

# Installing Packages in R

STAT2005

Chapter 7

# Packages in R

This chapter gives some information about the packages contained in R when R is installed and invoked.

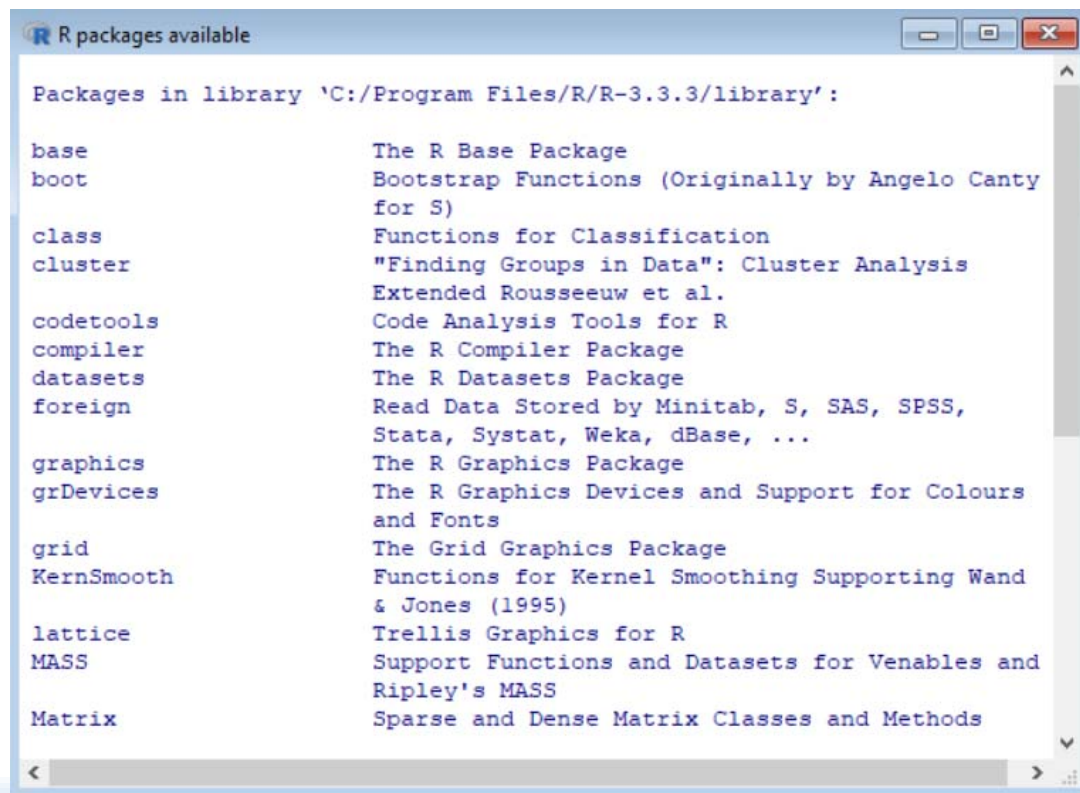
Furthermore, there are lots of contributed packages developed by R programmers all over the world.

We will demonstrate how to download and install these contributed packages from the R's official website.

# Pre-installed packages in the R library

When R is installed in the computer, there are some packages came with it.

You can look up the information of those packages installed by using the command `library()`:



```
R packages available

Packages in library 'C:/Program Files/R/R-3.3.3/library':

base           The R Base Package
boot           Bootstrap Functions (Originally by Angelo Canty
              for S)
class          Functions for Classification
cluster        "Finding Groups in Data": Cluster Analysis
              Extended Rousseeuw et al.
codetools      Code Analysis Tools for R
compiler       The R Compiler Package
datasets       The R Datasets Package
foreign        Read Data Stored by Minitab, S, SAS, SPSS,
              Stata, Systat, Weka, dBase, ...
graphics       The R Graphics Package
grDevices      The R Graphics Devices and Support for Colours
              and Fonts
grid           The Grid Graphics Package
KernSmooth     Functions for Kernel Smoothing Supporting Wand
              & Jones (1995)
lattice        Trellis Graphics for R
MASS           Support Functions and Datasets for Venables and
              Ripley's MASS
Matrix         Sparse and Dense Matrix Classes and Methods
```

These packages contain functions (or datasets) that are commonly used in computer programming, mathematics and statistics.

When R is launched, the base package is loaded to the memory space automatically.

To see a full list of functions inside this base package, use the command `library(help=base)`.

It opens a new window which contains a long list of functions contained in the base package. Many of these functions are the built-in functions we learned in the previous chapters.

For example, we can use the command `help(matrix)` to display the help manual of the function `matrix()`.

# Installing additional packages

To install new packages.

- Click `Packages -> install package(s)` from R's menu.
- A window with a list of CRAN mirror site appears.
- Choose a server that is close to your location.
- A new window contains a very long list of packages appear. These packages are listed in alphabetical order.

Alternatively, we can use the following command to install a particular package.

```
install.packages("package_name")
```

# The sas7bdat package

Let us try to download the sas7bdat package.

Once the sas7bdat package is downloaded, enter `library(sas7bdat)` to load the package, and enter `library(help=sas7bdat)` to display the functions in sas7bdat.

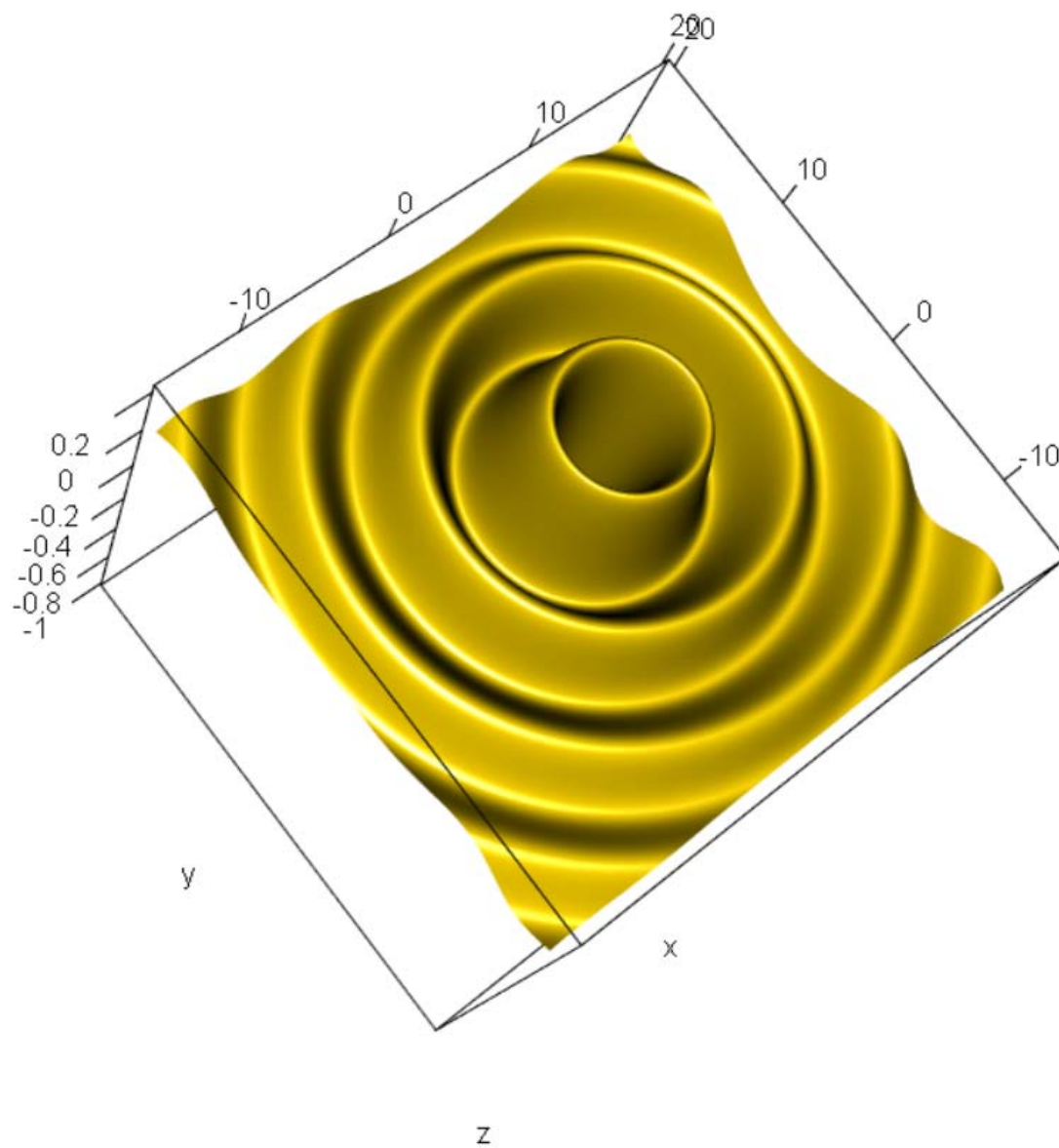
Next, we want to read in the SAS data set `country.sas7bdat`, we can run

```
x <- read.sas7bdat("country.sas7bdat")
```

# The rgl package

The `rgl` package provides medium to high level functions for 3D interactive graphics, output on screen using OpenGL.

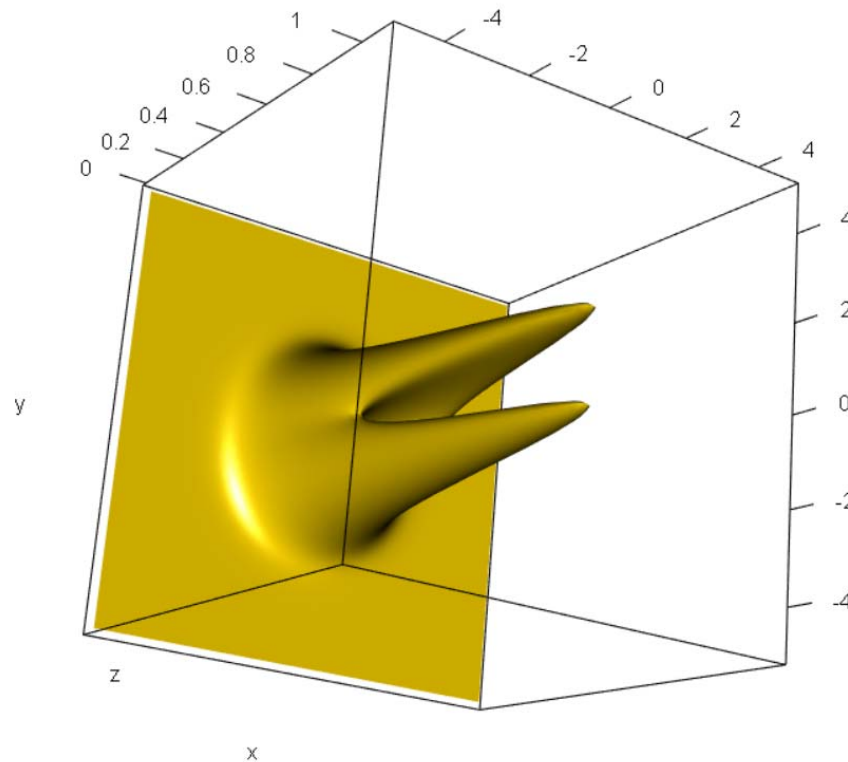
```
library(rgl)
fxy<-function(x,y) {
  p<-sqrt((x-5)^2+(y-5)^2)
  -sin(p)/p }
x<-seq(-15,20,by=0.1)
y<-x
z<-outer(x,y,fxy)
persp3d(x,y,z,theta=-30, phi=30, col="gold",
ticktype="detailed")
```



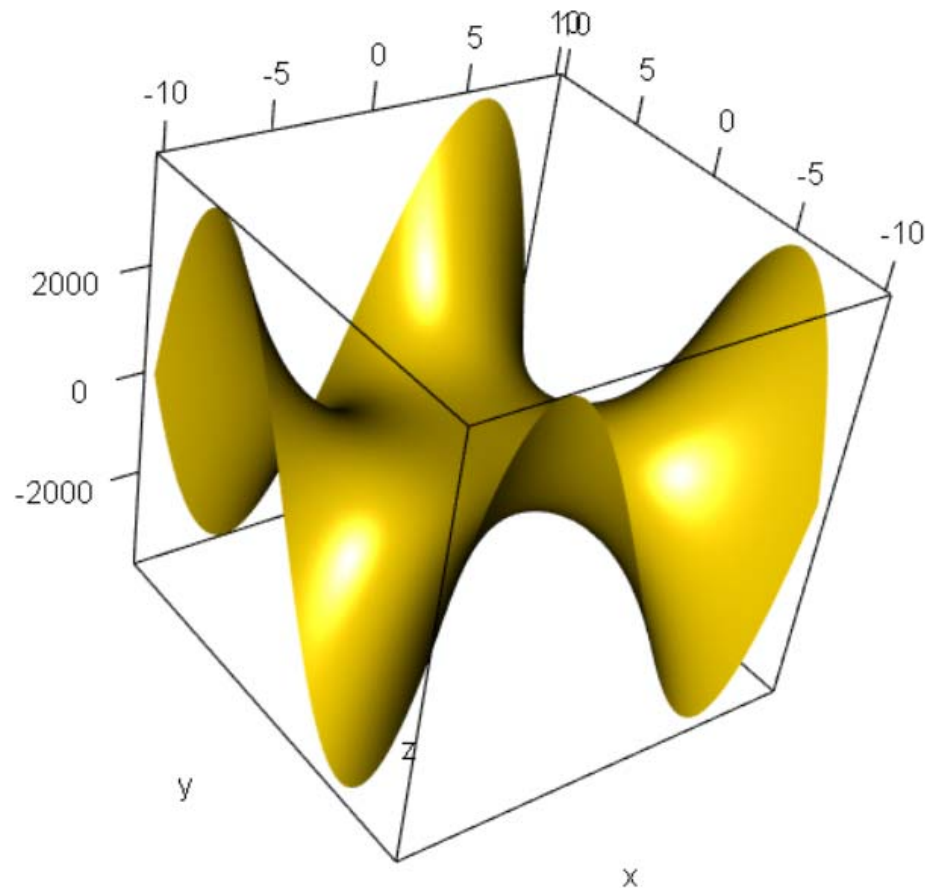


# Some interesting 3D plots

```
fx<-function(x,y) {(x^2+3*y^2)*exp(-x^2-y^2)}  
x<-seq(-5,5,by=0.1); y<-x; z<-outer(x,y,fx)  
persp3d(x,y,z, col="gold", ticktype="detailed")
```



```
fx<-function(x,y) {x*y^3-y*x^3}  
x<-seq(-10,10,by=0.1); y<-x; z<-outer(x,y,fx)  
persp3d(x,y,z, col="gold", ticktype="detailed")
```



```

fxy<-function(x,y) {((1-sign(-x-
.9+abs(y*2)))/3*(sign(.9-x)+1)/3)*(sign(x+.65)+1)/2 -
((1-sign(-x-.39+abs(y*2)))/3*(sign(.9-x)+1)/3) + ((1-
sign(-x-.39+abs(y*2)))/3*(sign(.6-x)+1)/3)*(sign(x-
.35)+1)/2}

```

```

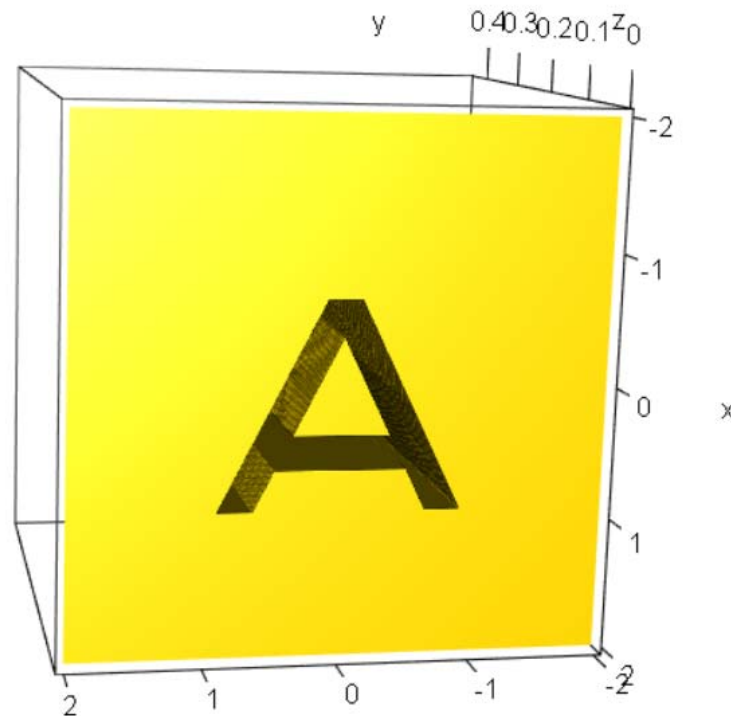
x<-seq(-2,2,by=0.005); y<-x; z<-outer(x,y,fxy)

```

```

persp3d(x,y,z, col="gold", ticktype="detailed")

```



# Torus (Doughnut)

Parameterization of a torus may be achieved with these equations

$$x = F(\theta_2; \alpha, \beta) \cos \theta_1 ,$$

$$y = F(\theta_2; \alpha, \beta) \sin \theta_1 ,$$

$$z = \alpha \sin \theta_2$$

where

$$F(\theta_2; \alpha, \beta) = \beta + \alpha \cos \theta_2$$

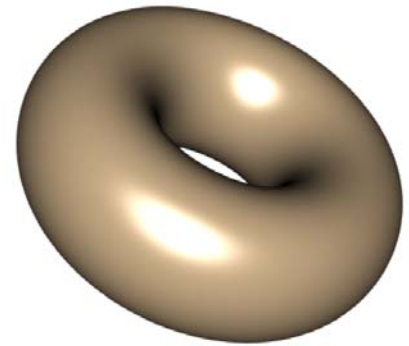
with  $0 \leq \theta_1, \theta_2 < 2\pi$ .

