# Data Visualization Tutorial

## Introduction

Visualizing your data is hands down the most important thing you can learn to do. Please review the resources linked at the end of this document for additional learning resources.

There are two audiences in mind when creating data visualizations:

1. For your eyes only. These are quick and dirty plots, without annotation. Meant to be looked at once or twice.
2. To share with others. These need to completely stand on their own. Axes labels, titles, colors as needed, possibly captions.

You will see, and slowly learn, how to add these annotations and how to clean up your graphics to make them sharable. `ggplot2` already does a lot of this work for you.

We will also use the two most common methods used to create plots. 1) Base graphics, 2) the `ggplot2` package. We will not cover `lattice` graphics in this lab but they are worth looking into.

For **almost** every plot discussed we will create the plot using first base graphics, then using `ggplot2`. Each have their own advantages and disadvantges. If you have not done so already, go ahead and install the `ggplot2` package now.

## The Data

We will use a subset of the `diamonds` dataset that comes with the `ggplot2` package. This dataset contains the prices and other attributes of almost 54,000 diamonds. Review `?diamonds` to learn about the variables we will be using.

```
library(ggplot2)
data("diamonds")
set.seed(1410) # Make the sample reproducible
dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
```

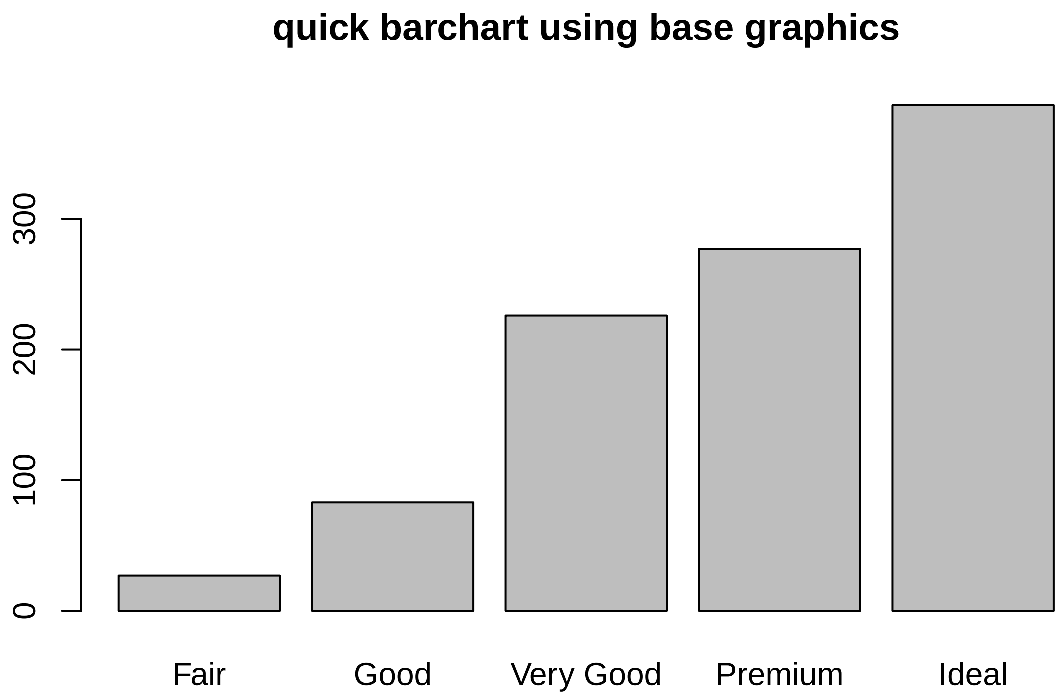
## One Categorical variable

Categorical variables are ones that cannot be measured (think like with a ruler). They describe characteristics of an observation. Here we will look at the cut, and clarity of diamonds.

### Barcharts / Barplots

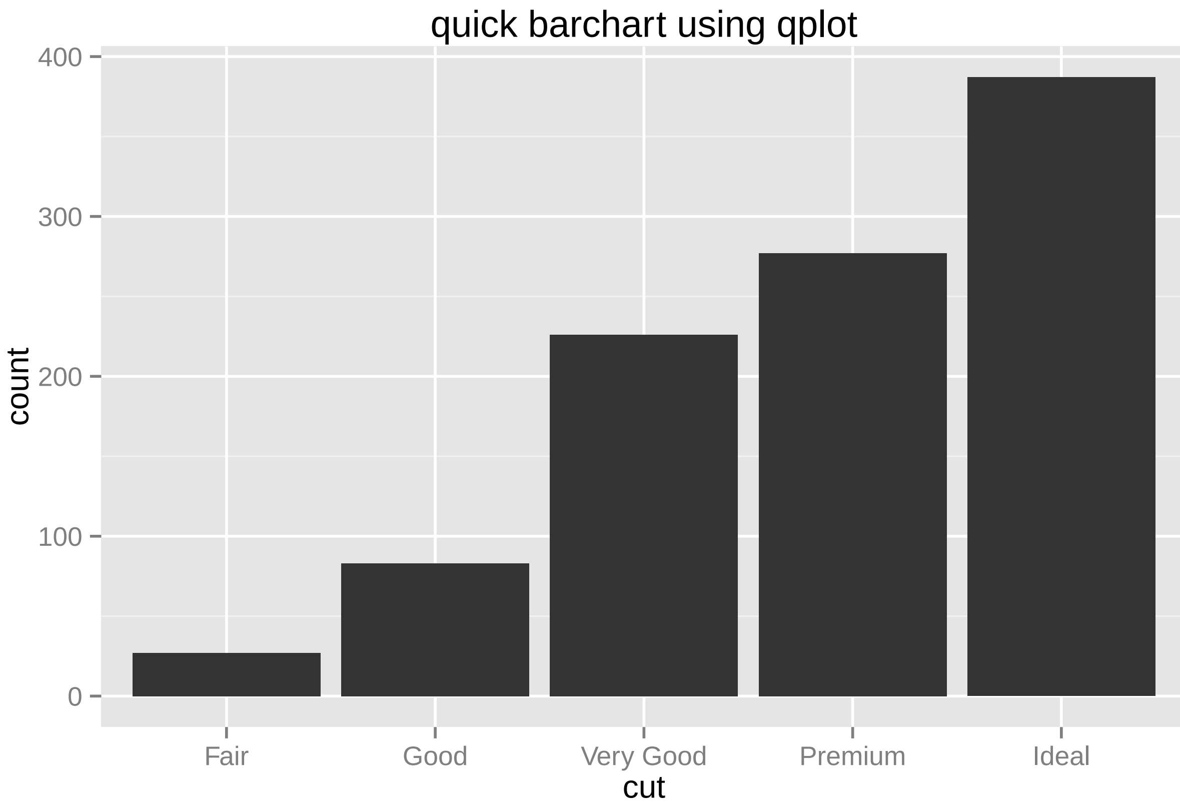
**Base Graphics** To create a barplot/barchart in base graphics requires the data to be in summarized in a table form first. Then the result of the table is plotted. The first argument is the table to be plotted, the `main` argument controls the title.

```
dc <- table(dsmall$cut)
barplot(dc, main="quick barchart using base graphics")
```




**ggplot** ggplot's "quick" plotting method uses the function `qplot()`. The syntax is pretty standard across plot type. You specify what you want plotted along the `x` axis, the `y` axis (optional depending on the type of plot), what geometric shape, or `geom` you want, the `data` set name, and a title using `main`.

```
qplot(x=cut, data=dsmall, geom="bar", main="quick barchart using qplot")
```




When you need something a little more detailed however, you will have to use the `ggplot()` function. It has similar type of arguments but is presented in a different manner. The generic syntax is built up in a stepwise fashion using `+` symbols to "add on" features to the plot.

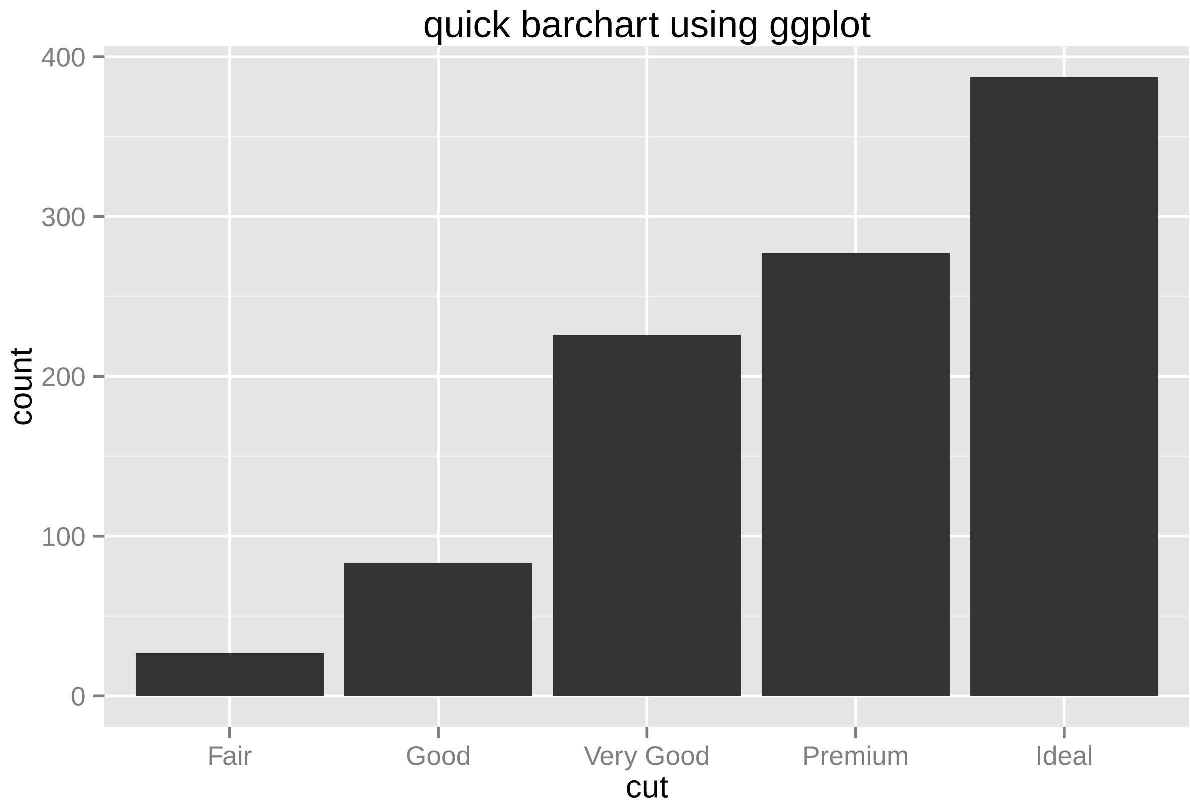
So first you specify what data set you're using, then the aestetic `aes()`, this is where you specify the main plotting variables.

```
ggplot(dsmall, aes(x=cut))
```

```
## Error: No layers in plot
```

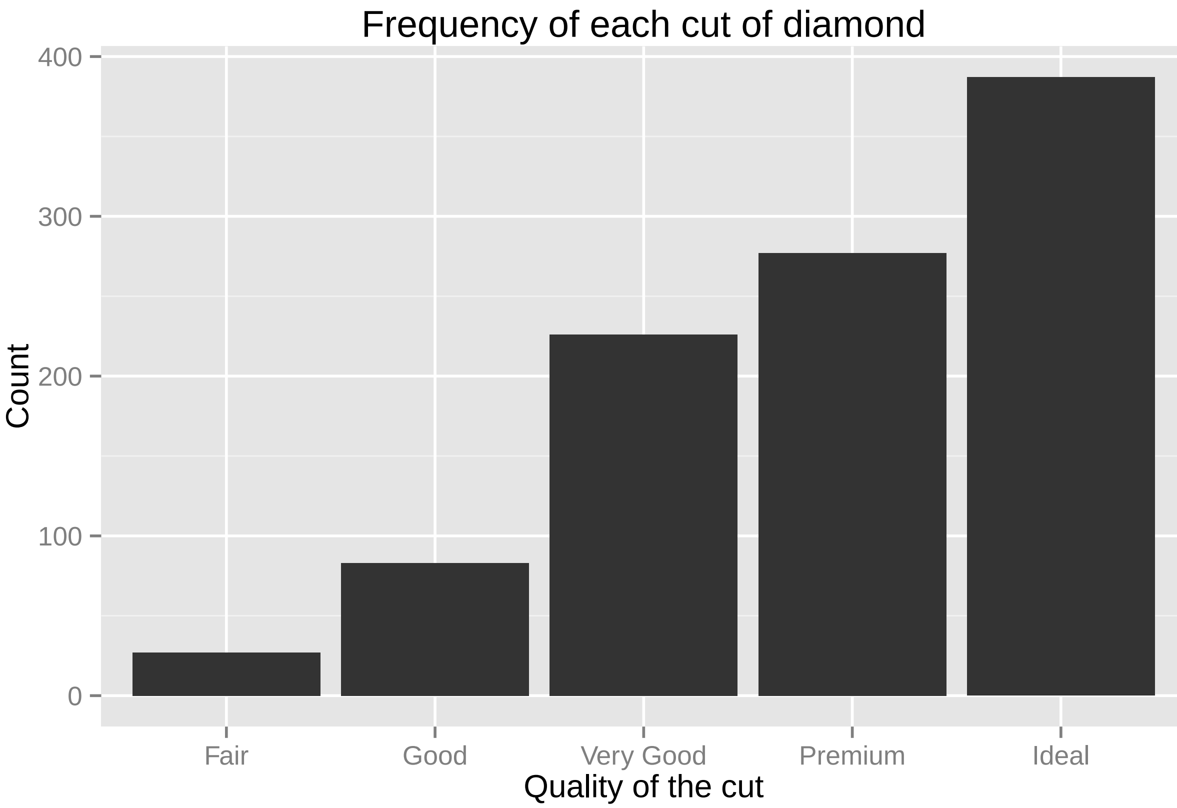
This error message is present because no `geom`'s have been specified. That's where the `geom_bar()` comes in. Note that it's also a function that can take additional arguments. We'll see how to use those later. Here is the `ggplot()` version of the barchart.

```
ggplot(dsmall, aes(x=cut)) +
  geom_bar() + ggtitle("quick barchart using ggplot")
```




And we'll finish off with a presentable version of this plot.

```
ggplot(dsmall, aes(x=cut)) + geom_bar() + ggtitle("Frequency of each cut of diamond") +
  xlab("Quality of the cut") + ylab("Count")
```



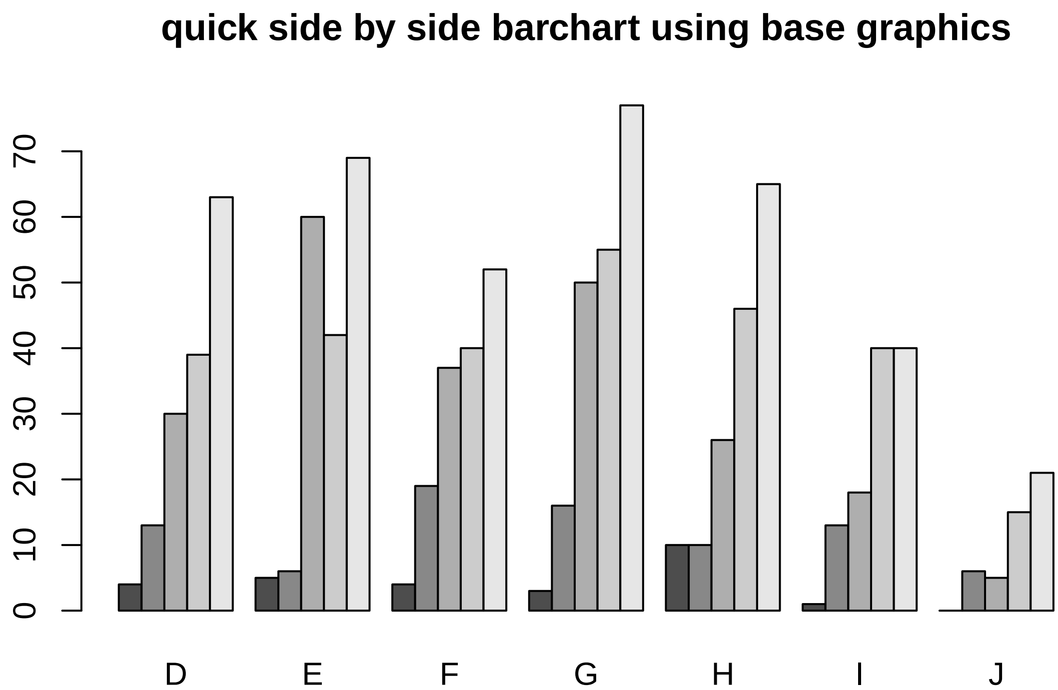

## Two Categorical variables

### Grouped bar charts

To compare proprtions of one categorical variable within the same level of another, is to use grouped barcharts.

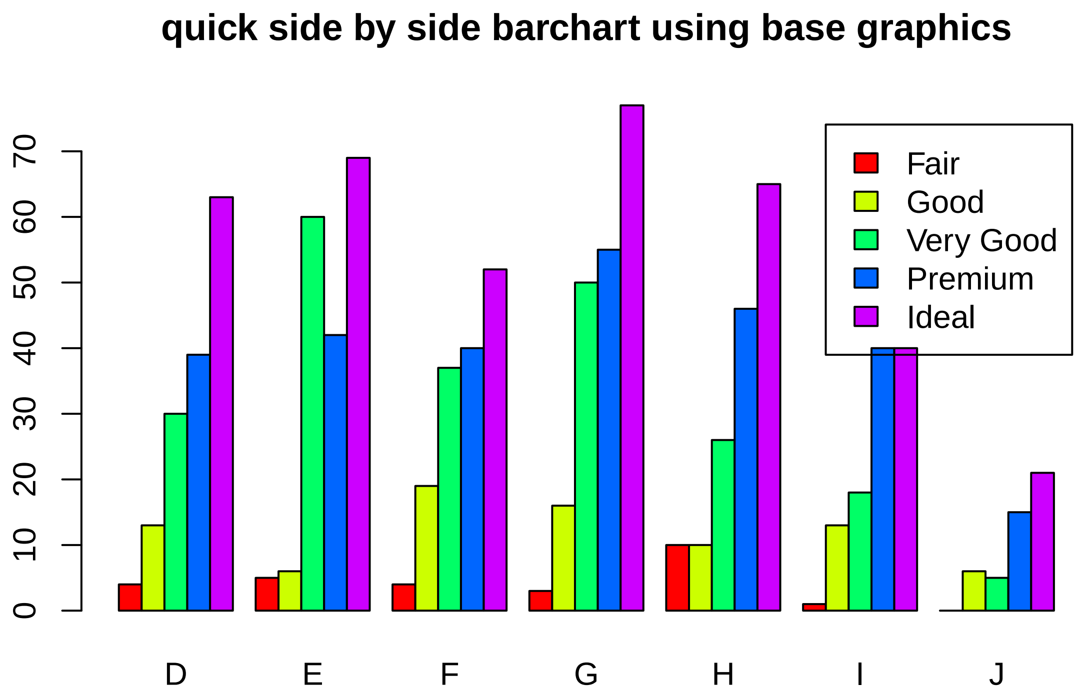
**Base Graphics** As before, the object to be plotted needs to be the result of a table. The `beside=TRUE` is what controls the placement of the bars.

```
cc <- table(dsmall$cut, dsmall$color)
barplot(cc, main="quick side by side barchart using base graphics", beside=TRUE)
```




Great, but what do the colors represent? We need to add a legend. And i'm going to customize the colors.

