ggplot2 Tutorial

Zjardyn Liera-Hood

2022-10-23

Getting started

The prerequisites for data visualization in R (Using ggplot2) are the following:

1. Install base R. Click the blue R to access links to the different versions based on your operating system.
2. Windows users: Install Rtools42. This is required if you are building R packages from source on Windows.
3. Mac users: Potentially install Xcode to install certain packages. The only package that gave me trouble was `traitformdata`, so I have included a supplemental dataset (pantheria.R) if you cannot load this package and do not wish to download Xcode (~20 Gb!).
4. Download Rstudio. RStudio is handier to use than regular R.
5. Install the following R packages. These are the ones I will be using for the tutorial. If you want to run and modify the code, open the `.Rmd` file with the same name as this file. The `.Rmd` file contains the written text and code chunks used to build this .pdf. You can click on a line of code and type Ctrl-Enter and the line(s) will be run in the console.

```r
# Install us
install.packages("ggplot2")
install.packages("dplyr")
install.packages("RColorBrewer")
install.packages("stringr")
install.packages("nlme")
# This one might give you trouble
install.packages("traitformdata")
install.packages("patchwork")
install.packages("viridisLite")
```

If there are no errors, the packages should be installed locally on your machine and you can continue. If not, message me on Slack and we can work this out. You should be able to run the following command with no errors:

```r
# Loading the package so that we can use it
library(ggplot2)
```

The purpose of this document is to show you a variety of plots that you can potentially use for your own work as well as showing you common pitfalls and some additional insight. I recommend you run the code for yourself and experiment with it as we go. Think of this as a cookbook of plot examples that can be tweaked, combined, and modified to your liking. I have tried to add comments to explain how the code works.

Data, Aesthetics and Geometries

Plotting starts with data. We will use the `iris` dataset, which gives measurements on 3 species of iris flower. `iris` is built into R. Going forward, I recommend you use the `?` operator on objects you don't understand. If you are unsure of what something does, you can type `?<thing>` into the console and get the documentation. Try: `?iris` and `?head`.

```
# Loads iris into our environment.
data(iris)
#  Prints the top 6 rows of iris
head(iris)
```

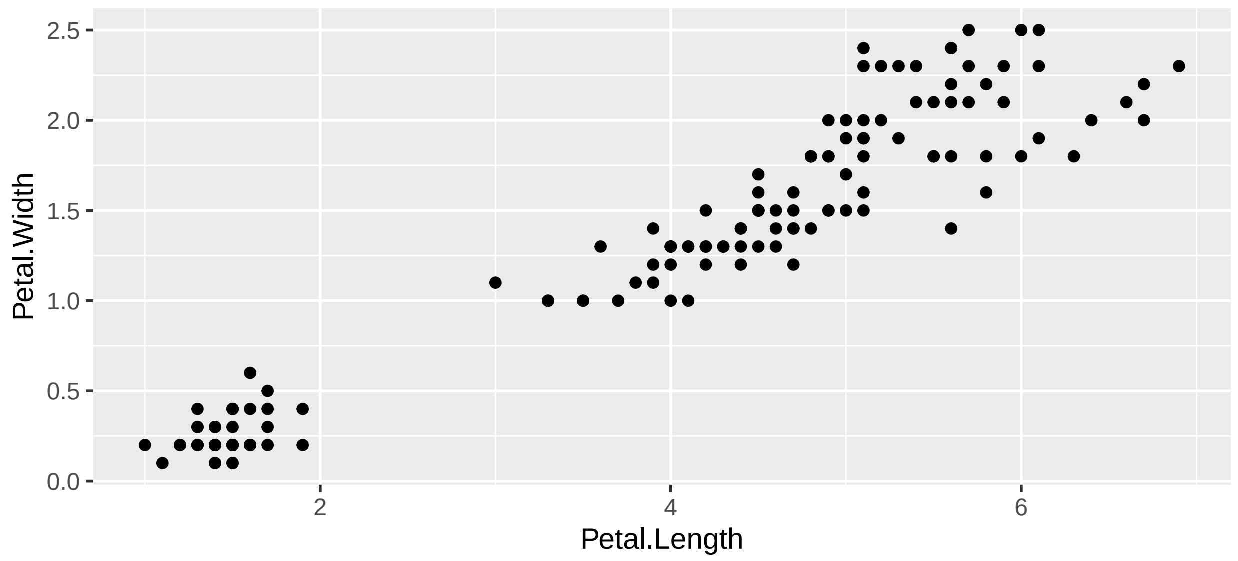
```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

To describe `iris`, the rows are samples, the columns are variables, and each cell contains a single value. This is known as **tidy data**. Tidy data is what ggplot expects as input. Data in the wild may not be tidy, especially in the bioinformatics field where there are varying data formats. Data may require prior import and manipulation (data wrangling). We will mostly use the built in datasets, but I will show you a few preprocessing examples. They may seem cryptic as they are outside the scope of this tutorial.

Here is the first example. The `ggplot` function call is passed the arguments `data` and `mapping` to specify the data and map data to aesthetics, respectively. We have to define our mappings in the `aes()` call, where we choose the `x` and `y` coordinates. We then add a geometry with the `+` operator and `geom_<type>` function, in this case choosing the type "point" for a scatterplot of `x` and `y`.

Data mapped to **aes**thetic properties of **geom**etric objects!

```
#  Scatterplot
ggplot(data = iris, mapping = aes(x = Petal.Length,  y = Petal.Width)) +
                geom_point()
```

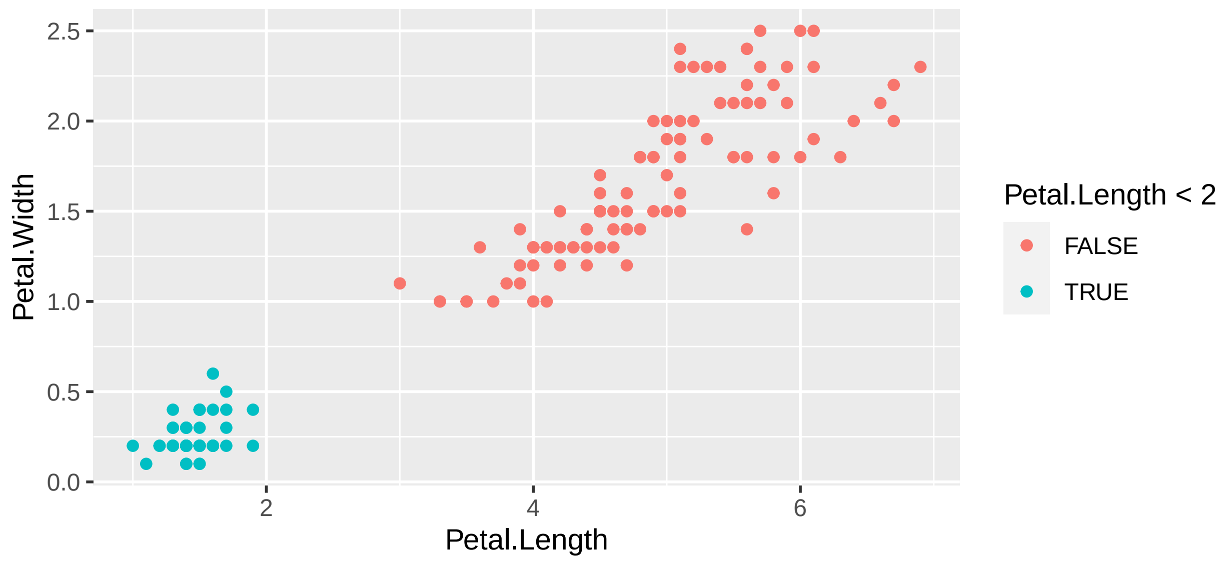


Variables (columns) are mapped within `aes()`. We can set our aesthetics in the base call, or in a layer.

```r
# Nothing is inherited from ggplot; no global mappings
ggplot() +
  geom_point(data = iris, aes(x = Petal.Length, y = Petal.Width))

# aes() assumes x,y coordinates
ggplot(iris) +
  geom_point(aes(Petal.Length, Petal.Width))
```

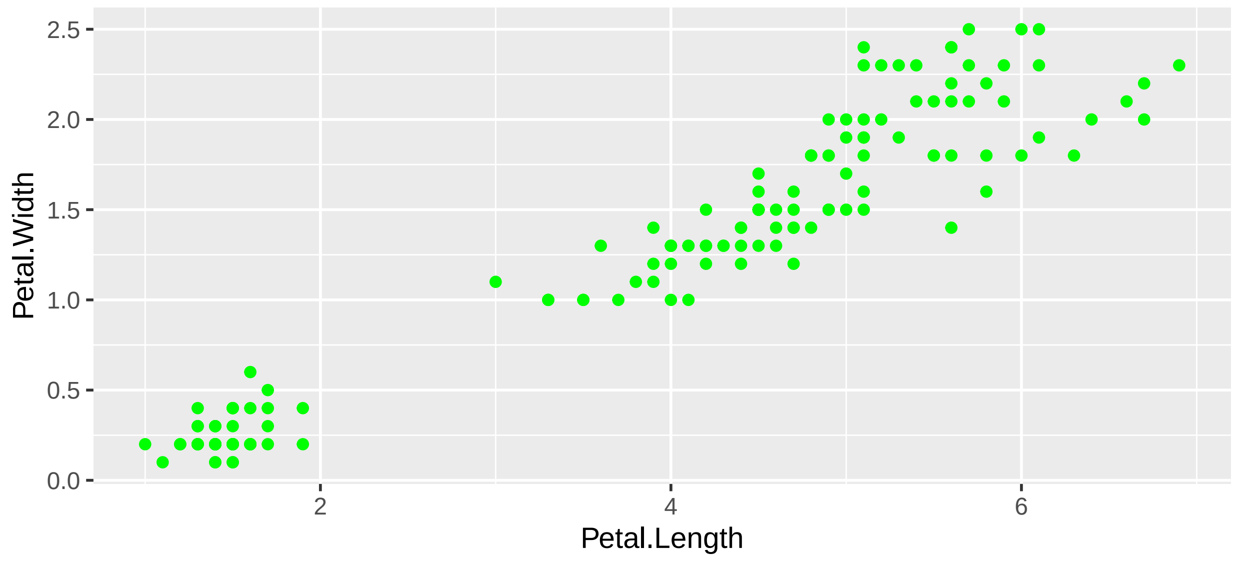
Lets look at some more aethestic mappings. You can map colour to an expression, such as colouring certain observations based on a threshold. Notice the legend takes the name of our expression.

```r
# Using an expression to colour the points
ggplot(iris) +
  geom_point(aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Petal.Length < 2))
```



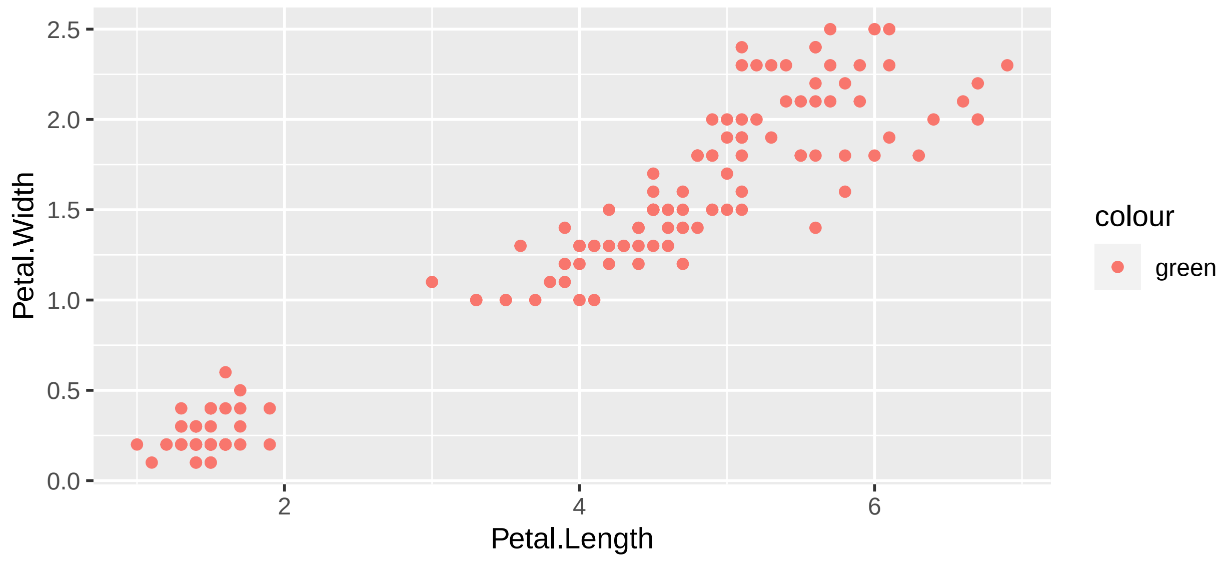
We can also manually set the colour of the points. Notice that the `colour` argument is outside of the `aes()` call.

