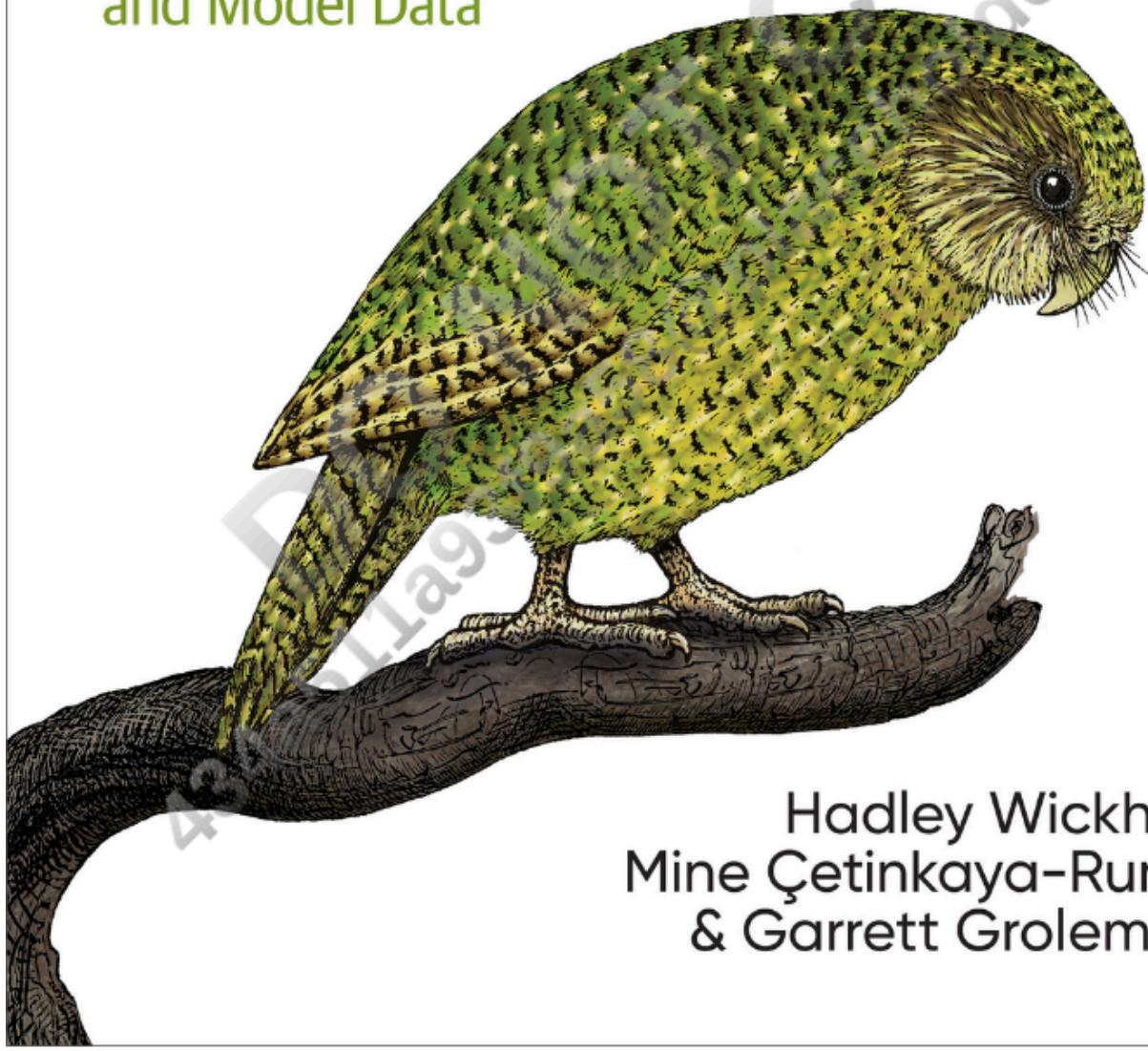


O'REILLY®

Second  
Edition

# R for Data Science

Import, Tidy, Transform, Visualize,  
and Model Data



Hadley Wickham,  
Mine Çetinkaya-Rundel  
& Garrett Grolemund

# R for Data Science

2ND EDITION

Import, Tidy, Transform, Visualize, and Model Data

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



Beijing • Boston • Farnham • Sebastopol • Tokyo

## R for Data Science

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

Copyright © 2023 Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund. All rights reserved.

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](mailto: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 O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *R for Data Science*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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]

## 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

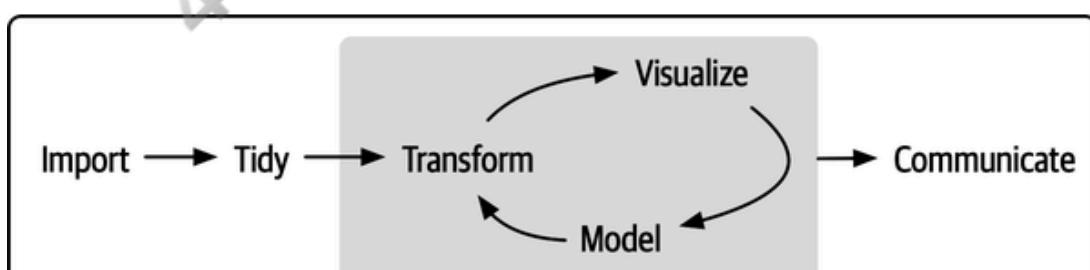
Welcome to the second edition of *R for Data Science* (*R4DS*)! This is a major reworking of the first edition, removing material we no longer think is useful, adding material we wish we included in the first edition, and generally updating the text and code to reflect changes in best practices. We're 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](#).



## Understand

### Program

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 comprehensive R archive network. 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

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!<sup>1</sup>

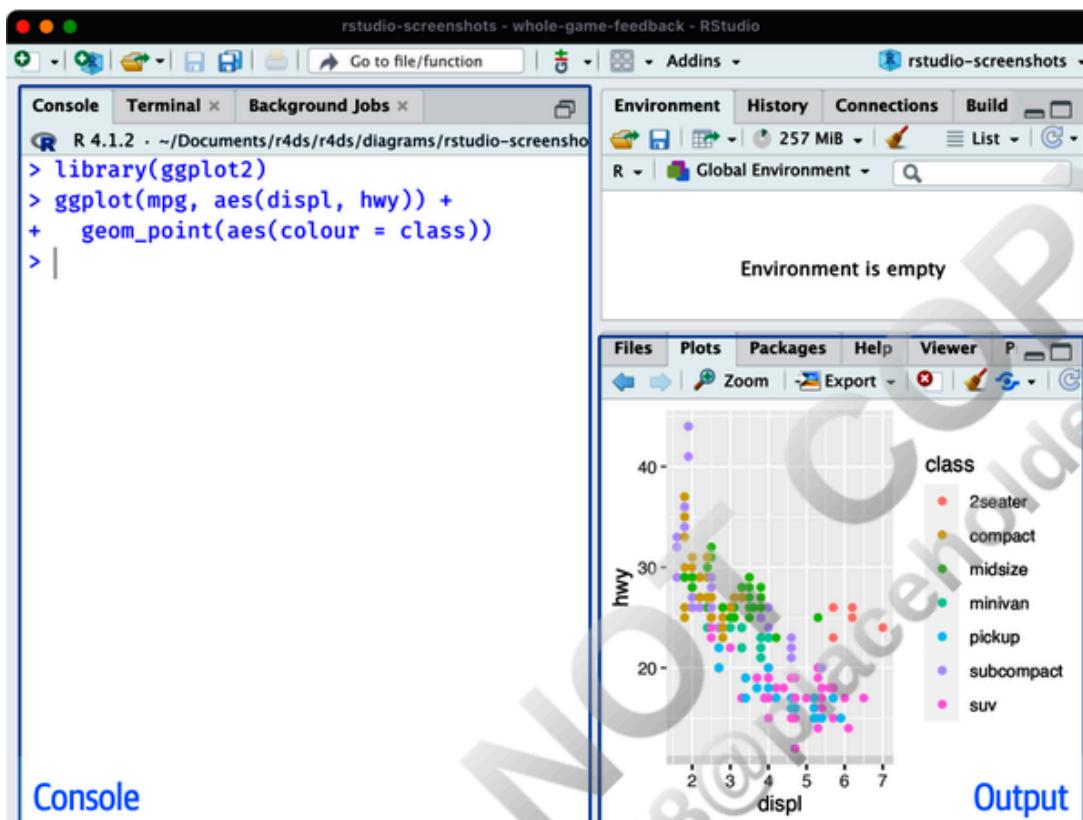


Figure I-2. The RStudio IDE has two key regions: type R code in the console pane on the left, and look for plots in the output pane on the right.