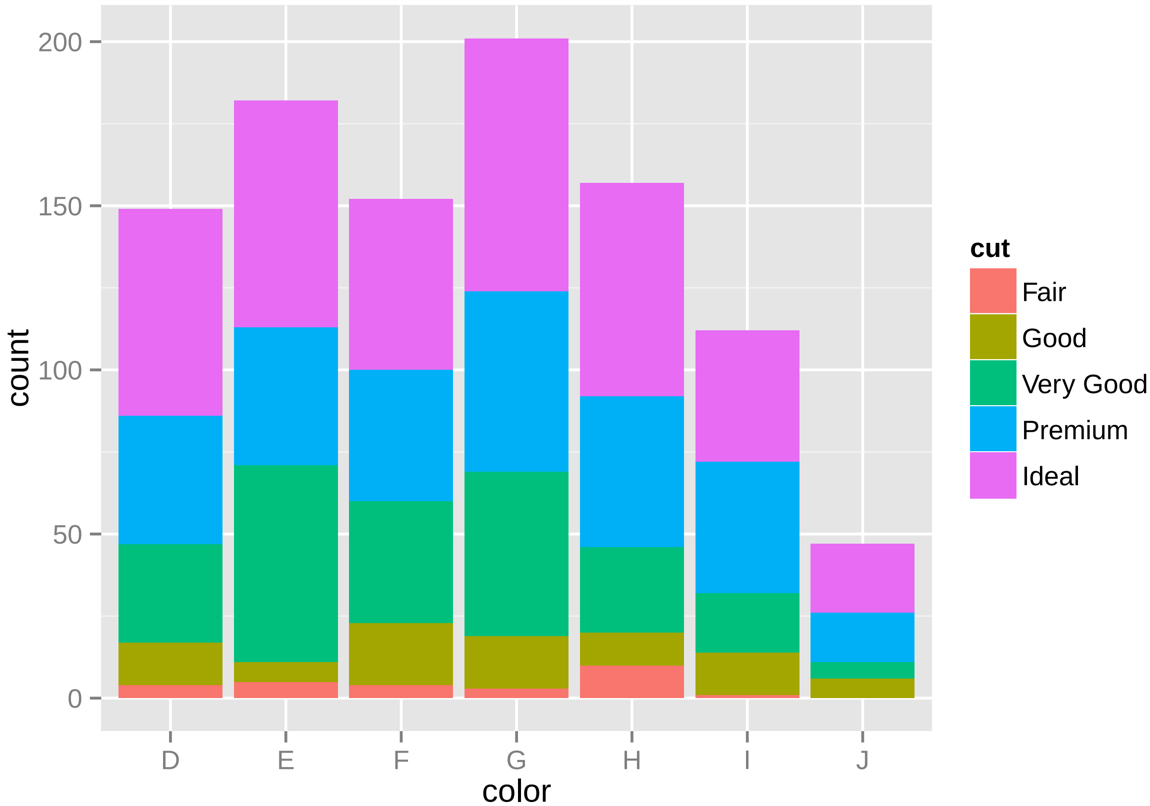
```
barplot(cc, main="quick side by side barchart using base graphics", beside=TRUE,
        col=rainbow(5), legend=rownames(cc))
```




For more than 2 colors I do not recommend choosing the colors yourself. I know little about color theory so I use the built-in color palettes. Here is a great cheatsheet about using color palettes.

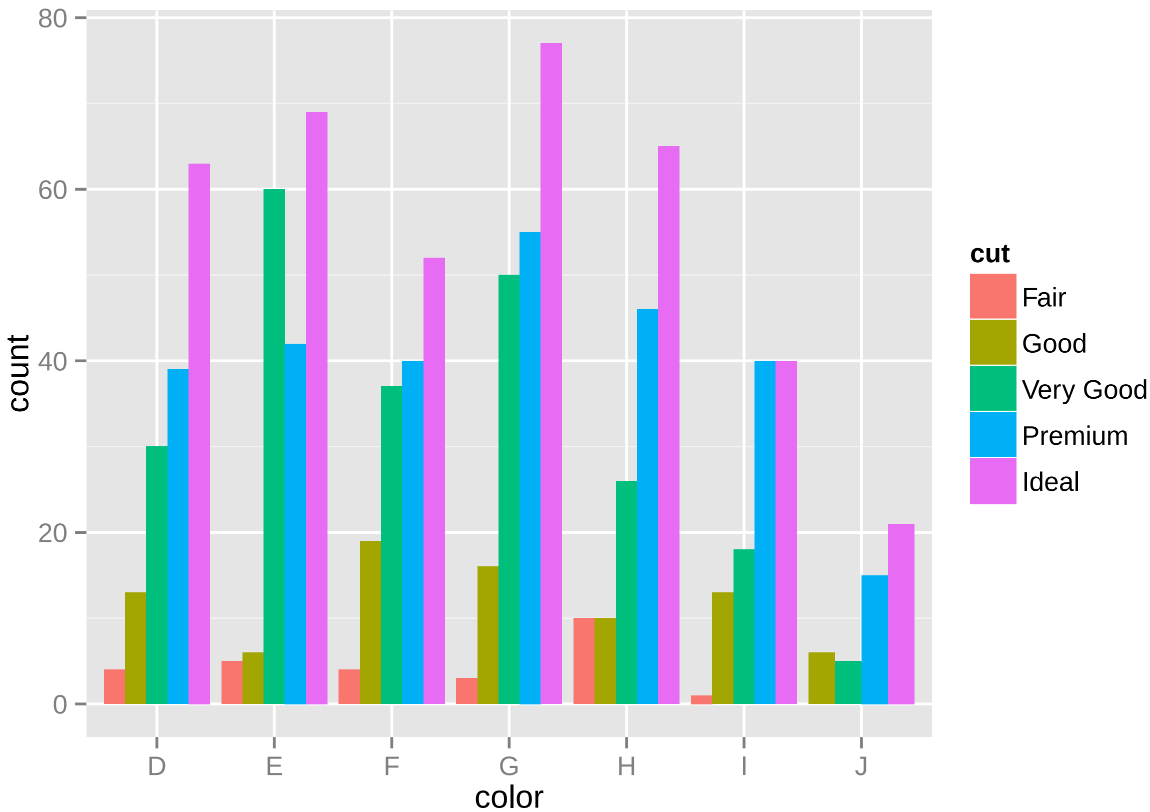
**ggplot** Again starting by using `qplot()` but this time we'll `fill` using the second categorical variable.

```
qplot(x=color, fill=cut, data=dsmall)
```



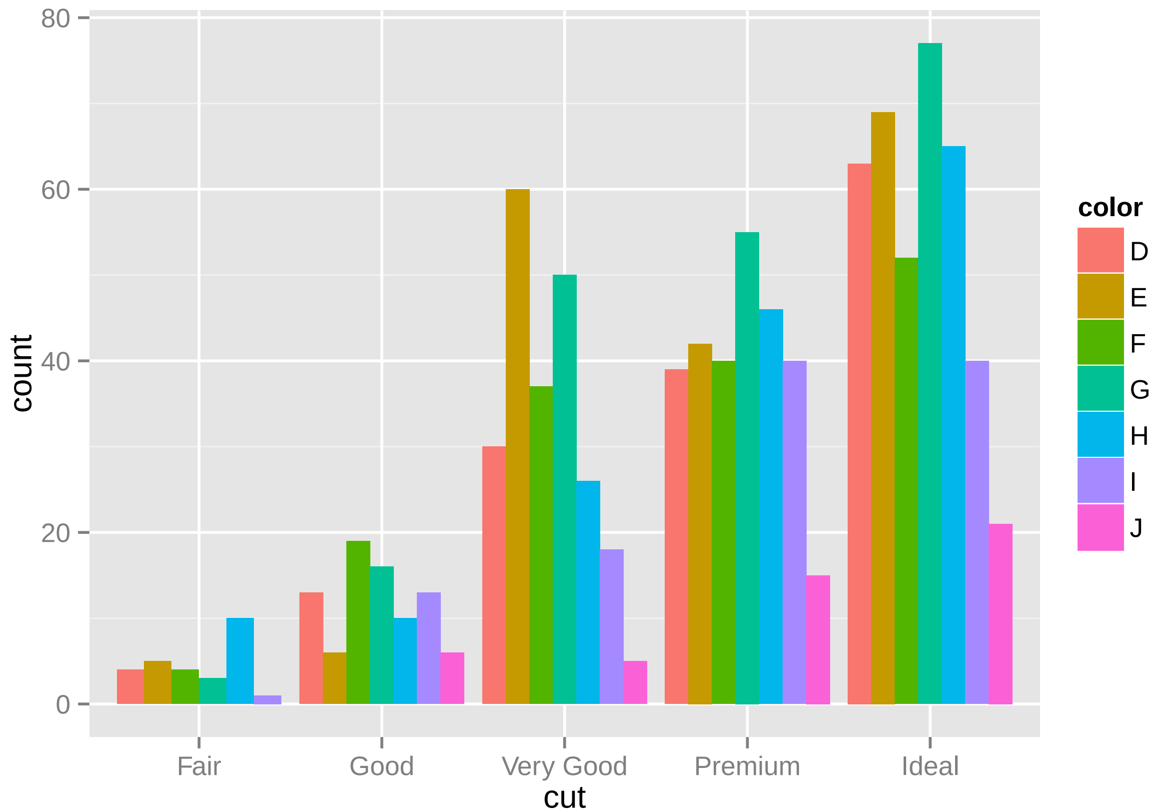

The colors are stacked! I **highly discourage** the use of this type of plot. You can compare across colors how many are fair, but after that it becomes to difficult to compare. What color has more ideal cuts, F or G? So we just specify `position=dodge` to put the bars side by side.

```
qplot(x=color, fill=cut, data=dsmall, position="dodge")
```




And look, an automatic legend. But what if I wanted to better compare color across cut? It's hard to compare individual bars across the groups. Just switch which variable is the x axis and which one is used to fill the colors!

```
qplot(x=cut, fill=color, data=dsmall, position="dodge")
```




### Mosaic plots

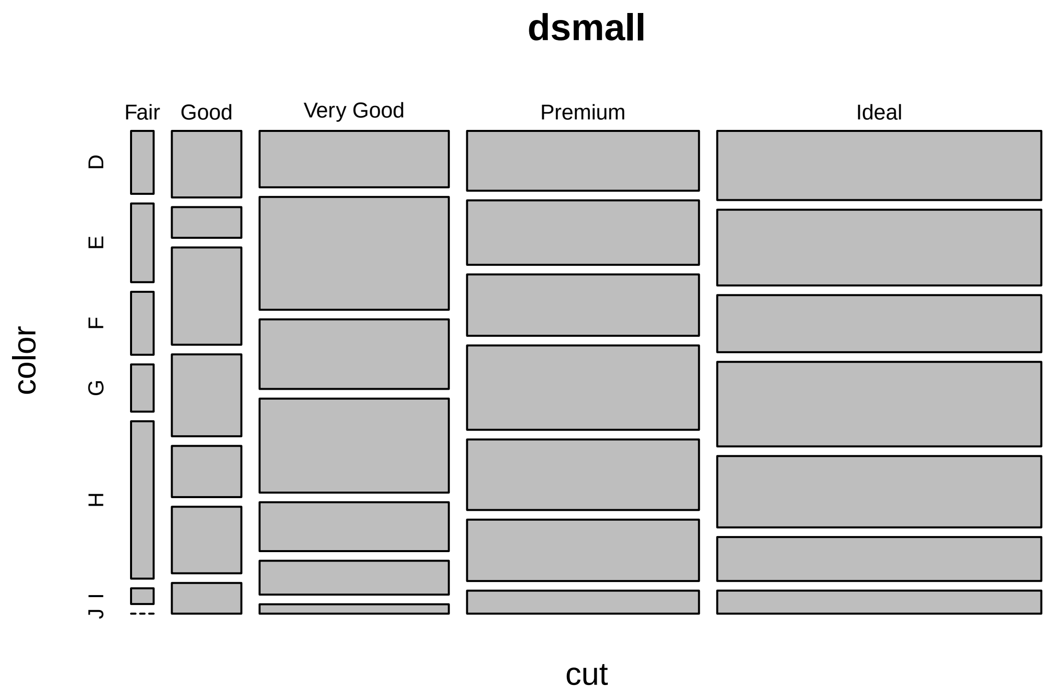
But what if you want to know how two categorical variables are related and you don't want to look at two different barplots? Mosaic plots are a way to visualize the proportions in a table. So here's the two-way table we'll be plotting.

```
table(dsmall$cut, dsmall$color)
```

```
##
##              D  E  F  G  H  I  J
##   Fair       4  5  4  3 10  1  0
##   Good      13  6 19 16 10 13  6
##   Very Good 30 60 37 50 26 18  5
##   Premium   39 42 40 55 46 40 15
##   Ideal     63 69 52 77 65 40 21
```

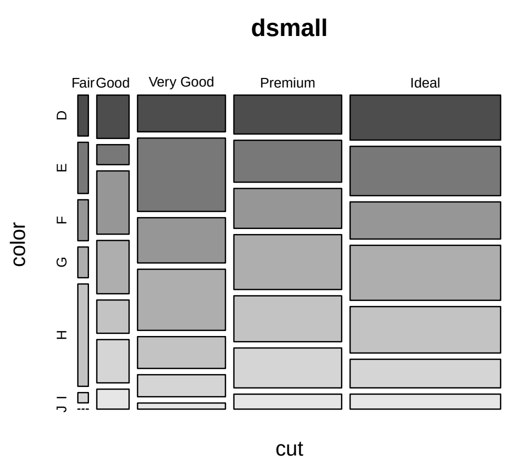
The syntax for a mosiac plot uses *model notation*, which is basically y ~ x where the ~ is read as "twiddle" or "tilde". It's to the left of your **1** key.

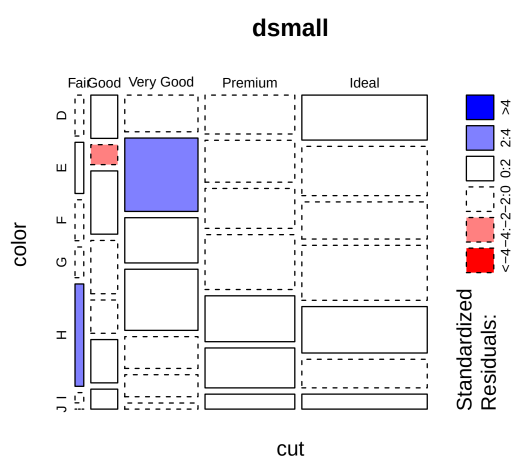
```
mosaicplot(cut~color, data=dsmall)
```




Helpful, ish. Here are two very useful options. In reverse obviousness, `color` applies shades of gray to one of the factor levels, and `shade` applies a color gradient scale to the cells in order of what is less than expected (red) to what is more than expected (blue) if these two factors were compltely independent.

```
par(mfrow=c(1,2)) # display the plots in 1 row and 2 columns
mosaicplot(cut~color, data=dsmall, color=TRUE)
mosaicplot(cut~color, data=dsmall, shade=TRUE)
```



For example, there are fewer 'Very Good' cut diamonds that are color 'G', and fewer 'Premium' cut diamons that are color 'H'. As you can see, knowing what your data means when trying to interpret what the plots are telling you is essential.

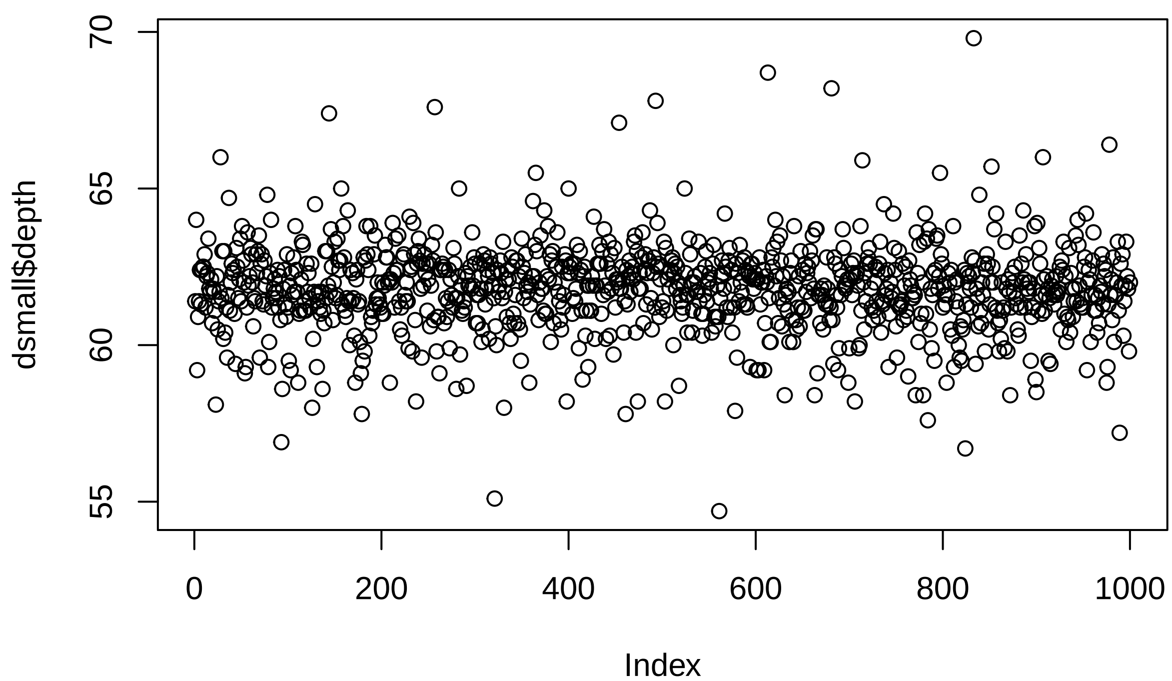
That's about all the ways you can plot categorical variables. If you are wondering why there was no pie charts or 3D barcharts demonstrated see here, and here, or here, here, and here for other ways you can really screw up your visualization.

## One Numeric variable

Numeric variables are ones that can be measured, these are also typically called continuous measurements. Here we can look at the price, carat, and depth of the diamonds.

**Plot** The most basic of basics of plots, which sometimes can still be useful is to use the base R function `plot()`.

```
plot(dsmall$depth)
```

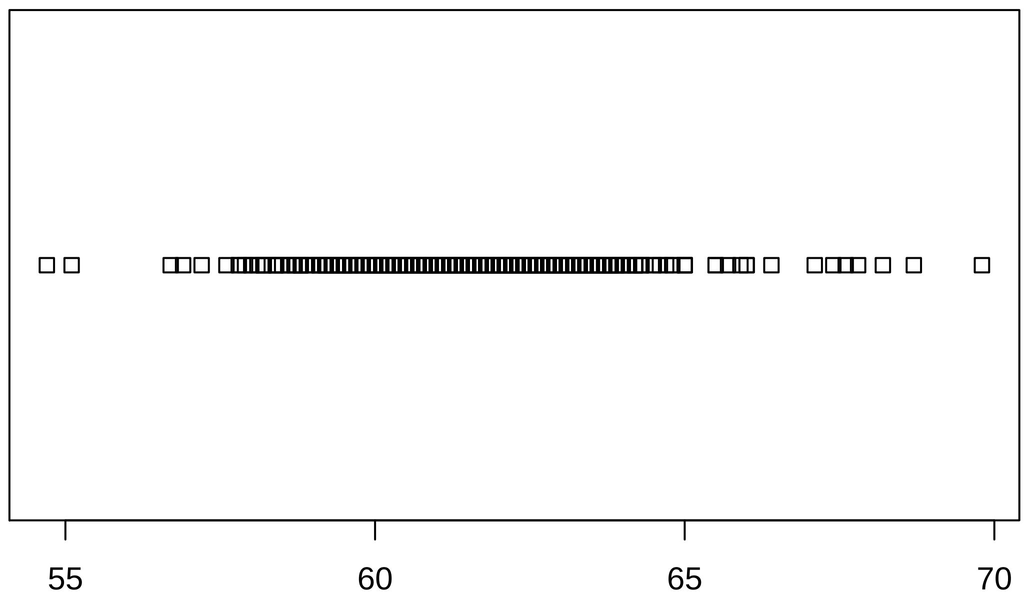



The value of the variable is plotted on the y axis, and the index, or row number, is plotted on the x axis.

