

**R for Data Science**

Import, Tidy, Transform, Visualize, and Model Data

**Hadley Wickham and Garrett Grolemund**

**R for Data Science** by Hadley Wickham and Garrett Grolemund

Copyright © 2017 Garrett Grolemund, Hadley Wickham. All rights reserved.

Printed in Canada.

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/safari>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Marie Beaugureau and Mike Loukides

Production Editor: Nicholas Adams

Copyeditor: Kim Cofer

Proofreader: Charles Roumeliotis

Indexer: Wendy Catalano

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

December 2016: First Edition

**Revision History for the First Edition**

2016-12-06: First Release

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

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.

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.

978-1-491-91039-9

[TI]

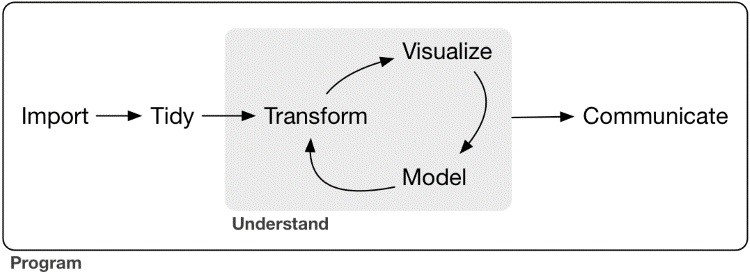
# 前言

数据科学是一门令人兴奋的学科，它可以帮助你您将原始数据转化为理解，洞察和知识。 R for Data Science这本书的目标是帮助您学习R中最重要的工具，使您能够掌握数据科学。 阅读本书后，您将拥有通过使用R里最棒的工具来应对数据科学家面临的各种挑战的能力。

## 从这本书里你可以学到什么

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

数据科学是一个庞大的领域，你无法只阅读一本书就完全掌握它。 本书的目标是为您提供一个掌握最重要工具的坚实基础。 我们在典型数据科学项目中所需的工具模型如下图所示：



First you must *import* your data into R. This typically means that you take data stored in a file, database, or web API, and load it into a data frame in R. If you can’t get your data into R, you can’t do data science on it! 首先，您必须将数据导入R.这通常意味着，您需要将存储在文件，数据库或Web API中的数据加载到R中的数据框中。如果你无法将数据导入R，那也就无所谓什么数据科学！

Once you’ve imported your data, it is a good idea to *tidy* it. Tidying your data means storing it in a consistent form that matches the semantics of the dataset with the way it is stored. In brief, when your data is tidy, each column is a variable, and each row is an observation. Tidy data is important because the consistent structure lets you focus your struggle on questions about the data, not fighting to get the data into the right form for different functions. 数据导入后，最好整理一下。 整理数据意味着将其存储在一致的形式中，该形式与数据集的语义与存储方式相匹配。 简而言之，当您的数据整洁时，每列都是一个变量，每一行都是一个样本。 整洁的数据很重要，因为一致的结构使您可以集中精力处理有关数据的问题，而不是费劲的将数据转换为适合不同函数的正确形式。

Once you have tidy data, a common first step is to *transform* it. Transformation includes narrowing in on observations of interest (like all people in one city, or all data from the last year), creating new variables that are functions of existing variables (like computing velocity from speed and time), and calculating a set of summary statistics (like counts or means). Together, tidying and transforming are called *wrangling*, because getting your data in a form that’s natural to work with often feels like a fight! 一旦你有整洁的数据，通常第一步是转换它。 转换包括缩小感兴趣的样本范围（如同一个城市的所有人，或去年的所有数据），创建作为现有变量函数的新变量（如计算速度和时间的速度），以及计算一组摘要 统计数据（如计数或均值）。整理和转换一起被统称为“争吵”，因为把数据整理成可以以自然方式工作的过程，会让人觉得是在“搏斗”！

Once you have tidy data with the variables you need, there are two main engines of knowledge generation: visualization and modeling. These have complementary strengths and weaknesses so any real analysis will iterate between them many times. 一旦你把所需要的变量数据整理完成，就会得到两个生产知识的引擎：可视化和建模。 它们具有互补的优点和缺点，因此任何真正的分析都会在它们之间多次迭代。

*Visualization* is a fundamentally human activity. A good visualization will show you things that you did not expect, or raise new questions about the data. A good visualization might also hint that you’re asking the wrong question, or you need to collect different data. Visualizations can surprise you, but don’t scale particularly well because they require a human to interpret them. 可视化是一项基本的人类活动。 良好的可视化将向您显示您没有预料到的事情，或者提出有关数据的新问题。 良好的可视化也可能暗示您提出错误的问题，或者您需要收集不同的数据。 可视化可能让您感到惊讶，但不能特别好地扩展，因为它们需要人来解释它们

*Models* are complementary tools to visualization. Once you have made your questions sufficiently precise, you can use a model to answer them. Models are a fundamentally mathematical or computational tool, so they generally scale well. Even when they don’t, it’s usually cheaper to buy more computers than it is to buy more brains! But every model makes assumptions, and by its very nature a model cannot question its own assumptions. That means a model cannot fundamentally surprise you. 模型是可视化的补充工具。 一旦您的问题足够精确，您就可以使用模型来回答它们。 模型是一种基本的数学或计算工具，因此它们通常可以很好地扩展。 即使他们不这样做，购买更多的电脑通常比购买更多的大脑更便宜！ 但是每个模型都会做出假设，而且就其本质而言，模型不能质疑自己的假设。 这意味着模型不能从根本上让您大吃一惊。

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. 数据科学的最后一步是沟通，这是任何数据分析项目的绝对关键部分。 除非您还可以将结果传达给其他人，否则您的模型和可视化对您的理解程度并不重要。

Surrounding all these tools is *programming*. Programming is a cross-cutting tool that you use in every part of the project. You don’t need to be an expert programmer to be a data scientist, but learning more about programming pays off because becoming a better programmer allows you to automate common tasks, and solve new problems with greater ease. 围绕所有这些工具是编程。 编程是一种跨领域工具，可用于项目的每个部分。 您不需要成为一名专业程序员来成为数据科学家，但了解更多有关编程的信息可以获得回报，因为成为更好的程序员可以让您自动执行常见任务，并更轻松地解决新问题。

You’ll use these tools in every data science project, but for most projects they’re not enough. There’s a rough 80-20 rule at play; you can tackle about 80% of every project using the tools that you’ll learn in this book, but you’ll need other tools to tackle the remaining 20%. Throughout this book we’ll point you to resources where you can learn more. 您将在每个数据科学项目中使用这些工具，但对于大多数项目而言，这些工具还不够。 有一个粗略的80-20规则在起作用; 你可以使用你在本书中学到的工具来解决每个项目的大约80％，但是你需要其他工具来解决剩下的20％。 在本书中，我们将为您提供可以了解更多信息的资源。

## How This Book Is Organized这边书是如何组织的

The previous description of the tools of data science is organized roughly according to the order in which you use them in an analysis (although of course you’ll iterate through them multiple times). In our experience, however, this is not the best way to learn them: 以前对数据科学工具的描述大致按照您在分析中使用它们的顺序进行组织（当然，您将多次迭代它们）。 然而，根据我们的经验，这不是学习它们的最佳方式：

Starting with data ingest and tidying is suboptimal because 80% of the time it’s routine and boring, and the other 20% of the time it’s weird and frustrating. That’s a bad place to start learning a new subject! Instead, we’ll start with visualization and transformation of data that’s already been imported and tidied. That way, when you ingest and tidy your own data, your motivation will stay high because you know the pain is worth it. 从数据摄取和整理开始是次优的，因为80％的时间是常规和无聊的，而另外20％的时间是奇怪和令人沮丧的。 这是开始学习新科目的好地方！ 相反，我们将从已经导入和整理的数据的可视化和转换开始。 这样，当您摄取并整理自己的数据时，您的动机会保持高水平，因为您知道痛苦是值得的。

Some topics are best explained with other tools. For example, we believe that it’s easier to understand how models work if you already know about visualization, tidy data, and programming. 最好用其他工具解释一些主题。 例如，如果您已经了解可视化，整洁的数据和编程，我们相信更容易理解模型的工作原理。

Programming tools are not necessarily interesting in their own right, but do allow you to tackle considerably more challenging problems. We’ll give you a selection of programming tools in the middle of the book, and then you’ll see they can combine with the data science tools to tackle interesting modeling problems. 编程工具本身并不一定有趣，但可以让您解决更具挑战性的问题。 我们将在本书的中间为您提供一系列编程工具，然后您将看到它们可以与数据科学工具结合使用来解决有趣的建模问题。

Within each chapter, we try to stick to a similar pattern: start with some motivating examples so you can see the bigger picture, and then dive into the details. Each section of the book is paired with exercises to help you practice what you’ve learned. While it’s tempting to skip the exercises, there’s no better way to learn than practicing on real problems. 在每一章中，我们都试图坚持类似的模式：从一些激励性的例子开始，这样你就能看到更大的图景，然后深入细节。 本书的每一部分都配有练习，以帮助您练习所学的内容。 尽管跳过练习很有吸引力，但没有比练习真正问题更好的学习方法了。

## What You Wont Learn书里不包括的内容

There are some important topics that this book doesn’t cover. We believe it’s important to stay ruthlessly focused on the essentials so you can get up and running as quickly as possible. That means this book can’t cover every important topic. 本书未涉及一些重要主题。 我们认为重要的是要坚持专注于必需品，以便您能够尽快启动和运行。 这意味着本书无法涵盖所有重要主题。

### Big Data大数据

This book proudly focuses on small, in-memory datasets. This is the right place to start because you can’t tackle big data unless you have experience with small data. The tools you learn in this book will easily handle hundreds of megabytes of data, and with a little care you can typically use them to work with 1–2 Gb of data. If you’re routinely working with larger data (10–100 Gb, say), you should learn more about [data.table.](http://bit.ly/Rdatatable) This book doesn’t teach data.table because it has a very concise interface, which makes it harder to learn since it offers fewer linguistic cues. But if you’re working with large data, the performance payoff is worth the extra effort required to learn it. 本书只专注于小型内存数据集。 这是一个正确的起点，因为除非您有小数据经验，否则无法处理大数据。 您在本书中学到的工具可以轻松处理数百兆字节的数据，只需要小心，您通常可以使用它们来处理1-2 Gb的数据。 如果您经常使用更大的数据（例如10-100 Gb），您应该了解有关data.table的更多信息。 本书没有教授data.table，因为它有一个非常简洁的界面，这使得它更难学，因为它提供了更少的语言提示。 但是，如果您正在处理大数据，那么性能支付值得花费额外的努力来学习它。

