но обычно, учитывая возможность возникновения путаницы, этого лучше избегать.

Заметьте, что в вызове функции знак `=` следует окружать пробелами и всегда помещать пробел после запятой, а не перед ней (как и в обычном языке). Использование пробелов облегчает восприятие важных компонентов.

```
# Хорошо
average <- mean(feet / 12 + inches, na.rm = TRUE)

# Плохо
average<-mean(feet/12+inches,na.rm=TRUE)
```

Выбор имен

Имена аргументов также играют важную роль. Для R они безразличны, но для читателей вашего кода (включая вас самих в будущем!) — вовсе нет. Обычно следует отдавать предпочтение длинным описательным именам, но существует множество обычных, очень коротких имен. Вот некоторые из них, которые вам стоит запомнить:

- `x`, `y`, `z` — векторы;
- `w` — вектор весов;
- `df` — фрейм данных;
- `i`, `j` — числовые индексы (обычно строки и столбцы);
- `n` — длина, или количество строк;
- `p` — количество столбцов.

Во всех остальных случаях попробуйте подыскать имена среди аналогичных аргументов существующих функций R. Например, используйте имя `na.rm` для переменной, определяющей, следует ли исключать отсутствующие значения.

Проверка значений

Написав изрядное количество функций, вы в конечном счете придете к тому, что не сможете точно вспомнить, как работает каждая из них. В такой ситуации можно очень легко вызвать функцию с некорректными входными данными. Во избежание этой проблемы часто бывает полезно сделать ограничения явными. Предположим, вы написали функции для вычисления взвешенных итоговых статистик.

```
wt_mean <- function(x, w) {
  sum(x * w) / sum(x)
}
wt_var <- function(x, w) {
  mu <- wt_mean(x, w)
  sum(w * (x - mu) ^ 2) / sum(w)
```

```
}
wt_sd <- function(x, w) {
  sqrt(wt_var(x, w))
}
```

Что произойдет в том случае, если `x` и `w` имеют разную длину?

```
wt_mean(1:6, 1:3)
#> [1] 2.19
```

В данном случае, учитывая правила "зацикливания" векторов R, мы не получим сообщение об ошибке.

Хорошая практика заключается в проверке важных предусловий и генерировании ошибки (с помощью функции `stop()`), если они не выполняются.

```
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(x)
}
```

Будьте осмотрительны и не перестарайтесь. Ищите разумный компромисс между тем, сколько времени вы потратите на то, чтобы сделать свою функцию робастной (устойчивой к ошибкам), и сколько времени заняло ее написание. Например, если бы вы добавили аргумент `na.rm`, то мы, вероятно, не проверяли бы его слишком тщательно.

```
wt_mean <- function(x, w, na.rm = FALSE) {
  if (!is.logical(na.rm)) {
    stop("`na.rm` must be logical")
  }
  if (length(na.rm) != 1) {
    stop("`na.rm` must be length 1")
  }
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
```

Для получения столь небольшого выигрыша нам пришлось проделать много дополнительной работы. Полезным компромиссом является использование встроенной функции `stopifnot()`, которая выполняет проверку того, что каждый аргумент равен `TRUE`, и генерирует типовое сообщение об ошибке, если это не так.

```
wt_mean <- function(x, w, na.rm = FALSE) {
  stopifnot(is.logical(na.rm), length(na.rm) == 1)
  stopifnot(length(x) == length(w))

  if (na.rm) {
    miss <- is.na(x) | is.na(w)
    x <- x[!miss]
    w <- w[!miss]
  }
  sum(w * x) / sum(x)
}
wt_mean(1:6, 6:1, na.rm = "foo")
#> Error: is.logical(na.rm) is not TRUE
```

Заметьте: когда вы используете функцию `stopifnot()`, вы утверждаете, что именно должно быть истинным, а не проверяете, что может быть неправильным.

Точка-точка-точка (...)

В R многие функции принимают произвольное количество входных значений.

```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

Как работают такие функции? Они полагаются на специальный аргумент `...` (многоточие). Этот специальный аргумент захватывает любое количество аргументов, которым не назначено явных имен.

Такой подход очень удобен, поскольку далее вы можете передать эти "точки" другой функции. Это особенно полезно, если ваша функция в основном используется в качестве оболочки другой функции. Например, обычно мы создаем следующие вспомогательные функции-оболочки для функции `str_c()`.

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
```

```
#> [1] "a, b, c, d, e, f, g, h, i, j"

rule <- function(..., pad = "-") {
  title <- paste0(...)
  width <- getOption("width") - nchar(title) - 5
  cat(title, " ", stringr::str_dup(pad, width), "\n", sep = "")
}
rule("Важный вывод")
#> Важный вывод ----------------------------------------
```

В данном случае точки (`...`) позволяют передать функции `str_c()` любые аргументы, не нуждающиеся в обработке. Это очень удобный прием. Но за все приходится платить: любые неправильно записанные аргументы не будут генерировать ошибку. Из-за этого опечатки могут легко пройти незамеченными.

```
x <- c(1, 2)
sum(x, na.mr = TRUE)
#> [1] 4
```

Если вы хотите всего лишь извлечь значения `...`, используйте функцию `list(...)`.

Ленивые вычисления

В R аргументы обрабатываются в соответствии с концепцией так называемых *ленивых* (отложенных) вычислений: вычисление аргументов откладывается до тех пор, пока в этом не возникнет необходимости. Отсюда следует, что те аргументы, которые не используются в функции, никогда не вычисляются. Однако при написании собственных функций для анализа данных это важное свойство R как языка программирования вам, вообще говоря, не понадобится. Более подробную информацию о ленивых вычислениях можно найти по адресу `http://adv-r.had.co.nz/Functions.html#lazy-evaluation`[1].

Упражнения

1. Что делает команда `commas(letters, collapse = "-")` и зачем?
2. Было бы неплохо иметь возможность предоставить аргументу `pad` несколько символов, например `rule("Title", pad = "-+")`. Почему сейчас это не реализовано? Как бы вы исправили эту ситуацию?

[1] См. также следующие статьи: `https://ru.wikipedia.org/wiki/Ленивые_вычисления` и `https://habrahabr.ru/post/247213/`. — *Примеч. ред.*

3. Какова роль аргумента `trim` функции `mean()`? В каких ситуациях вы будете его использовать?
4. Значением по умолчанию аргумента `method` функции `cor()` является `c("pearson", "kendall", "spearman")`. Что это означает? Какое значение используется по умолчанию?

Возвращаемые значения

Обычно определение того, что должна возвращать ваша функция, не составляет большого труда: ведь именно для этого прежде всего и создавалась функция! Однако при организации возврата значения полезно иметь в виду следующие моменты.

- Облегчит ли чтение текста функции преждевременный возврат?
- Нельзя ли сделать функцию пригодной для включения в цепочки канала?

Явные команды возврата

Обычно возврат значения является последней инструкцией функции, однако можно организовать ранний возврат с помощью функции `return()`. Мы считаем, что эту функцию следует приберегать для тех случаев, когда вы хотите подчеркнуть, что в случае простого решения возможен ранний возврат. Типичным примером ситуаций подобного рода могут служить пустые входные значения.

```
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }

  # Далее идет сложный код
}
```

Другой ситуацией, в которой целесообразно использовать ранний возврат, является инструкция `if` с одним сложным и одним простым блоками. Например, вы могли написать такую инструкцию `if`.

```
f <- function() {
  if (x) {
    # Сделать
    # что-то,
    # что
```

```
    # занимает
    # много
    # строк
    # кода
  } else {
    # вернуть что-то короткое
  }
}
```

Но если первый блок очень длинный, то к тому времени, когда вы дойдете до `else`, вы забудете об условии. Один из способов устранения этого недостатка заключается в использовании раннего возврата в простом случае.

```
f <- function() {
  if (!x) {
    return(something_short)
  }

    # Сделать
    # что-то,
    # что
    # занимает
    # много
    # строк
    # кода

}
```

В этом случае вы обратите внимание на ранний возврат сразу же, не просматривая весь длинный контекст.

Написание функций, пригодных для включения в канал

Если вы хотите писать собственные функции, способные работать в канале, то вопрос о возвращаемом значении приобретает особое значение. Такие функции можно разбить на два основных типа: преобразования и побочные эффекты.

В случае *преобразующих* функций имеется четко определенный "первичный" объект, который передается в качестве первого аргумента, и его модифицированная версия возвращается функцией. Например, для функций пакетов **dplyr** и **tidyr** ключевыми объектами являются фреймы данных. Если вы можете определить тип объекта для своей предметной области, то ваши функции будут отлично работать с каналом.

Функции с *побочным эффектом* вызываются в первую очередь для выполнения какого-либо действия, например для рисования графика или сохранения файла, а не для преобразования объекта. Эти функции должны возвращать первый аргумент "невидимо", поэтому по умолчанию они ничего не выводят, но могут быть использованы в канале. Например, следующая простая функция выводит количество отсутствующих значений во фрейме данных.

```
show_missings <- function(df) {
  n <- sum(is.na(df))
  cat("Отсутствующие значения: ", n, "\n", sep = "")

  invisible(df)
}
```

Если мы вызываем ее интерактивно, то вызов `invisible()` означает, что входной объект `df` не выводится.

```
show_missings(mtcars)
#> Отсутствующие значения: 0
```

Но он есть, просто по умолчанию не выводится.

```
x <- show_missings(mtcars)
#> Отсутствующие значения: 0
class(x)
#> [1] "data.frame"
dim(x)
#> [1] 32 11
```

И мы все еще можем использовать функцию в канале.

```
mtcars %>%
  show_missings() %>%
  mutate(mpg = ifelse(mpg < 20, NA, mpg)) %>%
  show_missings()
#> Отсутствующие значения: 0
#> Отсутствующие значения: 18
```

Окружение

Последний компонент функции — ее окружение. Это не та тема, в которой вы должны хорошо разбираться, когда пишете свои первые функции. И тем не менее очень важно, чтобы вы знали хотя бы что-то об окружении, поскольку оно в значительной мере определяет, как работают функции. Окружение

функции управляет тем, как R находит значение, связанное с именем. Возьмем, например, следующую функцию.

```
f <- function(x) {
  x + y
}
```

Во многих языках программирования этот код будет признан ошибочным, ведь в теле функции отсутствует определение переменной y. Но в данном случае это совершенно нормальный код, так как в R для определения значений переменных используется так называемая *лексическая область видимости*. Поскольку переменная y не определена в функции, R будет искать ее в *окружении*, содержащем определение этой функции.

```
y <- 100
f(10)
#> [1] 110

y <- 1000
f(10)
#> [1] 1010
```

Казалось бы, описанное поведение — прямой путь к ошибкам, и на практике следует избегать создания подобных функций, но в целом это не создает серьезных проблем (особенно, если вы регулярно перезапускаете R для получения чистого состояния).

Преимуществом же такого поведения является то, что с точки зрения языка оно позволяет R быть весьма последовательным. Все имена просматриваются с использованием одного и того же набора правил. В случае функции f() эти правила затрагивают поведение двух синтаксических элементов, которое может быть для вас неожиданным: { и +. Вот еще один пример необычного для других языков программирования кода, допустимого в R.

```
`+` <- function(x, y) {
  if (runif(1) < 0.1) {
    sum(x, y)
  } else {
    sum(x, y) * 1.1
  }
}
table(replicate(1000, 1 + 2))
#>
#>   3 3.3
#> 100 900
rm(`+`)
```

Эти примеры отражают общую картину, свойственную R. В среде R мало что ограничивает свободу ваших действий, и часто допускается делать даже то, что запрещено в других языках программирования. Для вас также доступны примерно 99% тех действий, которые настоятельно не рекомендуются к применению (как в данном случае, где переопределяется работа оператора сложения). Но именно эта мощь и гибкость сделали возможным создание таких инструментов, как **ggplot2** и **dplyr**. Изучение наиболее оптимальных способов применения подобной гибкости выходит за рамки данной книги, однако более подробную информацию об этом вы сможете найти в книге *Advanced R* (`http://adv-r.had.co.nz/`).

ГЛАВА 16

Векторы

Введение

До сих пор наше внимание в основном было сосредоточено на tibble-фреймах и пакетах, которые работают с ними. Но как только вы начнете писать собственные функции и углубитесь в R, вам понадобятся векторы — объекты, лежащие в основе tibble-фреймов. Если вы изучали R традиционным способом, то, вероятно, уже знакомы с векторами, поскольку в большинстве учебников изучение R начинают с векторов, переходя далее к tibble-фреймам. Мы считаем, что лучше начинать с tibble-фреймов, поскольку непосредственно используются именно они, и лишь затем переходить к изучению компонентов, лежащих в их основе.

В частности, векторы важны тем, что большинство функций, которые вы напишете, будут предназначены для работы с ними. Конечно, можно написать функции, работающие с tibble-фреймами (как функции пакетов **ggplot2**, **dplyr** и **tidyr**), но имеющиеся в настоящее время инструменты, необходимые для их написания, весьма своеобразны и все еще далеки от совершенства. Хэдли Уикем разрабатывает лучший подход (`https://github.com/hadley/lazyeval`), но к моменту выхода книги эта работа еще не будет закончена. Но даже после ее завершения вам все равно понадобится умение обращаться с векторами; новый подход лишь облегчит написание располагающегося поверх них слоя, дружественного по отношению к пользователю.

Необходимые ресурсы

В этой главе в центре нашего внимания будут базовые структуры данных R, поэтому загрузка пакетов не потребуется. Однако для того, чтобы

избежать некоторых несогласованностей, свойственных базовым пакетам R, мы используем ряд функций из пакета purrr.

```
library(tidyverse)
#> Loading tidyverse: ggplot2
#> Loading tidyverse: tibble
#> Loading tidyverse: tidyr
#> Loading tidyverse: readr
#> Loading tidyverse: purrr
#> Loading tidyverse: dplyr
#> Conflicts with tidy packages ----------------------------------
#> filter(): dplyr, stats
#> lag():    dplyr, stats
```

Основные сведения о векторах

Есть два основных типа векторов: атомарные векторы и списки.

- *Атомарные векторы* включают шесть подтипов: *логический* (logical), *целочисленный* (integer), *вещественный* (double), *символьный* (character), *комплексный* (complex) и *двоичные ("сырые") данные* (raw). Целочисленные и вещественные векторы совокупно называют *числовыми*.
- *Списки*, которые иногда называют *рекурсивными векторами*, поскольку они могут содержать другие списки.

Главное различие между атомарными векторами и списками заключается в том, что первые *однородны*, тогда как вторые могут быть *неоднородными*, т.е. могут содержать элементы разных типов. Существует еще один родственный им объект: NULL. Его часто используют для представления отсутствия вектора (в отличие от значения NA, используемого для представления отсутствующего значения в векторе). Обычно NULL ведет себя как вектор нулевой длины. Взаимосвязь различных типов векторов показана на рис. 16.1.

Каждый вектор характеризуется двумя основными свойствами.

- *Тип*. Для определения типа вектора служит функция typeof().

  ```
  typeof(letters)
  #> [1] "character"
  typeof(1:10)
  #> [1] "integer"
  ```

- *Длина*. Для определения длины вектора служит функция length().

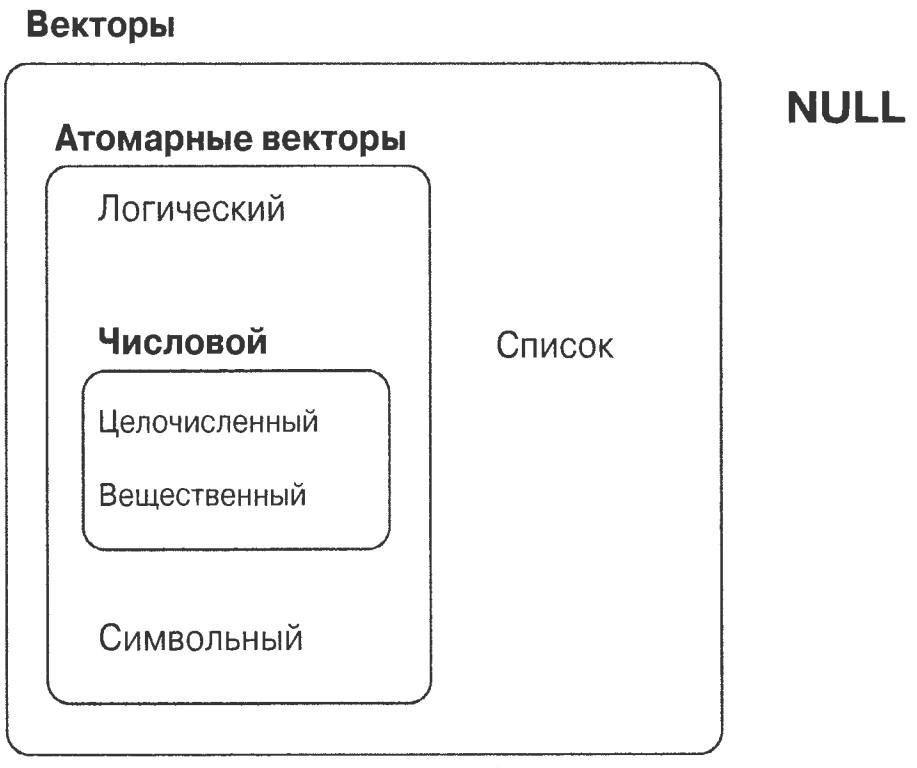


Рис. 16.1

```
x <- list("a", "b", 1:10)
length(x)
#> [1] 3
```

Кроме того, векторы могут содержать произвольные дополнительные метаданные в форме атрибутов, которые используются для создания *расширенных* векторов, обладающих дополнительными аспектами поведения. Существуют четыре важных типа расширенных векторов:

- *факторы* — строятся поверх целочисленных векторов;
- векторы *даты* и *даты/времени* — строятся поверх числовых векторов;
- *фреймы данных* и *tibble-фреймы* — строятся поверх списков.

В этой главе вы познакомитесь со всеми важными типами векторов, начиная с наиболее простых и заканчивая самыми сложными. Мы начнем с атомарных векторов, затем перейдем к спискам, а закончим расширенными векторами.

Важные типы атомарных векторов

Четырьмя наиболее важными типами атомарных векторов являются логический, целочисленный, вещественный и символьный. Двоичный и комплексный типы редко используются в анализе данных, поэтому здесь они не обсуждаются.

Логический тип

Логический тип — простейший тип атомарных векторов, поскольку логические векторы могут содержать только три возможных значения: `FALSE`, `TRUE` и `NA`. Обычно логические векторы создаются с помощью операторов сравнения, описанных в главе 3. Их можно создавать вручную с помощью функции `c()`.

```
1:10 %% 3 == 0
#>  [1] FALSE FALSE  TRUE FALSE FALSE
#>  [2] TRUE FALSE FALSE  TRUE FALSE
c(TRUE, TRUE, FALSE, NA)
#> [1]  TRUE  TRUE FALSE    NA
```

Числовой тип

Целочисленные и вещественные векторы совокупно называются числовыми векторами. В языке R числа являются вещественными по умолчанию. Число обозначается как целое следующей за ним буквой L.

```
typeof(1)
#> [1] "double"
typeof(1L)
#> [1] "integer"
1.5L
#> [1] 1.5
```

Как правило, различие между целым и вещественным типами не принципиально, за исключением двух важных случаев, о которых вам следует знать.

- Вещественные числа являются приближенными. Это значения, представляющие числа с плавающей точкой, которые, однако, не всегда могут быть точно представлены с использованием памяти фиксированного объема. Например, чему равен квадратный корень из двух?

  ```
  x <- sqrt(2) ^ 2
  x
  #> [1] 2
  ```

```
x - 2
#> [1] 4.44e-16
```

Такое поведение является типичным при работе с вещественными числами: большинство результатов вычислений включает ошибку аппроксимации. Вместо сравнения вещественных чисел между собой с помощью операторов `==` вы должны использовать функцию `dplyr::near()`, которая учитывает заданную величину допустимой погрешности.

- Целые числа включают одно специальное значение — `NA`, тогда как среди вещественных чисел таких значений четыре: `NA`, `NaN`, `Inf` и `-Inf`. Все они могут появляться при делении чисел.

```
c(-1, 0, 1) / 0
#> [1] -Inf  NaN  Inf
```

Избегайте использования оператора `==` для тестирования указанных специальных значений. Вместо этого используйте вспомогательные функции `is.finite()`, `is.infinite()`, `is.na()` и `is.nan()` (табл. 16.1).

Таблица 16.1

	0	Inf	NA	NaN
`is.finite()`	x			
`is.infinite()`		x		
`is.na()`			x	x
`is.nan()`				x

Символьный тип

Символьные векторы — самый сложный тип атомарных векторов, поскольку каждый элемент такого вектора является строкой, а строка может содержать произвольное количество данных.

В главе 11 вы уже познакомились с тем, как работать со строками. Здесь мы хотим отметить одно важное свойство базовой реализации строк: R использует глобальный пул строк. Это означает, что каждая уникальная строка сохраняется в памяти лишь один раз, и всякий раз, когда она используется, фактически используется указатель на представление строки. Это позволяет снизить расход памяти при дублировании строк, в чем можно непосредственно убедиться с помощью функции `pryr::object_size()`.

```
x <- "Это достаточно длинная строка."
pryr::object_size(x)
```

```
#> 136 B

y <- rep(x, 1000)
pryr::object_size(y)
#> 8.13 kB
```

Переменная y не занимает в 1000 раз больше памяти по сравнению с x, поскольку каждый элемент y — всего лишь указатель на одну и ту же строку. Указатель занимает 8 байт, поэтому для 1000 указателей на строку, занимающую 136 байт, требуется память объемом 8 * 1000 + 136 = 8,13 Кбайт.

Отсутствующие значения

Заметьте, что для каждого типа атомарных векторов предусмотрено свое отсутствующее значение.

```
NA            # логическое
#> [1] NA
NA_integer_   # целочисленное
#> [1] NA
NA_real_      # вещественное
#> [1] NA
NA_character_ # символьное
#> [1] NA
```

Обычно вам необязательно знать все возможные типы отсутствующих значений, поскольку вы всегда можете использовать значение NA, и оно будет автоматически преобразовано в корректный тип на основе правил неявного приведения типов, которые описаны далее. Однако существуют некоторые функции, которые строго относятся к типам своих входных данных, и поэтому знание перечисленных типов пригодится вам, когда ситуация потребует от вас быть более точным.

Упражнения

1. Опишите различие между выражениями `is.finite(x)` и `!is.infinite(x)`.
2. Прочитайте исходный код функции `dplyr::near()`. (Подсказка: для того чтобы просмотреть исходный код, опустите `()` после названия функции при вводе команды.) Как работает эта функция?
3. Логический вектор может принимать три возможных значения. Сколько возможных значений может принимать целочисленный вектор? Сколько

возможных значений может принимать вещественный вектор? Исследуйте эти вопросы с помощью Google.

4. Придумайте не менее четырех функций, позволяющих преобразовать вещественное число в целочисленное. Чем они отличаются друг от друга? Будьте точны.
5. Какие функции из пакета **readr** позволяют преобразовать строку в логический, целочисленный и вещественный вектор?

Использование атомарных векторов

Теперь, когда вам уже известны различные типы атомарных векторов, вам не помешает ближе познакомиться с некоторыми важными инструментами для работы с ними. В этом разделе обсуждаются следующие вопросы:

- как преобразовать один тип в другой и в каких случаях выполняется автоматическое преобразование типов;
- как определить принадлежность объекта к определенному типу векторов;
- что происходит, когда вы работаете с векторами, имеющими разную длину;
- как присвоить имена элементам вектора;
- как извлечь элементы, представляющие интерес.

Приведение типов

Существуют два способа преобразования, или *приведения*, одного векторного типа в другой.

- Явное приведение типов происходит тогда, когда вы вызываете такие функции, как `as.logical()`, `as.integer()`, `as.double()` или `as.character()`. Каждый раз, когда используется явное приведение типов, в первую очередь следует проверять, чтобы у вас не получился вектор неверного типа. Например, может потребоваться исправление спецификации `col_types` в пакете **readr**.
- Неявное преобразование типов происходит тогда, когда вы используете вектор в специфическом контексте, ожидающем вектор определенного типа. Например, это может происходить тогда, когда вы используете

логический вектор с функцией суммирования чисел или когда вещественный вектор используется там, где ожидается целочисленный.

Поскольку явное преобразование типов используется сравнительно редко и в основном его легко понять, мы обсудим здесь более подробно лишь неявное приведение.

Вы уже знакомы с примерами наиболее важного вида неявного приведения: использование логического вектора в числовом контексте. В этом случае `TRUE` превращается в 1, а `FALSE` — в 0. Это означает, что суммирование логического вектора дает количество истинных значений, а среднее логического вектора дает долю таких значений.

```
x <- sample(20, 100, replace = TRUE)
y <- x > 10
sum(y)  # сколько чисел, больших 10?
#> [1] 44
mean(y) # какова доля чисел, больших 10?
#> [1] 0.44
```

Возможно, вам встречался код (скорее всего, устаревший), рассчитанный на неявное преобразование типов в обратном направлении: от целочисленного типа к логическому.

```
if (length(x)) {
  # сделать что-то
}
```

В данном случае 0 преобразуется в `FALSE`, а все остальные значения преобразуются в `TRUE`. Мы считаем, что это затрудняет понимание кода, и не рекомендуем использовать такой подход. Вместо этого лучше явно выразить свои намерения: `length(x) > 0`.

Очень важно понимать, что происходит, когда вы пытаетесь создать вектор, содержащий несколько значений, с помощью функции `c()`, — верх одерживает наиболее сложный тип.

```
typeof(c(TRUE, 1L))
#> [1] "integer"
typeof(c(1L, 1.5))
#> [1] "double"
typeof(c(1.5, "a"))
#> [1] "character"
```

Атомарный вектор не может содержать смесь нескольких типов, поскольку тип — это свойство вектора в целом, а не его отдельных элементов. Если вам

нужна смесь нескольких типов в одном векторе, используйте списки, к рассмотрению которых мы вскоре перейдем.

Функции проверки типов

Иногда требуется выполнять разные действия в зависимости от типа вектора. Один из способов сделать это — использовать функцию `typeof()`. Другая возможность заключается в использовании тестовых функций, возвращающих значение `TRUE` или `FALSE`. Базовые пакеты R предоставляют ряд функций наподобие `is.vector()` и `is.atomic()`, но они часто возвращают неожиданные результаты. Вместо этого безопаснее использовать функции семейства `is_*()`, предоставляемые пакетом **purrr**, которые сведены в табл. 16.2.

Таблица 16.2

	lgl	int	dbl	chr	list
`is_logical()`	x				
`is_integer()`		x			
`is_double()`			x		
`is_numeric()`		x	x		
`is_character()`				x	
`is_atomic()`	x	x	x	x	
`is_list()`					x
`is_vector()`	x	x	x	x	x

Для каждого предиката также предусмотрена скалярная версия наподобие `is_scalar_atomic()`, которая проверяет, равна ли длина 1. Например, это может вам пригодиться, если вы хотите проверить, является ли аргумент вашей функции одиночным логическим значением.

Скаляры и правила зацикливания

Аналогично тому, как правила неявного приведения типов обеспечивают совместимость векторов, R неявно приводит в соответствие длины векторов. Этот процесс называется *зацикливанием* (recycling), поскольку более короткий вектор повторяется, или зацикливается, для достижения той же длины, что и более длинный.

Обычно это больше всего нужно в тех случаях, когда смешиваются векторы и "скаляры". Термин *скаляры* взят в кавычки, поскольку на самом деле в R скаляры не определены: вместо этого считается, что одиночное число — это

вектор единичной длины. Ввиду отсутствия скаляров большинство встроенных функций *векторизовано*, т.е. воздействует на числовой вектор. Именно поэтому, например, работает следующий код.

```
sample(10) + 100
#>  [1] 109 108 104 102 103 110 106 107 105 101
runif(10) > 0.5
#>  [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE
#>  [10]  TRUE
```

В R основные математические операции работают с векторами. Это означает, что при формировании простых математических вычислений вам никогда не придется явно организовывать итерирование по элементам.

В случае сложения двух векторов одинаковой длины или вектора и "скаляра" интуитивно понятно, что должно происходить, но что будет, если попытаться сложить два вектора разной длины?

```
1:10 + 1:2
#>  [1]  2  4  4  6  6  8  8 10 10 12
```

В подобных случаях R расширяет, или, как говорят, *зацикливает*, т.е. многократно повторяет, короткий вектор до достижения длины более длинного вектора. Этот процесс осуществляется без вывода какого-либо сообщения, за исключением случаев, когда длина более длинного вектора не является кратной длине короткого вектора.

```
1:10 + 1:3
#> Warning in 1:10 + 1:3:
#> longer object length is not a multiple of shorter
#> object length
#>  [1]  2  4  6  5  7  9  8 10 12 11
```

Несмотря на то что зацикливание векторов может быть эффективно использовано для создания компактного кода, это чревато возникновением проблем, причины которых будет трудно обнаружить. Поэтому при зацикливании любых объектов, кроме скаляров, векторизованные функции библиотеки *tidyverse* генерируют ошибки. Если необходимо использовать зацикливание, то это следует сделать явно с помощью функции `rep()`.

```
tibble(x = 1:4, y = 1:2)
#> Error: Variables must be length 1 or 4.
#> Problem variables: 'y'

tibble(x = 1:4, y = rep(1:2, 2))
#> # A tibble: 4 × 2
```

```
#>        x     y
#>    <int> <int>
#> 1      1     1
#> 2      2     2
#> 3      3     1
#> 4      4     2
```

```
tibble(x = 1:4, y = rep(1:2, each = 2))
#> # A tibble: 4 × 2
#>        x     y
#>    <int> <int>
#> 1      1     1
#> 2      2     1
#> 3      3     2
#> 4      4     2
```

Именование векторов

Векторам любого типа можно присваивать имена. Это можно сделать в процессе создания вектора с помощью функции `c()`.

```
c(x = 1, y = 2, z = 4)
#> x y z
#> 1 2 4
```

Также это можно сделать постфактум с помощью функции `purrr::set_names()`.

```
set_names(1:3, c("a", "b", "c"))
#> a b c
#> 1 2 3
```

Именованные векторы удобнее всего использовать для извлечения элементов векторов, о чем рассказано в следующем разделе.

Извлечение элементов

До сих пор для фильтрации строк tibble-фрейма мы использовали функцию `dplyr::filter()`. Функция `filter()` работает только с tibble-фреймами, поэтому для векторов необходимо использовать новый инструмент, каковым является функция `[`, которая вызывается примерно следующим образом: `x[a]`. Для управления извлечением элементов можно использовать одну из четырех перечисленных ниже структур.

- Числовой вектор, содержащий только целые числа. Целые числа должны быть либо все положительные, либо все отрицательные, либо нуль.

Извлекаются элементы вектора, соответствующие указанным позициям.

```
x <- c("one", "two", "three", "four", "five")
x[c(3, 2, 5)]
#> [1] "three" "two"    "five"
```

В действительности, повторяя некоторые позиции, можно получить выходной вектор большей длины, чем входной.

```
x[c(1, 1, 5, 5, 5, 2)]
#> [1] "one"  "one"  "five" "five" "five" "two"
```

Отрицательные значения отбрасывают соответствующие элементы.

```
x[c(-1, -3, -5)]
#> [1] "two"  "four"
```

Смешивать положительные и отрицательные значения нельзя.

```
x[c(1, -1)]
#> Error in x[c(1, -1)]:
#> only 0's may be mixed with negative subscripts
```

Последнее сообщение об ошибке уведомляет о том, что только нуль может смешиваться с отрицательными индексами. В свою очередь, нуль вообще не возвращает никаких значений:

```
x[0]
#> character(0)
```

Эта возможность не из тех, которыми пользуются очень часто, но она может оказаться очень полезной, если вы хотите создать необычные структуры данных для тестирования своих функций.

- При извлечении элементов с помощью логического вектора сохраняются все значения, соответствующие значениям TRUE. Этот способ наиболее полезен в сочетании с функциями сравнения.

```
x <- c(10, 3, NA, 5, 8, 1, NA)

# Все значения в x, кроме отсутствующих
x[!is.na(x)]
#> [1] 10  3  5  8  1

# Все четные (или отсутствующие!) значения в x
x[x %% 2 == 0]
#> [1] 10 NA  8 NA
```

- Элементы именованного вектора можно извлекать с помощью символьного вектора.

```
x <- c(abc = 1, def = 2, xyz = 5)
x[c("xyz", "def")]
#> xyz def
#>   5   2
```

Как и в случае целых чисел, символьный вектор также позволяет дублировать отдельные элементы.

- Простейшей из структур, управляющих извлечением элементов векторов, является пустая структура, `x[]`, которая возвращает целый вектор `x`. В случае векторов она бесполезна, но ее можно с успехом использовать для извлечения элементов матриц (и других высокоразмерных структур), поскольку, например, с ее помощью можно выбрать все строки или все столбцы, оставляя соответствующий индекс пустым. К примеру, если `x` — двумерная матрица, то `x[1, ]` выбирает все столбцы первой строки, а `x[,-1]` — все строки и все столбцы, за исключением первого.

Подробнее об извлечении элементов различных объектов можно прочитать в главе "Subsetting" книги *Advanced R*.

Существует важное видоизменение функции `[`: функция `[[`. Эта функция возвращает лишь один элемент и всегда отбрасывает имена. Она отлично подходит для тех случаев, когда вы хотите ясно указать, что извлекаете одиночный элемент, как, например, в цикле `for`. Различие между функциями `[` и `[[` играет очень важную роль в списках, к рассмотрению которых мы сейчас перейдем.

Упражнения

1. Что говорит результат выполнения функции `mean(is.na(x))` о векторе `x`? А что насчет функции `sum(!is.finite(x))`?
2. Внимательно прочитайте документацию по функции `is.vector()`. Что она фактически тестирует? Почему функция `is.atomic()` не согласуется с приведенным выше определением атомарных векторов?
3. Сравните функции `setNames()` и `purrr::set_names()`.
4. Создайте функции, которые принимают вектор в качестве входного значения, и возвращают:

   а) последнее значение (какую функцию следует использовать: `[` или `[[`?);

   б) элементы, занимающие четные позиции;

в) все элементы, за исключением последнего значения;

г) только четные числа (отсутствующие значения не должны включаться).

5. Почему `x[-which(x > 0)]` не то же самое, что `x[x <= 0]`?

6. Что произойдет при попытке извлечения элементов с помощью положительного числа, превышающего длину вектора? Что произойдет при попытке извлечения элемента с несуществующим именем?

Рекурсивные векторы (списки)

Списки — это следующая по сложности ступень по сравнению с атомарными векторами, поскольку они могут содержать другие списки. Благодаря этому списки очень удобны для представления иерархических или древовидных структур. Списки создаются с помощью функции `list()`.

```
x <- list(1, 2, 3)
x
#> [[1]]

#> [1] 1
#>
#> [[2]]
#> [1] 2
#>
#> [[3]]
#> [1] 3
```

Весьма удобным инструментом для работы со списками является функция `str()`, которая фокусируется на структуре, а не на содержимом.

```
str(x)
#> List of 3
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3

x_named <- list(a = 1, b = 2, c = 3)
str(x_named)
#> List of 3
#>  $ a: num 1
#>  $ b: num 2
#>  $ c: num 3
```

В отличие от атомарных векторов функция `list()` может содержать смесь объектов.

```
y <- list("a", 1L, 1.5, TRUE)
str(y)
#> List of 4
#>  $ : chr "a"
#>  $ : int 1
#>  $ : num 1.5
#>  $ : logi TRUE
```

Списки могут даже содержать другие списки!

```
z <- list(list(1, 2), list(3, 4))
str(z)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ :List of 2
#>   ..$ : num 3
#>   ..$ : num 4
```

Визуализация списков

Для объяснения работы более сложных функций, манипулирующих списками, удобно использовать визуальное представление. Возьмем, например, следующие три списка (рис. 16.2).

```
x1 <- list(c(1, 2), c(3, 4))
x2 <- list(list(1, 2), list(3, 4))
x3 <- list(1, list(2, list(3)))
```

Мы представляем их следующим образом.

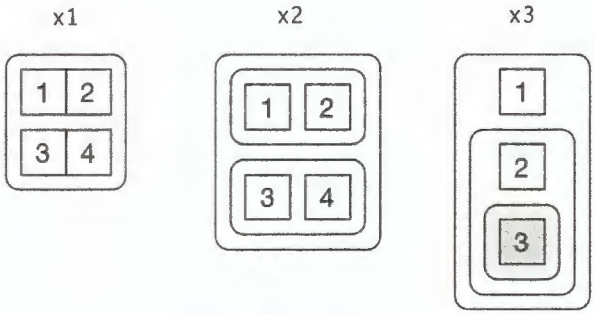


Рис. 16.2

Это представление интерпретируется в соответствии со следующими правилами.

- Спискам соответствуют прямоугольники со скругленными углами. В прямоугольниках, соответствующих атомарным векторам, углы не скруглены.
- Дочерние элементы отображаются в родительских и имеют несколько более темный фон, подчеркивающий структурную иерархию.
- Ориентация дочерних элементов (т.е. строк и столбцов) не имеет значения, поэтому мы выбираем ее такой, чтобы либо сэкономить место на рисунке, либо наилучшим образом проиллюстрировать важное свойство.

Извлечение элементов

Есть три способа извлечения элементов списка, которые мы проиллюстрируем на примере списка `a`.

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))
```

- `[` — извлекает подчиненный список. Результат всегда является списком.

  ```
  str(a[1:2])
  #> List of 2
  #>  $ a: int [1:3] 1 2 3
  #>  $ b: chr "a string"
  str(a[4])
  #> List of 1
  #>  $ d:List of 2
  #>   ..$ : num -1
  #>   ..$ : num -5
  ```

  Как и в случае векторов, извлечение элементов списка может осуществляться с помощью логического, целочисленного или символьного вектора.

- `[[` — извлекает одиночный компонент списка. Эта функция удаляет один уровень иерархии из списка.

  ```
  str(y[[1]])
  #>  chr "a"
  str(y[[4]])
  #>  logi TRUE
  ```

- `$` — сокращение для извлечения именованных элементов списка. Работает аналогично функции `[[`, но не требует использования кавычек.

```
a$a
#> [1] 1 2 3
a[["a"]]
#> [1] 1 2 3
```

В случае списков различие между функциями `[` и `[[` играет действительно важную роль, поскольку функция `[[` проникает в список, тогда как функция `[` возвращает новый список меньшего размера. Предыдущий код и его вывод проиллюстрированы на рис. 16.3.

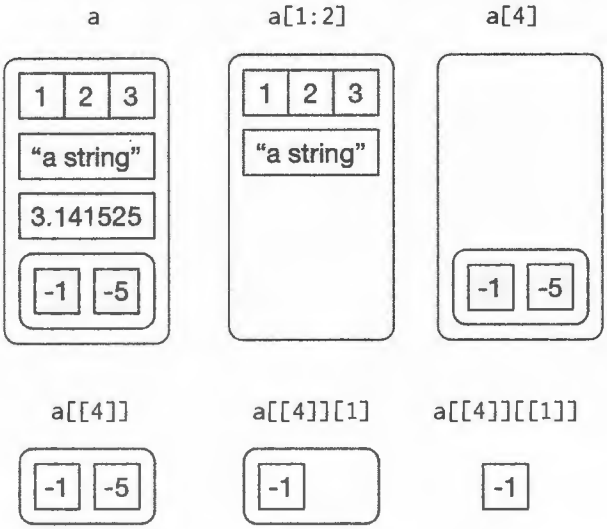


Рис. 16.3

Список специй

Различие между `[` и `[[` очень важно, но в нем легко запутаться. Чтобы помочь вам в этом разобраться, позвольте продемонстрировать вам необычную перечницу (рис. 16.4).

Если уподобить ваш список `x` этой перечнице, то `x[1]` — это перечница, содержащая один пакетик перца (рис. 16.5).

Элемент `x[2]` будет выглядеть точно так же, но будет содержать второй пакет.

Рис. 16.4

Рис. 16.5

Выражение `x[1:2]` будет представлять перечницу, содержащую два пакета перца.

Результат выражения `x[[1]]` представлен на рис. 16.6.

Рис. 16.6

Если хотите получить содержимое пакета, вам нужен элемент `x[[1]][[1]]` (рис. 16.7).

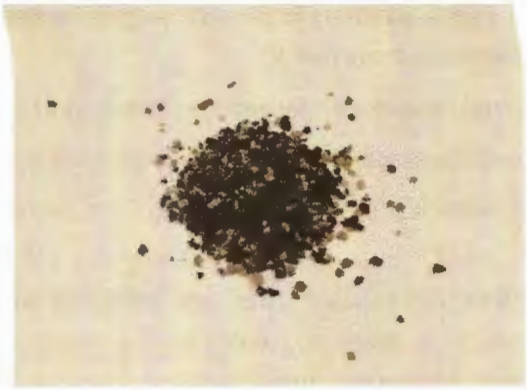


Рис. 16.7

Упражнения

1. Нарисуйте графическое представление списков как вложенных наборов:
   а) `list(a, b, list(c, d), list(e, f))`
   б) `list(list(list(list(list(list(a))))))`
2. Что произойдет, если вы попытаетесь извлекать компоненты tibble-фрейма так, как если бы это был список?
3. В чем заключаются основные различия между списком и tibble-фреймом?

Атрибуты

Любой вектор может содержать произвольные дополнительные метаданные, которые хранятся в атрибутах. Можете рассматривать атрибуты как именованные списки векторов, которые могут присоединяться к любому объекту. Значения отдельных атрибутов можно получать и устанавливать с помощью функции `attr()` или просматривать все сразу с помощью функции `attributes()`.

```
x <- 1:10
attr(x, "приветствие")
#> NULL
attr(x, "приветствие") <- "Привет!"
attr(x, "прощание") <- "Пока!"
attributes(x)
#> $greeting
#> [1] "Привет!"
#>
#> $farewell
#> [1] "Пока!"
```

Существуют три очень важных атрибута, которые используются для реализации фундаментальных понятий R:

- *имена* используются для именования элементов вектора;
- *размерности* обеспечивают поведение вектора как матрицы или массива;
- *классы* используются для реализации объектно-ориентированной системы S3.

До этого вы уже сталкивались с именами, поэтому мы не будем их здесь рассматривать, а вместе с ними и размерности, поскольку не используем матрицы в этой книге. Остается описать лишь классы, которые управляют работой *обобщенных функций*. Последние служат ключом к пониманию объектно-ориентированного программирования в R, поскольку обеспечивают различное поведение для различных классов входных данных. Детальное обсуждение объектно-ориентированного программирования выходит за рамки данной книги, но более подробно об этом можно прочитать в книге *Advanced R*.

Типичная обобщенная функция выглядит примерно так.

```
as.Date
#> function (x, ...)
#> UseMethod("as.Date")
#> <bytecode: 0x7fa61e0590d8>
#> <environment: namespace:base>
```

Вызов "UseMethod" означает, что это обобщенная функция и что она будет вызывать определенный *метод*, т.е. функцию, в зависимости от класса первого аргумента. (Все методы являются функциями, но не все функции являются методами.) Список всех методов обобщенной функции можно получить с помощью функции `methods()`.

```
methods("as.Date")
#> [1] as.Date.character     as.Date.date      as.Date.dates
#> [4] as.Date.default     as.Date.factor    as.Date.numeric
#> [7] as.Date.POSIXct    as.Date.POSIXlt
#> see '?methods' for accessing help and source code
```

Например, если `x` — символьный вектор, то `as.Date()` вызовет функцию `as.Date.character()`; если же это фактор, то будет вызвана функция `as.Date.factor()`.

Конкретную реализацию метода можно увидеть с помощью функции `getS3method()`.

```
getS3method("as.Date", "default")
#> function (x, ...)
#> {
#>     if (inherits(x, "Date"))
#>         return(x)
#>     if (is.logical(x) && all(is.na(x)))
#>         return(structure(as.numeric(x), class = "Date"))
#>     stop(
#>       gettextf("do not know how to convert '%s' to class %s",
#>       deparse(substitute(x)), dQuote("Date")), domain = NA)
#> }
#> <bytecode: 0x7fa61dd47e78>
#> <environment: namespace:base>

getS3method("as.Date", "numeric")
#> function (x, origin, ...)
#> {
#>     if (missing(origin))
#>         stop("'origin' must be supplied")
#>     as.Date(origin, ...) + x
#> }
#> <bytecode: 0x7fa61dd463b8>
#> <environment: namespace:base>
```

Наиболее важная из обобщенных функций S3 — `print()`: она управляет тем, как объект выводится на печать, когда вы вводите его имя в консоли. К числу других важных обобщенных функций относятся функции извлечения элементов `[`, `[[` и `$`.

Расширенные векторы

Атомарные векторы и списки выступают в качестве строительных кирпичиков для других векторных типов, таких как факторы и даты. Мы называем их *расширенными векторами*, поскольку они являются векторами с дополнительными атрибутами, включая класс. В связи с тем что расширенные векторы имеют класс, они ведут себя иначе по сравнению с атомарными векторами, поверх которых они строятся. В этой книге мы используем четыре важных расширенных вектора:

- факторы;
- объекты даты/времени и даты;
- tibble-фреймы.

Все они описаны ниже.

Факторы

Факторы предназначены для представления категориальных данных, которые могут принимать ряд фиксированных возможных значений. Факторы создаются поверх целых чисел и имеют атрибут `levels`.

```
x <- factor(c("ab", "cd", "ab"), levels = c("ab", "cd", "ef"))
typeof(x)
#> [1] "integer"
attributes(x)
#> $levels
#> [1] "ab" "cd" "ef"
#>
#> $class
#> [1] "factor"
```

Дата и дата/время

В R объекты даты — это числовые векторы, которые представляют количество дней, истекших с 1 января 1970 года.

```
x <- as.Date("1971-01-01")
unclass(x)
#> [1] 365

typeof(x)
#> [1] "double"
attributes(x)
#> $class
#> [1] "Date"
```

Объекты даты/времени — это числовые векторы с классом `POSIXct`, которые представляют количество секунд, истекших с 1 января 1970 года. (Если вам интересно, "POSIXct" — это сокращение от "Portable Operating System Interface calendar time" — календарное время, соответствующее стандарту POSIX.)

```
x <- lubridate::ymd_hm("1970-01-01 01:00")
unclass(x)
#> [1] 3600
#> attr(,"tzone")
#> [1] "UTC"
typeof(x)
#> [1] "double"
attributes(x)
#> $tzone
#> [1] "UTC"
```

```
#>
#> $class
#> [1] "POSIXct" "POSIXt"
```

Атрибут `tzone` — необязательный. Он управляет способом вывода времени на печать, а не абсолютным временем, на что указывает его название.

```
attr(x, "tzone") <- "US/Pacific"
x
#> [1] "1969-12-31 17:00:00 PST"

attr(x, "tzone") <- "US/Eastern"
x
#> [1] "1969-12-31 20:00:00 EST"
```

Существует другой тип объектов даты/времени, который называется `POSIXlt`. Эти объекты создаются на основе именованных списков.

```
y <- as.POSIXlt(x)
typeof(y)
#> [1] "list"
attributes(y)
#> $names
#>  [1] "sec"   "min"   "hour"   "mday"   "mon"   "year"
#>  [7] "wday"  "yday"  "isdst"  "zone"   "gmtoff"
#>
#> $class
#> [1] "POSIXlt" "POSIXt"
#>
#> $tzone
#> [1] "US/Eastern" "EST"       "EDT"
```

В библиотеке *tidyverse* объекты `POSIXlt` используются лишь в редких случаях. Их можно найти в базовых пакетах R, потому что они нужны для извлечения таких конкретных компонентов даты, как год или месяц. Поскольку пакет **lubridate** предоставляет для этого вспомогательные функции, объекты `POSIXlt` вам не нужны. С объектами `POSIXct` работать всегда легче, и поэтому, если вы обнаружите, что у вас имеются объекты `POSIXlt`, всегда преобразовывайте их в обычные объекты даты/времени с помощью функции `lubridate::as_date_time()`.

Tibble-фреймы

Tibble-фреймы — это расширенные списки. Они имеют три класса, `tbl_df`, `tbl` и `data.frame`, а также два атрибута: `names` (имена столбцов) и `row.names` (имена строк).

```
tb <- tibble::tibble(x = 1:5, y = 5:1)
typeof(tb)
#> [1] "list"
attributes(tb)
#> $names
#> [1] "x" "y"
#>
#> $class
#> [1] "tbl_df"       "tbl"        "data.frame"
#>
#> $row.names
#> [1] 1 2 3 4 5
```

Традиционный тип `data.frame` имеет аналогичную структуру.

```
df <- data.frame(x = 1:5, y = 5:1)
typeof(df)
#> [1] "list"
attributes(df)
#> $names
#> [1] "x" "y"
#>
#> $row.names
#> [1] 1 2 3 4 5
#>
#> $class
#> [1] "data.frame"
```

Основным отличием является класс. Класс tibble-фреймов включает "data.frame," а это означает, что tibble-фреймы наследуют поведение обычных фреймов данных по умолчанию.

Различие между tibble-фреймом или фреймом данных и списком заключается в том, что все элементы tibble-фрейма или фрейма данных должны быть векторами одинаковой длины. Все функции, которые работают с tibble-фреймами, устанавливают это ограничение.

Упражнения

1. Что возвращает функция `hms::hms(3600)`? Как возвращаемое значение выводится на печать? Поверх какого примитивного типа строится расширенный вектор? Какие атрибуты он использует?
2. Попытайтесь создать tibble-фрейм, столбцы которого имеют разную длину. Что при этом произойдет?
3. Исходя из приведенного ранее определения, может ли столбец tibble-фрейма быть списком?

ГЛАВА 17

Итерирование с помощью пакета purrr

Введение

В главе 15 обсуждался вопрос о том, насколько важно уменьшать дублирование кода, создавая функции вместо того, чтобы заниматься копированием и вставкой кода. Снижение дублирования кода обеспечивает три основных преимущества.

- Проследить намерения автора в таком коде гораздо легче, поскольку внимание концентрируется на содержательном коде, а не на его повторяющихся участках.
- Вносить исправления в такой код в случае изменения требований к нему становится гораздо легче. Вместо того чтобы держать в уме все многочисленные места программы, в которые был вставлен скопированный код, достаточно внести изменения только в одном месте.
- Это облегчает устранение ошибок, которые в противном случае могли бы распространиться во многие участки программы.

Одним из решений проблемы дублирования кода являются функции, которые обособляют повторяющиеся фрагменты кода и превращают их в независимые программные единицы, облегчающие многократное использование кода и его обновление. Другим средством, снижающим повторяемость кода, являются *итерации*, позволяющие выполнять одни и те же действия по отношению к различным входным данным, например выполнять одну и ту же операцию над различными столбцами или наборами данных. В этой главе вы познакомитесь с двумя важными парадигмами итерирования: императивным

программированием и функциональным программированием. В императивном программировании имеются такие инструменты, как циклы `for` и `while`, с которых удобно начинать, поскольку итерации в них представляются явным образом, делая очевидным смысл того, что происходит в коде. Но при этом циклы `for` приходится описывать достаточно подробно, используя довольно большой объем вспомогательного кода, который повторяется для каждого цикла. Функциональное программирование предлагает средства изоляции этого повторяющегося кода, так что каждый шаблон обычного цикла `for` получает собственную функцию. Как только вы освоите концепцию функционального программирования, вам будет гораздо проще решать многие итерационные задачи, причем с меньшей вероятностью совершения ошибок.

Необходимые ресурсы

После того как вы изучите циклы `for`, предоставляемые базовыми пакетами R, вы познакомитесь с несколькими мощными программными инструментами, предлагаемыми пакетом **purrr**, который является частью ядра библиотеки *tidyverse*.

```
library(tidyverse)
```

Циклы `for`

Предположим, имеется следующий простой tibble-фрейм.

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Мы хотим вычислить медиану для каждого столбца. Это можно сделать, используя метод копирования и вставки.

```
median(df$a)
#> [1] -0.246
median(df$b)
#> [1] -0.287
median(df$c)
#> [1] -0.0567
median(df$d)
#> [1] 0.144
```

Но это противоречит нашему правилу: не использовать копирование и вставку более двух раз. Поэтому лучше использовать цикл `for`.

```
output <- vector("double", ncol(df)) # 1. вывод
for (i in seq_along(df)) {           # 2. последовательность
  output[[i]] <- median(df[[i]])     # 3. тело
}
output
#> [1] -0.2458 -0.2873 -0.0567  0.1443
```

Каждый цикл `for` содержит три компонента.

вывод `output <- vector("double", length(x))`

Прежде чем выполнять цикл, всегда необходимо выделить достаточно места для хранения выходных результатов. Это очень важно для обеспечения эффективности вычислений: если наращивать выходные результаты цикла `for` на каждой итерации с помощью (например) вызовов `c()`, то цикл будет работать очень медленно.

Общий способ создания пустого вектора заданной длины предлагает функция `vector()`. Она имеет два аргумента: тип вектора ("логический", "целочисленный" "вещественный", "символьный" и т.п.) и длина вектора.

последовательность `i in seq_along(df)`

Этот компонент определяет, по какой переменной выполняется итерирование: на каждой итерации цикла переменной `i` будет присваиваться другое значение, берущееся из последовательности `seq_along(df)`.

Возможно, ранее вы не сталкивались с функцией `seq_along()`. Это безопасная версия знакомого вам выражения `1:length(l)` с тем важным отличием, что в случае вектора нулевой длины функция `seq_along()` делает все правильно.

```
y <- vector("double", 0)
seq_along(y)
#> integer(0)
1:length(y)
#> [1] 1 0
```

Скорее всего, вы не будете намеренно создавать векторы нулевой длины, но это может легко получиться у вас случайно. Если вместо функции `seq_along(x)` использовать выражение `1:length(x)`, то, вероятно, вы получите сообщение об ошибке.

тело цикла `output[[i]] <- median(df[[i]])`

Это код, который делает всю работу. Он выполняется многократно, каждый раз с другим значением `i`. На первой итерации будет выполняться инструкция `output[[1]] <- median(df[[1]])`, на второй — `output[[2]] <- median(df[[2]])` и т.д.

Вот и все, что нужно знать о цикле `for`! Теперь вам следует попрактиковаться в создании простых (и не совсем простых) циклов `for`, воспользовавшись приведенными ниже упражнениями. После этого мы перейдем к рассмотрению видоизмененных версий цикла `for`, с помощью которых вы сможете решать задачи, возникающие на практике.

Упражнения

1. Напишите циклы для решения перечисленных ниже задач:
   а) вычислить среднее значение для каждого из столбцов таблицы `mtcars`;
   б) определить тип каждого столбца таблицы `nycflights13::flights`;
   в) вычислить количество уникальных значений в каждом столбце таблицы `iris`.
   г) сгенерировать 10 случайных нормальных распределений для каждого из значений $\mu = -10$, 0, 10 и 100.

   Прежде чем писать цикл, хорошенько обдумайте последовательность и тело цикла.

2. Устраните циклы `for` в каждом из приведенных ниже примеров, воспользовавшись существующей функцией, которая работает с векторами.

```
out <- ""
for (x in letters) {
  out <- stringr::str_c(out, x)
}

x <- sample(100)
sd <- 0
for (i in seq_along(x)) {
  sd <- sd + (x[i] - mean(x)) ^ 2
}
sd <- sqrt(sd / (length(x) - 1))

x <- runif(100)
out <- vector("numeric", length(x))
```

```
out[1] <- x[1]
for (i in 2:length(x)) {
  out[i] <- out[i - 1] + x[i]
}
```

3. Используйте свои навыки написания функций и создания циклов `for` для решения перечисленных ниже задач.

   А. Напишите цикл `for`, который выводит с помощью функции `prints()` текст детской песенки-считалочки "Alice the Camel" (`http://www.metrolyrics.com/alice-the-camel-lyrics-children.html`).

   Б. Преобразуйте текст детской песенки-считалочки "Ten in the Bed" (`https://www.kididdles.com/lyrics/t003.html`) в функцию. Обобщите эту функцию на случай любого количества людей и спального места любого типа.

   В. Преобразуйте текст детской песенки-считалочки "99 Bottles of Beer on the Wall" (`https://en.wikipedia.org/wiki/99_Bottles_of_Beer`) в функцию. Обобщите эту функцию на любое количество сосудов, содержащих любую жидкость и находящихся на любой поверхности.

4. Часто можно встретить циклы `for`, в которых не предусмотрена переменная для хранения выходных результатов, а вместо этого на каждой итерации цикла увеличивается длина вектора.