#### Dotplot/stripchart

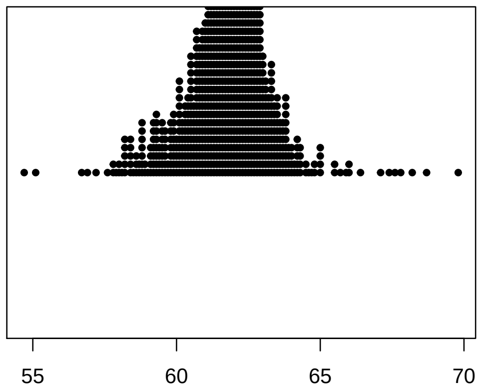
A dotplot or stripchart is the next most simplest plots and yet less informative. One point is plotted per observation, all are plotted on the same line.

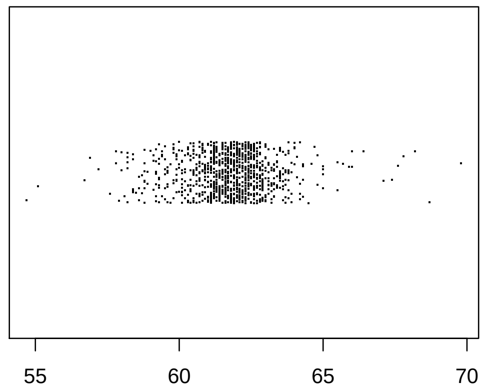
```r
stripchart(dsmall$depth)
```



The x-axis is the value of the `price` variable. So we can see that there seems to be a bit of clustering of point in the low price range. There is also a lot of "overplotting", where points are plotted on top of each other, which can hide various features. There are two main ways to deal with overplotting, we can `stack` the points on top of each other vertically, or `jitter` the points which just adds a little bit of random vertical movement to the point. I am also going to change the point shape using `pch` (look at `?pch` for the codes but 16 is my favorite.)

```r
par(mfrow=c(1,2))
stripchart(dsmall$depth, method="stack", pch=20)
stripchart(dsmall$depth, method="jitter", pch=".")
```





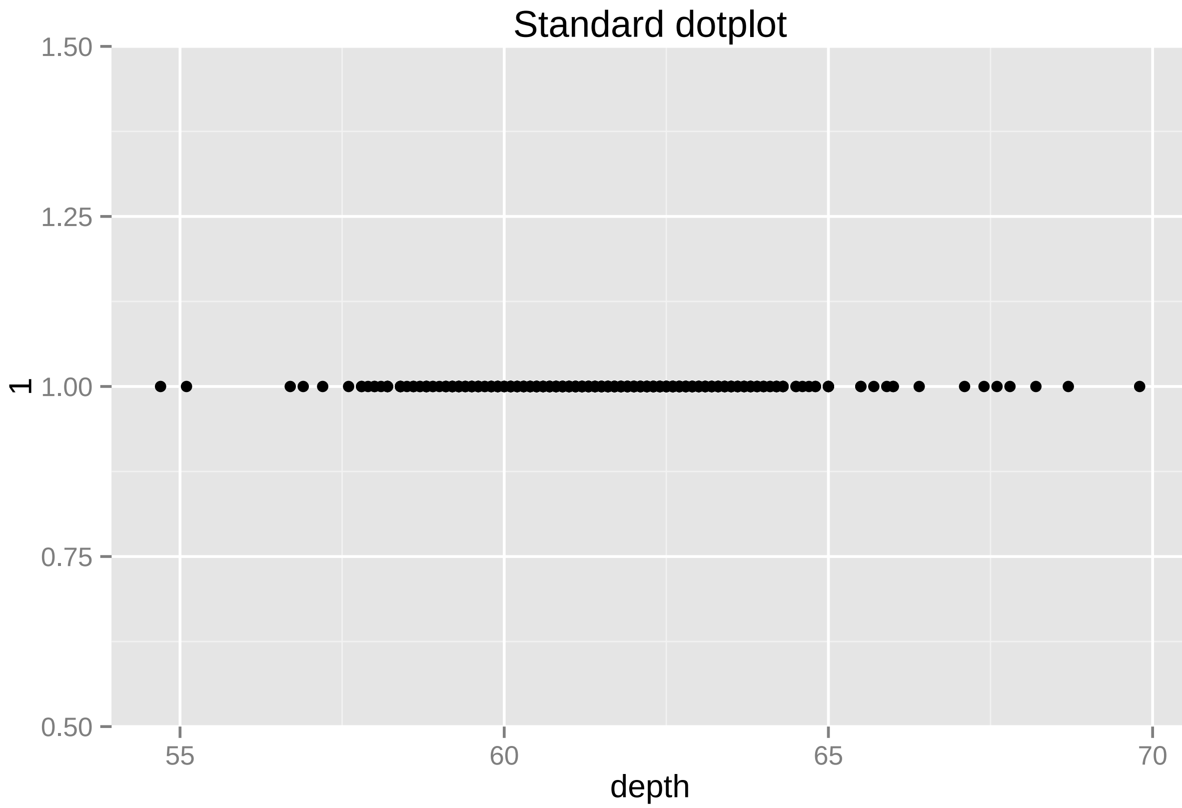

The `par(mfrow=c(1,2))` tells `R` to split the plotting region into 1 row and 2 columns. This is useful for displaying different plots side by side or stacked.

Remember the vertical axis doesn't mean anything here. Univariate dotplots are not helpful to plot raw data. We'll come back to them later.

All of those plots were made with base graphics.

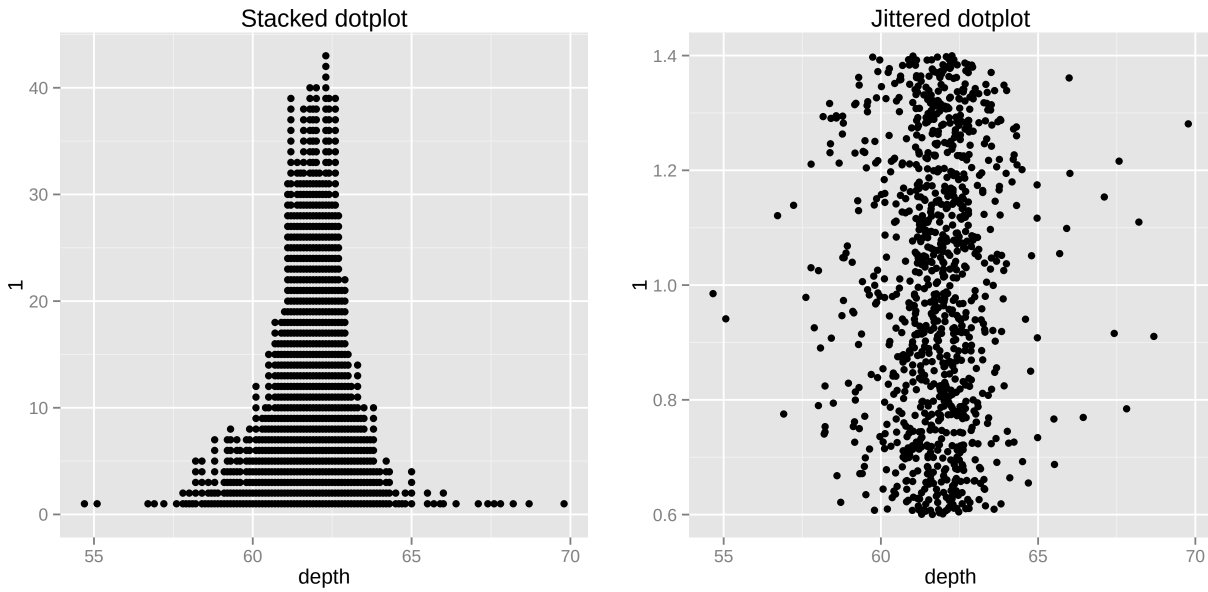
**ggplot** To plot the values of depth against the index you provide a value of 1 `y=1` to put all points on the same line, and use a `point` for the geom.

```
qplot(y=1, x=depth, data=dsmall,  geom="point", main="Standard dotplot")
```




If we want to jitter or stack we can use the `position` argument.

```
library(gridExtra)
jitter <- qplot(y=1, x=depth, data=dsmall,  geom="point", position="jitter", main="Jittered dotplot")
stack <- qplot(y=1, x=depth, data=dsmall,  geom="point", position="stack", main="Stacked dotplot")
grid.arrange(stack, jitter, ncol=2)
```



The `gridExtra` package allows us to arrange multiple `ggplot` plots. It's like the `mfrow()` statement in base graphics.

To get the exact same three plots (not shown) using `ggplot()` we can use the `geom_point()` with different `position =` arguments.

```
ggplot(dsmall, aes(x=depth, y=1)) + geom_point() + ggtitle("Standard dotplot")
ggplot(dsmall, aes(x=depth, y=1)) + geom_point(position = position_stack()) +
       ggtitle("stacked dotplot")
ggplot(dsmall, aes(x=depth, y=1)) + geom_point(position = position_jitter()) +
       ggtitle("jittered dotplot")
```

**Histograms**

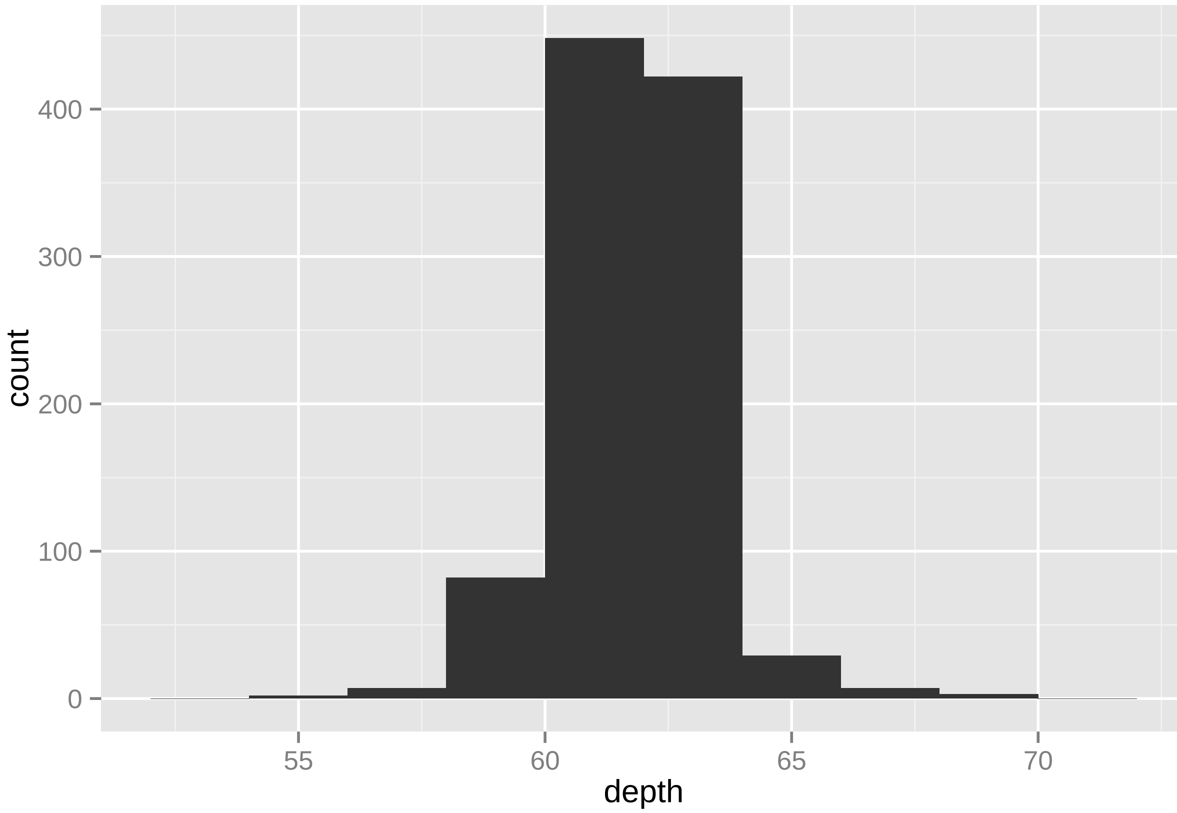
Look at the dotplot on the left, where the dots are stacked vertically. This can be helpful in that the y axis is now a measure of how frequent that x value occurs in the data. Rather than showing the value of each observation, we prefer to think of the value as belonging to a *bin*. The height of the bars in a histogram display the frequency of values that fall into those of those bins. For example if we cut the poverty rates into 7 bins of equal width, the frequency table would look like this:

% latex table generated in R 3.2.2 by xtable 1.7-4 package % Mon Sep 14 11:45:41 2015

| | (54.7,56.9] | (56.9,59] | (59,61.2] | (61.2,63.3] | (63.3,65.5] | (65.5,67.6] | (67.6,69.8] |
|---|---|---|---|---|---|---|---|
| 1 | 3 | 35 | 222 | 654 | 72 | 10 | 4 |

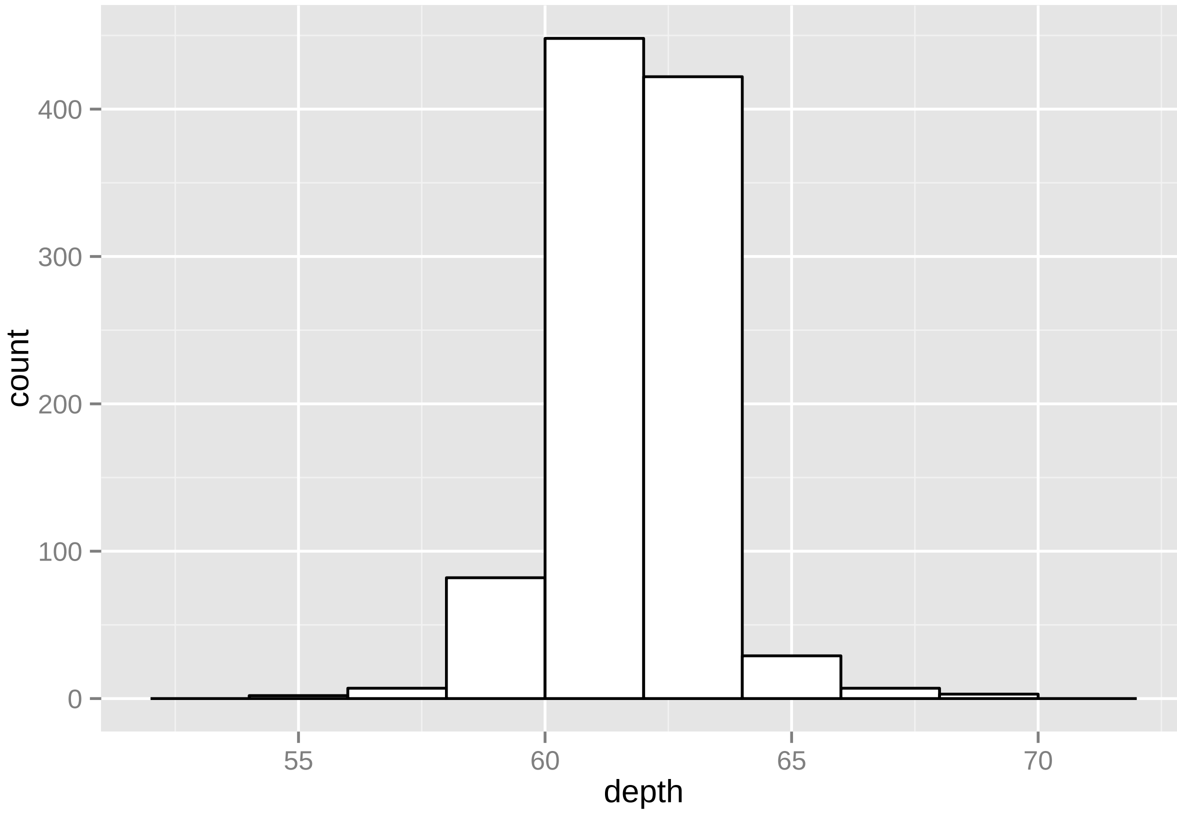
In a histogram, the binned counts are plotted as bars into a histogram. Note that the x-axis is continuous, so the bars touch. This is unlike the barchart that has a categorical x-axis, and vertical bars that are separated.

```
qplot(x=depth, data=dsmall, geom="histogram", binwidth=2)
```



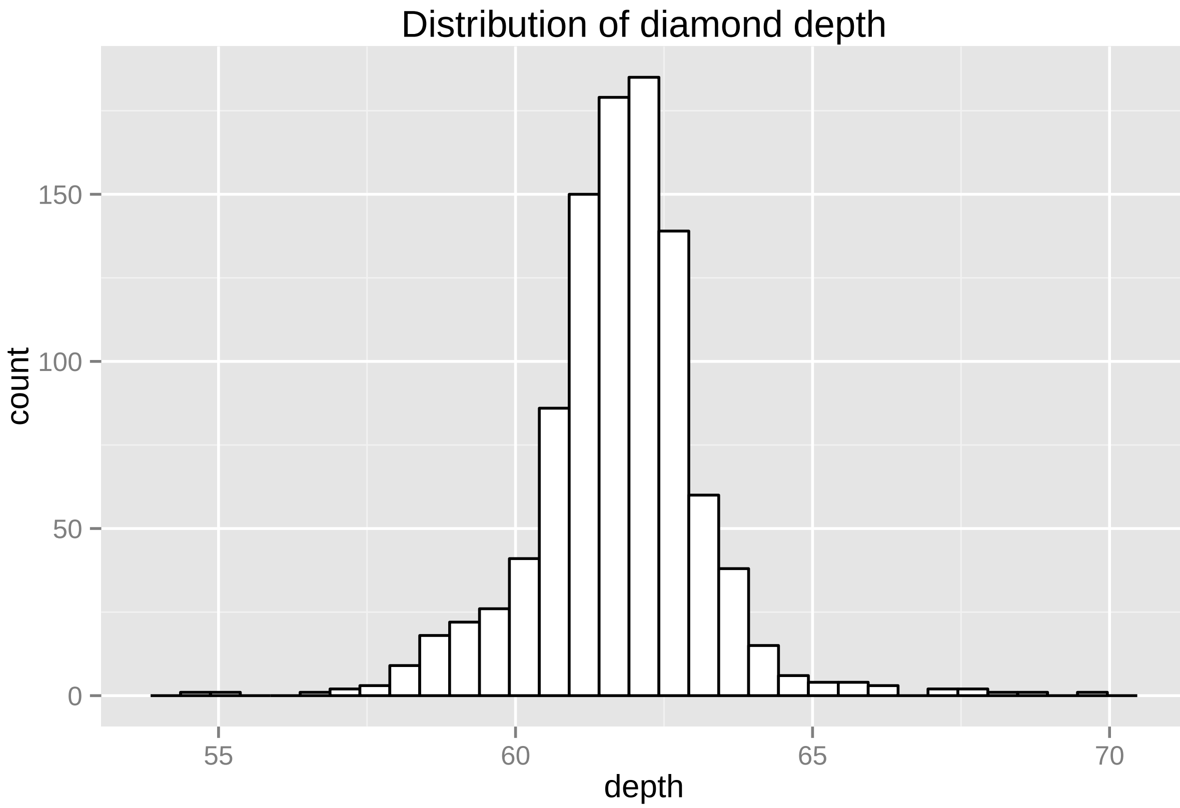

The binwidth is set by looking at the cut points above that were used to create 7 bins, the fill was set to white so that the outlines of the bars could be seen. Darkgrey is the default, but makes it hard to differentiate between the bars. So we'll make the outline black using `color`, and `fill` the bars with white. The `I()` is needed for qplot to use the value you tell it explicitly, the default behavior is to use another variable to determine the color. We'll see how that works later.

```
qplot(x=depth, data=dsmall, geom="histogram", binwidth=2,
      color=I("black"), fill=I("white"))
```




For much more control over histograms, which we will need in the next section, we turn to the `ggplot()` function. We still specifiy the data set, the `aes`tetic is that we want depth on the `x` axis, and then add a `geom_histogram` that has black lines and the bars filled white.

