R Workshop Part 3: ggplot2

Rose Driscoll

3/22/2019

ggplot2

Topics

1. The grammar of graphics (the philosophy behind ggplot2)
2. Aesthetics, mapping, geoms, etc (the nuts and bolts of ggplot2)
3. Making your plots pretty and readable

PART 1: The grammar of graphics

`ggplot2` is based upon a layered grammar of graphics. A plot begins with some **data**, and is built in one or more layers. In each **layer**, variables from the data are **mapped** to different **aesthetics** of the plot, such as x, y, color, shape, etc. Each layer produces some sort of a **geometric object**, such as points, a line, bars, etc. In addition, the element(s) in each layer can be **positioned** in a particular way, and sometimes a **statistic** will be applied to create the object in the layer. Multiple layers in the same plot may all use the same data and mapping, or different data and mapping may be used in different layers. The plot also needs to have a **coordinate system**, and may optionally be **faceted** to display an additional categorical variable (we'll talk more about faceting later).

For a much more detailed explanation of the layered grammar of graphics, see Wickham (2010).

PART 2: Aesthetics, mapping, geoms, etc

The good news is that while the grammar of graphics gives you the power to control pretty much every aspect of your plot, you won't have to explicity specify all of those things in every single plot you make. ggplot2 provides handy defaults for almost everything, so the only things you are *required* to specify to make a plot are a dataset, a "geom" function to specify what type of geometric object should be created, and mappings of variables to plot aesthetics. We'll start with just these three things and work our way up.

For this workshop, we'll use the `iris` built-in dataset:

```
data(iris)
head(iris)
```

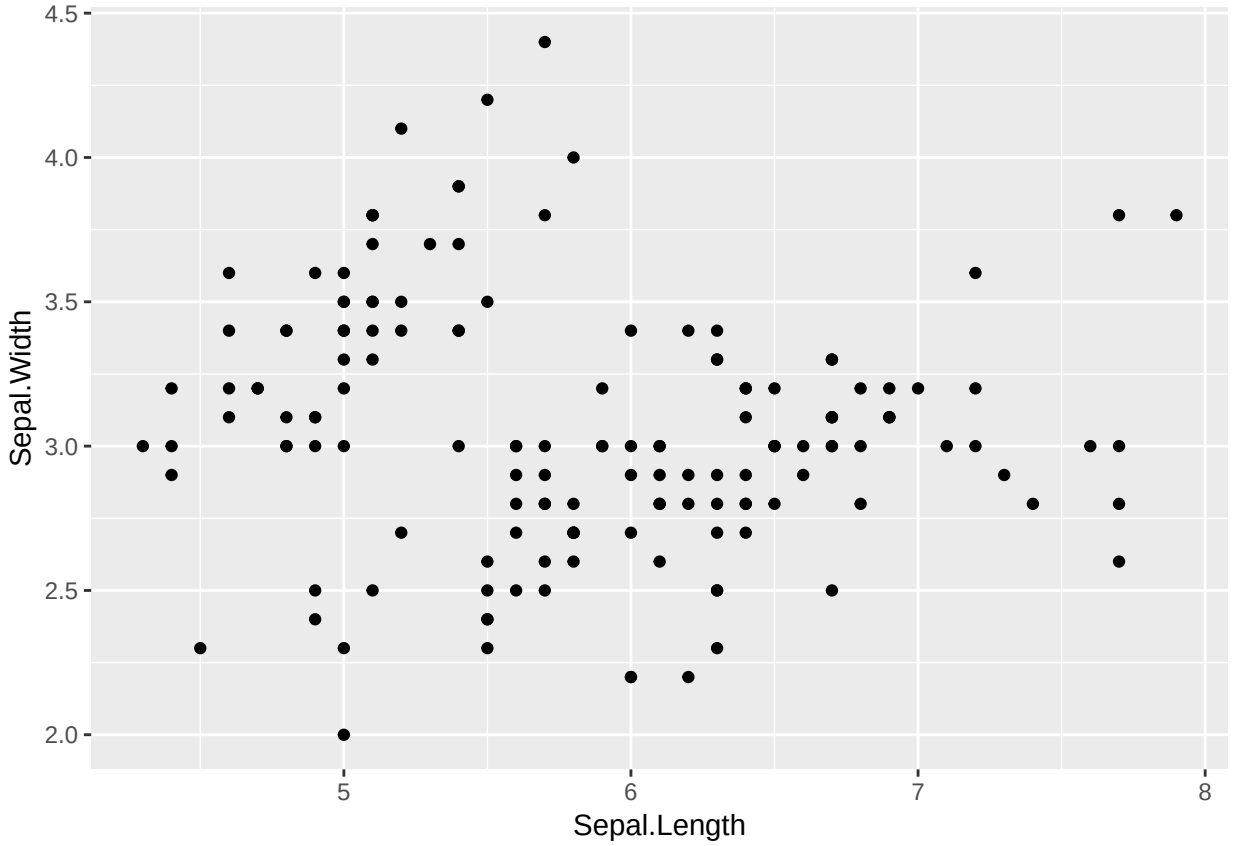
```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5  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

Here's a simple example of ggplot2 syntax. This chunk has the option `eval = FALSE` set since we're not actually supplying data, a geom, or mapping yet - just marking out where those things will go.

```
ggplot(data = YOUR_DATA_HERE) + YOUR_GEOM_HERE(mapping = aes(YOUR_MAPPINGS_HERE))
```

Here's a first pass at making a scatterplot of sepal length versus sepal width using the `iris` data.

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



The call to ggplot spans multiple lines, with the lines connected by a plus sign at the end of all but the last line. For now, the only argument we supply to the `ggplot()` function is the data frame. The second line is our geom function - for now we are using `geom_point()` to make a scatterplot. We'll come back to different geom functions later.

1. Aesthetics and mapping

Let's start with aesthetics and mapping. In our sepal length vs. sepal width plot, we are mapping variables to two aesthetics: x and y.

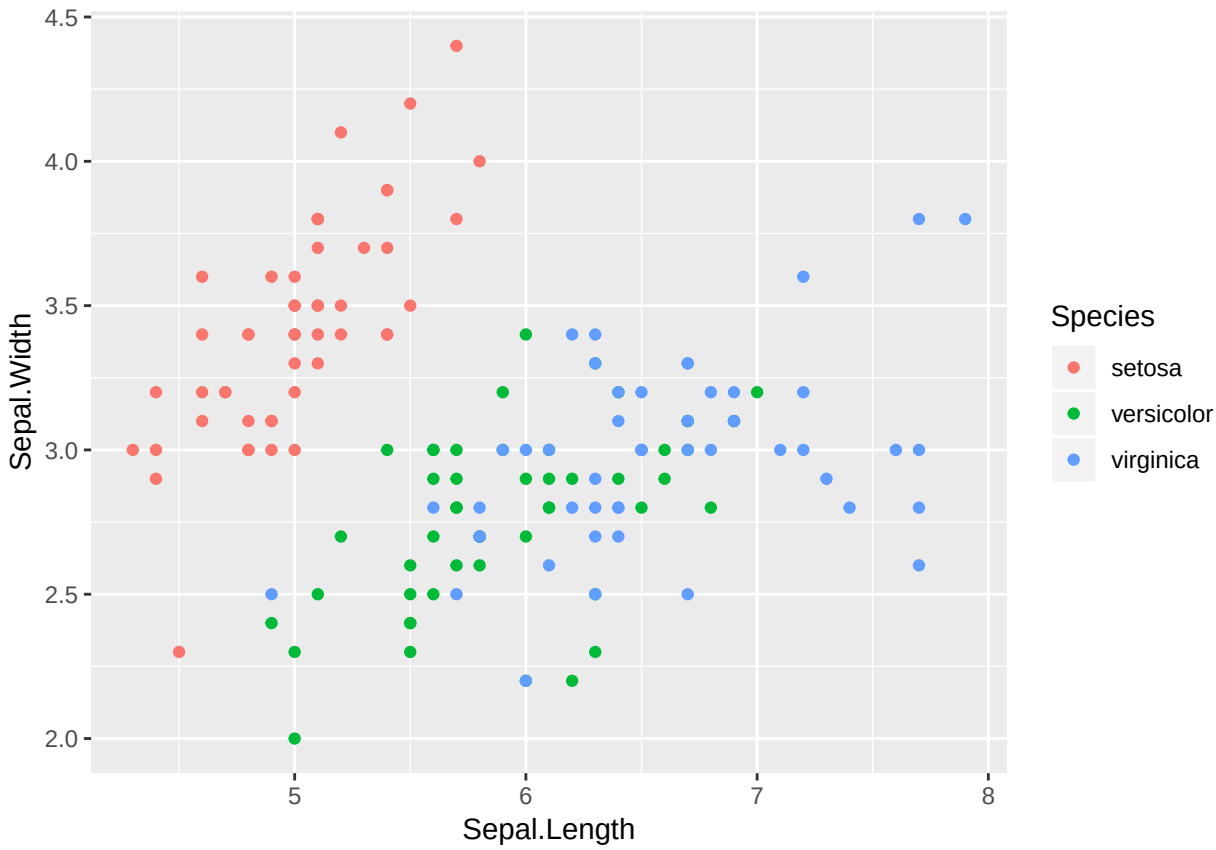
```
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width))
```

Mappings are supplied to the `mapping` argument, and need to be wrapped in the `aes()` function so that `ggplot` knows that they will become aesthetics of the plot. Inside the `aes()` function, we specify the aesthetic we want to map to followed by the name of the variable we want to map: `x = Sepal.Length`.

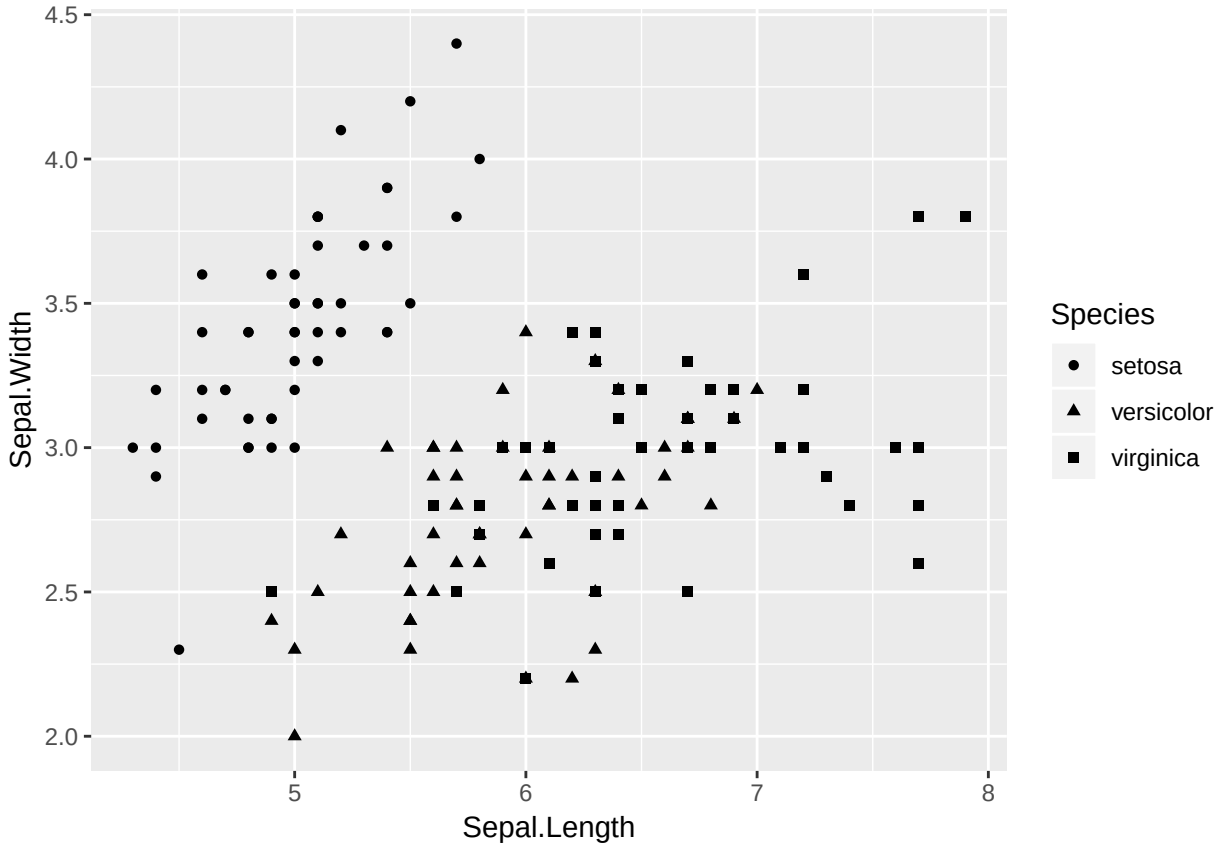
In our first example, we just used the `x` and `y` aesthetics, but there are lots of other aesthetics you can use, including `color`, `shape`, `size`, `alpha`, etc. The exact set of aesthetics available will depend on what geom you are using.

A few examples:

```
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species))
```



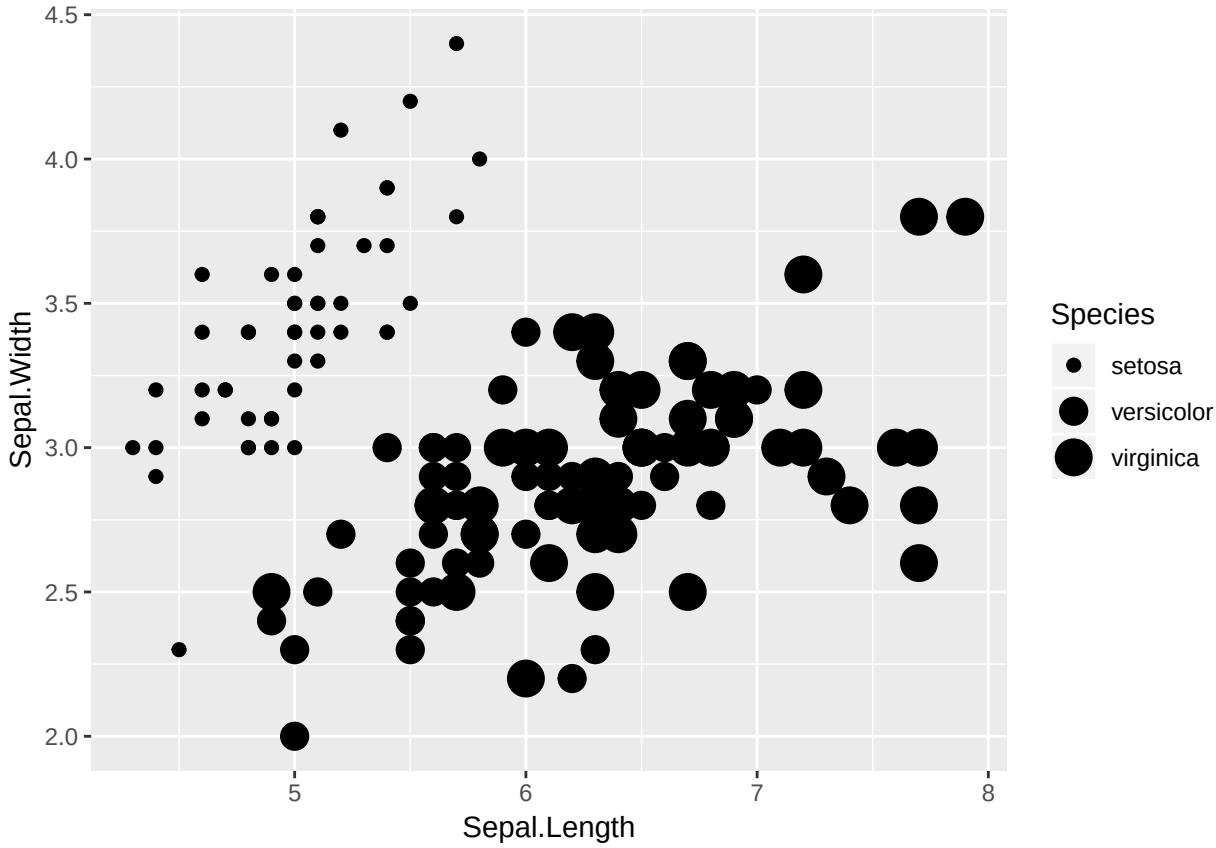

```
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, shape = Species))
```



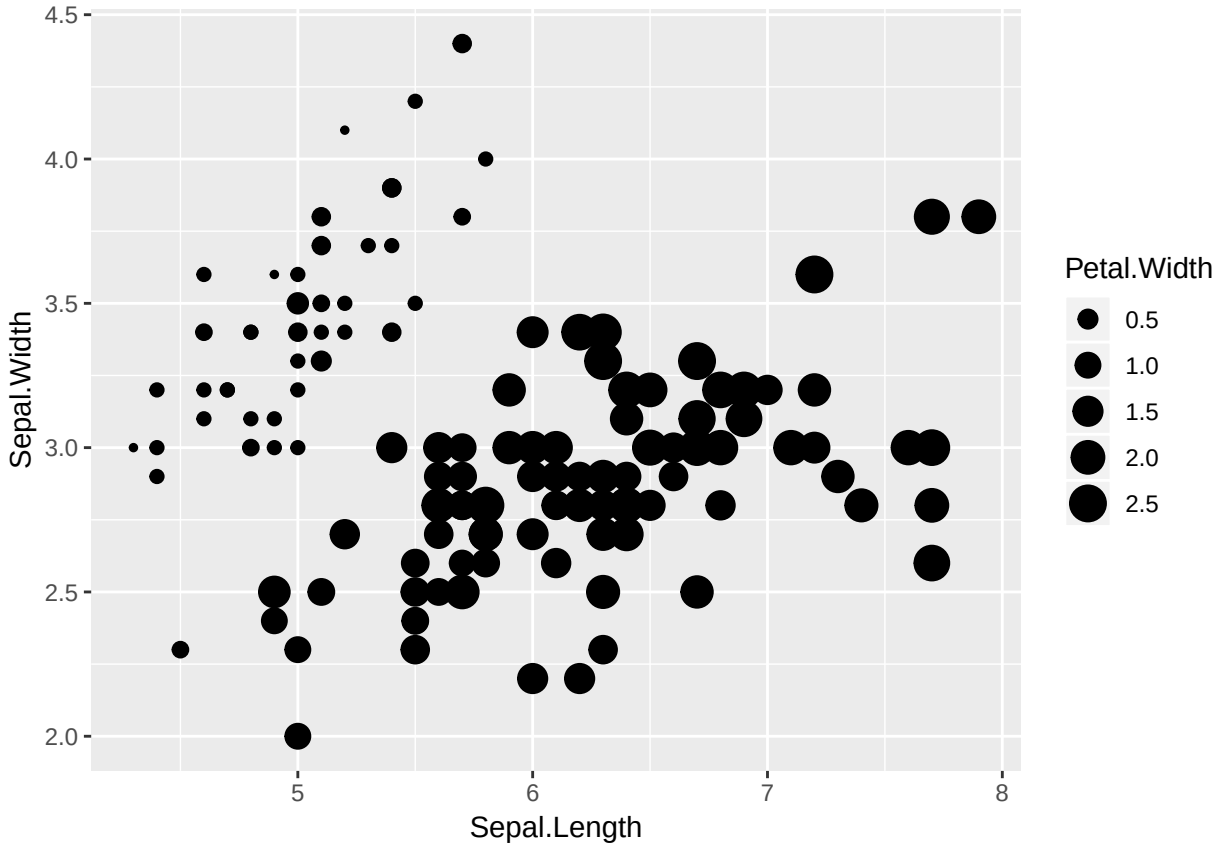
Some aesthetics are ordered, while others are unordered. It's best to use ordered aesthetics (for example `size` or `alpha`) for continuous variables and unordered aesthetics (for example `shape`) for categorical variables. `color` can be ordered or not ordered depending on what color scheme you are using (the same goes for `fill`, which we will talk about later.)

```
# This is a not-ideal use of size:
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, size = Species))
```

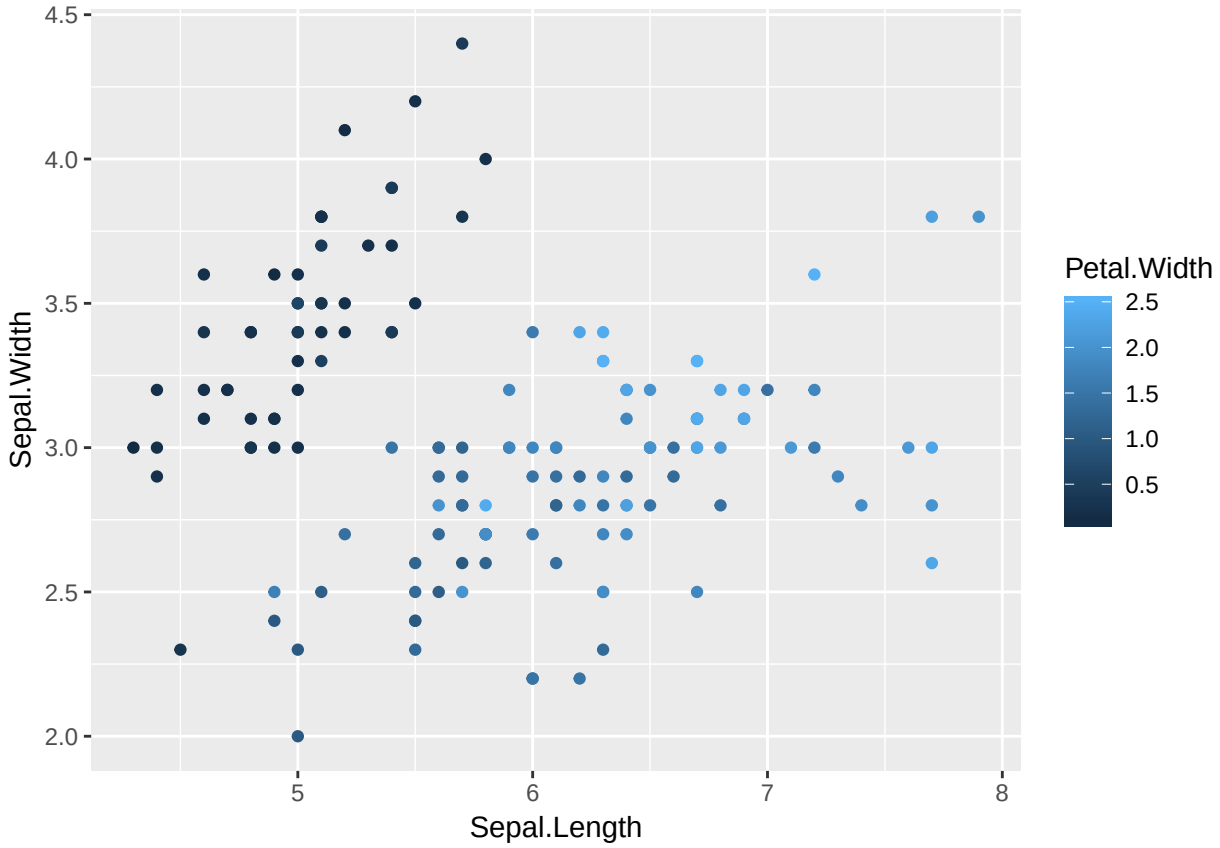
```
## Warning: Using size for a discrete variable is not advised.
```




```
# ggplot2 helpfully warns you that 'Using size for a discrete
# variable is not advised.'

# Here is a better use of size:
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, size = Petal.Width))
```

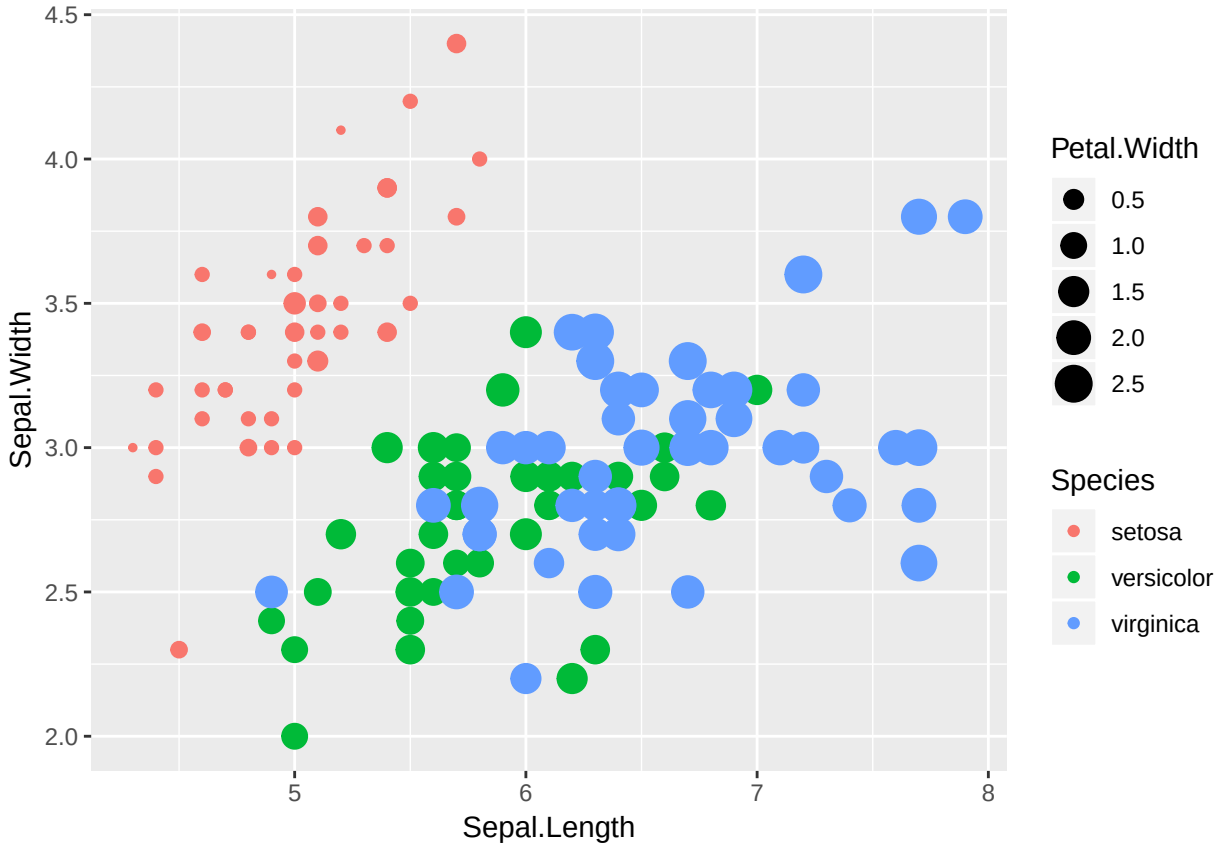



```
# Here color is used for a continuous variable
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Petal.Width))
```

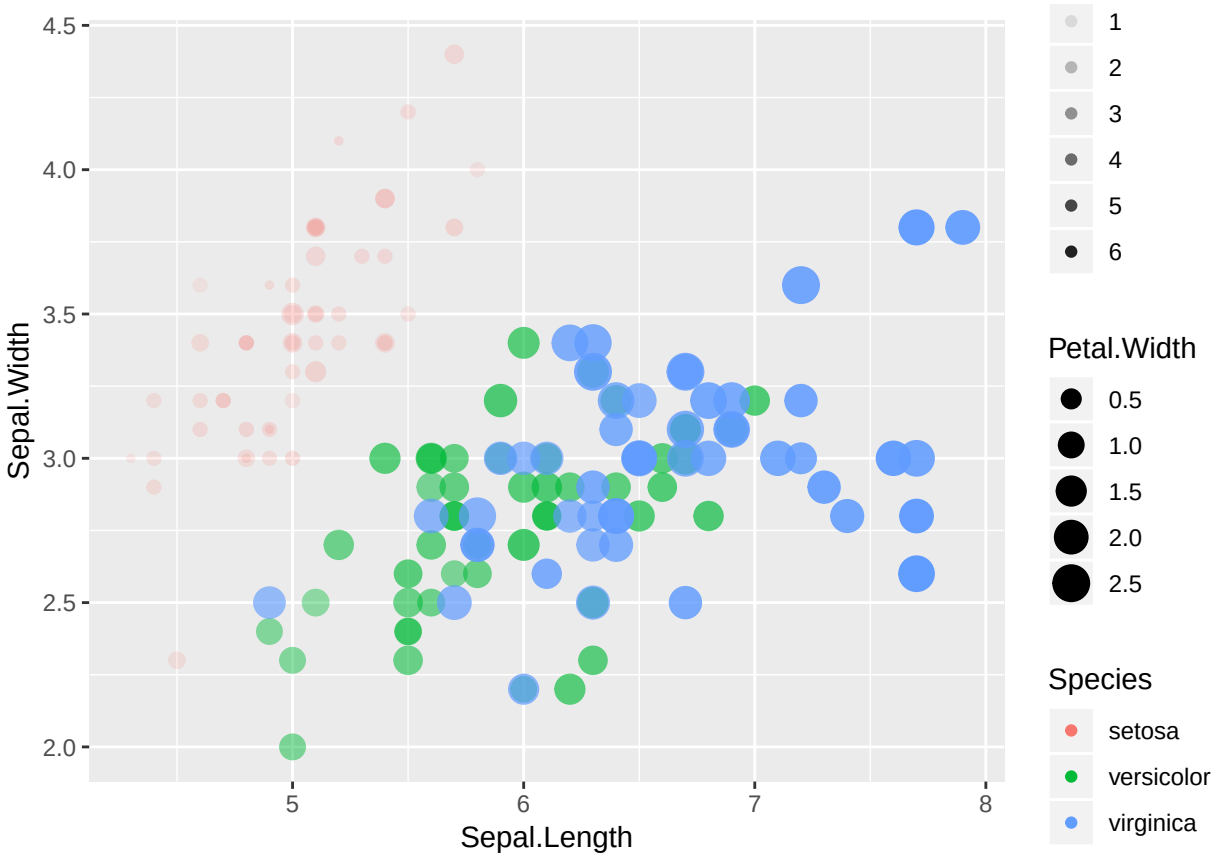


You can specify as many aesthetics as you like, although more than three or four tends to get very confusing...