## 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 —
```

```
#> — 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       ✓ tidyrr    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 () to force all
#>   conflicts to become errors
```

This tells you that tidyverse loads nine packages: dplyr,forcats, ggplot2, lubridate, purrr, readr, stringr, tibble, and tidyrr. 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",
  "gridridges", "ggthemes", "hexbin", "janitor", "Lahman", "leaflet", "maps",
  "nycflights13", "openxlsx", "palmerpenguins", "reprex", "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.

#### NOTE

This element signifies a general note.

#### WARNING

This element indicates a warning or caution.

## O'Reilly Online Learning

#### NOTE

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](mailto: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 Hidayatuloh (@aephidayatuloh), Andrea Gilardi (@agila5), Ajay Deonarine (@ajay-d), @AlanFeder, Daihe Sui (@alansuidaihe), @alberto-agudo, @AlbertRapp, @aleloj, pete (@alonzi), Alex (@ALShum), Andrew M. (@amacfarland), Andrew Landgraf (@andland), @andyhuynh92, Angela Li (@angela-li), Antti Rask (@AnttiRask), LOU Xun (@aquahead), @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 (@camilleleonard), @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, @dbirke, 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, @gabriera, 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 Krukova (@ivan-krukova), 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), Joo Young Seo (@jooyoungseo), Justinas Petuchovas (@jpetuchovas), Jordan (@jrdnbradford), Jeffrey Arnold (@jrnold), Jose Roberto Ayala Solares (@jroberayaras), 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 (@KitJonathan), @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 (@mitsuoxy), 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), Saghira (@saghira), 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

Printed by: 434a511a93f9a58@placeholder.18157.edu. Printing is for personal, private use only. No part of this book may be reproduced or transmitted without publisher's prior permission. Violators will be prosecuted.

The 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](https://oreil.ly/Q8z_O). The book is powered by **Quarto**, which makes it easy to write books that combine text and executable code.

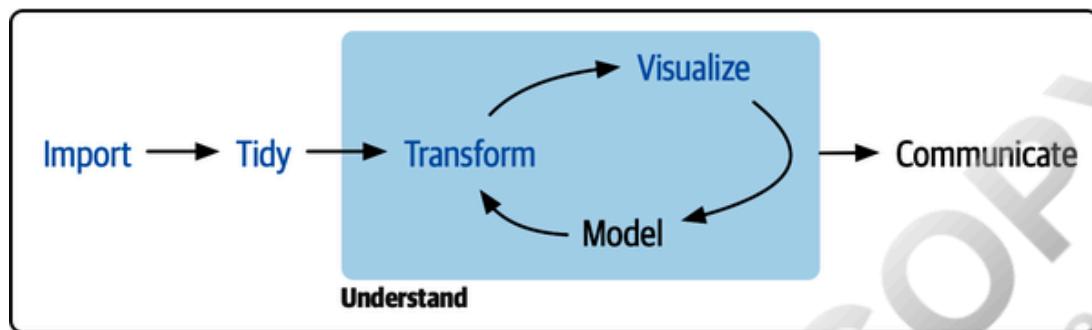
---

<sup>1</sup> If you'd like a comprehensive overview of all of RStudio's features, see the [RStudio User Guide](#).

DO NOT COPY  
434a511a93f9a58@placeholder.18157.edu

## 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.



### Program

*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      ✓ tidyrr   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 () 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).<sup>1</sup>

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?

positive? negative? Linear? Nonlinear? Does the relationship vary by the species or 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.<sup>2</sup>

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>            <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 , sex ,
#> #   year
```

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      Adelie, Adelie, Adelie, Adelie, Adelie, A...
#> $ island        Torgersen, Torgersen, Torgersen, Torgersen, Torg...
#> $ bill_length_mm 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, 34...
#> $ bill_depth_mm 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, 18...
#> $ flipper_length_mm 181, 186, 195, NA, 193, 190, 181, 195, 193, 190, ...
#> $ body_mass_g   3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, 347...
```

```
#> > body_mass_g      3750, 3800, 3250, NA, 3450, 3050, 3620, 4070, 3470...
#> $ sex              male, female, female, NA, female, male, female, m...
#> $ year             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

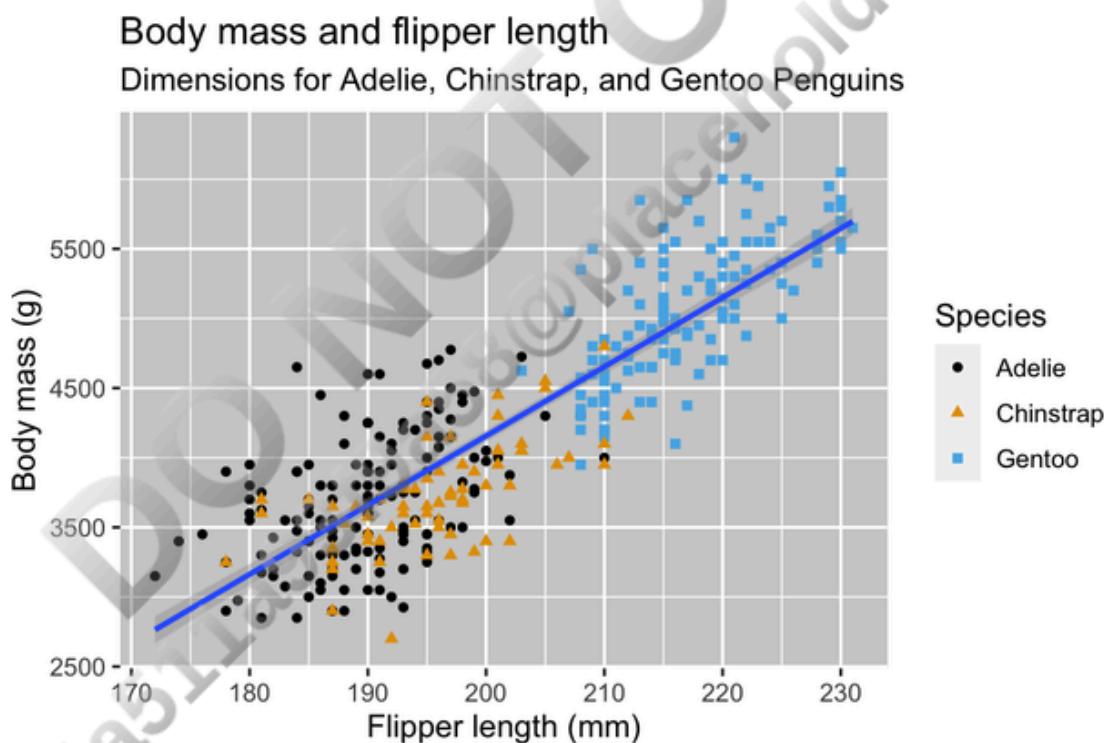
**body\_mass\_g**

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)
```

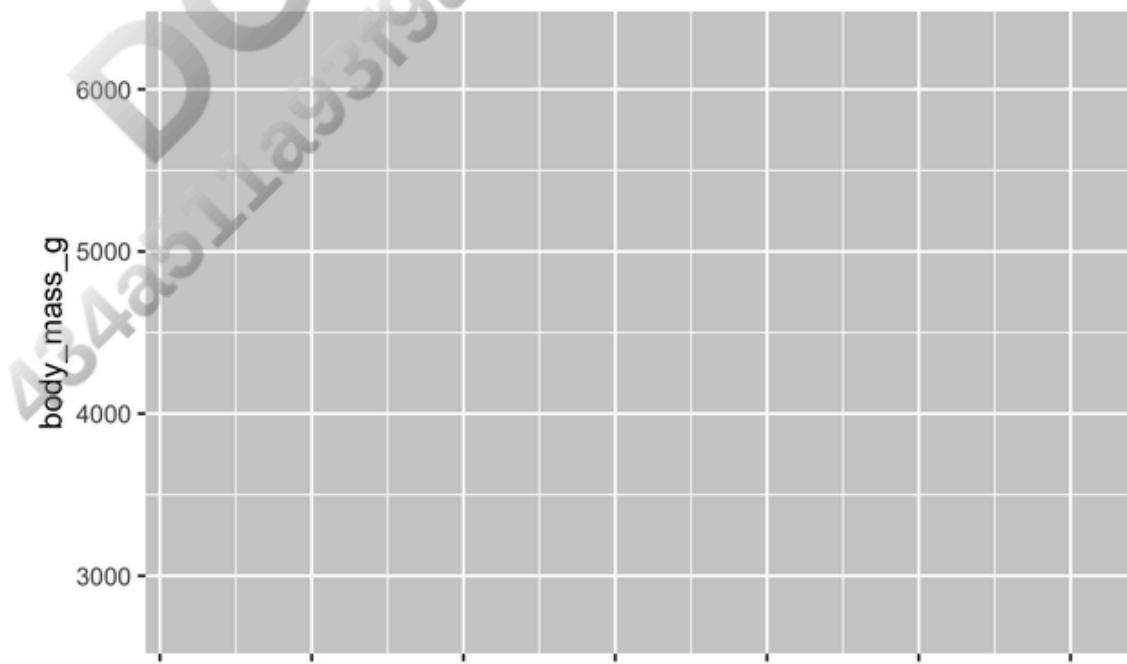
```
ggplot(data = penguins)
```

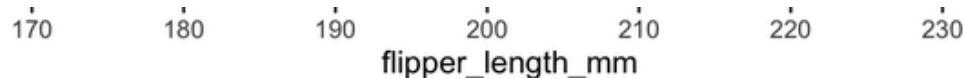


Next, we need to tell `ggplot()` how the information from our data will be visually represented. The `mapping` argument of the `ggplot()` function defines how variables in your dataset are mapped to visual properties (*aesthetics*) of your plot. The `mapping` argument is always defined in the `aes()` 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)  
)
```