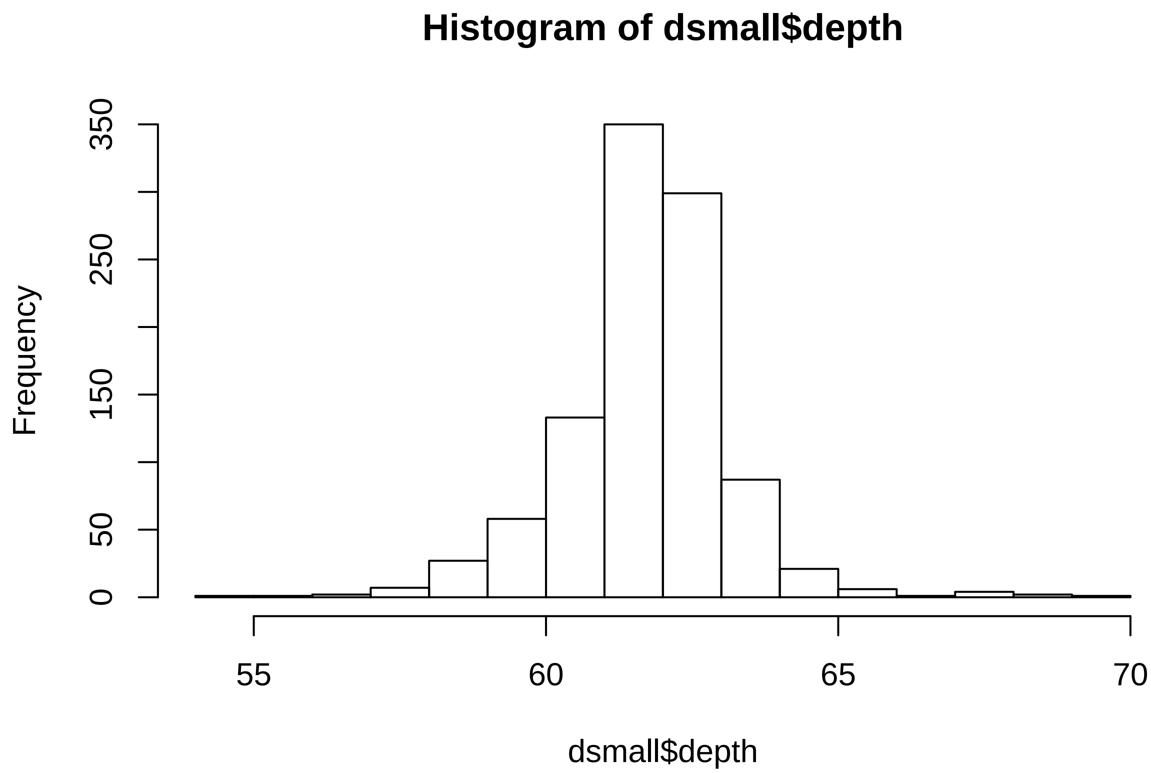
```
ggplot(dsmall, aes(x=depth)) + geom_histogram(colour="black", fill="white") +
  ggtitle("Distribution of diamond depth")
```




Note I did **not** specify the `binwidth` argument here. The size of the bins can hide features from your graph, the default value for ggplot2 is range/30 and usually is a good choice.

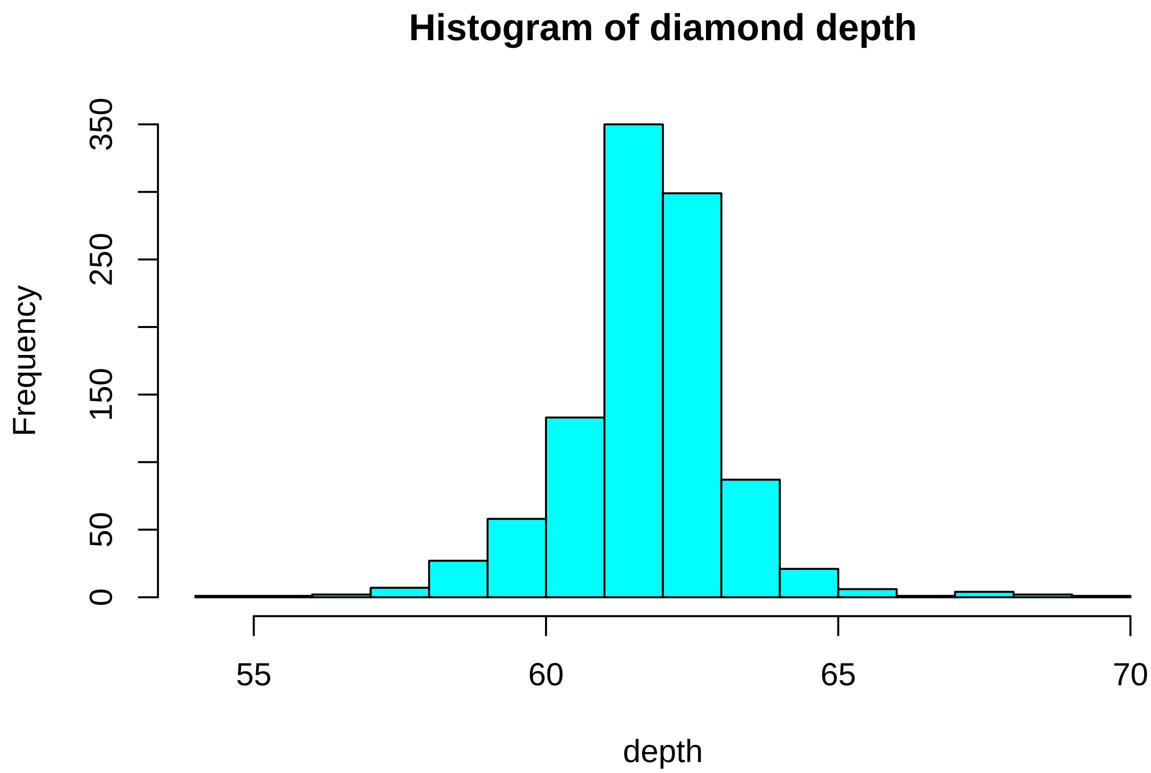
**Base graphics** But what about base graphics? You can make a histogram in base graphics super easy.

```
hist(dsmall$depth)
```




And it doesn't take too much to clean it up. Here you can specify the number of bins by specifying how many `breaks` should be made in the data and use `col` for the fill color.

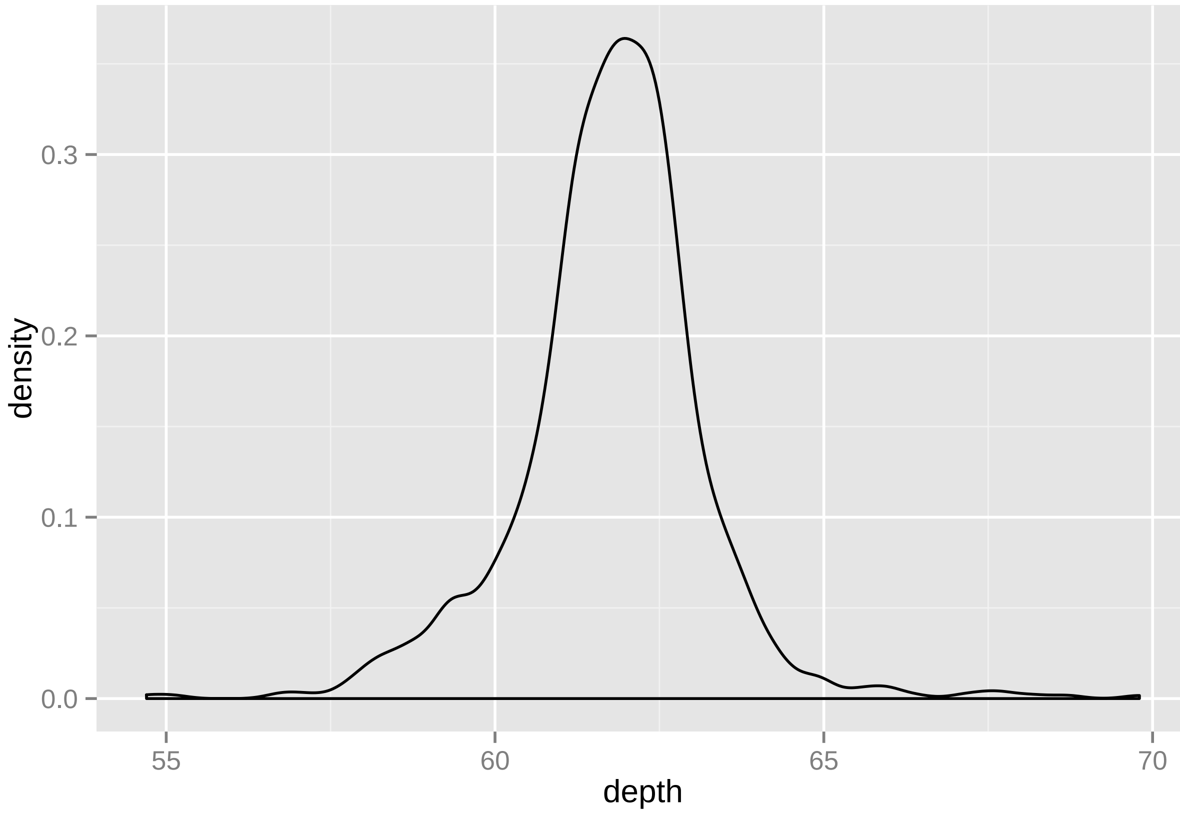
```
hist(dsmall$depth, xlab="depth", main="Histogram of diamond depth", col="cyan", breaks=20)
```




**Density plots**

To get a better idea of the true shape of the distribution we can "smooth" out the bins and create what's called a `density` plot or curve. Notice that the shape of this distribution curve is much more... "wigglier" than the histogram may have implied.

```
qplot(x=depth, data=dsmall, geom="density")
```

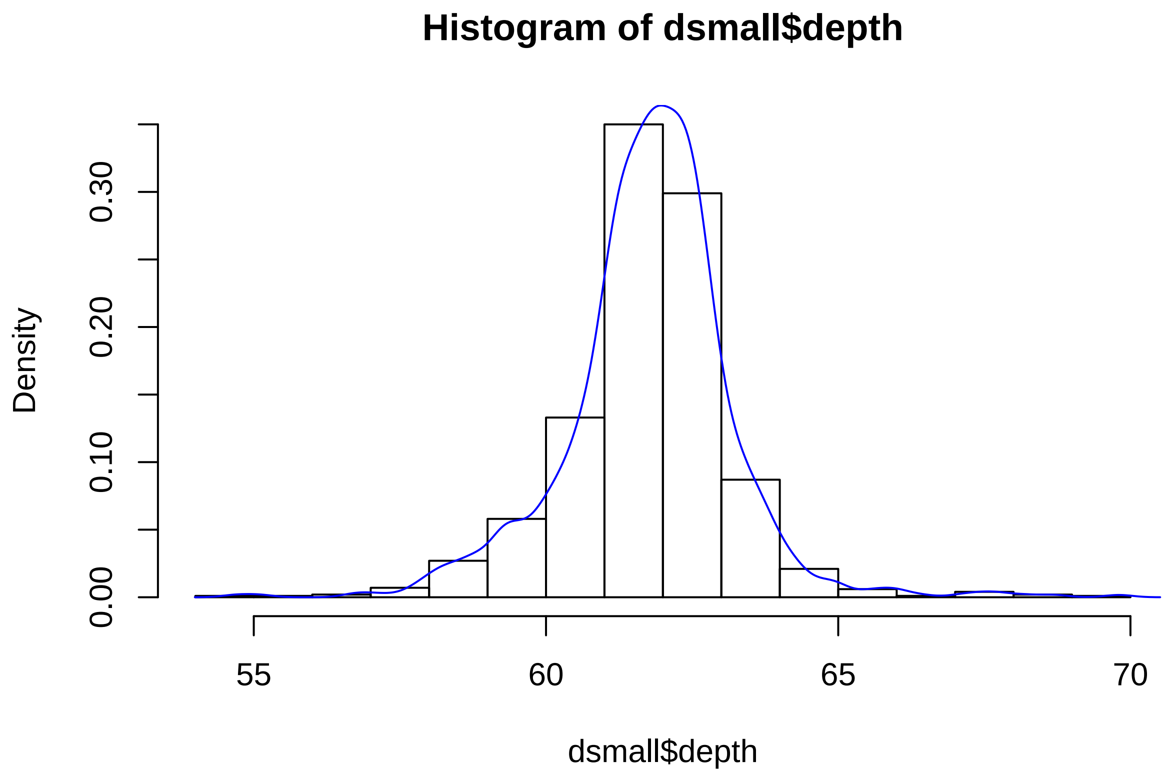



**Histograms with density plots overlaid**

Often is is more helpful to have the density (or kernal density) plot *on top of* a histogram plot.

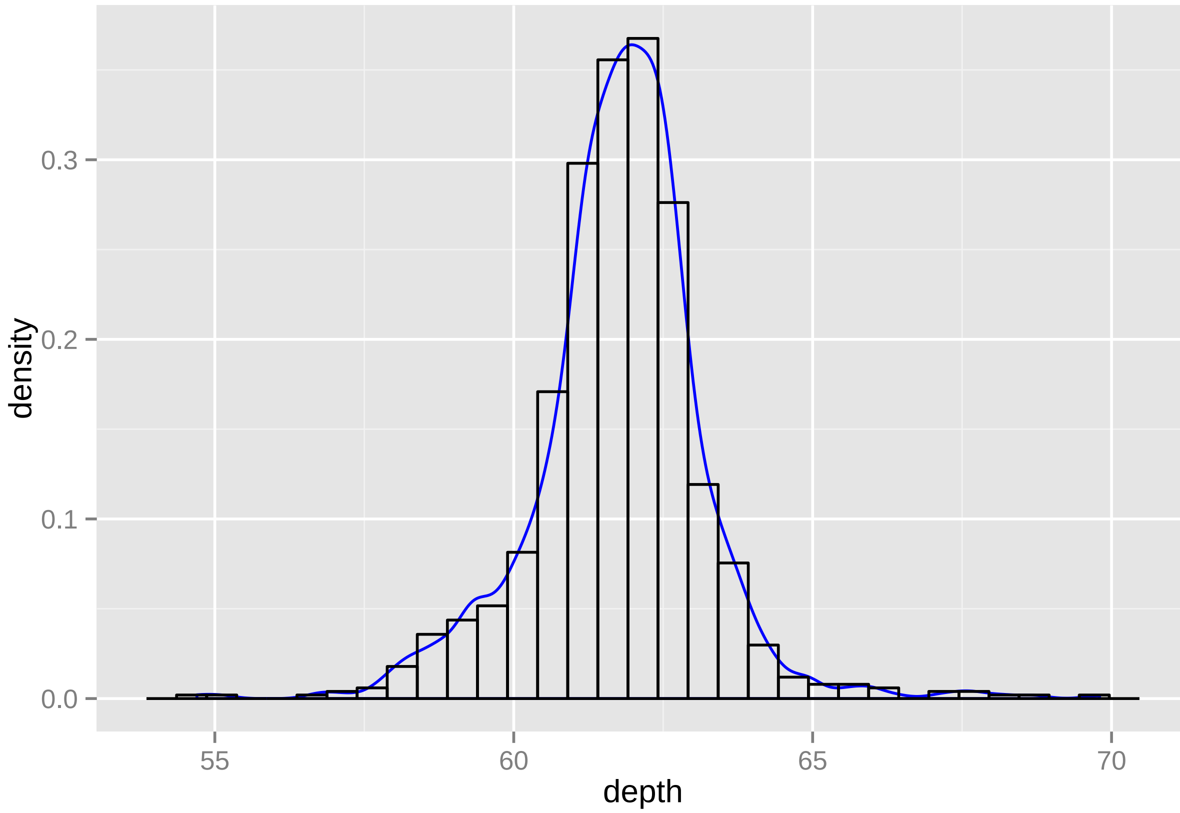
**Base graphics** Since the height of the bars in a histogram default to showing the frequency of records in the data set within that bin, we need to 1) scale the height so that it's a *relative frequency*, and then use the `lines()` function to add a `density()` line on top.

```
hist(dsmall$depth, prob=TRUE)
lines(density(dsmall$depth), col="blue")
```




**Ggplot2** This level of customization can't be achieved using `qplot`, but it's not that hard using `ggplot`. The syntax starts the same, we'll add a new geom, `geom_density` and color the line blue. Then we add the histogram geom using `geom_histogram` but must specify that the y axis should be on the density, not frequency, scale. Note that this has to go inside the aestetic statement `aes()`. I'm also going to get rid of the fill by using `NA` so it doesn't plot over the density line.

```
ggplot(dsmall, aes(x=depth)) + geom_density(col="blue") +
  geom_histogram(aes(y=..density..), colour="black", fill=NA)
```

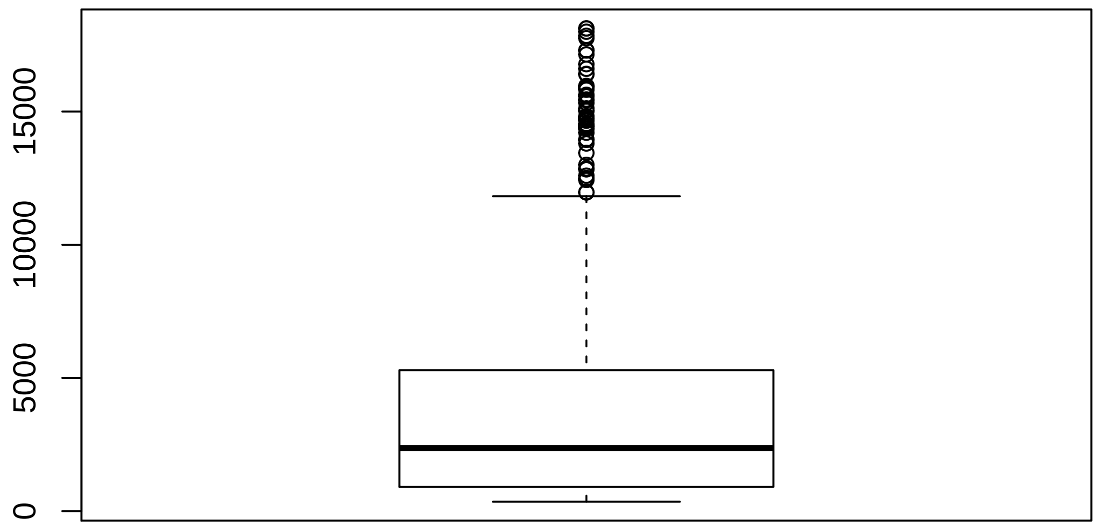



**Boxplots**

Another very common way to visualize the distribution of a continuous variable is using a boxplot. Boxplots are useful for quickly identifying where the bulk of your data lie. R specifically draws a "modified" boxplot where values that are considered outliers are plotted as dots.

