ggplot2

Hadley Wickham

October 8, 2008

Contents

Chapter 1

Preface

1.1 Introduction

What is `ggplot2`?

- An R package for producing statistical graphics.
- A language, based on the Grammar of Graphics (Wilkinson, 2005), for describing and creating plots.
- A set of independent components that minimise the code needed to produce complex graphics
- Plots that can be built up iteratively and edited later

It builds on top of the grid graphics system (Murrell, 2005a), and also provides many features that take the hassle out of making graphics. For example, it produces legends automatically, makes it easy to combine data from multiple sources, to produce the same plot for different subsets of a data set.

This book provides a practical 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 the it, the more effectively you'll be able to use `ggplot2`.

This book assumes basic some 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, and turn you into an expert who can build new components to extend the grammar.

1.2 Other resources

This book does not exhaustively cover every possible use of `ggplot2`. It does not document every function in detail and it does not describe every possible plot you could make. What it does do, however, is teach you what all the pieces are and how they fit together. By the end of reading this book, you should be able to build new and unique plots specifically tailored to your needs.

However, you still need to be able to get precise details of individual components. The best resource for this will always be the built in documentation. This is accessible online, http://had.co.nz/ggplot, and from within R, using the usual help syntax. This website also lists talks and papers related to ggplot2. The CRAN website, http://cran.r-project.org/web/packages/ggplot2/, is another useful resource. This will tell you provides convenient links to what's new and different in each release.

The book website, http://had.co.nz/ggplot2/book, provides updates to this book, All graphics used on the book are displayed on the site, along with the code and data needed to reproduce them. There is also a gallery of ggplot2 graphics used in real life. If you would like your graphics to be included in the gallery, please send me reproducible code and a paragraph or two describing your plot.

1.3 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? My take on the grammar is that a 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. Faceting 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. A detailed description of the formal grammar of ggplot2 and how it differs from Wilkinson's can be found in Wickham (Tentatively accepted).

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.

- The **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.
- Geometric objects (**geom**s for short) represent what you actually see on the plot: points, lines, polygons, etc.
- Statistics transformations (**stat**s for short) summarise data in many useful ways. For example, binning and counting to create a histogram. They are optional, but very useful.
- The **scale**s map values in the data space to values in an aesthetic space, whether it be colour, or size, or shape. Scales also provide an inverse mapping, a legend or axis, to make it possible to read the original data values off the graph.
- 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 many others are available, including polar, cartographic projections, and hierarchical coordinate systems for categorical data.

- 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.

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 Chambers et al. (1983); Cleveland (1993a); Robbins (2004); Tukey (1977).

- Ironically, the grammar doesn't specify what a graphic should look like. The finer points of display, for example, font size or background colour, are not specified by the grammar. In practice, a useful plotting system will need to describe these, as `ggplot2` does. Similarly, the grammar does not specify how to make an attractive graphic, and while the defaults in `ggplot2` have been chosen with care, you may need to consult other references to create an attractive plot: Tufte (1990, 1997, 2001, 2006).

- It does not describe interaction: the grammar of graphics describes only static graphics, and there is essentially no benefit to displaying on a computer screen as opposed to on a piece of paper. `ggplot2` can only create static graphics, so for dynamic and interactive graphics you will have to look elsewhere. Cook and Swayne (2007) provides an excellent introductions the interactive graphics package GGobi. GGobi can be connected to R with the `rggobi` package (Wickham et al., Under revision).

1.4 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 "hacked" together by Ross Ihaka based on experience implementing S graphics driver and partly looking at Chambers et al. (1983). Base graphics basically has a pen on paper model: you can only draw on top of the plot, you can not 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. When you create a single scatterplot, or histogram, or a set of boxplots, you're probably using 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 layout complex graphics. Grid provides drawing primitives, but no tools for producing statistical graphics.

- The `lattice` package (Sarkar, 2008a), developed by Deepayan Sarkar, uses grid graphics to implement the trellis graphics system of Cleveland (1993a, 1985), and is a considerable improvement over base graphics. You can easily produce conditioned plots, and some plotting details (eg. 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, 2008b).

- `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 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

Many other R packages, such as `vcd` (Meyer et al., 2006), `plotrix` (Lemon et al., 2008) and `gplots` (source code and/or documentation contributed by Ben Bolker and Lumley, 2007), implement specialist graphics, but no others provide a framework for producing statistical graphics. A comprehensive resource listing all graphics functionality available in other contributed packages is the graphics task view at `http://cran.r-project.org/web/views/Graphics.html`.

1.5 About this book

- Chapter 2 describes how to quickly get started using `qplot` to make graphics, just like you can using `plot`. This chapter introduces several important `ggplot2` concepts: geoms, aesthetic mappings and faceting.

- While `qplot` is a quick way to get started, you are not using the full power of the grammar. Chapter 3 describes the layered grammar of graphics which underlies `ggplot2`. The theory is illustrated in Chapter 4 with examples showing how to build up a plot piece by piece, exercising full control over the available options. You will learn about the different components of a plot, laying the ground for the following chapters which describe these components in detail and teach you how to build your own.

- Chapter **??** describes how assemble the components of `ggplot2` to solve particular plotting problems.

- Understanding how scales works 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 5 will show you what scales are available, how to adjust their parameters, and how to create your own.

- There are three different ways to tweak the position of plots: coordinate systems, faceting and position adjustments. These are described in Chapter **??**

- Sometimes you need more control over the output than `ggplot2` provides. In this case, you will need to modify the low level grid output used to draw the graphics. In Chapter 7, you will learn how this output is constructed, how to control and modify it, and how to add additional annotations to the plot.
- Two appendices provide additional useful information. Appendix **??** describes how colours, shapes, line types and sizes can be specified by hand, and Appendix **??** shows how to translate from base graphics, lattice graphics, and Wilkison's GPL to `ggplot2` syntax.

1.6 Installation

To use`ggplot2`, you must first install it. Make sure you have a recent version of R (at least version 2.7) from `http://r-project.org`, and then run the following line of code to download and install the `ggplot2` package.

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

`ggplot2` isn't perfect, so from time to time you may encounter something that doesn't work the way it should. If this happens, please email me at h.wickham@gmail.com with a reproducible example of your problem, as well as a description of what you think should have happened. The more information you provide, the easier it is for me to help you.

1.7 Acknowledgements

Many people have contributed to this book with high-level structural insights, spelling and grammar corrections and bug reports. In particular, I would to thank: Lee Wilkinson, for discussions that cemented my understanding of the grammar; Gabor Grothendienk, for early helpful comments; Heike Hofmann and Di Cook, for being great major professors; Charlotte Wickham; the students of stat480 and stat503 at ISU, for using it; Debby Swayne, for masses of helpful feedback and advice; Bob Muenchen; Reinhold Kleigl; ...

Chapter 2

Getting started with ggplot: `qplot`

2.1 Introduction

In this chapter, you will learn to make a wide variety of plots with your first ggplot function, `qplot`, short for **q**uick plot. qplot makes it easy to produce complex plots, often requiring several lines of code using other plotting systems, in one line. qplot can do this because it's based on the grammar of graphics, which allows you to create a simple, yet expressive, description of the plot. In later chapters you'll learn to use all of the expressive power of the grammar, but here we'll start simple so you can work your way up. You will also start to learn some of the ggplot terminology that will be used throughout the book.

`qplot` has been designed be very similar to `plot`, which should make it easy if you're already familiar with plotting in R. Remember, during an R session you can get a summary of all the arguments to `qplot` with R help, `?qplot`.

In this chapter you'll learn:

- The basic use of `qplot`—If you're already familiar with `plot`, this will be particularly easy, § 2.3.
- How to map variables to aesthetic attributes, like colour, size and shape, § 2.4.
- How to create many different types of plots by specifying different geoms, and how to combine multiple types in a single plot. Page § 2.5.
- The use of faceting, also known as trellising or conditioning, to break apart subsets of your data, § 3.5.4.
- How to tune the appearance of the plot by specifying some basic options, § 2.7.
- A few important differences between `plot` and `plot`, § 2.8

2.2 Data sets

In this chapter we'll just use one data source, so you can get familiar with the plotting details rather than having to familiarise yourself with different datasets. The `diamonds` dataset consists of prices and quality information about 54,000 diamonds, and is included in the ggplot package. The dataset has not been well cleaned, so as well as demonstrating

interesting relationships about diamonds, it also demonstrates some data quality problems. We'll also use another dataset, `dsmall`, which is a random sample of 1000 diamonds. We'll use this for plots which are more appropriate for smaller datasets. The first few rows of the data are shown in Table 2.1.

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.2	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.2	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.2	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.3	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.3	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
6	0.2	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48

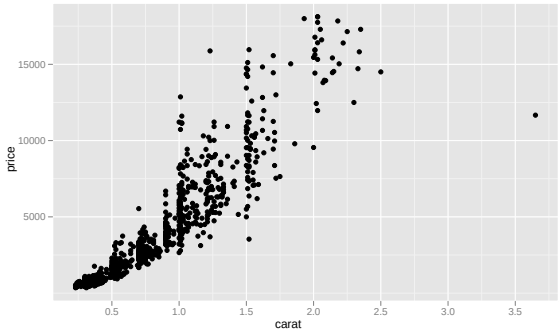
Table 2.1: `diamonds` dataset

2.3 Basic use

As with `plot`, the first two arguments to `qplot()` are `x` and `y`, giving the x- and y-coordinates for the objects on the plot. There is also an optional `data` argument. If this is specified, `qplot()` will look inside that data frame before looking for objects in your workspace. Using the `data` argument is recommended: it's a good idea to keep related data in a single data frame. If you don't specify one, `qplot()` will try to build one up for you and may find the other variables bearing the same names.

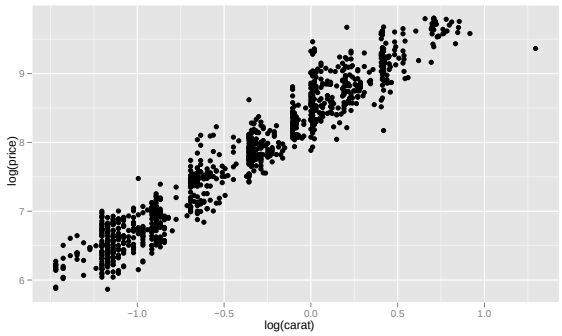
Here is a simple example of the use of `qplot()`. It produces a scatterplot showing the relationship between the price and carats (weight) of a diamond.

```
> qplot(carat, price, data=dsmall)
```



The plot shows a strong correlation with notable outliers and some interesting vertical striation. The relationship looks exponential, though, so the first thing we'd like to do is to transform the variables. Because `qplot()` accepts functions of variables as arguments, we plot log(price) vs. log(carat):

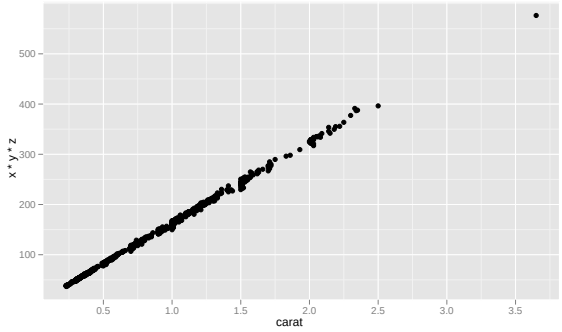
```
> qplot(log(carat), log(price), data=dsmall)
```



The relationship now looks linear. With this much overplotting, though, we need to be cautious about drawing firm conclusions.

Arguments can also be combinations of existing variables, so, if we are curious about the relationship between the volume of the diamond (approximately $x \times y \times z$) and its weight, we can look at the following:

```
> qplot(carat, x * y * z, data=dsmall)
```



We would expect the density of diamonds to be constant, and thus to see a linear relationship between volume and weight. The majority of diamonds do seem to fall along a line, but there are some large outliers.

2.4 Colour, size, shape and other aesthetic attributes

The first big difference when using `qplot` instead of `plot` comes when you want to assign colours—or sizes or shapes—to the points on your plot. With `plot`, it's your responsibility to convert a categorical variable in your data (e.g., "apples", "bananas", "pears") into something that `plot` knows how to use (e.g., "red", "yellow", "green"). `qplot` can do this for you automatically, and it will automatically provide a legend that maps the displayed attributes to the data values. This makes it easy to include additional data on the plot.

In the next example, we augment the plot of carat and price with information about diamond colour, clarity and cut.

Colour, size and shape are all examples of aesthetic attributes, visual properties that affect the way observations are displayed. For every aesthetic attribute, there is a function, called a *scale*, which maps data values to valid values for that aesthetic. It is this scale that

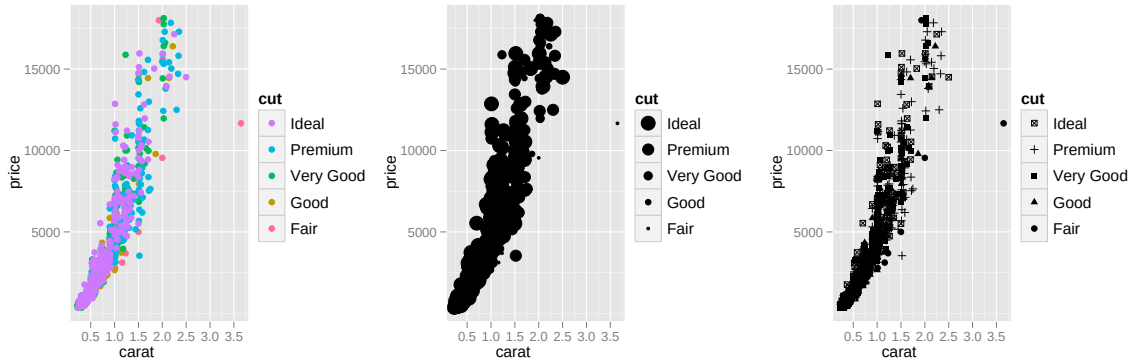


Figure 2.1: Mapping colour (using the argument "colour=cut", left), size ("size=cut", middle) and shape ("shape=cut", right) of points to quality of cut.

controls how the points appear. For example, in the above plots, the colour scale maps J to purple and F to green. (Note that while I use British spelling throughout this book, the software also accepts American spellings.)

You can also manually set the aesthetics using `I()`. For example, `colour = I("red")` or `size = I(2)`. This is different from mapping and is explained in more detail in Section **??**. For large data sets, like the diamonds data, semi-transparent points are often useful to alleviate some of the overplotting. To make a semi-transparent colour you can use `alpha(colour, transparency)`, where `colour` is an R colour (described in Appendix A) and transparency is a value between 0 (completely transparent) and 1 (complete opaque). It's often useful to specify the transparency as a fraction, e.g. `1/10` or `1/20`, as the denominator specifies the number of points that must overplot to get a completely opaque colour.

Scales are the essential difference between setting and mapping - mapping uses a scale, setting does not.

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 better with continuous variables. The amount of data also makes a difference: if there is a lot of data, like in the plots above, it can be hard to distinguish the different groups. An alternative solution is to use faceting, which will be introduced in Section 3.5.4.

2.5 Plot geoms

`qplot` is not limited to scatterplots, but can produce almost any kind of plot by varying the **geom** argument. Geom, short for geometric object, describes the type of object that is used to display the data. Some geoms have an associated statistical transformation, for example, a histogram is a binning statistic plus a bar geom. These different components are described in the next chapter. Here we'll introduce you to the most common and useful geoms, divided up by the dimensionality of data that they work with. The following geoms enable you to investigate two-dimensional relationships:

- `geom="point"` draws points to produce a scatterplot (the default), as described above.

- `geom="smooth"` fits a smoother to the data and displays the smooth and its standard error, § 2.5.1.
- `geom="quantile"` displays conditional density estimates. You can think of this as a extension of boxplots to deal with the case of a continuous conditioning variable, § 2.5.2.
- `geom="density2d"` adds contours of a 2d density estimate. This is very useful when you have a lot of overplotting, § 2.5.3
- `geom="boxplot"` produces a box and whisker plot to summarise the distribution of a set of points, § 2.5.4
- `geom="path"` and `geom="line"` draw lines between the data points. Traditionally these are used to explore relationships between time and another variable, but lines may to be use to join observations connected in some other way. A line plot is constrained to produce lines that travel from left to right, while paths can go in any direction. § 2.5.7.

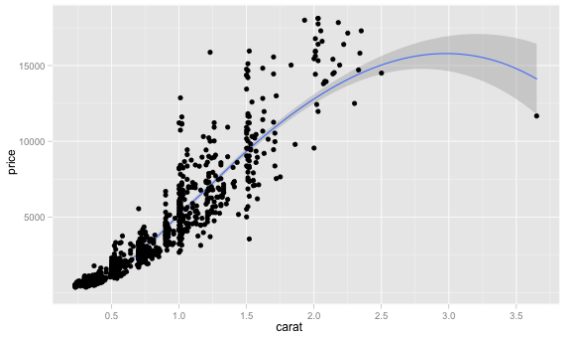
For 1d distributions, your choice of geoms is guided by the variable type:

- For continuous variables, `geom="histogram"` draws a histogram and `geom="density"` creates a density plot, § **??**.
- For discrete variables, `geom="bar"` makes a barchart, § 2.5.6.

2.5.1 Adding a smoother to a plot

If you have a scatterplot with many data points, it can be hard to see exactly what trend is shown by the data. In this case you may want to add a smoothed line to the plot. This is easily done using the `smooth` geom:

```
> qplot(carat, price, data=dsmall, geom=c("smooth", "point"))
```



Despite overplotting, our impression of a exponential relationship between price and carat was correct, but we'll have to do something about that influential outlier.

Notice that we have combined multiple geoms by supplying a vector of geom names to `qplot`. The geoms will be overlaid in the order in which they appear. By default, a

point-wise confidence interval is shown with a grey band around the smoother. If you want to turn it off, use `se=FALSE`.

There are many different smoothers you can choose using the `method` argument:

- `method="loess"`, the default, uses a smooth local regression. More details about the algorithm used can be found in `?loess`. The wiggliness of the line is controlled by the span parameter, which ranges from 0 (exceeding wiggly) to 1 (not so wiggly), as shown in Figure 2.5.1. Loess does not work well for large datasets (it's $O(n^2)$ in memory), so you'll need to use one of the other methods listed below.

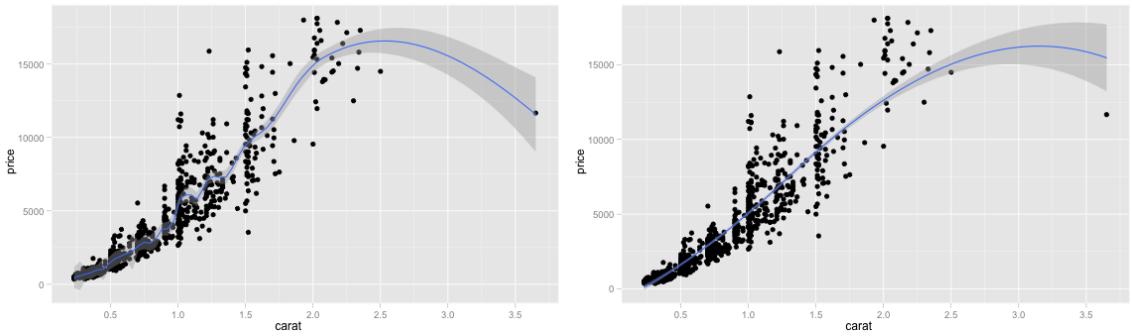


Figure 2.2: The effect of the span parameter. (Left) `span = 0.1`, and (right) `span = 1`

- `method="lm"` fits a linear model. The default will fit a straight line to your data, or you can specify `formula = y ~ poly(x, 2)` to specify a degree 2 polynomial, or better, load the `splines` library and use a natural spline: `formula = y ~ ns(x, 2)`. The second parameter is the degrees of freedom: a higher number will create a wigglier curve. You are free to specify any formula involving x and y.

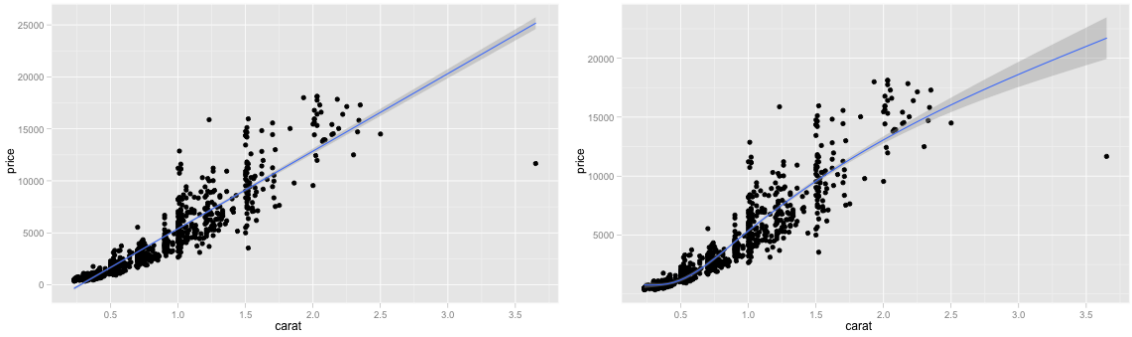


Figure 2.3: The effect of the formula parameter, using a linear model as a smoother. (Left) `formula = y ~ x`, the default; (Right) `formula = y ~ ns(x, 3)`

- `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.

- You could also load the `mgcv` library and use `method="gam", formula = y ~ s(x)` to fit a generalised additive model. This is similar to using a spline with `lm`, but the degree of smoothness is estimated from the data. For large data, use the formula `y ~ s(x, bs="cr")`

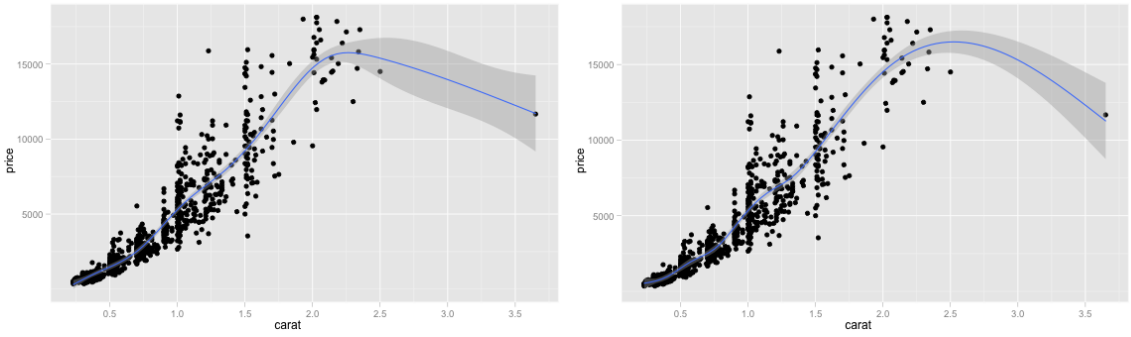


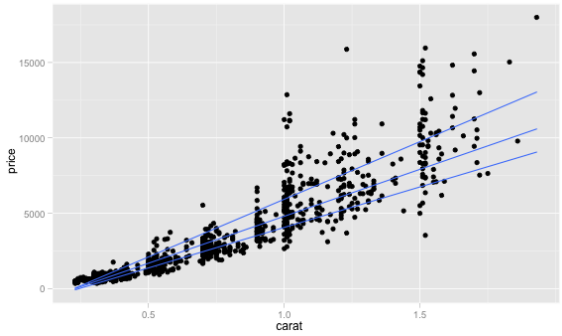
Figure 2.4: The effect of the formula parameter, using a linear model as a smoother. (Left) `formula = y ~ s(x)`, the default; (Right) `formula = y ~ s(x, bs="cr")`

2.5.2 Quantiles

We have just seen one example of the use of `qplot` to enhance plots of raw data with distributional statistics: the smoothed conditional mean and standard error. It is equally easy to add quantiles to describe the spread of the data. We can do this with quantile regression (Koenker, 2005), which estimates smoothed conditional distributions. The functional form is more limited than for the smooth geom, but we can learn more about the conditional distribution.

For this example we will zoom in on a small range of diamond weights to look more closely at the distribution of price vs. carat.

```
> dlittle <- subset(dsmall, carat < 2)
> qplot(carat, price, data=dlittle, geom=c("point", "quantile"))
```



The relationship between x and y is assumed by default to be linear, as shown in the above plot, but we can adjust the relationship using the `formula` argument, as we did for smoothing. Figure 2.5.2 shows a natural spline at left and a linear model with fixed break points at right. By default only the median and upper and lower quartiles are shown.

In Figure 2.5.2, we have added 5%, 15%, ..., 95% quantiles using the argument `quantiles = seq(0.05, 0.95, 0.1)`.