```r
ggplot(iris) +
  geom_point(aes(x = Petal.Length, y = Petal.Width), colour = "green")
```



Try to put `colour = "green"` inside `aes()`.

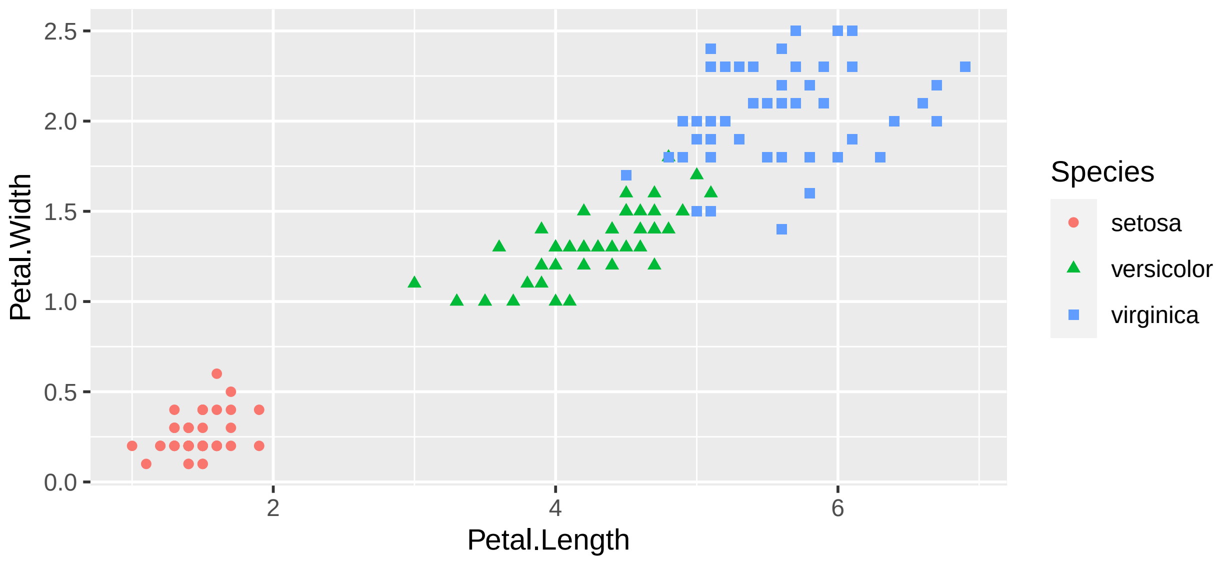
```
# Not the plot we expected
ggplot(iris) +
  geom_point(aes(x = Petal.Length, y = Petal.Width, colour = "green"))
```



The `aes()` call expects its arguments to come from the dataset. Since "green" does not exist in `iris`, it creates the variable `colour` (where every cell contains the string "green"), then the scale sets "green" to the default colour red. In short, when you want to map something from the data, put it in `aes()`, and when you want to set something manually, put it outside of `aes()`.

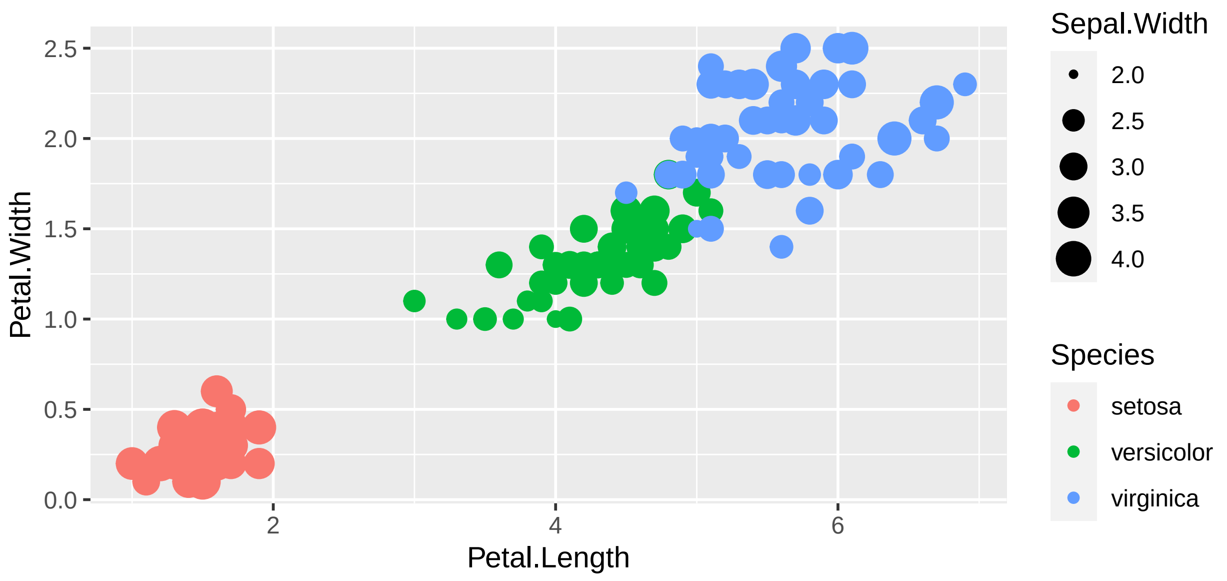
Categorical variables work well with `colour` and `shape` aesthetics, where order does not matter.

```
ggplot(iris, aes(x = Petal.Length,
                 y = Petal.Width,
                 colour = Species,
                     shape = Species)) +
    geom_point()
```



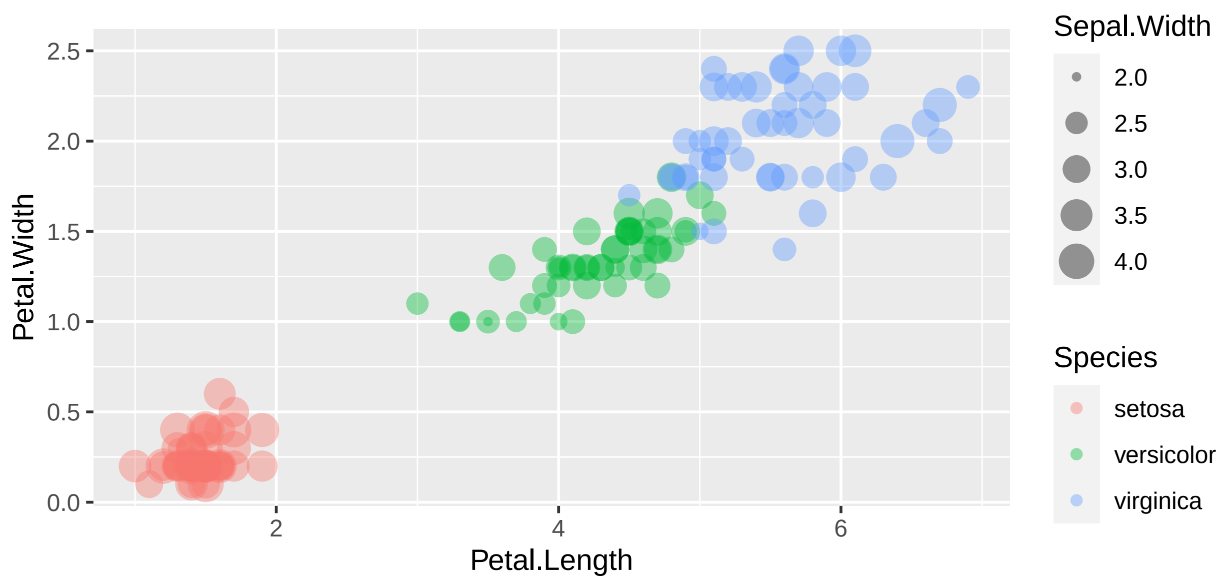
The **size** aesthetic works well with numerical variables, where order does matter.

```
# Adding aesthetic mappings to the point layer
ggplot(iris, aes(x = Petal.Length,
                     y = Petal.Width)) +
    geom_point(aes(colour = Species, size = Sepal.Width))
```



The **alpha** aesthetic adds transparency to the points. This can help when points are close together as it shows the density of points that overlap. If set manually, it goes outside of `aes()`. Try to set the alpha to Sepal.Length, within `aes`. What happens? Is this a useful plot?

```r
# Adding transparency
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
    geom_point(mapping = aes(colour = Species, size = Sepal.Width), alpha = 0.4)
```

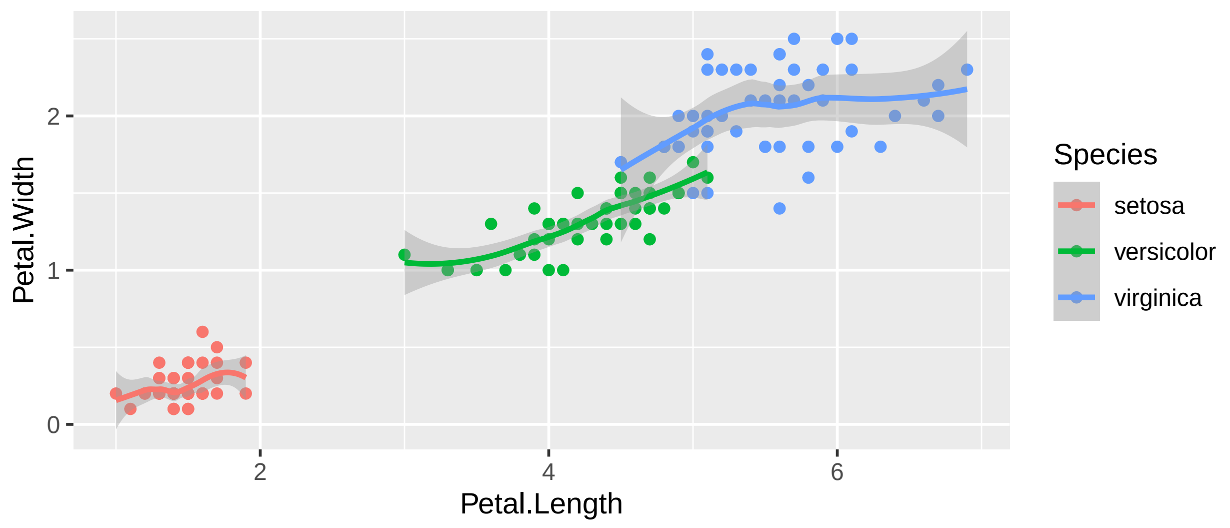


Inheritance

Calling `ggplot()` initializes a ggplot object to be used as input for data and aesthetics. This base call is global to all other layers preceeding it. This means that the `geom` layer will inherit whatever mappings we made, unless we set the mappings in the respective layer. Layers are added one by one with the `+` operator. Order does matter, as layers will be successively plotted over each other.

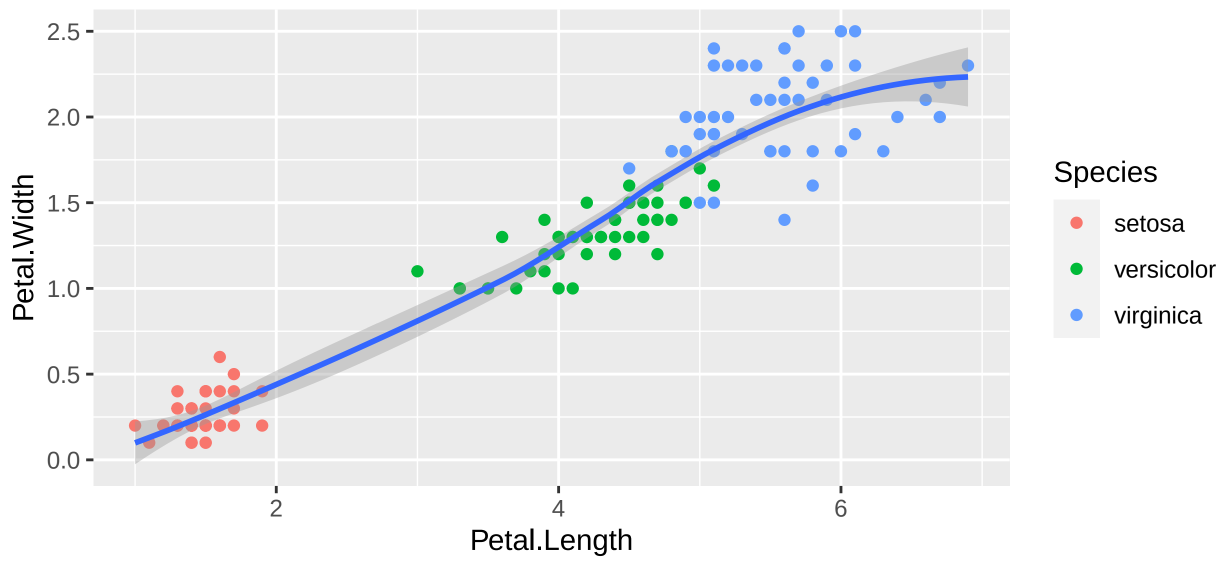
```r
# All layers will inherit these aesthetic mappings
ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) +
  geom_point() +
  geom_smooth()
```

```
## 'geom_smooth()' using method = 'loess' and formula 'y ~ x'
```



If we want to plot one line that summarizes the entire dataset, we need to change the `aes` mappings. Lets move the **colour** mapping to the `geom_point` layer so that it is no longer global. Now we get one line considering all the data.

```
# Only the point layer uses the colour mapping
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point(mapping = aes(colour = Species)) +
  geom_smooth()
```



Naturally, we can tweak ggplots to our liking. We do this by modifying the arguments in our layers. `geoms` have default settings which we can review. Run `?geom_smooth()` and look under **Arguments** to see the myriad of options that are available.Reading the documentation is extremely helpful. See if you can change the line to a linear model, remove the standard error confidence interval, and change its colour to black.

Statistical summaries

As you can imagine, different `geoms` have different mapping requirements. So far we have looked at scatterplots, where we map data to x,y coordinates. `geom_point` individually maps every row of our variable to a point. Other geoms are capable of visualizing statistical summaries of our data. E.g. Histograms, barplots, boxplots will compute these summaries and only require a single variable to be mapped. These collective geoms perform preplotting transformations using underlying stats specific to the `geom`.

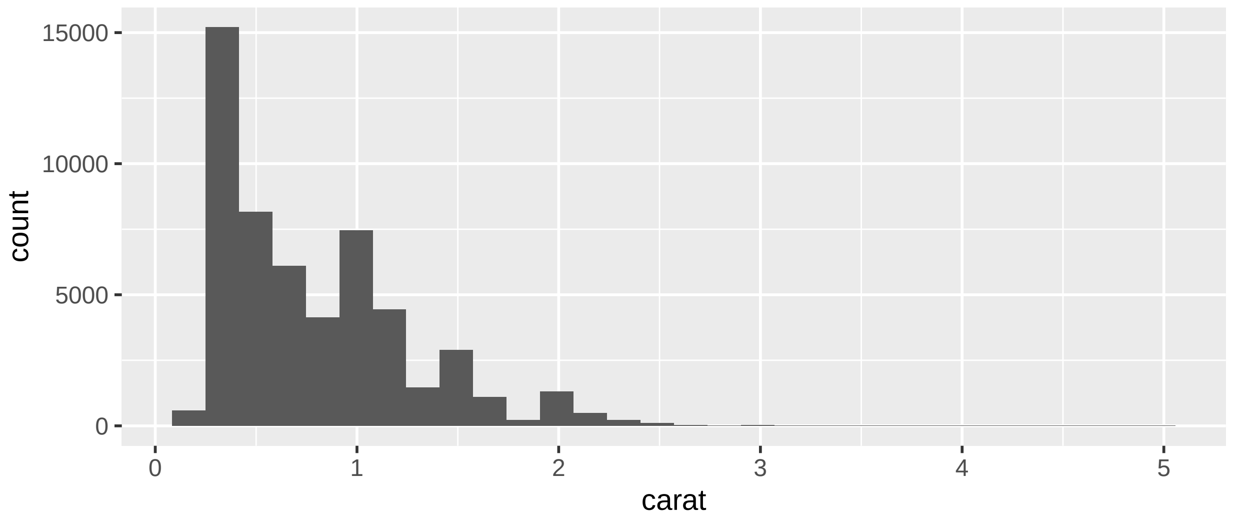
For example, a histogram will bin the data, count the number of samples that fall into each bin, and plot the height of each bin, proportional to its count. We only need to supply the one variable, and the underlying stats will do the rest.

Histograms

Lets look at the diamonds dataset, where we will assess the distribution of the `caret` variable (weight of the diamond).

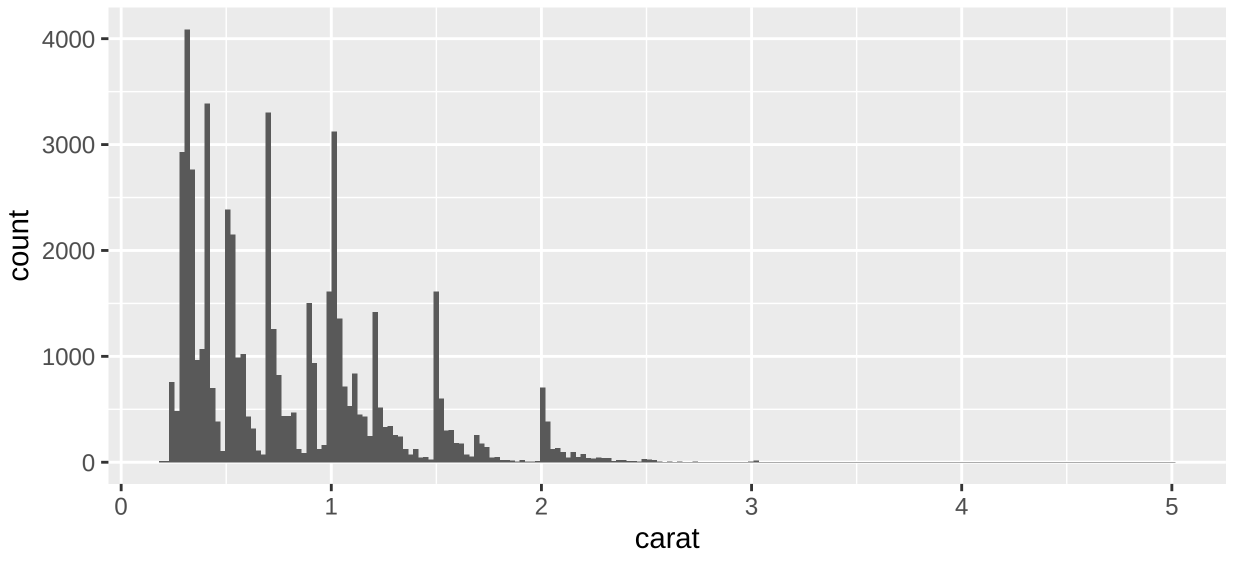
```r
data("diamonds")