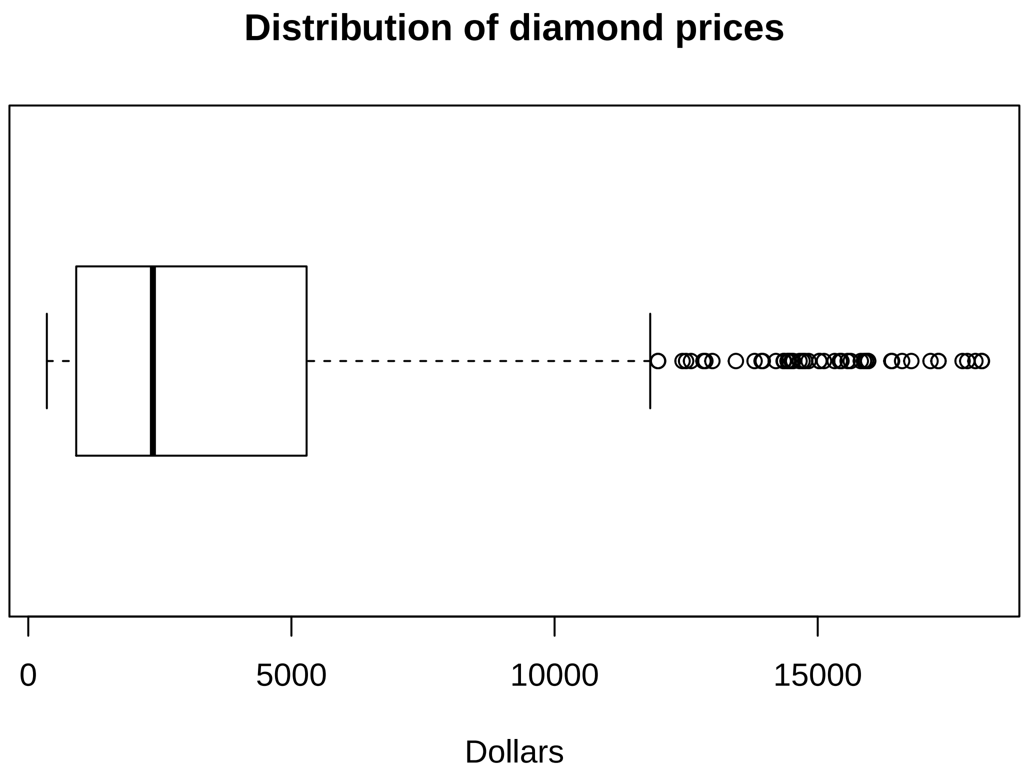
**Base Graphics**

```
boxplot(dsmall$price)
```




Notice that the only axis labeled is the y=axis. Like a dotplot the x axis, or "width", of the boxplot is meaningless here. We can make the axis more readable by flipping the plot on it's side.

```
boxplot(dsmall$price, horizontal = TRUE, main="Distribution of diamond prices", xlab="Dollars")
```



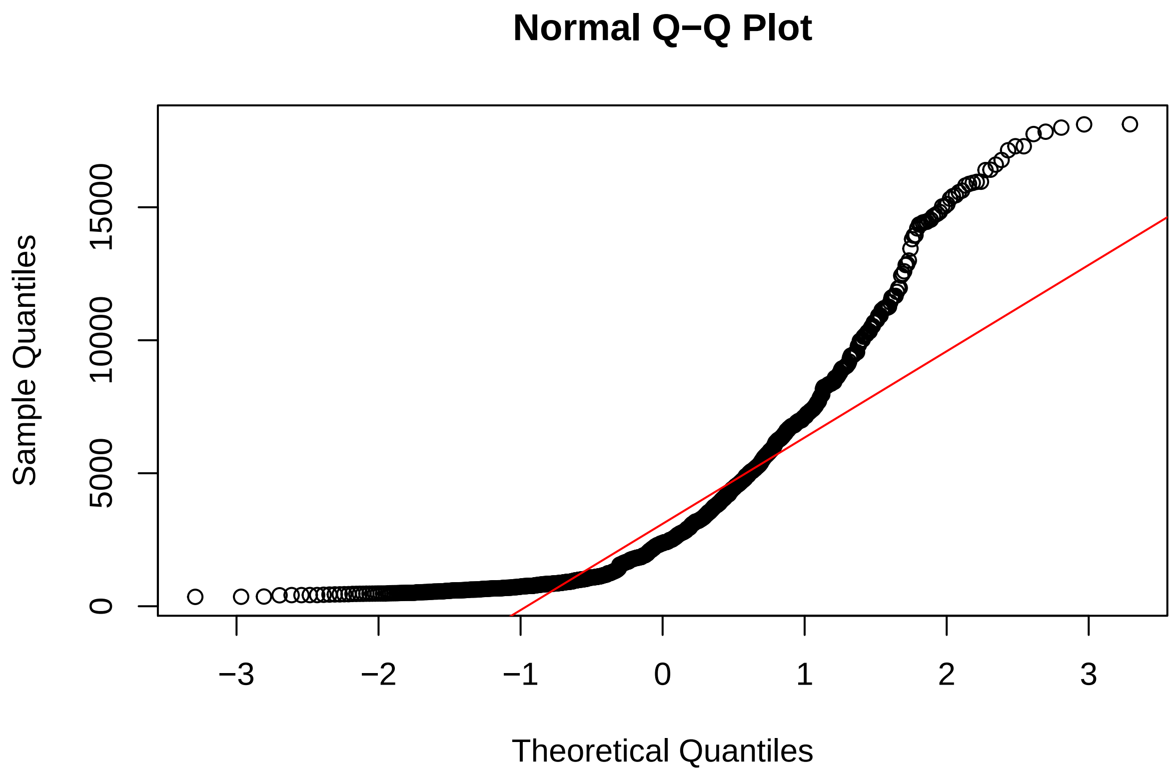

Horizontal is a bit easier to read in my opinion. What about ggplot? ggplot doesn't do univariate boxplots (that I can easily figure out.) We'll come back to grouped boxplots when we discuss plotting the relationship between a continuous and categorical variable.

## Normal QQ plots

The last useful plot that we will do on a single continuous variable is to assess the *normality* of the distribution. Basically how close the data follows a normal distribution.

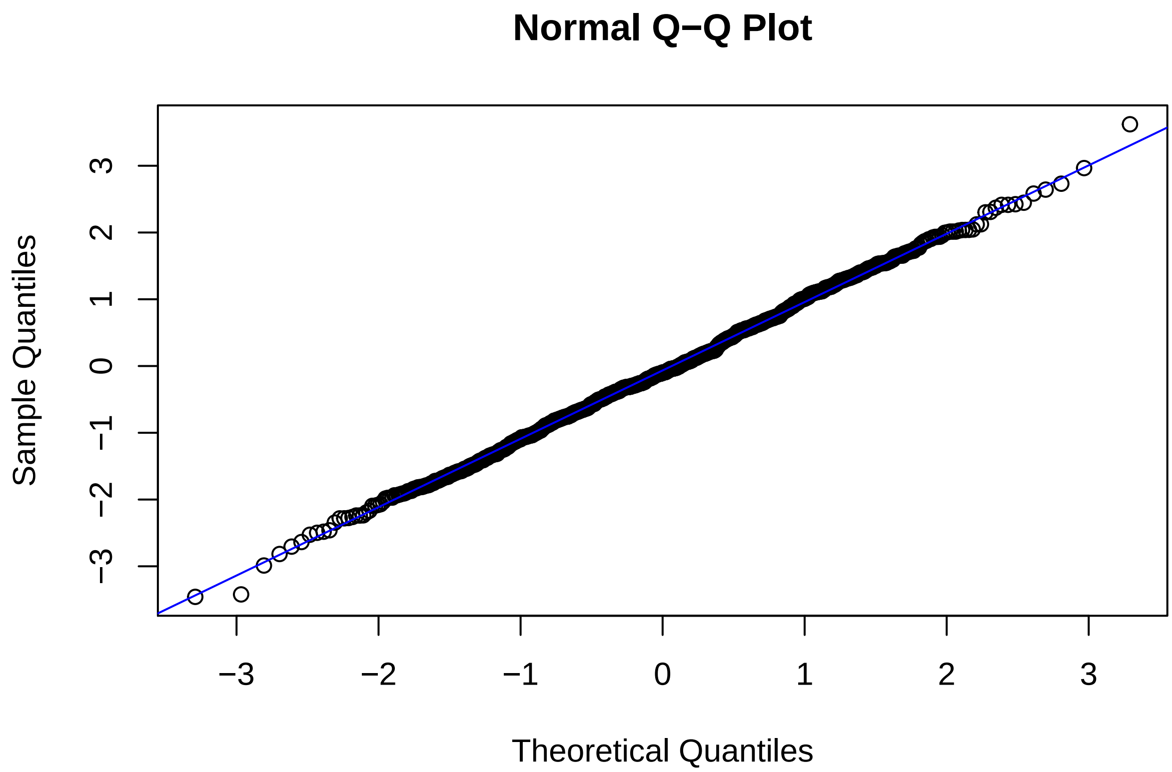
### Base graphics

```
qqnorm(dsmall$price)
qqline(dsmall$price, col="red")
```




The line I make red because it is a reference line. The closer the points are to following this line, the more “normal” the shape of the distribution is. Price has some pretty strong deviation away from that line. Below I have plotted what a normal distribution looks like as an example of a “perfect” fit.

```
z <- rnorm(1000)
qqnorm(z)
qqline(z, col="blue")
```




**ggplot** Its much harder to create a QQplot using ggplot so we are going to skip it.

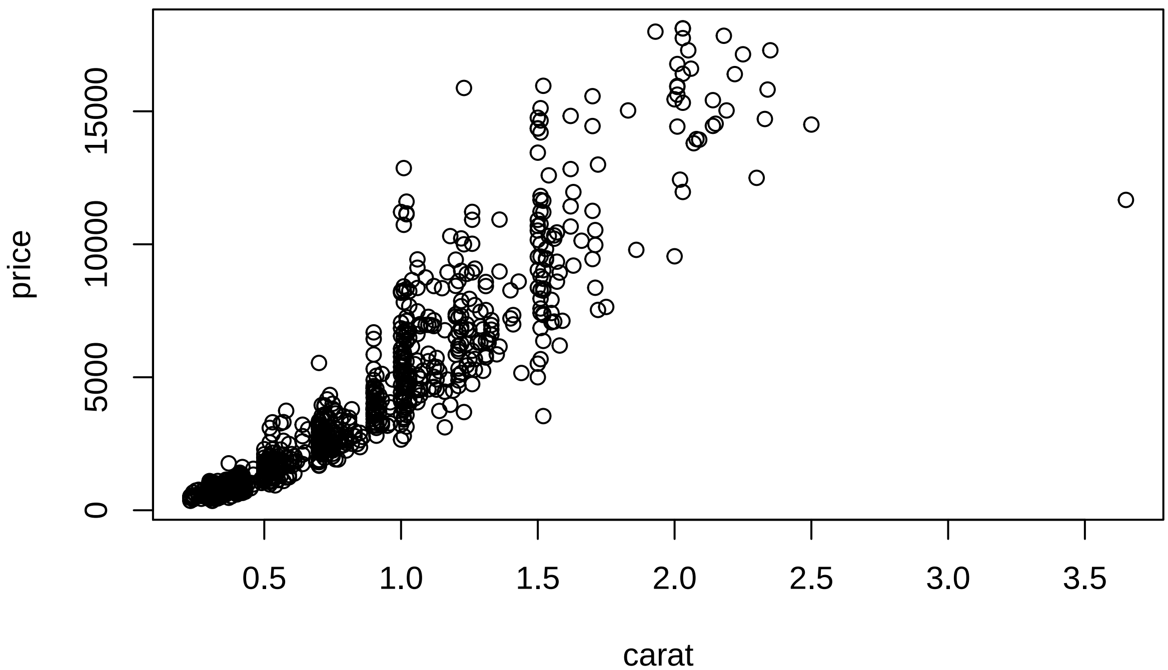
## Two numeric variables

### Scatterplot

The most common method of visualizing the relationship between two continuous variables is by using a scatterplot.

**Base graphics** Back to the `plot()` command.

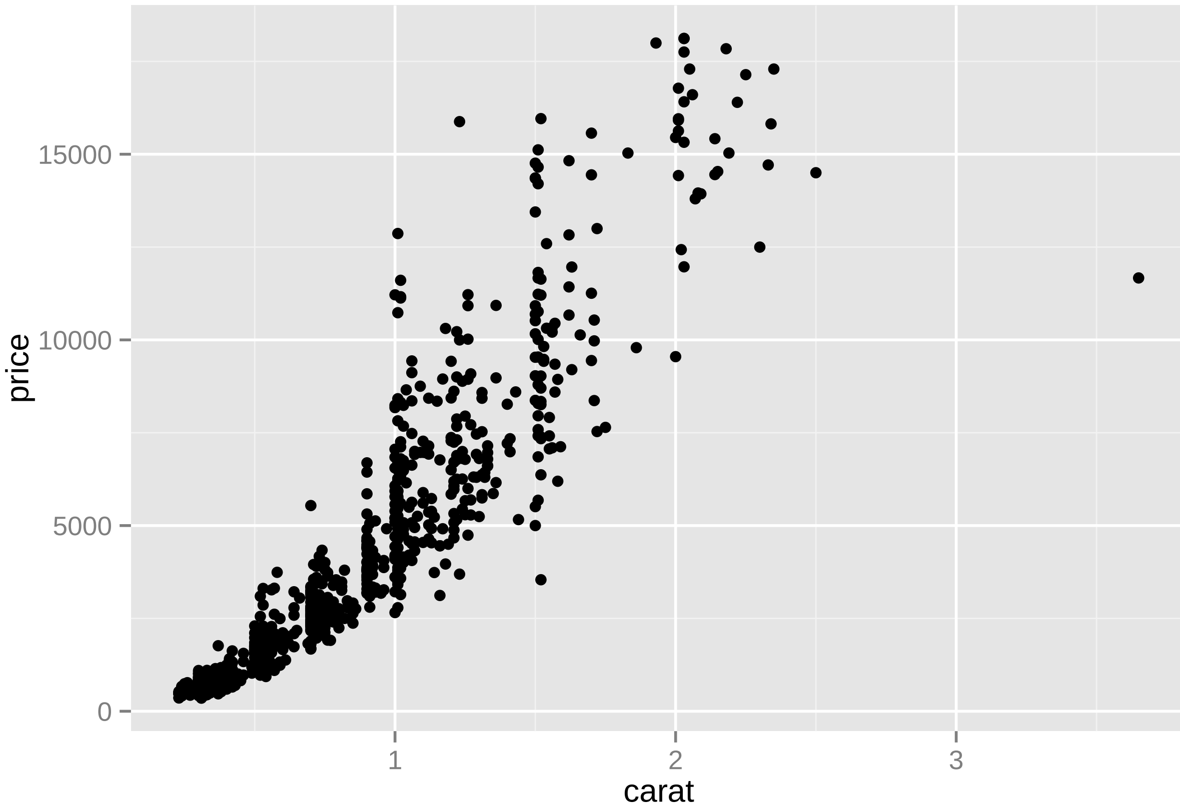
```
plot(price~carat, data=dsmall)
```




Looks like for the most part as the carat value increases so does price. That makes sense.

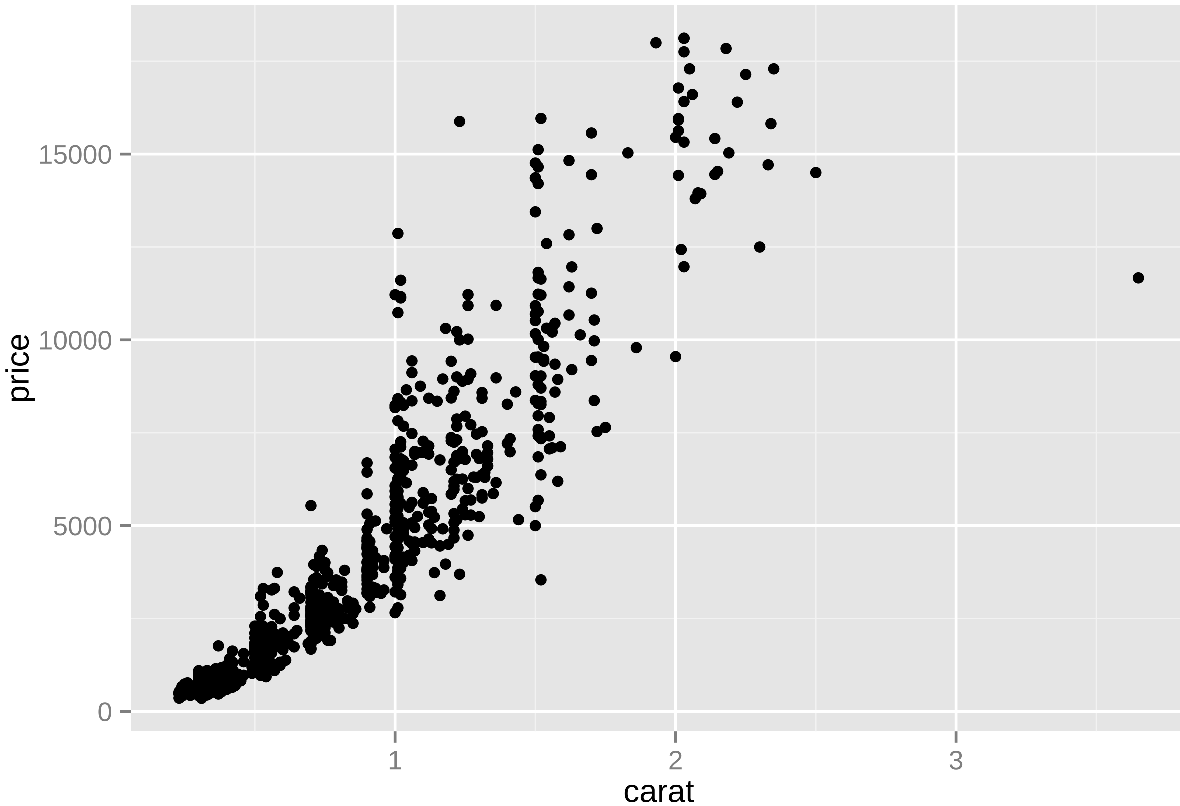
#### ggplot

```
qplot(x=carat, y=price, data=dsmall, geom='point')
```




Or alternatively we can use the ggplot argument with geom_point().

```
ggplot(dsmall, aes(x=carat, y=price)) + geom_point()
```

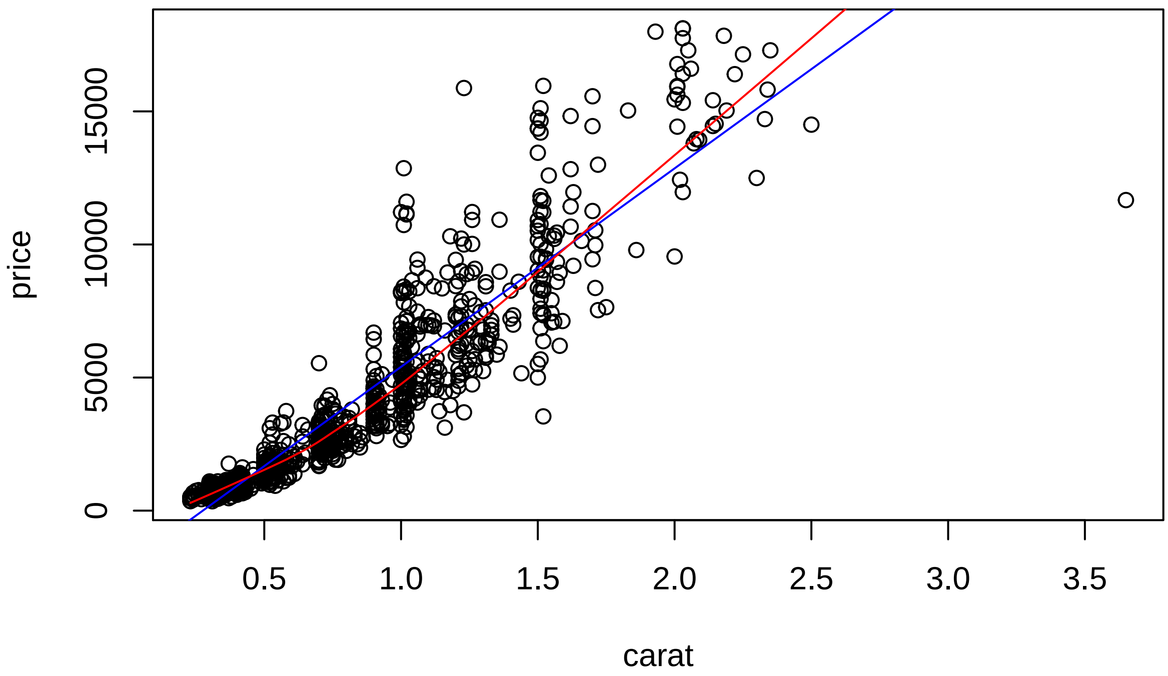



**Adding lines to the scatterplots**

Two most common trend lines added to a scatterplots are the "best fit" straight line and the "lowess" smoother line.