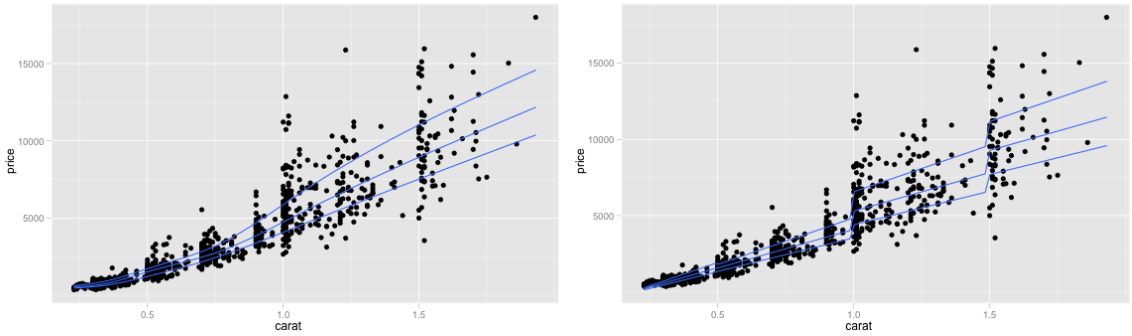


Figure 2.5: The `formula` argument is used to control the functional form of the relation. (LEFT) A natural spline with five degrees of freedom, `formula = y ~ ns(x, 5)` and (RIGHT) a linear model with break points at 0.5, 1, and 1.5 carats, `formula = y ~ x + I(x > 0.5) + I(x > 1) + I(x > 1.5))`

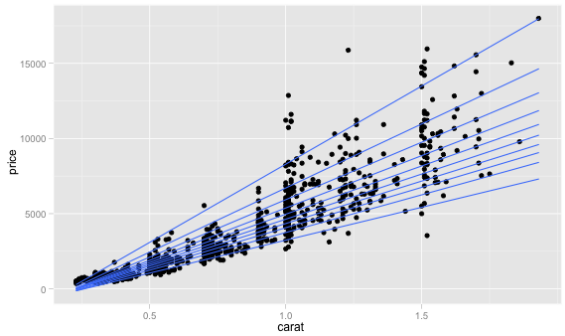


Figure 2.6: Showing 5%, 15%, ..., 95% quantiles with `quantiles = seq(0.05, 0.95, 0.1)`

2.5.3 Two-dimensional density contours

When a scatterplot suffers from overplotting, it is hard to judge the relative density of points. This can happen even when the number of points is not large, as with the scatterplot of price vs. carat for a subset of the data. One solution is to supplement the plot with contour lines from a 2d density estimate, using `geom=c(''point'', ''density2d'')`. Figure **??** shows the resulting plot for all the diamonds and makes it clear that most of them are relatively small and inexpensive.

The density estimation is carried out by `kde2d()`, which is described in more detail in its documentation.

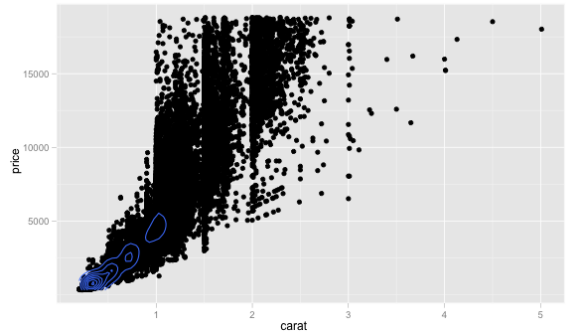


Figure 2.7: Scatterplot of price vs. carat supplemented with contours of a 2d density estimate (`geom=c(‘‘point’’, ‘‘density2d’’)`). Most diamonds are small and cheap.

2.5.4 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. Box plots and jittered points offer two ways to do this. Figure 2.5.4 explores how the distribution of price per carat varies with the colour of the diamond using jittering (`geom=‘‘jitter’’`, left) and box and whisker plots (`geom=’’boxplot’’`, right).

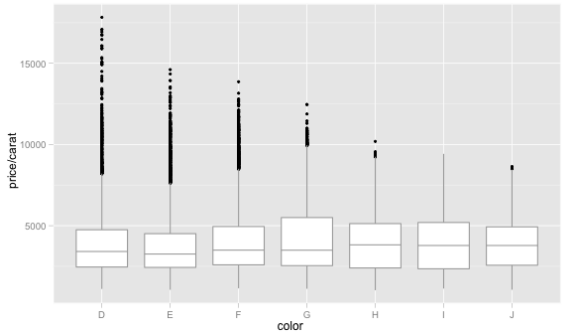


Figure 2.8: Using jittering (left) and boxplots (right) to investigate the distribution of price per carat conditional on colour. As the colour improves (from left to right) the spread of values decreases, but there is little change in the middle half of the distribution.

Each method has its strengths and weaknesses. Boxplots summarise the bulk of the distribution with only five numbers, while jittered plots can suffer from overplotting. In the example here, both plots show the dependency of the spread of price per carat on diamond colour, but the boxplots are more informative, indicating that there is very little change in the median and adjacent quartiles.

The overplotting seen in the plot of jittered values can be alleviated somewhat by using semi-transparent points using the `colour` argument. Figure 2.5.4 illustrates three different levels of transparency, which make it easier to see where the bulk of the points lie. For the leftmost plot, for example, the command is:

```
qplot(color, price/carat, data=diamonds, geom=''jitter'', colour=I(alpha(``black'', 1/5)))
```

This technique can't show the positions of the quantiles as well as a boxplot can, but it may reveal other features of the distribution that a boxplot can not.

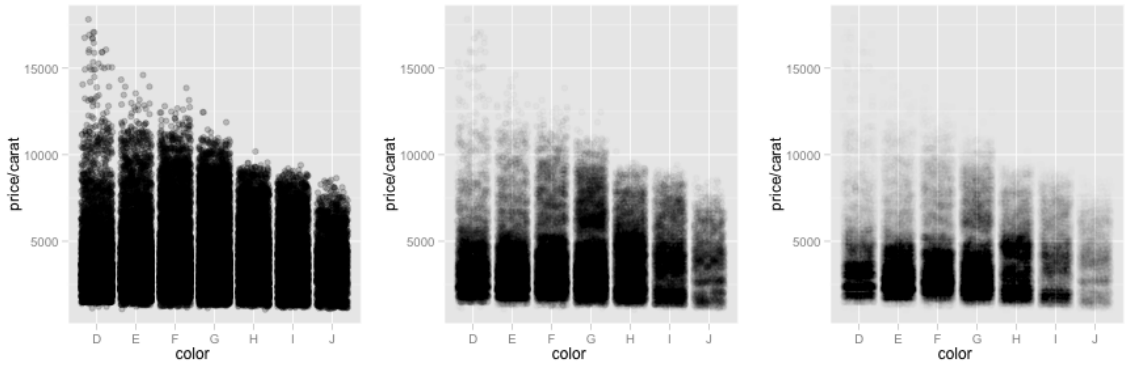


Figure 2.9: Varying the alpha level. From left to right: 1/5, 1/50, 1/200. As the opacity decreases we begin to see where the bulk of the data lies. However, the boxplot still does much better.

For jittered points, `qplot` offers the same control over aesthetics as it does for a normal scatterplot: `size`, `colour`, and `shape`. For boxplots you can control the outline colour (`colour`), the internal fill (`fill`) and the size of the lines (`size`).

Another way to look at conditional distributions is to use faceting to plot a separate histogram or density plot for each value of the categorical variable. This is demonstrated in Section 3.5.4.

2.5.5 Histogram and density plots

Histogram and density plots show the distribution of a single variable. They provide more information about the distribution of a single group than boxplots do, but it is harder to compare many groups (although we will look at one way to do so). Figure 2.5.5 shows the distribution of carats with a histogram and a density plot.

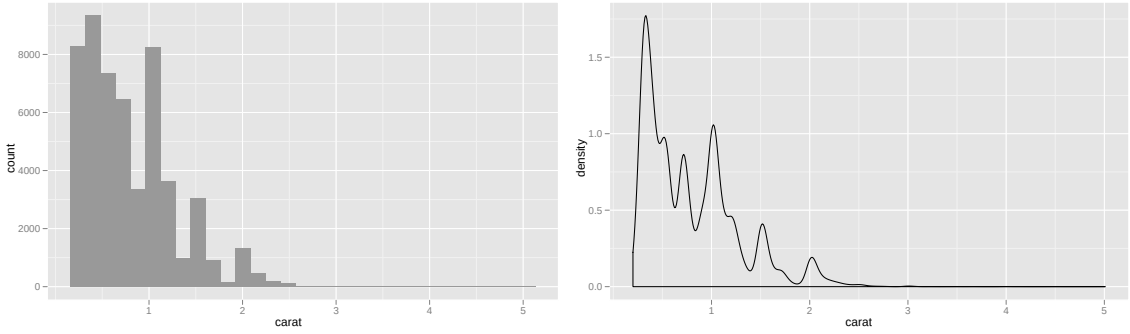


Figure 2.10: Displaying the distribution of diamonds. (Left) `geom = "histogram"` and (right) `geom = "density"`

For the density plot, the `adjust` argument controls the bandwidth of the smoother (high values of `adjust` produce smoother plots). For the histogram, the `binwidth` argument controls the amount of smoothing by setting the bin size. (Break points can also be specified explicitly, using the `breaks` argument.) It is **very important** to experiment with the level of smoothing. With a histogram you should try many bin widths: You may find that gross features of the data show up well at a large bin width, while finer features require a very narrow width.

In Figure 2.5.5, we experiment with three values of `binwidth`: 1.0, 0.1, and 0.01. It is only in the plot with the smallest bin width (right), that we see the striations we noted in an earlier scatterplot, most at "nice" numbers of carats. The full command is:

```
qplot(carat, data=diamonds, geom="histogram", binwidth=1, xlim=c(0,3))
```

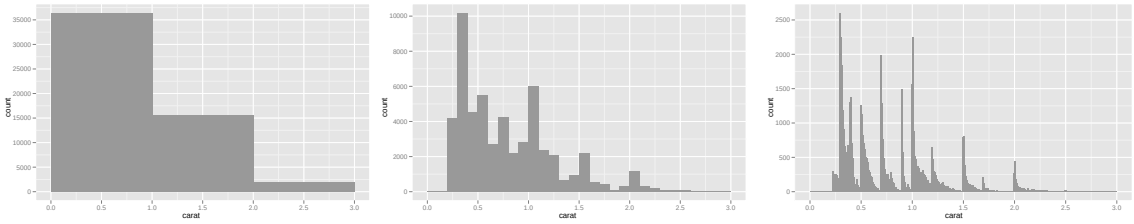


Figure 2.11: Varying the bin width on a histogram of carat reveals interesting patterns. Binwidths from left to right: 1, 0.1, and 0.01 carats. Only diamonds with carats between 0 and 3 shown.

To compare the distributions of different subgroups, just add an aesthetic mapping, as follows:

```
qplot(carat, data=diamonds, geom=''density'', colour=color, size=I(1.5))
```

Mapping a categorical variable to an aesthetic will automatically split up the geom by that variable, so this command instructs qplot to draw a density plot for each level of diamond color, and to draw each curve using a different colour and a fixed line width of 1.5. It produces the first plot in Figure 2.5.5.

The second plot is produced with a similar command:

```
qplot(carat, data=diamonds, geom=''histogram'', fill=color)
```

The density plot is more appealing at first because it seems easy to read and compare the various curves. However, it is more difficult to understand exactly what a density plot is showing. In addition, the density plot makes some assumptions that may not be true for our data; i.e., that it is unbounded, continuous and smooth.

2.5.6 Bar charts

The discrete analogue of histogram is the bar chart, `geom = ''bar''`. The bar geom counts the number of instances of each class so that you don't need to tabulate your values beforehand with `barchart` in base R. If the data has already been tabulated or if you'd like to tabulate class members in some other way, such as by summing up a continuous variable, you can use the `weight` geom. This is illustrated in Figure **??**. The first plot is a simple bar chart of diamond colour, and the second is a bar chart of diamond colour weighted by carat:

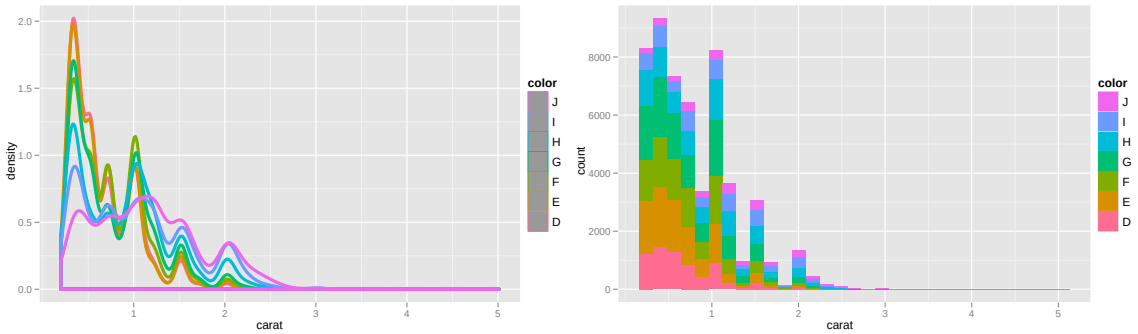


Figure 2.12: Mapping a categorical variable to an aesthetic will automatically split up the geom by that variable. (Left) Density plots are overlaid and (right) histograms are stacked.

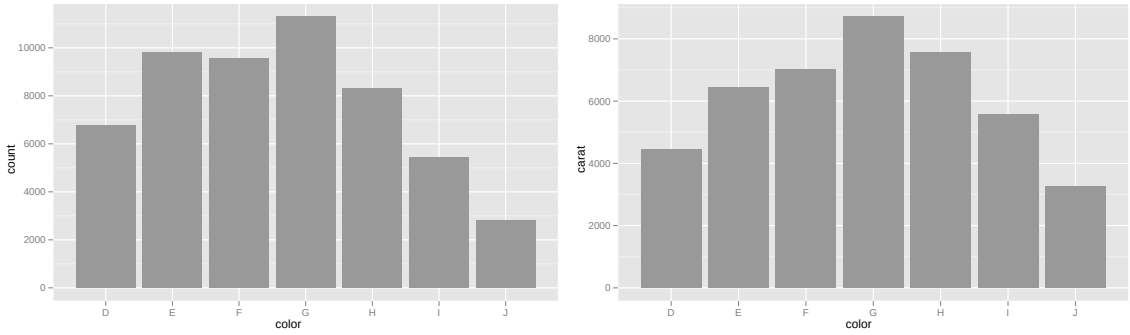


Figure 2.13: Bar charts of diamond colour. The left plot shows counts and the right plot is weighted by `weight = carat` to show the total weight of diamonds of each colour.

2.5.7 Time series with line and path plots

Line and path plots are typically used for time series data. Line plots always join the points from left to right, while path plots join them in the order that they appear in the data set (a line plot is just 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 the points are joined together.

Because there is no time variable in the diamonds data, we use the `economics` dataset, which contains economic data on the US measured over the last 40 years. Figure 2.5.7 shows two plots of unemployment over time, both produced using `geom="line"`. The first shows an unemployment rate and 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.

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.

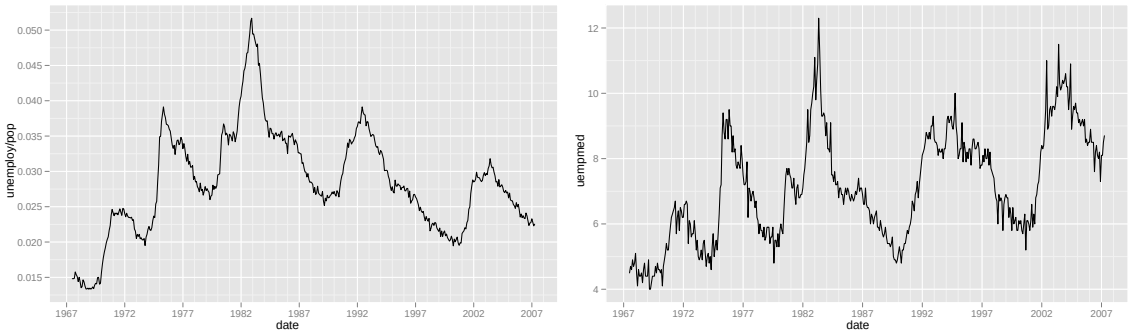


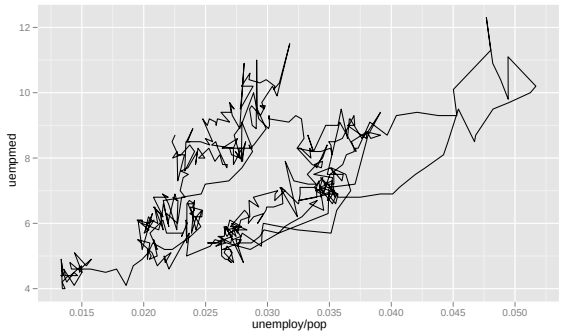
Figure 2.14: Two time series measuring amount of unemployment. (LEFT) Percent of population that is unemployed and (RIGHT) median number of weeks unemployed. Plots created with `geom="line"`.

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 apply the `size` aesthetic to the line, increasing its width as time advances.

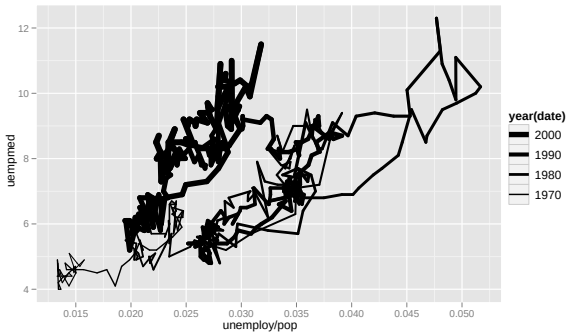
```
qplot(unemploy/pop, uempmed, data=economics, geom="path", size=year(date))
```

We can see that percent unemployed and length of unemployment is highly correlated, although in recent years the length of unemployment has been increasing relative to the unemployment rate.

```
> year <- function(x) as.POSIXlt(x)$year + 1900
> qplot(unemploy/pop, uempmed, data=economics, geom="path")
```




```
> qplot(unemploy/pop, uempmed, data=economics, geom="path", size=year(date))
```



With longitudinal data, you often want to display multiple time series on each plot, each series representing one individual. To do this with `qplot()`, 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 4.5.3.

2.6 Faceting

We have already discussed using aesthetics (colour and symbol) to compare subgroups, drawing all groups on the same plot. Faceting takes an alternative approach: It creates tables of graphics by splitting the data into subsets and displaying the same graph for each subset in an arrangement that facilitates comparison. Section **??** discusses the advantages and disadvantages of these two methods in more detail.

The default faceting method in `qplot()` creates plots arranged on a grid specified by a faceting formula which looks like `row_var ~ col_var`. You can specify as many row and column variables as you like, keeping in mind that using more than two variables will often produce a plot so large that it is difficult to see on screen. To facet on only one of columns or rows, use `.` as a place holder. For example, `row_var ~ .` will create a single column with multiple rows.

Figure 2.6 illustrates this technique with two plots, sets of histograms showing the distribution of carat conditional on colour. This command generates the first column of plots, which shows counts:

```
qplot(carat, data=diamonds, facets=color   ., geom="histogram", binwidth=0.1,
xlim=c(0, 3))
```

The second set of histograms shows proportions, making it easier to compare distributions regardless of the relative abundance of diamonds of each colour. High-quality diamonds (colour D) are skewed towards small sizes, and as quality declines the distribution becomes more flat.

2.7 Other options

These are a few other `qplot` options to control the graphic's appearance. These all have the same effect as their `plot` equivalents:

- `xlim`, `ylim`: set limits for the x- and y-axes, each a numeric vector of length two, e.g. `xlim=c(0, 20)` or `ylim=c(-0.9, -0.5)`.

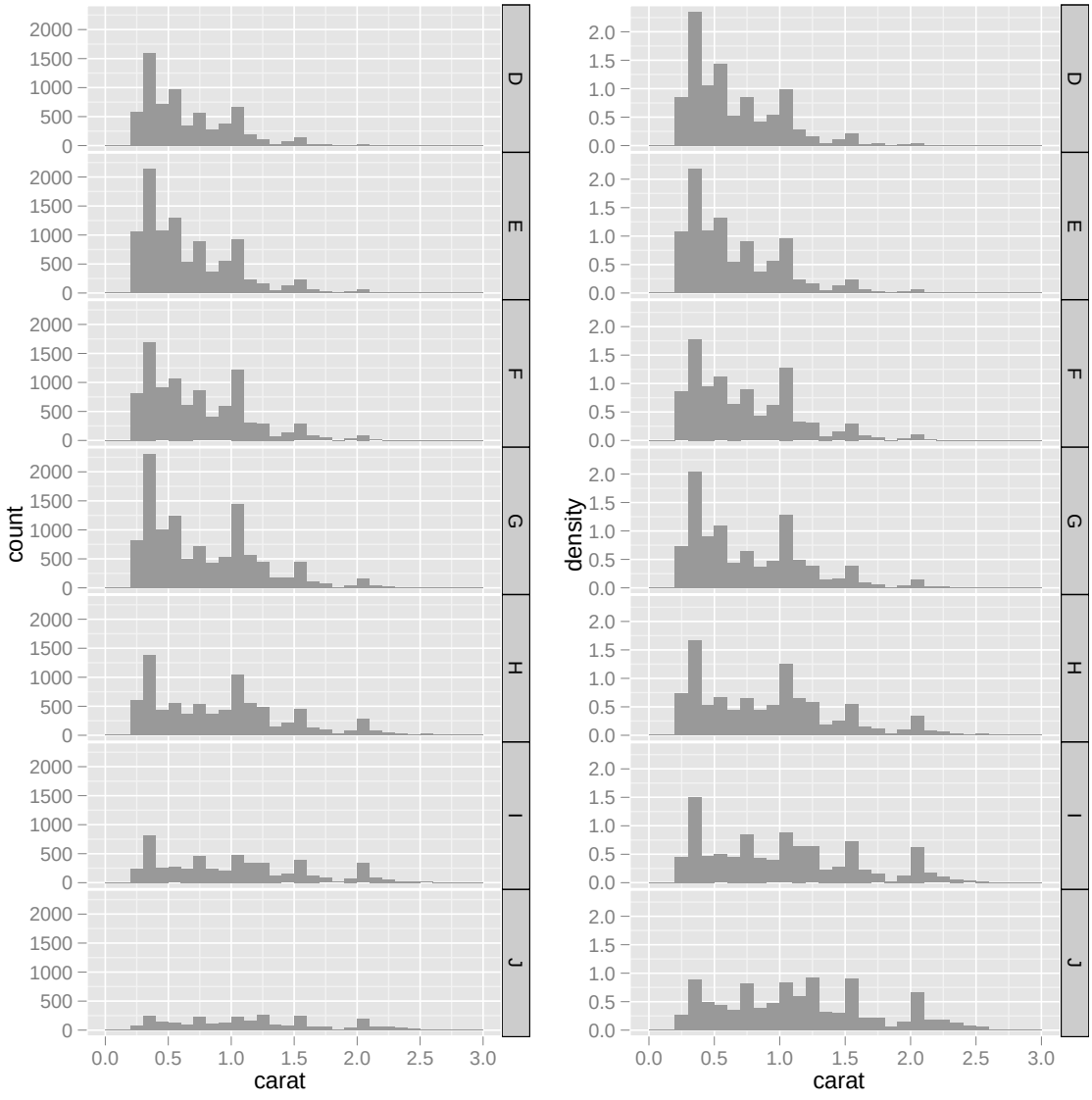
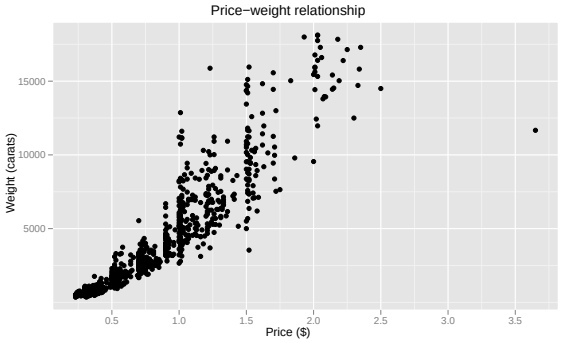


Figure 2.15: Histograms showing the distribution of carat conditional on colour. (Left) Bars show counts and (Right) bars show densities (proportions of the whole). The density plot makes it easier to compare distributions ignoring the relative abundance of diamonds within each colour. Facets created with `facets = colour ~ .`

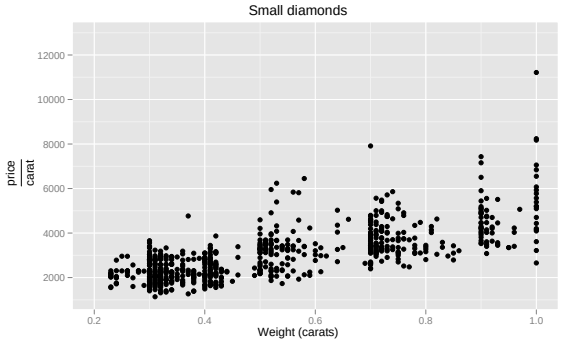
- `log`: a character vector indicating which (if any) axes should be logged. For example, `log="x"` will log the x-axis, `log="xy"` will log both.
- `main`: main title for the plot, centered in large text at the top of the plot. This can be a string (eg. `main="plot title"`) or an expression (eg. `main = expression(beta[1] == 1)`). See `?plotmath` for more examples of using mathematical formulae.
- `xlab`, `ylab`: labels for the x- and y-axes. As with the plot title, these can be character strings or mathematical expressions.