```
output <- vector("integer", 0)
for (i in seq_along(x)) {
  output <- c(output, lengths(x[[i]]))
}
output
```

   Как это влияет на производительность? Спланируйте и проведите эксперимент.

Варианты цикла `for`

Теперь, когда вы освоили простейший цикл `for`, вам следует изучить его видоизмененные варианты. Вам важно их знать не только потому, что они позволяют по-разному организовывать циклы, так что не забудьте о них, когда овладеете методиками функционального программирования, о которых вы узнаете в следующем разделе.

Существуют следующие четыре вариации на тему цикла `for`:

- изменение существующего объекта вместо создания нового;
- циклический перебор имен или значений, а не индексов;
- обработка выходных данных, длина которых неизвестна;
- обработка последовательностей, длина которых неизвестна.

Изменение существующего объекта

Иногда возникает необходимость в использовании цикла для изменения существующего объекта. Вспомните, например, задачу из главы 15. Мы хотели масштабировать все столбцы фрейма данных.

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

df$a <- rescale01(df$a)
df$b <- rescale01(df$b)
df$c <- rescale01(df$c)
df$d <- rescale01(df$d)
```

Чтобы решить эту задачу с помощью цикла `for`, вновь обратимся к трем его компонентам.

Вывод

У нас уже есть выходной объект — это тот же объект, что и входной!

Последовательность

Фрейм данных можно представлять как список столбцов, поэтому мы можем итерировать по столбцам с помощью функции `seq_along(df)`.

Тело цикла

Примените функцию `rescale01()`.

В результате получаем следующий код.

```
for (i in seq_along(df)) {
  df[[i]] <- rescale01(df[[i]])
}
```

Как правило, именно эту разновидность цикла вы будете применять для изменения списка или фрейма данных, поэтому не забудьте использовать функцию `[[`, а не `[`. Возможно, вы заметили, что в своих циклах мы использовали функцию `[[`: мы считаем, что ее лучше использовать даже в случае атомарных векторов, поскольку это отчетливо указывает на то, что мы хотим работать с одиночным элементом.

Шаблоны формирования циклов

Существуют три способа организации цикла для вектора. До сих пор мы демонстрировали вам наиболее общий способ: организация цикла `for` по числовым индексам (`i in seq_along(xs)`) и извлечение значения с помощью выражения `x[[i]]`. Кроме этой, существуют две другие формы, описанные ниже.

- Циклический перебор по элементам: `for (x in xs)`. Эта форма наиболее полезна в тех случаях, когда вы заинтересованы лишь в побочных эффектах, таких как вывод данных на печать или сохранение в файле, ввиду того, что организовать иным способом сохранение выходных данных затруднительно.
- Циклический перебор по именам: `for (nm in names(xs))`. В результате вы получаете имя, которое можно использовать для доступа к значению с помощью выражения `x[[nm]]`. Это полезно в тех случаях, когда вы хотите использовать имя в качестве названия графика или имени файла.

  Создавая именованные выходные данные, присваивайте имя результирующему вектору следующим образом.

  ```
  results <- vec[illegible]("list", [illegible]ngt[illegible](x))
  name[illegible](results) <- names(x)
  ```

Итерирование по числовым индексам — самая общая форма, поскольку, зная позицию элемента, вы сможете извлечь как его имя, так и значение.

```
[illegible] (i in seq_a[illegible]ng(x)) {
  name <- names(x)[[i]]
  value <- x[[i]]
}
```

Выходные данные неизвестной длины

Иногда длина выходных данных неизвестна заранее. Предположим, вы хотите имитировать набор случайных векторов, имеющих случайную длину. У вас может возникнуть соблазн попытаться решить эту задачу путем постепенного наращивания вектора.

```
means <- c(0, 1, 2)

output <- double()
for (i in seq_along(means)) {
  n <- sample(100, 1)
  output <- c(output, rnorm(n, means[[i]]))
}
str(output)
#>  num [1:202] 0.912 0.205 2.584 -0.789 0.588 ...
```

Однако такой подход не очень эффективен, поскольку на каждой итерации R приходится копировать все данные из предыдущих итераций. Говоря техническим языком, вы получаете "квадратичную" сложность $O(n^2)$, а это означает, что цикл, обрабатывающий в три раза больше элементов, будет выполняться в девять (3^2) раз дольше.

Лучшее решение заключается в том, чтобы сохранить результаты в виде списка, а затем объединить их в один вектор по завершении цикла.

```
out <- vector("list", length(means))
for (i in seq_along(means)) {
  n <- sample(100, 1)
  out[[i]] <- rnorm(n, means[[i]])
}
str(out)
#> List of 3
#>  $ : num [1:83] 0.367 1.13 -0.941 0.218 1.415 ...
#>  $ : num [1:21] -0.485 -0.425 2.937 1.688 1.324 ...
#>  $ : num [1:40] 2.34 1.59 2.93 3.84 1.3 ...

str(unlist(out))
#>  num [1:144] 0.367 1.13 -0.941 0.218 1.415 ...
```

В данном случае мы использовали функцию `unlist()` для сведения списка векторов в единый вектор. Более строгий подход заключается в использовании функции `purrr::flatten_dbl()` — в случае, если входные данные не являются списком вещественных чисел, она генерирует ошибку.

Такой шаблон также встречается в следующих двух ситуациях.

- Вы генерируете длинную строку. Вместо объединения результатов каждой итерации с предыдущими с помощью функции `paste()` сохраните выходные результаты в символьном векторе, в затем объедините этот вектор в одну строку посредством вызова `paste(output, collapse = "")`.
- Вы генерируете большой фрейм данных. Вместо последовательного объединения строк или столбцов на каждой итерации с помощью функции `rbind()` сохраните выходные данные в виде списка, а затем используйте вызов `dplyr::bind_rows(output)` для объединения выходных данных в один фрейм данных.

Всякий раз, когда вам встречается этот шаблон, переходите к более сложному объекту результата, а по завершении цикла объедините выходные данные за один прием.

Последовательность неизвестной длины

Иногда вы даже не знаете, какой длины должна быть входная последовательность. Это обычная ситуация при выполнении различных имитаций. Предположим, вы хотите, чтобы цикл выполнялся до тех пор, пока три раза подряд не выпадет "орел". С помощью цикла `for` вы не создадите итерационный цикл такого рода. Цикл `while` проще цикла `for`, поскольку имеет всего лишь два компонента: условие и тело цикла.

```
while (условие) {
  # тело цикла
}
```

Одновременно цикл `while` более общий по сравнению с циклом `for`, поскольку любой цикл `for` можно переписать как цикл `while`, тогда как не каждый цикл `while` может быть переписан как цикл `for`.

```
for (i in seq_along(x)) {
  # тело
}

# эквивалентно следующему
i <- 1
while (i <= length(x)) {
  # тело
  i <- i + 1
}
```

Ниже приведен пример использования цикла `while` для определения того, сколько потребуется попыток, пока три раза подряд не выпадет "орел".

```
flip <- function() sample(c("T", "H"), 1)

flips <- 0
nheads <- 0

while (nheads < 3) {
  if (flip() == "H") {
    nheads <- nheads + 1
  } else {
    nheads <- 0
  }
  flips <- flips + 1
}
flips
#> [1] 3
```

Мы лишь вкратце упомянули о циклах `while`, поскольку применяем их крайне редко. Чаще всего они используются для имитации процессов, но рассмотрение этой темы выходит за рамки данной книги. Однако вы должны знать об их существовании, чтобы чувствовать себя уверенно при решении задач, в которых количество итераций неизвестно заранее.

Упражнения

1. Предположим, в каталоге содержится большое количество CSV-файлов, которые вы хотите прочитать. Пути к ним хранятся в векторе, `files <- dir("data/", pattern = "\\.csv$", full.names = TRUE)`, и теперь нужно прочитать каждый из них с помощью функции `read_csv()`. Напишите цикл `for`, который загрузит их в один фрейм данных.
2. Что произойдет, если вы используете запись `for (nm in names(x))`, но `x` не содержит имен? Что если только некоторые элементы содержат имена? Что если имена не уникальны?
3. Напишите функцию, которая выводит среднее каждого числового столбца фрейма данных вместе с его именем. Например, вызов `show_mean(iris)` должен обеспечить такой вывод.

```
show_mean(iris)
#> Sepal.Length: 5.84
#> Sepal.Width:  3.06
```

```
#> Petal.Length: 3.76
#> Petal.Width:  1.20
```

(Дополнительное задание: какую функцию мы использовали для того, чтобы числа были выровнены, несмотря на различия в длине имен переменных?)

4. Что делает приведенный ниже код? Как он работает?

```
trans <- list(
  disp = function(x) x * 0.0163871,
  am = function(x) {
    factor(x, labels = c("auto", "manual"))
  }
)
for (var in names(trans)) {
  mtcars[[var]] <- trans[[var]](mtcars[[var]])
}
```

Циклы for и функционалы

В R циклы for не имеют такого значения, как в других языках, поскольку R — язык функционального программирования. Это означает, что вместо непосредственного использования цикла for его можно обернуть в функцию и вызвать ее.

Чтобы увидеть, почему это важно, рассмотрим (вновь) следующий простой фрейм данных.

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

Предположим, вы хотите вычислить среднее значение для каждого столбца. Это можно было бы сделать с помощью цикла for.

```
output <- vector("double", length(df))
for (i in seq_along(df)) {
  output[[i]] <- mean(df[[i]])
}
output
#> [1]  0.2026 -0.2068  0.1275 -0.0917
```

Вы осознаете, что с задачей вычисления среднего значения для каждого столбца вам придется сталкиваться довольно часто, и принимаете решение обернуть цикл в функцию.

```
col_mean <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- mean(df[[i]])
  }
  output
}
```

Но затем вы догадываетесь, что было бы неплохо вычислять таким же способом медиану и стандартное отклонение, и в связи с этим создаете копии функции `col_mean()`, в которых заменяете имя функции `mean` именами `median` и `sd`.

```
col_median <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- median(df[[i]])
  }
  output
}
col_sd <- function(df) {
  output <- vector("double", length(df))
  for (i in seq_along(df)) {
    output[i] <- sd(df[[i]])
  }
  output
}
```

Однако при этом вам пришлось дважды скопировать и вставить один и тот же код, а это означает, что самый раз подумать о создании обобщенного варианта функции. Обратите внимание на тот факт, что бо́льшая часть этого кода представляет собой заготовку для цикла `for`, и заметить в нем то, что отличает один вариант от другого (имена `mean`, `median` и `sd`), не так-то просто.

А какие мысли пришли бы вам в голову, если бы вы увидели набор функций, подобный приведенному ниже?

```
f1 <- function(x) abs(x - mean(x)) ^ 1
f2 <- function(x) abs(x - mean(x)) ^ 2
f3 <- function(x) abs(x - mean(x)) ^ 3
```

Надеюсь, вы заметили бы здесь повторяющийся код и вынесли его в дополнительный аргумент:

```
f <- function(x, i) abs(x - mean(x)) ^ i
```

Тем самым вы снизили бы вероятность возникновения ошибок (поскольку объем кода уменьшился в три раза) и облегчили его обобщение на другие ситуации.

Но то же самое мы можем сделать и с функциями `col_mean()`, `col_median()` и `col_sd()`, добавив аргумент, предоставляющий функцию, которая должна быть применена к каждому столбцу.

```
col_summary <- function(df, fun) {
  out <- vector("double", length(df))
  for (i in seq_along(df)) {
    out[i] <- fun(df[[i]])
  }
  out
}
col_summary(df, median)
#> [1]  0.237 -0.218  0.254 -0.133
col_summary(df, mean)
#> [1]  0.2026 -0.2068  0.1275 -0.0917
```

Идея передачи одной функции в качестве аргумента другой — необычайно мощная, и это один из вариантов поведения, которые делают R языком функционального программирования. Возможно, вам потребуется некоторое время для того, чтобы осмыслить ее, но это ценная инвестиция в ваше будущее. Оставшаяся часть главы посвящена изучению и применению пакета **purrr**, предоставляющего функции, которые устраняют необходимость в использовании многих распространенных циклов `for`. Для решения аналогичной проблемы предназначены функции семейства `apply`, предоставляемые базовыми пакетами R (`apply()`, `lapply()`, `tapply()` и т.п.), но пакет **purrr** более последователен, и поэтому его легче изучить.

Целью использования функций пакета **purrr** вместо циклов `for` является получение возможности разбивать задачи, включающие распространенные операции манипуляции списками, на независимые части. При этом вы должны задавать себе следующие вопросы.

- Как решить задачу для одиночного элемента списка? Как только эта задача будет решена, **purrr** позаботится об обобщении вашего решения на все элементы списка.
- Если вы решаете сложную задачу, можете ли вы разбить ее на меньшие части, последовательное решение которых приближает вас к цели?

Используя **purrr**, вы можете получить ряд частичных решений, которые можно объединить в одно целое с помощью канала.

Такой подход упрощает решение новых задач. Он также упрощает понимание ваших решений прежних задач, когда вы перечитываете разработанный ранее код.

Упражнения

1. Прочитайте документацию по функции `apply()`. Какие два цикла обобщает эта функция во втором примере?
2. Адаптируйте функцию `col_summary()` таким образом, чтобы она применялась только к числовым столбцам. Возможно, вы захотите начать с использования функции `is_numeric()`, возвращающей логический вектор, в котором значения `TRUE` соответствуют числовым столбцам.

Функции семейства map

Организация цикла по вектору, выполнение определенного набора действий по отношению к каждому элементу вектора и сохранение результатов является настолько типичной задачей, что пакет **purrr** предоставляет семейство функций, выполняющих все это вместо вас. Для каждого типа выходных результатов предусмотрена своя функция:

- `map()` — создает список;
- `map_lgl()` — создает логический вектор;
- `map_int()` — создает целочисленный вектор;
- `map_dbl()` — создает вещественный вектор;
- `map_chr()` — создает символьный вектор.

Каждая из этих функций принимает вектор в качестве входных данных, применяет функцию к каждой части и возвращает новый вектор той же длины (и с теми же именами), что и входной. Тип вектора определяется суффиксом в имени функции.

Освоив эти функции, вы увидите, что теперь на решение итерационных задач у вас уходит меньше времени. Однако никогда не стесняйтесь использовать цикл вместо функции отображения. Такие функции добавляют новые уровни абстракции, и должно пройти какое-то время, прежде чем вы освоите

работу с ними. Важно то, чтобы ваши усилия были направлены на решение задачи, над которой вы работаете, а не на написание наиболее компактного и элегантного кода. (Хотя, несомненно, это то, к чему вы всегда должны стремиться!)

Кое-кто может советовать вам избегать циклов из-за того, что они медленно работают. Они неправы! (Или, по крайней мере, отстали от жизни, поскольку циклы не работают медленно вот уже много лет.) Основным преимуществом функций наподобие `map()` является не скорость, а ясность: использующий их код легко читать и писать.

Описанные функции можно использовать для выполнения тех же вычислений, что и в последнем цикле `for`. Эти итоговые функции возвращают вещественные числа, поэтому мы должны использовать функцию `map_dbl()`.

```
map_dbl(df, mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
map_dbl(df, median)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
map_dbl(df, sd)
#>     a     b     c     d
#> 0.796 0.759 1.164 1.062
```

По сравнению с циклом `for` внимание здесь фокусируется на выполняемой операции (т.е. `mean()`, `median()` `sd()`), а не на вспомогательных действиях по обслуживанию цикла и сохранению результата. Это становится еще более очевидным, если использовать канал.

```
df %>% map_dbl(mean)
#>       a       b       c       d
#>  0.2026 -0.2068  0.1275 -0.0917
df %>% map_dbl(median)
#>       a       b       c       d
#>  0.237 -0.218  0.254 -0.133
df %>% map_dbl(sd)
#>       a       b       c       d
#> 0.796 0.759 1.164 1.062
```

Между функциями `map_*()` и `col_summary()` существует еще несколько различий.

- Все функции пакета **purrr** реализованы на языке C. Это несколько ускоряет их работу за счет некоторого снижения удобочитаемости.

- Второй аргумент, `.f`, т.е. применяемая функция, может быть формулой, а также символьным или целочисленным вектором. Об этих удобных сокращениях вы узнаете в следующем разделе.
- Для передачи дополнительных аргументов функции `.f` при каждом ее вызове в функциях семейства `map_*()` предусмотрен аргумент-многоточие `...` (см. главу 15).

```
map_dbl(df, mean, trim = 0.5)
#>      a      b      c      d
#>  0.237 -0.218  0.254 -0.133
```

- Кроме того, функции отображения сохраняют имена.

```
z <- list(x = 1:3, y = 4:5)
map_int(z, length)
#> x y
#> 3 2
```

Сокращения

Существуют определенные сокращения, которые можно использовать с функцией `.f` для того, чтобы немного уменьшить объем вводимого кода. Предположим, вы хотите применить линейную модель к каждой группе набора данных. Ниже представлен небольшой пример, разбивающий набор данных `mtcars` на три части (по одной для каждого значения переменной `cylinder`) и применяющий одну и ту же линейную модель к каждой из них.

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(function(df) lm(mpg ~ wt, data = df))
```

Синтаксис для создания анонимных функций в R довольно подробен, поэтому пакет **purrr** предоставляет удобное сокращение: одностороннюю формулу.

```
models <- mtcars %>%
  split(.$cyl) %>%
  map(~lm(mpg ~ wt, data = .))
```

Здесь мы использовали символ точки (`.`) в качестве местоимения: он ссылается на текущий элемент списка (точно так же, как `i` ссылается на текущий индекс в цикле `for`).

В случае работы одновременно со многими моделями целесообразно использовать такие итоговые статистики, как коэффициент детерминации R^2.

Для этого необходимо вызвать функцию `summary()`, а затем извлечь компонент под названием `r.squared`. Это можно было бы сделать с помощью сокращений для анонимных функций.

```
models %>%
  map(summary) %>%
  map_dbl(~.$r.squared)
#>     4     6     8
#> 0.509 0.465 0.423
```

Однако извлечение именованных компонентов — распространенная операция, поэтому пакет **purrr** предоставляет еще более экономное сокращение: строку.

```
models %>%
  map(summary) %>%
  map_dbl("r.squared")
#>     4     6     8
#> 0.509 0.465 0.423
```

Также можно использовать целые числа для выбора элементов по их позициям.

```
x <- list(list(1, 2, 3), list(4, 5, 6), list(7, 8, 9))
x %>% map_dbl(2)
#> [1] 2 5 8
```

Базовые возможности R

Если вы знакомы с функциями семейства `apply`, входящими в базовый комплект R, то могли обратить внимание на их сходство с функциями пакета **purrr**.

- Функция `lapply()` в основном идентична функции `map()`, за исключением того, что `map()` согласуется со всеми остальными функциями **purrr** и допускает использование сокращений для функции `.f`.
- Базовая функция `sapply()` служит оболочкой для функции `lapply()`, автоматически упрощая вывод. Это удобно для работы в интерактивном режиме, но проблематично для использования в функциях, поскольку вы никогда не знаете заранее, какой тип вывода будет получен.

  ```
  x1 <- list(
    c(0.27, 0.37, 0.57, 0.91, 0.20),
    c(0.90, 0.94, 0.66, 0.63, 0.06),
    c(0.21, 0.18, 0.69, 0.38, 0.77)
  ```

```
)
x2 <- list(
  c(0.50, 0.72, 0.99, 0.38, 0.78),
  c(0.93, 0.21, 0.65, 0.13, 0.27),
  c(0.39, 0.01, 0.38, 0.87, 0.34)
)

threshold <- function(x, cutoff = 0.8) x[x > cutoff]
x1 %>% sapply(threshold) %>% str()
#> List of 3
#>  $ : num 0.91
#>  $ : num [1:2] 0.9 0.94
#>  $ : num(0)
x2 %>% sapply(threshold) %>% str()
#>  num [1:3] 0.99 0.93 0.87
```

- Функция `vapply()` — это безопасная альтернатива функции `sapply()`, поскольку позволяет предоставлять дополнительный аргумент, который определяет тип. Единственной проблемой, связанной с функцией `vapply()`, является длина ввода: вызов `vapply(df, is.numeric,logical(1))` эквивалентен вызову `map_lgl(df, is.numeric)`. Функция `vapply()` обладает одним преимуществом по сравнению с функциями семейства `map`: она также может создавать матрицы, тогда как функции `map` всегда создают только векторы.

В этом разделе мы сфокусировали ваше внимание на функциях пакета **purrr** ввиду лучшего согласования их имен и аргументов, наличия полезных сокращенных способов вызова, а также ожидаемого в ближайшем будущем упрощения работы с параллельными вычислениями и предоставления индикаторов выполнения.

Упражнения

1. Напишите код, в котором функции семейства `map` используются для решения следующих задач:

   а) вычислите среднее значение для каждого столбца набора данных `mtcars`;

   б) определите тип каждого столбца набора данных `nycflights13::flights`;

   в) вычислите количество уникальных значений в каждом столбце набора данных `iris`;

г) сгенерируйте 10 случайных нормальных распределений для каждого из значений параметра $\mu = -10$, 0, 10 и 100.

2. Как создать одиночный вектор, который указывал бы для каждого столбца фрейма данных, является ли он фактором или нет?
3. Что произойдет, если вы используете функции семейства `map` по отношению к векторам, не являющимся списками? Что делает вызов функции `map(1:5, runif)`? Дайте пояснения.
4. Что делает вызов функции `map(-2:2, rnorm, n = 5)`? Дайте пояснения. Что делает вызов функции `map_dbl(-2:2, rnorm, n = 5)`? Дайте пояснения.
5. Перепишите функцию `map(x, function(df) lm(mpg ~ wt, data = df))` таким образом, чтобы устранить анонимную функцию.

Обработка ошибок

Когда функции семейства `map` используются для многократного выполнения многих операций, существует большая вероятность того, что одна из этих операций не сможет успешно завершиться. Если это случится, вы получите сообщение об ошибке без вывода какой-либо информации. Это раздражает: почему единственная ошибка делает невозможным доступ к результатам успешного выполнения остальных операций? Как добиться того, чтобы ложка дегтя не могла испортить бочку меда?

В этом разделе вы узнаете о том, как обрабатывать подобные ситуации с помощью новой функции: `safely()`. Функция `safely()` — наречие: она принимает функцию (глагол) и возвращает ее модифицированную версию. В данном случае модифицированная функция никогда не сгенерирует ошибку. Вместо этого она всегда возвращает список, состоящий из двух перечисленных ниже элементов.

`result`

Оригинальный результат. В случае ошибки это будет `NULL`.

`error`

Объект ошибки. Если операция завершилась успешно, это будет `NULL`.

(Возможно, вы знакомы с функцией `try()` из базового комплекта R. Она работает аналогично функции `safely()`, но поскольку иногда она возвращает оригинальный результат, а иногда — объект ошибки, с ней труднее работать.)

Проиллюстрируем это на простом примере, используя функцию safe_log().

```
safe_log <- safely(log)
str(safe_log(10))
#> List of 2
#>  $ result: num 2.3
#>  $ error : NULL
str(safe_log("a"))
#> List of 2
#>  $ result: NULL
#>  $ error :List of 2
#>   ..$ message: chr "non-numeric argument to mathematical ..."
#>   ..$ call   : language .f(...)
#>   ..- attr(*, "class")= chr [1:3] "simpleError" "error" ...
```

В случае успешного выполнения функции элемент result содержит результат, а элемент error — значение NULL. В случае аварийного завершения функции элемент result содержит значение NULL, а элемент error — объект ошибки.

Функция safely() способна работать с функциями семейства map.

```
x <- list(1, 10, "a")
y <- x %>% map(safely(log))
str(y)
#> List of 3
#>  $ :List of 2
#>   ..$ result: num 0
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: num 2.3
#>   ..$ error : NULL
#>  $ :List of 2
#>   ..$ result: NULL
#>   ..$ error :List of 2
#>   .. ..$ message: chr "non-numeric argument to ..."
#>   .. ..$ call  : language .f(...)
#>   .. ..- attr(*, "class")=chr [1:3] "simpleError" "error" ...
```

Было бы проще работать, если бы у нас было два списка: один — для всех ошибок, другой — для всех выходных результатов. Их легко получить с помощью функции purrr::transpose().

```
y <- y %>% transpose()
str(y)
#> List of 2
#>  $ result:List of 3
```

```
=>   ..$ : num 0
=>   ..$ : num 2.3
=>   ..$ : NULL
=>  $ error :List of 3
=>   ..$ : NULL
=>   ..$ : NULL
=>   ..$ :List of 2
=>   .. ..$ message: chr "non-numeric argument to ..."
=>   .. ..$ call : language .f(...)
=>   .. ..- attr(*, "class")=chr [1:3] "simpleError" "error" ...
```

Обработка ошибок полностью зависит от вас, но в типичных случаях вы будете либо просматривать значения x, для которых y содержит ошибку, либо работать с нормальными значениями y.

```
is_ok <- y$error %>%  map_lgl(is_null)
x[!is_ok]
#> [[1]]
#> [1] "a"
y$result[is_ok] %>% f...     dbl ()
#> [1] 0.0 2.3
```

Пакет **purrr** предоставляет два других полезных наречия.

- Как и функция safely(), функция possibly() всегда завершается успешно. Она проще функции safely(), поскольку вы передаете ей значение по умолчанию, которое будет возвращено в случае возникновения ошибки.

  ```
  x <- list(1, 10, "a")
  x %>% map_dbl(possibly(log, NA_real_))
  #> [1] 0.0 2.3  NA
  ```

- Функция quietly() играет аналогичную роль, но, в отличие от функции safely(), вместо перехвата ошибок она перехватывает выводимые результаты, сообщения и предупреждения.

  ```
  x <- list(1, -...)
  x %>% map(qui...(log)) %>%  ...()
  #> List of 2
  #>  $ :List of 4
  #>   ..$ result  : num 0
  #>   ..$ output  : chr ""
  #>   ..$ warnings: chr(0)
  #>   ..$ messages: chr(0)
  #>  $ :List of 4
  #>   ..$ result  : num NaN
  ```

```
#>   ..$ output  : chr ""
#>   ..$ warnings: chr "NaNs produced"
#>   ..$ messages: chr(0)
```

Функции семейства map с несколькими входными переменными

До сих пор мы занимались отображениями одной входной переменной. Однако часто возникают ситуации, когда требуется выполнять итерации параллельно по нескольким входным переменным. Эту задачу решают функции map2() и pmap(). Предположим, вы хотите имитировать нормальные распределения с разными средними значениями. Вы знаете, как это можно сделать с помощью функции map().

```
mu <- list(5, 10, -3)
mu %>%
  map(rnorm, n = 5) %>%
  str()
#> List of 3
#>  $ : num [1:5] 5.45 5.5 5.78 6.51 3.18
#>  $ : num [1:5] 10.79 9.03 10.89 10.76 10.65
#>  $ : num [1:5] -3.54 -3.08 -5.01 -3.51 -2.9
```

А как быть в том случае, если вы захотите варьировать также стандартное отклонение? Один из способов сделать это заключается в итерировании по индексам и индексировании векторов средних значений и стандартных отклонений.

```
sigma <- list(1, 5, 10)
seq_along(mu) %>%
  map(~rnorm(5, mu[[.]], sigma[[.]])) %>%
  str()
#> List of 3
#>  $ : num [1:5] 4.94 2.57 4.37 4.12 5.29
#>  $ : num [1:5] 11.72 5.32 11.46 10.24 12.22
#>  $ : num [1:5] 3.68 -6.12 22.24 -7.2 10.37
```

Но это затрудняет понимание назначения кода. Вместо этого можно использовать функцию map2(), выполняющую итерации одновременно по двум векторам (рис. 17.1).

```
map2(mu, sigma, rnorm, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 4.78 5.59 4.93 4.3 4.47
```

```
=>  $ : num [1:5] 10.85 10.57 6.02 8.82 15.93
=>  $ : num [1:5] -1.12 7.39 -7.5 -10.09 -2.7
```

Функция `map2()` генерирует следующую последовательность вызовов функций.

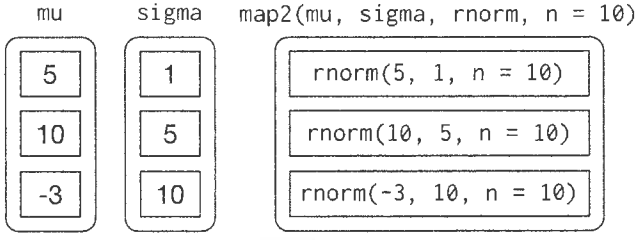


Рис. 17.1

Обратите внимание на то, что аргументы, которые меняются от вызова к вызову, предшествуют функции, тогда как аргументы, остающиеся постоянными для всех вызовов, указываются после нее.

Как и функция `map()`, функция `map2()` — это всего лишь оболочка вокруг цикла `for`.

```
map2 <- function(x, y, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)) {
    out[[i]] <- f(x[[i]], y[[i]], ...)
  }
  out
}
```

Несложно представить себе создание аналогичных функций `map3()`, `map4()`, `map5()`, `map6()` и т.д., но такой подход бытро наскучит. Вместо этого пакет **purrr** предоставляет функцию `pmap()`, которая принимает список аргументов. Вы могли бы использовать ее, если бы, например, хотели варьировать среднее значение, стандартное отклонение и количество выборок.

```
n <- list(1, 3, 5)
args1 <- list(n, mu, sigma)
args1 %>%
  pmap(rnorm) %>%
  str()
#> List of 3
#>  $ : num 4.55
#>  $ : num [1:3] 13.4 18.8 13.2
#>  $ : num [1:5] 0.685 10.801 -11.671 21.363 -2.562
```

Это выглядит примерно так (рис. 17.2).

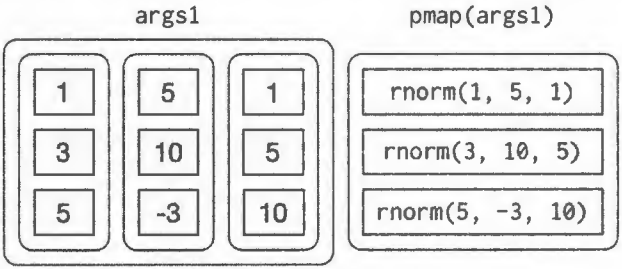


Рис. 17.2

Если элементы списка не поименованы, функция `pmap()` будет использовать при вызове функции позиционное соответствие. Это придает коду определенную хрупкость и затрудняет его чтение, так что лучше присваивать аргументам имена.

```
args2 <- list(mean = mu, sd = sigma, n = n)
args2 %>%
  pmap(rnorm) %>%
  str()
```

Данный код генерирует более длинные, но в то же время более безопасные вызовы (рис. 17.3).

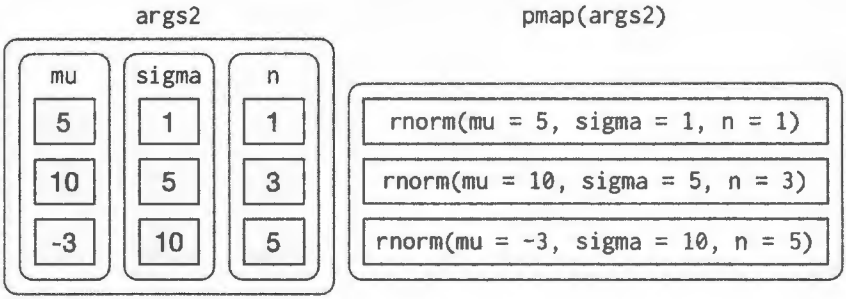


Рис. 17.3

Поскольку все аргументы имеют одинаковую длину, имеет смысл сохранить их во фрейме данных.

```
params <- tribble(
  ~mean, ~sd, ~n,
    5,     1,  1,
   10,     5,  3,
   -3,    10,  5
)
```

```
params %>%
  pmap(rnorm)
#> [[1]]
#> [1] 4.68
#>
#> [[2]]
#> [1] 23.44 12.85  7.28
#>
#> [[3]]
#> [1]  -5.34 -17.66 0.92 6.06 9.02
```

Как только мы видим, что код начинает усложняться, мы стараемся использовать фреймы данных, поскольку они гарантируют, что каждый столбец как-то назван, а все столбцы имеют одинаковую длину.

Вызов различных функций

Возможен еще один шаг в направлении возрастания сложности: наряду с варьированием аргументов функции можно варьировать саму функцию.

```
f <- c("runif", "rnorm", "rpois")
param <- list(
  list(min = -1, max = 1),
  list(sd = 5),
  list(lambda = 10)
)
```

Для обработки подобных случаев можно использовать функцию `invoke_map()` (рис. 17.4).

```
invoke_map(f, param, n = 5) %>% str()
#> List of 3
#>  $ : num [1:5] 0.762 0.36 -0.714 0.531 0.254
#>  $ : num [1:5] 3.07 -3.09 1.1 5.64 9.07
#>  $ : int [1:5] 9 14 8 9 7
```

Первый аргумент — это список функций или символьный вектор имен функций. Второй аргумент — это список списков, предоставляющий аргументы, разные для разных функций. Последующие аргументы передаются всем функциям.

И вновь, чтобы упростить создание этих согласованных пар, можно использовать функцию `tribble()`.

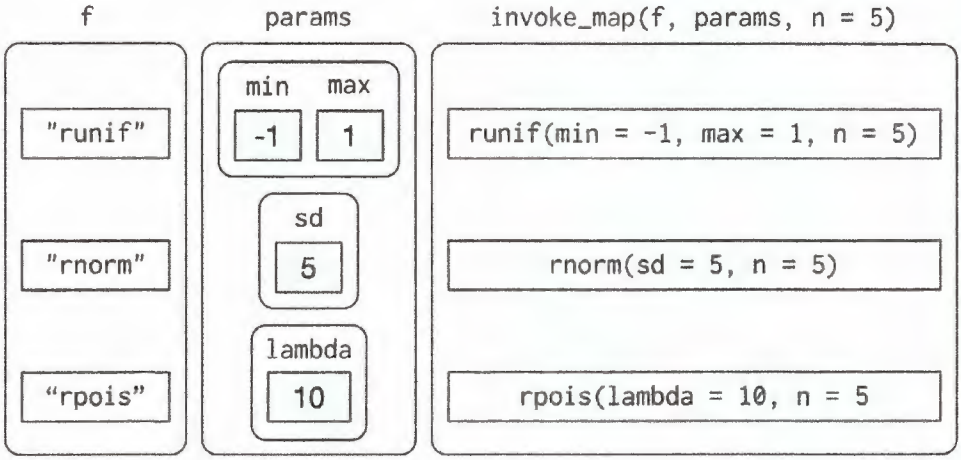


Рис. 17.4

```
sim <- tribble(
  ~f,       ~params,
  "runif", list(min = -1, max = 1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)
sim %>%
  mutate(sim = invoke_map(f, params, n = 10))
```

Функции семейства walk

Функция `walk()` — это альтернатива функции `map()`, используемая для вызова функции с целью получения побочных эффектов, а не возвращаемого значения. Как правило, это желательно в тех случаях, когда вы хотите отобразить вывод на экране иди сохранить файлы на диске, т.е. когда важно не получить возвращаемое значение, а выполнить какое-то действие. Обратимся к очень простому примеру.

```
x <- list(1, "a", 3)

x %>%
  walk(print)
#> [1] 1
#> [1] "a"
#> [1] 3
```

В типичных ситуациях функция `walk()` менее полезна, чем `walk2()` или `pwalk()`. Например, если имеется список графиков и вектор имен файлов, то

для сохранения каждого файла в соответствующем расположении на диске можно воспользоваться функцией `pwalk()`.

```
library(ggplot2)
plots <- mtcars %>%
  split(.$cyl) %>%
  map(~ggplot(., aes(mpg, wt)) + geom_point())
paths <- stringr::str_c(names(plots), ".pdf")

pwalk(list(paths, plots), ggsave, path = tempdir())
```

Каждая из функций `walk()`, `walk2()` и `pwalk()` неявно возвращает `.x` (первый аргумент), что делает их весьма удобными для использования в средних звеньях каналов.

Другие шаблоны циклов `for`

Пакет **purrr** предлагает ряд других функций, которые являются абстракциями, построенными поверх других типов циклов `for`. Они используются реже, чем функции семейства `map`, но их знание будем вам не лишним. Эти функции проиллюстрированы ниже на небольших примерах с той целью, чтобы вы могли вспомнить о них, столкнувшись с аналогичными проблемами в будущем. Более подробную информацию о них можно получить в справочной документации.

Функции-предикаты

Ряд функций работает с функциями-предикатами, которые возвращают одиночное значение `TRUE` или `FALSE`.

Функции `keep()` и `discard()` удерживают те элементы входного объекта, для которых предикатом является соответственно значение `TRUE` или `FALSE`.

```
iris %>%
  keep(is.factor) %>%
  str()
#> 'data.frame':     150 obs. of  1 variable:
#>  $ Species: Factor w/ 3 levels "setosa","versicolor",..: ...

iris %>%
  discard(is.factor) %>%
  str()
#> 'data.frame':     150 obs. of  4 variables:
#>  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#>  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3 ...
```

```
#>  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 ...
#>  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 ...
```

Функции `some()` и `every()` определяют, является ли предикат истинным для некоторых или для всех элементов.

```
x <- list(1:5, letters, list(10))

x %>%
  some(is_character)
#> [1] TRUE

x %>%
  every(is_vector)
#> [1] TRUE
```

Функция `detect()` находит первый элемент с истинным предикатом, а функция `detect_index()` возвращает позицию этого элемента.

```
x <- sample(10)
x
#>  [1]  8  7  5  6  9  2 10  1  3  4

x %>%
  detect(~ . > 5)
#> [1] 8

x %>%
  detect_index(~ . > 5)
#> [1] 1
```

Функции `head_while()` и `tail_while()` принимают элементы с начала или с конца вектора, до тех пор пока предикат имеет истинное значение.

```
x %>%
  head_while(~ . > 5)
#> [1] 8 7

x %>%
  tail_while(~ . > 5)
#> integer(0)
```

Функции reduce и accumulate

В случае сложного списка иногда возникает потребность в сведении его к более простому списку путем повторного применения функции, преобразующей пару в синглтон (объект-одиночку). Это может быть полезным, если вы

хотите применить двухтабличный глагол **dplyr** к нескольким таблицам. Например, у вас может быть список фреймов данных, и вы хотите свести его к одиночному фрейму, объединив все элементы.

```
dfs <- list(
  age = tibble(name = "John", age = 30),
  sex = tibble(name = c("John", "Mary"), sex = c("M", "F")),
  trt = tibble(name = "Mary", treatment = "A")
)

dfs %>% reduce(full_join)
#> Joining, by = "name"
#> Joining, by = "name"
#> # A tibble: 2 × 4
#>    name   age   sex treatment
#>   <chr> <dbl> <chr>     <chr>
#> 1  John    30     M      <NA>
#> 2  Mary    NA     F         A
```

Другим примером может быть список векторов, для которого вы хотите найти пересечение.

```
vs <- list(
  c(1, 3, 5, 6, 10),
  c(1, 2, 3, 7, 8, 10),
  c(1, 2, 3, 4, 8, 9, 10)
)

vs %>% reduce(intersect)
#> [1]  1  3 10
```

Функция `reduce()` принимает *бинарную* функцию (т.е. функцию с двумя основными входными переменными) и повторно применяет ее к списку до тех пор, пока остается хотя бы один элемент.

Функция `accumulate()` аналогична предыдущей, но удерживает все промежуточные результаты. Вы можете использовать ее для реализации кумулятивного суммирования.

```
x <- sample(10)
x
#>  [1]  6  9  8  5  2  4  7  1 10  3
x %>% accumulate(`+`)
#>  [1]  6 15 23 28 30 34 41 42 52 55
```

Упражнения

1. Реализуйте собственную версию функции `every()` с помощью цикла `for`. Сравните ее с функцией `purrr::every()`. Что из того, что не делает ваша версия, делает версия **purrr**?
2. Создайте усовершенствованную версию функции `col_sum()`, которая применяет итоговую функцию к каждому числовому столбцу фрейма данных.
3. Исходя из базовых возможностей R можно было бы предложить следующий эквивалент функции `col_sum()`.

```
col_sum3 <- function(df, f) {
  is_num <- sapply(df, is.numeric)
  df_num <- df[, is_num]

  sapply(df_num, f)
}
```

Однако эта функция содержит ряд логических ошибок, о чем свидетельствует следующий пример.

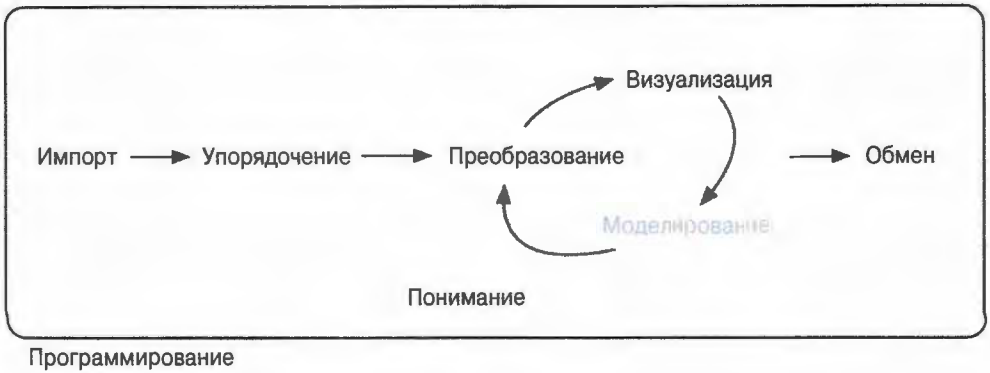
```
df <- tibble(
  x = 1:3,
  y = 3:1,
  z = c("a", "b", "c")
)
# OK
col_sum3(df, mean)
# Есть проблема: не всегда возвращается числовой вектор
col_sum3(df[1:2], mean)
col_sum3(df[1], mean)
col_sum3(df[0], mean)
```

Что послужило причиной ошибок?

ЧАСТЬ IV

Моделирование

Теперь, когда в вашем распоряжении имеются мощные программные инструменты, мы наконец-то можем вернуться к моделированию. Вы будете использовать свои новые инструменты для подготовки данных и их программной обработки в рамках многих моделей, чтобы понять, как они работают. В этой книге основное внимание уделяется *разведочному анализу данных* (exploratory data analysis — EDA), а не подтверждению гипотез или формальному статистическому выводу. Но мы познакомим вас с несколькими базовыми инструментами, которые помогут вам понять различия между моделями.



Целью моделирования является получение простых суммарных (сводных, итоговых) характеристик набора данных. В идеальном случае модель должна формально представлять истинные "сигналы" (т.е. закономерности, генерируемые явлением, которое представляет интерес) и игнорировать "шум" (т.е. случайные вариации, не представляющие для вас интереса). В данной книге рассматриваются лишь "предсказательные" (прогнозные) модели, которые, как говорит само название, генерируют прогнозы. Существует другой тип моделей, который мы не будем здесь обсуждать: модели "обнаружения данных" (data discovery). Эти модели не позволяют делать прогнозы, но облегчают выявление интересных закономерностей в массивах данных.

В данной книге не будет излагаться математическая теория, лежащая в основе этих моделей. Однако мы постараемся помочь вам добиться интуитивного понимания того, как работают статистические модели, и предоставим семейство полезных инструментов, с помощью которых вы сможете использовать модели, позволяющие лучше понимать свойства исследуемых данных.

- В главе 18 работа моделей обсуждается с чисто технической точки зрения с акцентом на важном семействе линейных моделей. Вы ознакомитесь с инструментами общего назначения, которые позволят вам глубже осмысливать информацию о ваших данных, предоставляемую прогнозной моделью, и научитесь работать с этими инструментами на примере простых наборов данных.
- В главе 19 вы узнаете о способах определения характеристик известных закономерностей, существующих в реальных данных, на основе использования моделей. Как только вам удастся обнаружить важную закономерность, ее стоит изучить с помощью модели, поскольку это облегчит выявление остающихся слабых сигналов.
- В главе 20 вы узнаете о том, как простые модели способствуют пониманию свойств сложных наборов данных. Это мощная методика, но ее применение требует совместного использования инструментов моделирования и программных средств.

Особенностью перечисленных тем является то, что они вообще не затрагивают инструментов, предназначенных для количественной оценки моделей. Это сделано намеренно: точное количественное описание моделей требует понимания концепций, для рассмотрения которых в этой книге просто не хватило бы места. Поэтому вам придется полагаться только на качественные оценки и собственный здоровый скептицизм. В конце главы 19 приведены ссылки на другие источники, в которых изложена более подробная информация по данной теме.

Выдвижение гипотез и их подтверждение

В этой книге мы будем исследовать данные путем моделирования, являющегося, наряду с визуализацией и преобразованием данных, одной из составляющих разведочного анализа данных (EDA), о чем говорилось в части I. Обычно модели обсуждаются в несколько ином разрезе, однако, как вы далее увидите, они являются эффективным инструментом EDA. Традиционный

подход предполагает применение моделирования в качестве инструмента статистического вывода или инструмента подтверждения справедливости выдвинутой гипотезы. Работа эта несложная, но трудоемкая.

Корректный статистический вывод требует соблюдения следующих двух базовых принципов.

- Каждое наблюдение можно использовать либо для разведочного анализа, либо для подтверждения гипотезы, но не для того и другого одновременно.
- В целях разведочного анализа допускается многократное использование наблюдения, но в целях подтверждения гипотез — только однократное. Использование одного и того же наблюдения более одного раза равносильно переходу от подтверждения гипотезы к разведочному анализу.

Необходимость следования этим принципам обусловлена тем, что данные, используемые для подтверждения гипотезы, не должны зависеть от данных, на основании которых эта гипотеза была выдвинута. В противном случае вы займете позицию сверхоптимиста. Многократное использование одних и тех же наблюдений в исследованиях вполне допустимо, в то время как подтверждение гипотезы на основе результатов предварительного исследования — в корне неверный подход.

Один из возможных подходов к анализу, направленному на подтверждение гипотезы, заключается в разбиении данных на три части еще до того, как вы приступите к анализу:

- 60% данных используются в качестве *тренировочного* набора (training set); с этими данными вы вправе делать все, что считаете нужным (можете визуализировать их и опробовать их на десятках моделей);
- 20% данных поступают в набор, используемый для *запросов* (query set); вы можете использовать эти данные для сравнения моделей или вариантов визуализации вручную, но их не разрешается использовать в качестве части автоматизированного процесса;
- 20% данных остаются в *тестовом* наборе (test set); эти данные можно использовать только ОДИН раз для тестирования окончательной модели.

Такое распределение данных позволяет вам исследовать тренировочные данные, при необходимости выдвигая гипотезы-кандидаты, которые можно проверять с помощью набора для запросов. Как только у вас появилась

уверенность в том, что вы построили подходящую модель, можете проверить ее только один раз, используя для этого тестовый набор.

Имейте в виду, что даже в случае моделирования с целью подтверждения гипотезы вы по-прежнему нуждаетесь в EDA. Не выполняя EDA, вы будете оставаться слепым в отношении проблем качества данных.

ГЛАВА 18

Базовое моделирование с помощью пакета `modelr`

Введение

Цель моделирования — определение суммарных характеристик набора данных. В контексте этой книги мы будем использовать модели для разделения данных на шаблоны, которые отражают определенные закономерности, и остатки. Сильные шаблоны стремятся скрыть слабые тренды, поэтому мы будем использовать модели для раскрытия слоев структуры по мере исследования набора данных.

Однако для того, чтобы применять модели к реальным данным, представляющим интерес, необходимо хотя бы в общих чертах представлять себе, как работают модели. В этом отношении данная глава особая, поскольку в ней используются лишь искусственные наборы данных. Эти наборы очень просты и совсем не интересны, но с ними вам будет легче понять суть моделирования, а изученные при этом методики вы сможете применить к реальным данным в следующей главе.

Моделирование включает два аспекта.

1. Прежде всего вы определяете *семейство моделей*, представляющих типичную закономерность, например линейную или квадратичную зависимость. Семейство моделей описывается уравнением наподобие `y = a_1 * x + a_2` или `y = a_1 * x ^ a_2`, где `x` и `y` — известные переменные из вашего набора данных, а `a_1` и `a_2` — варьируемые параметры.
2. Далее вы генерируете подходящую модель, выбирая ту из семейства моделей, которая больше всего соответствует вашим данным. В результате этого обобщенное уравнение семейства моделей конкретизируется, принимая вид наподобие `y = 3 * x + 7` или `y = 9 * x ^ 2`.

Важно, чтобы вы понимали, что подобранная таким образом модель — всего лишь ближайшая к вашему набору данных модель, выбранная из какого-то их семейства. Это означает, что вы получили "наилучшую" (в соответствии с некоторым критерием) модель в пределах данного семейства, но отсюда вовсе не следует, что вы получили хорошую модель, и это несомненно не означает, что данная модель "истинна". Джордж Бокс метко охарактеризовал эту ситуацию в своем знаменитом афоризме:

> Все модели неправильны, но некоторые из них полезны.

Стоит привести более полный фрагмент из цитаты:

> "Было бы замечательно, если бы любая из существующих в реальном мире систем могла быть точно описана какой-то простой моделью. Однако тщательно подобранная расчетливая модель часто способна обеспечить необычайно полезное приближение. Например, закон PV = RT, связывающий давление P, объем V и температуру T "идеального" газа посредством константы R, не является точным для реальных газов, но он часто служит достаточно хорошим приближением, а его структура информативна, поскольку следует из физической картины поведения молекул газа.
>
> Вопрос "А истинна ли сама модель?" по отношению к такой модели неуместен. Если под "истиной" понимать "полную истину" , то ответ на этот вопрос должен быть отрицательным. Единственный вопрос, которым имеет смысл задаваться: "Достаточно ли информативна данная модель и полезна ли она?"

Целью моделирования является не установление окончательной истины, а нахождение простого, но тем не менее полезного приближения.

Необходимые ресурсы

В этой главе мы будем использовать пакет **modelr**, выступающий в качестве обертки для базовых функций моделирования R, которая делает более естественным их использование в канале.

```
library(tidyverse)

library(modelr)
options(na.action = na.warn)
```

Простая модель

Обратимся к демонстрационному набору данных `sim1`, который содержит две непрерывные переменных: `x` и `y`. Отложим их на графике, чтобы увидеть, как они связаны между собой (рис. 18.1).

```
ggplot(sim1, aes(x, y)) +
  geom_point()
```

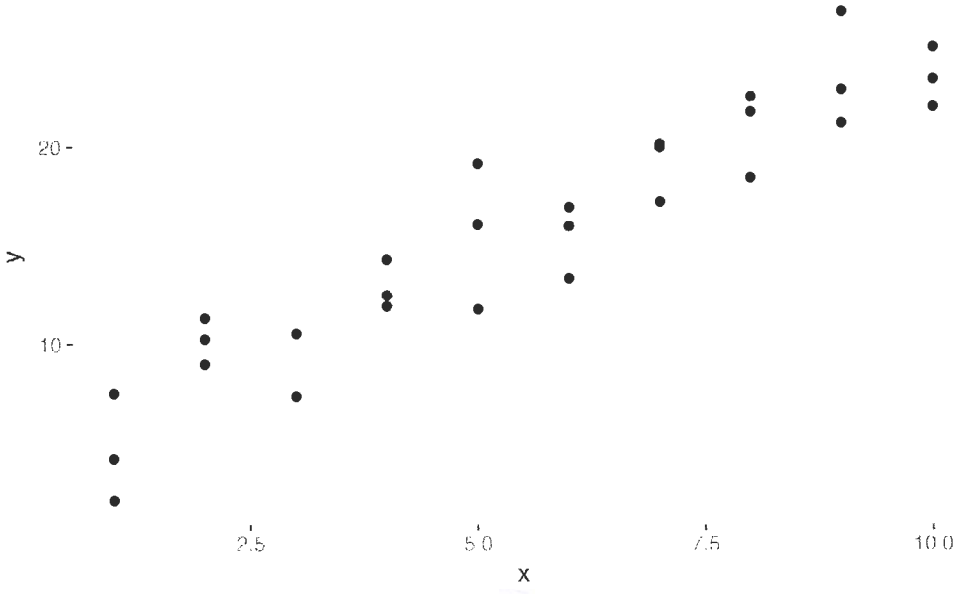


Рис. 18.1

Мы видим, что в данном случае между переменными наблюдается сильная взаимосвязь. Опишем эту зависимость с помощью модели и выразим ее в явном виде. Наша задача — предоставить базовую форму модели. В данном случае мы используем линейное соотношение между переменными, т.е. `y = a_0 + a_1 * x`. Начнем с получения общего представления о том, как выглядят модели этого семейства, путем случайной генерации некоторых из них и наложения их на данные. В нашем простом случае можно использовать функцию `geom_abline()`, которая принимает *наклон* (slope) и *y-пересечение* (intercept) в качестве параметров (рис. 18.2). Позже вы познакомитесь с более общими методиками, которые работают с любой моделью.

```
models <- tibble(
  a1 = runif(250, -20, 40),
  a2 = runif(250, -5, 5)
```

```
)

ggplot(sim1, aes(x, y)) +
  geom_abline(
    aes(intercept = a1, slope = a2),
    data = models, alpha = 1/4
) +
geom_point()
```

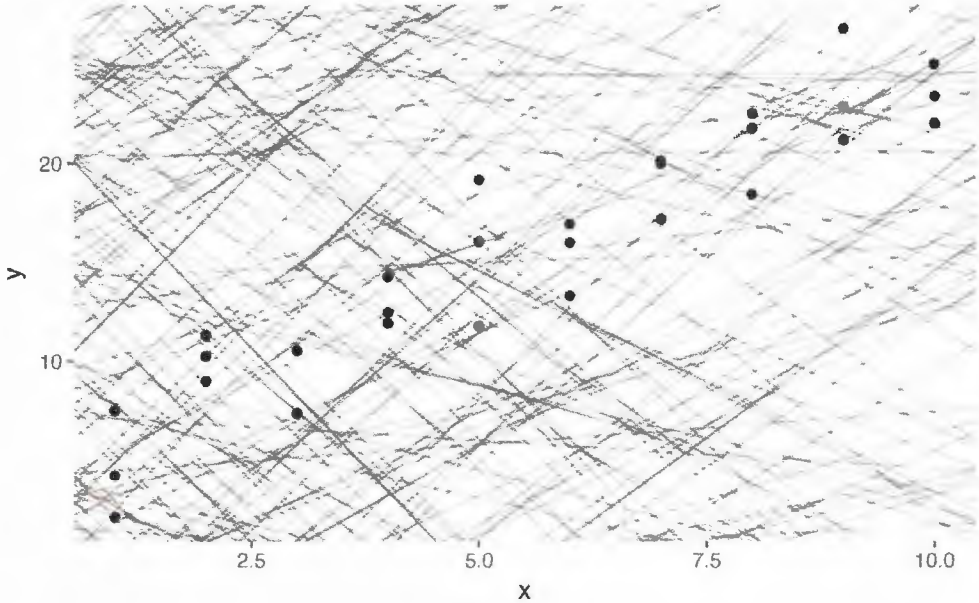


Рис. 18.2

На этом графике представлены 250 моделей, но большинство из них совершенно неудачны! Мы должны отыскать среди них удачные модели, уточнив наше интуитивное понимание того, что означает понятие "близость к данным" для хорошей модели. Нам нужен метод, обеспечивающий количественную оценку расстояния между данными и моделью. После этого мы сможем подогнать модель под данные, найдя такие значения параметров `a_0` и `a_1`, которые генерируют модель, отстоящую от данных на минимальное расстояние.

Для начала проще всего оттолкнуться от расстояния по вертикали между каждой точкой данных и моделью, как показано на следующей диаграмме (рис. 18.3). Обратите внимание на то, что мы немного сместили некоторые значения `x`, чтобы сделать более различимыми линии, отмечающие указанное расстояние для отдельных точек.

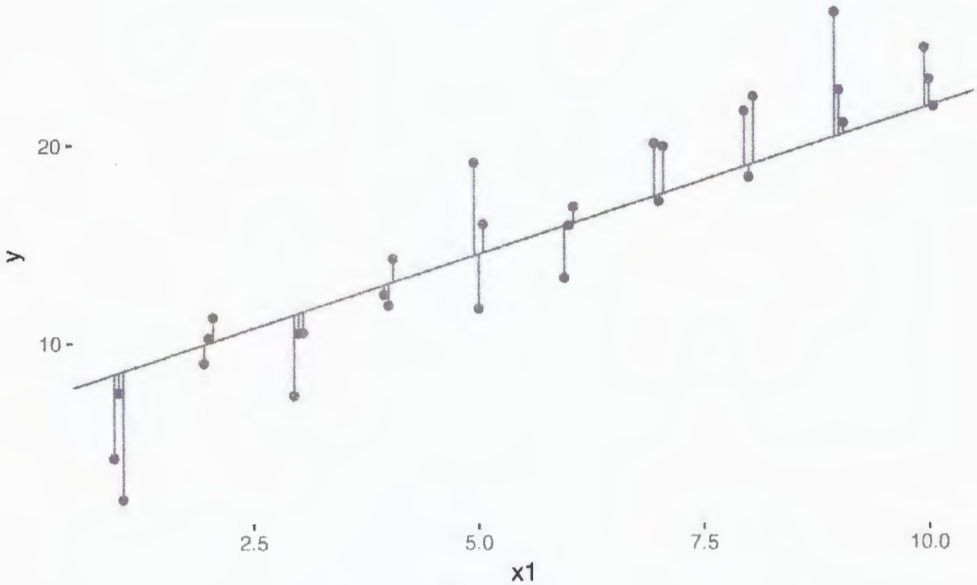


Рис. 18.3

Это расстояние — просто разность между значением `y`, предоставляемым моделью (*прогноз*), и фактическим значением `y` согласно данным `data` (*результат*).

Для вычисления расстояния мы прежде всего превратим семейство моделей в функцию R. В качестве аргументов эта функция принимает параметры модели и данные, а в качестве выходного результата предоставляет значения, предсказанные моделью.