```
# This is maybe okay:
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species, size = Petal.Width))
```



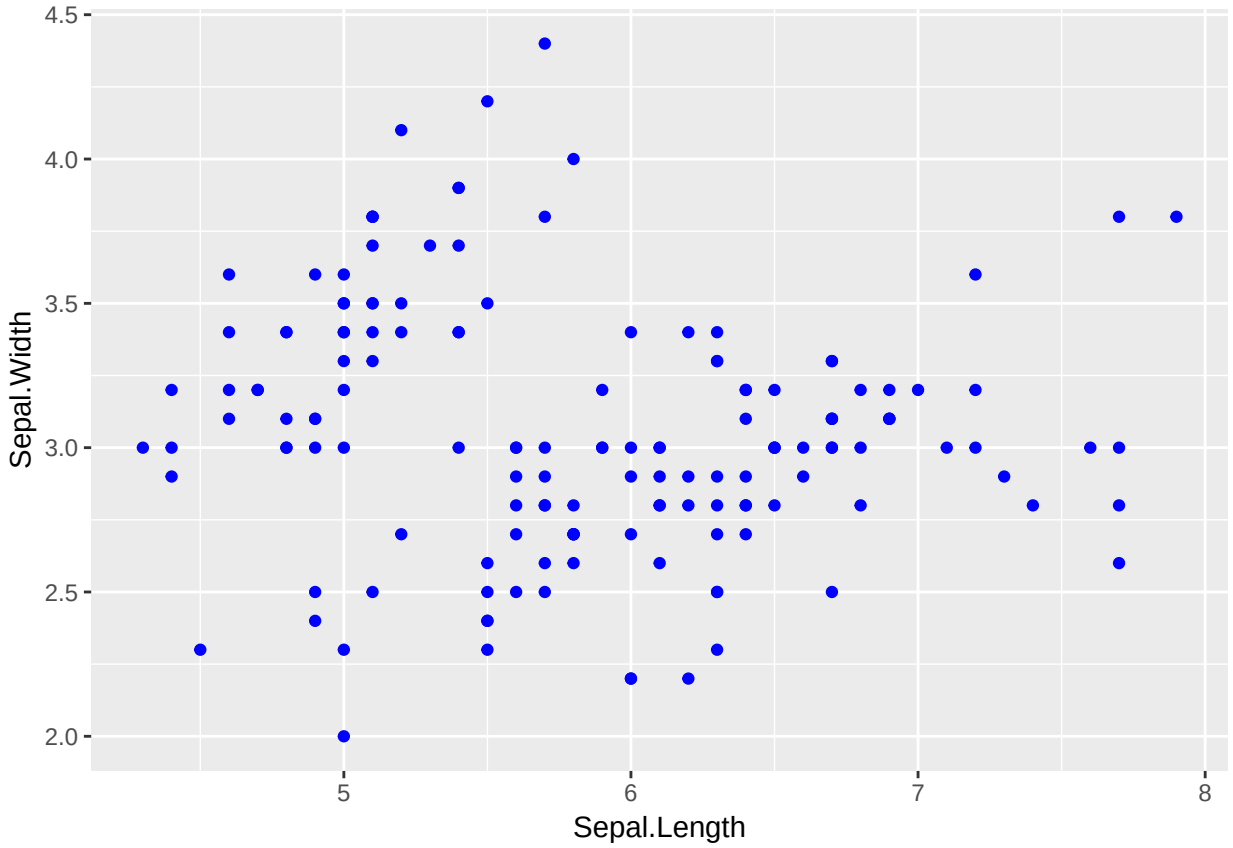

```
# This is probably too much for the person looking at your
# plot to take in all at once.
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species, size = Petal.Width, alpha = Petal.Length))
```



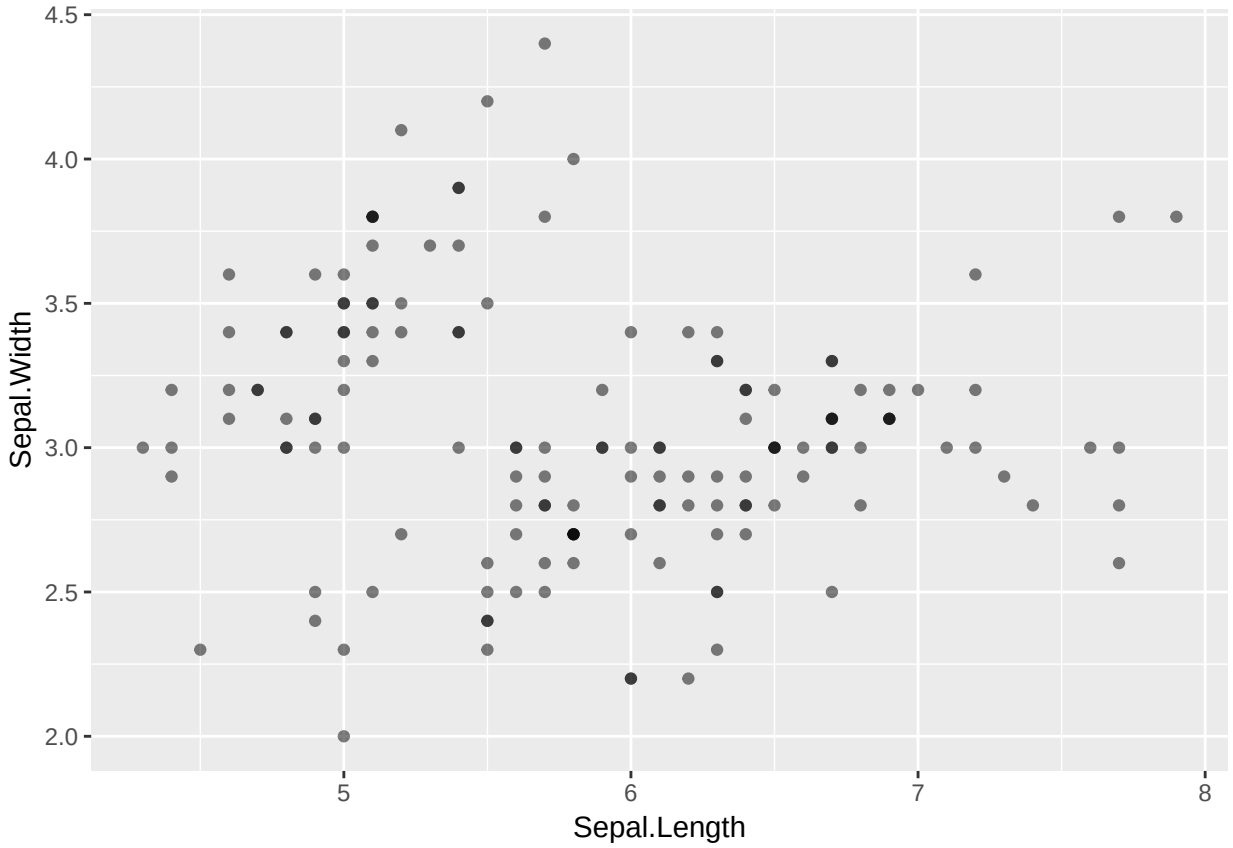
Part of making good plots is figuring out how much information you can present in a single plot while still keeping things readable and easy to understand.

Getting back to the nuts and bolts of aesthetics and mapping, you can also manually set an aesthetic without mapping it to a variable, for instance if you wanted to make all of the points on your plot blue. To manually set an aesthetic, provide the name of the aesthetic as a separate argument (*not* part of the `mapping = aes()` stuff) and then the value you want that aesthetic to take.

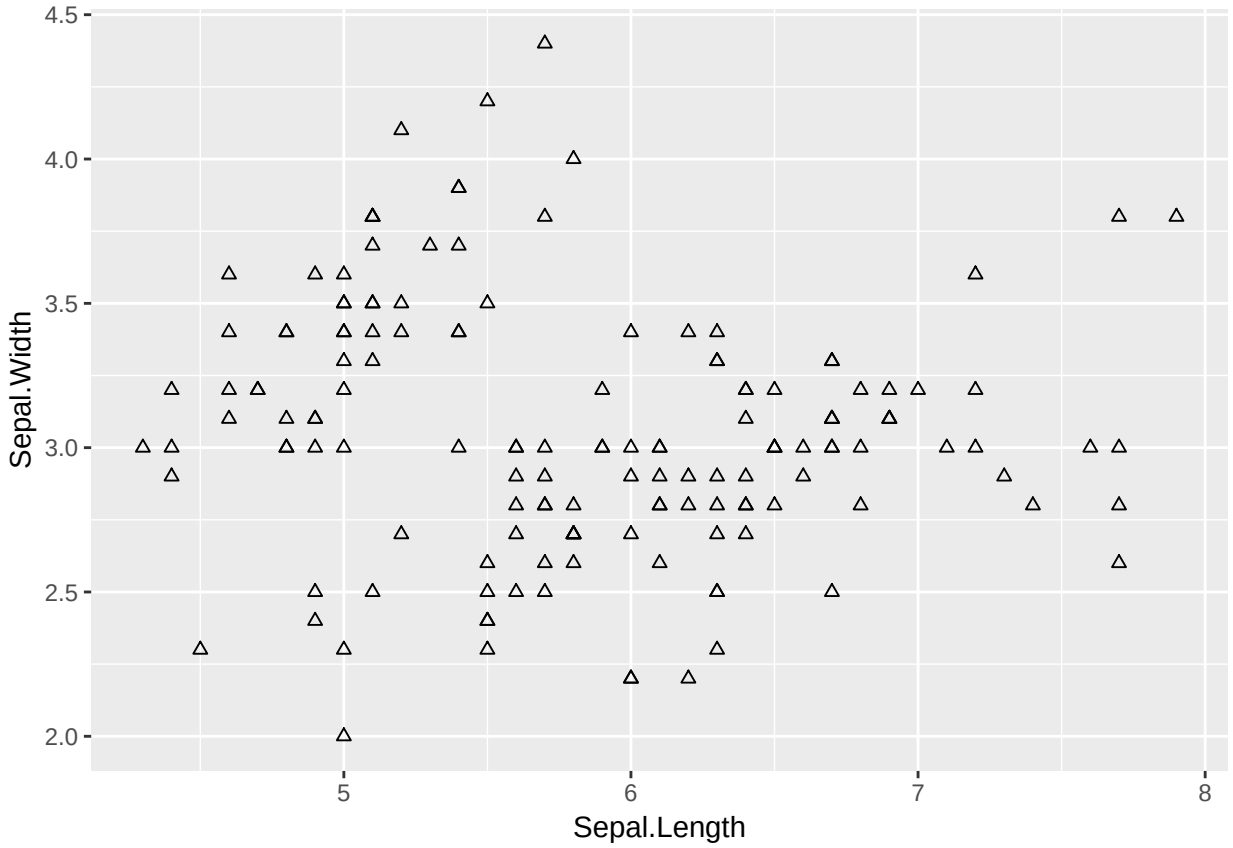
```
# set the color of the points
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), color = "blue")
```




```
# set the transparency of the points
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), alpha = 0.5)
```

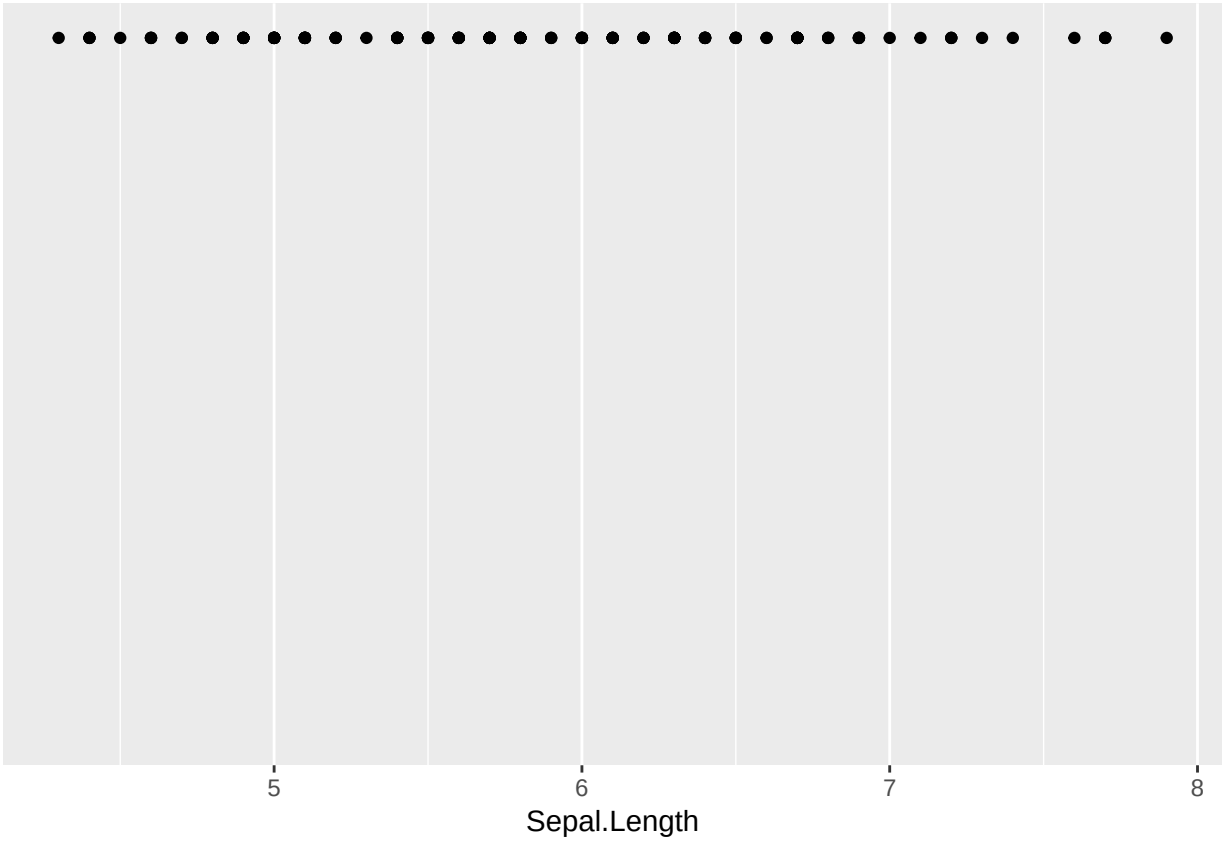



```
# some of the points in this plot look different because in
# some places multiple semi-transparent points are stacked up
# on top of each other.

# set the shape of the points
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), shape = 2)
```




```
# set the size of the points
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), size = 3)
```


```
# you can even manually set x and/or y, though this isn't
# very helpful in a scatterplot
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length),
    y = 1)
```



These examples just demonstrate how to use the `mapping` argument and `aes()` function, and how to set aesthetics manually. The particular set of aesthetics available to you depends on what geom function you are using. We'll talk about geoms next.

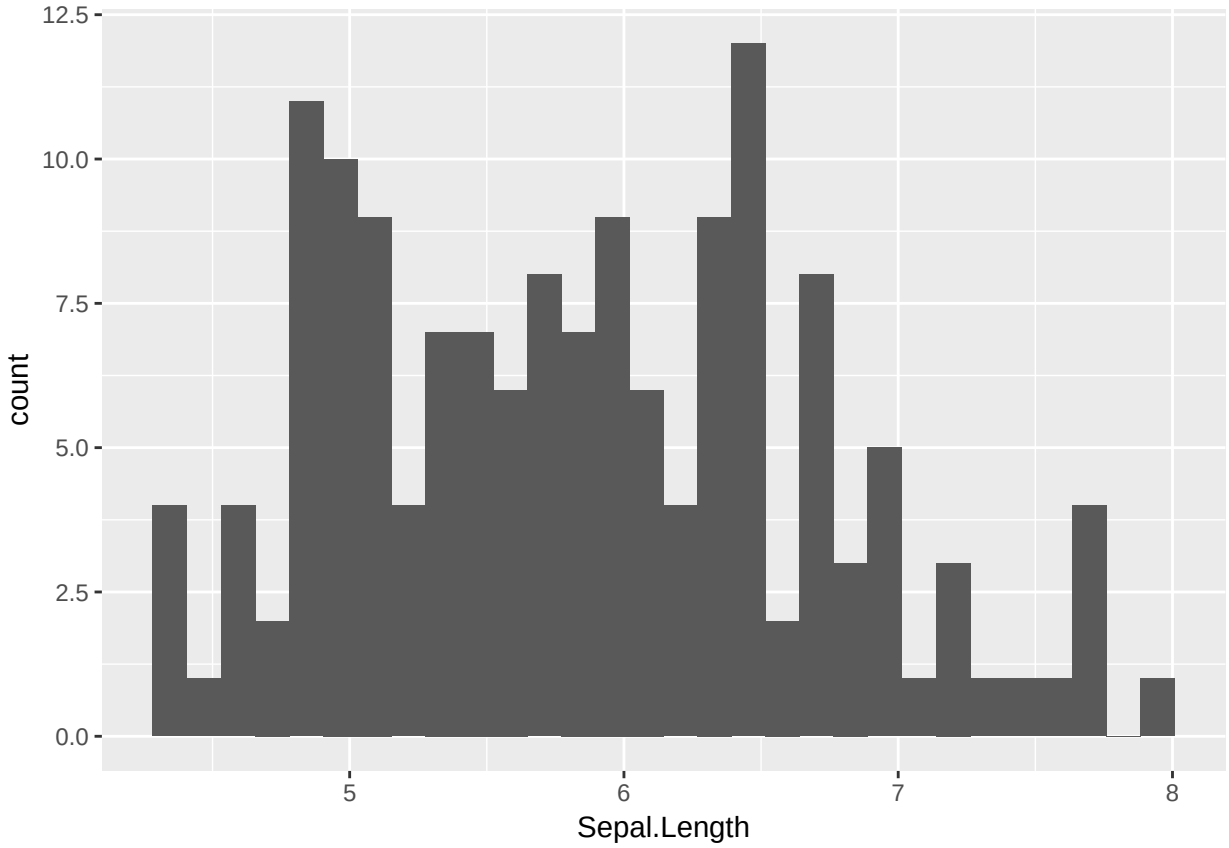
2. Geoms

We've already used one geom function, `geom_point()`, to make a scatterplot. Other geoms can be used to make lots of other kinds of plots; here are a few examples. Take a look at the `ggplot2` cheatsheet for even more geoms and to see what aesthetics are available for each geom.

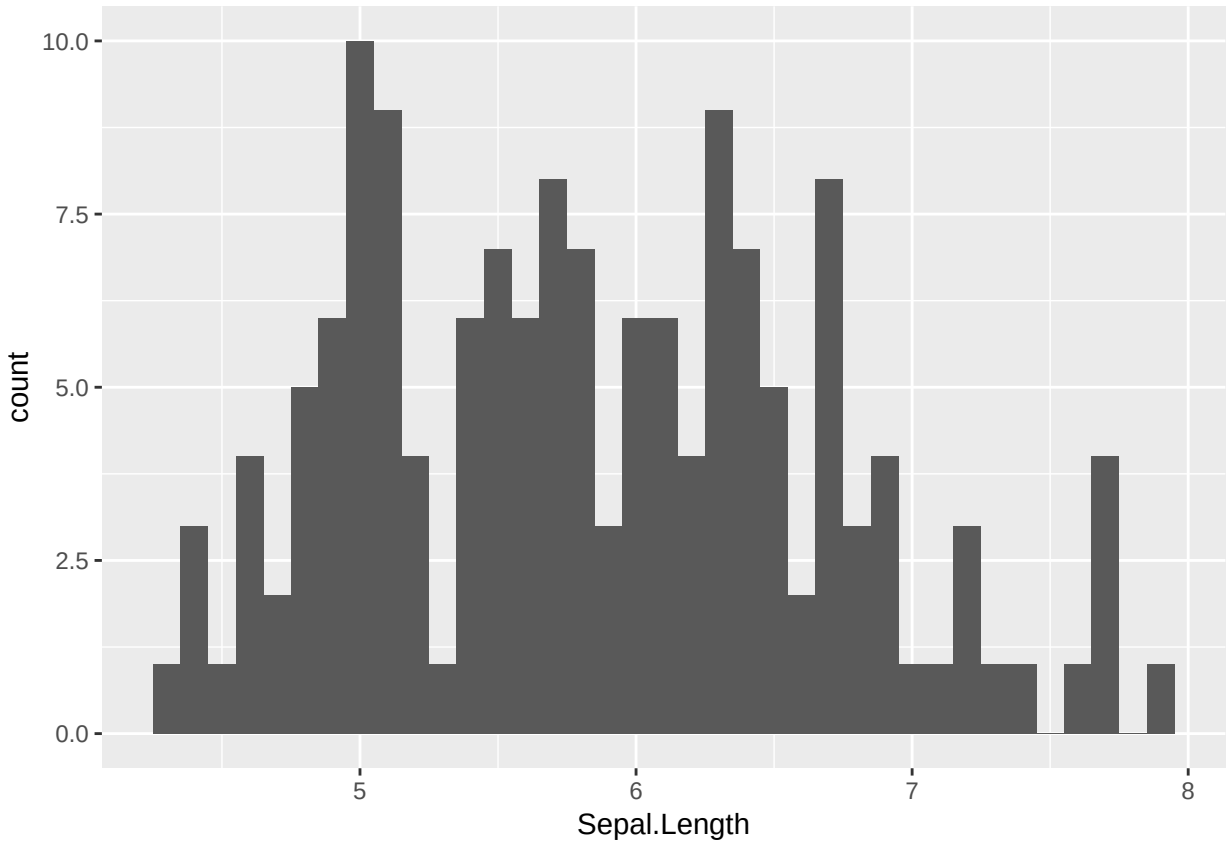
Histograms

```
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length))
```

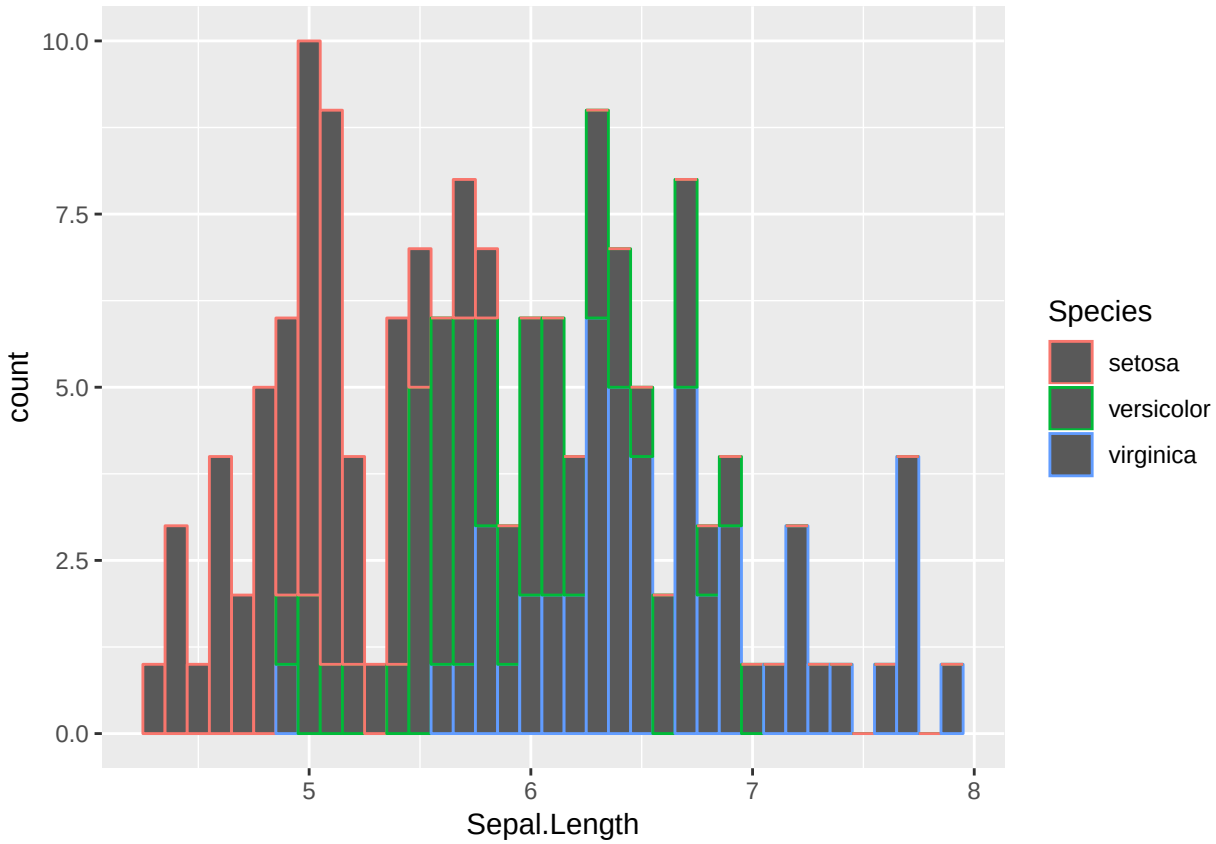
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



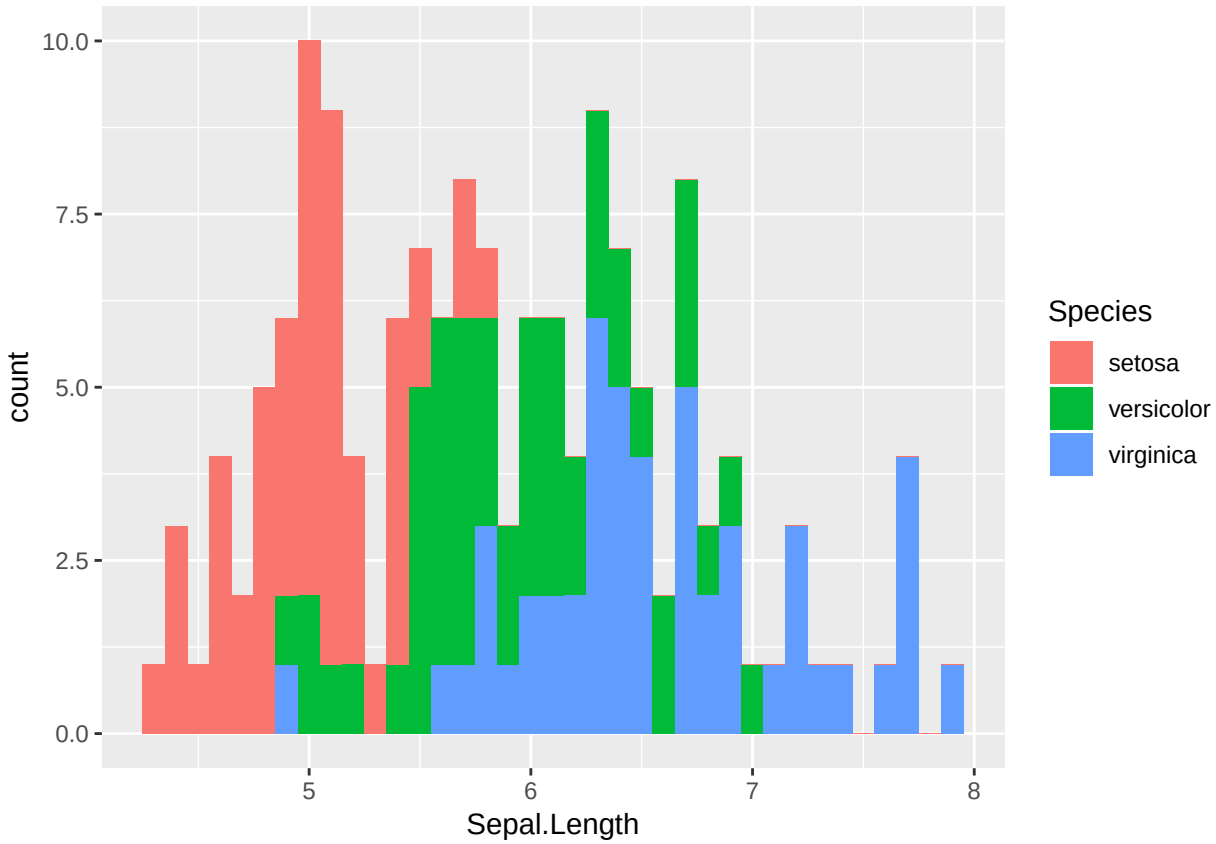

```
# `geom_histogram()` defaults to making 30 bins of equal
# size, but you can control this with the `binwidth` argument

ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length),
    binwidth = 0.1)
```




