O'REILLY®

R for Data Science

IMPORT, TIDY, TRANSFORM, VISUALIZE, AND MODEL DATA

Hadley Wickham &
Garrett Grolemund

O'REILLY®

R for Data Science

Learn how to use R to turn raw data into insight, knowledge, and understanding. This book introduces you to R, RStudio, and the tidyverse, a collection of R packages designed to work together to make data science fast, fluent, and fun. Suitable for readers with no previous programming experience, *R for Data Science* is designed to get you doing data science as quickly as possible.

Authors Hadley Wickham and Garrett Grolemund guide you through the steps of importing, wrangling, exploring, and modeling your data and communicating the results. You'll get a complete, big-picture understanding of the data science cycle, along with basic tools you need to manage the details.

You'll learn how to:

- **Wrangle**—transform your datasets into a form convenient for analysis
- **Program**—learn powerful R tools for solving data problems with greater clarity and ease
- **Explore**—examine your data, generate hypotheses, and quickly test them
- **Model**—provide a low-dimensional summary that captures true "signals" in your dataset
- **Communicate**—learn R Markdown for integrating prose, code, and results

"Hadley Wickham is a legend in the data science field for having invented a completely new way of doing data analysis that no one had thought of before. This new book with Garrett Grolemund codifies this novel approach and will serve as the Bible for a generation of data analysts."

—**Roger D. Peng**
Professor of Biostatistics,
Johns Hopkins Bloomberg
School of Public Health

Hadley Wickham is Chief Scientist at RStudio and a member of the R Foundation. He builds tools (both computational and cognitive) that make data science easier, faster, and more fun. Learn more on his website, http://hadley.nz.

Garrett Grolemund is a statistician, teacher, and Master Instructor at RStudio. He is the author of *Hands-On Programming with R* (O'Reilly). Many of Garrett's instructional videos are available on oreilly.com/safari.

DATA ANALYSIS/STATISTICAL SOFTWARE

US $39.99 **CAN $45.99**

ISBN: 978-1-491-91039-9

Twitter: @oreillymedia
facebook.com/oreilly

R for Data Science

Import, Tidy, Transform, Visualize, and Model Data

Hadley Wickham and Garrett Grolemund

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

R for Data Science
by Hadley Wickham and Garrett Grolemund

Printed in Canada.

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

Editors: Marie Beaugureau and Mike Loukides
Production Editor: Nicholas Adams
Copyeditor: Kim Cofer
Proofreader: Charles Roumeliotis

Indexer: Wendy Catalano
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

December 2016: First Edition

Revision History for the First Edition
2016-12-06: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491910399* for release details.

978-1-491-91039-9

[TI]

Table of Contents

Part III. Program

Part IV. Model

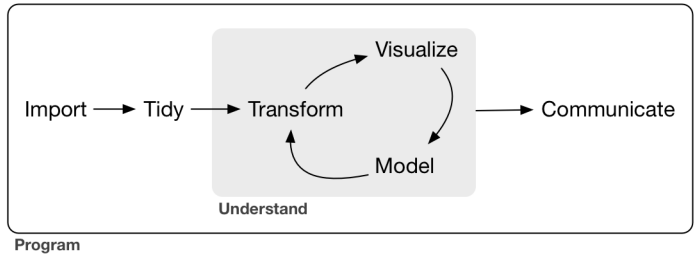
Part V. Communicate

Preface

Data science is an exciting discipline that allows you to turn raw data into understanding, insight, and knowledge. The goal of *R for Data Science* is to help you learn the most important tools in R that will allow you to do data science. After reading this book, you'll have the tools to tackle a wide variety of data science challenges, using the best parts of R.

What You Will Learn

Data science is a huge field, and there's no way you can master it by reading a single book. The goal of this book is to give you a solid foundation in the most important tools. Our model of the tools needed in a typical data science project looks something like this:



First you must *import* your data into R. This typically means that you take data stored in a file, database, or web API, and load it into a data frame in R. If you can't get your data into R, you can't do data science on it!

Once you've imported your data, it is a good idea to *tidy* it. Tidying your data means storing it in a consistent form that matches the semantics of the dataset with the way it is stored. In brief, when your data is tidy, each column is a variable, and each row is an observation. Tidy data is important because the consistent structure lets you focus your struggle on questions about the data, not fighting to get the data into the right form for different functions.

Once you have tidy data, a common first step is to *transform* it. Transformation includes narrowing in on observations of interest (like all people in one city, or all data from the last year), creating new variables that are functions of existing variables (like computing velocity from speed and time), and calculating a set of summary statistics (like counts or means). Together, tidying and transforming are called *wrangling*, because getting your data in a form that's natural to work with often feels like a fight!

Once you have tidy data with the variables you need, there are two main engines of knowledge generation: visualization and modeling. These have complementary strengths and weaknesses so any real analysis will iterate between them many times.

Visualization is a fundamentally human activity. A good visualization will show you things that you did not expect, or raise new questions about the data. A good visualization might also hint that you're asking the wrong question, or you need to collect different data. Visualizations can surprise you, but don't scale particularly well because they require a human to interpret them.

Models are complementary tools to visualization. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally mathematical or computational tool, so they generally scale well. Even when they don't, it's usually cheaper to buy more computers than it is to buy more brains! But every model makes assumptions, and by its very nature a model cannot question its own assumptions. That means a model cannot fundamentally surprise you.

The last step of data science is *communication*, an absolutely critical part of any data analysis project. It doesn't matter how well your models and visualization have led you to understand the data unless you can also communicate your results to others.

Surrounding all these tools is *programming*. Programming is a cross-cutting tool that you use in every part of the project. You don't need to be an expert programmer to be a data scientist, but learning more about programming pays off because becoming a better programmer allows you to automate common tasks, and solve new problems with greater ease.

You'll use these tools in every data science project, but for most projects they're not enough. There's a rough 80-20 rule at play; you can tackle about 80% of every project using the tools that you'll learn in this book, but you'll need other tools to tackle the remaining 20%. Throughout this book we'll point you to resources where you can learn more.

How This Book Is Organized

The previous description of the tools of data science is organized roughly according to the order in which you use them in an analysis (although of course you'll iterate through them multiple times). In our experience, however, this is not the best way to learn them:

- Starting with data ingest and tidying is suboptimal because 80% of the time it's routine and boring, and the other 20% of the time it's weird and frustrating. That's a bad place to start learning a new subject! Instead, we'll start with visualization and transformation of data that's already been imported and tidied. That way, when you ingest and tidy your own data, your motivation will stay high because you know the pain is worth it.
- Some topics are best explained with other tools. For example, we believe that it's easier to understand how models work if you already know about visualization, tidy data, and programming.
- Programming tools are not necessarily interesting in their own right, but do allow you to tackle considerably more challenging problems. We'll give you a selection of programming tools in the middle of the book, and then you'll see they can combine with the data science tools to tackle interesting modeling problems.

Within each chapter, we try to stick to a similar pattern: start with some motivating examples so you can see the bigger picture, and then dive into the details. Each section of the book is paired with exercises to help you practice what you've learned. While it's tempt-

ing to skip the exercises, there's no better way to learn than practicing on real problems.

What You Won't Learn

There are some important topics that this book doesn't cover. We believe it's important to stay ruthlessly focused on the essentials so you can get up and running as quickly as possible. That means this book can't cover every important topic.

Big Data

This book proudly focuses on small, in-memory datasets. This is the right place to start because you can't tackle big data unless you have experience with small data. The tools you learn in this book will easily handle hundreds of megabytes of data, and with a little care you can typically use them to work with 1–2 Gb of data. If you're routinely working with larger data (10–100 Gb, say), you should learn more about data.table. This book doesn't teach data.table because it has a very concise interface, which makes it harder to learn since it offers fewer linguistic cues. But if you're working with large data, the performance payoff is worth the extra effort required to learn it.

If your data is bigger than this, carefully consider if your big data problem might actually be a small data problem in disguise. While the complete data might be big, often the data needed to answer a specific question is small. You might be able to find a subset, subsample, or summary that fits in memory and still allows you to answer the question that you're interested in. The challenge here is finding the right small data, which often requires a lot of iteration.

Another possibility is that your big data problem is actually a large number of small data problems. Each individual problem might fit in memory, but you have millions of them. For example, you might want to fit a model to each person in your dataset. That would be trivial if you had just 10 or 100 people, but instead you have a million. Fortunately each problem is independent of the others (a setup that is sometimes called embarrassingly parallel), so you just need a system (like Hadoop or Spark) that allows you to send different datasets to different computers for processing. Once you've figured out how to answer the question for a single subset using the tools

described in this book, you learn new tools like sparklyr, rhipe, and ddr to solve it for the full dataset.

Python, Julia, and Friends

In this book, you won't learn anything about Python, Julia, or any other programming language useful for data science. This isn't because we think these tools are bad. They're not! And in practice, most data science teams use a mix of languages, often at least R and Python.

However, we strongly believe that it's best to master one tool at a time. You will get better faster if you dive deep, rather than spreading yourself thinly over many topics. This doesn't mean you should only know one thing, just that you'll generally learn faster if you stick to one thing at a time. You should strive to learn new things throughout your career, but make sure your understanding is solid before you move on to the next interesting thing.

We think R is a great place to start your data science journey because it is an environment designed from the ground up to support data science. R is not just a programming language, but it is also an interactive environment for doing data science. To support interaction, R is a much more flexible language than many of its peers. This flexibility comes with its downsides, but the big upside is how easy it is to evolve tailored grammars for specific parts of the data science process. These mini languages help you think about problems as a data scientist, while supporting fluent interaction between your brain and the computer.

Nonrectangular Data

This book focuses exclusively on rectangular data: collections of values that are each associated with a variable and an observation. There are lots of datasets that do not naturally fit in this paradigm: including images, sounds, trees, and text. But rectangular data frames are extremely common in science and industry, and we believe that they're a great place to start your data science journey.

Hypothesis Confirmation

It's possible to divide data analysis into two camps: hypothesis generation and hypothesis confirmation (sometimes called confirma-

tory analysis). The focus of this book is unabashedly on hypothesis generation, or data exploration. Here you'll look deeply at the data and, in combination with your subject knowledge, generate many interesting hypotheses to help explain why the data behaves the way it does. You evaluate the hypotheses informally, using your skepticism to challenge the data in multiple ways.

The complement of hypothesis generation is hypothesis confirmation. Hypothesis confirmation is hard for two reasons:

- You need a precise mathematical model in order to generate falsifiable predictions. This often requires considerable statistical sophistication.
- You can only use an observation once to confirm a hypothesis. As soon as you use it more than once you're back to doing exploratory analysis. This means to do hypothesis confirmation you need to "preregister" (write out in advance) your analysis plan, and not deviate from it even when you have seen the data. We'll talk a little about some strategies you can use to make this easier in Part IV.

It's common to think about modeling as a tool for hypothesis confirmation, and visualization as a tool for hypothesis generation. But that's a false dichotomy: models are often used for exploration, and with a little care you can use visualization for confirmation. The key difference is how often you look at each observation: if you look only once, it's confirmation; if you look more than once, it's exploration.

Prerequisites

We've made a few assumptions about what you already know in order to get the most out of this book. You should be generally numerically literate, and it's helpful if you have some programming experience already. If you've never programmed before, you might find *Hands-On Programming with R* by Garrett to be a useful adjunct to this book.

There are four things you need to run the code in this book: R, RStudio, a collection of R packages called the *tidyverse*, and a handful of other packages. Packages are the fundamental units of repro-

ducible R code. They include reusable functions, the documentation that describes how to use them, and sample data.

R

To download R, go to CRAN, the *comprehensive R archive network*. CRAN is composed of a set of mirror servers distributed around the world and is used to distribute R and R packages. Don't try and pick a mirror that's close to you: instead use the cloud mirror, *https://cloud.r-project.org*, which automatically figures it out for you.

A new major version of R comes out once a year, and there are 2–3 minor releases each year. It's a good idea to update regularly. Upgrading can be a bit of a hassle, especially for major versions, which require you to reinstall all your packages, but putting it off only makes it worse.

RStudio

RStudio is an integrated development environment, or IDE, for R programming. Download and install it from *http://www.rstudio.com/download*. RStudio is updated a couple of times a year. When a new version is available, RStudio will let you know. It's a good idea to upgrade regularly so you can take advantage of the latest and greatest features. For this book, make sure you have RStudio 1.0.0.

When you start RStudio, you'll see two key regions in the interface:

For now, all you need to know is that you type R code in the console pane, and press Enter to run it. You'll learn more as we go along!

The Tidyverse

You'll also need to install some R packages. An R *package* is a collection of functions, data, and documentation that extends the capabilities of base R. Using packages is key to the successful use of R. The majority of the packages that you will learn in this book are part of the so-called tidyverse. The packages in the tidyverse share a common philosophy of data and R programming, and are designed to work together naturally.

You can install the complete tidyverse with a single line of code:

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

On your own computer, type that line of code in the console, and then press Enter to run it. R will download the packages from CRAN and install them onto your computer. If you have problems installing, make sure that you are connected to the internet, and that *https://cloud.r-project.org/* isn't blocked by your firewall or proxy.

You will not be able to use the functions, objects, and help files in a package until you load it with `library()`. Once you have installed a package, you can load it with the `library()` function:

```
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
```

This tells you that tidyverse is loading the **ggplot2**, **tibble**, **tidyr**, **readr**, **purrr**, and **dplyr** packages. These are considered to be the *core* of the tidyverse because you'll use them in almost every analysis.

Packages in the tidyverse change fairly frequently. You can see if updates are available, and optionally install them, by running `tidyverse_update()`.

Other Packages

There are many other excellent packages that are not part of the tidyverse, because they solve problems in a different domain, or are designed with a different set of underlying principles. This doesn't make them better or worse, just different. In other words, the complement to the tidyverse is not the messyverse, but many other universes of interrelated packages. As you tackle more data science projects with R, you'll learn new packages and new ways of thinking about data.

In this book we'll use three data packages from outside the tidyverse:

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

These packages provide data on airline flights, world development, and baseball that we'll use to illustrate key data science ideas.

Running R Code

The previous section showed you a couple of examples of running R code. Code in the book looks like this:

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

If you run the same code in your local console, it will look like this:

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

There are two main differences. In your console, you type after the `>`, called the *prompt*; we don't show the prompt in the book. In the book, output is commented out with `#>`; in your console it appears directly after your code. These two differences mean that if you're working with an electronic version of the book, you can easily copy code out of the book and into the console.

Throughout the book we use a consistent set of conventions to refer to code:

- Functions are in a code font and followed by parentheses, like `sum()` or `mean()`.
- Other R objects (like data or function arguments) are in a code font, without parentheses, like `flights` or `x`.

- If we want to make it clear what package an object comes from, we'll use the package name followed by two colons, like `dplyr::mutate()` or `nycflights13::flights`. This is also valid R code.

Getting Help and Learning More

This book is not an island; there is no single resource that will allow you to master R. As you start to apply the techniques described in this book to your own data you will soon find questions that I do not answer. This section describes a few tips on how to get help, and to help you keep learning.

If you get stuck, start with Google. Typically, adding "R" to a query is enough to restrict it to relevant results: if the search isn't useful, it often means that there aren't any R-specific results available. Google is particularly useful for error messages. If you get an error message and you have no idea what it means, try googling it! Chances are that someone else has been confused by it in the past, and there will be help somewhere on the web. (If the error message isn't in English, run `Sys.setenv(LANGUAGE = "en")` and re-run the code; you're more likely to find help for English error messages.)

If Google doesn't help, try stackoverflow. Start by spending a little time searching for an existing answer; including `[R]` restricts your search to questions and answers that use R. If you don't find anything useful, prepare a minimal reproducible example or **reprex**. A good reprex makes it easier for other people to help you, and often you'll figure out the problem yourself in the course of making it.

There are three things you need to include to make your example reproducible: required packages, data, and code:

- *Packages* should be loaded at the top of the script, so it's easy to see which ones the example needs. This is a good time to check that you're using the latest version of each package; it's possible you've discovered a bug that's been fixed since you installed the package. For packages in the tidyverse, the easiest way to check is to run `tidyverse_update()`.
- The easiest way to include *data* in a question is to use `dput()` to generate the R code to re-create it. For example, to re-create the `mtcars` dataset in R, I'd perform the following steps:

1. Run `dput(mtcars)` in R.
2. Copy the output.
3. In my reproducible script, type `mtcars <-` then paste.

   Try and find the smallest subset of your data that still reveals the problem.

- Spend a little bit of time ensuring that your *code* is easy for others to read:
  - Make sure you've used spaces and your variable names are concise, yet informative.
  - Use comments to indicate where your problem lies.
  - Do your best to remove everything that is not related to the problem.

  The shorter your code is, the easier it is to understand, and the easier it is to fix.

Finish by checking that you have actually made a reproducible example by starting a fresh R session and copying and pasting your script in.

You should also spend some time preparing yourself to solve problems before they occur. Investing a little time in learning R each day will pay off handsomely in the long run. One way is to follow what Hadley, Garrett, and everyone else at RStudio are doing on the RStudio blog. This is where we post announcements about new packages, new IDE features, and in-person courses. You might also want to follow Hadley (@hadleywickham) or Garrett (@statgarrett) on Twitter, or follow @rstudiotips to keep up with new features in the IDE.

To keep up with the R community more broadly, we recommend reading *http://www.r-bloggers.com:* it aggregates over 500 blogs about R from around the world. If you're an active Twitter user, follow the #rstats hashtag. Twitter is one of the key tools that Hadley uses to keep up with new developments in the community.

Acknowledgments

This book isn't just the product of Hadley and Garrett, but is the result of many conversations (in person and online) that we've had with the many people in the R community. There are a few people

we'd like to thank in particular, because they have spent many hours answering our dumb questions and helping us to better think about data science:

- Jenny Bryan and Lionel Henry for many helpful discussions around working with lists and list-columns.
- The three chapters on workflow were adapted (with permission) from "R basics, workspace and working directory, RStudio projects" by Jenny Bryan.
- Genevera Allen for discussions about models, modeling, the statistical learning perspective, and the difference between hypothesis generation and hypothesis confirmation.
- Yihui Xie for his work on the **bookdown** package, and for tirelessly responding to my feature requests.
- Bill Behrman for his thoughtful reading of the entire book, and for trying it out with his data science class at Stanford.
- The #rstats twitter community who reviewed all of the draft chapters and provided tons of useful feedback.
- Tal Galili for augmenting his **dendextend** package to support a section on clustering that did not make it into the final draft.

This book was written in the open, and many people contributed pull requests to fix minor problems. Special thanks goes to everyone who contributed via GitHub (listed in alphabetical order): adi pradhan, Ahmed ElGabbas, Ajay Deonarine, @Alex, Andrew Landgraf, @batpigandme, @behrman, Ben Marwick, Bill Behrman, Brandon Greenwell, Brett Klamer, Christian G. Warden, Christian Mongeau, Colin Gillespie, Cooper Morris, Curtis Alexander, Daniel Gromer, David Clark, Derwin McGeary, Devin Pastoor, Dylan Cashman, Earl Brown, Eric Watt, Etienne B. Racine, Flemming Villalona, Gregory Jefferis, @harrismcgehee, Hengni Cai, Ian Lyttle, Ian Sealy, Jakub Nowosad, Jennifer (Jenny) Bryan, @jennybc, Jeroen Janssens, Jim Hester, @jjchern, Joanne Jang, John Sears, Jon Calder, Jonathan Page, @jonathanflint, Julia Stewart Lowndes, Julian During, Justinas Petuchovas, Kara Woo, @kdpsingh, Kenny Darrell, Kirill Sevastyanenko, @koalabearski, @KyleHumphrey, Lawrence Wu, Matthew Sedaghatfar, Mine Cetinkaya-Rundel, @MJMarshall, Mustafa Ascha, @nate-d-olson, Nelson Areal, Nick Clark, @nickelas, @nwaff, @OaCantona, Patrick Kennedy, Peter Hurford, Rademeyer Vermaak, Radu Grosu, @rlzijdeman, Robert Schuessler, @robinlovelace,

@robinsones, S'busiso Mkhondwane, @seamus-mckinsey, @seanpwilliams, Shannon Ellis, @shoili, @sibusiso16, @spirgel, Steve Mortimer, @svenski, Terence Teo, Thomas Klebel, TJ Mahr, Tom Prior, Will Beasley, Yihui Xie.

Online Version

An online version of this book is available at *http://r4ds.had.co.nz*. It will continue to evolve in between reprints of the physical book. The source of the book is available at *https://github.com/hadley/r4ds*. The book is powered by *https://bookdown.org*, which makes it easy to turn R markdown files into HTML, PDF, and EPUB.

This book was built with:

```
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
```

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

Bold
: Indicates the names of R packages.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This element signifies a tip or suggestion.

Using Code Examples

Source code is available for download at *https://github.com/hadley/r4ds*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*R for Data Science* by Hadley Wickham and Garrett Grolemund (O'Reilly). Copyright 2017 Garrett Grolemund, Hadley Wickham, 978-1-491-91039-9."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Safari

Safari (formerly Safari Books Online) is a membership-based training and reference platform for enterprise, government, educators, and individuals.

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit *http://oreilly.com/safari*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/r-for-data-science*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

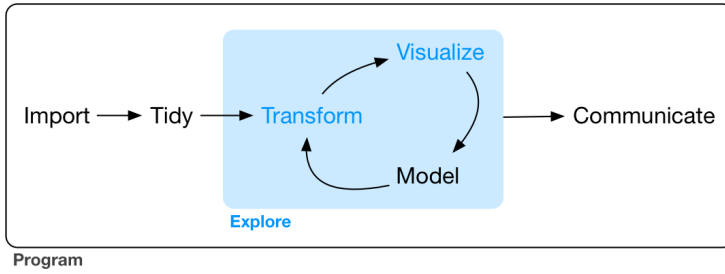
Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

PART I
Explore

The goal of the first part of this book is to get you up to speed with the basic tools of *data exploration* as quickly as possible. Data exploration is the art of looking at your data, rapidly generating hypotheses, quickly testing them, then repeating again and again and again. The goal of data exploration is to generate many promising leads that you can later explore in more depth.



In this part of the book you will learn some useful tools that have an immediate payoff:

- Visualization is a great place to start with R programming, because the payoff is so clear: you get to make elegant and informative plots that help you understand data. In Chapter 1 you'll

dive into visualization, learning the basic structure of a **ggplot2** plot, and powerful techniques for turning data into plots.

- Visualization alone is typically not enough, so in Chapter 3 you'll learn the key verbs that allow you to select important variables, filter out key observations, create new variables, and compute summaries.
- Finally, in Chapter 5, you'll combine visualization and transformation with your curiosity and skepticism to ask and answer interesting questions about data.

Modeling is an important part of the exploratory process, but you don't have the skills to effectively learn or apply it yet. We'll come back to it in Part IV, once you're better equipped with more data wrangling and programming tools.

Nestled among these three chapters that teach you the tools of exploration are three chapters that focus on your R workflow. In Chapter 2, Chapter 4, and Chapter 6 you'll learn good practices for writing and organizing your R code. These will set you up for success in the long run, as they'll give you the tools to stay organized when you tackle real projects.

CHAPTER 1

Data Visualization with ggplot2

Introduction

> The simple graph has brought more information to the data analyst's mind than any other device.
>
> —John Tukey

This chapter will teach you how to visualize your data using **ggplot2**. R has several systems for making graphs, but **ggplot2** is one of the most elegant and most versatile. **ggplot2** implements the *grammar of graphics*, a coherent system for describing and building graphs. With **ggplot2**, you can do more faster by learning one system and applying it in many places.

If you'd like to learn more about the theoretical underpinnings of **ggplot2** before you start, I'd recommend reading "A Layered Grammar of Graphics".

Prerequisites

This chapter focuses on **ggplot2**, one of the core members of the tidyverse. To access the datasets, help pages, and functions that we will use in this chapter, load the tidyverse by running this code:

```
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
```

That one line of code loads the core tidyverse, packages that you will use in almost every data analysis. It also tells you which functions from the tidyverse conflict with functions in base R (or from other packages you might have loaded).

If you run this code and get the error message "there is no package called 'tidyverse'," you'll need to first install it, then run `library()` once again:

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

You only need to install a package once, but you need to reload it every time you start a new session.

If we need to be explicit about where a function (or dataset) comes from, we'll use the special form `package::function()`. For example, `ggplot2::ggplot()` tells you explicitly that we're using the `ggplot()` function from the **ggplot2** package.

First Steps

Let's use our first graph to answer a question: do cars with big engines use more fuel than cars with small engines? You probably already have an answer, but try to make your answer precise. What does the relationship between engine size and fuel efficiency look like? Is it positive? Negative? Linear? Nonlinear?

The mpg Data Frame

You can test your answer with the `mpg` data frame found in **ggplot2** (aka `ggplot2::mpg`). A *data frame* is a rectangular collection of variables (in the columns) and observations (in the rows). `mpg` contains observations collected by the US Environment Protection Agency on 38 models of cars:

```
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>
```

Among the variables in `mpg` are:

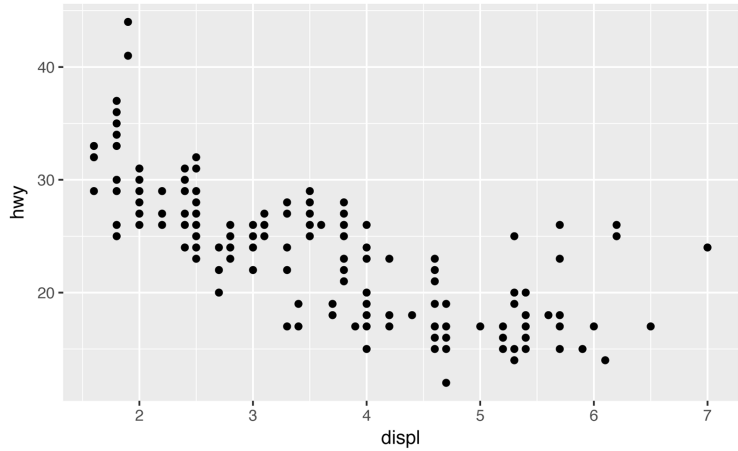
- `displ`, a car's engine size, in liters.
- `hwy`, a car's fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

To learn more about `mpg`, open its help page by running `?mpg`.

Creating a ggplot

To plot `mpg`, run this code to put `displ` on the x-axis and `hwy` on the y-axis:

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



The plot shows a negative relationship between engine size (`displ`) and fuel efficiency (`hwy`). In other words, cars with big engines use more fuel. Does this confirm or refute your hypothesis about fuel efficiency and engine size?

With **ggplot2**, you begin a plot with the function `ggplot()`. `ggplot()` creates a coordinate system that you can add layers to. The first argument of `ggplot()` is the dataset to use in the graph. So `ggplot(data = mpg)` creates an empty graph, but it's not very interesting so I'm not going to show it here.

You complete your graph by adding one or more layers to `ggplot()`. The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot. **ggplot2** comes with many geom functions that each add a different type of layer to a plot. You'll learn a whole bunch of them throughout this chapter.

Each geom function in **ggplot2** takes a `mapping` argument. This defines how variables in your dataset are mapped to visual properties. The `mapping` argument is always paired with `aes()`, and the `x` and `y` arguments of `aes()` specify which variables to map to the x- and y-axes. **ggplot2** looks for the mapped variable in the `data` argument, in this case, `mpg`.

A Graphing Template

Let's turn this code into a reusable template for making graphs with **ggplot2**. To make a graph, replace the bracketed sections in the following code with a dataset, a geom function, or a collection of mappings:

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

The rest of this chapter will show you how to complete and extend this template to make different types of graphs. We will begin with the `<MAPPINGS>` component.

Exercises

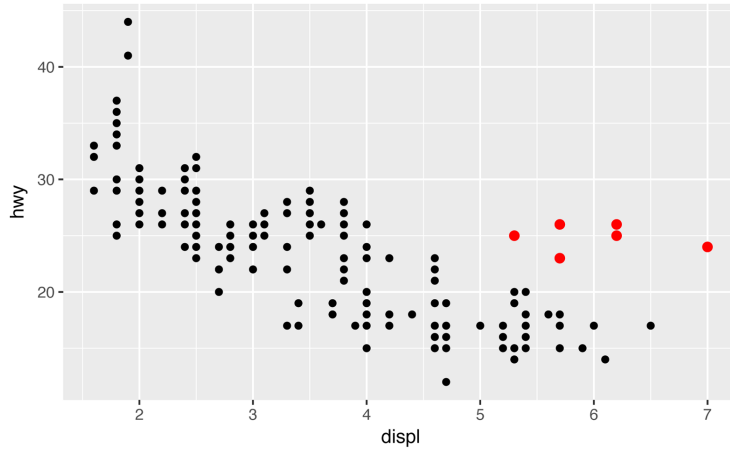
1. Run `ggplot(data = mpg)`. What do you see?
2. How many rows are in `mtcars`? How many columns?
3. What does the `drv` variable describe? Read the help for `?mpg` to find out.
4. Make a scatterplot of `hwy` versus `cyl`.

5. What happens if you make a scatterplot of `class` versus `drv`? Why is the plot not useful?

Aesthetic Mappings

> The greatest value of a picture is when it forces us to notice what we never expected to see.
>
> —John Tukey

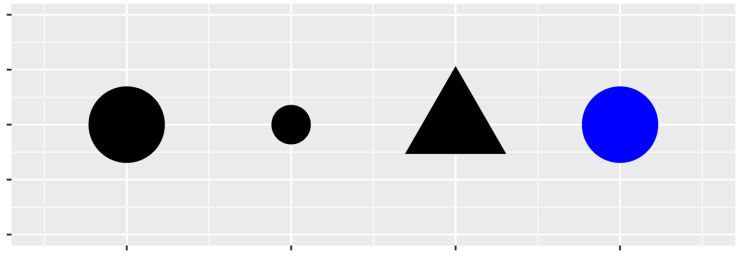
In the following plot, one group of points (highlighted in red) seems to fall outside of the linear trend. These cars have a higher mileage than you might expect. How can you explain these cars?



Let's hypothesize that the cars are hybrids. One way to test this hypothesis is to look at the `class` value for each car. The `class` variable of the `mpg` dataset classifies cars into groups such as compact, midsize, and SUV. If the outlying points are hybrids, they should be classified as compact cars or, perhaps, subcompact cars (keep in mind that this data was collected before hybrid trucks and SUVs became popular).

