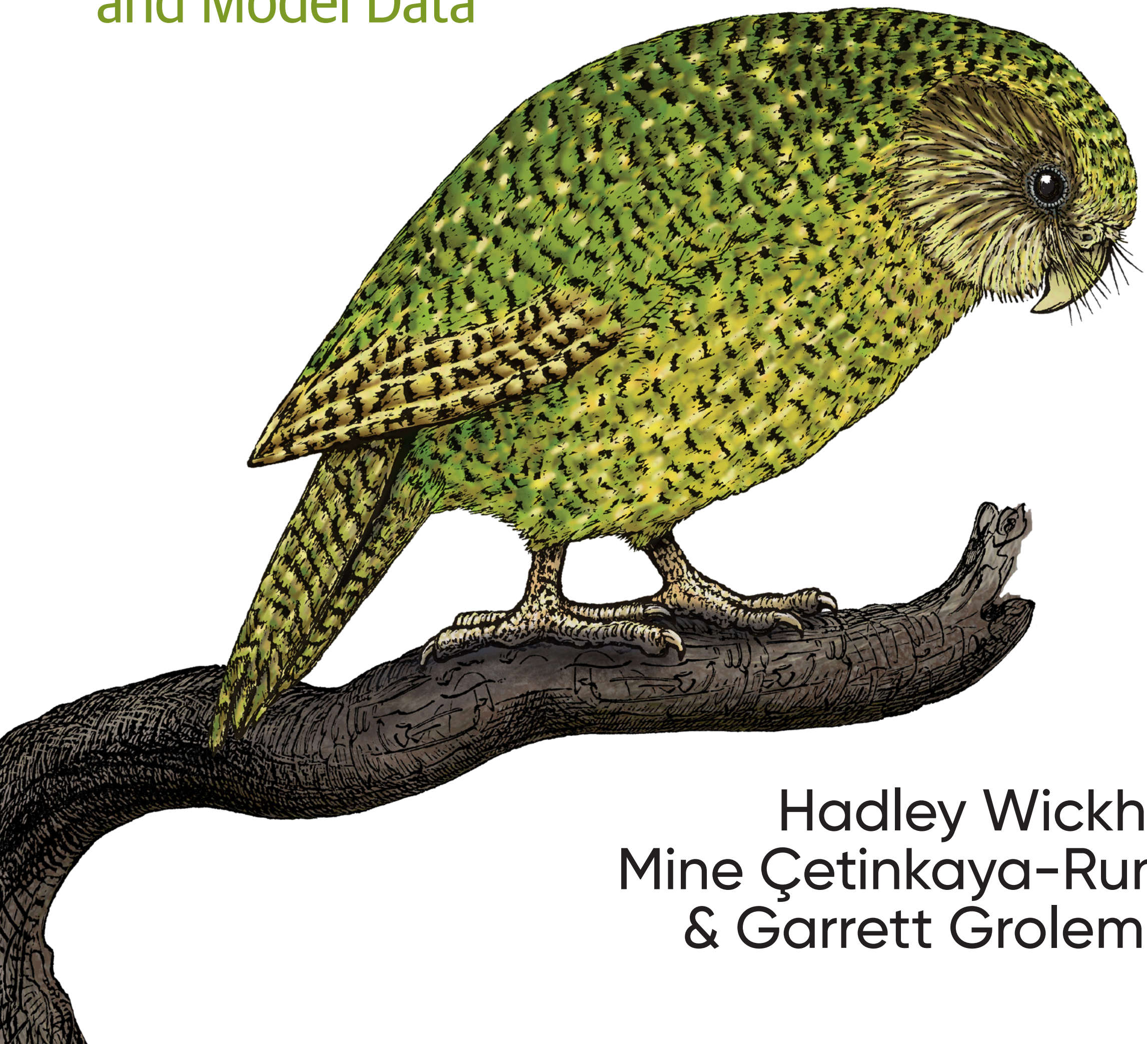
O'REILLY®

Second Edition

R for Data Science

Import, Tidy, Transform, Visualize, and Model Data



Hadley Wickham,
Mine Çetinkaya-Rundel
& Garrett Grolemund

O'REILLY®

R for Data Science

Use R to turn data into insight, knowledge, and understanding. With this practical book, aspiring data scientists will learn how to do data science with R and RStudio, along with the tidyverse—a collection of R packages designed to work together to make data science fast, fluent, and fun. Even if you have no programming experience, this updated edition will have you doing data science quickly.

You'll learn how to import, transform, and visualize your data and communicate the results. And you'll get a complete, big-picture understanding of the data science cycle and the basic tools you need to manage the details. Updated for the latest tidyverse features and best practices, new chapters show you how to get data from spreadsheets, databases, and websites. Exercises help you practice what you've learned along the way.

You'll understand how to:

- **Visualize:** Create plots for data exploration and communication of results
- **Transform:** Discover variable types and the tools to work with them
- **Import:** Get data into R and in a form convenient for analysis
- **Program:** Learn R tools for solving data problems with greater clarity and ease
- **Communicate:** Integrate prose, code, and results with Quarto

> "This is an astonishingly good update to a world-leading guide to doing data science with R. Everyone who works with data should read it!"
>
> **—Emma Rand**
> University of York, UK

Hadley Wickham is chief scientist at Posit and a member of the R Foundation. He builds computational and cognitive tools that make data science easier, faster, and more fun.

Mine Çetinkaya-Rundel is professor of the practice and director of undergraduate studies at the Department of Statistical Science at Duke University. She's also a developer educator at Posit.

Garrett Grolemund is the author of *Hands-On Programming with R* and director of learning at Posit.

DATA

US $79.99 CAN $99.99
ISBN: 978-1-492-09740-2

9 781492 097402 57999

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

2ND EDITION

R for Data Science

*Import, Tidy, Transform, Visualize,
and Model Data*

*Hadley Wickham, Mine Çetinkaya-Rundel,
and Garrett Grolemund*

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

R for Data Science

by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Melissa Potter
Production Editor: Ashley Stussy
Copyeditor: Kim Wimpsett
Proofreader: Charles Roumeliotis

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

June 2023: First Edition

Revision History for the Second Edition

2023-06-07: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492097402* for release details.

The views expressed in this work are those of the authors and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-09740-2

[LSI]

Table of Contents

Part II. Visualize

Part III. Transform

Part IV. Import

Part V. Program

Part VI. Communicate

Introduction

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

Preface to the Second Edition

Welcome to the second edition of *R for Data Science (R4DS)*! This is a major reworking of the first edition, removing material we no longer think is useful, adding material we wish we included in the first edition, and generally updating the text and code to reflect changes in best practices. We're also very excited to welcome a new co-author: Mine Çetinkaya-Rundel, a noted data science educator and one of our colleagues at Posit (the company formerly known as RStudio).

A brief summary of the biggest changes follows:

- The first part of the book has been renamed to "Whole Game." The goal of this section is to give you the rough details of the "whole game" of data science before we dive into the details.
- The second part of the book is "Visualize." This part gives data visualization tools and best practices a more thorough coverage compared to the first edition. The best place to get all the details is still the ggplot2 book, but now R4DS covers more of the most important techniques.
- The third part of the book is now called "Transform" and gains new chapters on numbers, logical vectors, and missing values. These were previously parts of the data transformation chapter but needed much more room to cover all the details.

- The fourth part of the book is called "Import." It's a new set of chapters that goes beyond reading flat text files to working with spreadsheets, getting data out of databases, working with big data, rectangling hierarchical data, and scraping data from websites.
- The "Program" part remains but has been rewritten from top to bottom to focus on the most important parts of function writing and iteration. Function writing now includes details on how to wrap tidyverse functions (dealing with the challenges of tidy evaluation), since this has become much easier and more important over the last few years. We've added a new chapter on important base R functions that you're likely to see in wild-caught R code.
- The "Modeling" part has been removed. We never had enough room to fully do modeling justice, and there are now much better resources available. We generally recommend using the tidymodels packages and reading *Tidy Modeling with R* by Max Kuhn and Julia Silge (O'Reilly).
- The "Communicate" part remains but has been thoroughly updated to feature Quarto instead of R Markdown. This edition of the book has been written in Quarto, and it's clearly the tool of the future.

What You Will Learn

Data science is a vast field, and there's no way you can master it all by reading a single book. This book aims to give you a solid foundation in the most important tools and enough knowledge to find the resources to learn more when necessary. Our model of the steps of a typical data science project looks something like Figure I-1.

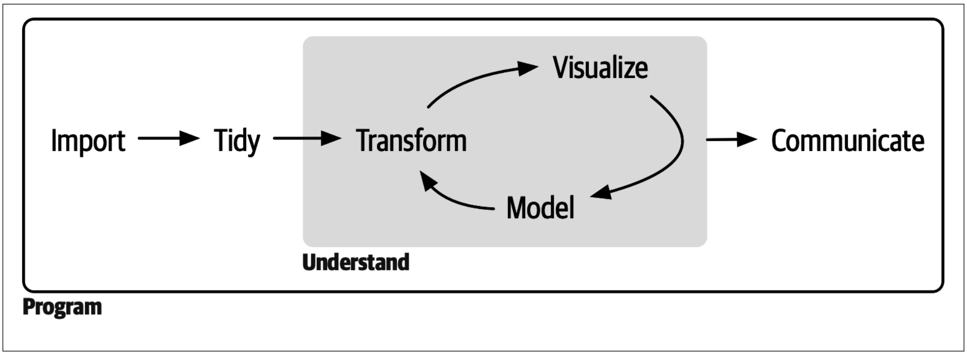


Figure I-1. In our model of the data science process, you start with data import and tidying. Next, you understand your data with an iterative cycle of transforming, visualizing, and modeling. You finish the process by communicating your results to other humans.

First, you must *import* your data into R. This typically means that you take data stored in a file, database, or web application programming interface (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 how 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 efforts on answering questions about the data, not fighting to get the data into the right form for different functions.

Once you have tidy data, a common next step is to *transform* it. Transformation includes narrowing in on observations of interest (such as all people in one city or all data from the last year), creating new variables that are functions of existing variables (such as computing speed from distance and time), and calculating a set of summary statistics (such as 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. They have complementary strengths and weaknesses, so any real data analysis will iterate between them many times.

Visualization is a fundamentally human activity. A good visualization will show you things 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 that you need to collect different data. Visualizations can surprise you, but they 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 fundamentally mathematical or computational tools, 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 nearly every part of a data science project. You don't need to be an expert programmer to be a successful 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 they're not enough for most projects. There's a rough 80/20 rule at play: you can tackle about 80% of every project using the tools 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, learning data importing and tidying first 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 the effort.

Within each chapter, we try to adhere to a consistent 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. Although it can be tempting to skip the exercises, there's no better way to learn than by practicing on real problems.

What You Won't Learn

There are several 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.

Modeling

Modeling is super important for data science, but it's a big topic, and unfortunately, we just don't have the space to give it the coverage it deserves here. To learn more about modeling, we highly recommend *Tidy Modeling with R* by our colleagues Max Kuhn and Julia Silge (O'Reilly). This book will teach you the tidymodels family of packages, which, as you might guess from the name, share many conventions with the tidyverse packages we use in this book.

Big Data

This book proudly and primarily 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 the majority of this book will easily handle hundreds of megabytes of data, and with a bit of care, you can typically use them to work with a few gigabytes of data. We'll also show you how to get data out of databases and parquet files, both of which are often used to store big data. You won't necessarily be able to work with the entire dataset, but that's not a problem because you need only a subset or subsample to answer the question you're interested in.

If you're routinely working with larger data (10–100 GB, say), we recommend learning more about data.table. We don't teach it here because it uses a different interface than the tidyverse and requires you to learn some different conventions. However, it is incredibly faster, and the performance payoff is worth investing some time in learning it if you're working with large data.

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. But we strongly believe that it's best to master one tool at a time, and R is a great place to start.

Prerequisites

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

You need four things 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 reproducible R code. They include reusable functions, documentation that describes how to use them, and sample data.

R

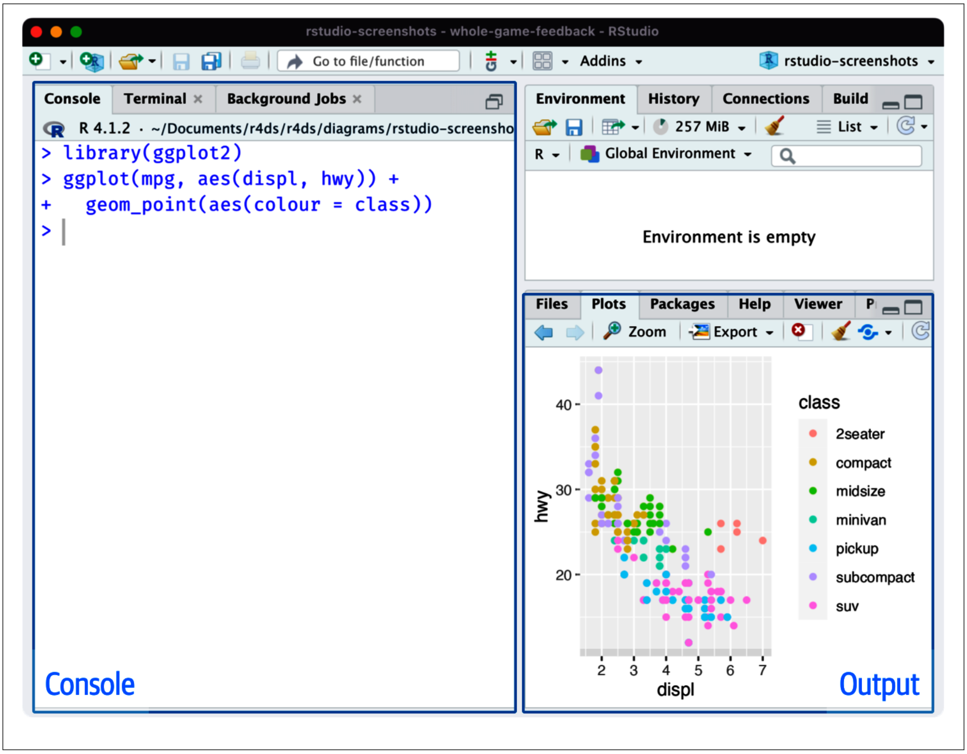
To download R, go to CRAN, the *c*omprehensive *R* *a*rchive *n*etwork. A new major version of R comes out once a year, and there are two to three minor releases each year. It's a good idea to update regularly. Upgrading can be a bit of a hassle, especially for major versions that require you to re-install all your packages, but putting it off only makes it worse. We recommend R 4.2.0 or later for this book.

RStudio

RStudio is an integrated development environment (IDE) for R programming, which you can download from the RStudio download page. RStudio is updated a couple of times a year, and it will automatically let you know when a new version is out, so there's no need to check back. It's a good idea to upgrade regularly to take advantage of the latest and greatest features. For this book, make sure you have at least RStudio 2022.02.0.

When you start RStudio, Figure I-2, you'll see two key regions in the interface: the console pane and the output pane. For now, all you need to know is that you type the R code in the console pane and press Enter to run it. You'll learn more as we go along![1]

[1] If you'd like a comprehensive overview of all of RStudio's features, see the RStudio User Guide.



```
> library(ggplot2)
> ggplot(mpg, aes(displ, hwy)) +
+   geom_point(aes(colour = class))
> |
```

Environment is empty

Figure I-2. The RStudio IDE has two key regions: type R code in the console pane on the left, and look for plots in the output pane on the right.

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. All packages in the tidyverse share a common philosophy of data and R programming and are designed to work together.

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

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

On your 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 on your computer.

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

```
library(tidyverse)
#> ── Attaching core tidyverse packages ───────────────────── tidyverse 2.0.0 ──
#> ✔ dplyr     1.1.0.9000     ✔ readr     2.1.4
#> ✔ forcats   1.0.0          ✔ stringr   1.5.0
#> ✔ ggplot2   3.4.1          ✔ tibble    3.1.8
#> ✔ lubridate 1.9.2          ✔ tidyr     1.3.0
#> ✔ purrr     1.0.1
#> ── Conflicts ─────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
#> ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all
#>   conflicts to become errors
```

This tells you that tidyverse loads nine packages: dplyr, forcats, ggplot2, lubridate, purrr, readr, stringr, tibble, and tidyr. These are considered 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 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; it just makes them 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.

We'll use many packages from outside the tidyverse in this book. For example, we'll use the following packages because they provide interesting data sets for us to work with in the process of learning R:

```
install.packages(c("arrow", "babynames", "curl", "duckdb", "gapminder", "ggrepel",
"ggridges", "ggthemes", "hexbin", "janitor", "Lahman", "leaflet", "maps",
"nycflights13", "openxlsx", "palmerpenguins", "repurrrsive", "tidymodels", "writexl"))
```

We'll also use a selection of other packages for one-off examples. You don't need to install them now, just remember that whenever you see an error like this:

```
library(ggrepel)
#> Error in library(ggrepel) : there is no package called 'ggrepel'
```

it means you need to run `install.packages("ggrepel")` to install the package.

Running R Code

The previous section showed you several examples of running R code. The 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, the 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 paste it into the console.

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

- Functions are displayed in a code font and followed by parentheses, like `sum()` or `mean()`.
- Other R objects (such as data or function arguments) are in a code font, without parentheses, like `flights` or `x`.
- Sometimes, to make it clear which 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.

Other Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
    Indicates URLs and email addresses.

`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, keywords, and filenames.

`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 general note.

This element indicates a warning or caution.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

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
800-889-8969 (in the United States or Canada)
707-829-7019 (international or local)
707-829-0104 (fax)
support@oreilly.com
https://www.oreilly.com/about/contact.html

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/r-for-data-science-2e*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*

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

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

Acknowledgments

This book isn't just the product of Hadley, Mine, and Garrett but is the result of many conversations (in person and online) that we've had with many people in the R community. We're incredibly grateful for all the conversations we've had with y'all; thank you so much!

We'd like to thank our technical reviewers for their valuable feedback: Ben Baumer, Lorna Barclay, Richard Cotton, Emma Rand, and Kelly Bodwin.

This book was written in the open, and many people contributed via pull requests. A special thanks to all 259 of you who contributed improvements via GitHub pull requests (in alphabetical order by username): @a-rosenberg, Tim Becker (@a2800276), Abinash Satapathy (@Abinashbunty), Adam Gruer (@adam-gruer), adi pradhan (@adidoit), A. s. (@Adrianzo), Aep Hidyatuloh (@aephidayatuloh), Andrea Gilardi (@agila5), Ajay Deonarine (@ajay-d), @AlanFeder, Daihe Sui (@alansuidaihe), @alberto-agudo, @AlbertRapp, @aleloi, pete (@alonzi), Alex (@ALShum), Andrew M. (@amacfarland), Andrew Landgraf (@andland), @andyhuynh92, Angela Li (@angela-li), Antti Rask (@AnttiRask), LOU Xun (@aquarhead), @ariespirgel, @august-18, Michael Henry (@aviast), Azza Ahmed (@azzaea), Steven Moran (@bambooforest), Brian G. Barkley (@BarkleyBG), Mara Averick (@batpigandme), Oluwafemi OYEDELE (@BB1464), Brent Brewington (@bbrewington), Bill Behrman (@behrman), Ben Herbertson (@benherbertson), Ben Marwick (@benmarwick), Ben Steinberg (@bensteinberg), Benjamin Yeh (@bentyeh), Betul Turkoglu (@betulturkoglu), Brandon Greenwell (@bgreenwell), Bianca Peterson (@BinxiePeterson), Birger Niklas (@BirgerNi), Brett Klamer (@bklamer), @boardtc, Christian (@c-hoh), Caddy (@caddycarine), Camille V Leonard (@camillevleonard), @canovasjm, Cedric Batailler (@cedricbatailler), Christina Wei (@christina-wei), Christian Mongeau (@chrMongeau), Cooper Morris (@coopermor), Colin Gillespie (@csgillespie), Rademeyer Vermaak (@csrvermaak), Chloe Thierstein (@cthierst), Chris Saunders (@ctsa), Abhinav Singh (@curious-abhinav), Curtis Alexander (@curtisalexander), Christian G. Warden (@cwarden), Charlotte Wickham (@cwickham), Kenny Darrell (@darrkj), David Kane (@davidkane9), David (@davidrsch), David Rubinger (@davidrubinger), David Clark (@DDClark), Derwin McGeary (@derwinmcgeary), Daniel Gromer (@dgromer), @Divider85, @djbirke, Danielle Navarro (@djnavarro), Russell Shean (@DOH-RPS1303), Zhuoer Dong (@dongzhuoer), Devin Pastoor (@dpastoor), @DSGeoff, Devarshi Thakkar (@dthakkar09), Julian During (@duju211), Dylan Cashman (@dylancashman), Dirk Eddelbuettel (@eddelbuettel), Edwin Thoen (@EdwinTh), Ahmed El-Gabbas (@elgabbas), Henry Webel (@enryH), Ercan Karadas (@ercan7), Eric Kitaif (@EricKit), Eric Watt (@ericwatt), Erik Erhardt (@erikerhardt), Etienne B. Racine (@etiennebr), Everett Robinson (@evjrob), @fellennert, Flemming Miguel (@flemmingmiguel), Floris Vanderhaeghe (@florisvdh), @funkybluehen, @gabrivera, Garrick Aden-Buie (@gadenbuie), Peter

Ganong (@ganong123), Gerome Meyer (@GeroVanMi), Gleb Ebert (@gl-eb), Josh Goldberg (@GoldbergData), bahadir cankardes (@gridgrad), Gustav W Delius (@gustavdelius), Hao Chen (@hao-trivago), Harris McGehee (@harrismcgehee), @hendrikweisser, Hengni Cai (@hengnicai), Iain (@Iain-S), Ian Sealy (@iansealy), Ian Lyttle (@ijlyttle), Ivan Krukov (@ivan-krukov), Jacob Kaplan (@jacobkap), Jazz Weisman (@jazzlw), John Blischak (@jdblischak), John D. Storey (@jdstorey), Gregory Jefferis (@jefferis), Jeffrey Stevens (@JeffreyRStevens), 蒋雨蒙 (@JeldorPKU), Jennifer (Jenny) Bryan (@jennybc), Jen Ren (@jenren), Jeroen Janssens (@jeroenjanssens), @jeromecholewa, Janet Wesner (@jilmun), Jim Hester (@jimhester), JJ Chen (@jjchern), Jacek Kolacz (@jkolacz), Joanne Jang (@joannejang), @johannes4998, John Sears (@johnsears), @jonathanflint, Jon Calder (@jonmcalder), Jonathan Page (@jonpage), Jon Harmon (@jonthegeek), JooYoung Seo (@jooyoungseo), Justinas Petuchovas (@jpetuchovas), Jordan (@jrdnbradford), Jeffrey Arnold (@jrnold), Jose Roberto Ayala Solares (@jroberayalas), Joyce Robbins (@jtr13), @juandering, Julia Stewart Lowndes (@jules32), Sonja (@kaetschap), Kara Woo (@karawoo), Katrin Leinweber (@katrinleinweber), Karandeep Singh (@kdpsingh), Kevin Perese (@kevinxperese), Kevin Ferris (@kferris10), Kirill Sevastyanenko (@kirillseva), Jonathan Kitt (@KittJonathan), @koalabearski, Kirill Müller (@krlmlr), Rafał Kucharski (@kucharsky), Kevin Wright (@kwstat), Noah Landesberg (@landesbergn), Lawrence Wu (@lawwu), @lindbrook, Luke W Johnston (@lwjohnst86), Kara de la Marck (@MarckK), Kunal Marwaha (@marwahaha), Matan Hakim (@matanhakim), Matthias Liew (@MatthiasLiew), Matt Wittbrodt (@MattWittbrodt), Mauro Lepore (@maurolepore), Mark Beveridge (@mbeveridge), @mcewenkhundi, mcsnowface, PhD (@mcsnowface), Matt Herman (@mfherman), Michael Boerman (@michaelboerman), Mitsuo Shiota (@mitsuoxv), Matthew Hendrickson (@mjhendrickson), @MJMarshall, Misty Knight-Finley (@mkfin7), Mohammed Hamdy (@mmhamdy), Maxim Nazarov (@mnazarov), Maria Paula Caldas (@mpaulacaldas), Mustafa Ascha (@mustafaascha), Nelson Areal (@nareal), Nate Olson (@nate-d-olson), Nathanael (@nateaff), @nattalides, Ned Western (@NedJWestern), Nick Clark (@nickclark1000), @nickelas, Nirmal Patel (@nirmalpatel), Nischal Shrestha (@nischalshrestha), Nicholas Tierney (@njtierney), Jakub Nowosad (@Nowosad), Nick Pullen (@nstjhp), @olivier6088, Olivier Cailloux (@oliviercailloux), Robin Penfold (@p0bs), Pablo E. Garcia (@pabloedug), Paul Adamson (@padamson), Penelope Y (@penelopeysm), Peter Hurford (@peterhurford), Peter Baumgartner (@petzi53), Patrick Kennedy (@pkq), Pooya Taherkhani (@pooyataher), Y. Yu (@PursuitOfDataScience), Radu Grosu (@radugrosu), Ranae Dietzel (@Ranae), Ralph Straumann (@rastrau), Rayna M Harris (@raynamharris), @ReeceGoding, Robin Gertenbach (@rgertenbach), Jajo (@RIngyao), Riva Quiroga (@rivaquiroga), Richard Knight (@RJHKnight), Richard Zijdeman (@rlzijdeman), @robertchu03, Robin Kohrs (@RobinKohrs), Robin (@Robinlovelace), Emily Robinson (@robinsones), Rob Tenorio (@robtenorio), Rod Mazloomi (@RodAli), Rohan Alexander (@RohanAlexander), Romero Morais (@RomeroBarata), Albert Y. Kim (@rudeboybert), Saghir (@saghirb), Hojjat Salmasian (@salmasian), Jonas (@sauercrowd), Vebash Naidoo (@sciencificity), Seamus McKinsey (@seamus-mckinsey), @seanpwilliams, Luke Smith (@seasmith), Matthew Sedaghatfar (@sedaghatfar), Sebastian Kraus (@sekR4), Sam Firke (@sfirke), Shannon Ellis (@ShanEllis), @shoili, Christian Heinrich (@Shurakai), S'busiso Mkhondwane (@sibusiso16), SM Raiyyan (@sm-raiyyan), Jakob Krigovsky (@sonicdoe), Stephan Koenig (@stephan-koenig), Stephen Balogun (@stephenbalogun), Steven M. Mortimer (@StevenMMortimer), Stéphane Guillou (@stragu), Sulgi Kim (@sulgik), Sergiusz Bleja (@svenski), Tal Galili (@talgalili), Alec Fisher (@Taurenamo), Todd Gerarden (@tgerarden), Tom Godfrey (@thomasggodfrey), Tim Broderick (@timbroderick), Tim Waterhouse (@timwaterhouse), TJ Mahr (@tjmahr), Thomas Klebel (@tklebel), Tom Prior (@tomjamesprior), Terence Teo (@tteo), @twgardner2, Ulrik Lyngs (@ulyngs), Shinya Uryu (@uribo), Martin Van der Linden (@vanderlindenma), Walter Somerville (@waltersom), @werkstattcodes, Will Beasley (@wibeasley), Yihui Xie (@yihui), Yiming (Paul) Li (@yimingli), @yingxingwu, Hiroaki Yutani (@yutannihilation), Yu Yu Aung (@yuyu-aung), Zach Bogart (@zachbogart), @zeal626, and Zeki Akyol (@zekiakyol).

Online Edition

An online version of this book is available at the book's GitHub repository. It will continue to evolve in between reprints of the physical book. The source of the book is available at *https://oreil.ly/Q8z_O*. The book is powered by Quarto, which makes it easy to write books that combine text and executable code.

PART I
Whole Game

Our goal in this part of the book is to give you a rapid overview of the main tools of data science: *importing*, *tidying*, *transforming*, and *visualizing data*, as shown in Figure I-1. We want to show you the "whole game" of data science, giving you just enough of all the major pieces so that you can tackle real, if simple, datasets. The later parts of the book will hit each of these topics in more depth, increasing the range of data science challenges that you can tackle.

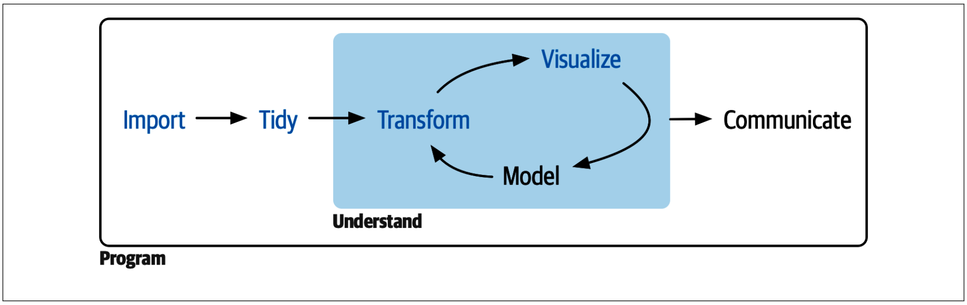


Figure I-1. In this section of the book, you'll learn how to import, tidy, transform, and visualize data.

Four chapters focus on the tools of data science:

- 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.
- In Chapter 5, 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.
- Before you can transform and visualize your data, you need to first get your data into R. In Chapter 7 you'll learn the basics of getting `.csv` files into R.

Nestled among these chapters are four other chapters that focus on your R workflow. In Chapter 2, Chapter 4, and Chapter 6 you'll learn good workflow 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. Finally, Chapter 8 will teach you how to get help and keep learning.

CHAPTER 1
Data Visualization

Introduction

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

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.

This chapter will teach you how to visualize your data using ggplot2. We will start by creating a simple scatterplot and use it to introduce aesthetic mappings and geometric objects—the fundamental building blocks of ggplot2. We will then walk you through visualizing distributions of single variables as well as visualizing relationships between two or more variables. We’ll finish off with saving your plots and troubleshooting tips.

Prerequisites

This chapter focuses on ggplot2, one of the core packages in the tidyverse. To access the datasets, help pages, and functions used in this chapter, load the tidyverse by running:

```
library(tidyverse)
#> ── Attaching core tidyverse packages ──────────────────── tidyverse 2.0.0 ──
#> ✔ dplyr     1.1.0.9000     ✔ readr     2.1.4
#> ✔ forcats   1.0.0          ✔ stringr   1.5.0
#> ✔ ggplot2   3.4.1          ✔ tibble    3.1.8
#> ✔ lubridate 1.9.2          ✔ tidyr     1.3.0
#> ✔ purrr     1.0.1
#> ── Conflicts ──────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
#> ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all
#>   conflicts to become errors
```

That one line of code loads the core tidyverse, the 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).[1]

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

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

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

In addition to tidyverse, we will use the palmerpenguins package, which includes the `penguins` dataset containing body measurements for penguins on three islands in the Palmer Archipelago, and the ggthemes package, which offers a colorblind safe color palette.

```
library(palmerpenguins)
library(ggthemes)
```

First Steps

Do penguins with longer flippers weigh more or less than penguins with shorter flippers? You probably already have an answer, but try to make your answer precise. What does the relationship between flipper length and body mass look like? Is it positive? Negative? Linear? Nonlinear? Does the relationship vary by the species of the penguin? How about by the island where the penguin lives? Let's create visualizations that we can use to answer these questions.

The penguins Data Frame

You can test your answers to these questions with the `penguins` data frame found in palmerpenguins (aka `palmerpenguins::penguins`). A data frame is a rectangular collection of variables (in the columns) and observations (in the rows). `penguins` contains 344 observations collected and made available by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER.[2]

[1] You can eliminate that message and force conflict resolution to happen on demand by using the conflicted package, which becomes more important as you load more packages. You can learn more about conflicted on the package website.

[2] Horst AM, Hill AP, Gorman KB (2020). palmerpenguins: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. *https://oreil.ly/ncwc5*. doi: 10.5281/zenodo.3960218.

To make the discussion easier, let's define some terms:

Variable
: A quantity, quality, or property that you can measure.

Value
: The state of a variable when you measure it. The value of a variable may change from measurement to measurement.

Observation
: 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. We'll sometimes refer to an observation as a *data point*.

Tabular data
: 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.

In this context, a variable refers to an attribute of all the penguins, and an observation refers to all the attributes of a single penguin.

Type the name of the data frame in the console, and R will print a preview of its contents. Note that it says `tibble` on top of this preview. In the tidyverse, we use special data frames called *tibbles* that you will learn about soon.

```
penguins
#> # A tibble: 344 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
#>   <fct>   <fct>              <dbl>         <dbl>             <int>
#> 1 Adelie  Torgersen           39.1          18.7               181
#> 2 Adelie  Torgersen           39.5          17.4               186
#> 3 Adelie  Torgersen           40.3          18                 195
#> 4 Adelie  Torgersen           NA            NA                  NA
#> 5 Adelie  Torgersen           36.7          19.3               193
#> 6 Adelie  Torgersen           39.3          20.6               190
#> # … with 338 more rows, and 3 more variables: body_mass_g <int>, sex <fct>,
#> #   year <int>
```

This data frame contains eight columns. For an alternative view, where you can see all variables and the first few observations of each variable, use `glimpse()`. Or, if you're in RStudio, run `View(penguins)` to open an interactive data viewer.

```
glimpse(penguins)
#> Rows: 344
#> Columns: 8
#> $ species           <fct> Adelie, Adelie, Adelie, Adelie, Adelie, Adelie, A…
#> $ island            <fct> Torgersen, Torgersen, Torgersen, Torgersen, Torge…
#> $ bill_length_mm    <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34.…
#> $ bill_depth_mm     <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18.…
#> $ flipper_length_mm <int> 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, …
#> $ body_mass_g       <int> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 347…
#> $ sex               <fct> male, female, female, NA, female, male, female, m…
#> $ year              <int> 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, 2…
```

Among the variables in `penguins` are:

`species`
A penguin's species (Adelie, Chinstrap, or Gentoo)

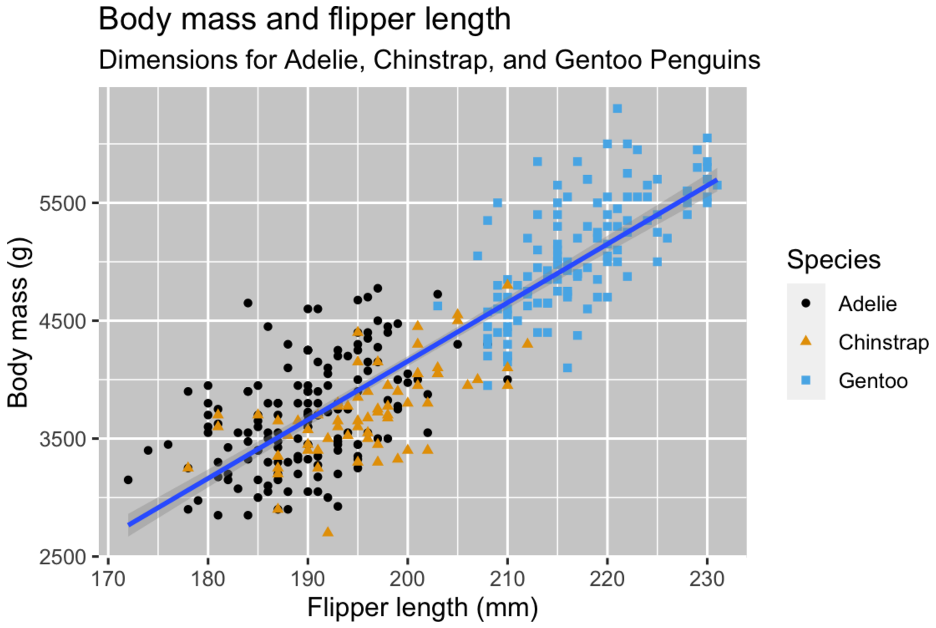
`flipper_length_mm`
The length of a penguin's flipper, in millimeters

`body_mass_g`
The body mass of a penguin, in grams

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

Ultimate Goal

Our ultimate goal in this chapter is to re-create the following visualization displaying the relationship between flipper lengths and body masses of these penguins, taking into consideration the species of the penguin.



Creating a ggplot

Let's re-create this plot step by step.

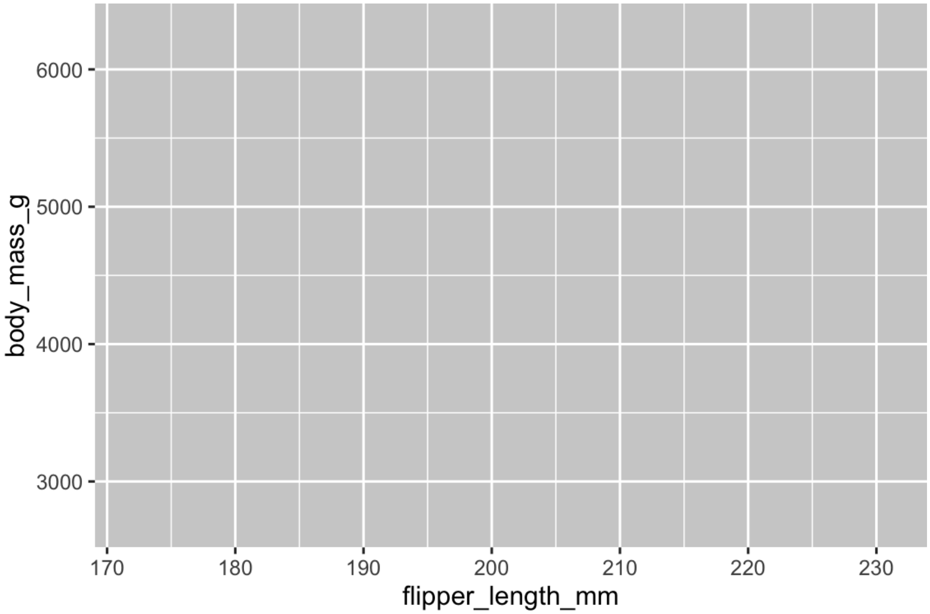
With ggplot2, you begin a plot with the function `ggplot()`, defining a plot object that you then add *layers* to. The first argument of `ggplot()` is the dataset to use in the graph, so `ggplot(data = penguins)` creates an empty graph that is primed to display the `penguins` data, but since we haven't told it how to visualize it yet, for now it's empty. This is not a very exciting plot, but you can think of it like an empty canvas where you'll paint the remaining layers of your plot.

```
ggplot(data = penguins)
```

Next, we need to tell `ggplot()` how the information from our data will be visually represented. The `mapping` argument of the `ggplot()` function defines how variables in your dataset are mapped to visual properties (*aesthetics*) of your plot. The `mapping` argument is always defined in the `aes()` function, and the `x` and `y` arguments of `aes()` specify which variables to map to the x- and y-axes. For now, we will map only the flipper length to the `x` aesthetic and body mass to the `y` aesthetic. ggplot2 looks for the mapped variables in the `data` argument, in this case, `penguins`.

The following plot shows the result of adding these mappings.

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
)
```

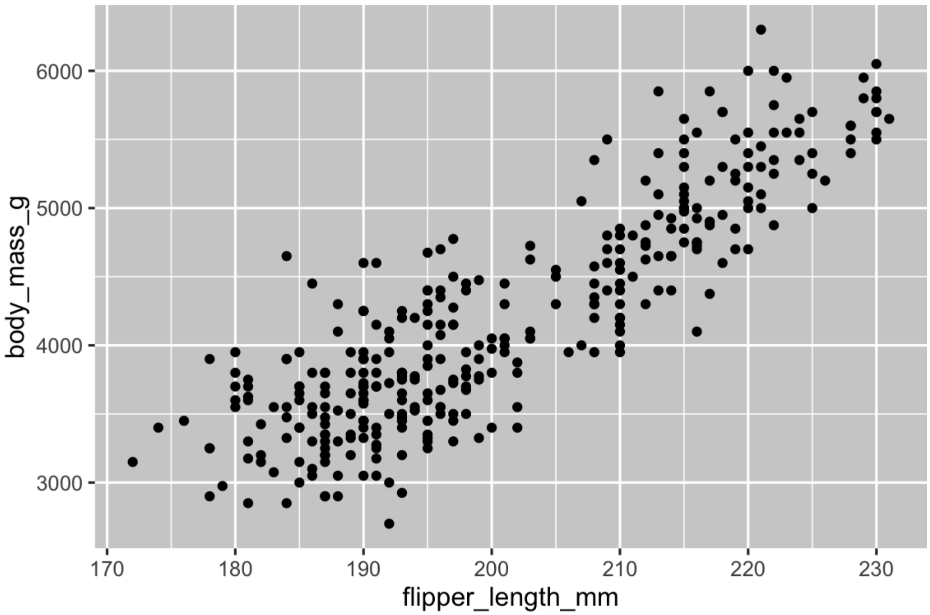


Our empty canvas now has more structure—it's clear where flipper lengths will be displayed (on the x-axis) and where body masses will be displayed (on the y-axis). But the penguins themselves are not yet on the plot. This is because we have not yet articulated, in our code, how to represent the observations from our data frame on our plot.

To do so, we need to define a *geom*: the geometrical object that a plot uses to represent data. These geometric objects are made available in ggplot2 with functions that start with `geom_`. People often describe plots by the type of geom that the plot uses. For example, bar charts use bar geoms (`geom_bar()`), line charts use line geoms (`geom_line()`), boxplots use boxplot geoms (`geom_boxplot()`), scatterplots use point geoms (`geom_point()`), and so on.

The function `geom_point()` adds a layer of points to your plot, which creates a scatterplot. ggplot2 comes with many geom functions, and each adds a different type of layer to a plot. You'll learn a whole bunch of geoms throughout the book, particularly in Chapter 9.

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point()
#> Warning: Removed 2 rows containing missing values (`geom_point()`).
```



Now we have something that looks like what we might think of as a "scatterplot." It doesn't yet match our "ultimate goal" plot, but using this plot we can start answering the question that motivated our exploration: "What does the relationship between flipper length and body mass look like?" The relationship appears to be positive (as flipper length increases, so does body mass), fairly linear (the points are clustered around a line instead of a curve), and moderately strong (there isn't too much scatter around such a line). Penguins with longer flippers are generally larger in terms of their body mass.

Before we add more layers to this plot, let's pause for a moment and review the warning message we got:

> Removed 2 rows containing missing values (`geom_point()`).

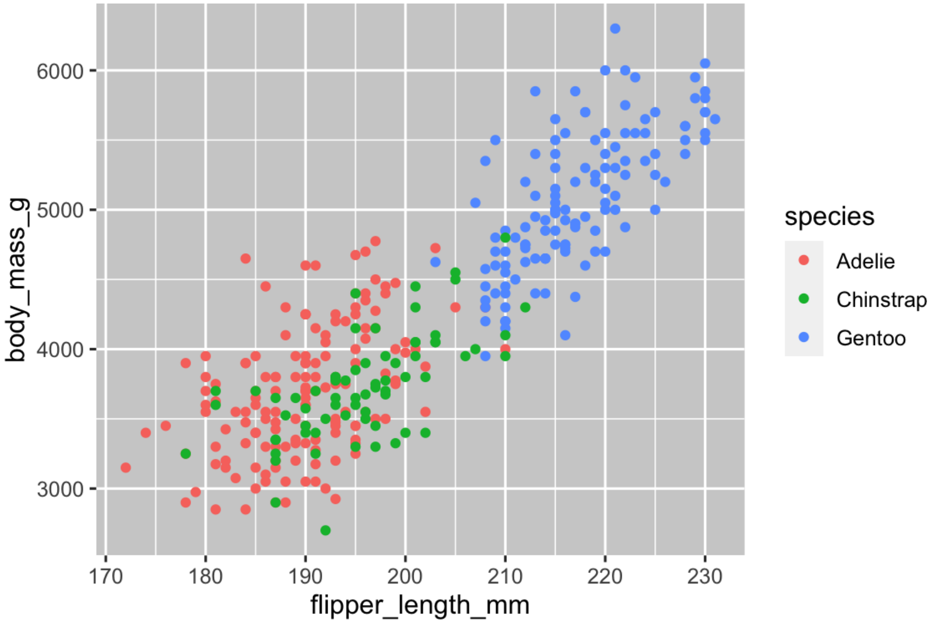
We're seeing this message because there are two penguins in our dataset with missing body mass and/or flipper length values and ggplot2 has no way of representing them on the plot without both of these values. Like R, ggplot2 subscribes to the philosophy that missing values should never silently go missing. This type of warning is probably one of the most common types of warnings you will see when working with real data—missing values are a common issue, and you'll learn more about them throughout the book, particularly in Chapter 18. For the remaining plots in this chapter we will suppress this warning so it's not printed alongside every single plot we make.

Adding Aesthetics and Layers

Scatterplots are useful for displaying the relationship between two numerical variables, but it's always a good idea to be skeptical of any apparent relationship between two variables and ask if there may be other variables that explain or change the nature of this apparent relationship. For example, does the relationship between flipper length and body mass differ by species? Let's incorporate species into our plot and see if this reveals any additional insights into the apparent relationship between these variables. We will do this by representing species with different colored points.

To achieve this, will we need to modify the aesthetic or the geom? If you guessed "in the aesthetic mapping, inside of `aes()`," you're already getting the hang of creating data visualizations with ggplot2! And if not, don't worry. Throughout the book you will make many more ggplots and have many more opportunities to check your intuition as you make them.

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)
) +
  geom_point()
```

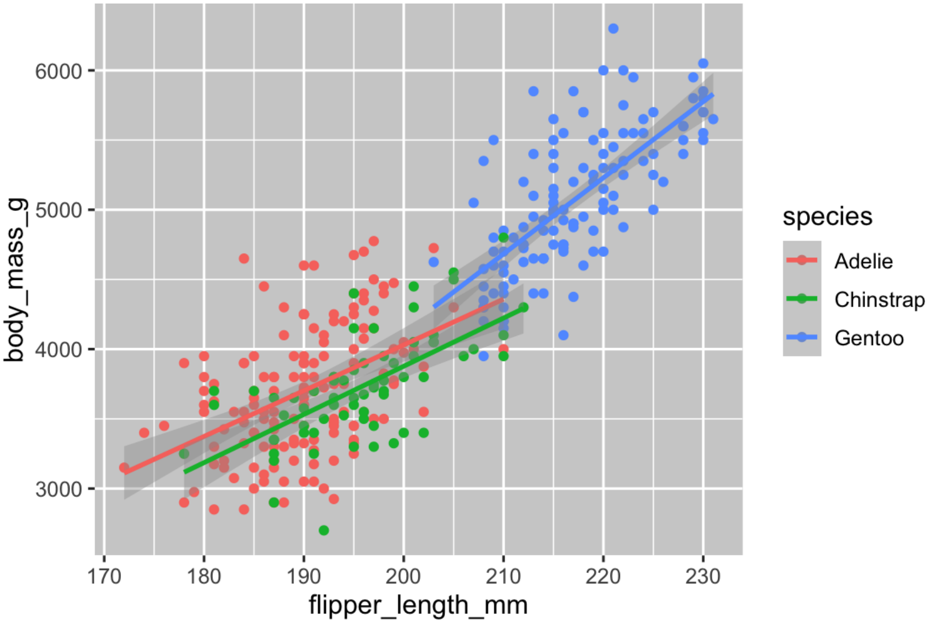


When a categorical variable is mapped to an aesthetic, ggplot2 will automatically assign a unique value of the aesthetic (here a unique color) to each unique level of the variable (each of the three species), a process known as *scaling*. ggplot2 will also add a legend that explains which values correspond to which levels.

Now let's add one more layer: a smooth curve displaying the relationship between body mass and flipper length. Before you proceed, refer to the previous code, and think about how we can add this to our existing plot.

Since this is a new geometric object representing our data, we will add a new geom as a layer on top of our point geom: `geom_smooth()`. And we will specify that we want to draw the line of best fit based on a `l`inear `m`odel with `method = "lm"`.

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)
) +
  geom_point() +
  geom_smooth(method = "lm")
```

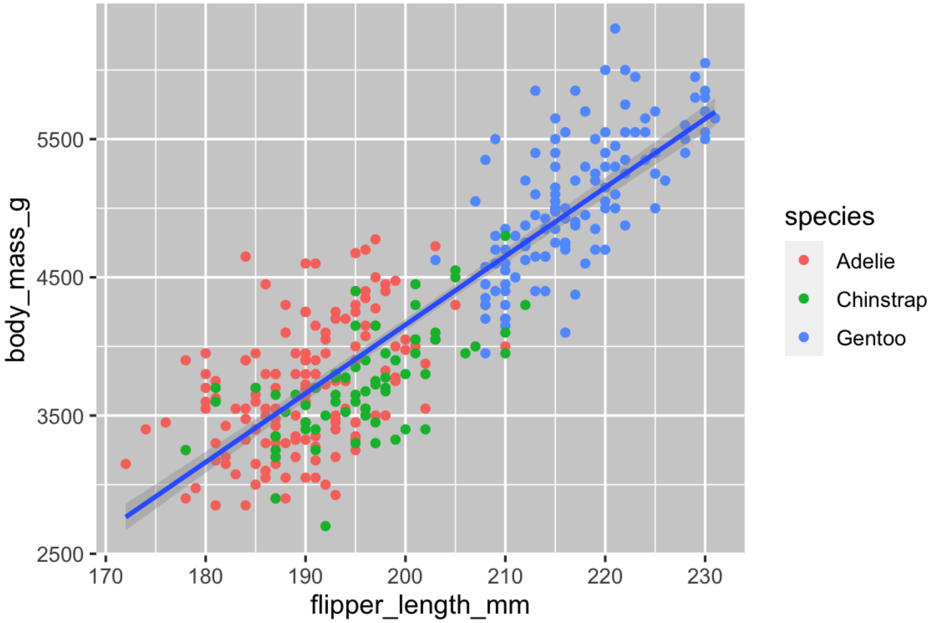


We have successfully added lines, but this plot doesn't look like the plot from "Ultimate Goal" on page 6, which has only one line for the entire dataset as opposed to separate lines for each of the penguin species.

When aesthetic mappings are defined in `ggplot()`, at the *global* level, they're passed down to each of the subsequent geom layers of the plot. However, each geom function in ggplot2 can also take a `mapping` argument, which allows for aesthetic mappings at the *local* level that are added to those inherited from the global level.

Since we want points to be colored based on species but don't want the lines to be separated out for them, we should specify `color = species` for `geom_point()` only.

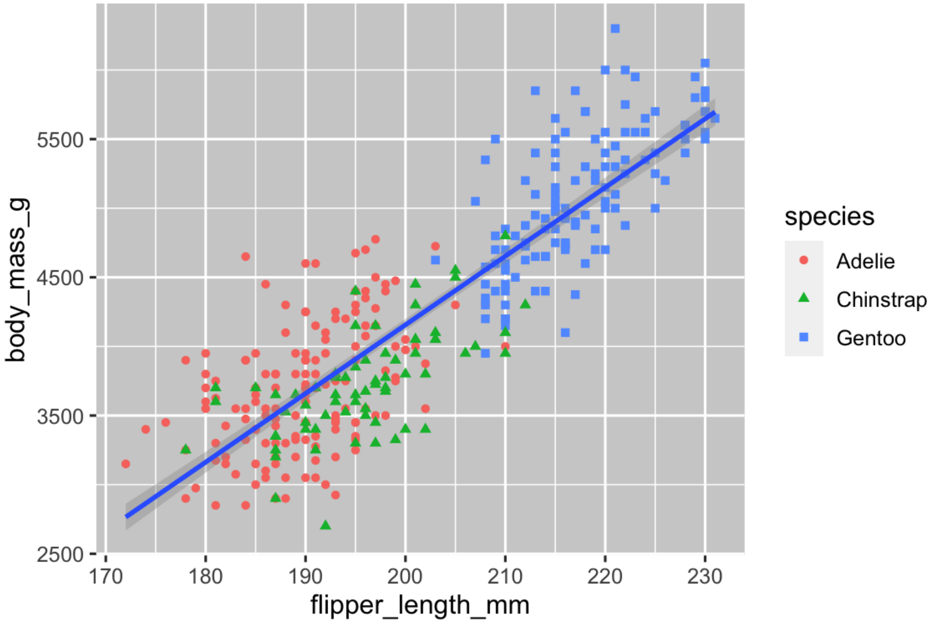
```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(mapping = aes(color = species)) +
  geom_smooth(method = "lm")
```



Voilà! We have something that looks very much like our ultimate goal, though it's not yet perfect. We still need to use different shapes for each species of penguins and improve labels.

It's generally not a good idea to represent information using only colors on a plot, as people perceive colors differently due to color blindness or other color vision differences. Therefore, in addition to color, we can map `species` to the `shape` aesthetic.

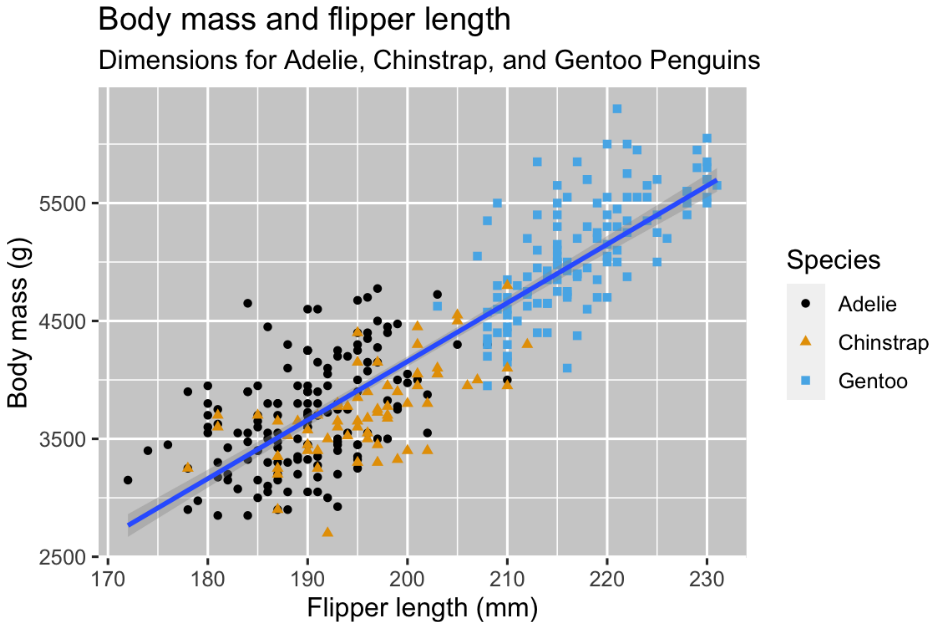
```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(mapping = aes(color = species, shape = species)) +
  geom_smooth(method = "lm")
```



Note that the legend is automatically updated to reflect the different shapes of the points as well.

Finally, we can improve the labels of our plot using the `labs()` function in a new layer. Some of the arguments to `labs()` might be self-explanatory: `title` adds a title, and `subtitle` adds a subtitle to the plot. Other arguments match the aesthetic mappings: `x` is the x-axis label, `y` is the y-axis label, and `color` and `shape` define the label for the legend. In addition, we can improve the color palette to be color-blind safe with the `scale_color_colorblind()` function from the ggthemes package.

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point(aes(color = species, shape = species)) +
  geom_smooth(method = "lm") +
  labs(
    title = "Body mass and flipper length",
    subtitle = "Dimensions for Adelie, Chinstrap, and Gentoo Penguins",
    x = "Flipper length (mm)", y = "Body mass (g)",
    color = "Species", shape = "Species"
  ) +
  scale_color_colorblind()
```



We finally have a plot that perfectly matches our "ultimate goal"!

Exercises

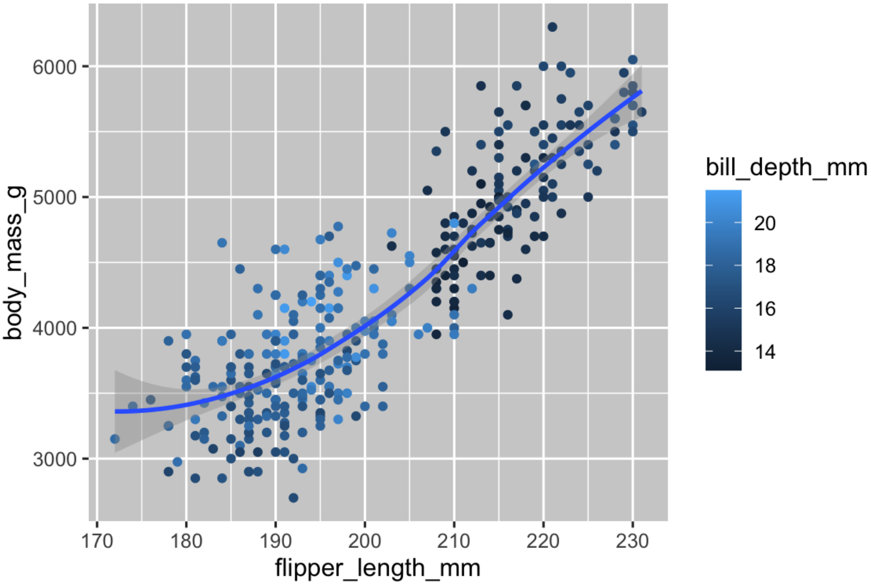
1. How many rows are in `penguins`? How many columns?
2. What does the `bill_depth_mm` variable in the `penguins` data frame describe? Read the help for `?penguins` to find out.
3. Make a scatterplot of `bill_depth_mm` versus `bill_length_mm`. That is, make a scatterplot with `bill_depth_mm` on the y-axis and `bill_length_mm` on the x-axis. Describe the relationship between these two variables.
4. What happens if you make a scatterplot of `species` versus `bill_depth_mm`? What might be a better choice of geom?
5. Why does the following give an error, and how would you fix it?

```
ggplot(data = penguins) +
  geom_point()
```

6. What does the `na.rm` argument do in `geom_point()`? What is the default value of the argument? Create a scatterplot where you successfully use this argument set to `TRUE`.

7. Add the following caption to the plot you made in the previous exercise: "Data come from the palmerpenguins package." Hint: Take a look at the documentation for `labs()`.

8. Re-create the following visualization. What aesthetic should `bill_depth_mm` be mapped to? And should it be mapped at the global level or at the geom level?



9. 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 = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g, color = island)
) +
  geom_point() +
  geom_smooth(se = FALSE)
```

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

```
ggplot(
  data = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point() +
  geom_smooth()

ggplot() +
  geom_point(
    data = penguins,
    mapping = aes(x = flipper_length_mm, y = body_mass_g)
  ) +
  geom_smooth(
    data = penguins,
    mapping = aes(x = flipper_length_mm, y = body_mass_g)
  )
```

ggplot2 Calls

As we move on from these introductory sections, 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 = penguins,
  mapping = aes(x = flipper_length_mm, y = body_mass_g)
) +
  geom_point()
```

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`; in the remainder of the book, we won't supply those names. That saves typing and, by reducing the amount of extra text, 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 25.

Rewriting the previous plot more concisely yields:

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

In the future, you'll also learn about the pipe, `|>`, which will allow you to create that plot with:

```
penguins |>
  ggplot(aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```

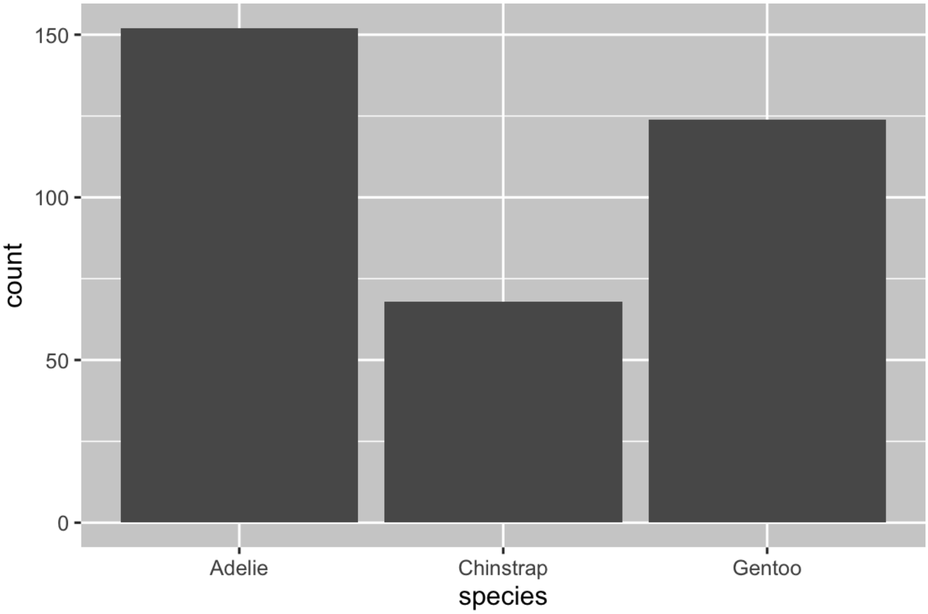
Visualizing Distributions

How you visualize the distribution of a variable depends on the type of variable: categorical or numerical.

A Categorical Variable

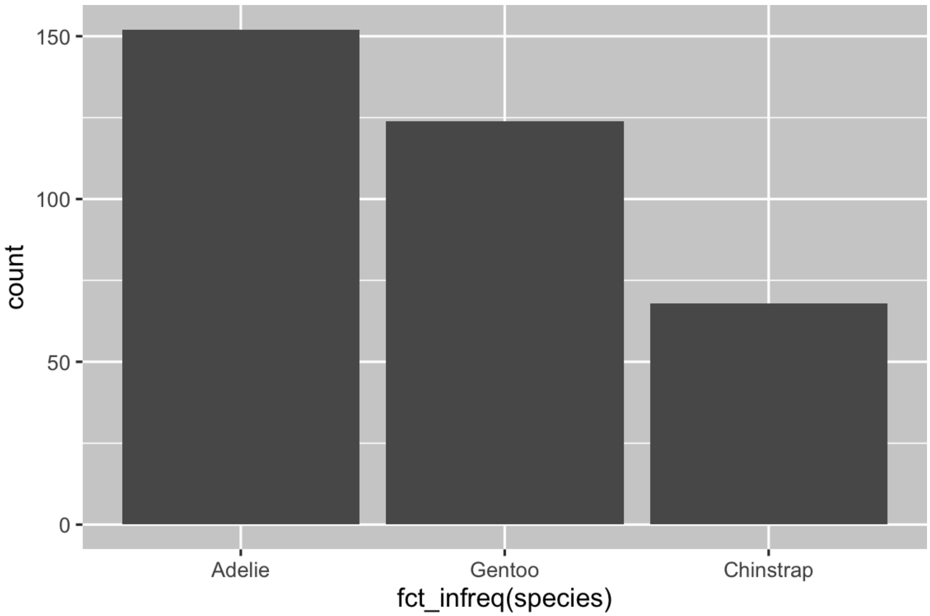
A variable is *categorical* if it can take only one of a small set of values. To examine the distribution of a categorical variable, you can use a bar chart. The height of the bars displays how many observations occurred with each x value.

```
ggplot(penguins, aes(x = species)) +
  geom_bar()
```



In bar plots of categorical variables with nonordered levels, like the previous penguin `species`, it's often preferable to reorder the bars based on their frequencies. Doing so requires transforming the variable to a factor (how R handles categorical data) and then reordering the levels of that factor.

```
ggplot(penguins, aes(x = fct_infreq(species))) +
  geom_bar()
```



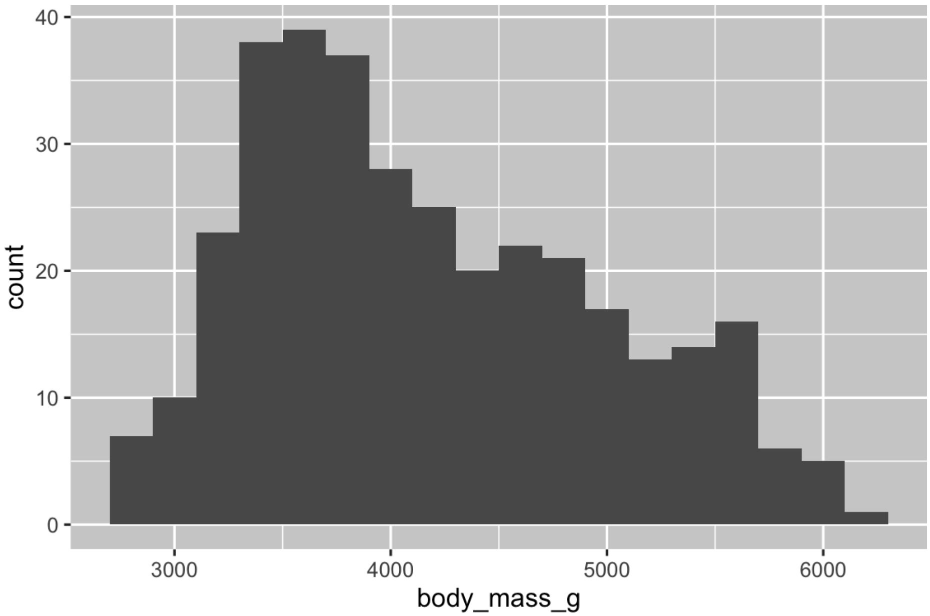
You will learn more about factors and functions for dealing with factors (such as `fct_infreq()`) in Chapter 16.

A Numerical Variable

A variable is *numerical* (or quantitative) if it can take on a wide range of numerical values and it is sensible to add, subtract, or take averages with those values. Numerical variables can be continuous or discrete.

One commonly used visualization for distributions of continuous variables is a histogram.

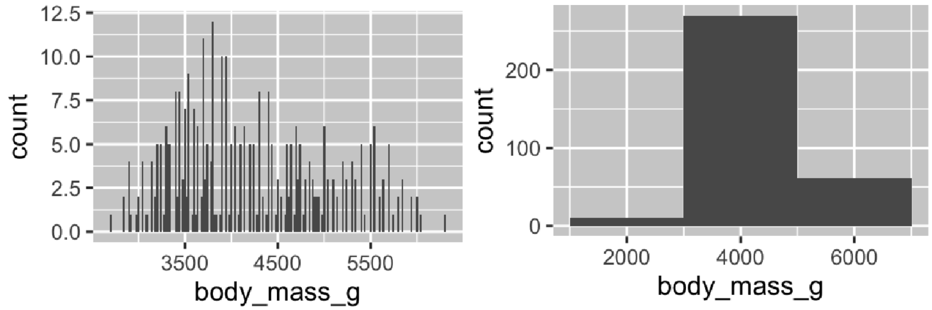
```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 200)
```



A histogram divides the x-axis into equally spaced bins and then uses the height of a bar to display the number of observations that fall in each bin. In the previous graph, the tallest bar shows that 39 observations have a `body_mass_g` value between 3,500 and 3,700 grams, which are the left and right edges of the bar.

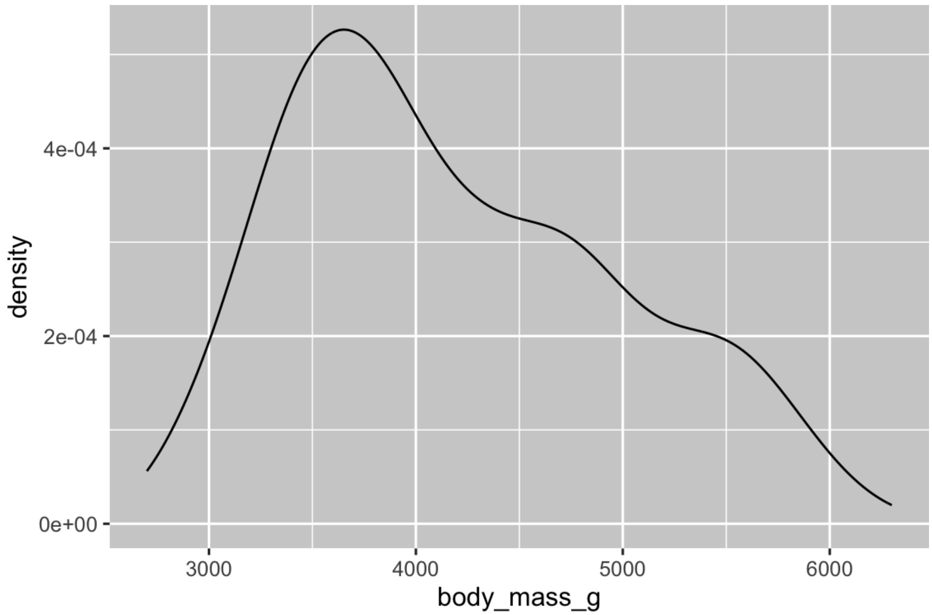
You can set the width of the intervals in a histogram with the `binwidth` argument, which is measured in the units of the `x` variable. You should always explore a variety of `binwidth` values when working with histograms, as different `binwidth` values can reveal different patterns. In the following plots, a `binwidth` of 20 is too narrow, resulting in too many bars, making it difficult to determine the shape of the distribution. Similarly, a `binwidth` of 2,000 is too high, resulting in all data being binned into only three bars and also making it difficult to determine the shape of the distribution. A `binwidth` of 200 provides a sensible balance.

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 20)
ggplot(penguins, aes(x = body_mass_g)) +
  geom_histogram(binwidth = 2000)
```



An alternative visualization for distributions of numerical variables is a density plot. A density plot is a smoothed-out version of a histogram and a practical alternative, particularly for continuous data that comes from an underlying smooth distribution. We won't go into how `geom_density()` estimates the density (you can read more about that in the function documentation), but let's explain how the density curve is drawn with an analogy. Imagine a histogram made out of wooden blocks. Then, imagine that you drop a cooked spaghetti string over it. The shape the spaghetti will take draped over blocks can be thought of as the shape of the density curve. It shows fewer details than a histogram but can make it easier to quickly glean the shape of the distribution, particularly with respect to modes and skewness.

```
ggplot(penguins, aes(x = body_mass_g)) +
  geom_density()
#> Warning: Removed 2 rows containing non-finite values (`stat_density()`).
```



Exercises

1. Make a bar plot of `species` of `penguins`, where you assign `species` to the `y` aesthetic. How is this plot different?
2. How are the following two plots different? Which aesthetic, `color` or `fill`, is more useful for changing the color of bars?

```
ggplot(penguins, aes(x = species)) +
  geom_bar(color = "red")

ggplot(penguins, aes(x = species)) +
  geom_bar(fill = "red")
```

3. What does the `bins` argument in `geom_histogram()` do?
4. Make a histogram of the `carat` variable in the `diamonds` dataset that is available when you load the tidyverse package. Experiment with different `binwidth` values. What value reveals the most interesting patterns?

Visualizing Relationships

To visualize a relationship we need to have at least two variables mapped to aesthetics of a plot. In the following sections you will learn about commonly used plots for visualizing relationships between two or more variables and the geoms used for creating them.

A Numerical and a Categorical Variable

To visualize the relationship between a numerical and a categorical variable we can use side-by-side box plots. A *boxplot* is a type of visual shorthand for measures of position (percentiles) that describe a distribution. It is also useful for identifying potential outliers. As shown in Figure 1-1, each boxplot consists of:

- A box that indicates the range of the middle half of the data, a distance known as the *interquartile range* (IQR), stretching from the 25th percentile of the distribution to the 75th percentile. 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 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.

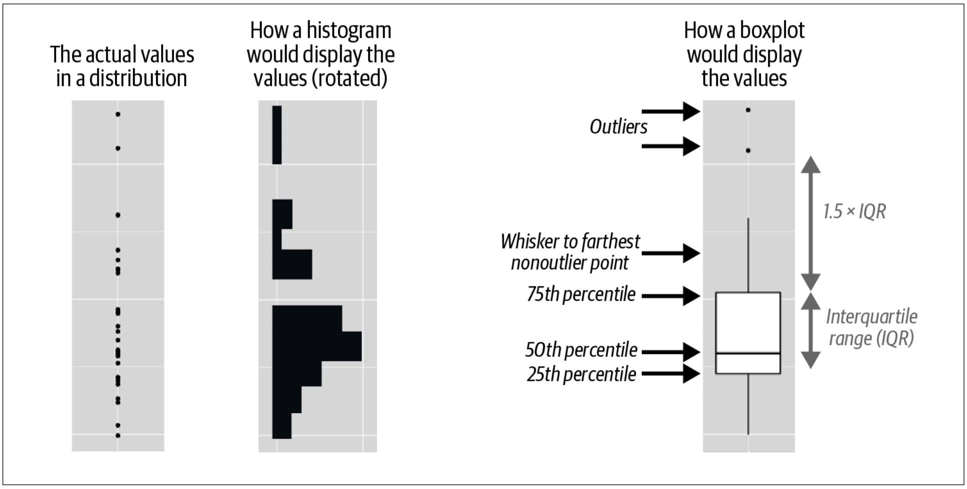
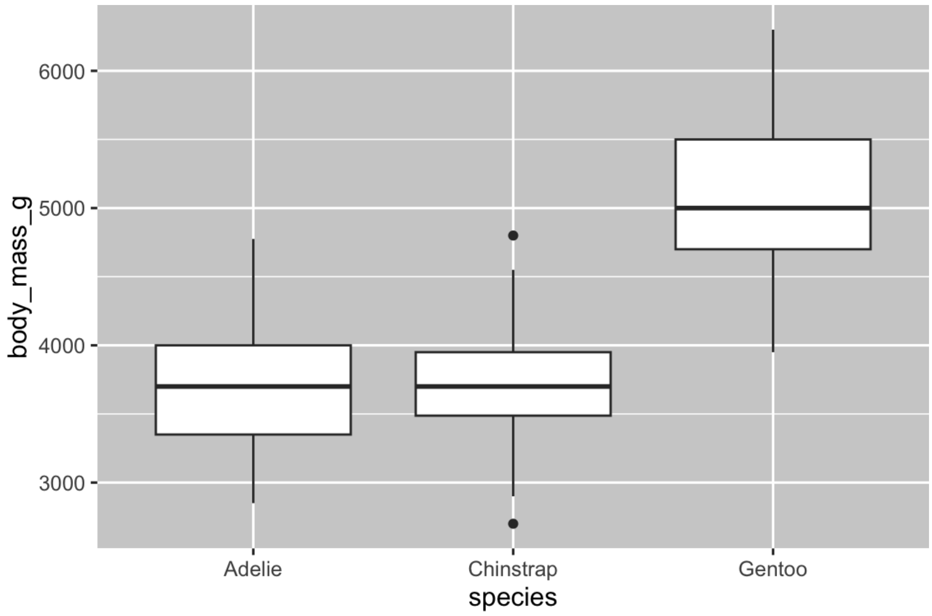

The actual values in a distribution
How a histogram would display the values (rotated)
How a boxplot would display the values
Outliers
1.5 × IQR
Whisker to farthest nonoutlier point
75th percentile
Interquartile range (IQR)
50th percentile
25th percentile

Figure 1-1. Diagram depicting how a boxplot is created.

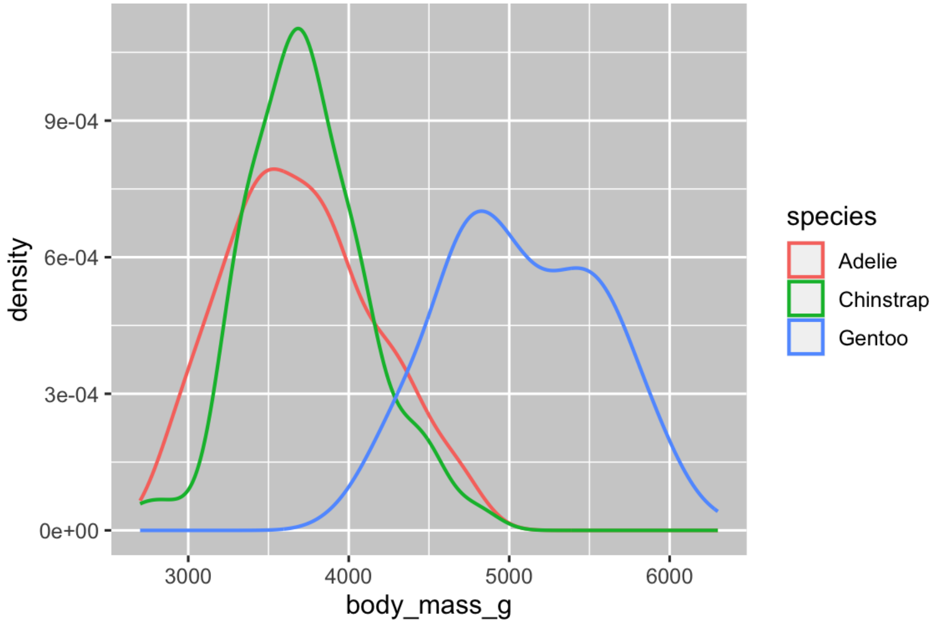
Let’s take a look at the distribution of body mass by species using `geom_boxplot()`:

```
ggplot(penguins, aes(x = species, y = body_mass_g)) +
  geom_boxplot()
```



Alternatively, we can make density plots with `geom_density()`:

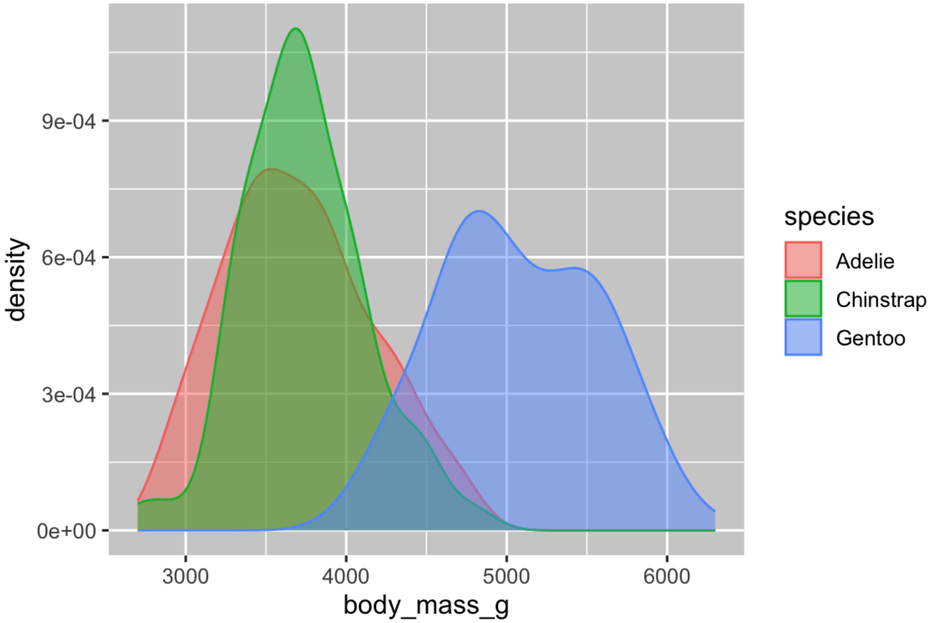
```
ggplot(penguins, aes(x = body_mass_g, color = species)) +
  geom_density(linewidth = 0.75)
```



We've also customized the thickness of the lines using the `linewidth` argument to make them stand out a bit more against the background.

Additionally, we can map `species` to both `color` and `fill` aesthetics and use the `alpha` aesthetic to add transparency to the filled density curves. This aesthetic takes values between 0 (completely transparent) and 1 (completely opaque). In the following plot it's set to 0.5:

```
ggplot(penguins, aes(x = body_mass_g, color = species, fill = species)) +
  geom_density(alpha = 0.5)
```



Note the terminology we have used here:

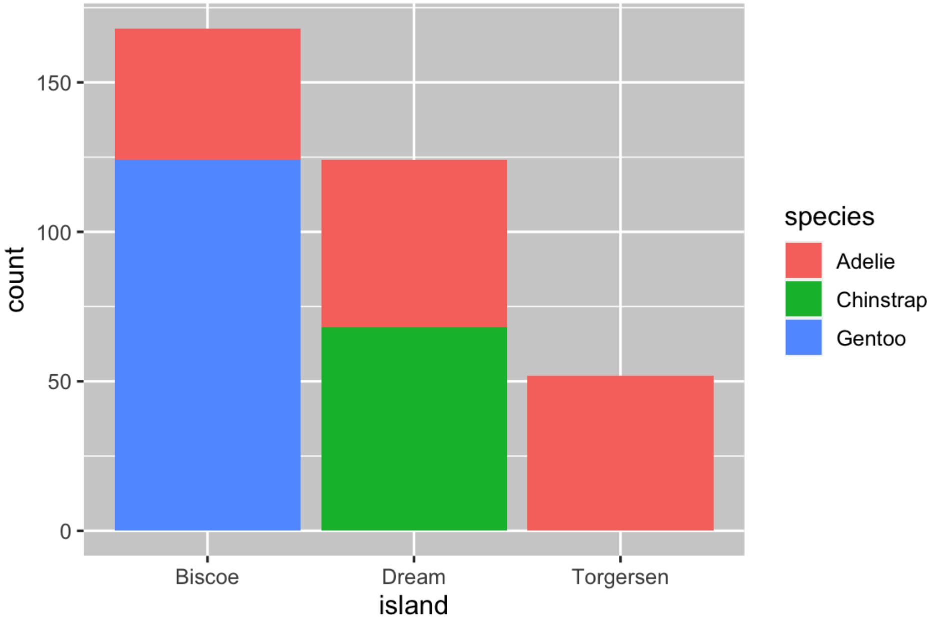
- We *map* variables to aesthetics if we want the visual attribute represented by that aesthetic to vary based on the values of that variable.
- Otherwise, we *set* the value of an aesthetic.

Two Categorical Variables

We can use stacked bar plots to visualize the relationship between two categorical variables. For example, the following two stacked bar plots both display the relationship between `island` and `species`, or, specifically, visualize the distribution of `species` within each island.

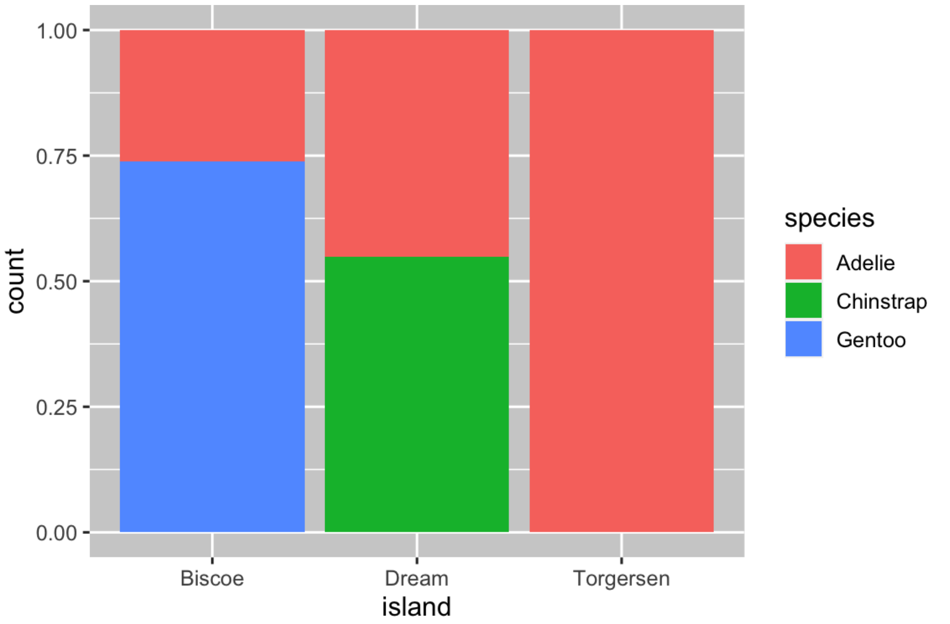
The first plot shows the frequencies of each species of penguins on each island. The plot of frequencies shows that there are equal numbers of Adelies on each island, but we don't have a good sense of the percentage balance within each island.

```
ggplot(penguins, aes(x = island, fill = species)) +
  geom_bar()
```



The second plot is a relative frequency plot, created by setting `position = "fill"` in the geom, and is more useful for comparing species distributions across islands since it's not affected by the unequal numbers of penguins across the islands. Using this plot we can see that Gentoo penguins all live on Biscoe island and make up roughly 75% of the penguins on that island, Chinstrap all live on Dream island and make up roughly 50% of the penguins on that island, and Adelie live on all three islands and make up all of the penguins on Torgersen.

```
ggplot(penguins, aes(x = island, fill = species)) +
  geom_bar(position = "fill")
```

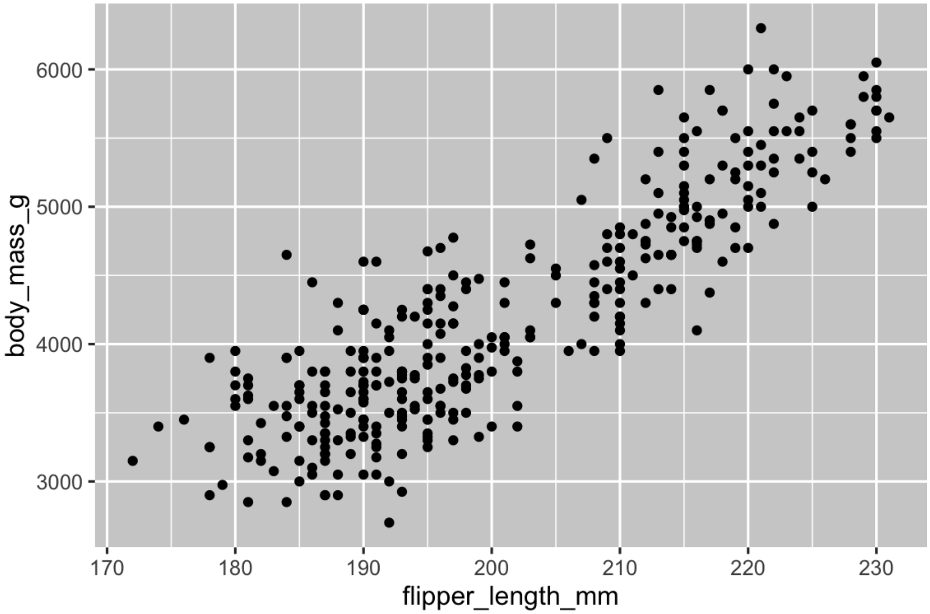


In creating these bar charts, we map the variable that will be separated into bars to the x aesthetic, and the variable that will change the colors inside the bars to the fill aesthetic.

Two Numerical Variables

So far you've learned about scatterplots (created with geom_point()) and smooth curves (created with geom_smooth()) for visualizing the relationship between two numerical variables. A scatterplot is probably the most commonly used plot for visualizing the relationship between two numerical variables.

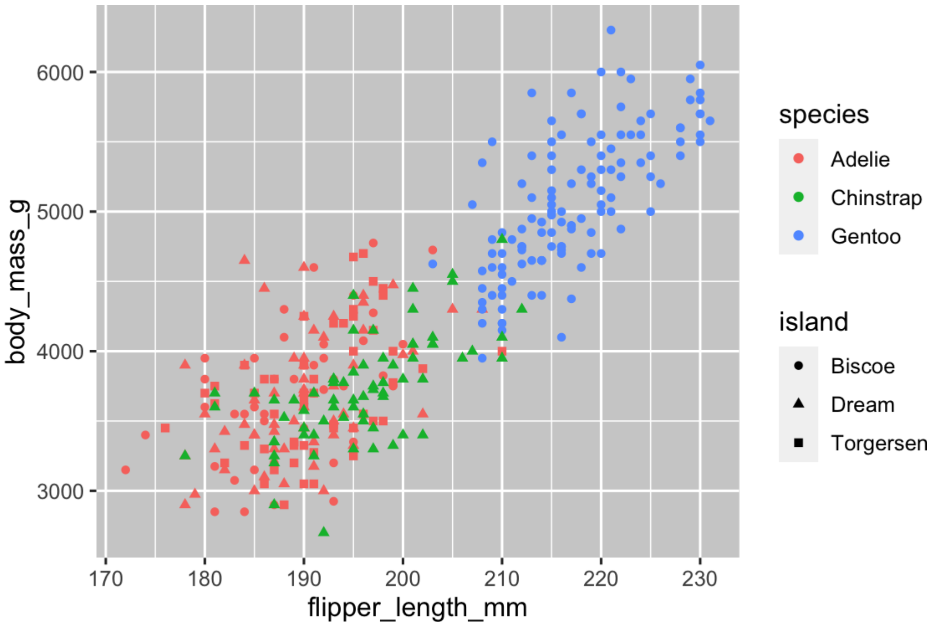
```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
```



Three or More Variables

As we saw in "Adding Aesthetics and Layers" on page 10, we can incorporate more variables into a plot by mapping them to additional aesthetics. For example, in the following scatterplot the colors of points represent species, and the shapes of points represent islands:

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(aes(color = species, shape = island))
```

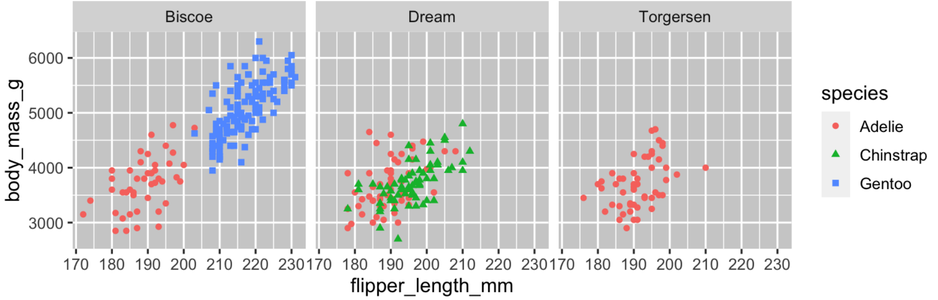


However, adding too many aesthetic mappings to a plot makes it cluttered and difficult to make sense of. Another option, which is particularly useful for categorical variables, is to split your plot into *facets*, subplots that each display one subset of the data.

To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` is a formula,[3] which you create with ~ followed by a variable name. The variable that you pass to `facet_wrap()` should be categorical.

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point(aes(color = species, shape = species)) +
  facet_wrap(~island)
```

[3] Here "formula" is the name of the thing created by ~, not a synonym for "equation."



You will learn about many other geoms for visualizing distributions of variables and relationships between them in Chapter 9.

Exercises

1. The `mpg` data frame that is bundled with the ggplot2 package contains 234 observations collected by the US Environmental Protection Agency on 38 car models. Which variables in `mpg` are categorical? Which variables are numerical? (Hint: Type `?mpg` to read the documentation for the dataset.) How can you see this information when you run `mpg`?

2. Make a scatterplot of `hwy` versus `displ` using the `mpg` data frame. Next, map a third, numerical variable to `color`, then `size`, then both `color` and `size`, and then `shape`. How do these aesthetics behave differently for categorical versus numerical variables?

3. In the scatterplot of `hwy` versus `displ`, what happens if you map a third variable to `linewidth`?

4. What happens if you map the same variable to multiple aesthetics?

5. Make a scatterplot of `bill_depth_mm` versus `bill_length_mm` and color the points by `species`. What does adding coloring by species reveal about the relationship between these two variables? What about faceting by species?

6. Why does the following yield two separate legends? How would you fix it to combine the two legends?

```
ggplot(
  data = penguins,
  mapping = aes(
    x = bill_length_mm, y = bill_depth_mm,
    color = species, shape = species
  )
) +
  geom_point() +
  labs(color = "Species")
```

7. Create the two following stacked bar plots. Which question can you answer with the first one? Which question can you answer with the second one?

   ```
   ggplot(penguins, aes(x = island, fill = species)) +
     geom_bar(position = "fill")
   ggplot(penguins, aes(x = species, fill = island)) +
     geom_bar(position = "fill")
   ```

Saving Your Plots

Once you've made a plot, you might want to get it out of R by saving it as an image that you can use elsewhere. That's the job of `ggsave()`, which will save the plot most recently created to disk:

```
ggplot(penguins, aes(x = flipper_length_mm, y = body_mass_g)) +
  geom_point()
ggsave(filename = "penguin-plot.png")
```

This will save your plot to your working directory, a concept you'll learn more about in Chapter 6.

If you don't specify the `width` and `height`, they will be taken from the dimensions of the current plotting device. For reproducible code, you'll want to specify them. You can learn more about `ggsave()` in the documentation.

Generally, however, we recommend that you assemble your final reports using Quarto, a reproducible authoring system that allows you to interleave your code and your prose and automatically include your plots in your write-ups. You will learn more about Quarto in Chapter 28.

Exercises

1. Run the following lines of code. Which of the two plots is saved as `mpg-plot.png`? Why?

   ```
   ggplot(mpg, aes(x = class)) +
     geom_bar()
   ggplot(mpg, aes(x = cty, y = hwy)) +
     geom_point()
   ggsave("mpg-plot.png")
   ```

2. What do you need to change in the previous code to save the plot as a PDF instead of a PNG? How could you find out what types of image files would work in `ggsave()`?

Common Problems

As you start to run R code, you're likely to run into problems. Don't worry—it happens to everyone. We have all been writing R code for years, but every day we still write code that doesn't work on the first try!

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 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 Escape 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 highlighting 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, even if the answer is in the error message, you might not yet know how to understand it. Another great tool is Google: try googling the error message, as it's likely someone else has had the same problem and has gotten help online.

Summary

In this chapter, you've learned the basics of data visualization with ggplot2. We started with the basic idea that underpins ggplot2: a visualization is a mapping from variables in your data to aesthetic properties such as position, color, size, and shape. You then learned about increasing the complexity and improving the presentation of your plots layer by layer. You also learned about commonly used plots for visualizing the distribution of a single variable, as well as for visualizing relationships between two or more variables, by levering additional aesthetic mappings and/or splitting your plot into small multiples using faceting.

We'll use visualizations again and again throughout this book, introducing new techniques as we need them, as well as do a deeper dive into creating visualizations with ggplot2 in Chapter 9 through Chapter 11.

Now that you understand the basics of visualization, in the next chapter we're going to switch gears a little and give you some practical workflow advice. We intersperse workflow advice with data science tools throughout this part of the book because it'll help you stay organized as you write increasing amounts of R code.

CHAPTER 2
Workflow: Basics

You now have some experience running R code. We 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 can cause it to complain. But while you should expect to be a little frustrated, take comfort in that this experience is 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 ensure you've got a solid foundation in running R code and that you know some of the most helpful RStudio features.

Coding Basics

Let's review some basics we've omitted so far in the interest of getting you plotting as quickly as possible. You can use R to do basic math calculations:

```
1 / 200 * 30
#> [1] 0.15
(59 + 73 + 2) / 3
#> [1] 44.66667
sin(pi / 2)
#> [1] 1
```

You can create new objects with the assignment operator `<-`:

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

Note that the value of `x` is not printed, it's just stored. If you want to view the value, type `x` in the console.

You can *c*ombine multiple elements into a vector with `c()`:

```
primes <- c(2, 3, 5, 7, 11, 13)
```

And basic arithmetic on vectors is applied to every element of the vector:

```
primes * 2
#> [1]  4  6 10 14 22 26
primes - 1
#> [1]  1  2  4  6 10 12
```

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. You can save time with RStudio's keyboard shortcut: Alt+– (the minus sign). Notice that RStudio automatically surrounds `<-` with spaces, which is a good code formatting practice. Code can be miserable to read on a good day, so giveyoureyesabreak and use spaces.

Comments

R will ignore any text after # for that line. This allows you to write *comments*, text that is ignored by R but read by humans. We'll sometimes include comments in examples to explain what's happening with the code.

Comments can be helpful for briefly describing what the code does:

```
# create vector of primes
primes <- c(2, 3, 5, 7, 11, 13)

# multiply primes by 2
primes * 2
#> [1]  4  6 10 14 22 26
```

With short pieces of code like this, leaving a comment for every single line of code might not be necessary. But as the code you're writing gets more complex, comments can save you (and your collaborators) a lot of time figuring out what was done in the code.

Use comments to explain the *why* of your code, not the *how* or the *what*. The *what* and *how* of your code are always possible to figure out, even if it might be tedious, by carefully reading it. If you describe every step in the comments and then change the code, you will have to remember to update the comments as well or it will be confusing when you return to your code in the future.

Figuring out *why* something was done is much more difficult, if not impossible. For example, `geom_smooth()` has an argument called `span`, which controls the smoothness of the curve, with larger values yielding a smoother curve. Suppose you decide to change the value of `span` from its default of 0.75 to 0.9: it's easy for a future reader to understand *what* is happening, but unless you note your thinking in a comment, no one will understand *why* you changed the default.

For data analysis code, use comments to explain your overall plan of attack and record important insights as you encounter them. There's no way to re-capture this knowledge from the code itself.

What's in a Name?

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

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

We'll return to names again when we discuss code style in Chapter 4.

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 RStudio's completion facility: type *this*, press Tab, add characters until you have a unique prefix, and then press Return.

Let's assume you made a mistake and that the value of `this_is_a_really_long_name` should be 3.5, not 2.5. You can use another keyboard shortcut to help you fix it. For example, you can press ↑ to bring the last command you typed and edit it. Or, type *this* and then press Cmd/Ctrl+↑ to list all the commands you've typed that start with those letters. Use the arrow keys to navigate and 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
```

This illustrates the implied contract between you and R: R will do the tedious computations for you, but in exchange, you must be completely precise in your instructions.

If not, you're likely to get an error that says the object you're looking for was not found. Typos matter; R can't read your mind and say, "Oh, they probably meant `r_rocks` when they typed `r_rock`." Case matters; similarly, R can't read your mind and say, "Oh, they probably meant `r_rocks` when they typed `R_rocks`."

Calling Functions

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

```
function_name(argument1 = value1, argument2 = value2, ...)
```

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 on the help tab in the lower-right pane.

When you've selected the function you want, press Tab again. RStudio will add matching opening (`(`) and closing (`)`) parentheses for you. Type the name of the first argument, `from`, and set it equal to `1`. Then, type the name of the second argument, `to`, and set it equal to `10`. Finally, hit Return.

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

We often omit the names of the first several arguments in function calls, so we can rewrite this as follows:

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

Type the following code and notice that RStudio provides similar assistance with the paired quotation marks:

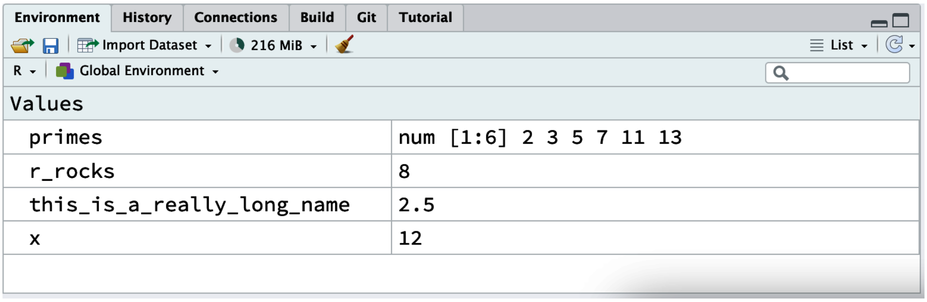
```
x <- "hello world"
```

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, this means you've forgotten either a `"` or a `)`. Either add the missing pair, or press Esc to abort the expression and try again.

Note that the Environment tab in the upper-right pane displays all of the objects that you've created:



Values	
primes	num [1:6] 2 3 5 7 11 13
r_rocks	8
this_is_a_really_long_name	2.5
x	12

Exercises

1. Why does this code not work?

   ```
   my_variable <- 10
   my_varıable
   #> Error in eval(expr, envir, enclos): object 'my_varıable' 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:

   ```
   libary(todyverse)

   ggplot(dTA = mpg) +
     geom_point(maping = aes(x = displ y = hwy)) +
     geom_smooth(method = "lm)
   ```

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

4. Let's revisit an exercise from "Saving Your Plots" on page 30. Run the following lines of code. Which of the two plots is saved as `mpg-plot.png`? Why?

   ```
   my_bar_plot <- ggplot(mpg, aes(x = class)) +
     geom_bar()
   my_scatter_plot <- ggplot(mpg, aes(x = cty, y = hwy)) +
     geom_point()
   ggsave(filename = "mpg-plot.png", plot = my_bar_plot)
   ```

Summary

Now that you've learned a little more about how R code works and gotten some tips to help you understand your code when you come back to it in the future, in the next chapter, we'll continue your data science journey by teaching you about dplyr, the tidyverse package that helps you transform data, whether it's selecting important variables, filtering down to rows of interest, or computing summary statistics.

CHAPTER 3

Data Transformation

Introduction

Visualization is an important tool for generating insight, but it's rare that you get the data in exactly the right form you need to make the graph you want. Often you'll need to create some new variables or summaries to answer your questions with your data, or maybe you just want to rename the variables or reorder the observations 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 introduce you to data transformation using the dplyr package and a new dataset on flights that departed New York City in 2013.

The goal of this chapter is to give you an overview of all the key tools for transforming a data frame. We'll start with functions that operate on rows and then columns of a data frame, and then we'll circle back to talk more about the pipe, an important tool that you use to combine verbs. We will then introduce the ability to work with groups. We will end the chapter with a case study that showcases these functions in action, and we'll come back to the functions in more detail in later chapters, as we start to dig into specific types of data (e.g., numbers, strings, dates).

Prerequisites

In this chapter we'll focus on 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)
#> ── Attaching core tidyverse packages ──────────────────── tidyverse 2.0.0 ──
#> ✔ dplyr     1.1.0.9000     ✔ readr     2.1.4
#> ✔ forcats   1.0.0          ✔ stringr   1.5.0
#> ✔ ggplot2   3.4.1          ✔ tibble    3.1.8
#> ✔ lubridate 1.9.2          ✔ tidyr     1.3.0
#> ✔ purrr     1.0.1
#> ── Conflicts ──────────────────────────────────────── tidyverse_conflicts() ──
#> ✖ dplyr::filter() masks stats::filter()
#> ✖ dplyr::lag()    masks stats::lag()
#> ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all
#>   conflicts to become errors
```

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()`. So far we've mostly ignored which package a function comes from because most of the time it doesn't matter. However, knowing the package can facilitate finding help as well as related functions, so when we need to be precise about which function a package comes from, we'll use the same syntax as R: `packagename::functionname()`.

nycflights13

To explore the basic dplyr verbs, we're going to use `nycflights13::flights`. This dataset 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 arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

`flights` is a tibble, a special type of data frame used by the tidyverse to avoid some common gotchas. The most important difference between tibbles and data frames is the way tibbles print; they are designed for large datasets, so they show only the first few rows and only the columns that fit on one screen. There are a few options to see everything. If you're using RStudio, the most convenient is probably `View(flights)`, which will open an interactive scrollable and filterable view. Otherwise, you can use `print(flights, width = Inf)` to show all columns or use `glimpse()`:

```
glimpse(flights)
#> Rows: 336,776
#> Columns: 19
#> $ year           <int> 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013, 2013…
#> $ month          <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
#> $ day            <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1…
#> $ dep_time       <int> 517, 533, 542, 544, 554, 554, 555, 557, 557, 558, 55…
#> $ sched_dep_time <int> 515, 529, 540, 545, 600, 558, 600, 600, 600, 600, 60…
#> $ dep_delay      <dbl> 2, 4, 2, -1, -6, -4, -5, -3, -3, -2, -2, -2, -2, -2,…
#> $ arr_time       <int> 830, 850, 923, 1004, 812, 740, 913, 709, 838, 753, 8…
#> $ sched_arr_time <int> 819, 830, 850, 1022, 837, 728, 854, 723, 846, 745, 8…
#> $ arr_delay      <dbl> 11, 20, 33, -18, -25, 12, 19, -14, -8, 8, -2, -3, 7,…
#> $ carrier        <chr> "UA", "UA", "AA", "B6", "DL", "UA", "B6", "EV", "B6"…
#> $ flight         <int> 1545, 1714, 1141, 725, 461, 1696, 507, 5708, 79, 301…
#> $ tailnum        <chr> "N14228", "N24211", "N619AA", "N804JB", "N668DN", "N…
#> $ origin         <chr> "EWR", "LGA", "JFK", "JFK", "LGA", "EWR", "EWR", "LG…
#> $ dest           <chr> "IAH", "IAH", "MIA", "BQN", "ATL", "ORD", "FLL", "IA…
#> $ air_time       <dbl> 227, 227, 160, 183, 116, 150, 158, 53, 140, 138, 149…
#> $ distance       <dbl> 1400, 1416, 1089, 1576, 762, 719, 1065, 229, 944, 73…
#> $ hour           <dbl> 5, 5, 5, 5, 6, 5, 6, 6, 6, 6, 6, 6, 6, 6, 6, 5, 6, 6…
#> $ minute         <dbl> 15, 29, 40, 45, 0, 58, 0, 0, 0, 0, 0, 0, 0, 0, 0, 59…
#> $ time_hour      <dttm> 2013-01-01 05:00:00, 2013-01-01 05:00:00, 2013-01-0…
```

In both views, the variables names are followed by abbreviations that tell you the type of each variable: `<int>` is short for integer, `<dbl>` is short for double (aka real numbers), `<chr>` for character (aka strings), and `<dttm>` for date-time. These are important because the operations you can perform on a column depend so much on its "type."

dplyr Basics

You're about to learn the primary dplyr verbs (functions), which will allow you to solve the vast majority of your data manipulation challenges. But before we discuss their individual differences, it's worth stating what they have in common:

- The first argument is always a data frame.
- The subsequent arguments typically describe which columns to operate on, using the variable names (without quotes).
- The output is always a new data frame.

Because each verb does one thing well, solving complex problems will usually require combining multiple verbs, and we'll do so with the pipe, `|>`. We'll discuss the pipe more in "The Pipe" on page 51, but in brief, the pipe takes the thing on its left and passes it along to the function on its right so that `x |> f(y)` is equivalent to `f(x, y)`, and `x |> f(y) |> g(z)` is equivalent to `g(f(x, y), z)`. The easiest way to pronounce the pipe is "then." That makes it possible to get a sense of the following code even though you haven't yet learned the details:

```
flights |>
  filter(dest == "IAH") |>
  group_by(year, month, day) |>
  summarize(
    arr_delay = mean(arr_delay, na.rm = TRUE)
  )
```

dplyr's verbs are organized into four groups based on what they operate on: *rows*, *columns*, *groups*, and *tables*. In the following sections, you'll learn the most important verbs for rows, columns, and groups; then we'll come back to the join verbs that work on tables in Chapter 19. Let's dive in!

Rows

The most important verbs that operate on rows of a dataset are `filter()`, which changes which rows are present without changing their order, and `arrange()`, which changes the order of the rows without changing which are present. Both functions affect only the rows, and the columns are left unchanged. We'll also discuss `distinct()`, which finds rows with unique values, but unlike `arrange()` and `filter()`, it can also optionally modify the columns.

filter()

`filter()` allows you to keep rows based on the values of the columns.[1] The first argument is the data frame. The second and subsequent arguments are the conditions that must be true to keep the row. For example, we could find all flights that departed more than 120 minutes (two hours) late:

```
flights |>
  filter(dep_delay > 120)
#> # A tibble: 9,723 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      848           1835       853     1001           1950
#> 2  2013     1     1      957            733       144     1056            853
#> 3  2013     1     1     1114            900       134     1447           1222
#> 4  2013     1     1     1540           1338       122     2020           1825
#> 5  2013     1     1     1815           1325       290     2120           1542
#> 6  2013     1     1     1842           1422       260     1958           1535
#> # … with 9,717 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

As well as `>` (greater than), you can use `>=` (greater than or equal to), `<` (less than), `<=` (less than or equal to), `==` (equal to), and `!=` (not equal to). You can also combine conditions with `&` or `,` to indicate "and" (check for both conditions) or with `|` to indicate "or" (check for either condition):

[1] Later, you'll learn about the `slice_*()` family, which allows you to choose rows based on their positions.

```
# Flights that departed on January 1
flights |>
  filter(month == 1 & day == 1)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 836 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …

# Flights that departed in January or February
flights |>
  filter(month == 1 | month == 2)
#> # A tibble: 51,955 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 51,949 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

There's a useful shortcut when you're combining | and ==: %in%. It keeps rows where the variable equals one of the values on the right:

```
# A shorter way to select flights that departed in January or February
flights |>
  filter(month %in% c(1, 2))
#> # A tibble: 51,955 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 51,949 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

We'll come back to these comparisons and logical operators in more detail in Chapter 12.

When you run filter(), dplyr executes the filtering operation, creating a new data frame, and then prints it. It doesn't modify the existing flights dataset because dplyr functions never modify their inputs. To save the result, you need to use the assignment operator, <-:

```
jan1 <- flights |>
  filter(month == 1 & day == 1)
```

Common Mistakes

When you're starting out with R, the easiest mistake to make is to use = instead of == when testing for equality. `filter()` will let you know when this happens:

```
flights |>
  filter(month = 1)
#> Error in `filter()`:
#> ! We detected a named input.
#> ℹ This usually means that you've used `=` instead of `==`.
#> ℹ Did you mean `month == 1`?
```

Another mistake is writing "or" statements like you would in English:

```
flights |>
  filter(month == 1 | 2)
```

This "works" in the sense that it doesn't throw an error, but it doesn't do what you want because `|` first checks the condition `month == 1` and then checks the condition `2`, which is not a sensible condition to check. We'll learn more about what's happening here and why in "Boolean Operations" on page 279.

arrange()

`arrange()` changes the order of the rows based on the value of the columns. 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. For example, the following code sorts by the departure time, which is spread over four columns. We get the earliest years first, then within a year the earliest months, etc.

```
flights |>
  arrange(year, month, day, dep_time)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

You can use `desc()` on a column inside of `arrange()` to reorder the data frame based on that column in descending (big-to-small) order. For example, this code orders flights from most to least delayed:

```
flights |>
  arrange(desc(dep_delay))
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     9      641            900      1301     1242           1530
#> 2  2013     6    15     1432           1935      1137     1607           2120
#> 3  2013     1    10     1121           1635      1126     1239           1810
#> 4  2013     9    20     1139           1845      1014     1457           2210
#> 5  2013     7    22      845           1600      1005     1044           1815
#> 6  2013     4    10     1100           1900       960     1342           2211
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

Note that the number of rows has not changed. We're only arranging the data; we're not filtering it.

distinct()

`distinct()` finds all the unique rows in a dataset, so in a technical sense, it primarily operates on the rows. Most of the time, however, you'll want the distinct combination of some variables, so you can also optionally supply column names:

```
# Remove duplicate rows, if any
flights |>
  distinct()
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …

# Find all unique origin and destination pairs
flights |>
  distinct(origin, dest)
#> # A tibble: 224 × 2
#>   origin dest
#>   <chr>  <chr>
#> 1 EWR    IAH
#> 2 LGA    IAH
#> 3 JFK    MIA
#> 4 JFK    BQN
#> 5 LGA    ATL
#> 6 EWR    ORD
#> # … with 218 more rows
```

Alternatively, if you want to keep the other columns when filtering for unique rows, you can use the `.keep_all = TRUE` option:

```
flights |>
  distinct(origin, dest, .keep_all = TRUE)
#> # A tibble: 224 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 218 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

It's not a coincidence that all of these distinct flights are on January 1: `distinct()` will find the first occurrence of a unique row in the dataset and discard the rest.

If you want to find the number of occurrences instead, you're better off swapping `distinct()` for `count()`, and with the `sort = TRUE` argument you can arrange them in descending order of number of occurrences. You'll learn more about count in "Counts" on page 222.

```
flights |>
  count(origin, dest, sort = TRUE)
#> # A tibble: 224 × 3
#>   origin dest      n
#>   <chr>  <chr> <int>
#> 1 JFK    LAX   11262
#> 2 LGA    ATL   10263
#> 3 LGA    ORD    8857
#> 4 JFK    SFO    8204
#> 5 LGA    CLT    6168
#> 6 EWR    ORD    6100
#> # … with 218 more rows
```

Exercises

1. In a single pipeline for each condition, find all flights that meet the condition:
   - Had an arrival delay of two or more hours
   - Flew to Houston (`IAH` or `HOU`)
   - Were operated by United, American, or Delta
   - Departed in summer (July, August, and September)
   - Arrived more than two hours late, but didn't leave late
   - Were delayed by at least an hour, but made up more than 30 minutes in flight
2. Sort `flights` to find the flights with the longest departure delays. Find the flights that left earliest in the morning.

3. Sort `flights` to find the fastest flights. (Hint: Try including a math calculation inside of your function.)
4. Was there a flight on every day of 2013?
5. Which flights traveled the farthest distance? Which traveled the least distance?
6. Does it matter what order you used `filter()` and `arrange()` if you're using both? Why/why not? Think about the results and how much work the functions would have to do.

Columns

There are four important verbs that affect the columns without changing the rows: `mutate()` creates new columns that are derived from the existing columns, `select()` changes which columns are present, `rename()` changes the names of the columns, and `relocate()` changes the positions of the columns.

mutate()

The job of `mutate()` is to add new columns that are calculated from the existing columns. In the transform chapters, you'll learn a large set of functions that you can use to manipulate different types of variables. For now, we'll stick with basic algebra, which allows us to compute the `gain`, how much time a delayed flight made up in the air, and the `speed` in miles per hour:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60
  )
#> # A tibble: 336,776 × 21
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 13 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

By default, `mutate()` adds new columns on the right side of your dataset, which makes it difficult to see what's happening here. We can use the `.before` argument to instead add the variables to the left side:[2]

[2] Remember that in RStudio, the easiest way to see a dataset with many columns is `View()`.

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .before = 1
  )
#> # A tibble: 336,776 × 21
#>    gain speed  year month   day dep_time sched_dep_time dep_delay arr_time
#>   <dbl> <dbl> <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1    -9  370.  2013     1     1      517            515         2      830
#> 2   -16  374.  2013     1     1      533            529         4      850
#> 3   -31  408.  2013     1     1      542            540         2      923
#> 4    17  517.  2013     1     1      544            545        -1     1004
#> 5    19  394.  2013     1     1      554            600        -6      812
#> 6   -16  288.  2013     1     1      554            558        -4      740
#> # … with 336,770 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>, …
```

The . is a sign that .before is an argument to the function, not the name of a third new variable we are creating. You can also use .after to add after a variable, and in both .before and .after you can use the variable name instead of a position. For example, we could add the new variables after day:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    speed = distance / air_time * 60,
    .after = day
  )
```

Alternatively, you can control which variables are kept with the .keep argument. A particularly useful argument is "used", which specifies that we keep only the columns that were involved or created in the mutate() step. For example, the following output will contain only the variables dep_delay, arr_delay, air_time, gain, hours, and gain_per_hour:

```
flights |>
  mutate(
    gain = dep_delay - arr_delay,
    hours = air_time / 60,
    gain_per_hour = gain / hours,
    .keep = "used"
  )
```

Note that since we haven't assigned the result of the previous computation back to flights, the new variables gain, hours, and gain_per_hour will be printed only and will not be stored in a data frame. And if we want them to be available in a data frame for future use, we should think carefully about whether we want the result to be assigned back to flights, overwriting the original data frame with many more variables, or to a new object. Often, the right answer is a new object that is named informatively to indicate its contents, e.g., delay_gain, but you might also have good reasons for overwriting flights.

select()

It's not uncommon to get datasets with hundreds or even thousands of variables. In this situation, the first challenge is often just focusing on the variables you're interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables:

- Select columns by name:

  ```
  flights |>
    select(year, month, day)
  ```

- Select all columns between year and day (inclusive):

  ```
  flights |>
    select(year:day)
  ```

- Select all columns except those from year to day (inclusive):

  ```
  flights |>
    select(!year:day)
  ```

  You can also use `-` instead of `!` (and you're likely to see that in the wild); we recommend `!` because it reads as "not" and combines well with `&` and `|`.

- Select all columns that are characters:

  ```
  flights |>
    select(where(is.character))
  ```

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"

`num_range("x", 1:3)`
: Matches `x1`, `x2`, and `x3`

See `?select` for more details. Once you know regular expressions (the topic of Chapter 15), you'll also be able to use `matches()` to select variables that match a pattern.

You can rename variables as you `select()` them by using `=`. The new name appears on the left side of the `=`, and the old variable appears on the right side:

```
flights |>
  select(tail_num = tailnum)
#> # A tibble: 336,776 × 1
#>   tail_num
#>   <chr>
```

```
#> 1 N14228
#> 2 N24211
#> 3 N619AA
#> 4 N804JB
#> 5 N668DN
#> 6 N39463
#> # … with 336,770 more rows
```

rename()

If you want to keep all the existing variables and just want to rename a few, you can use `rename()` instead of `select()`:

```
flights |>
  rename(tail_num = tailnum)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tail_num <chr>, origin <chr>, dest <chr>, …
```

If you have a bunch of inconsistently named columns and it would be painful to fix them all by hand, check out `janitor::clean_names()`, which provides some useful automated cleaning.

relocate()

Use `relocate()` to move variables around. You might want to collect related variables together or move important variables to the front. By default `relocate()` moves variables to the front:

```
flights |>
  relocate(time_hour, air_time)
#> # A tibble: 336,776 × 19
#>   time_hour           air_time  year month   day dep_time sched_dep_time
#>   <dttm>                 <dbl> <int> <int> <int>    <int>          <int>
#> 1 2013-01-01 05:00:00      227  2013     1     1      517            515
#> 2 2013-01-01 05:00:00      227  2013     1     1      533            529
#> 3 2013-01-01 05:00:00      160  2013     1     1      542            540
#> 4 2013-01-01 05:00:00      183  2013     1     1      544            545
#> 5 2013-01-01 06:00:00      116  2013     1     1      554            600
#> 6 2013-01-01 05:00:00      150  2013     1     1      554            558
#> # … with 336,770 more rows, and 12 more variables: dep_delay <dbl>,
#> #   arr_time <int>, sched_arr_time <int>, arr_delay <dbl>, carrier <chr>, …
```

You can also specify where to put them using the `.before` and `.after` arguments, just like in `mutate()`:

```
flights |>
  relocate(year:dep_time, .after = time_hour)
flights |>
  relocate(starts_with("arr"), .before = dep_time)
```

Exercises

1. Compare `dep_time`, `sched_dep_time`, and `dep_delay`. How would you expect those three numbers to be related?

2. Brainstorm as many ways as possible to select `dep_time`, `dep_delay`, `arr_time`, and `arr_delay` from `flights`.

3. What happens if you specify the name of the same variable multiple times in a `select()` call?

4. What does the `any_of()` function do? Why might it be helpful in conjunction with this vector?

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

5. Does the result of running the following code surprise you? How do the select helpers deal with upper- and lowercase by default? How can you change that default?

   ```
   flights |> select(contains("TIME"))
   ```

6. Rename `air_time` to `air_time_min` to indicate units of measurement and move it to the beginning of the data frame.

7. Why doesn't the following work, and what does the error mean?

   ```
   flights |>
     select(tailnum) |>
     arrange(arr_delay)
   #> Error in `arrange()`:
   #> ℹ In argument: `..1 = arr_delay`.
   #> Caused by error:
   #> ! object 'arr_delay' not found
   ```

The Pipe

We've shown you simple examples of the pipe, but its real power arises when you start to combine multiple verbs.

For example, imagine that you wanted to find the fast flights to Houston's IAH airport: you need to combine `filter()`, `mutate()`, `select()`, and `arrange()`:

```
flights |>
  filter(dest == "IAH") |>
  mutate(speed = distance / air_time * 60) |>
  select(year:day, dep_time, carrier, flight, speed) |>
  arrange(desc(speed))
#> # A tibble: 7,198 × 7
#>    year month   day dep_time carrier flight speed
#>   <int> <int> <int>    <int> <chr>    <int> <dbl>
#> 1  2013     7     9      707 UA         226  522.
#> 2  2013     8    27     1850 UA        1128  521.
#> 3  2013     8    28      902 UA        1711  519.
#> 4  2013     8    28     2122 UA        1022  519.
#> 5  2013     6    11     1628 UA        1178  515.
#> 6  2013     8    27     1017 UA         333  515.
#> # … with 7,192 more rows
```

Even though this pipeline has four steps, it's easy to skim because the verbs come at the start of each line: start with the `flights` data, then filter, then mutate, then select, and then arrange.

What would happen if we didn't have the pipe? We could nest each function call inside the previous call:

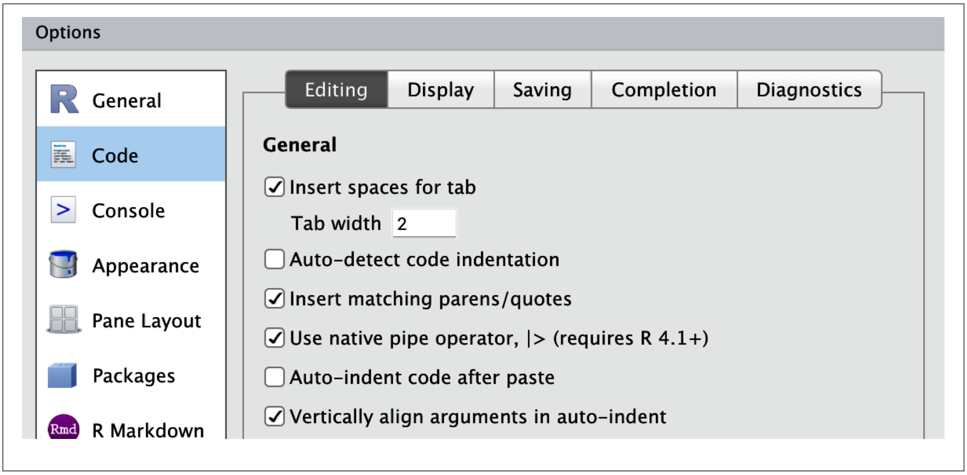
```
arrange(
  select(
    mutate(
      filter(
        flights,
        dest == "IAH"
      ),
      speed = distance / air_time * 60
    ),
    year:day, dep_time, carrier, flight, speed
  ),
  desc(speed)
)
```

Or we could use a bunch of intermediate objects:

```
flights1 <- filter(flights, dest == "IAH")
flights2 <- mutate(flights1, speed = distance / air_time * 60)
flights3 <- select(flights2, year:day, dep_time, carrier, flight, speed)
arrange(flights3, desc(speed))
```

While both forms have their time and place, the pipe generally produces data analysis code that is easier to write and read.

To add the pipe to your code, we recommend using the built-in keyboard shortcut Ctrl/Cmd+Shift+M. You'll need to make one change to your RStudio options to use `|>` instead of `%>%`, as shown in Figure 3-1; more on `%>%` shortly.



General

☑ Insert spaces for tab

Tab width 2

☐ Auto-detect code indentation

☑ Insert matching parens/quotes

☑ Use native pipe operator, |> (requires R 4.1+)

☐ Auto-indent code after paste

☑ Vertically align arguments in auto-indent

Figure 3-1. To insert |>, make sure the "Use native pipe operator" option is checked.

magrittr

If you've been using the tidyverse for a while, you might be familiar with the `%>%` pipe provided by the magrittr package. The magrittr package is included in the core tidyverse, so you can use `%>%` whenever you load the tidyverse:

```
library(tidyverse)

mtcars %>%
  group_by(cyl) %>%
  summarize(n = n())
```

For simple cases, `|>` and `%>%` behave identically. So why do we recommend the base pipe? First, because it's part of base R, it's always available for you to use, even when you're not using the tidyverse. Second, `|>` is quite a bit simpler than `%>%`: in the time between the invention of `%>%` in 2014 and the inclusion of `|>` in R 4.1.0 in 2021, we gained a better understanding of the pipe. This allowed the base implementation to jettison infrequently used and less important features.

Groups

So far you've learned about functions that work with rows and columns. dplyr gets even more powerful when you add in the ability to work with groups. In this section, we'll focus on the most important functions: `group_by()`, `summarize()`, and the slice family of functions.

group_by()

Use `group_by()` to divide your dataset into groups meaningful for your analysis:

```
flights |>
  group_by(month)
#> # A tibble: 336,776 × 19
#> # Groups:   month [12]
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

`group_by()` doesn't change the data, but if you look closely at the output, you'll notice that the output indicates that it is "grouped by" month (`Groups: month [12]`). This means subsequent operations will now work "by month." `group_by()` adds this grouped feature (referred to as *class*) to the data frame, which changes the behavior of the subsequent verbs applied to the data.

summarize()

The most important grouped operation is a summary, which, if being used to calculate a single summary statistic, reduces the data frame to have a single row for each group. In dplyr, this operation is performed by `summarize()`,[3] as shown by the following example, which computes the average departure delay by month:

```
flights |>
  group_by(month) |>
  summarize(
    avg_delay = mean(dep_delay)
  )
#> # A tibble: 12 × 2
#>   month avg_delay
#>   <int>     <dbl>
#> 1     1        NA
#> 2     2        NA
#> 3     3        NA
#> 4     4        NA
#> 5     5        NA
#> 6     6        NA
#> # … with 6 more rows
```

Uh-oh! Something has gone wrong, and all of our results are `NA`s (pronounced "N-A"), R's symbol for missing value. This happened because some of the observed flights

[3] Or `summarise()` if you prefer British English.

had missing data in the delay column, so when we calculated the mean including those values, we got an NA result. We'll come back to discuss missing values in detail in Chapter 18, but for now we'll tell the mean() function to ignore all missing values by setting the argument na.rm to TRUE:

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE)
  )
#> # A tibble: 12 × 2
#>   month delay
#>   <int> <dbl>
#> 1     1  10.0
#> 2     2  10.8
#> 3     3  13.2
#> 4     4  13.9
#> 5     5  13.0
#> 6     6  20.8
#> # … with 6 more rows
```

You can create any number of summaries in a single call to summarize(). You'll learn various useful summaries in the upcoming chapters, but one useful summary is n(), which returns the number of rows in each group:

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  )
#> # A tibble: 12 × 3
#>   month delay     n
#>   <int> <dbl> <int>
#> 1     1  10.0 27004
#> 2     2  10.8 24951
#> 3     3  13.2 28834
#> 4     4  13.9 28330
#> 5     5  13.0 28796
#> 6     6  20.8 28243
#> # … with 6 more rows
```

Means and counts can get you a surprisingly long way in data science!

The slice_ Functions

There are five handy functions that allow you extract specific rows within each group:

`df |> slice_head(n = 1)`
: Takes the first row from each group

`df |> slice_tail(n = 1)`
: Takes the last row in each group

`df |> slice_min(x, n = 1)`
: Takes the row with the smallest value of column x

`df |> slice_max(x, n = 1)`
: Takes the row with the largest value of column x

`df |> slice_sample(n = 1)`
: takes one random row.

You can vary `n` to select more than one row, or instead of `n =`, you can use `prop = 0.1` to select, say, 10% of the rows in each group. For example, the following code finds the flights that are most delayed upon arrival at each destination:

```
flights |>
  group_by(dest) |>
  slice_max(arr_delay, n = 1) |>
  relocate(dest)
#> # A tibble: 108 × 19
#> # Groups:   dest [105]
#>   dest   year month   day dep_time sched_dep_time dep_delay arr_time
#>   <chr> <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1 ABQ    2013     7    22     2145           2007        98      132
#> 2 ACK    2013     7    23     1139            800       219     1250
#> 3 ALB    2013     1    25      123           2000       323      229
#> 4 ANC    2013     8    17     1740           1625        75     2042
#> 5 ATL    2013     7    22     2257            759       898      121
#> 6 AUS    2013     7    10     2056           1505       351     2347
#> # … with 102 more rows, and 11 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>, …
```

Note that there are 105 destinations but we get 108 rows here. What's up? `slice_min()` and `slice_max()` keep tied values, so `n = 1` means give us all rows with the highest value. If you want exactly one row per group, you can set `with_ties = FALSE`.

This is similar to computing the max delay with `summarize()`, but you get the whole corresponding row (or rows if there's a tie) instead of the single summary statistic.

Grouping by Multiple Variables

You can create groups using more than one variable. For example, we could make a group for each date:

```
daily <- flights |>
  group_by(year, month, day)
daily
#> # A tibble: 336,776 × 19
#> # Groups:   year, month, day [365]
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

When you summarize a tibble grouped by more than one variable, each summary peels off the last group. In hindsight, this wasn't a great way to make this function work, but it's difficult to change without breaking existing code. To make it obvious what's happening, dplyr displays a message that tells you how you can change this behavior:

```
daily_flights <- daily |>
  summarize(n = n())
#> `summarise()` has grouped output by 'year', 'month'. You can override using
#> the `.groups` argument.
```

If you're happy with this behavior, you can explicitly request it to suppress the message:

```
daily_flights <- daily |>
  summarize(
    n = n(),
    .groups = "drop_last"
  )
```

Alternatively, change the default behavior by setting a different value, e.g., `"drop"` to drop all grouping or `"keep"` to preserve the same groups.