```
model1 <- function(a, data) {
  a[1] + data$x * a[2]
}
model1(c(7, 1.5), sim1)
#>  [1]  8.5  8.5  8.5 10.0 10.0 10.0 11.5 11.5 11.5 13.0 13.0
#> [12] 13.0 14.5 14.5 14.5 16.0 16.0 16.0 17.5 17.5 17.5 19.0
#> [23] 19.0 19.0 20.5 20.5 20.5 22.0 22.0 22.0
```

Нам также нужен способ вычисления общего расстояния между предсказанными и фактическими значениями. Иными словами, если на графике отображены 30 расстояний, то каким образом они могут быть сведены к одному числу?

Одним из распространенных в статистике способов является использование так называемого *среднеквадратического отклонения*. Мы вычисляем разности между фактическими и предсказанными значениями, возводим их

в квадрат и усредняем, а из полученного результата извлекаем квадратный корень. Это расстояние обладает рядом привлекательных математических свойств, которые мы здесь не будем обсуждать. Просто поверьте на слово!

```
measure_distance <- function(mod, data)
  { diff <- data$y - model1(mod, data)
  sqrt(mean(diff ^ 2))
}
measure_distance(c(7, 1.5), sim1)
#> [1] 2.67
```

Теперь мы можем использовать пакет **purrr** для того, чтобы вычислить это расстояние для всех ранее определенных моделей. Нам нужна вспомогательная функция, поскольку наша функция, вычисляющая расстояние, ожидает модель в качестве числового вектора с длиной 2.

```
sim1_dist <- function(a1, a2) {
  measure_distance(c(a1, a2), sim1)
}

models <- models %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))
models
#> # A tibble: 250 x 3
#>       a1      a2  dist
#>    <dbl>   <dbl> <dbl>
#> 1 -15.15  0.0889  30.8
#> 2  30.06 -0.8274  13.2
#> 3  16.05  2.2695  13.2
#> 4 -10.57  1.3769  18.7
#> 5 -19.56 -1.0359  41.8
#> 6   7.98  4.5948  19.3
#> # ... with 244 more rows
```

Давайте наложим 10 наших наилучших моделей на данные. Мы выделили модели цветом, зависящим от значения `-dist`: это наилучший способ добиться того, чтобы наилучшие модели (т.е. те, которые характеризуются наименьшим расстоянием) выглядели ярче других (рис. 18.4).

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, color = -dist),
    data = filter(models, rank(dist) <= 10)
)
```

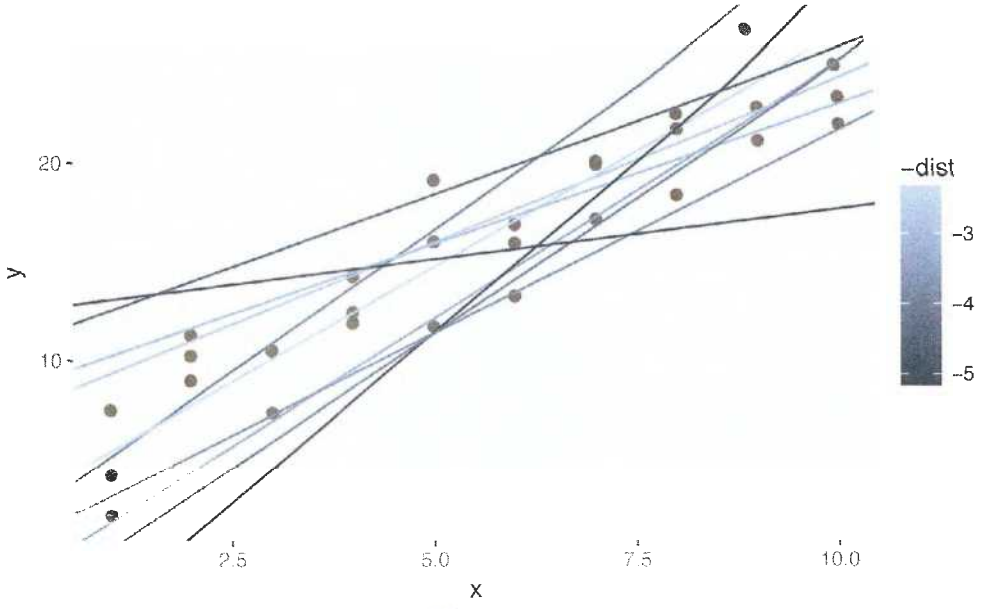


Рис. 18.4

Мы также можем рассматривать эти модели как наблюдения и визуализировать их с помощью точечной диаграммы с осями `a1` и `a2`, опять-таки расцветив ее в соответствии со значениями `-dist`. При этом мы уже не сможем непосредственно увидеть, как конкретная модель сопоставляется с данными, но зато увидим множество моделей одновременно. И вновь мы выделили цветом 10 наилучших моделей, на этот раз обведя их красными кружочками (рис. 18.5).

```
ggplot(models, aes(a1, a2)) +
  geom_point(
    data = filter(models, rank(dist) <= 10),
    size = 4, color = "red"
) +
geom_point(aes(colour = -dist))
```

Вместо испытания множества случайных моделей можно выбрать более систематический подход, сгенерировав точки данных, образующие равномерную сетку (так называемую *сетку поиска*). Мы выбрали параметры сетки довольно грубо, исходя из того, как были расположены на предыдущем графике лучшие модели (рис. 18.6).

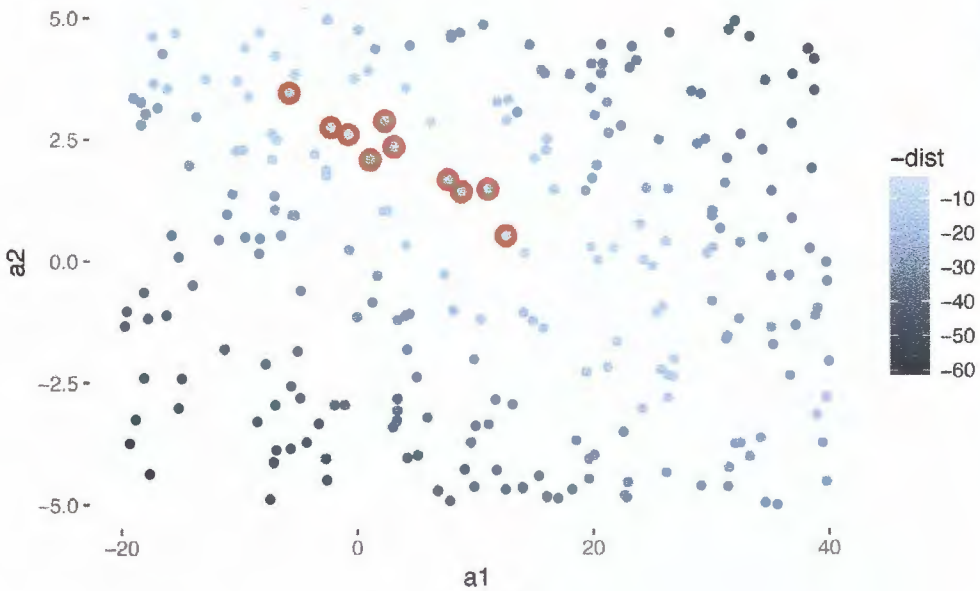


Рис. 18.5

```
grid <- expand.grid(
  a1 = seq(-5, 20, length = 25),
  a2 = seq(1, 3, length = 25)
  ) %>%
  mutate(dist = purrr::map2_dbl(a1, a2, sim1_dist))

grid %>%
  ggplot(aes(a1, a2)) +
  geom_point(
    data = filter(grid, rank(dist) <= 10),
    size = 4, colour = "red"
) +
geom_point(aes(color = -dist))
```

Если наложить эти 10 наилучших моделей на исходные данные, то они будут выглядеть довольно прилично (рис. 18.7).

```
ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(
    aes(intercept = a1, slope = a2, color = -dist),
    data = filter(grid, rank(dist) <= 10)
)
```

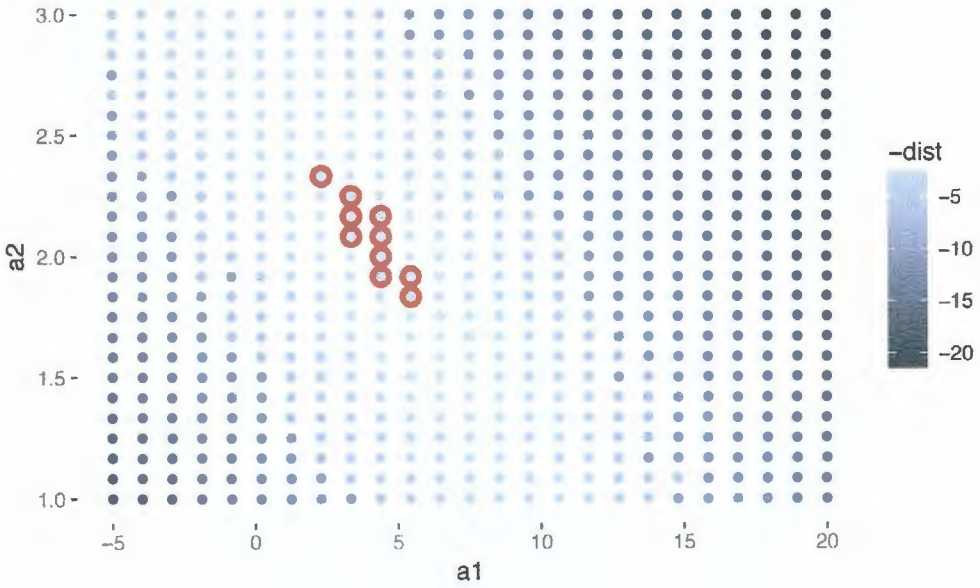


Рис. 18.6

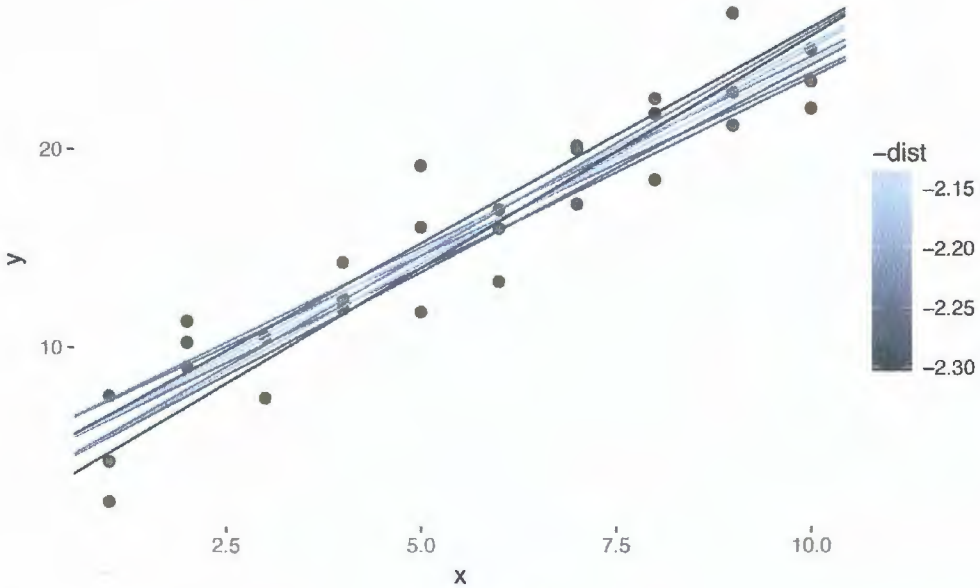


Рис. 18.7

Этот процесс можно представить себе как итеративное уменьшение шага сетки до тех пор, пока она не сойдется к наилучшей модели. Однако для данной проблемы имеется более эффективный способ решения: *численный*

метод минимизации Ньютона–Рафсона (другое название — *метод касательных*). Логика, лежащая в основе этого метода, довольно проста: вы выбираете начальную точку и ищете направление, характеризующееся наибольшей крутизной наклона. Далее делается небольшой шаг в этом направлении, и процедура повторяется вновь и вновь до тех пор, пока дальнейшее продвижение вниз не прекратится. В R это можно сделать с помощью функции `optim()` (рис. 18.8).

```
best <- optim(c(0, 0), measure_distance, data = sim1)
best$par
#> [1] 4.22 2.05

ggplot(sim1, aes(x, y)) +
  geom_point(size = 2, color = "grey30") +
  geom_abline(intercept = best$par[1], slope = best$par[2])
```

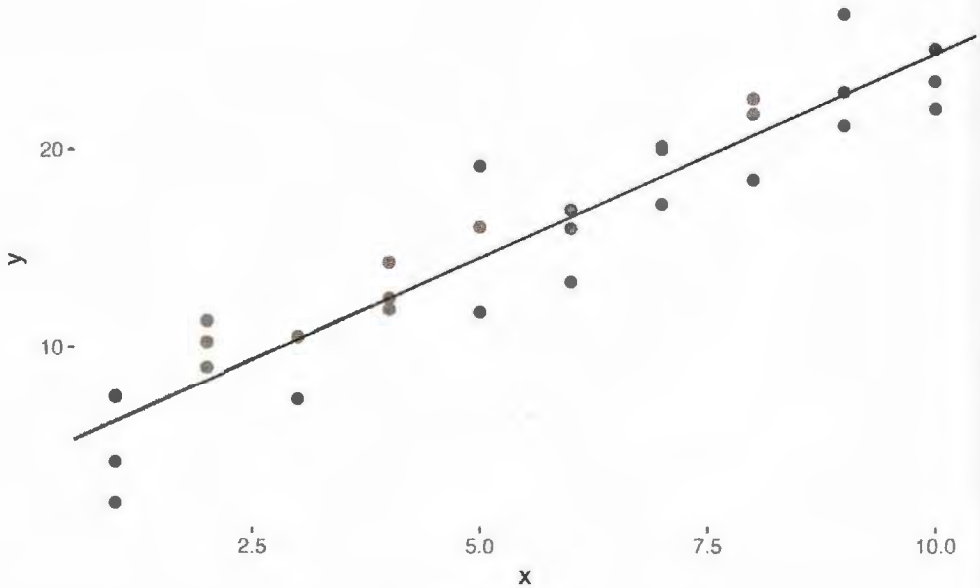


Рис. 18.8

Не задумывайтесь относительно того, как работает функция `optim()`, — важную роль здесь играет интуиция. Если у вас есть функция, определяющая расстояние между моделью и набором данных, и алгоритм минимизации этого расстояния путем варьирования параметров модели, то вы сможете найти наилучшую модель. Приятной особенностью данного подхода является то, что он будет работать для любого семейства моделей, которое можно описать уравнением.

Существует еще один подход, пригодный для этой модели, поскольку она является частным случаем более широкого семейства *линейных моделей*. Линейная модель описывается следующим уравнением общего вида:

```
y = a_1 + a_2 * x_1 + a_3 * x_2 + ... + a_n * x_(n - 1)
```

Поэтому наша модель эквивалентна общей линейной модели, в которой `n` равно 2, а `x_1` — это `x`. В R имеется специальный инструмент, предназначенный для подбора линейных моделей: функция `lm()`. Для функции `lm()` предусмотрен особый способ указания семейства моделей — формулы, которые выглядят примерно так: `y ~ x`, что транслируется функцией `lm()` к виду `y = a_1 + a_2 * x`. Давайте подберем модель и посмотрим на результат.

```
sim1_mod <- lm(y ~ x, data = sim1)
coef(sim1_mod)
#> (Intercept)           x
#>        4.22        2.05
```

Это в точности те же значения, которые мы получили с помощью функции `optim()`! За "кулисами" функция `lm()` не привлекает функцию `optim()`, а использует преимущества математической структуры линейных моделей. Используя определенные связи между геометрией, дифференциальным исчислением и линейной алгеброй, функция `lm()` фактически находит ближайшую модель за один шаг с помощью сложного алгоритма. Этот подход работает быстрее и гарантирует, что будет найден глобальный минимум.

Упражнения

1. Одним из недостатков линейной модели является то, что она чувствительна к необычным значениям, поскольку формула для расстояния включает квадратный член. Подберите линейную модель для генерируемых ниже искусственных данных и визуализируйте результаты. Проведите этот эксперимент несколько раз, генерируя различные данные. Выскажите свои соображения по поводу модели.

   ```
   sim1a <- tibble(
     x = rep(1:10, each = 3),
     y = x * 1.5 + 6 + rt(length(x), df = 2)
   )
   ```

2. Одним из способов повышения робастности линейной модели является использование различных мер расстояния. Например, вместо

среднеквадратического расстояния можно использовать среднее абсолютное расстояние.

```
measure_distance <- function(mod, data) {
  diff <- data$y - make_prediction(mod, data)
  mean(abs(diff))
}
```

Подгоните параметры модели под предыдущий искусственно сгенерированный набор данных с помощью функции `optim()` и сравните результаты с результатами для линейной модели.

3. Одна из проблем, связанных с выполнением численной оптимизации, заключается в том, что гарантируется нахождение лишь одного локального оптимума. Какая проблема связана с оптимизацией в рамках трехпараметрической модели, приведенной ниже?

```
model1 <- function(a, data) {
  a[1] + data$x * a[2] + a[3]
}
```

Визуализация моделей

Простые модели наподобие тех, которые мы рассмотрели в предыдущем разделе, удобны тем, что позволяют исследовать, какие закономерности отражает модель, путем тщательного изучения семейства моделей и подбора оптимальных значений коэффициентов. И если вашей целью является применение моделей для исследования статистических характеристик массивов данных, то именно этому, вероятнее всего, вы и будете посвящать большую часть времени. Однако сейчас мы рассмотрим моделирование под несколько иным углом и проанализируем возможности моделей как средства прогнозирования. Большим преимуществом является то, что любая предсказательная модель способна строить прогнозы (иначе, какая бы от нее была польза?), поэтому те же самые методики обеспечивают понимание предсказательных моделей любого типа.

Также полезно рассмотреть то, что моделью не охватывается, — так называемые остатки, т.е. то, что остается после вычитания прогнозных значений из фактических данных. Остатки предоставляют чрезвычайно полезную информацию, поскольку они позволяют использовать модели для устранения экстремальных пиков и тем самым дают возможность изучать слабые остаточные тренды.

Предсказания

Визуализацию предсказаний на основе модели мы начнем с того, что сгенерируем равномерную сетку значений, охватывающих всю область, в которой располагаются наши данные. Проще всего это можно сделать с помощью функции `modelr::data_grid()`. Ее первым аргументом является фрейм данных, а для всех последующих аргументов она находит уникальные переменные и генерирует все комбинации.

```
grid <- sim1 %>%
  data_grid(x)
grid
=> # A tibble: 10 x 1
=>       x
=>   <int>
=> 1     1
=> 2     2
=> 3     3
=> 4     4
=> 5     5
=> 6     6
=> # ... with 4 more rows
```

(Все станет гораздо интереснее, когда мы начнем добавлять переменные в нашу модель.)

Следующим нашим шагом является добавление предсказаний. Мы будем использовать функцию `modelr::add_predictions()`, которая принимает фрейм данных и модель. Полученные с помощью модели предсказания функция добавляет в новый столбец фрейма данных.

```
grid <- grid %>%
  add_predictions(sim1_mod)
grid
=> # A tibble: 10 x 2
=>       x  pred
=>   <int> <dbl>
=> 1     1  6.27
=> 2     2  8.32
=> 3     3 10.38
=> 4     4 12.43
=> 5     5 14.48
=> 6     6 16.53
=> # ... with 4 more rows
```

(Эту функцию также можно использовать для добавления предсказаний в исходный набор данных.)

Далее мы откладываем предсказания в виде графика. Возможно, вас удивляет, зачем мы выполняем всю эту дополнительную работу, если можно просто воспользоваться функцией `geom_abline()`. Преимуществом описываемого подхода является то, что он будет работать с любой моделью в R, от самой простой до самой сложной (рис. 18.9). Вас могут ограничивать лишь ваши навыки в области визуализации данных. Подробную информацию относительно визуализации более сложных типов моделей можно найти по адресу `http://vita.had.co.nz/papers/model-vis.html`.

```
ggplot(sim1, aes(x)) +
  geom_point(aes(y = y)) +
  geom_line(
    aes(y = pred),
    data = grid,
    colour = "red",
    size = 1
)
```

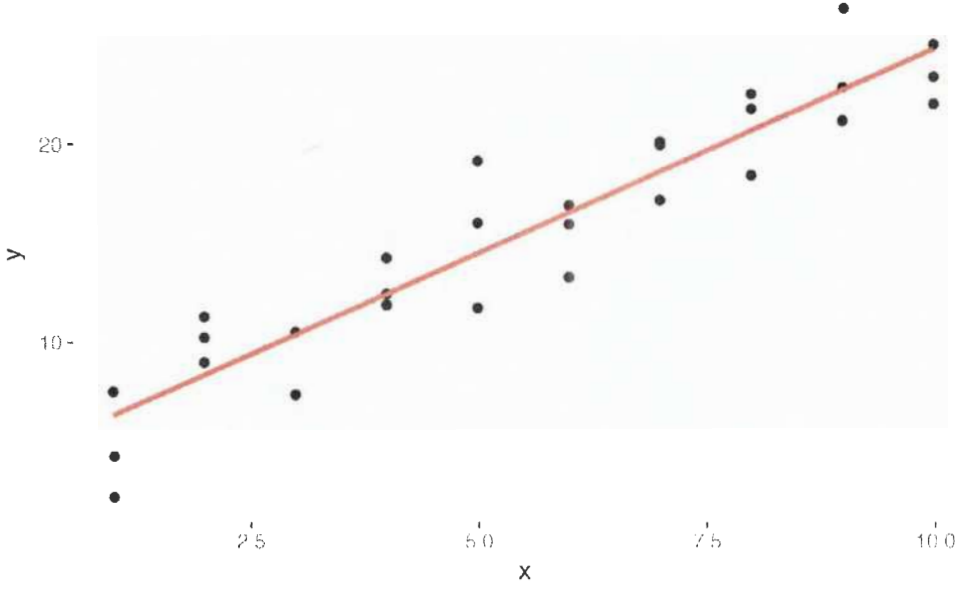


Рис. 18.9

Остатки

Обратной стороной предсказаний являются *остатки*. Предсказания отражают закономерность, которую модели удалось уловить, тогда как остатки предоставляют информацию о том, что не охватывается моделью. Остатки — это не что иное, как расстояния между наблюдениями и вычисленными нами прогнозными значениями.

Мы добавляем остатки к данным с помощью функции `add_residuals()`, которая работает во многом аналогично функции `add_predictions()`. Заметьте, однако, что мы используем исходный набор данных, а не искусственно созданную сетку. Это обусловлено тем, что для вычисления остатков нам нужны фактические значения `y`.

```
sim1 <- sim1 %>%
  add_residuals(sim1_mod)
sim1
#> # A tibble: 30 x 3
#>       x     y  resid
#>   <int> <dbl>  <dbl>
#> 1     1  4.20 -2.072
#> 2     1  7.51  1.238
#> 3     1  2.13 -4.147
#> 4     2  8.99  0.665
#> 5     2 10.24  1.919
#> 6     2 11.30  2.973
#> # ... with 24 more rows
```

Существует несколько способов извлечения информации о модели на основе остатков. Один из них — это графическое представление распределения остатков с помощью *полигона частот* (рис. 18.10).

```
ggplot(sim1, aes(resid)) +
  geom_freqpoly(binwidth = 0.5)
```

Это позволяет калибровать качество модели: насколько прогноз отличается от наблюдаемых значений? Обратите внимание на то, что среднее остатков всегда равно 0.

Потребность в построении графиков с использованием остатков вместо исходного предиктора будет возникать у вас довольно часто (рис. 18.11). Множество примеров этого вы найдете в следующей главе.

```
ggplot(sim1, aes(x, resid)) +
  geom_ref_line(h = 0) +
  geom_point()
```

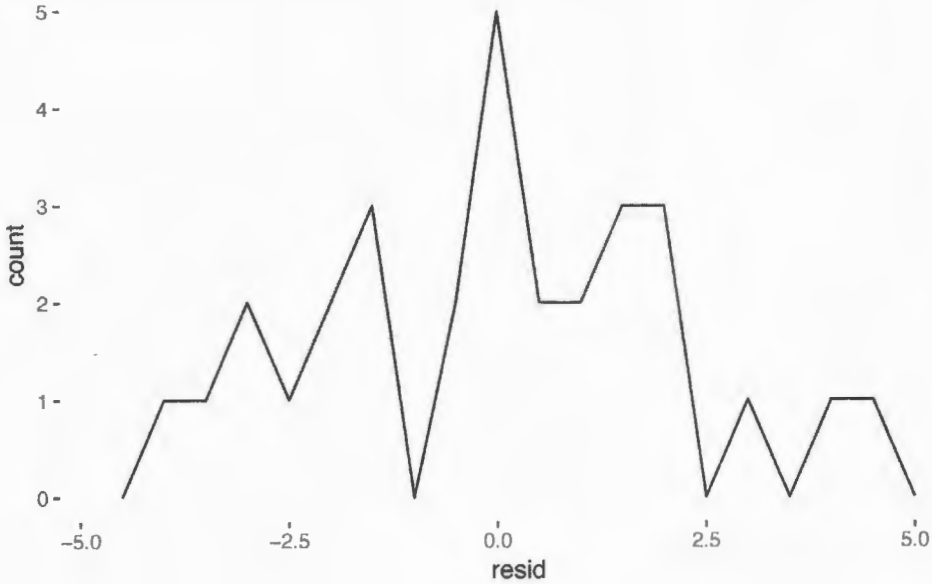


Рис. 18.10

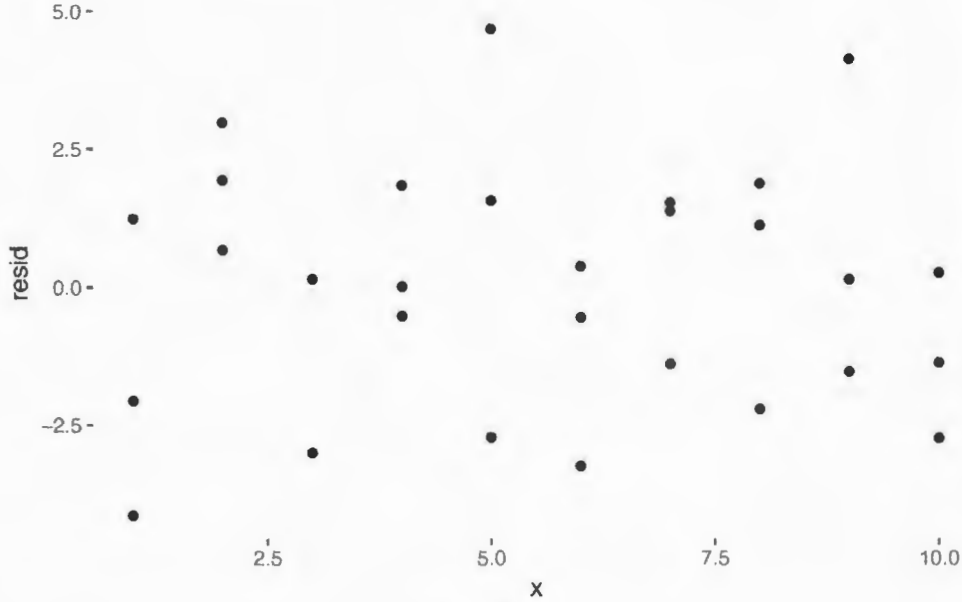


Рис. 18.11

Этот график напоминает случайный шум, указывая на то, что наша модель неплохо справилась со своей задачей определения закономерности в поведении данных.

Упражнения

1. Если с помощью функции `lm()` можно аппроксимировать данные прямой линией, то функция `loess()` позволяет аппроксимировать данные гладкой кривой. Повторите процесс подгонки модели, генерации сетки, прогнозирования и визуализации для набора данных `sim1`, используя функцию `loess()` вместо функции `lm()`. Сравните результаты с предыдущими.
2. У функции `add_predictions()` имеются родственные функции `gather_predictions()` и `spread_predictions()`. Чем различаются эти функции между собой?
3. Что делает функция `geom_ref_line()`? К какому пакету она принадлежит? Почему полезно и важно отображать базовую (нулевую) линию на графиках, представляющих остатки?
4. Зачем нужен полигон частот для абсолютных остатков? Укажите преимущества и недостатки этого способа по сравнению с использованием полигона частот для исходных остатков.

Формулы и семейства моделей

До этого вы уже сталкивались с формулами, когда использовали функции `facet_wrap()` и `facet_grid()`. В R формулы обеспечивают общий способ получения "специального поведения". Вместо того чтобы сразу же использовать переменные в вычислениях, они захватывают их для последующей интерпретации с помощью функции.

Большинство функций моделирования в R используют стандартное преобразование от формул к функциям. Один из примеров простого преобразования вы уже видели: `y ~ x` транслируется в `y = a_1 + a_2 * x`. Если вы хотите увидеть, что фактически делает R, используйте функцию `model_matrix()`. Эта функция принимает фрейм данных и формулу и возвращает tibble-фрейм, определяющий уравнение модели: каждый столбец результата связан с одним коэффициентом модели, а функция всегда имеет вид `y = a_1 * out1 + a_2 * out_2`. Для простого случая `y ~ x1` мы получаем нечто любопытное.

```
df <- tribble(
  ~y, ~x1, ~x2,
  4, 2, 5,
  5, 1, 6
)
```

```
model_matrix(df, y ~ x1)
#> # A tibble: 2 x 2
#>   `(Intercept)`    x1
#>           <dbl> <dbl>
#> 1             1     2
#> 2             1     1
```

Способ, используемый R для добавления y-пересечения в модель, сводится к простому добавлению столбца, заполненного единицами. По умолчанию R всегда добавляет такой столбец. Если хотите отказаться от него, то это делается добавлением "-1".

```
model_matrix(df, y ~ x1 - 1)
#> # A tibble: 2 x 1
#>      x1
#>   <dbl>
#> 1     2
#> 2     1
```

При добавлении дополнительных переменных в модель размер матрицы модели, как и следовало ожидать, увеличивается.

```
model_matrix(df, y ~ x1 + x2)
#> # A tibble: 2 x 3
#>   `(Intercept)`    x1    x2
#>           <dbl> <dbl> <dbl>
#> 1             1     2     5
#> 2             1     1     6
```

Эта формульная нотация, которую иногда называют *нотацией Уилкинсона–Роджерса*, была впервые описана в статье "Symbolic Description of Factorial Models for Analysis of Variance" (https://www.jstor.org/stable/2346786?seq=1#page_scan_tab_contents). Чтобы получить полную информацию об алгебре моделирования, прочитайте указанную статью.

В следующих разделах подробно рассказано о том, как формульная нотация работает в случае категориальных переменных, взаимодействий и преобразований.

Категориальные переменные

Если предиктором является непрерывная переменная, то генерация функции из формулы осуществляется непосредственным образом, но если предиктор — категориальная переменная, то все немного усложняется. Предположим, у вас имеется формула наподобие `y ~ sex`, где переменная `sex` (пол)

может принимать два значения, которые обозначают мужчину (male) и женщину (female). Преобразовывать эту формулу к виду y = x_0 + x_1 * sex не имеет смысла, поскольку переменная sex не является числом — она не может участвовать в умножении! Вместо этого R преобразует ее к виду y = x_0 + x_1 * sex_male, где sex_male принимает значение 1 для мужского пола и 0 — для женского.

```
df <- tribble(
  ~ sex, ~ response,
  "male", 1,
  "female", 2,
  "male", 1
)
model_matrix(df, response ~ sex)
#> # A tibble: 3 x 2
#>   `(Intercept)` sexmale
#>           <dbl>    <dbl>
#> 1             1        1
#> 2             1        0
#> 3             1        1
```

Возможно, вас удивляет, почему R не создает также столбец sexfemale. Проблема в том, что это создало бы столбец, состояние которого можно точно предсказать на основании состояний других столбцов (например, sexfemale = 1 - sexmale). К сожалению, рассмотрение всех деталей того, почему это является проблемой, выходит за рамки данной книги, но если говорить коротко, то суть проблемы в том, что это приводит к созданию чрезмерно гибкого семейства моделей, бесконечное количество которых будет равноудалено от данных.

К счастью, если основной интерес для вас представляет визуализация предсказаний, то вам не стоит беспокоиться о точной параметризации. Чтобы конкретизировать эти общие утверждения, обратимся к конкретным примерам наборов данных и моделей. В данном случае мы используем набор sim2 из пакета **modelr** (рис. 18.12).

```
ggplot(sim2) +
  geom_point(aes(x, y))
```

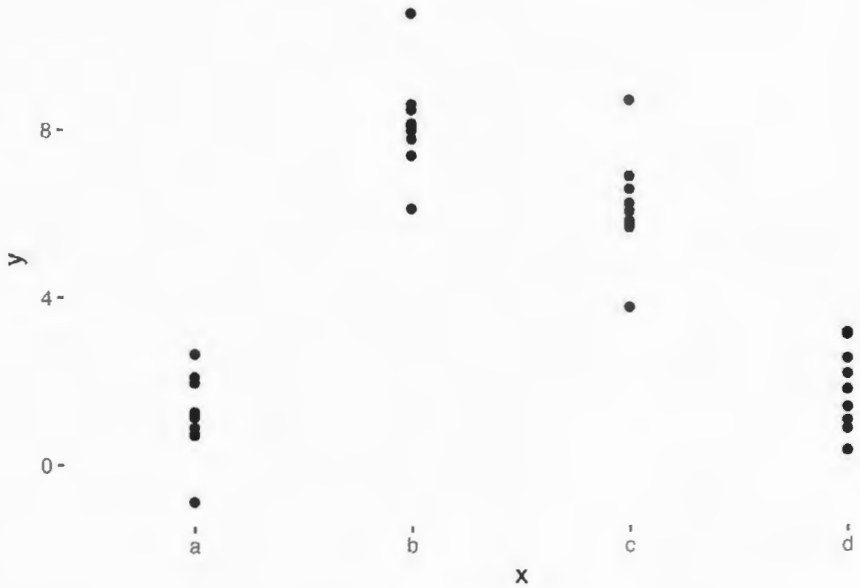


Рис. 18.12

Мы можем подобрать модель для этого набора данных и сгенерировать прогноз.

```
mod2 <- lm(y ~ x, data = sim2)
grid <- sim2 %>%
  data_grid(x) %>%
  add_predictions(mod2)
grid
#> # A tibble: 4 x 2
#>       x  pred
#>   <chr> <dbl>
#> 1     a  1.15
#> 2     b  8.12
#> 3     c  6.13
#> 4     d  1.91
```

В конечном счете модель с категориальной переменной `x` будет предсказывать среднее значение для каждой категории. (Почему? Потому что среднее минимизирует среднеквадратическое расстояние.) Это можно легко увидеть, если наложить предсказания на исходные данные (рис. 18.13).

```
ggplot(sim2, aes(x)) +
  geom_point(aes(y = y)) +
  geom_point(
    data = grid,
    aes(y = pred),
```

```
  color = "red",
  size = 4
```

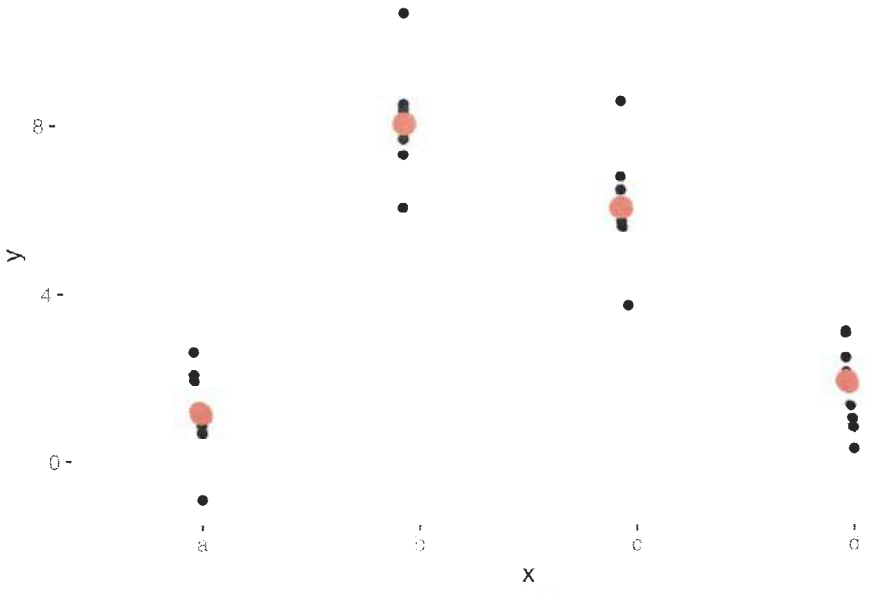


Рис. 18.13

Вы не можете делать прогнозы относительно уровней, которые не наблюдали. Иногда это будет получаться у вас случайно, поэтому стоит запомнить следующее сообщение об ошибке.

```
tibble(x = "e") %>%
  add_predictions(mod2)
#> Error in model.frame.default(Terms, newdata, na.action =
#> na.action, xlev = object$xlevels): factor x has new level e
```

Взаимодействия (непрерывная и категориальная переменные)

Что происходит в тех случаях, когда непрерывная переменная сочетается с категориальной? Набор данных `sim3` содержит категориальный и непрерывный предикторы. Мы можем визуализировать их с помощью простого графика (рис. 18.14).

```
ggplot(sim3, aes(x1, y)) +
  geom_point(aes(color = x2))
```

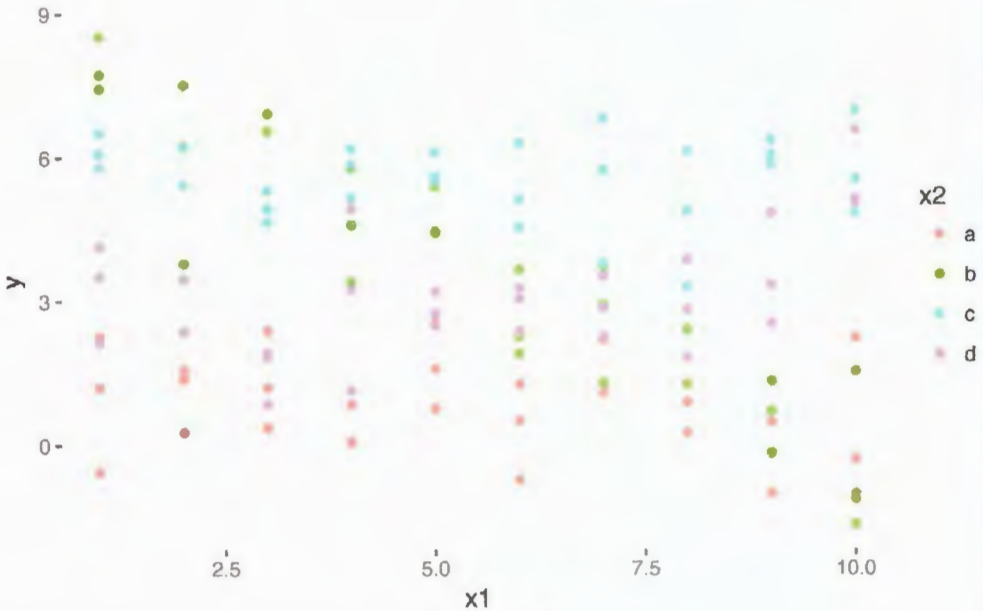


Рис. 18.14

Для этих данных возможны две модели.

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

Когда вы добавляете модели с помощью знака `+`, они оценивают каждый эффект независимо от всех остальных. Для подгонки так называемого взаимодействия можно использовать знак `*`. Например, `y ~ x1 * x2` транслируется в `y = a_0 + a_1 * a1 + a_2 * a2 + a_12 * a1 * a2`. Заметьте, что всякий раз, когда вы используете знак `*`, в модель включаются как взаимодействие, так и индивидуальные компоненты.

Для визуализации этих моделей нам понадобятся два новшества.

- У нас два предиктора, поэтому мы должны предоставить функции `data_grid()` обе переменные. Эта функция найдет все уникальные значения `x1` и `x2` и сгенерирует все их комбинации.
- Чтобы сгенерировать прогнозы на основе сразу двух моделей, можно использовать функцию `gather_predictions()`, которая добавляет каждый прогноз в качестве строки. Комплементарной к функции `gather_predictions()` является функция `spread_predictions()`, которая добавляет каждый прогноз в новый столбец.

Вместе это дает нам следующие результаты.

```
grid <- sim3 %>%
  data_grid(x1, x2) %>%
  gather_predictions(mod1, mod2)
grid
#> # A tibble: 80 x 4
#>   model    x1     x2  pred
#>   <chr> <int> <fctr> <dbl>
#> 1  mod1     1      a  1.67
#> 2  mod1     1      b  4.56
#> 3  mod1     1      c  6.48
#> 4  mod1     1      d  4.03
#> 5  mod1     2      a  1.48
#> 6  mod1     2      b  4.37
#> # ... with 74 more rows
```

Результаты для обеих моделей можно визуализировать на одном графике, отображая их на разных панелях (рис. 18.15).

```
ggplot(sim3, aes(x1, y, color = x2)) +
  geom_point() +
  geom_line(data = grid, aes(y = pred)) +
  facet_wrap(~ model)
```

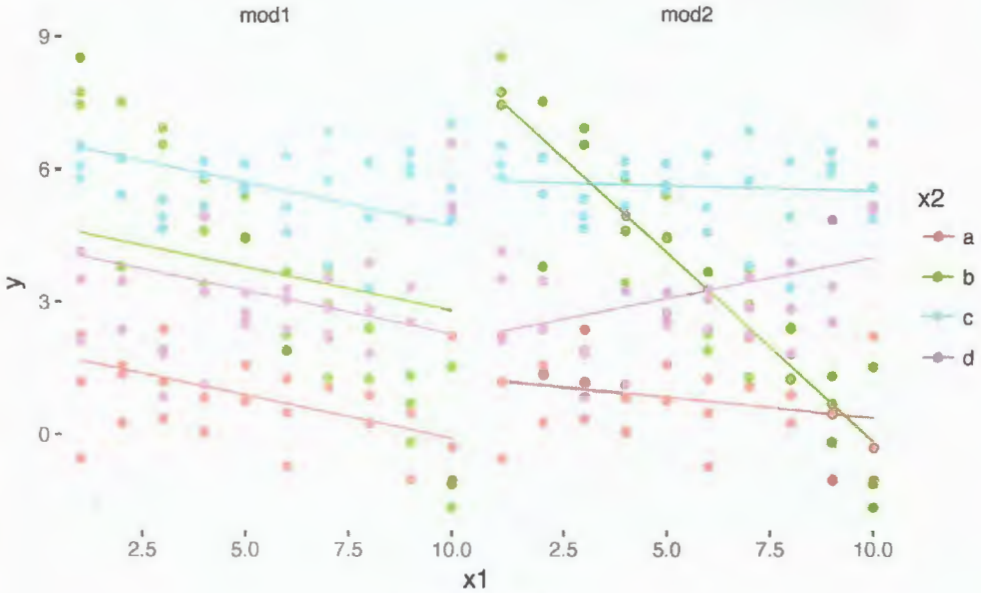


Рис. 18.15

Обратите внимание на то, что модель, в которой используется знак +, характеризуется одним и тем же наклоном всех прямых, но разными точками y-пересечения. В случае же модели, в которой используется знак *, и наклон, и точка *y*-пересечения разные для разных линий.

Какая модель лучше описывает рассматриваемый набор данных? Чтобы ответить на этот вопрос, мы можем рассмотреть остатки. В данном случае мы создали срезы (панели) как по модели, так и по x2, поскольку так проще заметить закономерности в каждой группе (рис. 18.16).

```
sim3 <- sim3 %>%
  gather_residuals(mod1, mod2)

ggplot(sim3, aes(x1, resid, color = x2)) +
  geom_point() +
  facet_grid(model ~ x2)
```

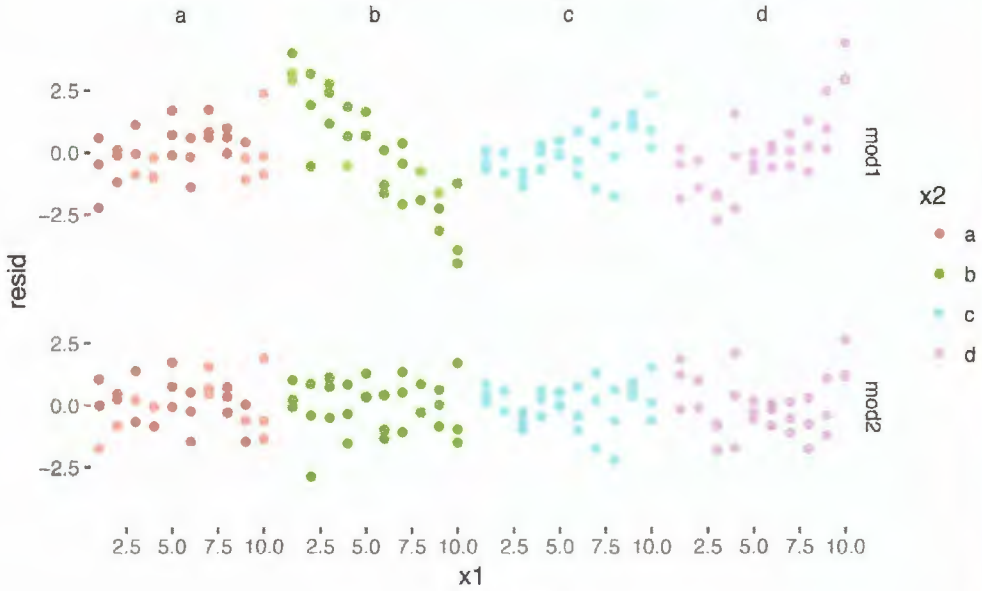


Рис. 18.16

В остатках для mod2 просматривается слабая закономерность. Остатки для mod1 демонстрируют, что модель явно упустила некоторую закономерность в b, которая, хотя и в меньшей степени, присутствует в c и d. Возможно, вы хотели бы знать, существует ли точный способ, позволяющий определить, какая из моделей, mod1 или mod2, лучше. Такой способ действительно существует, но он требует серьезной математической подготовки, и здесь мы его

- `trim = 0.1` отсекает 10% хвостовых значений. Это полезно, если распределение имеет длинные хвосты и вы хотите сосредоточиться на генерировании значений вблизи центра.

```
x1 <- rcauchy(100)
seq_range(x1, n = 5)
#> [1] -115.9  -83.5  -51.2  -18.8  13.5
seq_range(x1, n = 5, trim = 0.10)
#> [1] -13.84  -8.71  -3.58   1.55  6.68
seq_range(x1, n = 5, trim = 0.25)
#> [1] -2.1735 -1.0594  0.0547  1.1687  2.2828
seq_range(x1, n = 5, trim = 0.50)
#> [1] -0.725 -0.268  0.189  0.647  1.104
```

- `expand = 0.1` производит эффект, в некотором смысле противоположный эффекту `trim`, приводя к расширению диапазона на 10%.

```
x2 <- c(0, 1)
seq_range(x2, n = 5)
#> [1] 0.00 0.25 0.50 0.75 1.00
seq_range(x2, n = 5, expand = 0.10)
#> [1] -0.050  0.225  0.500  0.775  1.050
seq_range(x2, n = 5, expand = 0.25)
#> [1] -0.125  0.188  0.500  0.812  1.125
seq_range(x2, n = 5, expand = 0.50)
#> [1] -0.250  0.125  0.500  0.875  1.250
```

Попробуем визуализировать эту модель. У нас имеются два непрерывных предиктора, поэтому мы можем представить модель в виде трехмерной поверхности. Для ее отображения можно воспользоваться функцией `geom_tile()` (рис. 18.17).

```
ggplot(grid, aes(x1, x2)) +
  geom_tile(aes(fill = pred)) +
  facet_wrap(~ model)
```

Из результатов не следует, что модели сильно различаются! Отчасти это оптическая иллюзия: наши глаза и мозг недостаточно чувствительны, если речь идет о различении цветовых оттенков. Вместо того чтобы рассматривать поверхность сверху, мы можем посмотреть на нее с любой стороны и продемонстрировать не один срез (рис. 18.18 и 18.19).

```
ggplot(grid, aes(x1, pred, color = x2, group = x2)) +
  geom_line() +
  facet_wrap(~ model)
```

```
ggplot(grid, aes(x2, pred, color = x1, group = x1)) +
  geom_line() +
  facet_wrap(~ model)
```

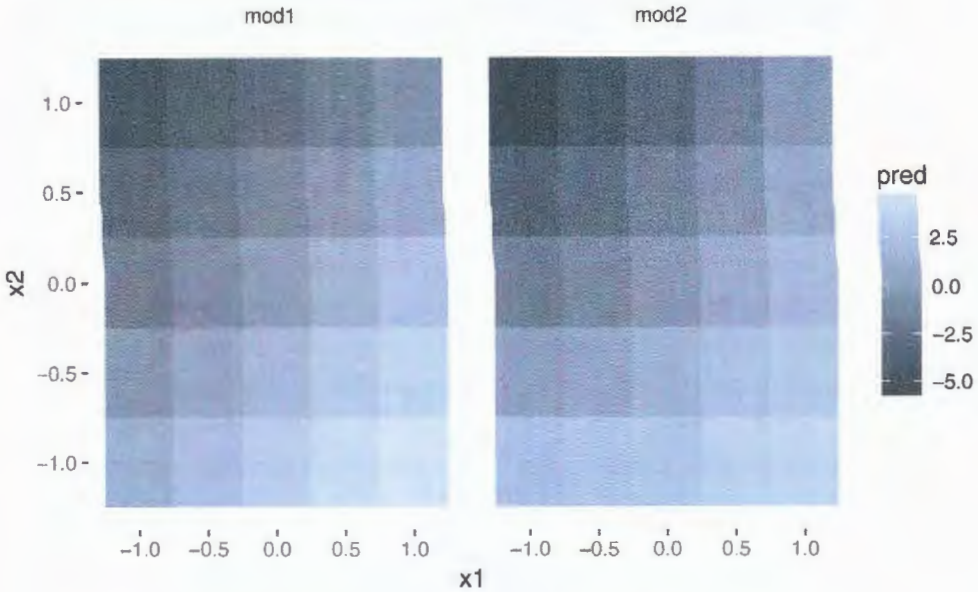


Рис. 18.17

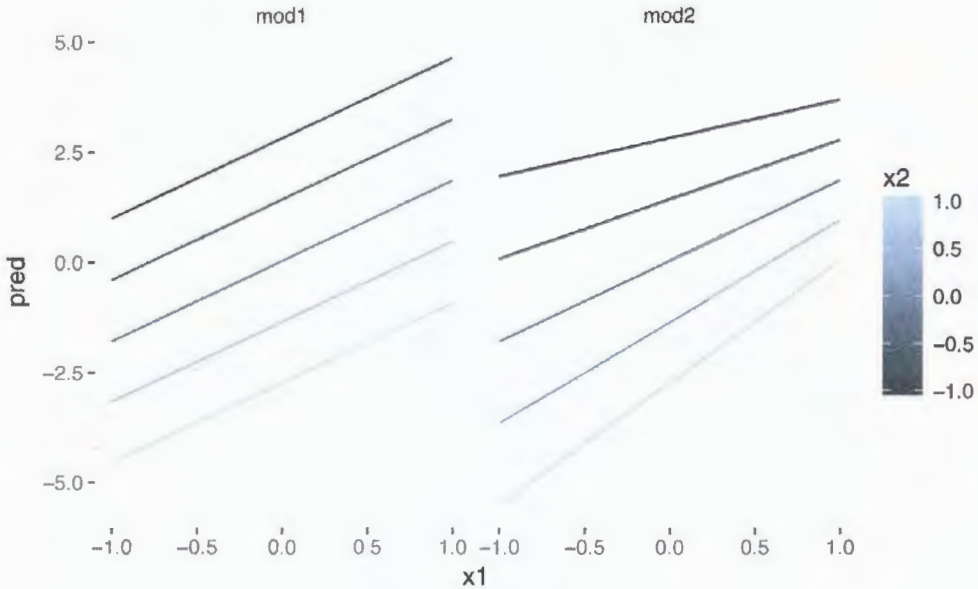


Рис. 18.18

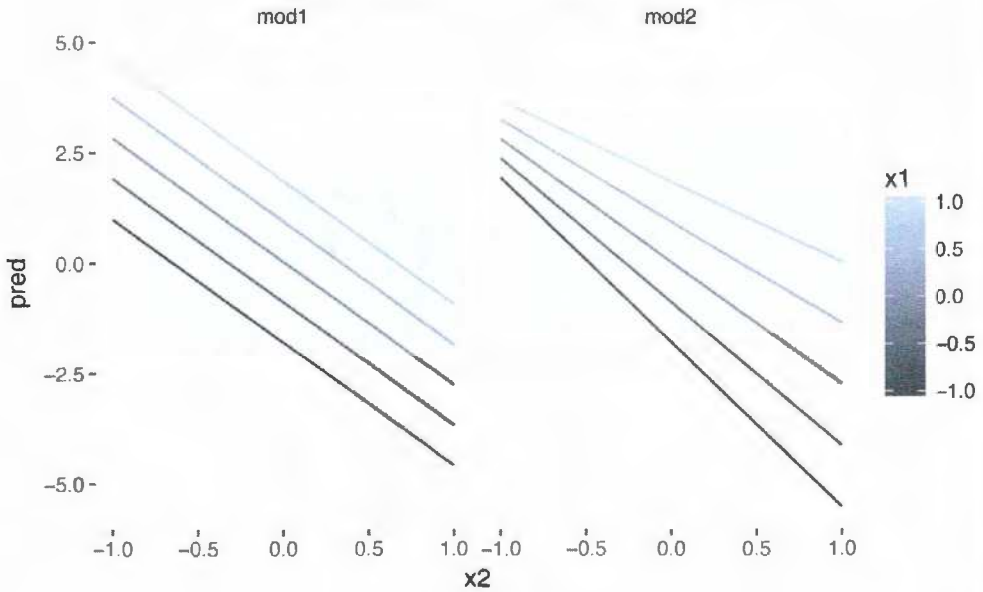


Рис. 18.19

Как свидетельствуют результаты, взаимодействие двух непрерывных переменных работает в основном так же, как и взаимодействие категориальной и непрерывной переменных. Взаимодействие указывает на отсутствие фиксированного сдвига: для того чтобы прогнозировать значения `y`, необходимо рассматривать одновременно значения `x1` и `x2`.

Вы видите, что даже в случае всего лишь двух непрерывных переменных получение хорошей визуализации — нелегкая задача. Но это вполне объяснимо: вряд ли можно было ожидать, что понять, каким образом взаимодействуют одновременно три и более переменных, будет нетрудно! Однако нас немного спасает то, что мы используем модели для разведочного анализа и можем постепенно усложнять модель с течением времени. Модель не должна быть идеальной, она должна лишь помогать обнаруживать пусть даже небольшую дополнительную информацию о ваших данных.

Мы посвятили некоторое время рассмотрению остатков, чтобы посмотреть, сможем ли мы определить, что `mod2` работает лучше, чем `mod1`. Судя по всему, это так, но различие между ними довольно незначительное. У вас еще будет возможность поработать над этим, выполняя упражнения.

Преобразования

В формуле модели также можно выполнять преобразования. Например, `log(y) ~ sqrt(x1) + x2` преобразуется в `y = a_1 + a_2 * x1 * sqrt(x) + a_3 * x2`. Если ваше преобразование включает операторы +, *, ^ или –, то те части формулы, в которых эти операторы используются в качестве арифметических, необходимо обернуть в функцию `I()`, чтобы R не обрабатывал их как символические операторы спецификации модели. Например, `y ~ x + I(x ^ 2)` транслируется в `y = a_1 + a_2 * x + a_3 * x^2`. Если забыть о функции `I()` и записать формулу в виде `y ~ x ^ 2 + x`, то R будет вычислять ее как `y ~ x * x + x`. Выражение `x * x` означает взаимодействие переменной `x` с самой собой, что является тем же самым, что `x`. R автоматически опускает избыточные переменные , поэтому `x + x` становится `x`, и, таким образом, `y ~ x ^ 2 + x` будет определять функцию `y = a_1 + a_2 * x`. Это, вероятно, не то, что вы намеревались сделать!

