Hadley Wickham

ggplot2

Elegant Graphics for Data Analysis

November 9, 2017

Springer

To my parents, Alison & Brian Wickham. Without them, and their unconditional love and support, none of this would have been possible.

Preface

Welcome to the second edition of "ggplot2: elegant graphics for data analysis". I'm so excited to have an updated book that shows off all the latest and greatest ggplot2 features, as well as the great things that have been happening in R and in the ggplot2 community the last five years. The ggplot2 community is vibrant: the ggplot2 mailing list has over 7,000 members and there is a very active Stack Overflow community, with nearly 10,000 questions tagged with ggplot2. While most of my development effort is no longer going into ggplot2 (more on that below), there's never been a better time to learn it and use it.

I am tremendously grateful for the success of ggplot2. It's one of the most commonly downloaded R packages (over a million downloads in the last year!) and has influenced the design of graphics packages for other languages. Personally, ggplot2 has bought me many exciting opportunities to travel the world and meet interesting people. I love hearing how people are using R and ggplot2 to understand the data that they care about.

A big thanks for this edition goes to Carson Sievert, who helped me modernise the code, including converting the sources to R Markdown. He also updated many of the examples and helped me proofread the book.

Major changes

I've spent a lot of effort ensuring that this edition is a true upgrade over the first. As well as updating the code everywhere to make sure it's fully compatible with the latest version of ggplot2, I have:

- Shown much more code in the book, so it's easier to use as a reference. Overall the book has a more "knitr"-ish sensibility: there are fewer floating figures and tables, and more inline code. This makes the layout a little less pretty but keeps related items closer together.

- Published the complete source online at `https://github.com/hadley/ggplot2-book`.
- Switched from `qplot()` to `ggplot()` in the introduction, Chapter 2. Feedback indicated that `qplot()` was a crutch: it makes simple plots a little easier, but it doesn't help with mastering the grammar.
- Added practice exercises throughout the book so you can practice new techniques immediately after learning about them.
- Added pointers to the rich ecosystem of packages that have built up around ggplot2. You'll now see a number of other packages highlighted in the book, and get pointers to other packages I think are particularly useful.
- Overhauled the toolbox chapter, Chapter 3, to cover all the new geoms. I've added a completely new section on text labels, Section 3.3, since it's important and not covered in detail elsewhere. The mapping section, Section 3.7, has been considerably expanded to talk more about the different types of map data, and where you might find them.
- Completely rewritten the scales chapter, Chapter 6, to focus on the most important tasks. It also discusses the new features that give finer control over legend appearance, Section 6.4, and shows off some of the new scales added to ggplot2, Section 6.6.
- Split the data analysis chapter into three pieces: data tidying (with tidyr), Chapter 9; data manipulation (with dplyr), Chapter 10; and model visualisation (with broom), Chapter 11. I discuss the latest iteration of my data manipulation tools, and introduce the fantastic broom package by David Robinson.

The book is accompanied by a new version of ggplot2: version 2.0.0. This includes a number of minor tweaks and improvements, and considerable improvements to the documentation. Coming back to ggplot2 development after a considerable pause has helped me to see many problems that previously escaped notice. ggplot2 2.0.0 (finally!) contains an official extension mechanism so that others can contribute new ggplot2 components in their own packages. This is documented in a new vignette, `vignette("extending-ggplot2")`.

The future

ggplot2 is now stable, and is unlikely to change much in the future. There will be bug fixes and there may be new geoms, but there will be no large changes to how ggplot2 works. The next iteration of ggplot2 is ggvis. ggvis is significantly more ambitious because it aims to provide a grammar of *interactive* graphics. ggvis is still young, and lacks many of the features of ggplot2 (most notably it currently lacks facetting and has no way to make static graphics), but over the coming years the goal is to make ggvis better than ggplot2.

The syntax of ggvis is a little different to ggplot2. You won't be able to trivially convert your ggplot2 plots to ggvis, but we think the cost is

worth it: the new syntax is considerably more consistent, and will be easier for newcomers to learn. If you've mastered ggplot2, you'll find your skills transfer very well to ggvis and after struggling with the syntax for a while, it will start to feel quite natural. The important skills you learn when mastering ggplot2 are not the programmatic details of describing a plot in code, but the much harder challenge of thinking about how to turn data into effective visualisations.

Acknowledgements

Many people have contributed to this book with high-level structural insights, spelling and grammar corrections and bug reports. I'd particularly like to thank William E. J. Doane, Alexander Forrence, Devin Pastoor, David Robinson, and Guangchuang Yu, for their detailed technical reviews of the book.

Many others have contributed over the (now quite long!) lifetime of ggplot2. I would like to thank: Leland Wilkinson, for discussions and comments that cemented my understanding of the grammar; Gabor Grothendieck, for early helpful comments; Heike Hofmann and Di Cook, for being great advisors and supporting the development of ggplot2 during my PhD; Charlotte Wickham; the students of stat480 and stat503 at ISU, for trying it out when it was very young; Debby Swayne, for masses of helpful feedback and advice; Bob Muenchen, Reinhold Kliegl, Philipp Pagel, Richard Stahlhut, Baptiste Auguie, Jean-Olivier Irisson, Thierry Onkelinx and the many others who have read draft versions of the book and given me feedback; and last, but not least, the members of R-help and the ggplot2 mailing list, for providing the many interesting and challenging graphics problems that have helped motivate this book.

Hadley Wickham
September 2015

Contents

Part I
Getting started

Chapter 1
Introduction

1.1 Welcome to ggplot2

ggplot2 is an R package for producing statistical, or data, graphics, but it is unlike most other graphics packages because it has a deep underlying grammar. This grammar, based on the Grammar of Graphics (Wilkinson 2005), is made up of a set of independent components that can be composed in many different ways. This makes ggplot2 very powerful because you are not limited to a set of pre-specified graphics, but you can create new graphics that are precisely tailored for your problem. This may sound overwhelming, but because there is a simple set of core principles and very few special cases, ggplot2 is also easy to learn (although it may take a little time to forget your preconceptions from other graphics tools).

Practically, ggplot2 provides beautiful, hassle-free plots that take care of fiddly details like drawing legends. The plots can be built up iteratively and edited later. A carefully chosen set of defaults means that most of the time you can produce a publication-quality graphic in seconds, but if you do have special formatting requirements, a comprehensive theming system makes it easy to do what you want. Instead of spending time making your graph look pretty, you can focus on creating a graph that best reveals the messages in your data.

ggplot2 is designed to work iteratively. You can start with a layer showing the raw data then add layers of annotations and statistical summaries. It allows you to produce graphics using the same structured thinking that you use to design an analysis, reducing the distance between a plot in your head and one on the page. It is especially helpful for students who have not yet developed the structured approach to analysis used by experts.

Learning the grammar not only will help you create graphics that you know about now, but will also help you to think about new graphics that would be even better. Without the grammar, there is no underlying theory, so most graphics packages are just a big collection of special cases. For example, in

base R, if you design a new graphic, it's composed of raw plot elements like points and lines, and it's hard to design new components that combine with existing plots. In ggplot2, the expressions used to create a new graphic are composed of higher-level elements like representations of the raw data and statistical transformations, and can easily be combined with new datasets and other plots.

This book provides a hands-on introduction to ggplot2 with lots of example code and graphics. It also explains the grammar on which ggplot2 is based. Like other formal systems, ggplot2 is useful even when you don't understand the underlying model. However, the more you learn about it, the more effectively you'll be able to use ggplot2. This book assumes some basic familiarity with R, to the level described in the first chapter of Dalgaard's *Introductory Statistics with R*.

This book will introduce you to ggplot2 as a novice, unfamiliar with the grammar; teach you the basics so that you can re-create plots you are already familiar with; show you how to use the grammar to create new types of graphics; and eventually turn you into an expert who can build new components to extend the grammar.

1.2 What is the grammar of graphics?

Wilkinson (2005) created the grammar of graphics to describe the deep features that underlie all statistical graphics. The grammar of graphics is an answer to a question: what is a statistical graphic? The layered grammar of graphics (Wickham 2009) builds on Wilkinson's grammar, focussing on the primacy of layers and adapting it for embedding within R. In brief, the grammar tells us that a statistical graphic is a mapping from data to aesthetic attributes (colour, shape, size) of geometric objects (points, lines, bars). The plot may also contain statistical transformations of the data and is drawn on a specific coordinate system. Facetting can be used to generate the same plot for different subsets of the dataset. It is the combination of these independent components that make up a graphic.

As the book progresses, the formal grammar will be explained in increasing detail. The first description of the components follows below. It introduces some of the terminology that will be used throughout the book and outlines the basic responsibilities of each component. Don't worry if it doesn't all make sense right away: you will have many more opportunities to learn about the pieces and how they fit together.

All plots are composed of:

- **Data** that you want to visualise and a set of aesthetic **mapping**s describing how variables in the data are mapped to aesthetic attributes that you can perceive.

- **Layers** made up of geometric elements and statistical transformation. Geometric objects, **geom**s for short, represent what you actually see on the plot: points, lines, polygons, etc. Statistical transformations, **stat**s for short, summarise data in many useful ways. For example, binning and counting observations to create a histogram, or summarising a 2d relationship with a linear model.
- The **scale**s map values in the data space to values in an aesthetic space, whether it be colour, or size, or shape. Scales draw a legend or axes, which provide an inverse mapping to make it possible to read the original data values from the plot.
- A coordinate system, **coord** for short, describes how data coordinates are mapped to the plane of the graphic. It also provides axes and gridlines to make it possible to read the graph. We normally use a Cartesian coordinate system, but a number of others are available, including polar coordinates and map projections.
- A **facet**ing specification describes how to break up the data into subsets and how to display those subsets as small multiples. This is also known as conditioning or latticing/trellising.
- A **theme** which controls the finer points of display, like the font size and background colour. While the defaults in ggplot2 have been chosen with care, you may need to consult other references to create an attractive plot. A good starting place is Tufte's early works (Tufte 1990; Tufte 1997; Tufte 2001).

It is also important to talk about what the grammar doesn't do:

- It doesn't suggest what graphics you should use to answer the questions you are interested in. While this book endeavours to promote a sensible process for producing plots of data, the focus of the book is on how to produce the plots you want, not knowing what plots to produce. For more advice on this topic, you may want to consult Robbins (2013), Cleveland (1993), Chambers et al. (1983), and J. W. Tukey (1977).
- It does not describe interactivity: the grammar of graphics describes only static graphics and there is essentially no benefit to displaying them on a computer screen as opposed to a piece of paper. ggplot2 can only create static graphics, so for dynamic and interactive graphics you will have to look elsewhere (perhaps at ggvis, described below). Cook and Swayne (2007) provides an excellent introduction to the interactive graphics package GGobi. GGobi can be connected to R with the rggobi package (Wickham et al. 2008).

1.3 How does ggplot2 fit in with other R graphics?

There are a number of other graphics systems available in R: base graphics, grid graphics and trellis/lattice graphics. How does ggplot2 differ from them?

- Base graphics were written by Ross Ihaka based on experience implementing the S graphics driver and partly looking at Chambers et al. (1983). Base graphics has a pen on paper model: you can only draw on top of the plot, you cannot modify or delete existing content. There is no (user accessible) representation of the graphics, apart from their appearance on the screen. Base graphics includes both tools for drawing primitives and entire plots. Base graphics functions are generally fast, but have limited scope. If you've created a single scatterplot, or histogram, or a set of boxplots in the past, you've probably used base graphics.
- The development of "grid" graphics, a much richer system of graphical primitives, started in 2000. Grid is developed by Paul Murrell, growing out of his PhD work (Murrell 1998). Grid grobs (graphical objects) can be represented independently of the plot and modified later. A system of viewports (each containing its own coordinate system) makes it easier to lay out complex graphics. Grid provides drawing primitives, but no tools for producing statistical graphics.
- The lattice package, developed by Deepayan Sarkar, uses grid graphics to implement the trellis graphics system of Cleveland (1993) and is a considerable improvement over base graphics. You can easily produce conditioned plots and some plotting details (e.g., legends) are taken care of automatically. However, lattice graphics lacks a formal model, which can make it hard to extend. Lattice graphics are explained in depth in Sarkar (2008).
- ggplot2, started in 2005, is an attempt to take the good things about base and lattice graphics and improve on them with a strong underlying model which supports the production of any kind of statistical graphic, based on the principles outlined above. The solid underlying model of ggplot2 makes it easy to describe a wide range of graphics with a compact syntax, and independent components make extension easy. Like lattice, ggplot2 uses grid to draw the graphics, which means you can exercise much low-level control over the appearance of the plot.
- Work on ggvis, the successor to ggplot2, started in 2014. It takes the foundational ideas of ggplot2 but extends them to the web and interactive graphics. The syntax is similar, but it's been re-designed from scratch to take advantage of what I've learned in the 10 years since creating ggplot2. The most exciting thing about ggvis is that it's interactive and dynamic, so plots automatically re-draw themselves when the underlying data or plot specification changes. However, ggvis is work in progress and currently can create only a fraction of the plots in ggplot2 can. Stay tuned for updates!
- htmlwidgets, `http://www.htmlwidgets.org`, provides a common framework for accessing web visualisation tools from R. Packages built on top

of htmlwidgets include leaflet (https://rstudio.github.io/leaflet/, maps), dygraph (http://rstudio.github.io/dygraphs/, time series) and networkD3 (http://christophergandrud.github.io/networkD3/, networks). htmlwidgets is to ggvis what the many specialised graphic packages are to ggplot2: it provides graphics honed for specific purposes.

Many other R packages, such as vcd (Meyer, Zeileis, and Hornik 2006), plotrix (Lemon et al. 2008) and gplots (Warnes 2007), implement specialist graphics, but no others provide a framework for producing statistical graphics. A comprehensive list of all graphical tools available in other packages can be found in the graphics task view at http://cran.r-project.org/web/views/Graphics.html.

1.4 About this book

The first chapter, Chapter 2, describes how to quickly get started using ggplot2 to make useful graphics. This chapter introduces several important ggplot2 concepts: geoms, aesthetic mappings and facetting. Chapter 3 dives into more details, giving you a toolbox designed to solve a wide range of problems.

Chapter 4 describes the layered grammar of graphics which underlies ggplot2. The theory is illustrated in Chapter 5 which demonstrates how to add additional layers to your plot, exercising full control over the geoms and stats used within them.

Understanding how scales work is crucial for fine-tuning the perceptual properties of your plot. Customising scales gives fine control over the exact appearance of the plot and helps to support the story that you are telling. Chapter 6 will show you what scales are available, how to adjust their parameters, and how to control the appearance of axes and legends.

Coordinate systems and facetting control the position of elements of the plot. These are described in Chapter 7. Facetting is a very powerful graphical tool as it allows you to rapidly compare different subsets of your data. Different coordinate systems are less commonly needed, but are very important for certain types of data.

To polish your plots for publication, you will need to learn about the tools described in Chapter 8. There you will learn about how to control the theming system of ggplot2 and how to save plots to disk.

The book concludes with four chapters that show how to use ggplot2 as part of a data analysis pipeline. ggplot2 works best when your data is tidy, so Chapter 9 discusses what that means and how to make your messy data tidy. Chapter 10 teaches you how to use the dplyr package to perform the most common data manipulation operations. Chapter 11 shows how to integrate visualisation and modelling in two useful ways. Duplicated code is a big inhibitor of flexibility and reduces your ability to respond to changes in

requirements. Chapter 12 covers useful techniques for reducing duplication in your code.

1.5 Installation

To use ggplot2, you must first install it. Make sure you have a recent version of R (at least version 3.2.0) from `http://r-project.org` and then run the following code to download and install ggplot2:

```
install.packages("ggplot2")
```

1.6 Other resources

This book teaches you the elements of ggplot2's grammar and how they fit together, but it does not document every function in complete detail. You will need additional documentation as your use of ggplot2 becomes more complex and varied.

The best resource for specific details of ggplot2 functions and their arguments will always be the built-in documentation. This is accessible online, `http://docs.ggplot2.org/`, and from within R using the usual help syntax. The advantage of the online documentation is that you can see all the example plots and navigate between topics more easily.

If you use ggplot2 regularly, it's a good idea to sign up for the ggplot2 mailing list, `http://groups.google.com/group/ggplot2`. The list has relatively low traffic and is very friendly to new users. Another useful resource is stackoverflow, `http://stackoverflow.com`. There is an active ggplot2 community on stackoverflow, and many common questions have already been asked and answered. In either place, you're much more likely to get help if you create a minimal reproducible example. The reprex (`https://github.com/jennybc/reprex`) package by Jenny Bryan provides a convenient way to do this, and also include advice on creating a good example. The more information you provide, the easier it is for the community to help you.

The number of functions in ggplot2 can be overwhelming, but RStudio provides some great cheatsheets to jog your memory at `http://www.rstudio.com/resources/cheatsheets/`.

Finally, the complete source code for the book is available online at `https://github.com/hadley/ggplot2-book`. This contains the complete text for the book, as well as all the code and data needed to recreate all the plots.

1.7 Colophon

This book was written in R Markdown (http://rmarkdown.rstudio.com/) inside RStudio (http://www.rstudio.com/ide/). knitr (http://yihui.name/knitr/) and pandoc (http://johnmacfarlane.net/pandoc/) converted the raw Rmarkdown to html and pdf. The complete source is available from github (https://github.com/hadley/ggplot2-book). This version of the book was built with:

```
devtools::session_info(c("ggplot2", "dplyr", "broom"))
#> Session info -------------------------------------------------------
#>  setting  value
#>  version  R version 3.4.2 (2017-09-28)
#>  system   x86_64, linux-gnu
#>  ui       X11
#>  language (EN)
#>  collate  en_CA.UTF-8
#>  tz       Canada/East-Saskatchewan
#>  date     2017-11-09
#> Packages -----------------------------------------------------------
#>  package      * version  date       source
#>  assertthat     0.2.0    2017-04-11 CRAN (R 3.4.1)
#>  BH             1.65.0-1 2017-08-24 CRAN (R 3.4.1)
#>  bindr          0.1      2016-11-13 CRAN (R 3.4.1)
#>  bindrcpp     * 0.2      2017-06-17 CRAN (R 3.4.1)
#>  broom          0.4.2    2017-02-13 CRAN (R 3.4.2)
#>  colorspace     1.3-2    2016-12-14 CRAN (R 3.4.1)
#>  compiler       3.4.2    2017-10-28 local
#>  dichromat      2.0-0    2013-01-24 CRAN (R 3.4.1)
#>  digest         0.6.12   2017-01-27 CRAN (R 3.4.1)
#>  dplyr        * 0.7.4    2017-09-28 CRAN (R 3.4.2)
#>  foreign        0.8-69   2017-06-21 CRAN (R 3.4.1)
#>  ggplot2      * 2.2.1    2016-12-30 CRAN (R 3.4.1)
#>  glue           1.2.0    2017-10-29 CRAN (R 3.4.2)
#>  graphics     * 3.4.2    2017-10-28 local
#>  grDevices    * 3.4.2    2017-10-28 local
#>  grid           3.4.2    2017-10-28 local
#>  gtable         0.2.0    2016-02-26 CRAN (R 3.4.1)
#>  labeling       0.3      2014-08-23 CRAN (R 3.4.1)
#>  lattice        0.20-35  2017-03-25 CRAN (R 3.3.3)
#>  lazyeval       0.2.1    2017-10-29 CRAN (R 3.4.2)
#>  magrittr       1.5      2014-11-22 CRAN (R 3.4.1)
#>  MASS           7.3-47   2017-04-21 CRAN (R 3.4.0)
#>  methods      * 3.4.2    2017-10-28 local
#>  mnormt         1.5-5    2016-10-15 CRAN (R 3.4.2)
```

```
#>  munsell        0.4.3    2016-02-13 CRAN (R 3.4.1)
#>  nlme           3.1-131  2017-02-06 CRAN (R 3.4.0)
#>  parallel       3.4.2    2017-10-28 local
#>  pkgconfig      2.0.1    2017-03-21 CRAN (R 3.4.1)
#>  plogr          0.1-1    2016-09-24 CRAN (R 3.4.1)
#>  plyr           1.8.4    2016-06-08 CRAN (R 3.4.1)
#>  psych          1.7.8    2017-09-09 CRAN (R 3.4.2)
#>  purrr          0.2.4    2017-10-18 CRAN (R 3.4.2)
#>  R6             2.2.2    2017-06-17 CRAN (R 3.4.1)
#>  RColorBrewer   1.1-2    2014-12-07 CRAN (R 3.4.1)
#>  Rcpp           0.12.13  2017-09-28 CRAN (R 3.4.2)
#>  reshape2       1.4.2    2016-10-22 CRAN (R 3.4.1)
#>  rlang          0.1.4    2017-11-05 CRAN (R 3.4.2)
#>  scales         0.5.0    2017-08-24 CRAN (R 3.4.1)
#>  stats        * 3.4.2    2017-10-28 local
#>  stringi        1.1.5    2017-04-07 CRAN (R 3.4.1)
#>  stringr        1.2.0    2017-02-18 CRAN (R 3.4.1)
#>  tibble         1.3.4    2017-08-22 CRAN (R 3.4.1)
#>  tidyr        * 0.7.2    2017-10-16 CRAN (R 3.4.2)
#>  tidyselect     0.2.3    2017-11-06 CRAN (R 3.4.2)
#>  tools          3.4.2    2017-10-28 local
#>  utils        * 3.4.2    2017-10-28 local
#>  viridisLite    0.2.0    2017-03-24 CRAN (R 3.4.1)
getOption("width")
#> [1] 67
```

References

Chambers, John, William Cleveland, Beat Kleiner, and Paul Tukey. 1983. *Graphical Methods for Data Analysis*. Wadsworth.

Cleveland, William. 1993. *Visualizing Data*. Hobart Press.

Cook, Dianne, and Deborah F. Swayne. 2007. *Interactive and Dynamic Graphics for Data Analysis: With Examples Using R and Ggobi*. Springer.

Lemon, Jim, Ben Bolker, Sander Oom, Eduardo Klein, Barry Rowlingson, Hadley Wickham, Anupam Tyagi, et al. 2008. *Plotrix: Various Plotting Functions*.

Meyer, David, Achim Zeileis, and Kurt Hornik. 2006. "The Strucplot Framework: Visualizing Multi-Way Contingency Tables with Vcd." *Journal of Statistical Software* 17 (3): 1–48. http://www.jstatsoft.org/v17/i03/.

Murrell, Paul. 1998. "Investigations in Graphical Statistics." PhD thesis, The University of Auckland.

Robbins, Naomi. 2013. *Creating More Effective Graphs*. Chart House.

Sarkar, Deepayan. 2008. *Lattice: Multivariate Data Visualization with R*. Springer.

Tufte, Edward R. 1990. *Envisioning Information*. Graphics Press.

———. 1997. *Visual Explanations*. Graphics Press.

———. 2001. *The Visual Display of Quantitative Information*. Second. Graphics Press.

Tukey, John W. 1977. *Exploratory Data Analysis*. Addison–Wesley.

Warnes, Gregory. 2007. *Gplots: Various R Programming Tools for Plotting Data*.

Wickham, Hadley. 2009. "A Layered Grammar of Graphics." *Journal of Computational and Graphical Statistics*.

Wickham, Hadley, Michael Lawrence, Duncan Temple Lang, and Deborah F Swayne. 2008. "An Introduction to Rggobi." *R-News* 8 (2): 3–7. http://CRAN.R-project.org/doc/Rnews/Rnews_2008-2.pdf.

Wilkinson, Leland. 2005. *The Grammar of Graphics*. 2nd ed. Statistics and Computing. Springer.

Chapter 2
Getting started with ggplot2

2.1 Introduction

The goal of this chapter is to teach you how to produce useful graphics with ggplot2 as quickly as possible. You'll learn the basics of `ggplot()` along with some useful "recipes" to make the most important plots. `ggplot()` allows you to make complex plots with just a few lines of code because it's based on a rich underlying theory, the grammar of graphics. Here we'll skip the theory and focus on the practice, and in later chapters you'll learn how to use the full expressive power of the grammar.

In this chapter you'll learn:

- About the `mpg` dataset included with ggplot2, Section 2.2.
- The three key components of every plot: data, aesthetics and geoms, Section 2.3.
- How to add additional variables to a plot with aesthetics, Section 2.4.
- How to display additional categorical variables in a plot using small multiples created by facetting, Section 2.5.
- A variety of different geoms that you can use to create different types of plots, Section 2.6.
- How to modify the axes, Section 2.7.
- Things you can do with a plot object other than display it, like save it to disk, Section 2.8.
- `qplot()`, a handy shortcut for when you just want to quickly bang out a simple plot without thinking about the grammar at all, Section 2.9.

2.2 Fuel economy data

In this chapter, we'll mostly use one data set that's bundled with ggplot2: `mpg`. It includes information about the fuel economy of popular car models

in 1999 and 2008, collected by the US Environmental Protection Agency, `http://fueleconomy.gov`. You can access the data by loading ggplot2:

```
library(ggplot2)
mpg
#> # A tibble: 234 x 11
#>   manufacturer model displ  year   cyl      trans   drv   cty
#>          <chr> <chr> <dbl> <int> <int>      <chr> <chr> <int>
#> 1         audi    a4   1.8  1999     4   auto(l5)     f    18
#> 2         audi    a4   1.8  1999     4 manual(m5)     f    21
#> 3         audi    a4   2.0  2008     4 manual(m6)     f    20
#> 4         audi    a4   2.0  2008     4   auto(av)     f    21
#> 5         audi    a4   2.8  1999     6   auto(l5)     f    16
#> 6         audi    a4   2.8  1999     6 manual(m5)     f    18
#> # ... with 228 more rows, and 3 more variables: hwy <int>,
#> #   fl <chr>, class <chr>
```

The variables are mostly self-explanatory:

- `cty` and `hwy` record miles per gallon (mpg) for city and highway driving.
- `displ` is the engine displacement in litres.
- `drv` is the drivetrain: front wheel (f), rear wheel (r) or four wheel (4).
- `model` is the model of car. There are 38 models, selected because they had a new edition every year between 1999 and 2008.
- `class` (not shown), is a categorical variable describing the "type" of car: two seater, SUV, compact, etc.

This dataset suggests many interesting questions. How are engine size and fuel economy related? Do certain manufacturers care more about fuel economy than others? Has fuel economy improved in the last ten years? We will try to answer some of these questions, and in the process learn how to create some basic plots with ggplot2.

2.2.1 Exercises

1. List five functions that you could use to get more information about the `mpg` dataset.
2. How can you find out what other datasets are included with ggplot2?
3. Apart from the US, most countries use fuel consumption (fuel consumed over fixed distance) rather than fuel economy (distance travelled with fixed amount of fuel). How could you convert `cty` and `hwy` into the European standard of l/100km?
4. Which manufacturer has the most the models in this dataset? Which model has the most variations? Does your answer change if you remove the re-

dundant specification of drive train (e.g. “pathfinder 4wd”, “a4 quattro”) from the model name?

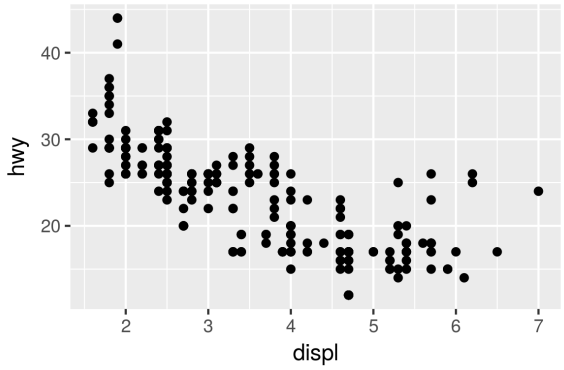
2.3 Key components

Every ggplot2 plot has three key components:

1. **data**,
2. A set of **aesthetic mappings** between variables in the data and visual properties, and
3. At least one layer which describes how to render each observation. Layers are usually created with a **geom** function.

Here’s a simple example:

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



This produces a scatterplot defined by:

1. Data: `mpg`.
2. Aesthetic mapping: engine size mapped to x position, fuel economy to y position.
3. Layer: points.

Pay attention to the structure of this function call: data and aesthetic mappings are supplied in `ggplot()`, then layers are added on with `+`. This is an important pattern, and as you learn more about ggplot2 you’ll construct increasingly sophisticated plots by adding on more types of components.

Almost every plot maps a variable to `x` and `y`, so naming these aesthetics is tedious, so the first two unnamed arguments to `aes()` will be mapped to `x` and `y`. This means that the following code is identical to the example above:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point()
```

I'll stick to that style throughout the book, so don't forget that the first two arguments to `aes()` are `x` and `y`. Note that I've put each command on a new line. I recommend doing this in your own code, so it's easy to scan a plot specification and see exactly what's there. In this chapter, I'll sometimes use just one line per plot, because it makes it easier to see the differences between plot variations.

The plot shows a strong correlation: as the engine size gets bigger, the fuel economy gets worse. There are also some interesting outliers: some cars with large engines get higher fuel economy than average. What sort of cars do you think they are?

2.3.1 Exercises

1. How would you describe the relationship between `cty` and `hwy`? Do you have any concerns about drawing conclusions from that plot?
2. What does `ggplot(mpg, aes(model, manufacturer)) + geom_point()` show? Is it useful? How could you modify the data to make it more informative?
3. Describe the data, aesthetic mappings and layers used for each of the following plots. You'll need to guess a little because you haven't seen all the datasets and functions yet, but use your common sense! See if you can predict what the plot will look like before running the code.

    1. `ggplot(mpg, aes(cty, hwy)) + geom_point()`
    2. `ggplot(diamonds, aes(carat, price)) + geom_point()`
    3. `ggplot(economics, aes(date, unemploy)) + geom_line()`
    4. `ggplot(mpg, aes(cty)) + geom_histogram()`

2.4 Colour, size, shape and other aesthetic attributes

To add additional variables to a plot, we can use other aesthetics like colour, shape, and size (NB: while I use British spelling throughout this book, ggplot2 also accepts American spellings). These work in the same way as the `x` and `y` aesthetics, and are added into the call to `aes()`:

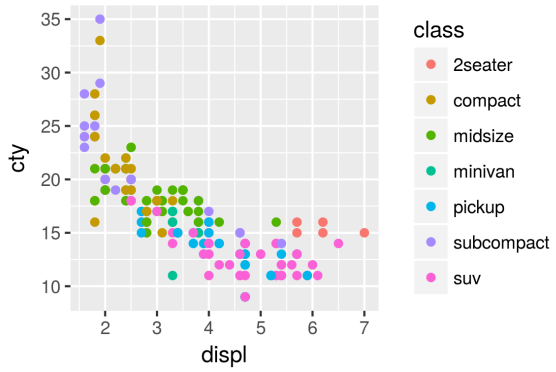
- `aes(displ, hwy, colour = class)`
- `aes(displ, hwy, shape = drv)`

- `aes(displ, hwy, size = cyl)`

ggplot2 takes care of the details of converting data (e.g., 'f', 'r', '4') into aesthetics (e.g., 'red', 'yellow', 'green') with a **scale**. There is one scale for each aesthetic mapping in a plot. The scale is also responsible for creating a guide, an axis or legend, that allows you to read the plot, converting aesthetic values back into data values. For now, we'll stick with the default scales provided by ggplot2. You'll learn how to override them in Chapter 6.

To learn more about those outlying variables in the previous scatterplot, we could map the class variable to colour:

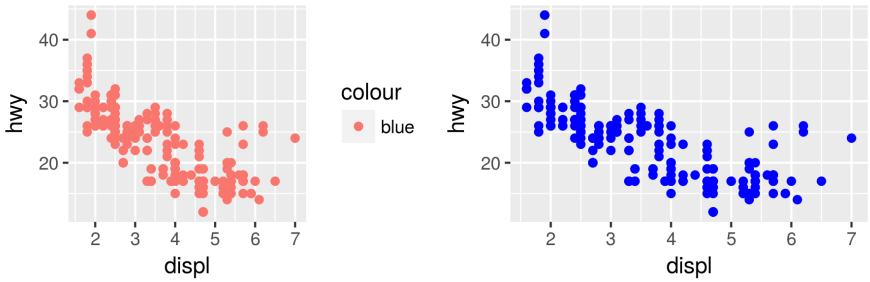
```
ggplot(mpg, aes(displ, cty, colour = class)) +
  geom_point()
```



This gives each point a unique colour corresponding to its class. The legend allows us to read data values from the colour, showing us that the group of cars with unusually high fuel economy for their engine size are two seaters: cars with big engines, but lightweight bodies.

If you want to set an aesthetic to a fixed value, without scaling it, do so in the individual layer outside of `aes()`. Compare the following two plots:

```
ggplot(mpg, aes(displ, hwy)) + geom_point(aes(colour = "blue"))
ggplot(mpg, aes(displ, hwy)) + geom_point(colour = "blue")
```



In the first plot, the value "blue" is scaled to a pinkish colour, and a legend is added. In the second plot, the points are given the R colour blue. This is an important technique and you'll learn more about it in Section 5.4.2. See `vignette("ggplot2-specs")` for the values needed for colour and other aesthetics.

Different types of aesthetic attributes work better with different types of variables. For example, colour and shape work well with categorical variables, while size works well for continuous variables. The amount of data also makes a difference: if there is a lot of data it can be hard to distinguish different groups. An alternative solution is to use facetting, as described next.

When using aesthetics in a plot, less is usually more. It's difficult to see the simultaneous relationships among colour and shape and size, so exercise restraint when using aesthetics. Instead of trying to make one very complex plot that shows everything at once, see if you can create a series of simple plots that tell a story, leading the reader from ignorance to knowledge.

2.4.1 Exercises

1. Experiment with the colour, shape and size aesthetics. What happens when you map them to continuous values? What about categorical values? What happens when you use more than one aesthetic in a plot?
2. What happens if you map a continuous variable to shape? Why? What happens if you map `trans` to shape? Why?
3. How is drive train related to fuel economy? How is drive train related to engine size and class?

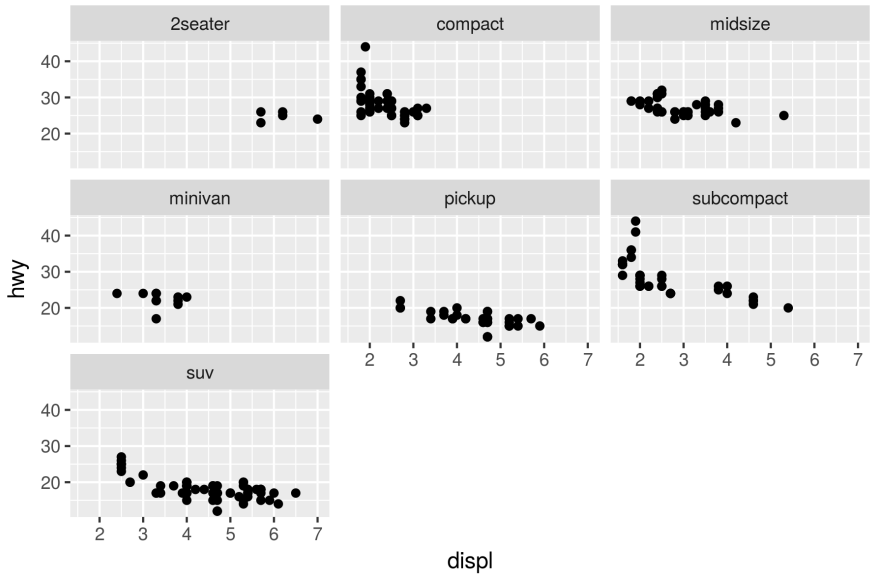
2.5 Facetting

Another technique for displaying additional categorical variables on a plot is facetting. Facetting creates tables of graphics by splitting the data into

subsets and displaying the same graph for each subset. You'll learn more about facetting in Section 7.2, but it's such a useful technique that you need to know it right away.

There are two types of facetting: grid and wrapped. Wrapped is the most useful, so we'll discuss it here, and you can learn about grid facetting later. To facet a plot you simply add a facetting specification with `facet_wrap()`, which takes the name of a variable preceded by `~`.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(~class)
```



You might wonder when to use facetting and when to use aesthetics. You'll learn more about the relative advantages and disadvantages of each in Section 7.2.5.

2.5.1 Exercises

1. What happens if you try to facet by a continuous variable like `hwy`? What about `cyl`? What's the key difference?
2. Use facetting to explore the 3-way relationship between fuel economy, engine size, and number of cylinders. How does facetting by number of cylin-

ders change your assessement of the relationship between engine size and fuel economy?
3. Read the documentation for `facet_wrap()`. What arguments can you use to control how many rows and columns appear in the output?
4. What does the `scales` argument to `facet_wrap()` do? When might you use it?

2.6 Plot geoms

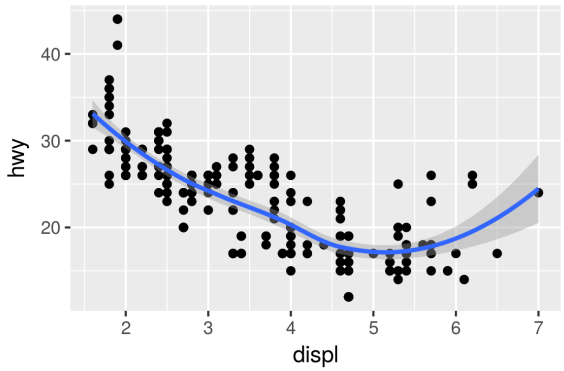
You might guess that by substituting `geom_point()` for a different geom function, you'd get a different type of plot. That's a great guess! In the following sections, you'll learn about some of the other important geoms provided in ggplot2. This isn't an exhaustive list, but should cover the most commonly used plot types. You'll learn more in Chapter 3.

- `geom_smooth()` fits a smoother to the data and displays the smooth and its standard error.
- `geom_boxplot()` produces a box-and-whisker plot to summarise the distribution of a set of points.
- `geom_histogram()` and `geom_freqpoly()` show the distribution of continuous variables.
- `geom_bar()` shows the distribution of categorical variables.
- `geom_path()` and `geom_line()` draw lines between the data points. A line plot is constrained to produce lines that travel from left to right, while paths can go in any direction. Lines are typically used to explore how things change over time.

2.6.1 Adding a smoother to a plot

If you have a scatterplot with a lot of noise, it can be hard to see the dominant pattern. In this case it's useful to add a smoothed line to the plot with `geom_smooth()`:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess'
```

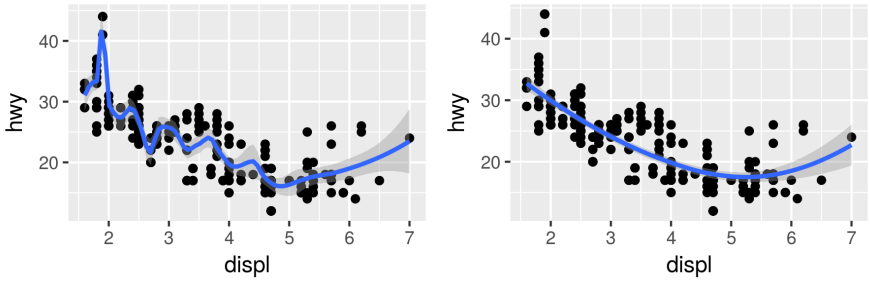


This overlays the scatterplot with a smooth curve, including an assessment of uncertainty in the form of point-wise confidence intervals shown in grey. If you're not interested in the confidence interval, turn it off with `geom_smooth(se = FALSE)`.

An important argument to `geom_smooth()` is the `method`, which allows you to choose which type of model is used to fit the smooth curve:

- `method = "loess"`, the default for small n, uses a smooth local regression (as described in `?loess`). The wiggliness of the line is controlled by the `span` parameter, which ranges from 0 (exceedingly wiggly) to 1 (not so wiggly).

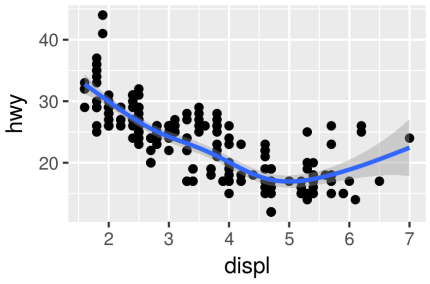
```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(span = 0.2)
#> `geom_smooth()` using method = 'loess'

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(span = 1)
#> `geom_smooth()` using method = 'loess'
```



Loess does not work well for large datasets (it's $O(n^2)$ in memory), so an alternative smoothing algorithm is used when n is greater than 1,000.

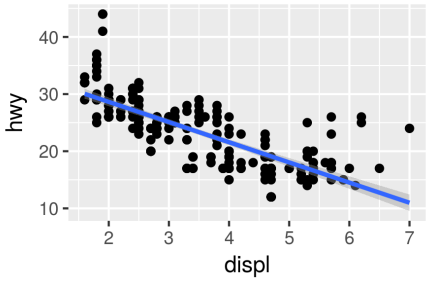
- `method = "gam"` fits a generalised additive model provided by the **mgcv** package. You need to first load mgcv, then use a formula like `formula = y ~ s(x)` or `y ~ s(x, bs = "cs")` (for large data). This is what ggplot2 uses when there are more than 1,000 points.

```
library(mgcv)
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(method = "gam", formula = y ~ s(x))
```



- `method = "lm"` fits a linear model, giving the line of best fit.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(method = "lm")
```

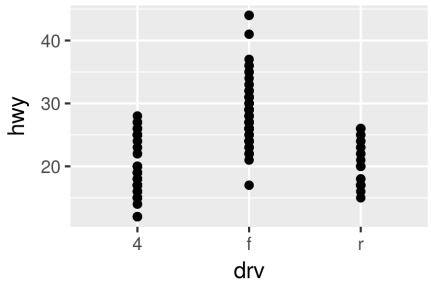


- `method = "rlm"` works like `lm()`, but uses a robust fitting algorithm so that outliers don't affect the fit as much. It's part of the **MASS** package, so remember to load that first.

2.6.2 Boxplots and jittered points

When a set of data includes a categorical variable and one or more continuous variables, you will probably be interested to know how the values of the continuous variables vary with the levels of the categorical variable. Say we're interested in seeing how fuel economy varies within car class. We might start with a scatterplot like this:

```
ggplot(mpg, aes(drv, hwy)) +
  geom_point()
```

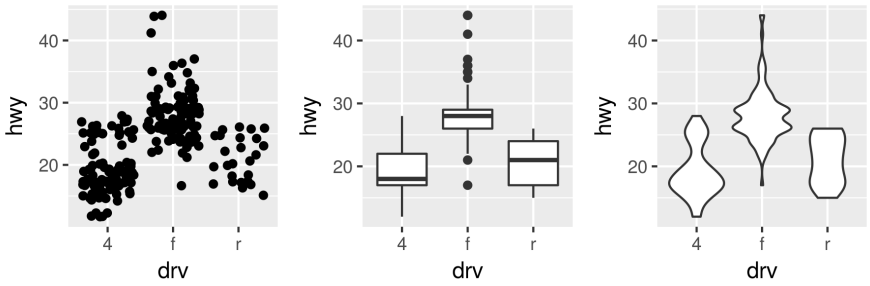


Because there are few unique values of both class and hwy, there is a lot of overplotting. Many points are plotted in the same location, and it's difficult to see the distribution. There are three useful techniques that help alleviate the problem:

- Jittering, `geom_jitter()`, adds a little random noise to the data which can help avoid overplotting.
- Boxplots, `geom_boxplot()`, summarise the shape of the distribution with a handful of summary statistics.
- Violin plots, `geom_violin()`, show a compact representation of the "density" of the distribution, highlighting the areas where more points are found.

These are illustrated below:

```
ggplot(mpg, aes(drv, hwy)) + geom_jitter()
ggplot(mpg, aes(drv, hwy)) + geom_boxplot()
ggplot(mpg, aes(drv, hwy)) + geom_violin()
```



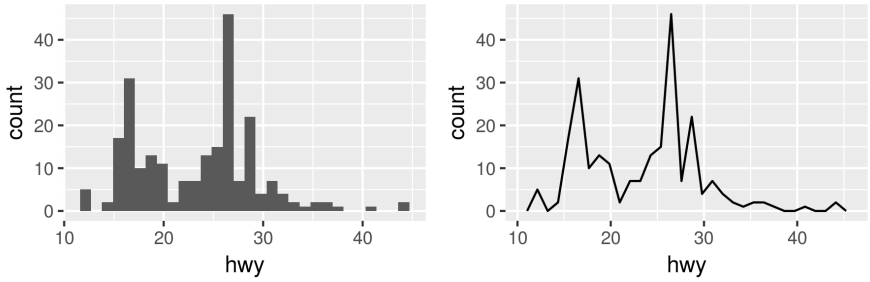
Each method has its strengths and weaknesses. Boxplots summarise the bulk of the distribution with only five numbers, while jittered plots show every point but only work with relatively small datasets. Violin plots give the richest display, but rely on the calculation of a density estimate, which can be hard to interpret.

For jittered points, `geom_jitter()` offers the same control over aesthetics as `geom_point()`: `size`, `colour`, and `shape`. For `geom_boxplot()` and `geom_violin()`, you can control the outline `colour` or the internal `fill` colour.

2.6.3 Histograms and frequency polygons

Histograms and frequency polygons show the distribution of a single numeric variable. They provide more information about the distribution of a single group than boxplots do, at the expense of needing more space.

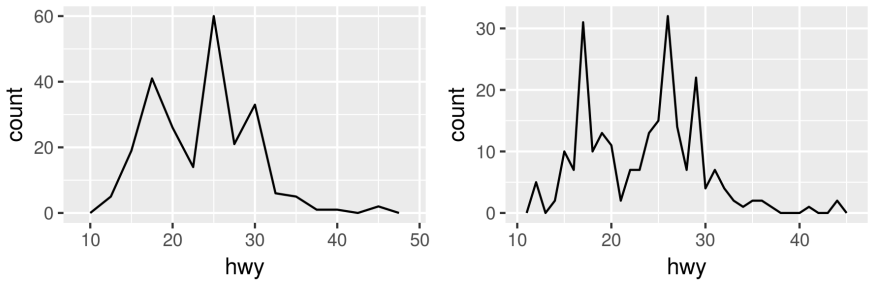
```
ggplot(mpg, aes(hwy)) + geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
ggplot(mpg, aes(hwy)) + geom_freqpoly()
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```



Both histograms and frequency polygons work in the same way: they bin the data, then count the number of observations in each bin. The only difference is the display: histograms use bars and frequency polygons use lines.

You can control the width of the bins with the `binwidth` argument (if you don't want evenly spaced bins you can use the `breaks` argument). It is **very important** to experiment with the bin width. The default just splits your data into 30 bins, which is unlikely to be the best choice. You should always try many bin widths, and you may find you need multiple bin widths to tell the full story of your data.

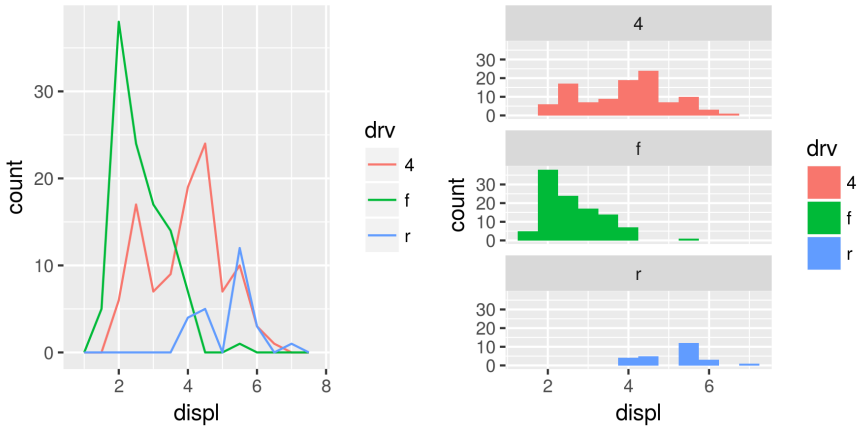
```
ggplot(mpg, aes(hwy)) +
  geom_freqpoly(binwidth = 2.5)
ggplot(mpg, aes(hwy)) +
  geom_freqpoly(binwidth = 1)
```



An alternative to the frequency polygon is the density plot, `geom_density()`. I'm not a fan of density plots because they are harder to interpret since the underlying computations are more complex. They also make assumptions that are not true for all data, namely that the underlying distribution is continuous, unbounded, and smooth.

To compare the distributions of different subgroups, you can map a categorical variable to either fill (for `geom_histogram()`) or colour (for `geom_freqpoly()`). It's easier to compare distributions using the frequency polygon because the underlying perceptual task is easier. You can also use facetting: this makes comparisons a little harder, but it's easier to see the distribution of each group.

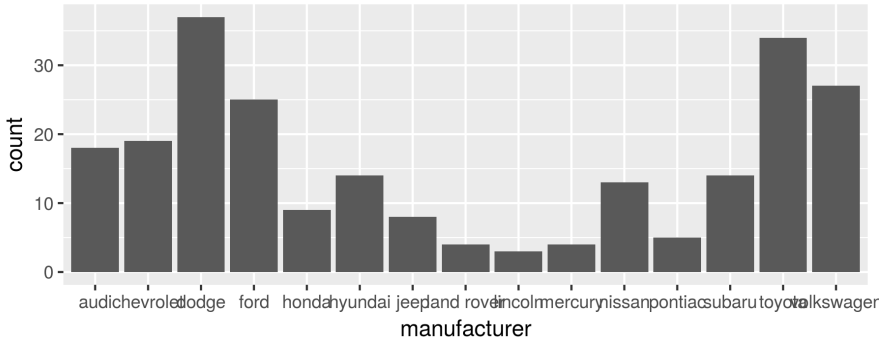
```
ggplot(mpg, aes(displ, colour = drv)) +
  geom_freqpoly(binwidth = 0.5)
ggplot(mpg, aes(displ, fill = drv)) +
  geom_histogram(binwidth = 0.5) +
  facet_wrap(~drv, ncol = 1)
```



2.6.4 Bar charts

The discrete analogue of the histogram is the bar chart, `geom_bar()`. It's easy to use:

```
ggplot(mpg, aes(manufacturer)) +
  geom_bar()
```



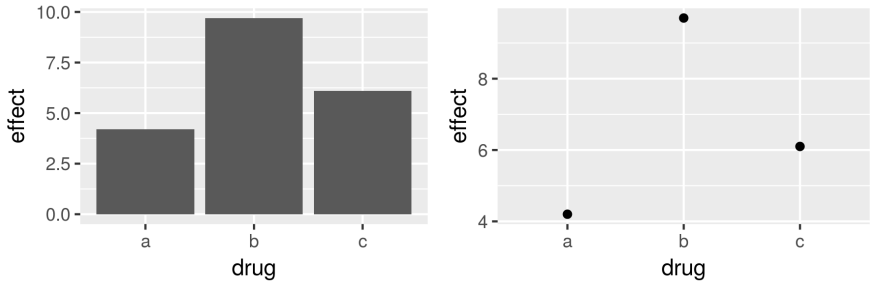
(You'll learn how to fix the labels in Section 8.4.2).

Bar charts can be confusing because there are two rather different plots that are both commonly called bar charts. The above form expects you to have unsummarised data, and each observation contributes one unit to the height of each bar. The other form of bar chart is used for presummarised data. For example, you might have three drugs with their average effect:

```
drugs <- data.frame(
  drug = c("a", "b", "c"),
  effect = c(4.2, 9.7, 6.1)
)
```

To display this sort of data, you need to tell `geom_bar()` to not run the default stat which bins and counts the data. However, I think it's even better to use `geom_point()` because points take up less space than bars, and don't require that the y axis includes 0.

```
ggplot(drugs, aes(drug, effect)) + geom_bar(stat = "identity")
ggplot(drugs, aes(drug, effect)) + geom_point()
```

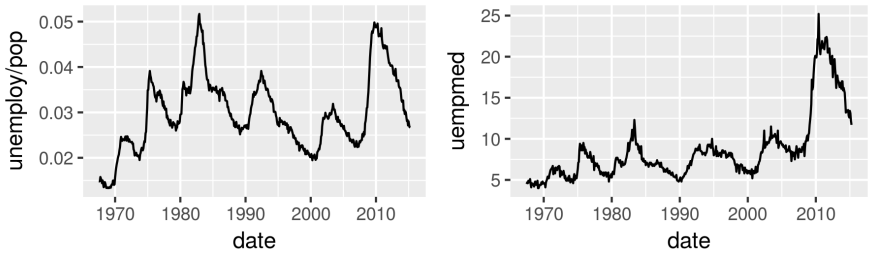


2.6.5 Time series with line and path plots

Line and path plots are typically used for time series data. Line plots join the points from left to right, while path plots join them in the order that they appear in the dataset (in other words, a line plot is a path plot of the data sorted by x value). Line plots usually have time on the x-axis, showing how a single variable has changed over time. Path plots show how two variables have simultaneously changed over time, with time encoded in the way that observations are connected.

Because the year variable in the `mpg` dataset only has two values, we'll show some time series plots using the `economics` dataset, which contains economic data on the US measured over the last 40 years. The figure below shows two plots of unemployment over time, both produced using `geom_line()`. The first shows the unemployment rate while the second shows the median number of weeks unemployed. We can already see some differences in these two variables, particularly in the last peak, where the unemployment percentage is lower than it was in the preceding peaks, but the length of unemployment is high.

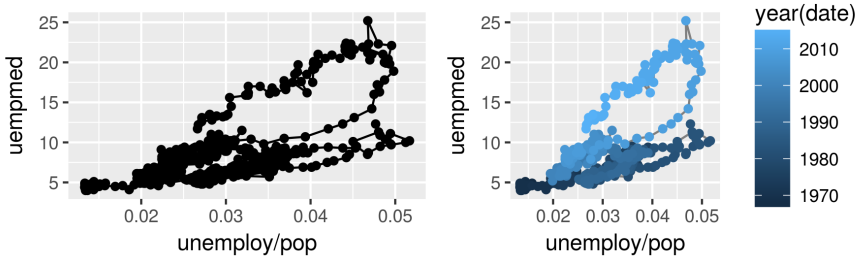
```
ggplot(economics, aes(date, unemploy / pop)) +
  geom_line()
ggplot(economics, aes(date, uempmed)) +
  geom_line()
```



To examine this relationship in greater detail, we would like to draw both time series on the same plot. We could draw a scatterplot of unemployment rate vs. length of unemployment, but then we could no longer see the evolution over time. The solution is to join points adjacent in time with line segments, forming a *path* plot.

Below we plot unemployment rate vs. length of unemployment and join the individual observations with a path. Because of the many line crossings, the direction in which time flows isn't easy to see in the first plot. In the second plot, we colour the points to make it easier to see the direction of time.

```
ggplot(economics, aes(unemploy / pop, uempmed)) +
  geom_path() +
  geom_point()

year <- function(x) as.POSIXlt(x)$year + 1900
ggplot(economics, aes(unemploy / pop, uempmed)) +
  geom_path(colour = "grey50") +
  geom_point(aes(colour = year(date)))
```



We can see that unemployment rate and length of unemployment are highly correlated, but in recent years the length of unemployment has been increasing relative to the unemployment rate.

With longitudinal data, you often want to display multiple time series on each plot, each series representing one individual. To do this you need to map the `group` aesthetic to a variable encoding the group membership of each observation. This is explained in more depth in Section 3.5.

2.6.6 Exercises

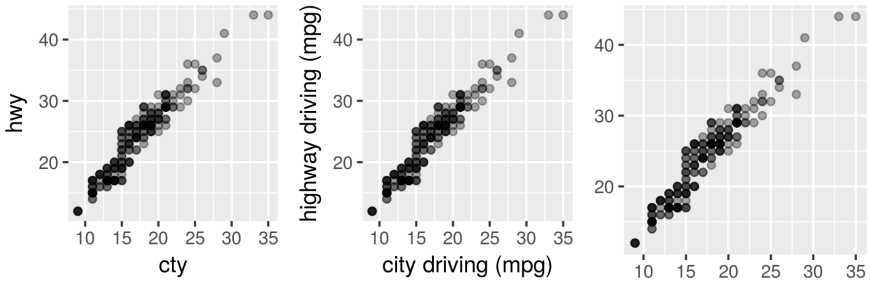
1. What's the problem with the plot created by `ggplot(mpg, aes(cty, hwy)) + geom_point()`? Which of the geoms described above is most effective at remedying the problem?
2. One challenge with `ggplot(mpg, aes(class, hwy)) + geom_boxplot()` is that the ordering of `class` is alphabetical, which is not terribly useful. How could you change the factor levels to be more informative?
   Rather than reordering the factor by hand, you can do it automatically based on the data: `ggplot(mpg, aes(reorder(class, hwy), hwy)) + geom_boxplot()`. What does `reorder()` do? Read the documentation.
3. Explore the distribution of the carat variable in the `diamonds` dataset. What binwidth reveals the most interesting patterns?
4. Explore the distribution of the price variable in the `diamonds` data. How does the distribution vary by cut?
5. You now know (at least) three ways to compare the distributions of subgroups: `geom_violin()`, `geom_freqpoly()` and the colour aesthetic, or `geom_histogram()` and facetting. What are the strengths and weaknesses of each approach? What other approaches could you try?
6. Read the documentation for `geom_bar()`. What does the `weight` aesthetic do?
7. Using the techniques already discussed in this chapter, come up with three ways to visualise a 2d categorical distribution. Try them out by visualising the distribution of `model` and `manufacturer`, `trans` and `class`, and `cyl` and `trans`.

2.7 Modifying the axes

You'll learn the full range of options available in Chapter 6, but two families of useful helpers let you make the most common modifications. `xlab()` and `ylab()` modify the x- and y-axis labels:

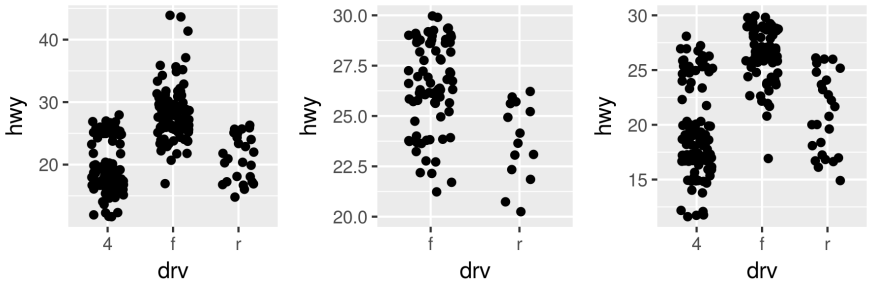
```
ggplot(mpg, aes(cty, hwy)) +
  geom_point(alpha = 1 / 3)
```

```
ggplot(mpg, aes(cty, hwy)) +
  geom_point(alpha = 1 / 3) +
  xlab("city driving (mpg)") +
  ylab("highway driving (mpg)")

# Remove the axis labels with NULL
ggplot(mpg, aes(cty, hwy)) +
  geom_point(alpha = 1 / 3) +
  xlab(NULL) +
  ylab(NULL)
```



`xlim()` and `ylim()` modify the limits of axes:

```
ggplot(mpg, aes(drv, hwy)) +
  geom_jitter(width = 0.25)

ggplot(mpg, aes(drv, hwy)) +
  geom_jitter(width = 0.25) +
  xlim("f", "r") +
  ylim(20, 30)
#> Warning: Removed 139 rows containing missing values (geom_point).