```
# `color` now controls the border around each bar
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    color = Species), binwidth = 0.1)
```



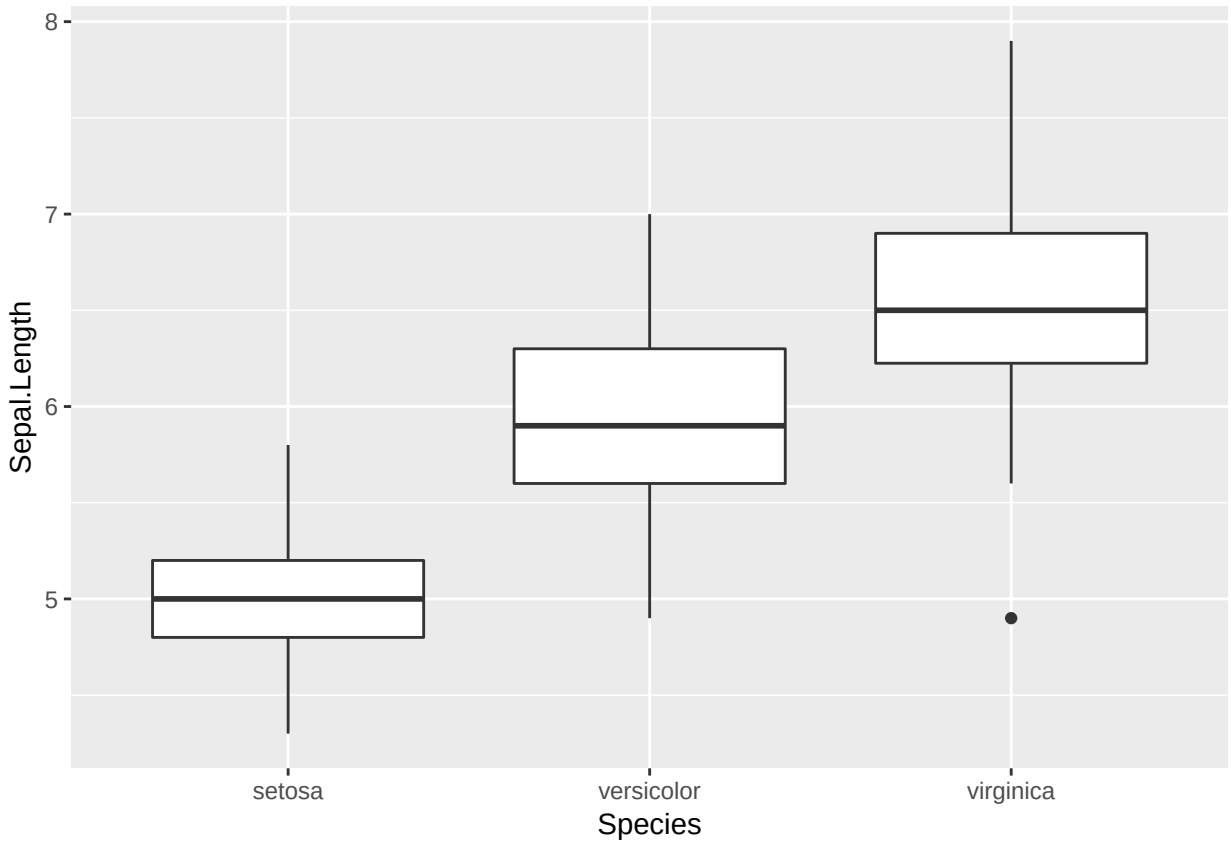

```
# `fill` controls the shading of the center of the bar
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1)
```



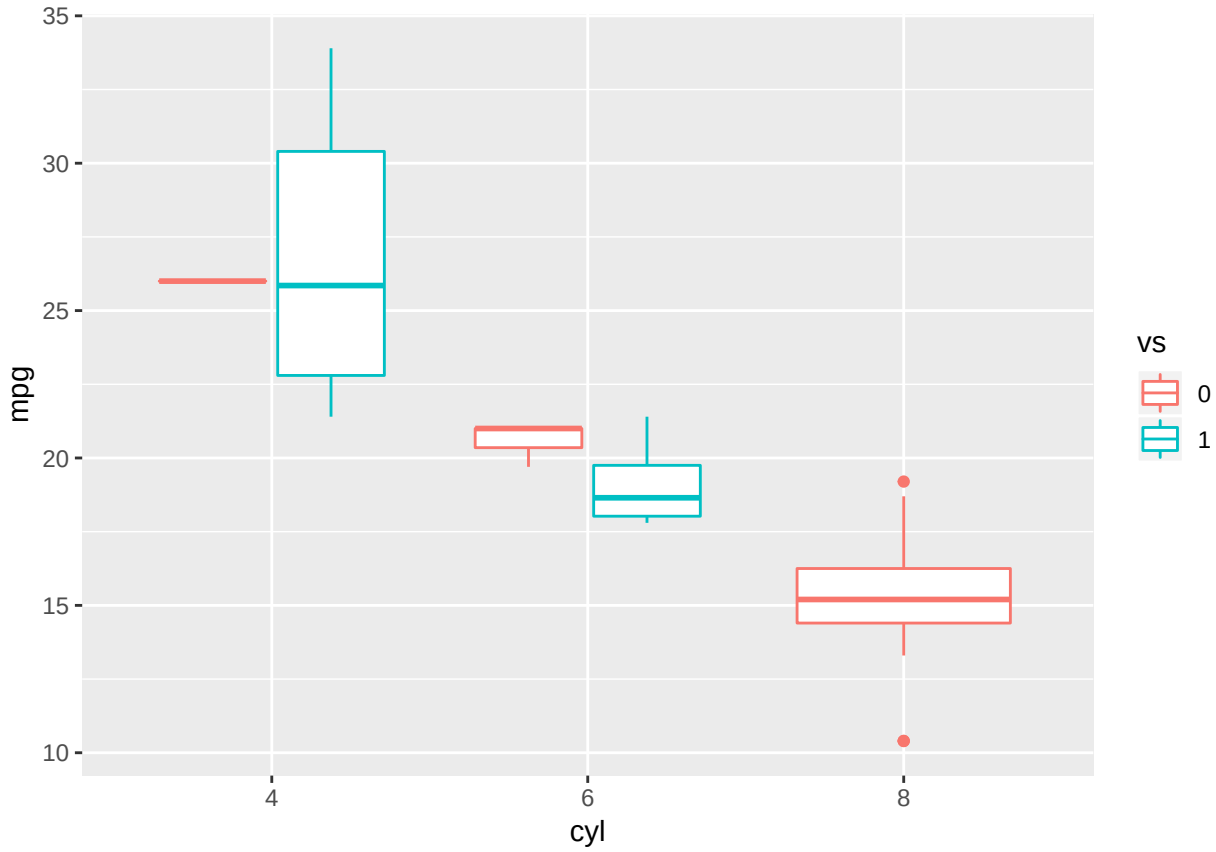

```
# both `color` and `fill` will result in stacked bars
```

Boxplots

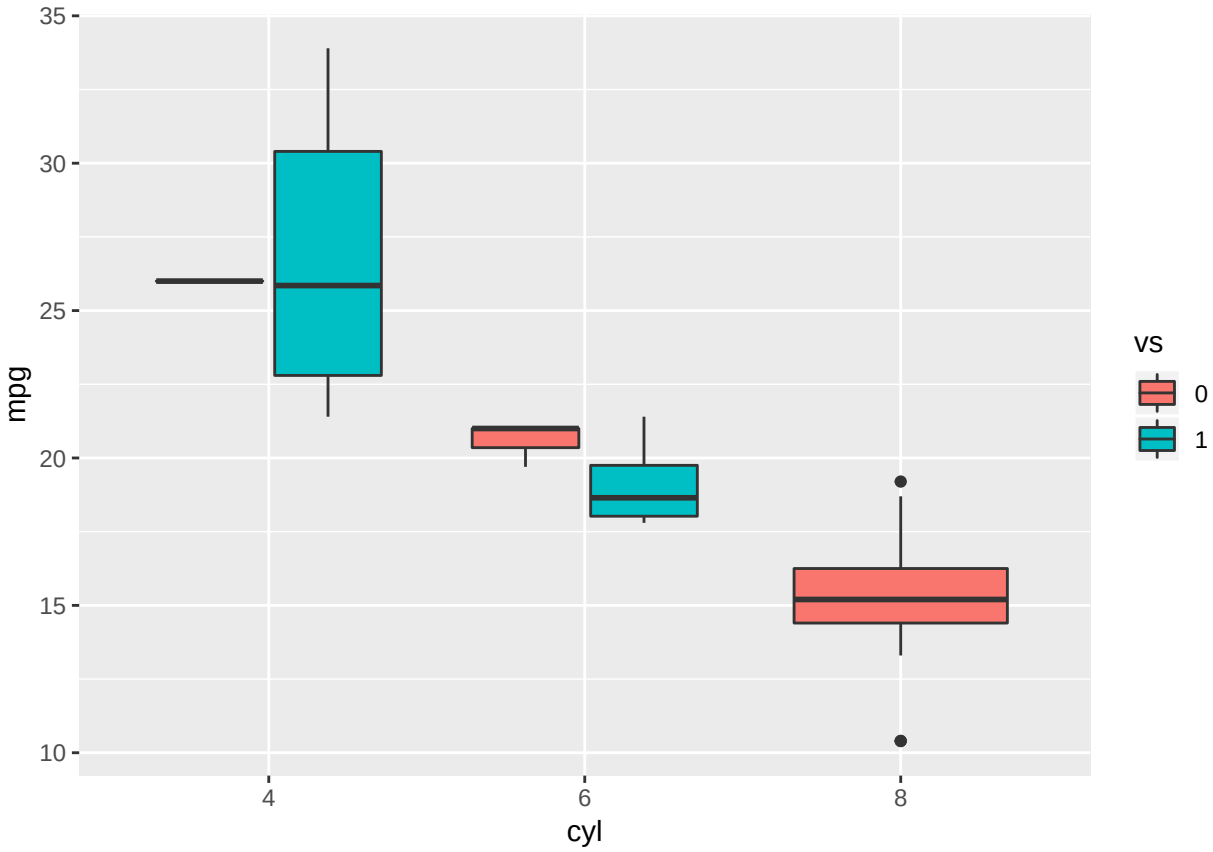
```
ggplot(data = iris) + geom_boxplot(mapping = aes(x = Species,
    y = Sepal.Length))
```




```
# if you have multiple categorical variables, `color` and
# `fill` will result in multiple side-by-side boxplots for
# each category
data(mtcars)
mtcars$cyl <- as.factor(mtcars$cyl)
mtcars$vs <- as.factor(mtcars$vs)
ggplot(data = mtcars) + geom_boxplot(mapping = aes(x = cyl, y = mpg,
    color = vs))
```

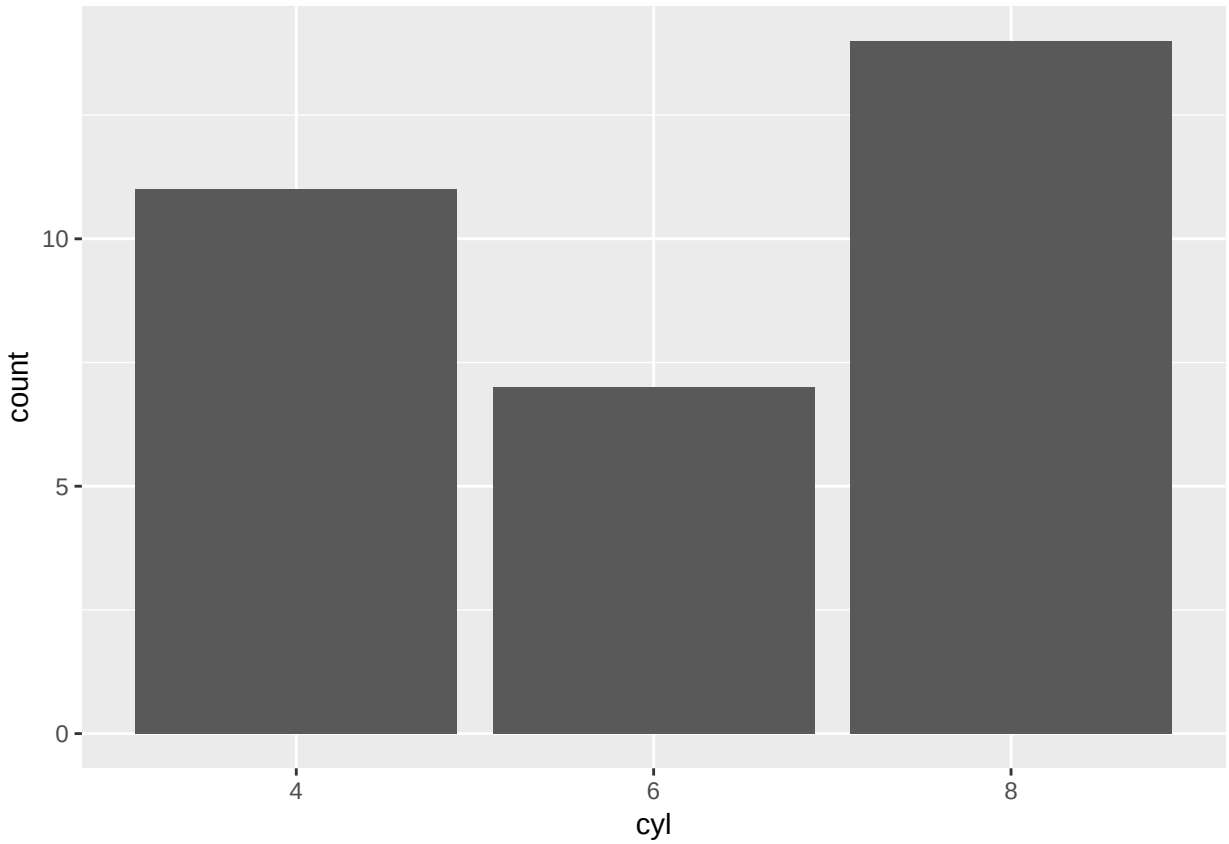



```
ggplot(data = mtcars) + geom_boxplot(mapping = aes(x = cyl, y = mpg,
    fill = vs))
```

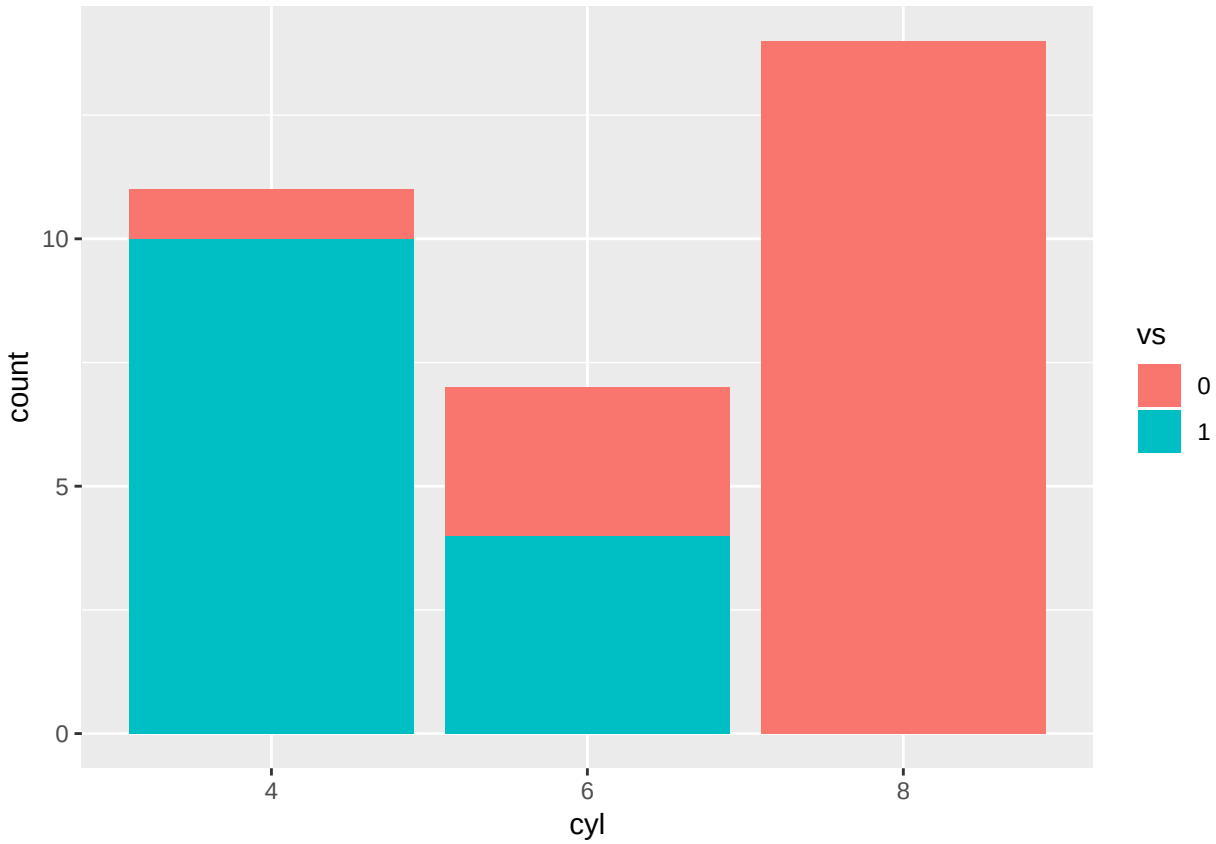


Bar plots

```
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl))
```




```
# if you have multiple categorical variables, `color` and
# `fill` will result in stacked bars
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl, fill = vs))
```

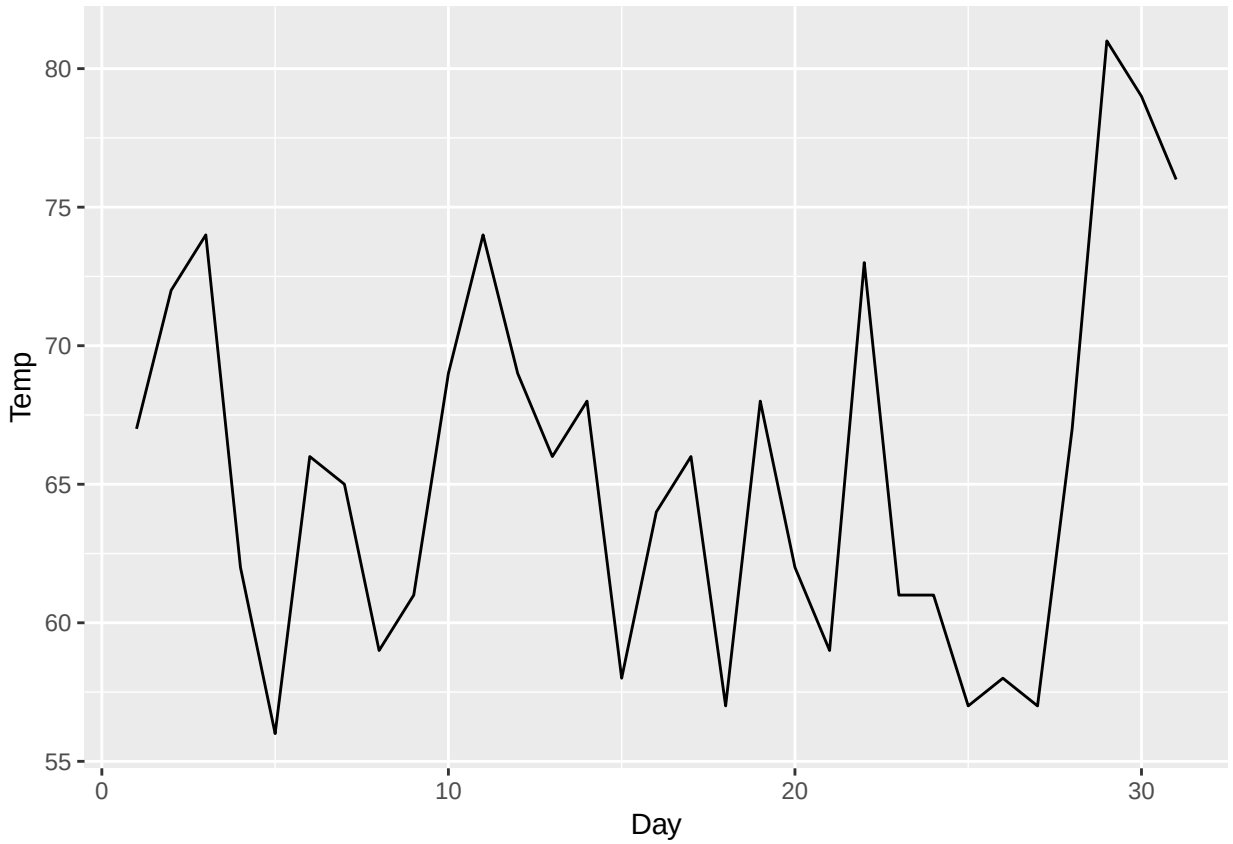


Lines

`ggplot2` can draw lines in lots of different ways. We'll look at 2 main ways: connecting points and smoothing.

Connecting a series of observations with `geom_path()`:

```
data(airquality)
filter(airquality, Month == 5) %>% ggplot() + geom_path(mapping = aes(x = Day,
    y = Temp))
```

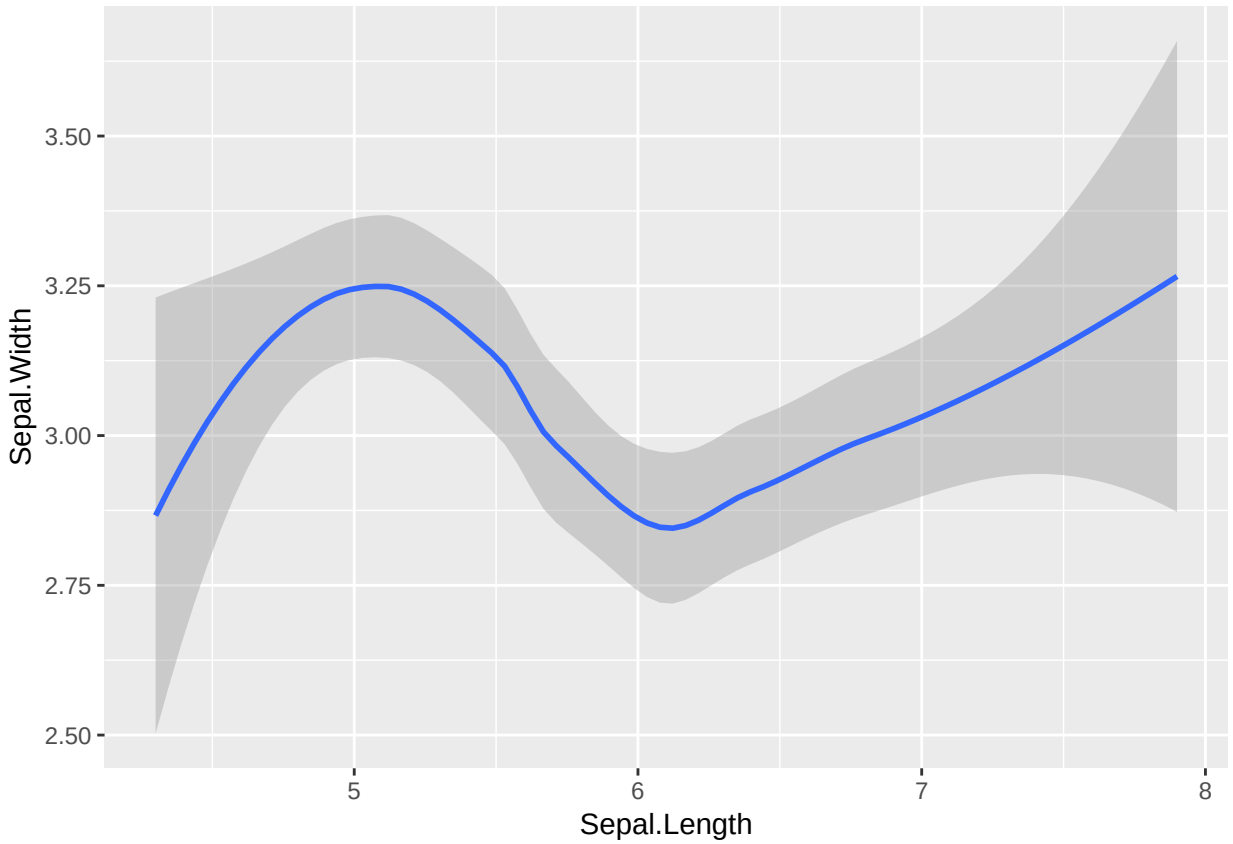



```
# By the way, yes, you can pipe data from another function
# into ggplot.  After all, it's a tidyverse function - the
# first argument is data!
```

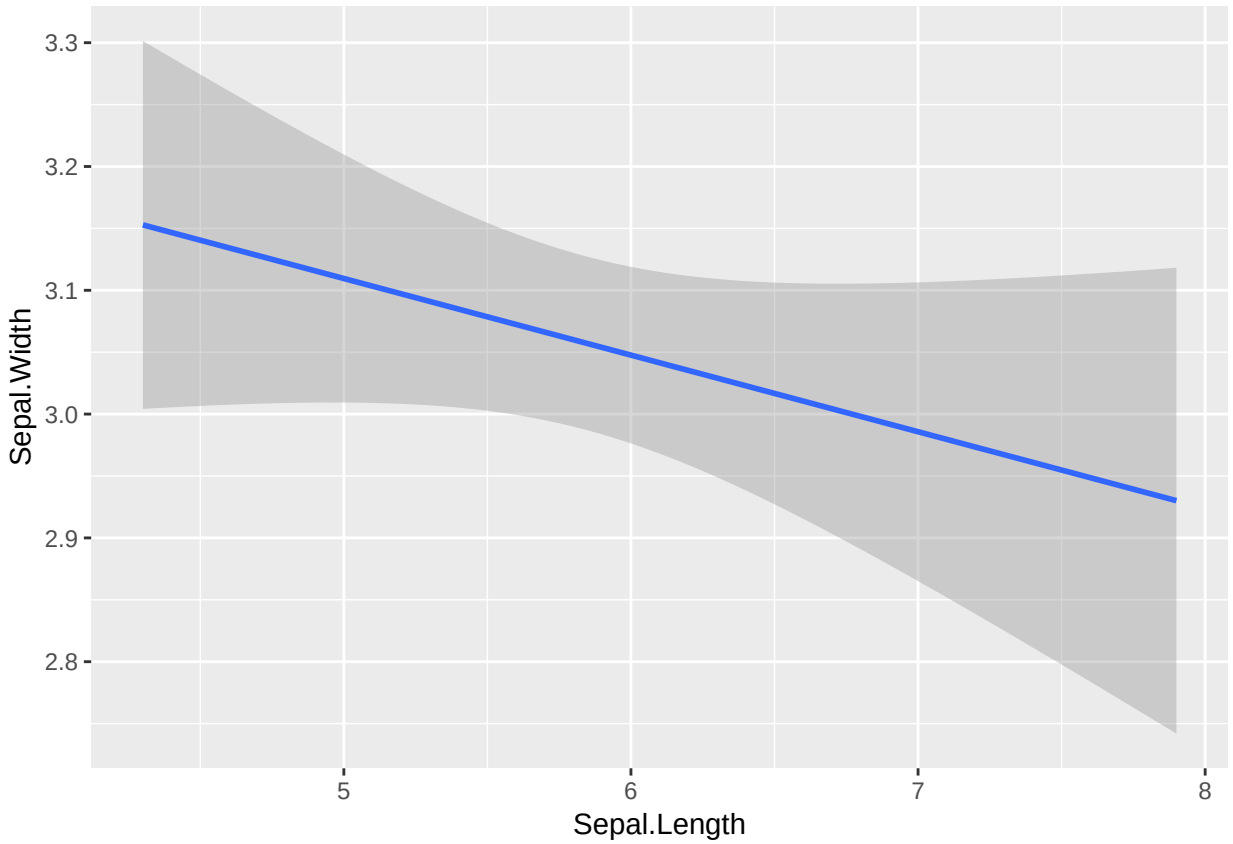
Fitting a curve through a cloud of observations with `geom_smooth()`:

```
ggplot(data = iris) + geom_smooth(mapping = aes(x = Sepal.Length,
    y = Sepal.Width))
```

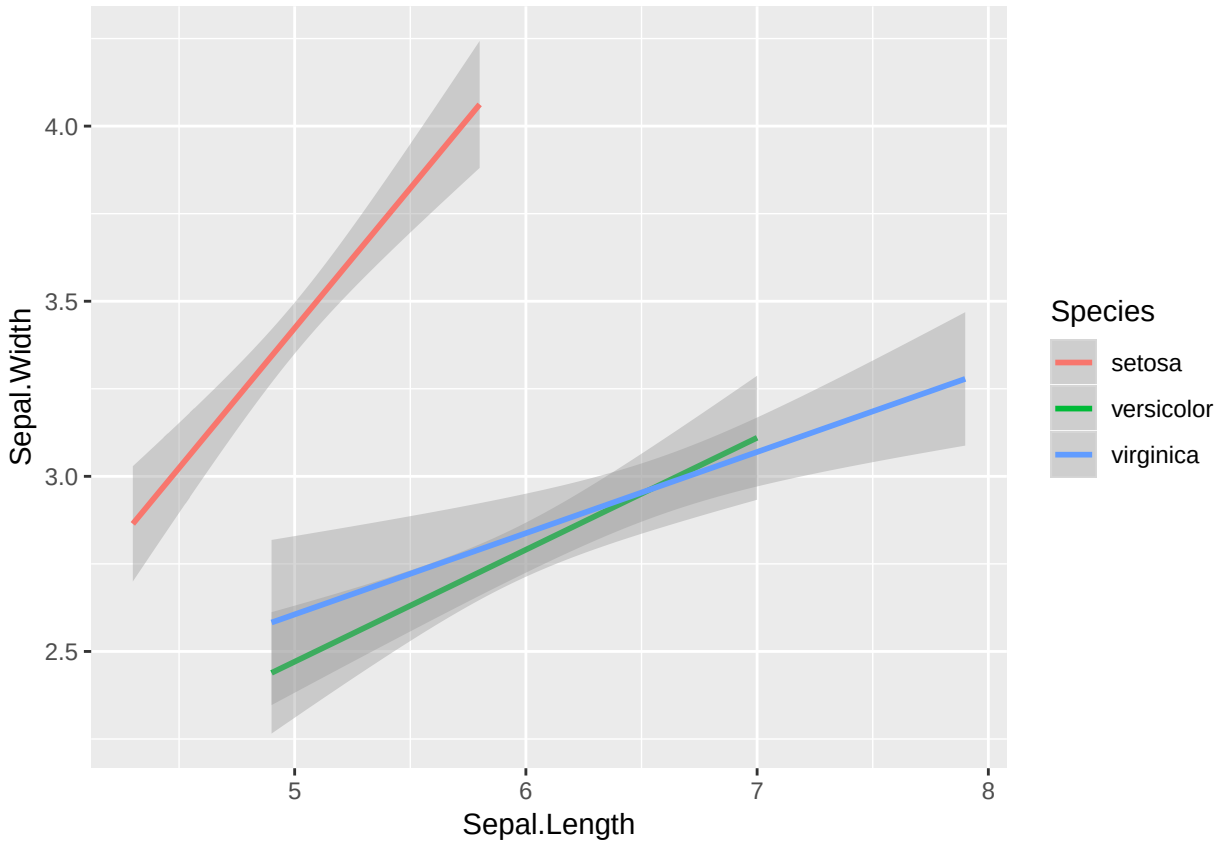
```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```




```
# geom_smooth has many different smoothing methods, 'loess'
# is the default but often it will be more useful to find a
# line of best fit with method = 'lm'
ggplot(data = iris) + geom_smooth(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), method = "lm")
```