Ungrouping

You might also want to remove grouping from a data frame without using `summarize()`. You can do this with `ungroup()`:

```
daily |>
  ungroup()
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

Now let's see what happens when you summarize an ungrouped data frame:

```
daily |>
  ungroup() |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    flights = n()
  )
```

```
#> # A tibble: 1 × 2
#>   avg_delay flights
#>       <dbl>   <int>
#> 1      12.6  336776
```

You get a single row back because dplyr treats all the rows in an ungrouped data frame as belonging to one group.

.by

dplyr 1.1.0 includes a new, experimental syntax for per-operation grouping, the `.by` argument. `group_by()` and `ungroup()` aren't going away, but you can now also use the `.by` argument to group within a single operation:

```
flights |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n(),
    .by = month
  )
```

Or if you want to group by multiple variables:

```
flights |>
  summarize(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n(),
    .by = c(origin, dest)
  )
```

`.by` works with all verbs and has the advantage that you don't need to use the `.groups` argument to suppress the grouping message or `ungroup()` when you're done.

We didn't focus on this syntax in this chapter because it was very new when we wrote the book. We did want to mention it because we think it has a lot of promise and it's likely to be quite popular. You can learn more about it in the dplyr 1.1.0 blog post.

Exercises

1. Which carrier has the worst average 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())`.)
2. Find the flights that are most delayed upon departure from each destination.
3. How do delays vary over the course of the day. Illustrate your answer with a plot.
4. What happens if you supply a negative `n` to `slice_min()` and friends?
5. Explain what `count()` does in terms of the dplyr verbs you just learned. What does the `sort` argument to `count()` do?
6. Suppose we have the following tiny data frame:

```
df <- tibble(
  x = 1:5,
  y = c("a", "b", "a", "a", "b"),
  z = c("K", "K", "L", "L", "K")
)
```

a. Write down what you think the output will look like; then check if you were correct and describe what `group_by()` does.

   ```
   df |>
     group_by(y)
   ```

b. Write down what you think the output will look like; then check if you were correct and describe what `arrange()` does. Also comment on how it's different from the `group_by()` in part (a).

   ```
   df |>
     arrange(y)
   ```

c. Write down what you think the output will look like; then check if you were correct and describe what the pipeline does.

   ```
   df |>
     group_by(y) |>
     summarize(mean_x = mean(x))
   ```

d. Write down what you think the output will look like; then check if you were correct and describe what the pipeline does. Then, comment on what the message says.

   ```
   df |>
     group_by(y, z) |>
     summarize(mean_x = mean(x))
   ```

e. Write down what you think the output will look like; then check if you were correct and describe what the pipeline does. How is the output different from the one in part (d)?

   ```
   df |>
     group_by(y, z) |>
     summarize(mean_x = mean(x), .groups = "drop")
   ```

f. Write down what you think the outputs will look like; then check if you were correct and describe what each pipeline does. How are the outputs of the two pipelines different?

   ```
   df |>
     group_by(y, z) |>
     summarize(mean_x = mean(x))

   df |>
     group_by(y, z) |>
     mutate(mean_x = mean(x))
   ```

Case Study: Aggregates and Sample Size

Whenever you do any aggregation, it's always a good idea to include a count (`n()`). That way, you can ensure that you're not drawing conclusions based on very small amounts of data. We'll demonstrate this with some baseball data from the Lahman package. Specifically, we will compare what proportion of times a player gets a hit (`H`) versus the number of times they try to put the ball in play (`AB`):

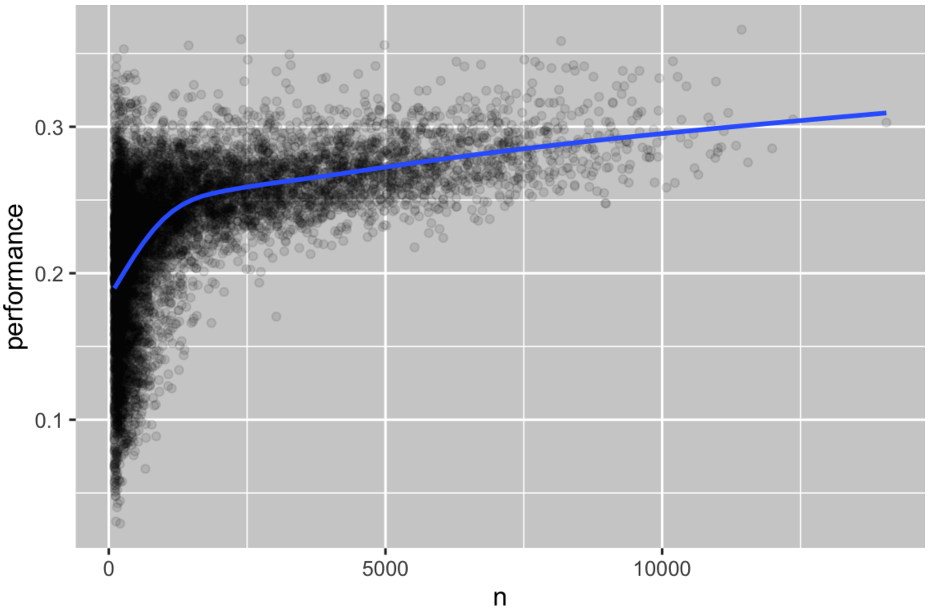
```
batters <- Lahman::Batting |>
  group_by(playerID) |>
  summarize(
    performance = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
    n = sum(AB, na.rm = TRUE)
  )
batters
#> # A tibble: 20,166 × 3
#>   playerID  performance     n
#>   <chr>           <dbl> <int>
#> 1 aardsda01      0          4
#> 2 aaronha01      0.305  12364
#> 3 aaronto01      0.229    944
#> 4 aasedo01       0          5
#> 5 abadan01       0.0952    21
#> 6 abadfe01       0.111      9
#> # … with 20,160 more rows
```

When we plot the skill of the batter (measured by the batting average, `performance`) against the number of opportunities to hit the ball (measured by times at bat, `n`), we see two patterns:

- The variation in `performance` is larger among players with fewer at-bats. The shape of this plot is very characteristic: whenever you plot a mean (or other summary statistics) versus group size, you'll see that the variation decreases as the sample size increases.[4]
- There's a positive correlation between skill (`performance`) and opportunities to hit the ball (`n`) because teams want to give their best batters the most opportunities to hit the ball.

```
batters |>
  filter(n > 100) |>
  ggplot(aes(x = n, y = performance)) +
  geom_point(alpha = 1 / 10) +
  geom_smooth(se = FALSE)
```

[4] *cough* the law of large numbers *cough*



Note the handy pattern for combining ggplot2 and dplyr. You just have to remember to switch from `|>`, for dataset processing, to `+` for adding layers to your plot.

This also has important implications for ranking. If you naively sort on `desc(performance)`, the people with the best batting averages are clearly the ones who tried to put the ball in play very few times and happened to get a hit; they're not necessarily the most skilled players:

```
batters |>
  arrange(desc(performance))
#> # A tibble: 20,166 × 3
#>   playerID  performance     n
#>   <chr>           <dbl> <int>
#> 1 abramge01           1     1
#> 2 alberan01           1     1
#> 3 banisje01           1     1
#> 4 bartocl01           1     1
#> 5 bassdo01            1     1
#> 6 birasst01           1     2
#> # … with 20,160 more rows
```

You can find a good explanation of this problem and how to overcome it on a blog posts by David Robinson and Evan Miller.

Summary

In this chapter, you've learned the tools that dplyr provides for working with data frames. The tools are roughly grouped into three categories: those that manipulate the rows (such as `filter()` and `arrange()`), those that manipulate the columns (such as `select()` and `mutate()`), and those that manipulate groups (such as `group_by()` and `summarize()`). In this chapter, we focused on these "whole data frame" tools, but you haven't yet learned much about what you can do with the individual variable. We'll come back to that in Part III, where each chapter will give you tools for a specific type of variable.

In the next chapter, we'll pivot back to workflow to discuss the importance of code style, keeping your code well organized to make it easy for you and others to read and understand your code.

CHAPTER 4

Workflow: Code Style

Good coding style is like correct punctuation: you can manage without it, butitsuremakesthingseasiertoread. Even as a very new programmer, it's a good idea to work on your code style. Using a consistent style makes it easier for others (including future you!) to read your work and is particularly important if you need to get help from someone else. This chapter will introduce the most important points of the tidyverse style guide, which is used throughout this book.

Styling your code will feel a bit tedious to start with, but if you practice it, it will soon become second nature. Additionally, there are some great tools to quickly restyle existing code, like the styler package by Lorenz Walthert. Once you've installed it with `install.packages("styler")`, an easy way to use it is via RStudio's *command palette*. The command palette lets you use any built-in RStudio command and many addins provided by packages. Open the palette by pressing Cmd/Ctrl+Shift+P and then type *styler* to see all the shortcuts offered by styler. Figure 4-1 shows the results.

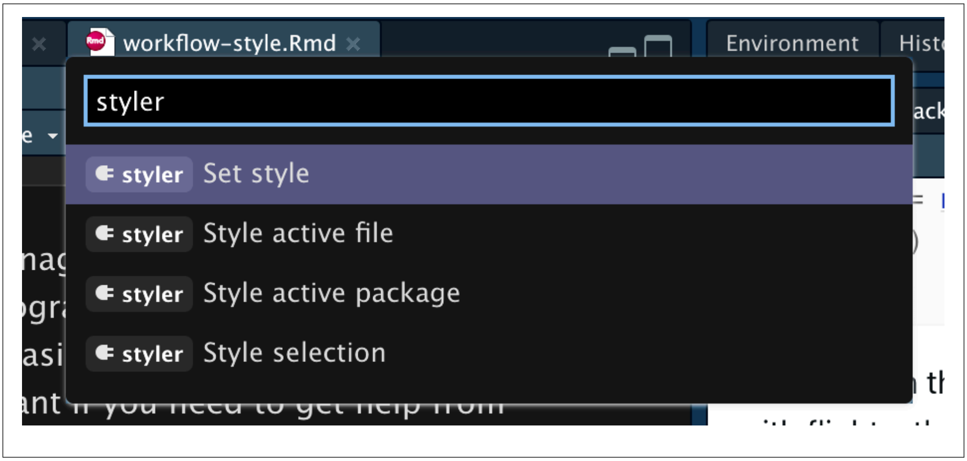


Figure 4-1. RStudio's command palette makes it easy to access every RStudio command using only the keyboard.

We'll use the tidyverse and nycflights13 packages for code examples in this chapter.

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

Names

We talked briefly about names in "What's in a Name?" on page 35. Remember that variable names (those created by `<-` and those created by `mutate()`) should use only lowercase letters, numbers, and _. Use _ to separate words within a name.

```
# Strive for:
short_flights <- flights |> filter(air_time < 60)

# Avoid:
SHORTFLIGHTS <- flights |> filter(air_time < 60)
```

As a general rule of thumb, it's better to prefer long, descriptive names that are easy to understand rather than concise names that are fast to type. Short names save relatively little time when writing code (especially since autocomplete will help you finish typing them), but it can be time-consuming when you come back to old code and are forced to puzzle out a cryptic abbreviation.

If you have a bunch of names for related things, do your best to be consistent. It's easy for inconsistencies to arise when you forget a previous convention, so don't feel bad if you have to go back and rename things. In general, if you have a bunch of variables that are a variation on a theme, you're better off giving them a common prefix rather than a common suffix because autocomplete works best on the start of a variable.

Spaces

Put spaces on either side of mathematical operators apart from `^` (i.e., `+`, `-`, `==`, `<`, ...) and around the assignment operator (`<-`).

```
# Strive for
z <- (a + b)^2 / d

# Avoid
z<-( a + b ) ^ 2/d
```

Don't put spaces inside or outside parentheses for regular function calls. Always put a space after a comma, just like in standard English.

```
# Strive for
mean(x, na.rm = TRUE)

# Avoid
mean (x ,na.rm=TRUE)
```

It's OK to add extra spaces if it improves alignment. For example, if you're creating multiple variables in `mutate()`, you might want to add spaces so that all the `=` line up.[1] This makes it easier to skim the code.

```
flights |>
  mutate(
    speed      = distance / air_time,
    dep_hour   = dep_time %/% 100,
    dep_minute = dep_time %%  100
  )
```

Pipes

`|>` should always have a space before it and should typically be the last thing on a line. This makes it easier to add new steps, rearrange existing steps, modify elements within a step, and get a 10,000-foot view by skimming the verbs on the left side.

```
# Strive for
flights |>
  filter(!is.na(arr_delay), !is.na(tailnum)) |>
  count(dest)

# Avoid
flights|>filter(!is.na(arr_delay), !is.na(tailnum))|>count(dest)
```

If the function you're piping into has named arguments (like `mutate()` or `summarize()`), put each argument on a new line. If the function doesn't have named arguments (like `select()` or `filter()`), keep everything on one line unless it doesn't fit, in which case you should put each argument on its own line.

[1] Since `dep_time` is in `HMM` or `HHMM` format, we use integer division (`%/%`) to get hour and remainder (also known as modulo, `%%`) to get minute.

```
# Strive for
flights |>
  group_by(tailnum) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

# Avoid
flights |>
  group_by(
    tailnum
  ) |>
  summarize(delay = mean(arr_delay, na.rm = TRUE), n = n())
```

After the first step of the pipeline, indent each line by two spaces. RStudio automatically puts the spaces in for you after a line break following a `|>`. If you're putting each argument on its own line, indent by an extra two spaces. Make sure `)` is on its own line and unindented to match the horizontal position of the function name.

```
# Strive for
flights |>
  group_by(tailnum) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

# Avoid
flights|>
  group_by(tailnum) |>
  summarize(
             delay = mean(arr_delay, na.rm = TRUE),
             n = n()
           )

# Avoid
flights|>
  group_by(tailnum) |>
  summarize(
  delay = mean(arr_delay, na.rm = TRUE),
  n = n()
  )
```

It's OK to shirk some of these rules if your pipeline fits easily on one line. But in our collective experience, it's common for short snippets to grow longer, so you'll usually save time in the long run by starting with all the vertical space you need.

```
# This fits compactly on one line
df |> mutate(y = x + 1)

# While this takes up 4x as many lines, it's easily extended to
# more variables and more steps in the future
df |>
  mutate(
    y = x + 1
  )
```

Finally, be wary of writing very long pipes, say longer than 10–15 lines. Try to break them up into smaller subtasks, giving each task an informative name. The names will help cue the reader into what's happening and makes it easier to check that intermediate results are as expected. Whenever you can give something an informative name, you should, for example when you fundamentally change the structure of the data, e.g., after pivoting or summarizing. Don't expect to get it right the first time! This means breaking up long pipelines if there are intermediate states that can get good names.

ggplot2

The same basic rules that apply to the pipe also apply to ggplot2; just treat + the same way as |>:

```
flights |>
  group_by(month) |>
  summarize(
    delay = mean(arr_delay, na.rm = TRUE)
  ) |>
  ggplot(aes(x = month, y = delay)) +
  geom_point() +
  geom_line()
```

Again, if you can't fit all of the arguments to a function onto a single line, put each argument on its own line:

```
flights |>
  group_by(dest) |>
  summarize(
    distance = mean(distance),
    speed = mean(distance / air_time, na.rm = TRUE)
  ) |>
  ggplot(aes(x = distance, y = speed)) +
  geom_smooth(
    method = "loess",
    span = 0.5,
    se = FALSE,
    color = "white",
    linewidth = 4
  ) +
  geom_point()
```

Watch for the transition from |> to +. We wish this transition wasn't necessary, but unfortunately, ggplot2 was written before the pipe was discovered.

Sectioning Comments

As your scripts get longer, you can use *sectioning* comments to break up your file into manageable pieces:

```
# Load data --------------------------------------

# Plot data --------------------------------------
```

RStudio provides a keyboard shortcut to create these headers (Cmd/Ctrl+Shift+R) and will display them in the code navigation drop-down at the bottom left of the editor, as shown in Figure 4-2.

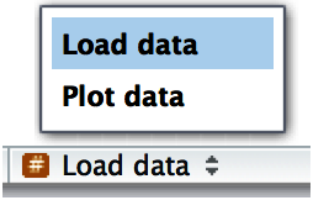


Figure 4-2. After adding sectioning comments to your script, you can easily navigate to them using the code navigation tool in the bottom left of the script editor.

Exercises

1. Restyle the following pipelines following the previous guidelines:

```
flights|>filter(dest=="IAH")|>group_by(year,month,day)|>summarize(n=n(),
delay=mean(arr_delay,na.rm=TRUE))|>filter(n>10)

flights|>filter(carrier=="UA",dest%in%c("IAH","HOU"),sched_dep_time>
0900,sched_arr_time<2000)|>group_by(flight)|>summarize(delay=mean(
arr_delay,na.rm=TRUE),cancelled=sum(is.na(arr_delay)),n=n())|>filter(n>10)
```

Summary

In this chapter, you learned the most important principles of code style. These may feel like a set of arbitrary rules to start with (because they are!), but over time, as you write more code and share code with more people, you'll see how important a consistent style is. And don't forget about the styler package: it's a great way to quickly improve the quality of poorly styled code.

In the next chapter, we switch back to data science tools, learning about tidy data. Tidy data is a consistent way of organizing your data frames that is used throughout the tidyverse. This consistency makes your life easier because once you have tidy data, it just works with the vast majority of tidyverse functions. Of course, life is never easy, and most datasets you encounter in the wild will not already be tidy. So we'll also teach you how to use the tidyr package to tidy your untidy data.

CHAPTER 5

Data Tidying

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 using a system called *tidy data*. Getting your data into this format requires some work up front, 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 data questions you care about.

In this chapter, you'll first learn the definition of tidy data and see it applied to a simple toy dataset. Then we'll dive into the primary tool you'll use for tidying data: pivoting. Pivoting allows you to change the form of your data without changing any of the values.

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)
```

From this chapter on, we'll suppress the loading message from `library(tidyverse)`.

Tidy Data

You can represent the same underlying data in multiple ways. The following example shows the same data organized in three different ways. Each dataset shows the same values of four variables: *country*, *year*, *population*, and number of documented *cases* of tuberculosis (TB), but each dataset organizes the values in a different way.

```
table1
#> # A tibble: 6 × 4
#>   country      year  cases population
#>   <chr>       <dbl>  <dbl>      <dbl>
#> 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>       <dbl> <chr>          <dbl>
#> 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>     <dbl>     <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
```

These are all representations of the same underlying data, but they are not equally easy to use. One of them, `table1`, will be much easier to work with inside the tidyverse because it's *tidy*.

There are three interrelated rules that make a dataset tidy:

1. Each variable is a column; each column is a variable.
2. Each observation is a row; each row is an observation.
3. Each value is a cell; each cell is a single value.

Figure 5-1 shows the rules visually.

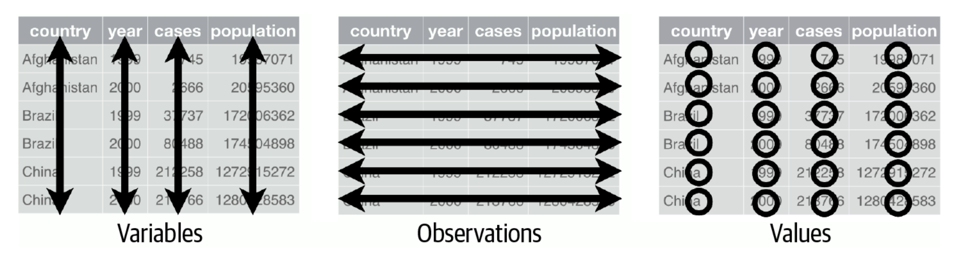


Figure 5-1. Three rules make a dataset tidy: variables are columns, observations are rows, and values are cells.

Why ensure that your data is tidy? There are two main advantages:

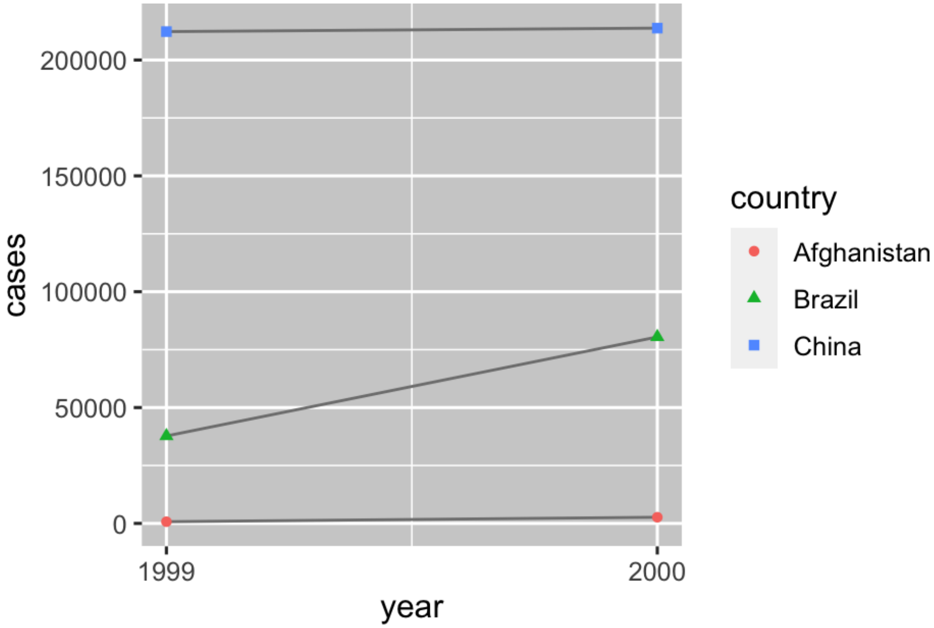
1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
2. There's a specific advantage to placing variables in columns because it allows R's vectorized nature to shine. As you learned in "mutate()" on page 47 and "summarize()" on page 54, 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 few 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>       <dbl>  <dbl>      <dbl> <dbl>
#> 1 Afghanistan  1999    745   19987071 0.373
#> 2 Afghanistan  2000   2666   20595360 1.29
#> 3 Brazil       1999  37737  172006362 2.19
#> 4 Brazil       2000  80488  174504898 4.61
#> 5 China        1999 212258 1272915272 1.67
#> 6 China        2000 213766 1280428583 1.67

# Compute total cases per year
table1 |>
  group_by(year) |>
  summarize(total_cases = sum(cases))
#> # A tibble: 2 × 2
#>    year total_cases
#>   <dbl>       <dbl>
#> 1  1999      250740
#> 2  2000      296920
```

```
# Visualize changes over time
ggplot(table1, aes(x = year, y = cases)) +
  geom_line(aes(group = country), color = "grey50") +
  geom_point(aes(color = country, shape = country)) +
  scale_x_continuous(breaks = c(1999, 2000)) # x-axis breaks at 1999 and 2000
```



Exercises

1. For each of the sample tables, describe what each observation and each column represents.
2. Sketch out the process you'd use to calculate the `rate` for `table2` and `table3`. 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.

   You haven't yet learned all the functions you'd need to actually perform these operations, but you should still be able to think through the transformations you'd need.

Lengthening Data

The principles of tidy data might seem so obvious that you wonder if you'll ever encounter a dataset that isn't tidy. Unfortunately, however, most real data is untidy. There are two main reasons:

1. Data is often organized to facilitate some goal other than analysis. For example, it's common for data to be structured to make data entry, not analysis, easy.
2. 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.

This means that most real analyses will require at least a little tidying. You'll begin by figuring out what the underlying variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data. Next, you'll *pivot* your data into a tidy form, with variables in the columns and observations in the rows.

tidyr provides two functions for pivoting data: `pivot_longer()` and `pivot_wider()`. We'll first start with `pivot_longer()` because it's the most common case. Let's dive into some examples.

Data in Column Names

The `billboard` dataset records the Billboard rank of songs in the year 2000:

```
billboard
#> # A tibble: 317 × 79
#>   artist       track               date.entered   wk1   wk2   wk3   wk4   wk5
#>   <chr>        <chr>               <date>       <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 2 Pac        Baby Don't Cry (Ke… 2000-02-26      87    82    72    77    87
#> 2 2Ge+her      The Hardest Part O… 2000-09-02      91    87    92    NA    NA
#> 3 3 Doors Down Kryptonite          2000-04-08      81    70    68    67    66
#> 4 3 Doors Down Loser               2000-10-21      76    76    72    69    67
#> 5 504 Boyz     Wobble Wobble       2000-04-15      57    34    25    17    17
#> 6 98^0         Give Me Just One N… 2000-08-19      51    39    34    26    26
#> # … with 311 more rows, and 71 more variables: wk6 <dbl>, wk7 <dbl>,
#> #   wk8 <dbl>, wk9 <dbl>, wk10 <dbl>, wk11 <dbl>, wk12 <dbl>, wk13 <dbl>, …
```

In this dataset, each observation is a song. The first three columns (`artist`, `track` and `date.entered`) are variables that describe the song. Then we have 76 columns (`wk1-wk76`) that describe the rank of the song in each week.[1] Here, the column names are one variable (the `week`), and the cell values are another (the `rank`).

To tidy this data, we'll use `pivot_longer()`:

[1] The song will be included as long as it was in the top 100 at some point in 2000 and is tracked for up to 72 weeks after it appears.

```
billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank"
  )
#> # A tibble: 24,092 × 5
#>    artist track                   date.entered week   rank
#>    <chr>  <chr>                   <date>       <chr> <dbl>
#>  1 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk1      87
#>  2 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk2      82
#>  3 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk3      72
#>  4 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk4      77
#>  5 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk5      87
#>  6 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk6      94
#>  7 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk7      99
#>  8 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk8      NA
#>  9 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk9      NA
#> 10 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk10     NA
#> # … with 24,082 more rows
```

After the data, there are three key arguments:

`cols`

Specifies which columns need to be pivoted (i.e., which columns aren't variables). This argument uses the same syntax as `select()`, so here we could use `!c(artist, track, date.entered)` or `starts_with("wk")`.

`names_to`

Names the variable stored in the column names; we named that variable `week`.

`values_to`

Names the variable stored in the cell values; we named that variable `rank`.

Note that in the code `"week"` and `"rank"` are quoted because those are new variables we're creating; they don't yet exist in the data when we run the `pivot_longer()` call.

Now let's turn our attention to the resulting longer data frame. What happens if a song is in the top 100 for less than 76 weeks? Take 2 Pac's "Baby Don't Cry," for example. The previous output suggests that it was only in the top 100 for 7 weeks, and all the remaining weeks are filled in with missing values. These `NA`s don't really represent unknown observations; they were forced to exist by the structure of the dataset,[2] so we can ask `pivot_longer()` to get rid of them by setting `values_drop_na = TRUE`:

[2] We'll come back to this idea in Chapter 18.

```
billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE
  )
#> # A tibble: 5,307 × 5
#>   artist track                   date.entered week   rank
#>   <chr>  <chr>                   <date>       <chr> <dbl>
#> 1 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk1      87
#> 2 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk2      82
#> 3 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk3      72
#> 4 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk4      77
#> 5 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk5      87
#> 6 2 Pac  Baby Don't Cry (Keep... 2000-02-26   wk6      94
#> # … with 5,301 more rows
```

The number of rows is now much lower, indicating that many rows with NAs were dropped.

You might also wonder what happens if a song is in the top 100 for more than 76 weeks. We can't tell from this data, but you might guess that additional columns such as wk77, wk78, … would be added to the dataset.

This data is now tidy, but we could make future computation a bit easier by converting values of week from character strings to numbers using mutate() and readr::parse_number(). parse_number() is a handy function that will extract the first number from a string, ignoring all other text.

```
billboard_longer <- billboard |>
  pivot_longer(
    cols = starts_with("wk"),
    names_to = "week",
    values_to = "rank",
    values_drop_na = TRUE
  ) |>
  mutate(
    week = parse_number(week)
  )
billboard_longer
#> # A tibble: 5,307 × 5
#>   artist track                   date.entered  week  rank
#>   <chr>  <chr>                   <date>       <dbl> <dbl>
#> 1 2 Pac  Baby Don't Cry (Keep... 2000-02-26       1    87
#> 2 2 Pac  Baby Don't Cry (Keep... 2000-02-26       2    82
#> 3 2 Pac  Baby Don't Cry (Keep... 2000-02-26       3    72
#> 4 2 Pac  Baby Don't Cry (Keep... 2000-02-26       4    77
#> 5 2 Pac  Baby Don't Cry (Keep... 2000-02-26       5    87
#> 6 2 Pac  Baby Don't Cry (Keep... 2000-02-26       6    94
#> # … with 5,301 more rows
```

Now that we have all the week numbers in one variable and all the rank values in another, we're in a good position to visualize how song ranks vary over time. The code is shown here and the result is in Figure 5-2. We can see that very few songs stay in the top 100 for more than 20 weeks.

```
billboard_longer |>
  ggplot(aes(x = week, y = rank, group = track)) +
  geom_line(alpha = 0.25) +
  scale_y_reverse()
```

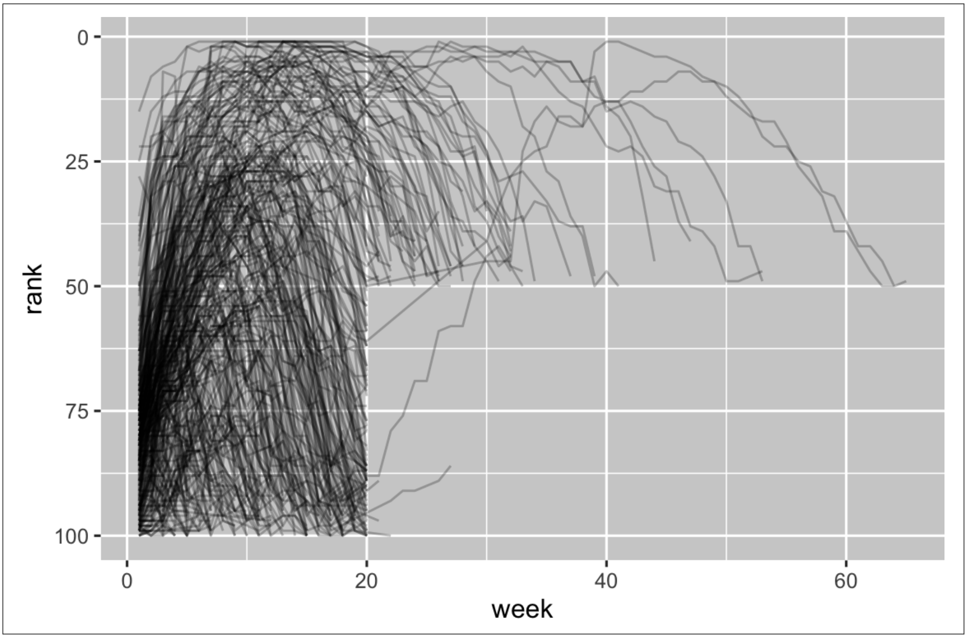


Figure 5-2. A line plot showing how the rank of a song changes over time.

How Does Pivoting Work?

Now that you've seen how we can use pivoting to reshape our data, let's take a little time to gain some intuition about what pivoting does to the data. Let's start with a simple dataset to make it easier to see what's happening. Suppose we have three patients with `id`s A, B, and C, and we take two blood pressure measurements on each patient. We'll create the data with `tribble()`, a handy function for constructing small tibbles by hand:

```
df <- tribble(
  ~id,  ~bp1, ~bp2,
   "A",  100,  120,
   "B",  140,  115,
   "C",  120,  125
)
```

We want our new dataset to have three variables: `id` (already exists), `measurement` (the column names), and `value` (the cell values). To achieve this, we need to pivot `df` longer:

```
df |>
  pivot_longer(
    cols = bp1:bp2,
    names_to = "measurement",
    values_to = "value"
  )
#> # A tibble: 6 × 3
#>   id    measurement value
#>   <chr> <chr>       <dbl>
#> 1 A     bp1           100
#> 2 A     bp2           120
#> 3 B     bp1           140
#> 4 B     bp2           115
#> 5 C     bp1           120
#> 6 C     bp2           125
```

How does the reshaping work? It's easier to see if we think about it column by column. As shown in Figure 5-3, the values in the column that was already a variable in the original dataset (`id`) need to be repeated, once for each column that is pivoted.

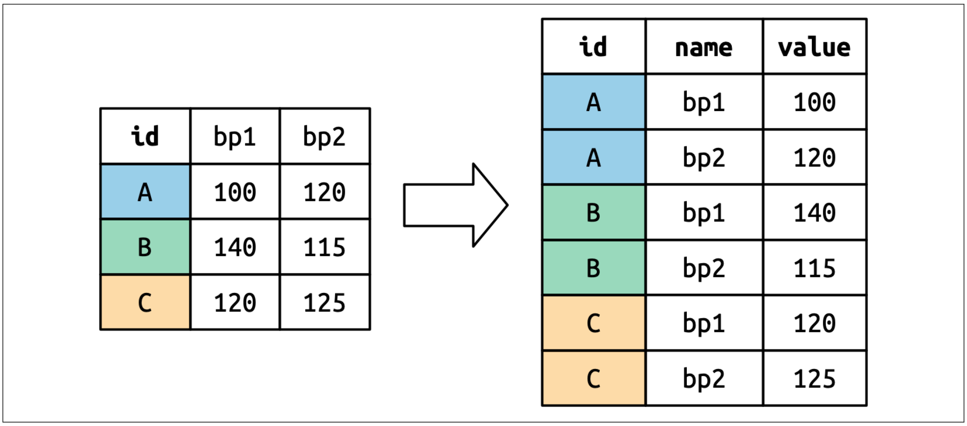
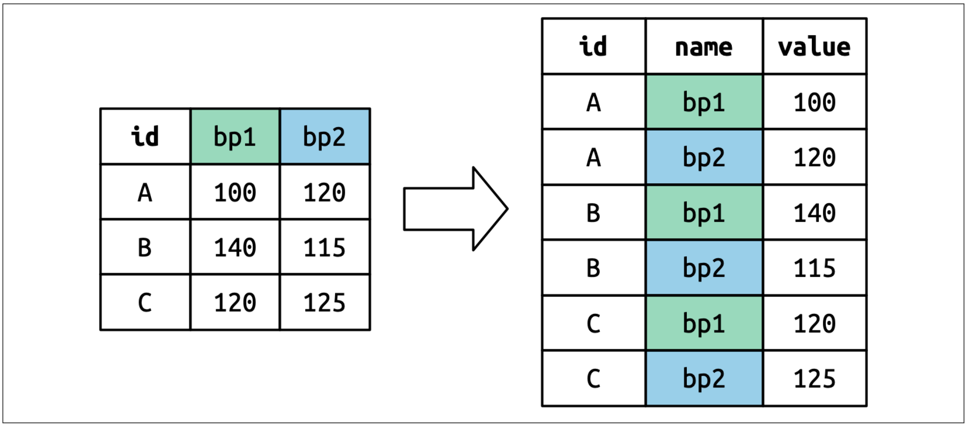


Figure 5-3. Columns that are already variables need to be repeated, once for each column that is pivoted.

The column names become values in a new variable, whose name is defined by `names_to`, as shown in Figure 5-4. They need to be repeated once for each row in the original dataset.

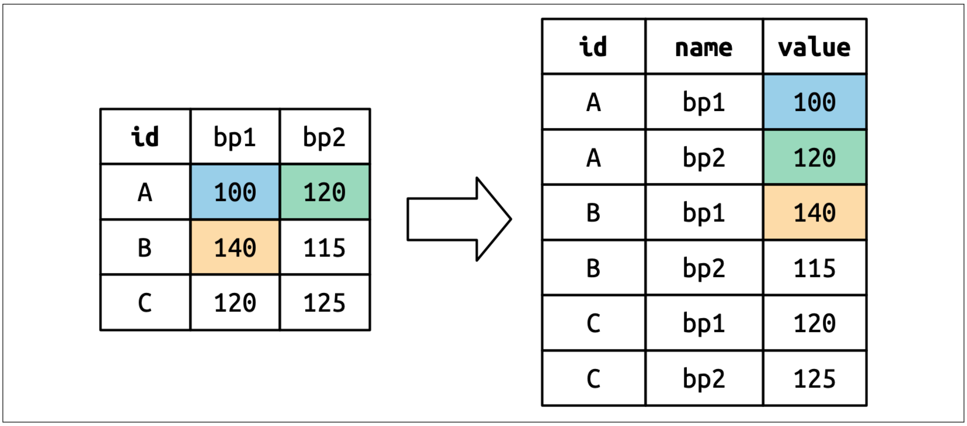


id	bp1	bp2
A	100	120
B	140	115
C	120	125

id	name	value
A	bp1	100
A	bp2	120
B	bp1	140
B	bp2	115
C	bp1	120
C	bp2	125

Figure 5-4. The column names of pivoted columns become values in a new column. The values need to be repeated once for each row of the original dataset.

The cell values also become values in a new variable, with a name defined by `values_to`. They are unwound row by row. Figure 5-5 illustrates the process.



id	bp1	bp2
A	100	120
B	140	115
C	120	125

id	name	value
A	bp1	100
A	bp2	120
B	bp1	140
B	bp2	115
C	bp1	120
C	bp2	125

Figure 5-5. The number of values is preserved (not repeated) but unwound row by row.

Many Variables in Column Names

A more challenging situation occurs when you have multiple pieces of information crammed into the column names and you would like to store these in separate new variables. For example, take the `who2` dataset, the source of `table1`, and friends that you saw earlier:

```
who2
#> # A tibble: 7,240 × 58
#>   country      year sp_m_014 sp_m_1524 sp_m_2534 sp_m_3544 sp_m_4554
#>   <chr>       <dbl>    <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
#> 1 Afghanistan  1980       NA        NA        NA        NA        NA
#> 2 Afghanistan  1981       NA        NA        NA        NA        NA
#> 3 Afghanistan  1982       NA        NA        NA        NA        NA
#> 4 Afghanistan  1983       NA        NA        NA        NA        NA
#> 5 Afghanistan  1984       NA        NA        NA        NA        NA
#> 6 Afghanistan  1985       NA        NA        NA        NA        NA
#> # … with 7,234 more rows, and 51 more variables: sp_m_5564 <dbl>,
#> #   sp_m_65 <dbl>, sp_f_014 <dbl>, sp_f_1524 <dbl>, sp_f_2534 <dbl>, …
```

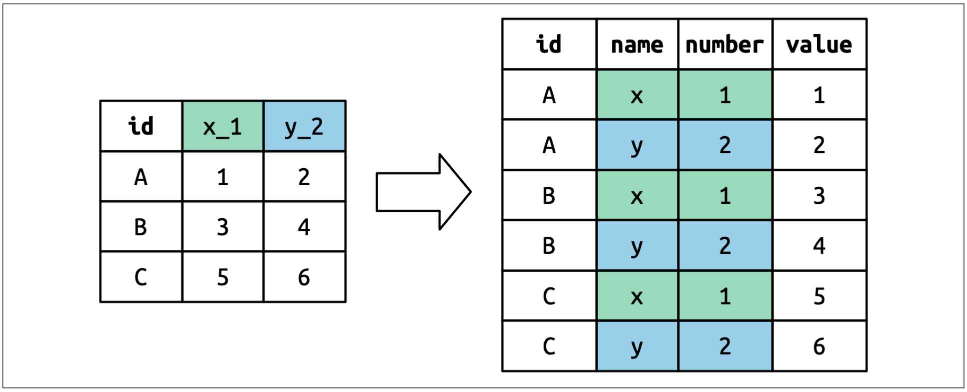
This dataset, collected by the World Health Organization, records information about tuberculosis diagnoses. There are two columns that are already variables and are easy to interpret: country and year. They are followed by 56 columns like sp_m_014, ep_m_4554, and rel_m_3544. If you stare at these columns for long enough, you'll notice there's a pattern. Each column name is made up of three pieces separated by _. The first piece, sp/rel/ep, describes the method used for the diagnosis; the second piece, m/f, is the gender (coded as a binary variable in this dataset); and the third piece, 014/1524/2534/3544/4554/65, is the age range (014 represents 0–14, for example).

So in this case we have six pieces of information recorded in who2: the country and the year (already columns); the method of diagnosis, the gender category, and the age range category (contained in the other column names); and the count of patients in that category (cell values). To organize these six pieces of information in six separate columns, we use pivot_longer() with a vector of column names for names_to and instructors for splitting the original variable names into pieces for names_sep as well as a column name for values_to:

```
who2 |>
  pivot_longer(
    cols = !(country:year),
    names_to = c("diagnosis", "gender", "age"),
    names_sep = "_",
    values_to = "count"
  )
#> # A tibble: 405,440 × 6
#>   country      year diagnosis gender age   count
#>   <chr>       <dbl> <chr>     <chr>  <chr> <dbl>
#> 1 Afghanistan  1980 sp        m      014      NA
#> 2 Afghanistan  1980 sp        m      1524     NA
#> 3 Afghanistan  1980 sp        m      2534     NA
#> 4 Afghanistan  1980 sp        m      3544     NA
#> 5 Afghanistan  1980 sp        m      4554     NA
#> 6 Afghanistan  1980 sp        m      5564     NA
#> # … with 405,434 more rows
```

An alternative to names_sep is names_pattern, which you can use to extract variables from more complicated naming scenarios, once you've learned about regular expressions in Chapter 15.

Conceptually, this is only a minor variation on the simpler case you've already seen. Figure 5-6 shows the basic idea: now, instead of the column names pivoting into a single column, they pivot into multiple columns. You can imagine this happening in two steps (first pivoting and then separating), but under the hood it happens in a single step because that's faster.



id	x_1	y_2
A	1	2
B	3	4
C	5	6

id	name	number	value
A	x	1	1
A	y	2	2
B	x	1	3
B	y	2	4
C	x	1	5
C	y	2	6

Figure 5-6. Pivoting columns with multiple pieces of information in the names means that each column name now fills in values in multiple output columns.

Data and Variable Names in the Column Headers

The next step up in complexity is when the column names include a mix of variable values and variable names. For example, take the `household` dataset:

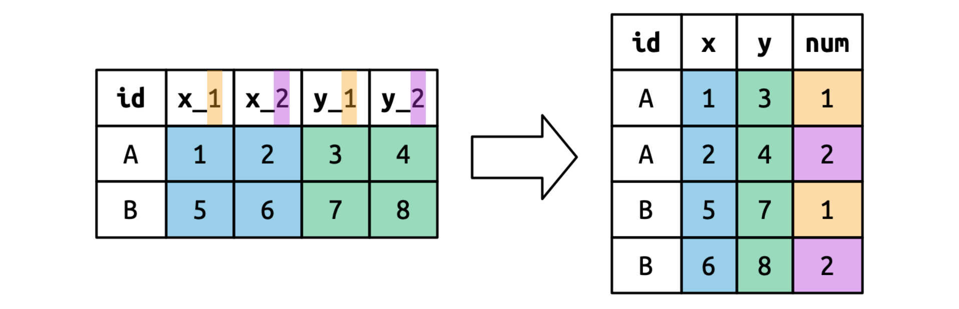
```
household
#> # A tibble: 5 × 5
#>   family dob_child1 dob_child2 name_child1 name_child2
#>    <int> <date>     <date>     <chr>       <chr>
#> 1      1 1998-11-26 2000-01-29 Susan       Jose
#> 2      2 1996-06-22 NA         Mark        <NA>
#> 3      3 2002-07-11 2004-04-05 Sam         Seth
#> 4      4 2004-10-10 2009-08-27 Craig       Khai
#> 5      5 2000-12-05 2005-02-28 Parker      Gracie
```

This dataset contains data about five families, with the names and dates of birth of up to two children. The new challenge in this dataset is that the column names contain the names of two variables (`dob`, `name`) and the values of another (`child`, with values 1 or 2). To solve this problem we again need to supply a vector to `names_to` but this time we use the special `".value"` sentinel; this isn't the name of a variable but a unique value that tells `pivot_longer()` to do something different. This overrides the usual `values_to` argument to use the first component of the pivoted column name as a variable name in the output.

```
household |>
  pivot_longer(
    cols = !family,
    names_to = c(".value", "child"),
    names_sep = "_",
    values_drop_na = TRUE
  )
#> # A tibble: 9 × 4
#>   family child  dob        name
#>    <int> <chr>  <date>     <chr>
#> 1      1 child1 1998-11-26 Susan
#> 2      1 child2 2000-01-29 Jose
#> 3      2 child1 1996-06-22 Mark
#> 4      3 child1 2002-07-11 Sam
#> 5      3 child2 2004-04-05 Seth
#> 6      4 child1 2004-10-10 Craig
#> # … with 3 more rows
```

We again use `values_drop_na = TRUE`, since the shape of the input forces the creation of explicit missing variables (e.g., for families with only one child).

Figure 5-7 illustrates the basic idea with a simpler example. When you use `".value"` in `names_to`, the column names in the input contribute to both values and variable names in the output.



id	x_1	x_2	y_1	y_2
A	1	2	3	4
B	5	6	7	8

id	x	y	num
A	1	3	1
A	2	4	2
B	5	7	1
B	6	8	2

Figure 5-7. Pivoting with `names_to = c(".value", "num")` splits the column names into two components: the first part determines the output column name (`x` or `y`), and the second part determines the value of the `num` column.

Widening Data

So far we've used `pivot_longer()` to solve the common class of problems where values have ended up in column names. Next we'll pivot (HA HA) to `pivot_wider()`, which makes datasets *wider* by increasing columns and reducing rows and helps when one observation is spread across multiple rows. This seems to arise less commonly in the wild, but it does seem to crop up a lot when dealing with governmental data.

We'll start by looking at `cms_patient_experience`, a dataset from the Centers of Medicare and Medicaid services that collects data about patient experiences:

```
cms_patient_experience
#> # A tibble: 500 × 5
#>   org_pac_id org_nm                     measure_cd   measure_title   prf_rate
#>   <chr>      <chr>                      <chr>        <chr>              <dbl>
#> 1 0446157747 USC CARE MEDICAL GROUP INC CAHPS_GRP_1  CAHPS for MIPS…       63
#> 2 0446157747 USC CARE MEDICAL GROUP INC CAHPS_GRP_2  CAHPS for MIPS…       87
#> 3 0446157747 USC CARE MEDICAL GROUP INC CAHPS_GRP_3  CAHPS for MIPS…       86
#> 4 0446157747 USC CARE MEDICAL GROUP INC CAHPS_GRP_5  CAHPS for MIPS…       57
#> 5 0446157747 USC CARE MEDICAL GROUP INC CAHPS_GRP_8  CAHPS for MIPS…       85
#> 6 0446157747 USC CARE MEDICAL GROUP INC CAHPS_GRP_12 CAHPS for MIPS…       24
#> # … with 494 more rows
```

The core unit being studied is an organization, but each organization is spread across six rows, with one row for each measurement taken in the survey organization. We can see the complete set of values for `measure_cd` and `measure_title` by using `distinct()`:

```
cms_patient_experience |>
  distinct(measure_cd, measure_title)
#> # A tibble: 6 × 2
#>   measure_cd   measure_title
#>   <chr>        <chr>
#> 1 CAHPS_GRP_1  CAHPS for MIPS SSM: Getting Timely Care, Appointments, and In…
#> 2 CAHPS_GRP_2  CAHPS for MIPS SSM: How Well Providers Communicate
#> 3 CAHPS_GRP_3  CAHPS for MIPS SSM: Patient's Rating of Provider
#> 4 CAHPS_GRP_5  CAHPS for MIPS SSM: Health Promotion and Education
#> 5 CAHPS_GRP_8  CAHPS for MIPS SSM: Courteous and Helpful Office Staff
#> 6 CAHPS_GRP_12 CAHPS for MIPS SSM: Stewardship of Patient Resources
```

Neither of these columns will make particularly great variable names: `measure_cd` doesn't hint at the meaning of the variable, and `measure_title` is a long sentence containing spaces. We'll use `measure_cd` as the source for our new column names for now, but in a real analysis you might want to create your own variable names that are both short and meaningful.

`pivot_wider()` has the opposite interface to `pivot_longer()`: instead of choosing new column names, we need to provide the existing columns that define the values (`values_from`) and the column name (`names_from`):

```
cms_patient_experience |>
  pivot_wider(
    names_from = measure_cd,
    values_from = prf_rate
  )
#> # A tibble: 500 × 9
#>   org_pac_id org_nm                   measure_title   CAHPS_GRP_1 CAHPS_GRP_2
#>   <chr>      <chr>                    <chr>                 <dbl>       <dbl>
#> 1 0446157747 USC CARE MEDICAL GROUP … CAHPS for MIPS…          63          NA
#> 2 0446157747 USC CARE MEDICAL GROUP … CAHPS for MIPS…          NA          87
#> 3 0446157747 USC CARE MEDICAL GROUP … CAHPS for MIPS…          NA          NA
#> 4 0446157747 USC CARE MEDICAL GROUP … CAHPS for MIPS…          NA          NA
#> 5 0446157747 USC CARE MEDICAL GROUP … CAHPS for MIPS…          NA          NA
#> 6 0446157747 USC CARE MEDICAL GROUP … CAHPS for MIPS…          NA          NA
#> # … with 494 more rows, and 4 more variables: CAHPS_GRP_3 <dbl>,
#> #   CAHPS_GRP_5 <dbl>, CAHPS_GRP_8 <dbl>, CAHPS_GRP_12 <dbl>
```

The output doesn't look quite right; we still seem to have multiple rows for each organization. That's because we also need to tell `pivot_wider()` which column or columns have values that uniquely identify each row; in this case those are the variables starting with `"org"`:

```
cms_patient_experience |>
  pivot_wider(
    id_cols = starts_with("org"),
    names_from = measure_cd,
    values_from = prf_rate
  )
#> # A tibble: 95 × 8
#>   org_pac_id org_nm           CAHPS_GRP_1 CAHPS_GRP_2 CAHPS_GRP_3 CAHPS_GRP_5
#>   <chr>      <chr>                  <dbl>       <dbl>       <dbl>       <dbl>
#> 1 0446157747 USC CARE MEDICA…          63          87          86          57
#> 2 0446162697 ASSOCIATION OF …          59          85          83          63
#> 3 0547164295 BEAVER MEDICAL …          49          NA          75          44
#> 4 0749333730 CAPE PHYSICIANS…          67          84          85          65
#> 5 0840104360 ALLIANCE PHYSIC…          66          87          87          64
#> 6 0840109864 REX HOSPITAL INC          73          87          84          67
#> # … with 89 more rows, and 2 more variables: CAHPS_GRP_8 <dbl>,
#> #   CAHPS_GRP_12 <dbl>
```

This gives us the output that we're looking for.

How Does pivot_wider() Work?

To understand how `pivot_wider()` works, let's again start with a simple dataset. This time we have two patients with `id`s A and B; we have three blood pressure measurements on patient A and two on patient B:

```
df <- tribble(
  ~id, ~measurement, ~value,
  "A",        "bp1",    100,
  "B",        "bp1",    140,
  "B",        "bp2",    115,
  "A",        "bp2",    120,
  "A",        "bp3",    105
)
```

We'll take the values from the `value` column and the names from the `measurement` column:

```
df |>
  pivot_wider(
    names_from = measurement,
    values_from = value
  )
#> # A tibble: 2 × 4
#>   id      bp1   bp2   bp3
#>   <chr> <dbl> <dbl> <dbl>
#> 1 A       100   120   105
#> 2 B       140   115    NA
```

To begin the process, pivot_wider() needs to first figure out what will go in the rows and columns. The new column names will be the unique values of measurement:

```
df |>
  distinct(measurement) |>
  pull()
#> [1] "bp1" "bp2" "bp3"
```

By default, the rows in the output are determined by all the variables that aren't going into the new names or values. These are called the id_cols. Here there is only one column, but in general there can be any number:

```
df |>
  select(-measurement, -value) |>
  distinct()
#> # A tibble: 2 × 1
#>   id
#>   <chr>
#> 1 A
#> 2 B
```

pivot_wider() then combines these results to generate an empty data frame:

```
df |>
  select(-measurement, -value) |>
  distinct() |>
  mutate(x = NA, y = NA, z = NA)
#> # A tibble: 2 × 4
#>   id    x     y     z
#>   <chr> <lgl> <lgl> <lgl>
#> 1 A     NA    NA    NA
#> 2 B     NA    NA    NA
```

It then fills in all the missing values using the data in the input. In this case, not every cell in the output has a corresponding value in the input as there's no third blood pressure measurement for patient B, so that cell remains missing. We'll come back to this idea that pivot_wider() can "make" missing values in Chapter 18.

You might also wonder what happens if there are multiple rows in the input that correspond to one cell in the output. The following example has two rows that correspond to id A and measurement bp1:

```
df <- tribble(
  ~id, ~measurement, ~value,
  "A",        "bp1",    100,
  "A",        "bp1",    102,
  "A",        "bp2",    120,
  "B",        "bp1",    140,
  "B",        "bp2",    115
)
```

If we attempt to pivot this, we get an output that contains list-columns, which you'll learn more about in Chapter 23:

```
df |>
  pivot_wider(
    names_from = measurement,
    values_from = value
  )
#> Warning: Values from `value` are not uniquely identified; output will contain
#> list-cols.
#> • Use `values_fn = list` to suppress this warning.
#> • Use `values_fn = {summary_fun}` to summarise duplicates.
#> • Use the following dplyr code to identify duplicates.
#>   {data} %>%
#>   dplyr::group_by(id, measurement) %>%
#>   dplyr::summarise(n = dplyr::n(), .groups = "drop") %>%
#>   dplyr::filter(n > 1L)
#> # A tibble: 2 × 3
#>   id    bp1       bp2
#>   <chr> <list>    <list>
#> 1 A     <dbl [2]> <dbl [1]>
#> 2 B     <dbl [1]> <dbl [1]>
```

Since you don't know how to work with this sort of data yet, you'll want to follow the hint in the warning to figure out where the problem is:

```
df |>
  group_by(id, measurement) |>
  summarize(n = n(), .groups = "drop") |>
  filter(n > 1)
#> # A tibble: 1 × 3
#>   id    measurement     n
#>   <chr> <chr>       <int>
#> 1 A     bp1             2
```

It's then up to you to figure out what's gone wrong with your data and either repair the underlying damage or use your grouping and summarizing skills to ensure that each combination of row and column values has only a single row.

Summary

In this chapter you learned about tidy data: data that has variables in columns and observations in rows. Tidy data makes working in the tidyverse easier, because it's a consistent structure understood by most functions; the main challenge is transforming the data from whatever structure you receive it in to a tidy format. To that end, you learned about `pivot_longer()` and `pivot_wider()`, which allow you to tidy up many untidy datasets. The examples we presented here are a selection of those from `vignette("pivot", package = "tidyr")`, so if you encounter a problem that this chapter doesn't help you with, that vignette is a good place to try next.

Another challenge is that, for a given dataset, it can be impossible to label the longer or the wider version as the "tidy" one. This is partly a reflection of our definition of tidy data, where we said tidy data has one variable in each column, but we didn't actually define what a variable is (and it's surprisingly hard to do so). It's totally fine to be pragmatic and to say a variable is whatever makes your analysis easiest. So if you're stuck figuring out how to do some computation, consider switching up the organization of your data; don't be afraid to untidy, transform, and re-tidy as needed!

If you enjoyed this chapter and want to learn more about the underlying theory, you can learn more about the history and theoretical underpinnings in the "Tidy Data" paper published in the *Journal of Statistical Software*.

Now that you're writing a substantial amount of R code, it's time to learn more about organizing your code into files and directories. In the next chapter, you'll learn all about the advantages of scripts and projects and some of the many tools that they provide to make your life easier.

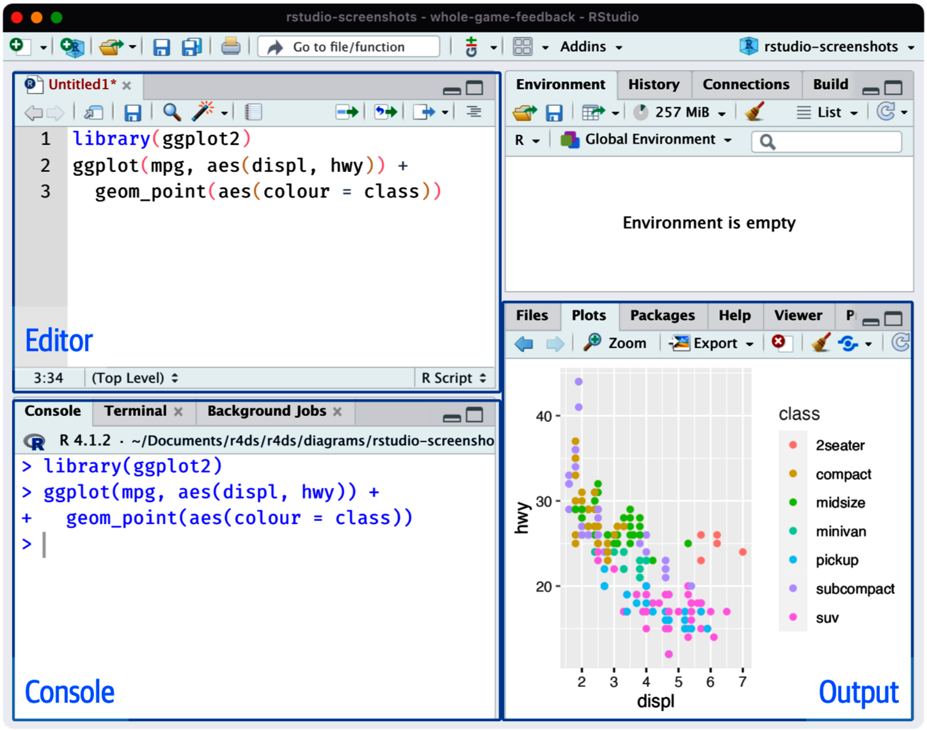
CHAPTER 6

Workflow: Scripts and Projects

This chapter will introduce you to two essential tools for organizing your code: scripts and projects.

Scripts

So far, you have used 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 longer dplyr pipelines. To give yourself more room to work, use the script editor. Open it by clicking the File menu, selecting New File, and then selecting R script, or using the keyboard shortcut Cmd/Ctrl+Shift+N. Now you'll see four panes, as in Figure 6-1. The script editor is a great place to experiment with your code. When you want to change something, you don't have to retype the whole thing; you can just edit the script and rerun it. And once you have written code that works and does what you want, you can save it as a script file to easily return to later.



```
library(ggplot2)
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class))
```

Environment is empty

```
R 4.1.2 · ~/Documents/r4ds/r4ds/diagrams/rstudio-screensho
> library(ggplot2)
> ggplot(mpg, aes(displ, hwy)) +
+   geom_point(aes(colour = class))
> |
```

Figure 6-1. Opening the script editor adds a new pane at the top left of the IDE.

Running Code

The script editor is an excellent place for building 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:

```
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))
```

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 following statement (beginning with `not_cancelled |>`). That makes it easy to step through your complete script by repeatedly pressing Cmd/Ctrl+Enter.

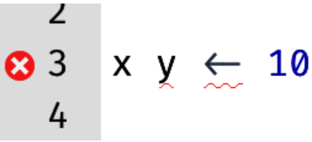
Instead of running your code expression by expression, you can execute the complete script in one step with Cmd/Ctrl+Shift+S. Doing this regularly is a great way to ensure that you've captured all the important parts of your code in the script.

We recommend you always start your script with the packages you need. That way, if you share your code with others, they can easily see which packages they need to install. Note, however, that you should never include `install.packages()` in a script you share. It's inconsiderate to hand off a script that will install something on their computer if they're not being careful!

When working through future chapters, we highly recommend starting in the script 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.

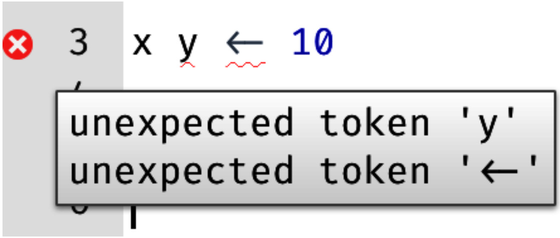
RStudio Diagnostics

In the script editor, RStudio will highlight syntax errors with a red squiggly line and a cross in the sidebar:



```
x y ← 10
```

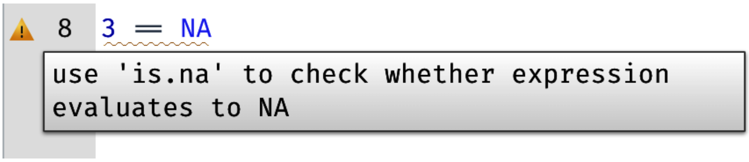
Hover over the cross to see what the problem is:



```
x y ← 10
```

```
unexpected token 'y'
unexpected token '←'
```

RStudio will also let you know about potential problems:



```
3 == NA
```

```
use 'is.na' to check whether expression
evaluates to NA
```

Saving and Naming

RStudio automatically saves the contents of the script editor when you quit and automatically reloads it when you re-open. Nevertheless, it's a good idea to avoid Untitled1, Untitled2, Untitled3, and so on, and instead save your scripts with informative names.

It might be tempting to name your files `code.R` or `myscript.R`, but you should think a bit harder before choosing a name for your file. Three important principles for file naming are as follows:

1. Filenames should be *machine* readable: avoid spaces, symbols, and special characters. Don't rely on case sensitivity to distinguish files.
2. Filenames should be *human* readable: use filenames to describe what's in the file.
3. Filenames should play well with default ordering: start filenames with numbers so that alphabetical sorting puts them in the order they get used.

For example, suppose you have the following files in a project folder:

```
alternative model.R
code for exploratory analysis.r
finalreport.qmd
FinalReport.qmd
fig 1.png
Figure_02.png
model_first_try.R
run-first.r
temp.txt
```

There are a variety of problems here: it's hard to find which file to run first, filenames contain spaces, there are two files with the same name but different capitalization (`finalreport` versus `FinalReport`[1]), and some names don't describe their contents (`run-first` and `temp`).

Here's a better way of naming and organizing the same set of files:

```
01-load-data.R
02-exploratory-analysis.R
03-model-approach-1.R
04-model-approach-2.R
fig-01.png
fig-02.png
report-2022-03-20.qmd
report-2022-04-02.qmd
report-draft-notes.txt
```

[1] Not to mention that you're tempting fate by using "final" in the name. The comic Piled Higher and Deeper has a fun strip on this.

Numbering the key scripts makes it obvious in which order to run them, and a consistent naming scheme makes it easier to see what varies. Additionally, the figures are labeled similarly, the reports are distinguished by dates included in the filenames, and `temp` is renamed to `report-draft-notes` to better describe its contents. If you have a lot of files in a directory, taking organization one step further and placing different types of files (scripts, figures, etc.) in different directories is recommended.

Projects

One day, you will need to quit R, go do something else, and return to your analysis later. One day, you will be working on multiple analyses simultaneously and 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:

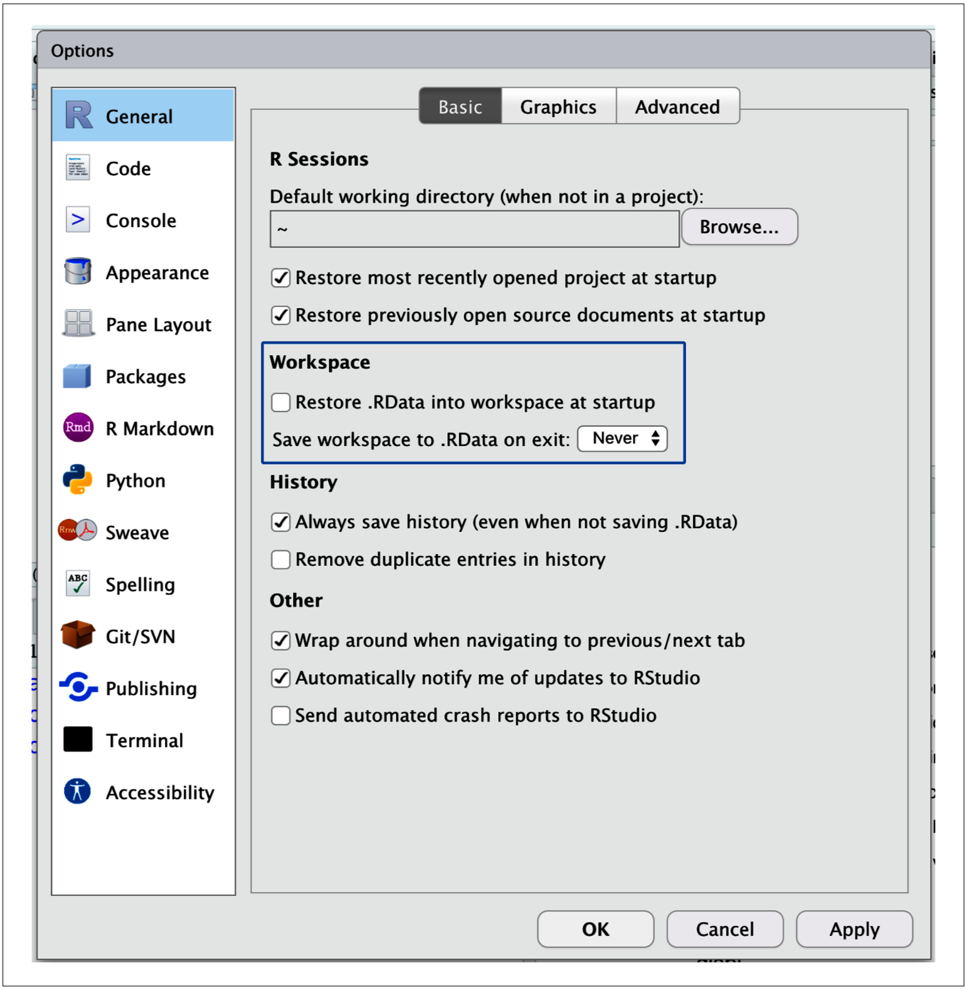
- What is the source of truth? What will you save as your lasting record of what happened?
- Where does your analysis live?

What Is the Source of Truth?

As a beginner, it's OK to rely on your current environment to contain all the objects you have created throughout your analysis. However, to make it easier to work on larger projects or collaborate with others, your source of truth should be the R scripts. With your R scripts (and your data files), you can re-create the environment. With only your environment, it's much harder to re-create your R scripts: either you'll have to retype a lot of code from memory (inevitably making mistakes along the way) or you'll have to carefully mine your R history.

To help keep your R scripts as the source of truth for your analysis, we highly recommend that you instruct RStudio not to preserve your workspace between sessions. You can do this either by running `usethis::use_blank_slate()`[2] or by mimicking the options shown in Figure 6-2. This will cause you some short-term pain, because now when you restart RStudio, it will no longer remember the code that you ran last time nor will the objects you created or the datasets you read be available to use. But this short-term pain saves you long-term agony because it forces you to capture all important procedures in your code. There's nothing worse than discovering three months after the fact that you've stored only the results of an important calculation in your environment, not the calculation itself in your code.

[2] If you don't have this installed, you can install it with `install.packages("usethis")`.



R Sessions

Default working directory (when not in a project): ~

☑ Restore most recently opened project at startup

☑ Restore previously open source documents at startup

Workspace

☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

History

☑ Always save history (even when not saving .RData)

☐ Remove duplicate entries in history

Other

☑ Wrap around when navigating to previous/next tab

☑ Automatically notify me of updates to RStudio

☐ Send automated crash reports to RStudio

Figure 6-2. Copy these selections in your RStudio options to always start your RStudio session with a clean slate.

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:

1. Press Cmd/Ctrl+Shift+0/F10 to restart R.
2. Press Cmd/Ctrl+Shift+S to rerun the current script.

We collectively use this pattern hundreds of times a week.

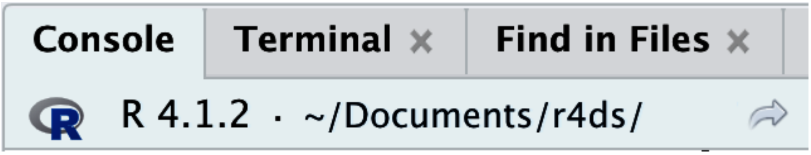
Alternatively, if you don't use keyboard shortcuts, you can select Session > Restart R and then highlight and rerun your current script.

RStudio Server

If you're using RStudio Server, your R session is never restarted by default. When you close your RStudio Server tab, it might feel like you're closing R, but the server actually keeps it running in the background. The next time you return, you'll be in exactly the same place you left. This makes it even more important to regularly restart R so that you're starting with a clean slate.

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:



R 4.1.2 · ~/Documents/r4ds/

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

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

In this R session, the current working directory (think of it as "home") is in Hadley's *Documents* folder, in a subfolder called *r4ds*. This code will return a different result when you run it, because your computer has a different directory structure than Hadley's!

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

You can set the working directory from within R, but *we do not recommend it*:

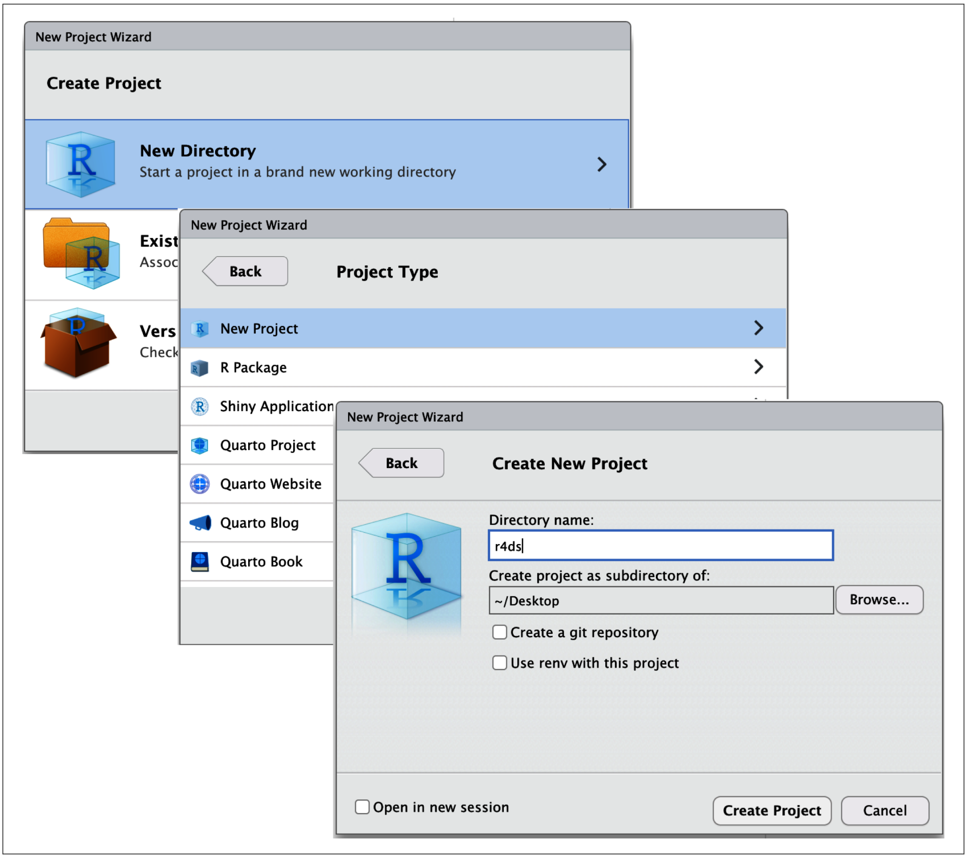
```
setwd("/path/to/my/CoolProject")
```

There's a better way—a way that also puts you on the path to managing your R work like an expert. That way is the *RStudio project*.

RStudio Projects

Keeping all the files associated with a given project (input data, R scripts, analytical results, and figures) in one directory 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. Select File > New Project, and then follow the steps shown in Figure 6-3.



Directory name: r4ds

Create project as subdirectory of: ~/Desktop

Browse...

☐ Create a git repository

☐ Use renv with this project

☐ Open in new session

Create Project

Cancel

Figure 6-3. To create new project: (top) first click New Directory, then (middle) click New Project, then (bottom) fill in the directory (project) name, choose a good subdirectory for its home, and click Create Project.

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" of your project is the current working directory:

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

Now enter the following commands in the script editor and save the file, calling it `diamonds.R`. Then, create a new folder called `data`. You can do this by clicking the New Folder button in the Files pane in RStudio. Finally, run the complete script, which will save a PNG 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(x = carat, y = price)) +
  geom_hex()
ggsave("diamonds.png")

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

Quit RStudio. Inspect the folder associated with your project—notice the `.Rproj` file. Double-click that file to re-open 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 our instructions, 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.png`, and you will find the PNG (no surprise) but *also the script that created it* (`diamonds.R`). This is a 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!

Relative and Absolute Paths

Once you're inside a project, you should only ever use relative paths, not absolute paths. What's the difference? A relative path is relative to the working directory, i.e., the project's home. When Hadley wrote `data/diamonds.csv` earlier, it was a shortcut for `/Users/hadley/Documents/r4ds/data/diamonds.csv`. But importantly, if Mine ran this code on her computer, it would point to `/Users/Mine/Documents/r4ds/data/diamonds.csv`. This is why relative paths are important: they'll work regardless of where the R project folder ends up.

Absolute paths point to the same place regardless of your working directory. They look a little different depending on your operating system. On Windows they start with a drive letter (e.g., `C:`) or two backslashes (e.g., `\\servername`) and on 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.

There's another important difference between operating systems: how you separate the components of the path. Mac and Linux uses slashes (e.g., `data/diamonds.csv`), and Windows uses backslashes (e.g., `data\diamonds.csv`). 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 frustrating, so we recommend always using the Linux/Mac style with forward slashes.

Exercises

1. Go to the RStudio Tips Twitter account and find one tip that looks interesting. Practice using it!
2. What other common mistakes will RStudio diagnostics report? Read this article on code diagnostics to find out.

Summary

In this chapter, you learned how to organize your R code in scripts (files) and projects (directories). Much like code style, this may feel like busywork at first. But as you accumulate more code across multiple projects, you'll learn to appreciate how a little up-front organization can save you a bunch of time later.

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

- Create one RStudio project for each data analysis project.
- Save your scripts (with informative names) in the project, edit them, and run them in bits or as a whole. Restart R frequently to make sure you've captured everything in your scripts.
- Only ever use relative paths, not absolute paths.

Then everything you need is in one place and cleanly separated from all the other projects you are working on.

So far, we've worked with datasets bundled in R packages. This makes it easier to get some practice on preprepared data, but obviously your data won't be available in this way. So in the next chapter, you're going to learn how load data from disk into your R session using the readr package.

CHAPTER 7
Data Import

Introduction

Working with data provided by R packages is a great way to learn data science tools, but you want to apply what you've learned to your own data at some point. In this chapter, you'll learn the basics of reading data files into R.

Specifically, this chapter will focus on reading plain-text rectangular files. We'll start with practical advice for handling features such as column names, types, and missing data. You will then learn about reading data from multiple files at once and writing data from R to a file. Finally, you'll learn how to handcraft data frames in R.

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)
```

Reading Data from a File

To begin, we'll focus on the most common rectangular data file type: CSV, which is short for "comma-separated values." Here is what a simple CSV file looks like. The first row, commonly called the *header row*, gives the column names, and the following six rows provide the data. The columns are separated, aka *delimited*, by commas.

```
Student ID,Full Name,favourite.food,mealPlan,AGE
1,Sunil Huffmann,Strawberry yoghurt,Lunch only,4
2,Barclay Lynn,French fries,Lunch only,5
3,Jayendra Lyne,N/A,Breakfast and lunch,7
4,Leon Rossini,Anchovies,Lunch only,
5,Chidiegwu Dunkel,Pizza,Breakfast and lunch,five
6,Güvenç Attila,Ice cream,Lunch only,6
```

Table 7-1 represents of the same data as a table.

Table 7-1. Data from the students.csv file as a table

Student ID	Full Name	favourite.food	mealPlan	AGE
1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4
2	Barclay Lynn	French fries	Lunch only	5
3	Jayendra Lyne	N/A	Breakfast and lunch	7
4	Leon Rossini	Anchovies	Lunch only	NA
5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five
6	Güvenç Attila	Ice cream	Lunch only	6

We can read this file into R using `read_csv()`. The first argument is the most important: the path to the file. You can think about the path as the address of the file: the file is called `students.csv`, and it lives in the `data` folder.

```
students <- read_csv("data/students.csv")
#> Rows: 6 Columns: 5
#> ── Column specification ────────────────────────────────────────────────────
#> Delimiter: ","
#> chr (4): Full Name, favourite.food, mealPlan, AGE
#> dbl (1): Student ID
#>
#> ℹ Use `spec()` to retrieve the full column specification for this data.
#> ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The previous code will work if you have the `students.csv` file in a `data` folder in your project. You can download the `students.csv` file or you can read it directly from that URL with this:

```
students <- read_csv("https://pos.it/r4ds-students-csv")
```

When you run `read_csv()`, it prints out a message telling you the number of rows and columns of data, the delimiter that was used, and the column specifications (names of columns organized by the type of data the column contains). It also prints out some information about retrieving the full column specification and how to quiet this message. This message is an integral part of readr, and we'll return to it in "Controlling Column Types" on page 104.

Practical Advice

Once you read data in, the first step usually involves transforming it in some way to make it easier to work with in the rest of your analysis. Let's take another look at the students data with that in mind:

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE  
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4    
#> 2            2 Barclay Lynn     French fries       Lunch only          5    
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch 7    
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA> 
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five 
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

In the favourite.food column, there are a bunch of food items, and then the character string N/A, which should have been a real NA that R will recognize as "not available." This is something we can address using the na argument. By default read_csv() recognizes only empty strings ("") in this dataset as NAs; we want it to also recognize the character string "N/A":

```
students <- read_csv("data/students.csv", na = c("N/A", ""))

students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE  
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4    
#> 2            2 Barclay Lynn     French fries       Lunch only          5    
#> 3            3 Jayendra Lyne    <NA>               Breakfast and lunch 7    
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NA> 
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch five 
#> 6            6 Güvenç Attila    Ice cream          Lunch only          6
```

You might also notice that the Student ID and Full Name columns are surrounded by backticks. That's because they contain spaces, breaking R's usual rules for variable names; they're *nonsyntactic* names. To refer to these variables, you need to surround them with backticks, `:

```
students |> 
  rename(
    student_id = `Student ID`,
    full_name = `Full Name`
  )
#> # A tibble: 6 × 5
#>   student_id full_name        favourite.food     mealPlan            AGE  
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4    
#> 2          2 Barclay Lynn     French fries       Lunch only          5    
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7    
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA> 
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five 
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

An alternative approach is to use `janitor::clean_names()` to use some heuristics to turn them all into snake case at once:[1]

```
students |> janitor::clean_names()
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age  
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4    
#> 2          2 Barclay Lynn     French fries       Lunch only          5    
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7    
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA> 
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five 
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Another common task after reading in data is to consider variable types. For example, `meal_plan` is a categorical variable with a known set of possible values, which in R should be represented as a factor:

```
students |>
  janitor::clean_names() |>
  mutate(meal_plan = factor(meal_plan))
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age  
#>        <dbl> <chr>            <chr>              <fct>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4    
#> 2          2 Barclay Lynn     French fries       Lunch only          5    
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7    
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA> 
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five 
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Note that the values in the `meal_plan` variable have stayed the same, but the type of variable denoted underneath the variable name has changed from character (`<chr>`) to factor (`<fct>`). You'll learn more about factors in Chapter 16.

Before you analyze these data, you'll probably want to fix the `age` and `id` columns. Currently, `age` is a character variable because one of the observations is typed out as `five` instead of a numeric `5`. We discuss the details of fixing this issue in Chapter 20.

```
students <- students |>
  janitor::clean_names() |>
  mutate(
    meal_plan = factor(meal_plan),
    age = parse_number(if_else(age == "five", "5", age))
  )

students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
```

[1] The janitor package is not part of the tidyverse, but it offers handy functions for data cleaning and works well within data pipelines that use `|>`.

```
#> 2         2 Barclay Lynn     French fries      Lunch only              5
#> 3         3 Jayendra Lyne    <NA>              Breakfast and lunch     7
#> 4         4 Leon Rossini     Anchovies         Lunch only             NA
#> 5         5 Chidiegwu Dunkel Pizza             Breakfast and lunch     5
#> 6         6 Güvenç Attila    Ice cream         Lunch only              6
```

A new function here is `if_else()`, which 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`. Here we're saying if `age` is the character string `"five"`, make it `"5"`, and if not, leave it as `age`. You will learn more about `if_else()` and logical vectors in Chapter 12.

Other Arguments

There are a couple of other important arguments that we need to mention, and they'll be easier to demonstrate if we first show you a handy trick: `read_csv()` can read text strings that you've created and formatted like a CSV file:

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

Usually, `read_csv()` uses the first line of the data for the column names, which is a common convention. But it's not uncommon for a few lines of metadata to be included 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, for example, `#`:

```
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
#>   <dbl> <dbl> <dbl>
#> 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
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
```

In other cases, 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
  4,5,6",
  col_names = FALSE
)
#> # A tibble: 2 × 3
#>      X1    X2    X3
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

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

```
read_csv(
  "1,2,3
  4,5,6",
  col_names = c("x", "y", "z")
)
#> # A tibble: 2 × 3
#>       x     y     z
#>   <dbl> <dbl> <dbl>
#> 1     1     2     3
#> 2     4     5     6
```

These arguments are all you need to know to read the majority of CSV files that you'll encounter in practice. (For the rest, you'll need to carefully inspect your `.csv` file and read the documentation for `read_csv()`'s many other arguments.)

Other File Types

Once you've mastered `read_csv()`, using readr's other functions is straightforward; it's just a matter of knowing which function to reach for:

`read_csv2()`
: Reads semicolon-separated files. These use `;` instead of `,` to separate fields and are common in countries that use `,` as the decimal marker.

`read_tsv()`
: Reads tab-delimited files.

`read_delim()`
: Reads in files with any delimiter, attempting to automatically guess the delimiter if you don't specify it.

`read_fwf()`
: Reads fixed-width files. You can specify fields by their widths with `fwf_widths()` or by their positions with `fwf_positions()`.

`read_table()`
: Reads a common variation of fixed-width files where columns are separated by whitespace.

`read_log()`
: Reads Apache-style log files.

Exercises

1. What function would you use to read a file where fields were 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 default, `read_csv()` assumes that the quoting character will be `"`. To read the following text into a data frame, what argument to `read_csv()` do you need to specify?

   ```
   "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")
   ```

6. 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`))
   )
   ```

Controlling Column Types

A CSV file doesn't contain any information about the type of each variable (i.e., whether it's a logical, number, string, etc.), so readr will try to guess the type. This section describes how the guessing process works, how to resolve some common problems that cause it to fail, and, if needed, how to supply the column types yourself. Finally, we'll mention a few general strategies that are useful if readr is failing catastrophically and you need to get more insight into the structure of your file.

Guessing Types

readr uses a heuristic to figure out the column types. For each column, it pulls the values of 1,000[2] rows spaced evenly from the first row to the last, ignoring missing values. It then works through the following questions:

- Does it contain only `F`, `T`, `FALSE`, or `TRUE` (ignoring case)? If so, it's a logical.
- Does it contain only numbers (e.g., `1`, `-4.5`, `5e6`, `Inf`)? If so, it's a number.
- Does it match the ISO8601 standard? If so, it's a date or date-time. (We'll return to date-times in more detail in "Creating Date/Times" on page 298.)
- Otherwise, it must be a string.

You can see that behavior in action in this simple example:

```
read_csv("
  logical,numeric,date,string
  TRUE,1,2021-01-15,abc
  false,4.5,2021-02-15,def
  T,Inf,2021-02-16,ghi
")
#> # A tibble: 3 × 4
#>   logical numeric date       string
#>   <lgl>     <dbl> <date>     <chr>
#> 1 TRUE        1   2021-01-15 abc
#> 2 FALSE       4.5 2021-02-15 def
#> 3 TRUE      Inf   2021-02-16 ghi
```

This heuristic works well if you have a clean dataset, but in real life, you'll encounter a selection of weird and beautiful failures.

Missing Values, Column Types, and Problems

The most common way column detection fails is that a column contains unexpected values, and you get a character column instead of a more specific type. One of the

[2] You can override the default of 1,000 with the `guess_max` argument.

most common causes for this is a missing value, recorded using something other than the NA that readr expects.

Take this simple one-column CSV file as an example:

```
simple_csv <- "
  x
  10
  .
  20
  30"
```

If we read it without any additional arguments, x becomes a character column:

```
read_csv(simple_csv)
#> # A tibble: 4 × 1
#>   x
#>   <chr>
#> 1 10
#> 2 .
#> 3 20
#> 4 30
```

In this small case, you can easily see the missing value `.`. But what happens if you have thousands of rows with only a few missing values represented by `.`s sprinkled among them? One approach is to tell readr that x is a numeric column and then see where it fails. You can do that with the `col_types` argument, which takes a named list where the names match the column names in the CSV file:

```
df <- read_csv(
  simple_csv,
  col_types = list(x = col_double())
)
#> Warning: One or more parsing issues, call `problems()` on your data frame for
#> details, e.g.:
#>   dat <- vroom(...)
#>   problems(dat)
```

Now `read_csv()` reports that there was a problem and tells us we can find out more with `problems()`:

```
problems(df)
#> # A tibble: 1 × 5
#>     row   col expected actual file
#>   <int> <int> <chr>    <chr>  <chr>
#> 1     3     1 a double .      /private/tmp/RtmpAYlSop/file392d445cf269
```

This tells us that there was a problem in row 3, column 1 where readr expected a double but got a `.`. That suggests this dataset uses `.` for missing values. So then we set `na = "."`, and the automatic guessing succeeds, giving us the numeric column that we want:

```
read_csv(simple_csv, na = ".")
#> # A tibble: 4 × 1
#>       x
#>   <dbl>
#> 1     10
#> 2     NA
#> 3     20
#> 4     30
```

Column Types

readr provides a total of nine column types for you to use:

- `col_logical()` and `col_double()` read logicals and real numbers. They're relatively rarely needed (except as shown previously), since readr will usually guess them for you.
- `col_integer()` reads integers. We seldom distinguish integers and doubles in this book because they're functionally equivalent, but reading integers explicitly can occasionally be useful because they occupy half the memory of doubles.
- `col_character()` reads strings. This can be useful to specify explicitly when you have a column that is a numeric identifier, i.e., long series of digits that identifies an object but doesn't make sense to apply mathematical operations to. Examples include phone numbers, Social Security numbers, credit card numbers, and so on.
- `col_factor()`, `col_date()`, and `col_datetime()` create factors, dates, and date-times, respectively; you'll learn more about those when we get to those data types in Chapter 16 and Chapter 17.
- `col_number()` is a permissive numeric parser that will ignore non-numeric components and is particularly useful for currencies. You'll learn more about it in Chapter 13.
- `col_skip()` skips a column so it's not included in the result, which can be useful for speeding up reading the data if you have a large CSV file and you want to use only some of the columns.

It's also possible to override the default column by switching from `list()` to `cols()` and specifying `.default`:

```
another_csv <- "
x,y,z
1,2,3"

read_csv(
  another_csv,
  col_types = cols(.default = col_character())
)
#> # A tibble: 1 × 3
#>   x     y     z
#>   <chr> <chr> <chr>
#> 1 1     2     3
```

Another useful helper is `cols_only()`, which will read in only the columns you specify:

```
read_csv(
  another_csv,
  col_types = cols_only(x = col_character())
)
#> # A tibble: 1 × 1
#>   x
#>   <chr>
#> 1 1
```

Reading Data from Multiple Files

Sometimes your data is split across multiple files instead of being contained in a single file. For example, you might have sales data for multiple months, with each month's data in a separate file: `01-sales.csv` for January, `02-sales.csv` for February, and `03-sales.csv` for March. With `read_csv()` you can read these data in at once and stack them on top of each other in a single data frame.

```
sales_files <- c("data/01-sales.csv", "data/02-sales.csv", "data/03-sales.csv")
read_csv(sales_files, id = "file")
#> # A tibble: 19 × 6
#>   file              month    year brand  item     n
#>   <chr>             <chr>   <dbl> <dbl> <dbl> <dbl>
#> 1 data/01-sales.csv January  2019     1  1234     3
#> 2 data/01-sales.csv January  2019     1  8721     9
#> 3 data/01-sales.csv January  2019     1  1822     2
#> 4 data/01-sales.csv January  2019     2  3333     1
#> 5 data/01-sales.csv January  2019     2  2156     9
#> 6 data/01-sales.csv January  2019     2  3987     6
#> # … with 13 more rows
```

Once again, the previous code will work if you have the CSV files in a `data` folder in your project. You can download these files from *https://oreil.ly/jVd8o*, *https://oreil.ly/RYsgM*, and *https://oreil.ly/4uZOm* or you can read them directly with:

```
sales_files <- c(
  "https://pos.it/r4ds-01-sales",
  "https://pos.it/r4ds-02-sales",
  "https://pos.it/r4ds-03-sales"
)
read_csv(sales_files, id = "file")
```

The `id` argument adds a new column called `file` to the resulting data frame that identifies the file the data come from. This is especially helpful in circumstances where the files you're reading in do not have an identifying column that can help you trace the observations back to their original sources.

If you have many files you want to read in, it can get cumbersome to write out their names as a list. Instead, you can use the base `list.files()` function to find the files for you by matching a pattern in the filenames. You'll learn more about these patterns in Chapter 15.

```
sales_files <- list.files("data", pattern = "sales\\.csv$", full.names = TRUE)
sales_files
#> [1] "data/01-sales.csv" "data/02-sales.csv" "data/03-sales.csv"
```

Writing to a File

readr also comes with two useful functions for writing data to disk: `write_csv()` and `write_tsv()`. The most important arguments to these functions are `x` (the data frame to save) and `file` (the location to save it). You can also specify how missing values are written with `na`, as well as whether you want to `append` to an existing file.

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

Now let's read that CSV file back in. Note that the variable type information that you just set up is lost when you save to CSV because you're starting over with reading from a plain-text file again:

```
students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <fct>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
write_csv(students, "students-2.csv")
read_csv("students-2.csv")
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

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 main 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. This means that when you reload the object, you are loading the *exact same* R object that you stored.

  ```
  write_rds(students, "students.rds")
  read_rds("students.rds")
  #> # A tibble: 6 × 5
  #>   student_id full_name        favourite_food     meal_plan             age
  #>        <dbl> <chr>            <chr>              <fct>               <dbl>
  #> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
  #> 2          2 Barclay Lynn     French fries       Lunch only              5
  #> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
  ```

```
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

- The arrow package allows you to read and write parquet files, a fast binary file format that can be shared across programming languages. We'll return to arrow in more depth in Chapter 22.

  ```
  library(arrow)
  write_parquet(students, "students.parquet")
  read_parquet("students.parquet")
  #> # A tibble: 6 × 5
  #>   student_id full_name        favourite_food     meal_plan             age
  #>        <dbl> <chr>            <chr>              <fct>               <dbl>
  #> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
  #> 2          2 Barclay Lynn     French fries       Lunch only              5
  #> 3          3 Jayendra Lyne    NA                 Breakfast and lunch     7
  #> 4          4 Leon Rossini     Anchovies          Lunch only             NA
  #> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
  #> 6          6 Güvenç Attila    Ice cream          Lunch only              6
  ```

Parquet tends to be much faster than RDS and is usable outside of R but does require the arrow package.

Data Entry

Sometimes you'll need to assemble a tibble "by hand" doing a little data entry in your R script. There are two useful functions to help you do this, which differ in whether you lay out the tibble by columns or by rows. `tibble()` works by column:

```
tibble(
  x = c(1, 2, 5),
  y = c("h", "m", "g"),
  z = c(0.08, 0.83, 0.60)
)
#> # A tibble: 3 × 3
#>       x y         z
#>   <dbl> <chr> <dbl>
#> 1     1 h      0.08
#> 2     2 m      0.83
#> 3     5 g      0.6
```

Laying out the data by column can make it hard to see how the rows are related, so an alternative is `tribble()`, short for *tr*ansposed t*ibble*, which lets you lay out your data row by row. `tribble()` is customized for data entry in code: column headings start with ~ and entries are separated by commas. This makes it possible to lay out small amounts of data in an easy-to-read form:

```
tribble(
  ~x, ~y, ~z,
  1, "h", 0.08,
  2, "m", 0.83,
  5, "g", 0.60
)
#> # A tibble: 3 × 3
```

```
#>   x         y     z
#>   <chr> <dbl> <dbl>
#> 1        1 h  0.08
#> 2        2 m  0.83
#> 3        5 g  0.6
```

Summary

In this chapter, you learned how to load CSV files with `read_csv()` and to do your own data entry with `tibble()` and `tribble()`. You've learned how CSV files work, some of the problems you might encounter, and how to overcome them. We'll come to data import a few times in this book: Chapter 20 will show you how to load data from Excel and Google Sheets, Chapter 21 from databases, Chapter 22 from parquet files, Chapter 23 from JSON, and Chapter 24 from websites.

We're just about at the end of this section of the book, but there's one important last topic to cover: how to get help. So in the next chapter, you'll learn some good places to look for help, how to create a reprex to maximize your chances of getting good help, and some general advice on keeping up with the world of R.

CHAPTER 8

Workflow: Getting Help

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

Google Is Your Friend

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. Additionally, adding package names like "tidyverse" or "ggplot2" will help narrow down the results to code that will feel more familiar to you as well, e.g., "how to make a boxplot in R" versus "how to make a boxplot in R with ggplot2." 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 rerun the code; you're more likely to find help for English error messages.)

If Google doesn't help, try Stack Overflow. Start by spending a little time searching for an existing answer, including `[R]`, to restrict your search to questions and answers that use R.

Making a reprex

If your googling doesn't find anything useful, it's a really good idea to prepare a *reprex*, short for minimal *repro*ducible *ex*ample. 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 two parts to creating a reprex:

- First, you need to make your code reproducible. This means you need to capture everything, i.e., include any `library()` calls and create all necessary objects. The easiest way to make sure you've done this is using the reprex package.
- Second, you need to make it minimal. Strip away everything that is not directly related to your problem. This usually involves creating a much smaller and simpler R object than the one you're facing in real life or even using built-in data.

That sounds like a lot of work! And it can be, but it has a great payoff:

- 80% of the time, creating an excellent reprex reveals the source of your problem. It's amazing how often the process of writing up a self-contained and minimal example allows you to answer your own question.
- The other 20% of the time, you will have captured the essence of your problem in a way that is easy for others to play with. This substantially improves your chances of getting help!

When creating a reprex by hand, it's easy to accidentally miss something, meaning your code can't be run on someone else's computer. Avoid this problem by using the reprex package, which is installed as part of the tidyverse. Let's say you copy this code onto your clipboard (or, on RStudio Server or Cloud, select it):

```
y <- 1:4
mean(y)
```

Then call `reprex()`, where the default output is formatted for GitHub:

```
reprex::reprex()
```

A nicely rendered HTML preview will display in RStudio's Viewer (if you're in RStudio) or your default browser otherwise. The reprex is automatically copied to your clipboard (on RStudio Server or Cloud, you will need to copy this yourself):

~~~
``` r
y <- 1:4
mean(y)
#> [1] 2.5
```
~~~

This text is formatted in a special way, called Markdown, which can be pasted to sites like StackOverflow or GitHub, which will automatically render it to look like code. Here's what that Markdown would look like rendered on GitHub:

```
y <- 1:4
mean(y)
#> [1] 2.5
```

Anyone else can copy, paste, and run this immediately.

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; you may have discovered a bug that's been fixed since you installed or last updated the package. For packages in the tidyverse, the easiest way to check is to run `tidyverse_update()`.
- The easiest way to include *data* is to use `dput()` to generate the R code needed to re-create it. For example, to re-create the `mtcars` dataset in R, perform the following steps:
  - Run `dput(mtcars)` in R.
  - Copy the output.
  - In reprex, type `mtcars <-`, and then paste.

  Try to use 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.

Creating reprexes is not trivial, and it will take some practice to learn to create good, truly minimal reprexes. However, learning to ask questions that include the code and investing the time to make it reproducible will continue to pay off as you learn and master R.

Investing in Yourself

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 the tidyverse team is doing on the tidyverse blog. To keep up with the R community more broadly, we recommend reading R Weekly: it's a community effort to aggregate the most interesting news in the R community each week.

Summary

This chapter concludes the "Whole Game" part of the book. You've now seen the most important parts of the data science process: visualization, transformation, tidying, and importing. Now that you've gotten a holistic view of the whole process, we can start to get into the details of small pieces.

The next part of the book, "Visualize," does a deeper dive into the grammar of graphics and creating data visualizations with ggplot2, showcases how to use the tools you've learned so far to conduct exploratory data analysis, and introduces good practices for creating plots for communication.

PART II
Visualize

After reading the first part of the book, you understand (at least superficially) the most important tools for doing data science. Now it's time to start diving into the details. In this part of the book, you'll learn about visualizing data in further depth in Figure II-1.

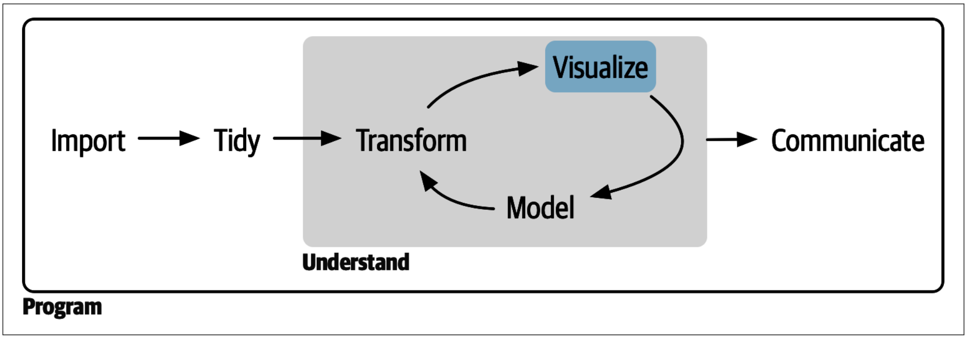


Figure II-1. Data visualization is often the first step in data exploration.

Each chapter addresses one to a few aspects of creating a data visualization:

- In Chapter 9 you will learn about the layered grammar of graphics.
- In Chapter 10, you'll combine visualization with your curiosity and skepticism to ask and answer interesting questions about data.

- Finally, in Chapter 11 you will learn how to take your exploratory graphics, elevate them, and turn them into expository graphics, graphics that help the newcomer to your analysis understand what's going on as quickly and easily as possible.

These three chapters get you started in the world of visualization, but there is much more to learn. The absolute best place to learn more is the ggplot2 book: *ggplot2: Elegant Graphics for Data Analysis* (Springer). It goes into much more depth about the underlying theory and has many more examples of how to combine the individual pieces to solve practical problems. Another great resource is the ggplot2 extensions gallery. This site lists many of the packages that extend ggplot2 with new geoms and scales. It's a great place to start if you're trying to do something that seems hard with ggplot2.

CHAPTER 9

Layers

Introduction

In Chapter 1, you learned much more than just how to make scatterplots, bar charts, and boxplots. You learned a foundation that you can use to make *any* type of plot with ggplot2.

In this chapter, you'll expand on that foundation as you learn about the layered grammar of graphics. We'll start with a deeper dive into aesthetic mappings, geometric objects, and facets. Then, you will learn about statistical transformations ggplot2 makes under the hood when creating a plot. These transformations are used to calculate new values to plot, such as the heights of bars in a bar plot or medians in a box plot. You will also learn about position adjustments, which modify how geoms are displayed in your plots. Finally, we'll briefly introduce coordinate systems.

We will not cover every single function and option for each of these layers, but we will walk you through the most important and commonly used functionality provided by ggplot2 as well as introduce you to packages that extend ggplot2.

Prerequisites

This chapter focuses on ggplot2. To access the datasets, help pages, and functions used in this chapter, load the tidyverse by running this code:

```
library(tidyverse)
```

Aesthetic Mappings

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

Remember that the mpg data frame bundled with the ggplot2 package contains 234 observations on 38 car models.

```
mpg
#> # A tibble: 234 × 11
#>   manufacturer model displ  year   cyl trans      drv     cty   hwy fl   
#>   <chr>        <chr> <dbl> <int> <int> <chr>      <chr> <int> <int> <chr>
#> 1 audi         a4      1.8  1999     4 auto(l5)   f        18    29 p    
#> 2 audi         a4      1.8  1999     4 manual(m5) f        21    29 p    
#> 3 audi         a4      2    2008     4 manual(m6) f        20    31 p    
#> 4 audi         a4      2    2008     4 auto(av)   f        21    30 p    
#> 5 audi         a4      2.8  1999     6 auto(l5)   f        16    26 p    
#> 6 audi         a4      2.8  1999     6 manual(m5) f        18    26 p    
#> # … with 228 more rows, and 1 more variable: class <chr>
```

Among the variables in mpg are:

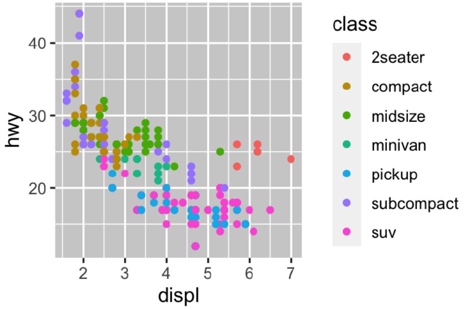
displ
: A car's engine size, in liters. A numerical variable.

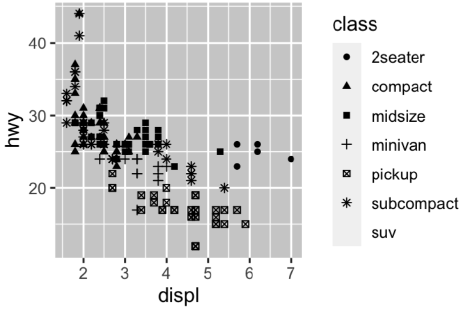
hwy
: A car's fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance. A numerical variable.

class
: Type of car. A categorical variable.

Let's start by visualizing the relationship between displ and hwy for various classes of cars. We can do this with a scatterplot where the numerical variables are mapped to the x and y aesthetics and the categorical variable is mapped to an aesthetic like color or shape.

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

# Right
ggplot(mpg, aes(x = displ, y = hwy, shape = class)) +
  geom_point()
#> Warning: The shape palette can deal with a maximum of 6 discrete values
#> because more than 6 becomes difficult to discriminate; you have 7.
#> Consider specifying shapes manually if you must have them.
#> Warning: Removed 62 rows containing missing values (`geom_point()`).
```





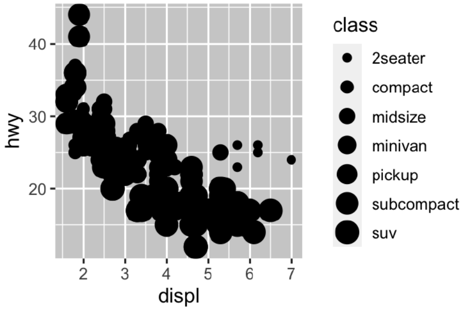
When `class` is mapped to `shape`, we get two warnings:

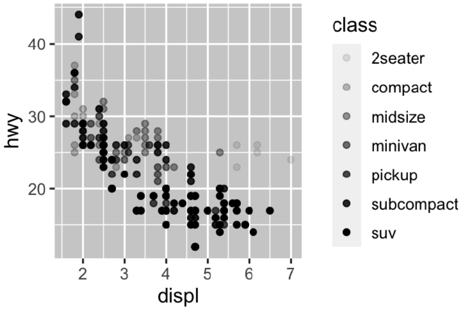
> 1: The shape palette can deal with a maximum of 6 discrete values because more than 6 becomes difficult to discriminate; you have 7. Consider specifying shapes manually if you must have them.
>
> 2: Removed 62 rows containing missing values (`geom_point()`).

Since ggplot2 will use only six shapes at a time, by default, additional groups will go unplotted when you use the shape aesthetic. The second warning is related—there are 62 SUVs in the dataset and they're not plotted.

Similarly, we can map `class` to `size` or `alpha` aesthetics as well, which control the shape and the transparency of the points, respectively.

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

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





Both of these produce warnings as well:

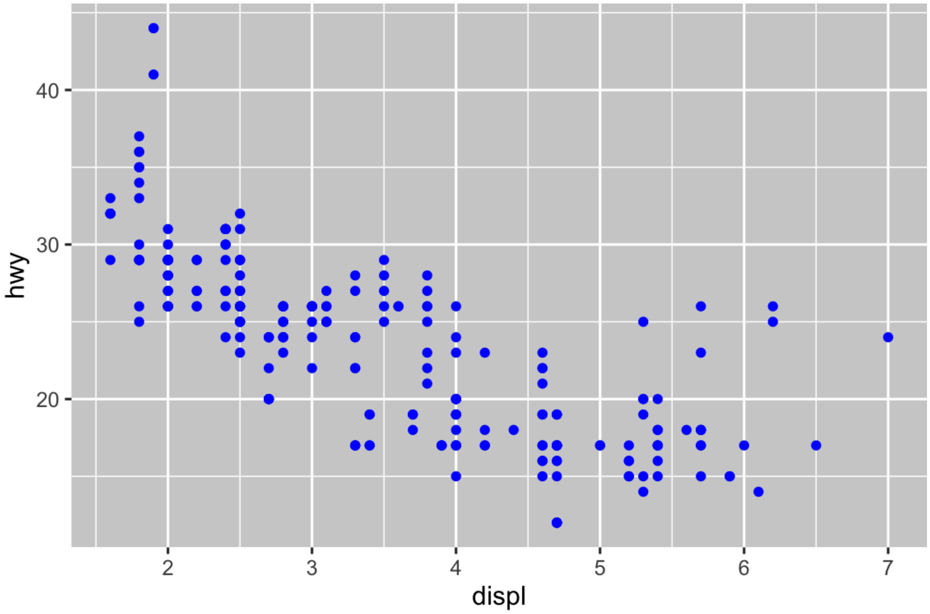
> Using alpha for a discrete variable is not advised.

Mapping an unordered discrete (categorical) variable (`class`) to an ordered aesthetic (`size` or `alpha`) is generally not a good idea because it implies a ranking that does not in fact exist.

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 provides the same information as a legend; it explains the mapping between locations and values.

You can also set the visual properties of your geom manually as an argument of your geom function (*outside* of `aes()`) instead of relying on a variable mapping to determine the appearance. For example, we can make all of the points in our plot blue:

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



Here, the color doesn't convey information about a variable; it changes only the appearance of the plot. You'll need to pick a value that makes sense for that aesthetic:

- The name of a color as a character string, e.g., `color = "blue"`
- The size of a point in mm, e.g., `size = 1`
- The shape of a point as a number, e.g, `shape = 1`, as shown in Figure 9-1

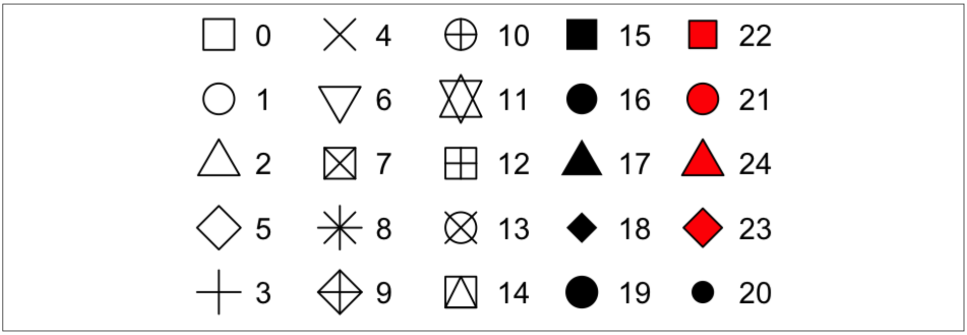


Figure 9-1. R has 25 built-in shapes that are identified by numbers. There are some seeming duplicates: for example, 0, 15, and 22 are all squares. The difference comes from the interaction of the `color` and `fill` aesthetics. The hollow shapes (0–14) have a border determined by `color`; the solid shapes (15–20) are filled with `color`; and the filled shapes (21–24) have a border of `color` and are filled with `fill`. Shapes are arranged to keep similar shapes next to each other.

So far we have discussed aesthetics that we can map or set in a scatterplot, when using a point geom. You can learn more about all possible aesthetic mappings in the aesthetic specifications vignette.

The specific aesthetics you can use for a plot depend on the geom you use to represent the data. In the next section we dive deeper into geoms.

Exercises

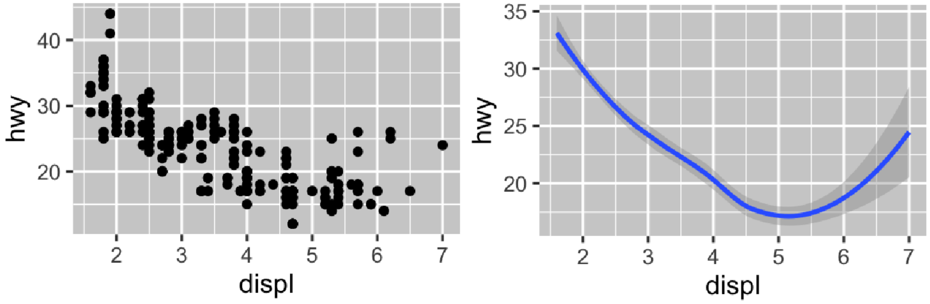
1. Create a scatterplot of `hwy` versus `displ` where the points are pink filled-in triangles.
2. Why did the following code not result in a plot with blue points?

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

3. What does the `stroke` aesthetic do? What shapes does it work with? (Hint: Use `?geom_point`.)
4. What happens if you map an aesthetic to something other than a variable name, like `aes(color = displ < 5)`? Note, you'll also need to specify x and y.

Geometric Objects

How are these two plots similar?



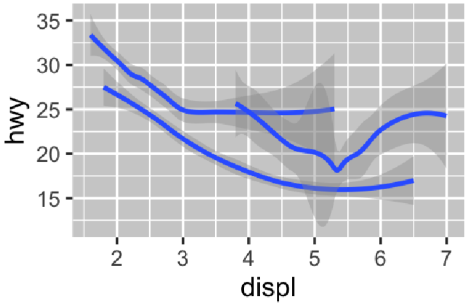
Both plots contain the same x variable and the same y variable, and both describe the same data. But the plots are not identical. Each plot uses a different geometric object, geom, to represent the data. The plot on the left uses the point geom, and the plot on the right uses the smooth geom, a smooth line fitted to the data.

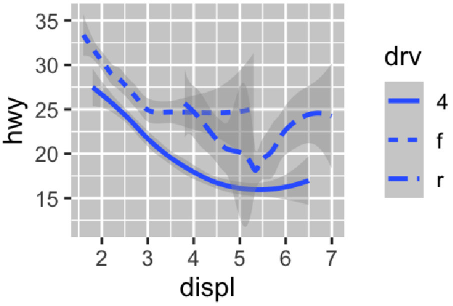
To change the geom in your plot, change the geom function that you add to `ggplot()`. For instance, to make the previous plot, you can use the following code:

```
# Left
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()

# Right
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_smooth()
#> `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```

Every geom function in ggplot2 takes a `mapping` argument, either defined locally in the geom layer or globally in the `ggplot()` layer. 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. If you try, ggplot2 will silently ignore that aesthetic mapping. 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.

```
# Left
ggplot(mpg, aes(x = displ, y = hwy, shape = drv)) +
  geom_smooth()

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

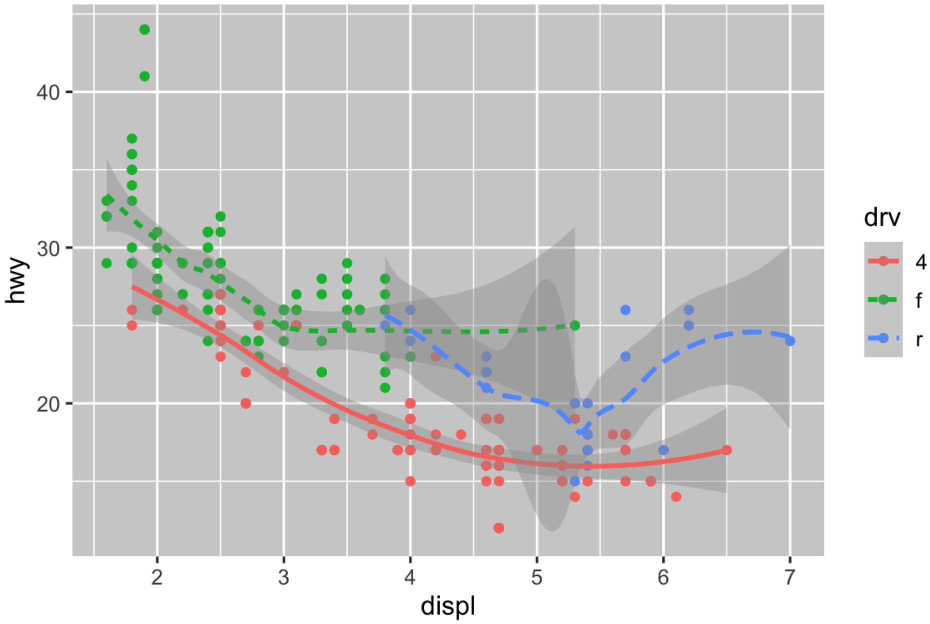




Here, geom_smooth() separates the cars into three lines based on their drv value, which describes a car's drivetrain. One line describes all of the points that have a 4 value, one line describes all of the points that have an f value, and one line describes all of the points that have an r value. Here, 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 clearer by overlaying the lines on top of the raw data and then coloring everything according to drv.

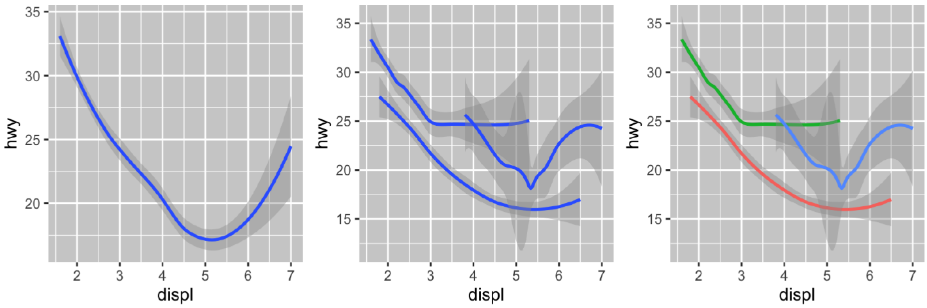
```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  geom_smooth(aes(linetype = drv))
```



Notice that this plot contains two geoms in the same graph.

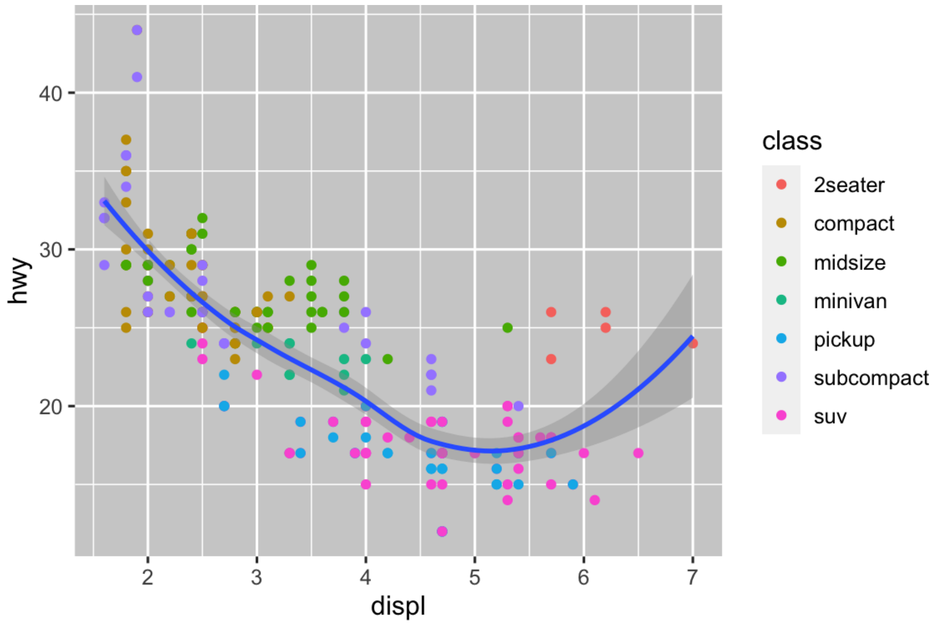
Many geoms, like `geom_smooth()`, use a single geometric object to display multiple rows of data. For these geoms, you can set the `group` aesthetic to a categorical variable to draw multiple objects. ggplot2 will draw a separate object for each unique value of the grouping variable. In practice, ggplot2 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.

```
# Left
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_smooth()

# Middle
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_smooth(aes(group = drv))

# Right
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_smooth(aes(color = drv), show.legend = FALSE)
```



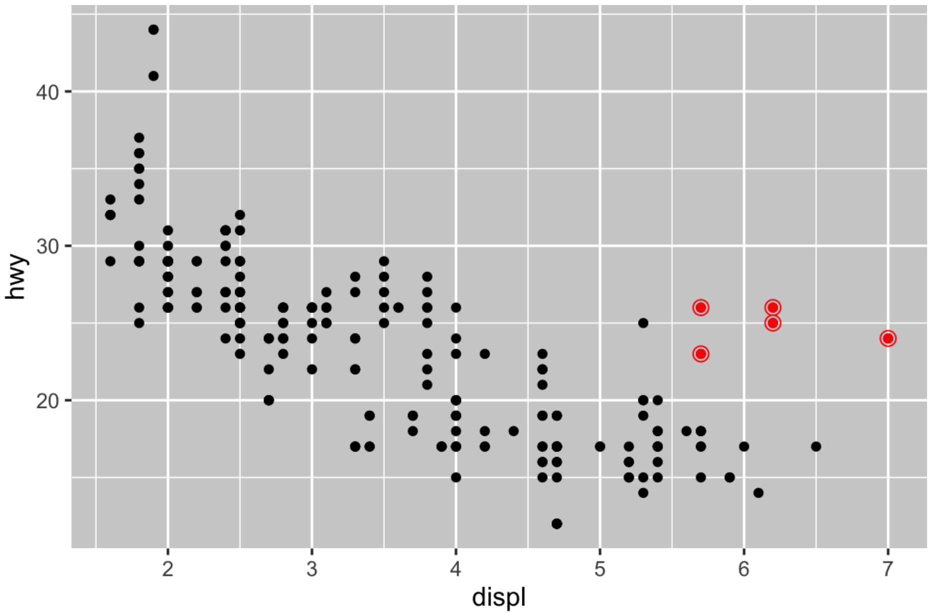
If you place mappings in a geom function, ggplot2 will treat them as local mappings for the layer. It will use these mappings to extend or overwrite the global mappings *for that layer only*. This makes it possible to display different aesthetics in different layers.

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



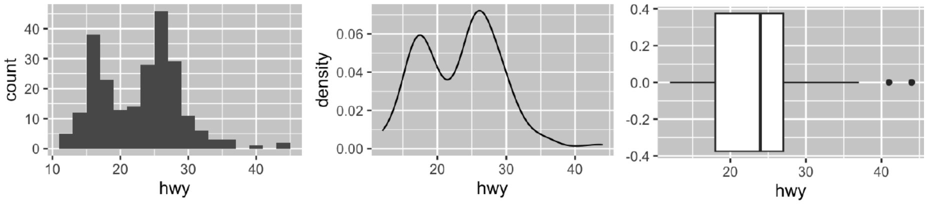
You can use the same idea to specify different `data` for each layer. Here, we use red points as well as open circles to highlight two-seater cars. The local data argument in `geom_point()` overrides the global data argument in `ggplot()` for that layer only.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_point(
    data = mpg |> filter(class == "2seater"),
    color = "red"
  ) +
  geom_point(
    data = mpg |> filter(class == "2seater"),
    shape = "circle open", size = 3, color = "red"
  )
```



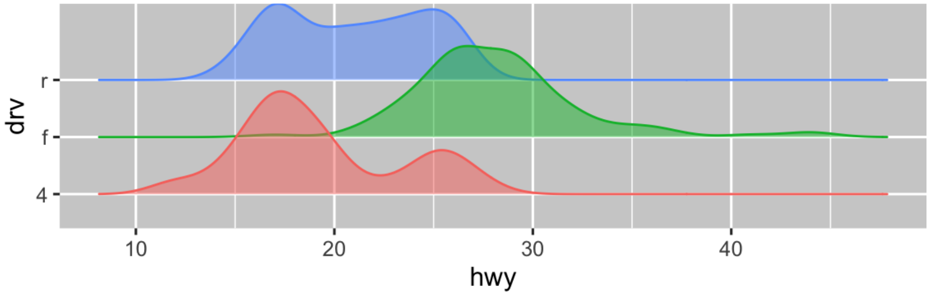
Geoms are the fundamental building blocks of ggplot2. You can completely transform the look of your plot by changing its geom, and different geoms can reveal different features of your data. For example, the following histogram and density plot reveal that the distribution of highway mileage is bimodal and right skewed, while the boxplot reveals two potential outliers:

```
# Left
ggplot(mpg, aes(x = hwy)) +
  geom_histogram(binwidth = 2)

# Middle
ggplot(mpg, aes(x = hwy)) +
  geom_density()

# Right
ggplot(mpg, aes(x = hwy)) +
  geom_boxplot()
```



ggplot2 provides more than 40 geoms, but these geoms don't cover all the possible plots one could make. If you need a different geom, look into extension packages first to see if someone else has already implemented it. For example, the ggridges package is useful for making ridgeline plots, which can be useful for visualizing the density of a numerical variable for different levels of a categorical variable. In the following plot, not only did we use a new geom (`geom_density_ridges()`), but we have also mapped the same variable to multiple aesthetics (`drv` to `y`, `fill`, and `color`) as well as set an aesthetic (`alpha = 0.5`) to make the density curves transparent.

```
library(ggridges)

ggplot(mpg, aes(x = hwy, y = drv, fill = drv, color = drv)) +
  geom_density_ridges(alpha = 0.5, show.legend = FALSE)
#> Picking joint bandwidth of 1.28
```



The best place to get a comprehensive overview of all of the geoms ggplot2 offers, as well as all functions in the package, is the reference page. To learn more about any single geom, use the help (e.g., `?geom_smooth`).

Exercises

1. What geom would you use to draw a line chart? A boxplot? A histogram? An area chart?

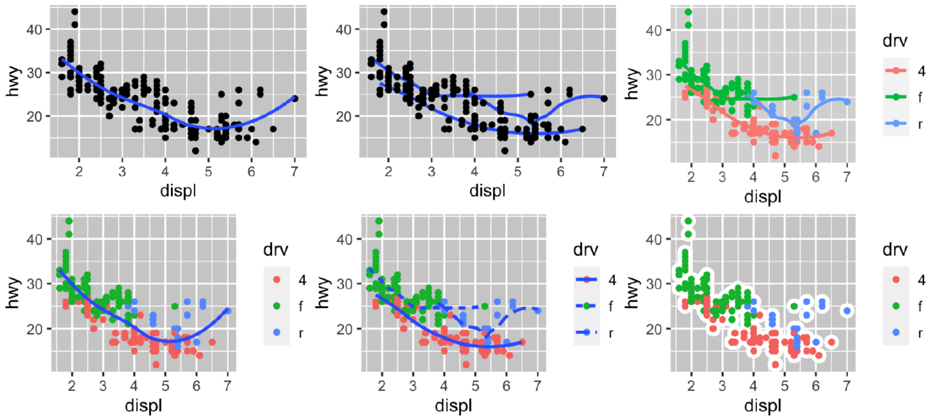
2. Earlier in this chapter we used `show.legend` without explaining it:

   ```
   ggplot(mpg, aes(x = displ, y = hwy)) +
     geom_smooth(aes(color = drv), show.legend = FALSE)
   ```

   What does `show.legend = FALSE` do here? What happens if you remove it? Why do you think we used it earlier?

3. What does the `se` argument to `geom_smooth()` do?

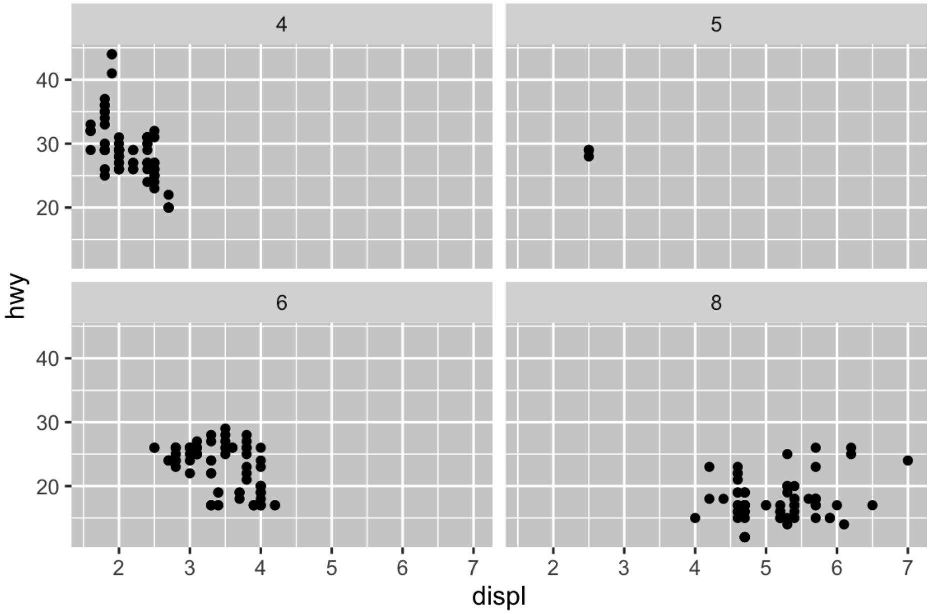
4. Re-create the R code necessary to generate the following graphs. Note that wherever a categorical variable is used in the plot, it's `drv`.



Facets

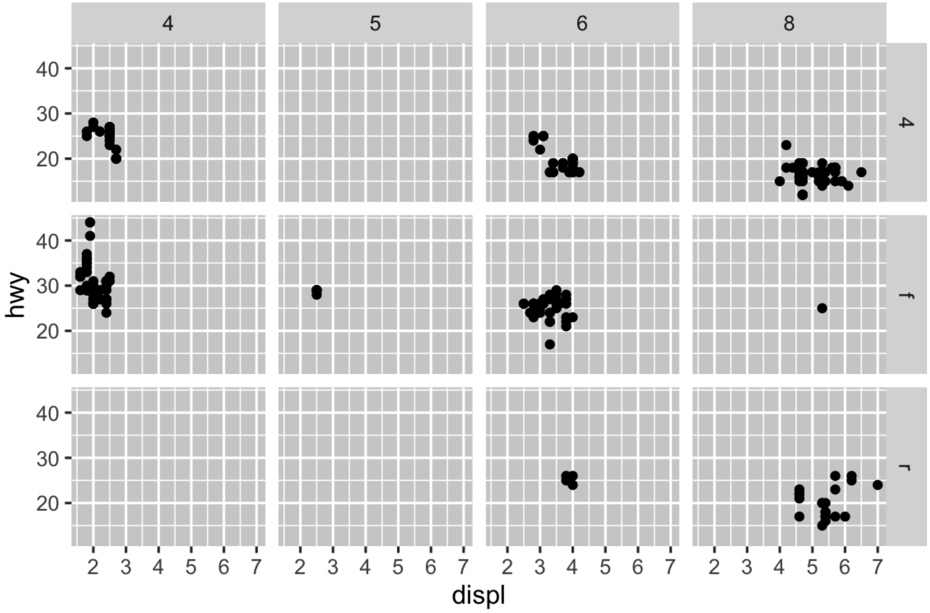
In Chapter 1 you learned about faceting with `facet_wrap()`, which splits a plot into subplots that each display one subset of the data based on a categorical variable.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_wrap(~cyl)
```



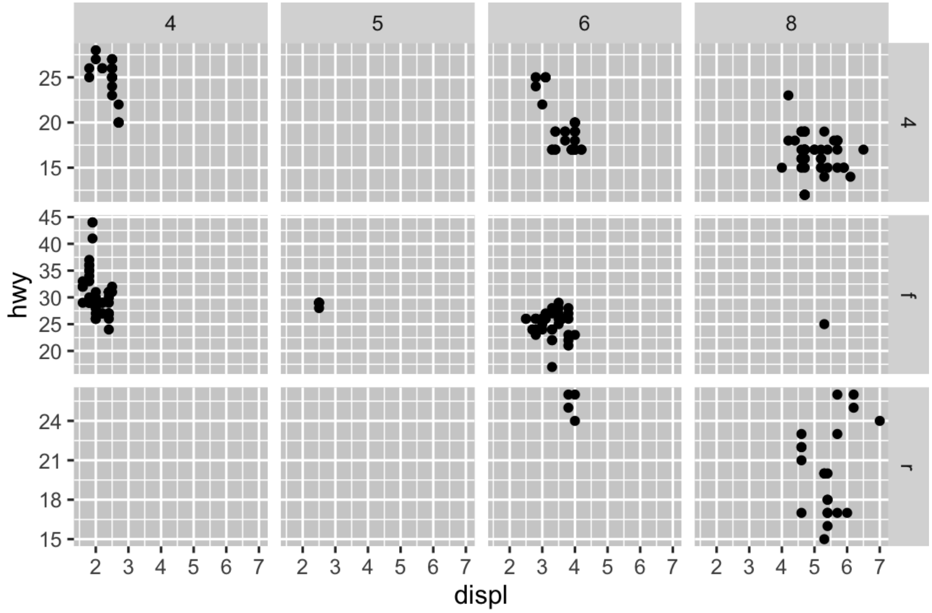
To facet your plot with the combination of two variables, switch from `facet_wrap()` to `facet_grid()`. The first argument of `facet_grid()` is also a formula, but now it's a double-sided formula: `rows ~ cols`.