# For continuous scales, use NA to set only one limit
ggplot(mpg, aes(drv, hwy)) +
  geom_jitter(width = 0.25, na.rm = TRUE) +
  ylim(NA, 30)
```



Changing the axes limits sets values outside the range to `NA`. You can suppress the associated warning with `na.rm = TRUE`.

2.8 Output

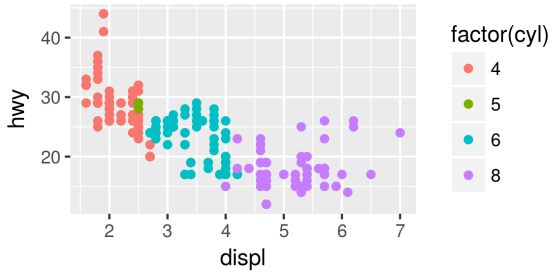
Most of the time you create a plot object and immediately plot it, but you can also save a plot to a variable and manipulate it:

```
p <- ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_point()
```

Once you have a plot object, there are a few things you can do with it:

- Render it on screen with `print()`. This happens automatically when running interactively, but inside a loop or function, you'll need to `print()` it yourself.

  ```
  print(p)
  ```



- Save it to disk with `ggsave()`, described in Section 8.5.

```
# Save png to disk
ggsave("plot.png", width = 5, height = 5)
```

- Briefly describe its structure with `summary()`.

```
summary(p)
#> data: manufacturer, model, displ, year, cyl, trans, drv,
#>   cty, hwy, fl, class [234x11]
#> mapping:  x = displ, y = hwy, colour = factor(cyl)
#> faceting: <ggproto object: Class FacetNull, Facet>
#>     compute_layout: function
#>     draw_back: function
#>     draw_front: function
#>     draw_labels: function
#>     draw_panels: function
#>     finish_data: function
#>     init_scales: function
#>     map: function
#>     map_data: function
#>     params: list
#>     render_back: function
#>     render_front: function
#>     render_panels: function
#>     setup_data: function
#>     setup_params: function
#>     shrink: TRUE
#>     train: function
#>     train_positions: function
#>     train_scales: function
#>     vars: function
#>     super:  <ggproto object: Class FacetNull, Facet>
#> -----------------------------------
#> geom_point: na.rm = FALSE
#> stat_identity: na.rm = FALSE
#> position_identity
```

- Save a cached copy of it to disk, with `saveRDS()`. This saves a complete copy of the plot object, so you can easily re-create it with `readRDS()`.

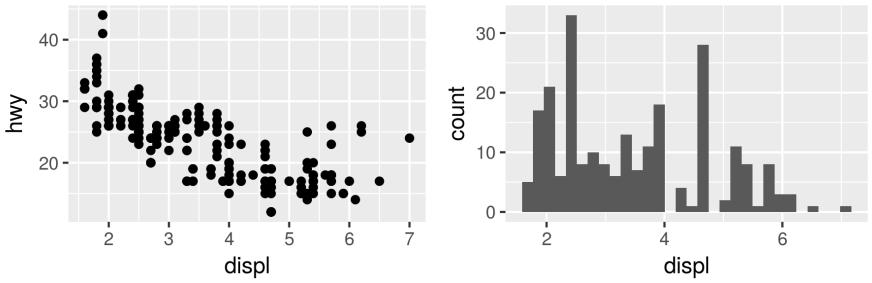
```
saveRDS(p, "plot.rds")
q <- readRDS("plot.rds")
```

You'll learn more about how to manipulate these objects in Chapter 12.

2.9 Quick plots

In some cases, you will want to create a quick plot with a minimum of typing. In these cases you may prefer to use `qplot()` over `ggplot()`. `qplot()` lets you define a plot in a single call, picking a geom by default if you don't supply one. To use it, provide a set of aesthetics and a data set:

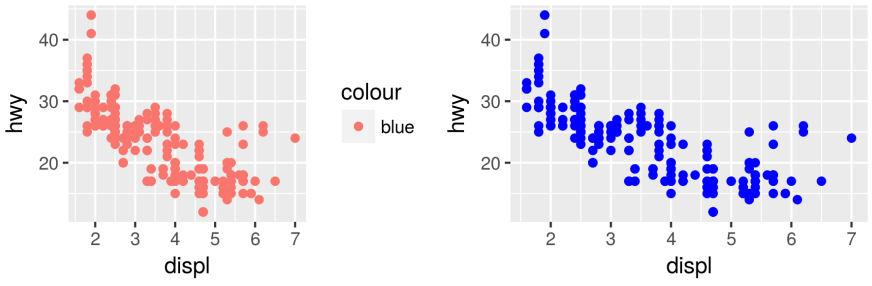
```
qplot(displ, hwy, data = mpg)
qplot(displ, data = mpg)
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
```



Unless otherwise specified, `qplot()` tries to pick a sensible geometry and statistic based on the arguments provided. For example, if you give `qplot()` `x` and `y` variables, it'll create a scatterplot. If you just give it an `x`, it'll create a histogram or bar chart depending on the type of variable.

`qplot()` assumes that all variables should be scaled by default. If you want to set an aesthetic to a constant, you need to use `I()`:

```
qplot(displ, hwy, data = mpg, colour = "blue")
qplot(displ, hwy, data = mpg, colour = I("blue"))
```



If you're used to `plot()` you may find `qplot()` to be a useful crutch to get up and running quickly. However, while it's possible to use `qplot()` to access all of the customizability of ggplot2, I don't recommend it. If you find yourself making a more complex graph, e.g. using different aesthetics in different layers or manually setting visual properties, use `ggplot()`, not `qplot()`.

Chapter 3
Toolbox

3.1 Introduction

The layered structure of ggplot2 encourages you to design and construct graphics in a structured manner. You've learned the basics in the previous chapter, and in this chapter you'll get a more comprehensive task-based introduction. The goal here is not to exhaustively explore every option of every geom, but instead to show the most important tools for a given task. For more information about individual geoms, along with many more examples illustrating their use, see the documentation.

It is useful to think about the purpose of each layer before it is added. In general, there are three purposes for a layer:

- To display the **data**. We plot the raw data for many reasons, relying on our skills at pattern detection to spot gross structure, local structure, and outliers. This layer appears on virtually every graphic. In the earliest stages of data exploration, it is often the only layer.
- To display a statistical **summary** of the data. As we develop and explore models of the data, it is useful to display model predictions in the context of the data. Showing the data helps us improve the model, and showing the model helps reveal subtleties of the data that we might otherwise miss. Summaries are usually drawn on top of the data.
- To add additional **metadata**: context, annotations, and references. A metadata layer displays background context, annotations that help to give meaning to the raw data, or fixed references that aid comparisons across panels. Metadata can be useful in the background and foreground.
  A map is often used as a background layer with spatial data. Background metadata should be rendered so that it doesn't interfere with your perception of the data, so is usually displayed underneath the data and formatted so that it is minimally perceptible. That is, if you concentrate on it, you can see it with ease, but it doesn't jump out at you when you are casually browsing the plot.

Other metadata is used to highlight important features of the data. If you have added explanatory labels to a couple of inflection points or outliers, then you want to render them so that they pop out at the viewer. In that case, you want this to be the very last layer drawn.

This chapter is broken up into the following sections, each of which deals with a particular graphical challenge. This is not an exhaustive or exclusive categorisation, and there are many other possible ways to break up graphics into different categories. Each geom can be used for many different purposes, especially if you are creative. However, this breakdown should cover many common tasks and help you learn about some of the possibilities.

- Basic plot types that produce common, 'named' graphics like scatterplots and line charts, Section 3.2.
- Displaying text, Section 3.3.
- Adding arbitrary additional anotations, Section 3.4.
- Working with collective geoms, like lines and polygons, that each display multiple rows of data, Section 3.5.
- Surface plots to display 3d surfaces in 2d, Section 3.6.
- Drawing maps, Section 3.7.
- Revealing uncertainty and error, with various 1d and 2d intervals, Section 3.8.
- Weighted data, Section 3.9.

In Section 3.10, you'll learn about the diamonds dataset. The final three sections use this data to discuss techniques for visualising larger datasets:

- Displaying distributions, continuous and discrete, 1d and 2d, joint and conditional, Section 3.11.
- Dealing with overplotting in scatterplots, a challenge with large datasets, Section 3.12.
- Displaying statistical summaries instead of the raw data, Section 3.13.

The chapter concludes in Section 3.14 with some pointers to other useful packages built on top of ggplot2.

3.2 Basic plot types

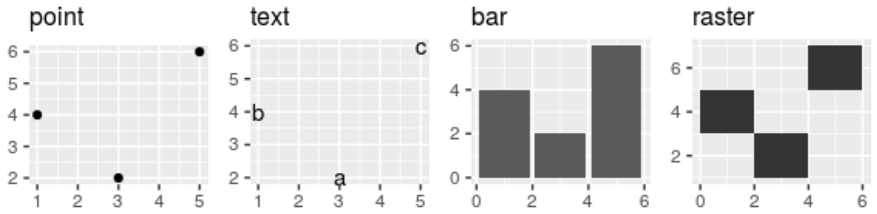
These geoms are the fundamental building blocks of ggplot2. They are useful in their own right, but are also used to construct more complex geoms. Most of these geoms are associated with a named plot: when that geom is used by itself in a plot, that plot has a special name.

Each of these geoms is two dimensional and requires both `x` and `y` aesthetics. All of them understand `colour` (or `color`) and `size` aesthetics, and the filled geoms (bar, tile and polygon) also understand `fill`.

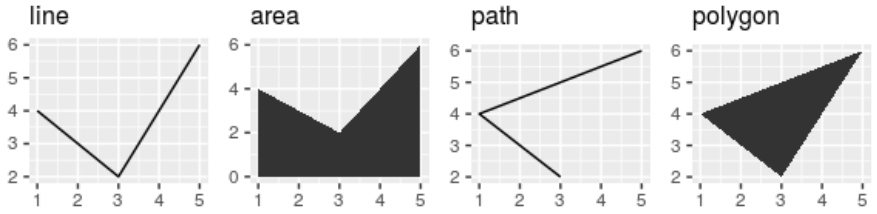
- `geom_area()` draws an **area plot**, which is a line plot filled to the y-axis (filled lines). Multiple groups will be stacked on top of each other.
- `geom_bar(stat = "identity")` makes a **bar chart**. We need `stat = "identity"` because the default stat automatically counts values (so is essentially a 1d geom, see Section 3.11. The identity stat leaves the data unchanged. Multiple bars in the same location will be stacked on top of one another.
- `geom_line()` makes a **line plot**. The `group` aesthetic determines which observations are connected; see Section 3.5 for more detail. `geom_line()` connects points from left to right; `geom_path()` is similar but connects points in the order they appear in the data. Both `geom_line()` and `geom_path()` also understand the aesthetic `linetype`, which maps a categorical variable to solid, dotted and dashed lines.
- `geom_point()` produces a **scatterplot**. `geom_point()` also understands the `shape` aesthetic.
- `geom_polygon()` draws polygons, which are filled paths. Each vertex of the polygon requires a separate row in the data. It is often useful to merge a data frame of polygon coordinates with the data just prior to plotting. Section 3.7 illustrates this concept in more detail for map data.
- `geom_rect()`, `geom_tile()` and `geom_raster()` draw rectangles. `geom_rect()` is parameterised by the four corners of the rectangle, `xmin`, `ymin`, `xmax` and `ymax`. `geom_tile()` is exactly the same, but parameterised by the center of the rect and its size, `x`, `y`, `width` and `height`. `geom_raster()` is a fast special case of `geom_tile()` used when all the tiles are the same size. .

Each geom is shown in the code below. Observe the different axis ranges for the bar, area and tile plots: these geoms take up space outside the range of the data, and so push the axes out.

```
df <- data.frame(
  x = c(3, 1, 5),
  y = c(2, 4, 6),
  label = c("a","b","c")
)
p <- ggplot(df, aes(x, y, label = label)) +
  labs(x = NULL, y = NULL) + # Hide axis label
  theme(plot.title = element_text(size = 12)) # Shrink plot title
p + geom_point() + ggtitle("point")
p + geom_text() + ggtitle("text")
p + geom_bar(stat = "identity") + ggtitle("bar")
p + geom_tile() + ggtitle("raster")
```




```
p + geom_line() + ggtitle("line")
p + geom_area() + ggtitle("area")
p + geom_path() + ggtitle("path")
p + geom_polygon() + ggtitle("polygon")
```



3.2.1 Exercises

1. What geoms would you use to draw each of the following named plots?

    1. Scatterplot
    2. Line chart
    3. Histogram
    4. Bar chart
    5. Pie chart

2. What's the difference between `geom_path()` and `geom_polygon()`? What's the difference between `geom_path()` and `geom_line()`?
3. What low-level geoms are used to draw `geom_smooth()`? What about `geom_boxplot()` and `geom_violin()`?

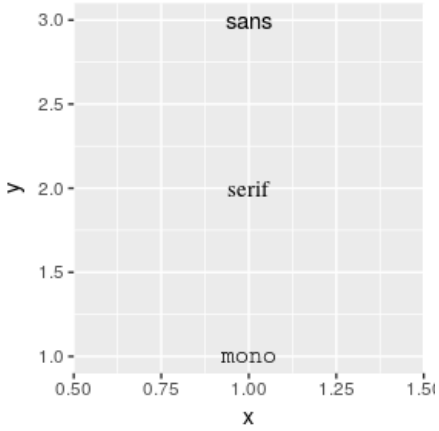
3.3 Labels

Adding text to a plot can be quite tricky. ggplot2 doesn't have all the answers, but does provide some tools to make your life a little easier. The main tool is `geom_text()`, which adds `labels` at the specified `x` and `y` positions.

`geom_text()` has the most aesthetics of any geom, because there are so many ways to control the appearance of a text:

- `family` gives the name of a font. There are only three fonts that are guaranteed to work everywhere: "sans" (the default), "serif", or "mono":

```
df <- data.frame(x = 1, y = 3:1, family = c("sans", "serif", "mono"))
ggplot(df, aes(x, y)) +
  geom_text(aes(label = family, family = family))
```

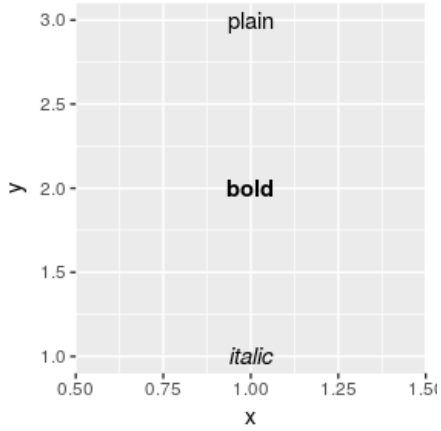


  It's trickier to include a system font on a plot because text drawing is done differently by each graphics device (GD). There are five GDs in common use (`png()`, `pdf()`, on screen devices for Windows, Mac and Linux), so to have a font work everywhere you need to configure five devices in five different ways. Two packages simplify the quandary a bit:

  - showtext, `https://github.com/yixuan/showtext`, by Yixuan Qiu, makes GD-independent plots by rendering all text as polygons.
  - extrafont, `https://github.com/wch/extrafont`, by Winston Chang, converts fonts to a standard format that all devices can use.

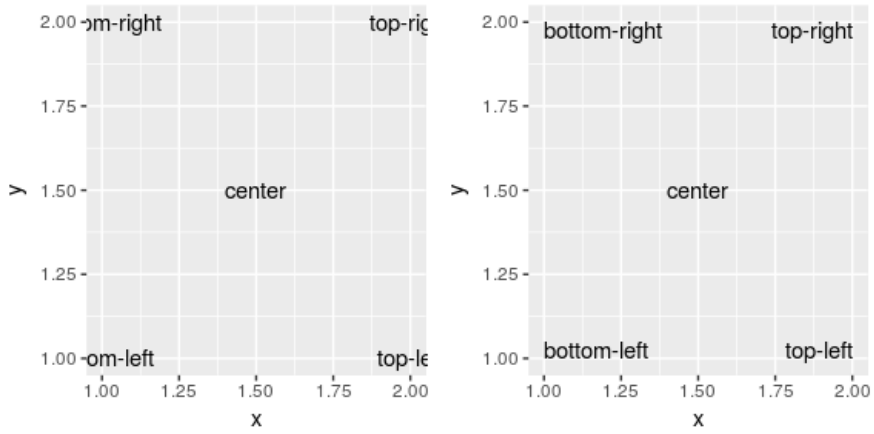
  Both approaches have pros and cons, so you will to need to try both of them and see which works best for your needs.
- `fontface` specifies the face: "plain" (the default), "bold" or "italic".

```
df <- data.frame(x = 1, y = 3:1, face = c("plain", "bold", "italic"))
ggplot(df, aes(x, y)) +
  geom_text(aes(label = face, fontface = face))
```



- You can adjust the alignment of the text with the `hjust` ("left", "center", "right", "inward", "outward") and `vjust` ("bottom", "middle", "top", "inward", "outward") aesthetics. The default alignment is centered. One of the most useful alignments is "inward": it aligns text towards the middle of the plot:

```
df <- data.frame(
  x = c(1, 1, 2, 2, 1.5),
  y = c(1, 2, 1, 2, 1.5),
  text = c(
    "bottom-left", "bottom-right",
    "top-left", "top-right", "center"
  )
)
ggplot(df, aes(x, y)) +
  geom_text(aes(label = text))
ggplot(df, aes(x, y)) +
  geom_text(aes(label = text), vjust = "inward", hjust = "inward")
```

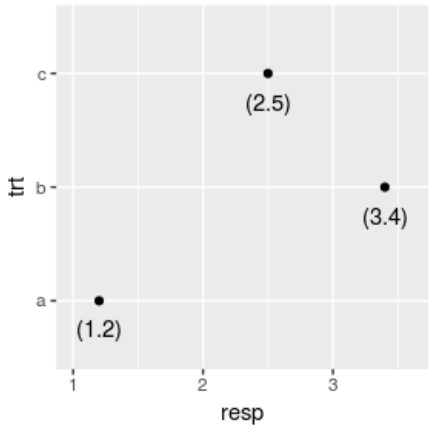


- `size` controls the font size. Unlike most tools, ggplot2 uses mm, rather than the usual points (pts). This makes it consistent with other size units in ggplot2. (There are 72.27 pts in a inch, so to convert from points to mm, just multiply by 72.27 / 25.4).
- `angle` specifies the rotation of the text in degrees.

You can map data values to these aesthetics, but use restraint: it is hard to percieve the relationship between variables mapped to these aesthetics. `geom_text()` also has three parameters. Unlike the aesthetics, these only take single values, so they must be the same for all labels:

- Often you want to label existing points on the plot. You don't want the text to overlap with the points (or bars etc), so it's useful to offset the text a little. The `nudge_x` and `nudge_y` parameters allow you to nudge the text a little horizontally or vertically:

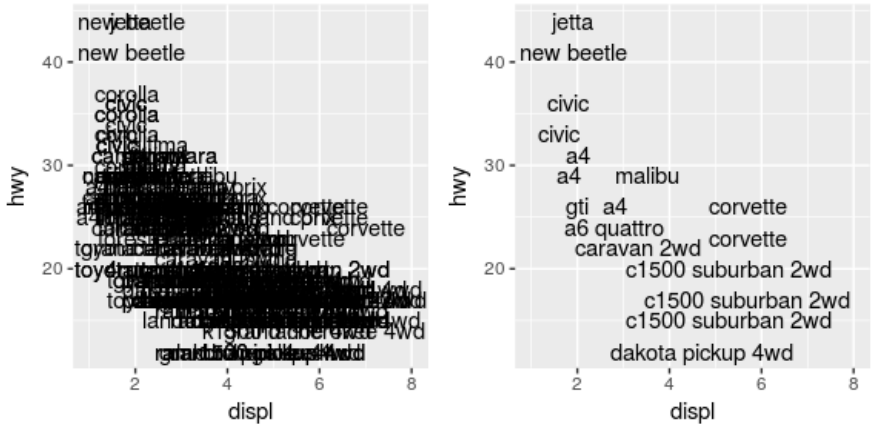
```
df <- data.frame(trt = c("a", "b", "c"), resp = c(1.2, 3.4, 2.5))
ggplot(df, aes(resp, trt)) +
  geom_point() +
  geom_text(aes(label = paste0("(", resp, ")")), nudge_y = -0.25) +
  xlim(1, 3.6)
```



(Note that I manually tweaked the x-axis limits to make sure all the text fit on the plot.)

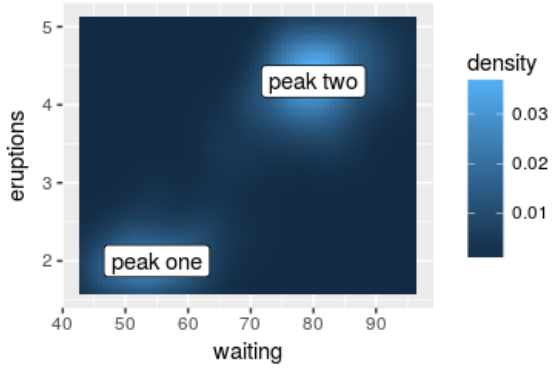
- If `check_overlap = TRUE`, overlapping labels will be automatically removed. The algorithm is simple: labels are plotted in the order they appear in the data frame; if a label would overlap with an existing point, it's omitted. This is not incredibly useful, but can be handy.

```
ggplot(mpg, aes(displ, hwy)) +
  geom_text(aes(label = model)) +
  xlim(1, 8)
ggplot(mpg, aes(displ, hwy)) +
  geom_text(aes(label = model), check_overlap = TRUE) +
  xlim(1, 8)
```



A variation on `geom_text()` is `geom_label()`: it draws a rounded rectangle behind the text. This makes it useful for adding labels to plots with busy backgrounds:

```
label <- data.frame(
  waiting = c(55, 80),
  eruptions = c(2, 4.3),
  label = c("peak one", "peak two")
)

ggplot(faithfuld, aes(waiting, eruptions)) +
  geom_tile(aes(fill = density)) +
  geom_label(data = label, aes(label = label))
```



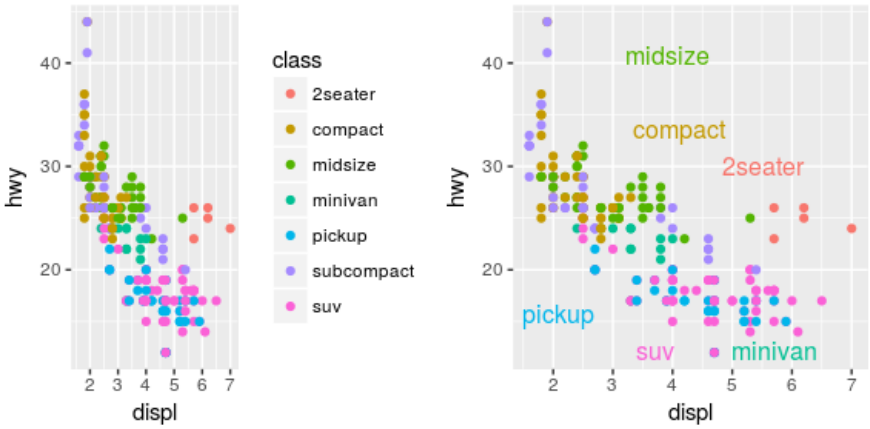
Labelling data well poses some challenges:

- Text does not affect the limits of the plot. Unfortunately there's no way to make this work since a label has an absolute size (e.g. 3 cm), regardless of the size of the plot. This means that the limits of a plot would need to be different depending on the size of the plot — there's just no way to make that happen with ggplot2. Instead, you'll need to tweak `xlim()` and `ylim()` based on your data and plot size.
- If you want to label many points, it is difficult to avoid overlaps. `check_overlap = TRUE` is useful, but offers little control over which labels are removed. There are a number of techniques available for base graphics, like `maptools::pointLabel()`, but they're not trivial to port to the grid graphics used by ggplot2. If all else fails, you may need to manually label points in a drawing tool.

Text labels can also serve as an alternative to a legend. This usually makes the plot easier to read because it puts the labels closer to the data. The di-

rectlabels (`https://github.com/tdhock/directlabels`) package, by Toby Dylan Hocking, provides a number of tools to make this easier:

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

ggplot(mpg, aes(displ, hwy, colour = class)) +
  geom_point(show.legend = FALSE) +
  directlabels::geom_dl(aes(label = class), method = "smart.grid")
```



Directlabels provides a number of position methods. `smart.grid` is a reasonable place to start for scatterplots, but there are other methods that are more useful for frequency polygons and line plots. See the directlabels website, `http://directlabels.r-forge.r-project.org`, for other techniques.

3.4 Annotations

Annotations add metadata to your plot. But metadata is just data, so you can use:

- `geom_text()` to add text descriptions or to label points Most plots will not benefit from adding text to every single observation on the plot, but labelling outliers and other important points is very useful.
- `geom_rect()` to highlight interesting rectangular regions of the plot. `geom_rect()` has aesthetics `xmin`, `xmax`, `ymin` and `ymax`.
- `geom_line()`, `geom_path()` and `geom_segment()` to add lines. All these geoms have an `arrow` parameter, which allows you to place an arrowhead on the

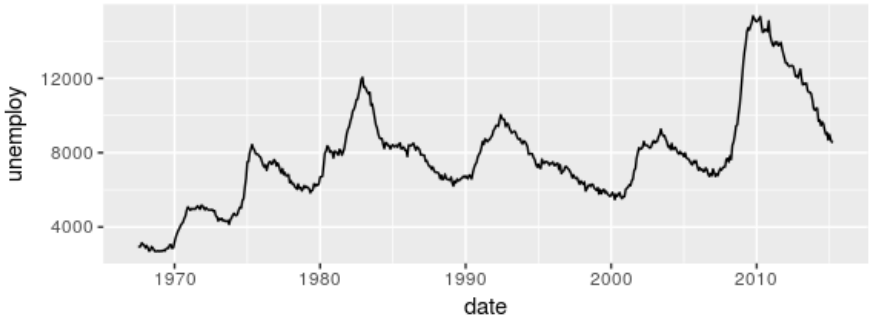
line. Create arrowheads with `arrow()`, which has arguments `angle`, `length`, `ends` and `type`.

- `geom_vline()`, `geom_hline()` and `geom_abline()` allow you to add reference lines (sometimes called rules), that span the full range of the plot.

Typically, you can either put annotations in the foreground (using `alpha` if needed so you can still see the data), or in the background. With the default background, a thick white line makes a useful reference: it's easy to see but it doesn't jump out at you.

To show off the basic idea, we'll draw a time series of unemployment:

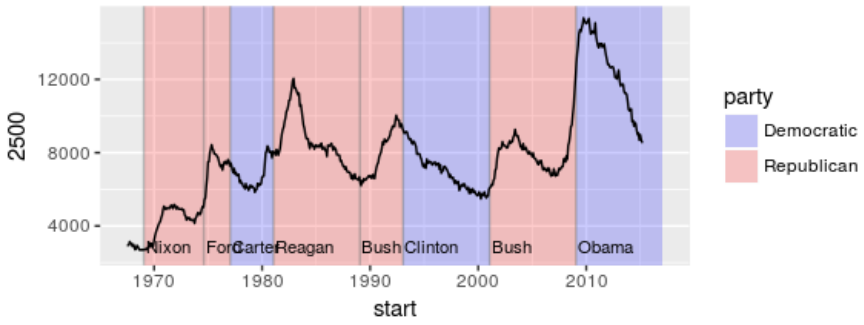
```
ggplot(economics, aes(date, unemploy)) +
  geom_line()
```



We can annotate this plot with which president was in power at the time. There is little new in this code - it's a straightforward manipulation of existing geoms. There is one special thing to note: the use of `-Inf` and `Inf` as positions. These refer to the top and bottom (or left and right) limits of the plot.

```
presidential <- subset(presidential, start > economics$date[1])

ggplot(economics) +
  geom_rect(
    aes(xmin = start, xmax = end, fill = party),
    ymin = -Inf, ymax = Inf, alpha = 0.2,
    data = presidential
  ) +
  geom_vline(
    aes(xintercept = as.numeric(start)),
    data = presidential,
    colour = "grey50", alpha = 0.5
  ) +
  geom_text(
```

```
    aes(x = start, y = 2500, label = name),
    data = presidential,
    size = 3, vjust = 0, hjust = 0, nudge_x = 50
  ) +
  geom_line(aes(date, unemploy)) +
  scale_fill_manual(values = c("blue", "red"))
```

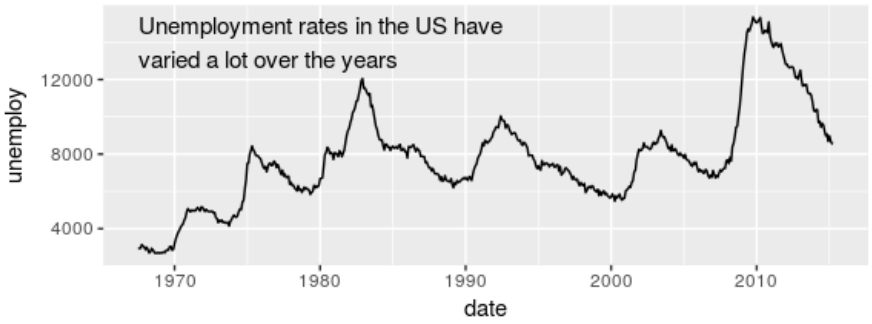


You can use the same technique to add a single annotation to a plot, but it's a bit fiddly because you have to create a one row data frame:

```
yrng <- range(economics$unemploy)
xrng <- range(economics$date)
caption <- paste(strwrap("Unemployment rates in the US have
  varied a lot over the years", 40), collapse = "\n")

ggplot(economics, aes(date, unemploy)) +
  geom_line() +
  geom_text(
    aes(x, y, label = caption),
    data = data.frame(x = xrng[1], y = yrng[2], caption = caption),
    hjust = 0, vjust = 1, size = 4
  )
```

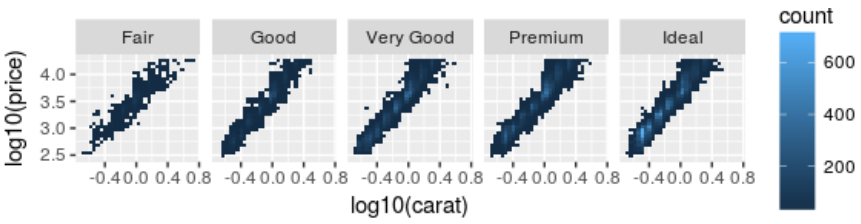
It's easier to use the `annotate()` helper function which creates the data frame for you:

```
ggplot(economics, aes(date, unemploy)) +
  geom_line() +
  annotate("text", x = xrng[1], y = yrng[2], label = caption,
    hjust = 0, vjust = 1, size = 4
  )
```



Annotations, particularly reference lines, are also useful when comparing groups across facets. In the following plot, it's much easier to see the subtle differences if we add a reference line.

```
ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_bin2d() +
  facet_wrap(~cut, nrow = 1)
```




```
mod_coef <- coef(lm(log10(price) ~ log10(carat), data = diamonds))
ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_bin2d() +
  geom_abline(intercept = mod_coef[1], slope = mod_coef[2],
    colour = "white", size = 1) +
  facet_wrap(~cut, nrow = 1)
```

3.5 Collective geoms

Geoms can be roughly divided into individual and collective geoms. An **individual** geom draws a distinct graphical object for each observation (row). For example, the point geom draws one point per row. A **collective** geom displays multiple observations with one geometric object. This may be a result of a statistical summary, like a boxplot, or may be fundamental to the display of the geom, like a polygon. Lines and paths fall somewhere in between: each line is composed of a set of straight segments, but each segment represents two points. How do we control the assignment of observations to graphical elements? This is the job of the `group` aesthetic.

By default, the `group` aesthetic is mapped to the interaction of all discrete variables in the plot. This often partitions the data correctly, but when it does not, or when no discrete variable is used in a plot, you'll need to explicitly define the grouping structure by mapping group to a variable that has a different value for each group.

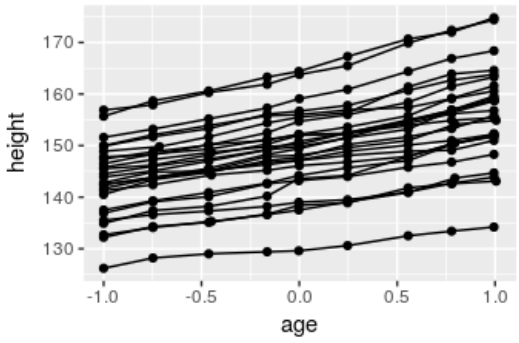
There are three common cases where the default is not enough, and we will consider each one below. In the following examples, we will use a simple longitudinal dataset, `Oxboys`, from the nlme package. It records the heights (`height`) and centered ages (`age`) of 26 boys (`Subject`), measured on nine occasions (`Occasion`). `Subject` and `Occassion` are stored as ordered factors.

```
data(Oxboys, package = "nlme")
head(Oxboys)
#>   Subject     age height Occasion
#> 1       1 -1.0000    140        1
#> 2       1 -0.7479    143        2
#> 3       1 -0.4630    145        3
#> 4       1 -0.1643    147        4
#> 5       1 -0.0027    148        5
#> 6       1  0.2466    150        6
```

3.5.1 Multiple groups, one aesthetic

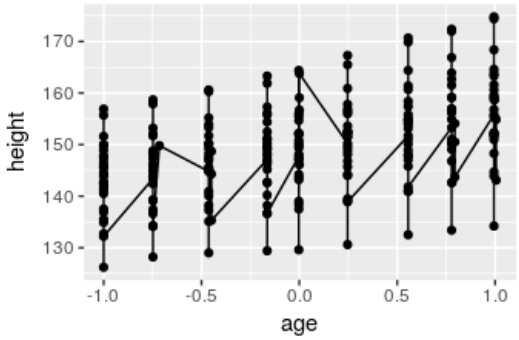
In many situations, you want to separate your data into groups, but render them in the same way. In other words, you want to be able to distinguish individual subjects, but not identify them. This is common in longitudinal studies with many subjects, where the plots are often descriptively called spaghetti plots. For example, the following plot shows the growth trajectory for each boy (each `Subject`):

```
ggplot(Oxboys, aes(age, height, group = Subject)) +
  geom_point() +
  geom_line()
```



If you incorrectly specify the grouping variable, you'll get a characteristic sawtooth appearance:

```
ggplot(Oxboys, aes(age, height)) +
  geom_point() +
  geom_line()
```



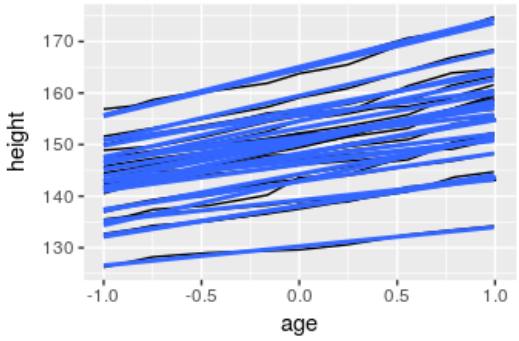
If a group isn't defined by a single variable, but instead by a combination of multiple variables, use `interaction()` to combine them, e.g. `aes(group = interaction(school_id, student_id))`.

3.5.2 Different groups on different layers

Sometimes we want to plot summaries that use different levels of aggregation: one layer might display individuals, while another displays an overall summary. Building on the previous example, suppose we want to add a sin-

gle smooth line, showing the overall trend for *all* boys. If we use the same grouping in both layers, we get one smooth per boy:

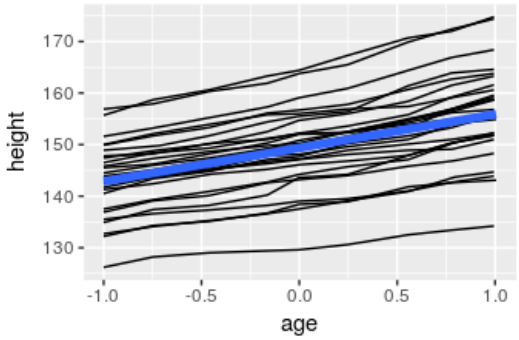
```
ggplot(Oxboys, aes(age, height, group = Subject)) +
  geom_line() +
  geom_smooth(method = "lm", se = FALSE)
```



This is not what we wanted; we have inadvertently added a smoothed line for each boy. Grouping controls both the display of the geoms, and the operation of the stats: one statistical transformation is run for each group.

Instead of setting the grouping aesthetic in `ggplot()`, where it will apply to all layers, we set it in `geom_line()` so it applies only to the lines. There are no discrete variables in the plot so the default grouping variable will be a constant and we get one smooth:

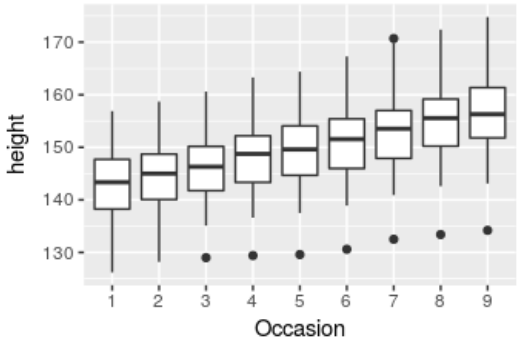
```
ggplot(Oxboys, aes(age, height)) +
  geom_line(aes(group = Subject)) +
  geom_smooth(method = "lm", size = 2, se = FALSE)
```



3.5.3 Overriding the default grouping

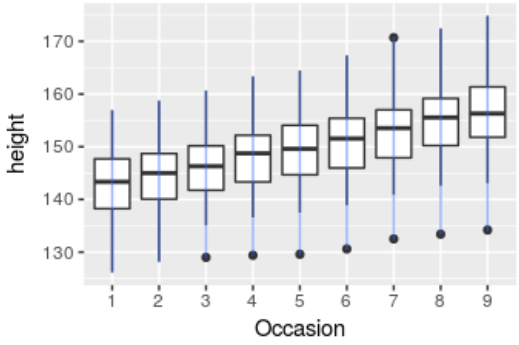
Some plots have a discrete x scale, but you still want to draw lines connecting *across* groups. This is the strategy used in interaction plots, profile plots, and parallel coordinate plots, among others. For example, imagine we've drawn boxplots of height at each measurement occasion:

```
ggplot(Oxboys, aes(Occasion, height)) +
  geom_boxplot()
```



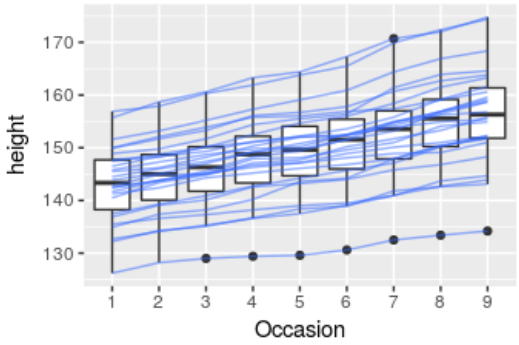
There is one discrete variable in this plot, `Occassion`, so we get one boxplot for each unique x value. Now we want to overlay lines that connect each individual boy. Simply adding `geom_line()` does not work: the lines are drawn within each occassion, not across each subject:

```
ggplot(Oxboys, aes(Occasion, height)) +
  geom_boxplot() +
  geom_line(colour = "#3366FF", alpha = 0.5)
```



To get the plot we want, we need to override the grouping to say we want one line per boy:

```
ggplot(Oxboys, aes(Occasion, height)) +
  geom_boxplot() +
  geom_line(aes(group = Subject), colour = "#3366FF", alpha = 0.5)
```

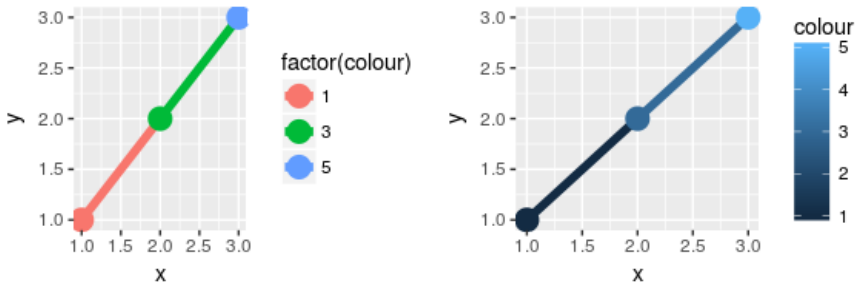


3.5.4 Matching aesthetics to graphic objects

A final important issue with collective geoms is how the aesthetics of the individual observations are mapped to the aesthetics of the complete entity. What happens when different aesthetics are mapped to a single geometric element?

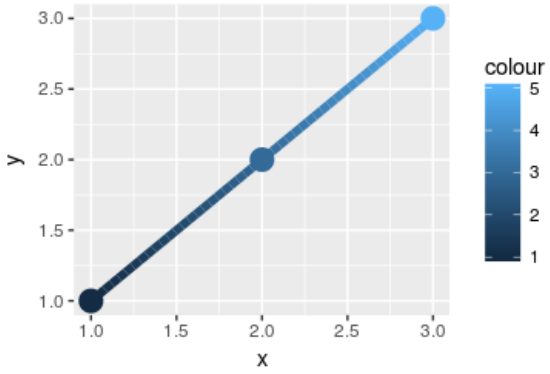
Lines and paths operate on an off-by-one principle: there is one more observation than line segment, and so the aesthetic for the first observation is used for the first segment, the second observation for the second segment and so on. This means that the aesthetic for the last observation is not used:

```
df <- data.frame(x = 1:3, y = 1:3, colour = c(1,3,5))

ggplot(df, aes(x, y, colour = factor(colour))) +
  geom_line(aes(group = 1), size = 2) +
  geom_point(size = 5)

ggplot(df, aes(x, y, colour = colour)) +
  geom_line(aes(group = 1), size = 2) +
  geom_point(size = 5)
```



You could imagine a more complicated system where segments smoothly blend from one aesthetic to another. This would work for continuous variables like size or colour, but not for discrete variables, and is not used in ggplot2. If this is the behaviour you want, you can perform the linear interpolation yourself:

```
xgrid <- with(df, seq(min(x), max(x), length = 50))
interp <- data.frame(
  x = xgrid,
  y = approx(df$x, df$y, xout = xgrid)$y,
  colour = approx(df$x, df$colour, xout = xgrid)$y
)
ggplot(interp, aes(x, y, colour = colour)) +
  geom_line(size = 2) +
  geom_point(data = df, size = 5)
```

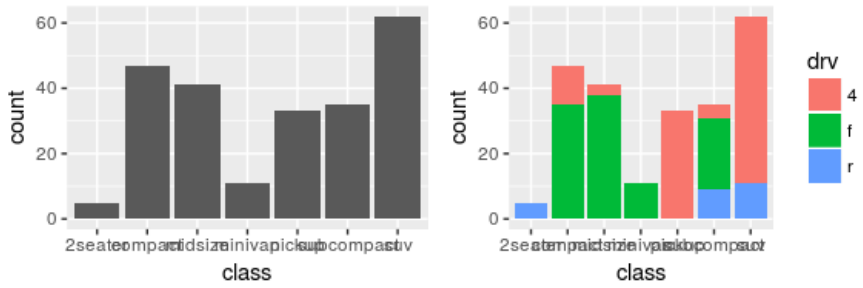


An additional limitation for paths and lines is that line type must be constant over each individual line. In R there is no way to draw a line which has varying line type.

For all other collective geoms, like polygons, the aesthetics from the individual components are only used if they are all the same, otherwise the default value is used. It's particularly clear why this makes sense for fill: how would you colour a polygon that had a different fill colour for each point on its border?

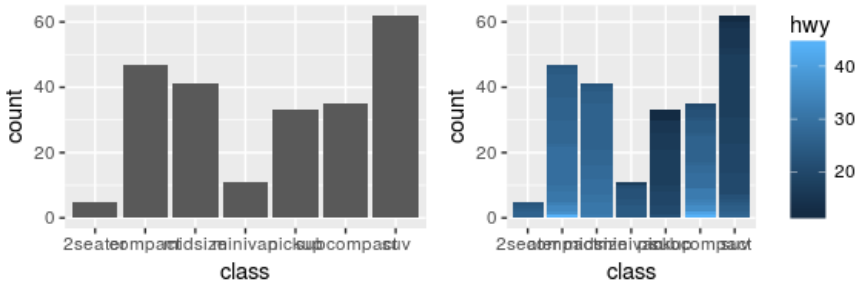
These issues are most relevant when mapping aesthetics to continuous variables, because, as described above, when you introduce a mapping to a discrete variable, it will by default split apart collective geoms into smaller pieces. This works particularly well for bar and area plots, because stacking the individual pieces produces the same shape as the original ungrouped data:

```
ggplot(mpg, aes(class)) +
  geom_bar()
ggplot(mpg, aes(class, fill = drv)) +
  geom_bar()
```



If you try to map fill to a continuous variable in the same way, it doesn't work. The default grouping will only be based on `class`, so each bar will be given multiple colours. Since a bar can only display one colour, it will use the default grey. To show multiple colours, we need multiple bars for each `class`, which we can get by overriding the grouping:

```
ggplot(mpg, aes(class, fill = hwy)) +
  geom_bar()
ggplot(mpg, aes(class, fill = hwy, group = hwy)) +
  geom_bar()
```



The bars will be stacked in the order defined by the grouping variable. If you need fine control, you'll need to create a factor with levels ordered as needed.

3.5.5 Exercises

1. Draw a boxplot of `hwy` for each value of `cyl`, without turning `cyl` into a factor. What extra aesthetic do you need to set?
2. Modify the following plot so that you get one boxplot per integer value value of `displ`.

```
ggplot(mpg, aes(displ, cty)) +
  geom_boxplot()
```

3. When illustrating the difference between mapping continuous and discrete colours to a line, the discrete example needed `aes(group = 1)`. Why? What happens if that is omitted? What's the difference between `aes(group = 1)` and `aes(group = 2)`? Why?
4. How many bars are in each of the following plots?

```
ggplot(mpg, aes(drv)) +
  geom_bar()

ggplot(mpg, aes(drv, fill = hwy, group = hwy)) +
  geom_bar()

library(dplyr)
mpg2 <- mpg %>% arrange(hwy) %>% mutate(id = seq_along(hwy))
ggplot(mpg2, aes(drv, fill = hwy, group = id)) +
  geom_bar()
```

(Hint: try adding an outline around each bar with `colour = "white"`)

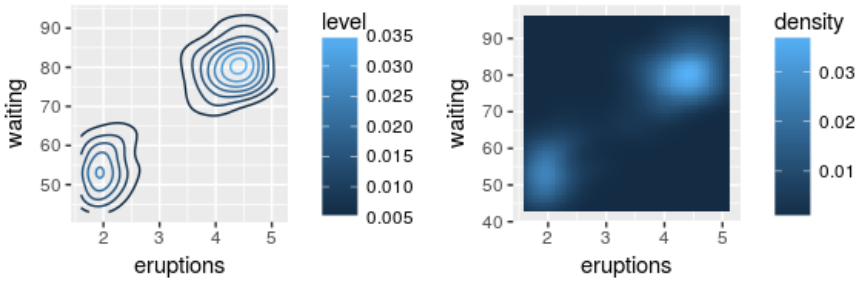
5. Install the babynames package. It contains data about the popularity of babynames in the US. Run the following code and fix the resulting graph. Why does this graph make me unhappy?

```
library(babynames)
hadley <- dplyr::filter(babynames, name == "Hadley")
ggplot(hadley, aes(year, n)) +
  geom_line()
```

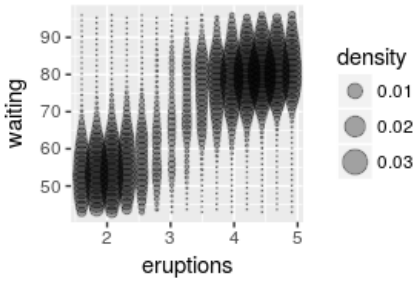
3.6 Surface plots

ggplot2 does not support true 3d surfaces. However, it does support many common tools for representing 3d surfaces in 2d: contours, coloured tiles and bubble plots. These all work similarly, differing only in the aesthetic used for the third dimension.

```
ggplot(faithfuld, aes(eruptions, waiting)) +
  geom_contour(aes(z = density, colour = ..level..))

ggplot(faithfuld, aes(eruptions, waiting)) +
  geom_raster(aes(fill = density))
```




```
# Bubble plots work better with fewer observations
small <- faithfuld[seq(1, nrow(faithfuld), by = 10), ]
ggplot(small, aes(eruptions, waiting)) +
  geom_point(aes(size = density), alpha = 1/3) +
  scale_size_area()
```



For interactive 3d plots, including true 3d surfaces, see RGL, `http://rgl.neoscientists.org/about.shtml`.

3.7 Drawing maps

There are four types of map data you might want to visualise: vector boundaries, point metadata, area metadata, and raster images. Typically, assembling these datasets is the most challenging part of drawing maps. Unfortunately ggplot2 can't help you with that part of the analysis, but I'll provide some hints about other R packages that you might want to look at.

I'll illustrate each of the four types of map data with some maps of Michigan.

3.7.1 Vector boundaries

Vector boundaries are defined by a data frame with one row for each "corner" of a geographical region like a country, state, or county. It requires four variables:

- `lat` and `long`, giving the location of a point.
- `group`, a unique identifier for each contiguous region.
- `id`, the name of the region.

Separate `group` and `id` variables are necessary because sometimes a geographical unit isn't a contiguous polygon. For example, Hawaii is composed of multiple islands that can't be drawn using a single polygon.

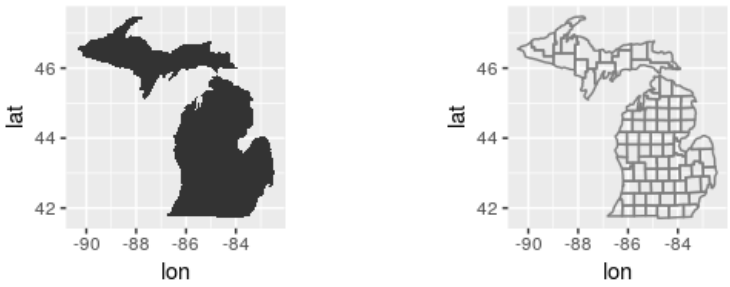
The following code extracts that data from the built in maps package using `ggplot2::map_data()`. The maps package isn't particularly accurate or up-to-date, but it's built into R so it's a reasonable place to start.

```
mi_counties <- map_data("county", "michigan") %>%
  select(lon = long, lat, group, id = subregion)
```

```
head(mi_counties)
#>     lon  lat group     id
#> 1 -83.9 44.9     1 alcona
#> 2 -83.4 44.9     1 alcona
#> 3 -83.4 44.9     1 alcona
#> 4 -83.3 44.8     1 alcona
#> 5 -83.3 44.8     1 alcona
#> 6 -83.3 44.8     1 alcona
```

You can visualise vector boundary data with `geom_polygon()`:

```
ggplot(mi_counties, aes(lon, lat)) +
  geom_polygon(aes(group = group)) +
  coord_quickmap()

ggplot(mi_counties, aes(lon, lat)) +
  geom_polygon(aes(group = group), fill = NA, colour = "grey50") +
  coord_quickmap()
```



Note the use of `coord_quickmap()`: it's a quick and dirty adjustment that ensures that the aspect ratio of the plot is set correctly.

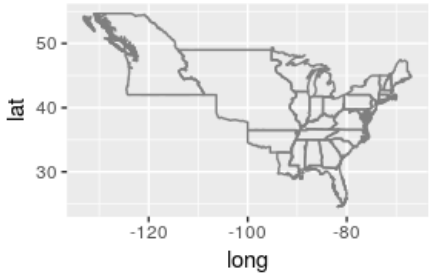
Other useful sources of vector boundary data are:

- The USAboundaries package, `https://github.com/ropensci/USAboundaries` which contains state, county and zip code data for the US. As well as current boundaries, it also has state and county boundaries going back to the 1600s.
- The tigris package, `https://github.com/walkerke/tigris`, makes it easy to access the US Census TIGRIS shapefiles. It contains state, county, zipcode, and census tract boundaries, as well as many other useful datasets.
- The rnaturalearth package bundles up the free, high-quality data from `http://naturalearthdata.com/`. It contains country borders, and borders for the top-level region within each country (e.g. states in the USA, regions in France, counties in the UK).

- The osmar package, `https://cran.r-project.org/package=osmar` wraps up the OpenStreetMap API so you can access a wide range of vector data including indvidual streets and buildings
- You may have your own shape files (`.shp`). You can load them into R with `maptools::readShapeSpatial()`.

These sources all generate spatial data frames defined by the sp package. You can convert them into a data frame with `fortify()`:

```
library(USAboundaries)
library(sf) # Necessary to do the conversion below
#> Linking to GEOS 3.5.1, GDAL 2.1.0, proj.4 4.9.2, lwgeom 2.3.2 r15302
c18 <- us_boundaries(as.Date("1820-01-01")) # Output has changed - now returns sf objects
c18 <- as(c18, "Spatial") # Convert back into sp format
c18df <- fortify(c18)
#> Regions defined for each Polygons
head(c18df)
#>    long lat order  hole piece id group
#> 1 -87.6  35     1 FALSE     1  5   5.1
#> 2 -87.6  35     2 FALSE     1  5   5.1
#> 3 -87.6  35     3 FALSE     1  5   5.1
#> 4 -87.6  35     4 FALSE     1  5   5.1
#> 5 -87.5  35     5 FALSE     1  5   5.1
#> 6 -87.3  35     6 FALSE     1  5   5.1

ggplot(c18df, aes(long, lat)) +
  geom_polygon(aes(group = group), colour = "grey50", fill = NA) +
  coord_quickmap()
```



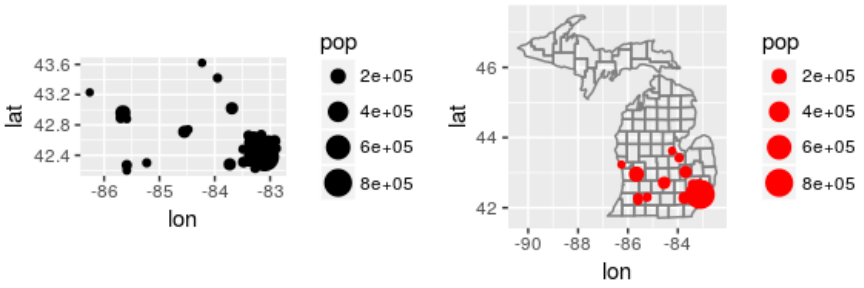
3.7.2 Point metadata

Point metadata connects locations (defined by lat and lon) with other variables. For example, the code below extracts the biggest cities in MI (as of 2006):

```
mi_cities <- maps::us.cities %>%
  tbl_df() %>%
  filter(country.etc == "MI") %>%
  select(-country.etc, lon = long) %>%
  arrange(desc(pop))
mi_cities
#> # A tibble: 36 x 5
#>                   name    pop   lat   lon capital
#>                  <chr>  <int> <dbl> <dbl>   <int>
#>  1          Detroit MI 871789  42.4 -83.1       0
#>  2     Grand Rapids MI 193006  43.0 -85.7       0
#>  3           Warren MI 132537  42.5 -83.0       0
#>  4 Sterling Heights MI 127027  42.6 -83.0       0
#>  5          Lansing MI 117236  42.7 -84.5       2
#>  6            Flint MI 115691  43.0 -83.7       0
#>  7        Ann Arbor MI 113716  42.3 -83.7       0
#>  8          Clinton MI 100517  42.6 -82.9       0
#>  9          Livonia MI  97722  42.4 -83.4       0
#> 10         Dearborn MI  94681  42.3 -83.2       0
#> # ... with 26 more rows
```

We could show this data with a scatterplot, but it's not terribly useful without a reference. You almost always combine point metadata with another layer to make it interpretable.

```
ggplot(mi_cities, aes(lon, lat)) +
  geom_point(aes(size = pop)) +
  scale_size_area() +
  coord_quickmap()

ggplot(mi_cities, aes(lon, lat)) +
  geom_polygon(aes(group = group), mi_counties, fill = NA, colour = "grey50") +
  geom_point(aes(size = pop), colour = "red") +
  scale_size_area() +
  coord_quickmap()
```



3.7.3 Raster images

Instead of displaying context with vector boundaries, you might want to draw a traditional map underneath. This is called a raster image. The easiest way to get a raster map of a given area is to use the ggmap package, which allows you to get data from a variety of online mapping sources including OpenStreetMap and Google Maps. Downloading the raster data is often time consuming so it's a good idea to cache it in a rds file.

```
if (file.exists("mi_raster.rds")) {
  mi_raster <- readRDS("mi_raster.rds")
} else {
  bbox <- c(
    min(mi_counties$lon), min(mi_counties$lat),
    max(mi_counties$lon), max(mi_counties$lat)
  )
  mi_raster <- ggmap::get_openstreetmap(bbox, scale = 8735660)
  saveRDS(mi_raster, "mi_raster.rds")
}
```

(Finding the appropriate `scale` required a lot of manual tweaking.)

