ggplot

Hadley Wickham

August 31, 2007

Contents

1 Introduction

ggplot is an R package for producing statistical graphics. It builds on top of the grid graphics system (Murrell, 2005a), and uses the philosophy outlined in the Grammar of Graphics (Wilkinson, 2005). Because ggplot is based on a formal model of graphics, it provides a powerful and flexible plotting system that is still easy to use. ggplot 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 ggplot with lots of example code and graphics. It also explains the grammar on which ggplot is based. Like other formal systems, ggplot is useful even when you don't understand the underlying model. However, the more you learn about the grammar, the more effectively you'll be able to use ggplot. This book will introduce you to ggplot as a novice, unfamiliar with the grammar, and turn you into an expert who can build new components to extend the grammar.

This book assumes some familiarity with R, to the level described in Dalgaard's Introductory Statistics with R.

1.1 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. Facetting can be used to generate the same plot for different subsets of the dataset. It is the combination of these independent components that make up a graphic.

The grammar, my interpretation of the grammar, and how it is used in ggplot is described in detail in chapter 3. Mastery of the grammar will go hand in hand with mastery of ggplot.

The components of the grammar are described in more detail below.

- Data is the most important thing, and the thing that you bring to the table.
- Geometric objects (or geoms for short) represent what you actually see on the plot: points, lines, polygons, etc.
- Statistics transform the data in many useful ways. For example, binning and counting to create a histogram. They are optional, but very useful.
- Scales 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, to make it possible to read the original data values off the graph.

- A coordinate system 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 facetting, or conditioning, specification. It is often useful to be able to reproduce the same plot for different subsets of the data. The facetting specification describes those subsets and how the facets should be arranged in to a plot.

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

- It doesn't advise 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 these sources: Chambers et al. (1983); Cleveland (1993); 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 ggplot does. Similarly, the grammar does not specify how to make an attractive graphic, and while the defaults in ggplot 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. ggplot can only create static graphics, so for dynamic and interactive graphics you will have to look elsewhere. [Mondrian book] and [GGobi book] provide excellent introductions to two different interactive graphics packages: Mondrian and GGobi.

1.2 How does ggplot 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 ggplot 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) (priv. comm.). Base graphics has basically a pen on paper model: you can only draw on top, not modify or delete existing content or change the axes etc. There is no (user accessible) representation of the graphics, apart from the appearance on the plot. Base graphics includes both tools for drawing primitives and entire plots. No other plotting system in R does this. 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 was, and is, developed by Paul Murrell, growing out of his PhD work, with the

initial aim of building a solid foundation for "something trellis-like". Deepayan Sarkar soon took on the development of the trellis-like stage, leaving Paul to build an excellent foundation (priv. comm.).

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 (**?**), developed by Deepayan Sarkar, uses grid graphics to implement the trellis graphics system of Cleveland (1993, 1994), 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.

- The vcd package Meyer et al. (2006), provides a framework for visualising high-dimensional contingency tables with mosaic and sieve plots.

- ggplot, 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 ggplot makes it easy to describe a wide range of graphics with a compact syntax, and independent components make extension easy. Like lattice, ggplot 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 implement specialist graphics, but no others provide a framework for producing statistical graphics. While this is fine if you just want to produce a one-off graphic, it can be hard to seamlessly combine these with other graphics you are using.

1.3 About this book

The book is structured to lead you from being a new user of ggplot to a developer creating new components for specialised plots:

- Chapter One describes how you can quickly get started using `qplot` to make graphics, just like you can using `plot`. This chapter introduces several important ggplot concepts: grob functions, aesthetic mappings and facetting.

- While `qplot` is a quick way to get started, you are not using the full power of the grammar. Chapter Two describes how the grammar of graphics and how the grammar of ggplot differs to the original. The theory will be illustrated 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. You will also learn some techniques using the reshape package to get data into a convenient form for ggplot.

The following chapters have yet to be written.

- The most crucial components of a plot are the geometric and statistic objects, and Chapters Three and Four describes what they do, how they work, and list the most commonly used. Mastery of this chapter will give you the ability to pick and choose the most appropriate tool for your visual display needs. The chapter concludes by showing you how to build your own geom and stat objects so that you can extend ggplot for your needs.

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

- Non-cartesian coordinate systems are somewhat rare, but when you need them, it's hard to go without. In Chapter Six, the different coordinate systems are described and illustrated, and you will learn how to create you own.

- Sometimes you need more control over the output than ggplot provides. In this case, you will need to modify the low level grid output used to draw the graphics. In Chapter Seven, you will learn how this output is constructed, how to control and modify it, and how to add additional annotations to the plot.

The ggplot website, `http://had.co.nz/ggplot`, provides updates to this book, information about features in the latest version of ggplot, and talks and papers related to ggplot. 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 ggplot 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.

1.4 Installation

To get started using ggplot, the first thing you need to do is install it. Make sure you have a recent version of R from `http://r-project.org`, and then follow the instructions below to download and install the ggplot package.

There are usually two versions of ggplot available, one stable version and one development version. The stable version is well-tested and well-documented before release. It is available on CRAN, and can be installed with the following R code:

```
install.packages("ggplot2", dep=TRUE)
```

A changelog listing changes between versions is available on the ggplot website. I will do my best to make sure that changes are backward compatible, so you shouldn't have to rewrite your old code. However, from time to time, I may need to make bigger changes that do affect your code. If you need to ensure that your old code will continue to run, I would recommend using use R's versioned installs:

```
install.packages("ggplot2", dep=TRUE, installWithVers=TRUE)
```

Now installed packages will have a version number associated with them, and you can load a specific version like so:

```
library(ggplot2, version="0.5")
```

`ggplot` isn't perfect, so from time-to-time you may encounter something that doesn't work the way you think it should. If this happens, please email me h.wickham@gmail.com a reproducible example of your problem, as well as a description of what you think should happen. The more information you provide, the more likely I am going to be able to help you.

1.5 Acknowledgements

Many people have contributed to this book with high-level structural insights, 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.

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, normally requiring several lines of code in R, 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.

`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. Page 10.
- How to map variables to aesthetic attributes, like colour, size and shape. Page 11.
- How to create many different types of plots by specifying different geoms, and how to combine multiple types in a single plot. Page 13.
- The use of facetting, also known as trellising or conditioning, to break apart subsets of your data. Page 27.
- How to tune the appearance of the plot by specifying some basic options. Page 29.

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 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

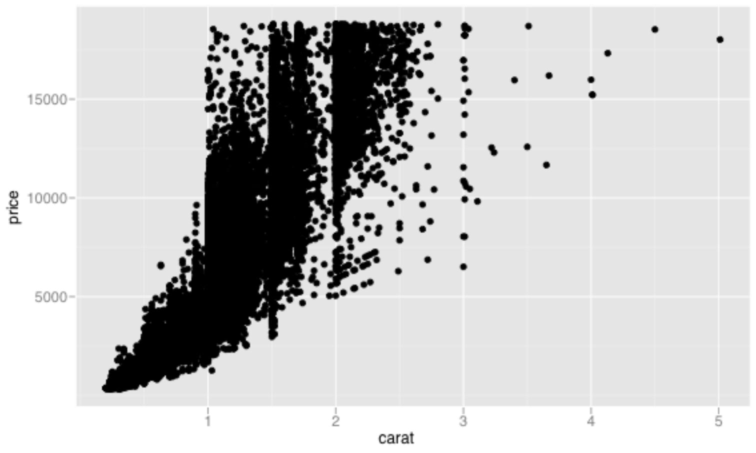
Table 2.1: `diamonds` dataset

2.3 Basic use

Just like `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. I recommend that you keep all the data for one plot in a data frame, instead of scatter across multiple vectors.

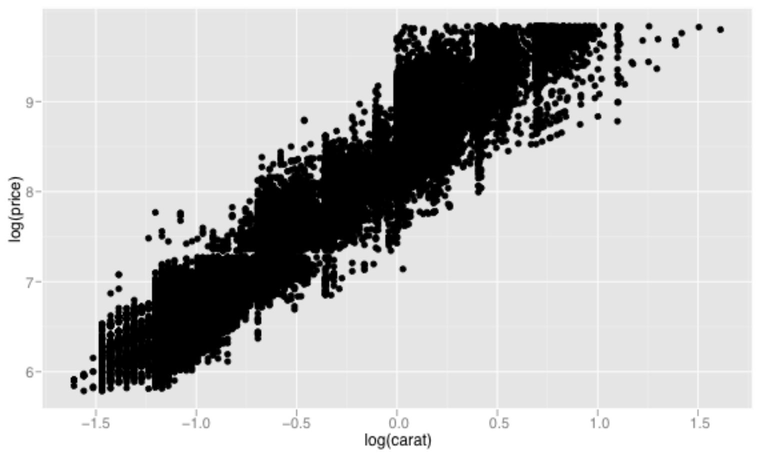
Here is a simple example of using `qplot`, investigating the relationship between price and carats (weight) of a diamond.

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

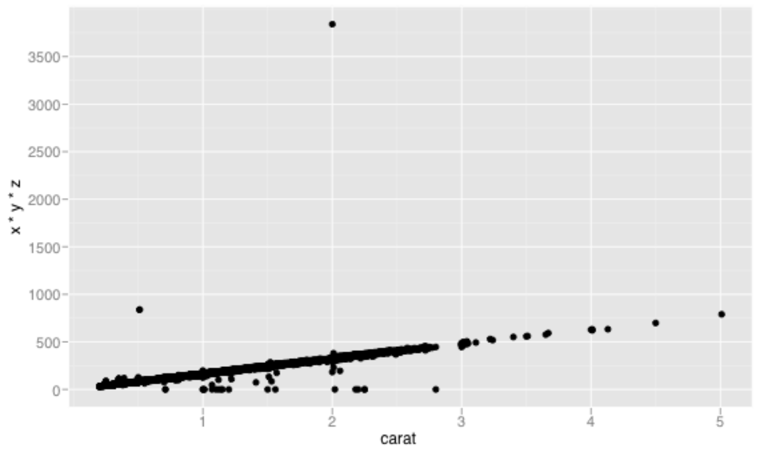


You are not limited to specifying names of existing vectors: you can use functions of them as well. Here we look at the relationship of log(price) to log(weight), and the relationship between weight and volume ($x * y * z$).

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



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



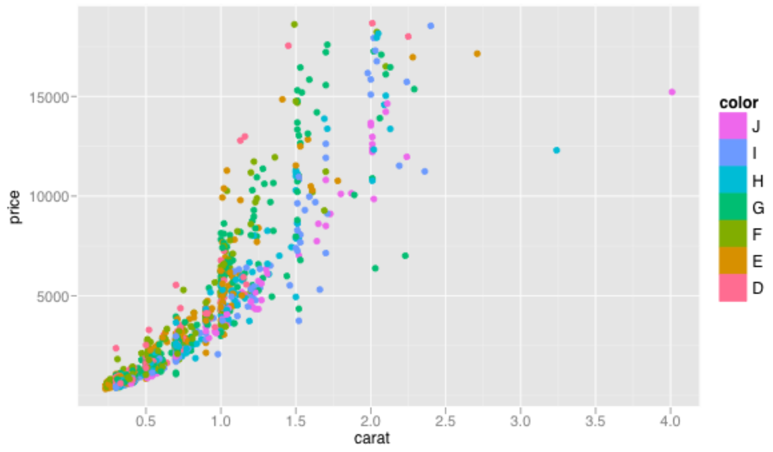
These plots illustrate some unusual features of this dataset. There seems to be a narrow band of prices for which there are no diamonds, and there are some diamonds with very unusual volumes for their weight. We will investigate these relationships more as we continue.