If your data is bigger than this, carefully consider if your big data problem might actually be a small data problem in disguise. While the complete data might be big, often the data needed to answer a specific question is small. You might be able to find a subset, subsample, or summary that fits in memory and still allows you to answer the question that you’re interested in. The challenge here is finding the right small data, which often requires a lot of iteration. 如果您的数据大于此值，请仔细考虑您的大数据问题是否可能实际上是伪装的小数据问题。 虽然完整的数据可能很大，但回答特定问题所需的数据通常很少。 您可能能够找到适合内存的子集，子样本或摘要，并且仍然允许您回答您感兴趣的问题。这里的挑战是找到正确的小数据，这通常需要大量迭代。

Another possibility is that your big data problem is actually a large number of small data problems. Each individual problem might fit in memory, but you have millions of them. For example, you might want to fit a model to each person in your dataset. That would be trivial if you had just 10 or 100 people, but instead you have a million. Fortunately each problem is independent of the others (a setup that is sometimes called embarrassingly parallel), so you just need a system (like Hadoop or Spark) that allows you to send different datasets to different computers for processing. Once you’ve figured out how to answer the question for a single subset using the tools described in this book, you learn new tools like sparklyr, rhipe, and ddr to solve it for the full dataset. 另一种可能性是你的大数据问题实际上是大量的小数据问题。 每个问题都可能适合记忆，但你有数百万个问题。 例如，您可能希望将模型拟合到数据集中的每个人。 如果你只有10或100个人，那将是微不足道的，但相反，你有一百万。 幸运的是，每个问题都独立于其他问题（这种设置有时被称为令人尴尬的并行），因此您只需要一个系统（如Hadoop或Spark），它允许您将不同的数据集发送到不同的计算机进行处理。 一旦你弄清楚如何使用本书中描述的工具回答单个子集的问题，你就可以学习像sparklyr，rhipe和ddr这样的新工具来解决整个数据集。

### Python, Julia, and Friends

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

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

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

### Nonrectangular Data

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

### Hypothesis Confirmation

It’s possible to divide data analysis into two camps: hypothesis generation and hypothesis confirmation (sometimes called confirmatory analysis). The focus of this book is unabashedly on hypothesis generation, or data exploration. Here you’ll look deeply at the data and, in combination with your subject knowledge, generate many interesting hypotheses to help explain why the data behaves the way it does. You evaluate the hypotheses informally, using your skepticism to challenge the data in multiple ways.

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

You need a precise mathematical model in order to generate falsifiable predictions. This often requires considerable statistical sophistication.

You can only use an observation once to confirm a hypothesis. As soon as you use it more than once you’re back to doing exploratory analysis. This means to do hypothesis confirmation you need to “preregister” (write out in advance) your analysis plan, and not deviate from it even when you have seen the data. We’ll talk a little about some strategies you can use to make this easier in Part IV.

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

## Prerequisites预备知识

We’ve made a few assumptions about what you already know in order to get the most out of this book. You should be generally numerically literate, and it’s helpful if you have some [programming experience already. If you’ve never programmed before, you might find *Hands-On Programming with R* by Garrett to be a useful adjunct to this book.](http://shop.oreilly.com/product/0636920028574.do)

There are four things you need to run the code in this book: R, RStudio, a collection of R packages called the *tidyverse*, and a handful of other packages. Packages are the fundamental units of reproducible R code. They include reusable functions, the documentation that describes how to use them, and sample data.

**R**

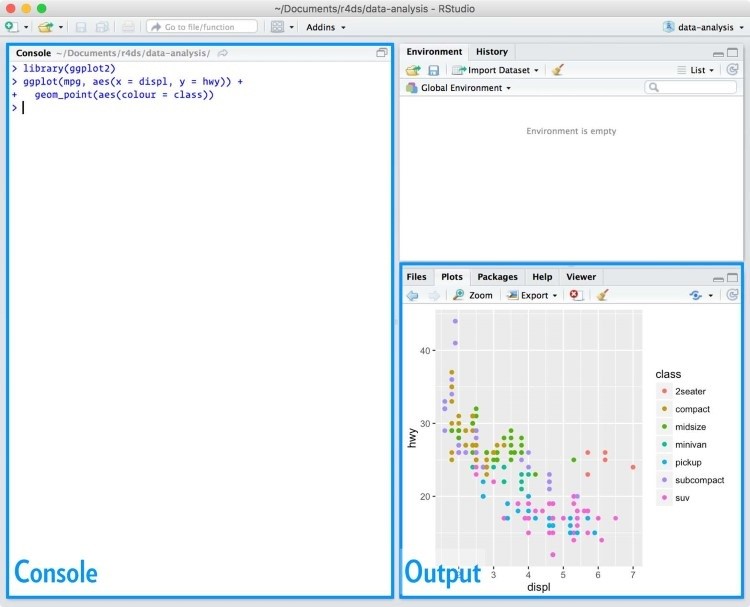
To download R, go to CRAN, the *comprehensive R archive network*. CRAN is composed of a set of mirror servers distributed around the world and is used to distribute R and R packages. Don’t try and pick a mirror that’s close to you: instead use the cloud mirror, [*https://cloud.r-project.org*,](https://cloud.r-project.org/)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

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

### RStudio

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

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



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

### The Tidyverse

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

You can install the complete tidyverse with a single line of code: install.packages("tidyverse")

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

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

library(tidyverse)

*#> Loading tidyverse: ggplot2*

*#> Loading tidyverse: tibble*

*#> Loading tidyverse: tidyr*

*#> Loading tidyverse: readr*

*#> Loading tidyverse: purrr*

*#> Loading tidyverse: dplyr*

*#> Conflicts with tidy packages --------------------------------*

*#> filter(): dplyr, stats*

*#> lag(): dplyr, stats*

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

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

### Other Packages

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

In this book we’ll use three data packages from outside the tidyverse: install.packages(c("nycflights13", "gapminder", "Lahman"))

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

## Running R Code

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

1 + 2

*#> [1] 3*

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

> 1 + 2

[1] 3

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

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

Functions are in a code font and followed by parentheses, like sum() or mean().

Other R objects (like data or function arguments) are in a code font, without parentheses, like flights or x.

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

## Getting Help and Learning More

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

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

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

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

*Packages* should be loaded at the top of the script, so it’s easy to see which ones the example needs. This is a good time to check that you’re using the latest version of each package; it’s possible you’ve discovered a bug that’s been fixed since you installed the package. For packages in the tidyverse, the easiest way to check is to run tidyverse\_update().

The easiest way to include *data* in a question is to use dput() to generate the R code to recreate it. For example, to re-create the mtcars dataset in R, I’d perform the following steps:

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

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

Spend a little bit of time ensuring that your *code* is easy for others to read:

Make sure you’ve used spaces and your variable names are concise, yet informative.

Use comments to indicate where your problem lies.

Do your best to remove everything that is not related to the problem.

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

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

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

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

## Acknowledgments

This book isn’t just the product of Hadley and Garrett, but is the result of many conversations (in person and online) that we’ve had with the many people in the R community. There are a few people we’d like to thank in particular, because they have spent many hours answering our dumb questions and helping us to better think about data science:

Jenny Bryan and Lionel Henry for many helpful discussions around working with lists and list-columns.

[The three chapters on workflow were adapted (with permission) from “R basics, workspace and working directory, RStudio projects” by Jenny Bryan.](http://bit.ly/Rbasicsworkflow)

Genevera Allen for discussions about models, modeling, the statistical learning perspective, and the difference between hypothesis generation and hypothesis confirmation.

Yihui Xie for his work on the [**bookdown**](https://github.com/rstudio/bookdown) package, and for tirelessly responding to my feature requests.

Bill Behrman for his thoughtful reading of the entire book, and for trying it out with his data science class at Stanford.

The #rstats twitter community who reviewed all of the draft chapters and provided tons of useful feedback.

Tal Galili for augmenting his **dendextend** package to support a section on clustering that did not make it into the final draft.

This book was written in the open, and many people contributed pull requests to fix minor problems. Special thanks goes to everyone who contributed via GitHub (listed in alphabetical order): adi pradhan, Ahmed ElGabbas, Ajay Deonarine, @Alex, Andrew Landgraf,

@batpigandme, @behrman, Ben Marwick, Bill Behrman, Brandon Greenwell, Brett Klamer,

Christian G. Warden, Christian Mongeau, Colin Gillespie, Cooper Morris, Curtis Alexander,

Daniel Gromer, David Clark, Derwin McGeary, Devin Pastoor, Dylan Cashman, Earl Brown,

Eric Watt, Etienne B. Racine, Flemming Villalona, Gregory Jefferis, @harrismcgehee, Hengni

Cai, Ian Lyttle, Ian Sealy, Jakub Nowosad, Jennifer (Jenny) Bryan, @jennybc, Jeroen Janssens,

Jim Hester, @jjchern, Joanne Jang, John Sears, Jon Calder, Jonathan Page, @jonathanflint, Julia

Stewart Lowndes, Julian During, Justinas Petuchovas, Kara Woo, @kdpsingh, Kenny Darrell,

Kirill Sevastyanenko, @koalabearski, @KyleHumphrey, Lawrence Wu, Matthew Sedaghatfar,

Mine Cetinkaya-Rundel, @MJMarshall, Mustafa Ascha, @nate-d-olson, Nelson Areal, Nick

Clark, @nickelas, @nwaff, @OaCantona, Patrick Kennedy, Peter Hurford, Rademeyer Vermaak,

Radu Grosu, @rlzijdeman, Robert Schuessler, @robinlovelace, @robinsones, S’busiso

Mkhondwane, @seamus-mckinsey, @seanpwilliams, Shannon Ellis, @shoili, @sibusiso16,

@spirgel, Steve Mortimer, @svenski, Terence Teo, Thomas Klebel, TJ Mahr, Tom Prior, Will Beasley, Yihui Xie.

## Online Version

An online version of this book is available at [*http://r4ds.had.co.nz*.](http://r4ds.had.co.nz/) It will continue to evolve in between reprints of the physical book. The source of the book is available at

[*https://github.com/hadley/r4ds*.](https://github.com/hadley/r4ds) The book is powered by [*https://bookdown.org*](https://bookdown.org/), which makes it easy to turn R markdown files into HTML, PDF, and EPUB.