You can then plot it with:

```
ggmap::ggmap(mi_raster)

ggmap::ggmap(mi_raster) +
  geom_point(aes(size = pop), mi_cities, colour = "red") +
  scale_size_area()
```

If you have raster data from the raster package, you can convert it to the form needed by ggplot2 with the following code:

```
df <- as.data.frame(raster::rasterToPoints(x))
names(df) <- c("lon", "lat", "x")
```

```
ggplot(df, aes(lon, lat)) +
  geom_raster(aes(fill = x))
```

3.7.4 Area metadata

Sometimes metadata is associated not with a point, but with an area. For example, we can create `mi_census` which provides census information about each county in MI:

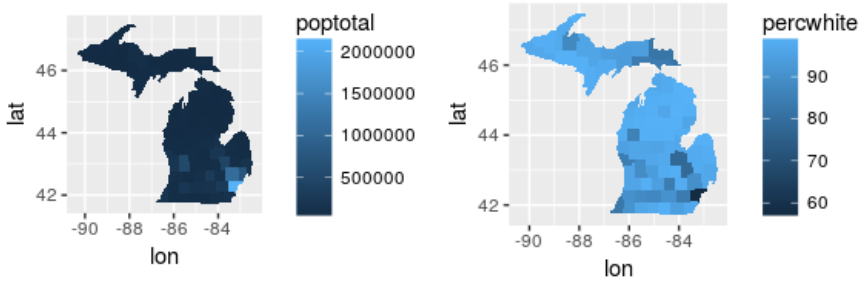
```
mi_census <- midwest %>%
  tbl_df() %>%
  filter(state == "MI") %>%
  mutate(county = tolower(county)) %>%
  select(county, area, poptotal, percwhite, percblack)
mi_census
#> # A tibble: 83 x 5
#>     county  area poptotal percwhite percblack
#>      <chr> <dbl>    <int>     <dbl>     <dbl>
#>  1  alcona 0.041    10145      98.8     0.266
#>  2   alger 0.051     8972      93.9     2.374
#>  3 allegan 0.049    90509      95.9     1.600
#>  4  alpena 0.034    30605      99.2     0.114
#>  5  antrim 0.031    18185      98.4     0.126
#>  6  arenac 0.021    14931      98.4     0.067
#>  7  baraga 0.054     7954      87.6     0.616
#>  8   barry 0.034    50057      98.7     0.208
#>  9     bay 0.026   111723      96.4     1.112
#> 10  benzie 0.020    12200      97.2     0.246
#> # ... with 73 more rows
```

We can't map this data directly because it has no spatial component. Instead, we must first join it to the vector boundaries data. This is not particularly space efficient, but it makes it easy to see exactly what data is being plotted. Here I use `dplyr::left_join()` to combine the two datasets and create a choropleth map.

```
census_counties <- left_join(mi_census, mi_counties, by = c("county" = "id"))
census_counties
#> # A tibble: 1,472 x 8
#>    county  area poptotal percwhite percblack   lon   lat group
#>     <chr> <dbl>    <int>     <dbl>     <dbl> <dbl> <dbl> <dbl>
#>  1 alcona 0.041    10145      98.8     0.266 -83.9  44.9     1
#>  2 alcona 0.041    10145      98.8     0.266 -83.4  44.9     1
#>  3 alcona 0.041    10145      98.8     0.266 -83.4  44.9     1
```

```
#>  4 alcona 0.041    10145      98.8     0.266 -83.3  44.8     1
#>  5 alcona 0.041    10145      98.8     0.266 -83.3  44.8     1
#>  6 alcona 0.041    10145      98.8     0.266 -83.3  44.8     1
#>  7 alcona 0.041    10145      98.8     0.266 -83.3  44.7     1
#>  8 alcona 0.041    10145      98.8     0.266 -83.3  44.7     1
#>  9 alcona 0.041    10145      98.8     0.266 -83.3  44.7     1
#> 10 alcona 0.041    10145      98.8     0.266 -83.3  44.6     1
#> # ... with 1,462 more rows
```

```
ggplot(census_counties, aes(lon, lat, group = county)) +
  geom_polygon(aes(fill = poptotal)) +
  coord_quickmap()

ggplot(census_counties, aes(lon, lat, group = county)) +
  geom_polygon(aes(fill = percwhite)) +
  coord_quickmap()
```



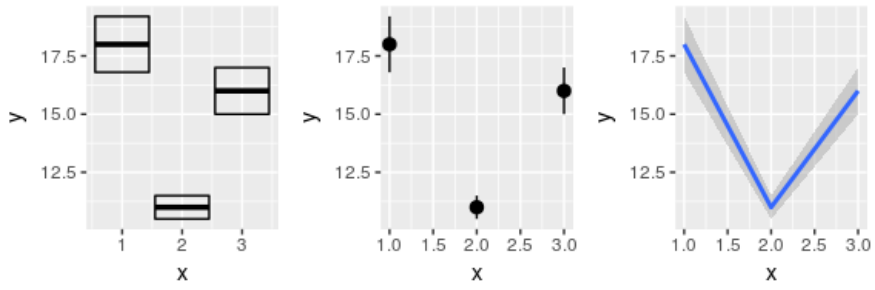
3.8 Revealing uncertainty

If you have information about the uncertainty present in your data, whether it be from a model or from distributional assumptions, it's a good idea to display it. There are four basic families of geoms that can be used for this job, depending on whether the x values are discrete or continuous, and whether or not you want to display the middle of the interval, or just the extent:

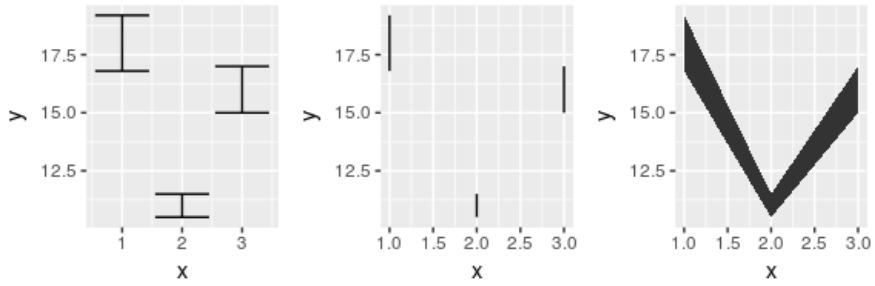
- Discrete x, range: `geom_errorbar()`, `geom_linerange()`
- Discrete x, range & center: `geom_crossbar()`, `geom_pointrange()`
- Continuous x, range: `geom_ribbon()`
- Continuous x, range & center: `geom_smooth(stat = "identity")`

These geoms assume that you are interested in the distribution of y conditional on x and use the aesthetics `ymin` and `ymax` to determine the range of the y values. If you want the opposite, see Section 7.4.2.

```
y <- c(18, 11, 16)
df <- data.frame(x = 1:3, y = y, se = c(1.2, 0.5, 1.0))

base <- ggplot(df, aes(x, y, ymin = y - se, ymax = y + se))
base + geom_crossbar()
base + geom_pointrange()
base + geom_smooth(stat = "identity")
```




```
base + geom_errorbar()
base + geom_linerange()
base + geom_ribbon()
```



Because there are so many different ways to calculate standard errors, the calculation is up to you. For very simple cases, ggplot2 provides some tools in the form of summary functions described below, otherwise you will have to do it yourself. Chapter 11 contains more advice on extracting confidence intervals from more sophisticated models.

3.9 Weighted data

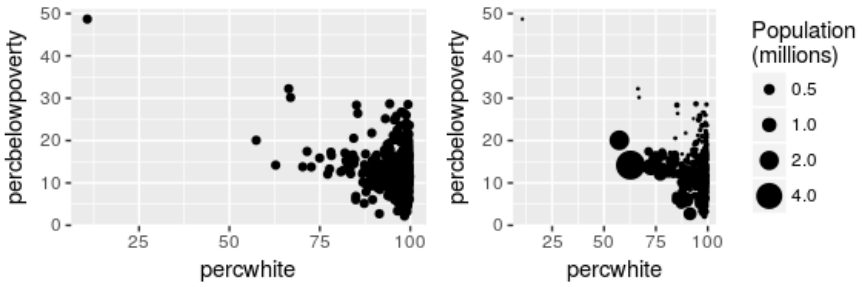
When you have aggregated data where each row in the dataset represents multiple observations, you need some way to take into account the weighting variable. We will use some data collected on Midwest states in the 2000 US census in the built-in `midwest` data frame. The data consists mainly of percentages (e.g., percent white, percent below poverty line, percent with college degree) and some information for each county (area, total population, population density).

There are a few different things we might want to weight by:

- Nothing, to look at numbers of counties.
- Total population, to work with absolute numbers.
- Area, to investigate geographic effects. (This isn't useful for `midwest`, but would be if we had variables like percentage of farmland.)

The choice of a weighting variable profoundly affects what we are looking at in the plot and the conclusions that we will draw. There are two aesthetic attributes that can be used to adjust for weights. Firstly, for simple geoms like lines and points, use the size aesthetic:

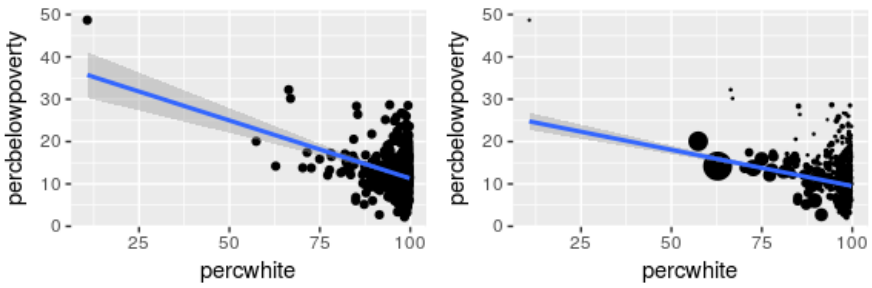
```
# Unweighted
ggplot(midwest, aes(percwhite, percbelowpoverty)) +
  geom_point()

# Weight by population
ggplot(midwest, aes(percwhite, percbelowpoverty)) +
  geom_point(aes(size = poptotal / 1e6)) +
  scale_size_area("Population\n(millions)", breaks = c(0.5, 1, 2, 4))
```



For more complicated grobs which involve some statistical transformation, we specify weights with the `weight` aesthetic. These weights will be passed on to the statistical summary function. Weights are supported for every case where it makes sense: smoothers, quantile regressions, boxplots, histograms,

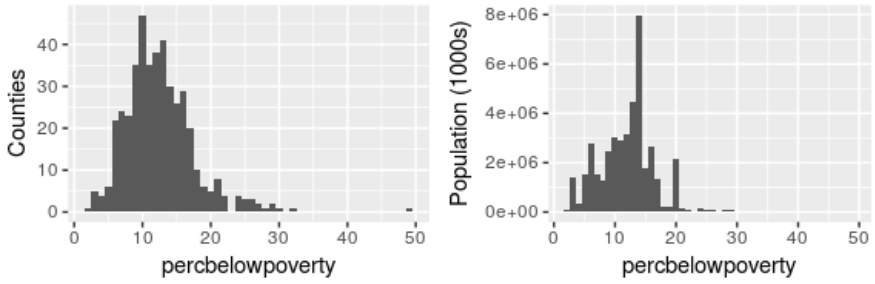
and density plots. You can't see this weighting variable directly, and it doesn't produce a legend, but it will change the results of the statistical summary. The following code shows how weighting by population density affects the relationship between percent white and percent below the poverty line.

```
# Unweighted
ggplot(midwest, aes(percwhite, percbelowpoverty)) +
  geom_point() +
  geom_smooth(method = lm, size = 1)

# Weighted by population
ggplot(midwest, aes(percwhite, percbelowpoverty)) +
  geom_point(aes(size = poptotal / 1e6)) +
  geom_smooth(aes(weight = poptotal), method = lm, size = 1) +
  scale_size_area(guide = "none")
```



When we weight a histogram or density plot by total population, we change from looking at the distribution of the number of counties, to the distribution of the number of people. The following code shows the difference this makes for a histogram of the percentage below the poverty line:

```
ggplot(midwest, aes(percbelowpoverty)) +
  geom_histogram(binwidth = 1) +
  ylab("Counties")

ggplot(midwest, aes(percbelowpoverty)) +
  geom_histogram(aes(weight = poptotal), binwidth = 1) +
  ylab("Population (1000s)")
#> Warning: Ignoring unknown aesthetics: weight
```



3.10 Diamonds data

To demonstrate tools for large datasets, we'll use the built in `diamonds` dataset, which consists of price and quality information for ˜54,000 diamonds:

```
diamonds
#> # A tibble: 53,940 x 10
#>    carat       cut color clarity depth table price     x     y
#>    <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl>
#>  1  0.23     Ideal     E     SI2  61.5    55   326  3.95  3.98
#>  2  0.21   Premium     E     SI1  59.8    61   326  3.89  3.84
#>  3  0.23      Good     E     VS1  56.9    65   327  4.05  4.07
#>  4  0.29   Premium     I     VS2  62.4    58   334  4.20  4.23
#>  5  0.31      Good     J     SI2  63.3    58   335  4.34  4.35
#>  6  0.24 Very Good     J    VVS2  62.8    57   336  3.94  3.96
#>  7  0.24 Very Good     I    VVS1  62.3    57   336  3.95  3.98
#>  8  0.26 Very Good     H     SI1  61.9    55   337  4.07  4.11
#>  9  0.22      Fair     E     VS2  65.1    61   337  3.87  3.78
#> 10  0.23 Very Good     H     VS1  59.4    61   338  4.00  4.05
#> # ... with 53,930 more rows, and 1 more variables: z <dbl>
```

The data contains the four C's of diamond quality: carat, cut, colour and clarity; and five physical measurements: depth, table, x, y and z, as described in Figure 3.1.

The dataset has not been well cleaned, so as well as demonstrating interesting facts about diamonds, it also shows some data quality problems.

3.11 Displaying distributions

There are a number of geoms that can be used to display distributions, depending on the dimensionality of the distribution, whether it is continuous

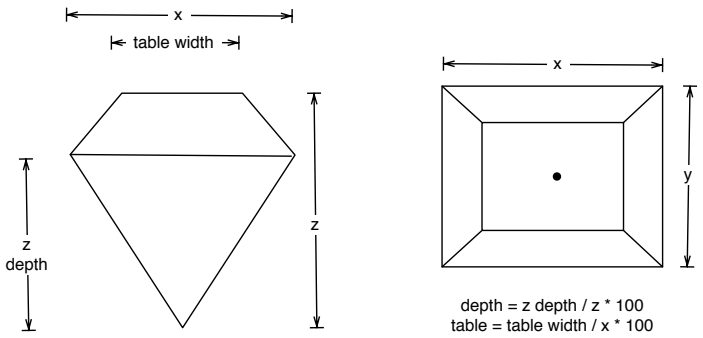
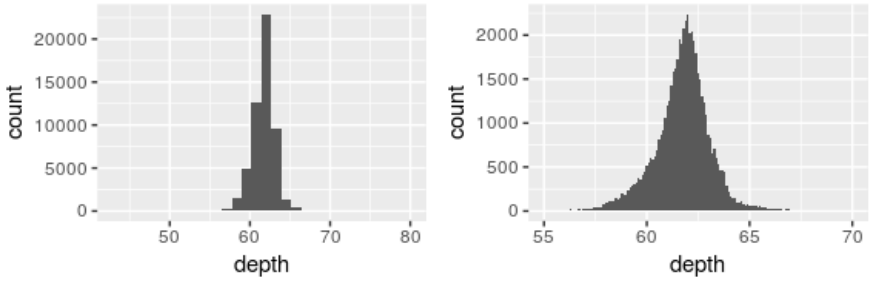


Fig. 3.1 How the variables x, y, z, table and depth are measured.

or discrete, and whether you are interested in the conditional or joint distribution.

For 1d continuous distributions the most important geom is the histogram, `geom_histogram()`:

```
ggplot(diamonds, aes(depth)) +
  geom_histogram()
#> `stat_bin()` using `bins = 30`. Pick better value with
#> `binwidth`.
ggplot(diamonds, aes(depth)) +
  geom_histogram(binwidth = 0.1) +
  xlim(55, 70)
#> Warning: Removed 45 rows containing non-finite values (stat_bin).
```



It is important to experiment with binning to find a revealing view. You can change the `binwidth`, specify the number of `bins`, or specify the exact location of the `breaks`. Never rely on the default parameters to get a revealing view of the distribution. Zooming in on the x axis, `xlim(55, 70)`, and selecting a smaller bin width, `binwidth = 0.1`, reveals far more detail.

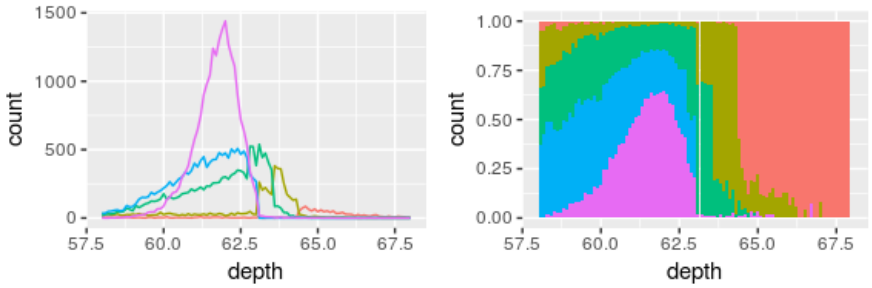
When publishing figures, don't forget to include information about important parameters (like bin width) in the caption.

If you want to compare the distribution between groups, you have a few options:

- Show small multiples of the histogram, `facet_wrap(~ var)`.
- Use colour and a frequency polygon, `geom_freqpoly()` .
- Use a "conditional density plot", `geom_histogram(position = "fill")`.

The frequency polygon and conditional density plots are shown below. The conditional density plot uses `position_fill()` to stack each bin, scaling it to the same height. This plot is perceptually challenging because you need to compare bar heights, not positions, but you can see the strongest patterns.

```
ggplot(diamonds, aes(depth)) +
  geom_freqpoly(aes(colour = cut), binwidth = 0.1, na.rm = TRUE) +
  xlim(58, 68) +
  theme(legend.position = "none")
ggplot(diamonds, aes(depth)) +
  geom_histogram(aes(fill = cut), binwidth = 0.1, position = "fill",
    na.rm = TRUE) +
  xlim(58, 68) +
  theme(legend.position = "none")
```



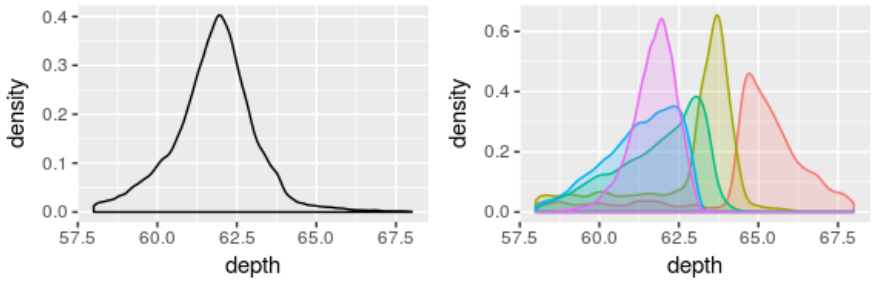
(I've suppressed the legends to focus on the display of the data.)

Both the histogram and frequency polygon geom use the same underlying statistical transformation: `stat = "bin"`. This statistic produces two output variables: `count` and `density`. By default, count is mapped to y-position, because it's most interpretable. The density is the count divided by the total count multiplied by the bin width, and is useful when you want to compare the shape of the distributions, not the overall size.

An alternative to a bin-based visualisation is a density estimate. `geom_density()` places a little normal distribution at each data point and sums up all the curves. It has desirable theoretical properties, but is more

difficult to relate back to the data. Use a density plot when you know that the underlying density is smooth, continuous and unbounded. You can use the `adjust` parameter to make the density more or less smooth.

```
ggplot(diamonds, aes(depth)) +
  geom_density(na.rm = TRUE) +
  xlim(58, 68) +
  theme(legend.position = "none")
ggplot(diamonds, aes(depth, fill = cut, colour = cut)) +
  geom_density(alpha = 0.2, na.rm = TRUE) +
  xlim(58, 68) +
  theme(legend.position = "none")
```

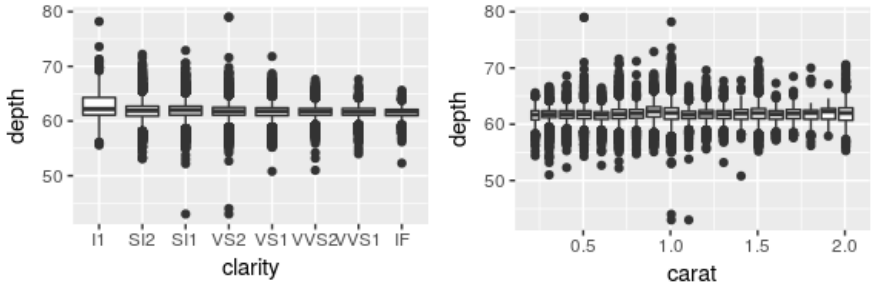


Note that the area of each density estimate is standardised to one so that you lose information about the relative size of each group.

The histogram, frequency polygon and density display a detailed view of the distribution. However, sometimes you want to compare many distributions, and it's useful to have alternative options that sacrifice quality for quantity. Here are three options:

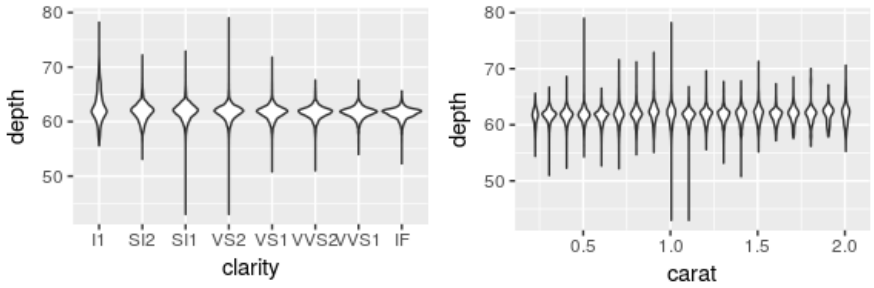
- `geom_boxplot()`: the box-and-whisker plot shows five summary statistics along with individual "outliers". It displays far less information than a histogram, but also takes up much less space.
  You can use boxplot with both categorical and continuous x. For continuous x, you'll also need to set the group aesthetic to define how the x variable is broken up into bins. A useful helper function is `cut_width()`:

  ```
  ggplot(diamonds, aes(clarity, depth)) +
    geom_boxplot()
  ggplot(diamonds, aes(carat, depth)) +
    geom_boxplot(aes(group = cut_width(carat, 0.1))) +
    xlim(NA, 2.05)
  #> Warning: Removed 997 rows containing non-finite values
  #> (stat_boxplot).
  ```



- `geom_violin()`: the violin plot is a compact version of the density plot. The underlying computation is the same, but the results are displayed in a similar fashion to the boxplot:

```
ggplot(diamonds, aes(clarity, depth)) +
  geom_violin()
ggplot(diamonds, aes(carat, depth)) +
  geom_violin(aes(group = cut_width(carat, 0.1))) +
  xlim(NA, 2.05)
#> Warning: Removed 997 rows containing non-finite values
#> (stat_ydensity).
```



- `geom_dotplot()`: draws one point for each observation, carefully adjusted in space to avoid overlaps and show the distribution. It is useful for smaller datasets.

3.11.1 Exercises

1. What binwidth tells you the most interesting story about the distribution of `carat`?

2. Draw a histogram of `price`. What interesting patterns do you see?
3. How does the distribution of `price` vary with `clarity`?
4. Overlay a frequency polygon and density plot of `depth`. What computed variable do you need to map to `y` to make the two plots comparable? (You can either modify `geom_freqpoly()` or `geom_density()`.)

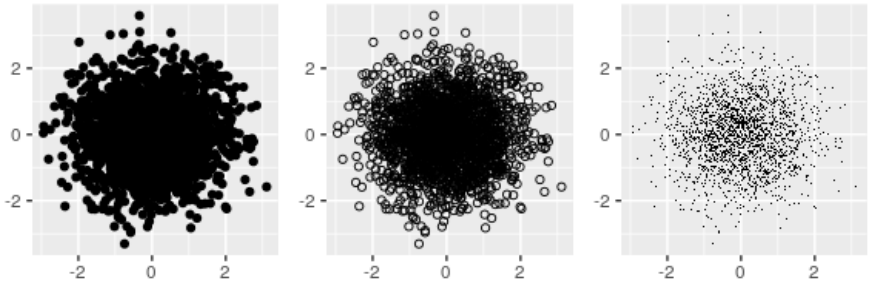
3.12 Dealing with overplotting

The scatterplot is a very important tool for assessing the relationship between two continuous variables. However, when the data is large, points will be often plotted on top of each other, obscuring the true relationship. In extreme cases, you will only be able to see the extent of the data, and any conclusions drawn from the graphic will be suspect. This problem is called **overplotting**.

There are a number of ways to deal with it depending on the size of the data and severity of the overplotting. The first set of techniques involves tweaking aesthetic properties. These tend to be most effective for smaller datasets:

- Very small amounts of overplotting can sometimes be alleviated by making the points smaller, or using hollow glyphs. The following code shows some options for 2000 points sampled from a bivariate normal distribution.

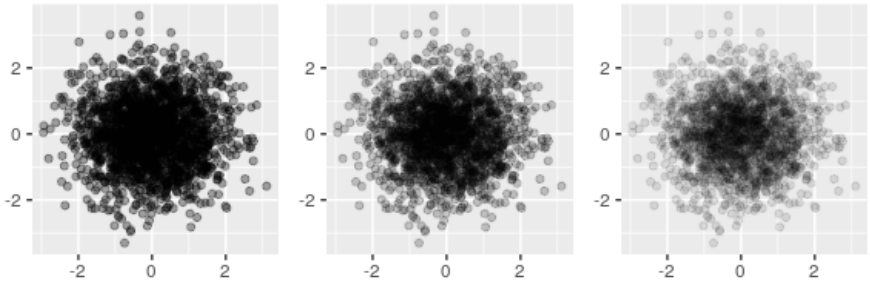
```
df <- data.frame(x = rnorm(2000), y = rnorm(2000))
norm <- ggplot(df, aes(x, y)) + xlab(NULL) + ylab(NULL)
norm + geom_point()
norm + geom_point(shape = 1) # Hollow circles
norm + geom_point(shape = ".") # Pixel sized
```



- For larger datasets with more overplotting, you can use alpha blending (transparency) to make the points transparent. If you specify `alpha` as a ratio, the denominator gives the number of points that must be overplotted

to give a solid colour. Values smaller than ˜1/500 are rounded down to zero, giving completely transparent points.

```
norm + geom_point(alpha = 1 / 3)
norm + geom_point(alpha = 1 / 5)
norm + geom_point(alpha = 1 / 10)
```

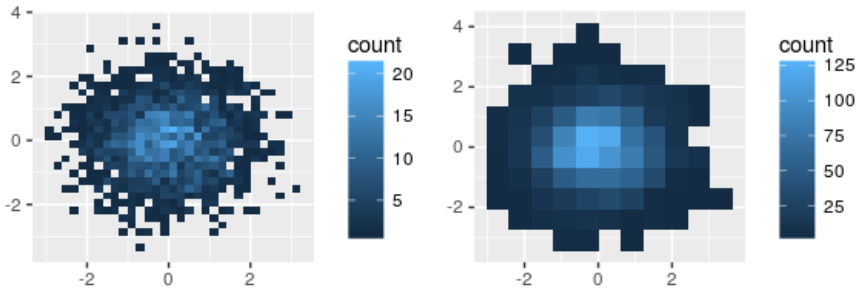


- If there is some discreteness in the data, you can randomly jitter the points to alleviate some overlaps with `geom_jitter()`. This can be particularly useful in conjunction with transparency. By default, the amount of jitter added is 40% of the resolution of the data, which leaves a small gap between adjacent regions. You can override the default with `width` and `height` arguments.

Alternatively, we can think of overplotting as a 2d density estimation problem, which gives rise to two more approaches:

- Bin the points and count the number in each bin, then visualise that count (the 2d generalisation of the histogram), `geom_bin2d()`. Breaking the plot into many small squares can produce distracting visual artefacts. (D. B. Carr et al. 1987) suggests using hexagons instead, and this is implemented in `geom_hex()`, using the **hexbin** package (D. Carr, Lewin-Koh, and Mchler 2014).
  The code below compares square and hexagonal bins, using parameters `bins` and `binwidth` to control the number and size of the bins.

```
norm + geom_bin2d()
norm + geom_bin2d(bins = 10)
```




```
norm + geom_hex()
#> Warning: Computation failed in `stat_binhex()`:
#> Package `hexbin` required for `stat_binhex`.
#> Please install and try again.
norm + geom_hex(bins = 10)
#> Warning: Computation failed in `stat_binhex()`:
#> Package `hexbin` required for `stat_binhex`.
#> Please install and try again.
```


- Estimate the 2d density with `stat_density2d()`, and then display using one of the techniques for showing 3d surfaces in Section 3.6.
- If you are interested in the conditional distribution of y given x, then the techniques of Section 2.6.3 will also be useful.

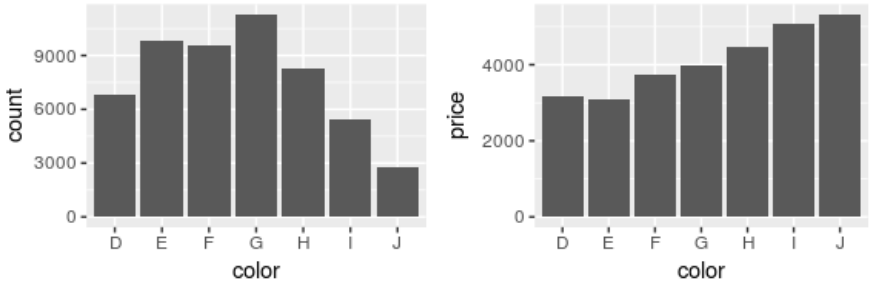
Another approach to dealing with overplotting is to add data summaries to help guide the eye to the true shape of the pattern within the data. For example, you could add a smooth line showing the centre of the data with `geom_smooth()` or use one of the summaries below.

3.13 Statistical summaries

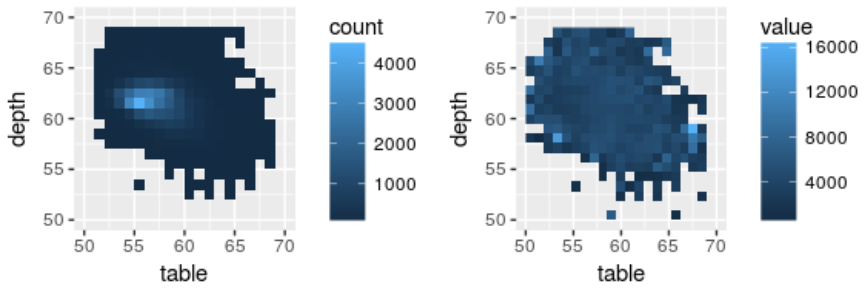
`geom_histogram()` and `geom_bin2d()` use a familiar geom, `geom_bar()` and `geom_raster()`, combined with a new statistical transformation, `stat_bin()` and `stat_bin2d()`. `stat_bin()` and `stat_bin2d()` combine the data into bins and count the number of observations in each bin. But what if we want a summary other than count? So far, we've just used the default statistical transformation associated with each geom. Now we're going to explore how to use `stat_summary_bin()` to `stat_summary_2d()` to compute different summaries.

Let's start with a couple of examples with the diamonds data. The first example in each pair shows how we can count the number of diamonds in each bin; the second shows how we can compute the average price.

```
ggplot(diamonds, aes(color)) +
  geom_bar()

ggplot(diamonds, aes(color, price)) +
  geom_bar(stat = "summary_bin", fun.y = mean)
```




```
ggplot(diamonds, aes(table, depth)) +
  geom_bin2d(binwidth = 1, na.rm = TRUE) +
  xlim(50, 70) +
  ylim(50, 70)

ggplot(diamonds, aes(table, depth, z = price)) +
  geom_raster(binwidth = 1, stat = "summary_2d", fun = mean,
    na.rm = TRUE) +
  xlim(50, 70) +
  ylim(50, 70)
```



To get more help on the arguments associated with the two transformations, look at the help for `stat_summary_bin()` and `stat_summary_2d()`. You can control the size of the bins and the summary functions. `stat_summary_bin()` can produce `y`, `ymin` and `ymax` aesthetics, also making it useful for displaying measures of spread. See the docs for more details. You'll learn more about how geoms and stats interact in Section 5.6.

These summary functions are quite constrained but are often useful for a quick first pass at a problem. If you find them restraining, you'll need to do the summaries yourself. See Section 10.4 for more details.

3.14 Add-on packages

If the built-in tools in ggplot2 don't do what you need, you might want to use a special purpose tool built into one of the packages built on top of ggplot2. Some of the packages that I was familiar with when the book was published include:

- animInt, `https://github.com/tdhock/animint`, lets you make you ggplot2 graphics interactive, adding querying, filtering and linking.
- GGally, `https://github.com/ggobi/ggally`, provides a very flexible scatterplot matrix, amongst other tools.
- ggbio, `http://www.tengfei.name/ggbio/`, provides a number of specialised geoms for genomic data.
- ggdendro, `https://github.com/andrie/ggdendro`, turns data from tree methods in to data frames that can easily be displayed with ggplot2.
- ggfortify, `https://github.com/sinhrks/ggfortify`, provides fortify and autoplot methods to handle objects from some popular R packages.
- ggenealogy, `https://cran.r-project.org/package=ggenealogy`, helps explore and visualise genealogy data.
- ggmcmc, `http://xavier-fim.net/packages/ggmcmc/`, provides a set of flexible tools for visualising the samples generated by MCMC methods.

- ggparallel, https://cran.r-project.org/package=ggparallel: easily draw parallel coordinates plots, and the closely related hammock and common angle plots.
- ggtern, http://www.ggtern.com, lets you use ggplot2 to draw ternary diagrams, used when you have three variables that always sum to one.
- ggtree, https://github.com/GuangchuangYu/ggtree, provides tools to view and annotate phylogenetic tree with different types of meta-data.
- granovaGG, https://github.com/briandk/granovaGG, provides tools to visualise ANOVA results.
- plotluck, https://github.com/stefan-schroedl/plotluck: the ggplot2 version of Google's "I'm feeling lucky". It automatically creates plots for one, two or three variables.

A great place to track new extensions is http://www.ggplot2-exts.org, by Daniel Emaasit.

References

Carr, D. B., R. J. Littlefield, W. L. Nicholson, and J. S. Littlefield. 1987. "Scatterplot Matrix Techniques for Large N." *Journal of the American Statistical Association* 82 (398): 424–36.

Carr, Dan, Nicholas Lewin-Koh, and Martin Mchler. 2014. *Hexbin: Hexagonal Binning Routines*.

Part II
The Grammar

Chapter 4
Mastering the grammar

4.1 Introduction

In order to unlock the full power of ggplot2, you'll need to master the underlying grammar. By understanding the grammar, and how its components fit together, you can create a wider range of visualizations, combine multiple sources of data, and customise to your heart's content.

This chapter describes the theoretical basis of ggplot2: the layered grammar of graphics. The layered grammar is based on Wilkinson's grammar of graphics (Wilkinson 2005), but adds a number of enhancements that help it to be more expressive and fit seamlessly into the R environment. The differences between the layered grammar and Wilkinson's grammar are described fully in Wickham (2008). In this chapter you will learn a little bit about each component of the grammar and how they all fit together. The next chapters discuss the components in more detail, and provide more examples of how you can use them in practice.

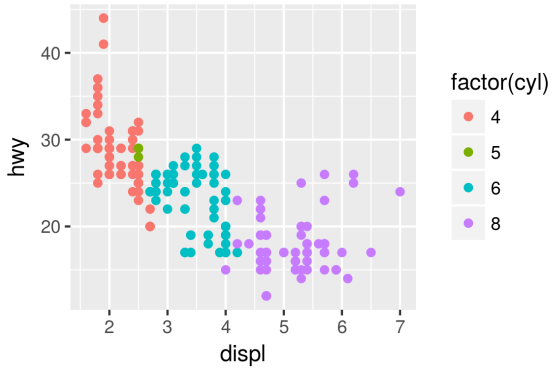
The grammar makes it easier for you to iteratively update a plot, changing a single feature at a time. The grammar is also useful because it suggests the high-level aspects of a plot that *can* be changed, giving you a framework to think about graphics, and hopefully shortening the distance from mind to paper. It also encourages the use of graphics customised to a particular problem, rather than relying on specific chart types.

This chapter begins by describing in detail the process of drawing a simple plot. Section 4.2 starts with a simple scatterplot, then Section 4.3 makes it more complex by adding a smooth line and facetting. While working through these examples you will be introduced to all six components of the grammar, which are then defined more precisely in Section 4.4.

4.2 Building a scatterplot

How are engine size and fuel economy related? We might create a scatterplot of engine displacement and highway mpg with points coloured by number of cylinders:

```
ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_point()
```



You can create plots like this easily, but what is going on underneath the surface? How does ggplot2 draw this plot?

4.2.1 Mapping aesthetics to data

What precisely is a scatterplot? You have seen many before and have probably even drawn some by hand. A scatterplot represents each observation as a point, positioned according to the value of two variables. As well as a horizontal and vertical position, each point also has a size, a colour and a shape. These attributes are called **aesthetics**, and are the properties that can be perceived on the graphic. Each aesthetic can be mapped to a variable, or set to a constant value. In the previous graphic, `displ` is mapped to horizontal position, `hwy` to vertical position and `cyl` to colour. Size and shape are not mapped to variables, but remain at their (constant) default values.

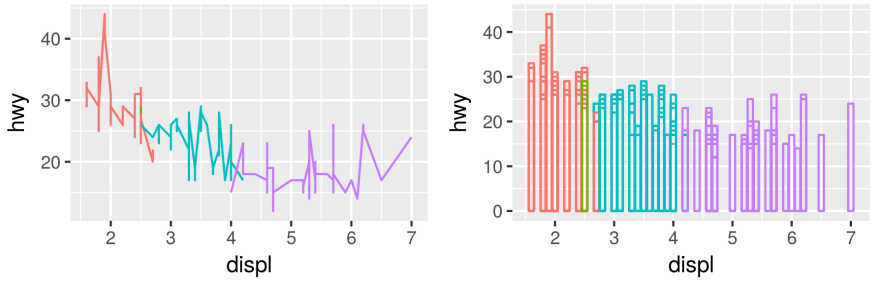
Once we have these mappings we can create a new dataset that records this information:

x	y	colour
1.8	29	4

x	y	colour
1.8	29	4
2.0	31	4
2.0	30	4
2.8	26	6
2.8	26	6
3.1	27	6
1.8	26	4

This new dataset is a result of applying the aesthetic mappings to the original data. We can create many different types of plots using this data. The scatterplot uses points, but were we instead to draw lines we would get a line plot. If we used bars, we'd get a bar plot. Neither of those examples makes sense for this data, but we could still draw them (I've omitted the legends to save space):

```
ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_line() +
  theme(legend.position = "none")
ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_bar(stat = "identity", position = "identity", fill = NA) +
  theme(legend.position = "none")
```



In ggplot, we can produce many plots that don't make sense, yet are grammatically valid. This is no different than English, where we can create senseless but grammatical sentences like the angry rock barked like a comma.

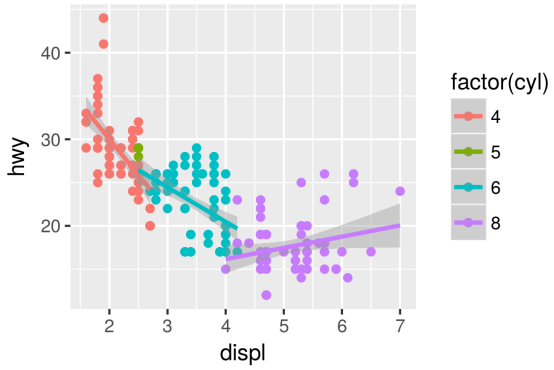
Points, lines and bars are all examples of geometric objects, or **geoms**. Geoms determine the "type" of the plot. Plots that use a single geom are often given a special name:

Named plot	Geom	Other features
scatterplot	point	

Named plot	Geom	Other features
bubblechart	point	size mapped to a variable
barchart	bar	
box-and-whisker plot	boxplot	
line chart	line	

More complex plots with combinations of multiple geoms don't have a special name, and we have to describe them by hand. For example, this plot overlays a per group regression line on top of a scatterplot:

```
ggplot(mpg, aes(displ, hwy, colour = factor(cyl))) +
  geom_point() +
  geom_smooth(method = "lm")
```



What would you call this plot? Once you've mastered the grammar, you'll find that many of the plots that you produce are uniquely tailored to your problems and will no longer have special names.

4.2.2 Scaling

The values in the previous table have no meaning to the computer. We need to convert them from data units (e.g., litres, miles per gallon and number of cylinders) to graphical units (e.g., pixels and colours) that the computer can display. This conversion process is called **scaling** and performed by scales. Now that these values are meaningful to the computer, they may not be meaningful to us: colours are represented by a six-letter hexadecimal string, sizes by a number and shapes by an integer. These aesthetic specifications that are meaningful to R are described in `vignette("ggplot2-specs")`.

In this example, we have three aesthetics that need to be scaled: horizontal position (`x`), vertical position (`y`) and `colour`. Scaling position is easy in this example because we are using the default linear scales. We need only a linear mapping from the range of the data to $[0, 1]$. We use $[0, 1]$ instead of exact pixels because the drawing system that ggplot2 uses, **grid**, takes care of that final conversion for us. A final step determines how the two positions (x and y) are combined to form the final location on the plot. This is done by the coordinate system, or **coord**. In most cases this will be Cartesian coordinates, but it might be polar coordinates, or a spherical projection used for a map.

The process for mapping the colour is a little more complicated, as we have a non-numeric result: colours. However, colours can be thought of as having three components, corresponding to the three types of colour-detecting cells in the human eye. These three cell types give rise to a three-dimensional colour space. Scaling then involves mapping the data values to points in this space. There are many ways to do this, but here since `cyl` is a categorical variable we map values to evenly spaced hues on the colour wheel, as shown in Figure 4.1. A different mapping is used when the variable is continuous.

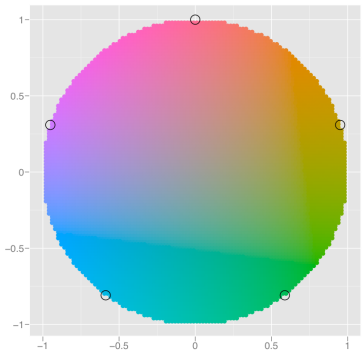


Fig. 4.1 A colour wheel illustrating the choice of five equally spaced colours. This is the default scale for discrete variables.

The result of these conversions is below. As well as aesthetics that have been mapped to variable, we also include aesthetics that are constant. We need these so that the aesthetics for each point are completely specified and R can draw the plot. The points will be filled circles (shape 19 in R) with a 1-mm diameter:

x	y	colour	size	shape
0.037	0.531	#F8766D	1	19
0.037	0.531	#F8766D	1	19
0.074	0.594	#F8766D	1	19
0.074	0.562	#F8766D	1	19

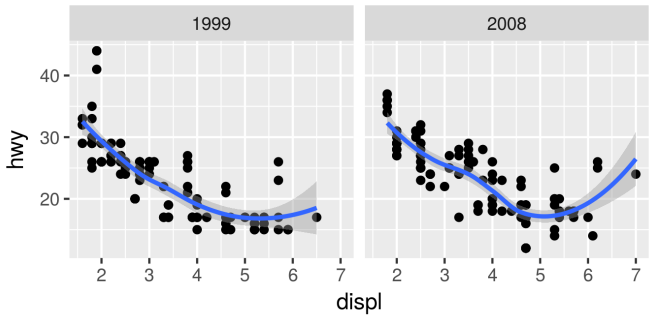
x	y	colour	size	shape
0.222	0.438	#00BFC4	1	19
0.222	0.438	#00BFC4	1	19
0.278	0.469	#00BFC4	1	19
0.037	0.438	#F8766D	1	19

Finally, we need to render this data to create the graphical objects that are displayed on the screen. To create a complete plot we need to combine graphical objects from three sources: the *data*, represented by the point geom; the *scales and coordinate system*, which generate axes and legends so that we can read values from the graph; and *plot annotations*, such as the background and plot title.

4.3 Adding complexity

With a simple example under our belts, let's now turn to look at this slightly more complicated example:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth() +
  facet_wrap(~year)
```



This plot adds three new components to the mix: facets, multiple layers and statistics. The facets and layers expand the data structure described above: each facet panel in each layer has its own dataset. You can think of this as a 3d array: the panels of the facets form a 2d grid, and the layers extend upwards in the 3rd dimension. In this case the data in the layers is the same, but in general we can plot different datasets on different layers.

The smooth layer is different to the point layer because it doesn't display the raw data, but instead displays a statistical transformation of the data. Specifically, the smooth layer fits a smooth line through the middle of the data. This requires an additional step in the process described above: after mapping the data to aesthetics, the data is passed to a statistical transformation, or **stat**, which manipulates the data in some useful way. In this example, the stat fits the data to a loess smoother, and then returns predictions from evenly spaced points within the range of the data. Other useful stats include 1 and 2d binning, group means, quantile regression and contouring.

As well as adding an additional step to summarise the data, we also need some extra steps when we get to the scales. This is because we now have multiple datasets (for the different facets and layers) and we need to make sure that the scales are the same across all of them. Scaling actually occurs in three parts: transforming, training and mapping. We haven't mentioned transformation before, but you have probably seen it before in log-log plots. In a log-log plot, the data values are not linearly mapped to position on the plot, but are first log-transformed.

- Scale transformation occurs before statistical transformation so that statistics are computed on the scale-transformed data. This ensures that a plot of $\log(x)$ vs. $\log(y)$ on linear scales looks the same as x vs. y on log scales. There are many different transformations that can be used, including taking square roots, logarithms and reciprocals. See Section 6.6.1 for more details.
- After the statistics are computed, each scale is trained on every dataset from all the layers and facets. The training operation combines the ranges of the individual datasets to get the range of the complete data. Without this step, scales could only make sense locally and we wouldn't be able to overlay different layers because their positions wouldn't line up. Sometimes we do want to vary position scales across facets (but never across layers), and this is described more fully in Section 7.2.3.
- Finally the scales map the data values into aesthetic values. This is a local operation: the variables in each dataset are mapped to their aesthetic values, producing a new dataset that can then be rendered by the geoms.

Figure 4.2 illustrates the complete process schematically.

4.4 Components of the layered grammar

In the examples above, we have seen some of the components that make up a plot: data and aesthetic mappings, geometric objects (geoms), statistical transformations (stats), scales, and facetting. We have also touched on the coordinate system. One thing we didn't mention is the position adjustment, which deals with overlapping graphic objects. Together, the data, mappings,

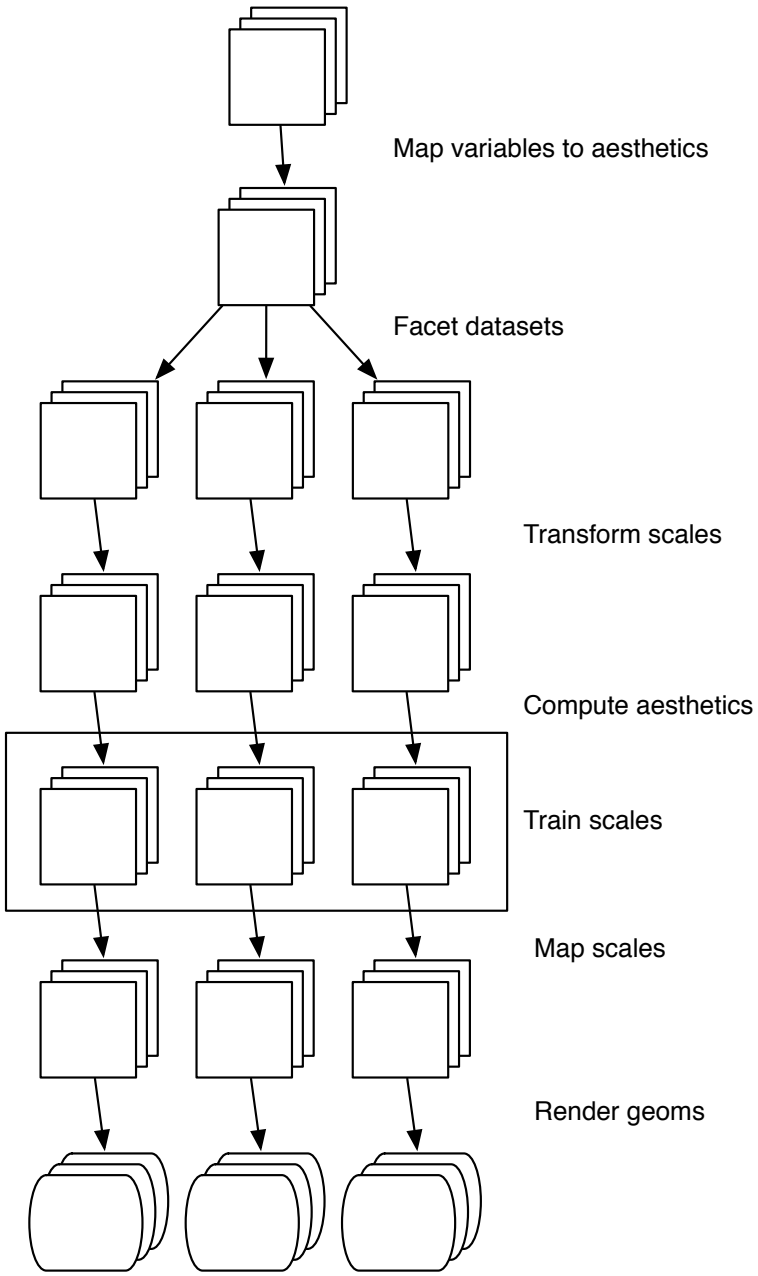


Fig. 4.2 Schematic description of the plot generation process. Each square represents a layer, and this schematic represents a plot with three layers and three panels. All steps work by transforming individual data frames except for training scales, which doesn't affect the data frame and operates across all datasets simultaneously.

stat, geom and position adjustment form a **layer**. A plot may have multiple layers, as in the example where we overlaid a smoothed line on a scatterplot. All together, the layered grammar defines a plot as the combination of:

- A default dataset and set of mappings from variables to aesthetics.
- One or more layers, each composed of a geometric object, a statistical transformation, a position adjustment, and optionally, a dataset and aesthetic mappings.
- One scale for each aesthetic mapping.
- A coordinate system.
- The facetting specification.

The following sections describe each of the higher-level components more precisely, and point you to the parts of the book where they are documented.

4.4.1 Layers

Layers are responsible for creating the objects that we perceive on the plot. A layer is composed of five parts:

1. Data
2. Aesthetic mappings.
3. A statistical transformation (stat).
4. A geometric object (geom).
5. A position adjustment.

The properties of a layer are described in Chapter 5 and their uses for data visualisation in Chapter 3.

4.4.2 Scales

A **scale** controls the mapping from data to aesthetic attributes, and we need a scale for every aesthetic used on a plot. Each scale operates across all the data in the plot, ensuring a consistent mapping from data to aesthetics. Some examples are shown in Figure 4.3.

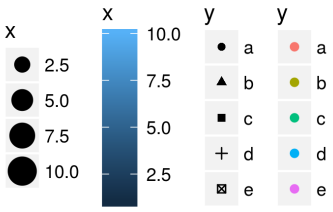


Fig. 4.3 Examples of legends from four different scales. From left to right: continuous variable mapped to size, and to colour, discrete variable mapped to shape, and to colour. The ordering of scales seems upside-down, but this matches the labelling of the y-axis: small values occur at the bottom.

A scale is a function and its inverse, along with a set of parameters. For example, the colour gradient scale maps a segment of the real line to a path through a colour space. The parameters of the function define whether the path is linear or curved, which colour space to use (e.g., LUV or RGB), and the colours at the start and end.

The inverse function is used to draw a guide so that you can read values from the graph. Guides are either axes (for position scales) or legends (for everything else). Most mappings have a unique inverse (i.e., the mapping function is one-to-one), but many do not. A unique inverse makes it possible to recover the original data, but this is not always desirable if we want to focus attention on a single aspect.

For more details, see Chapter 6.

4.4.3 Coordinate system

A coordinate system, or **coord** for short, maps the position of objects onto the plane of the plot. Position is often specified by two coordinates (x, y), but potentially could be three or more (although this is not implemented in ggplot2). The Cartesian coordinate system is the most common coordinate system for two dimensions, while polar coordinates and various map projections are used less frequently.

Coordinate systems affect all position variables simultaneously and differ from scales in that they also change the appearance of the geometric objects. For example, in polar coordinates, bar geoms look like segments of a circle. Additionally, scaling is performed before statistical transformation, while coordinate transformations occur afterward. The consequences of this are shown in Section 7.5.

Coordinate systems control how the axes and grid lines are drawn. Figure 4.4 illustrates three different types of coordinate systems. Very little advice is available for drawing these for non-Cartesian coordinate systems, so a lot of

work needs to be done to produce polished output. See Section 7.3 for more details.

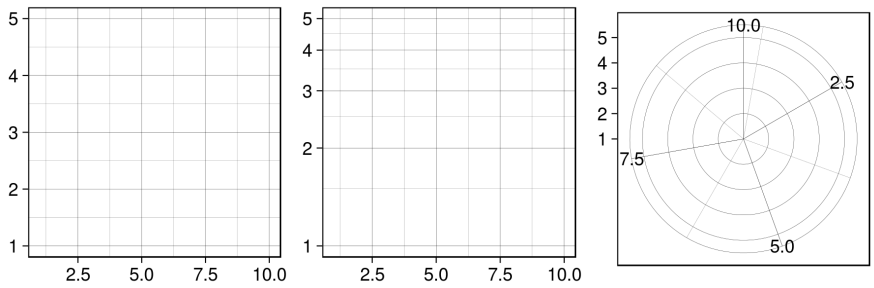


Fig. 4.4 Examples of axes and grid lines for three coordinate systems: Cartesian, semi-log and polar. The polar coordinate system illustrates the difficulties associated with non-Cartesian coordinates: it is hard to draw the axes well.

4.4.4 Facetting

There is also another thing that turns out to be sufficiently useful that we should include it in our general framework: facetting, a general case of conditioned or trellised plots. This makes it easy to create small multiples, each showing a different subset of the whole dataset. This is a powerful tool when investigating whether patterns hold across all conditions. The facetting specification describes which variables should be used to split up the data, and whether position scales should be free or constrained. Facetting is described in Chapter 7.

4.5 Exercises

1. One of the best ways to get a handle on how the grammar works is to apply it to the analysis of existing graphics. For each of the graphics listed below, write down the components of the graphic. Don't worry if you don't know what the corresponding functions in ggplot2 are called (or if they even exist!), instead focussing on recording the key elements of a plot so you could communicate it to someone else.

    1. "Napoleon's march" by Charles John Minard: `http://www.datavis.ca/gallery/re-minard.php`
    2. "Where the Heat and the Thunder Hit Their Shots", by Jeremy White, Joe Ward, and Matthew Ericson at The New York Times. `http://nyti.ms/1duzTvY`

3. "London Cycle Hire Journeys", by James Cheshire. http://bit.ly/1S2cyRy
4. The Pew Research Center's favorite data visualizations of 2014: http://pewrsr.ch/1KZSSN6
5. "The Tony's Have Never Been so Dominated by Women", by Joanna Kao at FiveThirtyEight: http://53eig.ht/1cJRCyG.
6. "In Climbing Income Ladder, Location Matters" by the Mike Bostock, Shan Carter, Amanda Cox, Matthew Ericson, Josh Keller, Alicia Parlapiano, Kevin Quealy and Josh Williams at the New York Times: http://nyti.ms/1S2dJQT
7. "Dissecting a Trailer: The Parts of the Film That Make the Cut", by Shan Carter, Amanda Cox, and Mike Bostock at the New York Times: http://nyti.ms/1KTJQOE

References

Wickham, Hadley. 2008. "Practical Tools for Exploring Data and Models." PhD thesis, Iowa State University. http://had.co.nz/thesis.

Wilkinson, Leland. 2005. *The Grammar of Graphics*. 2nd ed. Statistics and Computing. Springer.

Chapter 5
Build a plot layer by layer

5.1 Introduction

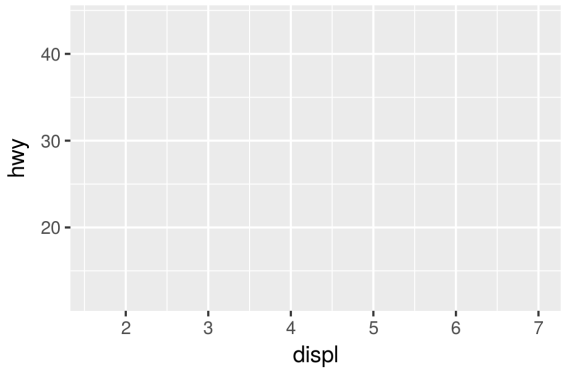
One of the key ideas behind ggplot2 is that it allows you to easily iterate, building up a complex plot a layer at a time. Each layer can come from a different dataset and have a different aesthetic mapping, making it possible to create sophisticated plots that display data from multiple sources.

You've already created layers with functions like `geom_point()` and `geom_histogram()`. In this chapter, you'll dive into the details of a layer, and how you can control all five components: data, the aesthetic mappings, the geom, stat, and position adjustments. The goal here is to give you the tools to build sophisticated plots tailored to the problem at hand.

5.2 Building a plot

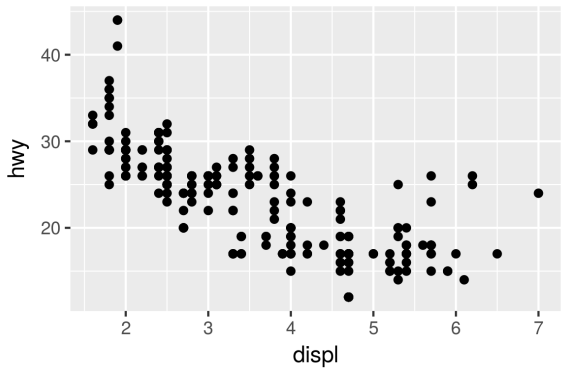
So far, whenever we've created a plot with `ggplot()`, we've immediately added on a layer with a geom function. But it's important to realise that there really are two distinct steps. First we create a plot with default dataset and aesthetic mappings:

```
p <- ggplot(mpg, aes(displ, hwy))
p
```



There's nothing to see yet, so we need to add a layer:

```
p + geom_point()
```



`geom_point()` is a shortcut. Behind the scenes it calls the `layer()` function to create a new layer:

```
p + layer(
  mapping = NULL,
  data = NULL,
  geom = "point",
  stat = "identity",
  position = "identity"
)
```

This call fully specifies the five components to the layer:

- **mapping**: A set of aesthetic mappings, specified using the `aes()` function and combined with the plot defaults as described in Section 5.4. If `NULL`, uses the default mapping set in `ggplot()`.
- **data**: A dataset which overrides the default plot dataset. It is usually omitted (set to `NULL`), in which case the layer will use the default data specified in `ggplot()`. The requirements for data are explained in more detail in Section 5.3.
- **geom**: The name of the geometric object to use to draw each observation. Geoms are discussed in more detail in Section 5.3, and Chapter 3 explores their use in more depth.
  Geoms can have additional arguments. All geoms take aesthetics as parameters. If you supply an aesthetic (e.g. colour) as a parameter, it will not be scaled, allowing you to control the appearance of the plot, as described in Section 5.4.2. You can pass params in ... (in which case stat and geom parameters are automatically teased apart), or in a list passed to `geom_params`.
- **stat**: The name of the statistical tranformation to use. A statistical transformation performs some useful statistical summary, and is key to histograms and smoothers. To keep the data as is, use the "identity" stat. Learn more in Section 5.6.
  You only need to set one of stat and geom: every geom has a default stat, and every stat a default geom.
  Most stats take additional parameters to specify the details of statistical transformation. You can supply params either in ... (in which case stat and geom parameters are automatically teased apart), or in a list called `stat_params`.
- **position**: The method used to adjust overlapping objects, like jittering, stacking or dodging. More details in Section 5.7.

It's useful to understand the `layer()` function so you have a better mental model of the layer object. But you'll rarely use the full `layer()` call because it's so verbose. Instead, you'll use the shortcut `geom_` functions: `geom_point(mapping, data, ...)` is exactly equivalent to `layer(mapping, data, geom = "point", ...)`.

5.3 Data

Every layer must have some data associated with it, and that data must be in a tidy data frame. You'll learn about tidy data in Chapter 9, but for now, all you need to know is that a tidy data frame has variables in the columns and observations in the rows. This is a strong restriction, but there are good reasons for it:

- Your data is very important, so it's best to be explicit about it.

- A single data frame is also easier to save than a multitude of vectors, which means it's easier to reproduce your results or send your data to someone else.
- It enforces a clean separation of concerns: ggplot2 turns data frames into visualisations. Other packages can make data frames in the right format (learn more about that in Section 11.4).

The data on each layer doesn't need to be the same, and it's often useful to combine multiple datasets in a single plot. To illustrate that idea I'm going to generate two new datasets related to the mpg dataset. First I'll fit a loess model and generate predictions from it. (This is what `geom_smooth()` does behind the scenes)

```
mod <- loess(hwy ~ displ, data = mpg)
grid <- data_frame(displ = seq(min(mpg$displ), max(mpg$displ), length = 50))
grid$hwy <- predict(mod, newdata = grid)
```

```
grid
#> # A tibble: 50 x 2
#>   displ   hwy
#>   <dbl> <dbl>
#> 1  1.60  33.1
#> 2  1.71  32.2
#> 3  1.82  31.3
#> 4  1.93  30.4
#> 5  2.04  29.6
#> 6  2.15  28.8
#> # ... with 44 more rows
```

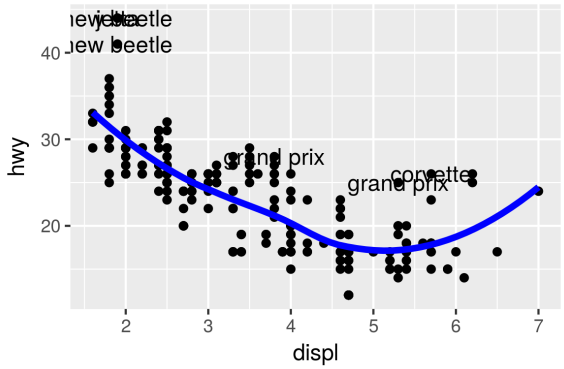
Next, I'll isolate observations that are particularly far away from their predicted values:

```
std_resid <- resid(mod) / mod$s
outlier <- filter(mpg, abs(std_resid) > 2)
outlier
#> # A tibble: 6 x 11
#>   manufacturer      model displ  year   cyl      trans   drv   cty
#>          <chr>      <chr> <dbl> <int> <int>      <chr> <chr> <int>
#> 1    chevrolet   corvette   5.7  1999     8 manual(m6)     r    16
#> 2      pontiac grand prix   3.8  2008     6   auto(l4)     f    18
#> 3      pontiac grand prix   5.3  2008     8   auto(s4)     f    16
#> 4   volkswagen      jetta   1.9  1999     4 manual(m5)     f    33
#> 5   volkswagen new beetle   1.9  1999     4 manual(m5)     f    35
#> 6   volkswagen new beetle   1.9  1999     4   auto(l4)     f    29
#> # ... with 3 more variables: hwy <int>, fl <chr>, class <chr>
```

I've generated these datasets because it's common to enhance the display of raw data with a statistical summary and some annotations. With these

new datasets, I can improve our initial scatterplot by overlaying a smoothed line, and labelling the outlying points:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_line(data = grid, colour = "blue", size = 1.5) +
  geom_text(data = outlier, aes(label = model))
```



(The labels aren't particularly easy to read, but you can fix that with some manual tweaking.)