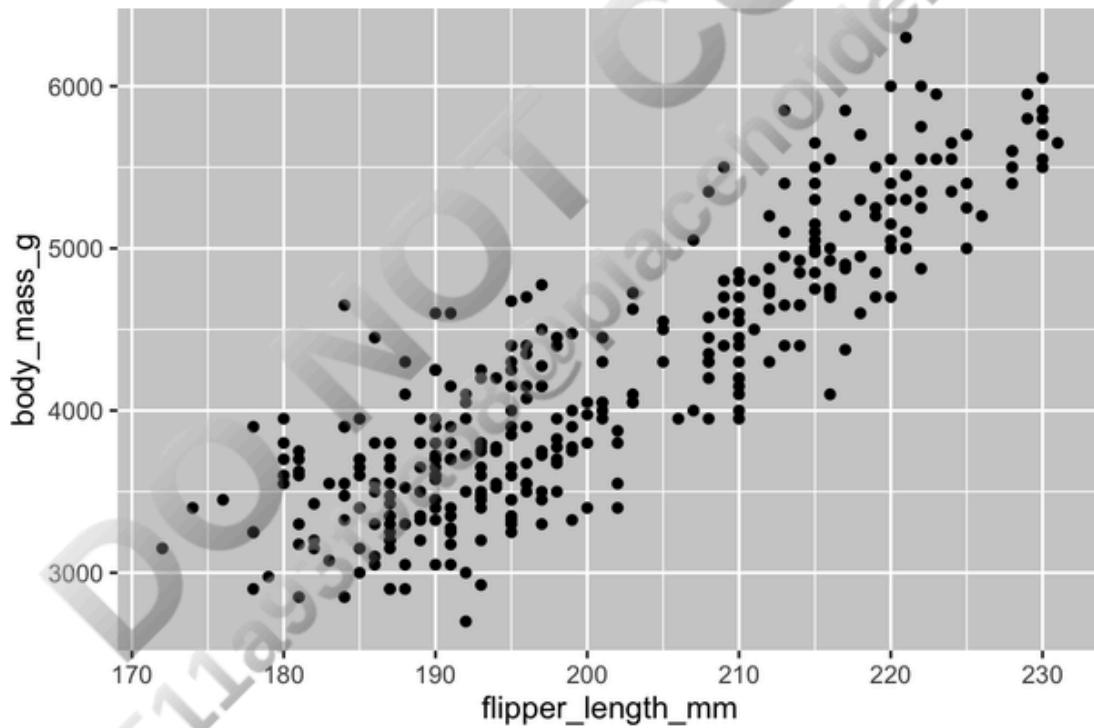


Our empty canvas now has more structure—it's clear where flipper lengths will be displayed (on the x-axis) and where body masses will be displayed (on the y-axis). But the penguins themselves are not yet on the plot. This is because we have not yet articulated, in our code, how to represent the observations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

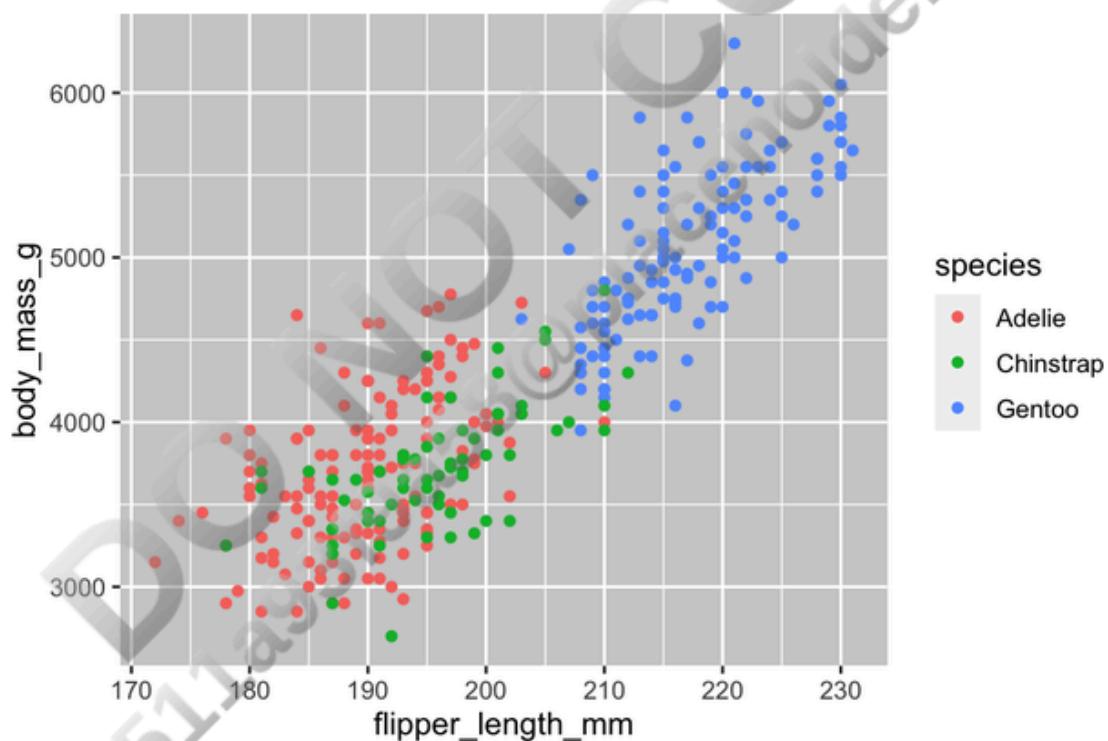
philosophy that missing values should never actually go missing. This type of warning is probably one of the most common types of warnings you will see when working with real data—missing values are a common issue, and you'll learn more about them throughout the book, particularly in [Chapter 18](#). 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()
```



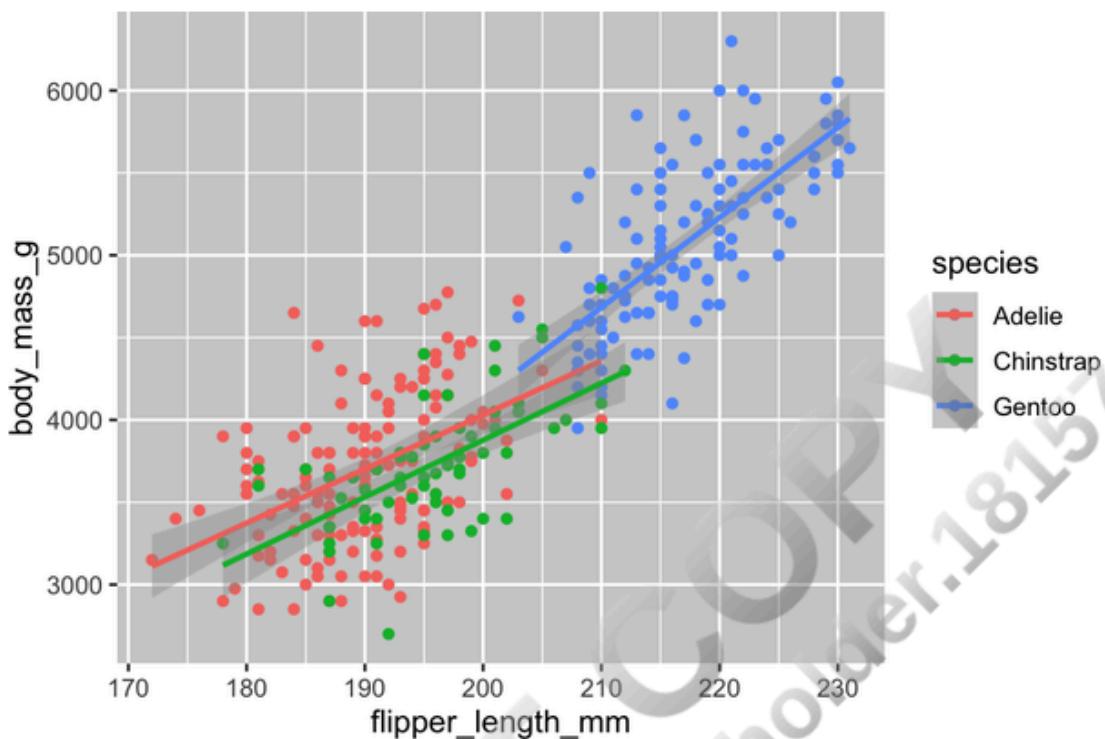
When a categorical variable is mapped to an aesthetic, `ggplot2` will automatically assign a unique value of the aesthetic (here a unique color) to each unique level of the variable (each of the three species), a process known as *scaling*. `ggplot2` will also add a legend that explains which values 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 linear model with `method = "lm"`.