Опять-таки, если вас смущает то, что делает ваша модель, вы всегда можете воспользоваться функцией `model_matrix()` и посмотреть, в соответствии с каким уравнением функция `lm()` строит модель.

```
df <- tribble(
  ~y, ~x,
  1,  1,
  2,  2,
  3,  3
)
model_matrix(df, y ~ x^2 + x)
#> # A tibble: 3 x 2
#>   `(Intercept)`     x
#>           <dbl> <dbl>
#> 1             1     1
#> 2             1     2
#> 3             1     3
model_matrix(df, y ~ I(x^2) + x)
#> # A tibble: 3 x 3
#>   `(Intercept)` `I(x^2)`     x
#>           <dbl>     <dbl> <dbl>
#> 1             1         1     1
#> 2             1         4     2
#> 3             1         9     3
```

Преобразования полезны тем, что их можно использовать для аппроксимации нелинейных функций. Если вы проходили дифференциальное исчисление, то, возможно, слышали о теореме Тейлора, в соответствии с которой

любую гладкую функцию можно аппроксимировать бесконечной суммой многочленов. Это означает, что вы можете использовать линейную функцию для получения близкой аппроксимации к гладкой функции, строя модель с помощью уравнения наподобие `y = a_1 + a_2 * x + a_3 * x^2 + a_4 * x ^ 3`. Вводить вручную такие длинные последовательности утомительно, в связи с чем для этого в R предусмотрена вспомогательная функция `poly()`.

```
model_matrix(df, y ~ poly(x, 2))
#> # A tibble: 3 x 3
#>   `(Intercept)` `poly(x, 2)1` `poly(x, 2)2`
#>           <dbl>         <dbl>         <dbl>
#> 1             1     -7.07e-01         0.408
#> 2             1     -7.85e-17        -0.816
#> 3             1      7.07e-01         0.408
```

Однако с использованием функции `poly()` связана одна серьезная проблема: за пределами диапазона данных значения полинома быстро стремятся к положительной или отрицательной бесконечности. Одна из безопасных альтернатив заключается в использовании натуральных сплайнов с помощью функции `splines::ns()`.

```
library(splines)
model_matrix(df, y ~ ns(x, 2))
#> # A tibble: 3 x 3
#>   `(Intercept)` `ns(x, 2)1` `ns(x, 2)2`
#>           <dbl>       <dbl>       <dbl>
#> 1             1       0.000       0.000
#> 2             1       0.566      -0.211
#> 3             1       0.344       0.771
```

Посмотрим, как это будет выглядеть при попытке аппроксимации нелинейной функции (рис. 18.20).

```
sim5 <- tibble(
  x = seq(0, 3.5 * pi, length = 50),
  y = 4 * sin(x) + rnorm(length(x))
)

ggplot(sim5, aes(x, y)) +
  geom_point()
```

Мы собираемся построить пять моделей для этих данных (рис. 18.21).

```
mod1 <- lm(y ~ ns(x, 1), data = sim5)
mod2 <- lm(y ~ ns(x, 2), data = sim5)
mod3 <- lm(y ~ ns(x, 3), data = sim5)
```

```
mod4 <- lm(y ~ ns(x, 4), data = sim5)
mod5 <- lm(y ~ ns(x, 5), data = sim5)

grid <- sim5 %>%
  data_grid(x = seq_range(x, n = 50, expand = 0.1)) %>%
  gather_predictions(mod1, mod2, mod3, mod4, mod5, .pred = "y")

ggplot(sim5, aes(x, y)) +
  geom_point() +
  geom_line(data = grid, color = "red") +
  facet_wrap(~ model)
```

Рис. 18.20

Обратите внимание на явно низкую точность экстраполяции за пределами набора данных. Таков недостаток аппроксимации функций с помощью полиномов. Однако это становится реальной проблемой для всех моделей: ни одна модель не позволяет судить о том, будет ли найденное поведение истинным при экстраполяции за пределы видимых данных. В подобном случае вы должны полагаться на выводы научной теории.

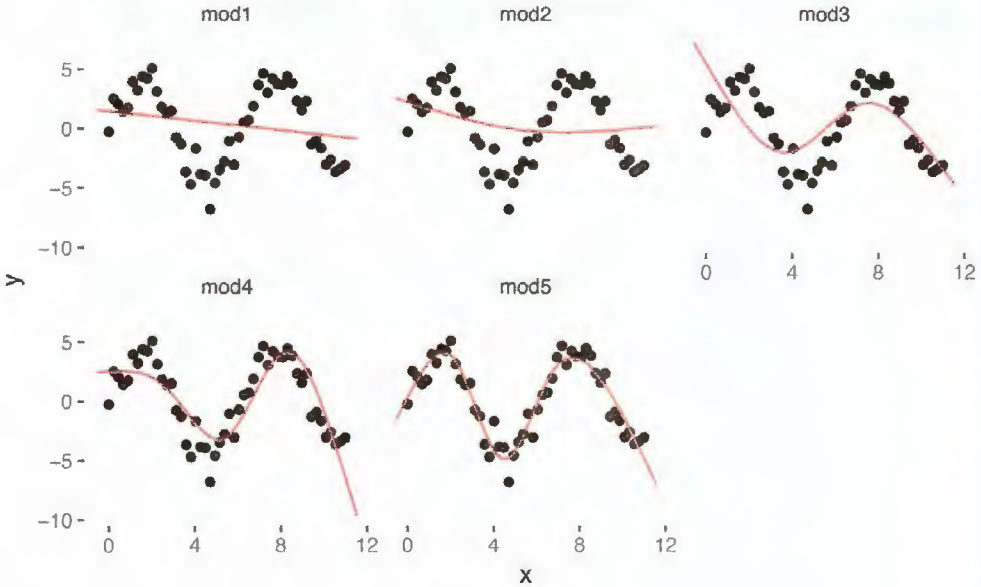


Рис. 18.21

Упражнения

1. Что произойдет, если повторить анализ `sim2`, используя модель без *y*-пересечения? Что произойдет с уравнением модели? Что произойдет с прогнозами?
2. Используйте функцию `model_matrix()` для исследования уравнений, сгенерированных для моделей, которые мы строили для наборов данных `sim3` и `sim4`. Почему оператор `*` является неплохим сокращением для взаимодействия?
3. Опираясь на базовые принципы, преобразуйте формулы двух приведенных ниже моделей в функции. (Подсказка: начните с преобразования категориальной переменной в переменные `0-1`.)

```
mod1 <- lm(y ~ x1 + x2, data = sim3)
mod2 <- lm(y ~ x1 * x2, data = sim3)
```

4. Какая из моделей, `mod1` и `mod2`, лучше подходит для набора данных `sim4`? Мы полагаем, что `mod2` лучше справляется с работой по удалению закономерностей поведения, но различия довольно слабые. Можете ли вы подтвердить графиком наше предположение?

Отсутствующие значения

Совершенно очевидно, что отсутствующие значения не могут сообщить никакой информации об отношениях между переменными, поэтому моделирующие функции будут пропускать любые строки, которые содержат отсутствующие значения. По умолчанию R не выводит при этом никаких сообщений, но вы можете задать их вывод с помощью команды `options(na.action = na.warn)`, выполнив ее перед началом работы.

```
df <- tribble(
  ~x, ~y,
  1, 2.2,
  2, NA,
  3, 3.5,
  4, 8.3,
  NA, 10
)
mod <- lm(y ~ x, data = df)
#> Warning: Dropping 2 rows with missing values
```

Для подавления вывода сообщений установите опцию `na.action = na.exclude`:

```
mod <- lm(y ~ x, data = df, na.action = na.exclude)
```

Вы всегда сможете узнать о том, сколько наблюдений было использовано, с помощью функции `nobs()`.

```
nobs(mod)
#> [1] 3
```

Другие семейства моделей

В этой главе мы сосредоточили внимание исключительно на классе линейных моделей, которые предполагают соотношение вида `y = a_1 * x1 + a_2 * x2 + ... + a_n * xn`. В случае линейных моделей дополнительно предполагается, что остатки характеризуются нормальным распределением, которое мы еще не обсуждали.

Существует большое разнообразие классов моделей, расширяющих модель всевозможными способами. Некоторые из них перечислены ниже.

- *Обобщенные линейные модели*, например `stats::glm()`. В линейных моделях предполагается, что отклик является непрерывным, а ошибки

подчиняются нормальному распределению. Обобщенные линейные модели расширяют линейные модели, дополнительно включая возможность отклика, не являющегося непрерывным (например, двоичные данные или счетчики). Их работа базируется на определении меры расстояния, основанной на статистической идее правдоподобия.

- *Обобщенные аддитивные модели*, например `mgcv::gam()`, расширяют обобщенные линейные модели, дополнительно включая произвольные гладкие функции. Это означает, что вы можете записать формулу наподобие `y ~ s(x)`, которая превращается в уравнение вида `y = f(x)` и предоставляет функции `gam()` возможность оценить, что собой представляет эта функция (при условии введения некоторых ограничений на гладкость, делающих возможным решение задачи).
- *Линейные модели со штрафами*, например `glmnet::glmnet()`, добавляют к расстоянию штрафное слагаемое, "наказывающее" слишком сложные модели. Как правило, это приводит к созданию более эффективных обобщенных моделей для новых выборок из той же генеральной совокупности.
- *Робастные линейные модели*, например `MASS:rlm()`, корректируют расстояние для уменьшения веса значительно удаленных точек данных. Это снижает чувствительность модели к наличию выбросов за счет некоторого ухудшения качества модели в отсутствие выбросов.
- *Деревья*, например `rpart::rpart()`, работают совершенно иначе, чем линейные модели. Они используют кусочно-постоянную модель, разделяя данные на все меньшие и меньшие части. Сами по себе деревья не отличаются особой эффективностью, но в сочетании с моделями типа "случайный лес" (например, `randomForest::randomForest()`) или моделями на основе машин градиентного бустинга (например, `xgboost::xgboost`), становятся весьма мощным средством.

С точки зрения программирования все эти модели работают одинаковым образом. Как только вы освоите линейные модели, вы обнаружите, что изучить модели других классов вам не составит большого труда. Умение создавать модели основывается на соблюдении ряда полезных общих принципов и владении широким набором соответствующих методик. Теперь, когда вы уже изучили часть общих инструментов и один полезный класс моделей, можете самостоятельно изучить другие модели, используя другие источники.

ГЛАВА 19

Создание модели

Введение

В предыдущей главе вы узнали о том, как работает линейная модель, и познакомились с некоторыми инструментами, позволяющими обрабатывать информацию, которую предоставляет линейная модель относительно ваших данных. Работа моделей изучалась с использованием фиктивных наборов данных. В этой главе для демонстрации процесса постепенного построения модели используются реальные данные.

Мы воспользуемся преимуществами того факта, что модели позволяют представлять данные посредством шаблонов, т.е. тенденций поведения, и остатков. Мы будем обнаруживать тенденции посредством визуализации, а затем конкретизировать и уточнять их с помощью модели. Затем мы повторим процесс, но заменим старую переменную отклика остатками, определенными из модели. Нашей целью является переход от неявных знаний, содержащихся в данных и наших головах, к явным знаниям, содержащимся в количественной модели. Это облегчит их применение в новых областях и использование другими людьми.

Для больших и сложных наборов данных это означает необходимость выполнения значительного объема работы. Безусловно, существуют альтернативы — в одном из таких подходов, основанном на методах машинного обучения, акцент переносится на предсказательные возможности модели. Можно сказать, что эти альтернативные подходы работают по принципу "черного ящика": модель дает действительно хорошие прогнозы, но вы не знаете, почему так происходит. Такой подход вполне разумен, однако он затрудняет применение к модели наших знаний реального мира. В свою очередь, это осложняет оценку того, будет ли полученная модель работать в долгосрочной перспективе, если изменятся основные предпосылки. Мы считаем, что

в большинстве реальных моделей целесообразно использовать какую-то комбинацию этого и более классического автоматизированного подхода.

Очень важно знать, когда следует остановиться. Вы должны научиться определять тот момент, когда модель уже достаточно хороша и маловероятно, чтобы дальнейшие затраты труда себя окупили. На сайте Reddit пользователем `Broseidon241` было опубликовано хорошее сообщение.

> Давным-давно учитель художественной школы сказал мне: "Художник должен чувствовать, когда произведение готово. Ты не можешь бесконечно доводить картину до совершенства — заканчивай. Если тебе не нравится то, что получилось, сделай все заново или же начни писать что-то новое". Много позже мне как-то довелось услышать такое высказывание: "Плохая швея допускает много огрехов. Хорошая швея усердно исправляет свои огрехи. Отличная швея не боится выбросить неудачную работу и начать шить заново".

Необходимые ресурсы

Мы будем пользоваться теми же инструментами, что и в предыдущей главе, но добавим реальные наборы данных: `diamonds` из пакета **ggplot2** и `flights` из пакета **nycflights13**. Кроме того, для работы с датами и временем из расписания авиарейсов нам понадобится библиотека *lubridate*.

```
library(tidyverse)
library(modelr)
options(na.action = na.warn)

library(nycflights13)
library(lubridate)
```

Почему бриллианты низкого качества оказались более дорогими

В предыдущих главах мы наблюдали неожиданное соотношение между качеством бриллиантов и их ценой: бриллианты низкого качества (плохая огранка, неудачный цвет и худшая чистота) стоят дороже (рис. 19.1–19.3).

```
ggplot(diamonds, aes(cut, price)) + geom_boxplot()
ggplot(diamonds, aes(color, price)) + geom_boxplot()
ggplot(diamonds, aes(clarity, price)) + geom_boxplot()
```

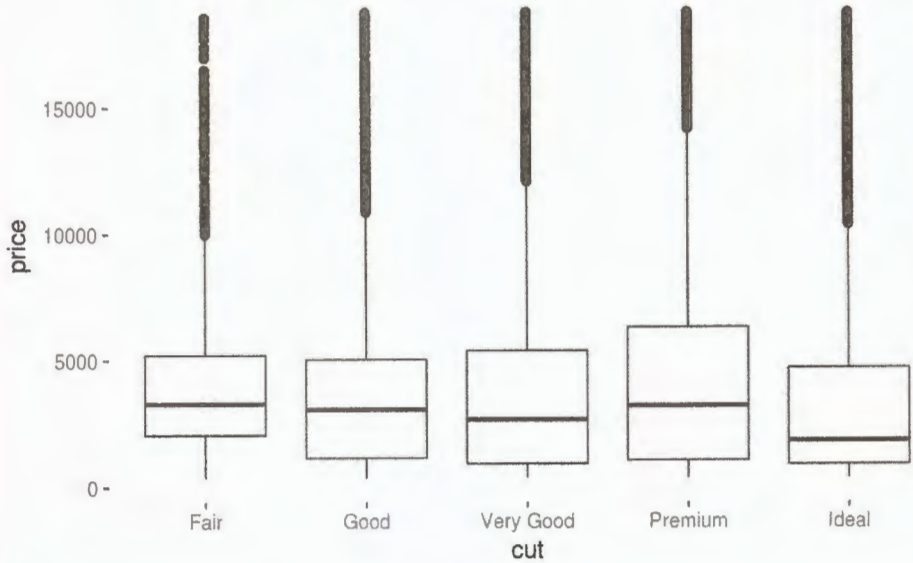


Рис. 19.1

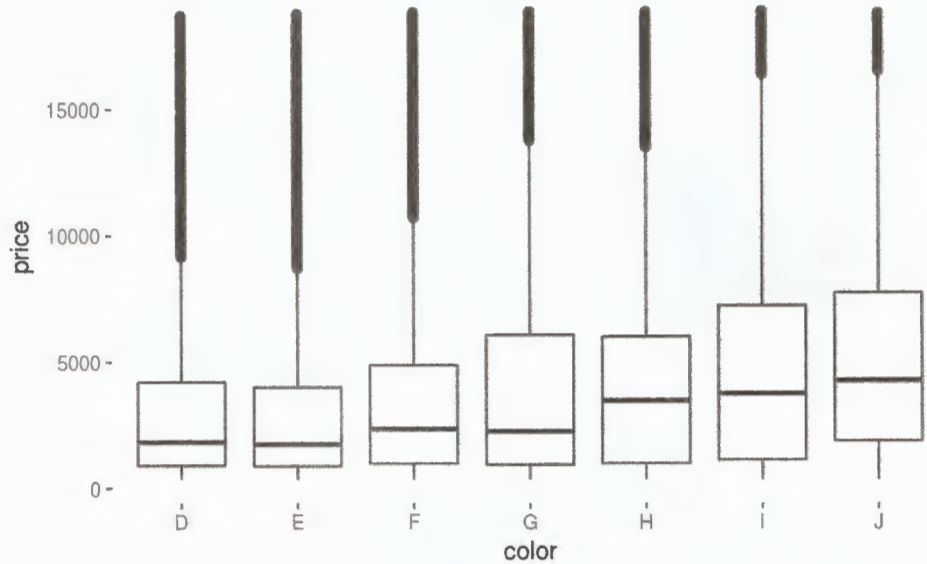


Рис. 19.2

Обратите внимание на то, что наихудшим цветом бриллиантов является J (желтоватый), а наихудшим классом чистоты — I1 (включения видны невооруженным глазом).

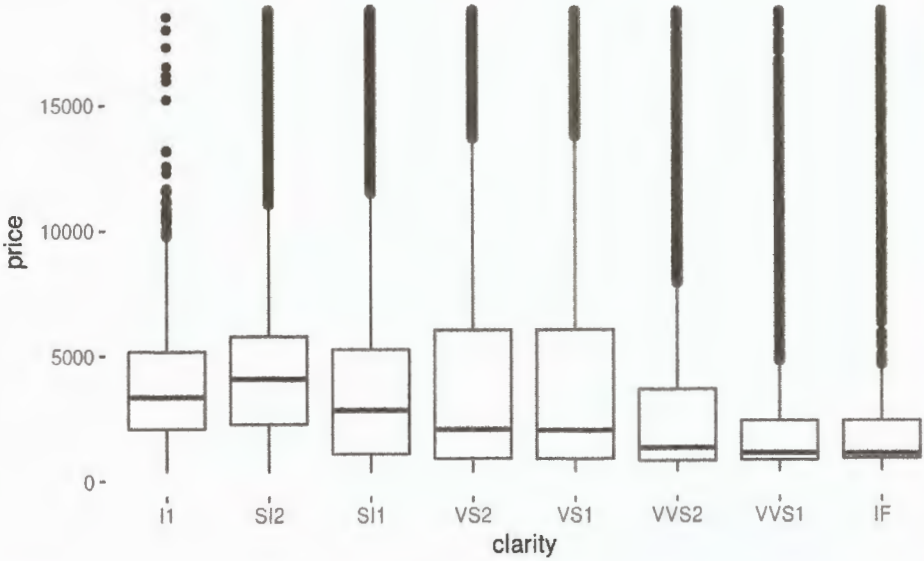


Рис. 19.3

Цена и вес

Похоже, цена низкокачественных бриллиантов оказалась выше из-за одной важной переменной, которая смешала все карты: `carat` (вес в каратах). Вес бриллианта — это единственно важный фактор для определения его цены, а низкокачественные бриллианты имеют, как правило, бо́льшие размеры (рис. 19.4).

```
ggplot(diamonds, aes(carat, price)) +
  geom_hex(bins = 50)
```

Нам будет проще увидеть, как другие атрибуты бриллианта влияют на его относительную цену, если мы построим модель, отделяющую эффект переменной `carat`. Но сначала немного подправим набор данных `diamonds`, чтобы с ним было легче работать.

1. Сфокусируем внимание на бриллиантах весом меньше 2,5 карата (99,7% данных).
2. Перейдем к логарифмам переменных `carat` и `price`.

```
diamonds2 <- diamonds %>%
  filter(carat <= 2.5) %>%
  mutate(lprice = log2(price), lcarat = log2(carat))
```

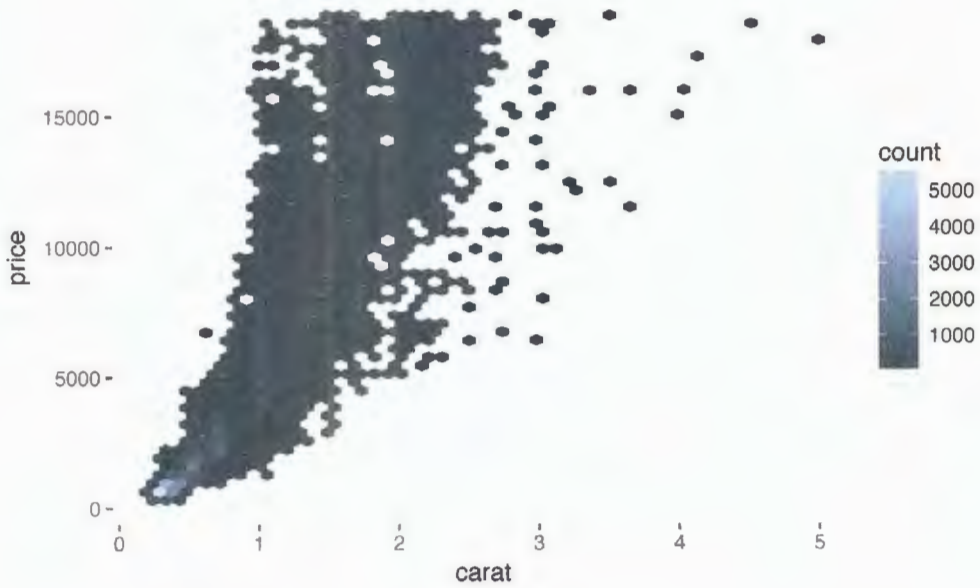


Рис. 19.4

Совместное внесение этих изменений упрощает выявление взаимосвязи переменных carat и price (рис. 19.5).

```
ggplot(diamonds2, aes(lcarat, lprice)) +
  geom_hex(bins = 50)
```

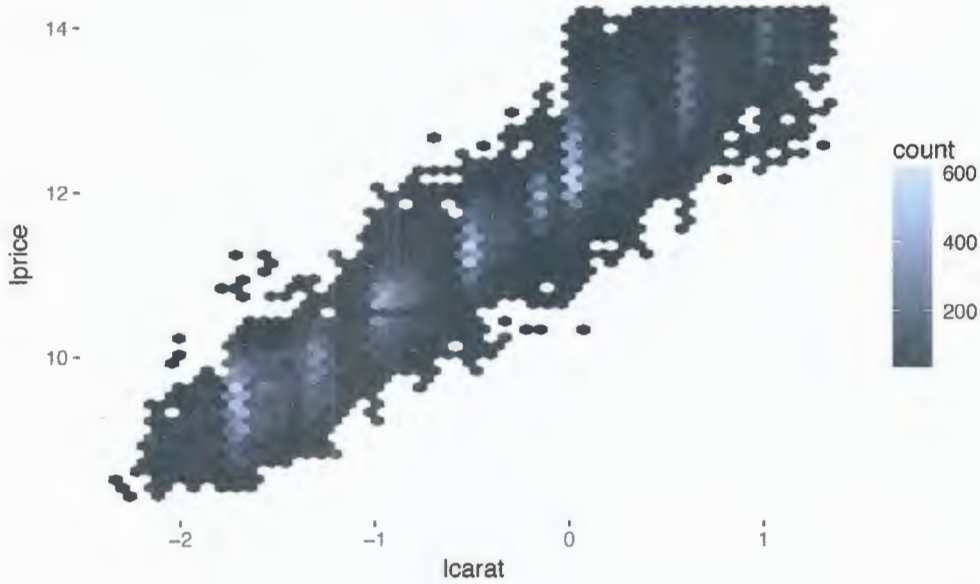


Рис. 19.5

В данном случае логарифмическое преобразование особенно удобно, поскольку оно линеаризует шаблон, а с линейными шаблонами легче работать. Сделаем еще один шаг и исключим эту ярко выраженную линейную тенденцию. Сначала мы явно создаем шаблон путем построения модели:

```
mod_diamond <- lm(lprice ~ lcarat, data = diamonds2)
```

Затем мы смотрим, что говорит о данных эта модель. Заметьте, что мы выполняем обратное преобразование предсказаний, отменяя логарифмическое преобразование, чтобы предсказания можно было наложить на исходные данные (рис. 19.6).

```
grid <- diamonds2 %>%
  data_grid(carat = seq_range(carat, 20)) %>%
  mutate(lcarat = log2(carat)) %>%
  add_predictions(mod_diamond, "lprice") %>%
  mutate(price = 2 ^ lprice)

ggplot(diamonds2, aes(carat, price)) +
  geom_hex(bins = 50) +
  geom_line(data = grid, color = "red", size = 1)
```

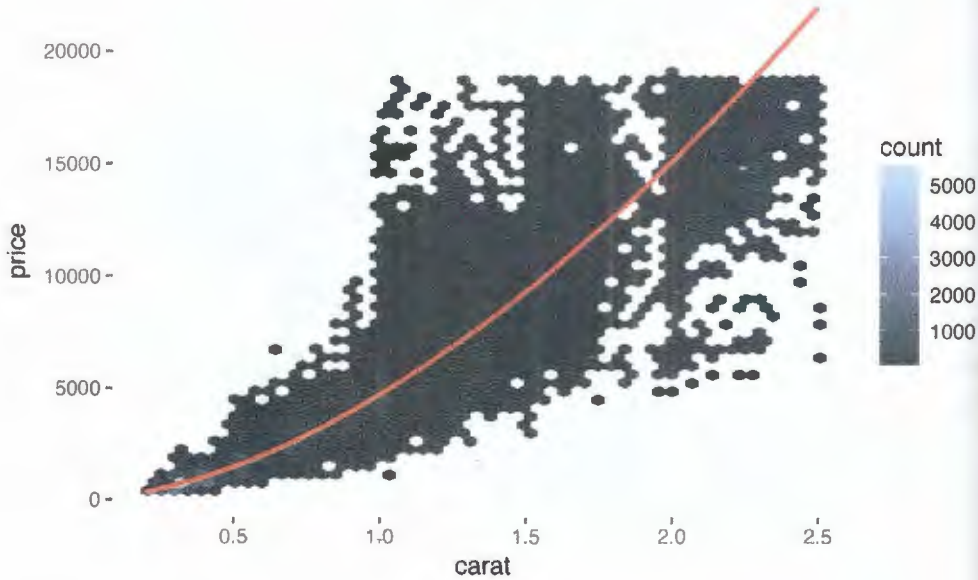


Рис. 19.6

В результате получаем интересную информацию о данных. Если мы доверяем нашей модели, то большие бриллианты намного дешевле, чем можно

было ожидать. Вероятно, это объясняется тем, что ни один бриллиант в данном наборе не стоит более 19000 долларов.

Рассмотрение остатков подтверждает, что мы успешно исключили ярко выраженный линейный шаблон (рис. 19.7).

```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond, "lresid")

ggplot(diamonds2, aes(lcarat, lresid)) +
  geom_hex(bins = 50)
```

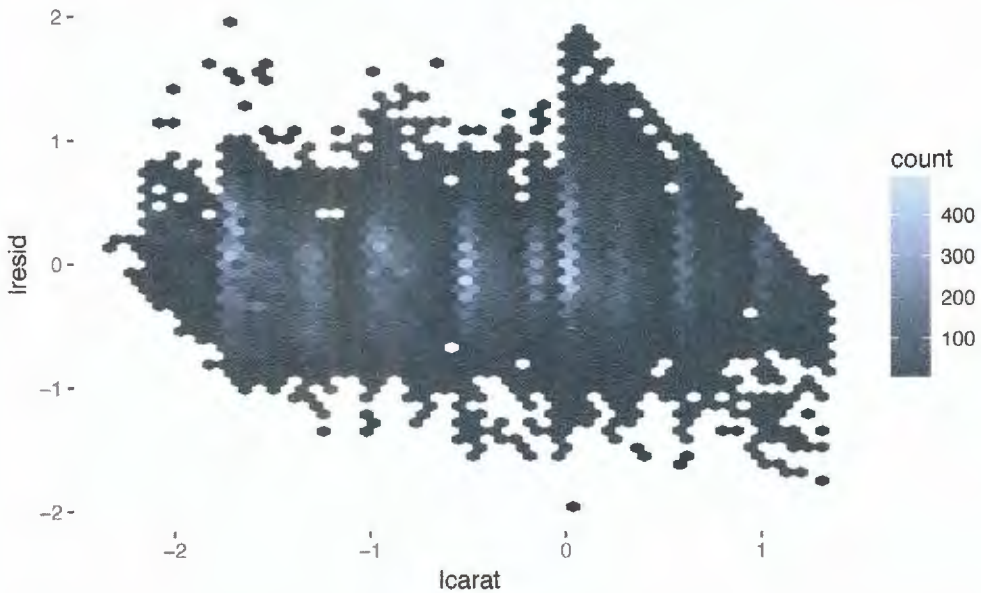


Рис. 19.7

Важно то, что теперь мы можем заново построить наши мотивирующие графики, используя остатки вместо переменной `price` (рис. 19.8–19.10).

```
ggplot(diamonds2, aes(cut, lresid)) + geom_boxplot()
ggplot(diamonds2, aes(color, lresid)) + geom_boxplot()
ggplot(diamonds2, aes(clarity, lresid)) + geom_boxplot()
```

Теперь мы видим ожидаемую взаимосвязь: по мере улучшения качества бриллиантов растет их относительная цена. Для интерпретации оси *y* необходимо подумать о том, о чем нам говорят остатки и по какой шкале они измеряются. Остаток −1 указывает на то, что переменная `lprice` была на единицу ниже предсказанного значения, если основываться исключительно на его весе. 2^{-1} равно $^1/_2$, поэтому точка со значением −1 соответствует половине

ожидаемой цены, а остатки со значением 1 соответствуют двойной предсказанной цене.

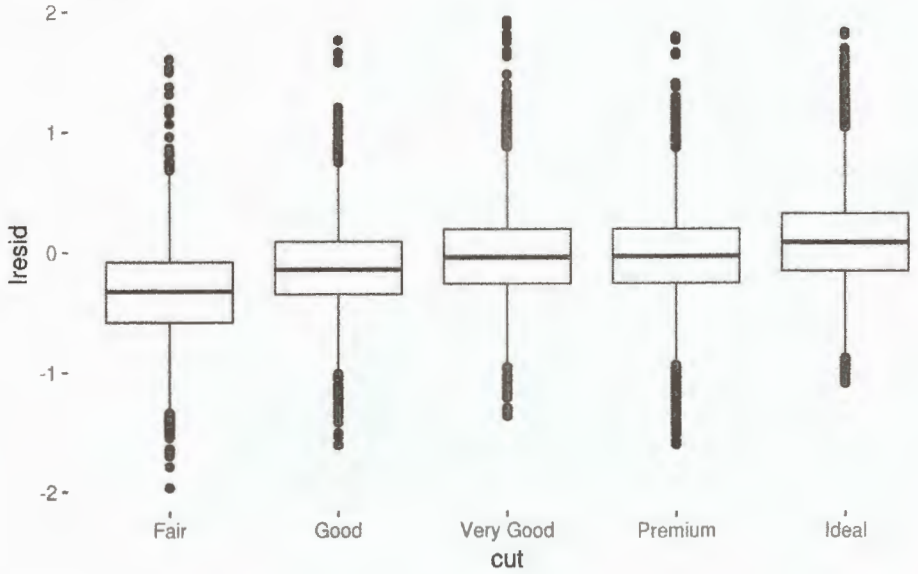


Рис. 19.8

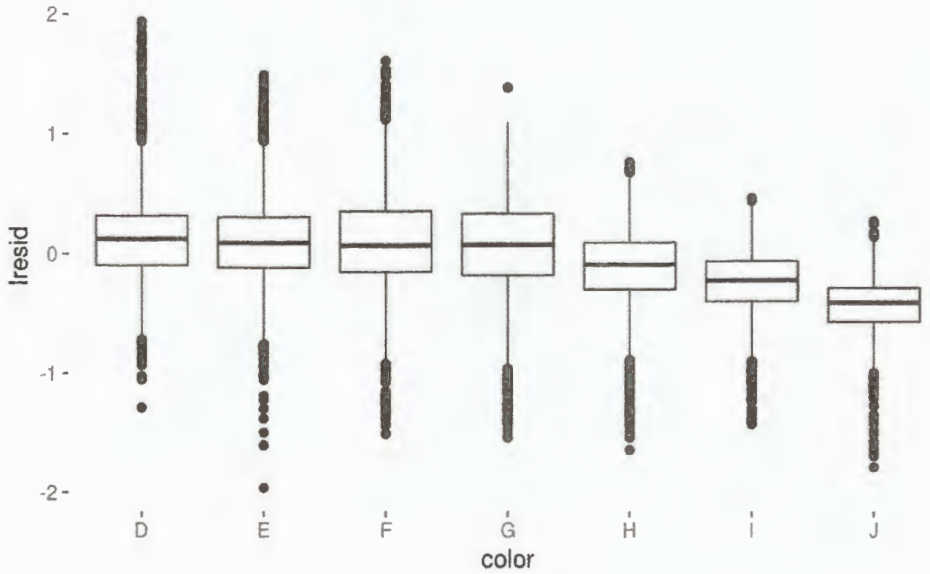


Рис. 19.9

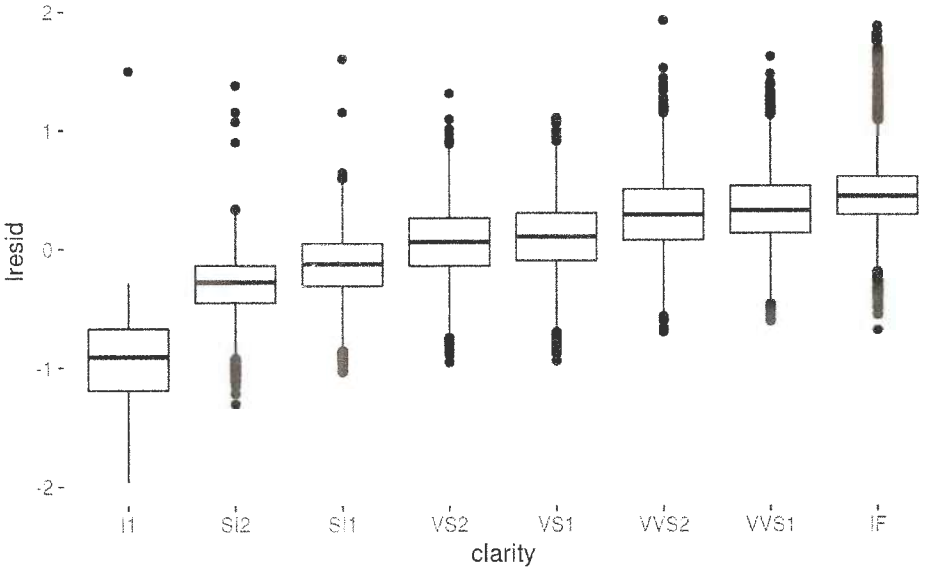


Рис. 19.10

Более сложная модель

При желании мы могли бы достраивать нашу модель, перемещая в нее наблюдаемые эффекты, чтобы они проявились более отчетливо. Например, мы могли бы включить в нее переменные `color`, `cut` и `clarity`, чтобы сделать явными эффекты этих трех категориальных переменных.

```
mod_diamond2 <- lm(
  lprice ~ lcarat + color + cut + clarity,
  data = diamonds2
)
```

Теперь модель включает четыре предиктора, и ее труднее визуализировать. К счастью, к настоящему моменту все они независимы, а это означает, что мы можем отобразить их по отдельности на четырех графиках (рис. 19.11). Чтобы немного упростить процесс, мы используем аргумент `.model` функции `data_grid()`.

```
grid <- diamonds2 %>%
  data_grid(cut, .model = mod_diamond2) %>%
  add_predictions(mod_diamond2)
grid
#> # A tibble: 5 x 5
#>         cut lcarat color clarity  pred
```

```
#>       <ord>  <dbl> <chr>   <chr> <dbl>
#> 1      Fair -0.515     G     SI1  11.0
#> 2      Good -0.515     G     SI1  11.1
#> 3 Very Good -0.515     G     SI1  11.2
#> 4   Premium -0.515     G     SI1  11.2
#> 5     Ideal -0.515     G     SI1  11.2
```

```
ggplot(grid, aes(cut, pred)) +
  geom_point()
```

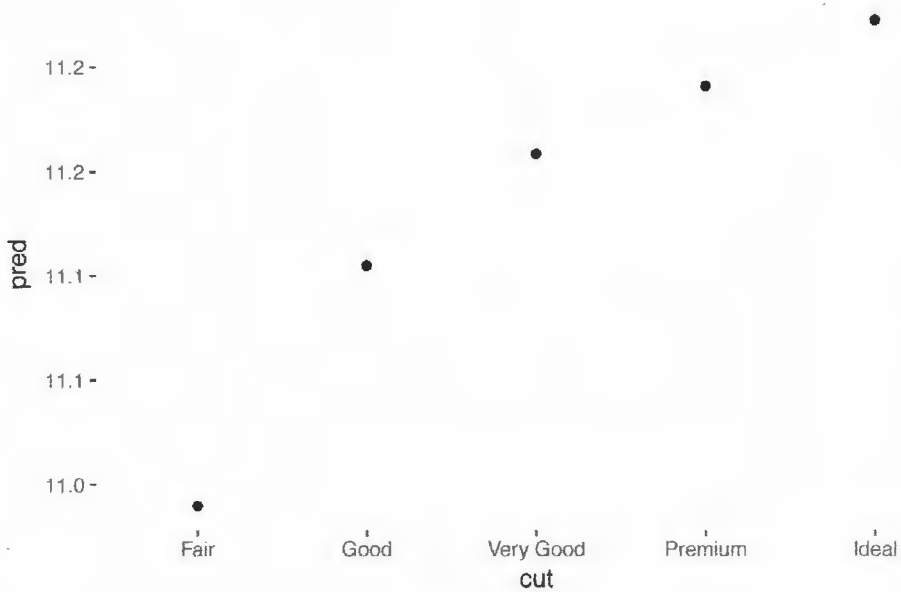


Рис. 19.11

Если модель нуждается в переменных, которые вы не предоставили ей явно, функция `data_grid()` автоматически заполнит их типичным значением. Для непрерывных переменных она использует медиану, а для категориальных — наиболее часто встречающееся значение (или значения, если имеется несколько таких связанных значений), как показано на рис. 19.12.

```
diamonds2 <- diamonds2 %>%
  add_residuals(mod_diamond2, "lresid2")
  ggplot(diamonds2, aes(lcarat, lresid2)) +
  geom_hex(bins = 50)
```

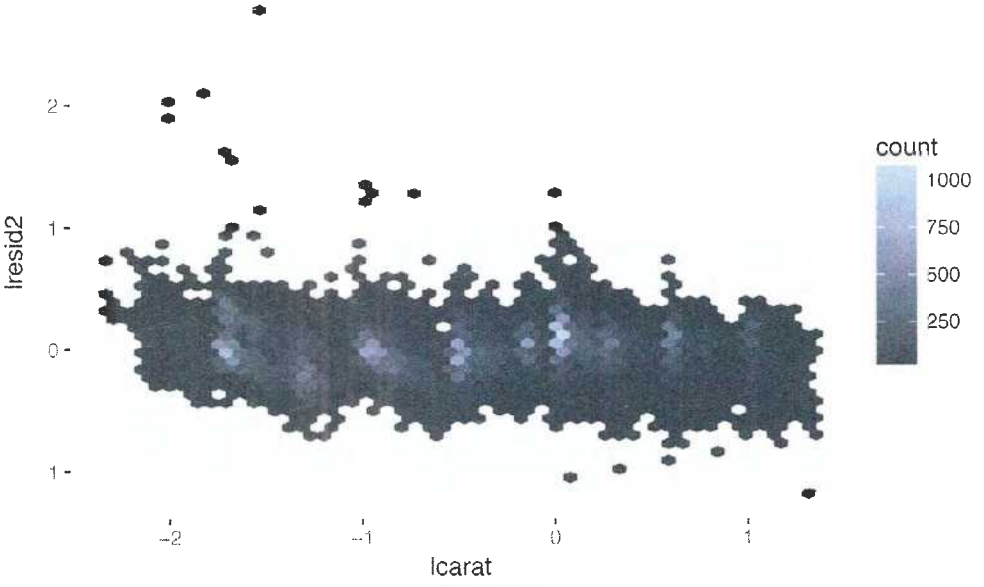


Рис. 19.12

Этот график указывает на то, что имеются бриллианты с довольно большими остатками: вспомните о том, что остаток 2 означает четырехкратное превышение цены по сравнению с ожидаемой. Часто целесообразно рассматривать необычные значения по отдельности.

```
diamonds2 %>%
  filter(abs(lresid2) > 1) %>%
  add_predictions(mod_diamond2) %>%
  mutate(pred = round(2 ^ pred)) %>%
  select(price, pred, carat:table, x:z) %>%
  arrange(price)
#> # A tibble: 16 x 11
#>   price  pred carat     cut color clarity depth table     x
#>   <int> <dbl> <dbl>   <ord> <ord>   <ord> <dbl> <dbl> <dbl>
#> 1  1013   264  0.25    Fair     F     SI2  54.4    64  4.30
#> 2  1186   284  0.25 Premium     G     SI2  59.0    60  5.33
#> 3  1186   284  0.25 Premium     G     SI2  58.8    60  5.33
#> 4  1262  2644  1.03    Fair     E      I1  78.2    54  5.72
#> 5  1415   639  0.35    Fair     G     VS2  65.9    54  5.57
#> 6  1415   639  0.35    Fair     G     VS2  65.9    54  5.57
#> # ... with 10 more rows, and 2 more variables: y <dbl>,
#> #   z <dbl>
```

В данном случае мы не находим ничего необычного, но, вероятно, стоило бы потратить какое-то время на выяснение того, не указывает ли это на

возможные проблемы с нашей моделью или на наличие ошибок в данных. В случае существования ошибок в данных это могло бы обернуться счастливой возможностью купить бриллианты по неоправданно заниженной цене.

Упражнения

1. На точечном графике для переменных `lcarat` и `lprice` выделяются яркие вертикальные полосы. Что они представляют?
2. Если `log(price) = a_0 + a_1 * log(carat)`, то что можно сказать о связи между переменными `price` и `carat`?
3. Извлеките данные о бриллиантах с очень большими и очень малыми остатками. Связано ли с этими бриллиантами что-либо необычное? Является ли качество этих бриллиантов высоким или низким, или вы считаете, что допущены ошибки в оценке?
4. Хорошо ли предсказывает цену бриллиантов окончательная модель, `mod_diamonds2`? Доверились бы вы ее предсказаниям относительно того, какую сумму вам следовало бы потратить, если бы вы покупали бриллианты?

Что влияет на количество ежедневных авиарейсов

Давайте проделаем аналогичные действия в отношении количества авиарейсов, ежедневно отправляемых из Нью-Йорка, т.е. в отношении набора данных, который на первый взгляд кажется еще более простым, чем предыдущий. Это действительно небольшой набор данных, включающий всего лишь 365 строк и 2 столбца, и мы не собираемся доводить процесс до полностью реализованной модели, но, как вы увидите далее, каждый наш шаг будет способствовать лучшему пониманию данных. Начнем с подсчета количества авиарейсов за день и визуализируем результаты с помощью **ggplot2** (рис. 19.13).

```
daily <- flights %>%
  mutate(date = make_date(year, month, day)) %>%
  group_by(date) %>%
  summarize(n = n())
daily
#> # A tibble: 365 x 2
#>         date     n
#>       <date> <int>
#> 1 2013-01-01   842
#> 2 2013-01-02   943
```

```
#> 3 2013-01-03   914
#> 4 2013-01-04   915
#> 5 2013-01-05   720
#> 6 2013-01-06   832
#> # ... with 359 more rows
```

```
ggplot(daily, aes(date, n)) +
  geom_line()
```

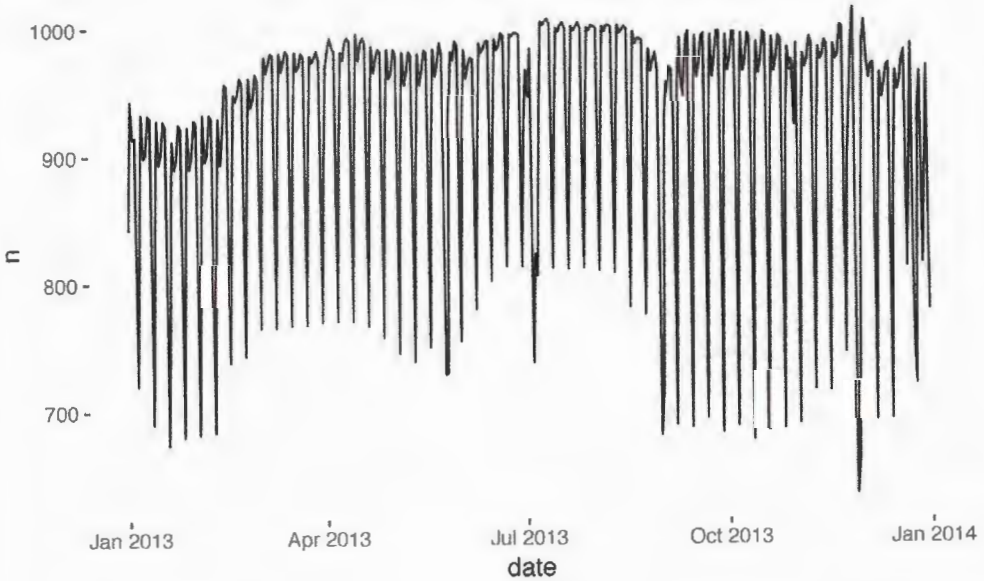


Рис. 19.13

День недели

Выявление долгосрочных тенденций — непростая задача из-за наличия сильного эффекта дня недели, который доминирует над более слабыми тенденциями. Начнем с рассмотрения распределения количества авиарейсов по дням недели (рис. 19.14).

```
daily <- daily %>%
  mutate(wday = wday(date, label = TRUE))
ggplot(daily, aes(wday, n)) +
  geom_boxplot()
```

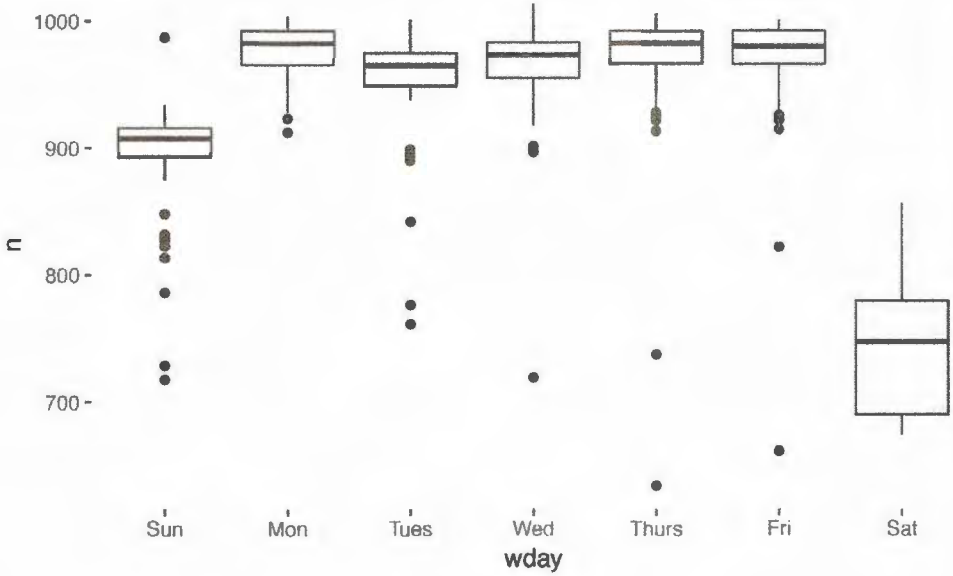


Рис. 19.14

В выходные дни полетов меньше, поскольку большинство людей путешествуют в деловых целях. Этот эффект особенно выражен по субботам: иногда вы можете вылететь в воскресенье, чтобы успеть на встречу, которая назначена на утро понедельника, но очень редко выберете для вылета субботу, поскольку предпочтете провести это время с семьей в домашней обстановке.

Один из способов исключения этой сильной тенденции — использовать модель. Сначала мы строим модель и отображаем ее предсказания поверх исходных данных (рис. 19.15).

```
mod <- lm(n ~ wday, data = daily)

grid <- daily %>%
  data_grid(wday) %>%
  add_predictions(mod, "n")

ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red", size = 4)
```

Затем вычисляем и визуализируем остатки (рис. 19.16).

```
daily <- daily %>%
  add_residuals(mod)
daily %>%
```

```
ggplot(aes(date, resid)) +
geom_ref_line(h = 0) +
geom_line()
```

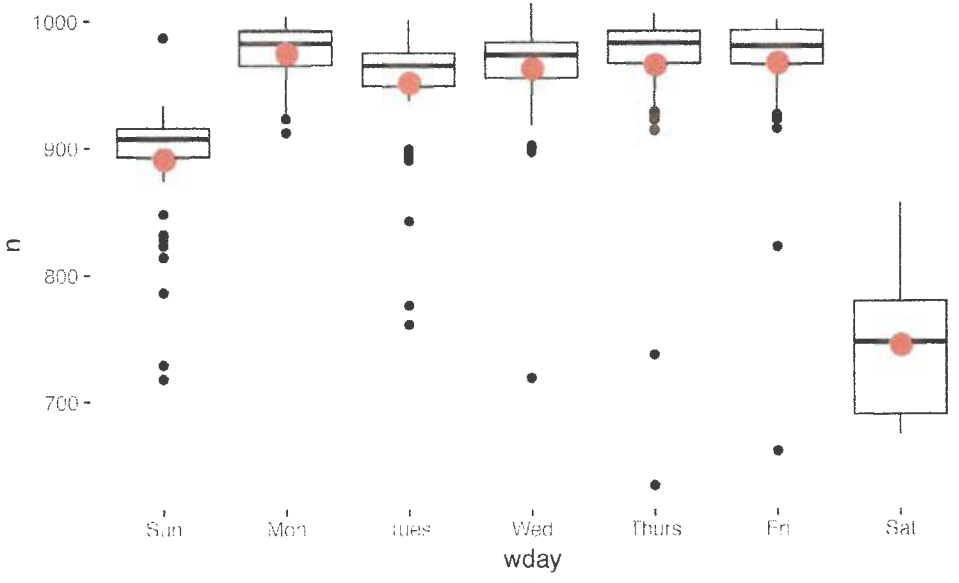


Рис. 19.15

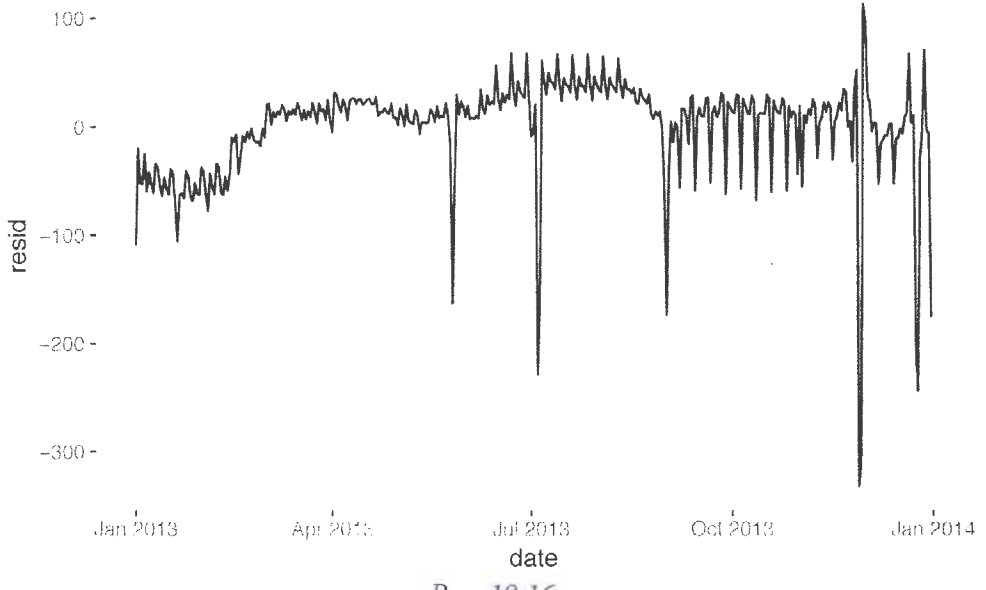


Рис. 19.16

Обратите внимание на изменение смысла оси *y*: теперь вдоль нее откладываются отклонения количества полетов от ожидаемого для разных дней недели. Этот график весьма полезен, поскольку теперь, когда из него исключена значительная часть сильного эффекта дня недели, мы можем видеть некоторые слабые оставшиеся тенденции.

- По-видимому, начиная с июня наша модель работает плохо: вы по-прежнему можете наблюдать сильный регулярный шаблон, который модели не удалось захватить. Вычерчивание графика с использованием отдельных линий для каждого дня недели позволяет лучше увидеть это (рис. 19.17).

```
ggplot(daily, aes(date, resid, color = wday)) +
  geom_ref_line(h = 0) +
  geom_line()
```

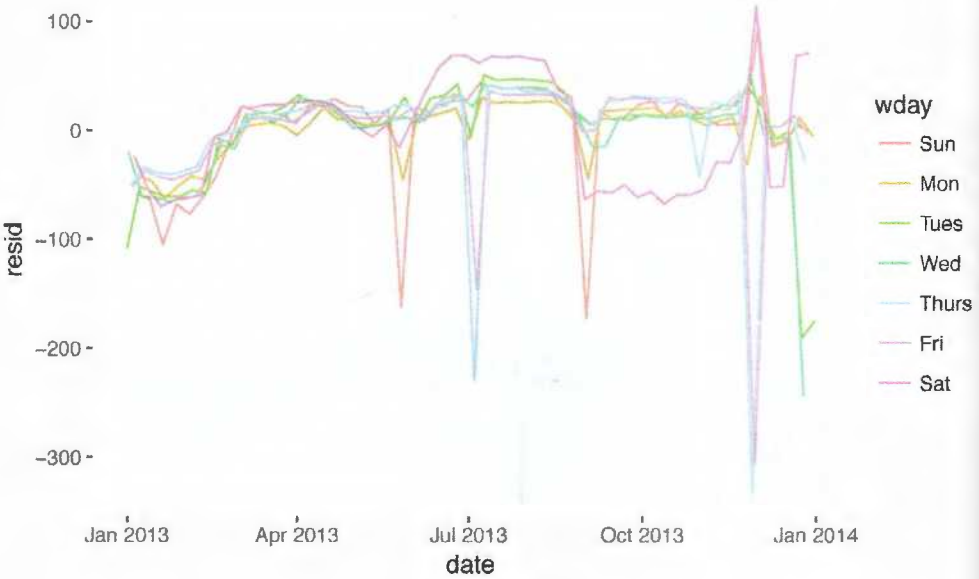


Рис. 19.17

Нашей модели не удается точно предсказать количество авиарейсов по субботам: летом количество полетов превышает ожидаемое значение, а осенью оно ниже ожидаемого значения. В следующем разделе будет показано, каким образом можно улучшить захват этого шаблона.

- Для некоторых дней количество авиарейсов значительно превышает ожидаемое значение.

```
daily %>%
  filter(resid < -100)
#> # A tibble: 11 x 4
#>         date     n  wday resid
#>       <date> <int> <ord> <dbl>
#> 1 2013-01-01   842  Tues  -109
#> 2 2013-01-20   786   Sun  -105
#> 3 2013-05-26   729   Sun  -162
#> 4 2013-07-04   737 Thurs  -229
#> 5 2013-07-05   822   Fri  -145
#> 6 2013-09-01   718   Sun  -173
#> # ... with 5 more rows
```

Если вам известны даты общенациональных праздников США, то вы, вероятно, обратите внимание на дни, которым соответствует Новый год, 4 июля, День благодарения и Рождество. Но обнаружены и другие дни, которые не соответствуют никаким праздникам. Вы поработаете над этим в одном из упражнений.

- Очевидно существование гладкого долгосрочного тренда на протяжении года. Этот тренд можно выделить с помощью функции `geom_smooth()` (рис. 19.18).

```
daily %>%
  ggplot(aes(date, resid)) +
  geom_ref_line(h = 0) +
  geom_line(color = "grey50") +
  geom_smooth(se = FALSE, span = 0.20)
#> `geom_smooth()` using method = 'loess'
```

В январе (и декабре) полетов меньше, а в летнее время (май–сентябрь) их больше. Количественная обработка данной тенденции даст нам немного, поскольку мы работаем с данными только за один год. Однако мы можем использовать свои знания в данной предметной области, чтобы выдвинуть возможные объяснения этого.

Сезонный субботний эффект

Прежде всего мы попытаемся справиться с проблемой точного предсказания количества субботних авиарейсов. Для начала вернемся к исходным данным, сосредоточившись на субботах (рис. 19.19).