Note that you need the explicit `data =` in the layers, but not in the call to `ggplot()`. That's because the argument order is different. This is a little inconsistent, but it reduces typing for the common case where you specify the data once in `ggplot()` and modify aesthetics in each layer.

In this example, every layer uses a different dataset. We could define the same plot in another way, omitting the default dataset, and specifying a dataset for each layer:

```
ggplot(mapping = aes(displ, hwy)) +
  geom_point(data = mpg) +
  geom_line(data = grid) +
  geom_text(data = outlier, aes(label = model))
```

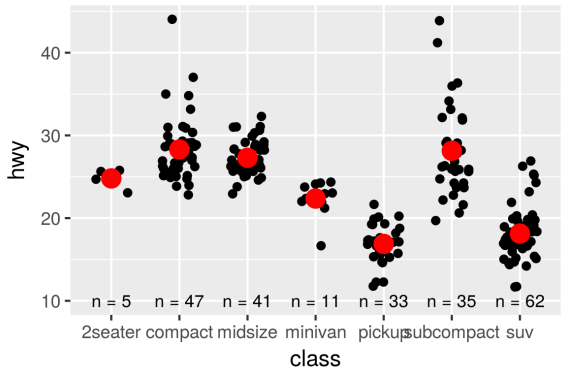
I don't particularly like this style in this example because it makes it less clear what the primary dataset is (and because of the way that the arguments to `ggplot()` are ordered, it actually requires more keystrokes). However, you may prefer it in cases where there isn't a clear primary dataset, or where the aesthetics also vary from layer to layer.

5.3.1 Exercises

1. The first two arguments to ggplot are `data` and `mapping`. The first two arguments to all layer functions are `mapping` and `data`. Why does the order of the arguments differ? (Hint: think about what you set most commonly.)
2. The following code uses dplyr to generate some summary statistics about each class of car (you'll learn how it works in Chapter 10).

```
library(dplyr)
class <- mpg %>%
  group_by(class) %>%
  summarise(n = n(), hwy = mean(hwy))
```

   Use the data to recreate this plot:



5.4 Aesthetic mappings

The aesthetic mappings, defined with `aes()`, describe how variables are mapped to visual properties or **aesthetics**. `aes()` takes a sequence of aesthetic-variable pairs like this:

```
aes(x = displ, y = hwy, colour = class)
```

(If you're American, you can use *color*, and behind the scenes ggplot2 will correct your spelling ;)

Here we map x-position to `displ`, y-position to `hwy`, and colour to `class`. The names for the first two arguments can be omitted, in which case they correspond to the x and y variables. That makes this specification equivalent to the one above:

```
aes(displ, hwy, colour = class)
```

While you can do data manipulation in `aes()`, e.g. `aes(log(carat), log(price))`, it's best to only do simple calculations. It's better to move complex transformations out of the `aes()` call and into an explicit `dplyr::mutate()` call, as you'll learn about in Section 10.3. This makes it easier to check your work and it's often faster because you need only do the transformation once, not every time the plot is drawn.

Never refer to a variable with `$` (e.g., `diamonds$carat`) in `aes()`. This breaks containment, so that the plot no longer contains everything it needs, and causes problems if ggplot2 changes the order of the rows, as it does when facetting.

5.4.1 Specifying the aesthetics in the plot vs. in the layers

Aesthetic mappings can be supplied in the initial `ggplot()` call, in individual layers, or in some combination of both. All of these calls create the same plot specification:

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

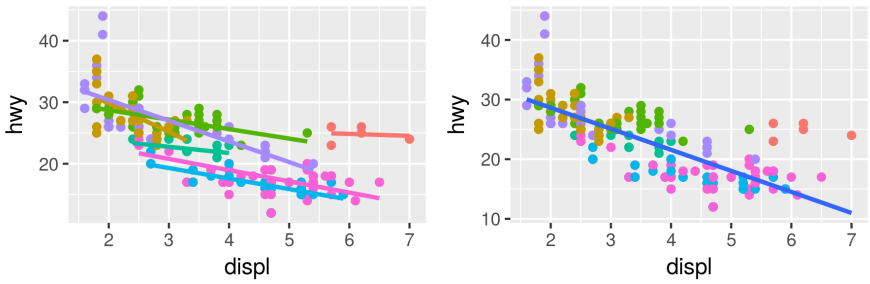
Within each layer, you can add, override, or remove mappings:

Operation	Layer aesthetics	Result
Add	`aes(colour = cyl)`	`aes(mpg, wt, colour = cyl)`
Override	`aes(y = disp)`	`aes(mpg, disp)`
Remove	`aes(y = NULL)`	`aes(mpg)`

If you only have one layer in the plot, the way you specify aesthetics doesn't make any difference. However, the distinction is important when you start adding additional layers. These two plots are both valid and interesting, but focus on quite different aspects of the data:

```
ggplot(mpg, aes(displ, hwy, colour = class)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE) +
  theme(legend.position = "none")
```

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  geom_smooth(method = "lm", se = FALSE) +
  theme(legend.position = "none")
```



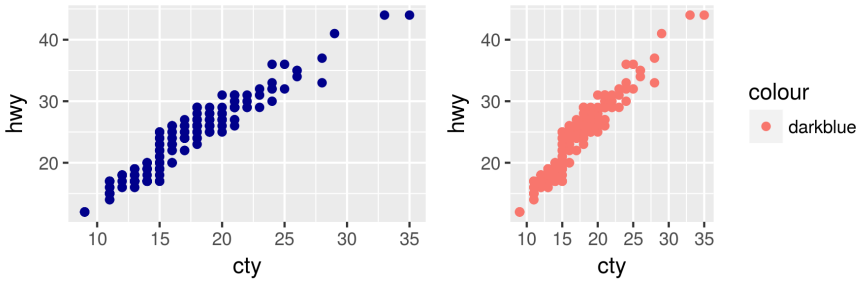
Generally, you want to set up the mappings to illuminate the structure underlying the graphic and minimise typing. It may take some time before the best approach is immediately obvious, so if you've iterated your way to a complex graphic, it may be worthwhile to rewrite it to make the structure more clear.

5.4.2 Setting vs. mapping

Instead of mapping an aesthetic property to a variable, you can set it to a *single* value by specifying it in the layer parameters. We **map** an aesthetic to a variable (e.g., `aes(colour = cut)`) or **set** it to a constant (e.g., `colour = "red"`). If you want appearance to be governed by a variable, put the specification inside `aes()`; if you want override the default size or colour, put the value outside of `aes()`.

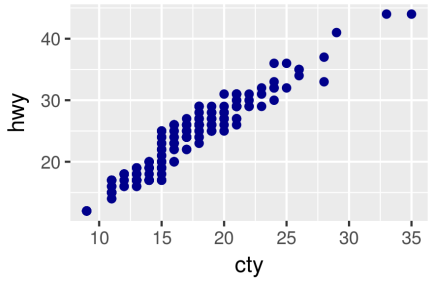
The following plots are created with similar code, but have rather different outputs. The second plot **maps** (not sets) the colour to the value 'darkblue'. This effectively creates a new variable containing only the value 'darkblue' and then scales it with a colour scale. Because this value is discrete, the default colour scale uses evenly spaced colours on the colour wheel, and since there is only one value this colour is pinkish.

```
ggplot(mpg, aes(cty, hwy)) +
  geom_point(colour = "darkblue")

ggplot(mpg, aes(cty, hwy)) +
  geom_point(aes(colour = "darkblue"))
```



A third approach is to map the value, but override the default scale:

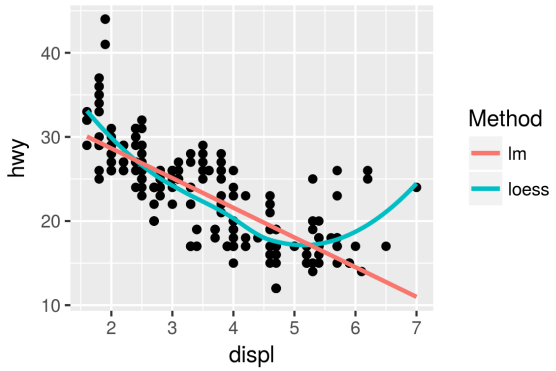
```
ggplot(mpg, aes(cty, hwy)) +
  geom_point(aes(colour = "darkblue")) +
  scale_colour_identity()
```



This is most useful if you always have a column that already contains colours. You'll learn more about that in Section 6.6.4.

It's sometimes useful to map aesthetics to constants. For example, if you want to display multiple layers with varying parameters, you can "name" each layer:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(aes(colour = "loess"), method = "loess", se = FALSE) +
  geom_smooth(aes(colour = "lm"), method = "lm", se = FALSE) +
  labs(colour = "Method")
```



5.4.3 Exercises

1. Simplify the following plot specifications:

```
ggplot(mpg) +
  geom_point(aes(mpg$disp, mpg$hwy))

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

ggplot(diamonds, aes(carat, price)) +
  geom_point(aes(log(brainwt), log(bodywt)), data = msleep)
```

2. What does the following code do? Does it work? Does it make sense? Why/why not?

```
ggplot(mpg) +
  geom_point(aes(class, cty)) +
  geom_boxplot(aes(trans, hwy))
```

3. What happens if you try to use a continuous variable on the x axis in one layer, and a categorical variable in another layer? What happens if you do it in the opposite order?

5.5 Geoms

Geometric objects, or **geoms** for short, perform the actual rendering of the layer, controlling the type of plot that you create. For example, using a point geom will create a scatterplot, while using a line geom will create a line plot.

- Graphical primitives:
  - `geom_blank()`: display nothing. Most useful for adjusting axes limits using data.
  - `geom_point()`: points.
  - `geom_path()`: paths.
  - `geom_ribbon()`: ribbons, a path with vertical thickness.
  - `geom_segment()`: a line segment, specified by start and end position.
  - `geom_rect()`: rectangles.
  - `geom_polyon()`: filled polygons.
  - `geom_text()`: text.
- One variable:
  - Discrete:
    - `geom_bar()`: display distribution of discrete variable.
  - Continuous
    - `geom_histogram()`: bin and count continuous variable, display with bars.
    - `geom_density()`: smoothed density estimate.
    - `geom_dotplot()`: stack individual points into a dot plot.
    - `geom_freqpoly()`: bin and count continuous variable, display with lines.
- Two variables:
  - Both continuous:
    - `geom_point()`: scatterplot.
    - `geom_quantile()`: smoothed quantile regression.
    - `geom_rug()`: marginal rug plots.
    - `geom_smooth()`: smoothed line of best fit.
    - `geom_text()`: text labels.
  - Show distribution:
    - `geom_bin2d()`: bin into rectangles and count.
    - `geom_density2d()`: smoothed 2d density estimate.
    - `geom_hex()`: bin into hexagons and count.
  - At least one discrete:
    - `geom_count()`: count number of point at distinct locations
    - `geom_jitter()`: randomly jitter overlapping points.
  - One continuous, one discrete:
    - `geom_bar(stat = "identity")`: a bar chart of precomputed summaries.

    - `geom_boxplot()`: boxplots.
    - `geom_violin()`: show density of values in each group.
  - One time, one continuous
    - `geom_area()`: area plot.
    - `geom_line()`: line plot.
    - `geom_step()`: step plot.
  - Display uncertainty:
    - `geom_crossbar()`: vertical bar with center.
    - `geom_errorbar()`: error bars.
    - `geom_linerange()`: vertical line.
    - `geom_pointrange()`: vertical line with center.
  - Spatial
    - `geom_map()`: fast version of `geom_polygon()` for map data.

- Three variables:

  - `geom_contour()`: contours.
  - `geom_tile()`: tile the plane with rectangles.
  - `geom_raster()`: fast version of `geom_tile()` for equal sized tiles.

Each geom has a set of aesthetics that it understands, some of which *must* be provided. For example, the point geoms requires x and y position, and understands colour, size and shape aesthetics. A bar requires height (`ymax`), and understands width, border colour and fill colour. Each geom lists its aesthetics in the documentation.

Some geoms differ primarily in the way that they are parameterised. For example, you can draw a square in three ways:

- By giving `geom_tile()` the location (`x` and `y`) and dimensions (`width` and `height`).
- By giving `geom_rect()` top (`ymax`), bottom (`ymin`), left (`xmin`) and right (`xmax`) positions.
- By giving `geom_polygon()` a four row data frame with the `x` and `y` positions of each corner.

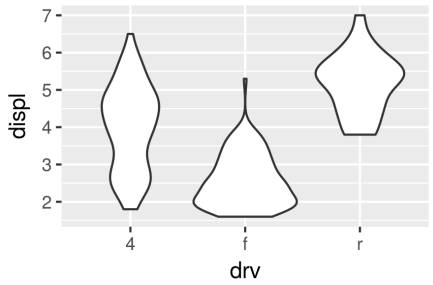
Other related geoms are:

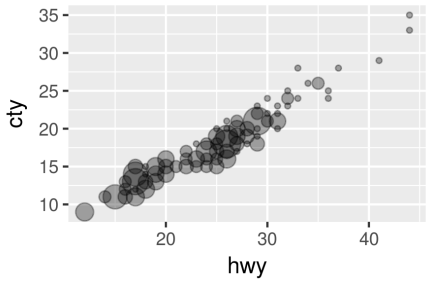
- `geom_segment()` and `geom_line()`
- `geom_area()` and `geom_ribbon()`.

If alternative parameterisations are available, picking the right one for your data will usually make it much easier to draw the plot you want.

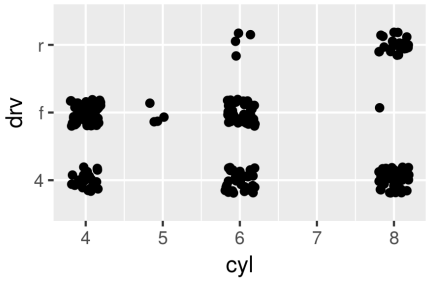
5.5.1 Exercises

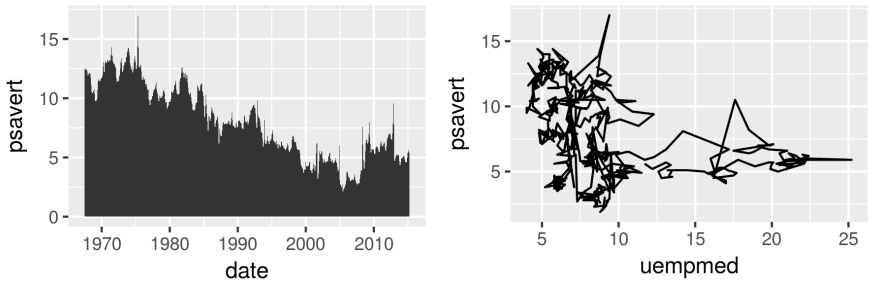
1. Download and print out the ggplot2 cheatsheet from `http://www.rstudio.com/resources/cheatsheets/` so you have a handy visual reference for all the geoms.
2. Look at the documentation for the graphical primitive geoms. Which aesthetics do they use? How can you summarise them in a compact form?
3. What's the best way to master an unfamiliar geom? List three resources to help you get started.
4. For each of the plots below, identify the geom used to draw it.






```
#> Warning: Computation failed in `stat_binhex()`:
#> Package `hexbin` required for `stat_binhex`.
#> Please install and try again.
```





5. For each of the following problems, suggest a useful geom:

   - Display how a variable has changed over time.
   - Show the detailed distribution of a single variable.
   - Focus attention on the overall trend in a large dataset.
   - Draw a map.
   - Label outlying points.

5.6 Stats

A statistical transformation, or **stat**, transforms the data, typically by summarising it in some manner. For example, a useful stat is the smoother, which calculates the smoothed mean of y, conditional on x. You've already used many of ggplot2's stats because they're used behind the scenes to generate many important geoms:

- `stat_bin()`: `geom_bar()`, `geom_freqpoly()`, `geom_histogram()`
- `stat_bin2d()`: `geom_bin2d()`
- `stat_bindot()`: `geom_dotplot()`
- `stat_binhex()`: `geom_hex()`
- `stat_boxplot()`: `geom_boxplot()`
- `stat_contour()`: `geom_contour()`
- `stat_quantile()`: `geom_quantile()`
- `stat_smooth()`: `geom_smooth()`
- `stat_sum()`: `geom_count()`

You'll rarely call these functions directly, but they are useful to know about because their documentation often provides more detail about the corresponding statistical transformation.

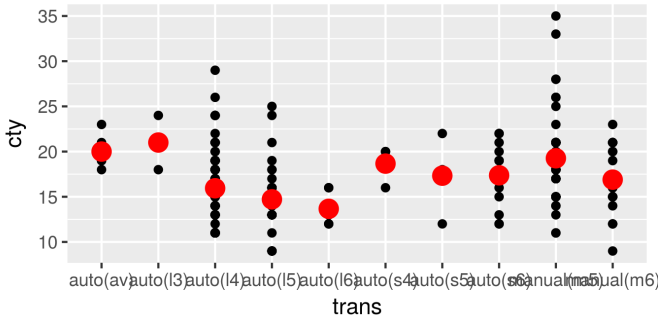
Other stats can't be created with a `geom_` function:

- `stat_ecdf()`: compute a empirical cumulative distribution plot.
- `stat_function()`: compute y values from a function of x values.
- `stat_summary()`: summarise y values at distinct x values.

- `stat_summary2d()`, `stat_summary_hex()`: summarise binned values.
- `stat_qq()`: perform calculations for a quantile-quantile plot.
- `stat_spoke()`: convert angle and radius to position.
- `stat_unique()`: remove duplicated rows.

There are two ways to use these functions. You can either add a `stat_()` function and override the default geom, or add a `geom_()` function and override the default stat:

```
ggplot(mpg, aes(trans, cty)) +
  geom_point() +
  stat_summary(geom = "point", fun.y = "mean", colour = "red", size = 4)

ggplot(mpg, aes(trans, cty)) +
  geom_point() +
  geom_point(stat = "summary", fun.y = "mean", colour = "red", size = 4)
```



I think it's best to use the second form because it makes it more clear that you're displaying a summary, not the raw data.

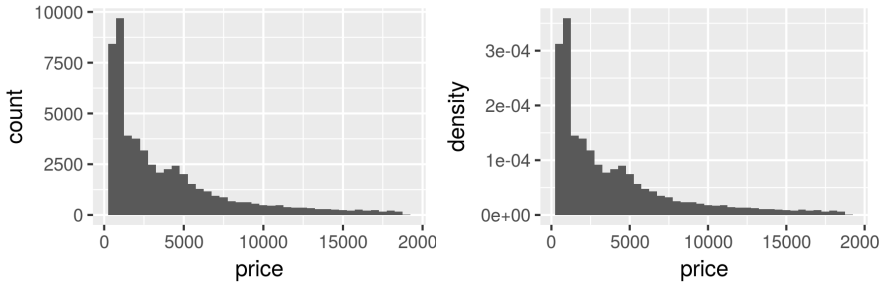
5.6.1 Generated variables

Internally, a stat takes a data frame as input and returns a data frame as output, and so a stat can add new variables to the original dataset. It is possible to map aesthetics to these new variables. For example, `stat_bin`, the statistic used to make histograms, produces the following variables:

- `count`, the number of observations in each bin
- `density`, the density of observations in each bin (percentage of total / bar width)
- `x`, the centre of the bin

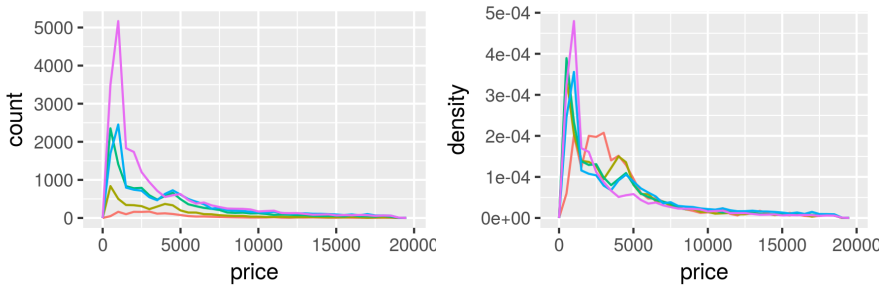
These generated variables can be used instead of the variables present in the original dataset. For example, the default histogram geom assigns the height of the bars to the number of observations (`count`), but if you'd prefer a more traditional histogram, you can use the density (`density`). To refer to a generated variable like density, "`..`" must surround the name. This prevents confusion in case the original dataset includes a variable with the same name as a generated variable, and it makes it clear to any later reader of the code that this variable was generated by a stat. Each statistic lists the variables that it creates in its documentation. Compare the y-axes on these two plots:

```
ggplot(diamonds, aes(price)) +
  geom_histogram(binwidth = 500)
ggplot(diamonds, aes(price)) +
  geom_histogram(aes(y = ..density..), binwidth = 500)
```



This technique is particularly useful when you want to compare the distribution of multiple groups that have very different sizes. For example, it's hard to compare the distribution of `price` within `cut` because some groups are quite small. It's easier to compare if we standardise each group to take up the same area:

```
ggplot(diamonds, aes(price, colour = cut)) +
  geom_freqpoly(binwidth = 500) +
  theme(legend.position = "none")

ggplot(diamonds, aes(price, colour = cut)) +
  geom_freqpoly(aes(y = ..density..), binwidth = 500) +
  theme(legend.position = "none")
```



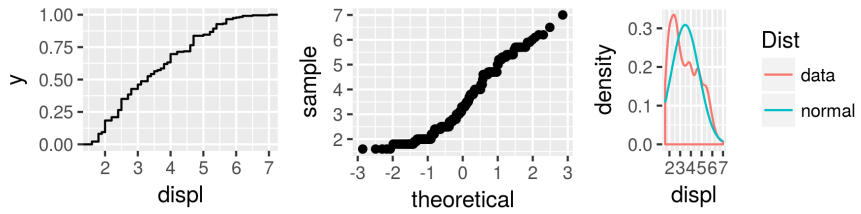
The result of this plot is rather surprising: low quality diamonds seem to be more expensive on average. We'll come back to this result in Section 11.2.

5.6.2 Exercises

1. The code below creates a similar dataset to `stat_smooth()`. Use the appropriate geoms to mimic the default `geom_smooth()` display.

```
mod <- loess(hwy ~ displ, data = mpg)
smoothed <- data.frame(displ = seq(1.6, 7, length = 50))
pred <- predict(mod, newdata = smoothed, se = TRUE)
smoothed$hwy <- pred$fit
smoothed$hwy_lwr <- pred$fit - 1.96 * pred$se.fit
smoothed$hwy_upr <- pred$fit + 1.96 * pred$se.fit
```

2. What stats were used to create the following plots?



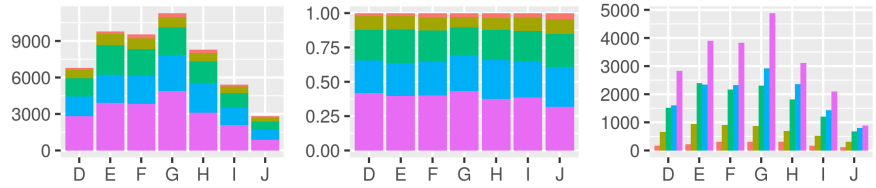
3. Read the help for `stat_sum()` then use `geom_count()` to create a plot that shows the proportion of cars that have each combination of `drv` and `trans`.

5.7 Position adjustments

Position adjustments apply minor tweaks to the position of elements within a layer. Three adjustments apply primarily to bars:

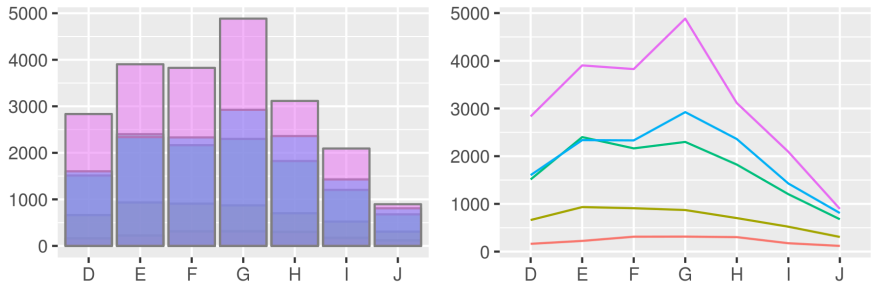
- `position_stack()`: stack overlapping bars (or areas) on top of each other.
- `position_fill()`: stack overlapping bars, scaling so the top is always at 1.
- `position_dodge()`: place overlapping bars (or boxplots) side-by-side.

```
dplot <- ggplot(diamonds, aes(color, fill = cut)) +
  xlab(NULL) + ylab(NULL) + theme(legend.position = "none")
# position stack is the default for bars, so `geom_bar()`
# is equivalent to `geom_bar(position = "stack")`.
dplot + geom_bar()
dplot + geom_bar(position = "fill")
dplot + geom_bar(position = "dodge")
```



There's also a position adjustment that does nothing: `position_identity()`. The identity position adjustment isn't useful for bars, because each bar obscures the bars behind, but there are many geoms that don't need adjusting, like lines:

```
dplot + geom_bar(position = "identity", alpha = 1 / 2, colour = "grey50")

ggplot(diamonds, aes(color, colour = cut)) +
  geom_line(aes(group = cut), stat = "count") +
  xlab(NULL) + ylab(NULL) +
  theme(legend.position = "none")
```

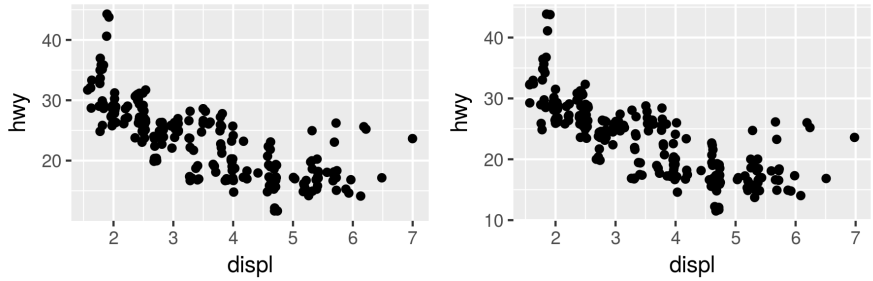


There are three position adjustments that are primarily useful for points:

- `position_nudge()`: move points by a fixed offset.
- `position_jitter()`: add a little random noise to every position.
- `position_jitterdodge()`: dodge points within groups, then add a little random noise.

Note that the way you pass parameters to position adjustments differs to stats and geoms. Instead of including additional arguments in ..., you construct a position adjustment object, supplying additional arguments in the call:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(position = "jitter")
ggplot(mpg, aes(displ, hwy)) +
  geom_point(position = position_jitter(width = 0.05, height = 0.5))
```



This is rather verbose, so `geom_jitter()` provides a convenient shortcut:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_jitter(width = 0.05, height = 0.5)
```

Continuous data typically doesn't overlap exactly, and when it does (because of high data density) minor adjustments, like jittering, are often insufficient to fix the problem. For this reason, position adjustments are generally most useful for discrete data.

5.7.1 Exercises

1. When might you use `position_nudge()`? Read the documentation.
2. Many position adjustments can only be used with a few geoms. For example, you can't stack boxplots or errors bars. Why not? What properties must a geom possess in order to be stackable? What properties must it possess to be dodgeable?
3. Why might you use `geom_jitter()` instead of `geom_count()`? What are the advantages and disadvantages of each technique?
4. When might you use a stacked area plot? What are the advantages and disadvantages compared to a line plot?

Chapter 6
Scales, axes and legends

6.1 Introduction

Scales control the mapping from data to aesthetics. They take your data and turn it into something that you can see, like size, colour, position or shape. Scales also provide the tools that let you read the plot: the axes and legends. Formally, each scale is a function from a region in data space (the domain of the scale) to a region in aesthetic space (the range of the scale). The axis or legend is the inverse function: it allows you to convert visual properties back to data.

You can generate many plots without knowing how scales work, but understanding scales and learning how to manipulate them will give you much more control. The basics of working with scales is described in Section 6.2. Section 6.3 discusses the common parameters that control the axes and legends. Legends are particularly complicated so have an additional set of options as described in Section 6.4. Section 6.5 shows how to use limits to both zoom into interesting parts of a plot, and to ensure that multiple plots have matching legends and axes. Section 6.6 gives an overview of the different types of scales available in ggplot2, which can be roughly divided into four categories: continuous position scales, colour scales, manual scales and identity scales.

6.2 Modifying scales

A scale is required for every aesthetic used on the plot. When you write:

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

What actually happens is this:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  scale_x_continuous() +
  scale_y_continuous() +
  scale_colour_discrete()
```

Default scales are named according to the aesthetic and the variable type: `scale_y_continuous()`, `scale_colour_discrete()`, etc.

It would be tedious to manually add a scale every time you used a new aesthetic, so ggplot2 does it for you. But if you want to override the defaults, you'll need to add the scale yourself, like this:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  scale_x_continuous("A really awesome x axis label") +
  scale_y_continuous("An amazingly great y axis label")
```

The use of `+` to "add" scales to a plot is a little misleading. When you `+` a scale, you're not actually adding it to the plot, but overriding the existing scale. This means that the following two specifications are equivalent:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_x_continuous("Label 1") +
  scale_x_continuous("Label 2")
#> Scale for 'x' is already present. Adding another scale for 'x',
#> which will replace the existing scale.

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_x_continuous("Label 2")
```

Note the message: if you see this in your own code, you need to reorganise your code specification to only add a single scale.

You can also use a different scale altogether:

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = class)) +
  scale_x_sqrt() +
  scale_colour_brewer()
```

You've probably already figured out the naming scheme for scales, but to be concrete, it's made up of three pieces separated by "_":

1. `scale`
2. The name of the aesthetic (e.g., `colour`, `shape` or `x`)
3. The name of the scale (e.g., `continuous`, `discrete`, `brewer`).

6.2.1 Exercises

1. What happens if you pair a discrete variable to a continuous scale? What happens if you pair a continuous variable to a discrete scale?
2. Simplify the following plot specifications to make them easier to understand.

```
ggplot(mpg, aes(displ)) +
  scale_y_continuous("Highway mpg") +
  scale_x_continuous() +
  geom_point(aes(y = hwy))

ggplot(mpg, aes(y = displ, x = class)) +
  scale_y_continuous("Displacement (l)") +
  scale_x_discrete("Car type") +
  scale_x_discrete("Type of car") +
  scale_colour_discrete() +
  geom_point(aes(colour = drv)) +
  scale_colour_discrete("Drive\ntrain")
```

6.3 Guides: legends and axes

The component of a scale that you're most likely to want to modify is the **guide**, the axis or legend associated with the scale. Guides allow you to read observations from the plot and map them back to their original values. In ggplot2, guides are produced automatically based on the layers in your plot. This is very different to base R graphics, where you are responsible for drawing the legends by hand. In ggplot2, you don't directly control the legend; instead you set up the data so that there's a clear mapping between data and aesthetics, and a legend is generated for you automatically. This can be frustrating when you first start using ggplot2, but once you get the hang of it, you'll find that it saves you time, and there is little you cannot do. If you're struggling to get the legend you want, it's likely that your data is in the wrong form. Read Chapter 9 to find out the right form.

You might find it surprising that axes and legends are the same type of thing, but while they look very different there are many natural correspondences between the two, as shown in table below and in Figure 6.1.

Axis	Legend	Argument name
Label	Title	`name`
Ticks & grid line	Key	`breaks`
Tick label	Key label	`labels`

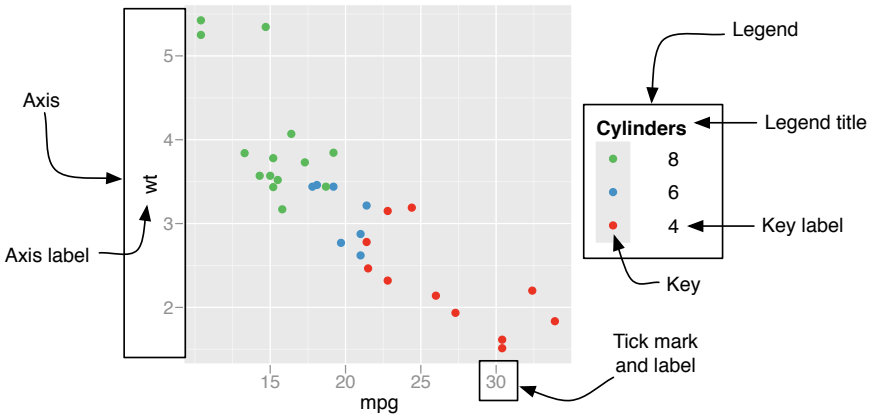


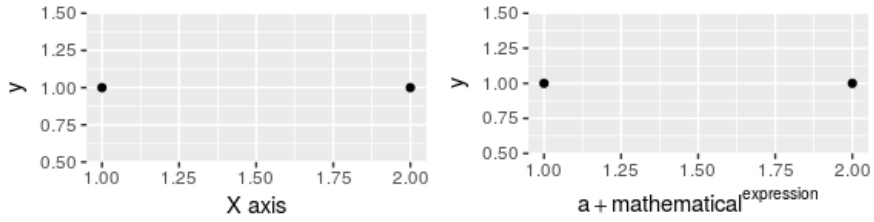
Fig. 6.1 Axis and legend components

The following sections covers each of the `name`, `breaks` and `labels` arguments in more detail.

6.3.1 Scale title

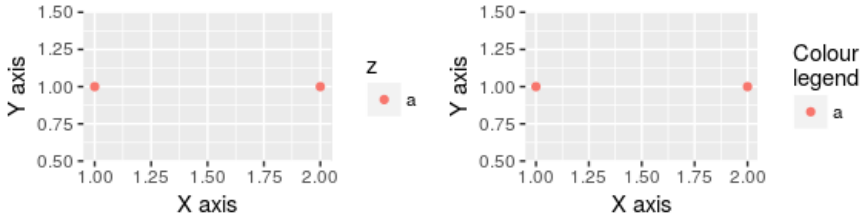
The first argument to the scale function, `name`, is the axes/legend title. You can supply text strings (using \n for line breaks) or mathematical expressions in `quote()` (as described in `?plotmath`):

```
df <- data.frame(x = 1:2, y = 1, z = "a")
p <- ggplot(df, aes(x, y)) + geom_point()
p + scale_x_continuous("X axis")
p + scale_x_continuous(quote(a + mathematical ^ expression))
```



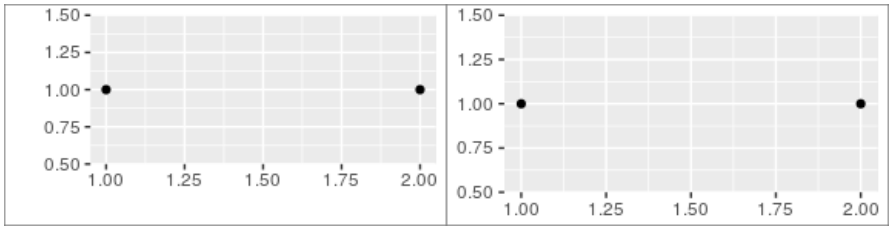
Because tweaking these labels is such a common task, there are three helpers that save you some typing: `xlab()`, `ylab()` and `labs()`:

```
p <- ggplot(df, aes(x, y)) + geom_point(aes(colour = z))
p +
  xlab("X axis") +
  ylab("Y axis")
p + labs(x = "X axis", y = "Y axis", colour = "Colour\nlegend")
```



There are two ways to remove the axis label. Setting it to "" omits the label, but still allocates space; NULL removes the label and its space. Look closely at the left and bottom borders of the following two plots. I've drawn a grey rectangle around the plot to make it easier to see the difference.

```
p <- ggplot(df, aes(x, y)) +
  geom_point() +
  theme(plot.background = element_rect(colour = "grey50"))
p + labs(x = "",  y = "")
p + labs(x = NULL, y = NULL)
```

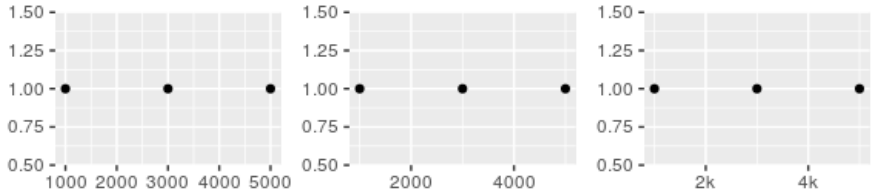


6.3.2 Breaks and labels

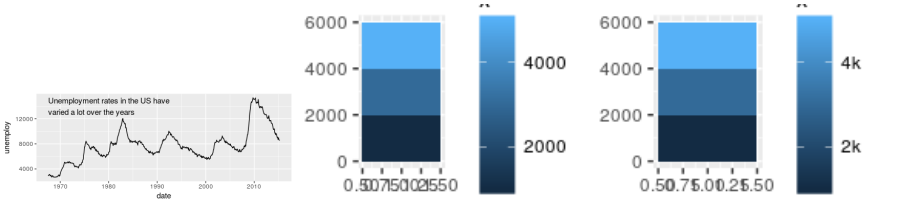
The breaks argument controls which values appear as tick marks on axes and keys on legends. Each break has an associated label, controlled by the labels argument. If you set labels, you must also set breaks; otherwise, if data changes, the breaks will no longer align with the labels.

The following code shows some basic examples for both axes and legends.

```
df <- data.frame(x = c(1, 3, 5) * 1000, y = 1)
axs <- ggplot(df, aes(x, y)) +
  geom_point() +
  labs(x = NULL, y = NULL)
axs
axs + scale_x_continuous(breaks = c(2000, 4000))
axs + scale_x_continuous(breaks = c(2000, 4000), labels = c("2k", "4k"))
```

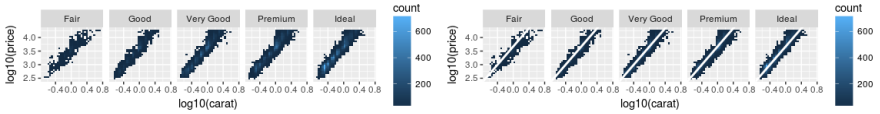



```
leg <- ggplot(df, aes(y, x, fill = x)) +
  geom_tile() +
  labs(x = NULL, y = NULL)
leg
leg + scale_fill_continuous(breaks = c(2000, 4000))
leg + scale_fill_continuous(breaks = c(2000, 4000), labels = c("2k", "4k"))
```



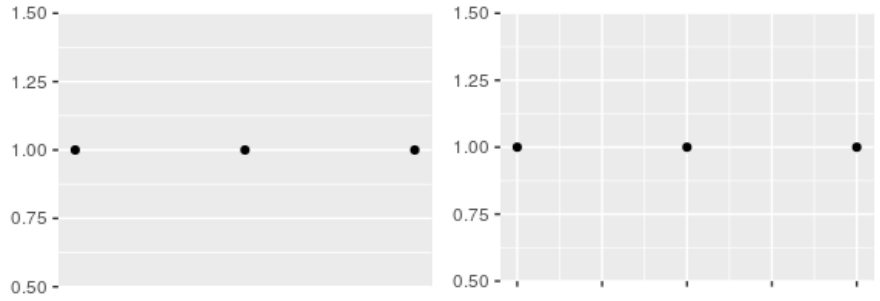
If you want to relabel the breaks in a categorical scale, you can use a named labels vector:

```
df2 <- data.frame(x = 1:3, y = c("a", "b", "c"))
ggplot(df2, aes(x, y)) +
  geom_point()
ggplot(df2, aes(x, y)) +
  geom_point() +
  scale_y_discrete(labels = c(a = "apple", b = "banana", c = "carrot"))
```

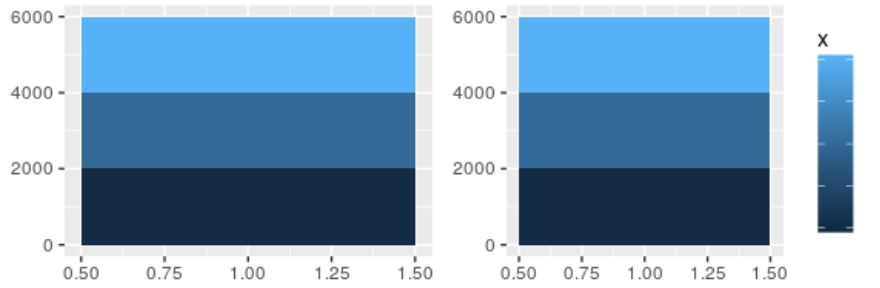


To suppress breaks (and for axes, grid lines) or labels, set them to `NULL`:

```
axs + scale_x_continuous(breaks = NULL)
axs + scale_x_continuous(labels = NULL)
```




```
leg + scale_fill_continuous(breaks = NULL)
leg + scale_fill_continuous(labels = NULL)
```



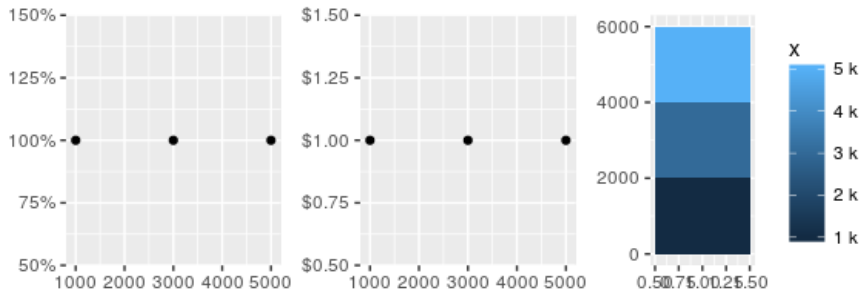
Additionally, you can supply a function to `breaks` or `labels`. The `breaks` function should have one argument, the limits (a numeric vector of length two), and should return a numeric vector of breaks. The `labels` function should accept a numeric vector of breaks and return a character vector of labels (the same length as the input). The scales package provides a number of useful labelling functions:

- `scales::comma_format()` adds commas to make it easier to read large numbers.

- `scales::unit_format(unit, scale)` adds a unit suffix, optionally scaling.
- `scales::dollar_format(prefix, suffix)` displays currency values, rounding to two decimal places and adding a prefix or suffix.
- `scales::wrap_format()` wraps long labels into multiple lines.

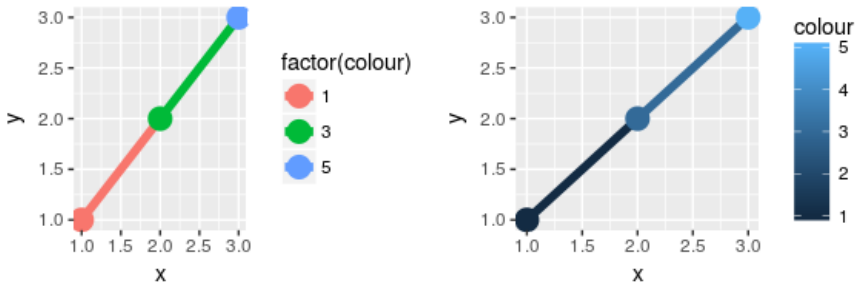
See the documentation of the scales package for more details.

```
axs + scale_y_continuous(labels = scales::percent_format())
axs + scale_y_continuous(labels = scales::dollar_format("$"))
leg + scale_fill_continuous(labels = scales::unit_format("k", 1e-3))
```



You can adjust the minor breaks (the faint grid lines that appear between the major grid lines) by supplying a numeric vector of positions to the `minor_breaks` argument. This is particularly useful for log scales:

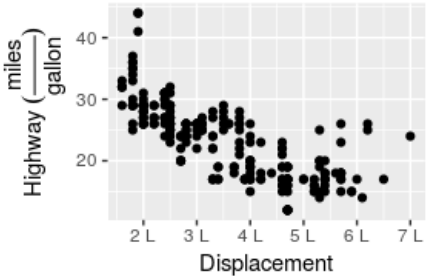
```
df <- data.frame(x = c(2, 3, 5, 10, 200, 3000), y = 1)
ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_log10()

mb <- as.numeric(1:10 %o% 10 ^ (0:4))
ggplot(df, aes(x, y)) +
  geom_point() +
  scale_x_log10(minor_breaks = log10(mb))
```



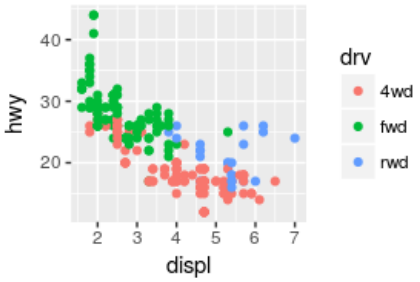
Note the use of `%o%` to quickly generate the multiplication table, and that the minor breaks must be supplied on the transformed scale.

6.3.3 Exercises

1. Recreate the following graphic:



   Adjust the y axis label so that the parentheses are the right size.
2. List the three different types of object you can supply to the `breaks` argument. How do `breaks` and `labels` differ?
3. Recreate the following plot:



4. What label function allows you to create mathematical expressions? What label function converts 1 to 1st, 2 to 2nd, and so on?
5. What are the three most important arguments that apply to both axes and legends? What do they do? Compare and contrast their operation for axes vs. legends.

6.4 Legends

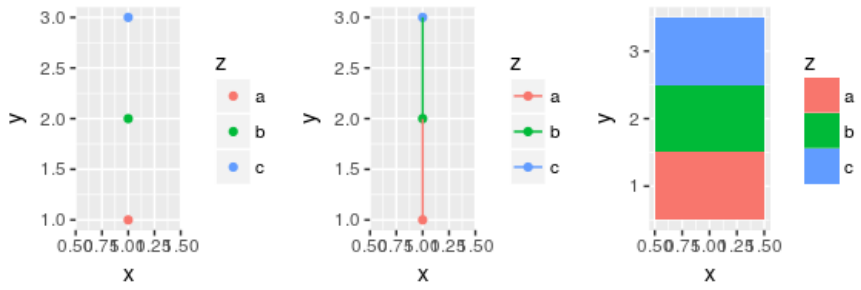
While the most important parameters are shared between axes and legends, there are some extra options that only apply to legends. Legends are more complicated than axes because:

1. A legend can display multiple aesthetics (e.g. colour and shape), from multiple layers, and the symbol displayed in a legend varies based on the geom used in the layer.
2. Axes always appear in the same place. Legends can appear in different places, so you need some global way of controlling them.
3. Legends have considerably more details that can be tweaked: should they be displayed vertically or horizontally? How many columns? How big should the keys be?

   The following sections describe the options that control these interactions.

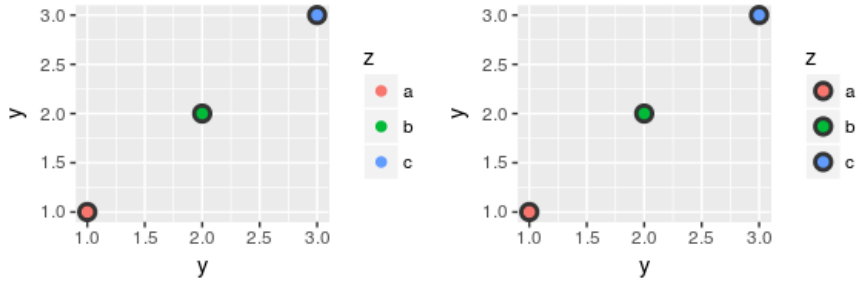
6.4.1 Layers and legends

A legend may need to draw symbols from multiple layers. For example, if you've mapped colour to both points and lines, the keys will show both points and lines. If you've mapped fill colour, you get a rectangle. Note the way the legend varies in the plots below:



By default, a layer will only appear if the corresponding aesthetic is mapped to a variable with `aes()`. You can override whether or not a layer appears in the legend with `show.legend`: `FALSE` to prevent a layer from ever appearing in the legend; `TRUE` forces it to appear when it otherwise wouldn't. Using `TRUE` can be useful in conjunction with the following trick to make points stand out:

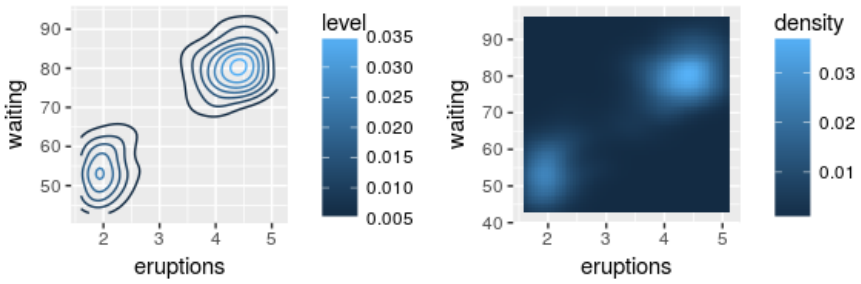
```
ggplot(df, aes(y, y)) +
  geom_point(size = 4, colour = "grey20") +
  geom_point(aes(colour = z), size = 2)
ggplot(df, aes(y, y)) +
  geom_point(size = 4, colour = "grey20", show.legend = TRUE) +
  geom_point(aes(colour = z), size = 2)
```



Sometimes you want the geoms in the legend to display differently to the geoms in the plot. This is particularly useful when you've used transparency or size to deal with moderate overplotting and also used colour in the plot. You can do this using the `override.aes` parameter of `guide_legend()`, which you'll learn more about shortly.

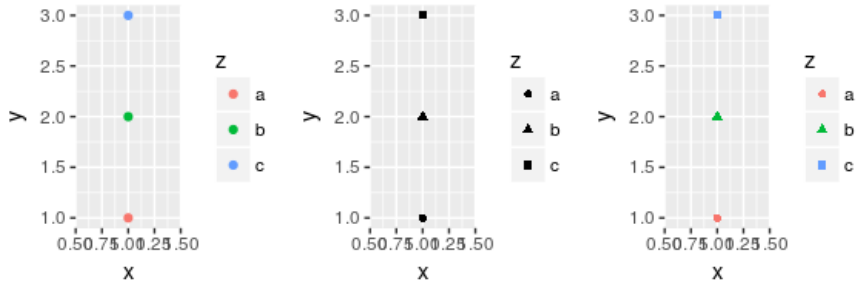
```
norm <- data.frame(x = rnorm(1000), y = rnorm(1000))
norm$z <- cut(norm$x, 3, labels = c("a", "b", "c"))
ggplot(norm, aes(x, y)) +
```

```
  geom_point(aes(colour = z), alpha = 0.1)
ggplot(norm, aes(x, y)) +
  geom_point(aes(colour = z), alpha = 0.1) +
  guides(colour = guide_legend(override.aes = list(alpha = 1)))
```



ggplot2 tries to use the fewest number of legends to accurately convey the aesthetics used in the plot. It does this by combining legends where the same variable is mapped to different aesthetics. The figure below shows how this works for points: if both colour and shape are mapped to the same variable, then only a single legend is necessary.

```
ggplot(df, aes(x, y)) + geom_point(aes(colour = z))
ggplot(df, aes(x, y)) + geom_point(aes(shape = z))
ggplot(df, aes(x, y)) + geom_point(aes(shape = z, colour = z))
```



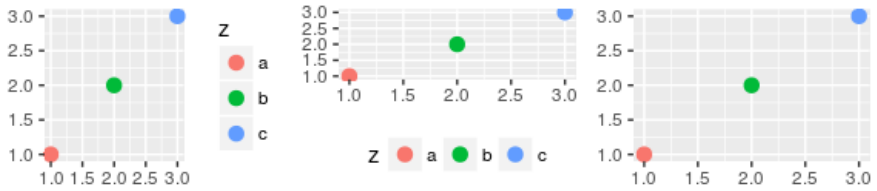
In order for legends to be merged, they must have the same `name`. So if you change the name of one of the scales, you'll need to change it for all of them.

6.4.2 Legend layout

A number of settings that affect the overall display of the legends are controlled through the theme system. You'll learn more about that in Section 8.2, but for now, all you need to know is that you modify theme settings with the `theme()` function.

The position and justification of legends are controlled by the theme setting `legend.position`, which takes values "right", "left", "top", "bottom", or "none" (no legend).

```
df <- data.frame(x = 1:3, y = 1:3, z = c("a", "b", "c"))
base <- ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z), size = 3) +
  xlab(NULL) +
  ylab(NULL)

base + theme(legend.position = "right") # the default
base + theme(legend.position = "bottom")
base + theme(legend.position = "none")
```



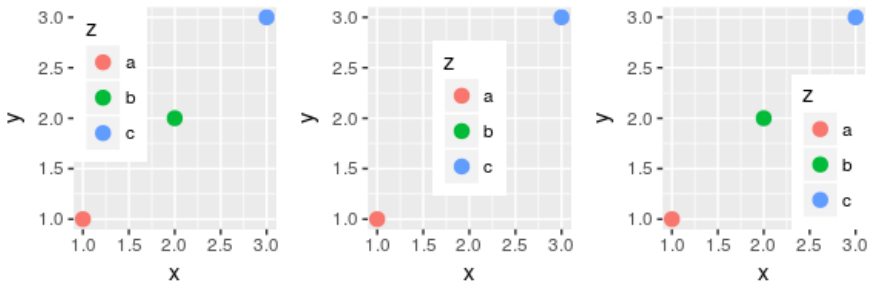
Switching between left/right and top/bottom modifies how the keys in each legend are laid out (horizontal or vertically), and how multiple legends are stacked (horizontal or vertically). If needed, you can adjust those options independently:

- `legend.direction`: layout of items in legends ("horizontal" or "vertical").
- `legend.box`: arrangement of multiple legends ("horizontal" or "vertical").
- `legend.box.just`: justification of each legend within the overall bounding box, when there are multiple legends ("top", "bottom", "left", or "right").

Alternatively, if there's a lot of blank space in your plot you might want to place the legend inside the plot. You can do this by setting `legend.position` to a numeric vector of length two. The numbers represent a relative location in the panel area: `c(0, 1)` is the top-left corner and `c(1, 0)` is the bottom-right corner. You control which corner of the legend the `legend.position` refers to with `legend.justification`, which is specified in a similar way. Unfortunately

positioning the legend exactly where you want it requires a lot of trial and error.

```
base <- ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z), size = 3)

base + theme(legend.position = c(0, 1), legend.justification = c(0, 1))
base + theme(legend.position = c(0.5, 0.5), legend.justification = c(0.5, 0.5))
base + theme(legend.position = c(1, 0), legend.justification = c(1, 0))
```



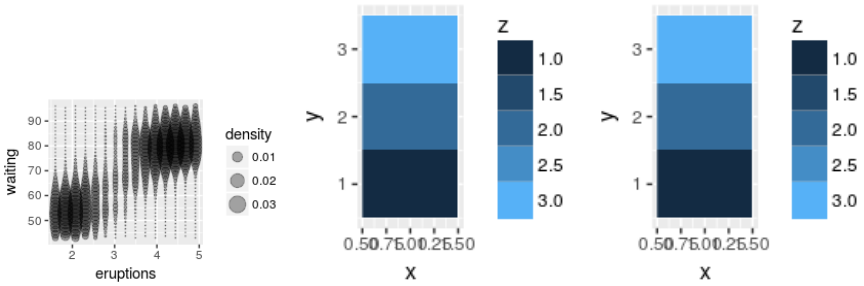
There's also a margin around the legends, which you can suppress with `legend.margin = unit(0, "mm")`.

6.4.3 Guide functions

The guide functions, `guide_colourbar()` and `guide_legend()`, offer additional control over the fine details of the legend. Legend guides can be used for any aesthetic (discrete or continuous) while the colour bar guide can only be used with continuous colour scales.

You can override the default guide using the `guide` argument of the corresponding scale function, or more conveniently, the `guides()` helper function. `guides()` works like `labs()`: you can override the default guide associated with each aesthetic.

```
df <- data.frame(x = 1, y = 1:3, z = 1:3)
base <- ggplot(df, aes(x, y)) + geom_raster(aes(fill = z))
base
base + scale_fill_continuous(guide = guide_legend())
base + guides(fill = guide_legend())
```



Both functions have numerous examples in their documentation help pages that illustrate all of their arguments. Most of the arguments to the guide function control the fine level details of the text colour, size, font etc. You'll learn about those in the themes chapter. Here I'll focus on the most important arguments.