This book was built with:

devtools::session\_info(c("tidyverse"))

*#> Session info ------------------------------------------------*

*#> setting value*

*#> version R version 3.3.1 (2016-06-21)*

*#> system x86\_64, darwin13.4.0*

*#> ui X11*

*#> language (EN)*

*#> collate en\_US.UTF-8*

*#> tz America/Los\_Angeles*

*#> date 2016-10-10*

*#> Packages ----------------------------------------------------*

*#> package \* version date source*

*#> assertthat 0.1 2013-12-06 CRAN (R 3.3.0)*

*#> BH 1.60.0-2 2016-05-07 CRAN (R 3.3.0)*

*#> broom 0.4.1 2016-06-24 CRAN (R 3.3.0)*

*#> colorspace 1.2-6 2015-03-11 CRAN (R 3.3.0)*

*#> curl 2.1 2016-09-22 CRAN (R 3.3.0)*

*#> DBI 0.5-1 2016-09-10 CRAN (R 3.3.0)*

*#> dichromat 2.0-0 2013-01-24 CRAN (R 3.3.0)*

*#> digest 0.6.10 2016-08-02 CRAN (R 3.3.0)*

*#> dplyr \* 0.5.0 2016-06-24 CRAN (R 3.3.0)*

*#> forcats 0.1.1 2016-09-16 CRAN (R 3.3.0)*

*#> foreign 0.8-67 2016-09-13 CRAN (R 3.3.0) #> ggplot2 \* 2.1.0.9001 2016-10-06 local*

*#> gtable 0.2.0 2016-02-26 CRAN (R 3.3.0) #> haven 1.0.0 2016-09-30 local*

*#> hms 0.2-1 2016-07-28 CRAN (R 3.3.1) #> httr 1.2.1 2016-07-03 cran (@1.2.1)*

*#> jsonlite 1.1 2016-09-14 CRAN (R 3.3.0)*

*#> labeling 0.3 2014-08-23 CRAN (R 3.3.0)*

*#> lattice 0.20-34 2016-09-06 CRAN (R 3.3.0)*

*#> lazyeval 0.2.0 2016-06-12 CRAN (R 3.3.0)*

*#> lubridate 1.6.0 2016-09-13 CRAN (R 3.3.0)*

*#> magrittr 1.5 2014-11-22 CRAN (R 3.3.0)*

*#> MASS 7.3-45 2016-04-21 CRAN (R 3.3.1) #> mime 0.5 2016-07-07 cran (@0.5)*

*#> mnormt 1.5-4 2016-03-09 CRAN (R 3.3.0)*

*#> modelr 0.1.0 2016-08-31 CRAN (R 3.3.0)*

*#> munsell 0.4.3 2016-02-13 CRAN (R 3.3.0)*

*#> nlme 3.1-128 2016-05-10 CRAN (R 3.3.1)*

*#> openssl 0.9.4 2016-05-25 cran (@0.9.4)*

*#> plyr 1.8.4 2016-06-08 cran (@1.8.4)*

*#> psych 1.6.9 2016-09-17 CRAN (R 3.3.0)*

*#> purrr \* 0.2.2 2016-06-18 CRAN (R 3.3.0)*

*#> R6 2.1.3 2016-08-19 CRAN (R 3.3.0)*

*#> RColorBrewer 1.1-2 2014-12-07 CRAN (R 3.3.0)*

*#> Rcpp 0.12.7 2016-09-05 CRAN (R 3.3.0)*

*#> readr \* 1.0.0 2016-08-03 CRAN (R 3.3.0)*

*#> readxl 0.1.1 2016-03-28 CRAN (R 3.3.0)*

*#> reshape2 1.4.1 2014-12-06 CRAN (R 3.3.0)*

*#> rvest 0.3.2 2016-06-17 CRAN (R 3.3.0) #> scales 0.4.0.9003 2016-10-06 local*

*#> selectr 0.3-0 2016-08-30 CRAN (R 3.3.0)*

*#> stringi 1.1.2 2016-10-01 CRAN (R 3.3.1) #> stringr 1.1.0 2016-08-19 cran (@1.1.0)*

*#> tibble \* 1.2 2016-08-26 CRAN (R 3.3.0)*

*#> tidyr \* 0.6.0 2016-08-12 CRAN (R 3.3.0)*

*#> tidyverse \* 1.0.0 2016-09-09 CRAN (R 3.3.0)*

*#> xml2 1.0.0.9001 2016-09-30 local*

## Conventions Used in This Book

The following typographical conventions are used in this book:

*Italic*

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

***Bold***

Indicates the names of R packages.

*Constant width*

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

***Constant width bold***

Shows commands or other text that should be typed literally by the user.

*Constant width italic*

Shows text that should be replaced with user-supplied values or by values determined by context.

**TIP**

This element signifies a tip or suggestion.

## Using Code Examples

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

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

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: “*R for Data Science* by Hadley Wickham and Garrett

Grolemund (O’Reilly). Copyright 2017 Garrett Grolemund, Hadley Wickham, 978-1-491-910399.”

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

## O’Reilly Safari

**NOTE**

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

Members have access to thousands of books, training videos, Learning Paths, interactive tutorials, and curated playlists from over 250 publishers, including O’Reilly Media, Harvard Business

Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que,

Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, and Course Technology, among others.

For more information, please visit [*http://oreilly.com/safari*](http://oreilly.com/safari).

## How to Contact Us

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

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

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

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

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

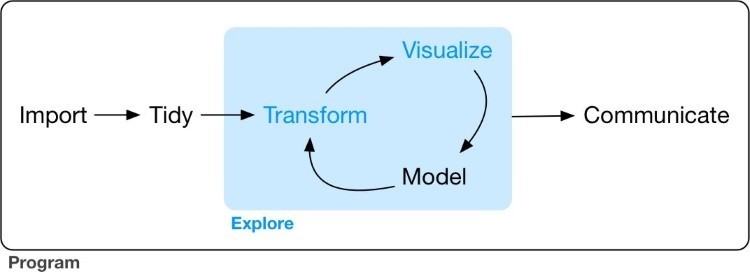
Find us on Facebook: [*http://facebook.com/oreilly*](http://facebook.com/oreilly)

Follow us on Twitter: [*http://twitter.com/oreillymedia*](http://twitter.com/oreillymedia)

Watch us on YouTube: [*http://www.youtube.com/oreillymedia*](http://www.youtube.com/oreillymedia)

# Part I. Explore

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



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

Visualization is a great place to start with R programming, because the payoff is so clear: you get to make elegant and informative plots that help you understand data. In Chapter 1 you’ll dive into visualization, learning the basic structure of a **ggplot2** plot, and powerful techniques for turning data into plots.

Visualization alone is typically not enough, so in Chapter 3 you’ll learn the key verbs that allow you to select important variables, filter out key observations, create new variables, and compute summaries.

Finally, in Chapter 5, you’ll combine visualization and transformation with your curiosity and skepticism to ask and answer interesting questions about data.

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

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

# Chapter 1. Data Visualization with ggplot2

## Introduction

The simple graph has brought more information to the data analyst’s mind than any other device.

John Tukey

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

If you’d like to learn more about the theoretical underpinnings of **ggplot2** before you start, I’d recommend reading [“A Layered Grammar of Graphics”.](http://vita.had.co.nz/papers/layered-grammar.pdf)

### Prerequisites

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

library(tidyverse)

*#> Loading tidyverse: ggplot2*

*#> Loading tidyverse: tibble*

*#> Loading tidyverse: tidyr*

*#> Loading tidyverse: readr*

*#> Loading tidyverse: purrr*

*#> Loading tidyverse: dplyr*

*#> Conflicts with tidy packages --------------------------------*

*#> filter(): dplyr, stats*

*#> lag(): dplyr, stats*

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

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

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

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

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

## First Steps

Let’s use our first graph to answer a question: do cars with big engines use more fuel than cars with small engines? You probably already have an answer, but try to make your answer precise.

What does the relationship between engine size and fuel efficiency look like? Is it positive? Negative? Linear? Nonlinear?

### The mpg Data Frame

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

mpg

*#> # A tibble: 234 × 11*

*#> manufacturer model displ year cyl trans drv*

*#> <chr> <chr> <dbl> <int> <int> <chr> <chr>*

*#> 1 audi a4 1.8 1999 4 auto(l5) f*

*#> 2 audi a4 1.8 1999 4 manual(m5) f*

*#> 3 audi a4 2.0 2008 4 manual(m6) f*

*#> 4 audi a4 2.0 2008 4 auto(av) f*

*#> 5 audi a4 2.8 1999 6 auto(l5) f*

*#> 6 audi a4 2.8 1999 6 manual(m5) f*

*#> # ... with 228 more rows, and 4 more variables: #> # cty <int>, hwy <int>, fl <chr>, class <chr>*

Among the variables in mpg are:

displ, a car’s engine size, in liters.

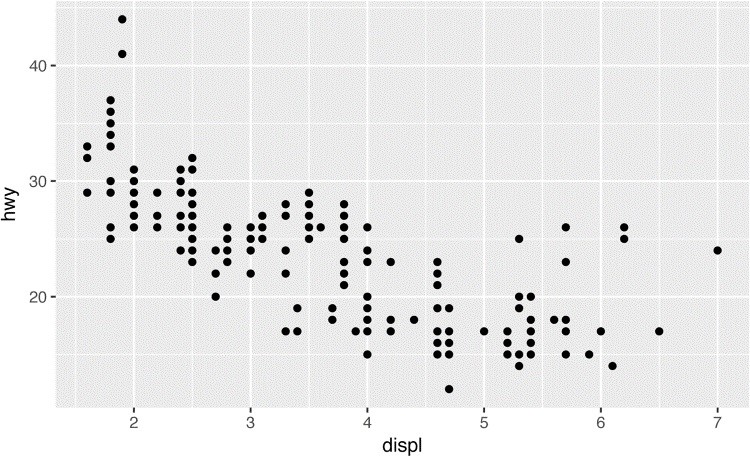
hwy, a car’s fuel efficiency on the highway, in miles per gallon (mpg). A car with a low fuel efficiency consumes more fuel than a car with a high fuel efficiency when they travel the same distance.