# Histogram mapped to one numerical variable
ggplot(diamonds, aes(x = carat)) +
    geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



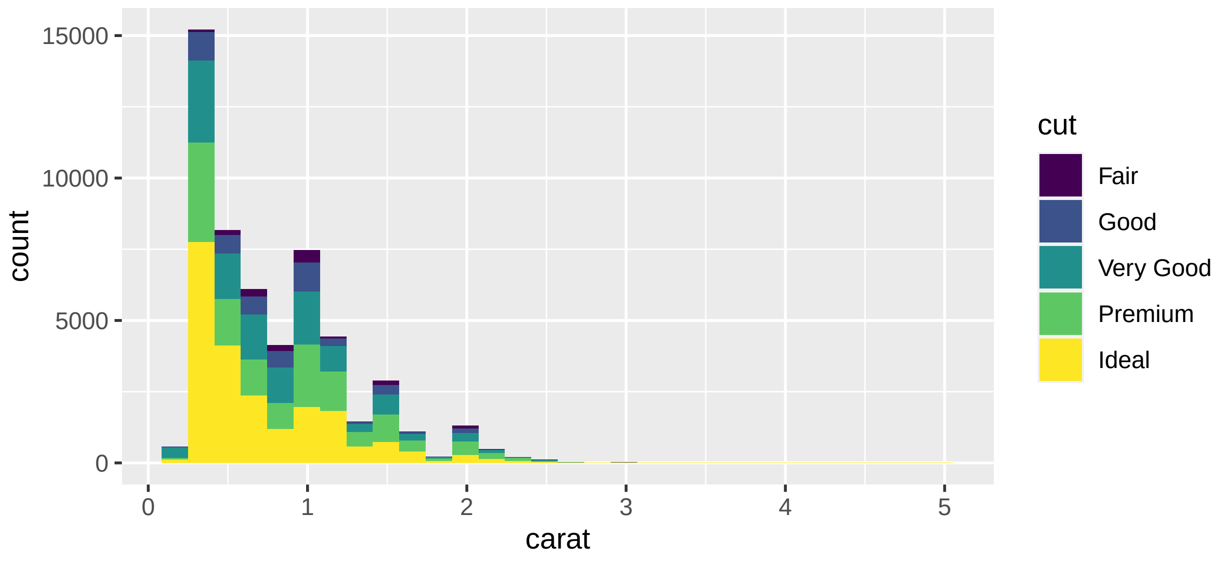
When we generate this plot, ggplot gives us a message that stat_bin was set 30. Since we didn't supply any arguments in `geom_histogram()` the function sets its own default. `ggplot` defaults to 30 bins, this may not be ideal for the data you are using so it is a good idea to always modify the bins argument in the `geom_histogram`. We see that at a higher resolution, a pattern emerges. There are more diamonds at whole numbers of caret.

```r
# Adjusting the number of bins
ggplot(diamonds, aes(x = carat)) +
    geom_histogram(bins = 200)
```



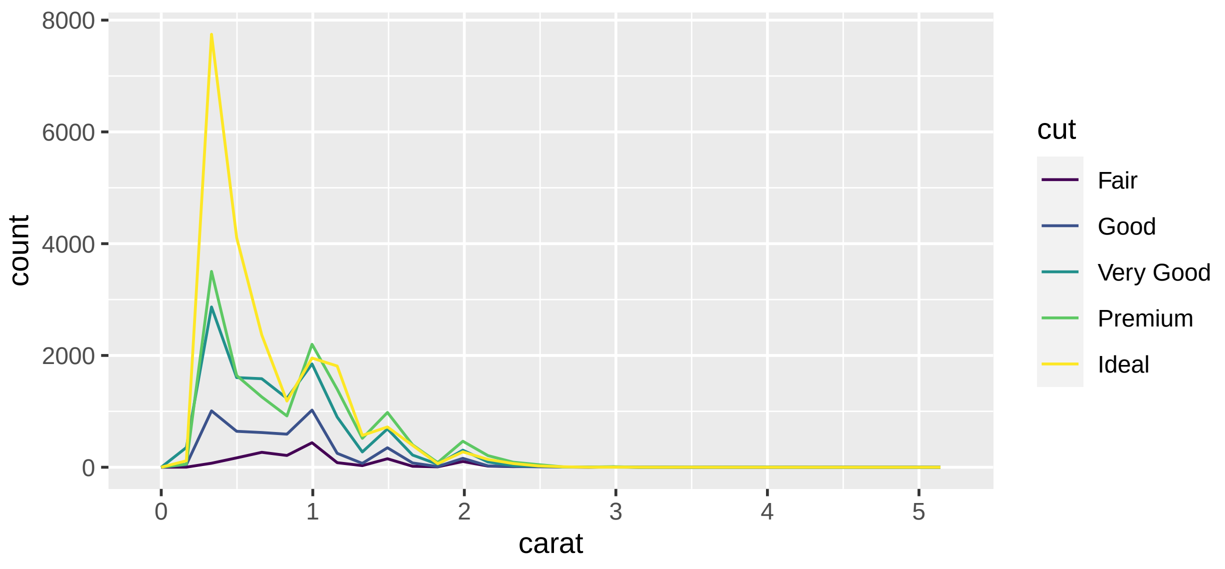
The **fill** aesthetic is used on plots that show data in polygon form (bars), rather than points or lines. When set to a categorical variable in a histogram, the default is a stacked histogram. Try to change the position to dodge (hint: run `?geom_histogram` and look at the position argument).

```
# Stacked histogram
ggplot(diamonds, aes(x = carat, fill = cut)) +
    geom_histogram()
```



`geom_freqpoly` is useful for visualizing the distribution of a single numerical variable across the levels of a categorical variable.

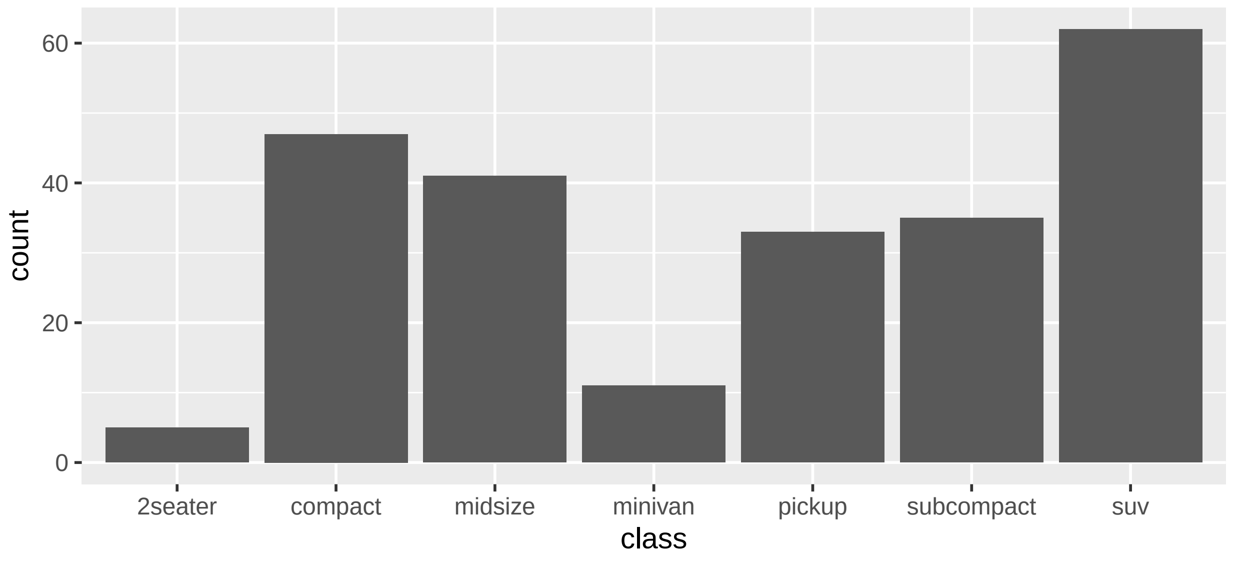
```
# Frequency polygon
ggplot(diamonds, aes(x = carat,  colour = cut)) +
    geom_freqpoly()
```



Barplots

Barplots are similar to histograms in that they bin and sort samples. The main difference is that the aesthetics of `geom_bar` can be set to a categorical variable so that each bar is a level of that variable. Lets look at the `mpg` dataset, where we might be interested in comparing the frequency of different types of vehicles.

```
# The class variable is categorical, with each level being a different type of vehicle
data(mpg)
ggplot(mpg) +
  geom_bar(aes(x = class))
```



Notice that both `geom_histogram` and `geom_bar` aesthetics only require us to map one variable. Internally, they are both using `stat_count` to count the bins and then plot their heights on the perpendicular axis. That is why you are able to use `stat_count` directly and produce the same plot.

```r
# We can also directly use stat_count...
ggplot(mpg) +
    stat_count(aes(x = class))
```

Sometimes we are dealing with data that has precomputed counts, thus we need to change the `stat`. Lets illustrate this by precomputing the counts from the plot above.

```r
# loading dplyr for some data wrangling
library(dplyr)

# Generating precomputed counts data
mpg_counted <- mpg %>%
  count(class, name = 'count')
mpg_counted
```

```
## # A tibble: 7 x 2
##   class      count
##   <chr>      <int>
## 1 2seater        5
## 2 compact       47
## 3 midsize       41
## 4 minivan       11
## 5 pickup        33
## 6 subcompact    35
## 7 suv           62
```

```r
# Change the stat to identity to recognize precomputed data
ggplot(mpg_counted) +
  geom_bar(aes(x = class, y = count), stat = 'identity')
```

This is such a common case that another `geom` exists for this purpose.

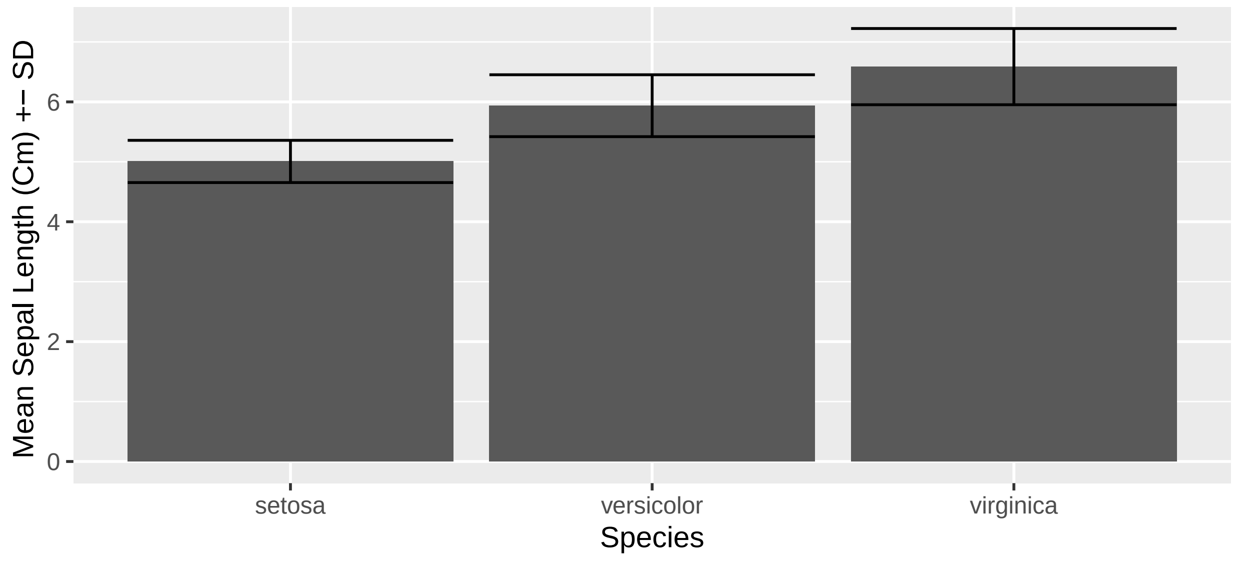
```r
# geom_col is made for precomputed data
ggplot(mpg_counted) +
  geom_col(aes(x = class, y = count))
```

Using precomputed variables gives us the freedom to plot almost anything. Going back to `iris`, lets say we want a barplot across Species of the mean `Sepal.Length` with error bars. This can be accomplished by precomputing the mean and standard deviation as new variables, and then passing those to ggplot.

```r
# More data wrangling...
library(dplyr)
iris_stats <- iris %>%
    group_by(Species) %>%
    summarise_all(list(my_mean = mean,
                       my_sd = sd))
iris_stats
```