The following examples show the options in action.

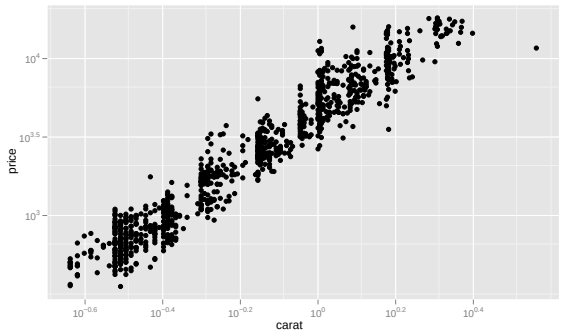
```
> qplot(
+   carat, price, data=dsmall,
+   xlab="Price ($)", ylab="Weight (carats)",
+   main="Price-weight relationship"
+ )
```




```
> qplot(
+    carat, price/carat, data=dsmall,
+    ylab = expression(frac(price,carat)),
+    xlab = "Weight (carats)",
+    main="Small diamonds",
+    xlim = c(.2,1)
+ )
```




```
> qplot(carat, price, data=dsmall, log="xy")
```



2.8 Differences from plot

There are a few important differences between `plot` and `qplot`:

- Because `qplot` can produce both 1d and 2d plots, you must always specify both x and y for 2d plots. This is different to the behaviour of plot, which uses `seq_along(y)` for x if it is not explicitly specified.
- `qplot` is not generic: you can not pass any type of R object to qplot and expect to get some kind of default plot. Note, however, that `ggplot()` is generic, and may provide a starting point for producing visualisations of arbitrary R objects.
- Values passed to the aesthetic attributes are mapped to values by scales. If you want the values to be interpreted directly, surround use `I()`: `colour = I("red")`. See also `scale_manual`, Page **??**.
- While you can continue to use the base R aesthetic names (`col`, `pch`, `cex`, etc.), it's a good idea to switch to the more descriptive ggplot2 aesthetic names (`colour`, `shape`, and `size`).
- To add further graphic elements to a plot produced in base graphics, you can use `points()`, `lines()` and `text()`. With `ggplot2`, you need to add additional **layers** to the existing plot, described in the next chapter.

Chapter 3

Mastering the grammar

3.1 Introduction

You can choose to use just `qplot()`, without any understanding of the underlying grammar, but you will not be able to use the full power of ggplot. By learning more about the grammar, and the components that make it up, you will be able to create a wider range of plots, as well as being able to combine multiple sources of data, and customise to your heart's content. You may want to skip this chapter in a first reading of the book, coming back to it when you want a deeper understanding of how all the pieces fit together.

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 smoothly into the R environment. The differences between the layered grammar and Wilkinson's grammar are described fully in (Wickham, 2008), and a guide for converting between GPL and `ggplot2` is included in Appendix **??**. 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.

This chapter begins by describing in detail the process of drawing a simple plot. Section 3.3 start with a simple scatterplot, then in Section 3.4 make 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 3.5. The chapter concludes with Section 3.6, which describes how the various components map to data structures in R.

3.2 Fuel economy data

Consider the fuel economy dataset illustrated in Table 3.1. It records make, model, class, engine size, transmission and fuel economy for a selection of US cars in 1999 and 2008. It contains the 38 models that had a new release every year, an indicator that the car was a popular model. These models are the Audi A4, Audi A4 Quattro, Audi A6 Quattro, Chevrolet C1500 Suburban 2wd, Chevrolet Corvette, Chevrolet K1500 Tahoe 4wd, Chevrolet Malibu, Dodge Caravan 2wd, Dodge Dakota Pickup 4wd, Dodge Durango 4wd, Dodge Ram 1500 Pickup 4wd, Ford Expedition 2wd, Ford Explorer 4wd, Ford F150 Pickup 4wd, Ford Mustang, Honda Civic, Hyundai Sonata, Hyundai Tiburon, Jeep Grand Cherokee

manufacturer	model	disp	year	cyl	cty	hwy	class
audi	a4	1.8	1999	4	18	29	compact
audi	a4	1.8	1999	4	21	29	compact
audi	a4	2.0	2008	4	20	31	compact
audi	a4	2.0	2008	4	21	30	compact
audi	a4	2.8	1999	6	16	26	compact
audi	a4	2.8	1999	6	18	26	compact
audi	a4	3.1	2008	6	18	27	compact
audi	a4 quattro	1.8	1999	4	18	26	compact
audi	a4 quattro	1.8	1999	4	16	25	compact
audi	a4 quattro	2.0	2008	4	20	28	compact

Table 3.1: The first 10 cars in the `mpg` data set, included in the `ggplot2` package. `cty` and `hwy` record miles per gallon (mpg) for city and highway driving respectively.

4wd, Land Rover Range Rover, Lincoln Navigator 2wd, Mercury Mountaineer 4wd, Nissan Altima, Nissan Maxima, Nissan Pathfinder 4wd, Pontiac Grand Prix, Subaru Forester Awd, Subaru Impreza Awd, Toyota 4runner 4wd, Toyota Camry, Toyota Camry Solara, Toyota Corolla, Toyota Land Cruiser Wagon 4wd, Toyota Toyota Tacoma 4wd, Volkswagen Gti, Volkswagen Jetta, Volkswagen New Beetle, and Volkswagen Passat. This data was collected from the EPA fuel economy website, `http://fueleconomy.gov`.

This dataset suggests many interesting questions. How are engine size and fuel economy related? Has fuel economy improved in the last ten years? We will try to answer the first question and in the process learn more detail about how the scatterplot is created.

3.3 Building a scatterplot

Consider Figure 3.3, one attempt to answer this question. It is a scatterplot of two continuous variables, with points coloured by a third variable. From your experience in the previous chapter, you should have a pretty good feel for how to create this plot with `qplot()`. But what is going on underneath the surface? How does `ggplot2` draw this plot?

Mapping aesthetics to data

What 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 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 called **aesthetics**, and are the properties that can perceived on the graphic. Each aesthetic can be mapped to a variable, or set to a constant value. In Figure 3.3 x-position is mapped to displ, y-position to hwy and colour to cyl. 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. Table 3.2 shows the first 10 rows of the data behind Figure 3.3. This new dataset encapsulates

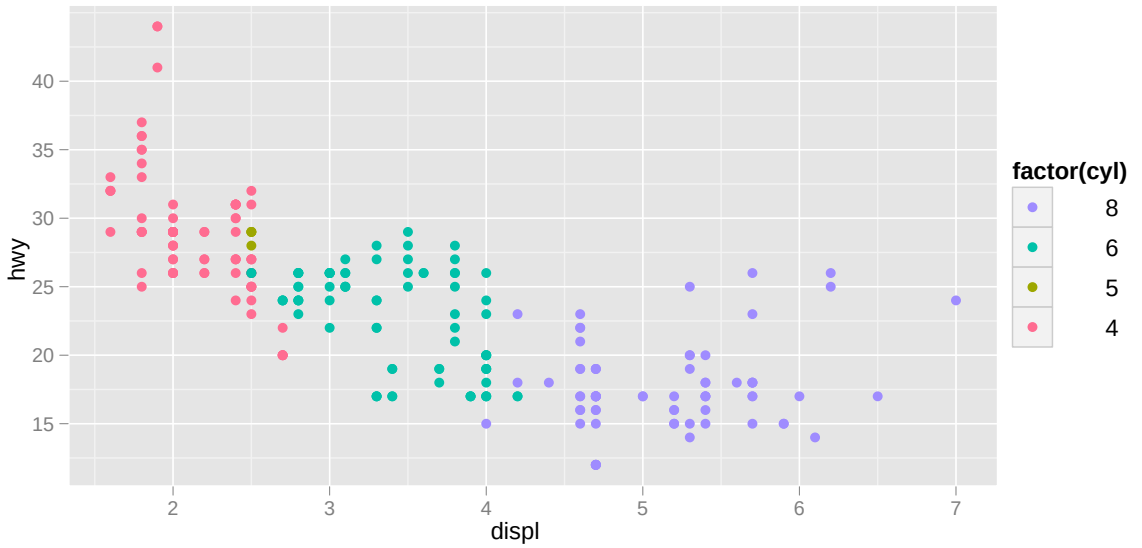


Figure 3.1: A scatterplot of engine displacement in litres (displ) vs average highway miles per gallon (hwy). Points are coloured according to number of cylinders. This plot summarises the most important factor governing fuel economy: engine size

the combination of the original data and the aesthetic mappings. We can create many different types of plots using this data. The scatterplot uses points, but we were instead to draw lines we would get a line plot. If we used bars, we'd get a bar plot. Neither of those examples make sense for this data, but we could still draw them, as in Figure 3.3. Much like in English, using `ggplot2` it is still possible to produce grammatically valid plots that don't make any sense.

x	y	colour
1.8	29	4
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
1.8	25	4
2.0	28	4

Table 3.2: First 10 rows from `mpg` rearranged into format for scatterplot. This is all the information we need to draw the scatterplot.

Bars, lines and points are all examples of geometric objects, or **geom**s. Geoms determine

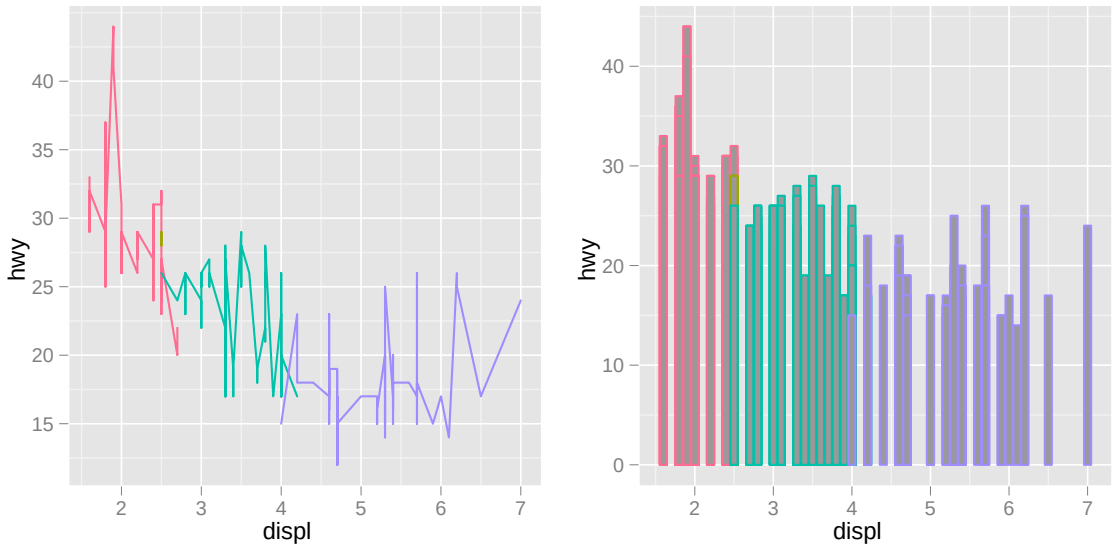


Figure 3.2: Instead of using points to represent the data, we could use other geoms like, *left,* lines or, *right,* bars. Neither of these geoms make much sense for this data, but they are still valid graphics.

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

Table 3.3: A selection of named plots and the geoms that they correspond to.

the "type" of the plot. Plots that use a single geom are often given a special name, a few of which are listed in Table 3.3. More complex plots with combinations of multiple geoms don't have a special name, and we have to describe them by hand. For example, Figure 3.3 overlays a per group regression line on the existing plot. What would you call this plot? Once you've mastered the grammar, I think you'll find that most of the plots that you produce are uniquely tailored to your problems and will no longer have common names.

Scaling

The values in Table 3.2 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 physical units (e.g. pixels and colours) that the computer can display. This conversion process is called **scaling** and performed by (surprise!) scales. Appendix **??** describes the way that specifies values for colours, sizes, shapes and so on.

For scales that control the horizontal and vertical position, we need an additional step

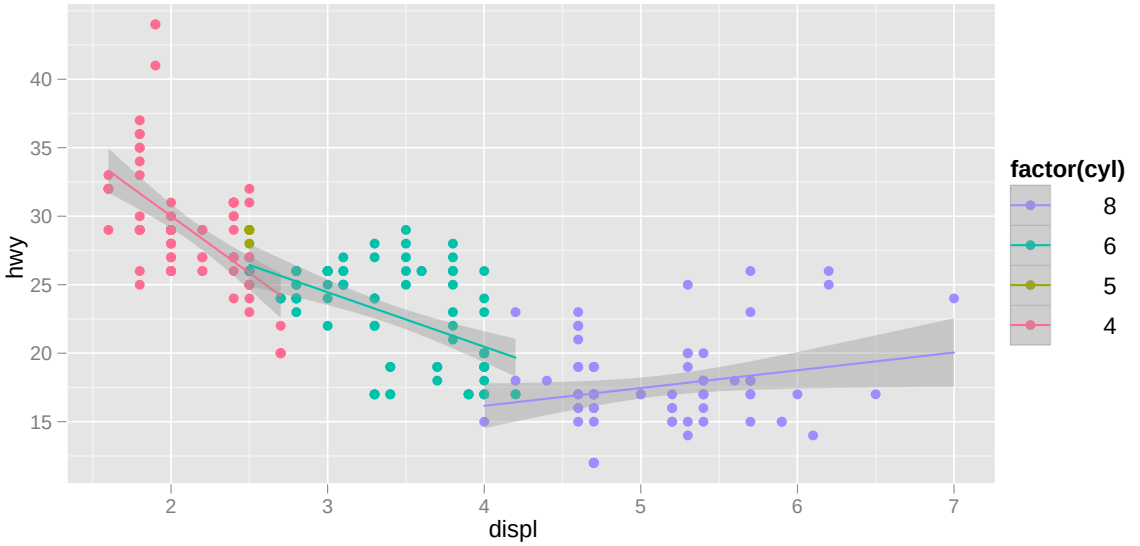


Figure 3.3: More complicated plots don't have their own names. This plot takes Figure 3.3 and adds a regression line to each group. What would you call this plot?

which determines how the two positions (x and y) combine to form the final position on the plot. This is done by the coordinate system, or **coord**. In most cases this will be Cartesian coordinates, but it might by polar coordinates, or a spherical projection used for a map.

Scaling position is easy in this example because we are using the default linear scales and Cartesian coordinate system. We only need 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.

The process for mapping the colour is a little more complicated, as we have a non-numeric result: colours. However, colours can be parameterised numerically, typically with three values. For discrete values, the default colour scale maps the to evenly spaced hues on a colour wheel, as shown in Figure 3.4.

The result of these conversions is Table 3.4, which contains values that have meaning to the computer. As well as the variables from the aesthetic mapping, we have also included the default values for the geom. We need these so that the aesthetics for each point are completely specified.

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: *data*, represented by the point geom; *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. Figure 3.5 removes the contribution of the data to show what elements the scales and plot contribute.

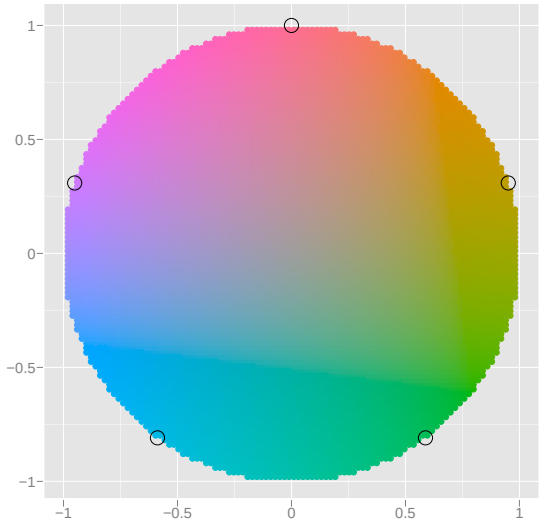
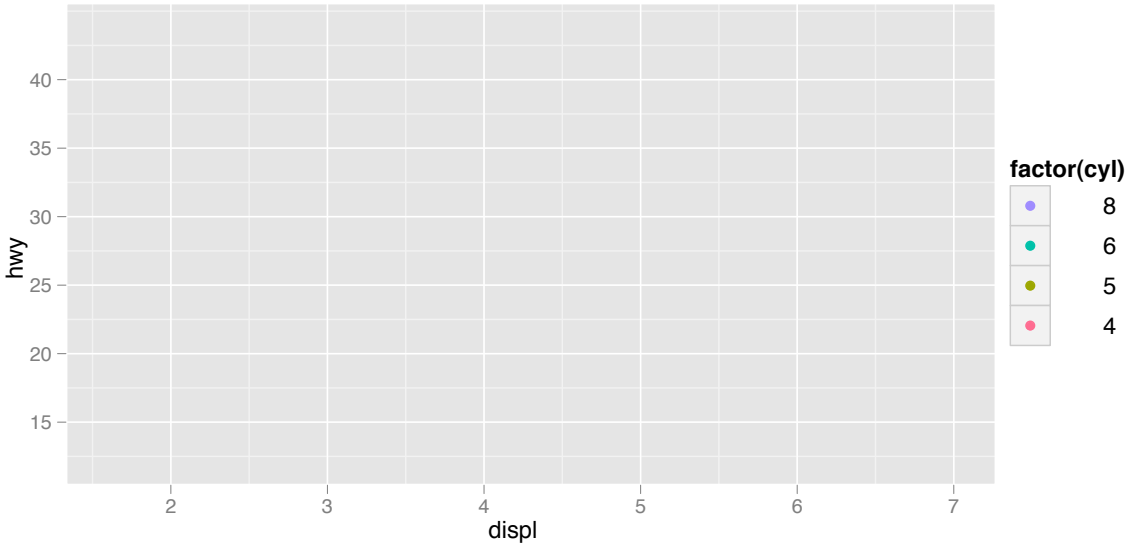


Figure 3.4: A colour wheel showing how the default colour scheme for discrete values is produced.

Figure 3.5: Contributions from the scales, the axes and legend and grid lines, and the plot background. Contributions from the data, the point geom, has been removed.



x	y	colour	size	shape
0.037	0.531	#FF6C91	1	20
0.037	0.531	#FF6C91	1	20
0.074	0.594	#FF6C91	1	20
0.074	0.562	#FF6C91	1	20
0.222	0.438	#00C1A9	1	20
0.222	0.438	#00C1A9	1	20
0.278	0.469	#00C1A9	1	20
0.037	0.438	#FF6C91	1	20
0.037	0.406	#FF6C91	1	20
0.074	0.500	#FF6C91	1	20

Table 3.4: Simple dataset with variables mapped into aesthetic space. Default values for other aesthetics are also included: the points will be filled circles (shape 20 in R) with a 1mm diameter.

3.4 A more complicated plot

With a simple example under our belts, it's now turn to look at the slightly more complicated plot in Figure 3.4. This plot adds three new components to the mix: facets, multiple layers and statistics. The facets and layers expand the data structure a little: each panel in each layer has its own dataset. Think of this like a 3d array of data frames. The smooth stat adds an addition step in the above process: after mapping the data to aesthetics, the relationship between the x and y variables is summarised with a flexible model. Other use statistical transformations include binning, for the histogram, and ...

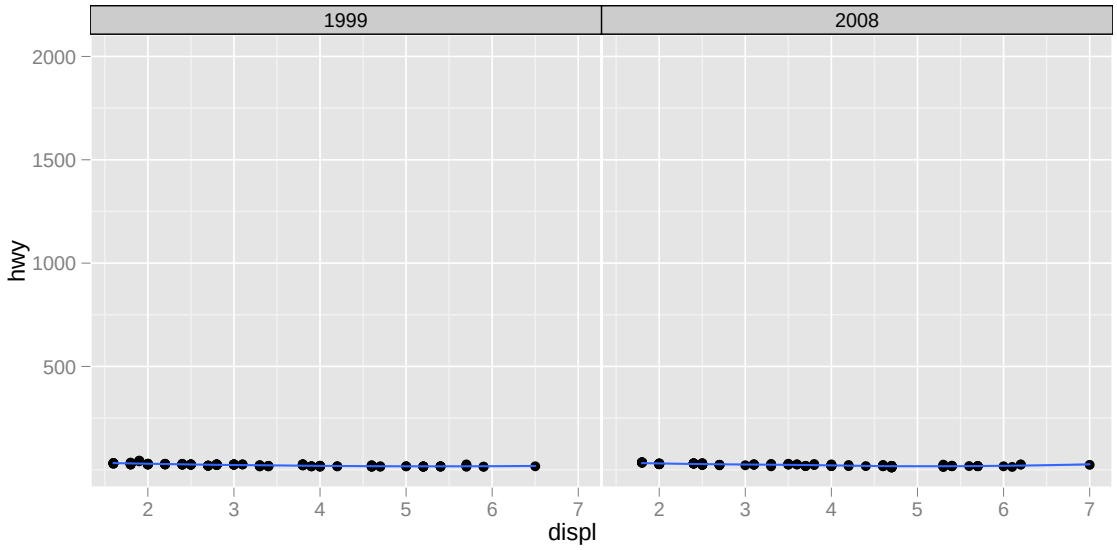


Figure 3.6: A more complex plot with facets and multiple layers.

Faceting splits the original dataset into a dataset for each subset, so the data that underlies Figure 3.4 looks like Table 3.5.

	x	y	*colour*
a	2	4	red
a	1	1	red
b	4	15	blue
b	9	80	blue

Table 3.5: Simple dataset faceted into subsets.

The first steps of plot creation proceed as before, but new steps are necessary when we get to the scales. Scaling actually occurs in three parts: transforming, training and mapping.

- 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. See Section **??** for more details. Transformation is only necessary for non-linear scales, because all statistics are location-scale invariant.

- After the statistics are computed, each scale is trained on every faceted dataset (a plot can contain multiple datasets, e.g. raw data and predictions from a model). The training operation combines the ranges of the individual datasets to get the range of the complete data. If scales were applied locally, comparisons would only be meaningful within a facet. This is shown in Table **??**.

- Finally the scales map the data values into aesthetic values. This gives Table **??** which is essentially identical to Table 3.2 apart from the structure of the datasets. Given that we end up with an essentially identical structure you might wonder why we don't simply split up the final result. There are several reasons for this. It makes writing statistical transformation functions easier, as they only need to operate on a single facet of data, and some need to operate on a single subset, for example, calculating a percentage. Also, in practice we may have a more complicated training scheme for the position scales so that different columns or rows can have different x and y scales.

3.5 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,
- statistical transformations,
- scales,

- and facet specification.

And have also touched on the coordinate system. Together, the data, mappings, statistical transformation and geometric object form a **layer**. A plot may have multiple layers, as in the example, where we overlaid a smoothed line on scatterplot. To be precise, the layered grammar defines the components of a plot as:

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

The layer component is particularly important as it determines the physical representation of the data, with the combination of stat and geom defining many familiar named graphics: the scatterplot, histogram, contourplot, and so. In practice, many plots have (at least) three layers: the data, context for the data, and a statistical summary of the data. For example, to visualise a spatial point process, we might display the points themselves, a map giving some context to the locations of points, and contours of a 2d density estimate.

This grammar is useful for both the user and the developer of statistical graphics. For the user, it makes it easier 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 us 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 generic named graphics.

For the developer, it makes it much easier to add new capabilities. You only need to add the one component that you need, and continue to use the all the other existing components. For example, you can add a new statistical transformation, and continue to use the existing scales and geoms. It is also useful for discovering new types of graphics, as the grammar effectively defines the parameter space of statistical graphics.

3.5.1 Layers

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

- data and aesthetic mapping,
- a statistical transformation (stat),
- a geometric object (geom)
- and a position adjustment.

These parts are described in detail below.

Usually all the layers on a plot have something in common, which is typically that they are different views of the same data, e.g. a scatterplot with overlaid smoother.

Data and mapping

Data is obviously a critical part of the plot, but it is important to remember that it is independent from the other components: we can construct a graphic that can be applied to multiple datasets. Data is what turns an abstract graphic into a concrete graphic.

Along with the data, we need a specification of which variables are mapped to which aesthetics. For example, we might map weight to x position, height to y position and age to size. The details of the mapping are described by the scales, Section 3.5.2. Choosing a good mapping is crucial for generating a useful graphic, as described in Section **??**.

Statistical transformation

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 mean of y, conditional on x, subject to some restriction that ensures smoothness. Table 3.6 lists some of the stats available in ggplot2. To make sense in a graphic context a stat must be location-scale invariant: $\mathrm{f}(x + a) = \mathrm{f}(x) + a$ and $\mathrm{f}(b \cdot x) = b \cdot \mathrm{f}(x)$. This ensures that the transformation is invariant under translation and scaling, which are common operations on a graphic.

A stat takes a dataset as input and returns a dataset 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, one way to describe a histogram is as a binning of a continuous variable, plotted with bars whose height is proportional to the number of points in each bin, as described in Section **??**. Another useful example is mapping the size of the lines in a contour plot to the height of the contour.