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

### Creating a ggplot

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

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy))



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

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

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

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

### A Graphing Template

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

ggplot(data = <DATA>) +

<GEOM\_FUNCTION>(mapping = aes(<MAPPINGS>))

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

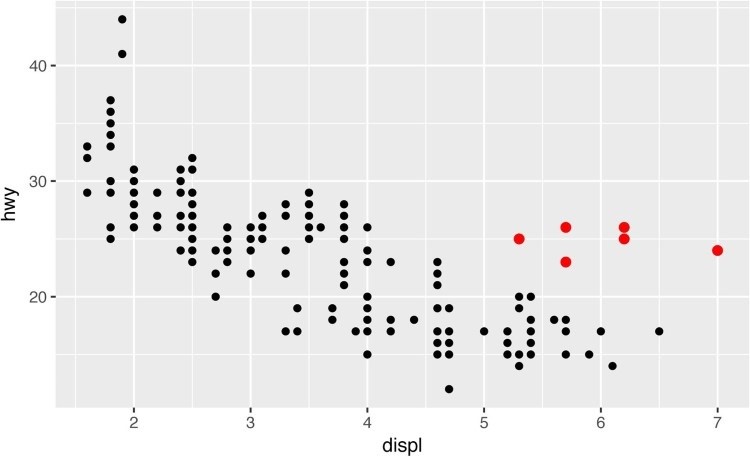
### Exercises

1. Run ggplot(data = mpg). What do you see?
2. How many rows are in mtcars? How many columns?
3. What does the drv variable describe? Read the help for ?mpg to find out.
4. Make a scatterplot of hwy versus cyl.
5. What happens if you make a scatterplot of class versus drv? Why is the plot not useful?

## Aesthetic Mappings

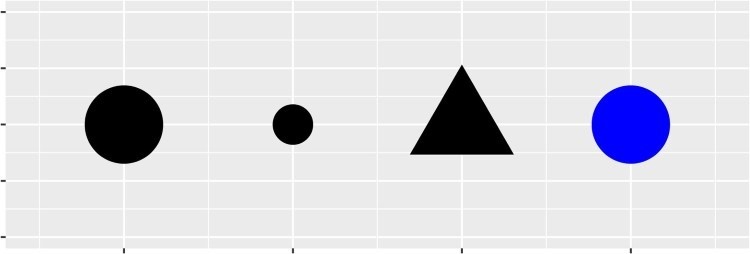
The greatest value of a picture is when it forces us to notice what we never expected to see. John Tukey

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



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

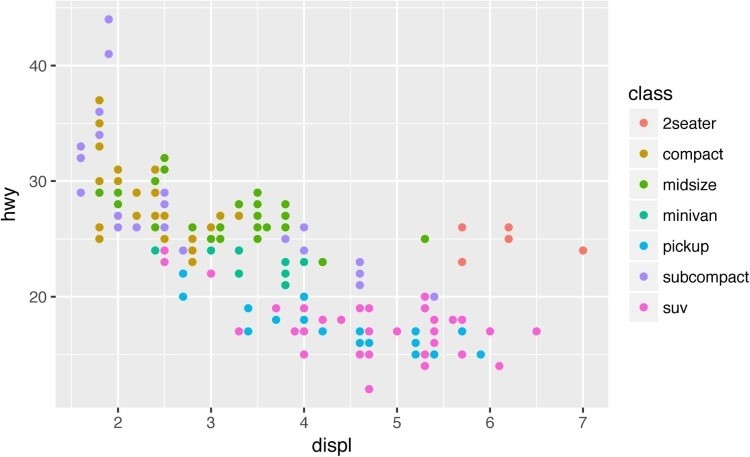
You can add a third variable, like class, to a two-dimensional scatterplot by mapping it to an *aesthetic*. An aesthetic is a visual property of the objects in your plot. Aesthetics include things like the size, the shape, or the color of your points. You can display a point (like the one shown next) in different ways by changing the values of its aesthetic properties. Since we already use the word “value” to describe data, let’s use the word “level” to describe aesthetic properties. Here we change the levels of a point’s size, shape, and color to make the point small, triangular, or blue:



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

ggplot(data = mpg) +

geom\_point(mapping = aes(x = displ, y = hwy, color = class))



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

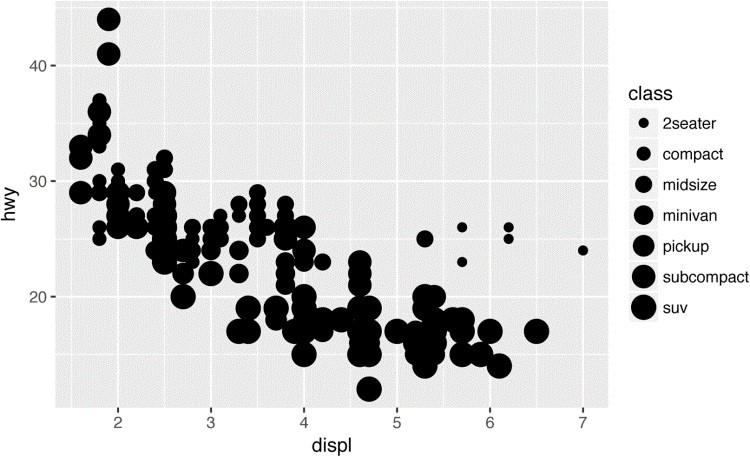
To map an aesthetic to a variable, associate the name of the aesthetic to the name of the variable inside aes(). **ggplot2** will automatically assign a unique level of the aesthetic (here a unique color) to each unique value of the variable, a process known as *scaling*. **ggplot2** will also add a legend that explains which levels correspond to which values.

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

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

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy, size = class))

*#> Warning: Using size for a discrete variable is not advised.*



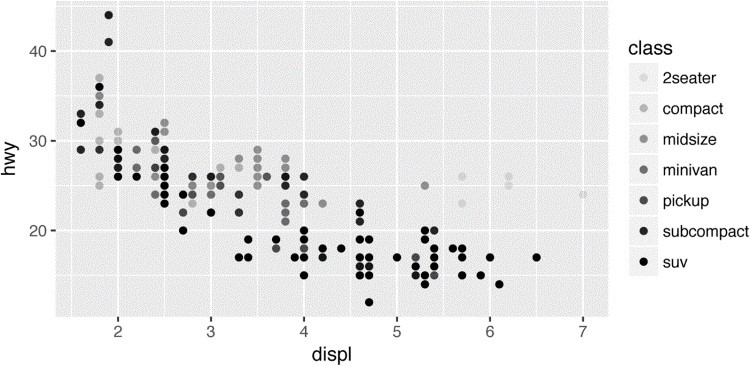
Or we could have mapped class to the *alpha* aesthetic, which controls the transparency of the points, or the shape of the points:

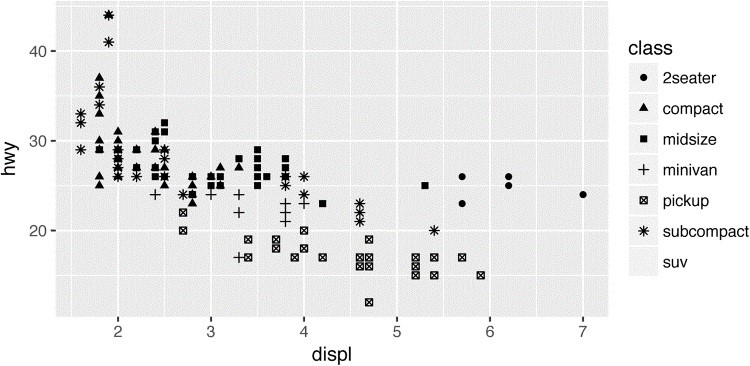
*# Top*

ggplot(data = mpg) +

geom\_point(mapping = aes(x = displ, y = hwy, alpha = class))

*# Bottom* ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy, shape = class))





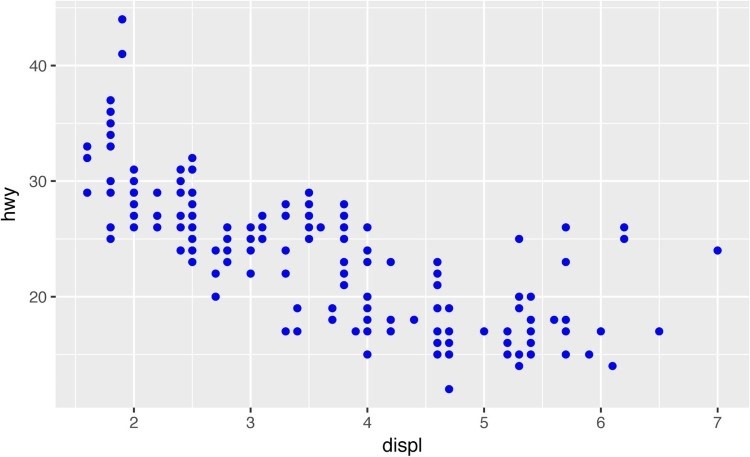
What happened to the SUVs? **ggplot2** will only use six shapes at a time. By default, additional groups will go unplotted when you use this aesthetic.

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

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

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

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy), color = "blue")

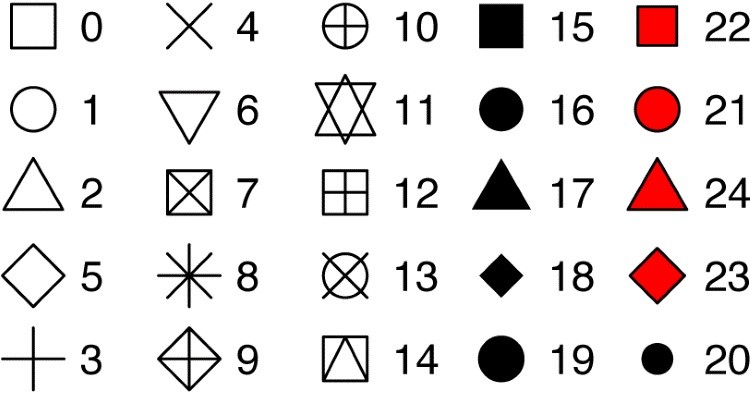


Here, the color doesn’t convey information about a variable, but only changes the appearance of the plot. To set an aesthetic manually, set the aesthetic by name as an argument of your geom function; i.e., it goes *outside* of aes(). You’ll need to pick a value that makes sense for that aesthetic:

The name of a color as a character string.

The size of a point in mm.