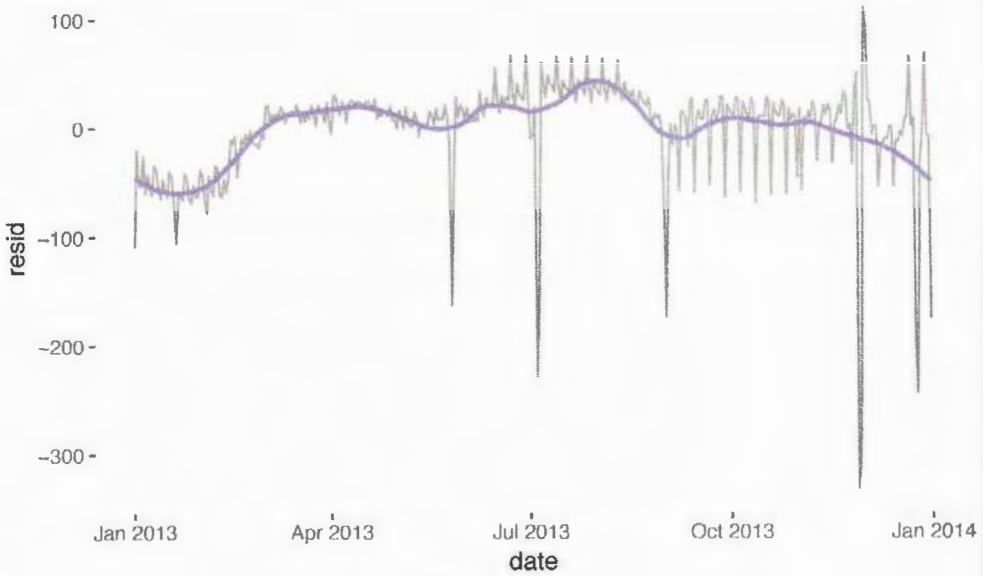


Рис. 19.18

```
daily %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n)) +
    geom_point() +
    geom_line() +
    scale_x_date(
      NULL,
      date_breaks = "1 month",
      date_labels = "%b"
    )
```

(При построении графика мы использовали как точки, так и линии, чтобы было отчетливо видно, где данные, а где результаты интерполяции.)

Полагаем, эта тенденция обусловлена летними отпусками: летом многие уезжают в отпуска, и нет никаких причин для того, чтобы не делать этого в субботу. Вид графика позволяет предположить, что период летних отпусков длится с начала июня до конца августа. Это довольно хорошо соответствует периодам обучения в школах: в 2013 году летние каникулы длились с 26 июня до 9 сентября.

Почему количество субботних авиарейсов весной больше, чем осенью? Мы задали этот вопрос кое-кому из наших американских друзей, и они предположили, что семейный отпуск планируют на осенний период не так часто

из-за того, что на этот период выпадают День благодарения и рождественские распродажи. Мы не располагаем надежными данными в подтверждение этого, но в качестве рабочей гипотезы это звучит довольно правдоподобно.

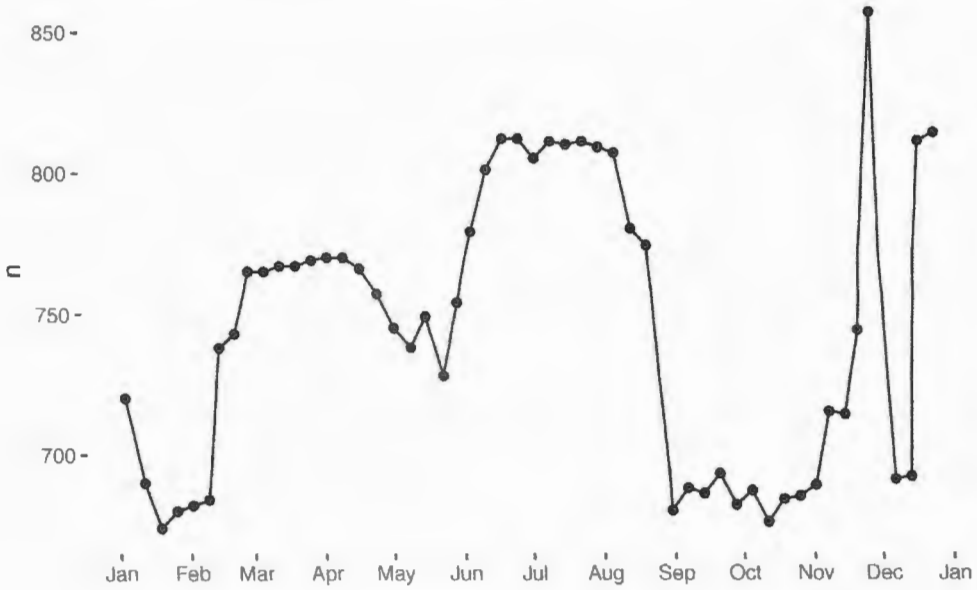


Рис. 19.19

Создадим переменную "term", которая охватывает три характерных школьных периода, и проверим нашу работу с помощью графика (рис. 19.20).

```
term <- function(date) {
  cut(date,
    breaks = ymd(20130101, 20130605, 20130825, 20140101),
    labels = c("spring", "summer", "fall")
  )
}

daily <- daily %>%
  mutate(term = term(date))

daily %>%
  filter(wday == "Sat") %>%
  ggplot(aes(date, n, color = term)) +
  geom_point(alpha = 1/3) +
  geom_line() +
  scale_x_date(
    NULL,
    date_breaks = "1 month",
```

```
  date_labels = "%b"
)
```

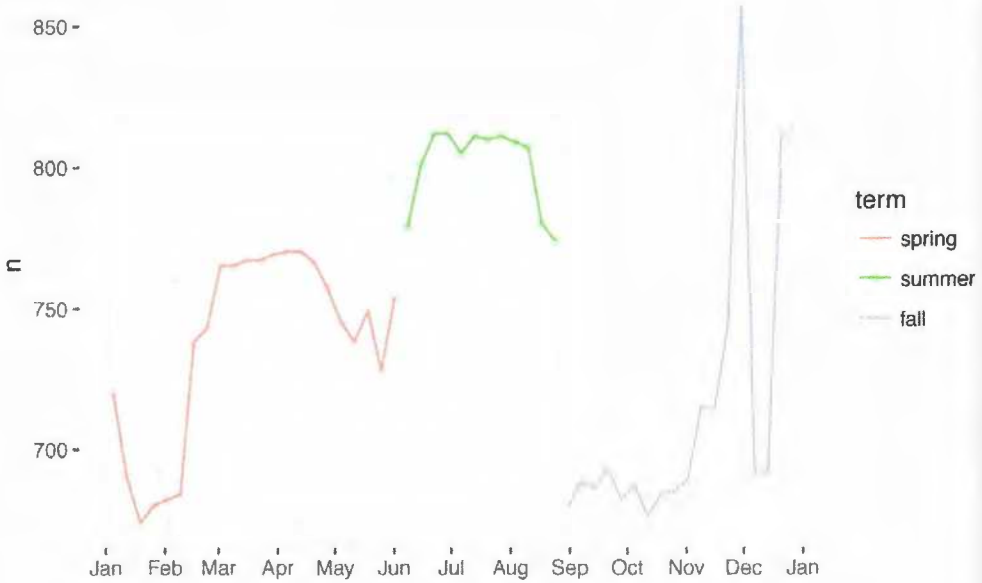


Рис. 19.20

(Мы вручную подправили некоторые даты, чтобы сделать разрывы в графике более аккуратными. Использование визуализации для того, чтобы лучше понять, что именно делает ваша функция, — действительно мощная и универсальная методика.)

Полезно посмотреть, какое влияние оказывает эта новая переменная на другие дни недели (рис. 19.21).

```
daily %>%
  ggplot(aes(wday, n, color = term)) +
    geom_boxplot()
```

Похоже на то, что количество субботних авиарейсов значительно варьируется от периода к периоду, поэтому идея раздельного учета эффекта дня недели для каждого периода представляется вполне обоснованной. Это улучшает нашу модель, но не так, как нам хотелось бы (рис. 19.22).

```
mod1 <- lm(n ~ wday, data = daily)
mod2 <- lm(n ~ wday * term, data = daily)

daily %>%
  gather_residuals(without_term = mod1, with_term = mod2) %>%
```

```
ggplot(aes(date, resid, color = model)) +
  geom_line(alpha = 0.75)
```

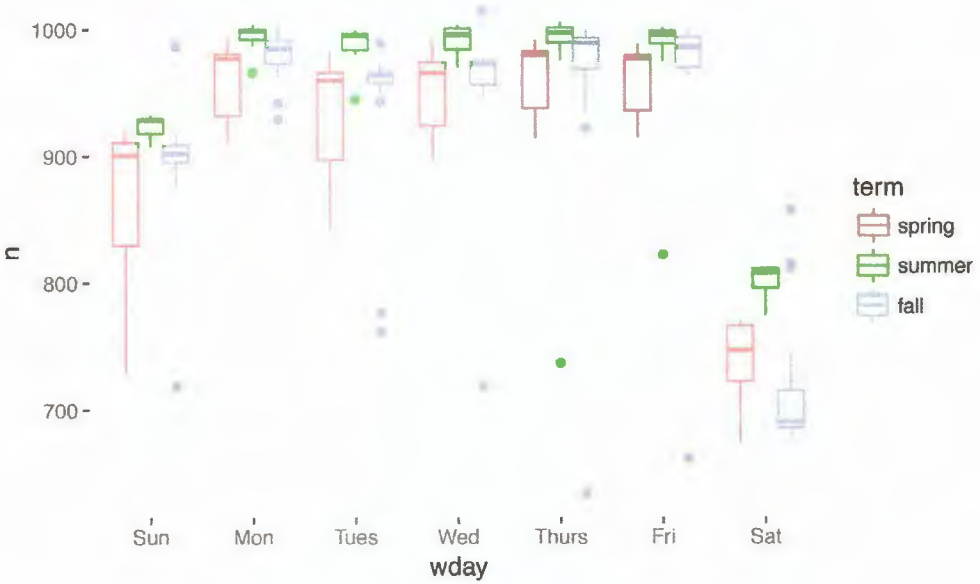


Рис. 19.21

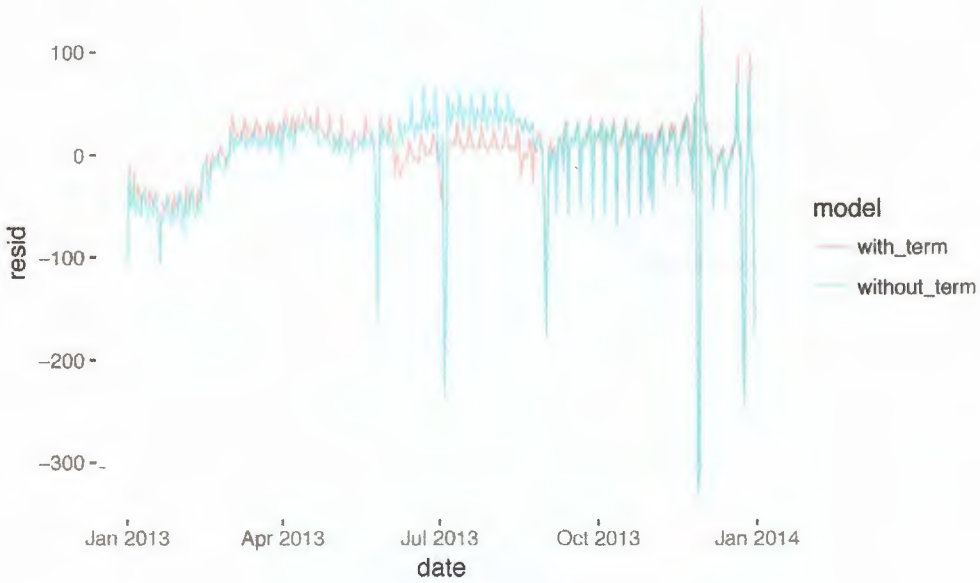


Рис. 19.22

Мы можем увидеть корень проблемы, наложив предсказания на исходные данные (рис. 19.23).

```
grid <- daily %>%
  data_grid(wday, term) %>%
  add_predictions(mod2, "n")

ggplot(daily, aes(wday, n)) +
  geom_boxplot() +
  geom_point(data = grid, color = "red") +
  facet_wrap(~ term)
```

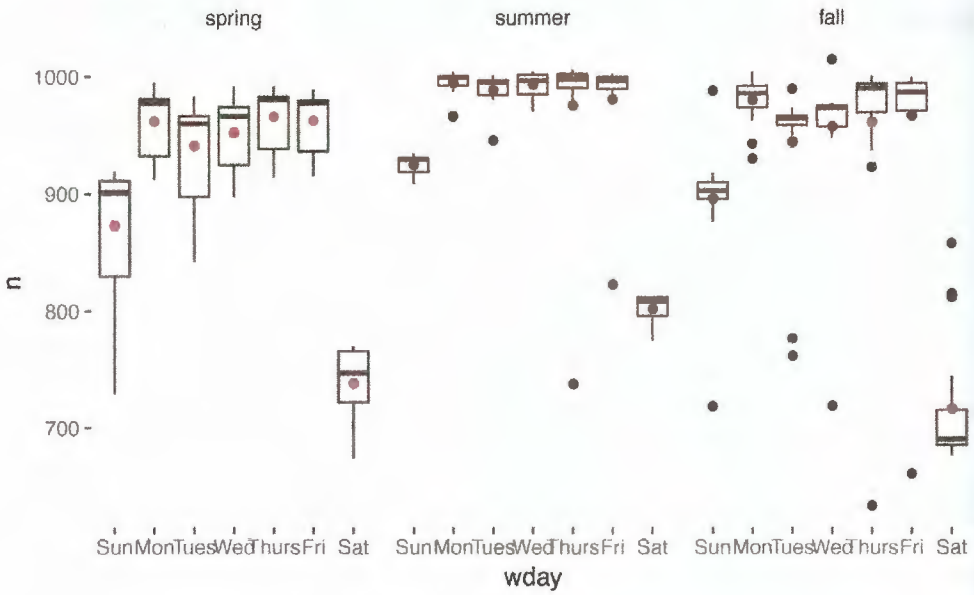


Рис. 19.23

Наша модель находит средний эффект, но имеется много заметных выбросов, которые удаляют среднее на большое расстояние от типичного значения. Данную проблему можно сгладить, используя модель, робастную по отношению к выбросам: `MASS::rlm()`. Это значительно снижает влияние выбросов на наши оценки и приводит к модели, которая отлично справляется с исключением шаблона зависимости от дня недели (рис. 19.24).

```
mod3 <- MASS::rlm(n ~ wday * term, data = daily)

daily %>%
  add_residuals(mod3, "resid") %>%
  ggplot(aes(date, resid)) +
```

```
geom_hline(yintercept = 0, size = 2, color = "white") +
geom_line()
```

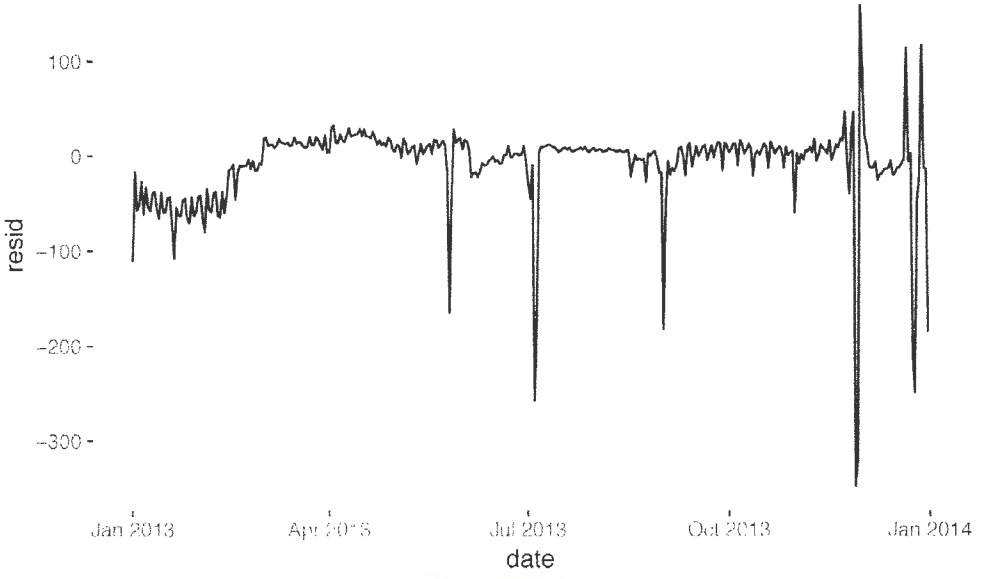


Рис. 19.24

Теперь заметить долгосрочный тренд, а также положительные и отрицательные выбросы стало намного легче.

Вычисляемые переменные

Если вы экспериментируете со многими моделями и вариантами визуализации, то имеет смысл объединить создание переменных в одну функцию, что позволит исключить случайное применение разных преобразований в разных местах. Например, мы могли бы записать следующий код.

```
compute_vars <- function (data) {
  data %>%
    mutate(
      term = term(date),
      wday = wday(date, label = TRUE)
    )
}
```

Есть еще одна возможность — поместить преобразования непосредственно в формулу модели.

```
wday2 <- function(x) wday(x, label = TRUE)
mod3 <- lm(n ~ wday2(date) * term(date), data = daily)
```

Разумен каждый из этих подходов. Явное создание преобразованных переменных целесообразно использовать в тех случаях, когда вы хотите проверить свою работу или использовать эти переменные в визуализации. Но не так-то легко использовать преобразования (наподобие сплайнов), которые возвращают несколько столбцов. Включение преобразования в функцию модели намного упрощает жизнь, если вы работаете с большим количеством различных наборов данных, поскольку модель становится самодостаточной.

Времена года: альтернативный подход

В предыдущем разделе мы использовали свои знания предметной области (каким образом распределение по году учебных периодов в школах США может влиять на частоту перемещения людей) для улучшения модели. Альтернативный подход к явному включению наших знаний в модель заключается в отведении данным более активной роли. Мы могли бы усилить гибкость модели, позволив ей захватить интересующий нас шаблон. В данном случае простой линейный тренд не подходит, в связи с чем мы попытаемся аппроксимировать данные по году гладкой кривой с помощью натурального сплайна (рис. 19.25).

```
library(splines)
mod <- MASS::rlm(n ~ wday * ns(date, 5), data = daily)

daily %>%
  data_grid(wday, date = seq_range(date, n = 13)) %>%
  add_predictions(mod) %>%
  ggplot(aes(date, pred, color = wday)) +
    geom_line() +
    geom_point()
```

Здесь наблюдается ярко выраженный шаблон поведения количества субботних полетов. Это обнадеживает, поскольку ту же тенденцию мы наблюдали и для исходных данных. Получение одного и того же сигнала при использовании разных подходов является хорошим знаком.

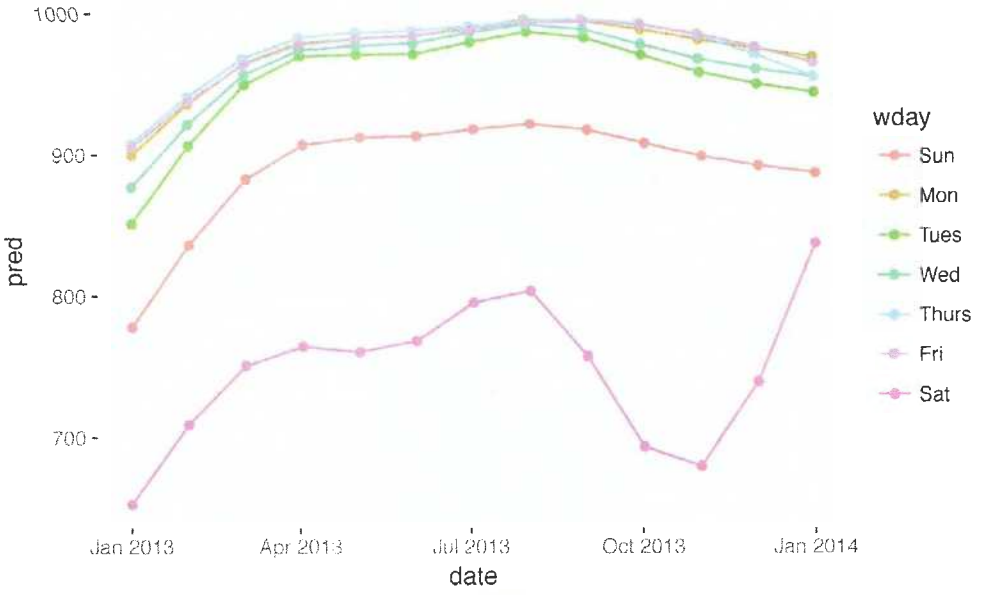


Рис. 19.25

Упражнения

1. Воспользуйтесь своими навыками выполнения поиска с помощью Google и предложите разумное объяснение того факта, что для 20 января, 26 мая и 1 сентября количество авиарейсов меньше ожидаемого. (Подсказка: эта особенность каждого из трех указанных дней имеет одно и то же объяснение.) Как бы вы обобщили эти дни на любой другой год?

2. Что представляют три дня, характеризующиеся высокими положительными значениями остатков? Как обобщить эти дни на любой другой год?

```
daily %>%
  top_n(3, resid)
#> # A tibble: 3 x 5
#>         date     n  wday resid   term
#>       <date> <int> <ord> <dbl> <fctr>
#> 1 2013-11-30   857   Sat 112.4   fall
#> 2 2013-12-01   987   Sun  95.5   fall
#> 3 2013-12-28   814   Sat  69.4   fall
```

3. Создайте новую переменную, разбивающую переменную `wday` на отдельные периоды, но только для субботы, т.е., например, для четверга и пятницы должны быть переменные `Thurs`, `Fri`, но для субботы — `Sat-summer`

(летний период), `Sat-spring` (весенний период), `Sat-fall` (осенний период). Сравните эту модель с той, в которой учитываются все комбинации переменных `wday` и `term`?

4. Создайте новую переменную `wday`, которая объединяет день недели, период (для субботы) и общенациональные праздники. Как выглядят остатки для этой модели?
5. Что произойдет, если вы построите модель, описывающую эффект дня недели, который меняется от месяца к месяцу (например, `n ~ wday * month`)? Почему эта модель не очень полезна?
6. Что, по вашему мнению, будет представлять собой модель `n ~ wday + ns(date, 5)`? Учитывая все то, что вам известно о данных, почему вы должны ожидать, что эта модель не будет особенно эффективной?
7. Мы выдвинули гипотезу о том, что люди, вылетающие воскресными рейсами, скорее всего, стараются попасть на деловую встречу, назначенную на понедельник. Исследуйте эту гипотезу, проследив за тем, как распределяются данные в зависимости от расстояния и времени: если она справедлива, то следует ожидать увидеть большее количество вечерних воскресных рейсов, вылетающих в отдаленные пункты назначения.
8. Несколько раздражает то, что воскресенье и суббота располагаются на разных концах графика. Напишите небольшую функцию, задающую уровни фактора таким образом, чтобы неделя начиналась с понедельника.

Дополнительная информация о моделях

Наше рассмотрение моделирования было весьма поверхностным, но хотелось бы надеяться, что этого вам будет достаточно для того, чтобы вы могли начать использовать простые универсальные инструменты для улучшения анализа данных, который вам приходится выполнять. Начинать с самого простого — это совершенно естественно! Как вы увидели, даже очень простые модели способны значительно расширить ваши возможности в изучении взаимодействия переменных.

В главах, посвященных моделированию данных, наши личные пристрастия выражены в еще большей степени, чем в остальных частях книги. Мы придерживаемся подхода к моделированию данных, который несколько отличается от подходов, используемых большинством других людей, но ввиду

нехватки места его изложение было довольно скупым. В действительности рассмотрение одного только моделирования данных заслуживает целой книги, в связи с чем мы настоятельно рекомендуем вам прочитать по крайней мере одну из перечисленных ниже трех книг.

- *Statistical Modeling: A Fresh Approach* (Danny Kaplan; `http://www.mosaic-web.org/go/StatisticalModeling/`). Эта книга представляет собой доступное введение в моделирование данных. Она поможет вам развить свою интуицию, научиться пользоваться математическими инструментами и параллельно приобрести навыки работы с R. Книгу можно использовать вместо традиционного вводного курса статистики, а содержащийся в ней материал соответствует современным требованиям и ориентирован на использование в науке о данных.
- *An Introduction to Statistical Learning* (Gareth James, Daniela Witten, Trevor Hastie, Robert Tibshirani; книга доступна для бесплатного скачивания по адресу `http://www-bcf.usc.edu/~gareth/ISL/`). В книге представлено семейство современных методик моделирования, совокупно известных как *статистическое обучение*. Для более глубокого изучения математики, лежащей в основе этих моделей, вам следует обратиться к классической книге *Elements of Statistical Learning* (Trevor Hastie, Robert Tibshirani, Jerome Friedman; также доступна для бесплатного скачивания по адресу `https://web.stanford.edu/~hastie/ElemStatLearn/`).
- *Applied Predictive Modeling* (Max Kuhn, Kjell Johnson). В этой книге представлены практические инструменты, предназначенные для решения реальных задач предсказательного моделирования.

ГЛАВА 20

Обработка множества моделей с помощью пакетов purrr и broom

Введение

В этой главе вы познакомитесь с тремя мощными решениями, которые облегчат вам работу с большим количеством моделей.

- Использование множества простых моделей для лучшего понимания свойств сложных наборов данных.
- Использование столбцов-списков для хранения произвольных структур данных во фрейме, что, например, позволяет иметь столбец, содержащий линейные модели.
- Использование пакета **broom** (разработчик Дэвид Робинсон) для превращения моделей в аккуратные данные. Эта мощная методика предназначена для работы с большим количеством моделей, поскольку, когда данные будут приведены к аккуратной форме, вы сможете применить к ним любую из методик, изученных вами в предыдущих главах книги.

Мы начнем с подробного рассмотрения мотивирующего примера, анализируя данные о средней продолжительности жизни в различных странах мира. Это небольшой набор данных, но он позволяет проиллюстрировать, какое значение может иметь моделирование для улучшения визуализации. Мы используем большое количество простых моделей для выделения нескольких самых сильных сигналов с той целью, чтобы можно было заметить более слабые сигналы, которые остаются после этого. Также будет показано,

как сводные данные модели облегчают обнаружение выбросов и необычных трендов.

В следующих разделах мы займемся подробным изучением отдельных методик.

- В разделе "Набор данных `gapminder`" будет представлен мотивирующий пример, в котором для подгонки моделей округов к общемировым экономическим данным используются столбцы списков.
- В разделе "Столбцы-списки" вы узнаете об одноименной структуре данных, а также о том, почему допустимо помещать списки во фреймы данных.
- В разделе "Создание столбцов-списков" вы познакомитесь с тремя основными способами создания столбцов-списков.
- В разделе "Упрощение столбцов-списков" вы научитесь выполнять обратное преобразование столбцов-списков в обычные атомарные векторы (или их наборы), чтобы с ними было легче работать.
- В разделе "Приведение данных к аккуратной форме с помощью пакета **broom**" вы познакомитесь с полным набором инструментов, предоставляемых пакетом **broom**, и увидите, как их можно применять к другим типам структур данных.

Данная глава предъявляет повышенные требования к читателю: если эта книга — ваше первое знакомство с R, то чтение приведенного ниже материала может вызвать у вас затруднения. От вас потребуется глубокое осмысление всей совокупности идей, касающихся моделирования, структур данных и итерационного подхода. Поэтому не огорчайтесь, если окажется, что вы не в состоянии осилить эту главу. В таком случае отложите ее чтение на несколько месяцев и вернитесь к ней тогда, когда почувствуете, что готовы к этому.

Необходимые ресурсы

Работа с большим количеством моделей требует использования множества пакетов библиотеки *tidyverse* (для выполнения разведочного анализа, подготовки данных и программирования), а также библиотеки *modelr*, упрощающей процесс моделирования.

```
library(modelr)
library(tidyverse)
```

Набор данных gapminder

Для демонстрации возможности подхода, основанного на привлечении одновременно многих простых моделей, мы будем использовать набор данных gapminder. Эти данные были популяризированы Хансом Рослингом — шведским врачом и специалистом в области статистики.

В наборе данных gapminder содержится информация об изменении во времени суммарных показателей развития стран мира, рассчитанных на основе таких статистик, как средняя продолжительность жизни и ВВП. Легкости доступа к этим данным в R мы обязаны Дженни Брайан, создавшей пакет **gapminder**.

```
library(gapminder)
gapminder
#> # A tibble: 1,704 x 6
#>       country continent  year lifeExp      pop gdpPercap
#>        <fctr>    <fctr> <int>   <dbl>    <int>     <dbl>
#> 1 Afghanistan      Asia  1952    28.8  8425333       779
#> 2 Afghanistan      Asia  1957    30.3  9240934       821
#> 3 Afghanistan      Asia  1962    32.0 10267083       853
#> 4 Afghanistan      Asia  1967    34.0 11537966       836
#> 5 Afghanistan      Asia  1972    36.1 13079460       740
#> 6 Afghanistan      Asia  1977    38.4 14880372       786

#> # ... with 1,698 more rows
```

В данном примере мы сосредоточим внимание всего лишь на трех переменных и попытаемся ответить на вопрос: "Как изменяется средняя продолжительность жизни (lifeExp) со временем (year) для каждой страны (country)?" Для этого неплохо начать с графика (рис. 20.1).

```
gapminder %>%
  ggplot(aes(year, lifeExp, group = country)) +
    geom_line(alpha = 1/3)
```

Этот набор данных небольшой: он содержит примерно 1700 наблюдений и 3 переменные. И тем не менее по виду графика трудно судить о том, что происходит на самом деле! В целом все выглядит так, будто средняя продолжительность жизни постоянно увеличивается. Однако при внимательном рассмотрении можно увидеть, что некоторые страны выпадают из этого шаблона. Каким образом можно более отчетливо выделить поведение данных для таких стран?

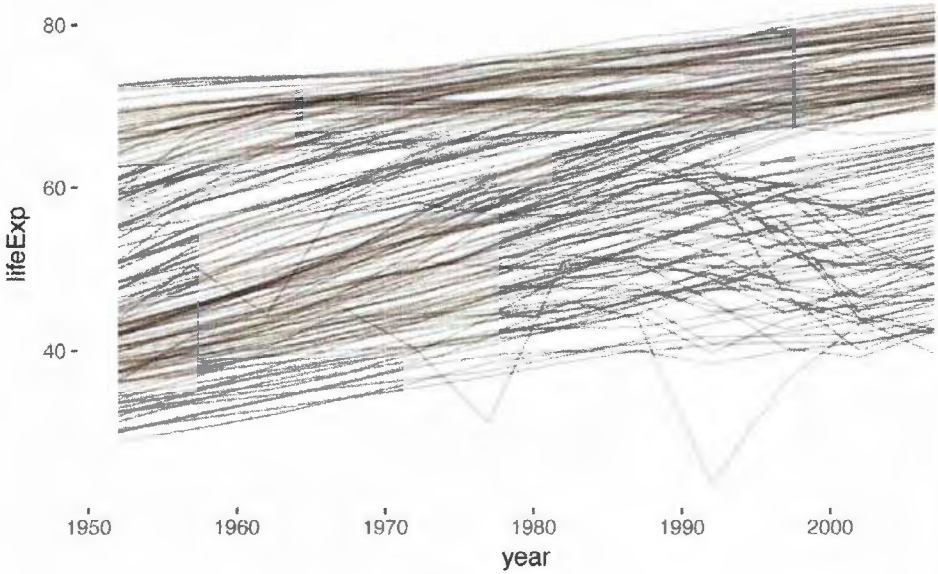


Рис. 20.1

Один из способов заключается в том, чтобы использовать тот же подход, что и в предыдущей главе: существует один сильный сигнал (общий линейный рост), который затрудняет наблюдение более слабых трендов. Мы разведем эти факторы посредством построения модели с линейным трендом. Эта модель захватит постоянный рост во времени, а остатки покажут, что остается.

Вы уже знаете, как это сделать, если бы страна была только одна (рис. 20.2).

```
nz <- filter(gapminder, country == "New Zealand")
nz %>%
  ggplot(aes(year, lifeExp)) +
  geom_line() +
  ggtitle("Full data = ")

nz_mod <- lm(lifeExp ~ year, data = nz)
nz %>%
  add_predictions(nz_mod) %>%
  ggplot(aes(year, pred)) +
  geom_line() +
  ggtitle("Linear trend + ")

nz %>%
  add_residuals(nz_mod) %>%
  ggplot(aes(year, resid)) +
```

```
  geom_hline(yintercept = 0, color = "white", size = 3) +
  geom_line() +
  ggtitle("Remaining pattern")
```

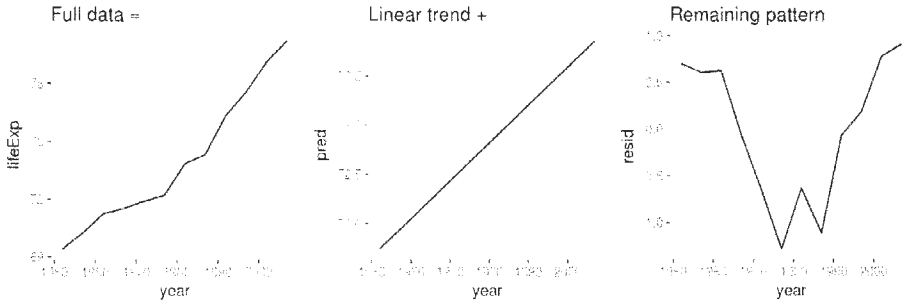


Рис. 20.2

А нет ли простого способа построения модели для каждой страны?

Вложенные данные

С этой задачей вполне можно было бы справиться путем многократного повторения процедуры копирования и вставки кода, однако вы уже знаете лучший способ! Извлеките общий код в функцию и повторно используйте функцию `map()` из пакета **purrr**. Эта задача структурируется несколько иначе по сравнению с тем, с чем вы сталкивались прежде. Вместо того чтобы повторять действие для каждой переменной, мы можем делать это для каждой страны (подмножества строк). Для этого нам нужна новая структура: вложенный фрейм данных. Чтобы создать такой фрейм, мы начинаем с группировки фрейма данных, а затем организуем его "вложение".

```
by_country <- gapminder %>%
  group_by(country, continent) %>%
  nest()

by_country
#> # A tibble: 142 x 3
#>       country continent              data
#>        <fctr>    <fctr>            <list>
#> 1 Afghanistan      Asia <tibble [12 x 4]>
#> 2     Albania    Europe <tibble [12 x 4]>
#> 3     Algeria    Africa <tibble [12 x 4]>
#> 4      Angola    Africa <tibble [12 x 4]>
#> 5   Argentina  Americas <tibble [12 x 4]>
#> 6   Australia   Oceania <tibble [12 x 4]>
#> # ... with 136 more rows
```

(Мы немного схитрили, сгруппировав данные как по континентам, так и по странам. При заданной переменной `country` переменная `continent` фиксирована, так что никакая дополнительная группа не добавляется, но это простой способ обзавестись дополнительной переменной.)

В результате мы получаем фрейм данных, в котором имеется по одной строке на группу (на одну страну) и один довольно необычный столбец: `data`. Этот столбец представляет собой список фреймов данных (вернее, tibble-фреймов, если быть точным). Поначалу эта идея кажется несколько странной: мы имеем фрейм данных, содержащий столбец, который является списком, состоящим из других фреймов данных! Вскоре мы объясним, почему считаем эту идею хорошей.

Рассматривать столбец `data` довольно неудобно, поскольку он представляет собой относительно сложный список, а разработка инструментов для исследования таких объектов еще не окончена. К сожалению, использование функции `str()` нельзя рекомендовать, поскольку она часто дает очень длинный вывод. Но если извлечь из столбца `data` одиночный элемент, то можно увидеть, что он содержит все данные для соответствующей страны (в данном случае — Афганистана).

```
by_country$data[[1]]
#> # A tibble: 12 x 4
#>    year lifeExp      pop gdpPercap
#>   <int>   <dbl>    <int>     <dbl>
#> 1  1952    28.8  8425333       779
#> 2  1957    30.3  9240934       821
#> 3  1962    32.0 10267083       853
#> 4  1967    34.0 11537966       836
#> 5  1972    36.1 13079460       740
#> 6  1977    38.4 14880372       786
#> # ... with 6 more rows
```

Обратите внимание на различие между стандартным сгруппированным фреймом данных и вложенным фреймом данных: в сгруппированном фрейме каждая строка является наблюдением, тогда как во вложенном фрейме каждая строка является группой. Вложенный набор данных также можно рассматривать как метанаблюдение — строку, которая представляет не одиночную точку во времени, а полную временную траекторию для страны.

Столбец-список

Теперь, когда у нас есть вложенный фрейм данных, мы имеем отличную возможность приступить к построению определенных моделей. Мы располагаем функцией для построения модели.

```
country_model <- function(df) {
  lm(lifeExp ~ year, data = df)
}
```

И мы хотим применить ее к каждому фрейму данных. Фреймы данных хранятся в списке, поэтому для применения функции `country_model` к каждому элементу мы можем использовать функцию `purrr::map()`:

```
models <- map(by_country$data, country_model)
```

Однако вместо того, чтобы оставить список моделей в виде свободного объекта, мы считаем более эффективным сохранить его в виде столбца фрейма данных `by_country`. Сохранение родственных объектов в столбцах в немалой степени определяет ценность фреймов данных, и это объясняет, почему мы считаем столбцы-списки неплохой идеей. В процессе работы с этими странами у нас будет множество списков, содержащих по одному элементу на одну страну. Так почему бы не хранить их вместе в одном фрейме данных?

Иными словами, вместо того чтобы создавать новый объект в глобальном окружении, мы создадим новую переменную во фрейме данных `by_country`. Это работа для функции `dplyr::mutate()`.

```
by_country <- by_country %>%
  mutate(model = map(data, country_model))
by_country
#> # A tibble: 142 x 4
#>       country continent              data    model
#>        <fctr>    <fctr>            <list>   <list>
#> 1 Afghanistan      Asia <tibble [12 x 4]> <S3: lm>
#> 2     Albania    Europe <tibble [12 x 4]> <S3: lm>
#> 3     Algeria    Africa <tibble [12 x 4]> <S3: lm>
#> 4      Angola    Africa <tibble [12 x 4]> <S3: lm>
#> 5   Argentina  Americas <tibble [12 x 4]> <S3: lm>
#> 6   Australia   Oceania <tibble [12 x 4]> <S3: lm>
#> # ... with 136 more rows
```

В этом есть большое преимущество: поскольку все родственные объекты хранятся вместе, вам не нужно вручную синхронизировать их в процессе фильтрации или перестановки. Семантика фрейма данных позаботится обо всем за вас.

```
by_country %>%
  filter(continent == "Europe")
#> # A tibble: 30 x 4
#> ?                 country continent              data    model
#> ?                  <fctr>    <fctr>            <list>   <list>
#> 1                 Albania    Europe <tibble [12 x 4]> <S3: lm>
#> 2                 Austria    Europe <tibble [12 x 4]> <S3: lm>
#> 3                 Belgium    Europe <tibble [12 x 4]> <S3: lm>
#> 4 Bosnia and Herzegovina    Europe <tibble [12 x 4]> <S3: lm>
#> 5                Bulgaria    Europe <tibble [12 x 4]> <S3: lm>
#> 6                 Croatia    Europe <tibble [12 x 4]> <S3: lm>
#> # ... with 24 more rows
by_country %>%
  arrange(continent, country)
#> # A tibble: 142 x 4
#>        country continent              data    model
#>         <fctr>    <fctr>            <list>   <list>
#> 1      Algeria    Africa <tibble [12 x 4]> <S3: lm>
#> 2       Angola    Africa <tibble [12 x 4]> <S3: lm>
#> 3        Benin    Africa <tibble [12 x 4]> <S3: lm>
#> 4     Botswana    Africa <tibble [12 x 4]> <S3: lm>
#> 5 Burkina Faso    Africa <tibble [12 x 4]> <S3: lm>
#> 6      Burundi    Africa <tibble [12 x 4]> <S3: lm>
#> # ... with 136 more rows
```

Если бы список фреймов данных и список моделей были отдельными объектами, то вам нужно было бы помнить о том, что каждый раз, когда вы переупорядочиваете один вектор или извлекаете из него элементы, то же самое следует сделать и с другими векторами с целью синхронизации их состояний. Если этого не сделать, то работоспособность вашего кода сохранится, но результаты будут неверными!

Отмена вложения данных

Перед этим мы вычисляли остатки для одной модели с одним набором данных. Теперь мы имеем 142 фрейма данных и 142 модели. Для вычисления остатков мы должны вызвать функцию `add_residuals()` для каждой пары "модель–данные".

```
by_country <- by_country %>%
  mutate(
    resids = map2(data, model, add_residuals)
  )
by_country
#> # A tibble: 142 x 5
#>        country continent              data    model
```

```
#>        <fctr>    <fctr>            <list>   <list>
#> 1 Afghanistan      Asia <tibble [12 x 4]> <S3: lm>
#> 2     Albania    Europe <tibble [12 x 4]> <S3: lm>
#> 3     Algeria    Africa <tibble [12 x 4]> <S3: lm>
#> 4      Angola    Africa <tibble [12 x 4]> <S3: lm>
#> 5   Argentina  Americas <tibble [12 x 4]> <S3: lm>
#> 6   Australia   Oceania <tibble [12 x 4]> <S3: lm>
#> # ... with 136 more rows, and 1 more variable:
#> #   resids <list>
```

Но как отложить на графике список фреймов данных? Вместо того чтобы ломать голову в поисках ответа на этот вопрос, давайте преобразуем список фреймов данных обратно в обычный фрейм. Ранее для преобразования обычного фрейма данных во вложенный мы использовали функцию `nest()`, а теперь мы сделаем противоположное с помощью функции `unnest()`.

```
resids <- unnest(by_country, resids)
resids
#> # A tibble: 1,704 x 7
#>       country continent  year lifeExp     pop gdpPercap
#>        <fctr>    <fctr> <int>   <dbl>   <int>     <dbl>
#> 1 Afghanistan      Asia  1952    28.8 8425333       779
#> 2 Afghanistan      Asia  1957    30.3 9240934       821
#> 3 Afghanistan      Asia  1962    32.010267083       853
#> 4 Afghanistan      Asia  1967    34.011537966       836
#> 5 Afghanistan      Asia  1972    36.113079460       740
#> 6 Afghanistan      Asia  1977    38.414880372       786
#> # ... with 1,698 more rows, and 1 more variable: resid <dbl>
```

Обратите внимание на то, что каждый обычный столбец повторяется один раз для каждой строки вложенного столбца.

Теперь, когда у нас имеется обычный фрейм данных, мы можем построить график остатков (рис. 20.3).

```
resids %>%
  ggplot(aes(year, resid)) +
    geom_line(aes(group = country), alpha = 1 / 3) +
    geom_smooth(se = FALSE)
#> `geom_smooth()` using method = 'gam'
```

Особенно информативно отображение аспектов по континентам на разных панелях (рис. 20.4).

```
resids %>%
  ggplot(aes(year, resid, group = country)) +
    geom_line(alpha = 1 / 3) +
    facet_wrap(~continent)
```

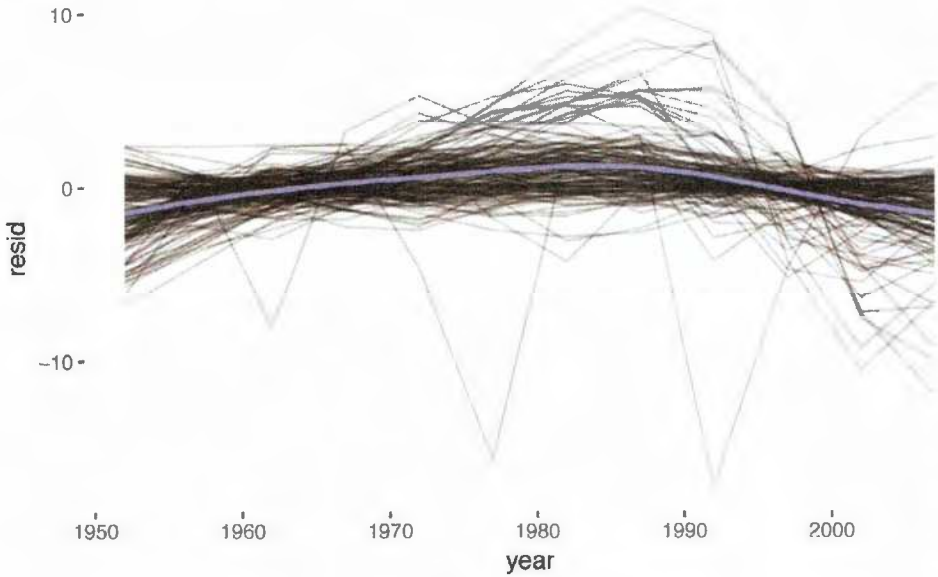


Рис. 20.3

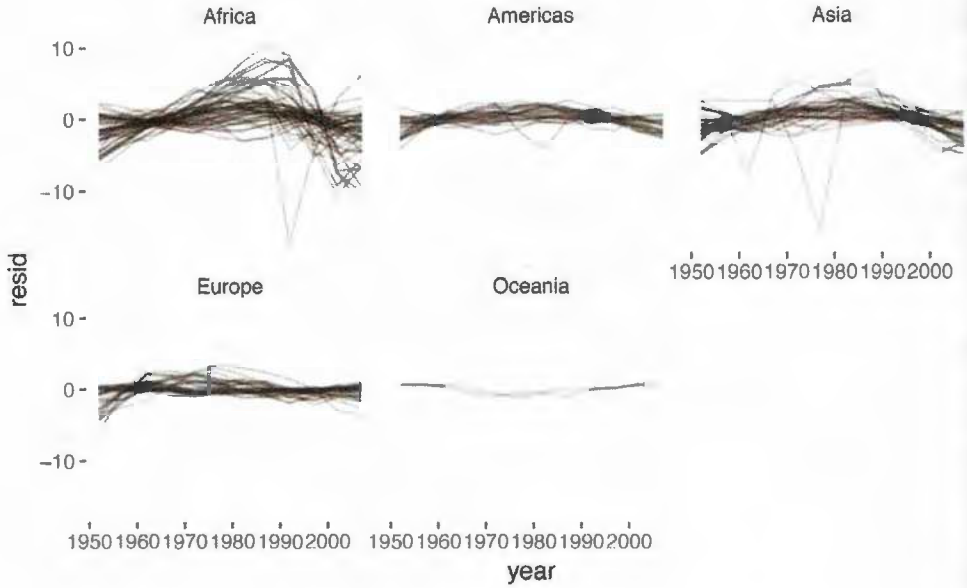


Рис. 20.4

Создается впечатление, что мы упустили какую-то слабую тенденцию. Также интересен график для Африки: заметны очень большие остатки, указывающие на то, что в этих случаях наша модель работает не слишком хорошо.

Мы исследуем это явление в следующем разделе, анализируя проблему под несколько другим углом.

Качество модели

Вместо анализа остатков, возникающих из модели, мы можем рассмотреть некоторые общие меры качества модели. В предыдущей главе вы узнали, как вычислять специализированные меры. В этом разделе будет продемонстрирован другой подход, основанный на использовании пакета **broom**. Пакет **broom** предоставляет общий набор функций, позволяющих преобразовывать модели в аккуратные данные. Здесь мы будем использовать функцию `broom::glance()` для извлечения качественных метрик модели. Применяя их к модели, мы получаем фрейм данных, состоящий из одной строки.

```
broom::glance(nz_mod)
#>   r.squared adj.r.squared sigma statistic  p.value df logLik
#>   AIC  BIC
#> 1     0.954         0.949 0.804       205 5.41e-08  2  -13.3
#>   32.6 34.1
#>   deviance df.residual
#> 1     6.47          10
```

Используя функции `mutate()` и `unnest()`, мы можем создать фрейм данных с одной строкой для каждой страны.

```
by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance)
#> # A tibble: 142 x 16
#>       country continent              data    model
#>        <fctr>    <fctr>            <list>   <list>
#> 1 Afghanistan      Asia <tibble [12 x 4]> <S3: lm>
#> 2     Albania    Europe <tibble [12 x 4]> <S3: lm>
#> 3     Algeria    Africa <tibble [12 x 4]> <S3: lm>
#> 4      Angola    Africa <tibble [12 x 4]> <S3: lm>
#> 5   Argentina  Americas <tibble [12 x 4]> <S3: lm>
#> 6   Australia   Oceania <tibble [12 x 4]> <S3: lm>
#> # ... with 136 more rows, and 12 more variables:
#> #   resids <list>, r.squared <dbl>, adj.r.squared <dbl>,
#> #   sigma <dbl>, statistic <dbl>, p.value <dbl>, df <int>,
#> #   logLik <dbl>, AIC <dbl>, BIC <dbl>, deviance <dbl>,
#> #   df.residual <int>
```

Это не совсем тот вывод, которого мы ожидали, поскольку он все еще включает все столбцы-списки. Таково поведение по умолчанию, когда

функция unnest() применяется к однострочным фреймам данных. Чтобы отменить это поведение, следует использовать аргумент .drop = TRUE.

```
glance <- by_country %>%
  mutate(glance = map(model, broom::glance)) %>%
  unnest(glance, .drop = TRUE)
glance
#> # A tibble: 142 x 13
#>       country continent r.squared adj.r.squared sigma
#>        <fctr>    <fctr>     <dbl>         <dbl> <dbl>
#> 1 Afghanistan      Asia     0.948         0.942 1.223
#> 2     Albania    Europe     0.911         0.902 1.983
#> 3     Algeria    Africa     0.985         0.984 1.323
#> 4      Angola    Africa     0.888         0.877 1.407
#> 5   Argentina  Americas     0.996         0.995 0.292
#> 6   Australia   Oceania     0.980         0.978 0.621
#> # ... with 136 more rows, and 8 more variables:
#> #   statistic <dbl>, p.value <dbl>, df <int>, logLik <dbl>,
#> #   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>
```

(Уделяйте внимание переменным, которые не выводятся на печать: в них содержится много полезной информации.)

Имея в своем распоряжении этот фрейм данных, мы можем приступить к рассмотрению неудачных моделей.

```
glance %>%
arrange(r.squared)
#> # A tibble: 142 x 13
#>       country continent r.squared adj.r.squared sigma
#>        <fctr>    <fctr>     <dbl>         <dbl> <dbl>
#> 1      Rwanda    Africa    0.0172      -0.08112  6.56
#> 2    Botswana    Africa    0.0340      -0.06257  6.11
#> 3    Zimbabwe    Africa    0.0562      -0.03814  7.21
#> 4      Zambia    Africa    0.0598      -0.03418  4.53
#> 5   Swaziland    Africa    0.0682      -0.02497  6.64
#> 6     Lesotho    Africa    0.0849      -0.00666  5.93
#> # ... with 136 more rows, and 8 more variables:
#> #   statistic <dbl>, p.value <dbl>, df <int>, logLik <dbl>,
#> #   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>
```

Как нетрудно заметить, все наихудшие модели соответствуют Африке. Перепроверим это с помощью графика. В данном случае количество наблюдений сравнительно невелико, и мы имеем дискретную переменную, поэтому нам подойдет функция geom_jitter() (рис. 20.5).

```
glance %>%
  ggplot(aes(continent, r.squared)) +
    geom_jitter(width = 0.5)
```

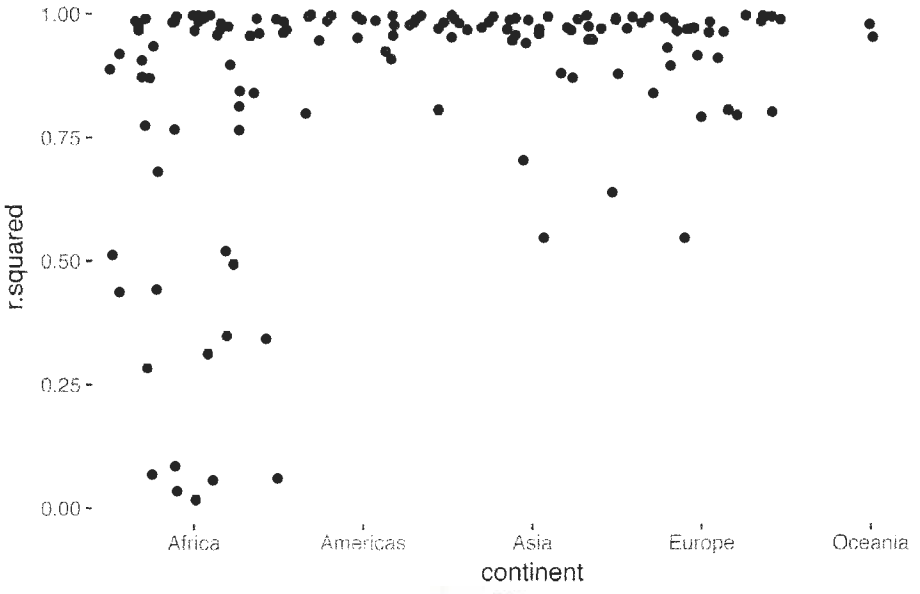


Рис. 20.5

Мы можем отфильтровать данные, соответствующие странам с особенно низкими значениями коэффициента детерминации R^2, и отобразить их в виде графика (рис. 20.6).

```
bad_fit <- filter(glance, r.squared < 0.25)

gapminder %>%
  semi_join(bad_fit, by = "country") %>%
  ggplot(aes(year, lifeExp, color = country)) +
    geom_line()
```

Здесь отчетливо выделяются два основных эффекта, вызванных двумя трагедиями: эпидемией СПИД (1999-2002 годы) и геноцидом в Руанде.

Упражнения

1. Похоже, линейный тренд является чрезмерным упрощением общего тренда. Можете ли вы улучшить результаты с помощью квадратичной полиномиальной аппроксимации? Как бы вы интерпретировали коэффициенты

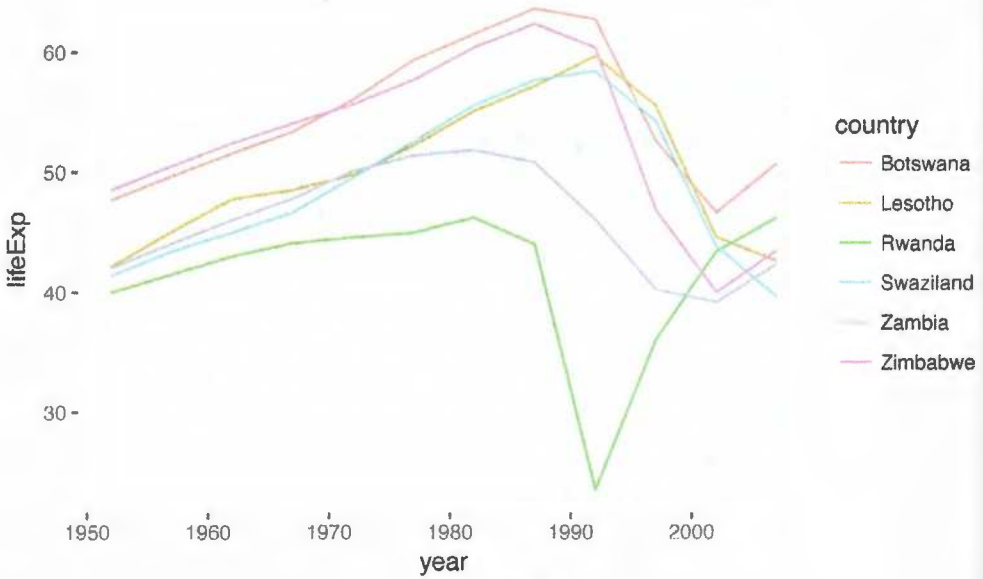


Рис. 20.6

полинома? (Подсказка: можно преобразовать переменную `year`, чтобы ее среднее было равно 0.)

2. Испытайте другие методы визуализации распределения R^2 для каждого континента. Попробуйте использовать пакет **ggbeeswarm**, который предоставляет методы, позволяющие избежать перекрывания, аналогичные методу `jitter()`, но детерминистические.
3. Чтобы создать последний график (отображающий данные для стран с наихудшей подгонкой модели), нам потребовалось два шага: мы создали фрейм данных с одной строкой на каждую страну, а затем частично объединили его с исходным набором данных. Этого объединения можно избежать, если вместо вызова функции `unnest(.drop = TRUE)` использовать вызов `unnest()`. Как это сделать?