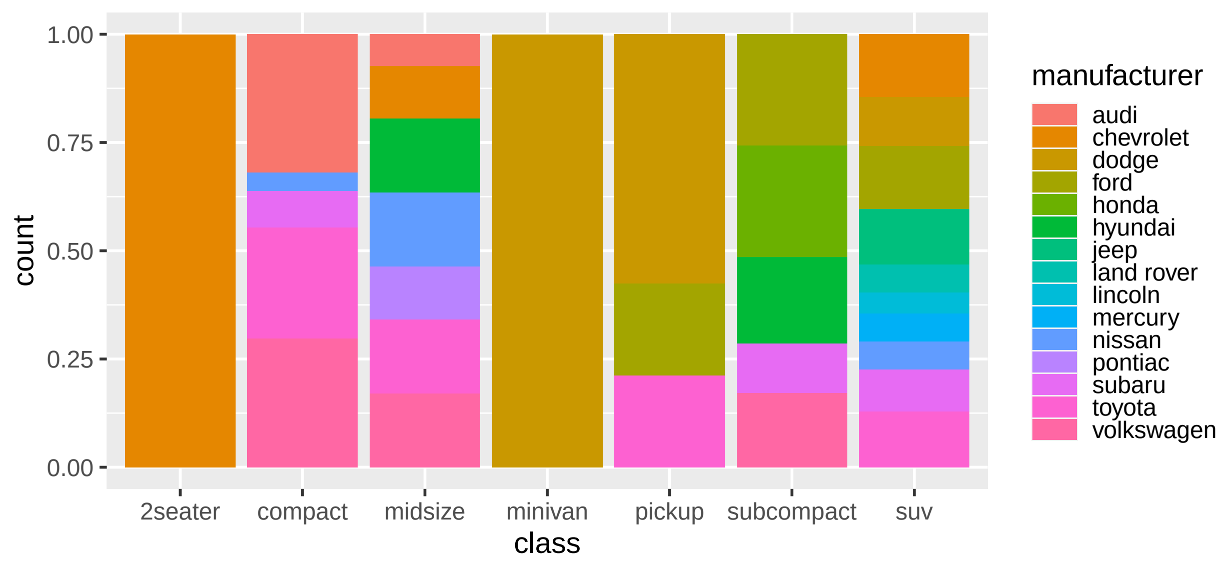
```
## # A tibble: 3 x 9
##   Species    Sepal.Len~1 Sepal~2 Petal~3 Petal~4 Sepal~5 Sepal~6 Petal~7 Petal~8
##   <fct>            <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
## 1 setosa           5.01    3.43    1.46    0.246    0.352    0.379    0.174    0.105
## 2 versicolor       5.94    2.77    4.26    1.33     0.516    0.314    0.470    0.198
## 3 virginica        6.59    2.97    5.55    2.03     0.636    0.322    0.552    0.275
## # ... with abbreviated variable names 1: Sepal.Length_my_mean,
## #   2: Sepal.Width_my_mean, 3: Petal.Length_my_mean, 4: Petal.Width_my_mean,
## #   5: Sepal.Length_my_sd, 6: Sepal.Width_my_sd, 7: Petal.Length_my_sd,
## #   8: Petal.Width_my_sd
```

```
# Barplot with errorbars
ggplot(iris_stats, aes(x = Species, y = Sepal.Length_my_mean)) +
    geom_col() +
    # We have to set the upper and lower bounds of our error bar
    geom_errorbar(aes(ymin = Sepal.Length_my_mean - Sepal.Length_my_sd,
                      ymax = Sepal.Length_my_mean + Sepal.Length_my_sd)) +
    ylab("Mean Sepal Length (Cm) +- SD ")
```



Ironically, simple plots can be complex to make, and complex plots can be simple to make.

```
# Stacked barplot, filling the plot area
ggplot(mpg) +
  geom_bar(aes(x = class, fill = manufacturer), position = "fill") +
    # shrink legend to fit on page
    theme(legend.key.height = unit(0.1,"cm"))
```

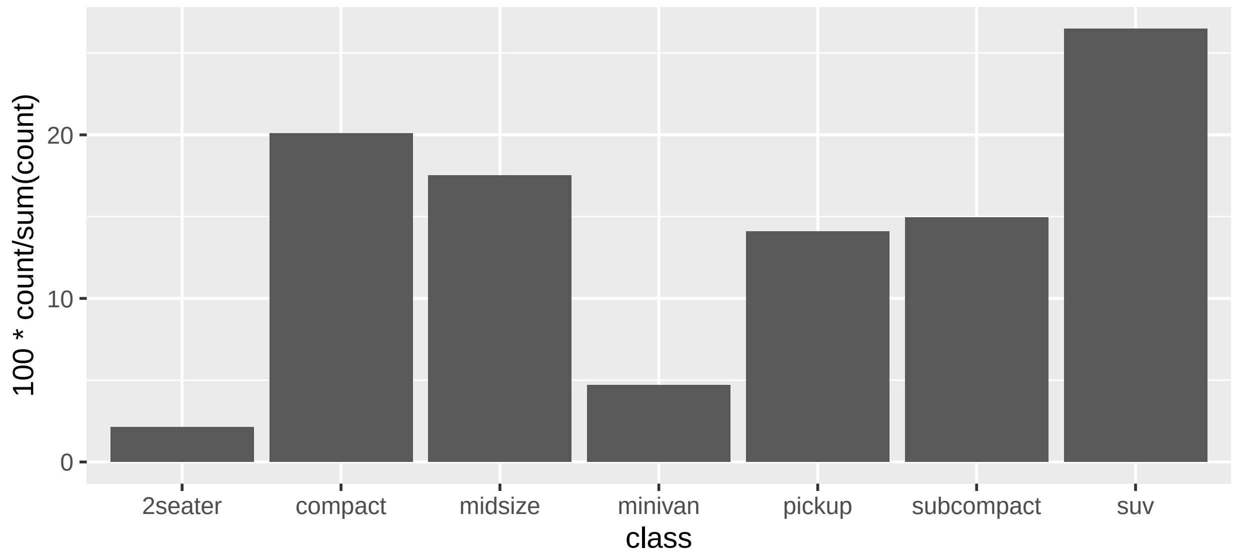


Modifying stats

A `geom` can have its `stat` modified directly with `after_stat`. Lets change our bars to frequencies to proportions, so that every bar will sum to 100.

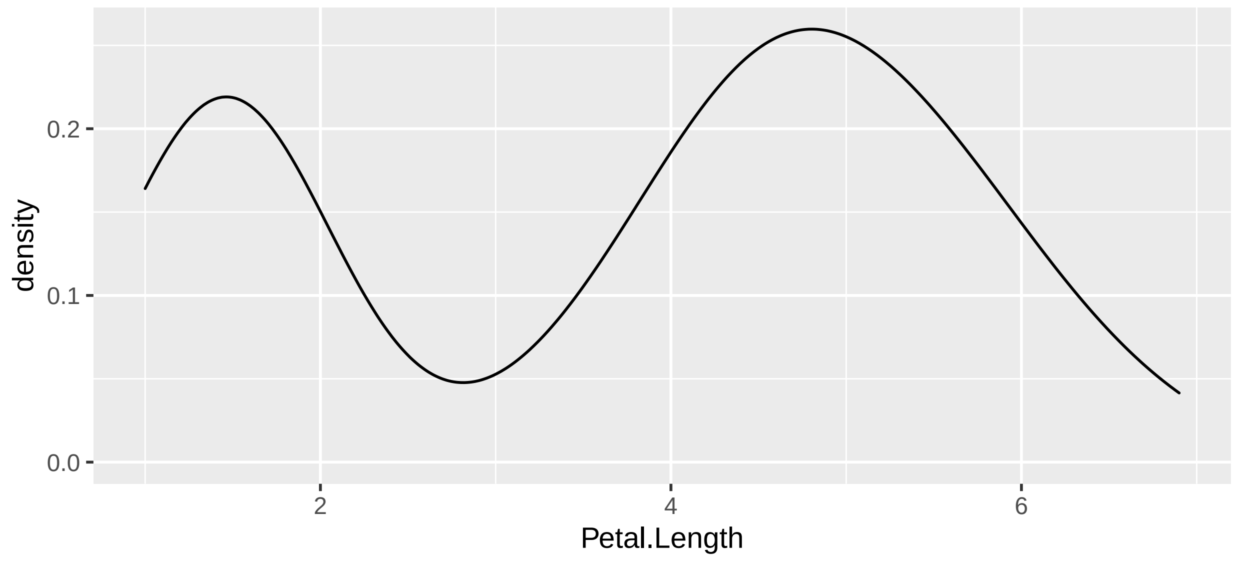
To adjust the `stat` of a `geom` you have to know what computed variables are used. Run `?geom_bar` and look under **Computed Variables**. We use `after_stat()` within `aes()`to modify the stat.

```
# You may have thought I was done showing this plot...
ggplot(mpg) +
# Calculating proportions
  geom_bar(aes(x = class, y = after_stat(100 * count / sum(count))))
```

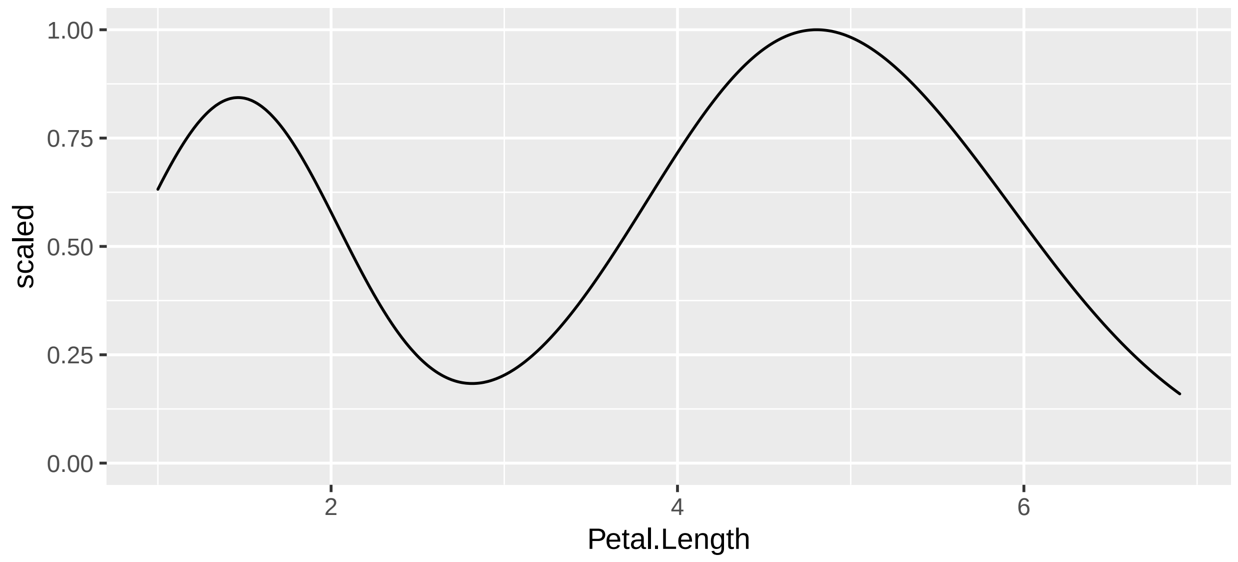


A `geom_density` plot computes a smoothed histogram of your data. This places a normal distributions at each point and sums up all the curves. Compared to histograms, density plots are prettier, but harder to interpret and have underlying assumptions about the data (continuous, unbounded, and smooth). Interestingly, we can scale the density to a maximum value of one by overriding the default stat. Look at the **y** axis to see the difference.

```r
# Default
ggplot(iris) +
  geom_density(aes(x = Petal.Length))
```

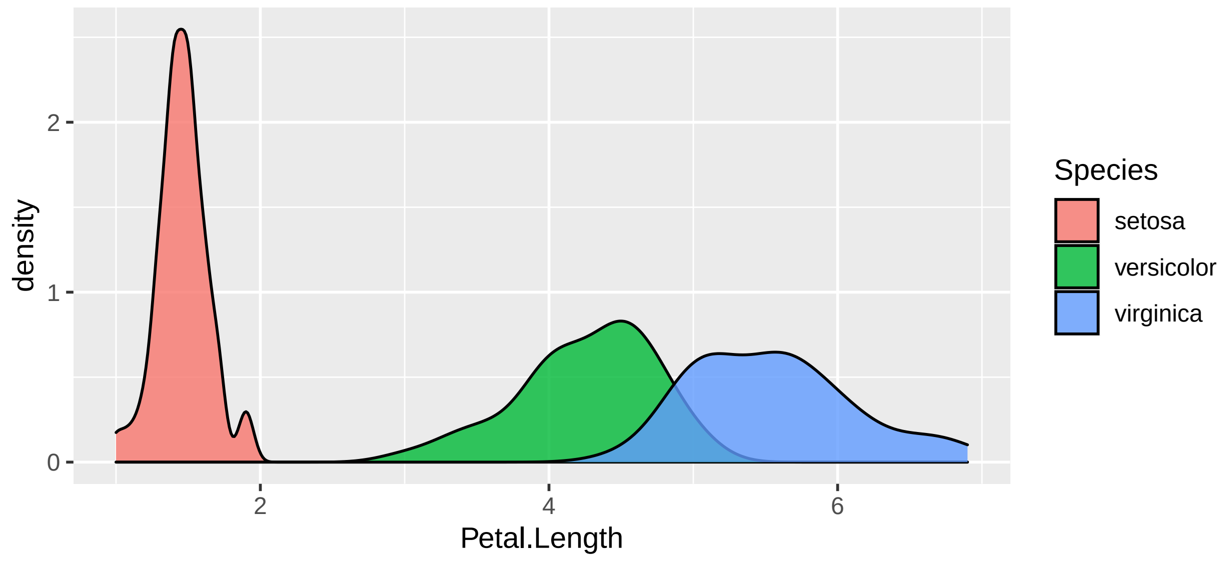


```r
# Scaled estimate
ggplot(iris) +
  geom_density(aes(x = Petal.Length, y = after_stat(scaled)))
```

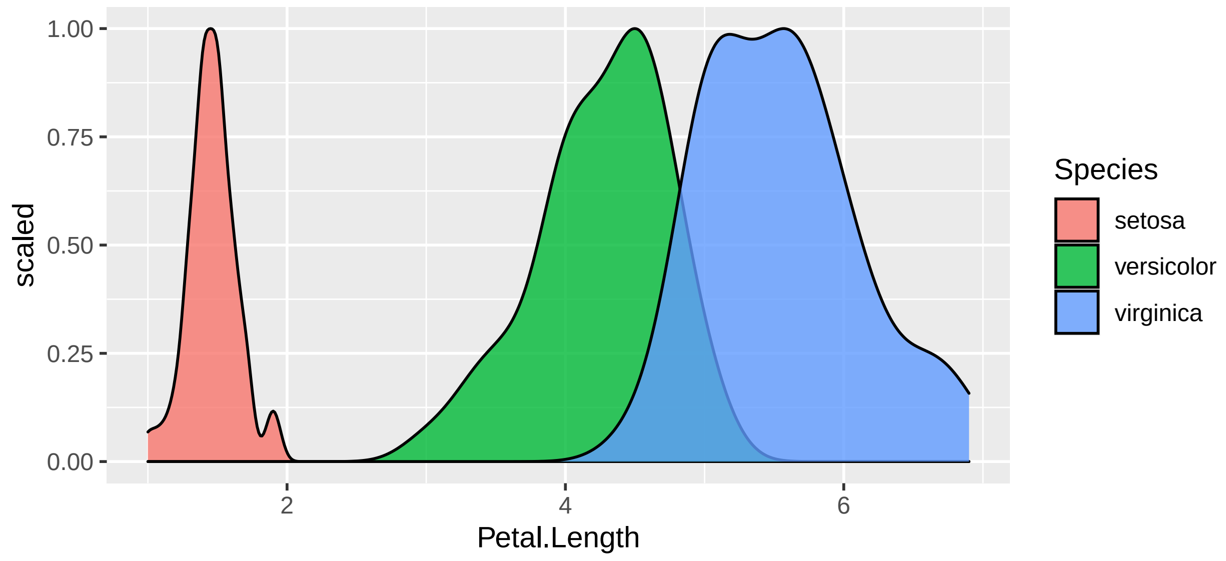


Using the scaled estimate does produce some undesirable properties. Since each density estimate is scaled to one, you lose info about the relative size of each group.

```r
# A fine plot
ggplot(iris) +
  geom_density(aes(x = Petal.Length, fill = Species),
               alpha = 0.8)
```