```
ggplot(  
  data = penguins,  
  ...)
```

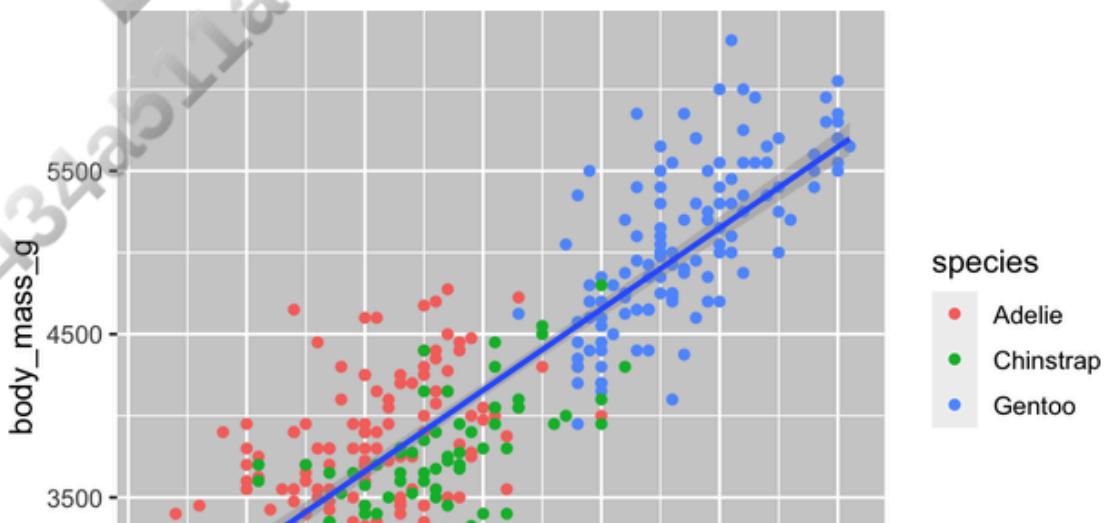
```
data = penguins,  
mapping = aes(x = flipper_length_mm, y = body_mass_g, color = species)  
) +  
geom_point() +  
geom_smooth(method = "lm")
```

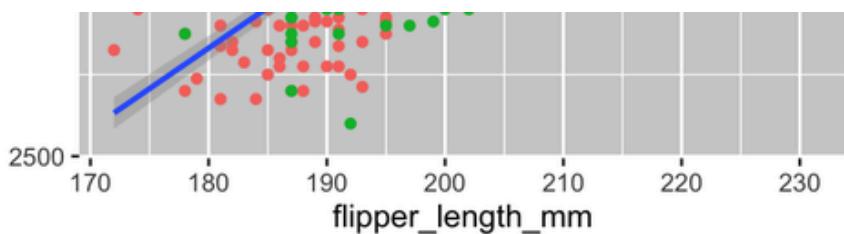


We have successfully added lines, but this plot doesn't look like the plot from "[Ultimate Goal](#)", which has only one line for the entire dataset as opposed to separate lines for each of the penguin species.

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")
```

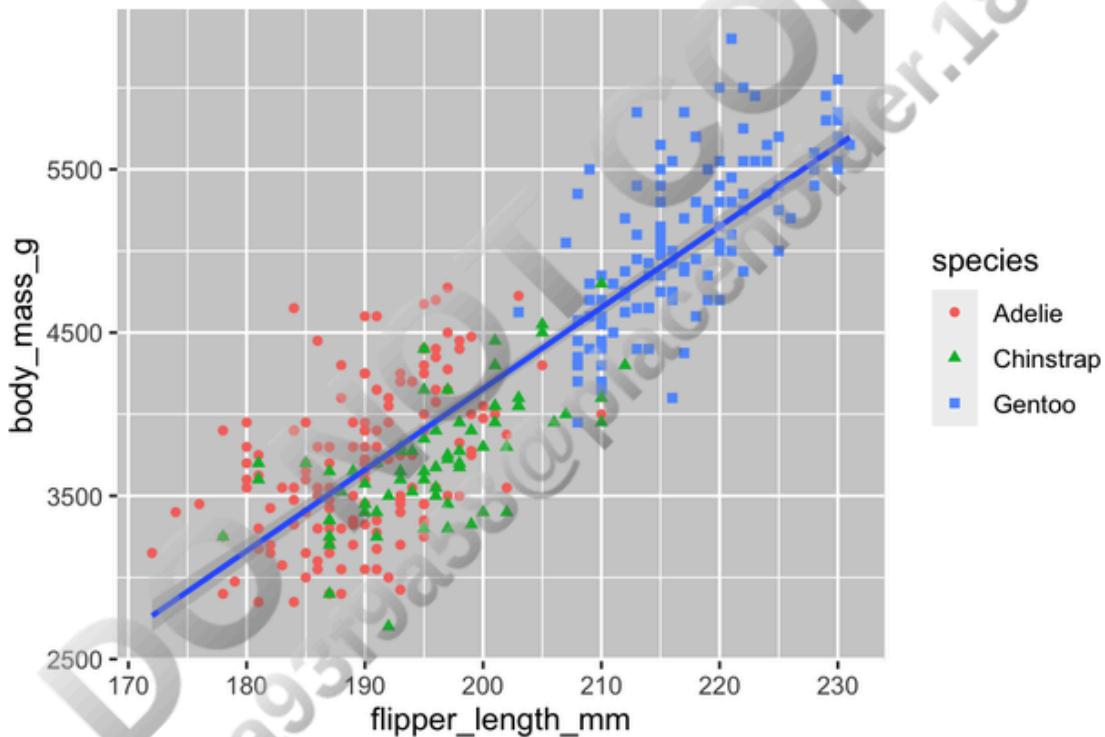




Voilà! We have something that looks very much like our ultimate goal, though it's not yet perfect. We still need to use different shapes for each species of penguins and improve labels.

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")
```

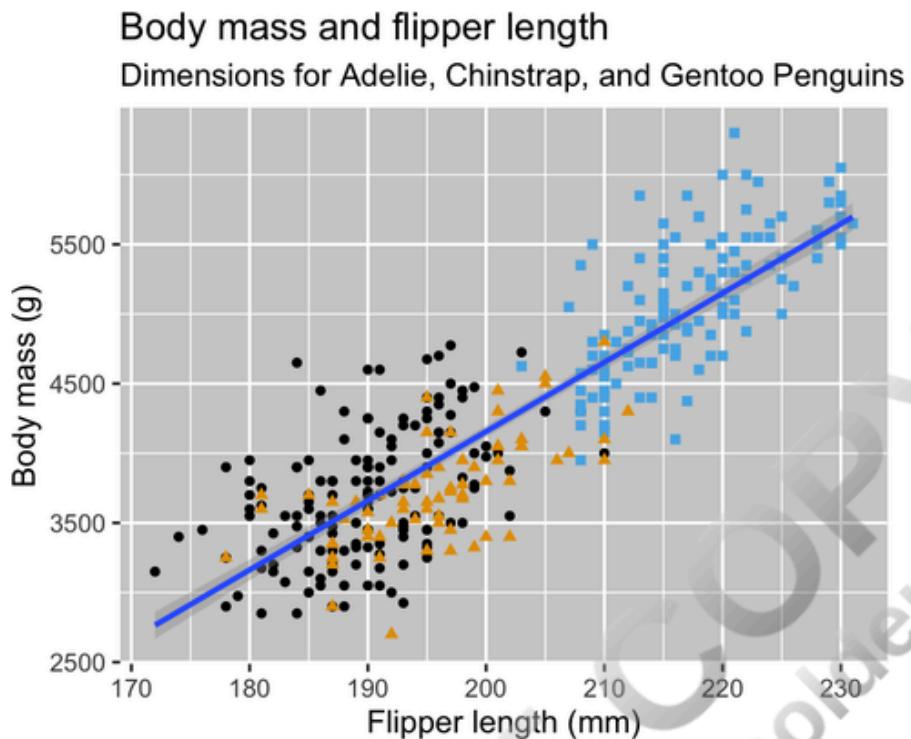


Note that the legend is automatically updated to reflect the different shapes of the points as well.

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(mapping = 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)")
```

```
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”!

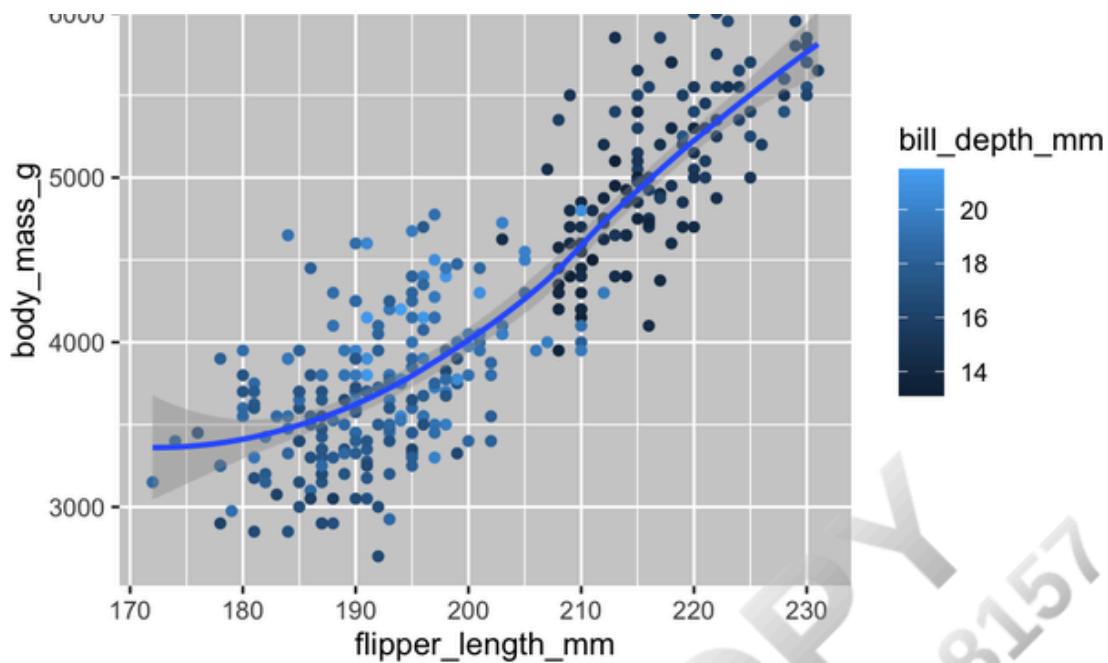
## Exercises

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()
```