Столбцы-списки

Теперь, когда вы уже знакомы в общих чертах с процессом одновременной обработки большого количества моделей, мы можем рассмотреть его более тщательно. В этом разделе мы рассмотрим столбцы-списки как структуру данных немного подробнее. Авторы книги лишь недавно по достоинству

оценили столбцы-списки. Эта структура данных неявно фигурирует в определении фрейма данных: фрейм данных — это именованный список векторов равной длины. Список — это вектор, поэтому всегда можно было на вполне законных основаниях использовать список в качестве столбца фрейма данных. Однако базовые пакеты R не позволяют столь же просто создавать столбцы-списки, и функция `data.frame()` трактует список как список столбцов.

```
data.frame(x = list(1:3, 3:5))
#>   x.1.3 x.3.5
#> 1     1     3
#> 2     2     4
#> 3     3     5
```

Такое поведение функции `data.frame()` можно предотвратить с помощью функции `I()`, но при этом результат выводится не в самом удобном виде.

```
data.frame(
  x = I(list(1:3, 3:5)),
  y = c("1, 2", "3, 4, 5")
)
#>          x       y
#> 1 1, 2, 3    1, 2
#> 2 3, 4, 5 3, 4, 5
```

Tibble-фрейм позволяет сгладить эту проблему за счет того, что он более "ленивый" (функция `tibble()` не изменяет свои входные данные), а также за счет того, что предоставляет более наглядный метод печати.

```
tibble(
  x = list(1:3, 3:5),
  y = c("1, 2", "3, 4, 5")
)
#> # A tibble: 2 x 2
#>           x       y
#>      <list>   <chr>
#> 1 <int [3]>    1, 2
#> 2 <int [3]> 3, 4, 5
```

Но с функцией `tribble()` еще проще работать по той причине, что она может автоматически определить, что вам нужен список.

```
tribble(
   ~x, ~y,
  1:3, "1, 2",
  3:5, "3, 4, 5"
```

```
)
#> # A tibble: 2 x 2
#>           x       y
#>      <list>   <chr>
#> 1 <int [3]>    1, 2
#> 2 <int [3]> 3, 4, 5
```

Столбцы-списки часто наиболее полезны в качестве промежуточных структур данных. С ними тяжело работать непосредственно, поскольку большинство функций R имеют дело с атомарными векторами или фреймами данных, но возможность содержать вместе все родственные элементы в одном фрейме данных стоит дополнительных хлопот.

В общем случае эффективный канал столбцов-списков состоит из трех частей.

1. Вы создаете столбец-список с помощью одного из следующих вариантов вызова функций: `nest()`, `summarize()` + `list()` или `mutate()` + функция `map`, как описано в разделе "Создание столбцов-списков".
2. Вы создаете другие промежуточные столбцы-списки, преобразуя существующие столбцы-списки с помощью функций `map()`, `map2()` или `pmap()`. Так, в предыдущем типовом примере мы создавали столбец — список моделей путем преобразования столбца — списка фреймов данных.
3. Вы упрощаете столбец-список путем его обратного преобразования во фрейм данных или атомарный вектор, как описано в разделе "Упрощение столбцов-списков".

Создание столбцов-списков

Как правило, вы не будете создавать столбцы-списки с помощью функции `tibble()`. Вместо этого вы будете создавать их из обычных столбцов, используя один из следующих трех методов:

1) с помощью функции `tidyr::nest()` для преобразования сгруппированного фрейма данных во вложенный фрейм, в котором вы имеете столбец — список фреймов данных;
2) с помощью функций `mutate()` и векторизованных функций, возвращающих список;

3) с помощью функций `summarize()` и `summary`, которые возвращают несколько результатов.

Альтернативный способ заключается в создании столбцов-списков из именованного списка с помощью функции `tibble::enframe()`.

Обычно, создавая столбцы-списки, вы должны убедиться в их однородности: каждый элемент должен содержать объекты одного и того же типа. Не существует способов проверить, что это действительно так, но если вы будете использовать пакет **purrr** и помнить то, что вы узнали о функциях, устойчивых к типам, то увидите, что это происходит естественным образом.

Создание столбцов-списков на основе вложений

Функция `nest()` создает вложенный фрейм данных, представляющий собой столбец-список фреймов данных. Во вложенном фрейме данных каждая строка является метанаблюдением: переменные, которые определяют наблюдение, предоставляются другими столбцами (как в случае переменных `country` и `continent`), а столбец — список фреймов данных предоставляет индивидуальные наблюдения, которые образуют метанаблюдение.

Функцию `nest()` можно использовать двумя способами. До этого вы видели примеры ее использования со сгруппированными фреймами данных. Когда функция `nest()` применяется к сгруппированному фрейму данных, она оставляет группирующие столбцы в том виде, как они есть, а все остальное объединяет в столбец-список.

```
gapminder %>%
  group_by(country, continent) %>%
  nest()
#> # A tibble: 142 x 3
#>       country continent              data
#>        <fctr>    <fctr>            <list>
#> 1 Afghanistan      Asia <tibble [12 x 4]>
#> 2     Albania    Europe <tibble [12 x 4]>
#> 3     Algeria    Africa <tibble [12 x 4]>
#> 4      Angola    Africa <tibble [12 x 4]>
#> 5   Argentina  Americas <tibble [12 x 4]>
#> 6   Australia   Oceania <tibble [12 x 4]>
#> # ... with 136 more rows
```

Эту функцию можно применять также к несгруппированным фреймам данных, указывая, какие столбцы должны быть вложенными.

```
gapminder %>%
  nest(year:gdpPercap)
#> # A tibble: 142 x 3
#>       country continent              data
#>        <fctr>    <fctr>            <list>
#> 1 Afghanistan      Asia <tibble [12 x 4]>
#> 2     Albania    Europe <tibble [12 x 4]>
#> 3     Algeria    Africa <tibble [12 x 4]>
#> 4      Angola    Africa <tibble [12 x 4]>
#> 5   Argentina  Americas <tibble [12 x 4]>
#> 6   Australia   Oceania           <tibble
#> # ... with 136 more rows
```

Создание столбцов-списков на основе векторизованных функций

Некоторые полезные функции принимают атомарный вектор и возвращают список. Например, в главе 11 вы познакомились с функцией `stringr::str_split()`, которая принимает символьный вектор и возвращает список символьных векторов.

Если вы используете этот подход в функции `mutate()`, то получите столбец-список.

```
df <- tribble(
  ~x1,
  "a,b,c",
  "d,e,f,g"
)

df %>%
  mutate(x2 = stringr::str_split(x1, ","))
#> # A tibble: 2 x 2
#>        x1        x2
#>     <chr>    <list>
#> 1   a,b,c <chr [3]>
#> 2 d,e,f,g <chr [4]>
```

Функция `unnest()` знает, как обрабатывать такие списки векторов.

```
df %>%
  mutate(x2 = stringr::str_split(x1, ",")) %>%
  unnest()
#> # A tibble: 7 x 2
#>        x1    x2
#>     <chr> <chr>
#> 1   a,b,c     a
```

```
=> 2   a,b,c     b
=> 3   a,b,c     c
=> 4 d,e,f,g     d
=> 5 d,e,f,g     e
=> 6 d,e,f,g     f
=> # ... with 1 more rows
```

(Если вы обнаружите, что вам часто приходится использовать этот шаблон кода, то по достоинству оцените функцию `tidyr:separate_rows()`, которая инкапсулирует его).

Альтернативный вариант этого шаблона предполагает использование функций `map()`, `map2()` и `pmap()` из пакета **purrr**. В качестве иллюстрации можно привести переделанный вариант примера из главы 12, в котором используется функция `mutate()`.

```
sim <- tribble(
  ~f,      ~params,
  "runif", list(min = -1, max = -1),
  "rnorm", list(sd = 5),
  "rpois", list(lambda = 10)
)

sim %>%
  mutate(sims = invoke_map(f, params, n = 10))
#> # A tibble: 3 x 3
#>       f     params       sims
#>   <chr>     <list>     <list>
#> 1 runif <list [2]> <dbl [10]>
#> 2 rnorm <list [1]> <dbl [10]>
#> 3 rpois <list [1]> <int [10]>
```

С технической точки зрения набор данных `sim` не является однородным, поскольку он содержит как вещественные, так и целочисленные векторы. Однако вряд ли это может привести к серьезным проблемам, поскольку и те, и другие являются числовыми векторами.

Создание столбцов-списков на основе многозначных итогов

Одним из ограничений функции `summarize()` является то, что она работает с итоговыми функциями, возвращающими одиночное значение. Таким образом, ее нельзя использовать с функциями наподобие `quantile()`, которая возвращает вектор произвольной длины.

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = quantile(mpg))
#> Error in eval(expr, envir, enclos): expecting a single value
```

Однако этот результат можно завернуть в список! Он подчиняется контракту функции `summarize()`, поскольку каждый итог теперь является списком (вектором) длины 1.

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg)))
#> # A tibble: 3 x 2
#>     cyl         q
#>   <dbl>    <list>
#> 1     4 <dbl [5]>
#> 2     6 <dbl [5]>
#> 3     8 <dbl [5]>
```

Для получения полезных результатов с помощью функции `unnest()` вам также потребуется получить вероятности.

```
probs <- c(0.01, 0.25, 0.5, 0.75, 0.99)
mtcars %>%
  group_by(cyl) %>%
  summarize(p = list(probs), q = list(quantile(mpg, probs))) %>%
  unnest()
#> # A tibble: 15 x 3
#>     cyl     p     q
#>   <dbl> <dbl> <dbl>
#> 1     4  0.01  21.4
#> 2     4  0.25  22.8
#> 3     4  0.50  26.0
#> 4     4  0.75  30.4
#> 5     4  0.99  33.8
#> 6     6  0.01  17.8
#> # ... with 9 more rows
```

Создание столбцов-списков на основе именованных списков

Фреймы данных со столбцами-списками обеспечивают решение одной часто встречающейся задачи: как быть, если необходимо итерировать как по содержимому списка, так и по его элементам? Вместо того чтобы пытаться втиснуть все в один объект, часто гораздо легче создать фрейм

данных: один столбец может содержать элементы, а другой — список. Простой способ создания такого фрейма данных из списка предлагает функция `tibble::enframe()`.

```
x <- list(
  a = 1:5,
  b = 3:4,
  c = 5:6
)
df <- enframe(x)
df
#> # A tibble: 3 x 2
#>    name     value
#>   <chr>    <list>
#> 1     a <int [5]>
#> 2     b <int [2]>
#> 3     c <int [2]>
```

Преимущество этой структуры заключается в том, что она обобщается самым непосредственным образом: имена полезны, если вы имеете дело с символьным вектором метаданных, но в случае других типов данных или нескольких векторов они вам не помогут.

Теперь, если вы хотите итерировать одновременно по именам и значениям, это можно сделать с помощью функции `map2()`.

```
df %>%
  mutate(
    smry = map2_chr(
      name,
      value,
      ~ stringr::str_c(.x, ": ", .y[1])
    )
  )
#> # A tibble: 3 x 3
#>    name     value  smry
#>   <chr>    <list> <chr>
#> 1     a <int [5]>  a: 1
#> 2     b <int [2]>  b: 3
#> 3     c <int [2]>  c: 5
```

Упражнения

1. Перечислите все функции, которые, по вашему мнению, принимают атомарный вектор и возвращают список.

2. Предложите полезные итоговые функции, которые, подобно функции `quantile()`, возвращают несколько значений.

3. Чего не хватает в приведенном ниже фрейме данных? Как функция `quantile()` возвращает эту недостающую часть?

```
mtcars %>%
  group_by(cyl) %>%
  summarize(q = list(quantile(mpg))) %>%
  unnest()
#> # A tibble: 15 x 2
#>     cyl     q
#>   <dbl> <dbl>
#> 1     4  21.4
#> 2     4  22.8
#> 3     4  26.0
#> 4     4  30.4
#> 5     4  33.9
#> 6     6  17.8
#> # ... with 9 more rows
```

4. Что делает приведенный ниже код? Чем он может быть полезен?

```
mtcars %>%
  group_by(cyl) %>%
  summarize_each(funs(list))
```

Упрощение столбцов-списков

Прежде чем вы сможете применить изученные вами методы манипуляции данными и их визуализации, вам понадобится упростить столбец-список путем его обратного преобразования в обычный столбец (атомарный вектор) или набор столбцов. Методика, которую вы используете для возврата к более простой структуре, зависит от того, что вам нужно: одно или несколько значений на элемент.

- Если вам нужно одиночное значение, используйте функцию `mutate()` с функциями `map_lgl()`, `map_int()`, `map_dbl()` и `map_chr()` для создания атомарного вектора.
- Если вам нужны несколько значений, используйте функцию `unnest()` для обратного преобразования столбца-списка в обычные столбцы, повторяя строки столько раз, сколько необходимо.

Эти случаи описаны более подробно в следующих разделах.

Преобразование списка в вектор

Преобразовав свой столбец-список в атомарный вектор, вы получите обычный столбец. Например, вы всегда можете суммировать объект по типу и длине, поэтому следующий код будет работать для любого столбца-списка:

```
df <- tribble(
  ~x,
  letters[1:5],
  1:3,
  runif(5)
)
df %>% mutate(
  type = map_chr(x, typeof),
  length = map_int(x, length)
)
#> # A tibble: 3 x 3
#>           x      type length
#>      <list>     <chr>  <int>
#> 1 <chr [5]> character      5
#> 2 <int [3]>   integer      3
#> 3 <dbl [5]>    double      5
```

Здесь получена та же базовая информация, которую выводит используемый по умолчанию метод `print`, но теперь вы можете использовать ее для фильтрации. Эта методика пригодится вам, если имеется неоднородный список и вы хотите отфильтровать части, которые вам не нужны.

Не забывайте о сокращенных вызовах функций `map_*()`: для извлечения строк, хранящихся в поле `apple` для каждого элемента `x`, можно использовать вызов `map_chr(x, "apple")`. Это, например, позволяет извлечь вложенные списки и разместить их в обычных столбцах. Используйте аргумент `.null` для предоставления элемента, который будет возвращаться (вместо `NULL`) в случае отсутствующего элемента списка.

```
df <- tribble(
  ~x,
  list(a = 1, b = 2),
  list(a = 2, c = 4)
)
df %>% mutate(
  a = map_dbl(x, "a"),
  b = map_dbl(x, "b", .null = NA_real_)
)
#> # A tibble: 2 x 3
#>             x     a     b
```

```
#>       <list> <dbl> <dbl>
#> 1 <list [2]>     1     2
#> 2 <list [2]>     2    NA
```

Отмена вложения

Работа функции `unnest()` заключается в повторении обычных столбцов по одному на каждый элемент столбца-списка. В следующем очень простом примере мы повторяем первую строку четыре раза (поскольку первый элемент `y` имеет длину 4), а вторую — один раз.

```
tibble(x = 1:2, y = list(1:4, 1)) %>% unnest(y)
#> # A tibble: 5 x 2
#>       x     y
#>   <int> <dbl>
#> 1     1     1
#> 2     1     2
#> 3     1     3
#> 4     1     4
#> 5     2     1
```

Это означает, что нельзя отменить вложение одновременно двух столбцов, которые содержат разное количество элементов.

```
# Работает, поскольку y и z содержат одинаковое количество элементов
# в каждой строке
df1 <- tribble(
  ~x, ~y,          ~z,
  1, c("a", "b"), 1:2,
  2, "c",           3
)
df1
#> # A tibble: 2 x 3
#>       x         y         z
#>   <dbl>    <list>    <list>
#> 1     1 <chr [2]> <int [2]>
#> 2     2 <chr [1]> <dbl [1]>
df1 %>% unnest(y, z)
#> # A tibble: 3 x 3
#>       x     y    z
#>   <dbl> <chr> <dbl>
#> 1     1     a     1
#> 2     1     b     2
#> 3     2     c     3

# Не работает, поскольку y и z имеют разное количество элементов
df2 <- tribble(
```

```
  ~x, ~y,          ~z,
  1, "a",          1:2,
  2, c("b", "c"),    3

df2
=> # A tibble: 2 x 3
=>       x         y        z
=>   <dbl>    <list>    <list>
=> 1     1 <chr [1]> <int [2]>
=> 2     2 <chr [2]> <dbl [1]>
df2 %>% unnest(y, z)
=> Error: All nested columns must have
=> the same number of elements.
```

Те же принципы применимы и в случае отмены вложения столбцов — списков фреймов данных. Отмена вложения одновременно нескольких столбцов-списков возможно тогда, когда во всех фреймах данных каждая строка содержит одно и то же количество строк.

Упражнения

1. Чем функция `lengths()` может быть полезна для создания столбцов атомарных секторов из столбцов-списков?
2. Перечислите наиболее распространенные типы векторов, встречающихся во фреймах данных. Что делает списки различными?

Приведение данных к аккуратной форме с помощью пакета broom

Пакет **broom** предоставляет три общих инструмента для превращения моделей в аккуратные фреймы данных.

- `broom::glance(model)` — возвращает строку для каждой модели. Каждый столбец предоставляет итоговую информацию о модели: либо меру качества модели, либо меру сложности модели, либо комбинацию обеих мер.
- `broom::tidy(model)` — возвращает строку для каждого коэффициента модели. Каждый столбец предоставляет информацию об оценке или ее изменчивости.

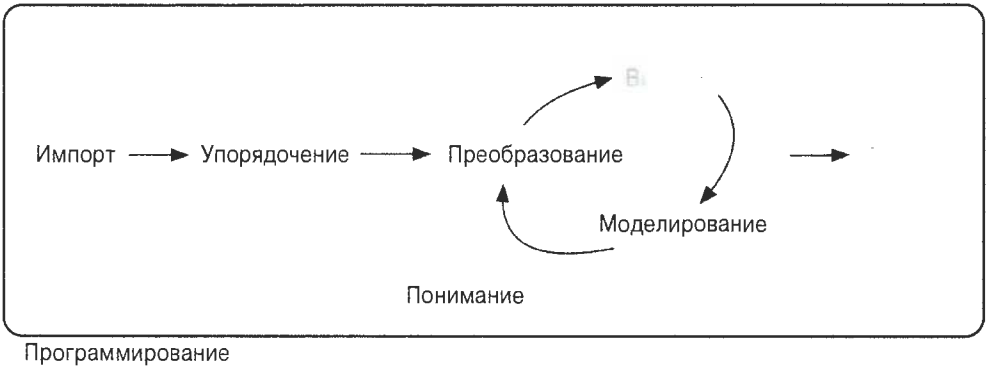
- `broom::augment(model, data)` — возвращает строку для каждой строки данных, добавляя дополнительные значения, такие как остатки и статистики влияний.

Пакет **broom** работает с широким рядом моделей, создаваемых наиболее популярными пакетами моделирования. Список поддерживаемых в настоящее время моделей можно найти по следующему адресу: https://github.com/tidyverse/broom.

ЧАСТЬ V

Коммуникация

До сих пор вы знакомились с инструментами, предназначенными для перемещения данных в R, приведения их к аккуратному виду, пригодному для анализа, и последующего исследования посредством преобразования, визуализации и моделирования. Однако чего бы стоил ваш анализ, если бы вы не могли изложить его суть другим людям? Для представления своих результатов вам нужны *средства коммуникации*.



Тема коммуникации является предметом обсуждения последующих четырех глав.

- В главе 21 вы познакомитесь со языком разметки R Markdown — инструментом, предназначенном для интеграции текста, кода и результатов. Его можно использовать в режиме блокнота для коммуникации между аналитиками и в режиме отчета — для коммуникации между аналитиками и руководителем. Благодаря мощным возможностям форматов языка R Markdown можно даже использовать один и тот же документ для обеих целей.
- В главе 22 показано, как превратить исследовательскую графику в объяснительную, которая поможет людям, впервые знакомящимся

с результатами вашего анализа, быстро и без затруднений вникнуть в суть дела.

- В главе 23 речь пойдет о многих других вариантах вывода, поддерживаемым языком R Markdown, включая панели, веб-сайты и книги.
- В завершающей главе 24 вы узнаете о "книге результатов анализа", а также о том, как организовать систематическую запись своих успехов и неудач, чтобы учиться на их примере.

К сожалению, в этих главах основное внимание уделяется техническим вопросам коммуникации, а не действительно сложным проблемам передачи своих мыслей другим людям. Однако теме информационной коммуникации посвящено множество замечательных книг, ссылки на которые приведены в конце каждой главы.

ГЛАВА 21

Язык разметки R Markdown

Введение

Для языка разметки R Markdown существует среда авторских разработок в области науки о данных, позволяющая объединять код, его результаты и поясняющие комментарии. Документы R Markdown полностью воспроизводимы и поддерживают десятки выходных форматов, включая PDF, Word, слайд-шоу и множество других.

Файлы R Markdown могут быть использованы для решения следующих задач:

- коммуникация с руководителями, желающими фокусировать внимание на выводах, а не на коде, лежащем в основе анализа;
- сотрудничество с другими специалистами, работающими в области науки о данных (включая вас самих в будущем!), которые заинтересованы как в ваших заключениях, так и в том, каким образом они были получены (т.е. в вашем коде);
- предоставление среды для авторских разработок в области науки о данных, которую можно было бы использовать в качестве лабораторного журнала, позволяющего фиксировать не только то, что вы сделали, но и ваши размышления, которые к этому привели.

R Markdown интегрирует ряд пакетов R и внешних инструментов. Это означает, что получение справки с помощью команды `?` будет недоступно. Вместо этого в процессе работы с R Markdown вам понадобятся следующие шпаргалки.

- *R Markdown Cheat Sheet*: доступна в интерактивной среде разработки RStudio по команде Help⇨Cheatsheets⇨R Markdown Cheat Sheet;

- *R Markdown Reference Guide*: доступна в интегрированной среде разработки RStudio по команде Help⇨Cheatsheets⇨R Markdown Reference Guide.

Обе шпаргалки также доступны по адресу http://rstudio.com/cheatsheets.

Необходимые ресурсы

Вам необходим пакет **rmarkdown**, но вы не должны явно устанавливать или загружать его, поскольку RStudio выполнит обе эти операции, когда возникнет такая необходимость.

Основы R Markdown

Перед вами файл R Markdown — простой текстовый файл с расширением *.Rmd*.

```
---
title: "Diamond sizes"
date: 2016-08-25
output: html_document
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

smaller <- diamonds %>%
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds. Only
`r nrow(diamonds) - nrow(smaller)` are larger than
2.5 carats. The distribution of the remainder is shown
below:

```{r, echo = FALSE}
smaller %>%
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```
```

В нем можно выделить три важных типа содержимого.

1. *Заголовок YAML* (необязательный), заключенный между группами символов `---`.
2. *Фрагменты* R-кода, заключенные между группами символов ```.
3. Текст, перемежаемый простыми командами форматирования наподобие `# heading` и `_italics_`.

Открыв файл *.Rmd*, вы получите интерфейс блокнота, в котором код перемежается с выводом. Вы можете выполнить любой фрагмент кода, щелкнув на кнопке Run (выглядит как кнопка пуска проигрывателя и находится над фрагментом кода) или нажав комбинацию клавиш <Ctrl+Shift+Enter>. RStudio выполнит код и отобразит под ним результаты (рис. 21.1).

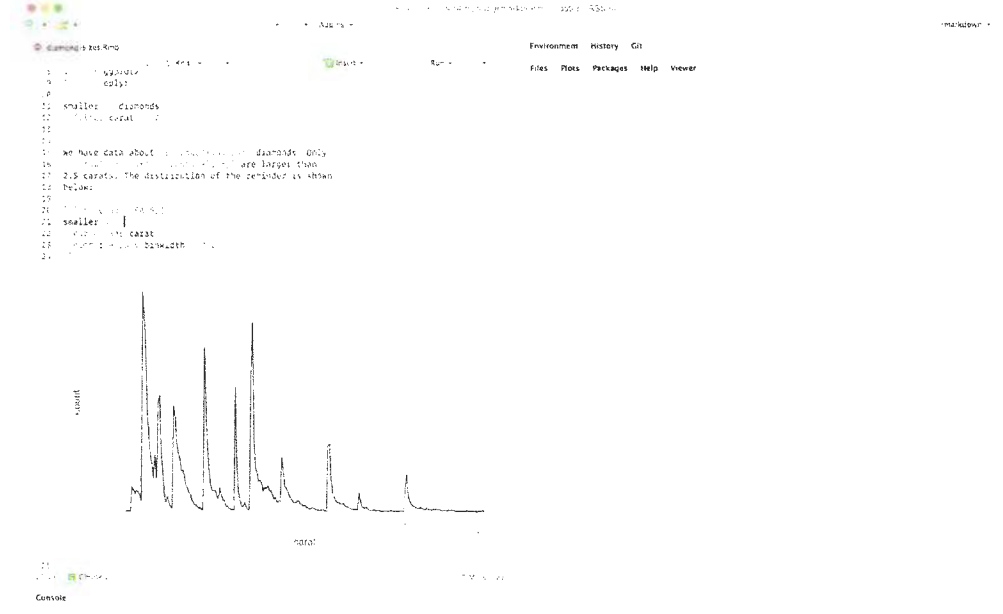


Рис. 21.1

Для получения полного отчета свяжите в одно целое текст, код и результаты, щелкнув на кнопке Knit или нажав комбинацию клавиш <Ctrl+Shift+K>. То же самое можно сделать программным путем с помощью команды `rmarkdown::render("1-example.Rmd")`. Это приведет к отображению отчета на панели просмотра и созданию полноценного HTML-файла, которым вы сможете поделиться с другими людьми (рис. 21.2).

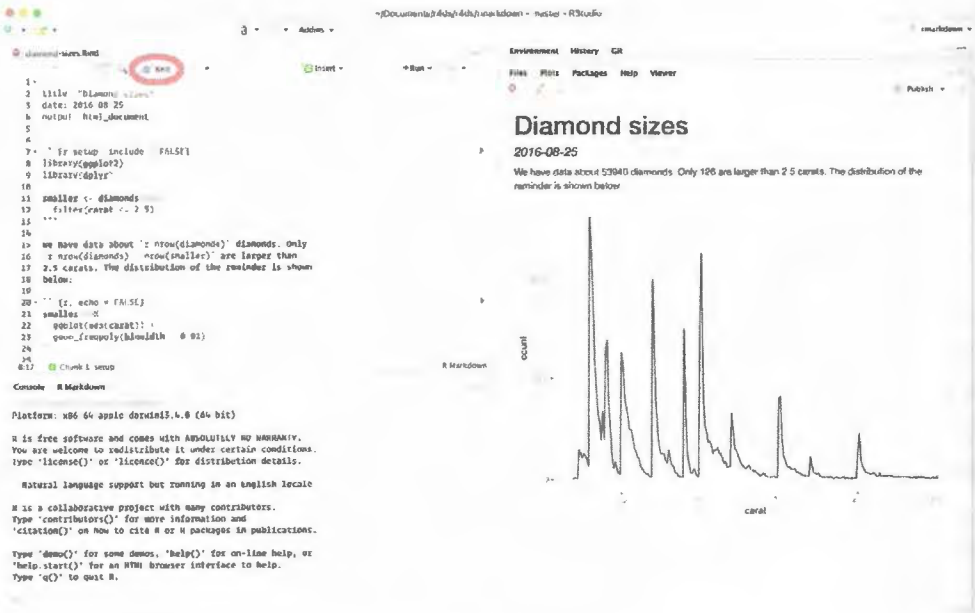


Рис. 21.2

R Markdown отправляет *.Rmd*-файл пакету **knitr**, который выполняет все фрагменты кода и создает новый документ Markdown (с расширением *.md*), включающий код и его вывод. Затем файл Markdown, сгенерированный пакетом **knitr**, обрабатывается конвертером **pandoc**, ответственным за создание конечного файла (рис. 21.3). Преимуществом такого двухшагового процесса является то, что он обеспечивает широкое разнообразие выходных форматов, о чем подробнее рассказано в главе 23.

Рис. 21.3

Чтобы приступить к работе с собственным *.Rmd*-файлом, выберите пункты меню File⇨New File⇨R Markdown. RStudio запустит программу-мастер, которую вы сможете использовать для предварительного заполнения файла полезным содержимым, напоминающим о том, как работают основные средства R Markdown.

Следующие три раздела посвящены подробному рассмотрению трех основных компонентов документа R Markdown: текста Markdown, кода фрагментов и заголовка YAML.

Упражнения

1. Создайте новый блокнот с помощью команды File⇨New File⇨R Notebook. Прочитайте инструкции. Попрактикуйтесь в выполнении фрагментов. Убедитесь в том, что можете изменить код, повторно выполнить его и получить измененный вывод.
2. Создайте новый документ R Markdown с помощью команды File⇨New File⇨R Markdown. Свяжите документ, щелкнув на соответствующей кнопке. Свяжите его, используя соответствующую комбинацию клавиш. Убедитесь в том, что можете изменить входные данные и увидеть обновленный вывод.
3. Сравните между собой созданные ранее файлы R Notebook и R Markdown. Чем сходны их выводы? Чем они различаются? Чем сходны входные данные? Чем они различаются? Что произойдет, если вы скопируете YAML-заголовок из одного файла в другой?
4. Создайте по одному новому документу R Markdown для каждого из трех встроенных форматов: HTML, PDF и Word. Свяжите каждый из этих трех документов. Чем различаются выводы? Чем различаются входные данные? (Возможно, вам понадобится установить программу LaTeX для создания вывода в формате PDF — RStudio подскажет вам, если в этом возникнет необходимость.)

Форматирование текста с помощью Markdown

Текстовые пояснения в *.Rmd*-файлах создаются с помощью облегченного языка разметки Markdown, предназначенного для форматирования простых текстовых файлов, который обеспечивает максимально удобное чтение и написание текста. Кроме того, изучение языка Markdown не составляет большого труда. Ниже показано, как использовать язык Markdown конвертера Pandoc, который представляет собой незначительно расширенную версию языка Markdown, распознаваемого средой R.

```
Форматирование текста
-------------------------------------------------------------

*курсив* или _курсив_
**полужирный** или __полужирный__
`код`
верхний индекс^2^ и нижний индекс~2~

Заголовки
-------------------------------------------------------------

# Заголовок 1 уровня

## Заголовок 2 уровня

### Заголовок 3 уровня

Списки
-------------------------------------------------------------

*   Элемент 1 маркерного списка

*   Элемент 2

    * Элемент 2a

    * Элемент 2b

1.  Элемент 1 нумерованного списка

1.  Элемент 2. Номера автоматически инкрементируются в выводе.

Ссылки и изображения
-------------------------------------------------------------

<http://example.com>

[связанная фраза](http://example.com)

![необязательная подпись](path/to/img.png)

Таблицы
-------------------------------------------------------------

Первый заголовок  | Второй заголовок
--------------- - | ----------------
Содержимое ячейки | Содержимое ячейки
Содержимое ячейки | Содержимое ячейки
```

Самый лучший способ изучения этих правил — пытаться их использовать. Это может занять у вас несколько дней, но вскоре они станут вашим

вторым "я", и вы даже не будете о них задумываться. Если вам понадобится подсказка, воспользуйтесь справочной информацией, выполнив команду Help⇨Markdown Quick Reference.

Упражнение

1. Попрактикуйтесь в применении своих знаний, создав краткое резюме. Оно должно быть озаглавлено вашим именем и включать подзаголовок по крайней мере для одного из разделов, касающихся вашего образования или трудового стажа. Каждый раздел должен включать маркированный список занимаемых вами должностей или научных степеней. Выделите годы полужирным шрифтом.
2. Воспользуйтесь средством получения быстрой справки (Markdown Quick Reference) для того, чтобы выяснить, как:
   а) добавить сноску;
   б) добавить горизонтальную линейку;
   в) добавить блочную цитату.
3. Скопируйте содержимое файла *diamond-sizes.Rmd*, доступного по адресу `https://github.com/hadley/r4ds/tree/master/rmarkdown`, в локальный документ R Markdown. Убедитесь в том, что можете выполнить этот файл, а затем добавьте после полигона частот текст, описывающий его впечатляющие возможности.

Фрагменты кода

Чтобы выполнить код, содержащийся в документе R Markdown, необходимо вставить фрагмент. Это можно сделать тремя способами:

1) с помощью комбинации клавиш <Ctrl+Alt+I>;
2) с помощью кнопки Insert панели инструментов редактора;
3) вводом вручную разделителей фрагментов кода ```{r} и ```.

Безусловно, мы рекомендовали бы вам запомнить указанную комбинацию клавиш. Это позволит сэкономить массу времени в долгосрочной перспективе!

Можете продолжать выполнение кода, используя комбинацию клавиш <Ctrl+Enter>, которую (мы надеемся!) вы запомнили и полюбили. Однако для фрагментов предусмотрена новая комбинация клавиш: <Ctrl+Shift+Enter>,

которая выполняет весь код фрагмента. Рассматривайте фрагмент как функцию. Фрагмент должен быть относительно самодостаточным и предназначенным для выполнения какой-то одной задачи.

В последующих разделах описывается заголовок фрагмента, состоящий из символов ```{r, за которыми следуют необязательное имя фрагмента, опции, разделенные запятыми, и закрывающая фигурная скобка }. Далее идет ваш R-код, а окончание фрагмента обозначается символами ```.

Имя фрагмента

Фрагментам можно присваивать необязательные имена ```{r *имя*}, что обеспечивает следующие три преимущества.

- Облегчаются переходы к конкретным фрагментам с помощью раскрывающегося меню навигации по коду, расположенного в левом нижнем углу окна редактора скриптов (рис. 21.4).

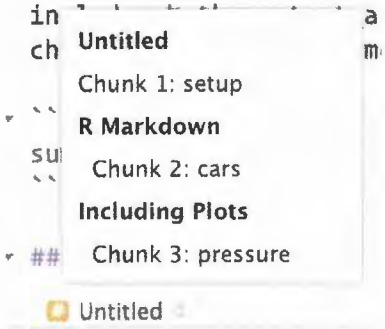


Рис. 21.4

- Графики, создаваемые фрагментами, будут снабжаться полезными именами, облегчающими их использование в других местах. Более подробно об этом рассказано в конце главы 22.
- Вы можете настраивать сети кешированных фрагментов, позволяющие избежать повторного выполнения дорогостоящих вычислений при каждом запуске. Вскоре этот вопрос будет рассмотрен более подробно.

Существует одно специальное имя, которое задает особое поведение: `setup`. Если вы работаете в режиме блокнота, фрагмент с именем `setup` будет автоматически однократно выполняться до выполнения любого другого кода.

Опции фрагмента

Вывод фрагмента можно настраивать с помощью опций — аргументов, передаваемых заголовку фрагмента. Пакет **knitr** предоставляет почти 60 опций, которые можно использовать для настройки фрагментов своего кода. Мы рассмотрим наиболее важные опции, которые вы будете использовать чаще других. С полным списком опций можно ознакомиться по адресу `http://yihui.name/knitr/options/`.

Наиболее важный набор опций, перечисленных ниже, управляет тем, будет ли выполняться блок кода и какие результаты должны быть вставлены в окончательный отчет.

- `eval = FALSE`. Предотвращает вычисление кода. (Совершенно очевидно, что если код не будет выполняться, то результаты не будут генерироваться.) Эта опция полезна для отображения кода примера или отключения большого блока кода без "закомментирования" строк.
- `include = FALSE`. Позволяет выполнить код, не отображая в конечном документе ни код, ни результаты. Используйте эту опцию для кода настройки, если не хотите, чтобы он загромождал отчет.
- `echo = FALSE`. Предотвращает отображение кода, но не результатов, в конечном файле. Используйте эту опцию при подготовке отчетов для людей, которых не интересует R-код, использованный для получения результатов.
- `message = FALSE` или `warning = FALSE`. Предотвращает появление сообщений или предупреждений в конечном файле.
- `results = 'hide'`. Скрывает печатаемый вывод
- `fig.show = 'hide'`. Скрывает графики.
- `error = TRUE`. Обеспечивает продолжение рендеринга, даже если код возвращает ошибку. Необходимость включать эту опцию в окончательную версию своего отчета может возникать у вас лишь в редких случаях, но она может быть весьма кстати при отладке *.Rmd*-файла. Также эта опция будет вам полезна, если вы преподаете язык R и сознательно включаете в файл ошибку в учебных целях. По умолчанию `error = FALSE`, что означает аварийное завершение, если в документе содержится хотя бы одна ошибка.

В следующей таблице указаны типы вывода, подавляемого каждой из опций.

Опция	Выполнение кода	Отображение кода	Результаты	Графики	Сообщения	Предупреждения
eval = FALSE	x	-	x	x	x	x
include = FALSE	-	x	x	x	x	x
echo = FALSE	-	x	-	-	-	-
results = "hide"	-	-	x	-	-	-
fig.show = "hide"	-	-	-	x	-	-
message = FALSE	-	-	-	-	x	-
warning = FALSE	-	-	-	-	-	x

Таблица

По умолчанию R Markdown выводит фреймы данных и матрицы в том виде, в каком они появляются в окне консоли.

```
mtcars[1:5, 1:10]
#>                    mpg cyl disp  hp drat   wt qsec vs am gear
#> Mazda RX4         21.0   6  160 110 3.90 2.62 16.5  0  1    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.88 17.0  0  1    4
#> Datsun 71         22.8   4  108  93 3.85 2.32 18.6  1  1    4
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.21 19.4  1  0    3
#> Hornet Sportabout 18.7   8  360 175 3.15 3.44 17.0  0  0    3
```

Если вы предпочитаете, чтобы данные отображались с дополнительным форматированием, используйте функцию `knitr::kable()`. Приведенный ниже код генерирует табл. 21.1.

```
knitr::kable(
  mtcars[1:5, ],
  caption = "Генератор таблиц knitr::kable()"
)
```

Таблица 21.1. Генератор таблиц `knitr::kable()`

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.62	16.5	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.88	17.0	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.32	18.6	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.21	19.4	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.0	0	0	3	2

Прочитайте документацию (`?knitr::kable`), чтобы ознакомиться с другими способами оформления таблиц. Также ознакомьтесь с более широкими

возможностями, которые предлагают пакеты **xtable**, **stargazer**, **pander**, **tables** и **ascii**. Каждый из них предоставляет набор средств, возвращающих форматированные таблицы из R-кода.

Кроме того, существует богатый набор опций, позволяющих управлять встраиванием рисунков. О них пойдет речь в главе 22.

Кеширование

Обычно формирование нового выходного документа с помощью функции `knit()` каждый раз начинается с чистого листа. Это отличный способ обеспечить воспроизводимость результатов, гарантирующий, что вы захватили все важные вычисления в коде. Однако он также может доставить головную боль, если некоторые вычисления занимают длительное время. Решением проблемы является опция `cache = TRUE`, установка которой позволяет сохранить вывод фрагмента на диске в файле, имеющем специальное имя. При последующих запусках **knitr** проверит, изменился ли код, и если он не изменился, то повторно использует кешированные результаты.

Систему кеширования следует использовать с осторожностью, поскольку по умолчанию она базируется исключительно на коде, а не на его зависимостях. Например, в приведенном ниже коде фрагмент `processed_data` зависит от фрагмента `raw_data`.

```
```{r raw_data}
rawdata <- readr::read_csv("a_very_large_file.csv")
```

```{r processed_data, cached = TRUE}
processed_data <- rawdata %>%
  filter(!is.na(import_var)) %>%
  mutate(new_variable = complicated_transformation(x, y, z))
```
```

Кеширование фрагмента `processed_data` означает, что в случае изменения канала **dplyr** он будет выполнен заново, но если будет изменен вызов `read_csv()`, то этого не произойдет. Проблемы можно избежать, используя опцию `dependson`.

```
```{r processed_data, cached = TRUE, dependson = "raw_data"}
processed_data <- rawdata %>%
  filter(!is.na(import_var)) %>%
  mutate(new_variable = complicated_transformation(x, y, z))
```
```

Опция dependson должна содержать символьный вектор всех фрагментов, от которых зависит кешируемый фрагмент. Пакет **knitr** будет обновлять результаты для кешированного фрагмента всякий раз, когда обнаружит, что одна из его зависимостей изменилась.

Заметьте, что в случае изменения файла *a_very_large_file.csv* фрагменты не будут обновляться, поскольку **knitr** отслеживает лишь изменения в файле *.Rmd*. Если хотите, чтобы отслеживались изменения также в *.csv*-файле, используйте опцию cache.extra. Это произвольное выражение R, которое будет делать недействительным кеш всякий раз, когда он будет изменяться. Для этого неплохо использовать функцию file.info(), которая возвращает массу информации о файле, включая дату его последнего изменения, что позволяет записать следующий код.

```
```{r raw_data, cache.extra = file.info("a_very_large_file.csv")}
rawdata <- readr::read_csv("a_very_large_file.csv")
```
```

По мере того, как ваши стратегии кеширования будут постепенно усложняться, имеет смысл регулярно очищать все кеши с помощью функции knitr::clean_cache().

Мы следуем совету Дэвида Робинсона (https://twitter.com/drob/status/738786604731490304) относительно именования таких фрагментов: каждому фрагменту присваивается имя, ассоциируемое с именем первичного объекта, который его создает. Это делает более понятной спецификацию dependson.

Глобальные опции

Поработав какое-то время с пакетом **knitr**, вы обнаружите, что некоторые из заданных по умолчанию опций фрагмента не удовлетворяют вашим нуждам, и захотите изменить их. Это можно сделать, вызвав функцию knitr::opts_chunk$set() во фрагменте кода. Например, при написании книг и руководств мы используем следующие опции.

```
knitr::opts_chunk$set(
  comment = "#>",
  collapse = TRUE
)
```

Этот код устанавливает предпочитаемый мною вариант форматирования комментариев и гарантирует тесную связь между кодом и выводом. С другой

стороны, при подготовке отчетов целесообразно устанавливать следующую опцию.

```
knitr::opts_chunk$set(
  echo = FALSE
)
```

Это приведет к сокрытию кода по умолчанию и отображению только тех фрагментов кода, для которых вы установите это явно (с помощью опции `echo = TRUE`). Иногда вы можете решить воспользоваться опциями `message = FALSE` и `warning = FALSE`, но это затруднит отладку кода, поскольку вы не будете видеть никаких сообщений в конечном документе.

Встроенный код

Для внедрения R-кода в документ R Markdown существует другой способ: непосредственная вставка кода в текст с помощью команды `` `r ` ``. Этот способ может быть очень полезным, если вы хотите упоминать свойства данных в тексте. Так, в примере документа, который приводился в начале этой главы, мы использовали следующий код:

> We have data about `` `r nrow(diamonds)` `` diamonds. Only
> `` `r nrow(diamonds) - nrow(smaller)` `` are larger than 2.5 carats.
> The distribution of the remainder is shown below:

При формировании отчета с помощью функции `knit()` результаты этих вычислений будут вставлены в текст:

> We have data about 53940 diamonds. Only 126 are larger than 2.5 carats. The distribution of the remainder is shown below:

Если вы вставляете в текст числа, воспользуйтесь функцией `format()`. Она позволит вам задать количество цифр в числе, чтобы оно не выводилось с бессмысленно высокой степенью точности, а опция `big.mark` облегчит чтение чисел. Мы часто сочетаем задание двух этих опций в одной вспомогательной функции.

```
comma <- function(x) {
  comma(x, digits = 2, big.mark = ",")
}
comma(3452345)
#> [1] "3,452,345"
comma(.12358124331)
#> [1] "0.12"
```

Упражнения

1. Добавьте раздел, в котором исследуется зависимость размеров бриллиантов от качества огранки, цвета и чистоты. Исходите из предположения, что вы пишете отчет для кого-то, кто не знает R, и вместо установки опции `echo = FALSE` для каждого фрагмента установите глобальную опцию.
2. Загрузите файл *diamond-sizes.Rmd*, доступный по адресу `https://github.com/hadley/r4ds/tree/master/rmarkdown`. Добавьте раздел, описывающий 20 самых крупных бриллиантов, который включает таблицу с описанием их наиболее важных атрибутов.
3. Измените файл `diamonds-sizes.Rmd` таким образом, чтобы можно было использовать функцию `comma()` для получения аккуратно форматированного вывода. Включите также вывод процентной доли бриллиантов с весом более 2,5 карата.
4. Настройте сеть фрагментов, в которых `d` зависит от `c` и `b`, а `b` и `c` зависят от `a`. Организуйте вывод даты и времени с помощью функции `lubridate::now()`, задайте опцию `cache = TRUE`, а затем проверьте, правильно ли вы понимаете смысл кеширования.

Устранение неполадок

Устранение неполадок в документах R Markdown может стать серьезной проблемой, поскольку вы уже не находитесь в интерактивной среде R, и вам придется освоить новые приемы. Первое, что вы всегда должны пытаться сделать, — это воспроизвести проблему в интерактивном сеансе. Перезапустите R и выполните команду "Run all chunks" (с помощью либо меню Code, либо комбинации клавиш <Ctrl+Alt+R>). Если вам повезет, вы воспроизведете проблему и сможете определить, что происходит на самом деле, в интерактивном режиме.

Если указанные меры не помогут, то это означает, что между интерактивной средой и средой R Markdown существуют какие-то различия. В подобных ситуациях вы должны внимательно исследовать опции. Чаще всего эти среды различаются рабочими каталогами: рабочим каталогом документа R Markdown является каталог, в котором он находится. Проверьте, соответствует ли рабочий каталог вашим ожиданиям, включив во фрагмент вызов функции `getwd()`.

Далее тщательно проанализируйте возможные причины ошибки. Вам нужны систематические подтверждения того, что эти причины одинаковы в

сеансах R и R Markdown. Проще всего это можно сделать, установив для проблемного фрагмента опцию `error = TRUE` и использовав функции `print()` и `str()` для проверки того, что параметры настройки совпадают с ожидаемыми.

Заголовок YAML

Вы можете управлять многими другими опциями, относящимися ко всему документу, настраивая параметры в заголовке YAML. Возможно, вас удивит расшифровка аббревиатуры YAML: "yet another markup language" (еще один язык разметки). Этот язык предназначен для представления иерархических данных способом, упрощающим их чтение и запись. Язык R Markdown использует YAML для управления многими деталями вывода. Ниже мы рассмотрим параметры документа и библиографические ссылки.

Параметры

Документы R Markdown могут включать один или несколько параметров, значения которых можно устанавливать при формировании отчета. Эти параметры полезны в тех случаях, когда вы хотите заново сформировать тот же отчет, но с отличающимися значениями важных входных данных. Например, вы можете создавать отчеты о продажах для каждого филиала, результатах сдачи экзаменов каждым из студентов или демографические сводки по разным странам. Для объявления одного или нескольких параметров используйте поле `params`.

В следующем примере параметр `my_class` определяет, данные о каком классе автомобилей должны отображаться.

```
---
output: html_document
params:
  my_class: "suv"
---

```{r setup, include = FALSE}
library(ggplot2)
library(dplyr)

class <- mpg %>% filter(class == params$my_class)
```

Fuel economy for `r params$my_class`s
```

```{r, message = FALSE}
ggplot(class, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
```

Как можно видеть, параметры присутствуют в коде фрагментов в виде списка `params`, доступного только для чтения.

Атомарные векторы можно включать непосредственно в YAML-заголовок. Вы также можете выполнять произвольные выражения, предваряя значение параметра префиксом `!r`. Это неплохой способ указания параметров даты/времени.

```
params:
  start: !r lubridate::ymd("2015-01-01")
  snapshot: !r lubridate::ymd_hms("2015-01-01 12:30:00")
```

В RStudio можно установить параметры, сформировать и предварительно просмотреть отчет, щелкнув на опции "Knit with Parameters" в раскрывающемся меню Knit. Вы можете настроить диалог, установив другие опции в заголовке. Более подробная информация по этому поводу доступна по адресу `http://bit.ly/ParamReports`.

Если необходимо создать много подобных параметризованных отчетов, можно поступить иначе, вызвав функцию `rmarkdown::render()` со списком `params`.

```
rmarkdown::render(
  "fuel-economy.Rmd",
  params = list(my_class = "suv")
)
```

Это особенно действенно в сочетании с вызовом функции `purrr:pwalk()`. В приведенном ниже примере создается отчет для каждого значения `class`, обнаруженного в `mpg`. Сначала мы создаем фрейм данных, который содержит по одной строке для каждого класса, предоставляя имя файла отчета и параметры.

```
reports <- tibble(
  class = unique(mpg$class),
  filename = stringr::str_c("fuel-economy-", class, ".html"),
  params = purrr::map(class, ~ list(my_class = .))
)
reports
#> # A tibble: 7 x 3
#>     class                  filename     params
```

```
#>     <chr>                      <chr>     <list>
#> 1 compact fuel-economy-compact.html <list [1]>
#> 2 midsize fuel-economy-midsize.html <list [1]>
#> 3     suv     fuel-economy-suv.html <list [1]>
#> 4 2seater fuel-economy-2seater.html <list [1]>
#> 5 minivan fuel-economy-minivan.html <list [1]>
#> 6  pickup  fuel-economy-pickup.html <list [1]>
#> # ... with 1 more rows
```

Затем мы сопоставляем имена столбцов с именами аргументов функции `render()` и используем вариант функции `walk()` из пакета **purrr**, предназначенный для *параллельных* вычислений, чтобы вызвать функцию `render()` по одному разу для каждой строки.

```
reports %>%
  select(output_file = filename, params) %>%
  purrr::pwalk(rmarkdown::render, input = "fuel-economy.Rmd")
```

Библиографические ссылки и цитаты

Конвертер **pandoc** может автоматически генерировать цитаты и библиографические ссылки в различных стилях. Чтобы воспользоваться этой возможностью, вы должны указать файл, содержащий библиографические ссылки, с помощью поля `bibliography` в заголовке своего файла. Это поле должно содержать путь из каталога, в котором содержится ваш *.Rmd*-файл, к файлу, содержащему библиографию:

```
bibliography: rmarkdown.bib
```

Вы можете использовать любой из множества распространенных библиографических форматов, включая BibLaTeX, BibTeX, *endnote* и *medline*.

Для создания цитаты в *.Rmd*-файле следует использовать ключ, состоящий из символа “@” *и идентификатора цитаты* из файла библиографии. Далее следует цитата, заключенная в квадратные скобки. Соответствующие примеры приведены ниже.

```
Для разделения нескольких цитат используйте символ `;`:
Blah blah [@smith04; @doe99].

Допускается включение комментариев в квадратных скобках:
Blah blah [see @doe99, pp. 33-35; also @smith04, ch. 1].

Встроенные цитаты создаются путем удаления квадратных скобок:
@smith04 says blah или @smith04 [p. 33] says blah.
```

```
Для подавления вывода имени автора поместите символ `-` перед цитатой:
Smith says blah [-@smith04].
```

В процессе визуализации вашего файла R Markdown создаст библиографический список и присоединит его к концу документа. Этот список будет включать каждую из цитируемых ссылок, содержащихся в библиографическом файле, но не будет содержать заголовок раздела. Как следствие, общепринятой практикой является завершение файла заголовком раздела библиографии, таким как `# References` или `# Bibliography`.

Стиль цитат и библиографических ссылок можно изменить, сославшись на файл с расширением *CSL* (от *Citation Style Language* — язык стилевого оформления цитат).

```
bibliography: rmarkdown.bib
csl: apa.csl
```

Как и в случае поля `bibliography`, ваш CSL-файл должен содержать путь к файлу. В данном случае мы предполагаем, что CSL-файл находится в одном каталоге с *.Rmd*-файлом. Неплохую коллекцию CSL-файлов для распространенных библиографических стилей можно найти по адресу `http://github.com/citation-style-language/styles`.

Дополнительная информация

Язык разметки R Markdown относительно новый и продолжает быстро развиваться. Лучшим местом для ознакомления с самыми последними новшествами, введенными в этот язык, является официальный сайт R Markdown, находящийся по адресу `http://rmarkdown.rstudio.com`.

Эта глава не охватывает две важные темы: организацию совместной работы и обмен идеями с другими людьми. Сотрудничество является неотъемлемой частью современной науки о данных, и вы можете значительно облегчить себе жизнь, используя такие средства управления версиями, как Git и GitHub. Для обучения работе с Git можно порекомендовать два следующих бесплатных источника.

- *Happy Git and GitHub for the useR* (Jenny Bryan). Книга предназначена для первого знакомства со средствами Git и GitHub и открыта для бесплатного доступа по адресу `http://happygitwithr.com/`.
- Глава “Git and GitHub” из книги *R Packages* (Хэдли Уикем). Также открыта для бесплатного доступа по адресу `http://r-pkgs.had.co.nz/git.html`.

ГЛАВА 22

Обмен графикой с помощью пакета ggplot2

Введение

В главе 5 вы ознакомились с использованием графиков в качестве исследовательского инструмента. Когда графики создаются для разведочных исследований, вы можете заранее, еще не видя графика, сказать, какие переменные будут на нем отображаться. При этом вы строите каждый график с определенной целью и после его быстрого изучения можете перейти к следующему графику. В большинстве случаев в ходе анализа создают десятки или сотни графиков, большую часть которых немедленно выбрасывают в корзину.

Разобравшись в том, что означают ваши данные, вы должны поделиться этим пониманием с другими людьми. Скорее всего, ваша аудитория не имеет того объема подготовки и знаний, которыми обладаете вы, и у нее не будет времени глубоко вникать в чужие данные. Чтобы облегчить другим людям задачу быстрого построения мысленной модели ваших данных, вы должны приложить значительные усилия к тому, чтобы графики в как можно большей степени говорили сами за себя. В этой главе вы познакомитесь с некоторыми инструментами, предоставляемыми для этой цели пакетом **ggplot2**.

В главе основное внимание уделяется инструментам, которые понадобятся вам для создания эффективной графики. Полагаем, вы ясно представляете, чего хотите, и вам остается лишь узнать, как это можно сделать. Потому настоятельно рекомендуем дополнительно прочитать какую-нибудь общую книгу, посвященную теме визуализации. Лично нам нравится книга *The Truthful Art* (Albert Cairo). В ней рассматриваются не чисто технические вопросы визуализации, а то, над чем вам следует задумываться для того, чтобы создавать эффективную графику.

Необходимые ресурсы

В этой главе в центре нашего внимания вновь будет находиться пакет **ggplot2**. В небольшой степени мы также будем использовать пакет **dplyr** для манипулирования данными и несколько пакетов расширений **ggplot2**, включая **ggrepel** и **viridis**. Вместо того чтобы загружать указанные пакеты, мы будем явно ссылаться на их функции с помощью нотации `::`. Благодаря этому вы будете точно знать, какие функции встроены в пакет **ggplot2**, а какие берутся из других пакетов. Не забывайте о том, что вам придется установить эти пакеты с помощью команды `install.packages()`, если это не было сделано ранее.

```
library(tidyverse)
```

Подписи

Рассмотрение темы перехода от исследовательской графики к объяснительной проще всего начать с обсуждения подписей. Подписи добавляют с помощью функции `labs()`. Вот простой пример добавления названия графика (рис. 22.1).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
   title = paste(
     "Fuel efficiency generally decreases with",
     "engine size"
   )
  )
```

Цель названия графика — предоставить краткие сведения об основном результате. Избегайте названий, которые лишь описывают, что собой представляет график, например: "A scatterplot of engine displacement vs. fuel economy".

В тех случаях, когда необходимо добавить больше текста, можете воспользоваться двумя другими полезными вариантами создания подписей, доступными в версиях пакета **ggplot2** 2.2.0 и более новых (которые уже должны быть созданы к тому времени, когда вы будете читать эту книгу):

- параметр `subtitle` добавляет дополнительное подробное описание, выводимое под названием более мелким шрифтом;
- параметр `caption` добавляет в правом нижнем углу графика текст, который часто используют для описания источника данных (рис. 22.2).