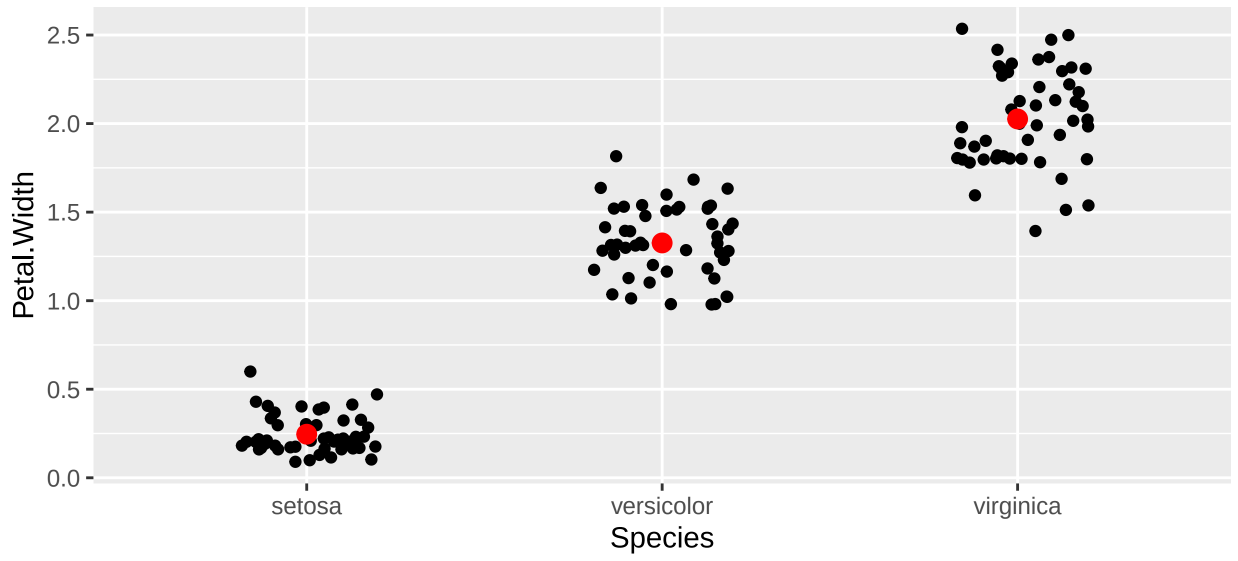


```r
# Not a good plot
ggplot(iris) +
  geom_density(aes(x = Petal.Length, y = after_stat(scaled), fill = Species),
               alpha = 0.8)
```



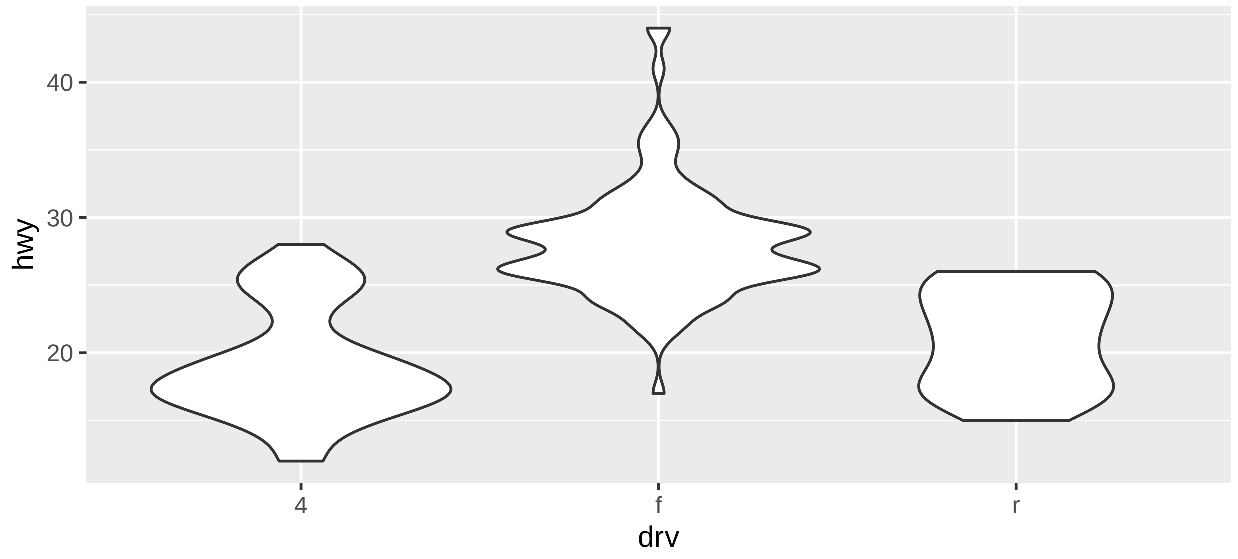
Aside from manipulating stats within a given geom, we can generate our own stats and plot them directly. We are using `stat_summary` to plot the mean Petal.Width as a red dot for each species. I am also introducing `geom_jitter`. Try to figure out what it is doing.

```r
ggplot(iris, aes(x = Species, y = Petal.Width)) +
  geom_jitter(width = 0.2) +
    stat_summary(fun = mean, geom = "point", colour = "red", size = 3)
```

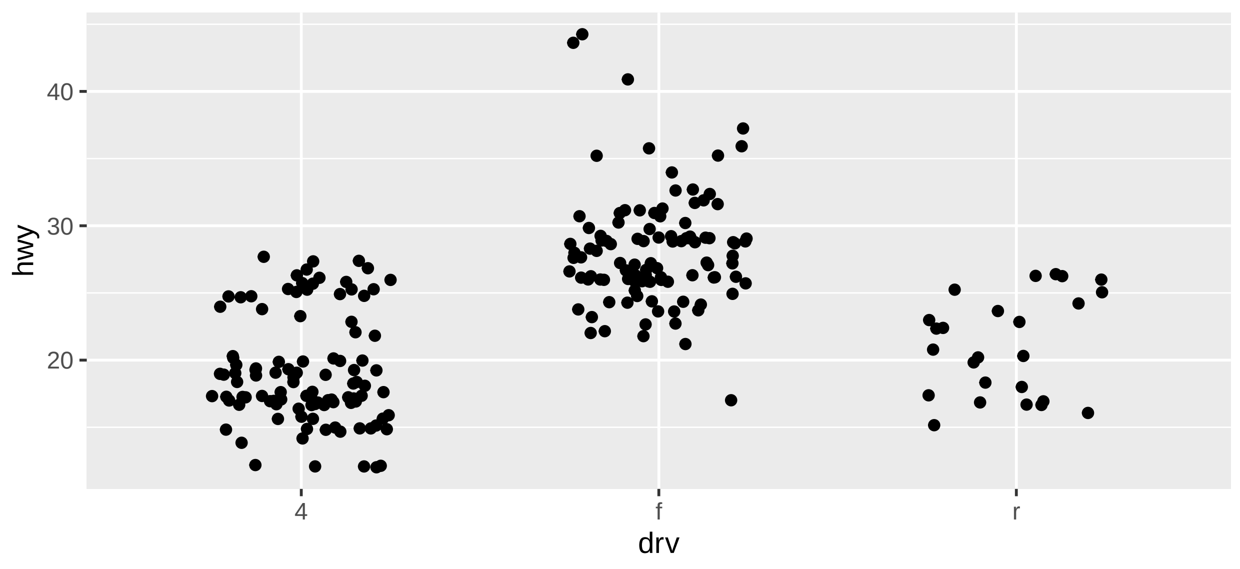


Here are some other `geoms` worth mentioning: `geom_violin` and `geom_jitter`. Violin plots are essentially vertical density plots and are sensitive to small datasets. Conversely, jitter plots work well for small datasets.

```r
ggplot(mpg, aes(x = drv,y = hwy)) + geom_violin()
```



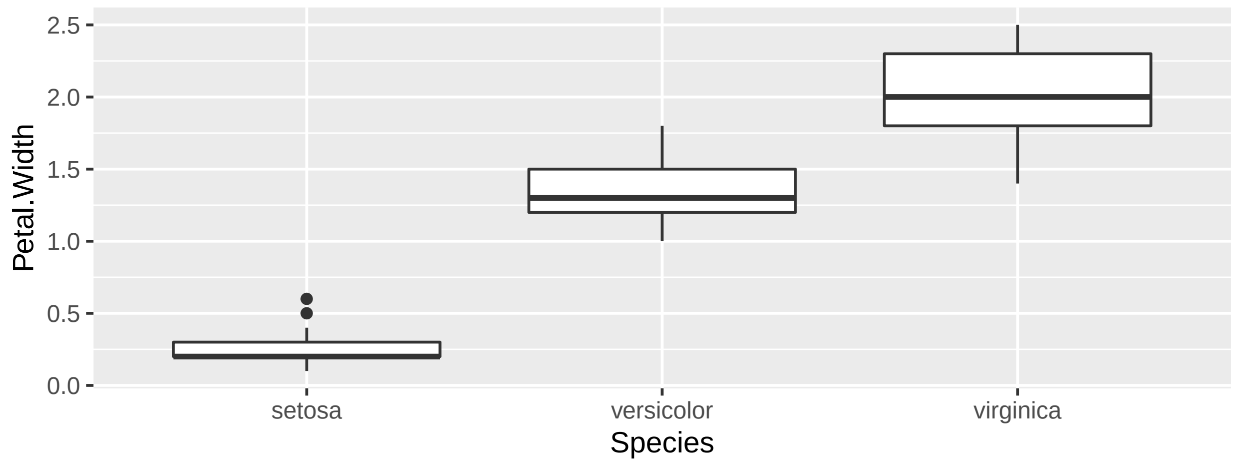
```r
ggplot(mpg, aes(x = drv, y = hwy)) + geom_jitter(width = 0.25)
```



Boxplots

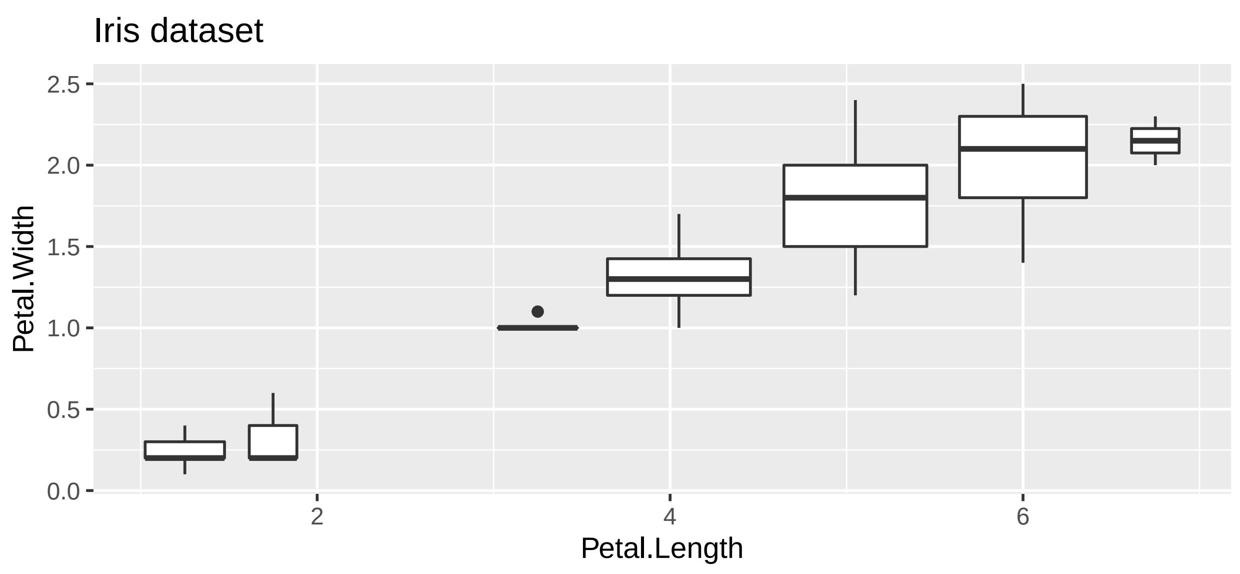
Boxplots will compute and plot their summary statistics. They can split a numerical variable across the levels of a categorical variable.

```
# Boxplot
ggplot(iris, aes(x = Species, y = Petal.Width))+
    stat_boxplot()
```

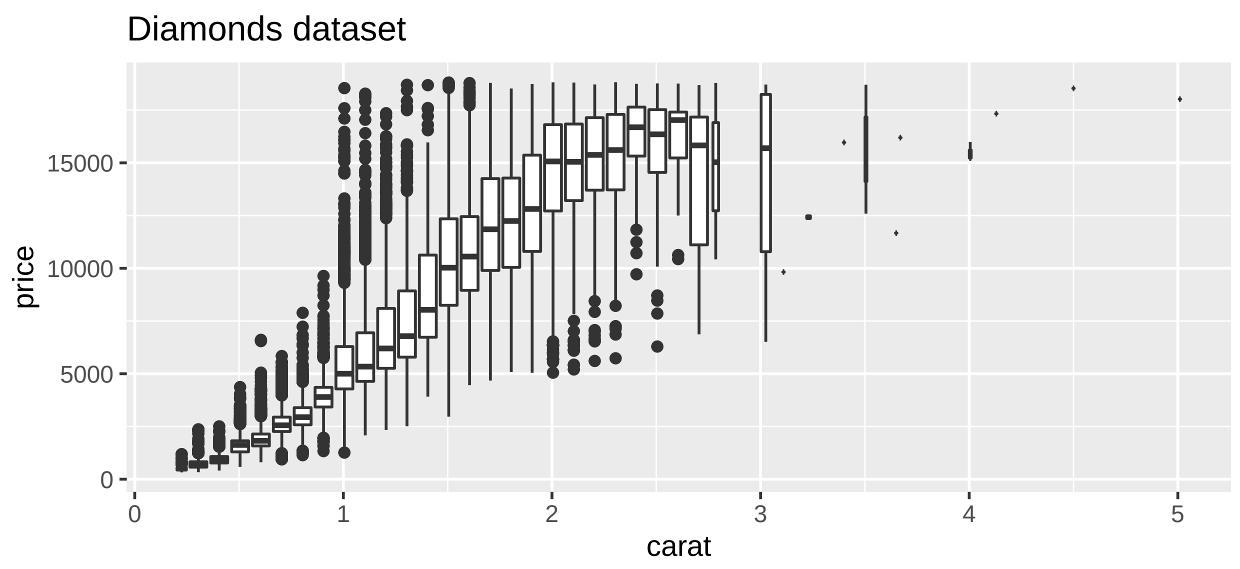


We can also split across a numerical variable by setting the `group` aesthetic to `cut_width()`. A good rule of thumb to display `n` boxplots is take the maximum value on the x-axis and divide by the number of plots you want. For the first plot, try adding a point layer and making the boxplots transparent. See how the boxplots match the position of the points?

```
# Small dataset
ggplot(iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_boxplot(aes(group = cut_width(Petal.Length, 7/7))) +
    ggtitle("Iris dataset")
```



```
# Large dataset
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_boxplot(aes(group = cut_width(carat, 0.1))) +
    ggtitle("Diamonds dataset")
```