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



By default each of the facets share the same scale and range for x and y axes. This is useful when you want to compare data across facets, but it can be limiting when you want to visualize the relationship within each facet better. Setting the `scales` argument in a faceting function to `"free"` will allow for different axis scales across both rows and columns, `"free_x"` will allow for different scales across rows, and `"free_y"` will allow for different scales across columns.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  facet_grid(drv ~ cyl, scales = "free_y")
```



Exercises

1. What happens if you facet on a continuous variable?

2. What do the empty cells in the plot with `facet_grid(drv ~ cyl)` mean? Run the following code. How do the cells relate to the resulting plot?

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

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

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

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

4. Take the first faceted plot in this section:

```
ggplot(mpg) +
  geom_point(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 arguments?

6. Which of the following plots makes it easier to compare engine size (displ) across cars with different drivetrains? What does this say about when to place a faceting variable across rows or columns?

```
ggplot(mpg, aes(x = displ)) + 
  geom_histogram() + 
  facet_grid(drv ~ .)

ggplot(mpg, aes(x = displ)) + 
  geom_histogram() +
  facet_grid(. ~ drv)
```

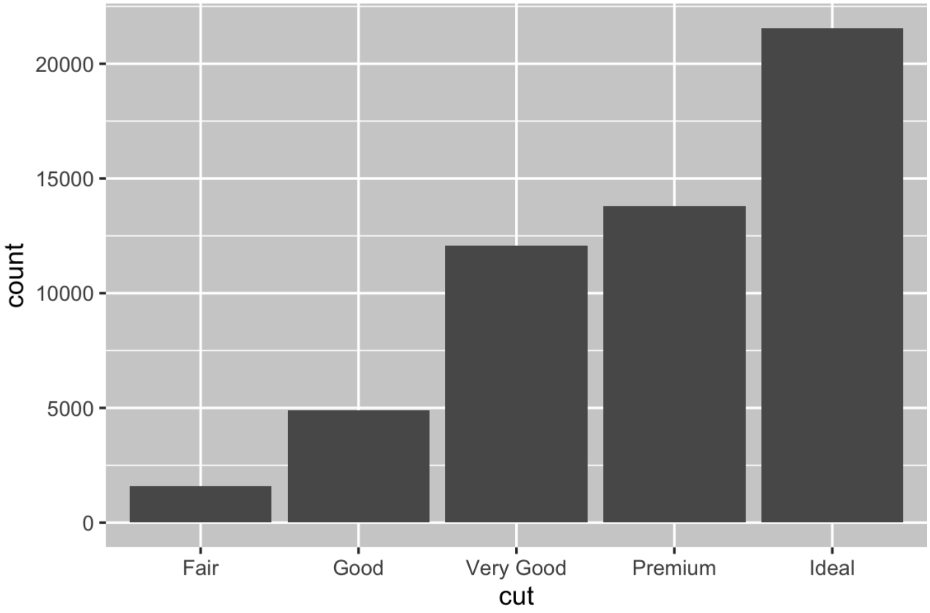
7. Re-create the following plot using facet_wrap() instead of facet_grid(). How do the positions of the facet labels change?

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

Statistical Transformations

Consider a basic bar chart drawn with geom_bar() or geom_col(). The following chart displays the total number of diamonds in the diamonds dataset, grouped by cut. The diamonds dataset is in the ggplot2 package and contains information on 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.

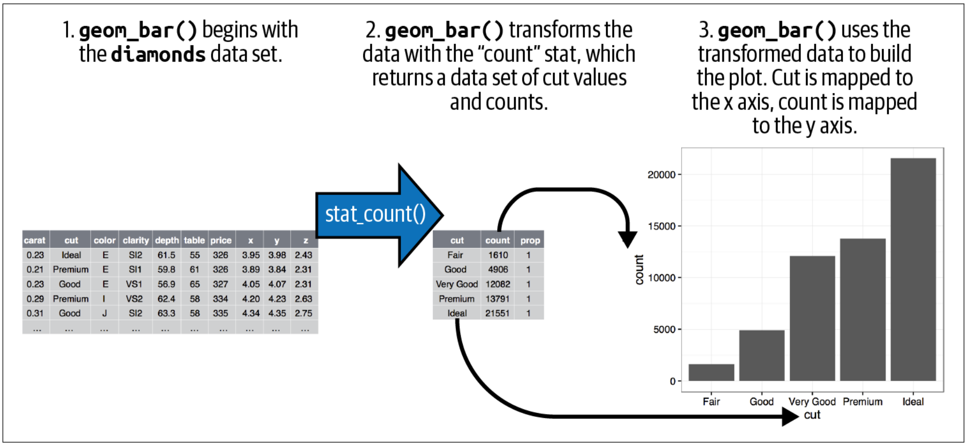
```
ggplot(diamonds, aes(x = cut)) + 
  geom_bar()
```



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 the five-number summary of the distribution and then display that summary as a specially formatted box.

The algorithm used to calculate new values for a graph is called a *stat*, short for statistical transformation. Figure 9-2 shows how this process works with `geom_bar()`.



1. `geom_bar()` begins with the `diamonds` data set.

2. `geom_bar()` transforms the data with the "count" stat, which returns a data set of cut values and counts.

3. `geom_bar()` uses the transformed data to build the plot. Cut is mapped to the x axis, count is mapped to the y axis.

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

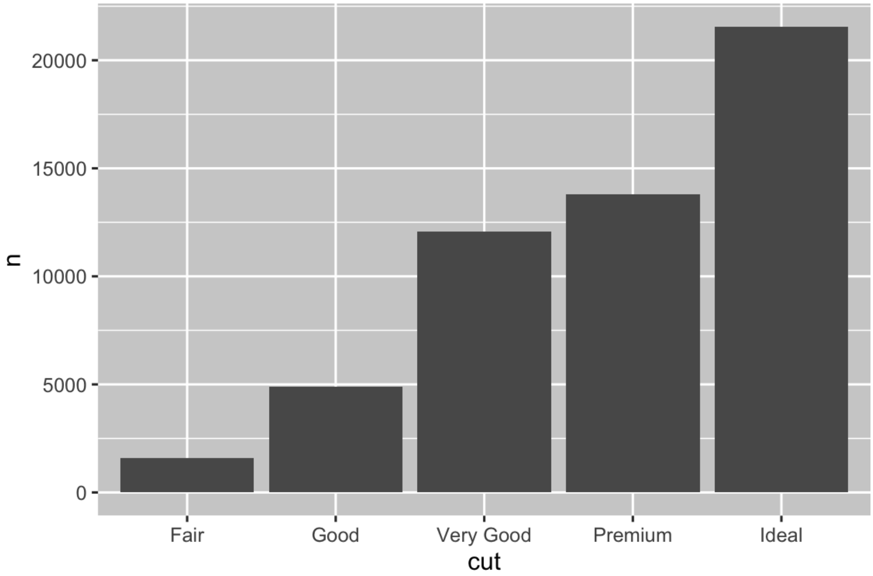
Figure 9-2. When creating a bar chart, we first start with the raw data, then aggregate it to count the number of observations in each bar, and finally map those computed variables to plot aesthetics.

You can learn which stat a geom uses by inspecting the default value for the `stat` argument. For example, `?geom_bar` shows that 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()`. If you scroll down, the section called "Computed variables" explains that it computes two new variables: `count` and `prop`.

Every geom has a default stat, and every stat has a default geom. This means you can typically use geoms without worrying about the underlying statistical transformation. However, there are three reasons why you might need to use a stat explicitly:

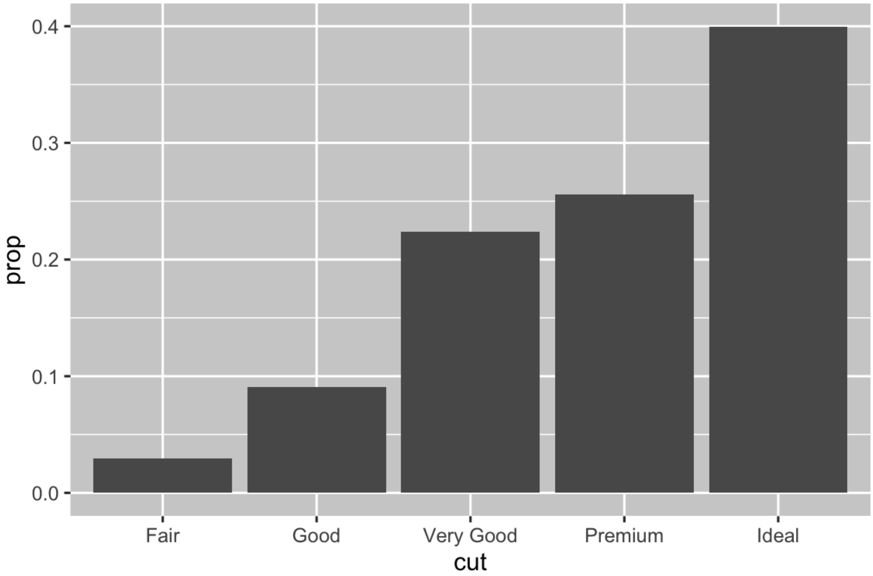
1. You might want to override the default stat. In the following code, we change the stat of `geom_bar()` from count (the default) to identity. This lets us map the height of the bars to the raw values of a y variable.

   ```
   diamonds |>
     count(cut) |>
     ggplot(aes(x = cut, y = n)) +
     geom_bar(stat = "identity")
   ```



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

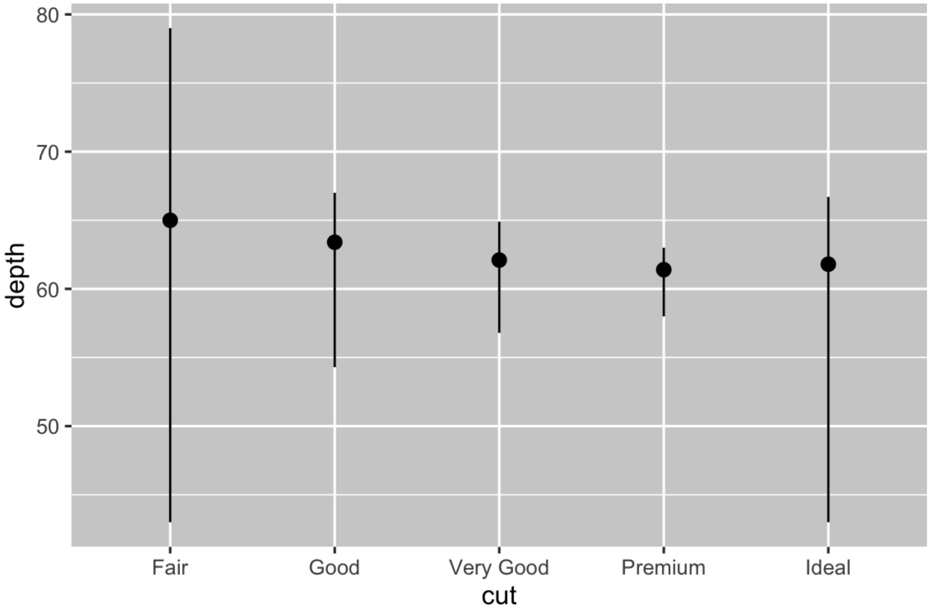
```
ggplot(diamonds, aes(x = cut, y = after_stat(prop), group = 1)) +
  geom_bar()
```



To find the possible variables that can be computed by the stat, look for the section titled "Computed variables" in the help for `geom_bar()`.

3. 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(diamonds) +
  stat_summary(
    aes(x = cut, y = depth),
    fun.min = min,
    fun.max = max,
    fun = median
  )
```



ggplot2 provides more than 20 stats for you to use. Each stat is a function, so you can get help in the usual way, e.g., `?stat_bin`.

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 from `geom_bar()`?

3. Most geoms and stats come in pairs that are almost always used in concert. Make a list of all the pairs. What do they have in common? (Hint: Read through the documentation.)

4. What variables does `stat_smooth()` compute? What arguments control its behavior?

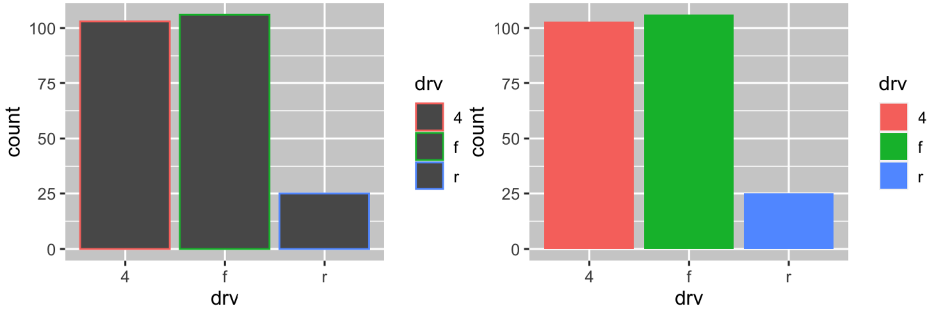
5. In our proportion bar chart, we need to set `group = 1`. Why? In other words, what is the problem with these two graphs?

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

Position Adjustments

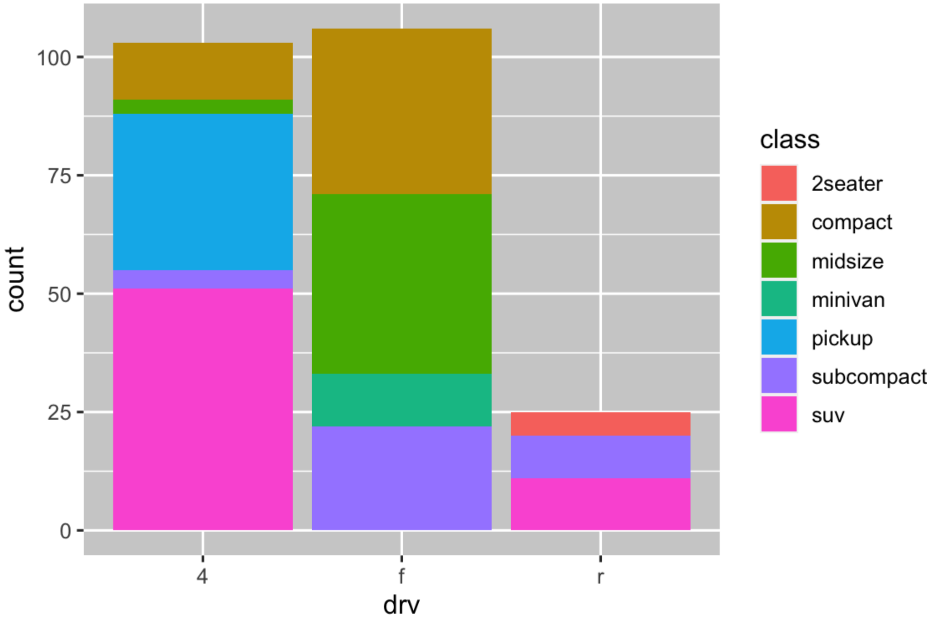
There's one more piece of magic associated with bar charts. You can color a bar chart using either the `color` aesthetic or, more usefully, the `fill` aesthetic:

```
# Left
ggplot(mpg, aes(x = drv, color = drv)) +
  geom_bar()

# Right
ggplot(mpg, aes(x = drv, fill = drv)) +
  geom_bar()
```



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

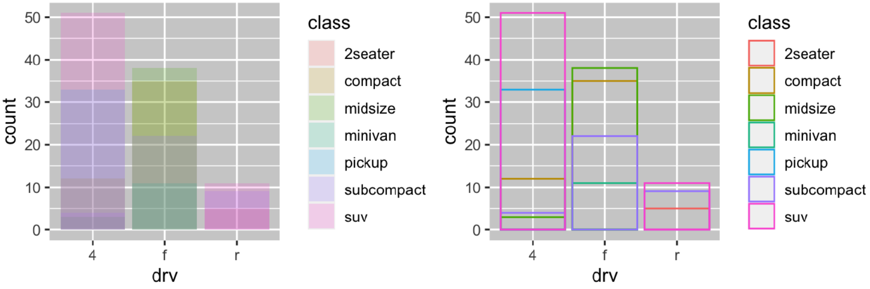
```
ggplot(mpg, aes(x = drv, fill = class)) +
  geom_bar()
```



The stacking is performed automatically using 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 need to make the bars either slightly transparent by setting `alpha` to a small value or completely transparent by setting `fill = NA`.

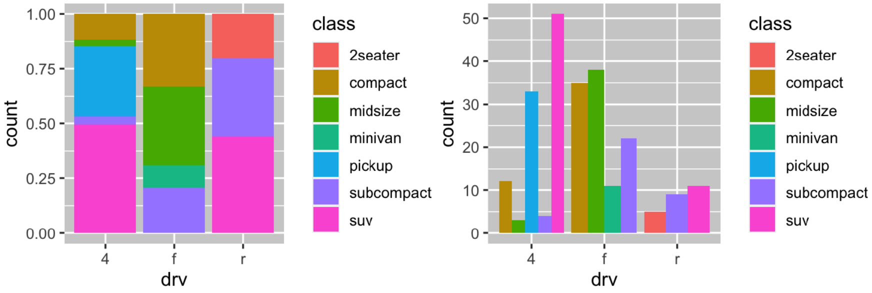
```
# Left
ggplot(mpg, aes(x = drv, fill = class)) +
  geom_bar(alpha = 1/5, position = "identity")

# Right
ggplot(mpg, aes(x = drv, color = class)) +
  geom_bar(fill = NA, position = "identity")
```



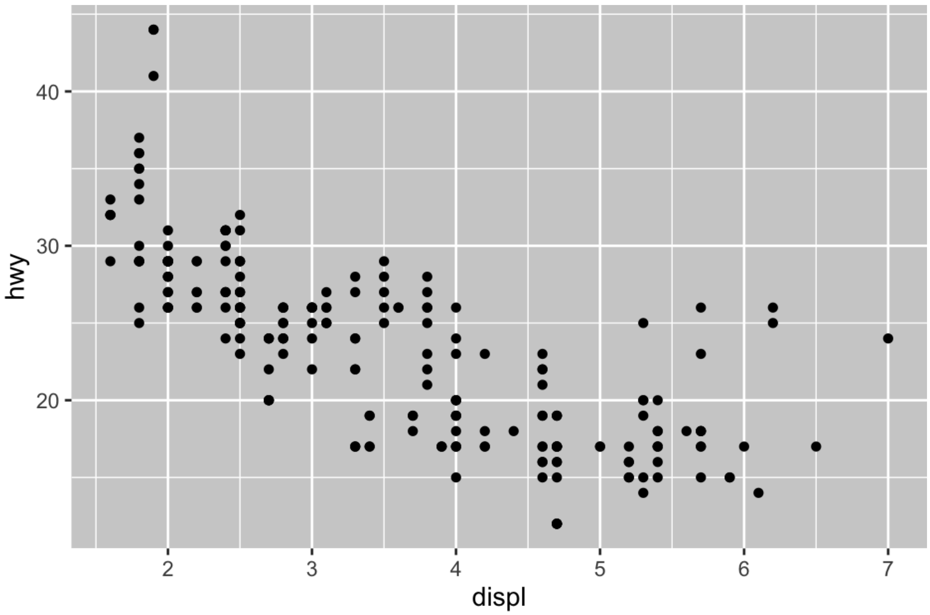
The identity position adjustment is more useful for 2D geoms, like points, where it is the default.

- `position = "fill"` works like stacking but makes each set of stacked bars the same height. This makes it easier to compare proportions across groups.
- `position = "dodge"` places overlapping objects directly *beside* one another. This makes it easier to compare individual values.

```
# Left
ggplot(mpg, aes(x = drv, fill = class)) +
  geom_bar(position = "fill")

# Right
ggplot(mpg, aes(x = drv, fill = class)) +
  geom_bar(position = "dodge")
```



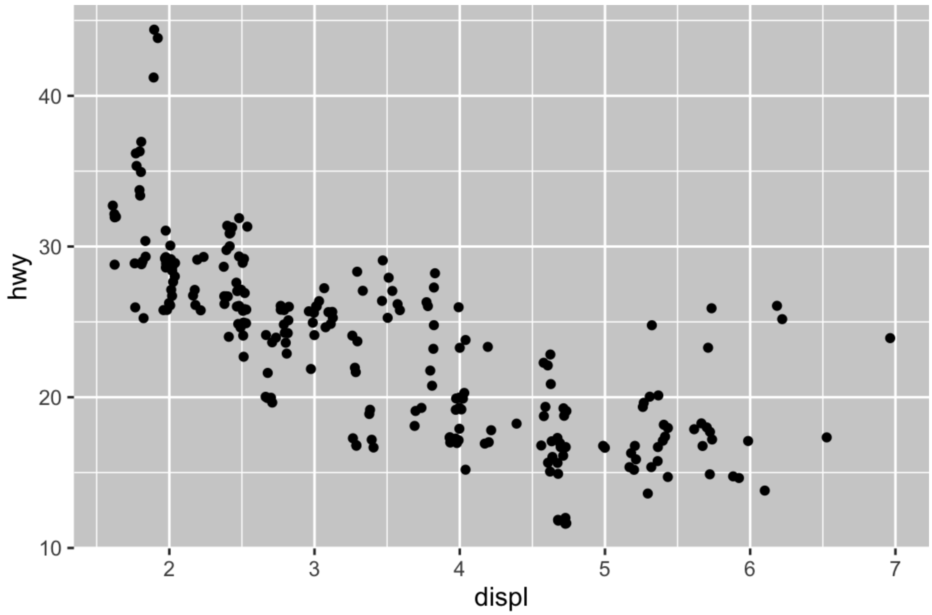
There's one other type of adjustment that's not useful for bar charts but can be very useful for scatterplots. Recall our first scatterplot. Did you notice that the plot displays only 126 points, even though there are 234 observations in the dataset?



The underlying 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 difficult to see the distribution of the data. 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`". Using `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(mpg, aes(x = displ, y = hwy)) +
  geom_point(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`
- `?position_stack`

Exercises

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

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

2. What, if anything, is the difference between the two plots? Why?

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

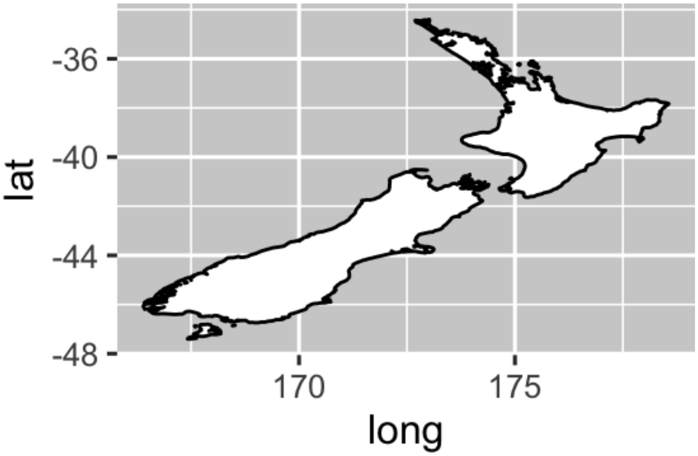
3. What parameters to `geom_jitter()` control the amount of jittering?
4. Compare and contrast `geom_jitter()` with `geom_count()`.
5. What's the default position adjustment for `geom_boxplot()`? Create a visualization of the `mpg` dataset that demonstrates it.

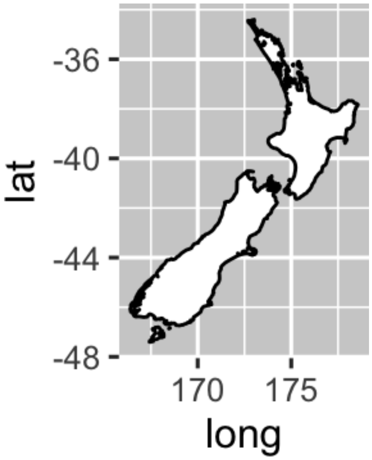
Coordinate Systems

Coordinate systems are probably the most complicated part of ggplot2. The default coordinate system is the Cartesian coordinate system where the x and y positions act independently to determine the location of each point. There are two other coordinate systems that are occasionally helpful.

- `coord_quickmap()` sets the aspect ratio correctly for geographic maps. This is important if you're plotting spatial data with ggplot2. We don't have the space to discuss maps in this book, but you can learn more in the Maps chapter of *ggplot2: Elegant Graphics for Data Analysis* (Springer).

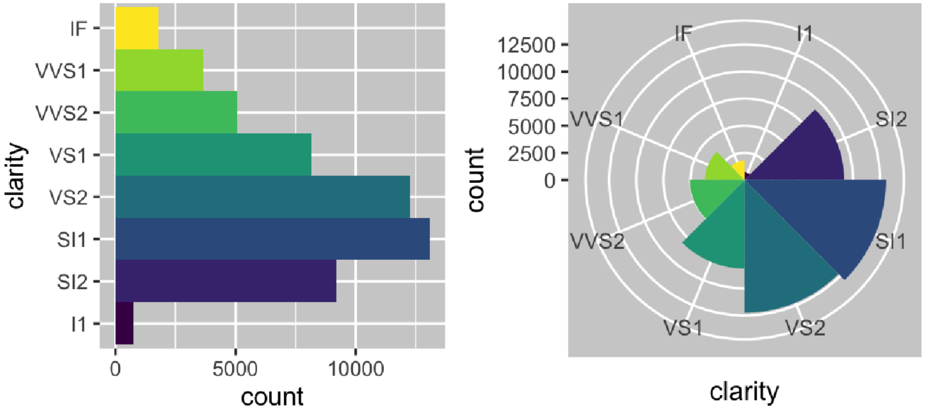
```
nz <- map_data("nz")

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

ggplot(nz, aes(x = long, y = 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 = clarity, fill = clarity),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1)

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



Exercises

1. Turn a stacked bar chart into a pie chart using `coord_polar()`.
2. What's the difference between `coord_quickmap()` and `coord_map()`?
3. 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

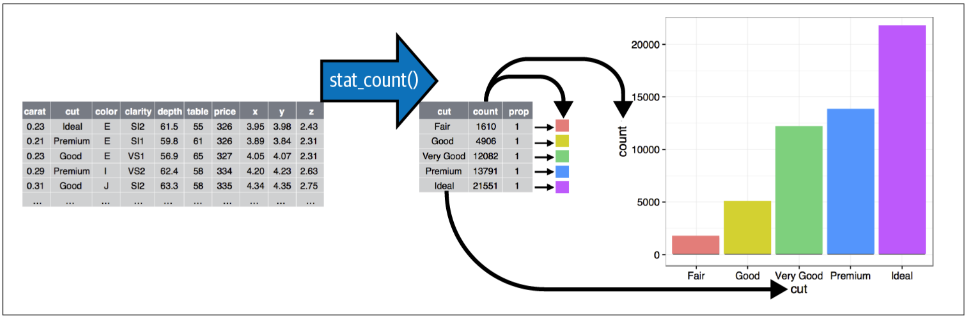
We can expand on the graphing template you learned in "ggplot2 Calls" on page 16 by adding position adjustments, stats, coordinate systems, and faceting:

```
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, a faceting scheme, and a theme.

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). 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. These steps are illustrated in Figure 9-3. You'd then select a coordinate system to place the geoms into, using the location of the objects (which is itself an aesthetic property) to display the values of the x and y variables.



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

Figure 9-3. These are the steps for going from raw data to a table of frequencies to a bar plot where the heights of the bar represent the frequencies.

At this 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.

If you'd like to learn more about the theoretical underpinnings of ggplot2, you might enjoy reading "A Layered Grammar of Graphics", the scientific paper that describes the theory of ggplot2 in detail.

Summary

In this chapter you learned about the layered grammar of graphics starting with aesthetics and geometries to build a simple plot, facets for splitting the plot into subsets, statistics for understanding how geoms are calculated, position adjustments for controlling the fine details of position when geoms might otherwise overlap, and coordinate systems that allow you to fundamentally change what x and y mean. One layer we have not yet touched on is theme, which we will introduce in "Themes" on page 193.

Two very useful resources for getting an overview of the complete ggplot2 functionality are the ggplot2 cheatsheet and the ggplot2 package website.

An important lesson you should take from this chapter is that when you feel the need for a geom that is not provided by ggplot2, it's always a good idea to look into whether someone else has already solved your problem by creating a ggplot2 extension package that offers that geom.

CHAPTER 10

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 home in on a few particularly productive insights that you'll eventually write up and communicate to others.

EDA is an important part of any data analysis, even if the primary research 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 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 can be gleaned from 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?

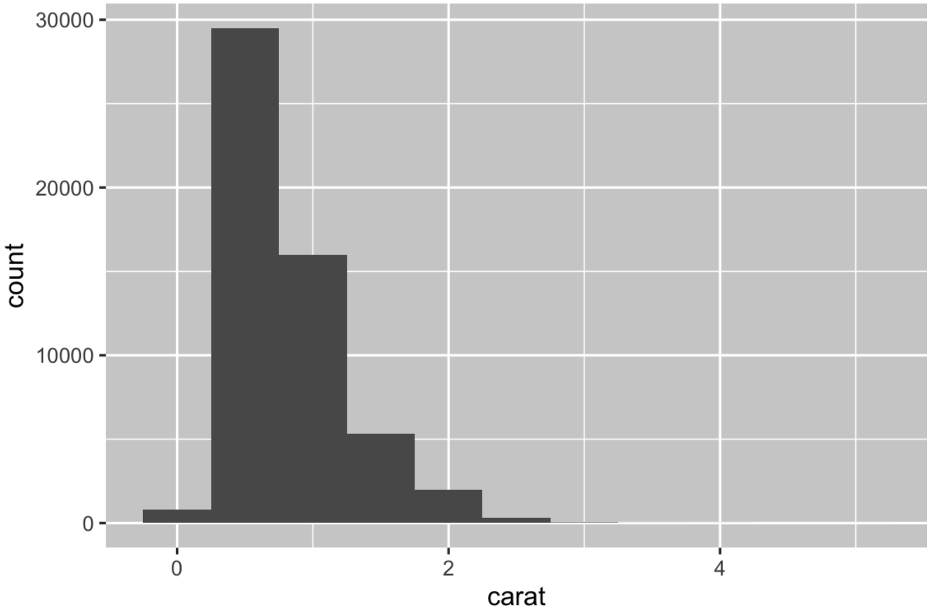
The rest of this chapter will look at these two questions. We'll explain what variation and covariation are, and we'll show you several ways to answer each question.

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. Variables can also vary if you measure across different subjects (e.g., the eye colors of different people) or at 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 about how it varies between measurements on the same observation as well as across observations. The best way to understand that pattern is to visualize the distribution of the variable's values, which you've learned about in Chapter 1.

We'll start our exploration by visualizing the distribution of weights (`carat`) of about 54,000 diamonds from the `diamonds` dataset. Since `carat` is a numerical variable, we can use a histogram:

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



Now that you can visualize variation, what should you look for in your plots? And what type of follow-up questions should you ask? We've put together a list in the next section of the most useful types of information that you will find in your graphs, along with some follow-up questions for each type of information. The key to asking good follow-up questions will be to rely on your curiosity (what do you want to learn more about?) as well as your skepticism (how could this be misleading?).

Typical Values

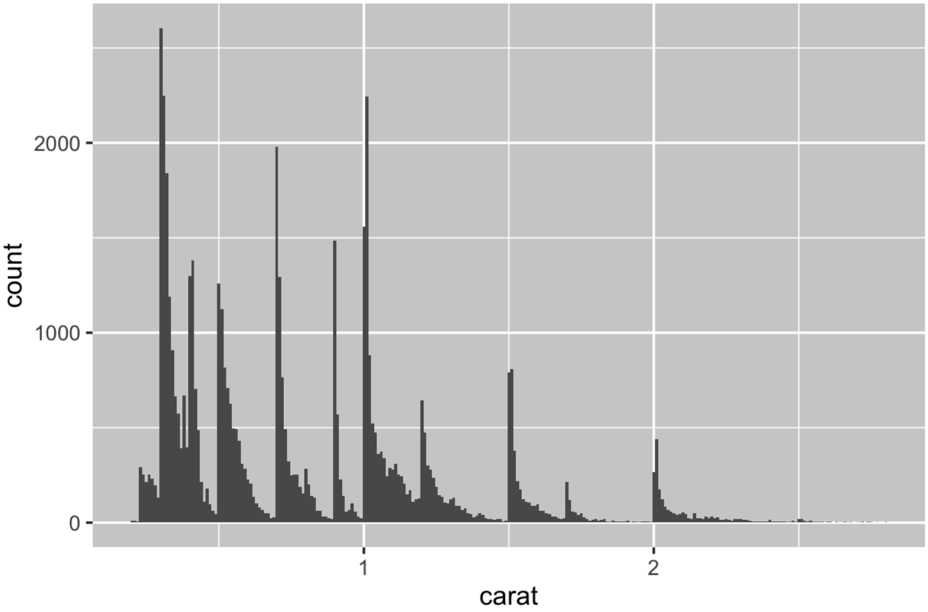
In both bar charts and histograms, tall bars show the common values of a variable, and shorter bars show less-common values. Places that do not have bars reveal values that were not seen in your data. To turn this information into useful questions, look for anything unexpected:

- 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?

Let's take a look at the distribution of `carat` for smaller diamonds:

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

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



This 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?

Visualizations can also reveal clusters, which suggest that subgroups exist in your data. To understand the subgroups, ask:

- How are the observations within each subgroup similar to each other?
- How are the observations in separate clusters different from each other?

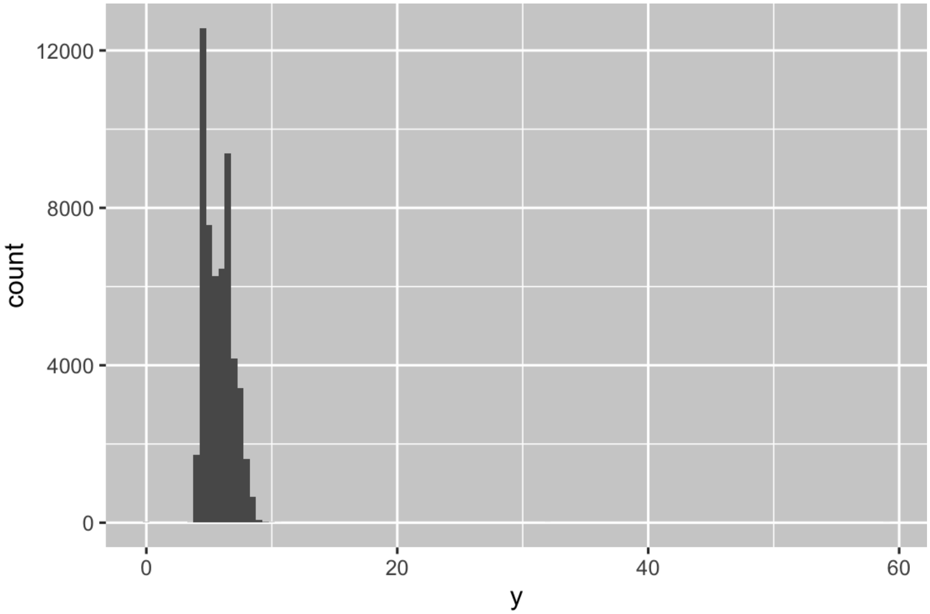
- How can you explain or describe the clusters?
- Why might the appearance of clusters be misleading?

Some of these questions can be answered with the data, while some will require domain expertise about the data. Many of them 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

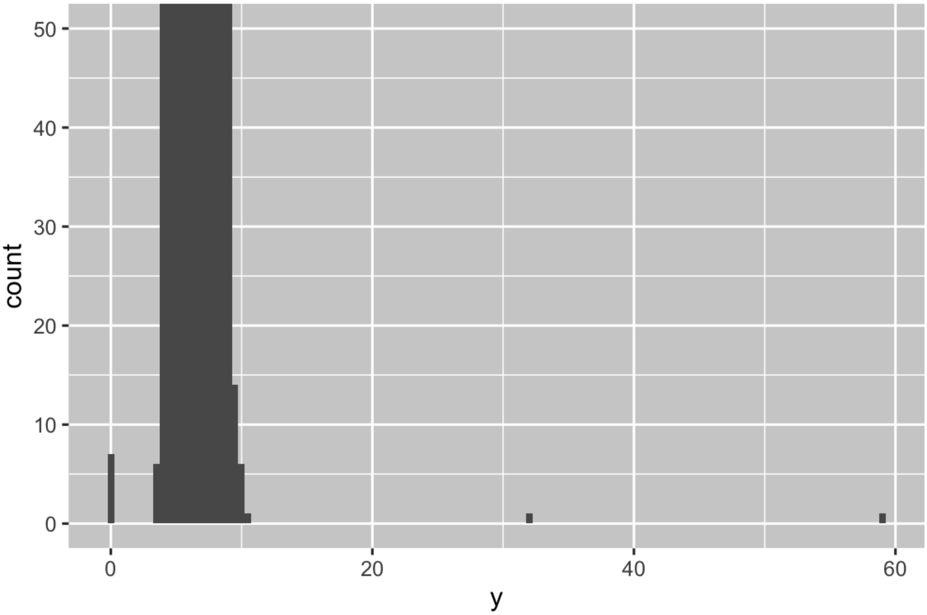
Outliers are observations that are unusual, in other words, data points that don't seem to fit the pattern. Sometimes outliers are data entry errors, sometimes they are simply values at the extremes that happened to be observed in this data collection, and other times they suggest important new discoveries. 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 diamonds dataset. The only evidence of outliers is the unusually wide limits on the x-axis.

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



There are so many observations in the common bins that the rare bins are very short, making it difficult to 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 to small values of the y-axis with coord_cartesian():

```
ggplot(diamonds, aes(x = y)) +
  geom_histogram(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) |>
  select(price, x, y, z) |>
  arrange(y)
unusual
#> # A tibble: 9 × 4
#>   price     x     y     z
#>   <int> <dbl> <dbl> <dbl>
#> 1  5139  0      0    0
#> 2  6381  0      0    0
#> 3 12800  0      0    0
#> 4 15686  0      0    0
#> 5 18034  0      0    0
#> 6  2130  0       0    0
#> 7  2130  0       0    0
#> 8  2075  5.15  31.8  5.12
#> 9 12210  8.09  58.9  8.06
```

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. By doing EDA, we have discovered missing data that were coded as 0, which we never would have found by simply searching for NAs. Going forward we might choose to re-code these values as NAs to prevent misleading calculations. We might also suspect that measurements of 32mm and 59mm are implausible: those diamonds are more than 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 omit them 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() and xlim() or ylim() when zooming in on a histogram. What happens if you leave binwidth unset? What happens if you try to zoom so only half a bar shows?

Unusual 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:

1. Drop the entire row with the strange values:

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

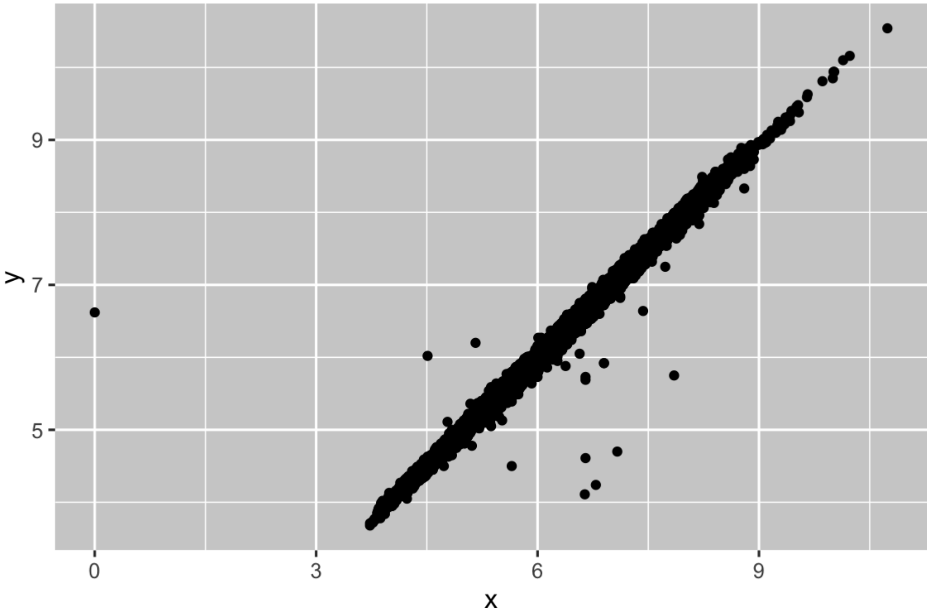
   We don't recommend this option because one invalid value doesn't imply that all the other values for that observation are also invalid. Additionally, if you have low-quality data, by the time that you've applied this approach to every variable you might find that you don't have any data left!

2. Instead, we 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 `if_else()` function to replace unusual values with `NA`:

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

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



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

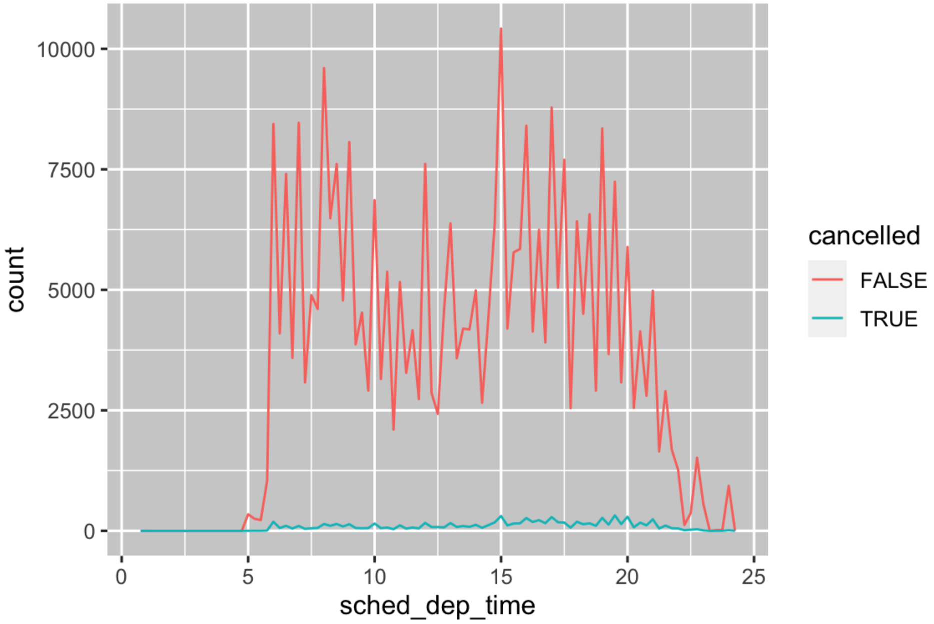
```
ggplot(diamonds2, aes(x = x, y = y)) +
  geom_point(na.rm = TRUE)
```

Other times you want to understand what makes observations with missing values different to observations with recorded values. For example, in `nycflights13::flights`,[1] missing values in the `dep_time` variable indicate that the

[1] Remember that when we need to be explicit about where a function (or dataset) comes from, we'll use the special form `package::function()` or `package::dataset`.

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, using `is.na()` to check whether `dep_time` is missing.

```
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(aes(x = sched_dep_time)) +
  geom_freqpoly(aes(color = cancelled), binwidth = 1/4)
```



However, this plot isn't great because there are many more noncancelled 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 in how missing values are handled in histograms and bar charts?

2. What does `na.rm = TRUE` do in `mean()` and `sum()`?

3. Re-create the frequency plot of `scheduled_dep_time` colored by whether the flight was cancelled or not. Also facet by the `cancelled` variable. Experiment with different values of the `scales` variable in the faceting function to mitigate the effect of more noncancelled flights than cancelled flights.

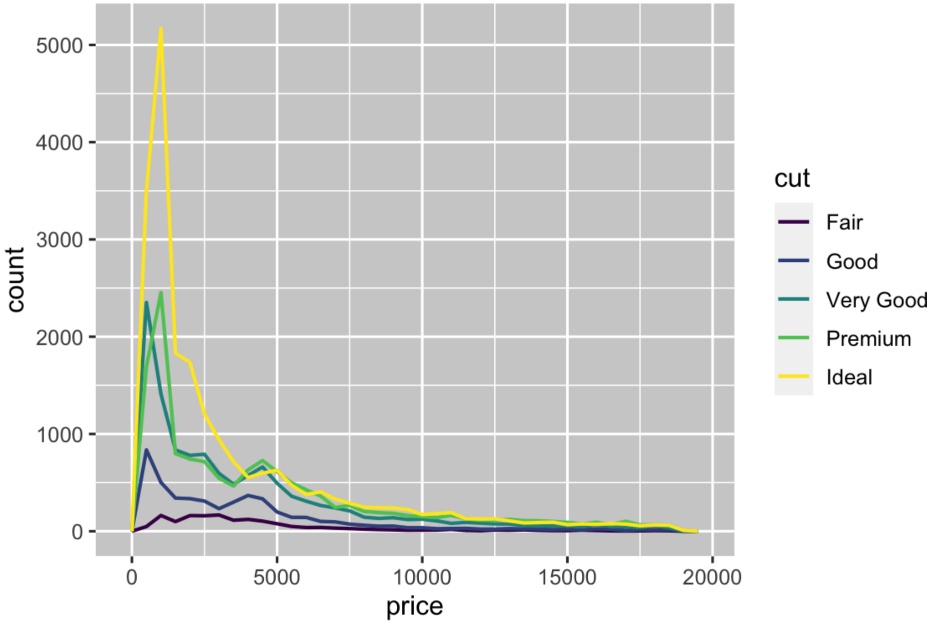
Covariation

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.

A Categorical and a Numerical Variable

For example, let's explore how the price of a diamond varies with its quality (measured by `cut`) using `geom_freqpoly()`:

```
ggplot(diamonds, aes(x = price)) +
  geom_freqpoly(aes(color = cut), binwidth = 500, linewidth = 0.75)
```

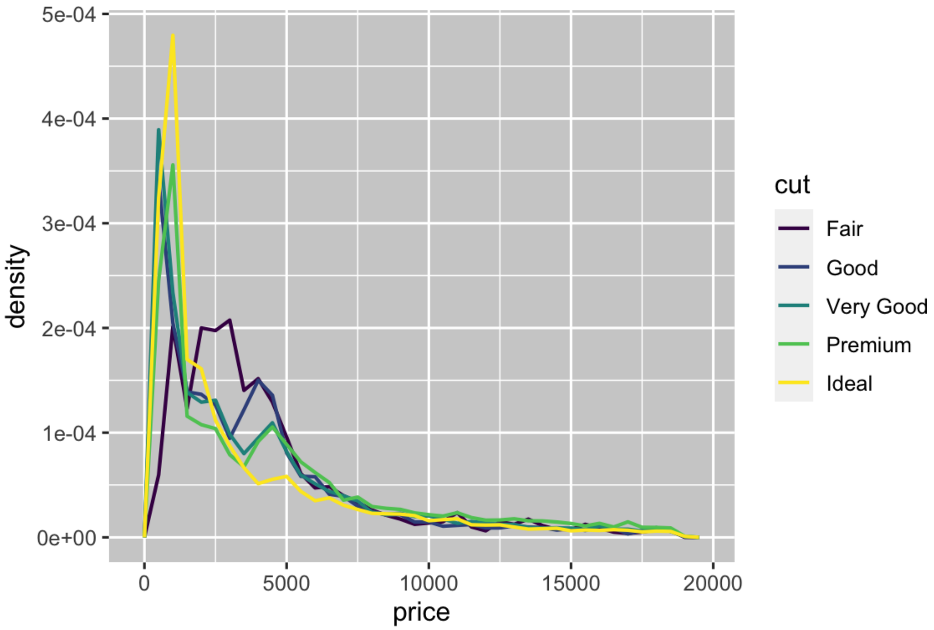


Note that ggplot2 uses an ordered color scale for `cut` because it's defined as an ordered factor variable in the data. You'll learn more about these in "Ordered Factors" on page 295.

The default appearance of geom_freqpoly() is not that useful here because the height, determined by the overall count, differs so much across cuts, making it hard to see the differences in the shapes of their distributions.

To make the comparison easier, we need to swap what is displayed on the y-axis. Instead of displaying count, we'll display the *density*, which is the count standardized so that the area under each frequency polygon is 1:

```
ggplot(diamonds, aes(x = price, y = after_stat(density))) +
  geom_freqpoly(aes(color = cut), binwidth = 500, linewidth = 0.75)
```

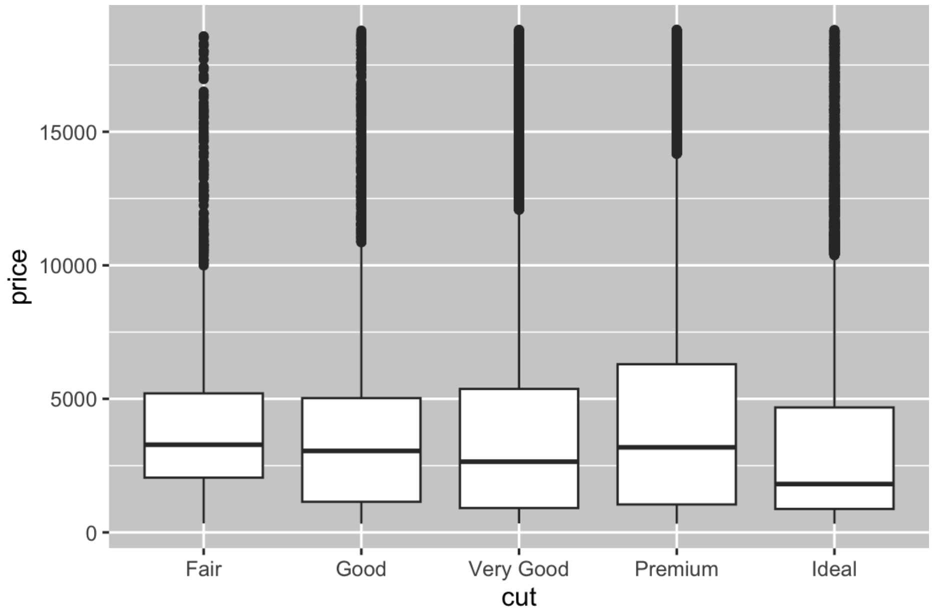


Note that we're mapping the density the y, but since density is not a variable in the diamonds dataset, we need to first calculate it. We use the after_stat() function to do so.

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.

A visually simpler plot for exploring this relationship is using side-by-side boxplots:

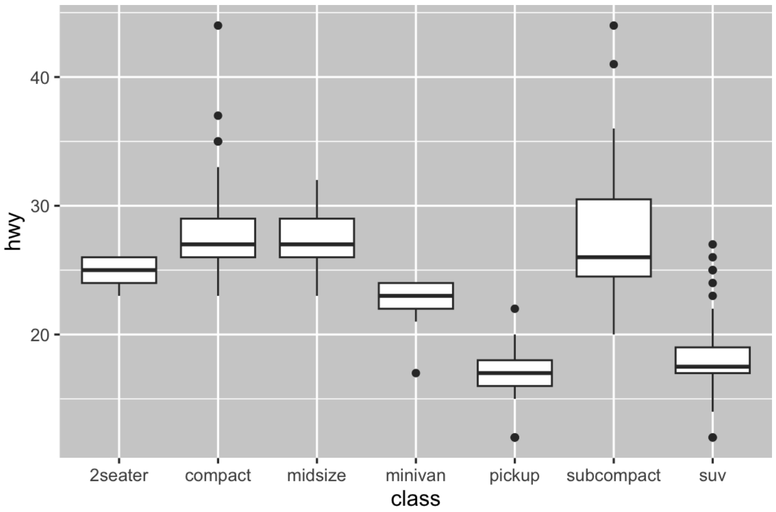
```
ggplot(diamonds, aes(x = cut, y = price)) +
  geom_boxplot()
```



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 typically cheaper! In the exercises, you'll be challenged to figure out why.

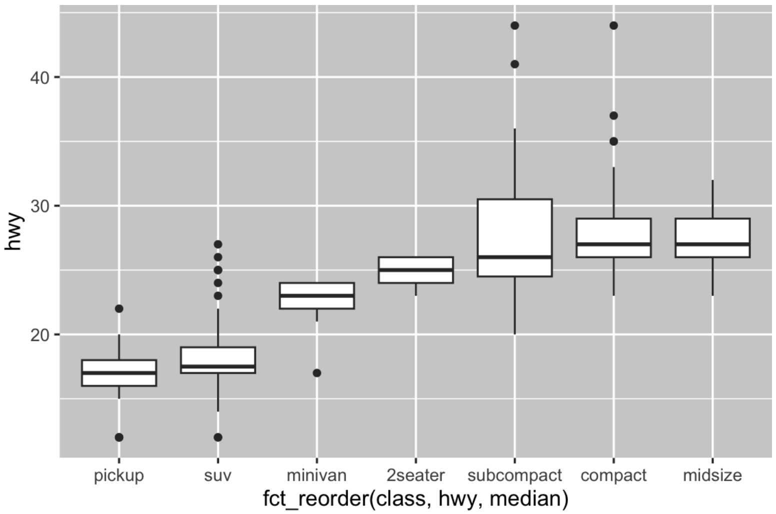
`cut` is an ordered factor: fair is worse than good, which is worse than very good and so on. Many categorical variables don't have such an intrinsic order, so you might want to reorder them to make a more informative display. One way to do that is with `fct_reorder()`. You'll learn more about that function in "Modifying Factor Order" on page 288, but we wanted to give you a quick preview here because it's so useful. For example, take the `class` variable in the `mpg` dataset. You might be interested to know how highway mileage varies across classes:

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



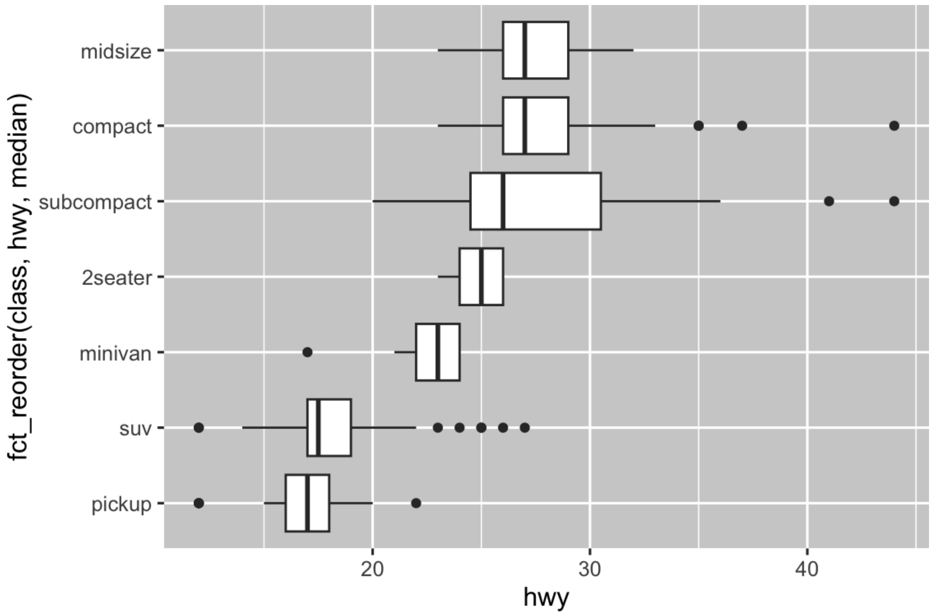
To make the trend easier to see, we can reorder `class` based on the median value of `hwy`:

```
ggplot(mpg, aes(x = fct_reorder(class, hwy, median), y = hwy)) +
  geom_boxplot()
```



If you have long variable names, `geom_boxplot()` will work better if you flip it 90°. You can do that by exchanging the x and y aesthetic mappings:

```
ggplot(mpg, aes(x = hwy, y = fct_reorder(class, hwy, median))) +
  geom_boxplot()
```



Exercises

1. Use what you've learned to improve the visualization of the departure times of cancelled versus noncancelled flights.
2. Based on EDA, what variable in the diamonds dataset appears to be 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. Instead of exchanging the x and y variables, add `coord_flip()` as a new layer to the vertical boxplot to create a horizontal one. How does this compare to exchanging the variables?
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. Create a visualization of diamond prices versus a categorical variable from the `diamonds` dataset using `geom_violin()`, then a faceted `geom_histogram()`, then a colored `geom_freqpoly()`, and then a colored `geom_density()`. Compare and contrast the four plots. What are the pros and cons of each method of visualizing

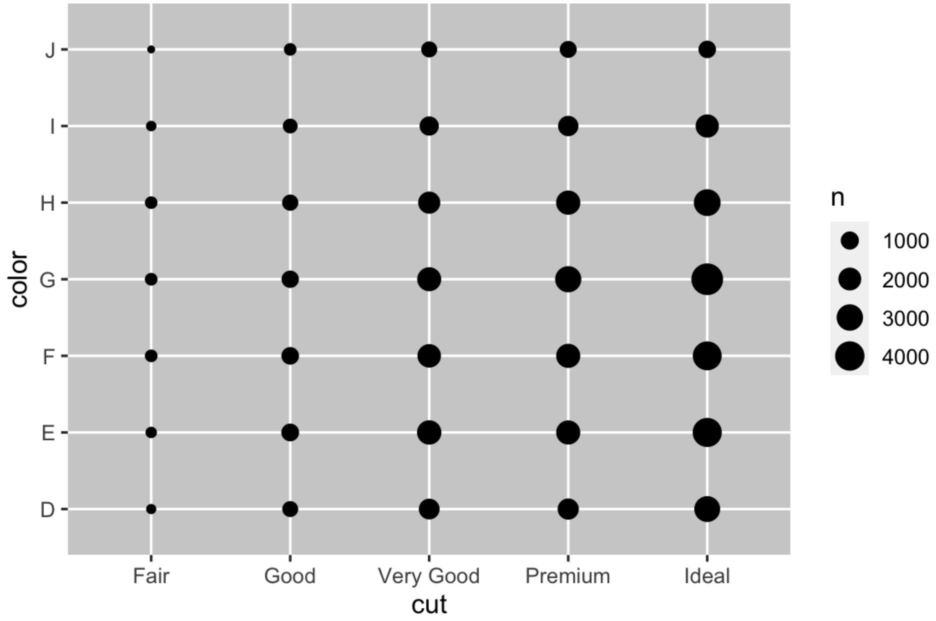
the distribution of a numerical variable based on the levels of a categorical variable?

6. If you have a small dataset, it's sometimes useful to use `geom_jitter()` to avoid overplotting to more easily 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 of levels of these categorical variables. One way to do that is to rely on the built-in `geom_count()`:

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



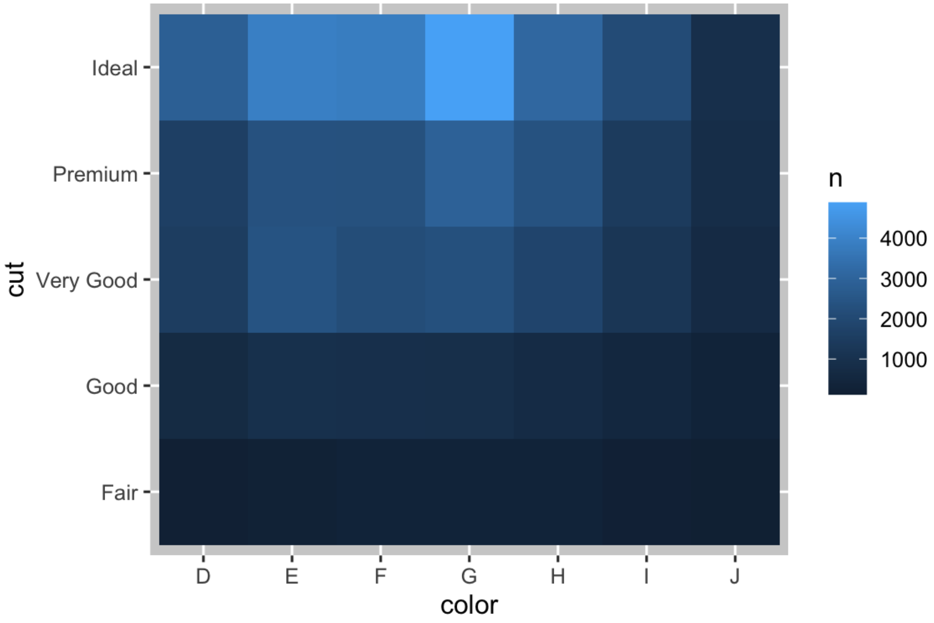
The size of each circle in the plot displays how many observations occurred at each combination of values. Covariation will appear as a strong correlation between specific x values and specific y values.

Another approach for exploring the relationship between these variables is computing the counts with dplyr:

```
diamonds |>
  count(color, cut)
#> # A tibble: 35 × 3
#>   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(aes(x = color, y = cut)) +
  geom_tile(aes(fill = n))
```



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

Exercises

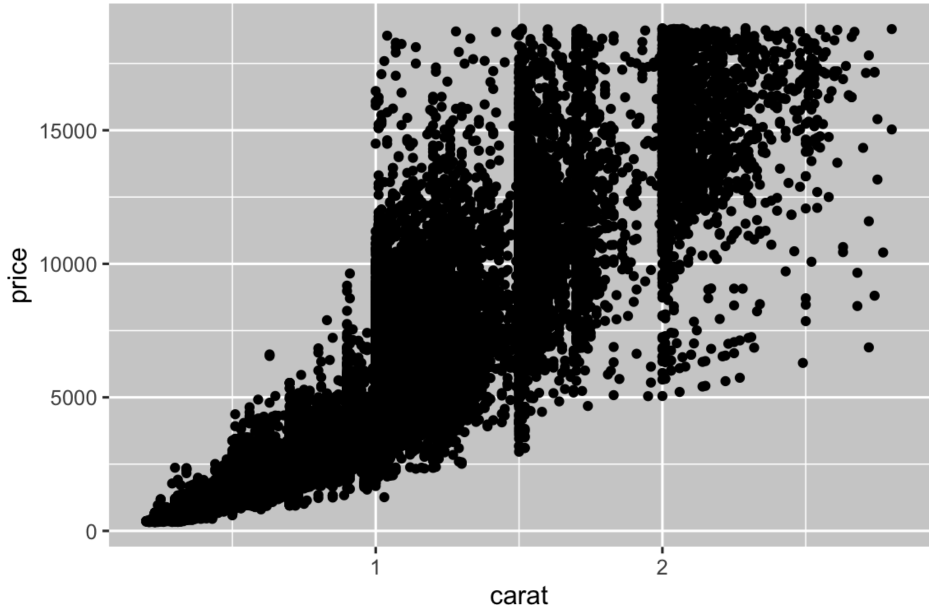
1. How could you rescale the previous count dataset to more clearly show the distribution of cut within color, or color within cut?

2. What different data insights do you get with a segmented bar chart if color is mapped to the x aesthetic and cut is mapped to the fill aesthetic? Calculate the counts that fall into each of the segments.
3. Use geom_tile() together with dplyr to explore how average flight departure delays vary by destination and month of year. What makes the plot difficult to read? How could you improve it?

Two Numerical Variables

You've already seen one great way to visualize the covariation between two numerical variables: draw a scatterplot with geom_point(). You can see covariation as a pattern in the points. For example, you can see a positive relationship between the carat size and price of a diamond: diamonds with more carats have a higher price. The relationship is exponential.

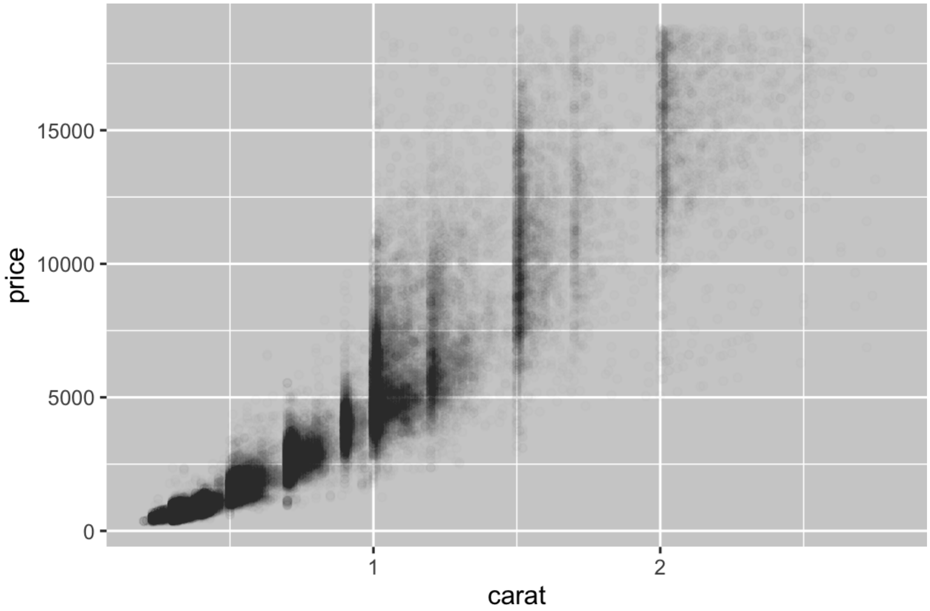
```
ggplot(smaller, aes(x = carat, y = price)) +
  geom_point()
```



(In this section we'll use the smaller dataset to stay focused on the bulk of the diamonds that are smaller than 3 carats.)

Scatterplots become less useful as the size of your dataset grows, because points begin to overplot and pile up into areas of uniform black, making it hard to judge differences in the density of the data across the two-dimensional space as well as making it hard to spot the trend. You've already seen one way to fix the problem: using the alpha aesthetic to add transparency.

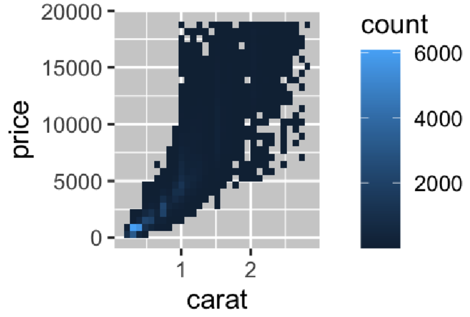
```
ggplot(smaller, aes(x = carat, y = price)) +
  geom_point(alpha = 1 / 100)
```

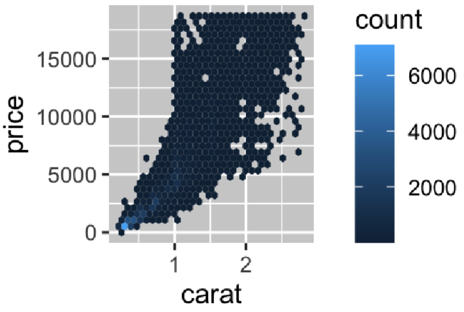


But using transparency can be challenging for very large datasets. Another solution is to use bins. Previously you used geom_histogram() and geom_freqpoly() to bin in one dimension. Now you'll learn how to use geom_bin2d() and geom_hex() to bin in two dimensions.

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().

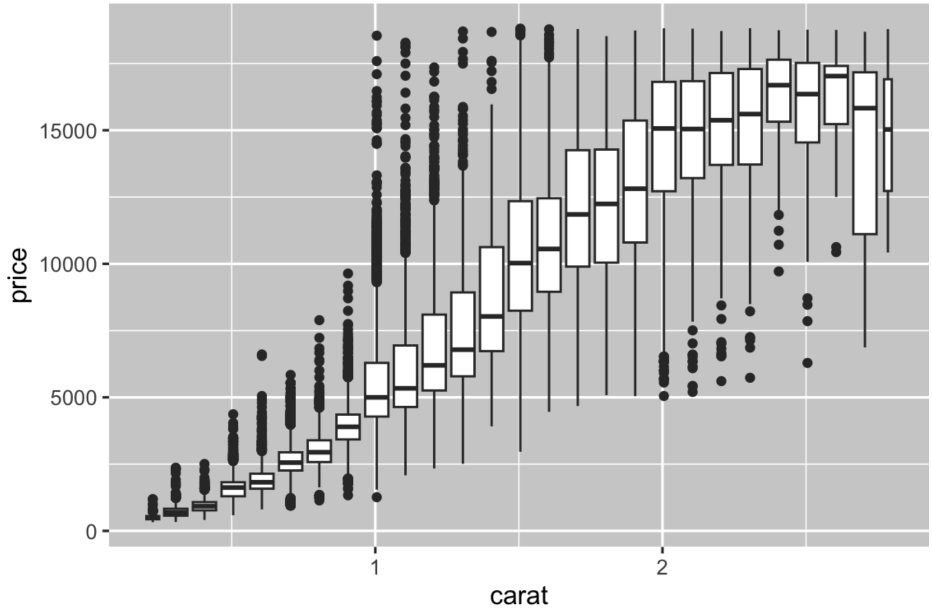
```
ggplot(smaller, aes(x = carat, y = price)) +
  geom_bin2d()

# install.packages("hexbin")
ggplot(smaller, aes(x = carat, y = price)) +
  geom_hex()
```





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(smaller, aes(x = carat, y = price)) +
  geom_boxplot(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`.

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

    ```
    diamonds |>
      filter(x >= 4) |>
      ggplot(aes(x = x, y = y)) +
      geom_point() +
      coord_cartesian(xlim = c(4, 11), ylim = c(4, 11))
    ```

6. Instead of creating boxes of equal width with `cut_width()`, we could create boxes that contain roughly equal number of points with `cut_number()`. What are the advantages and disadvantages of this approach?

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

Patterns and Models

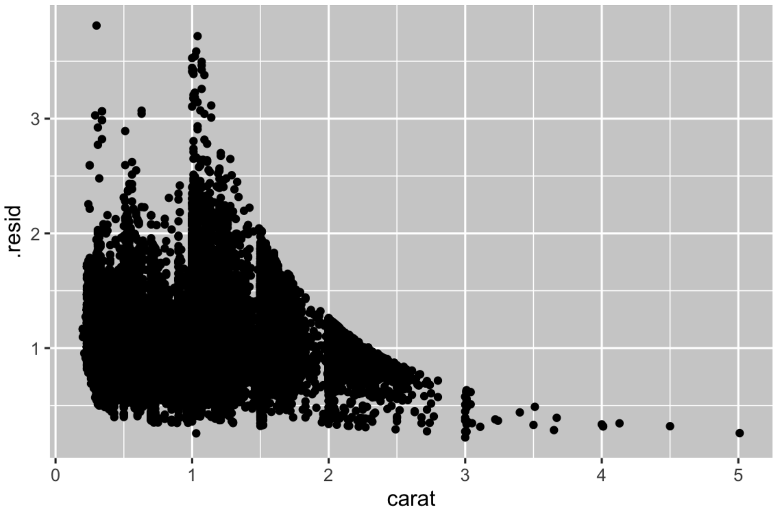
If a systematic relationship exists between two variables, it will appear as a pattern in the data. If you spot a pattern, ask yourself:

- 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?

Patterns in your data provide clues about relationships; i.e., 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.

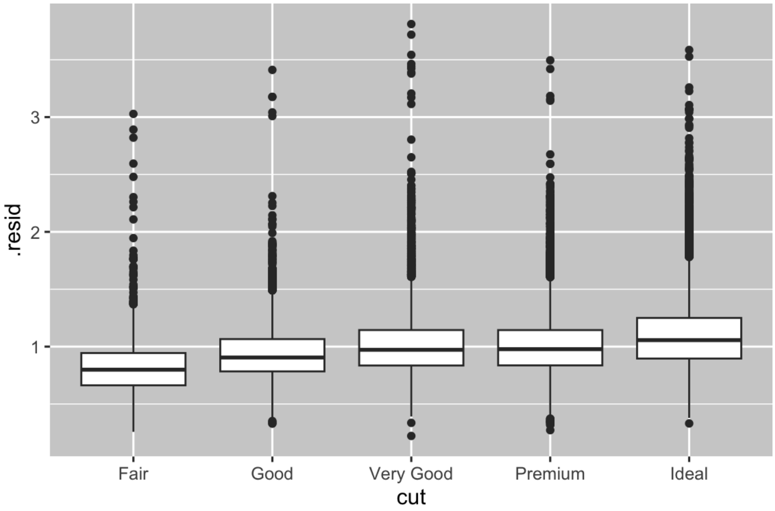
Models are a tool for extracting patterns out of data. For example, consider the diamonds data. It's hard to understand the relationship between cut and price, because cut and carat, and carat and price, are tightly related. It's possible to use a model to remove the very strong relationship between price and carat to 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. Note that instead of using the raw values of `price` and `carat`, we log transform them first and fit a model to the log-transformed values. Then, we exponentiate the residuals to put them back in the scale of raw prices.

```
library(tidymodels)

diamonds <- diamonds |>
  mutate(
    log_price = log(price),
    log_carat = log(carat)
  )

diamonds_fit <- linear_reg() |>
  fit(log_price ~ log_carat, data = diamonds)

diamonds_aug <- augment(diamonds_fit, new_data = diamonds) |>
  mutate(.resid = exp(.resid))

ggplot(diamonds_aug, aes(x = carat, y = .resid)) +
  geom_point()
```



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(diamonds_aug, aes(x = cut, y = .resid)) +
  geom_boxplot()
```



We're not discussing modeling in this book because understanding what models are and how they work is easiest once you have tools for data wrangling and programming in hand.

Summary

In this chapter you learned a variety of tools to help you understand the variation within your data. You saw a technique that works with a single variable at a time and with a pair of variables. This might seem painfully restrictive if you have tens or hundreds of variables in your data, but they're the foundation upon which all other techniques are built.

In the next chapter, we'll focus on the tools we can use to communicate our results.

CHAPTER 11

Communication

Introduction

In Chapter 10, you learned how to use plots as tools for *exploration*. When you make exploratory plots, you know—even before looking—which variables the plot will display. You made each plot for a purpose, could quickly look at it, and could then move on to the next plot. In the course of most analyses, you'll produce tens or hundreds of plots, most of which are immediately thrown away.

Now that you understand your data, you need to *communicate* your understanding to others. Your audience will likely not share your background knowledge and will not be deeply invested in the data. To help others quickly build up a good mental model of the data, you will need to invest considerable effort in making your plots as self-explanatory as possible. In this chapter, you'll learn some of the tools that ggplot2 provides to do so.

This chapter focuses on the tools you need to create good graphics. We assume that you know what you want and just need to know how to do it. For that reason, we highly recommend pairing this chapter with a good general visualization book. We particularly like *The Truthful Art* by Albert Cairo (New Riders). It doesn't teach the mechanics of creating visualizations but instead focuses on what you need to think about to create effective graphics.

Prerequisites

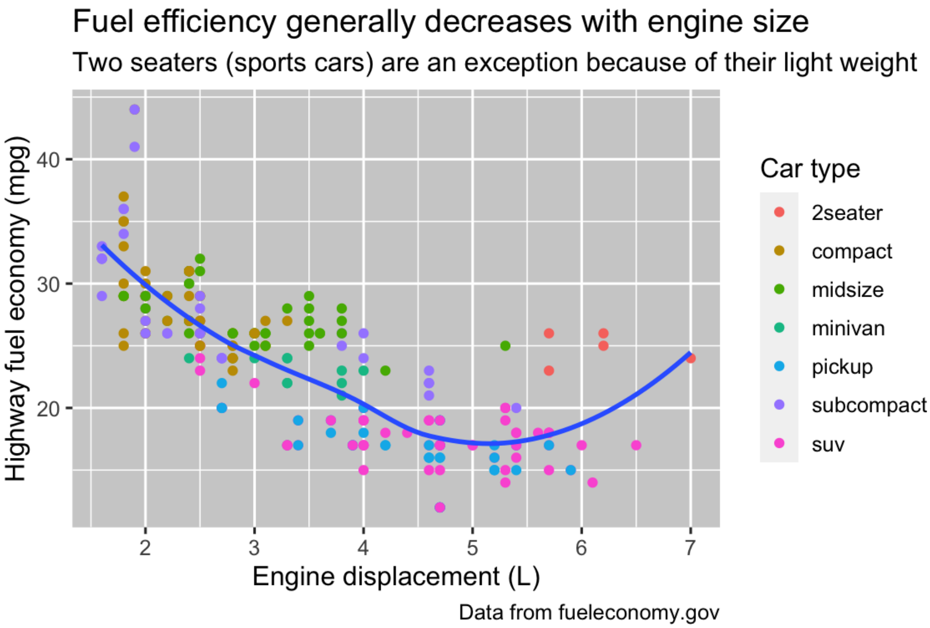
In this chapter, we'll focus once again on ggplot2. We'll also use a little dplyr for data manipulation; *scales* to override the default breaks, labels, transformations and palettes; and a few ggplot2 extension packages, including ggrepel by Kamil Slowikowski and patchwork by Thomas Lin Pedersen. Don't forget that you'll need to install those packages with `install.packages()` if you don't already have them.

```
library(tidyverse)
library(scales)
library(ggrepel)
library(patchwork)
```

Labels

The easiest place to start when turning an exploratory graphic into an expository graphic is with good labels. You add labels with the `labs()` function:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  labs(
    x = "Engine displacement (L)",
    y = "Highway fuel economy (mpg)",
    color = "Car type",
    title = "Fuel efficiency generally decreases with engine size",
    subtitle = "Two seaters (sports cars) are an exception because of their light weight",
    caption = "Data from fueleconomy.gov"
  )
```

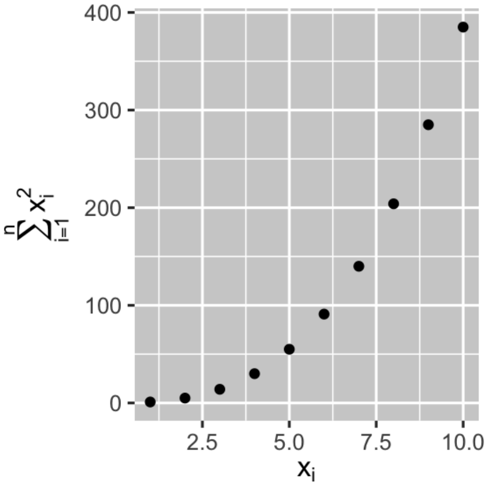


The purpose of a plot title is to summarize the main finding. Avoid titles that just describe what the plot is, e.g., "A scatterplot of engine displacement vs. fuel economy."

If you need to add more text, there are two other useful labels: `subtitle` adds additional detail in a smaller font beneath the title, and `caption` adds text at the bottom right of the plot, often used to describe the source of the data. You can also use `labs()` to replace the axis and legend titles. It's usually a good idea to replace short variable names with more detailed descriptions and to include the units.

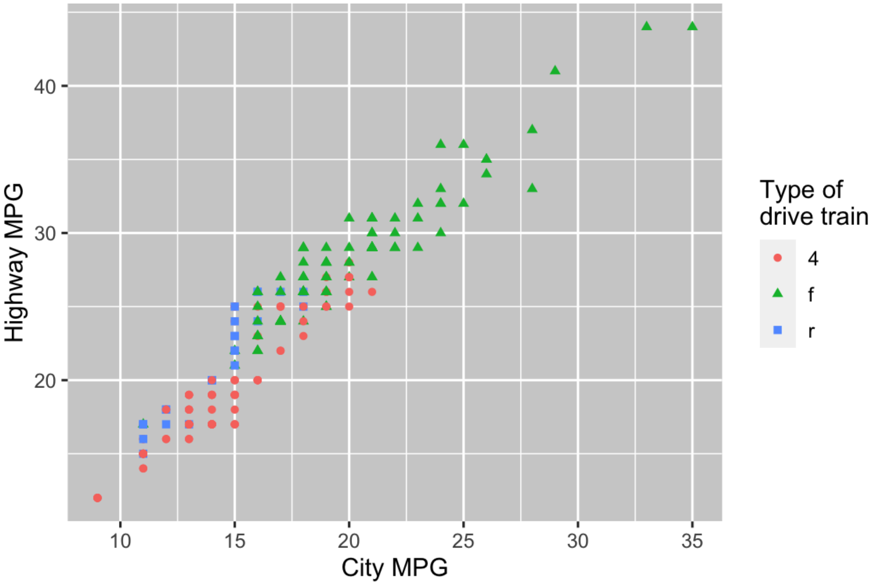
It's possible to use mathematical equations instead of text strings. Just switch `""` out for `quote()` and read about the available options in `?plotmath`:

```
df <- tibble(
  x = 1:10,
  y = cumsum(x^2)
)

ggplot(df, aes(x, y)) +
  geom_point() +
  labs(
    x = quote(x[i]),
    y = quote(sum(x[i] ^ 2, i == 1, n))
  )
```



Exercises

1. Create one plot on the fuel economy data with customized `title`, `subtitle`, `caption`, `x`, `y`, and `color` labels.
2. Re-create the following plot using the fuel economy data. Note that both the colors and shapes of points vary by type of drivetrain.



3. Take an exploratory graphic that you've created in the last month, and add informative titles to make it easier for others to understand.

Annotations

In addition to labeling major components of your plot, it's often useful to label individual observations or groups of observations. The first tool you have at your disposal is `geom_text()`. `geom_text()` is similar to `geom_point()`, but it has an additional aesthetic: `label`. This makes it possible to add textual labels to your plots.

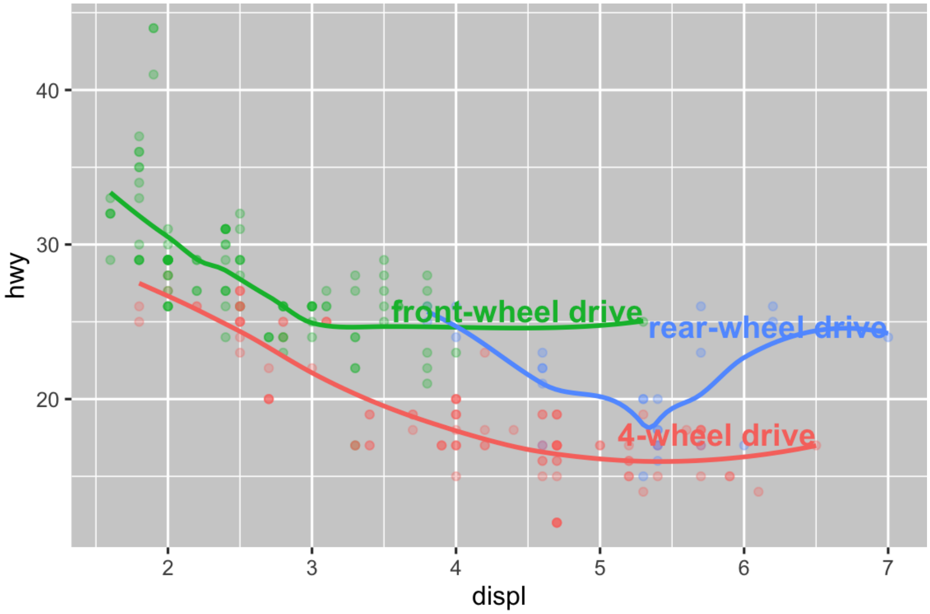
There are two possible sources of labels. First, you might have a tibble that provides labels. In the following plot we pull out the cars with the highest engine size in each drive type and save their information as a new data frame called `label_info`:

```
label_info <- mpg |>
  group_by(drv) |>
  arrange(desc(displ)) |>
  slice_head(n = 1) |>
  mutate(
    drive_type = case_when(
      drv == "f" ~ "front-wheel drive",
      drv == "r" ~ "rear-wheel drive",
      drv == "4" ~ "4-wheel drive"
    )
  ) |>
  select(displ, hwy, drv, drive_type)

label_info
```

```
#> # A tibble: 3 × 4
#> # Groups:   drv [3]
#>   displ   hwy drv   drive_type
#>   <dbl> <int> <chr> <chr>
#> 1   6.5    17 4     4-wheel drive
#> 2   5.3    25 f     front-wheel drive
#> 3   7      24 r     rear-wheel drive
```

Then, we use this new data frame to directly label the three groups to replace the legend with labels placed directly on the plot. Using the `fontface` and `size` arguments we can customize the look of the text labels. They're larger than the rest of the text on the plot and bolded. (`theme(legend.position = "none")` turns all the legends off—we'll talk about it more shortly.)

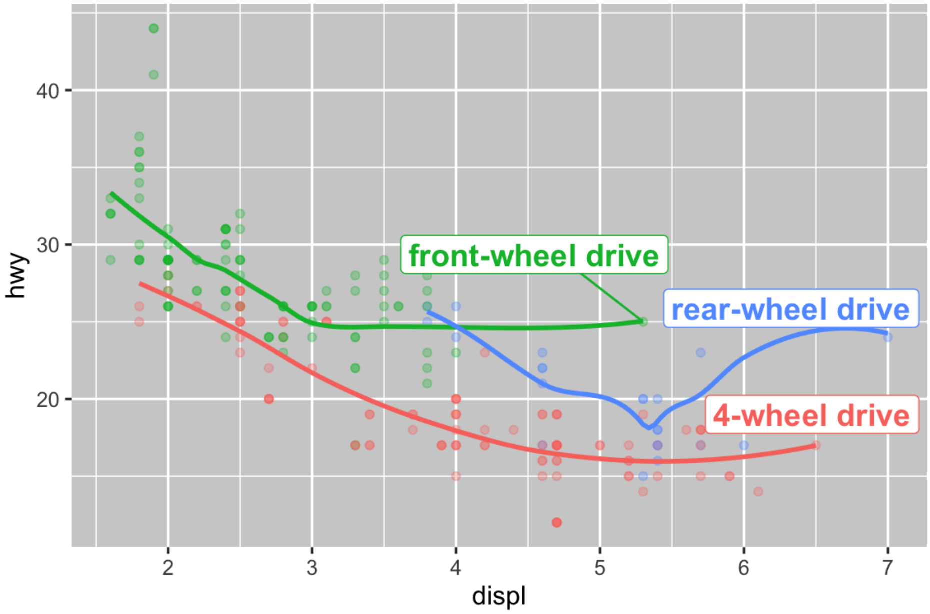
```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point(alpha = 0.3) +
  geom_smooth(se = FALSE) +
  geom_text(
    data = label_info,
    aes(x = displ, y = hwy, label = drive_type),
    fontface = "bold", size = 5, hjust = "right", vjust = "bottom"
  ) +
  theme(legend.position = "none")
#> `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Note the use of `hjust` (horizontal justification) and `vjust` (vertical justification) to control the alignment of the label.

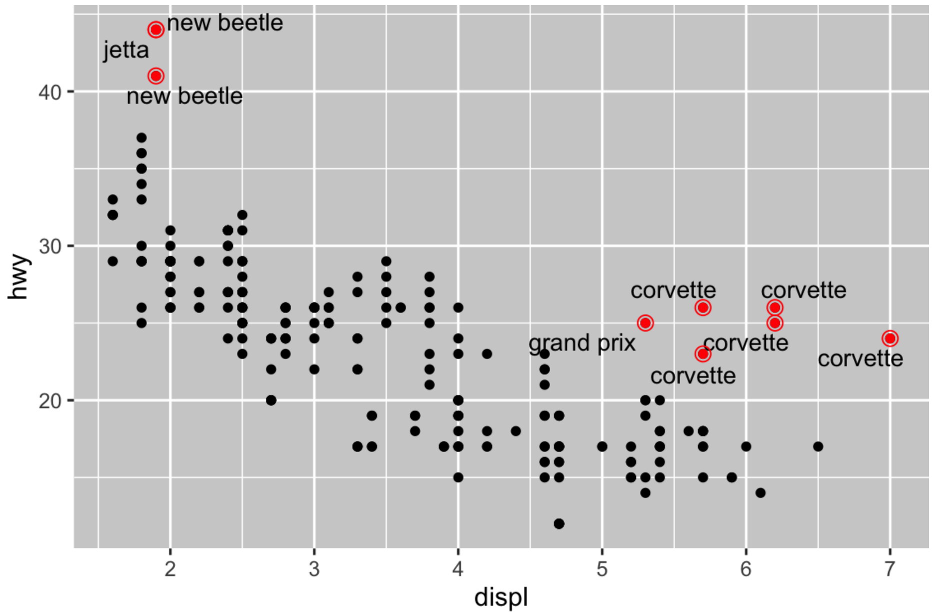
However, the annotated plot we just made is hard to read because the labels overlap with each other and with the points. We can use the `geom_label_repel()` function from the ggrepel package to address both of these issues. This useful package will automatically adjust labels so that they don't overlap:

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point(alpha = 0.3) +
  geom_smooth(se = FALSE) +
  geom_label_repel(
    data = label_info,
    aes(x = displ, y = hwy, label = drive_type),
    fontface = "bold", size = 5, nudge_y = 2
  ) +
  theme(legend.position = "none")
#> `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



You can also use the same idea to highlight certain points on a plot with `geom_text_repel()` from the ggrepel package. Note another handy technique used here: we added a second layer of large, hollow points to further highlight the labeled points.

```
potential_outliers <- mpg |>
  filter(hwy > 40 | (hwy > 20 & displ > 5))

ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_text_repel(data = potential_outliers, aes(label = model)) +
  geom_point(data = potential_outliers, color = "red") +
  geom_point(
    data = potential_outliers,
    color = "red", size = 3, shape = "circle open"
  )
```



Remember, in addition to `geom_text()` and `geom_label()`, you have many other geoms in ggplot2 available to help annotate your plot. A couple ideas:

- Use `geom_hline()` and `geom_vline()` to add reference lines. We often make them thick (`linewidth = 2`) and white (`color = white`) and draw them underneath the primary data layer. That makes them easy to see, without drawing attention away from the data.
- Use `geom_rect()` to draw a rectangle around points of interest. The boundaries of the rectangle are defined by aesthetics `xmin`, `xmax`, `ymin`, and `ymax`. Alternatively, look into the ggforce package, specifically `geom_mark_hull()`, which allows you to annotate subsets of points with hulls.
- Use `geom_segment()` with the `arrow` argument to draw attention to a point with an arrow. Use aesthetics `x` and `y` to define the starting location, and use `xend` and `yend` to define the end location.

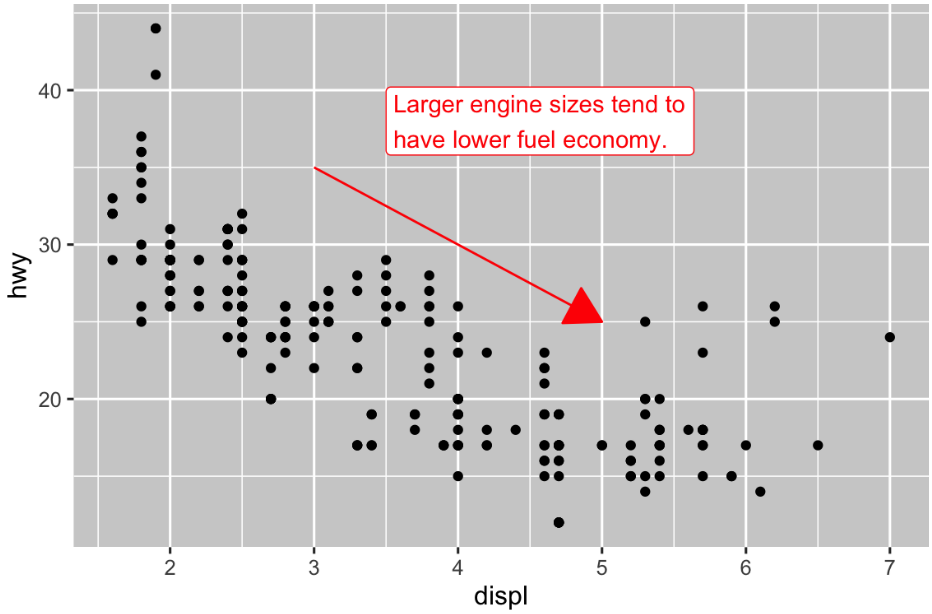
Another handy function for adding annotations to plots is `annotate()`. As a rule of thumb, geoms are generally useful for highlighting a subset of the data, while `annotate()` is useful for adding one or a few annotation elements to a plot.

To demonstrate using `annotate()`, let's create some text to add to our plot. The text is a bit long, so we'll use `stringr::str_wrap()` to automatically add line breaks to it given the number of characters you want per line:

```
trend_text <- "Larger engine sizes tend to\nhave lower fuel economy." |>
  str_wrap(width = 30)
trend_text
#> [1] "Larger engine sizes tend to\nhave lower fuel economy."
```

Then, we add two layers of annotation: one with a label geom and the other with a segment geom. The `x` and `y` aesthetics in both define where the annotation should start, and the `xend` and `yend` aesthetics in the segment annotation define the starting location of the end location of the segment. Note also that the segment is styled as an arrow.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  annotate(
    geom = "label", x = 3.5, y = 38,
    label = trend_text,
    hjust = "left", color = "red"
  ) +
  annotate(
    geom = "segment",
    x = 3, y = 35, xend = 5, yend = 25, color = "red",
    arrow = arrow(type = "closed")
  )
```



Annotation is a powerful tool for communicating main takeaways and interesting features of your visualizations. The only limit is your imagination (and your patience with positioning annotations to be aesthetically pleasing)!

Exercises

1. Use `geom_text()` with infinite positions to place text at the four corners of the plot.
2. Use `annotate()` to add a point geom in the middle of your last plot without having to create a tibble. Customize the shape, size, or color of the point.
3. How do labels with `geom_text()` interact with faceting? How can you add a label to a single facet? How can you put a different label in each facet? (Hint: Think about the dataset that is being passed to `geom_text()`.)
4. What arguments to `geom_label()` control the appearance of the background box?
5. What are the four arguments to `arrow()`? How do they work? Create a series of plots that demonstrate the most important options.

Scales

The third way you can make your plot better for communication is to adjust the scales. Scales control how the aesthetic mappings manifest visually.

Default Scales

Normally, ggplot2 automatically adds scales for you. For example, when you type:

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

ggplot2 automatically adds default scales behind the scenes:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_color_discrete()
```

Note the naming scheme for scales: `scale_` followed by the name of the aesthetic, then `_`, and then the name of the scale. The default scales are named according to the type of variable they align with: continuous, discrete, date-time, or date. `scale_x_continuous()` puts the numeric values from `displ` on a continuous number line on the x-axis, `scale_color_discrete()` chooses colors for each `class` of car, etc. There are lots of nondefault scales, which you'll learn about next.

The default scales have been carefully chosen to do a good job for a wide range of inputs. Nevertheless, you might want to override the defaults for two reasons:

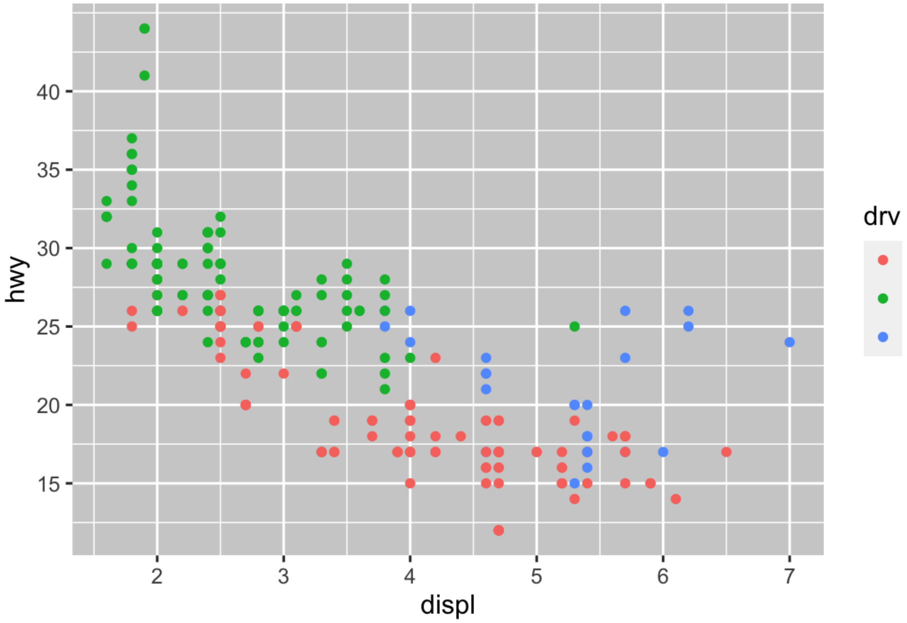
- You might want to tweak some of the parameters of the default scale. This allows you to do things like change the breaks on the axes, or the key labels on the legend.
- You might want to replace the scale altogether and use a completely different algorithm. Often you can do better than the default because you know more about the data.

Axis Ticks and Legend Keys

Collectively axes and legends are called *guides*. Axes are used for `x` and `y` aesthetics; legends are used for everything else.

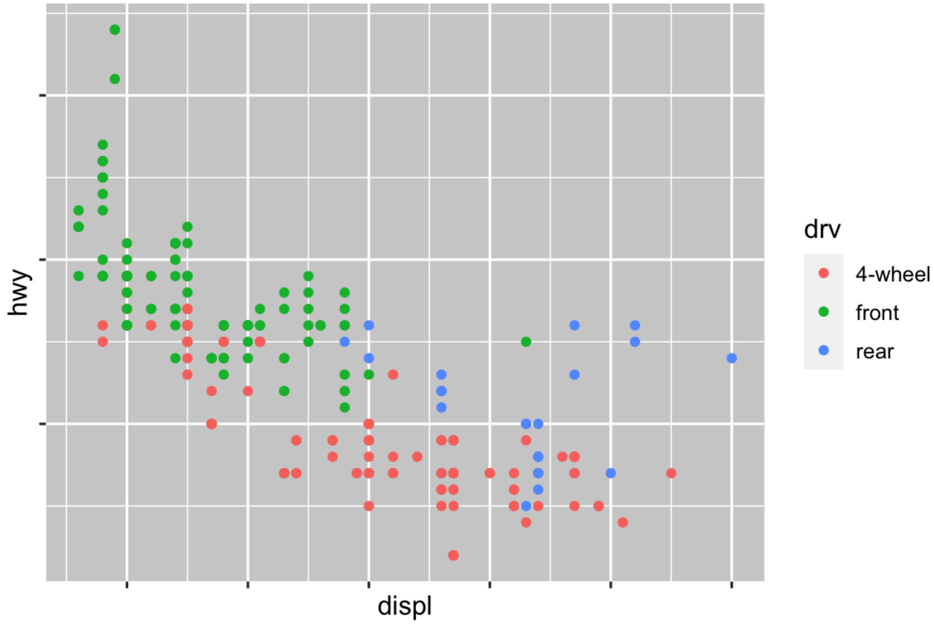
There are two primary arguments that affect the appearance of the ticks on the axes and the keys on the legend: `breaks` and `labels`. The `breaks` argument controls the position of the ticks or the values associated with the keys. The `labels` argument controls the text label associated with each tick/key. The most common use of `breaks` is to override the default choice:

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  scale_y_continuous(breaks = seq(15, 40, by = 5))
```



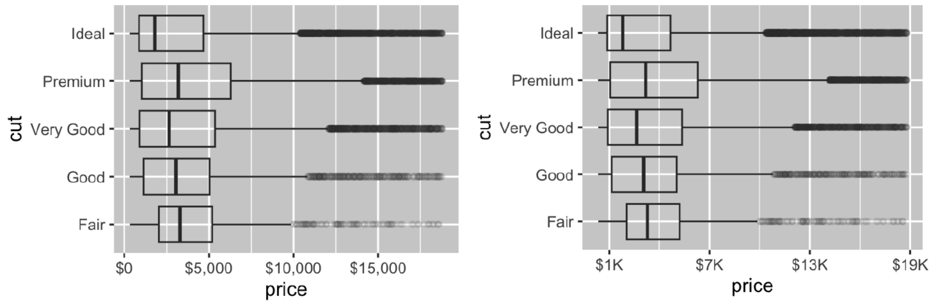
You can use `labels` in the same way (a character vector the same length as `breaks`), but you can also set it to `NULL` to suppress the labels altogether. This can be useful for maps or for publishing plots where you can't share the absolute numbers. You can also use `breaks` and `labels` to control the appearance of legends. For discrete scales for categorical variables, `labels` can be a named list of the existing levels names and the desired labels for them.

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  scale_x_continuous(labels = NULL) +
  scale_y_continuous(labels = NULL) +
  scale_color_discrete(labels = c("4" = "4-wheel", "f" = "front", "r" = "rear"))
```



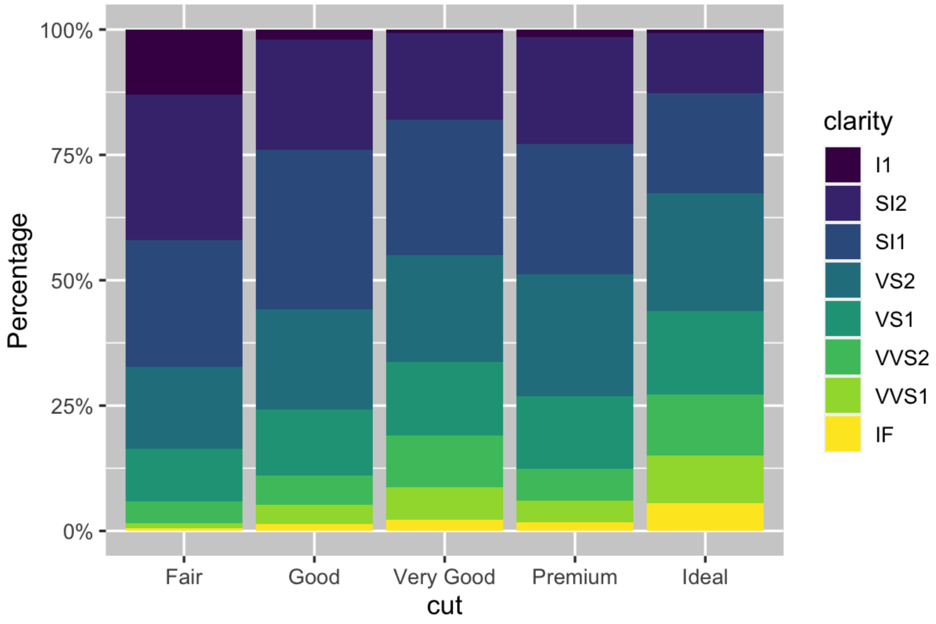
The `labels` argument coupled with labeling functions from the scales package is also useful for formatting numbers as currency, percent, etc. The plot on the left shows default labeling with `label_dollar()`, which adds a dollar sign as well as a thousand separator comma. The plot on the right adds further customization by dividing dollar values by 1,000 and adding a suffix "K" (for "thousands") as well as adding custom breaks. Note that `breaks` is in the original scale of the data.

```
# Left
ggplot(diamonds, aes(x = price, y = cut)) +
  geom_boxplot(alpha = 0.05) +
  scale_x_continuous(labels = label_dollar())

# Right
ggplot(diamonds, aes(x = price, y = cut)) +
  geom_boxplot(alpha = 0.05) +
  scale_x_continuous(
    labels = label_dollar(scale = 1/1000, suffix = "K"),
    breaks = seq(1000, 19000, by = 6000)
  )
```



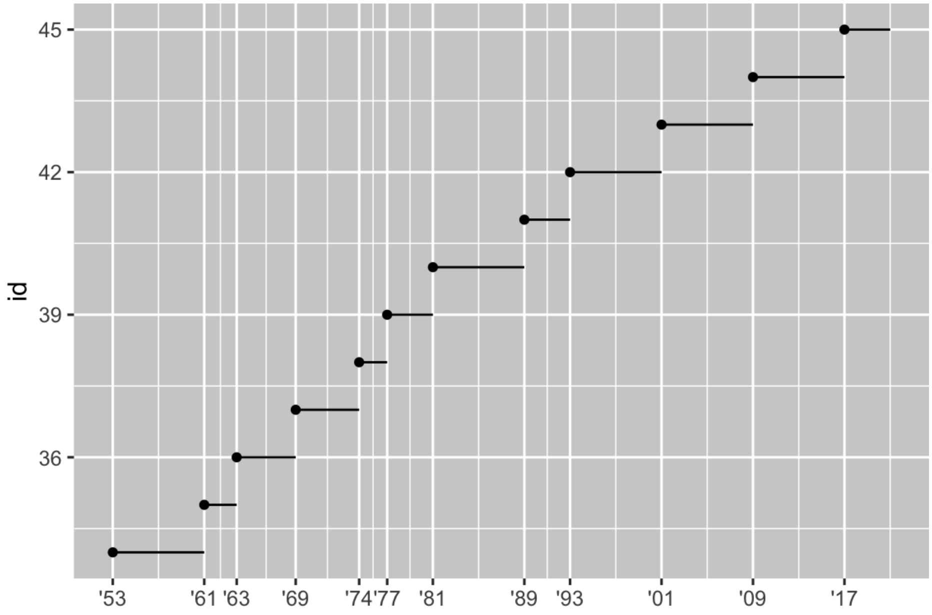
Another handy label function is `label_percent()`:

```
ggplot(diamonds, aes(x = cut, fill = clarity)) +
  geom_bar(position = "fill") +
  scale_y_continuous(name = "Percentage", labels = label_percent())
```



Another use of `breaks` is when you have relatively few data points and want to highlight exactly where the observations occur. For example, take this plot that shows when each US president started and ended their term:

```
presidential |>
  mutate(id = 33 + row_number()) |>
  ggplot(aes(x = start, y = id)) +
  geom_point() +
  geom_segment(aes(xend = end, yend = id)) +
  scale_x_date(name = NULL, breaks = presidential$start, date_labels = "'%y")
```



Note that for the `breaks` argument we pulled out the `start` variable as a vector with `presidential$start` because we can't do an aesthetic mapping for this argument. Also note that the specification of breaks and labels for date and date-time scales is a little different:

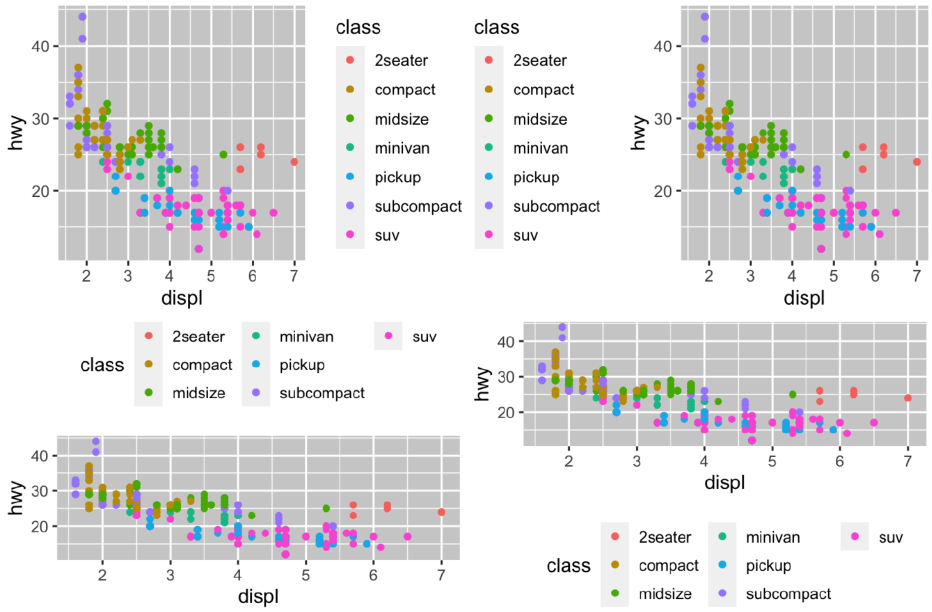
- `date_labels` takes a format specification, in the same form as `parse_datetime()`.
- `date_breaks` (not shown here) takes a string like "2 days" or "1 month."

Legend Layout

You will most often use `breaks` and `labels` to tweak the axes. While they both also work for legends, there are a few other techniques you are more likely to use.

To control the overall position of the legend, you need to use a `theme()` setting. We'll come back to themes at the end of the chapter, but in brief, they control the nondata parts of the plot. The theme setting `legend.position` controls where the legend is drawn:

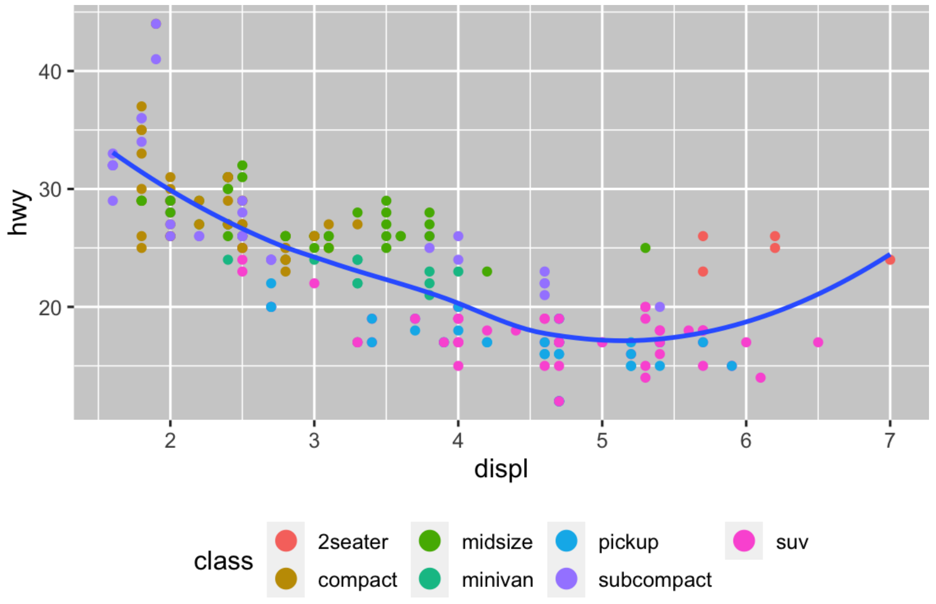
```
base <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class))

base + theme(legend.position = "right") # the default
base + theme(legend.position = "left")
base +
  theme(legend.position = "top") +
  guides(col = guide_legend(nrow = 3))
base +
  theme(legend.position = "bottom") +
  guides(col = guide_legend(nrow = 3))
```



If your plot is short and wide, place the legend at the top or bottom, and if it's tall and narrow, place the legend at the left or right. You can also use `legend.position = "none"` to suppress the display of the legend altogether.

To control the display of individual legends, use `guides()` along with `guide_legend()` or `guide_colorbar()`. The following example shows two important settings: controlling the number of rows the legend uses with `nrow`, and overriding one of the aesthetics to make the points bigger. This is particularly useful if you have used a low `alpha` to display many points on a plot.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme(legend.position = "bottom") +
  guides(color = guide_legend(nrow = 2, override.aes = list(size = 4)))
#> `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



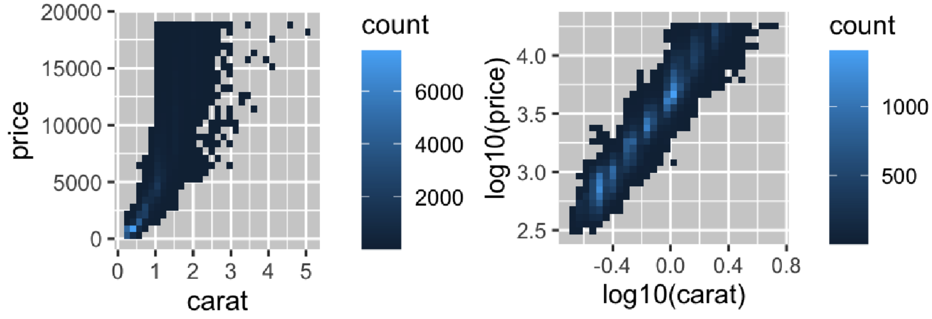
Note that the name of the argument in `guides()` matches the name of the aesthetic, just like in `labs()`.

Replacing a Scale

Instead of just tweaking the details a little, you can instead replace the scale altogether. There are two types of scales you're most likely to want to switch out: continuous position scales and color scales. Fortunately, the same principles apply to all the other aesthetics, so once you've mastered position and color, you'll be able to quickly pick up other scale replacements.

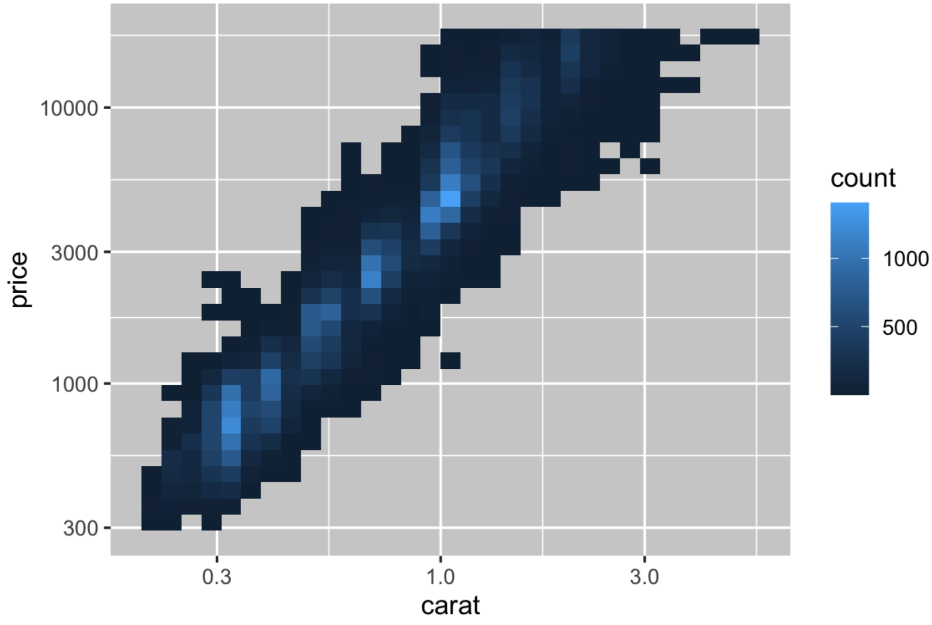
It's useful to plot transformations of your variable. For example, it's easier to see the precise relationship between `carat` and `price` if we log transform them:

```
# Left
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_bin2d()

# Right
ggplot(diamonds, aes(x = log10(carat), y = log10(price))) +
  geom_bin2d()
```



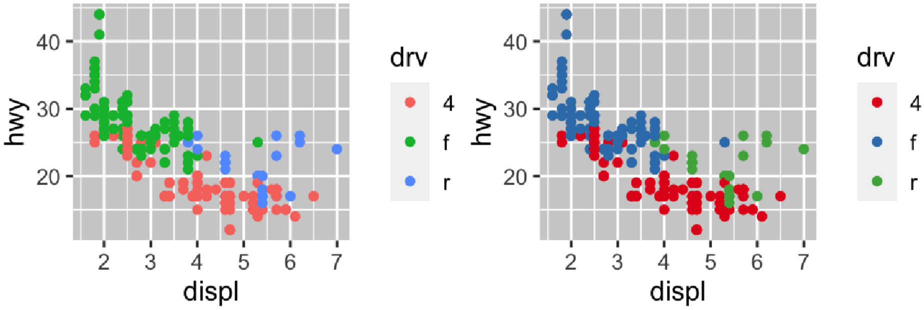
However, the disadvantage of this transformation is that the axes are now labeled with the transformed values, making it hard to interpret the plot. Instead of doing the transformation in the aesthetic mapping, we can instead do it with the scale. This is visually identical, except the axes are labeled on the original data scale.

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_bin2d() +
  scale_x_log10() +
  scale_y_log10()
```



Another scale that is frequently customized is color. The default categorical scale picks colors that are evenly spaced around the color wheel. Useful alternatives are the ColorBrewer scales, which have been hand tuned to work better for people with common types of color blindness. The following two plots look similar, but there is enough difference in the shades of red and green that the dots on the right can be distinguished even by people with red-green color blindness.[1]

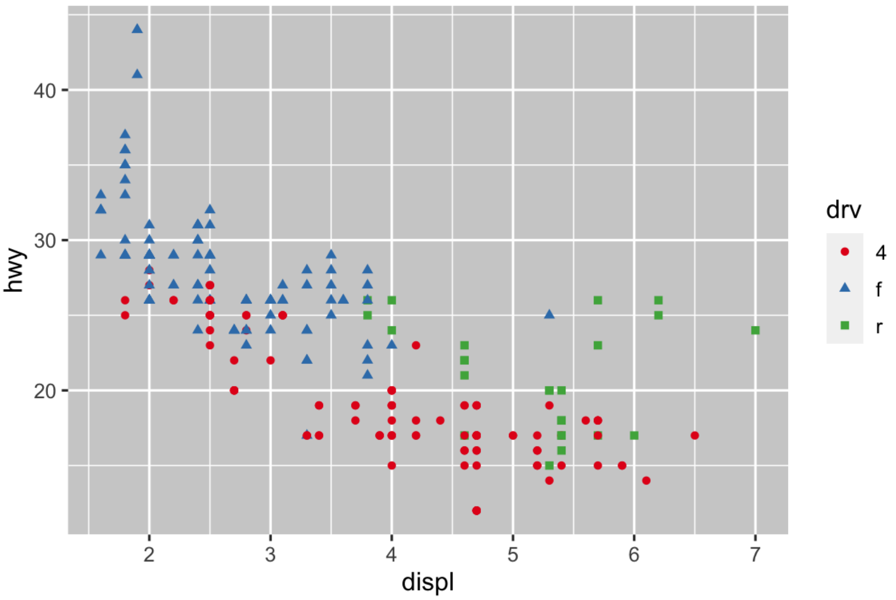
```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv))

ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv)) +
  scale_color_brewer(palette = "Set1")
```



Don't forget simpler techniques for improving accessibility. If there are just a few colors, you can add a redundant shape mapping. This will also help ensure your plot is interpretable in black and white.

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv, shape = drv)) +
  scale_color_brewer(palette = "Set1")
```

[1] You can use a tool like SimDaltonism to simulate color blindness to test these images.



The ColorBrewer scales are documented online and made available in R via the RColorBrewer package, by Erich Neuwirth. Figure 11-1 shows the complete list of all palettes. The sequential (top) and diverging (bottom) palettes are particularly useful if your categorical values are ordered or have a "middle." This often arises if you've used `cut()` to make a continuous variable into a categorical variable.

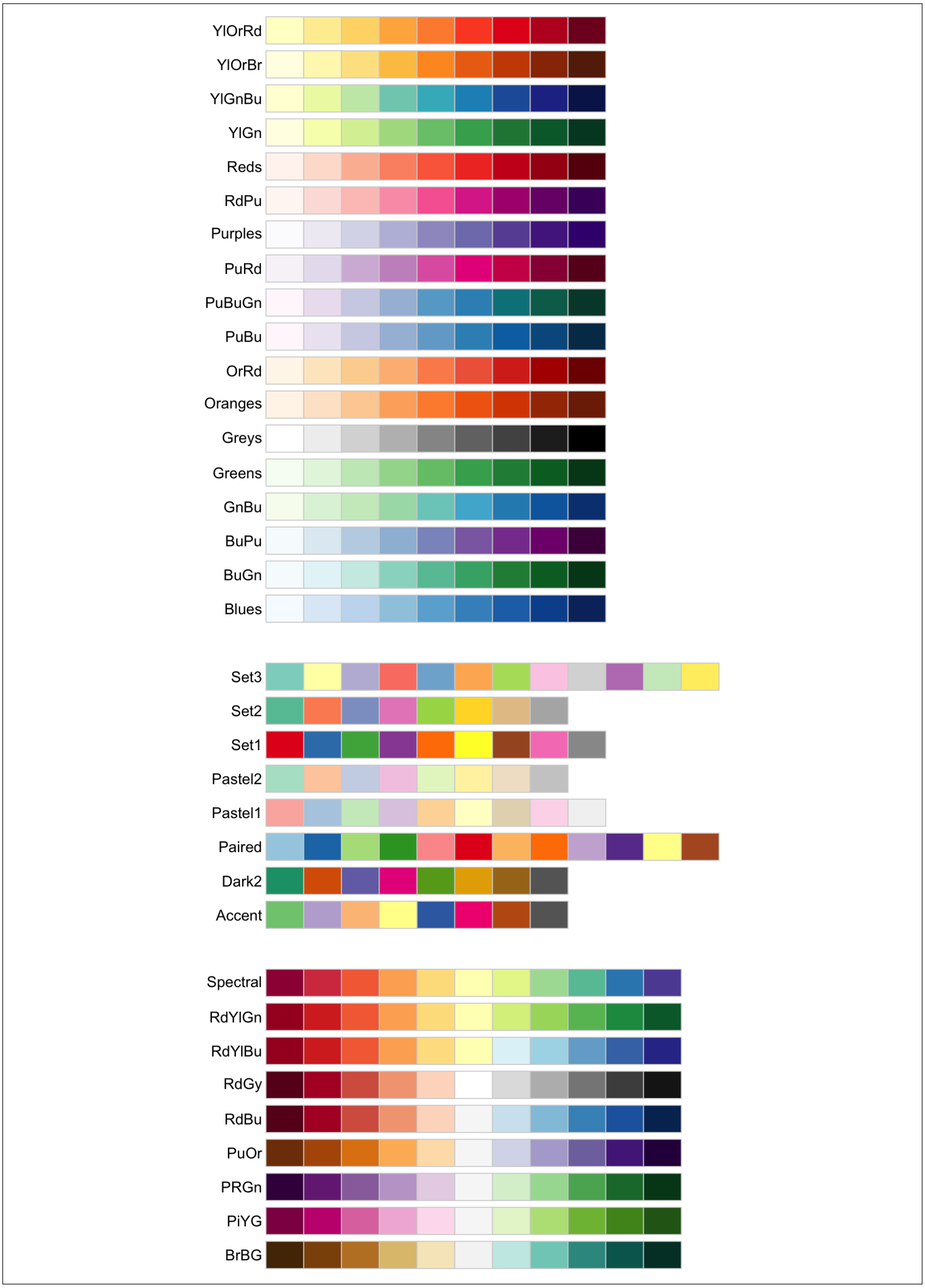
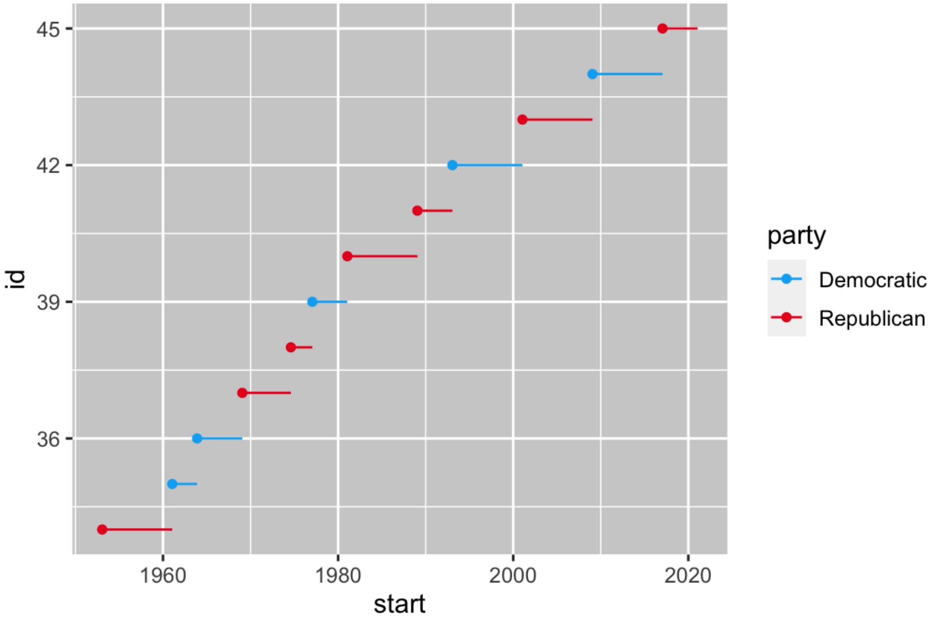


Figure 11-1. All ColorBrewer scales.

When you have a predefined mapping between values and colors, use scale_color_manual(). For example, if we map presidential party to color, we want to use the standard mapping of red for Republicans and blue for Democrats. One approach for assigning these colors is using hex color codes:

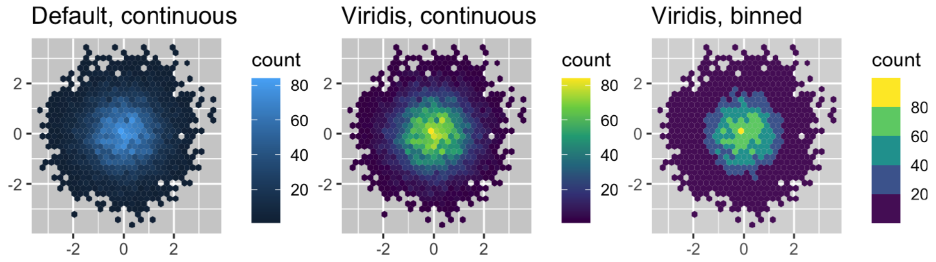
```
presidential |>
  mutate(id = 33 + row_number()) |>
  ggplot(aes(x = start, y = id, color = party)) +
  geom_point() +
  geom_segment(aes(xend = end, yend = id)) +
  scale_color_manual(values = c(Republican = "#E81B23", Democratic = "#00AEF3"))
```



For continuous color, you can use the built-in scale_color_gradient() or scale_fill_gradient(). If you have a diverging scale, you can use scale_color_gradient2(). That allows you to give, for example, positive and negative values different colors. That's sometimes also useful if you want to distinguish points above or below the mean.