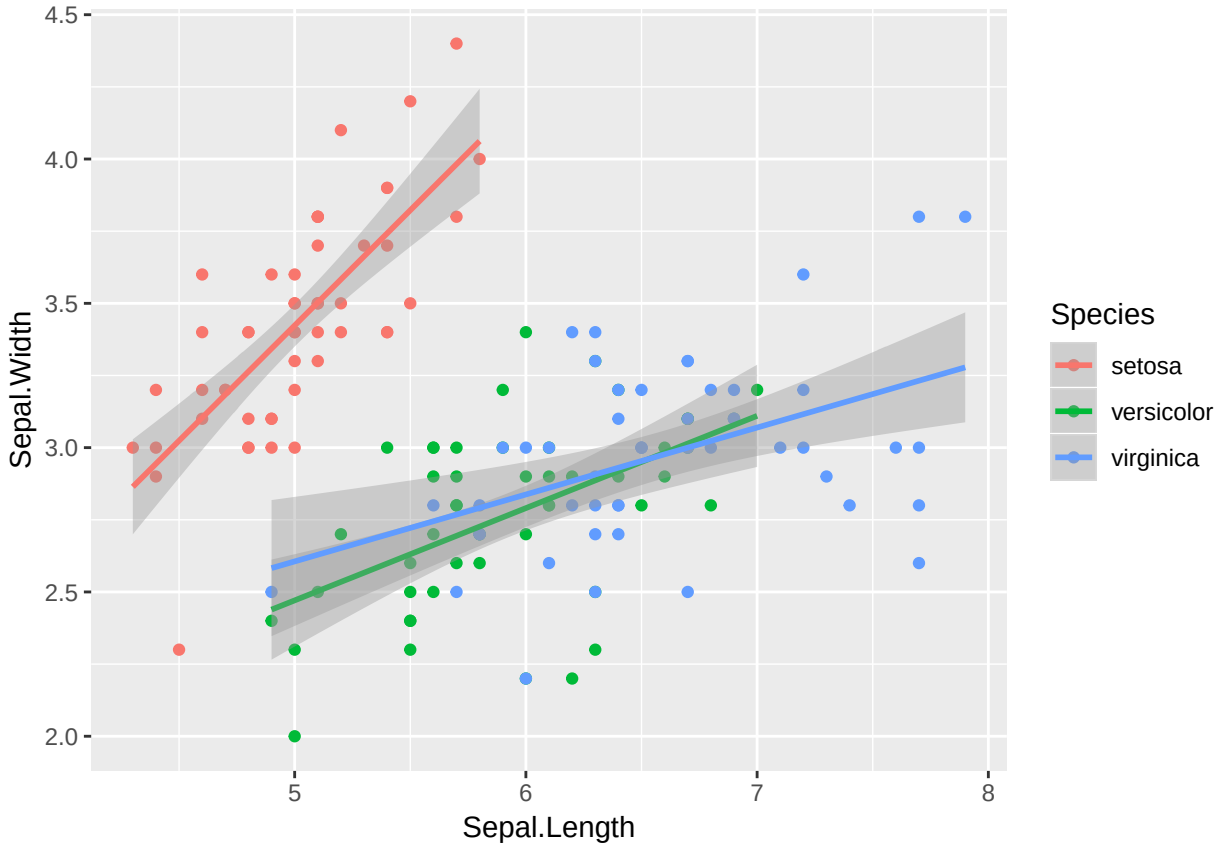

```
# you can also map the `color` aesthetic to a categorical
# variable in order to fit a separate curve to each category
ggplot(data = iris) + geom_smooth(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species), method = "lm")
```



Combining multiple geoms

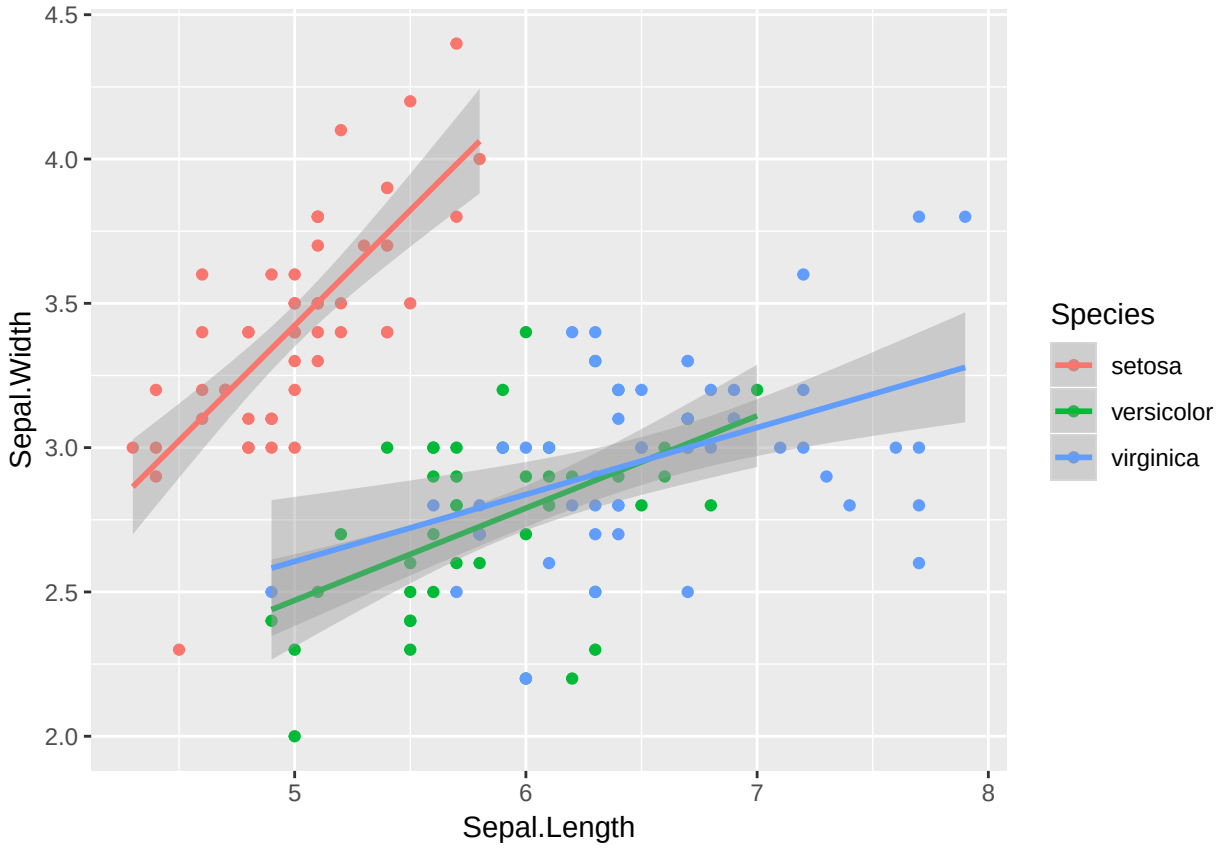
The great thing about the layered grammar of graphics is that you can use multiple geoms to create different layers in a single plot. The linear models we made above aren't very meaningful unless we can also see the points, so let's use `geom_point()` and `geom_smooth()` together to make a better plot:

```
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + geom_smooth(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species), method = "lm")
```



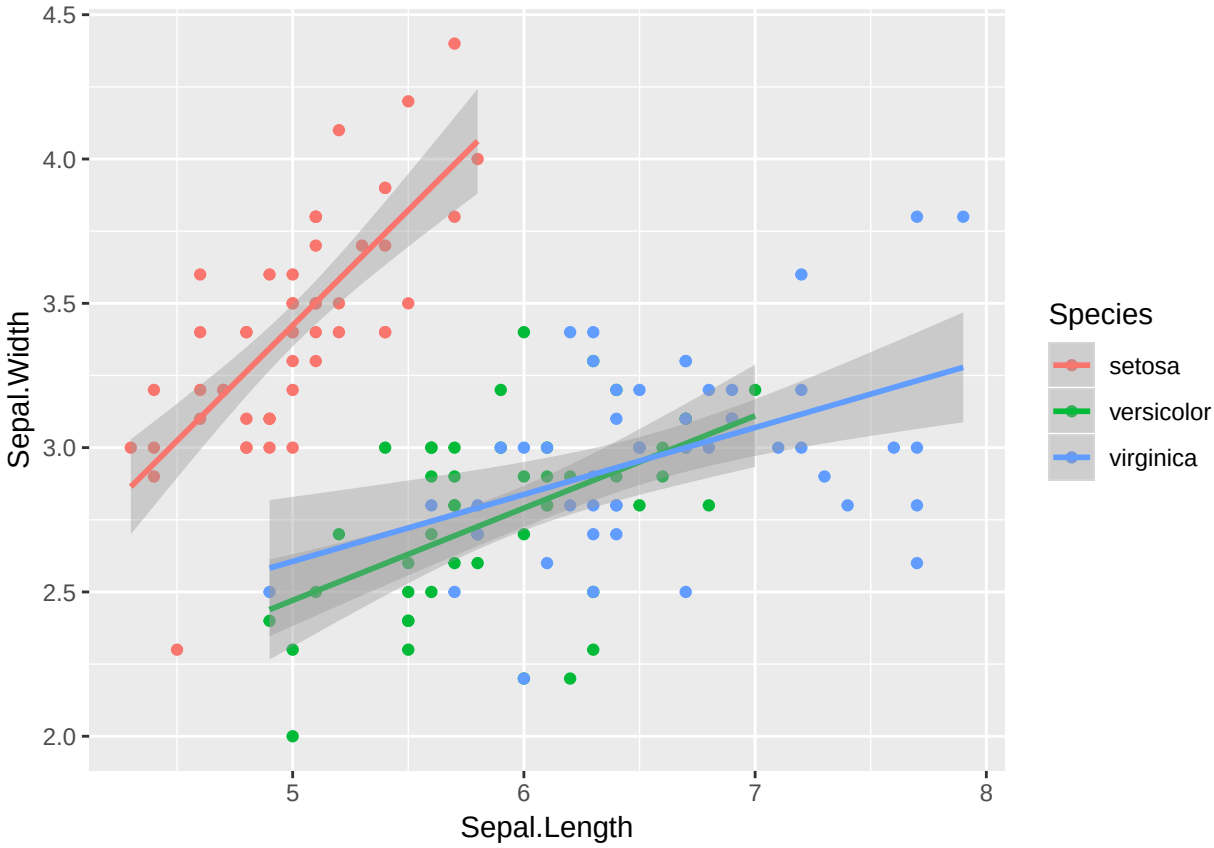
If you're using multiple geoms together and they have aesthetics in common, like `x`, `y`, and `color` in the example above, you can also put these in the call to `ggplot()` and they will be inherited by subsequent layers of the plot:

```
ggplot(data = iris, mapping = aes(x = Sepal.Length, y = Sepal.Width,
    color = Species)) + geom_point() + geom_smooth(method = "lm")
```



And the opposite also works - you can specify `data` separately for each geom. This is especially useful if you're plotting data from multiple datasets in a single plot.

```
ggplot() + geom_point(data = iris, mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + geom_smooth(data = iris,
    mapping = aes(x = Sepal.Length, y = Sepal.Width, color = Species),
    method = "lm")
```

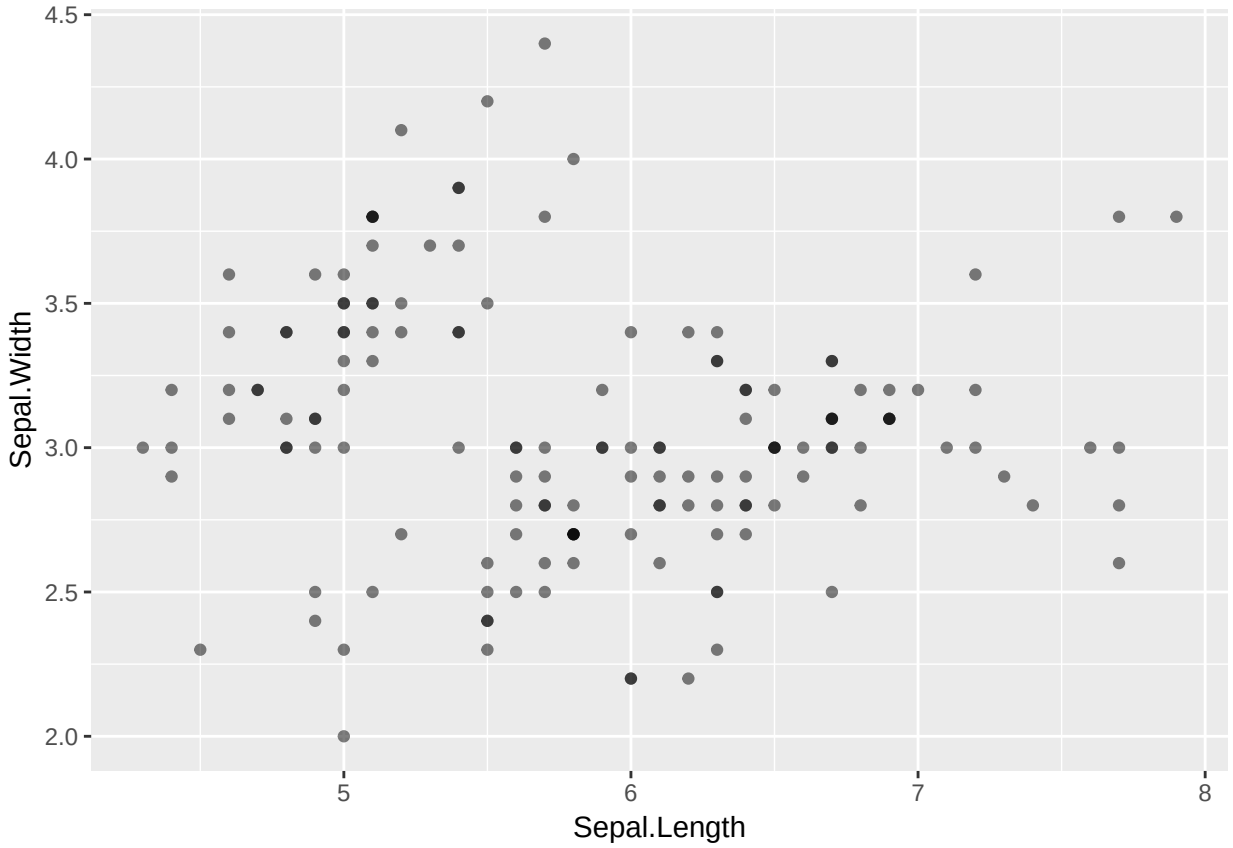


3. The `position` argument

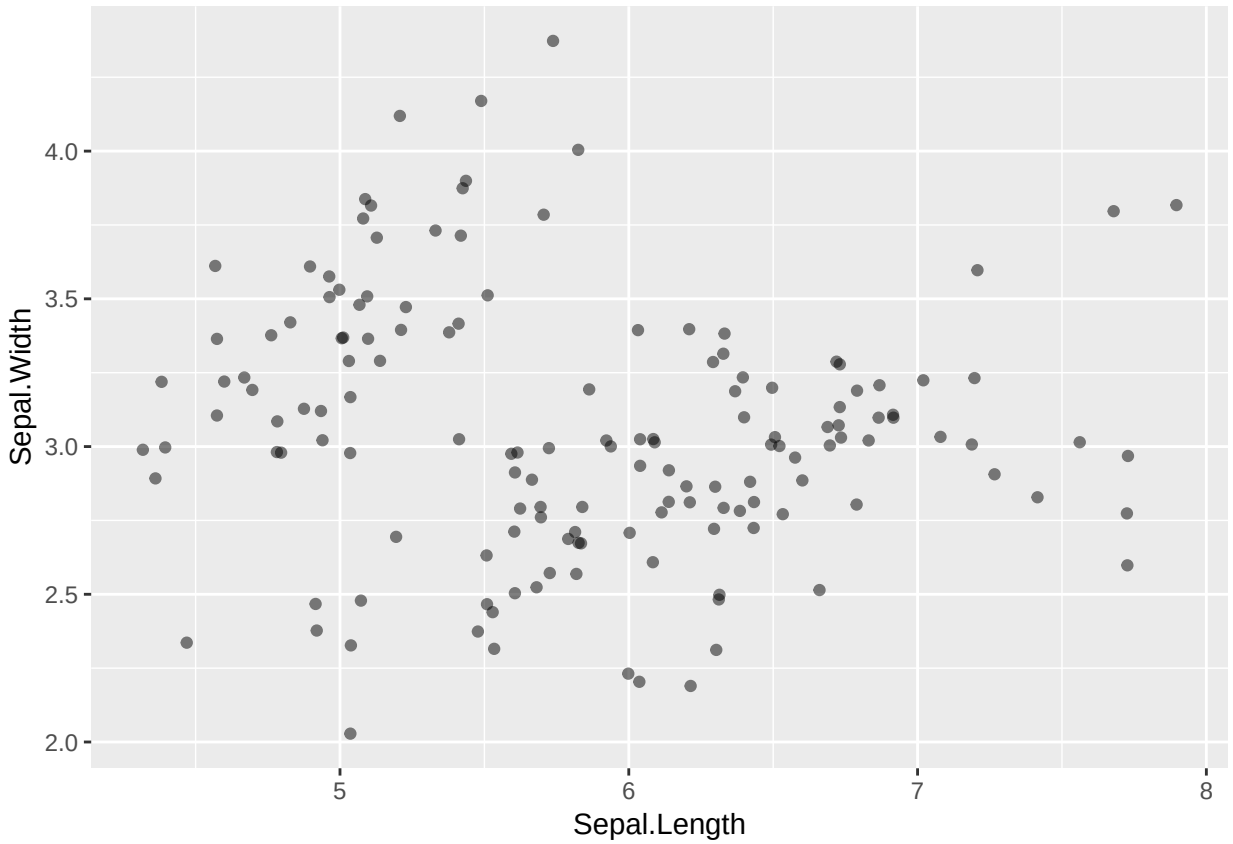
The `position` argument controls where different graphical objects in a single layer fall out relative to each other. It behaves somewhat differently for different geoms, so we'll run through a few examples.

For scatterplots:

```
# the default is 'identity', but as we saw before this can
# result in points stacked on top of each other
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), alpha = 0.5)
```

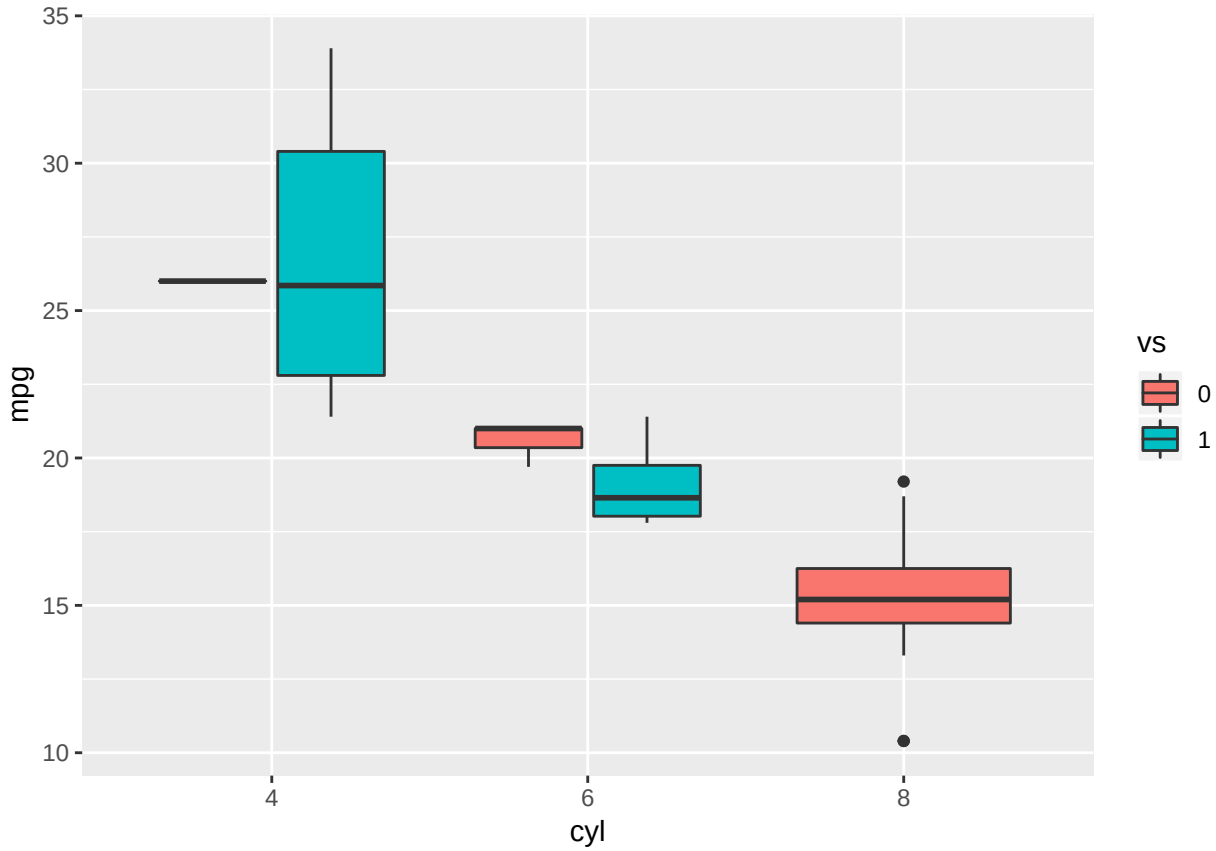



```
# position = 'jitter' adds a little bit of random noise to
# each point so they don't end up on top of each other
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width), alpha = 0.5, position = "jitter")
```

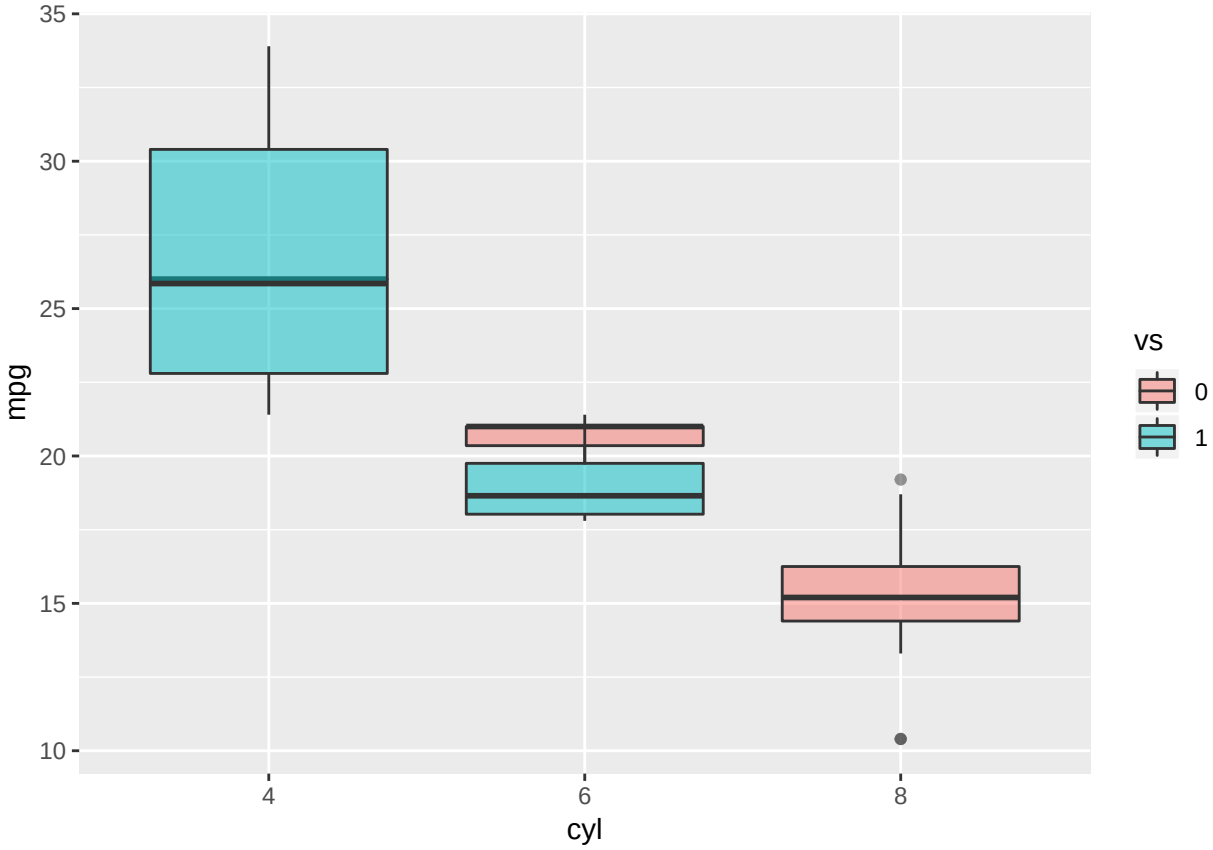


For boxplots:

```
# default is side-by-side
ggplot(data = mtcars) + geom_boxplot(mapping = aes(x = cyl, y = mpg,
    fill = vs))
```

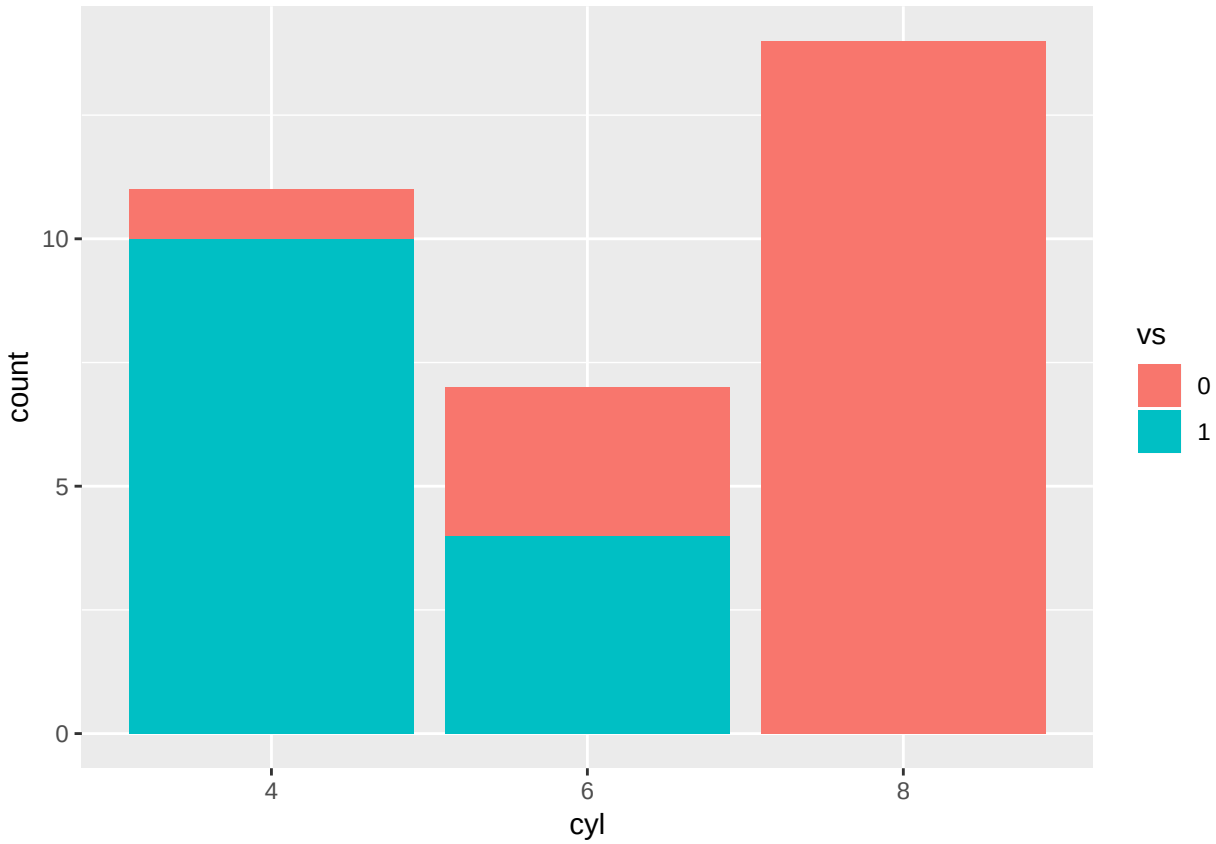



```
# position = 'identity' overlaps the boxplots, but this could
# be useful if you make them transparent
ggplot(data = mtcars) + geom_boxplot(mapping = aes(x = cyl, y = mpg,
    fill = vs), alpha = 0.5, position = "identity")
```