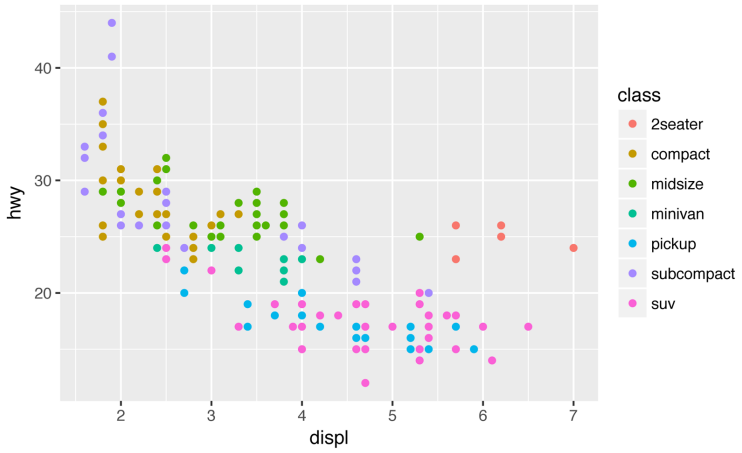
You can add a third variable, like `class`, to a two-dimensional scatterplot by mapping it to an *aesthetic*. An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point (like the one shown next) in different ways by changing the values of its aesthetic properties. Since we already use the word "value" to

describe data, let's use the word "level" to describe aesthetic properties. Here we change the levels of a point's size, shape, and color to make the point small, triangular, or blue:



You can convey information about your data by mapping the aesthetics in your plot to the variables in your dataset. For example, you can map the colors of your points to the `class` variable to reveal the class of each car:

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



(If you prefer British English, like Hadley, you can use `colour` instead of `color`.)

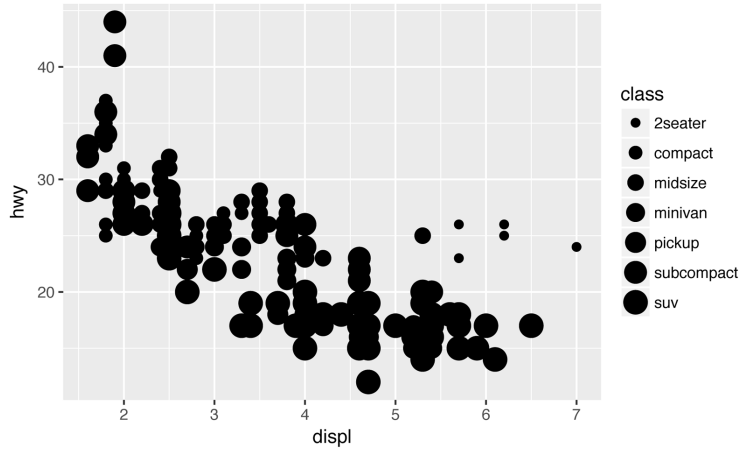
To map an aesthetic to a variable, associate the name of the aesthetic to the name of the variable inside `aes()`. **ggplot2** will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as *scaling*. **ggplot2** will

also add a legend that explains which levels correspond to which values.

The colors reveal that many of the unusual points are two-seater cars. These cars don't seem like hybrids, and are, in fact, sports cars! Sports cars have large engines like SUVs and pickup trucks, but small bodies like midsize and compact cars, which improves their gas mileage. In hindsight, these cars were unlikely to be hybrids since they have large engines.

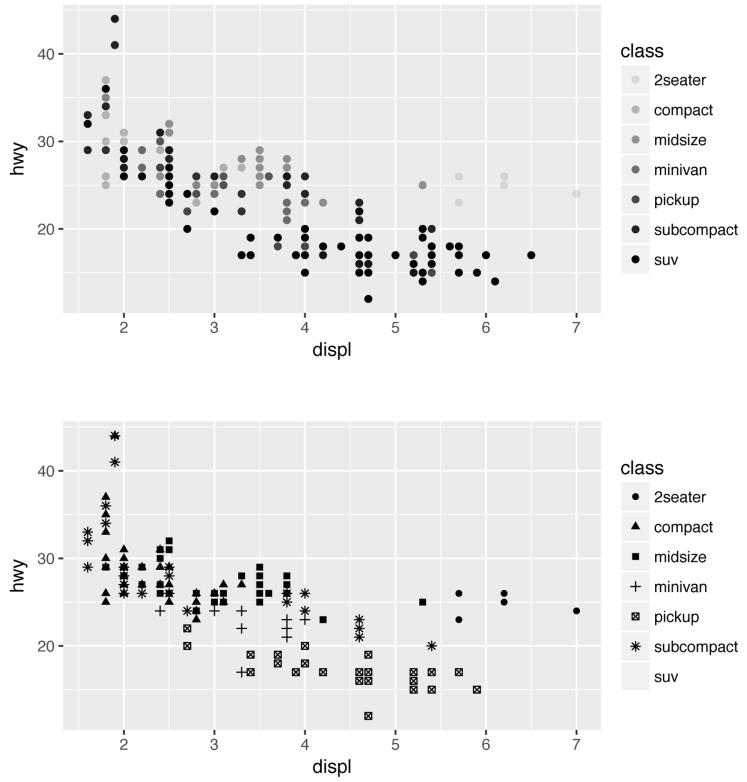
In the preceding example, we mapped `class` to the color aesthetic, but we could have mapped `class` to the size aesthetic in the same way. In this case, the exact size of each point would reveal its class affiliation. We get a *warning* here, because mapping an unordered variable (`class`) to an ordered aesthetic (`size`) is not a good idea:

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



Or we could have mapped `class` to the *alpha* aesthetic, which controls the transparency of the points, or the shape of the points:

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

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



What happened to the SUVs? **ggplot2** will only use six shapes at a time. By default, additional groups will go unplotted when you use this aesthetic.

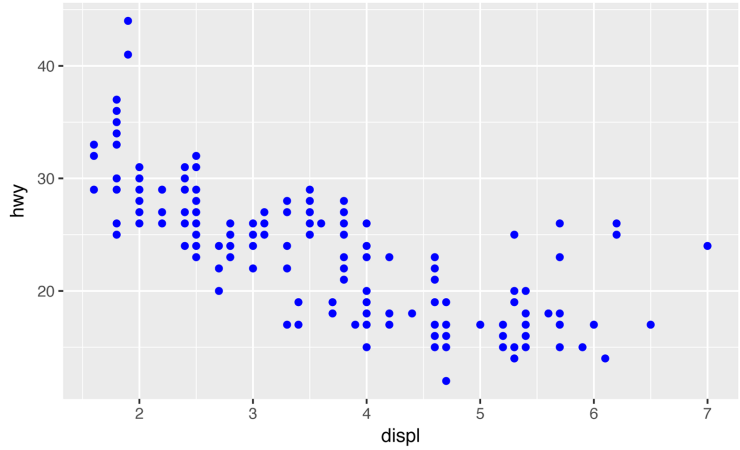
For each aesthetic you use, the `aes()` to associate the name of the aesthetic with a variable to display. The `aes()` function gathers together each of the aesthetic mappings used by a layer and passes them to the layer's mapping argument. The syntax highlights a useful insight about `x` and `y`: the x and y locations of a point are themselves aesthetics, visual properties that you can map to variables to display information about the data.

Once you map an aesthetic, **ggplot2** takes care of the rest. It selects a reasonable scale to use with the aesthetic, and it constructs a legend that explains the mapping between levels and values. For x and y aesthetics, **ggplot2** does not create a legend, but it creates an axis

line with tick marks and a label. The axis line acts as a legend; it explains the mapping between locations and values.

You can also *set* the aesthetic properties of your geom manually. For example, we can make all of the points in our plot blue:

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



Here, the color doesn't convey information about a variable, but only changes the appearance of the plot. To set an aesthetic manually, set the aesthetic by name as an argument of your geom function; i.e., it goes *outside* of `aes()`. You'll need to pick a value that makes sense for that aesthetic:

- The name of a color as a character string.
- The size of a point in mm.
- The shape of a point as a number, as shown in Figure 1-1. There are some seeming duplicates: for example, 0, 15, and 22 are all squares. The difference comes from the interaction of the `color` and `fill` aesthetics. The hollow shapes (0–14) have a border determined by `color`; the solid shapes (15–18) are filled with `color`; and the filled shapes (21–24) have a border of `color` and are filled with `fill`.

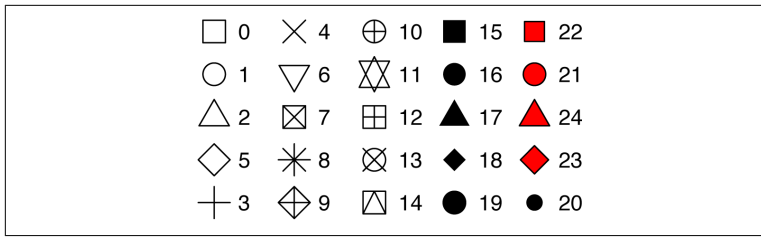
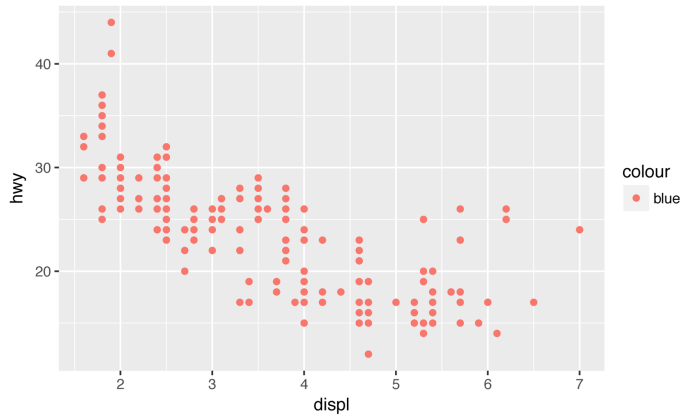


Figure 1-1. R has 25 built-in shapes that are identified by numbers

Exercises

1. What's gone wrong with this code? Why are the points not blue?

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



2. Which variables in `mpg` are categorical? Which variables are continuous? (Hint: type `?mpg` to read the documentation for the dataset.) How can you see this information when you run `mpg`?
3. Map a continuous variable to `color`, `size`, and `shape`. How do these aesthetics behave differently for categorical versus continuous variables?
4. What happens if you map the same variable to multiple aesthetics?
5. What does the `stroke` aesthetic do? What shapes does it work with? (Hint: use `?geom_point`.)

6. What happens if you map an aesthetic to something other than a variable name, like `aes(color = displ < 5)`?

Common Problems

As you start to run R code, you're likely to run into problems. Don't worry—it happens to everyone. I have been writing R code for years, and every day I still write code that doesn't work!

Start by carefully comparing the code that you're running to the code in the book. R is extremely picky, and a misplaced character can make all the difference. Make sure that every `(` is matched with a `)` and every `"` is paired with another `"`. Sometimes you'll run the code and nothing happens. Check the left-hand side of your console: if it's a `+`, it means that R doesn't think you've typed a complete expression and it's waiting for you to finish it. In this case, it's usually easy to start from scratch again by pressing Esc to abort processing the current command.

One common problem when creating **ggplot2** graphics is to put the `+` in the wrong place: it has to come at the end of the line, not the start. In other words, make sure you haven't accidentally written code like this:

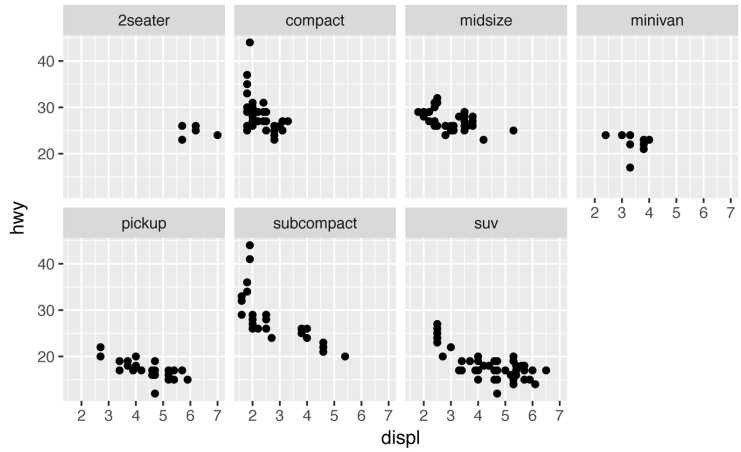
```
ggplot(data = mpg)
+ geom_point(mapping = aes(x = displ, y = hwy))
```

If you're still stuck, try the help. You can get help about any R function by running `?function_name` in the console, or selecting the function name and pressing F1 in RStudio. Don't worry if the help doesn't seem that helpful—instead skip down to the examples and look for code that matches what you're trying to do.

If that doesn't help, carefully read the error message. Sometimes the answer will be buried there! But when you're new to R, the answer might be in the error message but you don't yet know how to understand it. Another great tool is Google: trying googling the error message, as it's likely someone else has had the same problem, and has received help online.

Facets

One way to add additional variables is with aesthetics. Another way, particularly useful for categorical variables, is to split your plot into *facets*, subplots that each display one subset of the data.

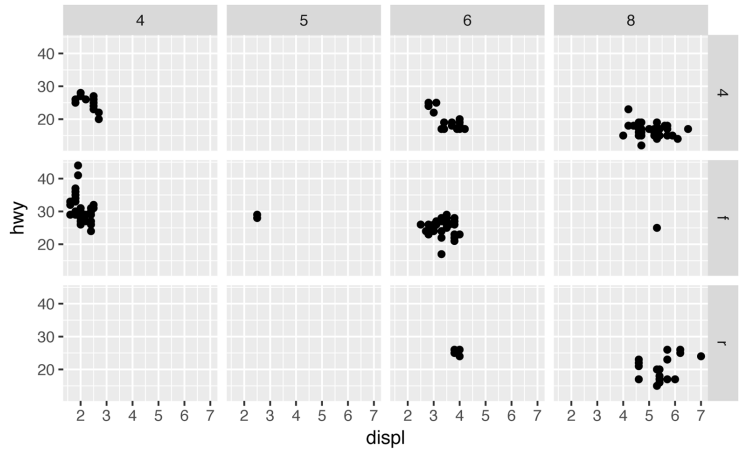


To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` should be a formula, which you create with ~ followed by a variable name (here "formula" is the name of a data structure in R, not a synonym for "equation"). The variable that you pass to `facet_wrap()` should be discrete:

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

To facet your plot on the combination of two variables, add `facet_grid()` to your plot call. The first argument of `facet_grid()` is also a formula. This time the formula should contain two variable names separated by a ~:

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



If you prefer to not facet in the rows or columns dimension, use a `.` instead of a variable name, e.g., `+ facet_grid(. ~ cyl)`.

Exercises

1. What happens if you facet on a continuous variable?
2. What do the empty cells in a plot with `facet_grid(drv ~ cyl)` mean? How do they relate to this plot?

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

3. What plots does the following code make? What does `.` do?

```
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. Take the first faceted plot in this section:

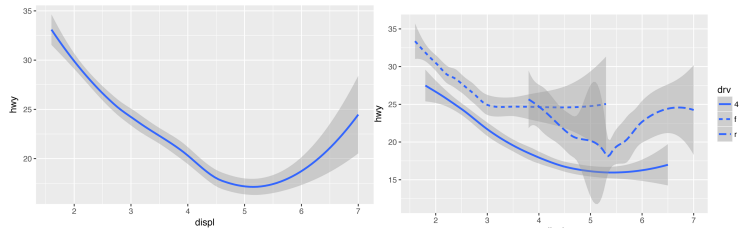
```
ggplot(data = mpg) +
  geom_point(mapping = aes(x = displ, y = hwy)) +
  facet_wrap(~ class, nrow = 2)
```

   What are the advantages to using faceting instead of the color aesthetic? What are the disadvantages? How might the balance change if you had a larger dataset?

5. Read `?facet_wrap`. What does `nrow` do? What does `ncol` do? What other options control the layout of the individual panels? Why doesn't `facet_grid()` have `nrow` and `ncol` variables?
6. When using `facet_grid()` you should usually put the variable with more unique levels in the columns. Why?

Geometric Objects

How are these two plots similar?



Both plots contain the same `x` variable and the same `y` variable, and both describe the same data. But the plots are not identical. Each plot uses a different visual object to represent the data. In **ggplot2** syntax, we say that they use different *geoms*.

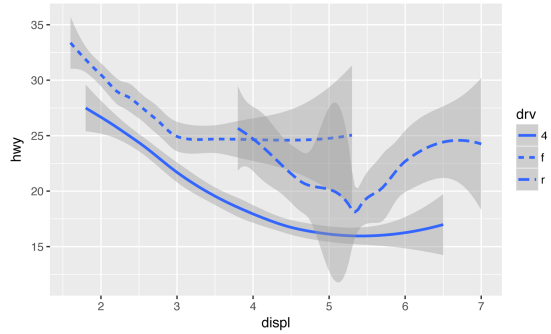
A *geom* is the geometrical object that a plot uses to represent data. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms, line charts use line geoms, boxplots use boxplot geoms, and so on. Scatterplots break the trend; they use the point geom. As we see in the preceding plots, you can use different geoms to plot the same data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

To change the geom in your plot, change the geom function that you add to `ggplot()`. For instance, to make the preceding plots, you can use this code:

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

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

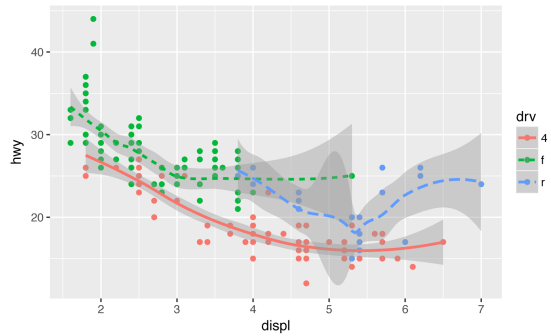
Every geom function in **ggplot2** takes a `mapping` argument. However, not every aesthetic works with every geom. You could set the shape of a point, but you couldn't set the "shape" of a line. On the other hand, you *could* set the linetype of a line. `geom_smooth()` will draw a different line, with a different linetype, for each unique value of the variable that you map to linetype:

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



Here `geom_smooth()` separates the cars into three lines based on their `drv` value, which describes a car's drivetrain. One line describes all of the points with a `4` value, one line describes all of the points with an `f` value, and one line describes all of the points with an `r` value. Here, `4` stands for four-wheel drive, `f` for front-wheel drive, and `r` for rear-wheel drive.

If this sounds strange, we can make it more clear by overlaying the lines on top of the raw data and then coloring everything according to `drv`.

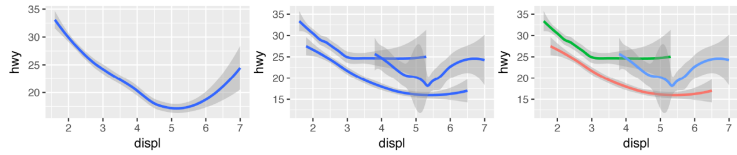


Notice that this plot contains two geoms in the same graph! If this makes you excited, buckle up. In the next section, we will learn how to place multiple geoms in the same plot.

ggplot2 provides over 30 geoms, and extension packages provide even more (see *https://www.ggplot2-exts.org* for a sampling). The best way to get a comprehensive overview is the **ggplot2** cheatsheet, which you can find at *http://rstudio.com/cheatsheets*. To learn more about any single geom, use help: `?geom_smooth`.

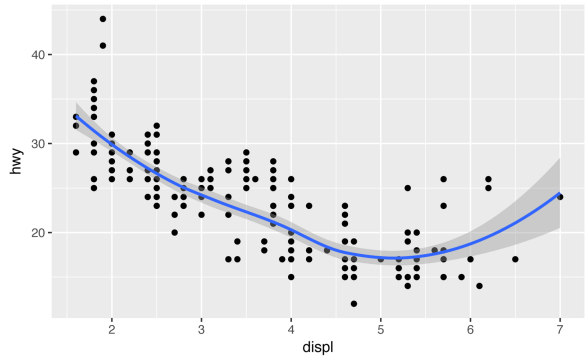
Many geoms, like `geom_smooth()`, use a single geometric object to display multiple rows of data. For these geoms, you can set the `group` aesthetic to a categorical variable to draw multiple objects. **ggplot2** will draw a separate object for each unique value of the grouping variable. In practice, **ggplot2** will automatically group the data for these geoms whenever you map an aesthetic to a discrete variable (as in the `linetype` example). It is convenient to rely on this feature because the group aesthetic by itself does not add a legend or distinguishing features to the geoms:

```
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_smooth(
    mapping = aes(x = displ, y = hwy, color = drv),
    show.legend = FALSE
  )
```



To display multiple geoms in the same plot, add multiple geom functions to `ggplot()`:

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

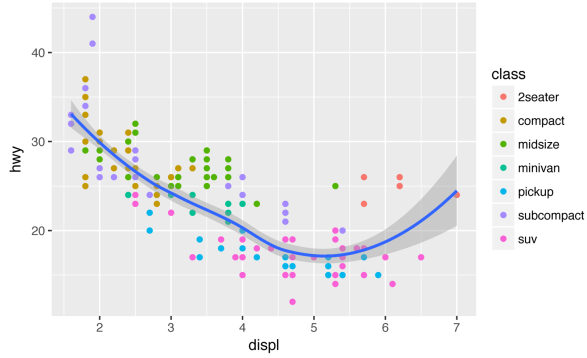


This, however, introduces some duplication in our code. Imagine if you wanted to change the y-axis to display `cty` instead of `hwy`. You'd need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to `ggplot()`. **ggplot2** will treat these mappings as global mappings that apply to each geom in the graph. In other words, this code will produce the same plot as the previous code:

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

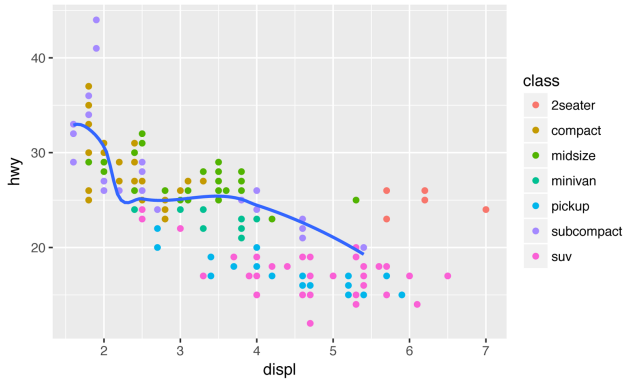
If you place mappings in a geom function, **ggplot2** will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. This makes it possible to display different aesthetics in different layers:

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



You can use the same idea to specify different `data` for each layer. Here, our smooth line displays just a subset of the `mpg` dataset, the subcompact cars. The local data argument in `geom_smooth()` overrides the global data argument in `ggplot()` for that layer only:

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



(You'll learn how `filter()` works in the next chapter: for now, just know that this command selects only the subcompact cars.)

Exercises

1. What geom would you use to draw a line chart? A boxplot? A histogram? An area chart?
2. Run this code in your head and predict what the output will look like. Then, run the code in R and check your predictions:

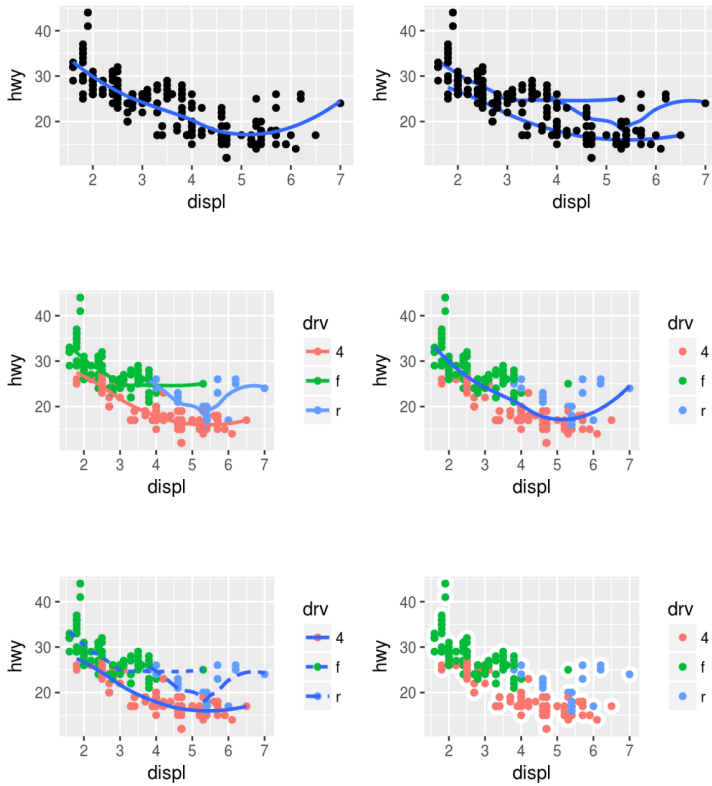
```
ggplot(
  data = mpg,
  mapping = aes(x = displ, y = hwy, color = drv)
) +
  geom_point() +
  geom_smooth(se = FALSE)
```

3. What does `show.legend = FALSE` do? What happens if you remove it? Why do you think I used it earlier in the chapter?
4. What does the `se` argument to `geom_smooth()` do?

5. Will these two graphs look different? Why/why not?

```
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)
  )
```

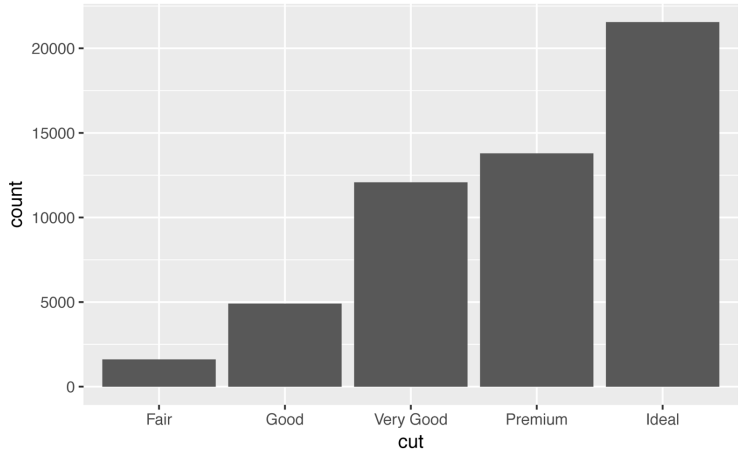
6. Re-create the R code necessary to generate the following graphs.



Statistical Transformations

Next, let's take a look at a bar chart. Bar charts seem simple, but they are interesting because they reveal something subtle about plots. Consider a basic bar chart, as drawn with `geom_bar()`. The following chart displays the total number of diamonds in the `diamonds` dataset, grouped by `cut`. The `diamonds` dataset comes in **ggplot2** and contains information about ~54,000 diamonds, including the `price`, `carat`, `color`, `clarity`, and `cut` of each diamond. The chart shows that more diamonds are available with high-quality cuts than with low quality cuts:

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

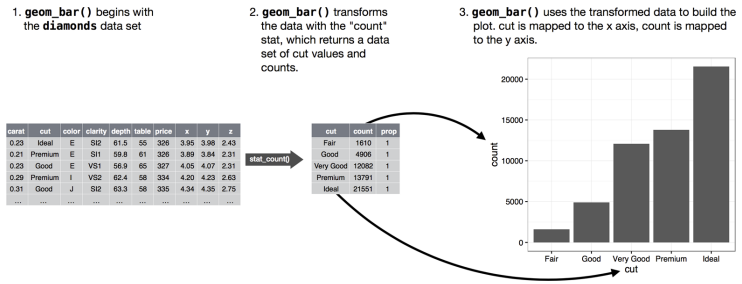


On the x-axis, the chart displays `cut`, a 'variable from `diamonds`. On the y-axis, it displays `count`, but `count` is not a variable in `diamonds`! Where does `count` come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

- Bar charts, histograms, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.
- Smoothers fit a model to your data and then plot predictions from the model.

- Boxplots compute a robust summary of the distribution and display a specially formatted box.

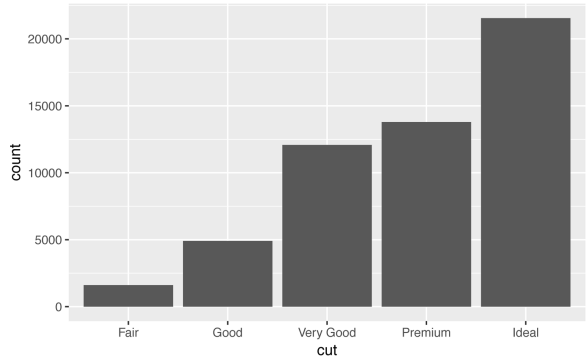
The algorithm used to calculate new values for a graph is called a *stat*, short for statistical transformation. The following figure describes how this process works with `geom_bar()`.



You can learn which stat a geom uses by inspecting the default value for the `stat` argument. For example, `?geom_bar` shows the default value for `stat` is "count," which means that `geom_bar()` uses `stat_count()`. `stat_count()` is documented on the same page as `geom_bar()`, and if you scroll down you can find a section called "Computed variables." That tells that it computes two new variables: `count` and `prop`.

You can generally use geoms and stats interchangeably. For example, you can re-create the previous plot using `stat_count()` instead of `geom_bar()`:

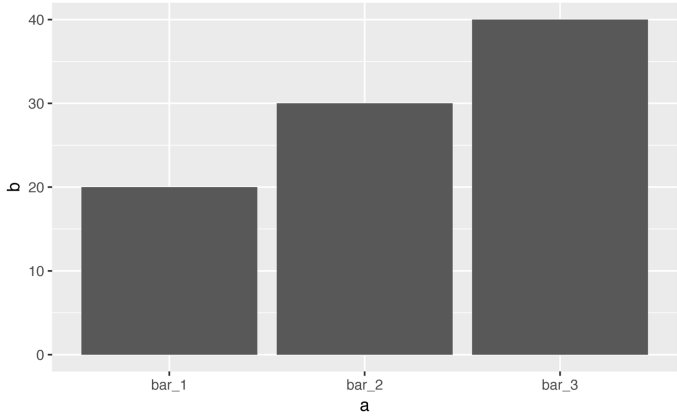
```
ggplot(data = diamonds) +
  stat_count(mapping = aes(x = cut))
```



This works because every geom has a default stat, and every stat has a default geom. This means that you can typically use geoms without worrying about the underlying statistical transformation. There are three reasons you might need to use a stat explicitly:

- You might want to override the default stat. In the following code, I change the stat of `geom_bar()` from count (the default) to identity. This lets me map the height of the bars to the raw values of a *y* variable. Unfortunately when people talk about bar charts casually, they might be referring to this type of bar chart, where the height of the bar is already present in the data, or the previous bar chart where the height of the bar is generated by counting rows.

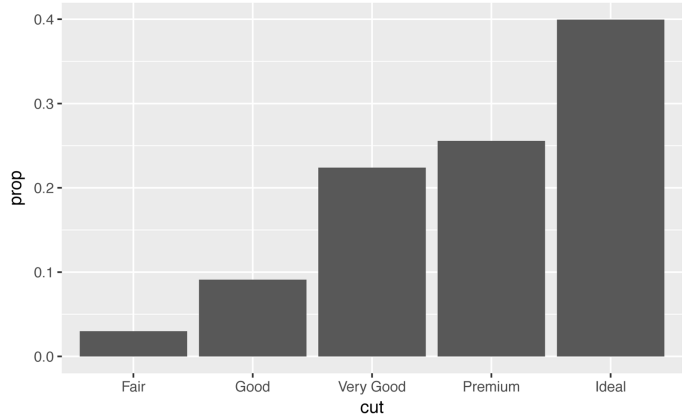
```
demo <- tribble(
  ~a,      ~b,
  "bar_1", 20,
  "bar_2", 30,
  "bar_3", 40
)

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



(Don't worry that you haven't seen `<-` or `tibble()` before. You might be able to guess at their meaning from the context, and you'll learn exactly what they do soon!)