Another option is to use the viridis color scales. The designers, Nathaniel Smith and Stéfan van der Walt, carefully tailored continuous color schemes that are perceptible to people with various forms of color blindness as well as perceptually uniform in both color and black and white. These scales are available as continuous (c), discrete (d), and binned (b) palettes in ggplot2.

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)

ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed() +
  labs(title = "Default, continuous", x = NULL, y = NULL)

ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed() +
  scale_fill_viridis_c() +
  labs(title = "Viridis, continuous", x = NULL, y = NULL)

ggplot(df, aes(x, y)) +
  geom_hex() +
  coord_fixed() +
  scale_fill_viridis_b() +
  labs(title = "Viridis, binned", x = NULL, y = NULL)
```



Note that all color scales come in two varieties: `scale_color_*()` and `scale_fill_*()` for the `color` and `fill` aesthetics, respectively (the color scales are available in both UK and US spellings).

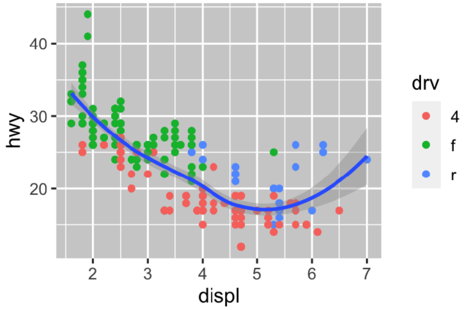
Zooming

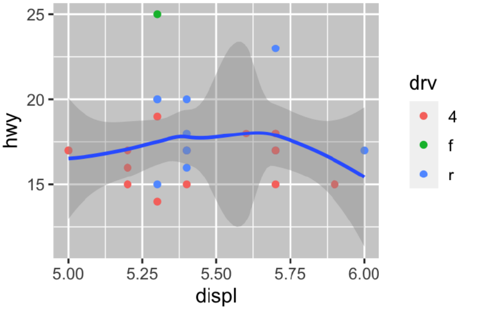
There are three ways to control the plot limits:

- Adjusting what data are plotted
- Setting the limits in each scale
- Setting `xlim` and `ylim` in `coord_cartesian()`

We'll demonstrate these options in a series of plots. The plot on the left shows the relationship between engine size and fuel efficiency, colored by type of drivetrain. The plot on the right shows the same variables but subsets the data plotted. Subsetting the data has affected the x and y scales as well as the smooth curve.

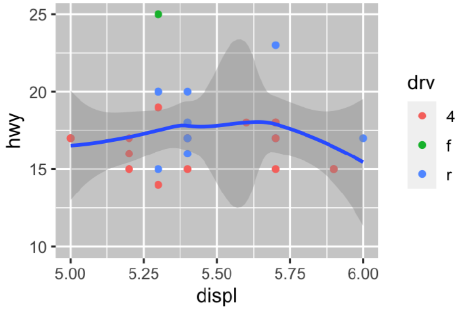
```
# Left
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth()

# Right
mpg |>
  filter(displ >= 5 & displ <= 6 & hwy >= 10 & hwy <= 25) |>
  ggplot(aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth()
```

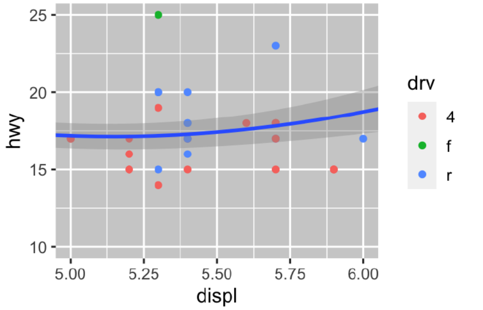




Let's compare these to the two following plots where the plot on the left sets the `limits` on individual scales and the plot on the right sets them in `coord_cartesian()`. We can see that reducing the limits is equivalent to subsetting the data. Therefore, to zoom in on a region of the plot, it's generally best to use `coord_cartesian()`.

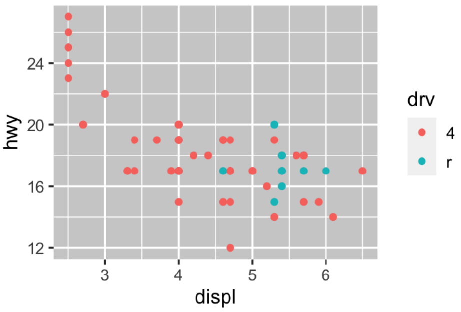
```
# Left
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth() +
  scale_x_continuous(limits = c(5, 6)) +
  scale_y_continuous(limits = c(10, 25))

# Right
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = drv)) +
  geom_smooth() +
  coord_cartesian(xlim = c(5, 6), ylim = c(10, 25))
```

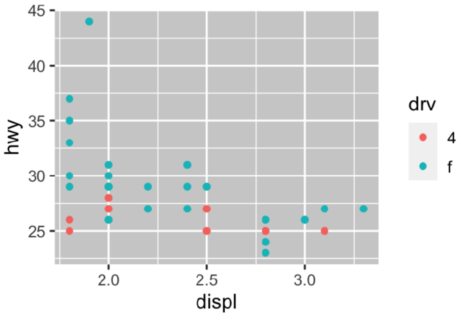




On the other hand, setting the `limits` on individual scales is generally more useful if you want to *expand* the limits, e.g., to match scales across different plots. For example, if we extract two classes of cars and plot them separately, it's difficult to compare the plots because all three scales (the x-axis, the y-axis, and the color aesthetic) have different ranges.

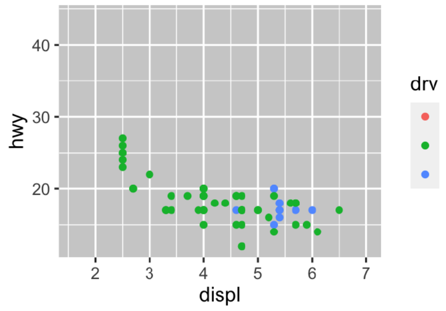
```
suv <- mpg |> filter(class == "suv")
compact <- mpg |> filter(class == "compact")

# Left
ggplot(suv, aes(x = displ, y = hwy, color = drv)) +
  geom_point()

# Right
ggplot(compact, aes(x = displ, y = hwy, color = drv)) +
  geom_point()
```

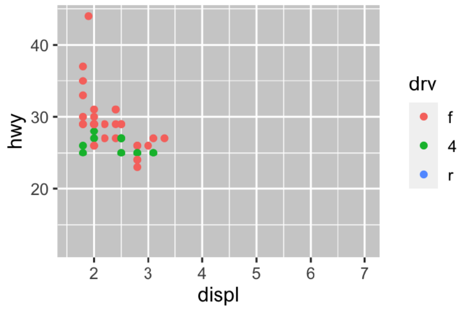




One way to overcome this problem is to share scales across multiple plots, training the scales with the `limits` of the full data.

```
x_scale <- scale_x_continuous(limits = range(mpg$displ))
y_scale <- scale_y_continuous(limits = range(mpg$hwy))
col_scale <- scale_color_discrete(limits = unique(mpg$drv))

# Left
ggplot(suv, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale

# Right
ggplot(compact, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  x_scale +
  y_scale +
  col_scale
```





In this particular case, you could have simply used faceting, but this technique is useful more generally, if, for instance, you want to spread plots over multiple pages of a report.

Exercises

1. Why doesn't the following code override the default scale?

```
df <- tibble(
  x = rnorm(10000),
  y = rnorm(10000)
)

ggplot(df, aes(x, y)) +
  geom_hex() +
  scale_color_gradient(low = "white", high = "red") +
  coord_fixed()
```

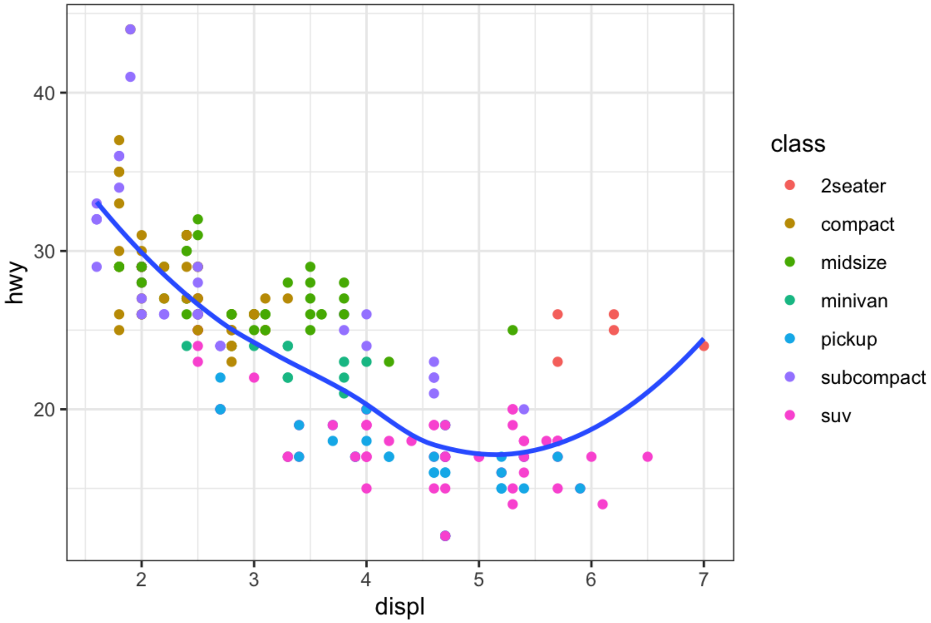
2. What is the first argument to every scale? How does it compare to `labs()`?
3. Change the display of the presidential terms by:
    a. Combining the two variants that customize colors and x-axis breaks
    b. Improving the display of the y-axis
    c. Labeling each term with the name of the president
    d. Adding informative plot labels
    e. Placing breaks every four years (this is trickier than it seems!)
4. First, create the following plot. Then, modify the code using `override.aes` to make the legend easier to see.

```
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point(aes(color = cut), alpha = 1/20)
```

Themes

Finally, you can customize the nondata elements of your plot with a theme:

```
ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point(aes(color = class)) +
  geom_smooth(se = FALSE) +
  theme_bw()
```



ggplot2 includes the eight themes shown in Figure 11-2, with `theme_gray()` as the default.[2] Many more are included in add-on packages like ggthemes, by Jeffrey Arnold. You can also create your own themes, if you are trying to match a particular corporate or journal style.

[2] Many people wonder why the default theme has a gray background. This was a deliberate choice because it puts the data forward while still making the grid lines visible. The white grid lines are visible (which is important because they significantly aid position judgments), but they have little visual impact, and we can easily tune them out. The gray background gives the plot a similar typographic color to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background. Finally, the gray background creates a continuous field of color, which ensures that the plot is perceived as a single visual entity.

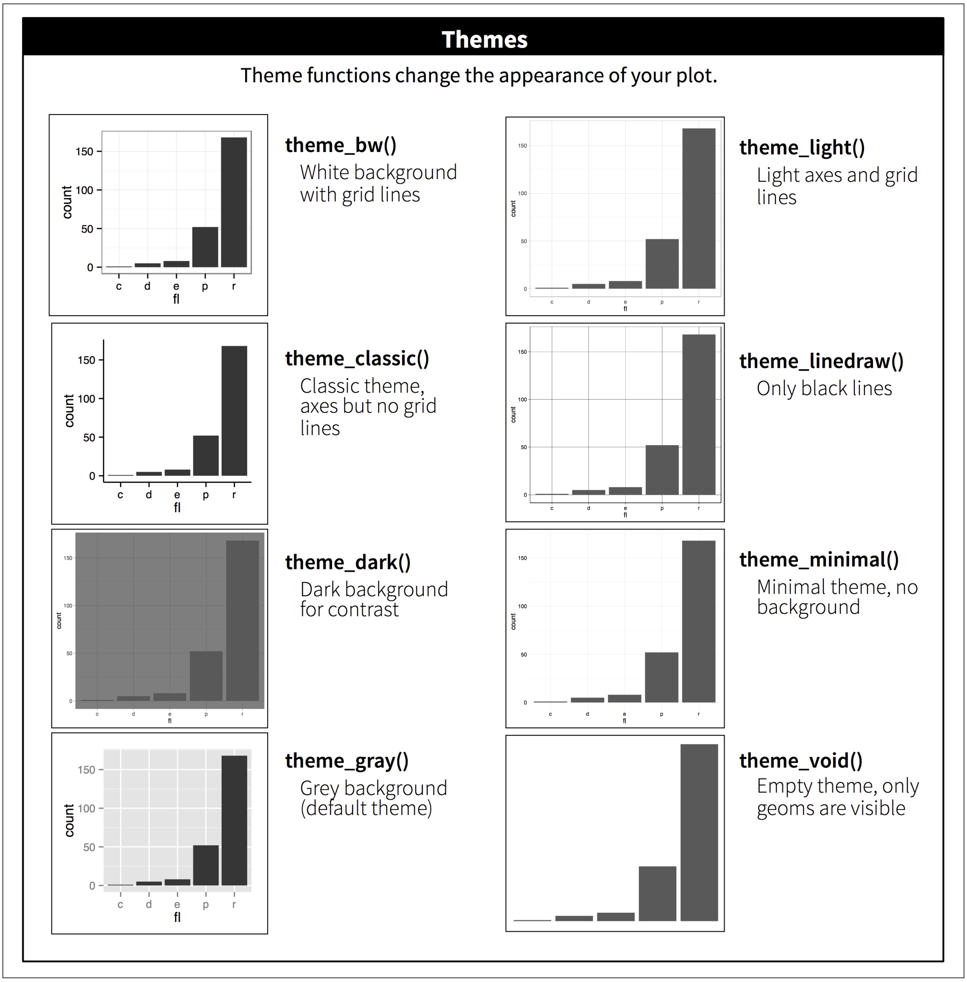
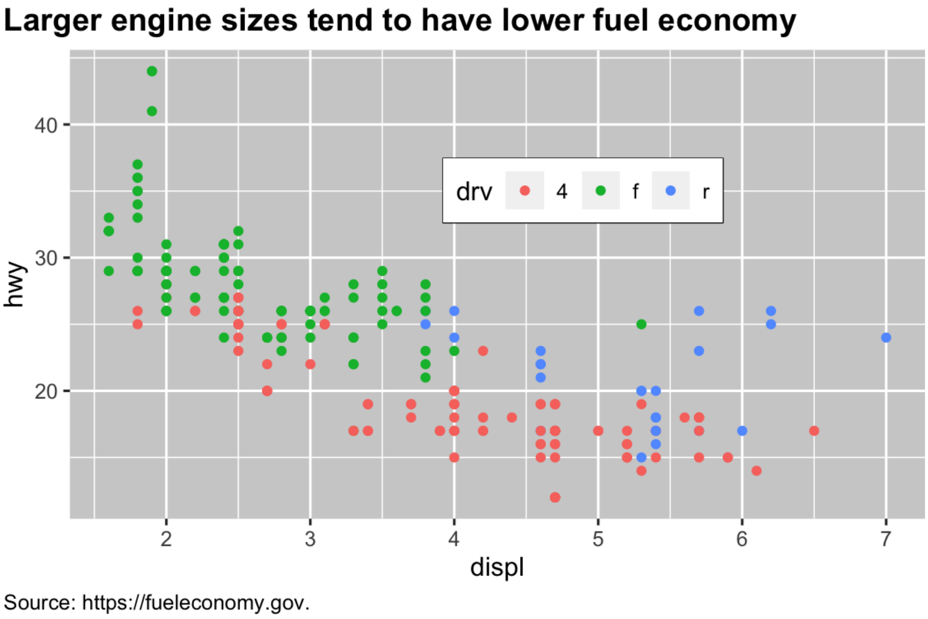


Figure 11-2. The eight themes built in to ggplot2.

It's also possible to control individual components of each theme, such as the size and color of the font used for the y-axis. We've already seen that `legend.position` controls where the legend is drawn. There are many other aspects of the legend that can be customized with `theme()`. For example, in the following plot we change the direction of the legend as well as put a black border around it. Note that customization of the legend box and plot title elements of the theme are done with `element_*()` functions. These functions specify the styling of nondata components; e.g., the title text is bolded in the `face` argument of `element_text()`, and the legend border color is defined in the `color` argument of `element_rect()`. The theme elements that control the position of the title and the caption are `plot.title.position` and `plot.caption.position`, respectively. In the following plot these are set to `"plot"` to indicate these elements are aligned to the entire plot area, instead of the plot panel (the default). A few other helpful `theme()` components are used to change the placement for formatting the title and caption text.

```
ggplot(mpg, aes(x = displ, y = hwy, color = drv)) +
  geom_point() +
  labs(
    title = "Larger engine sizes tend to have lower fuel economy",
    caption = "Source: https://fueleconomy.gov."
  ) +
  theme(
    legend.position = c(0.6, 0.7),
    legend.direction = "horizontal",
    legend.box.background = element_rect(color = "black"),
    plot.title = element_text(face = "bold"),
    plot.title.position = "plot",
    plot.caption.position = "plot",
    plot.caption = element_text(hjust = 0)
  )
```


Larger engine sizes tend to have lower fuel economy

Source: https://fueleconomy.gov.

For an overview of all `theme()` components, see the help with `?theme`. The ggplot2 book is also a great place to go for the full details on theming.

Exercises

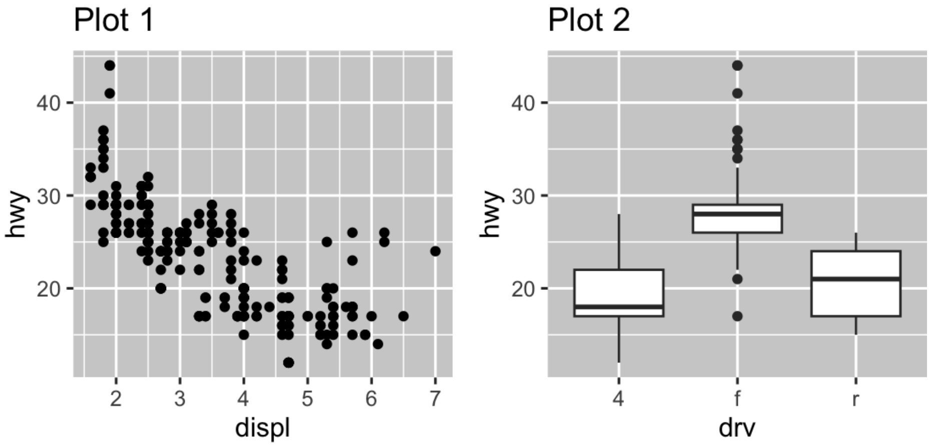
1. Pick a theme offered by the ggthemes package and apply it to the last plot you made.
2. Make the axis labels of your plot blue and bold.

Layout

So far we talked about how to create and modify a single plot. What if you have multiple plots you want to lay out in a certain way? The patchwork package allows you to combine separate plots into the same graphic. We loaded this package earlier in the chapter.

To place two plots next to each other, you can simply add them to each other. Note that you first need to create the plots and save them as objects (in the following example they're called `p1` and `p2`). Then, you place them next to each other with `+`.

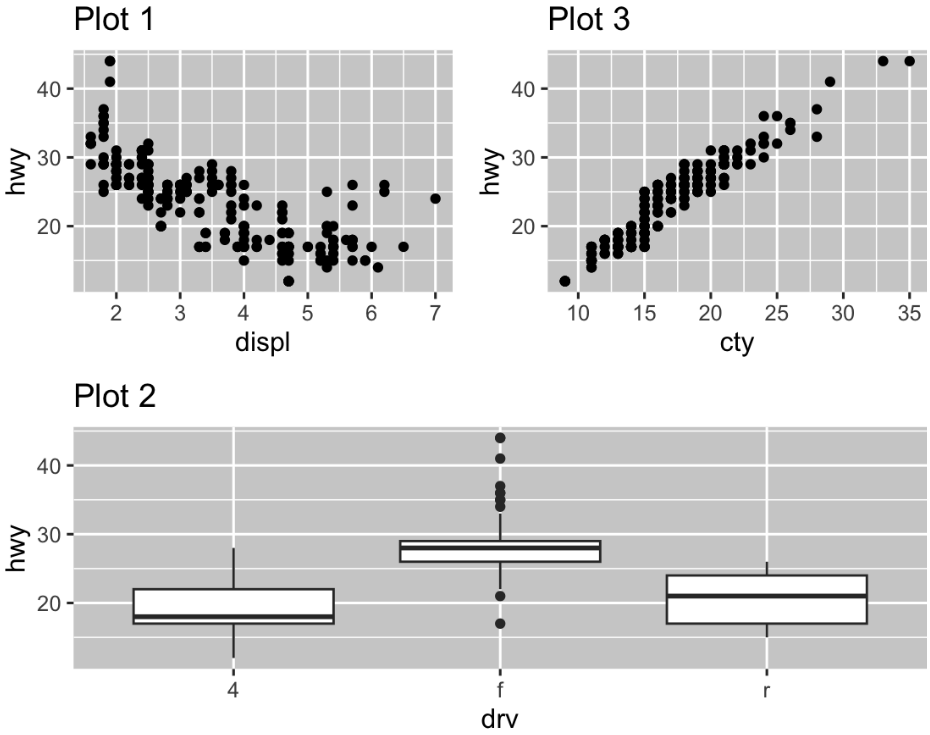
```
p1 <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  labs(title = "Plot 1")
p2 <- ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_boxplot() +
  labs(title = "Plot 2")
p1 + p2
```



It's important to note that in the previous code chunk we did not use a new function from the patchwork package. Instead, the package added a new functionality to the `+` operator.

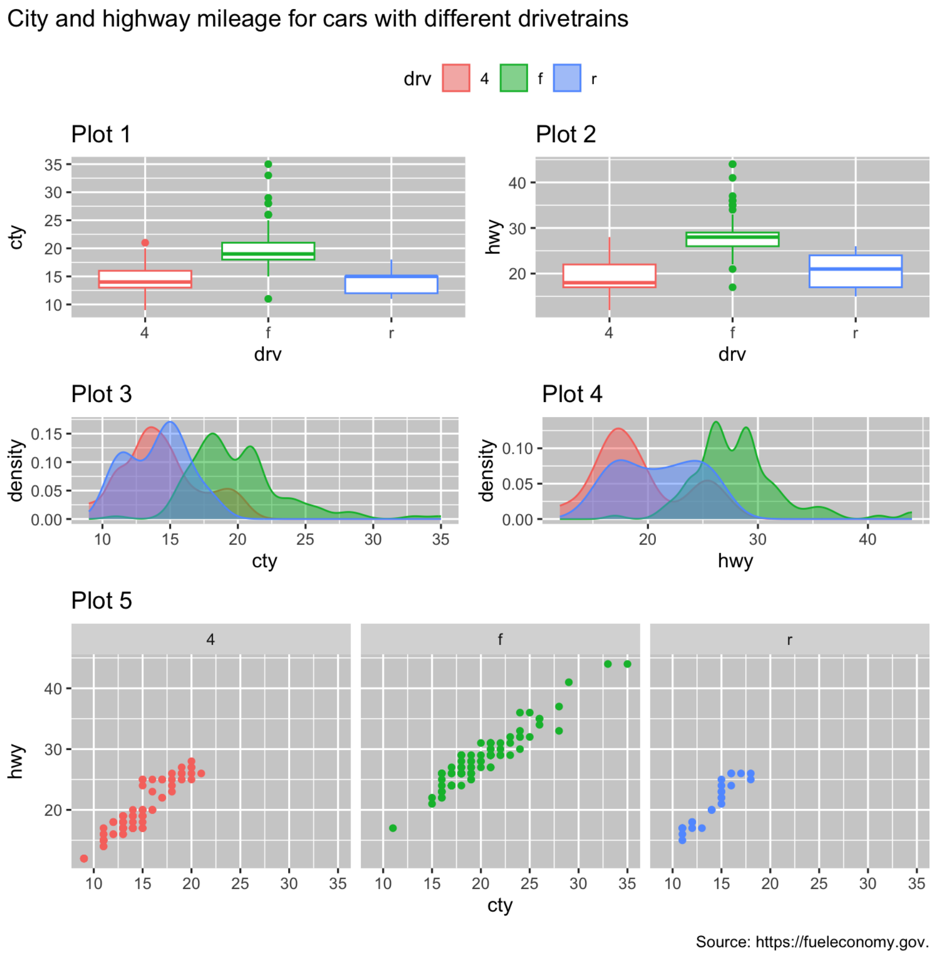
You can also create complex plot layouts with patchwork. In the following, | places the p1 and p3 next to each other, and / moves p2 to the next line:

```
p3 <- ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point() +
  labs(title = "Plot 3")
(p1 | p3) / p2
```



Additionally, patchwork allows you to collect legends from multiple plots into one common legend, customize the placement of the legend as well as dimensions of the plots, and add a common title, subtitle, caption, etc., to your plots. Here we created five plots. We turned off the legends on the box plots and the scatterplot and collected the legends for the density plots at the top of the plot with `& theme(legend.position = "top")`. Note the use of the `&` operator here instead of the usual `+`. This is because we're modifying the theme for the patchwork plot as opposed to the individual ggplots. The legend is placed on top, inside the `guide_area()`. Finally, we have also customized the heights of the various components of our patchwork—the guide has a height of 1, the box plots 3, the density plots 2, and the faceted scatterplot 4. Patchwork divides up the area you have allotted for your plot using this scale and places the components accordingly.

```
p1 <- ggplot(mpg, aes(x = drv, y = cty, color = drv)) +
  geom_boxplot(show.legend = FALSE) +
  labs(title = "Plot 1")

p2 <- ggplot(mpg, aes(x = drv, y = hwy, color = drv)) +
  geom_boxplot(show.legend = FALSE) +
  labs(title = "Plot 2")

p3 <- ggplot(mpg, aes(x = cty, color = drv, fill = drv)) +
  geom_density(alpha = 0.5) +
  labs(title = "Plot 3")

p4 <- ggplot(mpg, aes(x = hwy, color = drv, fill = drv)) +
  geom_density(alpha = 0.5) +
  labs(title = "Plot 4")

p5 <- ggplot(mpg, aes(x = cty, y = hwy, color = drv)) +
  geom_point(show.legend = FALSE) +
  facet_wrap(~drv) +
  labs(title = "Plot 5")

(guide_area() / (p1 + p2) / (p3 + p4) / p5) +
  plot_annotation(
    title = "City and highway mileage for cars with different drivetrains",
    caption = "Source: https://fueleconomy.gov."
  ) +
  plot_layout(
    guides = "collect",
    heights = c(1, 3, 2, 4)
    ) &
  theme(legend.position = "top")
```



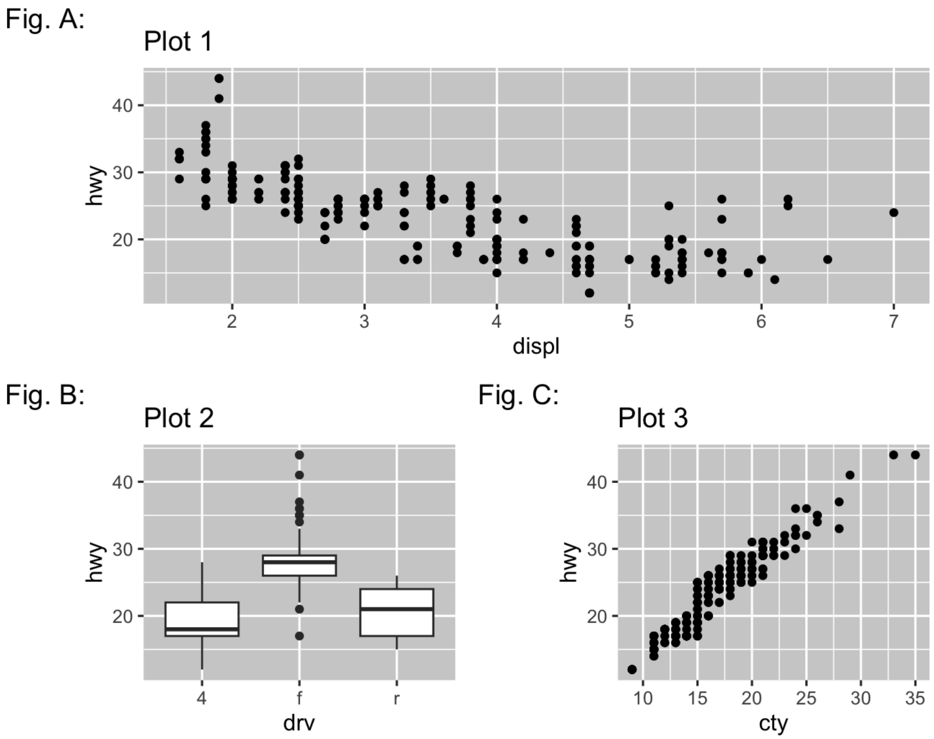
If you'd like to learn more about combining and laying out multiple plots with patchwork, we recommend looking through the guides on the package website.

Exercises

1. What happens if you omit the parentheses in the following plot layout. Can you explain why this happens?

```
p1 <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point() +
  labs(title = "Plot 1")
p2 <- ggplot(mpg, aes(x = drv, y = hwy)) +
  geom_boxplot() +
  labs(title = "Plot 2")
p3 <- ggplot(mpg, aes(x = cty, y = hwy)) +
  geom_point() +
  labs(title = "Plot 3")

(p1 | p2) / p3
```

Using the three plots from the previous exercise, re-create the following patchwork:



Summary

In this chapter you learned about adding plot labels such as title, subtitle, and caption as well as modifying default axis labels, using annotation to add informational text to your plot or to highlight specific data points, customizing the axis scales, and changing the theme of your plot. You also learned about combining multiple plots in a single graph using both simple and complex plot layouts.

While you've so far learned about how to make many different types of plots and how to customize them using a variety of techniques, we've barely scratched the surface of what you can create with ggplot2. If you want to get a comprehensive understanding of ggplot2, we recommend reading the book *ggplot2: Elegant Graphics for Data Analysis* (Springer). Other useful resources are the *R Graphics Cookbook* by Winston Chang (O'Reilly) and *Fundamentals of Data Visualization* by Claus Wilke (O'Reilly).

PART III
Transform

The second part of the book was a deep dive into data visualization. In this part of the book, you'll learn about the most important types of variables that you'll encounter inside a data frame and learn the tools you can use to work with them.

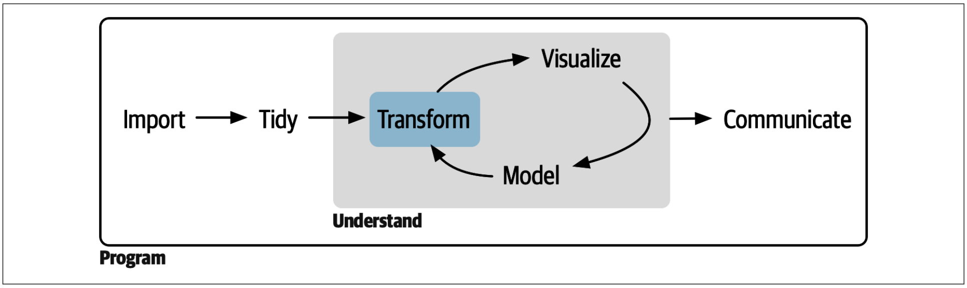


Figure III-1. The options for data transformation depend heavily on the type of data involved, the subject of this part of the book.

You can read these chapters as you need them; they're designed to be largely standalone so that they can be read out of order.

- Chapter 12 teaches you about logical vectors. These are the simplest types of vectors, but they are extremely powerful. You'll learn how to create them with numeric comparisons, how to combine them with Boolean algebra, how to use them in summaries, and how to use them for condition transformations.

- Chapter 13 dives into tools for vectors of numbers, the powerhouse of data science. You'll learn more about counting and a bunch of important transformation and summary functions.
- Chapter 14 gives you the tools to work with strings: you'll slice them, you'll dice them, and you'll stick them back together again. This chapter mostly focuses on the stringr package, but you'll also learn some more tidyr functions devoted to extracting data from character strings.
- Chapter 15 introduces you to regular expressions, a powerful tool for manipulating strings. This chapter will take you from thinking that a cat walked over your keyboard to reading and writing complex string patterns.
- Chapter 16 introduces factors: the data type that R uses to store categorical data. You use a factor when a variable has a fixed set of possible values, or when you want to use a nonalphabetical ordering of a string.
- Chapter 17 gives you the key tools for working with dates and date-times. Unfortunately, the more you learn about date-times, the more complicated they seem to get, but with the help of the lubridate package, you'll learn to how to overcome the most common challenges.
- Chapter 18 discusses missing values in depth. We've discussed them a couple of times in isolation, but now it's time to discuss them holistically, helping you come to grips with the difference between implicit and explicit missing values and how and why you might convert between them.
- Chapter 19 finishes up this part of the book by giving you the tools to join two (or more) data frames together. Learning about joins will force you to grapple with the idea of keys and think about how you identify each row in a dataset.

CHAPTER 12

Logical Vectors

Introduction

In this chapter, you'll learn tools for working with logical vectors. Logical vectors are the simplest type of vector because each element can be only one of three possible values: TRUE, FALSE, and NA. It's relatively rare to find logical vectors in your raw data, but you'll create and manipulate them in the course of almost every analysis.

We'll begin by discussing the most common way of creating logical vectors: with numeric comparisons. Then you'll learn about how you can use Boolean algebra to combine different logical vectors, as well as some useful summaries. We'll finish off with if_else() and case_when(), two useful functions for making conditional changes powered by logical vectors.

Prerequisites

Most of the functions you'll learn about in this chapter are provided by base R, so we don't need the tidyverse, but we'll still load it so we can use mutate(), filter(), and friends to work with data frames. We'll also continue to draw examples from the nycflights13::flights dataset.

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

However, as we start to cover more tools, there won't always be a perfect real example. So we'll start making up some dummy data with c():

```
x <- c(1, 2, 3, 5, 7, 11, 13)
x * 2
#> [1]  2  4  6 10 14 22 26
```

This makes it easier to explain individual functions at the cost of making it harder to see how it might apply to your data problems. Just remember that any manipulation we do to a free-floating vector, you can do to a variable inside a data frame with `mutate()` and friends.

```
df <- tibble(x)
df |>
  mutate(y = x *  2)
#> # A tibble: 7 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1     2
#> 2     2     4
#> 3     3     6
#> 4     5    10
#> 5     7    14
#> 6    11    22
#> # … with 1 more row
```

Comparisons

A common way to create a logical vector is via a numeric comparison with <, <=, >, >=, !=, and ==. So far, we've mostly created logical variables transiently within `filter()`—they are computed, used, and then thrown away. For example, the following filter finds all daytime departures that arrive roughly on time:

```
flights |>
  filter(dep_time > 600 & dep_time < 2000 & abs(arr_delay) < 20)
#> # A tibble: 172,286 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      601            600         1      844            850
#> 2  2013     1     1      602            610        -8      812            820
#> 3  2013     1     1      602            605        -3      821            805
#> 4  2013     1     1      606            610        -4      858            910
#> 5  2013     1     1      606            610        -4      837            845
#> 6  2013     1     1      607            607         0      858            915
#> # … with 172,280 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

It's useful to know that this is a shortcut and you can explicitly create the underlying logical variables with `mutate()`:

```
flights |>
  mutate(
    daytime = dep_time > 600 & dep_time < 2000,
    approx_ontime = abs(arr_delay) < 20,
    .keep = "used"
  )
#> # A tibble: 336,776 × 4
#>   dep_time arr_delay daytime approx_ontime
#>      <int>     <dbl> <lgl>   <lgl>
#> 1      517        11 FALSE   TRUE
#> 2      533        20 FALSE   FALSE
#> 3      542        33 FALSE   FALSE
#> 4       544         -18 FALSE    TRUE
#> 5       554         -25 FALSE    FALSE
#> 6       554          12 FALSE    TRUE
#> # … with 336,770 more rows
```

This is particularly useful for more complicated logic because naming the intermediate steps makes it easier to both read your code and check that each step has been computed correctly.

All told, the initial filter is equivalent to the following:

```
flights |>
  mutate(
    daytime = dep_time > 600 & dep_time < 2000,
    approx_ontime = abs(arr_delay) < 20,
  ) |>
  filter(daytime & approx_ontime)
```

Floating-Point Comparison

Beware of using == with numbers. For example, it looks like this vector contains the numbers 1 and 2:

```
x <- c(1 / 49 * 49, sqrt(2) ^ 2)
x
#> [1] 1 2
```

But if you test them for equality, you get `FALSE`:

```
x == c(1, 2)
#> [1] FALSE FALSE
```

What's going on? Computers store numbers with a fixed number of decimal places, so there's no way to exactly represent 1/49 or `sqrt(2)`, and subsequent computations will be very slightly off. We can see the exact values by calling `print()` with the `digits`[1] argument:

```
print(x, digits = 16)
#> [1] 0.9999999999999999 2.0000000000000004
```

You can see why R defaults to rounding these numbers; they really are very close to what you expect.

Now that you've seen why == is failing, what can you do about it? One option is to use `dplyr::near()`, which ignores small differences:

```
near(x, c(1, 2))
#> [1] TRUE TRUE
```

[1] R normally calls print for you (i.e., `x` is a shortcut for `print(x)`), but calling it explicitly is useful if you want to provide other arguments.

Missing Values

Missing values represent the unknown, so they are "contagious": almost any operation involving an unknown value will also be unknown:

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

The most confusing result is this one:

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

It's easiest to understand why this is true if we artificially supply a little more context:

```
# We don't know how old Mary is
age_mary <- NA

# We don't know how old John is
age_john <- NA

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

So if you want to find all flights where `dep_time` is missing, the following code doesn't work because `dep_time == NA` will yield `NA` for every single row, and `filter()` automatically drops missing values:

```
flights |>
  filter(dep_time == NA)
#> # A tibble: 0 × 19
#> # … with 19 variables: year <int>, month <int>, day <int>, dep_time <int>,
#> #   sched_dep_time <int>, dep_delay <dbl>, arr_time <int>, …
```

Instead we'll need a new tool: `is.na()`.

is.na()

`is.na(x)` works with any type of vector and returns `TRUE` for missing values and `FALSE` for everything else:

```
is.na(c(TRUE, NA, FALSE))
#> [1] FALSE  TRUE FALSE
is.na(c(1, NA, 3))
#> [1] FALSE  TRUE FALSE
is.na(c("a", NA, "b"))
#> [1] FALSE  TRUE FALSE
```

We can use is.na() to find all the rows with a missing dep_time:

```
flights |>
  filter(is.na(dep_time))
#> # A tibble: 8,255 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1       NA           1630        NA       NA           1815
#> 2  2013     1     1       NA           1935        NA       NA           2240
#> 3  2013     1     1       NA           1500        NA       NA           1825
#> 4  2013     1     1       NA            600        NA       NA            901
#> 5  2013     1     2       NA           1540        NA       NA           1747
#> 6  2013     1     2       NA           1620        NA       NA           1746
#> # … with 8,249 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

is.na() can also be useful in arrange(). arrange() usually puts all the missing values at the end, but you can override this default by first sorting by is.na():

```
flights |>
  filter(month == 1, day == 1) |>
  arrange(dep_time)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 836 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …

flights |>
  filter(month == 1, day == 1) |>
  arrange(desc(is.na(dep_time)), dep_time)
#> # A tibble: 842 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1       NA           1630        NA       NA           1815
#> 2  2013     1     1       NA           1935        NA       NA           2240
#> 3  2013     1     1       NA           1500        NA       NA           1825
#> 4  2013     1     1       NA            600        NA       NA            901
#> 5  2013     1     1      517            515         2      830            819
#> 6  2013     1     1      533            529         4      850            830
#> # … with 836 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

We'll come back to cover missing values in more depth in Chapter 18.

Exercises

1. How does `dplyr::near()` work? Type `near` to see the source code. Is `sqrt(2)^2` near 2?
2. Use `mutate()`, `is.na()`, and `count()` together to describe how the missing values in `dep_time`, `sched_dep_time`, and `dep_delay` are connected.

Boolean Algebra

Once you have multiple logical vectors, you can combine them using Boolean algebra. In R, `&` is "and," `|` is "or," `!` is "not," and `xor()` is exclusive or.[2] For example, `df |> filter(!is.na(x))` finds all rows where `x` is not missing, and `df |> filter(x < -10 | x > 0)` finds all rows where `x` is smaller than -10 or bigger than 0. Figure 12-1 shows the complete set of Boolean operations and how they work.

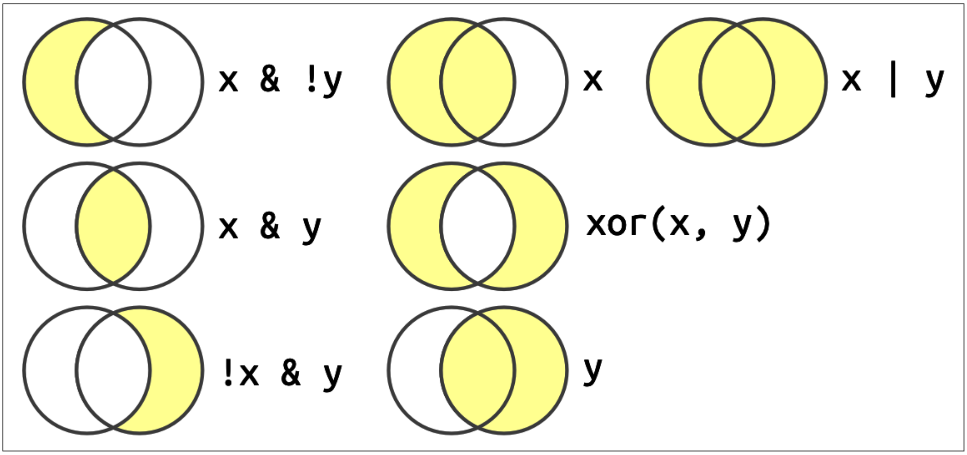


Figure 12-1. The complete set of Boolean operations. `x` is the left circle, `y` is the right circle, and the shaded region show which parts each operator selects.

As well as `&` and `|`, R also has `&&` and `||`. Don't use them in dplyr functions! These are called *short-circuiting operators* and only ever return a single `TRUE` or `FALSE`. They're important for programming, not data science.

[2] That is, `xor(x, y)` is true if `x` is true or `y` is true, but not both. This is how we usually use "or" in English. "Both" is not usually an acceptable answer to the question "Would you like ice cream or cake?"

Missing Values

The rules for missing values in Boolean algebra are a little tricky to explain because they seem inconsistent at first glance:

```
df <- tibble(x = c(TRUE, FALSE, NA))

df |>
  mutate(
    and = x & NA,
    or = x | NA
  )
#> # A tibble: 3 × 3
#>   x     and   or   
#>   <lgl> <lgl> <lgl>
#> 1 TRUE  NA    TRUE 
#> 2 FALSE FALSE NA   
#> 3 NA    NA    NA
```

To understand what's going on, think about `NA | TRUE`. A missing value in a logical vector means that the value could be either `TRUE` or `FALSE`. `TRUE | TRUE` and `FALSE | TRUE` are both `TRUE` because at least one of them is `TRUE`. So `NA | TRUE` must also be `TRUE` because `NA` can either be `TRUE` or `FALSE`. However, `NA | FALSE` is `NA` because we don't know if `NA` is `TRUE` or `FALSE`. Similar reasoning applies with `NA & FALSE`.

Order of Operations

Note that the order of operations doesn't work like English. Take the following code that finds all flights that departed in November or December:

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

You might be tempted to write it like you'd say in English: "Find all flights that departed in November or December":

```
flights |>
   filter(month == 11 | 12)
#> # A tibble: 336,776 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      533            529         4      850            830
#> 3  2013     1     1      542            540         2      923            850
#> 4  2013     1     1      544            545        -1     1004           1022
#> 5  2013     1     1      554            600        -6      812            837
#> 6  2013     1     1      554            558        -4      740            728
#> # … with 336,770 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

This code doesn't error, but it also doesn't seem to have worked. What's going on? Here, R first evaluates `month == 11` creating a logical vector, which we call `nov`. It computes `nov | 12`. When you use a number with a logical operator, it converts everything apart from 0 to TRUE, so this is equivalent to nov | TRUE, which will always be TRUE, so every row will be selected:

```
flights |>
  mutate(
    nov = month == 11,
    final = nov | 12,
    .keep = "used"
  )
#> # A tibble: 336,776 × 3
#>   month nov   final
#>   <int> <lgl> <lgl>
#> 1     1 FALSE TRUE
#> 2     1 FALSE TRUE
#> 3     1 FALSE TRUE
#> 4     1 FALSE TRUE
#> 5     1 FALSE TRUE
#> 6     1 FALSE TRUE
#> # … with 336,770 more rows
```

%in%

An easy way to avoid the problem of getting your ==s and |s in the right order is to use %in%. x %in% y returns a logical vector the same length as x that is TRUE whenever a value in x is anywhere in y.

```
1:12 %in% c(1, 5, 11)
#>  [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
letters[1:10] %in% c("a", "e", "i", "o", "u")
#>  [1]  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE
```

So to find all flights in November and December, we could write:

```
flights |>
  filter(month %in% c(11, 12))
```

Note that %in% obeys different rules for NA to ==, as NA %in% NA is TRUE.

```
c(1, 2, NA) == NA
#> [1] NA NA NA
c(1, 2, NA) %in% NA
#> [1] FALSE FALSE  TRUE
```

This can make for a useful shortcut:

```
flights |>
  filter(dep_time %in% c(NA, 0800))
#> # A tibble: 8,803 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      800            800         0     1022           1014
#> 2  2013     1     1      800            810       -10      949            955
#> 3  2013     1     1       NA           1630        NA       NA           1815
#> 4  2013     1     1       NA           1935        NA       NA           2240
#> 5  2013     1     1       NA           1500        NA       NA           1825
#> 6  2013     1     1       NA            600        NA       NA            901
#> # … with 8,797 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

Exercises

1. Find all flights where `arr_delay` is missing but `dep_delay` is not. Find all flights where neither `arr_time` nor `sched_arr_time` is missing, but `arr_delay` is.
2. How many flights have a missing `dep_time`? What other variables are missing in these rows? What might these rows represent?
3. Assuming that a missing `dep_time` implies that a flight is cancelled, look at the number of cancelled flights per day. Is there a pattern? Is there a connection between the proportion of canceled flights and the average delay of non-cancelled flights?

Summaries

The following sections describe some useful techniques for summarizing logical vectors. As well as functions that only work specifically with logical vectors, you can also use functions that work with numeric vectors.

Logical Summaries

There are two main logical summaries: `any()` and `all()`. `any(x)` is the equivalent of `|`; it'll return `TRUE` if there are any `TRUE`s in `x`. `all(x)` is equivalent of `&`; it'll return `TRUE` only if all values of `x` are `TRUE`s. Like all summary functions, they'll return `NA` if there are any missing values present, and as usual you can make the missing values go away with `na.rm = TRUE`.

For example, we could use `all()` and `any()` to find out if every flight was delayed on departure by at most an hour or if any flights were delayed on arrival by five hours or more. And using `group_by()` allows us to do that by day:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    all_delayed = all(dep_delay <= 60, na.rm = TRUE),
    any_long_delay = any(arr_delay >= 300, na.rm = TRUE),
    .groups = "drop"
  )
#> # A tibble: 365 × 5
#>    year month   day all_delayed any_long_delay
#>   <int> <int> <int> <lgl>       <lgl>
#> 1  2013     1     1 FALSE       TRUE
#> 2  2013     1     2 FALSE       TRUE
#> 3  2013     1     3 FALSE       FALSE
#> 4  2013     1     4 FALSE       FALSE
#> 5  2013     1     5 FALSE       TRUE
#> 6  2013     1     6 FALSE       FALSE
#> # … with 359 more rows
```

In most cases, however, any() and all() are a little too crude, and it would be nice to be able to get a little more detail about how many values are TRUE or FALSE. That leads us to the numeric summaries.

Numeric Summaries of Logical Vectors

When you use a logical vector in a numeric context, TRUE becomes 1, and FALSE becomes 0. This makes sum() and mean() useful with logical vectors because sum(x) gives the number of TRUEs and mean(x) gives the proportion of TRUEs (because mean() is just sum() divided by length()).

That, for example, allows us to see the proportion of flights that were delayed on departure by at most an hour and the number of flights that were delayed on arrival by five hours or more:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    all_delayed = mean(dep_delay <= 60, na.rm = TRUE),
    any_long_delay = sum(arr_delay >= 300, na.rm = TRUE),
    .groups = "drop"
  )
#> # A tibble: 365 × 5
#>    year month   day all_delayed any_long_delay
#>   <int> <int> <int>       <dbl>          <int>
#> 1  2013     1     1       0.939              3
#> 2  2013     1     2       0.914              3
#> 3  2013     1     3       0.941              0
#> 4  2013     1     4       0.953              0
#> 5  2013     1     5       0.964              1
#> 6  2013     1     6       0.959              0
#> # … with 359 more rows
```

Logical Subsetting

There's one final use for logical vectors in summaries: you can use a logical vector to filter a single variable to a subset of interest. This makes use of the base [(pronounced subset) operator, which you'll learn more about in "Selecting Multiple Elements with [" on page 490.

Imagine we wanted to look at the average delay just for flights that were actually delayed. One way to do so would be to first filter the flights and then calculate the average delay:

```
flights |>
  filter(arr_delay > 0) |>
  group_by(year, month, day) |>
  summarize(
    behind = mean(arr_delay),
    n = n(),
    .groups = "drop"
  )
```

```
#> # A tibble: 365 × 5
#>    year month   day behind     n
#>   <int> <int> <int>  <dbl> <int>
#> 1  2013     1     1   32.5   461
#> 2  2013     1     2   32.0   535
#> 3  2013     1     3   27.7   460
#> 4  2013     1     4   28.3   297
#> 5  2013     1     5   22.6   238
#> 6  2013     1     6   24.4   381
#> # … with 359 more rows
```

This works, but what if we wanted to also compute the average delay for flights that arrived early? We'd need to perform a separate filter step and then figure out how to combine the two data frames together.[3] Instead, you could use `[` to perform an inline filtering: `arr_delay[arr_delay > 0]` will yield only the positive arrival delays.

This leads to:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    behind = mean(arr_delay[arr_delay > 0], na.rm = TRUE),
    ahead = mean(arr_delay[arr_delay < 0], na.rm = TRUE),
    n = n(),
    .groups = "drop"
  )
#> # A tibble: 365 × 6
#>    year month   day behind ahead     n
#>   <int> <int> <int>  <dbl> <dbl> <int>
#> 1  2013     1     1   32.5 -12.5   842
#> 2  2013     1     2   32.0 -14.3   943
#> 3  2013     1     3   27.7 -18.2   914
#> 4  2013     1     4   28.3 -17.0   915
#> 5  2013     1     5   22.6 -14.0   720
#> 6  2013     1     6   24.4 -13.6   832
#> # … with 359 more rows
```

Also note the difference in the group size: in the first chunk, `n()` gives the number of delayed flights per day; in the second, `n()` gives the total number of flights.

Exercises

1. What will `sum(is.na(x))` tell you? How about `mean(is.na(x))`?
2. What does `prod()` return when applied to a logical vector? What logical summary function is it equivalent to? What does `min()` return when applied to a logical vector? What logical summary function is it equivalent to? Read the documentation and perform a few experiments.

[3] We'll cover this in Chapter 19.

Conditional Transformations

One of the most powerful features of logical vectors are their use for conditional transformations, i.e., doing one thing for condition x and doing something different for condition y. There are two important tools for this: `if_else()` and `case_when()`.

if_else()

If you want to use one value when a condition is `TRUE` and another value when it's `FALSE`, you can use `dplyr::if_else()`.[4] You'll always use the first three argument of `if_else()`. The first argument, `condition`, is a logical vector; the second, `true`, gives the output when the condition is true; and the third, `false`, gives the output if the condition is false.

Let's begin with a simple example of labeling a numeric vector as either "+ve" (positive) or "-ve" (negative):

```
x <- c(-3:3, NA)
if_else(x > 0, "+ve", "-ve")
#> [1] "-ve" "-ve" "-ve" "-ve" "+ve" "+ve" "+ve" NA
```

There's an optional fourth argument, `missing`, which will be used if the input is `NA`:

```
if_else(x > 0, "+ve", "-ve", "???")
#> [1] "-ve" "-ve" "-ve" "-ve" "+ve" "+ve" "+ve" "???"
```

You can also use vectors for the `true` and `false` arguments. For example, this allows us to create a minimal implementation of `abs()`:

```
if_else(x < 0, -x, x)
#> [1]  3  2  1  0  1  2  3 NA
```

So far all the arguments have used the same vectors, but you can of course mix and match. For example, you could implement a simple version of `coalesce()` like this:

```
x1 <- c(NA, 1, 2, NA)
y1 <- c(3, NA, 4, 6)
if_else(is.na(x1), y1, x1)
#> [1] 3 1 2 6
```

You might have noticed a small infelicity in our previous labeling example: zero is neither positive nor negative. We could resolve this by adding an additional `if_else()`:

```
if_else(x == 0, "0", if_else(x < 0, "-ve", "+ve"), "???")
#> [1] "-ve" "-ve" "-ve" "0"   "+ve" "+ve" "+ve" "???"
```

[4] dplyr's `if_else()` is similar to base R's `ifelse()`. There are two main advantages of `if_else()` over `ifelse()`: you can choose what should happen to missing values, and `if_else()` is much more likely to give you a meaningful error if your variables have incompatible types.

This is already a little hard to read, and you can imagine it would only get harder if you have more conditions. Instead, you can switch to `dplyr::case_when()`.

case_when()

dplyr's `case_when()` is inspired by SQL's `CASE` statement and provides a flexible way of performing different computations for different conditions. It has a special syntax that unfortunately looks like nothing else you'll use in the tidyverse. It takes pairs that look like `condition ~ output`. `condition` must be a logical vector; when it's `TRUE`, `output` will be used.

This means we could re-create our previous nested `if_else()` as follows:

```
x <- c(-3:3, NA)
case_when(
  x == 0   ~ "0",
  x < 0    ~ "-ve",
  x > 0    ~ "+ve",
  is.na(x) ~ "???"
)
#> [1] "-ve" "-ve" "-ve" "0"   "+ve" "+ve" "+ve" "???"
```

This is more code, but it's also more explicit.

To explain how `case_when()` works, let's explore some simpler cases. If none of the cases matches, the output gets an `NA`:

```
case_when(
  x < 0 ~ "-ve",
  x > 0 ~ "+ve"
)
#> [1] "-ve" "-ve" "-ve" NA    "+ve" "+ve" "+ve" NA
```

If you want to create a "default"/catchall value, use `TRUE` on the left side:

```
case_when(
  x < 0 ~ "-ve",
  x > 0 ~ "+ve",
  TRUE ~ "???"
)
#> [1] "-ve" "-ve" "-ve" "???" "+ve" "+ve" "+ve" "???"
```

Note that if multiple conditions match, only the first will be used:

```
case_when(
  x > 0 ~ "+ve",
  x > 2 ~ "big"
)
#> [1] NA    NA    NA    NA    "+ve" "+ve" "+ve" NA
```

Just like with `if_else()` you can use variables on both sides of the `~`, and you can mix and match variables as needed for your problem. For example, we could use `case_when()` to provide some human-readable labels for the arrival delay:

```
flights |>
  mutate(
    status = case_when(
      is.na(arr_delay)      ~ "cancelled",
      arr_delay < -30       ~ "very early",
      arr_delay < -15       ~ "early",
      abs(arr_delay) <= 15  ~ "on time",
      arr_delay < 60        ~ "late",
      arr_delay < Inf       ~ "very late",
    ),
    .keep = "used"
  )
#> # A tibble: 336,776 × 2
#>   arr_delay status
#>       <dbl> <chr>
#> 1        11 on time
#> 2        20 late
#> 3        33 late
#> 4       -18 early
#> 5       -25 early
#> 6        12 on time
#> # … with 336,770 more rows
```

Be wary when writing this sort of complex `case_when()` statement; my first two attempts used a mix of `<` and `>`, and I kept accidentally creating overlapping conditions.

Compatible Types

Note that both `if_else()` and `case_when()` require *compatible* types in the output. If they're not compatible, you'll see errors like this:

```
if_else(TRUE, "a", 1)
#> Error in `if_else()`:
#> ! Can't combine `true` <character> and `false` <double>.

case_when(
  x < -1 ~ TRUE,
  x > 0  ~ now()
)
#> Error in `case_when()`:
#> ! Can't combine `..1 (right)` <logical> and `..2 (right)` <datetime<local>>.
```

Overall, relatively few types are compatible, because automatically converting one type of vector to another is a common source of errors. Here are the most important cases that are compatible:

- Numeric and logical vectors are compatible, as we discussed in "Numeric Summaries of Logical Vectors" on page 214.
- Strings and factors (Chapter 16) are compatible, because you can think of a factor as a string with a restricted set of values.

- Dates and date-times, which we'll discuss in Chapter 17, are compatible because you can think of a date as a special case of date-time.
- NA, which is technically a logical vector, is compatible with everything because every vector has some way of representing a missing value.

We don't expect you to memorize these rules, but they should become second nature over time because they are applied consistently throughout the tidyverse.

Exercises

1. A number is even if it's divisible by two, which in R you can find out with `x %% 2 == 0`. Use this fact and `if_else()` to determine whether each number between 0 and 20 is even or odd.
2. Given a vector of days like `x <- c("Monday", "Saturday", "Wednesday")`, use an `ifelse()` statement to label them as weekends or weekdays.
3. Use `ifelse()` to compute the absolute value of a numeric vector called `x`.
4. Write a `case_when()` statement that uses the `month` and `day` columns from `flights` to label a selection of important US holidays (e.g., New Years Day, Fourth of July, Thanksgiving, and Christmas). First create a logical column that is either `TRUE` or `FALSE`, and then create a character column that either gives the name of the holiday or is `NA`.

Summary

The definition of a logical vector is simple because each value must be either `TRUE`, `FALSE`, or `NA`. But logical vectors provide a huge amount of power. In this chapter, you learned how to create logical vectors with `>`, `<`, `<=`, `=>`, `==`, `!=`, and `is.na()`; how to combine them with `!`, `&`, and `|`; and how to summarize them with `any()`, `all()`, `sum()`, and `mean()`. You also learned the powerful `if_else()` and `case_when()` functions that allow you to return values depending on the value of a logical vector.

We'll see logical vectors again and again in the following chapters. For example, in Chapter 14, you'll learn about `str_detect(x, pattern)`, which returns a logical vector that's `TRUE` for the elements of `x` that match the `pattern`, and in Chapter 17, you'll create logical vectors from the comparison of dates and times. But for now, we're going to move onto the next most important type of vector: numeric vectors.

CHAPTER 13

Numbers

Introduction

Numeric vectors are the backbone of data science, and you've already used them a bunch of times earlier in the book. Now it's time to systematically survey what you can do with them in R, ensuring that you're well situated to tackle any future problem involving numeric vectors.

We'll start by giving you a couple of tools to make numbers if you have strings and then go into a little more detail on `count()`. Then we'll dive into various numeric transformations that pair well with `mutate()`, including more general transformations that can be applied to other types of vectors but are often used with numeric vectors. We'll finish off by covering the summary functions that pair well with `summarize()` and show you how they can also be used with `mutate()`.

Prerequisites

This chapter mostly uses functions from base R, which are available without loading any packages. But we still need the tidyverse because we'll use these base R functions inside of tidyverse functions such as `mutate()` and `filter()`. Like in the previous chapter, we'll use real examples from nycflights13, as well as toy examples made with `c()` and `tribble()`.

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

Making Numbers

In most cases, you'll get numbers already recorded in one of R's numeric types: integer or double. In some cases, however, you'll encounter them as strings, possibly because you've created them by pivoting from column headers or because something has gone wrong in your data import process.

readr provides two useful functions for parsing strings into numbers: parse_double() and parse_number(). Use parse_double() when you have numbers that have been written as strings:

```
x <- c("1.2", "5.6", "1e3")
parse_double(x)
#> [1]    1.2    5.6 1000.0
```

Use parse_number() when the string contains non-numeric text that you want to ignore. This is particularly useful for currency data and percentages:

```
x <- c("$1,234", "USD 3,513", "59%")
parse_number(x)
#> [1] 1234 3513   59
```

Counts

It's surprising how much data science you can do with just counts and a little basic arithmetic, so dplyr strives to make counting as easy as possible with count(). This function is great for quick exploration and checks during analysis:

```
flights |> count(dest)
#> # A tibble: 105 × 2
#>   dest      n
#>   <chr> <int>
#> 1 ABQ     254
#> 2 ACK     265
#> 3 ALB     439
#> 4 ANC       8
#> 5 ATL   17215
#> 6 AUS    2439
#> # … with 99 more rows
```

(Despite the advice in Chapter 4, we usually put count() on a single line because it's usually used at the console for a quick check that a calculation is working as expected.)

If you want to see the most common values, add sort = TRUE:

```
flights |> count(dest, sort = TRUE)
#> # A tibble: 105 × 2
#>   dest      n
#>   <chr> <int>
#> 1 ORD   17283
#> 2 ATL   17215
#> 3 LAX   16174
#> 4 BOS   15508
#> 5 MCO   14082
#> 6 CLT   14064
#> # … with 99 more rows
```

And remember that if you want to see all the values, you can use `|> View()` or `|> print(n = Inf)`.

You can perform the same computation "by hand" with `group_by()`, `summarize()`, and `n()`. This is useful because it allows you to compute other summaries at the same time:

```
flights |>
  group_by(dest) |>
  summarize(
    n = n(),
    delay = mean(arr_delay, na.rm = TRUE)
  )
#> # A tibble: 105 × 3
#>   dest      n delay
#>   <chr> <int> <dbl>
#> 1 ABQ     254  4.38
#> 2 ACK     265  4.85
#> 3 ALB     439 14.4
#> 4 ANC       8 -2.5
#> 5 ATL   17215 11.3
#> 6 AUS    2439  6.02
#> # … with 99 more rows
```

`n()` is a special summary function that doesn't take any arguments and instead accesses information about the "current" group. This means that it works only inside dplyr verbs:

```
n()
#> Error in `n()`:
#> ! Must only be used inside data-masking verbs like `mutate()`,
#>   `filter()`, and `group_by()`.
```

There are a couple of variants of `n()` and `count()` that you might find useful:

- `n_distinct(x)` counts the number of distinct (unique) values of one or more variables. For example, we could figure out which destinations are served by the most carriers:

  ```
  flights |>
    group_by(dest) |>
    summarize(carriers = n_distinct(carrier)) |>
    arrange(desc(carriers))
  #> # A tibble: 105 × 2
  #>   dest  carriers
  #>   <chr>    <int>
  #> 1 ATL          7
  #> 2 BOS          7
  #> 3 CLT          7
  #> 4 ORD          7
  #> 5 TPA          7
  #> 6 AUS          6
  #> # … with 99 more rows
  ```

- A weighted count is a sum. For example, you could "count" the number of miles each plane flew:

```
flights |>
  group_by(tailnum) |>
  summarize(miles = sum(distance))
#> # A tibble: 4,044 × 2
#>   tailnum  miles
#>   <chr>    <dbl>
#> 1 D942DN    3418
#> 2 N0EGMQ  250866
#> 3 N10156  115966
#> 4 N102UW   25722
#> 5 N103US   24619
#> 6 N104UW   25157
#> # … with 4,038 more rows
```

Weighted counts are a common problem, so `count()` has a `wt` argument that does the same thing:

```
flights |> count(tailnum, wt = distance)
```

- You can count missing values by combining `sum()` and `is.na()`. In the `flights` dataset this represents flights that are cancelled:

```
flights |>
  group_by(dest) |>
  summarize(n_cancelled = sum(is.na(dep_time)))
#> # A tibble: 105 × 2
#>   dest  n_cancelled
#>   <chr>       <int>
#> 1 ABQ             0
#> 2 ACK             0
#> 3 ALB            20
#> 4 ANC             0
#> 5 ATL           317
#> 6 AUS            21
#> # … with 99 more rows
```

Exercises

1. How can you use `count()` to count the number rows with a missing value for a given variable?
2. Expand the following calls to `count()` to instead use `group_by()`, `summarize()`, and `arrange()`:

   a. `flights |> count(dest, sort = TRUE)`

   b. `flights |> count(tailnum, wt = distance)`

Numeric Transformations

Transformation functions work well with `mutate()` because their output is the same length as the input. The vast majority of transformation functions are already built into base R. It's impractical to list them all, so this section will show the most useful ones. As an example, while R provides all the trigonometric functions that you might dream of, we don't list them here because they're rarely needed for data science.

Arithmetic and Recycling Rules

We introduced the basics of arithmetic (`+`, `-`, `*`, `/`, `^`) in Chapter 2 and have used them a bunch since. These functions don't need a huge amount of explanation because they do what you learned in grade school. But we need to briefly talk about the *recycling rules*, which determine what happens when the left and right sides have different lengths. This is important for operations like `flights |> mutate(air_time = air_time / 60)` because there are 336,776 numbers on the left of `/` but only one on the right.

R handles mismatched lengths by *recycling*, or repeating, the short vector. We can see this in operation more easily if we create some vectors outside of a data frame:

```
x <- c(1, 2, 10, 20)
x / 5
#> [1] 0.2 0.4 2.0 4.0
# is shorthand for
x / c(5, 5, 5, 5)
#> [1] 0.2 0.4 2.0 4.0
```

Generally, you want to recycle only single numbers (i.e., vectors of length 1), but R will recycle any shorter length vector. It usually (but not always) gives you a warning if the longer vector isn't a multiple of the shorter:

```
x * c(1, 2)
#> [1]  1  4 10 40
x * c(1, 2, 3)
#> Warning in x * c(1, 2, 3): longer object length is not a multiple of shorter
#> object length
#> [1]  1  4 30 20
```

These recycling rules are also applied to logical comparisons (`==`, `<`, `<=`, `>`, `>=`, `!=`) and can lead to a surprising result if you accidentally use `==` instead of `%in%` and the data frame has an unfortunate number of rows. For example, take this code, which attempts to find all flights in January and February:

```
flights |>
  filter(month == c(1, 2))
#> # A tibble: 25,977 × 19
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1      542            540         2      923            850
#> 3  2013     1     1      554            600        -6      812            837
#> 4  2013     1     1      555            600        -5      913            854
#> 5  2013     1     1      557            600        -3      838            846
#> 6  2013     1     1      558            600        -2      849            851
#> # … with 25,971 more rows, and 11 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

The code runs without error, but it doesn't return what you want. Because of the recycling rules, it finds flights in odd-numbered rows that departed in January and flights in even numbered rows that departed in February. Unfortunately, there's no warning because `flights` has an even number of rows.

To protect you from this type of silent failure, most tidyverse functions use a stricter form of recycling that recycles only single values. Unfortunately, that doesn't help here, or in many other cases, because the key computation is performed by the base R function ==, not `filter()`.

Minimum and Maximum

The arithmetic functions work with pairs of variables. Two closely related functions are `pmin()` and `pmax()`, which when given two or more variables will return the smallest or largest value in each row:

```
df <- tribble(
  ~x, ~y,
  1,  3,
  5,  2,
  7, NA,
)

df |>
  mutate(
    min = pmin(x, y, na.rm = TRUE),
    max = pmax(x, y, na.rm = TRUE)
  )
#> # A tibble: 3 × 4
#>       x     y   min   max
#>   <dbl> <dbl> <dbl> <dbl>
#> 1     1     3     1     3
#> 2     5     2     2     5
#> 3     7    NA     7     7
```

Note that these are different from the summary functions `min()` and `max()`, which take multiple observations and return a single value. You can tell that you've used the wrong form when all the minimums and all the maximums have the same value:

```
df |>
  mutate(
    min = min(x, y, na.rm = TRUE),
    max = max(x, y, na.rm = TRUE)
  )
#> # A tibble: 3 × 4
#>       x     y   min   max
#>   <dbl> <dbl> <dbl> <dbl>
#> 1     1     3     1     7
#> 2     5     2     1     7
#> 3     7    NA     1     7
```

Modular Arithmetic

Modular arithmetic is the technical name for the type of math you did before you learned about decimal places, i.e., division that yields a whole number and a remainder. In R, `%/%` does integer division, and `%%` computes the remainder:

```
1:10 %/% 3
#>  [1] 0 0 1 1 1 2 2 2 3 3
1:10 %% 3
#>  [1] 1 2 0 1 2 0 1 2 0 1
```

Modular arithmetic is handy for the `flights` dataset, because we can use it to unpack the `sched_dep_time` variable into `hour` and `minute`:

```
flights |>
  mutate(
    hour = sched_dep_time %/% 100,
    minute = sched_dep_time %% 100,
    .keep = "used"
  )
#> # A tibble: 336,776 × 3
#>   sched_dep_time  hour minute
#>            <int> <dbl>  <dbl>
#> 1            515     5     15
#> 2            529     5     29
#> 3            540     5     40
#> 4            545     5     45
#> 5            600     6      0
#> 6            558     5     58
#> # … with 336,770 more rows
```

We can combine that with the `mean(is.na(x))` trick from "Summaries" on page 213 to see how the proportion of cancelled flights varies over the course of the day. The results are shown in Figure 13-1.

```
flights |>
  group_by(hour = sched_dep_time %/% 100) |>
  summarize(prop_cancelled = mean(is.na(dep_time)), n = n()) |>
  filter(hour > 1) |>
  ggplot(aes(x = hour, y = prop_cancelled)) +
  geom_line(color = "grey50") +
  geom_point(aes(size = n))
```

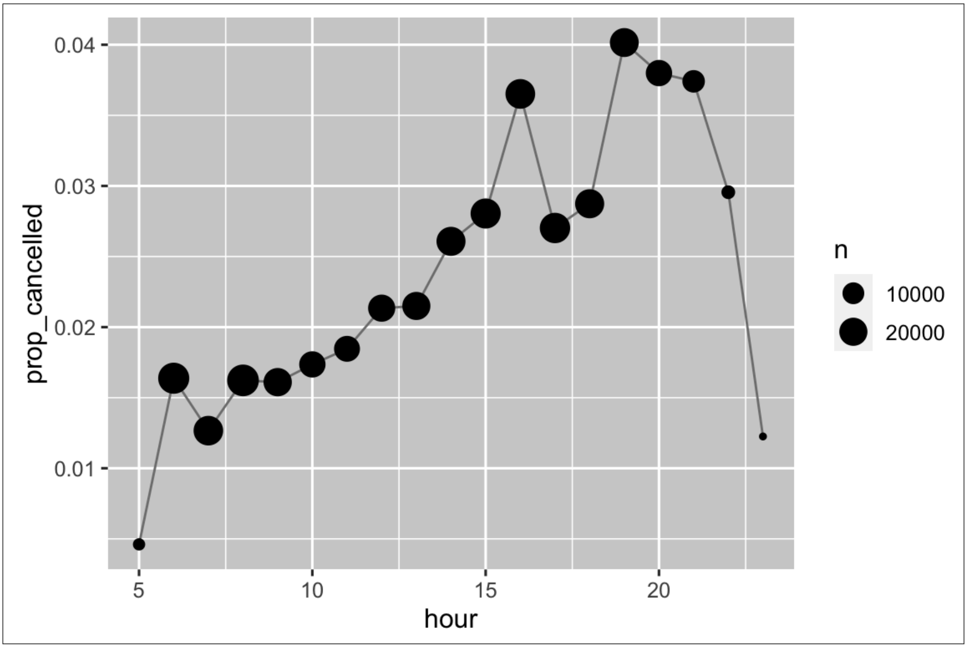


Figure 13-1. A line plot with scheduled departure hour on the x-axis, and proportion of cancelled flights on the y-axis. Cancellations seem to accumulate over the course of the day until 8 p.m., and very late flights are much less likely to be cancelled.

Logarithms

Logarithms are an incredibly useful transformation for dealing with data that ranges across multiple orders of magnitude and for converting exponential growth to linear growth. In R, you have a choice of three logarithms: `log()` (the natural log, base e), `log2()` (base 2), and `log10()` (base 10). We recommend using `log2()` or `log10()`. `log2()` is easy to interpret because a difference of 1 on the log scale corresponds to doubling on the original scale, and a difference of -1 corresponds to halving, whereas `log10()` is easy to back-transform because, for example, 3 is 10^3 = 1000. The inverse of `log()` is `exp()`; to compute the inverse of `log2()` or `log10()`, you'll need to use `2^` or `10^`.

Rounding

Use `round(x)` to round a number to the nearest integer:

```
round(123.456)
#> [1] 123
```

You can control the precision of the rounding with the second argument, `digits`. `round(x, digits)` rounds to the nearest `10^-n`, so `digits = 2` will round to the nearest 0.01. This definition is useful because it implies `round(x, -3)` will round to the nearest thousand, which indeed it does:

```
round(123.456, 2)  # two digits
#> [1] 123.46
round(123.456, 1)  # one digit
#> [1] 123.5
round(123.456, -1) # round to nearest ten
#> [1] 120
round(123.456, -2) # round to nearest hundred
#> [1] 100
```

There's one weirdness with `round()` that seems surprising at first glance:

```
round(c(1.5, 2.5))
#> [1] 2 2
```

`round()` uses what's known as "round half to even" or Banker's rounding: if a number is halfway between two integers, it will be rounded to the *even* integer. This is a good strategy because it keeps the rounding unbiased: half of all 0.5s are rounded up, and half are rounded down.

`round()` is paired with `floor()`, which always rounds down, and `ceiling()`, which always rounds up:

```
x <- 123.456

floor(x)
#> [1] 123
ceiling(x)
#> [1] 124
```

These functions don't have a `digits` argument, so you can instead scale down, round, and then scale back up:

```
# Round down to nearest two digits
floor(x / 0.01) * 0.01
#> [1] 123.45
# Round up to nearest two digits
ceiling(x / 0.01) * 0.01
#> [1] 123.46
```

You can use the same technique if you want to `round()` to a multiple of some other number:

```
# Round to nearest multiple of 4
round(x / 4) * 4
#> [1] 124

# Round to nearest 0.25
round(x / 0.25) * 0.25
#> [1] 123.5
```

Cutting Numbers into Ranges

Use `cut()`[1] to break up (aka *bin*) a numeric vector into discrete buckets:

```
x <- c(1, 2, 5, 10, 15, 20)
cut(x, breaks = c(0, 5, 10, 15, 20))
#> [1] (0,5]   (0,5]   (0,5]   (5,10]  (10,15] (15,20]
#> Levels: (0,5] (5,10] (10,15] (15,20]
```

The breaks don't need to be evenly spaced:

```
cut(x, breaks = c(0, 5, 10, 100))
#> [1] (0,5]    (0,5]    (0,5]    (5,10]   (10,100] (10,100]
#> Levels: (0,5] (5,10] (10,100]
```

You can optionally supply your own `labels`. Note that there should be one less `labels` than `breaks`.

```
cut(x,
  breaks = c(0, 5, 10, 15, 20),
  labels = c("sm", "md", "lg", "xl")
)
#> [1] sm sm sm md lg xl
#> Levels: sm md lg xl
```

Any values outside of the range of the breaks will become `NA`:

```
y <- c(NA, -10, 5, 10, 30)
cut(y, breaks = c(0, 5, 10, 15, 20))
#> [1] <NA>   <NA>   (0,5]  (5,10] <NA>
#> Levels: (0,5] (5,10] (10,15] (15,20]
```

See the documentation for other useful arguments such as `right` and `include.lowest`, which control if the intervals are `[a, b)` or `(a, b]` and if the lowest interval should be `[a, b]`.

Cumulative and Rolling Aggregates

Base R provides `cumsum()`, `cumprod()`, `cummin()`, and `cummax()` for running, or cumulative, sums, products, and mins and maxes. dplyr provides `cummean()` for cumulative means. Cumulative sums tend to come up the most in practice:

```
x <- 1:10
cumsum(x)
#>  [1]  1  3  6 10 15 21 28 36 45 55
```

If you need more complex rolling or sliding aggregates, try the slider package.

[1] ggplot2 provides some helpers for common cases in `cut_interval()`, `cut_number()`, and `cut_width()`. ggplot2 is an admittedly weird place for these functions to live, but they are useful as part of histogram computation and were written before any other parts of the tidyverse existed.

Exercises

1. Explain in words what each line of the code used to generate Figure 13-1 does.
2. What trigonometric functions does R provide? Guess some names and look up the documentation. Do they use degrees or radians?
3. Currently `dep_time` and `sched_dep_time` are convenient to look at but hard to compute with because they're not really continuous numbers. You can see the basic problem by running the following code; there's a gap between each hour:

    ```
    flights |>
      filter(month == 1, day == 1) |>
      ggplot(aes(x = sched_dep_time, y = dep_delay)) +
      geom_point()
    ```

    Convert them to a more truthful representation of time (either fractional hours or minutes since midnight).
4. Round `dep_time` and `arr_time` to the nearest five minutes.

General Transformations

The following sections describe some general transformations that are often used with numeric vectors but can be applied to all other column types.

Ranks

dplyr provides a number of ranking functions inspired by SQL, but you should always start with `dplyr::min_rank()`. It uses the typical method for dealing with ties, e.g., 1st, 2nd, 2nd, 4th.

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

Note that the smallest values get the lowest ranks; use `desc(x)` to give the largest values the smallest ranks:

```
min_rank(desc(x))
#> [1]  5  3  3  2  1 NA
```

If `min_rank()` doesn't do what you need, look at the variants `dplyr::row_number()`, `dplyr::dense_rank()`, `dplyr::percent_rank()`, and `dplyr::cume_dist()`. See the documentation for details.

```
df <- tibble(x = x)
df |>
  mutate(
    row_number = row_number(x),
    dense_rank = dense_rank(x),
    percent_rank = percent_rank(x),
    cume_dist = cume_dist(x)
```

```
  )
#> # A tibble: 6 × 5
#>       x row_number dense_rank percent_rank cume_dist
#>   <dbl>      <int>      <int>        <dbl>     <dbl>
#> 1     1          1          1         0          0.2
#> 2     2          2          2         0.25       0.6
#> 3     2          3          2         0.25       0.6
#> 4     3          4          3         0.75       0.8
#> 5     4          5          4         1          1
#> 6    NA         NA         NA        NA         NA
```

You can achieve many of the same results by picking the appropriate `ties.method` argument to base R's `rank()`; you'll probably also want to set `na.last = "keep"` to keep `NA`s as `NA`.

`row_number()` can also be used without any arguments when inside a dplyr verb. In this case, it'll give the number of the "current" row. When combined with `%%` or `%/%`, this can be a useful tool for dividing data into similarly sized groups:

```
df <- tibble(id = 1:10)

df |> 
  mutate(
    row0 = row_number() - 1,
    three_groups = row0 %% 3,
    three_in_each_group = row0 %/% 3
  )
#> # A tibble: 10 × 4
#>      id  row0 three_groups three_in_each_group
#>   <int> <dbl>        <dbl>               <dbl>
#> 1     1     0            0                   0
#> 2     2     1            1                   0
#> 3     3     2            2                   0
#> 4     4     3            0                   1
#> 5     5     4            1                   1
#> 6     6     5            2                   1
#> # … with 4 more rows
```

Offsets

`dplyr::lead()` and `dplyr::lag()` allow you to refer the values just before or just after the "current" value. They return a vector of the same length as the input, padded with `NA`s at the start or end:

```
x <- c(2, 5, 11, 11, 19, 35)
lag(x)
#> [1] NA  2  5 11 11 19
lead(x)
#> [1]  5 11 11 19 35 NA
```

- `x - lag(x)` gives you the difference between the current and previous value:

  ```
  x - lag(x)
  #> [1] NA  3  6  0  8 16
  ```

- `x == lag(x)` tells you when the current value changes:

```
x == lag(x)
#> [1]    NA FALSE FALSE  TRUE FALSE FALSE
```

You can lead or lag by more than one position by using the second argument, n.

Consecutive Identifiers

Sometimes you want to start a new group every time some event occurs. For example, when you're looking at website data, it's common to want to break up events into sessions, where you begin a new session after a gap of more than x minutes since the last activity. For example, imagine you have the times when someone visited a website:

```
events <- tibble(
  time = c(0, 1, 2, 3, 5, 10, 12, 15, 17, 19, 20, 27, 28, 30)
)
```

You've computed the time between each event and figured out if there's a gap that's big enough to qualify:

```
events <- events |>
  mutate(
    diff = time - lag(time, default = first(time)),
    has_gap = diff >= 5
  )
events
#> # A tibble: 14 × 3
#>    time  diff has_gap
#>   <dbl> <dbl> <lgl>  
#> 1     0     0 FALSE  
#> 2     1     1 FALSE  
#> 3     2     1 FALSE  
#> 4     3     1 FALSE  
#> 5     5     2 FALSE  
#> 6    10     5 TRUE   
#> # … with 8 more rows
```

But how do we go from that logical vector to something that we can group_by()? cumsum(), from "Cumulative and Rolling Aggregates" on page 230, comes to the rescue as gap, i.e., has_gap is TRUE, will increment group by one ("Numeric Summaries of Logical Vectors" on page 214):

```
events |> mutate(
  group = cumsum(has_gap)
)
#> # A tibble: 14 × 4
#>    time  diff has_gap group
#>   <dbl> <dbl> <lgl>   <int>
#> 1     0     0 FALSE       0
#> 2     1     1 FALSE       0
#> 3     2     1 FALSE       0
#> 4     3     1 FALSE       0
#> 5     5     2 FALSE       0
#> 6    10     5 TRUE        1
#> # … with 8 more rows
```

Another approach for creating grouping variables is consecutive_id(), which starts a new group every time one of its arguments changes. For example, inspired by this StackOverflow question, imagine you have a data frame with a bunch of repeated values:

```
df <- tibble(
  x = c("a", "a", "a", "b", "c", "c", "d", "e", "a", "a", "b", "b"),
  y = c(1, 2, 3, 2, 4, 1, 3, 9, 4, 8, 10, 199)
)
```

If you want to keep the first row from each repeated x, you could use group_by(), consecutive_id(), and slice_head():

```
df |>
  group_by(id = consecutive_id(x)) |>
  slice_head(n = 1)
#> # A tibble: 7 × 3
#> # Groups:   id [7]
#>   x         y    id
#>   <chr> <dbl> <int>
#> 1 a         1     1
#> 2 b         2     2
#> 3 c         4     3
#> 4 d         3     4
#> 5 e         9     5
#> 6 a         4     6
#> # … with 1 more row
```

Exercises

1. Find the 10 most delayed flights using a ranking function. How do you want to handle ties? Carefully read the documentation for min_rank().
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. What does flights |> group_by(dest) |> filter(row_number() < 4) do? What does flights |> group_by(dest) |> filter(row_number(dep_delay) < 4) do?
5. For each destination, compute the total minutes of delay. For each flight, compute the proportion of the total delay for its destination.
6. 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(), explore how the average flight delay for an hour is related to the average delay for the previous hour.

```
flights |>
  mutate(hour = dep_time %/% 100) |>
  group_by(year, month, day, hour) |>
  summarize(
    dep_delay = mean(dep_delay, na.rm = TRUE),
    n = n(),
    .groups = "drop"
  ) |>
  filter(n > 5)
```

7. Look at each destination. Can you find flights that are suspiciously fast (i.e., 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?
8. Find all destinations that are flown by at least two carriers. Use those destinations to come up with a relative ranking of the carriers based on their performance for the same destination.

Numeric Summaries

Just using the counts, means, and sums that we've introduced already can get you a long way, but R provides many other useful summary functions. Here is a selection that you might find useful.

Center

So far, we've mostly used `mean()` to summarize the center of a vector of values. As we've seen in "Case Study: Aggregates and Sample Size" on page 60, because the mean is the sum divided by the count, it is sensitive to even just a few unusually high or low values. An alternative is to use the `median()`, which finds a value that lies in the "middle" of the vector, i.e., 50% of the values are above it and 50% are below it. Depending on the shape of the distribution of the variable you're interested in, mean or median might be a better measure of center. For example, for symmetric distributions we generally report the mean, while for skewed distributions we usually report the median.

Figure 13-2 compares the mean to the median departure delay (in minutes) for each destination. The median delay is always smaller than the mean delay because flights sometimes leave multiple hours late, but they never leave multiple hours early.

```
flights |>
  group_by(year, month, day) |>
  summarize(
    mean = mean(dep_delay, na.rm = TRUE),
    median = median(dep_delay, na.rm = TRUE),
    n = n(),
    .groups = "drop"
  ) |>
  ggplot(aes(x = mean, y = median)) +
  geom_abline(slope = 1, intercept = 0, color = "white", linewidth = 2) +
  geom_point()
```

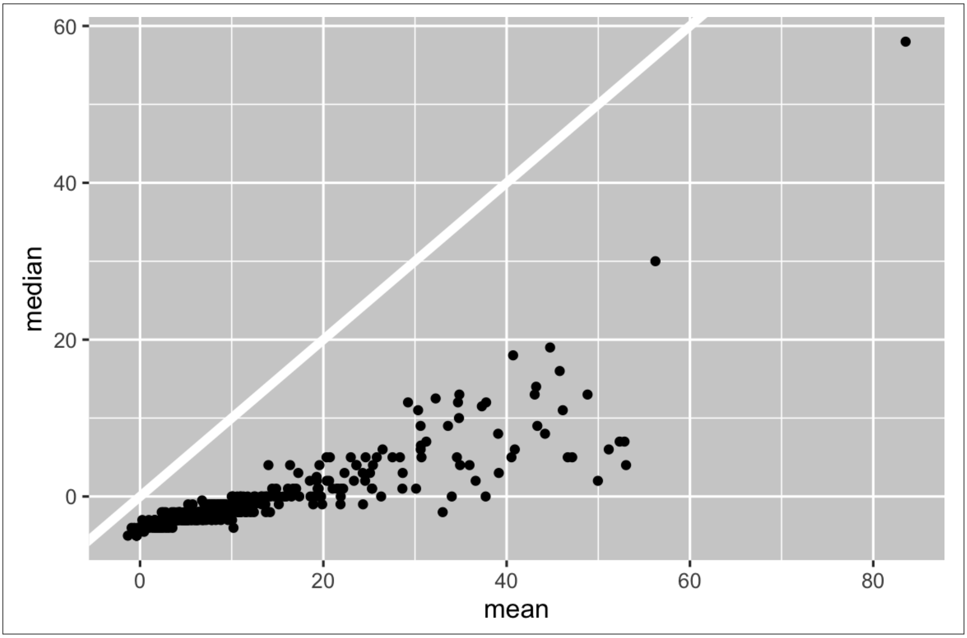


Figure 13-2. A scatterplot showing the differences of summarizing hourly departure delay with median instead of mean.

You might also wonder about the *mode*, or the most common value. This is a summary that works well only for very simple cases (which is why you might have learned about it in high school), but it doesn't work well for many real datasets. If the data is discrete, there may be multiple most common values, and if the data is continuous, there might be no most common value because every value is ever so slightly different. For these reasons, the mode tends not to be used by statisticians, and there's no mode function included in base R.[2]

Minimum, Maximum, and Quantiles

What if you're interested in locations other than the center? `min()` and `max()` will give you the largest and smallest values. Another powerful tool is `quantile()`, which is a generalization of the median: `quantile(x, 0.25)` will find the value of `x` that is greater than 25% of the values, `quantile(x, 0.5)` is equivalent to the median, and `quantile(x, 0.95)` will find the value that's greater than 95% of the values.

[2] The `mode()` function does something quite different!

For the `flights` data, you might want to look at the 95% quantile of delays rather than the maximum, because it will ignore the 5% of most delayed flights, which can be quite extreme.

```
flights |>
  group_by(year, month, day) |>
  summarize(
    max = max(dep_delay, na.rm = TRUE),
    q95 = quantile(dep_delay, 0.95, na.rm = TRUE),
    .groups = "drop"
  )
#> # A tibble: 365 × 5
#>    year month   day   max   q95
#>   <int> <int> <int> <dbl> <dbl>
#> 1  2013     1     1   853  70.1
#> 2  2013     1     2   379  85
#> 3  2013     1     3   291  68
#> 4  2013     1     4   288  60
#> 5  2013     1     5   327  41
#> 6  2013     1     6   202  51
#> # … with 359 more rows
```

Spread

Sometimes you're not so interested in where the bulk of the data lies, but in how it is spread out. Two commonly used summaries are the standard deviation, `sd(x)`, and the inter-quartile range, `IQR()`. We won't explain `sd()` here since you're probably already familiar with it, but `IQR()` might be new—it's `quantile(x, 0.75) - quantile(x, 0.25)` and gives you the range that contains the middle 50% of the data.

We can use this to reveal a small oddity in the `flights` data. You might expect the spread of the distance between origin and destination to be zero, since airports are always in the same place. But the following code reveals a data oddity for airport EGE:

```
flights |>
  group_by(origin, dest) |>
  summarize(
    distance_sd = IQR(distance),
    n = n(),
    .groups = "drop"
  ) |>
  filter(distance_sd > 0)
#> # A tibble: 2 × 4
#>   origin dest  distance_sd     n
#>   <chr>  <chr>       <dbl> <int>
#> 1 EWR    EGE             1   110
#> 2 JFK    EGE             1   103
```

Distributions

It's worth remembering that all of the summary statistics described earlier are a way of reducing the distribution to a single number. This means they're fundamentally reductive, and if you pick the wrong summary, you can easily miss important differences between groups. That's why it's always a good idea to visualize the distribution before committing to your summary statistics.

Figure 13-3 shows the overall distribution of departure delays. The distribution is so skewed that we have to zoom in to see the bulk of the data. This suggests that the mean is unlikely to be a good summary, and we might prefer the median instead.

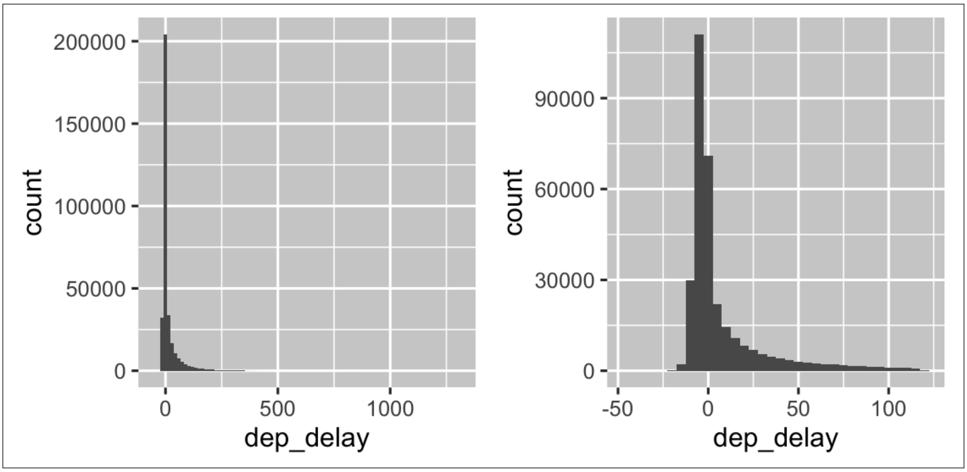
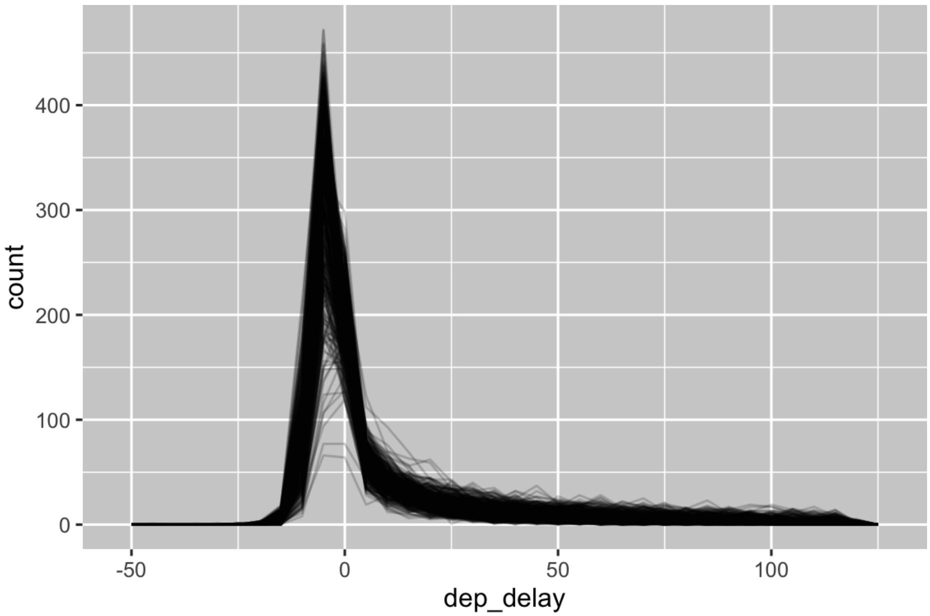


Figure 13-3. (Left) The histogram of the full data is extremely skewed, making it hard to get any details. (Right) Zooming into delays of less than two hours makes it possible to see what's happening with the bulk of the observations.

It's also a good idea to check that distributions for subgroups resemble the whole. In the following plot, 365 frequency polygons of `dep_delay`, one for each day, are overlaid. The distributions seem to follow a common pattern, suggesting it's fine to use the same summary for each day.

```
flights |>
  filter(dep_delay < 120) |>
  ggplot(aes(x = dep_delay, group = interaction(day, month))) +
  geom_freqpoly(binwidth = 5, alpha = 1/5)
```



Don't be afraid to explore your own custom summaries specifically tailored for the data that you're working with. In this case, that might mean separately summarizing the flights that left early versus the flights that left late, or given that the values are so heavily skewed, you might try a log transformation. Finally, don't forget what you learned in "Case Study: Aggregates and Sample Size" on page 60: whenever creating numerical summaries, it's a good idea to include the number of observations in each group.

Positions

There's one final type of summary that's useful for numeric vectors but also works with every other type of value: extracting a value at a specific position: `first(x)`, `last(x)`, and `nth(x, n)`.

For example, we can find the first and last departure for each day:

```
flights |>
  group_by(year, month, day) |>
  summarize(
    first_dep = first(dep_time, na_rm = TRUE),
    fifth_dep = nth(dep_time, 5, na_rm = TRUE),
    last_dep = last(dep_time, na_rm = TRUE)
  )
#> `summarise()` has grouped output by 'year', 'month'. You can override using
#> the `.groups` argument.
#> # A tibble: 365 × 6
#> # Groups:   year, month [12]
#>    year month   day first_dep fifth_dep last_dep
#>   <int> <int> <int>     <int>     <int>    <int>
#> 1  2013     1     1       517       554     2356
#> 2  2013     1     2        42       535     2354
#> 3  2013     1     3        32       520     2349
#> 4  2013     1     4        25       531     2358
#> 5  2013     1     5        14       534     2357
#> 6  2013     1     6        16       555     2355
#> # … with 359 more rows
```

(Note that because dplyr functions use _ to separate components of function and arguments names, these functions use `na_rm` instead of `na.rm`.)

If you're familiar with `[`, which we'll come back to in "Selecting Multiple Elements with [" on page 490, you might wonder if you ever need these functions. There are three reasons: the `default` argument allows you to provide a default if the specified position doesn't exist, the `order_by` argument allows you to locally override the order of the rows, and the `na_rm` argument allows you to drop missing values.

Extracting values at positions is complementary to filtering on ranks. Filtering gives you all variables, with each observation in a separate row:

```
flights |>
  group_by(year, month, day) |>
  mutate(r = min_rank(sched_dep_time)) |>
  filter(r %in% c(1, max(r)))
#> # A tibble: 1,195 × 20
#> # Groups:   year, month, day [365]
#>    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
#>   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
#> 1  2013     1     1      517            515         2      830            819
#> 2  2013     1     1     2353           2359        -6      425            445
#> 3  2013     1     1     2353           2359        -6      418            442
#> 4  2013     1     1     2356           2359        -3      425            437
#> 5  2013     1     2       42           2359        43      518            442
#> 6  2013     1     2      458            500        -2      703            650
#> # … with 1,189 more rows, and 12 more variables: arr_delay <dbl>,
#> #   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>, dest <chr>, …
```

With mutate()

As the names suggest, the summary functions are typically paired with `summarize()`. However, because of the recycling rules we discussed in "Arithmetic and Recycling Rules" on page 225, they can also be usefully paired with `mutate()`, particularly when you want do some sort of group standardization. For example:

`x / sum(x)`
: Calculates the proportion of a total.

`(x - mean(x)) / sd(x)`
: Computes a Z-score (standardized to mean 0 and standard deviation 1).

`(x - min(x)) / (max(x) - min(x))`
: Standardizes to range [0, 1].

`x / first(x)`
: Computes an index based on the first observation.

Exercises

1. Brainstorm at least five ways to assess the typical delay characteristics of a group of flights. When is `mean()` useful? When is `median()` useful? When might you want to use something else? Should you use arrival delay or departure delay? Why might you want to use data from `planes`?
2. Which destinations show the greatest variation in air speed?
3. Create a plot to further explore the adventures of EGE. Can you find any evidence that the airport moved locations? Can you find another variable that might explain the difference?

Summary

You're already familiar with many tools for working with numbers, and after reading this chapter you now know how to use them in R. You also learned a handful of useful general transformations that are commonly, but not exclusively, applied to numeric vectors such as ranks and offsets. Finally, you worked through a number of numeric summaries and discussed a few of the statistical challenges that you should consider.

Over the next two chapters, we'll dive into working with strings with the stringr package. Strings are a big topic, so they get two chapters, one on the fundamentals of strings and one on regular expressions.

CHAPTER 14
Strings

Introduction

So far, you've used a bunch of strings without learning much about the details. Now it's time to dive into them, learn what makes strings tick, and master some of the powerful string manipulation tools you have at your disposal.

We'll begin with the details of creating strings and character vectors. You'll then dive into creating strings from data, then the opposite: extracting strings from data. We'll then discuss tools that work with individual letters. The chapter finishes with functions that work with individual letters and a brief discussion of where your expectations from English might steer you wrong when working with other languages.

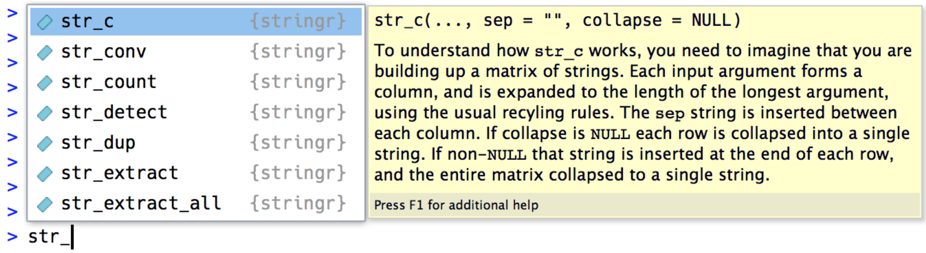
We'll keep working with strings in the next chapter, where you'll learn more about the power of regular expressions.

Prerequisites

In this chapter, we'll use functions from the stringr package, which is part of the core tidyverse. We'll also use the babynames data since it provides some fun strings to manipulate.

```
library(tidyverse)
library(babynames)
```

You can quickly tell when you're using a stringr function because all stringr functions start with `str_`. This is particularly useful if you use RStudio because typing `str_` will trigger autocomplete, allowing you to jog your memory of the available functions.



```
str_c            {stringr}
str_conv         {stringr}
str_count        {stringr}
str_detect       {stringr}
str_dup          {stringr}
str_extract      {stringr}
str_extract_all  {stringr}
> str_
```

str_c(..., sep = "", collapse = NULL)

To understand how str_c works, you need to imagine that you are building up a matrix of strings. Each input argument forms a column, and is expanded to the length of the longest argument, using the usual recyling rules. The sep string is inserted between each column. If collapse is NULL each row is collapsed into a single string. If non-NULL that string is inserted at the end of each row, and the entire matrix collapsed to a single string.

Press F1 for additional help

Creating a String

We created strings in passing earlier in the book but didn't discuss the details. First, you can create a string using either single quotes (') or double quotes ("). There's no difference in behavior between the two, so in the interest of consistency, the tidyverse style guide recommends using ", unless the string contains multiple ".

```
string1 <- "This is a string"
string2 <- 'If I want to include a "quote" inside a string, I use single quotes'
```

If you forget to close a quote, you'll see +, the continuation prompt:

```
> "This is a string without a closing quote
+
+
+ HELP I'M STUCK IN A STRING
```

If this happens to you and you can't figure out which quote to close, press Escape to cancel and try again.

Escapes

To include a literal single or double quote in a string, you can use \ to "escape" it:

```
double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'"
```

So if you want to include a literal backslash in your string, you'll need to escape it: "\\":

```
backslash <- "\\"
```

Beware that the printed representation of a string is not the same as the string itself because the printed representation shows the escapes (in other words, when you print a string, you can copy and paste the output to re-create that string). To see the raw contents of the string, use str_view():[1]

[1] Or use the base R function writeLines().

```
x <- c(single_quote, double_quote, backslash)
x
#> [1] "'"  "\"" "\\"

str_view(x)
#> [1] │ '
#> [2] │ "
#> [3] │ \
```

Raw Strings

Creating a string with multiple quotes or backslashes gets confusing quickly. To illustrate the problem, let's create a string that contains the contents of the code block where we define the double_quote and single_quote variables:

```
tricky <- "double_quote <- \"\\\"\" # or '\"'
single_quote <- '\\'' # or \"'\""
str_view(tricky)
#> [1] │ double_quote <- "\"" # or '"'
#>     │ single_quote <- '\'' # or "'"
```

That's a lot of backslashes! (This is sometimes called leaning toothpick syndrome.) To eliminate the escaping, you can instead use a *raw string*:[2]

```
tricky <- r"(double_quote <- "\"" # or '"'
single_quote <- '\'' # or "'")"
str_view(tricky)
#> [1] │ double_quote <- "\"" # or '"'
#>     │ single_quote <- '\'' # or "'"
```

A raw string usually starts with r"(and finishes with)". But if your string contains)", you can instead use r"[]" or r"{}", and if that's still not enough, you can insert any number of dashes to make the opening and closing pairs unique, e.g., `r"--()--", `r"---()---", etc. Raw strings are flexible enough to handle any text.

Other Special Characters

As well as \", \', and \\, there are a handful of other special characters that may come in handy. The most common are \n, a new line, and \t, tab. You'll also sometimes see strings containing Unicode escapes that start with \u or \U. This is a way of writing non-English characters that work on all systems. You can see the complete list of other special characters in ?Quotes.

```
x <- c("one\ntwo", "one\ttwo", "\u00b5", "\U0001f604")
x
#> [1] "one\ntwo" "one\ttwo" "µ"        " □str_view(x)
#> [1] │ one
#>     │ two
#> [2] │ one{\t}two
```

[2] Available in R 4.0.0 and newer.

```
#> [3] │ µ
#> [4] │   伯
```

Note that `str_view()` uses a blue background for tabs to make them easier to spot. One of the challenges of working with text is that there's a variety of ways that whitespace can end up in the text, so this background helps you recognize that something strange is going on.

Exercises

1. Create strings that contain the following values:

   a. `He said "That's amazing!"`

   b. `\a\b\c\d`

   c. `\\\\\\`

2. Create the following string in your R session and print it. What happens to the special "\u00a0"? How does `str_view()` display it? Can you do a little Googling to figure out what this special character is?

   ```
   x <- "This\u00a0is\u00a0tricky"
   ```

Creating Many Strings from Data

Now that you've learned the basics of creating a string or two by "hand," we'll go into the details of creating strings from other strings. This will help you solve the common problem where you have some text you wrote that you want to combine with strings from a data frame. For example, you might combine "Hello" with a `name` variable to create a greeting. We'll show you how to do this with `str_c()` and `str_glue()` and how you can use them with `mutate()`. That naturally raises the question of what stringr functions you might use with `summarize()`, so we'll finish this section with a discussion of `str_flatten()`, which is a summary function for strings.

str_c()

`str_c()` takes any number of vectors as arguments and returns a character vector:

```
str_c("x", "y")
#> [1] "xy"
str_c("x", "y", "z")
#> [1] "xyz"
str_c("Hello ", c("John", "Susan"))
#> [1] "Hello John"  "Hello Susan"
```

`str_c()` is similar to the base `paste0()` but is designed to be used with `mutate()` by obeying the usual tidyverse rules for recycling and propagating missing values:

```
df <- tibble(name = c("Flora", "David", "Terra", NA))
df |> mutate(greeting = str_c("Hi ", name, "!"))
```

```
#> # A tibble: 4 × 2
#>   name  greeting
#>   <chr> <chr>
#> 1 Flora Hi Flora!
#> 2 David Hi David!
#> 3 Terra Hi Terra!
#> 4 <NA>  <NA>
```

If you want missing values to display in another way, use `coalesce()` to replace them. Depending on what you want, you might use it either inside or outside of `str_c()`:

```
df |>
  mutate(
    greeting1 = str_c("Hi ", coalesce(name, "you"), "!"),
    greeting2 = coalesce(str_c("Hi ", name, "!"), "Hi!")
  )
#> # A tibble: 4 × 3
#>   name  greeting1 greeting2
#>   <chr> <chr>     <chr>
#> 1 Flora Hi Flora! Hi Flora!
#> 2 David Hi David! Hi David!
#> 3 Terra Hi Terra! Hi Terra!
#> 4 <NA>  Hi you!   Hi!
```

str_glue()

If you are mixing many fixed and variable strings with `str_c()`, you'll notice that you type a lot of "s, making it hard to see the overall goal of the code. An alternative approach is provided by the glue package via `str_glue()`.[3] You give it a single string that has a special feature: anything inside `{}` will be evaluated like it's outside of the quotes:

```
df |> mutate(greeting = str_glue("Hi {name}!"))
#> # A tibble: 4 × 2
#>   name  greeting
#>   <chr> <glue>
#> 1 Flora Hi Flora!
#> 2 David Hi David!
#> 3 Terra Hi Terra!
#> 4 <NA>  Hi NA!
```

As you can see, `str_glue()` currently converts missing values to the string `"NA"`, unfortunately making it inconsistent with `str_c()`.

You also might wonder what happens if you need to include a regular { or } in your string. You're on the right track if you guess you'll need to escape it somehow. The trick is that glue uses a slightly different escaping technique: instead of prefixing with a special character like \, you double up the special characters:

```
df |> mutate(greeting = str_glue("{{Hi {name}!}}"))
#> # A tibble: 4 × 2
```

[3] If you're not using stringr, you can also access it directly with `glue::glue()`.

```
#>   name  greeting
#>   <chr> <glue>
#> 1 Flora {Hi Flora!}
#> 2 David {Hi David!}
#> 3 Terra {Hi Terra!}
#> 4 <NA>  {Hi NA!}
```

str_flatten()

str_c() and str_glue() work well with mutate() because their output is the same length as their inputs. What if you want a function that works well with summarize(), i.e., something that always returns a single string? That's the job of str_flatten():[4] it takes a character vector and combines each element of the vector into a single string:

```
str_flatten(c("x", "y", "z"))
#> [1] "xyz"
str_flatten(c("x", "y", "z"), ", ")
#> [1] "x, y, z"
str_flatten(c("x", "y", "z"), ", ", last = ", and ")
#> [1] "x, y, and z"
```

This makes it work well with summarize():

```
df <- tribble(
  ~ name, ~ fruit,
  "Carmen", "banana",
  "Carmen", "apple",
  "Marvin", "nectarine",
  "Terence", "cantaloupe",
  "Terence", "papaya",
  "Terence", "mandarin"
)
df |>
  group_by(name) |>
  summarize(fruits = str_flatten(fruit, ", "))
#> # A tibble: 3 × 2
#>   name    fruits
#>   <chr>   <chr>
#> 1 Carmen  banana, apple
#> 2 Marvin  nectarine
#> 3 Terence cantaloupe, papaya, mandarin
```

Exercises

1. Compare and contrast the results of paste0() with str_c() for the following inputs:

   ```
   str_c("hi ", NA)
   str_c(letters[1:2], letters[1:3])
   ```

2. What's the difference between paste() and paste0()? How can you re-create the equivalent of paste() with str_c()?

[4] The base R equivalent is paste() used with the collapse argument.

3. Convert the following expressions from `str_c()` to `str_glue()` or vice versa:
   a. `str_c("The price of ", food, " is ", price)`
   b. `str_glue("I'm {age} years old and live in {country}")`
   c. `str_c("\\section{", title, "}")`

Extracting Data from Strings

It's common for multiple variables to be crammed together into a single string. In this section, you'll learn how to use four tidyr functions to extract them:

- `df |> separate_longer_delim(col, delim)`
- `df |> separate_longer_position(col, width)`
- `df |> separate_wider_delim(col, delim, names)`
- `df |> separate_wider_position(col, widths)`

If you look closely, you can see there's a common pattern here: `separate_`, then `longer` or `wider`, then `_`, then `delim` or `position`. That's because these four functions are composed of two simpler primitives:

- Just like with `pivot_longer()` and `pivot_wider()`, `_longer` functions make the input data frame longer by creating new rows, and `_wider` functions make the input data frame wider by generating new columns.
- `delim` splits up a string with a delimiter like `", "` or `" "`; `position` splits at specified widths, like `c(3, 5, 2)`.

We'll return to the last member of this family, `separate_wider_regex()`, in Chapter 15. It's the most flexible of the `wider` functions, but you need to know something about regular expressions before you can use it.

The following two sections will give you the basic idea behind these separate functions, first separating into rows (which is a little simpler) and then separating into columns. We'll finish off by discussing the tools that the `wider` functions give you to diagnose problems.

Separating into Rows

Separating a string into rows tends to be most useful when the number of components varies from row to row. The most common case is requiring `separate_longer_delim()` to split based on a delimiter:

```
df1 <- tibble(x = c("a,b,c", "d,e", "f"))
df1 |>
```

```
  separate_longer_delim(x, delim = ",")
#> # A tibble: 6 × 1
#>   x
#>   <chr>
#> 1 a
#> 2 b
#> 3 c
#> 4 d
#> 5 e
#> 6 f
```

It's rarer to see `separate_longer_position()` in the wild, but some older datasets do use a compact format where each character is used to record a value:

```
df2 <- tibble(x = c("1211", "131", "21"))
df2 |>
  separate_longer_position(x, width = 1)
#> # A tibble: 9 × 1
#>   x
#>   <chr>
#> 1 1
#> 2 2
#> 3 1
#> 4 1
#> 5 1
#> 6 3
#> # … with 3 more rows
```

Separating into Columns

Separating a string into columns tends to be most useful when there are a fixed number of components in each string, and you want to spread them into columns. They are slightly more complicated than their `longer` equivalents because you need to name the columns. For example, in the following dataset, `x` is made up of a code, an edition number, and a year, separated by `"."`. To use `separate_wider_delim()`, we supply the delimiter and the names in two arguments:

```
df3 <- tibble(x = c("a10.1.2022", "b10.2.2011", "e15.1.2015"))
df3 |>
  separate_wider_delim(
    x,
    delim = ".",
    names = c("code", "edition", "year")
  )
#> # A tibble: 3 × 3
#>   code  edition year
#>   <chr> <chr>   <chr>
#> 1 a10   1       2022
#> 2 b10   2       2011
#> 3 e15   1       2015
```

If a specific piece is not useful, you can use an `NA` name to omit it from the results:

```
df3 |>
  separate_wider_delim(
    x,
    delim = ".",
    names = c("code", NA, "year")
  )
#> # A tibble: 3 × 2
#>   code  year
#>   <chr> <chr>
#> 1 a10   2022
#> 2 b10   2011
#> 3 e15   2015
```

separate_wider_position() works a little differently because you typically want to specify the width of each column. So you give it a named integer vector, where the name gives the name of the new column, and the value is the number of characters it occupies. You can omit values from the output by not naming them:

```
df4 <- tibble(x = c("202215TX", "202122LA", "202325CA"))
df4 |>
  separate_wider_position(
    x,
    widths = c(year = 4, age = 2, state = 2)
  )
#> # A tibble: 3 × 3
#>   year  age   state
#>   <chr> <chr> <chr>
#> 1 2022  15    TX
#> 2 2021  22    LA
#> 3 2023  25    CA
```

Diagnosing Widening Problems

separate_wider_delim()[5] requires a fixed and known set of columns. What happens if some of the rows don't have the expected number of pieces? There are two possible problems, too few or too many pieces, so separate_wider_delim() provides two arguments to help: too_few and too_many. Let's first look at the too_few case with the following sample dataset:

```
df <- tibble(x = c("1-1-1", "1-1-2", "1-3", "1-3-2", "1"))

df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z")
  )
#> Error in `separate_wider_delim()`:
#> ! Expected 3 pieces in each element of `x`.
#> ! 2 values were too short.
#> ℹ Use `too_few = "debug"` to diagnose the problem.
#> ℹ Use `too_few = "align_start"/"align_end"` to silence this message.
```

[5] The same principles apply to separate_wider_position() and separate_wider_regex().

You'll notice that we get an error, but the error gives us some suggestions on how you might proceed. Let's start by debugging the problem:

```
debug <- df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_few = "debug"
  )
#> Warning: Debug mode activated: adding variables `x_ok`, `x_pieces`, and
#> `x_remainder`.
debug
#> # A tibble: 5 × 6
#>   x     y     z     x_ok  x_pieces x_remainder
#>   <chr> <chr> <chr> <lgl>    <int> <chr>
#> 1 1-1-1 1     1     TRUE         3 ""
#> 2 1-1-2 1     2     TRUE         3 ""
#> 3 1-3   3     <NA>  FALSE        2 ""
#> 4 1-3-2 3     2     TRUE         3 ""
#> 5 1     <NA>  <NA>  FALSE        1 ""
```

When you use the debug mode, you get three extra columns added to the output: `x_ok`, `x_pieces`, and `x_remainder` (if you separate a variable with a different name, you'll get a different prefix). Here, `x_ok` lets you quickly find the inputs that failed:

```
debug |> filter(!x_ok)
#> # A tibble: 2 × 6
#>   x     y     z     x_ok  x_pieces x_remainder
#>   <chr> <chr> <chr> <lgl>    <int> <chr>
#> 1 1-3   3     <NA>  FALSE        2 ""
#> 2 1     <NA>  <NA>  FALSE        1 ""
```

`x_pieces` tells us how many pieces were found, compared to the expected three (the length of `names`). `x_remainder` isn't useful when there are too few pieces, but we'll see it again shortly.

Sometimes looking at this debugging information will reveal a problem with your delimiter strategy or suggest that you need to do more preprocessing before separating. In that case, fix the problem upstream and make sure to remove `too_few = "debug"` to ensure that new problems become errors.

In other cases, you may want to fill in the missing pieces with `NA`s and move on. That's the job of `too_few = "align_start"` and `too_few = "align_end"`, which allow you to control where the `NA`s should go:

```
df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_few = "align_start"
  )
#> # A tibble: 5 × 3
#>   x     y     z
#>   <chr> <chr> <chr>
#> 1 1     1     1
#> 2 1     1     2
#> 3 1     3     <NA>
#> 4 1     3     2
#> 5 1     <NA>  <NA>
```

The same principles apply if you have too many pieces:

```
df <- tibble(x = c("1-1-1", "1-1-2", "1-3-5-6", "1-3-2", "1-3-5-7-9"))

df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z")
  )
#> Error in `separate_wider_delim()`:
#> ! Expected 3 pieces in each element of `x`.
#> ! 2 values were too long.
#> ℹ Use `too_many = "debug"` to diagnose the problem.
#> ℹ Use `too_many = "drop"/"merge"` to silence this message.
```

But now, when we debug the result, you can see the purpose of `x_remainder`:

```
debug <- df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_many = "debug"
  )
#> Warning: Debug mode activated: adding variables `x_ok`, `x_pieces`, and
#> `x_remainder`.
debug |> filter(!x_ok)
#> # A tibble: 2 × 6
#>   x         y     z     x_ok  x_pieces x_remainder
#>   <chr>     <chr> <chr> <lgl>    <int> <chr>
#> 1 1-3-5-6   3     5     FALSE        4 -6
#> 2 1-3-5-7-9 3     5     FALSE        5 -7-9
```

You have a slightly different set of options for handling too many pieces: you can either silently "drop" any additional pieces or "merge" them all into the final column:

```
df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_many = "drop"
  )
#> # A tibble: 5 × 3
#>   x     y     z
#>   <chr> <chr> <chr>
#> 1 1     1     1
#> 2 1     1     2
#> 3 1     3     5
#> 4 1     3     2
#> 5 1     3     5
```

```
df |>
  separate_wider_delim(
    x,
    delim = "-",
    names = c("x", "y", "z"),
    too_many = "merge"
  )
#> # A tibble: 5 × 3
#>   x     y     z
#>   <chr> <chr> <chr>
#> 1 1     1     1
#> 2 1     1     2
#> 3 1     3     5-6
#> 4 1     3     2
#> 5 1     3     5-7-9
```

Letters

In this section, we'll introduce you to functions that allow you to work with the individual letters within a string. You'll learn how to find the length of a string, extract substrings, and handle long strings in plots and tables.

Length

`str_length()` tells you the number of letters in the string:

```
str_length(c("a", "R for data science", NA))
#> [1]  1 18 NA
```

You could use this with `count()` to find the distribution of lengths of US baby names and then with `filter()` to look at the longest names, which happen to have 15 letters:[6]

```
babynames |>
  count(length = str_length(name), wt = n)
#> # A tibble: 14 × 2
#>   length        n
#>    <int>    <int>
#> 1      2   338150
#> 2      3  8589596
#> 3      4 48506739
#> 4      5 87011607
#> 5      6 90749404
#> 6      7 72120767
#> # … with 8 more rows

babynames |>
  filter(str_length(name) == 15) |>
  count(name, wt = n, sort = TRUE)
#> # A tibble: 34 × 2
```

[6] Looking at these entries, we'd guess that the babynames data drops spaces or hyphens and truncates after 15 letters.

```
#>   name                n
#>   <chr>           <int>
#> 1 Franciscojavier   123
#> 2 Christopherjohn   118
#> 3 Johnchristopher   118
#> 4 Christopherjame   108
#> 5 Christophermich    52
#> 6 Ryanchristopher    45
#> # … with 28 more rows
```

Subsetting

You can extract parts of a string using `str_sub(string, start, end)`, where `start` and `end` are the positions where the substring should start and end. The `start` and `end` arguments are inclusive, so the length of the returned string will be `end - start + 1`:

```
x <- c("Apple", "Banana", "Pear")
str_sub(x, 1, 3)
#> [1] "App" "Ban" "Pea"
```

You can use negative values to count back from the end of the string: -1 is the last character, -2 is the second to last character, etc.

```
str_sub(x, -3, -1)
#> [1] "ple" "ana" "ear"
```

Note that `str_sub()` won't fail if the string is too short: it will just return as much as possible:

```
str_sub("a", 1, 5)
#> [1] "a"
```

We could use `str_sub()` with `mutate()` to find the first and last letters of each name:

```
babynames |>
  mutate(
    first = str_sub(name, 1, 1),
    last = str_sub(name, -1, -1)
  )
#> # A tibble: 1,924,665 × 7
#>    year sex   name          n   prop first last
#>   <dbl> <chr> <chr>     <int>  <dbl> <chr> <chr>
#> 1  1880 F     Mary       7065 0.0724 M     y
#> 2  1880 F     Anna       2604 0.0267 A     a
#> 3  1880 F     Emma       2003 0.0205 E     a
#> 4  1880 F     Elizabeth  1939 0.0199 E     h
#> 5  1880 F     Minnie     1746 0.0179 M     e
#> 6  1880 F     Margaret   1578 0.0162 M     t
#> # … with 1,924,659 more rows
```

Exercises

1. When computing the distribution of the length of baby names, why did we use `wt = n`?

2. Use `str_length()` and `str_sub()` to extract the middle letter from each baby name. What will you do if the string has an even number of characters?
3. Are there any major trends in the length of baby names over time? What about the popularity of first and last letters?

Non-English Text

So far, we've focused on English language text, which is particularly easy to work with for two reasons. First, the English alphabet is relatively simple: there are just 26 letters. Second (and maybe more important), the computing infrastructure we use today was predominantly designed by English speakers. Unfortunately, we don't have room for a full treatment of non-English languages. Still, we wanted to draw your attention to some of the biggest challenges you might encounter: encoding, letter variations, and locale-dependent functions.

Encoding

When working with non-English text, the first challenge is often the *encoding*. To understand what's going on, we need to dive into 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 of these six hexadecimal numbers represents one letter: `48` is H, `61` is a, and so on. The mapping from hexadecimal number to character is 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 aren't so easy for languages other than English. In the early days of computing, there were many competing standards for encoding non-English characters. For example, there were two different encodings for Europe: Latin1 (aka ISO-8859-1) was used for Western European languages, and Latin2 (aka ISO-8859-2) was used for Central 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 and many extra symbols like emojis.

readr uses UTF-8 everywhere. This is a good default but will fail for data produced by older systems that don't use UTF-8. If this happens, 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, here are two inline CSVs with unusual encodings:[7]

```
x1 <- "text\nEl Ni\xf1o was particularly bad this year"
read_csv(x1)
#> # A tibble: 1 × 1
#>   text
#>   <chr>
#> 1 "El Ni\xf1o was particularly bad this year"

x2 <- "text\n\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
read_csv(x2)
#> # A tibble: 1 × 1
#>   text
#>   <chr>
#> 1 "\x82\xb1\x82\xf1\x82\xc9\x82\xbf\x82\xcd"
```

To read these correctly, you specify the encoding via the `locale` argument:

```
read_csv(x1, locale = locale(encoding = "Latin1"))
#> # A tibble: 1 × 1
#>   text
#>   <chr>
#> 1 El Niño was particularly bad this year

read_csv(x2, locale = locale(encoding = "Shift-JIS"))
#> # A tibble: 1 × 1
#>   text
#>   <chr>
#> 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 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.

Encodings are a rich and complex topic; we've only scratched the surface here. If you'd like to learn more, we recommend reading the detailed explanation.

Letter Variations

Working in languages with accents poses a significant challenge when determining the position of letters (e.g., with `str_length()` and `str_sub()`) as accented letters might be encoded as a single individual character (e.g., ü) or as two characters by combining an unaccented letter (e.g., u) with a diacritic mark (e.g., ¨). For example, this code shows two ways of representing ü that look identical:

[7] Here I'm using the special \x to encode binary data directly into a string.

```
u <- c("\u00fc", "u\u0308")
str_view(u)
#> [1] │ ü
#> [2] │ ü
```

But both strings differ in length, and their first characters are different:

```
str_length(u)
#> [1] 1 2
str_sub(u, 1, 1)
#> [1] "ü" "u"
```

Finally, note that a comparison of these strings with == interprets these strings as different, while the handy `str_equal()` function in stringr recognizes that both have the same appearance:

```
u[[1]] == u[[2]]
#> [1] FALSE

str_equal(u[[1]], u[[2]])
#> [1] TRUE
```

Locale-Dependent Functions

Finally, there are a handful of stringr functions whose behavior depends on your *locale*. A locale is similar to a language but includes an optional region specifier to handle regional variations within a language. A locale is specified by a lowercase language abbreviation, optionally followed by a _ and an uppercase region identifier. For example, "en" is English, "en_GB" is British English, and "en_US" is American English. If you don't already know the code for your language, Wikipedia has a good list, and you can see which are supported in stringr by looking at `stringi::stri_locale_list()`.

Base R string functions automatically use the locale set by your operating system. This means that base R string functions do what you expect for your language, but your code might work differently if you share it with someone who lives in a different country. To avoid this problem, stringr defaults to English rules by using the "en" locale and requires you to specify the `locale` argument to override it. Fortunately, there are two sets of functions where the locale really matters: changing case and sorting.

The rules for changing cases differ among languages. For example, Turkish has two i's: with and without a dot. Since they're two distinct letters, they're capitalized differently:

```
str_to_upper(c("i", "ı"))
#> [1] "I" "I"
str_to_upper(c("i", "ı"), locale = "tr")
#> [1] "İ" "I"
```

Sorting strings depends on the order of the alphabet, and the order of the alphabet is not the same in every language![8] Here's an example: in Czech, "ch" is a compound letter that appears after h in the alphabet.

```
str_sort(c("a", "c", "ch", "h", "z"))
#> [1] "a"  "c"  "ch" "h"  "z"
str_sort(c("a", "c", "ch", "h", "z"), locale = "cs")
#> [1] "a"  "c"  "h"  "ch" "z"
```

This also comes up when sorting strings with `dplyr::arrange()`, which is why it also has a `locale` argument.

Summary

In this chapter, you learned about some of the power of the stringr package such as how to create, combine, and extract strings, and about some of the challenges you might face with non-English strings. Now it's time to learn one of the most important and powerful tools for working with strings: regular expressions. Regular expressions are a concise but expressive language for describing patterns within strings and are the topic of the next chapter.

[8] Sorting in languages that don't have an alphabet, like Chinese, is more complicated still.

CHAPTER 15

Regular Expressions

Introduction

In Chapter 14, you learned a whole bunch of useful functions for working with strings. This chapter will focus on functions that use *regular expressions*, a concise and powerful language for describing patterns within strings. The term *regular expression* is a bit of a mouthful, so most people abbreviate it to *regex*[1] or *regexp*.

The chapter starts with the basics of regular expressions and the most useful stringr functions for data analysis. We'll then expand your knowledge of patterns and cover seven important new topics (escaping, anchoring, character classes, shorthand classes, quantifiers, precedence, and grouping). Next, we'll talk about some of the other types of patterns that stringr functions can work with and the various "flags" that allow you to tweak the operation of regular expressions. We'll finish with a survey of other places in the tidyverse and base R where you might use regexes.

Prerequisites

In this chapter, we'll use regular expression functions from stringr and tidyr, both core members of the tidyverse, as well as data from the babynames package:

```
library(tidyverse)
library(babynames)
```

Through this chapter, we'll use a mix of simple inline examples so you can get the basic idea, the baby names data, and three character vectors from stringr:

- `fruit` contains the names of 80 fruits.

[1] You can pronounce it with either a hard-g ("reg-x") or a soft-g ("rej-x").

- `words` contains 980 common English words.
- `sentences` contains 720 short sentences.

Pattern Basics

We'll use `str_view()` to learn how regex patterns work. We used `str_view()` in the previous chapter to better understand a string versus its printed representation, and now we'll use it with its second argument, a regular expression. When this is supplied, `str_view()` will show only the elements of the string vector that match, surrounding each match with <> and, where possible, highlighting the match in blue.

The simplest patterns consist of letters and numbers that match those characters exactly:

```
str_view(fruit, "berry")
#>  [6] │ bil<berry>
#>  [7] │ black<berry>
#> [10] │ blue<berry>
#> [11] │ boysen<berry>
#> [19] │ cloud<berry>
#> [21] │ cran<berry>
#> ... and 8 more
```

Letters and numbers match exactly and are called *literal characters*. Most punctuation characters, like `.`, `+`, `*`, `[`, `]`, and `?`, have special meanings[2] and are called *metacharacters*. For example, `.` will match any character,[3] so `"a."` will match any string that contains an "a" followed by another character:

```
str_view(c("a", "ab", "ae", "bd", "ea", "eab"), "a.")
#> [2] │ <ab>
#> [3] │ <ae>
#> [6] │ e<ab>
```

Or we could find all the fruits that contain an "a," followed by three letters, followed by an "e":

```
str_view(fruit, "a...e")
#>  [1] │ <apple>
#>  [7] │ bl<ackbe>rry
#> [48] │ mand<arine>
#> [51] │ nect<arine>
#> [62] │ pine<apple>
#> [64] │ pomegr<anate>
#> ... and 2 more
```

Quantifiers control how many times a pattern can match:

[2] You'll learn how to escape these special meanings in "Escaping" on page 269.