The constants  $\alpha, \beta$  control the radius of the "tube" and the overall size of the torus.

```

library(rgl)
res <- 200
theta <- seq(0,2*pi,length=res)
alpha <- 1
beta <- 2
xm <- outer(theta,theta,
             function(t1,t2) (beta+alpha*cos(t2))*cos(t1)
             )
ym <- outer(theta,theta,
             function(t1,t2) (beta+alpha*cos(t2))*sin(t1)
             )
zm <- outer(theta,theta,
             function(t1,t2) alpha*sin(t2)
             )
persp3d(x=xm,y=ym,z=zm,
        col="tan",aspect=c(1,1,0.4),
        axes=FALSE,xlab=" ",ylab=" ",zlab=" ")

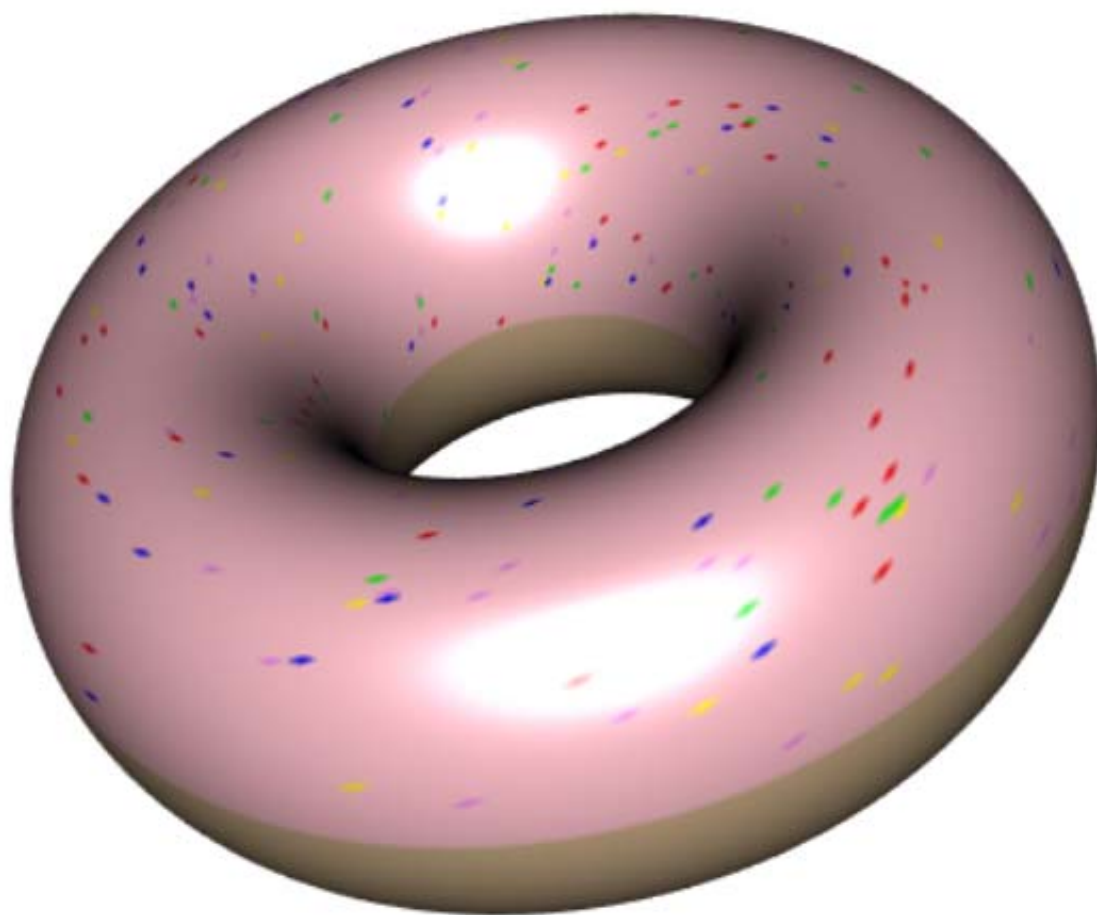
```



# Decorating the doughnut

To add sprinkles, we need a mechanism to identify some random locations on the top half of the surface and color them appropriately.

```
donutcols <- rep("tan",res^2)
donutcols[as.vector(zm)>0] <- "pink"
sprinkles <- c("blue","green","red","violet","yellow")
donutcols[sample(
  x=which(as.vector(zm)>0),size=300
)] <- sprinkles
persp3d(xm,ym,zm,
  col=donutcols,aspect=c(1,1,0.4),
  axes=FALSE,xlab=" ",ylab=" ",zlab=" "
)
```



# The quantmod package

The `quantmod` package for R is designed to assist the quantitative trader in the development, testing, and deployment of statistically based trading models.

It offers charting facilities that is not available elsewhere in R. The `quantmod` package makes financial modelling easier and analysis simple.

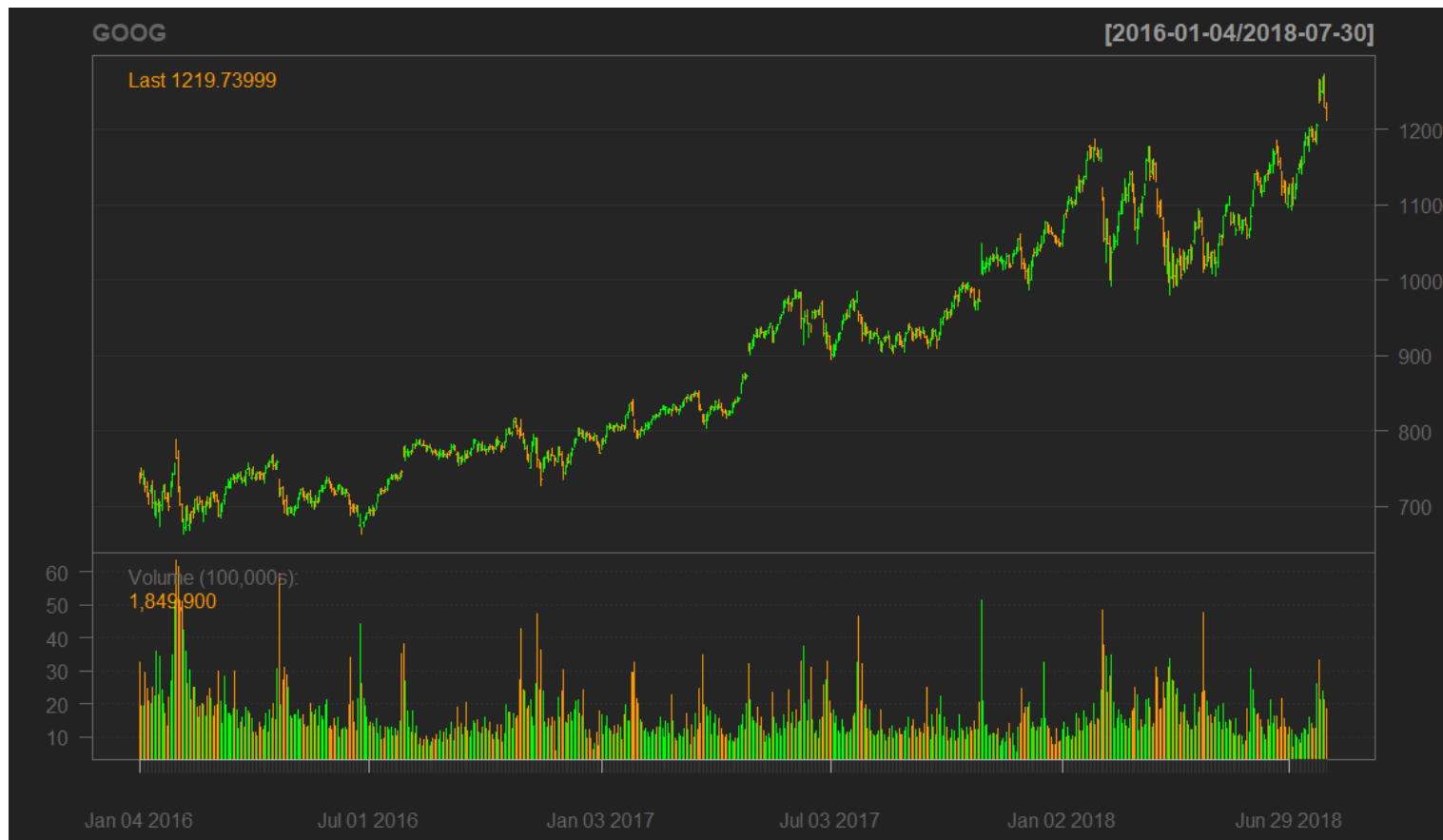
For example, to read in HSBC stock price from January to March of 2017 from Yahoo! Finance, one can use

```
library(quantmod)
stk_price<-getSymbols("0005.HK", from="2017-01-01", to="2017-03-31", src="yahoo",
auto.assign=FALSE)
```



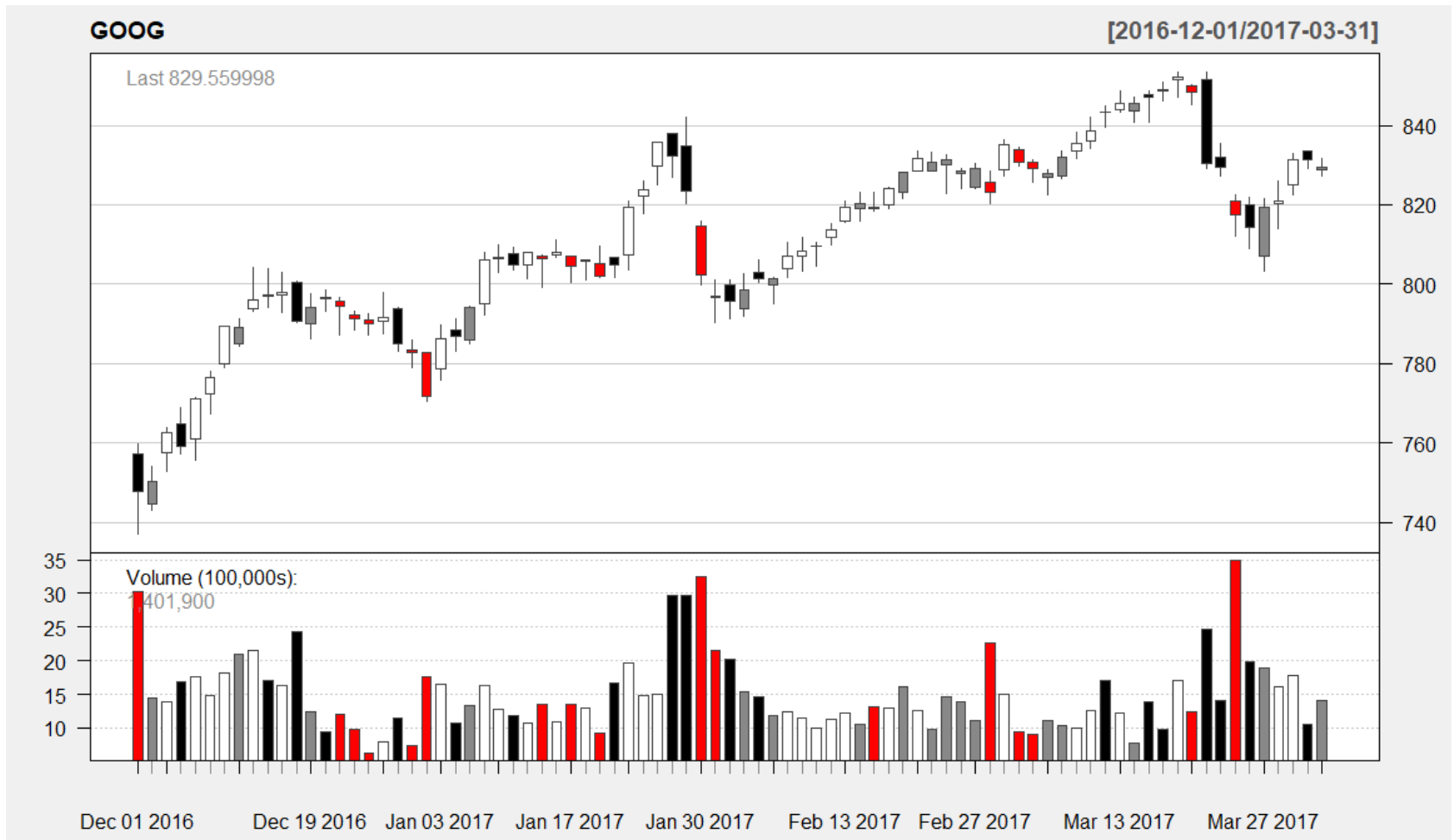
# Example: Plotting financial data

```
getSymbols("GOOG", from="2016-01-01", to="2018-07-31")  
barChart(GOOG)
```