- You might want to override the default mapping from transformed variables to aesthetics. For example, you might want to display a bar chart of proportion, rather than count:

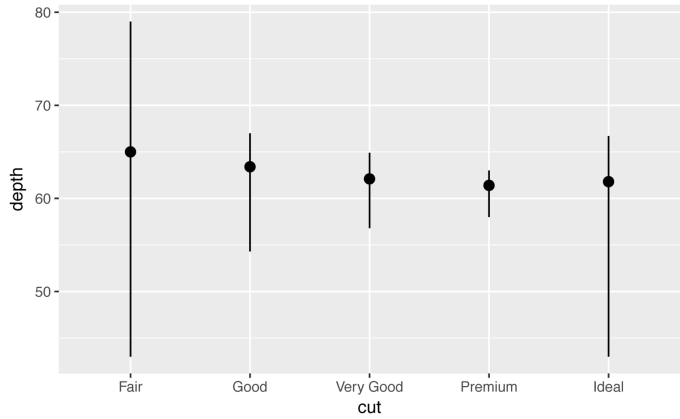
```
ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, y = ..prop.., group = 1)
  )
```



To find the variables computed by the stat, look for the help section titled "Computed variables."

- You might want to draw greater attention to the statistical transformation in your code. For example, you might use `stat_summary()`, which summarizes the `y` values for each unique `x` value, to draw attention to the summary that you're computing:

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



ggplot2 provides over 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g., `?stat_bin`. To see a complete list of stats, try the **ggplot2** cheatsheet.

Exercises

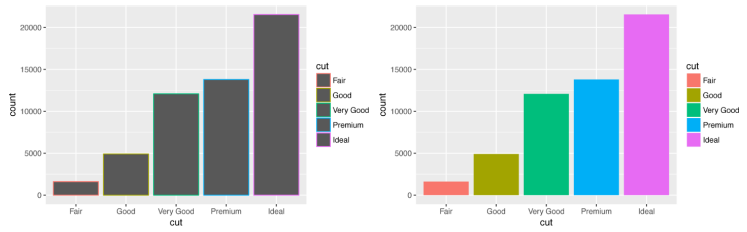
1. What is the default geom associated with `stat_summary()`? How could you rewrite the previous plot to use that geom function instead of the stat function?
2. What does `geom_col()` do? How is it different to `geom_bar()`?
3. Most geoms and stats come in pairs that are almost always used in concert. Read through the documentation and make a list of all the pairs. What do they have in common?
4. What variables does `stat_smooth()` compute? What parameters control its behavior?
5. In our proportion bar chart, we need to set `group = 1`. Why? In other words what is the problem with these two graphs?

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

Position Adjustments

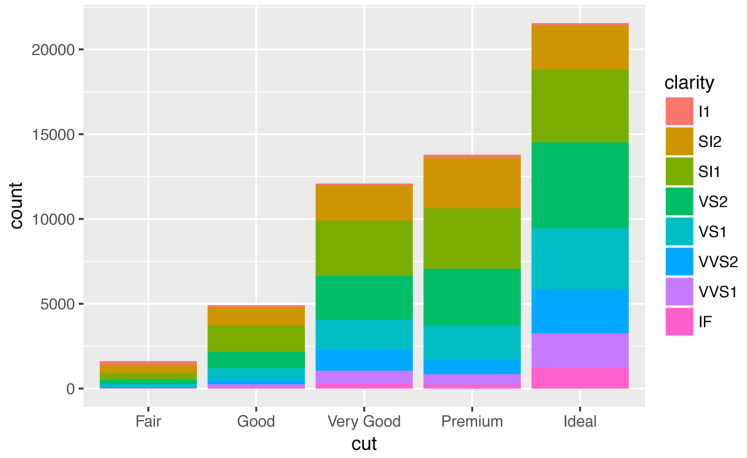
There's one more piece of magic associated with bar charts. You can color a bar chart using either the `color` aesthetic, or more usefully, `fill`:

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



Note what happens if you map the `fill` aesthetic to another variable, like `clarity`: the bars are automatically stacked. Each colored rectangle represents a combination of `cut` and `clarity`:

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



The stacking is performed automatically by the *position adjustment* specified by the `position` argument. If you don't want a stacked bar

chart, you can use one of three other options: `"identity"`, `"dodge"` or `"fill"`:

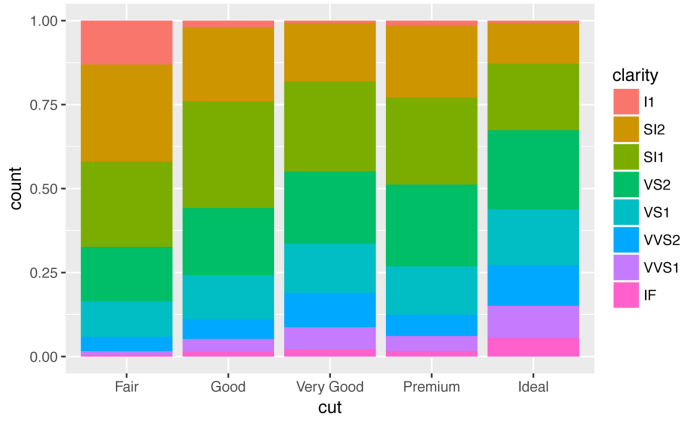
- `position = "identity"` will place each object exactly where it falls in the context of the graph. This is not very useful for bars, because it overlaps them. To see that overlapping we either need to make the bars slightly transparent by setting `alpha` to a small value, or completely transparent by setting `fill = NA`:

```
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")
```

  The identity position adjustment is more useful for 2D geoms, like points, where it is the default.

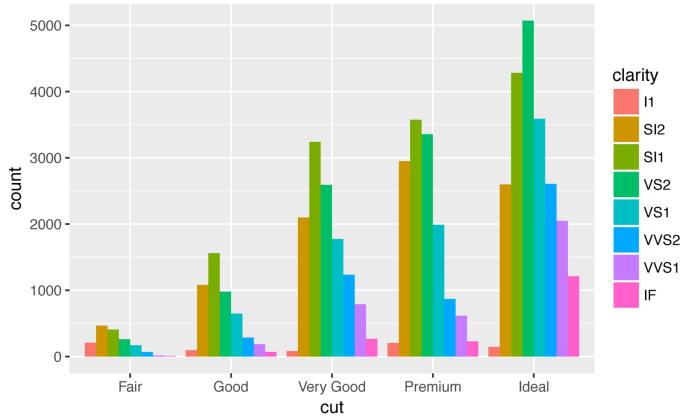
- `position = "fill"` works like stacking, but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups:

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

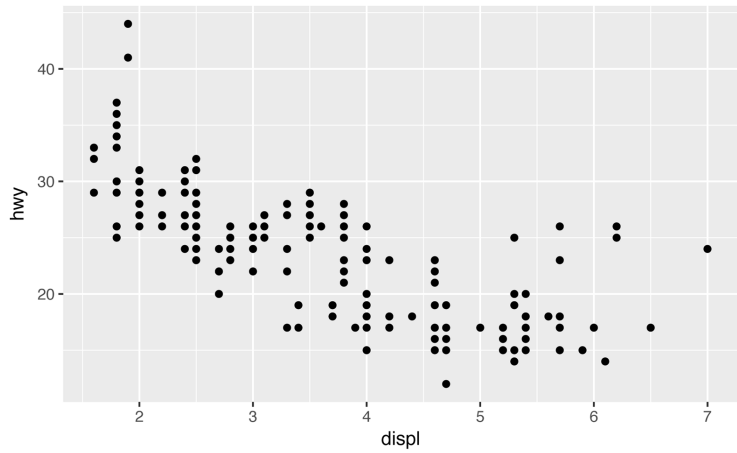


- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values:

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



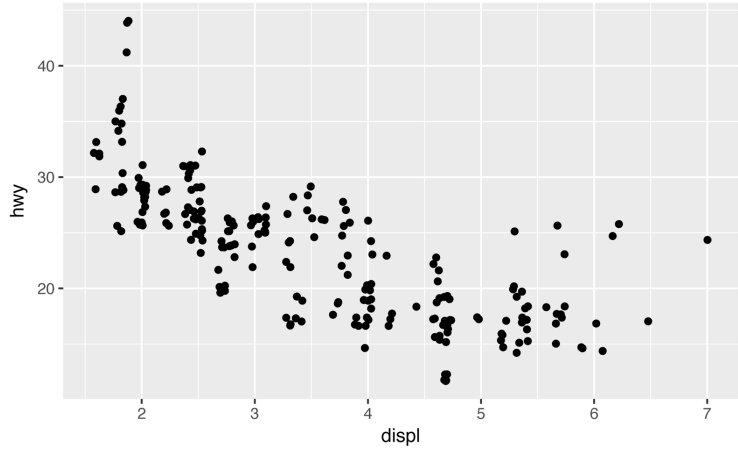
There's one other type of adjustment that's not useful for bar charts, but it can be very useful for scatterplots. Recall our first scatterplot. Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset?



The values of `hwy` and `displ` are rounded so the points appear on a grid and many points overlap each other. This problem is known as *overplotting*. This arrangement makes it hard to see where the mass of the data is. Are the data points spread equally throughout the graph, or is there one special combination of `hwy` and `displ` that contains 109 values?

You can avoid this gridding by setting the position adjustment to "jitter." `position = "jitter"` adds a small amount of random noise to each point. This spreads the points out because no two points are likely to receive the same amount of random noise:

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



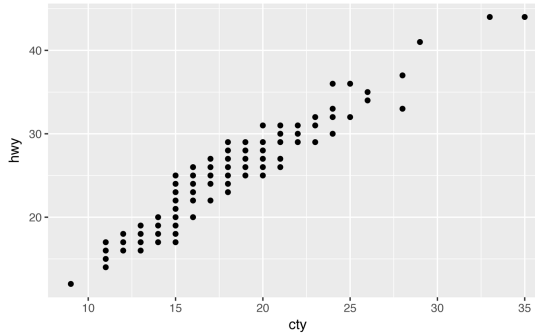
Adding randomness seems like a strange way to improve your plot, but while it makes your graph less accurate at small scales, it makes your graph *more* revealing at large scales. Because this is such a useful operation, **ggplot2** comes with a shorthand for `geom_point(position = "jitter")`: `geom_jitter()`.

To learn more about a position adjustment, look up the help page associated with each adjustment: `?position_dodge`, `?position_fill`, `?position_identity`, `?position_jitter`, and `?position_stack`.

Exercises

1. What is the problem with this plot? How could you improve it?

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



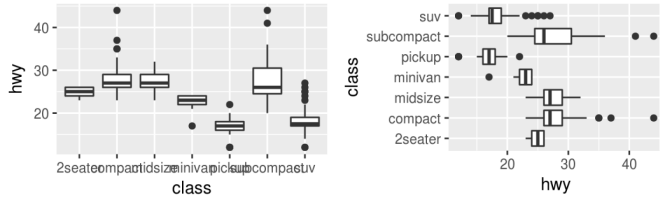
2. What parameters to `geom_jitter()` control the amount of jittering?
3. Compare and contrast `geom_jitter()` with `geom_count()`.
4. What's the default position adjustment for `geom_boxplot()`? Create a visualization of the `mpg` dataset that demonstrates it.

Coordinate Systems

Coordinate systems are probably the most complicated part of **ggplot2**. The default coordinate system is the Cartesian coordinate system where the x and y position act independently to find the location of each point. There are a number of other coordinate systems that are occasionally helpful:

- `coord_flip()` switches the x- and y-axes. This is useful (for example) if you want horizontal boxplots. It's also useful for long labels—it's hard to get them to fit without overlapping on the x-axis:

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

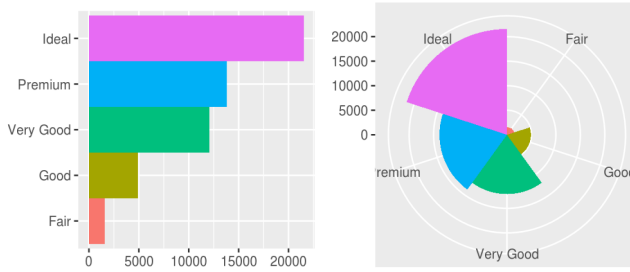


- `coord_quickmap()` sets the aspect ratio correctly for maps. This is very important if you're plotting spatial data with **ggplot2** (which unfortunately we don't have the space to cover in this book):

```
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()
```

- `coord_polar()` uses polar coordinates. Polar coordinates reveal an interesting connection between a bar chart and a Coxcomb chart:

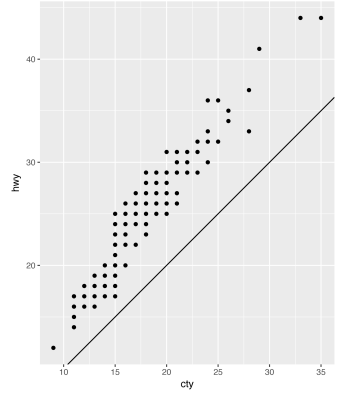
```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)

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



Exercises

1. Turn a stacked bar chart into a pie chart using `coord_polar()`.
2. What does `labs()` do? Read the documentation.
3. What's the difference between `coord_quickmap()` and `coord_map()`?
4. What does the following plot tell you about the relationship between city and highway mpg? Why is `coord_fixed()` important? What does `geom_abline()` do?

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



The Layered Grammar of Graphics

In the previous sections, you learned much more than how to make scatterplots, bar charts, and boxplots. You learned a foundation that you can use to make *any* type of plot with **ggplot2**. To see this, let's add position adjustments, stats, coordinate systems, and faceting to our code template:

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

Our new template takes seven parameters, the bracketed words that appear in the template. In practice, you rarely need to supply all seven parameters to make a graph because **ggplot2** will provide useful defaults for everything except the data, the mappings, and the geom function.

The seven parameters in the template compose the grammar of graphics, a formal system for building plots. The grammar of graphics is based on the insight that you can uniquely describe *any* plot as a combination of a dataset, a geom, a set of mappings, a stat, a position adjustment, a coordinate system, and a faceting scheme.

To see how this works, consider how you could build a basic plot from scratch: you could start with a dataset and then transform it into the information that you want to display (with a stat):

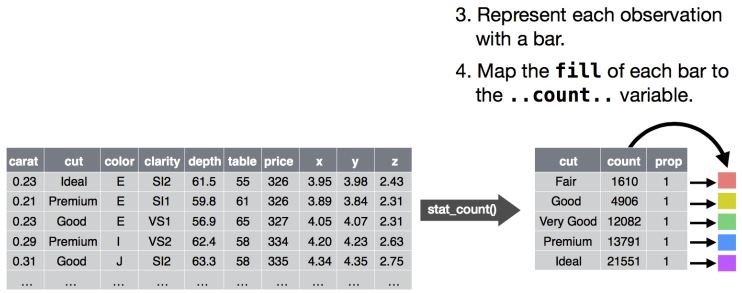
1. Begin with the **`diamonds`** data set

2. Compute counts for each cut value with **`stat_count()`**.

carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
…	…	…	…	…	…	…	…	…	…

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

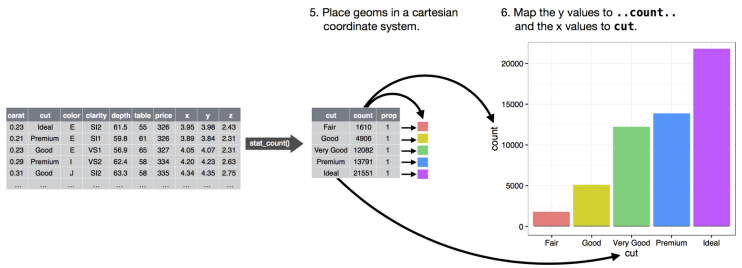
Next, you could choose a geometric object to represent each observation in the transformed data. You could then use the aesthetic properties of the geoms to represent variables in the data. You would map the values of each variable to the levels of an aesthetic:



carat	cut	color	clarity	depth	table	price	x	y	z
0.23	Ideal	E	SI2	61.5	55	326	3.95	3.98	2.43
0.21	Premium	E	SI1	59.8	61	326	3.89	3.84	2.31
0.23	Good	E	VS1	56.9	65	327	4.05	4.07	2.31
0.29	Premium	I	VS2	62.4	58	334	4.20	4.23	2.63
0.31	Good	J	SI2	63.3	58	335	4.34	4.35	2.75
...	...	...	...	...	...	...	...	...	...

cut	count	prop
Fair	1610	1
Good	4906	1
Very Good	12082	1
Premium	13791	1
Ideal	21551	1

You'd then select a coordinate system to place the geoms into. You'd use the location of the objects (which is itself an aesthetic property) to display the values of the `x` and `y` variables. At that point, you would have a complete graph, but you could further adjust the positions of the geoms within the coordinate system (a position adjustment) or split the graph into subplots (faceting). You could also extend the plot by adding one or more additional layers, where each additional layer uses a dataset, a geom, a set of mappings, a stat, and a position adjustment:



You could use this method to build *any* plot that you imagine. In other words, you can use the code template that you've learned in this chapter to build hundreds of thousands of unique plots.

CHAPTER 2

Workflow: Basics

You now have some experience running R code. I didn't give you many details, but you've obviously figured out the basics, or you would've thrown this book away in frustration! Frustration is natural when you start programming in R, because it is such a stickler for punctuation, and even one character out of place will cause it to complain. But while you should expect to be a little frustrated, take comfort in that it's both typical and temporary: it happens to everyone, and the only way to get over it is to keep trying.

Before we go any further, let's make sure you've got a solid foundation in running R code, and that you know about some of the most helpful RStudio features.

Coding Basics

Let's review some basics we've so far omitted in the interests of getting you plotting as quickly as possible. You can use R as a calculator:

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

You can create new objects with `<-`:

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

All R statements where you create objects, *assignment* statements, have the same form:

```
object_name <- value
```

When reading that code say "object name gets value" in your head.

You will make lots of assignments and `<-` is a pain to type. Don't be lazy and use `=`: it will work, but it will cause confusion later. Instead, use RStudio's keyboard shortcut: Alt-– (the minus sign). Notice that RStudio automagically surrounds `<-` with spaces, which is a good code formatting practice. Code is miserable to read on a good day, so giveyoureyesabreak and use spaces.

What's in a Name?

Object names must start with a letter, and can only contain letters, numbers, `_`, and `.`. You want your object names to be descriptive, so you'll need a convention for multiple words. I recommend *snake_case* where you separate lowercase words with `_`:

```
i_use_snake_case
otherPeopleUseCamelCase
some.people.use.periods
And_aFew.People_RENOUNCEconvention
```

We'll come back to code style later, in Chapter 15.

You can inspect an object by typing its name:

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

Make another assignment:

```
this_is_a_really_long_name <- 2.5
```

To inspect this object, try out RStudio's completion facility: type "this," press Tab, add characters until you have a unique prefix, then press Return.

Oops, you made a mistake! `this_is_a_really_long_name` should have value 3.5 not 2.5. Use another keyboard shortcut to help you fix it. Type "this" then press Cmd/Ctrl-↑. That will list all the commands you've typed that start with those letters. Use the arrow keys to navigate, then press Enter to retype the command. Change 2.5 to 3.5 and rerun.

Make yet another assignment:

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

Let's try to inspect it:

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

There's an implied contract between you and R: it will do the tedious computation for you, but in return, you must be completely precise in your instructions. Typos matter. Case matters.

Calling Functions

R has a large collection of built-in functions that are called like this:

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

Let's try using `seq()`, which makes regular *seq*uences of numbers and, while we're at it, learn more helpful features of RStudio. Type `se` and hit Tab. A pop-up shows you possible completions. Specify `seq()` by typing more (a "q") to disambiguate, or by using ↑/↓ arrows to select. Notice the floating tooltip that pops up, reminding you of the function's arguments and purpose. If you want more help, press F1 to get all the details in the help tab in the lower-right pane.

Press Tab once more when you've selected the function you want. RStudio will add matching opening (`(`) and closing (`)`) parentheses for you. Type the arguments `1, 10` and hit Return:

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

Type this code and notice similar assistance help with the paired quotation marks:

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

Quotation marks and parentheses must always come in a pair. RStudio does its best to help you, but it's still possible to mess up and end up with a mismatch. If this happens, R will show you the continuation character "+":

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

The + tells you that R is waiting for more input; it doesn't think you're done yet. Usually that means you've forgotten either a " or a). Either add the missing pair, or press Esc to abort the expression and try again.

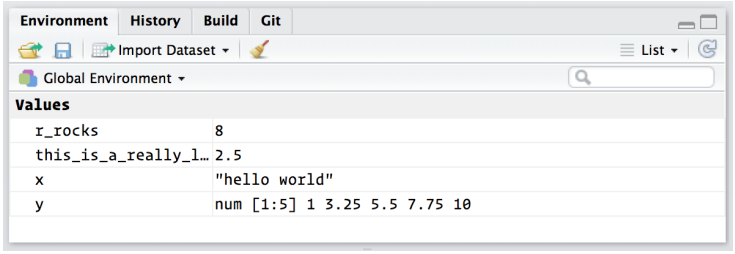
If you make an assignment, you don't get to see the value. You're then tempted to immediately double-check the result:

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

This common action can be shortened by surrounding the assignment with parentheses, which causes assignment and "print to screen" to happen:

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

Now look at your environment in the upper-right pane:



Here you can see all of the objects that you've created.

Exercises

1. Why does this code not work?

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

   Look carefully! (This may seem like an exercise in pointlessness, but training your brain to notice even the tiniest difference will pay off when programming.)

2. Tweak each of the following R commands so that they run correctly:

```
library(tidyverse)

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

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

3. Press Alt-Shift-K. What happens? How can you get to the same place using the menus?

CHAPTER 3

Data Transformation with dplyr

Introduction

Visualization is an important tool for insight generation, but it is rare that you get the data in exactly the right form you need. Often you'll need to create some new variables or summaries, or maybe you just want to rename the variables or reorder the observations in order to make the data a little easier to work with. You'll learn how to do all that (and more!) in this chapter, which will teach you how to transform your data using the **dplyr** package and a new dataset on flights departing New York City in 2013.

Prerequisites

In this chapter we're going to focus on how to use the **dplyr** package, another core member of the tidyverse. We'll illustrate the key ideas using data from the **nycflights13** package, and use **ggplot2** to help us understand the data.

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

Take careful note of the conflicts message that's printed when you load the tidyverse. It tells you that **dplyr** overwrites some functions in base R. If you want to use the base version of these functions after loading **dplyr**, you'll need to use their full names: `stats::filter()` and `stats::lag()`.

nycflights13

To explore the basic data manipulation verbs of **dplyr**, we'll use `nycflights13::flights`. This data frame contains all 336,776 flights that departed from New York City in 2013. The data comes from the US Bureau of Transportation Statistics, and is documented in `?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>
```

You might notice that this data frame prints a little differently from other data frames you might have used in the past: it only shows the first few rows and all the columns that fit on one screen. (To see the whole dataset, you can run `View(flights)`, which will open the dataset in the RStudio viewer.) It prints differently because it's a *tibble*. Tibbles are data frames, but slightly tweaked to work better in the tidyverse. For now, you don't need to worry about the differences; we'll come back to tibbles in more detail in Part II.

You might also have noticed the row of three- (or four-) letter abbreviations under the column names. These describe the type of each variable:

- `int` stands for integers.
- `dbl` stands for doubles, or real numbers.
- `chr` stands for character vectors, or strings.
- `dttm` stands for date-times (a date + a time).

There are three other common types of variables that aren't used in this dataset but you'll encounter later in the book:

- `lgl` stands for logical, vectors that contain only `TRUE` or `FALSE`.
- `fctr` stands for factors, which R uses to represent categorical variables with fixed possible values.
- `date` stands for dates.

dplyr Basics

In this chapter you are going to learn the five key **dplyr** functions that allow you to solve the vast majority of your data-manipulation challenges:

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarize()`).

These can all be used in conjunction with `group_by()`, which changes the scope of each function from operating on the entire dataset to operating on it group-by-group. These six functions provide the verbs for a language of data manipulation.

All verbs work similarly:

1. The first argument is a data frame.
2. The subsequent arguments describe what to do with the data frame, using the variable names (without quotes).
3. The result is a new data frame.

Together these properties make it easy to chain together multiple simple steps to achieve a complex result. Let's dive in and see how these verbs work.

Filter Rows with filter()

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and

subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

```
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>
```

When you run that line of code, **dplyr** executes the filtering operation and returns a new data frame. **dplyr** functions never modify their inputs, so if you want to save the result, you'll need to use the assignment operator, `<-`:

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

R either prints out the results, or saves them to a variable. If you want to do both, you can wrap the assignment in parentheses:

```
(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>
```

Comparisons

To use filtering effectively, you have to know how to select the observations that you want using the comparison operators. R provides the standard suite: `>`, `>=`, `<`, `<=`, `!=` (not equal), and `==` (equal).

When you're starting out with R, the easiest mistake to make is to use = instead of == when testing for equality. When this happens you'll get an informative error:

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

There's another common problem you might encounter when using ==: floating-point numbers. These results might surprise you!

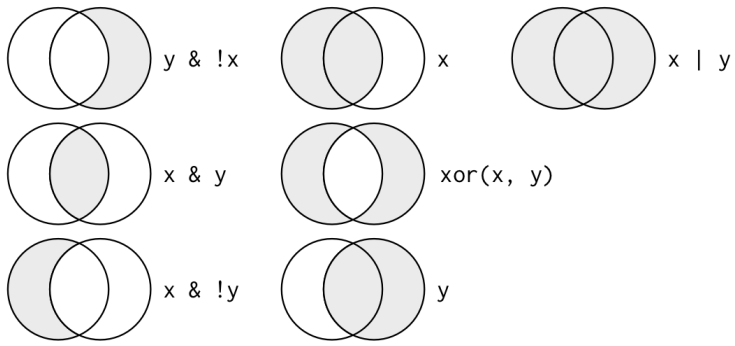
```
sqrt(2) ^ 2 == 2
#> [1] FALSE
1/49 * 49 == 1
#> [1] FALSE
```

Computers use finite precision arithmetic (they obviously can't store an infinite number of digits!) so remember that every number you see is an approximation. Instead of relying on ==, use `near()`:

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

Logical Operators

Multiple arguments to `filter()` are combined with "and": every expression must be true in order for a row to be included in the output. For other types of combinations, you'll need to use Boolean operators yourself: `&` is "and," `|` is "or," and `!` is "not." The following figure shows the complete set of Boolean operations.



The following code finds all flights that departed in November or December:

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

The order of operations doesn't work like English. You can't write `filter(flights, month == 11 | 12)`, which you might literally translate into "finds all flights that departed in November or December." Instead it finds all months that equal `11 | 12`, an expression that evaluates to `TRUE`. In a numeric context (like here), `TRUE` becomes one, so this finds all flights in January, not November or December. This is quite confusing!

A useful shorthand for this problem is `x %in% y`. This will select every row where `x` is one of the values in `y`. We could use it to rewrite the preceding code:

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

Sometimes you can simplify complicated subsetting by remembering De Morgan's law: `!(x & y)` is the same as `!x | !y`, and `!(x | y)` is the same as `!x & !y`. For example, if you wanted to find flights that weren't delayed (on arrival or departure) by more than two hours, you could use either of the following two filters:

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

As well as `&` and `|`, R also has `&&` and `||`. Don't use them here! You'll learn when you should use them in "Conditional Execution" on page 276.

Whenever you start using complicated, multipart expressions in `filter()`, consider making them explicit variables instead. That makes it much easier to check your work. You'll learn how to create new variables shortly.

Missing Values

One important feature of R that can make comparison tricky is missing values, or `NA`s ("not availables"). `NA` represents an unknown value so missing values are "contagious"; almost any operation involving an unknown value will also be unknown:

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

```
NA / 2
#> [1] NA
```

The most confusing result is this one:

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

It's easiest to understand why this is true with a bit more context:

```
# Let x be Mary's age. We don't know how old she is.
x <- NA

# Let y be John's age. We don't know how old he is.
y <- NA

# Are John and Mary the same age?
x == y
#> [1] NA
# We don't know!
```

If you want to determine if a value is missing, use `is.na()`:

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

`filter()` only includes rows where the condition is `TRUE`; it excludes both `FALSE` and `NA` values. If you want to preserve missing values, ask for them explicitly:

```
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
```

Exercises

1. Find all flights that:
   a. Had an arrival delay of two or more hours
   b. Flew to Houston (`IAH` or `HOU`)
   c. Were operated by United, American, or Delta

d. Departed in summer (July, August, and September)

e. Arrived more than two hours late, but didn't leave late

f. Were delayed by at least an hour, but made up over 30 minutes in flight

g. Departed between midnight and 6 a.m. (inclusive)

2. Another useful **dplyr** filtering helper is `between()`. What does it do? Can you use it to simplify the code needed to answer the previous challenges?

3. How many flights have a missing `dep_time`? What other variables are missing? What might these rows represent?

4. Why is `NA ^ 0` not missing? Why is `NA | TRUE` not missing? Why is `FALSE & NA` not missing? Can you figure out the general rule? (`NA * 0` is a tricky counterexample!)

Arrange Rows with arrange()

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
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>
```

Use `desc()` to reorder by a column in descending order:

```
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>,
```

Missing values are always sorted at the end:

```
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
```

Exercises

1. How could you use `arrange()` to sort all missing values to the start? (Hint: use `is.na()`.)
2. Sort `flights` to find the most delayed flights. Find the flights that left earliest.
3. Sort `flights` to find the fastest flights.
4. Which flights traveled the longest? Which traveled the shortest?