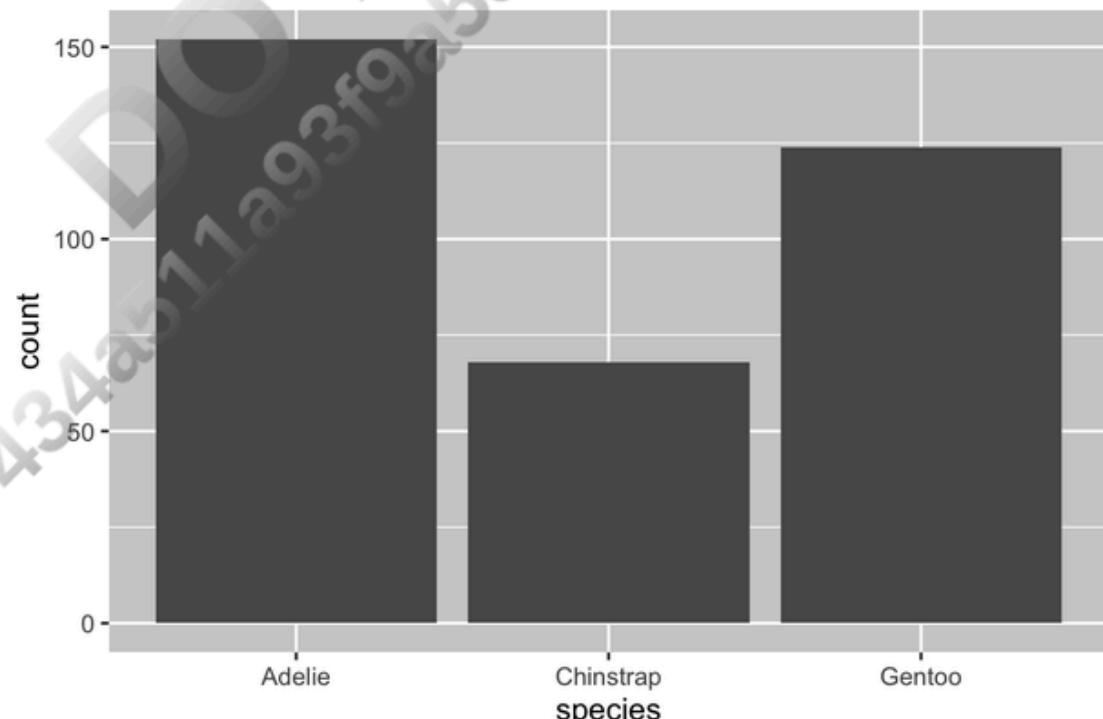
## Visualizing Distributions

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

### A Categorical Variable

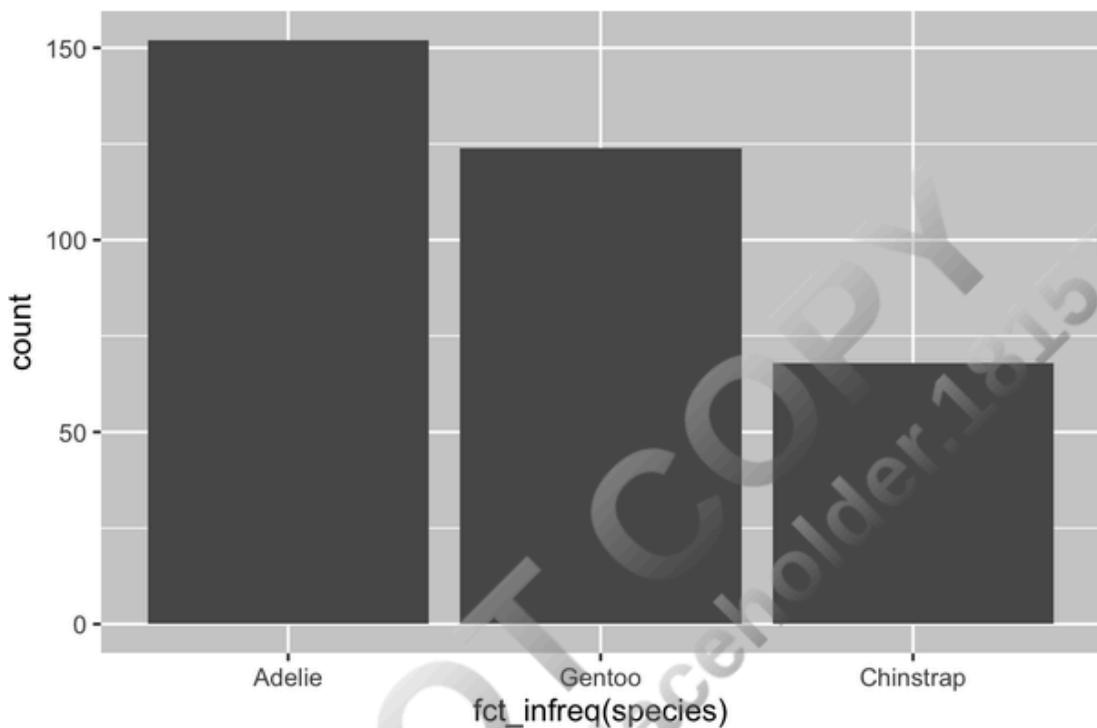
A variable is *categorical* if it can take only one of a small set of values. To examine the distribution of a categorical variable, you can use a bar chart. The height of the bars displays how many observations occurred with each `x` value.

```
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()
```



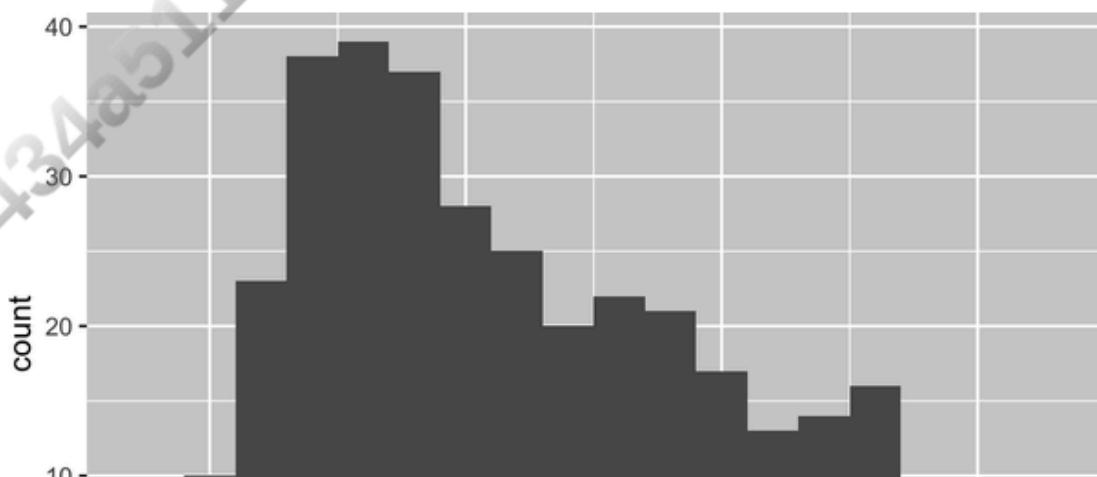
You will learn more about factors and functions for dealing with factors (such as `fct_infreq()`) in [Chapter 16](#).