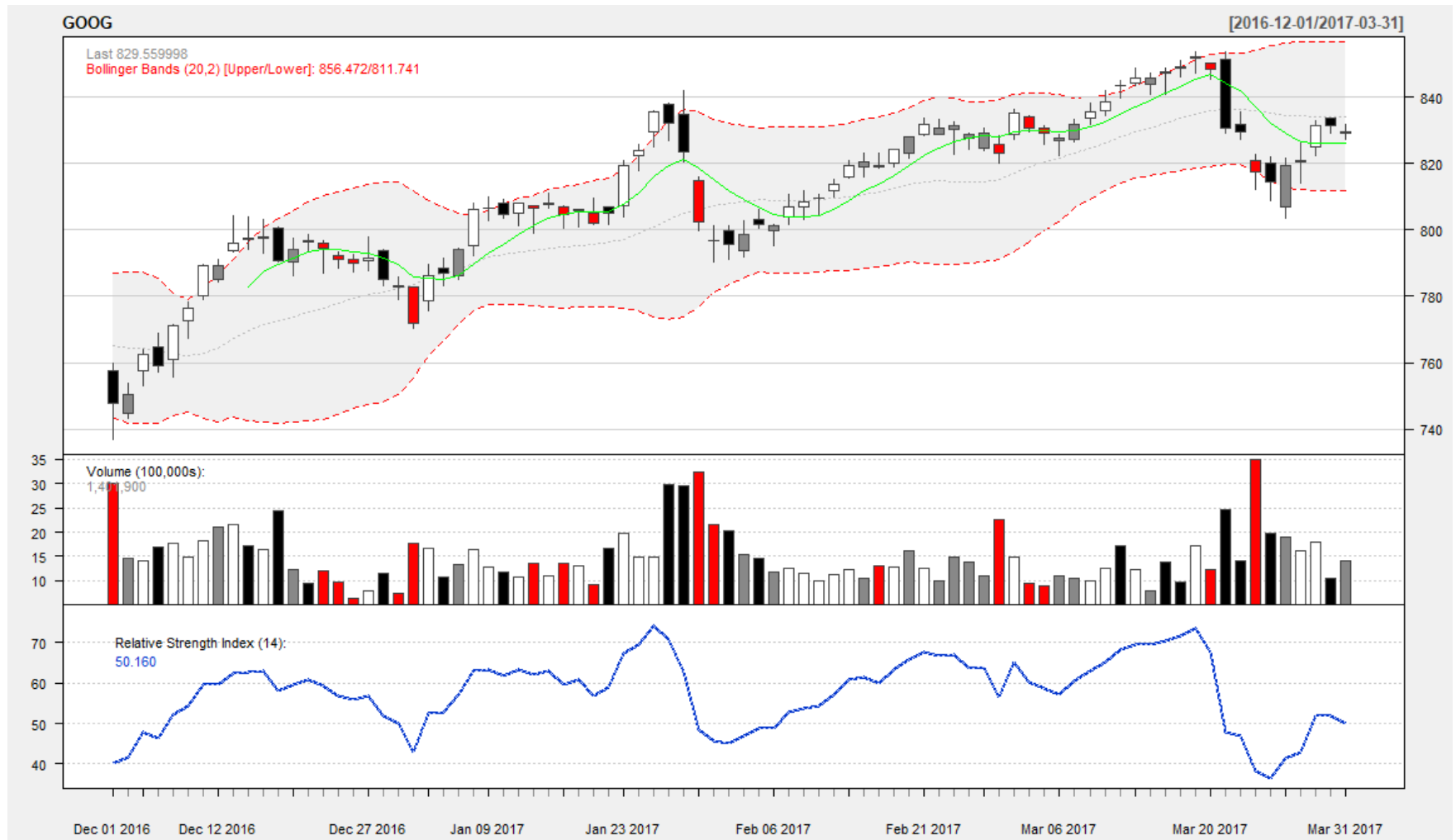


```
# Add multi-coloring and change background to white
```

```
candleChart(GOOG, multi.col=TRUE, theme='white',  
subset='2016-12::2017-03')
```



`addBBands()`      # Bollinger Bands  
`addWMA()`        # Weighted Moving Average  
`addRSI()`        # Relative Strength Indicator



# Technical indicators

Indicator	quantmod Name
Bollinger Bands	addBBands
Chaiken Money Flow	addCMF
Double Exponential Moving Average	addDEMA
Exponential Moving Average	addEMA
Exponential Volume Weigthed Moving Average	addEVWMA
Options and Futures Expiration	addExpiry
Moving Average Convergence Divergence	addMACD
Momentum	addMomentum
Rate of Change	addROC
Relative Strength Indicator	addRSI
Simple Moving Average	addSMA
Stochastic Momentum Index	addSMI
Volume	addVo
Weighted Moving Average	addWMA

# Tidyverse packages

*Tidyverse* is a collection of R packages designed to follow a consistent design philosophy.

Some of the well known packages include

- `ggplot2`: graphics system based on grammar of graphics
- `tibble`: a dataframe improvement
- `readr`: reading data into the tidyverse
- `stringr`: manipulating strings
- `dplyr`: manipulating data sets

# The tidyverse principles

Hadley Wickham is the principal author behind the tidyverse. He has written down his design principles, which can be accessed by running

```
vignette("manifesto", package = "tidyverse")
```

A summary of these principles is given as follows.

## Re-use existing data structures

We should place a high value on consistency: focus on a small number of structures that are already familiar to users, rather than inventing new ones.

## Compose simple functions with the pipe

The pipe is the `%>%` pipe operator that to pass data between functions.

The idea behind `%>%` is fairly simple:

rather than typing a function call as  $f(x, y)$ , it is typed as  $x \%>\% f(y)$ .

We can read this as *start with  $x$ , and use  $f(y)$  to modify it*.

This sometimes leads to more readable code than the equivalent standard notation.

### *Example*

```
round(exp(diff(log(x))), 1)
```

can be written as

```
x %>% log() %>%  
  diff() %>%  
  exp() %>%  
  round(1)
```

## Embrace functional programming

Functional style means decomposing a big problem into smaller pieces, then solving each piece with a function or combination of functions. Each function by itself is simple and easy to understand; complexity is handled by composing functions in various way.

## Design for humans

Computational efficiency should be a secondary concern to usability



# The ggplot2 package

The ggplot2 package is a very popular alternative to R's base graphics package. It is based on a “grammar of graphics” concept (and hence the name ggplot2):

Independently specify plot building blocks and combine them to create just about any kind of graphical display you want. Building blocks of a graph include:

- The **data** that you want to visualise and a set of aesthetic **mappings** describing how variables in the data are mapped to aesthetic attributes that you can perceive.
- Geometric objects, **geoms** for short, represent what you actually see on the plot: points, lines, polygons, etc.

- Statistical transformations, **stats** for short, summarise data in a useful ways.
  - For example, binning and counting observations to create a histogram, or summarising a 2D relationship with a linear model.
- The **scales** map values in the data space to values in an aesthetic space, whether it be colour, or size, or shape. Scales draw a legend or axes, which provide an inverse mapping to make it possible to read the original data values from the graph.

- A coordinate system, **coord** for short, describes how data coordinates are mapped to the plane of the graphic. It also provides axes and gridlines to make it possible to read the graph.
  - We normally use a Cartesian coordinate system, but a number of others are available, including polar coordinates and map projections.
- A **faceting** specification describes how to break up the data into subsets and how to display those subsets as small multiples in a multi-frame plot. This is also known as conditioning or latticing.

# The `qplot()` function

`qplot()` is a function for quick plot in the `ggplot2` package, it has been designed to be very similar to the built-in `plot()` function.

Its general syntax is

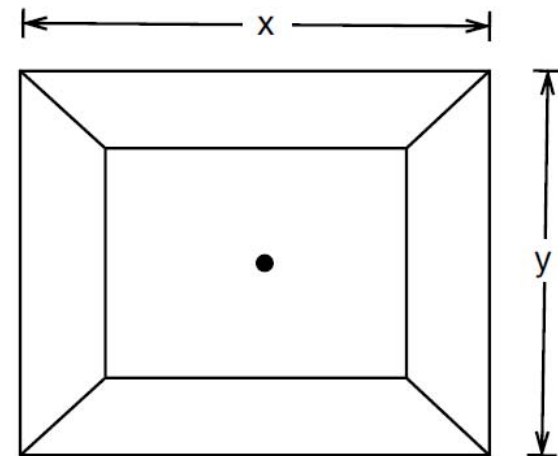
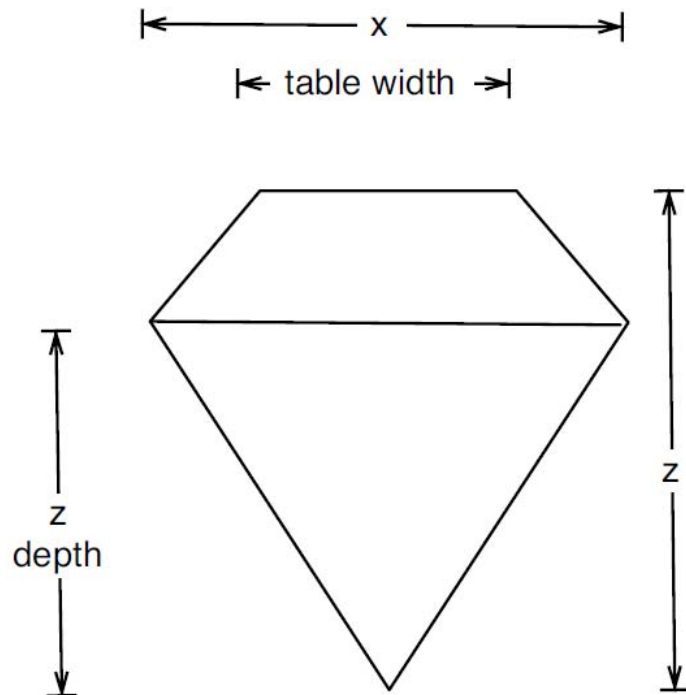
```
qplot(  
  x = , y = , ...,  
  data = , facets = NULL, margins = FALSE,  
  geom = "auto", xlim = c(NA, NA),  
  ylim = c(NA, NA), alpha = , main = NULL,  
  xlab = deparse(substitute(x)),  
  log = "", ylab = deparse(substitute(y)),  
  asp = NA, stat = NULL, position = NULL  
)
```

# Example

Consider the `diamonds` dataset included in the `ggplot2` package.

It consists of prices and quality information about 54,000 diamonds, which includes

- The four C's of diamond quality, carat, cut, colour and clarity; and
- Five physical measurements, depth, table, x, y and z, as described



$$\text{depth} = z \text{ depth} / z * 100$$

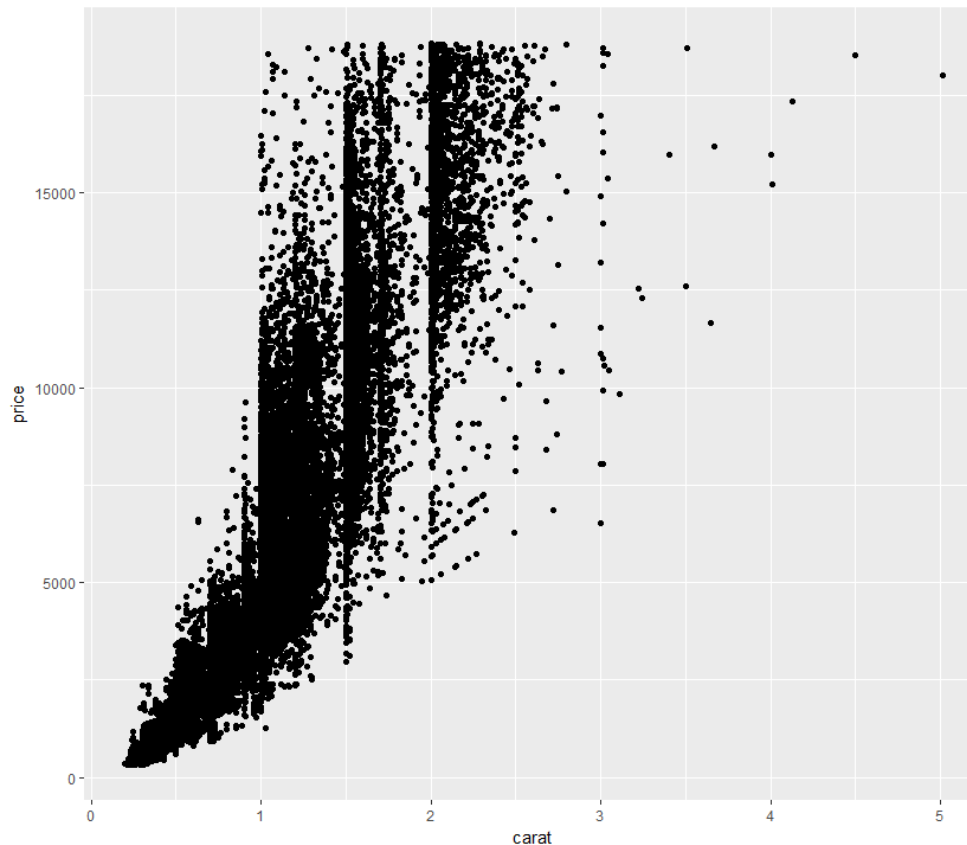
$$\text{table} = \text{table width} / x * 100$$

```
library(ggplot2)
```

```
set.seed(1410)
```

```
dsmall <- diamonds[sample(nrow(diamonds), 100), ]
```

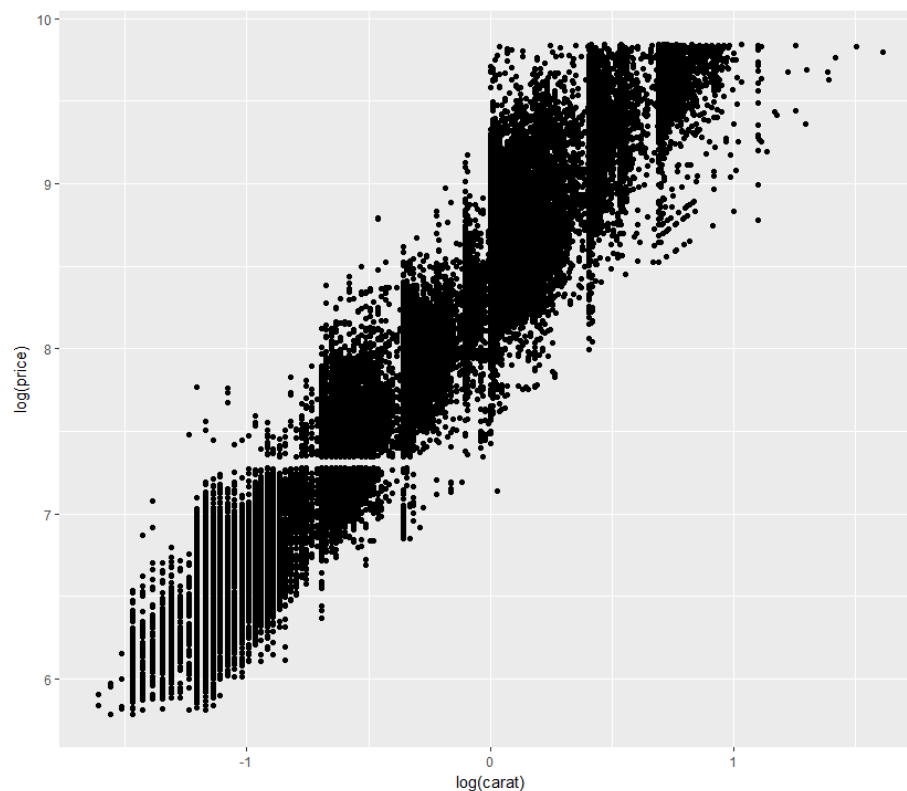
```
ggplot(carat, price, data = diamonds)
```



The plot shows a strong correlation with notable outliers and some interesting vertical striation.

The relationship looks exponential, so the first thing we'd like to do is to transform the variables.