6.4.3.1 `guide_legend()`

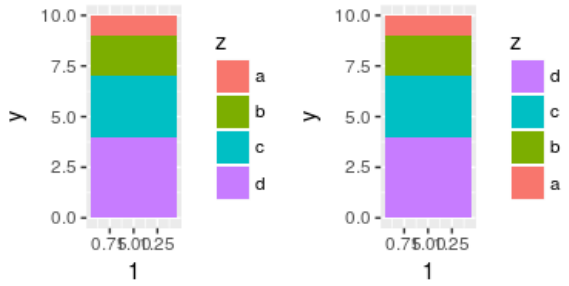
The legend guide displays individual keys in a table. The most useful options are:

- `nrow` or `ncol` which specify the dimensions of the table. `byrow` controls how the table is filled: `FALSE` fills it by column (the default), `TRUE` fills it by row.

```
df <- data.frame(x = 1, y = 1:4, z = letters[1:4])
# Base plot
p <- ggplot(df, aes(x, y)) + geom_raster(aes(fill = z))
p
p + guides(fill = guide_legend(ncol = 2))
p + guides(fill = guide_legend(ncol = 2, byrow = TRUE))
```

- `reverse` reverses the order of the keys. This is particularly useful when you have stacked bars because the default stacking and legend orders are different:

```
p <- ggplot(df, aes(1, y)) + geom_bar(stat = "identity", aes(fill = z))
p
p + guides(fill = guide_legend(reverse = TRUE))
```



- `override.aes`: override some of the aesthetic settings derived from each layer. This is useful if you want to make the elements in the legend more visually prominent. See discussion in Section 6.4.1.
- `keywidth` and `keyheight` (along with `default.unit`) allow you to specify the size of the keys. These are grid units, e.g. `unit(1, "cm")`.

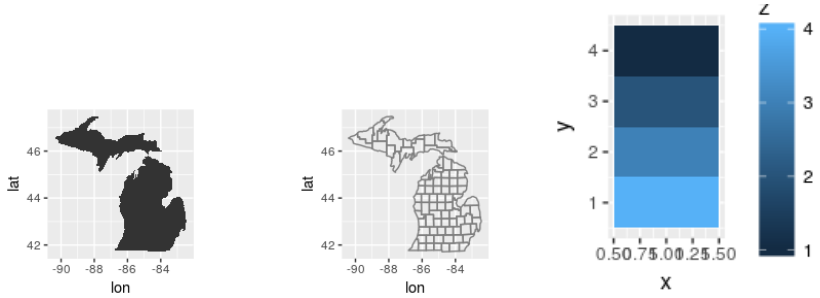
6.4.3.2 `guide_colourbar`

The colour bar guide is designed for continuous ranges of colors—as its name implies, it outputs a rectangle over which the color gradient varies. The most important arguments are:

- `barwidth` and `barheight` (along with `default.unit`) allow you to specify the size of the bar. These are grid units, e.g. `unit(1, "cm")`.
- `nbin` controls the number of slices. You may want to increase this from the default value of 20 if you draw a very long bar.
- `reverse` flips the colour bar to put the lowest values at the top.

These options are illustrated below:

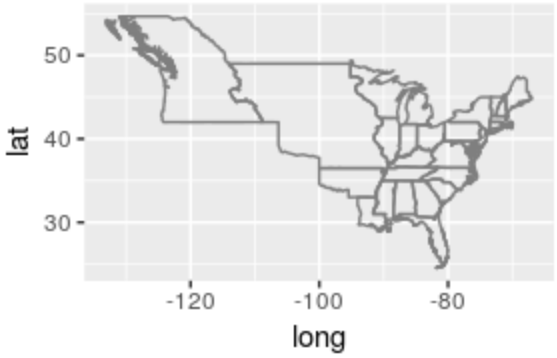
```
df <- data.frame(x = 1, y = 1:4, z = 4:1)
p <- ggplot(df, aes(x, y)) + geom_tile(aes(fill = z))

p
p + guides(fill = guide_colorbar(reverse = TRUE))
p + guides(fill = guide_colorbar(barheight = unit(4, "cm")))
```



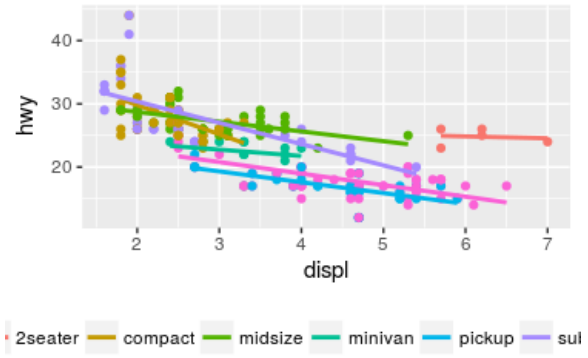
6.4.4 Exercises

1. How do you make legends appear to the left of the plot?
2. What's gone wrong with this plot? How could you fix it?

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(colour = drv, shape = drv)) +
  scale_colour_discrete("Drive train")
```



3. Can you recreate the code for this plot?



6.5 Limits

The limits, or domain, of a scale are usually derived from the range of the data. There are two reasons you might want to specify limits rather than relying on the data:

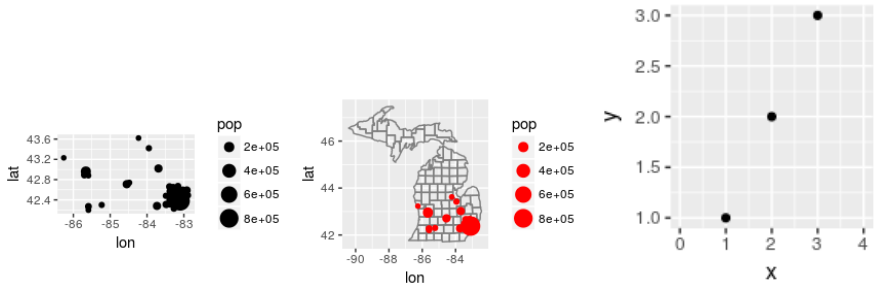
1. You want to make limits smaller than the range of the data to focus on an interesting area of the plot.
2. You want to make the limits larger than the range of the data because you want multiple plots to match up.

It's most natural to think about the limits of position scales: they map directly to the ranges of the axes. But limits also apply to scales that have legends, like colour, size, and shape. This is particularly important to realise if you want your colours to match up across multiple plots in your paper.

You can modify the limits using the `limits` parameter of the scale:

- For continuous scales, this should be a numeric vector of length two. If you only want to set the upper or lower limit, you can set the other value to `NA`.
- For discrete scales, this is a character vector which enumerates all possible values.

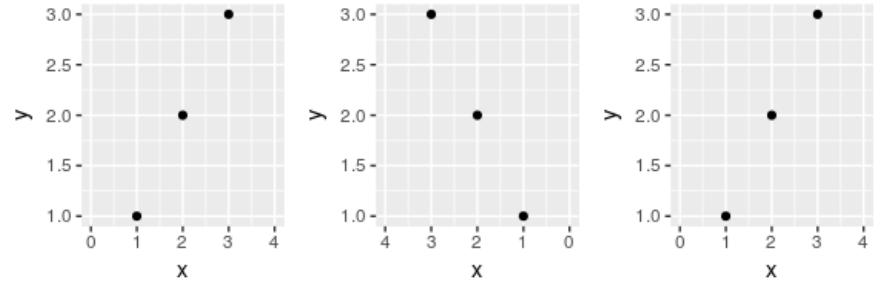
```
df <- data.frame(x = 1:3, y = 1:3)
base <- ggplot(df, aes(x, y)) + geom_point()

base
base + scale_x_continuous(limits = c(1.5, 2.5))
#> Warning: Removed 2 rows containing missing values (geom_point).
base + scale_x_continuous(limits = c(0, 4))
```



Because modifying the limits is such a common task, ggplot2 provides some helper to make this even easier: `xlim()`, `ylim()` and `lims()` These functions inspect their input and then create the appropriate scale, as follows:

- `xlim(10, 20)`: a continuous scale from 10 to 20
- `ylim(20, 10)`: a reversed continuous scale from 20 to 10
- `xlim("a", "b", "c")`: a discrete scale
- `xlim(as.Date(c("2008-05-01", "2008-08-01")))`: a date scale from May 1 to August 1 2008.

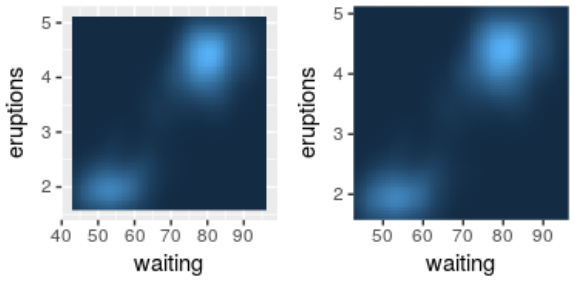
```
base + xlim(0, 4)
base + xlim(4, 0)
base + lims(x = c(0, 4))
```



If you have eagle eyes, you'll have noticed that the range of the axes actually extends a little bit past the limits that you've specified. This ensures that the data does not overlap the axes. To eliminate this space, set `expand = c(0, 0)`. This is useful in conjunction with `geom_raster()`:

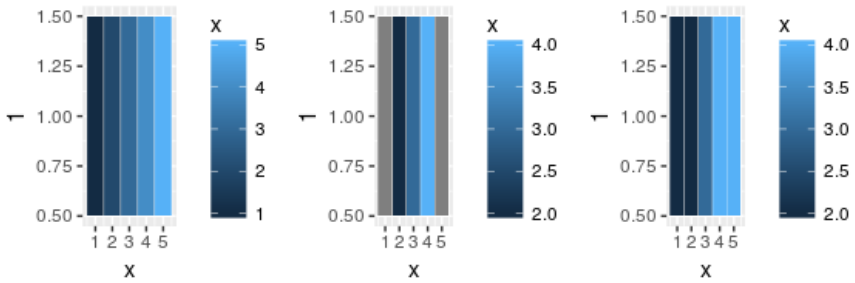
```
ggplot(faithfuld, aes(waiting, eruptions)) +
  geom_raster(aes(fill = density)) +
  theme(legend.position = "none")
ggplot(faithfuld, aes(waiting, eruptions)) +
```

```
  geom_raster(aes(fill = density)) +
  scale_x_continuous(expand = c(0,0)) +
  scale_y_continuous(expand = c(0,0)) +
  theme(legend.position = "none")
```



By default, any data outside the limits is converted to `NA`. This means that setting the limits is not the same as visually zooming in to a region of the plot. To do that, you need to use the `xlim` and `ylim` arguments to `coord_cartesian()`, described in Section 7.4. This performs purely visual zooming and does not affect the underlying data. You can override this with the `oob` (out of bounds) argument to the scale. The default is `scales::censor()` which replaces any value outside the limits with `NA`. Another option is `scales::squish()` which squishes all values into the range:

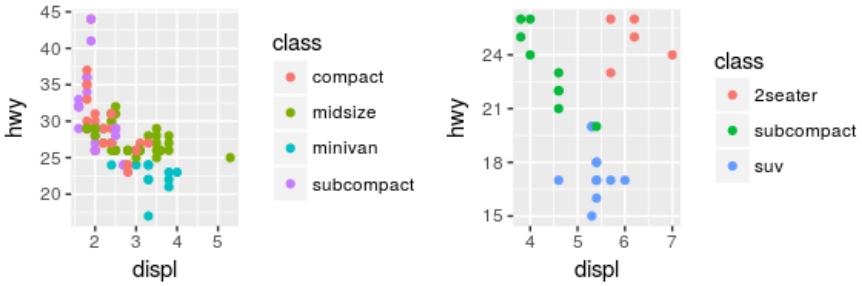
```
df <- data.frame(x = 1:5)
p <- ggplot(df, aes(x, 1)) + geom_tile(aes(fill = x), colour = "white")
p
p + scale_fill_gradient(limits = c(2, 4))
p + scale_fill_gradient(limits = c(2, 4), oob = scales::squish)
```



6.5.1 Exercises

1. The following code creates two plots of the mpg dataset. Modify the code so that the legend and axes match, without using facetting!

```
fwd <- subset(mpg, drv == "f")
rwd <- subset(mpg, drv == "r")

ggplot(fwd, aes(displ, hwy, colour = class)) + geom_point()
ggplot(rwd, aes(displ, hwy, colour = class)) + geom_point()
```



2. What does `expand_limits()` do and how does it work? Read the source code.
3. What happens if you add two `xlim()` calls to the same plot? Why?
4. What does `scale_x_continuous(limits = c(NA, NA))` do?

6.6 Scales toolbox

As well as tweaking the options of the default scales, you can also override them completely with new scales. Scales can be divided roughly into four families:

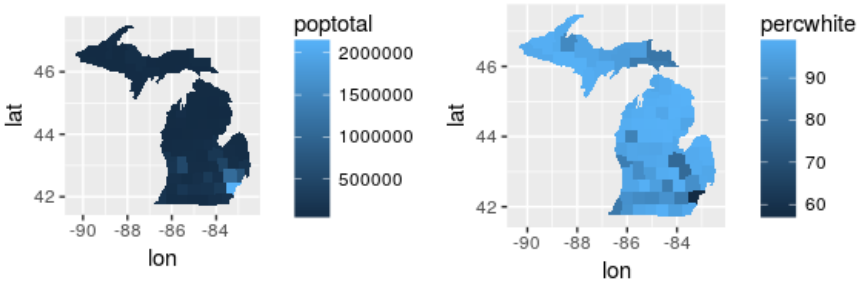
- Continuous position scales used to map integer, numeric, and date/time data to x and y position.
- Colour scales, used to map continuous and discrete data to colours.
- Manual scales, used to map discrete variables to your choice of size, line type, shape or colour.
- The identity scale, paradoxically used to plot variables *without* scaling them. This is useful if your data is already a vector of colour names.

The follow sections describe each family in more detail.

6.6.1 Continuous position scales

Every plot has two position scales, x and y. The most common continuous position scales are `scale_x_continuous()` and `scale_y_continuous()`, which linearly map data to the x and y axis. The most interesting variations are produced using transformations. Every continuous scale takes a `trans` argument, allowing the use of a variety of transformations:

```
# Convert from fuel economy to fuel consumption
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  scale_y_continuous(trans = "reciprocal")

# Log transform x and y axes
ggplot(diamonds, aes(price, carat)) +
  geom_bin2d() +
  scale_x_continuous(trans = "log10") +
  scale_y_continuous(trans = "log10")
```



The transformation is carried out by a "transformer", which describes the transformation, its inverse, and how to draw the labels. The following table lists the most common variants:

Name	Function $f(x)$	Inverse $f^{-1}(y)$
asn	$\tanh^{-1}(x)$	$\tanh(y)$
exp	e^x	$\log(y)$
identity	x	y
log	$\log(x)$	e^y
log10	$\log_{10}(x)$	10^y
log2	$\log_2(x)$	2^y
logit	$\log(\frac{x}{1-x})$	$\frac{1}{1+e(y)}$
pow10	10^x	$\log_{10}(y)$

Name	Function $f(x)$	Inverse $f^{-1}(y)$
probit	$\Phi(x)$	$\Phi^{-1}(y)$
reciprocal	x^{-1}	y^{-1}
reverse	$-x$	$-y$
sqrt	$x^{1/2}$	y^2

There are shortcuts for the most common: `scale_x_log10()`, `scale_x_sqrt()` and `scale_x_reverse()` (and similarly for `y`.)

Of course, you can also perform the transformation yourself. For example, instead of using `scale_x_log10()`, you could plot `log10(x)`. The appearance of the geom will be the same, but the tick labels will be different. If you use a transformed scale, the axes will be labelled in the original data space; if you transform the data, the axes will be labelled in the transformed space.

In either case, the transformation occurs before any statistical summaries. To transform, *after* statistical computation, use `coord_trans()`. See Section 7.4 for more details.

Date and date/time data are continuous variables with special labels. ggplot2 works with `Date` (for dates) and `POSIXct` (for date/times) classes: if your dates are in a different format you will need to convert them with `as.Date()` or `as.POSIXct()`. `scale_x_date()` and `scale_x_datetime()` work similarly to `scale_x_continuous()` but have special `date_breaks` and `date_labels` arguments that work in date-friendly units:

- `date_breaks` and `date_minor_breaks()` allows you to position breaks by date units (years, months, weeks, days, hours, minutes, and seconds). For example, `date_breaks = "2 weeks"` will place a major tick mark every two weeks.
- `date_labels` controls the display of the labels using the same formatting strings as in `strptime()` and `format()`:

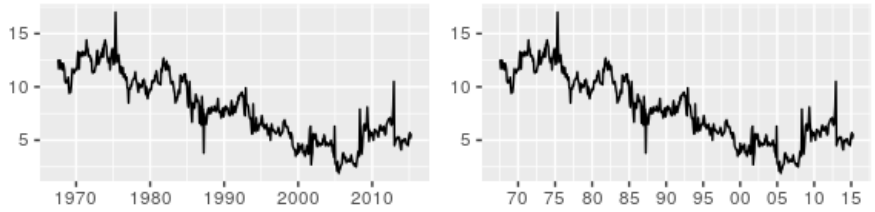
String	Meaning
`%S`	second (00-59)
`%M`	minute (00-59)
`%l`	hour, in 12-hour clock (1-12)
`%I`	hour, in 12-hour clock (01-12)
`%p`	am/pm
`%H`	hour, in 24-hour clock (00-23)
`%a`	day of week, abbreviated (Mon-Sun)
`%A`	day of week, full (Monday-Sunday)
`%e`	day of month (1-31)
`%d`	day of month (01-31)
`%m`	month, numeric (01-12)
`%b`	month, abbreviated (Jan-Dec)
`%B`	month, full (January-December)

String	Meaning
%y	year, without century (00-99)
%Y	year, with century (0000-9999)

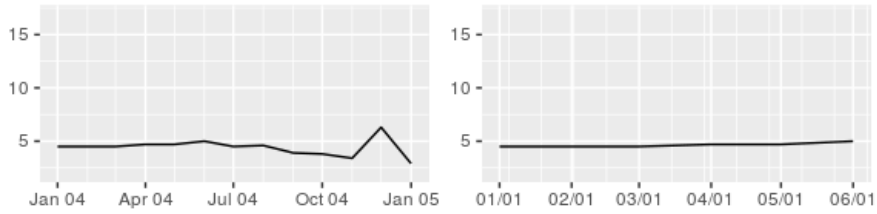
For example, if you wanted to display dates like 14/10/1979, you would use the string "%d/%m/%Y".

The code below illustrates some of these parameters.

```
base <- ggplot(economics, aes(date, psavert)) +
  geom_line(na.rm = TRUE) +
  labs(x = NULL, y = NULL)

base # Default breaks and labels
base + scale_x_date(date_labels = "%y", date_breaks = "5 years")
```




```
base + scale_x_date(
  limits = as.Date(c("2004-01-01", "2005-01-01")),
  date_labels = "%b %y",
  date_minor_breaks = "1 month"
)
base + scale_x_date(
  limits = as.Date(c("2004-01-01", "2004-06-01")),
  date_labels = "%m/%d",
  date_minor_breaks = "2 weeks"
)
```



6.6.2 Colour

After position, the most commonly used aesthetic is colour. There are quite a few different ways of mapping values to colours in ggplot2: four different gradient-based methods for continuous values, and two methods for mapping discrete values. But before we look at the details of the different methods, it's useful to learn a little bit of colour theory. Colour theory is complex because the underlying biology of the eye and brain is complex, and this introduction will only touch on some of the more important issues. An excellent and more detailed exposition is available online at `http://tinyurl.com/clrdtls`.

At the physical level, colour is produced by a mixture of wavelengths of light. To characterise a colour completely, we need to know the complete mixture of wavelengths. Fortunately for us the human eye only has three different colour receptors, and so we can summarise the perception of any colour with just three numbers. You may be familiar with the RGB encoding of colour space, which defines a colour by the intensities of red, green and blue light needed to produce it. One problem with this space is that it is not perceptually uniform: the two colours that are one unit apart may look similar or very different depending on where they are in the colour space. This makes it difficult to create a mapping from a continuous variable to a set of colours. There have been many attempts to come up with colours spaces that are more perceptually uniform. We'll use a modern attempt called the HCL colour space, which has three components of **h**ue, **c**hroma and **l**uminance:

- Hue is a number between 0 and 360 (an angle) which gives the "colour" of the colour: like blue, red, orange, etc.
- Chroma is the purity of a colour. A chroma of 0 is grey, and the maximum value of chroma varies with luminance.
- Luminance is the lightness of the colour. A luminance of 0 produces black, and a luminance of 1 produces white.

Hues are not perceived as being ordered: e.g. green does not seem "larger" than red. The perception of chroma and luminance are ordered.

The combination of these three components does not produce a simple geometric shape. Figure 6.2 attempts to show the 3d shape of the space. Each slice is a constant luminance (brightness) with hue mapped to angle and chroma to radius. You can see the centre of each slice is grey and the colours get more intense as they get closer to the edge.

An additional complication is that many people (˜10% of men) do not possess the normal complement of colour receptors and so can distinguish fewer colours than usual. In brief, it's best to avoid red-green contrasts, and to check your plots with systems that simulate colour blindness. Visicheck is one online solution. Another alternative is the **dichromat** package (Lumley 2007) which provides tools for simulating colour blindness, and a set of colour schemes known to work well for colour-blind people. You can also help people

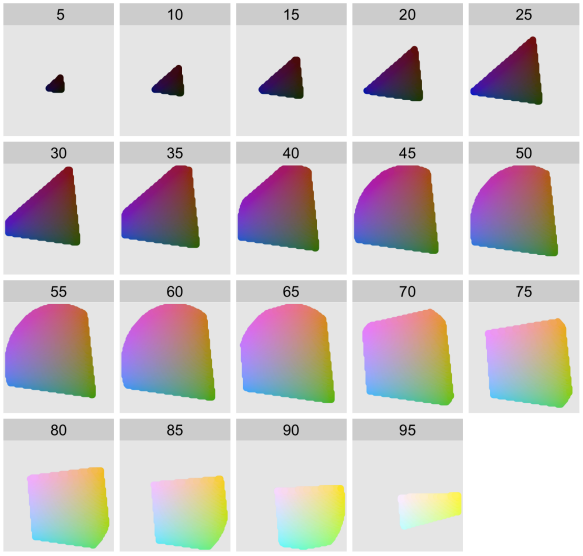


Fig. 6.2 The shape of the HCL colour space. Hue is mapped to angle, chroma to radius and each slice shows a different luminance. The HCL space is a pretty odd shape, but you can see that colours near the centre of each slice are grey, and as you move towards the edges they become more intense. Slices for luminance 0 and 100 are omitted because they would, respectively, be a single black point and a single white point.

with colour blindness in the same way that you can help people with black-and-white printers: by providing redundant mappings to other aesthetics like size, line type or shape.

6.6.2.1 Continuous

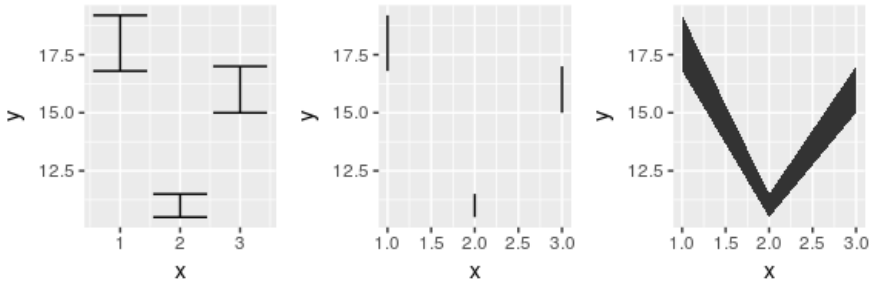
Colour gradients are often used to show the height of a 2d surface. In the following example we'll use the surface of a 2d density estimate of the `faithful` dataset (Azzalini and Bowman 1990), which records the waiting time between eruptions and during each eruption for the Old Faithful geyser in Yellowstone Park. I hide the legends and set `expand` to 0, to focus on the appearance of the data. . Remember: I'm illustrating these scales with filled tiles, but you can also use them with coloured lines and points.

```
erupt <- ggplot(faithfuld, aes(waiting, eruptions, fill = density)) +
  geom_raster() +
  scale_x_continuous(NULL, expand = c(0, 0)) +
  scale_y_continuous(NULL, expand = c(0, 0)) +
  theme(legend.position = "none")
```

There are four continuous colour scales:

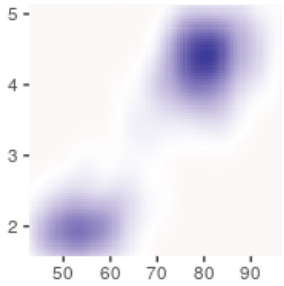
- `scale_colour_gradient()` and `scale_fill_gradient()`: a two-colour gradient, low-high (light blue-dark blue). This is the default scale for continuous colour, and is the same as `scale_colour_continuous()`. Arguments `low` and `high` control the colours at either end of the gradient.
  Generally, for continuous colour scales you want to keep hue constant, and vary chroma and luminance. The munsell colour system is useful for this as it provides an easy way of specifying colours based on their hue, chroma and luminance. Use `munsell::hue_slice("5Y")` to see the valid chroma and luminance values for a given hue.

```
erupt

erupt + scale_fill_gradient(low = "white", high = "black")

erupt + scale_fill_gradient(
  low = munsell::mnsl("5G 9/2"),
  high = munsell::mnsl("5G 6/8")
)
```



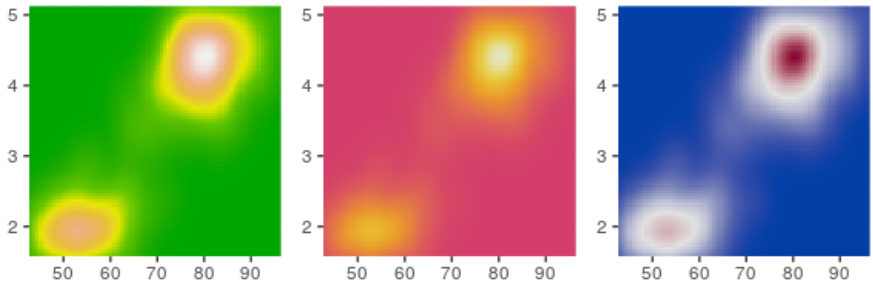
- `scale_colour_gradient2()` and `scale_fill_gradient2()`: a three-colour gradient, low-med-high (red-white-blue). As well as `low` and `high` colours, these scales also have a `mid` colour for the colour of the midpoint. The midpoint defaults to 0, but can be set to any value with the `midpoint` argument.
  It's artificial to use this colour scale with this dataset, but we can force it by using the median of the density as the midpoint. Note that the blues are much more intense than the reds (which you only see as a very pale pink)

```
mid <- median(faithfuld$density)
erupt + scale_fill_gradient2(midpoint = mid)
```



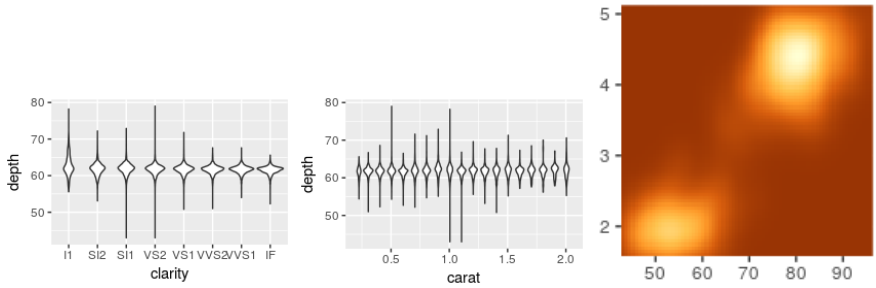
- `scale_colour_gradientn()` and `scale_fill_gradientn()`: a custom n-colour gradient. This is useful if you have colours that are meaningful for your data (e.g., black body colours or standard terrain colours), or you'd like to use a palette produced by another package. The following code includes palettes generated from routines in the **colorspace** package. (Zeileis, Hornik, and Murrell 2008) describes the philosophy behind these palettes and provides a good introduction to some of the complexities of creating good colour scales.

```
erupt + scale_fill_gradientn(colours = terrain.colors(7))
erupt + scale_fill_gradientn(colours = colorspace::heat_hcl(7))
erupt + scale_fill_gradientn(colours = colorspace::diverge_hcl(7))
```



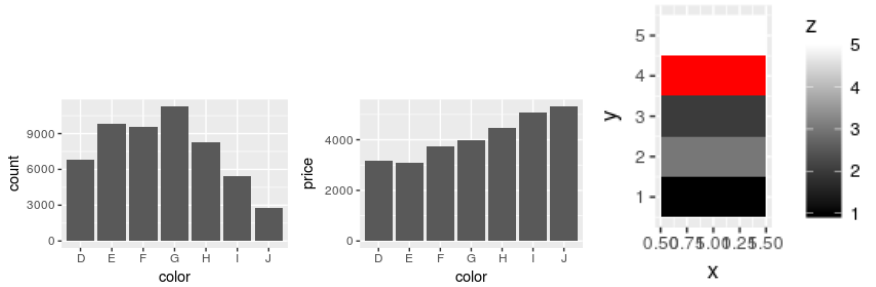
  By default, `colours` will be evenly spaced along the range of the data. To make them unevenly spaced, use the `values` argument, which should be a vector of values between 0 and 1.
- `scale_color_distiller()` and `scale_fill_gradient()` apply the ColorBrewer colour scales to continuous data. You use it the same way as `scale_fill_brewer()`, described below:

```
erupt + scale_fill_distiller()
erupt + scale_fill_distiller(palette = "RdPu")
erupt + scale_fill_distiller(palette = "YlOrBr")
```



All continuous colour scales have an `na.value` parameter that controls what colour is used for missing values (including values outside the range of the scale limits). By default it is set to grey, which will stand out when you use a colourful scale. If you use a black and white scale, you might want to set it to something else to make it more obvious.

```
df <- data.frame(x = 1, y = 1:5, z = c(1, 3, 2, NA, 5))
p <- ggplot(df, aes(x, y)) + geom_tile(aes(fill = z), size = 5)
p
# Make missing colours invisible
p + scale_fill_gradient(na.value = NA)
# Customise on a black and white scale
p + scale_fill_gradient(low = "black", high = "white", na.value = "red")
```



6.6.2.2 Discrete

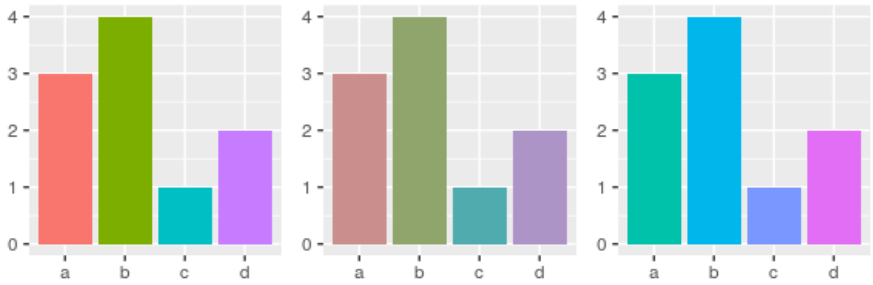
There are four colour scales for discrete data. We illustrate them with a barchart that encodes both position and fill to the same variable:

```
df <- data.frame(x = c("a", "b", "c", "d"), y = c(3, 4, 1, 2))
bars <- ggplot(df, aes(x, y, fill = x)) +
  geom_bar(stat = "identity") +
```

```
  labs(x = NULL, y = NULL) +
  theme(legend.position = "none")
```

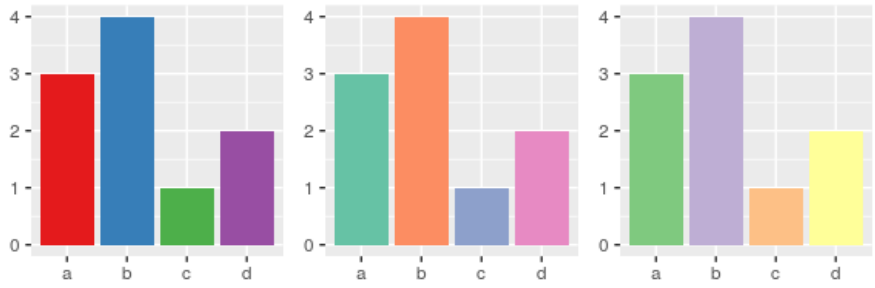
- The default colour scheme, `scale_colour_hue()`, picks evenly spaced hues around the HCL colour wheel. This works well for up to about eight colours, but after that it becomes hard to tell the different colours apart. You can control the default chroma and luminance, and the range of hues, with the `h`, `c` and `l` arguments:

  ```
  bars
  bars + scale_fill_hue(c = 40)
  bars + scale_fill_hue(h = c(180, 300))
  ```



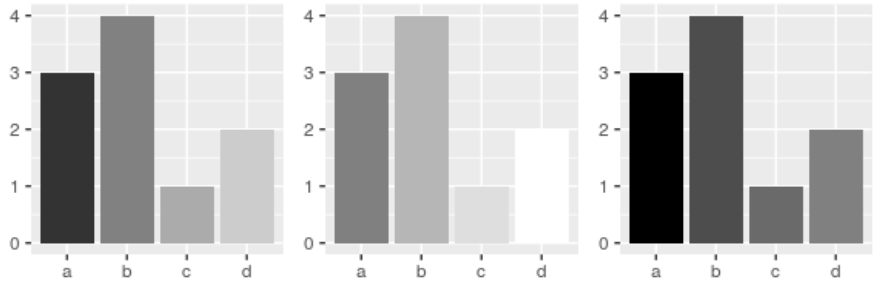
  One disadvantage of the default colour scheme is that because the colours all have the same luminance and chroma, when you print them in black and white, they all appear as an identical shade of grey.
- `scale_colour_brewer()` uses handpicked "ColorBrewer" colours, `http://colorbrewer2.org/`. These colours have been designed to work well in a wide variety of situations, although the focus is on maps and so the colours tend to work better when displayed in large areas. For categorical data, the palettes most of interest are 'Set1' and 'Dark2' for points and 'Set2', 'Pastel1', 'Pastel2' and 'Accent' for areas. Use `RColorBrewer::display.brewer.all()` to list all palettes.

  ```
  bars + scale_fill_brewer(palette = "Set1")
  bars + scale_fill_brewer(palette = "Set2")
  bars + scale_fill_brewer(palette = "Accent")
  ```



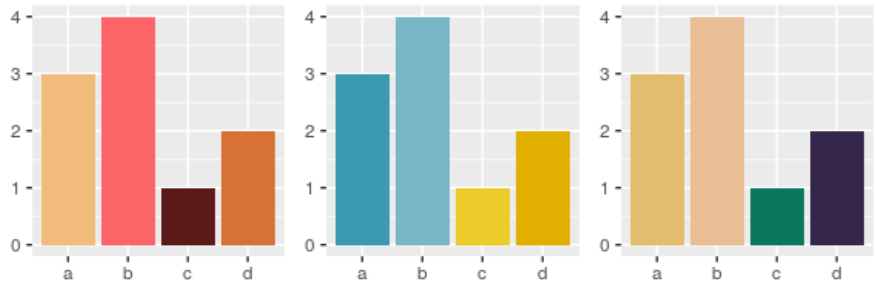
- `scale_colour_grey()` maps discrete data to grays, from light to dark.

```
bars + scale_fill_grey()
bars + scale_fill_grey(start = 0.5, end = 1)
bars + scale_fill_grey(start = 0, end = 0.5)
```



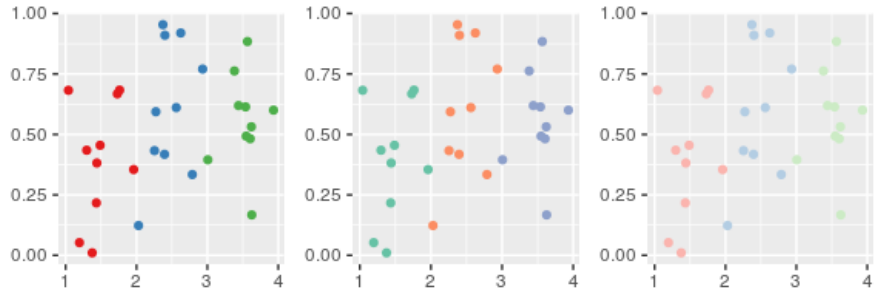
- `scale_colour_manual()` is useful if you have your own discrete colour palette. The following examples show colour palettes inspired by Wes Anderson movies, as provided by the wesanderson package, `https://github.com/karthik/wesanderson`. These are not designed for perceptual uniformity, but are fun!

```
library(wesanderson)
bars + scale_fill_manual(values = wes_palette("GrandBudapest"))
bars + scale_fill_manual(values = wes_palette("Zissou"))
bars + scale_fill_manual(values = wes_palette("Rushmore"))
```

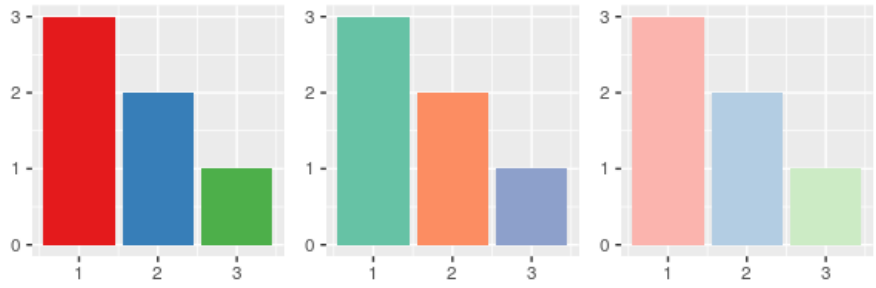


Note that one set of colours is not uniformly good for all purposes: bright colours work well for points, but are overwhelming on bars. Subtle colours work well for bars, but are hard to see on points:

```
# Bright colours work best with points
df <- data.frame(x = 1:3 + runif(30), y = runif(30), z = c("a", "b", "c"))
point <- ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z))  +
  theme(legend.position = "none") +
  labs(x = NULL, y = NULL)
point + scale_colour_brewer(palette = "Set1")
point + scale_colour_brewer(palette = "Set2")
point + scale_colour_brewer(palette = "Pastel1")
```




```
# Subtler colours work better with areas
df <- data.frame(x = 1:3, y = 3:1, z = c("a", "b", "c"))
area <- ggplot(df, aes(x, y)) +
  geom_bar(aes(fill = z), stat = "identity") +
  theme(legend.position = "none") +
  labs(x = NULL, y = NULL)
area + scale_fill_brewer(palette = "Set1")
area + scale_fill_brewer(palette = "Set2")
area + scale_fill_brewer(palette = "Pastel1")
```



6.6.3 The manual discrete scale

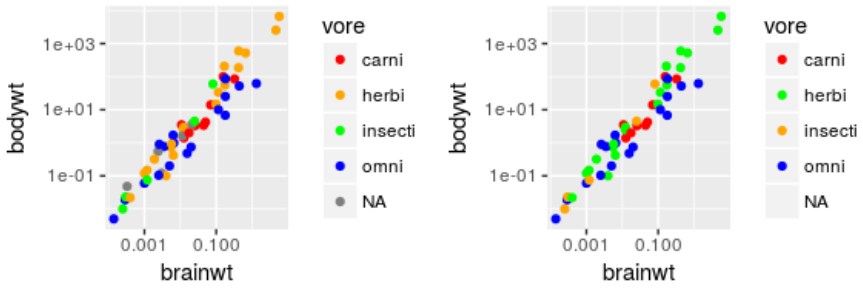
The discrete scales, `scale_linetype()`, `scale_shape()`, and `scale_size_discrete()` basically have no options. These scales are just a list of valid values that are mapped to the unique discrete values.

If you want to customise these scales, you need to create your own new scale with the manual scale: `scale_shape_manual()`, `scale_linetype_manual()`, `scale_colour_manual()`. The manual scale has one important argument, `values`, where you specify the values that the scale should produce. If this vector is named, it will match the values of the output to the values of the input; otherwise it will match in order of the levels of the discrete variable. You will need some knowledge of the valid aesthetic values, which are described in `vignette("ggplot2-specs")`.

The following code demonstrates the use of `scale_colour_manual()`:

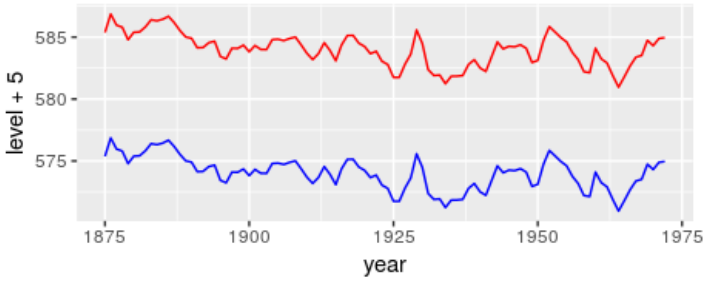
```
plot <- ggplot(msleep, aes(brainwt, bodywt)) +
  scale_x_log10() +
  scale_y_log10()
plot +
  geom_point(aes(colour = vore)) +
  scale_colour_manual(
    values = c("red", "orange", "green", "blue"),
    na.value = "grey50"
  )
#> Warning: Removed 27 rows containing missing values (geom_point).

colours <- c(
  carni = "red",
  insecti = "orange",
  herbi = "green",
  omni = "blue"
)
plot +
```

```
  geom_point(aes(colour = vore)) +
  scale_colour_manual(values = colours)
#> Warning: Removed 32 rows containing missing values (geom_point).
```



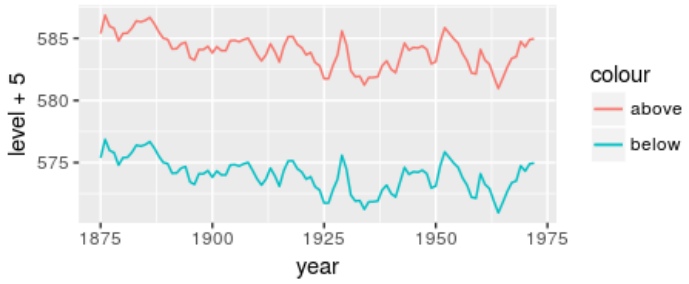
The following example shows a creative use of `scale_colour_manual()` to display multiple variables on the same plot and show a useful legend. In most other plotting systems, you'd colour the lines and then add a legend:

```
huron <- data.frame(year = 1875:1972, level = as.numeric(LakeHuron))
ggplot(huron, aes(year)) +
  geom_line(aes(y = level + 5), colour = "red") +
  geom_line(aes(y = level - 5), colour = "blue")
```



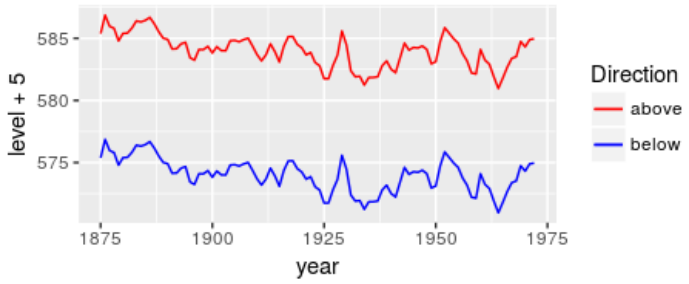
That doesn't work in ggplot because there's no way to add a legend manually. Instead, give the lines informative labels:

```
ggplot(huron, aes(year)) +
  geom_line(aes(y = level + 5, colour = "above")) +
  geom_line(aes(y = level - 5, colour = "below"))
```



And then tell the scale how to map labels to colours:

```
ggplot(huron, aes(year)) +
  geom_line(aes(y = level + 5, colour = "above")) +
  geom_line(aes(y = level - 5, colour = "below")) +
  scale_colour_manual("Direction",
    values = c("above" = "red", "below" = "blue")
  )
```

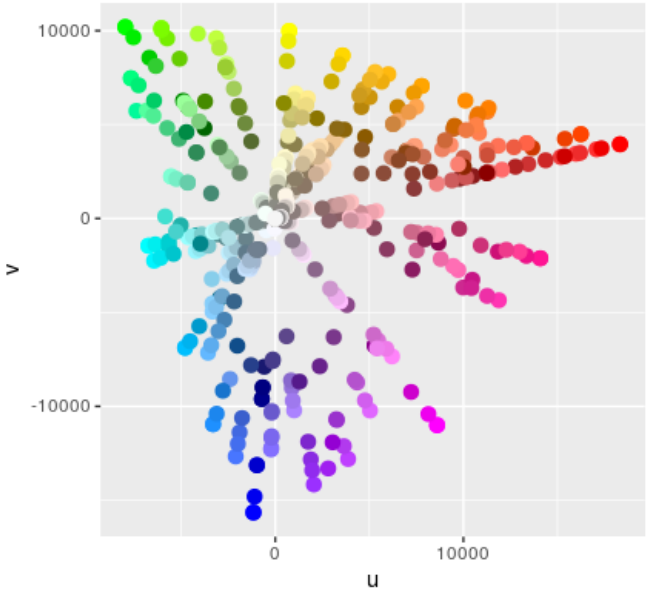


See Section 9.3 for another approach.

6.6.4 The identity scale

The identity scale is used when your data is already scaled, when the data and aesthetic spaces are the same. The code below shows an example where the identity scale is useful. `luv_colours` contains the locations of all R's built-in colours in the LUV colour space (the space that HCL is based on). A legend is unnecessary, because the point colour represents itself: the data and aesthetic spaces are the same.

```
head(luv_colours)
#>      L         u    v           col
#> 1 9342 -3.37e-12    0         white
#> 2 9101 -4.75e+02 -635     aliceblue
#> 3 8810  1.01e+03 1668  antiquewhite
#> 4 8935  1.07e+03 1675 antiquewhite1
#> 5 8452  1.01e+03 1610 antiquewhite2
#> 6 7498  9.03e+02 1402 antiquewhite3

ggplot(luv_colours, aes(u, v)) +
  geom_point(aes(colour = col), size = 3) +
  scale_color_identity() +
  coord_equal()
```



6.6.5 Exercises

1. Compare and contrast the four continuous colour scales with the four discrete scales.
2. Explore the distribution of the built-in `colors()` using the `luv_colours` dataset.

References

Azzalini, A., and A. W. Bowman. 1990. "A Look at Some Data on the Old Faithful Geyser." *Applied Statistics* 39: 357–65.

Lumley, Thomas. 2007. *Dichromat: Color Schemes for Dichromats*.

Zeileis, Achim, Kurt Hornik, and Paul Murrell. 2008. "Escaping RGB-land: Selecting Colors for Statistical Graphics." *Computational Statistics & Data Analysis*. http://statmath.wu-wien.ac.at/~zeileis/papers/Zeileis+Hornik+Murrell-2008.pdf.

Chapter 7
Positioning

7.1 Introduction

This chapter discusses position, particularly how facets are laid out on a page, and how coordinate systems within a panel work. There are four components that control position. You have already learned about two of them that work within a facet:

- **Position adjustments** adjust the position of overlapping objects within a layer. These are most useful for bar and other interval geoms, but can be useful in other situations (Section 5.7).
- **Position scales** control how the values in the data are mapped to positions on the plot (Section 6.6.1).

This chapter will describe the other two components and show you how all four pieces fit together:

- **Facetting** is a mechanism for automatically laying out multiple plots on a page. It splits the data into subsets, and then plots each subset in a different panel. Such plots are often called small multiples or trellis graphics (Section 7.2).
- **Coordinate systems** control how the two independent position scales are combined to create a 2d coordinate system. The most common coordinate system is Cartesian, but other coordinate systems can be useful in special circumstances (Section 7.3).

7.2 Facetting

You first encountered facetting in Section 2.5. Facetting generates small multiples each showing a different subset of the data. Small multiples are a powerful tool for exploratory data analysis: you can rapidly compare patterns in

different parts of the data and see whether they are the same or different. This section will discuss how you can fine-tune facets, particularly the way in which they interact with position scales.

There are three types of facetting:

- `facet_null()`: a single plot, the default.
- `facet_wrap()`: "wraps" a 1d ribbon of panels into 2d.
- `facet_grid()`: produces a 2d grid of panels defined by variables which form the rows and columns.

The differences between `facet_wrap()` and `facet_grid()` are illustrated in Figure 7.1.

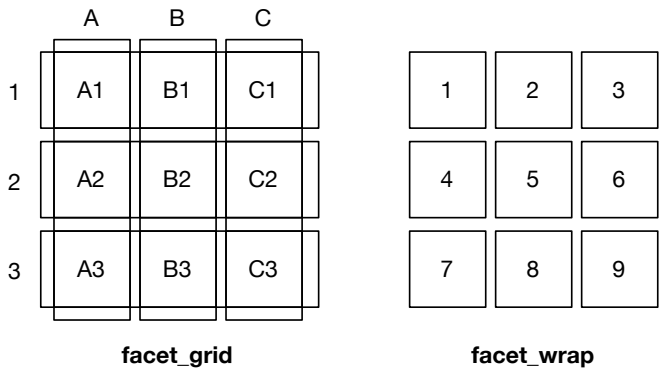


Fig. 7.1 A sketch illustrating the difference between the two facetting systems. `facet_grid()` (left) is fundamentally 2d, being made up of two independent components. `facet_wrap()` (right) is 1d, but wrapped into 2d to save space.

Faceted plots have the capability to fill up a lot of space, so for this chapter we will use a subset of the mpg dataset that has a manageable number of levels: three cylinders (4, 6, 8), two types of drive train (4 and f), and six classes.

```
mpg2 <- subset(mpg, cyl != 5 & drv %in% c("4", "f") & class != "2seater")
```

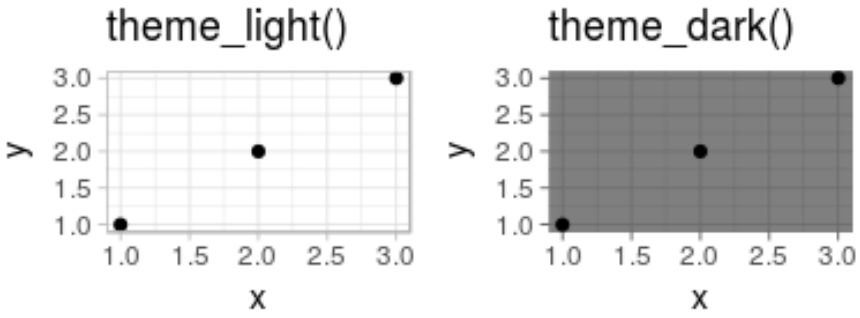
7.2.1 Facet wrap

`facet_wrap()` makes a long ribbon of panels (generated by any number of variables) and wraps it into 2d. This is useful if you have a single variable with many levels and want to arrange the plots in a more space efficient manner.

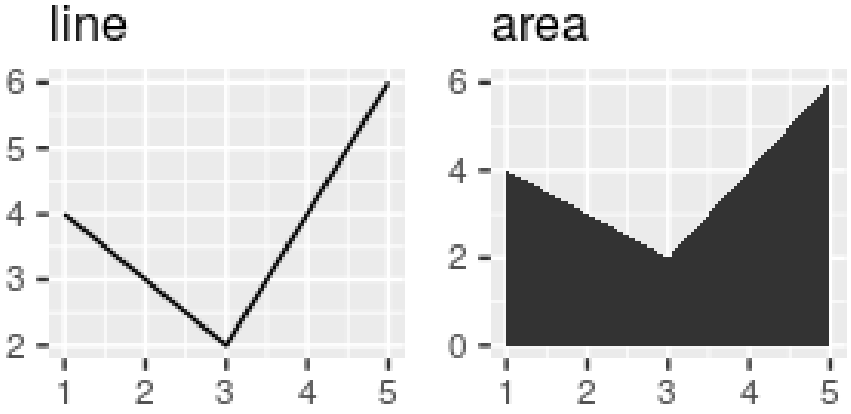
You can control how the ribbon is wrapped into a grid with `ncol`, `nrow`, `as.table` and `dir`. `ncol` and `nrow` control how many columns and rows (you

only need to set one). `as.table` controls whether the facets are laid out like a table (`TRUE`), with highest values at the bottom-right, or a plot (`FALSE`), with the highest values at the top-right. `dir` controls the direction of wrap: **h**orizontal or **v**ertical.

```
base <- ggplot(mpg2, aes(displ, hwy)) +
  geom_blank() +
  xlab(NULL) +
  ylab(NULL)

base + facet_wrap(~class, ncol = 3)
base + facet_wrap(~class, ncol = 3, as.table = FALSE)
```




```
base + facet_wrap(~class, nrow = 3)
base + facet_wrap(~class, nrow = 3, dir = "v")
```

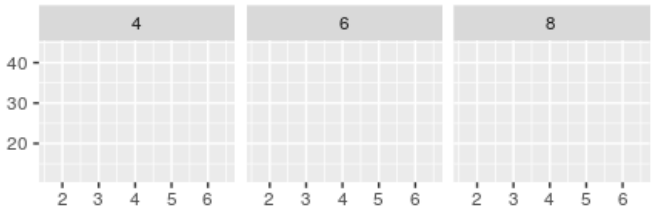


7.2.2 Facet grid

`facet_grid()` lays out plots in a 2d grid, as defined by a formula:

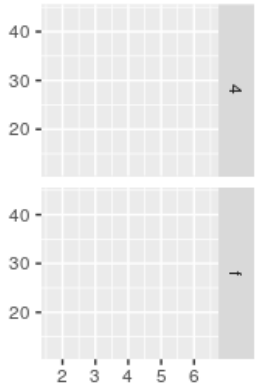
- `. ~ a` spreads the values of `a` across the columns. This direction facilitates comparisons of y position, because the vertical scales are aligned.

```
base + facet_grid(. ~ cyl)
```



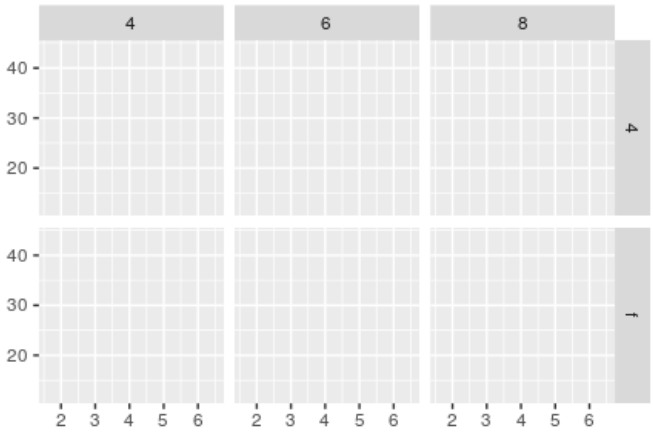
- `b ~ .` spreads the values of `b` down the rows. This direction facilitates comparison of x position because the horizontal scales are aligned. This makes it particularly useful for comparing distributions.

```
base + facet_grid(drv ~ .)
```



- `a ~ b` spreads `a` across columns and `b` down rows. You'll usually want to put the variable with the greatest number of levels in the columns, to take advantage of the aspect ratio of your screen.

```
base + facet_grid(drv ~ cyl)
```



You can use multiple variables in the rows or columns, by "adding" them together, e.g. `a + b ~ c + d`. Variables appearing together on the rows or columns are nested in the sense that only combinations that appear in the data will appear in the plot. Variables that are specified on rows and columns will be crossed: all combinations will be shown, including those that didn't appear in the original dataset: this may result in empty panels.

7.2.3 Controlling scales

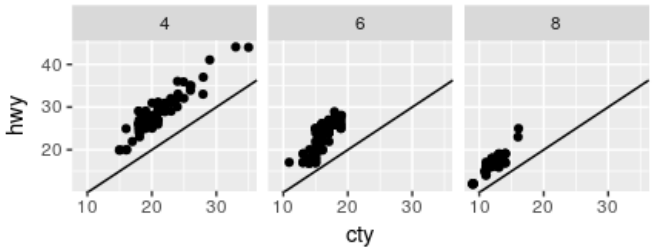
For both `facet_wrap()` and `facet_grid()` you can control whether the position scales are the same in all panels (fixed) or allowed to vary between panels (free) with the `scales` parameter:

- `scales = "fixed"`: x and y scales are fixed across all panels.
- `scales = "free_x"`: the x scale is free, and the y scale is fixed.
- `scales = "free_y"`: the y scale is free, and the x scale is fixed.
- `scales = "free"`: x and y scales vary across panels.

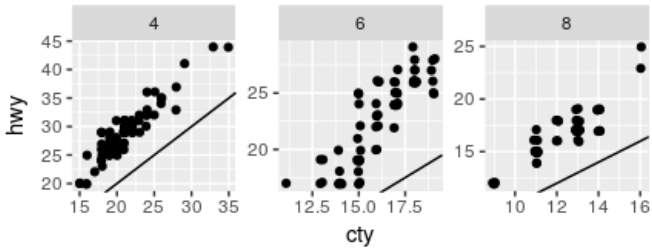
`facet_grid()` imposes an additional constraint on the scales: all panels in a column must have the same x scale, and all panels in a row must have the same y scale. This is because each column shares an x axis, and each row shares a y axis.

Fixed scales make it easier to see patterns across panels; free scales make it easier to see patterns within panels.

```
p <- ggplot(mpg2, aes(cty, hwy)) +
  geom_abline() +
  geom_jitter(width = 0.1, height = 0.1)
p + facet_wrap(~cyl)
```



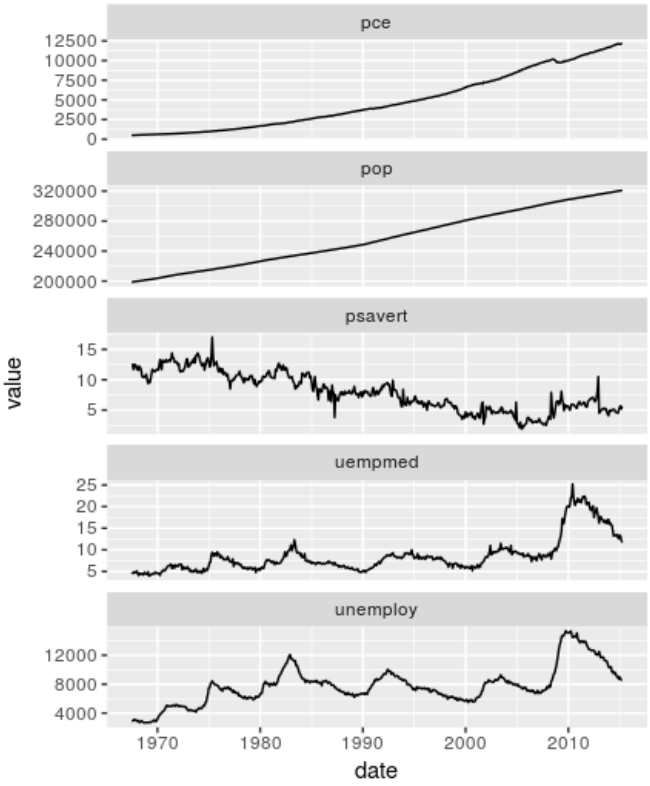

```
p + facet_wrap(~cyl, scales = "free")
```



Free scales are also useful when we want to display multiple time series that were measured on different scales. To do this, we first need to change from 'wide' to 'long' data, stacking the separate variables into a single column. An example of this is shown below with the long form of the `economics` data, and the topic is discussed in more detail in Section 9.3.