[3] Well, any character apart from `\n`.

- ? makes a pattern optional (i.e., it matches 0 or 1 times).
- + lets a pattern repeat (i.e., it matches at least once).
- * lets a pattern be optional or repeat (i.e., it matches any number of times, including 0).

```
# ab? matches an "a", optionally followed by a "b".
str_view(c("a", "ab", "abb"), "ab?")
#> [1] │ <a>
#> [2] │ <ab>
#> [3] │ <ab>b

# ab+ matches an "a", followed by at least one "b".
str_view(c("a", "ab", "abb"), "ab+")
#> [2] │ <ab>
#> [3] │ <abb>

# ab* matches an "a", followed by any number of "b"s.
str_view(c("a", "ab", "abb"), "ab*")
#> [1] │ <a>
#> [2] │ <ab>
#> [3] │ <abb>
```

Character classes are defined by `[]` and let you match a set of characters; e.g., `[abcd]` matches "a", "b", "c", or "d." You can also invert the match by starting with `^`: `[^abcd]` matches anything *except* "a", "b", "c", or "d." We can use this idea to find the words containing an "x" surrounded by vowels or a "y" surrounded by consonants:

```
str_view(words, "[aeiou]x[aeiou]")
#> [284] │ <exa>ct
#> [285] │ <exa>mple
#> [288] │ <exe>rcise
#> [289] │ <exi>st
str_view(words, "[^aeiou]y[^aeiou]")
#> [836] │ <sys>tem
#> [901] │ <typ>e
```

You can use *alternation*, `|`, to pick between one or more alternative patterns. For example, the following patterns look for fruits containing "apple," "melon," or "nut" or a repeated vowel:

```
str_view(fruit, "apple|melon|nut")
#>  [1] │ <apple>
#> [13] │ canary <melon>
#> [20] │ coco<nut>
#> [52] │ <nut>
#> [62] │ pine<apple>
#> [72] │ rock <melon>
#> ... and 1 more
str_view(fruit, "aa|ee|ii|oo|uu")
#>  [9] │ bl<oo>d orange
#> [33] │ g<oo>seberry
#> [47] │ lych<ee>
#> [66] │ purple mangost<ee>n
```

Regular expressions are very compact and use a lot of punctuation characters, so they can seem overwhelming and hard to read at first. Don't worry: you'll get better with practice, and simple patterns will soon become second nature. Let's kick off that process by practicing with some useful stringr functions.

Key Functions

Now that you understand the basics of regular expressions, let's use them with some stringr and tidyr functions. In the following section, you'll learn how to detect the presence or absence of a match, how to count the number of matches, how to replace a match with fixed text, and how to extract text using a pattern.

Detect Matches

`str_detect()` returns a logical vector that is `TRUE` if the pattern matches an element of the character vector and `FALSE` otherwise:

```
str_detect(c("a", "b", "c"), "[aeiou]")
#> [1]  TRUE FALSE FALSE
```

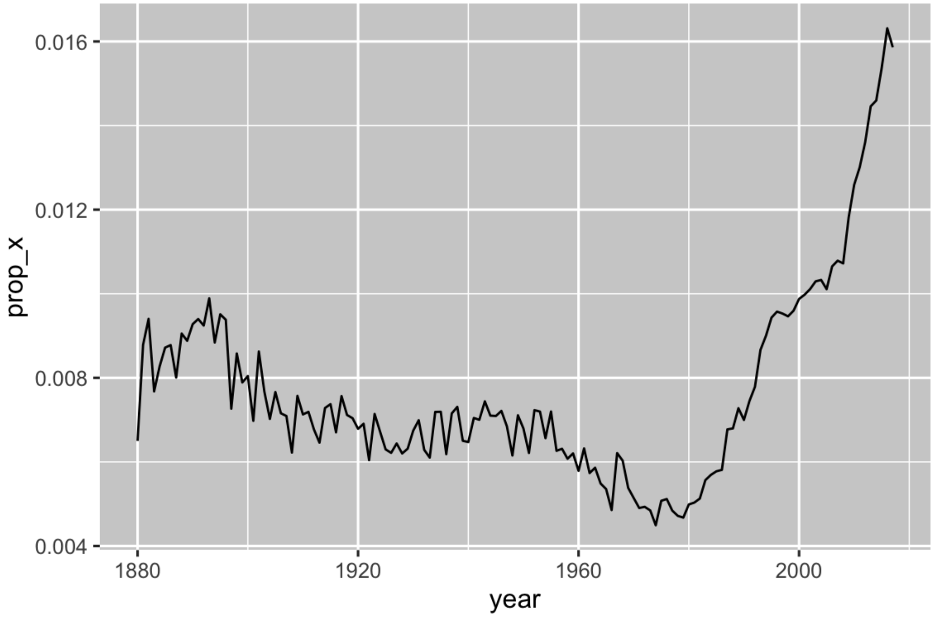
Since `str_detect()` returns a logical vector of the same length as the initial vector, it pairs well with `filter()`. For example, this code finds all the most popular names containing a lowercase "x":

```
babynames |>
  filter(str_detect(name, "x")) |>
  count(name, wt = n, sort = TRUE)
#> # A tibble: 974 × 2
#>   name           n
#>   <chr>      <int>
#> 1 Alexander 665492
#> 2 Alexis    399551
#> 3 Alex      278705
#> 4 Alexandra 232223
#> 5 Max       148787
#> 6 Alexa     123032
#> # … with 968 more rows
```

We can also use `str_detect()` with `summarize()` by pairing it with `sum()` or `mean()`: `sum(str_detect(x, pattern))` tells you the number of observations that match, and `mean(str_detect(x, pattern))` tells you the proportion that match. For example, the following snippet computes and visualizes the proportion of baby names[4] that contain "x," broken down by year. It looks like they've radically increased in popularity lately!

[4] This gives us the proportion of *names* that contain an "x"; if you wanted the proportion of babies with a name containing an x, you'd need to perform a weighted mean.

```
babynames |>
  group_by(year) |>
  summarize(prop_x = mean(str_detect(name, "x"))) |>
  ggplot(aes(x = year, y = prop_x)) +
  geom_line()
```



There are two functions that are closely related to `str_detect()`: `str_subset()` and `str_which()`. `str_subset()` returns a character vector containing only the strings that match. `str_which()` returns an integer vector giving the positions of the strings that match.

Count Matches

The next step up in complexity from `str_detect()` is `str_count()`: rather than a true or false, it tells you how many matches there are in each string.

```
x <- c("apple", "banana", "pear")
str_count(x, "p")
#> [1] 2 0 1
```

Note that each match starts at the end of the previous match; i.e., regex matches never overlap. For example, in `"abababa"`, how many times will the pattern `"aba"` match? Regular expressions say two, not three:

```
str_count("abababa", "aba")
#> [1] 2
str_view("abababa", "aba")
#> [1] │ <aba>b<aba>
```

It's natural to use str_count() with mutate(). The following example uses str_count() with character classes to count the number of vowels and consonants in each name:

```
babynames |>
  count(name) |>
  mutate(
    vowels = str_count(name, "[aeiou]"),
    consonants = str_count(name, "[^aeiou]")
  )
#> # A tibble: 97,310 × 4
#>   name          n vowels consonants
#>   <chr>     <int>  <int>      <int>
#> 1 Aaban        10      2          3
#> 2 Aabha         5      2          3
#> 3 Aabid         2      2          3
#> 4 Aabir         1      2          3
#> 5 Aabriella     5      4          5
#> 6 Aada          1      2          2
#> # … with 97,304 more rows
```

If you look closely, you'll notice that there's something off with our calculations: "Aaban" contains three a's, but our summary reports only two vowels. That's because regular expressions are case sensitive. There are three ways we could fix this:

- Add the uppercase vowels to the character class: str_count(name, "[aeiouAEIOU]").
- Tell the regular expression to ignore case: str_count(name, regex("[aeiou]", ignore_case = TRUE)). We'll talk about more in "Regex Flags" on page 275.
- Use str_to_lower() to convert the names to lowercase: str_count(str_to_lower(name), "[aeiou]").

This variety of approaches is pretty typical when working with strings—there are often multiple ways to reach your goal, either by making your pattern more complicated or by doing some preprocessing on your string. If you get stuck trying one approach, it can often be useful to switch gears and tackle the problem from a different perspective.

Since we're applying two functions to the name, I think it's easier to transform it first:

```
babynames |>
  count(name) |>
  mutate(
    name = str_to_lower(name),
    vowels = str_count(name, "[aeiou]"),
    consonants = str_count(name, "[^aeiou]")
  )
#> # A tibble: 97,310 × 4
#>   name          n vowels consonants
#>   <chr>     <int>  <int>      <int>
#> 1 aaban        10      3          2
#> 2 aabha         5      3          2
#> 3 aabid         2      3           2
#> 4 aabir         1      3           2
#> 5 aabriella     5      5           4
#> 6 aada          1      3           1
#> # … with 97,304 more rows
```

Replace Values

As well as detecting and counting matches, we can also modify them with `str_replace()` and `str_replace_all()`. `str_replace()` replaces the first match, and as the name suggests, `str_replace_all()` replaces all matches:

```
x <- c("apple", "pear", "banana")
str_replace_all(x, "[aeiou]", "-")
#> [1] "-ppl-"  "p--r"   "b-n-n-"
```

`str_remove()` and `str_remove_all()` are handy shortcuts for `str_replace(x, pattern, "")`:

```
x <- c("apple", "pear", "banana")
str_remove_all(x, "[aeiou]")
#> [1] "ppl" "pr"  "bnn"
```

These functions are naturally paired with `mutate()` when doing data cleaning, and you'll often apply them repeatedly to peel off layers of inconsistent formatting.

Extract Variables

The last function we'll discuss uses regular expressions to extract data out of one column into one or more new columns: `separate_wider_regex()`. It's a peer of the `separate_wider_position()` and `separate_wider_delim()` functions that you learned about in "Separating into Columns" on page 250. These functions live in tidyr because they operate on (columns of) data frames, rather than individual vectors.

Let's create a simple dataset to show how it works. Here we have some data derived from `babynames` where we have the name, gender, and age of a bunch of people in a rather weird format:[5]

```
df <- tribble(
  ~str,
  "<Sheryl>-F_34",
  "<Kisha>-F_45",
  "<Brandon>-N_33",
  "<Sharon>-F_38",
  "<Penny>-F_58",
  "<Justin>-M_41",
  "<Patricia>-F_84",
)
```

[5] We wish we could reassure you that you'd never see something this weird in real life, but unfortunately over the course of your career you're likely to see much weirder!

To extract this data using separate_wider_regex() we just need to construct a sequence of regular expressions that match each piece. If we want the contents of that piece to appear in the output, we give it a name:

```
df |>
  separate_wider_regex(
    str,
    patterns = c(
      "<",
      name = "[A-Za-z]+",
      ">-",
      gender = ".", "_",
      age = "[0-9]+"
    )
  )
#> # A tibble: 7 × 3
#>   name    gender age
#>   <chr>   <chr>  <chr>
#> 1 Sheryl  F      34
#> 2 Kisha   F      45
#> 3 Brandon N      33
#> 4 Sharon  F      38
#> 5 Penny   F      58
#> 6 Justin  M      41
#> # … with 1 more row
```

If the match fails, you can use `too_short = "debug"` to figure out what went wrong, just like separate_wider_delim() and separate_wider_position().

Exercises

1. What baby name has the most vowels? What name has the highest proportion of vowels? (Hint: What is the denominator?)
2. Replace all forward slashes in "a/b/c/d/e" with backslashes. What happens if you attempt to undo the transformation by replacing all backslashes with forward slashes? (We'll discuss the problem very soon.)
3. Implement a simple version of str_to_lower() using str_replace_all().
4. Create a regular expression that will match telephone numbers as commonly written in your country.

Pattern Details

Now that you understand the basics of the pattern language and how to use it with some stringr and tidyr functions, it's time to dig into more of the details. First, we'll start with *escaping*, which allows you to match metacharacters that would otherwise be treated specially. Next, you'll learn about *anchors*, which allow you to match the start or end of the string. Then, you'll more learn about *character classes* and their shortcuts, which allow you to match any character from a set. Next, you'll learn the final details of *quantifiers*, which control how many times a pattern can match. Then, we have to cover the important (but complex) topic of *operator precedence* and parentheses. And we'll finish off with some details of *grouping* components of the pattern.

The terms we use here are the technical names for each component. They're not always the most evocative of their purpose, but it's helpful to know the correct terms if you later want to google for more details.

Escaping

To match a literal `.`, you need an *escape*, which tells the regular expression to match metacharacters[6] literally. Like strings, regexps use the backslash for escaping. So, to match a `.`, you need the regexp `\.`. Unfortunately, this creates a problem. We use strings to represent regular expressions, and \ is also used as an escape symbol in strings. So to create the regular expression `\.`, we need the string `"\\."`, as the following example shows:

```
# To create the regular expression \., we need to use \\.
dot <- "\\."

# But the expression itself only contains one \
str_view(dot)
#> [1] │ \.

# And this tells R to look for an explicit .
str_view(c("abc", "a.c", "bef"), "a\\.c")
#> [2] │ <a.c>
```

In this book, we'll usually write regular expression without quotes, like `\.`. If we need to emphasize what you'll actually type, we'll surround it with quotes and add extra escapes, like `"\\."`.

If \ is used as an escape character in regular expressions, how do you match a literal \? Well, you need to escape it, creating the regular expression `\\`. To create that regular expression, you need to use a string, which also needs to escape \. That means to match a literal \ you need to write `"\\\\"`—you need four backslashes to match one!

```
x <- "a\\b"
str_view(x)
#> [1] │ a\b
str_view(x, "\\\\")
#> [1] │ a<\>b
```

Alternatively, you might find it easier to use the raw strings you learned about in "Raw Strings" on page 245. That lets you avoid one layer of escaping:

[6] The complete set of metacharacters is `.^$\|*+?{}[]()`.

```
str_view(x, r"{\\}")
#> [1] │ a<\>b
```

If you're trying to match a literal `.`, `$`, `|`, `*`, `+`, `?`, `{`, `}`, `(`, `)`, there's an alternative to using a backslash escape. You can use a character class: `[.]`, `[$]`, `[|]`, ... all match the literal values:

```
str_view(c("abc", "a.c", "a*c", "a c"), "a[.]c")
#> [2] │ <a.c>
str_view(c("abc", "a.c", "a*c", "a c"), ".[*]c")
#> [3] │ <a*c>
```

Anchors

By default, regular expressions will match any part of a string. If you want to match at the start or end you need to *anchor* the regular expression using `^` to match the start or `$` to match the end:

```
str_view(fruit, "^a")
#> [1] │ <a>pple
#> [2] │ <a>pricot
#> [3] │ <a>vocado
str_view(fruit, "a$")
#>  [4] │ banan<a>
#> [15] │ cherimoy<a>
#> [30] │ feijo<a>
#> [36] │ guav<a>
#> [56] │ papay<a>
#> [74] │ satsum<a>
```

It's tempting to think that `$` should match the start of a string, because that's how we write dollar amounts, but that's not what regular expressions want.

To force a regular expression to match only the full string, anchor it with both `^` and `$`:

```
str_view(fruit, "apple")
#>  [1] │ <apple>
#> [62] │ pine<apple>
str_view(fruit, "^apple$")
#> [1] │ <apple>
```

You can also match the boundary between words (i.e., the start or end of a word) with `\b`. This can be particularly useful when using RStudio's find and replace tool. For example, to find all uses of `sum()`, you can search for `\bsum\b` to avoid matching `summarize`, `summary`, `rowsum`, and so on:

```
x <- c("summary(x)", "summarize(df)", "rowsum(x)", "sum(x)")
str_view(x, "sum")
#> [1] │ <sum>mary(x)
#> [2] │ <sum>marize(df)
#> [3] │ row<sum>(x)
#> [4] │ <sum>(x)
str_view(x, "\\bsum\\b")
#> [4] │ <sum>(x)
```

When used alone, anchors will produce a zero-width match:

```
str_view("abc", c("$", "^", "\\b"))
#> [1] │ abc<>
#> [2] │ <>abc
#> [3] │ <>abc<>
```

This helps you understand what happens when you replace a standalone anchor:

```
str_replace_all("abc", c("$", "^", "\\b"), "--")
#> [1] "abc--"   "--abc"   "--abc--"
```

Character Classes

A *character class*, or character *set*, allows you to match any character in a set. As we discussed, you can construct your own sets with `[]`, where `[abc]` matches "a," "b," or "c" and `[^abc]` matches any character except "a," "b," or "c." Apart from `^` there are two other characters that have special meaning inside `[]`:

- `-` defines a range; e.g., `[a-z]` matches any lowercase letter, and `[0-9]` matches any number.
- `\` escapes special characters, so `[\^\-\]]` matches `^`, `-`, or `]`.

Here are a few examples:

```
x <- "abcd ABCD 12345 -!@#%."
str_view(x, "[abc]+")
#> [1] │ <abc>d ABCD 12345 -!@#%.
str_view(x, "[a-z]+")
#> [1] │ <abcd> ABCD 12345 -!@#%.
str_view(x, "[^a-z0-9]+")
#> [1] │ abcd< ABCD >12345< -!@#%.>

# You need an escape to match characters that are otherwise
# special inside of []
str_view("a-b-c", "[a-c]")
#> [1] │ <a>-<b>-<c>
str_view("a-b-c", "[a\\-c]")
#> [1] │ <a><->b<-><c>
```

Some character classes are used so commonly that they get their own shortcut. You've already seen `.`, which matches any character apart from a newline. There are three other particularly useful pairs:[7]

- `\d` matches any digit.
  `\D` matches anything that isn't a digit.

[7] Remember, to create a regular expression containing `\d` or `\s`, you'll need to escape the `\` for the string, so you'll type `"\\d"` or `"\\s"`.

- \s matches any whitespace (e.g., space, tab, newline).
  \S matches anything that isn't whitespace.
- \w matches any "word" character, i.e., letters and numbers.
  \W matches any "nonword" character.

The following code demonstrates the six shortcuts with a selection of letters, numbers, and punctuation characters:

```
x <- "abcd ABCD 12345 -!@#%."
str_view(x, "\\d+")
#> [1] │ abcd ABCD <12345> -!@#%.
str_view(x, "\\D+")
#> [1] │ <abcd ABCD >12345< -!@#%.>
str_view(x, "\\s+")
#> [1] │ abcd< >ABCD< >12345< >-!@#%.
str_view(x, "\\S+")
#> [1] │ <abcd> <ABCD> <12345> <-!@#%.>
str_view(x, "\\w+")
#> [1] │ <abcd> <ABCD> <12345> -!@#%.
str_view(x, "\\W+")
#> [1] │ abcd< >ABCD< >12345< -!@#%.>
```

Quantifiers

Quantifiers control how many times a pattern matches. In "Pattern Basics" on page 262 you learned about ? (0 or 1 matches), + (1 or more matches), and * (0 or more matches). For example, colou?r will match American or British spelling, \d+ will match one or more digits, and \s? will optionally match a single item of whitespace. You can also specify the number of matches precisely with {}:

- {n} matches exactly n times.
- {n,} matches at least n times.
- {n,m} matches between n and m times.

Operator Precedence and Parentheses

What does ab+ match? Does it match "a" followed by one or more "b"s, or does it match "ab" repeated any number of times? What does ^a|b$ match? Does it match the complete string a or the complete string b, or does it match a string starting with a or a string ending with b?

The answer to these questions is determined by operator precedence, similar to the PEMDAS or BEDMAS rules you might have learned in school. You know that a + b * c is equivalent to a + (b * c) not (a + b) * c because * has higher precedence and + has lower precedence: you compute * before +.

Similarly, regular expressions have their own precedence rules: quantifiers have high precedence, and alternation has low precedence, which means that ab+ is equivalent to a(b+), and ^a|b$ is equivalent to (^a)|(b$). Just like with algebra, you can use parentheses to override the usual order. But unlike algebra, you're unlikely to remember the precedence rules for regexes, so feel free to use parentheses liberally.

Grouping and Capturing

As well as overriding operator precedence, parentheses have another important effect: they create *capturing groups* that allow you to use subcomponents of the match.

The first way to use a capturing group is to refer to it within a match with a *back reference*: \1 refers to the match contained in the first parenthesis, \2 in the second parenthesis, and so on. For example, the following pattern finds all fruits that have a repeated pair of letters:

```
str_view(fruit, "(..)\\1")
#>  [4] │ b<anan>a
#> [20] │ <coco>nut
#> [22] │ <cucu>mber
#> [41] │ <juju>be
#> [56] │ <papa>ya
#> [73] │ s<alal> berry
```

This one finds all words that start and end with the same pair of letters:

```
str_view(words, "^(..).*\\1$")
#> [152] │ <church>
#> [217] │ <decide>
#> [617] │ <photograph>
#> [699] │ <require>
#> [739] │ <sense>
```

You can also use back references in str_replace(). For example, this code switches the order of the second and third words in sentences:

```
sentences |>
  str_replace("(\\w+) (\\w+) (\\w+)", "\\1 \\3 \\2") |>
  str_view()
#> [1] │ The canoe birch slid on the smooth planks.
#> [2] │ Glue sheet the to the dark blue background.
#> [3] │ It's to easy tell the depth of a well.
#> [4] │ These a days chicken leg is a rare dish.
#> [5] │ Rice often is served in round bowls.
#> [6] │ The of juice lemons makes fine punch.
#> ... and 714 more
```

If you want to extract the matches for each group, you can use str_match(). But str_match() returns a matrix, so it's not particularly easy to work with:[8]

[8] Mostly because we never discuss matrices in this book!

```
sentences |>
  str_match("the (\\w+) (\\w+)") |>
  head()
#>      [,1]                [,2]     [,3]
#> [1,] "the smooth planks" "smooth" "planks"
#> [2,] "the sheet to"      "sheet"  "to"
#> [3,] "the depth of"      "depth"  "of"
#> [4,] NA                  NA       NA
#> [5,] NA                  NA       NA
#> [6,] NA                  NA       NA
```

You could convert to a tibble and name the columns:

```
sentences |>
  str_match("the (\\w+) (\\w+)") |>
  as_tibble(.name_repair = "minimal") |>
  set_names("match", "word1", "word2")
#> # A tibble: 720 × 3
#>   match             word1  word2
#>   <chr>             <chr>  <chr>
#> 1 the smooth planks smooth planks
#> 2 the sheet to      sheet  to
#> 3 the depth of      depth  of
#> 4 <NA>              <NA>   <NA>
#> 5 <NA>              <NA>   <NA>
#> 6 <NA>              <NA>   <NA>
#> # … with 714 more rows
```

But then you've basically re-created your own version of separate_wider_regex(). Indeed, behind the scenes, separate_wider_regex() converts your vector of patterns to a single regex that uses grouping to capture the named components.

Occasionally, you'll want to use parentheses without creating matching groups. You can create a noncapturing group with (?:).

```
x <- c("a gray cat", "a grey dog")
str_match(x, "gr(e|a)y")
#>      [,1]   [,2]
#> [1,] "gray" "a"
#> [2,] "grey" "e"
str_match(x, "gr(?:e|a)y")
#>      [,1]
#> [1,] "gray"
#> [2,] "grey"
```

Exercises

1. How would you match the literal string "'\? How about "$^$"?
2. Explain why each of these patterns don't match a \: "\", "\\", "\\\".
3. Given the corpus of common words in stringr::words, create regular expressions that find all words that:
   a. Start with "y."
   b. Don't start with "y."

c. End with “x.”

d. Are exactly three letters long. (Don’t cheat by using `str_length()`!)

e. Have seven letters or more.

f. Contain a vowel-consonant pair.

g. Contain at least two vowel-consonant pairs in a row.

h. Only consist of repeated vowel-consonant pairs.

4. Create 11 regular expressions that match the British or American spellings for each of the following words: airplane/aeroplane, aluminum/aluminium, analog/analogue, ass/arse, center/centre, defense/defence, donut/doughnut, gray/grey, modeling/modelling, skeptic/sceptic, summarize/summarise. Try to make the shortest possible regex!

5. Switch the first and last letters in `words`. Which of those strings are still `words`?

6. Describe in words what these regular expressions match (read carefully to see if each entry is a regular expression or a string that defines a regular expression):

   a. `^.*$`

   b. `"\\{.+\\}"`

   c. `\d{4}-\d{2}-\d{2}`

   d. `"\\\\{4}"`

   e. `\..\..\..`

   f. `(.)\1\1`

   g. `"(..)\\1"`

7. Solve the beginner regexp crosswords.

Pattern Control

It’s possible to exercise extra control over the details of the match by using a pattern object instead of just a string. This allows you to control the so-called regex flags and match various types of fixed strings, as described next.

Regex Flags

A number of settings can be used to control the details of the regexp. These settings are often called *flags* in other programming languages. In stringr, you can use them by wrapping the pattern in a call to `regex()`. The most useful flag is probably `ignore_case = TRUE` because it allows characters to match either their uppercase or lowercase forms:

```
bananas <- c("banana", "Banana", "BANANA")
str_view(bananas, "banana")
#> [1] │ <banana>
str_view(bananas, regex("banana", ignore_case = TRUE))
#> [1] │ <banana>
#> [2] │ <Banana>
#> [3] │ <BANANA>
```

If you're doing a lot of work with multiline strings (i.e., strings that contain `\n`), `dotall` and `multiline` may also be useful:

- `dotall = TRUE` lets `.` match everything, including `\n`:

  ```
  x <- "Line 1\nLine 2\nLine 3"
  str_view(x, ".Line")
  str_view(x, regex(".Line", dotall = TRUE))
  #> [1] │ Line 1<
  #>     │ Line> 2<
  #>     │ Line> 3
  ```

- `multiline = TRUE` makes `^` and `$` match the start and end of each line rather than the start and end of the complete string:

  ```
  x <- "Line 1\nLine 2\nLine 3"
  str_view(x, "^Line")
  #> [1] │ <Line> 1
  #>     │ Line 2
  #>     │ Line 3
  str_view(x, regex("^Line", multiline = TRUE))
  #> [1] │ <Line> 1
  #>     │ <Line> 2
  #>     │ <Line> 3
  ```

Finally, if you're writing a complicated regular expression and you're worried you might not understand it in the future, you might try `comments = TRUE`. It tweaks the pattern language to ignore spaces and new lines, as well as everything after `#`. This allows you to use comments and whitespace to make complex regular expressions more understandable,[9] as in the following example:

```
phone <- regex(
  r"(
    \(?     # optional opening parens
    (\d{3}) # area code
    [)\-]?  # optional closing parens or dash
    \ ?     # optional space
    (\d{3}) # another three numbers
    [\ -]?  # optional space or dash
    (\d{4}) # four more numbers
  )",
  comments = TRUE
)

str_extract(c("514-791-8141", "(123) 456 7890", "123456"), phone)
#> [1] "514-791-8141"   "(123) 456 7890" NA
```

[9] `comments = TRUE` is particularly effective in combination with a raw string, as we use here.

If you're using comments and want to match a space, newline, or #, you'll need to escape it with \.

Fixed Matches

You can opt out of the regular expression rules by using `fixed()`:

```
str_view(c("", "a", "."), fixed("."))
#> [3] │ <.>
```

`fixed()` also gives you the ability to ignore case:

```
str_view("x X", "X")
#> [1] │ x <X>
str_view("x X", fixed("X", ignore_case = TRUE))
#> [1] │ <x> <X>
```

If you're working with non-English text, you will probably want `coll()` instead of `fixed()`, as it implements the full rules for capitalization as used by the `locale` you specify. See "Non-English Text" on page 256 for more details on locales.

```
str_view("i İ ı I", fixed("İ", ignore_case = TRUE))
#> [1] │ i <İ> ı I
str_view("i İ ı I", coll("İ", ignore_case = TRUE, locale = "tr"))
#> [1] │ <i> <İ> ı I
```

Practice

To put these ideas into practice, we'll solve a few semi-authentic problems next. We'll discuss three general techniques:

- Checking your work by creating simple positive and negative controls
- Combining regular expressions with Boolean algebra
- Creating complex patterns using string manipulation

Check Your Work

First, let's find all sentences that start with "The." Using the ^ anchor alone is not enough:

```
str_view(sentences, "^The")
#>  [1] │ <The> birch canoe slid on the smooth planks.
#>  [4] │ <The>se days a chicken leg is a rare dish.
#>  [6] │ <The> juice of lemons makes fine punch.
#>  [7] │ <The> box was thrown beside the parked truck.
#>  [8] │ <The> hogs were fed chopped corn and garbage.
#> [11] │ <The> boy was there when the sun rose.
#> ... and 271 more
```

That pattern also matches sentences starting with words like `They` or `These`. We need to make sure that the "e" is the last letter in the word, which we can do by adding a word boundary:

```
str_view(sentences, "^The\\b")
#>  [1] │ <The> birch canoe slid on the smooth planks.
#>  [6] │ <The> juice of lemons makes fine punch.
#>  [7] │ <The> box was thrown beside the parked truck.
#>  [8] │ <The> hogs were fed chopped corn and garbage.
#> [11] │ <The> boy was there when the sun rose.
#> [13] │ <The> source of the huge river is the clear spring.
#> ... and 250 more
```

What about finding all sentences that begin with a pronoun?

```
str_view(sentences, "^She|He|It|They\\b")
#>  [3] │ <It>'s easy to tell the depth of a well.
#> [15] │ <He>lp the woman get back to her feet.
#> [27] │ <He>r purse was full of useless trash.
#> [29] │ <It> snowed, rained, and hailed the same morning.
#> [63] │ <He> ran half way to the hardware store.
#> [90] │ <He> lay prone and hardly moved a limb.
#> ... and 57 more
```

A quick inspection of the results shows that we're getting some spurious matches. That's because we've forgotten to use parentheses:

```
str_view(sentences, "^(She|He|It|They)\\b")
#>   [3] │ <It>'s easy to tell the depth of a well.
#>  [29] │ <It> snowed, rained, and hailed the same morning.
#>  [63] │ <He> ran half way to the hardware store.
#>  [90] │ <He> lay prone and hardly moved a limb.
#> [116] │ <He> ordered peach pie with ice cream.
#> [127] │ <It> caught its hind paw in a rusty trap.
#> ... and 51 more
```

You might wonder how you might spot such a mistake if it didn't occur in the first few matches. A good technique is to create a few positive and negative matches and use them to test that your pattern works as expected:

```
pos <- c("He is a boy", "She had a good time")
neg <- c("Shells come from the sea", "Hadley said 'It's a great day'")

pattern <- "^(She|He|It|They)\\b"
str_detect(pos, pattern)
#> [1] TRUE TRUE
str_detect(neg, pattern)
#> [1] FALSE FALSE
```

It's typically much easier to come up with good positive examples than negative examples, because it takes a while before you're good enough with regular expressions to predict where your weaknesses are. Nevertheless, they're still useful: as you work on the problem, you can slowly accumulate a collection of your mistakes, ensuring that you never make the same mistake twice.

Boolean Operations

Imagine we want to find words that contain only consonants. One technique is to create a character class that contains all letters except for the vowels ([^aeiou]), then allow that to match any number of letters ([^aeiou]+), and then force it to match the whole string by anchoring to the beginning and the end (^[^aeiou]+$):

```
str_view(words, "^[^aeiou]+$")
#> [123] │ <by>
#> [249] │ <dry>
#> [328] │ <fly>
#> [538] │ <mrs>
#> [895] │ <try>
#> [952] │ <why>
```

But you can make this problem a bit easier by flipping the problem around. Instead of looking for words that contain only consonants, we could look for words that don't contain any vowels:

```
str_view(words[!str_detect(words, "[aeiou]")])
#> [1] │ by
#> [2] │ dry
#> [3] │ fly
#> [4] │ mrs
#> [5] │ try
#> [6] │ why
```

This is a useful technique whenever you're dealing with logical combinations, particularly those involving "and" or "not." For example, imagine if you want to find all words that contain "a" and "b." There's no "and" operator built in to regular expressions, so we have to tackle it by looking for all words that contain an "a" followed by a "b," or a "b" followed by an "a":

```
str_view(words, "a.*b|b.*a")
#>  [2] │ <ab>le
#>  [3] │ <ab>out
#>  [4] │ <ab>solute
#> [62] │ <availab>le
#> [66] │ <ba>by
#> [67] │ <ba>ck
#> ... and 24 more
```

It's simpler to combine the results of two calls to `str_detect()`:

```
words[str_detect(words, "a") & str_detect(words, "b")]
#>  [1] "able"      "about"     "absolute"  "available" "baby"      "back"
#>  [7] "bad"       "bag"       "balance"   "ball"      "bank"      "bar"
#> [13] "base"      "basis"     "bear"      "beat"      "beauty"    "because"
#> [19] "black"     "board"     "boat"      "break"     "brilliant" "britain"
#> [25] "debate"    "husband"   "labour"    "maybe"     "probable"  "table"
```

What if we wanted to see if there was a word that contains all vowels? If we did it with patterns, we'd need to generate 5! (120) different patterns:

```
words[str_detect(words, "a.*e.*i.*o.*u")]
# ...
words[str_detect(words, "u.*o.*i.*e.*a")]
```

It's much simpler to combine five calls to `str_detect()`:

```
words[
  str_detect(words, "a") &
  str_detect(words, "e") &
  str_detect(words, "i") &
  str_detect(words, "o") &
  str_detect(words, "u")
]
#> character(0)
```

In general, if you get stuck trying to create a single regexp that solves your problem, take a step back and think if you could break the problem down into smaller pieces, solving each challenge before moving onto the next one.

Creating a Pattern with Code

What if we wanted to find all `sentences` that mention a color? The basic idea is simple: we just combine alternation with word boundaries:

```
str_view(sentences, "\\b(red|green|blue)\\b")
#>   [2] │ Glue the sheet to the dark <blue> background.
#>  [26] │ Two <blue> fish swam in the tank.
#>  [92] │ A wisp of cloud hung in the <blue> air.
#> [148] │ The spot on the blotter was made by <green> ink.
#> [160] │ The sofa cushion is <red> and of light weight.
#> [174] │ The sky that morning was clear and bright <blue>.
#> ... and 20 more
```

But as the number of colors grows, it would quickly get tedious to construct this pattern by hand. Wouldn't it be nice if we could store the colors in a vector?

```
rgb <- c("red", "green", "blue")
```

Well, we can! We'd just need to create the pattern from the vector using `str_c()` and `str_flatten()`:

```
str_c("\\b(", str_flatten(rgb, "|"), ")\\b")
#> [1] "\\b(red|green|blue)\\b"
```

We could make this pattern more comprehensive if we had a good list of colors. One place we could start from is the list of built-in colors that R can use for plots:

```
str_view(colors())
#> [1] │ white
#> [2] │ aliceblue
#> [3] │ antiquewhite
#> [4] │ antiquewhite1
#> [5] │ antiquewhite2
#> [6] │ antiquewhite3
#> ... and 651 more
```

But let's first eliminate the numbered variants:

```
cols <- colors()
cols <- cols[!str_detect(cols, "\\d")]
str_view(cols)
#> [1] │ white
#> [2] │ aliceblue
#> [3] │ antiquewhite
#> [4] │ aquamarine
#> [5] │ azure
#> [6] │ beige
#> ... and 137 more
```

Then we can turn this into one giant pattern. We won't show the pattern here because it's huge, but you can see it working:

```
pattern <- str_c("\\b(", str_flatten(cols, "|"), ")\\b")
str_view(sentences, pattern)
#>   [2] │ Glue the sheet to the dark <blue> background.
#>  [12] │ A rod is used to catch <pink> <salmon>.
#>  [26] │ Two <blue> fish swam in the tank.
#>  [66] │ Cars and busses stalled in <snow> drifts.
#>  [92] │ A wisp of cloud hung in the <blue> air.
#> [112] │ Leaves turn <brown> and <yellow> in the fall.
#> ... and 57 more
```

In this example, `cols` contains only numbers and letters, so you don't need to worry about metacharacters. But in general, whenever you create patterns from existing strings, it's wise to run them through `str_escape()` to ensure they match literally.

Exercises

1. For each of the following challenges, try solving them by using both a single regular expression and a combination of multiple `str_detect()` calls:
   a. Find all `words` that start or end with `x`.
   b. Find all `words` that start with a vowel and end with a consonant.
   c. Are there any `words` that contain at least one of each different vowel?
2. Construct patterns to find evidence for and against the rule "i before e except after c."
3. `colors()` contains a number of modifiers like "lightgray" and "darkblue." How could you automatically identify these modifiers? (Think about how you might detect and then remove the colors that are modified.)
4. Create a regular expression that finds any base R dataset. You can get a list of these datasets via a special use of the `data()` function: `data(package = "datasets")$results[, "Item"]`. Note that a number of old datasets are individual vectors; these contain the name of the grouping "data frame" in parentheses, so you'll need to strip them off.

Regular Expressions in Other Places

Just like in the stringr and tidyr functions, there are many other places in R where you can use regular expressions. The following sections describe some other useful functions in the wider tidyverse and base R.

Tidyverse

There are three other particularly useful places where you might want to use regular expressions:

- `matches(pattern)` will select all variables whose name matches the supplied pattern. It's a "tidyselect" function that you can use anywhere in any tidyverse function that selects variables (e.g., `select()`, `rename_with()`, and `across()`).
- `pivot_longer()`'s `names_pattern` argument takes a vector of regular expressions, just like `separate_wider_regex()`. It's useful when extracting data from variable names with a complex structure.
- The `delim` argument in `separate_longer_delim()` and `separate_wider_delim()` usually matches a fixed string, but you can use `regex()` to make it match a pattern. This is useful, for example, if you want to match a comma that is optionally followed by a space, i.e., `regex(", ?")`.

Base R

`apropos(pattern)` searches all objects available from the global environment that match the given pattern. This is useful if you can't quite remember the name of a function:

```
apropos("replace")
#> [1] "%+replace%"       "replace"          "replace_na"
#> [4] "setReplaceMethod" "str_replace"      "str_replace_all"
#> [7] "str_replace_na"   "theme_replace"
```

`list.files(path, pattern)` lists all files in `path` that match a regular expression `pattern`. For example, you can find all the R Markdown files in the current directory with:

```
head(list.files(pattern = "\\.Rmd$"))
#> character(0)
```

It's worth noting that the pattern language used by base R is slightly different from that used by stringr. That's because stringr is built on top of the stringi package, which is in turn built on top of the ICU engine, whereas base R functions use either the TRE engine or the PCRE engine, depending on whether you've set `perl = TRUE`. Fortunately, the basics of regular expressions are so well established that you'll encounter few variations when working with the patterns you'll learn in this book.

You only need to be aware of the difference when you start to rely on advanced features like complex Unicode character ranges or special features that use the `(?…)` syntax.

Summary

With every punctuation character potentially overloaded with meaning, regular expressions are one of the most compact languages out there. They're definitely confusing at first, but as you train your eyes to read them and your brain to understand them, you unlock a powerful skill that you can use in R and in many other places.

In this chapter, you've started your journey to become a regular expression master by learning the most useful stringr functions and the most important components of the regular expression language. And there are plenty of resources to learn more.

A good place to start is `vignette("regular-expressions", package = "stringr")`: it documents the full set of syntax supported by stringr. Another useful reference is *https://oreil.ly/MVwoC*. It's not R specific, but you can use it to learn about the most advanced features of regexes and how they work under the hood.

It's also good to know that stringr is implemented on top of the stringi package by Marek Gagolewski. If you're struggling to find a function that does what you need in stringr, don't be afraid to look in stringi. You'll find stringi easy to pick up because it follows many of the same conventions as stringr.

In the next chapter, we'll talk about a data structure closely related to strings: factors. Factors are used to represent categorical data in R, i.e., data with a fixed and known set of possible values identified by a vector of strings.

CHAPTER 16

Factors

Introduction

Factors are used for categorical variables, variables that have a fixed and known set of possible values. They are also useful when you want to display character vectors in a nonalphabetical order.

We'll start by motivating why factors are needed for data analysis[1] and how you can create them with `factor()`. We'll then introduce you to the `gss_cat` dataset, which contains a bunch of categorical variables to experiment with. You'll then use that dataset to practice modifying the order and values of factors, before we finish up with a discussion of ordered factors.

Prerequisites

Base R provides some basic tools for creating and manipulating factors. We'll supplement these with the forcats package, which is part of the core tidyverse. It provides tools for dealing with *cat*egorical variables (and it's an anagram of factors!) using a wide range of helpers for working with factors.

```
library(tidyverse)
```

Factor Basics

Imagine that you have a variable that records the month:

```
x1 <- c("Dec", "Apr", "Jan", "Mar")
```

Using a string to record this variable has two problems:

[1] They're also really important for modeling.

1. There are only 12 possible months, and there's nothing saving you from typos:

   ```
   x2 <- c("Dec", "Apr", "Jam", "Mar")
   ```

2. It doesn't sort in a useful way:

   ```
   sort(x1)
   #> [1] "Apr" "Dec" "Jan" "Mar"
   ```

You can fix both of these problems with a factor. To create a factor, you must start by creating a list of the valid *levels*:

```
month_levels <- c(
  "Jan", "Feb", "Mar", "Apr", "May", "Jun",
  "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
)
```

Now you can create a factor:

```
y1 <- factor(x1, levels = month_levels)
y1
#> [1] Dec Apr Jan Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

sort(y1)
#> [1] Jan Mar Apr Dec
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

Any values not in the level will be silently converted to `NA`:

```
y2 <- factor(x2, levels = month_levels)
y2
#> [1] Dec  Apr  <NA> Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

This seems risky, so you might want to use `forcats::fct()` instead:

```
y2 <- fct(x2, levels = month_levels)
#> Error in `fct()`:
#> ! All values of `x` must appear in `levels` or `na`
#> ℹ Missing level: "Jam"
```

If you omit the levels, they'll be taken from the data in alphabetical order:

```
factor(x1)
#> [1] Dec Apr Jan Mar
#> Levels: Apr Dec Jan Mar
```

Sorting alphabetically is slightly risky because not every computer will sort strings in the same way. So `forcats::fct()` orders by first appearance:

```
fct(x1)
#> [1] Dec Apr Jan Mar
#> Levels: Dec Apr Jan Mar
```

If you ever need to access the set of valid levels directly, you can do so with `levels()`:

```
levels(y2)
#>  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

You can also create a factor when reading your data with readr with col_factor():

```
csv <- "
month,value
Jan,12
Feb,56
Mar,12"

df <- read_csv(csv, col_types = cols(month = col_factor(month_levels)))
df$month
#> [1] Jan Feb Mar
#> Levels: Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
```

General Social Survey

For the rest of this chapter, we're going to use forcats::gss_cat. It's a sample of data from the General Social Survey, a long-running US survey conducted by the independent research organization NORC at the University of Chicago. The survey has thousands of questions, so in gss_cat Hadley selected a handful that will illustrate some common challenges you'll encounter when working with factors.

```
gss_cat
#> # A tibble: 21,483 × 9
#>    year marital         age race  rincome        partyid
#>   <int> <fct>         <int> <fct> <fct>          <fct>
#> 1  2000 Never married    26 White $8000 to 9999  Ind,near rep
#> 2  2000 Divorced         48 White $8000 to 9999  Not str republican
#> 3  2000 Widowed          67 White Not applicable Independent
#> 4  2000 Never married    39 White Not applicable Ind,near rep
#> 5  2000 Divorced         25 White Not applicable Not str democrat
#> 6  2000 Married          25 White $20000 - 24999 Strong democrat
#> # … with 21,477 more rows, and 3 more variables: relig <fct>, denom <fct>,
#> #   tvhours <int>
```

(Remember, since this dataset is provided by a package, you can get more information about the variables with ?gss_cat.)

When factors are stored in a tibble, you can't see their levels so easily. One way to view them is with count():

```
gss_cat |>
  count(race)
#> # A tibble: 3 × 2
#>   race      n
#>   <fct> <int>
#> 1 Other  1959
#> 2 Black  3129
#> 3 White 16395
```

When working with factors, the two most common operations are changing the order of the levels and changing the values of the levels. Those operations are described in the following sections.

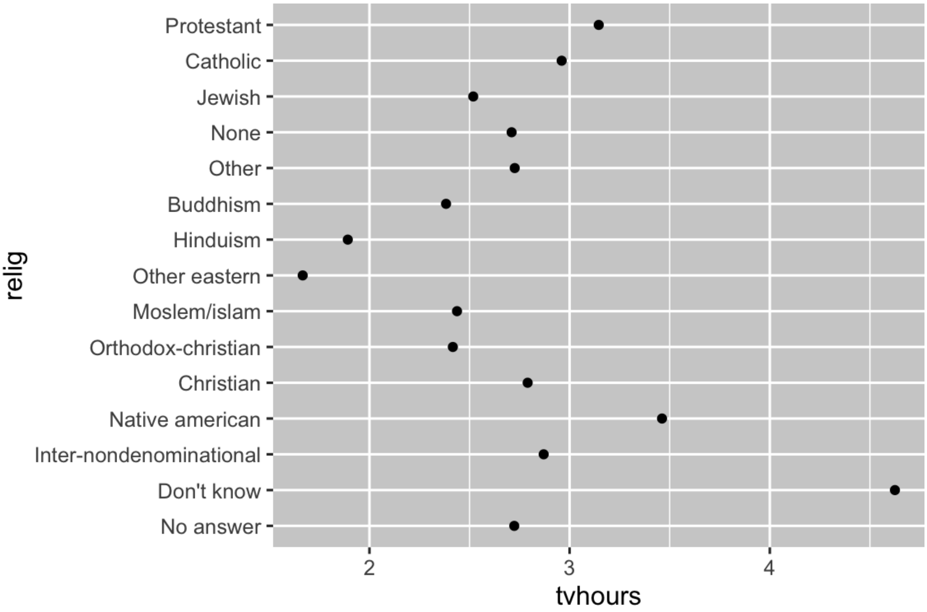
Exercise

1. Explore the distribution of `rincome` (reported income). What makes the default bar chart hard to understand? How could you improve the plot?
2. What is the most common `relig` in this survey? What's the most common `partyid`?
3. Which `relig` does `denom` (denomination) apply to? How can you find out with a table? How can you find out with a visualization?

Modifying Factor Order

It's often useful to change the order of the factor levels in a visualization. For example, imagine you want to explore the average number of hours spent watching TV per day across religions:

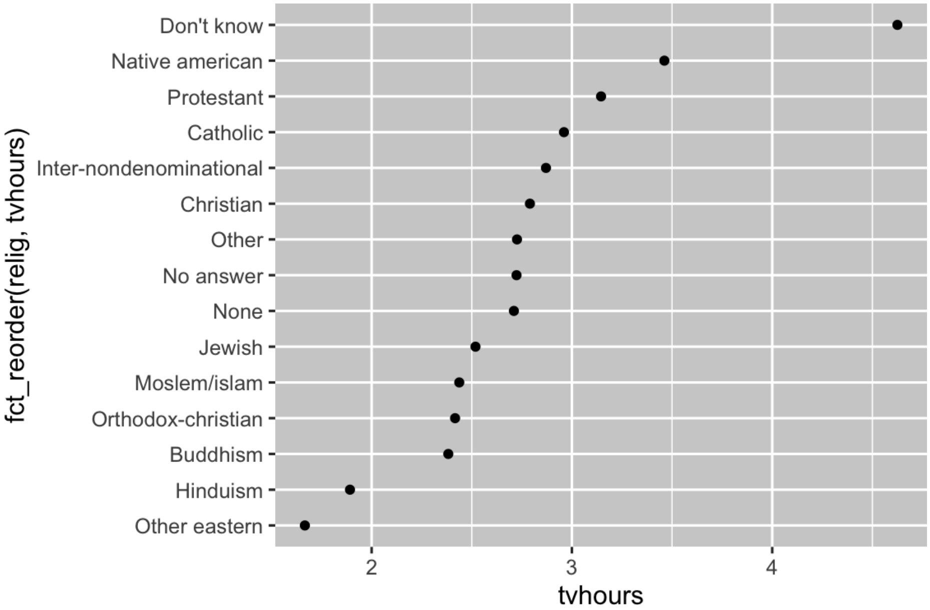
```
relig_summary <- gss_cat |>
  group_by(relig) |>
  summarize(
    tvhours = mean(tvhours, na.rm = TRUE),
    n = n()
  )

ggplot(relig_summary, aes(x = tvhours, y = relig)) +
  geom_point()
```



It is hard to read this plot because there's no overall pattern. We can improve it by reordering the levels of `relig` using `fct_reorder()`. `fct_reorder()` takes three arguments:

- `f`, the factor whose levels you want to modify.
- `x`, a numeric vector that you want to use to reorder the levels.
- Optionally, `fun`, a function that's used if there are multiple values of `x` for each value of `f`. The default value is `median`.

```
ggplot(relig_summary, aes(x = tvhours, y = fct_reorder(relig, tvhours))) +
  geom_point()
```



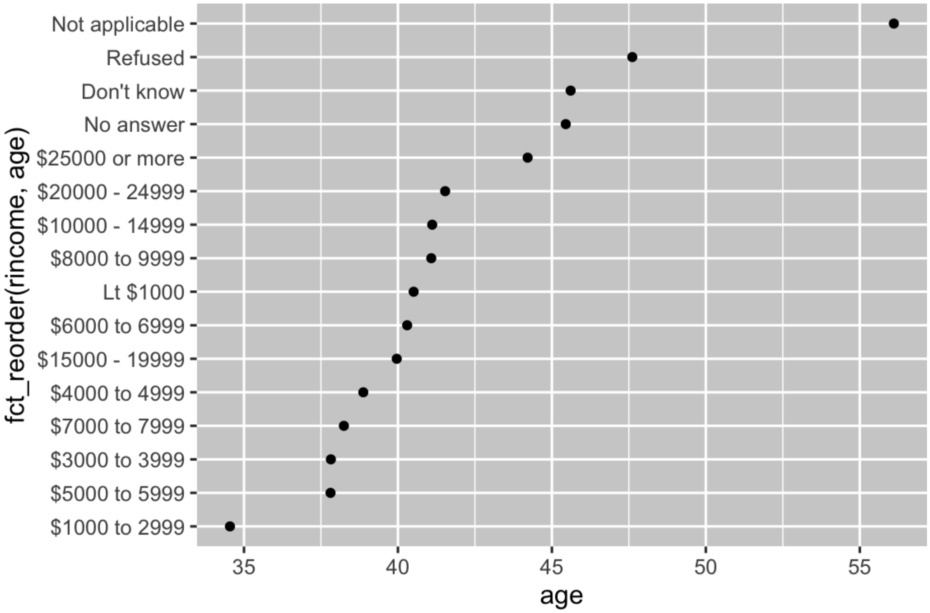
Reordering religion makes it much easier to see that people in the "Don't know" category watch much more TV, and Hinduism and other Eastern religions watch much less.

As you start making more complicated transformations, we recommend moving them out of `aes()` and into a separate `mutate()` step. For example, you could rewrite the previous plot as:

```
relig_summary |>
  mutate(
    relig = fct_reorder(relig, tvhours)
  ) |>
  ggplot(aes(x = tvhours, y = relig)) +
  geom_point()
```

What if we create a similar plot looking at how average age varies across reported income level?

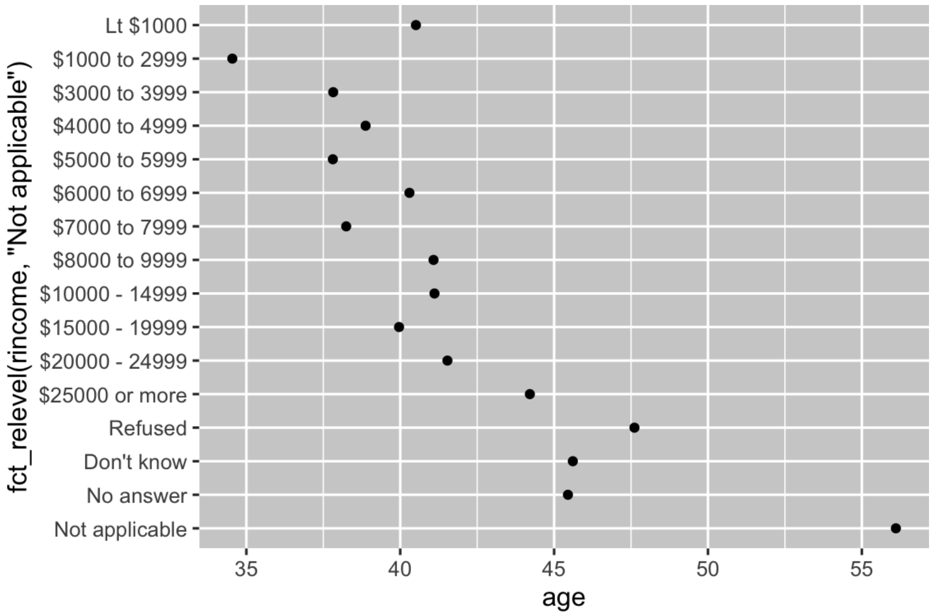
```
rincome_summary <- gss_cat |>
  group_by(rincome) |>
  summarize(
    age = mean(age, na.rm = TRUE),
    n = n()
  )

ggplot(rincome_summary, aes(x = age, y = fct_reorder(rincome, age))) +
  geom_point()
```



Here, arbitrarily reordering the levels isn't a good idea! That's because `rincome` already has a principled order that we shouldn't mess with. Reserve `fct_reorder()` for factors whose levels are arbitrarily ordered.

However, it does make sense to pull "Not applicable" to the front with the other special levels. You can use `fct_relevel()`. It takes a factor, `f`, and then any number of levels that you want to move to the front of the line.

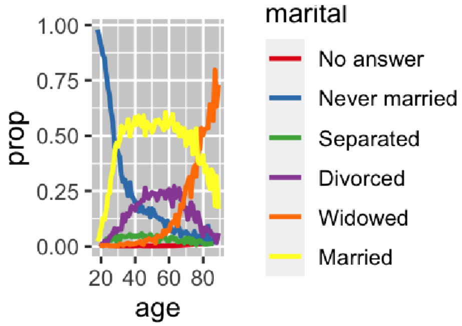
```
ggplot(rincome_summary, aes(x = age, y = fct_relevel(rincome, "Not applicable"))) +
  geom_point()
```

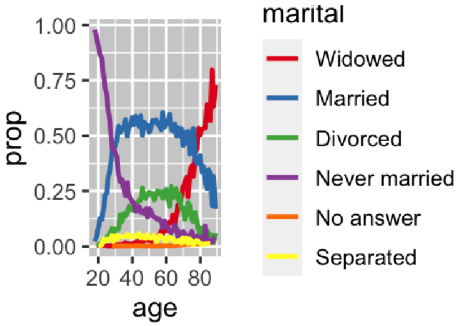


Why do you think the average age for “Not applicable” is so high?

Another type of reordering is useful when you are coloring the lines on a plot. `fct_reorder2(f, x, y)` reorders the factor `f` by the `y` values associated with the largest `x` values. This makes the plot easier to read because the colors of the line at the far right of the plot will line up with the legend.

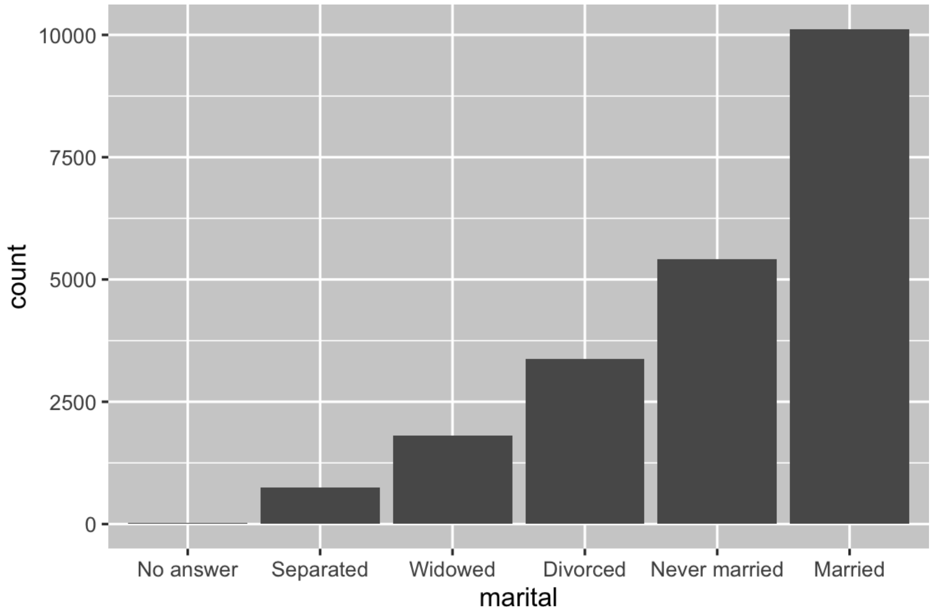
```
by_age <- gss_cat |>
  filter(!is.na(age)) |>
  count(age, marital) |>
  group_by(age) |>
  mutate(
    prop = n / sum(n)
  )

ggplot(by_age, aes(x = age, y = prop, color = marital)) +
  geom_line(linewidth = 1) +
  scale_color_brewer(palette = "Set1")

ggplot(by_age, aes(x = age, y = prop, color = fct_reorder2(marital, age, prop))) +
  geom_line(linewidth = 1) +
  scale_color_brewer(palette = "Set1") +
  labs(color = "marital")
```





Finally, for bar plots, you can use `fct_infreq()` to order levels in decreasing frequency: this is the simplest type of reordering because it doesn't need any extra variables. Combine it with `fct_rev()` if you want them in increasing frequency so that in the bar plot the largest values are on the right, not the left.

```
gss_cat |>
  mutate(marital = marital |> fct_infreq() |> fct_rev()) |>
  ggplot(aes(x = marital)) +
  geom_bar()
```



Exercises

1. There are some suspiciously high numbers in tvhours. Is the mean a good summary?
2. For each factor in gss_cat identify whether the order of the levels is arbitrary or principled.
3. Why did moving "Not applicable" to the front of the levels move it to the bottom of the plot?

Modifying Factor Levels

More powerful than changing the orders of the levels is changing their values. This allows you to clarify labels for publication and collapse levels for high-level displays. The most general and powerful tool is fct_recode(). It allows you to recode, or change, the value of each level. For example, take the partyid variable from the gss_cat data frame:

```
gss_cat |> count(partyid)
#> # A tibble: 10 × 2
#>   partyid                n
#>   <fct>              <int>
#> 1 No answer            154
#> 2 Don't know             1
#> 3 Other party          393
#> 4 Strong republican   2314
#> 5 Not str republican  3032
#> 6 Ind,near rep        1791
#> # … with 4 more rows
```

The levels are terse and inconsistent. Let's tweak them to be longer and use a parallel construction. Like most rename and recoding functions in the tidyverse, the new values go on the left, and the old values go on the right:

```
gss_cat |>
  mutate(
    partyid = fct_recode(partyid,
      "Republican, strong"    = "Strong republican",
      "Republican, weak"      = "Not str republican",
      "Independent, near rep" = "Ind,near rep",
      "Independent, near dem" = "Ind,near dem",
      "Democrat, weak"        = "Not str democrat",
      "Democrat, strong"      = "Strong democrat"
    )
  ) |>
  count(partyid)
#> # A tibble: 10 × 2
#>   partyid                   n
#>   <fct>                 <int>
#> 1 No answer               154
#> 2 Don't know                1
#> 3 Other party             393
#> 4 Republican, strong     2314
#> 5 Republican, weak       3032
#> 6 Independent, near rep  1791
#> # … with 4 more rows
```

fct_recode() will leave the levels that aren't explicitly mentioned as is and will warn you if you accidentally refer to a level that doesn't exist.

To combine groups, you can assign multiple old levels to the same new level:

```
gss_cat |>
  mutate(
    partyid = fct_recode(partyid,
      "Republican, strong"    = "Strong republican",
      "Republican, weak"      = "Not str republican",
      "Independent, near rep" = "Ind,near rep",
      "Independent, near dem" = "Ind,near dem",
      "Democrat, weak"        = "Not str democrat",
      "Democrat, strong"      = "Strong democrat",
      "Other"                 = "No answer",
      "Other"                 = "Don't know",
      "Other"                 = "Other party"
    )
  )
```

Use this technique with care: if you group levels that are truly different, you will end up with misleading results.

If you want to collapse a lot of levels, fct_collapse() is a useful variant of fct_recode(). For each new variable, you can provide a vector of old levels:

```
gss_cat |>
  mutate(
    partyid = fct_collapse(partyid,
      "other" = c("No answer", "Don't know", "Other party"),
      "rep" = c("Strong republican", "Not str republican"),
      "ind" = c("Ind,near rep", "Independent", "Ind,near dem"),
      "dem" = c("Not str democrat", "Strong democrat")
    )
  ) |>
  count(partyid)
#> # A tibble: 4 × 2
#>   partyid     n
#>   <fct>   <int>
#> 1 other     548
#> 2 rep      5346
#> 3 ind      8409
#> 4 dem      7180
```

Sometimes you just want to lump together the small groups to make a plot or table simpler. That's the job of the fct_lump_*() family of functions. fct_lump_lowfreq() is a simple starting point that progressively lumps the smallest group's categories into "Other," always keeping "Other" as the smallest category.

```
gss_cat |>
  mutate(relig = fct_lump_lowfreq(relig)) |>
  count(relig)
#> # A tibble: 2 × 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
#> 2 Other      10637
```

In this case it's not very helpful: it is true that the majority of Americans in this survey are Protestant, but we'd probably like to see some more details! Instead, we can use `fct_lump_n()` to specify that we want exactly 10 groups:

```
gss_cat |>
  mutate(relig = fct_lump_n(relig, n = 10)) |>
  count(relig, sort = TRUE)
#> # A tibble: 10 × 2
#>   relig          n
#>   <fct>      <int>
#> 1 Protestant 10846
#> 2 Catholic    5124
#> 3 None        3523
#> 4 Christian    689
#> 5 Other        458
#> 6 Jewish       388
#> # … with 4 more rows
```

Read the documentation to learn about `fct_lump_min()` and `fct_lump_prop()`, which are useful in other cases.

Exercises

1. How have the proportions of people identifying as Democrat, Republican, and Independent changed over time?
2. How could you collapse `rincome` into a small set of categories?
3. Notice there are 9 groups (excluding other) in the previous `fct_lump` example. Why not 10? (Hint: Type `?fct_lump`, and find the default for the argument `other_level` is "Other.")

Ordered Factors

Before we go on, there's a special type of factor that needs to be mentioned briefly: ordered factors. Ordered factors, created with `ordered()`, imply a strict ordering and equal distance between levels: the first level is "less than" the second level by the same amount that the second level is "less than" the third level, and so on. You can recognize them when printing because they use < between the factor levels:

```
ordered(c("a", "b", "c"))
#> [1] a b c
#> Levels: a < b < c
```

In practice, `ordered()` factors behave similarly to regular factors. There are only two places where you might notice different behavior:

- If you map an ordered factor to color or fill in ggplot2, it will default to `scale_color_viridis()`/`scale_fill_viridis()`, a color scale that implies a ranking.
- If you use an ordered function in a linear model, it will use "polygonal contrasts." These are mildly useful, but you are unlikely to have heard of them unless you have a PhD in statistics, and even then you probably don't routinely interpret them. If you want to learn more, we recommend `vignette("contrasts", package = "faux")` by Lisa DeBruine.

Given the arguable utility of these differences, we don't generally recommend using ordered factors.

Summary

This chapter introduced you to the handy forcats package for working with factors, explaining the most commonly used functions. forcats contains a wide range of other helpers that we didn't have space to discuss here, so whenever you're facing a factor analysis challenge that you haven't encountered before, I highly recommend skimming the reference index to see if there's a canned function that can help solve your problem.

If you want to learn more about factors after reading this chapter, we recommend reading Amelia McNamara and Nicholas Horton's paper, "Wrangling categorical data in R". This paper lays out some of the history discussed in "stringsAsFactors: An unauthorized biography" and "stringsAsFactors = <sigh>", and compares the tidy approaches to categorical data outlined in this book with base R methods. An early version of the paper helped motivate and scope the forcats package; thanks, Amelia and Nick!

In the next chapter we'll switch gears to start learning about dates and times in R. Dates and times seem deceptively simple, but as you'll soon see, the more you learn about them, the more complex they seem to get!

CHAPTER 17
Dates and Times

Introduction

This chapter will show you how to work with dates and times in R. At first glance, dates and times seem simple. You use them all the time in your regular life, and they don't seem to cause much confusion. However, the more you learn about dates and times, the more complicated they seem to get!

To warm up, think about how many days there are in a year and how many hours there are in a day. You probably remembered that most years have 365 days, but leap years have 366. Do you know the full rule for determining if a year is a leap year?[1] The number of hours in a day is a little less obvious: most days have 24 hours, but in places that use daylight saving time (DST), one day each year has 23 hours and another has 25.

Dates and times are hard because they have to reconcile two physical phenomena (the rotation of Earth and its orbit around the sun) with a whole raft of geopolitical phenomena including months, time zones, and DST. This chapter won't teach you every last detail about dates and times, but it will give you a solid grounding of practical skills that will help you with common data analysis challenges.

We'll begin by showing you how to create date-times from various inputs, and then once you've got a date-time, you'll learn how you can extract components such as year, month, and day. We'll then dive into the tricky topic of working with time spans, which come in a variety of flavors depending on what you're trying to do. We'll conclude with a brief discussion of the additional challenges posed by time zones.

[1] A year is a leap year if it's divisible by 4, unless it's also divisible by 100, except if it's also divisible by 400. In other words, in every set of 400 years, there's 97 leap years.

Prerequisites

This chapter will focus on the lubridate package, which makes it easier to work with dates and times in R. As of the latest tidyverse release, lubridate is part of core tidyverse. We will also need nycflights13 for practice data.

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

Creating Date/Times

There are three types of date/time data that refer to an instant in time:

- A *date*. Tibbles print this as `<date>`.
- A *time* within a day. Tibbles print this as `<time>`.
- A *date-time* is a date plus a time: it uniquely identifies an instant in time (typically to the nearest second). Tibbles print this as `<dttm>`. Base R calls these POSIXct, but that doesn't exactly trip off the tongue.

In this chapter we are going to focus on dates and date-times as R doesn't have a native class for storing times. If you need one, you can use the hms package.

You should always use the simplest possible data type that works for your needs. That means if you can use a date instead of a date-time, you should. Date-times are substantially more complicated because of the need to handle time zones, which we'll come back to at the end of the chapter.

To get the current date or date-time, you can use `today()` or `now()`:

```
today()
#> [1] "2023-03-12"
now()
#> [1] "2023-03-12 13:07:31 CDT"
```

Otherwise, the following sections describe the four ways you're likely to create a date/time:

- While reading a file with readr
- From a string
- From individual date-time components
- From an existing date/time object

During Import

If your CSV contains an ISO8601 date or date-time, you don't need to do anything; readr will automatically recognize it:

```
csv <- "
  date,datetime
  2022-01-02,2022-01-02 05:12
"
read_csv(csv)
#> # A tibble: 1 × 2
#>   date       datetime
#>   <date>     <dttm>
#> 1 2022-01-02 2022-01-02 05:12:00
```

If you haven't heard of *ISO8601* before, it's an international standard for writing dates where the components of a date are organized from biggest to smallest separated by `-`. For example, in ISO8601 May 3, 2022, is `2022-05-03`. ISO8601 dates can also include times, where hour, minute, and second are separated by `:`, and the date and time components are separated by either a `T` or a space. For example, you could write 4:26 p.m. on May 3, 2022, as either `2022-05-03 16:26` or `2022-05-03T16:26`.

For other date-time formats, you'll need to use `col_types` plus `col_date()` or `col_datetime()` along with a date-time format. The date-time format used by readr is a standard used across many programming languages, describing a date component with a `%` followed by a single character. For example, `%Y-%m-%d` specifies a date that's a year, `-`, month (as number) `-`, day. Table 17-1 lists all the options.

Table 17-1. All date formats understood by readr

Type	Code	Meaning	Example
Year	`%Y`	4-digit year	2021
	`%y`	2-digit year	21
Month	`%m`	Number	2
	`%b`	Abbreviated name	Feb
	`%B`	Full name	February
Day	`%d`	Two digits	02
	`%e`	One or two digits	2
Time	`%H`	24-hour hour	13
	`%I`	12-hour hour	1
	`%p`	a.m./p.m.	pm
	`%M`	Minutes	35
	`%S`	Seconds	45
	`%OS`	Seconds with decimal component	45.35
	`%Z`	Time zone name	America/Chicago
	`%z`	Offset from UTC	+0800

Type	Code	Meaning	Example
Other	%.	Skip one nondigit	:
	%*	Skip any number of nondigits	

This code shows a few options applied to a very ambiguous date:

```
csv <- "
  date
  01/02/15
"

read_csv(csv, col_types = cols(date = col_date("%m/%d/%y")))
#> # A tibble: 1 × 1
#>   date
#>   <date>
#> 1 2015-01-02

read_csv(csv, col_types = cols(date = col_date("%d/%m/%y")))
#> # A tibble: 1 × 1
#>   date
#>   <date>
#> 1 2015-02-01

read_csv(csv, col_types = cols(date = col_date("%y/%m/%d")))
#> # A tibble: 1 × 1
#>   date
#>   <date>
#> 1 2001-02-15
```

Note that no matter how you specify the date format, it's always displayed the same way once you get it into R.

If you're using `%b` or `%B` and working with non-English dates, you'll also need to provide a `locale()`. See the list of built-in languages in `date_names_langs()`, or create your own with `date_names()`.

From Strings

The date-time specification language is powerful but requires careful analysis of the date format. An alternative approach is to use lubridate's helpers, which attempt to automatically determine the format once you specify the order of the component. To use them, identify the order in which year, month, and day appear in your dates; then arrange "y," "m," and "d" in the same order. That gives you the name of the lubridate function that will parse your date. For example:

```
ymd("2017-01-31")
#> [1] "2017-01-31"
mdy("January 31st, 2017")
#> [1] "2017-01-31"
dmy("31-Jan-2017")
#> [1] "2017-01-31"
```

ymd() and friends create dates. To create a date-time, add an underscore and one or more of "h", "m", and "s" to the name of the parsing function:

```
ymd_hms("2017-01-31 20:11:59")
#> [1] "2017-01-31 20:11:59 UTC"
mdy_hm("01/31/2017 08:01")
#> [1] "2017-01-31 08:01:00 UTC"
```

You can also force the creation of a date-time from a date by supplying a time zone:

```
ymd("2017-01-31", tz = "UTC")
#> [1] "2017-01-31 UTC"
```

Here I use the UTC[2] timezone, which you might also know as GMT, or Greenwich Mean Time, the time at 0° longitude.[3] It doesn't use daylight saving time, making it a bit easier to compute with.

From Individual Components

Instead of a single string, sometimes you'll have the individual components of the date-time spread across multiple columns. This is what we have in the `flights` data:

```
flights |>
  select(year, month, day, hour, minute)
#> # A tibble: 336,776 × 5
#>    year month   day  hour minute
#>   <int> <int> <int> <dbl>  <dbl>
#> 1  2013     1     1     5     15
#> 2  2013     1     1     5     29
#> 3  2013     1     1     5     40
#> 4  2013     1     1     5     45
#> 5  2013     1     1     6      0
#> 6  2013     1     1     5     58
#> # … with 336,770 more rows
```

To create a date/time from this sort of input, use make_date() for dates, or use make_datetime() for date-times:

```
flights |>
  select(year, month, day, hour, minute) |>
  mutate(departure = make_datetime(year, month, day, hour, minute))
#> # A tibble: 336,776 × 6
#>    year month   day  hour minute departure
#>   <int> <int> <int> <dbl>  <dbl> <dttm>
#> 1  2013     1     1     5     15 2013-01-01 05:15:00
#> 2  2013     1     1     5     29 2013-01-01 05:29:00
#> 3  2013     1     1     5     40 2013-01-01 05:40:00
#> 4  2013     1     1     5     45 2013-01-01 05:45:00
#> 5  2013     1     1     6      0 2013-01-01 06:00:00
```

[2] You might wonder what UTC stands for. It's a compromise between the English "Coordinated Universal Time" and French "Temps Universel Coordonné."

[3] No prizes for guessing which country came up with the longitude system.

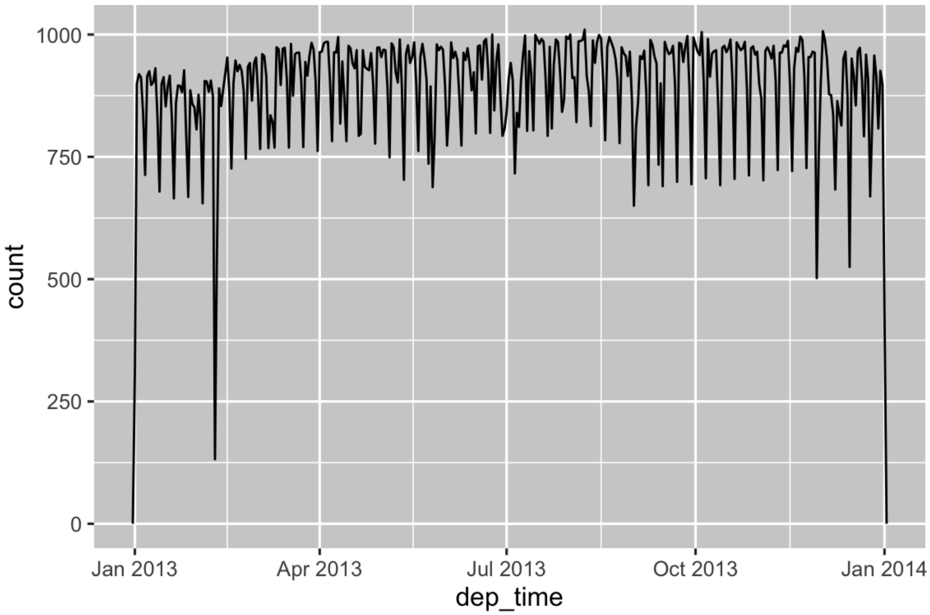
```
#> 6  2013     1     1     5     58 2013-01-01 05:58:00
#> # … with 336,770 more rows
```

Let's do the same thing for each of the four time columns in `flights`. The times are represented in a slightly odd format, so we use modulus arithmetic to pull out the hour and minute components. Once we've created the date-time variables, we focus in on the variables we'll explore in the rest of the chapter.

```
make_datetime_100 <- function(year, month, day, time) {
  make_datetime(year, month, day, time %/% 100, time %% 100)
}

flights_dt <- flights |>
  filter(!is.na(dep_time), !is.na(arr_time)) |>
  mutate(
    dep_time = make_datetime_100(year, month, day, dep_time),
    arr_time = make_datetime_100(year, month, day, arr_time),
    sched_dep_time = make_datetime_100(year, month, day, sched_dep_time),
    sched_arr_time = make_datetime_100(year, month, day, sched_arr_time)
  ) |>
  select(origin, dest, ends_with("delay"), ends_with("time"))

flights_dt
#> # A tibble: 328,063 × 9
#>   origin dest  dep_delay arr_delay dep_time            sched_dep_time     
#>   <chr>  <chr>     <dbl>     <dbl> <dttm>              <dttm>             
#> 1 EWR    IAH           2        11 2013-01-01 05:17:00 2013-01-01 05:15:00
#> 2 LGA    IAH           4        20 2013-01-01 05:33:00 2013-01-01 05:29:00
#> 3 JFK    MIA           2        33 2013-01-01 05:42:00 2013-01-01 05:40:00
#> 4 JFK    BQN          -1       -18 2013-01-01 05:44:00 2013-01-01 05:45:00
#> 5 LGA    ATL          -6       -25 2013-01-01 05:54:00 2013-01-01 06:00:00
#> 6 EWR    ORD          -4        12 2013-01-01 05:54:00 2013-01-01 05:58:00
#> # … with 328,057 more rows, and 3 more variables: arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

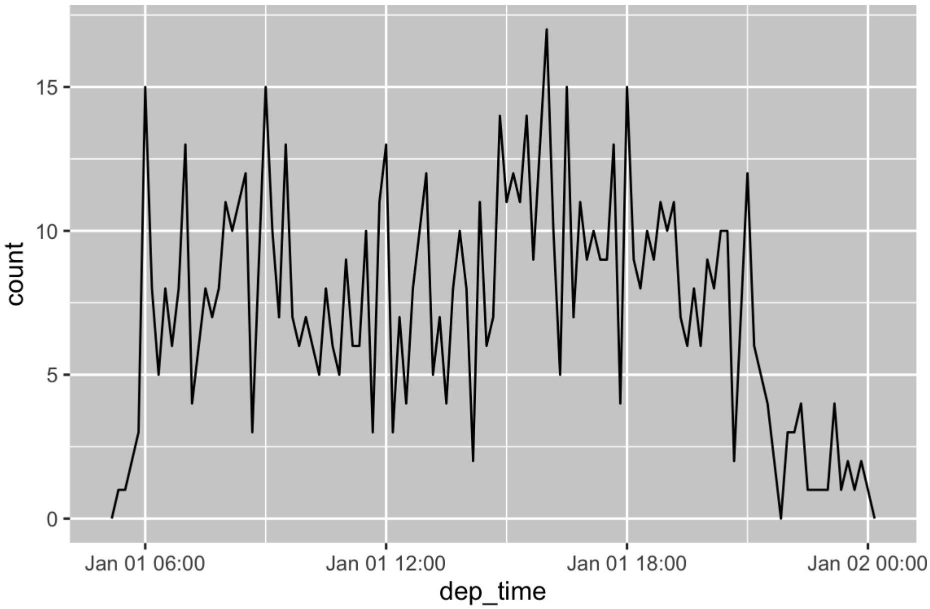
With this data, we can visualize the distribution of departure times across the year:

```
flights_dt |>
  ggplot(aes(x = dep_time)) +
  geom_freqpoly(binwidth = 86400) # 86400 seconds = 1 day
```



Or within a single day:

```
flights_dt |>
  filter(dep_time < ymd(20130102)) |>
  ggplot(aes(x = dep_time)) +
  geom_freqpoly(binwidth = 600) # 600 s = 10 minutes
```



Note that when you use date-times in a numeric context (like in a histogram), 1 means 1 second, so a binwidth of 86400 means one day. For dates, 1 means 1 day.

From Other Types

You may want to switch between a date-time and a date. That's the job of `as_datetime()` and `as_date()`:

```
as_datetime(today())
#> [1] "2023-03-12 UTC"
as_date(now())
#> [1] "2023-03-12"
```

Sometimes you'll get date/times as numeric offsets from the "Unix epoch," 1970-01-01. If the offset is in seconds, use `as_datetime()`; if it's in days, use `as_date()`.

```
as_datetime(60 * 60 * 10)
#> [1] "1970-01-01 10:00:00 UTC"
as_date(365 * 10 + 2)
#> [1] "1980-01-01"
```

Exercises

1. What happens if you parse a string that contains invalid dates?

   ```
   ymd(c("2010-10-10", "bananas"))
   ```

2. What does the `tzone` argument to `today()` do? Why is it important?

3. For each of the following date-times, show how you'd parse it using a readr column specification and a lubridate function.

   ```
   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"
   ```

Date-Time Components

Now that you know how to get date-time data into R's date-time data structures, let's explore what you can do with them. This section will focus on the accessor functions that let you get and set individual components. The next section will look at how arithmetic works with date-times.

Getting Components

You can pull out individual parts of the date with the accessor functions `year()`, `month()`, `mday()` (day of the month), `yday()` (day of the year), `wday()` (day of the week), `hour()`, `minute()`, and `second()`. These are effectively the opposites of `make_datetime()`.

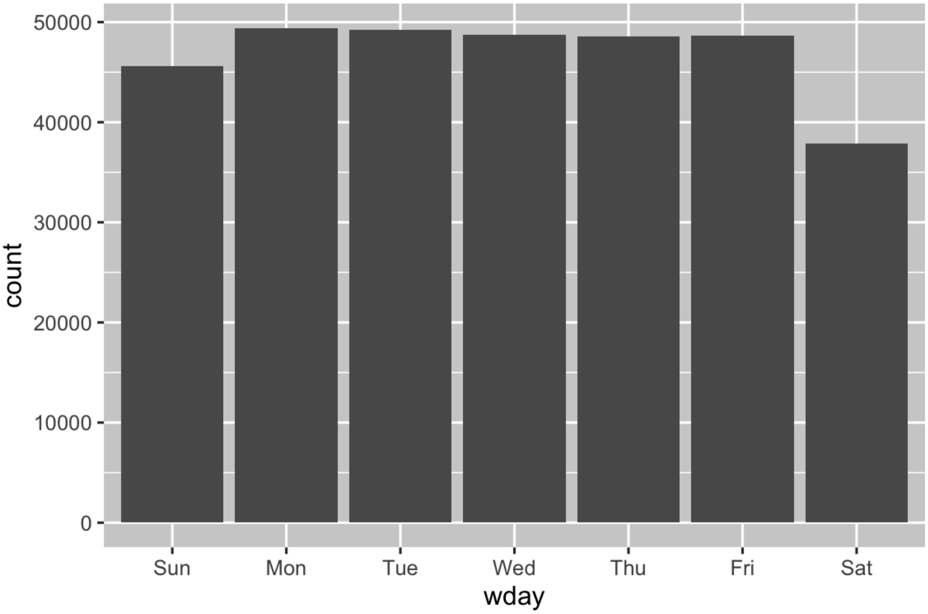
```
datetime <- ymd_hms("2026-07-08 12:34:56")

year(datetime)
#> [1] 2026
month(datetime)
#> [1] 7
mday(datetime)
#> [1] 8

yday(datetime)
#> [1] 189
wday(datetime)
#> [1] 4
```

For `month()` and `wday()` you can set `label = TRUE` to return the abbreviated name of the month or day of the week. Set `abbr = FALSE` to return the full name.

```
month(datetime, label = TRUE)
#> [1] Jul
#> 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
wday(datetime, label = TRUE, abbr = FALSE)
#> [1] Wednesday
#> 7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday
```

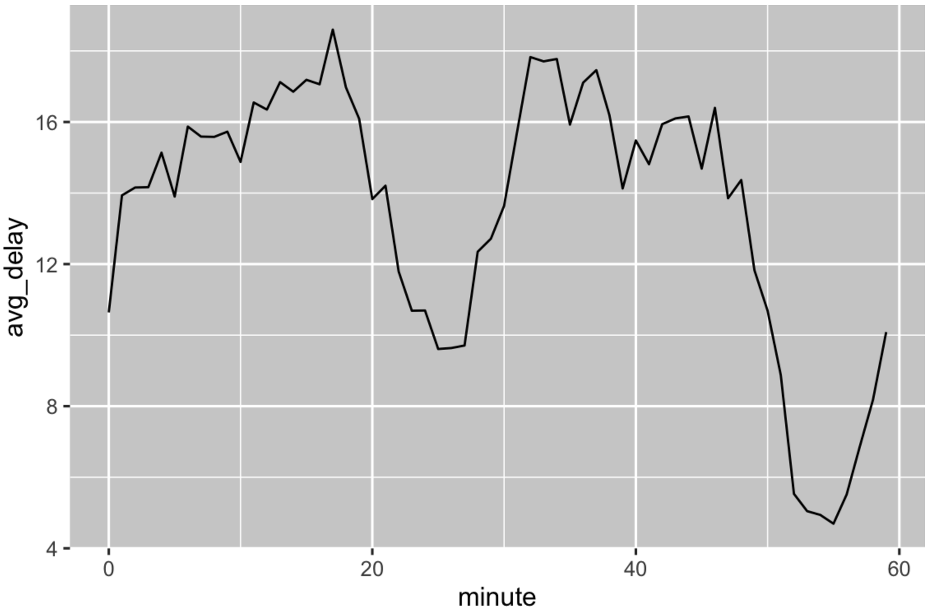
We can use wday() to see that more flights depart during the week than on the weekend:

```
flights_dt |>
  mutate(wday = wday(dep_time, label = TRUE)) |>
  ggplot(aes(x = wday)) +
  geom_bar()
```



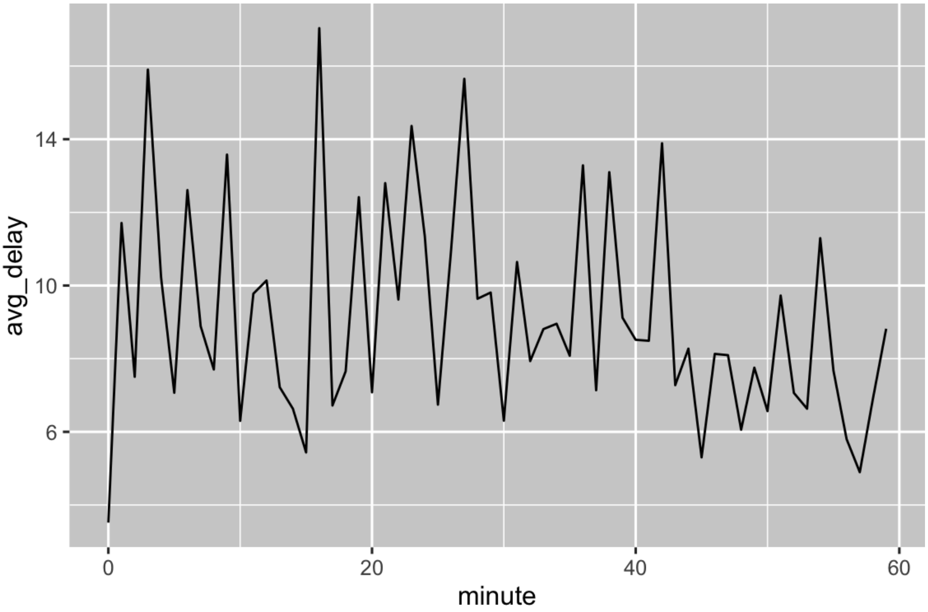
We can also look at the average departure delay by minute within the hour. There's an interesting pattern: flights leaving in minutes 20–30 and 50–60 have much lower delays than the rest of the hour!

```
flights_dt |>
  mutate(minute = minute(dep_time)) |>
  group_by(minute) |>
  summarize(
    avg_delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  ) |>
  ggplot(aes(x = minute, y = avg_delay)) +
  geom_line()
```



Interestingly, if we look at the *scheduled* departure time, we don't see such a strong pattern:

```
sched_dep <- flights_dt |>
  mutate(minute = minute(sched_dep_time)) |>
  group_by(minute) |>
  summarize(
    avg_delay = mean(arr_delay, na.rm = TRUE),
    n = n()
  )

ggplot(sched_dep, aes(x = minute, y = avg_delay)) +
  geom_line()
```



So why do we see that pattern with the actual departure times? Well, like much data collected by humans, there's a strong bias toward flights leaving at "nice" departure times, as Figure 17-1 shows. Always be alert for this sort of pattern whenever you work with data that involves human judgment!

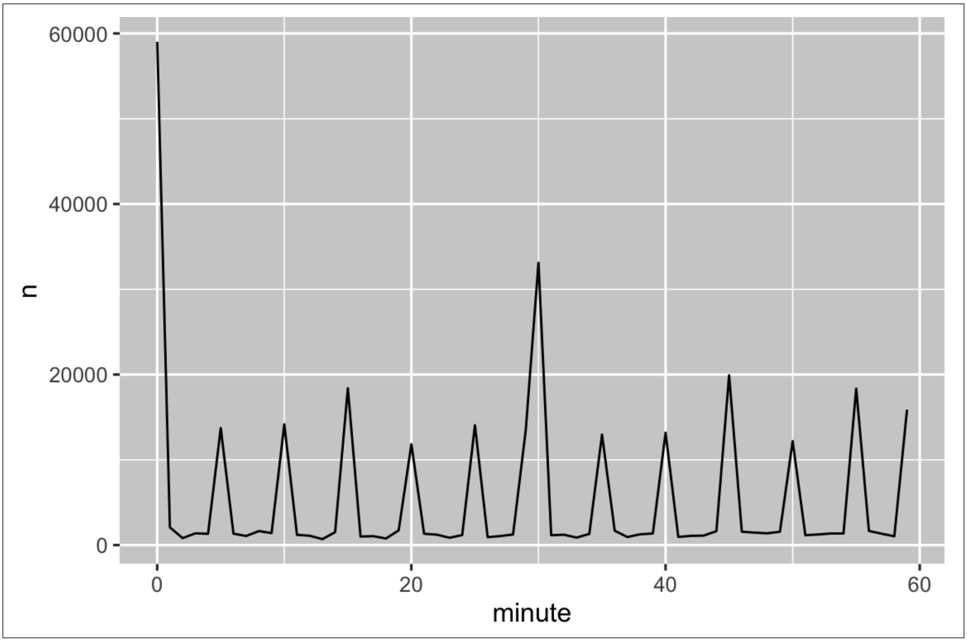
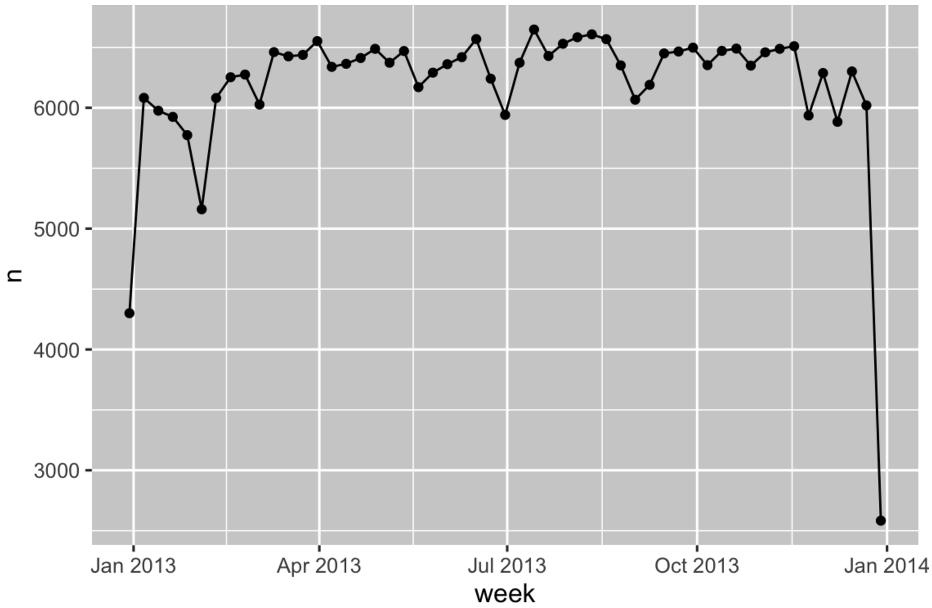


Figure 17-1. A frequency polygon showing the number of flights scheduled to depart each hour. You can see a strong preference for round numbers like 0 and 30 and generally for numbers that are a multiple of five.

Rounding

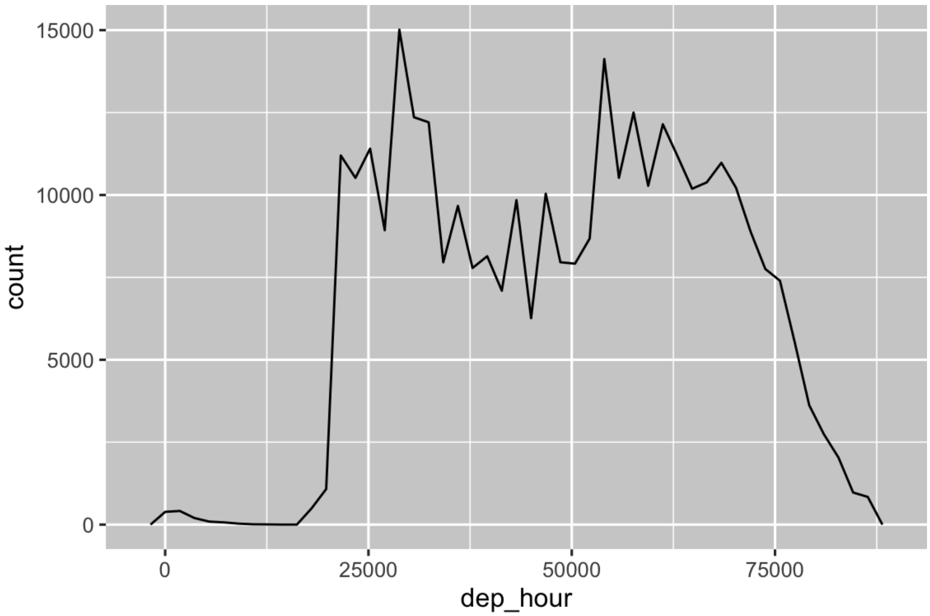
An alternative approach to plotting individual components is to round the date to a nearby unit of time, with `floor_date()`, `round_date()`, and `ceiling_date()`. Each function takes a vector of dates to adjust and then the name of the unit to round down (floor), round up (ceiling), or round to. This, for example, allows us to plot the number of flights per week:

```
flights_dt |>
  count(week = floor_date(dep_time, "week")) |>
  ggplot(aes(x = week, y = n)) +
  geom_line() +
  geom_point()
```



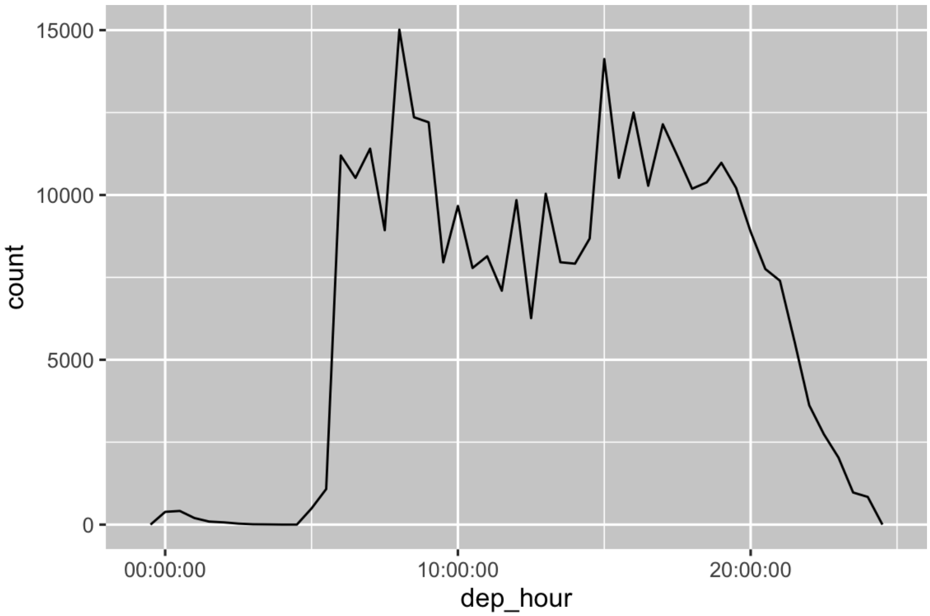
You can use rounding to show the distribution of flights across the course of a day by computing the difference between `dep_time` and the earliest instant of that day:

```
flights_dt |>
  mutate(dep_hour = dep_time - floor_date(dep_time, "day")) |>
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)
#> Don't know how to automatically pick scale for object of type <difftime>.
#> Defaulting to continuous.
```



Computing the difference between a pair of date-times yields a difftime (more on that in "Intervals" on page 316). We can convert that to an hms object to get a more useful x-axis:

```
flights_dt |>
  mutate(dep_hour = hms::as_hms(dep_time - floor_date(dep_time, "day"))) |>
  ggplot(aes(x = dep_hour)) +
  geom_freqpoly(binwidth = 60 * 30)
```



Modifying Components

You can also use each accessor function to modify the components of a date/time. This doesn't come up much in data analysis but can be useful when cleaning data that has clearly incorrect dates.

```
(datetime <- ymd_hms("2026-07-08 12:34:56"))
#> [1] "2026-07-08 12:34:56 UTC"

year(datetime) <- 2030
datetime
#> [1] "2030-07-08 12:34:56 UTC"
month(datetime) <- 01
datetime
#> [1] "2030-01-08 12:34:56 UTC"
hour(datetime) <- hour(datetime) + 1
datetime
#> [1] "2030-01-08 13:34:56 UTC"
```

Alternatively, rather than modifying an existing variable, you can create a new date-time with `update()`. This also allows you to set multiple values in one step:

```
update(datetime, year = 2030, month = 2, mday = 2, hour = 2)
#> [1] "2030-02-02 02:34:56 UTC"
```

If values are too big, they will roll over:

```
update(ymd("2023-02-01"), mday = 30)
#> [1] "2023-03-02"
```

```
update(ymd("2023-02-01"), hour = 400)
#> [1] "2023-02-17 16:00:00 UTC"
```

Exercises

1. How does the distribution of flight times within a day change over the course of the year?
2. Compare `dep_time`, `sched_dep_time`, and `dep_delay`. Are they consistent? Explain your findings.
3. Compare `air_time` with the duration between the departure and arrival. Explain your findings. (Hint: Consider the location of the airport.)
4. How does the average delay time change over the course of a day? Should you use `dep_time` or `sched_dep_time`? Why?
5. On what day of the week should you leave if you want to minimize the chance of a delay?
6. What makes the distribution of `diamonds$carat` and `flights$sched_dep_time` similar?
7. Confirm our hypothesis that the early departures of flights in minutes 20–30 and 50–60 are caused by scheduled flights that leave early. Hint: Create a binary variable that tells you whether a flight was delayed.

Time Spans

Next you'll learn about how arithmetic with dates works, including subtraction, addition, and division. Along the way, you'll learn about three important classes that represent time spans:

Durations
: Represent an exact number of seconds

Periods
: Represent human units like weeks and months

Intervals
: Represent a starting and ending point

How do you pick between duration, periods, and intervals? As always, pick the simplest data structure that solves your problem. If you care only about physical time, use a duration; if you need to add human times, use a period; and if you need to figure out how long a span is in human units, use an interval.

Durations

In R, when you subtract two dates, you get a difftime object:

```
# How old is Hadley?
h_age <- today() - ymd("1979-10-14")
h_age
#> Time difference of 15855 days
```

A difftime class object records a time span of seconds, minutes, hours, days, or weeks. This ambiguity can make difftimes a little painful to work with, so lubridate provides an alternative that always uses seconds: the *duration*.

```
as.duration(h_age)
#> [1] "1369872000s (~43.41 years)"
```

Durations come with a bunch of convenient constructors:

```
dseconds(15)
#> [1] "15s"
dminutes(10)
#> [1] "600s (~10 minutes)"
dhours(c(12, 24))
#> [1] "43200s (~12 hours)" "86400s (~1 days)"
ddays(0:5)
#> [1] "0s"                "86400s (~1 days)"  "172800s (~2 days)"
#> [4] "259200s (~3 days)" "345600s (~4 days)" "432000s (~5 days)"
dweeks(3)
#> [1] "1814400s (~3 weeks)"
dyears(1)
#> [1] "31557600s (~1 years)"
```

Durations always record the time span in seconds. Larger units are created by converting minutes, hours, days, weeks, and years to seconds: 60 seconds in a minute, 60 minutes in an hour, 24 hours in a day, and 7 days in a week. Larger time units are more problematic. A year uses the "average" number of days in a year, i.e., 365.25. There's no way to convert a month to a duration, because there's just too much variation.

You can add and multiply durations:

```
2 * dyears(1)
#> [1] "63115200s (~2 years)"
dyears(1) + dweeks(12) + dhours(15)
#> [1] "38869200s (~1.23 years)"
```

You can add and subtract durations to and from days:

```
tomorrow <- today() + ddays(1)
last_year <- today() - dyears(1)
```

However, because durations represent an exact number of seconds, sometimes you might get an unexpected result:

```
one_am <- ymd_hms("2026-03-08 01:00:00", tz = "America/New_York")

one_am
```

```
#> [1] "2026-03-08 01:00:00 EST"
one_am + ddays(1)
#> [1] "2026-03-09 02:00:00 EDT"
```

Why is one day after 1 a.m. March 8, returning as 2 a.m. on March 9? If you look carefully at the date, you might also notice that the time zones have changed. March 8 has only 23 hours because it's when DST starts, so if we add a full day's worth of seconds, we end up with a different time.

Periods

To solve this problem, lubridate provides *periods*. Periods are time spans but don't have a fixed length in seconds; instead, they work with "human" times, like days and months. That allows them to work in a more intuitive way:

```
one_am
#> [1] "2026-03-08 01:00:00 EST"
one_am + days(1)
#> [1] "2026-03-09 01:00:00 EDT"
```

Like durations, periods can be created with a number of friendly constructor functions:

```
hours(c(12, 24))
#> [1] "12H 0M 0S" "24H 0M 0S"
days(7)
#> [1] "7d 0H 0M 0S"
months(1:6)
#> [1] "1m 0d 0H 0M 0S" "2m 0d 0H 0M 0S" "3m 0d 0H 0M 0S" "4m 0d 0H 0M 0S"
#> [5] "5m 0d 0H 0M 0S" "6m 0d 0H 0M 0S"
```

You can add and multiply periods:

```
10 * (months(6) + days(1))
#> [1] "60m 10d 0H 0M 0S"
days(50) + hours(25) + minutes(2)
#> [1] "50d 25H 2M 0S"
```

And of course, add them to dates. Compared to durations, periods are more likely to do what you expect:

```
# A leap year
ymd("2024-01-01") + dyears(1)
#> [1] "2024-12-31 06:00:00 UTC"
ymd("2024-01-01") + years(1)
#> [1] "2025-01-01"

# Daylight savings time
one_am + ddays(1)
#> [1] "2026-03-09 02:00:00 EDT"
one_am + days(1)
#> [1] "2026-03-09 01:00:00 EDT"
```

Let's use periods to fix an oddity related to our flight dates. Some planes appear to have arrived at their destination *before* they departed from New York City:

```
flights_dt |> 
  filter(arr_time < dep_time) 
#> # A tibble: 10,633 × 9
#>   origin dest  dep_delay arr_delay dep_time            sched_dep_time     
#>   <chr>  <chr>     <dbl>     <dbl> <dttm>              <dttm>             
#> 1 EWR    BQN           9        -4 2013-01-01 19:29:00 2013-01-01 19:20:00
#> 2 JFK    DFW          59        NA 2013-01-01 19:39:00 2013-01-01 18:40:00
#> 3 EWR    TPA          -2         9 2013-01-01 20:58:00 2013-01-01 21:00:00
#> 4 EWR    SJU          -6       -12 2013-01-01 21:02:00 2013-01-01 21:08:00
#> 5 EWR    SFO          11       -14 2013-01-01 21:08:00 2013-01-01 20:57:00
#> 6 LGA    FLL         -10        -2 2013-01-01 21:20:00 2013-01-01 21:30:00
#> # … with 10,627 more rows, and 3 more variables: arr_time <dttm>,
#> #   sched_arr_time <dttm>, air_time <dbl>
```

These are overnight flights. We used the same date information for both the departure and the arrival times, but these flights arrived on the following day. We can fix this by adding `days(1)` to the arrival time of each overnight flight:

```
flights_dt <- flights_dt |> 
  mutate(
    overnight = arr_time < dep_time,
    arr_time = arr_time + days(overnight),
    sched_arr_time = sched_arr_time + days(overnight)
  )
```

Now all of our flights obey the laws of physics:

```
flights_dt |> 
  filter(arr_time < dep_time) 
#> # A tibble: 0 × 10
# … with 10 variables: origin <chr>, dest <chr>, dep_delay <dbl>,
#   arr_delay <dbl>, dep_time <dttm>, sched_dep_time <dttm>, …
# ℹ Use `colnames()` to see all variable names
#> # … with 10,627 more rows, and 4 more variables:
```

Intervals

What does `dyears(1) / ddays(365)` return? It's not quite 1, because `dyears()` is defined as the number of seconds per average year, which is 365.25 days.

What does `years(1) / days(1)` return? Well, if the year is 2015, it should return 365, but if it is 2016, it should return 366! There's not quite enough information for lubridate to give a single clear answer. What it does instead is give an estimate:

```
years(1) / days(1)
#> [1] 365.25
```

If you want a more accurate measurement, you'll have to use an *interval*. An interval is a pair of starting and ending date times, or you can think of it as a duration with a starting point.

You can create an interval by writing `start %--% end`:

```
y2023 <- ymd("2023-01-01") %--% ymd("2024-01-01")
y2024 <- ymd("2024-01-01") %--% ymd("2025-01-01")

y2023
#> [1] 2023-01-01 UTC--2024-01-01 UTC
y2024
#> [1] 2024-01-01 UTC--2025-01-01 UTC
```

You could then divide it by `days()` to find out how many days fit in the year:

```
y2023 / days(1)
#> [1] 365
y2024 / days(1)
#> [1] 366
```

Exercises

1. Explain `days(!overnight)` and `days(overnight)` to someone who has just started learning R. What is the key fact you need to know?
2. Create a vector of dates giving the first day of every month in 2015. Create a vector of dates giving the first day of every month in the *current* year.
3. Write a function that, given your birthday (as a date), returns how old you are in years.
4. Why can't `(today() %--% (today() + years(1))) / months(1)` work?

Time Zones

Time zones are an enormously complicated topic because of their interaction with geopolitical entities. Fortunately we don't need to dig into all the details as they're not all important for data analysis, but there are a few challenges we'll need to tackle head on.

The first challenge is that everyday names of time zones tend to be ambiguous. For example, if you're American, you're probably familiar with Eastern Standard Time (EST). However, both Australia and Canada also have EST! To avoid confusion, R uses the international standard IANA time zones. These use a consistent naming scheme `{area}/{location}`, typically in the form `{continent}/{city}` or `{ocean}/{city}`. Examples include "America/New_York," "Europe/Paris," and "Pacific/Auckland."

You might wonder why the time zone uses a city when typically you think of time zones as associated with a country or region within a country. This is because the IANA database has to record decades worth of time zone rules. Over the course of decades, countries change names (or break apart) fairly frequently, but city names tend to stay the same. Another problem is that the name needs to reflect not only the current behavior but also the complete history. For example, there are time zones for both "America/New_York" and "America/Detroit." These cities both currently use Eastern Standard Time, but in 1969–1972 Michigan (the state in which Detroit is located) did not follow DST, so it needs a different name. It's worth reading the raw time zone database just to read some of these stories!

You can find out what R thinks your current time zone is with `Sys.timezone()`:

```
Sys.timezone()
#> [1] "America/Chicago"
```

(If R doesn't know, you'll get an `NA`.)

And see the complete list of all time zone names with `OlsonNames()`:

```
length(OlsonNames())
#> [1] 597
head(OlsonNames())
#> [1] "Africa/Abidjan"     "Africa/Accra"       "Africa/Addis_Ababa"
#> [4] "Africa/Algiers"     "Africa/Asmara"      "Africa/Asmera"
```

In R, the time zone is an attribute of the date-time that only controls printing. For example, these three objects represent the same instant in time:

```
x1 <- ymd_hms("2024-06-01 12:00:00", tz = "America/New_York")
x1
#> [1] "2024-06-01 12:00:00 EDT"

x2 <- ymd_hms("2024-06-01 18:00:00", tz = "Europe/Copenhagen")
x2
#> [1] "2024-06-01 18:00:00 CEST"

x3 <- ymd_hms("2024-06-02 04:00:00", tz = "Pacific/Auckland")
x3
#> [1] "2024-06-02 04:00:00 NZST"
```

You can verify that they're the same time using subtraction:

```
x1 - x2
#> Time difference of 0 secs
x1 - x3
#> Time difference of 0 secs
```

Unless otherwise specified, lubridate always uses UTC. UTC is the standard time zone used by the scientific community and is roughly equivalent to GMT. It does not have DST, which makes a convenient representation for computation. Operations that combine date-times, like `c()`, will often drop the time zone. In that case, the date-times will display in the time zone of the first element:

```
x4 <- c(x1, x2, x3)
x4
#> [1] "2024-06-01 12:00:00 EDT" "2024-06-01 12:00:00 EDT"
#> [3] "2024-06-01 12:00:00 EDT"
```

You can change the time zone in two ways:

- Keep the instant in time the same, and change how it's displayed. Use this when the instant is correct but you want a more natural display.

  ```
  x4a <- with_tz(x4, tzone = "Australia/Lord_Howe")
  x4a
  #> [1] "2024-06-02 02:30:00 +1030" "2024-06-02 02:30:00 +1030"
  #> [3] "2024-06-02 02:30:00 +1030"
  x4a - x4
  #> Time differences in secs
  #> [1] 0 0 0
  ```

  (This also illustrates another challenge of time zones: they're not all integer hour offsets!)

- Change the underlying instant in time. Use this when you have an instant that has been labeled with the incorrect time zone and you need to fix it.

  ```
  x4b <- force_tz(x4, tzone = "Australia/Lord_Howe")
  x4b
  #> [1] "2024-06-01 12:00:00 +1030" "2024-06-01 12:00:00 +1030"
  #> [3] "2024-06-01 12:00:00 +1030"
  x4b - x4
  #> Time differences in hours
  #> [1] -14.5 -14.5 -14.5
  ```

Summary

This chapter introduced you to the tools that lubridate provides to help you work with date-time data. Working with dates and times can seem harder than necessary, but we hope this chapter has helped you see why—date-times are more complex than they seem at first glance, and handling every possible situation adds complexity. Even if your data never crosses a DST boundary or involves a leap year, the functions need to be able to handle it.

The next chapter gives a roundup of missing values. You've seen them in a few places and have no doubt encountered them in your own analysis, and it's now time to provide a grab bag of useful techniques for dealing with them.

CHAPTER 18

Missing Values

Introduction

You’ve already learned the basics of missing values earlier in the book. You first saw them in Chapter 1 where they resulted in a warning when making a plot as well as in “summarize()” on page 54 where they interfered with computing summary statistics, and you learned about their infectious nature and how to check for their presence in “Missing Values” on page 208. Now we’ll come back to them in more depth so you can learn more of the details.

We’ll start by discussing some general tools for working with missing values recorded as NAs. We’ll then explore the idea of implicitly missing values, values are that are simply absent from your data, and show some tools you can use to make them explicit. We’ll finish off with a related discussion of empty groups, caused by factor levels that don’t appear in the data.

Prerequisites

The functions for working with missing data mostly come from dplyr and tidyr, which are core members of the tidyverse.

```
library(tidyverse)
```

Explicit Missing Values

To begin, let’s explore a few handy tools for creating or eliminating missing explicit values, i.e., cells where you see an NA.

Last Observation Carried Forward

A common use for missing values is as a data entry convenience. When data is entered by hand, missing values sometimes indicate that the value in the previous row has been repeated (or carried forward):

```
treatment <- tribble(
  ~person,           ~treatment, ~response,
  "Derrick Whitmore", 1,         7,
  NA,                 2,         10,
  NA,                 3,         NA,
  "Katherine Burke",  1,         4
)
```

You can fill in these missing values with `tidyr::fill()`. It works like `select()`, taking a set of columns:

```
treatment |>
  fill(everything())
#> # A tibble: 4 × 3
#>   person           treatment response
#>   <chr>                <dbl>    <dbl>
#> 1 Derrick Whitmore         1        7
#> 2 Derrick Whitmore         2       10
#> 3 Derrick Whitmore         3       10
#> 4 Katherine Burke          1        4
```

This treatment is sometimes called "last observation carried forward," or *locf* for short. You can use the `.direction` argument to fill in missing values that have been generated in more exotic ways.

Fixed Values

Sometimes missing values represent some fixed and known value, most commonly 0. You can use `dplyr::coalesce()` to replace them:

```
x <- c(1, 4, 5, 7, NA)
coalesce(x, 0)
#> [1] 1 4 5 7 0
```

Sometimes you'll hit the opposite problem where some concrete value actually represents a missing value. This typically arises in data generated by older software that doesn't have a proper way to represent missing values, so it must instead use some special value like 99 or -999.

If possible, handle this when reading in the data, for example, by using the `na` argument to `readr::read_csv()`, e.g., `read_csv(path, na = "99")`. If you discover the problem later or your data source doesn't provide a way to handle it on read, you can use `dplyr::na_if()`:

```
x <- c(1, 4, 5, 7, -99)
na_if(x, -99)
#> [1]  1  4  5  7 NA
```

NaN

Before we continue, there's one special type of missing value that you'll encounter from time to time: a `NaN` (pronounced "nan"), or not a number. It's not that important to know about because it generally behaves just like `NA`:

```
x <- c(NA, NaN)
x * 10
#> [1]  NA NaN
x == 1
#> [1] NA NA
is.na(x)
#> [1] TRUE TRUE
```

In the rare case you need to distinguish an `NA` from a `NaN`, you can use `is.nan(x)`.

You'll generally encounter a `NaN` when you perform a mathematical operation that has an indeterminate result:

```
0 / 0
#> [1] NaN
0 * Inf
#> [1] NaN
Inf - Inf
#> [1] NaN
sqrt(-1)
#> Warning in sqrt(-1): NaNs produced
#> [1] NaN
```

Implicit Missing Values

So far we've talked about missing values that are *explicitly* missing; i.e., you can see an `NA` in your data. But missing values can also be *implicitly* missing, if an entire row of data is simply absent from the data. Let's illustrate the difference with a simple dataset that records the price of some stock each quarter:

```
stocks <- tibble(
  year  = c(2020, 2020, 2020, 2020, 2021, 2021, 2021),
  qtr   = c(   1,    2,    3,    4,    2,    3,    4),
  price = c(1.88, 0.59, 0.35,   NA, 0.92, 0.17, 2.66)
)
```

This dataset has two missing observations:

- The `price` in the fourth quarter of 2020 is explicitly missing, because its value is `NA`.
- The `price` for the first quarter of 2021 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.

Sometimes you want to make implicit missings explicit to have something physical to work with. In other cases, explicit missings are forced upon you by the structure of the data, and you want to get rid of them. The following sections discuss some tools for moving between implicit and explicit missingness.

Pivoting

You've already seen one tool that can make implicit missings explicit, and vice versa: pivoting. Making data wider can make implicit missing values explicit because every combination of the rows and new columns must have some value. For example, if we pivot `stocks` to put the `quarter` in the columns, both missing values become explicit:

```
stocks |>
  pivot_wider(
    names_from = qtr,
    values_from = price
  )
#> # A tibble: 2 × 5
#>    year   `1`   `2`   `3`   `4`
#>   <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  2020  1.88  0.59  0.35 NA
#> 2  2021 NA     0.92  0.17  2.66
```

By default, making data longer preserves explicit missing values, but if they are structurally missing values that exist only because the data is not tidy, you can drop them (make them implicit) by setting `values_drop_na = TRUE`. See the examples in "Tidy Data" on page 70 for more details.

Complete

`tidyr::complete()` allows you to generate explicit missing values by providing a set of variables that define the combination of rows that should exist. For example, we know that all combinations of `year` and `qtr` should exist in the `stocks` data:

```
stocks |>
  complete(year, qtr)
#> # A tibble: 8 × 3
#>    year   qtr price
#>   <dbl> <dbl> <dbl>
#> 1  2020     1  1.88
#> 2  2020     2  0.59
#> 3  2020     3  0.35
#> 4  2020     4 NA
#> 5  2021     1 NA
#> 6  2021     2  0.92
#> # … with 2 more rows
```

Typically, you'll call complete() with names of existing variables, filling in the missing combinations. However, sometimes the individual variables are themselves incomplete, so you can instead provide your own data. For example, you might know that the stocks dataset is supposed to run from 2019 to 2021, so you could explicitly supply those values for year:

```
stocks |>
  complete(year = 2019:2021, qtr)
#> # A tibble: 12 × 3
#>    year   qtr price
#>   <dbl> <dbl> <dbl>
#> 1  2019     1 NA
#> 2  2019     2 NA
#> 3  2019     3 NA
#> 4  2019     4 NA
#> 5  2020     1  1.88
#> 6  2020     2  0.59
#> # … with 6 more rows
```

If the range of a variable is correct but not all values are present, you could use full_seq(x, 1) to generate all values from min(x) to max(x) spaced out by 1.

In some cases, the complete set of observations can't be generated by a simple combination of variables. In that case, you can do manually what complete() does for you: create a data frame that contains all the rows that should exist (using whatever combination of techniques you need) and then combine it with your original dataset with dplyr::full_join().

Joins

This brings us to another important way of revealing implicitly missing observations: joins. You'll learn more about joins in Chapter 19, but we wanted to quickly mention them to you here since you can often know that values are missing from one dataset only when you compare it another.

dplyr::anti_join(x, y) is a useful tool here because it selects only the rows in x that don't have a match in y. For example, we can use two anti_join()s to reveal that we're missing information for 4 airports and 722 planes mentioned in flights:

```
library(nycflights13)

flights |>
  distinct(faa = dest) |>
  anti_join(airports)
#> Joining with `by = join_by(faa)`
#> # A tibble: 4 × 1
#>   faa
#>   <chr>
#> 1 BQN
#> 2 SJU
#> 3 STT
#> 4 PSE
```

```
flights |>
  distinct(tailnum) |>
  anti_join(planes)
#> Joining with `by = join_by(tailnum)`
#> # A tibble: 722 × 1
#>   tailnum
#>   <chr>
#> 1 N3ALAA
#> 2 N3DUAA
#> 3 N542MQ
#> 4 N730MQ
#> 5 N9EAMQ
#> 6 N532UA
#> # … with 716 more rows
```

Exercises

1. Can you find any relationship between the carrier and the rows that appear to be missing from `planes`?

Factors and Empty Groups

A final type of missingness is the empty group, a group that doesn't contain any observations, which can arise when working with factors. For example, imagine we have a dataset that contains some health information about people:

```
health <- tibble(
  name   = c("Ikaia", "Oletta", "Leriah", "Dashay", "Tresaun"),
  smoker = factor(c("no", "no", "no", "no", "no"), levels = c("yes", "no")),
  age    = c(34, 88, 75, 47, 56),
)
```

And say we want to count the number of smokers with `dplyr::count()`:

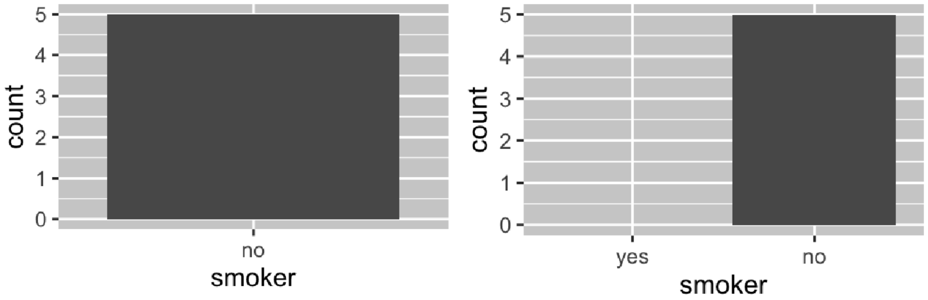
```
health |> count(smoker)
#> # A tibble: 1 × 2
#>   smoker     n
#>   <fct>  <int>
#> 1 no         5
```

This dataset contains only nonsmokers, but we know that smokers exist; the group of nonsmoker is empty. We can request `count()` to keep all the groups, even those not seen in the data, by using `.drop = FALSE`:

```
health |> count(smoker, .drop = FALSE)
#> # A tibble: 2 × 2
#>   smoker     n
#>   <fct>  <int>
#> 1 yes        0
#> 2 no         5
```

The same principle applies to ggplot2's discrete axes, which will also drop levels that don't have any values. You can force them to display by supplying `drop = FALSE` to the appropriate discrete axis:

```
ggplot(health, aes(x = smoker)) +
  geom_bar() +
  scale_x_discrete()

ggplot(health, aes(x = smoker)) +
  geom_bar() +
  scale_x_discrete(drop = FALSE)
```



The same problem comes up more generally with `dplyr::group_by()`. And again you can use `.drop = FALSE` to preserve all factor levels:

```
health |>
  group_by(smoker, .drop = FALSE) |>
  summarize(
    n = n(),
    mean_age = mean(age),
    min_age = min(age),
    max_age = max(age),
    sd_age = sd(age)
  )
#> # A tibble: 2 × 6
#>   smoker     n mean_age min_age max_age sd_age
#>   <fct>  <int>    <dbl>   <dbl>   <dbl>  <dbl>
#> 1 yes        0      NaN     Inf    -Inf   NA  
#> 2 no         5       60      34      88   21.6
```

We get some interesting results here because when summarizing an empty group, the summary functions are applied to zero-length vectors. There's an important distinction between empty vectors, which have length 0, and missing values, each of which has length 1.

```
# A vector containing two missing values
x1 <- c(NA, NA)
length(x1)
#> [1] 2

# A vector containing nothing
x2 <- numeric()
```

```
length(x2)
#> [1] 0
```

All summary functions work with zero-length vectors, but they may return results that are surprising at first glance. Here we see `mean(age)` returning `NaN` because `mean(age)` = `sum(age)/length(age)`, which here is 0/0. `max()` and `min()` return -Inf and Inf for empty vectors, so if you combine the results with a nonempty vector of new data and recompute, you'll get the minimum or maximum of the new data.[1]

Sometimes a simpler approach is to perform the summary and then make the implicit missings explicit with `complete()`:

```
health |>
  group_by(smoker) |>
  summarize(
    n = n(),
    mean_age = mean(age),
    min_age = min(age),
    max_age = max(age),
    sd_age = sd(age)
  ) |>
  complete(smoker)
#> # A tibble: 2 × 6
#>   smoker     n mean_age min_age max_age sd_age
#>   <fct>  <int>    <dbl>   <dbl>   <dbl>  <dbl>
#> 1 yes       NA       NA      NA      NA   NA
#> 2 no         5       60      34      88   21.6
```

The main drawback of this approach is that you get an `NA` for the count, even though you know that it should be zero.

Summary

Missing values are weird! Sometimes they're recorded as an explicit `NA`, but other times you notice them only by their absence. This chapter has given you some tools for working with explicit missing values and some tools for uncovering implicit missing values, and we discussed some of the ways that implicit can become explicit, and vice versa.

In the next chapter, we tackle the final chapter in this part of the book: joins. This is a bit of a change from the chapters so far because we're going to discuss tools that work with data frames as a whole, not something that you put inside a data frame.

[1] In other words, `min(c(x, y))` is always equal to `min(min(x), min(y))`.

CHAPTER 19

Joins

Introduction

It's rare that a data analysis involves only a single data frame. Typically you have many data frames, and you must *join* them together to answer the questions that you're interested in. This chapter will introduce you to two important types of joins:

- 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 they match an observation in another.

We'll begin by discussing keys, the variables used to connect a pair of data frames in a join. We cement the theory with an examination of the keys in the datasets from the nycflights13 package and then use that knowledge to start joining data frames together. Next we'll discuss how joins work, focusing on their action on the rows. We'll finish up with a discussion of non-equi joins, a family of joins that provide a more flexible way of matching keys than the default equality relationship.

Prerequisites

In this chapter, we'll explore the five related datasets from nycflights13 using the join functions from dplyr.

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

Keys

To understand joins, you need to first understand how two tables can be connected through a pair of keys, within each table. In this section, you'll learn about the two types of key and see examples of both in the datasets of the nycflights13 package. You'll also learn how to check that your keys are valid and what to do if your table lacks a key.

Primary and Foreign Keys

Every join involves a pair of keys: a primary key and a foreign key. A *primary key* is a variable or set of variables that uniquely identifies each observation. When more than one variable is needed, the key is called a *compound key*. For example, in nycflights13:

- `airlines` records two pieces of data about each airline: its carrier code and its full name. You can identify an airline with its two-letter carrier code, making `carrier` the primary key.

```
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` records data about each airport. You can identify each airport by its three-letter airport code, making `faa` the primary key.

```
airports
#> # A tibble: 1,458 × 8
#>   faa   name                            lat   lon   alt    tz dst
#>   <chr> <chr>                         <dbl> <dbl> <dbl> <dbl> <chr>
#> 1 04G   Lansdowne Airport              41.1 -80.6  1044    -5 A
#> 2 06A   Moton Field Municipal Airport  32.5 -85.7   264    -6 A
#> 3 06C   Schaumburg Regional            42.0 -88.1   801    -6 A
#> 4 06N   Randall Airport                41.4 -74.4   523    -5 A
#> 5 09J   Jekyll Island Airport          31.1 -81.4    11    -5 A
#> 6 0A9   Elizabethton Municipal Airpo…  36.4 -82.2  1593    -5 A
#> # … with 1,452 more rows, and 1 more variable: tzone <chr>
```

- `planes` records data about each plane. You can identify a plane by its tail number, making `tailnum` the primary key.

```
planes
#> # A tibble: 3,322 × 9
#>   tailnum  year type              manufacturer    model     engines
#>   <chr>   <int> <chr>             <chr>           <chr>       <int>
#> 1 N10156   2004 Fixed wing multi… EMBRAER         EMB-145XR       2
#> 2 N102UW   1998 Fixed wing multi… AIRBUS INDUSTR… A320-214        2
#> 3 N103US   1999 Fixed wing multi… AIRBUS INDUSTR… A320-214        2
#> 4 N104UW   1999 Fixed wing multi… AIRBUS INDUSTR… A320-214        2
#> 5 N10575   2002 Fixed wing multi… EMBRAER         EMB-145LR       2
#> 6 N105UW   1999 Fixed wing multi… AIRBUS INDUSTR… A320-214        2
#> # … with 3,316 more rows, and 3 more variables: seats <int>,
#> #   speed <int>, engine <chr>
```

- `weather` records data about the weather at the origin airports. You can identify each observation by the combination of location and time, making `origin` and `time_hour` the compound primary key.