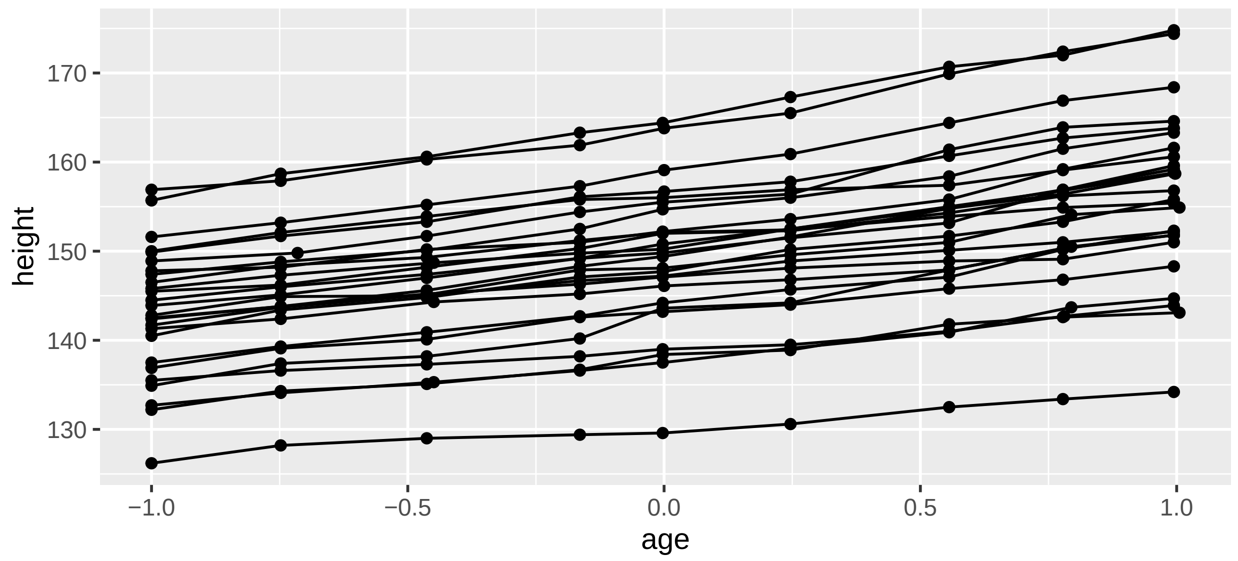
Grouping

Those last two plots introduced the `group` aesthetic, which is worth discussing. Grouping can be used to compare the same samples across multiple intervals. Lets illustrate this with the `Oxboys` dataset, which records the height and age of boys from Oxford. Try removing the `group` aesthetic and see what happens.

```
data(Oxboys, package = "nlme")
head(Oxboys)
```

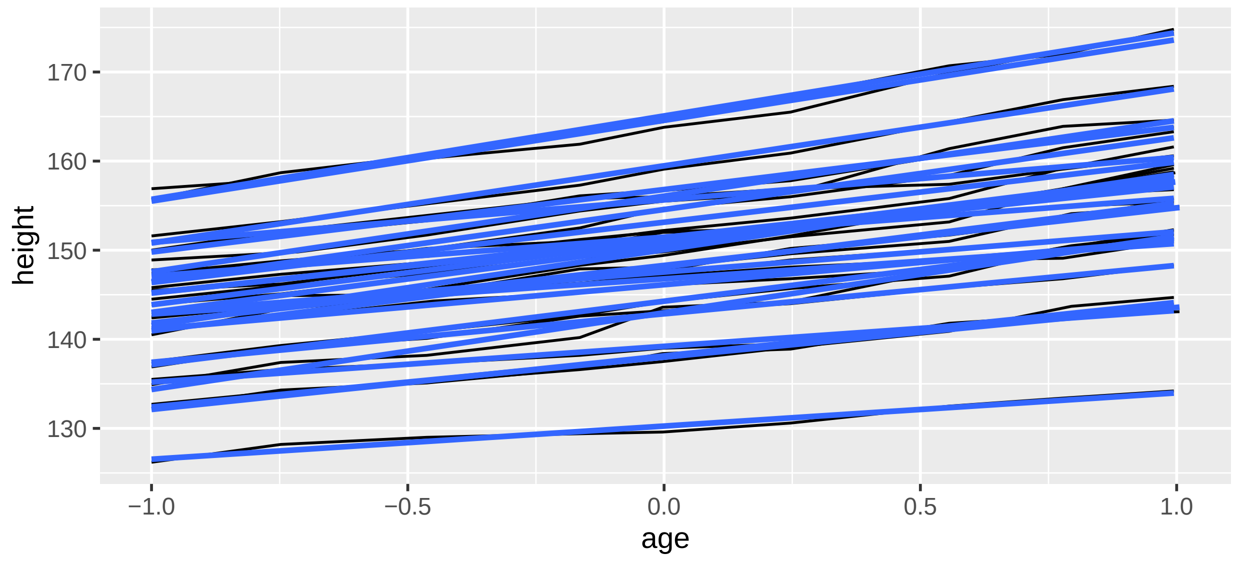
```
## Grouped Data: height ~ age | Subject
##   Subject     age height Occasion
## 1       1 -1.0000  140.5        1
## 2       1 -0.7479  143.4        2
## 3       1 -0.4630  144.8        3
## 4       1 -0.1643  147.1        4
## 5       1 -0.0027  147.7        5
## 6       1  0.2466  150.2        6
```

```
# Grouping individuals across time
ggplot(Oxboys, aes(x = age, y = height, group = Subject)) +
  geom_point() +
  geom_line()
```



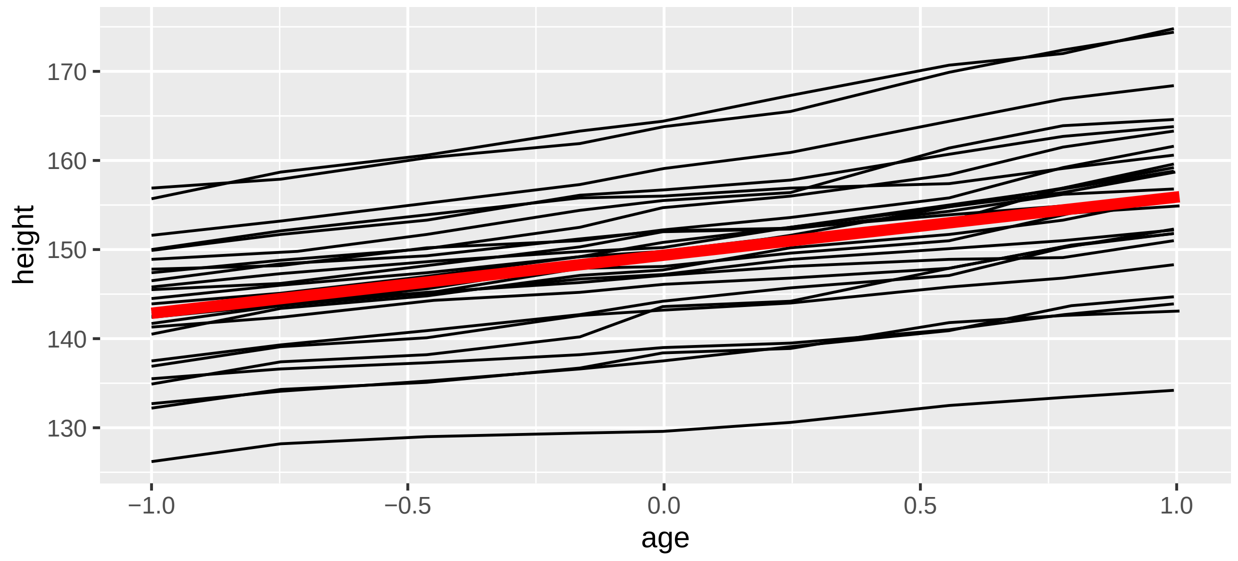
Here is another inheritance example. Say we want to fit a linear model to the data. Lets try to do that by adding a `geom_smooth` layer.

```
# Every layer is inheriting the group aesthetic...
ggplot(Oxboys, aes(x = age, y = height, group = Subject)) +
    geom_line() +
  geom_smooth(method = "lm", se = FALSE)
```



This isn't what we want. The `group` aesthetic is forcing `geom_smooth` to draw a line for each subject, where what we really want is one line describing all the data. We need to move the `group` aesthetic to its respective layer so that it isn't globally affecting `geom_smooth`.

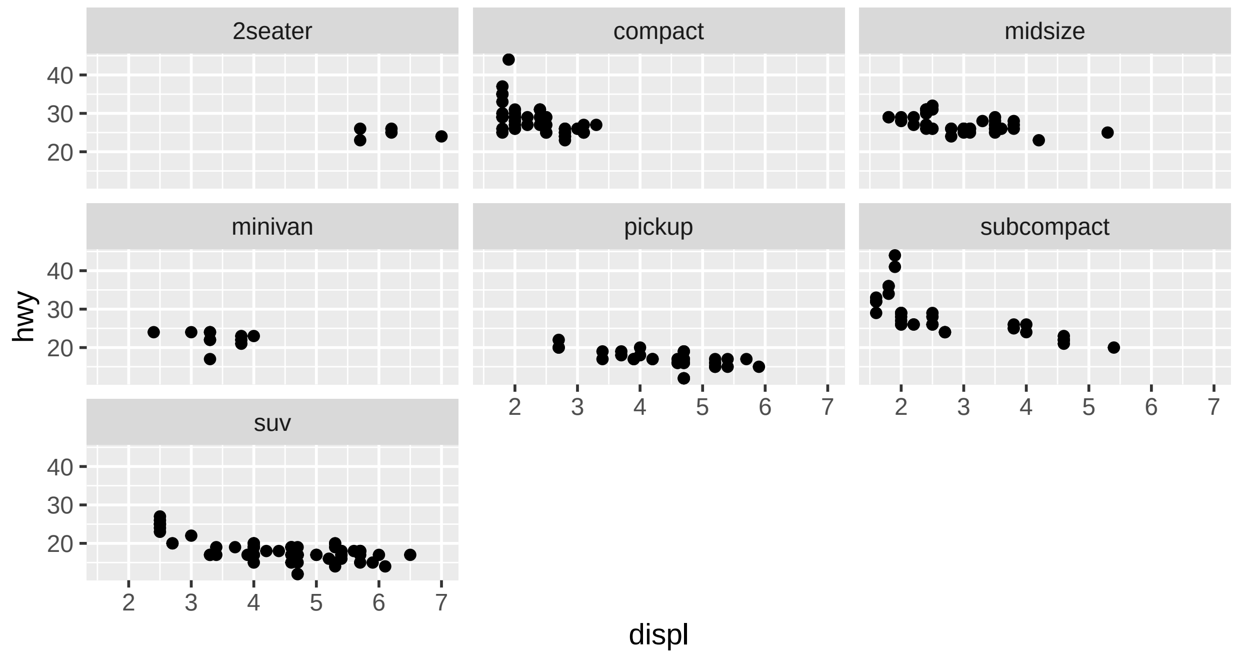
```r
# Group is local to geom_line()
ggplot(Oxboys, aes(x = age, y = height)) +
  geom_line(aes(group = Subject)) +
  geom_smooth(method = "lm", se = FALSE, size = 2, colour = "red")
```



Faceting

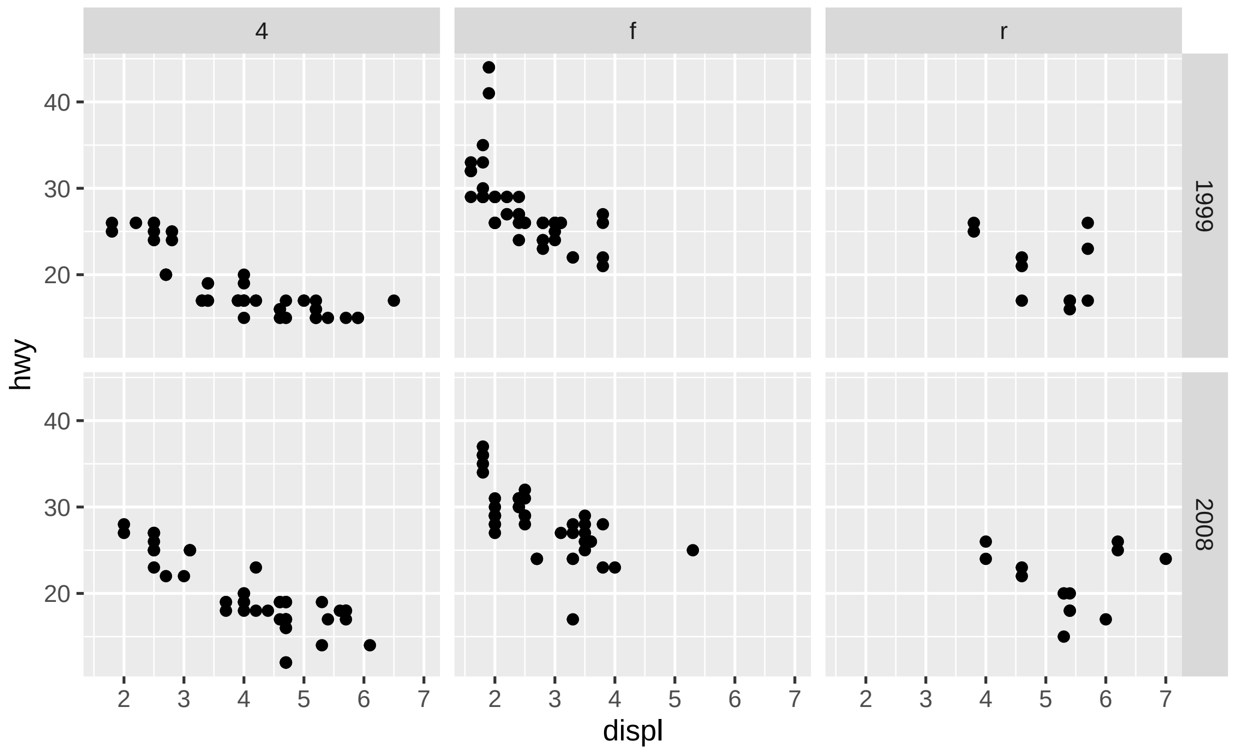
Faceting is a fantastic method to split your data into multiples. This greatly vastly improves interpretation and removes overplotting. We can split data across one categorical variable using `facet_wrap(~<variable>)` The `~` notation means that our split is *dependent* on the given variable. Lets look at engine displacement (in liters) versus highway miles/gallon, where each multiple is a different class of vehicle.

```r
# Faceting onto one variable
ggplot(mpg) +
  geom_point(aes(x = displ, y = hwy)) +
  facet_wrap(~ class)
```