Select Columns with select()

It's not uncommon to get datasets with hundreds or even thousands of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to

rapidly zoom in on a useful subset using operations based on the names of the variables.

`select()` is not terribly useful with the flight data because we only have 19 variables, but you can still get the general idea:

```
# Select columns by name
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

# Select all columns between year and day (inclusive)
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

# Select all columns except those from year to day (inclusive)
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>
```

There are a number of helper functions you can use within `select()`:

- `starts_with("abc")` matches names that begin with "abc".
- `ends_with("xyz")` matches names that end with "xyz".
- `contains("ijk")` matches names that contain "ijk".
- `matches("(.)\\1")` selects variables that match a regular expression. This one matches any variables that contain repeated characters. You'll learn more about regular expressions in Chapter 11.
- `num_range("x", 1:3)` matches `x1`, `x2`, and `x3`.

See `?select` for more details.

`select()` can be used to rename variables, but it's rarely useful because it drops all of the variables not explicitly mentioned. Instead, use `rename()`, which is a variant of `select()` that keeps all the variables that aren't explicitly mentioned:

```
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>
```

Another option is to use `select()` in conjunction with the `everything()` helper. This is useful if you have a handful of variables you'd like to move to the start of the data frame:

```
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>
```

Exercises

1. Brainstorm as many ways as possible to select `dep_time`, `dep_delay`, `arr_time`, and `arr_delay` from `flights`.
2. What happens if you include the name of a variable multiple times in a `select()` call?
3. What does the `one_of()` function do? Why might it be helpful in conjunction with this vector?

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

4. Does the result of running the following code surprise you? How do the select helpers deal with case by default? How can you change that default?

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

Add New Variables with mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset so we'll start by creating a narrower dataset so we can see the new variables. Remember that when you're in RStudio, the easiest way to see all the columns is `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>
```

Note that you can refer to columns that you've just created:

```
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>
```

If you only want to keep the new variables, use `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
```

Useful Creation Functions

There are many functions for creating new variables that you can use with `mutate()`. The key property is that the function must be vectorized: it must take a vector of values as input, and return a vector with the same number of values as output. There's no way to list every possible function that you might use, but here's a selection of functions that are frequently useful:

Arithmetic operators `+`, `-`, `*`, `/`, `^`
: These are all vectorized, using the so-called "recycling rules." If one parameter is shorter than the other, it will be automatically extended to be the same length. This is most useful when one of the arguments is a single number: `air_time / 60`, `hours * 60 + minute`, etc.

  Arithmetic operators are also useful in conjunction with the aggregate functions you'll learn about later. For example, `x / sum(x)` calculates the proportion of a total, and `y - mean(y)` computes the difference from the mean.

Modular arithmetic (`%/%` and `%%`)
: `%/%` (integer division) and `%%` (remainder), where `x == y * (x %/% y) + (x %% y)`. Modular arithmetic is a handy tool because it allows you to break integers into pieces. For example, in the flights dataset, you can compute `hour` and `minute` from `dep_time` with:

```
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
```

Logs `log()`, `log2()`, `log10()`
: Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude. They also convert multiplicative relationships to additive, a feature we'll come back to in Part IV.

  All else being equal, I recommend using `log2()` because it's easy to interpret: a difference of 1 on the log scale corresponds to doubling on the original scale and a difference of –1 corresponds to halving.

Offsets
: `lead()` and `lag()` allow you to refer to leading or lagging values. This allows you to compute running differences (e.g., `x - lag(x)`) or find when values change (`x != lag(x)`). They are most useful in conjunction with `group_by()`, which you'll learn about shortly:

```
(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
```

Cumulative and rolling aggregates
: R provides functions for running sums, products, mins, and maxes: `cumsum()`, `cumprod()`, `cummin()`, `cummax()`; and **dplyr** provides `cummean()` for cumulative means. If you need rolling aggregates (i.e., a sum computed over a rolling window), try the **RcppRoll** package:

```
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
```

Logical comparisons `<`, `<=`, `>`, `>=`, `!=`
: If you're doing a complex sequence of logical operations it's often a good idea to store the interim values in new variables so you can check that each step is working as expected.

Ranking

There are a number of ranking functions, but you should start with `min_rank()`. It does the most usual type of ranking (e.g., first, second, third, fourth). The default gives the smallest values the smallest ranks; use `desc(x)` to give the largest values the smallest ranks:

```
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
```

If `min_rank()` doesn't do what you need, look at the variants `row_number()`, `dense_rank()`, `percent_rank()`, `cume_dist()`, and `ntile()`. See their help pages for more details:

```
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
```

Exercises

1. Currently `dep_time` and `sched_dep_time` are convenient to look at, but hard to compute with because they're not really continuous numbers. Convert them to a more convenient representation of number of minutes since midnight.
2. Compare `air_time` with `arr_time - dep_time`. What do you expect to see? What do you see? What do you need to do to fix it?
3. Compare `dep_time`, `sched_dep_time`, and `dep_delay`. How would you expect those three numbers to be related?
4. Find the 10 most delayed flights using a ranking function. How do you want to handle ties? Carefully read the documentation for `min_rank()`.
5. What does `1:3 + 1:10` return? Why?
6. What trigonometric functions does R provide?

Grouped Summaries with summarize()

The last key verb is `summarize()`. It collapses a data frame to a single row:

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

(We'll come back to what that `na.rm = TRUE` means very shortly.)

`summarize()` is not terribly useful unless we pair it with `group_by()`. This changes the unit of analysis from the complete dataset to individual groups. Then, when you use the **dplyr** verbs on a grouped data frame they'll be automatically applied "by group." For example, if we applied exactly the same code to a data frame grouped by date, we get the average delay per date:

```
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
```

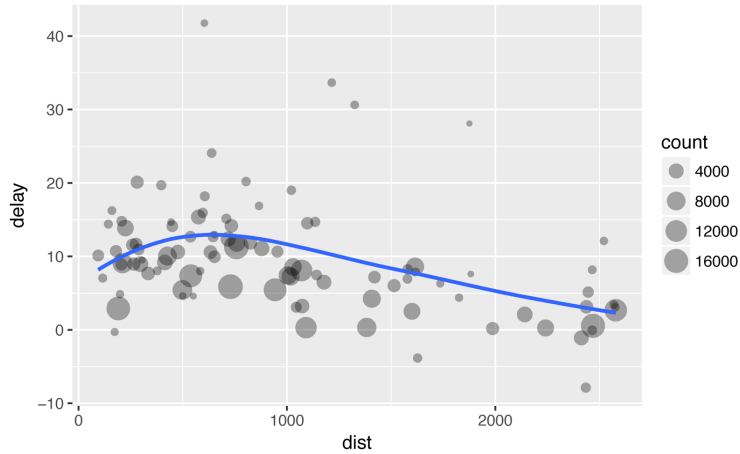
Together `group_by()` and `summarize()` provide one of the tools that you'll use most commonly when working with **dplyr**: grouped summaries. But before we go any further with this, we need to introduce a powerful new idea: the pipe.

Combining Multiple Operations with the Pipe

Imagine that we want to explore the relationship between the distance and average delay for each location. Using what you know about **dplyr**, you might write code like this:

```
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")

# It looks like delays increase with distance up to ~750 miles
# and then decrease. Maybe as flights get longer there's more
# ability to make up delays in the air?
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'
```



There are three steps to prepare this data:

1. Group flights by destination.
2. Summarize to compute distance, average delay, and number of flights.
3. Filter to remove noisy points and Honolulu airport, which is almost twice as far away as the next closest airport.

This code is a little frustrating to write because we have to give each intermediate data frame a name, even though we don't care about it. Naming things is hard, so this slows down our analysis.

There's another way to tackle the same problem with the pipe, `%>%`:

```
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")
```

This focuses on the transformations, not what's being transformed, which makes the code easier to read. You can read it as a series of imperative statements: group, then summarize, then filter. As suggested by this reading, a good way to pronounce `%>%` when reading code is "then."

Behind the scenes, `x %>% f(y)` turns into `f(x, y)`, and `x %>% f(y) %>% g(z)` turns into `g(f(x, y), z)`, and so on. You can use the pipe to rewrite multiple operations in a way that you can read left-to-right, top-to-bottom. We'll use piping frequently from now on because it considerably improves the readability of code, and we'll come back to it in more detail in Chapter 14.

Working with the pipe is one of the key criteria for belonging to the tidyverse. The only exception is **ggplot2**: it was written before the pipe was discovered. Unfortunately, the next iteration of **ggplot2**, **ggvis**, which does use the pipe, isn't ready for prime time yet.

Missing Values

You may have wondered about the `na.rm` argument we used earlier. What happens if we don't set it?

```
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
```

We get a lot of missing values! That's because aggregation functions obey the usual rule of missing values: if there's any missing value in the input, the output will be a missing value. Fortunately, all aggre-

gation functions have an `na.rm` argument, which removes the missing values prior to computation:

```
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
```

In this case, where missing values represent cancelled flights, we could also tackle the problem by first removing the cancelled flights. We'll save this dataset so we can reuse it in the next few examples:

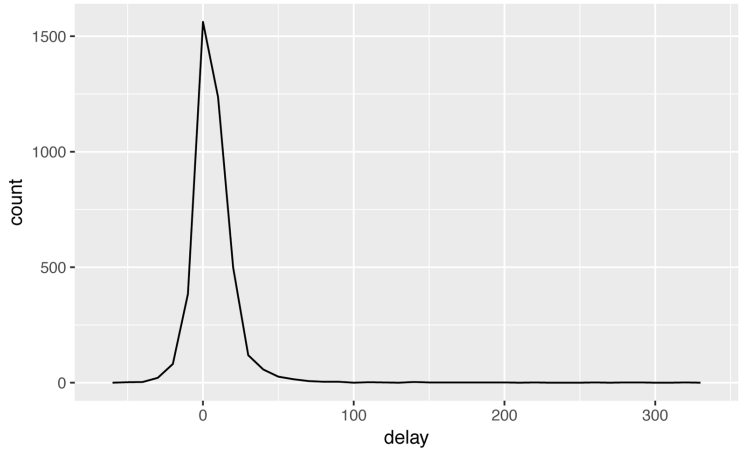
```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

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 rows
```

Counts

Whenever you do any aggregation, it's always a good idea to include either a count (`n()`), or a count of nonmissing values (`sum(!is.na(x))`). That way you can check that you're not drawing conclusions based on very small amounts of data. For example, let's

look at the planes (identified by their tail number) that have the highest average delays:

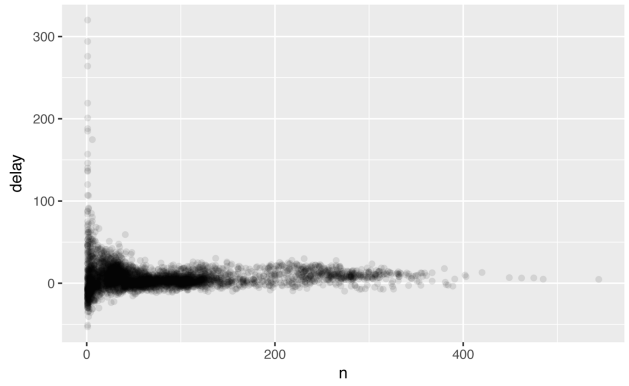
```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay)
  )

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



Wow, there are some planes that have an *average* delay of 5 hours (300 minutes)!

The story is actually a little more nuanced. We can get more insight if we draw a scatterplot of number of flights versus average delay:

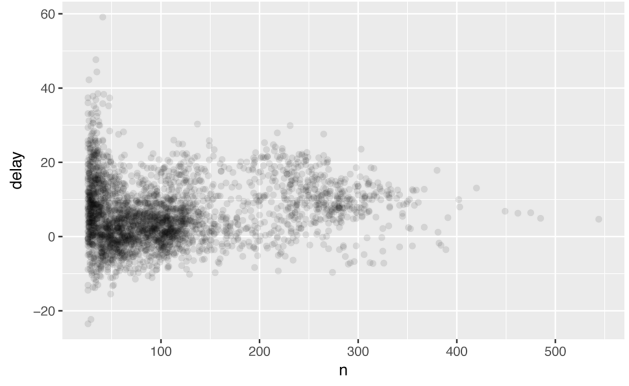
```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

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



Not surprisingly, there is much greater variation in the average delay when there are few flights. The shape of this plot is very characteristic: whenever you plot a mean (or other summary) versus group size, you'll see that the variation decreases as the sample size increases.

When looking at this sort of plot, it's often useful to filter out the groups with the smallest numbers of observations, so you can see more of the pattern and less of the extreme variation in the smallest groups. This is what the following code does, as well as showing you a handy pattern for integrating **ggplot2** into **dplyr** flows. It's a bit painful that you have to switch from `%>%` to `+`, but once you get the hang of it, it's quite convenient:

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



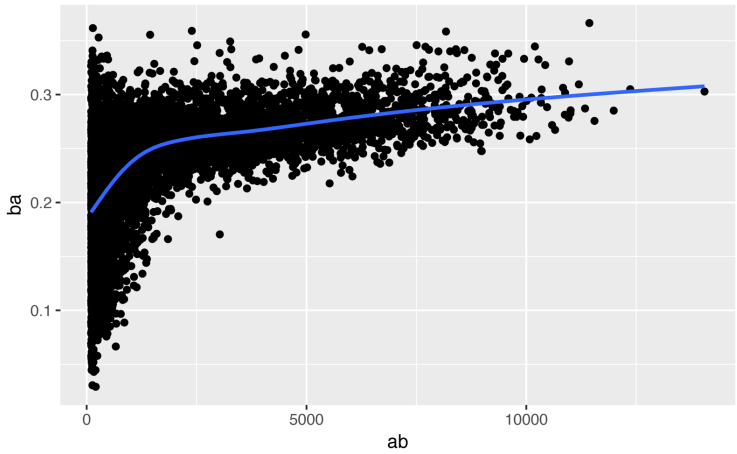
RStudio tip: a useful keyboard shortcut is Cmd/Ctrl-Shift-P. This resends the previously sent chunk from the editor to the console. This is very convenient when you're (e.g.) exploring the value of `n` in the preceding example. You send the whole block once with Cmd/Ctrl-Enter, then you modify the value of `n` and press Cmd/Ctrl-Shift-P to resend the complete block.

There's another common variation of this type of pattern. Let's look at how the average performance of batters in baseball is related to the number of times they're at bat. Here I use data from the **Lahman** package to compute the batting average (number of hits / number of attempts) of every major league baseball player.

When I plot the skill of the batter (measured by the batting average, `ba`) against the number of opportunities to hit the ball (measured by at bat, `ab`), you see two patterns:

- As above, the variation in our aggregate decreases as we get more data points.
- There's a positive correlation between skill (`ba`) and opportunities to hit the ball (`ab`). This is because teams control who gets to play, and obviously they'll pick their best players:

```
# Convert to a tibble so it prints nicely
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'
```



This also has important implications for ranking. If you naively sort on `desc(ba)`, the people with the best batting averages are clearly lucky, not skilled:

```
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
```

You can find a good explanation of this problem at *http://bit.ly/Bayesbbal* and *http://bit.ly/notsortavg*.

Useful Summary Functions

Just using means, counts, and sum can get you a long way, but R provides many other useful summary functions:

Measures of location

We've used `mean(x)`, but `median(x)` is also useful. The mean is the sum divided by the length; the median is a value where 50% of x is above it, and 50% is below it.

It's sometimes useful to combine aggregation with logical subsetting. We haven't talked about this sort of subsetting yet, but you'll learn more about it in "Subsetting" on page 304:

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarize(
    # average delay:
    avg_delay1 = mean(arr_delay),
    # average positive delay:
    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
```

Measures of spread `sd(x)`, `IQR(x)`, `mad(x)`

The mean squared deviation, or standard deviation or sd for short, is the standard measure of spread. The interquartile range `IQR()` and median absolute deviation `mad(x)` are robust equivalents that may be more useful if you have outliers:

```
# Why is distance to some destinations more variable
# than to others?
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
```

Measures of rank `min(x)`, `quantile(x, 0.25)`, `max(x)`

Quantiles are a generalization of the median. For example, `quantile(x, 0.25)` will find a value of `x` that is greater than 25% of the values, and less than the remaining 75%:

```
# When do the first and last flights leave each day?
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
```

Measures of position `first(x)`, `nth(x, 2)`, `last(x)`

These work similarly to `x[1]`, `x[2]`, and `x[length(x)]` but let you set a default value if that position does not exist (i.e., you're trying to get the third element from a group that only has two elements). For example, we can find the first and last departure for each day:

```
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
```

These functions are complementary to filtering on ranks. Filtering gives you all variables, with each observation in a separate row:

```
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>
```

Counts

You've seen `n()`, which takes no arguments, and returns the size of the current group. To count the number of non-missing values, use `sum(!is.na(x))`. To count the number of distinct (unique) values, use `n_distinct(x)`:

```
# Which destinations have the most carriers?
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
```

Counts are so useful that **dplyr** provides a simple helper if all you want is a count:

```
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
```

You can optionally provide a weight variable. For example, you could use this to "count" (sum) the total number of miles a plane flew:

```
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
```

Counts and proportions of logical values `sum(x > 10)`, `mean(y == 0)`

When used with numeric functions, `TRUE` is converted to 1 and `FALSE` to 0. This makes `sum()` and `mean()` very useful: `sum(x)` gives the number of `TRUE`s in `x`, and `mean(x)` gives the proportion:

```
# How many flights left before 5am? (these usually
# indicate delayed flights from the previous day)
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

# What proportion of flights are delayed by more
# than an hour?
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
```

Grouping by Multiple Variables

When you group by multiple variables, each summary peels off one level of the grouping. That makes it easy to progressively roll up a dataset:

```
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
```

Be careful when progressively rolling up summaries: it's OK for sums and counts, but you need to think about weighting means and variances, and it's not possible to do it exactly for rank-based statistics like the median. In other words, the sum of groupwise sums is the overall sum, but the median of groupwise medians is not the overall median.

Ungrouping

If you need to remove grouping, and return to operations on ungrouped data, use `ungroup()`:

```
daily %>%
  ungroup() %>%             # no longer grouped by date
  summarize(flights = n())  # all flights
#> # A tibble: 1 × 1
#>   flights
#>     <int>
#> 1  336776
```

Exercises

1. Brainstorm at least five different ways to assess the typical delay characteristics of a group of flights. Consider the following scenarios:

- A flight is 15 minutes early 50% of the time, and 15 minutes late 50% of the time.
- A flight is always 10 minutes late.
- A flight is 30 minutes early 50% of the time, and 30 minutes late 50% of the time.
- 99% of the time a flight is on time. 1% of the time it's 2 hours late.

Which is more important: arrival delay or departure delay?

2. Come up with another approach that will give you the same output as `not_cancelled %>% count(dest)` and `not_cancelled %>% count(tailnum, wt = distance)` (without using `count()`).
3. Our definition of cancelled flights (`is.na(dep_delay) | is.na(arr_delay)`) is slightly suboptimal. Why? Which is the most important column?
4. Look at the number of cancelled flights per day. Is there a pattern? Is the proportion of cancelled flights related to the average delay?
5. Which carrier has the worst delays? Challenge: can you disentangle the effects of bad airports versus bad carriers? Why/why not? (Hint: think about `flights %>% group_by(carrier, dest) %>% summarize(n())`.)
6. For each plane, count the number of flights before the first delay of greater than 1 hour.
7. What does the `sort` argument to `count()` do? When might you use it?

Grouped Mutates (and Filters)

Grouping is most useful in conjunction with `summarize()`, but you can also do convenient operations with `mutate()` and `filter()`:

- Find the worst members of each group:

```
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>
```

- Find all groups bigger than a threshold:

```
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>
```

- Standardize to compute per group metrics:

```
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
```

A grouped filter is a grouped mutate followed by an ungrouped filter. I generally avoid them except for quick-and-dirty manipulations: otherwise it's hard to check that you've done the manipulation correctly.

Functions that work most naturally in grouped mutates and filters are known as window functions (versus the summary functions used for summaries). You can learn more about useful window functions in the corresponding vignette: `vignette("window-functions")`.

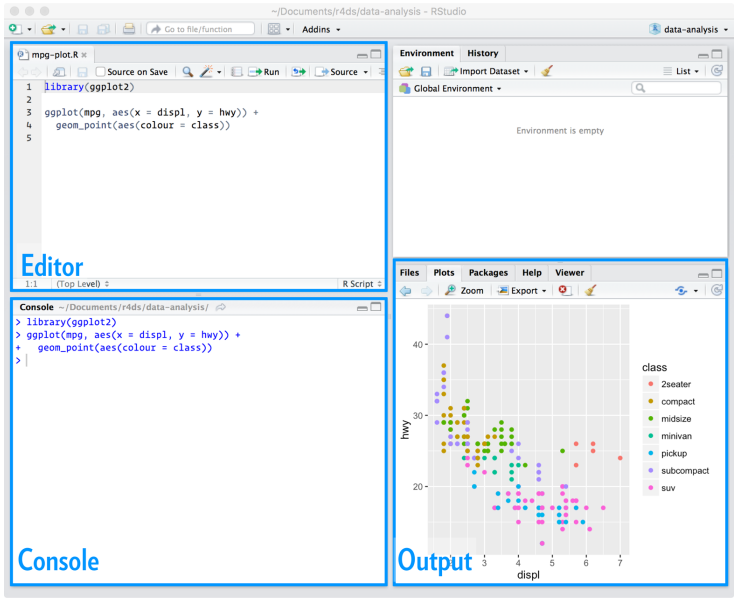
Exercises

1. Refer back to the table of useful mutate and filtering functions. Describe how each operation changes when you combine it with grouping.
2. Which plane (`tailnum`) has the worst on-time record?
3. What time of day should you fly if you want to avoid delays as much as possible?
4. For each destination, compute the total minutes of delay. For each flight, compute the proportion of the total delay for its destination.
5. Delays are typically temporally correlated: even once the problem that caused the initial delay has been resolved, later flights are delayed to allow earlier flights to leave. Using `lag()` explores how the delay of a flight is related to the delay of the immediately preceding flight.
6. Look at each destination. Can you find flights that are suspiciously fast? (That is, flights that represent a potential data entry error.) Compute the air time of a flight relative to the shortest flight to that destination. Which flights were most delayed in the air?

7. Find all destinations that are flown by at least two carriers. Use that information to rank the carriers.

CHAPTER 4
Workflow: Scripts

So far you've been using the console to run code. That's a great place to start, but you'll find it gets cramped pretty quickly as you create more complex **ggplot2** graphics and **dplyr** pipes. To give yourself more room to work, it's a great idea to use the script editor. Open it up either by clicking the File menu and selecting New File, then R script, or using the keyboard shortcut Cmd/Ctrl-Shift-N. Now you'll see four panes:



The script editor is a great place to put code you care about. Keep experimenting in the console, but once you have written code that works and does what you want, put it in the script editor. RStudio will automatically save the contents of the editor when you quit RStudio, and will automatically load it when you reopen. Nevertheless, it's a good idea to save your scripts regularly and to back them up.

Running Code

The script editor is also a great place to build up complex **ggplot2** plots or long sequences of **dplyr** manipulations. The key to using the script editor effectively is to memorize one of the most important keyboard shortcuts: Cmd/Ctrl-Enter. This executes the current R expression in the console. For example, take the following code. If your cursor is at █, pressing Cmd/Ctrl-Enter will run the complete command that generates `not_cancelled`. It will also move the cursor to the next statement (beginning with `not_cancelled %>%`). That makes it easy to run your complete script by repeatedly pressing Cmd/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))
```

Instead of running expression-by-expression, you can also execute the complete script in one step: Cmd/Ctrl-Shift-S. Doing this regularly is a great way to check that you've captured all the important parts of your code in the script.

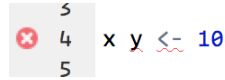
I recommend that you always start your script with the packages that you need. That way, if you share your code with others, they can easily see what packages they need to install. Note, however, that you should never include `install.packages()` or `setwd()` in a script that you share. It's very antisocial to change settings on someone else's computer!

When working through future chapters, I highly recommend starting in the editor and practicing your keyboard shortcuts. Over time,

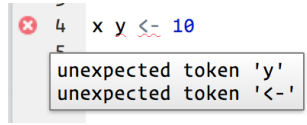
sending code to the console in this way will become so natural that you won't even think about it.

RStudio Diagnostics

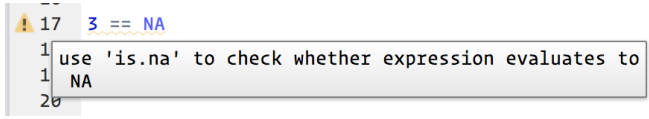
The script editor will also highlight syntax errors with a red squiggly line and a cross in the sidebar:



Hover over the cross to see what the problem is:



RStudio will also let you know about potential problems:



Exercises

1. Go to the RStudio Tips twitter account at *@rstudiotips* and find one tip that looks interesting. Practice using it!
2. What other common mistakes will RStudio diagnostics report? Read *http://bit.ly/RStudiocodediag* to find out.

CHAPTER 5

Exploratory Data Analysis

Introduction

This chapter will show you how to use visualization and transformation to explore your data in a systematic way, a task that statisticians call exploratory data analysis, or EDA for short. EDA is an iterative cycle. You:

1. Generate questions about your data.
2. Search for answers by visualizing, transforming, and modeling your data.
3. Use what you learn to refine your questions and/or generate new questions.

EDA is not a formal process with a strict set of rules. More than anything, EDA is a state of mind. During the initial phases of EDA you should feel free to investigate every idea that occurs to you. Some of these ideas will pan out, and some will be dead ends. As your exploration continues, you will hone in on a few particularly productive areas that you'll eventually write up and communicate to others.

EDA is an important part of any data analysis, even if the questions are handed to you on a platter, because you always need to investigate the quality of your data. Data cleaning is just one application of EDA: you ask questions about whether or not your data meets your expectations. To do data cleaning, you'll need to deploy all the tools of EDA: visualization, transformation, and modeling.

Prerequisites

In this chapter we'll combine what you've learned about **dplyr** and **ggplot2** to interactively ask questions, answer them with data, and then ask new questions.

```
library(tidyverse)
```

Questions

> There are no routine statistical questions, only questionable statistical routines.
>
> —Sir David Cox

> Far better an approximate answer to the right question, which is often vague, than an exact answer to the wrong question, which can always be made precise.
>
> —John Tukey

Your goal during EDA is to develop an understanding of your data. The easiest way to do this is to use questions as tools to guide your investigation. When you ask a question, the question focuses your attention on a specific part of your dataset and helps you decide which graphs, models, or transformations to make.

EDA is fundamentally a creative process. And like most creative processes, the key to asking *quality* questions is to generate a large *quantity* of questions. It is difficult to ask revealing questions at the start of your analysis because you do not know what insights are contained in your dataset. On the other hand, each new question that you ask will expose you to a new aspect of your data and increase your chance of making a discovery. You can quickly drill down into the most interesting parts of your data—and develop a set of thought-provoking questions—if you follow up each question with a new question based on what you find.

There is no rule about which questions you should ask to guide your research. However, two types of questions will always be useful for making discoveries within your data. You can loosely word these questions as:

1. What type of variation occurs within my variables?
2. What type of covariation occurs between my variables?

The rest of this chapter will look at these two questions. I'll explain what variation and covariation are, and I'll show you several ways to answer each question. To make the discussion easier, let's define some terms:

- A *variable* is a quantity, quality, or property that you can measure.
- A *value* is the state of a variable when you measure it. The value of a variable may change from measurement to measurement.
- An *observation*, or a *case*, is a set of measurements made under similar conditions (you usually make all of the measurements in an observation at the same time and on the same object). An observation will contain several values, each associated with a different variable. I'll sometimes refer to an observation as a data point.
- *Tabular data* is a set of values, each associated with a variable and an observation. Tabular data is *tidy* if each value is placed in its own "cell," each variable in its own column, and each observation in its own row.

So far, all of the data that you've seen has been tidy. In real life, most data isn't tidy, so we'll come back to these ideas again in Chapter 9.

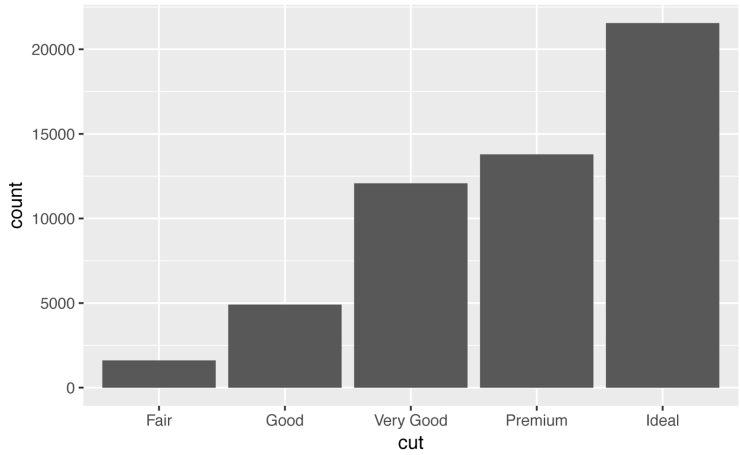
Variation

Variation is the tendency of the values of a variable to change from measurement to measurement. You can see variation easily in real life; if you measure any continuous variable twice, you will get two different results. This is true even if you measure quantities that are constant, like the speed of light. Each of your measurements will include a small amount of error that varies from measurement to measurement. Categorical variables can also vary if you measure across different subjects (e.g., the eye colors of different people), or different times (e.g., the energy levels of an electron at different moments). Every variable has its own pattern of variation, which can reveal interesting information. The best way to understand that pattern is to visualize the distribution of variables' values.

