IB R-workshop: ggplot2

Keith Bouma-Gregson

January 2018

Visualizations with R

We will go over visualizations with the functions available with base R, as well as the ggplot package. In general base R is good for quick visualizations, but for most public purposes, PPTs and publications, you will want to use a visualization package. A common package is ggplot, so that is what we will teach you here.

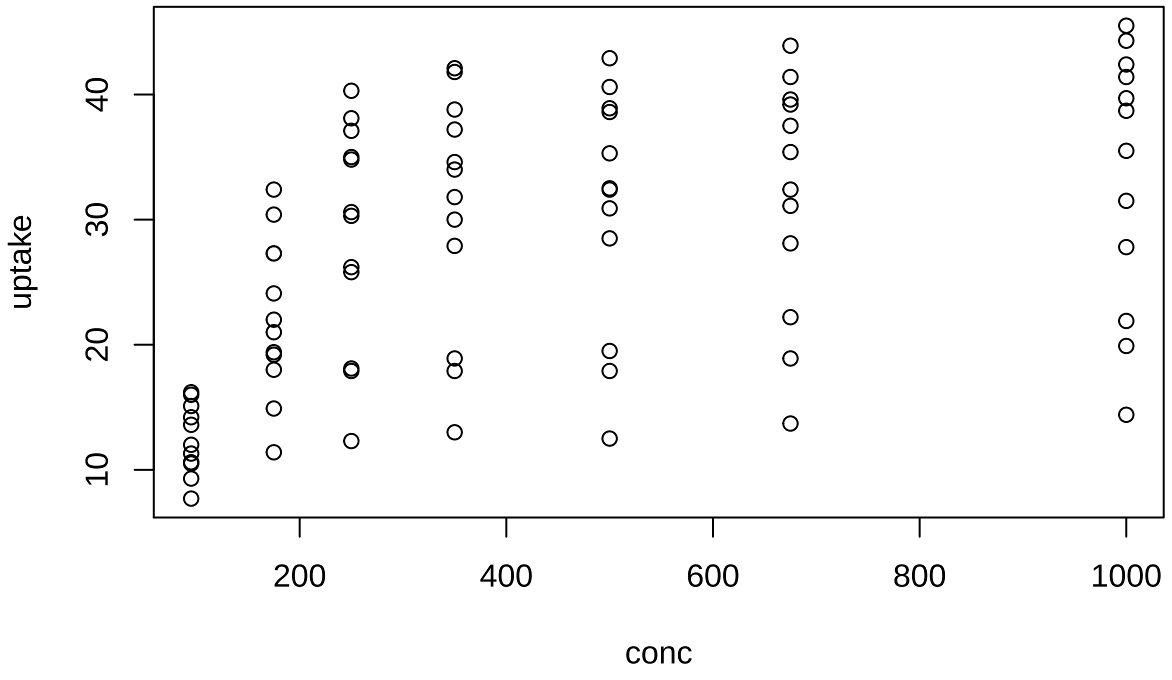
Plotting in base R

The basic plotting function is `plot()`, which takes a formula `plot(y ~ x)` or explicit definitions of the x and y axis `plot(x= , y= )`. The inputs can be vectors or columns in a data frame. If columns in a data frame, then the data frame must also be defined, `plot(y ~ x, data= ?)`. Let's make some plots with `CO2` data frame, which shows different CO2 uptake rates for plants.

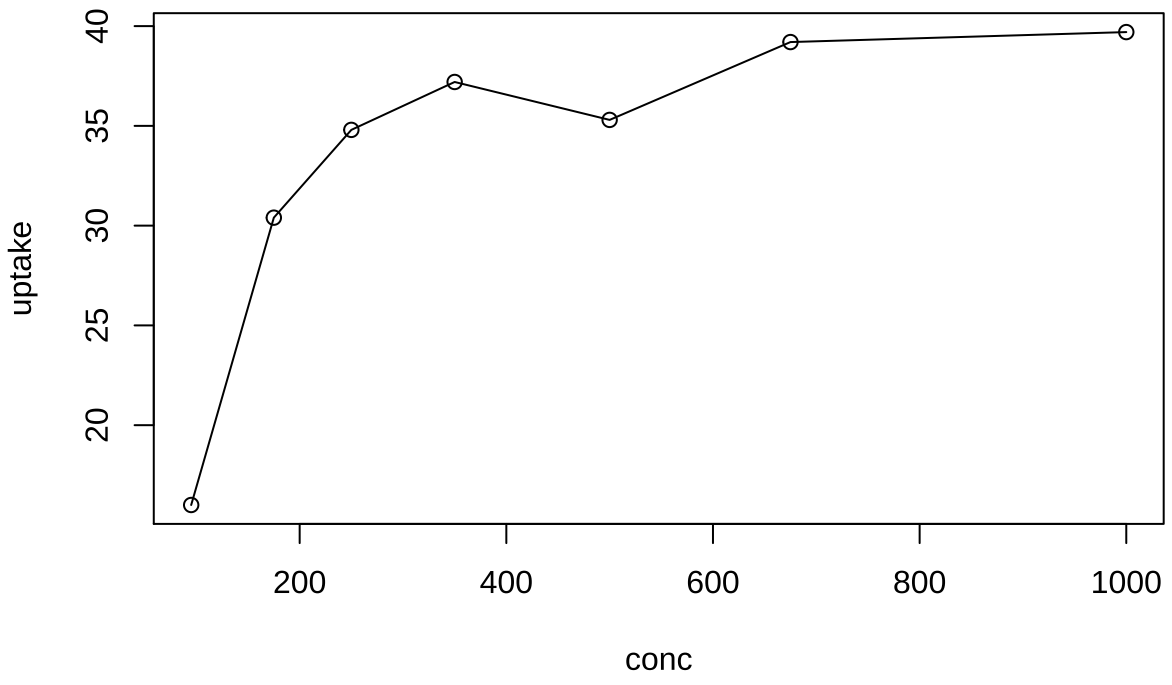
```
head(CO2)
```

```
##   Plant   Type  Treatment conc uptake
## 1   Qn1 Quebec nonchilled   95   16.0
## 2   Qn1 Quebec nonchilled  175   30.4
## 3   Qn1 Quebec nonchilled  250   34.8
## 4   Qn1 Quebec nonchilled  350   37.2
## 5   Qn1 Quebec nonchilled  500   35.3
## 6   Qn1 Quebec nonchilled  675   39.2
```

```
#  Plot of uptake vs. CO2 concentration
plot(uptake ~ conc, data= CO2)
```



```r
#  Add lines to the points
plot(uptake ~ conc, data= CO2[which(CO2$Plant == "Qn1"), ])
lines(uptake ~ conc, data= CO2[which(CO2$Plant == "Qn1"), ])
```



---

ggplot: the grammar of graphics

```r
# install.packages(ggplot2)
library(ggplot2)
```

ggplot uses the "grammar of graphics" (Wilkinson 2005) to build visualizations. This grammar describes what a statistical graphic is: mapping data onto geometric objects (points, lines, etc.) with particular aesthetic characteristics (colors, shapes, size, etc.). Ggplot2 works like a language, but instead of words you, string together different layers to build your plot. Just like any language ggplot has a grammar to make it interpretable.

Before we can plot the data we need to go over a few terms.

Geoms These are the actual marks that are placed on the plot
- points, lines, bars, boxplots, etc.

Aesthetics These define the visual qualities (i.e. aesthetics) of the geoms on the plot
- position, size, color, fill, linetype.

We will start with the `Orange` dataset.

```r
head(Orange)
```

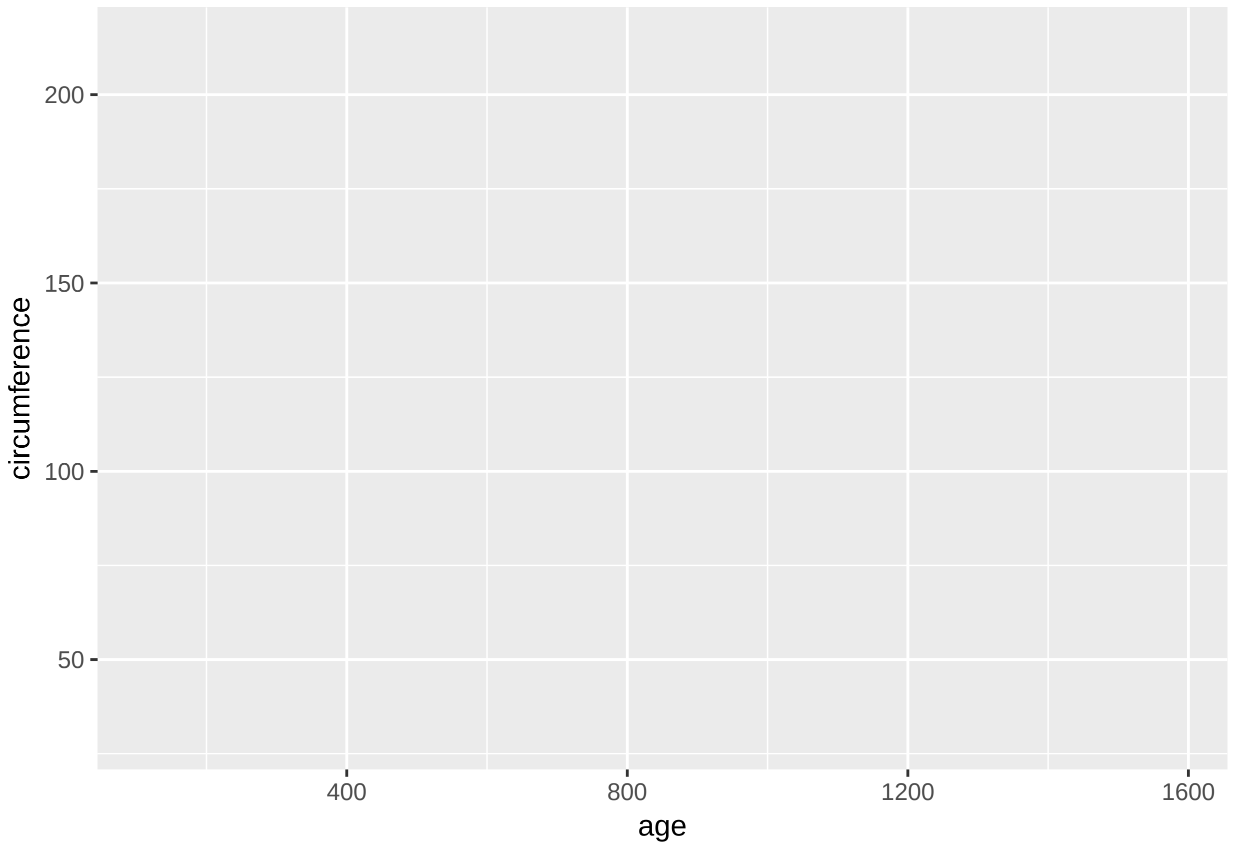
Building a ggplot

We will use the `Orange` data set in base R. Which shows the circumference of 5 orange trees at different ages.