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

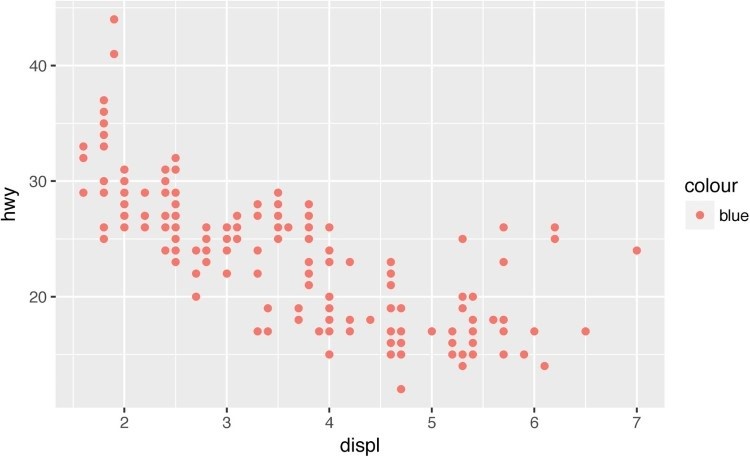


*Figure 1-1. R has 25 built-in shapes that are identified by numbers*

### Exercises

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

ggplot(data = mpg) + geom\_point( mapping = aes(x = displ, y = hwy, color = "blue") )



1. Which variables in mpg are categorical? Which variables are continuous? (Hint: type ?

mpg to read the documentation for the dataset.) How can you see this information when you run mpg?

1. Map a continuous variable to color, size, and shape. How do these aesthetics behave differently for categorical versus continuous variables?
2. What happens if you map the same variable to multiple aesthetics?
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)?

## Common Problems

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

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

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

ggplot(data = mpg)

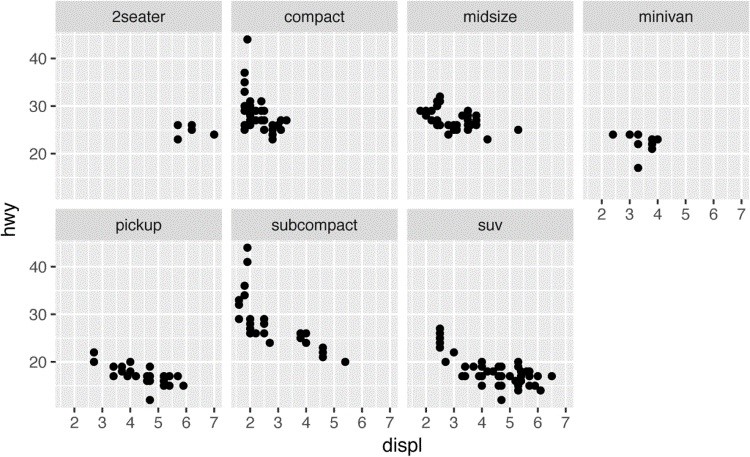
+ geom\_point(mapping = aes(x = displ, y = hwy))

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

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

## Facets

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

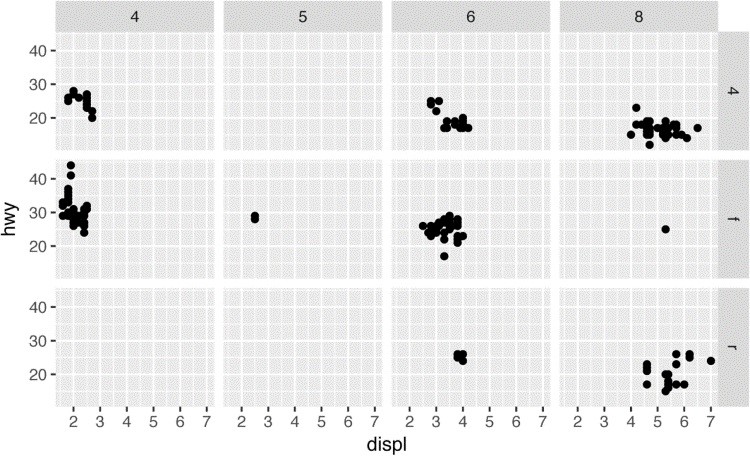


To facet your plot by a single variable, use facet\_wrap(). The first argument of facet\_wrap() should be a formula, which you create with ~ followed by a variable name (here “formula” is the name of a data structure in R, not a synonym for “equation”). The variable that you pass to facet\_wrap() should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

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy)) + facet\_wrap(~ class, nrow = 2)

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

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy)) + facet\_grid(drv ~ cyl)



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

### Exercises

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

ggplot(data = mpg) +

geom\_point(mapping = aes(x = drv, y = cyl))

1. What plots does the following code make? What does . do?

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy)) + facet\_grid(drv ~ .)

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy)) + facet\_grid(. ~ cyl)

1. Take the first faceted plot in this section:

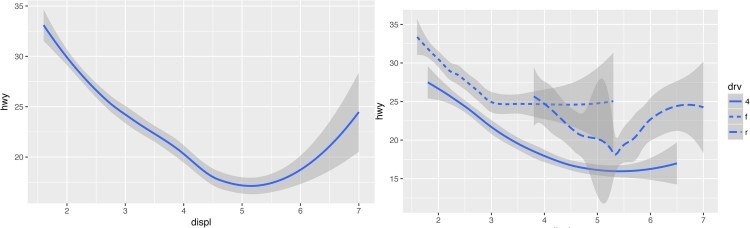
ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy)) + facet\_wrap(~ class, nrow = 2)

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

1. Read ?facet\_wrap. What does nrow do? What does ncol do? What other options control the layout of the individual panels? Why doesn’t facet\_grid() have nrow and ncol variables?
2. When using facet\_grid() you should usually put the variable with more unique levels in the columns. Why?

## Geometric Objects

How are these two plots similar?



Both plots contain the same x variable and the same y variable, and both describe the same data.

But the plots are not identical. Each plot uses a different visual object to represent the data. In **ggplot2** syntax, we say that they use different *geoms*.

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

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

*# left*

ggplot(data = mpg) +

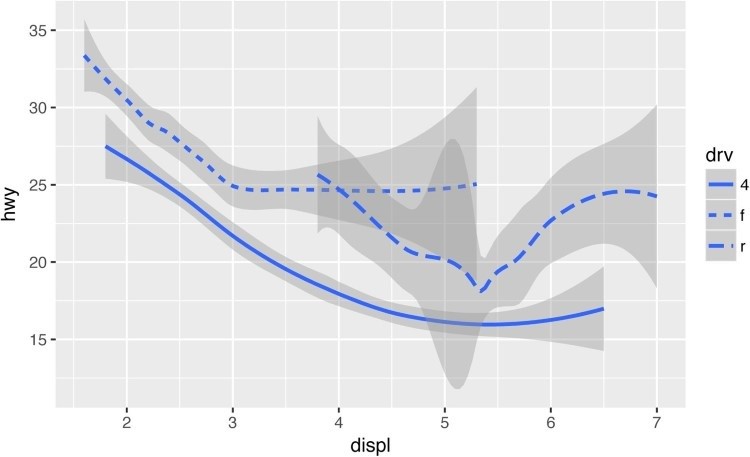
geom\_point(mapping = aes(x = displ, y = hwy))

*# right*

ggplot(data = mpg) + geom\_smooth(mapping = aes(x = displ, y = hwy))

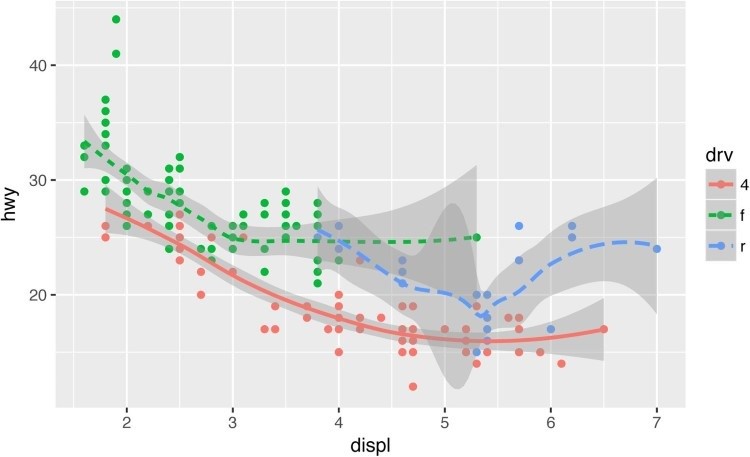
Every geom function in **ggplot2** takes a mapping argument. However, not every aesthetic works with every geom. You could set the shape of a point, but you couldn’t set the “shape” of a line. On the other hand, you *could* set the linetype of a line. geom\_smooth() will draw a different line, with a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

ggplot(data = mpg) + geom\_smooth(mapping = aes(x = displ, y = hwy, linetype = drv))



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

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



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

**ggplot2** provides over 30 geoms, and extension packages provide even more (see

[*https://www.ggplot2-exts.org*](https://www.ggplot2-exts.org/) for a sampling). The best way to get a comprehensive overview is the **ggplot2** cheatsheet, which you can find at [*http://rstudio.com/cheatsheets*](http://rstudio.com/cheatsheets). To learn more about any single geom, use help: ?geom\_smooth.

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

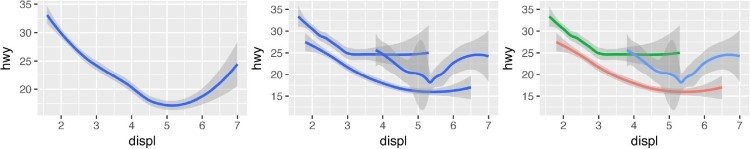
ggplot(data = mpg) +

geom\_smooth(mapping = aes(x = displ, y = hwy))

ggplot(data = mpg) +

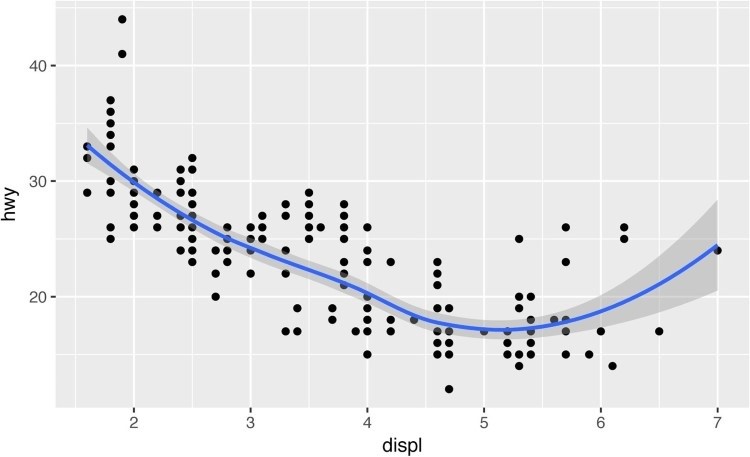
geom\_smooth(mapping = aes(x = displ, y = hwy, group = drv))

ggplot(data = mpg) + geom\_smooth( mapping = aes(x = displ, y = hwy, color = drv), show.legend = **FALSE**  )