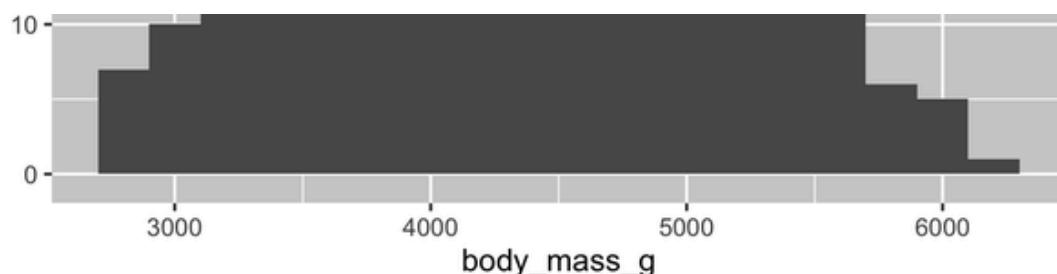
## A Numerical Variable

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)
```

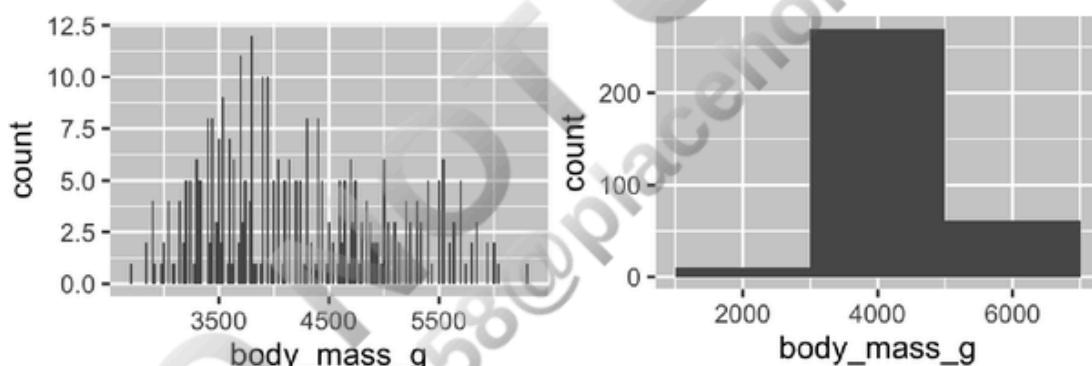




A histogram divides the x-axis into equally spaced bins and then uses the height of a bar to display the number of observations that fall in each bin. In the previous graph, the tallest bar shows that 39 observations have a `body_mass_g` value between 3,500 and 3,700 grams, which are the left and right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

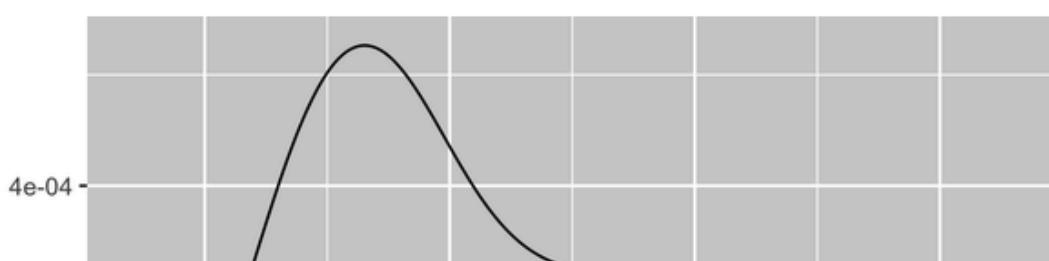
You can set the width of the intervals in a histogram with the `binwidth` argument, which is measured in the units of the x variable. You should always explore a variety of `binwidth` values when working with histograms, as different `binwidth` values can reveal different patterns. In the following plots, 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.

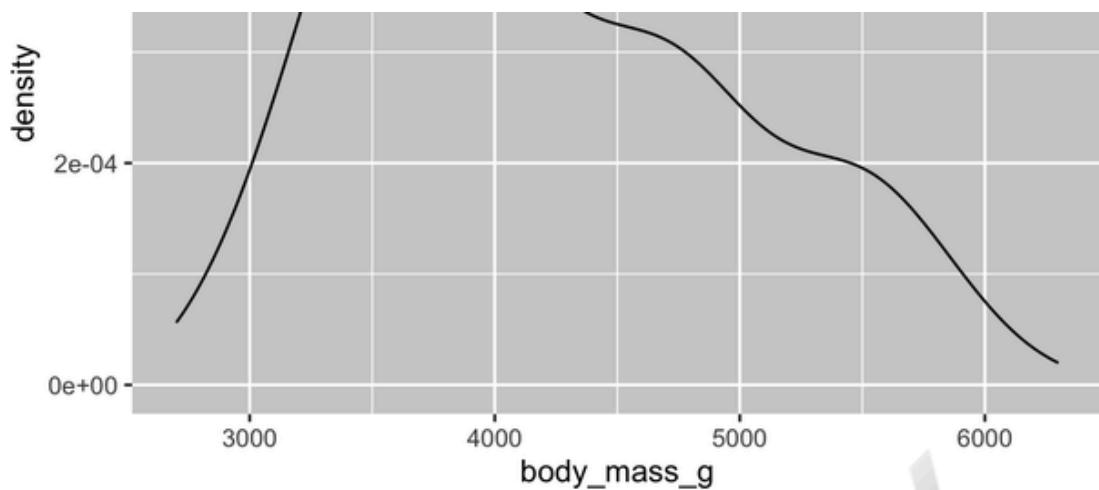
```
ggplot(penguins, aes(x = body_mass_g)) +  
  geom_histogram(binwidth = 20)  
ggplot(penguins, aes(x = body_mass_g)) +  
  geom_histogram(binwidth = 2000)
```



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.

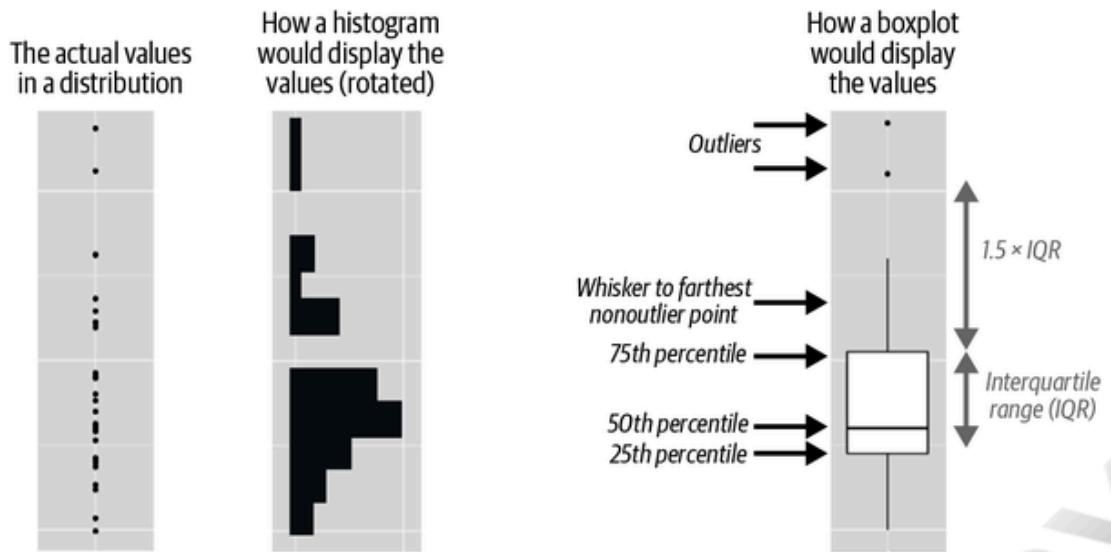
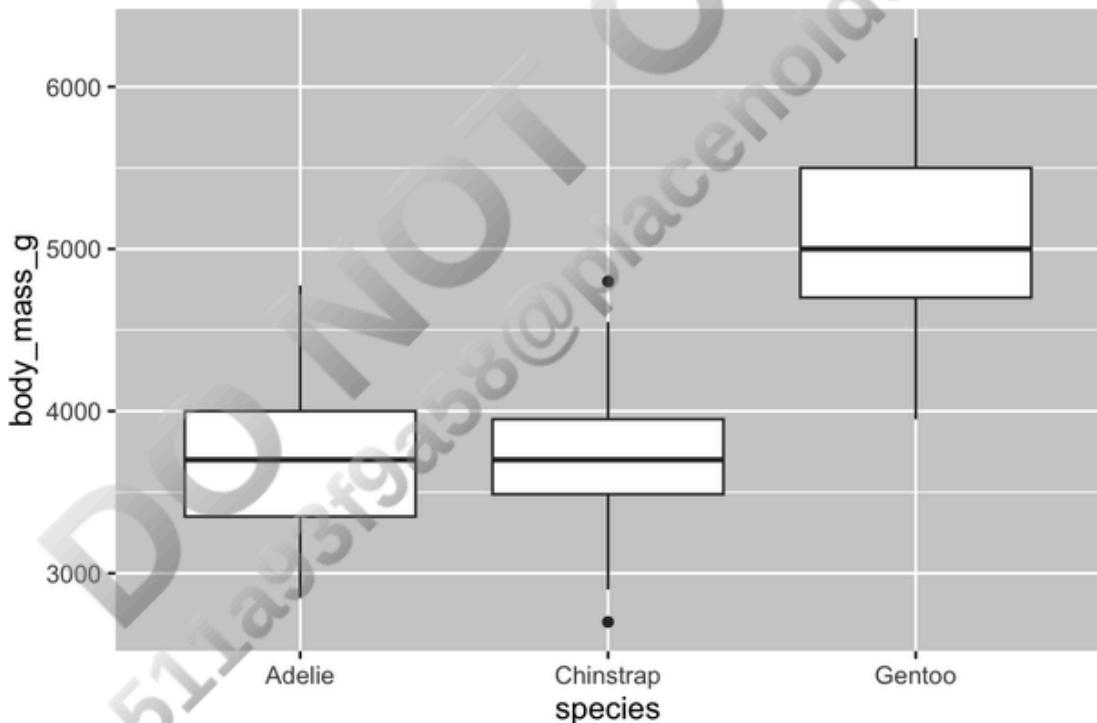