```
weather
#> # A tibble: 26,115 × 15
#>   origin  year month   day  hour  temp  dewp humid wind_dir
#>   <chr>  <int> <int> <int> <int> <dbl> <dbl> <dbl>    <dbl>
#> 1 EWR     2013     1     1     1  39.0  26.1  59.4      270
#> 2 EWR     2013     1     1     2  39.0  27.0  61.6      250
#> 3 EWR     2013     1     1     3  39.0  28.0  64.4      240
#> 4 EWR     2013     1     1     4  39.9  28.0  62.2      250
#> 5 EWR     2013     1     1     5  39.0  28.0  64.4      260
#> 6 EWR     2013     1     1     6  37.9  28.0  67.2      240
#> # … with 26,109 more rows, and 6 more variables: wind_speed <dbl>,
#> #   wind_gust <dbl>, precip <dbl>, pressure <dbl>, visib <dbl>, …
```

A *foreign key* is a variable (or set of variables) that corresponds to a primary key in another table. For example:

- `flights$tailnum` is a foreign key that corresponds to the primary key `planes$tailnum`.
- `flights$carrier` is a foreign key that corresponds to the primary key `airlines$carrier`.
- `flights$origin` is a foreign key that corresponds to the primary key `airports$faa`.
- `flights$dest` is a foreign key that corresponds to the primary key `airports$faa`.
- `flights$origin-flights$time_hour` is a compound foreign key that corresponds to the compound primary key `weather$origin-weather$time_hour`.

These relationships are summarized visually in Figure 19-1.

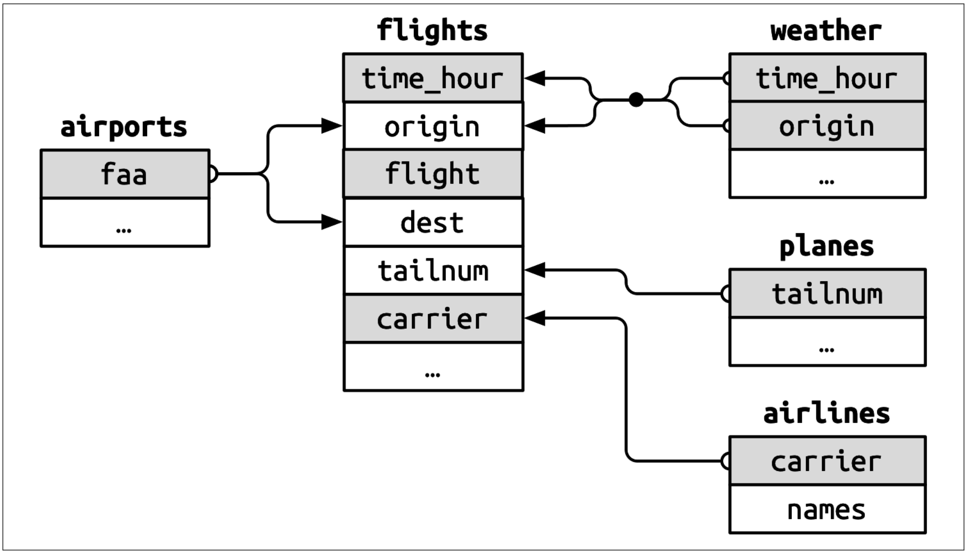


Figure 19-1. Connections between all five data frames in the nycflights13 package. Variables making up a primary key are gray and are connected to their corresponding foreign keys with arrows.

You'll notice a nice feature in the design of these keys: the primary and foreign keys almost always have the same names, which, as you'll see shortly, will make your joining life much easier. It's also worth noting the opposite relationship: almost every variable name used in multiple tables has the same meaning in each place. There's only one exception: `year` means year of departure in `flights` and year of manufacturer in `planes`. This will become important when we start actually joining tables together.

Checking Primary Keys

Now that that we've identified the primary keys in each table, 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. This reveals that `planes` and `weather` both look good:

```
planes |>
  count(tailnum) |>
  filter(n > 1)
#> # A tibble: 0 × 2
#> # … with 2 variables: tailnum <chr>, n <int>

weather |>
  count(time_hour, origin) |>
  filter(n > 1)
#> # A tibble: 0 × 3
#> # … with 3 variables: time_hour <dttm>, origin <chr>, n <int>
```

You should also check for missing values in your primary keys—if a value is missing, then it can't identify an observation!

```
planes |>
  filter(is.na(tailnum))
#> # A tibble: 0 × 9
#> # … with 9 variables: tailnum <chr>, year <int>, type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>, …

weather |>
  filter(is.na(time_hour) | is.na(origin))
#> # A tibble: 0 × 15
#> # … with 15 variables: origin <chr>, year <int>, month <int>, day <int>,
#> #   hour <int>, temp <dbl>, dewp <dbl>, humid <dbl>, wind_dir <dbl>, …
```

Surrogate Keys

So far we haven't talked about the primary key for `flights`. It's not super important here, because there are no data frames that use it as a foreign key, but it's still useful to consider because it's easier to work with observations if we have some way to describe them to others.

After a little thinking and experimentation, we determined that there are three variables that together uniquely identify each flight:

```
flights |>
  count(time_hour, carrier, flight) |>
  filter(n > 1)
#> # A tibble: 0 × 4
#> # … with 4 variables: time_hour <dttm>, carrier <chr>, flight <int>, n <int>
```

Does the absence of duplicates automatically make `time_hour`-`carrier`-`flight` a primary key? It's certainly a good start, but it doesn't guarantee it. For example, are altitude and latitude a good primary key for `airports`?

```
airports |>
  count(alt, lat) |>
  filter(n > 1)
#> # A tibble: 1 × 3
#>     alt   lat     n
#>   <dbl> <dbl> <int>
#> 1    13  40.6     2
```

Identifying an airport by its altitude and latitude is clearly a bad idea, and in general it's not possible to know from the data alone whether a combination of variables makes a good primary key. But for flights, the combination of `time_hour`, `carrier`, and `flight` seems reasonable because it would be really confusing for an airline and its customers if there were multiple flights with the same flight number in the air at the same time.

That said, we might be better off introducing a simple numeric surrogate key using the row number:

```
flights2 <- flights |>
  mutate(id = row_number(), .before = 1)
flights2
#> # A tibble: 336,776 × 20
#>      id  year month   day dep_time sched_dep_time dep_delay arr_time
#>   <int> <int> <int> <int>    <int>          <int>     <dbl>    <int>
#> 1     1  2013     1     1      517            515         2      830
#> 2     2  2013     1     1      533            529         4      850
#> 3     3  2013     1     1      542            540         2      923
#> 4     4  2013     1     1      544            545        -1     1004
#> 5     5  2013     1     1      554            600        -6      812
#> 6     6  2013     1     1      554            558        -4      740
#> # … with 336,770 more rows, and 12 more variables: sched_arr_time <int>,
#> #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>, …
```

Surrogate keys can be particularly useful when communicating to other humans: it's much easier to tell someone to take a look at flight 2001 than to say look at UA430, which departed at 9 a.m. on January 3, 2013.

Exercises

1. We forgot to draw the relationship between `weather` and `airports` in Figure 19-1. What is the relationship, and how should it appear in the diagram?
2. `weather` contains information for only the three origin airports in NYC. If it contained weather records for all airports in the US, what additional connection would it make to `flights`?
3. The `year`, `month`, `day`, `hour`, and `origin` variables almost form a compound key for `weather`, but there's one hour that has duplicate observations. Can you figure out what's special about that hour?
4. We know that some days of the year are special and fewer people than usual fly on them (e.g., Christmas Eve and Christmas Day). How might you represent that data as a data frame? What would be the primary key? How would it connect to the existing data frames?
5. Draw a diagram illustrating the connections between the `Batting`, `People`, and `Salaries` data frames in the Lahman package. Draw another diagram that shows the relationship between `People`, `Managers`, and `AwardsManagers`. How would you characterize the relationship between the `Batting`, `Pitching`, and `Fielding` data frames?

Basic Joins

Now that you understand how data frames are connected via keys, we can start using joins to better understand the `flights` dataset. dplyr provides six join functions:

- `left_join()`

- `inner_join()`
- `right_join()`
- `full_join()`
- `semi_join()`
- `anti_join()`

They all have the same interface: they take a pair of data frames (`x` and `y`) and return a data frame. The order of the rows and columns in the output is primarily determined by `x`.

In this section, you'll learn how to use one mutating join, `left_join()`, and two filtering joins, `semi_join()` and `anti_join()`. In the next section, you'll learn exactly how these functions work and about the remaining `inner_join()`, `right_join()`, and `full_join()`.

Mutating Joins

A *mutating join* allows you to combine variables from two data frames: it first matches observations by their keys and then copies across variables from one data frame to the other. Like `mutate()`, the join functions add variables to the right, so if your dataset has many variables, you won't see the new ones. For these examples, we'll make it easier to see what's going on by creating a narrower dataset with just six variables:[1]

```
flights2 <- flights |>
  select(year, time_hour, origin, dest, tailnum, carrier)
flights2
#> # A tibble: 336,776 × 6
#>    year time_hour           origin dest  tailnum carrier
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>  
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA     
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA     
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA     
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6     
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL     
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA     
#> # … with 336,770 more rows
```

There are four types of mutating join, but there's one that you'll use almost all of the time: `left_join()`. It's special because the output will always have the same rows as `x`.[2] The primary use of `left_join()` is to add additional metadata. For example, we can use `left_join()` to add the full airline name to the `flights2` data:

[1] Remember that in RStudio you can also use `View()` to avoid this problem.

[2] That's not 100% true, but you'll get a warning whenever it isn't.

```
flights2 |>
  left_join(airlines)
#> Joining with `by = join_by(carrier)`
#> # A tibble: 336,776 × 7
#>    year time_hour           origin dest  tailnum carrier name
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <chr>
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA      United Air Lines In…
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA      United Air Lines In…
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      American Airlines I…
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      JetBlue Airways
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      Delta Air Lines Inc.
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA      United Air Lines In…
#> # … with 336,770 more rows
```

Or we could find out the temperature and wind speed when each plane departed:

```
flights2 |>
  left_join(weather |> select(origin, time_hour, temp, wind_speed))
#> Joining with `by = join_by(time_hour, origin)`
#> # A tibble: 336,776 × 8
#>    year time_hour           origin dest  tailnum carrier  temp wind_speed
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <dbl>      <dbl>
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA       39.0       12.7
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA       39.9       15.0
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA       39.0       15.0
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6       39.0       15.0
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL       39.9       16.1
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA       39.0       12.7
#> # … with 336,770 more rows
```

Or what size of plane was flying:

```
flights2 |>
  left_join(planes |> select(tailnum, type, engines, seats))
#> Joining with `by = join_by(tailnum)`
#> # A tibble: 336,776 × 9
#>    year time_hour           origin dest  tailnum carrier type
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <chr>
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA      Fixed wing multi en…
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA      Fixed wing multi en…
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      Fixed wing multi en…
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      Fixed wing multi en…
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      Fixed wing multi en…
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA      Fixed wing multi en…
#> # … with 336,770 more rows, and 2 more variables: engines <int>, seats <int>
```

When `left_join()` fails to find a match for a row in `x`, it fills in the new variables with missing values. For example, there's no information about the plane with tail number `N3ALAA` so the `type`, `engines`, and `seats` will be missing:

```
flights2 |>
  filter(tailnum == "N3ALAA") |>
  left_join(planes |> select(tailnum, type, engines, seats))
#> Joining with `by = join_by(tailnum)`
#> # A tibble: 63 × 9
#>    year time_hour           origin dest  tailnum carrier type  engines seats
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <chr>   <int> <int>
#> 1  2013 2013-01-01 06:00:00 LGA    ORD   N3ALAA  AA      <NA>       NA    NA
#> 2  2013 2013-01-02 18:00:00 LGA    ORD   N3ALAA  AA      <NA>       NA    NA
#> 3  2013 2013-01-03 06:00:00 LGA    ORD   N3ALAA  AA      <NA>       NA     NA
#> 4  2013 2013-01-07 19:00:00 LGA    ORD   N3ALAA  AA      <NA>       NA     NA
#> 5  2013 2013-01-08 17:00:00 JFK    ORD   N3ALAA  AA      <NA>       NA     NA
#> 6  2013 2013-01-16 06:00:00 LGA    ORD   N3ALAA  AA      <NA>       NA     NA
#> # … with 57 more rows
```

We'll come back to this problem a few times in the rest of the chapter.

Specifying Join Keys

By default, `left_join()` will use all variables that appear in both data frames as the join key, the so-called *natural* join. This is a useful heuristic, but it doesn't always work. For example, what happens if we try to join `flights2` with the complete `planes` dataset?

```
flights2 |>
  left_join(planes)
#> Joining with `by = join_by(year, tailnum)`
#> # A tibble: 336,776 × 13
#>    year time_hour           origin dest  tailnum carrier type  manufacturer
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <chr> <chr>
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA      <NA>  <NA>
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA      <NA>  <NA>
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      <NA>  <NA>
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      <NA>  <NA>
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      <NA>  <NA>
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA      <NA>  <NA>
#> # … with 336,770 more rows, and 5 more variables: model <chr>,
#> #   engines <int>, seats <int>, speed <int>, engine <chr>
```

We get a lot of missing matches because our join is trying to use `tailnum` and `year` as a compound key. Both `flights` and `planes` have a `year` column, but they mean different things: `flights$year` is the year the flight occurred, and `planes$year` is the year the plane was built. We only want to join on `tailnum`, so we need to provide an explicit specification with `join_by()`:

```
flights2 |>
  left_join(planes, join_by(tailnum))
#> # A tibble: 336,776 × 14
#>   year.x time_hour           origin dest  tailnum carrier year.y
#>    <int> <dttm>              <chr>  <chr> <chr>   <chr>    <int>
#> 1   2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA        1999
#> 2   2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA        1998
#> 3   2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA        1990
#> 4   2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6        2012
#> 5   2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL        1991
#> 6   2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA        2012
#> # … with 336,770 more rows, and 7 more variables: type <chr>,
#> #   manufacturer <chr>, model <chr>, engines <int>, seats <int>, …
```

Note that the `year` variables are disambiguated in the output with a suffix (`year.x` and `year.y`), which tells you whether the variable came from the `x` or `y` argument. You can override the default suffixes with the `suffix` argument.

join_by(tailnum) is short for join_by(tailnum == tailnum). It's important to know about this fuller form for two reasons. First, it describes the relationship between the two tables: the keys must be equal. That's why this type of join is often called an *equi join*. You'll learn about non-equi joins in "Filtering Joins" on page 346.

Second, it's how you specify different join keys in each table. For example, there are two ways to join the flight2 and airports table: either by dest or by origin:

```
flights2 |>
  left_join(airports, join_by(dest == faa))
#> # A tibble: 336,776 × 13
#>    year time_hour           origin dest  tailnum carrier name
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <chr>
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA      George Bush Interco…
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA      George Bush Interco…
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      Miami Intl
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      <NA>
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      Hartsfield Jackson …
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA      Chicago Ohare Intl
#> # … with 336,770 more rows, and 6 more variables: lat <dbl>, lon <dbl>,
#> #   alt <dbl>, tz <dbl>, dst <chr>, tzone <chr>

flights2 |>
  left_join(airports, join_by(origin == faa))
#> # A tibble: 336,776 × 13
#>    year time_hour           origin dest  tailnum carrier name
#>   <int> <dttm>              <chr>  <chr> <chr>   <chr>   <chr>
#> 1  2013 2013-01-01 05:00:00 EWR    IAH   N14228  UA      Newark Liberty Intl
#> 2  2013 2013-01-01 05:00:00 LGA    IAH   N24211  UA      La Guardia
#> 3  2013 2013-01-01 05:00:00 JFK    MIA   N619AA  AA      John F Kennedy Intl
#> 4  2013 2013-01-01 05:00:00 JFK    BQN   N804JB  B6      John F Kennedy Intl
#> 5  2013 2013-01-01 06:00:00 LGA    ATL   N668DN  DL      La Guardia
#> 6  2013 2013-01-01 05:00:00 EWR    ORD   N39463  UA      Newark Liberty Intl
#> # … with 336,770 more rows, and 6 more variables: lat <dbl>, lon <dbl>,
#> #   alt <dbl>, tz <dbl>, dst <chr>, tzone <chr>
```

In older code you might see a different way of specifying the join keys, using a character vector:

- by = "x" corresponds to join_by(x).
- by = c("a" = "x") corresponds to join_by(a == x).

Now that it exists, we prefer join_by() since it provides a clearer and more flexible specification.

inner_join(), right_join(), and full_join() have the same interface as left_join(). The difference is which rows they keep: left join keeps all the rows in x, the right join keeps all rows in y, the full join keeps all rows in either x or y, and the inner join keeps only those rows that occur in both x and y. We'll come back to these in more detail later.

Filtering Joins

As you might guess, the primary action of a *filtering join* is to filter the rows. There are two types: semi-joins and anti-joins. *Semi-joins* keep all rows in x that have a match in y. For example, we could use a semi-join to filter the airports dataset to show just the origin airports:

```
airports |>
  semi_join(flights2, join_by(faa == origin))
#> # A tibble: 3 × 8
#>   faa   name                  lat   lon   alt    tz dst   tzone
#>   <chr> <chr>               <dbl> <dbl> <dbl> <dbl> <chr> <chr>
#> 1 EWR   Newark Liberty Intl  40.7 -74.2    18    -5 A     America/New_York
#> 2 JFK   John F Kennedy Intl  40.6 -73.8    13    -5 A     America/New_York
#> 3 LGA   La Guardia           40.8 -73.9    22    -5 A     America/New_York
```

Or just the destinations:

```
airports |>
  semi_join(flights2, join_by(faa == dest))
#> # A tibble: 101 × 8
#>   faa   name                     lat    lon   alt    tz dst   tzone
#>   <chr> <chr>                  <dbl>  <dbl> <dbl> <dbl> <chr> <chr>
#> 1 ABQ   Albuquerque Internati…  35.0 -107.   5355    -7 A     America/Denver
#> 2 ACK   Nantucket Mem           41.3  -70.1    48    -5 A     America/New_Yo…
#> 3 ALB   Albany Intl             42.7  -73.8   285    -5 A     America/New_Yo…
#> 4 ANC   Ted Stevens Anchorage…  61.2 -150.    152    -9 A     America/Anchor…
#> 5 ATL   Hartsfield Jackson At…  33.6  -84.4  1026    -5 A     America/New_Yo…
#> 6 AUS   Austin Bergstrom Intl   30.2  -97.7   542    -6 A     America/Chicago
#> # … with 95 more rows
```

Anti-joins are the opposite: they return all rows in x that don't have a match in y. They're useful for finding missing values that are *implicit* in the data, the topic of "Implicit Missing Values" on page 323. Implicitly missing values don't show up as NAs but instead exist only as an absence. For example, we can find rows that are missing from airports by looking for flights that don't have a matching destination airport:

```
flights2 |>
  anti_join(airports, join_by(dest == faa)) |>
  distinct(dest)
#> # A tibble: 4 × 1
#>   dest
#>   <chr>
#> 1 BQN
#> 2 SJU
#> 3 STT
#> 4 PSE
```

Or we can find which tailnums are missing from planes:

```
flights2 |>
  anti_join(planes, join_by(tailnum)) |>
  distinct(tailnum)
#> # A tibble: 722 × 1
#>   tailnum
#>   <chr>
#> 1 N3ALAA
#> 2 N3DUAA
#> 3 N542MQ
#> 4 N730MQ
#> 5 N9EAMQ
#> 6 N532UA
#> # … with 716 more rows
```

Exercises

1. 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?
2. Imagine you've found the top 10 most popular destinations using this code:

   ```
   top_dest <- flights2 |>
     count(dest, sort = TRUE) |>
     head(10)
   ```

   How can you find all flights to those destinations?
3. Does every departing flight have corresponding weather data for that hour?
4. What do the tail numbers that don't have a matching record in `planes` have in common? (Hint: One variable explains about 90% of the problems.)
5. Add a column to `planes` that lists every `carrier` that has flown that plane. 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 previous chapters.
6. Add the latitude and the longitude of the origin *and* destination airport to `flights`. Is it easier to rename the columns before or after the join?
7. Compute the average delay by destination and 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:

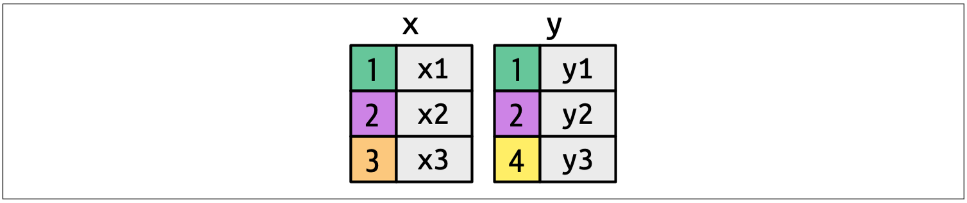
   ```
   airports |>
     semi_join(flights, join_by(faa == dest)) |>
     ggplot(aes(x = lon, y = lat)) +
       borders("state") +
       geom_point() +
       coord_quickmap()
   ```

   You might want to use the `size` or `color` of the points to display the average delay for each airport.
8. What happened on June 13, 2013? Draw a map of the delays, and then use Google to cross-reference with the weather.

How Do Joins Work?

Now that you've used joins a few times, it's time to learn more about how they work, focusing on how each row in x matches rows in y. We'll begin by introducing a visual representation of joins, using the simple tibbles defined next and shown in Figure 19-2. In these examples we'll use a single key called key and a single value column (val_x and val_y), but the ideas all generalize to multiple keys and multiple values.

```
x <- tribble(
  ~key, ~val_x,
     1, "x1",
     2, "x2",
     3, "x3"
)
y <- tribble(
  ~key, ~val_y,
     1, "y1",
     2, "y2",
     4, "y3"
)
```



x	
1	x1
2	x2
3	x3

y	
1	y1
2	y2
4	y3

Figure 19-2. Graphical representation of two simple tables. The colored key columns map background color to key value. The gray columns represent the "value" columns that are carried along for the ride.

Figure 19-3 introduces the foundation for our visual representation. It shows all potential matches between x and y as the intersection between lines drawn from each row of x and each row of y. The rows and columns in the output are primarily determined by x, so the x table is horizontal and lines up with the output.

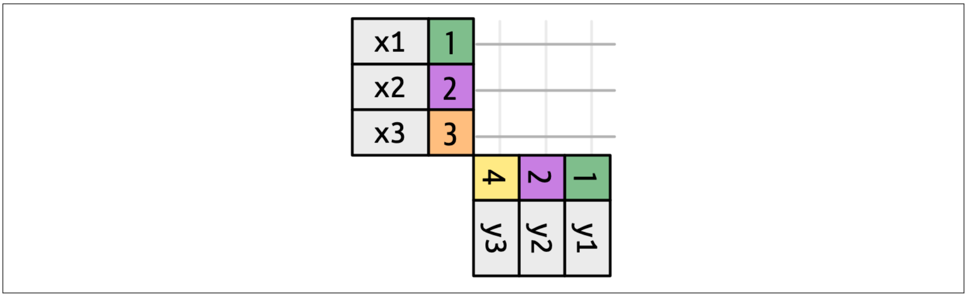
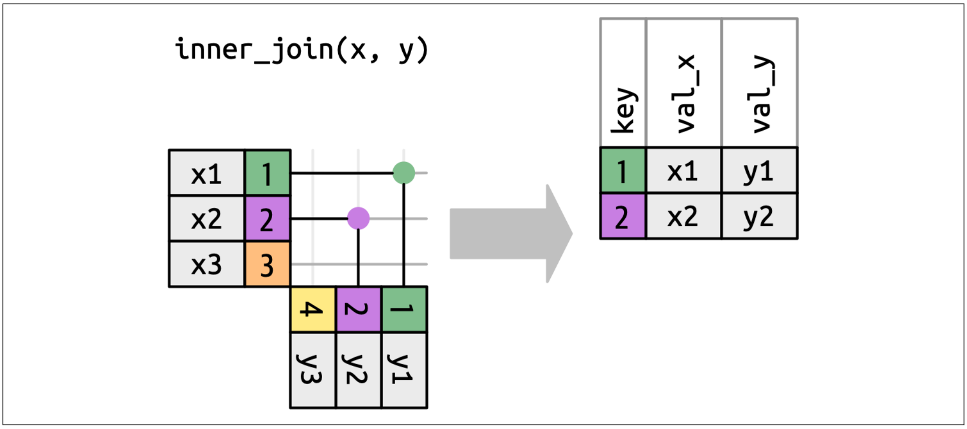


Figure 19-3. To understand how joins work, it's useful to think of every possible match. Here we show that with a grid of connecting lines.

To describe a specific type of join, we indicate matches with dots. The matches determine the rows in the output, a new data frame that contains the key, the x values, and the y values. For example, Figure 19-4 shows an inner join, where rows are retained if and only if the keys are equal.



key	val_x	val_y
1	x1	y1
2	x2	y2

Figure 19-4. An inner join matches each row in x to the row in y that has the same value of `key`. Each match becomes a row in the output.

We can apply the same principles to explain the *outer joins*, which keep observations that appear in at least one of the data frames. These joins work by adding an additional "virtual" observation to each data frame. This observation has a key that matches if no other key matches, as well as values filled with `NA`. There are three types of outer joins:

- A *left join* keeps all observations in `x`, as shown in Figure 19-5. Every row of `x` is preserved in the output because it can fall back to matching a row of `NA`s in `y`.

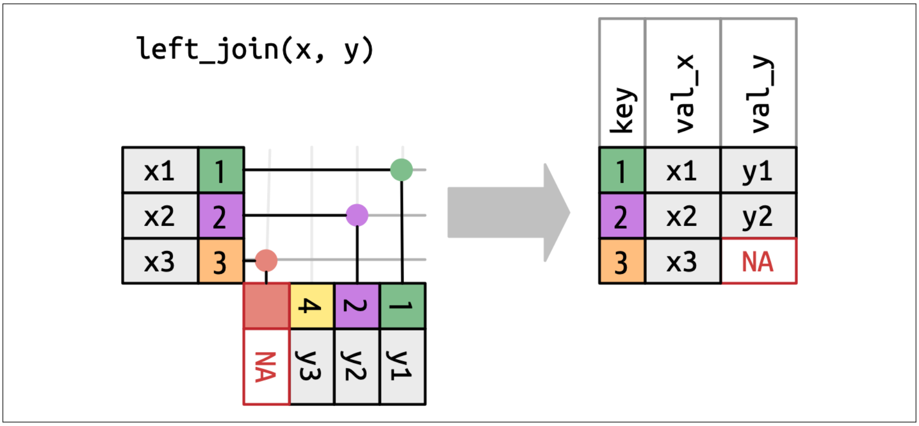


Figure 19-5. A visual representation of the left join where every row in x appears in the output.

- A *right join* keeps all observations in y, as shown in Figure 19-6. Every row of y is preserved in the output because it can fall back to matching a row of NAs in x. The output still matches x as much as possible; any extra rows from y are added to the end.

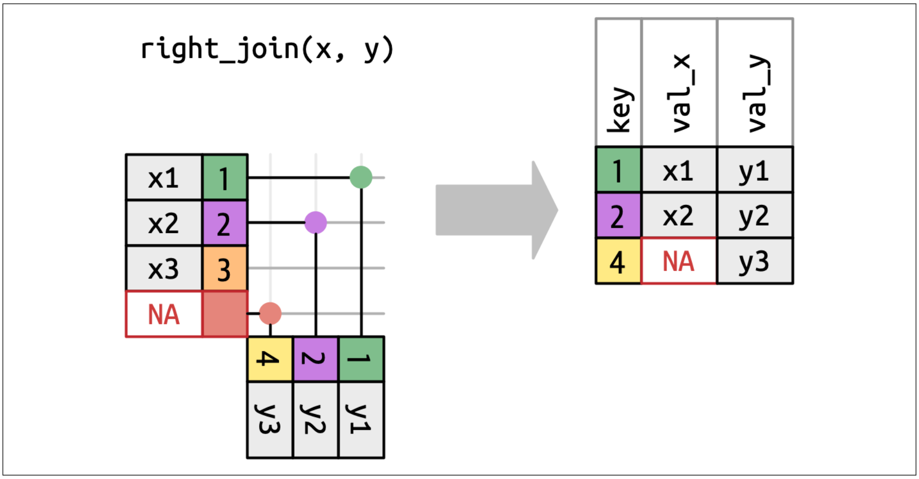


Figure 19-6. A visual representation of the right join where every row of y appears in the output.

- A *full join* keeps all observations that appear in x or y, as shown in Figure 19-7. Every row of x and y is included in the output because both x and y have a fallback row of NAs. Again, the output starts with all rows from x, followed by the remaining unmatched y rows.

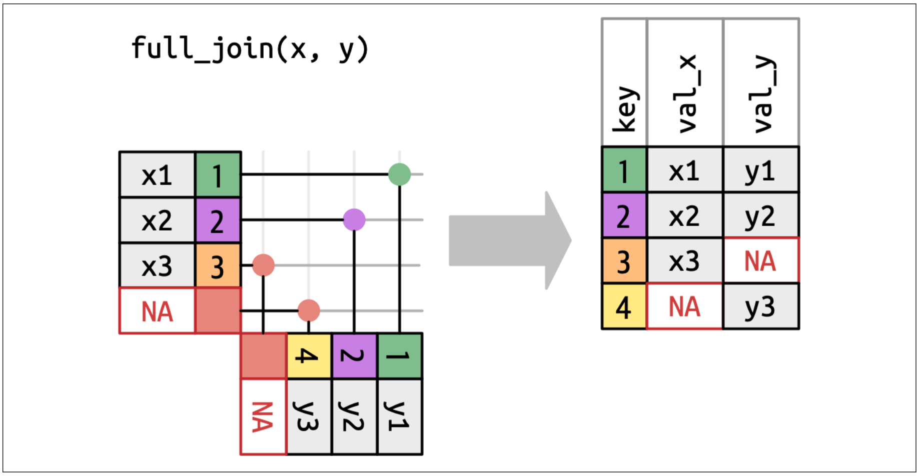


Figure 19-7. A visual representation of the full join where every row in x and y appears in the output.

Another way to show how the types of outer join differ is with a Venn diagram, as in Figure 19-8. However, this is not a great representation because while it might jog your memory about which rows are preserved, it fails to illustrate what's happening with the columns.

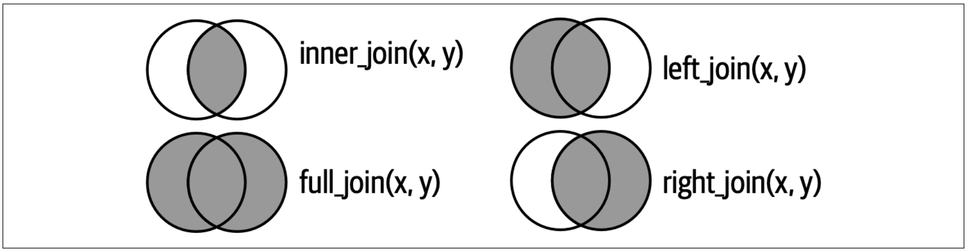


Figure 19-8. Venn diagrams showing the difference between inner, left, right, and full joins.

The joins shown here are the so-called *equi joins*, where rows match if the keys are equal. Equi joins are the most common type of join, so we'll typically omit the equi prefix and just say "inner join" rather than "equi inner join." We'll come back to non-equi joins in "Filtering Joins" on page 346.

Row Matching

So far we've explored what happens if a row in x matches zero or one rows in y. What happens if it matches more than one row? To understand what's going on, let's first narrow our focus to `inner_join()` and then draw a picture, as shown in Figure 19-9.

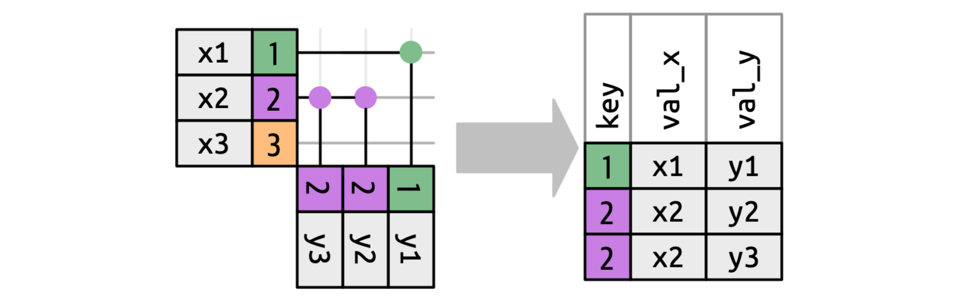


Figure 19-9. The three ways a row in x can match. x1 matches one row in y, x2 matches two rows in y, and x3 matches zero rows in y. Note that while there are three rows in x and three rows in the output, there isn't a direct correspondence between the rows.

There are three possible outcomes for a row in x:

- If it doesn't match anything, it's dropped.
- If it matches one row in y, it's preserved.
- If it matches more than one row in y, it's duplicated once for each match.

In principle, this means there's no guaranteed correspondence between the rows in the output and the rows in x, but in practice, this rarely causes problems. There is, however, one particularly dangerous case that can cause a combinatorial explosion of rows. Imagine joining the following two tables:

```
df1 <- tibble(key = c(1, 2, 2), val_x = c("x1", "x2", "x3"))
df2 <- tibble(key = c(1, 2, 2), val_y = c("y1", "y2", "y3"))
```

While the first row in `df1` matches only one row in `df2`, the second and third rows both match two rows. This is sometimes called a *many-to-many* join and will cause dplyr to emit a warning:

```
df1 |>
  inner_join(df2, join_by(key))
#> Warning in inner_join(df1, df2, join_by(key)):
#> Detected an unexpected many-to-many relationship between `x` and `y`.
#> ℹ Row 2 of `x` matches multiple rows in `y`.
#> ℹ Row 2 of `y` matches multiple rows in `x`.
#> ℹ If a many-to-many relationship is expected, set `relationship =
#>   "many-to-many"` to silence this warning.
#> # A tibble: 5 × 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
```

If you are doing this deliberately, you can set `relationship = "many-to-many"`, as the warning suggests.

Filtering Joins

The number of matches also determines the behavior of the filtering joins. The semi-join keeps rows in `x` that have one or more matches in `y`, as in Figure 19-10. The anti-join keeps rows in `x` that match zero rows in `y`, as in Figure 19-11. In both cases, only the existence of a match is important; it doesn't matter how many times it matches. This means that filtering joins never duplicate rows like mutating joins do.

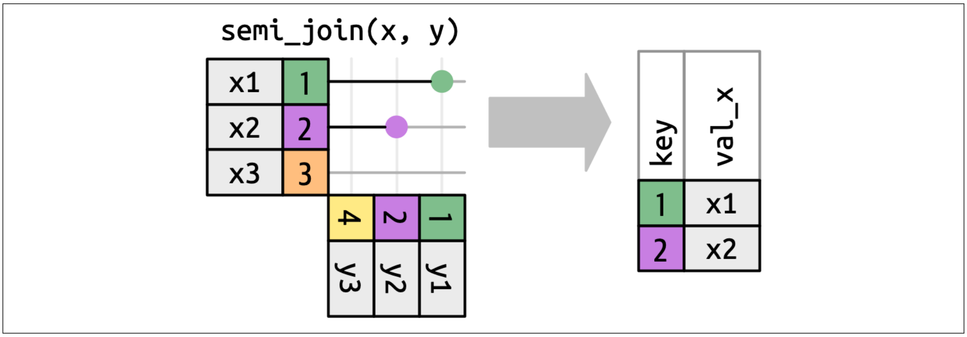


Figure 19-10. In a semi-join it only matters that there is a match; otherwise, values in y don't affect the output.

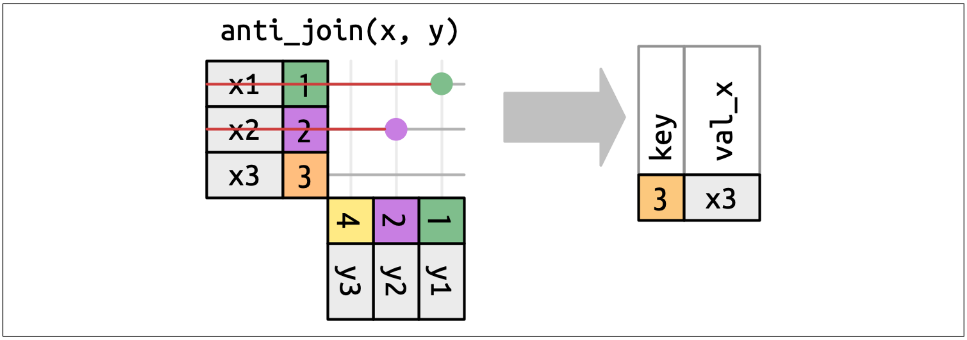


Figure 19-11. An anti-join is the inverse of a semi-join, dropping rows from x that have a match in y.

Non-Equi Joins

So far you've seen only equi joins, joins where the rows match if the x key equals the y key. Now we're going to relax that restriction and discuss other ways of determining if a pair of rows match.

But before we can do that, we need to revisit a simplification we made previously. In equi joins the x keys and y are always equal, so we need to show only one in the output. We can request that dplyr keep both keys with `keep = TRUE`, leading to the following code and the redrawn `inner_join()` in Figure 19-12.

```
x |> left_join(y, by = "key", keep = TRUE)
#> # A tibble: 3 × 4
#>   key.x val_x key.y val_y
#>   <dbl> <chr> <dbl> <chr>
#> 1     1 x1        1 y1   
#> 2     2 x2        2 y2   
#> 3     3 x3       NA <NA>
```

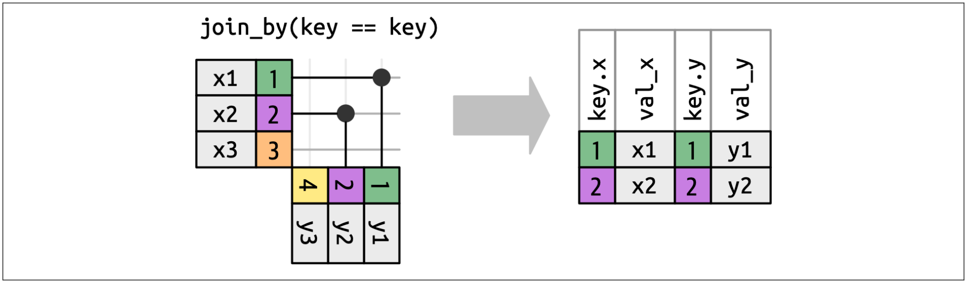


Figure 19-12. An inner join showing both x and y keys in the output.

When we move away from equi joins, we'll always show the keys, because the key values will often be different. For example, instead of matching only when the `x$key` and `y$key` are equal, we could match whenever the `x$key` is greater than or equal to the `y$key`, leading to Figure 19-13. dplyr's join functions understand this distinction between equi and non-equi joins so will always show both keys when you perform a non-equi join.

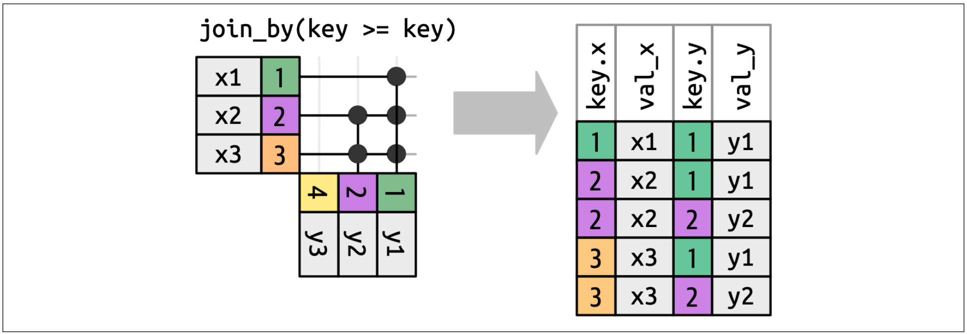


Figure 19-13. A non-equi join where the x key must be greater than or equal to the y key. Many rows generate multiple matches.

Non-equi join isn't a particularly useful term because it only tells you what the join is not, not what it is. dplyr helps by identifying four particularly useful types of non-equi join:

Cross joins
: Match every pair of rows.

Inequality joins
: Use <, <=, >, and >= instead of ==.

Rolling joins
: Similar to inequality joins but only find the closest match.

Overlap joins
: A special type of inequality join designed to work with ranges.

Each of these is described in more detail in the following sections.

Cross Joins

A cross join matches everything, as in Figure 19-14, generating the Cartesian product of rows. This means the output will have `nrow(x) * nrow(y)` rows.

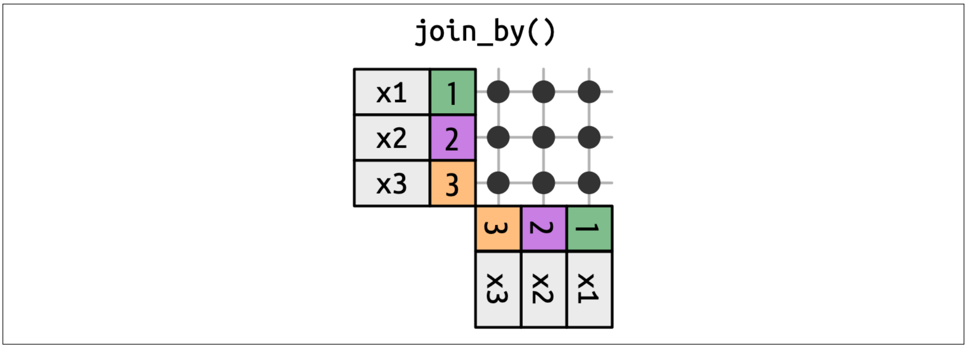


Figure 19-14. A cross join matches each row in x with every row in y.

Cross joins are useful when generating permutations. For example, the following code generates every possible pair of names. Since we're joining `df` to itself, this is sometimes called a *self-join*. Cross joins use a different join function because there's no distinction between inner/left/right/full when you're matching every row.

```
df <- tibble(name = c("John", "Simon", "Tracy", "Max"))
df |> cross_join(df)
#> # A tibble: 16 × 2
#>   name.x name.y
#>   <chr>  <chr> 
#> 1 John   John  
#> 2 John   Simon 
#> 3 John   Tracy
#> 4 John   Max
#> 5 Simon  John
#> 6 Simon  Simon
#> # … with 10 more rows
```

Inequality Joins

Inequality joins use `<`, `<=`, `>=`, or `>` to restrict the set of possible matches, as in Figure 19-13 and Figure 19-15.

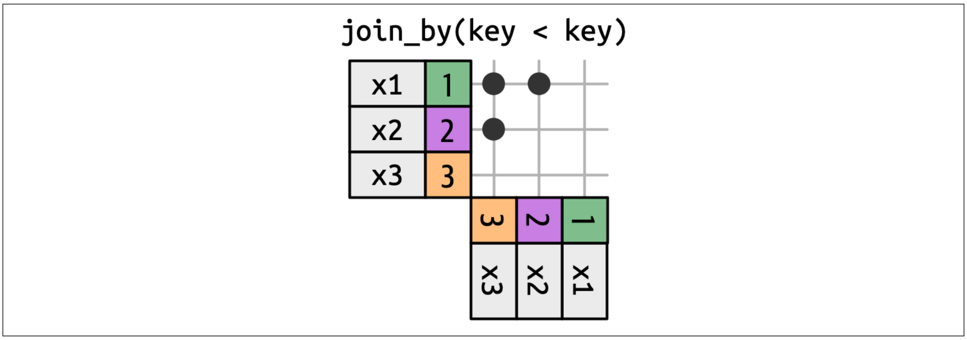


Figure 19-15. An inequality join where x is joined to y on rows where the key of x is less than the key of y. This makes a triangular shape in the top-left corner.

Inequality joins are extremely general, so general that it's hard to come up with meaningful specific use cases. One small useful technique is to use them to restrict the cross join so that instead of generating all permutations, we generate all combinations:

```
df <- tibble(id = 1:4, name = c("John", "Simon", "Tracy", "Max"))

df |> left_join(df, join_by(id < id))
#> # A tibble: 7 × 4
#>    id.x name.x  id.y name.y
#>   <int> <chr>  <int> <chr> 
#> 1     1 John       2 Simon 
#> 2     1 John       3 Tracy 
#> 3     1 John       4 Max   
#> 4     2 Simon      3 Tracy 
#> 5     2 Simon      4 Max   
#> 6     3 Tracy      4 Max   
#> # … with 1 more row
```

Rolling Joins

Rolling joins are a special type of inequality join where instead of getting *every* row that satisfies the inequality, you get just the closest row, as in Figure 19-16. You can turn any inequality join into a rolling join by adding `closest()`. For example, `join_by(closest(x <= y))` matches the smallest `y` that's greater than or equal to x, and `join_by(closest(x > y))` matches the biggest `y` that's less than x.

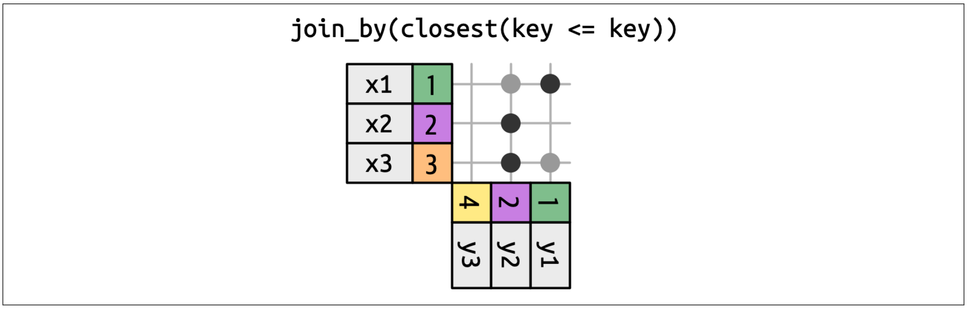


Figure 19-16. A rolling join is similar to a greater-than-or-equal inequality join but matches only the first value.

Rolling joins are particularly useful when you have two tables of dates that don't perfectly line up and you want to find, for example, the closest date in table 1 that comes before (or after) some date in table 2.

For example, imagine that you're in charge of the party planning commission for your office. Your company is rather cheap so instead of having individual parties, you have a party only once each quarter. The rules for determining when a party will be held are a little complex: parties are always on a Monday, you skip the first week of January since a lot of people are on holiday, and the first Monday of Q3 2022 is July 4, so that has to be pushed back a week. That leads to the following party days:

```
parties <- tibble(
  q = 1:4,
  party = ymd(c("2022-01-10", "2022-04-04", "2022-07-11", "2022-10-03"))
)
```

Now imagine that you have a table of employee birthdays:

```
employees <- tibble(
  name = sample(babynames::babynames$name, 100),
  birthday = ymd("2022-01-01") + (sample(365, 100, replace = TRUE) - 1)
)
employees
#> # A tibble: 100 × 2
#>   name     birthday  
#>   <chr>    <date>    
#> 1 Case     2022-09-13
#> 2 Shonnie  2022-03-30
#> 3 Burnard  2022-01-10
#> 4 Omer     2022-11-25
#> 5 Hillel   2022-07-30
#> 6 Curlie   2022-12-11
#> # … with 94 more rows
```

And for each employee we want to find the first party date that comes after (or on) their birthday. We can express that with a rolling join:

```
employees |>
  left_join(parties, join_by(closest(birthday >= party)))
```

```
#> # A tibble: 100 × 4
#>   name    birthday        q party
#>   <chr>   <date>      <int> <date>
#> 1 Case    2022-09-13      3 2022-07-11
#> 2 Shonnie 2022-03-30      1 2022-01-10
#> 3 Burnard 2022-01-10      1 2022-01-10
#> 4 Omer    2022-11-25      4 2022-10-03
#> 5 Hillel  2022-07-30      3 2022-07-11
#> 6 Curlie  2022-12-11      4 2022-10-03
#> # … with 94 more rows
```

There is, however, one problem with this approach: the folks with birthdays before January 10 don't get a party:

```
employees |>
  anti_join(parties, join_by(closest(birthday >= party)))
#> # A tibble: 0 × 2
#> # … with 2 variables: name <chr>, birthday <date>
```

To resolve that issue we'll need to tackle the problem a different way, with overlap joins.

Overlap Joins

Overlap joins provide three helpers that use inequality joins to make it easier to work with intervals:

- `between(x, y_lower, y_upper)` is short for `x >= y_lower, x <= y_upper`.
- `within(x_lower, x_upper, y_lower, y_upper)` is short for `x_lower >= y_lower, x_upper <= y_upper`.
- `overlaps(x_lower, x_upper, y_lower, y_upper)` is short for `x_lower <= y_upper, x_upper >= y_lower`.

Let's continue the birthday example to see how you might use them. There's one problem with the strategy we used earlier: there's no party preceding the birthdays from January 1 to 9. So it might be better to to be explicit about the date ranges that each party span, and make a special case for those early birthdays:

```
parties <- tibble(
  q = 1:4,
  party = ymd(c("2022-01-10", "2022-04-04", "2022-07-11", "2022-10-03")),
  start = ymd(c("2022-01-01", "2022-04-04", "2022-07-11", "2022-10-03")),
  end = ymd(c("2022-04-03", "2022-07-11", "2022-10-02", "2022-12-31"))
)
parties
#> # A tibble: 4 × 4
#>       q party      start      end
#>   <int> <date>     <date>     <date>
#> 1     1 2022-01-10 2022-01-01 2022-04-03
#> 2     2 2022-04-04 2022-04-04 2022-07-11
#> 3     3 2022-07-11 2022-07-11 2022-10-02
#> 4     4 2022-10-03 2022-10-03 2022-12-31
```

Hadley is hopelessly bad at data entry, so he also wanted to check that the party periods don't overlap. One way to do this is by using a self-join to check whether any start-end interval overlaps with another:

```
parties |>
  inner_join(parties, join_by(overlaps(start, end, start, end), q < q)) |>
  select(start.x, end.x, start.y, end.y)
#> # A tibble: 1 × 4
#>   start.x    end.x      start.y    end.y
#>   <date>     <date>     <date>     <date>
#> 1 2022-04-04 2022-07-11 2022-07-11 2022-10-02
```

Oops, there is an overlap, so let's fix that problem and continue:

```
parties <- tibble(
  q = 1:4,
  party = ymd(c("2022-01-10", "2022-04-04", "2022-07-11", "2022-10-03")),
  start = ymd(c("2022-01-01", "2022-04-04", "2022-07-11", "2022-10-03")),
  end = ymd(c("2022-04-03", "2022-07-10", "2022-10-02", "2022-12-31"))
)
```

Now we can match each employee to their party. This is a good place to use `unmatched = "error"` because we want to quickly find out if any employees didn't get assigned a party:

```
employees |>
  inner_join(parties, join_by(between(birthday, start, end)), unmatched = "error")
#> # A tibble: 100 × 6
#>   name     birthday       q party      start      end
#>   <chr>    <date>     <int> <date>     <date>     <date>
#> 1 Case     2022-09-13     3 2022-07-11 2022-07-11 2022-10-02
#> 2 Shonnie  2022-03-30     1 2022-01-10 2022-01-01 2022-04-03
#> 3 Burnard  2022-01-10     1 2022-01-10 2022-01-01 2022-04-03
#> 4 Omer     2022-11-25     4 2022-10-03 2022-10-03 2022-12-31
#> 5 Hillel   2022-07-30     3 2022-07-11 2022-07-11 2022-10-02
#> 6 Curlie   2022-12-11     4 2022-10-03 2022-10-03 2022-12-31
#> # … with 94 more rows
```

Exercises

1. Can you explain what's happening with the keys in this equi join? Why are they different?

```
x |> full_join(y, by = "key")
#> # A tibble: 4 × 3
#>     key val_x val_y
#>   <dbl> <chr> <chr>
#> 1     1 x1    y1
#> 2     2 x2    y2
#> 3     3 x3    <NA>
#> 4     4 <NA>  y3

x |> full_join(y, by = "key", keep = TRUE)
#> # A tibble: 4 × 4
#>   key.x val_x key.y val_y
#>   <dbl> <chr> <dbl> <chr>
#> 1     1 x1        1 y1
#> 2     2 x2       2 y2
#> 3     3 x3      NA <NA>
#> 4    NA <NA>     4 y3
```

2. When finding if any party period overlapped with another party period, we used `q < q` in the `join_by()`? Why? What happens if you remove this inequality?

Summary

In this chapter, you learned how to use mutating and filtering joins to combine data from a pair of data frames. Along the way you learned how to identify keys, and you learned the difference between primary and foreign keys. You also understand how joins work and how to figure out how many rows the output will have. Finally, you gained a glimpse into the power of non-equi joins and saw a few interesting use cases.

This chapter concludes the "Transform" part of the book where the focus was on the tools you could use with individual columns and tibbles. You learned about dplyr and base functions for working with logical vectors, numbers, and complete tables; stringr functions for working strings; lubridate functions for working with date-times; and forcats functions for working with factors.

In the next part of the book, you'll learn more about getting various types of data into R in a tidy form.

PART IV
Import

In this part of the book, you'll learn how to import a wider range of data into R, as well as how to get it into a form useful form for analysis. Sometimes this is just a matter of calling a function from the appropriate data import package. But in more complex cases it might require both tidying and transformation to get to the tidy rectangle that you'd prefer to work with.

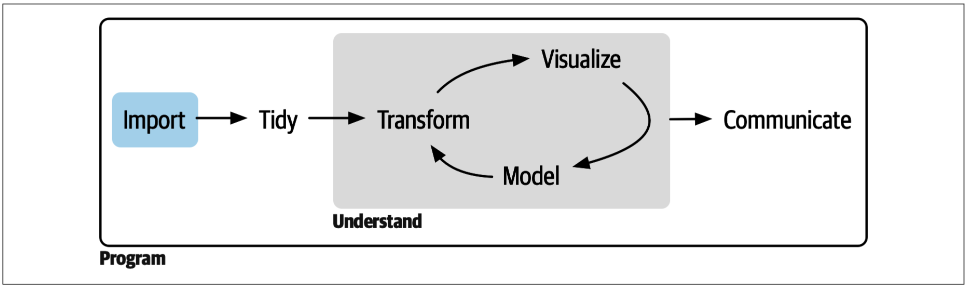


Figure IV-1. Data import is the beginning of the data science process; without data you can't do data science!

In this part of the book you'll learn how to access data stored in the following ways:

- In Chapter 20, you'll learn how to import data from Excel spreadsheets and Google Sheets.
- In Chapter 21, you'll learn about getting data out of a database and into R (and you'll also learn a little about how to get data out of R and into a database).

- In Chapter 22, you'll learn about Arrow, a powerful tool for working with out-of-memory data, particularly when it's stored in the parquet format.
- In Chapter 23, you'll learn how to work with hierarchical data, including the deeply nested lists produced by data stored in the JSON format.
- In Chapter 24, you'll learn web "scraping," the art and science of extracting data from web pages.

There are two important tidyverse packages that we don't discuss here: haven and xml2. If you are working with data from SPSS, Stata, and SAS files, check out the haven package. If you're working with XML data, check out the xml2 package. Otherwise, you'll need to do some research to figure out which package you'll need to use; Google is your friend here.

CHAPTER 20

Spreadsheets

Introduction

In Chapter 7 you learned about importing data from plain-text files like .csv and .tsv. Now it's time to learn how to get data out of a spreadsheet, either an Excel spreadsheet or a Google Sheet. This will build on much of what you've learned in Chapter 7, but we will also discuss additional considerations and complexities when working with data from spreadsheets.

If you or your collaborators are using spreadsheets for organizing data, we strongly recommend reading the paper "Data Organization in Spreadsheets" by Karl Broman and Kara Woo. The best practices presented in this paper will save you much headache when you import data from a spreadsheet into R to analyze and visualize.

Excel

Microsoft Excel is a widely used spreadsheet software program where data are organized in worksheets inside of spreadsheet files.

Prerequisites

In this section, you'll learn how to load data from Excel spreadsheets in R with the readxl package. This package is noncore tidyverse, so you need to load it explicitly, but it is installed automatically when you install the tidyverse package. Later, we'll also use the writexl package, which allows us to create Excel spreadsheets.

```
library(readxl)
library(tidyverse)
library(writexl)
```

Getting Started

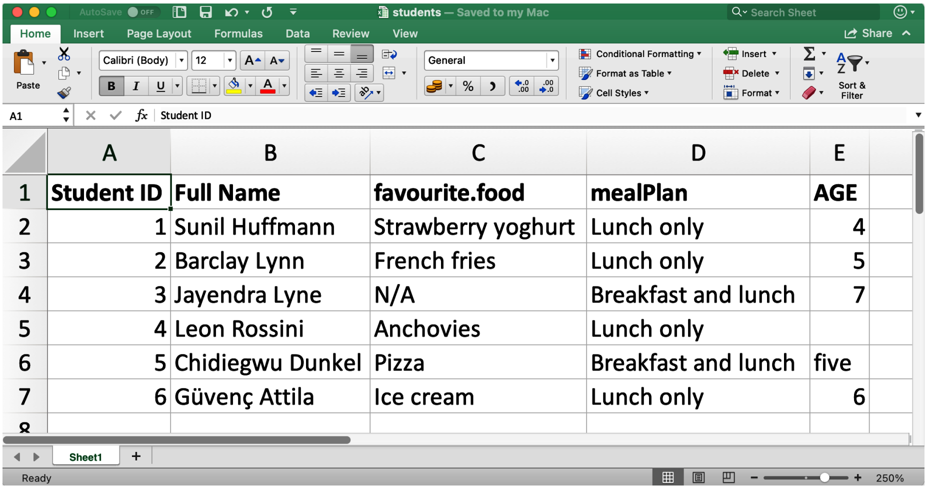
Most of readxl's functions allow you to load Excel spreadsheets into R:

- `read_xls()` reads Excel files with the `XLS` format.
- `read_xlsx()` reads Excel files with the `XLSX` format.
- `read_excel()` can read files with both the `XLS` and `XLSX` formats. It guesses the file type based on the input.

These functions all have similar syntax just like other functions we have previously introduced for reading other types of files, e.g., `read_csv()`, `read_table()`, etc. For the rest of the chapter we will focus on using `read_excel()`.

Reading Excel Spreadsheets

Figure 20-1 shows what the spreadsheet we're going to read into R looks like in Excel.



A1 fx Student ID

	A	B	C	D	E
1	Student ID	Full Name	favourite.food	mealPlan	AGE
2	1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4
3	2	Barclay Lynn	French fries	Lunch only	5
4	3	Jayendra Lyne	N/A	Breakfast and lunch	7
5	4	Leon Rossini	Anchovies	Lunch only	
6	5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five
7	6	Güvenç Attila	Ice cream	Lunch only	6
8					

Figure 20-1. Spreadsheet called `students.xlsx` in Excel.

The first argument to `read_excel()` is the path to the file to read.

```
students <- read_excel("data/students.xlsx")
```

`read_excel()` will read the file in as a tibble.

```
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE  
#>          <dbl> <chr>            <chr>              <chr>               <chr>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          4    
#> 2            2 Barclay Lynn     French fries       Lunch only          5    
#> 3           3 Jayendra Lyne    N/A                Breakfast and lunch 7
#> 4           4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5           5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6           6 Güvenç Attila    Ice cream          Lunch only          6
```

We have six students in the data and five variables on each student. However, there are a few things we might want to address in this dataset:

1. The column names are all over the place. You can provide column names that follow a consistent format; we recommend `snake_case` using the `col_names` argument.

   ```
   read_excel(
     "data/students.xlsx",
     col_names = c(
       "student_id", "full_name", "favourite_food", "meal_plan", "age")
   )
   #> # A tibble: 7 × 5
   #>   student_id full_name        favourite_food     meal_plan           age
   #>   <chr>      <chr>            <chr>              <chr>               <chr>
   #> 1 Student ID Full Name        favourite.food     mealPlan            AGE
   #> 2 1          Sunil Huffmann   Strawberry yoghurt Lunch only          4
   #> 3 2          Barclay Lynn     French fries       Lunch only          5
   #> 4 3          Jayendra Lyne    N/A                Breakfast and lunch 7
   #> 5 4          Leon Rossini     Anchovies          Lunch only          <NA>
   #> 6 5          Chidiegwu Dunkel Pizza              Breakfast and lunch five
   #> 7 6          Güvenç Attila    Ice cream          Lunch only          6
   ```

   Unfortunately, this didn't quite do the trick. We now have the variable names we want, but what was previously the header row now shows up as the first observation in the data. You can explicitly skip that row using the `skip` argument.

   ```
   read_excel(
     "data/students.xlsx",
     col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
     skip = 1
   )
   #> # A tibble: 6 × 5
   #>   student_id full_name        favourite_food     meal_plan           age
   #>        <dbl> <chr>            <chr>              <chr>               <chr>
   #> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
   #> 2          2 Barclay Lynn     French fries       Lunch only          5
   #> 3          3 Jayendra Lyne    N/A                Breakfast and lunch 7
   #> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
   #> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
   #> 6          6 Güvenç Attila    Ice cream          Lunch only          6
   ```

2. In the `favourite_food` column, one of the observations is `N/A`, which stands for "not available," but it's currently not recognized as an `NA` (note the contrast between this `N/A` and the age of the fourth student in the list). You can specify which character strings should be recognized as `NA`s with the `na` argument. By default, only `""` (empty string, or, in the case of reading from a spreadsheet, an empty cell or a cell with the formula `=NA()`) is recognized as an `NA`.

   ```
   read_excel(
     "data/students.xlsx",
     col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
   ```

```
  skip = 1,
  na = c("", "N/A")
)
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

3. One other remaining issue is that `age` is read in as a character variable, but it really should be numeric. Just like with `read_csv()` and friends for reading data from flat files, you can supply a `col_types` argument to `read_excel()` and specify the column types for the variables you read in. The syntax is a bit different, though. Your options are `"skip"`, `"guess"`, `"logical"`, `"numeric"`, `"date"`, `"text"`, or `"list"`.

```
read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A"),
  col_types = c("numeric", "text", "text", "text", "numeric")
)
#> Warning: Expecting numeric in E6 / R6C5: got 'five'
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch    NA
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

However, this didn't quite produce the desired result either. By specifying that `age` should be numeric, we have turned the one cell with the non-numeric entry (which had the value `five`) into an `NA`. In this case, we should read age in as `"text"` and then make the change once the data is loaded in R.

```
students <- read_excel(
  "data/students.xlsx",
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A"),
  col_types = c("numeric", "text", "text", "text", "text")
)

students <- students |>
  mutate(
    age = if_else(age == "five", "5", age),
    age = parse_number(age)
  )

students
```

```
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan             age
#>        <dbl> <chr>            <chr>              <chr>               <dbl>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only              4
#> 2          2 Barclay Lynn     French fries       Lunch only              5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch     7
#> 4          4 Leon Rossini     Anchovies          Lunch only             NA
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch     5
#> 6          6 Güvenç Attila    Ice cream          Lunch only              6
```

It took us multiple steps and trial and error to load the data in exactly the format we want, and this is not unexpected. Data science is an iterative process, and the process of iteration can be even more tedious when reading data in from spreadsheets compared to other plain-text, rectangular data files because humans tend to input data into spreadsheets and use them not just for data storage but also for sharing and communication.

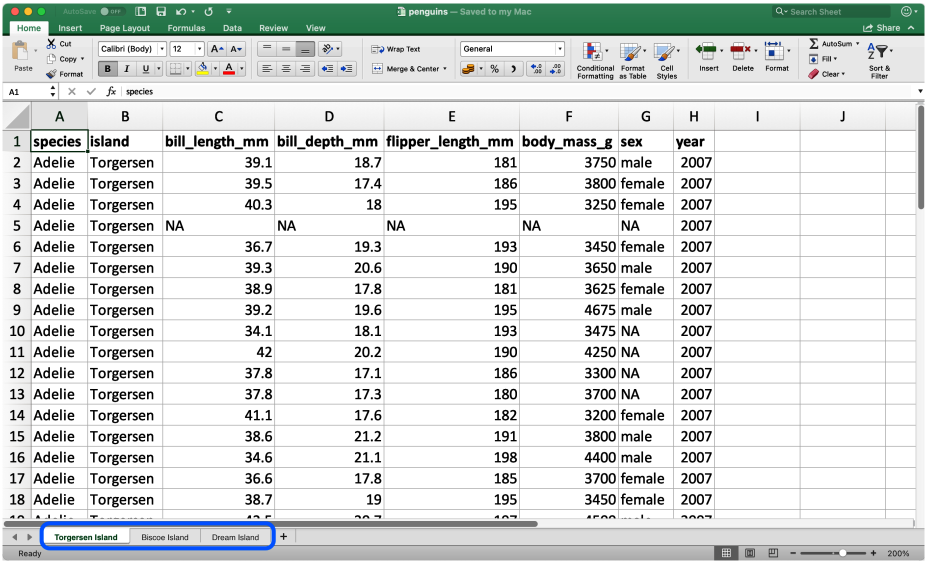
There is no way to know exactly what the data will look like until you load it and take a look at it. Well, there is one way, actually. You can open the file in Excel and take a peek. If you're going to do so, we recommend making a copy of the Excel file to open and browse interactively while leaving the original data file untouched and reading into R from the untouched file. This will ensure you don't accidentally overwrite anything in the spreadsheet while inspecting it. You should also not be afraid of doing what we did here: load the data, take a peek, make adjustments to your code, load it again, and repeat until you're happy with the result.

Reading Worksheets

An important feature that distinguishes spreadsheets from flat files is the notion of multiple sheets, called *worksheets*. Figure 20-2 shows an Excel spreadsheet with multiple worksheets. The data come from the palmerpenguins package. Each worksheet contains information on penguins from a different island where data were collected.

You can read a single worksheet from a spreadsheet with the `sheet` argument in `read_excel()`. The default, which we've been relying on up until now, is the first sheet.

```
read_excel("data/penguins.xlsx", sheet = "Torgersen Island")
#> # A tibble: 52 × 8
#>   species island    bill_length_mm     bill_depth_mm      flipper_length_mm
#>   <chr>   <chr>     <chr>              <chr>              <chr>
#> 1 Adelie  Torgersen 39.1               18.7               181
#> 2 Adelie  Torgersen 39.5               17.399999999999999 186
#> 3 Adelie  Torgersen 40.299999999999997 18                 195
#> 4 Adelie  Torgersen NA                 NA                 NA
#> 5 Adelie  Torgersen 36.700000000000003 19.3               193
#> 6 Adelie  Torgersen 39.299999999999997 20.6               190
#> # … with 46 more rows, and 3 more variables: body_mass_g <chr>, sex <chr>,
#> #   year <dbl>
```



	A	B	C	D	E	F	G	H	I	J
1	species	island	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g	sex	year		
2	Adelie	Torgersen	39.1	18.7	181	3750	male	2007		
3	Adelie	Torgersen	39.5	17.4	186	3800	female	2007		
4	Adelie	Torgersen	40.3	18	195	3250	female	2007		
5	Adelie	Torgersen	NA	NA	NA	NA	NA	2007		
6	Adelie	Torgersen	36.7	19.3	193	3450	female	2007		
7	Adelie	Torgersen	39.3	20.6	190	3650	male	2007		
8	Adelie	Torgersen	38.9	17.8	181	3625	female	2007		
9	Adelie	Torgersen	39.2	19.6	195	4675	male	2007		
10	Adelie	Torgersen	34.1	18.1	193	3475	NA	2007		
11	Adelie	Torgersen	42	20.2	190	4250	NA	2007		
12	Adelie	Torgersen	37.8	17.1	186	3300	NA	2007		
13	Adelie	Torgersen	37.8	17.3	180	3700	NA	2007		
14	Adelie	Torgersen	41.1	17.6	182	3200	female	2007		
15	Adelie	Torgersen	38.6	21.2	191	3800	male	2007		
16	Adelie	Torgersen	34.6	21.1	198	4400	male	2007		
17	Adelie	Torgersen	36.6	17.8	185	3700	female	2007		
18	Adelie	Torgersen	38.7	19	195	3450	female	2007		

Figure 20-2. Spreadsheet called `penguins.xlsx` in Excel containing three worksheets.

Some variables that appear to contain numerical data are read in as characters due to the character string `"NA"` not being recognized as a true `NA`.

```
penguins_torgersen <- read_excel(
  "data/penguins.xlsx", sheet = "Torgersen Island", na = "NA"
)

penguins_torgersen
#> # A tibble: 52 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
#>   <chr>   <chr>              <dbl>         <dbl>             <dbl>
#> 1 Adelie  Torgersen           39.1          18.7               181
#> 2 Adelie  Torgersen           39.5          17.4               186
#> 3 Adelie  Torgersen           40.3          18                 195
#> 4 Adelie  Torgersen           NA            NA                  NA
#> 5 Adelie  Torgersen           36.7          19.3               193
#> 6 Adelie  Torgersen           39.3          20.6               190
#> # … with 46 more rows, and 3 more variables: body_mass_g <dbl>, sex <chr>,
#> #   year <dbl>
```

Alternatively, you can use `excel_sheets()` to get information on all worksheets in an Excel spreadsheet and then read the one(s) you're interested in.

```
excel_sheets("data/penguins.xlsx")
#> [1] "Torgersen Island" "Biscoe Island"    "Dream Island"
```

Once you know the names of the worksheets, you can read them in individually with `read_excel()`.

```
penguins_biscoe <- read_excel("data/penguins.xlsx", sheet = "Biscoe Island", na = "NA")
penguins_dream  <- read_excel("data/penguins.xlsx", sheet = "Dream Island", na = "NA")
```

In this case, the full penguins dataset is spread across three worksheets in the spreadsheet. Each worksheet has the same number of columns but different numbers of rows.

```
dim(penguins_torgersen)
#> [1] 52  8
dim(penguins_biscoe)
#> [1] 168   8
dim(penguins_dream)
#> [1] 124   8
```

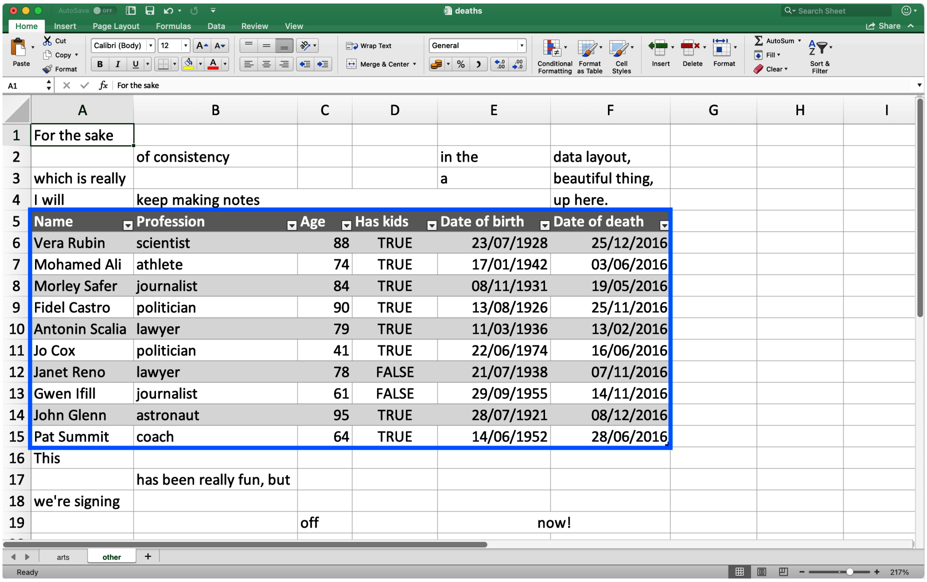
We can put them together with `bind_rows()`:

```
penguins <- bind_rows(penguins_torgersen, penguins_biscoe, penguins_dream)
penguins
#> # A tibble: 344 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
#>   <chr>   <chr>              <dbl>         <dbl>             <dbl>
#> 1 Adelie  Torgersen           39.1          18.7               181
#> 2 Adelie  Torgersen           39.5          17.4               186
#> 3 Adelie  Torgersen           40.3          18                 195
#> 4 Adelie  Torgersen           NA            NA                  NA
#> 5 Adelie  Torgersen           36.7          19.3               193
#> 6 Adelie  Torgersen           39.3          20.6               190
#> # … with 338 more rows, and 3 more variables: body_mass_g <dbl>, sex <chr>,
#> #   year <dbl>
```

In Chapter 26 we'll talk about ways of doing this sort of task without repetitive code.

Reading Part of a Sheet

Since many use Excel spreadsheets for presentation as well as for data storage, it's quite common to find cell entries in a spreadsheet that are not part of the data you want to read into R. Figure 20-3 shows such a spreadsheet: in the middle of the sheet is what looks like a data frame, but there is extraneous text in cells above and below the data.



	A	B	C	D	E	F	G	H	I
1	For the sake								
2		of consistency			in the	data layout,			
3	which is really				a	beautiful thing,			
4	I will	keep making notes				up here.			
5	Name	Profession	Age	Has kids	Date of birth	Date of death			
6	Vera Rubin	scientist	88	TRUE	23/07/1928	25/12/2016			
7	Mohamed Ali	athlete	74	TRUE	17/01/1942	03/06/2016			
8	Morley Safer	journalist	84	TRUE	08/11/1931	19/05/2016			
9	Fidel Castro	politician	90	TRUE	13/08/1926	25/11/2016			
10	Antonin Scalia	lawyer	79	TRUE	11/03/1936	13/02/2016			
11	Jo Cox	politician	41	TRUE	22/06/1974	16/06/2016			
12	Janet Reno	lawyer	78	FALSE	21/07/1938	07/11/2016			
13	Gwen Ifill	journalist	61	FALSE	29/09/1955	14/11/2016			
14	John Glenn	astronaut	95	TRUE	28/07/1921	08/12/2016			
15	Pat Summit	coach	64	TRUE	14/06/1952	28/06/2016			
16	This								
17		has been really fun, but							
18	we're signing								
19			off		now!				

Figure 20-3. Spreadsheet called `deaths.xlsx` in Excel.

This spreadsheet is one of the example spreadsheets provided in the readxl package. You can use the `readxl_example()` function to locate the spreadsheet on your system in the directory where the package is installed. This function returns the path to the spreadsheet, which you can use in `read_excel()` as usual.

```
deaths_path <- readxl_example("deaths.xlsx")
deaths <- read_excel(deaths_path)
#> New names:
#> • `` -> `...2`
#> • `` -> `...3`
#> • `` -> `...4`
#> • `` -> `...5`
#> • `` -> `...6`
deaths
#> # A tibble: 18 × 6
#>   `Lots of people`    ...2       ...3  ...4     ...5          ...6
#>   <chr>               <chr>      <chr> <chr>    <chr>         <chr>
#> 1 simply cannot resi… <NA>       <NA>  <NA>     <NA>          some notes
#> 2 at                  the        top   <NA>     of            their spreadsh…
#> 3 or                  merging    <NA>  <NA>     <NA>          cells
#> 4 Name                Profession Age   Has kids Date of birth Date of death
#> 5 David Bowie         musician   69    TRUE     17175         42379
#> 6 Carrie Fisher       actor      60    TRUE     20749         42731
#> # … with 12 more rows
```

The top three rows and the bottom four rows are not part of the data frame. It's possible to eliminate these extraneous rows using the `skip` and `n_max` arguments, but we recommend using cell ranges. In Excel, the top-left cell is `A1`. As you move across columns to the right, the cell label moves down the alphabet, i.e., B1, C1, etc. And as you move down a column, the number in the cell label increases, i.e., A2, A3, etc.

Here the data we want to read in starts in cell A5 and ends in cell F15. In spreadsheet notation, this is A5:F15, which we supply to the range argument:

```
read_excel(deaths_path, range = "A5:F15")
#> # A tibble: 10 × 6
#>   Name          Profession   Age `Has kids` `Date of birth`
#>   <chr>         <chr>      <dbl> <lgl>      <dttm>
#> 1 David Bowie   musician      69 TRUE       1947-01-08 00:00:00
#> 2 Carrie Fisher actor         60 TRUE       1956-10-21 00:00:00
#> 3 Chuck Berry   musician      90 TRUE       1926-10-18 00:00:00
#> 4 Bill Paxton   actor         61 TRUE       1955-05-17 00:00:00
#> 5 Prince        musician      57 TRUE       1958-06-07 00:00:00
#> 6 Alan Rickman  actor         69 FALSE      1946-02-21 00:00:00
#> # … with 4 more rows, and 1 more variable: `Date of death` <dttm>
```

Data Types

In CSV files, all values are strings. This is not particularly true to the data, but it is simple: everything is a string.

The underlying data in Excel spreadsheets is more complex. A cell can be one of four things:

- A Boolean, like TRUE, FALSE, or NA
- A number, like "10" or "10.5"
- A datetime, which can also include time like "11/1/21" or "11/1/21 3:00 PM"
- A text string, like "ten"

When working with spreadsheet data, it's important to keep in mind that the underlying data can be very different than what you see in the cell. For example, Excel has no notion of an integer. All numbers are stored as floating points, but you can choose to display the data with a customizable number of decimal points. Similarly, dates are actually stored as numbers, specifically the number of seconds since January 1, 1970. You can customize how you display the date by applying formatting in Excel. Confusingly, it's also possible to have something that looks like a number but is actually a string (e.g., type '10 into a cell in Excel).

These differences between how the underlying data are stored versus how they're displayed can cause surprises when the data are loaded into R. By default readxl will guess the data type in a given column. A recommended workflow is to let readxl guess the column types, confirm that you're happy with the guessed column types, and if not, go back and re-import specifying col_types, as shown in "Reading Excel Spreadsheets" on page 358.

Another challenge is when you have a column in your Excel spreadsheet that has a mix of these types, e.g., some cells are numeric, others text, others dates. When importing the data into R, readxl has to make some decisions. In these cases you can set the type for this column to "list", which will load the column as a list of length 1 vectors, where the type of each element of the vector is guessed.

> Sometimes data is stored in more exotic ways, like the color of the cell background or whether the text is bold. In such cases, you might find the tidyxl package useful. See *https://oreil.ly/jNskS* for more on strategies for working with nontabular data from Excel.

Writing to Excel

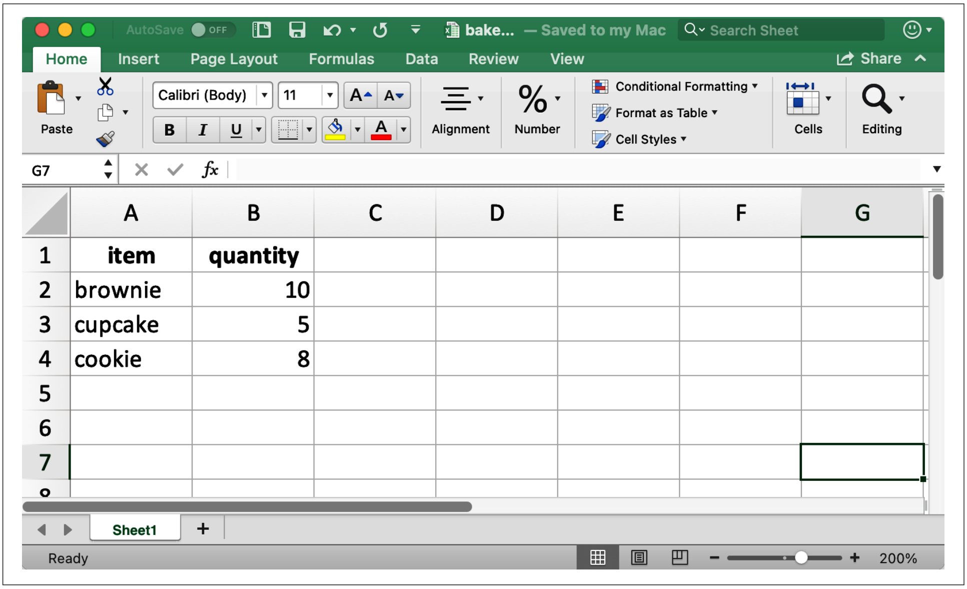
Let's create a small data frame that we can then write out. Note that item is a factor and quantity is an integer.

```
bake_sale <- tibble(
  item     = factor(c("brownie", "cupcake", "cookie")),
  quantity = c(10, 5, 8)
)

bake_sale
#> # A tibble: 3 × 2
#>   item    quantity
#>   <fct>      <dbl>
#> 1 brownie       10
#> 2 cupcake        5
#> 3 cookie         8
```

You can write data back to disk as an Excel file using write_xlsx() from the writexl package:

```
write_xlsx(bake_sale, path = "data/bake-sale.xlsx")
```

Figure 20-4 shows what the data looks like in Excel. Note that column names are included and bold. These names can be turned off by setting the col_names and format_headers arguments to FALSE.



	A	B	C	D	E	F	G
1	item	quantity					
2	brownie	10					
3	cupcake	5					
4	cookie	8					
5							
6							
7							

Figure 20-4. Spreadsheet called `bake_sale.xlsx` in Excel.

Just like reading from a CSV, information on data type is lost when we read the data back in. This makes Excel files unreliable for caching interim results as well. For alternatives, see "Writing to a File" on page 108.

```
read_excel("data/bake-sale.xlsx")
#> # A tibble: 3 × 2
#>   item    quantity
#>   <chr>      <dbl>
#> 1 brownie       10
#> 2 cupcake        5
#> 3 cookie         8
```

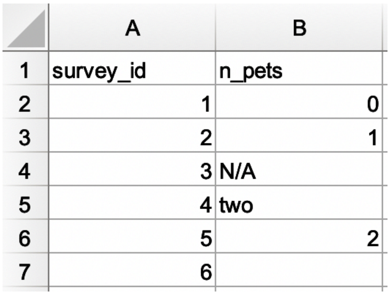
Formatted Output

The writexl package is a lightweight solution for writing a simple Excel spreadsheet, but if you're interested in additional features such as writing to sheets within a spreadsheet and styling, you will want to use the openxlsx package. We won't go into the details of using this package here, but we recommend reading *https://oreil.ly/clwtE* for an extensive discussion on further formatting functionality for data written from R to Excel with openxlsx.

Note that this package is not part of the tidyverse, so the functions and workflows may feel unfamiliar. For example, function names are camelCase, multiple functions can't be composed in pipelines, and arguments are in a different order than they tend to be in the tidyverse. However, this is OK. As your R learning and usage expands outside of this book, you will encounter lots of different styles used in various R packages that you might use to accomplish specific goals in R. A good way of familiarizing yourself with the coding style used in a new package is to run the examples provided in the function documentation to get a feel for the syntax and the output formats as well as reading any vignettes that might come with the package.

Exercises

1. In an Excel file, create the following dataset and save it as `survey.xlsx`. Alternatively, you can download it as an Excel file.

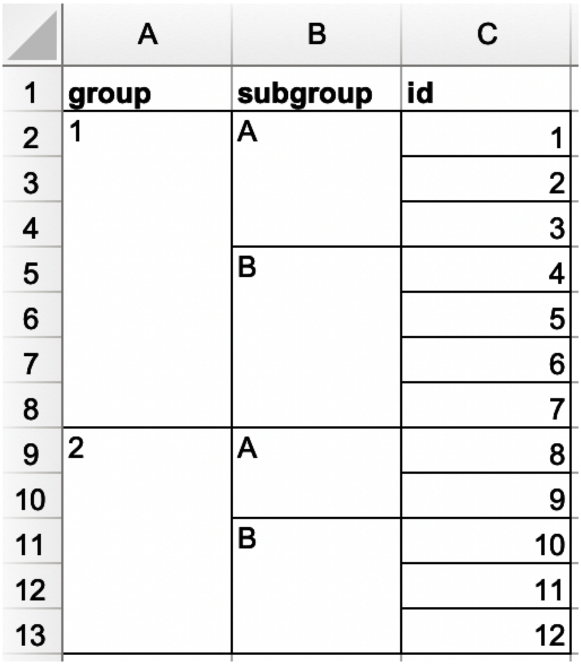
   

	A	B
1	survey_id	n_pets
2	1	0
3	2	1
4	3	N/A
5	4	two
6	5	2
7	6	

   Then, read it into R, with `survey_id` as a character variable and `n_pets` as a numerical variable.

   ```
   #> # A tibble: 6 × 2
   #>   survey_id n_pets
   #>     <chr>  <dbl>
   #>   1 1        0
   #>   2 2        1
   #>   3 3       NA
   #>   4 4        2
   #>   5 5        2
   #>   6 6       NA
   ```

2. In another Excel file, create the following dataset and save it as `roster.xlsx`. Alternatively, you can download it as an Excel file.

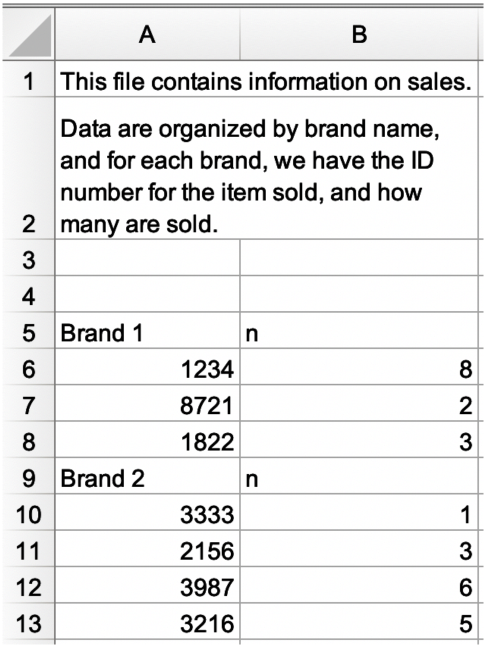


	A	B	C
1	group	subgroup	id
2	1	A	1
3			2
4			3
5		B	4
6			5
7			6
8			7
9	2	A	8
10			9
11		B	10
12			11
13			12

Then, read it into R. The resulting data frame should be called `roster` and should look like the following:

```
#> # A tibble: 12 × 3
#>    group subgroup    id
#>    <dbl> <chr>    <dbl>
#>  1     1 A            1
#>  2     1 A            2
#>  3     1 A            3
#>  4     1 B            4
#>  5     1 B            5
#>  6     1 B            6
#>  7     1 B            7
#>  8     2 A            8
#>  9     2 A            9
#> 10     2 B           10
#> 11     2 B           11
#> 12     2 B           12
```

3. In a new Excel file, create the following dataset and save it as `sales.xlsx`. Alternatively, you can download it as an Excel file.



	A	B
1	This file contains information on sales.	
2	Data are organized by brand name, and for each brand, we have the ID number for the item sold, and how many are sold.	
3		
4		
5	Brand 1	n
6	1234	8
7	8721	2
8	1822	3
9	Brand 2	n
10	3333	1
11	2156	3
12	3987	6
13	3216	5

a. Read `sales.xlsx` in and save as `sales`. The data frame should look like the following, with `id` and `n` as column names and nine rows:

```
#> # A tibble: 9 × 2
#>   id      n    
#>   <chr>   <chr>
#> 1 Brand 1 n    
#> 2 1234    8    
#> 3 8721    2    
#> 4 1822    3    
#> 5 Brand 2 n    
#> 6 3333    1    
#> 7 2156    3    
#> 8 3987    6    
#> 9 3216    5
```

b. Modify `sales` further to get it into the following tidy format with three columns (`brand`, `id`, and `n`) and seven rows of data. Note that `id` and `n` are numeric, and `brand` is a character variable.

```
#> # A tibble: 7 × 3
#>   brand      id     n
#>   <chr>   <dbl> <dbl>
#> 1 Brand 1  1234     8
#> 2 Brand 1  8721     2
#> 3 Brand 1  1822     3
#> 4 Brand 2  3333     1
#> 5 Brand 2  2156     3
#> 6 Brand 2  3987     6
#> 7 Brand 2  3216     5
```

4. Re-create the bake_sale data frame, and write it out to an Excel file using the write.xlsx() function from the openxlsx package.
5. In Chapter 7 you learned about the janitor::clean_names() function to turn column names into snake case. Read the students.xlsx file that we introduced earlier in this section and use this function to "clean" the column names.
6. What happens if you try to read in a file with an .xlsx extension with read_xls()?

Google Sheets

Google Sheets is another widely used spreadsheet program. It's free and web-based. Just like with Excel, in Google Sheets data are organized in worksheets (also called *sheets*) inside of spreadsheet files.

Prerequisites

This section will also focus on spreadsheets, but this time you'll be loading data from a Google Sheet with the googlesheets4 package. This package is noncore tidyverse as well, so you need to load it explicitly:

```
library(googlesheets4)
library(tidyverse)
```

A quick note about the name of the package: googlesheets4 uses v4 of the Sheets API v4 to provide an R interface to Google Sheets.

Getting Started

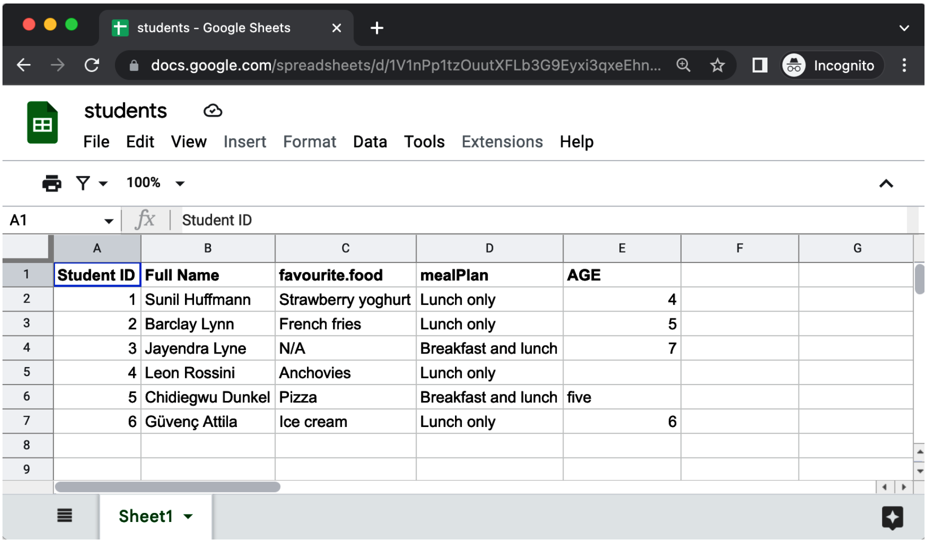
The main function of the googlesheets4 package is read_sheet(), which reads a Google Sheet from a URL or a file ID. This function also goes by the name range_read().

You can also create a new sheet with gs4_create() or write to an existing sheet with sheet_write() and friends.

In this section we'll work with the same datasets as the ones in the Excel section to highlight similarities and differences between workflows for reading data from Excel and Google Sheets. The readxl and googlesheets4 packages are both designed to mimic the functionality of the readr package, which provides the read_csv() function you saw in Chapter 7. Therefore, many of the tasks can be accomplished with simply swapping out read_excel() for read_sheet(). However you'll also see that Excel and Google Sheets don't behave in the same way; therefore, other tasks may require further updates to the function calls.

Reading Google Sheets

Figure 20-5 shows what the spreadsheet we're going to read into R looks like in Google Sheets. This is the same dataset as in Figure 20-1, except it's stored in a Google Sheet instead of Excel.



	A	B	C	D	E	F	G
1	Student ID	Full Name	favourite.food	mealPlan	AGE		
2	1	Sunil Huffmann	Strawberry yoghurt	Lunch only	4		
3	2	Barclay Lynn	French fries	Lunch only	5		
4	3	Jayendra Lyne	N/A	Breakfast and lunch	7		
5	4	Leon Rossini	Anchovies	Lunch only			
6	5	Chidiegwu Dunkel	Pizza	Breakfast and lunch	five		
7	6	Güvenç Attila	Ice cream	Lunch only	6		
8							
9							

Figure 20-5. Google Sheet called students in a browser window.

The first argument to `read_sheet()` is the URL of the file to read, and it returns a tibble.

These URLs are not pleasant to work with, so you'll often want to identify a sheet by its ID.

```
students_sheet_id <- "1V1nPp1tzOuutXFLb3G9Eyxi3qxeEhnOXUzL5_BcCQ0w"
students <- read_sheet(students_sheet_id)
#> ✔ Reading from students.
#> ✔ Range Sheet1.
students
#> # A tibble: 6 × 5
#>   `Student ID` `Full Name`      favourite.food     mealPlan            AGE
#>          <dbl> <chr>            <chr>              <chr>               <list>
#> 1            1 Sunil Huffmann   Strawberry yoghurt Lunch only          <dbl>
#> 2            2 Barclay Lynn     French fries       Lunch only          <dbl>
#> 3            3 Jayendra Lyne    N/A                Breakfast and lunch <dbl>
#> 4            4 Leon Rossini     Anchovies          Lunch only          <NULL>
#> 5            5 Chidiegwu Dunkel Pizza              Breakfast and lunch <chr>
#> 6            6 Güvenç Attila    Ice cream          Lunch only          <dbl>
```

Just like we did with `read_excel()`, we can supply column names, NA strings, and column types to `read_sheet()`.

```
students <- read_sheet(
  students_sheet_id,
  col_names = c("student_id", "full_name", "favourite_food", "meal_plan", "age"),
  skip = 1,
  na = c("", "N/A"),
  col_types = "dcccc"
)
#> ✓ Reading from students.
#> ✓ Range 2:10000000.

students
#> # A tibble: 6 × 5
#>   student_id full_name        favourite_food     meal_plan           age
#>        <dbl> <chr>            <chr>              <chr>               <chr>
#> 1          1 Sunil Huffmann   Strawberry yoghurt Lunch only          4
#> 2          2 Barclay Lynn     French fries       Lunch only          5
#> 3          3 Jayendra Lyne    <NA>               Breakfast and lunch 7
#> 4          4 Leon Rossini     Anchovies          Lunch only          <NA>
#> 5          5 Chidiegwu Dunkel Pizza              Breakfast and lunch five
#> 6          6 Güvenç Attila    Ice cream          Lunch only          6
```

Note that we defined column types a bit differently here, using short codes. For example, "dcccc" stands for "double, character, character, character, character."

It's also possible to read individual sheets from Google Sheets. Let's read the "Torgersen Island" sheet from the penguins Google Sheet:

```
penguins_sheet_id <- "1aFu8lnD_g0yjF5O-K6SFgSEWiHPpgvFCF0NY9D6LXnY"
read_sheet(penguins_sheet_id, sheet = "Torgersen Island")
#> ✓ Reading from penguins.
#> ✓ Range ''Torgersen Island''.
#> # A tibble: 52 × 8
#>   species island    bill_length_mm bill_depth_mm flipper_length_mm
#>   <chr>   <chr>     <list>         <list>        <list>
#> 1 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 2 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 3 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 4 Adelie  Torgersen <chr [1]>      <chr [1]>     <chr [1]>
#> 5 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> 6 Adelie  Torgersen <dbl [1]>      <dbl [1]>     <dbl [1]>
#> # … with 46 more rows, and 3 more variables: body_mass_g <list>, sex <chr>,
#> #   year <dbl>
```

You can obtain a list of all sheets within a Google Sheet with `sheet_names()`:

```
sheet_names(penguins_sheet_id)
#> [1] "Torgersen Island" "Biscoe Island"    "Dream Island"
```

Finally, just like with `read_excel()`, we can read in a portion of a Google Sheet by defining a `range` in `read_sheet()`. Note that we're also using the `gs4_example()` function to locate an example Google Sheet that comes with the following googlesheets4 package:

```
deaths_url <- gs4_example("deaths")
deaths <- read_sheet(deaths_url, range = "A5:F15")
#> ✓ Reading from deaths.
#> ✓ Range A5:F15.
deaths
```

```
#> # A tibble: 10 × 6
#>   Name          Profession   Age `Has kids` `Date of birth`
#>   <chr>         <chr>      <dbl> <lgl>      <dttm>
#> 1 David Bowie   musician      69 TRUE       1947-01-08 00:00:00
#> 2 Carrie Fisher actor         60 TRUE       1956-10-21 00:00:00
#> 3 Chuck Berry   musician      90 TRUE       1926-10-18 00:00:00
#> 4 Bill Paxton   actor         61 TRUE       1955-05-17 00:00:00
#> 5 Prince        musician      57 TRUE       1958-06-07 00:00:00
#> 6 Alan Rickman  actor         69 FALSE      1946-02-21 00:00:00
#> # … with 4 more rows, and 1 more variable: `Date of death` <dttm>
```

Writing to Google Sheets

You can write from R to Google Sheets with `write_sheet()`. The first argument is the data frame to write, and the second argument is the name (or other identifier) of the Google Sheet to write to:

```
write_sheet(bake_sale, ss = "bake-sale")
```

If you'd like to write your data to a specific (work)sheet inside a Google Sheet, you can specify that with the `sheet` argument as well:

```
write_sheet(bake_sale, ss = "bake-sale", sheet = "Sales")
```

Authentication

While you can read from a public Google Sheet without authenticating with your Google account, reading a private sheet or writing to a sheet requires authentication so that googlesheets4 can view and manage *your* Google Sheets.

When you attempt to read in a sheet that requires authentication, googlesheets4 will direct you to a web browser with a prompt to sign in to your Google account and grant permission to operate on your behalf with Google Sheets. However, if you want to specify a specific Google account, authentication scope, etc., you can do so with `gs4_auth()`, e.g., `gs4_auth(email = "mine@example.com")`, which will force the use of a token associated with a specific email. For further authentication details, we recommend reading the googlesheets4 auth vignette.

Exercises

1. Read the `students` dataset from earlier in the chapter from Excel and also from Google Sheets, with no additional arguments supplied to the `read_excel()` and `read_sheet()` functions. Are the resulting data frames in R exactly the same? If not, how are they different?
2. Read the Google Sheet titled survey, with `survey_id` as a character variable and `n_pets` as a numerical variable.
3. Read the Google Sheet titled roster. The resulting data frame should be called `roster` and should look like the following:

```
#> # A tibble: 12 × 3
#>    group subgroup    id
#>    <dbl> <chr>    <dbl>
#>  1     1 A            1
#>  2     1 A            2
#>  3     1 A            3
#>  4     1 B            4
#>  5     1 B            5
#>  6     1 B            6
#>  7     1 B            7
#>  8     2 A            8
#>  9     2 A            9
#> 10     2 B           10
#> 11     2 B           11
#> 12     2 B           12
```

Summary

Microsoft Excel and Google Sheets are two of the most popular spreadsheet systems. Being able to interact with data stored in Excel and Google Sheets files directly from R is a superpower! In this chapter, you learned how to read data into R from spreadsheets from Excel with `read_excel()` from the readxl package and from Google Sheets with `read_sheet()` from the googlesheets4 package. These functions work very similarly to each other and have similar arguments for specifying column names, `NA` strings, rows to skip on top of the file you're reading in, etc. Additionally, both functions make it possible to read a single sheet from a spreadsheet.

On the other hand, writing to an Excel file requires a different package and function (`writexl::write_xlsx()`), while you can write to a Google Sheet with the googlesheets4 package, with `write_sheet()`.

In the next chapter, you'll learn about a different data source, databases, and how to read data from that source into R.

CHAPTER 21
Databases

Introduction

A huge amount of data lives in databases, so it's essential that you know how to access it. Sometimes you can ask someone to download a snapshot into a `.csv` file for you, but this gets painful quickly: every time you need to make a change, you'll have to communicate with another human. You want to be able to reach into the database directly to get the data you need, when you need it.

In this chapter, you'll first learn the basics of the DBI package: how to use it to connect to a database and then retrieve data with a SQL[1] query. *SQL*, short for Structured Query Language, is the lingua franca of databases and is an important language for all data scientists to learn. That said, we're not going to start with SQL, but instead we'll teach you dbplyr, which can translate your dplyr code to SQL. We'll use that as a way to teach you some of the most important features of SQL. You won't become a SQL master by the end of the chapter, but you will be able to identify the most important components and understand what they do.

Prerequisites

In this chapter, we'll introduce DBI and dbplyr. DBI is a low-level interface that connects to databases and executes SQL; dbplyr is a high-level interface that translates your dplyr code to SQL queries and then executes them with DBI.

```
library(DBI)
library(dbplyr)
library(tidyverse)
```

[1] SQL is either pronounced "s"-"q"-"l" or "sequel."

Database Basics

At the simplest level, you can think about a database as a collection of data frames, called *tables* in database terminology. Like a `data.frame`, a database table is a collection of named columns, where every value in the column is the same type. There are three high-level differences between data frames and database tables:

- Database tables are stored on disk and can be arbitrarily large. Data frames are stored in memory and are fundamentally limited (although that limit is still plenty large for many problems).
- Database tables almost always have indexes. Much like the index of a book, a database index makes it possible to quickly find rows of interest without having to look at every single row. Data frames and tibbles don't have indexes, but data tables do, which is one of the reasons that they're so fast.
- Most classical databases are optimized for rapidly collecting data, not analyzing existing data. These databases are called *row-oriented* because the data is stored row by row, rather than column by column like R. More recently, there's been much development of *column-oriented* databases that make analyzing the existing data much faster.

Databases are run by database management systems (*DBMS* for short), which come in three basic forms:

- *Client-server* DBMS run on a powerful central server, which you connect from your computer (the client). They are great for sharing data with multiple people in an organization. Popular client-server DBMS include PostgreSQL, MariaDB, SQL Server, and Oracle.
- *Cloud* DBMS, like Snowflake, Amazon's RedShift, and Google's BigQuery, are similar to client-server DBMS, but they run in the cloud. This means they can easily handle extremely large datasets and can automatically provide more compute resources as needed.
- *In-process* DBMS, like SQLite or duckdb, run entirely on your computer. They're great for working with large datasets where you're the primary user.

Connecting to a Database

To connect to the database from R, you'll use a pair of packages:

- You'll always use DBI (*d*ata*b*ase *i*nterface) because it provides a set of generic functions that connect to the database, upload data, run SQL queries, etc.
- You'll also use a package tailored for the DBMS you're connecting to. This package translates the generic DBI commands into the specifics needed for a

given DBMS. There's usually one package for each DBMS, e.g., RPostgres for PostgreSQL and RMariaDB for MySQL.

If you can't find a specific package for your DBMS, you can usually use the odbc package instead. This uses the ODBC protocol supported by many DBMS. odbc requires a little more setup because you'll also need to install an ODBC driver and tell the odbc package where to find it.

Concretely, you create a database connection using `DBI::dbConnect()`. The first argument selects the DBMS,[2] and then the second and subsequent arguments describe how to connect to it (i.e., where it lives and the credentials that you need to access it). The following code shows a couple of typical examples:

```
con <- DBI::dbConnect(
  RMariaDB::MariaDB(),
  username = "foo"
)
con <- DBI::dbConnect(
  RPostgres::Postgres(),
  hostname = "databases.mycompany.com",
  port = 1234
)
```

The precise details of the connection vary a lot from DBMS to DBMS, so unfortunately we can't cover all the details here. This means you'll need to do a little research on your own. Typically you can ask the other data scientists in your team or talk to your DBA (*d*ata*b*ase *a*dministrator). The initial setup will often take a little fiddling (and maybe some googling) to get it right, but you'll generally need to do it only once.

In This Book

Setting up a client-server or cloud DBMS would be a pain for this book, so we'll instead use an in-process DBMS that lives entirely in an R package: duckdb. Thanks to the magic of DBI, the only difference between using duckdb and any other DBMS is how you'll connect to the database. This makes it great to teach with because you can easily run this code as well as easily take what you learn and apply it elsewhere.

Connecting to duckdb is particularly simple because the defaults create a temporary database that is deleted when you quit R. That's great for learning because it guarantees that you'll start from a clean slate every time you restart R:

```
con <- DBI::dbConnect(duckdb::duckdb())
```

duckdb is a high-performance database that's designed very much for the needs of a data scientist. We use it here because it's easy to get started with, but it's also capable

[2] Typically, this is the only function you'll use from the client package, so we recommend using `::` to pull out that one function, rather than loading the complete package with `library()`.

of handling gigabytes of data with great speed. If you want to use duckdb for a real data analysis project, you'll also need to supply the `dbdir` argument to make a persistent database and tell duckdb where to save it. Assuming you're using a project (Chapter 6), it's reasonable to store it in the `duckdb` directory of the current project:

```
con <- DBI::dbConnect(duckdb::duckdb(), dbdir = "duckdb")
```

Load Some Data

Since this is a new database, we need to start by adding some data. Here we'll add the `mpg` and `diamonds` datasets from ggplot2 using `DBI::dbWriteTable()`. The simplest usage of `dbWriteTable()` needs three arguments: a database connection, the name of the table to create in the database, and a data frame of data.

```
dbWriteTable(con, "mpg", ggplot2::mpg)
dbWriteTable(con, "diamonds", ggplot2::diamonds)
```

If you're using duckdb in a real project, we highly recommend learning about `duckdb_read_csv()` and `duckdb_register_arrow()`. These give you powerful and performant ways to quickly load data directly into duckdb, without having to first load it into R. We'll also show off a useful technique for loading multiple files into a database in "Writing to a Database" on page 483.

DBI Basics

You can check that the data is loaded correctly by using a couple of other DBI functions: `dbListTable()` lists all tables in the database,[3] and `dbReadTable()` retrieves the contents of a table.

```
dbListTables(con)
#> [1] "diamonds" "mpg"

con |>
  dbReadTable("diamonds") |>
  as_tibble()
#> # A tibble: 53,940 × 10
#>   carat cut       color clarity depth table price     x     y     z
#>   <dbl> <fct>     <fct> <fct>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
#> 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
#> 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
#> 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
#> 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
#> 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
#> # … with 53,934 more rows
```

`dbReadTable()` returns a `data.frame`, so we use `as_tibble()` to convert it into a tibble so that it prints nicely.

[3] At least, all the tables that you have permission to see.

If you already know SQL, you can use `dbGetQuery()` to get the results of running a query on the database:

```
sql <- "
  SELECT carat, cut, clarity, color, price
  FROM diamonds
  WHERE price > 15000
"
as_tibble(dbGetQuery(con, sql))
#> # A tibble: 1,655 × 5
#>   carat cut       clarity color price
#>   <dbl> <fct>     <fct>   <fct> <int>
#> 1  1.54 Premium   VS2     E     15002
#> 2  1.19 Ideal     VVS1    F     15005
#> 3  2.1  Premium   SI1     I     15007
#> 4  1.69 Ideal     SI1     D     15011
#> 5  1.5  Very Good VVS2    G     15013
#> 6  1.73 Very Good VS1     G     15014
#> # … with 1,649 more rows
```

If you've never seen SQL before, don't worry! You'll learn more about it shortly. But if you read it carefully, you might guess that it selects five columns of the `diamonds` dataset and all the rows where `price` is greater than 15,000.

dbplyr Basics

Now that we've connected to a database and loaded up some data, we can start to learn about dbplyr. dbplyr is a dplyr *backend*, which means you keep writing dplyr code but the backend executes it differently. In this, dbplyr translates to SQL; other backends include dtplyr, which translates to data.table, and multidplyr, which executes your code on multiple cores.

To use dbplyr, you must first use `tbl()` to create an object that represents a database table:

```
diamonds_db <- tbl(con, "diamonds")
diamonds_db
#> # Source:   table<diamonds> [?? x 10]
#> # Database: DuckDB 0.6.1 [root@Darwin 22.3.0:R 4.2.1/:memory:]
#>   carat cut       color clarity depth table price     x     y     z
#>   <dbl> <fct>     <fct> <fct>   <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1  0.23 Ideal     E     SI2      61.5    55   326  3.95  3.98  2.43
#> 2  0.21 Premium   E     SI1      59.8    61   326  3.89  3.84  2.31
#> 3  0.23 Good      E     VS1      56.9    65   327  4.05  4.07  2.31
#> 4  0.29 Premium   I     VS2      62.4    58   334  4.2   4.23  2.63
#> 5  0.31 Good      J     SI2      63.3    58   335  4.34  4.35  2.75
#> 6  0.24 Very Good J     VVS2     62.8    57   336  3.94  3.96  2.48
#> # … with more rows
```

There are two other common ways to interact with a database. First, many corporate databases are very large so you need some hierarchy to keep all the tables organized. In that case you might need to supply a schema, or a catalog and a schema, to pick the table you're interested in:

```
diamonds_db <- tbl(con, in_schema("sales", "diamonds"))
diamonds_db <- tbl(
  con, in_catalog("north_america", "sales", "diamonds")
  )
```

Other times you might want to use your own SQL query as a starting point:

```
diamonds_db <- tbl(con, sql("SELECT * FROM diamonds"))
```

This object is *lazy*; when you use dplyr verbs on it, dplyr doesn't do any work: it just records the sequence of operations that you want to perform and performs them only when needed. For example, take the following pipeline:

```
big_diamonds_db <- diamonds_db |>
  filter(price > 15000) |>
  select(carat:clarity, price)

big_diamonds_db
#> # Source:   SQL [?? x 5]
#> # Database: DuckDB 0.6.1 [root@Darwin 22.3.0:R 4.2.1/:memory:]
#>   carat cut       color clarity price
#>   <dbl> <fct>     <fct> <fct>   <int>
#> 1  1.54 Premium   E     VS2     15002
#> 2  1.19 Ideal     F     VVS1    15005
#> 3  2.1  Premium   I     SI1     15007
#> 4  1.69 Ideal     D     SI1     15011
#> 5  1.5  Very Good G     VVS2    15013
#> 6  1.73 Very Good G     VS1     15014
#> # … with more rows
```

You can tell this object represents a database query because it prints the DBMS name at the top, and while it tells you the number of columns, it typically doesn't know the number of rows. This is because finding the total number of rows usually requires executing the complete query, something we're trying to avoid.

You can see the SQL code generated by dplyr with the `show_query()` function. If you know dplyr, this is a great way to learn SQL! Write some dplyr code, get dbplyr to translate it to SQL and then try to figure out how the two languages match up.

```
big_diamonds_db |>
  show_query()
#> <SQL>
#> SELECT carat, cut, color, clarity, price
#> FROM diamonds
#> WHERE (price > 15000.0)
```

To get all the data back into R, you call `collect()`. Behind the scenes, this generates the SQL, calls `dbGetQuery()` to get the data, and then turns the result into a tibble:

```
big_diamonds <- big_diamonds_db |>
  collect()
big_diamonds
#> # A tibble: 1,655 × 5
#>   carat cut       color clarity price
#>   <dbl> <fct>     <fct> <fct>   <int>
#> 1  1.54 Premium   E     VS2     15002
#> 2  1.19 Ideal     F     VVS1    15005
#> 3  2.1  Premium   I     SI1     15007
#> 4  1.69 Ideal     D     SI1     15011
#> 5  1.5  Very Good G     VVS2    15013
#> 6  1.73 Very Good G     VS1     15014
#> # … with 1,649 more rows
```

Typically, you'll use dbplyr to select the data you want from the database, performing basic filtering and aggregation using the translations described next. Then, once you're ready to analyze the data with functions that are unique to R, you'll collect the data using `collect()` to get an in-memory tibble and continue your work with pure R code.

SQL

The rest of the chapter will teach you a little SQL through the lens of dbplyr. It's a rather nontraditional introduction to SQL, but we hope it will get you quickly up to speed with the basics. Luckily, if you understand dplyr, you're in a great place to quickly pick up SQL because so many of the concepts are the same.

We'll explore the relationship between dplyr and SQL using a couple of old friends from the nycflights13 package: `flights` and `planes`. These datasets are easy to get into our learning database because dbplyr comes with a function that copies the tables from nycflights13 to our database:

```
dbplyr::copy_nycflights13(con)
#> Creating table: airlines
#> Creating table: airports
#> Creating table: flights
#> Creating table: planes
#> Creating table: weather
flights <- tbl(con, "flights")
planes <- tbl(con, "planes")
```

SQL Basics

The top-level components of SQL are called *statements*. Common statements include `CREATE` for defining new tables, `INSERT` for adding data, and `SELECT` for retrieving data. We will on focus on SELECT statements, also called *queries*, because they are almost exclusively what you'll use as a data scientist.

A query is made up of *clauses*. There are five important clauses: SELECT, FROM, WHERE, ORDER BY, and GROUP BY. Every query must have the SELECT[4] and FROM[5] clauses, and the simplest query is SELECT * FROM table, which selects all columns from the specified table. This is what dbplyr generates for an unadulterated table:

```
flights |> show_query()
#> <SQL>
#> SELECT *
#> FROM flights
planes |> show_query()
#> <SQL>
#> SELECT *
#> FROM planes
```

WHERE and ORDER BY control which rows are included and how they are ordered:

```
flights |>
  filter(dest == "IAH") |>
  arrange(dep_delay) |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM flights
#> WHERE (dest = 'IAH')
#> ORDER BY dep_delay
```

GROUP BY converts the query to a summary, causing aggregation to happen:

```
flights |>
  group_by(dest) |>
  summarize(dep_delay = mean(dep_delay, na.rm = TRUE)) |>
  show_query()
#> <SQL>
#> SELECT dest, AVG(dep_delay) AS dep_delay
#> FROM flights
#> GROUP BY dest
```

There are two important differences between dplyr verbs and SELECT clauses:

- In SQL, case doesn't matter: you can write select, SELECT, or even SeLeCt. In this book we'll stick with the common convention of writing SQL keywords in uppercase to distinguish them from table or variables names.
- In SQL, order matters: you must always write the clauses in the order SELECT, FROM, WHERE, GROUP BY, and ORDER BY. Confusingly, this order doesn't match how

[4] Confusingly, depending on the context, SELECT is either a statement or a clause. To avoid this confusion, we'll generally use SELECT query instead of SELECT statement.

[5] Technically, only the SELECT is required, since you can write queries like SELECT 1+1 to perform basic calculations. But if you want to work with data (as you always do!), you'll also need a FROM clause.

the clauses are actually evaluated, which is first FROM and then WHERE, GROUP BY, SELECT, and ORDER BY.

The following sections explore each clause in more detail.

> Note that while SQL is a standard, it is extremely complex, and no database follows the standard exactly. While the main components that we'll focus on in this book are similar between DBMSs, there are many minor variations. Fortunately, dbplyr is designed to handle this problem and generates different translations for different databases. It's not perfect, but it's continually improving, and if you hit a problem, you can file an issue on GitHub to help us do better.

SELECT

The SELECT clause is the workhorse of queries and performs the same job as select(), mutate(), rename(), relocate(), and, as you'll learn in the next section, summarize().

select(), rename(), and relocate() have very direct translations to SELECT as they just affect where a column appears (if at all) along with its name:

```
planes |>
  select(tailnum, type, manufacturer, model, year) |>
  show_query()
#> <SQL>
#> SELECT tailnum, "type", manufacturer, model, "year"
#> FROM planes

planes |>
  select(tailnum, type, manufacturer, model, year) |>
  rename(year_built = year) |>
  show_query()
#> <SQL>
#> SELECT tailnum, "type", manufacturer, model, "year" AS year_built
#> FROM planes

planes |>
  select(tailnum, type, manufacturer, model, year) |>
  relocate(manufacturer, model, .before = type) |>
  show_query()
#> <SQL>
#> SELECT tailnum, manufacturer, model, "type", "year"
#> FROM planes
```

This example also shows you how SQL does renaming. In SQL terminology, renaming is called *aliasing* and is done with AS. Note that unlike mutate(), the old name is on the left, and the new name is on the right.

In the previous examples, note that `"year"` and `"type"` are wrapped in double quotes. That's because these are *reserved words* in duckdb, so dbplyr quotes them to avoid any potential confusion between column/table names and SQL operators.

When working with other databases, you're likely to see every variable name quoted because only a handful of client packages, like duckdb, know what all the reserved words are, so they quote everything to be safe:

```
SELECT "tailnum", "type", "manufacturer", "model", "year"
FROM "planes"
```

Some other database systems use backticks instead of quotes:

```
SELECT `tailnum`, `type`, `manufacturer`, `model`, `year`
FROM `planes`
```

The translations for `mutate()` are similarly straightforward: each variable becomes a new expression in `SELECT`:

```
flights |>
  mutate(
    speed = distance / (air_time / 60)
  ) |>
  show_query()
#> <SQL>
#> SELECT *, distance / (air_time / 60.0) AS speed
#> FROM flights
```

We'll come back to the translation of individual components (like `/`) in "Function Translations" on page 391.

FROM

The `FROM` clause defines the data source. It's going to be rather uninteresting for a little while, because we're just using single tables. You'll see more complex examples once we hit the join functions.

GROUP BY

`group_by()` is translated to the `GROUP BY`[6] clause, and `summarize()` is translated to the `SELECT` clause:

```
diamonds_db |>
  group_by(cut) |>
  summarize(
    n = n(),
    avg_price = mean(price, na.rm = TRUE)
```

[6] This is no coincidence: the dplyr function name was inspired by the SQL clause.

```
  ) |>
  show_query()
#> <SQL>
#> SELECT cut, COUNT(*) AS n, AVG(price) AS avg_price
#> FROM diamonds
#> GROUP BY cut
```

We'll come back to what's happening with translating `n()` and `mean()` in "Function Translations" on page 391.

WHERE

`filter()` is translated to the `WHERE` clause:

```
flights |>
  filter(dest == "IAH" | dest == "HOU") |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM flights
#> WHERE (dest = 'IAH' OR dest = 'HOU')

flights |>
  filter(arr_delay > 0 & arr_delay < 20) |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM flights
#> WHERE (arr_delay > 0.0 AND arr_delay < 20.0)
```

There are a few important details to note here:

- | becomes OR, and & becomes AND.
- SQL uses = for comparison, not ==. SQL doesn't have assignment, so there's no potential for confusion there.
- SQL uses only '' for strings, not "". In SQL, "" is used to identify variables, like R's ``.

Another useful SQL operator is `IN`, which is close to R's `%in%`:

```
flights |>
  filter(dest %in% c("IAH", "HOU")) |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM flights
#> WHERE (dest IN ('IAH', 'HOU'))
```

SQL uses `NULL` instead of `NA`. `NULL`s behave similarly to `NA`s. The main difference is that while they're "infectious" in comparisons and arithmetic, they are silently dropped when summarizing. dbplyr will remind you about this behavior the first time you hit it:

```
flights |>
  group_by(dest) |>
  summarize(delay = mean(arr_delay))
#> Warning: Missing values are always removed in SQL aggregation functions.
#> Use `na.rm = TRUE` to silence this warning
#> This warning is displayed once every 8 hours.
#> # Source:   SQL [?? x 2]
#> # Database: DuckDB 0.6.1 [root@Darwin 22.3.0:R 4.2.1/:memory:]
#>   dest   delay
#>   <chr>  <dbl>
#> 1 ATL   11.3
#> 2 ORD    5.88
#> 3 RDU   10.1
#> 4 IAD   13.9
#> 5 DTW    5.43
#> 6 LAX    0.547
#> # … with more rows
```

If you want to learn more about how NULLs work, you might enjoy "The Three-Valued Logic of SQL" by Markus Winand.

In general, you can work with NULLs using the functions you'd use for NAs in R:

```
flights |>
  filter(!is.na(dep_delay)) |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM flights
#> WHERE (NOT((dep_delay IS NULL)))
```

This SQL query illustrates one of the drawbacks of dbplyr: while the SQL is correct, it isn't as simple as you might write by hand. In this case, you could drop the parentheses and use a special operator that's easier to read:

```
WHERE "dep_delay" IS NOT NULL
```

Note that if you `filter()` a variable that you created using a summarize, dbplyr will generate a HAVING clause, rather than a WHERE clause. This is a one of the idiosyncrasies of SQL: WHERE is evaluated before SELECT and GROUP BY, so SQL needs another clause that's evaluated afterward.

```
diamonds_db |>
  group_by(cut) |>
  summarize(n = n()) |>
  filter(n > 100) |>
  show_query()
#> <SQL>
#> SELECT cut, COUNT(*) AS n
#> FROM diamonds
#> GROUP BY cut
#> HAVING (COUNT(*) > 100.0)
```

ORDER BY

Ordering rows involves a straightforward translation from `arrange()` to the `ORDER BY` clause:

```
flights |>
  arrange(year, month, day, desc(dep_delay)) |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM flights
#> ORDER BY "year", "month", "day", dep_delay DESC
```

Notice how `desc()` is translated to `DESC`: this is one of the many dplyr functions whose name was directly inspired by SQL.

Subqueries

Sometimes it's not possible to translate a dplyr pipeline into a single `SELECT` statement and you need to use a subquery. A *subquery* is just a query used as a data source in the `FROM` clause, instead of the usual table.

dbplyr typically uses subqueries to work around the limitations of SQL. For example, expressions in the `SELECT` clause can't refer to columns that were just created. That means that the following (silly) dplyr pipeline needs to happen in two steps: the first (inner) query computes `year1`, and then the second (outer) query can compute `year2`:

```
flights |>
  mutate(
    year1 = year + 1,
    year2 = year1 + 1
  ) |>
  show_query()
#> <SQL>
#> SELECT *, year1 + 1.0 AS year2
#> FROM (
#>   SELECT *, "year" + 1.0 AS year1
#>   FROM flights
#> ) q01
```

You'll also see this if you attempted to `filter()` a variable that you just created. Remember, even though `WHERE` is written after `SELECT`, it's evaluated before it, so we need a subquery in this (silly) example:

```
flights |>
  mutate(year1 = year + 1) |>
  filter(year1 == 2014) |>
  show_query()
#> <SQL>
#> SELECT *
#> FROM (
#>   SELECT *, "year" + 1.0 AS year1
#>   FROM flights
#> ) q01
#> WHERE (year1 = 2014.0)
```

Sometimes dbplyr will create a subquery where it's not needed because it doesn't yet know how to optimize that translation. As dbplyr improves over time, these cases will get rarer but will probably never go away.

Joins

If you're familiar with dplyr's joins, SQL joins are similar. Here's a simple example:

```
flights |>
  left_join(planes |> rename(year_built = year), by = "tailnum") |>
  show_query()
#> <SQL>
#> SELECT
#>   flights.*,
#>   planes."year" AS year_built,
#>   "type",
#>   manufacturer,
#>   model,
#>   engines,
#>   seats,
#>   speed,
#>   engine
#> FROM flights
#> LEFT JOIN planes
#>   ON (flights.tailnum = planes.tailnum)
```

The main thing to notice here is the syntax: SQL joins use subclauses of the `FROM` clause to bring in additional tables, using `ON` to define how the tables are related.

dplyr's names for these functions are so closely connected to SQL that you can easily guess the equivalent SQL for `inner_join()`, `right_join()`, and `full_join()`:

```
SELECT flights.*, "type", manufacturer, model, engines, seats, speed
FROM flights
INNER JOIN planes ON (flights.tailnum = planes.tailnum)

SELECT flights.*, "type", manufacturer, model, engines, seats, speed
FROM flights
RIGHT JOIN planes ON (flights.tailnum = planes.tailnum)

SELECT flights.*, "type", manufacturer, model, engines, seats, speed
FROM flights
FULL JOIN planes ON (flights.tailnum = planes.tailnum)
```

You're likely to need many joins when working with data from a database. That's because database tables are often stored in a highly normalized form, where each "fact" is stored in a single place, and to keep a complete dataset for analysis, you need to navigate a complex network of tables connected by primary and foreign keys. If you hit this scenario, the dm package, by Tobias Schieferdecker, Kirill Müller, and Darko Bergant, is a lifesaver. It can automatically determine the connections between tables using the constraints that DBAs often supply, visualize the connections so you can see what's going on, and generate the joins you need to connect one table to another.

Other Verbs

dbplyr also translates other verbs such as distinct(), slice_*(), and intersect(), as well as a growing selection of tidyr functions such as pivot_longer() and pivot_wider(). The easiest way to see the full set of what's currently available is to visit the dbplyr website.

Exercises

1. What is distinct() translated to? How about head()?
2. Explain what each of the following SQL queries do and try to re-create them using dbplyr:

```
SELECT *
FROM flights
WHERE dep_delay < arr_delay

SELECT *, distance / (airtime / 60) AS speed
FROM flights
```

Function Translations

So far we've focused on the big picture of how dplyr verbs are translated to the clauses of a query. Now we're going to zoom in a little and talk about the translation of the R functions that work with individual columns; e.g., what happens when you use mean(x) in summarize()?

To help see what's going on, we'll use a couple of little helper functions that run a summarize() or mutate() and show the generated SQL. That will make it a little easier to explore a few variations and see how summaries and transformations can differ.

```
summarize_query <- function(df, ...) {
  df |>
    summarize(...) |>
    show_query()
}
mutate_query <- function(df, ...) {
  df |>
    mutate(..., .keep = "none") |>
    show_query()
}
```

Let's dive in with some summaries! Looking at the following code, you'll notice that some summary functions, such as mean(), have a relatively simple translation, while others like `median()` are much more complex. The complexity is typically higher for operations that are common in statistics but less common in databases.

```
flights |>
  group_by(year, month, day) |>
  summarize_query(
    mean = mean(arr_delay, na.rm = TRUE),
    median = median(arr_delay, na.rm = TRUE)
  )
#> `summarise()` has grouped output by "year" and "month". You can override
#> using the `.groups` argument.
#> <SQL>
#> SELECT
#>   "year",
#>   "month",
#>   "day",
#>   AVG(arr_delay) AS mean,
#>   PERCENTILE_CONT(0.5) WITHIN GROUP (ORDER BY arr_delay) AS median
#> FROM flights
#> GROUP BY "year", "month", "day"
```

The translation of summary functions becomes more complicated when you use them inside a `mutate()` because they have to turn into so-called *window* functions. In SQL, you turn an ordinary aggregation function into a window function by adding `OVER` after it:

```
flights |>
  group_by(year, month, day) |>
  mutate_query(
    mean = mean(arr_delay, na.rm = TRUE),
  )
#> <SQL>
#> SELECT
#>   "year",
#>   "month",
#>   "day",
#>   AVG(arr_delay) OVER (PARTITION BY "year", "month", "day") AS mean
#> FROM flights
```

In SQL, the `GROUP BY` clause is used exclusively for summaries, so here you can see that the grouping has moved from the `PARTITION BY` argument to `OVER`.

Window functions include all functions that look forward or backward, such as `lead()` and `lag()`, which look at the "previous" or "next" value, respectively:

```
flights |>
  group_by(dest) |>
  arrange(time_hour) |>
  mutate_query(
    lead = lead(arr_delay),
    lag = lag(arr_delay)
  )
#> <SQL>
#> SELECT
#>   dest,
#>   LEAD(arr_delay, 1, NULL) OVER (PARTITION BY dest ORDER BY time_hour) AS lead,
#>   LAG(arr_delay, 1, NULL) OVER (PARTITION BY dest ORDER BY time_hour) AS lag
#> FROM flights
#> ORDER BY time_hour
```

Here it's important to `arrange()` the data, because SQL tables have no intrinsic order. In fact, if you don't use `arrange()`, you might get the rows back in a different order every time! Notice for window functions, the ordering information is repeated: the `ORDER BY` clause of the main query doesn't automatically apply to window functions.

Another important SQL function is `CASE WHEN`. It's used as the translation of `if_else()` and `case_when()`, the dplyr function that it directly inspired. Here are a couple of simple examples:

```
flights |>
  mutate_query(
    description = if_else(arr_delay > 0, "delayed", "on-time")
  )
#> <SQL>
#> SELECT CASE WHEN
#>   (arr_delay > 0.0) THEN 'delayed'
#>   WHEN NOT (arr_delay > 0.0) THEN 'on-time' END AS description
#> FROM flights
flights |>
  mutate_query(
    description =
      case_when(
        arr_delay < -5 ~ "early",
        arr_delay < 5 ~ "on-time",
        arr_delay >= 5 ~ "late"
      )
  )
#> <SQL>
#> SELECT CASE
#> WHEN (arr_delay < -5.0) THEN 'early'
#> WHEN (arr_delay < 5.0) THEN 'on-time'
#> WHEN (arr_delay >= 5.0) THEN 'late'
#> END AS description
#> FROM flights
```

`CASE WHEN` is also used for some other functions that don't have a direct translation from R to SQL. A good example of this is `cut()`:

```
flights |>
  mutate_query(
    description =  cut(
      arr_delay,
      breaks = c(-Inf, -5, 5, Inf),
      labels = c("early", "on-time", "late")
    )
  )
#> <SQL>
#> SELECT CASE
#> WHEN (arr_delay <= -5.0) THEN 'early'
#> WHEN (arr_delay <= 5.0) THEN 'on-time'
#> WHEN (arr_delay > 5.0) THEN 'late'
#> END AS description
#> FROM flights
```

dbplyr also translates common string and date-time manipulation functions, which you can learn about in `vignette("translation-function", package = "dbplyr")`. dbplyr's translations are certainly not perfect, and there are many R functions that aren't translated yet, but dbplyr does a surprisingly good job covering the functions that you'll use most of the time.

Summary

In this chapter you learned how to access data from databases. We focused on dbplyr, a dplyr "backend" that allows you to write the dplyr code you're familiar with and have it be automatically translated to SQL. We used that translation to teach you a little SQL; it's important to learn some SQL because it's *the* most commonly used language for working with data and knowing some will make it easier for you to communicate with other data folks who don't use R. If you've finished this chapter and would like to learn more about SQL, we have two recommendations:

- *SQL for Data Scientists* by Renée M. P. Teate is an introduction to SQL designed specifically for the needs of data scientists and includes examples of the sort of highly interconnected data you're likely to encounter in real organizations.
- *Practical SQL* by Anthony DeBarros is written from the perspective of a data journalist (a data scientist specialized in telling compelling stories) and goes into more detail about getting your data into a database and running your own DBMS.

In the next chapter, we'll learn about another dplyr backend for working with large data: arrow. The arrow package is designed for working with large files on disk and is a natural complement to databases.

CHAPTER 22

Arrow

Introduction

CSV files are designed to be easily read by humans. They're a good interchange format because they're simple and they can be read by every tool under the sun. But CSV files aren't efficient: you have to do quite a lot of work to read the data into R. In this chapter, you'll learn about a powerful alternative: the parquet format, an open standards–based format widely used by big data systems.

We'll pair parquet files with Apache Arrow, a multilanguage toolbox designed for efficient analysis and transport of large datasets. We'll use Apache Arrow via the arrow package, which provides a dplyr backend allowing you to analyze larger-than-memory datasets using familiar dplyr syntax. As an additional benefit, arrow is extremely fast; you'll see some examples later in the chapter.

Both arrow and dbplyr provide dplyr backends, so you might wonder when to use each. In many cases, the choice is made for you, as in the data is already in a database or in parquet files, and you'll want to work with it as is. But if you're starting with your own data (perhaps CSV files), you can either load it into a database or convert it to parquet. In general, it's hard to know what will work best, so in the early stages of your analysis, we encourage you to try both and pick the one that works the best for you.

(A big thanks to Danielle Navarro who contributed the initial version of this chapter.)

Prerequisites

In this chapter, we'll continue to use the tidyverse, particularly dplyr, but we'll pair it with the arrow package, which was designed specifically for working with large data:

```
library(tidyverse)
library(arrow)
```

Later in the chapter, we'll also see some connections between arrow and duckdb, so we'll also need dbplyr and duckdb:

```
library(dbplyr, warn.conflicts = FALSE)
library(duckdb)
#> Loading required package: DBI
```

Getting the Data

We begin by getting a dataset worthy of these tools: a dataset of item checkouts from Seattle public libraries, available online at Seattle Open Data. This dataset contains 41,389,465 rows that tell you how many times each book was checked out each month from April 2005 to October 2022.

The following code will get you a cached copy of the data. The data is a 9 GB CSV file, so it will take some time to download. I highly recommend using `curl::multi download()` to get very large files as it's built for exactly this purpose: it gives you a progress bar, and it can resume the download if it's interrupted.