Let's plot the circumference of trees against their age.
We start by using `ggplot()` to define the data and how it will be mapped
`aes()` stands for aesthetics

```
ggplot(data = Orange, mapping= aes(x = age, y = circumference))
```

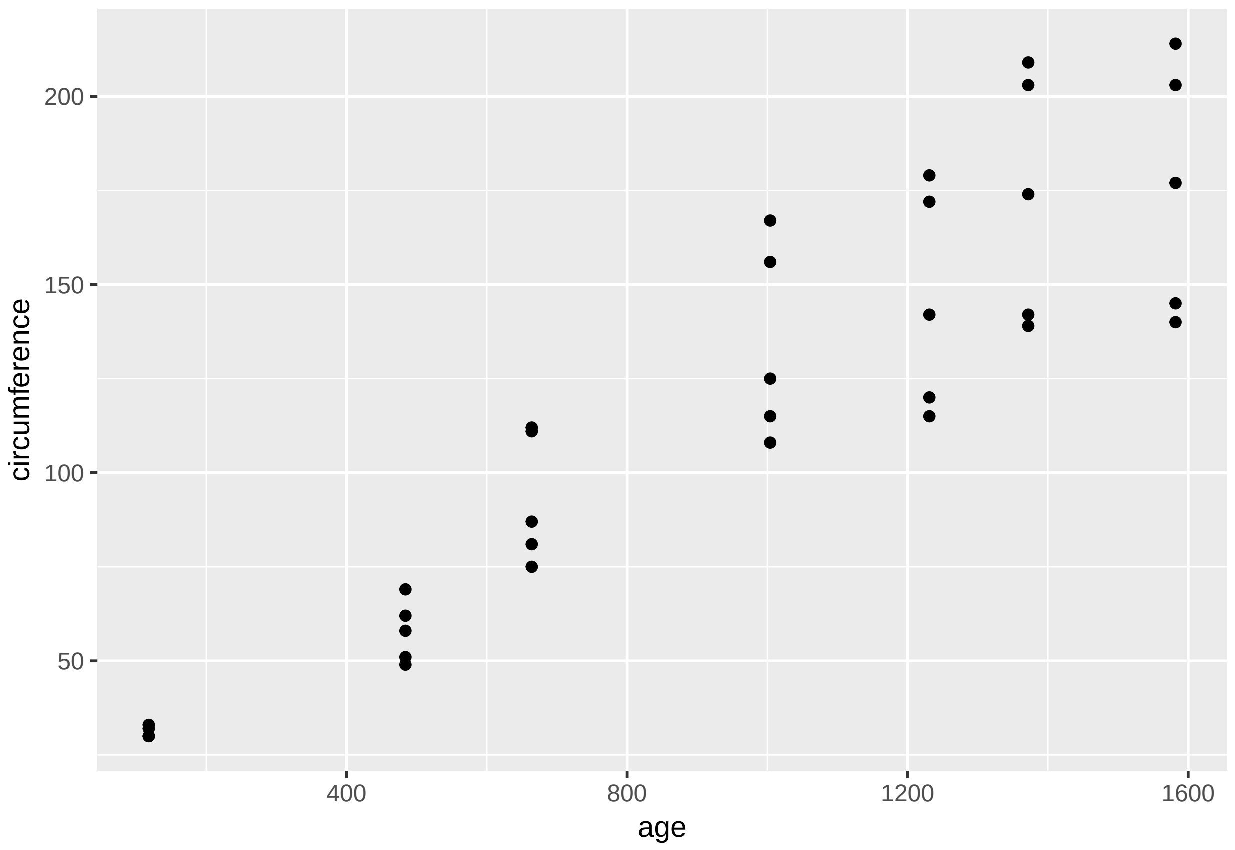
By defining our data and positions, we have created the foundational plot upon which we can now add our layers. You can see below there are no points, because we have not specified any geoms.



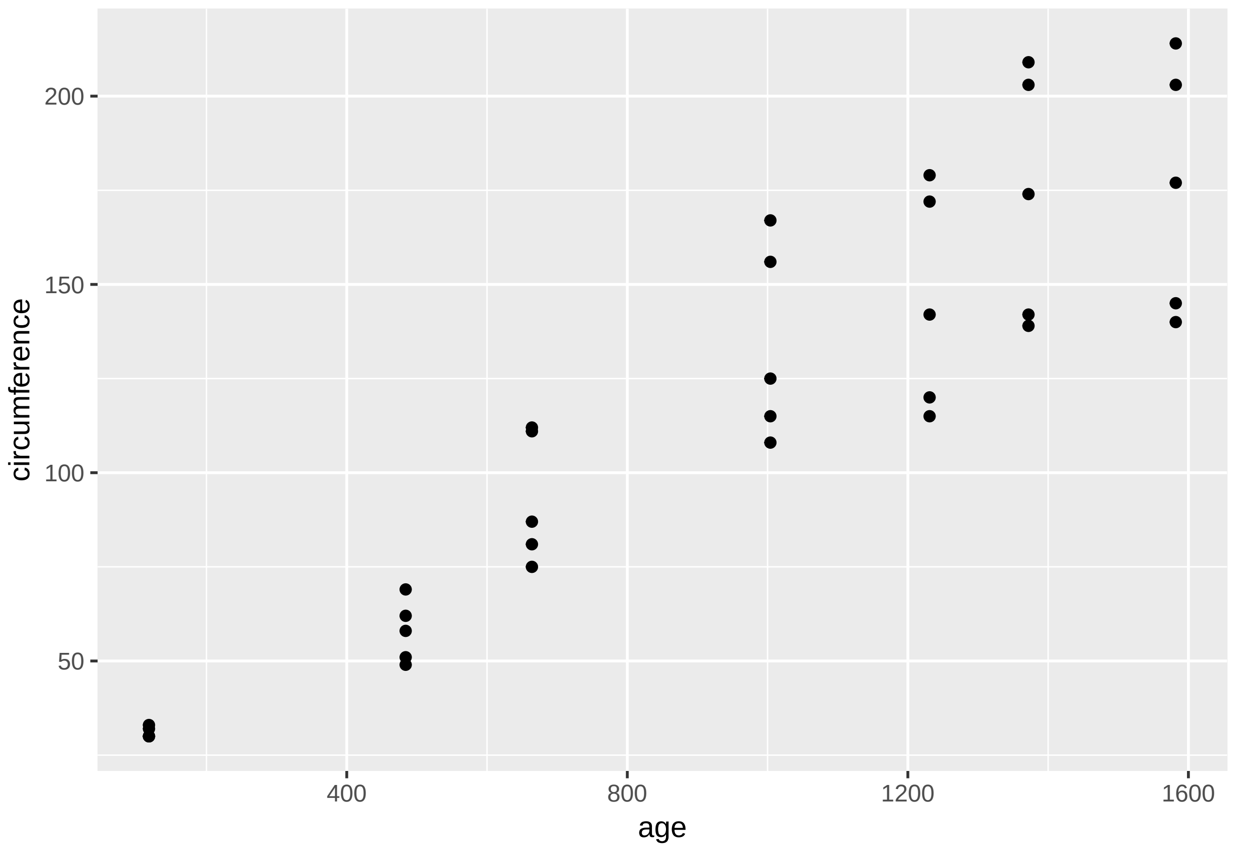
We add additional layers using the `+` sign.
We need to add a geom, so that we can place some actual marks on our plot.

- Geoms all take the form `geom_xxx`
  - `geom_point`
  - `geom_line`
  - `geom_bar`
  - `geom_boxplot`
  - etc.

```
## All data and mapping set globally
ggplot(data = Orange, mapping= aes(x = age, y = circumference)) +
  geom_point()
```



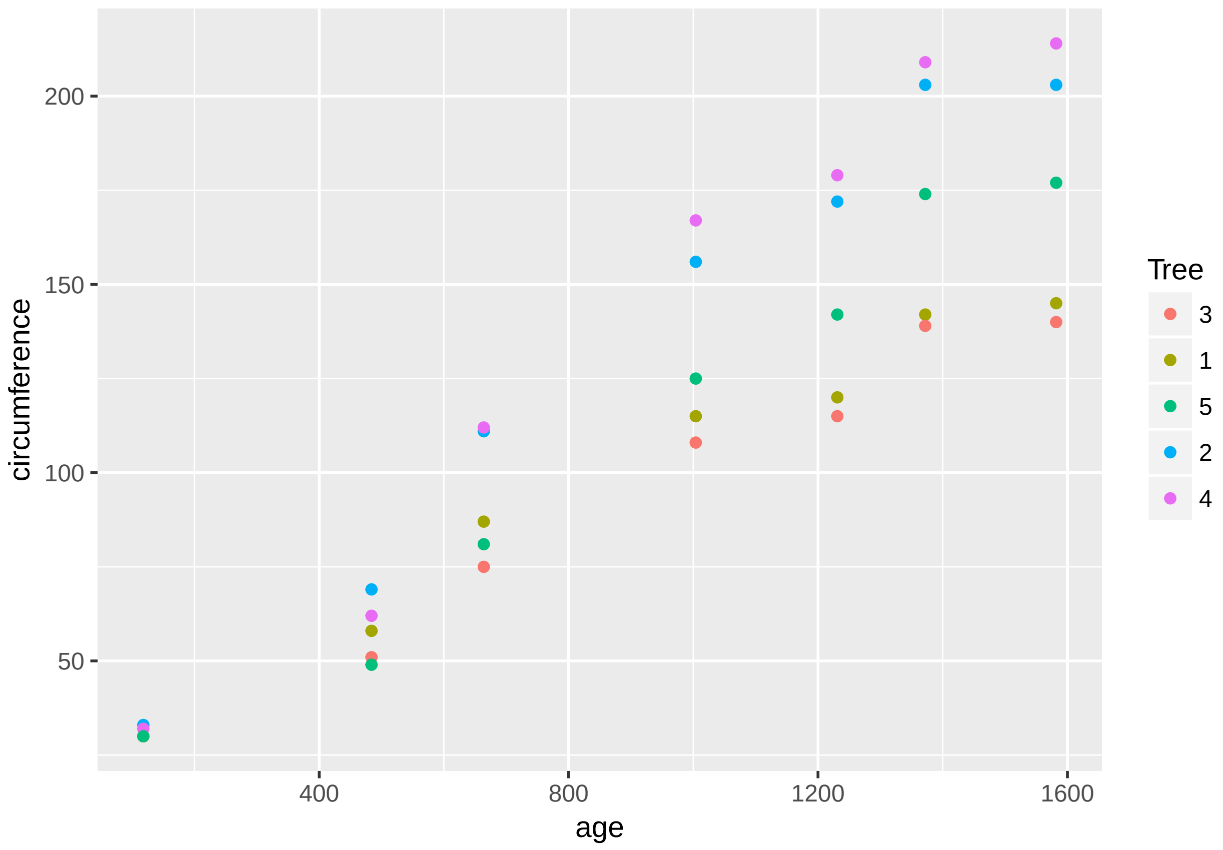
```
## Moving mapping and data to individual geom layer
ggplot(data= Orange) +
geom_point(mapping= aes(x = age, y = circumference))
```



```
## Or
# ggplot() +
# geom_point(data= Orange, mapping= aes(x = age, y = circumference))
```

But we have no idea which points correspond to which trees. Let's assign each tree a different color in a new plot.

```
ggplot(data = Orange, aes(x = age, y = circumference)) +
  geom_point(aes(color= Tree))
```



Notice how I can have my layers be on a separates line? Placing each layer on a separate line can increase the readibility of your code

Variables it is possible to use variables to save time while making a ggplot. I can save my base `ggplot()` function to a variable, so that I do not have to keep typing it as I make different plots.

```
orange.plot <- ggplot(data = Orange, aes(x = age, y = circumference))
```

EXERCISE What happens when you run these commands?

```
# 1
orange.plot +
  geom_point(color= Tree)

# 2
orange.plot +
  geom_point(color= "purple")
```

Anytime you specify an aesthetic characteristic outside the `aes()` command, ggplot will apply that quality to all marks in that particular geom. The size of points is something that we usually want to apply globally, but sometimes we might want it to relate to the variables.