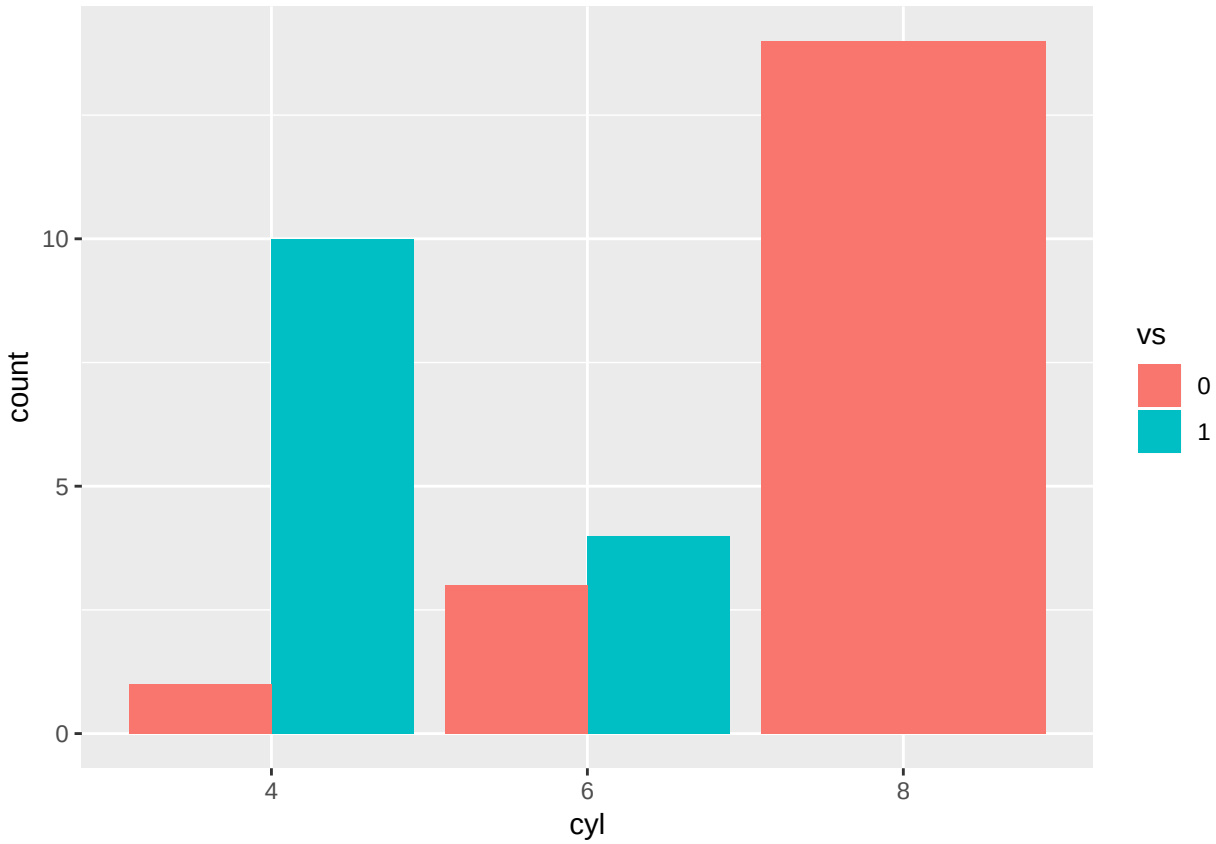


For bar plots:

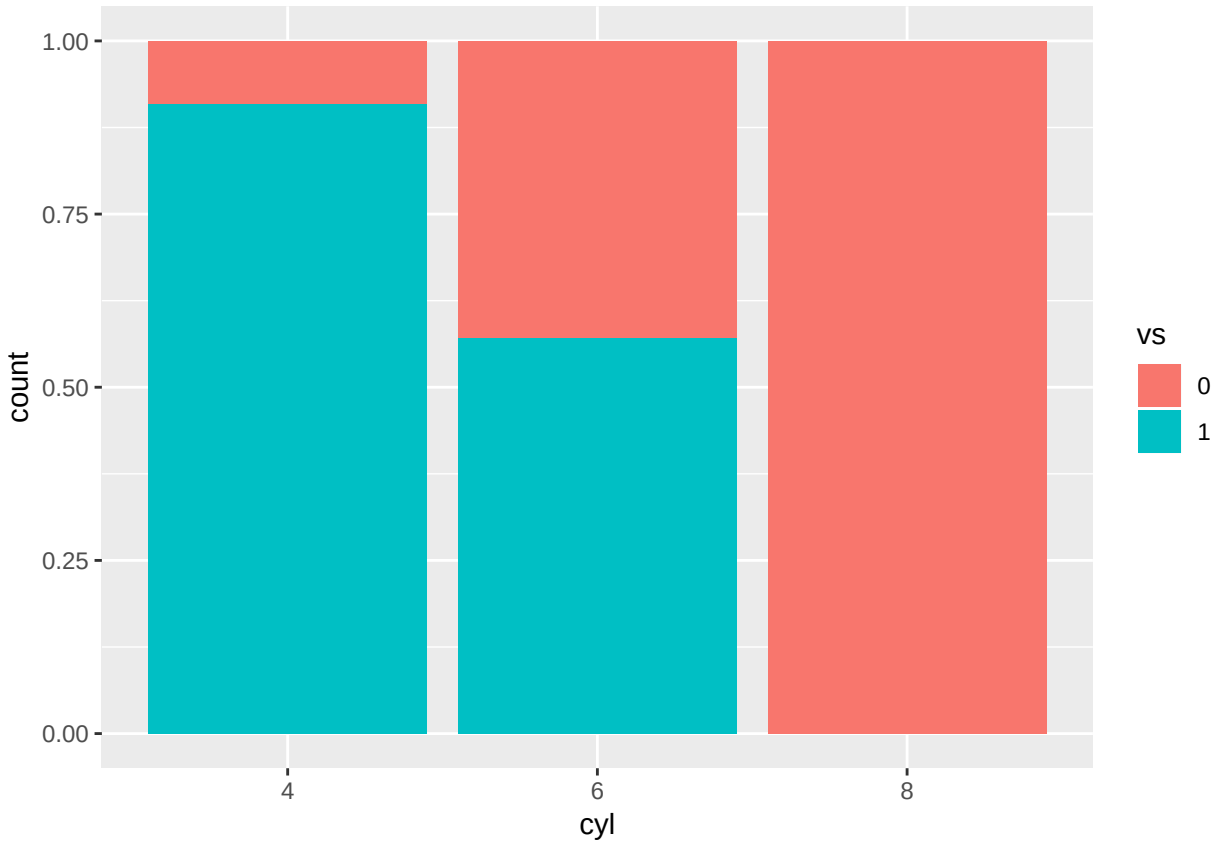
```
# the default is stacked bars
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl, fill = vs))
```



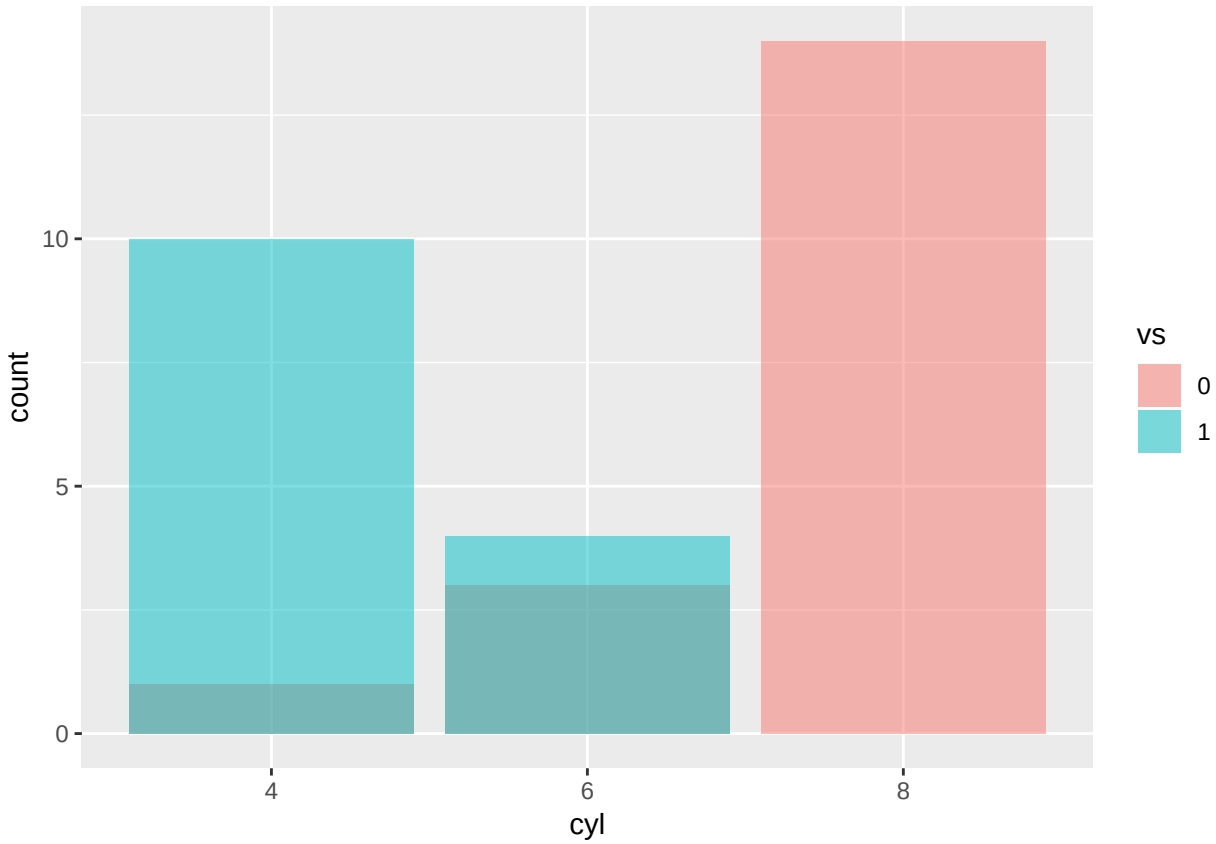

```
# position = 'dodge' puts the bars next to each other instead
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl, fill = vs),
    position = "dodge")
```




```
# position = 'fill' leaves the bars stacked but converts the
# count to a ratio out of 1.0 so that each stack of bars is
# the same height
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl, fill = vs),
    position = "fill")
```

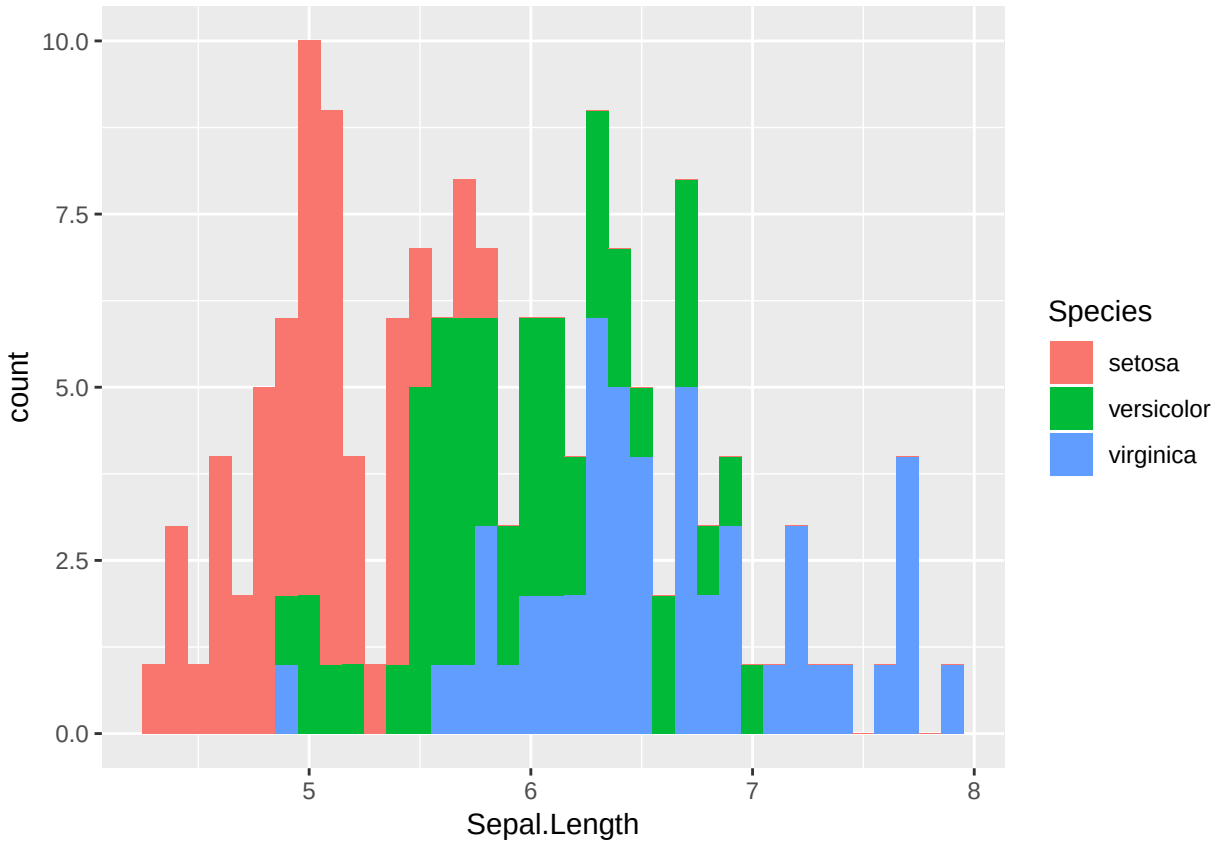



```
# position = 'identity' overlaps the bars, this can be useful
# if you make them transparent
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl, fill = vs),
    alpha = 0.5, position = "identity")
```

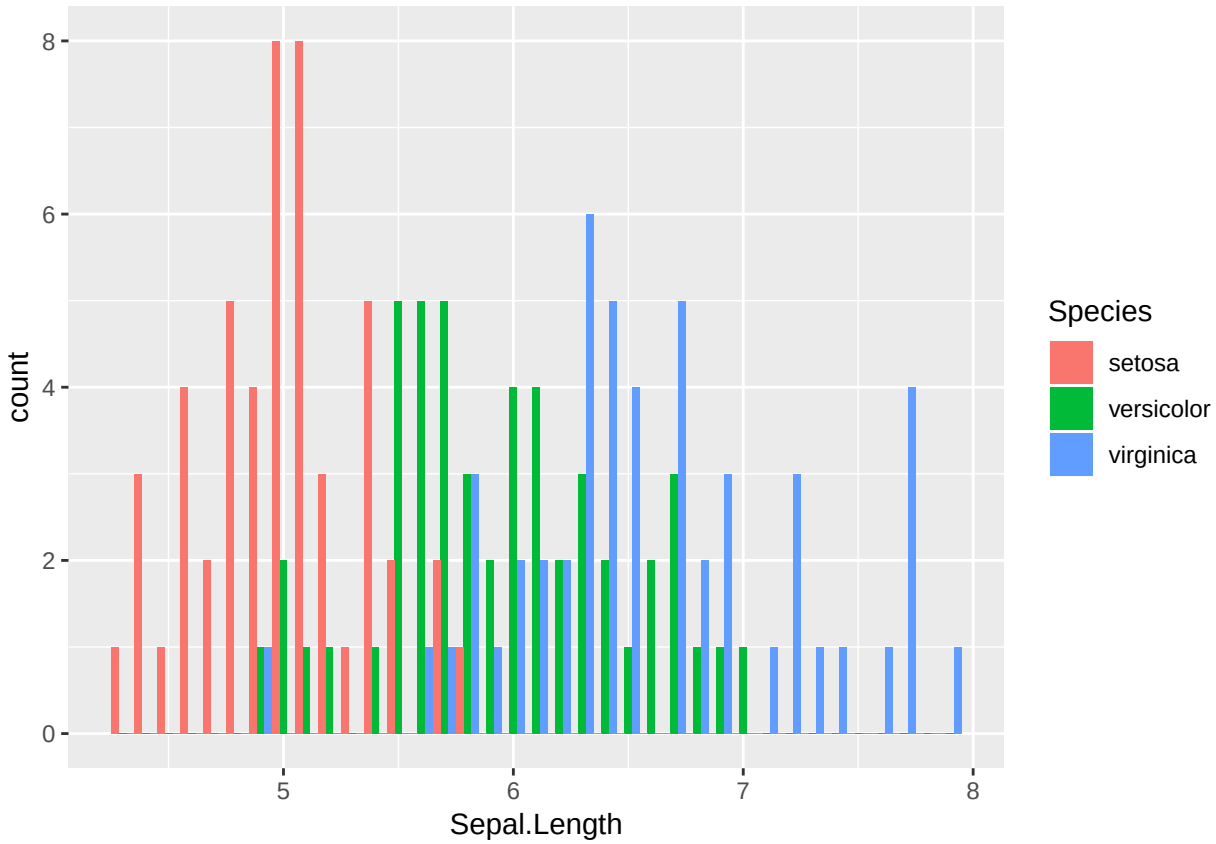


For histograms:

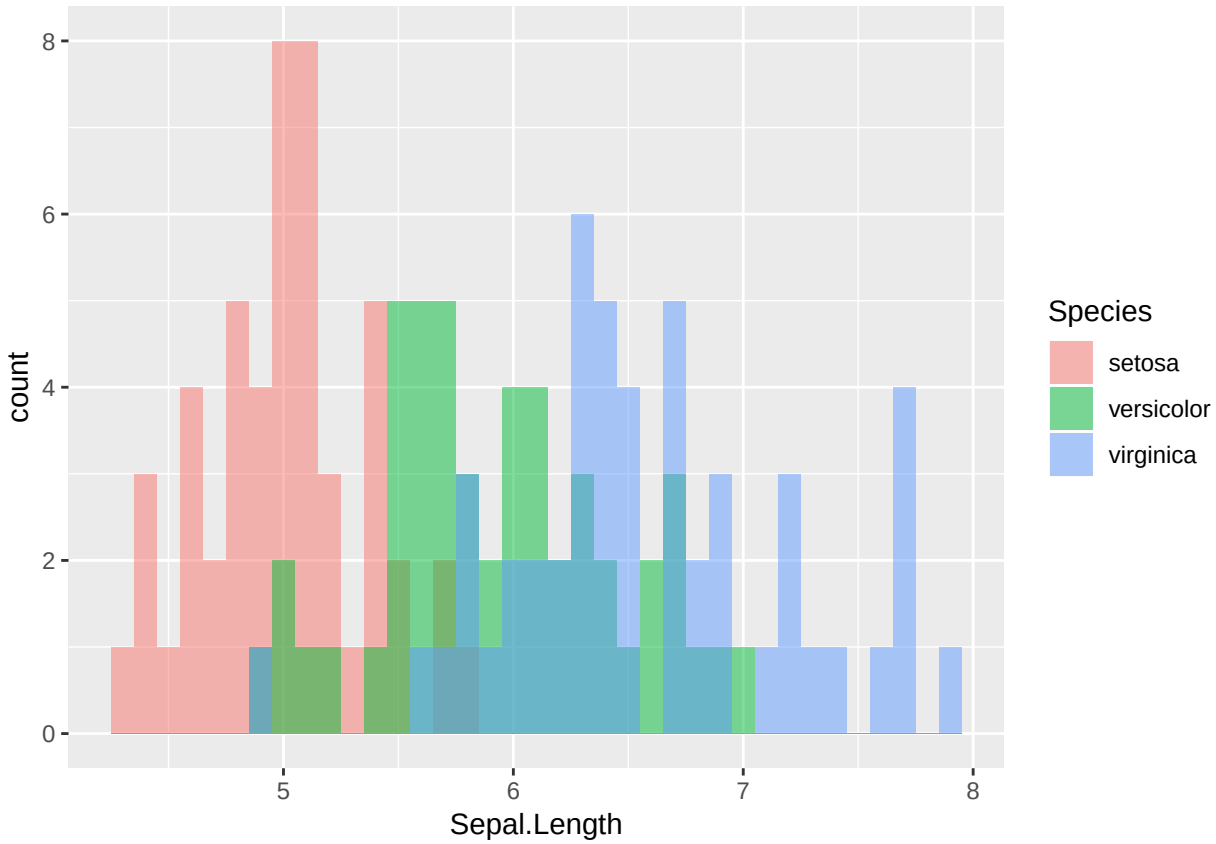
```
# the default is stacked bars
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1)
```




```
# I don't recommend position = 'dodge'
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1, position = "dodge")
```



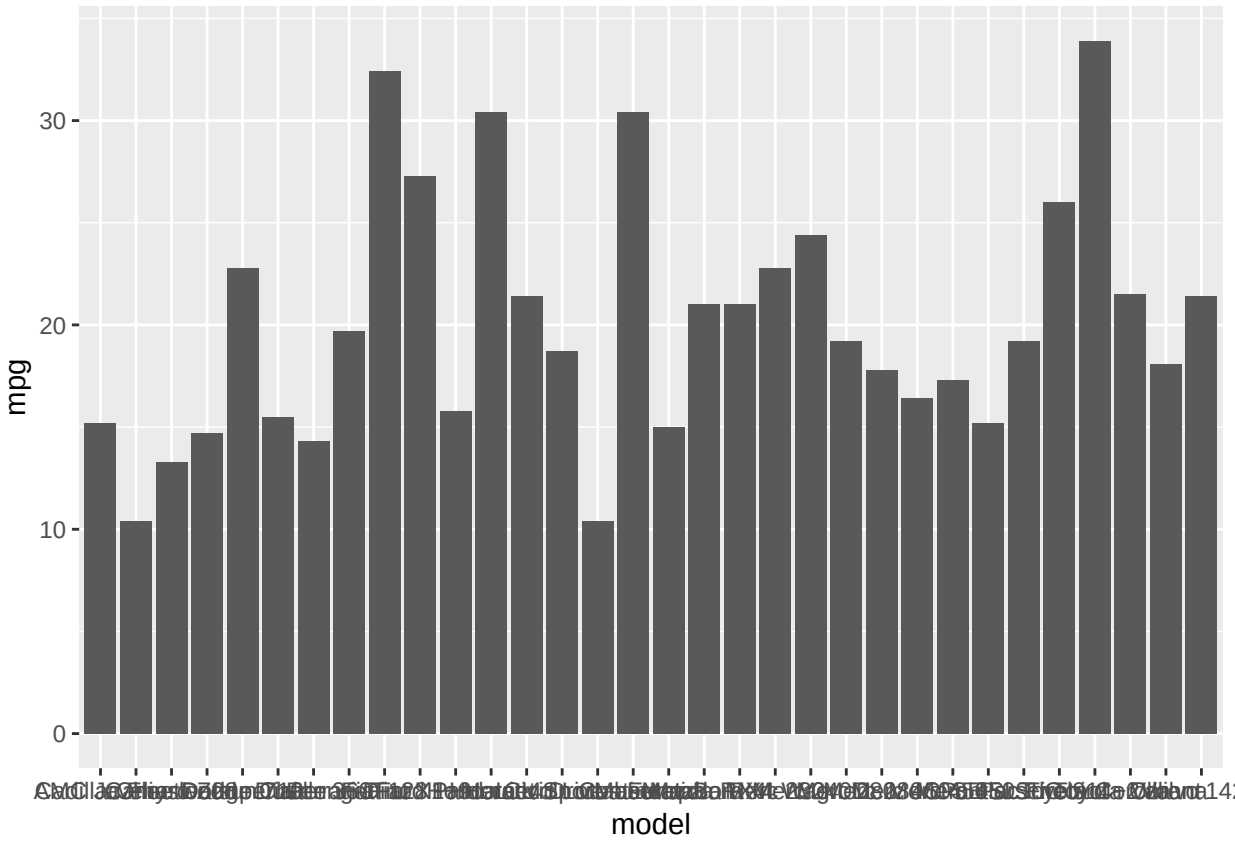

```
# position = 'identity' can be useful, though
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), alpha = 0.5, binwidth = 0.1, position = "identity")
```



4. The `stat` argument

The `stat` argument controls whether a statistic is applied to the data when making a layer. In general, most geoms have a good default `stat` - for example, `geom_histogram()` applies the `count()` stat in order to get a count of how many observations are in each bin. `geom_bar()` does the same for each category. But sometimes you might want to be able to turn that off, for example if you want to make a bar plot and already have a variable that you want to be the height of the bars:

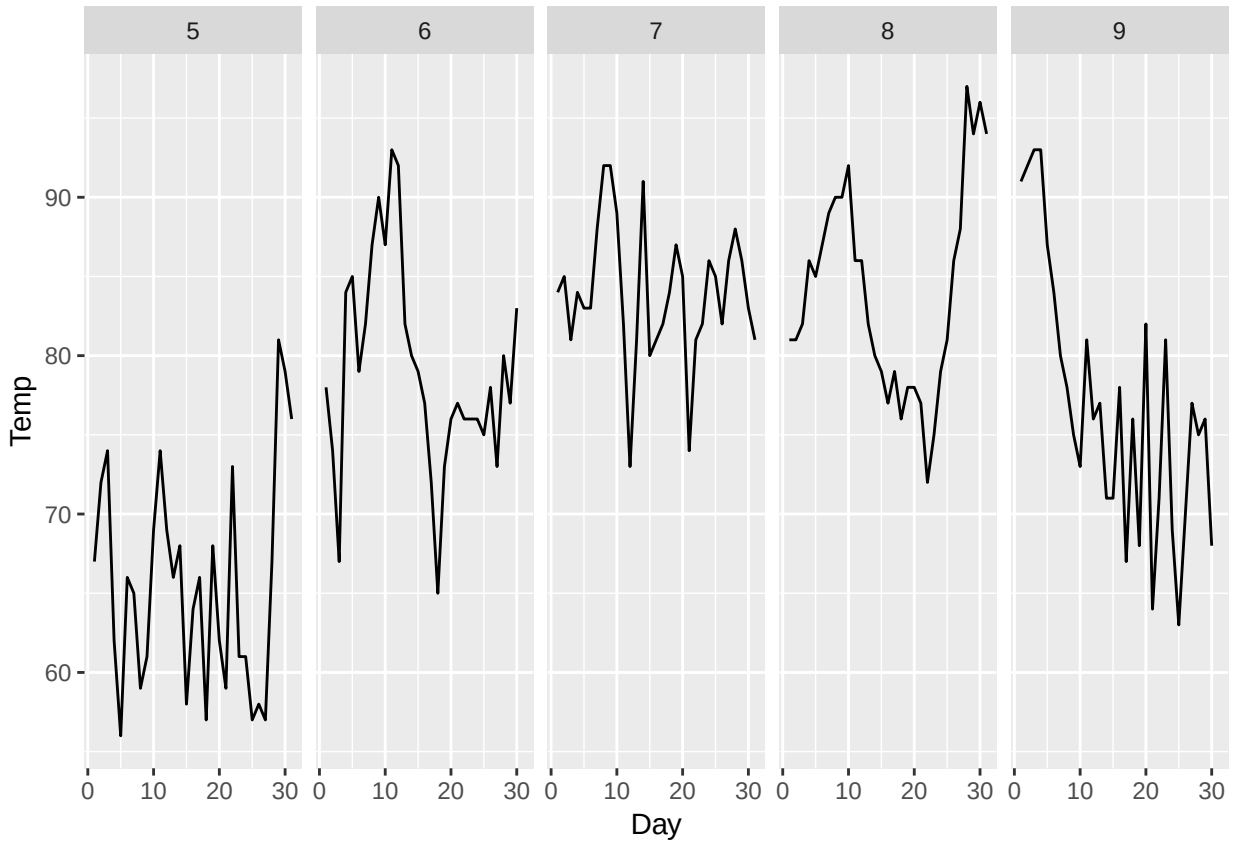
```
mtcars <- rownames_to_column(mtcars, "model")

ggplot(data = mtcars) + geom_bar(mapping = aes(x = model, y = mpg),
    stat = "identity")
```



5. Faceting

Changing directions now, let's talk about faceting. Plots can get pretty messy if you have more than three aesthetics (for example, x, y, and color). Faceting is a way to display more variables at the same time while still keeping your plot readable.

```
ggplot(data = airquality) + geom_path(mapping = aes(x = Day,
    y = Temp)) + facet_grid(. ~ Month)
```



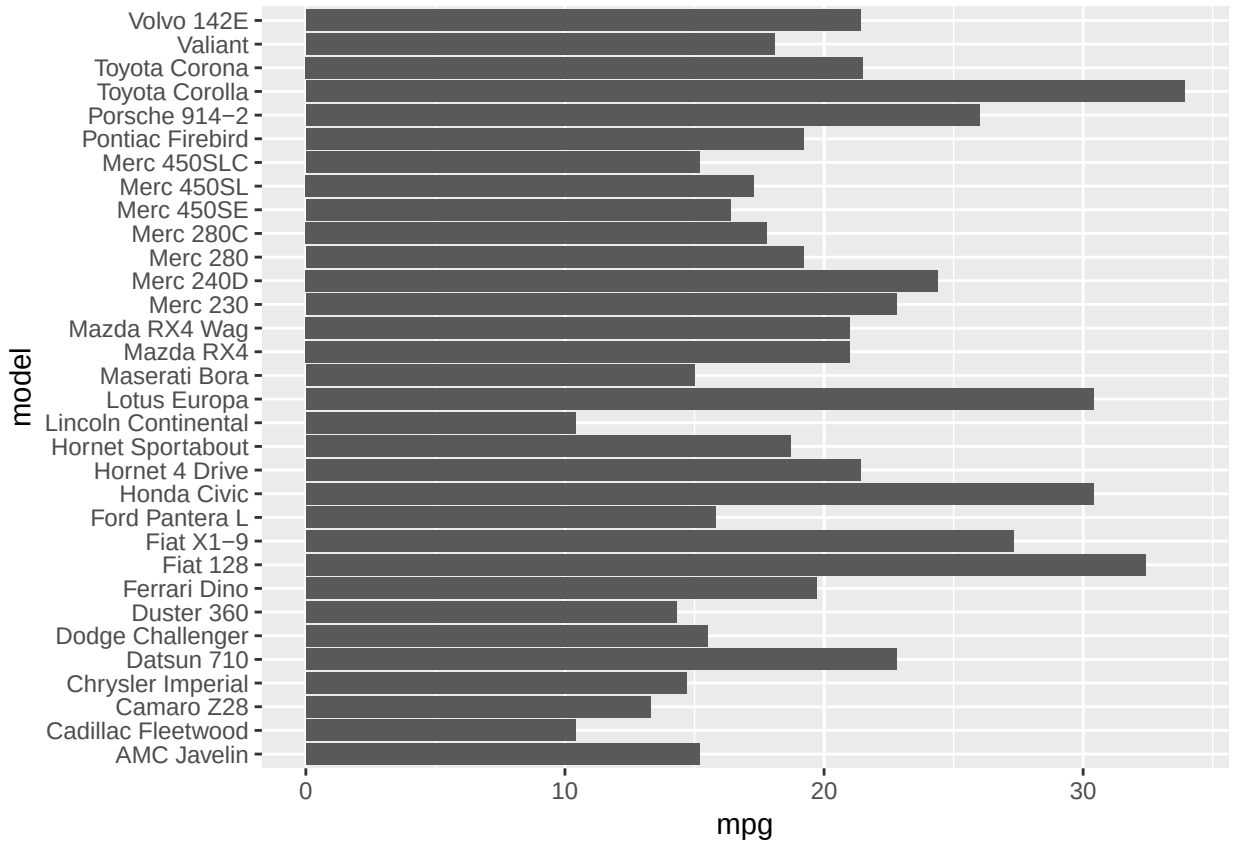
You can facet by one or two variables - take a look at the `ggplot2` cheatsheet for more information.