With facetting, we can focus on one type of vehicle at a time without being distracted by other types. We can also facet using two categorical variables, with `facet_grid`, multiplying every combination of those two variables.

```r
# Faceting onto two variables
ggplot(mpg) +
  geom_point(aes(x = displ, y = hwy)) +
  facet_grid(year ~ drv)
```



Coordinates and scales

Every aesthetic mapping has a `scale`, which is responsible for drawing the axes and legends. There is an implied difference between scales of different data types. Again, we can modify these scales to suit our needs. I am loading in `pantheria`, which contains trait data for mammals. `pantheria` requires some preprocessing to fix variable names and filter low abundance samples at the taxonmic level of Order.

```
# Loading mammal data from traitform data package. If you are unable to install
# this package, use the pantheria dataset I have sent you.
library(traitdataform)
library(dplyr)
library(stringr)

# loading in pantheria dataset
pulldata("pantheria")
```

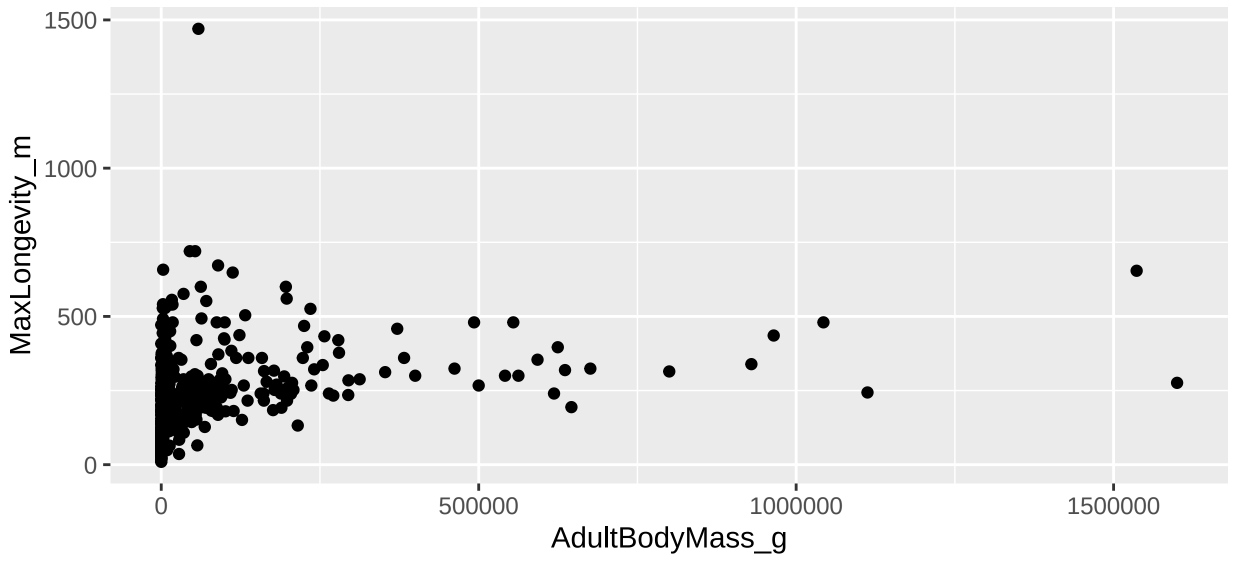
```
## The dataset 'pantheria' has successfully been downloaded!
```

```
df <- pantheria

# Removing suffix from pantheria variables
vars <- colnames(df)
vars <- str_remove(vars, "^[:alnum:]+_") %>%
  str_remove("^[:alnum:]+\\.[:digit:]_")
colnames(df) <- vars
```

```r
# Filter for order rich mammals
df <- df %>%
  group_by(Order) %>%
  filter(n() >= 200)

save(df, file = "pantheria.Rdata")

# Uncomment to load pantheria data
# load(file = "pantheria.Rdata")
```

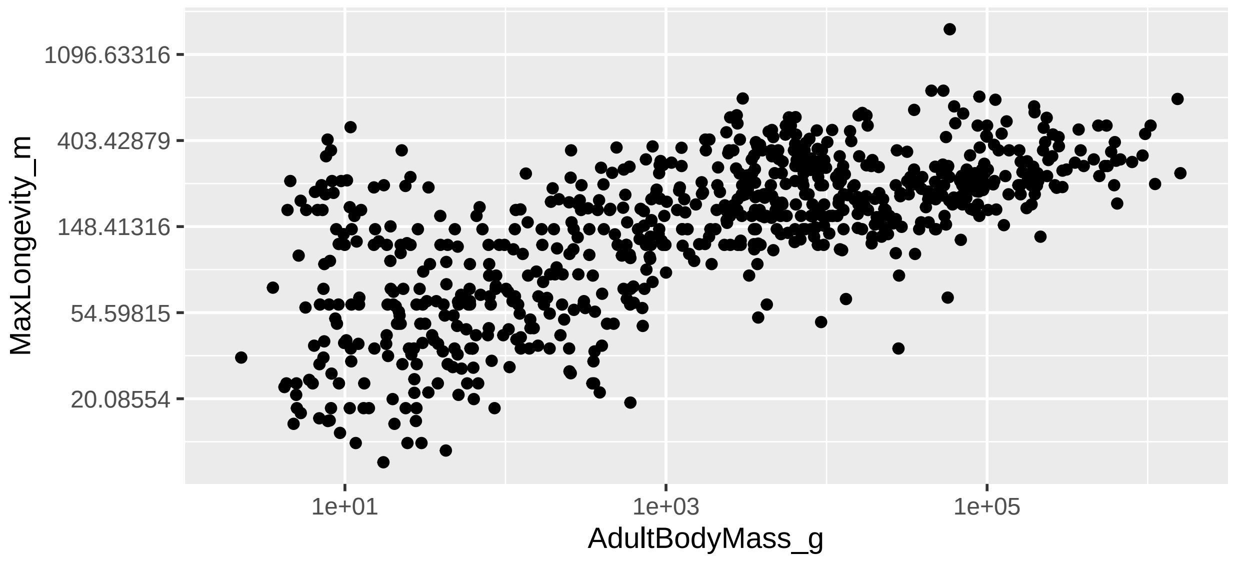
Lets look at body size (in grams), plotted against longevity (in months).

```r
# plotting body mass vs. longevity as a scatterplot
ggplot(data = df) +
  geom_point(mapping = aes(x = AdultBodyMass_g, y = MaxLongevity_m))
```



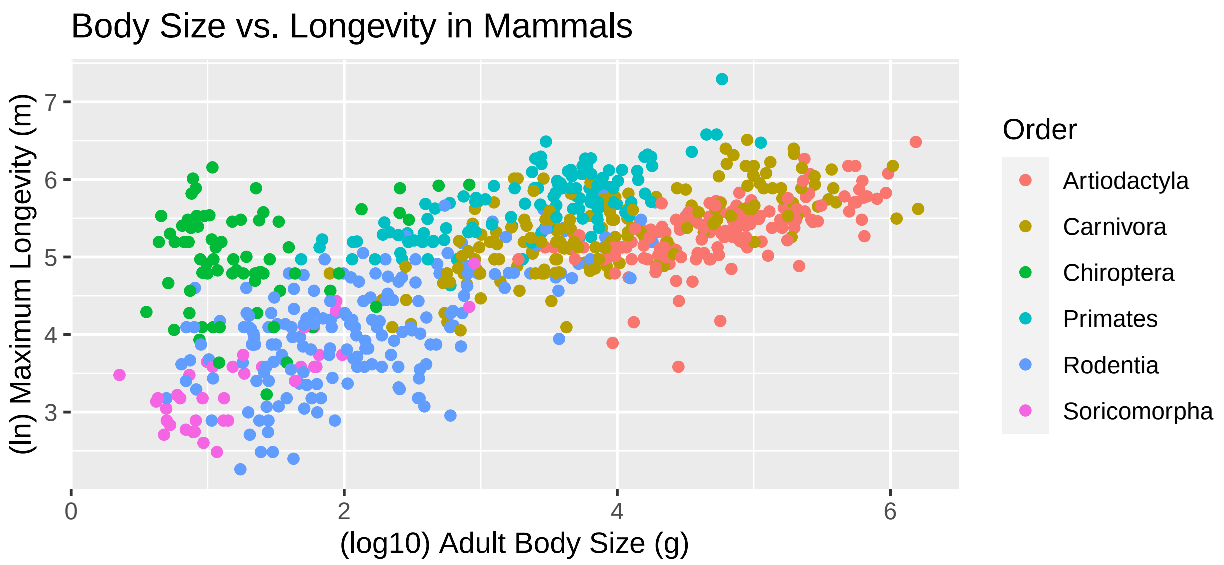
The problem here is that there are some outliers that weigh a lot (whales!), swamping other mammals and causing overplotting at the lower end of weight. We can apply a $\log_{10}$-log transformation to our scales to fix this.

```r
# Transforming the axes
ggplot(data = df) +
  geom_point(mapping = aes(x = AdultBodyMass_g, y = MaxLongevity_m)) +
    scale_x_continuous(trans = "log10") +
    scale_y_continuous(trans = "log")
```



This is better but the axes represent our data plotted on adifferent coordinate system. Another method is to apply the transformation directly in `aes()`.

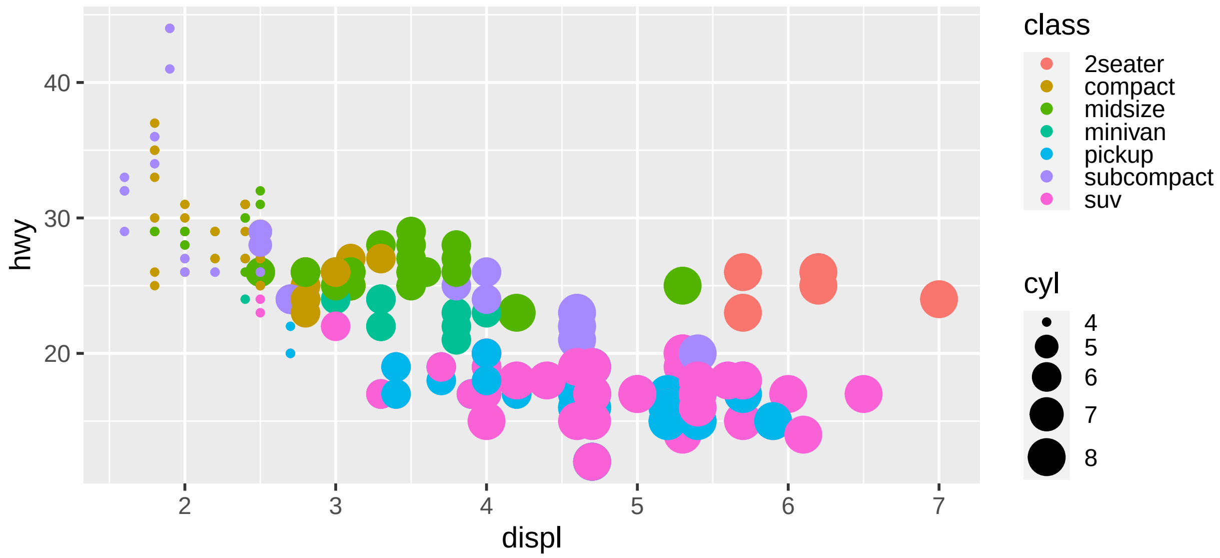
```r
# Transforming the variables
ggplot(data = df) +
  geom_point(mapping = aes(x = log10(AdultBodyMass_g),
                           y = log(MaxLongevity_m),
                           colour = Order)) +
  labs(title = "Body Size vs. Longevity in Mammals",
       x = "(log10) Adult Body Size (g)",
       y = "(ln) Maximum Longevity (m)")
```



Bats (Chiroptera) are outliers in this plot.

Here is an example of where modifying the scales is useful.

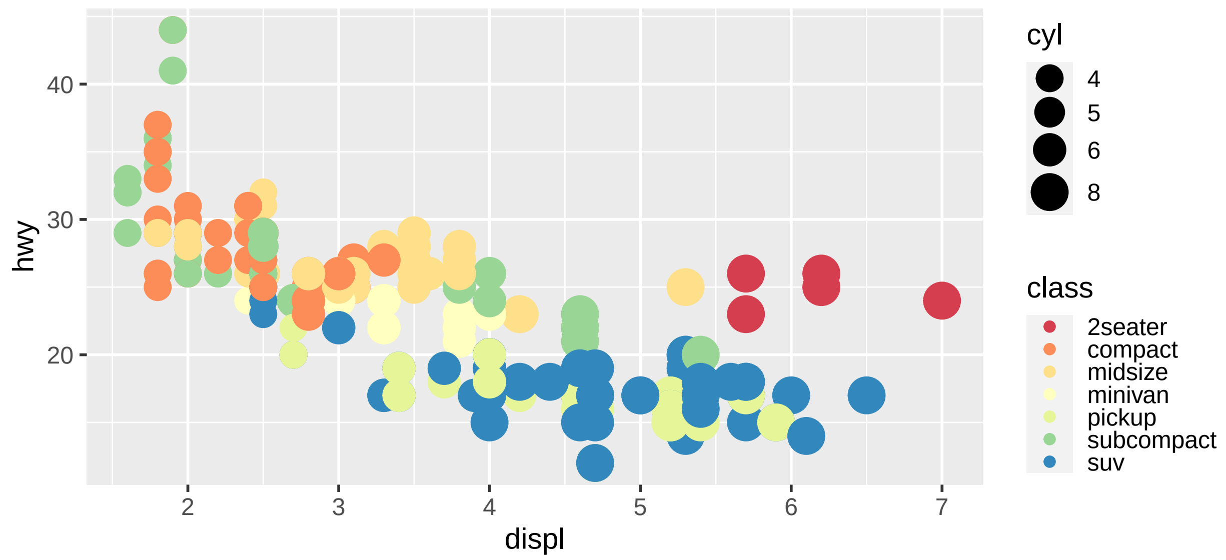
```
data("mpg")

# Not an ideal plot
ggplot(mpg) +
    geom_point(aes(x = displ, y = hwy, colour = class, size = cyl)) +
    theme(legend.key.height = unit(0.1,"cm"))
```