The actual statistical method used by a stat is conditional on the coordinate system. For example, a smoother in polar coordinates should use circular regression, and in 3d should return a 2d surface rather than a 1d curve. However, many statistical operations have not been derived for non-Cartesian coordinates and we so we generally fall back to Cartesian coordinates for calculation, which, while not strictly correct, will normally be a fairly close approximation.

Geometric object

Geometric objects, or **geom**s for short, control 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. We can classify geoms by their dimensionality:

- 0d: point, text
- 1d: path, line (ordered path)
- 2d: polygon, interval

Geometric objects are an abstract component and can be rendered in different ways. Figure 3.7 illustrates four possible renderings of the interval geom.

Geoms are mostly general purpose, but do require certain outputs from a statistic. For example, the boxplot geom requires the position of the upper and lower fences, upper and

Name	Description
bin	Divide continuous range into bins, and count number of points in each
boxplot	Compute statistics necessary for boxplot
contour	Calculate contour lines
density	Compute 1d density estimate
identity	Identity transformation, $f(x) = x$
jitter	Jitter values by adding small random value
qq	Calculate values for quantile-quantile plot
quantile	Quantile regression
smooth	Smoothed conditional mean of y given x
summary	Aggregate values of y for given x
sortx	Sort values in order of ascending x
unique	Remove duplicated observations

Table 3.6: Some statistical transformations provided by ggplot2. The user is able to supplement this list in a straight forward manner.

Figure 3.7: Four representations of an interval geom. From left to right: as a bar, as a line, as a error bar, and (for continuous x) as a ribbon.

lower hinges, the middle bar and the outliers. Any statistic used with the boxplot needs to provide these values.

Every geom has a default statistic, and every statistic a default geom. For example, the bin statistic defaults to using the bar geom to produce a histogram. Over-riding these defaults will still produce valid plots, but they may violate graphical conventions.

Each geom can only display certain aesthetics. For example, a point geom has position, colour, and size aesthetics. A bar geom has all those, plus height, width and fill colour. Different parameterisations may be useful. For example, instead of location and dimension, we could parameterise the bar with locations representing the four corners. Parameterisations which involve dimension (e.g. height and width) only make sense for Cartesian coordinate systems. For example, height of a bar geom in polar coordinates corresponds to radius of a segment. For this reason location based parameterisations are used internally.

Position adjustment

Sometimes we need to tweak the position of the geometric elements on the plot, when otherwise they would obscure each other. This is most common in bar plots, where we stack or dodge (place side-by-side) the bar to avoid overlaps. In scatterplots with few unique x and y values, we sometimes randomly jitter (Chambers et al., 1983) the points to reduce overplotting.

3.5.2 Scales

A **scale** controls the mapping from data to aesthetic attributes, and so we need one scale for each aesthetic property used in a layer. Scales are common across layers to ensure a consistent mapping from data to aesthetics. Some scales are illustrated in Figure 3.8.

Figure 3.8: Examples of four scales from ggplot2. From left to right: continuous variable mapped to size and colour, discrete variable mapped to shape and 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 (eg. LUV or RGB), and the start and end colours.

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.ethe 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.

Scales typically map from a single variable to a single aesthetic, but there are exceptions. For example, we can map one variable to hue and another to saturation, to create a single aesthetic, colour. We can also create redundant mappings, mapping the same variable to multiple aesthetics. This is particularly useful when producing a graphic that works in both colour and black and white.

3.5.3 Coordinate system

A coordinate system, **coord** for short, maps the position of objects onto the plane of the plot. Position is often specified by two coordinates (x, y), but could be any number of coordinates. The Cartesian coordinate system is the most common coordinate system for two dimensions, while polar coordinates and various map projections are used less frequently. For higher dimensions, we have parallel coordinates (a projective geometry), mosaic plots (a hierarchical coordinate system) and linear projections onto the plane.

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 **??**.

Coordinate systems control how the axes and grid lines are drawn. Figure 3.9 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.

3.5.4 Faceting

There is also another thing that turns out to be sufficiently useful that we should include it in our general framework: faceting (also known as conditioned or trellis plots). This

Figure 3.9: 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 correctly!

makes it easy to create small multiples of different subsets of an entire dataset. This is a powerful tool when investigating whether patterns hold across all conditions. The faceting specification describes which variables should be used to split up the data, and how they should be arranged in a grid.

3.6 Data structures

These principles are encoded as data structures in a fairly straightforward way.

There are two ways to create these plot objects: all at once with `qplot()`, as shown in the previous chapter, or piece-by-piece with `ggplot()` and layer functions, as described in the next chapter.

Regardless of how you make the plot, you will always end up with a plot object. This can be saved or displayed.

One thing to note is that all ggplot2 objects (with the exception of the main plot object) are proto objects. Proto is a package which implements the prototype-style of object-oriented programming. There are some major differences between this and the typical S3 or S4 style of OO in R, but the good news is that you only need to worry about them if you want to develop your own extensions to ggplot2. For everyday use, the proto objects are hidden behind a facade which makes them act like normal R objects.

`str` to see full structure (it can be large!)

`summary` briefly describes the structure of the plot

`recreate` to see (one way) to recreate the plot with code

Data stored inside the plot - if you change the data outside of the plot, and then redraw a saved plot, it will not be updated. Consequence of R copying semantics.

Chapter 4

Build a plot layer by layer

4.1 Introduction

Layering is the mechanism by which additional elements are added to a plot. Each added element can come from a different dataset and have a different aesthetic mapping, allowing us to create plots that could not be generated using `qplot()`, which permits only a single dataset and a single aesthetic mapping.

Section 4.2 tells you how to initialize a plot object. The plot object is not yet ready to be displayed until at least one layer is added, as as described in Section 4.3. Sections 4.4 and 4.5 describe the data and aesthetic mappings in more detail, including more information about how layer settings override the plot defaults, the difference between setting values and mapping aesthetics, and the important group aesthetic. Sections 4.6 and 4.7 list the geoms and stats available in `ggplot2`. Section 4.8 concludes by introducing you to some plotting techniques that take advantage of what you have learned in this chapter.

This chapter is mainly a technical description of how layers, geoms and statistics work - how you call and customise them. The next chapter, the `ggplot2` "toolbox", describes how you can use different geoms and stats to do data analysis. These two chapters are companions, with this one explaining how layers work and the next one how you can use layers to achieve your graphical goals.

4.2 Creating a plot

To build up a plot layer by layer, we start by creating a plot object. When discussing `qplot()`, we didn't note that it creates a plot object, but it does. It creates a plot object, adds layers, and shows the result, applying a lot of default values along the way. To initialise a plot object without any shortcuts, simply call `ggplot()` with no arguments; assign the result to a variable in order to add to it afterward. ggplot accepts two optional arguments: **data** and aesthetic **mapping**. The data argument needs little explanation: It's the data frame that you want to visualise. You are already familiar with aesthetic mappings from `qplot()`, and the syntax here is quite similar, although you need to wrap the pairs of aesthetic attribute and variable name in an `aes()`:

```
p <- ggplot(mtcars, aes(x = wt, y = mpg, colour = cyl))
```

These arguments set up defaults for the plot and can be omitted if you specify data and aesthetics when adding each layer.

This initial plot object can not be rendered until we add at least one layer.

4.3 Layers

A minimal layer may do nothing more than specify a **geom**, a way of visually representing the data. The plot object we just initialized can be rendered after this addition, specifying a scatterplot:

```
p <- p + layer(geom = ‘‘point’’)
```

```
layer(geom = "point", stat = "identity", position = "identity")
```

Note the use of + to add the layer to the plot. This layer uses the plot defaults for data and aesthetic mapping and it uses default values for two optional arguments: the statistical transformation (the stat) and the position adjustment. (Position adjustments are described in detail in Chapter **??**; in brief, they shift objects on the plot to avoid overplotting.) A more fully specified layer can take any or all of these arguments:

```
layer(geom, geom_params, stat, stat_params, data, mapping, position)
```

This more complicated layer calls for a histogram in “steelblue” and a bin width of 2:

```
p <- ggplot(mtcars, aes(x = mpg))
p <- p + layer(
  geom = ‘‘histogram’’,
  geom_params = list(fill = "steelblue"),
  stat = ‘‘bin’’,
  stat_params = list(binwidth = 2)
)
p
```

This layer specification is precise but verbose. We can simplify it by using shortcuts that rely on the fact that every geom is associated with a default statistic and position, and every statistic with a default geom. This means that you only need to specify one of `stat` or `geom` to get a completely specified layer, with parameters passed on to the geom or stat as appropriate. This expression generates the same layer as the full layer command above:

```
geom_histogram(binwidth = 2, fill = "steelblue")
```

All the shortcut functions have the same basic form, beginning with **geom_** or **stat_**:

```
geom_XXX(mapping, data, ..., geom, position)
stat_XXX(mapping, data, ..., stat, position)
```

Their common parameters define the components of the layer:

- **mapping** (optional): A set of aesthetic mappings, specified using the `aes()` function and combined with the plot defaults as described in Section 4.5.

- **data** (optional): A data set which overrides the default plot data set. It is most commonly omitted, in which case the layer will use the default plot data.
- **...**: Parameters for the geom or stat, such as bin width in the histogram or bandwidth for a loess smoother. Any aesthetic that the geom recognises can also be specified as a parameter to the layer. See Section 4.5.2.
- **geom** or **stat** (optional): You can override the default `stat` for a `geom`, or the default `geom` for a `stat`. This is a text string containing the name of the geom to use. Using the default will give you a standard plot; overriding the defaults allows you to achieve something more exotic, as described in Section 4.8.1.

Simple layers can be used with `qplot()`, without the option of changing the data or aesthetic mapping:

```
qplot(mtcars, aes(mpg, wt)) + geom_point()
qplot(mtcars, aes(mpg, wt, colour = factor(cyl))) + geom_smooth()
```

You've seen that plot objects can be stored as variables. The summary function can be helpful for inspecting the structure of a plot without plotting it:

```
> p <- ggplot(data=mtcars, aes(mpg, wt))
> summary(p)
Title:
-----------------------------------
Data:     mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb [32x11]
Mapping:  x=mpg, y=wt
Scales:   x,y -> x,y
Faceting: facet_grid(. ~ ., FALSE)
>
> p <- p + geom_point()
> summary(p)
Title:
-----------------------------------
Data:     mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb [32x11]
Mapping:  x=mpg, y=wt
Scales:   x,y -> x,y
Faceting: facet_grid(. ~ ., FALSE)
-----------------------------------
mapping:
geom_point:
stat_identity:
position_identity: ()
```

Layers can also be stored as variables, so that it is easy to write clean code that generates a family of related plots. For example, a set of plots can be initialised using different data then enhanced identically with carefully constructed layers.

Note that the order of `data` and `mapping` arguments is switched between `ggplot()` and the layer functions. This should improve ease of use, because you almost always specify data for the plot, and almost always specify aesthetics – but *not* data – for the layers. However, I suggest explicitly naming other arguments rather than relying on positional matching. This makes the code more readable and it is the style followed in this book.

The following sections describe the data and aesthetic mappings in more detail, then go on to describe the available geoms and stats.

4.4 Data

The restriction on the data is simple: It must be a data frame. It is not necessary to specify a default dataset except when using faceting; faceting is a global operation (i.e., it works on all layers) and it needs to have some a dataset to add in any missing columns. See Section **??** for more details. If the dataset is omitted, every layer must supply its own data.

It is important to remember that the data is stored in the plot object as a copy, not a reference. This is important because ggplot2 objects are entirely self-contained. You can save one to disk and later plot it without needing anything else from that session.

If you want to update (or change) the data later on, you can replace the default dataset with `%+%`:

```
p <- ggplot(mtcars, aes(mpg, wt, colour = cyl)) + geom_point()
p
mtcars <- transform(mtcars, mpg = mpg ^ 2)
p %+% mtcars
```

Any change of values or dimensions is legitimate as long as the variables used in the plot are still part of the data, and the variable do not change from discrete to continuous or vice versa. This facility can be useful if you need to produce the same plot for different datasets. It's also an easy way to experiment with influential points or imputation schemes.

4.5 Aesthetic mapping

To describe the way that variables in the data are mapped to things that we can perceive on the plot (the "aesthetics"), we use the `aes` function. The `aes` function takes a list of aesthetic-variable pairs like these:

```
aes(x = weight, y = height, colour = age)
```

Here we are mapping x-position to weight, y-position to height and colour to age. The first two arguments can be left without names, in which case they are assumed to correspond to the x and y variables. (This matches the way that `qplot()` is normally used.)

```
aes(weight, height, colour = sqrt(age))
```

Note that functions of variables can be used.

Any variable in an `aes()` specification must be contained inside the plot or layer data. This is one of the ways in which ggplot2 objects are guaranteed to be entirely self-contained, so that they can be stored and re-used.

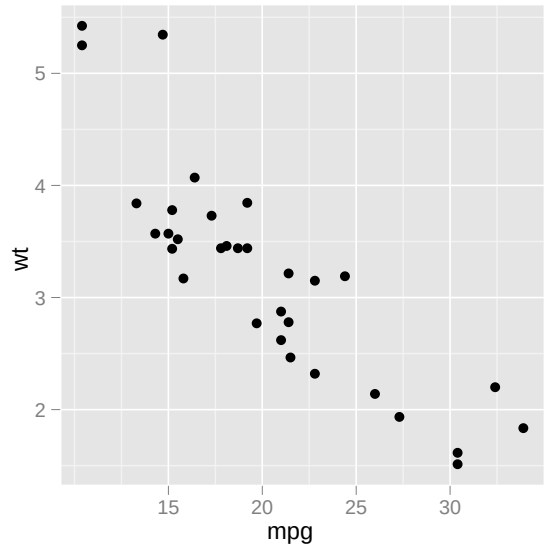
4.5.1 Plots and layers

When the aesthetic mappings are part of the plot defaults, they can be set when the plot is initialized or added later using +, as in this example:

```
> p <- ggplot(mtcars)
> summary(p)
Title:
-----------------------------------
Data:     mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb [32x11]
Faceting: facet_grid(. ~ ., FALSE)
>
> p <- p + aes(wt, hp)
> summary(p)
Title:
-----------------------------------
Data:     mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear, carb [32x11]
Mapping:  x=wt, y=hp
Faceting: facet_grid(. ~ ., FALSE)
```

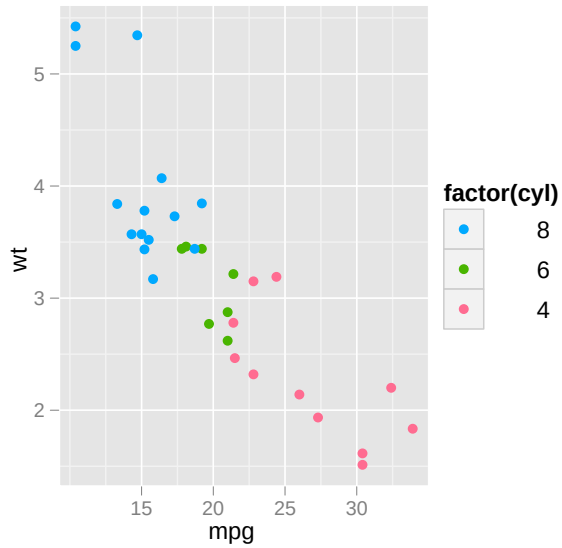
We have seen several examples of using the default mapping when adding a layer to a plot:

```
> p <- ggplot(mtcars, aes(x = mpg, y = wt))
> p + geom_point()
```

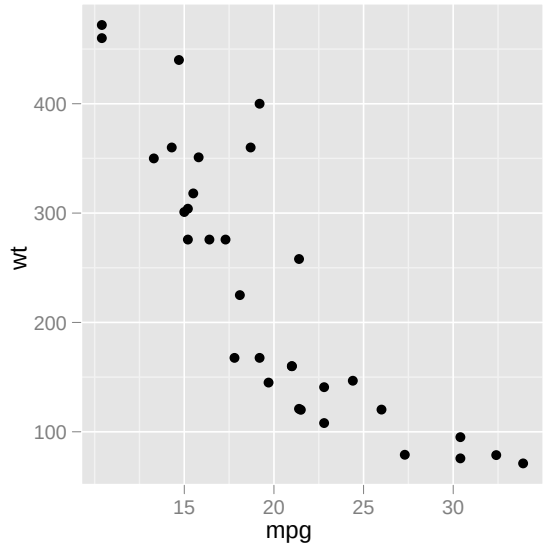


As these two examples show, the default mappings in the plot p just defined can be extended or overridden in added layers:

```
> p + geom_point(aes(colour = factor(cyl)))
```




```
> p + geom_point(aes(y = disp))
```



The rules are summarised in Table 4.1. Note that you are overriding the aesthetics only in that layer. Unless you specify otherwise, the axis and legend names, as well as mappings in subsequent layers, will use the default assignments.

4.5.2 Setting vs. mapping

For every aesthetic the geom function understands, you can also set that aesthetic as an parameter to the function. Aesthetics can vary for each observation being plotted, while parameters can not. For example, the following layer sets a parameter but not an aesthetic mapping:

```
p <- ggplot(mtcars, aes(x=mpg, y=wt))
```

Operation	Layer aesthetics	Result
Add	colour = cyl	x = mpg, y = wt, colour = cyl
Override	y = disp	x = mpg, y = disp
Delete	y = NULL	x = mpg

Table 4.1: Rules for combining layer aesthetic mapping with default mapping of `aes(x = mpg, y= wt)`: additional aesthetics can be added, overridden, and removed.

```
p + geom_point(colour="darkblue")
```

will set the point colour to be dark blue instead of black. This is quite different to:

```
p + geom_point(aes(colour="darkblue"))
```

This **maps** (not sets) the colour to the value "darkblue". This effectively creates a new variable containing only the value "darkblue" and then maps colour to that new variable. 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 reddish.

When the string refers to a variable, it does what we've already described. In this case, it doesn't correspond to a variable in the dataset but a constant, so the color scale uses its default highlighting color. For the default color scale, that's the pinkish color you see in figure blah.

Chapter A describes how values should be specified for the various aesthetics. With qplot, you can do the same thing by putting the value inside of `I()`, e.g., `colour = I("darkblue")`. The difference between setting and mapping is illustrated in Figure 4.5.2.

4.5.3 Grouping

The `group` aesthetic partitions the the data set into discrete components. It is designed to be used with data comprised of multiple groups of linked records, especially longitudinal data; for example, a medical experiment in which measurements were taken on each subject at more than one time point. The `group` is used by the line geom to determine which observations to connect, by the boxplot geom to determine which points to summarise in each box, and by the smooth geom to determine which group of points should be included in the smooth. When one or more discrete variables are used in the plot, the default group aesthetic is their combination (formed by `interaction()`); this often partitions the data correctly. When it does not, or when no discrete variable is used in the plot, the group needs to be explicitly defined. There are three common cases where this occurs, and we will consider each one separately in the following examples.

In these examples, we will use a simple longitudinal data set, `Oxboys` from the `nlme` package. It records the heights (height) and ages (age) of 26 boys (Subject), measured on nine occasions (Occasion).

Multiple groups, one aesthetic: In many situations, you want to separate your data into groups but render them with the same aesthetic. This is common in longitudinal

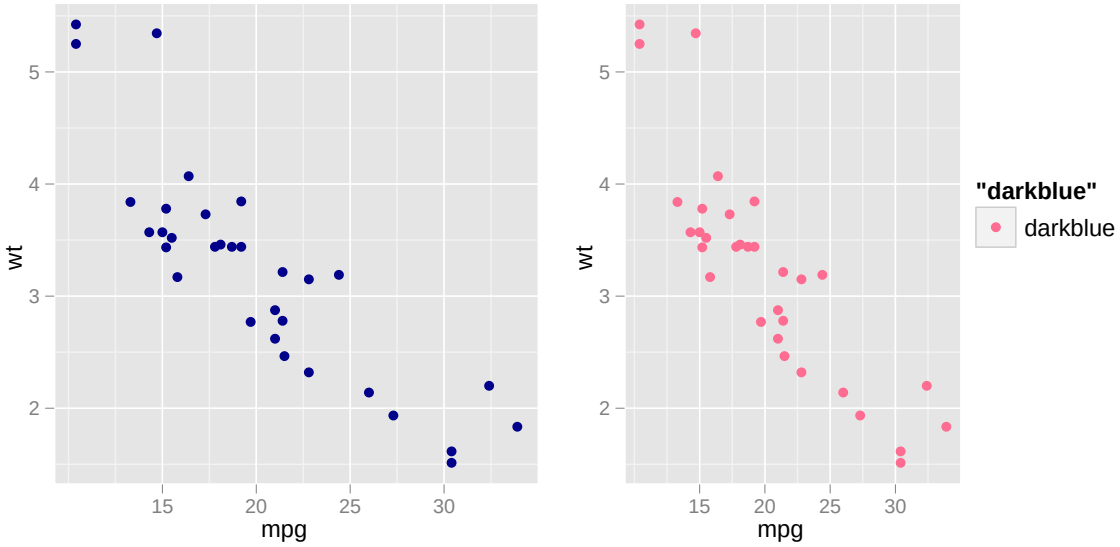


Figure 4.1: The difference between *Left,* setting colour to `"darkblue"` and *Right,* mapping colour to `"darkblue"`. When `"darkblue"` is mapped to colour, it is treated as a regular value and scaled with the default colour scale. This results in pinkish points and a legend.

studies with many subjects. When looking at the data in aggregate you want to be able to distinguish individual subjects, but you don't need to identify them.

The first plot in Figure 4.5.3 shows a set of time series plots, one for each boy. It was generated generated as follows:

```
p <- ggplot(Oxboys, aes(x=age, y=height)) +
  geom_line(aes(group=Subject))
p
```

We specified the Subject as the grouping variable to create a partition for each boy. The second plot in Figure 4.5.3 shows the result of omitting the group; plots with an incorrect group aesthetic often look something like this.

Different groupings in the same plot: Sometimes we want to plot summaries based on different levels of aggregation of the data. Here, different layers might have different grouping aesthetics, so that some display the full data while others display summaries of larger groups.

Building on the current example, suppose we want to add a single smooth line to the plot just created, based on the ages and heights of *all* the boys. If we use the same grouping for the smooth that we used for the line, we get the first plot in Figure 4.5.3.

```
p + geom_smooth(aes(group=Subject), method="lm")
```

This is not what we wanted; we have inadvertently added a smoothed line for each boy. This new layer needs a different group aesthetic, so that the new smoothed line will be based on all the data, as shown in the second plot in the figure:

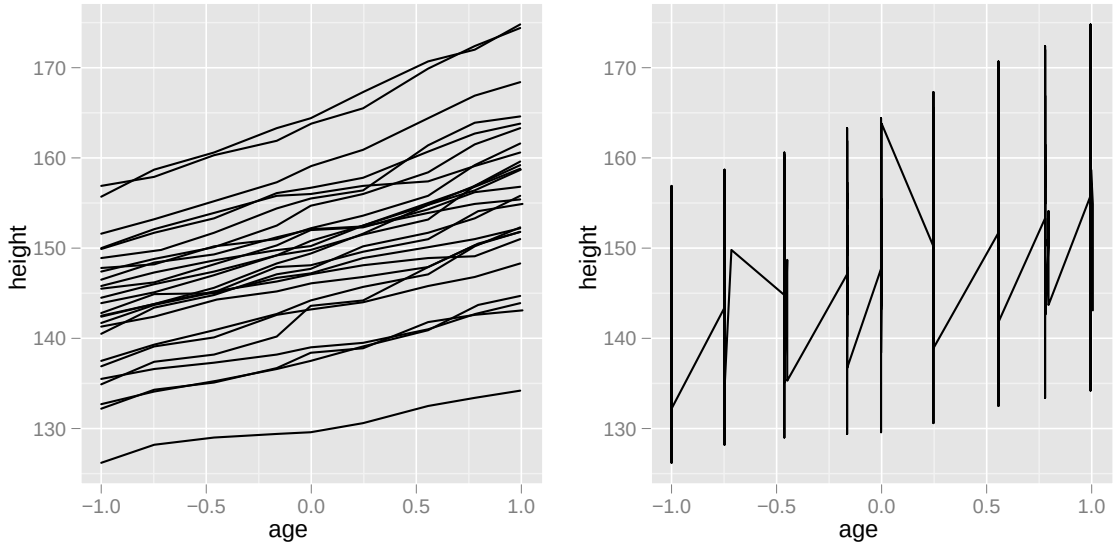


Figure 4.2: *Left,* Correctly specifying `group = Subject` produces one line per subject. *Right,* This pattern is characteristic of an incorrect grouping aesthetic.

```
p + geom_smooth(aes(group=1), method="lm")
```

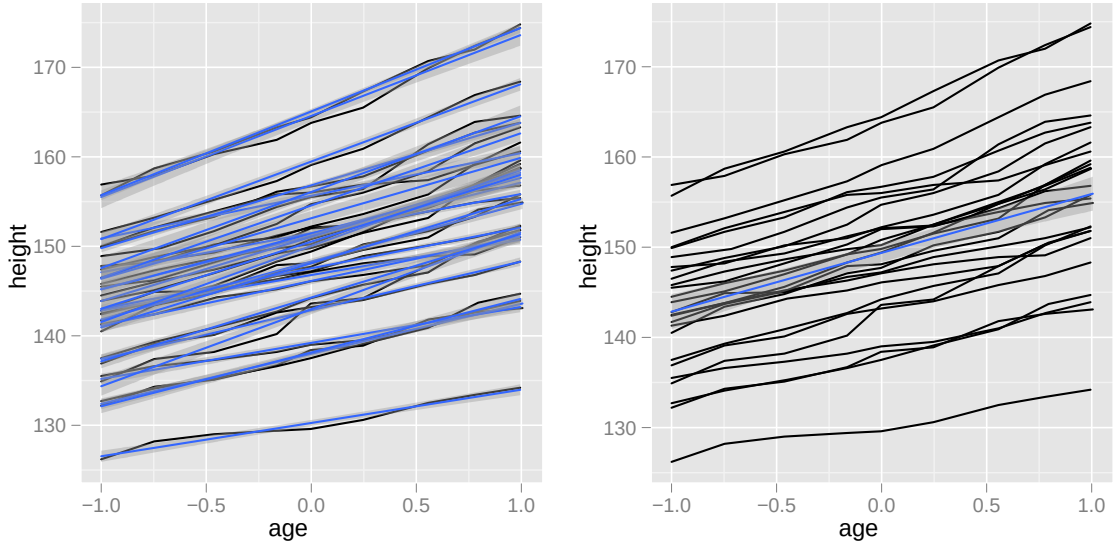


Figure 4.3: Adding smooths to the Oxboys data. *Left,* Using the same grouping as the lines results in a line of best fit for each boy. *Right,* Using `aes(group = 1)` in the smooth layer fits a single line of best fit across all boys.