6. Coordinate systems

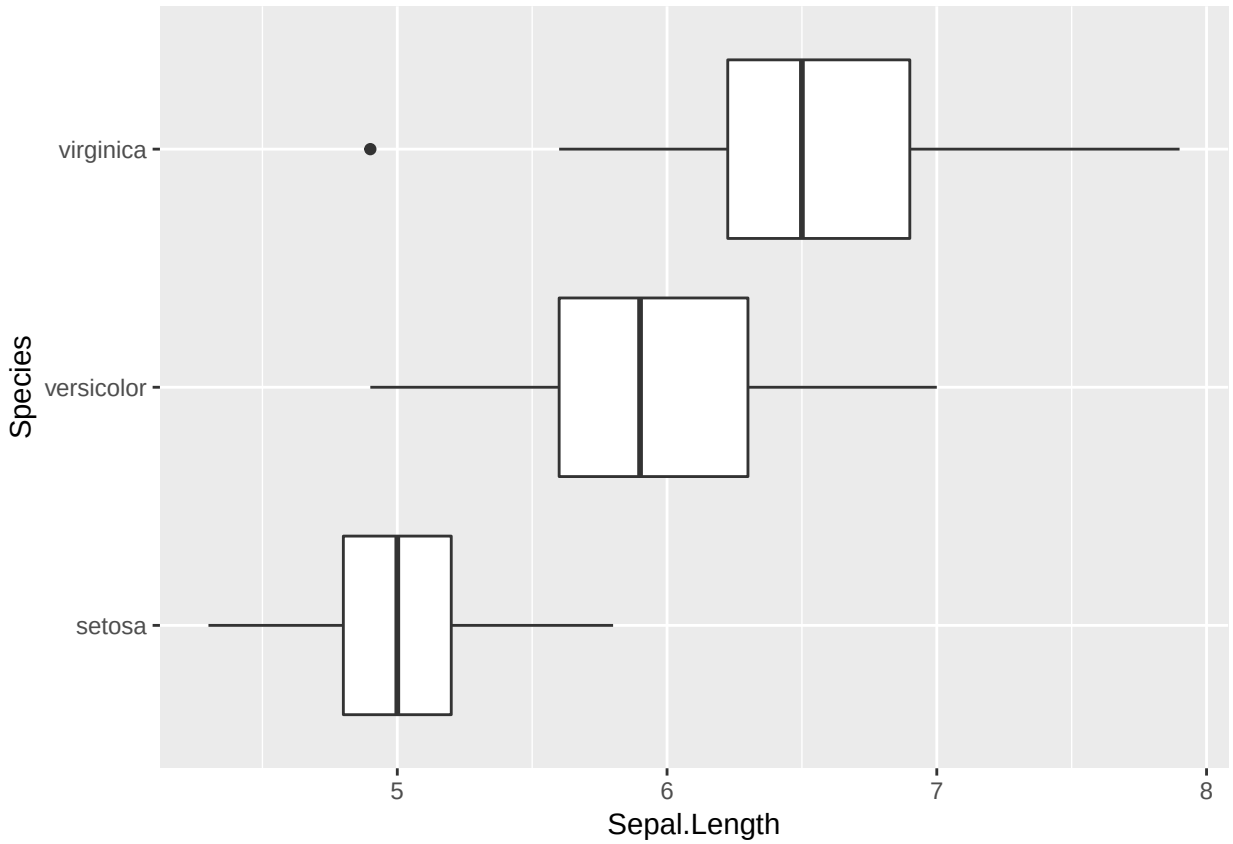
This isn't something you should have to use very often; the default coordinate system is Cartesian coordinates, which work great for almost everything.

However, `coord_flip()` can be useful to make sideways bar plots or boxplots:

```
ggplot(data = mtcars) + geom_bar(mapping = aes(x = model, y = mpg),
    stat = "identity") + coord_flip()
```

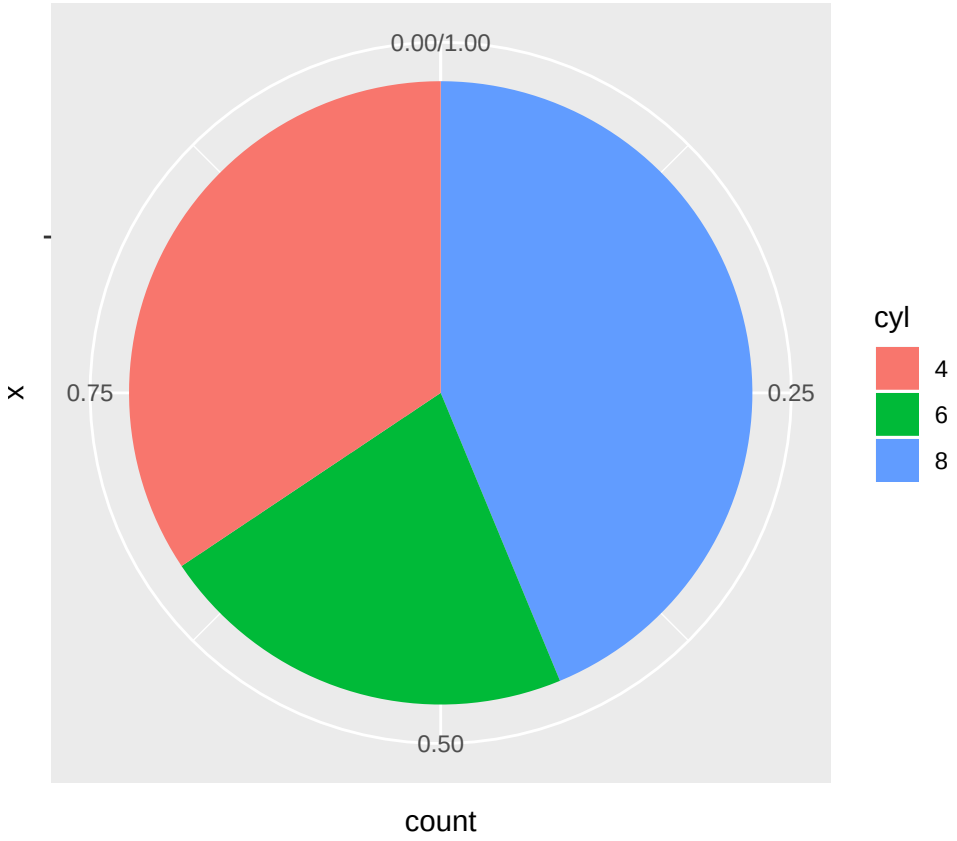



```
ggplot(data = iris) + geom_boxplot(mapping = aes(x = Species,
    y = Sepal.Length)) + coord_flip()
```

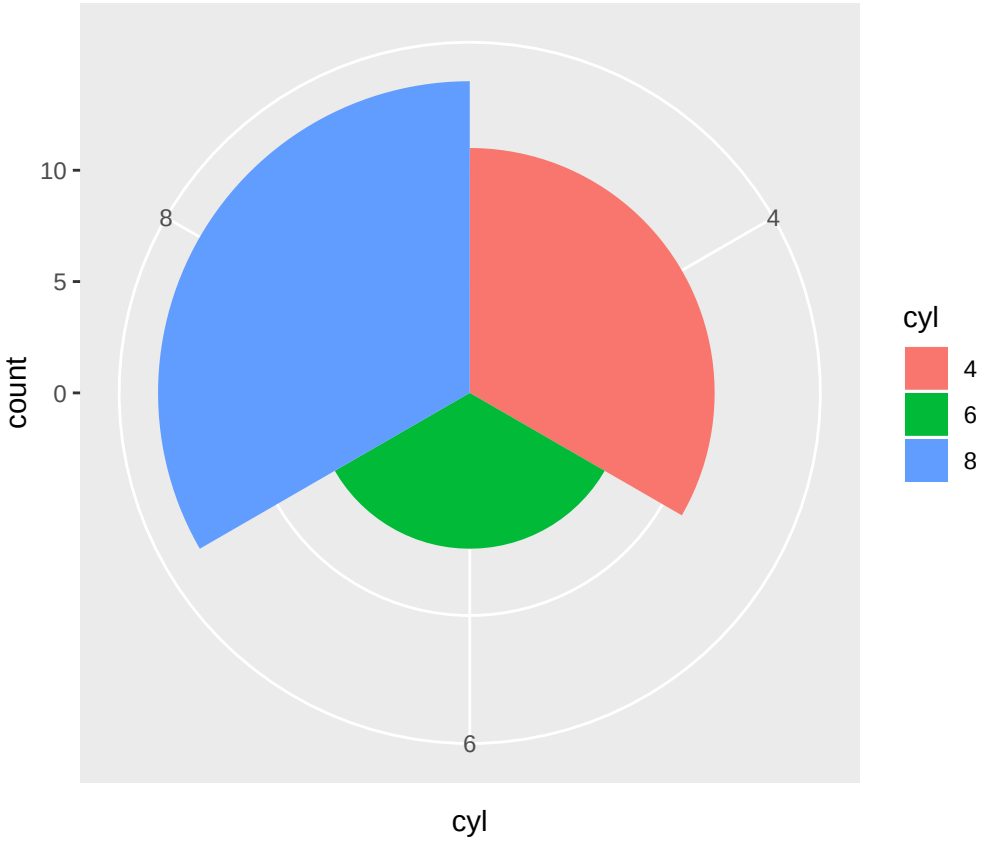


And `coord_polar()` can be useful on the off chance that you need to make a pie chart or coxcomb chart:

```
# pie chart
ggplot(data = mtcars) + geom_bar(mapping = aes(x = "", fill = cyl),
    position = "fill", width = 1) + coord_polar("y")
```




```
# coxcomb chart
ggplot(data = mtcars) + geom_bar(mapping = aes(x = cyl, fill = cyl),
    position = "dodge", width = 1) + coord_polar()
```



PART 3: Making your plots pretty and readable

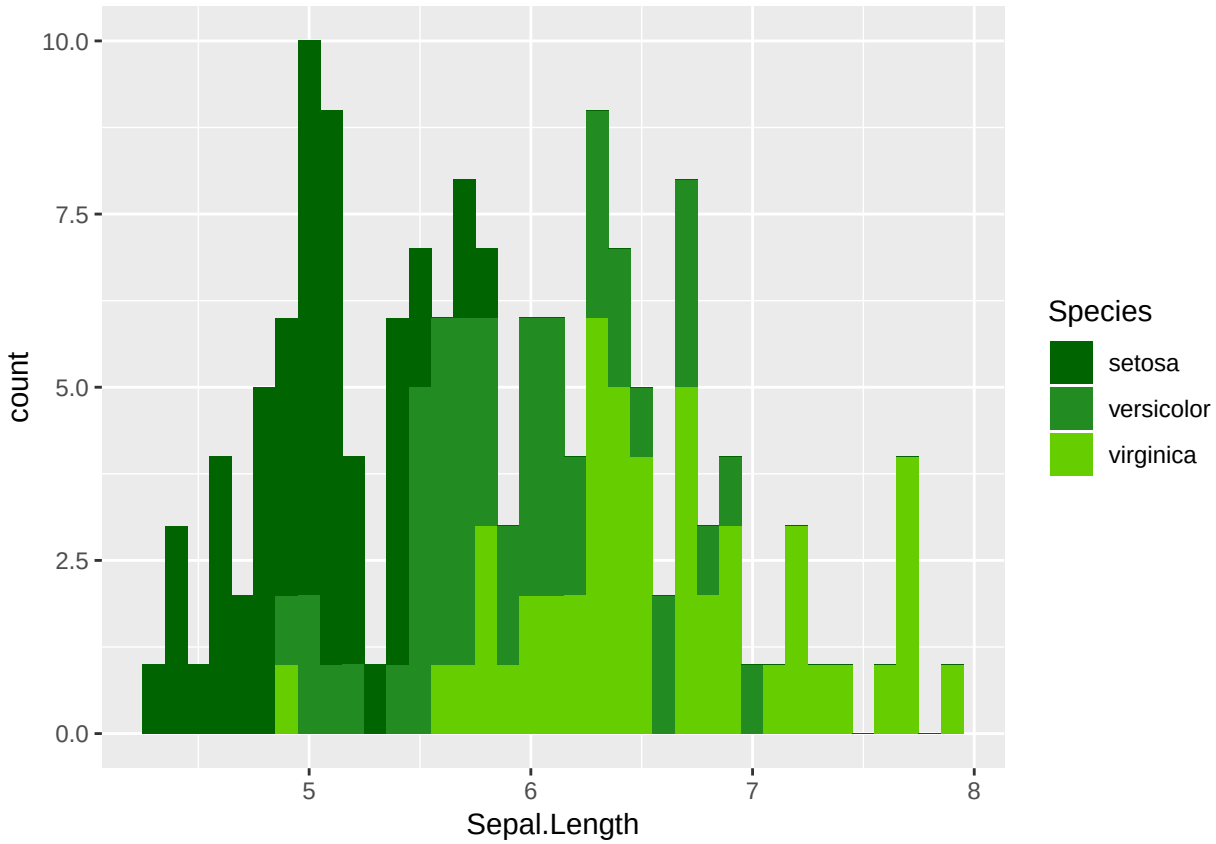
`ggplot2`'s grammar of graphics makes it possible for you to create any sort of plot you can imagine... but once you have all of the elements on the page, there are still lots of things you can tweak to make your plots *pretty*. We'll go over a couple major categories: choosing colors, controlling axis labels/ranges/breakpoints, annotating a plot with text/shapes, and dealing with legends.

1. Color

By default, `ggplot2` uses colors that are evenly distributed around the color wheel for categorical variables, and a range of dark to light blues for continuous variables. While these defaults are fine for just getting a first glance at your data, you'll probably want to choose your own colors when making a plot for someone else to look at.

If you have `color` and/or `fill` mapped to a categorical variable, use the functions `scale_color_manual()` and/or `scale_fill_manual()`:

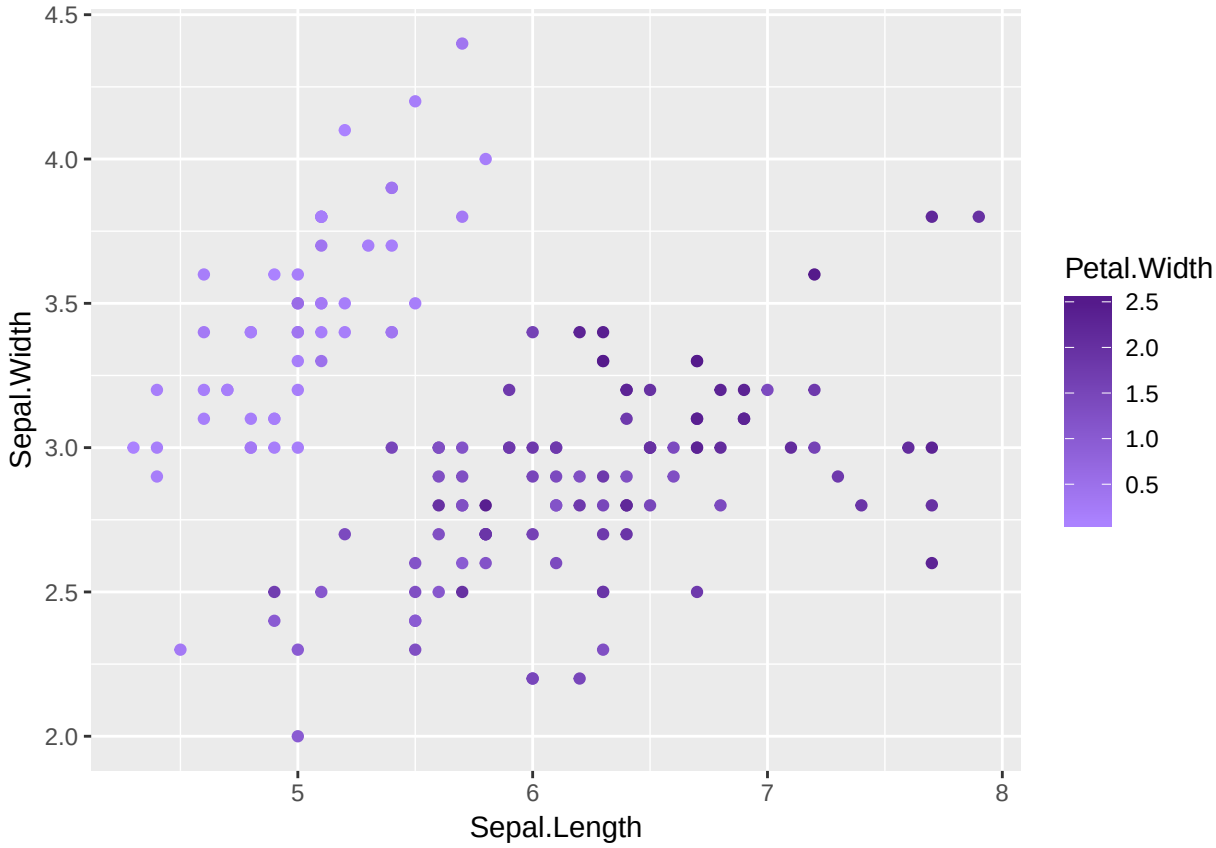
```
# specify the 3 colors to use for the 3 species
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1) + scale_fill_manual(values = c("darkgreen",
    "forestgreen", "chartreuse3"))
```



Note that color names are supplied to the `values` argument as a vector of strings. For the names of colors in R, refer to the resource document **Rcolor.pdf**.

If you have `color` and/or `fill` mapped to a continuous variable, use the `scale_color_gradient()` / `scale_fill_gradient()` family of functions.

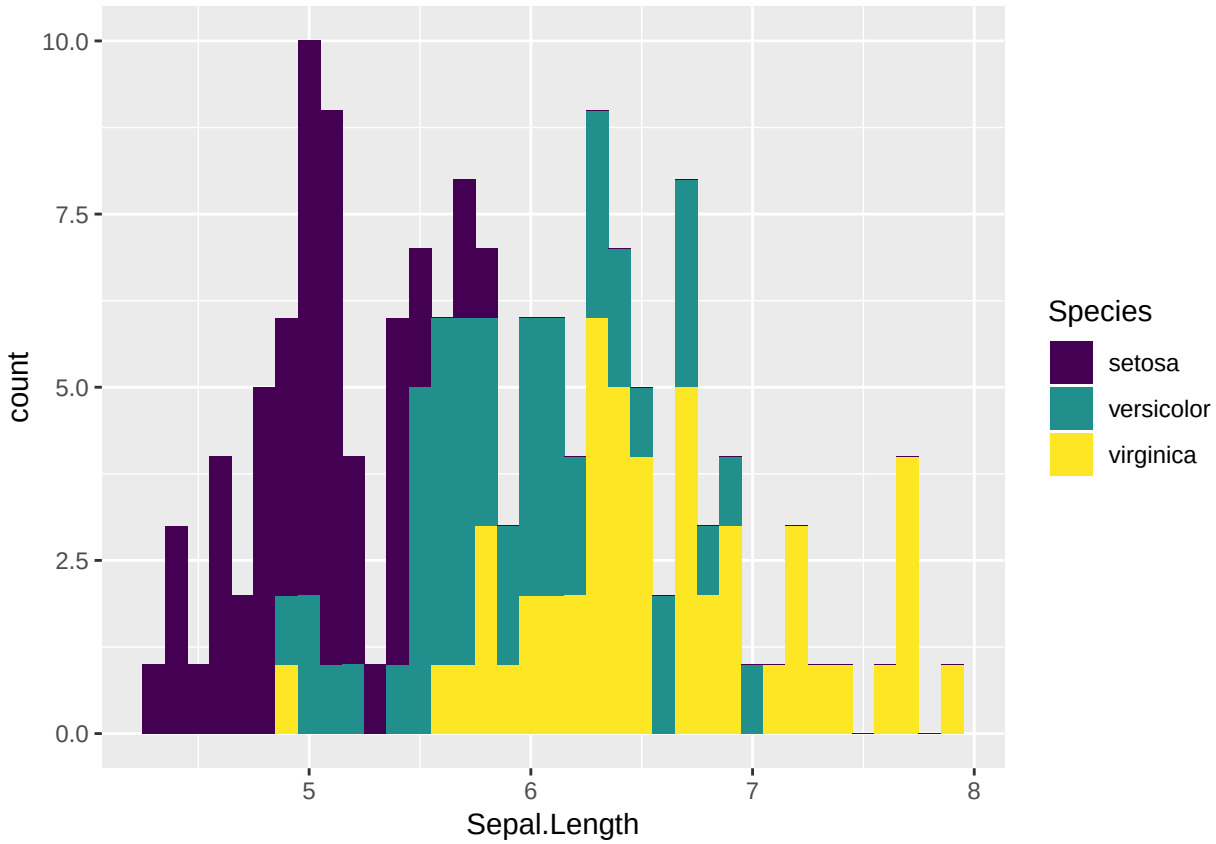
```
# to get a gradient between two specified colors
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Petal.Width)) + scale_color_gradient(low = "mediumpurple1",
    high = "purple4")
```



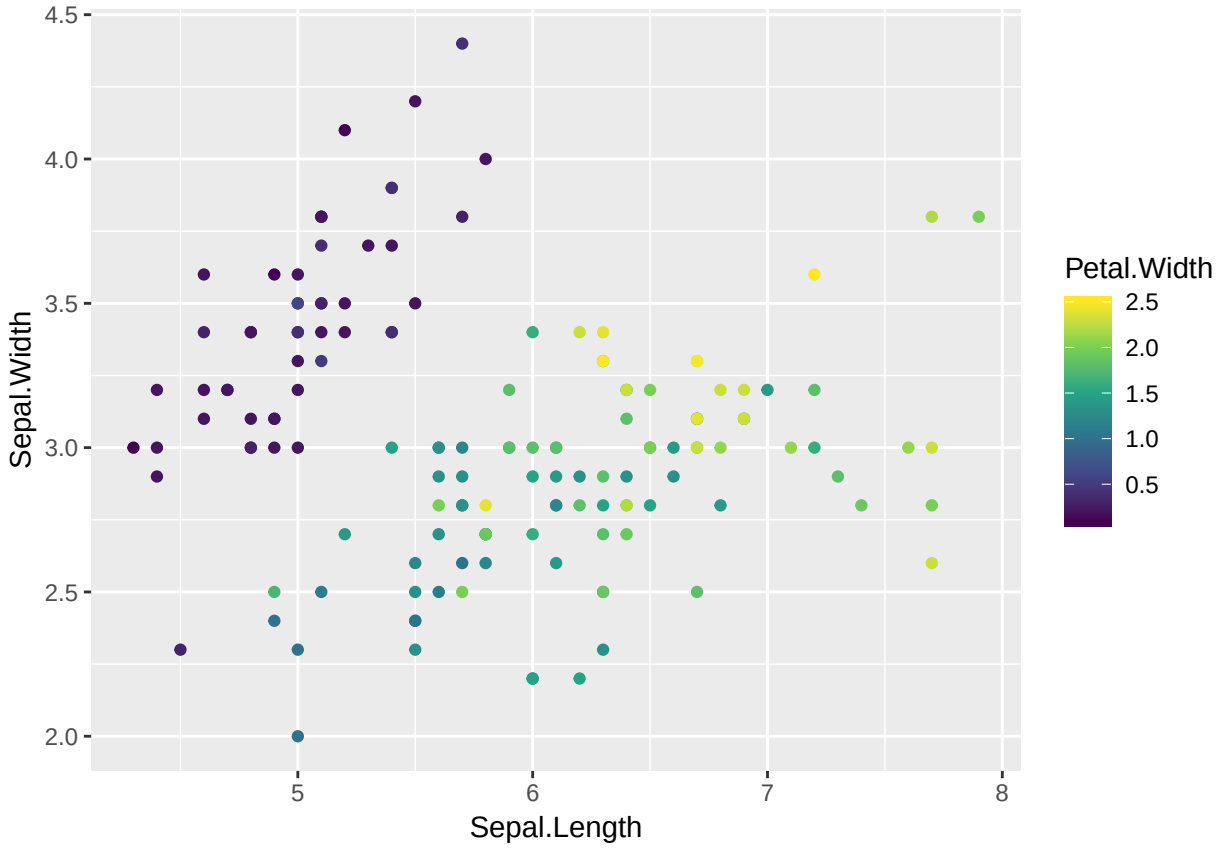

```
# use scale_color_gradient2(low = , mid = , high = ) for a
# diverging color gradient centered on zero
```

Alternately, use the `viridis` scales for beautiful and colorblindness-friendly colors.