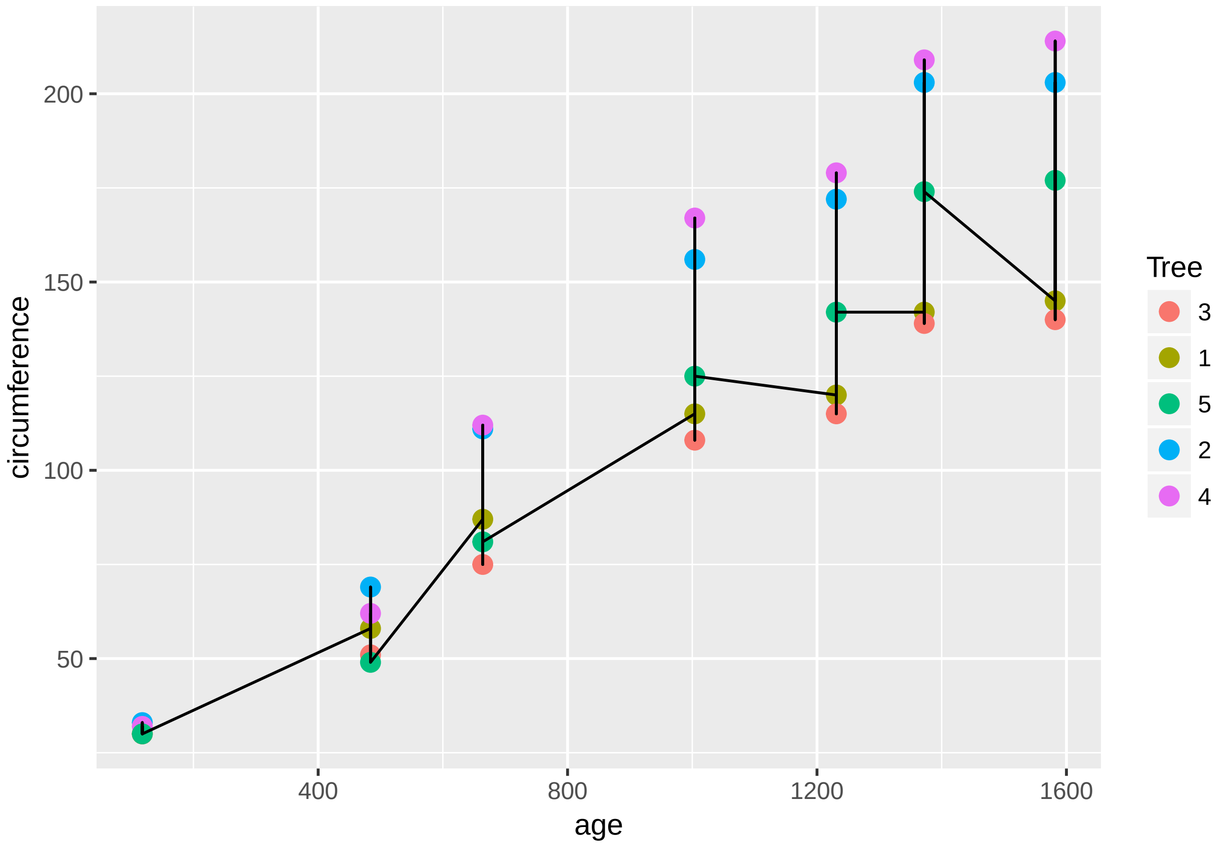
EXERCISE What does the `size` command do in each of these plots?

```
# Remember orange.plot <- ggplot(data = Orange, aes(x = age, y = circumference))

# 1
orange.plot +
  geom_point(aes(color= Tree, size= Tree))
```

```r
# 2
orange.plot +
  geom_point(aes(color= Tree), size= 3)

# 3
orange.plot +
  geom_point(aes(color= Tree, size= 3))
```

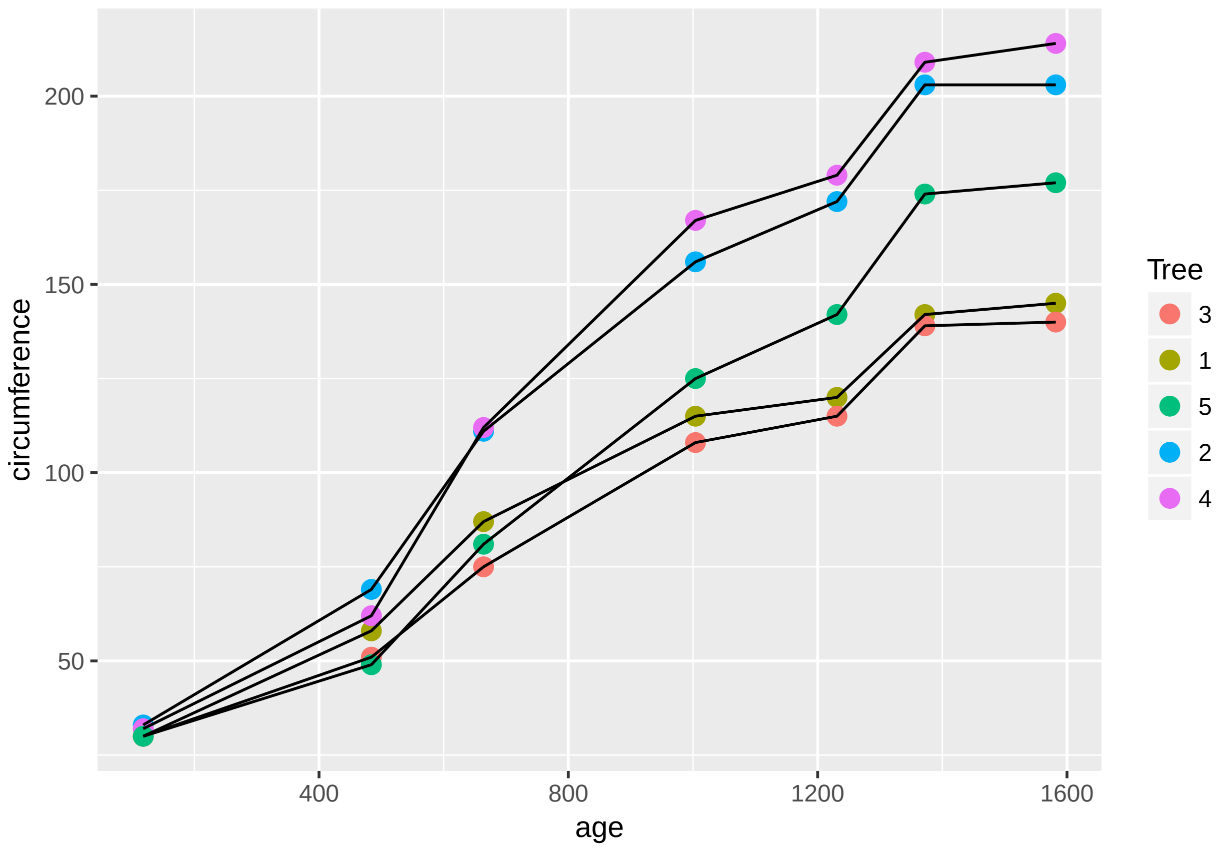
We can continue to build our plot with more layers. Let's add lines to connect the points that belong to each tree. Let's make the size of our points so they are easier to see too.

```r
orange.plot +
  geom_point(aes(color= Tree), size= 3) +
  geom_line()
```



Oops, that did not work, ggplot connected all the points with one line, because we never told ggplot we wanted the points grouped together so that only points belonging to the same tree would be connected. To do this we use the `group` command.

```r
orange.plot.group <- ggplot(data = Orange, aes(x = age,
                                               y = circumference,
                                               group= Tree))

orange.plot.group +
  geom_point(aes(color= Tree), size= 3) +
  geom_line()
```



Much better What happens if you switch the order of the `geom_point()` and `geom_line()` layers? How could we get the color of the lines to match the `Tree`?

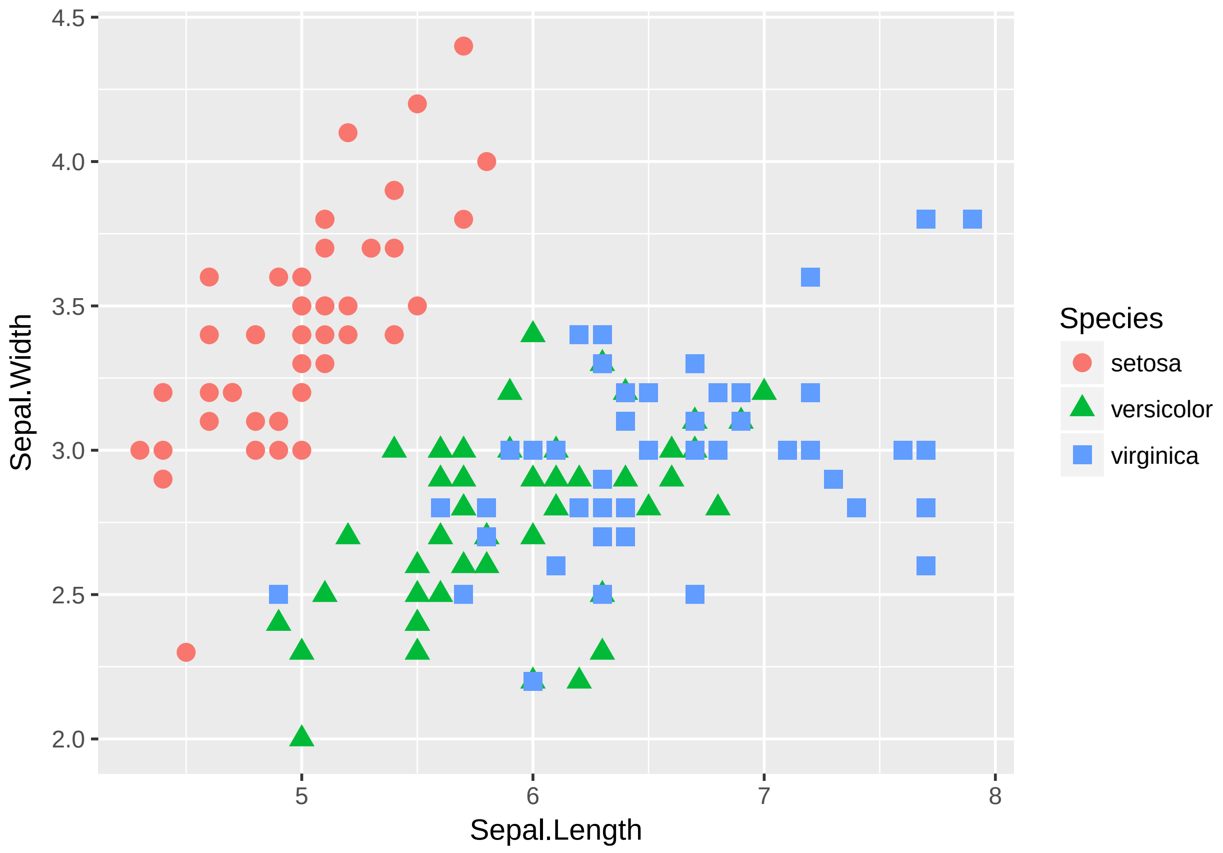
EXERCISE: Building your own plot

Look at the the 'iris' dataset. This is a dataset of leaf measurements for 3 different species of iris.