Visualizing Distributions

How you visualize the distribution of a variable will depend on whether the variable is categorical or continuous. A variable is *categorical* if it can only take one of a small set of values. In R, categorical variables are usually saved as factors or character vectors. To examine the distribution of a categorical variable, use a bar chart:

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

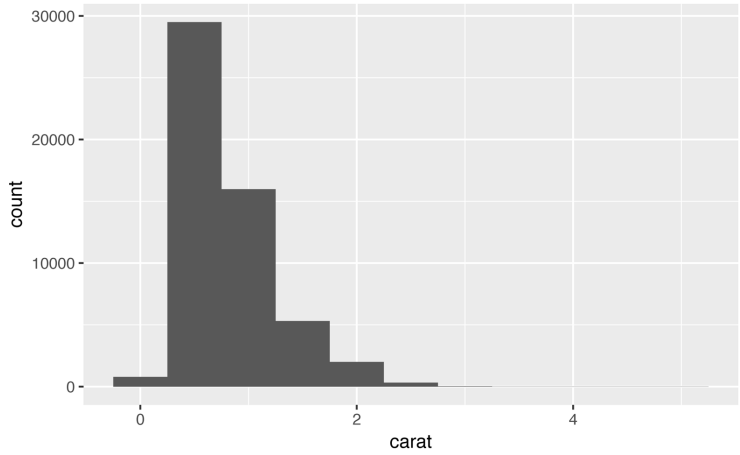


The height of the bars displays how many observations occurred with each x value. You can compute these values manually with `dplyr::count()`:

```
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
```

A variable is *continuous* if it can take any of an infinite set of ordered values. Numbers and date-times are two examples of continuous variables. To examine the distribution of a continuous variable, use a histogram:

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



You can compute this by hand by combining `dplyr::count()` and `ggplot2::cut_width()`:

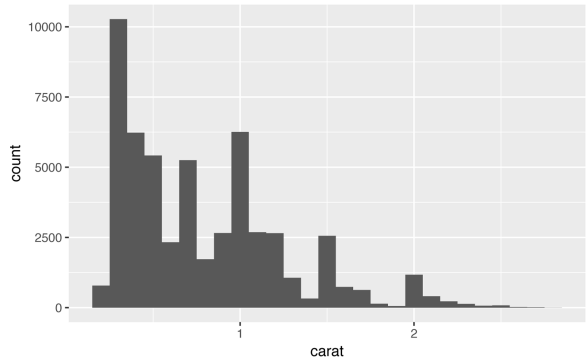
```
diamonds %>%
  count(cut_width(carat, 0.5))
#> # A tibble: 11 × 2
#>   `cut_width(carat, 0.5)`     n
#>                    <fctr> <int>
#> 1           [-0.25,0.25]   785
#> 2            (0.25,0.75] 29498
#> 3            (0.75,1.25] 15977
#> 4            (1.25,1.75]  5313
#> 5            (1.75,2.25]  2002
#> 6            (2.25,2.75]   322
#> # ... with 5 more rows
```

A histogram divides the x-axis into equally spaced bins and then uses the height of each bar to display the number of observations that fall in each bin. In the preceding graph, the tallest bar shows that almost 30,000 observations have a `carat` value between 0.25 and 0.75, which are the left and right edges of the bar.

You can set the width of the intervals in a histogram with the `binwidth` argument, which is measured in the units of the `x` variable. You should always explore a variety of binwidths when working with histograms, as different binwidths can reveal different patterns. For example, here is how the preceding graph looks when we zoom

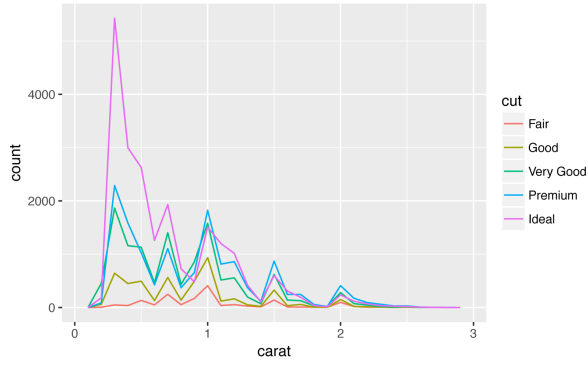
into just the diamonds with a size of less than three carats and choose a smaller binwidth:

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

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



If you wish to overlay multiple histograms in the same plot, I recommend using `geom_freqpoly()` instead of `geom_histogram()`. `geom_freqpoly()` performs the same calculation as `geom_histogram()`, but instead of displaying the counts with bars, uses lines instead. It's much easier to understand overlapping lines than bars:

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



There are a few challenges with this type of plot, which we will come back to in "A Categorical and Continuous Variable" on page 93.

Typical Values

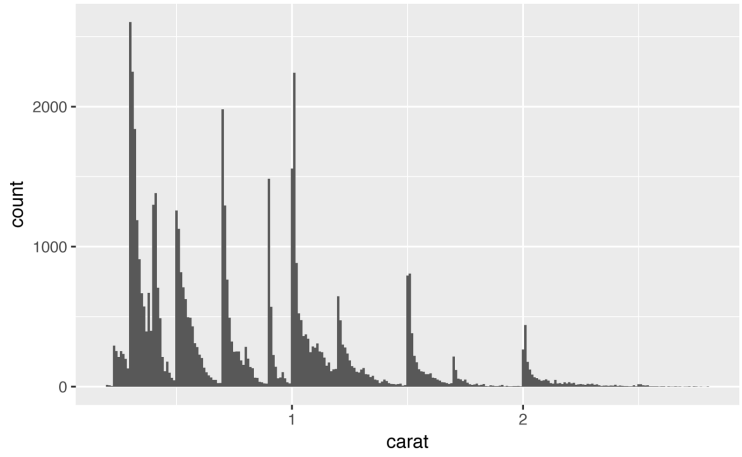
In both bar charts and histograms, tall bars show the common values of a variable, and shorter bars show less-common values. Places that do not have bars reveal values that were not seen in your data. To turn this information into useful questions, look for anything unexpected:

- Which values are the most common? Why?
- Which values are rare? Why? Does that match your expectations?
- Can you see any unusual patterns? What might explain them?

As an example, the following histogram suggests several interesting questions:

- Why are there more diamonds at whole carats and common fractions of carats?
- Why are there more diamonds slightly to the right of each peak than there are slightly to the left of each peak?
- Why are there no diamonds bigger than 3 carats?

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

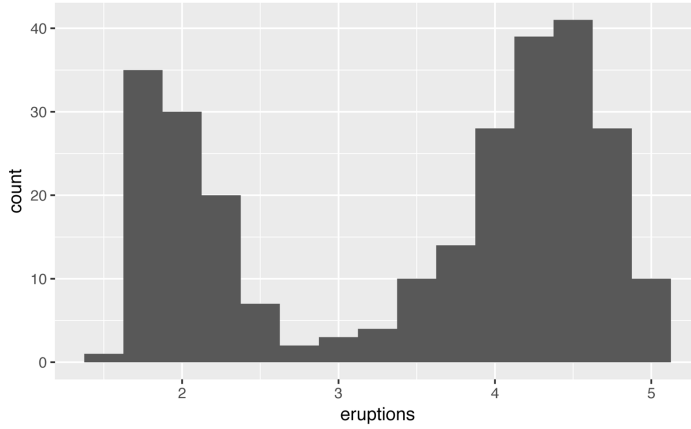


In general, clusters of similar values suggest that subgroups exist in your data. To understand the subgroups, ask:

- How are the observations within each cluster similar to each other?
- How are the observations in separate clusters different from each other?
- How can you explain or describe the clusters?
- Why might the appearance of clusters be misleading?

The following histogram shows the length (in minutes) of 272 eruptions of the Old Faithful Geyser in Yellowstone National Park. Eruption times appear to be clustered into two groups: there are short eruptions (of around 2 minutes) and long eruptions (4–5 minutes), but little in between:

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



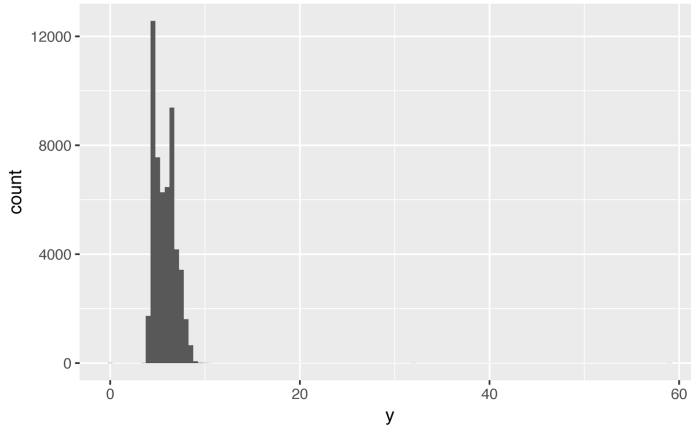
Many of the preceding questions will prompt you to explore a relationship *between* variables, for example, to see if the values of one variable can explain the behavior of another variable. We'll get to that shortly.

Unusual Values

Outliers are observations that are unusual; data points that don't seem to fit the pattern. Sometimes outliers are data entry errors; other times outliers suggest important new science. When you have a lot of data, outliers are sometimes difficult to see in a histogram. For example, take the distribution of the `y` variable from the dia-

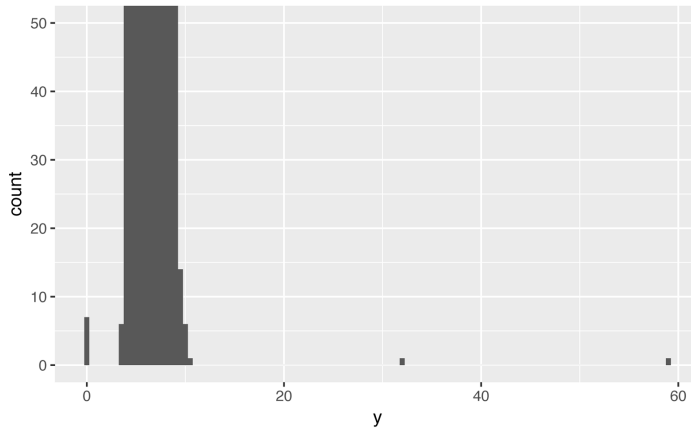
monds dataset. The only evidence of outliers is the unusually wide limits on the y-axis:

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



There are so many observations in the common bins that the rare bins are so short that you can't see them (although maybe if you stare intently at 0 you'll spot something). To make it easy to see the unusual values, we need to zoom in to small values of the y-axis with `coord_cartesian()`:

```
ggplot(diamonds) +
  geom_histogram(mapping = aes(x = y), binwidth = 0.5) +
  coord_cartesian(ylim = c(0, 50))
```



(`coord_cartesian()` also has an `xlim()` argument for when you need to zoom into the x-axis. **ggplot2** also has `xlim()` and `ylim()` functions that work slightly differently: they throw away the data outside the limits.)

This allows us to see that there are three unusual values: 0, ~30, and ~60. We pluck them out with **dplyr**:

```
unusual <- diamonds %>%
  filter(y < 3 | y > 20) %>%
  arrange(y)
unusual
#> # A tibble: 9 × 10
#>   carat       cut color clarity depth table price     x
#>   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl>
#> 1  1.00 Very Good     H     VS2  63.3    53  5139  0.00
#> 2  1.14      Fair     G     VS1  57.5    67  6381  0.00
#> 3  1.56     Ideal     G     VS2  62.2    54 12800  0.00
#> 4  1.20   Premium     D    VVS1  62.1    59 15686  0.00
#> 5  2.25   Premium     H     SI2  62.8    59 18034  0.00
#> 6  0.71      Good     F     SI2  64.1    60  2130  0.00
#> 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>
```

The `y` variable measures one of the three dimensions of these diamonds, in mm. We know that diamonds can't have a width of 0mm, so these values must be incorrect. We might also suspect that measurements of 32mm and 59mm are implausible: those diamonds are over an inch long, but don't cost hundreds of thousands of dollars!

It's good practice to repeat your analysis with and without the outliers. If they have minimal effect on the results, and you can't figure out why they're there, it's reasonable to replace them with missing values and move on. However, if they have a substantial effect on your results, you shouldn't drop them without justification. You'll need to figure out what caused them (e.g., a data entry error) and disclose that you removed them in your write-up.

Exercises

1. Explore the distribution of each of the `x`, `y`, and `z` variables in `diamonds`. What do you learn? Think about a diamond and how you might decide which dimension is the length, width, and depth.

2. Explore the distribution of `price`. Do you discover anything unusual or surprising? (Hint: carefully think about the `binwidth` and make sure you try a wide range of values.)
3. How many diamonds are 0.99 carat? How many are 1 carat? What do you think is the cause of the difference?
4. Compare and contrast `coord_cartesian()` versus `xlim()` or `ylim()` when zooming in on a histogram. What happens if you leave `binwidth` unset? What happens if you try and zoom so only half a bar shows?

Missing Values

If you've encountered unusual values in your dataset, and simply want to move on to the rest of your analysis, you have two options:

- Drop the entire row with the strange values:

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

  I don't recommend this option as just because one measurement is invalid, doesn't mean all the measurements are. Additionally, if you have low-quality data, by time that you've applied this approach to every variable you might find that you don't have any data left!

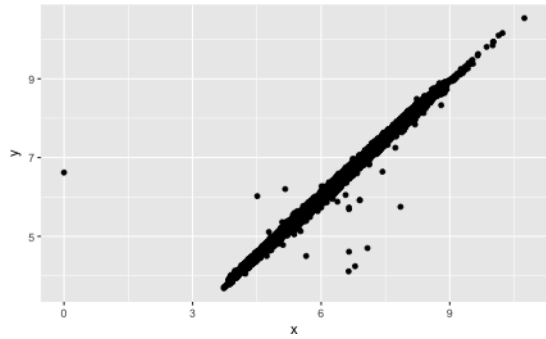
- Instead, I recommend replacing the unusual values with missing values. The easiest way to do this is to use `mutate()` to replace the variable with a modified copy. You can use the `ifelse()` function to replace unusual values with `NA`:

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

`ifelse()` has three arguments. The first argument `test` should be a logical vector. The result will contain the value of the second argument, `yes`, when `test` is `TRUE`, and the value of the third argument, `no`, when it is false.

Like R, **ggplot2** subscribes to the philosophy that missing values should never silently go missing. It's not obvious where you should plot missing values, so **ggplot2** doesn't include them in the plot, but it does warn that they've been removed:

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

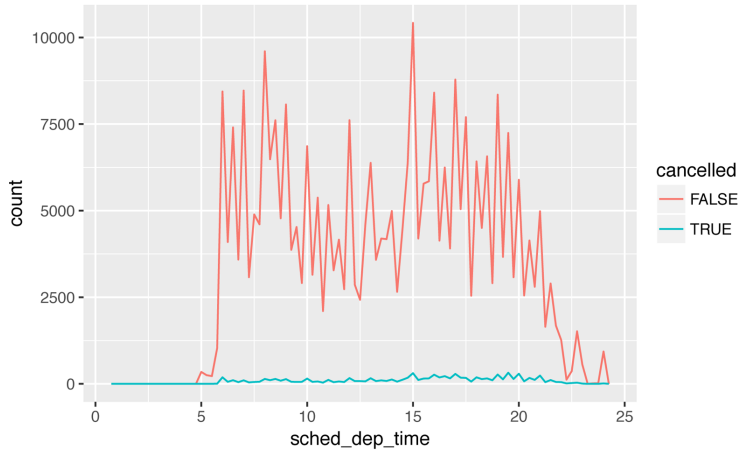


To suppress that warning, set `na.rm = TRUE`:

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

Other times you want to understand what makes observations with missing values different from observations with recorded values. For example, in `nycflights13::flights`, missing values in the `dep_time` variable indicate that the flight was cancelled. So you might want to compare the scheduled departure times for cancelled and noncancelled times. You can do this by making a new variable with `is.na()`:

```
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
    )
```



However, this plot isn't great because there are many more non-cancelled flights than cancelled flights. In the next section we'll explore some techniques for improving this comparison.

Exercises

1. What happens to missing values in a histogram? What happens to missing values in a bar chart? Why is there a difference?
2. What does `na.rm = TRUE` do in `mean()` and `sum()`?

Covariation

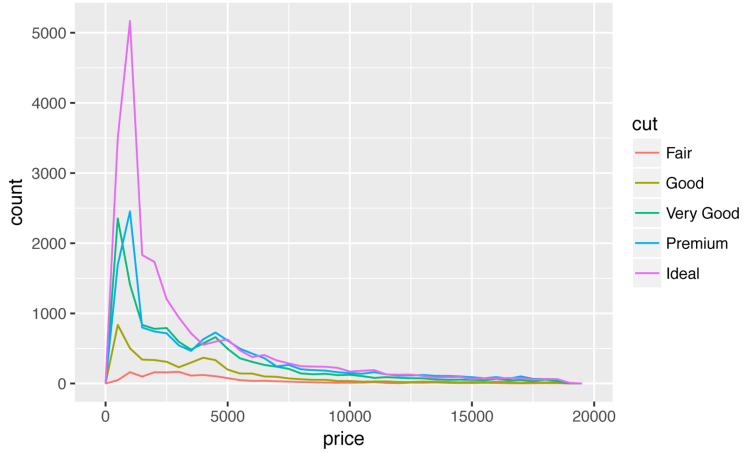
If variation describes the behavior *within* a variable, covariation describes the behavior *between* variables. *Covariation* is the tendency for the values of two or more variables to vary together in a related way. The best way to spot covariation is to visualize the relationship between two or more variables. How you do that should again depend on the type of variables involved.

A Categorical and Continuous Variable

It's common to want to explore the distribution of a continuous variable broken down by a categorical variable, as in the previous frequency polygon. The default appearance of `geom_freqpoly()` is not that useful for that sort of comparison because the height is

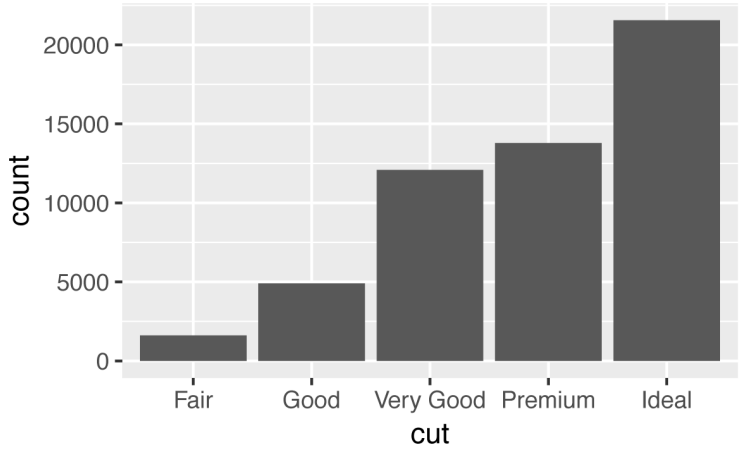
given by the count. That means if one of the groups is much smaller than the others, it's hard to see the differences in shape. For example, let's explore how the price of a diamond varies with its quality:

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



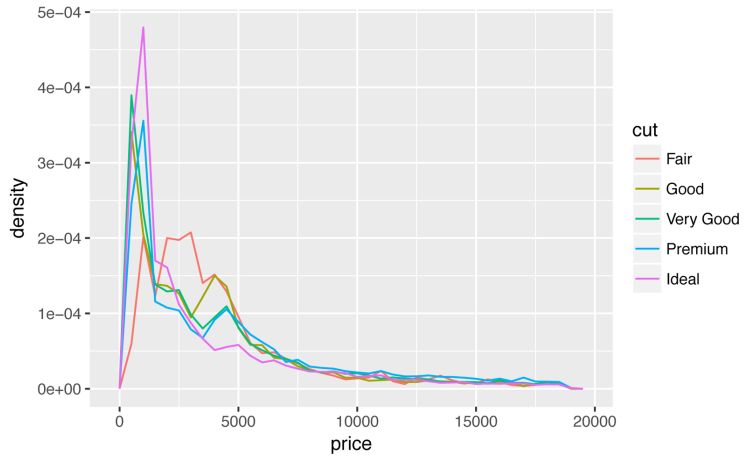
It's hard to see the difference in distribution because the overall counts differ so much:

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



To make the comparison easier we need to swap what is displayed on the y-axis. Instead of displaying count, we'll display *density*, which is the count standardized so that the area under each frequency polygon is one:

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

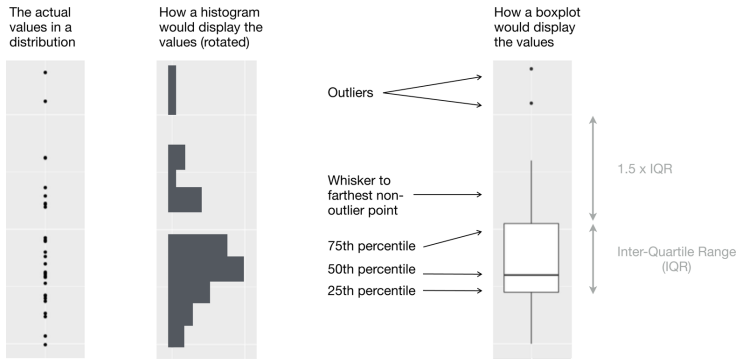


There's something rather surprising about this plot—it appears that fair diamonds (the lowest quality) have the highest average price! But maybe that's because frequency polygons are a little hard to interpret—there's a lot going on in this plot.

Another alternative to display the distribution of a continuous variable broken down by a categorical variable is the boxplot. A *boxplot* is a type of visual shorthand for a distribution of values that is popular among statisticians. Each boxplot consists of:

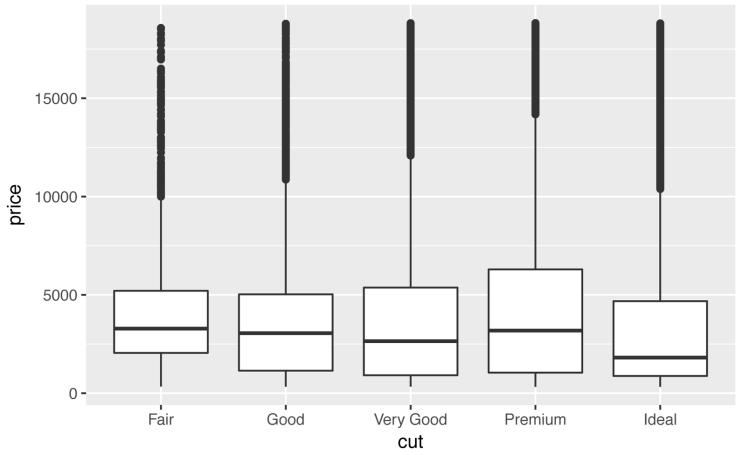
- A box that stretches from the 25th percentile of the distribution to the 75th percentile, a distance known as the interquartile range (IQR). In the middle of the box is a line that displays the median, i.e., 50th percentile, of the distribution. These three lines give you a sense of the spread of the distribution and whether or not the distribution is symmetric about the median or skewed to one side.

- Visual points that display observations that fall more than 1.5 times the IQR from either edge of the box. These outlying points are unusual, so they are plotted individually.
- A line (or whisker) that extends from each end of the box and goes to the farthest nonoutlier point in the distribution.



Let's take a look at the distribution of price by cut using `geom_boxplot()`:

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



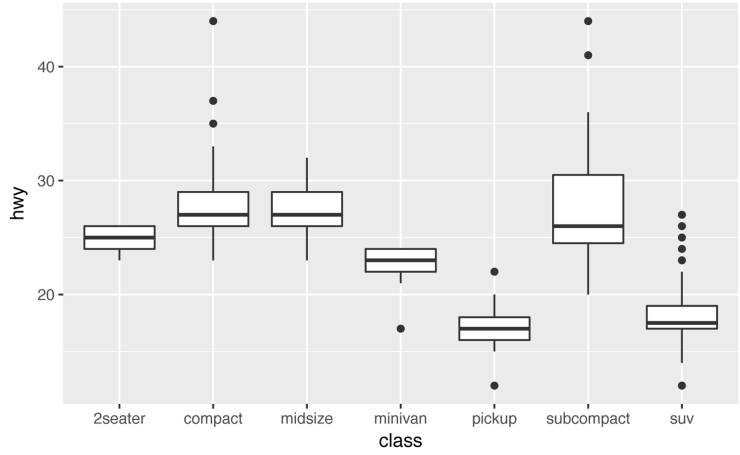
We see much less information about the distribution, but the boxplots are much more compact so we can more easily compare them (and fit more on one plot). It supports the counterintuitive finding

that better quality diamonds are cheaper on average! In the exercises, you'll be challenged to figure out why.

`cut` is an ordered factor: fair is worse than good, which is worse than very good, and so on. Many categorical variables don't have such an intrinsic order, so you might want to reorder them to make a more informative display. One way to do that is with the `reorder()` function.

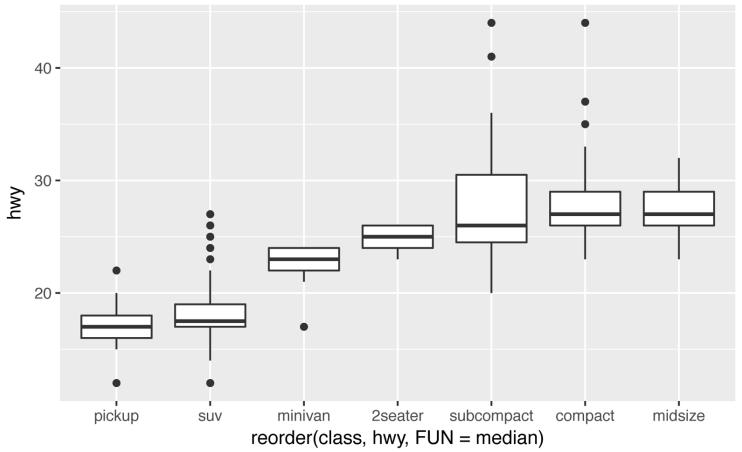
For example, take the `class` variable in the `mpg` dataset. You might be interested to know how highway mileage varies across classes:

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



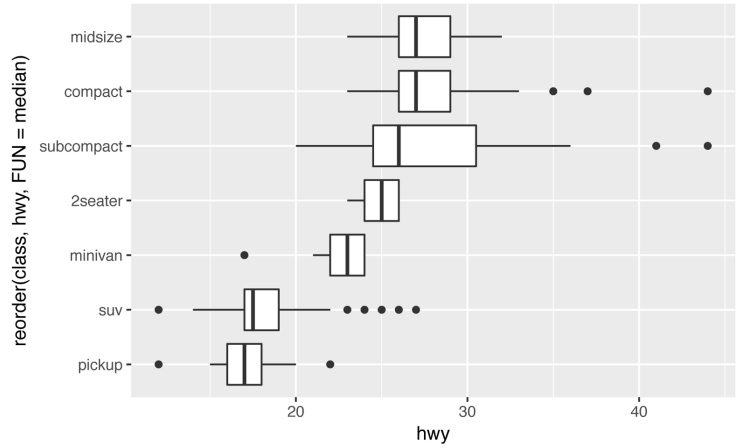
To make the trend easier to see, we can reorder `class` based on the median value of `hwy`:

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



If you have long variable names, `geom_boxplot()` will work better if you flip it 90°. You can do that with `coord_flip()`:

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



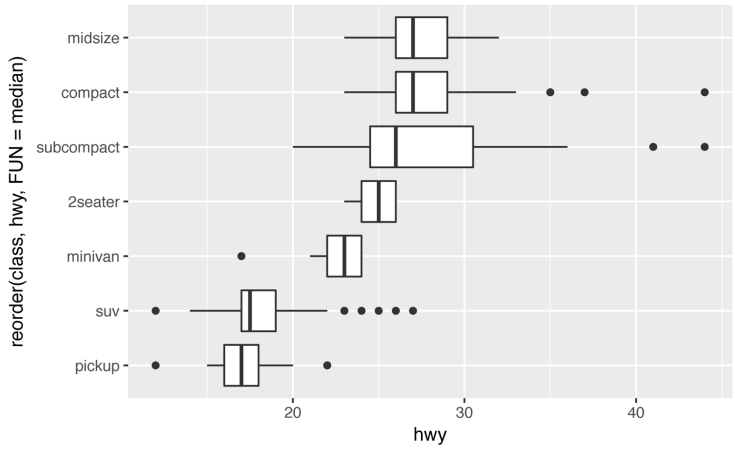
Exercises

1. Use what you've learned to improve the visualization of the departure times of cancelled versus noncancelled flights.
2. What variable in the diamonds dataset is most important for predicting the price of a diamond? How is that variable correlated with cut? Why does the combination of those two relationships lead to lower quality diamonds being more expensive?
3. Install the **ggstance** package, and create a horizontal boxplot. How does this compare to using `coord_flip()`?
4. One problem with boxplots is that they were developed in an era of much smaller datasets and tend to display a prohibitively large number of "outlying values." One approach to remedy this problem is the letter value plot. Install the **lvplot** package, and try using `geom_lv()` to display the distribution of price versus cut. What do you learn? How do you interpret the plots?
5. Compare and contrast `geom_violin()` with a faceted `geom_histogram()`, or a colored `geom_freqpoly()`. What are the pros and cons of each method?
6. If you have a small dataset, it's sometimes useful to use `geom_jitter()` to see the relationship between a continuous and categorical variable. The **ggbeeswarm** package provides a number of methods similar to `geom_jitter()`. List them and briefly describe what each one does.

Two Categorical Variables

To visualize the covariation between categorical variables, you'll need to count the number of observations for each combination. One way to do that is to rely on the built-in `geom_count()`:

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



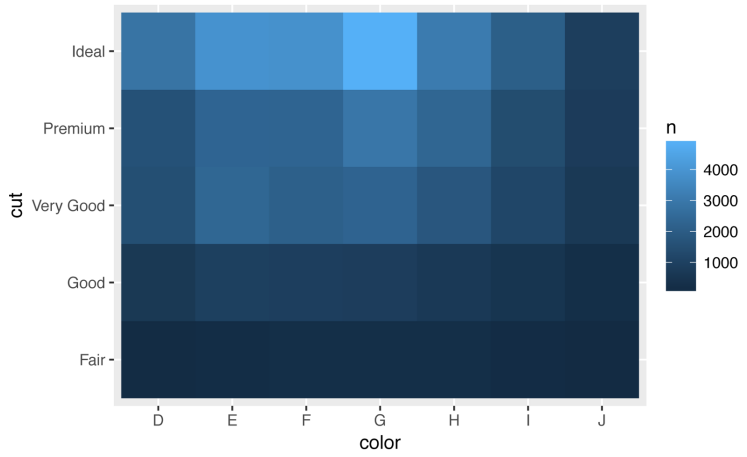
The size of each circle in the plot displays how many observations occurred at each combination of values. Covariation will appear as a strong correlation between specific `x` values and specific `y` values.