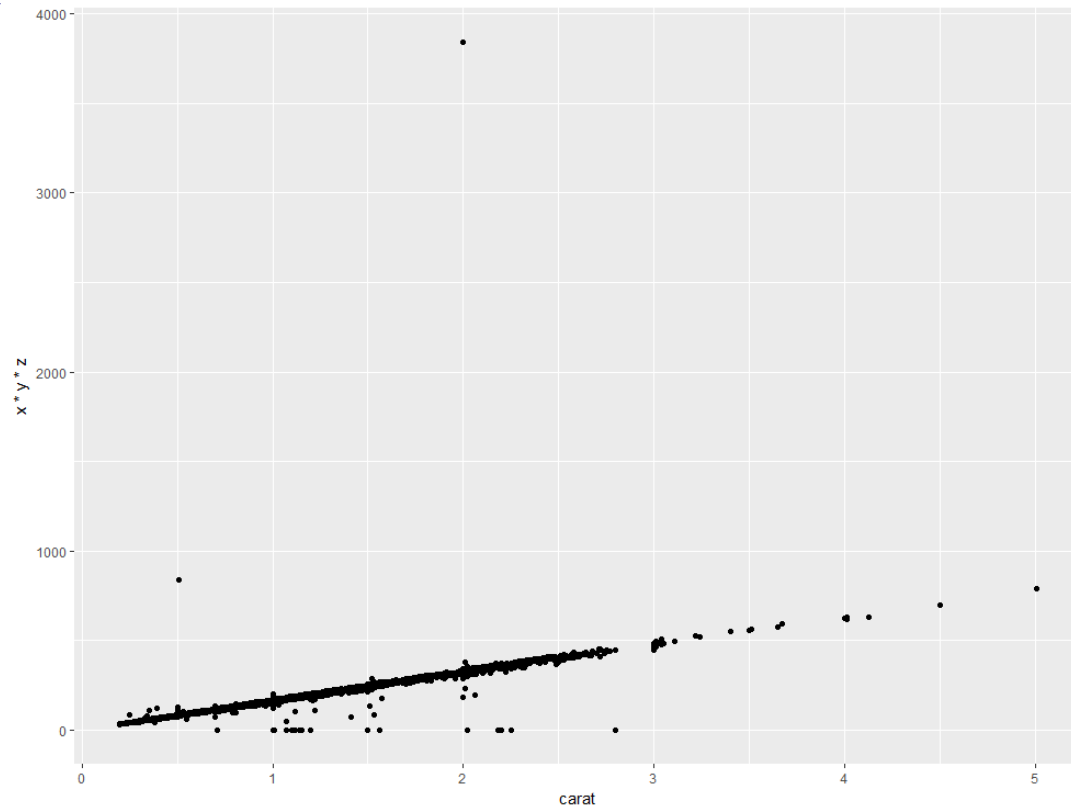
```
ggplot(log(carat), log(price), data = diamonds)
```





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

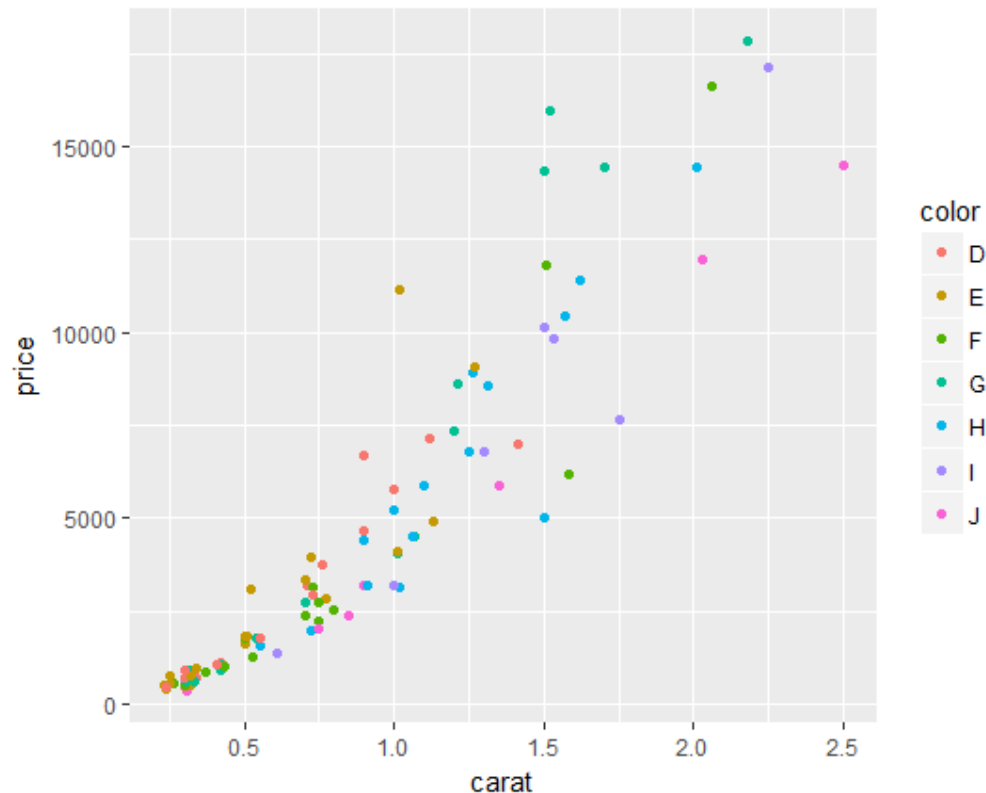
```
qplot(carat, x * y * z, data = diamonds)
```



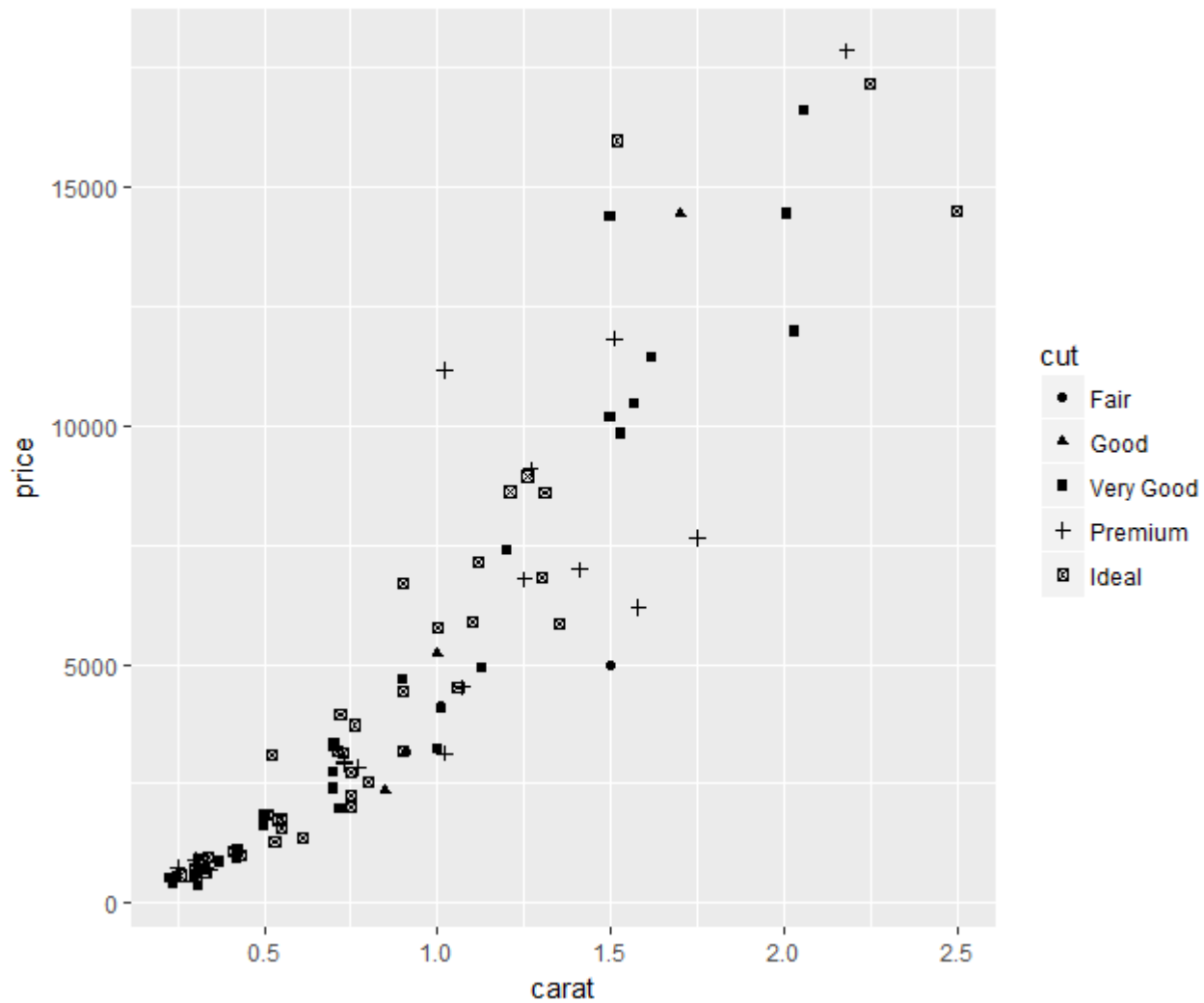
# Colour, size, shape and other aesthetic attributes

`ggplot` could automatically create a categorical variable in your data, and it also automatically provide a legend that maps the displayed attributes to the data values.

```
ggplot(carat, price, data = dsmall, colour = color)
```



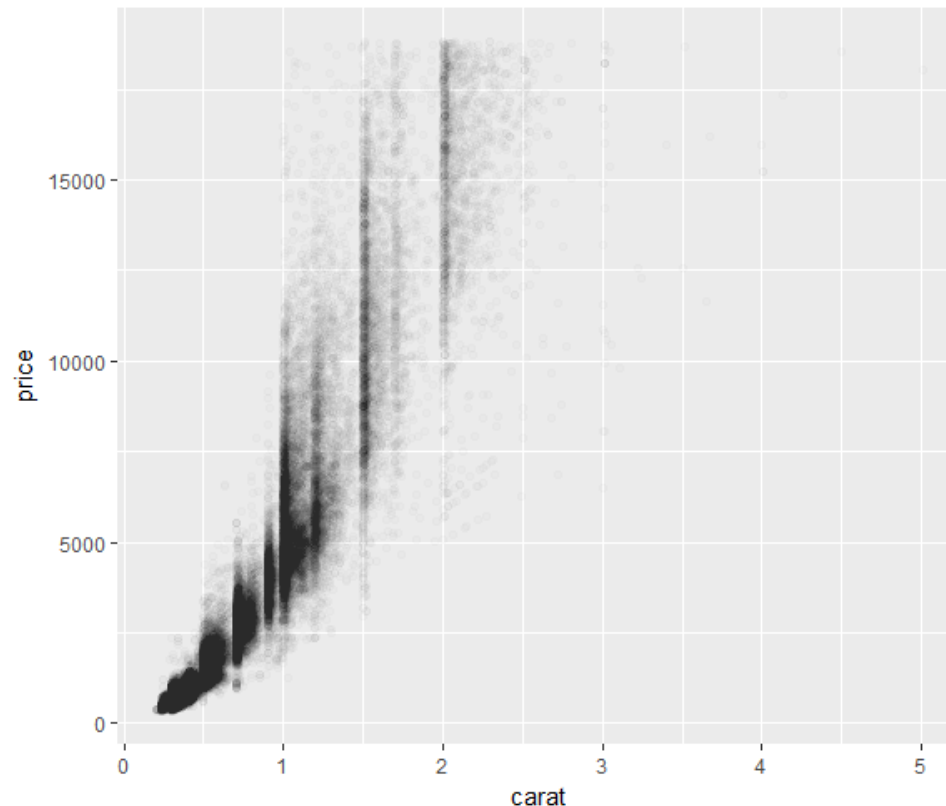
```
qplot(carat, price, data = dsmall, shape = cut)
```



For large datasets, like the `diamonds` data, semi-transparent points are often useful to avoid overplotting.

To make a semi-transparent colour you can use the `alpha` aesthetic, which takes a value between 0 (completely transparent) and 1 (complete opaque).

```
ggplot(carat, price, data = diamonds, alpha = I(1/100))
```



# geom

`qplot` is not limited to scatterplots, but can produce almost any kind of plot by varying the argument `geom`.

Geom 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.

The following geoms enable you to investigate two-dimensional relationships:

- `geom = "point"` draws points to produce a scatterplot. This is the default when you supply both `x` and `y` arguments to `qplot()`.
- `geom = "smooth"` fits a smoother to the data and displays the smooth and its standard error.
- `geom = "boxplot"` produces a box-and-whisker plot to summarise the distribution of a set of points.
- `geom = "path"` and `geom = "line"` draw lines between the data points.
  - A line plot is constrained to produce lines that travel from left to right, while paths can go in any direction.

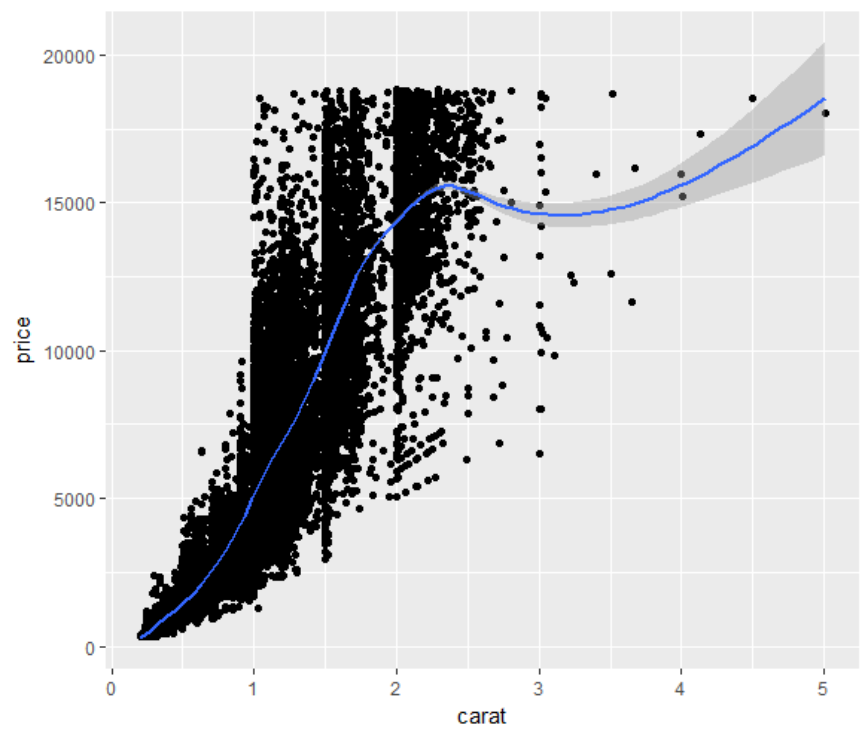
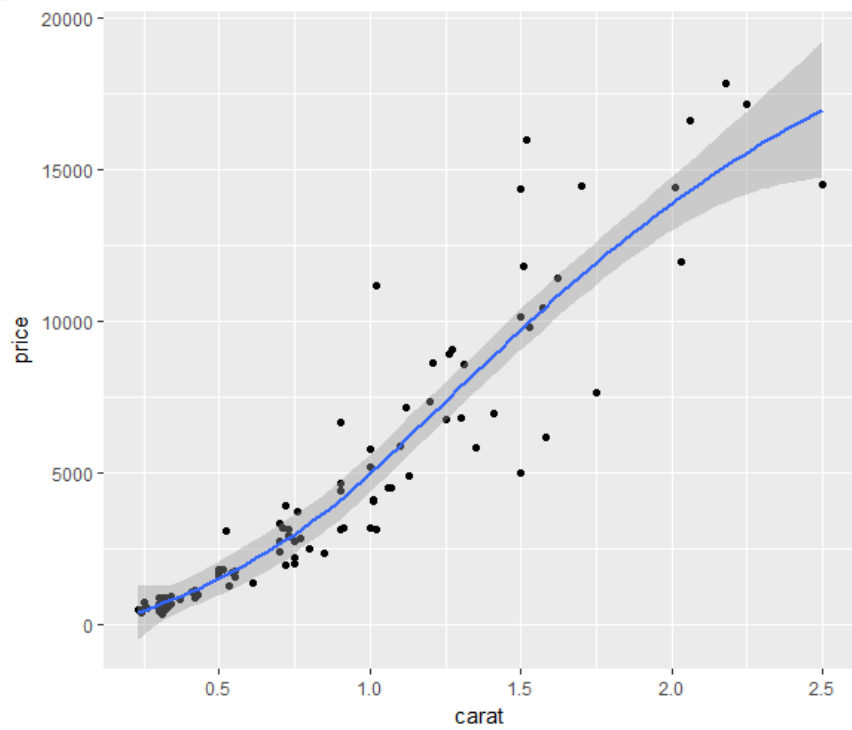
# Adding a smoother to a plot