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

ggplot(data = mpg) + geom\_point(mapping = aes(x = displ, y = hwy)) + geom\_smooth(mapping = aes(x = displ, y = hwy))

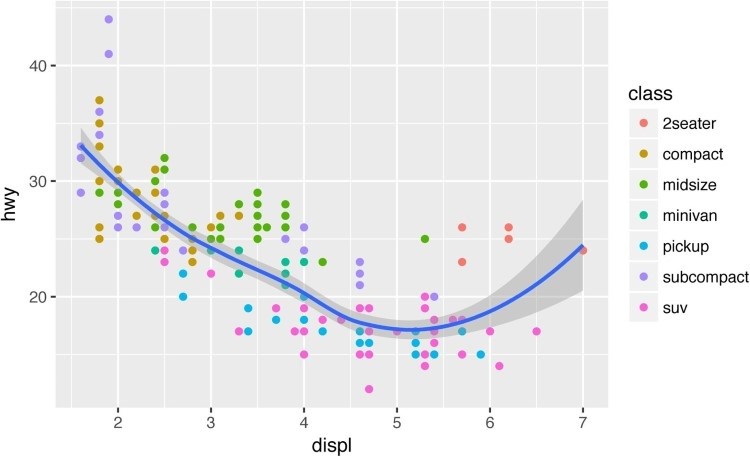


This, however, introduces some duplication in our code. Imagine if you wanted to change the yaxis to display cty instead of hwy. You’d need to change the variable in two places, and you might forget to update one. You can avoid this type of repetition by passing a set of mappings to ggplot(). **ggplot2**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

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom\_point() + geom\_smooth()

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

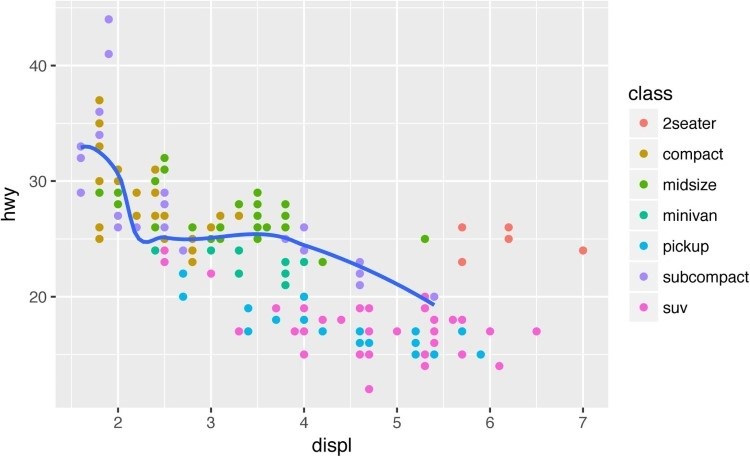
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom\_point(mapping = aes(color = class)) + geom\_smooth()



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

ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom\_point(mapping = aes(color = class)) + geom\_smooth( data = filter(mpg, class == "subcompact"), se = **FALSE**

)



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

### Exercises

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

ggplot( data = mpg, mapping = aes(x = displ, y = hwy, color = drv)

) +

geom\_point() + geom\_smooth(se = **FALSE**)

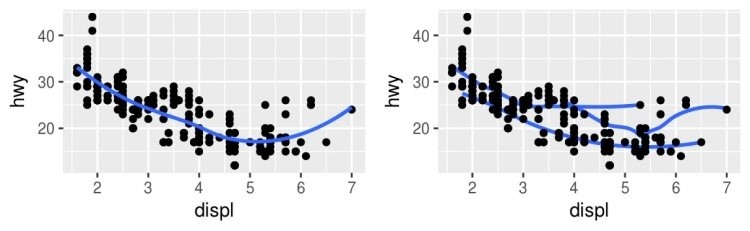
1. What does show.legend = FALSE do? What happens if you remove it? Why do you think I used it earlier in the chapter?
2. What does the se argument to geom\_smooth() do?
3. Will these two graphs look different? Why/why not?

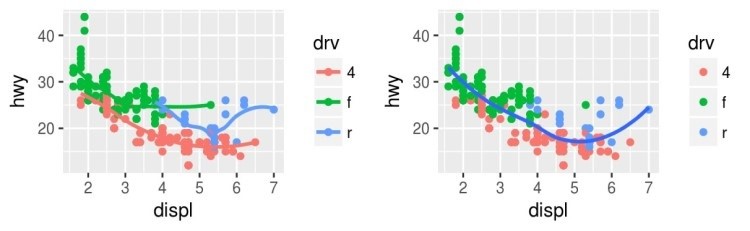
ggplot(data = mpg, mapping = aes(x = displ, y = hwy)) + geom\_point() + geom\_smooth()

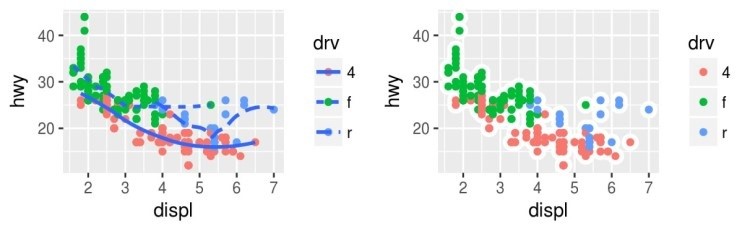
ggplot() + geom\_point( data = mpg, mapping = aes(x = displ, y = hwy)

) + geom\_smooth( data = mpg, mapping = aes(x = displ, y = hwy) )

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



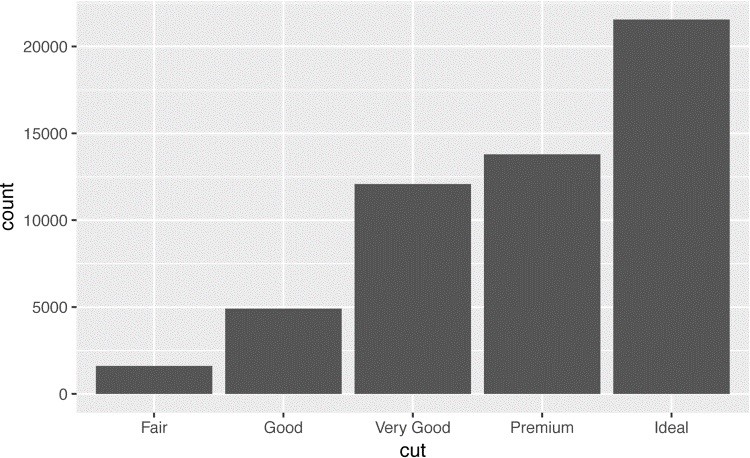




## Statistical Transformations

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

ggplot(data = diamonds) + geom\_bar(mapping = aes(x = cut))



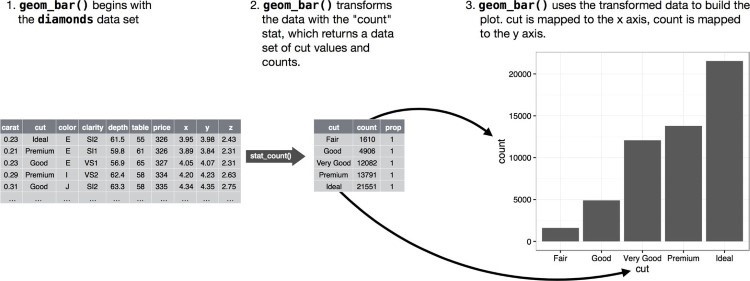
On the x-axis, the chart displays cut, a ‘variable from diamonds. On the y-axis, it displays count, but count is not a variable in diamonds! Where does count come from? Many graphs, like scatterplots, plot the raw values of your dataset. Other graphs, like bar charts, calculate new values to plot:

Bar charts, histograms, and frequency polygons bin your data and then plot bin counts, the number of points that fall in each bin.

Smoothers fit a model to your data and then plot predictions from the model.

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

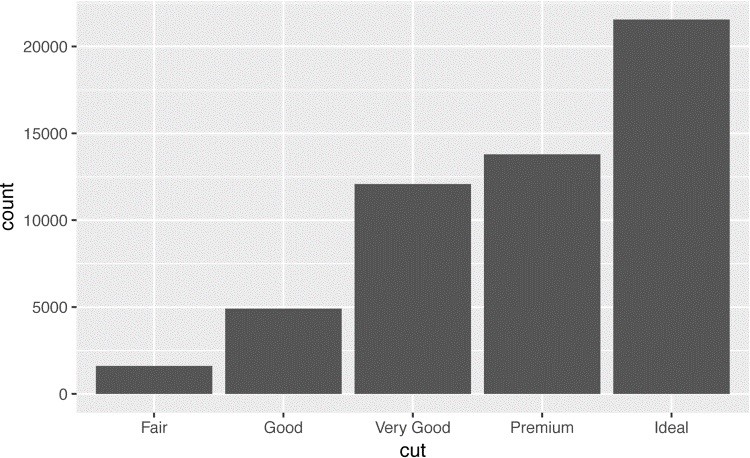
The algorithm used to calculate new values for a graph is called a *stat*, short for statistical transformation. The following figure describes how this process works with geom\_bar().