2.4 Aesthetic attributes

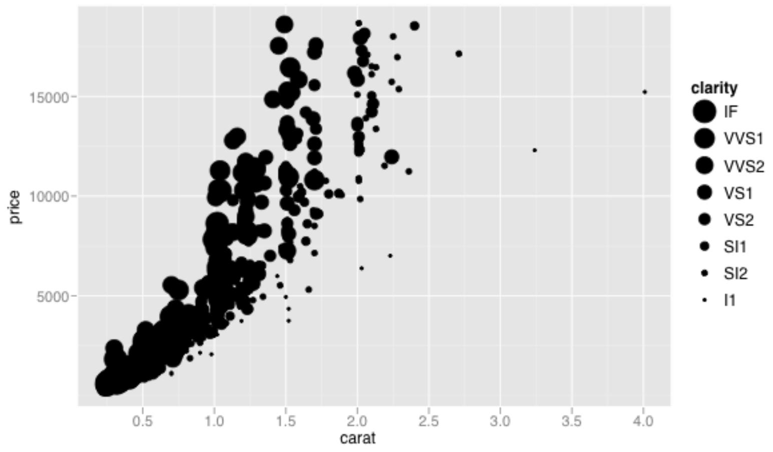
The first big difference with using `qplot` compared to `plot` comes when you want to assign colours—or sizes or shapes—to the points on your plot. With `plot`, it's your responsibility to change your data (eg. "apples", "bananas", "pears") into something that `plot` knows how to use (eg. "red", "yellow", "green"). `qplot` will do this for you automatically, and will automatically provide a legend to make it easier to look up the actual 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 colour, clarity and cut of the diamonds.

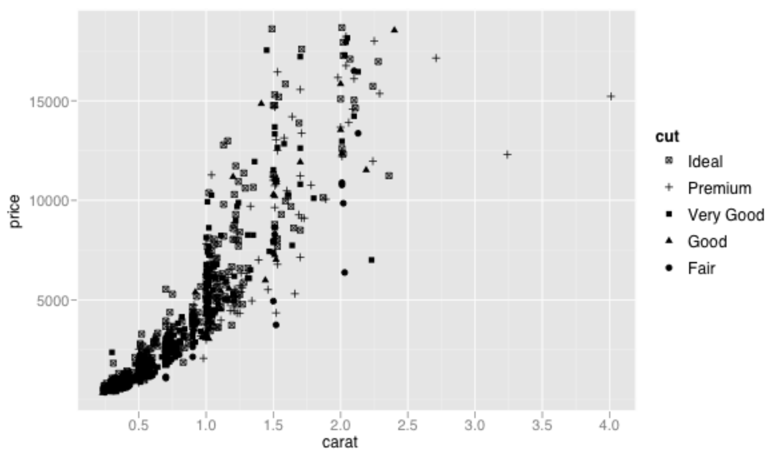
```
> dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
> qplot(carat, price, data = dsmall, colour = color)
```



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



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



Colour, size and shape are all examples of aesthetic attributes. An aesthetic attribute is some property that affects how the observations are displayed. For every aesthetic attribute, there is some function, called a scale, which maps data values to valid values for that aesthetic. It is this scale that controls how the points appear. For example, in the above plots, the colour scale maps J to purple and F to green. We will learn how to configure this in Chapter X.

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. You can always convert a continuous variable to a categorical one using the `chop` function. The amount of data also makes a difference: size works poorly when you have a lot of data, because you can't distinguish the individual points. These issues are discussed more in chapter X.

2.5 Plot geoms

`qplot` is not limited to just producing scatterplots. In fact, it can produce almost any type of plots, by varying the **geom** used. 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, categorised by whether they are useful for 1D or 2D data.

The following geoms enable you to investigate 2D 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.
- `geom="quantiles"` displays conditional density estimates. You can think of this as a extension of boxplots to deal with the case of a continuous conditioning variable.
- `geom="density2d"` adds contours of a 2d density estimate. This is very useful when you have a lot of overplotting.

- `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.
- `geom="boxplot"` produces a box and whisker plot to summarise the distribution of a set of points.

These geoms are useful for exploring 1d distributions:

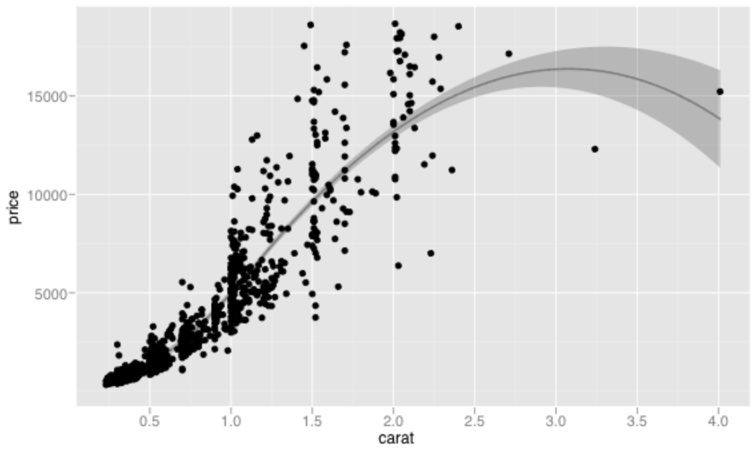
- `geom="histogram"` draws a histogram of the x variable (continuous or categorical)
- `geom="density"` creates a density plot for the x variable (continuous only)

Other types of plots for dealing with categorical data (bar charts, mosaic plots, spineplots, spinograms) are dealt with in chapter X, using the special categorical plotting function `catplot`.

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"))
```



Notice that we have combined multiple geoms by supplying a vector of geom names to `qplot`. The geoms will be overlaid in the order that you specified them.

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`. You can modify the wiggliness of the line by varying the span between 0, exceeding wiggly, and 1, not so wiggly. 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.

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

```
> qplot(carat, price, data = dsmall, geom = c("smooth", "point"),
+     span = 0.2)
```

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

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

  ```
  > library(splines)
  > qplot(carat, price, data = dsmall, geom = c("smooth", "point"),
  +     method = "lm", formula = y ~ ns(x, 3))
  ```

- `method="rlm"` works the same as `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, you should use the formula `y ~ s(x, bs="cr")`

  ```
  > library(mgcv)
  > qplot(carat, price, data = dsmall, geom = c("smooth", "point"),
  +     method = "gam", formula = y ~ s(x))
  ```

  ```
  > qplot(carat, price, data = diamonds, geom = c("smooth", "point"),
  +     method = "gam", formula = y ~ s(x, bs = "cr"))
  ```

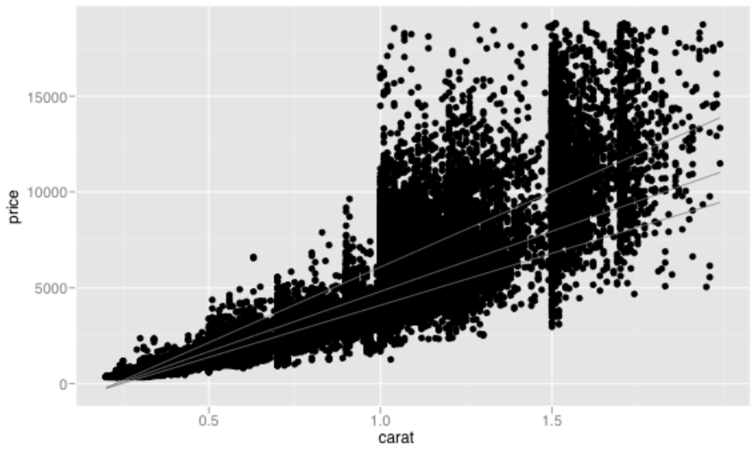
By default, the standard errors are shown with a grey band around the smoother. If you want to turn them off, use `se=FALSE`.

2.5.2 Quantiles

A smoother displays a smoothed conditional mean. It's often useful to see a smooth estimate of other summaries of the distribution, for example the upper and lower quartiles to show the spread of the data. We can do this with quantile regression (Koenker, 2005), which is basically a process to estimate smoothed conditional quantiles. The types of smoothers we can use is much more limited compared to `geom="smooth"`, but we can learn more about the conditional distribution.

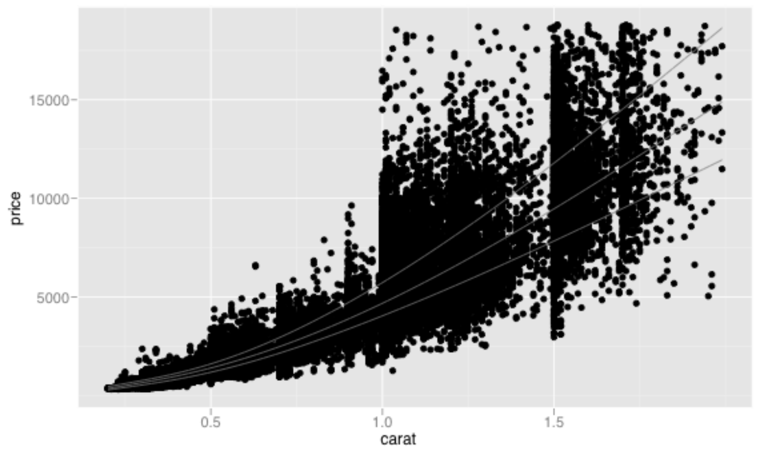
For this example, we're going to zoom into a small range of diamond sizes so we can look at the distribution more closely.

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

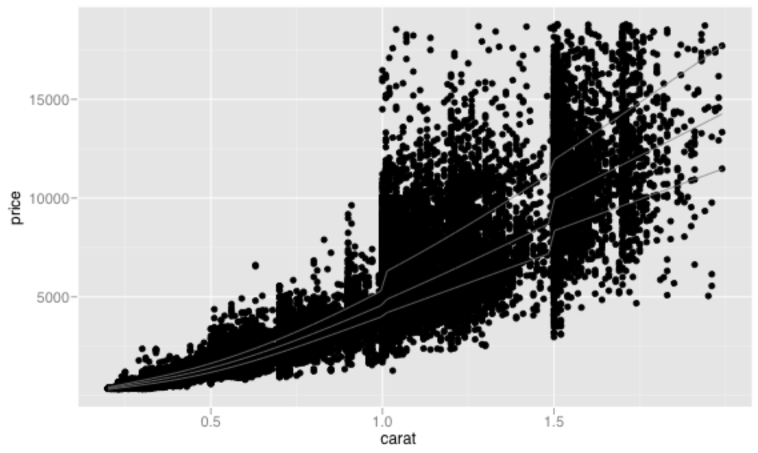


By default, the relationship between x and y is assumed to be linear, but we can adjust this using the `formula` argument, in the same way that we could for smoothers. In this example we'll try using a natural spline, and then manually adding some change points where we can see an obvious jump on the graph.