```
economics_long
#> # A tibble: 2,870 x 4
#> # Groups:   variable [5]
#>          date variable value  value01
#>        <date>   <fctr> <dbl>    <dbl>
#>  1 1967-07-01      pce   507 0.000000
#>  2 1967-08-01      pce   510 0.000266
#>  3 1967-09-01      pce   516 0.000764
#>  4 1967-10-01      pce   513 0.000472
#>  5 1967-11-01      pce   518 0.000918
#>  6 1967-12-01      pce   526 0.001579
#>  7 1968-01-01      pce   532 0.002068
#>  8 1968-02-01      pce   534 0.002300
#>  9 1968-03-01      pce   545 0.003218
#> 10 1968-04-01      pce   545 0.003192
#> # ... with 2,860 more rows
```

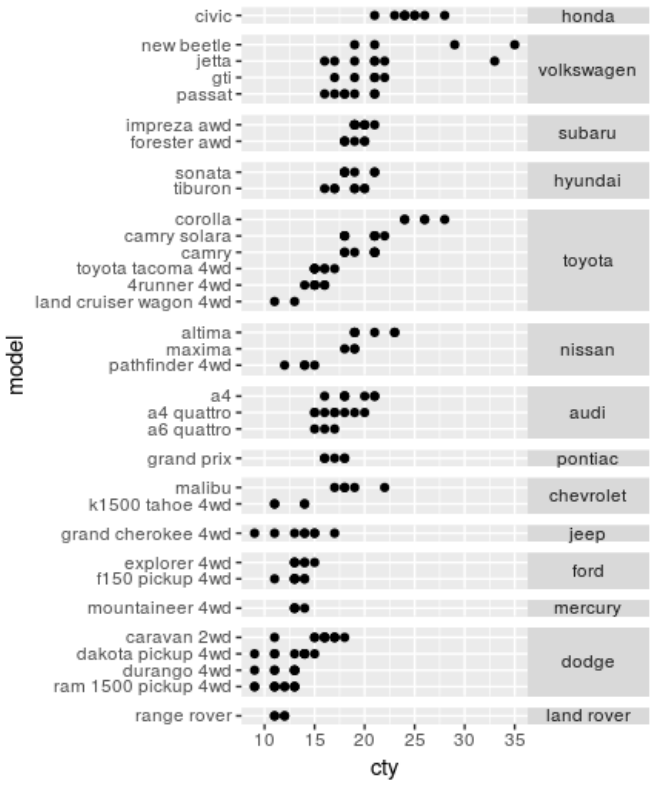
```
ggplot(economics_long, aes(date, value)) +
  geom_line() +
  facet_wrap(~variable, scales = "free_y", ncol = 1)
```



`facet_grid()` has an additional parameter called `space`, which takes the same values as `scales`. When space is "free", each column (or row) will have width (or height) proportional to the range of the scale for that column (or row). This makes the scaling equal across the whole plot: 1 cm on each panel maps to the same range of data. (This is somewhat analogous to the 'sliced' axis limits of lattice.) For example, if panel a had range 2 and panel b had range 4, one-third of the space would be given to a, and two-thirds to b. This is most useful for categorical scales, where we can assign space proportionally based on the number of levels in each facet, as illustrated below.

```
mpg2$model <- reorder(mpg2$model, mpg2$cty)
mpg2$manufacturer <- reorder(mpg2$manufacturer, -mpg2$cty)
ggplot(mpg2, aes(cty, model)) +
  geom_point() +
```

```
  facet_grid(manufacturer ~ ., scales = "free", space = "free") +
  theme(strip.text.y = element_text(angle = 0))
```

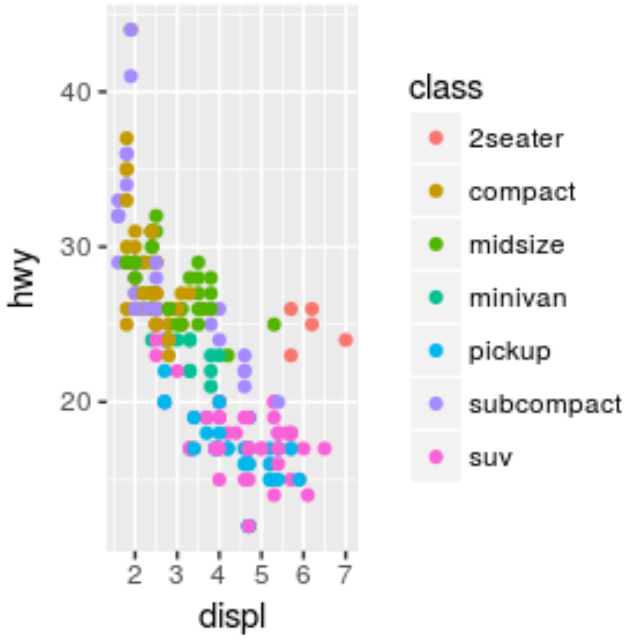


7.2.4 Missing facetting variables

If you are using facetting on a plot with multiple datasets, what happens when one of those datasets is missing the facetting variables? This situation commonly arises when you are adding contextual information that should be the same in all panels. For example, imagine you have a spatial display of disease faceted by gender. What happens when you add a map layer that does not contain the gender variable? Here ggplot will do what you expect: it will display the map in every facet: missing facetting variables are treated like they have all values.

Here's a simple example. Note how the single red point from `df2` appears in both panels.

```
df1 <- data.frame(x = 1:3, y = 1:3, gender = c("f", "f", "m"))
df2 <- data.frame(x = 2, y = 2)

ggplot(df1, aes(x, y)) +
  geom_point(data = df2, colour = "red", size = 2) +
  geom_point() +
  facet_wrap(~gender)
```

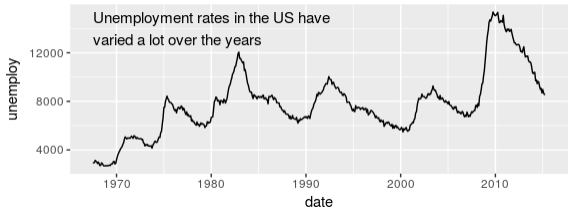


This technique is particularly useful when you add annotations to make it easier to compare between facets, as shown in the next section.

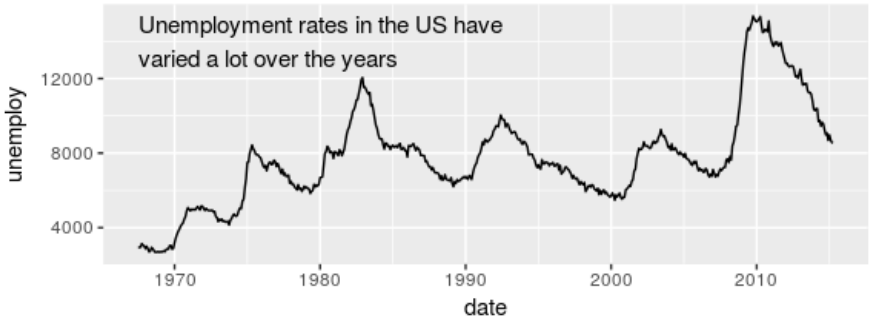
7.2.5 Grouping vs. facetting

Facetting is an alternative to using aesthetics (like colour, shape or size) to differentiate groups. Both techniques have strengths and weaknesses, based around the relative positions of the subsets. With facetting, each group is quite far apart in its own panel, and there is no overlap between the groups. This is good if the groups overlap a lot, but it does make small differences harder to see. When using aesthetics to differentiate groups, the groups are close together and may overlap, but small differences are easier to see.

```
df <- data.frame(
  x = rnorm(120, c(0, 2, 4)),
  y = rnorm(120, c(1, 2, 1)),
  z = letters[1:3]
)

ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z))
```



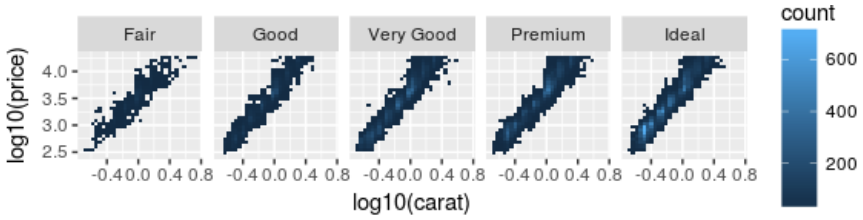

```
ggplot(df, aes(x, y)) +
  geom_point() +
  facet_wrap(~z)
```



Comparisons between facets often benefit from some thoughtful annotation. For example, in this case we could show the mean of each group in every panel. You'll learn how to write summary code like this in Chapter 10. Note that we need two "z" variables: one for the facets and one for the colours.

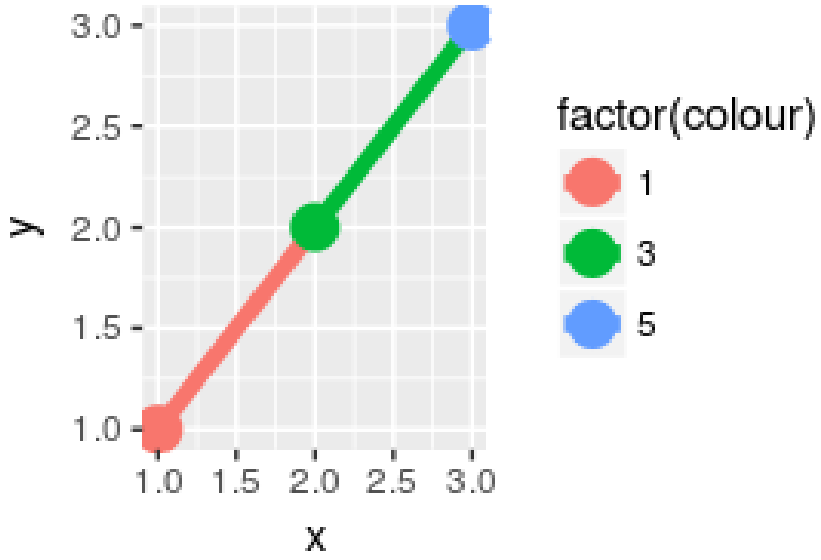
```
df_sum <- df %>%
  group_by(z) %>%
  summarise(x = mean(x), y = mean(y)) %>%
  rename(z2 = z)
```

```
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_point(data = df_sum, aes(colour = z2), size = 4) +
  facet_wrap(~z)
```



Another useful technique is to put all the data in the background of each panel:

```
df2 <- dplyr::select(df, -z)

ggplot(df, aes(x, y)) +
  geom_point(data = df2, colour = "grey70") +
  geom_point(aes(colour = z)) +
  facet_wrap(~z)
```



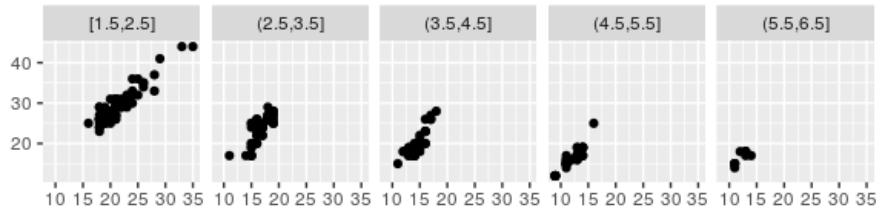
7.2.6 Continuous variables

To facet continuous variables, you must first discretise them. ggplot2 provides three helper functions to do so:

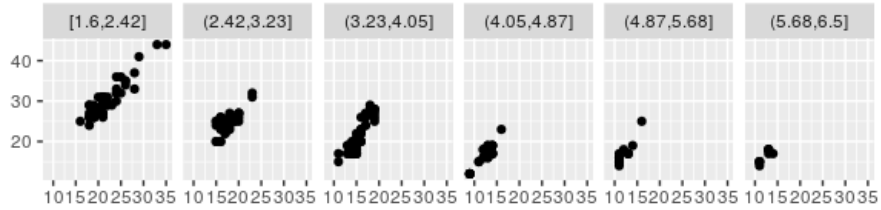
- Divide the data into `n` bins each of the same length: `cut_interval(x, n)`
- Divide the data into bins of width `width`: `cut_width(x, width)`.
- Divide the data into n bins each containing (approximately) the same number of points: `cut_number(x, n = 10)`.

They are illustrated below:

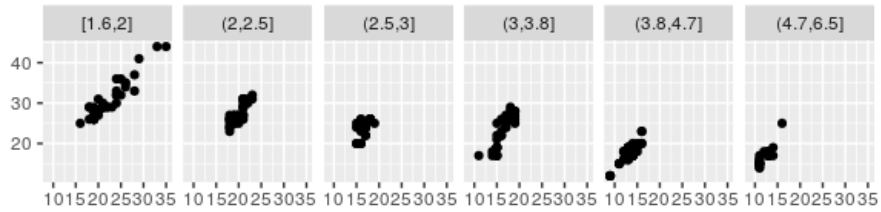
```
# Bins of width 1
mpg2$disp_w <- cut_width(mpg2$displ, 1)
# Six bins of equal length
mpg2$disp_i <- cut_interval(mpg2$displ, 6)
# Six bins containing equal numbers of points
mpg2$disp_n <- cut_number(mpg2$displ, 6)

plot <- ggplot(mpg2, aes(cty, hwy)) +
  geom_point() +
  labs(x = NULL, y = NULL)
plot + facet_wrap(~disp_w, nrow = 1)
```




```
plot + facet_wrap(~disp_i, nrow = 1)
```




```
plot + facet_wrap(~disp_n, nrow = 1)
```

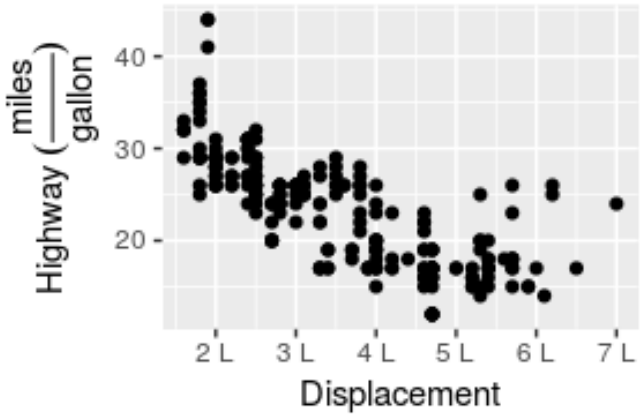


Note that the facetting formula does not evaluate functions, so you must first create a new variable containing the discretised data.

7.2.7 Exercises

1. Diamonds: display the distribution of price conditional on cut and carat. Try facetting by cut and grouping by carat. Try facetting by carat and grouping by cut. Which do you prefer?
2. Diamonds: compare the relationship between price and carat for each colour. What makes it hard to compare the groups? Is grouping better or facetting? If you use facetting, what annotation might you add to make it easier to see the differences between panels?
3. Why is `facet_wrap()` generally more useful than `facet_grid()`?
4. Recreate the following plot. It facets `mpg2` by class, overlaying a smooth curve fit to the full dataset.

```
#> `geom_smooth()` using method = 'loess'
```



7.3 Coordinate systems

Coordinate systems have two main jobs:

- Combine the two position aesthetics to produce a 2d position on the plot. The position aesthetics are called `x` and `y`, but they might be better called position 1 and 2 because their meaning depends on the coordinate system used. For example, with the polar coordinate system they become angle and radius (or radius and angle), and with maps they become latitude and longitude.
- In coordination with the faceter, coordinate systems draw axes and panel backgrounds. While the scales control the values that appear on the axes, and how they map from data to position, it is the coordinate system which actually draws them. This is because their appearance depends on the coordinate system: an angle axis looks quite different than an x axis.

There are two types of coordinate system. Linear coordinate systems preserve the shape of geoms:

- `coord_cartesian()`: the default Cartesian coordinate system, where the 2d position of an element is given by the combination of the x and y positions.
- `coord_flip()`: Cartesian coordinate system with x and y axes flipped.
- `coord_fixed()`: Cartesian coordinate system with a fixed aspect ratio.

On the other hand, non-linear coordinate systems can change the shapes: a straight line may no longer be straight. The closest distance between two points may no longer be a straight line.

- `coord_map()`/`coord_quickmap()`: Map projections.
- `coord_polar()`: Polar coordinates.
- `coord_trans()`: Apply arbitrary transformations to x and y positions, after the data has been processed by the stat.

Each coordinate system is described in more detail below.

7.4 Linear coordinate systems

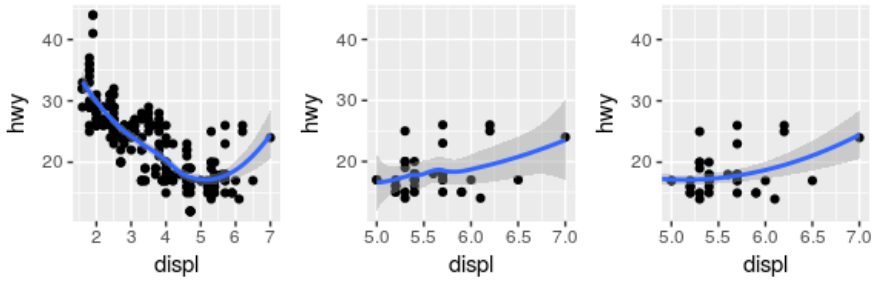
There are three linear coordinate systems: `coord_cartesian()`, `coord_flip()`, `coord_fixed()`.

7.4.1 Zooming into a plot with `coord_cartesian()`

`coord_cartesian()` has arguments `xlim` and `ylim`. If you think back to the scales chapter, you might wonder why we need these. Doesn't the limits argument

of the scales already allow us to control what appears on the plot? The key difference is how the limits work: when setting scale limits, any data outside the limits is thrown away; but when setting coordinate system limits we still use all the data, but we only display a small region of the plot. Setting coordinate system limits is like looking at the plot under a magnifying glass.

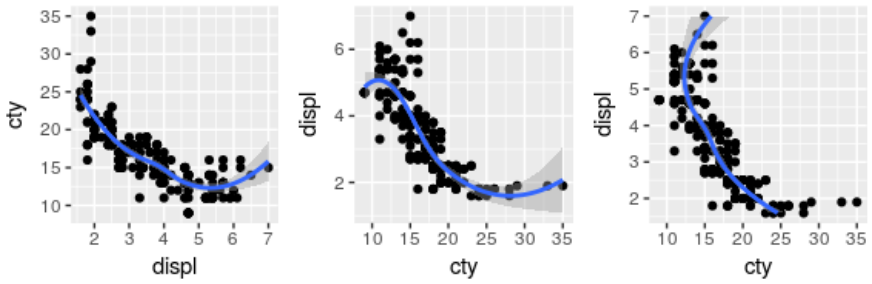
```
base <- ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth()

# Full dataset
base
#> `geom_smooth()` using method = 'loess'
# Scaling to 5--7 throws away data outside that range
base + scale_x_continuous(limits = c(5, 7))
#> `geom_smooth()` using method = 'loess'
#> Warning: Removed 196 rows containing non-finite values
#> (stat_smooth).
#> Warning: Removed 196 rows containing missing values (geom_point).
# Zooming to 5--7 keeps all the data but only shows some of it
base + coord_cartesian(xlim = c(5, 7))
#> `geom_smooth()` using method = 'loess'
```



7.4.2 Flipping the axes with `coord_flip()`

Most statistics and geoms assume you are interested in y values conditional on x values (e.g., smooth, summary, boxplot, line): in most statistical models, the x values are assumed to be measured without error. If you are interested in x conditional on y (or you just want to rotate the plot 90 degrees), you can use `coord_flip()` to exchange the x and y axes. Compare this with just exchanging the variables mapped to x and y:

```
ggplot(mpg, aes(displ, cty)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess'
# Exchanging cty and displ rotates the plot 90 degrees, but the smooth
# is fit to the rotated data.
ggplot(mpg, aes(cty, displ)) +
  geom_point() +
  geom_smooth()
#> `geom_smooth()` using method = 'loess'
# coord_flip() fits the smooth to the original data, and then rotates
# the output
ggplot(mpg, aes(displ, cty)) +
  geom_point() +
  geom_smooth() +
  coord_flip()
#> `geom_smooth()` using method = 'loess'
```



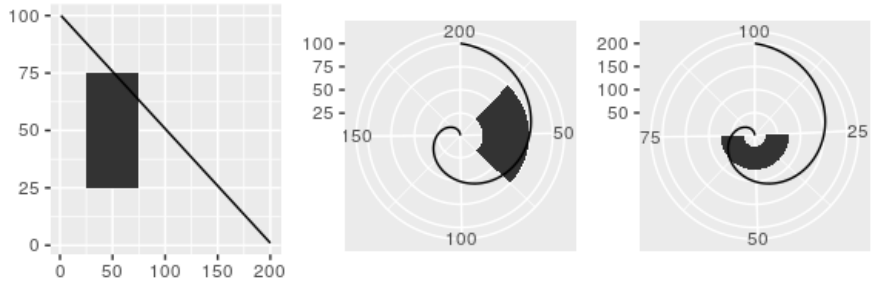
7.4.3 Equal scales with `coord_fixed()`

`coord_fixed()` fixes the ratio of length on the x and y axes. The default `ratio` ensures that the x and y axes have equal scales: i.e., 1 cm along the x axis represents the same range of data as 1 cm along the y axis. The aspect ratio will also be set to ensure that the mapping is maintained regardless of the shape of the output device. See the documentation of `coord_fixed()` for more details.

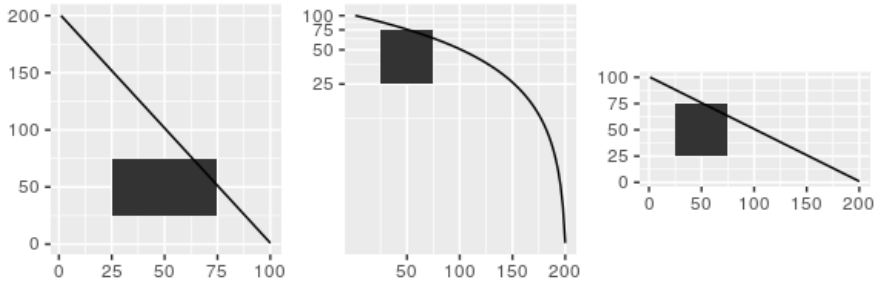
7.5 Non-linear coordinate systems

Unlike linear coordinates, non-linear coordinates can change the shape of geoms. For example, in polar coordinates a rectangle becomes an arc; in a map projection, the shortest path between two points is not necessarily a straight line. The code below shows how a line and a rectangle are rendered in a few different coordinate systems.

```
rect <- data.frame(x = 50, y = 50)
line <- data.frame(x = c(1, 200), y = c(100, 1))
base <- ggplot(mapping = aes(x, y)) +
  geom_tile(data = rect, aes(width = 50, height = 50)) +
  geom_line(data = line) +
  xlab(NULL) + ylab(NULL)
#> Warning: Ignoring unknown aesthetics: width, height
base
base + coord_polar("x")
base + coord_polar("y")
```




```
base + coord_flip()
base + coord_trans(y = "log10")
base + coord_fixed()
```

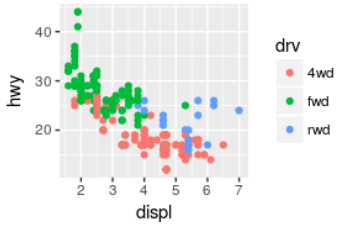


The transformation takes part in two steps. Firstly, the parameterisation of each geom is changed to be purely location-based, rather than location- and dimension-based. For example, a bar can be represented as an x position (a location), a height and a width (two dimensions). Interpreting height and width in a non-Cartesian coordinate system is hard because a rectangle may no longer have constant height and width, so we convert to a purely location-based representation, a polygon defined by the four corners. This effectively converts all geoms to a combination of points, lines and polygons.

Once all geoms have a location-based representation, the next step is to transform each location into the new coordinate system. It is easy to transform points, because a point is still a point no matter what coordinate system you are in. Lines and polygons are harder, because a straight line may no longer be straight in the new coordinate system. To make the problem tractable we assume that all coordinate transformations are smooth, in the sense that all very short lines will still be very short straight lines in the new coordinate system. With this assumption in hand, we can transform lines and polygons by breaking them up into many small line segments and transforming each segment. This process is called munching and is illustrated below:

1. We start with a line parameterised by its two endpoints:

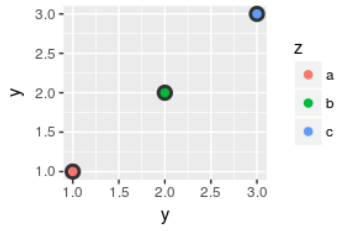
```
df <- data.frame(r = c(0, 1), theta = c(0, 3 / 2 * pi))
ggplot(df, aes(r, theta)) +
  geom_line() +
  geom_point(size = 2, colour = "red")
```



2. We break it into multiple line segments, each with two endpoints.

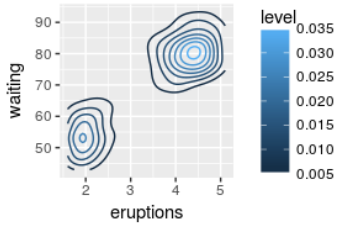
```
interp <- function(rng, n) {
  seq(rng[1], rng[2], length = n)
}
munched <- data.frame(
  r = interp(df$r, 15),
  theta = interp(df$theta, 15)
)
```

```
ggplot(munched, aes(r, theta)) +
  geom_line() +
  geom_point(size = 2, colour = "red")
```



3. We transform the locations of each piece:

```
transformed <- transform(munched,
  x = r * sin(theta),
  y = r * cos(theta)
)

ggplot(transformed, aes(x, y)) +
  geom_path() +
  geom_point(size = 2, colour = "red") +
  coord_fixed()
```

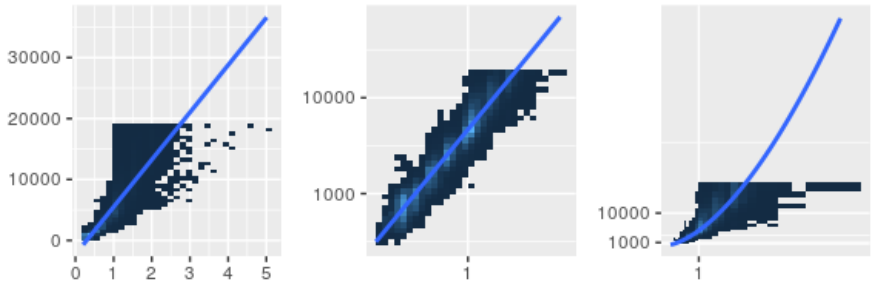


Internally ggplot2 uses many more segments so that the result looks smooth.

7.5.1 Transformations with `coord_trans()`

Like limits, we can also transform the data in two places: at the scale level or at the coordinate system level. `coord_trans()` has arguments `x` and `y` which should be strings naming the transformer or transformer objects (see Section 6.6.1). Transforming at the scale level occurs before statistics are computed and does not change the shape of the geom. Transforming at the coordinate system level occurs after the statistics have been computed, and does affect the shape of the geom. Using both together allows us to model the data on a transformed scale and then backtransform it for interpretation: a common pattern in analysis.

```
# Linear model on original scale is poor fit
base <- ggplot(diamonds, aes(carat, price)) +
  stat_bin2d() +
  geom_smooth(method = "lm") +
  xlab(NULL) +
  ylab(NULL) +
  theme(legend.position = "none")
base

# Better fit on log scale, but harder to interpret
base +
  scale_x_log10() +
  scale_y_log10()

# Fit on log scale, then backtransform to original.
# Highlights lack of expensive diamonds with large carats
pow10 <- scales::exp_trans(10)
base +
  scale_x_log10() +
  scale_y_log10() +
  coord_trans(x = pow10, y = pow10)
```

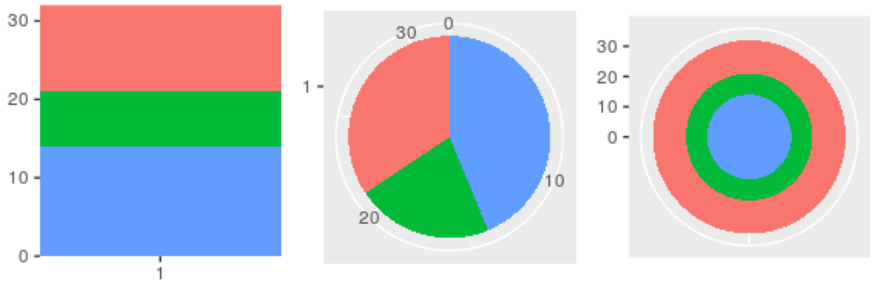


7.5.2 Polar coordinates with `coord_polar()`

Using polar coordinates gives rise to pie charts and wind roses (from bar geoms), and radar charts (from line geoms). Polar coordinates are often used for circular data, particularly time or direction, but the perceptual properties are not good because the angle is harder to perceive for small radii than it is for large radii. The `theta` argument determines which position variable is mapped to angle (by default, x) and which to radius.

The code below shows how we can turn a bar into a pie chart or a bullseye chart by changing the coordinate system. The documentation includes other examples.

```
base <- ggplot(mtcars, aes(factor(1), fill = factor(cyl))) +
  geom_bar(width = 1) +
  theme(legend.position = "none") +
  scale_x_discrete(NULL, expand = c(0, 0)) +
  scale_y_continuous(NULL, expand = c(0, 0))

# Stacked barchart
base

# Pie chart
base + coord_polar(theta = "y")

# The bullseye chart
base + coord_polar()
```

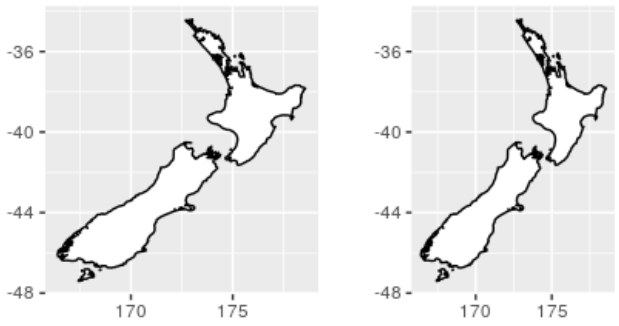


7.5.3 Map projections with `coord_map()`

Maps are intrinsically displays of spherical data. Simply plotting raw longitudes and latitudes is misleading, so we must *project* the data. There are two ways to do this with ggplot2:

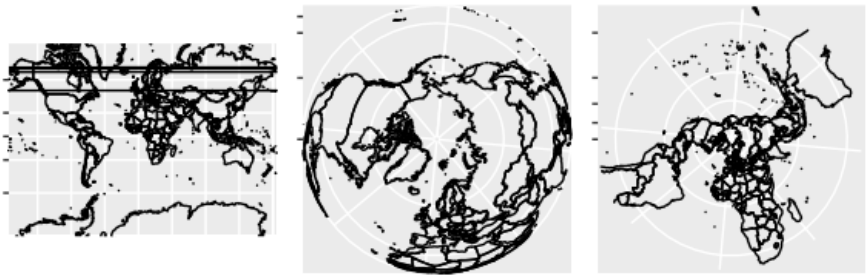
- `coord_quickmap()` is a quick and dirty approximation that sets the aspect ratio to ensure than 1m of latitude and 1m of longitude are the same distance in the middle of the plot. These is a reasonable place to start for smaller regions, and is very faster.

```
# Prepare a map of NZ
nzmap <- ggplot(map_data("nz"), aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black") +
  xlab(NULL) + ylab(NULL)

# Plot it in cartesian coordinates
nzmap
# With the aspect ratio approximation
nzmap + coord_quickmap()
```



- `coord_map()` uses the **mapproj** package, `https://cran.r-project.org/package=mapproj` to do a formal map projection. It takes the same arguments as `mapproj::mapproject()` for controlling the projection. It is much slower than `coord_quickmap()` because it must munch the data and transform each piece.

```
world <- map_data("world")
worldmap <- ggplot(world, aes(long, lat, group = group)) +
  geom_path() +
  scale_y_continuous(NULL, breaks = (-2:3) * 30, labels = NULL) +
  scale_x_continuous(NULL, breaks = (-4:4) * 45, labels = NULL)

worldmap + coord_map()
# Some crazier projections
worldmap + coord_map("ortho")
worldmap + coord_map("stereographic")
```



Chapter 8
Themes

8.1 Introduction

In this chapter you will learn how to use the ggplot2 theme system, which allows you to exercise fine control over the non-data elements of your plot. The theme system does not affect how the data is rendered by geoms, or how it is transformed by scales. Themes don't change the perceptual properties of the plot, but they do help you make the plot aesthetically pleasing or match an existing style guide. Themes give you control over things like fonts, ticks, panel strips, and backgrounds.

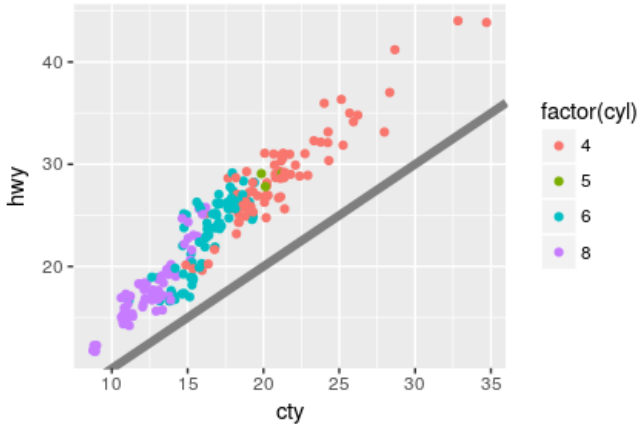
This separation of control into data and non-data parts is quite different from base and lattice graphics. In base and lattice graphics, most functions take a large number of arguments that specify both data and non-data appearance, which makes the functions complicated and harder to learn. ggplot2 takes a different approach: when creating the plot you determine how the data is displayed, then *after* it has been created you can edit every detail of the rendering, using the theming system.

The theming system is composed of four main components:

- Theme **elements** specify the non-data elements that you can control. For example, the `plot.title` element controls the appearance of the plot title; `axis.ticks.x`, the ticks on the x axis; `legend.key.height`, the height of the keys in the legend.
- Each element is associated with an **element function**, which describes the visual properties of the element. For example, `element_text()` sets the font size, colour and face of text elements like `plot.title`.
- The `theme()` function which allows you to override the default theme elements by calling element functions, like `theme(plot.title = element_text(colour = "red"))`.
- Complete **themes**, like `theme_grey()` set all of the theme elements to values designed to work together harmoniously.

For example, imagine you've made the following plot of your data.

```
base <- ggplot(mpg, aes(cty, hwy, color = factor(cyl))) +
  geom_jitter() +
  geom_abline(colour = "grey50", size = 2)
base
```

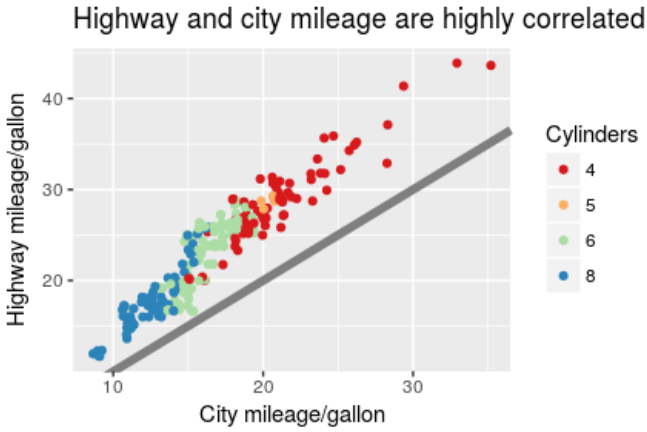


It's served its purpose for you: you've learned that `cty` and `hwy` are highly correlated, both are tightly coupled with `cyl`, and that `hwy` is always greater than `cty` (and the difference increases as `cty` increases). Now you want to share the plot with others, perhaps by publishing it in a paper. That requires some changes. First, you need to make sure the plot can stand alone by:

- Improving the axes and legend labels.
- Adding a title for the plot.
- Tweaking the colour scale.

Fortunately you know how to do that already because you've read Chapter 6:

```
labelled <- base +
  labs(
    x = "City mileage/gallon",
    y = "Highway mileage/gallon",
    colour = "Cylinders",
    title = "Highway and city mileage are highly correlated"
  ) +
  scale_colour_brewer(type = "seq", palette = "Spectral")
labelled
```

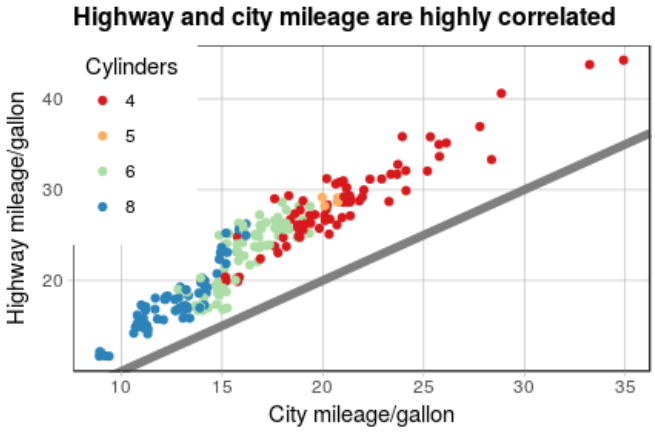


Next, you need to make sure the plot matches the style guidelines of your journal:

- The background should be white, not pale grey.
- The legend should be placed inside the plot if there's room.
- Major gridlines should be a pale grey and minor gridlines should be removed.
- The plot title should be 12pt bold text.

In this chapter, you'll learn how to use the theming system to make those changes, as shown below:

```
styled <- labelled +
  theme_bw() +
  theme(
    plot.title = element_text(face = "bold", size = 12),
    legend.background = element_rect(fill = "white", size = 4, colour = "white"),
    legend.justification = c(0, 1),
    legend.position = c(0, 1),
    axis.ticks = element_line(colour = "grey70", size = 0.2),
    panel.grid.major = element_line(colour = "grey70", size = 0.2),
    panel.grid.minor = element_blank()
  )
styled
```



Finally, the journal wants the figure as a 600 dpi TIFF file. You'll learn the fine details of `ggsave()` in Section 8.5.

8.2 Complete themes

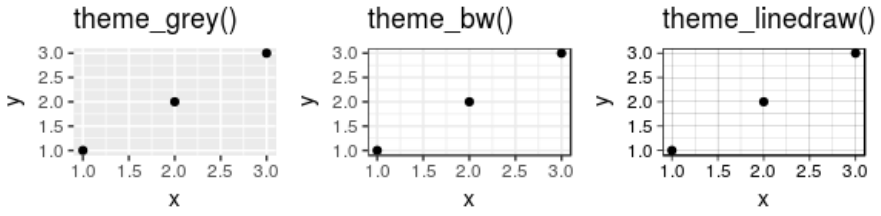
ggplot2 comes with a number of built in themes. The most important is `theme_grey()`, the signature ggplot2 theme with a light grey background and white gridlines. The theme is designed to put the data forward while supporting comparisons, following the advice of (Tufte 2006; Brewer 1994; Carr 2002; Carr 1994; Carr and Sun 1999). We can still see the gridlines to aid in the judgement of position (Cleveland 1993), but they have little visual impact and we can easily 'tune' them out. The grey background gives the plot a similar typographic colour to the text, ensuring that the graphics fit in with the flow of a document without jumping out with a bright white background. Finally, the grey background creates a continuous field of colour which ensures that the plot is perceived as a single visual entity.

There are seven other themes built in to ggplot2 1.1.0:

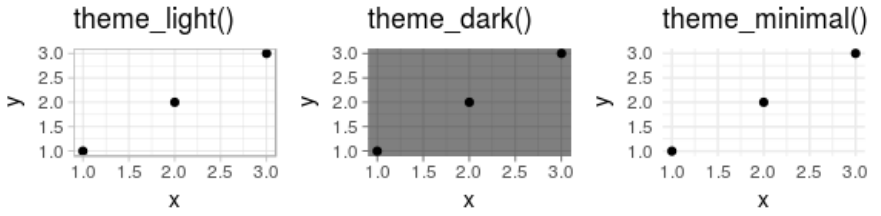
- `theme_bw()`: a variation on `theme_grey()` that uses a white background and thin grey grid lines.
- `theme_linedraw()`: A theme with only black lines of various widths on white backgrounds, reminiscent of a line drawing.
- `theme_light()`: similar to `theme_linedraw()` but with light grey lines and axes, to direct more attention towards the data.
- `theme_dark()`: the dark cousin of `theme_light()`, with similar line sizes but a dark background. Useful to make thin coloured lines pop out.
- `theme_minimal()`: A minimalistic theme with no background annotations.

- `theme_classic()`: A classic-looking theme, with x and y axis lines and no gridlines.
- `theme_void()`: A completely empty theme.

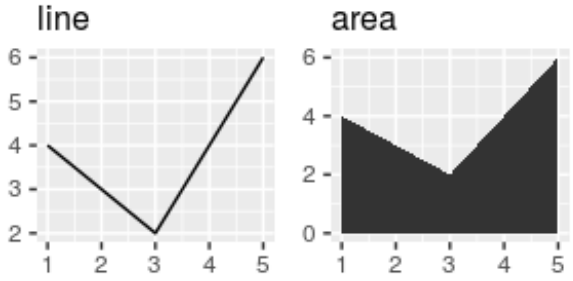
```
df <- data.frame(x = 1:3, y = 1:3)
base <- ggplot(df, aes(x, y)) + geom_point()
base + theme_grey() + ggtitle("theme_grey()")
base + theme_bw() + ggtitle("theme_bw()")
base + theme_linedraw() + ggtitle("theme_linedraw()")
```




```
base + theme_light() + ggtitle("theme_light()")
base + theme_dark() + ggtitle("theme_dark()")
base + theme_minimal()  + ggtitle("theme_minimal()")
```




```
base + theme_classic() + ggtitle("theme_classic()")
base + theme_void() + ggtitle("theme_void()")
```

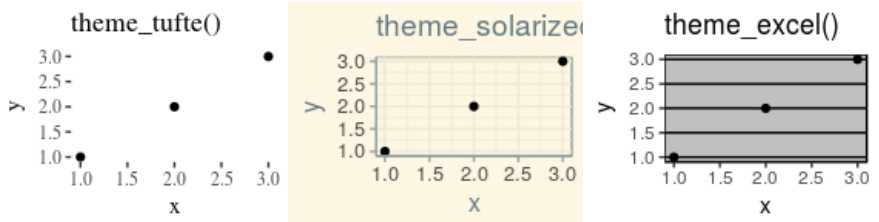


All themes have a `base_size` parameter which controls the base font size. The base font size is the size that the axis titles use: the plot title is usually bigger (1.2x), and the tick and strip labels are smaller (0.8x). If you want to control these sizes separately, you'll need to modify the individual elements as described below.

As well as applying themes a plot at a time, you can change the default theme with `theme_set()`. For example, if you really hate the default grey background, run `theme_set(theme_bw())` to use a white background for all plots.

You're not limited to the themes built-in to ggplot2. Other packages, like ggthemes by Jeffrey Arnold, add even more. Here's a few of my favourites from ggthemes:

```
library(ggthemes)
base + theme_tufte() + ggtitle("theme_tufte()")
base + theme_solarized() + ggtitle("theme_solarized()")
base + theme_excel() + ggtitle("theme_excel()") # ;)
```



The complete themes are a great place to start but don't give you a lot of control. To modify individual elements, you need to use `theme()` to override the default setting for an element with an element function.

8.2.1 Exercises

1. Try out all the themes in ggthemes. Which do you like the best?
2. What aspects of the default theme do you like? What don't you like? What would you change?
3. Look at the plots in your favourite scientific journal. What theme do they most resemble? What are the main differences?

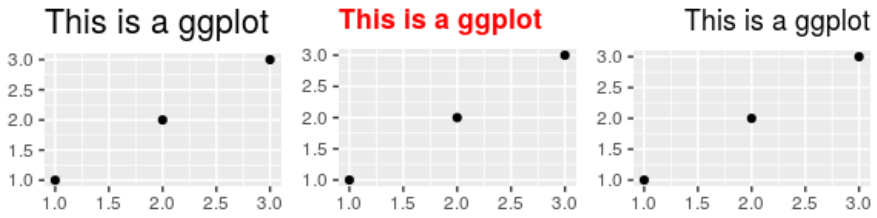
8.3 Modifying theme components

To modify an individual theme component you use code like `plot + theme(element.name = element_function())`. In this section you'll learn about the basic element functions, and then in the next section, you'll see all the elements that you can modify.

There are four basic types of built-in element functions: text, lines, rectangles, and blank. Each element function has a set of parameters that control the appearance:

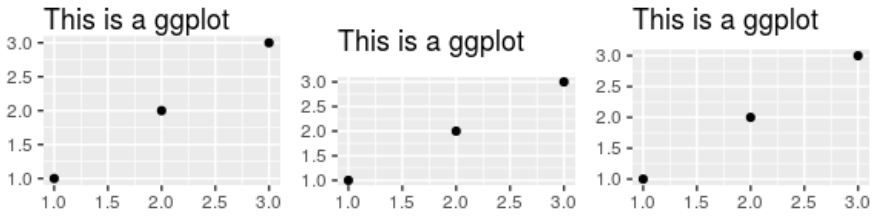
- `element_text()` draws labels and headings. You can control the font `family`, `face`, `colour`, `size` (in points), `hjust`, `vjust`, `angle` (in degrees) and `lineheight` (as ratio of `fontcase`). More details on the parameters can be found in `vignette("ggplot2-specs")`. Setting the font face is particularly challenging.

```
base_t <- base + labs(title = "This is a ggplot") + xlab(NULL) + ylab(NULL)
base_t + theme(plot.title = element_text(size = 16))
base_t + theme(plot.title = element_text(face = "bold", colour = "red"))
base_t + theme(plot.title = element_text(hjust = 1))
```



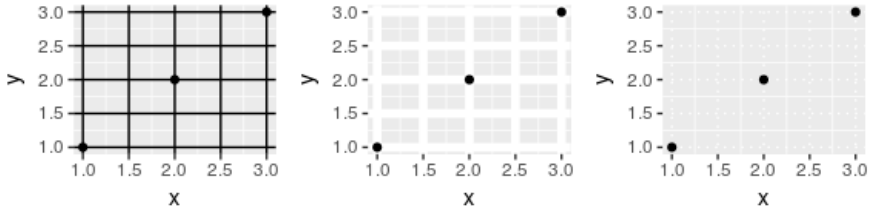
You can control the margins around the text with the `margin` argument and `margin()` function. `margin()` has four arguments: the amount of space (in points) to add to the top, right, bottom and left sides of the text. Any elements not specified default to 0.

```
# The margins here look asymmetric because there are also plot margins
base_t + theme(plot.title = element_text(margin = margin()))
base_t + theme(plot.title = element_text(margin = margin(t = 10, b = 10)))
base_t + theme(axis.title.y = element_text(margin = margin(r = 10)))
```



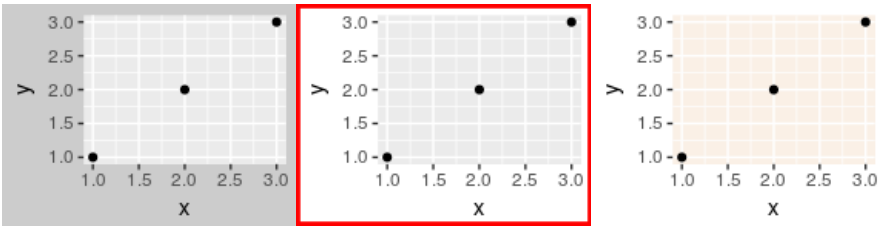
- `element_line()` draws lines parameterised by `colour`, `size` and `linetype`:

```
base + theme(panel.grid.major = element_line(colour = "black"))
base + theme(panel.grid.major = element_line(size = 2))
base + theme(panel.grid.major = element_line(linetype = "dotted"))
```



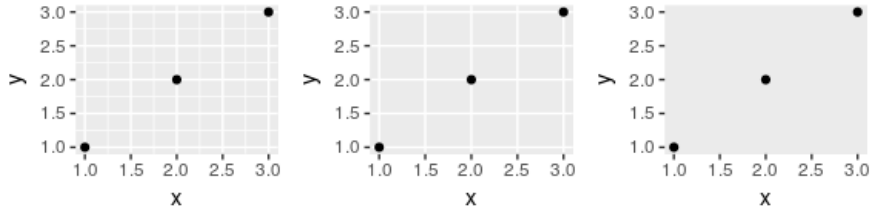
- `element_rect()` draws rectangles, mostly used for backgrounds, parameterised by `fill` colour and border `colour`, `size` and `linetype`.

```
base + theme(plot.background = element_rect(fill = "grey80", colour = NA))
base + theme(plot.background = element_rect(colour = "red", size = 2))
base + theme(panel.background = element_rect(fill = "linen"))
```

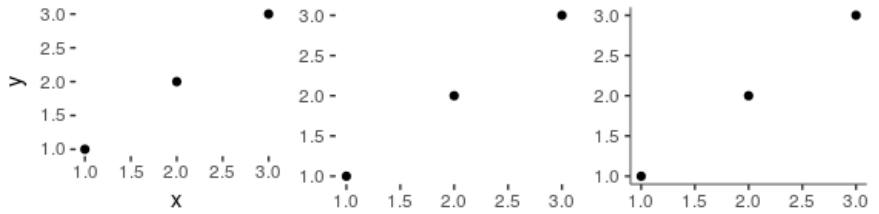


- `element_blank()` draws nothing. Use this if you don't want anything drawn, and no space allocated for that element. The following example uses `element_blank()` to progressively suppress the appearance of elements we're not interested in. Notice how the plot automatically reclaims the space previously used by these elements: if you don't want this to happen (perhaps because they need to line up with other plots on the page), use `colour = NA, fill = NA` to create invisible elements that still take up space.

```
base
last_plot() + theme(panel.grid.minor = element_blank())
last_plot() + theme(panel.grid.major = element_blank())
```



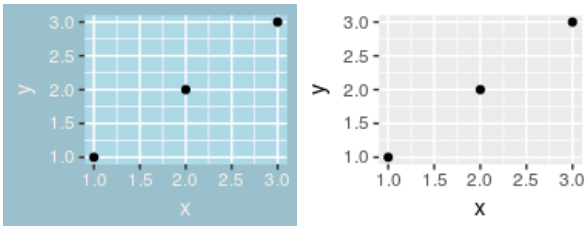

```
last_plot() + theme(panel.background = element_blank())
last_plot() + theme(
  axis.title.x = element_blank(),
  axis.title.y = element_blank()
)
last_plot() + theme(axis.line = element_line(colour = "grey50"))
```



- A few other settings take grid units. Create them with `unit(1, "cm")` or `unit(0.25, "in")`.

To modify theme elements for all future plots, use `theme_update()`. It returns the previous theme settings, so you can easily restore the original parameters once you're done.

```
old_theme <- theme_update(
  plot.background = element_rect(fill = "lightblue3", colour = NA),
  panel.background = element_rect(fill = "lightblue", colour = NA),
  axis.text = element_text(colour = "linen"),
  axis.title = element_text(colour = "linen")
)
base
theme_set(old_theme)
base
```



8.4 Theme elements

There are around 40 unique elements that control the appearance of the plot. They can be roughly grouped into five categories: plot, axis, legend, panel and facet. The following sections describe each in turn.

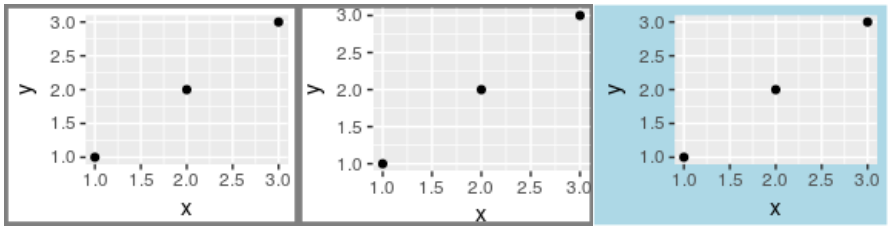
8.4.1 Plot elements

Some elements affect the plot as a whole:

Element	Setter	Description
plot.background	`element_rect()`	plot background
plot.title	`element_text()`	plot title
plot.margin	`margin()`	margins around plot

`plot.background` draws a rectangle that underlies everything else on the plot. By default, ggplot2 uses a white background which ensures that the plot is usable wherever it might end up (e.g. even if you save as a png and put on a slide with a black background). When exporting plots to use in other systems, you might want to make the background transparent with `fill = NA`. Similarly, if you're embedding a plot in a system that already has margins you might want to eliminate the built-in margins. Note that a small margin is still necessary if you want to draw a border around the plot.

```
base + theme(plot.background = element_rect(colour = "grey50", size = 2))
base + theme(
  plot.background = element_rect(colour = "grey50", size = 2),
  plot.margin = margin(2, 2, 2, 2)
)
base + theme(plot.background = element_rect(fill = "lightblue"))
```



8.4.2 Axis elements

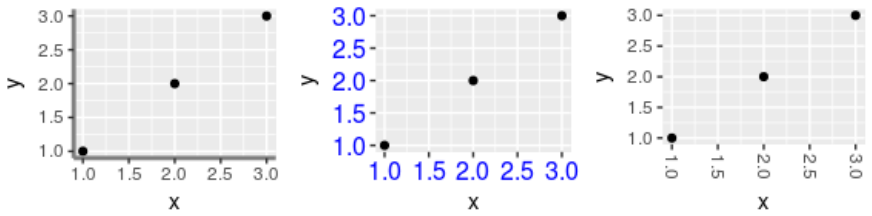
The axis elements control the apperance of the axes:

Element	Setter	Description
axis.line	`element_line()`	line parallel to axis (hidden in default themes)
axis.text	`element_text()`	tick labels
axis.text.x	`element_text()`	x-axis tick labels
axis.text.y	`element_text()`	y-axis tick labels
axis.title	`element_text()`	axis titles
axis.title.x	`element_text()`	x-axis title
axis.title.y	`element_text()`	y-axis title
axis.ticks	`element_line()`	axis tick marks
axis.ticks.length	`unit()`	length of tick marks

Note that `axis.text` (and `axis.title`) comes in three forms: `axis.text`, `axis.text.x`, and `axis.text.y`. Use the first form if you want to modify the

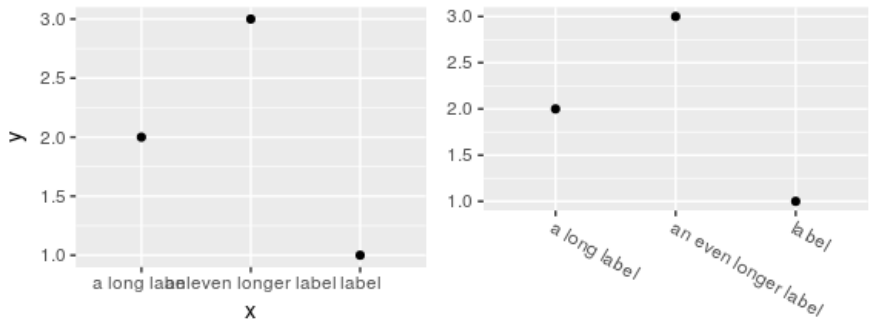
properties of both axes at once: any properties that you don't explicitly set in `axis.text.x` and `axis.text.y` will be inherited from `axis.text`.

```
df <- data.frame(x = 1:3, y = 1:3)
base <- ggplot(df, aes(x, y)) + geom_point()

# Accentuate the axes
base + theme(axis.line = element_line(colour = "grey50", size = 1))
# Style both x and y axis labels
base + theme(axis.text = element_text(color = "blue", size = 12))
# Useful for long labels
base + theme(axis.text.x = element_text(angle = -90, vjust = 0.5))
```



The most common adjustment is to rotate the x-axis labels to avoid long overlapping labels. If you do this, note negative angles tend to look best and you should set `hjust = 0` and `vjust = 1`:

```
df <- data.frame(
  x = c("label", "a long label", "an even longer label"),
  y = 1:3
)
base <- ggplot(df, aes(x, y)) + geom_point()
base
base +
  theme(axis.text.x = element_text(angle = -30, vjust = 1, hjust = 0)) +
  xlab(NULL) +
  ylab(NULL)
```



8.4.3 Legend elements

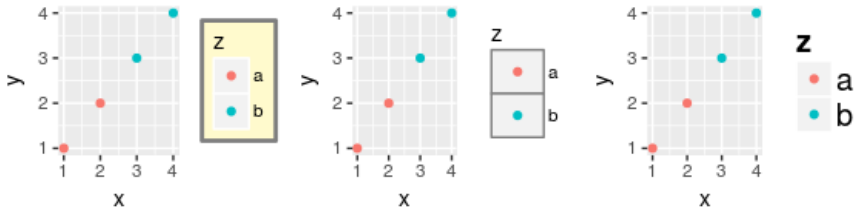
The legend elements control the apperance of all legends. You can also modify the appearance of individual legends by modifying the same elements in `guide_legend()` or `guide_colourbar()`.

Element	Setter	Description
legend.background	`element_rect()`	legend background
legend.key	`element_rect()`	background of legend keys
legend.key.size	`unit()`	legend key size
legend.key.height	`unit()`	legend key height
legend.key.width	`unit()`	legend key width
legend.margin	`unit()`	legend margin
legend.text	`element_text()`	legend labels
legend.text.align	0–1	legend label alignment (0 = right, 1 = left)
legend.title	`element_text()`	legend name
legend.title.align	0–1	legend name alignment (0 = right, 1 = left)

These options are illustrated below:

```
df <- data.frame(x = 1:4, y = 1:4, z = rep(c("a", "b"), each = 2))
base <- ggplot(df, aes(x, y, colour = z)) + geom_point()

base + theme(
  legend.background = element_rect(
    fill = "lemonchiffon",
    colour = "grey50",
    size = 1
  )
)
base + theme(
```

```
  legend.key = element_rect(color = "grey50"),
  legend.key.width = unit(0.9, "cm"),
  legend.key.height = unit(0.75, "cm")
)
base + theme(
  legend.text = element_text(size = 15),
  legend.title = element_text(size = 15, face = "bold")
)
```



There are four other properties that control how legends are laid out in the context of the plot (`legend.position`, `legend.direction`, `legend.justification`, `legend.box`). They are described in Section 6.4.2.

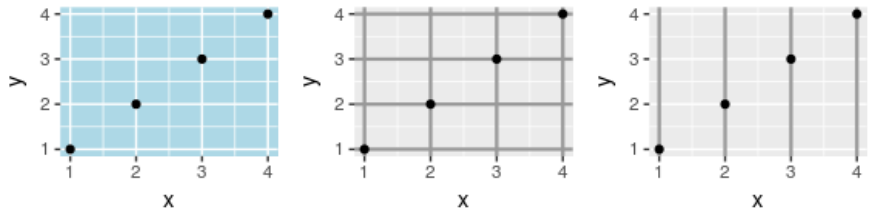
8.4.4 Panel elements

Panel elements control the appearance of the plotting panels:

Element	Setter	Description
panel.background	`element_rect()`	panel background (under data)
panel.border	`element_rect()`	panel border (over data)
panel.grid.major	`element_line()`	major grid lines
panel.grid.major.x	`element_line()`	vertical major grid lines
panel.grid.major.y	`element_line()`	horizontal major grid lines
panel.grid.minor	`element_line()`	minor grid lines
panel.grid.minor.x	`element_line()`	vertical minor grid lines
panel.grid.minor.y	`element_line()`	horizontal minor grid lines
aspect.ratio	numeric	plot aspect ratio

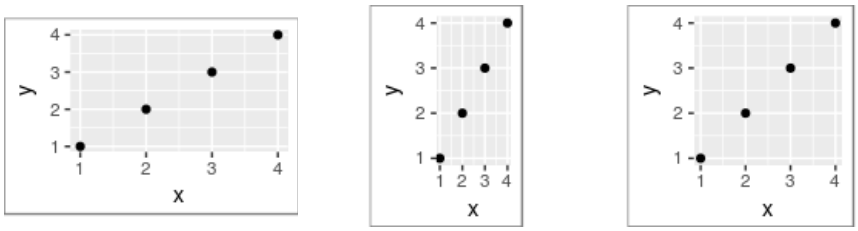
The main difference between `panel.background` and `panel.border` is that the background is drawn underneath the data, and the border is drawn on top of it. For that reason, you'll always need to assign `fill = NA` when overriding `panel.border`.