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("point", "smooth"))
```

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





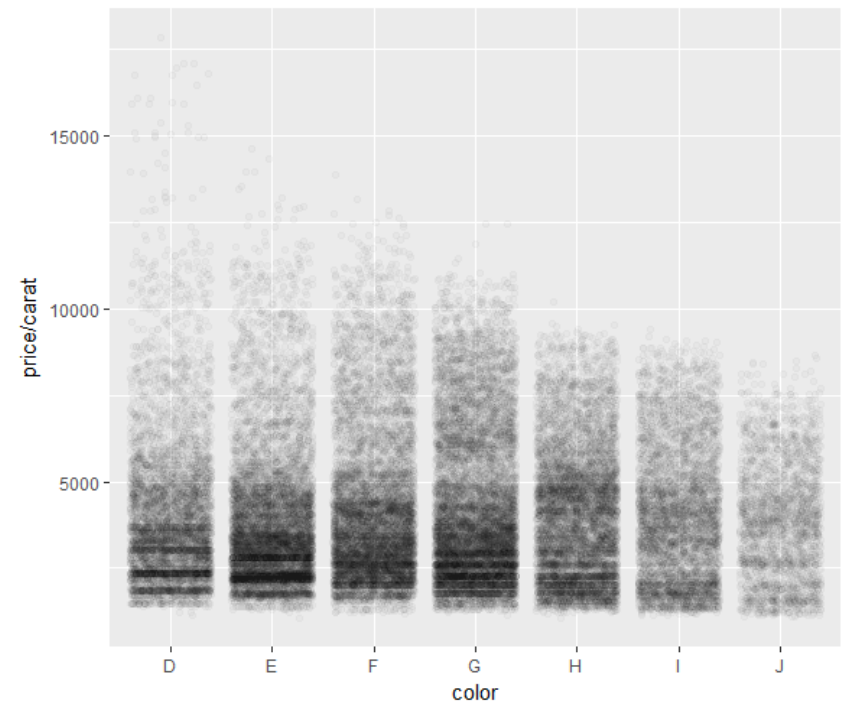
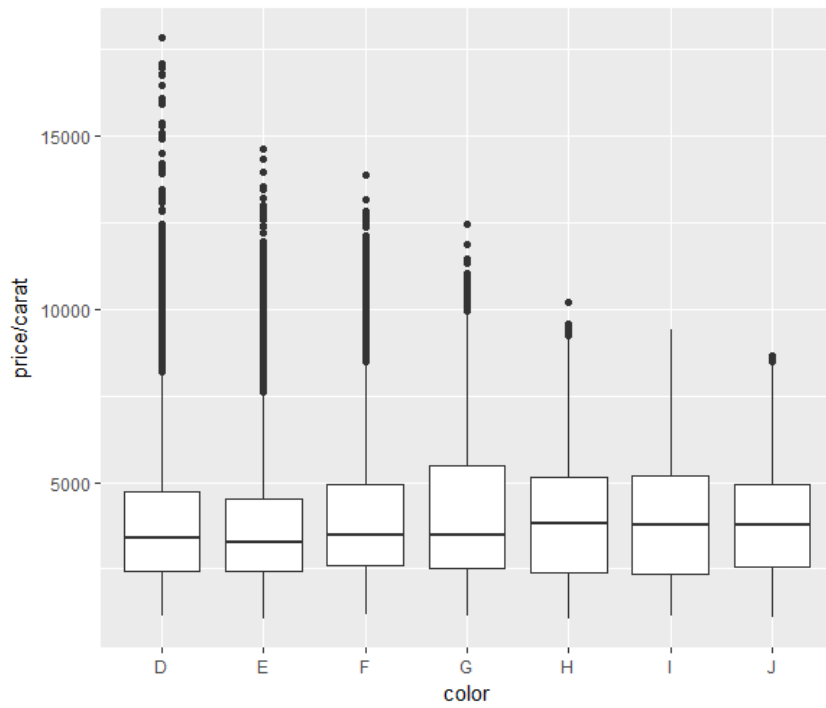
# Boxplots and jittered points

Boxplots summarise the bulk of the distribution with only five numbers, while jittered plots show every point but can suffer from overplotting.

To explore how the distribution of price per carat varies with the colour of the diamond

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

```
ggplot(color, price / carat, data = diamonds, geom  
= "jitter", alpha = I(1 / 50))
```



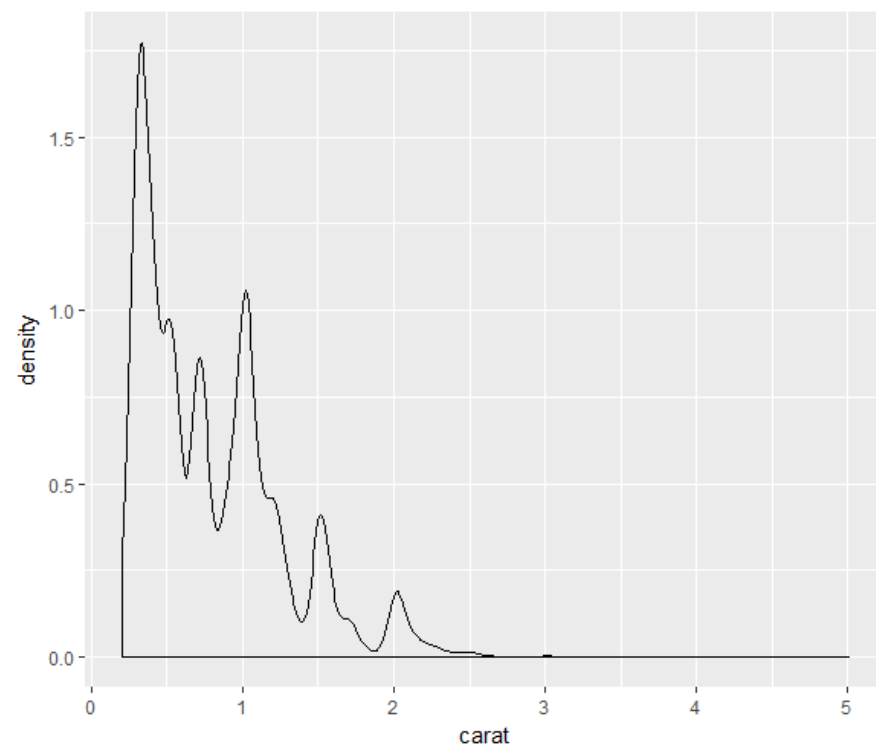
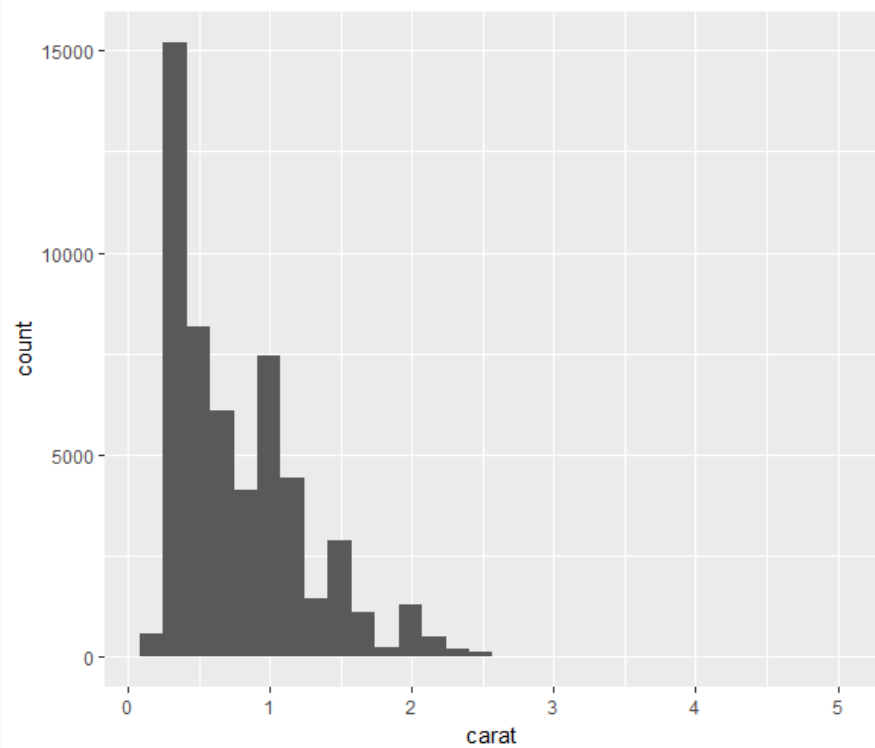
# Histogram and density plots

Histogram and density plots show the distribution of a single variable.

To show the distribution of carats with a histogram and a density plot.

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

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

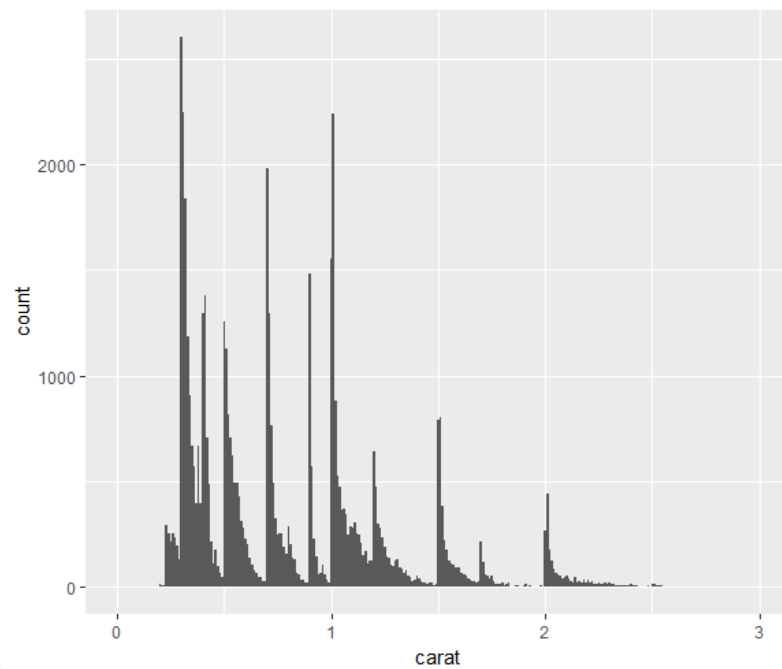
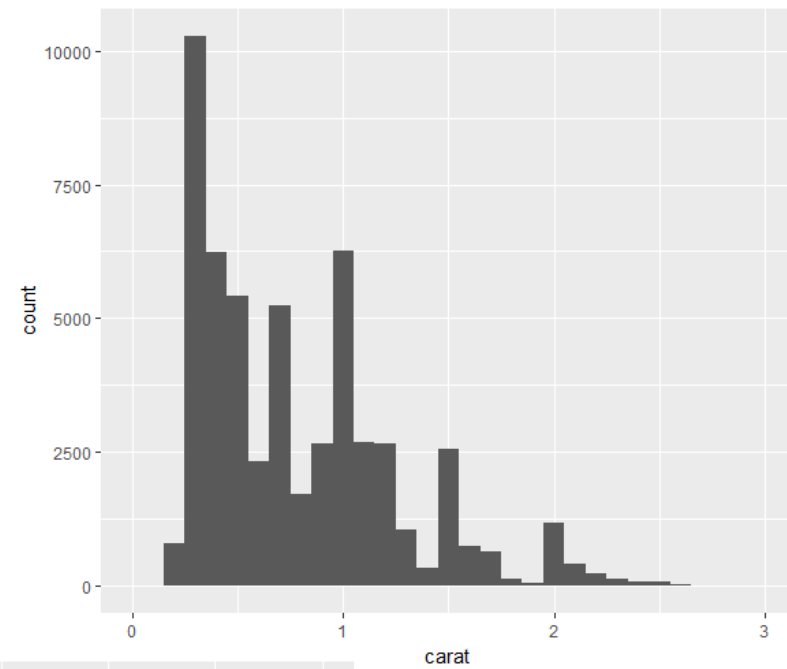
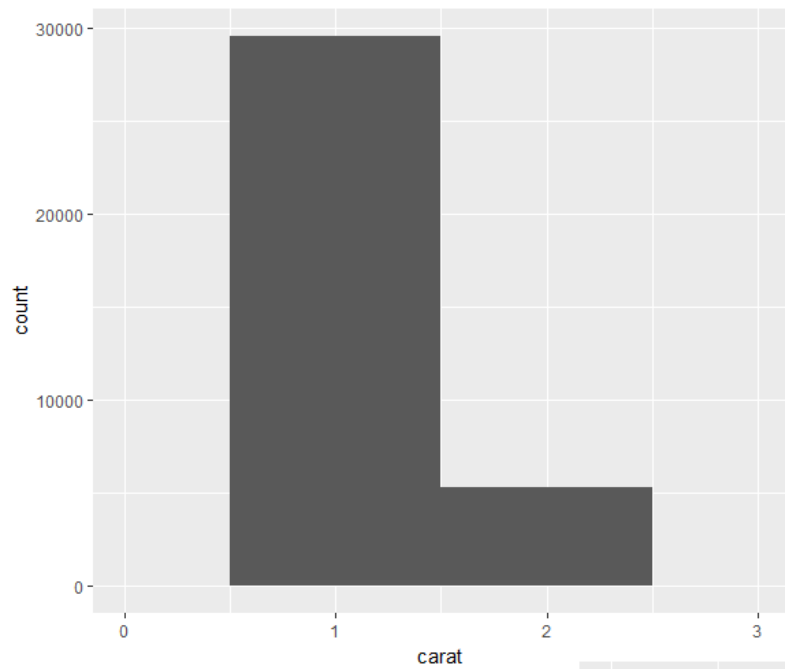


We could also experiment with three values of binwidth: 1.0, 0.1 and 0.01.

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

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