```
dir.create("data", showWarnings = FALSE)

curl::multi_download(
  "https://r4ds.s3.us-west-2.amazonaws.com/seattle-library-checkouts.csv",
  "data/seattle-library-checkouts.csv",
  resume = TRUE
)
```

Opening a Dataset

Let's start by taking a look at the data. At 9 GB, this file is large enough that we probably don't want to load the whole thing into memory. A good rule of thumb is that you usually want at least twice as much memory as the size of the data, and many laptops top out at 16 GB. This means we want to avoid `read_csv()` and instead use `arrow::open_dataset()`:

```
seattle_csv <- open_dataset(
  sources = "data/seattle-library-checkouts.csv",
  format = "csv"
)
```

What happens when this code is run? `open_dataset()` will scan a few thousand rows to figure out the structure of the dataset. Then it records what it's found and stops; it will only read further rows as you specifically request them. This metadata is what we see if we print `seattle_csv`:

```
seattle_csv
#> FileSystemDataset with 1 csv file
#> UsageClass: string
#> CheckoutType: string
#> MaterialType: string
#> CheckoutYear: int64
#> CheckoutMonth: int64
#> Checkouts: int64
#> Title: string
#> ISBN: null
#> Creator: string
#> Subjects: string
#> Publisher: string
#> PublicationYear: string
```

The first line in the output tells you that `seattle_csv` is stored locally on disk as a single CSV file; it will be loaded into memory only as needed. The remainder of the output tells you the column type that arrow has imputed for each column.

We can see what's actually in with `glimpse()`. This reveals that there are ~41 million rows and 12 columns and shows us a few values.

```
seattle_csv |> glimpse()
#> FileSystemDataset with 1 csv file
#> 41,389,465 rows x 12 columns
#> $ UsageClass      <string> "Physical", "Physical", "Digital", "Physical", "Ph…
#> $ CheckoutType    <string> "Horizon", "Horizon", "OverDrive", "Horizon", "Hor…
#> $ MaterialType    <string> "BOOK", "BOOK", "EBOOK", "BOOK", "SOUNDDISC", "BOO…
#> $ CheckoutYear     <int64> 2016, 2016, 2016, 2016, 2016, 2016, 2016, 2016, 20…
#> $ CheckoutMonth    <int64> 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6,…
#> $ Checkouts        <int64> 1, 1, 1, 1, 1, 1, 1, 1, 4, 1, 1, 2, 3, 2, 1, 3, 2,…
#> $ Title           <string> "Super rich : a guide to having it all / Russell S…
#> $ ISBN            <string> "", "", "", "", "", "", "", "", "", "", "", "", ""…
#> $ Creator         <string> "Simmons, Russell", "Barclay, James, 1965-", "Tim …
#> $ Subjects        <string> "Self realization, Conduct of life, Attitude Psych…
#> $ Publisher       <string> "Gotham Books,", "Pyr,", "Random House, Inc.", "Di…
#> $ PublicationYear <string> "c2011.", "2010.", "2015", "2005.", "c2004.", "c20…
```

We can start to use this dataset with dplyr verbs, using `collect()` to force arrow to perform the computation and return some data. For example, this code tells us the total number of checkouts per year:

```
seattle_csv |>
  count(CheckoutYear, wt = Checkouts) |>
  arrange(CheckoutYear) |>
  collect()
#> # A tibble: 18 × 2
#>   CheckoutYear       n
#>          <int>   <int>
#> 1         2005 3798685
#> 2         2006 6599318
#> 3         2007 7126627
#> 4         2008 8438486
#> 5         2009 9135167
#> 6         2010 8608966
#> # … with 12 more rows
```

Thanks to arrow, this code will work regardless of how large the underlying dataset is. But it's currently rather slow: on Hadley's computer, it took ~10s to run. That's not terrible given how much data we have, but we can make it much faster by switching to a better format.

The Parquet Format

To make this data easier to work with, let's switch to the parquet file format and split it up into multiple files. The following sections will first introduce you to parquet and partitioning and then apply what we learned to the Seattle library data.

Advantages of Parquet

Like CSV, parquet is used for rectangular data, but instead of being a text format that you can read with any file editor, it's a custom binary format designed specifically for the needs of big data. This means that:

- Parquet files are usually smaller than the equivalent CSV file. Parquet relies on efficient encodings to keep file size down and supports file compression. This helps make parquet files fast because there's less data to move from disk to memory.
- Parquet files have a rich type system. As we talked about in "Controlling Column Types" on page 104, a CSV file does not provide any information about column types. For example, a CSV reader has to guess whether `"08-10-2022"` should be parsed as a string or a date. In contrast, parquet files store data in a way that records the type along with the data.
- Parquet files are "column-oriented." This means they're organized column by column, much like R's data frame. This typically leads to better performance for data analysis tasks compared to CSV files, which are organized row by row.
- Parquet files are "chunked," which makes it possible to work on different parts of the file at the same time and, if you're lucky, to skip some chunks altogether.

Partitioning

As datasets get larger and larger, storing all the data in a single file gets increasingly painful, and it's often useful to split large datasets across many files. When this structuring is done intelligently, this strategy can lead to significant improvements in performance because many analyses will require only a subset of the files.

There are no hard and fast rules about how to partition your dataset: the results will depend on your data, access patterns, and the systems that read the data. You're likely to need to do some experimentation before you find the ideal partitioning for your situation. As a rough guide, arrow suggests that you avoid files smaller than 20 MB and larger than 2 GB and avoid partitions that produce more than 10,000 files. You should also try to partition by variables that you filter by; as you'll see shortly, that allows arrow to skip a lot of work by reading only the relevant files.

Rewriting the Seattle Library Data

Let's apply these ideas to the Seattle library data to see how they play out in practice. We're going to partition by `CheckoutYear`, since it's likely some analyses will want to look at only recent data and partitioning by year yields 18 chunks of a reasonable size.

To rewrite the data, we define the partition using `dplyr::group_by()` and then save the partitions to a directory with `arrow::write_dataset()`. `write_dataset()` has two important arguments: a directory where we'll create the files and the format we'll use.

```
pq_path <- "data/seattle-library-checkouts"

seattle_csv |>
  group_by(CheckoutYear) |>
  write_dataset(path = pq_path, format = "parquet")
```

This takes about a minute to run; as we'll see shortly this is an initial investment that pays off by making future operations much much faster.

Let's take a look at what we just produced:

```
tibble(
  files = list.files(pq_path, recursive = TRUE),
  size_MB = file.size(file.path(pq_path, files)) / 1024^2
)
#> # A tibble: 18 × 2
#>   files                            size_MB
#>   <chr>                              <dbl>
#> 1 CheckoutYear=2005/part-0.parquet    109.
#> 2 CheckoutYear=2006/part-0.parquet    164.
#> 3 CheckoutYear=2007/part-0.parquet    178.
#> 4 CheckoutYear=2008/part-0.parquet    195.
#> 5 CheckoutYear=2009/part-0.parquet    214.
#> 6 CheckoutYear=2010/part-0.parquet    222.
#> # … with 12 more rows
```

Our single 9 GB CSV file has been rewritten into 18 parquet files. The filenames use a "self-describing" convention used by the Apache Hive project. Hive-style partitions name folders with a "key=value" convention, so as you might guess, the `CheckoutYear=2005` directory contains all the data where `CheckoutYear` is 2005. Each file is between 100 and 300 MB and the total size is now around 4 GB, a little more than half the size of the original CSV file. This is as we expect since parquet is a much more efficient format.

Using dplyr with Arrow

Now that we've created these parquet files, we'll need to read them in again. We use `open_dataset()` again, but this time we give it a directory:

```
seattle_pq <- open_dataset(pq_path)
```

Now we can write our dplyr pipeline. For example, we could count the total number of books checked out in each month for the last five years:

```
query <- seattle_pq |>
  filter(CheckoutYear >= 2018, MaterialType == "BOOK") |>
  group_by(CheckoutYear, CheckoutMonth) |>
  summarize(TotalCheckouts = sum(Checkouts)) |>
  arrange(CheckoutYear, CheckoutMonth)
```

Writing dplyr code for arrow data is conceptually similar to dbplyr, as discussed in Chapter 21: you write dplyr code, which is automatically transformed into a query that the Apache Arrow C++ library understands, which is then executed when you call `collect()`. If we print out the `query` object, we can see a little information about what we expect Arrow to return when the execution takes place:

```
query
#> FileSystemDataset (query)
#> CheckoutYear: int32
#> CheckoutMonth: int64
#> TotalCheckouts: int64
#>
#> * Grouped by CheckoutYear
#> * Sorted by CheckoutYear [asc], CheckoutMonth [asc]
#> See $.data for the source Arrow object
```

And we can get the results by calling `collect()`:

```
query |> collect()
#> # A tibble: 58 × 3
#> # Groups:   CheckoutYear [5]
#>   CheckoutYear CheckoutMonth TotalCheckouts
#>          <int>         <int>          <int>
#> 1         2018             1         355101
#> 2         2018             2         309813
#> 3         2018             3         344487
#> 4         2018             4         330988
#> 5         2018             5         318049
#> 6         2018             6         341825
#> # … with 52 more rows
```

Like dbplyr, arrow understands only some R expressions, so you may not be able to write exactly the same code you usually would. However, the list of operations and functions supported is fairly extensive and continues to grow; find a complete list of currently supported functions in `?acero`.

Performance

Let's take a quick look at the performance impact of switching from CSV to parquet. First, let's time how long it takes to calculate the number of books checked out in each month of 2021, when the data is stored as a single large CSV file:

```
seattle_csv |>
  filter(CheckoutYear == 2021, MaterialType == "BOOK") |>
  group_by(CheckoutMonth) |>
  summarize(TotalCheckouts = sum(Checkouts)) |>
  arrange(desc(CheckoutMonth)) |>
  collect() |>
  system.time()
#>    user  system elapsed
#>  11.997   1.189  11.343
```

Now let's use our new version of the dataset in which the Seattle library checkout data has been partitioned into 18 smaller parquet files:

```
seattle_pq |>
  filter(CheckoutYear == 2021, MaterialType == "BOOK") |>
  group_by(CheckoutMonth) |>
  summarize(TotalCheckouts = sum(Checkouts)) |>
  arrange(desc(CheckoutMonth)) |>
  collect() |>
  system.time()
#>    user  system elapsed
#>   0.272   0.063   0.063
```

The ~100x speedup in performance is attributable to two factors: the multifile partitioning and the format of individual files:

- Partitioning improves performance because this query uses `CheckoutYear == 2021` to filter the data, and arrow is smart enough to recognize that it needs to read only 1 of the 18 parquet files.
- The parquet format improves performance by storing data in a binary format that can be read more directly into memory. The column-wise format and rich metadata means that arrow needs to read only the four columns actually used in the query (`CheckoutYear`, `MaterialType`, `CheckoutMonth`, and `Checkouts`).

This massive difference in performance is why it pays off to convert large CSVs to parquet!

Using dbplyr with Arrow

There's one last advantage of parquet and arrow—it's easy to turn an arrow dataset into a DuckDB database (Chapter 21) by calling `arrow::to_duckdb()`:

```
seattle_pq |>
  to_duckdb() |>
  filter(CheckoutYear >= 2018, MaterialType == "BOOK") |>
  group_by(CheckoutYear) |>
  summarize(TotalCheckouts = sum(Checkouts)) |>
  arrange(desc(CheckoutYear)) |>
  collect()
#> Warning: Missing values are always removed in SQL aggregation functions.
#> Use `na.rm = TRUE` to silence this warning
#> This warning is displayed once every 8 hours.
#> # A tibble: 5 × 2
#>   CheckoutYear TotalCheckouts
#>          <int>          <dbl>
#> 1         2022        2431502
#> 2         2021        2266438
#> 3         2020        1241999
#> 4         2019        3931688
#> 5         2018        3987569
```

The neat thing about `to_duckdb()` is that the transfer doesn't involve any memory copying and speaks to the goals of the arrow ecosystem: enabling seamless transitions from one computing environment to another.

Summary

In this chapter, you got a taste of the arrow package, which provides a dplyr backend for working with large on-disk datasets. It can work with CSV files, and it's much much faster if you convert your data to parquet. Parquet is a binary data format that's designed specifically for data analysis on modern computers. Far fewer tools can work with parquet files compared to CSV, but its partitioned, compressed, and columnar structure makes it much more efficient to analyze.

Next up you'll learn about your first nonrectangular data source, which you'll handle using tools provided by the tidyr package. We'll focus on data that comes from JSON files, but the general principles apply to tree-like data regardless of its source.

CHAPTER 23

Hierarchical Data

Introduction

In this chapter, you'll learn the art of data *rectangling*, taking data that is fundamentally hierarchical, or tree-like, and converting it into a rectangular data frame made up of rows and columns. This is important because hierarchical data is surprisingly common, especially when working with data that comes from the web.

To learn about rectangling, you'll need to first learn about lists, the data structure that makes hierarchical data possible. Then you'll learn about two crucial tidyr functions: `tidyr::unnest_longer()` and `tidyr::unnest_wider()`. We'll then show you a few case studies, applying these simple functions again and again to solve real problems. We'll finish off by talking about JSON, the most frequent source of hierarchical datasets and a common format for data exchange on the web.

Prerequisites

In this chapter, we'll use many functions from tidyr, a core member of the tidyverse. We'll also use *repurrrsive* to provide some interesting datasets for rectangling practice, and we'll finish by using *jsonlite* to read JSON files into R lists.

```
library(tidyverse)
library(repurrrsive)
library(jsonlite)
```

Lists

So far you've worked with data frames that contain simple vectors such as integers, numbers, characters, date-times, and factors. These vectors are simple because they're homogeneous: every element is of the same data type. If you want to store elements of different types in the same vector, you'll need a *list*, which you create with `list()`:

```
x1 <- list(1:4, "a", TRUE)
x1
#> [[1]]
#> [1] 1 2 3 4
#> 
#> [[2]]
#> [1] "a"
#> 
#> [[3]]
#> [1] TRUE
```

It's often convenient to name the components, or *children*, of a list, which you can do in the same way as naming the columns of a tibble:

```
x2 <- list(a = 1:2, b = 1:3, c = 1:4)
x2
#> $a
#> [1] 1 2
#> 
#> $b
#> [1] 1 2 3
#> 
#> $c
#> [1] 1 2 3 4
```

Even for these simple lists, printing takes up quite a lot of space. A useful alternative is `str()`, which generates a compact display of the *str*ucture, de-emphasizing the contents:

```
str(x1)
#> List of 3
#>  $ : int [1:4] 1 2 3 4
#>  $ : chr "a"
#>  $ : logi TRUE
str(x2)
#> List of 3
#>  $ a: int [1:2] 1 2
#>  $ b: int [1:3] 1 2 3
#>  $ c: int [1:4] 1 2 3 4
```

As you can see, `str()` displays each child of the list on its own line. It displays the name, if present; then an abbreviation of the type; and then the first few values.

Hierarchy

Lists can contain any type of object, including other lists. This makes them suitable for representing hierarchical (tree-like) structures:

```
x3 <- list(list(1, 2), list(3, 4))
str(x3)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ :List of 2
#>   ..$ : num 3
#>   ..$ : num 4
```

This is notably different from `c()`, which generates a flat vector:

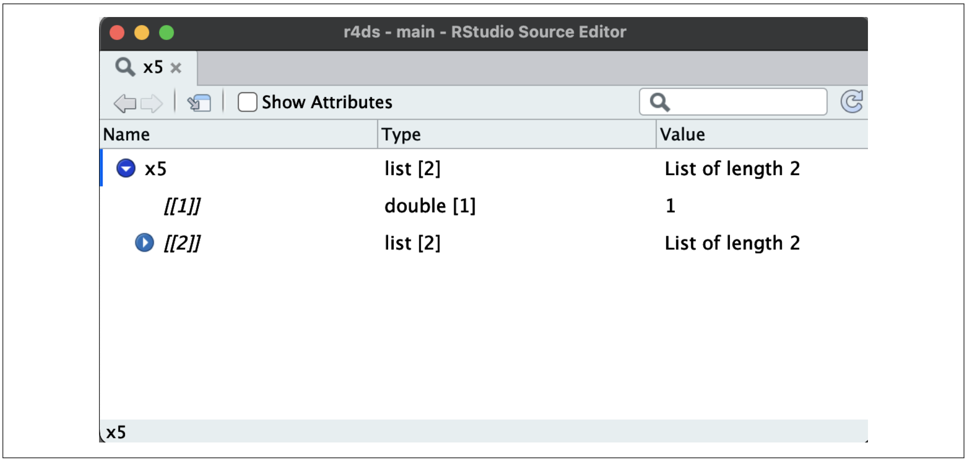
```
c(c(1, 2), c(3, 4))
#> [1] 1 2 3 4

x4 <- c(list(1, 2), list(3, 4))
str(x4)
#> List of 4
#>  $ : num 1
#>  $ : num 2
#>  $ : num 3
#>  $ : num 4
```

As lists get more complex, `str()` gets more useful, as it lets you see the hierarchy at a glance:

```
x5 <- list(1, list(2, list(3, list(4, list(5)))))
str(x5)
#> List of 2
#>  $ : num 1
#>  $ :List of 2
#>   ..$ : num 2
#>   ..$ :List of 2
#>   .. ..$ : num 3
#>   .. ..$ :List of 2
#>   .. .. ..$ : num 4
#>   .. .. ..$ :List of 1
#>   .. .. .. ..$ : num 5
```

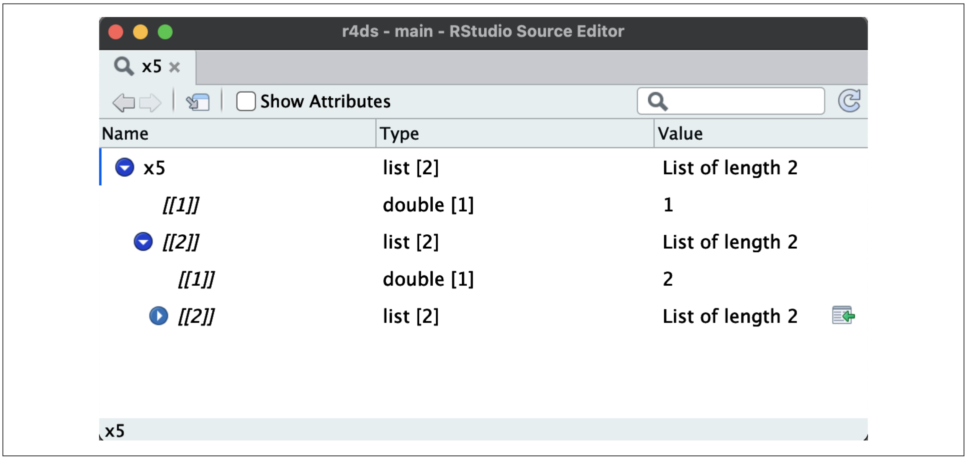
As lists get even larger and more complex, `str()` eventually starts to fail, and you'll need to switch to `View()`.[1] Figure 23-1 shows the result of calling `View(x5)`. The viewer starts by showing just the top level of the list, but you can interactively expand any of the components to see more, as in Figure 23-2. RStudio will also show you the code you need to access that element, as in Figure 23-3. We'll come back to how this code works in "Selecting a Single Element with $ and [[" on page 494.

[1] This is an RStudio feature.



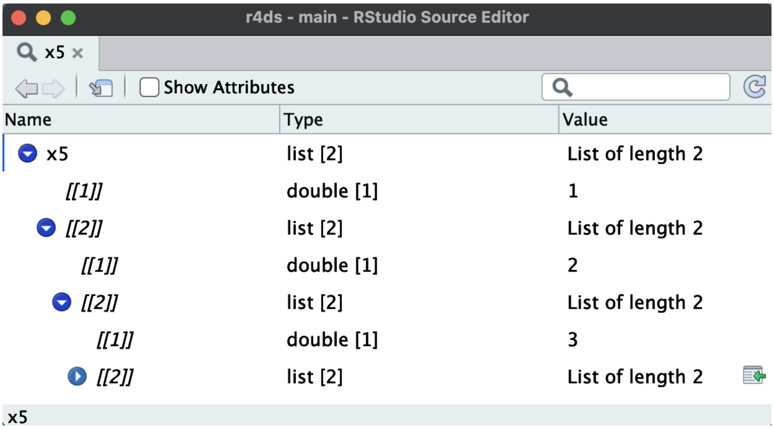
Name	Type	Value
x5	list [2]	List of length 2
[[1]]	double [1]	1
[[2]]	list [2]	List of length 2

Figure 23-1. The RStudio view lets you interactively explore a complex list. The viewer opens showing only the top level of the list.



Name	Type	Value
x5	list [2]	List of length 2
[[1]]	double [1]	1
[[2]]	list [2]	List of length 2
[[1]]	double [1]	2
[[2]]	list [2]	List of length 2

Figure 23-2. Clicking the right-facing triangle expands that component of the list so that you can also see its children.



Name	Type	Value
x5	list [2]	List of length 2
[[1]]	double [1]	1
[[2]]	list [2]	List of length 2
[[1]]	double [1]	2
[[2]]	list [2]	List of length 2
[[1]]	double [1]	3
[[2]]	list [2]	List of length 2

x5

Figure 23-3. You can repeat this operation as many times as needed to get to the data you're interested in. Note the bottom-left corner: if you click an element of the list, RStudio will give you the subsetting code needed to access it, in this case `x5[[2]][[2]][[2]]`.

List Columns

Lists can also live inside a tibble, where we call them list columns. List columns are useful because they allow you to place objects in a tibble that wouldn't usually belong in there. In particular, list columns are used a lot in the tidymodels ecosystem, because they allow you to store things like model outputs or resamples in a data frame.

Here's a simple example of a list column:

```
df <- tibble(
  x = 1:2,
  y = c("a", "b"),
  z = list(list(1, 2), list(3, 4, 5))
)
df
#> # A tibble: 2 × 3
#>       x y     z
#>   <int> <chr> <list>
#> 1     1 a     <list [2]>
#> 2     2 b     <list [3]>
```

There's nothing special about lists in a tibble; they behave like any other column:

```
df |>
  filter(x == 1)
#> # A tibble: 1 × 3
#>       x y     z
#>   <int> <chr> <list>
#> 1     1 a     <list [2]>
```

Computing with list columns is harder, but that's because computing with lists is harder in general; we'll come back to that in Chapter 26. In this chapter, we'll focus on unnesting list columns into regular variables so you can use your existing tools on them.

The default print method just displays a rough summary of the contents. The list column could be arbitrarily complex, so there's no good way to print it. If you want to see it, you'll need to pull out just the one list column and apply one of the techniques that you've learned previously, like `df |> pull(z) |> str()` or `df |> pull(z) |> View()`.

Base R

It's possible to put a list in a column of a `data.frame`, but it's a lot fiddlier because `data.frame()` treats a list as a list of columns:

```
data.frame(x = list(1:3, 3:5))
#>   x.1.3 x.3.5
#> 1     1     3
#> 2     2     4
#> 3     3     5
```

You can force `data.frame()` to treat a list as a list of rows by wrapping it in list `I()`, but the result doesn't print particularly well:

```
data.frame(
  x = I(list(1:2, 3:5)),
  y = c("1, 2", "3, 4, 5")
)
#>         x       y
#> 1    1, 2    1, 2
#> 2 3, 4, 5 3, 4, 5
```

It's easier to use list columns with tibbles because `tibble()` treats lists like vectors and the print method has been designed with lists in mind.

Unnesting

Now that you've learned the basics of lists and list columns, let's explore how you can turn them back into regular rows and columns. Here we'll use simple sample data so you can get the basic idea; in the next section we'll switch to real data.

List columns tend to come in two basic forms: named and unnamed. When the children are *named*, they tend to have the same names in every row. For example, in `df1`, every element of list column `y` has two elements named `a` and `b`. Named list columns naturally unnest into columns: each named element becomes a new named column.

```
df1 <- tribble(
  ~x, ~y,
    1, list(a = 11, b = 12),
    2, list(a = 21, b = 22),
    3, list(a = 31, b = 32),
  )
```

When the children are *unnamed*, the number of elements tends to vary from row to row. For example, in df2, the elements of list column y are unnamed and vary in length from one to three. Unnamed list columns naturally unnest into rows: you'll get one row for each child.

```
df2 <- tribble(
  ~x, ~y,
  1, list(11, 12, 13),
  2, list(21),
  3, list(31, 32),
)
```

tidyr provides two functions for these two cases: unnest_wider() and unnest_longer(). The following sections explain how they work.

unnest_wider()

When each row has the same number of elements with the same names, like df1, it's natural to put each component into its own column with unnest_wider():

```
df1 |>
  unnest_wider(y)
#> # A tibble: 3 × 3
#>       x     a     b
#>   <dbl> <dbl> <dbl>
#> 1     1    11    12
#> 2     2    21    22
#> 3     3    31    32
```

By default, the names of the new columns come exclusively from the names of the list elements, but you can use the names_sep argument to request that they combine the column name and the element name. This is useful for disambiguating repeated names.

```
df1 |>
  unnest_wider(y, names_sep = "_")
#> # A tibble: 3 × 3
#>       x   y_a   y_b
#>   <dbl> <dbl> <dbl>
#> 1     1    11    12
#> 2     2    21    22
#> 3     3    31    32
```

unnest_longer()

When each row contains an unnamed list, it's most natural to put each element into its own row with unnest_longer():

```
df2 |>
  unnest_longer(y)
#> # A tibble: 6 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1     1    11
#> 2     1    12
#> 3     1    13
#> 4     2    21
#> 5     3    31
#> 6     3    32
```

Note how x is duplicated for each element inside of y: we get one row of output for each element inside the list column. But what happens if one of the elements is empty, as in the following example?

```
df6 <- tribble(
  ~x, ~y,
  "a", list(1, 2),
  "b", list(3),
  "c", list()
)
df6 |> unnest_longer(y)
#> # A tibble: 3 × 2
#>   x         y
#>   <chr> <dbl>
#> 1 a         1
#> 2 a         2
#> 3 b         3
```

We get zero rows in the output, so the row effectively disappears. If you want to preserve that row, add NA in y, set keep_empty = TRUE.

Inconsistent Types

What happens if you unnest a list column that contains different types of vectors? For example, take the following dataset where list column y contains two numbers, a character, and a logical, which can't normally be mixed in a single column:

```
df4 <- tribble(
  ~x, ~y,
  "a", list(1),
  "b", list("a", TRUE, 5)
)
```

unnest_longer() always keeps the set of columns unchanged, while changing the number of rows. So what happens? How does unnest_longer() produce five rows while keeping everything in y?

```
df4 |>
  unnest_longer(y)
#> # A tibble: 4 × 2
#>   x     y
#>   <chr> <list>
#> 1 a     <dbl [1]>
#> 2 b     <chr [1]>
#> 3 b     <lgl [1]>
#> 4 b     <dbl [1]>
```

As you can see, the output contains a list column, but every element of the list column contains a single element. Because unnest_longer() can't find a common type of vector, it keeps the original types in a list column. You might wonder if this breaks the commandment that every element of a column must be the same type. It doesn't: every element is a list, even though the contents are of different types.

Dealing with inconsistent types is challenging and the details depend on the precise nature of the problem and your goals, but you'll most likely need tools from Chapter 26.

Other Functions

tidyr has a few other useful rectangling functions that we're not going to cover in this book:

- unnest_auto() automatically picks between unnest_longer() and unnest_wider() based on the structure of the list column. It's great for rapid exploration, but ultimately it's a bad idea because it doesn't force you to understand how your data is structured and makes your code harder to understand.
- unnest() expands both rows and columns. It's useful when you have a list column that contains a 2D structure like a data frame, which you don't see in this book, but you might encounter if you use the tidymodels ecosystem.

These functions are good to know about as you might encounter them when reading other people's code or tackling rarer rectangling challenges yourself.

Exercises

1. What happens when you use unnest_wider() with unnamed list columns like df2? What argument is now necessary? What happens to missing values?
2. What happens when you use unnest_longer() with named list columns like df1? What additional information do you get in the output? How can you suppress that extra detail?
3. From time to time you encounter data frames with multiple list columns with aligned values. For example, in the following data frame, the values of y and z are aligned (i.e., y and z will always have the same length within a row, and the first value of y corresponds to the first value of z). What happens if you apply two unnest_longer() calls to this data frame? How can you preserve the relationship between x and y? (Hint: Carefully read the docs.)

```
df4 <- tribble(
  ~x, ~y, ~z,
  "a", list("y-a-1", "y-a-2"), list("z-a-1", "z-a-2"),
  "b", list("y-b-1", "y-b-2", "y-b-3"), list("z-b-1", "z-b-2", "z-b-3")
)
```

Case Studies

The main difference between the simple examples we used earlier and real data is that real data typically contains multiple levels of nesting that require multiple calls to `unnest_longer()` and/or `unnest_wider()`. To show that in action, this section works through three real rectangling challenges using datasets from the repurrrsive package.

Very Wide Data

We'll start with `gh_repos`. This is a list that contains data about a collection of GitHub repositories retrieved using the GitHub API. It's a deeply nested list, so it's difficult to show the structure in this book; we recommend exploring a little on your own with `View(gh_repos)` before we continue.

`gh_repos` is a list, but our tools work with list columns, so we'll begin by putting it into a tibble. We call this column `json` for reasons we'll get to later.

```
repos <- tibble(json = gh_repos)
repos
#> # A tibble: 6 × 1
#>   json
#>   <list>
#> 1 <list [30]>
#> 2 <list [30]>
#> 3 <list [30]>
#> 4 <list [26]>
#> 5 <list [30]>
#> 6 <list [30]>
```

This tibble contains six rows, one row for each child of `gh_repos`. Each row contains a unnamed list with either 26 or 30 rows. Since these are unnamed, we'll start with `unnest_longer()` to put each child in its own row:

```
repos |>
  unnest_longer(json)
#> # A tibble: 176 × 1
#>   json
#>   <list>
#> 1 <named list [68]>
#> 2 <named list [68]>
#> 3 <named list [68]>
#> 4 <named list [68]>
#> 5 <named list [68]>
#> 6 <named list [68]>
#> # … with 170 more rows
```

At first glance, it might seem like we haven't improved the situation: while we have more rows (176 instead of 6), each element of `json` is still a list. However, there's an important difference: now each element is a *named* list, so we can use `unnest_wider()` to put each element into its own column:

```
repos |>
  unnest_longer(json) |>
  unnest_wider(json)
#> # A tibble: 176 × 68
#>         id name        full_name         owner        private html_url
#>      <int> <chr>       <chr>             <list>       <lgl>   <chr>
#> 1 61160198 after       gaborcsardi/after <named list> FALSE   https://github…
#> 2 40500181 argufy      gaborcsardi/argu… <named list> FALSE   https://github…
#> 3 36442442 ask         gaborcsardi/ask   <named list> FALSE   https://github…
#> 4 34924886 baseimports gaborcsardi/base… <named list> FALSE   https://github…
#> 5 61620661 citest      gaborcsardi/cite… <named list> FALSE   https://github…
#> 6 33907457 clisymbols  gaborcsardi/clis… <named list> FALSE   https://github…
#> # … with 170 more rows, and 62 more variables: description <chr>,
#> #   fork <lgl>, url <chr>, forks_url <chr>, keys_url <chr>, …
```

This has worked, but the result is a little overwhelming: there are so many columns that tibble doesn't even print all of them! We can see them all with `names()` and here we look at the first 10:

```
repos |>
  unnest_longer(json) |>
  unnest_wider(json) |>
  names() |>
  head(10)
#>  [1] "id"          "name"        "full_name"   "owner"       "private"
#>  [6] "html_url"    "description" "fork"        "url"         "forks_url"
```

Let's pull out a few that look interesting:

```
repos |>
  unnest_longer(json) |>
  unnest_wider(json) |>
  select(id, full_name, owner, description)
#> # A tibble: 176 × 4
#>         id full_name               owner             description
#>      <int> <chr>                   <list>            <chr>
#> 1 61160198 gaborcsardi/after       <named list [17]> Run Code in the Backgro…
#> 2 40500181 gaborcsardi/argufy      <named list [17]> Declarative function ar…
#> 3 36442442 gaborcsardi/ask         <named list [17]> Friendly CLI interactio…
#> 4 34924886 gaborcsardi/baseimports <named list [17]> Do we get warnings for …
#> 5 61620661 gaborcsardi/citest      <named list [17]> Test R package and repo…
#> 6 33907457 gaborcsardi/clisymbols  <named list [17]> Unicode symbols for CLI…
#> # … with 170 more rows
```

You can use this to work back to understand how `gh_repos` was structured: each child was a GitHub user containing a list of up to 30 GitHub repositories that they created.

`owner` is another list column, and since it contains a named list, we can use `unnest_wider()` to get at the values:

```
repos |>
  unnest_longer(json) |>
  unnest_wider(json) |>
  select(id, full_name, owner, description) |>
  unnest_wider(owner)
#> Error in `unnest_wider()`:
#> ! Can't duplicate names between the affected columns and the original
#>   data.
#> ✖ These names are duplicated:
#>   ℹ `id`, from `owner`.
#> ℹ Use `names_sep` to disambiguate using the column name.
#> ℹ Or use `names_repair` to specify a repair strategy.
```

Uh-oh, this list column also contains an `id` column, and we can't have two `id` columns in the same data frame. As suggested, let's use `names_sep` to resolve the problem:

```
repos |>
  unnest_longer(json) |>
  unnest_wider(json) |>
  select(id, full_name, owner, description) |>
  unnest_wider(owner, names_sep = "_")
#> # A tibble: 176 × 20
#>         id full_name               owner_login owner_id owner_avatar_url
#>      <int> <chr>                   <chr>          <int> <chr>
#> 1 61160198 gaborcsardi/after       gaborcsardi   660288 https://avatars.gith…
#> 2 40500181 gaborcsardi/argufy      gaborcsardi   660288 https://avatars.gith…
#> 3 36442442 gaborcsardi/ask         gaborcsardi   660288 https://avatars.gith…
#> 4 34924886 gaborcsardi/baseimports gaborcsardi   660288 https://avatars.gith…
#> 5 61620661 gaborcsardi/citest      gaborcsardi   660288 https://avatars.gith…
#> 6 33907457 gaborcsardi/clisymbols  gaborcsardi   660288 https://avatars.gith…
#> # … with 170 more rows, and 15 more variables: owner_gravatar_id <chr>,
#> #   owner_url <chr>, owner_html_url <chr>, owner_followers_url <chr>, …
```

This gives another wide dataset, but you can get the sense that `owner` appears to contain a lot of additional data about the person who "owns" the repository.

Relational Data

Nested data is sometimes used to represent data that we'd usually spread across multiple data frames. For example, take `got_chars`, which contains data about characters that appear in the *Game of Thrones* books and TV series. Like `gh_repos`, it's a list, so we start by turning it into a list column of a tibble:

```
chars <- tibble(json = got_chars)
chars
#> # A tibble: 30 × 1
#>   json
#>   <list>
#> 1 <named list [18]>
#> 2 <named list [18]>
#> 3 <named list [18]>
#> 4 <named list [18]>
#> 5 <named list [18]>
#> 6 <named list [18]>
#> # … with 24 more rows
```

The json column contains named elements, so we'll start by widening it:

```
chars |>
  unnest_wider(json)
#> # A tibble: 30 × 18
#>   url                    id name              gender culture    born
#>   <chr>               <int> <chr>             <chr>  <chr>      <chr>
#> 1 https://www.anapio…  1022 Theon Greyjoy     Male   "Ironborn" "In 278 AC or …
#> 2 https://www.anapio…  1052 Tyrion Lannist… Male   ""         "In 273 AC, at…
#> 3 https://www.anapio…  1074 Victarion Grey… Male   "Ironborn" "In 268 AC or …
#> 4 https://www.anapio…  1109 Will              Male   ""         ""
#> 5 https://www.anapio…  1166 Areo Hotah        Male   "Norvoshi" "In 257 AC or …
#> 6 https://www.anapio…  1267 Chett             Male   ""         "At Hag's Mire"
#> # … with 24 more rows, and 12 more variables: died <chr>, alive <lgl>,
#> #   titles <list>, aliases <list>, father <chr>, mother <chr>, …
```

Then we select a few columns to make it easier to read:

```
characters <- chars |>
  unnest_wider(json) |>
  select(id, name, gender, culture, born, died, alive)
characters
#> # A tibble: 30 × 7
#>      id name              gender culture    born              died
#>   <int> <chr>             <chr>  <chr>      <chr>             <chr>
#> 1  1022 Theon Greyjoy     Male   "Ironborn" "In 278 AC or 27… ""
#> 2  1052 Tyrion Lannister  Male   ""         "In 273 AC, at C… ""
#> 3  1074 Victarion Greyjoy Male   "Ironborn" "In 268 AC or be… ""
#> 4  1109 Will              Male   ""         ""                "In 297 AC, at…
#> 5  1166 Areo Hotah        Male   "Norvoshi" "In 257 AC or be… ""
#> 6  1267 Chett             Male   ""         "At Hag's Mire"   "In 299 AC, at…
#> # … with 24 more rows, and 1 more variable: alive <lgl>
```

This dataset also contains many list columns:

```
chars |>
  unnest_wider(json) |>
  select(id, where(is.list))
#> # A tibble: 30 × 8
#>      id titles    aliases     allegiances books     povBooks tvSeries playedBy
#>   <int> <list>    <list>      <list>      <list>    <list>   <list>   <list>
#> 1  1022 <chr [2]> <chr [4]>  <chr [1]>   <chr [3]> <chr>    <chr>    <chr>
#> 2  1052 <chr [2]> <chr [11]> <chr [1]>   <chr [2]> <chr>    <chr>    <chr>
#> 3  1074 <chr [2]> <chr [1]>  <chr [1]>   <chr [3]> <chr>    <chr>    <chr>
#> 4  1109 <chr [1]> <chr [1]>  <NULL>      <chr [1]> <chr>    <chr>    <chr>
#> 5  1166 <chr [1]> <chr [1]>  <chr [1]>   <chr [3]> <chr>    <chr>    <chr>
#> 6  1267 <chr [1]> <chr [1]>  <NULL>      <chr [2]> <chr>    <chr>    <chr>
#> # … with 24 more rows
```

Let's explore the titles column. It's an unnamed list column, so we'll unnest it into rows:

```
chars |>
  unnest_wider(json) |>
  select(id, titles) |>
  unnest_longer(titles)
#> # A tibble: 59 × 2
#>      id titles
#>   <int> <chr>
#> 1  1022 Prince of Winterfell
#> 2  1022 Lord of the Iron Islands (by law of the green lands)
#> 3  1052 Acting Hand of the King (former)
#> 4  1052 Master of Coin (former)
#> 5  1074 Lord Captain of the Iron Fleet
#> 6  1074 Master of the Iron Victory
#> # … with 53 more rows
```

You might expect to see this data in its own table because it would be easy to join to the characters data as needed. Let's do that, which requires a little cleaning: removing the rows containing empty strings and renaming `titles` to `title` since each row now contains only a single title.

```
titles <- chars |>
  unnest_wider(json) |>
  select(id, titles) |>
  unnest_longer(titles) |>
  filter(titles != "") |>
  rename(title = titles)
titles
#> # A tibble: 52 × 2
#>      id title
#>   <int> <chr>
#> 1  1022 Prince of Winterfell
#> 2  1022 Lord of the Iron Islands (by law of the green lands)
#> 3  1052 Acting Hand of the King (former)
#> 4  1052 Master of Coin (former)
#> 5  1074 Lord Captain of the Iron Fleet
#> 6  1074 Master of the Iron Victory
#> # … with 46 more rows
```

You could imagine creating a table like this for each of the list columns and then using joins to combine them with the character data as you need it.

Deeply Nested

We'll finish off these case studies with a list column that's very deeply nested and requires repeated rounds of `unnest_wider()` and `unnest_longer()` to unravel: `gmaps_cities`. This is a two-column tibble containing five city names and the results of using Google's geocoding API to determine their location:

```
gmaps_cities
#> # A tibble: 5 × 2
#>   city       json
#>   <chr>      <list>
#> 1 Houston    <named list [2]>
#> 2 Washington <named list [2]>
#> 3 New York   <named list [2]>
#> 4 Chicago    <named list [2]>
#> 5 Arlington  <named list [2]>
```

`json` is a list column with internal names, so we start with an `unnest_wider()`:

```
gmaps_cities |>
  unnest_wider(json)
#> # A tibble: 5 × 3
#>   city       results    status
#>   <chr>      <list>     <chr>
#> 1 Houston    <list [1]> OK
#> 2 Washington <list [2]> OK
#> 3 New York   <list [1]> OK
#> 4 Chicago    <list [1]> OK
#> 5 Arlington  <list [2]> OK
```

This gives us the `status` and the `results`. We'll drop the status column since they're all `OK`; in a real analysis, you'd also want to capture all the rows where `status != "OK"` and figure out what went wrong. `results` is an unnamed list, with either one or two elements (we'll see why shortly), so we'll unnest it into rows:

```
gmaps_cities |>
  unnest_wider(json) |>
  select(-status) |>
  unnest_longer(results)
#> # A tibble: 7 × 2
#>   city       results
#>   <chr>      <list>
#> 1 Houston    <named list [5]>
#> 2 Washington <named list [5]>
#> 3 Washington <named list [5]>
#> 4 New York   <named list [5]>
#> 5 Chicago    <named list [5]>
#> 6 Arlington  <named list [5]>
#> # … with 1 more row
```

Now `results` is a named list, so we'll use `unnest_wider()`:

```
locations <- gmaps_cities |>
  unnest_wider(json) |>
  select(-status) |>
  unnest_longer(results) |>
  unnest_wider(results)
locations
#> # A tibble: 7 × 6
#>   city       address_compone…¹ formatted_address geometry     place_id
#>   <chr>      <list>            <chr>             <list>       <chr>
#> 1 Houston    <list [4]>        Houston, TX, USA  <named list> ChIJAYWNSLS4QI…
#> 2 Washington <list [2]>        Washington, USA   <named list> ChIJ-bDD5__lhV…
#> 3 Washington <list [4]>        Washington, DC, … <named list> ChIJW-T2Wt7Gt4…
#> 4 New York   <list [3]>        New York, NY, USA <named list> ChIJOwg_06VPwo…
#> 5 Chicago    <list [4]>        Chicago, IL, USA  <named list> ChIJ7cv00DwsDo…
#> 6 Arlington  <list [4]>        Arlington, TX, U… <named list> ChIJ05gI5NJiTo…
#> # … with 1 more row, 1 more variable: types <list>, and abbreviated variable
#> #   name ¹address_components
```

Now we can see why two cities got two results: Washington matched both Washington state and Washington, DC, and Arlington matched Arlington, Virginia, and Arlington, Texas.

There are a few different places we could go from here. We might want to determine the exact location of the match, which is stored in the `geometry` list column:

```
locations |>
  select(city, formatted_address, geometry) |>
  unnest_wider(geometry)
```

```
#> # A tibble: 7 × 6
#>   city       formatted_address   bounds           location     location_type
#>   <chr>      <chr>               <list>           <list>       <chr>
#> 1 Houston    Houston, TX, USA    <named list [2]> <named list> APPROXIMATE
#> 2 Washington Washington, USA     <named list [2]> <named list> APPROXIMATE
#> 3 Washington Washington, DC, USA <named list [2]> <named list> APPROXIMATE
#> 4 New York   New York, NY, USA   <named list [2]> <named list> APPROXIMATE
#> 5 Chicago    Chicago, IL, USA    <named list [2]> <named list> APPROXIMATE
#> 6 Arlington  Arlington, TX, USA  <named list [2]> <named list> APPROXIMATE
#> # … with 1 more row, and 1 more variable: viewport <list>
```

That gives us new `bounds` (a rectangular region) and `location` (a point). We can unnest `location` to see the latitude (`lat`) and longitude (`lng`):

```
locations |>
  select(city, formatted_address, geometry) |>
  unnest_wider(geometry) |>
  unnest_wider(location)
#> # A tibble: 7 × 7
#>   city       formatted_address   bounds             lat    lng location_type
#>   <chr>      <chr>               <list>           <dbl>  <dbl> <chr>
#> 1 Houston    Houston, TX, USA    <named list [2]>  29.8  -95.4 APPROXIMATE
#> 2 Washington Washington, USA     <named list [2]>  47.8 -121.  APPROXIMATE
#> 3 Washington Washington, DC, USA <named list [2]>  38.9  -77.0 APPROXIMATE
#> 4 New York   New York, NY, USA   <named list [2]>  40.7  -74.0 APPROXIMATE
#> 5 Chicago    Chicago, IL, USA    <named list [2]>  41.9  -87.6 APPROXIMATE
#> 6 Arlington  Arlington, TX, USA  <named list [2]>  32.7  -97.1 APPROXIMATE
#> # … with 1 more row, and 1 more variable: viewport <list>
```

Extracting the bounds requires a few more steps:

```
locations |>
  select(city, formatted_address, geometry) |>
  unnest_wider(geometry) |>
  # focus on the variables of interest
  select(!location:viewport) |>
  unnest_wider(bounds)
#> # A tibble: 7 × 4
#>   city       formatted_address   northeast        southwest
#>   <chr>      <chr>               <list>           <list>
#> 1 Houston    Houston, TX, USA    <named list [2]> <named list [2]>
#> 2 Washington Washington, USA     <named list [2]> <named list [2]>
#> 3 Washington Washington, DC, USA <named list [2]> <named list [2]>
#> 4 New York   New York, NY, USA   <named list [2]> <named list [2]>
#> 5 Chicago    Chicago, IL, USA    <named list [2]> <named list [2]>
#> 6 Arlington  Arlington, TX, USA  <named list [2]> <named list [2]>
#> # … with 1 more row
```

We then rename `southwest` and `northeast` (the corners of the rectangle) so we can use `names_sep` to create short but evocative names:

```
locations |>
  select(city, formatted_address, geometry) |>
  unnest_wider(geometry) |>
  select(!location:viewport) |>
  unnest_wider(bounds) |>
  rename(ne = northeast, sw = southwest) |>
  unnest_wider(c(ne, sw), names_sep = "_")
#> # A tibble: 7 × 6
#>   city       formatted_address    ne_lat ne_lng sw_lat sw_lng
#>   <chr>      <chr>                 <dbl>  <dbl>  <dbl>  <dbl>
#> 1 Houston    Houston, TX, USA       30.1  -95.0   29.5  -95.8
#> 2 Washington Washington, USA        49.0 -117.    45.5 -125.
#> 3 Washington Washington, DC, USA    39.0  -76.9   38.8  -77.1
#> 4 New York   New York, NY, USA      40.9  -73.7   40.5  -74.3
#> 5 Chicago    Chicago, IL, USA       42.0  -87.5   41.6  -87.9
#> 6 Arlington  Arlington, TX, USA     32.8  -97.0   32.6  -97.2
#> # … with 1 more row
```

Note how we unnest two columns simultaneously by supplying a vector of variable names to `unnest_wider()`.

Once you've discovered the path to get to the components you're interested in, you can extract them directly using another tidyr function, `hoist()`:

```
locations |>
  select(city, formatted_address, geometry) |>
  hoist(
    geometry,
    ne_lat = c("bounds", "northeast", "lat"),
    sw_lat = c("bounds", "southwest", "lat"),
    ne_lng = c("bounds", "northeast", "lng"),
    sw_lng = c("bounds", "southwest", "lng"),
  )
```

If these case studies have whetted your appetite for more real-life rectangling, you can see a few more examples in `vignette("rectangling", package = "tidyr")`.

Exercises

1. Roughly estimate when `gh_repos` was created. Why can you only roughly estimate the date?

2. The `owner` column of `gh_repo` contains a lot of duplicated information because each owner can have many repos. Can you construct an `owners` data frame that contains one row for each owner? (Hint: Does `distinct()` work with `list-cols`?)

3. Follow the steps used for `titles` to create similar tables for the aliases, allegiances, books, and TV series for the *Game of Thrones* characters.

4. Explain the following code line by line. Why is it interesting? Why does it work for `got_chars` but might not work in general?

```
tibble(json = got_chars) |>
  unnest_wider(json) |>
  select(id, where(is.list)) |>
  pivot_longer(
    where(is.list),
    names_to = "name",
    values_to = "value"
  ) |>
  unnest_longer(value)
```

5. In gmaps_cities, what does address_components contain? Why does the length vary between rows? Unnest it appropriately to figure it out. (Hint: types always appears to contain two elements. Does unnest_wider() make it easier to work with than unnest_longer()?)

JSON

All of the case studies in the previous section were sourced from wild-caught JSON. JSON is short for JavaScript Object Notation and is the way that most web APIs return data. It's important to understand it because while JSON and R's data types are pretty similar, there isn't a perfect one-to-one mapping, so it's good to understand a bit about JSON if things go wrong.

Data Types

JSON is a simple format designed to be easily read and written by machines, not humans. It has six key data types. Four of them are scalars:

- The simplest type is a null (null), which plays the same role as NA in R. It represents the absence of data.
- A *string* is much like a string in R but must always use double quotes.
- A *number* is similar to R's numbers: they can use integer (e.g., 123), decimal (e.g., 123.45), or scientific (e.g., 1.23e3) notation. JSON doesn't support Inf, -Inf, or NaN.
- A *boolean* is similar to R's TRUE and FALSE but uses lowercase true and false.

JSON's strings, numbers, and Booleans are pretty similar to R's character, numeric, and logical vectors. The main difference is that JSON's scalars can represent only a single value. To represent multiple values you need to use one of the two remaining types: arrays and objects.

Both arrays and objects are similar to lists in R; the difference is whether they're named. An *array* is like an unnamed list and is written with []. For example, [1, 2, 3] is an array containing three numbers, and [null, 1, "string", false] is an array that contains a null, a number, a string, and a Boolean. An *object* is like a named list and is written with {}. The names (keys in JSON terminology) are strings, so they must be surrounded by quotes. For example, {"x": 1, "y": 2} is an object that maps x to 1 and y to 2.

Note that JSON doesn't have any native way to represent dates or date-times, so they're often stored as strings, and you'll need to use readr::parse_date() or readr::parse_datetime() to turn them into the correct data structure. Similarly, JSON's rules for representing floating-point numbers in JSON are a little imprecise, so you'll also sometimes find numbers stored in strings. Apply readr::parse_double() as needed to get the correct variable type.

jsonlite

To convert JSON into R data structures, we recommend the jsonlite package, by Jeroen Ooms. We'll use only two jsonlite functions: read_json() and parse_json(). In real life, you'll use read_json() to read a JSON file from disk. For example, the repurrsive package also provides the source for gh_user as a JSON file, and you can read it with read_json():

```
# A path to a json file inside the package:
gh_users_json()
#> [1] "/Users/hadley/Library/R/arm64/4.2/library/repurrrsive/extdata/gh_users.json"

# Read it with read_json()
gh_users2 <- read_json(gh_users_json())

# Check it's the same as the data we were using previously
identical(gh_users, gh_users2)
#> [1] TRUE
```

In this book, we'll also use parse_json(), since it takes a string containing JSON, which makes it good for generating simple examples. To get started, here are three simple JSON datasets, starting with a number, then putting a few numbers in an array, and then putting that array in an object:

```
str(parse_json('1'))
#>  int 1
str(parse_json('[1, 2, 3]'))
#> List of 3
#>  $ : int 1
#>  $ : int 2
#>  $ : int 3
str(parse_json('{"x": [1, 2, 3]}'))
#> List of 1
#>  $ x:List of 3
#>   ..$ : int 1
#>   ..$ : int 2
#>   ..$ : int 3
```

jsonlite has another important function called fromJSON(). We don't use it here because it performs automatic simplification (simplifyVector = TRUE). This often works well, particularly in simple cases, but we think you're better off doing the rectangling yourself so you know exactly what's happening and can more easily handle the most complicated nested structures.

Starting the Rectangling Process

In most cases, JSON files contain a single top-level array, because they're designed to provide data about multiple "things," e.g., multiple pages, multiple records, or multiple results. In this case, you'll start your rectangling with `tibble(json)` so that each element becomes a row:

```
json <- '[
  {"name": "John", "age": 34},
  {"name": "Susan", "age": 27}
]'
df <- tibble(json = parse_json(json))
df
#> # A tibble: 2 × 1
#>   json
#>   <list>
#> 1 <named list [2]>
#> 2 <named list [2]>

df |>
  unnest_wider(json)
#> # A tibble: 2 × 2
#>   name    age
#>   <chr> <int>
#> 1 John     34
#> 2 Susan    27
```

In rarer cases, the JSON file consists of a single top-level JSON object, representing one "thing." In this case, you'll need to kick off the rectangling process by wrapping it in a list, before you put it in a tibble:

```
json <- '{
  "status": "OK",
  "results": [
    {"name": "John", "age": 34},
    {"name": "Susan", "age": 27}
 ]
}
'
df <- tibble(json = list(parse_json(json)))
df
#> # A tibble: 1 × 1
#>   json
#>   <list>
#> 1 <named list [2]>

df |>
  unnest_wider(json) |>
  unnest_longer(results) |>
  unnest_wider(results)
#> # A tibble: 2 × 3
#>   status name    age
#>   <chr>  <chr> <int>
#> 1 OK     John     34
#> 2 OK     Susan    27
```

Alternatively, you can reach inside the parsed JSON and start with the bit that you actually care about:

```
df <- tibble(results = parse_json(json)$results)
df |>
  unnest_wider(results)
#> # A tibble: 2 × 2
#>   name    age
#>   <chr> <int>
#> 1 John     34
#> 2 Susan    27
```

Exercises

1. Rectangle the following `df_col` and `df_row`. They represent the two ways of encoding a data frame in JSON.

```
json_col <- parse_json('
  {
    "x": ["a", "x", "z"],
    "y": [10, null, 3]
  }
')
json_row <- parse_json('
  [
    {"x": "a", "y": 10},
    {"x": "x", "y": null},
    {"x": "z", "y": 3}
  ]
')

df_col <- tibble(json = list(json_col))
df_row <- tibble(json = json_row)
```

Summary

In this chapter, you learned what lists are, how you can generate them from JSON files, and how to turn them into rectangular data frames. Surprisingly we need only two new functions: `unnest_longer()` to put list elements into rows and `unnest_wider()` to put list elements into columns. It doesn't matter how deeply nested the list column is; all you need to do is repeatedly call these two functions.

JSON is the most common data format returned by web APIs. What happens if the website doesn't have an API but you can see data you want on the website? That's the topic of the next chapter: web scraping, extracting data from HTML web pages.

CHAPTER 24

Web Scraping

Introduction

This chapter introduces you to the basics of web scraping with rvest. Web scraping is a useful tool for extracting data from web pages. Some websites will offer an API, a set of structured HTTP requests that return data as JSON, which you handle using the techniques from Chapter 23. Where possible, you should use the API,[1] because typically it will give you more reliable data. Unfortunately, however, programming with web APIs is out of scope for this book. Instead, we are teaching scraping, a technique that works whether or not a site provides an API.

In this chapter, we'll first discuss the ethics and legalities of scraping before we dive into the basics of HTML. You'll then learn the basics of CSS selectors to locate specific elements on the page and how to use rvest functions to get data from text and attributes out of HTML and into R. We'll then discuss some techniques to figure out what CSS selector you need for the page you're scraping, before finishing up with a couple of case studies and a brief discussion of dynamic websites.

Prerequisites

In this chapter, we'll focus on tools provided by rvest. rvest is a member of the tidyverse but is not a core member, so you'll need to load it explicitly. We'll also load the full tidyverse since we'll find it generally useful working with the data we've scraped.

```
library(tidyverse)
library(rvest)
```

[1] Many popular APIs already have CRAN packages that wrap them, so start with a little research first!

Scraping Ethics and Legalities

Before we get started discussing the code you'll need to perform web scraping, we need to talk about whether it's legal and ethical for you to do so. Overall, the situation is complicated with regard to both of these.

Legalities depend a lot on where you live. However, as a general principle, if the data is public, nonpersonal, and factual, you're likely to be OK.[2] These three factors are important because they're connected to the site's terms and conditions, personally identifiable information, and copyright, as we'll discuss.

If the data isn't public, nonpersonal, or factual or if you're scraping the data specifically to make money with it, you'll need to talk to a lawyer. In any case, you should be respectful of the resources of the server hosting the pages you are scraping. Most important, this means that if you're scraping many pages, you should make sure to wait a little between each request. One easy way to do so is to use the polite package by Dmytro Perepolkin. It will automatically pause between requests and cache the results so you never ask for the same page twice.

Terms of Service

If you look closely, you'll find many websites include a "terms and conditions" or "terms of service" link somewhere on the page, and if you read that page closely, you'll often discover that the site specifically prohibits web scraping. These pages tend to be a legal land grab where companies make very broad claims. It's polite to respect these terms of service where possible, but take any claims with a grain of salt.

US courts have generally found that simply putting the terms of service in the footer of the website isn't sufficient for you to be bound by them, e.g., *HiQ Labs v. LinkedIn*. Generally, to be bound to the terms of service, you must have taken some explicit action such as creating an account or checking a box. This is why whether or not the data is *public* is important; if you don't need an account to access them, it is unlikely that you are bound to the terms of service. Note, however, the situation is rather different in Europe where courts have found that terms of service are enforceable even if you don't explicitly agree to them.

Personally Identifiable Information

Even if the data is public, you should be extremely careful about scraping personally identifiable information such as names, email addresses, phone numbers, dates of birth, etc. Europe has particularly strict laws about the collection of storage of

[2] Obviously we're not lawyers, and this is not legal advice. But this is the best summary we can give having read a bunch about this topic.

such data (GDPR), and regardless of where you live, you're likely to be entering an ethical quagmire. For example, in 2016, a group of researchers scraped public profile information (e.g., username, age, gender, location, etc.) about 70,000 people on the dating site OkCupid and publicly released these data without any attempts for anonymization. While the researchers felt that there was nothing wrong with this since the data were already public, this work was widely condemned due to ethics concerns around identifiability of users whose information was released in the dataset. If your work involves scraping personally identifiable information, we strongly recommend reading about the OkCupid study[3] as well as similar studies with questionable research ethics involving the acquisition and release of personally identifiable information.

Copyright

Finally, you also need to worry about copyright law. Copyright law is complicated, but it's worth taking a look at the US law, which describes exactly what's protected: "[…] original works of authorship fixed in any tangible medium of expression, […]." It then goes on to describe specific categories that it applies to such as literary works, musical works, motion pictures, and more. Notably absent from copyright protection are data. This means that as long as you limit your scraping to facts, copyright protection does not apply. (But note that Europe has a separate "sui generis" right that protects databases.)

As a brief example, in the US, lists of ingredients and instructions are not copyrightable, so copyright cannot be used to protect a recipe. But if that list of recipes is accompanied by substantial novel literary content, that is copyrightable. This is why when you're looking for a recipe on the internet, there's always so much content beforehand.

If you do need to scrape original content (like text or images), you may still be protected under the doctrine of fair use. Fair use is not a hard and fast rule but weighs up a number of factors. It's more likely to apply if you are collecting the data for research or noncommercial purposes and if you limit what you scrape to just what you need.

HTML Basics

To scrape web pages, you need to first understand a little bit about *HTML*, the language that describes web pages. HTML stands for HyperText Markup Language and looks something like this:

[3] One example of an article on the OkCupid study was published by Wired.

```
<html>
<head>
  <title>Page title</title>
</head>
<body>
  <h1 id='first'>A heading</h1>
  <p>Some text &amp; <b>some bold text.</b></p>
  <img src='myimg.png' width='100' height='100'>
</body>
```

HTML has a hierarchical structure formed by *elements*, which consist of a start tag (e.g., `<tag>`), optional *attributes* (`id='first'`), an end tag[4] (like `</tag>`), and *contents* (everything in between the start and end tags).

Since `<` and `>` are used for start and end tags, you can't write them directly. Instead, you have to use the HTML *escapes* `&gt;` (greater than) and `&lt;` (less than). And since those escapes use `&`, if you want a literal ampersand, you have to escape it as `&amp;`. There are a wide range of possible HTML escapes, but you don't need to worry about them too much because rvest automatically handles them for you.

Web scraping is possible because most pages that contain data that you want to scrape generally have a consistent structure.

Elements

There are more than 100 HTML elements. Some of the most important are:

- Every HTML page must be in an `<html>` element, and it must have two children: `<head>`, which contains document metadata like the page title, and `<body>`, which contains the content you see in the browser.
- Block tags like `<h1>` (heading 1), `<section>` (section), `<p>` (paragraph), and `<ol>` (ordered list) form the overall structure of the page.
- Inline tags like `<b>` (bold), `<i>` (italics), and `<a>` (link) format text inside block tags.

If you encounter a tag that you've never seen before, you can find out what it does with a little googling. Another good place to start is the MDN Web Docs, which describe just about every aspect of web programming.

Most elements can have content in between their start and end tags. This content can be either text or more elements. For example, the following HTML contains a paragraph of text, with one word in bold:

[4] A number of tags (including `<p>` and `<li>`) don't require end tags, but we think it's best to include them because it makes seeing the structure of the HTML a little easier.

```
<p>
  Hi! My <b>name</b> is Hadley.
</p>
```

The *children* are the elements it contains, so the previous `<p>` element has one child, the `<b>` element. The `<b>` element has no children, but it does have contents (the text "name").

Attributes

Tags can have named *attributes*, which look like `name1='value1' name2='value2'`. Two of the most important attributes are `id` and `class`, which are used in conjunction with Cascading Style Sheets (CSS) to control the visual appearance of the page. These are often useful when scraping data off a page. Attributes are also used to record the destination of links (the `href` attribute of `<a>` elements) and the source of images (the `src` attribute of the `<img>` element).

Extracting Data

To get started scraping, you'll need the URL of the page you want to scrape, which you can usually copy from your web browser. You'll then need to read the HTML for that page into R with `read_html()`. This returns an `xml_document`[5] object, which you'll then manipulate using rvest functions:

```
html <- read_html("http://rvest.tidyverse.org/")
html
#> {html_document}
#> <html lang="en">
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UT ...
#> [2] <body>\n    <a href="#container" class="visually-hidden-focusable">Ski ...
```

rvest also includes a function that lets you write HTML inline. We'll use this a bunch in this chapter as we teach how the various rvest functions work with simple examples.

```
html <- minimal_html("
  <p>This is a paragraph</p>
  <ul>
    <li>This is a bulleted list</li>
  </ul>
")
html
#> {html_document}
#> <html>
#> [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UT ...
#> [2] <body>\n<p>This is a paragraph</p>\n<p>\n  </p>\n<ul>\n<li>This is a b ...
```

[5] This class comes from the xml2 package. xml2 is a low-level package that rvest builds on top of.

Now that you have the HTML in R, it's time to extract the data of interest. You'll first learn about the CSS selectors that allow you to identify the elements of interest and the rvest functions that you can use to extract data from them. Then we'll briefly cover HTML tables, which have some special tools.

Find Elements

CSS is a tool for defining the visual styling of HTML documents. CSS includes a miniature language for selecting elements on a page called *CSS selectors*. CSS selectors define patterns for locating HTML elements and are useful for scraping because they provide a concise way of describing which elements you want to extract.

We'll come back to CSS selectors in more detail in "Finding the Right Selectors" on page 433, but luckily you can get a long way with just three:

`p`
: Selects all `<p>` elements.

`.title`
: Selects all elements with `class` "title."

`#title`
: Selects the element with the `id` attribute that equals "title." `id` attributes must be unique within a document, so this will only ever select a single element.

Let's try these selectors with a simple example:

```
html <- minimal_html("
  <h1>This is a heading</h1>
  <p id='first'>This is a paragraph</p>
  <p class='important'>This is an important paragraph</p>
")
```

Use `html_elements()` to find all elements that match the selector:

```
html |> html_elements("p")
#> {xml_nodeset (2)}
#> [1] <p id="first">This is a paragraph</p>
#> [2] <p class="important">This is an important paragraph</p>
html |> html_elements(".important")
#> {xml_nodeset (1)}
#> [1] <p class="important">This is an important paragraph</p>
html |> html_elements("#first")
#> {xml_nodeset (1)}
#> [1] <p id="first">This is a paragraph</p>
```

Another important function is `html_element()`, which always returns the same number of outputs as inputs. If you apply it to a whole document, it'll give you the first match:

```
html |> html_element("p")
#> {html_node}
#> <p id="first">
```

There's an important difference between `html_element()` and `html_elements()` when you use a selector that doesn't match any elements. `html_elements()` returns a vector of length 0, where `html_element()` returns a missing value. This will be important shortly.

```
html |> html_elements("b")
#> {xml_nodeset (0)}
html |> html_element("b")
#> {xml_missing}
#> <NA>
```

Nesting Selections

In most cases, you'll use `html_elements()` and `html_element()` together, typically using `html_elements()` to identify elements that will become observations and then using `html_element()` to find elements that will become variables. Let's see this in action using a simple example. Here we have an unordered list (`<ul>`) where each list item (`<li>`) contains some information about four characters from *Star Wars*:

```
html <- minimal_html("
  <ul>
    <li><b>C-3PO</b> is a <i>droid</i> that weighs <span class='weight'>167 kg</span></li>
    <li><b>R4-P17</b> is a <i>droid</i></li>
    <li><b>R2-D2</b> is a <i>droid</i> that weighs <span class='weight'>96 kg</span></li>
    <li><b>Yoda</b> weighs <span class='weight'>66 kg</span></li>
  </ul>
  ")
```

We can use `html_elements()` to make a vector where each element corresponds to a different character:

```
characters <- html |> html_elements("li")
characters
#> {xml_nodeset (4)}
#> [1] <li>\n<b>C-3PO</b> is a <i>droid</i> that weighs <span class="weight"> ...
#> [2] <li>\n<b>R4-P17</b> is a <i>droid</i>\n</li>
#> [3] <li>\n<b>R2-D2</b> is a <i>droid</i> that weighs <span class="weight"> ...
#> [4] <li>\n<b>Yoda</b> weighs <span class="weight">66 kg</span>\n</li>
```

To extract the name of each character, we use `html_element()`, because when applied to the output of `html_elements()`, it's guaranteed to return one response per element:

```
characters |> html_element("b")
#> {xml_nodeset (4)}
#> [1] <b>C-3PO</b>
#> [2] <b>R4-P17</b>
#> [3] <b>R2-D2</b>
#> [4] <b>Yoda</b>
```

The distinction between `html_element()` and `html_elements()` isn't important for the name, but it is important for the weight. We want to get one weight for each character, even if there's no weight `<span>`. That's what `html_element()` does:

```
characters |> html_element(".weight")
#> {xml_nodeset (4)}
#> [1] <span class="weight">167 kg</span>
#> [2] <NA>
#> [3] <span class="weight">96 kg</span>
#> [4] <span class="weight">66 kg</span>
```

`html_elements()` finds all weight `<span>`s that are children of `characters`. There's only three of these, so we lose the connection between names and weights:

```
characters |> html_elements(".weight")
#> {xml_nodeset (3)}
#> [1] <span class="weight">167 kg</span>
#> [2] <span class="weight">96 kg</span>
#> [3] <span class="weight">66 kg</span>
```

Now that you've selected the elements of interest, you'll need to extract the data, either from the text contents or from some attributes.

Text and Attributes

`html_text2()`[6] extracts the plain-text contents of an HTML element:

```
characters |>
  html_element("b") |>
  html_text2()
#> [1] "C-3PO"  "R4-P17" "R2-D2"  "Yoda"

characters |>
  html_element(".weight") |>
  html_text2()
#> [1] "167 kg" NA       "96 kg"  "66 kg"
```

Note that any escapes will be automatically handled; you'll only ever see HTML escapes in the source HTML, not in the data returned by rvest.

`html_attr()` extracts data from attributes:

```
html <- minimal_html("
  <p><a href='https://en.wikipedia.org/wiki/Cat'>cats</a></p>
  <p><a href='https://en.wikipedia.org/wiki/Dog'>dogs</a></p>
")

html |>
  html_elements("p") |>
  html_element("a") |>
  html_attr("href")
#> [1] "https://en.wikipedia.org/wiki/Cat" "https://en.wikipedia.org/wiki/Dog"
```

[6] rvest also provides `html_text()`, but you should almost always use `html_text2()` since it does a better job of converting nested HTML to text.

html_attr() always returns a string, so if you're extracting numbers or dates, you'll need to do some post-processing.

Tables

If you're lucky, your data will be already stored in an HTML table, and it'll be a matter of just reading it from that table. It's usually straightforward to recognize a table in your browser: it'll have a rectangular structure of rows and columns, and you can copy and paste it into a tool like Excel.

HTML tables are built up from four main elements: <table>, <tr> (table row), <th> (table heading), and <td> (table data). Here's a simple HTML table with two columns and three rows:

```
html <- minimal_html("
  <table class='mytable'>
    <tr><th>x</th>   <th>y</th></tr>
    <tr><td>1.5</td> <td>2.7</td></tr>
    <tr><td>4.9</td> <td>1.3</td></tr>
    <tr><td>7.2</td> <td>8.1</td></tr>
  </table>
  ")
```

rvest provides a function that knows how to read this sort of data: html_table(). It returns a list containing one tibble for each table found on the page. Use html_element() to identify the table you want to extract:

```
html |>
  html_element(".mytable") |>
  html_table()
#> # A tibble: 3 × 2
#>       x     y
#>   <dbl> <dbl>
#> 1   1.5   2.7
#> 2   4.9   1.3
#> 3   7.2   8.1
```

Note that x and y have automatically been converted to numbers. This automatic conversion doesn't always work, so in more complex scenarios you may want to turn it off with convert = FALSE and then do your own conversion.

Finding the Right Selectors

Figuring out the selector you need for your data is typically the hardest part of the problem. You'll often need to do some experimenting to find a selector that is both specific (i.e., it doesn't select things you don't care about) and sensitive (i.e., it does select everything you care about). Lots of trial and error is a normal part of the process! Two main tools are available to help you with this process: SelectorGadget and your browser's developer tools.

SelectorGadget is a JavaScript bookmarklet that automatically generates CSS selectors based on the positive and negative examples that you provide. It doesn't always work, but when it does, it's magic! You can learn how to install and use SelectorGadget either by reading the vignette or by watching Mine's video.

Every modern browser comes with some toolkit for developers, but we recommend Chrome, even if it isn't your regular browser: its web developer tools are some of the best, and they're immediately available. Right-click an element on the page and click Inspect. This will open an expandable view of the complete HTML page, centered on the element that you just clicked. You can use this to explore the page and get a sense of what selectors might work. Pay particular attention to the `class` and `id` attributes, since these are often used to form the visual structure of the page and hence make for good tools to extract the data that you're looking for.

Inside the Elements view, you can also right-click an element and choose Copy as Selector to generate a selector that will uniquely identify the element of interest.

If either SelectorGadget or Chrome DevTools has generated a CSS selector that you don't understand, try Selectors Explained, which translates CSS selectors into plain English. If you find yourself doing this a lot, you might want to learn more about CSS selectors generally. We recommend starting with the fun CSS dinner tutorial and then referring to the MDN web docs.

Putting It All Together

Let's put this all together to scrape some websites. There's some risk that these examples may no longer work when you run them—that's the fundamental challenge of web scraping; if the structure of the site changes, then you'll have to change your scraping code.

Star Wars

rvest includes a very simple example in `vignette("starwars")`. This is a simple page with minimal HTML, so it's a good place to start. We encourage you to navigate to that page now and use Inspect Element to inspect one of the headings that's the title of a *Star Wars* movie. Use the keyboard or mouse to explore the hierarchy of the HTML and see if you can get a sense of the shared structure used by each movie.

You should be able to see that each movie has a shared structure that looks like this:

```
<section>
  <h2 data-id="1">The Phantom Menace</h2>
  <p>Released: 1999-05-19</p>
  <p>Director: <span class="director">George Lucas</span></p>

  <div class="crawl">
    <p>...</p>
    <p>...</p>
    <p>...</p>
  </div>
</section>
```

Our goal is to turn this data into a seven-row data frame with the variables `title`, `year`, `director`, and `intro`. We'll start by reading the HTML and extracting all the `<section>` elements:

```
url <- "https://rvest.tidyverse.org/articles/starwars.html"
html <- read_html(url)

section <- html |> html_elements("section")
section
#> {xml_nodeset (7)}
#> [1] <section><h2 data-id="1">\nThe Phantom Menace\n</h2>\n<p>\nReleased: 1 ...
#> [2] <section><h2 data-id="2">\nAttack of the Clones\n</h2>\n<p>\nReleased: ...
#> [3] <section><h2 data-id="3">\nRevenge of the Sith\n</h2>\n<p>\nReleased:  ...
#> [4] <section><h2 data-id="4">\nA New Hope\n</h2>\n<p>\nReleased: 1977-05-2 ...
#> [5] <section><h2 data-id="5">\nThe Empire Strikes Back\n</h2>\n<p>\nReleas ...
#> [6] <section><h2 data-id="6">\nReturn of the Jedi\n</h2>\n<p>\nReleased: 1 ...
#> [7] <section><h2 data-id="7">\nThe Force Awakens\n</h2>\n<p>\nReleased: 20 ...
```

This retrieves seven elements matching the seven movies found on that page, suggesting that using `section` as a selector is good. Extracting the individual elements is straightforward since the data is always found in the text. It's just a matter of finding the right selector:

```
section |> html_element("h2") |> html_text2()
#> [1] "The Phantom Menace"      "Attack of the Clones"
#> [3] "Revenge of the Sith"     "A New Hope"
#> [5] "The Empire Strikes Back" "Return of the Jedi"
#> [7] "The Force Awakens"

section |> html_element(".director") |> html_text2()
#> [1] "George Lucas"     "George Lucas"     "George Lucas"
#> [4] "George Lucas"     "Irvin Kershner"   "Richard Marquand"
#> [7] "J. J. Abrams"
```

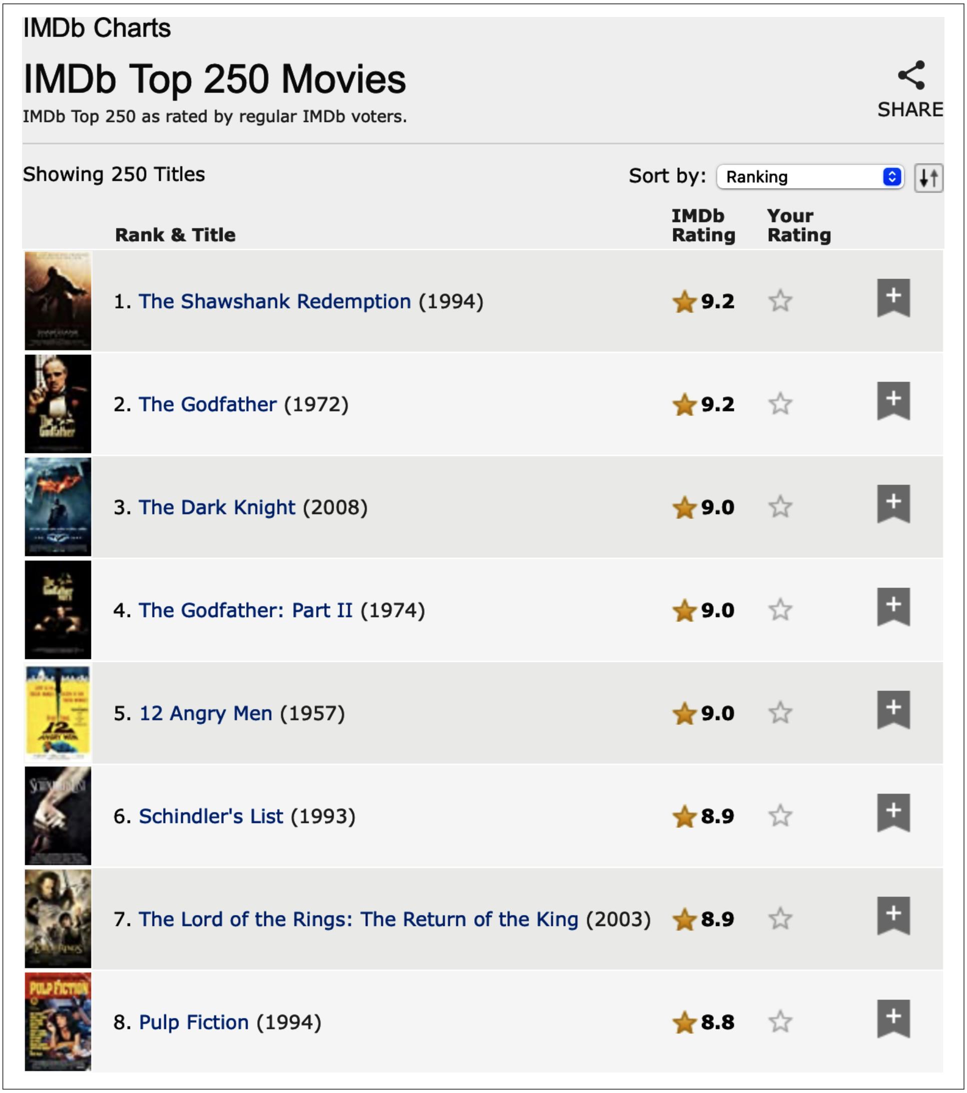
Once we've done that for each component, we can wrap up all the results into a tibble:

```
tibble(
  title = section |>
    html_element("h2") |>
    html_text2(),
  released = section |>
    html_element("p") |>
    html_text2() |>
    str_remove("Released: ") |>
    parse_date(),
  director = section |>
    html_element(".director") |>
    html_text2(),
  intro = section |>
    html_element(".crawl") |>
    html_text2()
)
#> # A tibble: 7 × 4
#>   title                   released   director         intro
#>   <chr>                   <date>     <chr>            <chr>
#> 1 The Phantom Menace      1999-05-19 George Lucas     "Turmoil has engulfed …
#> 2 Attack of the Clones    2002-05-16 George Lucas     "There is unrest in th…
#> 3 Revenge of the Sith     2005-05-19 George Lucas     "War! The Republic is …
#> 4 A New Hope              1977-05-25 George Lucas     "It is a period of civ…
#> 5 The Empire Strikes Back 1980-05-17 Irvin Kershner   "It is a dark time for…
#> 6 Return of the Jedi      1983-05-25 Richard Marquand "Luke Skywalker has re…
#> # … with 1 more row
```

We did a little more processing of `released` to get a variable that will be easy to use later in our analysis.

IMDb Top Films

For our next task we'll tackle something a little trickier, extracting the top 250 movies from IMDb. At the time we wrote this chapter, the page looked like Figure 24-1.



IMDb Charts

IMDb Top 250 Movies

IMDb Top 250 as rated by regular IMDb voters.

SHARE

Showing 250 Titles

Sort by: Ranking

Rank & Title	IMDb Rating	Your Rating
1. The Shawshank Redemption (1994)	★9.2	☆
2. The Godfather (1972)	★9.2	☆
3. The Dark Knight (2008)	★9.0	☆
4. The Godfather: Part II (1974)	★9.0	☆
5. 12 Angry Men (1957)	★9.0	☆
6. Schindler's List (1993)	★8.9	☆
7. The Lord of the Rings: The Return of the King (2003)	★8.9	☆
8. Pulp Fiction (1994)	★8.8	☆

Figure 24-1. IMDb top movies web page taken on 2022-12-05.

This data has a clear tabular structure, so it's worth starting with `html_table()`:

```
url <- "https://www.imdb.com/chart/top"
html <- read_html(url)

table <- html |>
  html_element("table") |>
  html_table()
table
#> # A tibble: 250 × 5
#>    ``    `Rank & Title`                    `IMDb Rating` `Your Rating`   ``
#>   <lgl> <chr>                                   <dbl> <chr>            <lgl>
#> 1 NA    "1.\n      The Shawshank Redempt…          9.2 "12345678910\n… NA
#> 2 NA    "2.\n      The Godfather\n       …          9.2 "12345678910\n… NA
#> 3 NA    "3.\n      The Dark Knight\n     …          9   "12345678910\n… NA
#> 4 NA    "4.\n      The Godfather Part II…          9   "12345678910\n… NA
#> 5 NA    "5.\n      12 Angry Men\n        …          9   "12345678910\n… NA
#> 6 NA    "6.\n      Schindler's List\n    …          8.9 "12345678910\n… NA
#> # … with 244 more rows
```

This includes a few empty columns but overall does a good job of capturing the information from the table. However, we need to do some more processing to make it easier to use. First, we'll rename the columns to be easier to work with and remove the extraneous whitespace in rank and title. We will do this with `select()` (instead of `rename()`) to do the renaming and selecting of just these two columns in one step. Then we'll remove the new lines and extra spaces and then apply `separate_wider_regex()` (from "Extract Variables" on page 267) to pull out the title, year, and rank into their own variables.

```
ratings <- table |>
  select(
    rank_title_year = `Rank & Title`,
    rating = `IMDb Rating`
  ) |>
  mutate(
    rank_title_year = str_replace_all(rank_title_year, "\n +", " ")
  ) |>
  separate_wider_regex(
    rank_title_year,
    patterns = c(
      rank = "\\d+", "\\. ",
      title = ".+", " +\\(",
      year = "\\d+", "\\)"
    )
  )
ratings
#> # A tibble: 250 × 4
#>   rank  title                    year  rating
#>   <chr> <chr>                    <chr>  <dbl>
#> 1 1     The Shawshank Redemption 1994     9.2
#> 2 2     The Godfather            1972     9.2
#> 3 3     The Dark Knight          2008     9  
#> 4 4     The Godfather Part II    1974     9  
#> 5 5     12 Angry Men             1957     9  
#> 6 6     Schindler's List         1993     8.9
#> # … with 244 more rows
```

Even in this case where most of the data comes from table cells, it's still worth looking at the raw HTML. If you do so, you'll discover that we can add a little extra data by using one of the attributes. This is one of the reasons it's worth spending a little time spelunking the source of the page; you might find extra data or a parsing route that's slightly easier.

```
html |>
  html_elements("td strong") |>
  head() |>
  html_attr("title")
#> [1] "9.2 based on 2,712,990 user ratings"
#> [2] "9.2 based on 1,884,423 user ratings"
#> [3] "9.0 based on 2,685,826 user ratings"
#> [4] "9.0 based on 1,286,204 user ratings"
#> [5] "9.0 based on 801,579 user ratings"
#> [6] "8.9 based on 1,370,458 user ratings"
```

We can combine this with the tabular data and again apply `separate_wider_regex()` to extract the bit of data we care about:

```
ratings |>
  mutate(
    rating_n = html |> html_elements("td strong") |> html_attr("title")
  ) |>
  separate_wider_regex(
    rating_n,
    patterns = c(
      "[0-9.]+ based on ",
      number = "[0-9,]+",
      " user ratings"
    )
  ) |>
  mutate(
    number = parse_number(number)
  )
#> # A tibble: 250 × 5
#>   rank  title                    year  rating  number
#>   <chr> <chr>                    <chr>  <dbl>   <dbl>
#> 1 1     The Shawshank Redemption 1994     9.2 2712990
#> 2 2     The Godfather            1972     9.2 1884423
#> 3 3     The Dark Knight          2008     9   2685826
#> 4 4     The Godfather Part II    1974     9   1286204
#> 5 5     12 Angry Men             1957     9    801579
#> 6 6     Schindler's List         1993     8.9 1370458
#> # … with 244 more rows
```

Dynamic Sites

So far we focused on websites where `html_elements()` returns what you see in the browser and discussed how to parse what it returns and how to organize that information in tidy data frames. From time to time, however, you'll hit a site where `html_elements()` and friends don't return anything like what you see in the browser. In many cases, that's because you're trying to scrape a website that dynamically generates the content of the page with JavaScript. This doesn't currently work with rvest, because rvest downloads the raw HTML and doesn't run any JavaScript.

It's still possible to scrape these types of sites, but rvest needs to use a more expensive process: fully simulating the web browser including running all JavaScript. This functionality is not available at the time of writing, but it's something we're actively working on and might be available by the time you read this. It uses the chromote package, which actually runs the Chrome browser in the background, and gives you additional tools to interact with the site, like a human typing text and clicking buttons. Check out the rvest website for more details.

Summary

In this chapter, you learned about the why, the why not, and the how of scraping data from web pages. First, you learned about the basics of HTML and using CSS selectors to refer to specific elements, and then you learned about using the rvest package to get data out of HTML into R. We then demonstrated web scraping with two case studies: a simpler scenario on scraping data on *Star Wars* films from the rvest package website and a more complex scenario on scraping the top 250 films from IMDb.

Technical details of scraping data off the web can be complex, particularly when dealing with sites; however, legal and ethical considerations can be even more complex. It's important for you to educate yourself about both of these before setting out to scrape data.

This brings us to the end of the import part of the book where you've learned techniques to get data from where it lives (spreadsheets, databases, JSON files, and websites) into a tidy form in R. Now it's time to turn our sights to a new topic: making the most of R as a programming language.

PART V
Program

In this part of the book, you'll improve your programming skills. Programming is a cross-cutting skill needed for all data science work: you must use a computer to do data science; you cannot do it in your head or with pencil and paper.

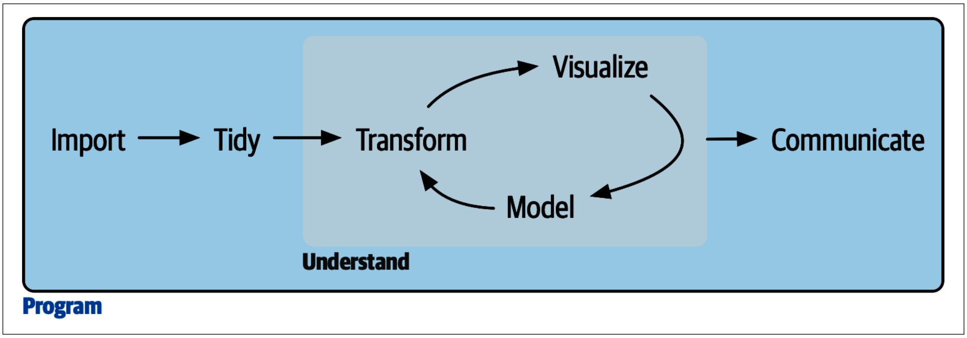


Figure V-1. Programming is the water in which all the other components swim.

Programming produces code, and code is a tool of communication. Obviously code tells the computer what you want it to do. But it also communicates meaning to other humans. Thinking about code as a vehicle for communication is important because every project you do is fundamentally collaborative. Even if you're not working with other people, you'll definitely be working with future-you! Writing clear code is important so that others (like future-you) can understand why you tackled an analysis in the way you did. That means getting better at programming also involves getting better at communicating. Over time, you want your code to become not just easier to write but easier for others to read.

In the following three chapters, you'll learn skills to improve your programming skills:

- Copy and paste is a powerful tool, but you should avoid doing it more than twice. Repeating yourself in code is dangerous because it can easily lead to errors and inconsistencies. Instead, in Chapter 25, you'll learn how to write *functions*, which let you extract repeated tidyverse code so that it can be easily reused.
- Functions extract repeated code, but you often need to repeat the same actions on different inputs. You need tools for *iteration* that let you do similar things again and again. These tools include for loops and functional programming, which you'll learn about in Chapter 26.
- As you read more code written by others, you'll see more code that doesn't use the tidyverse. In Chapter 27, you'll learn some of the most important base R functions that you'll see in the wild.

The goal of these chapters is to teach you the minimum about programming that you need for data science. Once you have mastered the material here, we strongly recommend you continue to invest in your programming skills. We've written two books that you might find helpful. *Hands on Programming with R* by Garrett Grolemund (O'Reilly) is an introduction to R as a programming language and is a great place to start if R is your first programming language. *Advanced R* by Hadley Wickham (CRC Press) dives into the details of R the programming language; it's a great place to start if you have existing programming experience and a great next step once you've internalized the ideas in these chapters.

CHAPTER 25
Functions

Introduction

One of the best ways to improve your reach as a data scientist is to write functions. Functions allow you to automate common tasks in a more powerful and general way than copy and pasting. Writing a function has three big advantages over using copy and paste:

- You can give a function an evocative name that makes your code easier to understand.
- As requirements change, you need to update code only in one place, instead of many.
- You eliminate the chance of making incidental mistakes when you copy and paste (i.e., updating a variable name in one place but not in another).
- It makes it easier to reuse work from project to project, increasing your productivity over time.

A good rule of thumb is to consider writing a function whenever you've copied and pasted a block of code more than twice (i.e., you now have three copies of the same code). In this chapter, you'll learn about three useful types of functions:

- Vector functions take one or more vectors as input and return a vector as output.
- Data frame functions take a data frame as input and return a data frame as output.
- Plot functions take a data frame as input and return a plot as output.

Each of these sections includes many examples to help you generalize the patterns that you see. These examples wouldn't be possible without the help of the folks of

Twitter, and we encourage you to follow the links in the comment to see original inspirations. You might also want to read the original motivating tweets for general functions and plotting functions to see even more functions.

Prerequisites

We'll wrap up a variety of functions from around the tidyverse. We'll also use nycflights13 as a source of familiar data to use our functions with:

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

Vector Functions

We'll begin with vector functions: functions that take one or more vectors and return a vector result. For example, take a look at this code. What does it do?

```
df <- tibble(
  a = rnorm(5),
  b = rnorm(5),
  c = rnorm(5),
  d = rnorm(5),
)

df |> mutate(
  a = (a - min(a, na.rm = TRUE)) / 
    (max(a, na.rm = TRUE) - min(a, na.rm = TRUE)),
  b = (b - min(b, na.rm = TRUE)) / 
    (max(b, na.rm = TRUE) - min(a, na.rm = TRUE)),
  c = (c - min(c, na.rm = TRUE)) / 
    (max(c, na.rm = TRUE) - min(c, na.rm = TRUE)),
  d = (d - min(d, na.rm = TRUE)) / 
    (max(d, na.rm = TRUE) - min(d, na.rm = TRUE)),
)
#> # A tibble: 5 × 4
#>       a     b     c     d
#>   <dbl> <dbl> <dbl> <dbl>
#> 1 0.339  2.59 0.291 0
#> 2 0.880  0    0.611 0.557
#> 3 0      1.37 1     0.752
#> 4 0.795  1.37 0     1
#> 5 1      1.34 0.580 0.394
```

You might be able to puzzle out that this rescales each column to have a range from 0 to 1. But did you spot the mistake? When Hadley wrote this code, he made an error when copying and pasting and forgot to change an `a` to a `b`. Preventing this type of mistake is one good reason to learn how to write functions.

Writing a Function

To write a function, you need to first analyze your repeated code to figure what parts are constant and what parts vary. If we take the preceding code and pull it outside of `mutate()`, it's a little easier to see the pattern because each repetition is now one line:

```
(a - min(a, na.rm = TRUE)) / (max(a, na.rm = TRUE) - min(a, na.rm = TRUE))
(b - min(b, na.rm = TRUE)) / (max(b, na.rm = TRUE) - min(b, na.rm = TRUE))
(c - min(c, na.rm = TRUE)) / (max(c, na.rm = TRUE) - min(c, na.rm = TRUE))
(d - min(d, na.rm = TRUE)) / (max(d, na.rm = TRUE) - min(d, na.rm = TRUE))
```

To make this a bit clearer, we can replace the bit that varies with █:

```
(█ - min(█, na.rm = TRUE)) / (max(█, na.rm = TRUE) - min(█, na.rm = TRUE))
```

To turn this into a function, you need three things:

- A *name*. Here we'll use `rescale01` because this function rescales a vector to sit between 0 and 1.
- The *arguments*. The arguments are things that vary across calls and our analysis tells us that we have just one. We'll call it `x` because this is the conventional name for a numeric vector.
- The *body*. The body is the code that's repeated across all the calls.

Then you create a function by following the template:

```
name <- function(arguments) {
  body
}
```

For this case that leads to:

```
rescale01 <- function(x) {
  (x - min(x, na.rm = TRUE)) / (max(x, na.rm = TRUE) - min(x, na.rm = TRUE))
}
```

At this point you might test with a few simple inputs to make sure you've captured the logic correctly:

```
rescale01(c(-10, 0, 10))
#> [1] 0.0 0.5 1.0
rescale01(c(1, 2, 3, NA, 5))
#> [1] 0.00 0.25 0.50   NA 1.00
```

Then you can rewrite the call to `mutate()` as:

```
df |> mutate(
  a = rescale01(a),
  b = rescale01(b),
  c = rescale01(c),
  d = rescale01(d),
)
#> # A tibble: 5 × 4
#>       a     b     c     d
#>   <dbl> <dbl> <dbl> <dbl>
#> 1 0.339 1     0.291 0
#> 2 0.880 0     0.611 0.557
#> 3 0     0.530 1     0.752
#> 4 0.795 0.531 0     1
#> 5 1     0.518 0.580 0.394
```

(In Chapter 26, you'll learn how to use `across()` to reduce the duplication even further so all you need is `df |> mutate(across(a:d, rescale01))`.)

Improving Our Function

You might notice that the `rescale01()` function does some unnecessary work—instead of computing `min()` twice and `max()` once, we could compute both the minimum and maximum in one step with `range()`:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}
```

Or you might try this function on a vector that includes an infinite value:

```
x <- c(1:10, Inf)
rescale01(x)
#>  [1]   0   0   0   0   0   0   0   0   0   0 NaN
```

That result is not particularly useful, so we could ask `range()` to ignore infinite values:

```
rescale01 <- function(x) {
  rng <- range(x, na.rm = TRUE, finite = TRUE)
  (x - rng[1]) / (rng[2] - rng[1])
}

rescale01(x)
#>  [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667
#>  [8] 0.7777778 0.8888889 1.0000000       Inf
```

These changes illustrate an important benefit of functions: because we've moved the repeated code into a function, we need to make the change in only one place.

Mutate Functions

Now that you understand the basic idea of functions, let's take a look at a whole bunch of examples. We'll start by looking at "mutate" functions, i.e., functions that work well inside of `mutate()` and `filter()` because they return an output of the same length as the input.

Let's start with a simple variation of `rescale01()`. Maybe you want to compute the Z-score, rescaling a vector to have a mean of 0 and a standard deviation of 1:

```
z_score <- function(x) {
  (x - mean(x, na.rm = TRUE)) / sd(x, na.rm = TRUE)
}
```

Or maybe you want to wrap up a straightforward case_when() and give it a useful name. For example, this clamp() function ensures all values of a vector lie in between a minimum or a maximum:

```
clamp <- function(x, min, max) {
  case_when(
    x < min ~ min,
    x > max ~ max,
    .default = x
  )
}

clamp(1:10, min = 3, max = 7)
#>  [1] 3 3 3 4 5 6 7 7 7 7
```

Of course, functions don't just need to work with numeric variables. You might want to do some repeated string manipulation. Maybe you need to make the first character uppercase:

```
first_upper <- function(x) {
  str_sub(x, 1, 1) <- str_to_upper(str_sub(x, 1, 1))
  x
}

first_upper("hello")
#> [1] "Hello"
```

Or maybe you want to strip percent signs, commas, and dollar signs from a string before converting it into a number:

```
# https://twitter.com/NVlabormarket/status/1571939851922198530
clean_number <- function(x) {
  is_pct <- str_detect(x, "%")
  num <- x |>
    str_remove_all("%") |>
    str_remove_all(",") |>
    str_remove_all(fixed("$")) |>
    as.numeric(x)
  if_else(is_pct, num / 100, num)
}

clean_number("$12,300")
#> [1] 12300
clean_number("45%")
#> [1] 0.45
```

Sometimes your functions will be highly specialized for one data analysis step. For example, if you have a bunch of variables that record missing values as 997, 998, or 999, you might want to write a function to replace them with NA:

```
fix_na <- function(x) {
  if_else(x %in% c(997, 998, 999), NA, x)
}
```

We've focused on examples that take a single vector because we think they're the most common. But there's no reason that your function can't take multiple vector inputs.

Summary Functions

Another important family of vector functions is summary functions, functions that return a single value for use in `summarize()`. Sometimes this can just be a matter of setting a default argument or two:

```
commas <- function(x) {
  str_flatten(x, collapse = ", ", last = " and ")
}

commas(c("cat", "dog", "pigeon"))
#> [1] "cat, dog and pigeon"
```

Or you might wrap up a simple computation, like for the coefficient of variation, which divides the standard deviation by the mean:

```
cv <- function(x, na.rm = FALSE) {
  sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)
}

cv(runif(100, min = 0, max = 50))
#> [1] 0.5196276
cv(runif(100, min = 0, max = 500))
#> [1] 0.5652554
```

Or maybe you just want to make a common pattern easier to remember by giving it a memorable name:

```
# https://twitter.com/gbganalyst/status/1571619641390252033
n_missing <- function(x) {
  sum(is.na(x))
}
```

You can also write functions with multiple vector inputs. For example, maybe you want to compute the mean absolute prediction error to help you compare model predictions with actual values:

```
# https://twitter.com/neilgcurrie/status/1571607727255834625
mape <- function(actual, predicted) {
  sum(abs((actual - predicted) / actual)) / length(actual)
}
```

RStudio

Once you start writing functions, there are two RStudio shortcuts that are super useful:

- To find the definition of a function that you've written, place the cursor on the name of the function and press F2.
- To quickly jump to a function, press Ctrl+. to open the fuzzy file and function finder and type the first few letters of your function name. You can also navigate to files, Quarto sections, and more, making it a handy navigation tool.

Exercises

1. Practice turning the following code snippets into functions. Think about what each function does. What would you call it? How many arguments does it need?

   ```
   mean(is.na(x))
   mean(is.na(y))
   mean(is.na(z))

   x / sum(x, na.rm = TRUE)
   y / sum(y, na.rm = TRUE)
   z / sum(z, na.rm = TRUE)

   round(x / sum(x, na.rm = TRUE) * 100, 1)
   round(y / sum(y, na.rm = TRUE) * 100, 1)
   round(z / sum(z, na.rm = TRUE) * 100, 1)
   ```

2. In the second variant of `rescale01()`, infinite values are left unchanged. Can you rewrite `rescale01()` so that `-Inf` is mapped to 0, and `Inf` is mapped to 1?

3. Given a vector of birthdates, write a function to compute the age in years.

4. Write your own functions to compute the variance and skewness of a numeric vector. You can look up the definitions on Wikipedia or elsewhere.

5. Write `both_na()`, a summary function that takes two vectors of the same length and returns the number of positions that have an `NA` in both vectors.

6. Read the documentation to figure out what the following functions do. Why are they useful even though they are so short?

   ```
   is_directory <- function(x) {
     file.info(x)$isdir
   }
   is_readable <- function(x) {
     file.access(x, 4) == 0
   }
   ```

Data Frame Functions

Vector functions are useful for pulling out code that's repeated within a dplyr verb. But you'll often also repeat the verbs themselves, particularly within a large pipeline. When you notice yourself copying and pasting multiple verbs multiple times, you might think about writing a data frame function. Data frame functions work like dplyr verbs: they take a data frame as the first argument and some extra arguments that say what to do with it and return a data frame or vector.

To let you write a function that uses dplyr verbs, we'll first introduce you to the challenge of indirection and how you can overcome it with embracing, `{{ }}`. We'll then show you a bunch of examples to illustrate what you might do with it.

Indirection and Tidy Evaluation

When you start writing functions that use dplyr verbs, you rapidly hit the problem of indirection. Let's illustrate the problem with a simple function: `grouped_mean()`. The goal of this function is to compute the mean of `mean_var` grouped by `group_var`:

```
grouped_mean <- function(df, group_var, mean_var) {
  df |>
    group_by(group_var) |>
    summarize(mean(mean_var))
}
```

If we try and use it, we get an error:

```
diamonds |> grouped_mean(cut, carat)
#> Error in `group_by()`:
#> ! Must group by variables found in `.data`.
#> ✖ Column `group_var` is not found.
```

To make the problem a bit clearer, we can use a made-up data frame:

```
df <- tibble(
  mean_var = 1,
  group_var = "g",
  group = 1,
  x = 10,
  y = 100
)

df |> grouped_mean(group, x)
#> # A tibble: 1 × 2
#>   group_var `mean(mean_var)`
#>   <chr>                <dbl>
#> 1 g                        1
df |> grouped_mean(group, y)
#> # A tibble: 1 × 2
#>   group_var `mean(mean_var)`
#>   <chr>                <dbl>
#> 1 g                        1
```

Regardless of how we call `grouped_mean()` it always does `df |> group_by(group_var) |> summarize(mean(mean_var))`, instead of `df |> group_by(group) |> summarize(mean(x))` or `df |> group_by(group) |> summarize(mean(y))`. This is a problem of indirection, and it arises because dplyr uses *tidy evaluation* to allow you to refer to the names of variables inside your data frame without any special treatment.

Tidy evaluation is great 95% of the time because it makes your data analyses very concise as you never have to say which data frame a variable comes from; it's obvious from the context. The downside of tidy evaluation comes when we want to wrap up repeated tidyverse code into a function. Here we need some way to tell `group_mean()` and `summarize()` not to treat `group_var` and `mean_var` as the name of the variables but instead look inside them for the variable we actually want to use.

Tidy evaluation includes a solution to this problem called *embracing*. Embracing a variable means to wrap it in braces, so, for example, var becomes {{ var }}. Embracing a variable tells dplyr to use the value stored inside the argument, not the argument as the literal variable name. One way to remember what's happening is to think of {{ }} as looking down a tunnel—{{ var }} will make a dplyr function look inside of var rather than looking for a variable called var.

So to make grouped_mean() work, we need to surround group_var and mean_var with {{ }}:

```
grouped_mean <- function(df, group_var, mean_var) {
  df |>
    group_by({{ group_var }}) |>
    summarize(mean({{ mean_var }}))
}

df |> grouped_mean(group, x)
#> # A tibble: 1 × 2
#>   group `mean(x)`
#>   <dbl>     <dbl>
#> 1     1        10
```

Success!

When to Embrace?

The key challenge in writing data frame functions is figuring out which arguments need to be embraced. Fortunately, this is easy because you can look it up in the documentation. There are two terms to look for in the docs that correspond to the two most common subtypes of tidy evaluation:

Data masking
: This is used in functions such as arrange(), filter(), and summarize() that compute with variables.

Tidy selection
: This is used for functions such as select(), relocate(), and rename() that select variables.

Your intuition about which arguments use tidy evaluation should be good for many common functions—just think about whether you can compute (e.g., x + 1) or select (e.g., a:x).

In the following sections, we'll explore the sorts of handy functions you might write once you understand embracing.

Common Use Cases

If you commonly perform the same set of summaries when doing initial data exploration, you might consider wrapping them up in a helper function:

```
summary6 <- function(data, var) {
  data |> summarize(
    min = min({{ var }}, na.rm = TRUE),
    mean = mean({{ var }}, na.rm = TRUE),
    median = median({{ var }}, na.rm = TRUE),
    max = max({{ var }}, na.rm = TRUE),
    n = n(),
    n_miss = sum(is.na({{ var }})),
    .groups = "drop"
  )
}

diamonds |> summary6(carat)
#> # A tibble: 1 × 6
#>     min  mean median   max     n n_miss
#>   <dbl> <dbl>  <dbl> <dbl> <int>  <int>
#> 1   0.2 0.798    0.7  5.01 53940      0
```

(Whenever you wrap `summarize()` in a helper, we think it's good practice to set `.groups = "drop"` to both avoid the message and leave the data in an ungrouped state.)

The nice thing about this function is that because it wraps `summarize()`, you can use it on grouped data:

```
diamonds |>
  group_by(cut) |>
  summary6(carat)
#> # A tibble: 5 × 7
#>   cut         min  mean median   max     n n_miss
#>   <ord>     <dbl> <dbl>  <dbl> <dbl> <int>  <int>
#> 1 Fair       0.22 1.05    1     5.01  1610      0
#> 2 Good       0.23 0.849   0.82  3.01  4906      0
#> 3 Very Good  0.2  0.806   0.71  4    12082      0
#> 4 Premium    0.2  0.892   0.86  4.01 13791      0
#> 5 Ideal      0.2  0.703   0.54  3.5  21551      0
```

Furthermore, since the arguments to summarize are data masking, the `var` argument to `summary6()` is also data masking. That means you can also summarize computed variables:

```
diamonds |>
  group_by(cut) |>
  summary6(log10(carat))
#> # A tibble: 5 × 7
#>   cut          min    mean  median   max     n n_miss
#>   <ord>      <dbl>   <dbl>   <dbl> <dbl> <int>  <int>
#> 1 Fair      -0.658 -0.0273  0      0.700  1610      0
#> 2 Good      -0.638 -0.133  -0.0862 0.479  4906      0
#> 3 Very Good -0.699 -0.164  -0.149  0.602 12082      0
#> 4 Premium   -0.699 -0.125  -0.0655 0.603 13791      0
#> 5 Ideal     -0.699 -0.225  -0.268  0.544 21551      0
```

To summarize multiple variables, you'll need to wait until "Modifying Multiple Columns" on page 466, where you'll learn how to use across().

Another popular summarize() helper function is a version of count() that also computes proportions:

```
# https://twitter.com/Diabb6/status/1571635146658402309
count_prop <- function(df, var, sort = FALSE) {
  df |>
    count({{ var }}, sort = sort) |>
    mutate(prop = n / sum(n))
}

diamonds |> count_prop(clarity)
#> # A tibble: 8 × 3
#>   clarity     n   prop
#>   <ord>   <int>  <dbl>
#> 1 I1        741 0.0137
#> 2 SI2      9194 0.170
#> 3 SI1     13065 0.242
#> 4 VS2     12258 0.227
#> 5 VS1      8171 0.151
#> 6 VVS2     5066 0.0939
#> # … with 2 more rows
```

This function has three arguments: df, var, and sort. Only var needs to be embraced because it's passed to count(), which uses data masking for all variables. Note that we use a default value for sort so that if the user doesn't supply their own value, it will default to FALSE.

Or maybe you want to find the sorted unique values of a variable for a subset of the data. Rather than supplying a variable and a value to do the filtering, we'll allow the user to supply a condition:

```
unique_where <- function(df, condition, var) {
  df |>
    filter({{ condition }}) |>
    distinct({{ var }}) |>
    arrange({{ var }})
}

# Find all the destinations in December
flights |> unique_where(month == 12, dest)
#> # A tibble: 96 × 1
#>   dest
#>   <chr>
#> 1 ABQ
#> 2 ALB
#> 3 ATL
#> 4 AUS
#> 5 AVL
#> 6 BDL
#> # … with 90 more rows
```

Here we embrace condition because it's passed to filter() and var because it's passed to distinct() and arrange().

We've made all these examples to take a data frame as the first argument, but if you're working repeatedly with the same data, it can make sense to hardcode it. For example, the following function always works with the flights dataset and always selects time_hour, carrier, and flight since they form the compound primary key that allows you to identify a row:

```
subset_flights <- function(rows, cols) {
  flights |>
    filter({{ rows }}) |>
    select(time_hour, carrier, flight, {{ cols }})
}
```

Data Masking Versus Tidy Selection

Sometimes you want to select variables inside a function that uses data masking. For example, imagine you want to write a count_missing() method that counts the number of missing observations in rows. You might try writing something like:

```
count_missing <- function(df, group_vars, x_var) {
  df |>
    group_by({{ group_vars }}) |>
    summarize(
      n_miss = sum(is.na({{ x_var }})),
      .groups = "drop"
    )
}

flights |>
  count_missing(c(year, month, day), dep_time)
#> Error in `group_by()`:
#> ℹ In argument: `c(year, month, day)`.
#> Caused by error:
#> ! `c(year, month, day)` must be size 336776 or 1, not 1010328.
```

This doesn't work because group_by() uses data masking, not tidy selection. We can work around that problem by using the handy pick() function, which allows you to use tidy selection inside data-masking functions:

```
count_missing <- function(df, group_vars, x_var) {
  df |>
    group_by(pick({{ group_vars }})) |>
    summarize(
      n_miss = sum(is.na({{ x_var }})),
      .groups = "drop"
  )
}

flights |>
  count_missing(c(year, month, day), dep_time)
#> # A tibble: 365 × 4
#>     year month   day n_miss
#>    <int> <int> <int>  <int>
#> 1  2013     1     1      4
#> 2  2013     1     2      8
#> 3  2013     1     3     10
#> 4  2013     1     4     6
#> 5  2013     1     5     3
#> 6  2013     1     6     1
#> # … with 359 more rows
```

Another convenient use of `pick()` is to make a 2D table of counts. Here we count using all the variables in the `rows` and `columns` and then use `pivot_wider()` to rearrange the counts into a grid:

```
# https://twitter.com/pollicipes/status/1571606508944719876
count_wide <- function(data, rows, cols) {
  data |>
    count(pick(c({{ rows }}, {{ cols }}))) |>
    pivot_wider(
      names_from = {{ cols }},
      values_from = n,
      names_sort = TRUE,
      values_fill = 0
    )
}

diamonds |> count_wide(c(clarity, color), cut)
#> # A tibble: 56 × 7
#>   clarity color  Fair  Good `Very Good` Premium Ideal
#>   <ord>   <ord> <int> <int>       <int>   <int> <int>
#> 1 I1      D         4     8           5      12    13
#> 2 I1      E         9    23          22      30    18
#> 3 I1      F        35    19          13      34    42
#> 4 I1      G        53    19          16      46    16
#> 5 I1      H        52    14          12      46    38
#> 6 I1      I        34     9           8      24    17
#> # … with 50 more rows
```

While our examples have mostly focused on dplyr, tidy evaluation also underpins tidyr, and if you look at the `pivot_wider()` docs, you can see that `names_from` uses tidy selection.

Exercises

1. Using the datasets from nycflights13, write a function that:

   a. Finds all flights that were cancelled (i.e., `is.na(arr_time)`) or delayed by more than an hour:

      ```
      flights |> filter_severe()
      ```

   b. Counts the number of cancelled flights and the number of flights delayed by more than an hour:

      ```
      flights |> group_by(dest) |> summarize_severe()
      ```

   c. Finds all flights that were cancelled or delayed by more than a user-supplied number of hours:

      ```
      flights |> filter_severe(hours = 2)
      ```

d. Summarizes the weather to compute the minimum, mean, and maximum of a user-supplied variable:

```
weather |> summarize_weather(temp)
```

e. Converts the user-supplied variable that uses clock time (e.g., `dep_time`, `arr_time`, etc.) into a decimal time (i.e., hours + [minutes / 60]):

```
weather |> standardize_time(sched_dep_time)
```

2. For each of the following functions, list all arguments that use tidy evaluation and describe whether they use data masking or tidy selection: `distinct()`, `count()`, `group_by()`, `rename_with()`, `slice_min()`, `slice_sample()`.

3. Generalize the following function so that you can supply any number of variables to count:

```
count_prop <- function(df, var, sort = FALSE) {
  df |>
    count({{ var }}, sort = sort) |>
    mutate(prop = n / sum(n))
}
```

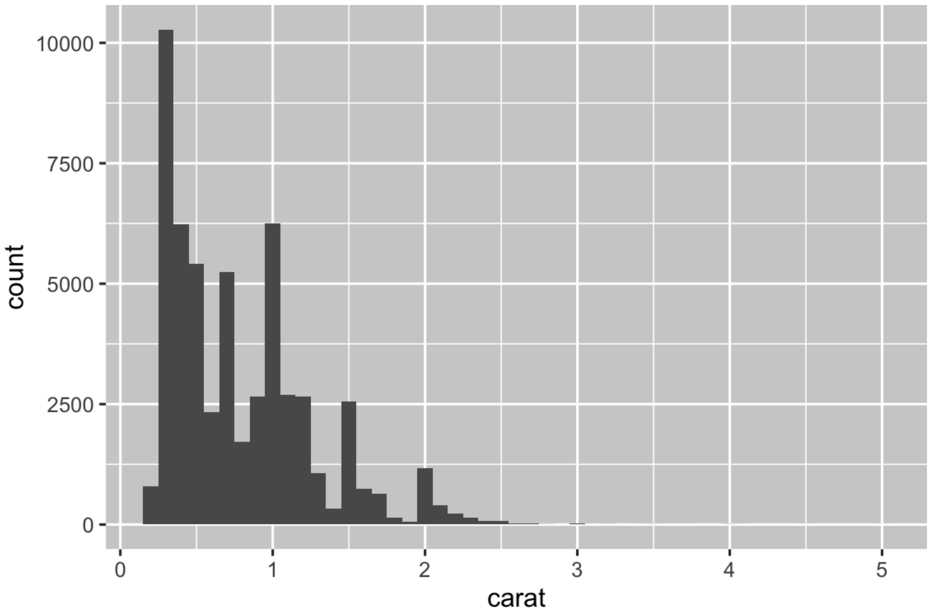
Plot Functions

Instead of returning a data frame, you might want to return a plot. Fortunately, you can use the same techniques with ggplot2, because `aes()` is a data-masking function. For example, imagine that you're making a lot of histograms:

```
diamonds |>
  ggplot(aes(x = carat)) +
  geom_histogram(binwidth = 0.1)

diamonds |>
  ggplot(aes(x = carat)) +
  geom_histogram(binwidth = 0.05)
```

Wouldn't it be nice if you could wrap this up into a histogram function? This is easy as pie once you know that `aes()` is a data-masking function and you need to embrace:

```
histogram <- function(df, var, binwidth = NULL) {
  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth)
}

diamonds |> histogram(carat, 0.1)
```



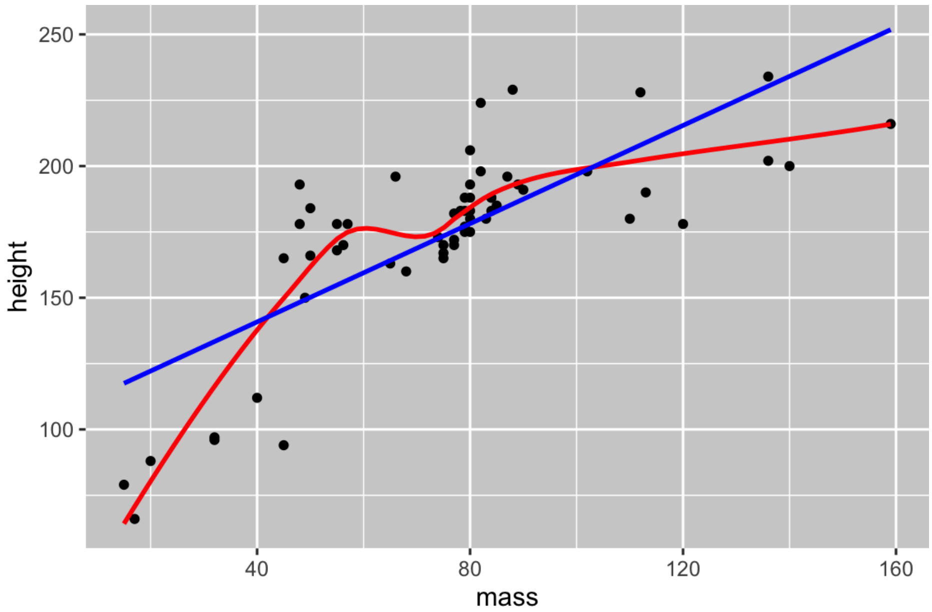
Note that `histogram()` returns a ggplot2 plot, meaning you can still add components if you want. Just remember to switch from `|>` to `+`:

```
diamonds |>
  histogram(carat, 0.1) +
  labs(x = "Size (in carats)", y = "Number of diamonds")
```

More Variables

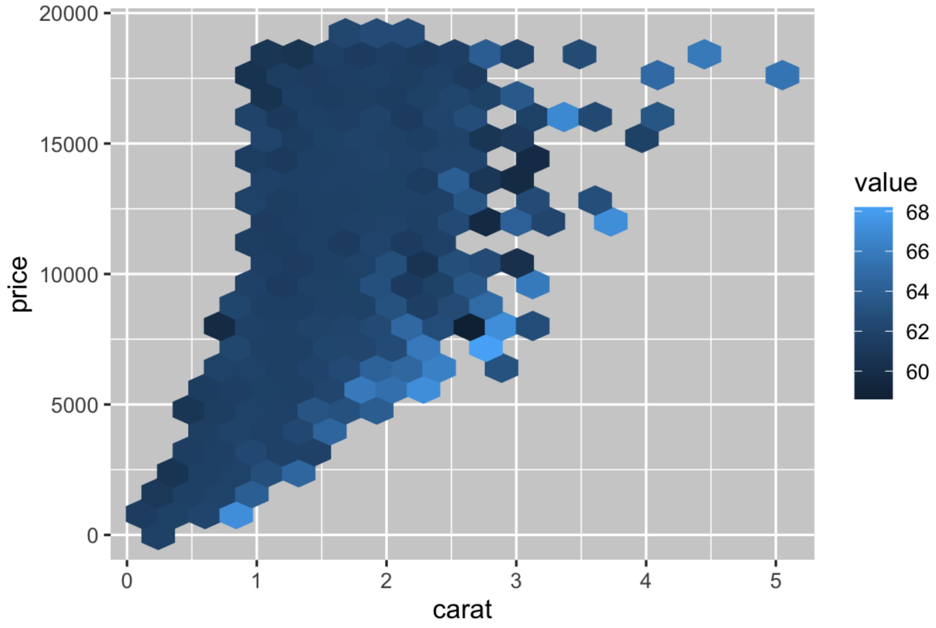
It's straightforward to add more variables to the mix. For example, maybe you want an easy way to eyeball whether a dataset is linear by overlaying a smooth line and a straight line:

```
# https://twitter.com/tyler_js_smith/status/1574377116988104704
linearity_check <- function(df, x, y) {
  df |>
    ggplot(aes(x = {{ x }}, y = {{ y }})) +
    geom_point() +
    geom_smooth(method = "loess", formula = y ~ x, color = "red", se = FALSE) +
    geom_smooth(method = "lm", formula = y ~ x, color = "blue", se = FALSE)
}

starwars |>
  filter(mass < 1000) |>
  linearity_check(mass, height)
```



Or maybe you want an alternative to colored scatterplots for very large datasets where overplotting is a problem:

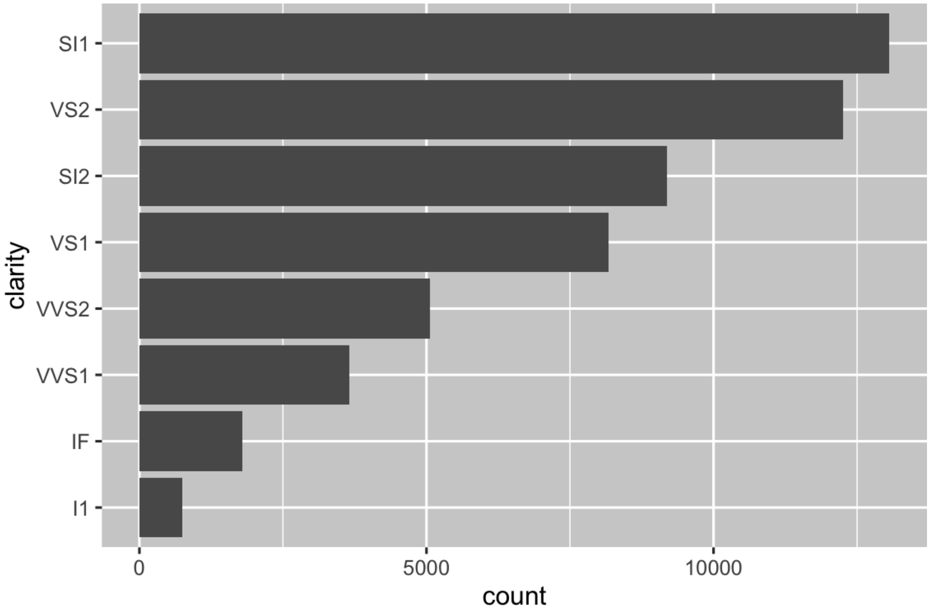
```
# https://twitter.com/ppaxisa/status/1574398423175921665
hex_plot <- function(df, x, y, z, bins = 20, fun = "mean") {
  df |>
    ggplot(aes(x = {{ x }}, y = {{ y }}, z = {{ z }})) +
    stat_summary_hex(
      aes(color = after_scale(fill)), # make border same color as fill
      bins = bins,
      fun = fun,
    )
}

diamonds |> hex_plot(carat, price, depth)
```



Combining with Other Tidyverse Packages

Some of the most useful helpers combine a dash of data manipulation with ggplot2. For example, you might want to do a vertical bar chart where you automatically sort the bars in frequency order using `fct_infreq()`. Since the bar chart is vertical, we also need to reverse the usual order to get the highest values at the top:

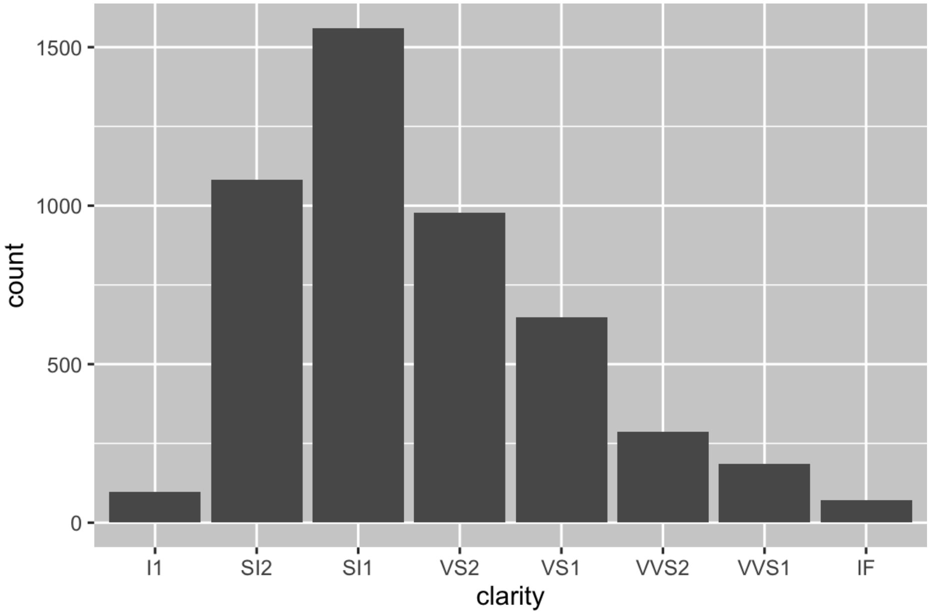
```
sorted_bars <- function(df, var) {
  df |>
    mutate({{ var }} := fct_rev(fct_infreq({{ var }})))  |>
    ggplot(aes(y = {{ var }})) +
    geom_bar()
}

diamonds |> sorted_bars(clarity)
```



We have to use a new operator here, `:=`, because we are generating the variable name based on user-supplied data. Variable names go on the left of =, but R's syntax doesn't allow anything to the left of = except for a single literal name. To work around this problem, we use the special operator `:=`, which tidy evaluation treats in the same way as =.

Or maybe you want to make it easy to draw a bar plot just for a subset of the data:

```
conditional_bars <- function(df, condition, var) {
  df |>
    filter({{ condition }}) |>
    ggplot(aes(x = {{ var }})) +
    geom_bar()
}

diamonds |> conditional_bars(cut == "Good", clarity)
```



You can also get creative and display data summaries in other ways. You can find a cool application at *https://oreil.ly/MV4kQ*; it uses the axis labels to display the highest value. As you learn more about ggplot2, the power of your functions will continue to increase.

We'll finish with a more complicated case: labeling the plots you create.

Labeling

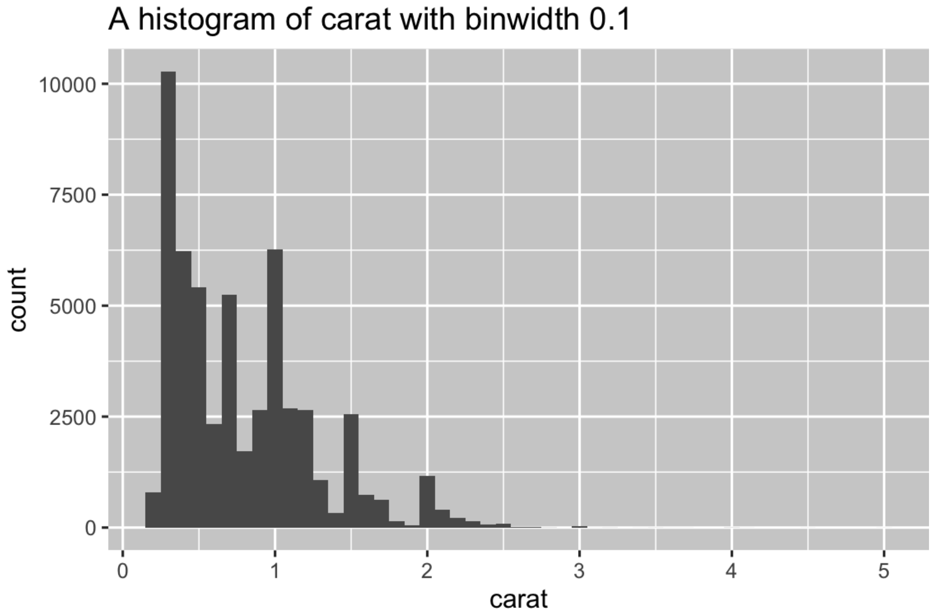
Remember the histogram function we showed you earlier?

```
histogram <- function(df, var, binwidth = NULL) {
  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth)
}
```

Wouldn't it be nice if we could label the output with the variable and the bin width that was used? To do so, we're going to have to go under the covers of tidy evaluation and use a function from the package we haven't talked about yet: rlang. rlang is a low-level package that's used by just about every other package in the tidyverse because it implements tidy evaluation (as well as many other useful tools).

To solve the labeling problem, we can use `rlang::englue()`. This works similarly to `str_glue()`, so any value wrapped in `{ }` will be inserted into the string. But it also understands `{{ }}`, which automatically inserts the appropriate variable name:

```
histogram <- function(df, var, binwidth) {
  label <- rlang::englue("A histogram of {{var}} with binwidth {binwidth}")

  df |>
    ggplot(aes(x = {{ var }})) +
    geom_histogram(binwidth = binwidth) +
    labs(title = label)
}

diamonds |> histogram(carat, 0.1)
```



You can use the same approach in any other place where you want to supply a string in a ggplot2 plot.