The problem with this plot is that there are no 7 cylinder cars, also the progression in size seems wrong. Small points are too small. Lets fix it by modifying its scales. We are also choosing another colour scale to use. You can explore all the colour scales available in `RColorBrewer` by uncommenting (Ctrl-Shift-C) and running the last line of the next code chunk.

```
library(RColorBrewer)

# Much better
ggplot(mpg) +
    geom_point(aes(x = displ, y = hwy, colour = class, size = cyl)) +
```

```r
    # Adjusting the area size and removing 7 cylinder cars
    scale_size_area(breaks = c(4,5,6,8)) +
    #
        scale_colour_brewer(palette = "Spectral") +
    theme(legend.key.height = unit(0.1,"cm"))
```

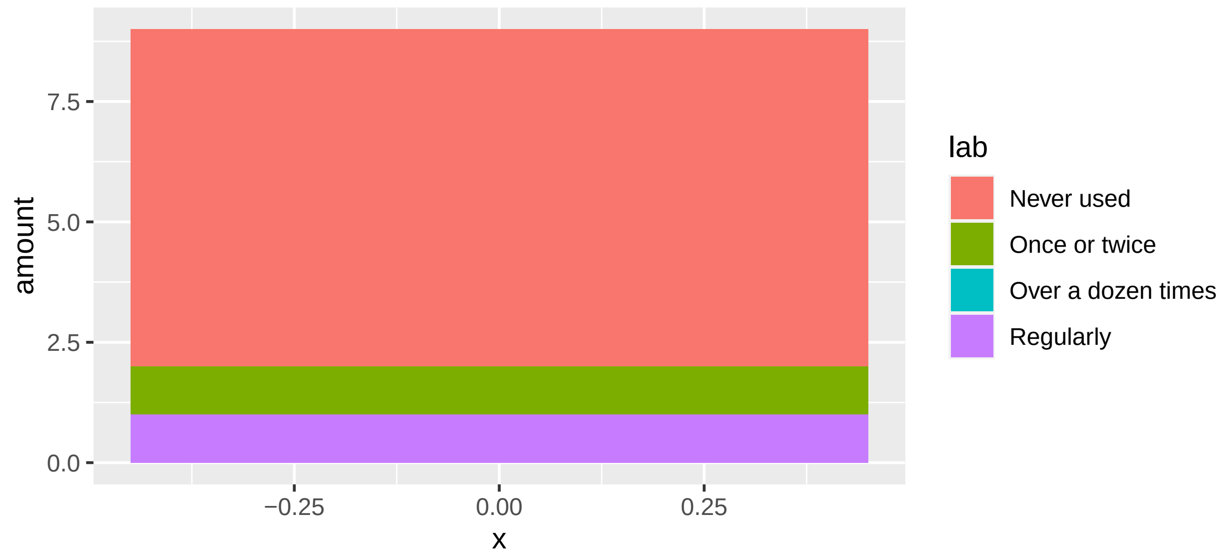


```r
# RColorBrewer::display.brewer.all()
```

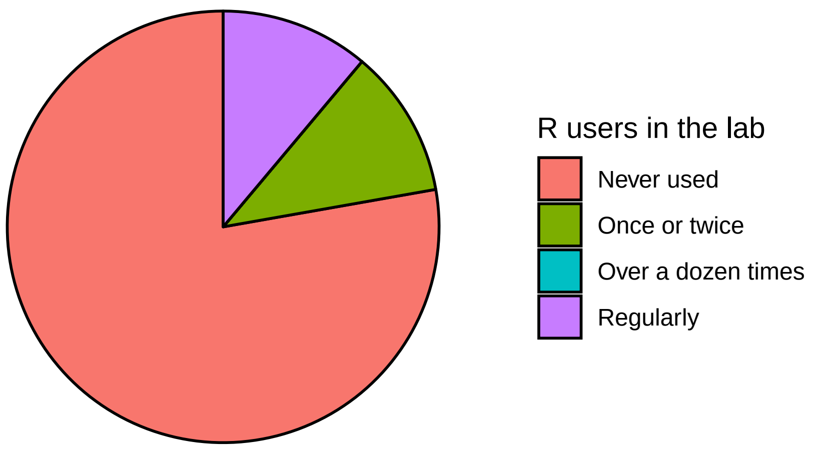
Pie chart

A pie chart can be thought of as a stacked barplot, projected onto a polar coordinate system. Lets make a pie chart from the poll I ran last week. Pie charts are not recommended due to major perceptual problems. People have a harder time quantifying area than length. Use with caution.

```r
# String vector of the poll choices
results <- c( 'Never used', 'Once or twice', 'Over a dozen times', 'Regularly'
)

#  Precomputed counts dataframe to use for plotting
pie <- data.frame(
  lab = factor(results, levels = results),
  amount = c(7, 1, 0, 1),
  stringsAsFactors = FALSE
)

# Stacked barplot
ggplot(pie) +
  geom_col(aes(x = 0, y = amount, fill = lab))
```



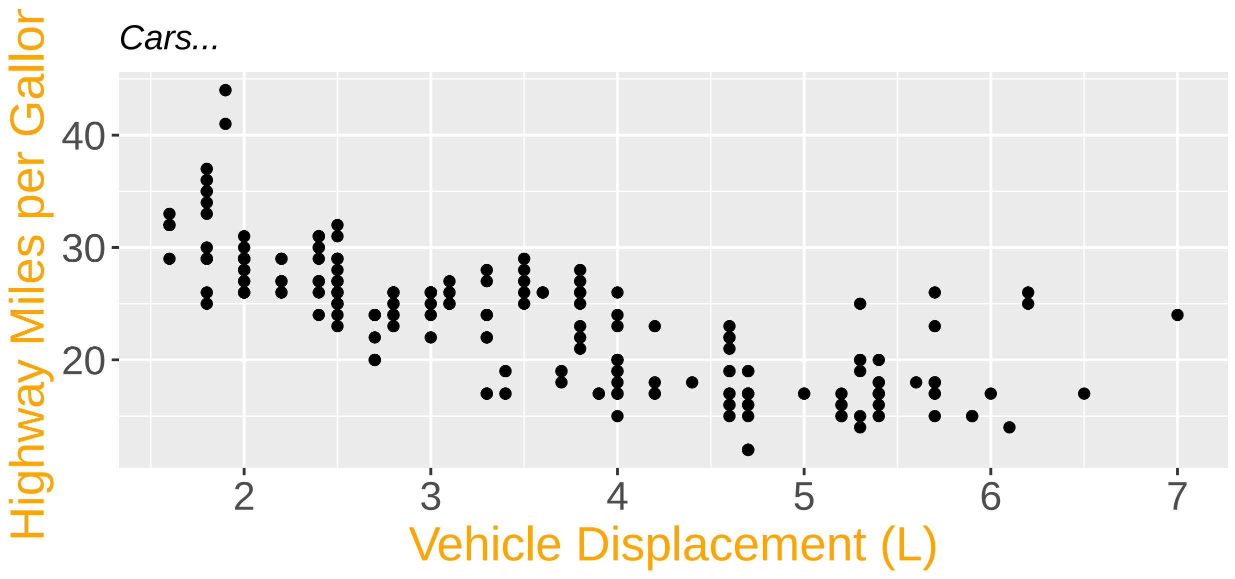
```
# Pie chart
ggplot(pie) +
  geom_col(aes(x = 0, y = amount, fill = lab), colour = "black") +
    # projects onto a polar coordinate system
    coord_polar(theta = 'y') +
    # Changes legend title
    guides(fill = guide_legend(title = "R users in the lab")) +
    # Sets a white background
    theme_bw() +
    # Removes axis info and panel graphics
    theme(axis.title.x=element_blank(),
        axis.text.x=element_blank(),
        axis.ticks.x=element_blank(),
        axis.title.y=element_blank(),
        axis.text.y=element_blank(),
        axis.ticks.y=element_blank(),
        panel.border = element_blank(),
        panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        axis.line = element_blank())
```



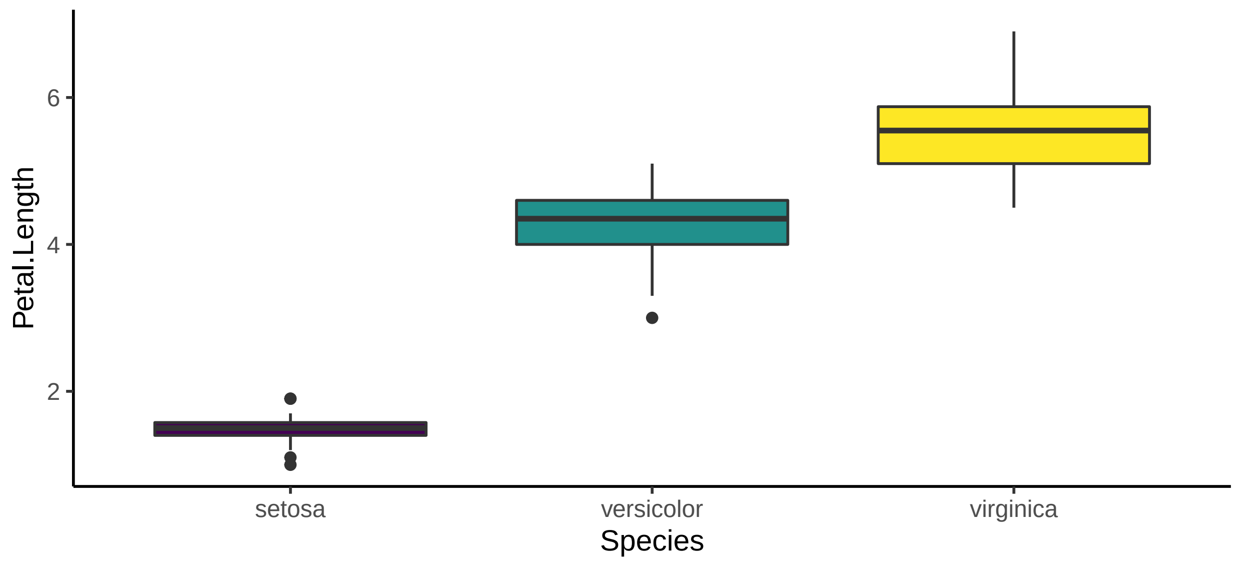
Themes

Change size of text, remove legends, etc.

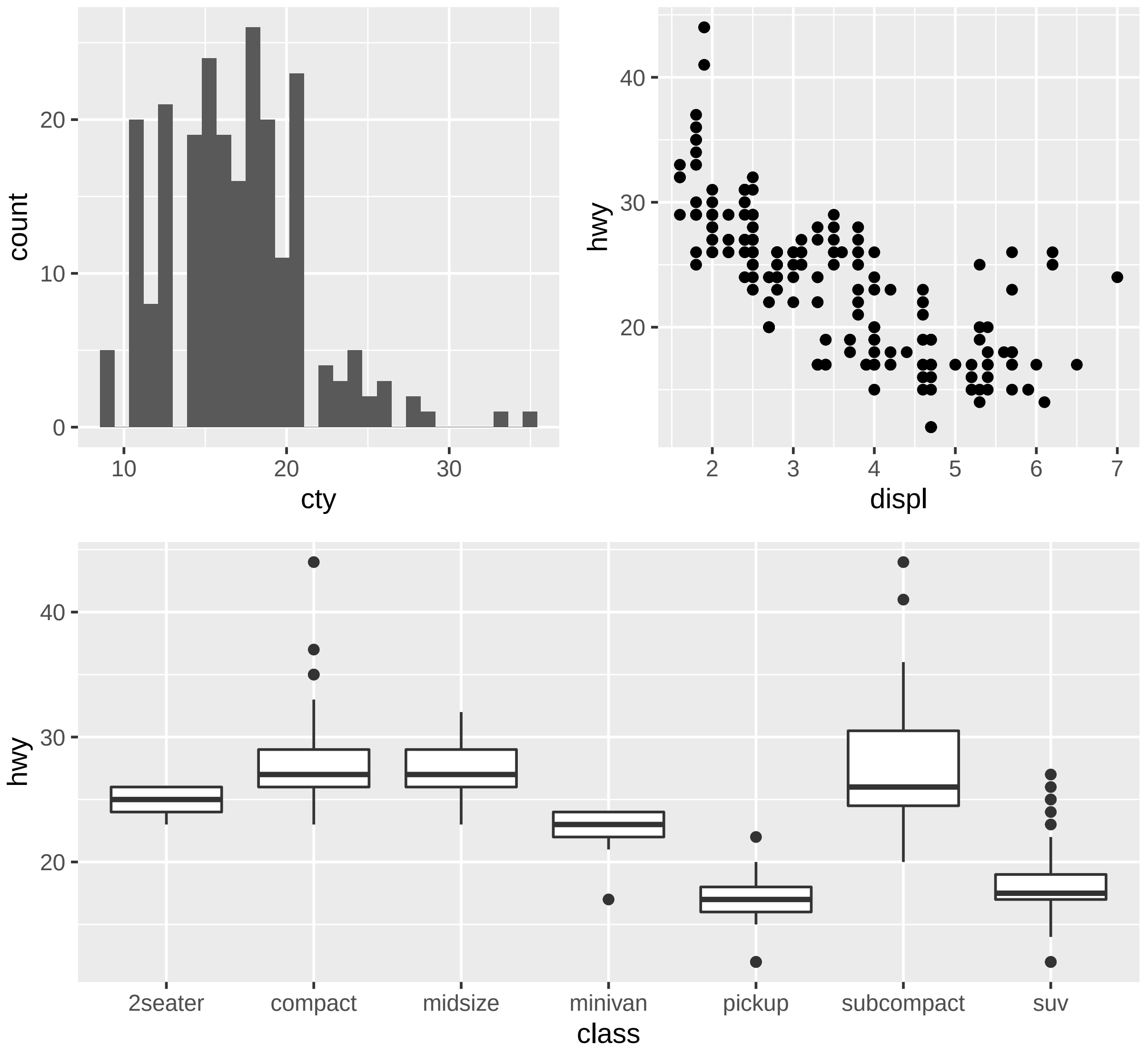
```
ggplot(mpg, aes(x = displ, y = hwy)) +
    geom_point() +
    xlab("Vehicle Displacement (L)") +
    ylab("Highway Miles per Gallon") +
    ggtitle(expression(italic("Cars..."))) +
    theme(axis.text = element_text(size = 15)) +
    theme(axis.title = element_text(size = 18, colour = "orange"))
```



```r
library(viridisLite)

ggplot(iris, aes(x = Species, y = Petal.Length, fill = Species)) +
    geom_boxplot() +
    theme_bw() +
    theme(panel.border = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          axis.line = element_line(colour = "black"),
          legend.position = "None") +
    scale_fill_viridis_d()
```



```r
library(patchwork)

p1 <- ggplot(mpg) + geom_histogram(aes(cty))
p2 <- ggplot(mpg) + geom_point(aes(displ, hwy))
p3 <- ggplot(mpg) + geom_boxplot(aes(class, hwy))

 (p1 + p2) / p3
```


count
0
10
20
10
20
30
cty
hwy
20
30
40
2
3
4
5
6
7
displ
hwy
20
30
40
2seater
compact
midsize
minivan
pickup
subcompact
suv
class