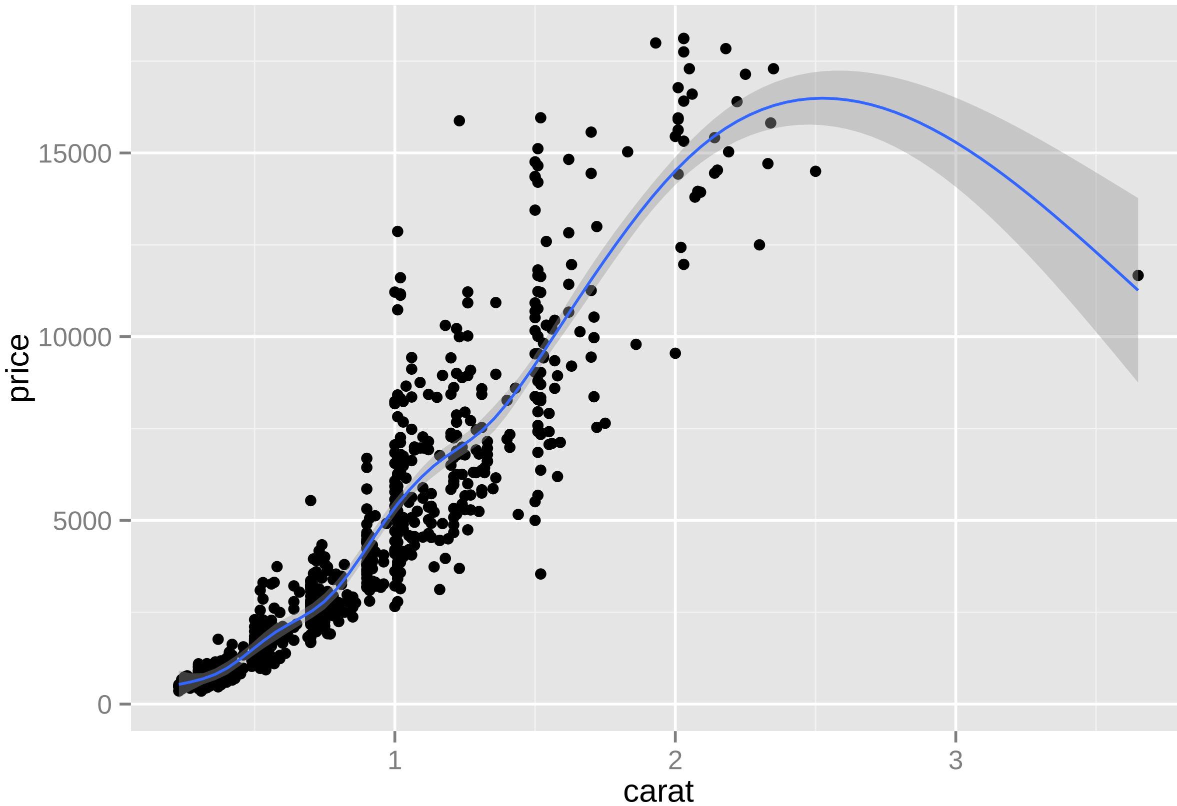
**base graphics** The best fit line (in blue) gets added by using the `abline()` function wrapped around the linear model function `lm()`. Note it uses the same model notation syntax and the `data=` statement as the `plot()` function does. The lowess line is added using the `lines()` function, but the `lowess()` function itself doesn't allow for the `data=` statement so we have to use `$` sign notation.

```
plot(price~carat, data=dsmall)
abline(lm(price~carat, data=dsmall), col="blue")
lines(lowess(dsmall$price~dsmall$carat), col="red")
```



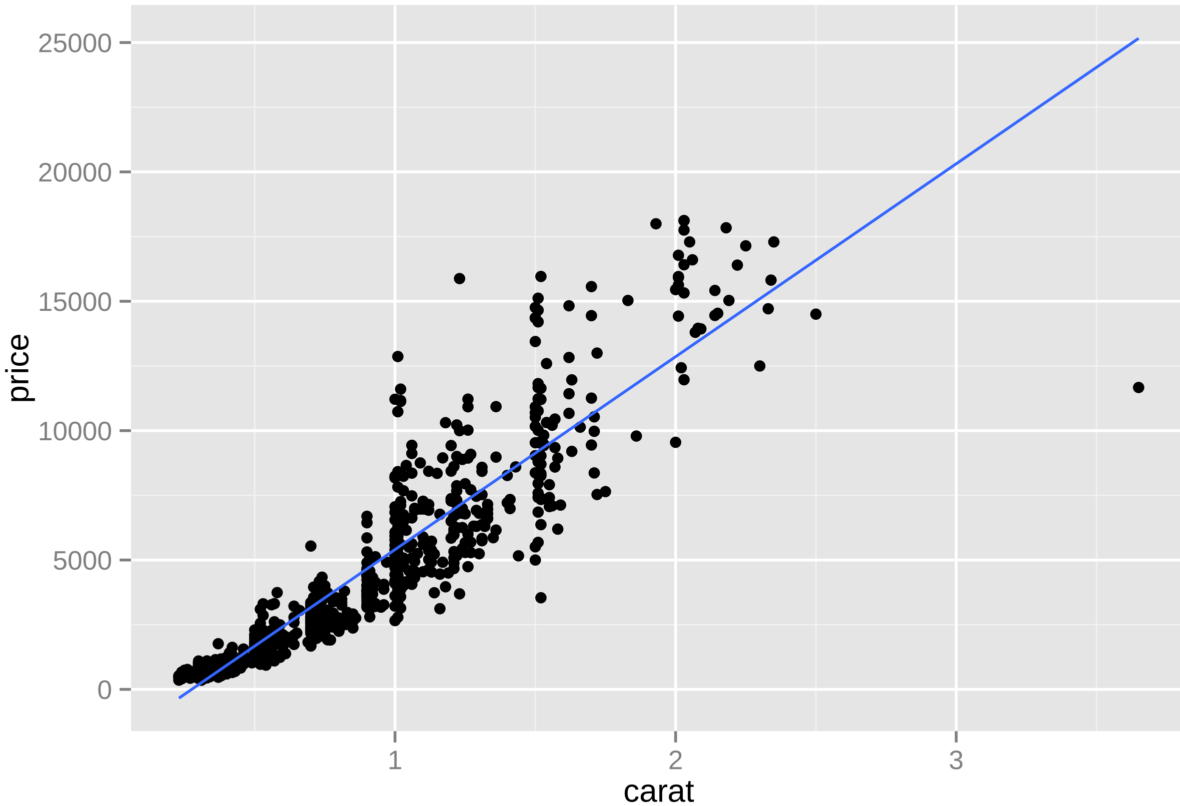

**ggplot**

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




Here the point-wise confidence interval is shown in grey. If you want to turn the confidence interval off, use `se=FALSE`. Also notice that the smoothing geom uses a different function or window than the `lowess` function used in base graphics. Here it is again using the `ggplot` plotting function and adding the `geom_smooth()` function and plotting the `lm` (linear model) line instead of the lowess smoothing algorithm.

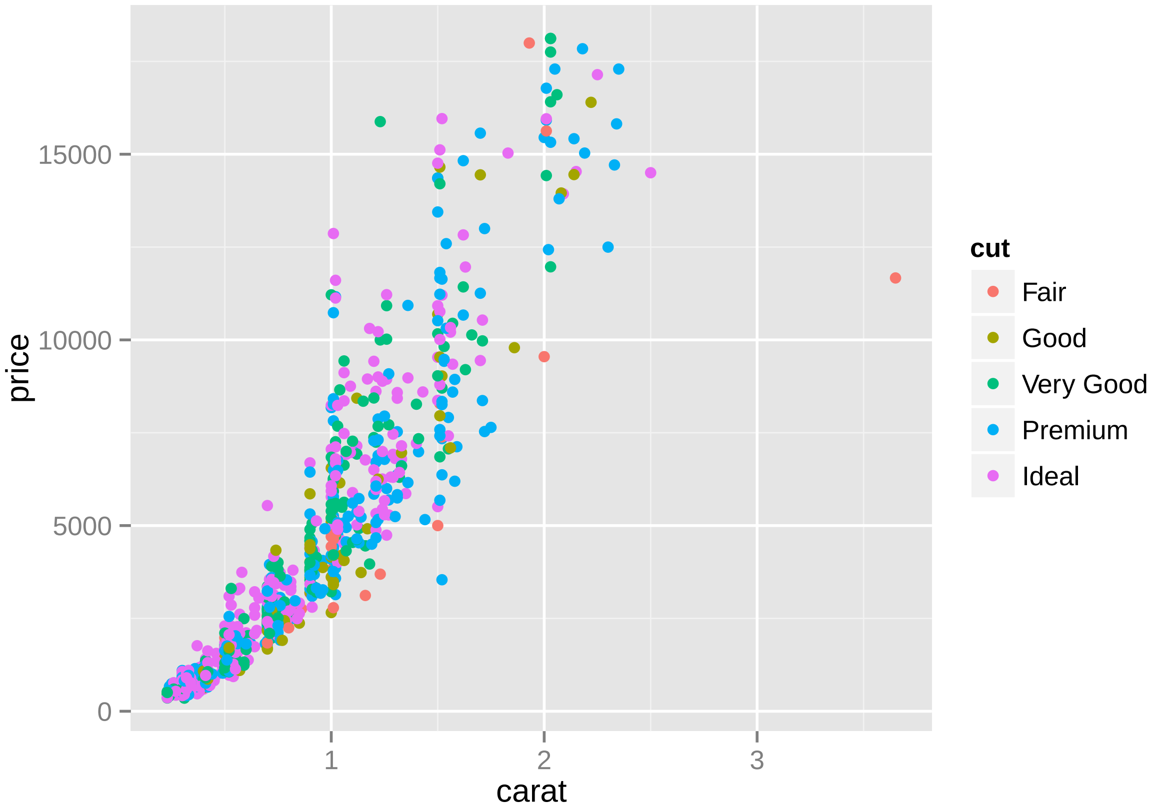
```
ggplot(dsmall, aes(x=carat, y=price)) + geom_point() + geom_smooth(se=FALSE, method="lm")
```




## Two numerical and one categorical

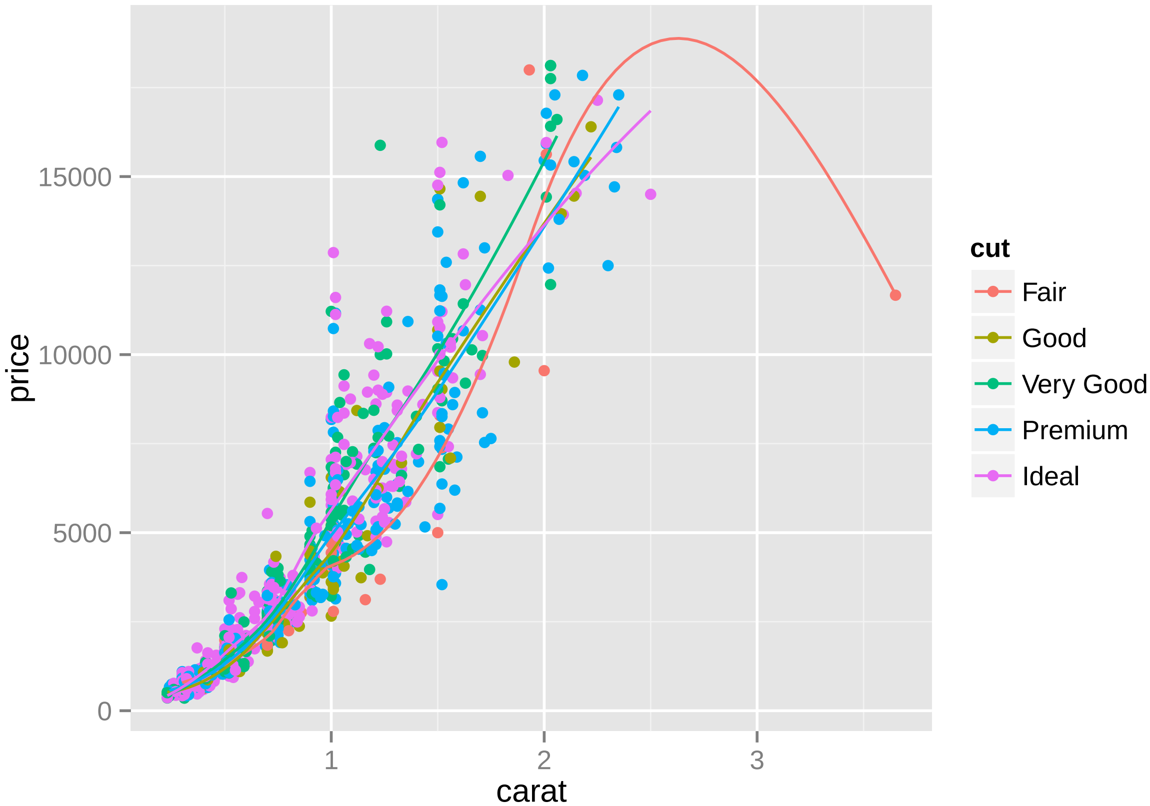
And lastly let's look back at how we can play with scatterplots of using a third categorical variable (using `ggplot2` only). We can color the points by cut,

```
ggplot(dsmall, aes(x=carat, y=price, color=cut)) + geom_point()
```



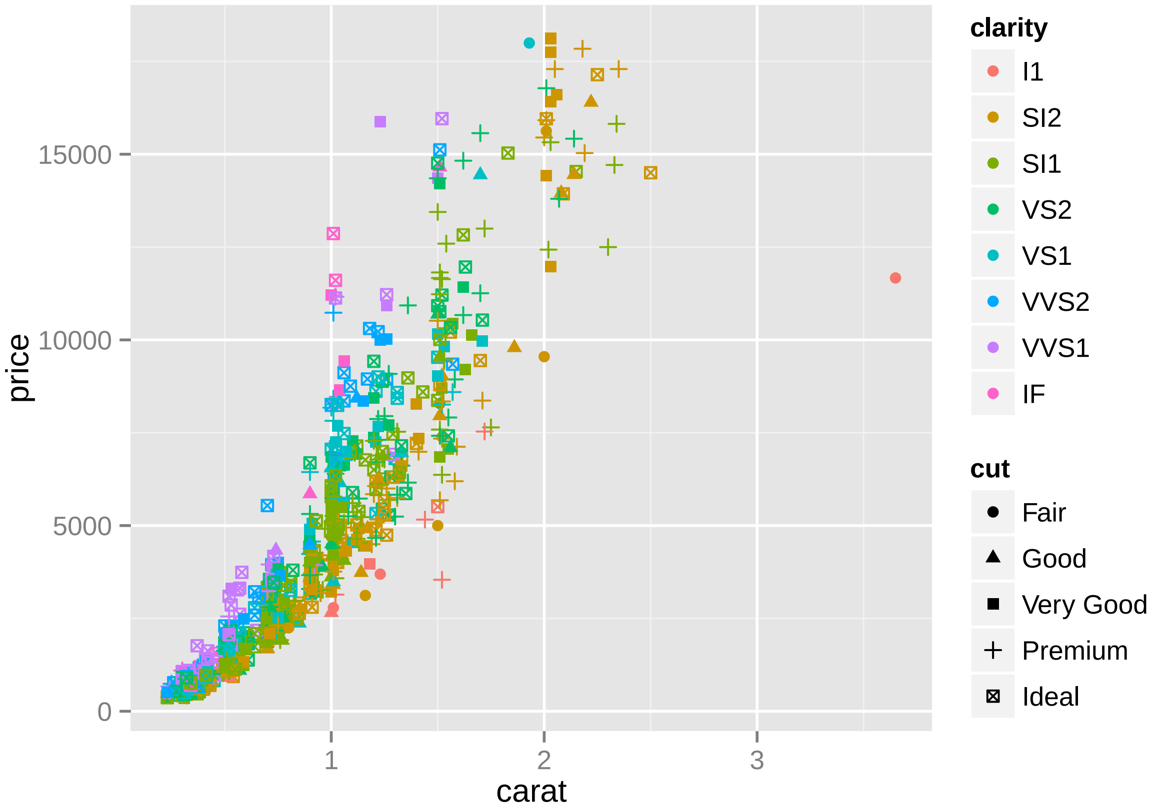

We could add a smoothing lowess line for each cut separately,

```
ggplot(dsmall, aes(x=carat, y=price, color=cut)) + geom_point() + geom_smooth(se=FALSE)
```




We could change the color by clarity, and shape by cut.

```
ggplot(dsmall, aes(x=carat, y=price, color=clarity, shape=cut)) + geom_point()
```




That's pretty hard to read. So note that just because you **can** change an aestetic, doesn't mean you should. And just because you can plot things on the same axis, doesn't mean you have to.

## One numeric and one categorical

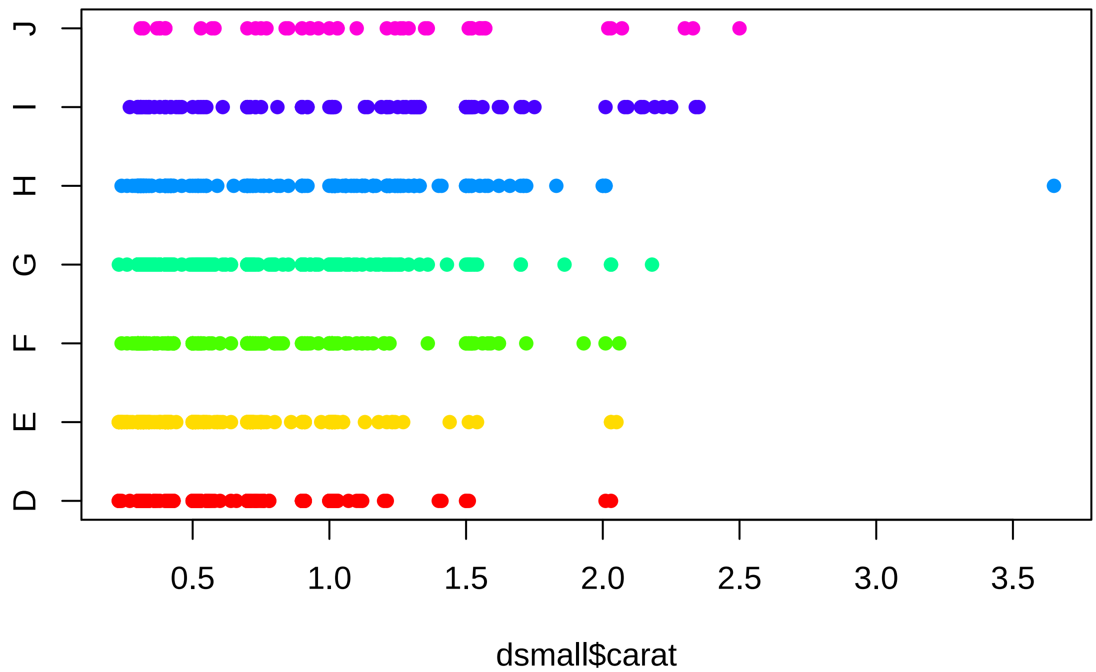
How does the color of a diamond affect the carat?