Exercises

Build up a rich plotting function by incrementally implementing each of these steps:

1. Draw a scatterplot given a dataset and x and y variables.
2. Add a line of best fit (i.e., a linear model with no standard errors).
3. Add a title.

Style

R doesn't care what your function or arguments are called, but the names make a big difference for humans. Ideally, the name of your function will be short but clearly evoke what the function does. That's hard! But it's better to be clear than short, as RStudio's autocomplete makes it easy to type long names.

Generally, function names should be verbs, and arguments should be nouns. There are some exceptions: nouns are OK if the function computes a well-known noun (i.e., `mean()` is better than `compute_mean()`) or accesses some property of an object (i.e., `coef()` is better than `get_coefficients()`). Use your best judgment and don't be afraid to rename a function if you figure out a better name later.

```
# Too short
f()

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

R also doesn't care about how you use whitespace in your functions, but future readers will. Continue to follow the rules from Chapter 4. Additionally, `function()` should always be followed by squiggly brackets (`{}`), and the contents should be indented by an additional two spaces. This makes it easier to see the hierarchy in your code by skimming the left margin.

```
# Missing extra two spaces
density <- function(color, facets, binwidth = 0.1) {
diamonds |>
  ggplot(aes(x = carat, y = after_stat(density), color = {{ color }})) +
  geom_freqpoly(binwidth = binwidth) +
  facet_wrap(vars({{ facets }}))
}

# Pipe indented incorrectly
density <- function(color, facets, binwidth = 0.1) {
  diamonds |>
  ggplot(aes(x = carat, y = after_stat(density), color = {{ color }})) +
  geom_freqpoly(binwidth = binwidth) +
  facet_wrap(vars({{ facets }}))
}
```

As you can see, we recommend putting extra spaces inside `{{ }}`. This makes it obvious that something unusual is happening.

Exercises

1. Read the source code for each of the following two functions, puzzle out what they do, and then brainstorm better names:

   ```
   f1 <- function(string, prefix) {
     str_sub(string, 1, str_length(prefix)) == prefix
   }

   f3 <- function(x, y) {
     rep(y, length.out = length(x))
   }
   ```

2. Take a function that you've written recently and spend five minutes brainstorming a better name for it and its arguments.

3. Make a case for why `norm_r()`, `norm_d()`, etc., would be better than `rnorm()` and `dnorm()`. Make a case for the opposite. How could you make the names even clearer?

Summary

In this chapter, you learned how to write functions for three useful scenarios: creating a vector, creating a data frame, or creating a plot. Along the way you saw many examples, which ideally started to get your creative juices flowing, and gave you some ideas for where functions might help your analysis code.

We have shown you only the bare minimum to get started with functions and there's much more to learn. A few places to learn more are:

- To learn more about programming with tidy evaluation, see useful recipes in programming with dplyr and programming with tidyr and learn more about the theory in "What is data masking and why do I need {{?".
- To learn more about reducing duplication in your ggplot2 code, read the "Programming with ggplot2" chapter of the ggplot2 book.
- For more advice on function style, see the tidyverse style guide.

In the next chapter, we'll dive into iteration which gives you further tools for reducing code duplication.

CHAPTER 26

Iteration

Introduction

In this chapter, you'll learn tools for iteration, repeatedly performing the same action on different objects. Iteration in R generally tends to look rather different from other programming languages because so much of it is implicit and we get it for free. For example, if you want to double a numeric vector x in R, you can just write 2 * x. In most other languages, you'd need to explicitly double each element of x using some sort of for loop.

This book has already given you a small but powerful number of tools that perform the same action for multiple "things":

- facet_wrap() and facet_grid() draw a plot for each subset.
- group_by() plus summarize() computes a summary statistics for each subset.
- unnest_wider() and unnest_longer() create new rows and columns for each element of a list column.

Now it's time to learn some more general tools, often called *functional programming* tools because they are built around functions that take other functions as inputs. Learning functional programming can easily veer into the abstract, but in this chapter we'll keep things concrete by focusing on three common tasks: modifying multiple columns, reading multiple files, and saving multiple objects.

Prerequisites

In this chapter, we'll focus on tools provided by dplyr and purrr, both core members of the tidyverse. You've seen dplyr before, but purrr is new. We're just going to use a couple of purrr functions in this chapter, but it's a great package to explore as you improve your programming skills:

```
library(tidyverse)
```

Modifying Multiple Columns

Imagine you have this simple tibble and you want to count the number of observations and compute the median of every column:

```
df <- tibble(
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)
```

You could do it with copy and paste:

```
df |> summarize(
  n = n(),
  a = median(a),
  b = median(b),
  c = median(c),
  d = median(d),
)
#> # A tibble: 1 × 5
#>       n      a      b       c     d
#>   <int>  <dbl>  <dbl>   <dbl> <dbl>
#> 1    10 -0.246 -0.287 -0.0567 0.144
```

That breaks our rule of thumb to never copy and paste more than twice, and you can imagine that this will get tedious if you have tens or even hundreds of columns. Instead, you can use `across()`:

```
df |> summarize(
  n = n(),
  across(a:d, median),
)
#> # A tibble: 1 × 5
#>       n      a      b       c     d
#>   <int>  <dbl>  <dbl>   <dbl> <dbl>
#> 1    10 -0.246 -0.287 -0.0567 0.144
```

`across()` has three particularly important arguments, which we'll discuss in detail in the following sections. You'll use the first two every time you use `across()`: the first argument, `.cols`, specifies which columns you want to iterate over, and the second argument, `.fns`, specifies what to do with each column. You can use the `.names` argument when you need additional control over the names of output columns, which is particularly important when you use `across()` with `mutate()`. We'll also discuss two important variations, `if_any()` and `if_all()`, which work with `filter()`.

Selecting Columns with .cols

The first argument to across(), .cols, selects the columns to transform. This uses the same specifications as select(), "select()" on page 49, so you can use functions such as starts_with() and ends_with() to select columns based on their name.

There are two additional selection techniques that are particularly useful for across(): everything() and where(). everything() is straightforward: it selects every (nongrouping) column:

```
df <- tibble(
  grp = sample(2, 10, replace = TRUE),
  a = rnorm(10),
  b = rnorm(10),
  c = rnorm(10),
  d = rnorm(10)
)

df |>
  group_by(grp) |>
  summarize(across(everything(), median))
#> # A tibble: 2 × 5
#>     grp       a       b     c     d
#>   <int>   <dbl>   <dbl> <dbl> <dbl>
#> 1     1 -0.0935 -0.0163 0.363 0.364
#> 2     2  0.312  -0.0576 0.208 0.565
```

Note grouping columns (grp here) are not included in across(), because they're automatically preserved by summarize().

where() allows you to select columns based on their type:

where(is.numeric)
: Selects all numeric columns.

where(is.character)
: Selects all string columns.

where(is.Date)
: Selects all date columns.

where(is.POSIXct)
: Selects all date-time columns.

where(is.logical)
: selects all logical columns.

Just like other selectors, you can combine these with Boolean algebra. For example, !where(is.numeric) selects all non-numeric columns, and starts_with("a") & where(is.logical) selects all logical columns whose name starts with "a."

Calling a Single Function

The second argument to `across()` defines how each column will be transformed. In simple cases, as shown, this will be a single existing function. This is a pretty special feature of R: we're passing one function (`median`, `mean`, `str_flatten`, ...) to another function (`across`). This is one of the features that makes R a functional programming language.

It's important to note that we're passing this function to `across()`, so `across()` can call it; we're not calling it ourselves. That means the function name should never be followed by `()`. If you forget, you'll get an error:

```
df |>
  group_by(grp) |>
  summarize(across(everything(), median()))
#> Error in `summarize()`:
#> ℹ In argument: `across(everything(), median())`.
#> Caused by error in `is.factor()`:
#> ! argument "x" is missing, with no default
```

This error arises because you're calling the function with no input, e.g.:

```
median()
#> Error in is.factor(x): argument "x" is missing, with no default
```

Calling Multiple Functions

In more complex cases, you might want to supply additional arguments or perform multiple transformations. Let's motivate this problem with a simple example: what happens if we have some missing values in our data? `median()` propagates those missing values, giving us a suboptimal output:

```
rnorm_na <- function(n, n_na, mean = 0, sd = 1) {
  sample(c(rnorm(n - n_na, mean = mean, sd = sd), rep(NA, n_na)))
}

df_miss <- tibble(
  a = rnorm_na(5, 1),
  b = rnorm_na(5, 1),
  c = rnorm_na(5, 2),
  d = rnorm(5)
)
df_miss |>
  summarize(
    across(a:d, median),
    n = n()
  )
#> # A tibble: 1 × 5
#>       a     b     c     d     n
#>   <dbl> <dbl> <dbl> <dbl> <int>
#> 1    NA    NA    NA  1.15     5
```

It would be nice if we could pass along `na.rm = TRUE` to `median()` to remove these missing values. To do so, instead of calling `median()` directly, we need to create a new function that calls `median()` with the desired arguments:

```
df_miss |>
  summarize(
    across(a:d, function(x) median(x, na.rm = TRUE)),
    n = n()
  )
#> # A tibble: 1 × 5
#>       a     b      c     d     n
#>   <dbl> <dbl>  <dbl> <dbl> <int>
#> 1 0.139 -1.11 -0.387  1.15     5
```

This is a little verbose, so R comes with a handy shortcut: for this sort of throwaway (or *anonymous*)[1] function, you can replace `function` with `\`:[2]

```
df_miss |>
  summarize(
    across(a:d, \(x) median(x, na.rm = TRUE)),
    n = n()
  )
```

In either case, `across()` effectively expands to the following code:

```
df_miss |>
  summarize(
    a = median(a, na.rm = TRUE),
    b = median(b, na.rm = TRUE),
    c = median(c, na.rm = TRUE),
    d = median(d, na.rm = TRUE),
    n = n()
  )
```

When we remove the missing values from the `median()`, it would be nice to know just how many values were removed. We can find that out by supplying two functions to `across()`: one to compute the median and the other to count the missing values. You supply multiple functions by using a named list to `.fns`:

```
df_miss |>
  summarize(
    across(a:d, list(
      median = \(x) median(x, na.rm = TRUE),
      n_miss = \(x) sum(is.na(x))
    )),
    n = n()
  )
#> # A tibble: 1 × 9
```

[1] Anonymous, because we never explicitly gave it a name with `<-`. Another term programmers use for this is *lambda function*.

[2] In older code you might see syntax that looks like `~ .x + 1`. This is another way to write anonymous functions, but it works only inside tidyverse functions and always uses the variable name `.x`. We now recommend the base syntax, `\(x) x + 1`.

```
#>   a_median a_n_miss b_median b_n_miss c_median c_n_miss d_median d_n_miss
#>      <dbl>    <int>    <dbl>    <int>    <dbl>    <int>    <dbl>    <int>
#> 1    0.139        1    -1.11        1   -0.387        2     1.15        0
#> # … with 1 more variable: n <int>
```

If you look carefully, you might intuit that the columns are named using a glue specification ("str_glue()" on page 247) like {.col}_{.fn} where .col is the name of the original column and .fn is the name of the function. That's not a coincidence! As you'll learn in the next section, you can use the .names argument to supply your own glue spec.

Column Names

The result of across() is named according to the specification provided in the .names argument. We could specify our own if we wanted the name of the function to come first:[3]

```
df_miss |>
  summarize(
    across(
      a:d,
      list(
        median = \(x) median(x, na.rm = TRUE),
        n_miss = \(x) sum(is.na(x))
      ),
      .names = "{.fn}_{.col}"
    ),
    n = n(),
  )
#> # A tibble: 1 × 9
#>   median_a n_miss_a median_b n_miss_b median_c n_miss_c median_d n_miss_d
#>      <dbl>    <int>    <dbl>    <int>    <dbl>    <int>    <dbl>    <int>
#> 1    0.139        1    -1.11        1   -0.387        2     1.15        0
#> # … with 1 more variable: n <int>
```

The .names argument is particularly important when you use across() with mutate(). By default, the output of across() is given the same names as the inputs. This means that across() in mutate() will replace existing columns. For example, here we use coalesce() to replace NAs with 0:

```
df_miss |>
  mutate(
    across(a:d, \(x) coalesce(x, 0))
  )
#> # A tibble: 5 × 4
#>        a      b      c     d
#>    <dbl>  <dbl>  <dbl> <dbl>
#> 1  0.434 -1.25   0     1.60
#> 2  0     -1.43  -0.297 0.776
#> 3 -0.156 -0.980  0     1.15
```

[3] You can't currently change the order of the columns, but you could reorder them after the fact using relocate() or similar.

```
#> 4 -2.61  -0.683 -0.785 2.13 
#> 5  1.11   0     -0.387 0.704
```

If you'd like to instead create new columns, you can use the `.names` argument to give the output new names:

```
df_miss |> 
  mutate(
    across(a:d, \(x) abs(x), .names = "{.col}_abs")
  )
#> # A tibble: 5 × 8
#>        a      b      c     d  a_abs  b_abs  c_abs d_abs
#>    <dbl>  <dbl>  <dbl> <dbl>  <dbl>  <dbl>  <dbl> <dbl>
#> 1  0.434 -1.25  NA     1.60   0.434  1.25  NA     1.60 
#> 2 NA     -1.43  -0.297 0.776 NA      1.43   0.297 0.776
#> 3 -0.156 -0.980 NA     1.15   0.156  0.980 NA     1.15 
#> 4 -2.61  -0.683 -0.785 2.13   2.61   0.683  0.785 2.13 
#> 5  1.11  NA     -0.387 0.704  1.11  NA      0.387 0.704
```

Filtering

`across()` is a great match for `summarize()` and `mutate()`, but it's more awkward to use with `filter()`, because you usually combine multiple conditions with either `|` or `&`. It's clear that `across()` can help to create multiple logical columns, but then what? So dplyr provides two variants of `across()` called `if_any()` and `if_all()`:

```
# same as df_miss |> filter(is.na(a) | is.na(b) | is.na(c) | is.na(d))
df_miss |> filter(if_any(a:d, is.na))
#> # A tibble: 4 × 4
#>        a      b      c     d
#>    <dbl>  <dbl>  <dbl> <dbl>
#> 1  0.434 -1.25  NA     1.60 
#> 2 NA     -1.43  -0.297 0.776
#> 3 -0.156 -0.980 NA     1.15 
#> 4  1.11  NA     -0.387 0.704

# same as df_miss |> filter(is.na(a) & is.na(b) & is.na(c) & is.na(d))
df_miss |> filter(if_all(a:d, is.na))
#> # A tibble: 0 × 4
#> # … with 4 variables: a <dbl>, b <dbl>, c <dbl>, d <dbl>
```

across() in Functions

`across()` is particularly useful to program with because it allows you to operate on multiple columns. For example, Jacob Scott uses this little helper that wraps a bunch of lubridate functions to expand all date columns into year, month, and day columns:

```
expand_dates <- function(df) {
  df |> 
    mutate(
      across(where(is.Date), list(year = year, month = month, day = mday))
    )
}

df_date <- tibble(
  name = c("Amy", "Bob"),
  date = ymd(c("2009-08-03", "2010-01-16"))
)

df_date |>
  expand_dates()
#> # A tibble: 2 × 5
#>   name  date       date_year date_month date_day
#>   <chr> <date>         <dbl>      <dbl>    <int>
#> 1 Amy   2009-08-03      2009          8        3
#> 2 Bob   2010-01-16      2010          1       16
```

across() also makes it easy to supply multiple columns in a single argument because the first argument uses tidy-select; you just need to remember to embrace that argument, as we discussed in "When to Embrace?" on page 451. For example, this function will compute the means of numeric columns by default. But by supplying the second argument you can choose to summarize just selected columns:

```
summarize_means <- function(df, summary_vars = where(is.numeric)) {
  df |>
    summarize(
      across({{ summary_vars }}, \(x) mean(x, na.rm = TRUE)),
      n = n()
    )
}
diamonds |>
  group_by(cut) |>
  summarize_means()
#> # A tibble: 5 × 9
#>   cut       carat depth table price     x     y     z     n
#>   <ord>     <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <int>
#> 1 Fair      1.05   64.0  59.1 4359.  6.25  6.18  3.98  1610
#> 2 Good      0.849  62.4  58.7 3929.  5.84  5.85  3.64  4906
#> 3 Very Good 0.806  61.8  58.0 3982.  5.74  5.77  3.56 12082
#> 4 Premium   0.892  61.3  58.7 4584.  5.97  5.94  3.65 13791
#> 5 Ideal     0.703  61.7  56.0 3458.  5.51  5.52  3.40 21551

diamonds |>
  group_by(cut) |>
  summarize_means(c(carat, x:z))
#> # A tibble: 5 × 6
#>   cut       carat     x     y     z     n
#>   <ord>     <dbl> <dbl> <dbl> <dbl> <int>
#> 1 Fair      1.05   6.25  6.18  3.98  1610
#> 2 Good      0.849  5.84  5.85  3.64  4906
#> 3 Very Good 0.806  5.74  5.77  3.56 12082
#> 4 Premium   0.892  5.97  5.94  3.65 13791
#> 5 Ideal     0.703  5.51  5.52  3.40 21551
```

Versus pivot_longer()

Before we go on, it's worth pointing out an interesting connection between across() and pivot_longer() ("Lengthening Data" on page 73). In many cases, you perform the same calculations by first pivoting the data and then performing the operations by group rather than by column. For example, take this multifunction summary:

```
df |>
  summarize(across(a:d, list(median = median, mean = mean)))
#> # A tibble: 1 × 8
#>   a_median a_mean b_median b_mean c_median c_mean d_median d_mean
#>      <dbl>  <dbl>    <dbl>  <dbl>    <dbl>  <dbl>    <dbl>  <dbl>
#> 1   0.0380  0.205  -0.0163 0.0910    0.260 0.0716    0.540  0.508
```

We could compute the same values by pivoting longer and then summarizing:

```
long <- df |>
  pivot_longer(a:d) |>
  group_by(name) |>
  summarize(
    median = median(value),
    mean = mean(value)
  )
long
#> # A tibble: 4 × 3
#>   name   median   mean
#>   <chr>   <dbl>  <dbl>
#> 1 a      0.0380 0.205
#> 2 b     -0.0163 0.0910
#> 3 c      0.260  0.0716
#> 4 d      0.540  0.508
```

And if you wanted the same structure as `across()`, you could pivot again:

```
long |>
  pivot_wider(
    names_from = name,
    values_from = c(median, mean),
    names_vary = "slowest",
    names_glue = "{name}_{.value}"
  )
#> # A tibble: 1 × 8
#>   a_median a_mean b_median b_mean c_median c_mean d_median d_mean
#>      <dbl>  <dbl>    <dbl>  <dbl>    <dbl>  <dbl>    <dbl>  <dbl>
#> 1   0.0380  0.205  -0.0163 0.0910    0.260 0.0716    0.540  0.508
```

This is a useful technique to know about because sometimes you'll hit a problem that's not currently possible to solve with `across()`: when you have groups of columns that you want to compute with simultaneously. For example, imagine that our data frame contains both values and weights and we want to compute a weighted mean:

```
df_paired <- tibble(
  a_val = rnorm(10),
  a_wts = runif(10),
  b_val = rnorm(10),
  b_wts = runif(10),
  c_val = rnorm(10),
  c_wts = runif(10),
  d_val = rnorm(10),
  d_wts = runif(10)
)
```

There's currently no way to do this with across(),[4] but it's relatively straightforward with pivot_longer():

```
df_long <- df_paired |>
  pivot_longer(
    everything(),
    names_to = c("group", ".value"),
    names_sep = "_"
  )
df_long
#> # A tibble: 40 × 3
#>   group    val   wts
#>   <chr>  <dbl> <dbl>
#> 1 a      0.715 0.518
#> 2 b     -0.709 0.691
#> 3 c      0.718 0.216
#> 4 d     -0.217 0.733
#> 5 a     -1.09  0.979
#> 6 b     -0.209 0.675
#> # … with 34 more rows

df_long |>
  group_by(group) |>
  summarize(mean = weighted.mean(val, wts))
#> # A tibble: 4 × 2
#>   group    mean
#>   <chr>   <dbl>
#> 1 a      0.126
#> 2 b     -0.0704
#> 3 c     -0.360
#> 4 d     -0.248
```

If needed, you could pivot_wider() this back to the original form.

Exercises

1. Practice your across() skills by:
   a. Computing the number of unique values in each column of palmerpenguins::penguins.
   b. Computing the mean of every column in mtcars.
   c. Grouping diamonds by cut, clarity, and color and then counting the number of observations and computing the mean of each numeric column.
2. What happens if you use a list of functions in across(), but don't name them? How is the output named?
3. Adjust expand_dates() to automatically remove the date columns after they've been expanded. Do you need to embrace any arguments?

[4] Maybe there will be one day, but currently we don't see how.

4. Explain what each step of the pipeline in this function does. What special feature of `where()` are we taking advantage of?

   ```
   show_missing <- function(df, group_vars, summary_vars = everything()) {
     df |>
       group_by(pick({{ group_vars }})) |>
       summarize(
         across({{ summary_vars }}, \(x) sum(is.na(x))),
         .groups = "drop"
       ) |>
       select(where(\(x) any(x > 0)))
   }
   nycflights13::flights |> show_missing(c(year, month, day))
   ```

Reading Multiple Files

In the previous section, you learned how to use `dplyr::across()` to repeat a transformation on multiple columns. In this section, you'll learn how to use `purrr::map()` to do something to every file in a directory. Let's start with a little motivation: imagine you have a directory full of Excel spreadsheets[5] you want to read. You could do it with copy and paste:

```
data2019 <- readxl::read_excel("data/y2019.xlsx")
data2020 <- readxl::read_excel("data/y2020.xlsx")
data2021 <- readxl::read_excel("data/y2021.xlsx")
data2022 <- readxl::read_excel("data/y2022.xlsx")
```

Then use `dplyr::bind_rows()` to combine them all together:

```
data <- bind_rows(data2019, data2020, data2021, data2022)
```

You can imagine that this would get tedious quickly, especially if you had hundreds of files, not just four. The following sections show you how to automate this sort of task. There are three basic steps: use `list.files()` to list all the files in a directory, then use `purrr::map()` to read each of them into a list, and then use `purrr::list_rbind()` to combine them into a single data frame. We'll then discuss how you can handle situations of increasing heterogeneity, where you can't do the same thing to every file.

Listing Files in a Directory

As the name suggests, `list.files()` lists the files in a directory. You'll almost always use three arguments:

- The first argument, `path`, is the directory to look in.

[5] If you instead had a directory of CSV files with the same format, you can use the technique from "Reading Data from Multiple Files" on page 107.

- pattern is a regular expression used to filter the filenames. The most common pattern is something like [.]xlsx$ or [.]csv$ to find all files with a specified extension.
- full.names determines whether the directory name should be included in the output. You almost always want this to be TRUE.

To make our motivating example concrete, this book contains a folder with 12 Excel spreadsheets containing data from the gapminder package. Each file contains one year's worth of data for 142 countries. We can list them all with the appropriate call to list.files():

```
paths <- list.files("data/gapminder", pattern = "[.]xlsx$", full.names = TRUE)
paths
#>  [1] "data/gapminder/1952.xlsx" "data/gapminder/1957.xlsx"
#>  [3] "data/gapminder/1962.xlsx" "data/gapminder/1967.xlsx"
#>  [5] "data/gapminder/1972.xlsx" "data/gapminder/1977.xlsx"
#>  [7] "data/gapminder/1982.xlsx" "data/gapminder/1987.xlsx"
#>  [9] "data/gapminder/1992.xlsx" "data/gapminder/1997.xlsx"
#> [11] "data/gapminder/2002.xlsx" "data/gapminder/2007.xlsx"
```

Lists

Now that we have these 12 paths, we could call read_excel() 12 times to get 12 data frames:

```
gapminder_1952 <- readxl::read_excel("data/gapminder/1952.xlsx")
gapminder_1957 <- readxl::read_excel("data/gapminder/1957.xlsx")
gapminder_1962 <- readxl::read_excel("data/gapminder/1962.xlsx")
 ...,
gapminder_2007 <- readxl::read_excel("data/gapminder/2007.xlsx")
```

But putting each sheet into its own variable is going to make it hard to work with them a few steps down the road. Instead, they'll be easier to work with if we put them into a single object. A list is the perfect tool for this job:

```
files <- list(
  readxl::read_excel("data/gapminder/1952.xlsx"),
  readxl::read_excel("data/gapminder/1957.xlsx"),
  readxl::read_excel("data/gapminder/1962.xlsx"),
  ...,
  readxl::read_excel("data/gapminder/2007.xlsx")
)
```

Now that you have these data frames in a list, how do you get one out? You can use files[[i]] to extract the *i*th element:

```
files[[3]]
#> # A tibble: 142 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         32.0 10267083      853.
#> 2 Albania     Europe       64.8  1728137     2313.
#> 3 Algeria     Africa       48.3 11000948     2551.
#> 4 Angola      Africa      34    4826015     4269.
#> 5 Argentina   Americas    65.1 21283783     7133.
#> 6 Australia   Oceania     70.9 10794968    12217.
#> # … with 136 more rows
```

We'll come back to `[[` in more detail in "Selecting a Single Element with $ and [[" on page 494.

purrr::map() and list_rbind()

The code to collect those data frames in a list "by hand" is basically just as tedious to type as code that reads the files one by one. Happily, we can use `purrr::map()` to make even better use of our `paths` vector. `map()` is similar to `across()`, but instead of doing something to each column in a data frame, it does something to each element of a vector. `map(x, f)` is shorthand for:

```
list(
  f(x[[1]]),
  f(x[[2]]),
  ...,
  f(x[[n]])
)
```

So we can use `map()` to get a list of 12 data frames:

```
files <- map(paths, readxl::read_excel)
length(files)
#> [1] 12

files[[1]]
#> # A tibble: 142 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Albania     Europe       55.2  1282697     1601.
#> 3 Algeria     Africa       43.1  9279525     2449.
#> 4 Angola      Africa       30.0  4232095     3521.
#> 5 Argentina   Americas     62.5 17876956     5911.
#> 6 Australia   Oceania      69.1  8691212    10040.
#> # … with 136 more rows
```

(This is another data structure that doesn't display particularly compactly with `str()`, so you might want to load it into RStudio and inspect it with `View()`).

Now we can use `purrr::list_rbind()` to combine that list of data frames into a single data frame:

```
list_rbind(files)
#> # A tibble: 1,704 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Albania     Europe       55.2  1282697     1601.
#> 3 Algeria     Africa       43.1  9279525     2449.
#> 4 Angola      Africa       30.0  4232095     3521.
#> 5 Argentina   Americas     62.5 17876956     5911.
#> 6 Australia   Oceania      69.1  8691212    10040.
#> # … with 1,698 more rows
```

Or we could do both steps at once in a pipeline:

```
paths |>
  map(readxl::read_excel) |>
  list_rbind()
```

What if we want to pass in extra arguments to `read_excel()`? We use the same technique that we used with `across()`. For example, it's often useful to peak at the first few rows of the data with `n_max = 1`:

```
paths |>
  map(\(path) readxl::read_excel(path, n_max = 1)) |>
  list_rbind()
#> # A tibble: 12 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Afghanistan Asia         30.3  9240934      821.
#> 3 Afghanistan Asia         32.0 10267083      853.
#> 4 Afghanistan Asia         34.0 11537966      836.
#> 5 Afghanistan Asia         36.1 13079460      740.
#> 6 Afghanistan Asia         38.4 14880372      786.
#> # … with 6 more rows
```

This makes it clear that something is missing: there's no `year` column because that value is recorded in the path, not the individual files. We'll tackle that problem next.

Data in the Path

Sometimes the name of the file is data itself. In this example, the filename contains the year, which is not otherwise recorded in the individual files. To get that column into the final data frame, we need to do two things.

First, we name the vector of paths. The easiest way to do this is with the `set_names()` function, which can take a function. Here we use `basename()` to extract just the file name from the full path:

```
paths |> set_names(basename)
#>                  1952.xlsx                  1957.xlsx
#> "data/gapminder/1952.xlsx" "data/gapminder/1957.xlsx"
#>                  1962.xlsx                  1967.xlsx
#> "data/gapminder/1962.xlsx" "data/gapminder/1967.xlsx"
#>                  1972.xlsx                  1977.xlsx
#> "data/gapminder/1972.xlsx" "data/gapminder/1977.xlsx"
#>                  1982.xlsx                  1987.xlsx
#> "data/gapminder/1982.xlsx" "data/gapminder/1987.xlsx"
#>                  1992.xlsx                  1997.xlsx
#> "data/gapminder/1992.xlsx" "data/gapminder/1997.xlsx"
#>                  2002.xlsx                  2007.xlsx
#> "data/gapminder/2002.xlsx" "data/gapminder/2007.xlsx"
```

Those names are automatically carried along by all the map functions, so the list of data frames will have those same names:

```
files <- paths |>
  set_names(basename) |>
  map(readxl::read_excel)
```

That makes this call to `map()` shorthand for:

```
files <- list(
  "1952.xlsx" = readxl::read_excel("data/gapminder/1952.xlsx"),
  "1957.xlsx" = readxl::read_excel("data/gapminder/1957.xlsx"),
  "1962.xlsx" = readxl::read_excel("data/gapminder/1962.xlsx"),
  ...,
  "2007.xlsx" = readxl::read_excel("data/gapminder/2007.xlsx")
)
```

You can also use `[[` to extract elements by name:

```
files[["1962.xlsx"]]
#> # A tibble: 142 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         32.0 10267083      853.
#> 2 Albania     Europe       64.8  1728137     2313.
#> 3 Algeria     Africa       48.3 11000948     2551.
#> 4 Angola      Africa       34    4826015     4269.
#> 5 Argentina   Americas     65.1 21283783     7133.
#> 6 Australia   Oceania      70.9 10794968    12217.
#> # … with 136 more rows
```

Then we use the `names_to` argument to `list_rbind()` to tell it to save the names into a new column called `year` and then use `readr::parse_number()` to extract the number from the string:

```
paths |>
  set_names(basename) |>
  map(readxl::read_excel) |>
  list_rbind(names_to = "year") |>
  mutate(year = parse_number(year))
#> # A tibble: 1,704 × 6
#>    year country     continent lifeExp      pop gdpPercap
#>   <dbl> <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1  1952 Afghanistan Asia         28.8  8425333      779.
#> 2  1952 Albania     Europe       55.2  1282697     1601.
#> 3  1952 Algeria     Africa       43.1  9279525     2449.
#> 4  1952 Angola      Africa       30.0  4232095     3521.
#> 5  1952 Argentina   Americas     62.5 17876956     5911.
#> 6  1952 Australia   Oceania      69.1  8691212    10040.
#> # … with 1,698 more rows
```

In more complicated cases, there might be other variables stored in the directory name, or maybe the filename contains multiple bits of data. In that case, use `set_names()` (without any arguments) to record the full path and then use `tidyr::separate_wider_delim()` and friends to turn them into useful columns:

```
paths |>
  set_names() |>
  map(readxl::read_excel) |>
  list_rbind(names_to = "year") |>
  separate_wider_delim(year, delim = "/", names = c(NA, "dir", "file")) |>
  separate_wider_delim(file, delim = ".", names = c("file", "ext"))
#> # A tibble: 1,704 × 8
#>   dir       file  ext   country     continent lifeExp      pop gdpPercap
#>   <chr>     <chr> <chr> <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 gapminder 1952  xlsx  Afghanistan Asia         28.8  8425333      779.
#> 2 gapminder 1952  xlsx  Albania     Europe       55.2  1282697     1601.
#> 3 gapminder 1952  xlsx  Algeria     Africa       43.1  9279525     2449.
#> 4 gapminder 1952  xlsx  Angola      Africa       30.0  4232095     3521.
#> 5 gapminder 1952  xlsx  Argentina   Americas     62.5 17876956     5911.
#> 6 gapminder 1952  xlsx  Australia   Oceania      69.1  8691212    10040.
#> # … with 1,698 more rows
```

Save Your Work

Now that you've done all this hard work to get to a nice tidy data frame, it's a great time to save your work:

```
gapminder <- paths |>
  set_names(basename) |>
  map(readxl::read_excel) |>
  list_rbind(names_to = "year") |>
  mutate(year = parse_number(year))

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

Now when you come back to this problem in the future, you can read in a single CSV file. For large and richer datasets, using parquet might be a better choice than .csv, as discussed in "The Parquet Format" on page 398.

If you're working in a project, we suggest calling the file that does this sort of data prep work, something like 0-cleanup.R. The 0 in the filename suggests that this should be run before anything else.

If your input data files change over time, you might consider learning a tool like targets to set up your data cleaning code to automatically rerun whenever one of the input files is modified.

Many Simple Iterations

Here we loaded the data directly from disk and were lucky enough to get a tidy dataset. In most cases, you'll need to do some additional tidying, and you have two basic options: you can do one round of iteration with a complex function or do multiple rounds of iteration with simple functions. In our experience, most folks reach first for one complex iteration, but you're often better off doing multiple simple iterations.

For example, imagine that you want to read in a bunch of files, filter out missing values, pivot, and then combine. One way to approach the problem is to write a function that takes a file and does all those steps and then call `map()` once:

```
process_file <- function(path) {
  df <- read_csv(path)

  df |>
    filter(!is.na(id)) |>
    mutate(id = tolower(id)) |>
    pivot_longer(jan:dec, names_to = "month")
}

paths |>
  map(process_file) |>
  list_rbind()
```

Alternatively, you could perform each step of `process_file()` for every file:

```
paths |>
  map(read_csv) |>
  map(\(df) df |> filter(!is.na(id))) |>
  map(\(df) df |> mutate(id = tolower(id))) |>
  map(\(df) df |> pivot_longer(jan:dec, names_to = "month")) |>
  list_rbind()
```

We recommend this approach because it stops you from getting fixated on getting the first file right before moving on to the rest. By considering all of the data when doing tidying and cleaning, you're more likely to think holistically and end up with a higher-quality result.

In this particular example, there's another optimization you could make, by binding all the data frames together earlier. Then you can rely on regular dplyr behavior:

```
paths |>
  map(read_csv) |>
  list_rbind() |>
  filter(!is.na(id)) |>
  mutate(id = tolower(id)) |>
  pivot_longer(jan:dec, names_to = "month")
```

Heterogeneous Data

Unfortunately, sometimes it's not possible to go from `map()` straight to `list_rbind()` because the data frames are so heterogeneous that `list_rbind()` either fails or yields a data frame that's not useful. In that case, it's still useful to start by loading all of the files:

```
files <- paths |>
  map(readxl::read_excel)
```

Then a useful strategy is to capture the structure of the data frames so that you can explore it using your data science skills. One way to do so is with this handy `df_types` function[6] that returns a tibble with one row for each column:

```
df_types <- function(df) {
  tibble(
    col_name = names(df),
    col_type = map_chr(df, vctrs::vec_ptype_full),
    n_miss = map_int(df, \(x) sum(is.na(x)))
  )
}

df_types(gapminder)
#> # A tibble: 6 × 3
#>   col_name  col_type  n_miss
#>   <chr>     <chr>      <int>
#> 1 year      double         0
#> 2 country   character      0
#> 3 continent character      0
#> 4 lifeExp   double         0
#> 5 pop       double         0
#> 6 gdpPercap double         0
```

You can then apply this function to all of the files and maybe do some pivoting to make it easier to see where the differences are. For example, this makes it easy to verify that the gapminder spreadsheets that we've been working with are all quite homogeneous:

```
files |>
  map(df_types) |>
  list_rbind(names_to = "file_name") |>
  select(-n_miss) |>
  pivot_wider(names_from = col_name, values_from = col_type)
#> # A tibble: 12 × 6
#>   file_name country   continent lifeExp pop    gdpPercap
#>   <chr>     <chr>     <chr>     <chr>   <chr>  <chr>
#> 1 1952.xlsx character character double  double double
#> 2 1957.xlsx character character double  double double
#> 3 1962.xlsx character character double  double double
#> 4 1967.xlsx character character double  double double
#> 5 1972.xlsx character character double  double double
#> 6 1977.xlsx character character double  double double
#> # … with 6 more rows
```

If the files have heterogeneous formats, you might need to do more processing before you can successfully merge them. Unfortunately, we're now going to leave you to figure that out on your own, but you might want to read about `map_if()` and `map_at()`. `map_if()` allows you to selectively modify elements of a list based on their values; `map_at()` allows you to selectively modify elements based on their names.

[6] We're not going to explain how it works, but if you look at the docs for the functions used, you should be able to puzzle it out.

Handling Failures

Sometimes the structure of your data might be sufficiently wild that you can't even read all the files with a single command. And then you'll encounter one of the downsides of map(): it succeeds or fails as a whole. map() will either successfully read all of the files in a directory or fail with an error, reading zero files. This is annoying: why does one failure prevent you from accessing all the other successes?

Luckily, purrr comes with a helper to tackle this problem: possibly(). possibly() is what's known as a *function operator*: it takes a function and returns a function with modified behavior. In particular, possibly() changes a function from erroring to returning a value that you specify:

```
files <- paths |>
  map(possibly(\(path) readxl::read_excel(path), NULL))

data <- files |> list_rbind()
```

This works particularly well here because list_rbind(), like many tidyverse functions, automatically ignores NULLs.

Now you have all the data that can be read easily, and it's time to tackle the hard part of figuring out why some files failed to load and what to do about it. Start by getting the paths that failed:

```
failed <- map_vec(files, is.null)
paths[failed]
#> character(0)
```

Then call the import function again for each failure and figure out what went wrong.

Saving Multiple Outputs

In the previous section, you learned about map(), which is useful for reading multiple files into a single object. In this section, we'll now explore sort of the opposite problem: how can you take one or more R objects and save it to one or more files? We'll explore this challenge using three examples:

- Saving multiple data frames into one database
- Saving multiple data frames into multiple .csv files
- Saving multiple plots to multiple .png files

Writing to a Database

Sometimes when working with many files at once, it's not possible to fit all your data into memory at once, and you can't do map(files, read_csv). One approach to deal with this problem is to load your data into a database so you can access just the bits you need with dbplyr.

If you're lucky, the database package you're using will provide a handy function that takes a vector of paths and loads them all into the database. This is the case with duckdb's `duckdb_read_csv()`:

```
con <- DBI::dbConnect(duckdb::duckdb())
duckdb::duckdb_read_csv(con, "gapminder", paths)
```

This would work well here, but we don't have CSV files; instead, we have Excel spreadsheets. So we're going to have to do it "by hand." Learning to do it by hand will also help you when you have a bunch of CSV files and the database that you're working with doesn't have one function that will load them all in.

We need to start by creating a table that will fill in with data. The easiest way to do this is by creating a template, a dummy data frame that contains all the columns we want, but only a sampling of the data. For the gapminder data, we can make that template by reading a single file and adding the year to it:

```
template <- readxl::read_excel(paths[[1]])
template$year <- 1952
template
#> # A tibble: 142 × 6
#>   country     continent lifeExp      pop gdpPercap  year
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl> <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.  1952
#> 2 Albania     Europe       55.2  1282697     1601.  1952
#> 3 Algeria     Africa       43.1  9279525     2449.  1952
#> 4 Angola      Africa       30.0  4232095     3521.  1952
#> 5 Argentina   Americas     62.5 17876956     5911.  1952
#> 6 Australia   Oceania      69.1  8691212    10040.  1952
#> # … with 136 more rows
```

Now we can connect to the database and use `DBI::dbCreateTable()` to turn our template into a database table:

```
con <- DBI::dbConnect(duckdb::duckdb())
DBI::dbCreateTable(con, "gapminder", template)
```

`dbCreateTable()` doesn't use the data in `template`, just the variable names and types. So if we inspect the `gapminder` table now, you'll see that it's empty, but it has the variables we need with the types we expect:

```
con |> tbl("gapminder")
#> # Source:   table<gapminder> [0 x 6]
#> # Database: DuckDB 0.6.1 [root@Darwin 22.3.0:R 4.2.1/:memory:]
#> # … with 6 variables: country <chr>, continent <chr>, lifeExp <dbl>,
#> #   pop <dbl>, gdpPercap <dbl>, year <dbl>
```

Next, we need a function that takes a single file path, reads it into R, and adds the result to the `gapminder` table. We can do that by combining `read_excel()` with `DBI::dbAppendTable()`:

```
append_file <- function(path) {
  df <- readxl::read_excel(path)
  df$year <- parse_number(basename(path))

  DBI::dbAppendTable(con, "gapminder", df)
}
```

Now we need to call `append_file()` once for each element of `paths`. That's certainly possible with `map()`:

```
paths |> map(append_file)
```

But we don't care about the output of `append_file()`, so instead of `map()`, it's slightly nicer to use `walk()`. `walk()` does exactly the same thing as `map()` but throws the output away:

```
paths |> walk(append_file)
```

Now we can see if we have all the data in our table:

```
con |>
  tbl("gapminder") |>
  count(year)
#> # Source:   SQL [?? x 2]
#> # Database: DuckDB 0.6.1 [root@Darwin 22.3.0:R 4.2.1/:memory:]
#>    year     n
#>   <dbl> <dbl>
#> 1  1952   142
#> 2  1957   142
#> 3  1962   142
#> 4  1967   142
#> 5  1972   142
#> 6  1977   142
#> # … with more rows
```

Writing CSV Files

The same basic principle applies if we want to write multiple CSV files, one for each group. Let's imagine that we want to take the `ggplot2::diamonds` data and save one CSV file for each `clarity`. First we need to make those individual datasets. There are many ways you could do that, but there's one way we particularly like: `group_nest()`.

```
by_clarity <- diamonds |>
  group_nest(clarity)

by_clarity
#> # A tibble: 8 × 2
#>   clarity               data
#>   <ord>   <list<tibble[,9]>>
#> 1 I1               [741 × 9]
#> 2 SI2            [9,194 × 9]
#> 3 SI1           [13,065 × 9]
#> 4 VS2           [12,258 × 9]
#> 5 VS1            [8,171 × 9]
#> 6 VVS2           [5,066 × 9]
#> # … with 2 more rows
```

This gives us a new tibble with eight rows and two columns. `clarity` is our grouping variable, and `data` is a list column containing one tibble for each unique value of `clarity`:

```
by_clarity$data[[1]]
#> # A tibble: 741 × 9
#>   carat cut       color depth table price     x     y     z
#>   <dbl> <ord>     <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1  0.32 Premium   E      60.9    58   345  4.38  4.42  2.68
#> 2  1.17 Very Good J      60.2    61  2774  6.83  6.9   4.13
#> 3  1.01 Premium   F      61.8    60  2781  6.39  6.36  3.94
#> 4  1.01 Fair      E      64.5    58  2788  6.29  6.21  4.03
#> 5  0.96 Ideal     F      60.7    55  2801  6.37  6.41  3.88
#> 6  1.04 Premium   G      62.2    58  2801  6.46  6.41  4
#> # … with 735 more rows
```

While we're here, let's create a column that gives the name of the output file, using `mutate()` and `str_glue()`:

```
by_clarity <- by_clarity |> 
  mutate(path = str_glue("diamonds-{clarity}.csv"))

by_clarity
#> # A tibble: 8 × 3
#>   clarity               data path 
#>   <ord>   <list<tibble[,9]>> <glue>
#> 1 I1               [741 × 9] diamonds-I1.csv
#> 2 SI2            [9,194 × 9] diamonds-SI2.csv
#> 3 SI1           [13,065 × 9] diamonds-SI1.csv
#> 4 VS2           [12,258 × 9] diamonds-VS2.csv
#> 5 VS1            [8,171 × 9] diamonds-VS1.csv
#> 6 VVS2           [5,066 × 9] diamonds-VVS2.csv
#> # … with 2 more rows
```

So if we were going to save these data frames by hand, we might write something like:

```
write_csv(by_clarity$data[[1]], by_clarity$path[[1]])
write_csv(by_clarity$data[[2]], by_clarity$path[[2]])
write_csv(by_clarity$data[[3]], by_clarity$path[[3]])
...
write_csv(by_clarity$by_clarity[[8]], by_clarity$path[[8]])
```

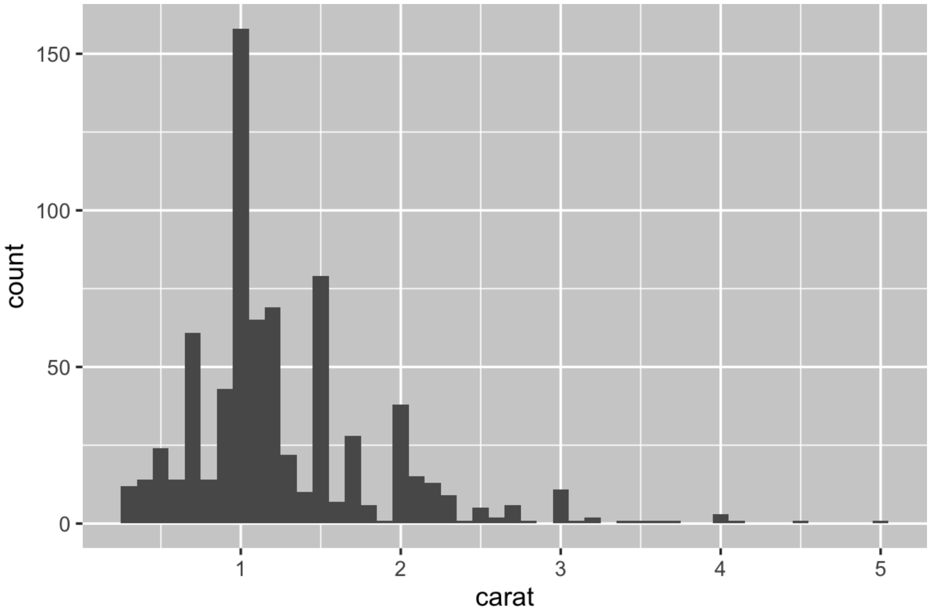
This is a little different from our previous uses of `map()` because there are two arguments that are changing, not just one. That means we need a new function: `map2()`, which varies both the first and second arguments. And because we again don't care about the output, we want `walk2()` rather than `map2()`. That gives us:

```
walk2(by_clarity$data, by_clarity$path, write_csv)
```

Saving Plots

We can take the same basic approach to create many plots. Let's first make a function that draws the plot we want:

```
carat_histogram <- function(df) {
  ggplot(df, aes(x = carat)) + geom_histogram(binwidth = 0.1)
}

carat_histogram(by_clarity$data[[1]])
```



Now we can use `map()` to create a list of many plots[7] and their eventual file paths:

```
by_clarity <- by_clarity |>
  mutate(
    plot = map(data, carat_histogram),
    path = str_glue("clarity-{clarity}.png")
  )
```

Then use `walk2()` with `ggsave()` to save each plot:

```
walk2(
  by_clarity$path,
  by_clarity$plot,
  \(path, plot) ggsave(path, plot, width = 6, height = 6)
)
```

[7] You can print `by_clarity$plot` to get a crude animation—you'll get one plot for each element of `plots`.

This is shorthand for:

```
ggsave(by_clarity$path[[1]], by_clarity$plot[[1]], width = 6, height = 6)
ggsave(by_clarity$path[[2]], by_clarity$plot[[2]], width = 6, height = 6)
ggsave(by_clarity$path[[3]], by_clarity$plot[[3]], width = 6, height = 6)
...
ggsave(by_clarity$path[[8]], by_clarity$plot[[8]], width = 6, height = 6)
```

Summary

In this chapter, you saw how to use explicit iteration to solve three problems that come up frequently when doing data science: manipulating multiple columns, reading multiple files, and saving multiple outputs. But in general, iteration is a superpower: if you know the right iteration technique, you can easily go from fixing one problem to fixing all the problems. Once you've mastered the techniques in this chapter, we highly recommend learning more by reading the "Functionals" chapter of *Advanced R* and consulting the purrr website.

If you know much about iteration in other languages, you might be surprised that we didn't discuss the `for` loop. That's because R's orientation toward data analysis changes how we iterate: in most cases you can rely on an existing idiom to do something to each column or each group. And when you can't, you can often use a functional programming tool like `map()` that does something to each element of a list. However, you will see `for` loops in wild-caught code, so you'll learn about them in the next chapter where we'll discuss some important base R tools.

CHAPTER 27

A Field Guide to Base R

Introduction

To finish off the programming section, we're going to give you a quick tour of the most important base R functions that we don't otherwise discuss in the book. These tools are particularly useful as you do more programming and will help you read code you encounter in the wild.

This is a good place to remind you that the tidyverse is not the only way to solve data science problems. We teach the tidyverse in this book because tidyverse packages share a common design philosophy, increasing the consistency across functions, and making each new function or package a little easier to learn and use. It's not possible to use the tidyverse without using base R, so we've actually already taught you a *lot* of base R functions, including `library()` to load packages; `sum()` and `mean()` for numeric summaries; the factor, date, and POSIXct data types; and of course all the basic operators such as `+`, `-`, `/`, `*`, `|`, `&`, and `!`. What we haven't focused on so far is base R workflows, so we will highlight a few of those in this chapter.

After you read this book, you'll learn other approaches to the same problems using base R, data.table, and other packages. You'll undoubtedly encounter these other approaches when you start reading R code written by others, particularly if you're using StackOverflow. It's 100% OK to write code that uses a mix of approaches, and don't let anyone tell you otherwise!

In this chapter, we'll focus on four big topics: subsetting with `[`, subsetting with `[[` and `$`, using the apply family of functions, and using `for` loops. To finish off, we'll briefly discuss two essential plotting functions.

Prerequisites

This package focuses on base R so it doesn't have any real prerequisites, but we'll load the tidyverse to explain some of the differences:

```
library(tidyverse)
```

Selecting Multiple Elements with [

`[` is used to extract subcomponents from vectors and data frames and is called like `x[i]` or `x[i, j]`. In this section, we'll introduce you to the power of `[`, first showing you how you can use it with vectors, and then showing how the same principles extend in a straightforward way to 2D structures like data frames. We'll then help you cement that knowledge by showing how various dplyr verbs are special cases of `[`.

Subsetting Vectors

There are five main types of things that you can subset a vector with, i.e., that can be the `i` in `x[i]`:

- *A vector of positive integers*. Subsetting with positive integers keeps the elements at those positions:

  ```
  x <- c("one", "two", "three", "four", "five")
  x[c(3, 2, 5)]
  #> [1] "three" "two"   "five"
  ```

  By repeating a position, you can actually make a longer output than input, making the term "subsetting" a bit of a misnomer:

  ```
  x[c(1, 1, 5, 5, 5, 2)]
  #> [1] "one"  "one"  "five" "five" "five" "two"
  ```

- *A vector of negative integers*. Negative values drop the elements at the specified positions:

  ```
  x[c(-1, -3, -5)]
  #> [1] "two"  "four"
  ```

- *A logical vector*. Subsetting with a logical vector keeps all values corresponding to a `TRUE` value. This is most often useful in conjunction with the comparison functions:

  ```
  x <- c(10, 3, NA, 5, 8, 1, NA)

  # All non-missing values of x
  x[!is.na(x)]
  #> [1] 10  3  5  8  1

  # All even (or missing!) values of x
  x[x %% 2 == 0]
  #> [1] 10 NA  8 NA
  ```

  Unlike `filter()`, `NA` indices will be included in the output as `NA`s.

- *A character vector*. If you have a named vector, you can subset it with a character vector:

  ```
  x <- c(abc = 1, def = 2, xyz = 5)
  x[c("xyz", "def")]
  #> xyz def
  #>   5   2
  ```

  As with subsetting with positive integers, you can use a character vector to duplicate individual entries.

- *Nothing*. The final type of subsetting is nothing, `x[]`, which returns the complete `x`. This is not useful for subsetting vectors, but as we'll see shortly, it is useful when subsetting 2D structures like tibbles.

Subsetting Data Frames

There are quite a few different ways[1] that you can use `[` with a data frame, but the most important way is to select rows and columns independently with `df[rows, cols]`. Here `rows` and `cols` are vectors as described earlier. For example, `df[rows, ]` and `df[, cols]` select just rows or just columns, using the empty subset to preserve the other dimension.

Here are a couple of examples:

```
df <- tibble(
  x = 1:3,
  y = c("a", "e", "f"),
  z = runif(3)
)

# Select first row and second column
df[1, 2]
#> # A tibble: 1 × 1
#>   y
#>   <chr>
#> 1 a

# Select all rows and columns x and y
df[, c("x" , "y")]
#> # A tibble: 3 × 2
#>       x y
#>   <int> <chr>
#> 1     1 a
#> 2     2 e
#> 3     3 f

# Select rows where `x` is greater than 1 and all columns
df[df$x > 1, ]
#> # A tibble: 2 × 3
#>       x y         z
```

[1] Read the Selecting multiple elements section in *Advanced R* to see how you can also subset a data frame like it is a 1D object and how you can subset it with a matrix.

```
#>   <int> <chr> <dbl>
#> 1     2 e     0.834
#> 2     3 f     0.601
```

We'll come back to `$` shortly, but you should be able to guess what `df$x` does from the context: it extracts the `x` variable from `df`. We need to use it here because `[` doesn't use tidy evaluation, so you need to be explicit about the source of the `x` variable.

There's an important difference between tibbles and data frames when it comes to `[`. In this book, we've mainly used tibbles, which *are* data frames, but they tweak some behaviors to make your life a little easier. In most places, you can use "tibble" and "data frame" interchangeably, so when we want to draw particular attention to R's built-in data frame, we'll write `data.frame`. If `df` is a `data.frame`, then `df[, cols]` will return a vector if `col` selects a single column and will return a data frame if it selects more than one column. If `df` is a tibble, then `[` will always return a tibble.

```
df1 <- data.frame(x = 1:3)
df1[, "x"]
#> [1] 1 2 3

df2 <- tibble(x = 1:3)
df2[, "x"]
#> # A tibble: 3 × 1
#>       x
#>   <int>
#> 1     1
#> 2     2
#> 3     3
```

One way to avoid this ambiguity with `data.frames` is to explicitly specify `drop = FALSE`:

```
df1[, "x" , drop = FALSE]
#>   x
#> 1 1
#> 2 2
#> 3 3
```

dplyr Equivalents

Several dplyr verbs are special cases of `[`:

- `filter()` is equivalent to subsetting the rows with a logical vector, taking care to exclude missing values:

  ```
  df <- tibble(
    x = c(2, 3, 1, 1, NA),
    y = letters[1:5],
    z = runif(5)
  )
  df |> filter(x > 1)

  # same as
  df[!is.na(df$x) & df$x > 1, ]
  ```

Another common technique in the wild is to use which() for its side effect of dropping missing values: df[which(df$x > 1),].

- arrange() is equivalent to subsetting the rows with an integer vector, usually created with order():

  ```
  df |> arrange(x, y)

  # same as
  df[order(df$x, df$y), ]
  ```

  You can use order(decreasing = TRUE) to sort all columns in descending order or -rank(col) to sort columns in decreasing order individually.

- Both select() and relocate() are similar to subsetting the columns with a character vector:

  ```
  df |> select(x, z)

  # same as
  df[, c("x", "z")]
  ```

Base R also provides a function that combines the features of filter() and select()[2] called subset():

```
df |>
  filter(x > 1) |>
  select(y, z)
#> # A tibble: 2 × 2
#>   y           z
#>   <chr>   <dbl>
#> 1 a     0.157
#> 2 b     0.00740

# same as
df |> subset(x > 1, c(y, z))
```

This function was the inspiration for much of dplyr's syntax.

Exercises

1. Create functions that take a vector as input and return:
   a. The elements at even-numbered positions
   b. Every element except the last value
   c. Only even values (and no missing values)
2. Why is x[-which(x > 0)] not the same as x[x <= 0]? Read the documentation for which() and do some experiments to figure it out.

[2] But it doesn't handle grouped data frames differently, and it doesn't support selection helper functions like starts_with().

Selecting a Single Element with $ and [[

[, which selects many elements, is paired with [[and $, which extract a single element. In this section, we'll show you how to use [[and $ to pull columns out of data frames, discuss a couple more differences between data.frames and tibbles, and emphasize some important differences between [and [[when used with lists.

Data Frames

[[and $ can be used to extract columns out of a data frame. [[can access by position or by name, and $ is specialized for access by name:

```
tb <- tibble(
  x = 1:4,
  y = c(10, 4, 1, 21)
)

# by position
tb[[1]]
#> [1] 1 2 3 4

# by name
tb[["x"]]
#> [1] 1 2 3 4
tb$x
#> [1] 1 2 3 4
```

They can also be used to create new columns, the base R equivalent of mutate():

```
tb$z <- tb$x + tb$y
tb
#> # A tibble: 4 × 3
#>       x     y     z
#>   <int> <dbl> <dbl>
#> 1     1    10    11
#> 2     2     4     6
#> 3     3     1     4
#> 4     4    21    25
```

There are several other base R approaches to creating new columns including with transform(), with(), and within(). Hadley collected a few examples.

Using $ directly is convenient when performing quick summaries. For example, if you just want to find the size of the biggest diamond or the possible values of cut, there's no need to use summarize():

```
max(diamonds$carat)
#> [1] 5.01

levels(diamonds$cut)
#> [1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

dplyr also provides an equivalent to `[[`/`$` that we didn't mention in Chapter 3: `pull()`. `pull()` takes either a variable name or a variable position and returns just that column. That means we could rewrite the previous code to use the pipe:

```
diamonds |> pull(carat) |> mean()
#> [1] 0.7979397

diamonds |> pull(cut) |> levels()
#> [1] "Fair"      "Good"      "Very Good" "Premium"   "Ideal"
```

Tibbles

There are a couple of important differences between tibbles and base `data.frames` when it comes to `$`. Data frames match the prefix of any variable names (so-called *partial matching*) and don't complain if a column doesn't exist:

```
df <- data.frame(x1 = 1)
df$x
#> Warning in df$x: partial match of 'x' to 'x1'
#> [1] 1
df$z
#> NULL
```

Tibbles are more strict: they only ever match variable names exactly and they will generate a warning if the column you are trying to access doesn't exist:

```
tb <- tibble(x1 = 1)

tb$x
#> Warning: Unknown or uninitialised column: `x`.
#> NULL
tb$z
#> Warning: Unknown or uninitialised column: `z`.
#> NULL
```

For this reason we sometimes joke that tibbles are lazy and surly: they do less and complain more.

Lists

`[[` and `$` are also really important for working with lists, and it's important to understand how they differ from `[`. Let's illustrate the differences with a list named `l`:

```
l <- list(
  a = 1:3,
  b = "a string",
  c = pi,
  d = list(-1, -5)
)
```

- `[` extracts a sublist. It doesn't matter how many elements you extract, the result will always be a list.

  ```
  str(l[1:2])
  #> List of 2
  #>  $ a: int [1:3] 1 2 3
  #>  $ b: chr "a string"

  str(l[1])
  #> List of 1
  #>  $ a: int [1:3] 1 2 3

  str(l[4])
  #> List of 1
  #>  $ d:List of 2
  #>   ..$ : num -1
  #>   ..$ : num -5
  ```

  Like with vectors, you can subset with a logical, integer, or character vector.

- `[[` and `$` extract a single component from a list. They remove a level of hierarchy from the list.

  ```
  str(l[[1]])
  #>  int [1:3] 1 2 3

  str(l[[4]])
  #> List of 2
  #>  $ : num -1
  #>  $ : num -5

  str(l$a)
  #>  int [1:3] 1 2 3
  ```

The difference between `[` and `[[` is particularly important for lists because `[[` drills down into the list, while `[` returns a new, smaller list. To help you remember the difference, take a look at the unusual pepper shaker shown in Figure 27-1. If this pepper shaker is your list `pepper`, then `pepper[1]` is a pepper shaker containing a single pepper packet. `pepper[2]` would look the same but would contain the second packet. `pepper[1:2]` would be a pepper shaker containing two pepper packets. `pepper[[1]]` would extract the pepper packet itself.

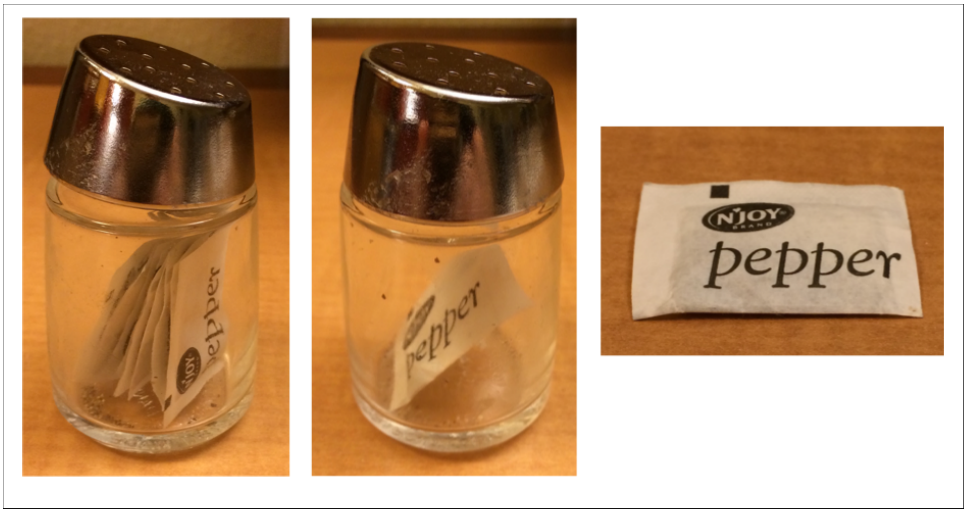


Figure 27-1. (Left) A pepper shaker that Hadley once found in his hotel room. (Middle) `pepper[1]`. (Right) `pepper[[1]]`.

This same principle applies when you use 1D `[` with a data frame: `df["x"]` returns a one-column data frame, and `df[["x"]]` returns a vector.

Exercises

1. What happens when you use `[[` with a positive integer that's bigger than the length of the vector? What happens when you subset with a name that doesn't exist?
2. What would `pepper[[1]][1]` be? What about `pepper[[1]][[1]]`?

Apply Family

In Chapter 26, you learned tidyverse techniques for iteration like `dplyr::across()` and the map family of functions. In this section, you'll learn about their base equivalents, the *apply family*. In this context, apply and map are synonyms because another way of saying "map a function over each element of a vector" is "apply a function over each element of a vector." Here we'll give you a quick overview of this family so you can recognize them in the wild.

The most important member of this family is lapply(), which is similar to purrr::map().[3] In fact, because we haven't used any of map()'s more advanced features, you can replace every map() call in Chapter 26 with lapply().

There's no exact base R equivalent to across(), but you can get close by using [with lapply(). This works because under the hood, data frames are lists of columns, so calling lapply() on a data frame applies the function to each column.

```
df <- tibble(a = 1, b = 2, c = "a", d = "b", e = 4)

# First find numeric columns
num_cols <- sapply(df, is.numeric)
num_cols
#>     a     b     c     d     e
#>  TRUE  TRUE FALSE FALSE  TRUE

# Then transform each column with lapply() then replace the original values
df[, num_cols] <- lapply(df[, num_cols, drop = FALSE], \(x) x * 2)
df
#> # A tibble: 1 × 5
#>       a     b c     d         e
#>   <dbl> <dbl> <chr> <chr> <dbl>
#> 1     2     4 a     b         8
```

The previous code uses a new function, sapply(). It's similar to lapply(), but it always tries to simplify the result, which is the reason for the s in its name, here producing a logical vector instead of a list. We don't recommend using it for programming, because the simplification can fail and give you an unexpected type, but it's usually fine for interactive use. purrr has a similar function called map_vec() that we didn't mention in Chapter 26.

Base R provides a stricter version of sapply() called vapply(), short for *v*ector apply. It takes an additional argument that specifies the expected type, ensuring that simplification occurs the same way regardless of the input. For example, we could replace the previous sapply() call with this vapply() where we specify that we expect is.numeric() to return a logical vector of length 1:

```
vapply(df, is.numeric, logical(1))
#>     a     b     c     d     e
#>  TRUE  TRUE FALSE FALSE  TRUE
```

The distinction between sapply() and vapply() is really important when they're inside a function (because it makes a big difference to the function's robustness to unusual inputs), but it doesn't usually matter in data analysis.

Another important member of the apply family is tapply(), which computes a single grouped summary:

[3] It just lacks convenient features such as progress bars and reporting which element caused the problem if there's an error.

```
diamonds |>
  group_by(cut) |>
  summarize(price = mean(price))
#> # A tibble: 5 × 2
#>   cut       price
#>   <ord>     <dbl>
#> 1 Fair      4359.
#> 2 Good      3929.
#> 3 Very Good 3982.
#> 4 Premium   4584.
#> 5 Ideal     3458.

tapply(diamonds$price, diamonds$cut, mean)
#>      Fair      Good Very Good   Premium     Ideal
#>  4358.758  3928.864  3981.760  4584.258  3457.542
```

Unfortunately, `tapply()` returns its results in a named vector, which requires some gymnastics if you want to collect multiple summaries and grouping variables into a data frame (it's certainly possible to not do this and just work with free-floating vectors, but in our experience that just delays the work). If you want to see how you might use `tapply()` or other base techniques to perform other grouped summaries, Hadley has collected a few techniques in a gist.

The final member of the apply family is the titular `apply()`, which works with matrices and arrays. In particular, watch out for `apply(df, 2, something)`, which is a slow and potentially dangerous way of doing `lapply(df, something)`. This rarely comes up in data science because we usually work with data frames and not matrices.

for Loops

`for` loops are the fundamental building block of iteration that both the apply and map families use under the hood. `for` loops are powerful and general tools that are important to learn as you become a more experienced R programmer. The basic structure of a `for` loop looks like this:

```
for (element in vector) {
  # do something with element
}
```

The most straightforward use of `for` loops is to achieve the same effect as `walk()`: call some function with a side effect on each element of a list. For example, in "Writing to a Database" on page 483, instead of using `walk()`:

```
paths |> walk(append_file)
```

we could have used a `for` loop:

```
for (path in paths) {
  append_file(path)
}
```

Things get a little trickier if you want to save the output of the for loop, for example reading all of the Excel files in a directory like we did in Chapter 26:

```
paths <- dir("data/gapminder", pattern = "\\.xlsx$", full.names = TRUE)
files <- map(paths, readxl::read_excel)
```

There are a few different techniques that you can use, but we recommend being explicit about what the output is going to look like up front. In this case, we're going to want a list the same length as paths, which we can create with vector():

```
files <- vector("list", length(paths))
```

Then instead of iterating over the elements of paths, we'll iterate over their indices, using seq_along() to generate one index for each element of paths:

```
seq_along(paths)
#>  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

Using the indices is important because it allows us to link to each position in the input with the corresponding position in the output:

```
for (i in seq_along(paths)) {
  files[[i]] <- readxl::read_excel(paths[[i]])
}
```

To combine the list of tibbles into a single tibble, you can use do.call() + rbind():

```
do.call(rbind, files)
#> # A tibble: 1,704 × 5
#>   country     continent lifeExp      pop gdpPercap
#>   <chr>       <chr>       <dbl>    <dbl>     <dbl>
#> 1 Afghanistan Asia         28.8  8425333      779.
#> 2 Albania     Europe       55.2  1282697     1601.
#> 3 Algeria     Africa       43.1  9279525     2449.
#> 4 Angola      Africa       30.0  4232095     3521.
#> 5 Argentina   Americas     62.5 17876956     5911.
#> 6 Australia   Oceania      69.1  8691212    10040.
#> # … with 1,698 more rows
```

Rather than making a list and saving the results as we go, a simpler approach is to build up the data frame piece by piece:

```
out <- NULL
for (path in paths) {
  out <- rbind(out, readxl::read_excel(path))
}
```

We recommend avoiding this pattern because it can become slow when the vector is long. This is the source of the persistent canard that for loops are slow: they're not, but iteratively growing a vector is.

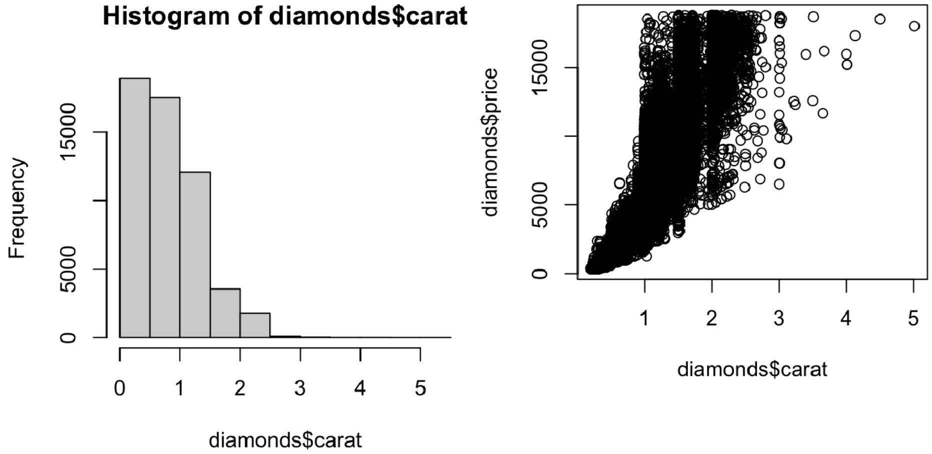
Plots

Many R users who don't otherwise use the tidyverse prefer ggplot2 for plotting due to helpful features such as sensible defaults, automatic legends, and a modern look.

However, base R plotting functions can still be useful because they're so concise—it takes very little typing to do a basic exploratory plot.

There are two main types of base plot you'll see in the wild: scatterplots and histograms, produced with `plot()` and `hist()`, respectively. Here's a quick example from the `diamonds` dataset:

```
# Left
hist(diamonds$carat)

# Right
plot(diamonds$carat, diamonds$price)
```



Note that base plotting functions work with vectors, so you need to pull columns out of the data frame using `$` or some other technique.

Summary

In this chapter, we showed you a selection of base R functions useful for subsetting and iteration. Compared to approaches discussed elsewhere in the book, these functions tend to have more of a "vector" flavor than a "data frame" flavor because base R functions tend to take individual vectors, rather than a data frame and some column specification. This often makes life easier for programming and so becomes more important as you write more functions and begin to write your own packages.

This chapter concludes the programming section of the book. You made a solid start on your journey to becoming not just a data scientist who uses R, but a data scientist who can *program* in R. We hope these chapters have sparked your interest in programming and that you're looking forward to learning more outside of this book.

PART VI
Communicate

So far, you've learned the tools to get your data into R, tidy it into a form convenient for analysis, and then understand your data through transformation and visualization. However, it doesn't matter how great your analysis is unless you can explain it to others: you need to *communicate* your results.

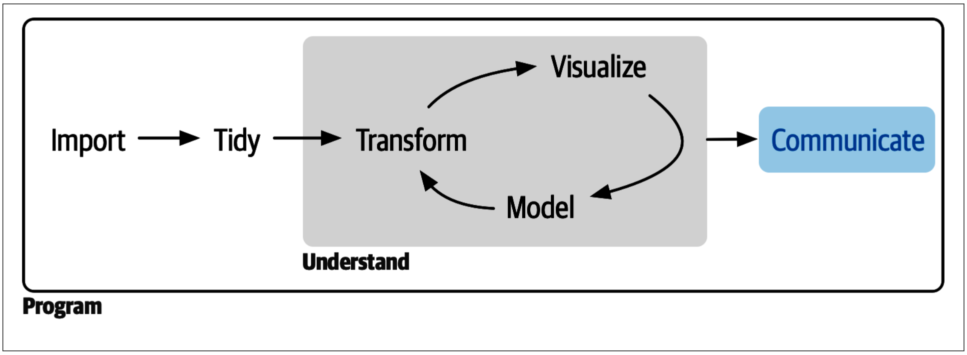


Figure VI-1. Communication is the final part of the data science process; if you can't communicate your results to other humans, it doesn't matter how great your analysis is.

Communication is the theme of the following two chapters:

- In Chapter 28, you will learn about Quarto, a tool for integrating prose, code, and results. You can use Quarto for analyst-to-analyst communication as well as analyst-to-decision-maker communication. Thanks to the power of Quarto formats, you can even use the same document for both purposes.

- In Chapter 29, you'll learn a little about the many other varieties of outputs you can produce using Quarto, including dashboards, websites, and books.

These chapters focus mostly on the technical mechanics of communication, not the really hard problems of communicating your thoughts to other humans. However, there are lot of other great books about communication, which we'll point you to at the end of each chapter.

CHAPTER 28
Quarto

Introduction

Quarto provides a unified authoring framework for data science, combining your code, its results, and your prose. Quarto documents are fully reproducible and support dozens of output formats, such as PDFs, Word files, presentations, and more.

Quarto files are designed to be used in three ways:

- For communicating to decision-makers, who want to focus on the conclusions, not the code behind the analysis
- For collaborating with other data scientists (including future you!), who are interested in both your conclusions and how you reached them (i.e., the code)
- As an environment in which to *do* data science, as a modern-day lab notebook where you can capture not only what you did but also what you were thinking

Quarto is a command-line interface tool, not an R package. This means that help is, by and large, not available through `?`. Instead, as you work through this chapter and use Quarto in the future, you should refer to the Quarto documentation.

If you're an R Markdown user, you might be thinking, "Quarto sounds a lot like R Markdown." You're not wrong! Quarto unifies the functionality of many packages from the R Markdown ecosystem (rmarkdown, bookdown, distill, xaringan, etc.) into a single consistent system as well as extends it with native support for multiple programming languages such as Python and Julia in addition to R. In a way, Quarto reflects everything that was learned from expanding and supporting the R Markdown ecosystem for a decade.

Prerequisites

You need the Quarto command-line interface (Quarto CLI), but you don't need to explicitly install it or load it, as RStudio automatically does both when needed.

Quarto Basics

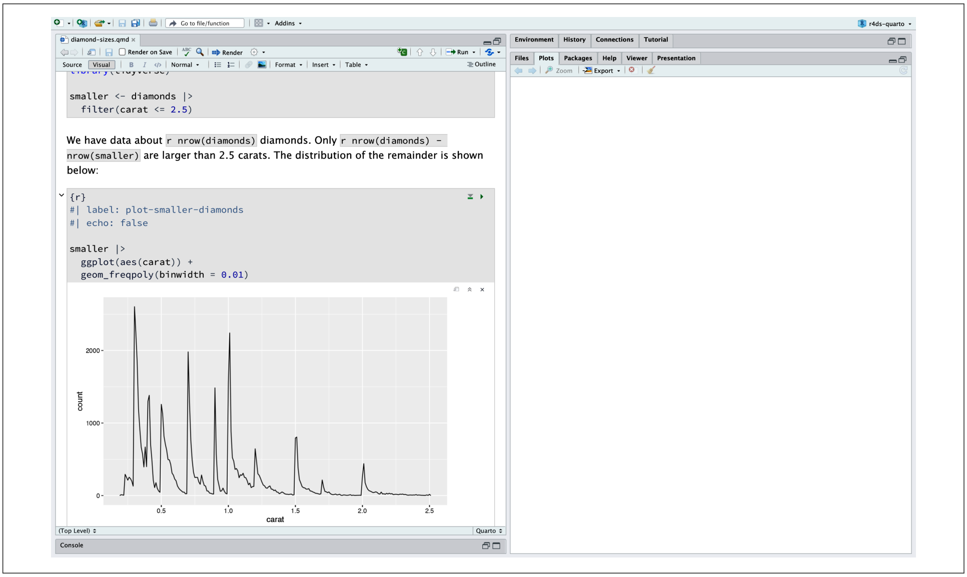
This is a Quarto file—a plain-text file that has the extension `.qmd`:

~~~
---
title: "Diamond sizes"
date: 2022-09-12
format: html
---

```{r}
#| label: setup
#| include: false

library(tidyverse)

smaller <- diamonds |>
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds.
Only `r nrow(diamonds) - nrow(smaller)` are larger than 2.5 carats.
The distribution of the remainder is shown below:

```{r}
#| label: plot-smaller-diamonds
#| echo: false

smaller |>
  ggplot(aes(x = carat)) +
  geom_freqpoly(binwidth = 0.01)
```
~~~

It contains three important types of content:

- An (optional) *YAML header* surrounded by `---`
- *Chunks* of R code surrounded by ```` ``` ````
- Text mixed with simple text formatting like `# heading` and `_italics_`

Figure 28-1 shows a `.qmd` document in RStudio with a notebook interface where code and output are interleaved. You can run each code chunk by clicking the Run icon (it looks like a play button at the top of the chunk) or by pressing Cmd/Ctrl+Shift+Enter. RStudio executes the code and displays the results inline with the code.



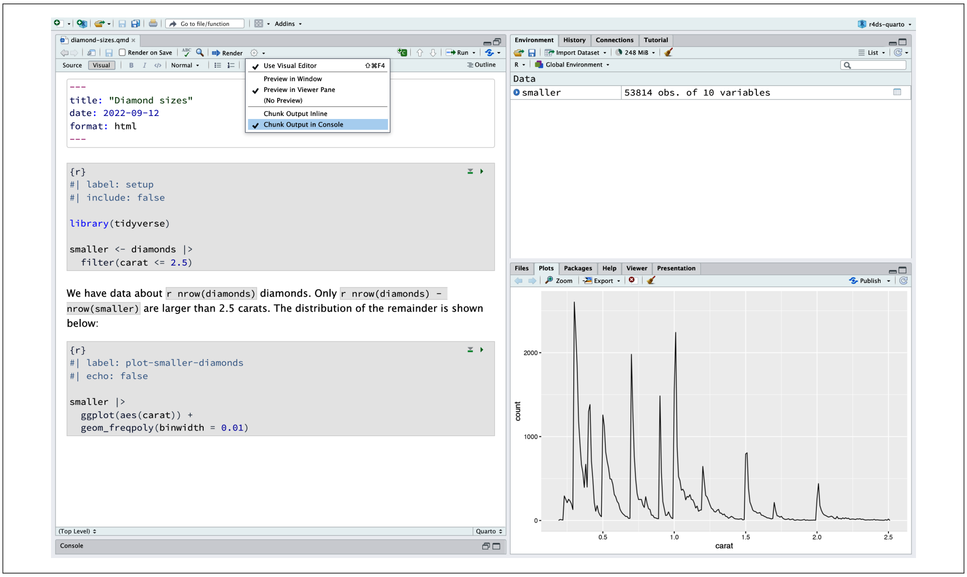
```
smaller <- diamonds |>
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds. Only `r nrow(diamonds) - nrow(smaller)` are larger than 2.5 carats. The distribution of the remainder is shown below:

```
{r}
#| label: plot-smaller-diamonds
#| echo: false

smaller |>
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```

Figure 28-1. A Quarto document in RStudio. Code and output are interleaved in the document, with the plot output appearing right underneath the code.

If you don't like seeing your plots and output in your document and would rather make use of RStudio's Console and Plots panes, you can click the gear icon next to Render and switch to Chunk Output in Console, as shown in Figure 28-2.



Data
smaller 53814 obs. of 10 variables

```
---
title: "Diamond sizes"
date: 2022-09-12
format: html
---
```

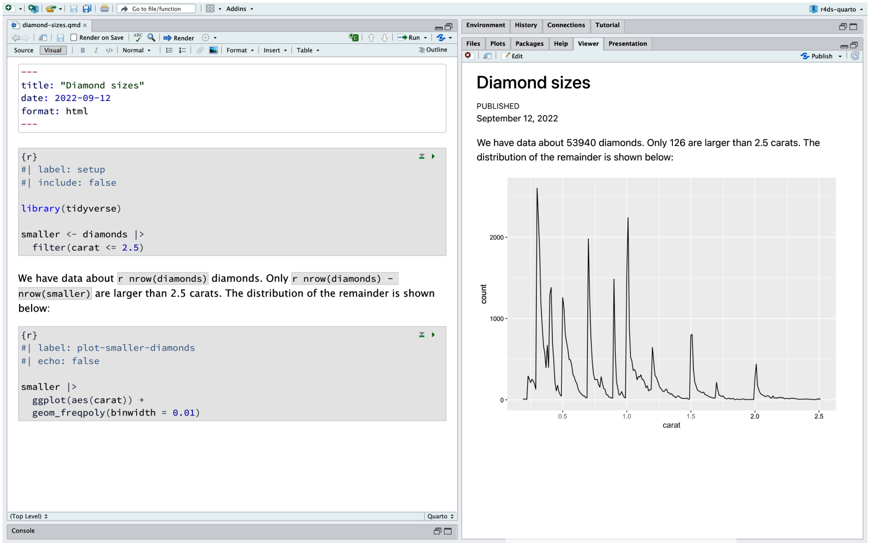
```
{r}
#| label: setup
#| include: false

library(tidyverse)

smaller <- diamonds |>
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds. Only `r nrow(diamonds) - nrow(smaller)` are larger than 2.5 carats. The distribution of the remainder is shown below:

```
{r}
#| label: plot-smaller-diamonds
#| echo: false

smaller |>
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```

Figure 28-2. A Quarto document in RStudio with the plot output in the Plots pane.

To produce a complete report containing all text, code, and results, click Render or press Cmd/Ctrl+Shift+K. You can also do this programmatically with `quarto::quarto_render("diamond-sizes.qmd")`. This will display the report in the viewer pane as shown in Figure 28-3 and create an HTML file.



```
---
title: "Diamond sizes"
date: 2022-09-12
format: html
---

{r}
#| label: setup
#| include: false

library(tidyverse)

smaller <- diamonds |>
  filter(carat <= 2.5)
```

We have data about `r nrow(diamonds)` diamonds. Only `r nrow(diamonds) - nrow(smaller)` are larger than 2.5 carats. The distribution of the remainder is shown below:

```
{r}
#| label: plot-smaller-diamonds
#| echo: false

smaller |>
  ggplot(aes(carat)) +
  geom_freqpoly(binwidth = 0.01)
```

Diamond sizes

PUBLISHED
September 12, 2022

We have data about 53940 diamonds. Only 126 are larger than 2.5 carats. The distribution of the remainder is shown below:

Figure 28-3. A Quarto document in RStudio with the rendered document in the Viewer pane.

When you render the document, Quarto sends the `.qmd` file to knitr, which executes all of the code chunks and creates a new Markdown (`.md`) document that includes the code and its output. The Markdown file generated by knitr is then processed by pandoc, which is responsible for creating the finished file. Figure 28-4 shows this process. The advantage of this two-step workflow is that you can create a very wide range of output formats, as you'll learn about in Chapter 29.

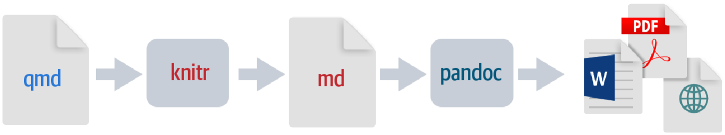


Figure 28-4. Diagram of Quarto workflow from qmd, to knitr, to md, to pandoc, to output in PDF, MS Word, or HTML formats.

To get started with your own `.qmd` file, select File > New File > Quarto Document… in the menu bar. RStudio will launch a wizard that you can use to prepopulate your file with useful content that reminds you how the key features of Quarto work.

The following sections dive into the three components of a Quarto document in more details: the Markdown text, the code chunks, and the YAML header.

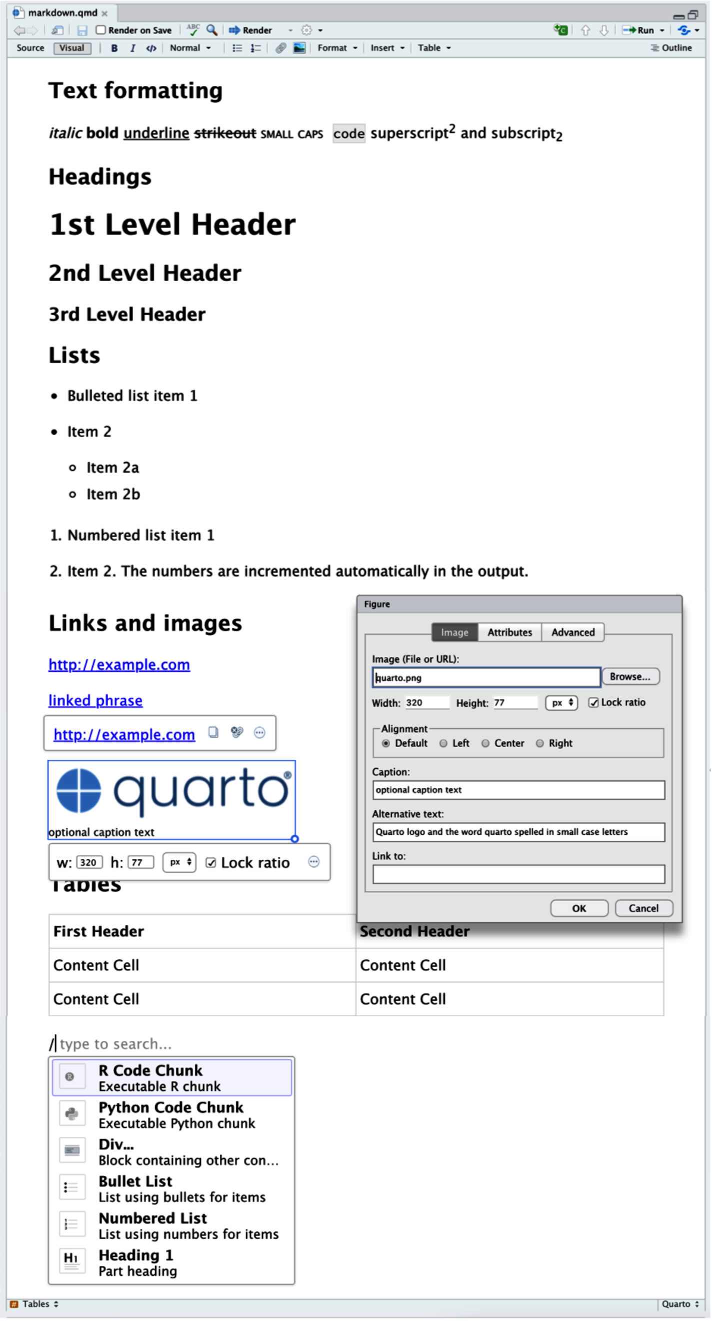
Exercises

1. Create a new Quarto document by selecting File > New File > Quarto Document. Read the instructions. Practice running the chunks individually. Then render the document by clicking the appropriate button and then by using the appropriate keyboard shortcut. Verify that you can modify the code, rerun it, and see modified output.
2. Create one new Quarto document for each of the three built-in formats: HTML, PDF, and Word. Render each of the three documents. How do the outputs differ? How do the inputs differ? (You may need to install LaTeX to build the PDF output—RStudio will prompt you if this is necessary.)

Visual Editor

The visual editor in RStudio provides a WYSIWYM interface for authoring Quarto documents. Under the hood, prose in Quarto documents (`.qmd` files) is written in Markdown, a lightweight set of conventions for formatting plain-text files. In fact, Quarto uses Pandoc markdown (a slightly extended version of Markdown that Quarto understands), including tables, citations, cross-references, footnotes, divs/spans, definition lists, attributes, raw HTML/TeX, and more, as well as support for executing code cells and viewing their output inline. While Markdown is designed to be easy to read and write, as you will see in "Source Editor" on page 511, it still requires learning new syntax. Therefore, if you're new to computational documents like `.qmd` files but have experience using tools like Google Docs or MS Word, the easiest way to get started with Quarto in RStudio is the visual editor.

In the visual editor either you can use the buttons on the menu bar to insert images, tables, cross-references, etc., or you can use the catch-all Cmd/Ctrl+/ shortcut to insert just about anything. If you are at the beginning of a line (as shown in Figure 28-5), you can also enter just / to invoke the shortcut.



Text formatting

italic **bold** underline ~~strikeout~~ SMALL CAPS `code` superscript^2 and subscript$_2$

Headings

1st Level Header

2nd Level Header

3rd Level Header

Lists

- Bulleted list item 1
- Item 2
  - Item 2a
  - Item 2b

1. Numbered list item 1
2. Item 2. The numbers are incremented automatically in the output.

Links and images

http://example.com

linked phrase

http://example.com

optional caption text

w: 320 h: 77 px ☑ Lock ratio

Figure

Image | Attributes | Advanced

Image (File or URL): quarto.png

Width: 320 Height: 77 px ☑ Lock ratio

Alignment ● Default ○ Left ○ Center ○ Right

Caption: optional caption text

Alternative text: Quarto logo and the word quarto spelled in small case letters

Link to:

OK Cancel

Tables

First Header	Second Header
Content Cell	Content Cell
Content Cell	Content Cell

/type to search...

R Code Chunk
Executable R chunk

Python Code Chunk
Executable Python chunk

Div...
Block containing other con...

Bullet List
List using bullets for items

Numbered List
List using numbers for items

Heading 1
Part heading

Figure 28-5. Quarto visual editor.

Inserting images and customizing how they are displayed is also facilitated with the visual editor. Either you can paste an image from your clipboard directly into the visual editor (and RStudio will place a copy of that image in the project directory and link to it) or you can use the visual editor's Insert > Figure/Image menu to browse to the image you want to insert or paste its URL. In addition, using the same menu you can resize the image as well as add a caption, alternative text, and a link.

The visual editor has many more features that we haven't enumerated here that you might find useful as you gain experience authoring with it.

Most importantly, while the visual editor displays your content with formatting, under the hood, it saves your content in plain Markdown, and you can switch back and forth between the visual and source editors to view and edit your content using either tool.

Exercises

1. Re-create the document in Figure 28-5 using the visual editor.
2. Using the visual editor, insert a code chunk using the Insert menu and then the insert anything tool.
3. Using the visual editor, figure out how to:
    a. Add a footnote.
    b. Add a horizontal rule.
    c. Add a block quote.
4. In the visual editor, select Insert > Citation and insert a citation to the paper titled "Welcome to the Tidyverse" using its digital object identifier (DOI), which is 10.21105/joss.01686. Render the document and observe how the reference shows up in the document. What change do you observe in the YAML of your document?

Source Editor

You can also edit Quarto documents using the source editor in RStudio, without the assist of the visual editor. While the visual editor will feel familiar to those with experience writing in tools like Google Docs, the source editor will feel familiar to those with experience writing R scripts or R Markdown documents. The source editor can also be useful for debugging any Quarto syntax errors since it's often easier to catch these in plain text.

The following guide shows how to use Pandoc's Markdown for authoring Quarto documents in the source editor:

```
## Text formatting

*italic* **bold** ~~strikeout~~ `code`

superscript^2^ subscript~2~

[underline]{.underline} [small caps]{.smallcaps}

## Headings

# 1st Level Header

## 2nd Level Header

### 3rd Level Header

## Lists

-   Bulleted list item 1

-   Item 2

    -   Item 2a

    -   Item 2b

1.  Numbered list item 1

2.  Item 2.
    The numbers are incremented automatically in the output.

## Links and images

<http://example.com>

[linked phrase](http://example.com)

![optional caption text](quarto.png){
  fig-alt="Quarto logo and the word quarto spelled in small case letters"}

## Tables

| First Header | Second Header |
|--------------|---------------|
| Content Cell | Content Cell  |
| Content Cell | Content Cell  |
```

The best way to learn these is simply to try them. It will take a few days, but soon they will become second nature, and you won't need to think about them. If you forget, you can get to a handy reference sheet with Help > Markdown Quick Reference.

Exercises

1. Practice what you've learned by creating a brief résumé. The title should be your name, and you should include headings for (at least) education or employment. Each of the sections should include a bulleted list of jobs/degrees. Highlight the year in bold.
2. Using the source editor and the Markdown quick reference, figure out how to:
   a. Add a footnote.
   b. Add a horizontal rule.
   c. Add a block quote.
3. Copy and paste the contents of `diamond-sizes.qmd` into a local R Quarto document. Check that you can run it, and then add text after the frequency polygon that describes its most striking features.
4. Create a document in Google Docs or MS Word (or locate a document you have created previously) with some content in it such as headings, hyperlinks, formatted text, etc. Copy the contents of this document and paste it into a Quarto document in the visual editor. Then, switch to the source editor and inspect the source code.

Code Chunks

To run code inside a Quarto document, you need to insert a chunk. There are three ways to do so:

- Pressing the keyboard shortcut Cmd+Option+I/Ctrl+Alt+I
- Clicking the insert button icon in the editor toolbar
- Manually typing the chunk delimiters ```` ```{r} ```` and ```` ``` ````

We'd recommend you learn the keyboard shortcut. It will save you a lot of time in the long run!

You can continue to run the code using the keyboard shortcut that by now (we hope!) you know and love: Cmd/Ctrl+Enter. However, chunks get a new keyboard shortcut, Cmd/Ctrl+Shift+Enter, which runs all the code in the chunk. Think of a chunk like a function. A chunk should be relatively self-contained and focused around a single task.

The following sections describe the chunk header that consists of ```` ```{r} ````, followed by an optional chunk label and various other chunk options, each on their own line, marked by `#|`.

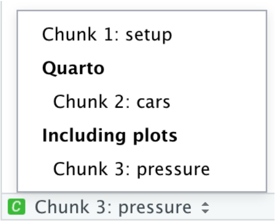
Chunk Label

Chunks can be given an optional label:

~~~
```{r}
#| label: simple-addition

1 + 1
```

#> [1] 2
~~~

This has three advantages:

- You can more easily navigate to specific chunks using the drop-down code navigator in the bottom left of the script editor:



- Graphics produced by the chunks will have useful names that make them easier to use elsewhere. More on that in "Figures" on page 517.
- You can set up networks of cached chunks to avoid re-performing expensive computations on every run. More on that in "Caching" on page 522.

Your chunk labels should be short but evocative and should not contain spaces. We recommend using dashes (`-`) to separate words (instead of underscores, `_`) and avoiding other special characters in chunk labels.

You are generally free to label your chunk however you like, but there is one chunk name that imbues special behavior: `setup`. When you're in a notebook mode, the chunk named `setup` will be run automatically once, before any other code is run.

Additionally, chunk labels cannot be duplicated. Each chunk label must be unique.

Chunk Options

Chunk output can be customized with *options*, fields supplied to the chunk header. Knitr provides almost 60 options that you can use to customize your code chunks. Here we'll cover the most important chunk options that you'll use frequently. You can see the full list here.

The most important set of options controls if your code block is executed and what results are inserted in the finished report:

`eval: false`
: Prevents code from being evaluated. (And obviously if the code is not run, no results will be generated.) This is useful for displaying example code, or for disabling a large block of code without commenting each line.

`include: false`
: Runs the code but doesn't show the code or results in the final document. Use this for setup code that you don't want cluttering your report.

`echo: false`
: Prevents code, but not the results, from appearing in the finished file. Use this when writing reports aimed at people who don't want to see the underlying R code.

`message: false` *or* `warning: false`
: Prevents messages or warnings from appearing in the finished file.

`results: hide`
: Hides printed output.

`fig-show: hide`
: Hides plots.

`error: true`
: Causes the render to continue even if code returns an error. This is rarely something you'll want to include in the final version of your report, but can be useful if you need to debug exactly what is going on inside your `.qmd`. It's also useful if you're teaching R and want to deliberately include an error. The default, `error: false`, causes rendering to fail if there is a single error in the document.

Each of these chunk options gets added to the header of the chunk, following `#|`. For example, in the following chunk, the result is not printed since `eval` is set to false:

~~~
```{r}
#| label: simple-multiplication
#| eval: false

2 * 2
```
~~~

The following table summarizes which types of output each option suppresses:

Option	Run Code	Show Code	Output	Plots	Messages	Warnings
`eval: false`	X		X	X	X	X
`include: false`		X	X	X	X	X
`echo: false`		X				
`results: hide`			X			
`fig-show: hide`				X		
`message: false`					X	
`warning: false`						X

Global Options

As you work more with knitr, you will discover that some of the default chunk options don't fit your needs and you want to change them.

You can do this by adding the preferred options in the document YAML, under `execute`. For example, if you are preparing a report for an audience who does not need to see your code but only your results and narrative, you might set `echo: false` at the document level. That will hide the code by default and show only the chunks you deliberately choose to show (with `echo: true`). You might consider setting `message: false` and `warning: false`, but that would make it harder to debug problems because you wouldn't see any messages in the final document.

```
title: "My report"
execute:
  echo: false
```

Since Quarto is designed to be multilingual (it works with R as well as other languages like Python, Julia, etc.), all of the knitr options are not available at the document execution level since some of them work only with knitr and not other engines Quarto uses for running code in other languages (e.g., Jupyter). You can, however, still set these as global options for your document under the `knitr` field, under `opts_chunk`. For example, when writing books and tutorials we set:

```
title: "Tutorial"
knitr:
  opts_chunk:
    comment: "#>"
    collapse: true
```

This uses our preferred comment formatting and ensures that the code and output are kept closely entwined.

Inline Code

There is one other way to embed R code into a Quarto document: directly into the text, with `r `. This can be useful if you mention properties of your data in the text. For example, the example document used at the start of the chapter had:

> We have data about `r nrow(diamonds)` diamonds. Only `r nrow(diamonds) - nrow(smaller)` are larger than 2.5 carats. The distribution of the remainder is shown below:

When the report is rendered, the results of these computations are inserted into the text:

> We have data about 53940 diamonds. Only 126 are larger than 2.5 carats. The distribution of the remainder is shown below:

When inserting numbers into text, format() is your friend. It allows you to set the number of digits so you don't print to a ridiculous degree of accuracy, and you can use big.mark to make numbers easier to read. You might combine these into a helper function:

```
comma <- function(x) format(x, digits = 2, big.mark = ",")
comma(3452345)
#> [1] "3,452,345"
comma(.12358124331)
#> [1] "0.12"
```

Exercises

1. Add a section that explores how diamond sizes vary by cut, color, and clarity. Assume you're writing a report for someone who doesn't know R, and instead of setting echo: false on each chunk, set a global option.
2. Download diamond-sizes.qmd. Add a section that describes the largest 20 diamonds, including a table that displays their most important attributes.
3. Modify diamonds-sizes.qmd to use label_comma() to produce nicely formatted output. Also include the percentage of diamonds that are larger than 2.5 carats.

Figures

The figures in a Quarto document can be embedded (e.g., a PNG or JPEG file) or generated as a result of a code chunk.

To embed an image from an external file, you can use the Insert menu in the Visual Editor RStudio and select Figure/Image. This will pop open a menu where you can browse to the image you want to insert as well as add alternative text or a caption to it and adjust its size. In the visual editor you can also simply paste an image from your clipboard into your document and RStudio will place a copy of that image in your project folder.

If you include a code chunk that generates a figure (e.g., includes a `ggplot()` call), the resulting figure will be automatically included in your Quarto document.

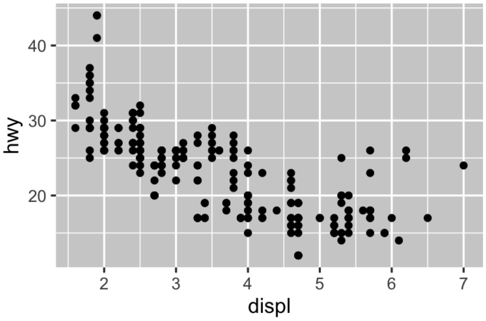
Figure Sizing

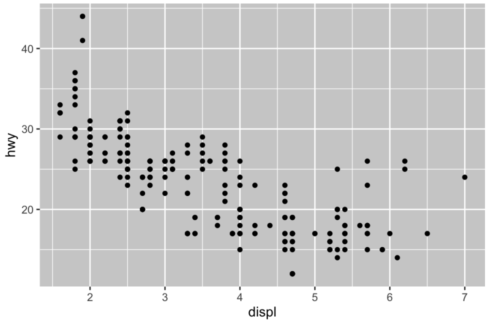
The biggest challenge of graphics in Quarto is getting your figures the right size and shape. There are five main options that control figure sizing: `fig-width`, `fig-height`, `fig-asp`, `out-width`, and `out-height`. Image sizing is challenging because there are two sizes (the size of the figure created by R and the size at which it is inserted in the output document) and multiple ways of specifying the size (i.e., height, width, and aspect ratio: pick two of three).

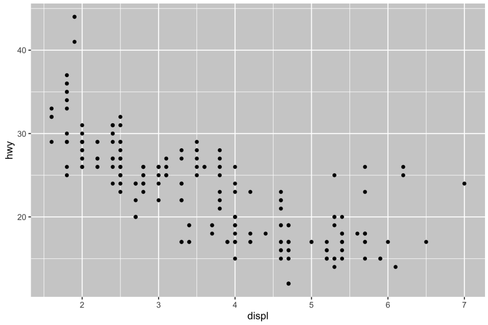
We recommend three of the five options:

- Plots tend to be more aesthetically pleasing if they have consistent width. To enforce this, set `fig-width: 6` (6”) and `fig-asp: 0.618` (the golden ratio) in the defaults. Then in individual chunks, adjust only `fig-asp`.
- Control the output size with `out-width` and set it to a percentage of the body width of the output document. We suggest `out-width: "70%"` and `fig-align: center`. That gives plots room to breathe, without taking up too much space.
- To put multiple plots in a single row, set `layout-ncol` to 2 for two plots, 3 for three plots, etc. Depending on what you’re trying to illustrate (e.g., show data or show plot variations), you might also tweak `fig-width`, as discussed next.

If you find that you’re having to squint to read the text in your plot, you need to tweak `fig-width`. If `fig-width` is larger than the size the figure is rendered in the final doc, the text will be too small; if `fig-width` is smaller, the text will be too big. You’ll often need to do a little experimentation to figure out the right ratio between the `fig-width` and the eventual width in your document. To illustrate the principle, the following three plots have `fig-width` of 4, 6, and 8, respectively:







If you want to make sure the font size is consistent across all your figures, whenever you set `out-width`, you'll also need to adjust `fig-width` to maintain the same ratio with your default `out-width`. For example, if your default `fig-width` is 6 and `out-width` is "70%" when you set `out-width: "50%"`, you'll need to set `fig-width` to 4.3 (6 * 0.5 / 0.7).

Figure sizing and scaling is an art and science, and getting things right can require an iterative trial-and-error approach. You can learn more about figure sizing in the "Taking Control of Plot Scaling" blog post.

Other Important Options

When mingling code and text, like in this book, you can set `fig-show: hold` so that plots are shown after the code. This has the pleasant side effect of forcing you to break up large blocks of code with their explanations.

To add a caption to the plot, use `fig-cap`. In Quarto this will change the figure from inline to "floating."

If you're producing PDF output, the default graphics type is PDF. This is a good default because PDFs are high-quality vector graphics. However, they can produce large and slow plots if you are displaying thousands of points. In that case, set `fig-format: "png"` to force the use of PNGs. They are slightly lower quality but will be much more compact.

It's a good idea to name code chunks that produce figures, even if you don't routinely label other chunks. The chunk label is used to generate the filename of the graphic on disk, so naming your chunks makes it much easier to pick out plots and reuse them in other circumstances (e.g., if you want to quickly drop a single plot into an email).

Exercises

1. Open `diamond-sizes.qmd` in the visual editor, find an image of a diamond, copy it, and paste it into the document. Double-click the image and add a caption. Resize the image and render your document. Observe how the image is saved in your current working directory.
2. Edit the label of the code chunk in `diamond-sizes.qmd` that generates a plot to start with the prefix `fig-` and add a caption to the figure with the chunk option `fig-cap`. Then, edit the text above the code chunk to add a cross-reference to the figure with Insert > Cross Reference.
3. Change the size of the figure with the following chunk options, one at a time; render your document; and describe how the figure changes.

    a. `fig-width: 10`

    b. `fig-height: 3`

    c. `out-width: "100%"`

    d. `out-width: "20%"`

Tables

Similar to figures, you can include two types of tables in a Quarto document. They can be Markdown tables that you create in directly in your Quarto document (using the Insert Table menu), or they can be tables generated as a result of a code chunk. In this section we will focus on the latter, tables generated via computation.

By default, Quarto prints data frames and matrices as you'd see them in the console:

```
mtcars[1:5, ]
#>                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
#> Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
#> Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
#> Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
#> Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
#> Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

If you prefer that data be displayed with additional formatting, you can use the `knitr::kable()` function. The following code generates Table 28-1:

```
knitr::kable(mtcars[1:5, ], )
```

Table 28-1. A knitr kable

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2

Read the documentation for `?knitr::kable` to see the other ways in which you can customize the table. For even deeper customization, consider the gt, huxtable, reactable, kableExtra, xtable, stargazer, pander, tables, and ascii packages. Each provides a set of tools for returning formatted tables from R code.

Exercises

1. Open `diamond-sizes.qmd` in the visual editor, insert a code chunk, and add a table with `knitr::kable()` that shows the first five rows of the `diamonds` data frame.
2. Display the same table with `gt::gt()` instead.
3. Add a chunk label that starts with the prefix `tbl-` and add a caption to the table with the chunk option `tbl-cap`. Then, edit the text above the code chunk to add a cross-reference to the table with Insert > Cross Reference.

Caching

Normally, each render of a document starts from a completely clean slate. This is great for reproducibility, because it ensures that you've captured every important computation in code. However, it can be painful if you have some computations that take a long time. The solution is `cache: true`.

You can enable the knitr cache at the document level for caching the results of all computations in a document using standard YAML options:

```
---
title: "My Document"
execute:
  cache: true
---
```

You can also enable caching at the chunk level for caching the results of computation in a specific chunk:

~~~
```{r}
#| cache: true

# code for lengthy computation...
```
~~~

When set, this will save the output of the chunk to a specially named file on disk. On subsequent runs, knitr will check to see if the code has changed, and if it hasn't, it will reuse the cached results.

The caching system must be used with care, because by default it is based on the code only, not its dependencies. For example, here the `processed_data` chunk depends on the `raw-data` chunk:

~~~
```{r}
#| label: raw-data
#| cache: true

rawdata <- readr::read_csv("a_very_large_file.csv")
```

```{r}
#| label: processed_data
#| cache: true

processed_data <- rawdata |>
  filter(!is.na(import_var)) |>
  mutate(new_variable = complicated_transformation(x, y, z))
```
~~~

Caching the `processed_data` chunk means that it will get rerun if the dplyr pipeline is changed, but it won't get rerun if the `read_csv()` call changes. You can avoid that problem with the `dependson` chunk option:

~~~
```{r}
#| label: processed-data
#| cache: true
#| dependson: "raw-data"

processed_data <- rawdata |>
  filter(!is.na(import_var)) |>
  mutate(new_variable = complicated_transformation(x, y, z))
```
~~~

dependson should contain a character vector of *every* chunk that the cached chunk depends on. Knitr will update the results for the cached chunk whenever it detects that one of its dependencies has changed.

Note that the chunks won't update if a_very_large_file.csv changes, because knitr caching tracks changes only within the .qmd file. If you want to also track changes to that file, you can use the cache.extra option. This is an arbitrary R expression that will invalidate the cache whenever it changes. A good function to use is file.mtime(): it returns when it was last modified. Then you can write:

~~~
```{r}
#| label: raw-data
#| cache: true
#| cache.extra: !expr file.mtime("a_very_large_file.csv")

rawdata <- readr::read_csv("a_very_large_file.csv")
```
~~~

We've followed the advice of David Robinson to name these chunks: each chunk is named after the primary object that it creates. This makes it easier to understand the dependson specification.

As your caching strategies get progressively more complicated, it's a good idea to regularly clear out all your caches with knitr::clean_cache().

Exercises

1. Set up a network of chunks where d depends on c and b, and both b and c depend on a. Have each chunk print lubridate::now(), set cache: true, and then verify your understanding of caching.

Troubleshooting

Troubleshooting Quarto documents can be challenging because you are no longer in an interactive R environment, and you will need to learn some new tricks. Additionally, the error could be due to issues with the Quarto document itself or due to the R code in the Quarto document.

One common error in documents with code chunks is duplicated chunk labels, which are especially pervasive if your workflow involves copying and pasting code chunks. To address this issue, all you need to do is to change one of your duplicated labels.

If the errors are due to the R code in the document, the first thing you should always try is to re-create the problem in an interactive session. Restart R, and then select "Run all chunks," either from the Code menu, under the Run region, or by pressing the keyboard shortcut Ctrl+Alt+R. If you're lucky, that will re-create the problem, and you can figure out what's going on interactively.

If that doesn't help, there must be something different between your interactive environment and the Quarto environment. You're going to need to systematically explore the options. The most common difference is the working directory: the working directory of a Quarto is the directory in which it lives. Check the working directory is what you expect by including `getwd()` in a chunk.

Next, brainstorm all the things that might cause the bug. You'll need to systematically check that they're the same in your R session and your Quarto session. The easiest way to do that is to set `error: true` on the chunk causing the problem and then use `print()` and `str()` to check that settings are as you expect.

YAML Header

You can control many other "whole document" settings by tweaking the parameters of the YAML header. You might wonder what YAML stands for: it's "YAML Ain't Markup Language," which is designed for representing hierarchical data in a way that's easy for humans to read and write. Quarto uses it to control many details of the output. Here we'll discuss three: self-contained documents, document parameters, and bibliographies.

Self-Contained

HTML documents typically have a number of external dependencies (e.g., images, CSS style sheets, JavaScript, etc.) and, by default, Quarto places these dependencies in a `_files` folder in the same directory as your `.qmd` file. If you publish the HTML file on a hosting platform (e.g., QuartoPub), the dependencies in this directory are published with your document and hence are available in the published report. However, if you want to email the report to a colleague, you might prefer to have a single, self-contained, HTML document that embeds all of its dependencies. You can do this by specifying the `embed-resources` option.

```
format:
  html:
    embed-resources: true
```

the resulting file will be self-contained, such that it will need no external files and no internet access to be displayed properly by a browser.

Parameters

Quarto documents can include one or more parameters whose values can be set when you render the report. Parameters are useful when you want to re-render the same report with distinct values for various key inputs. For example, you might be producing sales reports per branch, exam results by student, or demographic summaries by country. To declare one or more parameters, use the `params` field.

This example uses a `my_class` parameter to determine which class of cars to display:

~~~
---
format: html
params:
  my_class: "suv"
---

```{r}
#| label: setup
#| include: false

library(tidyverse)

class <- mpg |> filter(class == params$my_class)
```

# Fuel economy for `r params$my_class`s

```{r}
#| message: false

ggplot(class, aes(x = displ, y = hwy)) +
  geom_point() +
  geom_smooth(se = FALSE)
```
~~~

As you can see, parameters are available within the code chunks as a read-only list named `params`.

You can write atomic vectors directly into the YAML header. You can also run arbitrary R expressions by prefacing the parameter value with `!expr`. This is a good way to specify date/time parameters.

```
params:
  start: !expr lubridate::ymd("2015-01-01")
  snapshot: !expr lubridate::ymd_hms("2015-01-01 12:30:00")
```

Bibliographies and Citations

Quarto can automatically generate citations and a bibliography in a number of styles. The most straightforward way of adding citations and bibliographies to a Quarto document is using the visual editor in RStudio.

To add a citation using the visual editor, select Insert > Citation. Citations can be inserted from a variety of sources:

- DOI references
- Zotero personal or group libraries.
- Searches of Crossref, DataCite, or PubMed.
- Your document bibliography (a `.bib` file in the directory of your document)

Under the hood, the visual mode uses the standard Pandoc Markdown representation for citations (e.g., `[@citation]`).

If you add a citation using one of the first three methods, the visual editor will automatically create a `bibliography.bib` file for you and add the reference to it. It will also add a `bibliography` field to the document YAML. As you add more references, this file will get populated with their citations. You can also directly edit this file using many common bibliography formats including BibLaTeX, BibTeX, EndNote, and Medline.

To create a citation within your `.qmd` file in the source editor, use a key composed of @ plus the citation identifier from the bibliography file. Then place the citation in square brackets. Here are some examples:

```
Separate multiple citations with a `;`: Blah blah [@smith04; @doe99].

You can add arbitrary comments inside the square brackets:
Blah blah [see @doe99, pp. 33-35; also @smith04, ch. 1].

Remove the square brackets to create an in-text citation: @smith04
says blah, or @smith04 [p. 33] says blah.

Add a `-` before the citation to suppress the author's name:
Smith says blah [-@smith04].
```

When Quarto renders your file, it will build and append a bibliography to the end of your document. The bibliography will contain each of the cited references from your bibliography file, but it will not contain a section heading. As a result it is common practice to end your file with a section header for the bibliography, such as `# References` or `# Bibliography`.

You can change the style of your citations and bibliography by referencing a citation style language (CSL) file in the `csl` field:

```
bibliography: rmarkdown.bib
csl: apa.csl
```

As with the bibliography field, your CSL file should contain a path to the file. Here we assume that the CSL file is in the same directory as the `.qmd` file. A good place to find CSL style files for common bibliography styles is the official repository for citation styles.

Workflow

Earlier, we discussed a basic workflow for capturing your R code where you work interactively in the *console* and then capture what works in the *script editor*. Quarto brings together the console and the script editor, blurring the lines between interactive exploration and long-term code capture. You can rapidly iterate within a chunk, editing and re-executing with Cmd/Ctrl+Shift+Enter. When you're happy, you move on and start a new chunk.

Quarto is also important because it so tightly integrates prose and code. This makes it a great *analysis notebook* because it lets you develop code and record your thoughts. An analysis notebook shares many of the same goals as a classic lab notebook in the physical sciences. It:

- Records what you did and why you did it. Regardless of how great your memory is, if you don't record what you do, there will come a time when you have forgotten important details. Write them down so you don't forget!
- Supports rigorous thinking. You are more likely to come up with a strong analysis if you record your thoughts as you go and continue to reflect on them. This also saves you time when you eventually write up your analysis to share with others.
- Helps others understand your work. It is rare to do data analysis by yourself, and you'll often be working as part of a team. A lab notebook helps you share not only what you've done but why you did it with your colleagues or lab mates.

Much of the good advice about using lab notebooks effectively can also be translated to analysis notebooks. We've drawn on our own experiences and Colin Purrington's advice on lab notebooks to come up with the following tips:

- Ensure each notebook has a descriptive title, an evocative filename, and a first paragraph that briefly describes the aims of the analysis.

- Use the YAML header date field to record the date you started working on the notebook:

  ```
  date: 2016-08-23
  ```

  Use ISO8601 YYYY-MM-DD format so that's there no ambiguity. Use it even if you don't normally write dates that way!
- If you spend a lot of time on an analysis idea and it turns out to be a dead end, don't delete it! Write up a brief note about why it failed and leave it in the notebook. That will help you avoid going down the same dead end when you come back to the analysis in the future.
- Generally, you're better off doing data entry outside of R. But if you do need to record a small snippet of data, clearly lay it out using `tibble::tribble()`.
- If you discover an error in a data file, never modify it directly, but instead write code to correct the value. Explain why you made the fix.
- Before you finish for the day, make sure you can render the notebook. If you're using caching, make sure to clear the caches. That will let you fix any problems while the code is still fresh in your mind.
- If you want your code to be reproducible in the long run (i.e., so you can come back to run it next month or next year), you'll need to track the versions of the packages that your code uses. A rigorous approach is to use *renv*, which stores packages in your project directory. A quick and dirty hack is to include a chunk that runs `sessionInfo()`—that won't let you easily re-create your packages as they are today, but at least you'll know what they were.
- You are going to create many, many, many analysis notebooks over the course of your career. How are you going to organize them so you can find them again in the future? We recommend storing them in individual projects and coming up with a good naming scheme.

Summary

This chapter introduced you to Quarto for authoring and publishing reproducible computational documents that include your code and your prose in one place. You learned about writing Quarto documents in RStudio with the visual or source editor, how code chunks work and how to customize options for them, how to include figures and tables in your Quarto documents, and options for caching for computations. Additionally, you learned about adjusting YAML header options for creating self-contained or parameterized documents as well as including citations and a bibliography. We also gave you some troubleshooting and workflow tips.

While this introduction should be sufficient to get you started with Quarto, there is still a lot more to learn. Quarto is still relatively young and is still growing rapidly. The best place to stay on top of innovations is the official Quarto website.

There are two important topics that we haven't covered here: collaboration and the details of accurately communicating your ideas to other humans. Collaboration is a vital part of modern data science, and you can make your life much easier by using version control tools, like Git and GitHub. We recommend *Happy Git with R*, a user-friendly introduction to Git and GitHub from R users, by Jenny Bryan. The book is freely available online.

We have also not touched on what you should actually write to clearly communicate the results of your analysis. To improve your writing, we highly recommend reading either *Style: Lessons in Clarity and Grace* by Joseph M. Williams & Joseph Bizup (Pearson) or *The Sense of Structure: Writing from the Reader's Perspective* by George Gopen (Pearson). Both books will help you understand the structure of sentences and paragraphs and give you the tools to make your writing clearer. (These books are rather expensive if purchased new, but they're used by many English classes, so there are plenty of cheap second-hand copies.) George Gopen also has a number of short articles on writing. They are aimed at lawyers, but almost everything applies to data scientists too.

CHAPTER 29

Quarto Formats

Introduction

So far, you've seen Quarto used to produce HTML documents. This chapter gives a brief overview of some of the many other types of output you can produce with Quarto.

There are two ways to set the output of a document:

- Permanently, by modifying the YAML header:

  ```
  title: "Diamond sizes"
  format: html
  ```

- Transiently, by calling `quarto::quarto_render()` by hand:

  ```
  quarto::quarto_render("diamond-sizes.qmd", output_format = "docx")
  ```

  This is useful if you want to programmatically produce multiple types of output since the `output_format` argument can also take a list of values:

  ```
  quarto::quarto_render(
    "diamond-sizes.qmd", output_format = c("docx", "pdf")
  )
  ```

Output Options

Quarto offers a wide range of output formats. You can find the complete list on the Quarto documentation on all formats. Many formats share some output options (e.g., `toc: true` for including a table of contents), but others have options that are format specific (e.g., `code-fold: true` collapses code chunks into a `<details>` tag for HTML output so the user can display it on demand; it's not applicable in a PDF or Word document).

To override the default options, you need to use an expanded format field. For example, if you wanted to render an HTML document, with a floating table of contents, you'd use:

```
format:
  html:
    toc: true
    toc_float: true
```

You can even render to multiple outputs by supplying a list of formats:

```
format:
  html:
    toc: true
    toc_float: true
  pdf: default
  docx: default
```

Note the special syntax (pdf: default) if you don't want to override any default options.

To render to all formats specified in the YAML of a document, you can use output_format = "all":

```
quarto::quarto_render("diamond-sizes.qmd", output_format = "all")
```

Documents

The previous chapter focused on the default html output. There are several basic variations on that theme, generating different types of documents. For example:

- pdf makes a PDF with LaTeX (an open-source document layout system), which you'll need to install. RStudio will prompt you if you don't already have it.
- docx for Microsoft Word (.docx) documents.
- odt for OpenDocument Text (.odt) documents.
- rtf for Rich Text Format (.rtf) documents.
- gfm for a GitHub Flavored Markdown (.md) document.
- ipynb for Jupyter Notebooks (.ipynb).

Remember, when generating a document to share with decision-makers, you can turn off the default display of code by setting global options in the document YAML:

```
execute:
  echo: false
```

For HTML documents, another option is to make the code chunks hidden by default but visible with a click:

```
format:
  html:
    code: true
```

Presentations

You can also use Quarto to produce presentations. You get less visual control than with a tool like Keynote or PowerPoint, but automatically inserting the results of your R code into a presentation can save a huge amount of time. Presentations work by dividing your content into slides, with a new slide beginning at each second (##) level header. Additionally, first (#) level headers indicate the beginning of a new section with a section title slide that is, by default, centered in the middle.

Quarto supports a variety of presentation formats, including:

`revealjs`
: HTML presentation with revealjs

`pptx`
: PowerPoint presentation

`beamer`
: PDF presentation with LaTeX Beamer

You can read more about creating presentations with Quarto.

Interactivity

Just like any HTML document, HTML documents created with Quarto can contain interactive components as well. Here we introduce two options for including interactivity in your Quarto documents: htmlwidgets and Shiny.

htmlwidgets

HTML is an interactive format, and you can take advantage of that interactivity with *htmlwidgets*, R functions that produce interactive HTML visualizations. For example, take the *leaflet* map shown next. If you're viewing this page on the web, you can drag the map around, zoom in and out, etc. You obviously can't do that in a book, so Quarto automatically inserts a static screenshot for you.

```
library(leaflet)
leaflet() |>
  setView(174.764, -36.877, zoom = 16) |>
  addTiles() |>
  addMarkers(174.764, -36.877, popup = "Maungawhau")
```

+
−
View Road
Bellevue Road
Esplanade Road
Percy Street
Mount Eden Road
Lovelock Avenue
Charlton Avenue
Coles Avenue
Batger Road
Rautangi Road
Puhi Huia Road
Clifton House
Glenfell Place
Wilkies Lookout
Omana Avenue
Owens Road
Leaflet | © OpenStreetMap contributors, CC-BY-SA

The great thing about htmlwidgets is that you don't need to know anything about HTML or JavaScript to use them. All the details are wrapped inside the package, so you don't need to worry about it.

There are many packages that provide htmlwidgets, including:

- dygraphs for interactive time series visualizations
- DT for interactive tables
- threejs for interactive 3D plots
- DiagrammeR for diagrams (like flow charts and simple node-link diagrams)

To learn more about htmlwidgets and see a complete list of packages that provide them, visit *https://oreil.ly/lmdha*.

Shiny

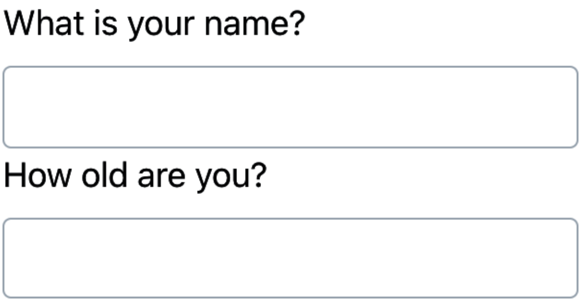
htmlwidgets provide *client-side* interactivity—all the interactivity happens in the browser, independently of R. That's great because you can distribute the HTML file without any connection to R. However, that fundamentally limits what you can do to things that have been implemented in HTML and JavaScript. An alternative approach is to use shiny, a package that allows you to create interactivity using R code, not JavaScript.

To call Shiny code from a Quarto document, add `server: shiny` to the YAML header:

```
title: "Shiny Web App"
format: html
server: shiny
```

Then you can use the "input" functions to add interactive components to the document:

```
library(shiny)

textInput("name", "What is your name?")
numericInput("age", "How old are you?", NA, min = 0, max = 150)
```



And you also need a code chunk with the chunk option `context: server`, which contains the code that needs to run in a Shiny server.

You can then refer to the values with `input$name` and `input$age`, and the code that uses them will be automatically rerun whenever they change.

We can't show you a live Shiny app here because Shiny interactions occur on the *server side*. This means you can write interactive apps without knowing JavaScript, but you need a server to run them on. This introduces a logistical issue: Shiny apps need a Shiny server to be run online. When you run Shiny apps on your own computer, Shiny automatically sets up a Shiny server for you, but you need a public-facing Shiny server if you want to publish this sort of interactivity online. That's the fundamental trade-off of Shiny: you can do anything in a Shiny document that you can do in R, but it requires someone to be running R.

To learn more about Shiny, we recommend reading *Mastering Shiny* by Hadley Wickham.

Websites and Books

With a bit of additional infrastructure, you can use Quarto to generate a complete website or book:

- Put your .qmd files in a single directory. index.qmd will become the home page.
- Add a YAML file named _quarto.yml that provides the navigation for the site. In this file, set the project type to either book or website, e.g.:

  ```
  project:
    type: book
  ```

For example, the following _quarto.yml file creates a website from three source files: index.qmd (the home page), viridis-colors.qmd, and terrain-colors.qmd.

```
project:
  type: website

website:
  title: "A website on color scales"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - href: viridis-colors.qmd
        text: Viridis colors
      - href: terrain-colors.qmd
        text: Terrain colors
```

The _quarto.yml file you need for a book is similarly structured. The following example shows how you can create a book with four chapters that renders to three different outputs (html, pdf, and epub). Once again, the source files are .qmd files.

```
project:
  type: book

book:
  title: "A book on color scales"
  author: "Jane Coloriste"
  chapters:
    - index.qmd
    - intro.qmd
    - viridis-colors.qmd
    - terrain-colors.qmd

format:
  html:
    theme: cosmo
  pdf: default
  epub: default
```

We recommend that you use an RStudio project for your websites and books. Based on the _quarto.yml file, RStudio will recognize the type of project you're working on and add a Build tab to the IDE that you can use to render and preview your websites and books. Both websites and books can also be rendered using `quarto::render()`.

Read more about Quarto websites and books.

Other Formats

Quarto offers even more output formats:

- You can write journal articles using Quarto Journal Templates.
- You can output Quarto documents to Jupyter Notebooks with `format: ipynb`.

See the Quarto formats documentation for a list of even more formats.

Summary

In this chapter we presented you with a variety of options for communicating your results with Quarto, from static and interactive documents to presentations to websites and books.

To learn more about effective communication in these different formats, we recommend the following resources:

- To improve your presentation skills, try *Presentation Patterns* by Neal Ford, Matthew McCollough, and Nathaniel Schutta. It provides a set of effective patterns (both low- and high-level) that you can apply to improve your presentations.
- If you give academic talks, you might like "The Leek group guide to giving talks".
- We haven't taken it ourselves, but we've heard good things about Matt McGarrity's online course on public speaking.
- If you are creating many dashboards, make sure to read Stephen Few's *Information Dashboard Design: The Effective Visual Communication of Data* (O'Reilly). It will help you create dashboards that are truly useful, not just pretty to look at.
- Effectively communicating your ideas often benefits from some knowledge of graphic design. Robin Williams's *The Non-Designer's Design Book* (Peachpit Press) is a great place to start.

Index

About the Authors

Hadley Wickham is Chief Scientist at Posit, PBC, winner of the 2019 COPSS award, and a member of the R Foundation. He builds tools (both computational and cognitive) to make data science easier, faster, and more fun. His work includes packages for data science (like the tidyverse, which includes ggplot2, dplyr, and tidyr) and principled software development (e.g. roxygen2, testthat, and pkgdown). He is also a writer, educator, and speaker promoting the use of R for data science. Learn more on his website.

Mine Çetinkaya-Rundel is Professor of the Practice at the Department of Statistical Science at Duke University and Developer Educator at Posit, PBC. Mine's work focuses on innovation in statistics and data science pedagogy, with an emphasis on computing, reproducible research, student-centered learning, and open source education. Mine has authored introductory statistics textbooks as part of the OpenIntro project, she is the creator and maintainer of Data Science in a Box, and she teaches the popular Statistics with R specialization on Coursera. Mine is the winner of the 2021 Hogg Award for Excellence in Teaching Introductory Statistics, the 2018 Harvard Pickard Award, and the 2016 ASA Waller Education Award. Learn more on her website.

Garrett Grolemund is a statistician, teacher, and the director of learning at Posit Academy. He is the author of *Hands-On Programming with R* (O'Reilly) and an early contributor to the tidyverse.

Colophon

The animal on the cover of *R for Data Science* is the kakapo (*Strigops habroptilus*). Also known as the owl parrot, the kakapo is a large flightless bird native to New Zealand. Adult kakapos can grow up to 64 centimeters in height and 4 kilograms in weight. Their feathers are generally yellow and green, although there is significant variation between individuals. Kakapos are nocturnal and use their robust sense of smell to navigate at night. Although they cannot fly, kakapos have strong legs that enable them to run and climb much better than most birds.

The name kakapo comes from the language of the native Maori people of New Zealand. Kakapos were an important part of Maori culture, both as a food source and as a part of Maori mythology. Kakapo skin and feathers were also used to make cloaks and capes.

Due to the introduction of predators to New Zealand during European colonization, kakapos are now critically endangered, with less than 200 individuals currently living. The government of New Zealand has been actively attempting to revive the kakapo population by providing special conservation zones on three predator-free islands.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on *Wood's Animate Creations*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.