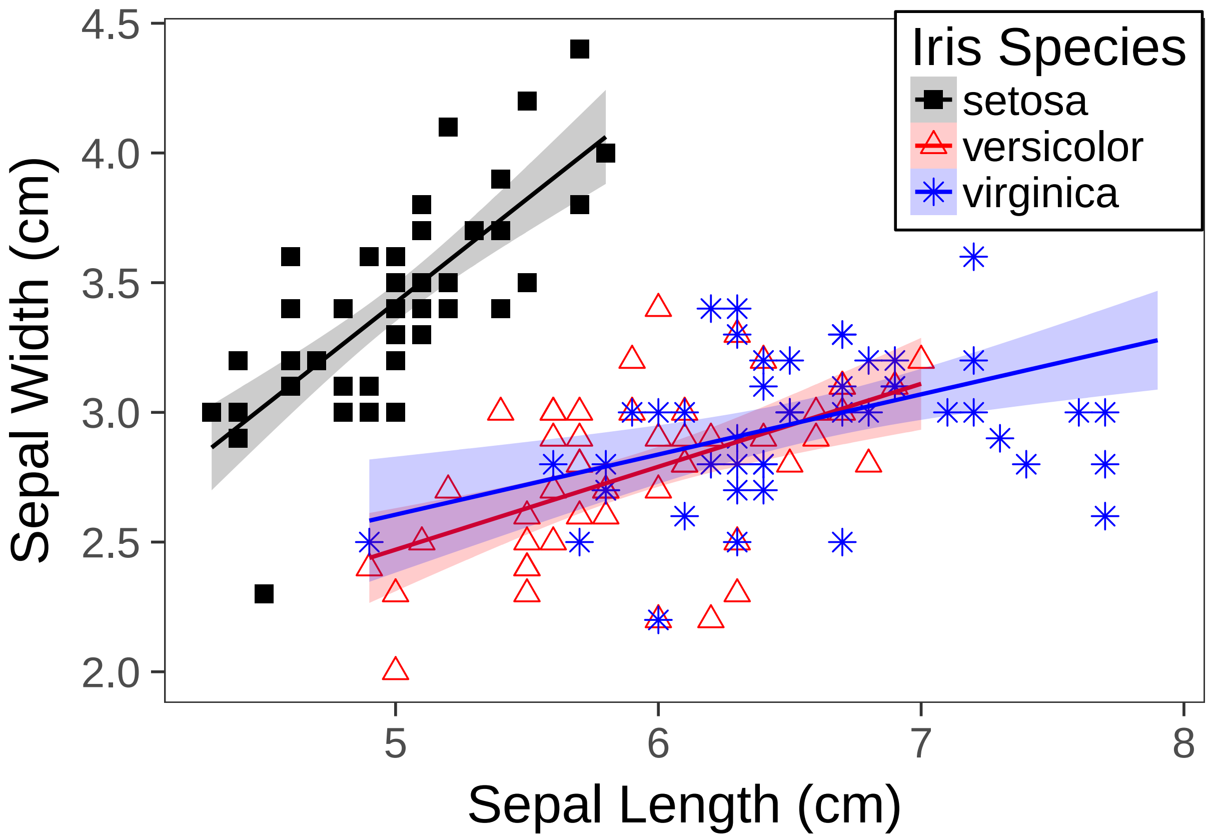
Build a scatter plot (`geom_point`) of y= Sepal.Width and x= Sepal.Length with each species being a different `color` and `shape`, also change the `size` of the points to make the plot easier to read. (Hint: `shape` is another aesthetic quality and functions just like `color` and `size`).

```
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
```



Now we will see how to use the same data and make a more complicated plot. Here will be our end result, so lets work through how to build this figure.



Adding trendlines

`geom_smooth` will plot linear regression lines on your data, as well as non-linear smoothing lines. This is a helpful way to aid your eye in seeing the patterns in your data, think of these as qualitative. They should not be a substitute for a formal statistical test.

- Arguments for `geom_smooth(aes(), method= ?, se= ?, alpha= ?)`
  - method: smoothing method (we will use `"lm"`= linear model)
  - se: should standard errors be plotted (logical T/F)
  - alpha: controls the transparency of a filled region (0-1)
  - size: similar to 'geom_point' size controls the width of the line

Task 1

- Use `geom_smooth()` to add trend lines to our previous plot to create the following plot.
- The color of the line and the std. error ribbon should match the points
- Decrease the transparency, so it is similar to the following plot

```
ggplot(data= iris, aes(x= Sepal.Length, y= Sepal.Width)) +
 geom_point(aes(color= Species, shape= Species), size= 3) +
 geom_smooth(aes(color= Species, fill= Species, group= Species),
             size= 0.75,
             method= "lm",
             alpha= 0.2)
```

Customization using scales

This is our first introduction to ggplots use of scales. Scales exemplify the grammar of graphics. You will see a set of rules and conventions consistently applied to a variety of commands and functions.
Scales associate values in your data to values within a given aesthetic characteristic (e.g. `shape`, `color`, etc.). The scaling process also converts your data into values that are informative to the computer. For example, x and y axis location need to be converted into pixel values. You do not need to worry about the meaning of these converted values, rather 'scales' are you to specify how you want your data to be mapped to your aesthetic attributes. Your data are most likely either continuous or discrete, and aesthetics are also continuous or discrete.

Scales have a constant naming convention:

first `scale_`,
then the name of the aesthetic `scale_color`,
then type of data `scale_color_discrete()`
This applies to all aesthetics: shape, color, fill, size, x, y, linetype

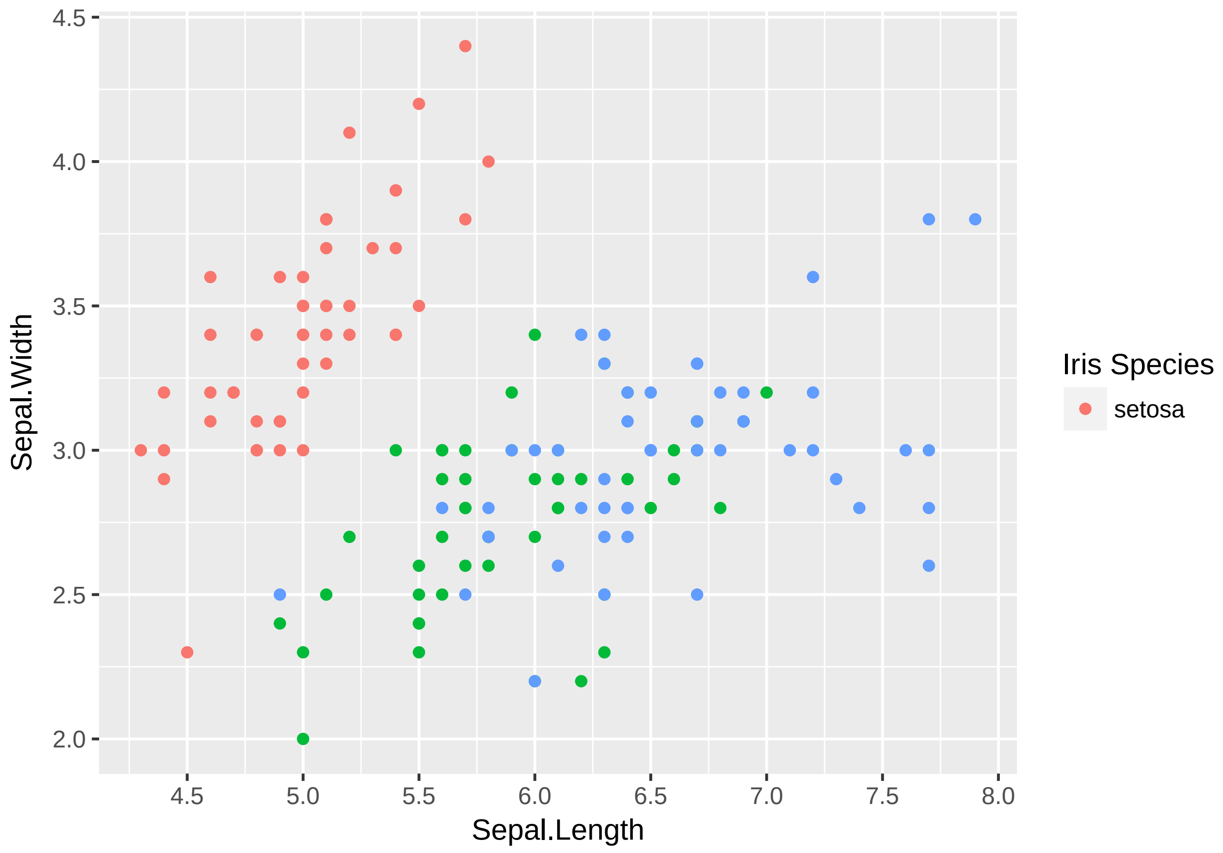
There are a few arguments common to all scales:

- **name**: the label that will appear on the axis or legend
- **limits**: set the domain of the scale (what appears on the plot)
  - continuous scales take a max and min value
  - discrete scales take a character vector
  - if you set limits smaller than the data range, then values outside the limits will be dropped
- **breaks**: what will appear on the axis or legend
  - how tick marks will appear on an axis or how aesthetics will be segmented
- **labels**: the label that appears at each breakpoint

Legends display the aesthetics specified by the scales.