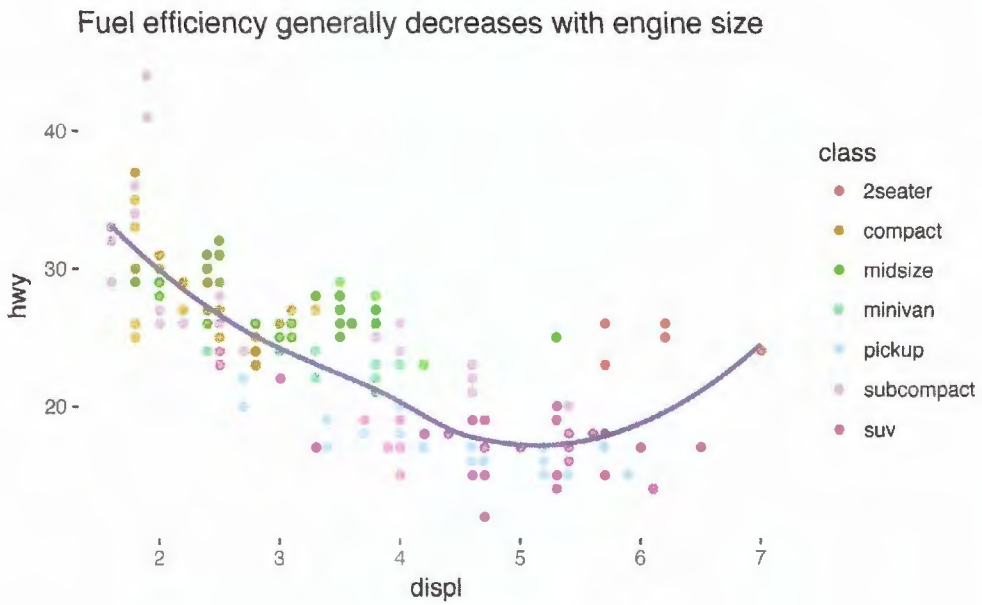


Рис. 22.1

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    title = paste(
      "Fuel efficiency generally decreases with",
      "engine size"
    ),
    subtitle = paste(
      "Two seaters (sports cars) are an exception",
      "because of their light weight"
    ),
    caption = "Data from fueleconomy.gov"
  )
```

Функция `labs()` позволяет изменить названия осей и легенды. Обычно всегда неплохо заменить короткие имена переменных более подробными описаниями и указать единицы измерения (рис. 22.3).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Engine displacement (L)",
```

```
    y = "Highway fuel economy (mpg)",
    colour = "Car type"
)
```

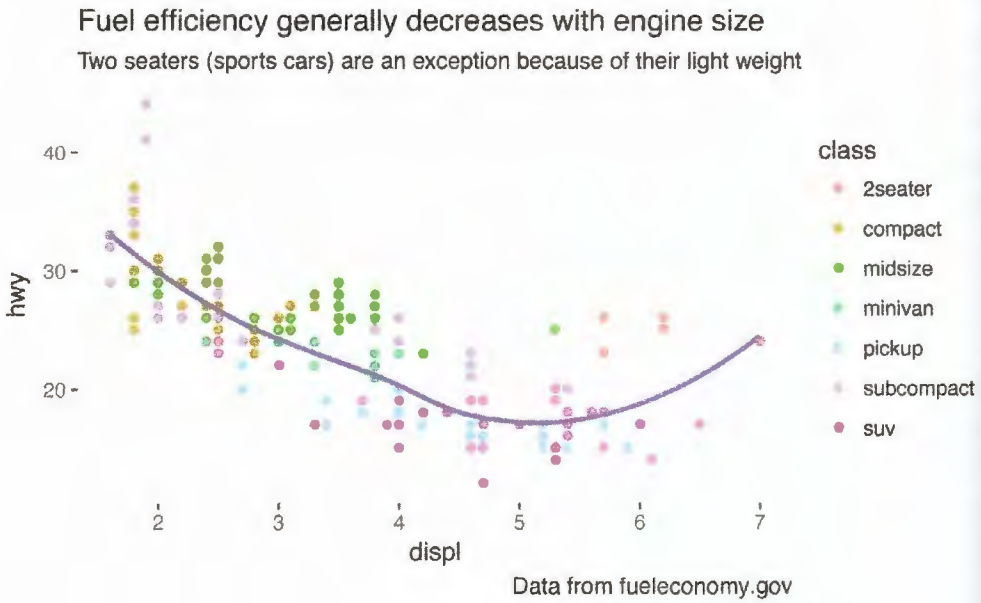


Рис. 22.2

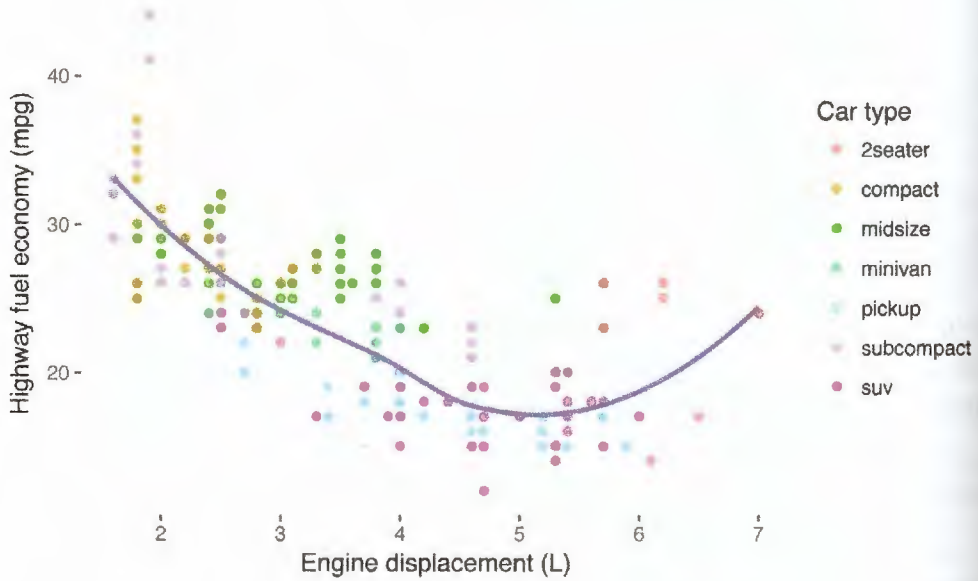


Рис. 22.3

Вместо текстовых строк можно использовать математические выражения. Для этого достаточно отказаться от кавычек и использовать функцию `quote()`, предварительно ознакомившись с доступными опциями с помощью команды `?plotmath` (рис. 22.4).

```
df <- tibble(
  x = runif(10),
  y = runif(10)
)
ggplot(df, aes(x, y)) +
  geom_point() +
  labs(
    x = quote(sum(x[i] ^ 2, i == 1, n)),
    y = quote(alpha + beta + frac(delta, theta))
  )
```

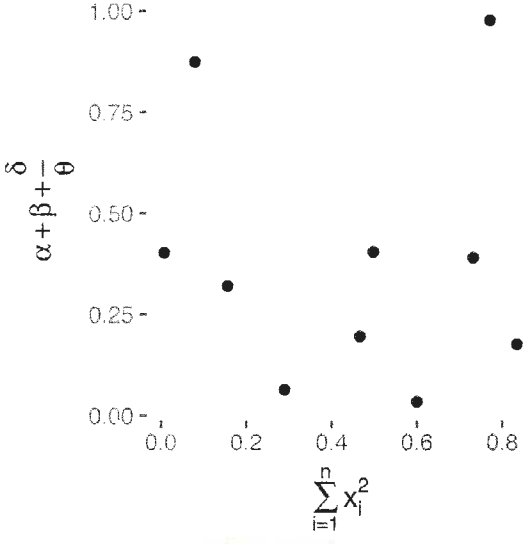


Рис. 22.4

Упражнения

1. Создайте один график на основе данных по расходу топлива, изменив в нем заголовок (путем включения в него дополнительного подзаголовка) и подписи осей и включив цветные метки.
2. Функция `geom_smooth()` может вводить в заблуждение, поскольку создает некоторый перекос в значениях переменной `hwy` в большую сторону для многолитражных двигателей, вызванный данными для легких

спортивных автомобилей с мощными двигателями. Используйте свои средства моделирования для построения и отображения лучшей модели.

3. Добавьте в какой-либо из созданных вами за последний месяц исследовательских графиков информативные названия, облегчающие понимание графика другими людьми.

Аннотации

Кроме добавления подписей к основным компонентам графика, часто целесообразно снабжать подписями отдельные наблюдения или группы наблюдений. В первую очередь обратите внимание на функцию `geom_text()`. Она аналогична функции `geom_point()`, но имеет дополнительный визуальный атрибут: `label`. Это позволяет добавлять на график текстовые метки.

Существуют два возможных источника меток. Прежде всего, у вас может быть tibble-фрейм, который их предоставляет. Приведенный ниже график не особенно полезен, но он иллюстрирует один эффективный подход, суть которого сводится к определению наиболее экономичного автомобиля в каждом классе и снабжению его подписью на графике (рис. 22.5).

```
best_in_class <- mpg %>%
  group_by(class) %>%
  filter(row_number(desc(hwy)) == 1)
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_text(aes(label = model), data = best_in_class)
```

Разобраться в этом графике очень трудно из-за наложения подписей на другие подписи и точки данных. Это можно немного улучшить, использовав функцию `geom_label()`, которая вычерчивает прямоугольник позади текста (рис. 22.6). Кроме того, мы используем параметр `nudge_y` для незначительного смещения подписей в направлении вверх от соответствующих точек.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_label(
    aes(label = model),
    data = best_in_class,
    nudge_y = 2,
    alpha = 0.5
  )
```

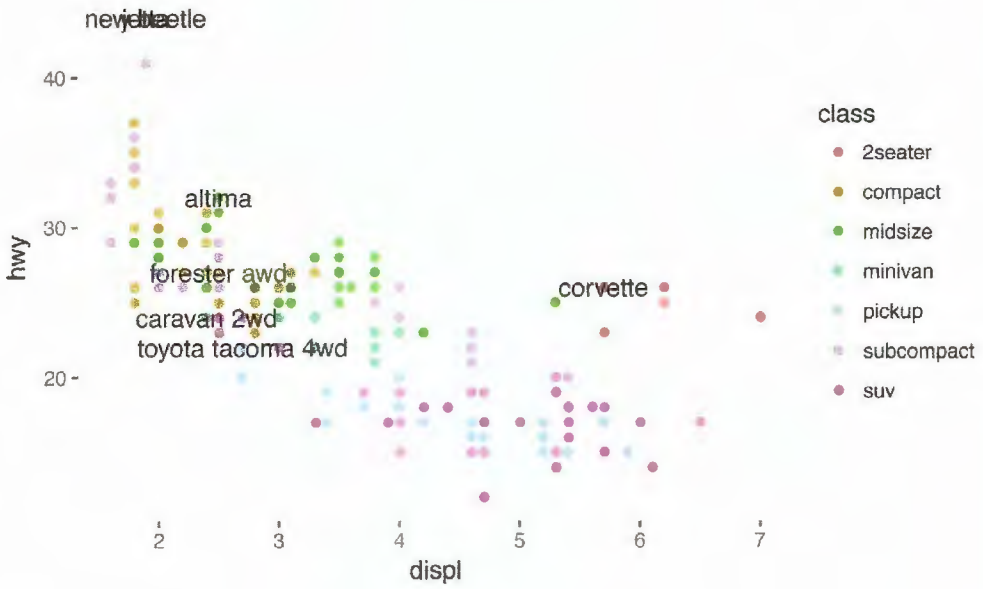


Рис. 22.5

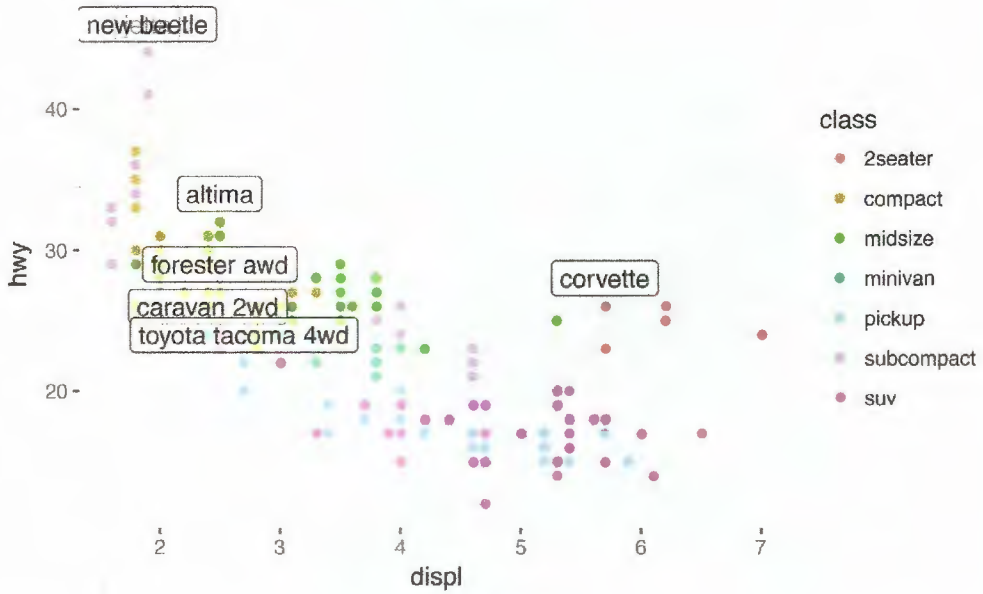


Рис. 22.6

Это немного помогло, но если внимательно присмотреться к левому верхнему углу графика, то можно заметить, что две подписи практически полностью перекрываются. Так произошло потому, что значения переменных `hwy`

и `displ` для наиболее экономичных компактных и субкомпактных автомобилей в точности совпадают. Не существует способа исправить этот недостаток путем применения одного и того же преобразования к каждой из подписей. Вместо этого можно использовать пакет **ggrepel** (разработчик Камиль Словиковски). Он автоматически корректирует расположение подписей таким образом, чтобы они не перекрывались (рис. 22.7).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_point(size = 3, shape = 1, data = best_in_class) +
  ggrepel::geom_label_repel(
    aes(label = model),
    data = best_in_class
  )
```

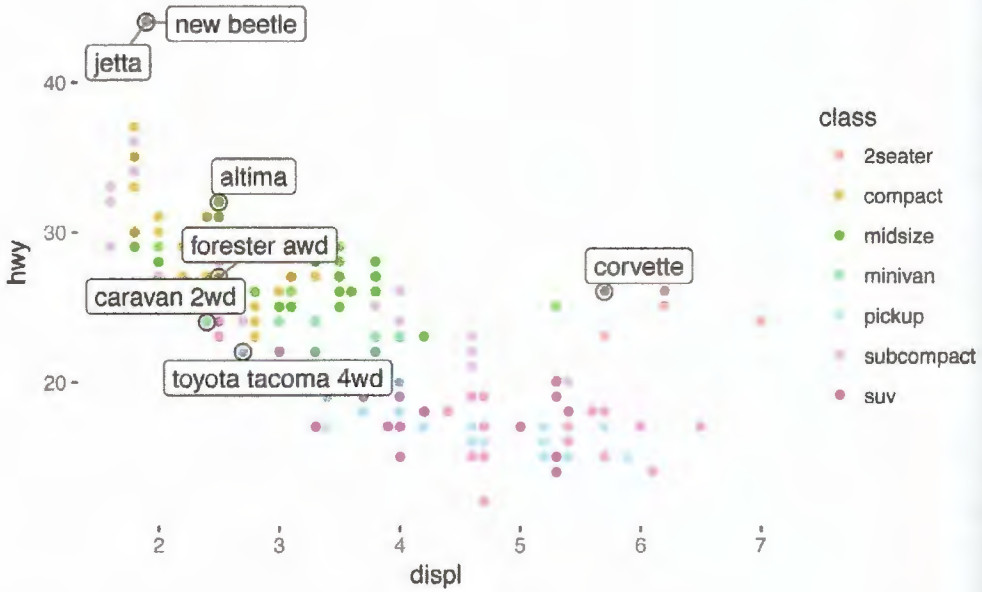


Рис. 22.7

Обратите внимание на еще один полезный прием, который был здесь использован: для выделения точек, снабженных подписями, мы добавили второй слой с крупными полыми маркерами точек данных.

Иногда ту же идею можно использовать для замены легенды подписями, расположенными непосредственно на графике (рис. 22.8). Для данного графика этот прием не очень подходит, но в целом он довольно эффективен. (Для отключения легенды можно воспользоваться командой `theme(legend.position = "none")`; вскоре мы об этом поговорим.)

```
class_avg <- mpg %>%
  group_by(class) %>%
  summarize(
    displ = median(displ),
    hwy = median(hwy)
  )
ggplot(mpg, aes(displ, hwy, color = class)) +
  ggrepel::geom_label_repel(aes(label = class),
    data = class_avg,
    size = 6,
    label.size = 0,
    segment.color = NA
  ) +
  geom_point() +
  theme(legend.position = "none")
```

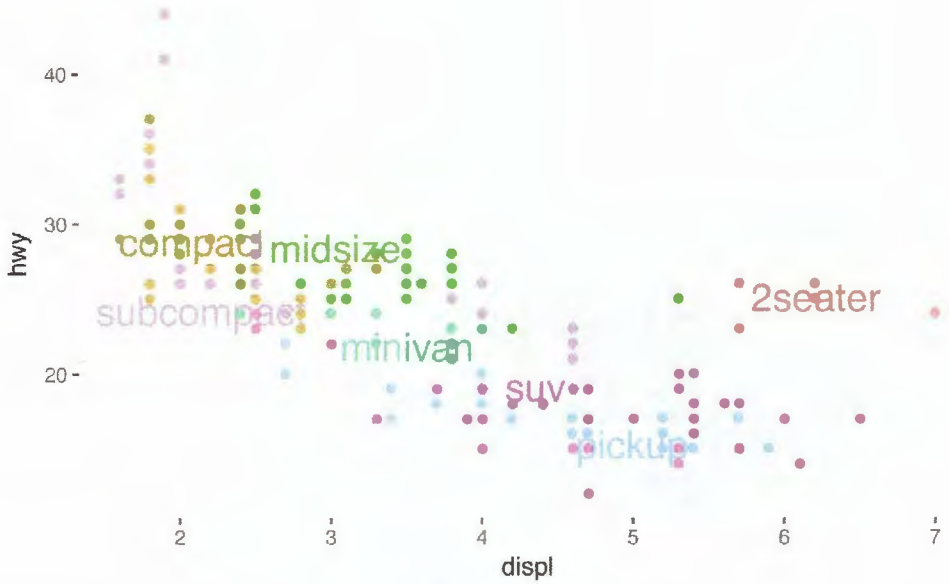


Рис. 22.8

Возможно, вы захотите добавить на график только одну подпись, но для этого вам все равно потребуется создать фрейм данных. Часто желательно добавить подпись в углу графика, поэтому для вычисления максимальных значений `x` и `y` удобно создать новый фрейм данных, используя функцию `summarize()` (рис. 22.9).

```
label <- mpg %>%
  summarize(
    displ = max(displ),
```

```
    hwy = max(hwy),
    label = paste(
      "Increasing engine size is \nrelated to"
      "decreasing fuel economy."
    )
  )

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(
    aes(label = label),
    data = label,
    vjust = "top",
    hjust = "right"
  )
```

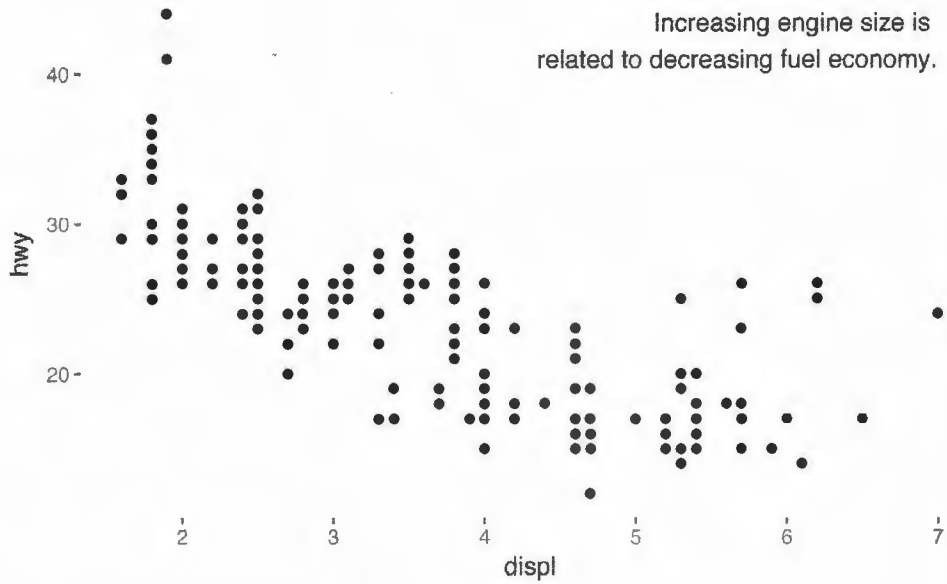


Рис. 22.9

Если хотите расположить текст точно вдоль границ графика, используйте опции `+Inf` и `-Inf`. Поскольку теперь для вычисления позиции не требуется использовать данные `mpg`, мы можем создать фрейм данных с помощью функции `tibble()` (рис. 22.10).

```
label <- tibble(
  displ = Inf,
  hwy = Inf,
  label = paste(
```

```
    "Increasing engine size is \nrelated to"
    "decreasing fuel economy."
  )
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_text(
    aes(label = label),
    data = label,
    vjust = "top",
    hjust = "right"
)
```

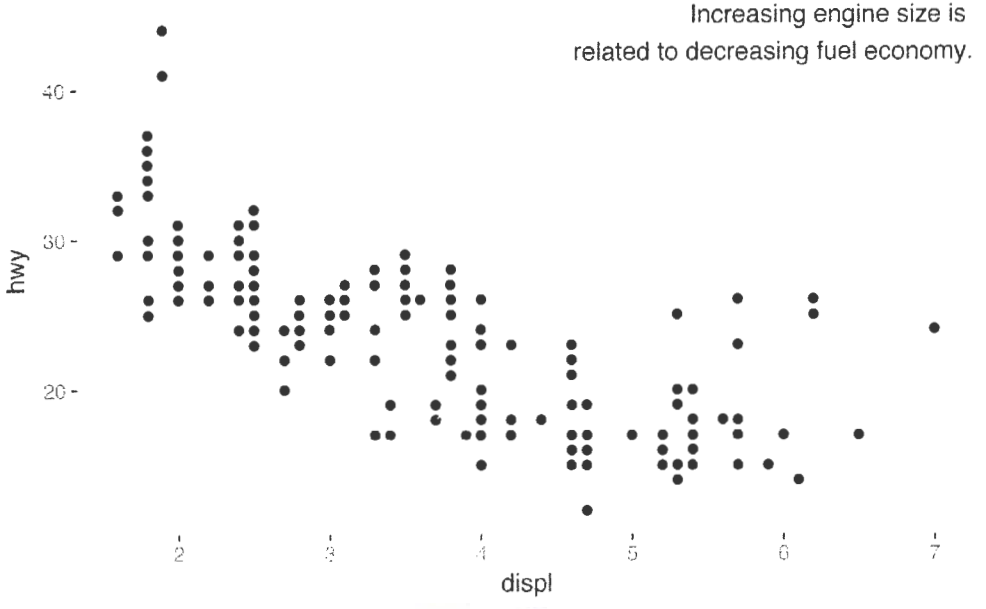


Рис. 22.10

В этих примерах мы вручную переносили текст на другую строку, используя символы `"\n"`. Другой подход заключается в автоматическом добавлении символов разрыва строки с помощью функции `stringr::str_wrap()`, которой в качестве аргумента передается требуемое количество символов в строке.

```
"Increasing engine size related to decreasing fuel economy." %>%
  stringr::str_wrap(width = 40) %>%
  writeLines()
#> Increasing engine size is related to
#> decreasing fuel economy.
```

Обратите внимание на опции `hjust` и `vjust`, используемые для управления выравниванием подписи. Девять возможных комбинаций их значений показаны на рис. 22.11.

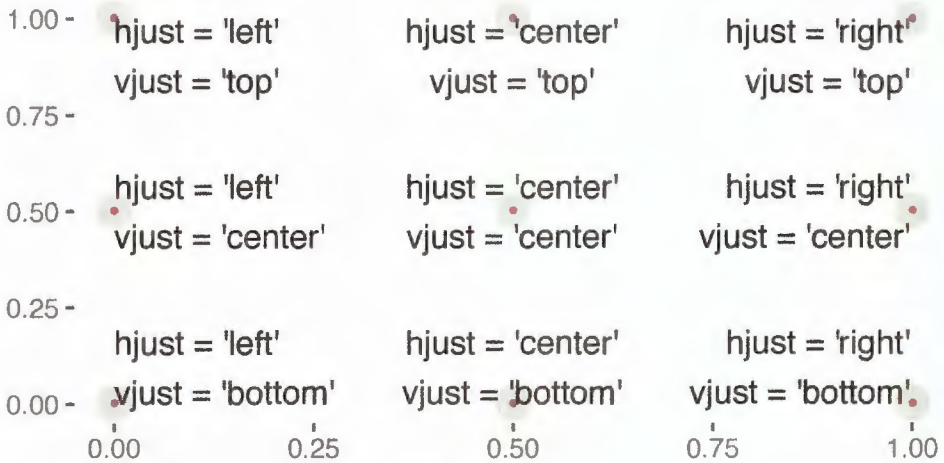


Рис. 22.11

Не забывайте о том, что в дополнение к функции `geom_text()` пакет **ggplot2** предлагает много других функций данного семейства, которые помогут вам аннотировать свои графики. Ниже приведено несколько идей по этому поводу.

- Используйте функции `geom_hline()` и `geom_vline()` для добавления опорных линий. Мы часто делаем их толстыми (`size = 2`) и белого цвета (`color = white`) и создаем их под слоем первичных данных. Благодаря этому они отчетливо видны, но в то же время не отвлекают внимание от данных.
- Используйте функцию `geom_rect()` для вычерчивания прямоугольников вокруг интересующих вас точек. Границы прямоугольника определяются параметрами эстетики `xmin`, `xmax`, `ymin` и `ymax`.
- Используйте функцию `geom_segment()` с аргументом `arrow` для привлечения внимания к точке данных с помощью стрелки. Начальная позиция стрелки определяется параметрами эстетики `x` и `y`, а конечная — параметрами `xend` и `yend`.

Границы возможного зависят лишь от вашего воображения (и терпения при подборе позиций аннотаций, обеспечивающих привлекательный вид графика)!

Упражнения

1. Используйте функцию `geom_text()` с бесконечными (`Inf`) позициями для расположения текста во всех четырех углах графика.
2. Прочитайте документацию к функции `annotate()`. Как ее можно использовать для добавления текстовой подписи в график без создания tibble-фрейма?
3. Как создать подписи с помощью функции `geom_text()`, используя представление различных аспектов на отдельных панелях? Как добавить подпись в один аспект? Как поместить разные подписи в разные аспекты? (Подсказка: подумайте о базовых данных.)
4. Какие аргументы функции `geom_label()` управляют внешним видом фонового прямоугольника?
5. Перечислите четыре аргумента функции `arrow()`. Как они работают? Создайте серию графиков, демонстрирующих работу наиболее важных опций.

Шкалы

Третьим способом улучшения графиков, которыми вы собираетесь делиться с другими людьми, является настройка шкал. Шкалы управляют преобразованием значений данных в воспринимаемые человеческим глазом объекты. Как правило, **ggplot2** автоматически добавляет шкалы. Например, если вы введете такую команду:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))
```

то **ggplot2** автоматически добавит шкалы, предусмотренные по умолчанию.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_color_discrete()
```

Обратите внимание на схему именования шкал: за префиксом `scale_` следует имя визуального атрибута, затем символ подчеркивания (_), за которым следует наименование шкалы. Шкалы, используемые по умолчанию, именуются в соответствии с типом переменной, которую они представляют:

continuous, discrete, datetime или date. Существует множество других шкал, не относящихся к категории используемых по умолчанию, с которыми вы также вскоре познакомитесь.

Используемые по умолчанию шкалы тщательно подбирались таким образом, чтобы они хорошо справлялись со своей работой для широкого диапазона входных данных. Желание заменить эти шкалы может возникнуть у вас по двум причинам.

- Возможно, вы захотите скорректировать параметры шкалы, используемой по умолчанию, например изменить метки на осях или подписи ключей легенды.
- Возможно, вы захотите заменить вообще всю шкалу и использовать совершенно другой алгоритм. Во многих случаях ваш вариант может оказаться более предпочтительным, поскольку вы лучше знаете данные, с которыми работаете.

Метки на осях и ключи легенды

Внешним видом меток на осях и ключей легенды в основном управляют два аргумента: breaks и labels. Первый управляет положением меток или значениями, связанными с ключами, а второй — текстовыми подписями, связанными с каждой меткой/ключом. Чаще всего аргумент breaks используется для замены значений, заданных по умолчанию (рис. 22.12).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```

Точно так же можно использовать и аргумент labels (символьный вектор той же длины, что и вектор breaks), но для него можно указать значение NULL, чтобы полностью подавить вывод подписей. Это может пригодиться при создании карт или публикации графиков, абсолютные данные которых вы не хотите предоставлять широкой публике (рис. 22.13).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_x_continuous(labels = NULL) +
  scale_y_continuous(labels = NULL)
```

Опции breaks и labels также можно использовать для управления внешним видом легенд. Оси и легенды совокупно называют *ориентирами* (guides).

Оси используются для визуальных атрибутов x и y, легенды — для всего остального.

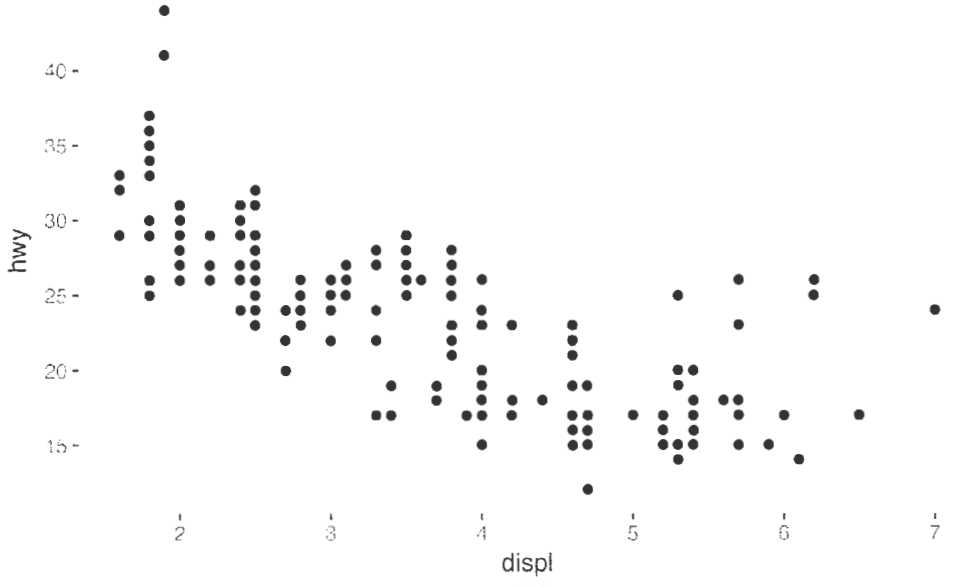


Рис. 22.12

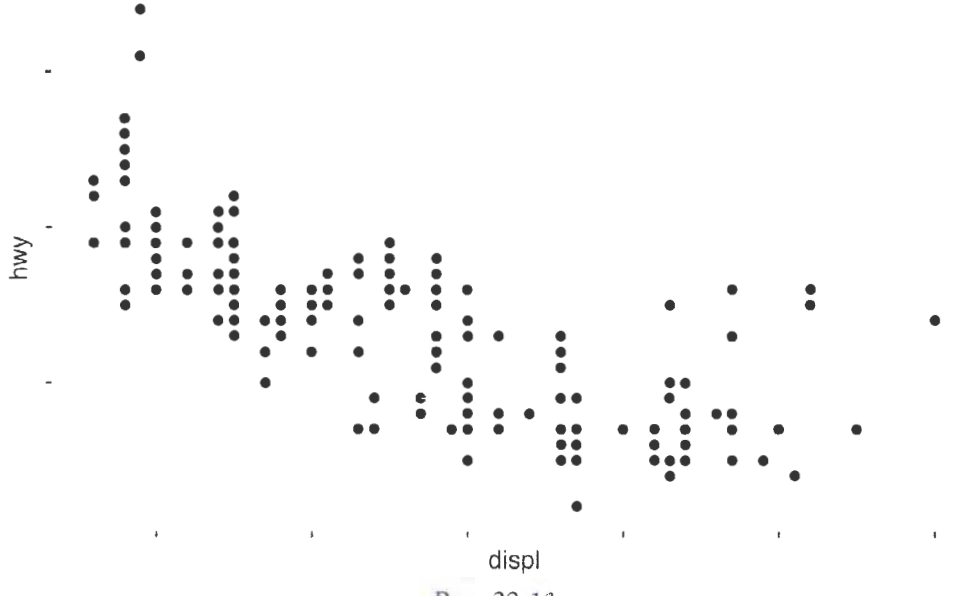


Рис. 22.13

Опция breaks пригодится вам и в тех случаях, когда количество точек данных сравнительно невелико и вы хотите точно обозначить ту точку на оси, в которой встречается наблюдение. Рассмотрим, например, следующий график, на котором отображаются начало и конец срока пребывания на посту каждого из президентов США (рис. 22.14).

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = id)) +
    scale_x_date(
      NULL,
      breaks = presidential$start,
      date_labels = "'%y"
    )
```

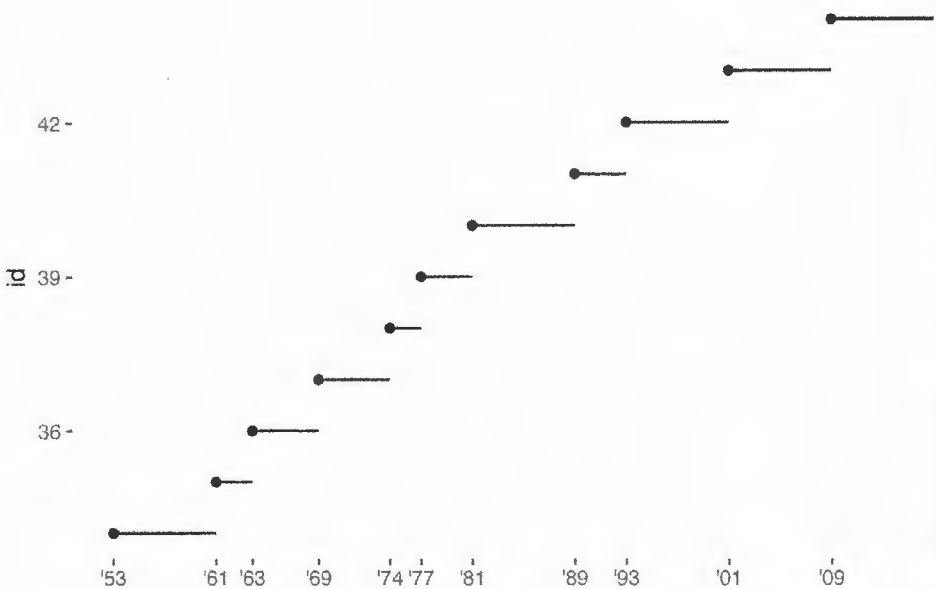


Рис. 22.14

Обратите внимание на то, что спецификации breaks и labels для шкал даты и даты/времени несколько различаются:

- date_labels принимает спецификацию формата в том же виде, что и функция parse_datetime();
- date_breaks (здесь отсутствует) принимает строку наподобие "2 days" или "1 month".

Компоновка легенды

Опции breaks и labels чаще всего применяют для настройки параметров осей. И хотя с их помощью можно также настраивать параметры легенд, для этой цели, вероятнее всего, вы будете использовать иные средства.

Для управления общим расположением легенды вам потребуется функция theme(). К рассмотрению тем мы еще вернемся в конце главы, а сейчас лишь отметим, что темы управляют компонентами графика, не являющимися данными. Параметр legend.position темы определяет, в каком месте графика будет отображаться легенда.

```
base <- ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class))

base + theme(legend.position = "left")
base + theme(legend.position = "top")
base + theme(legend.position = "bottom")
base + theme(legend.position = "right") # по умолчанию
```

Опция legend.position = "none" полностью отключает отображение легенды.

Отображением отдельных легенд можно управлять с помощью функции guides(), используемой совместно с функцией guide_legend() или guide_colorbar(). В следующем примере продемонстрирована работа двух важных настроек: управление количеством строк, используемых легендой, с помощью параметра nrow и замена одного из визуальных атрибутов для увеличения размера точек. Это оказывается особенно полезным в случае использования низкого значения параметра alpha при отображении большого количества точек на графике (рис. 22.15).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme(legend.position = "bottom") +
  guides(
    color = guide_legend(
      nrow = 1,
      override.aes = list(size = 4)
    )
  )
#> `geom_smooth()` using method = 'loess'
```

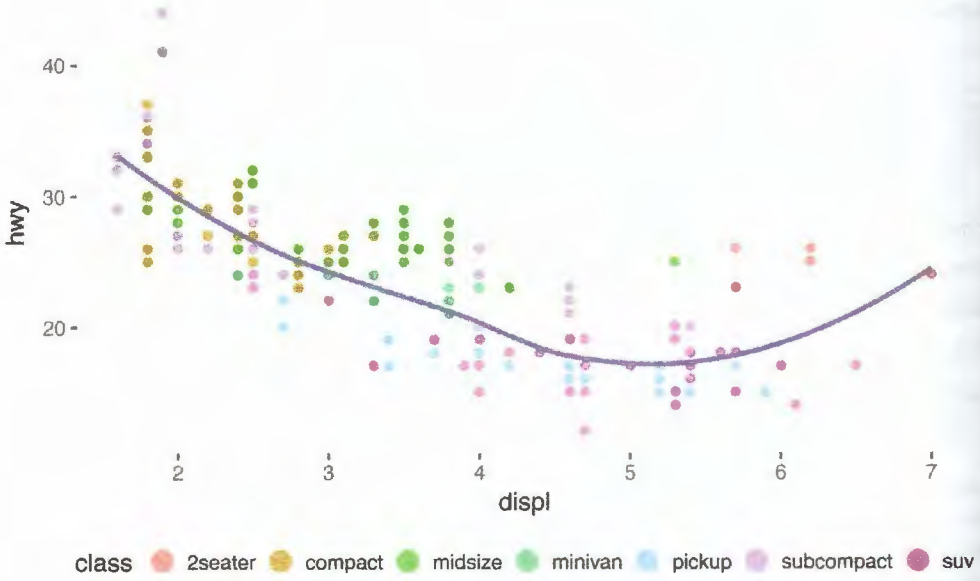


Рис. 22.15

Замена шкалы

Вместо того чтобы лишь слегка подправлять некоторые детали шкалы, вы можете полностью заменить ее. Существуют два типа шкал, к которым вы в большинстве случаев будете переходить: непрерывные позиционные и цветовые шкалы. К счастью, ко всем остальным эстетическим атрибутам применимы одни и те же принципы, поэтому, как только вы освоите работу с позициями и цветом, вы очень быстро научитесь выбирать другие варианты замены шкал.

Чрезвычайно полезно строить графики с использованием преобразованных переменных. Например, как было показано в главе 19, увидеть истинное соотношение между весом бриллиантов и их ценой гораздо легче, если использовать логарифмическое преобразование переменных.

```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d()

ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_bin2d()
```

Недостатком такого подхода является то, что теперь подписи соответствуют преобразованным значениям переменных, и это затрудняет

интерпретацию графика. Вместо того чтобы выполнять преобразование переменных в функции aes(), можно преобразовать шкалы. Визуальная картина от этого никак не изменится, но теперь метки вдоль осей будут соответствовать первоначальным шкалам (рис. 22.16).

```
ggplot(diamonds, aes(carat, price)) +
  geom_bin2d() +
  scale_x_log10() +
  scale_y_log10()
```

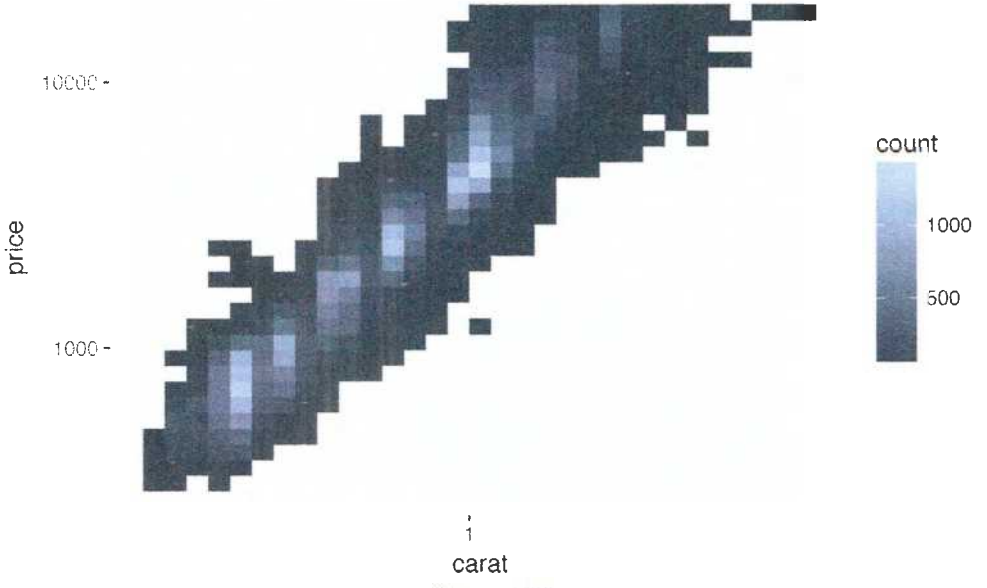


Рис. 22.16

Другой шкалой, которую приходится часто настраивать, является шкала цвета. Используемая по умолчанию категориальная шкала выбирает цвета, равномерно распределенные по цветовому кругу. Полезной альтернативой являются шкалы ColorBrewer, которые настраивались вручную таким образом, чтобы учесть особенности восприятия цвета людьми с распространенными типами цветовой слепоты. Следующие два графика выглядят похожими, но различия в оттенках красного и зеленого цветов достаточны для того, чтобы точки на графике справа могли различить даже люди с нарушениями цветовосприятия в красной и зеленой областях спектра.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv))
```

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv)) +
  scale_color_brewer(palette = "Set1")
```

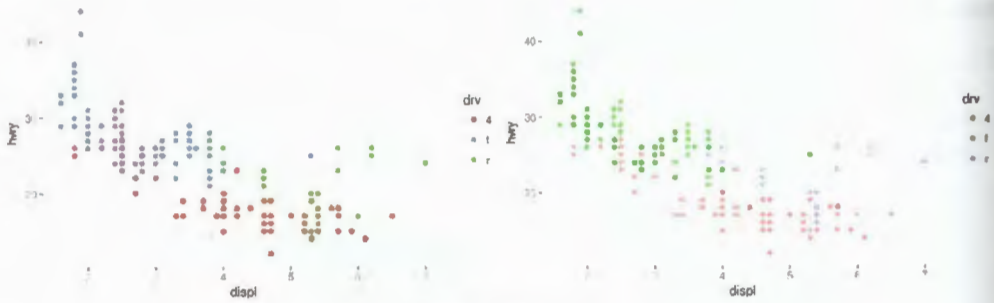


Рис. 22.17

Не забывайте и о более простых подходах. Если имеется всего лишь несколько цветов, можно ввести дополнительный параметр отображения, задающий форму точек. Это также будет служить гарантией того, что ваш график будет поддаваться интерпретации даже в случае черно-белого изображения.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = drv, shape = drv)) +
  scale_color_brewer(palette = "Set1")
```

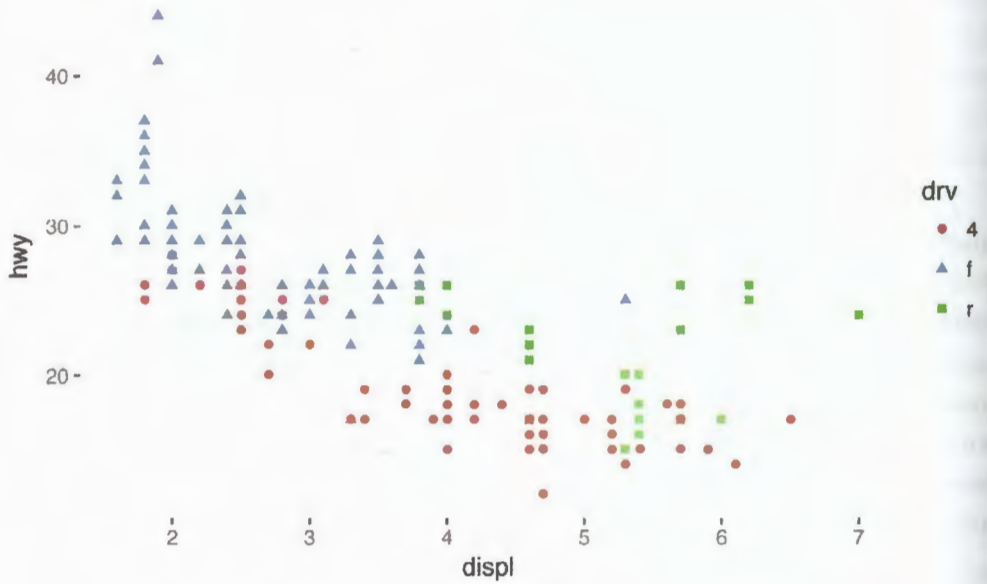


Рис. 22.18

Шкалы ColorBrewer задокументированы по адресу http://colorbrewer2.org/ и доступны в R через пакет **RColorBrewer**, разработанный Эрихом Нойвиртом. На рис. 22.19 представлен полный список всех палитр. Последовательные (вверху) и дивергентные (внизу) палитры особенно полезны в тех случаях, когда категориальные значения упорядочены или имеют естественную центральную точку. Такие ситуации часто возникают при превращении непрерывной переменной в категориальную с помощью функции cut().

При наличии заранее установленного соответствия между значениями и цветами используйте функцию scale_color_manual(). Например, если мы сопоставим президентские партии с цветами, то сможем использовать стандартное сопоставление красного цвета республиканцам и синего — демократам (рис. 22.20).

```
presidential %>%
  mutate(id = 33 + row_number()) %>%
  ggplot(aes(start, id, color = party)) +
    geom_point() +
    geom_segment(aes(xend = end, yend = id)) +
    scale_colour_manual(
      values = c(Republican = "red", Democratic = "blue")
)
```

Для непрерывного цвета можно использовать встроенную функцию scale_color_gradient() или scale_fill_gradient(). В случае дивергентной шкалы используйте функцию scale_color_gradient2(). Например, вы могли бы воспользоваться ею для выделения разными цветами положительных и отрицательных значений или точек, расположенных выше и ниже среднего значения.

Другие возможности предлагает функция scale_color_viridis(), входящая в состав пакета **viridis**. Она представляет собой непрерывный аналог категориальных шкал ColorBrewer. Ее создатели, Натаниэль Смит и Стивен Ван дер Вальт, тщательно подобрали цветовую схему со свойствами, оптимальными для цветовосприятия разными людьми. Вот один из примеров, иллюстрирующих работу пакета **viridis** (рис. 22.21).

YlOrRd
YlOrBr
YlGnBu
YlGn
Reds
RdPu
Purples
PuRd
PuBuGn
PuBu
OrRd
Oranges
Greys
Greens
GnBu
BuPu
BuGn
Blues
Set3
Set2
Set1
Pastel2
Pastel1
Paired
Dark2
Accent
Spectral
RdYlGn
RdYlBu
RdGy
RdBu
PuOr
PRGn
PiYG
BrBG

Рис. 22.19

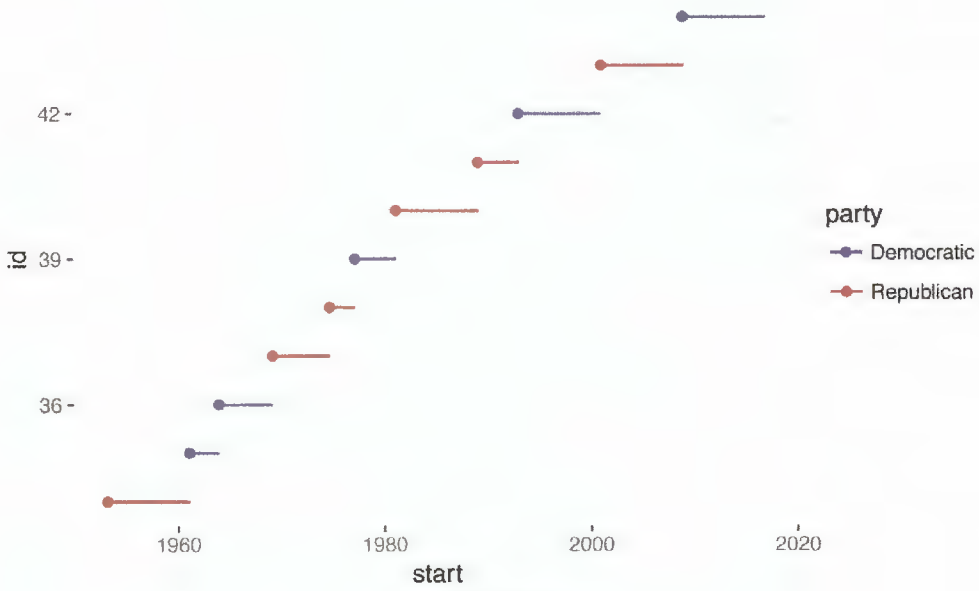


Рис. 22.20

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)
ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed()
#> Loading required package: methods

ggplot(df, aes(x, y)) +
  geom_hex() +
  viridis::scale_fill_viridis() +
  coord_fixed()
```

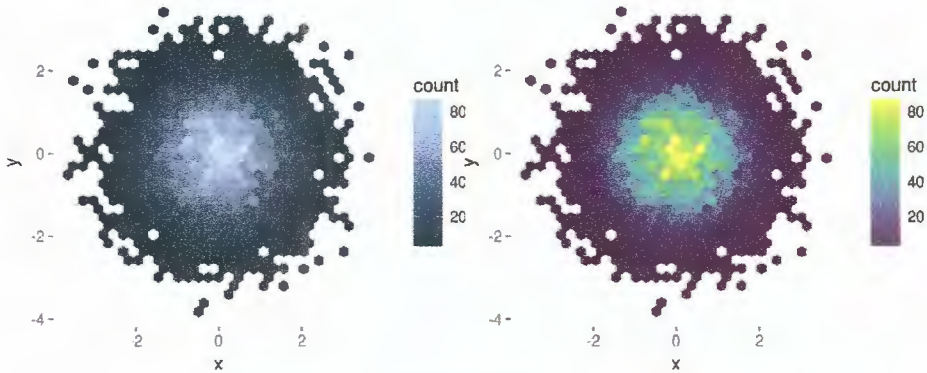


Рис. 22.21

Заметьте, что все цветовые шкалы предоставляются в двух разновидностях, `scale_color_x()` и `scale_fill_x()`, предназначенных для использования соответственно в качестве цвета линий и цвета заливки (варианты написания `color` и `colour` эквивалентны).

Упражнения

1. Почему приведенный ниже код не заменит шкалу, используемую по умолчанию?

```
ggplot(df, aes(x, y)) +
  geom_hex() +
  scale_color_gradient(low = "white", high = "red") +
  coord_fixed()
```

2. Что указывается в качестве первого аргумента для каждой шкалы? Как это соотносится с заданием аргументов функции `labs()`?
3. Измените способ отображения сроков пребывания в должности президента путем:
   а) сочетания двух представленных выше вариантов;
   б) улучшения отображения оси *y*;
   в) снабжения каждого срока подписью с именем президента;
   г) добавления в график информативных подписей;
   д) добавления меток через каждые четыре года (это немного сложнее, чем может показаться).
4. Используйте опцию `override.aes` для улучшения удобочитаемости легенды следующего графика (рис. 2.22).

```
ggplot(diamonds, aes(carat, price)) +
  geom_point(aes(color = cut), alpha = 1/20)
```

Изменение масштаба изображения

Управлять границами графиков можно тремя способами:

- подбором данных, отображаемых на графике;
- установкой границ каждой шкалы;
- заданием аргументов `xlim` и `ylim` функции `coord_cartesian()`.

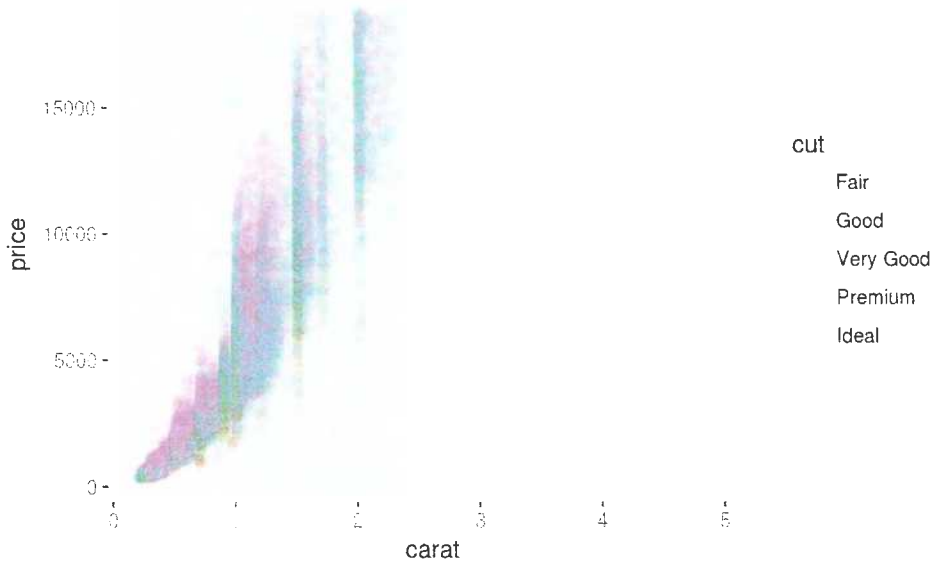


Рис. 22.22

Обычно для увеличения какой-либо области графика лучше всего использовать функцию `coord_cartesian()`. Сравните между собой следующие два графика.

```
ggplot(mpg, mapping = aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 7), ylim = c(10, 30))

mpg %>%
  filter(displ >= 5, displ <= 7, hwy >= 10, hwy <= 30) %>%
  ggplot(aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth()
```

Вы также можете устанавливать граничные значения для отдельных шкал. Уменьшение граничных значений фактически эквивалентно отображению лишь поднаборов данных. Обычно этот способ наиболее полезен в тех случаях, когда вы хотите расширить пределы графика, скажем, для приведения в соответствие шкал различных графиков. Например, если вы извлечете два класса автомобилей и построите для них отдельные графики, то эти графики будет трудно сравнивать между собой, поскольку диапазоны трех шкал (соответствующих осям `x` и `y` и эстетике цвета) будут разными.

```
suv <- mpg %>% filter(class == "suv")
compact <- mpg %>% filter(class == "compact")

ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point()

ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point()
```

Один из способов преодоления этой проблемы заключается в совместном использовании шкал несколькими графиками, границы которых определяются на основе полного набора данных.

```
x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_color_discrete(limits = unique(mpg$drv))

ggplot(suv, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale

ggplot(compact, aes(displ, hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```

В данном конкретном примере вы могли бы просто использовать многоаспектное отображение данных на нескольких панелях, но обсуждаемая техника полезна в более общих случаях, например, если вы хотите распределить графики на несколько страниц отчета.

Темы

Наконец, для настройки элементов графиков, не являющихся данными, можно использовать темы (рис. 22.23).

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```

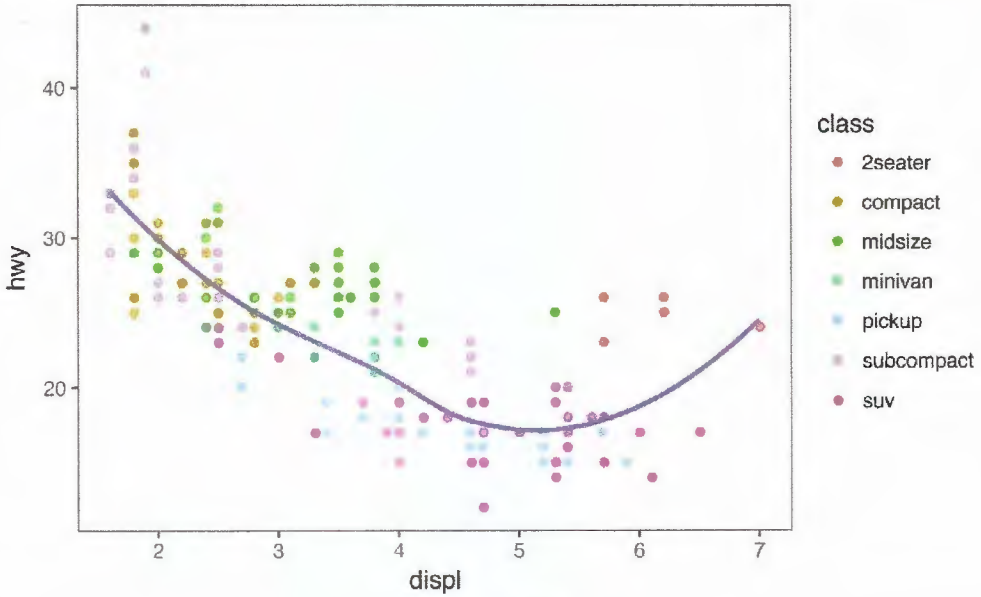


Рис. 22.23

Пакет **ggplot2** включает восемь встроенных тем (рис. 22.24). Гораздо больше тем содержится в пакетах расширения, таких как пакет **ggthemes**, разработанный Джеффри Арнольдом.

Многих удивляет, почему в теме, заданной по умолчанию, используется серый фон. Это был сознательный выбор, поскольку данные в таком случае выходят на первый план, а линии сетки остаются легко различимыми. Белые линии сетки хорошо видны (это важно, поскольку они значительно облегчают определение позиций данных), но не напрягают зрение, и мы можем легко абстрагироваться от них. Серый фон графика близок к типографскому цвету текста, благодаря чему графика, встроенная в документ, не будет создавать резкий цветовой переход, как было бы в случае белого цвета фона. Наконец, серый фон создает сплошное цветовое поле, в результате чего график воспринимается как единый визуальный объект.

Также существует возможность управления отдельными компонентами каждой темы, такими, например, как размер и цвет шрифта, используемого для оси *y*. К сожалению, рассмотрение этого уровня детализации выходит за рамки нашей книги, и для получения более полной информации по данной теме вам следует обратиться к документации пакета **ggplot2**. Вы также можете создавать собственные темы, если хотите следовать какому-либо конкретному корпоративному или журнальному стилю.