```
qplot(carat, data = diamonds, geom = "histogram",  
binwidth = 0.01, xlim = c(0,3))
```

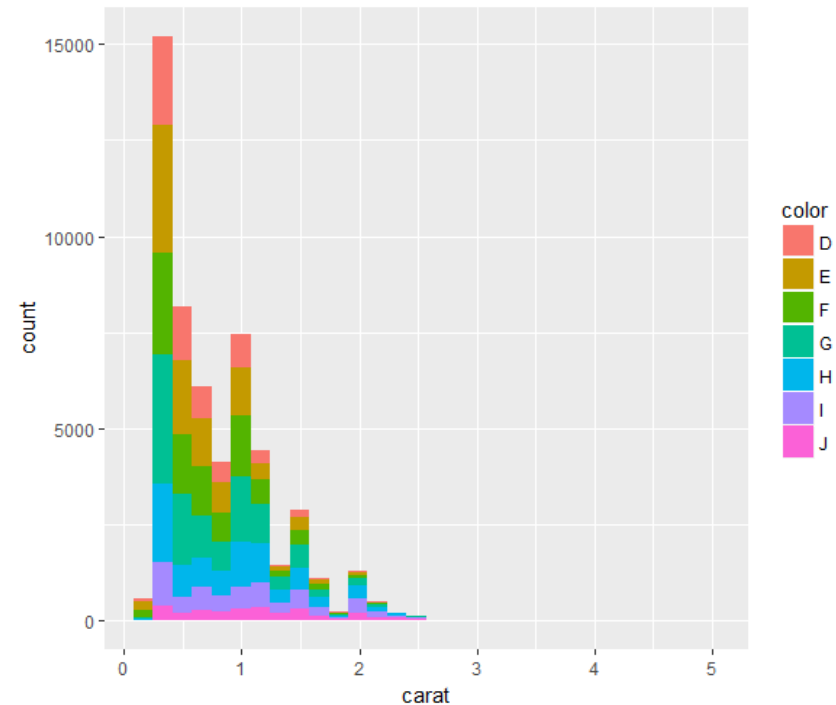
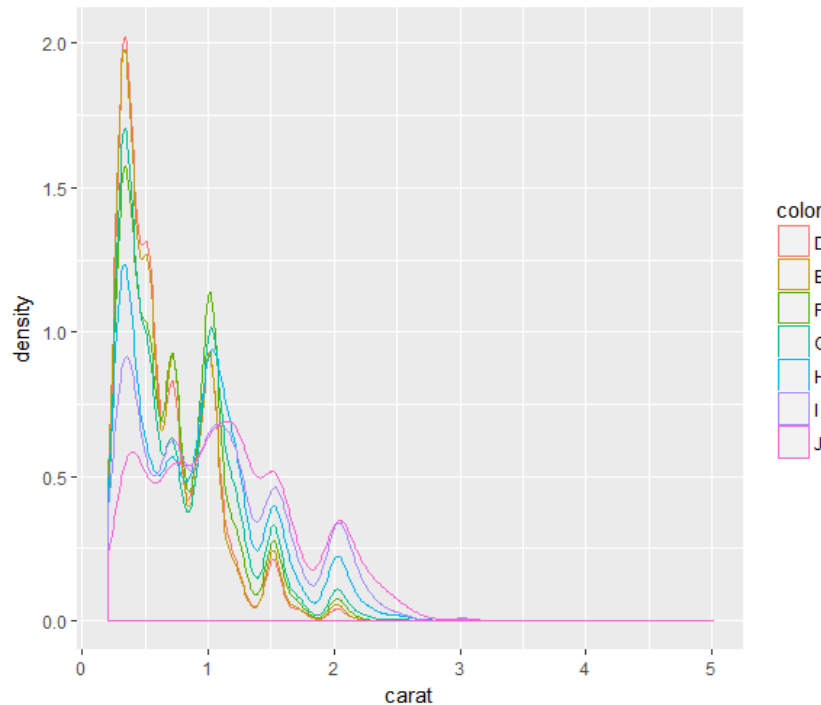


To compare the distributions of different subgroups, we could add an aesthetic mapping.

Mapping a categorical variable to an aesthetic will automatically split up the geom by that variable.

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

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



# Time series with line and path plots

Line plots usually have time on the x-axis, showing how a single variable has changed over time.

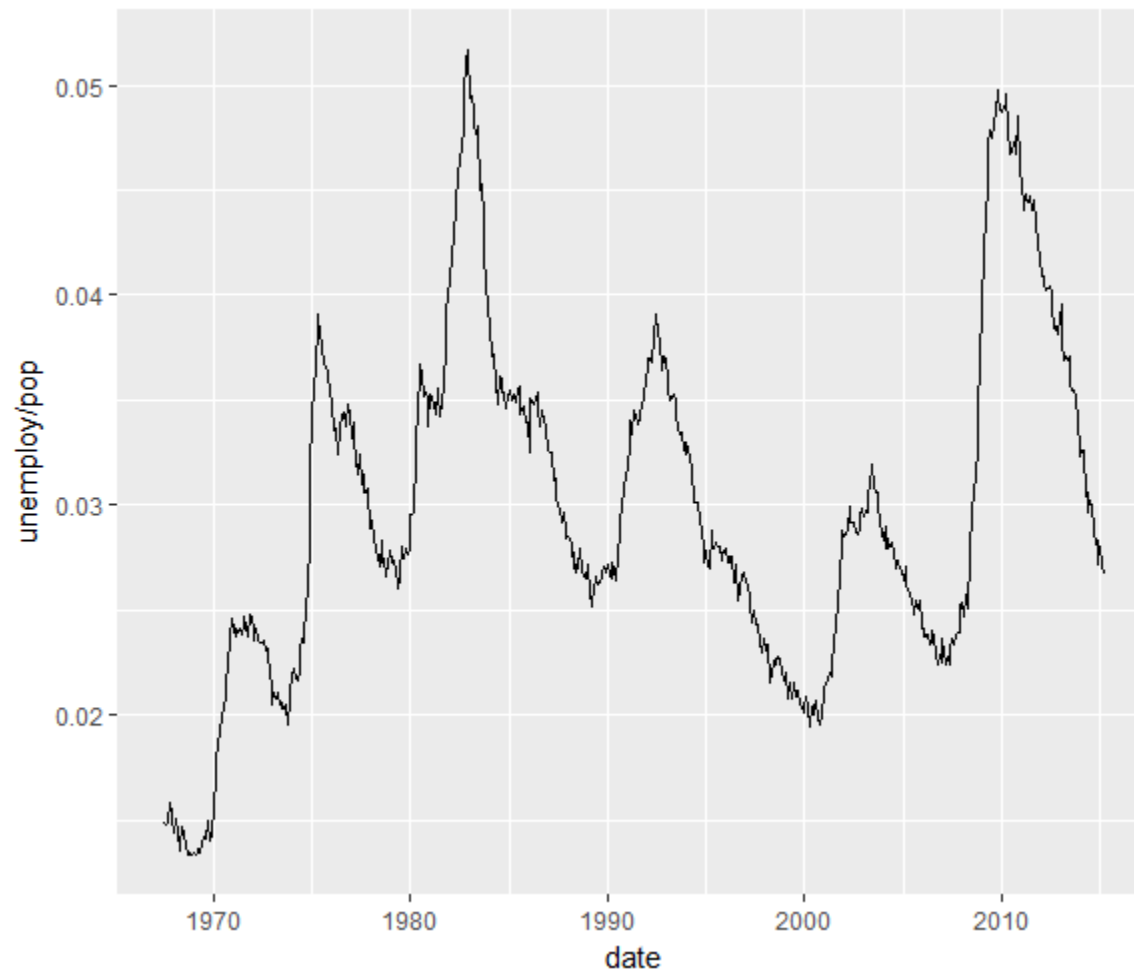
Path plots show how two variables have simultaneously changed over time, with time encoded in the way that the points are joined together.

Consider the `economics` dataset, which contains economic data on the US including

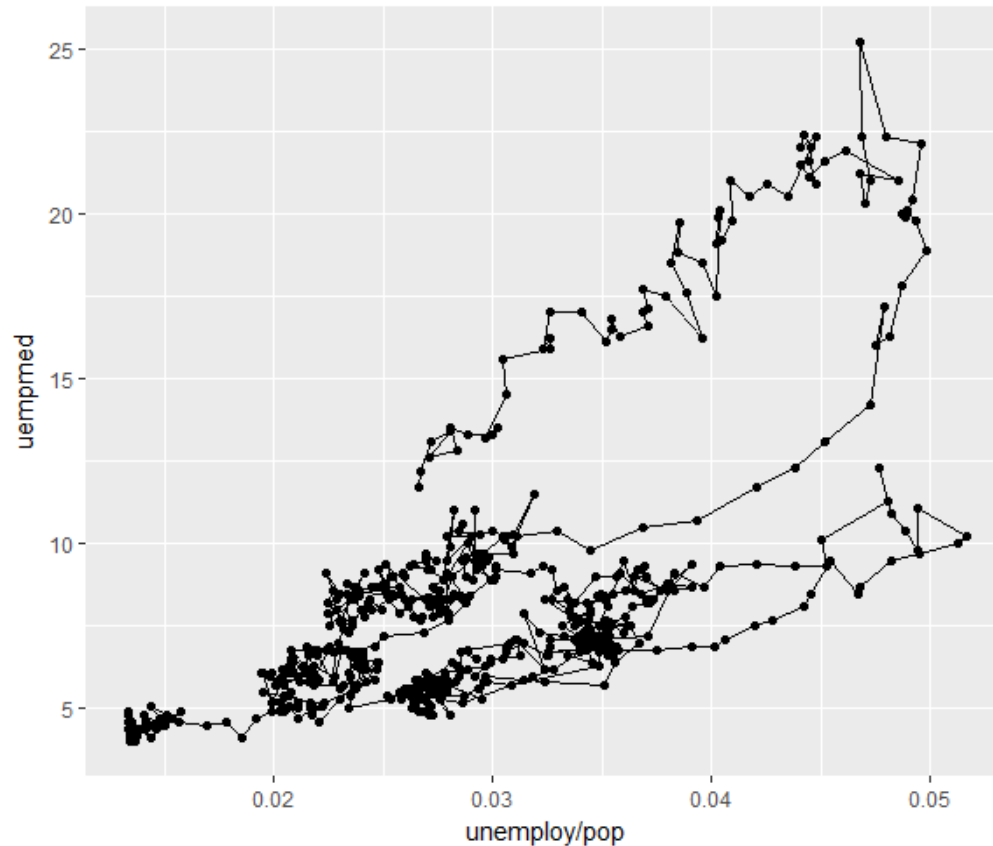
- `psavert`: personal savings rate.
- `pce`: personal consumption expenditures, in billions of dollars.
- `unemploy`: number of unemployed in thousands.
- `uempmed`: median duration of unemployment, in weeks.
- `pop`: total population.



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



```
ggplot(unemploy / pop, uempmed, data = economics,  
geom = c("point", "path"))
```



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

# Faceting

Faceting creates tables of graphics by splitting the data into subsets and displaying the same graph for each subset in an arrangement that facilitates comparison.

The default faceting method in `ggplot()` creates plots arranged on a grid specified by a faceting formula which looks like

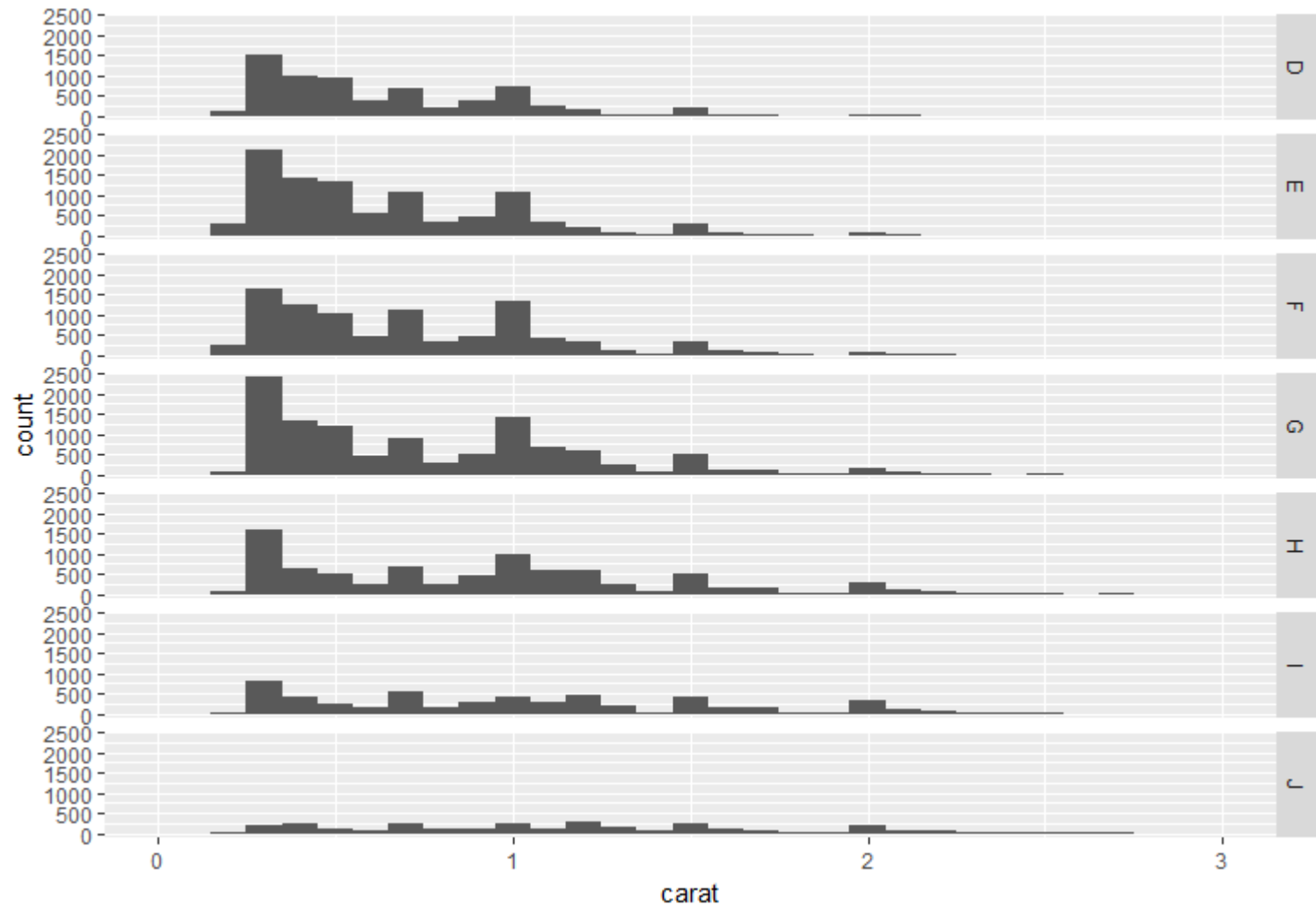
$$\text{row.var} \sim \text{col.var}$$

To facet on only one of columns or rows, use "." as a place holder. For example,

$$\text{row.var} \sim .$$

will create a single column with multiple rows.

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



# Build a plot layer by layer

- Layering is the mechanism by which additional data elements are added to a plot.
- Each layer can come from a different dataset and have a different aesthetic mapping, allowing us to create plots that could not be generated using `qplot()`, which permits only a single dataset and a single set of aesthetic mappings.
- There are five components of a layer: data, a set of aesthetic mappings, geom, stat, and position adjustment.

# Creating a plot

- When we used `qplot()`, it did a lot of things for us: it created a plot object, added layers, and displayed the result, using many default values along the way.
- To create the plot object ourselves, we use `ggplot()`.
- `ggplot()` has two arguments: data and aesthetic mapping.
- These arguments set up defaults for the plot and can be omitted if you specify data and aesthetics when adding each layer.

- For example,

```
p <- ggplot(diamonds, aes(carat, price,  
  colour = cut))
```

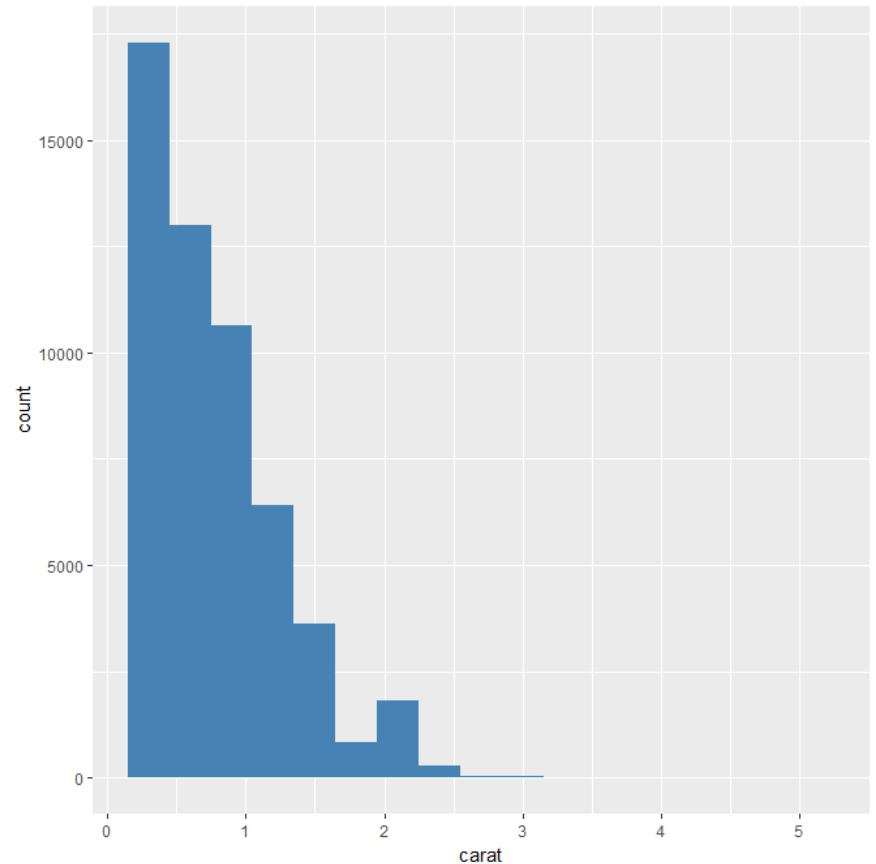
specifies a default mapping of `x` to `carat`, `y` to `price` and `colour` to `cut`.

- This plot object cannot be displayed until we add a layer: there is nothing to see!
- We need to use `+` to **add** the layer to the plot.
- A fully specified layer can take these arguments

```
layer(geom, stat, params, data, mapping,  
  position)
```

The following layer produces a histogram (a combination of bars and binning) coloured "steelblue" with a bin width of 0.3.