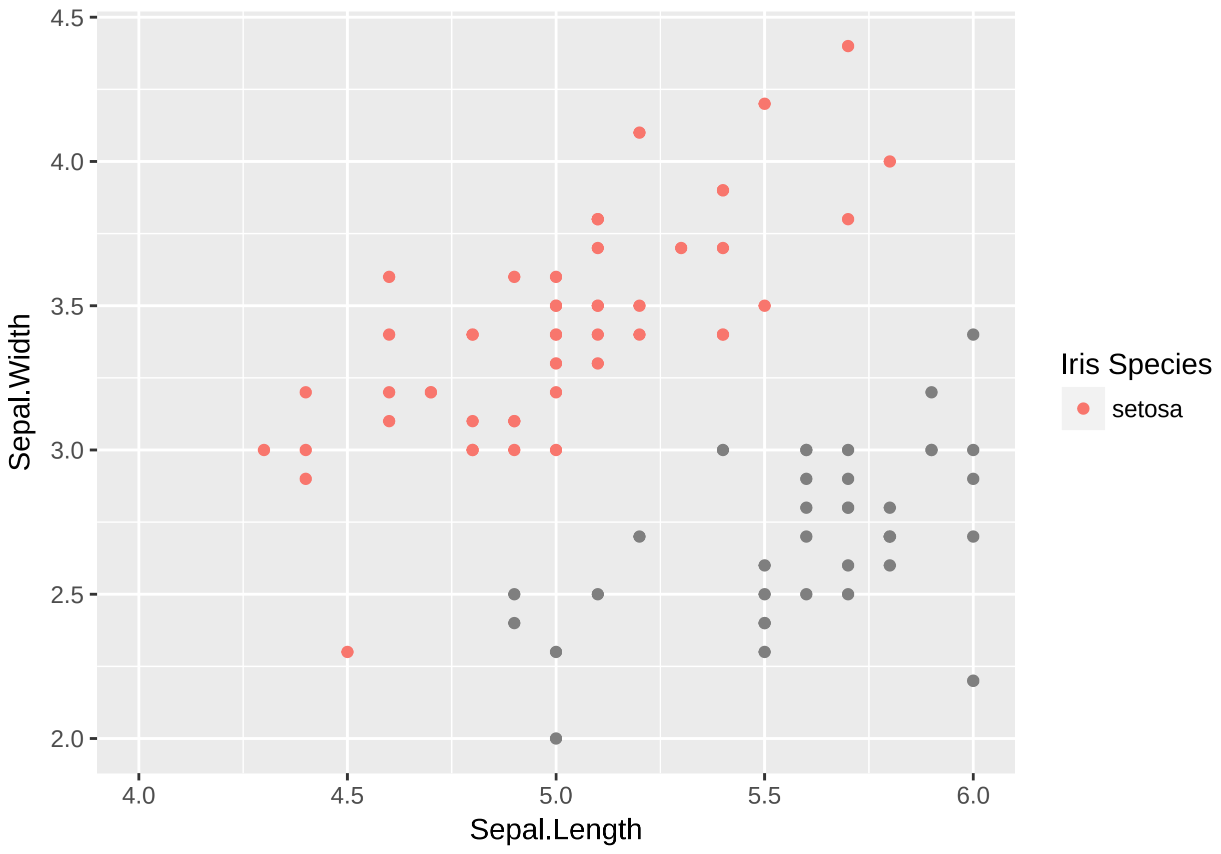
- legends are also referred to as guides
- Legends consist of:
  - Title: at the top
  - Key: display of aesthetic (line, point, etc.)
  - Key label: variable that aesthetic represents
- ggplot will automatically create the legend
  - if the same variables are used with more than one aesthetic, then ggplot will merge the legends together
- the `breaks` argument in `scale_xx_xx()` commands determine how to draw the keys
- All merged legends must have the same `name` in the `scale_xx_xx()` command

```r
# Breaks= what  appears on the axis or legend
ggplot(data= iris, aes(x= Sepal.Length, y= Sepal.Width)) +
  geom_point(aes(color= Species)) +
  scale_x_continuous(breaks= seq(4,8, by= 0.5)) +
  scale_color_hue(name= "Iris Species", breaks= "setosa")
```



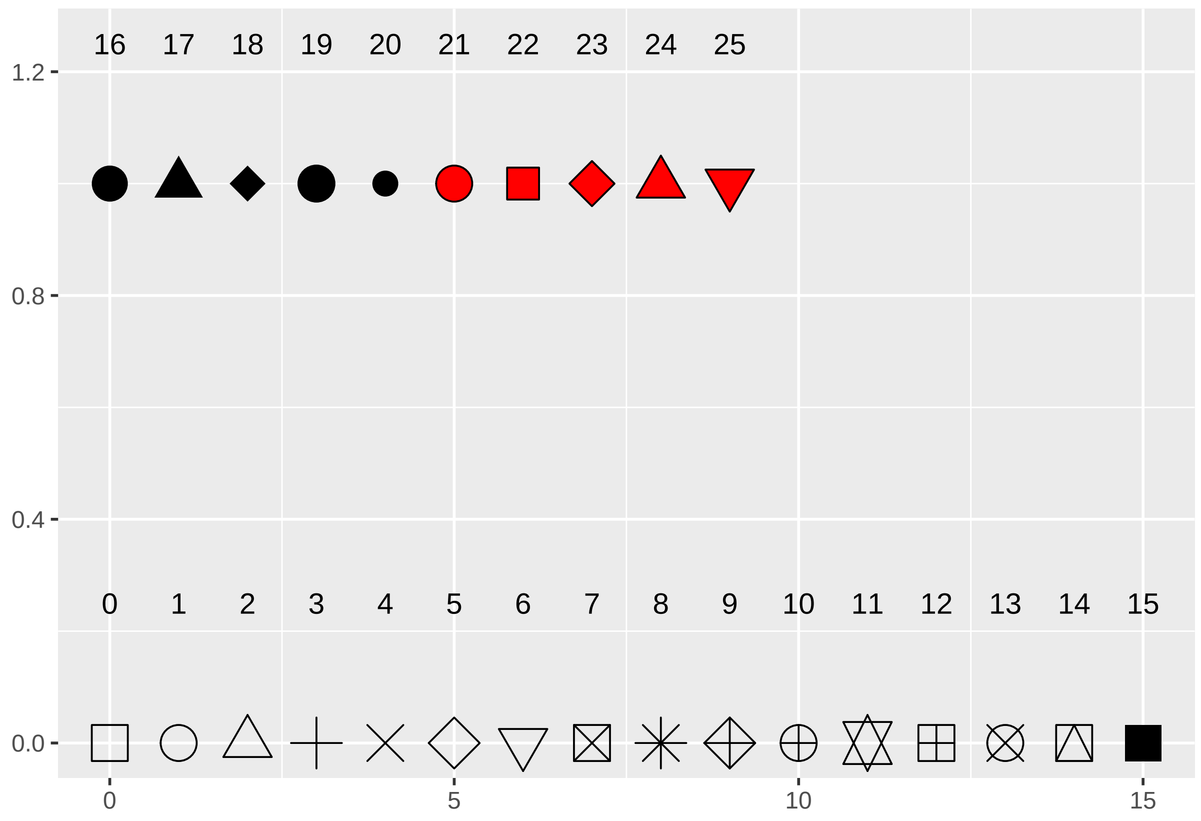
```
# Limits= what appears on the plot
ggplot(data= iris, aes(x= Sepal.Length, y= Sepal.Width)) +
  geom_point(aes(color= Species)) +
  scale_x_continuous(limits= c(4, 6)) +
  scale_color_hue(name= "Iris Species", limits= "setosa")
```

```
## Warning: Removed 61 rows containing missing values (geom_point).
```



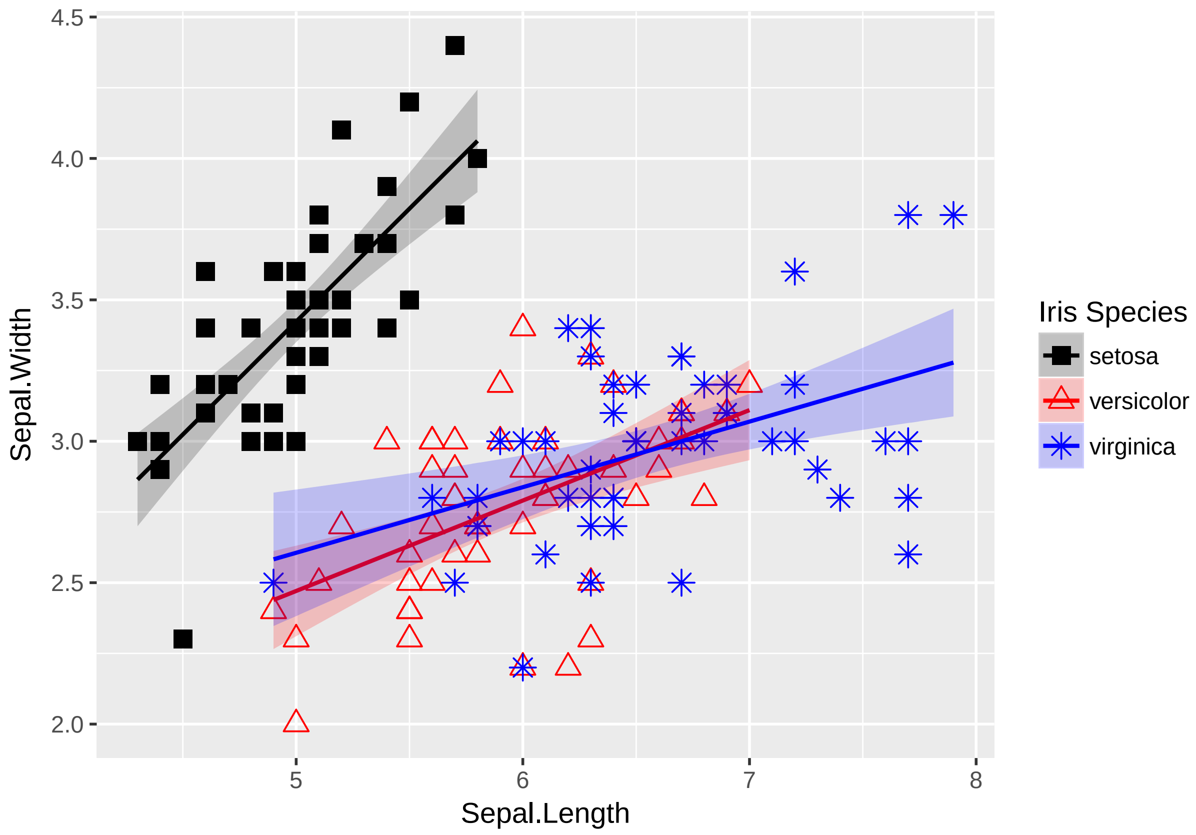
There is also an option to make manual scales `scale_color_manual`, which allows you to create your own discrete scale using the `values` argument.
`scale_color_manual(values= c("white", "orange", "purple"))`
Shapes can also be customized using `scale_shape_manual`

- type `?pch` to get the help document for shapes
- ggplot will map up to 6 shapes, any more need to be coded manually



Task 2

- Use `scale_xxx_manual` commands to adjust the colors to match the black, red, and blue of the following plot. The point, lines, and std. error ribbon should all be the same colors.
- Change the shapes of the points
- Use the 'names' argument to change the legend title to also match the plot



Variables and ggplot

Making variables for your ggplots can save time and help you organize your plots.