#### base graphics

#### Stripchart

These plots still only work well for small amounts of data, or if you're plotting summary statitics like the mean. This looks like model notation, but notice that it plots the groups on the y axis and carat on the x.

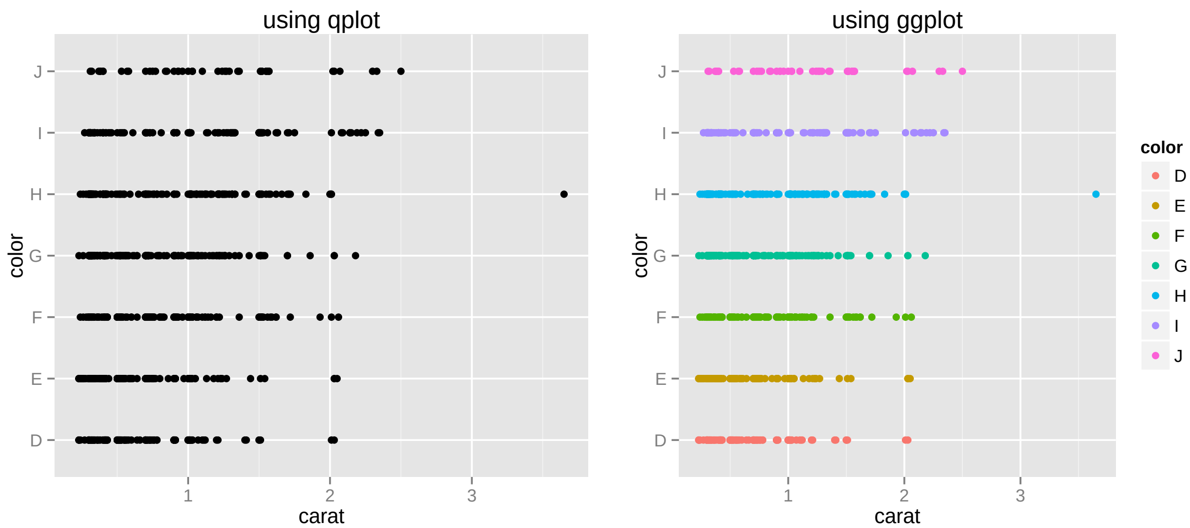
```
stripchart(dsmall$carat ~ dsmall$color, pch=16, col=rainbow(7))
```




Note that the number that goes into the `rainbow()` argumnet is the number of categories that are to be colored. If you put a lower number here the colors will recycle. So you could end up with 2 red strips etc.

**ggplot** Here is an example using `qplot` and one with `ggplot`.

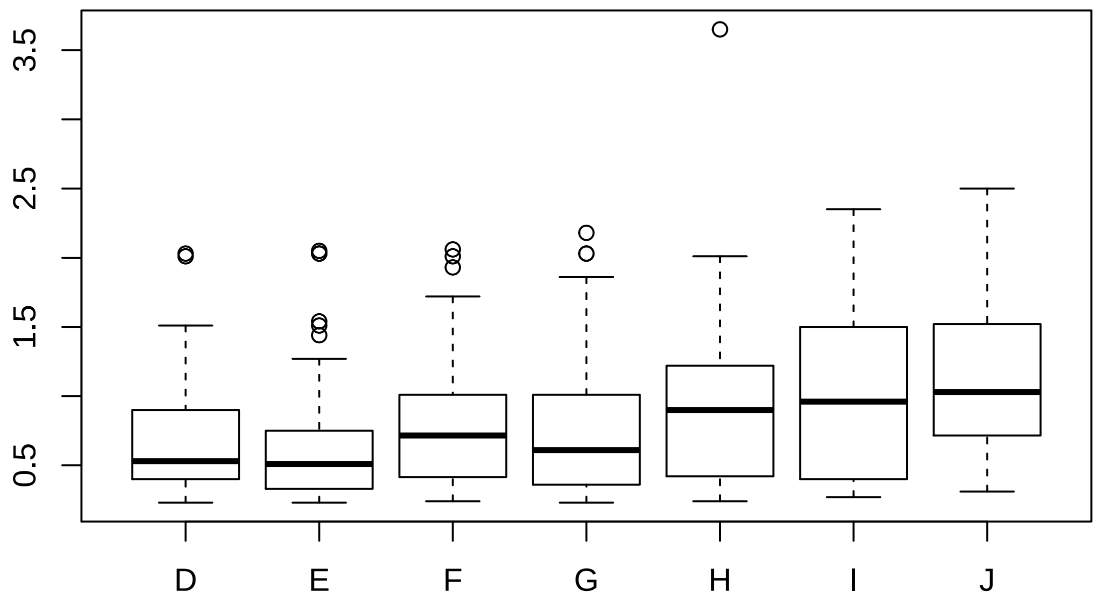
```
a <- qplot(x=carat, y=color, data=dsmall, geom="point", main="using qplot")
b <- ggplot(dsmall, aes(x=carat, y=color, col=color)) + geom_point() + ggtitle("using ggplot")
grid.arrange(a, b, ncol=2)
```




### Grouped boxplots

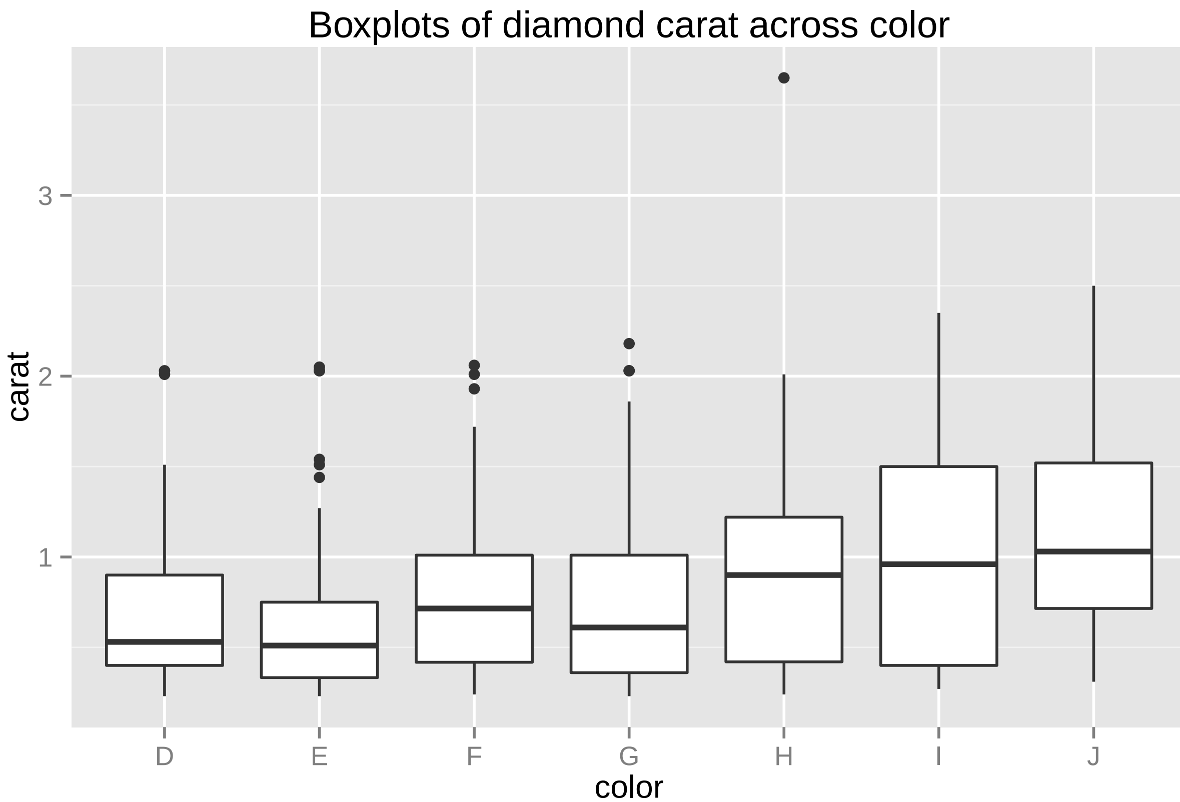
**base graphics** Base graphics plots grouped boxplots with also just the addition of a twiddle ~. Another example of where model notation works.

```
boxplot(carat~color, data=dsmall)
```



**ggplot** A simple addition, just define your x and y accordingly.

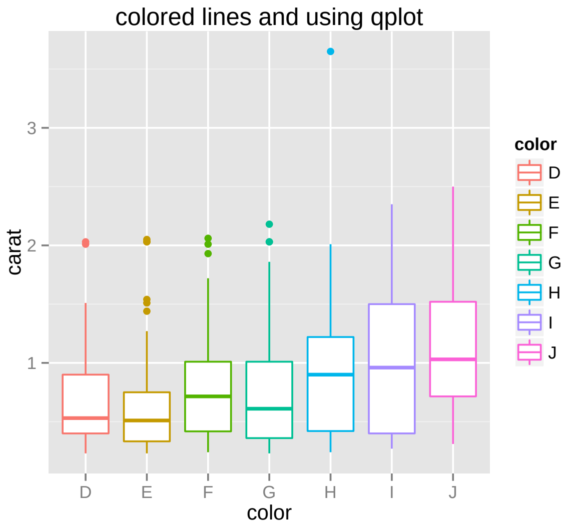
```
qplot(x=color, y=carat, data=dsmall, geom="boxplot", main="Boxplots of diamond carat across color")
```

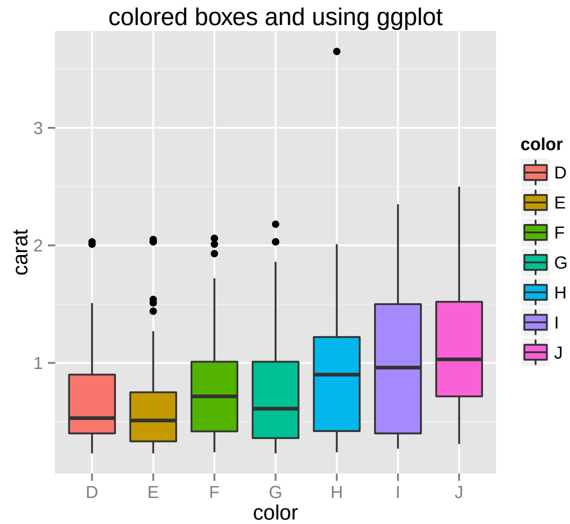



But what about the plotting colors man?

Add the variable you want to control the colors as an argument to color (for the outline) or fill. Using ggplot you have to add these options into the `aes()` statement.

```
cl <- qplot(x=color, y=carat, data=dsmall, geom="boxplot", color=color,
         main="colored lines and using qplot")
ct <- ggplot(dsmall, aes(x=color, y=carat, fill=color)) + geom_boxplot() +
          ggtitle("colored boxes and using ggplot")
grid.arrange(cl, ct, ncol=2)
```



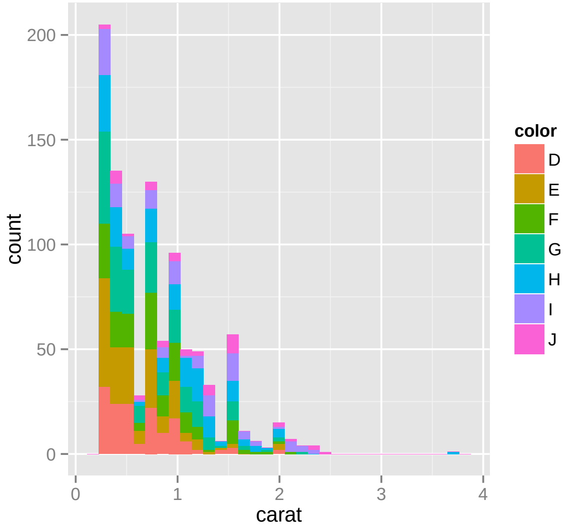



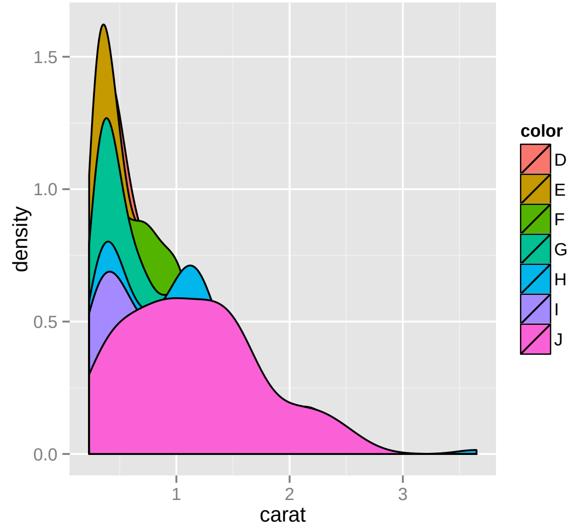

### Grouped histograms

**base graphics** There is no easy way to create grouped histograms in base graphics we will skip it.

**ggplot** By default ggplot wants to overlay all plots on the same grid. This doesn't look to good with histograms. Instead you can overlay density plots

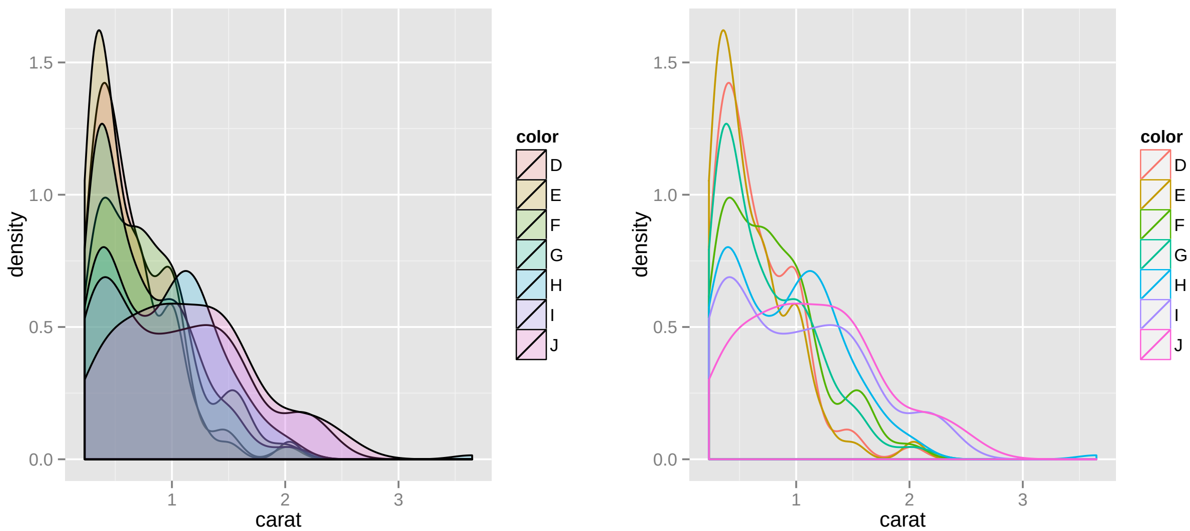
```
a <- qplot(x=carat, data=dsmall, geom="histogram", fill=color)
b <- qplot(x=carat, data=dsmall, geom="density", fill=color)
grid.arrange(a,b, ncol=2)
```




The solid fills are still difficult to read, so we can either turn down the alpha (turn up the transparency) or only color the lines and not the fill.

```
c <- ggplot(dsmall, aes(x=carat, fill=color)) + geom_density(alpha=.2)
d <- ggplot(dsmall, aes(x=carat, col=color)) + geom_density()
grid.arrange(c,d, ncol=2)
```



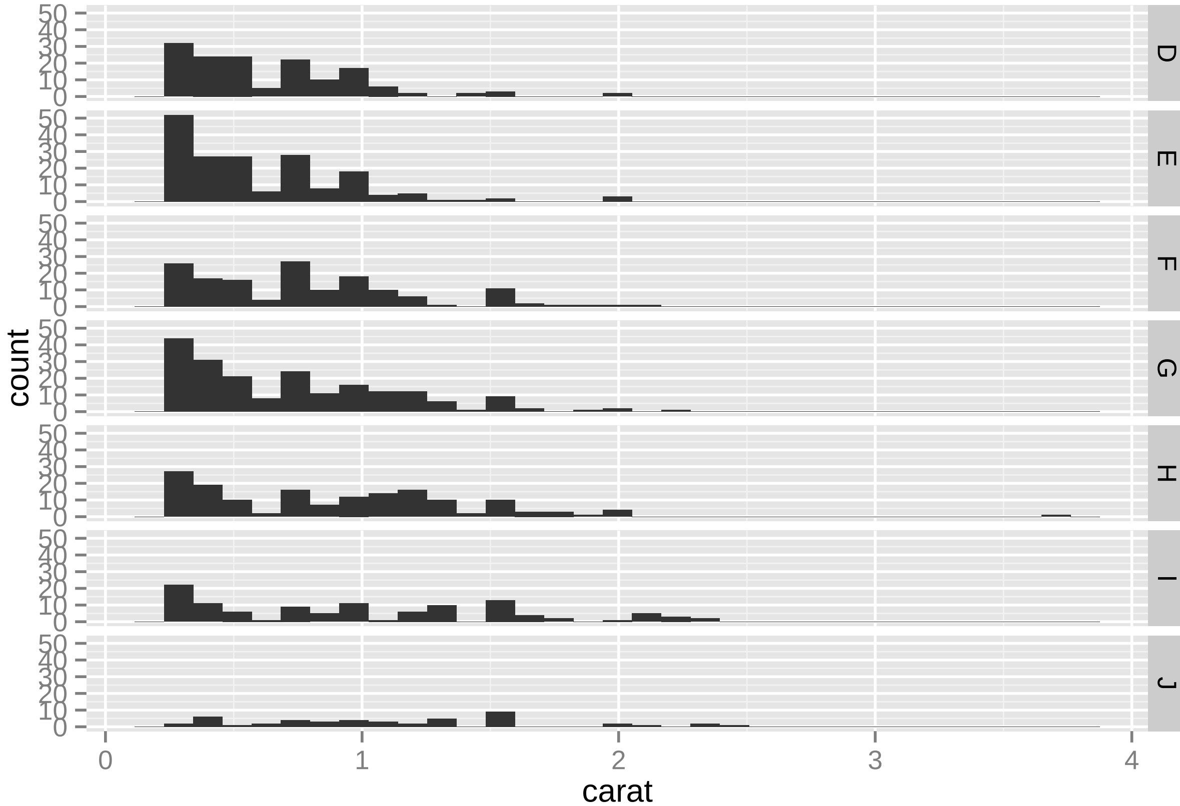

But what if we **really** wanted to compare histograms?