Figure 1-1. Diagram depicting how a boxplot is created.

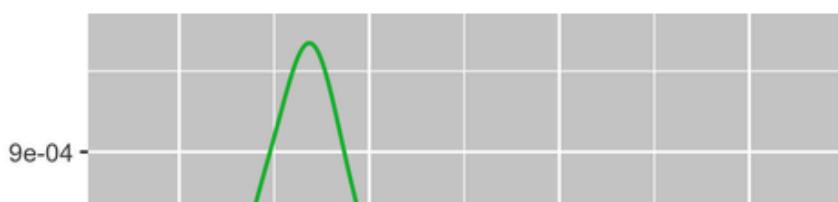
Let's take a look at the distribution of body mass by species using `geom_boxplot()`:

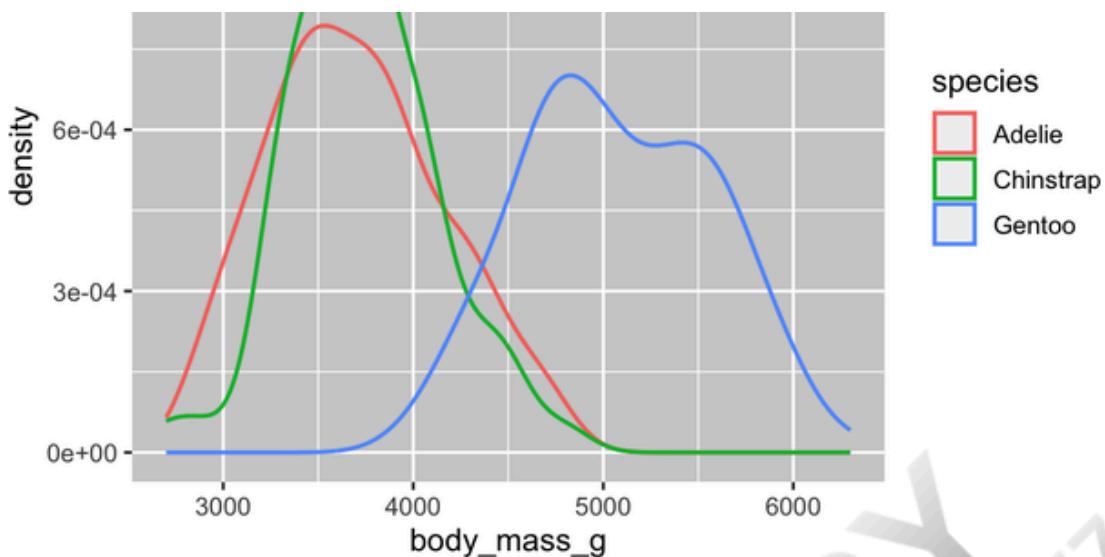
```
ggplot(penguins, aes(x = species, y = body_mass_g)) +  
  geom_boxplot()
```



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

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

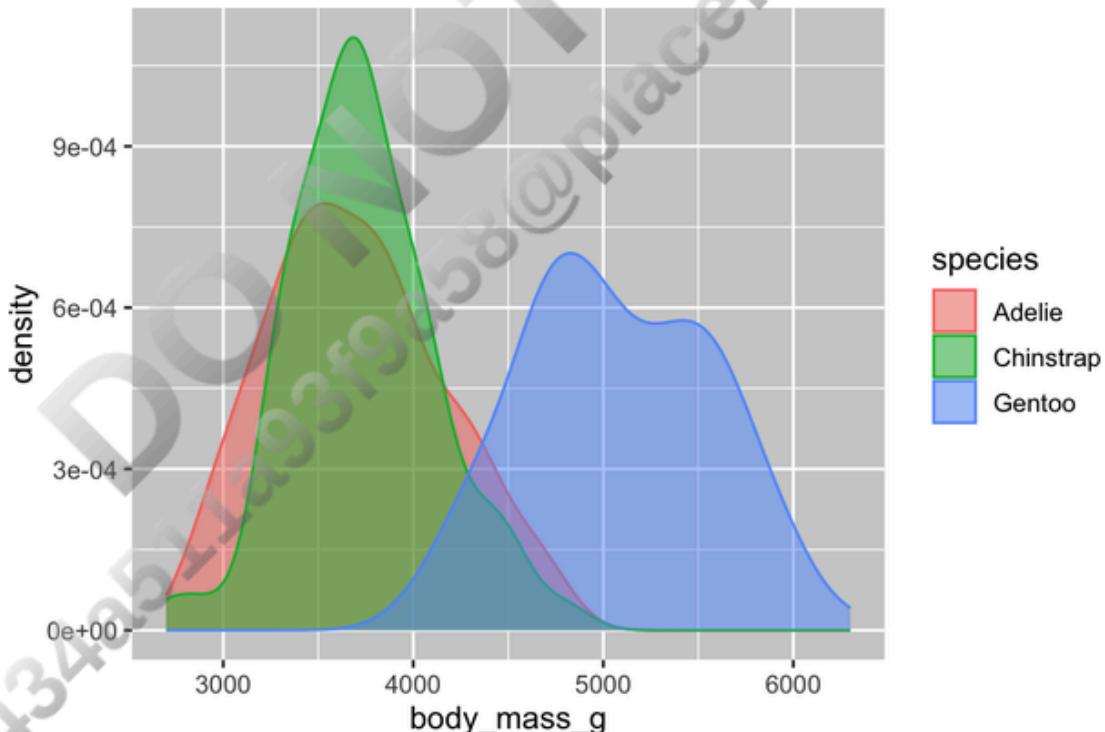




We've also customized the thickness of the lines using the `linewidth` argument to make them stand out a bit more against the background.

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:

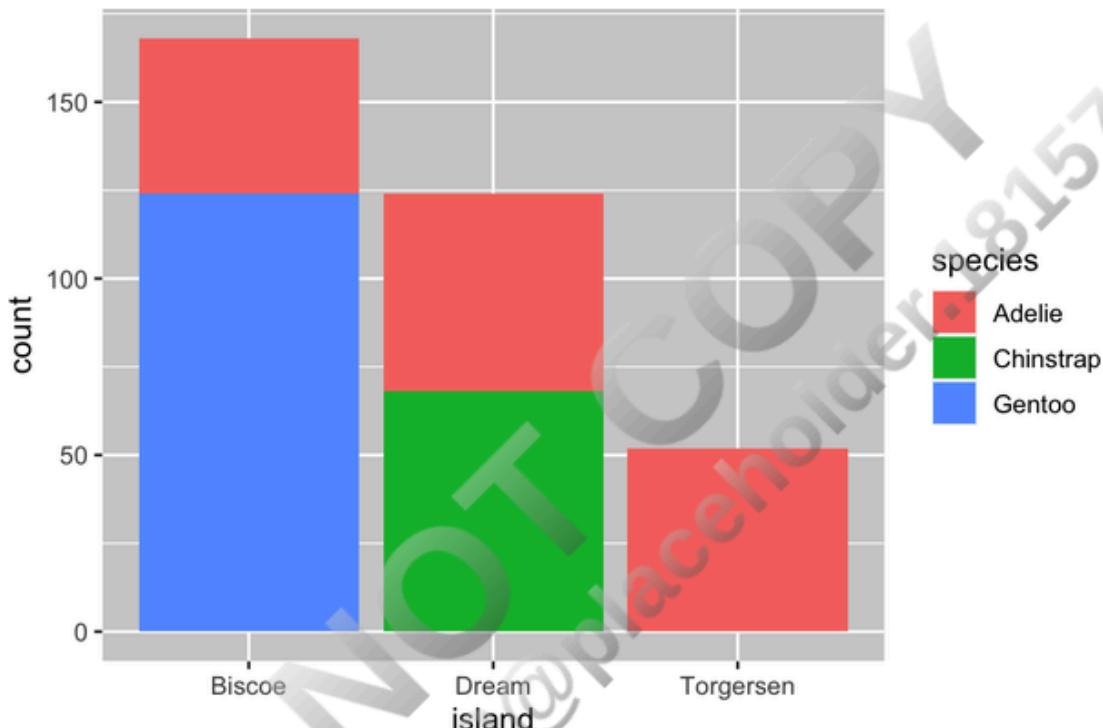
- We *map* variables to aesthetics if we want the visual attribute represented by that aesthetic to vary based on the values of that variable.
- Otherwise, we *set* the value of an aesthetic.

## 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.

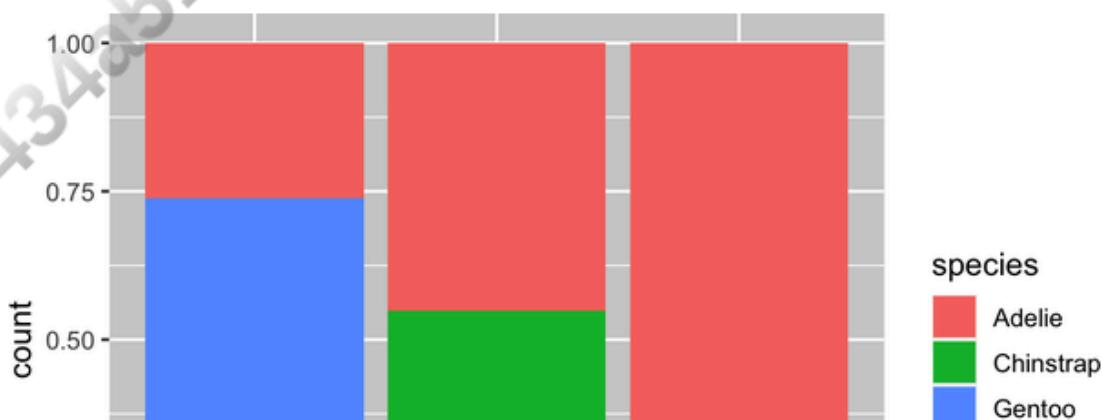
The first plot shows the frequencies of each species of penguins on each island. The plot of frequencies shows that there are equal numbers of Adelies on each island, but we don't have a good sense of the percentage balance within each island.