- It is common for people to assign their `ggplot()` command to a variable, then add layers onto that variable.
- You can also assign information that will be used on multiple layers in your plots to a variable, then when you need to change that information. You can change your variable once, rather than changing the information on each layer.
- If a layer has many arguments and is becoming long, you can split it up over multiple lines. This will increase the readibility of your code. Hitting cmd-i, will auto align the lines.

```
iris.plot <- ggplot(data= iris, aes(x= Sepal.Length, y= Sepal.Width))
species.colors <- c("black", "red", "blue")
species.shapes <- c(15, 2, 8)
legend.title <- "Iris Species"

iris.plot +
  geom_point(aes(color= Species, shape= Species), size= 3) +
  geom_smooth(aes(color= Species, fill= Species, group= Species),
              size= 0.75,  method= "lm", alpha= 0.2) +
  scale_color_manual(name= "Iris Species", values= c("black", "red", "blue")) +
  scale_fill_manual(name= legend.title, values= species.colors) +
  scale_shape_manual(name= legend.title, values= species.shapes)
```

Task 3
Recode your ggplot with variables to simplify and streamline your ggplot

Customizing axes

Axis components include:

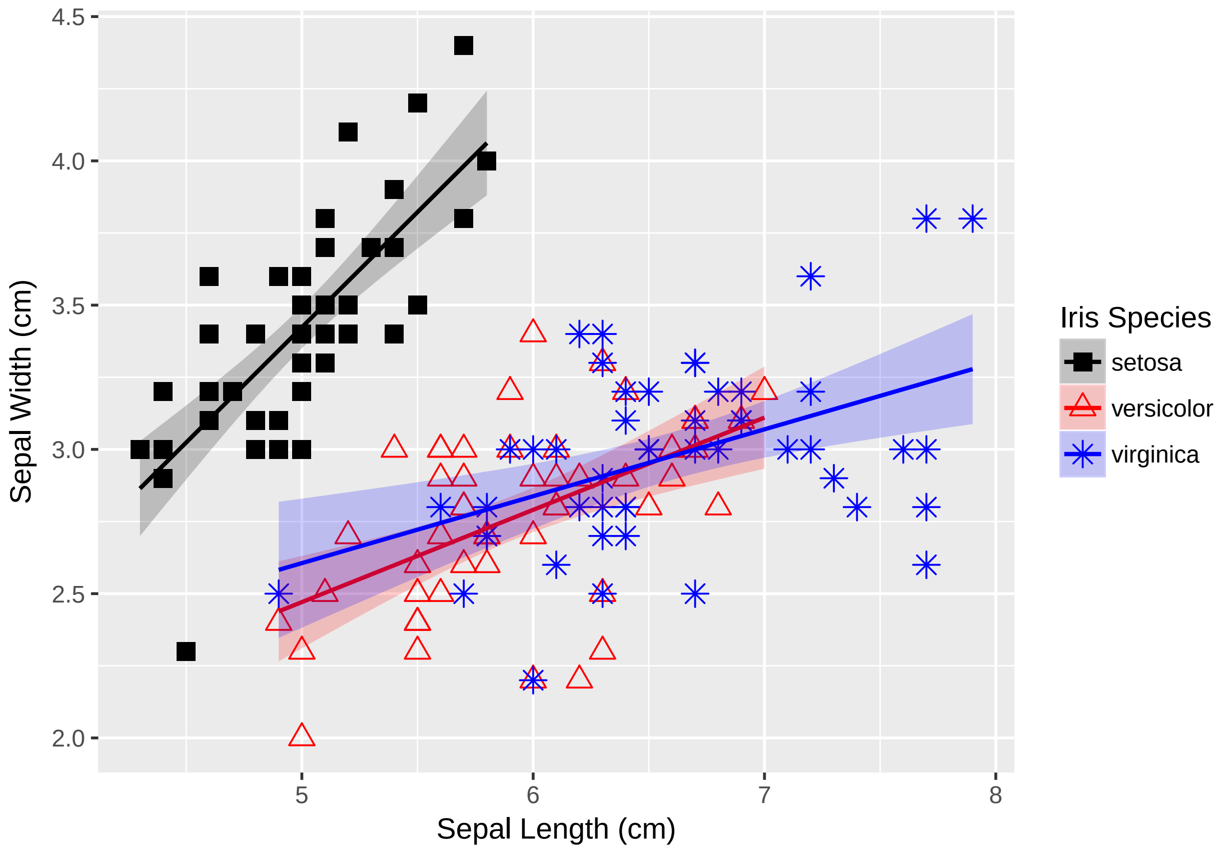
- name: title of the axis
- breaks and minor-breaks (tick marks)
- labels: identity of each tick mark
- limits: range of the axis

Customizing axes is possible using `scales_xx_xx()` commands

`scale_x_continuous(name= "?", breaks= c(?), minor_breaks= c(?), labels= c("?"), limits= c(min, max))`

Other commands are also available if you just want to modify some specifics
`labs(x= "?", y="?")`: modify the axis names
`ylim(min, max)`: modify only the y axis limits
`xlim(min, max)`: modify only the x axis limits

Task 4 Update the axis name to match the plot below



```
labs(x= "Sepal Length (cm)", y= "Sepal Width (cm)") +
```

Themes

To ggplot, themes are all the non-data elements of a plot. And consist of two components:

- **theme elements**: individual attributes of the plot that are independent of the data
  - axis.text, legend.background, panel.grid, etc.
- **element functions**: the commands that enable you to modify different theme elements
  - `element_text()`: labels and headings
  - `element_line()`: lines, tick marks, grid lines

  - `element_rect()`: rectangles, backgrounds, legend keys
  - `element_blank()`: draw nothing, used to disable theme elements

The theme is modified using a nested syntax with the `theme()` function. You can then string together as many theme elements as you desire, each one separated by a comma.
`theme(axis.text = element_text(size= 20), panel.background = element_rect(color= "white"))`
There are also several pre-loaded theme functions that can be called

- `theme_grey()` (the default theme)
- `theme_bw()`: similar to `theme_grey` but with a white background
- `theme_classic()`: no grid lines
- and more

It is possible to call an pre-loaded theme and then alter it further by adding an additional `theme()` layer.

`myplot + theme_bw() + theme(legend.position = "top")`

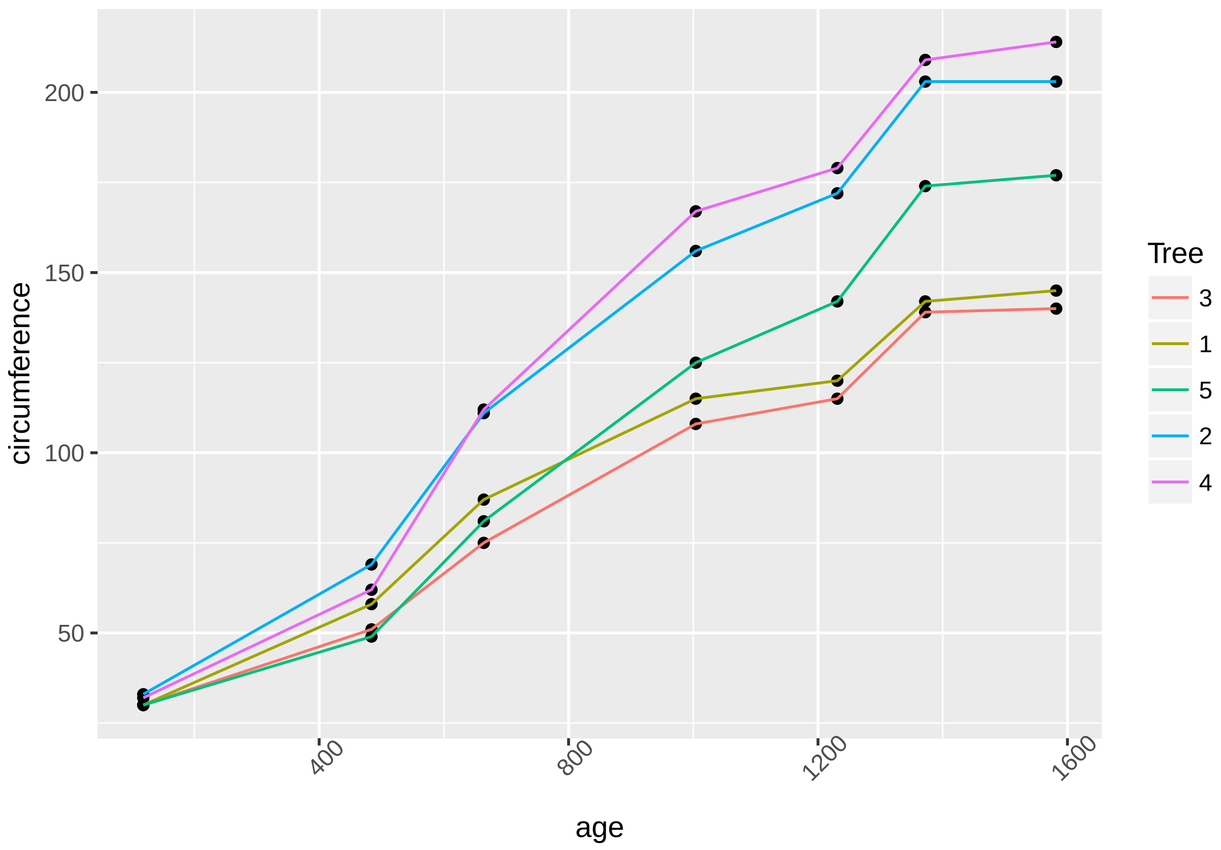
You can also save your own custom themes as their own variable, then add the new variable to your ggplot

```r
# Create custom theme
my.theme <- theme(axis.text = element_text(size= 20),
                  panel.background = element_rect(color= "white"))

# Add new theme to plot
my.plot + geom_point() + my.theme
```

The command `base_size` is a quick convenient way to increase the font size of all text in your plots. This is helpful in preparing figures for PPT presentations
`myplot + theme_bw(base_size= 22)`

There are many theme elements and the ggplot website has excellent documentation for modifying themes: http://docs.ggplot2.org/dev/vignettes/themes.html

```r
orange.plot.group +
  geom_point() +
  geom_line(aes(color= Tree)) +
  theme(axis.text.x = element_text(angle = 45))
```



Task 5 Change the theme from the default theme to match the plot below. After each step below, re-generate the plot so you can see how the theme changes.