Note because the first plot was stored in the variable p, we could experiment with the

code to generate the added layer without having to re-enter any of the code for the first layer.

Using the default group aesthetic: In this case, the plot has a discrete scale but you want to draw lines that connect *across* groups. This is the strategy used in interaction plots, profile plots, and parallel coordinate plots, among others.

We start with boxplots of height at each occasion of measurement, as shown in the first figure in Figure 4.5.3, created this way:

```
ggplot(Oxboys, aes(x=Occasion, y=height)) + geom_boxplot()
```

There is no need to specify the group aesthetic here; the default grouping works here because occasion is a discrete variable. To overlay individual trajectories we again need to override the default grouping for that layer with `aes(group = Subject)`, as shown in the second plot in the figure.

```
p <- ggplot(Oxboys, aes(x=Occasion, y=height)) +
 geom_boxplot()
p + geom_line(aes(group=Subject), colour="#3366FF")
```

There's another difference in the second plot. The boxplots are drawn in thick blue lines so that they will still be visible once the line segments for each subject have been added. Note that the color and size in the boxplots in the second plots are settings rather than mappings, as discussed earlier in this chapter. They are rendering attributes, but they have no correspondance to variables in the data.

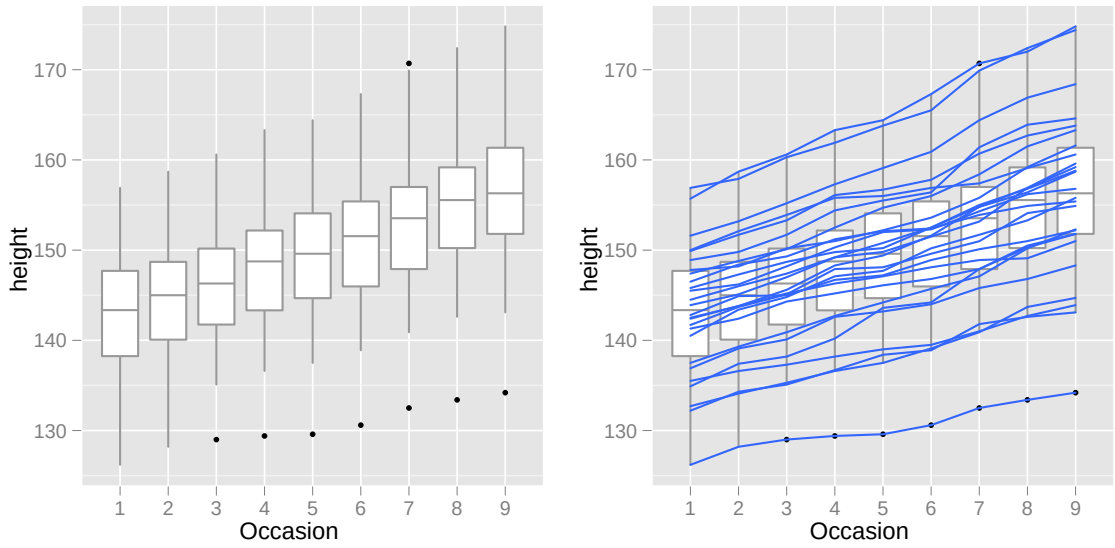


Figure 4.4: *Left,* if boxplots are used to look at the distribution of heights at each occasion (a discrete variable), the default grouping works correctly. *Right,* if trajectories of individual boys are overlaid with `geom_line()` then `aes(group = Subject)` must be set for the new layer.

The `interaction()` function is particularly useful if there isn't a pre-existing variable that separates the groups you are interested in, but a combination of variables does.

4.5.4 Matching aesthetics to graphic objects

In ggplot2, geoms can be roughly divided into individual and collective geoms. An individual geom has a distinctive graphical object for each row in the data frame. For example, the point geom has a single point for each . On the other, collective geoms represent multiple observations. This maybe a result of a statistical summary, or may be fundamental to the display of the geom, as with polygons. Lines and paths fall somewhere in between - each overall line is composed of a set of straight segments, but each segment represents two points (compare this with `geom_segment` where each segment represents a single row). What happens when the aesthetic attributes of the points are different?

For individual geoms, this isn't a problem at the drawing level, but still may be a problem at the perceptual level - overplotting means that you can't distinguish between individual points and in some sense the point geom becomes a collective geom.

For lines and paths, operates on an off by one principle. Blending too complicated in general (how would you blend from green dotted lines to red dashed lines?). Rules for colour and line type (due to technical limitations in R).

For all other collective geoms, all aesthetic attributes must be the same, or the default value will be used. This makes sense for fill as it is a property of the entire object - it doesn't make sense to think about having a different fill for each point on the border.

There is of course also an interaction with grouping. This results in an intuitive display for bars and area plots because stacking the individual bars or areas results in the same shape as the original data.

4.6 Geoms

Geoms, or geometric elements, perform the actual rendering of the plot. Geoms are also responsible for drawing legends, as explained in Section **??**. Table 4.2 lists all of the geoms available in ggplot.

4.7 Stat

Statistical transformations, stats, summarise the data in some way. All available stats are listed in Table 4.3.

Each stat generates a number of output variables that can be used in aesthetic mappings. 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 data set. 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

Name	Description
abline	Line, specified by slope and intercept
area	Area plots
bar	Bars, rectangles with bases on y-axis
blank	Blank, draws nothing
boxplot	Box and whiskers plot
contour	Display contours of a 3d surface in 2d
crossbar	Hollow bar with middle indicated by horizontal line
density	Display a smooth density estimate
density_2d	Contours from a 2d density estimate
errorbar	Error bars
histogram	Histogram
hline	Line, horizontal
interval	Base for all interval (range) geoms
jitter	Points, jittered to reduce overplotting
line	Connect observations, in ordered by x value
linerange	An interval represented by a vertical line
path	Connect observations, in original order
point	Points, as for a scatterplot
pointrange	An interval represented by a vertical line, with a point in the middle
polygon	Polygon, a filled path
quantile	Add quantile lines from a quantile regression
ribbon	Ribbons, y range with continuous x values
rug	Marginal rug plots
segment	Single line segments
smooth	Add a smoothed condition mean.
step	Connect observations by stairs
text	Textual annotations
tile	Tile plot as densely as possible, assuming that every tile is the same size.
vline	Line, vertical

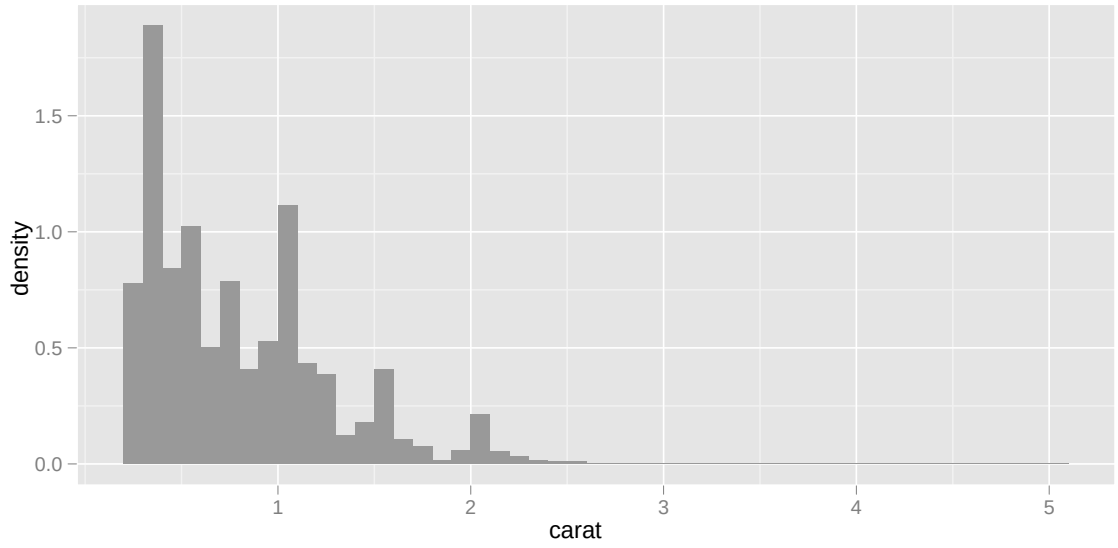
Table 4.2: Geoms in `ggplot2`

Name	Description
bin	Bin data
boxplot	Calculate components of box and whisker plot
contour	Contours of 3d data
density	Density estimation, 1D
density_2d	Density estimation, 2D
function	Superimpose a function
identity	Don't transform data
qq	Calculation for quantile-quantile plot
quantile	Continuous quantiles
smooth	Add a smoother
spoke	Convert angle and radius to xend and yend
step	Create stair steps
sum	Sum unique values. Useful for overplotting on scatterplots
summary	Summarise y values at every unique x
unique	Remove duplicates

Table 4.3: Stats in `ggplot2`

use the density (density). The following example shows a density histogram of carat from the diamonds dataset.

```
> ggplot(diamonds, aes(x=carat)) + geom_histogram(aes(y=..density..), binwidth=.1)
```



The names of generated variables must be surrounded with `..` when used. This prevents confusion in case the original data set 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.

As a reminder, the same plot can be produced using qplot because it does not include any layer which changes the default aesthetic mappings:

```
qplot(carat, ..density.., data = diamonds, geom="histogram", binwidth = .1)
```

4.8 Pulling it all together

Once you have become comfortable with combining layers, you will be able to create graphics that are both intricate and useful. The following examples demonstrate some of the ways to use the capabilities of layers that have been introduced in this chapter. These are just to get you started – you are limited only by your imagination!

4.8.1 Combining geoms and stats

By connecting geoms with different statistics, you can easily create new graphics. Figure 4.8.1 shows three variations on a histogram. They all use the same statistical transformation underlying a histogram (the bin stat), but use different geoms to display the results: the area geom, the point geom and the tile geom.

```
d <- ggplot(diamonds, aes(x=carat)) + xlim(0, 3)
d + stat_bin(aes(ymax = ..count..), binwidth = 0.1, geom = "area")
d + stat_bin(
  aes(size = ..density..), binwidth = 0.1,
  geom = "point", position="identity"
)
d + stat_bin(
  aes(y = 1, fill = ..count..), binwidth = 0.1,
  geom = "tile", position="identity"
)
```

(The use of xlim in ggplot will be discussed in 5, in the presentation of the use of scales and axes, but any R user can already guess that it is used here to fix the limits of the horizontal axis.)

A number of the geoms available in ggplot were derived from other geoms in a process like the one just described, by starting with an existing geom and making a few changes in the default aesthetics or stat. For example, the jitter geom is simply the point geom with the default position reset from NULL to jitter. Once it becomes clear that a particular variant is going to be used a lot or used in a very different context, it makes sense to create a new geom. Table 4.4 lists these "aliased" geoms.

4.8.2 Varying aesthetics and data

One of the more powerful capabilities of `ggplot2` is the ability to plot different data sets on different layers. This may seem strange: Why would you want to plot different data on

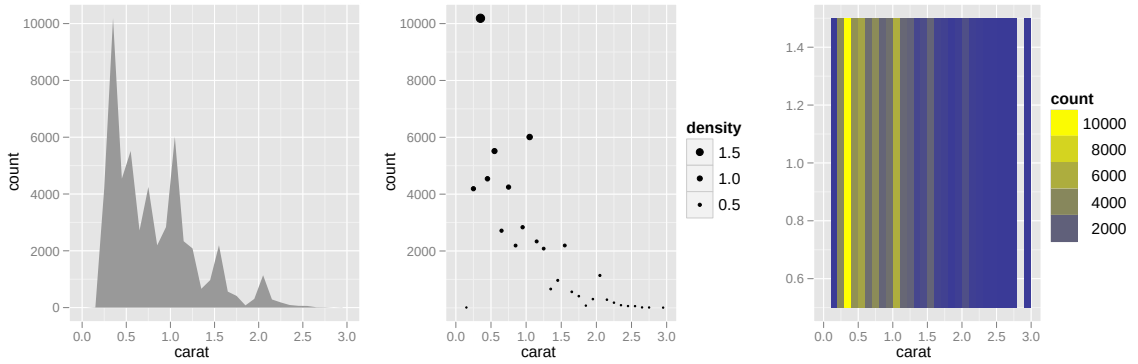


Figure 4.5: Three variations on the histogram. *Left,* a frequency polygon; *middle,* a scatterplot with both size and height mapped to frequency; *right,* an heatmap representing frequency with colour.

Aliased geom	Base geom	Changes in default
area	ribbon	`aes(min = 0, max = y), position = "stack"`
density	area	`stat = "density"`
histogram	bar	`stat = "bin"`
jitter	point	`position = "jitter"`
quantile	line	`stat = "quantile"`
smooth	ribbon	`stat = "smooth"`

Table 4.4: Geoms that were created by modifying the defaults of another geom.

the same plot? In practice, you often have related data sets that should be shown together. A very common example is supplementing the data with predictions from a model. While the smooth geom can add a wide range of different smooths to your plot, it is no substitute for a external quantitative model that summarises your understanding of the data.

Let's look again at the `Oxboys` dataset which used in Section 4.5.3. In Figure 4.5.3, we showed linear fits for individual boys (left) and for the whole group (right). Neither model is particularly appropriate: The group model ignores the within–subject correlation and the individual model doesn't use information about the typical growth pattern to more accurately predict individuals. In practice we might use a linear mixed effects model to do better. This section explores how we can combine the output from this more sophisticated model with the original data to gain more insight into both the data and the model.

First we'll load the `nlme` package, and fit a model with varying intercepts and slopes. (Exploring the fit of individual models shows that this is a reasonable first pass.) We'll also create a plot to use as a template. This regenerates the first plot in Figure 4.5.2, but we're not going to render it in its default state.

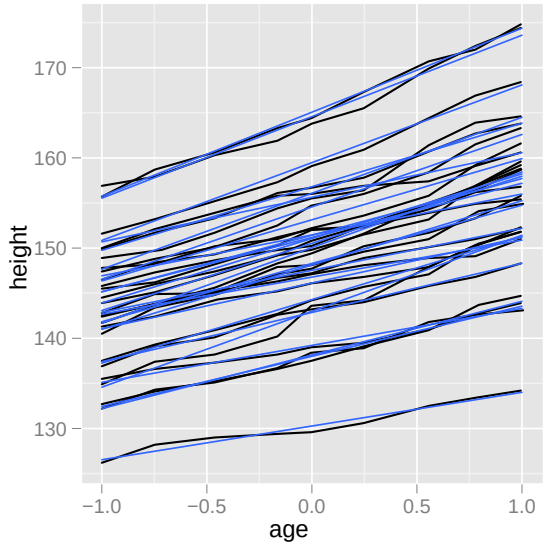
```
> require(nlme, quiet = TRUE, warn.conflicts = FALSE)
> model <- lme(height ~ age, data = Oxboys, random = ~ 1 + age | Subject)
> oplot <- ggplot(data=Oxboys, aes(x=age, y=height, group=Subject)) +
+   geom_line()
```

Next we'll compare the predicted trajectories to the actual trajectories. We do this by building up a grid that contains all combinations of ages and subjects. This is overkill for this simple linear case, where we only need two values of age to draw the predicted straight line, but we show it here because it is necessary when the model is more complex. Next we add the predictions from the model back into this dataset, as a variable called height.

```
> age_grid <- seq(-1, 1, length = 10)
> subjects <- unique(Oxboys$Subject)
>
> preds <- expand.grid(age = age_grid, Subject = subjects)
> preds$height <- predict(model, preds)
```

Once we have the predictions we can display them along with the original data. Because we have used the same variable names as the original Oxboys dataset, and we want the same group aesthetic, we don't need to specify any aesthetics; we only need to override the default dataset. We also set two aesthetic parameters to make it a bit easier to compare the predictions to the actual values.

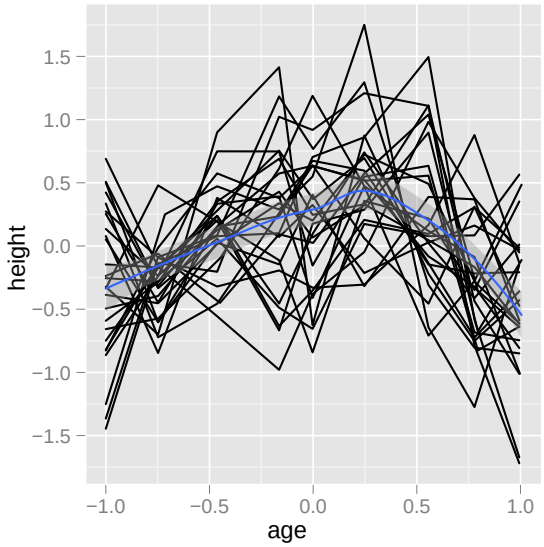
```
> oplot + geom_line(data = preds, colour = "#3366FF", size= 0.4)
```



It seems that the model does a good job of capturing the high-level structure of the data, but it's hard to see the details – plots of longitudinal data are often called spaghetti plots, and with good reason. Another way to compare the model to the data is to look at residuals, so let's do that. We add the predictions from the model to the original data (fitted), calculate residuals (resid), and add the residuals as well. The next plot is a little more complicated: We update the plot dataset (recall the use of %+% to update the default data), change the default y aesthetic to resid, and add a smooth line for all observations.

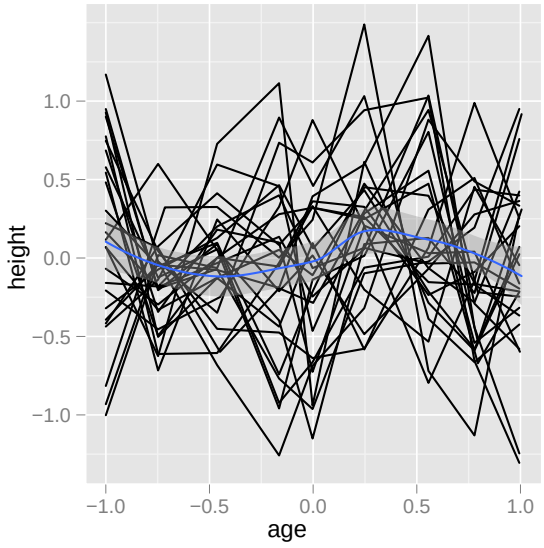
```
> Oxboys$fitted <- predict(model)
> Oxboys$resid <- with(Oxboys, fitted - height)
```

```
>
> oplot %+% Oxboys + aes(y = resid) + geom_smooth(aes(group=1))
```



The smooth line makes it evident that the residuals are not random, showing a deficiency in the model. We add a quadratic term, refit the model, recalculate predictions and residuals, and replot. There now much less evidence of model inadequacy.

```
> model2 <- update(model, height ~ age + I(age ^ 2))
> Oxboys$fitted2 <- predict(model2)
> Oxboys$resid2 <- with(Oxboys, fitted2 - height)
>
> oplot %+% Oxboys + aes(y = resid2) + geom_smooth(aes(group=1))
```



Notice how easily we were able to modify the plot object. We updated the data and

replotted twice without needing to reinitialize oplot. Layering in ggplot is designed to work well with the iterative nature of fitting and evaluating models.

Chapter 5

Scales, axes and legends

5.1 Introduction

Scales control the mapping from data to aesthetics. Each scale is a function from a region in data space (the domain) to a region in aesthetic space (the range). The domain, as we already know, can be continuous or discrete, ordered or unordered. The range consists of concrete aesthetics that you can perceive and that R can understand, such as position, colour, shape, size, and line type.

If you blinked when you read that scales map data both to position and colour, you are not alone. The notion that the same kind of object is used to map data to positions and symbols strikes some people as unintuitive. However, you will see the logic and power of this notion as you read further in the chapter.

For each scale there is also a **guide** which allows the viewer to perform the inverse mapping, from aesthetic space to data space. For position aesthetics, the axes are the guides; for all other aesthetics, the legends do that job.

When a scale (and guide) are needed, ggplot automatically adds them using default values, so you can generate quite a lot of plots without knowing how scales work. However, understanding scales and learning how to manipulate them gives you much more control over your plots.

This chapter covers:

- How scales work, § 5.2.
- Using scales: their names, arguments and roles, § 5.3.
- More details about individual scales, § 5.4.
- Controlling the appearance of axes and legends, § 5.5.

5.2 How scales work

To describe how scales work, we will first describe the domain (the data space) and the range (the aesthetic space), and then outline the process by which one is mapped to the other.

Since an input variable is either discrete or continuous, the domain is either a set of values (stored as a factor, character vector, or logical vector) or an interval on the real

line (stored as a numeric vector of length 2). For example, in the mammals sleep dataset, the domain of the discrete variable vore is {carni, herbi, omni}, and the domain of the continuous variable bodywt is $[0.005, 6654]$.

The range can also be discrete or continuous. For discrete scales, it is a vector of aesthetic values corresponding to the input values. For continuous scales, it is a 1d path through some more complicated space. For example, the continuous colour scales have a range which is a path through colour space.

The range is either specified by the user when the scale is created, or by the scale itself. The process of mapping the domain to the range often includes the following stages:

- **transformation**: For continuous variables, it is often useful to display a transformation of the data, such as a logarithm or square root. This ensures that a plot of $log(x)$ vs $log(y)$ on linear scales looks the same as x vs y on log scales. Transformations are described in more depth in Section **??**.

  After any transformations have been applied, the statistical summaries for each layer are computed based on the transformed data.

- **training**: During this key stage, the domain is learned. Sometimes learning the domain of a scale is extremely straightforward: In a plot with only one layer, representing only raw data, it may consist simply of determining the minimum and maximum values of a continuous variable (after transformation), or listing the unique levels of a categorical variable. However, sometimes the domain must reflect multiple layers. For example, imagine a scale that will be used to create an axis; the minimum and maxiumum values of the raw data in the first layer and the statistical summary in the second layer are likely to be different, but they must all eventually be drawn on the same plot.

  The domain can also be specified directly, overriding the training process, by manually setting the domain of the scale with the `limits` argument, as described in Section 5.3. Any data values outside of the domain of the scale will be set to `NA`.

- **mapping:** The global domain has now been determined, and we already knew the range before we started this process. The last thing to do, then, is to apply the scaling function that maps data values to aesthetic values. Nothing needs to be done for some scales: For example, for continuous position scales, all the difficult work has already been done by the transformation step.

We have left a few stages out of this description of the process for simplicity. For example, we haven't discussed the role facetting plays in training, and we have also ignored position adjustments.

5.3 Constructing and using scales

Every aesthetic has a default scale that is added to the plot whenever you use that aesthetic in a layer. These are listed in Table 5.1.

To add a different scale or to modify some features of the default scale, create a scale object and add it to a plot (using `+`). All scale constructors have a common naming

Aesthetic	Discrete	Continuous
Colour and fill	hue	gradient
Position	discrete	continuous
Shape	shape	—
Line type	linetype	—
Size	discrete	size

Table 5.1: Default scales, by aesthetic and variable type. The default scale varies depending on whether the variable is continuous or discrete. Shape and line type do not not have a default continuous scale; size does not have a default discrete scale.

scheme. They start with `scale_`, followed by the name of the aesthetic (e.g., `colour_`, `shape_`, or `x_`), and finally by the name of the scale (e.g., `gradient`, `hue`, or `discrete`). For example, the name of the default scale for the colour aesthetic based on discrete data is `scale_colour_hue()`, and the name of the Brewer colour scale for filled points is `scale_fill_brewer()`.