```
p <- ggplot(  
  diamonds,  
  aes(x = carat))  
p <- p + layer(  
  geom = "bar",  
  stat = "bin",  
  position = "identity",  
  params = list(fill = "steelblue", binwidth = 0.3)  
)  
p
```





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

```
p + geom_histogram(binwidth = 0.3, fill =  
"steelblue" )
```

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

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

Their common parameters define the components of the layer.

- `mapping` (optional): A set of aesthetic mappings, specified using the `aes()` function and combined with the plot defaults.
- `data` (optional): A dataset which overrides the default plot dataset. It is most commonly omitted, in which case the layer will use the default plot data.

- ...: Parameters for the `geom` or `stat`, such as bin width in the histogram. The example above showed setting the fill colour of the histogram to `"steelblue"`.
- `geom` or `stat` (optional): You can override the default `stat` for a `geom`, or the default `geom` for a `stat`. This is a text string containing the name of the geom to use. Using the default will give you a standard plot; overriding the default allows you to achieve something more exotic.
- `position` (optional): Choose a method for adjusting overlapping objects.

For a list of geoms, see

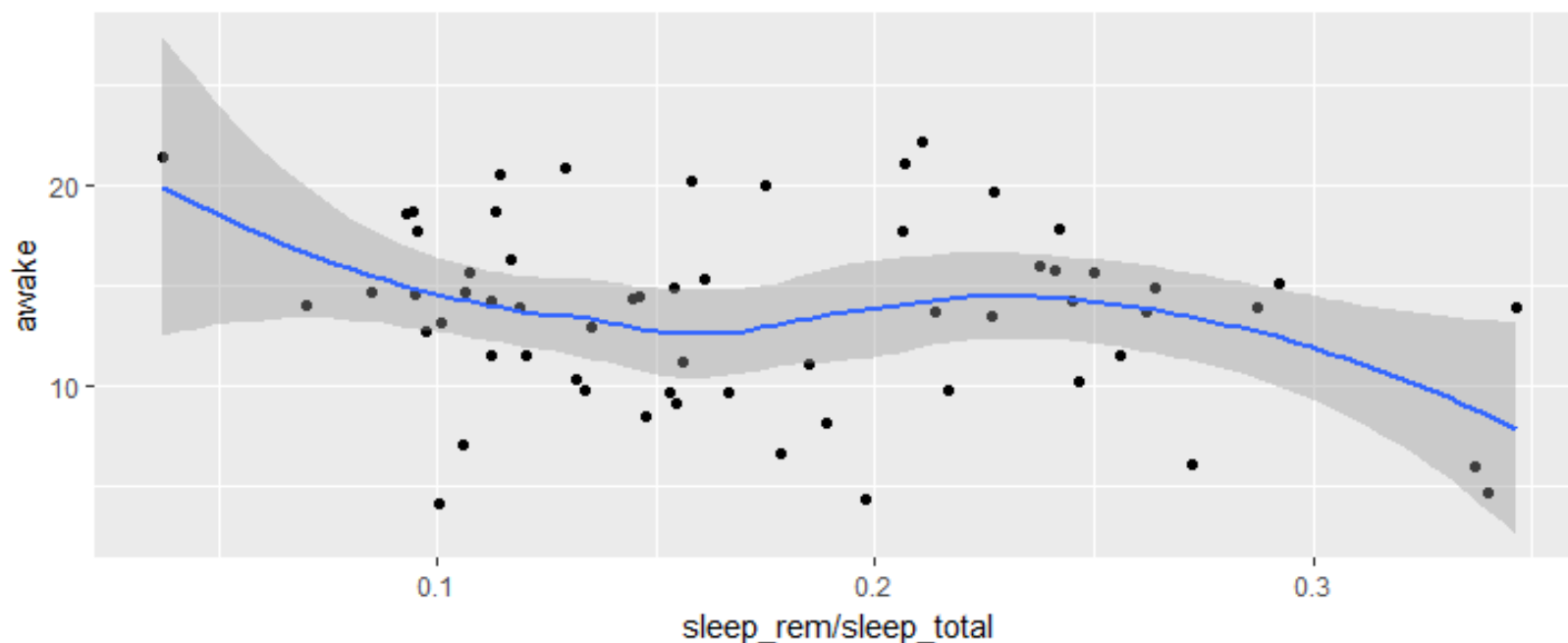
```
geoms <- help.search( "^geom_", package =  
  "ggplot2" )  
geoms$matches[, 1:2]
```

The following plots are equivalent.

```
qplot(sleep_rem / sleep_total, awake, data =  
msleep) + geom_smooth()
```

```
qplot(sleep_rem / sleep_total, awake, data =  
msleep, geom = c("point", "smooth"))
```

```
ggplot(msleep, aes(sleep_rem / sleep_total,  
awake)) + geom_point() + geom_smooth()
```



# Aesthetic mappings

The `aes()` function takes a list of aesthetic-variable pairs like these:

```
p <- ggplot(data=mtcars, aes(x=hp, y=mpg,  
                             color=cyl))
```

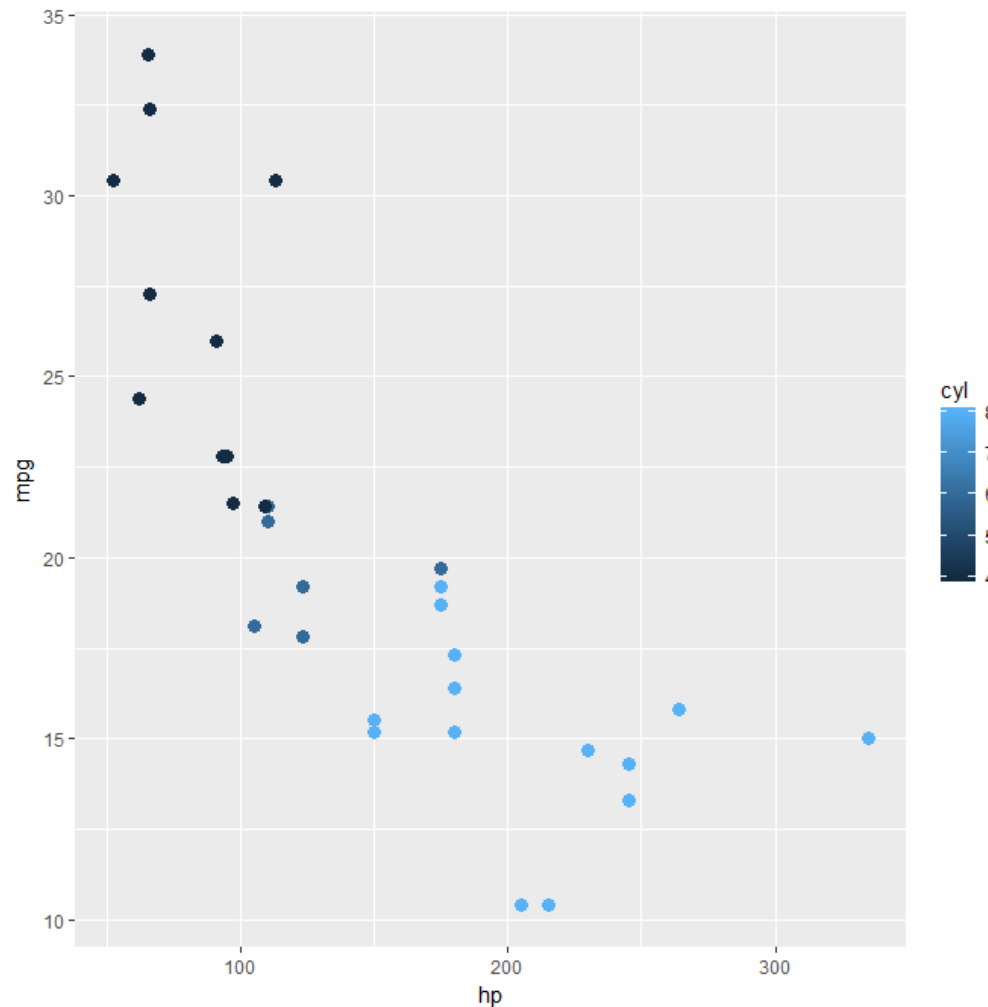
Here we are mapping x-position to `hp`, y-position to `mpg`, and `color` to `cyl` within the data frame `mtcars`.

The first two arguments can be left without names, in which case they correspond to the `x` and `y` variables. i.e.,

```
aes(hp, mpg , color = cyl)
```

A scatterplot could then be created using

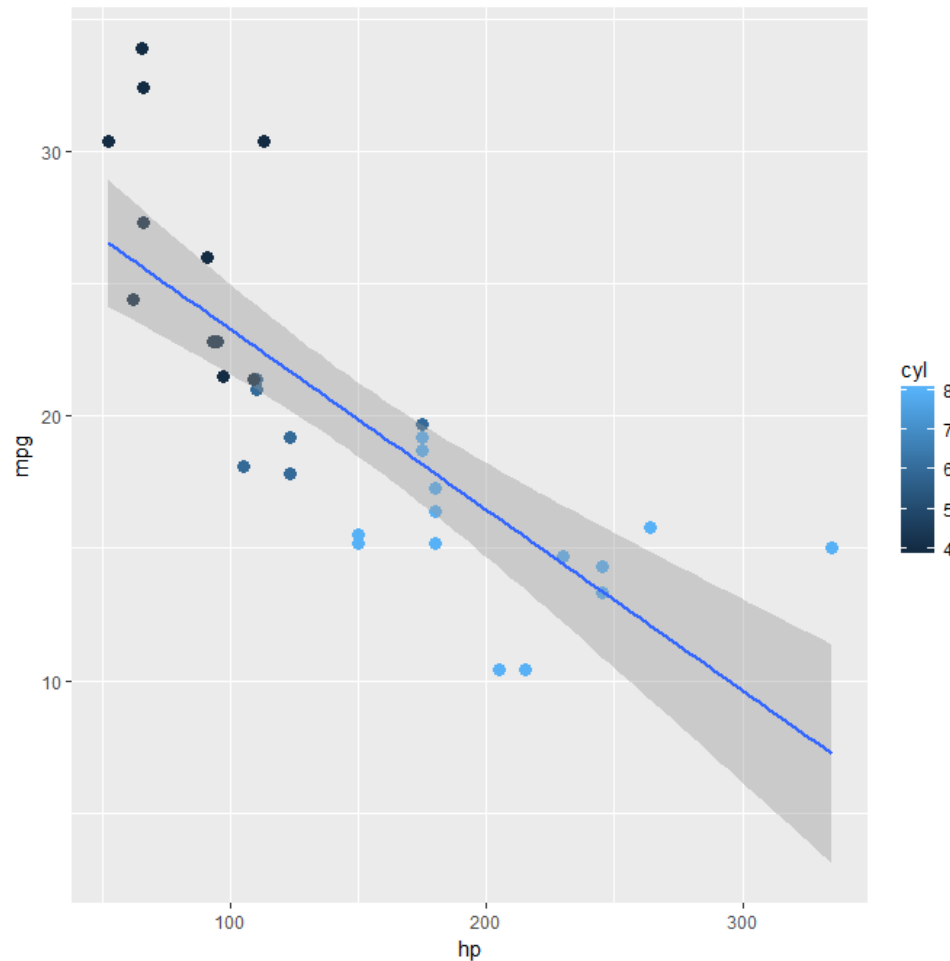
```
p + geom_point(size=3)
```



We can add an additional layer using `geom_smooth`

```
p + geom_point(size=3)
```

```
+ geom_smooth(method="lm", aes(fill=cyl))
```



# stat

A stat takes a dataset as input and returns a dataset as output, and so a stat can add new variables to the original dataset. It is possible to map aesthetics to these new variables.

For a list of stats, see

```
stats <- help.search("^stat_", package= "ggplot2")  
stats$matches[, 1:2]
```

For example, `stat_bin`, the statistic used to make histograms, produces the following variables:

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

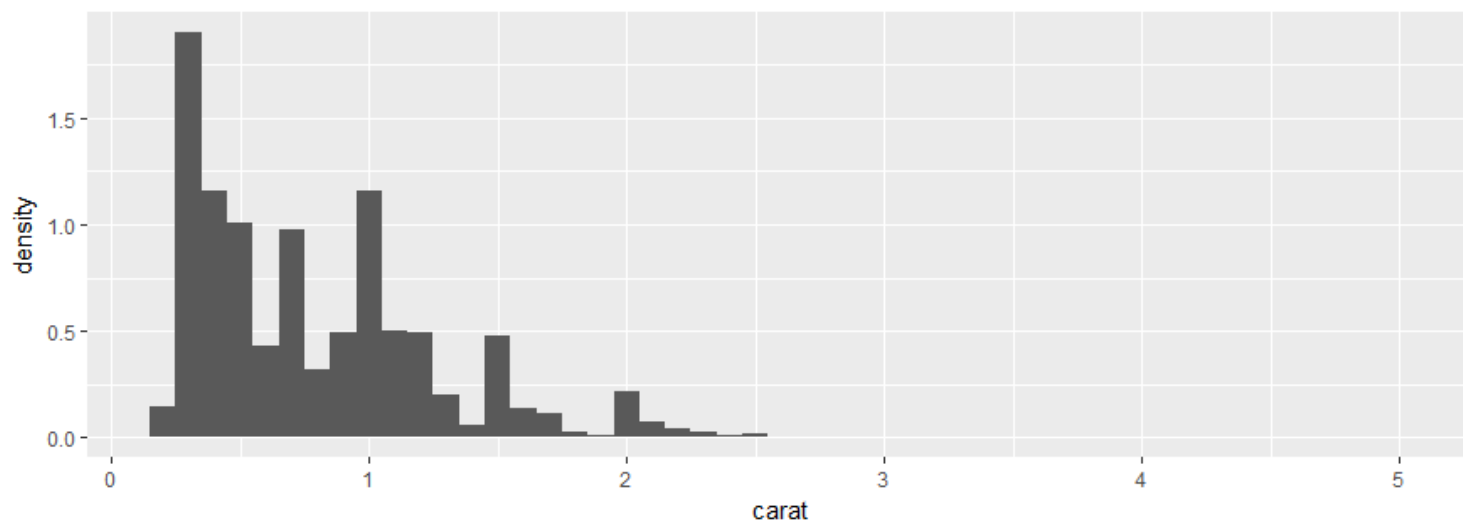


These generated variables can be used instead of the variables present in the original dataset.

For example, the default histogram geom assigns the height of the bars to the number of observations (`count`), but if you'd prefer a more traditional histogram, you can use the density (`density`).

The following example shows a density histogram of `carat` from the `diamonds` dataset. `"..density.."` tells `ggplot2` to map the density to the y-axis instead of the default use of `count`.