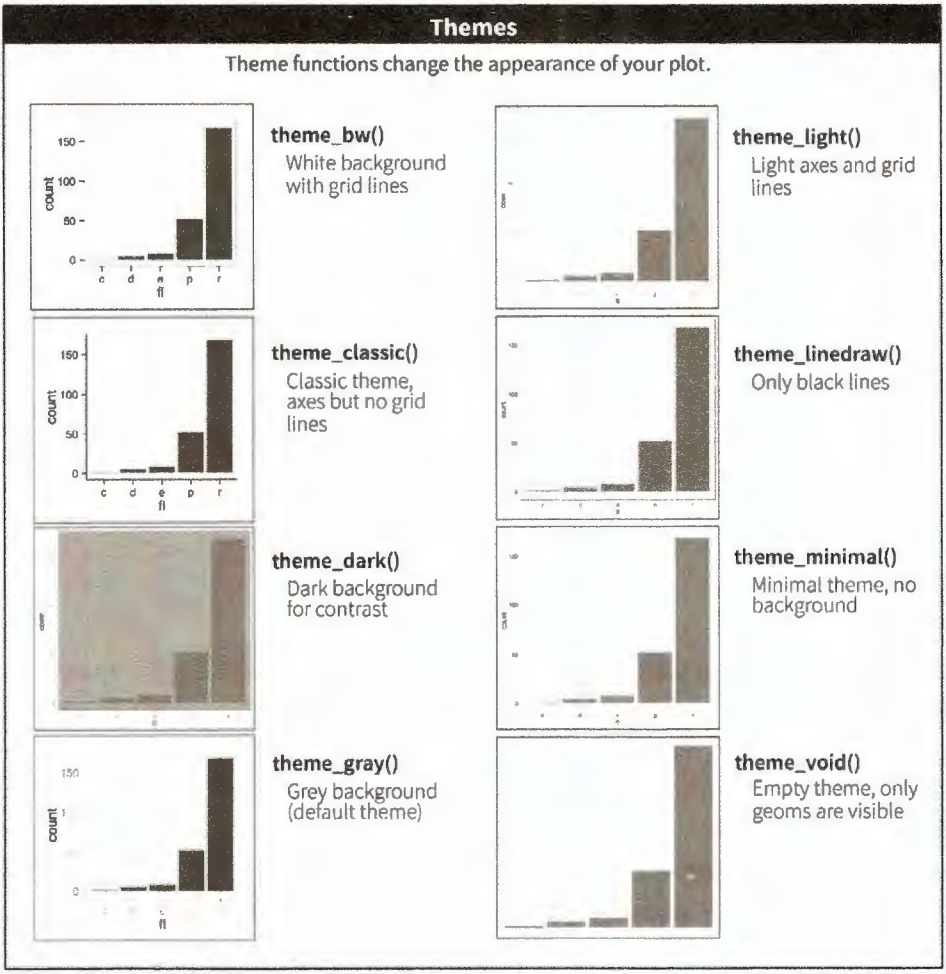


Рис. 22.24

Сохранение графиков

Для сохранения графиков, созданных в R, в виде файла на жестком диске предусмотрены два основных способа: с помощью функции `ggsave()` и пакета **knitr**. Функция `ggsave()` сохраняет самую последнюю версию графика (рис. 22.25):

```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```

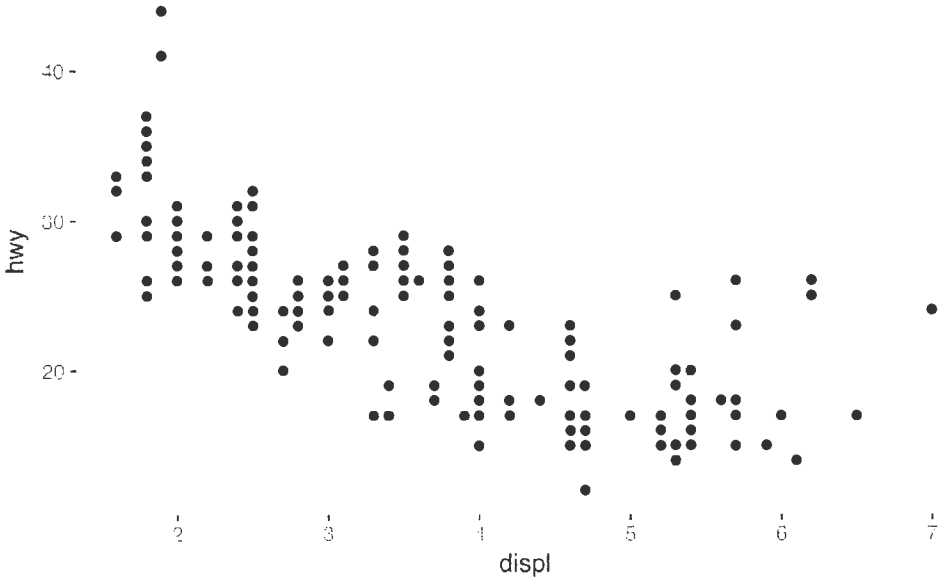


Рис. 22.25

```
ggsave("my-plot.pdf")
#> Saving 6 x 3.71 in image
```

Если ширина и высота графика не указаны, то в качестве соответствующих значений будут взяты размеры листа, установленные для текущего устройства вывода. Старайтесь всегда указывать размеры графика, чтобы обеспечить воспроизводимость кода.

Однако мы полагаем, что в большинстве случаев вы будете готовить окончательные отчеты с помощью R Markdown, в связи с чем рекомендуем внимательно ознакомиться с опциями фрагментов кода, имеющими отношение к графике. Более подробную информацию о функции `ggsave()` вы найдете в документации.

Изменение размера рисунка

При обработке рисунков средствами R Markdown наибольшие трудности связаны с выбором их размеров. Для управления размерами рисунков предназначены пять основных опций: `fig.width`, `fig.height`, `fig.asp`, `out.width` и `out.height`. Вышеупомянутые трудности обусловлены необходимостью определения двух категорий размеров (размеров рисунка, созданного в R, и размеров рисунка, вставляемого в выходной документ) и

существованием нескольких способов указания размеров (для этого можно выбрать любые два из следующих трех параметров: высота, ширина и соотношение сторон).

В своей практике мы обходимся всего лишь тремя из перечисленных выше опций.

- Мы считаем, что графики выглядят эстетически более привлекательными, если они имеют одинаковую ширину. Поэтому по умолчанию мы используем опции `fig.width = 6` (6 дюймов) и `fig.asp = 0.618` (золотое сечение). Затем мы лишь корректируем для отдельных фрагментов опцию `fig.asp`.
- Для управления выходными размерами мы используем опцию `out.width`, значение которой устанавливаем в виде процентного отношения к ширине строки. В качестве значений по умолчанию мы используем `out.width = "70%"` и `fig.align = "center"`. Это обеспечивает достаточно большое пространство для расположения графика и умеренный размер полей.
- Если требуется разместить в одной строке несколько графиков, мы задаем для опции `out.width` значение `50%` в случае двух графиков, `33%` — в случае трех или `25%` — в случае четырех, устанавливая при этом опцию `fig.align = "default"`. В зависимости от того, что мы пытаемся проиллюстрировать (например, представить данные или продемонстрировать различия между графиками), мы подстраиваем опцию `fig.width`, о чем будет сказано далее.

Если вы обнаруживаете, что для чтения подписей на графике вам приходится прищуривать глаза, настройте опцию `fig.width`. Если значение этой опции превышает размеры рисунка, сформированного в готовом документе, то размер текста окажется слишком маленьким; в противном случае размер текста будет слишком большим. Чтобы правильно подобрать подходящее соотношение между значением опции `fig.width` и результирующей шириной графика в готовом документе, часто требуется немного поэкспериментировать. Этот принцип иллюстрируют следующие графики, для которых значения опции `fig.width` составляют соответственно `4`, `6` и `8` (рис. 22.26–22.28).

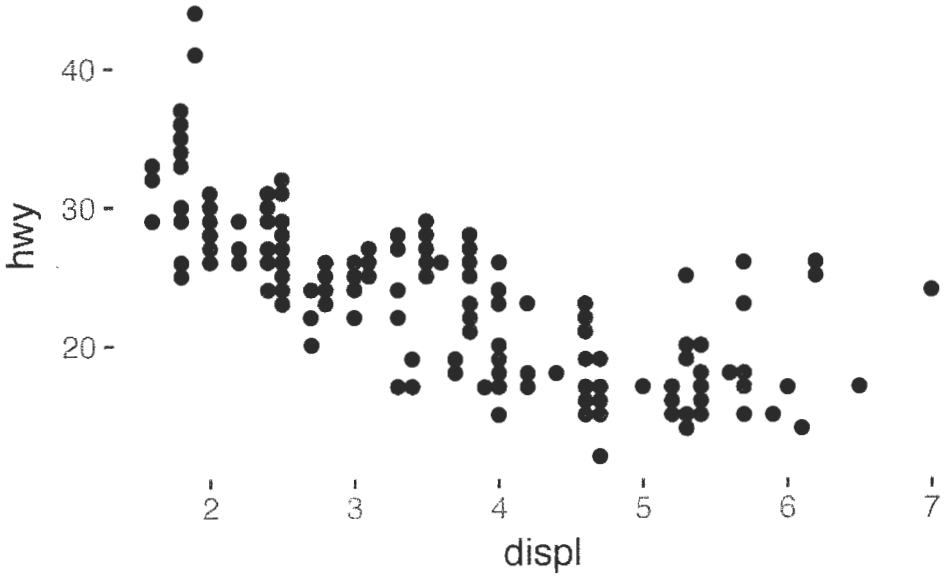


Рис. 22.26

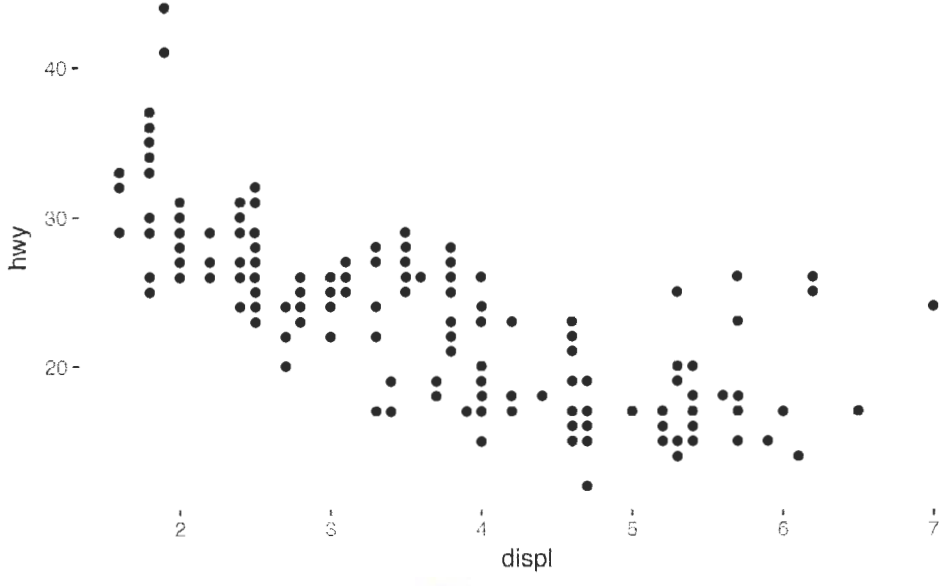


Рис. 22.27

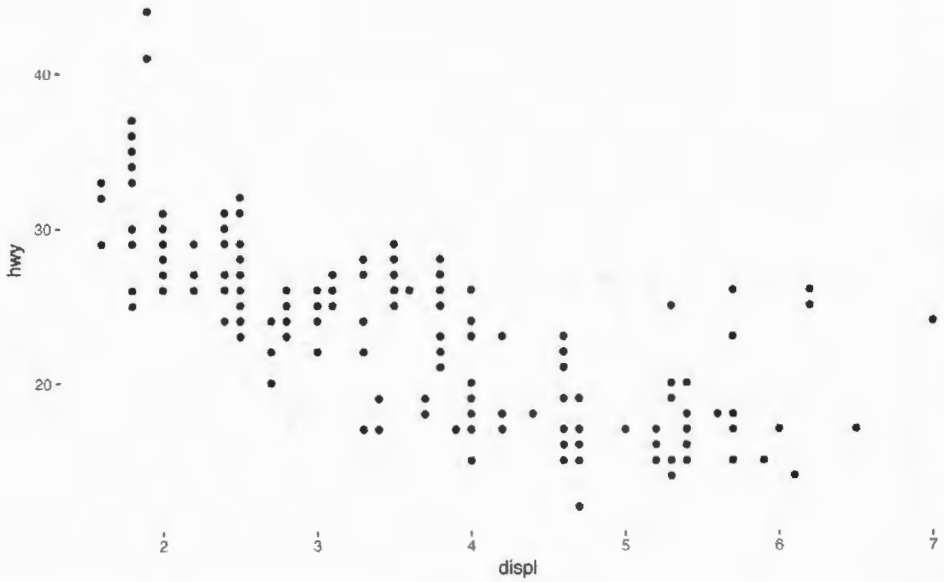


Рис. 22.28

Чтобы обеспечить согласование размера шрифтов по всем рисункам, следует при каждой установке опции `out.width` одновременно корректировать опцию `fig.width` для поддержания одного и того же соотношения с установленным вами по умолчанию значением опции `out.width`. Например, если значение по умолчанию для опции `fig.width` равно `6`, а значение `out.width` равно `0.7`, то в случае установки для опции `out.width` значения `"50%"` вы одновременно должны установить для опции `fig.width` значение `4.3` (`6 * 0.5 / 0.7`).

Другие важные опции

Если вам приходится перемежать код и текст, как это сделано нами в данной книге, рекомендуем установить опцию `fig.show = "hold"`, чтобы после кода отображались соответствующие графики. Полезным побочным эффектом является то, что это вынуждает вас разрывать большие блоки кода, включая в них свои пояснения.

Чтобы добавить подпись к графику, используйте опцию `fig.cap`. В R Markdown это приведет к изменению типа компоновки рисунка со встроенного на "плавающий".

В случае создания PDF-документа типом графики по умолчанию является PDF. Использование этого варианта целесообразно по той причине, что он

обеспечивает получение высококачественной векторной графики. Однако в случае графиков, насчитывающих тысячи точек, это замедляет работу и приводит к созданию файлов большого размера. В подобных ситуациях полезна опция `dev = "png"`, задающая использование изображений в формате PNG. При этом качество изображения немного снизится, зато оно станет намного более компактным.

Фрагментам кода, создающим рисунки, целесообразно присваивать имена, даже если вы, как правило, не помечаете другие фрагменты. Метка фрагмента используется при формировании имени графического файла, сохраняемого на диске, поэтому именование фрагментов значительно упрощает отбор графиков и их повторное использование при других обстоятельствах (например, если вы захотите быстро переслать одиночный график коллеге по электронной почте).

Дополнительная информация

Наилучшим источником дополнительной информации по данной теме является книга *ggplot2: Elegant graphics for data analysis*. Эти вопросы рассмотрены в ней на гораздо более глубоком уровне и с привлечением базовой теории. Кроме того, в ней содержится намного больше примеров, демонстрирующих сочетание разрозненных приемов работы с графикой для решения практических задач. К сожалению, онлайновый доступ к книге не бесплатный, но соответствующий исходный код можно найти по адресу `https://github.com/hadley/ggplot2-book`.

Другим замечательным источником является руководство по расширениям **ggplot2** (`http://www.ggplot2-exts.org/`). В нем вы найдете список множества других пакетов, расширяющих пакет **ggplot2** новыми геометриями и шкалами. С него отлично начинать, если вы пытаетесь сделать нечто, что вам представляется затруднительным в случае использования только возможностей **ggplot2**.

ГЛАВА 23

Выходные форматы R Markdown

Введение

Итак, вы уже знаете, как использовать R Markdown для создания HTML-документов. В этой главе представлен краткий обзор некоторых других выходных форматов, поддерживаемых R Markdown. Существуют два способа задания формата выходного документа.

1. Задание постоянного формата путем изменения YAML-заголовка.

```
title: "Viridis Demo"
output: html_document
```

2. Задание временного формата путем вызова функции `rmarkdown::render()` вручную.

```
rmarkdown::render(
  "diamond-sizes.Rmd",
  output_format = "word_document"
)
```

   Этот способ полезен, если вы хотите обеспечить создание выходных документов в различных форматах программным путем.

Щелчок на кнопке “Knit” в RStudio приводит к созданию выходного файла в формате, указанном первым в его поле `output`. Для раскрытия списка других доступных форматов следует щелкнуть на изображении стрелки рядом с названием кнопки Knit (рис. 23.1).

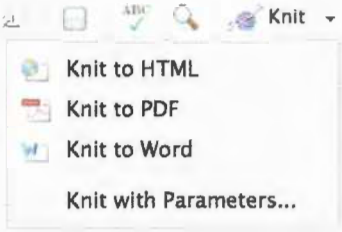


Рис. 23.1

Опции выходных форматов

С каждым выходным форматом ассоциирована функция R. Для вызова функций можно использовать синтаксис `foo` или `pkg::foo`. Если название пакета (`pkg`) опущено, то по умолчанию предполагается, что это пакет **rmarkdown**. Важно знать имя функции, обеспечивающей вывод, поскольку это позволяет получить соответствующую справку. Например, чтобы узнать, какие параметры можно устанавливать с помощью функции `html_document`, следует выполнить команду `?rmarkdown:html_document()`.

Чтобы переопределить значения параметров, заданные по умолчанию, необходимо использовать расширенное поле `output`. Например, если вы хотите сформировать HTML-документ с плавающим оглавлением, используйте следующий код.

```
output:
  html_document:
    toc: true
    toc_float: true
```

Кроме того, можно сформировать сразу нескольких выходных файлов, предоставив список соответствующих форматов.

```
output:
  html_document:
    toc: true
    toc_float: true
  pdf_document: default
```

Обратите внимание на специальный синтаксис, используемый для указания того, что вы не хотите переопределять значения опций, заданные по умолчанию.

Документы

В предыдущей главе мы имели дело в основном с выходными файлами в формате HTML. Однако кроме используемой по умолчанию опции `html_document` существует целый ряд других опций, позволяющих создавать выходные документы в различных базовых форматах.

- Опция `pdf_document` создает PDF-документ с помощью LaTeX — системы компьютерной верстки с открытым исходным кодом, которую вам понадобится установить. RStudio напомнит вам о необходимости установки, если ранее это не было сделано).
- Опция `word_document` предназначена для создания документов Microsoft Word (*.docx*).
- Опция `odt_document` предназначена для создания документов в формате OpenDocument Text (*.odt*).
- Опция `rtf_document` предназначена для создания документов в формате Rich Text (*.rtf*).
- Опция `md_document` предназначена для создания документов Markdown. Как правило, эта опция сама по себе не очень полезна, но она может пригодиться, если, например, в вашей корпоративной системе управления содержимым (CMS) или лабораторной wiki-системе используется язык Markdown.
- Опция `github_document` — это модифицированная версия опции `md_document`, позволяющая создавать документы для GitHub.

Вспомните о том, что в случае генерации документов, предназначенных для руководства, вы можете отключить заданное по умолчанию отображение кода, установив глобальные опции фрагмента `setup`:

```
knitr::opts_chunk$set(echo = FALSE)
```

В случае опции `html_documents` другая возможность заключается в том, чтобы сделать фрагменты кода скрытыми по умолчанию, но видимыми после щелчка на них.

```
output:
  html_document:
    code_folding: hide
```

Блокноты

Блокнот (опция `html_notebook`) — это видоизмененный вариант документа, создаваемого с помощью опции `html_document`. Оба формата очень схожи, но предназначены для разных целей. Если документы типа `html_document` предназначены для руководителей, то блокноты предназначены для ваших коллег, работающих в области науки о данных. Различие в назначении документов диктует различия в выходных HTML-документах. HTML-документы обоих типов содержат полностью сформированный вывод, но блокноты дополнительно содержат полный исходный код. Это означает, что файл *.nb.html*, сгенерированный блокнотом, можно использовать двумя способами.

- Вы можете просматривать его в браузере и видеть визуализированный вывод. В отличие от файла, созданного с помощью опции `html_document`, эта визуализация всегда включает встроенный вариант сгенерировавшего ее кода.
- Такой файл можно редактировать в RStudio. Когда вы открываете *.nb.html*-файл, RStudio автоматически воссоздает *.Rmd*-файл, который его сгенерировал. В будущем вы также сможете включать в этот файл вспомогательные файлы (например, *.csv*-файлы данных), которые будут автоматически извлекаться, если в этом возникнет необходимость.

Отправка *.nb.html*-файлов по электронной почте — простой способ поделиться результатами своего анализа с коллегами. Однако все значительно усложняется, как только они захотят внести в него свои изменения. Когда наступит такой момент, самое время познакомиться с тем, как работают сайты Git и GitHub. Поначалу это может доставить массу головной боли, однако получаемые при этом преимущества огромны. Как уже отмечалось ранее, рассмотрение работы с Git и GitHub выходит за рамки книги, но можем дать один совет, который пригодится вам, если вы уже работаете с этими сайтами: используйте выходные форматы `html_notebook` и `github_document`.

```
output:
  html_notebook: default
  github_document: default
```

Документ `html_notebook` предоставляет возможность локального предварительного просмотра и файл, которым можно поделиться по электронной почте. Документ `github_document` создает MD-файл минимального размера,

который можно зарегистрировать на Git. В этом случае вы сможете легко контролировать, как результаты вашего анализа (а не только код) изменяются со временем, а GitHub обеспечивает их онлайн-визуализацию.

Презентации

R Markdown также можно использовать для создания презентаций. В этом случае вы немного теряете в возможностях визуального контроля по сравнению с такими инструментами, как Keynote или PowerPoint, но автоматическая вставка результатов работы вашего R-кода в презентацию может сэкономить вам массу времени. Презентации работают путем разбиения содержимого на слайды, причем началом нового слайда служит каждый заголовок первого (`#`) или второго (`##`) уровня. Кроме того, можно использовать горизонтальную линейку (`***`) для создания нового слайда без заголовка.

R Markdown поставляется с тремя встроенными форматами презентаций.

`ioslides_presentation`

HTML-презентация с использованием формата *ioslides*;

`slidy_presentation`

HTML-презентация с использованием формата *W3C Slidy*;

`beamer_presentation`

PDF-презентация с использованием формата *LaTeX Beamer*.

Два других популярных формата предоставляются следующими пакетами.

`revealjs::revealjs_presentation`

HTML-презентация с помощью библиотеки *reveal.js* (требуется пакет **revealjs**).

`rmdshower`

Предоставляет оболочку движка презентаций *shower*.

Демонстрационные панели

Демонстрационные панели (dashboards) — удобный способ быстрого визуального обмена большими объемами информации. Пакет **flexdashboard** позволяет очень легко создавать демонстрационные панели, используя R Markdown и следующий набор правил, определяющих влияние заголовков на компоновку панели:

- каждый заголовок 1-го уровня (#) начинает новую страницу;
- каждый заголовок 2-го уровня (##) начинает новую колонку;
- каждый заголовок 3-го уровня (###) начинает новую строку.

Вот пример демонстрационной панели, созданной с помощью показанного ниже кода.

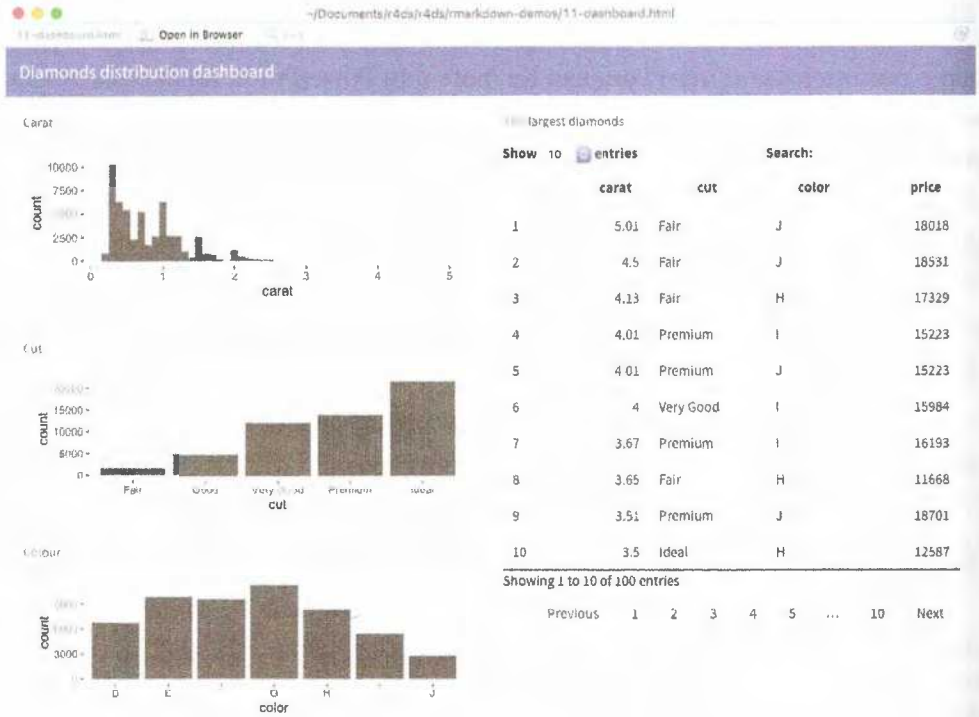


Рис. 23.2

```
---
title: "Diamonds distribution dashboard"
output: flexdashboard::flex_dashboard
---

```{r setup, include = FALSE} library(ggplot2) library(dplyr)
knitr::opts_chunk$set(fig.width = 5, fig.asp = 1/3)
```

Column 1

Carat
```

```
```{r}
ggplot(diamonds, aes(carat)) + geom_histogram(binwidth = 0.1)
```

Cut

```{r}
ggplot(diamonds, aes(cut)) + geom_bar()
```

Color

```{r}
ggplot(diamonds, aes(color)) + geom_bar()
```

Column 2

The largest diamonds

```{r}
diamonds %>%
  arrange(desc(carat)) %>%
  head(100) %>%
  select(carat, cut, color, price) %>%
  DT::datatable()
```
```

Кроме того, пакет **flexdashboard** предоставляет простые инструменты для создания врезок, таблиц, гистограммных столбцов и индикаторов. Для получения более подробной информации о пакете посетите сайт http://rmarkdown.rstudio.com/flexdashboard/.

Интерактивность

Любой HTML-формат (документ, блокнот, презентация или демонстрационная панель) может содержать интерактивные компоненты.

Виджеты

HTML — интерактивный формат, и в вашем распоряжении имеются виджеты — функции R, предназначенные для создания интерактивных элементов HTML-визуализации. В качестве примера рассмотрим карту, приведенную на рис. 23.3. При просмотре этой карты в Интернете вы можете перетаскивать

ее, увеличивать или уменьшать масштаб и т.п. Разумеется, подобное невозможно сделать в книге, поэтому пакет **rmarkdown** автоматически вставляет статическое изображение.

```
library(leaflet)
leaflet() %>%
  setView(174.764, -36.877, zoom = 16) %>%
  addTiles() %>%
  addMarkers(174.764, -36.877, popup = "Maungawhau")
```

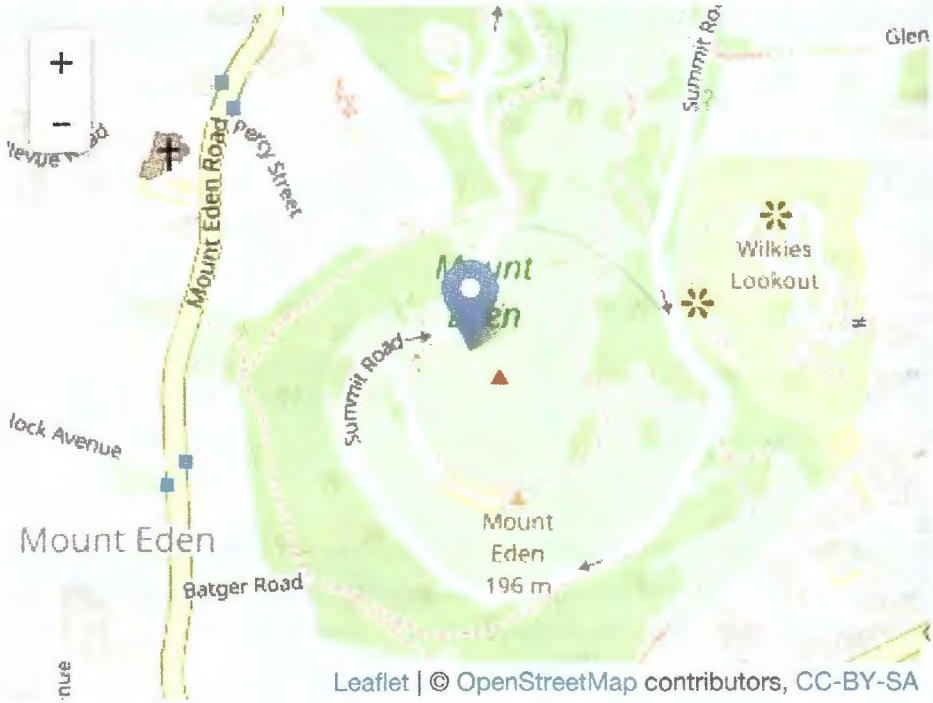


Рис. 23.3

Виджеты замечательны тем, что их использование не требует знания HTML или JavaScript. Все детали инкапсулированы в пакете, так что вам не о чем беспокоиться.

Виджеты предоставляются многими пакетами, включая следующие:

- **dygraphs** — интерактивная визуализация временных рядов;
- **DT** — интерактивные таблицы;
- **rthreejs** — интерактивные 3D-графики;
- **DiagrammeR** — диаграммы (включая блок-схемы и графы).

Чтобы узнать больше о виджетах и ознакомиться с более полным перечнем предоставляющих их пакетов, посетите сайт `http://www.htmlwidgets.org/`.

Пакет Shiny

Виджеты обеспечивают интерактивные возможности на клиентской стороне — вся интерактивность реализуется в браузере, независимо от R. С другой стороны, это и хорошо, поскольку вы можете распространять HTML-файл без подключения его к R. Однако это существенно ограничивает ваши возможности в отношении воздействия на элементы, реализованные с использованием HTML и JavaScript. Альтернативный подход заключается в использовании **Shiny** — пакета, который позволяет создавать интерактивные элементы, используя язык R, а не JavaScript.

Чтобы вызвать код **Shiny** из документа R Markdown, добавьте в заголовок строку `runtime: shiny`.

```
title: "Shiny Web App"
output: html_document
runtime: shiny
```

После этого вы сможете добавлять в документ интерактивные элементы с помощью функций семейства “input”.

```
library(shiny)

textInput("name", "What is your name?")
numericInput("age", "How old are you?", NA, min = 0, max = 150)
```

Теперь вы можете ссылаться на введенные значения с помощью имен `input$name` и `input$age`, и код, который их использует, будет автоматически заново исполняться при каждом их изменении (рис. 23.4).

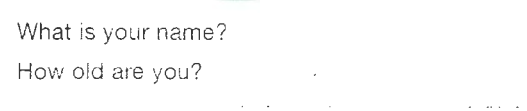


Рис. 23.4

Мы не можем продемонстрировать вам, как работает реальное приложение Shiny, поскольку все взаимодействие происходит на серверной стороне. Отсюда следует, что вы можете создать интерактивное приложение, даже не зная языка JavaScript, но это также означает, что для его выполнения вам

понадобится сервер. Здесь возникает проблема логистики: приложению Shiny, чтобы выполняться в онлайн-режиме, нужен сервер Shiny. Если вы запустите приложение Shiny на своем компьютере, оно автоматически установит сервер Shiny, но если вы захотите опубликовать такого рода интерактивное приложение в сети, вам понадобится общедоступный сервер. В том-то и состоит существенный компромисс Shiny: вы можете делать в документе Shiny то, что делаете в R, но для этого требуется, чтобы кто-то другой выполнял R.

Подробнее о пакете **Shiny** можно узнать на сайте `http://shiny.rstudio.com/`.

Веб-сайты

Выполнив ряд дополнительных действий, вы сможете использовать R Markdown для создания полноценного сайта.

- Поместите свои *.Rmd*-файлы в один каталог. Домашней страницей станет файл *index.Rmd*.
- Добавьте YAML-файл с именем *_site.yml*, обеспечивающий навигацию по сайту. Ниже показан пример такого файла.

```
name: "my-website"
navbar:
  title: "My Website"
  left:
    - text: "Home"
      href: index.html
    - text: "Viridis Colors"
      href: 1-example.html
    - text: "Terrain Colors"
      href: 3-inline.html
```

Выполните функцию `rmarkdown::render_site()` и создайте каталог *_site*, в котором будут храниться HTML-файлы, готовые для развертывания в виде автономного статического веб-сайта или, если вы используете проект RStudio, для выполнения в каталоге вашего веб-сайта.

Более подробно об этом можно прочитать по адресу `http://bit.ly/RMarkdownWebsites`.

Другие форматы

Существует ряд пакетов, которые предоставляют дополнительные выходные форматы.

- Пакет **bookdown** (`https://github.com/rstudio/bookdown`) упрощает написание книг, подобных этой. Для получения дополнительной информации прочитайте книгу *Authoring Books with R Markdown* (Yihui Xie; `https://bookdown.org/yihui/bookdown/`), которая, разумеется, написана с использованием пакета **bookdown**. Посетите сайт `http://www.bookdown.org` для ознакомления с другими книгами, подготовленными членами R-сообщества с помощью этого пакета.
- Пакет **prettydoc** (`https://github.com/yixuan/prettydoc/`) предоставляет облегченные версии форматов документов, предлагающие темы с привлекательным внешним видом.
- В пакете **rticles** (`https://github.com/rstudio/rticles`) собраны форматы, обеспечивающие соответствие форматам ряда научных журналов.

Для ознакомления с более широким списком доступных форматов посетите сайт `http://rmarkdown.rstudio.com/formats.html`. Кроме того, можно создавать собственные форматы, следуя инструкциям, представленным по адресу `http://bit.ly/CreatingNewFormats`.

Дополнительная информация

Чтобы более подробно ознакомиться с использованием описанных форматов для организации эффективного обмена информацией, рекомендуем обратиться к следующим источникам.

- Книга *Presentation Patterns* (Neal Ford, Matthew McCollough, Nathaniel Schutta) поможет вам усовершенствовать свое мастерство в области подготовки презентаций. В этой книге представлен ряд эффективных шаблонов (как низко-, так и высокоуровневых), используя которые вы сможете улучшить качество своих презентаций.
- Если вам приходится выступать с академическими докладами, рекомендуем прочесть руководство *Leek group guide to giving talks* (`https://github.com/jtleek/talkguide`).

- Лично мы еще не воспользовались этим ресурсом, однако слышали много хороших отзывов об онлайн-курсе по ораторскому искусству, подготовленном Мэттом МакГэррити (https://www.coursera.org/learn/public-speaking).
- Если вам приходится посвящать много времени подготовке демонстрационных панелей, обязательно прочтите книгу Стивена Фью *Information Dashboard Design: The Effective Visual Communication of Data* (https://www.amazon.com/Information-Dashboard-Design-Effective-Communication/dp/0596100167). С ее помощью вы сможете создавать демонстрационные панели, которые будут не только внешне привлекательными, но и действительно полезными.
- Вам будет проще доносить свои идеи другим людям, если вы приобретете знания в области графического дизайна. Неплохой стартовой площадкой для этого может послужить книга Робина Уильямса *The Non-Designer's Design Book* (https://ptgmedia.pearsoncmg.com/images/9780133966152/samplepages/9780133966152.pdf).

ГЛАВА 24

Организация работы с R Markdown

Ранее мы обсудили ключевые принципы интерактивной работы с кодом в *консоли*, а затем рассмотрели, как работать с кодом, находящимся в окне *редактора скриптов*. R Markdown объединяет консоль с редактором скриптов, размывая границы между интерактивным исследованием и долгосрочной фиксацией кода. Вы можете быстро манипулировать кодом в пределах фрагмента, внося в него правки и повторно выполняя с помощью комбинации клавиш <Ctrl+Shift+Enter>. Добившись результатов, которые вас удовлетворяют, вы переходите к работе с новым фрагментом кода.

Важное значение языка разметки R Markdown объясняется также тем, что он тесно интегрирует описательный текст и код. Это позволяет создавать отличные рабочие блокноты для записей о ходе исследования — инструмент, позволяющий сопровождать разработку кода записью своих мыслей и идей. Такой блокнот имеет много общего с классическим лабораторным журналом, используемым в физических науках. В частности, R Markdown обеспечивает следующее.

- Позволяет записывать свои действия и сопровождать их соответствующими пояснениями. Какой бы замечательной памятью вы ни обладали, в самый неподходящий момент вдруг обнаружится, что вы о чем-то забыли. Чтобы этого не произошло, обязательно записывайте важные детали!
- Стимулирует строгость мышления. Ваш анализ усилится, если вы будете записывать свои мысли по ходу его проведения и впоследствии обдумывать их. Это также сэкономит вам время, если вы решите поделиться результатами анализа с другими людьми.
- Облегчает понимание результатов вашей работы другими людьми. Лишь в редких случаях анализ выполняют в одиночку; чаще всего вы

будете работать в составе команды. Лабораторный журнал позволит вам информировать своих коллег не только о том, что вы сделали, но и с какой целью это совершалось.

Многие советы, относящиеся к ведению лабораторных журналов, в равной степени применимы к блокноту R Markdown. Исходя из собственного опыта и рекомендаций Колина Пуррингтона по ведению лабораторных журналов (http://colinpurrington.com/tips/lab-notebooks), можем посоветовать вам следующее.

- Следите за тем, чтобы каждый блокнот имел описательное название и информативное имя файла, а первый абзац содержал краткое описание целей анализа.
- Используйте поле даты в заголовке YAML для фиксации даты начала работы с блокнотом:

  ```
  date: 2016-08-23
  ```

  При указании даты придерживайтесь формата ISO8601 ГГГГ-ММ-ДД, чтобы избежать неоднозначности. Используйте этот формат даже в том случае, если обычно вы записываете даты по-другому!
- Если вы потратили много времени на анализ какой-либо идеи, но в конечном счете зашли в тупик, не удаляйте результат! Кратко опишите, почему не сработала идея, и сохраните ее в блокноте. Это не даст вам повторно очутиться в том же тупике, если в будущем вам придется вернуться к анализу.
- Как правило, подготовку исходных данных лучше выполнять вне R. Но если вам необходим небольшой набор данных, создавайте его с помощью функции `tibble::tribble()`.
- Если обнаруживается, что файл данных содержит ошибку, никогда не исправляйте ее непосредственно в файле. Вместо этого напишите код, корректирующий ошибочное значение. Объясните, почему вы внесли данную коррекцию.
- Прежде чем завершить свой рабочий день, убедитесь в том, что вы можете сформировать блокнот с помощью функции `knit()`. (Если вы используете кеширование, обязательно очистите кеш.) Это позволит вам устранить любые возникшие проблемы, пока все, что связано с кодом, еще свежо в вашей памяти.

- Если вы хотите обеспечить воспроизводимость кода в долгосрочной перспективе (т.е. чтобы вы могли гарантированно выполнить его в следующем месяце или году), вам потребуется организовать контроль версий пакетов, которые задействуются вашим кодом. Строгий подход заключается в использовании пакета **packrat**, сохраняющего пакеты в каталоге вашего проекта, или пакета **checkpoint**, который переустановит пакеты, доступные на момент указанной даты. Быстрое, хотя и грубое, решение этой задачи заключается в создании фрагмента кода, запускающего функцию `sessionInfo()`. Это не позволит вам легко воссоздать свои пакеты в их нынешнем состоянии, но зато вы будете знать, что они собой представляли.
- На протяжении всей своей карьеры вы создадите огромное множество блокнотов анализа. Как их следует организовать, чтобы в будущем можно было легко обратиться к каждому из них? Мы рекомендуем сохранять блокноты в отдельных проектах, придерживаясь какой-либо удобной для вас системы присвоения им имен.

Предметный указатель

A

ASCII 192

C

ColorBrewer 553
CRAN 28
CSL 532

E

EDA 421
EST 326

G

GMT 328

I

IANA 326

L

lv-диаграмма 152

R

R Markdown 515
  блокнот 570
  веб-сайты 576
  встроенный код 527
  выходной формат 567, 577
  демонстрационная панель 571
  документы 569
  интерактивность 573
  кеширование 525
  презентация 571
  форматирование текста 519
  фрагменты кода 521
RStudio 28, 350
  диагностика 129
  проект 169
  сохранение рабочего пространства 166
R-скрипт 165

S

SQL 251

T

tibble-фрейм 90, 175, 389
  извлечение поднабора 179
  транспонированный 177

U

UTC 327
UTF-8 192

Y

YAML 517, 529

А

Аддитивная модель 458
Аккуратные данные 22, 133
Аннотация 538
Антисоединение 254

Б

Библиографическая ссылка 531
Библиотека
  lubridate 460
  modelr 488
  reveal.js 571
  tidyverse 28, 30, 41, 89
Бинарная функция 419
Блокнот 570
Блочное тестирование 346
Большие данные 25

В

Вариация 133, 161
Вектор 367
  атомарный 370
    отсутствующие значения 372
  атрибуты 385
  именование 377
  логический 370
  приведение типов 373
  расширенный 387
  рекурсивный 380
  символьный 371
  создание 404
  числовой 370
Векторизация функций 376
Виджет 573
Визуализация 22, 39, 89, 436
Визуальный атрибут 47
Вложенные данные 491
  отмена вложения 494
Внешнее соединение 244
Внешний ключ 237
Внутреннее соединение 243
Выбор столбцов 99
Выброс 140
Выходной формат 567

Г

Геометрический объект 57
Геометрия 57
Гистограмма 136
Градация 47
Грамматика графики 41, 79
  глаголы 91
График 533
  аннотации 538
  легенда 546, 549
  подписи 534
  создание 43
  сохранение 560
  тема 558
  шкала 545, 550
Группирование переменных 121
Групповой итог 107

Д

Декартово произведение 247
Демонстрационная панель 571
Деревья 458
Диаграмма
  Венна 245
  отношений 236
  размаха 148
  столбчатая 134
Длительность 321

Е

Естественное соединение 248

З

Зацикливание вектора 375
Змеиный регистр 84
Значение 133
Золотое сечение 562

И

Императивное программирование 391
Импорт данных 21
Интерактивное приглашение 31
Интервал 324

К

Канал 108, 333
Каталог 168
Категориальная переменная 291
Качество модели 497
Квантиль 117
Кеширование 525
Кластер значений 138
Ключ 237
  легенды 546
  неуникальный 246
  суррогатный 239
Ковариация 133, 145, 161
Композиция функций 336

Л

Левое соединение 244
Легенда 549
Лексическая область видимости 365
Ленивые вычисления 361
Линейная модель 435
  обобщенная 457
  робастная 458
  со штрафами 458
Логический вектор 370
Логическое выделение поднаборов 116
Локаль 190, 263

М

Масштаб изображения 556
Медиана 116, 148
Медианное абсолютное отклонение 117
Межквартильный размах 117, 148
Мера
  положения 116
  порядковая 118
  разброса 117
  ранжирования 117
Метод касательных (Ньютона-Рафсона) 434
Моделирование 425, 459
  формулы 441
Модель 161
Модулярная арифметика 104
Мутирующее соединение 240

Н

Наблюдение 133
Наука о данных 21
Неаккуратные данные 230
Несинтаксическое имя 177
Нотация Уилкинсона–Роджерса 442

О

Обобщенная функция 386
Обратный апостроф 177, 351
Общая линейная модель 435
Объединение
  операций 108
  столбцов 220
  строк 261
Односторонняя формула 406
Оператор
  [180
  [[179
  %>% 337
  $ 179
  арифметический 103
  логический 93, 352
  сравнения 93, 105
Остатки 161, 439
Отсутствующее значение 95, 110, 143, 222, 372, 457

П

Пакет 29
  bookdown 577
  broom 487, 497, 511
  checkpoint 581
  data.table 25
  DBI 205
  DiagrammeR 574
  dplyr 89, 91, 128, 234
  DT 574
  Dygraphs 574
  feather 205
  flexdashboard 571
  forcats 292
  gapminder 489
  ggplot2 41, 163, 294, 533
    системы координат 76
    темы 559
    шпаргалка 59
  ggrepel 534, 540
  haven 205
  hexbin 157
  hms 195
  jsonlite 206

knitr 523
Lahman 114
lubridate 308
lvplot 152
magrittr 333, 339
modelr 426
nycflights13 89, 234, 308
packrat 581
prettydoc 577
purrr 375, 392, 403
RColorBrewer 553
RcppRoll 105
readr 183, 197
readxl 205
rio 206
rmarkdown 516, 568
rthreejs 574
rticles 577
seriation 155
Shiny 575
stringi 290
stringr 259, 261, 274, 290
tibble 175
tidyr 207, 226
viridis 534, 553
webreadr 184
xml2 206
загрузка 32
Панель 54
Параллельные вычисления 531
Парсер 189
Первичный ключ 237
Переменная 133
вычисляемая 481
категориальная 134, 146, 291, 442, 445
непрерывная 135, 146, 445, 449
Перестановка строк 97
Период 322
Печать 178
Подготовка данных 22, 173
Подтверждающий анализ 27
Полигон частот 439
Полное соединение 244
Полусоединение 252
Правое соединение 244
Предварительный анализ 39, 131
Презентация 571
Преобразование 22, 453
Приведение типов 373
Проверка типов 375
Проект 169
Путь 168

Р

Рабочий каталог 167
Разведочный анализ 27
Разгруппирование переменных 122
Разделение столбцов 218
Размер рисунка 561
Ранжирование 106
Рассредоточение столбцов 215
Расширенный вектор 387
Регулярное выражение 265
группы 273
жадный поиск 272
замена совпадений 283
извлечение совпадений 279
квантификатор 270
поиск совпадений 276
символьный класс 269
чередование 269
якорь 267
Редактор скриптов 127, 579
средства диагностики 129
Реляционные данные 233

С

Сведение столбцов 213
Семейство моделей 425
Сетка поиска 431
Символьный вектор 188, 249, 371
зацикливание 262
Синглтон 418

Синтаксический анализ
векторы 188
дата и время 194
строки 192
файлы 197
числа 190
Система координат 76
Соединение 242
внешнее 244
внутреннее 243
естественное 248
мутирующее 240
по эквивалентности 244
фильтрующее 252
Список 380, 501
визуализация 381
Справка 54
Среднее 116
Среднеквадратическое отклонение 117, 429
Статистика графика 65
Статистическое преобразование 64
Столбец-список 493, 500
создание 502
упрощение 508
Строка 259
длина 261
извлечение подстрок 262
объединение 261
Счетчик 111, 119

Т

Табличные данные 133
Тема 558
Тестовый набор 423
Тип переменной 90
Точка данных 133
Тренировочный набор 423

У

Условное выполнение 351
множественные условия 353

Ф

Фактор 193, 291, 388
порядок следования 296
уровень 302
Фильтрующее соединение 252
Фрейм данных 43
data.frame 178
tibble 175
отображение 90
преобразование 180
Функциональное программирование 392
Функция 343
accumulate() 419
add_predictions() 437
add_residuals() 439
aes() 45, 49
all() 352
anti_join() 256
any() 352
apropos() 289
arrange() 91, 97
as.data.frame() 180
as_date() 313
as_datetime() 313
assign() 337
as_tibble() 176
attr() 385
attributes() 385
boundary() 289
c() 261, 370, 377
ceiling_date() 318
charToRaw() 192
coll() 288
collapse() 262
complete() 223
contains() 100
coord_cartesian() 140, 557
coord_flip() 77, 151
coord_polar() 77
coord_quickmap() 77

count() 120, 134, 238, 294
cume_dist() 106
cummean() 105
cumsum() 105
cut_number() 157
cut_width() 135, 157
data.frame() 501
data_grid() 437, 446, 467
date_names() 196
date_names_langs() 196
dense_rank() 106
desc() 97, 115
detect() 418
detect_index() 418
dir() 290
discard() 417
dput() 33
ends_with() 100
enframe() 507
every() 418
everything() 101
extract() 282
facet_grid() 55
facet_wrap() 55
fct_collapse() 304
fct_infreq() 300
fct_inorder() 293
fct_lump() 304
fct_recode() 302, 304
fct_relevel() 298
fct_reorder() 296
fct_reorder2() 300
fct_rev() 301
fill() 224
filter() 91, 92, 93, 124
first() 118
fixed() 287
floor_date() 318
format() 527
fwf_positions() 184
fwf_widths() 184
gather() 213
gather_predictions() 446
geom_abline() 427
geom_bin2d() 157
geom_boxplot() 149, 151
geom_count() 153
geom_freqpoly() 146
geom_freqpoly() 137
geom_hex() 157
geom_histogram() 137
geom_hline() 544
geom_jitter() 498
geom_label() 538
geom_lv() 152
geom_point() 44, 155
geom_rect() 544
geom_segment() 544
geom_smooth 58
geom_smooth() 475
geom_text() 538
geom_tile() 154
geom_title() 450
geom_vline() 544
getS3method() 386
getwd() 167
ggplot() 44
ggsave() 560
glance() 497
glob2rx() 290
group_by() 91
guess_encoding() 193
guess_parser() 197
guide_legend() 549
guides() 549
head_while() 418
hour() 314
I() 501
identical() 353
ifelse() 143
invoke_map() 415
IQR() 117

is.na() 96, 144
kable() 524
keep() 417
labs() 534
lag() 105
lapply() 407
last() 118
lead() 105
left_join() 241
length() 368
levels() 293
library() 30, 42
list(...) 361
list() 380
lm() 435
locale() 196
log2() 104
mad() 117
make_date() 310
make_datetime() 310
map() 404, 412, 493
map2() 412
map_dbl() 404
matches() 100
max() 117
mday() 314
mean() 116, 276
median() 116
merge 251
methods() 386
min() 117
min_rank() 106
minute() 314
model_matrix() 441, 453
month() 314
mutate() 91, 102, 124, 143, 241, 277, 493, 504
n_distinct() 119
near() 93
nest() 503
nobs() 457
now() 309
ns() 454
nth() 118
num_range() 100
object_size() 335, 371
OlsonNames() 327
optim() 434
parse_character() 189, 192
parse_date() 190, 194
parse_datetime() 194
parse_double() 189
parse_factor() 190, 193, 293
parse_guess() 197
parse_integer() 189
parse_logical() 189
parse_number() 189, 191
parse_time() 190, 195
percent_rank() 106
pmap() 413
poly() 454
possibly() 411
print() 178
problems() 189, 199
pwalk() 417
quantile() 117, 505
quietly() 411
quote() 537
read_csv() 183
read_csv2() 183
read_delim() 184
read_file() 203
read_fwf() 184
read_lines() 203
read_log() 184
read_rds() 204
readRDS() 204
readr_example() 199
read_table() 184
read_tsv() 184
reduce() 418
regex() 286

rename() 100
render() 530, 567
reorder() 150
rep() 376
return() 362
rlm() 480
round_date() 318
row_number() 106
safely() 409
sapply() 407
saveRDS() 204
scale_color_gradient() 553
scale_color_manual() 553
scale_color_viridis() 553
scale_fill_gradient() 553
sd() 117
second() 314
select() 91, 99
separate() 218
separate_rows() 505
seq() 85
seq_along() 393
seq_range() 449
set_names() 377
setwd() 167
some() 418
spread() 216
starts_with() 100
stat_count() 65
stat_summary() 68
stop() 359
stop_for_problems() 201
stopifnot() 360
str() 380
str_c() 261, 360
str_count() 277
str_detect() 276
str_extract() 279, 281
str_extract_all() 280
str_length() 261
str_locate() 286
str_match() 281
str_replace() 228, 283
str_replace_all() 283
str_replace_na() 262
str_sort() 264
str_split() 283
str_sub() 262, 286
str_subset() 277
str_to_upper() 263
str_view() 265
str_view_all() 265, 278
str_wrap() 543
sum() 120, 276
summarize() 91, 107, 505, 541
Sys.timezone() 327
tail_while() 418
theme() 549
tibble() 176, 501, 542
tidyverse_update() 33
today() 309
transmute() 103
transpose() 410
tribble() 177, 415, 501
tryCatch() 338
type_convert() 202
typeof() 368
ungroup() 122
unite() 220
unnest() 495, 504, 510
update() 319
vapply() 408
vector() 393
View() 241
walk() 416
wday() 315
write_csv() 203
write_excel_csv() 203
writeLines() 260
write_rds() 204
write_tsv() 203
yday() 314

year() 314
ymd() 309
аргументы 345, 356, 358
бинарная 419
векторизация 103, 504
возвращаемое значение 362
итоговая 116
логарифмическая 104
обобщенная 386
оконная 125
окружение 364
создание 345
с побочным эффектом 364
Функция-предикат 417

Ц

Цикл for 392, 395
Цитата 531

Ч

Часовой пояс 326
Числовой вектор 370

Ш

Шкала 545
ColorBrewer 553
замена 550

Э

Эвристика 197
Экранирование символов 260, 266
Эстетика 46

Я

Якорь 267
Ящик с усами 148, 157

ВВЕДЕНИЕ В МАШИННОЕ ОБУЧЕНИЕ С ПОМОЩЬЮ PYTHON

РУКОВОДСТВО ДЛЯ СПЕЦИАЛИСТОВ ПО РАБОТЕ С ДАННЫМИ

Андреас Мюллер
Сара Гвидо

www.williamspublishing.com

Эта полноцветная книга — отличный источник информации для каждого, кто собирается использовать машинное обучение на практике. Ныне машинное обучение стало неотъемлемой частью различных коммерческих и исследовательских проектов, и не следует думать, что эта область — прерогатива исключительно крупных компаний с мощными командами аналитиков. Эта книга научит вас практическим способам построения систем МО, даже если вы еще новичок в этой области. В ней подробно объясняются все этапы, необходимые для создания успешного проекта машинного обучения, с использованием языка Python и библиотек scikit-learn, NumPy и matplotlib. Авторы сосредоточили свое внимание исключительно на практических аспектах применения алгоритмов машинного обучения, оставив за рамками книги их математическое обоснование. Данная книга адресована специалистам, решающим реальные задачи, а поскольку область применения методов МО практически безгранична, прочитав эту книгу, вы сможете собственными силами построить действующую систему машинного обучения в любой научной или коммерческой сфере.

ISBN 978-5-9908910-8-1

в продаже

O'REILLY®

Язык R в задачах науки о данных

Овладейте искусством превращения необработанных первичных данных в плодотворные догадки, гипотезы и новые знания с помощью языка R. Эта книга задумана как введение в вычислительную среду R, среду разработки RStudio и библиотеку tidyverse — коллекцию пакетов, совместное использование которых обеспечивает быстроту и легкость анализа данных. Книга ориентирована на читателей, не имеющих предварительного опыта программирования, и предназначена для того, чтобы помочь им в как можно более короткие сроки начать решать задачи науки о данных.

Авторы книги познакомят вас со всеми стадиями процесса анализа данных, включая импорт, предварительную подготовку, разведочный анализ и моделирование данных, а также демонстрацию результатов. Прочитав книгу, вы получите цельное представление как о процессе анализа данных в целом, так и об основных инструментах, используемых при его проведении.

Основные темы книги:

- **Предварительная обработка данных** — преобразование наборов данных к виду, удобному для анализа.
- **Программирование** — освоение мощных инструментов R, упрощающих анализ данных и делающих его более понятным.
- **Разведочный анализ** — исследование данных, выдвижение и быстрая проверка гипотез.
- **Моделирование** — предоставление сжатых сводных данных, отражающих выявленные тенденции в наборе данных.
- **Обмен информацией** — изучение языка R Markdown, обеспечивающего интеграцию описательного текста, кода и результатов анализа.

Категория: анализ данных/статистические расчеты
Уровень: средний

"Хэдли Уикем — живая легенда в области науки о данных, изобретатель совершенно нового подхода к анализу данных, идея которого никому до него даже не приходила в голову. Данная книга, написанная в соавторстве с Гарреттом Гроулмундом, кодифицирует этот новейший подход и несомненно станет настольным пособием для целого поколения аналитиков данных".

Роджер Д. Пенг,
профессор биостатистики, Школа общественного здравоохранения при университете Джона Хопкинса

Хэдли Уикем — старший научный сотрудник компании RStudio и член организации R Foundation. К числу его разработок относятся пакеты для науки о данных (библиотека tidyverse: ggplot2, dplyr, tidyr, purrr, readr и др.) и инструментальные средства разработки пакетов (roxygen2, testthat, devtools). Более подробную информацию о нем можно найти на сайте `http://hadley.nz`.

Гарретт Гроулмунд — специалист в области статистики, преподаватель и главный инструктор компании RStudio. Он автор известного пакета lubridate, а также книги *Hands-On Programming with R*.

ISBN 978-5-9909446-8-8

www.williamspublishing.com