1

The following example shows the difference between the default discrete and continuous scales for colour, as well as how to override the default scale; this is the code used to generate the plots:

```
p <- qplot(sleep_total, sleep_cycle, data=msleep, colour=vore)
p
p + scale_colour_discrete("What does\nit eat?",
   breaks = rev(c("herbi", "carni", "omni", NA)),
   labels = rev(c("plants", "meat", "both", "don't know")))
p + scale_colour_brewer(pal="Set1")
```

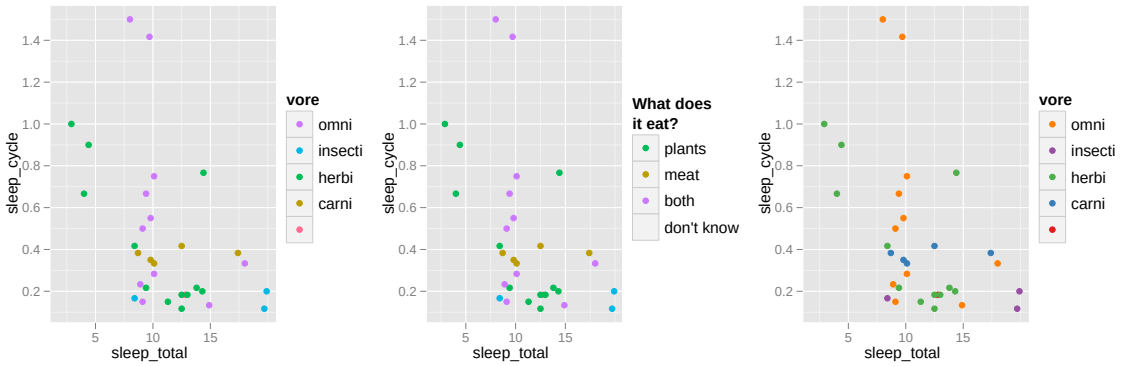


Figure 5.1: Differences between default discrete and continuous colour scales

The default colour scale for discrete values uses equally spaced hues, the default scale for continuous values uses a gradient of colours between blue and yellow, and there are

a number of Brewer colour scales designed to work well in a variety of situations (see http://colorbrewer.org for more detail).

As well as referring to scales explicitly by name, you can also refer to the default scales with the shorthand scale name `discrete` or `continuous`. For example, the default discrete colour scale is `scale_colour_discrete`. You can change the default scales with `set_default_scale(aesthetic, variable_type, scale_name, ...)`. Extra arguments are passed to the scale constructor. For example, if you wanted to set up default black and white colour scales you could execute:

```
set_default_scale("colour", "discrete", "grey")
set_default_scale("colour", "continous", "gradient",
 low = "white", high = "black"
)
```

As well as having a common naming scheme, all scales share a set of common arguments. These arguments control the basic operation of the scale and are described below.

- **name**: sets the label which will appear on the axis or legend. You can supply text strings (using "\n" for line breaks) or mathematical expressions (as described by `?plotmath`). The plots in the following figure are produced using this code:

  ```
  p <- qplot(tip, total_bill, data=tips, colour=tip/total_bill)
  p + scale_colour_gradient("Tip rate")
  p + scale_colour_gradient("The amount of the tip\ndivided by the total bill")
  p + scale_colour_gradient(expression(frac(tip, total_bill)))
  ```

  Because `scale_colour_gradient()` is the default scale for the colour aesthetic and continuous data, the only variation we see in the the plots is the different labels on the legends – and the resulting change in the plot's aspect ratio, in the case of the middle plot.

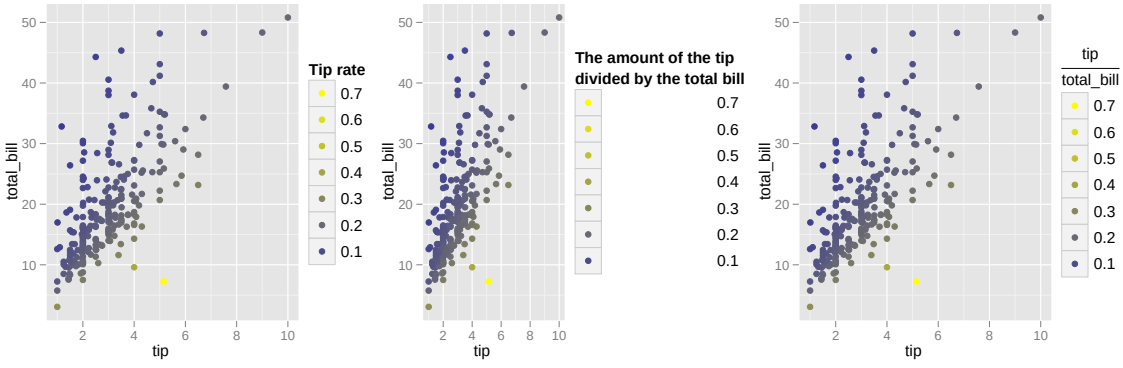


Figure 5.2: Legends with names given by (from left to right): `"Tip rate"`, `"The amount of the tip\ndivided by the total bill"` and `expression(frac(tip, total_bill)`

- **limits**: fixes the domain of the scale. Continuous scales take a numeric vector of length two; discrete scales take a character vector. If limits are set, no training of the data will be performed.

  There are shortcut functions for setting the limits of continuous x and y scales: `xlim()` and `ylim()`. Each has two arguments specifying the endpoints of the new domain, and each creates a scale object.

  This is particularly useful for zooming (i.e., setting limits that are smaller than the full range of data), and for ensuring that limits are consistent across multiple plots intended to be compared (i.e., setting limits that are larger or smaller than some of the default ranges).

  Any value not in the domain of the scale is not displayed; i.e., for an observation to be displayed it must be in the domain of every scale on the plot.

- **breaks** and **labels**: **breaks** controls which values appear on the axis or legend – e.g., at what values tick marks should appear on an axis or how a continuous scale is segmented in a legend. bf labels specifies the values that should appear at the breakpoints. (If `labels` is set, you must also specify `breaks`, so that the two can be matched up correctly.)

  To distinguish breaks from limits, remember that breaks affect what appears on the guides, while limits affect what appears on the plot. See by Figure 5.3 for an illustration. The first plot uses the default settings for both breaks and limits, which happen to be limits = c(4, 8) and breaks = 4:8. In the second plot, the breaks have been reset: The plotted region is the same, but the tick positions and labels have shifted. In the third plot, it is the limits which have been redefined, so much of the data now falls outside the plotting region.

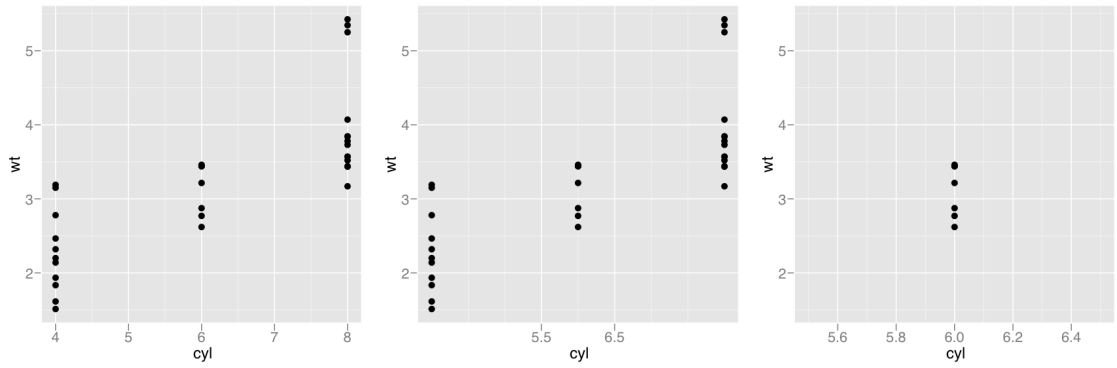


Figure 5.3: The difference between breaks and limits. (Left) default plot `limits = c(4, 8), breaks = 4:8`. (Middle) `breaks = c(5.5,6.5)` and (right) `limits = c(5.5,6.5)`.

This code generates the plots in the figure:

```
p <- qplot(cyl, wt, data=mtcars)
```

```
p
p + scale_x_continuous(breaks=c(5.5, 6.5))
p + scale_x_continuous(limits=c(5.5, 6.5))
```

5.4 More details

Scales can be divided roughly into four separate groups:

- Continuous position scales, used to map continuous variables onto the plotting region and to construct the corresponding axes.
- Colour gradients, used to map continuous variables to points or sub-regions in the plotting area and to construct the corresponding legends. For example, colour gradients are used in rendering density surfaces.
- Discrete scales, used to map discrete variables to symbol size, shape or colour, and to create the corresponding legend.
- The identity scale, used to plot variable values directly to the aesthetic rather than mapping them. For example, if the variable you want to map to symbol colour is itself a vector of colours, you want to plot those values rather than mapping them to some other colour scale.

This section describes each type in more detail. Precise details about individual scales can be found in the documentation, which can be used either within R (e.g. `?scale_brewer`), or online at http://had.co.nz/ggplot2. The advantage of the online documentation is that you can see all the example plots, and navigate between pages more easily.

5.4.1 Continuous position scales

The most common continuous position scales are `scale_x_continuous` and `scale_y_continuous`, which 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 specification of a variety of transformations, both linear and non-linear. The transformation is carried out by a "transformer," which describes the transformation, its inverse, and how to draw the labels. Table 5.2 lists some of the more common transformers.

Transformations are most often used to modify position scales, so there are shortcuts for x, y, and z scales: `scale_x_log10()` is equivalent to `scale_x_continuous(trans = "log10")`.

Of course, you can also perform the transformation yourself. For example instead of adding `scale_x_log`, you could plot `log(x)`. That produces an identical result inside the plotting region, but the the axis and tick labels won't be the same. If you use a transformed scale, the axes will be labelled in the original data space. In both cases, the transformation occurs before the statistical summary. Figure 5.4 illustrates this difference, and these commands produce the two plots in the figure:

```
qplot(carat, price, data=diamonds) + scale_x_log10() + scale_y_log10()
qplot(log10(carat), log10(price), data=diamonds)
```

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

Table 5.2: List of built-in transformers.

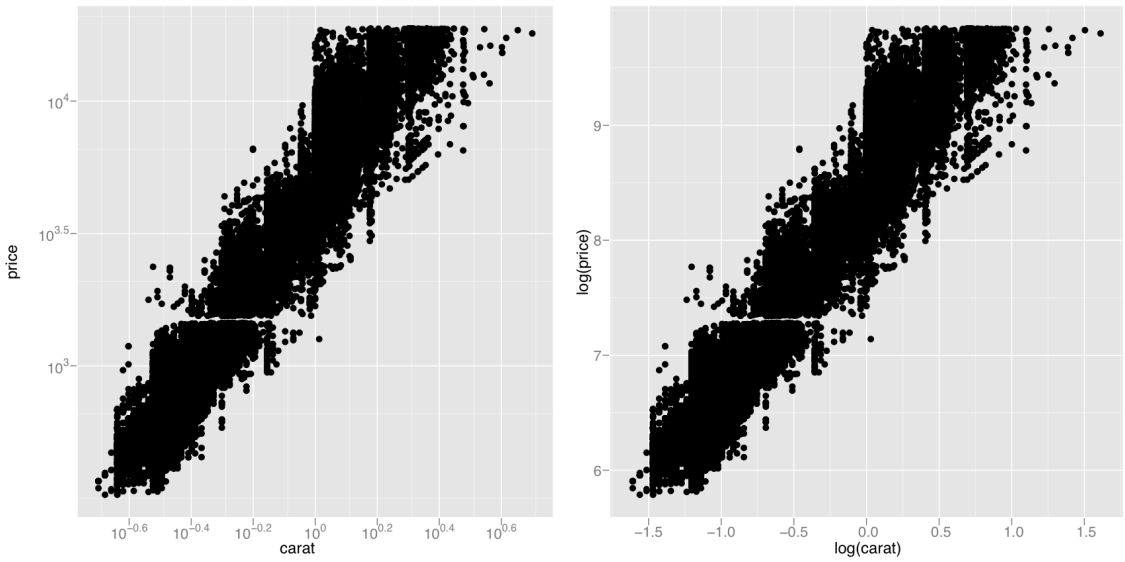


Figure 5.4: A scatterplot of diamond price vs. carat illustrating the difference between log transforming the scale (left) and log transforming the data (right). The plots are identical, but the axis labels are different.

Transformers are also used in `coord_trans`, where the transformation occurs after the statistic has been calculated, and affect the shape of the grob. `coord_trans` is described in more detail in Section XXX.

The `trans` argument works for any continuous scale, including the colour gradients described below, but the shortcuts only exist for position scales.

5.4.2 Colour gradients

There are three types of colour gradients available. A two–colour gradient, a three–colour gradient and a custom n–colour gradient. This section introduces you to a little bit of theory how these gradients work and shows you how to create your own for specific purposes.

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 of each eruption for the Old Faithful geyser in Yellowstone Park.

Figure 5.5 shows three gradients applied to this data.

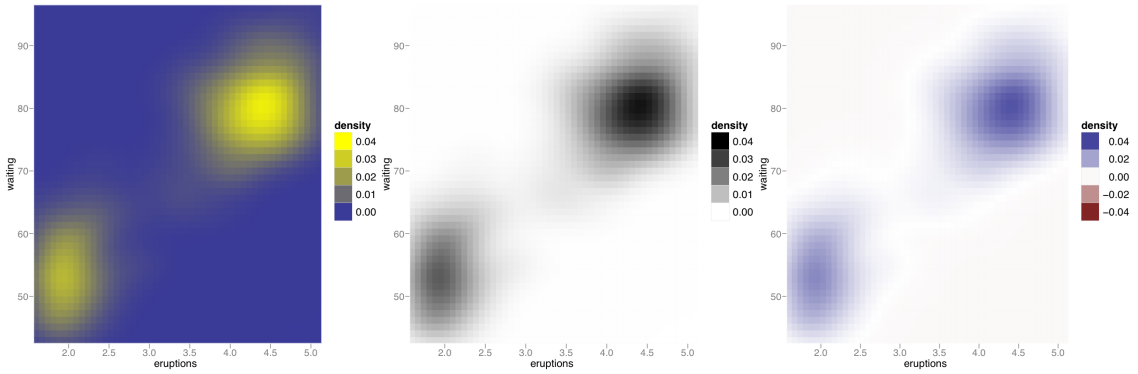


Figure 5.5: Density of eruptions with three colour schemes. (Left) default gradient colour scheme, (mid) customised gradient from white to black and (right) 3 point gradient with midpoint set to the median density.

You can also create your own custom gradient with `scale_colour_custom()`. 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. Figure 5.6 shows show palettes generates from routines in the `vcd` package. The technical report Zeileis et al. (2007) that describes the philosophy behind these palettes is a good introduction to some of the complexities of creating good colour scales.

5.4.3 Discrete scales

The discrete scales, `scale_linetype()`, `scale_shape()` and `scale_size_discrete()` basically have no options (although for the shape scale you can choose whether points should be fixed or solid). If you want to customise these scales, you need to create your own new scale with the manual scale. It has one important argument, `values` in which you specify the values that the scale should produce. If this vector is named, it will match the values

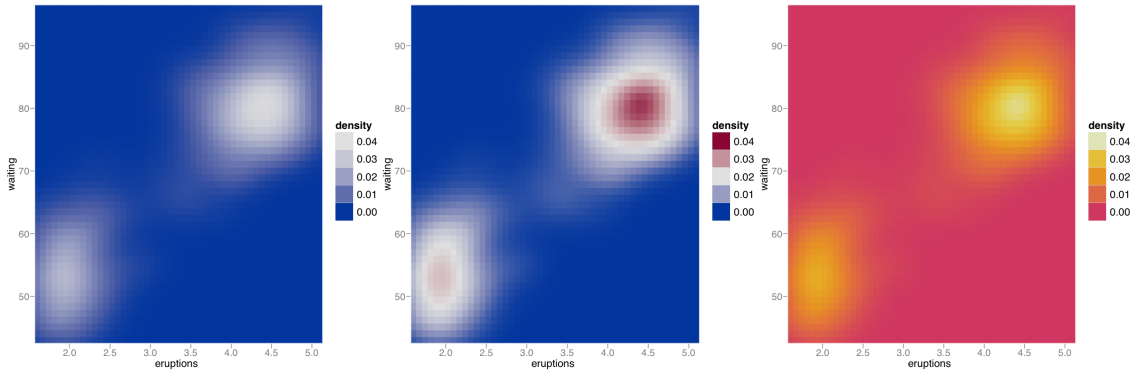


Figure 5.6: Gradient colour scales using perceptually well-formed palettes produced by the `vcd` package. From left to right: sequential, diverging and heat hcl palettes.

of the output to the values of the input, otherwise it will match in order of the levels of the discrete variable.

This scale is particularly useful if you'd like to override the default shape, size or linetype scales. You will need some knowledge of the valid aesthetic values, which are described in Appendix A.

5.4.4 The identity scale

The identity scale is used when your data is already in a form that the plotting functions in R understand, i.e. when the data and aesthetic spaces are the same. This means there is no way to derive a meaningful legend from the data alone, and by default a legend is not drawn. However, you can still use the `breaks` and `labels` arguments to set up a legend yourself.

Figure 5.7 shows one sort of data where `scale_identity` is useful. Here the data themselves are colours, and there's no way we could make a meaningful legend. The identity scale can also be useful in the case where you have manually scaled the data to aesthetic values. In that situation, you will have to figure out what breaks and labels make sense for your data.

5.5 Legends and axes

Collectively, axes and legends are called guides, and they are like the inverse of the scale: they allow you to read observations from the plot and map them back to their original values. Figure 5.8 labels the guides and their components. There are natural equivalents between the legend and the axis: legend name and axis label; legend keys and tick labels.

In `ggplot2`, legends and axes are produced automatically based on the scales and geoms that you used in the plot. This requires collecting information about how each aesthetic is used. We use the domain of the scale for the aesthetic to determine the value of the legend keys; we use a list of the geoms that use the aesthetic to determine how to draw the keys. For example, the point geom has points in the legend key and the lines geom has lines. If both points and lines are used then both will be drawn.

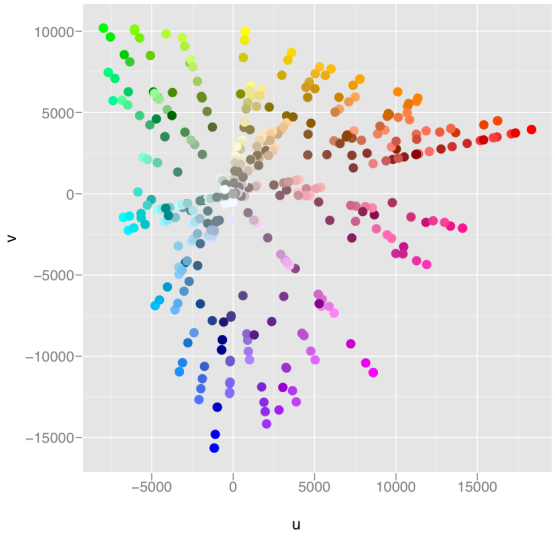


Figure 5.7: A plot of R colours in Luv space. A legend is unnecessary, because the colour of the points represents itself: the data and aesthetic spaces are the same.

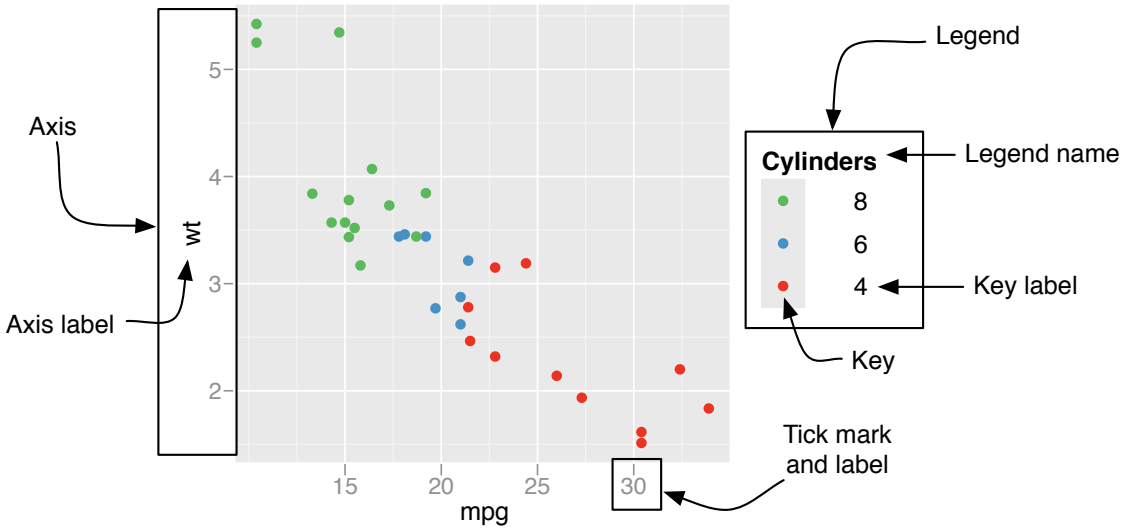


Figure 5.8: The components of the axes and legend.

`ggplot2` tries to use the smallest possible number of legends that accurately conveys the scales used in the plot. It does this by combining legends for the same variable with different aesthetics. Figure 5.9 shows an example of this.

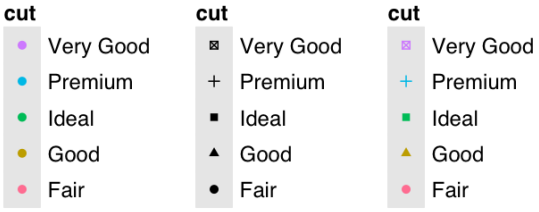


Figure 5.9: Colour legend, shape legend, colour + shape legend.

5.5.1 Customising appearance

It is possible to customize both the contents of the legends and the way they are rendered using the following arguments and options.

- The `breaks` and `labels` arguments to the scale function, introduced earlier in this chapter, are particularly important because they control what tick marks appear on the axis and what keys appear on the legend. If the breaks chosen by default are not appropriate (or you want to use more informative labels) setting these arguments will adjust the appearance of the legend keys and axis tick marks.

- The theme settings `axis.box`, `axis.title`, `axis.ticks`, `legend.box`, `legend.title`, and `legend.keys` control the visual appearance of axes and legends. For more details on how to manipulate these settings, see Section **??**.

- The internal grid lines are controlled by the breaks and minor breaks arguments. By default minor grid lines are spaced evenly in the original data space - this gives the common behaviour of log-log plots where major grid lines are multiplicative and minor grid lines are additive. You can override the minor grid lines with the `minor_breaks` argument.

- Position and justification of legends. Plot level option setting: `legend.position`, can be right, left, top, bottom, none (no legend), or a numeric position. The numeric position gives (in npc coordinates) the position of the corner given by `legend.justification`.

- Position scales also have the **expand** argument, which controls the amount of extra space added to axis limits. This is a numeric vector of length two: the first number is a multiplicative amount and the second is an additive constant. The default for continuous scales is `c(0.05, 0)` (i.e., add 5% extra space on each end); for discrete scales it is `c(0, 0.75)`). Set to `c(0, 0)` to eliminate extra space.

5.6 More resources

As you experiment with different aesthetic choices and new scales, it's important to keep in mind how the plot will be perceived. Some particularly good references to consult are:

- Cleveland (1993a, 1985); Cleveland and McGill (1987) for research on how plots are perceived and the best ways to encode data.
- Tufte (1990, 1997, 2001, 2006) for how to make beautiful, data-rich, graphics.
- Brewer (1994a,b) for how to colours that work well in a wide variety of situations, particularly for area plots.
- Carr (1994, 2002); Carr and Sun (1999), for the use of colour in general.

Chapter 6

Polishing your plots for publication

In this chapter you will learn how to prepare polished plots for publication. Most of this chapter focusses on the theming capability of `ggplot2` which allows you to control many non-data aspects of plot appearance, but you will also learn how to adjust geom, stat and scale defaults, and the best way to save plots for inclusion into other software packages. Together with the next chapter, manipulating plot rendering with `grid`, you will learn how to control every visual aspect of the plot to get exactly the appearance that you want.

The visual appearance of the plot is determined by both data and non-data related components. Section 6.1 introduces the theme system which controls all aspects of non-data display.

By now you should be familiar with the many ways that you can alter the data related components of the plot—layers and scales—to visualise your data and change the appearance of the plot. In Section 6.2 you will learn how you can change the defaults for these, so that you do not need to repeat the same parameters again and again.

Finally, Section 6.3 concludes the chapter with a discussion about how to get your graphics out of R and into LaTeX, Word or other presentation or word-processing software.

6.1 Themes

The appearance of non-data elements of the plot are controlled by the theme system. They do 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 customise the plot to be aesthetically pleasing or match existing style guides. Themes give you control over the things like the fonts in all parts of the plot: the title, axis labels, axis tick labels, strips, legend labels and legend key labels; and the colour of ticks, grid lines, and backgrounds (panel, plot, strip and legend).