Another approach is to compute the count with **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
```

Then visualize with `geom_tile()` and the fill aesthetic:

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



If the categorical variables are unordered, you might want to use the **seriation** package to simultaneously reorder the rows and columns in order to more clearly reveal interesting patterns. For larger plots, you might want to try the **d3heatmap** or **heatmaply** packages, which create interactive plots.

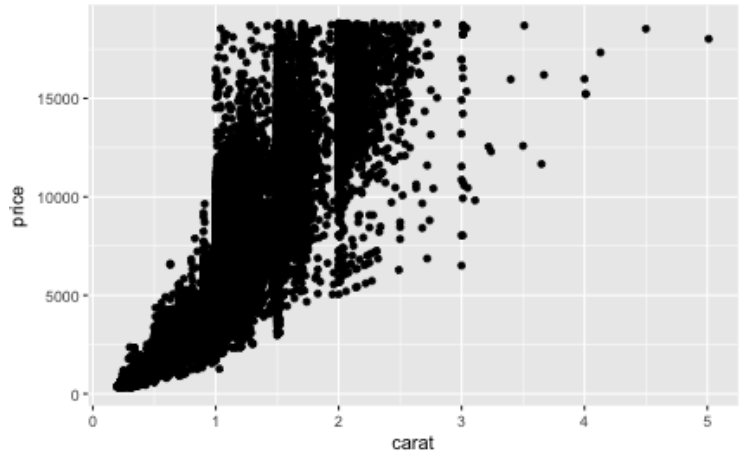
Exercises

1. How could you rescale the `count` dataset to more clearly show the distribution of cut within color, or color within cut?
2. Use `geom_tile()` together with **dplyr** to explore how average flight delays vary by destination and month of year. What makes the plot difficult to read? How could you improve it?
3. Why is it slightly better to use `aes(x = color, y = cut)` rather than `aes(x = cut, y = color)` in the previous example?

Two Continuous Variables

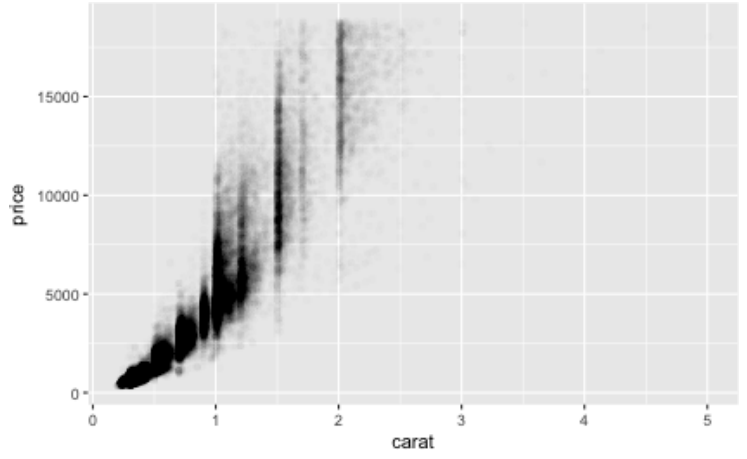
You've already seen one great way to visualize the covariation between two continuous variables: draw a scatterplot with `geom_point()`. You can see covariation as a pattern in the points. For example, you can see an exponential relationship between the carat size and price of a diamond:

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



Scatterplots become less useful as the size of your dataset grows, because points begin to overplot, and pile up into areas of uniform black (as in the preceding scatterplot). You've already seen one way to fix the problem, using the `alpha` aesthetic to add transparency:

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



But using transparency can be challenging for very large datasets. Another solution is to use bin. Previously you used `geom_histo`

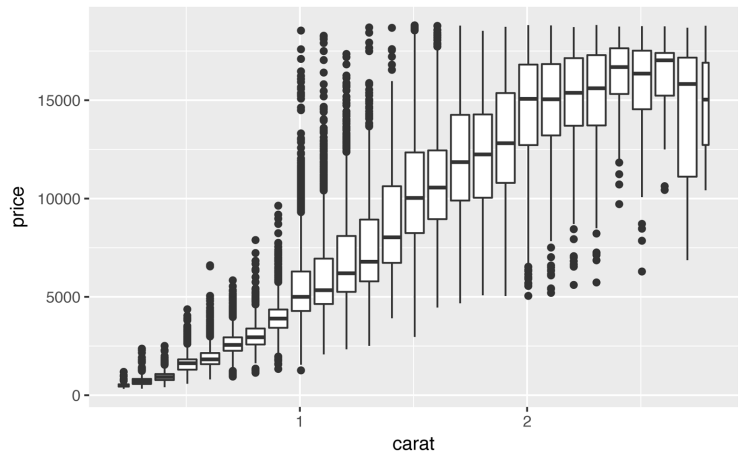
gram() and geom_freqpoly() to bin in one dimension. Now you'll learn how to use geom_bin2d() and geom_hex() to bin in two dimensions.

geom_bin2d() and geom_hex() divide the coordinate plane into 2D bins and then use a fill color to display how many points fall into each bin. geom_bin2d() creates rectangular bins. geom_hex() creates hexagonal bins. You will need to install the **hexbin** package to use geom_hex():

```
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
```

Another option is to bin one continuous variable so it acts like a categorical variable. Then you can use one of the techniques for visualizing the combination of a categorical and a continuous variable that you learned about. For example, you could bin carat and then for each group, display a boxplot:

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

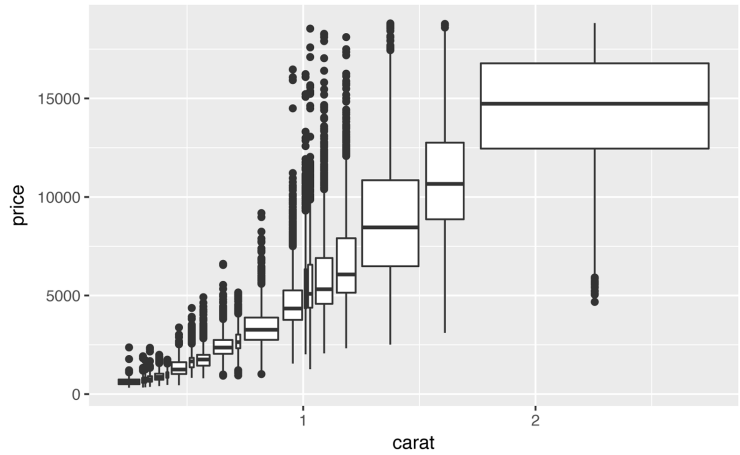


cut_width(x, width), as used here, divides x into bins of width width. By default, boxplots look roughly the same (apart from the number of outliers) regardless of how many observations there are,

so it's difficult to tell that each boxplot summarizes a different number of points. One way to show that is to make the width of the boxplot proportional to the number of points with `varwidth = TRUE`.

Another approach is to display approximately the same number of points in each bin. That's the job of `cut_number()`:

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

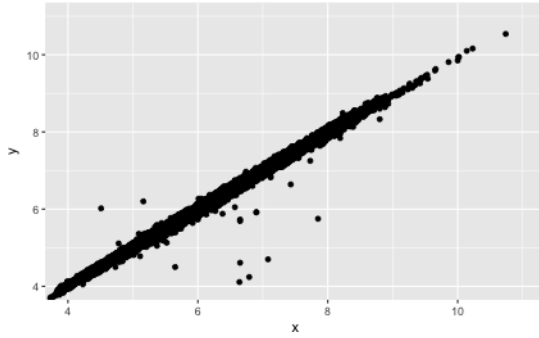


Exercises

1. Instead of summarizing the conditional distribution with a boxplot, you could use a frequency polygon. What do you need to consider when using `cut_width()` versus `cut_number()`? How does that impact a visualization of the 2D distribution of `carat` and `price`?
2. Visualize the distribution of `carat`, partitioned by `price`.
3. How does the price distribution of very large diamonds compare to small diamonds. Is it as you expect, or does it surprise you?
4. Combine two of the techniques you've learned to visualize the combined distribution of `cut`, `carat`, and `price`.
5. Two-dimensional plots reveal outliers that are not visible in one-dimensional plots. For example, some points in the following plot have an unusual combination of `x` and `y` values, which

makes the points outliers even though their x and y values appear normal when examined separately:

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



Why is a scatterplot a better display than a binned plot for this case?

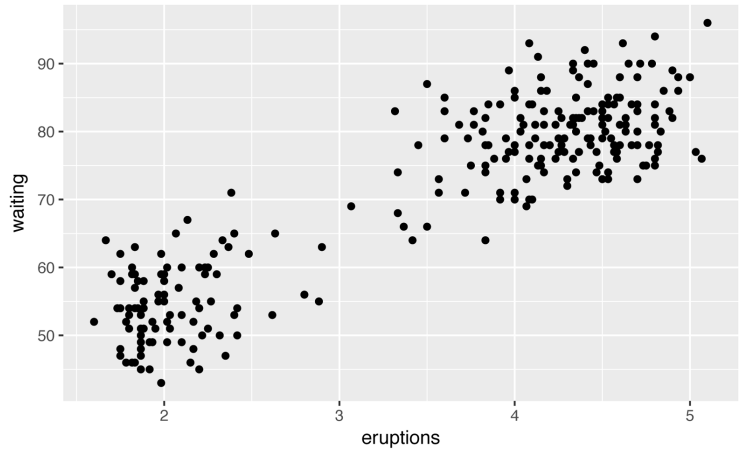
Patterns and Models

Patterns in your data provide clues about relationships. If a systematic relationship exists between two variables it will appear as a pattern in the data. If you spot a pattern, ask yourself:

- Could this pattern be due to coincidence (i.e., random chance)?
- How can you describe the relationship implied by the pattern?
- How strong is the relationship implied by the pattern?
- What other variables might affect the relationship?
- Does the relationship change if you look at individual subgroups of the data?

A scatterplot of Old Faithful eruption lengths versus the wait time between eruptions shows a pattern: longer wait times are associated with longer eruptions. The scatterplot also displays the two clusters that we noticed earlier:

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

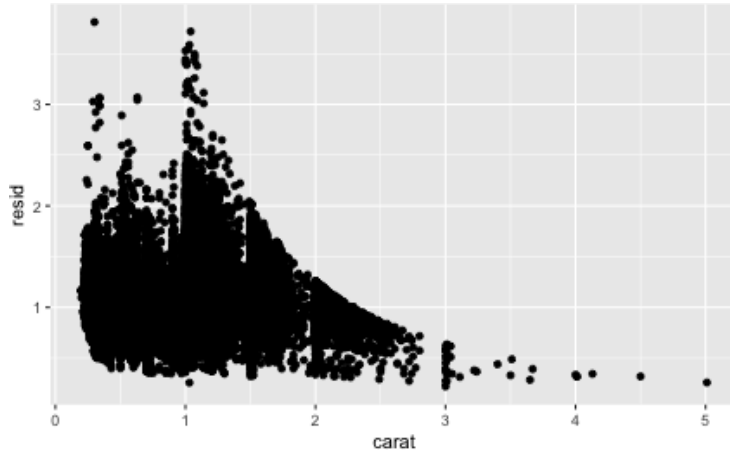


Patterns provide one of the most useful tools for data scientists because they reveal covariation. If you think of variation as a phenomenon that creates uncertainty, covariation is a phenomenon that reduces it. If two variables covary, you can use the values of one variable to make better predictions about the values of the second. If the covariation is due to a causal relationship (a special case), then you can use the value of one variable to control the value of the second.

Models are a tool for extracting patterns out of data. For example, consider the diamonds data. It's hard to understand the relationship between cut and price, because cut and carat, and carat and price, are tightly related. It's possible to use a model to remove the very strong relationship between price and carat so we can explore the subtleties that remain. The following code fits a model that predicts `price` from `carat` and then computes the residuals (the difference between the predicted value and the actual value). The residuals give us a view of the price of the diamond, once the effect of carat has been removed:

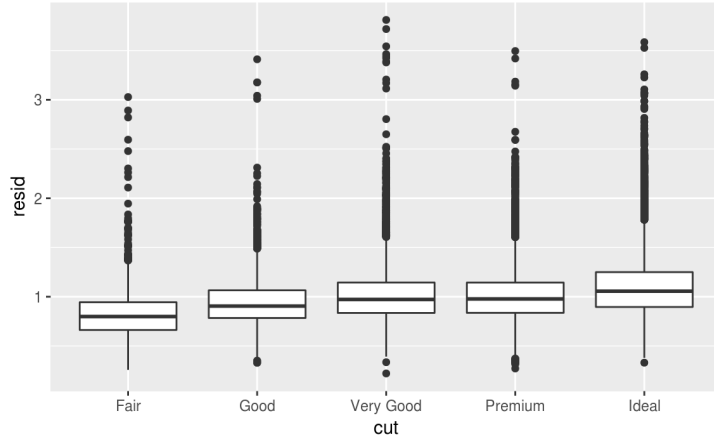
```
library(modelr)

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))
```



Once you've removed the strong relationship between carat and price, you can see what you expect in the relationship between cut and price—relative to their size, better quality diamonds are more expensive:

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



You'll learn how models, and the **modelr** package, work in the final part of the book, Part IV. We're saving modeling for later because

understanding what models are and how they work is easiest once you have the tools of data wrangling and programming in hand.

ggplot2 Calls

As we move on from these introductory chapters, we'll transition to a more concise expression of **ggplot2** code. So far we've been very explicit, which is helpful when you are learning:

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

Typically, the first one or two arguments to a function are so important that you should know them by heart. The first two arguments to `ggplot()` are `data` and `mapping`, and the first two arguments to `aes()` are `x` and `y`. In the remainder of the book, we won't supply those names. That saves typing, and, by reducing the amount of boilerplate, makes it easier to see what's different between plots. That's a really important programming concern that we'll come back to in Chapter 15.

Rewriting the previous plot more concisely yields:

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

Sometimes we'll turn the end of a pipeline of data transformation into a plot. Watch for the transition from `%>%` to `+`. I wish this transition wasn't necessary but unfortunately **ggplot2** was created before the pipe was discovered:

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

Learning More

If you want learn more about the mechanics of **ggplot2**, I'd highly recommend grabbing a copy of the **ggplot2** book. It's been recently updated, so it includes **dplyr** and **tidyr** code, and has much more space to explore all the facets of visualization. Unfortunately the book isn't generally available for free, but if you have a connection to a university you can probably get an electronic version for free through SpringerLink.

Another useful resource is the *R Graphics Cookbook* by Winston Chang. Much of the contents are available online at *http://www.cookbook-r.com/Graphs/*.

I also recommend *Graphical Data Analysis with R*, by Antony Unwin. This is a book-length treatment similar to the material covered in this chapter, but has the space to go into much greater depth.

CHAPTER 6

Workflow: Projects

One day you will need to quit R, go do something else, and return to your analysis the next day. One day you will be working on multiple analyses simultaneously that all use R and you want to keep them separate. One day you will need to bring data from the outside world into R and send numerical results and figures from R back out into the world. To handle these real-life situations, you need to make two decisions:

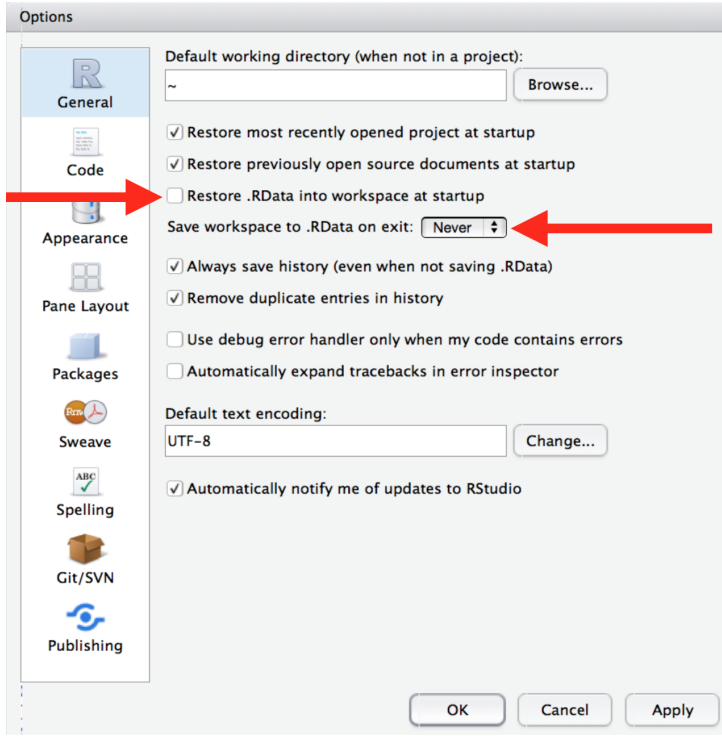
1. What about your analysis is "real," i.e., what will you save as your lasting record of what happened?
2. Where does your analysis "live"?

What Is Real?

As a beginning R user, it's OK to consider your environment (i.e., the objects listed in the environment pane) "real." However, in the long run, you'll be much better off if you consider your R scripts as "real."

With your R scripts (and your data files), you can re-create the environment. It's much harder to re-create your R scripts from your environment! You'll either have to retype a lot of code from memory (making mistakes all the way) or you'll have to carefully mine your R history.

To foster this behavior, I highly recommend that you instruct RStudio not to preserve your workspace between sessions:



This will cause you some short-term pain, because now when you restart RStudio it will not remember the results of the code that you ran last time. But this short-term pain will save you long-term agony because it forces you to capture all important interactions in your code. There's nothing worse than discovering three months after the fact that you've only stored the results of an important calculation in your workspace, not the calculation itself in your code.

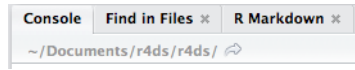
There is a great pair of keyboard shortcuts that will work together to make sure you've captured the important parts of your code in the editor:

- Press Cmd/Ctrl-Shift-F10 to restart RStudio.
- Press Cmd/Ctrl-Shift-S to rerun the current script.

I use this pattern hundreds of times a week.

Where Does Your Analysis Live?

R has a powerful notion of the *working directory*. This is where R looks for files that you ask it to load, and where it will put any files that you ask it to save. RStudio shows your current working directory at the top of the console:



And you can print this out in R code by running `getwd()`:

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

As a beginning R user, it's OK to let your home directory, documents directory, or any other weird directory on your computer be R's working directory. But you're six chapters into this book, and you're no longer a rank beginner. Very soon now you should evolve to organizing your analytical projects into directories and, when working on a project, setting R's working directory to the associated directory.

I do not recommend it, but you can also set the working directory from within R:

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

But you should never do this because there's a better way; a way that also puts you on the path to managing your R work like an expert.

Paths and Directories

Paths and directories are a little complicated because there are two basic styles of paths: Mac/Linux and Windows. There are three chief ways in which they differ:

- The most important difference is how you separate the components of the path. Mac and Linux use slashes (e.g., `plots/diamonds.pdf`) and Windows uses backslashes (e.g., `plots\diamonds.pdf`). R can work with either type (no matter what platform you're currently using), but unfortunately, backslashes mean something special to R, and to get a single backslash in the path, you need to type two backslashes! That makes life frus-

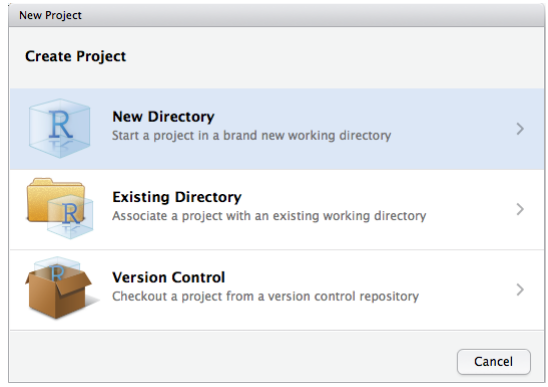
trating, so I recommend always using the Linux/Max style with forward slashes.

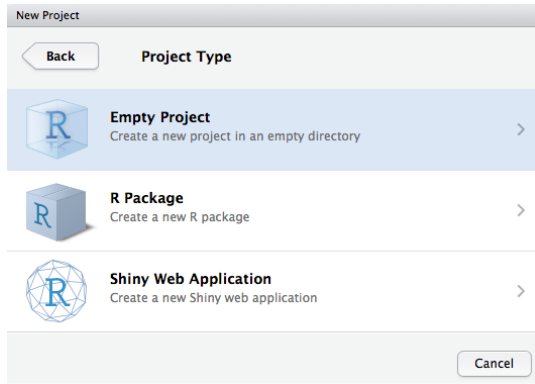
- Absolute paths (i.e., paths that point to the same place regardless of your working directory) look different. In Windows they start with a drive letter (e.g., `C:`) or two backslashes (e.g., `\\servername`) and in Mac/Linux they start with a slash "/" (e.g., `/users/hadley`). You should *never* use absolute paths in your scripts, because they hinder sharing: no one else will have exactly the same directory configuration as you.
- The last minor difference is the place that ~ points to. ~ is a convenient shortcut to your home directory. Windows doesn't really have the notion of a home directory, so it instead points to your documents directory.

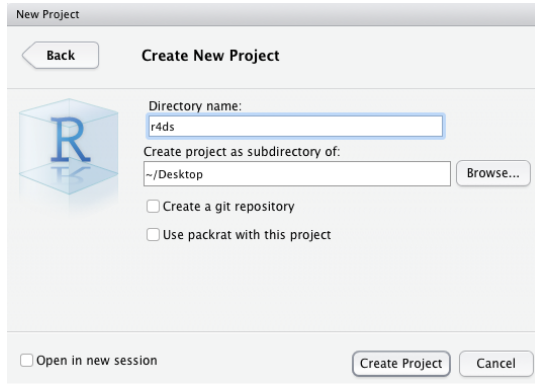
RStudio Projects

R experts keep all the files associated with a project together—input data, R scripts, analytical results, figures. This is such a wise and common practice that RStudio has built-in support for this via *projects*.

Let's make a project for you to use while you're working through the rest of this book. Click File → New Project, then:







Call your project `r4ds` and think carefully about which *subdirectory* you put the project in. If you don't store it somewhere sensible, it will be hard to find it in the future!

Once this process is complete, you'll get a new RStudio project just for this book. Check that the "home" directory of your project is the current working directory:

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

Whenever you refer to a file with a relative path it will look for it here.

Now enter the following commands in the script editor, and save the file, calling it *diamonds.R*. Next, run the complete script, which will

save a PDF and CSV file into your project directory. Don't worry about the details, you'll learn them later in the book:

```
library(tidyverse)

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

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

Quit RStudio. Inspect the folder associated with your project—notice the *.Rproj* file. Double-click that file to reopen the project. Notice you get back to where you left off: it's the same working directory and command history, and all the files you were working on are still open. Because you followed my instructions above, you will, however, have a completely fresh environment, guaranteeing that you're starting with a clean slate.

In your favorite OS-specific way, search your computer for *diamonds.pdf* and you will find the PDF (no surprise) but *also the script that created it* (*diamonds.r*). This is huge win! One day you will want to remake a figure or just understand where it came from. If you rigorously save figures to files *with R code* and never with the mouse or the clipboard, you will be able to reproduce old work with ease!

Summary

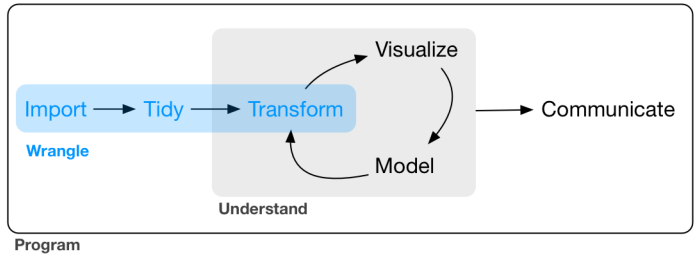
In summary, RStudio projects give you a solid workflow that will serve you well in the future:

- Create an RStudio project for each data analyis project.
- Keep data files there; we'll talk about loading them into R in Chapter 8.
- Keep scripts there; edit them, and run them in bits or as a whole.
- Save your outputs (plots and cleaned data) there.
- Only ever use relative paths, not absolute paths.

Everything you need is in one place, and cleanly separated from all the other projects that you are working on.

PART II
Wrangle

In this part of the book, you'll learn about data wrangling, the art of getting your data into R in a useful form for visualization and modeling. Data wrangling is very important: without it you can't work with your own data! There are three main parts to data wrangling:



This part of the book proceeds as follows:

- In Chapter 7, you'll learn about the variant of the data frame that we use in this book: the *tibble*. You'll learn what makes them different from regular data frames, and how you can construct them "by hand."
- In Chapter 8, you'll learn how to get your data from disk and into R. We'll focus on plain-text rectangular formats, but will give you pointers to packages that help with other types of data.

- In Chapter 9, you'll learn about tidy data, a consistent way of storing your data that makes transformation, visualization, and modeling easier. You'll learn the underlying principles, and how to get your data into a tidy form.

Data wrangling also encompasses data transformation, which you've already learned a little about. Now we'll focus on new skills for three specific types of data you will frequently encounter in practice:

- Chapter 10 will give you tools for working with multiple interrelated datasets.
- Chapter 11 will introduce regular expressions, a powerful tool for manipulating strings.
- Chapter 12 will show you how R stores categorical data. They are used when a variable has a fixed set of possible values, or when you want to use a nonalphabetical ordering of a string.
- Chapter 13 will give you the key tools for working with dates and date-times.

CHAPTER 7

Tibbles with tibble

Introduction

Throughout this book we work with "tibbles" instead of R's traditional `data.frame`. Tibbles *are* data frames, but they tweak some older behaviors to make life a little easier. R is an old language, and some things that were useful 10 or 20 years ago now get in your way. It's difficult to change base R without breaking existing code, so most innovation occurs in packages. Here we will describe the *tibble* package, which provides opinionated data frames that make working in the tidyverse a little easier. In most places, I'll use the terms tibble and data frame interchangeably; when I want to draw particular attention to R's built-in data frame, I'll call them `data.frame`s.

If this chapter leaves you wanting to learn more about tibbles, you might enjoy `vignette("tibble")`.

Prerequisites

In this chapter we'll explore the **tibble** package, part of the core tidyverse.

```
library(tidyverse)
```

Creating Tibbles

Almost all of the functions that you'll use in this book produce tibbles, as tibbles are one of the unifying features of the tidyverse. Most

other R packages use regular data frames, so you might want to coerce a data frame to a tibble. You can do that with `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
```

You can create a new tibble from individual vectors with `tibble()`. `tibble()` will automatically recycle inputs of length 1, and allows you to refer to variables that you just created, as shown here:

```
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
```

If you're already familiar with `data.frame()`, note that `tibble()` does much less: it never changes the type of the inputs (e.g., it never converts strings to factors!), it never changes the names of variables, and it never creates row names.

It's possible for a tibble to have column names that are not valid R variable names, aka *nonsyntactic* names. For example, they might not start with a letter, or they might contain unusual characters like a space. To refer to these variables, you need to surround them with backticks, `` ` ``:

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

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

You'll also need the backticks when working with these variables in other packages, like **ggplot2**, **dplyr**, and **tidyr**.

Another way to create a tibble is with `tribble()`, short for *transposed tibble*. `tribble()` is customized for data entry in code: column headings are defined by formulas (i.e., they start with ~), and entries are separated by commas. This makes it possible to lay out small amounts of data in easy-to-read form:

```
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
```

I often add a comment (the line starting with #) to make it really clear where the header is.

Tibbles Versus data.frame

There are two main differences in the usage of a tibble versus a classic `data.frame`: printing and subsetting.

Printing

Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type, a nice feature borrowed from `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
```

Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. But sometimes you need more output than the default display. There are a few options that can help.

First, you can explicitly `print()` the data frame and control the number of rows (`n`) and the `width` of the display. `width = Inf` will display all columns:

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

You can also control the default print behavior by setting options:

- `options(tibble.print_max = n, tibble.print_min = m)`: if more than `m` rows, print only `n` rows. Use `options(dplyr.print_min = Inf)` to always show all rows.
- Use `options(tibble.width = Inf)` to always print all columns, regardless of the width of the screen.

You can see a complete list of options by looking at the package help with `package?tibble`.

A final option is to use RStudio's built-in data viewer to get a scrollable view of the complete dataset. This is also often useful at the end of a long chain of manipulations:

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

Subsetting

So far all the tools you've learned have worked with complete data frames. If you want to pull out a single variable, you need some new tools, `$` and `[[`. `[[` can extract by name or position; `$` only extracts by name but is a little less typing:

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

# Extract by name
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

# Extract by position
df[[1]]
#> [1] 0.434 0.395 0.548 0.762 0.254
```

To use these in a pipe, you'll need to use the special placeholder `.`:

```
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
```

Compared to a `data.frame`, tibbles are more strict: they never do partial matching, and they will generate a warning if the column you are trying to access does not exist.

Interacting with Older Code

Some older functions don't work with tibbles. If you encounter one of these functions, use `as.data.frame()` to turn a tibble back to a `data.frame`:

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

The main reason that some older functions don't work with tibbles is the `[` function. We don't use `[` much in this book because `dplyr::filter()` and `dplyr::select()` allow you to solve the same problems with clearer code (but you will learn a little about it in "Subsetting" on page 300). With base R data frames, `[` sometimes returns a data frame, and sometimes returns a vector. With tibbles, `[` always returns another tibble.

Exercises

1. How can you tell if an object is a tibble? (Hint: try printing `mtcars`, which is a regular data frame.)

2. Compare and contrast the following operations on a `data.frame` and equivalent tibble. What is different? Why might the default data frame behaviors cause you frustration?

```
df <- data.frame(abc = 1, xyz = "a")
df$x
df[, "xyz"]
df[, c("abc", "xyz")]
```

3. If you have the name of a variable stored in an object, e.g., `var <- "mpg"`, how can you extract the reference variable from a tibble?
4. Practice referring to nonsyntactic names in the following data frame by:
   a. Extracting the variable called `1`.
   b. Plotting a scatterplot of `1` versus `2`.
   c. Creating a new column called `3`, which is `2` divided by `1`.
   d. Renaming the columns to `one`, `two`, and `three`:

```
annoying <- tibble(
  `1` = 1:10,
  `2` = `1` * 2 + rnorm(length(`1`))
)
```

5. What does `tibble::enframe()` do? When might you use it?
6. What option controls how many additional column names are printed at the footer of a tibble?

CHAPTER 8

Data Import with readr

Introduction

Working with data provided by R packages is a great way to learn the tools of data science, but at some point you want to stop learning and start working with your own data. In this chapter, you'll learn how to read plain-text rectangular files into R. Here, we'll only scratch the surface of data import, but many of the principles will translate to other forms of data. We'll finish with a few pointers to packages that are useful for other types of data.

Prerequisites

In this chapter, you'll learn how to load flat files in R with the **readr** package, which is part of the core tidyverse.