```
> qplot(carat, price, data = dlittle, geom = c("point", "quantile"),
+     formula = y ~ ns(x, 3))
```

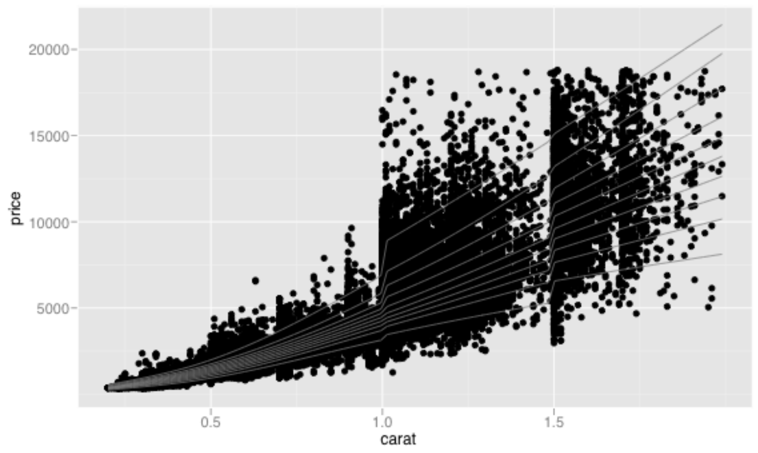


```
> qplot(carat, price, data = dlittle, geom = c("point", "quantile"),
+     formula = y ~ ns(x, 3) + I(x > 1) + I(x > 1.5))
```



You can also adjust the quantiles displayed with the `quantiles` argument:

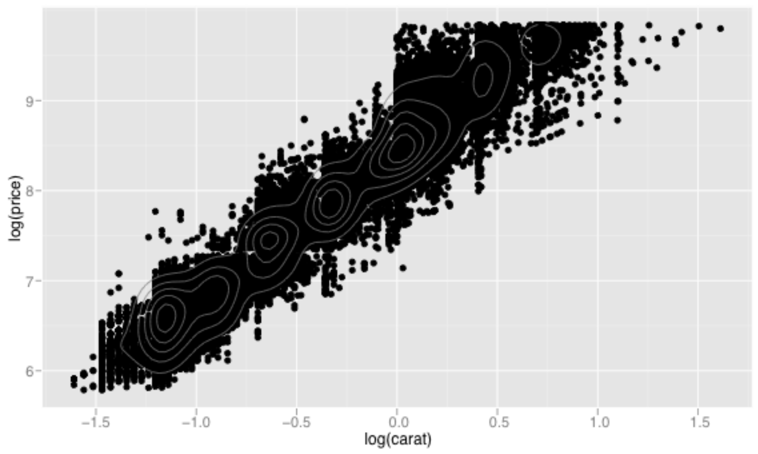
```
> qplot(carat, price, data = dlittle, geom = c("point", "quantile"),
+     formula = y ~ ns(x, 3) + I(x > 1) + I(x > 1.5), quantiles = seq(0.05,
+         0.95, 0.1))
```



2.5.3 2d density contours

When there is a lot of over-plotting on a plot, it is very hard to judge the relative density of points. One way to get around this is to supplement the plot with contour lines from a 2d density estimate. This is shown below.

```
> qplot(log(carat), log(price), data = diamonds, geom = c("point",
+      "density2d"))
```



The density estimation is described in more detail in `?kde2d`.

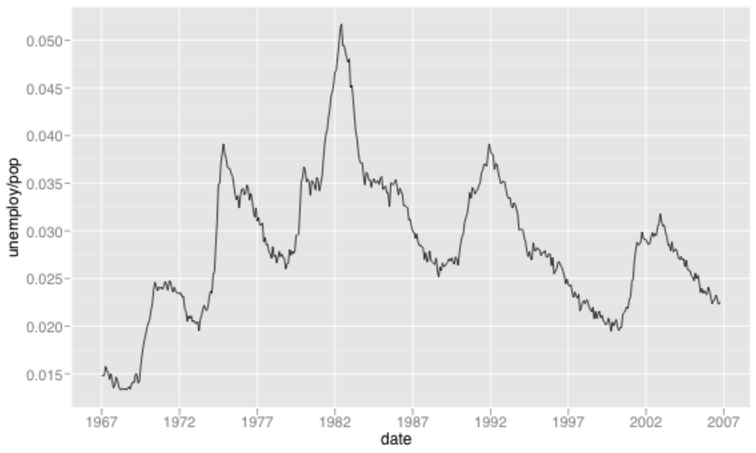
2.5.4 Time series with line and path plots

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.

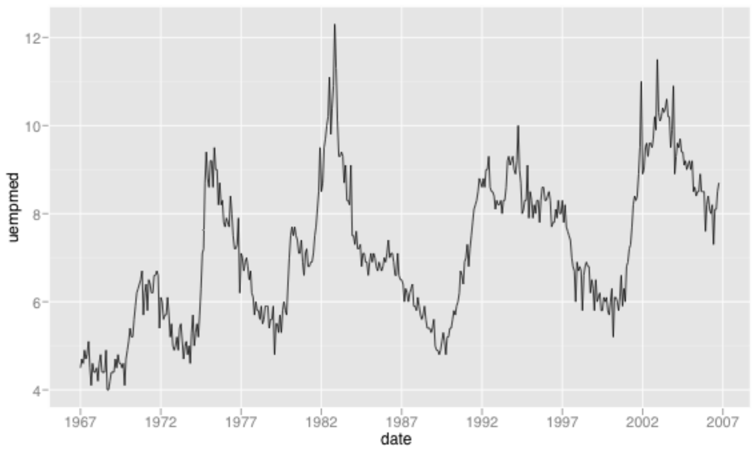
Because there isn't a time variable in the diamonds data, we will use another dataset, `economics`, which contains some economic data on the US measured over the last 40 years.

Let's start with a plot of unemployment over time, first as percent of people unemployed, and second as the median number of weeks unemployed:

```
> qplot(date, unemploy/pop, data = economics, geom = "line")
```



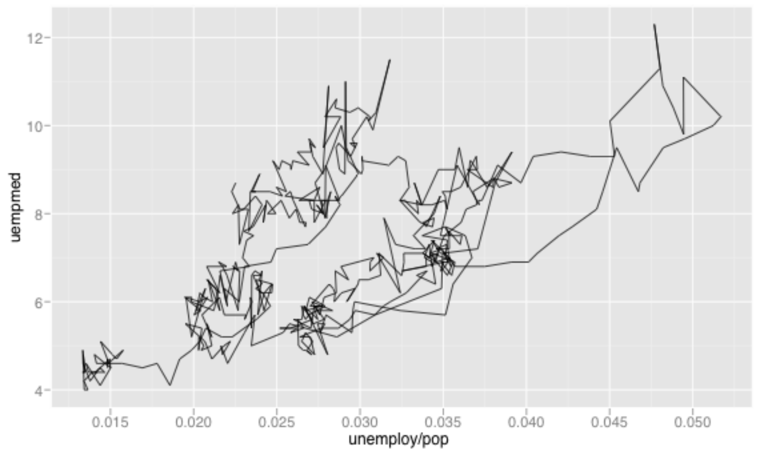
```
> qplot(date, uempmed, data = economics, geom = "line")
```



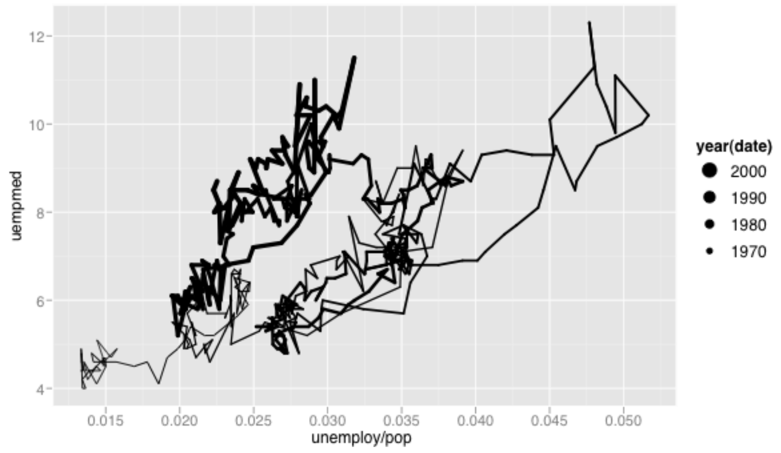
These plots show the behaviour of percent unemployed, and length of unemployment individually, but it is not so easy to see the joint pattern. Each time point occupies a point on the 2d grid of length and number, and if we joint up each point to its neighbours we get a 2d trajectory.

This can be illustrated with a path plot. Below we plot number of unemployed vs length of unemployment and then join the individual observations with a path to show the pattern over time. To make it more obvious in which direction time flows, we can use the `size` aesthetic, as in the second plot. We can see that number of unemployed and length of unemployment seems highly correlated, although in recent years the length of unemployment has been increasing relative to the total number of unemployed.

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



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

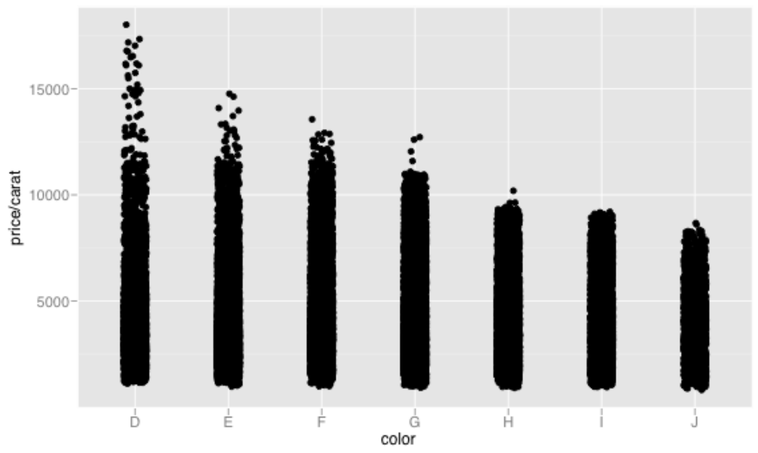


2.5.5 Boxplots and jittered points

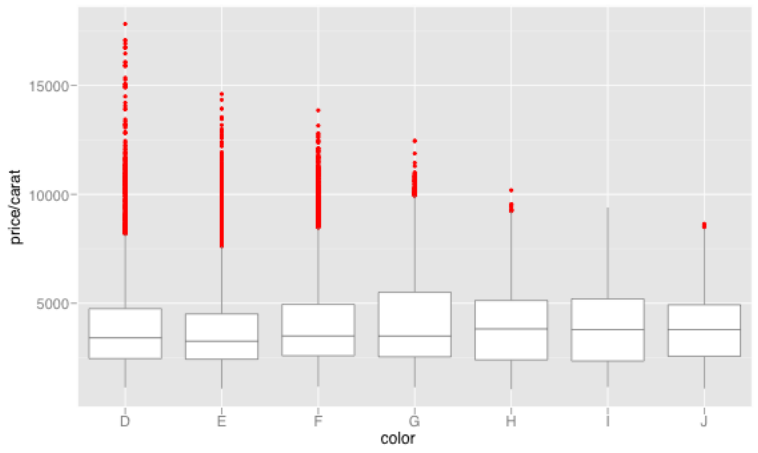
If you have one categorical variable, and one or more continuous variables, you will probably be interested to know how the values of the continuous variables vary with the categorical. Box plots and jittered points offer to ways to do this.

This example looks at how the price per carat varies with colour of the diamond.