```
base <- ggplot(df, aes(x, y)) + geom_point()
# Modify background
base + theme(panel.background = element_rect(fill = "lightblue"))

# Tweak major grid lines
base + theme(
  panel.grid.major = element_line(color = "gray60", size = 0.8)
)
# Just in one direction
base + theme(
  panel.grid.major.x = element_line(color = "gray60", size = 0.8)
)
```



Note that aspect ratio controls the aspect ratio of the *panel*, not the overall plot:

```
base2 <- base + theme(plot.background = element_rect(colour = "grey50"))
# Wide screen
base2 + theme(aspect.ratio = 9 / 16)
# Long and skiny
base2 + theme(aspect.ratio = 2 / 1)
# Square
base2 + theme(aspect.ratio = 1)
```



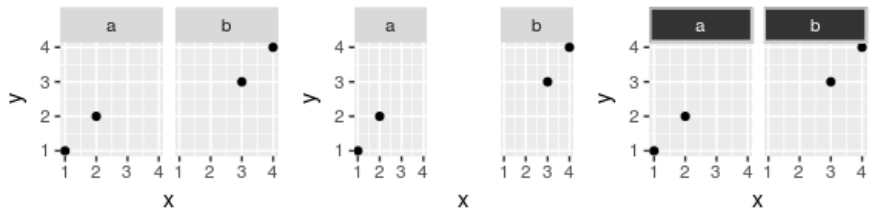
8.4.5 Facetting elements

The following theme elements are associated with faceted ggplots:

Element	Setter	Description
strip.background	`element_rect()`	background of panel strips
strip.text	`element_text()`	strip text
strip.text.x	`element_text()`	horizontal strip text
strip.text.y	`element_text()`	vertical strip text
panel.margin	`unit()`	margin between facets
panel.margin.x	`unit()`	margin between facets (vertical)
panel.margin.y	`unit()`	margin between facets (horizontal)

Element `strip.text.x` affects both `facet_wrap()` or `facet_grid()`; `strip.text.y` only affects `facet_grid()`.

```
df <- data.frame(x = 1:4, y = 1:4, z = c("a", "a", "b", "b"))
base_f <- ggplot(df, aes(x, y)) + geom_point() + facet_wrap(~z)

base_f
base_f + theme(panel.margin = unit(0.5, "in"))
#> Warning: `panel.margin` is deprecated. Please use `panel.spacing`
#> property instead
base_f + theme(
  strip.background = element_rect(fill = "grey20", color = "grey80", size = 1),
  strip.text = element_text(colour = "white")
)
```



8.4.6 Exercises

1. Create the ugliest plot possible! (Contributed by Andrew D. Steen, University of Tennessee - Knoxville)

2. `theme_dark()` makes the inside of the plot dark, but not the outside. Change the plot background to black, and then update the text settings so you can still read the labels.
3. Make an elegant theme that uses "linen" as the background colour and a serif font for the text.
4. Systematically explore the effects of `hjust` when you have a multiline title. Why doesn't `vjust` do anything?

8.5 Saving your output

When saving a plot to use in another program, you have two basic choices of output: raster or vector:

- Vector graphics describe a plot as sequence of operations: draw a line from (x_1, y_1) to (x_2, y_2), draw a circle at (x_3, x_4) with radius r. This means that they are effectively 'infinitely' zoomable; there is no loss of detail. The most useful vector graphic formats are pdf and svg.
- Raster graphics are stored as an array of pixel colours and have a fixed optimal viewing size. The most useful raster graphic format is png.

Figure 8.1 illustrates the basic differences in these formats for a circle. A good description is available at `http://tinyurl.com/rstrvctr`.

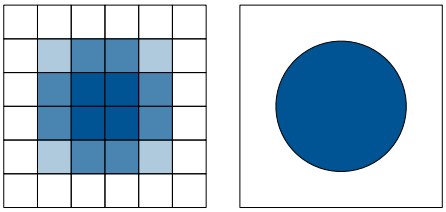


Fig. 8.1 The schematic difference between raster (left) and vector (right) graphics.

Unless there is a compelling reason not to, use vector graphics: they look better in more places. There are two main reasons to use raster graphics:

- You have a plot (e.g. a scatterplot) with thousands of graphical objects (i.e. points). A vector version will be large and slow to render.
- You want to embed the graphic in MS Office. MS has poor support for vector graphics (except for their own DrawingXML format which is not currently easy to make from R), so raster graphics are easier.

There are two ways to save output from ggplot2. You can use the standard R approach where you open a graphics device, generate the plot, then close the device:

```
pdf("output.pdf", width = 6, height = 6)
ggplot(mpg, aes(displ, cty)) + geom_point()
dev.off()
```

This works for all packages, but is verbose. ggplot2 provides a convenient shorthand with `ggsave()`:

```
ggplot(mpg, aes(displ, cty)) + geom_point()
ggsave("output.pdf")
```

`ggsave()` is optimised for interactive use: you can use it after you've drawn a plot. It has the following important arguments:

- The first argument, `path`, specifies the path where the image should be saved. The file extension will be used to automatically select the correct graphics device. `ggsave()` can produce `.eps`, `.pdf`, `.svg`, `.wmf`, `.png`, `.jpg`, `.bmp`, and `.tiff`.
- `width` and `height` control the output size, specified in inches. If left blank, they'll use the size of the on-screen graphics device.
- For raster graphics (i.e. `.png`, `.jpg`), the `dpi` argument controls the resolution of the plot. It defaults to 300, which is appropriate for most printers, but you may want to use 600 for particularly high-resolution output, or 96 for on-screen (e.g., web) display.

See `?ggsave` for more details.

References

Brewer, Cynthia A. 1994. "Color Use Guidelines for Mapping and Visualization." In *Visualization in Modern Cartography*, edited by A.M. MacEachren and D.R.F. Taylor, 123–47. Elsevier Science.

Carr, Dan. 1994. "Using Gray in Plots." *ASA Statistical Computing and Graphics Newsletter* 2 (5): 11–14. http://www.galaxy.gmu.edu/~dcarr/lib/v5n2.pdf.

———. 2002. "Graphical Displays." In *Encyclopedia of Environmetrics*, edited by Abdel H. El-Shaarawi and Walter W. Piegorsch, 2:933–60. John Wiley & Sons. http://www.galaxy.gmu.edu/~dcarr/lib/EnvironmentalGraphics.pdf.

Carr, Dan, and Ru Sun. 1999. "Using Layering and Perceptual Grouping in Statistical Graphics." *ASA Statistical Computing and Graphics Newsletter* 10 (1): 25–31.

Cleveland, William. 1993. "A Model for Studying Display Methods of Statistical Graphics." *Journal of Computational and Graphical Statistics* 2: 323–64. http://stat.bell-labs.com/doc/93.4.ps.

Tufte, Edward R. 2006. *Beautiful Evidence*. Graphics Press.

Part III
Data analysis

Chapter 9
Data analysis

9.1 Introduction

So far, every example in this book has started with a nice dataset that's easy to plot. That's great for learning (because you don't want to struggle with data handling while you're learning visualisation), but in real life, datasets hardly ever come in exactly the right structure. To use ggplot2 in practice, you'll need to learn some data wrangling skills. Indeed, in my experience, visualisation is often the easiest part of the data analysis process: once you have the right data, in the right format, aggregated in the right way, the right visualisation is often obvious.

The goal of this part of the book is to show you how to integrate ggplot2 with other tools needed for a complete data analysis:

- In this chapter, you'll learn the principles of tidy data (Wickham 2014), which help you organise your data in a way that makes it easy to visualise with ggplot2, manipulate with dplyr and model with the many modelling packages. The principles of tidy data are supported by the **tidyr** package, which helps you tidy messy datasets.
- Most visualisations require some data transformation whether it's creating a new variable from existing variables, or performing simple aggregations so you can see the forest for the trees. Chapter 10 will show you how to do this with the **dplyr** package.
- If you're using R, you're almost certainly using it for its fantastic modelling capabilities. While there's an R package for almost every type of model that you can think of, the results of these models can be hard to visualise. In Chapter 11, you'll learn about the **broom** package, by David Robinson, to convert models into tidy datasets so you can easily visualise them with ggplot2.

Tidy data is the foundation for data manipulation and visualising models. In the following sections, you'll learn the definition of tidy data, and the

tools you need to make messy data tidy. The chapter concludes with two case studies that show how to apply the tools in sequence to work with real(istic) data.

9.2 Tidy data

The principle behind tidy data is simple: storing your data in a consistent way makes it easier to work with it. Tidy data is a mapping between the statistical structure of a data frame (variables and observations) and the physical structure (columns and rows). Tidy data follows two main principles:

1. Variables go in columns.
2. Observations go in rows.

Tidy data is particularly important for ggplot2 because the job of ggplot2 is to map variables to visual properties: if your data isn't tidy, you'll have a hard time visualising it.

Sometimes you'll find a dataset that you have no idea how to plot. That's normally because it's not tidy. For example, take this data frame that contains monthly employment data for the United States:

```
ec2
#> # A tibble: 12 x 11
#>   month `2006` `2007` `2008` `2009` `2010` `2011` `2012` `2013`
#> * <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#> 1     1    8.6    8.3    9.0   10.7   20.0   21.6   21.0   16.2
#> 2     2    9.1    8.5    8.7   11.7   19.9   21.1   19.8   17.5
#> 3     3    8.7    9.1    8.7   12.3   20.4   21.5   19.2   17.7
#> 4     4    8.4    8.6    9.4   13.1   22.1   20.9   19.1   17.1
#> 5     5    8.5    8.2    7.9   14.2   22.3   21.6   19.9   17.0
#> 6     6    7.3    7.7    9.0   17.2   25.2   22.3   20.1   16.6
#> # ... with 6 more rows, and 2 more variables: `2014` <dbl>,
#> #   `2015` <dbl>
```

(If it looks familiar, it's because it's derived from the `economics` dataset that we used earlier in the book.)

Imagine you want to plot a time series showing how unemployment has changed over the last 10 years. Can you picture the ggplot2 command you'd need to do it? What if you wanted to focus on the seasonal component of unemployment by putting months on the x-axis and drawing one line for each year? It's difficult to see how to create those plots because the data is not tidy. There are three variables, month, year and unemployment rate, but each variable is stored in a different way:

- `month` is stored in a column.
- `year` is spread across the column names.
- `rate` is the value of each cell.

To make it possible to plot this data we first need to tidy it. There are two important pairs of tools:

- Spread & gather.
- Separate & unite.

9.3 Spread and gather

Take a look at the two tables below:

x	y	z
a	A	1
b	D	5
c	A	4
c	B	10
d	C	9

x	A	B	C	D
a	1	NA	NA	NA
b	NA	NA	NA	5
c	4	10	NA	NA
d	NA	NA	9	NA

If you study them for a little while, you'll notice that they contain the same data in different forms. I call the first form **indexed** data, because you look up a value using an index (the values of the `x` and `y` variables). I call the second form **Cartesian** data, because you find a value by looking at intersection of a row and a column. We can't tell if these datasets are tidy or not. Either form could be tidy depending on what the values "A", "B", "C", "D" mean.

(Also note the missing values: missing values that are explicit in one form may be implicit in the other. An `NA` is the presence of an absense; but sometimes a missing value is the absense of a presence.)

Tidying your data will often require translating Cartesian indexed forms, called **gathering**, and less commonly, indexed Cartesian, called **spreading**. The tidyr package provides the `spread()` and `gather()` functions to perform these operations, as described below.

(You can imagine generalising these ideas to higher dimensions. However, data is almost always stored in 2d (rows & columns), so these generalisations are fun to think about, but not that practical. I explore the idea more in Wickham (2007).

9.3.1 Gather

`gather()` has four main arguments:

- `data`: the dataset to translate.
- `key` & `value`: the key is the name of the variable that will be created from the column names, and the value is the name of the variable that will be created from the cell values.
- `...`: which variables to gather. You can specify individually, `A, B, C, D`, or as a range `A:D`. Alternatively, you can specify which columns are *not* to be gathered with `-`: `-E, -F`.

To tidy the economics dataset shown above, you first need to identify the variables: `year`, `month` and `rate`. `month` is already in a column, but `year` and `rate` are in Cartesian form, and we want them in indexed form, so we need to use `gather()`. In this example, the key is `year`, the value is `unemp` and we want to select columns from `2006` to `2015`:

```
gather(ec2, key = year, value = unemp, `2006`:`2015`)
#> # A tibble: 120 x 3
#>   month  year unemp
#>   <dbl> <chr> <dbl>
#> 1     1  2006   8.6
#> 2     2  2006   9.1
#> 3     3  2006   8.7
#> 4     4  2006   8.4
#> 5     5  2006   8.5
#> 6     6  2006   7.3
#> # ... with 114 more rows
```

Note that the columns have names that are not standard variable names in R (they don't start with a letter). This means that we need to surround them in backticks, i.e. `` `2006` `` to refer to them.

Alternatively, we could gather all columns except `month`:

```
gather(ec2, key = year, value = unemp, -month)
#> # A tibble: 120 x 3
#>   month  year unemp
#>   <dbl> <chr> <dbl>
#> 1     1  2006   8.6
```

```
#> 2     2  2006   9.1
#> 3     3  2006   8.7
#> 4     4  2006   8.4
#> 5     5  2006   8.5
#> 6     6  2006   7.3
#> # ... with 114 more rows
```

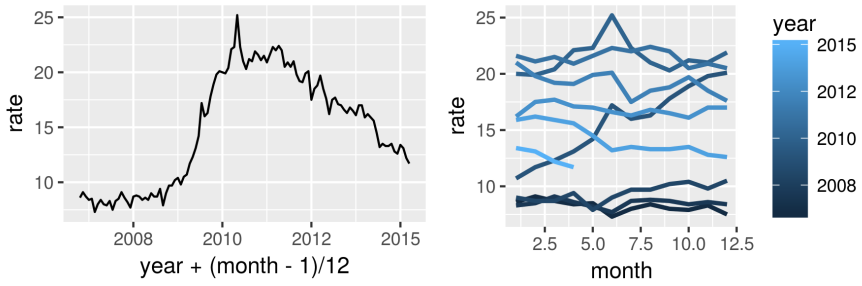
To be most useful, we can provide two extra arguments:

```
economics_2 <- gather(ec2, year, rate, `2006`:`2015`,
  convert = TRUE, na.rm = TRUE)
economics_2
#> # A tibble: 112 x 3
#>   month  year  rate
#> * <dbl> <int> <dbl>
#> 1     1  2006   8.6
#> 2     2  2006   9.1
#> 3     3  2006   8.7
#> 4     4  2006   8.4
#> 5     5  2006   8.5
#> 6     6  2006   7.3
#> # ... with 106 more rows
```

We use `convert = TRUE` to automatically convert the years from character strings to numbers, and `na.rm = TRUE` to remove the months with no data. (In some sense the data isn't actually missing because it represents dates that haven't occurred yet.)

When the data is in this form, it's easy to visualise in many different ways. For example, we can choose to emphasise either long term trend or seasonal variations:

```
ggplot(economics_2, aes(year + (month - 1) / 12, rate)) +
  geom_line()

ggplot(economics_2, aes(month, rate, group = year)) +
  geom_line(aes(colour = year), size = 1)
```



9.3.2 Spread

`spread()` is the opposite of `gather()`. You use it when you have a pair of columns that are in indexed form, instead of Cartesian form. For example, the following example dataset contains three variables (`day`, `rain` and `temp`), but `rain` and `temp` are stored in indexed form.

```
weather <- dplyr::data_frame(
  day = rep(1:3, 2),
  obs = rep(c("temp", "rain"), each = 3),
  val = c(c(23, 22, 20), c(0, 0, 5))
)
weather
#> # A tibble: 6 x 3
#>     day   obs   val
#>   <int> <chr> <dbl>
#> 1     1  temp    23
#> 2     2  temp    22
#> 3     3  temp    20
#> 4     1  rain     0
#> 5     2  rain     0
#> 6     3  rain     5
```

Spread allows us to turn this messy indexed form into a tidy Cartesian form. It shares many of the arguments with `gather()`. You'll need to supply the `data` to translate, as well as the name of the `key` column which gives the variable names, and the `value` column which contains the cell values. Here the key is `obs` and the value is `val`:

```
spread(weather, key = obs, value = val)
#> # A tibble: 3 x 3
#>     day  rain  temp
#> * <int> <dbl> <dbl>
#> 1     1     0    23
#> 2     2     0    22
#> 3     3     5    20
```

9.3.3 Exercises

1. How can you translate each of the initial example datasets into the other form?
2. How can you convert back and forth between the `economics` and `economics_long` datasets built into ggplot2?

3. Install the EDAWR package from `https://github.com/rstudio/EDAWR`. Tidy the `storms`, `population` and `tb` datasets.

9.4 Separate and unite

Spread and gather help when the variables are in the wrong place in the dataset. Separate and unite help when multiple variables are crammed into one column, or spread across multiple columns.

For example, the following dataset stores some information about the response to a medical treatment. There are three variables (time, treatment and value), but time and treatment are jammed in one variable together:

```
trt <- dplyr::data_frame(
  var = paste0(rep(c("beg", "end"), each = 3), "_", rep(c("a", "b", "c"))),
  val = c(1, 4, 2, 10, 5, 11)
)
trt
#> # A tibble: 6 x 2
#>     var   val
#>   <chr> <dbl>
#> 1 beg_a     1
#> 2 beg_b     4
#> 3 beg_c     2
#> 4 end_a    10
#> 5 end_b     5
#> 6 end_c    11
```

The `separate()` function makes it easy to tease apart multiple variables stored in one column. It takes four arguments:

- `data`: the data frame to modify.
- `col`: the name of the variable to split into pieces.
- `into`: a character vector giving the names of the new variables.
- `sep`: a description of how to split the variable apart. This can either be a regular expression, e.g. `_` to split by underscores, or `[^a-z]` to split by any non-letter, or an integer giving a position.

In this case, we want to split by the `_` character:

```
separate(trt, var, c("time", "treatment"), "_")
#> # A tibble: 6 x 3
#>    time treatment   val
#> * <chr>     <chr> <dbl>
#> 1   beg         a     1
#> 2   beg         b     4
```

```
#> 3   beg         c     2
#> 4   end         a    10
#> 5   end         b     5
#> 6   end         c    11
```

(If the variables are combined in a more complex form, have a look at `extract()`. Alternatively, you might need to create columns individually yourself using other calculations. A useful tool for this is `mutate()` which you'll learn about in the next chapter.)

`unite()` is the inverse of `separate()` - it joins together multiple columns into one column. This is less common, but it's useful to know about as the inverse of `separate()`.

9.4.1 Exercises

1. Install the EDAWR package from `https://github.com/rstudio/EDAWR`. Tidy the `who` dataset.
2. Work through the demos included in the tidyr package (`demo(package = "tidyr")`)

9.5 Case studies

For most real datasets, you'll need to use more than one tidying verb. There many be multiple ways to get there, but as long as each step makes the data tidier, you'll eventually get to the tidy dataset. That said, you typically apply the functions in the same order: `gather()`, `separate()` and `spread()` (although you might not use all three).

9.5.1 Blood pressure

The first step when tidying a new dataset is always to identify the variables. Take the following simulated medical data. There are seven variables in this dataset: name, age, start date, week, systolic & diastolic blood pressure. Can you see how they're stored?

```
# Adapted from example by Barry Rowlingson,
# http://barryrowlingson.github.io/hadleyverse/
bpd <- readr::read_table(
"name age      start  week1  week2  week3
Anne  35 2014-03-27 100/80 100/75 120/90
```

```
 Ben  41 2014-03-09 110/65 100/65 135/70
Carl  33 2014-04-02 125/80   <NA>   <NA>
", na = "<NA>")
```

The first step is to convert from Cartesian to indexed form:

```
bpd_1 <- gather(bpd, week, bp, week1:week3)
bpd_1
#> # A tibble: 9 x 5
#>    name   age      start  week     bp
#>   <chr> <int>     <date> <chr>  <chr>
#> 1  Anne    35 2014-03-27 week1 100/80
#> 2   Ben    41 2014-03-09 week1 110/65
#> 3  Carl    33 2014-04-02 week1 125/80
#> 4  Anne    35 2014-03-27 week2 100/75
#> 5   Ben    41 2014-03-09 week2 100/65
#> 6  Carl    33 2014-04-02 week2   <NA>
#> # ... with 3 more rows
```

This is tidier, but we have two variables combined together in the `bp` variable. This is a common way of writing down the blood pressure, but analysis is easier if we break it into two variables. That's the job of separate:

```
bpd_2 <- separate(bpd_1, bp, c("sys", "dia"), "/")
bpd_2
#> # A tibble: 9 x 6
#>    name   age      start  week   sys   dia
#> * <chr> <int>     <date> <chr> <chr> <chr>
#> 1  Anne    35 2014-03-27 week1   100    80
#> 2   Ben    41 2014-03-09 week1   110    65
#> 3  Carl    33 2014-04-02 week1   125    80
#> 4  Anne    35 2014-03-27 week2   100    75
#> 5   Ben    41 2014-03-09 week2   100    65
#> 6  Carl    33 2014-04-02 week2  <NA>  <NA>
#> # ... with 3 more rows
```

This dataset is now tidy, but we could do a little more to make it easier to use. The following code uses `extract()` to pull the week number out into its own variable (using regular expressions is beyond the scope of the book, but `\\d` stands for any digit). I also use `arrange()` (which you'll learn about in the next chapter) to order the rows to keep the records for each person together.

```
bpd_3 <- extract(bpd_2, week, "week", "(\\d)", convert = TRUE)
bpd_4 <- dplyr::arrange(bpd_3, name, week)
bpd_4
#> # A tibble: 9 x 6
```

```
#>    name   age      start  week   sys   dia
#>   <chr> <int>     <date> <int> <chr> <chr>
#> 1  Anne    35 2014-03-27     1   100    80
#> 2  Anne    35 2014-03-27     2   100    75
#> 3  Anne    35 2014-03-27     3   120    90
#> 4   Ben    41 2014-03-09     1   110    65
#> 5   Ben    41 2014-03-09     2   100    65
#> 6   Ben    41 2014-03-09     3   135    70
#> # ... with 3 more rows
```

You might notice that there's some repetition in this dataset: if you know the name, then you also know the age and start date. This reflects a third condition of tidyness that I don't discuss here: each data frame should contain one and only one data set. Here there are really two datasets: information about each person that doesn't change over time, and their weekly blood pressure measurements. You can learn more about this sort of messiness in the resources mentioned at the end of the chapter.

9.5.2 Test scores

Imagine you're interested in the effect of an intervention on test scores. You've collected the following data. What are the variables?

```
# Adapted from http://stackoverflow.com/questions/29775461
scores <- dplyr::data_frame(
  person = rep(c("Greg", "Sally", "Sue"), each = 2),
  time   = rep(c("pre", "post"), 3),
  test1  = round(rnorm(6, mean = 80, sd = 4), 0),
  test2  = round(jitter(test1, 15), 0)
)
scores
#> # A tibble: 6 x 4
#>   person  time test1 test2
#>    <chr> <chr> <dbl> <dbl>
#> 1   Greg   pre    74    71
#> 2   Greg  post    81    80
#> 3  Sally   pre    70    69
#> 4  Sally  post    80    78
#> 5    Sue   pre    82    81
#> 6    Sue  post    85    82
```

I think the variables are person, test, pre-test score and post-test score. As usual, we start by converting columns in Cartesian form (`test1` and `test2`) to indexed form (`test` and `score`):

```
scores_1 <- gather(scores, test, score, test1:test2)
scores_1
#> # A tibble: 12 x 4
#>   person  time  test score
#>    <chr> <chr> <chr> <dbl>
#> 1   Greg   pre test1    74
#> 2   Greg  post test1    81
#> 3  Sally   pre test1    70
#> 4  Sally  post test1    80
#> 5    Sue   pre test1    82
#> 6    Sue  post test1    85
#> # ... with 6 more rows
```

Now we need to do the opposite: `pre` and `post` should be variables, not values, so we need to spread `time` and `score`:

```
scores_2 <- spread(scores_1, time, score)
scores_2
#> # A tibble: 6 x 4
#>   person  test  post   pre
#> *  <chr> <chr> <dbl> <dbl>
#> 1   Greg test1    81    74
#> 2   Greg test2    80    71
#> 3  Sally test1    80    70
#> 4  Sally test2    78    69
#> 5    Sue test1    85    82
#> 6    Sue test2    82    81
```

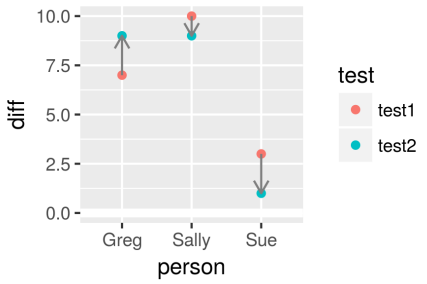
A good indication that we have made a tidy dataset is that it's now easy to calculate the statistic of interest: the difference between pre- and post-intervention scores:

```
scores_3 <- mutate(scores_2, diff = post - pre)
scores_3
#> # A tibble: 6 x 5
#>   person  test  post   pre  diff
#>    <chr> <chr> <dbl> <dbl> <dbl>
#> 1   Greg test1    81    74     7
#> 2   Greg test2    80    71     9
#> 3  Sally test1    80    70    10
#> 4  Sally test2    78    69     9
#> 5    Sue test1    85    82     3
#> 6    Sue test2    82    81     1
```

And it's similarly easy to plot:

```
ggplot(scores_3, aes(person, diff, color = test)) +
  geom_hline(size = 2, colour = "white", yintercept = 0) +
```

```
  geom_point() +
  geom_path(aes(group = person), colour = "grey50",
    arrow = arrow(length = unit(0.25, "cm")))
```



(Again, you'll learn about `mutate()` in the next chapter.)

9.6 Learning more

Data tidying is a big topic and this chapter only scratches the surface. I recommend the following references which go into considerably more depth on this topic:

- The tidyr documentation. I've described the most important arguments, but most functions have other arguments that help deal with less common situations. If you're struggling, make sure to read the documentation to see if there's an argument that might help you.
- "Tidy data (`http://www.jstatsoft.org/v59/i10/`)", an article in the *Journal of Statistical Software*. It describes the ideas of tidy data in more depth and shows other types of messy data. Unfortunately the paper was written before tidyr existed, so to see how to use tidyr instead of reshape2, consult the tidyr vignette (`http://cran.r-project.org/web/packages/tidyr/vignettes/tidy-data.html`).
- The data wrangling cheatsheet (`http://rstudio.com/cheatsheets`) by RStudio, includes the most common tidyr verbs in a form designed to jog your memory when you're stuck.

References

Wickham, Hadley. 2007. "Reshaping Data with the Reshape Package." *Journal of Statistical Software* 21 (12). http://www.jstatsoft.org/v21/i12/paper.

———. 2014. "Tidy Data." *The Journal of Statistical Software* 59. http://www.jstatsoft.org/v59/i10/.

Chapter 10
Data transformation

10.1 Introduction

Tidy data is important, but it's not the end of the road. Often you won't have quite the right variables, or your data might need a little aggregation before you visualise it. This chapter will show you how to solve these problems (and more!) with the **dplyr** package.

The goal of dplyr is to provide verbs (functions) that help you solve the most common 95% of data manipulation problems. dplyr is similar to ggplot2, but instead of providing a grammar of graphics, it provides a grammar of data manipulation. Like ggplot2, dplyr helps you not just by giving you functions, but it also helps you think about data manipulation. In particular, dplyr helps by constraining you: instead of struggling to think about which of the thousands of functions that might help, you can just pick from a handful that are design to be very likely to be helpful. In this chapter you'll learn four of the most important dplyr verbs:

- `filter()`
- `mutate()`
- `group_by()` & `summarise()`

These verbs are easy to learn because they all work the same way: they take a data frame as the first argument, and return a modified data frame. The other arguments control the details of the transformation, and are always interpreted in the context of the data frame so you can refer to variables directly. I'll also explain each in the same way: I'll show you a motivating example using the `diamonds` data, give you more details about how the function works, and finish up with some exercises for you to practice your skills with.

You'll also learn how to create data transformation pipelines using `%>%`. `%>%` plays a similar role to `+` in ggplot2: it allows you to solve complex problems by combining small pieces that are easily understood in isolation.

This chapter only scratches the surface of dplyr's capabilities but it should be enough to help you with visualisation problems. You can learn more by using the resources discussed at the end of the chapter.

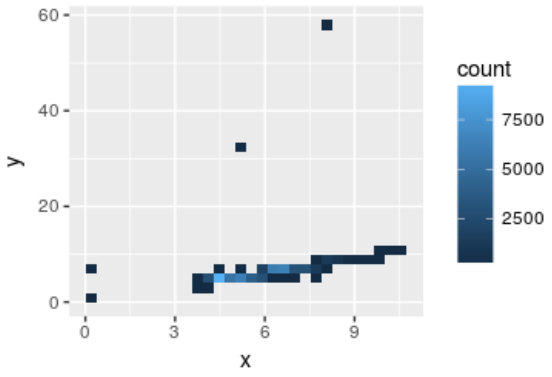
10.2 Filter observations

It's common to only want to explore one part of a dataset. A great data analysis strategy is to start with just one observation unit (one person, one city, etc), and understand how it works before attempting to generalise the conclusion to others. This is a great technique if you ever feel overwhelmed by an analysis: zoom down to a small subset, master it, and then zoom back out, to apply your conclusions to the full dataset.

Filtering is also useful for extracting outliers. Generally, you don't want to just throw outliers away, as they're often highly revealing, but it's useful to think about partitioning the data into the common and the unusual. You summarise the common to look at the broad trends; you examine the outliers individually to see if you can figure out what's going on.

For example, look at this plot that shows how the x and y dimensions of the diamonds are related:

```
ggplot(diamonds, aes(x, y)) +
  geom_bin2d()
```



There are around 50,000 points in this dataset: most of them lie along the diagonal, but there are a handful of outliers. One clear set of incorrect values are those diamonds with zero dimensions. We can use `filter()` to pull them out:

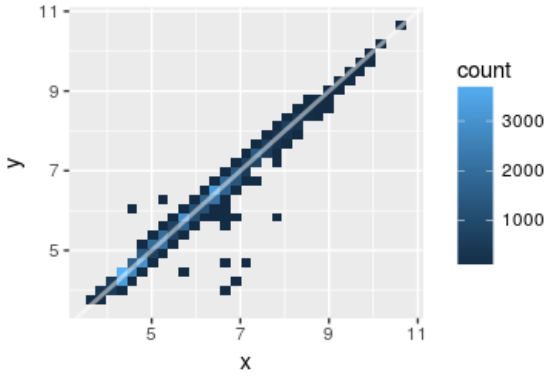
```
filter(diamonds, x == 0 | y == 0)
#> # A tibble: 8 x 10
#>   carat       cut color clarity depth table price     x     y
#>   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl>
#> 1  1.07     Ideal     F     SI2  61.6    56  4954     0  6.62
#> 2  1.00 Very Good     H     VS2  63.3    53  5139     0  0.00
#> 3  1.14      Fair     G     VS1  57.5    67  6381     0  0.00
#> 4  1.56     Ideal     G     VS2  62.2    54 12800     0  0.00
#> 5  1.20   Premium     D    VVS1  62.1    59 15686     0  0.00
#> 6  2.25   Premium     H     SI2  62.8    59 18034     0  0.00
#> 7  0.71      Good     F     SI2  64.1    60  2130     0  0.00
#> 8  0.71      Good     F     SI2  64.1    60  2130     0  0.00
#> # ... with 1 more variables: z <dbl>
```

This is equivalent to the base R code `diamonds[diamonds$x == 0 | diamonds$y == 0, ]`, but is more concise because `filter()` knows to look for the bare `x` in the data frame.

(If you've used `subset()` before, you'll notice that it has very similar behaviour. The biggest difference is that `subset()` can select both observations and variables, where in dplyr, `filter()` works exclusively with observations and `select()` with variables. There are some other subtle differences, but the main advantage to using `filter()` is that it behaves identically to the other dplyr verbs and it tends to be a bit faster than `subset()`.)

In a real analysis, you'd look at the outliers in more detail to see if you can find the root cause of the data quality problem. In this case, we're just going to throw them out and focus on what remains. To save some typing, we may provide multiple arguments to `filter()` which combines them.

```
diamonds_ok <- filter(diamonds, x > 0, y > 0, y < 20)
ggplot(diamonds_ok, aes(x, y)) +
  geom_bin2d() +
  geom_abline(slope = 1, colour = "white", size = 1, alpha = 0.5)
```



This plot is now more informative - we can see a very strong relationship between `x` and `y`. I've added the reference line to make it clear that for most diamonds, `x` and `y` are very similar. However, this plot still has problems:

- The plot is mostly empty, because most of the data lies along the diagonal.
- There are some clear bivariate outliers, but it's hard to select them with a simple filter.

We'll solve both of these problem in the next section by adding a new variable that's a transformation of x and y. But before we continue on to that, let's talk more about the details of `filter()`.

10.2.1 Useful tools

The first argument to `filter()` is a data frame. The second and subsequent arguments must be logical vectors: `filter()` selects every row where all the logical expressions are `TRUE`. The logical vectors must always be the same length as the data frame: if not, you'll get an error. Typically you create the logical vector with the comparison operators:

- `x == y`: x and y are equal.
- `x != y`: x and y are not equal.
- `x %in% c("a", "b", "c")`: `x` is one of the values in the right hand side.
- `x > y`, `x >= y`, `x < y`, `x <= y`: greater than, greater than or equal to, less than, less than or equal to.

And combine them with logical operators:

- `!x` (pronounced "not x"), flips `TRUE` and `FALSE` so it keeps all the values where `x` is `FALSE`.
- `x & y`: `TRUE` if both `x` and `y` are `TRUE`.
- `x | y`: `TRUE` if either `x` or `y` (or both) are `TRUE`.
- `xor(x, y)`: `TRUE` if either `x` or `y` are `TRUE`, but not both (exclusive or).

Most real queries involve some combination of both:

- Price less than $500: `price < 500`
- Size between 1 and 2 carats: `carat >= 1 & carat < 2`
- Cut is ideal or premium: `cut == "Premium" | cut == "Ideal"`, or `cut %in% c("Premium", "Ideal")` (note that R is case sensitive)
- Worst colour, cut and clarity: `cut == "Fair" & color == "J" & clarity == "SI2"`

You can also use functions in the filtering expression:

- Size is between 1 and 2 carats: `floor(carat) == 1`

- An average dimension greater than 3: `(x + y + z) / 3 > 3`

This is useful for simple expressions, but as things get more complicated it's better to create a new variable first so you can check that you've done the computation correctly before doing the subsetting. You'll learn how to do that in the next section.

The rules for `NA` are a bit trickier, so I'll explain them next.

10.2.2 Missing values

`NA`, R's missing value indicator, can be frustrating to work with. R's underlying philosophy is to force you to recognise that you have missing values, and make a deliberate choice to deal with them: missing values never silently go missing. This is a pain because you almost always want to just get rid of them, but it's a good principle to force you to think about the correct option.

The most important thing to understand about missing values is that they are infectious: with few exceptions, the result of any operation that includes a missing value will be a missing value. This happens because `NA` represents an unknown value, and there are few operations that turn an unknown value into a known value.

```
x <- c(1, NA, 2)
x == 1
#> [1]  TRUE    NA FALSE
x > 2
#> [1] FALSE    NA FALSE
x + 10
#> [1] 11 NA 12
```

When you first learn R, you might be tempted to find missing values using `==`:

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

But that doesn't work! A little thought reveals why: there's no reason why two unknown values should be the same. Instead, use `is.na(X)` to determine if a value is missing:

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

`filter()` only includes observations where all arguments are `TRUE`, so `NA` values are automatically dropped. If you want to include missing values, be explicit: `x > 10 | is.na(x)`. In other parts of R, you'll sometimes need to convert missing values into `FALSE`. You can do that with `x > 10 & !is.na(x)`

10.2.3 Exercises

1. Practice your filtering skills by:

   - Finding all the diamonds with equal x and y dimensions.
   - A depth between 55 and 70.
   - A carat smaller than the median carat.
   - Cost more than $10,000 per carat
   - Are of good or better quality

2. Fill in the question marks in this table:

Expression	`TRUE`	`FALSE`	`NA`
`x`	x		
?		x	
`is.na(x)`			x
`!is.na(x)`	?	?	?
?	x		x
?		x	x

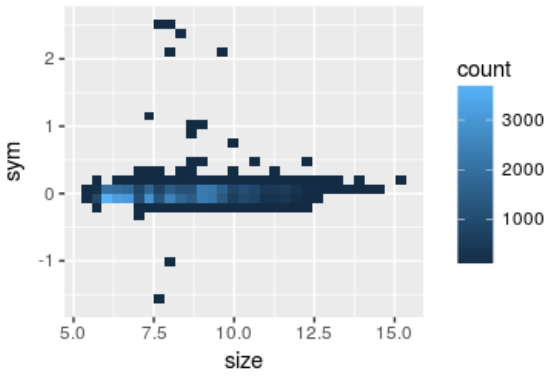
3. Repeat the analysis of outlying values with `z`. Compared to `x` and `y`, how would you characterise the relationship of `x` and `z`, or `y` and `z`?
4. Install the **ggplot2movies** package and look at the movies that have a missing budget. How are they different from the movies with a budget? (Hint: try a frequency polygon plus `colour = is.na(budget)`.)
5. What is `NA & FALSE` and `NA | TRUE`? Why? Why doesn't `NA * 0` equal zero? What number times zero does not equal 0? What do you expect `NA ^ 0` to equal? Why?

10.3 Create new variables

To better explore the relationship between `x` and `y`, it's useful to "rotate" the plot so that the data is flat, not diagonal. We can do that by creating two new variables: one that represents the difference between `x` and `y` (which in this context represents the symmetry of the diamond) and one that represents its size (the length of the diagonal).

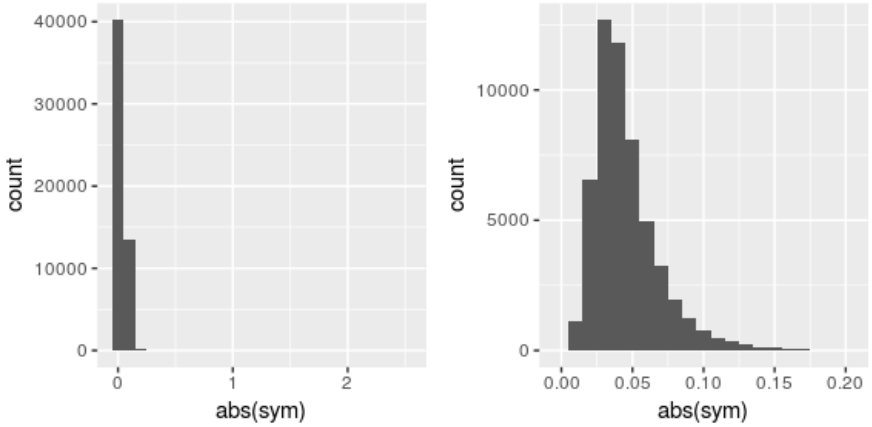
To create new variables use `mutate()`. Like `filter()` it takes a data frame as its first argument and returns a data frame. Its second and subsequent arguments are named expressions that generate new variables. Like `filter()` you can refer to variables just by their name, you don't need to also include the name of the dataset.

```
diamonds_ok2 <- mutate(diamonds_ok,
  sym = x - y,
  size = sqrt(x ^ 2 + y ^ 2)
)
diamonds_ok2
#> # A tibble: 53,930 x 12
#>    carat       cut color clarity depth table price     x     y
#>    <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl>
#>  1  0.23     Ideal     E     SI2  61.5    55   326  3.95  3.98
#>  2  0.21   Premium     E     SI1  59.8    61   326  3.89  3.84
#>  3  0.23      Good     E     VS1  56.9    65   327  4.05  4.07
#>  4  0.29   Premium     I     VS2  62.4    58   334  4.20  4.23
#>  5  0.31      Good     J     SI2  63.3    58   335  4.34  4.35
#>  6  0.24 Very Good     J    VVS2  62.8    57   336  3.94  3.96
#>  7  0.24 Very Good     I    VVS1  62.3    57   336  3.95  3.98
#>  8  0.26 Very Good     H     SI1  61.9    55   337  4.07  4.11
#>  9  0.22      Fair     E     VS2  65.1    61   337  3.87  3.78
#> 10  0.23 Very Good     H     VS1  59.4    61   338  4.00  4.05
#> # ... with 53,920 more rows, and 3 more variables: z <dbl>,
#> #   sym <dbl>, size <dbl>

ggplot(diamonds_ok2, aes(size, sym)) +
  stat_bin2d()
```



This plot has two advantages: we can more easily see the pattern followed by most diamonds, and we can easily select outliers. Here, it doesn't seem important whether the outliers are positive (i.e. `x` is bigger than `y`) or negative (i.e. `y` is bigger `x`). So we can use the absolute value of the symmetry variable to pull out the outliers. Based on the plot, and a little experimentation, I came up with a threshold of 0.20. We'll check out the results with a histogram.

```
ggplot(diamonds_ok2, aes(abs(sym))) +
  geom_histogram(binwidth = 0.10)

diamonds_ok3 <- filter(diamonds_ok2, abs(sym) < 0.20)
ggplot(diamonds_ok3, aes(abs(sym))) +
  geom_histogram(binwidth = 0.01)
```



That's an interesting histogram! While most diamonds are close to being symmetric there are very few that are perfectly symmetric (i.e. `x == y`).

10.3.1 Useful tools

Typically, transformations will be suggested by your domain knowledge. However, there are a few transformations that are useful in a surprisingly wide range of circumstances.

- Log-transformations are often useful. They turn multiplicative relationships into additive relationships; they compress data that varies over orders of magnitude; they convert power relationships to linear relationship. See examples at `http://stats.stackexchange.com/questions/27951`

- Relative difference: If you're interested in the relative difference between two variables, use `log(x / y)`. It's better than `x / y` because it's symmetric: if x < y, `x / y` takes values [0, 1), but if x > y, `x / y` takes values (1, Inf). See Trnqvist, Vartia, and Vartia (1985) for more details.
- Sometimes integrating or differentiating might make the data more interpretable: if you have distance and time, would speed or acceleration be more useful? (or vice versa). (Note that integration makes data more smooth; differentiation makes it less smooth.)
- Partition a number into magnitude and direction with `abs(x)` and `sign(x)`.

There are also a few useful ways to transform pairs of variables:

- Partitioning into overall size and difference is often useful, as seen above.
- If you see a strong trend, use a model to partition it into pattern and residuals is often useful. You'll learn more about that in the next chapter.
- Sometimes it's useful to change positions to polar coordinates (or vice versa): distance (`sqrt(x^2 + y^2)`) and angle (`atan2(y, x)`).

10.3.2 Exercises

1. Practice your variable creation skills by creating the following new variables:
   - The approximate volume of the diamond (using x, y, and z).
   - The approximate density of the diamond.
   - The price per carat.
   - Log transformation of carat and price.
2. How can you improve the data density of `ggplot(diamonds, aes(x, z)) + stat_bin2d()`. What transformation makes it easier to extract outliers?
3. The depth variable is just the width of the diamond (average of `x` and `y`) divided by its height (`z`) multiplied by 100 and round to the nearest integer. Compute the depth yourself and compare it to the existing depth variable. Summarise your findings with a plot.
4. Compare the distribution of symmetry for diamonds with $x > y$ vs. $x < y$.

10.4 Group-wise summaries

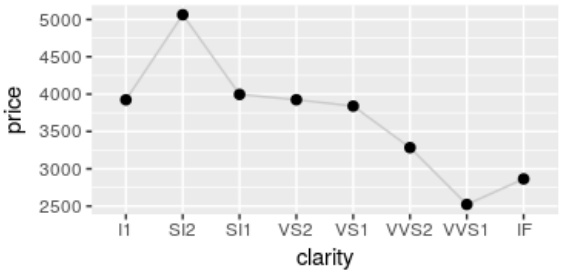
Many insightful visualisations require that you reduce the full dataset down to a meaningful summary. ggplot2 provides a number of geoms that will do summaries for you. But it's often useful to do summaries by hand: that gives you more flexibility and you can use the summaries for other purposes.

dplyr does summaries in two steps:

1. Define the grouping variables with `group_by()`.
2. Describe how to summarise each group with a single row with `summarise()`

For example, to look at the average price per clarity, we first group by clarity, then summarise:

```
by_clarity <- group_by(diamonds, clarity)
sum_clarity <- summarise(by_clarity, price = mean(price))
sum_clarity
#> # A tibble: 8 x 2
#>   clarity price
#>     <ord> <dbl>
#> 1      I1  3924
#> 2     SI2  5063
#> 3     SI1  3996
#> 4     VS2  3925
#> 5     VS1  3839
#> 6    VVS2  3284
#> 7    VVS1  2523
#> 8      IF  2865

ggplot(sum_clarity, aes(clarity, price)) +
  geom_line(aes(group = 1), colour = "grey80") +
  geom_point(size = 2)
```

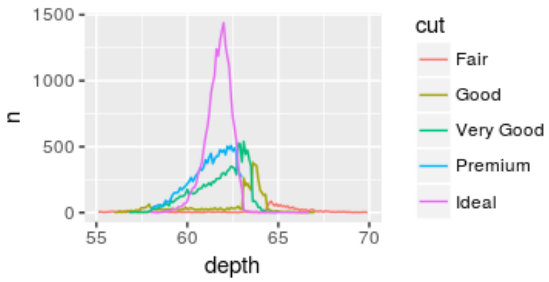


You might be surprised by this pattern: why do diamonds with better clarity have lower prices? We'll see why this is the case and what to do about it in Section 11.2.

Supply additional variables to `group_by()` to create groups based on more than one variable. The next example shows how we can compute (by hand) a frequency polygon that shows how cut and depth interact. The special summary function `n()` counts the number of observations in each group.

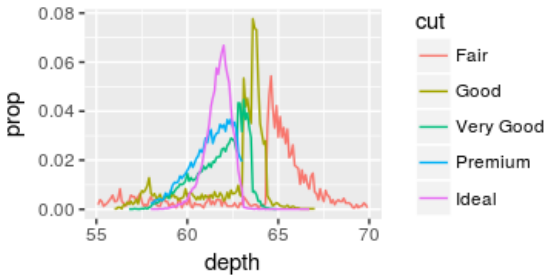
```
cut_depth <- summarise(group_by(diamonds, cut, depth), n = n())
cut_depth <- filter(cut_depth, depth > 55, depth < 70)
```

```
cut_depth
#> # A tibble: 455 x 3
#> # Groups:   cut [5]
#>       cut depth     n
#>    <ord> <dbl> <int>
#>  1  Fair  55.1     3
#>  2  Fair  55.2     6
#>  3  Fair  55.3     5
#>  4  Fair  55.4     2
#>  5  Fair  55.5     3
#>  6  Fair  55.6     4
#>  7  Fair  55.8     7
#>  8  Fair  55.9     9
#>  9  Fair  56.0     5
#> 10  Fair  56.1     5
#> # ... with 445 more rows

ggplot(cut_depth, aes(depth, n, colour = cut)) +
  geom_line()
```



We can use a grouped `mutate()` to convert counts to proportions, so it's easier to compare across the cuts. `summarise()` strips one level of grouping off, so `cut_depth` will be grouped by cut.

```
cut_depth <- mutate(cut_depth, prop = n / sum(n))
ggplot(cut_depth, aes(depth, prop, colour = cut)) +
  geom_line()
```



10.4.1 Useful tools

`summarise()` needs to be used with functions that take a vector of n values and always return a single value. Those functions include:

- Counts: `n()`, `n_distinct(x)`.
- Middle: `mean(x)`, `median(x)`.
- Spread: `sd(x)`, `mad(x)`, `IQR(x)`.
- Extremes: `quartile(x)`, `min(x)`, `max(x)`.
- Positions: `first(x)`, `last(x)`, `nth(x, 2)`.

Another extremely useful technique is to use `sum()` or `mean()` with a logical vector. When a logical vector is treated as numeric, `TRUE` becomes 1 and `FALSE` becomes 0. This means that `sum()` tells you the number of `TRUE`s, and `mean()` tells you the proportion of `TRUE`s. For example, the following code counts the number of diamonds with carat greater than or equal to 4, and the proportion of diamonds that cost less than $1000.

```
summarise(diamonds,
  n_big = sum(carat >= 4),
  prop_cheap = mean(price < 1000)
)
#> # A tibble: 1 x 2
#>   n_big prop_cheap
#>   <int>      <dbl>
#> 1     6      0.269
```

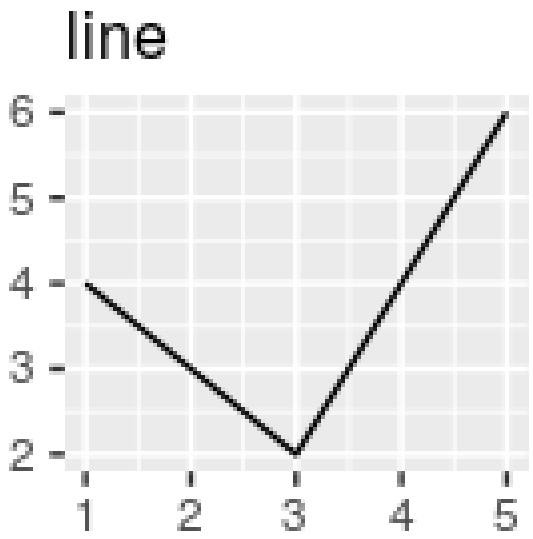
Most summary functions have a `na.rm` argument: `na.rm = TRUE` tells the summary function to remove any missing values prior to summiarisation. This is a convenient shortcut: rather than removing the missing values then summarising, you can do it in one step.

10.4.2 Statistical considerations

When summarising with the mean or median, it's always a good idea to include a count and a measure of spread. This helps you calibrate your assessments - if you don't include them you're likely to think that the data is less variable than it really is, and potentially draw unwarranted conclusions.

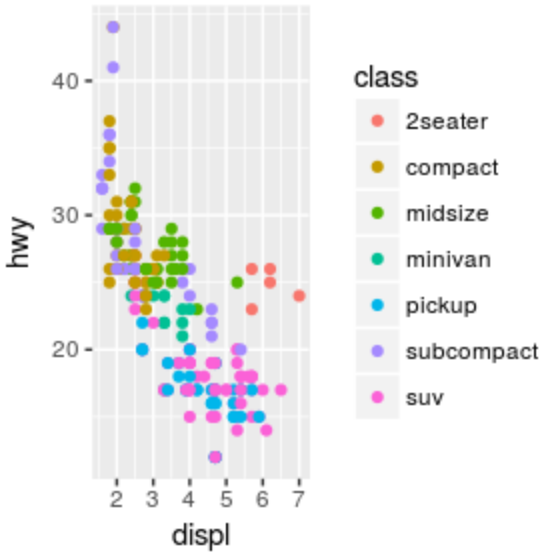
The following example extends our previous summary of the average price by clarity to also include the number of observations in each group, and the upper and lower quartiles. It suggests the mean might be a bad summary for this data - the distributions of price are so highly skewed that the mean is higher than the upper quartile for some of the groups!

```
by_clarity <- diamonds %>%
  group_by(clarity) %>%
  summarise(
    n = n(),
    mean = mean(price),
    lq = quantile(price, 0.25),
    uq = quantile(price, 0.75)
  )
by_clarity
#> # A tibble: 8 x 5
#>   clarity     n  mean    lq    uq
#>     <ord> <int> <dbl> <dbl> <dbl>
#> 1      I1   741  3924  2080  5161
#> 2     SI2  9194  5063  2264  5777
#> 3     SI1 13065  3996  1089  5250
#> 4     VS2 12258  3925   900  6024
#> 5     VS1  8171  3839   876  6023
#> 6    VVS2  5066  3284   794  3638
#> 7    VVS1  3655  2523   816  2379
#> 8      IF  1790  2865   895  2388
ggplot(by_clarity, aes(clarity, mean)) +
  geom_linerange(aes(ymin = lq, ymax = uq)) +
  geom_line(aes(group = 1), colour = "grey50") +
  geom_point(aes(size = n))
```



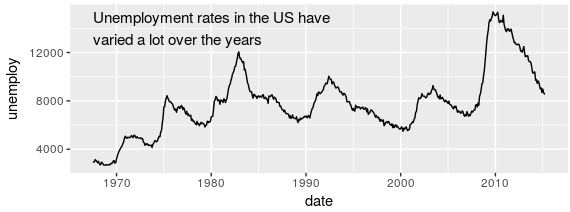
Another example of this comes from baseball. Let's take the MLB batting data from the Lahman package and calculate the batting average: the number of hits divided by the number of at bats. Who's the best batter according to this metric?

```
data(Batting, package = "Lahman")
batters <- filter(Batting, AB > 0)
per_player <- group_by(batters, playerID)
ba <- summarise(per_player,
  ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE)
)
ggplot(ba, aes(ba)) +
  geom_histogram(binwidth = 0.01)
```



Wow, there are a lot of players who can hit the ball every single time! Would you want them on your fantasy baseball team? Let's double check they're really that good by calibrating also showing the total number of at bats:

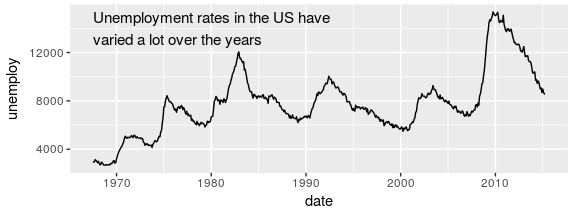
```
ba <- summarise(per_player,
  ba = sum(H, na.rm = TRUE) / sum(AB, na.rm = TRUE),
  ab = sum(AB, na.rm = TRUE)
)
ggplot(ba, aes(ab, ba)) +
  geom_bin2d(bins = 100) +
  geom_smooth()
#> `geom_smooth()` using method = 'gam'
```



The highest batting averages occur for the players with the smallest number of at bats - it's not hard to hit the ball every time if you've only had

two pitches. We can make the pattern a little more clear by getting rid of the players with less than 10 at bats.

```
ggplot(filter(ba, ab >= 10), aes(ab, ba)) +
  geom_bin2d() +
  geom_smooth()
#> `geom_smooth()` using method = 'gam'
```



You'll often see a similar pattern whenever you plot number of observations vs. an average. Be aware!

10.4.3 Exercises

1. For each year in the `ggplot2movies::movies` data determine the percent of movies with missing budgets. Visualise the result.
2. How does the average length of a movie change over time? Display your answer with a plot, including some display of uncertainty.
3. For each combination of diamond quality (e.g. cut, colour and clarity), count the number of diamonds, the average price and the average size. Visualise the results.
4. Compute a histogram of carat by "hand" using a binwidth of 0.1. Display the results with `geom_bar(stat = "identity")`. (Hint: you might need to create a new variable first).
5. In the baseball example, the batting average seems to increase as the number of at bats increases. Why?

10.5 Transformation pipelines

Most real analyses require you to string together multiple `mutate()`s, `filter()`s, `group_by()`s , and `summarise()`s. For example, above, we created a frequency polygon by hand with a combination of all four verbs:

```
# By using intermediate values
cut_depth <- group_by(diamonds, cut, depth)
cut_depth <- summarise(cut_depth, n = n())
cut_depth <- filter(cut_depth, depth > 55, depth < 70)
cut_depth <- mutate(cut_depth, prop = n / sum(n))
```

This sequence of operations is a bit painful because we repeated the name of the data frame many times. An alternative is just to do it with one sequence of function calls:

```
# By "composing" functions
mutate(
  filter(
    summarise(
      group_by(
        diamonds,
        cut,
        depth
      ),
      n = n()
    ),
    depth > 55,
    depth < 70
  ),
  prop = n / sum(n)
)
```

But this is also hard to read because the sequence of operations is inside out, and the arguments to each function can be quite far apart. dplyr provides an alternative approach with the **pipe**, `%>%`. With the pipe, we can write the above sequence of operations as:

```
cut_depth <- diamonds %>%
  group_by(cut, depth) %>%
  summarise(n = n()) %>%
  filter(depth > 55, depth < 70) %>%
  mutate(prop = n / sum(n))
```

This makes it easier to understand what's going on as we can read it almost like an English sentence: first group, then summarise, then filter, then mutate. In fact, the best way to pronounce `%>%` when reading a sequence of code is as "then". `%>%` comes from the magrittr package, by Stefan Milton Bache. It provides a number of other tools that dplyr doesn't expose by default, so I highly recommend that you check out the magrittr website (`https://github.com/smbache/magrittr`).

`%>%` works by taking the thing on the left hand side (LHS) and supplying it as the first argument to the function on the right hand side (RHS). Each of these pairs of calls is equivalent:

```
f(x, y)
# is the same as
x %>% f(y)

g(f(x, y), z)
# is the same as
x %>% f(y) %>% g(z)
```

10.5.1 Exercises

1. Translate each of the examples in this chapter to use the pipe.
2. What does the following pipe do?

   ```
   library(magrittr)
   x <- runif(100)
   x %>%
     subtract(mean(.)) %>%
     raise_to_power(2) %>%
     mean() %>%
     sqrt()
   ```

3. Which player in the `Batting` dataset has had the most consistently good performance over the course of their career?

10.6 Learning more

dplyr provides a number of other verbs that are less useful for visualisation, but important to know about in general:

- `arrange()` orders observations according to variable(s). This is most useful when you're looking at the data from the console. It can also be useful for visualisations if you want to control which points are plotted on top.
- `select()` picks variables based on their names. Useful when you have many variables and want to focus on just a few for analysis.
- `rename()` allows you to change the name of variables.
- Grouped mutates and filters are also useful, but more advanced. See `vignette("window-functions", package = "dplyr")` for more details.
- There are a number of verbs designed to work with two tables of data at a time. These include SQL joins (like the base `merge()` function) and set operations. Learn more about them in `vignette("two-table", package = "dplyr")`.

- dplyr can work directly with data stored in a database - you use the same R code as you do for local data and dplyr generates SQL to send to the database. See `vignette("databases", package = "dplyr")` for the details.

Finally, RStudio provides a handy dplyr cheatsheet that will help jog your memory when you're wondering which function to use. Get it from `http://rstudio.com/cheatsheets`.

References

Trnqvist, Leo, Pentti Vartia, and Yrj O Vartia. 1985. "How Should Relative Changes Be Measured?" *The American Statistician* 39 (1): 43–46.

Chapter 11
Modelling for visualisation

11.1 Introduction

Modelling is an essential tool for visualisation. There are two particularly strong connections between modelling and visualisation that I want to explore in this chapter:

- Using models as a tool to remove obvious patterns in your plots. This is useful because strong patterns mask subtler effects. Often the strongest effects are already known and expected, and removing them allows you to see surprises more easily.
- Other times you have a lot of data, too much to show on a handful of plots. Models can be a powerful tool for summarising data so that you get a higher level view.

In this chapter, I'm going to focus on the use of linear models to acheive these goals. Linear models are a basic, but powerful, tool of statistics, and I recommend that everyone serious about visualisation learns at least the basics of how to use them. To this end, I highly recommend two books by Julian J. Faraway:

- Linear Models with R `http://amzn.com/1439887330`
- Extending the Linear Model with R `http://amzn.com/158488424X`

These books cover some of the theory of linear models, but are pragmatic and focussed on how to actually use linear models (and their extensions) in R.

There are many other modelling tools, which I don't have the space to show. If you understand how linear models can help improve your visualisations, you should be able to translate the basic idea to other families of models. This chapter just scratches the surface of what you can do. But hopefully it reinforces how visualisation can combine with modelling to help you

build a powerful data analysis toolbox. For more ideas, check out Wickham, Cook, and Hofmann (2015).

This chapter only scratches the surface of the intersection between visualisation and modelling. In my opinion, mastering the combination of visualisations and models is key to being an effective data scientist. Unfortunately most books (like this one!) only focus on either visualisation or modelling, but not both. There's a lot of interesting work to be done.

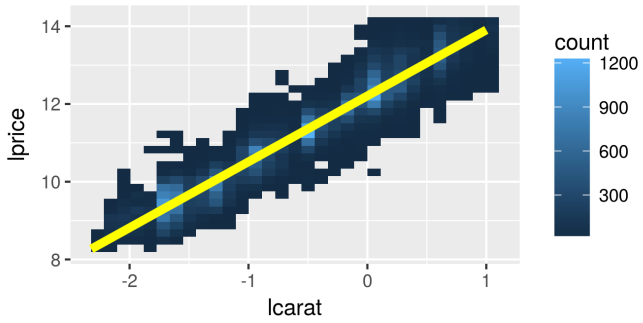
11.2 Removing trend

So far our analysis of the diamonds data has been plagued by the powerful relationship between size and price. It makes it very difficult to see the impact of cut, colour and clarity because higher quality diamonds tend to be smaller, and hence cheaper. This challenge is often called confounding. We can use a linear model to remove the effect of size on price. Instead of looking at the raw price, we can look at the relative price: how valuable is this diamond relative to the average diamond of the same size.

To get started, we'll focus on diamonds of size two carats or less (96% of the dataset). This avoids some incidental problems that you can explore in the exercises if you're interested. We'll also create two new variables: log price and log carat. These variables are useful because they produce a plot with a strong linear trend.