```
ggplot(diamonds, aes(carat)) + geom_histogram(aes(y = ..density..), binwidth = 0.1)
```



# Position adjustments

Position adjustments apply minor tweaks to the position of elements within a layer. Position adjustments available within `ggplot2` are

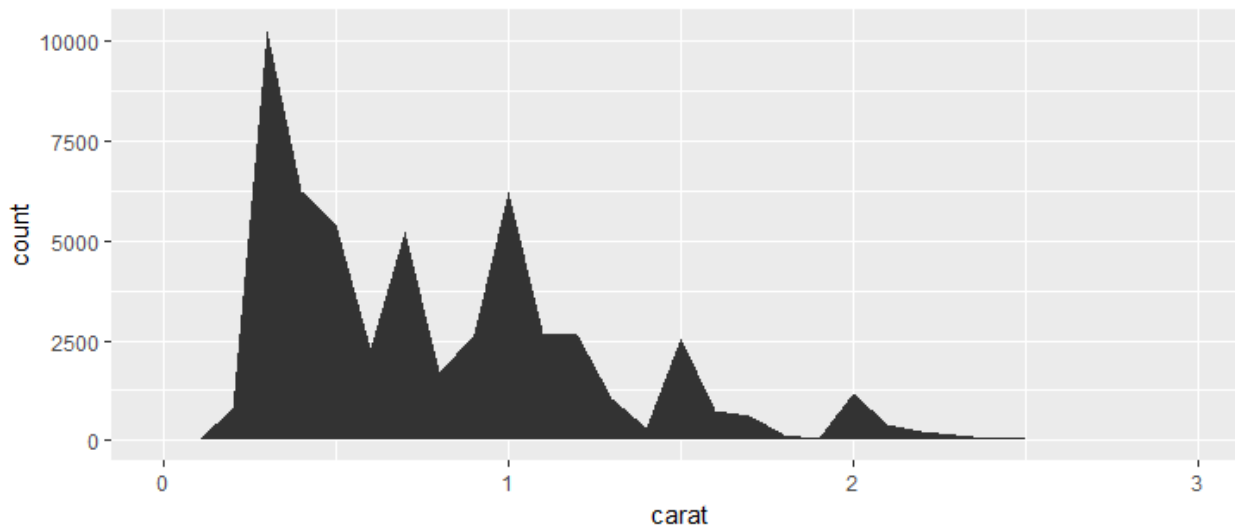
- `dodge`: Adjust position by dodging overlaps to the side.
- `fill`: Stack overlapping objects and standardise have equal height.
- `identity`: Don't adjust position.
- `jitter`: Jitter points to avoid overplotting.
- `stack`: Stack overlapping objects on top of one another.

# Combining geoms and stats

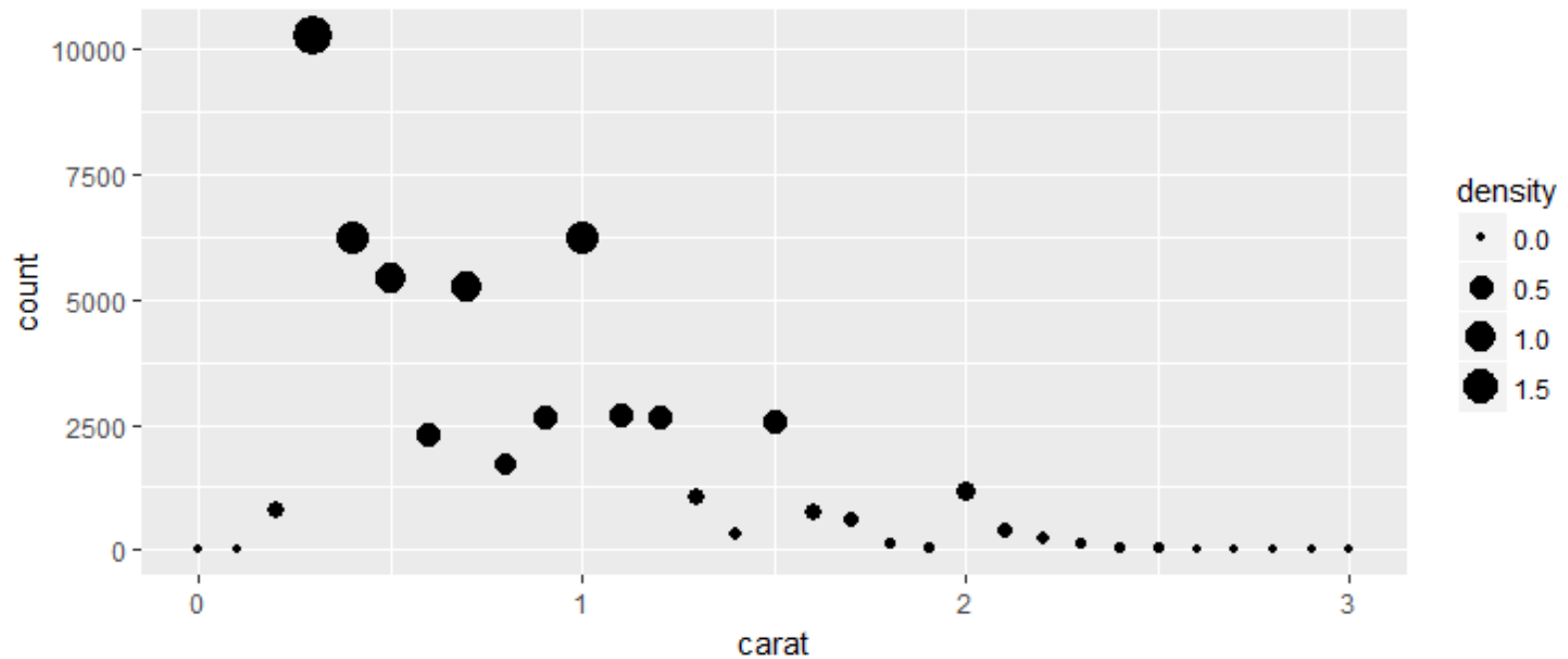
By connecting geoms with different statistics, you can easily create new graphics.

The following plots use the same statistical transformation underlying a histogram (the bin stat), but use different geoms to display the results: the area geom, and the point geom

```
d <- ggplot(diamonds, aes(carat)) + xlim(0, 3)
d + stat_bin(aes(ymax = ..count..), binwidth =
0.1, geom = "area")
```



```
d + stat_bin(  
  aes(size = ..density..), binwidth = 0.1,  
  geom = "point", position="identity"  
)
```

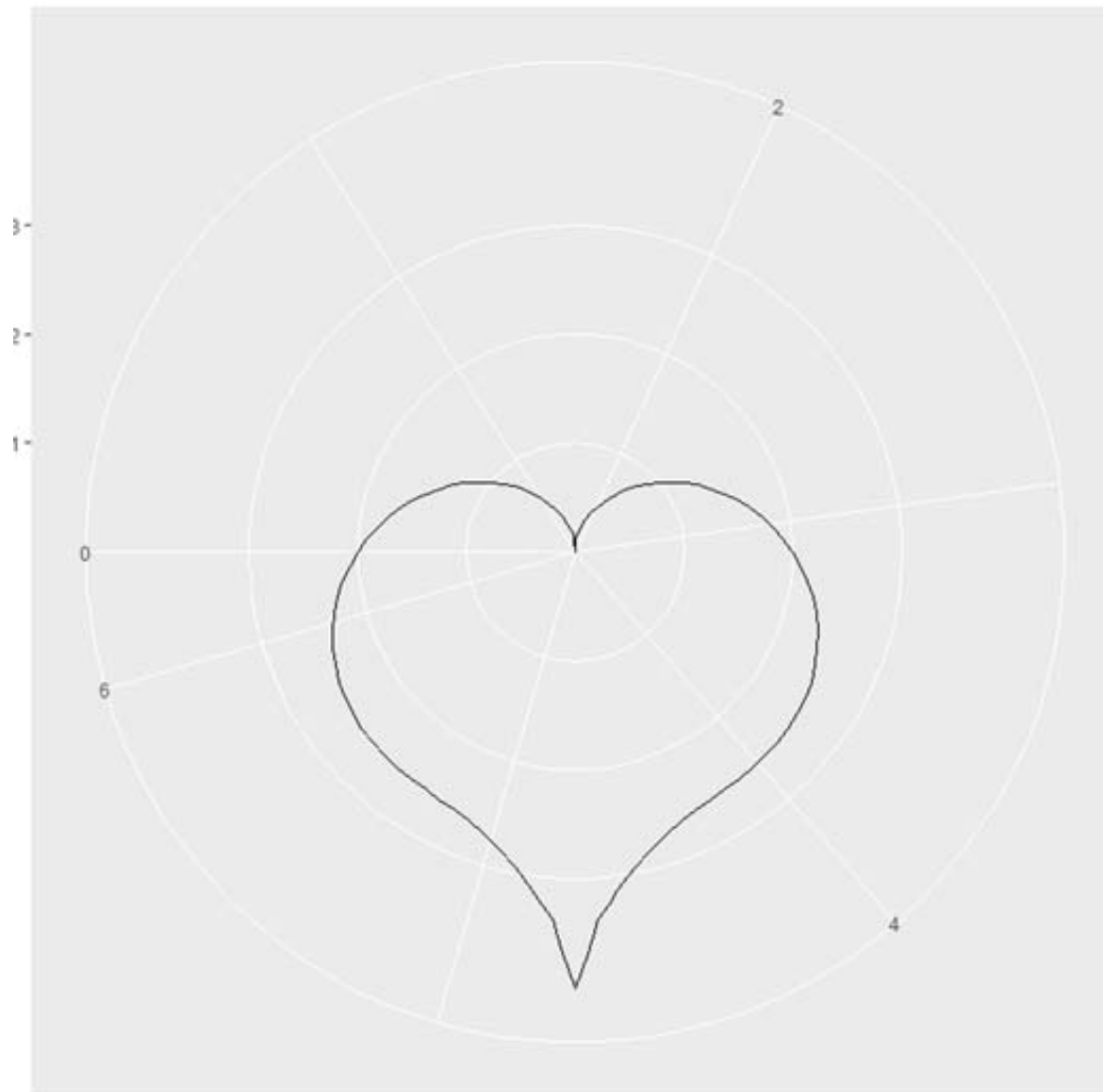


# Example: Plotting a heart

A heart shaped curve can be represented in terms of the polar coordinate system using the following equation

$$r = \frac{\sin \theta \sqrt{|\cos \theta|}}{\sin \theta + \frac{7}{5}} - 2 \sin \theta + 2, \quad -\pi \leq \theta \leq \pi$$

```
dat <- data.frame(x=seq(0, 2*pi, length=100))  
heart <- function(x) {2-2*sin(x) +  
sin(x)*(sqrt(abs(cos(x))))/(sin(x)+1.4)}  
ggplot(dat, aes(x=x)) + stat_function(fun=heart) +  
coord_polar(start=-pi/2)
```



# The `tibble` package

The data frames in base R have a few defaults that may cause confusion:

- In R versions earlier than 4.0.0, character vectors are automatically changed to factors.
- Names are changed to legal R variable names, e.g. spaces are replaced with dots, for example

```
> data.frame("Annual Income" = 100000)
  Annual.Income
1           1e+05
```

- Printing very large data frames takes up a lot of space in the console
- Taking a subset of a data frame sometimes produces a data frame (multi-column), and sometimes produces a vector (single column)

The `tibble` package introduces a new kind of data frame called a `tibble` that fixes them by default:

- character vectors and names are left as is,
- printing defaults to a summary, and
- subsetting produces another `tibble`.

Besides those changes, the `tibble()` function has other differences from `data.frame()`:

- The arguments must all be the same length, or length 1; recycling of other lengths is assumed to be an error.
- The arguments are evaluated in order, and earlier arguments can be used in the definition of later ones.

For example,

```
tibble(x = 1:2, y = x + 1)
```

```
# A tibble: 2 × 2
```

	x	y
	<int>	<dbl>
1	1	2
2	2	3



# The `readr` package

The main goal of the `readr` package is to allow the user to read a file and output a `tibble`. This involves three steps:

1. The file needs to be converted to a rectangular matrix of strings.
2. The type of each column needs to be determined.
3. Each column needs to be converted into the appropriate type.

The readr package has several nice features. It can read from a variety of sources besides files, including reading directly from a character vector.

```
data <- "x,y,z
        1,2,3
        4,5,6
        7,B,9"

read_csv(data)
# A tibble: 3 × 3
```

	x	y	z
	<dbl>	<chr>	<dbl>
1	1	2	3
2	4	5	6
3	7	B	9

We see that the middle column is read as character because of the B in the third row of data.

We could force it to read numbers by specifying the types of columns

```
read_csv(data, col_types = "nnn") # three  
numeric columns
```

```
# A tibble: 3 × 3
```

	x	y	z
	<dbl>	<dbl>	<dbl>
1	1	2	3
2	4	5	6
3	7	NA	9

The package also contains some functions such as `write_csv()` to write files, this is actually an easier task than reading them.

# The `stringr` package

When we read data from a file, often it needs further manipulation to be useful. For example, we may read some phone numbers in one format and others in a different format, and need to make them consistent so they can be compared.

The `stringr` package is a collection of functions to deal with problems like these. It classifies the functions into four groups

1. Working with individual characters within longer strings.
2. Adding and removing blank space.
3. Operations that depend on the conventions in different languages, such as sorting strings.
4. Pattern matching.

# Example: Handling phone numbers

Our data frame has a mix of different phone number styles and we would like to standardize them.

```
~ 2f
# A tibble: 5 × 1
  phone
  <chr>
1 705 555 0100
2 +1 519 555 0101
3 4165550102
4 514 555-0103
5 011-800-555-0104
```

Our strategy to standardize the numbers is as follows

1. Remove all non-digits.
2. If 11 digits are left and the first is a 1, remove it too.
3. If anything other than 10 digits are left, it's an error and should give NA.
4. Otherwise reformat to the standard format.

```

phone <- df$phone %>%
  str_remove_all("[^0-9]") %>%
  str_replace("1(.....)", "\\1") %>%
  "[<-"(., str_length(.) != 10, NA) %>%
  "str_sub<-"(7, 6, value = "-") %>%
  "str_sub<-"(4, 3, value = "-")
phone
[1] "705-555-0100" "519-555-0101" "416-555-0102"
"514-555-0103"
[5] NA

```

We use replacement functions on the last three lines of the pipe. See descriptions in the R file for details.

# The dplyr package

The functions in `dplyr` are designed to work in pipes. They are intended to be thought of as verbs: do this, then do that, etc., and are named that way.

For illustration purpose, consider the `mpg` dataset which describe mileage per gallon performances of various cars.

```
> mpg
# A tibble: 234 × 11
  manufacturer model   displ  year   cyl trans drv   cty   hwy fl   class
  <chr>         <chr>   <dbl> <int> <int> <chr> <chr> <int> <int> <chr> <chr>
1 audi         a4       1.8   1999     4 auto... f     18    29 p     comp...
2 audi         a4       1.8   1999     4 manu... f     21    29 p     comp...
3 audi         a4       2     2008     4 manu... f     20    31 p     comp...
4 audi         a4       2     2008     4 auto... f     21    30 p     comp...
5 audi         a4       2.8   1999     6 auto... f     16    26 p     comp...
...
```

`filter()` selects cases based on their values.

To show the four cars in the `mpg` data frame that have a five-cylinder engine.

```
mpg %>% filter(cyl == 5)
```

`arrange()` sorts the rows of a data frame.

We can sort the cars into ascending city mileage, with ties broken by highway mileage, using

```
mpg %>% arrange(cty, hwy)
```

To have descending order, we use

```
mpg %>% arrange(desc(cty), desc(hwy))
```

`select()` selects variables based on their names.

```
mpg %>% select(manufacturer, model,  
cty, hwy)
```

We can rename columns as we select them

```
mpg %>% select(manufacturer, model,  
City = cty, Highway = hwy)
```



`mutate()` computes variables that may be functions of existing variables.

These functions operate sequentially, so later changes can refer to earlier ones.

To change units from miles per gallon to litres per 100 km, we could use

```
lper100km <- mpg %>%  
mutate(cty = 235.215/cty, hwy = 235.215/hwy)  
lper100km
```

`summarize()` applies functions to groups of values. The `group_by()` function defines the groups it works with.

To find the mean litres per 100 km by number of cylinders, we could use

```
lper100km %>% group_by(cyl) %>%  
summarize(cty = mean(cty), hwy = mean(hwy))
```

`sample_n()` and `sample_frac()` sample values, possibly from within groups. Here is a random vehicle for each cylinder count

```
lper100km %>%  
group_by(cyl) %>% sample_n(1)
```

# Other tidyverse packages

The tidyverse project includes dozens of R packages besides the ones we've discussed. Other core tidyverse packages include

- `tidyr`: tools for rearranging data sets into the standard *tidy* format of one case per row. Data may arrive with multiple observations per row, stored in different columns; these functions help you to fix that.
- `forcats`: functions for working with factors. These allow entries in tables and plots to be easily merged and reordered.
- `purrr`: functions for implementing functional programming in a consistent way.

See <https://www.tidyverse.org/packages/> for details.