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

You can generally use geoms and stats interchangeably. For example, you can re-create the previous plot using stat\_count() instead of geom\_bar():

ggplot(data = diamonds) + stat\_count(mapping = aes(x = cut))



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

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

demo <- tribble( ~a, ~b,

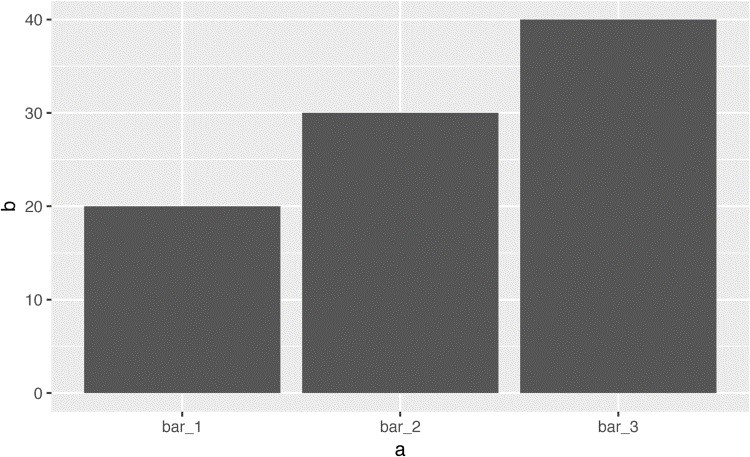
"bar\_1", 20,

"bar\_2", 30,

"bar\_3", 40

)

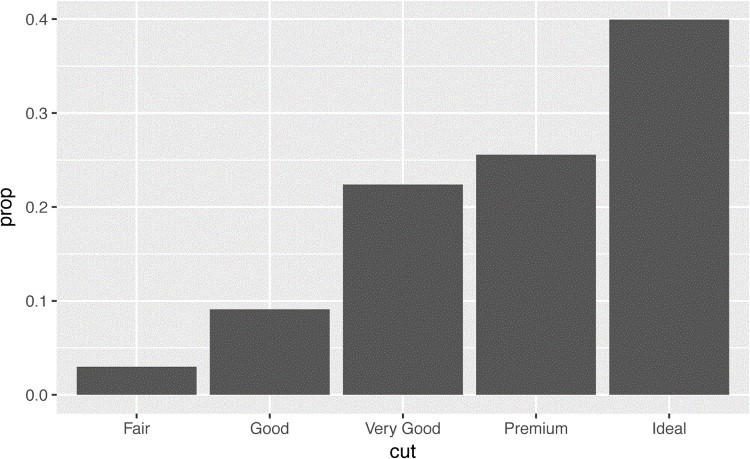
ggplot(data = demo) + geom\_bar( mapping = aes(x = a, y = b), stat = "identity" )



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

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

ggplot(data = diamonds) + geom\_bar( mapping = aes(x = cut, y = ..prop.., group = 1) )

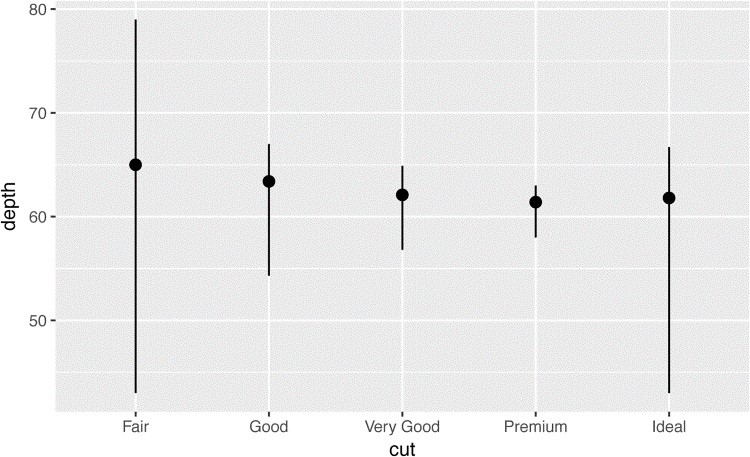


To find the variables computed by the stat, look for the help section titled “Computed variables.”

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

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

)



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

### Exercises

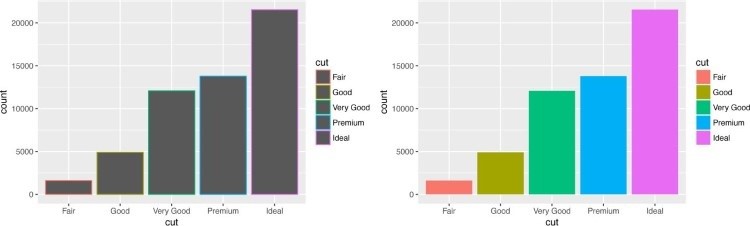
1. What is the default geom associated with stat\_summary()? How could you rewrite the previous plot to use that geom function instead of the stat function?
2. What does geom\_col() do? How is it different to geom\_bar()?
3. Most geoms and stats come in pairs that are almost always used in concert. Read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

ggplot(data = diamonds) + geom\_bar(mapping = aes(x = cut, y = ..prop..)) ggplot(data = diamonds) + geom\_bar( mapping = aes(x = cut, fill = color, y = ..prop..) )

## Position Adjustments

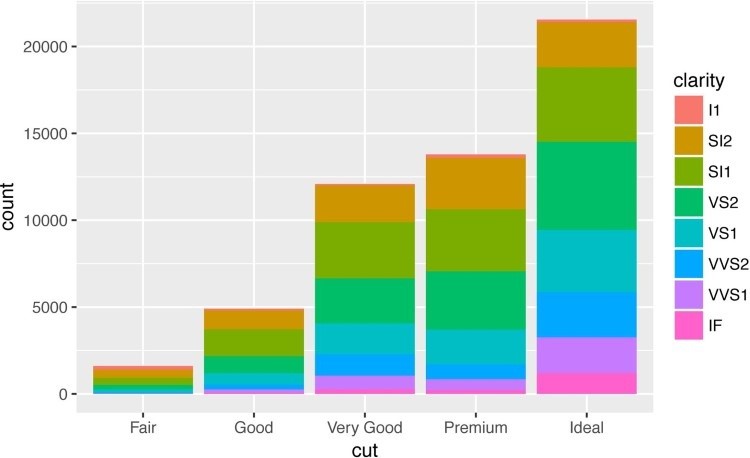
There’s one more piece of magic associated with bar charts. You can color a bar chart using either the color aesthetic, or more usefully, fill:

ggplot(data = diamonds) + geom\_bar(mapping = aes(x = cut, color = cut)) ggplot(data = diamonds) + geom\_bar(mapping = aes(x = cut, fill = cut))



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

ggplot(data = diamonds) + geom\_bar(mapping = aes(x = cut, fill = clarity))



The stacking is performed automatically by the *position adjustment* specified by the position argument. If you don’t want a stacked bar chart, you can use one of three other options:

"identity", "dodge" or "fill":

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

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

) +

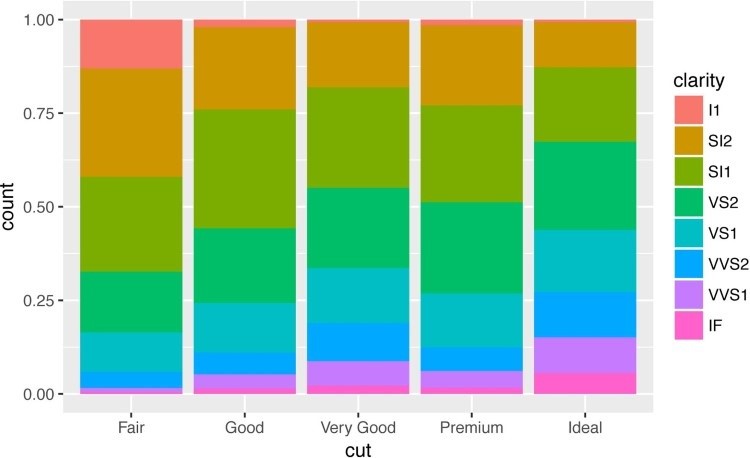
geom\_bar(alpha = 1/5, position = "identity") ggplot( data = diamonds, mapping = aes(x = cut, color = clarity)

) + geom\_bar(fill = **NA**, position = "identity")

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

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

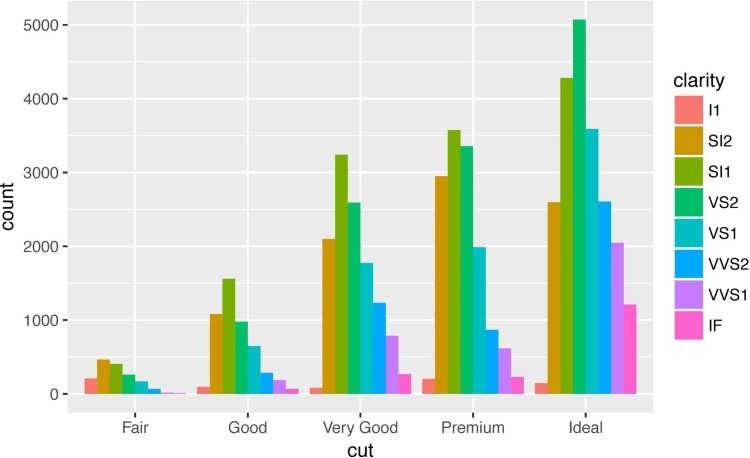
ggplot(data = diamonds) + geom\_bar( mapping = aes(x = cut, fill = clarity), position = "fill" )



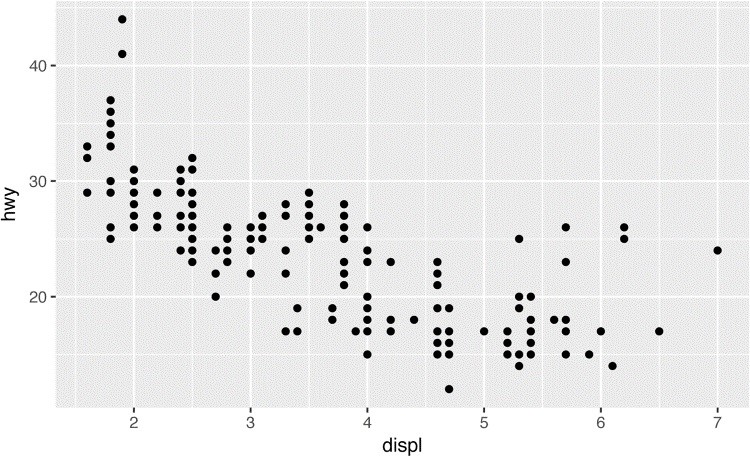
position = "dodge" places overlapping objects directly *beside* one another. This makes it easier to compare individual values:

ggplot(data = diamonds) + geom\_bar( mapping = aes(x = cut, fill = clarity), position = "dodge"

)



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

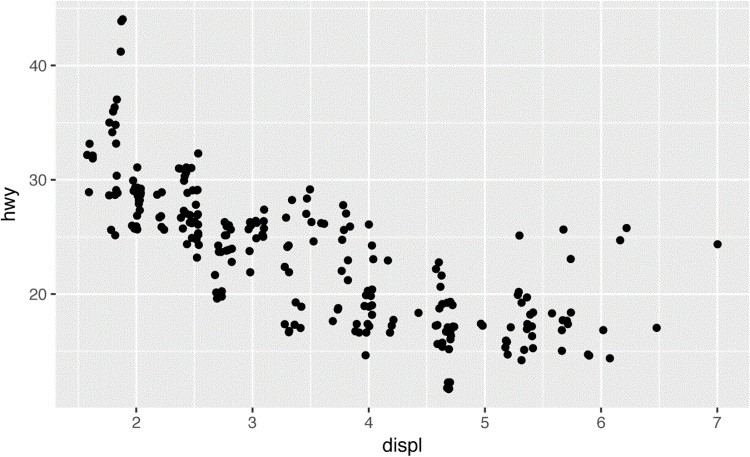


The values of hwy and displ are rounded so the points appear on a grid and many points overlap each other. This problem is known as *overplotting*. This arrangement makes it hard to see where the mass of the data is. Are the data points spread equally throughout the graph, or is there one special combination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