```
> qplot(color, price/carat, data = diamonds, geom = "jitter")
```



```
> qplot(color, price/carat, data = diamonds, geom = "boxplot")
```



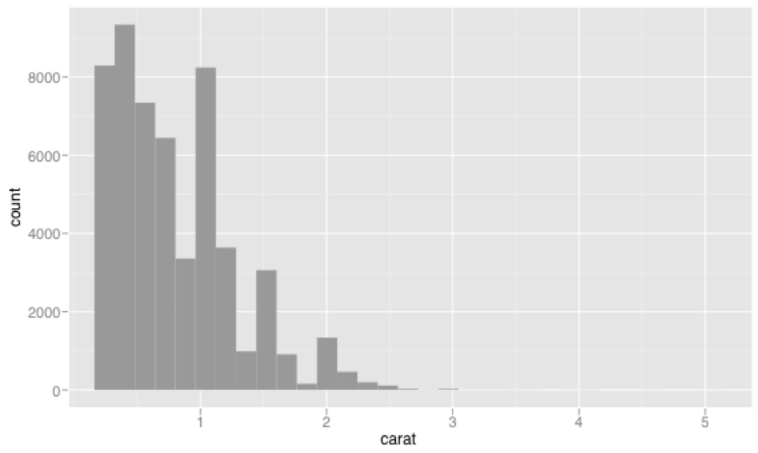
For jittered points, you have the same control over aesthetics as you do with a normal scatterplot: `size`, `colour`, `shape`. The options for boxplots are more limited (and it is hard to imagine when they would be useful), you can control only the outline colour (`colour`) and the internal fill `fill`.

Another way to look at conditional distributions is to plot a separate histogram or density plot for each value of the categorical variable.

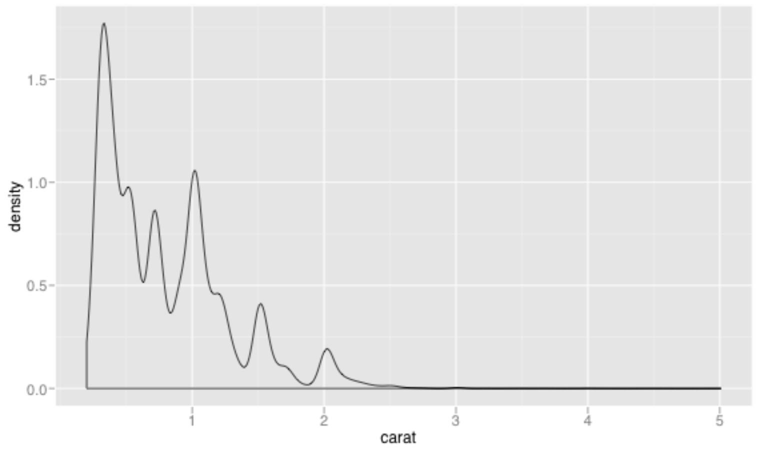
2.5.6 Histogram and density plots

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). The next example shows the distribution of carats with a histogram and a density plot.

```
> qplot(carat, data = diamonds, geom = "histogram")
```



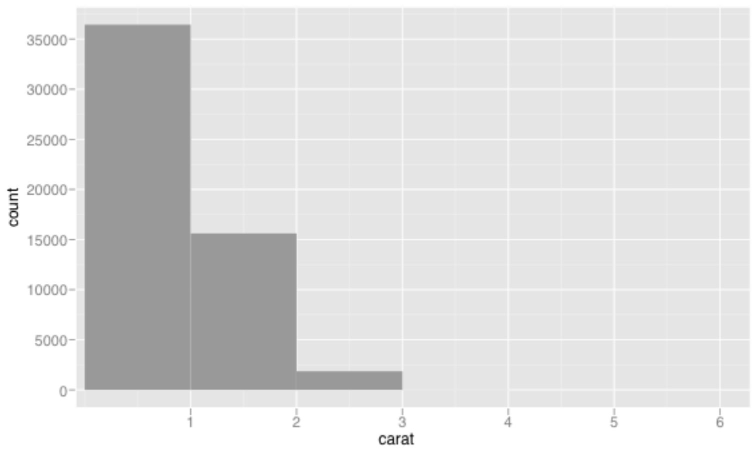
```
> qplot(carat, data = diamonds, geom = "density")
```



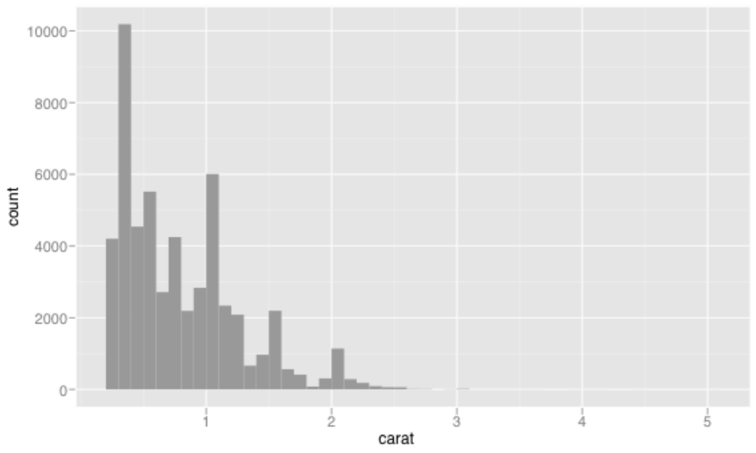
You can control the amount of smoothing using the `binwidth` argument for the histogram, which specifies the bin size to use. You can also specify the break points explicitly, using the `breaks` argument. For the density plot, you can use `adjust` argument, which adjusts the bandwidth of the smoother (high values of `adjust` produce smoother plots). It is **very important** to experiment with the level of smoothing. With a histogram you should try many bin widths before you decide on the one (or two, or three) which best describe the data.

For this example, we need to use a rather small binwidth before we can see the full story.

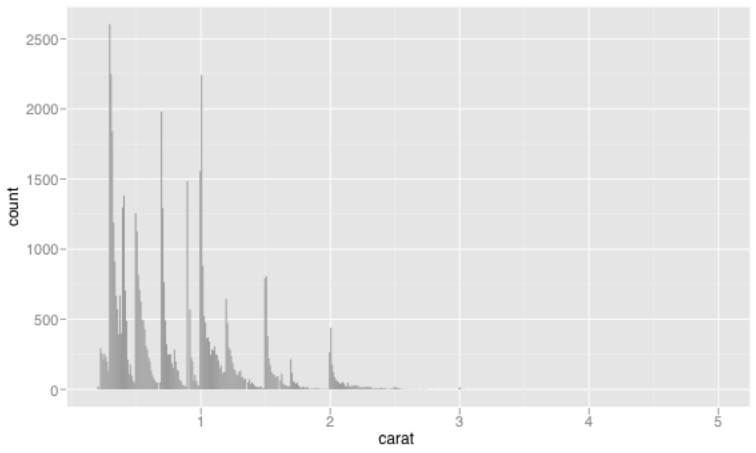
```
> qplot(carat, data = diamonds, geom = "histogram", binwidth = 1)
```



```
> qplot(carat, data = diamonds, geom = "histogram", binwidth = 0.1)
```

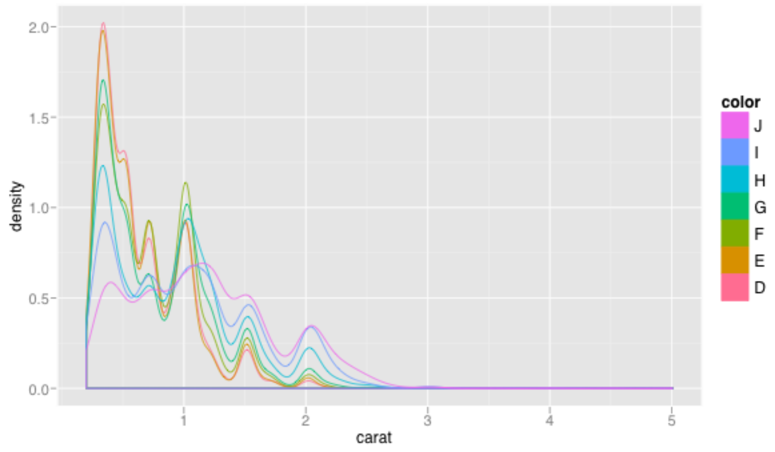


```
> qplot(carat, data = diamonds, geom = "histogram", binwidth = 0.01)
```



Density plots are useful in that they are easier to overlay, but are generally less flexible than histograms, and it is more difficult to understand exactly what a density plot is showing. (And we are **not** trying to estimate a density: we're trying to see what's going on with our data) If you want to compare the distributions of different subgroups, all you need to do is add an aesthetic mapping that differentiates the different groups, as follows:

```
> qplot(carat, data = diamonds, geom = "density", colour = color)
```



2.6 Plots for weighted data

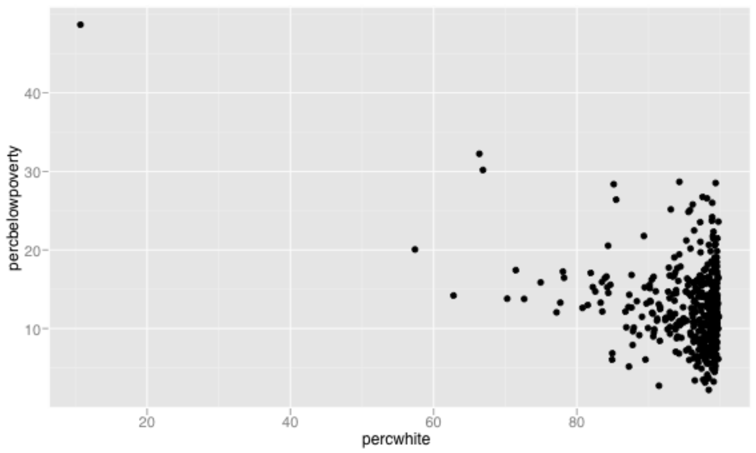
When you have aggregated data where each row in the dataset represents multiple observations, you need some way to take into account the weighting variable. Since there are no variables appropriate for weighting in the diamonds data, we will use some data collected on Midwest states in the 2000 US census. The data consists mainly of percentages (eg. percent white, percent below poverty line, percentage with college degree) and some information for each county (area, total population, population density).

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

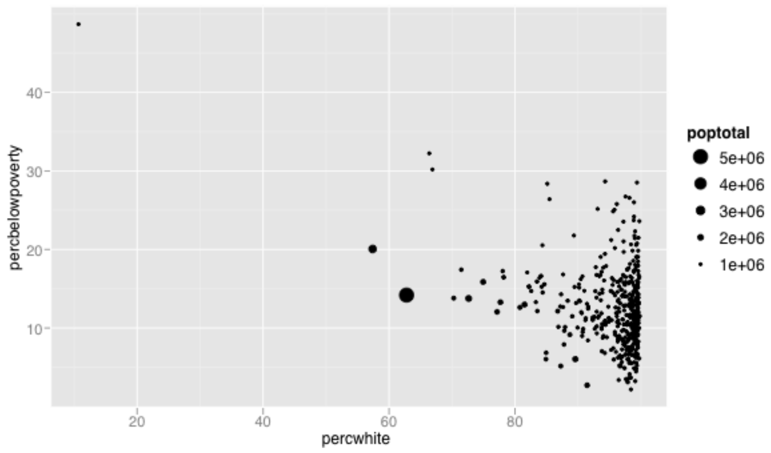
- nothing, to look at county numbers
- total population, to work with absolute numbers
- area, to investigate geographic effects

The choice of a weighting variable profoundly effects what we are looking at in the plot and the conclusions that we will draw. There are two aesthetic attributes that can be used to adjust for weights. Firstly, for simple geoms like lines and points, you can make the size of the grob proportional to the number of points, using the `size` aesthetic, as follows:

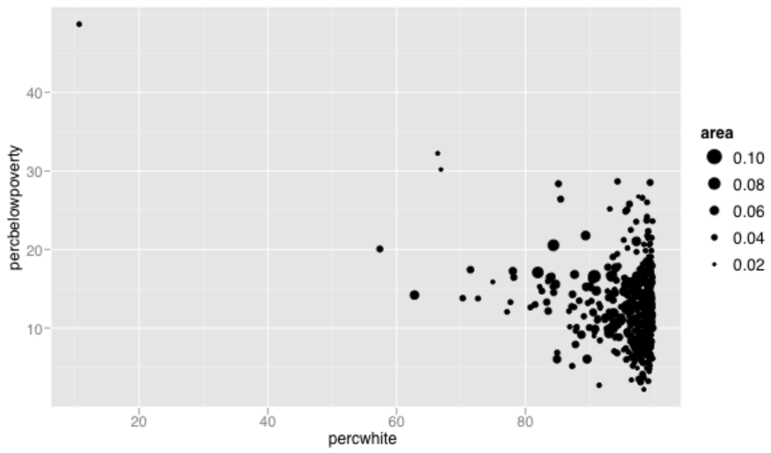
```
> midwest <- read.csv("~/Documents/graphics/weighted/midwest.csv")
> qplot(percwhite, percbelowpoverty, data = midwest)
```



```
> qplot(percwhite, percbelowpoverty, data = midwest, size = poptotal)
```



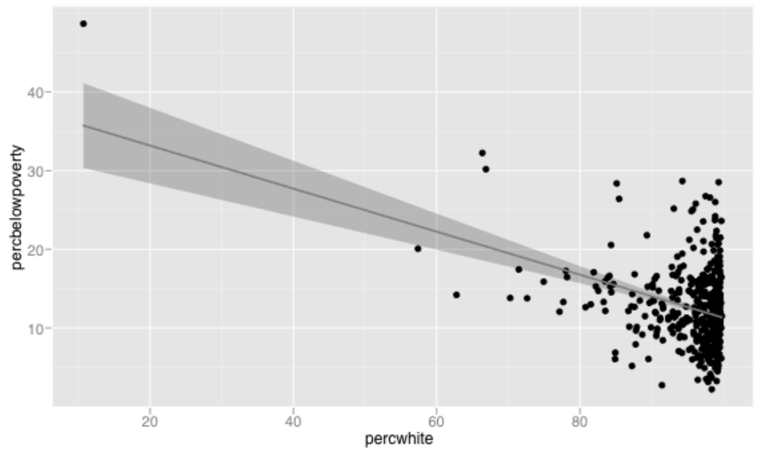
```
> qplot(percwhite, percbelowpoverty, data = midwest, size = area)
```



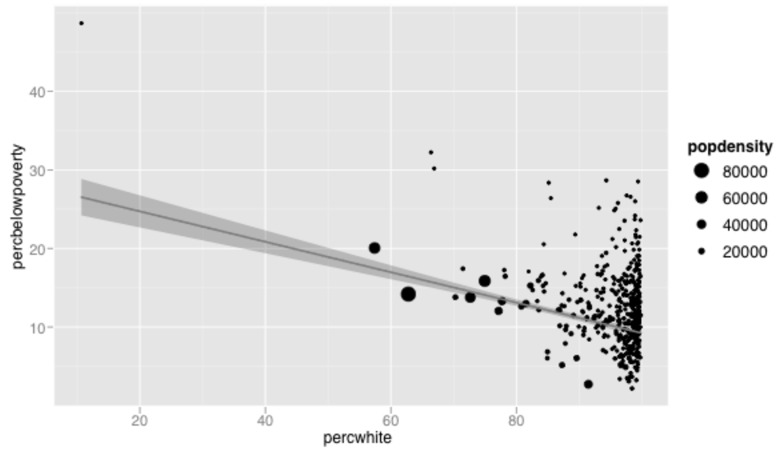
For more complicated grobs which involve some statistical transformation, we specify weights with the `weight` aesthetic. These weights will be passed on to the statistical summary function. Weights are supported for every case where it makes sense: smoothers, quantile regressions, box plots, histograms, and density plots. You can't see this weighting variable directly, and it doesn't produce a legend, but it will change the results of the statistical summary.

The following example shows how weighting by population density effects the relationship between percent white and percent below the poverty line.

```
> qplot(percwhite, percbelowpoverty, data = midwest, geom = c("point",
+     "smooth"), method = lm)
```

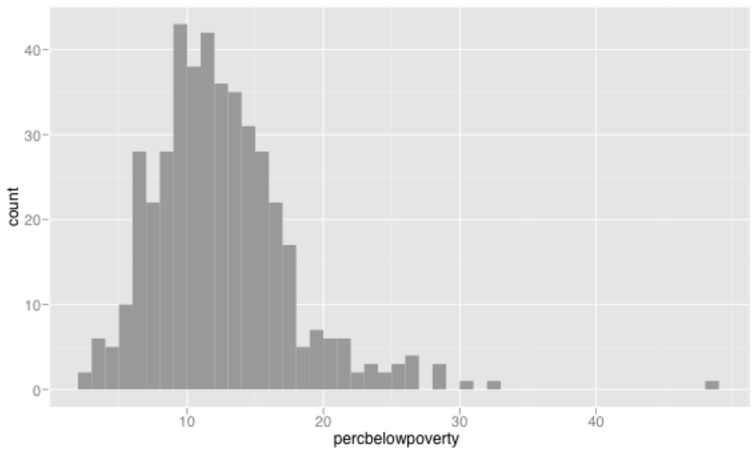


```
> qplot(percwhite, percbelowpoverty, data = midwest, size = popdensity,
+     weight = popdensity, geom = c("point", "smooth"), method = lm)
```

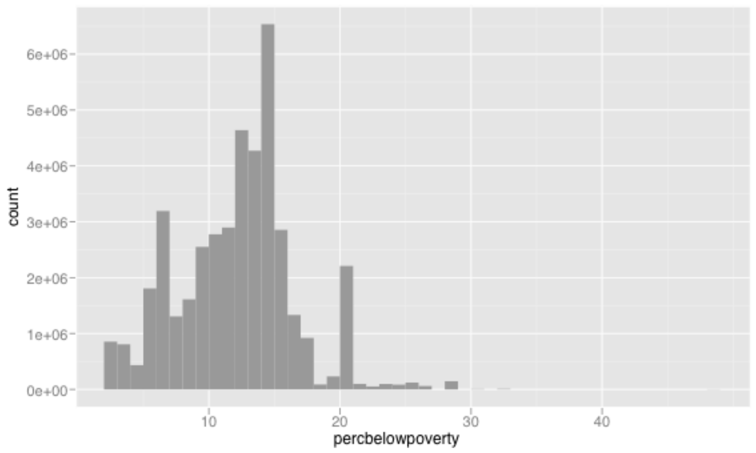


When we weight a histogram or density plot by total population, we change from looking at the distribution of the number of counties, to the distribution of the number of people. This example shows the difference this makes for a histogram and density plot of the percentage below the poverty line.

```
> qplot(percbelowpoverty, data = midwest, geom = "histogram", binwidth = 1)
```



```
> qplot(percbelowpoverty, data = midwest, geom = "histogram", weight = poptotal,
+     binwidth = 1)
```

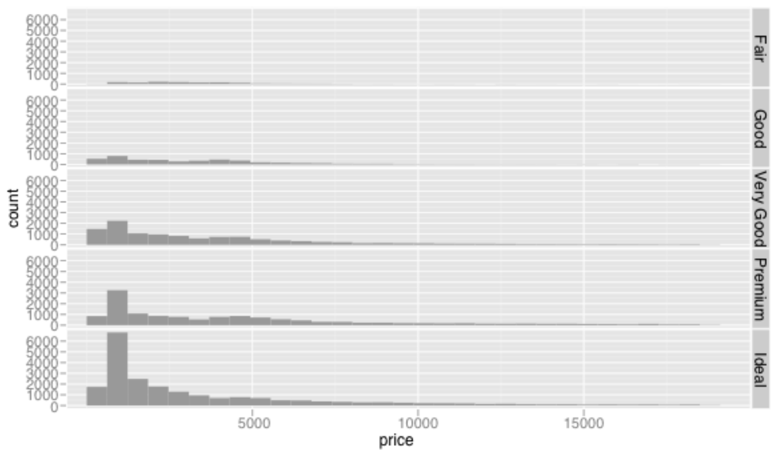


2.7 Facetting

rewrite!

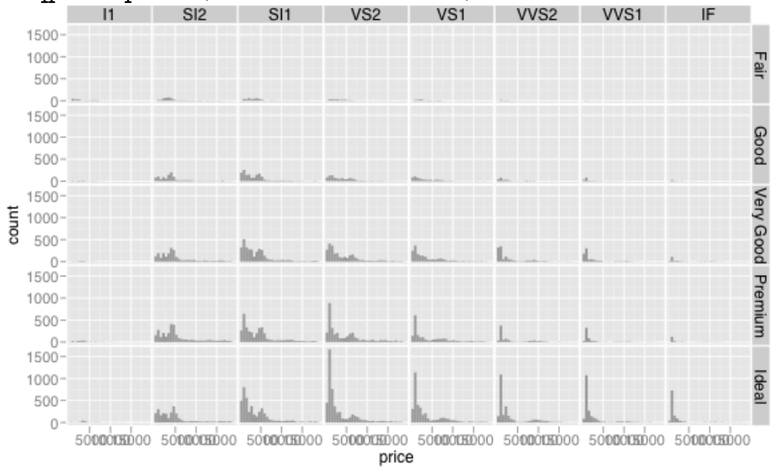
Facetting, also known as trellising or conditioning, allows you to display small multiples of subsets of your data. This example displays a histogram of price for each value of cut:

```
> qplot(price, data = diamonds, facets = cut ~ ., geom = "histogram")
```



Each small multiple is called a facet, and contains the same plot for a different subset of the data. The grid is specified with a facetting formula which looks like $row\_var \sim col\_var$. You can specify as many row and column variables as you like, but in most cases more than one or two variables will produce a plot so large that it is difficult to see on screen. If you want to facet on columns, or rows, not both, you can use `.` as a place holder. For example, $row\_var \sim .$ will facet by rows with a single variable.