```
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")
```



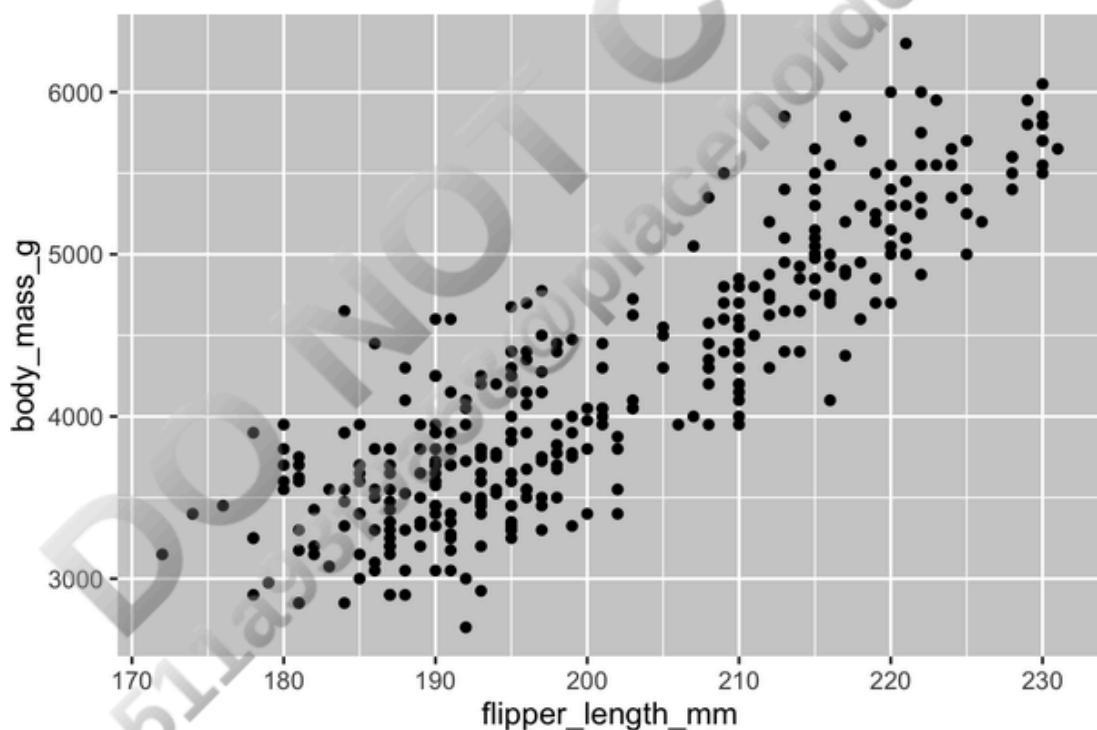


In creating these bar charts, we map the variable that will be separated into bars to the `x` aesthetic, and the variable that will change the colors inside the bars to the `fill` aesthetic.

## 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()
```

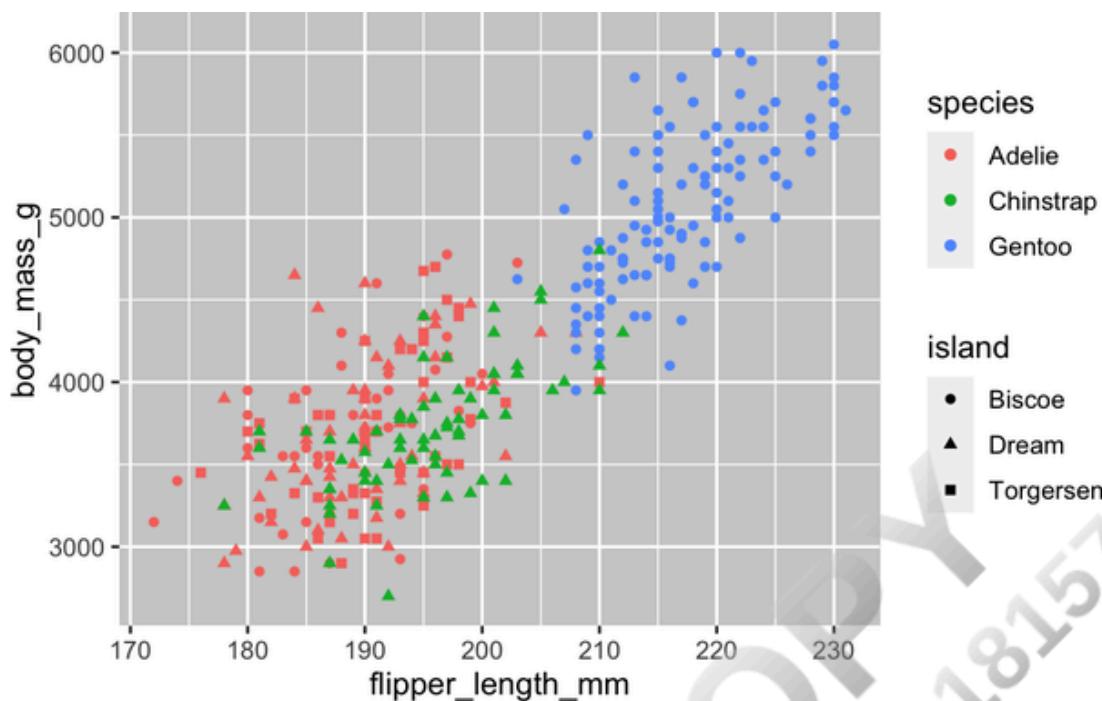


## Three or More Variables

As we saw in “[Adding Aesthetics and Layers](#)”, 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))
```

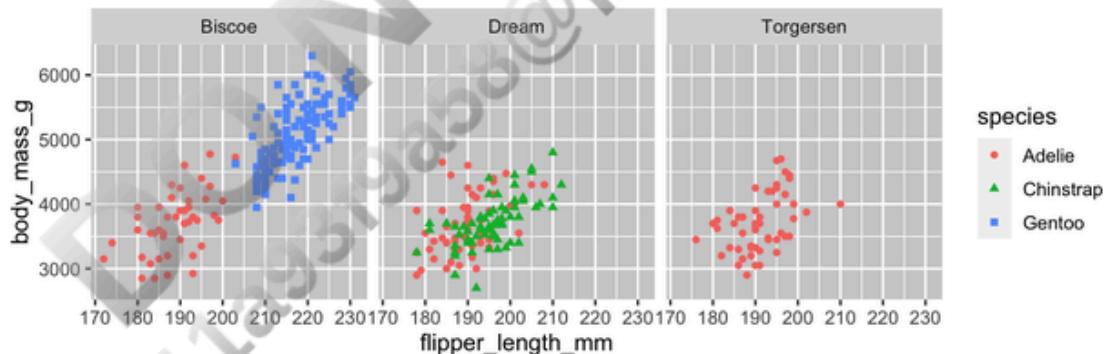




However, adding too many aesthetic mappings to a plot makes it cluttered and difficult to make sense of. Another option, which is particularly useful for categorical variables, is to split your plot into *facets*, subplots that each display one subset of the data.

To facet your plot by a single variable, use `facet_wrap()`. The first argument of `facet_wrap()` is a formula,<sup>3</sup> 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)
```



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()
```

```
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 leveraging 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.

---

<sup>1</sup> 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](#).

<sup>2</sup> 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.

<sup>3</sup> Here "formula" is the name of the thing created by `~`, not a synonym for "equation."