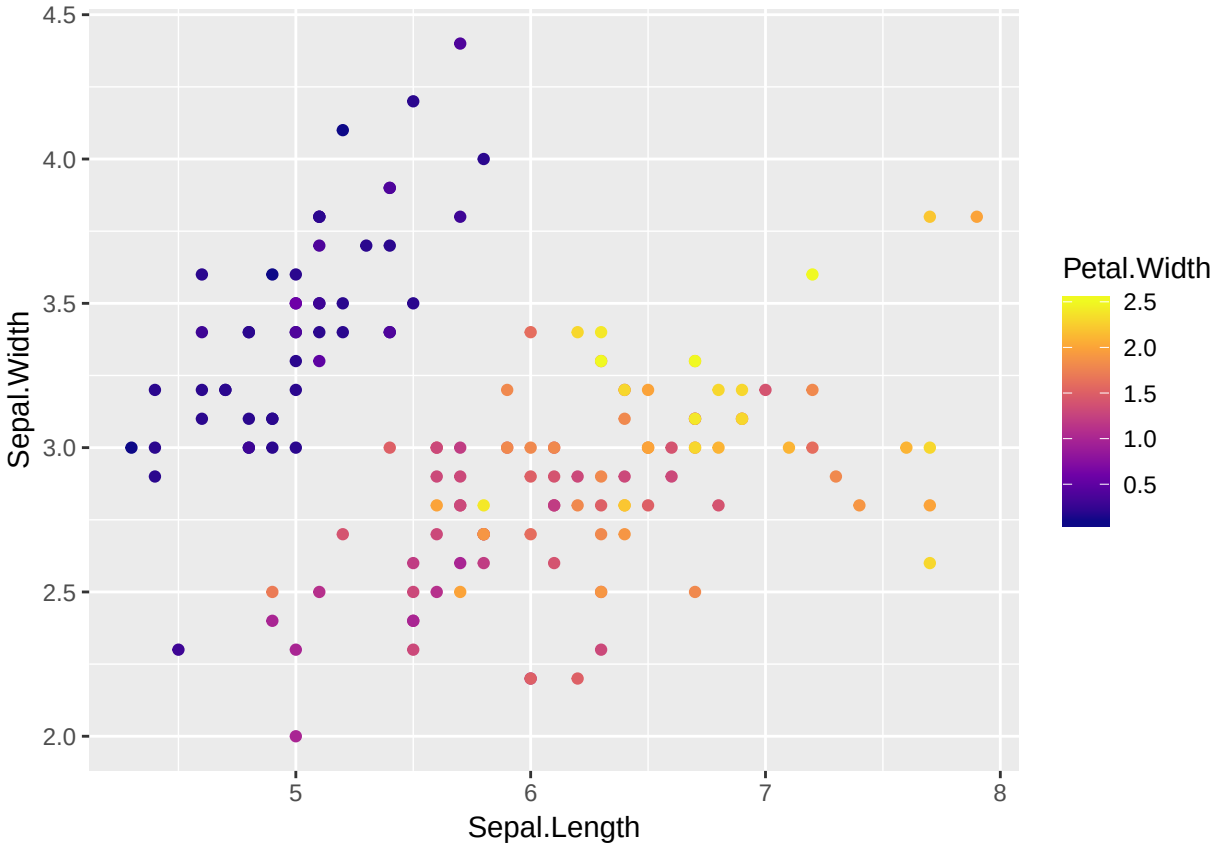
```
# scale_color_viridis_d or scale_fill_viridis_d for discrete
# scales
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1) + scale_fill_viridis_d()
```




```
# scale_color_viridis_c or scale_fill_viridis_c for
# continuous scales
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Petal.Width)) + scale_color_viridis_c()
```




```
# there are also alternative viridis scales named magma,
# plasma, and inferno use the `option` argument to select one
# of these
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Petal.Width)) + scale_color_viridis_c(option = "plasma")
```

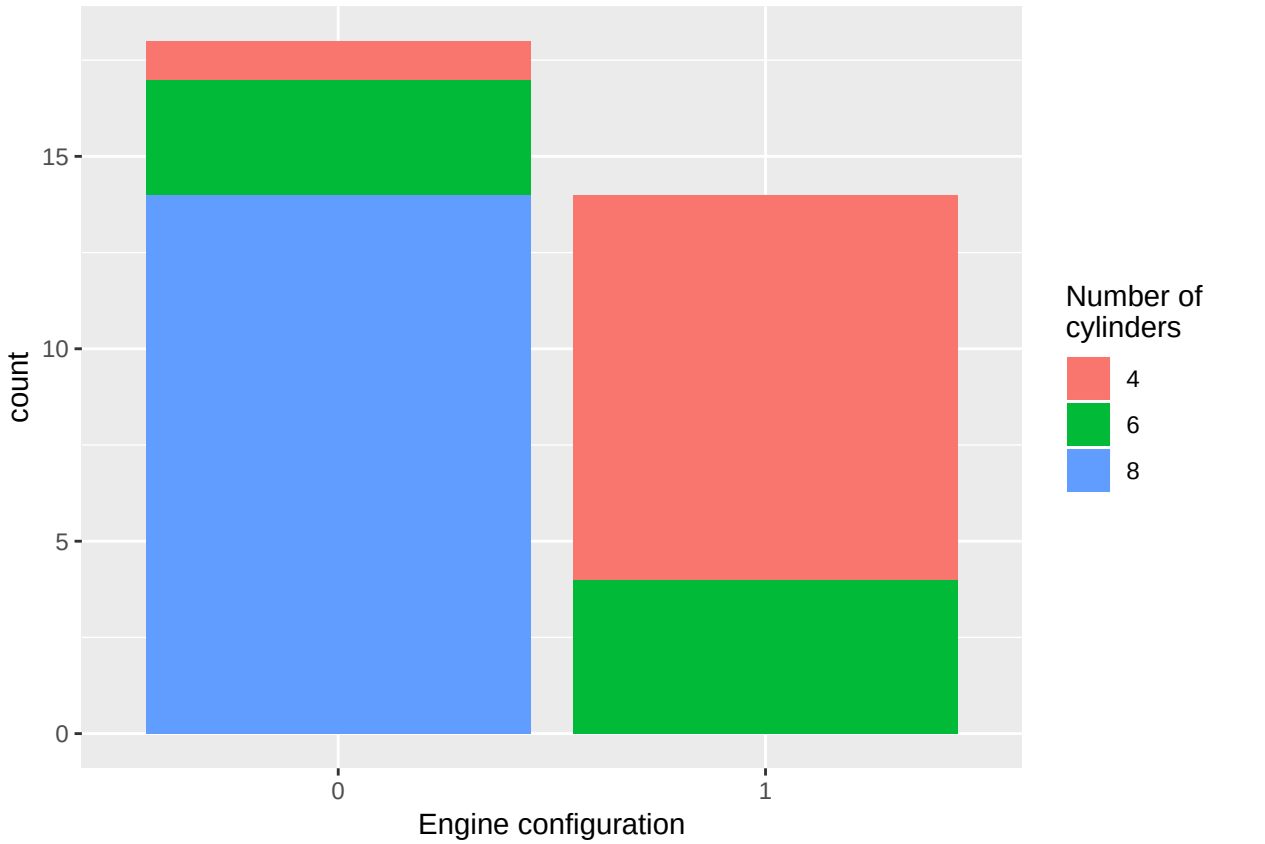


For more information on the `viridis` scales, see: https://cran.r-project.org/web/packages/viridis/vignettes/intro-to-viridis.html

2. Axes

`ggplot2` automatically labels axes with the name of the variable that is mapped to each axis, but your variable names may not be meaningful to another person. To re-label your axes, use `labs()`:

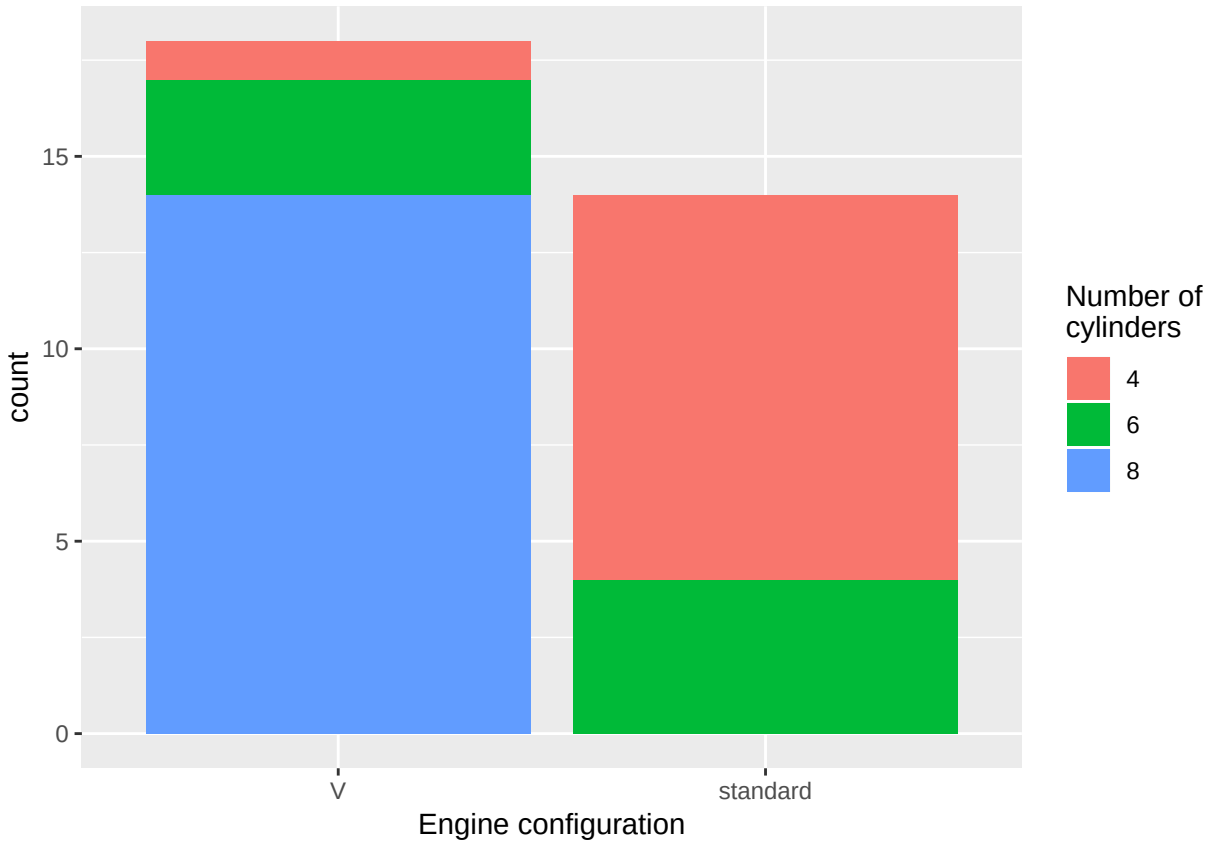
```
ggplot(data = mtcars) + geom_bar(mapping = aes(x = vs, fill = cyl)) +
    labs(x = "Engine configuration", fill = "Number of \ncylinders")
```




```
# \n creates a line break between 'Number of' and
# 'cylinders' so things fit a little more comfortably on the
# page
```

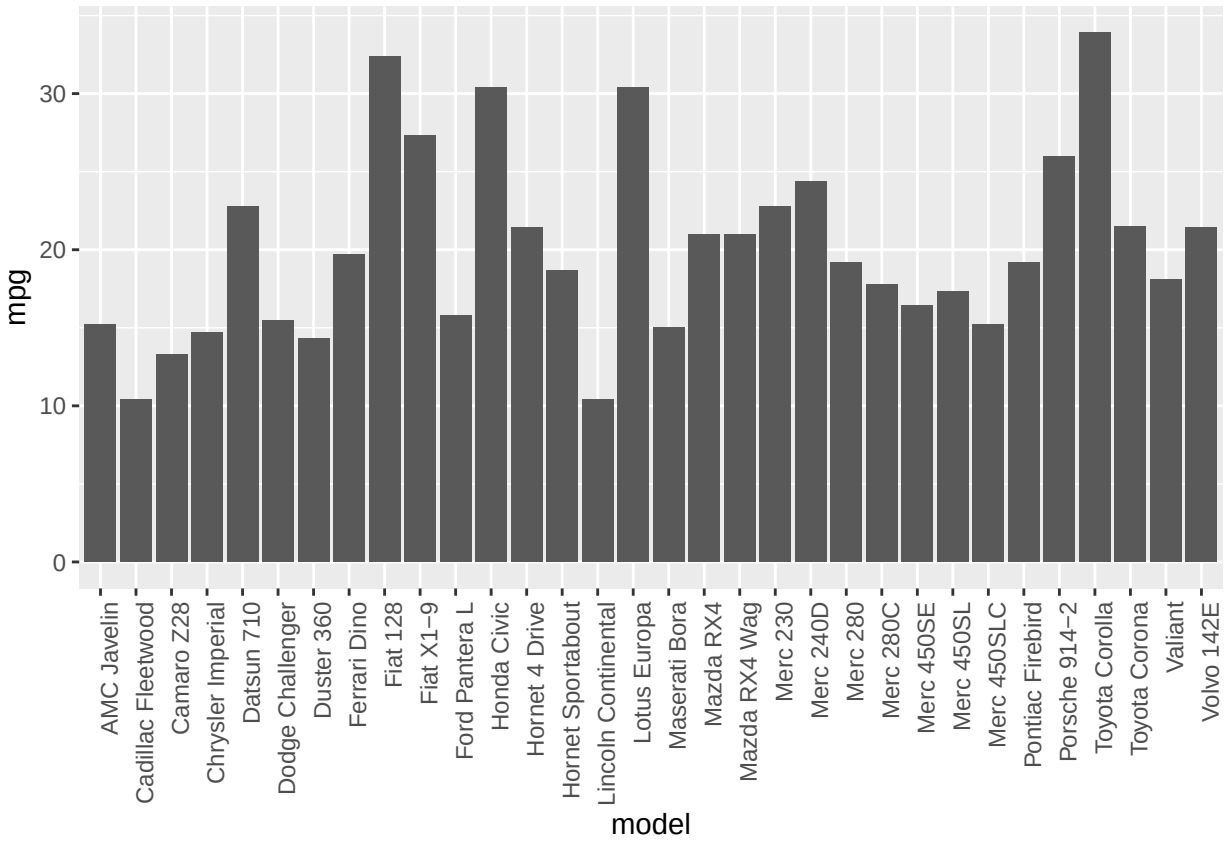
Similarly, `ggplot2` automatically labels the bins for a categorical variable with the values of that variable. To change this, you can use the `labels` argument of the `scale_x_discrete()` function:

```
ggplot(data = mtcars) + geom_bar(mapping = aes(x = vs, fill = cyl)) +
    labs(x = "Engine configuration", fill = "Number of \ncylinders") +
    scale_x_discrete(labels = c("V", "standard"))
```



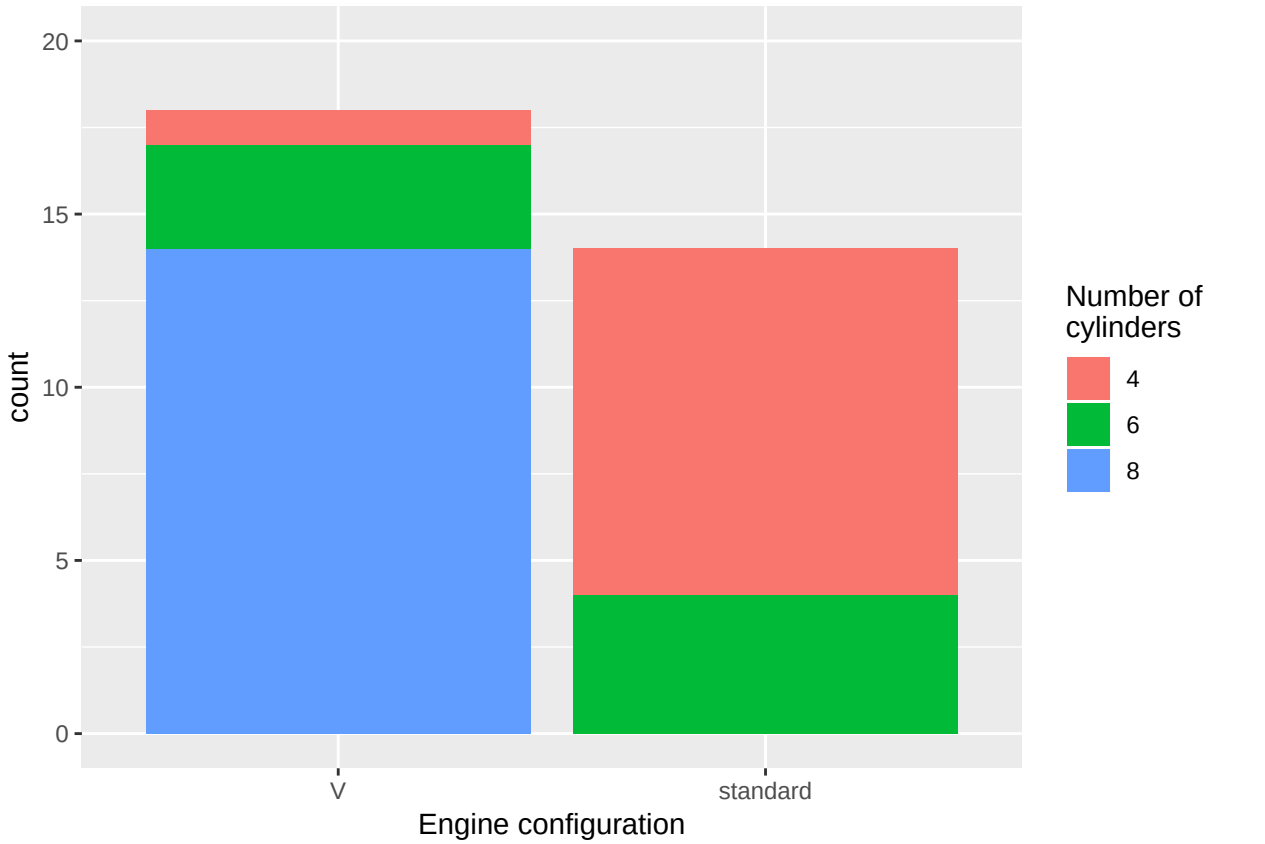
It's also possible to rotate labels using `theme(axis.text.x = )`.

```
ggplot(data = mtcars) + geom_bar(mapping = aes(x = model, y = mpg),
    stat = "identity") + theme(axis.text.x = element_text(angle = 90,
    hjust = 1))
```



For continuous scales, it can be useful to control the range and breakpoints. Range is fairly self-explanatory; we will demonstrate how it can be set with `scale_y_continuous()`.

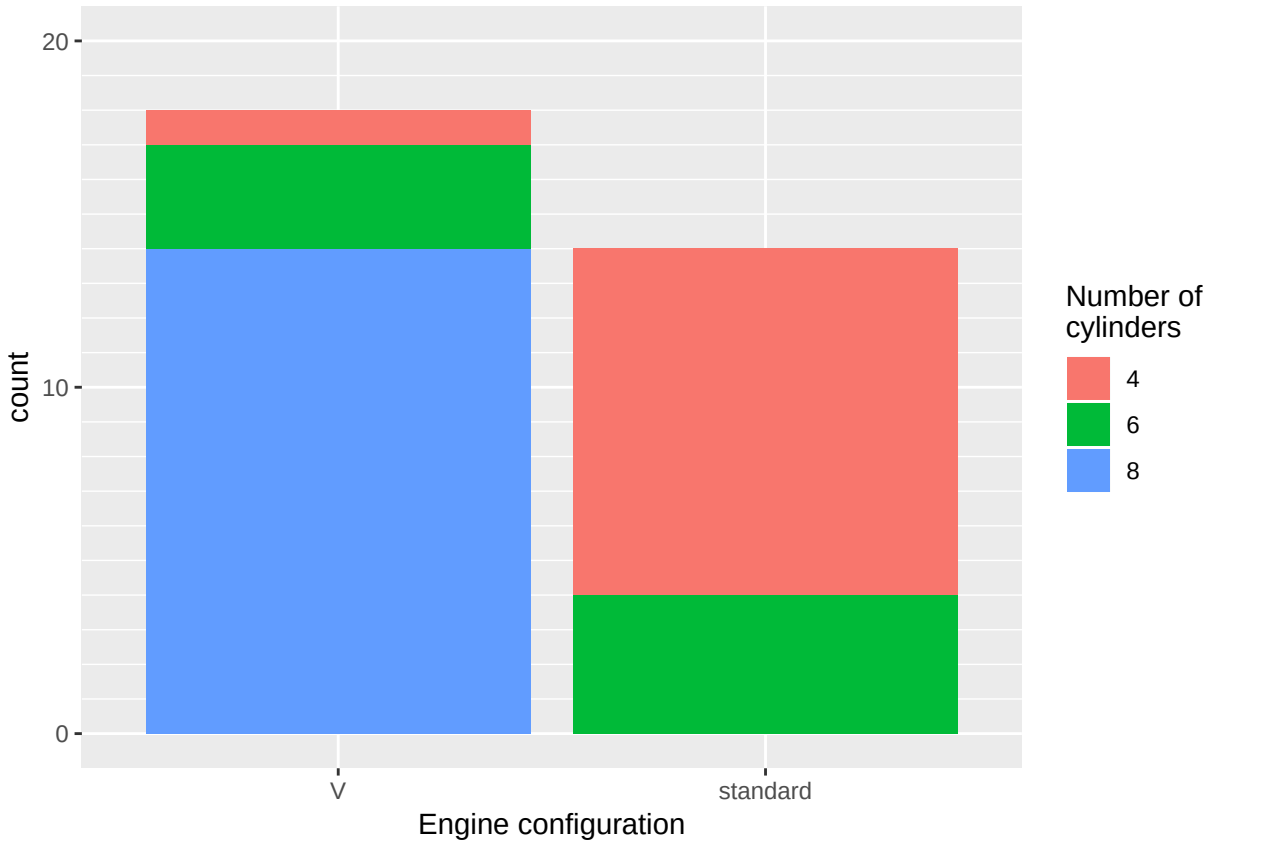
```
# make this plot go to 20, instead of ending immediately
# above the data
ggplot(data = mtcars) + geom_bar(mapping = aes(x = vs, fill = cyl)) +
    labs(x = "Engine configuration", fill = "Number of \ncylinders") +
    scale_x_discrete(labels = c("V", "standard")) + scale_y_continuous(limits = c(0,
    20))
```




```
# this is not terribly useful on its own, but can be
# extremely helpful when working with percentage data or when
# you plan to put two plots side-by-side and want their
# scales to match.
```

`scale_y_continuous()` and the like can also be used to control breakpoints. `ggplot2` recognizes both major and minor breakpoints. Major breaks are indicated by a labeled tickmark and a thick white gridline; minor breaks receive a thinner gridline and no tickmark.

```
ggplot(data = mtcars) + geom_bar(mapping = aes(x = vs, fill = cyl)) +
    labs(x = "Engine configuration", fill = "Number of \ncylinders") +
    scale_x_discrete(labels = c("V", "standard")) + scale_y_continuous(limits = c(0,
    20), breaks = seq(0, 20, by = 10), minor_breaks = seq(0,
    20, by = 1))
```




```
# I am using seq() to create a sequence of numbers that
# denote break locations, but you can also supply any vector
# of numbers
```

There are so many little tiny ways in which axis labels can be modified, so it's a good idea to read the `ggplot2` cheatsheet and documentation and spend some time googling if there's something particular you would like to do. Chances are, if you can imagine it, there's a way to do it with `ggplot2`.

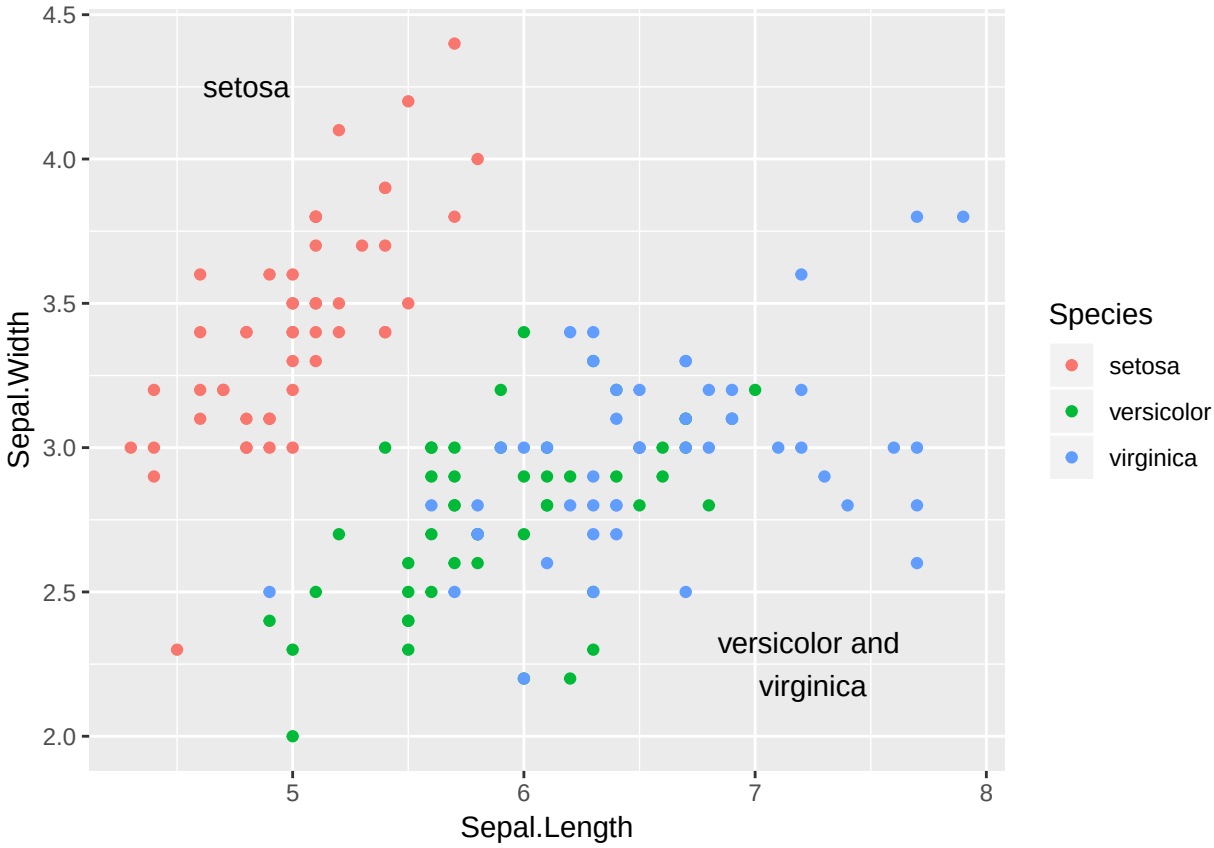
3. Annotations

Another thing that can be useful to add to a plot is annotations. These are marks (usually text, lines, or shapes) that you manually create on the plot.

Text annotations

To add text annotations, use `annotate()`.

```
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + annotate("text", x = 4.8,
    y = 4.25, label = "setosa") + annotate("text", x = 7.25,
    y = 2.25, label = "versicolor and \nvirginica")
```

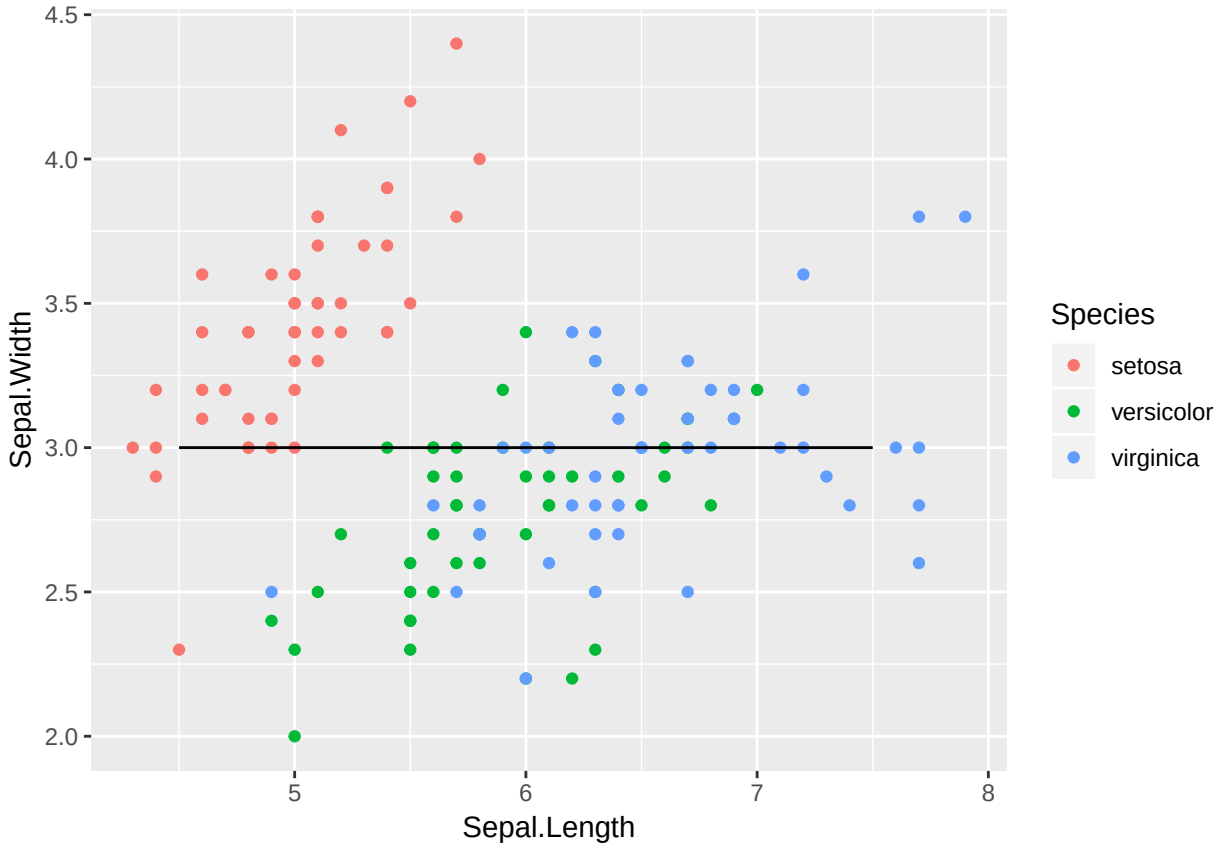



```
# 'text' tells it that you are creating a text annotation x
# and y tell it the coordinates the annotation should be
# placed at label tells it what the text annotation should
# say
```

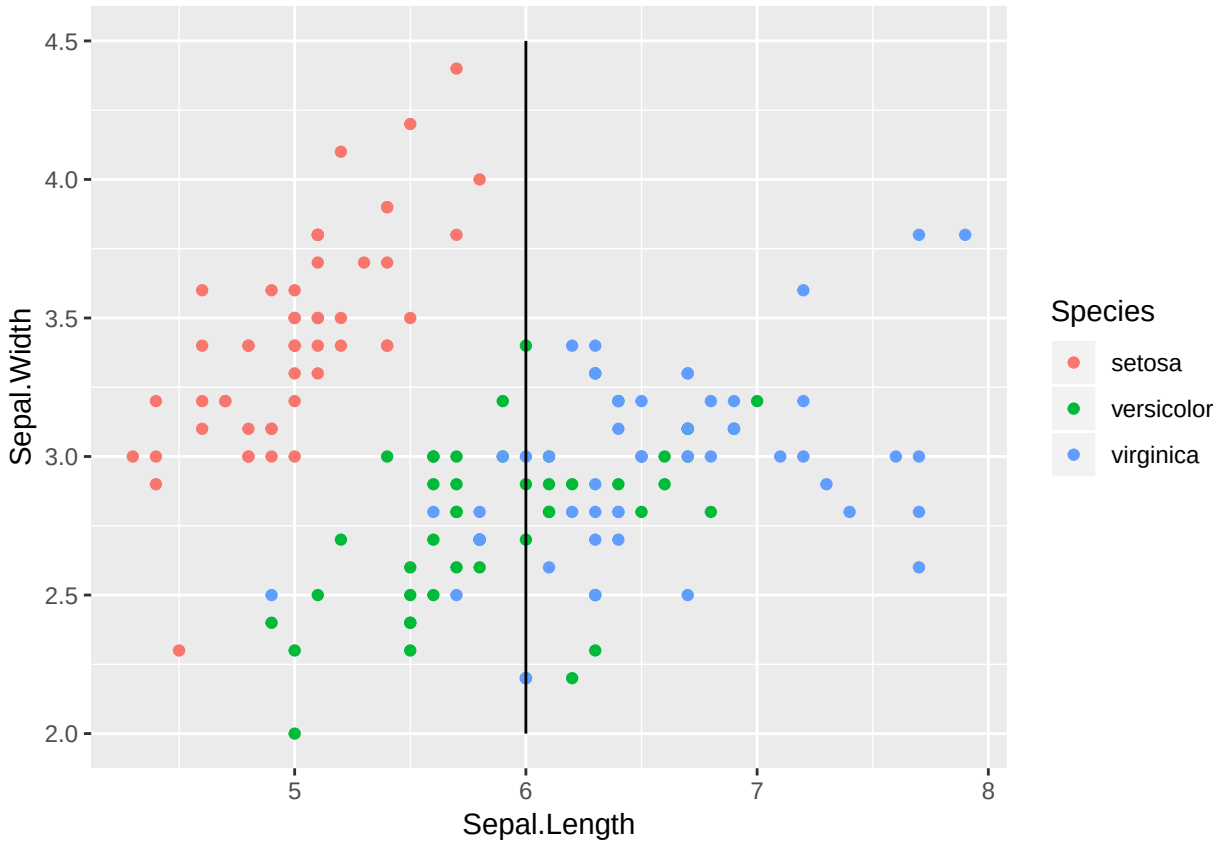
Lines and shapes

To make a line or rectangle on a plot, you can continue to use `annotate()`:

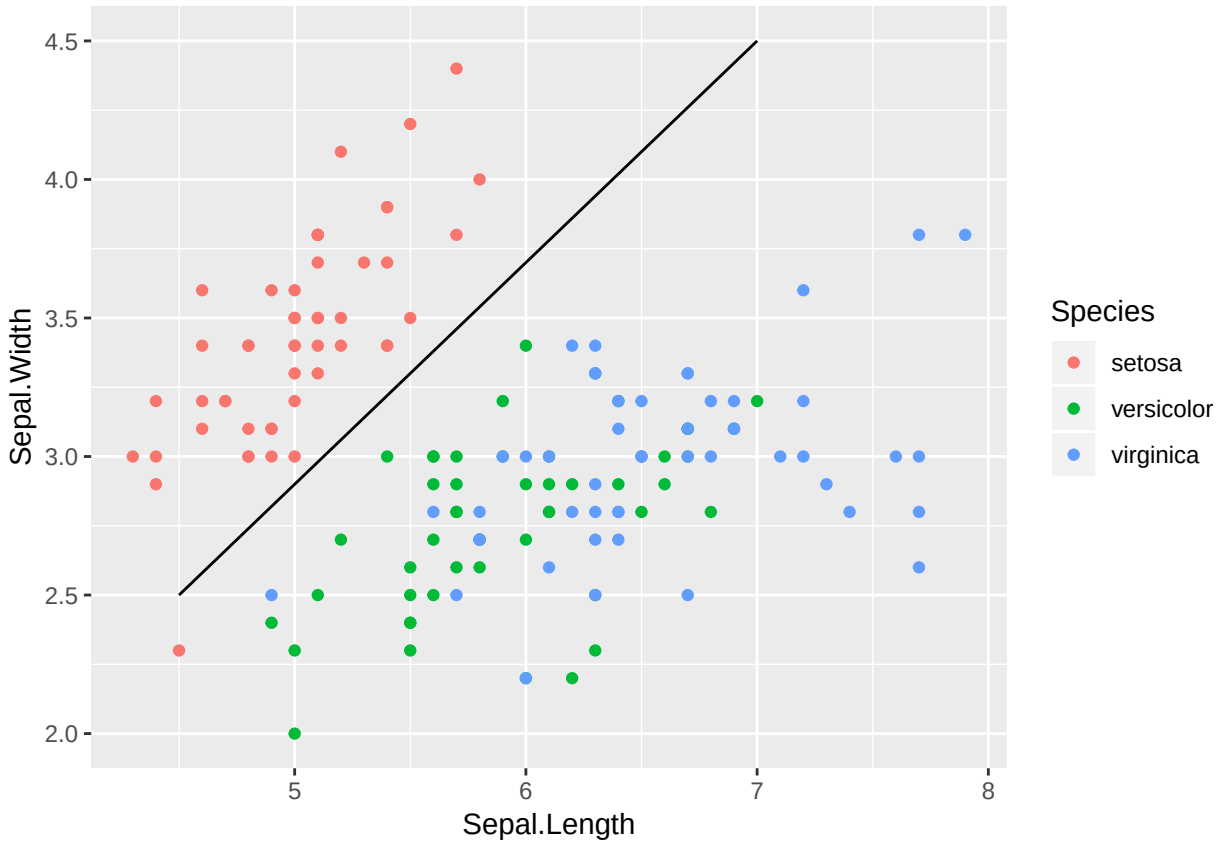
```
# random horizontal line
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + annotate("segment",
    x = 4.5, xend = 7.5, y = 3, yend = 3)
```



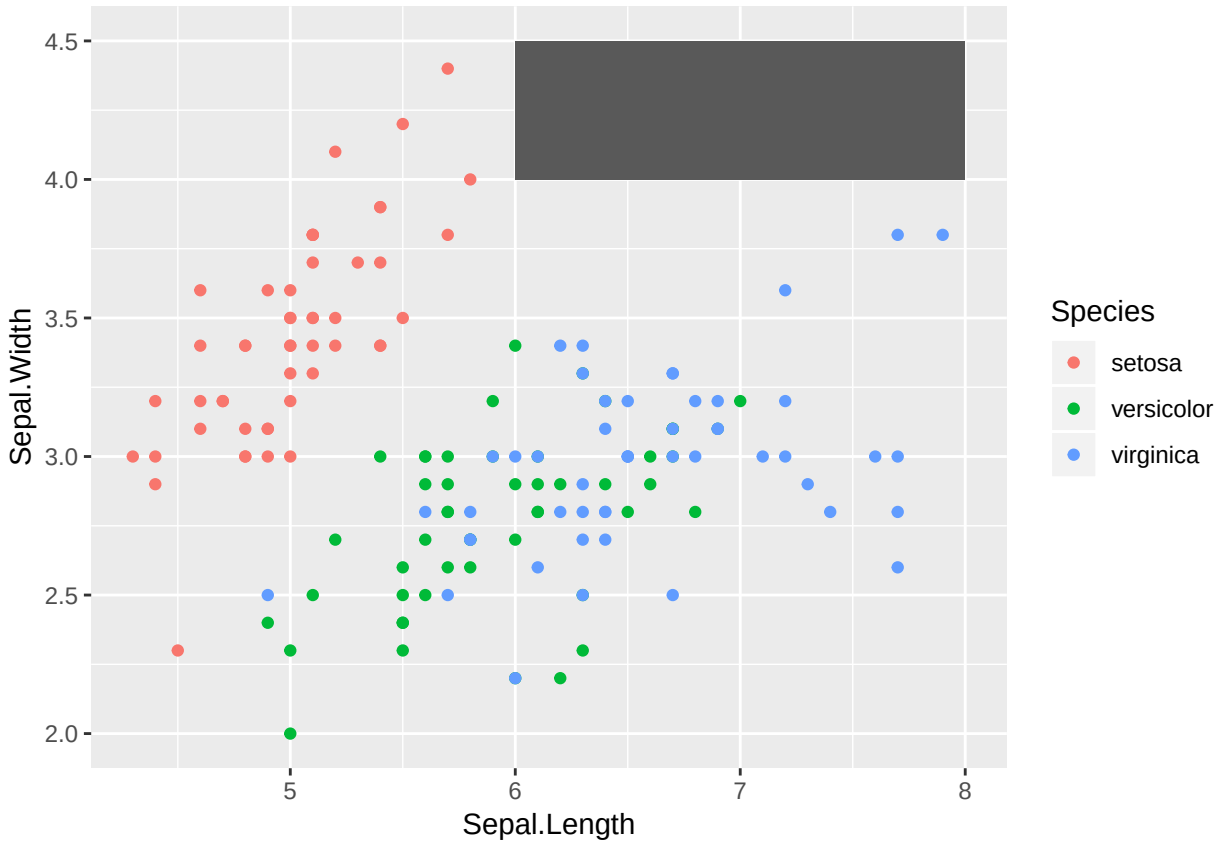

```
# random vertical line
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + annotate("segment",
    x = 6, xend = 6, y = 2, yend = 4.5)
```



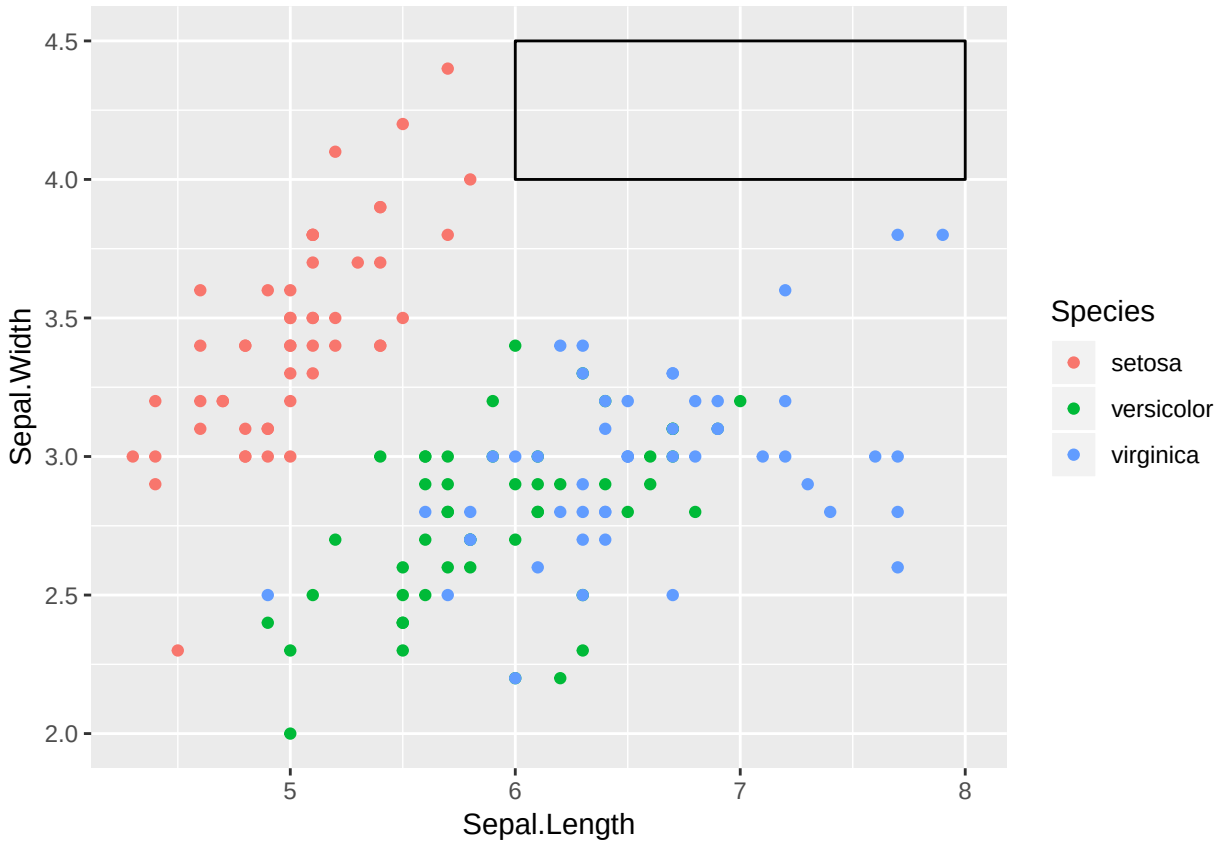

```
# random diagonal line
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + annotate("segment",
    x = 4.5, xend = 7, y = 2.5, yend = 4.5)
```




```
# random rectangle
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + annotate("rect", xmin = 6,
    xmax = 8, ymin = 4, ymax = 4.5)
```

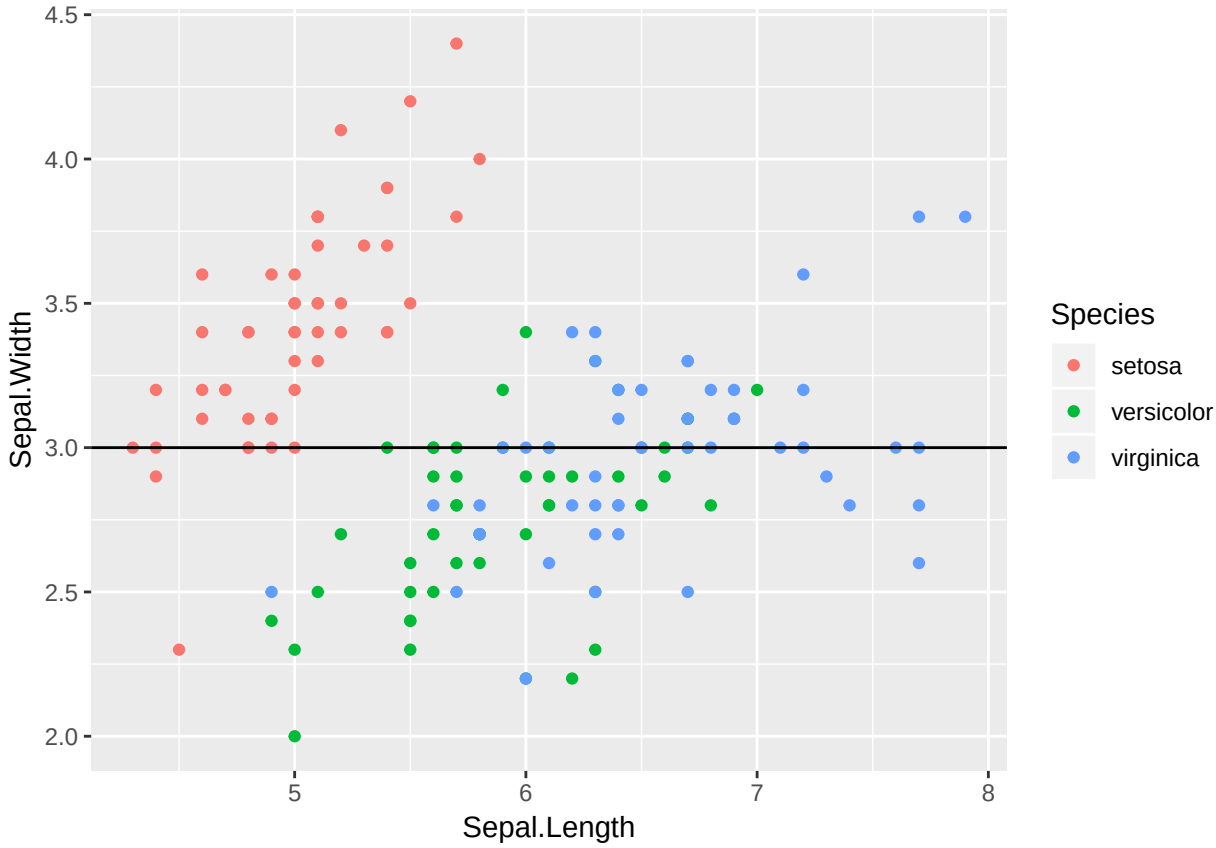



```
# for a rectangle, use xmin/xmax/ymin/ymax instead of
# x/xend/y/yend

# can control color, fill, etc. of annotations
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + annotate("rect", xmin = 6,
    xmax = 8, ymin = 4, ymax = 4.5, color = "black", fill = NA)
```

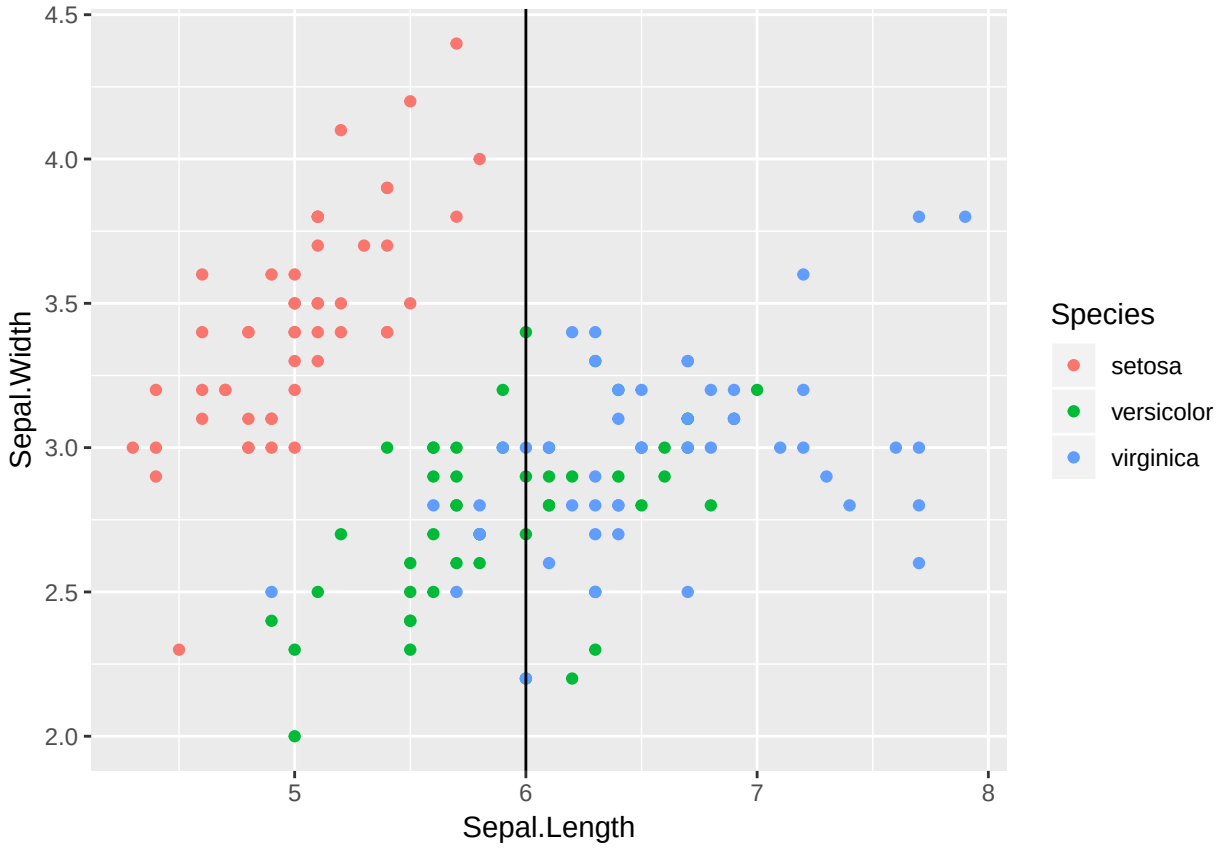


There are also specialized functions you can use for making horizontal lines (`geom_hline()`), vertical lines (`geom_vline()`), and rectangles (`geom_rect()`).

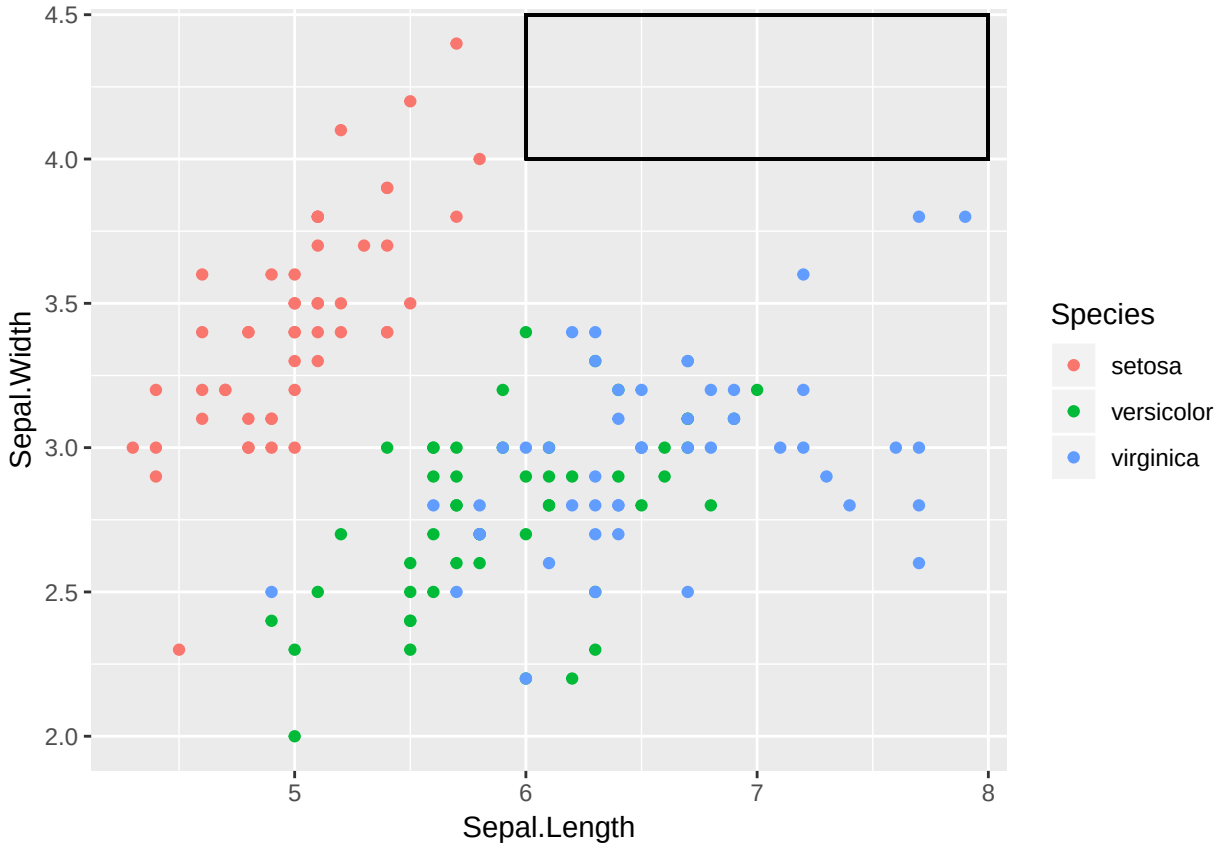
```
# horizontal line
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + geom_hline(yintercept = 3)
```




```
# vertical line
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + geom_vline(xintercept = 6)
```



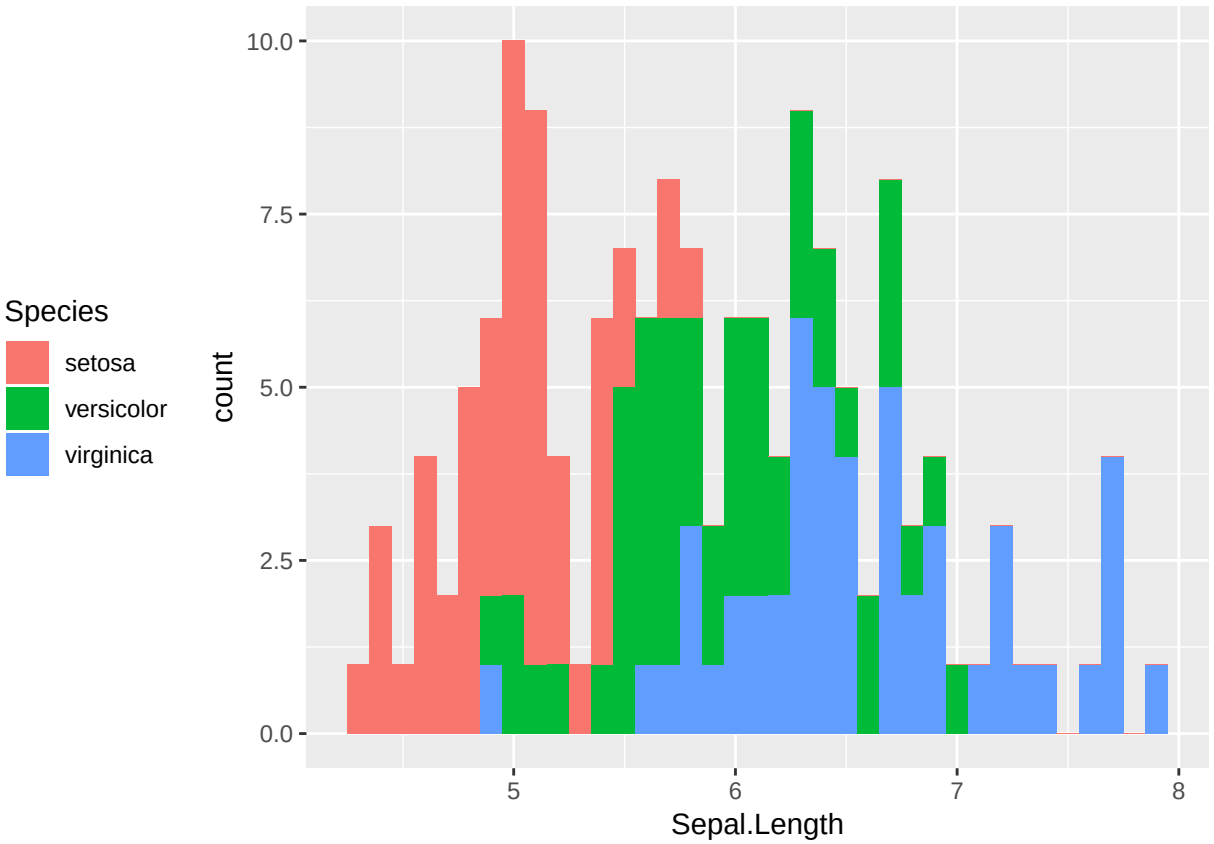

```
# rectangle
ggplot(data = iris) + geom_point(mapping = aes(x = Sepal.Length,
    y = Sepal.Width, color = Species)) + geom_rect(xmin = 6,
    xmax = 8, ymin = 4, ymax = 4.5, color = "black", fill = NA)
```



4. Legends

ggplot2 automatically creates legends for you when you use non-xy aesthetics. By default, the legend is positioned to the right of the plot, but this can be adjusted manually:

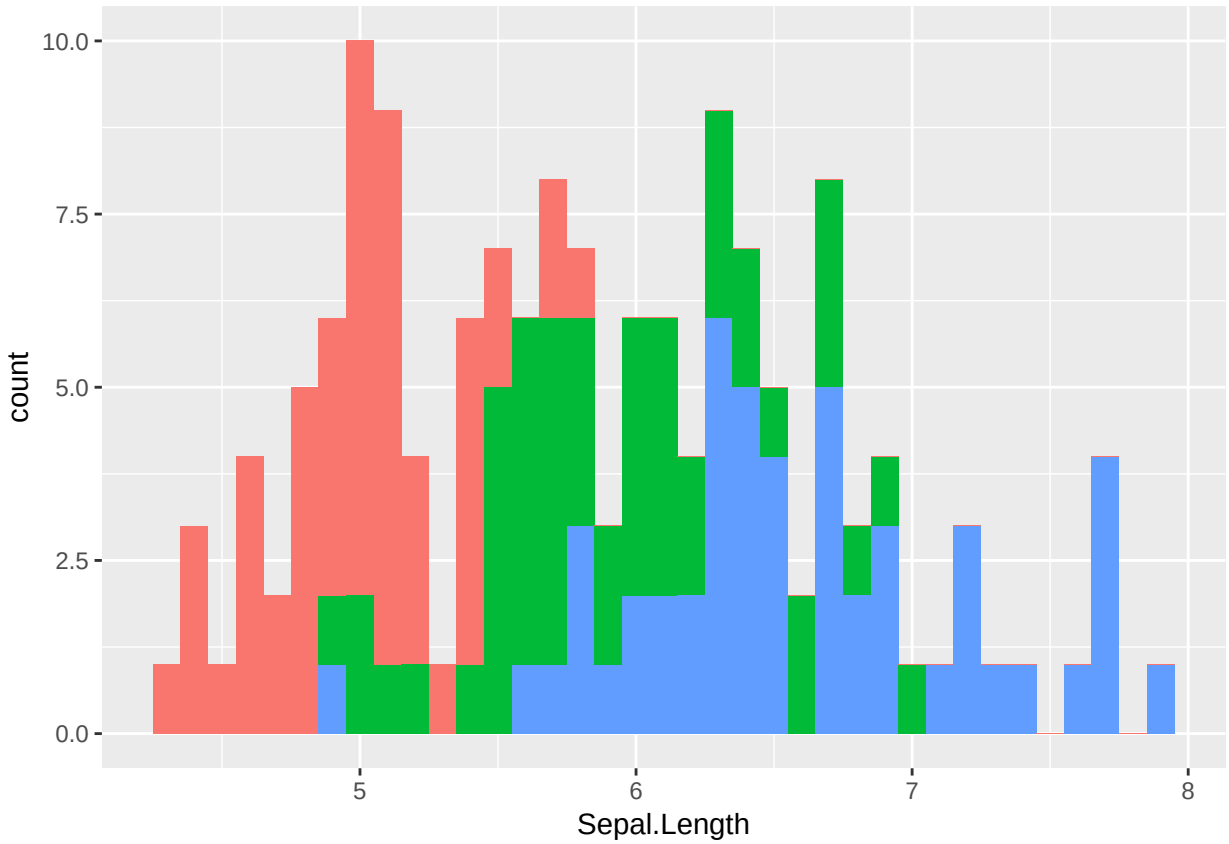
```
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1) + theme(legend.position = "left")
```




```
# and likewise for top and bottom
```

And you can use the same method to get rid of the legend entirely, if you like:

```
ggplot(data = iris) + geom_histogram(mapping = aes(x = Sepal.Length,
    fill = Species), binwidth = 0.1) + theme(legend.position = "none")
```



Resources

`ggplot2` function reference page: https://ggplot2.tidyverse.org/reference/

Chapters 3 and 28 of the book *R for Data Science* (Wickham and Grolemund 2017), which give tons more examples of how to use `ggplot2`: https://r4ds.had.co.nz/data-visualisation.html and https://r4ds.had.co.nz/graphics-for-communication.html

References

Wickham, H. 2010. A Layered Grammar of Graphics. Journal of Computational and Graphical Statistics **19**:3-28

Wickham, H., and G. Grolemund. 2017. R for Data Science: Import, Tidy, Transform, Visualize, and Model Data. O'Reilly Media, Inc.