```
> qplot(price, data = diamonds, facets = cut ~ clarity, geom = "histogram")
```



Facetting is used to investigate conditional relationships, e.g. conditional on sex, what is the relationship between amount of smoking and lung cancer. Facetting can also be useful for creating tables of graphics. For some examples of this, and more ways to use facetting, see chapter XXX.

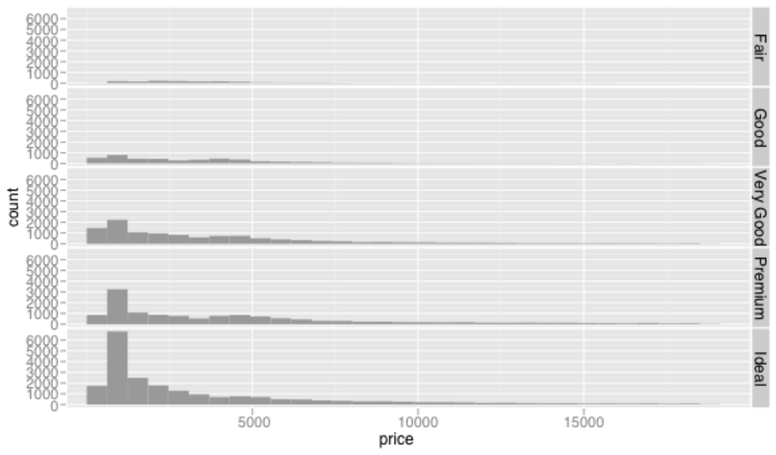
2.7.1 Margins

Facetting a plot is like creating a contingency table. In contingency tables it is often useful to display marginal totals (totals over a row or column) as well as the individual cells. It is also useful to be able to do this with graphics. We can produce graphical margins using the the `margins` argument. This allows you to compare the conditional patterns with the marginal patterns.

You can either specify that all margins should be displayed, using `margins = TRUE`, or by listing the names of the variables that you want margins for, `margins = c("sex","age")`. You can also use `"grand_row"` or `"grand_col"` to produce grand row and grand column margins respectively.

This example shows how the margins appear. In the first plot, there are no margins, and we only see conditional plots. In the second example, we see margins over columns, but not rows, and in the final example we see all possible margins. The facet in the lower right corner displays all data points.

```
> qplot(price, data = diamonds, facets = cut ~ ., geom = "histogram")
```



Plots with many facets and margins may be more appropriate for printing, rather than on screen display, as the higher resolution allows you to compare many more subsets.

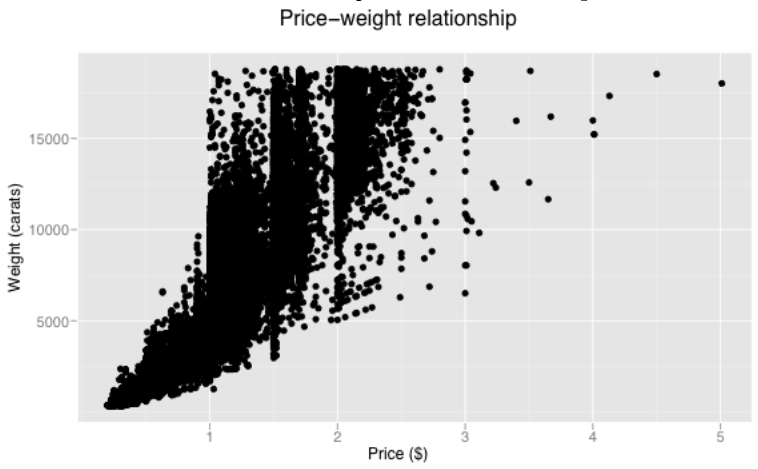
2.8 Other options

There are a few other options that `qplot` provides to control the output of your graphic. These all have the same effect as their `plot` equivalents:

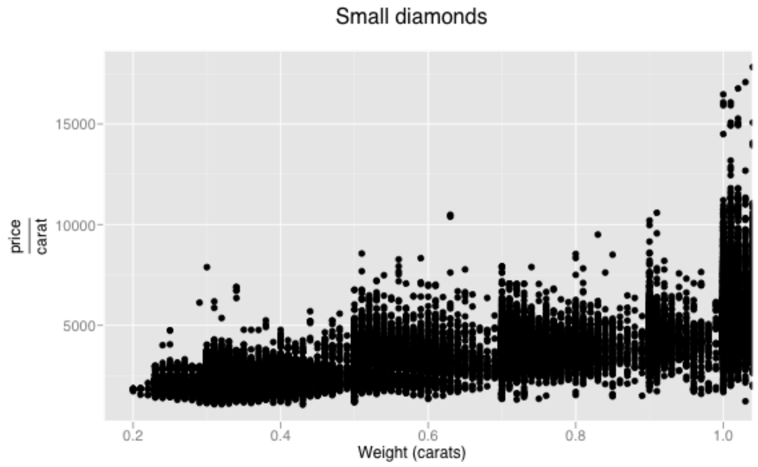
- `xlim`, `ylim`: set limits for the x- and y-axes, each a numeric vector of length two, e.g. `xlim=c(0, 20)` or `ylim=c(-0.9, -0.5)`.
- `log`: a character vector indicating which (if any) axes should be logged. For example, `log="x"` will log the x-axis, `log="xy"` will log both.
- `main`: main title for the plot, displayed in large text at the top-centre 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.

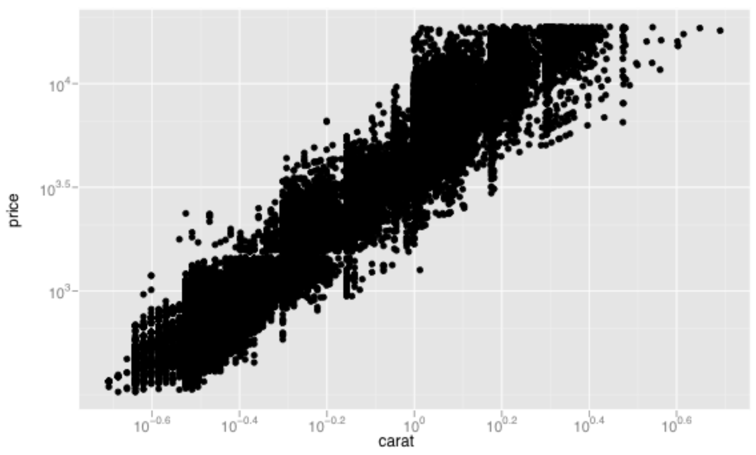
```
> qplot(carat, price, data = diamonds, xlab = "Price ($)", ylab = "Weight (carats)",
+     main = "Price-weight relationship")
```



```
> qplot(carat, price/carat, data = diamonds, ylab = expression(frac(price,
+     carat)), xlab = "Weight (carats)", main = "Small diamonds",
+     xlim = c(0.2, 1))
```



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


price
10^4
$10^{3.5}$
10^3
$10^{-0.6}$
$10^{-0.4}$
$10^{-0.2}$
10^0
$10^{0.2}$
$10^{0.4}$
$10^{0.6}$
carat

3 Understanding 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 write functions that operate on plots.

This chapter describes of the grammar of graphics, and how the grammar of ggplot differs from the grammar of (Wilkinson, 2005). In this chapter I will refer to the grammar of ggplot as my grammar, and The Grammar of Graphics as Wilkinson's grammar, and for the remainder of the book it may be assumed that whenever I refer to the grammar, I refer to my grammar.

In this chapter I will present two ways to think about the grammar: as an answer to a question, and as a pipeline of transformations. You will also learn the R syntax used to create and combine the components to form an actual plot. This chapter is fairly theoretical, so you won't find too many examples of drawing plots. However, a good understanding of how everything fits together will help you in creating your own plots. The following chapters, 3-7, describe the different components in depth and provide many practical examples.

3.2 What is a plot?

One way to think about the grammar of graphics is as a question: what is a plot? The grammar answers this by describing a plot as a collection of independent components, each describing an independent part of the plot. There are three basic things we need for a plot: one or more layers, scales to map variables from data space to visual space, and a coordinate system. These are described below.

- One or more layers. A layer is composed of data and a description of which data variables should be mapped to which aesthetic properties, a geometric object, and a statistical transformation:
  - Data is obviously the most important part, and it is what you provide. This is what you are displaying visually to aid communication or analysis. You also need to describe how variables in the dataset are mapped to visual properties. For example, in Figure 2.X we mapped diamond price to y position, carat to y position and colour to colour.
  - **Geoms**, short for geometric objects, 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.
  - **Stats**, or statistical transformations, reduce or augment the data in a statistical manner. For example, a useful stat is the smoother, which shows the mean of y, conditional on

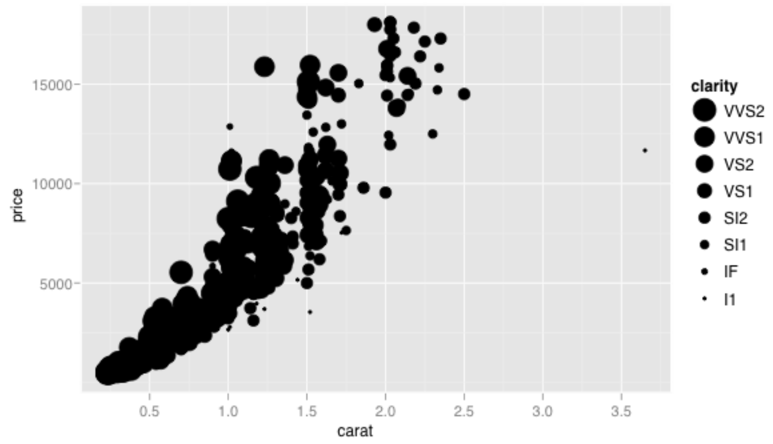
x. Another common stat is the binner, which bins data in to bins. Every geom has a default statistic, and every statistic a default geom. For example, the bin statistic has defaults to using the bar geom to produce a histogram.

- A scale for all the aesthetic properties. **Scales** control the mapping from data attributes to aesthetic attributes. They also provide an inverse mapping in the form of a guide, an axis or legend, which facilitates reading the final graphic. Aesthetic attributes are things like position, size, colour—anything that you can perceive. The function that maps data to aesthetic attributes is a scale. It takes values in data space (continuous or categorical) and maps them into an aesthetic space (eg. colour, size, shape). A scale also provides guides to convert back from the aesthetic attribute to the original data. Guides are either axes (for position) or legends (for everything else)
- A coordinate system. A **coord**, or coordinate systems, maps the position of objects on to the plane of the plot. Typically we will use the cartesian coordinate system, but sometimes others are useful.

There is also another thing that turns out to be sufficiently useful that we should include it in our general framework: facetting (also known as conditioned or trellis plots). This allows us to easily create small multiples of different subsets of an entire dataset. This is a powerful tool when investigating whether patterns hold across all conditions.

To clarify the role of each of these elements, here are a couple of simple examples, taken from Chapter 2. First, we'll start with a bubble chart of price by carat, with size proportional to the clarity of the diamond.

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



This plot has one layer, three scales and a cartesian coordinate system. To make these parts more clear, we can instead create the plot using these commands:

```
ggplot() +
layer(
  data = diamonds, aes(x = carat, y = price, size = clarity),
  geom = "point", stat = "identity"
)
```

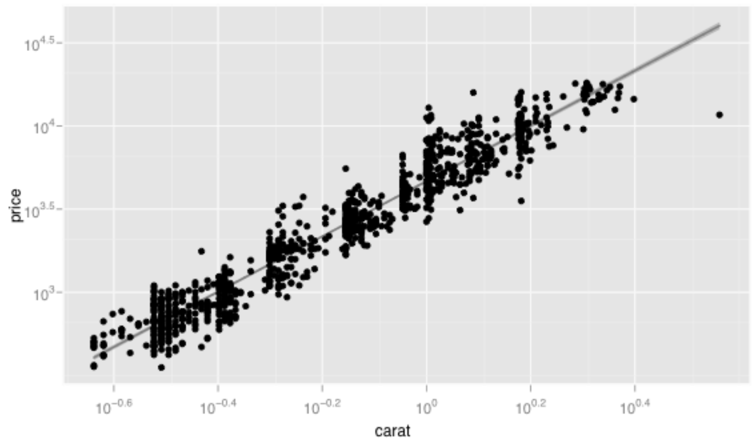
This makes the single layer obvious, it uses a point geom and the identity transformation, but where are the definitions of the scales and the coordinate system? By default, ggplot automatically adds sensible scales and the default coordinate system, however, we can be more explicit and add them ourselves:

```
ggplot() +
layer(
  data = diamonds, mapping = aes(x = carat, y = price, colour = clarity),
  geom = "point", stat = "identity"
) +
scale_size() +
scale_y_continuous() +
scale_x_continuous() +
coord_cartesian()
```

Compared to the initial qplot example, this is extremely verbose, but perfectly explicit. We can clearly see the single layer, the three scales (x position, y position and size) and the coordinate system.

The next example includes multiple layers, and demonstrates the effect of modifying the scale parameters. This is another example from chapter two: a scatterplot of price vs carat, with logged axes and a linear smooth layered on top. The qplot code is shown below.

```
> qplot(carat, price, data = dsmall, geom = c("smooth", "point"),
+     method = "lm", log = "xy")
```



Here we have two layers, two scales and the same cartesian coordinate system:

```
ggplot() +
layer(
  data = diamonds, mapping = aes(x = carat, y = price),
  geom = "point", stat = "identity"
) +
layer(
  data = diamonds, mapping = aes(x = carat, y = price),
  geom = "smooth", stat = "smooth", method = "lm"

) +
scale_y_log10() +
scale_x_log10() +
coord_cartesian()
```

There is some duplication in this example, which we can reduce by using plot defaults.

```
ggplot(data = diamonds, mapping = aes(x = carat, y = price)) +
layer(geom = "point", stat = "identity") +
layer(geom = "smooth", stat = "smooth", method = "lm") +
scale_y_log10() +
scale_x_log10() +
coord_cartesian()
```

3.2.1 Differences from Wilkinson's grammar

The breakdown into components is very similar Wilkinson's grammar, but there are some important differences. In Wilkinson's grammar, there is no notion of a layer, and there are few examples where multiple datasets are plotted on the same figure. There are also three components that are missing in my grammar: the algebra, the transformations, and the guides. These are not included in ggplot because they can performed using other functionality in R:

- The algebra describes how to reshape data for display. In R, this can be done instead with the reshape package (Wickham, 2005). This is part of my philosophy that data manipulation should be separate from display.
- The explicit transformation stage was dropped because variable transformations are already so easy in R: they do not need to be part of the grammar.
- In ggplot, guides (axes and legends) are largely drawn automatically. For example, in Wilkinson's grammar you explicitly specify that you want a legend, or how the axes should be annotated, but this is determined by the scale in ggplot. Annotative guides can be created with geoms if data dependent, or grid can be used to draw directly onto the plot.

Finally, I have renamed the element component to geom (after consultation with Lee Wilkinson) to make the name less generic.

3.3 Converting between grammar syntaxes

The biggest difference between Lee's grammar and mine is how the grammar is written; the syntax of the grammars are quite different. The Grammar of Graphics uses two specifications. A concise format is used to caption figures, and a more detailed xml format stored on disk. The following example of the concise format is adapted from Wilkinson (2005, Figure 1.5, page 13).

```
DATA: source("demographics")
DATA: longitude, latitude = map(source("World"))
TRANS: bd = max(birth - death, 0)
COORD: project.mercator()
ELEMENT: point(position(lon * lat), size(bf), color(color.red))
ELEMENT: polygon(position(longitude * latitude))
```

This is relatively simple to adapt to the syntax of ggplot:

- `ggplot()` is used to specify the default data and default aesthetic mappings. `aes` is short for aesthetic mapping and specifies which variables in the data should be mapped to which aesthetic attributes.
- Data is provided as standard R data.frames existing in the global environment; it does not need to be explicitly loaded. We also use a slightly different world data set, with columns lat and long. This lets us use the same aesthetic mappings for both datasets. Layers can override the default data and aesthetic mappings provided by the plot.
- We replace TRANS with an explicit transformation by R code.
- ELEMENTs are replaced with layers, which explicitly specify where the data comes from. Each geom has a default statistic which is used to transform the data prior to plotting. For the geoms in this example, the default statistic is the identity function. Fixed aesthetics (the colour red in this example) are supplied as additional arguments to the layer, rather than as special constants.
- The SCALE component has been omitted from this example (so that the defaults are used)In both the ggplot and GoG examples, scales are defined by default. In ggplot you can override the defaults by adding a scale object, e.g. `scale_colour` or `scale_size`
- COORD uses a slightly different format. In general, most of the components specifications in ggplot are slightly different to those in GoG, in order to be more familiar to R users.
- Each component is added together with + to create the final plot

This gives us:

```
demographics <- transform(demographics, bd = max(birth - death, 0))

ggplot(data = demographic, mapping = aes(x = lon, y = lat)) +
layer(geom = "point", mapping = aes(size = bd), colour="red") +
layer(geom = "polygon", data = world) +
coord_map(projection = "mercator")
```

4 Customising the display of ggplot graphics

In ggplot, the appearance and the structure of the plot are quite separate. This is different to base and lattice graphics in that you do not specify the appearance of the plot while you are creating it (defining its structure). 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. Instead, in ggplot you create the plot in one step, and then *after* it has been created you can edit every detail of the rendering.

The are two ways to do this: common options are exposed by the `ggopt` function, or you can use the grid package to modify the graphical output of ggplot. The few options provided by `ggopt` allow you to easily modify the most commonly tweaked parts of ggplot graphics, but for more control you can use the power of grid to delve into the dark depths of `ggplot` and tweak to your heart's content.

What is grid? It's the graphics engine that powers ggplot. It is responsible for drawing the graphic object onto the screen or other output device. It provides a system of viewports, which define regions on the plot and the coordinate systems inside. A useful reference is `http://www.stat.auckland.ac.nz/~paul/RGraphics/chapter5.pdf` a sample chapter from Paul Murrell's Grid Graphics.

Please remember that ggplot has been carefully designed to provide perceptually well-founded defaults, and you should think carefully about what you are doing before you make any big changes. Some particularly good references to consult are:

- Cleveland (1993, 1994); 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.

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.

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.

Option	Valid values	Details
background.fill	any colour	Background fill colour behind entire plot
background.colour	any colour	
grid.colour	any colour	Colour of grid lines within panel
grid.fill	any colour	Panel background colour
strip.gp	gpar object	Graphical parameters used for strip. Colour and fill
strip.text.gp	any colour	Graphical parameters used for strip text.
strip.text	a function	Function should accept two arguments, variable and
legend.position	left, right, top, bottom, none, c(x, y)	Position of legend. Can use numeric vector of length
aspect.ratio	a number or NULL	Aspect ratio of plot.

Table 4.1: ggplot options

Option	Description
fill, col	Background and foreground colours. You can supply a named colour (see `?colors`) or a h
lty	Line type: solid, dotted, dashed etc.
lwd	Width of line, in points.
fontfamily, fontface	Refer to R-news article
fontsize	Font size, in points.

Table 4.2: gpar specifications. See `?gpar` for more details

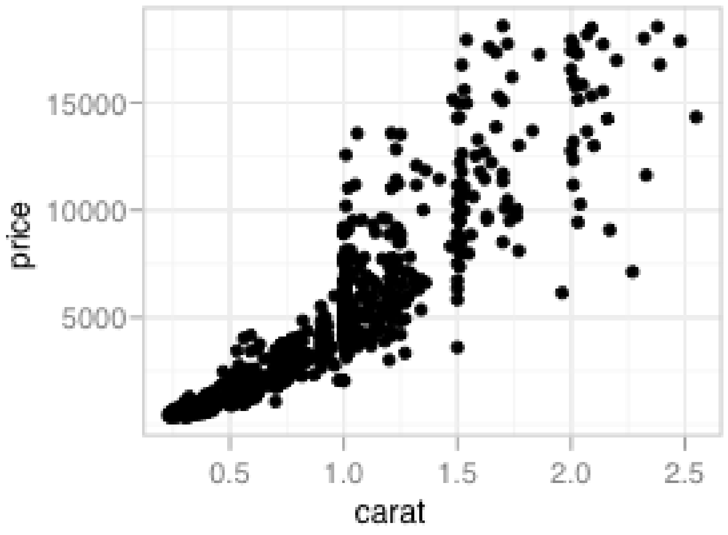
4.1 Options

ggplot provides convenient access to a select set of commonly used options, described in Table 4.1. These options are described in more detpth, with many examples, in the online documentation `?ggopt`. There are two ways to set options:

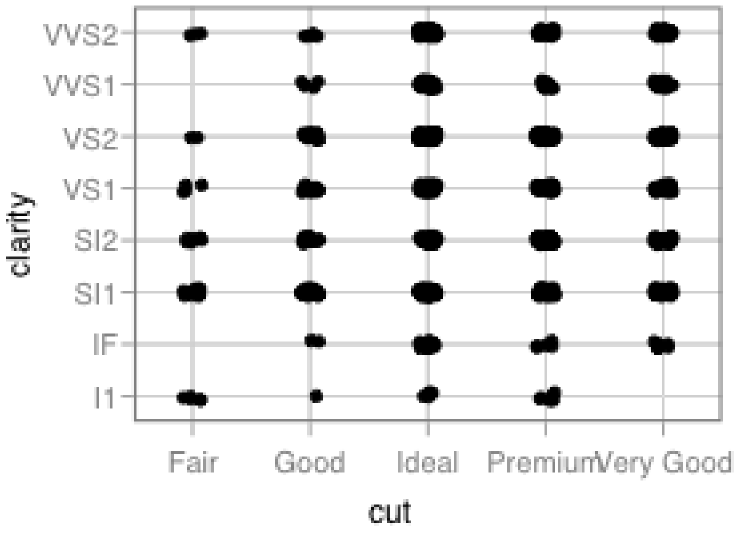
- Globally (for all plots), using `ggopt`. For example `ggopt(grid.fill = "white")`. Global options are applied when a plot is rendered, not when it is created. This lets you experiment with plot structure and appearance independently.
- Locally (for one plot), using $ or `update`. For example, `p$grid.fill <- "white"`, or `update(p, grid.fill = "white")`. The first modifies the object in place, while the second creates a modified copy. Local options override the more general global options.

The following example demonstrates some of the possibilities.

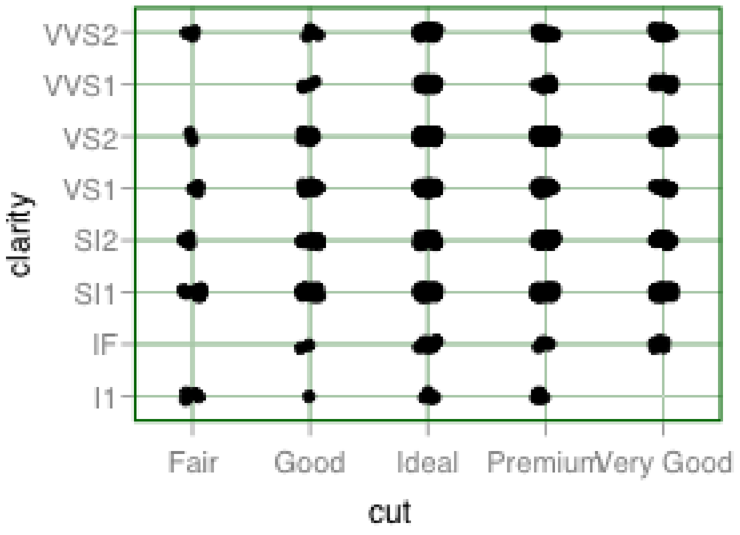
```
> old <- ggopt(grid.colour = "grey50", grid.fill = "white")
> qplot(carat, price, data = dsmall)
```



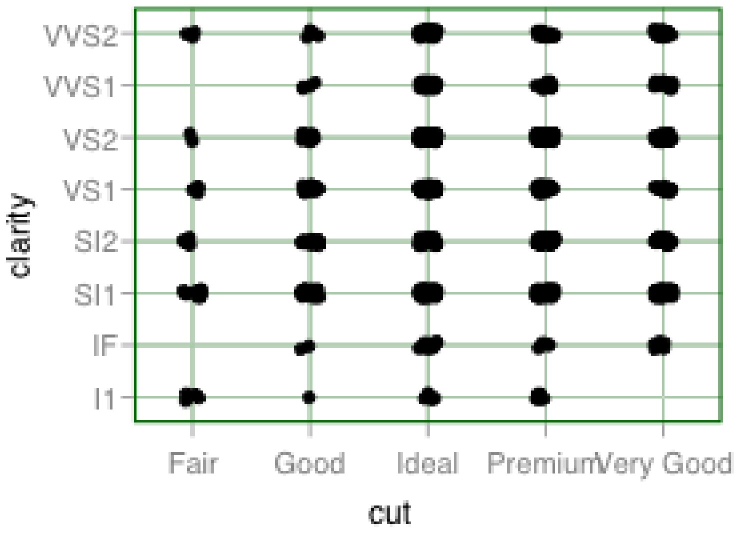
```
> (p <- qplot(cut, clarity, data = dsmall, geom = "jitter"))
```