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

ggplot(data = mpg) + geom\_point( mapping = aes(x = displ, y = hwy), position = "jitter"

)



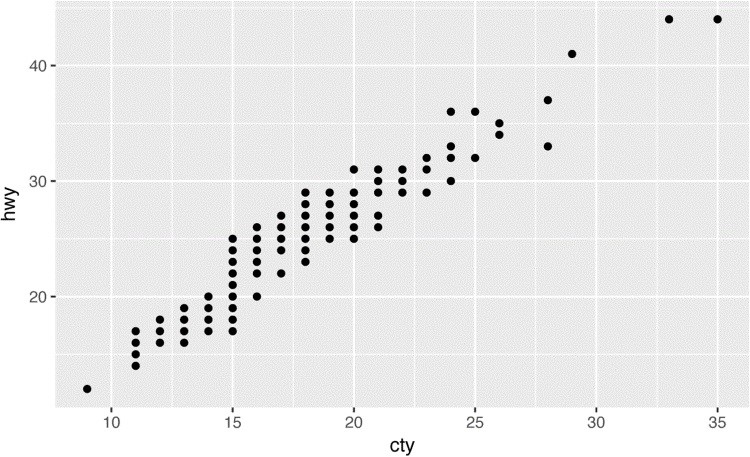
Adding randomness seems like a strange way to improve your plot, but while it makes your graph less accurate at small scales, it makes your graph *more* revealing at large scales. Because this is such a useful operation, **ggplot2** comes with a shorthand for geom\_point(position = "jitter"): geom\_jitter().

To learn more about a position adjustment, look up the help page associated with each adjustment: ?position\_dodge, ?position\_fill, ?position\_identity, ?position\_jitter, and ?position\_stack.

### Exercises

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

ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) + geom\_point()



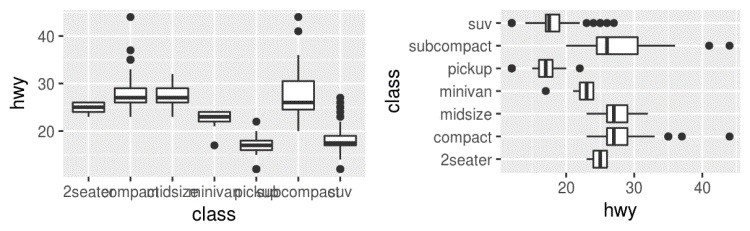
1. What parameters to geom\_jitter() control the amount of jittering?
2. Compare and contrast geom\_jitter() with geom\_count().
3. What’s the default position adjustment for geom\_boxplot()? Create a visualization of the mpg dataset that demonstrates it.

## Coordinate Systems

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

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

ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom\_boxplot() ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom\_boxplot() + coord\_flip()



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

nz <- map\_data("nz")

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

ggplot(nz, aes(long, lat, group = group)) + geom\_polygon(fill = "white", color = "black") + coord\_quickmap()

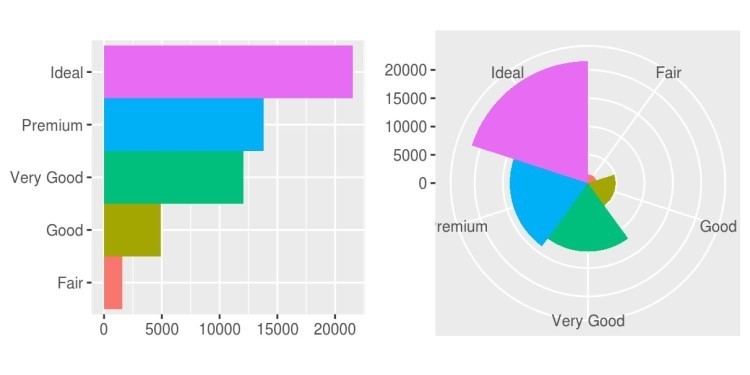


coord\_polar() uses polar coordinates. Polar coordinates reveal an interesting connection between a bar chart and a Coxcomb chart:

bar <- ggplot(data = diamonds) + geom\_bar( mapping = aes(x = cut, fill = cut), show.legend = **FALSE**, width = 1 ) +

theme(aspect.ratio = 1) + labs(x = **NULL**, y = **NULL**)

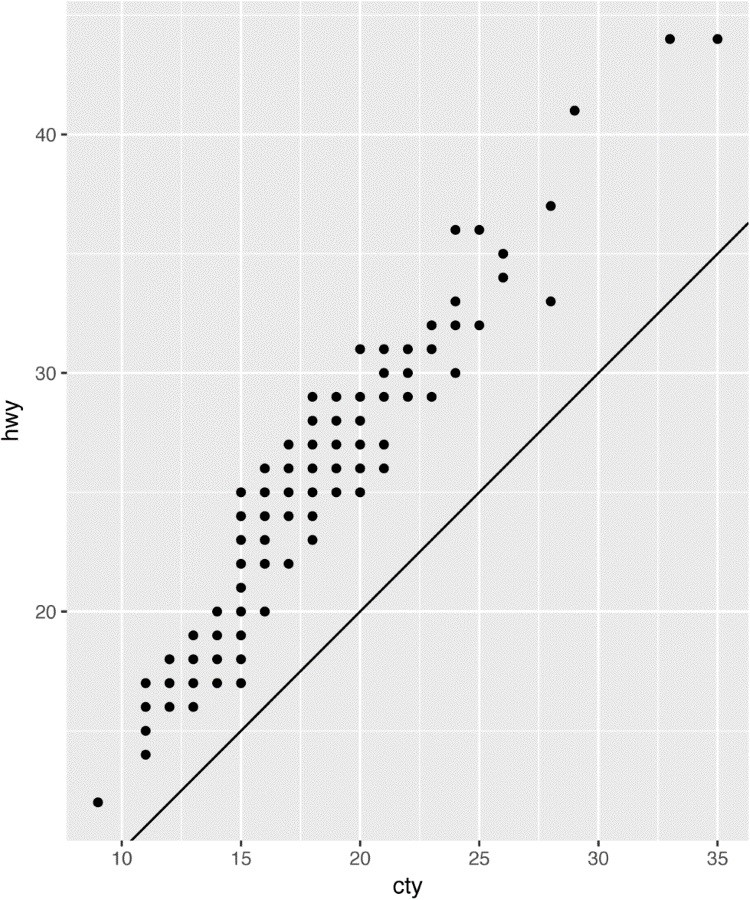
bar + coord\_flip() bar + coord\_polar()



### Exercises

1. Turn a stacked bar chart into a pie chart using coord\_polar().
2. What does labs() do? Read the documentation.
3. What’s the difference between coord\_quickmap() and coord\_map()?
4. What does the following plot tell you about the relationship between city and highway mpg? Why is coord\_fixed() important? What does geom\_abline() do?

ggplot(data = mpg, mapping = aes(x = cty, y = hwy)) + geom\_point() + geom\_abline() + coord\_fixed()



## The Layered Grammar of Graphics

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

ggplot(data = <DATA>) + <GEOM\_FUNCTION>(

mapping = aes(<MAPPINGS>), stat = <STAT>, position = <POSITION>

) +

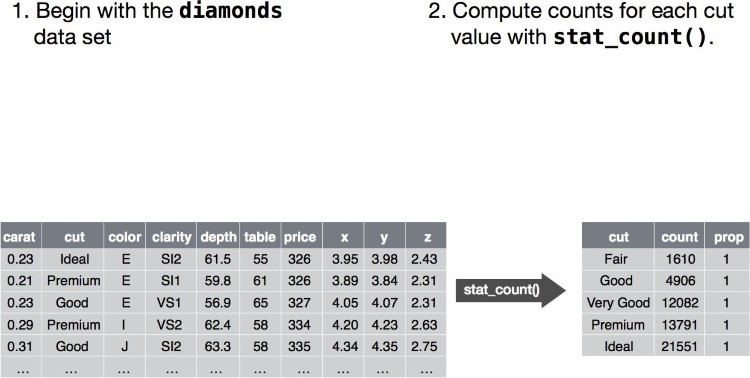
<COORDINATE\_FUNCTION> +

<FACET\_FUNCTION>

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

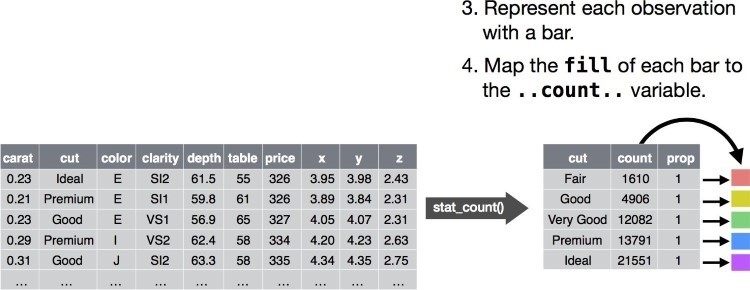
The seven parameters in the template compose the grammar of graphics, a formal system for building plots. The grammar of graphics is based on the insight that you can uniquely describe *any* plot as a combination of a dataset, a geom, a set of mappings, a stat, a position adjustment, a coordinate system,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

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

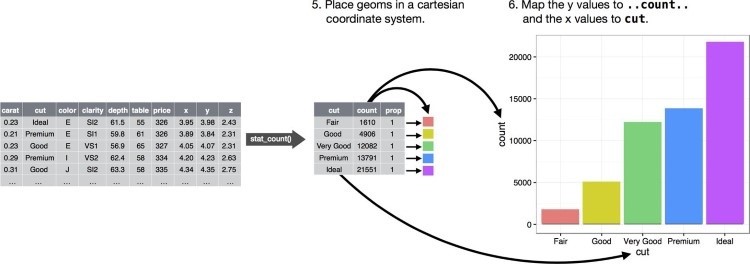


Next, you could choose a geometric object to represent each observation in the transformed data.

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:



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



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