1. add `theme_bw(base_size = 20)` to your plot
2. after `theme_bw()` add `theme(panel.grid.major = element_blank(), panel.grid.minor = element_blank())`
3. inside the `theme()` function from step 2, add `legend.position = c(0.85, 0.85), legend.background = element_rect(color = "black")` after the panel.grid elements

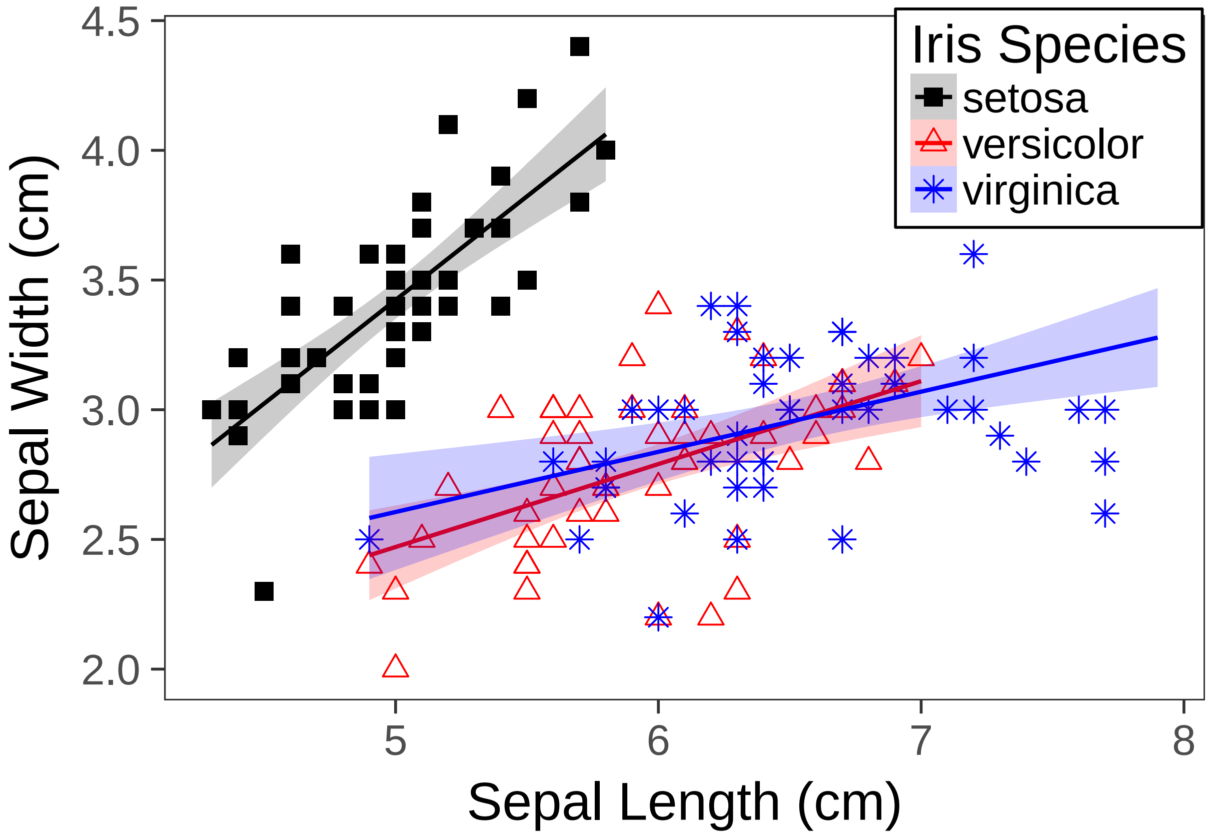
```
## After finishing task 5, your added code should look like this:
 theme_bw(base_size= 20) +
  theme(panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        legend.position = c(0.85, 0.85),
        legend.background = element_rect(color = "black"))
```

The plot looks quite different. There are many different ways to change themes, now that you are familiar with the basic syntax your options are limitless!

We are finished!

Your code and plot should now look something like this:

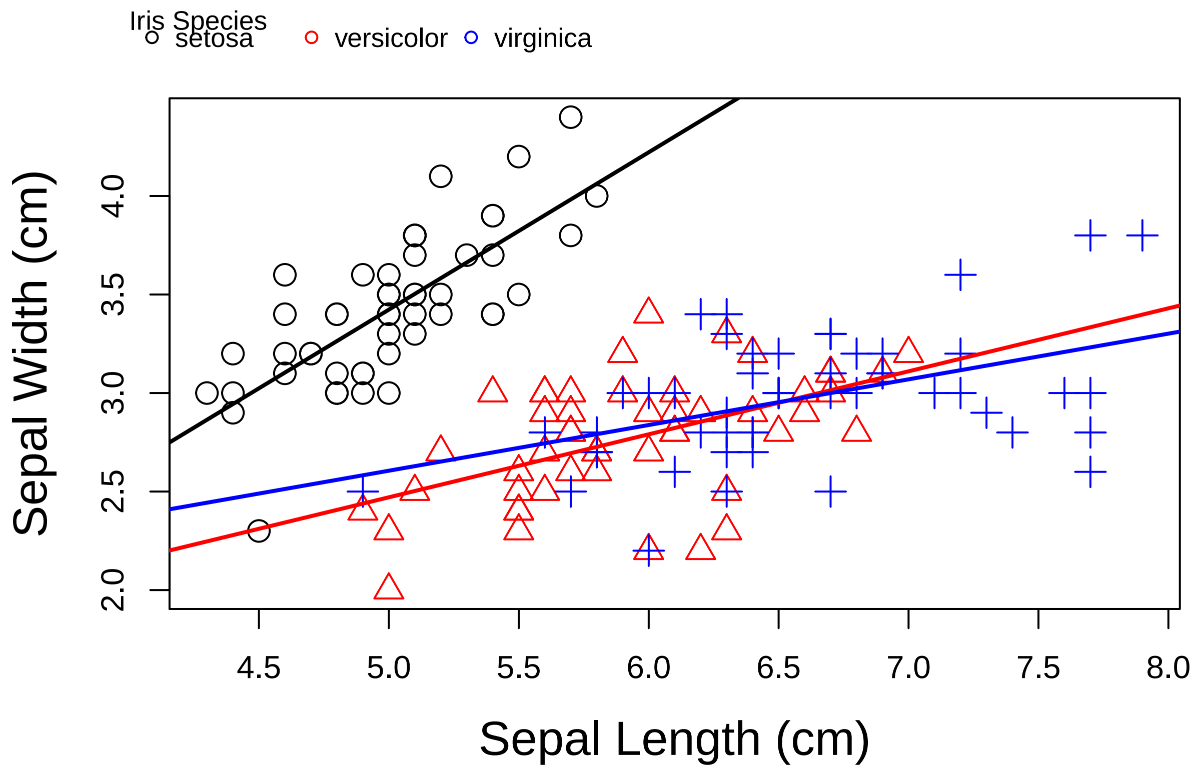
```
# Define data and positionings
iris.plot <- ggplot(data= iris, aes(x= Sepal.Length, y= Sepal.Width))

# Save aesthetic qualities as variables
species.colors <- c("black", "red", "blue")
species.shapes <- c(15, 2, 8)
legend.title <- "Iris Species"


# Generate the plot by combining layers
iris.plot +
  geom_point(aes(color= Species, shape= Species), size= 3) +
  geom_smooth(aes(color= Species, fill= Species, group= Species),
              size= 0.75,  method= "lm", alpha= 0.2) +
  labs(x= "Sepal Length (cm)", y= "Sepal Width (cm)") +
  scale_color_manual(name= "Iris Species", values= c("black", "red", "blue")) +
  scale_fill_manual(name= legend.title, values= species.colors) +
  scale_shape_manual(name= legend.title, values= species.shapes) +
  theme_bw(base_size= 20) +
  theme(panel.grid.major = element_blank(),
        panel.grid.minor = element_blank(),
        legend.position = c(0.85, 0.85),
        legend.background = element_rect(color = "black"))
```



ggplot vs. base R graphics

Here is that same plot generated with base R graphics. Way more code and it still doesn't look as good. (I eventually got tired of figuring out how to further customize the plot)

```
# Save aesthetics as variables
cols <- c("black", "red", "blue")
cols_species <- cols[iris$Species]
pch_species <- as.numeric(iris$Species)

# Create the plot
plot(Sepal.Width ~ Sepal.Length, data= iris, col= cols_species, pch= pch_species, cex= 1.5, xlab= "Sepal

# Create the legend
legend(x= 4, y= 5, title= "Iris Species",
       legend=c("setosa", "versicolor", "virginica"),
       col= cols, pch= as.numeric(iris$Species),
       cex= 0.8, pt.cex= 0.8,
       bty= "n", xpd= TRUE, y.intersp= 0.5, title.adj= 0, horiz= T)

# Run linear models to get regression coefficients
fit.setosa <- lm(Sepal.Width ~ Sepal.Length,
                 data= iris[which(iris$Species == "setosa"), ])
fit.versicolor <- lm(Sepal.Width ~ Sepal.Length,
                 data= iris[which(iris$Species == "versicolor"), ])
fit.virginica <- lm(Sepal.Width ~ Sepal.Length,
                 data= iris[which(iris$Species == "virginica"), ])

# Add trend lines to the plot
abline(fit.setosa, col= "black", lwd= 2)
abline(fit.versicolor, col= "red", lwd= 2)
abline(fit.virginica, col= "blue", lwd= 2)
```



Saving plots

Arguments for `ggsave()`

- plot name: an R variable where your plot is saved
- filename: for newly created file
  - file will be deposited in your working directory
  - alternatively you can provide an entire file path to save outside your wd using the `path` argument
- device: the type of file to be saved (.pdf, .jpg, .png., etc.)
  - alternatively you can specify the extension in the filename
- width and height: numeric dimensions for the plot
- units: specify the units for the width and height arguments
  - "cm", "in", "mm"

```
ggsave(our.plot, filename= "RWorkshopPlot.pdf", width= 7, height= 5, units= "in", path= "??")
?ggsave
```

You can also use the export tab in the RStudio graphics window to save plots

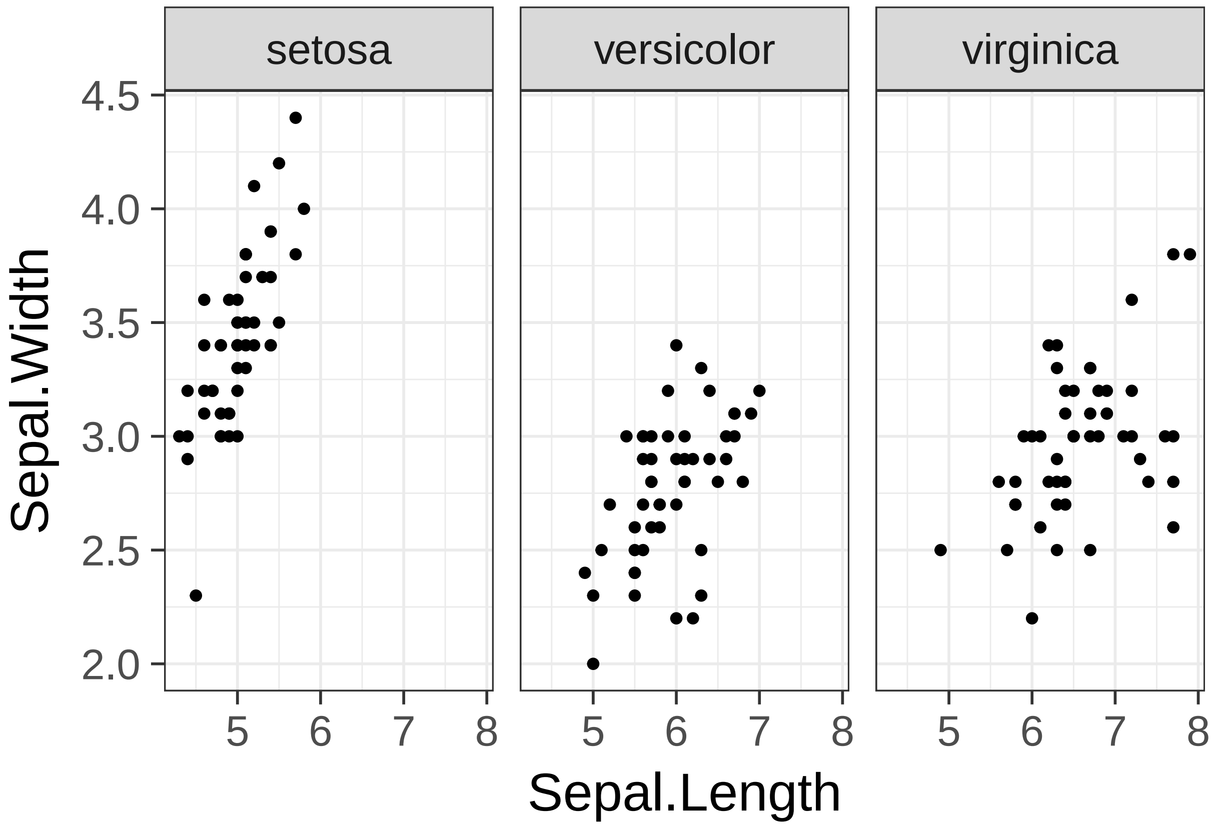
Faceting

Ggplot allows you to break up your data into differen subsets and then plot these subsets together. This called faceting, and is also known as latticing. This is a powerful way to detect patterns among subsets of your data, however it is designed to be used on subsets that share a common scale.