```
> p$grid.colour <- "darkgreen"
> p
```



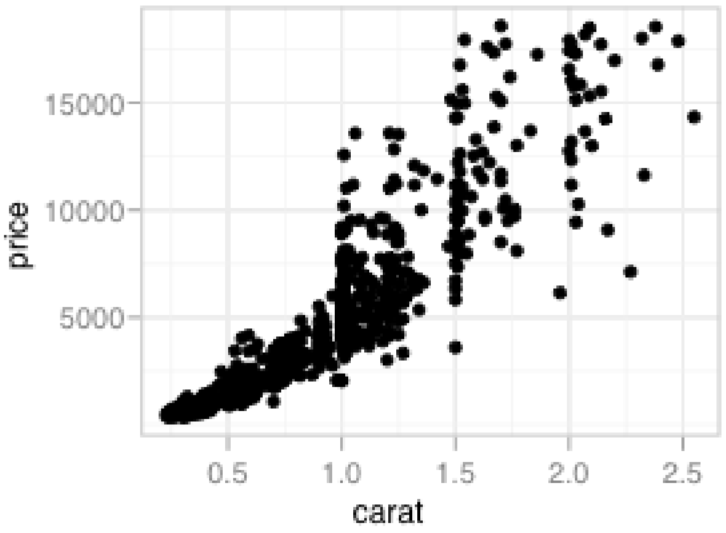
```
> ggtheme(old)
> p
```



4.1.1 Themes

Like lattice, it is also possible to create a theme which encapsulates multiple options. A theme is a very simple structure, just a list of multiple options, so it's easy to create your own. One theme is included by default: `theme_bw`, which sets up a white background with black grid lines. You can use a theme with `ggtheme(theme)`, or for a single plot `update(plot, theme)`.

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



4.2 Modifying the plot with grid

There are three principle ways to modify a plot:

- Edit existing objects on the plot
- Add annotations to the plot
- Arrange multiple plots on a single page

To annotate or edit a plot, you first need to figure out what you want to change, and what it is called. If you are annotating plots, you will need to know the name of the appropriate viewport. If you are editing plots, you will need to know the name of the appropriate grob. This following sections describe the structure of the plot and how to modify it

4.2.1 Editing existing objects on the plot

4.2.2 Grob names

To edit existing grobs, you need to know their names. 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 `current.grobTree()`. If you only want to see the ggplot name of the grob, `current.grobTree(only.name=TRUE)` will reduce a lot of the output. Here's an example after drawing the previous 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
```

Notice that the grobs are arranged in a hierarchical manner.

The most important components are:

- `guide`, the internal guides within a panel (background, and grid lines)
- `xaxis` and `yaxis`, the axes, containing `labels` and `ticks`.
- Axis labels and title: `xlabel`, `ylabel`, `title`.
- The geom displayed in the plot: `geom_point`.
- Another important component not shown in this example are `strips`: containing the `background` background fill and `label`.

4.2.3 Modifying grobs

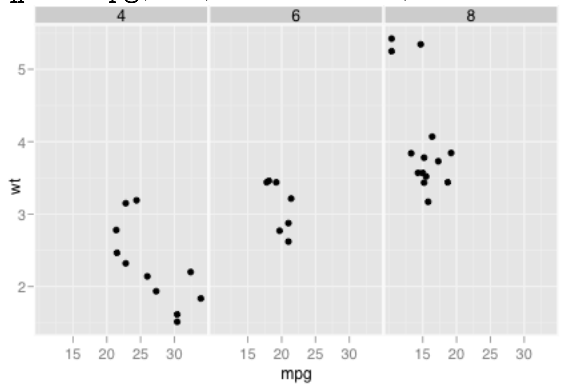
Most of the difficulty in modifying stuff on the plot is figuring out the name of the grob you want to modify, and once you have that you can use `grid.gedit`. `grid.gedit` does two things: it finds the grobs you want to modify, and then changes their appearance.

You describe which grobs to modify with a gPath. A gPath can either be a string specifying a single grob name, or a sequence of grob names 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.

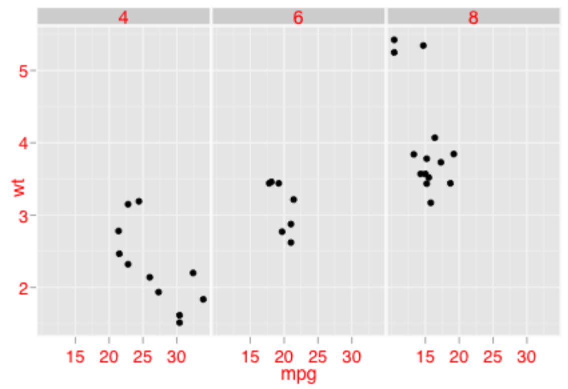
Modifying a grob requires some knowledge of the different parameters of the grob. This is where the second part of the grob name is useful, as it will tell you whether you are modifying a line, or a rect or a text grob. You can get more information by looking at the documentation for that grob, eg. `?grid.rect, ?grid.text, ?grid.lines` All grobs share a common set of graphical parameters described in Table 4.2. Note that grid graphic parameters (gpar) have the difference argument names to the usual ggplot names (cex instead of size, pch instead of shape, col instead of colour)

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

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

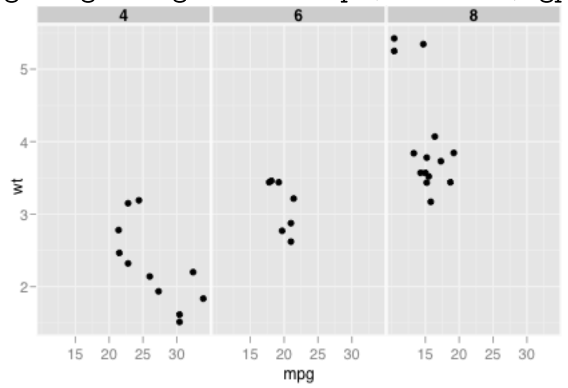


```
grid.gedit("label", gp=gpar(fontsize=14, col="red"))
```

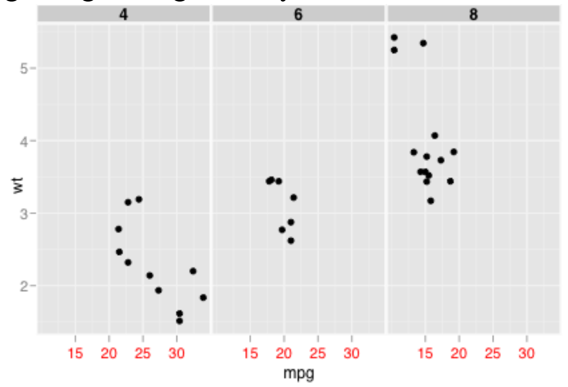


To edit just one type of label, we need to use the hierarchy of grobs and the gPath function:

```
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"))
```



4.2.4 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 facetting strips.

The structure of viewports will vary slightly from plot to plot, depending on the type of facetting. For a plot produced with `facet_grid`, the default, the viewports are described below. For other types of facetting, the details will vary slightly, and are described in the documentation for that facetting system.

The `layout` viewport contains the meat of the plot: strip labels, axes and facetting panels. The viewports are named according to their role and their position. A viewport name is made up of a prefix (listed below) which describes the contents of the viewport, and x and y position (counting from bottom left) separated by "_".

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

You can see all the viewports on the current plot by typing `current.vpTree(all=TRUE)`.

> Unfortunately there is a bug in grid which currently prevents all the viewports being accessed and displayed. You can get around this by printing only the bare necessities of the plot:
>
> ```
> p <- qplot(wt, mpg, data=mtcars, colour=cyl)
> print(p, pretty = FALSE)
> current.vpTree(all=TRUE)
> ```

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

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

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

4.2.5 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 facetting. 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.

Bibliography

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/%7Edcarr/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, 1993.

William Cleveland. *The Elements of Graphing Data.* Hobart Press, 1994.

William 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.

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

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

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

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, 2 edition, 2001.

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

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

Hadley Wickham. *reshape: Flexibly reshape data.*, 2005. URL `http://had.co.nz/reshape/`. R package version 0.7.1.

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