```
diamonds2 <- diamonds %>%
  filter(carat <= 2) %>%
  mutate(
    lcarat = log2(carat),
    lprice = log2(price)
  )
diamonds2
#> # A tibble: 52,051 x 12
#>   carat       cut color clarity depth table price     x     y
#>   <dbl>     <ord> <ord>   <ord> <dbl> <dbl> <int> <dbl> <dbl>
#> 1  0.23     Ideal     E     SI2  61.5    55   326  3.95  3.98
#> 2  0.21   Premium     E     SI1  59.8    61   326  3.89  3.84
#> 3  0.23      Good     E     VS1  56.9    65   327  4.05  4.07
#> 4  0.29   Premium     I     VS2  62.4    58   334  4.20  4.23
#> 5  0.31      Good     J     SI2  63.3    58   335  4.34  4.35
#> 6  0.24 Very Good     J    VVS2  62.8    57   336  3.94  3.96
#> # ... with 5.204e+04 more rows, and 3 more variables: z <dbl>,
#> #   lcarat <dbl>, lprice <dbl>

ggplot(diamonds2, aes(lcarat, lprice)) +
```

```
  geom_bin2d() +
  geom_smooth(method = "lm", se = FALSE, size = 2, colour = "yellow")
```

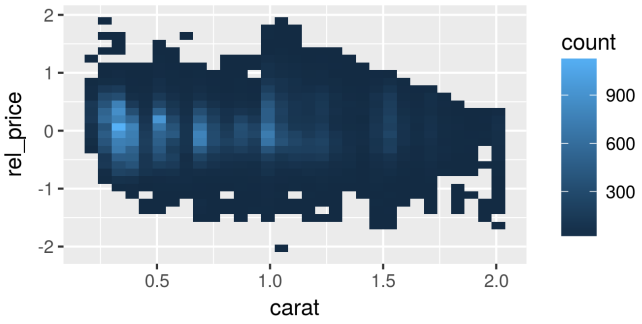


In the graphic we used `geom_smooth()` to overlay the line of best fit to the data. We can replicate this outside of ggplot2 by fitting a linear model with `lm()`. This allows us to find out the slope and intercept of the line:

```
mod <- lm(lprice ~ lcarat, data = diamonds2)
coef(summary(mod))
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)     12.2    0.00211    5789        0
#> lcarat           1.7    0.00208     816        0
```

If you're familiar with linear models, you might want to interpret those coefficients: $\log_2(price) = 12.2 + 1.7 \cdot \log_2(carat)$, which implies $price = 4900 \cdot carat^{1.7}$. Interpreting those coefficients certainly is useful, but even if you don't understand them, the model can still be useful. We can use it to subtract the trend away by looking at the residuals: the price of each diamond minus its predicted price, based on weight alone. Geometrically, the residuals are the vertical distance between each point and the line of best fit. They tell us the price relative to the "average" diamond of that size.

```
diamonds2 <- diamonds2 %>% mutate(rel_price = resid(mod))
ggplot(diamonds2, aes(carat, rel_price)) +
  geom_bin2d()
```



A relative price of zero means that the diamond was at the average price; positive means that it's more expensive than expected (based on its size), and negative means that it's cheaper than expected.

Interpreting the values precisely is a little tricky here because we've log-transformed price. The residuals give the absolute difference ($x - expected$), but here we have $\log_2(price) - \log_2(expectedprice)$, or equivalently $\log_2(price/expectedprice)$. If we "back-transform" to the original scale by applying the opposite transformation (2^x) we get $price/expectedprice$. This makes the values more interpretable, at the cost of the nice symmetry property of the logged values (i.e. both relatively cheaper and relatively more expensive diamonds have the same range). We can make a little table to help interpret the values:

```
xgrid <- seq(-2, 1, by = 1/3)
data.frame(logx = xgrid, x = round(2 ^ xgrid, 2))
#>      logx    x
#> 1  -2.000 0.25
#> 2  -1.667 0.31
#> 3  -1.333 0.40
#> 4  -1.000 0.50
#> 5  -0.667 0.63
#> 6  -0.333 0.79
#> 7   0.000 1.00
#> 8   0.333 1.26
#> 9   0.667 1.59
#> 10  1.000 2.00
```

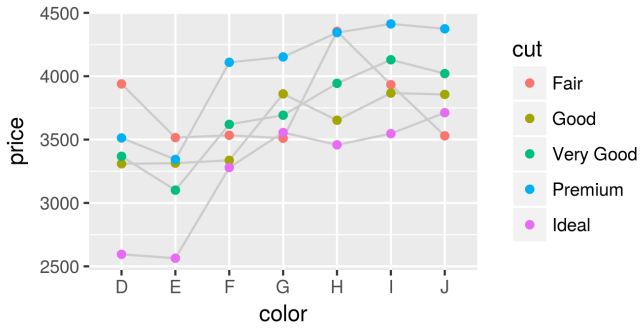
This table illustrates why we used `log2()` rather than `log()`: a change of 1 unit on the logged scale, corresponding to a doubling on the original scale. For example, a `rel_price` of -1 means that it's half of the expected price; a relative price of 1 means that it's twice the expected price.

Let's use both price and relative price to see how colour and cut affect the value of a diamond. We'll compute the average price and average relative price for each combination of colour and cut:

```
color_cut <- diamonds2 %>%
  group_by(color, cut) %>%
  summarise(
    price = mean(price),
    rel_price = mean(rel_price)
  )
color_cut
#> # A tibble: 35 x 4
#> # Groups:   color [?]
#>   color       cut price rel_price
#>   <ord>     <ord> <dbl>     <dbl>
#> 1     D      Fair  3939   -0.0755
#> 2     D      Good  3309   -0.0472
#> 3     D Very Good  3368    0.1038
#> 4     D   Premium  3513    0.1093
#> 5     D     Ideal  2595    0.2173
#> 6     E      Fair  3516   -0.1720
#> # ... with 29 more rows
```

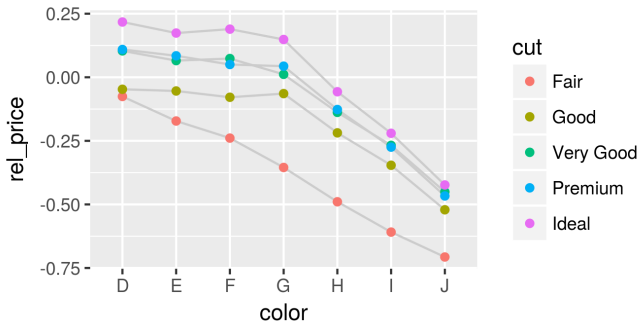
If we look at price, it's hard to see how the quality of the diamond affects the price. The lowest quality diamonds (fair cut with colour J) have the highest average value! This is because those diamonds also tend to be larger: size and quality are confounded.

```
ggplot(color_cut, aes(color, price)) +
  geom_line(aes(group = cut), colour = "grey80") +
  geom_point(aes(colour = cut))
```



If however, we plot the relative price, you see the pattern that you expect: as the quality of the diamonds decreases, the relative price decreases. The worst quality diamond is 0.61x ($2^{-0.7}$) the price of an "average" diamond.

```
ggplot(color_cut, aes(color, rel_price)) +
  geom_line(aes(group = cut), colour = "grey80") +
  geom_point(aes(colour = cut))
```



This technique can be employed in a wide range of situations. Wherever you can explicitly model a strong pattern that you see in a plot, it's worthwhile to use a model to remove that strong pattern so that you can see what interesting trends remain.

11.2.1 Exercises

1. What happens if you repeat the above analysis with all diamonds? (Not just all diamonds with two or fewer carats). What does the strange geometry of `log(carat)` vs relative price represent? What does the diagonal line without any points represent?
2. I made an unsupported assertion that lower-quality diamonds tend to be larger. Support my claim with a plot.
3. Can you create a plot that simultaneously shows the effect of colour, cut, and clarity on relative price? If there's too much information to show on one plot, think about how you might create a sequence of plots to convey the same message.
4. How do depth and table relate to the relative price?

11.3 Texas housing data

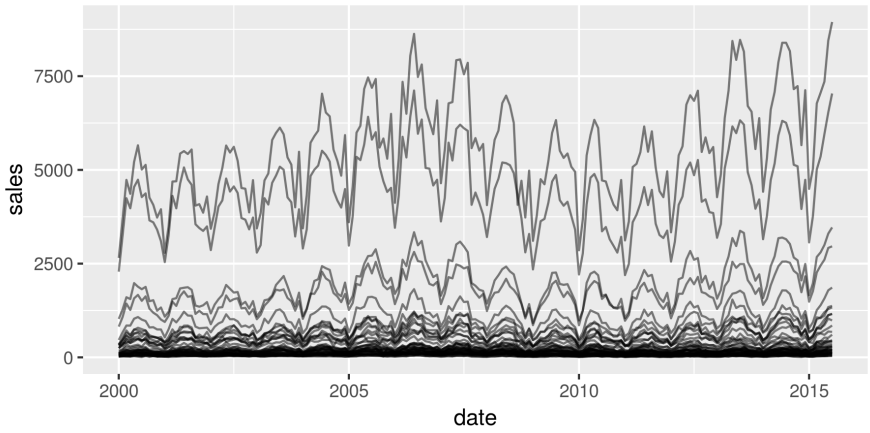
We'll continue to explore the connection between modelling and visualisation with the `txhousing` dataset:

```
txhousing
#> # A tibble: 8,034 x 9
#>      city  year month sales   volume median listings inventory
#>     <chr> <int> <int> <dbl>    <dbl>  <dbl>    <dbl>     <dbl>
#> 1 Abilene  2000     1    72  5380000  71400      701       6.3
#> 2 Abilene  2000     2    98  6505000  58700      746       6.6
#> 3 Abilene  2000     3   130  9285000  58100      784       6.8
#> 4 Abilene  2000     4    98  9730000  68600      785       6.9
#> 5 Abilene  2000     5   141 10590000  67300      794       6.8
#> 6 Abilene  2000     6   156 13910000  66900      780       6.6
#> # ... with 8,028 more rows, and 1 more variables: date <dbl>
```

This data was collected by the Real Estate Center at Texas A&M University, `http://recenter.tamu.edu/Data/hs/`. The data contains information about 46 Texas cities, recording the number of house sales (`sales`), the total volume of sales (`volume`), the `average` and `median` sale prices, the number of houses listed for sale (`listings`) and the number of months inventory (`inventory`). Data is recorded monthly from Jan 2000 to Apr 2015, 187 entries for each city.

We're going to explore how sales have varied over time for each city as it shows some interesting trends and poses some interesting challenges. Let's start with an overview: a time series of sales for each city:

```
ggplot(txhousing, aes(date, sales)) +
  geom_line(aes(group = city), alpha = 1/2)
```

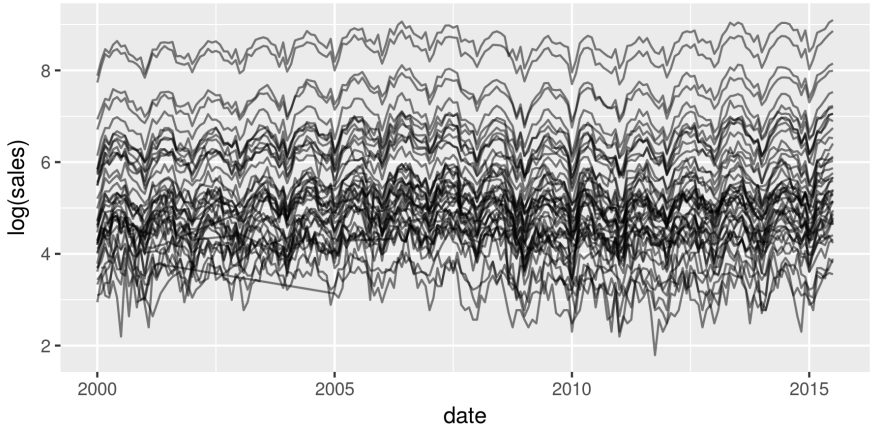


Two factors make it hard to see the long-term trend in this plot:

1. The range of sales varies over multiple orders of magnitude. The biggest city, Houston, averages over ~4000 sales per month; the smallest city, San Marcos, only averages ~20 sales per month.
2. There is a strong seasonal trend: sales are much higher in the summer than in the winter.

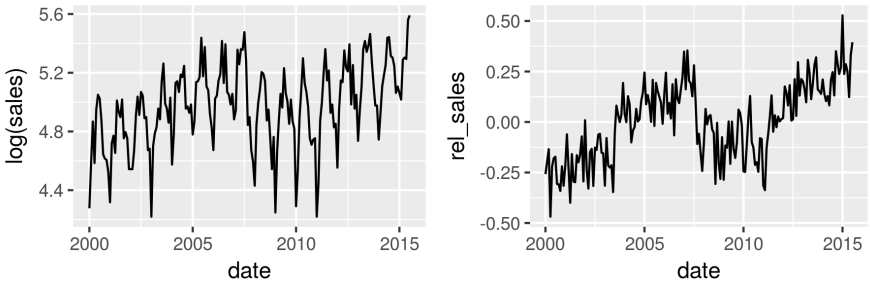
We can fix the first problem by plotting log sales:

```
ggplot(txhousing, aes(date, log(sales))) +
  geom_line(aes(group = city), alpha = 1/2)
```



We can fix the second problem using the same technique we used for removing the trend in the diamonds data: we'll fit a linear model and look at the residuals. This time we'll use a categorical predictor to remove the month effect. First we check that the technique works by applying it to a single city. It's always a good idea to start simple so that if something goes wrong you can more easily pinpoint the problem.

```
abilene <- txhousing %>% filter(city == "Abilene")
ggplot(abilene, aes(date, log(sales))) +
  geom_line()

mod <- lm(log(sales) ~ factor(month), data = abilene)
abilene$rel_sales <- resid(mod)
ggplot(abilene, aes(date, rel_sales)) +
  geom_line()
```

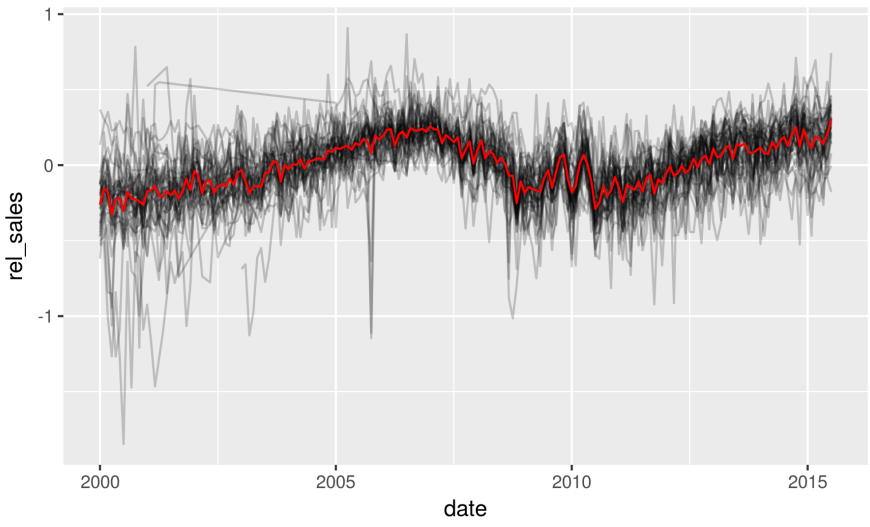


We can apply this transformation to every city with `group_by()` and `mutate()`. Note the use of `na.action = na.exclude` argument to `lm()`. Counterintuitively this ensures that missing values in the input are matched with missing values in the output predictions and residuals. Without this argument, missing values are just dropped, and the residuals don't line up with the inputs.

```
deseas <- function(x, month) {
  resid(lm(x ~ factor(month), na.action = na.exclude))
}

txhousing <- txhousing %>%
  group_by(city) %>%
  mutate(rel_sales = deseas(log(sales), month))
```

With this data in hand, we can re-plot the data. Now that we have log-transformed the data and removed the strong seasonal effects we can see there is a strong common pattern: a consistent increase from 2000-2007, a drop until 2010 (with quite some noise), and then a gradual rebound. To make that more clear, I included a summary line that shows the mean relative sales across all cities.

```
ggplot(txhousing, aes(date, rel_sales)) +
  geom_line(aes(group = city), alpha = 1/5) +
  geom_line(stat = "summary", fun.y = "mean", colour = "red")
```



(Note that removing the seasonal effect also removed the intercept - we see the trend for each city relative to its average number of sales.)

11.3.1 Exercises

1. The final plot shows a lot of short-term noise in the overall trend. How could you smooth this further to focus on long-term changes?
2. If you look closely (e.g. `+ xlim(2008, 2012)`) at the long-term trend you'll notice a weird pattern in 2009-2011. It looks like there was a big dip in 2010. Is this dip "real"? (i.e. can you spot it in the original data)
3. What other variables in the TX housing data show strong seasonal effects? Does this technique help to remove them?
4. Not all the cities in this data set have complete time series. Use your dplyr skills to figure out how much data each city is missing. Display the results with a visualisation.
5. Replicate the computation that `stat_summary()` did with dplyr so you can plot the data "by hand".

11.4 Visualising models

The previous examples used the linear model just as a tool for removing trend: we fit the model and immediately threw it away. We didn't care about the model itself, just what it could do for us. But the models themselves

contain useful information and if we keep them around, there are many new problems that we can solve:

- We might be interested in cities where the model didn't fit well: a poorly fitting model suggests that there isn't much of a seasonal pattern, which contradicts our implicit hypothesis that all cities share a similar pattern.
- The coefficients themselves might be interesting. In this case, looking at the coefficients will show us how the seasonal pattern varies between cities.
- We may want to dive into the details of the model itself, and see exactly what it says about each observation. For this data, it might help us find suspicious data points that might reflect data entry errors.

To take advantage of this data, we need to store the models. We can do this using a new dplyr verb: `do()`. It allows us to store the result of arbitrary computation in a column. Here we'll use it to store that linear model:

```
models <- txhousing %>%
  group_by(city) %>%
  do(mod = lm(
    log2(sales) ~ factor(month),
    data = .,
    na.action = na.exclude
  ))
models
#> Source: local data frame [46 x 2]
#> Groups: <by row>
#>
#> # A tibble: 46 x 2
#>        city      mod
#> *     <chr>   <list>
#> 1   Abilene <S3: lm>
#> 2  Amarillo <S3: lm>
#> 3 Arlington <S3: lm>
#> 4    Austin <S3: lm>
#> 5  Bay Area <S3: lm>
#> 6  Beaumont <S3: lm>
#> # ... with 40 more rows
```

There are two important things to note in this code:

- `do()` creates a new column called `mod`. This is a special type of column: instead of containing an atomic vector (a logical, integer, numeric, or character) like usual, it's a list. Lists are R's most flexible data structure and can hold anything, including linear models.
- `.` is a special pronoun used by `do()`. It refers to the "current" data frame. In this example, `do()` fits the model 46 times (once for each city), each time replacing `.` with the data for one city.

If you're an experienced modeller, you might wonder why I didn't fit one model to all cities simultaneously. That's a great next step, but it's often useful to start off simple. Once we have a model that works for each city individually, you can figure out how to generalise it to fit all cities simultaneously.

To visualise these models, we'll turn them into tidy data frames. We'll do that with the **broom** package by David Robinson.

```
library(broom)
```

Broom provides three key verbs, each corresponding to one of the challenges outlined above:

- `glance()` extracts **model**-level summaries with one row of data for each model. It contains summary statistics like the R^2 and degrees of freedom.
- `tidy()` extracts **coefficient**-level summaries with one row of data for each coefficient in each model. It contains information about individual coefficients like their estimate and standard error.
- `augment()` extracts **observation**-level summaries with one row of data for each observation in each model. It includes variables like the residual and influence metrics useful for diagnosing outliers.

We'll learn more about each of these functions in the following three sections.

11.5 Model-level summaries

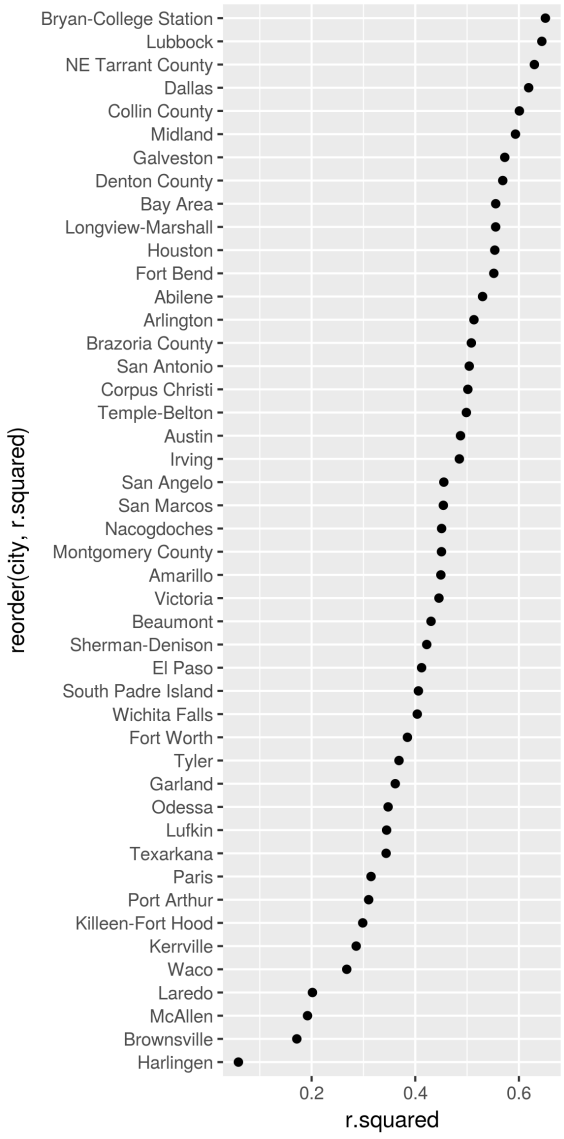
We'll begin by looking at how well the model fit to each city with `glance()`:

```
model_sum <- models %>% glance(mod)
model_sum
#> # A tibble: 46 x 12
#> # Groups:   city [46]
#>        city r.squared adj.r.squared sigma statistic  p.value    df
#>       <chr>     <dbl>         <dbl> <dbl>     <dbl>    <dbl> <int>
#> 1   Abilene     0.530         0.500 0.282      17.9 1.50e-23    12
#> 2  Amarillo     0.449         0.415 0.302      13.0 7.41e-18    12
#> 3 Arlington     0.513         0.483 0.267      16.8 2.75e-22    12
#> 4    Austin     0.487         0.455 0.310      15.1 2.04e-20    12
#> 5  Bay Area     0.555         0.527 0.265      19.9 1.45e-25    12
#> 6  Beaumont     0.430         0.395 0.275      12.0 1.18e-16    12
#> # ... with 40 more rows, and 5 more variables: logLik <dbl>,
#> #   AIC <dbl>, BIC <dbl>, deviance <dbl>, df.residual <int>
```

This creates a variable with one row for each city, and variables that either summarise complexity (e.g. `df`) or fit (e.g. `r.squared`, `p.value`, `AIC`). Since all

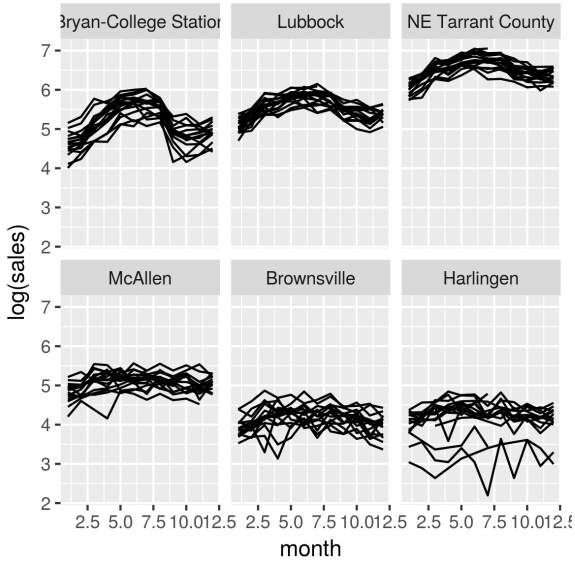
the models we fit have the same complexity (12 terms: one for each month), we'll focus on the model fit summaries. R^2 is a reasonable place to start because it's well known. We can use a dot plot to see the variation across cities:

```
ggplot(model_sum, aes(r.squared, reorder(city, r.squared))) +
  geom_point()
```



It's hard to picture exactly what those values of R^2 mean, so it's helpful to pick out a few exemplars. The following code extracts and plots out the three cities with the highest and lowest R^2:

```
top3 <- c("Bryan-College Station", "Lubbock", "NE Tarrant County")
bottom3 <- c("McAllen", "Brownsville", "Harlingen")
extreme <- txhousing %>% ungroup() %>%
  filter(city %in% c(top3, bottom3), !is.na(sales)) %>%
  mutate(city = factor(city, c(top3, bottom3)))

ggplot(extreme, aes(month, log(sales))) +
  geom_line(aes(group = year)) +
  facet_wrap(~city)
```



The cities with low R^2 have weaker seasonal patterns and more variation between years. The data for Harlingen seems particularly noisy.

11.5.1 Exercises

1. Do your conclusions change if you use a different measurement of model fit like AIC or deviance? Why/why not?

2. One possible hypothesis that explains why McAllen, Harlingen and Brownsville have lower R^2 is that they're smaller towns so there are fewer sales and more noise. Confirm or refute this hypothesis.
3. McAllen, Harlingen and Brownsville seem to have much more year-to-year variation than Bryan-College Station, Lubbock, and NE Tarrant County. How does the model change if you also include a linear trend for year? (i.e. `log(sales) ~ factor(month) + year`).
4. Create a faceted plot that shows the seasonal patterns for all cities. Order the facets by the R^2 for the city.

11.6 Coefficient-level summaries

The model fit summaries suggest that there are some important differences in seasonality between the different cities. Let's dive into those differences by using `tidy()` to extract detail about each individual coefficient:

```
coefs <- models %>% tidy(mod)
coefs
#> # A tibble: 552 x 6
#> # Groups:   city [46]
#>      city           term estimate std.error statistic   p.value
#>     <chr>          <chr>    <dbl>     <dbl>     <dbl>     <dbl>
#> 1 Abilene    (Intercept)    6.542    0.0704     92.88 7.90e-151
#> 2 Abilene factor(month)2    0.354    0.0996      3.55  4.91e-04
#> 3 Abilene factor(month)3    0.675    0.0996      6.77  1.83e-10
#> 4 Abilene factor(month)4    0.749    0.0996      7.52  2.76e-12
#> 5 Abilene factor(month)5    0.916    0.0996      9.20  1.06e-16
#> 6 Abilene factor(month)6    1.002    0.0996     10.06  4.37e-19
#> # ... with 546 more rows
```

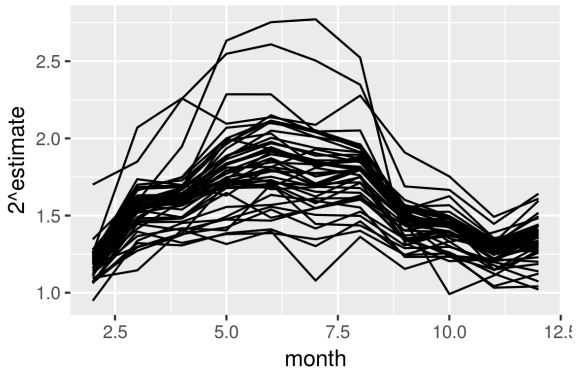
We're more interested in the month effect, so we'll do a little extra tidying to only look at the month coefficients, and then to extract the month value into a numeric variable:

```
months <- coefs %>%
  filter(grepl("factor", term)) %>%
  tidyr::extract(term, "month", "(\\d+)", convert = TRUE)
months
#> # A tibble: 506 x 6
#> # Groups:   city [46]
#>      city month estimate std.error statistic  p.value
#> *   <chr> <int>    <dbl>     <dbl>     <dbl>    <dbl>
#> 1 Abilene     2    0.354    0.0996      3.55 4.91e-04
#> 2 Abilene     3    0.675    0.0996      6.77 1.83e-10
```

```
#> 3 Abilene     4    0.749    0.0996      7.52 2.76e-12
#> 4 Abilene     5    0.916    0.0996      9.20 1.06e-16
#> 5 Abilene     6    1.002    0.0996     10.06 4.37e-19
#> 6 Abilene     7    0.954    0.0996      9.58 9.81e-18
#> # ... with 500 more rows
```

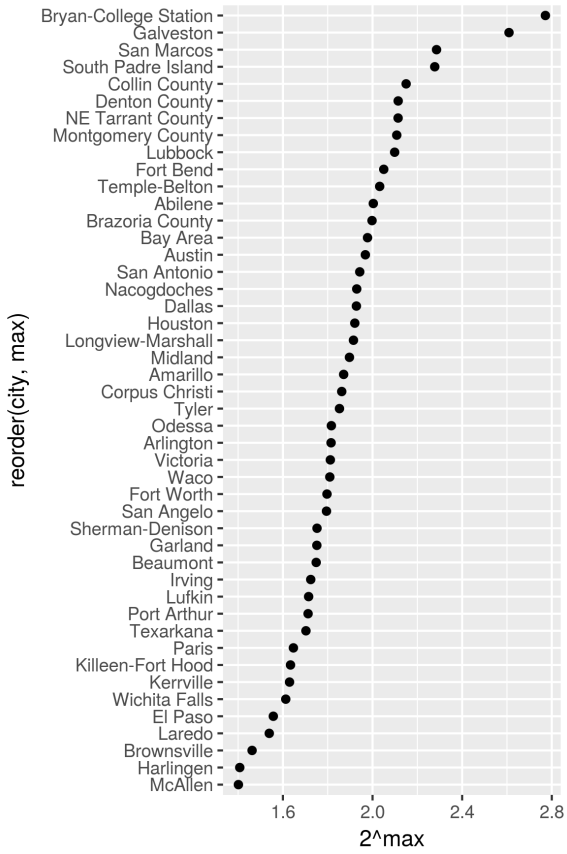
This is a common pattern. You need to use your data tidying skills at many points in an analysis. Once you have the correct tidy dataset, creating the plot is usually easy. Here we'll put month on the x-axis, estimate on the y-axis, and draw one line for each city. I've back-transformed to make the coefficients more interpretable: these are now ratios of sales compared to January.

```
ggplot(months, aes(month, 2 ^ estimate)) +
  geom_line(aes(group = city))
```



The pattern seems similar across the cities. The main difference is the strength of the seasonal effect. Let's pull that out and plot it:

```
coef_sum <- months %>%
  group_by(city) %>%
  summarise(max = max(estimate))
ggplot(coef_sum, aes(2 ^ max, reorder(city, max))) +
  geom_point()
```



The cities with the strongest seasonal effect are College Station and San Marcos (both college towns) and Galveston and South Padre Island (beach cities). It makes sense that these cities would have very strong seasonal effects.

11.6.1 Exercises

1. Pull out the three cities with highest and lowest seasonal effect. Plot their coefficients.
2. How does strength of seasonal effect relate to the R^2 for the model? Answer with a plot.
3. You should be extra cautious when your results agree with your prior beliefs. How can you confirm or refute my hypothesis about the causes of strong seasonal patterns?

4. Group the diamonds data by cut, clarity and colour. Fit a linear model `log(price) ~ log(carat)`. What does the intercept tell you? What does the slope tell you? How do the slope and intercept vary across the groups? Answer with a plot.

11.7 Observation data

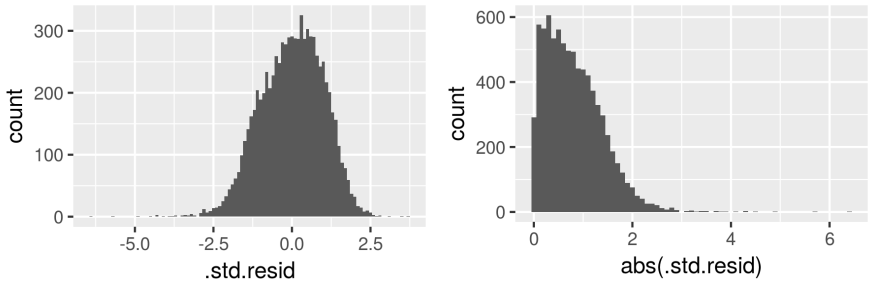
Observation-level data, which include residual diagnostics, is most useful in the traditional model fitting scenario, because it can helps you find "high-leverage" points, point that have a big influence on the final model. It's also useful in conjunction with visualisation, particularly because it provides an alternative way to access the residuals.

Extracting observation-level data is the job of the `augment()` function. This adds one row for each observation. It includes the variables used in the original model, the residuals, and a number of common influence statistics (see `?augment.lm` for more details):

```
obs_sum <- models %>% augment(mod)
obs_sum
#> # A tibble: 8,034 x 10
#> # Groups:   city [46]
#>      city log2.sales. factor.month. .fitted .se.fit .resid   .hat
#>     <chr>       <dbl>        <fctr>   <dbl>   <dbl>  <dbl>  <dbl>
#> 1 Abilene        6.17             1    6.54  0.0704 -0.372 0.0625
#> 2 Abilene        6.61             2    6.90  0.0704 -0.281 0.0625
#> 3 Abilene        7.02             3    7.22  0.0704 -0.194 0.0625
#> 4 Abilene        6.61             4    7.29  0.0704 -0.676 0.0625
#> 5 Abilene        7.14             5    7.46  0.0704 -0.319 0.0625
#> 6 Abilene        7.29             6    7.54  0.0704 -0.259 0.0625
#> # ... with 8,028 more rows, and 3 more variables: .sigma <dbl>,
#> #   .cooksd <dbl>, .std.resid <dbl>
```

For example, it might be interesting to look at the distribution of standardised residuals. (These are residuals standardised to have a variance of one in each model, making them more comparable). We're looking for unusual values that might need deeper exploration:

```
ggplot(obs_sum, aes(.std.resid)) +
  geom_histogram(binwidth = 0.1)
ggplot(obs_sum, aes(abs(.std.resid))) +
  geom_histogram(binwidth = 0.1)
```



A threshold of 2 seems like a reasonable threshold to explore individually:

```
obs_sum %>%
  filter(abs(.std.resid) > 2) %>%
  group_by(city) %>%
  summarise(n = n(), avg = mean(abs(.std.resid))) %>%
  arrange(desc(n))
#> # A tibble: 43 x 3
#>              city     n   avg
#>             <chr> <int> <dbl>
#> 1       Texarkana    12  2.43
#> 2       Harlingen    11  2.73
#> 3            Waco    11  2.96
#> 4        Victoria    10  2.49
#> 5 Brazoria County     9  2.31
#> 6     Brownsville     9  2.48
#> # ... with 37 more rows
```

In a real analysis, you'd want to look into these cities in more detail.

11.7.1 Exercises

1. A common diagnotic plot is fitted values (`.fitted`) vs. residuals (`.resid`). Do you see any patterns? What if you include the city or month on the same plot?
2. Create a time series of log(sales) for each city. Highlight points that have a standardised residual of greater than 2.

References

Wickham, Hadley, Dianne Cook, and Heike Hofmann. 2015. "Visualizing Statistical Models: Removing the Blindfold." *Statistical Analysis and Data Mining: The ASA Data Science Journal* 8 (4): 203–25.

Chapter 12
Programming with ggplot2

12.1 Introduction

A major requirement of a good data analysis is flexibility. If your data changes, or you discover something that makes you rethink your basic assumptions, you need to be able to easily change many plots at once. The main inhibitor of flexibility is code duplication. If you have the same plotting statement repeated over and over again, you'll have to make the same change in many different places. Often just the thought of making all those changes is exhausting! This chapter will help you overcome that problem by showing you how to program with ggplot2.

To make your code more flexible, you need to reduce duplicated code by writing functions. When you notice you're doing the same thing over and over again, think about how you might generalise it and turn it into a function. If you're not that familiar with how functions work in R, you might want to brush up your knowledge at `http://adv-r.had.co.nz/Functions.html`.

In this chapter I'll show how to write functions that create:

- A single ggplot2 component.
- Multiple ggplot2 components.
- A complete plot.

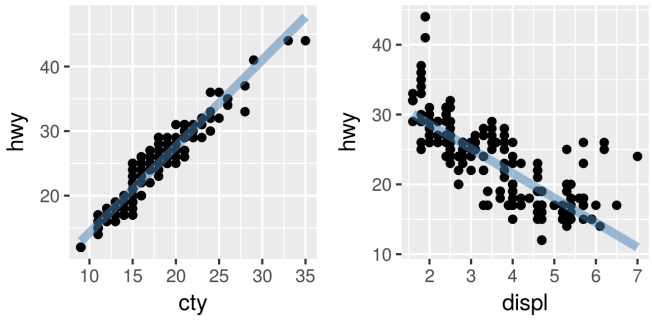
And then I'll finish off with a brief illustration of how you can apply functional programming techniques to ggplot2 objects.

You might also find the cowplot (`https://github.com/wilkelab/cowplot`) and ggthemes (`https://github.com/jrnold/ggthemes`) packages helpful. As well as providing reusable components that help you directly, you can also read the source code of the packages to figure out how they work.

12.2 Single components

Each component of a ggplot plot is an object. Most of the time you create the component and immediately add it to a plot, but you don't have to. Instead, you can save any component to a variable (giving it a name), and then add it to multiple plots:

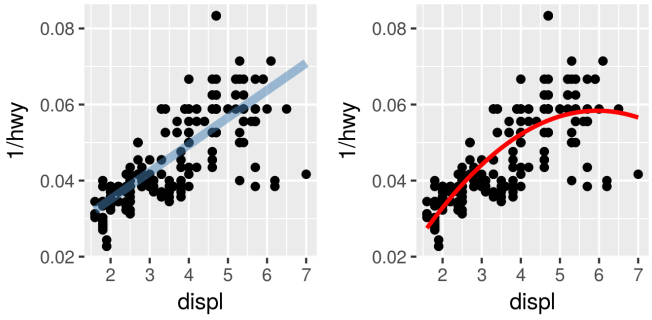
```
bestfit <- geom_smooth(
  method = "lm",
  se = FALSE,
  colour = alpha("steelblue", 0.5),
  size = 2
)
ggplot(mpg, aes(cty, hwy)) +
  geom_point() +
  bestfit
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  bestfit
```



That's a great way to reduce simple types of duplication (it's much better than copying-and-pasting!), but requires that the component be exactly the same each time. If you need more flexibility, you can wrap these reusable snippets in a function. For example, we could extend our `bestfit` object to a more general function for adding lines of best fit to a plot. The following code creates a `geom_lm()` with three parameters: the model `formula`, the line `colour` and the line `size`:

```
geom_lm <- function(formula = y ~ x, colour = alpha("steelblue", 0.5),
                    size = 2, ...)  {
  geom_smooth(formula = formula, se = FALSE, method = "lm", colour = colour,
    size = size, ...)
```

```
}
ggplot(mpg, aes(displ, 1 / hwy)) +
  geom_point() +
  geom_lm()
ggplot(mpg, aes(displ, 1 / hwy)) +
  geom_point() +
  geom_lm(y ~ poly(x, 2), size = 1, colour = "red")
```



Pay close attention to the use of "...". When included in the function definition "..." allows a function to accept arbitrary additional arguments. Inside the function, you can then use "..." to pass those arguments on to another function. Here we pass "..." onto `geom_smooth()` so the user can still modify all the other arguments we haven't explicitly overridden. When you write your own component functions, it's a good idea to always use "..." in this way.

Finally, note that you can only *add* components to a plot; you can't modify or remove existing objects.

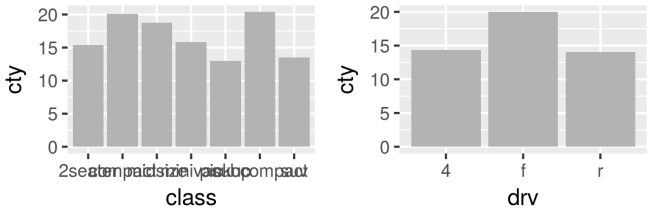
12.2.1 Exercises

1. Create an object that represents a pink histogram with 100 bins.
2. Create an object that represents a fill scale with the Blues ColorBrewer palette.
3. Read the source code for `theme_grey()`. What are its arguments? How does it work?
4. Create `scale_colour_wesanderson()`. It should have a parameter to pick the palette from the wesanderson package, and create either a continuous or discrete scale.

12.3 Multiple components

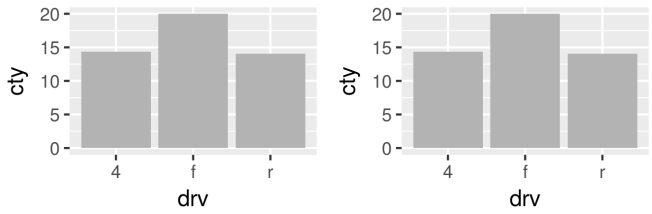
It's not always possible to achieve your goals with a single component. Fortunately, ggplot2 has a convenient way of adding multiple components to a plot in one step with a list. The following function adds two layers: one to show the mean, and one to show its standard error:

```
geom_mean <- function() {
  list(
    stat_summary(fun.y = "mean", geom = "bar", fill = "grey70"),
    stat_summary(fun.data = "mean_cl_normal", geom = "errorbar", width = 0.4)
  )
}
ggplot(mpg, aes(class, cty)) + geom_mean()
ggplot(mpg, aes(drv, cty)) + geom_mean()
```



If the list contains any `NULL` elements, they're ignored. This makes it easy to conditionally add components:

```
geom_mean <- function(se = TRUE) {
  list(
    stat_summary(fun.y = "mean", geom = "bar", fill = "grey70"),
    if (se)
      stat_summary(fun.data = "mean_cl_normal", geom = "errorbar", width = 0.4)
  )
}
ggplot(mpg, aes(drv, cty)) + geom_mean()
ggplot(mpg, aes(drv, cty)) + geom_mean(se = FALSE)
```



12.3.1 Plot components

You're not just limited to adding layers in this way. You can also include any of the following object types in the list:

- A data.frame, which will override the default dataset associated with the plot. (If you add a data frame by itself, you'll need to use `%+%`, but this is not necessary if the data frame is in a list.)
- An `aes()` object, which will be combined with the existing default aesthetic mapping.
- Scales, which override existing scales, with a warning if they've already been set by the user.
- Coordinate systems and facetting specification, which override the existing settings.
- Theme components, which override the specified components.

12.3.2 Annotation

It's often useful to add standard annotations to a plot. In this case, your function will also set the data in the layer function, rather than inheriting it from the plot. There are two other options that you should set when you do this. These ensure that the layer is self-contained:

- `inherit.aes = FALSE` prevents the layer from inheriting aesthetics from the parent plot. This ensures your annotation works regardless of what else is on the plot.
- `show.legend = FALSE` ensures that your annotation won't appear in the legend.

One example of this technique is the `borders()` function built into ggplot2. It's designed to add map borders from one of the datasets in the maps package:

```
borders <- function(database = "world", regions = ".", fill = NA,
                    colour = "grey50", ...) {
```

```
  df <- map_data(database, regions)
  geom_polygon(
    aes_(~lat, ~long, group = ~group),
    data = df, fill = fill, colour = colour, ...,
    inherit.aes = FALSE, show.legend = FALSE
  )
}
```

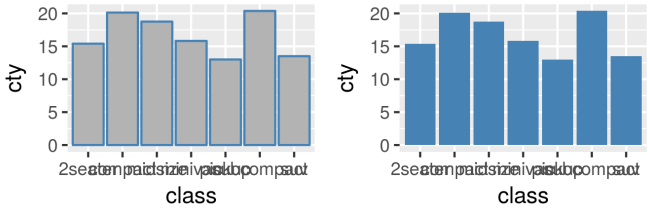
12.3.3 Additional arguments

If you want to pass additional arguments to the components in your function, ... is no good: there's no way to direct different arguments to different components. Instead, you'll need to think about how you want your function to work, balancing the benefits of having one function that does it all vs. the cost of having a complex function that's harder to understand.

To get you started, here's one approach using `modifyList()` and `do.call()`:

```
geom_mean <- function(..., bar.params = list(), errorbar.params = list()) {
  params <- list(...)
  bar.params <- modifyList(params, bar.params)
  errorbar.params  <- modifyList(params, errorbar.params)

  bar <- do.call("stat_summary", modifyList(
    list(fun.y = "mean", geom = "bar", fill = "grey70"),
    bar.params)
  )
  errorbar <- do.call("stat_summary", modifyList(
    list(fun.data = "mean_cl_normal", geom = "errorbar", width = 0.4),
    errorbar.params)
  )

  list(bar, errorbar)
}

ggplot(mpg, aes(class, cty)) +
  geom_mean(
    colour = "steelblue",
    errorbar.params = list(width = 0.5, size = 1)
  )
#> Warning: Computation failed in `stat_summary()`:
#> Hmisc package required for this function
ggplot(mpg, aes(class, cty)) +
  geom_mean(
```

```
    bar.params = list(fill = "steelblue"),
    errorbar.params = list(colour = "blue")
  )
#> Warning: Computation failed in `stat_summary()`:
#> Hmisc package required for this function
```



If you need more complex behaviour, it might be easier to create a custom geom or stat. You can learn about that in the extending ggplot2 vignette included with the package. Read it by running `vignette("extending-ggplot2")`.

12.3.4 Exercises

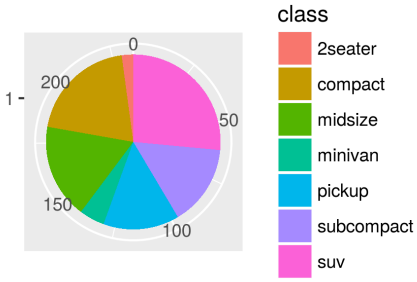
1. To make the best use of space, many examples in this book hide the axes labels and legend. I've just copied-and-pasted the same code into multiple places, but it would make more sense to create a reusable function. What would that function look like?
2. Extend the `borders()` function to also add `coord_quickmap()` to the plot.
3. Look through your own code. What combinations of geoms or scales do you use all the time? How could you extract the pattern into a reusable function?

12.4 Plot functions

Creating small reusable components is most in line with the ggplot2 spirit: you can recombine them flexibly to create whatever plot you want. But sometimes you're creating the same plot over and over again, and you don't need that flexibility. Instead of creating components, you might want to write a function that takes data and parameters and returns a complete plot.

For example, you could wrap up the complete code needed to make a piechart:

```
piechart <- function(data, mapping) {
  ggplot(data, mapping) +
    geom_bar(width = 1) +
    coord_polar(theta = "y") +
    xlab(NULL) +
    ylab(NULL)
}
piechart(mpg, aes(factor(1), fill = class))
```

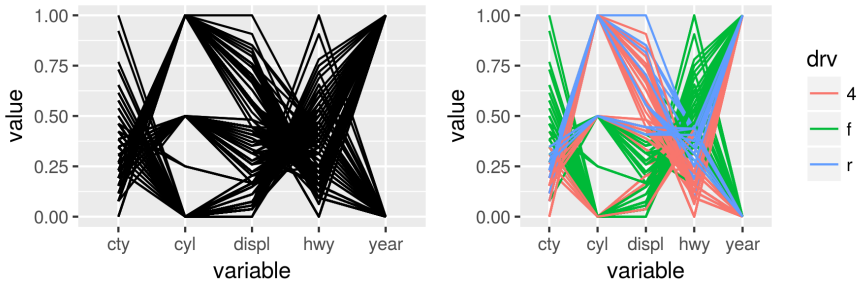


This is much less flexible than the component based approach, but equally, it's much more concise. Note that I was careful to return the plot object, rather than printing it. That makes it possible add on other ggplot2 components.

You can take a similar approach to drawing parallel coordinates plots (PCPs). PCPs require a transformation of the data, so I recommend writing two functions: one that does the transformation and one that generates the plot. Keeping these two pieces separate makes life much easier if you later want to reuse the same transformation for a different visualisation.

```
pcp_data <- function(df) {
  is_numeric <- vapply(df, is.numeric, logical(1))

  # Rescale numeric columns
  rescale01 <- function(x) {
    rng <- range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
  }
  df[is_numeric] <- lapply(df[is_numeric], rescale01)

  # Add row identifier
  df$.row <- rownames(df)

  # Treat numerics as value (aka measure) variables
```

```
  # gather_ is the standard-evaluation version of gather, and
  # is usually easier to program with.
  tidyr::gather_(df, "variable", "value", names(df)[is_numeric])
}
pcp <- function(df, ...) {
  df <- pcp_data(df)
  ggplot(df, aes(variable, value, group = .row)) + geom_line(...)
}
pcp(mpg)
pcp(mpg, aes(colour = drv))
```



A complete exploration of this idea is `qplot()`, which provides a fairly deep wrapper around the most common `ggplot()` options. I recommend studying the source code if you want to see how far these basic techniques can take you.

12.4.1 Indirectly referring to variables

The `piechart()` function above is a little unappealing because it requires the user to know the exact `aes()` specification that generates a pie chart. It would be more convenient if the user could simply specify the name of the variable to plot. To do that you'll need to learn a bit more about how `aes()` works.

`aes()` uses non-standard evaluation: rather than looking at the values of its arguments, it looks at their expressions. This makes it difficult to work with programmatically as there's no way to store the name of a variable in an object and then refer to it later:

```
x_var <- "displ"
aes(x_var)
#> * x -> x_var
```

Instead we need to use `aes_()`, which uses regular evaluation. There are two basic ways to create a mapping with `aes_()`:

- Using a *quoted call*, created by `quote()`, `substitute()`, `as.name()`, or `parse()`.

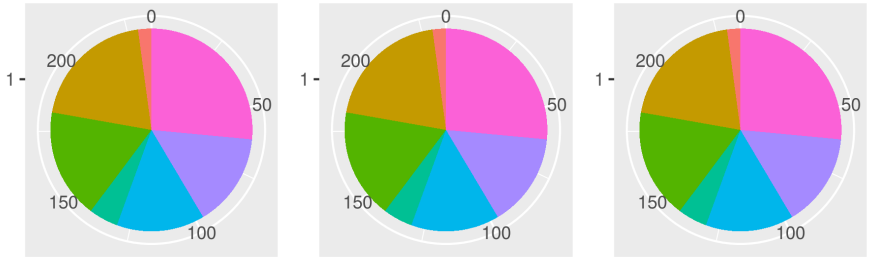
```
aes_(quote(displ))
#> * x -> displ
aes_(as.name(x_var))
#> * x -> displ
aes_(parse(text = x_var)[[1]])
#> * x -> displ

f <- function(x_var) {
  aes_(substitute(x_var))
}
f(displ)
#> * x -> displ
```

  The difference between `as.name()` and `parse()` is subtle. If `x_var` is "a + b", `as.name()` will turn it into a variable called `` `a + b` ``, `parse()` will turn it into the function call `a + b`. (If this is confusing, `http://adv-r.had.co.nz/Expressions.html` might help).
- Using a formula, created with `~`.

```
aes_(~displ)
#> * x -> displ
```

`aes_()` gives us three options for how a user can supply variables: as a string, as a formula, or as a bare expression. These three options are illustrated below

```
piechart1 <- function(data, var, ...) {
  piechart(data, aes_(~factor(1), fill = as.name(var)))
}
piechart1(mpg, "class") + theme(legend.position = "none")

piechart2 <- function(data, var, ...) {
  piechart(data, aes_(~factor(1), fill = var))
}
piechart2(mpg, ~class) + theme(legend.position = "none")

piechart3 <- function(data, var, ...) {
  piechart(data, aes_(~factor(1), fill = substitute(var)))
}
piechart3(mpg, class) + theme(legend.position = "none")
```



There's another advantage to `aes_()` over `aes()` if you're writing ggplot2 plots inside a package: using `aes_(~x, ~y)` instead of `aes(x, y)` avoids the global variables NOTE in `R CMD check`.

12.4.2 The plot environment

As you create more sophisticated plotting functions, you'll need to understand a bit more about ggplot2's scoping rules. ggplot2 was written well before I understood the full intricacies of non-standard evaluation, so it has a rather simple scoping system. If a variable is not found in the `data`, it is looked for in *the* plot environment. There is only one environment for a plot (not one for each layer), and it is the environment in which `ggplot()` is called from (i.e. the `parent.frame()`).

This means that the following function won't work because `n` is not stored in an environment accessible when the expressions in `aes()` are evaluated.

```
f <- function() {
  n <- 10
  geom_line(aes(x / n))
}
df <- data.frame(x = 1:3, y = 1:3)
ggplot(df, aes(x, y)) + f()
#> Error in x/n: non-numeric argument to binary operator
```

Note that this is only a problem with the `mapping` argument. All other arguments are evaluated immediately so their values (not a reference to a name) are stored in the plot object. This means the following function will work:

```
f <- function() {
  colour <- "blue"
  geom_line(colour = colour)
}
ggplot(df, aes(x, y)) + f()
```

If you need to use a different environment for the plot, you can specify it with the `environment` argument to `ggplot()`. You'll need to do this if you're creating a plot function that takes user provided data. See `qplot()` for an example.

12.4.3 Exercises

1. Create a `distribution()` function specially designed for visualising continuous distributions. Allow the user to supply a dataset and the name of a variable to visualise. Let them choose between histograms, frequency polygons, and density plots. What other arguments might you want to include?
2. What additional arguments should `pcp()` take? What are the downsides of how `...` is used in the current code?
3. Advanced: why doesn't this code work? How can you fix it?

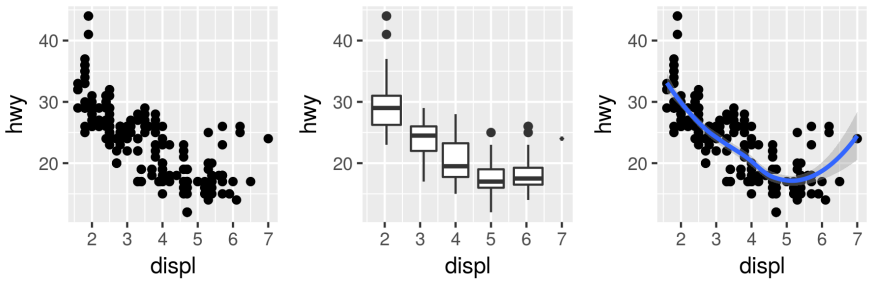
```
f <- function() {
  levs <- c("2seater", "compact", "midsize", "minivan", "pickup",
    "subcompact", "suv")
  piechart3(mpg, factor(class, levels = levs))
}
f()
#> Error in factor(class, levels = levs): object 'levs' not found
```

12.5 Functional programming

Since ggplot2 objects are just regular R objects, you can put them in a list. This means you can apply all of R's great functional programming tools. For example, if you wanted to add different geoms to the same base plot, you could put them in a list and use `lapply()`.

```
geoms <- list(
  geom_point(),
  geom_boxplot(aes(group = cut_width(displ, 1))),
  list(geom_point(), geom_smooth())
)

p <- ggplot(mpg, aes(displ, hwy))
lapply(geoms, function(g) p + g)
#> [[1]]
#>
#> [[2]]
```

```
#>
#> [[3]]
#> `geom_smooth()` using method = 'loess'
```



If you're not familiar with functional programming, read through `http://adv-r.had.co.nz/Functional-programming.html` and think about how you might apply the techniques to your duplicated plotting code.

12.5.1 Exercises

1. How could you add a `geom_point()` layer to each element of the following list?

```
plots <- list(
  ggplot(mpg, aes(displ, hwy)),
  ggplot(diamonds, aes(carat, price)),
  ggplot(faithfuld, aes(waiting, eruptions, size = density))
)
```

2. What does the following function do? What's a better name for it?

```
mystery <- function(...) {
  Reduce(`+`, list(...), accumulate = TRUE)
}

mystery(
  ggplot(mpg, aes(displ, hwy)) + geom_point(),
  geom_smooth(),
  xlab(NULL),
  ylab(NULL)
)
```

Index

R code index