```
library(tidyverse)
```

Getting Started

Most of **readr**'s functions are concerned with turning flat files into data frames:

- `read_csv()` reads comma-delimited files, `read_csv2()` reads semicolon-separated files (common in countries where `,` is used as the decimal place), `read_tsv()` reads tab-delimited files, and `read_delim()` reads in files with any delimiter.

- `read_fwf()` reads fixed-width files. You can specify fields either by their widths with `fwf_widths()` or their position with `fwf_positions()`. `read_table()` reads a common variation of fixed-width files where columns are separated by white space.
- `read_log()` reads Apache style log files. (But also check out **webreadr**, which is built on top of `read_log()` and provides many more helpful tools.)

These functions all have similar syntax: once you've mastered one, you can use the others with ease. For the rest of this chapter we'll focus on `read_csv()`. Not only are CSV files one of the most common forms of data storage, but once you understand `read_csv()`, you can easily apply your knowledge to all the other functions in **readr**.

The first argument to `read_csv()` is the most important; it's the path to the file to read:

```
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()
#> )
```

When you run `read_csv()` it prints out a column specification that gives the name and type of each column. That's an important part of **readr**, which we'll come back to in "Parsing a File" on page 137.

You can also supply an inline CSV file. This is useful for experimenting with **readr** and for creating reproducible examples to share with others:

```
read_csv("a,b,c
1,2,3
4,5,6")
#> # A tibble: 2 × 3
#>       a     b     c
#>   <int> <int> <int>
#> 1     1     2     3
#> 2     4     5     6
```

In both cases `read_csv()` uses the first line of the data for the column names, which is a very common convention. There are two cases where you might want to tweak this behavior:

- Sometimes there are a few lines of metadata at the top of the file. You can use `skip = n` to skip the first `n` lines; or use `comment = "#"` to drop all lines that start with (e.g.) `#`:

```
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
```

- The data might not have column names. You can use `col_names = FALSE` to tell `read_csv()` not to treat the first row as headings, and instead label them sequentially from `X1` to `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"` is a convenient shortcut for adding a new line. You'll learn more about it and other types of string escape in "String Basics" on page 195.)

  Alternatively you can pass `col_names` a character vector, which will be used as the column names:

```
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
```

Another option that commonly needs tweaking is `na`. This specifies the value (or values) that are used to represent missing values in your file:

```
read_csv("a,b,c\n1,2,.", na = ".")
#> # A tibble: 1 × 3
#>       a     b     c
#>   <int> <int> <chr>
#> 1     1     2  <NA>
```

This is all you need to know to read ~75% of CSV files that you'll encounter in practice. You can also easily adapt what you've learned to read tab-separated files with `read_tsv()` and fixed-width files with `read_fwf()`. To read in more challenging files, you'll need to learn more about how **readr** parses each column, turning them into R vectors.

Compared to Base R

If you've used R before, you might wonder why we're not using `read.csv()`. There are a few good reasons to favor **readr** functions over the base equivalents:

- They are typically much faster (~10x) than their base equivalents. Long-running jobs have a progress bar, so you can see what's happening. If you're looking for raw speed, try `data.table::fread()`. It doesn't fit quite so well into the tidyverse, but it can be quite a bit faster.
- They produce tibbles, and they don't convert character vectors to factors, use row names, or munge the column names. These are common sources of frustration with the base R functions.
- They are more reproducible. Base R functions inherit some behavior from your operating system and environment variables, so import code that works on your computer might not work on someone else's.

Exercises

1. What function would you use to read a file where fields are separated with "|"?

2. Apart from `file`, `skip`, and `comment`, what other arguments do `read_csv()` and `read_tsv()` have in common?
3. What are the most important arguments to `read_fwf()`?
4. Sometimes strings in a CSV file contain commas. To prevent them from causing problems they need to be surrounded by a quoting character, like `"` or `'`. By convention, `read_csv()` assumes that the quoting character will be `"`, and if you want to change it you'll need to use `read_delim()` instead. What arguments do you need to specify to read the following text into a data frame?

   ```
   "x,y\n1,'a,b'"
   ```

5. Identify what is wrong with each of the following inline CSV files. What happens when you run the code?

   ```
   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")
   ```

Parsing a Vector

Before we get into the details of how **readr** reads files from disk, we need to take a little detour to talk about the `parse_*()` functions. These functions take a character vector and return a more specialized vector like a logical, integer, or date:

```
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"
```

These functions are useful in their own right, but are also an important building block for **readr**. Once you've learned how the individual parsers work in this section, we'll circle back and see how they fit together to parse a complete file in the next section.

Like all functions in the tidyverse, the `parse_*()` functions are uniform; the first argument is a character vector to parse, and the `na` argument specifies which strings should be treated as missing:

```
parse_integer(c("1", "231", ".", "456"), na = ".")
#> [1]   1 231  NA 456
```

If parsing fails, you'll get a warning:

```
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
```

And the failures will be missing in the output:

```
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
```

If there are many parsing failures, you'll need to use `problems()` to get the complete set. This returns a tibble, which you can then manipulate with **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
```

Using parsers is mostly a matter of understanding what's available and how they deal with different types of input. There are eight particularly important parsers:

- `parse_logical()` and `parse_integer()` parse logicals and integers, respectively. There's basically nothing that can go wrong with these parsers so I won't describe them here further.
- `parse_double()` is a strict numeric parser, and `parse_number()` is a flexible numeric parser. These are more complicated than you might expect because different parts of the world write numbers in different ways.
- `parse_character()` seems so simple that it shouldn't be necessary. But one complication makes it quite important: character encodings.

- `parse_factor()` creates factors, the data structure that R uses to represent categorical variables with fixed and known values.
- `parse_datetime()`, `parse_date()`, and `parse_time()` allow you to parse various date and time specifications. These are the most complicated because there are so many different ways of writing dates.

The following sections describe these parsers in more detail.

Numbers

It seems like it should be straightforward to parse a number, but three problems make it tricky:

- People write numbers differently in different parts of the world. For example, some countries use `.` in between the integer and fractional parts of a real number, while others use `,`.
- Numbers are often surrounded by other characters that provide some context, like "$1000" or "10%".
- Numbers often contain "grouping" characters to make them easier to read, like "1,000,000", and these grouping characters vary around the world.

To address the first problem, **readr** has the notion of a "locale," an object that specifies parsing options that differ from place to place. When parsing numbers, the most important option is the character you use for the decimal mark. You can override the default value of `.` by creating a new locale and setting the `decimal_mark` argument:

```
parse_double("1.23")
#> [1] 1.23
parse_double("1,23", locale = locale(decimal_mark = ","))
#> [1] 1.23
```

readr's default locale is US-centric, because generally R is US-centric (i.e., the documentation of base R is written in American English). An alternative approach would be to try and guess the defaults from your operating system. This is hard to do well, and, more importantly, makes your code fragile: even if it works on your computer, it might fail when you email it to a colleague in another country.

`parse_number()` addresses the second problem: it ignores non-numeric characters before and after the number. This is particularly useful for currencies and percentages, but also works to extract numbers embedded in text:

```
parse_number("$100")
#> [1] 100
parse_number("20%")
#> [1] 20
parse_number("It cost $123.45")
#> [1] 123
```

The final problem is addressed by the combination of `parse_number()` and the locale as `parse_number()` will ignore the "grouping mark":

```
# Used in America
parse_number("$123,456,789")
#> [1] 1.23e+08

# Used in many parts of Europe
parse_number(
  "123.456.789",
  locale = locale(grouping_mark = ".")
)
#> [1] 1.23e+08

# Used in Switzerland
parse_number(
  "123'456'789",
  locale = locale(grouping_mark = "'")
)
#> [1] 1.23e+08
```

Strings

It seems like `parse_character()` should be really simple—it could just return its input. Unfortunately life isn't so simple, as there are multiple ways to represent the same string. To understand what's going on, we need to dive into the details of how computers represent strings. In R, we can get at the underlying representation of a string using `charToRaw()`:

```
charToRaw("Hadley")
#> [1] 48 61 64 6c 65 79
```

Each hexadecimal number represents a byte of information: `48` is H, `61` is a, and so on. The mapping from hexadecimal number to character is called the encoding, and in this case the encoding is called

ASCII. ASCII does a great job of representing English characters, because it's the *American* Standard Code for Information Interchange.

Things get more complicated for languages other than English. In the early days of computing there were many competing standards for encoding non-English characters, and to correctly interpret a string you needed to know both the values and the encoding. For example, two common encodings are Latin1 (aka ISO-8859-1, used for Western European languages) and Latin2 (aka ISO-8859-2, used for Eastern European languages). In Latin1, the byte `b1` is "±", but in Latin2, it's "ą"! Fortunately, today there is one standard that is supported almost everywhere: UTF-8. UTF-8 can encode just about every character used by humans today, as well as many extra symbols (like emoji!).

readr uses UTF-8 everywhere: it assumes your data is UTF-8 encoded when you read it, and always uses it when writing. This is a good default, but will fail for data produced by older systems that don't understand UTF-8. If this happens to you, your strings will look weird when you print them. Sometimes just one or two characters might be messed up; other times you'll get complete gibberish. For example:

```
x1 <- "El Ni\xf1o was particularly bad this year"
x2 <- "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
```

To fix the problem you need to specify the encoding in `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"
```

How do you find the correct encoding? If you're lucky, it'll be included somewhere in the data documentation. Unfortunately, that's rarely the case, so **readr** provides `guess_encoding()` to help you figure it out. It's not foolproof, and it works better when you have lots of text (unlike here), but it's a reasonable place to start. Expect to try a few different encodings before you find the right one:

```
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
```

The first argument to `guess_encoding()` can either be a path to a file, or, as in this case, a raw vector (useful if the strings are already in R).

Encodings are a rich and complex topic, and I've only scratched the surface here. If you'd like to learn more I'd recommend reading the detailed explanation at *http://kunststube.net/encoding/*.

Factors

R uses factors to represent categorical variables that have a known set of possible values. Give `parse_factor()` a vector of known `levels` to generate a warning whenever an unexpected value is present:

```
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
```

But if you have many problematic entries, it's often easier to leave them as character vectors and then use the tools you'll learn about in Chapter 11 and Chapter 12 to clean them up.

Dates, Date-Times, and Times

You pick between three parsers depending on whether you want a date (the number of days since 1970-01-01), a date-time (the number of seconds since midnight 1970-01-01), or a time (the number of seconds since midnight). When called without any additional arguments:

- `parse_datetime()` expects an ISO8601 date-time. ISO8601 is an international standard in which the components of a date are organized from biggest to smallest: year, month, day, hour, minute, second:

```
parse_datetime("2010-10-01T2010")
#> [1] "2010-10-01 20:10:00 UTC"

# If time is omitted, it will be set to midnight
parse_datetime("20101010")
#> [1] "2010-10-10 UTC"
```

This is the most important date/time standard, and if you work with dates and times frequently, I recommend reading *https://en.wikipedia.org/wiki/ISO_8601*.

- `parse_date()` expects a four-digit year, a `-` or `/`, the month, a `-` or `/`, then the day:

```
parse_date("2010-10-01")
#> [1] "2010-10-01"
```

- `parse_time()` expects the hour, `:`, minutes, optionally `:` and seconds, and an optional a.m./p.m. specifier:

```
library(hms)
parse_time("01:10 am")
#> 01:10:00
parse_time("20:10:01")
#> 20:10:01
```

Base R doesn't have a great built-in class for time data, so we use the one provided in the **hms** package.

If these defaults don't work for your data you can supply your own date-time `format`, built up of the following pieces:

Year

`%Y` (4 digits).

`%y` (2 digits; 00-69 → 2000-2069, 70-99 → 1970-1999).

Month

`%m` (2 digits).

`%b` (abbreviated name, like "Jan").

`%B` (full name, "January").

Day

`%d` (2 digits).

`%e` (optional leading space).

Time

`%H` (0-23 hour format).

`%I` (0-12, must be used with `%p`).

`%p` (a.m./p.m. indicator).

`%M` (minutes).

`%S` (integer seconds).

`%OS` (real seconds).

`%Z` (time zone [a name, e.g., `America/Chicago`]). Note: beware of abbreviations. If you're American, note that "EST" is a Canadian time zone that does not have daylight saving time. It is Eastern Standard Time! We'll come back to this in "Time Zones" on page 254.

`%z` (as offset from UTC, e.g., `+0800`).

Nondigits

`%.` (skips one nondigit character).

`%*` (skips any number of nondigits).

The best way to figure out the correct format is to create a few examples in a character vector, and test with one of the parsing functions. For example:

```
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"
```

If you're using `%b` or `%B` with non-English month names, you'll need to set the `lang` argument to `locale()`. See the list of built-in languages in `date_names_langs()`, or if your language is not already included, create your own with `date_names()`:

```
parse_date("1 janvier 2015", "%d %B %Y", locale = locale("fr"))
#> [1] "2015-01-01"
```

Exercises

1. What are the most important arguments to `locale()`?

2. What happens if you try and set `decimal_mark` and `grouping_mark` to the same character? What happens to the default value of `grouping_mark` when you set `decimal_mark` to `","`? What happens to the default value of `decimal_mark` when you set the `grouping_mark` to `"."`?
3. I didn't discuss the `date_format` and `time_format` options to `locale()`. What do they do? Construct an example that shows when they might be useful.
4. If you live outside the US, create a new locale object that encapsulates the settings for the types of files you read most commonly.
5. What's the difference between `read_csv()` and `read_csv2()`?
6. What are the most common encodings used in Europe? What are the most common encodings used in Asia? Do some googling to find out.
7. Generate the correct format string to parse each of the following dates and times:

```
d1 <- "January 1, 2010"
d2 <- "2015-Mar-07"
d3 <- "06-Jun-2017"
d4 <- c("August 19 (2015)", "July 1 (2015)")
d5 <- "12/30/14" # Dec 30, 2014
t1 <- "1705"
t2 <- "11:15:10.12 PM"
```

Parsing a File

Now that you've learned how to parse an individual vector, it's time to return to the beginning and explore how **readr** parses a file. There are two new things that you'll learn about in this section:

- How **readr** automatically guesses the type of each column.
- How to override the default specification.

Strategy

readr uses a heuristic to figure out the type of each column: it reads the first 1000 rows and uses some (moderately conservative) heuristics to figure out the type of each column. You can emulate this pro-

cess with a character vector using `guess_parser()`, which returns **readr**'s best guess, and `parse_guess()`, which uses that guess to parse the column:

```
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"
```

The heuristic tries each of the following types, stopping when it finds a match:

logical
: Contains only "F", "T", "FALSE", or "TRUE".

integer
: Contains only numeric characters (and `-`).

double
: Contains only valid doubles (including numbers like `4.5e-5`).

number
: Contains valid doubles with the grouping mark inside.

time
: Matches the default `time_format`.

date
: Matches the default `date_format`.

date-time
: Any ISO8601 date.

If none of these rules apply, then the column will stay as a vector of strings.

Problems

These defaults don't always work for larger files. There are two basic problems:

- The first thousand rows might be a special case, and **readr** guesses a type that is not sufficiently general. For example, you might have a column of doubles that only contains integers in the first 1000 rows.
- The column might contain a lot of missing values. If the first 1000 rows contain only `NA`s, **readr** will guess that it's a character vector, whereas you probably want to parse it as something more specific.

readr contains a challenging CSV that illustrates both of these problems:

```
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.
```

(Note the use of `readr_example()`, which finds the path to one of the files included with the package.)

There are two printed outputs: the column specification generated by looking at the first 1000 rows, and the first five parsing failures. It's always a good idea to explicitly pull out the `problems()`, so you can explore them in more depth:

```
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
```

A good strategy is to work column by column until there are no problems remaining. Here we can see that there are a lot of parsing problems with the `x` column—there are trailing characters after the integer value. That suggests we need to use a double parser instead.

To fix the call, start by copying and pasting the column specification into your original call:

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_integer(),
    y = col_character()
  )
)
```

Then you can tweak the type of the `x` column:

```
challenge <- read_csv(
  readr_example("challenge.csv"),
  col_types = cols(
    x = col_double(),
    y = col_character()
  )
)
```

That fixes the first problem, but if we look at the last few rows, you'll see that they're dates stored in a character vector:

```
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
```

You can fix that by specifying that `y` is a date column:

```
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
```

Every `parse_xyz()` function has a corresponding `col_xyz()` function. You use `parse_xyz()` when the data is in a character vector in R already; you use `col_xyz()` when you want to tell **readr** how to load the data.

I highly recommend always supplying `col_types`, building up from the printout provided by **readr**. This ensures that you have a consistent and reproducible data import script. If you rely on the default guesses and your data changes, **readr** will continue to read it in. If you want to be really strict, use `stop_for_problems()`: that will throw an error and stop your script if there are any parsing problems.

Other Strategies

There are a few other general strategies to help you parse files:

- In the previous example, we just got unlucky: if we look at just one more row than the default, we can correctly parse in one shot:

```
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
```

- Sometimes it's easier to diagnose problems if you just read in all the columns as character vectors:

```
challenge2 <- read_csv(readr_example("challenge.csv"),
  col_types = cols(.default = col_character())
)
```

  This is particularly useful in conjunction with `type_convert()`, which applies the parsing heuristics to the character columns in a data frame:

```
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

# Note the column types
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
```

- If you're reading a very large file, you might want to set `n_max` to a smallish number like 10,000 or 100,000. That will accelerate your iterations while you eliminate common problems.

- If you're having major parsing problems, sometimes it's easier to just read into a character vector of lines with `read_lines()`, or even a character vector of length 1 with `read_file()`. Then you can use the string parsing skills you'll learn later to parse more exotic formats.

Writing to a File

readr also comes with two useful functions for writing data back to disk: `write_csv()` and `write_tsv()`. Both functions increase the chances of the output file being read back in correctly by:

- Always encoding strings in UTF-8.
- Saving dates and date-times in ISO8601 format so they are easily parsed elsewhere.

If you want to export a CSV file to Excel, use `write_excel_csv()`—this writes a special character (a "byte order mark") at the start of the file, which tells Excel that you're using the UTF-8 encoding.

The most important arguments are `x` (the data frame to save) and `path` (the location to save it). You can also specify how missing values are written with `na`, and if you want to `append` to an existing file:

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

Note that the type information is lost when you save to CSV:

```
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
```

This makes CSVs a little unreliable for caching interim results—you need to re-create the column specification every time you load in. There are two alternatives:

- `write_rds()` and `read_rds()` are uniform wrappers around the base functions `readRDS()` and `saveRDS()`. These store data in R's custom binary format called 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
```

- The **feather** package implements a fast binary file format that can be shared across programming languages:

```
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
```

feather tends to be faster than RDS and is usable outside of R. RDS supports list-columns (which you'll learn about in Chapter 20); **feather** currently does not.

Other Types of Data

To get other types of data into R, we recommend starting with the tidyverse packages listed next. They're certainly not perfect, but they are a good place to start. For rectangular data:

- **haven** reads SPSS, Stata, and SAS files.
- **readxl** reads Excel files (both *.xls* and *.xlsx*).
- **DBI**, along with a database-specific backend (e.g., **RMySQL**, **RSQLite**, **RPostgreSQL**, etc.) allows you to run SQL queries against a database and return a data frame.

For hierarchical data: use **jsonlite** (by Jeroen Ooms) for JSON, and **xml2** for XML. Jenny Bryan has some excellent worked examples at *https://jennybc.github.io/purrr-tutorial/*.

For other file types, try the R data import/export manual and the **rio** package.

CHAPTER 9

Tidy Data with tidyr

Introduction

> Happy families are all alike; every unhappy family is unhappy in its own way.
>
> —Leo Tolstoy

> Tidy datasets are all alike, but every messy dataset is messy in its own way.
>
> —Hadley Wickham

In this chapter, you will learn a consistent way to organize your data in R, an organization called *tidy data*. Getting your data into this format requires some up-front work, but that work pays off in the long term. Once you have tidy data and the tidy tools provided by packages in the tidyverse, you will spend much less time munging data from one representation to another, allowing you to spend more time on the analytic questions at hand.

This chapter will give you a practical introduction to tidy data and the accompanying tools in the **tidyr** package. If you'd like to learn more about the underlying theory, you might enjoy the *Tidy Data* paper published in the *Journal of Statistical Software*.

Prerequisites

In this chapter we'll focus on **tidyr**, a package that provides a bunch of tools to help tidy up your messy datasets. **tidyr** is a member of the core tidyverse.

```
library(tidyverse)
```

Tidy Data

You can represent the same underlying data in multiple ways. The following example shows the same data organized in four different ways. Each dataset shows the same values of four variables, *country*, *year*, *population*, and *cases*, but each dataset organizes the values in a different way:

```
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

# Spread across two tibbles
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
```

These are all representations of the same underlying data, but they are not equally easy to use. One dataset, the tidy dataset, will be much easier to work with inside the tidyverse.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

Figure 9-1 shows the rules visually.

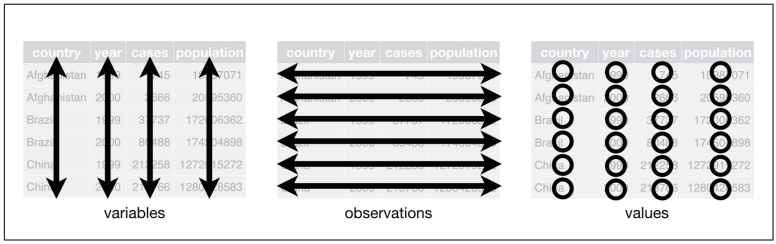


Figure 9-1. The following three rules make a dataset tidy: variables are in columns, observations are in rows, and values are in cells

These three rules are interrelated because it's impossible to only satisfy two of the three. That interrelationship leads to an even simpler set of practical instructions:

1. Put each dataset in a tibble.
2. Put each variable in a column.

In this example, only `table1` is tidy. It's the only representation where each column is a variable.

Why ensure that your data is tidy? There are two main advantages:

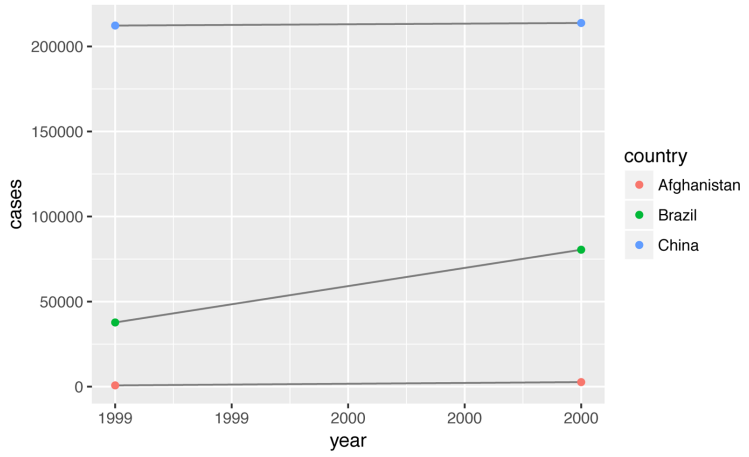
- There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to

learn the tools that work with it because they have an underlying uniformity.

- There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine. As you learned in "Useful Creation Functions" on page 56 and "Useful Summary Functions" on page 66, most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

dplyr, **ggplot2**, and all the other packages in the tidyverse are designed to work with tidy data. Here are a couple of small examples showing how you might work with `table1`:

```
# Compute rate per 10,000
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

# Compute cases per year
table1 %>%
  count(year, wt = cases)
#> # A tibble: 2 × 2
#>    year      n
#>   <int>  <int>
#> 1  1999 250740
#> 2  2000 296920

# Visualize changes over time
library(ggplot2)
ggplot(table1, aes(year, cases)) +
  geom_line(aes(group = country), color = "grey50") +
  geom_point(aes(color = country))
```



Exercises

1. Using prose, describe how the variables and observations are organized in each of the sample tables.
2. Compute the `rate` for `table2`, and `table4a` + `table4b`. You will need to perform four operations:
   a. Extract the number of TB cases per country per year.
   b. Extract the matching population per country per year.
   c. Divide cases by population, and multiply by 10,000.
   d. Store back in the appropriate place.

   Which representation is easiest to work with? Which is hardest? Why?
3. Re-create the plot showing change in cases over time using `table2` instead of `table1`. What do you need to do first?

Spreading and Gathering

The principles of tidy data seem so obvious that you might wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most data that you will encounter will be untidy. There are two main reasons:

- Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
- Data is often organized to facilitate some use other than analysis. For example, data is often organized to make entry as easy as possible.

This means for most real analyses, you'll need to do some tidying. The first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. The second step is to resolve one of two common problems:

- One variable might be spread across multiple columns.
- One observation might be scattered across multiple rows.

Typically a dataset will only suffer from one of these problems; it'll only suffer from both if you're really unlucky! To fix these problems, you'll need the two most important functions in **tidyr**: `gather()` and `spread()`.

Gathering

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`; the column names `1999` and `2000` represent values of the `year` variable, and each row represents two observations, not one:

```
table4a
#> # A tibble: 3 × 3
#>       country `1999` `2000`
#> *       <chr>  <int>  <int>
#> 1 Afghanistan    745   2666
#> 2      Brazil  37737  80488
#> 3       China 212258 213766
```

To tidy a dataset like this, we need to *gather* those columns into a new pair of variables. To describe that operation we need three parameters:

- The set of columns that represent values, not variables. In this example, those are the columns `1999` and `2000`.

- The name of the variable whose values form the column names. I call that the `key`, and here it is `year`.
- The name of the variable whose values are spread over the cells. I call that `value`, and here it's the number of `cases`.

Together those parameters generate the call to `gather()`:

```
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  1999 212258
#> 4 Afghanistan  2000   2666
#> 5      Brazil  2000  80488
#> 6       China  2000 213766
```

The columns to gather are specified with `dplyr::select()` style notation. Here there are only two columns, so we list them individually. Note that "1999" and "2000" are nonsyntactic names so we have to surround them in backticks. To refresh your memory of the other ways to select columns, see "Select Columns with select()" on page 51.

In the final result, the gathered columns are dropped, and we get new `key` and `value` columns. Otherwise, the relationships between the original variables are preserved. Visually, this is shown in Figure 9-2. We can use `gather()` to tidy `table4b` in a similar fashion. The only difference is the variable stored in the cell values:

```
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
```

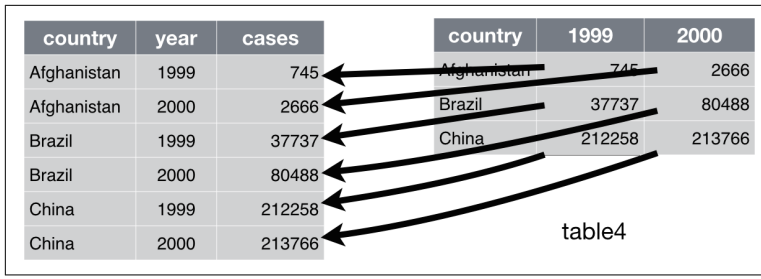


Figure 9-2. Gathering table4 into a tidy form

To combine the tidied versions of `table4a` and `table4b` into a single tibble, we need to use `dplyr::left_join()`, which you'll learn about in Chapter 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
```

Spreading

Spreading is the opposite of gathering. You use it when an observation is scattered across multiple rows. For example, take `table2`—an observation is a country in a year, but each observation is spread across two rows:

```
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
```

To tidy this up, we first analyze the representation in a similar way to `gather()`. This time, however, we only need two parameters:

- The column that contains variable names, the `key` column. Here, it's `type`.
- The column that contains values forms multiple variables, the `value` column. Here, it's `count`.

Once we've figured that out, we can use `spread()`, as shown programmatically here, and visually in Figure 9-3:

```
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
```

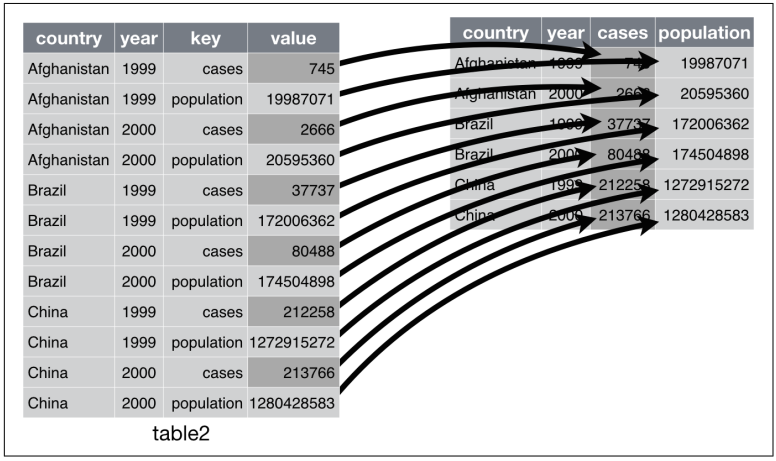


Figure 9-3. Spreading table2 makes it tidy

As you might have guessed from the common `key` and `value` arguments, `spread()` and `gather()` are complements. `gather()` makes wide tables narrower and longer; `spread()` makes long tables shorter and wider.

Exercises

1. Why are `gather()` and `spread()` not perfectly symmetrical? Carefully consider the following example:

```
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`)
```

   (Hint: look at the variable types and think about column *names*.)

   Both `spread()` and `gather()` have a `convert` argument. What does it do?

2. Why does this code fail?

```
table4a %>%
  gather(1999, 2000, key = "year", value = "cases")
#> Error in eval(expr, envir, enclos):
#> Position must be between 0 and n
```

3. Why does spreading this tibble fail? How could you add a new column to fix the problem?

```
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. Tidy this simple tibble. Do you need to spread or gather it? What are the variables?

```
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes",     NA,    10,
  "no",      20,    12
)
```

Separating and Pull

So far you've learned how to tidy `table2` and `table4`, but not `table3`. `table3` has a different problem: we have one column (`rate`) that contains two variables (`cases` and `population`). To fix this problem, we'll need the `separate()` function. You'll also learn about the complement of `separate()`: `unite()`, which you use if a single variable is spread across multiple columns.

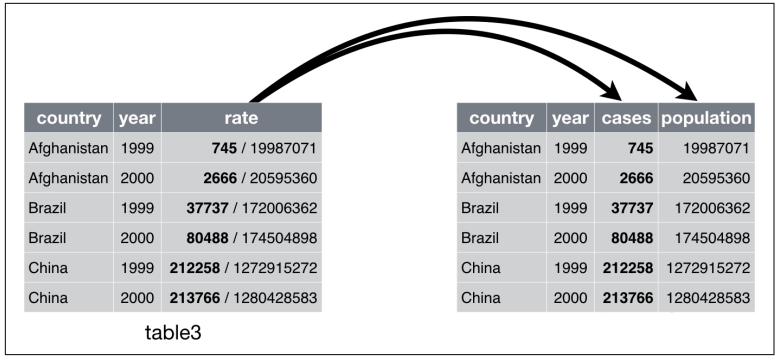
Separate

`separate()` pulls apart one column into multiple columns, by splitting wherever a separator character appears. Take `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
```

The `rate` column contains both `cases` and `population` variables, and we need to split it into two variables. `separate()` takes the name of the column to separate, and the names of the columns to separate into, as shown in Figure 9-4 and the following code:

```
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
```



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

Figure 9-4. Separating table3 makes it tidy

By default, `separate()` will split values wherever it sees a non-alphanumeric character (i.e., a character that isn't a number or letter). For example, in the preceding code, `separate()` split the values of `rate` at the forward slash characters. If you wish to use a specific character to separate a column, you can pass the character to the `sep` argument of `separate()`. For example, we could rewrite the preceding code as:

```
table3 %>%
  separate(rate, into = c("cases", "population"), sep = "/")
```

(Formally, `sep` is a regular expression, which you'll learn more about in Chapter 11.)