# Faceting / paneling

ggplot introduces yet another term called `faceting`. The definition is *a particular aspect or feature of something*, or *one side of something many-sided, especially of a cut gem*. Basically instead of plotting the grouped graphics on the same plotting area, we let each group have it's own plot, or facet.

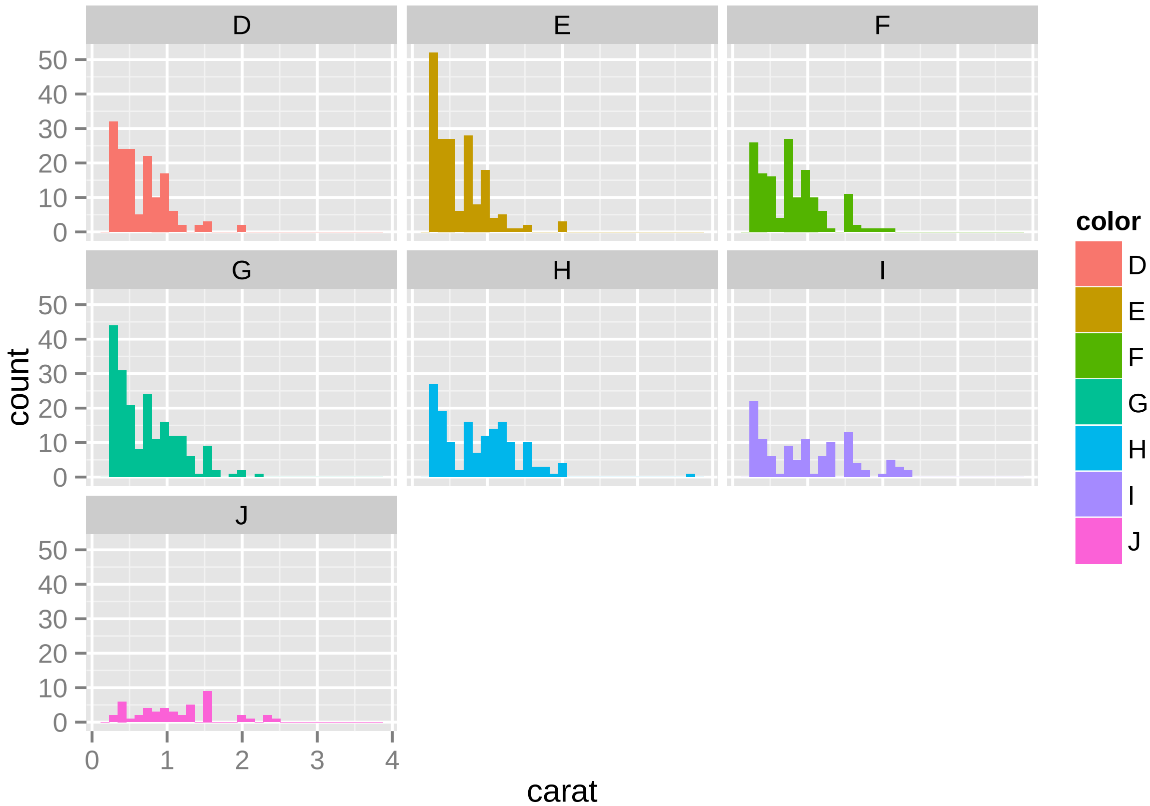
Using `qplot` is a nice default. It is important to compare distriubtions across groups on the same scale, and our eyes can compare items vertically better than horizontally.

```
qplot(x=carat, data=dsmall, geom="histogram", facets=color ~.)
```



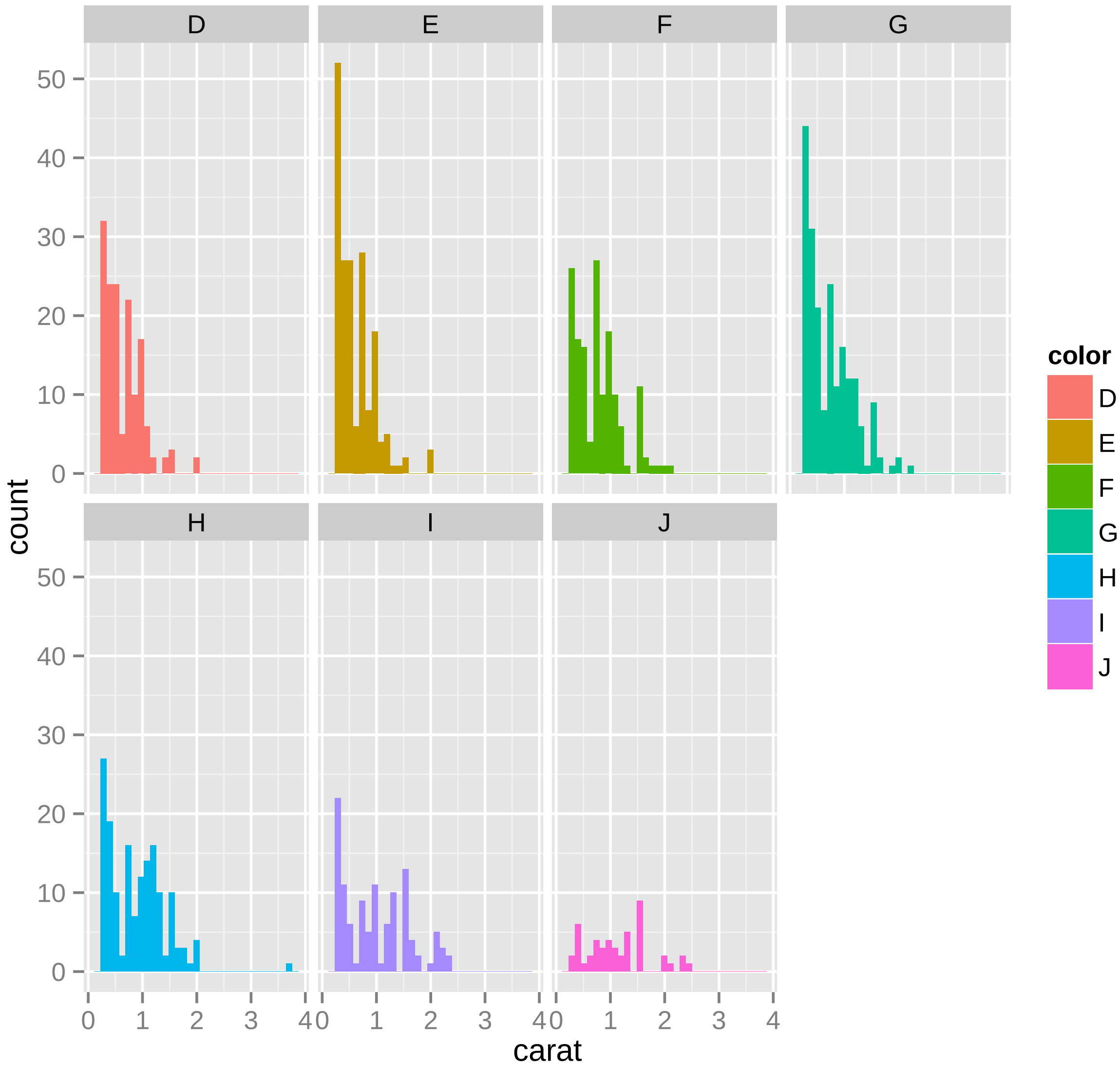

If we use the ggplot function, after adding a geom_histogram we add a facet_wrap() and specify that we want to wrap on the color group. Note the twiddle in front of color and no period.

```
ggplot(dsmall, aes(x=carat, fill=color)) + geom_histogram() + facet_wrap(~color)
```



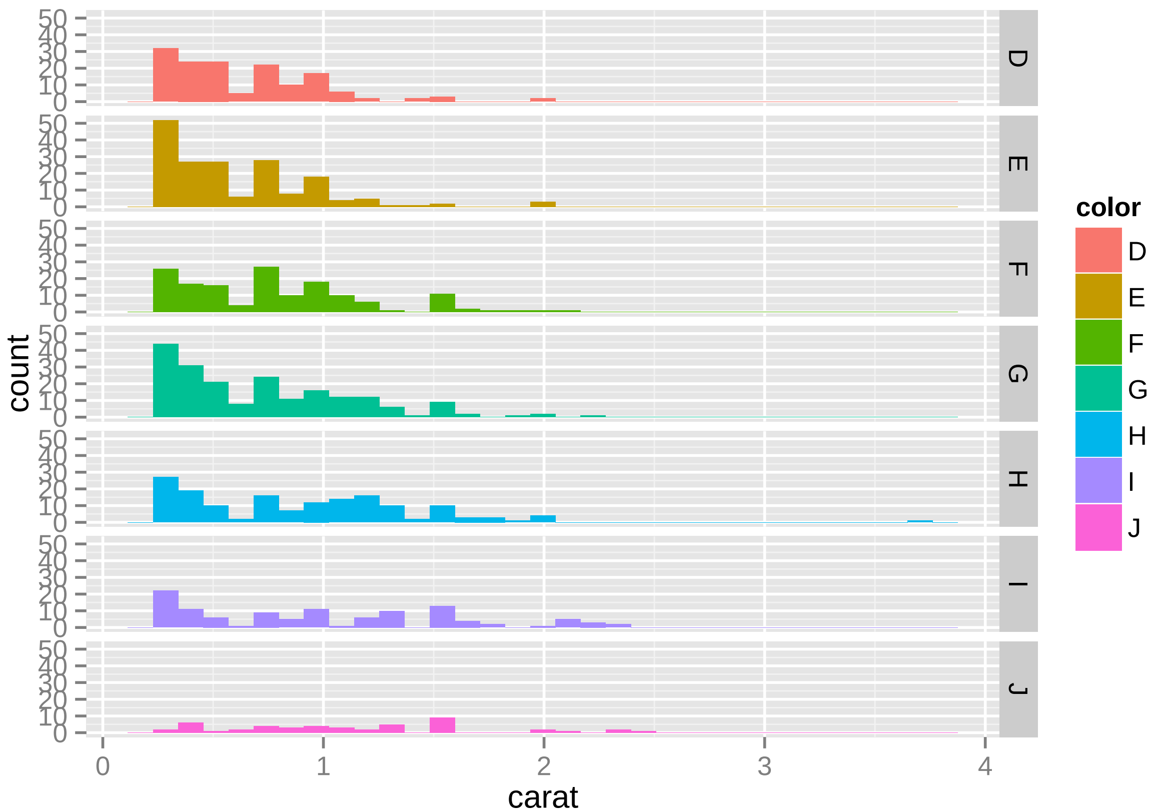

The grid placement can be semi-controlled by using the `ncol` argument in the facet_wrap() statement.

```
ggplot(dsmall, aes(x=carat, fill=color)) + geom_histogram() + facet_wrap(~color, ncol=4)
```

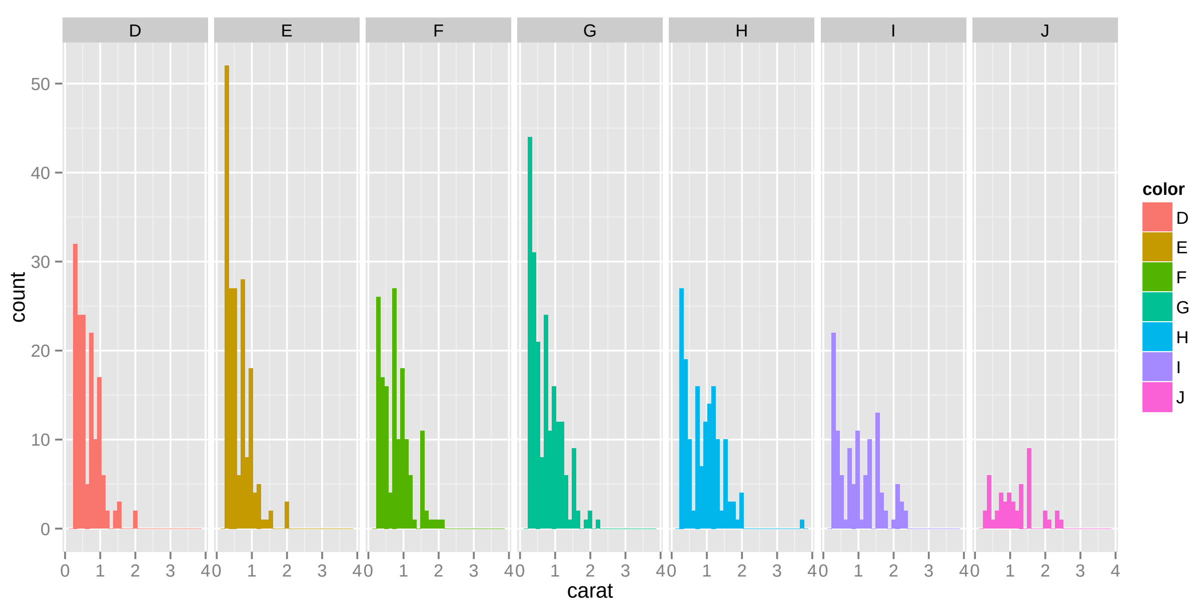



But what did i just say about not being able to compare distributions easily when they're horizontal? No problem, use `facet_grid()` instead. Look at the difference when the `~.` is on the left, and right hand side of the paneling variable.

```
ggplot(dsmall, aes(x=carat, fill=color)) + geom_histogram() + facet_grid(color~.)
```



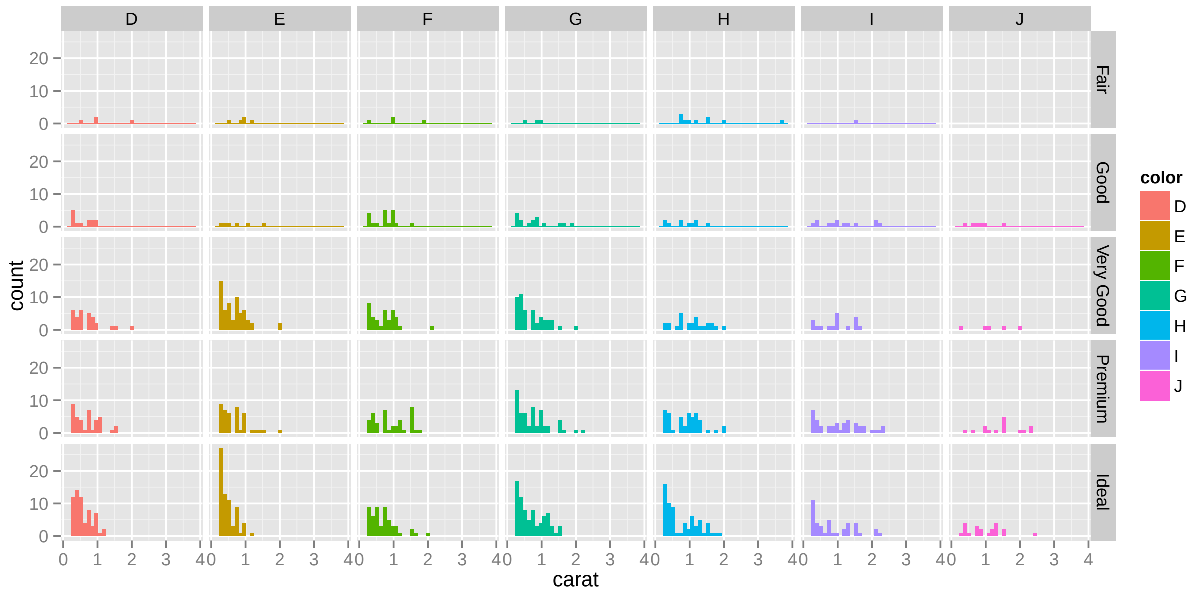

```
ggplot(dsmall, aes(x=carat, fill=color)) + geom_histogram() + facet_grid(.~color)
```



### Paneling on two variables

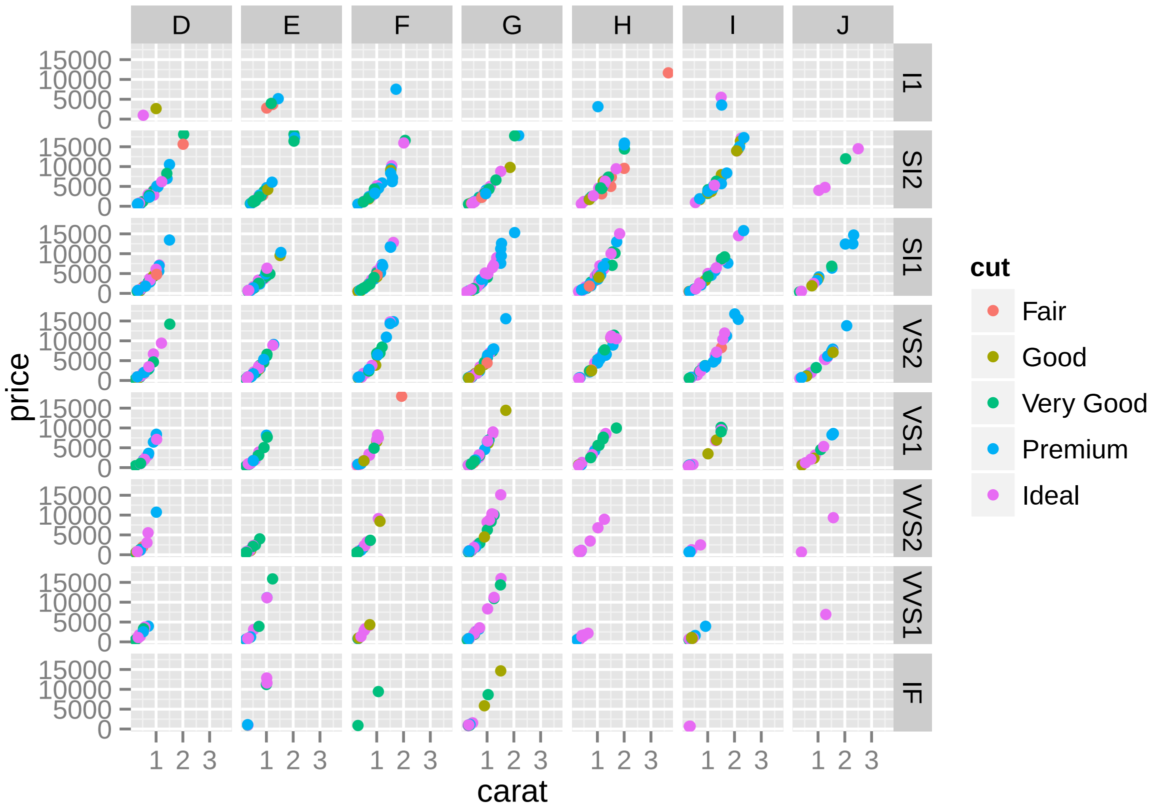
Who says we're stuck with only faceting on one variable?

```
ggplot(dsmall, aes(x=carat, fill=color)) + geom_histogram() + facet_grid(cut~color)
```




How about plotting price against caret, for all combinations of color and clarity, with the points further separated by cut?

```
ggplot(dsmall, aes(x=carat, y=price, color=cut)) + geom_point() + facet_grid(clarity~color)
```




Before you share your plot with any other eyes, always take a step back and try to expain what it is telling you. If you have to take more than a minute to get to the point then it may be too complex and simpler graphics are likely warranted.

## Additional Resources

- R Graphics Cookbook: http://www.cookbook-r.com/Graphs/ or http://amzn.com/1449316956 **Highly Highly Recommended**
- Quick-R: Basic Graphs
- Quick-R: ggplot2
- Books
  - ggplot2 http://ggplot2.org/book/ or http://amzn.com/0387981403
  - qplot http://ggplot2.org/book/qplot.pdf
- ggplot2 mailing list http://groups.google.com/group/ggplot2
- stackoverflow http://stackoverflow.com/tags/ggplot2