This separation into data and non-data components is quite different from base and lattice graphics. In base and lattice graphics, most functions take a very large number arguments that specify the finer points of appearance, which can make the functions complicated and hard to understand. `ggplot2` takes a different approach: when creating the plot you determine how the data is display, then *after* it has been created you can edit every detail of the rendering, using the theming system. Figure 6.1 shows some of the effects of changing the theme of a plot. The two examples show the two themes included by default in `ggplot2`.

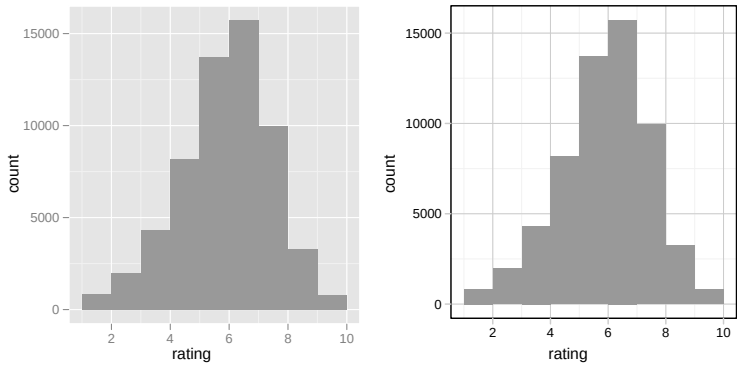


Figure 6.1: The effect of changing themes. *Left,* the default grey theme with grey background and white gridlines. *Right,* the alternative black and white theme with white background and grey gridlines. Notice how the bars, data elements, are identical in both plots.

Like many other areas of `ggplot2`, themes can be controlled on multiple levels from the very coarse to the very fine:

- Use a built-in theme. This affects every element of the plot in a visually consistent manner. The default theme uses a grey panel background with white gridlines, while the alternative theme uses a white background with grey gridlines. § 6.1.1.
- Modify a single element of a built-in themes. Each theme is made up of multiple elements The theme system comes with a number of built-in element rendering functions with a limited set of parameters. By adjusting these parameters you can control things like text size and colour, background and grid line colours and text orientation. By combining multiple elements you can create your own theme.
- Write a custom element function with grid. This allows you to absolutely customise the appearance of every element - you are not restricted to a fixed set of drawing options. § 6.1.2
- Use grid to alter a single item on the plot. Using grid, the underlying drawing system, gives you the ability to alter any element drawn on the plot. However, it comes at a cost of requiring a much deeper understand of how the plot is drawn. This is described in Chapter 7.

Generally each of these theme settings can be applied globally, to all plots, or locally to a single plot. How to do this is described individually for each section.

6.1.1 Built-in themes

There are two built-in themes. The default, `theme_gray()`, uses a very light grey background with white gridlines. This follows from the advice of Tufte (1990, 1997, 2001, 2006) and Brewer (1994a); Carr (1994, 2002); Carr and Sun (1999). We can still see the gridlines to aid in the judgement of position (Cleveland, 1993b), but they have little visual impact and

we can easily "tune" them out. The grey background gives the plot a similar colour (in a typographical sense) to the remainder of the text, ensuring that the graphics fit in with the flow of a text 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.

The other built-in theme, `theme_bw()`, has a more traditional white background with dark grey grid lines. Figure 6.1 show some of the difference between these themes.

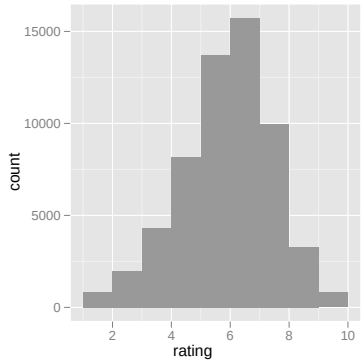
Both themes have a single parameter, `base_size`, which controls the base font size. The base font size is the size that the axis titles use: the plot title is 20% bigger, and the tick and strip labels are 20% smaller. If you want to control these sizes separately, you'll need to modify the individual elements as described in the following section.

You can apply themes in two ways:

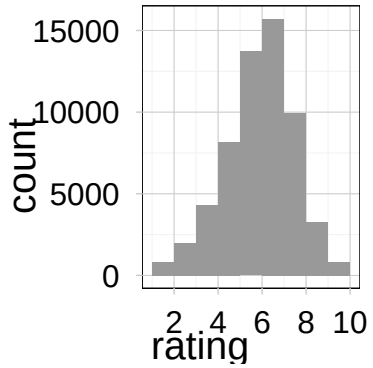
- Globally, affecting all plots when they are drawn: `set_theme(theme_grey())` or `set_theme(theme_bw())`. `theme_set()` returns the previous theme so that you can restore it later if you want.
- Locally, for an individual plot: `qplot(...) + theme_grey()`. A locally applied theme will override the global default.

The following example shows a few of these combinations:

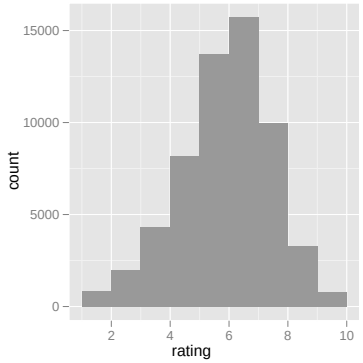
```
> histogram <- qplot(rating, data=movies, geom="histogram", binwidth=1)
> previous_theme <- theme_set(theme_bw())
>
> # Themes affect the plot when they are drawn, not when they are created
> histogram
```




```
>
> # Override the theme for a single plot by adding it on
> histogram + theme_bw(30)
```




```
>
> # Apply the original theme to a single plot
> histogram + previous_theme
```




```
>
> # Permanently restore the original theme
> theme_set(previous_theme)
```

6.1.2 Theme elements

A theme is made up of multiple *elements* which control the appearance of a single item on the plot, as listed in Table 6.1. The appearance of each element is controlled an *element function*. There are four main types of elements: segments and lines, rectangles, and text. Each of these elements has parameters that you can tune, as described in Section **??**, or you can write your own, as described in Section 7.4.

There are three elements that have individual `x` and `y` settings: `axis.text`, `axis.title` and `strip.text`. Having a different setting for the horizontal and vertical elements allows you to control how text should appear in different orientations.

6.1.3 Element functions

There are four basic types of built-in element functions: text, lines and segments, rectangles and blank. Each element function has a set of parameters that control the appearance as described below:

Theme element	Type	Description
`axis.line`	segment	Line along axis
`axis.text.x`	text	x axis label
`axis.text.y`	text	y axis label
`axis.ticks`	segment	axis tick marks
`axis.ticks.y`	segment	axis tick marks
`axis.title.x`	text	horizontal tick labels
`axis.title.y`	text	vertical tick labels
`legend.background`	rect	background of legend
`legend.key`	rect	background underneath legend keys
`legend.text`	text	legend labels
`legend.title`	text	legend name
`panel.background`	rect	
`panel.border`	rect	
`panel.grid.major`	line	major grid lines
`panel.grid.minor`	line	minor grid lines
`panel.empty`	rect	panel with no data
`plot.background`	rect	background of the entire plot
`plot.title`	text	plot title
`strip.background`	rect	
`strip.text.x`	text	text for horizontal strips
`strip.text.y`	text	text for vertical strips

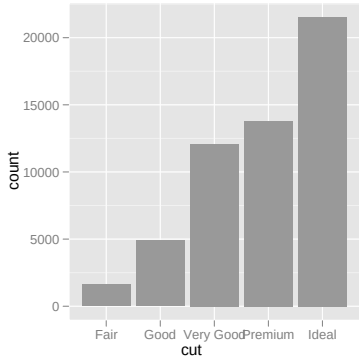
Table 6.1: Theme elements

- `theme_text()` draws labels and headings. You can control the font `family`, `face`, `colour`, `size`, `hjust`, `vjust`, `angle`, and `lineheight`.
- `theme_line()` and `theme_segment()` draw lines and segments with the same options but in a slightly different way. Make sure you match the appropriate type or you will get strange grid errors. For these element functions you can control the `colour`, `size`, and `linetype`.
- `theme_rect()` draws rectangles, mostly used for backgrounds, you can control the `fill`, `colour`, `size`, and `linetype`.
- `theme_blank` draws nothing. Use this element type if you don't want anything drawn, or any space allocated for that element.

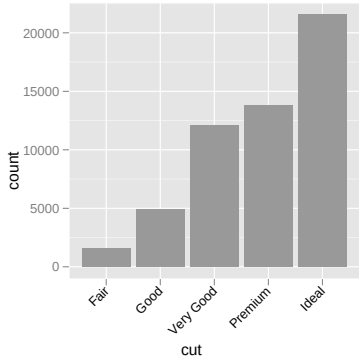
You can see the settings for the current theme with `theme_get()`. I haven't included the output here because it takes up several pages. (Debby: do you think I should include it regardless? In an appendix?). You can modify the elements locally for a single plot

with `opts()`, or globally for all future plots with `theme_update()`. The following examples shows how you can use these functions. It's a good idea to look at an existing theme for details about setting the correct angles and justifications for text elements.

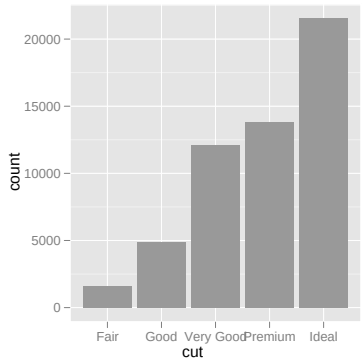
```
> (p <- qplot(cut, data=diamonds, geom="bar"))
```




```
> p + opts(axis.text.x = theme_text(angle = 45, hjust=1))
```




```
> old_theme <- theme_update(
+   plot.background = theme_rect(fill = "#3366FF"),
+   panel.background = theme_rect(fill = "#003DF5"),
+   axis.text.x = theme_text(colour = "#CCFF33"),
+   axis.text.y = theme_text(colour = "#CCFF33", hjust = 1),
+   axis.title.x = theme_text(colour = "#CCFF33", face = "bold"),
+   axis.title.y = theme_text(colour = "#CCFF33", face = "bold", angle = 90)
+ )
> p
```

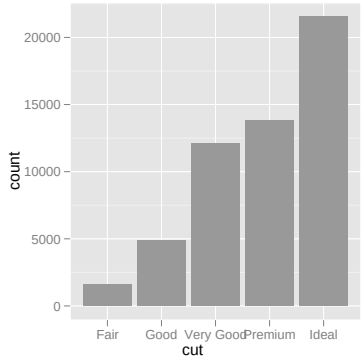



```
> theme_set(old_theme)
```

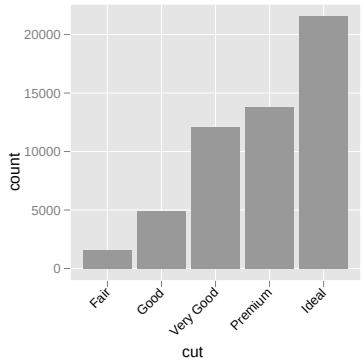
There is some duplication in this example because we have to specify the x and y elements separately. This is a necessary evil so that you can have total control over the appearance of the elements.

In the following example we use `theme_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` as parameter to create invisible elements that still take up space.

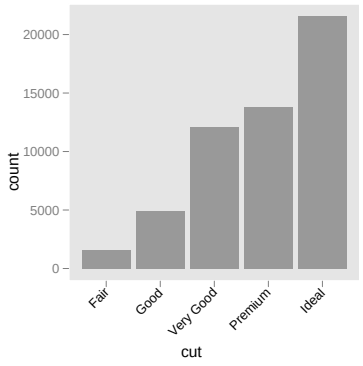
```
> p
```



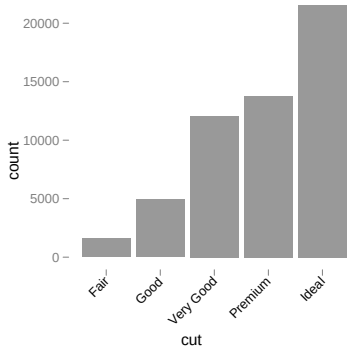

```
> last_plot() + opts(panel.grid.minor = theme_blank())
```



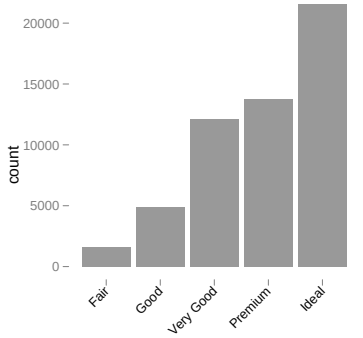

```
> last_plot() + opts(panel.grid.major = theme_blank())
```



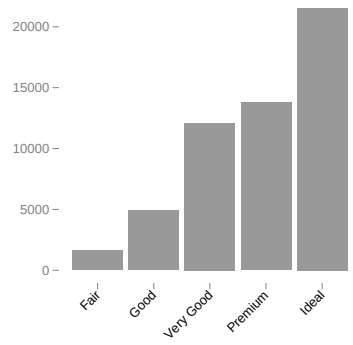

```
> last_plot() + opts(panel.background = theme_blank())
```



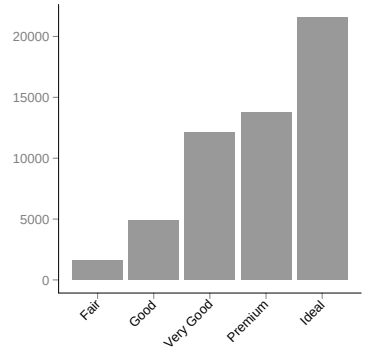

```
> last_plot() + opts(axis.title.x = theme_blank())
```




```
> last_plot() + opts(axis.title.y = theme_blank())
```




```
> last_plot() + opts(axis.line = theme_segment())
```



6.1.4 More advanced control

You can also write your own custom element functions. You'll need to know more about grid to do this, so this is described in Section 7.4.

6.2 Customising scales and geoms

When producing a consistent theme, you may also want to tune some of the scale and geom defaults. Rather than having to manually specify the changes every time you add the scale or geom, you can use the following functions to alter the default settings for scales and geoms.

6.2.1 Scales

To change the default scale associated with an aesthetic, use `set_default_scale()`. (See Table 5.1 for the defaults.) This function takes three arguments: the name of the aesthetic, the type of variable (discrete or continuous) and the name of the scale to use as the default. Further arguments override the default parameters of the scale. The following example sets up colour and fill scales for black and white printing:

```
> set_default_scale("colour", "discrete", "grey")
> set_default_scale("fill", "discrete", "grey")
> set_default_scale("colour", "continuous", "gradient",
```

Aesthetic	Default value	Geoms
colour	#3366FF	contour, density2d, quantile, smooth
colour	NA	area, bar, histogram, polygon, rect, tile
colour	black	abline, crossbar, density, errorbar, hline, line, linerange, path, pointrange, rug, segment, step, text, vline
colour	darkblue	jitter, point
colour	grey60	boxplot, ribbon
fill	NA	crossbar, density, jitter, point, pointrange
fill	grey60	area, bar, histogram, polygon, rect, ribbon, smooth, tile
linetype	1	abline, area, bar, contour, crossbar, density, density2d, errorbar, histogram, hline, line, linerange, path, pointrange, polygon, quantile, rect, ribbon, rug, segment, smooth, step, tile, vline
shape	19	jitter, point, pointrange
size	0.5	abline, area, bar, boxplot, contour, crossbar, density, density2d, errorbar, histogram, hline, line, linerange, path, pointrange, polygon, quantile, rect, ribbon, rug, segment, smooth, step, vline
size	2	jitter, point
weight	1	bar, boxplot, contour, density, density2d, histogram, quantile, smooth

Table 6.2: Default aesthetic values for geoms. See Chapter A for how the values are interpreted by R.

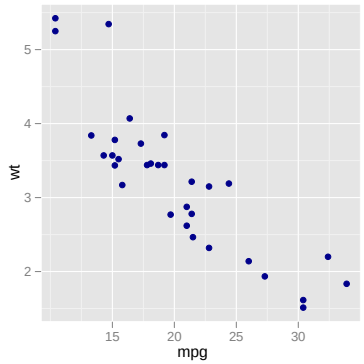
```
+   low = "white", high = "black")
> set_default_scale("fill", "continuous", "gradient",
+   low = "white", high = "black")
```

6.2.2 Geoms and stats

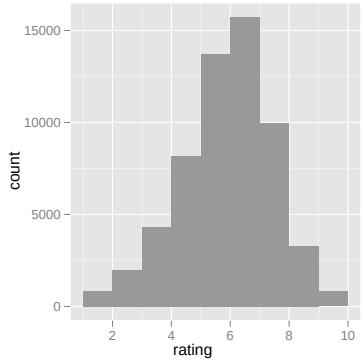
You can customise geoms and stats in a similar way with `update_geom_defaults()` and `update_stat_defaults()`. Unlike the other theme settings these will only affect plots *created* after the setting has been changed, not all plots drawn after the setting has been changed. The following example demonstrates changing the default point colour and changing the default histogram to a density ("true") histogram.

Table 6.2 lists all of the common aesthetic defaults. If you change one, it's a good idea to change all the other that you use to ensure that your plots look consistent. See Chapter A for how the values are interpreted by R.

```
> update_geom_defaults("point", aes(colour = "darkblue"))
> qplot(mpg, wt, data=mtcars)
```




```
> # update_stat_defaults("bin", aes(y = ..density..), binwidth = 1)
> qplot(rating, data=movies, geom="histogram", binwidth=1)
```



6.3 Saving your output

For interactive use, `ggsave()`, will use the size of the current graphics device (useful for ensuring a good aspect ratio), but when creating final versions it's recommended to set width and height so that all graphics for a document are the same size.

You have two basic choices of output: raster or vector. Vector graphics are procedural: . This means that they are essentially "infinitely" zoomable - there is no loss of detail. Raster graphics are stored as an array of pixels. Fixed resolution. Generally, vector output is more desirable, but for complex graphics containing thousands of graphical objects it can be slow to render. In this case, it may be better to switch to raster output. For printed use, a high-resolution (e.g. 600 dpi) graphic may be an acceptable compromise, but can be large.

Table 6.3 lists recommended graphic formats for various tasks. R output generally works best as part of a *nix development tool chain: using png or pdf output in LaTeXdocuments. With Microsoft Office things are little more complicated. You have two choices of vector output neither of which are perfect. The first option is to use Windows meta files (`wmf`), which are supported natively by Office but do not support transparency. PDFs do support transparency, but when included into a Office do not.

If you are using LaTeX, I recommend including `\DeclareGraphicsExtensions{.png, .pdf}` in the preamble. Then you don't need to specify the file extension in `includegraphics` commands, but LaTeXwill pick png files in preference to pdf. I choose this order because you can produce all your files in pdf, and then go back and re-render any big ones as pngs.

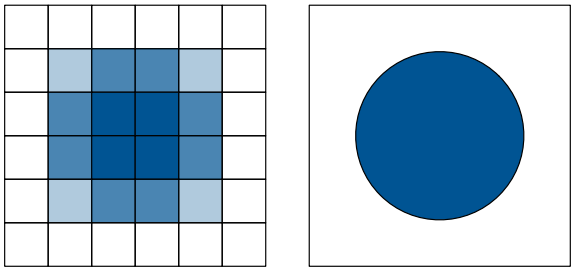


Figure 6.2: The difference between raster, *left*, and vector, *right*, graphics.

Graphics device	Type	Recommended for
pdf	vector	pdflatex
ps	vector	latex
wmf	vector	MS office
png	raster	web, pdflatex
tiff	raster	some publishers

Table 6.3: Recommended graphic output for different purposes.

Chapter 7

Manipulating plot rendering with `grid`

7.1 Introduction

What is `grid`? It's the graphics engine that powers ggplot. It is responsible for drawing the graphic object onto the screen or saving it to a graphics file. It provides a system of viewports, which define regions on the plot, and a comprehensive set of units for describing size and position. This chapter can not hope to provide a comprehensive introduction to grid, but should hopefully provide enough examples to get you going. I highly recommend the book "R Graphics" (Murrell, 2005b), by the author of grid, as a companion to this chapter. If you can't get the book, at least read Chapter 5, "The grid graphics model", which is available online for free at `http://www.stat.auckland.ac.nz/~paul/RGraphics/chapter5.pdf`.

The grobs (graphical objects) used in this chapter are a bit different to the geoms (geometric objects) used in previous chapters. A grob is the object that is actually drawn onto the screen, while a geom is a more abstract object which describes the type of object used to draw a plot. An example may make this more clear. In a line plot, the geom describes that the data should be visualised with a line, and the grobs draw the line itself, as well as the other lines that appear in the grid and axes.

The chapter begins with a discussion of the structure of viewports (§ 7.2) and grobs (§ 7.3) used by `ggplot2`, and then continues to describe the four principle ways to enhance a plot with grid:

- Custom element functions
- Edit existing objects on the plot, § 7.5.
- Add annotations to the plot, § 7.7.
- Removing grobs from a plot, § 7.6.
- Arrange multiple plots on a single page, § 7.8.

7.2 Plot viewports

Viewports define the basic regions of the plot. The structure will vary slightly from plot to plot, depending on the type of faceting used, but the basics will remain the same.

The panels viewport contains the meat of the plot: strip labels, axes and faceted panels. The viewports are named according to both their job and their position on the plot. A prefix (listed below) describes the contents of the viewport, and is followed by integer x and y position (counting from bottom left) separated by "_". Figure 7.1 illustrates this naming scheme for a 2×2 plot.

- strip_h: horizontal strip labels
- strip_v: vertical strip labels
- axis_h: horizontal axes
- axis_v: vertical axes
- panel: faceting panels

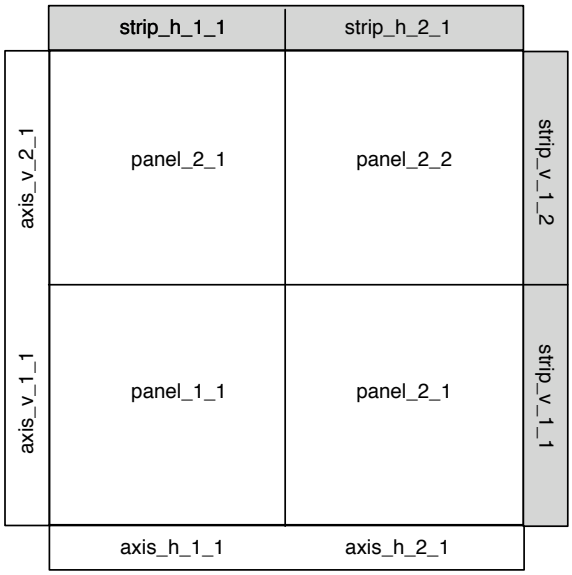


Figure 7.1: Naming scheming of the panel viewports

The panels viewport is contained inside the background viewport which also contains the following viewports:

- title, xlabel, and ylabel: for the plot title, and x and y axis labels
- legend_box: for all of the legends for the plot

Figure 7.2 labels a plot with a representative sample of these viewports. To get a list of all viewports on the current plot, run current.vpTree(all=TRUE).

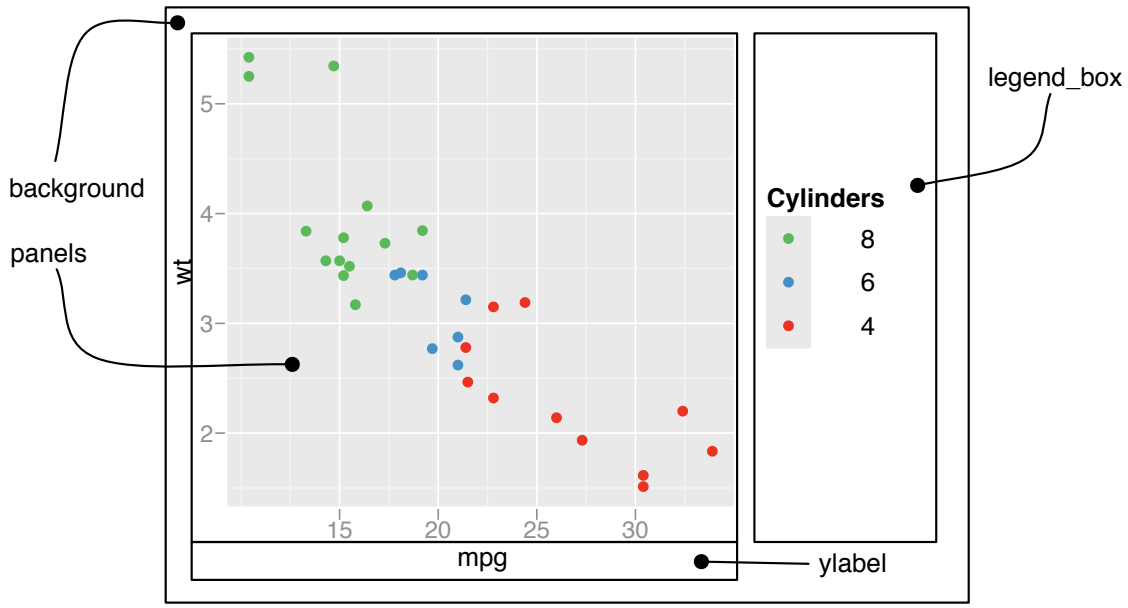


Figure 7.2: Diagram showing the structure and names of viewports.