Look carefully at the column types: you'll notice that `case` and `population` are character columns. This is the default behavior in `separate()`: it leaves the type of the column as is. Here, however, it's not very useful as those really are numbers. We can ask `separate()` to try and convert to better types using `convert = TRUE`:

```
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
```

You can also pass a vector of integers to `sep`. `separate()` will interpret the integers as positions to split at. Positive values start at 1 on the far left of the strings; negative values start at –1 on the far right of the strings. When using integers to separate strings, the length of `sep` should be one less than the number of names in `into`.

You can use this arrangement to separate the last two digits of each year. This makes this data less tidy, but is useful in other cases, as you'll see in a little bit:

```
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
```

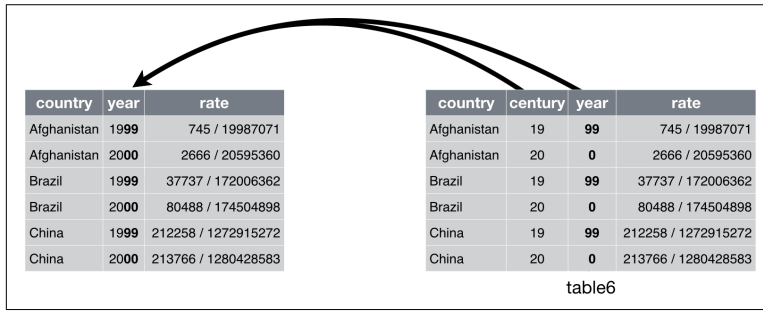
Unite

`unite()` is the inverse of `separate()`: it combines multiple columns into a single column. You'll need it much less frequently than `separate()`, but it's still a useful tool to have in your back pocket.

We can use `unite()` to rejoin the *century* and *year* columns that we created in the last example. That data is saved as `tidyr::table5`. `unite()` takes a data frame, the name of the new variable to create, and a set of columns to combine, again specified in `dplyr::select()` . The result is shown in Figure 9-5 and in the following code:

```
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
```



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

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

Figure 9-5. Uniting table5 makes it tidy

In this case we also need to use the `sep` argument. The default will place an underscore (_) between the values from different columns. Here we don't want any separator so we use `""`:

```
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
```

Exercises

1. What do the `extra` and `fill` arguments do in `separate()`? Experiment with the various options for the following two toy datasets:

```
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. Both `unite()` and `separate()` have a `remove` argument. What does it do? Why would you set it to `FALSE`?

3. Compare and contrast `separate()` and `extract()`. Why are there three variations of separation (by position, by separator, and with groups), but only one unite?

Missing Values

Changing the representation of a dataset brings up an important subtlety of missing values. Surprisingly, a value can be missing in one of two possible ways:

- *Explicitly*, i.e., flagged with `NA`.
- *Implicitly*, i.e., simply not present in the data.

Let's illustrate this idea with a very simple dataset:

```
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)
)
```

There are two missing values in this dataset:

- The return for the fourth quarter of 2015 is explicitly missing, because the cell where its value should be instead contains `NA`.
- The return for the first quarter of 2016 is implicitly missing, because it simply does not appear in the dataset.

One way to think about the difference is with this Zen-like koan: an explicit missing value is the presence of an absence; an implicit missing value is the absence of a presence.

The way that a dataset is represented can make implicit values explicit. For example, we can make the implicit missing value explicit by putting years in the columns:

```
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
```

Because these explicit missing values may not be important in other representations of the data, you can set `na.rm = TRUE` in `gather()` to turn explicit missing values implicit:

```
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
```

Another important tool for making missing values explicit in tidy data is `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
```

`complete()` takes a set of columns, and finds all unique combinations. It then ensures the original dataset contains all those values, filling in explicit `NA`s where necessary.

There's one other important tool that you should know for working with missing values. Sometimes when a data source has primarily been used for data entry, missing values indicate that the previous value should be carried forward:

```
treatment <- tribble(
  ~ person,           ~ treatment, ~response,
  "Derrick Whitmore", 1,           7,
  NA,                 2,           10,
  NA,                 3,           9,
  "Katherine Burke",  1,           4
)
```

You can fill in these missing values with `fill()`. It takes a set of columns where you want missing values to be replaced by the most recent nonmissing value (sometimes called last observation carried forward):

```
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
```

Exercises

1. Compare and contrast the `fill` arguments to `spread()` and `complete()`.
2. What does the direction argument to `fill()` do?

Case Study

To finish off the chapter, let's pull together everything you've learned to tackle a realistic data tidying problem. The `tidyr::who` dataset contains tuberculosis (TB) cases broken down by year, country, age, gender, and diagnosis method. The data comes from the *2014 World Health Organization Global Tuberculosis Report*, available at *http://www.who.int/tb/country/data/download/en/*.

There's a wealth of epidemiological information in this dataset, but it's challenging to work with the data in the form that it's provided:

```
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>
```

This is a very typical real-life dataset. It contains redundant columns, odd variable codes, and many missing values. In short, `who` is messy, and we'll need multiple steps to tidy it. Like **dplyr**, **tidyr** is designed so that each function does one thing well. That means in real-life situations you'll usually need to string together multiple verbs into a pipeline.

The best place to start is almost always to gather together the columns that are not variables. Let's have a look at what we've got:

- It looks like `country`, `iso2`, and `iso3` are three variables that redundantly specify the country.
- `year` is clearly also a variable.
- We don't know what all the other columns are yet, but given the structure in the variable names (e.g., `new_sp_m014`, `new_ep_m014`, `new_ep_f014`) these are likely to be values, not variables.

So we need to gather together all the columns from `new_sp_m014` to `newrel_f65`. We don't know what those values represent yet, so we'll

give them the generic name "key". We know the cells represent the count of cases, so we'll use the variable cases. There are a lot of missing values in the current representation, so for now we'll use na.rm just so we can focus on the values that are present:

```
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
```

We can get some hint of the structure of the values in the new key column by counting them:

```
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
```

You might be able to parse this out by yourself with a little thought and some experimentation, but luckily we have the data dictionary handy. It tells us:

1. The first three letters of each column denote whether the column contains new or old cases of TB. In this dataset, each column contains new cases.
2. The next two letters describe the type of TB:

   - rel stands for cases of relapse.

- `ep` stands for cases of extrapulmonary TB.
- `sn` stands for cases of pulmonary TB that could not be diagnosed by a pulmonary smear (smear negative).
- `sp` stands for cases of pulmonary TB that could be diagnosed be a pulmonary smear (smear positive).

3. The sixth letter gives the sex of TB patients. The dataset groups cases by males (`m`) and females (`f`).
4. The remaining numbers give the age group. The dataset groups cases into seven age groups:

   - `014` = 0–14 years old
   - `1524` = 15–24 years old
   - `2534` = 25–34 years old
   - `3544` = 35–44 years old
   - `4554` = 45–54 years old
   - `5564` = 55–64 years old
   - `65` = 65 or older

We need to make a minor fix to the format of the column names: unfortunately the names are slightly inconsistent because instead of `new_rel` we have `newrel` (it's hard to spot this here but if you don't fix it we'll get errors in subsequent steps). You'll learn about `str_replace()` in Chapter 11, but the basic idea is pretty simple: replace the characters "newrel" with "new_rel". This makes all variable names consistent:

```
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
```

We can separate the values in each code with two passes of `separate()`. The first pass will split the codes at each underscore:

```
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
```

Then we might as well drop the `new` column because it's constant in this dataset. While we're dropping columns, let's also drop `iso2` and `iso3` since they're redundant:

```
who3 %>%
  count(new)
#> # A tibble: 1 × 2
#>     new     n
#>   <chr> <int>
#> 1   new 76046
who4 <- who3 %>%
  select(-new, -iso2, -iso3)
```

Next we'll separate `sexage` into `sex` and `age` by splitting after the first character:

```
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
```

The `who` dataset is now tidy!

I've shown you the code a piece at a time, assigning each interim result to a new variable. This typically isn't how you'd work interactively. Instead, you'd gradually build up a complex pipe:

```
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)
```

Exercises

1. In this case study I set `na.rm = TRUE` just to make it easier to check that we had the correct values. Is this reasonable? Think about how missing values are represented in this dataset. Are there implicit missing values? What's the difference between an `NA` and zero?
2. What happens if you neglect the `mutate()` step? (`mutate(key = stringr::str_replace(key, "newrel", "new_rel"))`).
3. I claimed that `iso2` and `iso3` were redundant with `country`. Confirm this claim.
4. For each country, year, and sex compute the total number of cases of TB. Make an informative visualization of the data.

Nontidy Data

Before we continue on to other topics, it's worth talking briefly about nontidy data. Earlier in the chapter, I used the pejorative term "messy" to refer to nontidy data. That's an oversimplification: there are lots of useful and well-founded data structures that are not tidy data. There are two main reasons to use other data structures:

- Alternative representations may have substantial performance or space advantages.
- Specialized fields have evolved their own conventions for storing data that may be quite different to the conventions of tidy data.

Either of these reasons means you'll need something other than a tibble (or data frame). If your data does fit naturally into a rectangular structure composed of observations and variables, I think tidy data should be your default choice. But there are good reasons to use other structures; tidy data is not the only way. If you'd like to learn more about nontidy data, I'd highly recommend this thoughtful blog post by Jeff Leek.

CHAPTER 10

Relational Data with dplyr

Introduction

It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called *relational data* because it is the relations, not just the individual datasets, that are important.

Relations are always defined between a pair of tables. All other relations are built up from this simple idea: the relations of three or more tables are always a property of the relations between each pair. Sometimes both elements of a pair can be the same table! This is needed if, for example, you have a table of people, and each person has a reference to their parents.

To work with relational data you need verbs that work with pairs of tables. There are three families of verbs designed to work with relational data:

- *Mutating joins*, which add new variables to one data frame from matching observations in another.
- *Filtering joins*, which filter observations from one data frame based on whether or not they match an observation in the other table.
- *Set operations*, which treat observations as if they were set elements.

The most common place to find relational data is in a *relational* database management system (or RDBMS), a term that encompasses almost all modern databases. If you've used a database before, you've almost certainly used SQL. If so, you should find the concepts in this chapter familiar, although their expression in **dplyr** is a little different. Generally, **dplyr** is a little easier to use than SQL because **dplyr** is specialized to do data analysis: it makes common data analysis operations easier, at the expense of making it more difficult to do other things that aren't commonly needed for data analysis.

Prerequisites

We will explore relational data from **nycflights13** using the two-table verbs from **dplyr**.

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

nycflights13

We will use the **nycflights13** package to learn about relational data. **nycflights13** contains four tibbles that are related to the `flights` table that you used in Chapter 3:

- `airlines` lets you look up the full carrier name from its abbreviated code:

```
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
```

- `airports` gives information about each airport, identified by the `faa` airport code:

```
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>
```

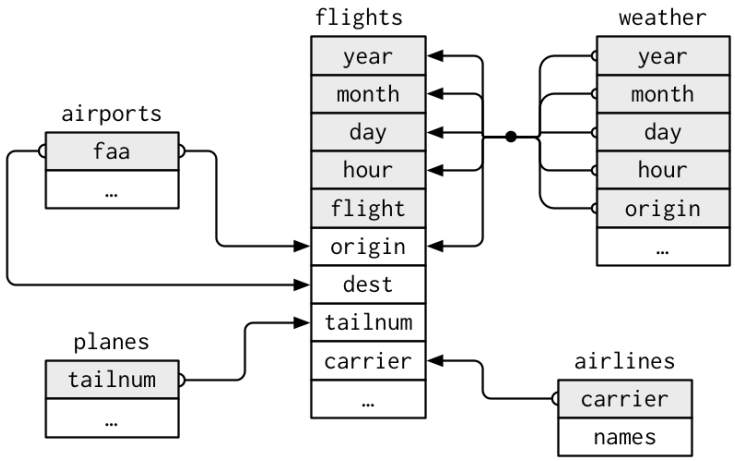
- `planes` gives information about each plane, identified by its `tailnum`:

```
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>
```

- `weather` gives the weather at each NYC airport for each hour:

```
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>
```

One way to show the relationships between the different tables is with a drawing:



This diagram is a little overwhelming, but it's simple compared to some you'll see in the wild! The key to understanding diagrams like this is to remember each relation always concerns a pair of tables. You don't need to understand the whole thing; you just need to understand the chain of relations between the tables that you are interested in.

For **nycflights13**:

- `flights` connects to `planes` via a single variable, `tailnum`.
- `flights` connects to `airlines` through the `carrier` variable.
- `flights` connects to `airports` in two ways: via the `origin` and `dest` variables.
- `flights` connects to `weather` via `origin` (the location), and `year`, `month`, `day`, and `hour` (the time).

Exercises

1. Imagine you wanted to draw (approximately) the route each plane flies from its origin to its destination. What variables would you need? What tables would you need to combine?
2. I forgot to draw the relationship between `weather` and `airports`. What is the relationship and how should it appear in the diagram?

3. `weather` only contains information for the origin (NYC) airports. If it contained weather records for all airports in the USA, what additional relation would it define with `flights`?
4. We know that some days of the year are "special," and fewer people than usual fly on them. How might you represent that data as a data frame? What would be the primary keys of that table? How would it connect to the existing tables?

Keys

The variables used to connect each pair of tables are called *keys*. A key is a variable (or set of variables) that uniquely identifies an observation. In simple cases, a single variable is sufficient to identify an observation. For example, each plane is uniquely identified by its `tailnum`. In other cases, multiple variables may be needed. For example, to identify an observation in `weather` you need five variables: `year`, `month`, `day`, `hour`, and `origin`.

There are two types of keys:

- A *primary key* uniquely identifies an observation in its own table. For example, `planes$tailnum` is a primary key because it uniquely identifies each plane in the `planes` table.
- A *foreign key* uniquely identifies an observation in another table. For example, `flights$tailnum` is a foreign key because it appears in the `flights` table where it matches each flight to a unique plane.

A variable can be both a primary key *and* a foreign key. For example, `origin` is part of the `weather` primary key, and is also a foreign key for the `airport` table.

Once you've identified the primary keys in your tables, it's good practice to verify that they do indeed uniquely identify each observation. One way to do that is to `count()` the primary keys and look for entries where `n` is greater than one:

```
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>
```

Sometimes a table doesn't have an explicit primary key: each row is an observation, but no combination of variables reliably identifies it. For example, what's the primary key in the `flights` table? You might think it would be the date plus the flight or tail number, but neither of those are unique:

```
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
```

When starting to work with this data, I had naively assumed that each flight number would be only used once per day: that would make it much easier to communicate problems with a specific flight. Unfortunately that is not the case! If a table lacks a primary key, it's sometimes useful to add one with `mutate()` and `row_number()`. That makes it easier to match observations if you've done some filtering and want to check back in with the original data. This is called a *surrogate key*.

A primary key and the corresponding foreign key in another table form a *relation*. Relations are typically one-to-many. For example, each flight has one plane, but each plane has many flights. In other data, you'll occasionally see a 1-to-1 relationship. You can think of this as a special case of 1-to-many. You can model many-to-many relations with a many-to-1 relation plus a 1-to-many relation. For example, in this data there's a many-to-many relationship between airlines and airports: each airline flies to many airports; each airport hosts many airlines.

Exercises

1. Add a surrogate key to `flights`.
2. Identify the keys in the following datasets:
   a. `Lahman::Batting`
   b. `babynames::babynames`
   c. `nasaweather::atmos`
   d. `fueleconomy::vehicles`
   e. `ggplot2::diamonds`

   (You might need to install some packages and read some documentation.)
3. Draw a diagram illustrating the connections between the `Batting`, `Master`, and `Salaries` tables in the **Lahman** package. Draw another diagram that shows the relationship between `Master`, `Managers`, and `AwardsManagers`.

   How would you characterize the relationship between the `Batting`, `Pitching`, and `Fielding` tables?

Mutating Joins

The first tool we'll look at for combining a pair of tables is the *mutating join*. A mutating join allows you to combine variables from two tables. It first matches observations by their keys, then copies across variables from one table to the other.

Like `mutate()`, the join functions add variables to the right, so if you have a lot of variables already, the new variables won't get printed out. For these examples, we'll make it easier to see what's going on in the examples by creating a narrower dataset:

```
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
```

(Remember, when you're in RStudio, you can also use `View()` to avoid this problem.)

Imagine you want to add the full airline name to the `flights2` data. You can combine the `airlines` and `flights2` data frames with `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>
```

The result of joining airlines to `flights2` is an additional variable: `name`. This is why I call this type of join a mutating join. In this case, you could have got to the same place using `mutate()` and R's base subsetting:

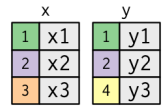
```
flights2 %>%
  select(-origin, -dest) %>%
  mutate(name = airlines$name[match(carrier, airlines$carrier)])
#> # A tibble: 336,776 × 7
#>    year month   day  hour tailnum carrier
#>   <int> <int> <int> <dbl>   <chr>   <chr>
#> 1  2013     1     1     5  N14228      UA
#> 2  2013     1     1     5  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>
```

But this is hard to generalize when you need to match multiple variables, and takes close reading to figure out the overall intent.

The following sections explain, in detail, how mutating joins work. You'll start by learning a useful visual representation of joins. We'll then use that to explain the four mutating join functions: the inner join, and the three outer joins. When working with real data, keys don't always uniquely identify observations, so next we'll talk about what happens when there isn't a unique match. Finally, you'll learn how to tell **dplyr** which variables are the keys for a given join.

Understanding Joins

To help you learn how joins work, I'm going to use a visual representation:

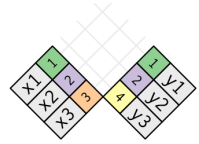



```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
```

```
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```

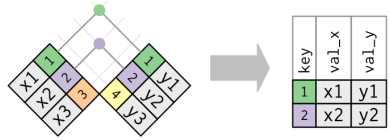
The colored column represents the "key" variable: these are used to match the rows between the tables. The gray column represents the "value" column that is carried along for the ride. In these examples I'll show a single key variable and single value variable, but the idea generalizes in a straightforward way to multiple keys and multiple values.

A join is a way of connecting each row in `x` to zero, one, or more rows in `y`. The following diagram shows each potential match as an intersection of a pair of lines:



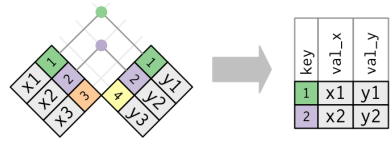
(If you look closely, you might notice that we've switched the order of the key and value columns in `x`. This is to emphasize that joins match based on the key; the value is just carried along for the ride.)

In an actual join, matches will be indicated with dots. The number of dots = the number of matches = the number of rows in the output.



Inner Join

The simplest type of join is the *inner join*. An inner join matches pairs of observations whenever their keys are equal:



(To be precise, this is an inner *equijoin* because the keys are matched using the equality operator. Since most joins are equijoins we usually drop that specification.)

The output of an inner join is a new data frame that contains the key, the `x` values, and the `y` values. We use `by` to tell **dplyr** which variable is the key:

```
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
```

The most important property of an inner join is that unmatched rows are not included in the result. This means that generally inner joins are usually not appropriate for use in analysis because it's too easy to lose observations.

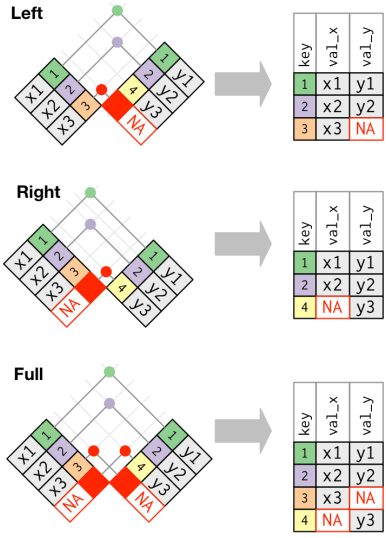
Outer Joins

An inner join keeps observations that appear in both tables. An *outer join* keeps observations that appear in at least one of the tables. There are three types of outer joins:

- A *left join* keeps all observations in `x`.
- A *right join* keeps all observations in `y`.
- A *full join* keeps all observations in `x` and `y`.

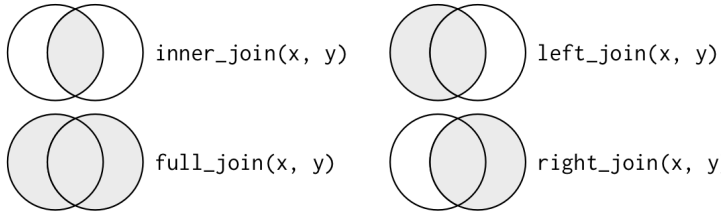
These joins work by adding an additional "virtual" observation to each table. This observation has a key that always matches (if no other key matches), and a value filled with `NA`.

Graphically, that looks like:



The most commonly used join is the left join: you use this whenever you look up additional data from another table, because it preserves the original observations even when there isn't a match. The left join should be your default join: use it unless you have a strong reason to prefer one of the others.

Another way to depict the different types of joins is with a Venn diagram:

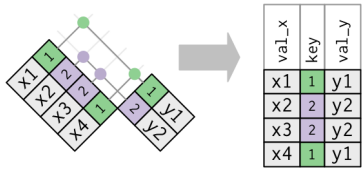


However, this is not a great representation. It might jog your memory about which join preserves the observations in which table, but it suffers from a major limitation: a Venn diagram can't show what happens when keys don't uniquely identify an observation.

Duplicate Keys

So far all the diagrams have assumed that the keys are unique. But that's not always the case. This section explains what happens when the keys are not unique. There are two possibilities:

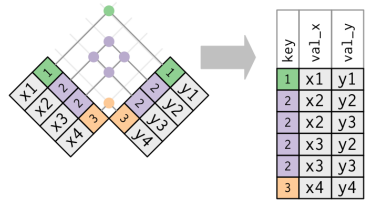
- One table has duplicate keys. This is useful when you want to add in additional information as there is typically a one-to-many relationship:



  Note that I've put the key column in a slightly different position in the output. This reflects that the key is a primary key in `y` and a foreign key in `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
```

- Both tables have duplicate keys. This is usually an error because in neither table do the keys uniquely identify an observation. When you join duplicated keys, you get all possible combinations, the Cartesian product:




```
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
```

Defining the Key Columns

So far, the pairs of tables have always been joined by a single variable, and that variable has the same name in both tables. That constraint was encoded by `by = "key"`. You can use other values for `by` to connect the tables in other ways:

- The default, `by = NULL`, uses all variables that appear in both tables, the so-called *natural* join. For example, the flights and weather tables match on their common variables: `year`, `month`, `day`, `hour`, and `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>
```

- A character vector, `by = "x"`. This is like a natural join, but uses only some of the common variables. For example, `flights` and `planes` have `year` variables, but they mean different things so we only want to join by `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>
```

  Note that the `year` variables (which appear in both input data frames, but are not constrained to be equal) are disambiguated in the output with a suffix.

- A named character vector: `by = c("a" = "b")`. This will match variable `a` in table `x` to variable `b` in table `y`. The variables from `x` will be used in the output.

For example, if we want to draw a map we need to combine the flights data with the airports data, which contains the location (`lat` and `long`) of each airport. Each flight has an origin and destination `airport`, so we need to specify which one we want to join to:

```
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>
```

Exercises

1. Compute the average delay by destination, then join on the `airports` data frame so you can show the spatial distribution of delays. Here's an easy way to draw a map of the United States:

```
airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
    borders("state") +
```

```
  geom_point() +
  coord_quickmap()
```

(Don't worry if you don't understand what `semi_join()` does—you'll learn about it next.)

You might want to use the `size` or `color` of the points to display the average delay for each airport.

2. Add the location of the origin *and* destination (i.e., the `lat` and `lon`) to `flights`.
3. Is there a relationship between the age of a plane and its delays?
4. What weather conditions make it more likely to see a delay?
5. What happened on June 13, 2013? Display the spatial pattern of delays, and then use Google to cross-reference with the weather.

Other Implementations

`base::merge()` can perform all four types of mutating join:

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)`

The advantages of the specific **dplyr** verbs is that they more clearly convey the intent of your code: the difference between the joins is really important but concealed in the arguments of `merge()`. **dplyr**'s joins are considerably faster and don't mess with the order of the rows.

SQL is the inspiration for **dplyr**'s conventions, so the translation is straightforward:

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)`

dplyr	SQL
`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)`

Note that "`INNER`" and "`OUTER`" are optional, and often omitted.

Joining different variables between the tables, e.g., `inner_join(x, y, by = c("a" = "b"))`, uses a slightly different syntax in SQL: `SELECT * FROM x INNER JOIN y ON x.a = y.b`. As this syntax suggests, SQL supports a wider range of join types than **dplyr** because you can connect the tables using constraints other than equality (sometimes called non-equijoins).

Filtering Joins

Filtering joins match observations in the same way as mutating joins, but affect the observations, not the variables. There are two types:

- `semi_join(x, y)` *keeps* all observations in `x` that have a match in `y`.
- `anti_join(x, y)` *drops* all observations in `x` that have a match in `y`.

Semi-joins are useful for matching filtered summary tables back to the original rows. For example, imagine you've found the top-10 most popular destinations:

```
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
```

Now you want to find each flight that went to one of those destinations. You could construct a filter yourself:

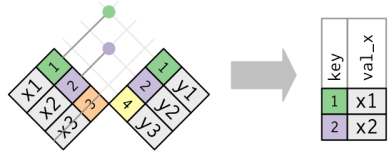
```
flights %>%
  filter(dest %in% top_dest$dest)
#> # A tibble: 141,145 × 19
#>    year month   day dep_time sched_dep_time dep_delay
#>   <int> <int> <int>    <int>          <int>     <dbl>
#> 1  2013     1     1      542            540         2
#> 2  2013     1     1      554            600        -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>
```

But it's difficult to extend that approach to multiple variables. For example, imagine that you'd found the 10 days with the highest average delays. How would you construct the filter statement that used `year`, `month`, and `day` to match it back to `flights`?

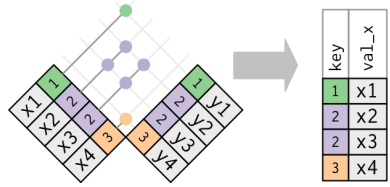
Instead you can use a semi-join, which connects the two tables like a mutating join, but instead of adding new columns, only keeps the rows in `x` that have a match in `y`:

```
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>
```

Graphically, a semi-join looks like this:



Only the existence of a match is important; it doesn't matter which observation is matched. This means that filtering joins never duplicate rows like mutating joins do:



The inverse of a semi-join is an anti-join. An anti-join keeps the rows that *don't* have a match:

Anti-joins are useful for diagnosing join mismatches. For example, when connecting `flights` and `planes`, you might be interested to know that there are many `flights` that don't have a match in `planes`:

```
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
```

Exercises

1. What does it mean for a flight to have a missing `tailnum`? What do the tail numbers that don't have a matching record in `planes` have in common? (Hint: one variable explains ~90% of the problems.)
2. Filter flights to only show flights with planes that have flown at least 100 flights.
3. Combine `fueleconomy::vehicles` and `fueleconomy::common` to find only the records for the most common models.
4. Find the 48 hours (over the course of the whole year) that have the worst delays. Cross-reference it with the `weather` data. Can you see any patterns?
5. What does `anti_join(flights, airports, by = c("dest" = "faa"))` tell you? What does `anti_join(airports, flights, by = c("faa" = "dest"))` tell you?
6. You might expect that there's an implicit relationship between plane and airline, because each plane is flown by a single airline. Confirm or reject this hypothesis using the tools you've learned in the preceding section.

Join Problems

The data you've been working with in this chapter has been cleaned up so that you'll have as few problems as possible. Your own data is unlikely to be so nice, so there are a few things that you should do with your own data to make your joins go smoothly:

1. Start by identifying the variables that form the primary key in each table. You should usually do this based on your understanding of the data, not empirically by looking for a combination of variables that give a unique identifier. If you just look for variables without thinking about what they mean, you might get (un)lucky and find a combination that's unique in your current data but the relationship might not be true in general.

For example, the altitude and longitude uniquely identify each airport, but they are not good identifiers!

```
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. Check that none of the variables in the primary key are missing. If a value is missing then it can't identify an observation!
3. Check that your foreign keys match primary keys in another table. The best way to do this is with an `anti_join()`. It's common for keys not to match because of data entry errors. Fixing these is often a lot of work.

   If you do have missing keys, you'll need to be thoughtful about your use of inner versus outer joins, carefully considering whether or not you want to drop rows that don't have a match.

Be aware that simply checking the number of rows before and after the join is not sufficient to ensure that your join has gone smoothly. If you have an inner join with duplicate keys in both tables, you might get unlucky as the number of dropped rows might exactly equal the number of duplicated rows!

Set Operations

The final type of two-table verb are the set operations. Generally, I use these the least frequently, but they are occasionally useful when you want to break a single complex filter into simpler pieces. All these operations work with a complete row, comparing the values of every variable. These expect the `x` and `y` inputs to have the same variables, and treat the observations like sets:

`intersect(x, y)`
: Return only observations in both `x` and `y`.

`union(x, y)`
: Return unique observations in `x` and `y`.

`setdiff(x, y)`
: Return observations in `x`, but not in `y`.

Given this simple data:

```
df1 <- tribble(
  ~x, ~y,
   1,  1,
   2,  1
)
df2 <- tribble(
  ~x, ~y,
   1,  1,
   1,  2
)
```

The four possibilities are:

```
intersect(df1, df2)
#> # A tibble: 1 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     1

# Note that we get 3 rows, not 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
```