7.3 Plot grobs

Grob names have three components: the name of the grob, the class of the grob, and a unique numeric suffix. The three components are joined together with "." to give a name like `title.text.435` or `ticks.segments.15`. These three components ensure that all grob names are unique, and allow you to select multiple grobs with the same name at the same time.

You can see a list of all the grobs in the current plot with `grid.ls()`. If you only want to see the ggplot name of the grob, `grid.ls(only.name=TRUE)` will reduce a lot of the output. Here's an example after drawing the a simple plot:

```
plot-surrounds::
 background
 plot::
  background
  guide:: (background, major-horizontal, major-vertical,
           minor-horizontal, minor-vertical, border)
  xaxis::
   ticks
   labels:: (label, label, label, label, label, label, label, label)
  yaxis::
   ticks
   labels:: (label, label, label, label, label)
  geom_point
 ylabel
 xlabel
```

`title`

Figure 7.3 labels some of these grobs. The grobs are arranged hierarchically, but it's hard to capture this in a diagram.

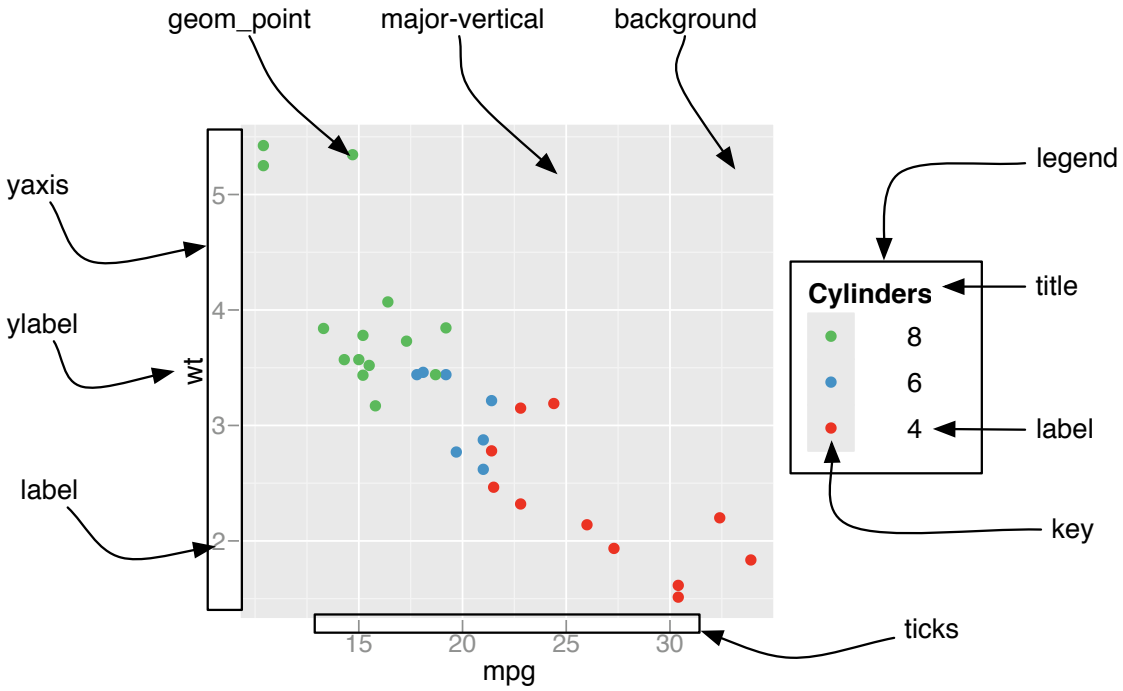


Figure 7.3: A selection of the most important grobs. `geom_point`, `major-vertical` and `label` point to a single element of the grob.

The most important components are:

- The geom displayed in the plot: `geom_point`.
- `xaxis` and `yaxis`, the axes, containing `labels` and `ticks`.
- Axis labels and title: `xlabel`, `ylabel`, `title`.
- `guide`, the internal guides within a panel (background, and grid lines)

7.4 Custom element functions

To see how to write custom element functions, it's good to start by seeing how the built-in element functions work:

```
> str(args(theme_text()))
function (label, x = 0.5, y = 0.5, ..., vjust = vj, hjust = hj,
    default.units = "npc")
> str(args(theme_rect()))
```

Grid parameter	ggplot2 aesthetic	Description
lwd	size	Line width (in pts)
col	colour	Border colour
fill	fill	Fill colour
fontsize	size	Font size (in pts)
fontface	—	Font face (bold, italic, ...)

Table 7.1: Common graphical parameters for grid grobs. Note that point size is controlled separately.

```
function (x = 0.5, y = 0.5, width = 1, height = 1, ...)
> str(args(theme_line()))
function (x = 0:1, y = 0:1, ..., default.units = "npc")
```

You'll notice that these are very similar to the arguments to `textGrob()`, `rectGrob()` and `polylineGrob()` and these are exactly the functions that they are based on. All that the element function do is set up some defaults.

If you want to write your own, you need to copy this basic idea: take position arguments, and return a grid grob. For example, let's say we'd like to give the strips a 3d appearance. We can do this by drawing a rectangle, and then drawing highlights on the top-right and low-lights (shadows) on the bottom-left.

7.5 Editing existing objects on the plot

From time to time, the theming system will not give you enough control. This may occur if you want to modify a single element of a

Where possible, using the theming system will be easier, because using grid will affect a component, but not the space saved for that element.

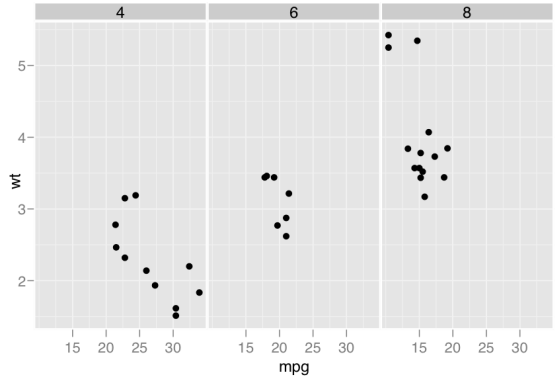
Most of the difficulty in modifying elements of the plot is figuring out what the grob you want to modify is called. Once you have that you can use `grid.gedit`, to locate and then modify that grob.

To fully identify a grob, you need to use a `gPath`. A gPath can either be a string specifying a single grob name, or a sequence of grob names that describe hierarchy to travel down to get to the grob of interest with the `gPath` function. Using a string will find all grobs with that name regardless of their position in the hierarchy. For example, `"label"` will find all grobs called label, regardless of where they are. To be more specific, using `gPath("parent", "child")` will only find grobs named "child" with a parent called "parent". For example, `gPath("xaxis", "label")` will locate only labels on the x-axis.

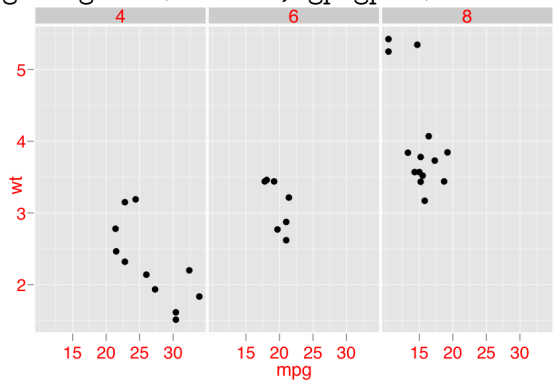
Modifying a grob requires some knowledge of the different parameters of the grob. This is where the second part of the grob name is useful, as it will tell you whether you are modifying a line, or a rect or a text grob. You can get more information by looking at the documentation for that grob, eg. `?grid.rect, ?grid.text, ?grid.lines` As well as individual parameters, all grobs share a common set of graphical parameters described in Table 7.1. Appendix A describes the values that these may take.

In this example, we edit the font of all labels.

```
qplot(mpg, wt, data=mtcars, facets = . ~ cyl)
```

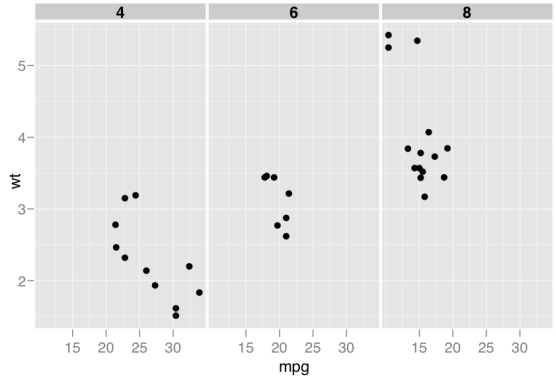



```
grid.gedit("label", gp=gpar(fontsize=14, col="red"))
```

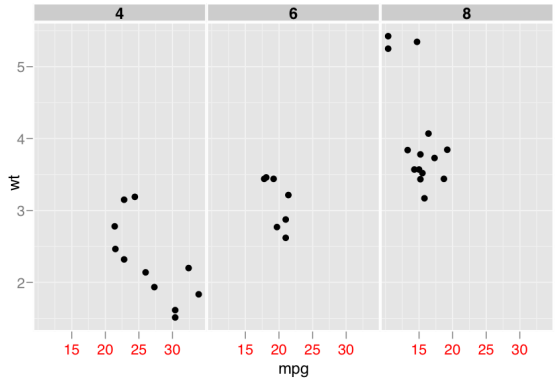


To edit just one type of label, we need to use the hierarchy of grobs and the `gPath` function:

```
qplot(mpg, wt, data=mtcars, facets = . ~ cyl)
grid.gedit(gPath("strip","label"), gp=gpar(fontface="bold"))
```




```
grid.gedit(gPath("yaxis", "labels"), gp=gpar(col="red"))
```



7.6 Removing grobs

You can use `grid.remove()` to completely remove a grob. However, the space it takes up will remain there:

```
qplot(mpg, wt, data=mtcars, facets = . ~ cyl)
grid.gremove(gPath("strip", "background"))
grid.gremove(gPath("guide", "background"))
grid.gremove("major")
grid.gremove("axis")
```

You can use an alternative method to remove top level viewports. There are plot-level options `keep` and `drop`, which specify which viewports to keep or drop respectively. For example:

```
p <- qplot(mpg, wt, data=mtcars, main = "My plot")
p + opts(keep = "panel")
p + opts(ignore = c("xlabel", "ylabel"))
```

7.7 Adding annotations

Many annotations can be done with `geom_text`, `geom_abline`, `geom_vline` and `geom_hline`, so try those first. If you need more flexibility you can add annotations with grid. When you add annotations to a plot you need to specify where they will appear. In grid this is described by a system of viewports. Different viewports describe different regions of output on the plot, for example, the axes, the plotting region and the faceting strips.

To add annotations to a plot you have to specify the viewport when you add extra grobs. For example:

```
qplot(wt, mpg, data=mtcars, colour=cyl)
grid.circle(vp="layout::panel_1_1")
```

Panel viewports will have a coordinate system set up for points, while x- and y- axes will only have one dimension defined. For example, on the x-axis there will be native coordinates for the x-dimension, but not the y-dimension.

```
qplot(wt, mpg, data=mtcars, colour=cyl)
grid.lines(x=unit(c(0,1), "npc"), y=unit(23, "native"), vp="layout::panel_1_1")
grid.lines(x=unit(c(0,1), "npc"), y=unit(23, "native"), vp="layout::axis_v_1_1")
```

7.8 Customising layout

By default, showing a ggplot object at the R command prompt will display to the screen. To exercise more control, you can call print explicitly. This section describes some of the things you can do. For more details see ?print.ggplot and ?ggplot_print.

If you just want the plot (no labels, titles or legends) you can use pretty = FALSE

```
p <- qplot(wt, mpg, data=mtcars, colour=cyl)
print(p, pretty = FALSE)
```

By default, ggplot always clears the screen and draws to the entire device. You customise this in two ways. One way is to setup a viewport and push it on to the display, then draw the plot with newpage=FALSE. pushViewport adds the viewport to the list of viewports on the display. Afterwards, upViewport returns you to the viewport for the entire page, preparing you for the next set of output.

```
p <- qplot(wt, mpg, data=mtcars, colour=cyl)
grid.newpage()
pushViewport(viewport(height=0.4, width=0.4, x=0.4, y=0.8))
print(p, newpage=FALSE, pretty=FALSE)
upViewport()
```

Alternatively, you can set up your own set of viewports, and then specify which one the plot should be drawn to. Here we use upViewport before displaying the plot so we are in the top level viewport before we start plotting.

```
grid.newpage()
pushViewport(viewport(height=0.5, width=0.5, x=0.5, y=0.5, name="small", angle=40))
upViewport()
print(p, vp="small")
```

Obviously, this is very useful if you want to layout plots in a complicated grid. In this case, grid.layout is very useful, as it allows you to set up a grid of viewports with arbitrary heights and widths. You still need to create each viewport, but instead of explicitly specifying the position and size, you can specify the row and column of the layout.

```
p <- qplot(wt, mpg, data=mtcars, colour=cyl)

vplayout <- function(x, y) viewport(layout.pos.row=x, layout.pos.col=y)
grid.newpage()
pushViewport(viewport(layout=grid.layout(3,3)))

print(p, vp=vplayout(1,1))
print(p, vp=vplayout(2:3,2:3))
print(p, vp=vplayout(1, 2:3))
```

```
print(p, vp=vplayout(2:3, 1))
```

This is useful for arranging plots in a wider range of ways than what you can do with faceting. You should be careful to ensure that scales are consistent over the different plots. There is currently no easy way to do this, except to keep track of the maximum and minimum yourself, and then manually set the scales of the plot.

7.9 Saving your work

Using `grid.gedit()` works fine if you are editing the plot on screen, but if you want to save it to disk you need take some extra steps, or you will end up with multiple pages of output, each showing one change. The key is not to modify the plot on screen, but to modify the plot grob, and then draw it once you have made all the changes.

```
p <- qplot(wt, mpg, data=mtcars, colour=cyl)
# Get the plot grob
grob <- ggplotGrob(p)
# Modify it place
grob <- geditGrob(grob, gPath("strip","label"), gp=gpar(fontface="bold"))

# Draw it
grid.newpage()
grid.draw(grob)
```

Appendices

Appendix A

Aesthetic specifications

This appendix summarises the various formats that `grid` drawing functions take. Most of this information is available scattered throughout the R documentation. This appendix brings it all together in one place.

A.1 Colour

Colours can be specified with:

- A **name**, e.g. `"red"`. The colours are displayed in Figure A.1(a), and can be listed in more detail with `colours()`. The Stower's institute provides a nice printable pdf that lists all colours: http://research.stowers-institute.org/efg/R/Color/Chart/.
- An **rgb specification**, with a string of the form `"#RRGGBB"` where each of the pairs `RR`, `GG`, `BB` consist of two hexadecimal digits giving a value in the range `00` to `FF`. Partially transparent can be made with `alpha()`, e.g. `alpha("red", 0.5)`
- An **NA**, for a completely transparent colour.

The functions `rgb()`, `hsv()`, `hcl()` can be used to create colours specified in different colour spaces.

A.2 Line type

Line types can be specified with:

- A **integer** or **name**: 0=blank, 1=solid, 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash), illustrated in Figure A.1(b)
- The lengths of on/off stretches of line. This is done with a string of an even number (up to eight) of hexadecimal digits which give the lengths in consecutive positions in the string. For example, the string `"33"` specifies three units on followed by three off and `"3313"` specifies three units on followed by three off followed by one on and finally three off.

  The five standard dash-dot line types described above correspond to 44, 13, 134, 73, and 2262.

Note that `NA` is not a valid value for `lty`.

A.3 Shape

Shapes take four types of values:

- An **integer** in $[0, 25]$, illustrated in Figure A.1(c).
- A **single character**, to use that character as a plotting symbol.
- A `.` to draw the smallest rectangle that is visible (i.e. about one pixel).
- An **NA**, to draw nothing.

While all symbols have a foreground colour, symbols 19–25 also take a background colour (fill).

A.4 Size

Throughout `ggplot2`, for text height, point size and line width, size is specified in millimetres.

A.5 Justification

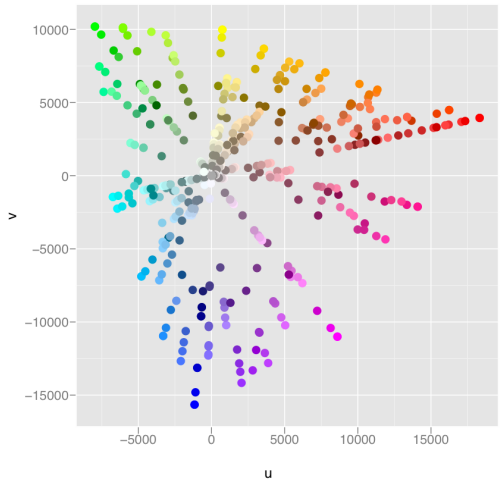
Justification of a string (or legend) defines the location within the string that is placed at the given position. There are two values for horizontal and vertical justification. The values can be:

- A **string**: `"left"`, `"right"`, `"centre"`, `"center"`, `"bottom"`, and `"top"`.
- A **number** between 0 and 1, giving the position within the string (from bottom-left corner). These values are demonstrated in Figure A.1(d).

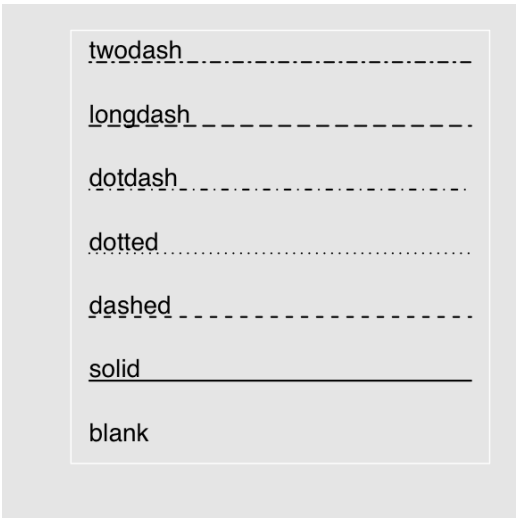
A.6 Fonts

postscriptFonts, pdfFonts, quartzFonts
Find R news article

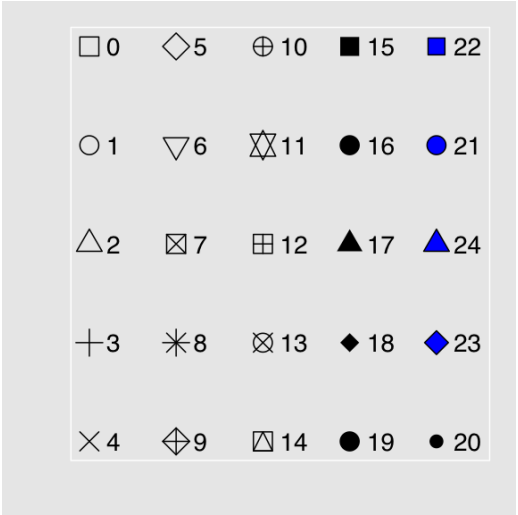
- `face`
- `family`
- `lineheight`
- `fontsize`



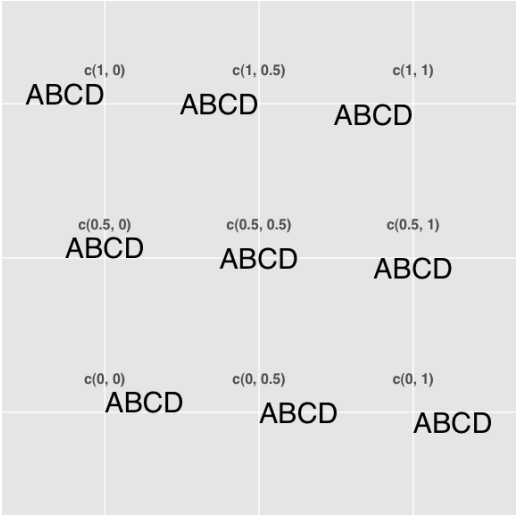
(a) All named colours in Luv space



(b) Built-in line types



(c) R plotting symbols. Colour is black, and fill is blue. Symbol 25 (not shown) is symbol 24 rotated 180 degrees.



(d) Horizontal and vertical justification settings.

Figure A.1: Examples illustrating different aesthetic settings.

Bibliography

A. Azzalini and A. W Bowman. A look at some data on the old faithful geyser. *Applied Statistics*, 39:357–365, 1990.

Cynthia A. Brewer. Color use guidelines for mapping and visualization. In A.M. MacEachren and D.R.F. Taylor, editors, *Visualization in Modern Cartography*, chapter 7, pages 123–147. Elsevier Science, Tarrytown, NY, 1994a.

Cynthia A. Brewer. Guidelines for use of the perceptual dimensions of color for mapping and visualization. In *Color Hard Copy and Graphic Arts III, Proceedings of the International Society for Optical Engineering (SPIE), San Jose*, volume 2171, pages 54–63, 1994b.

Dan Carr. Using gray in plots. *ASA Statistical Computing and Graphics Newsletter*, 2(5): 11–14, 1994. URL http://www.galaxy.gmu.edu/~dcarr/lib/v5n2.pdf.

Dan Carr. Graphical displays. In Abdel H. El-Shaarawi and Walter W. Piegorsch, editors, *Encyclopedia of Environmetrics*, volume 2, pages 933–960. John Wiley & Sons, Ltd, Chichester, 2002. URL http://www.galaxy.gmu.edu/%7Edcarr/lib/EnvironmentalGraphics.pdf.

Dan Carr and Ru Sun. Using layering and perceptual grouping in statistical graphics. *ASA Statistical Computing and Graphics Newsletter*, 10(1):25–31, 1999.

John Chambers, William Cleveland, Beat Kleiner, and Paul Tukey. *Graphical methods for data analysis*. Wadsworth, 1983.

William Cleveland. *Visualizing data*. Hobart Press, 1993a.

William Cleveland. A model for studying display methods of statistical graphics. *Journal of Computational and Graphical Statistics*, 2:323–364, 1993b. URL http://stat.bell-labs.com/doc/93.4.ps.

William Cleveland. *The Elements of Graphing Data*. Hobart Press, 1985.

William S Cleveland and Robert McGill. Graphical perception: The visual decoding of quantitative information on graphical displays of data. *Journal of the Royal Statistical Society. Series A (General)*, 150(3):192–229, 1987.

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

R. Koenker. *Quantile Regression*. Econometric Society Monograph Series. Cambridge University Press, 2005.

Jim Lemon, Ben Bolker, Sander Oom, Eduardo Klein, Barry Rowlingson, Hadley Wickham, Anupam Tyagi, Olivier Eterradossi, Gabor Grothendieck, Michael Toews, and John Kane. *plotrix: Various plotting functions*, 2008. R package version 2.4-3.

David Meyer, Achim Zeileis, and Kurt Hornik. The strucplot framework: Visualizing multi-way contingency tables with vcd. *Journal of Statistical Software*, 17(3):1–48, 2006. URL http://www.jstatsoft.org/v17/i03/.

Paul Murrell. *grid: Grid Graphics*, 2005a. R package version 2.2.0.

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

Paul Murrell. *R graphics*. Chapman & Hall/CRC, 2005b.

Naomi Robbins. *Creating More Effective Graphs*. Wiley-Interscience, 2004.

Deepayan Sarkar. *lattice: Lattice Graphics*, 2008a. R package version 0.17-6.

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

Gregory R. Warnes. Includes R source code and/or documentation contributed by Ben Bolker and Thomas Lumley. *gplots: Various R programming tools for plotting data*, 2007. R package version 2.6.0.

Edward R. Tufte. *Envisioning information*. Graphics Press, Chesire, Connecticut, 1990.

Edward R. Tufte. *Visual explanations*. Graphics Press, Chesire, Connecticut, 1997.

Edward R. Tufte. *The visual display of quantitative information*. Graphics Press, Chesire, Connecticut, 2001.

Edward R. Tufte. *Beautiful evidence*. Graphics Press, Chesire, Connecticut, 2006.

John W. Tukey. *Exploratory data analysis*. Addison Wesley, 1977.

Hadley Wickham. A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, Tentatively accepted.

Hadley Wickham. *Practical tools for exploring data and models*. PhD thesis, Iowa State University, 2008. URL http://had.co.nz/thesis.

Hadley Wickham, Michael Lawrence, Duncan Temple Lang, and Deborah F Swayne. An introduction to rggobi. *R-news*, Under revision. URL http://ggobi.org/rggobi.

Leland Wilkinson. *The Grammar of graphics*. Statistics and Computing. Springer, 2nd edition, 2005.

Achim Zeileis, Kurt Hornik, and Paul Murrell. Escaping RGBland: Selecting colors for statistical graphics. Report 61, Department of Statistics and Mathematics, Wirtschaftsuniversität Wien, Research Report Series, November 2007. URL http://epub.wu-wien.ac.at/dyn/openURL?id=oai:epub.wu-wien.ac.at:epub-wu-01_c87.