There are two ggplot functions to generate facets: `facet_grid()`and `facet_wrap()`

```
# We could facet our iris data rather than use colors/shapes to differentiate
# the different species
```

```r
iris.plot +
  geom_point() +
  facet_grid(. ~ Species) +
  theme_bw(base_size = 20)
```



- `facet_grid` arguments
  - Creates a 2-dimensional grid of facets based on the variables you assign
  - Uses a formula to specify the variables to be subset with a `.` as a placeholder
  - `facet_grid(. ~ Species)`: single row, `Species` variable as multiple columns
  - `facet_grid(Species ~ .)`: single column, `Species` variable as multiple rows
  - `facet_grid(Species ~ another variable)`: `Species` as rows, another variable as columns
- `facet_wrap` arguments
  - takes a 1d vector and wraps it into 2 dimensions.
  - This is helpful if you have a variable with many levels and you want to specify the number of rows and columns in your plot
  - `facet_wrap(~ Species, nrow= ?, ncol= ?)`: you specify a single variable, always preceded by a `~`, then define the rows and columns for the wrap.

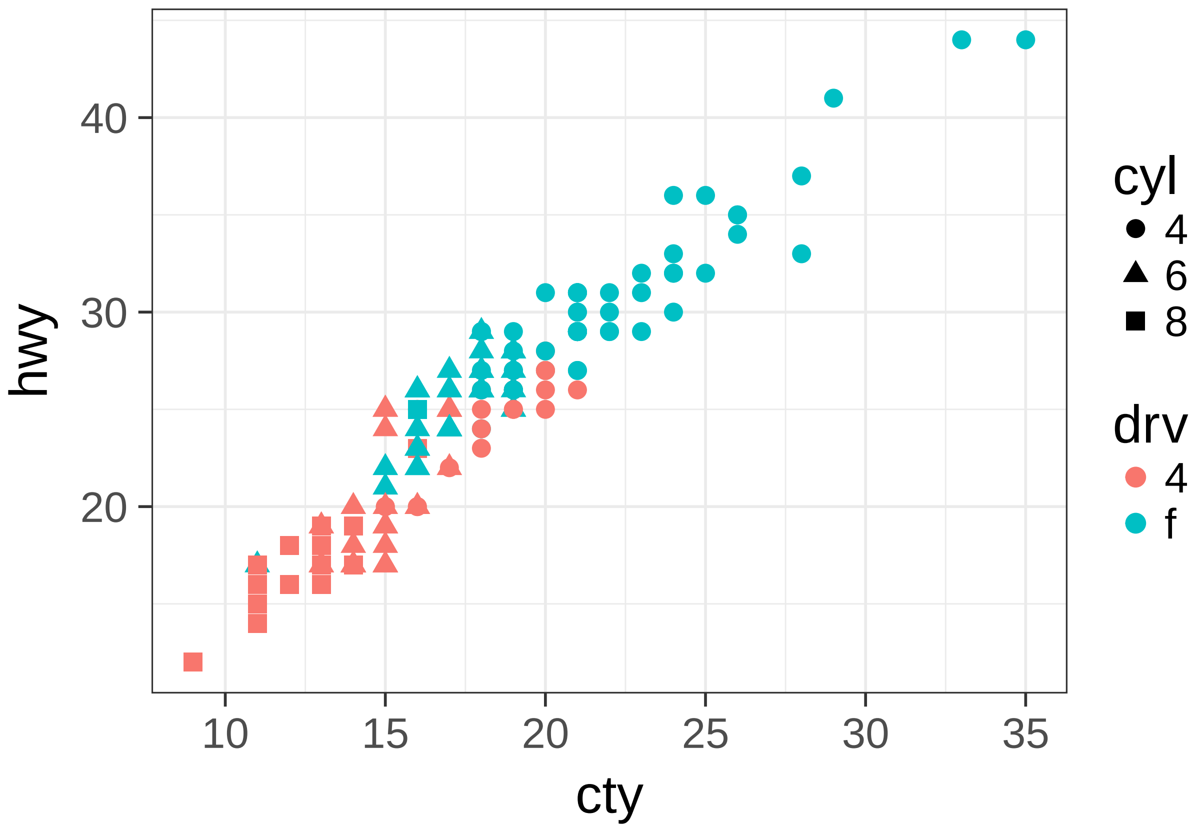
To practice faceting we will use fuel efficency data (`mpg3`) from the EPA based on number of cylinders (`cyl`) in the car and the type of drive train (`drv`) front wheel drive (`"f"`) and 4wd (`"4"`).

```r
mpg2 <- as.data.frame(subset(mpg, cyl != 5 & drv %in% c("4", "f")))
mpg3 <- mpg2[ , c(4, 5, 7, 8, 9) ]
mpg3 <- transform(mpg3,
                  cyl = as.factor(cyl),
                  drv = as.factor(drv))
head(mpg3)
```

```
##   year cyl drv cty hwy
## 1 1999   4   f  18  29
## 2 1999   4   f  21  29
## 3 2008   4   f  20  31
## 4 2008   4   f  21  30
## 5 1999   6   f  16  26
## 6 1999   6   f  18  26
```

Task 6
You will make some plots using `mpg3` to practice faceting.

1. First, make a scatter plot with `cty` on the x-axis and `hwy` on the y and `cyl` as a different shape and `drv` as a different color. Your plot should look something like this.



```
##   year cyl drv cty hwy
## 1 1999   4   f  18  29
## 2 1999   4   f  21  29
## 3 2008   4   f  20  31
## 4 2008   4   f  21  30
## 5 1999   6   f  16  26
## 6 1999   6   f  18  26
```

2. Now plot each `cyl` and `drv` as different facets using `facet_grid()`. Explore different combinations of rows and columns, but eventually finish by faceting `cyl` as 3 columns.
3. Add the argument `scales= "free_y"` to `facet_grid()`. What happened?
4. Change the background of each facet label using `theme()`

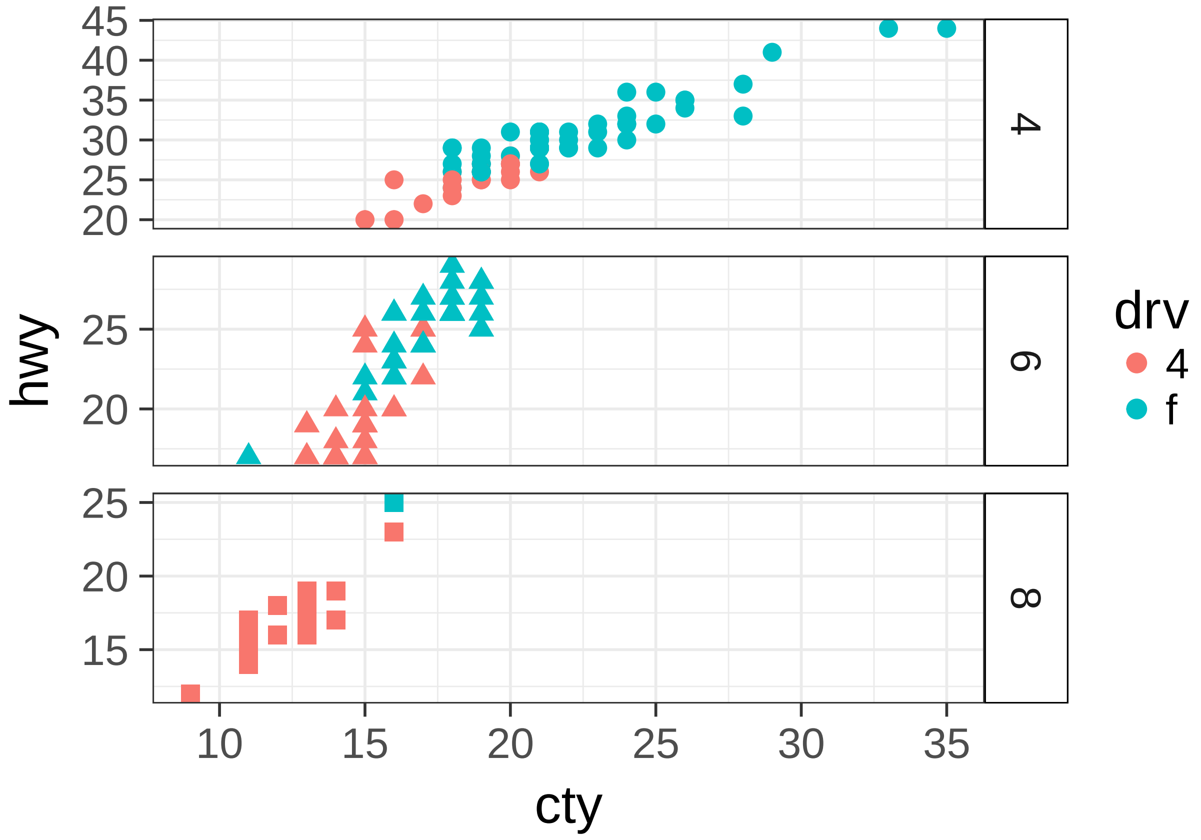
- Facet labels are called strips. Use `theme(strip.background = element_rect())` to change the color and fill. Pick any color combinations you like.

5. Remove the `cyl` legend using the `guide= FALSE` argument inside a `scale_xx_xx` function. You need

to figure out which scale function to use.

Your plot should look something like this:

```
mpg.plot <- ggplot(mpg3, aes(x= cty, y= hwy))

mpg.plot +
  geom_point(aes(shape= cyl, color= drv), size = 3) +
  facet_grid(cyl~., scales= "free_y") +
  scale_shape_discrete(guide= FALSE) +
  theme_bw(base_size= 20) +
  theme(strip.background = element_rect(fill = "white", color= "black"))
```



Bar plots and histograms

Bar plots are used for showing counts of things, the ggplot function is `geom_bar()`

When making a bar plot, `geom_bar()` can either plot the count of whatever variable you specify in the x-axis, or an actual value specified in the y-axis this is controlled using the `stat` argument

- `stat= "count"`: plots counts of the variable in the x-axis, the y value is left empty
  - this is the default if `stat` is not specified
- `stat= "identity`": plots the value specified in the y-axis

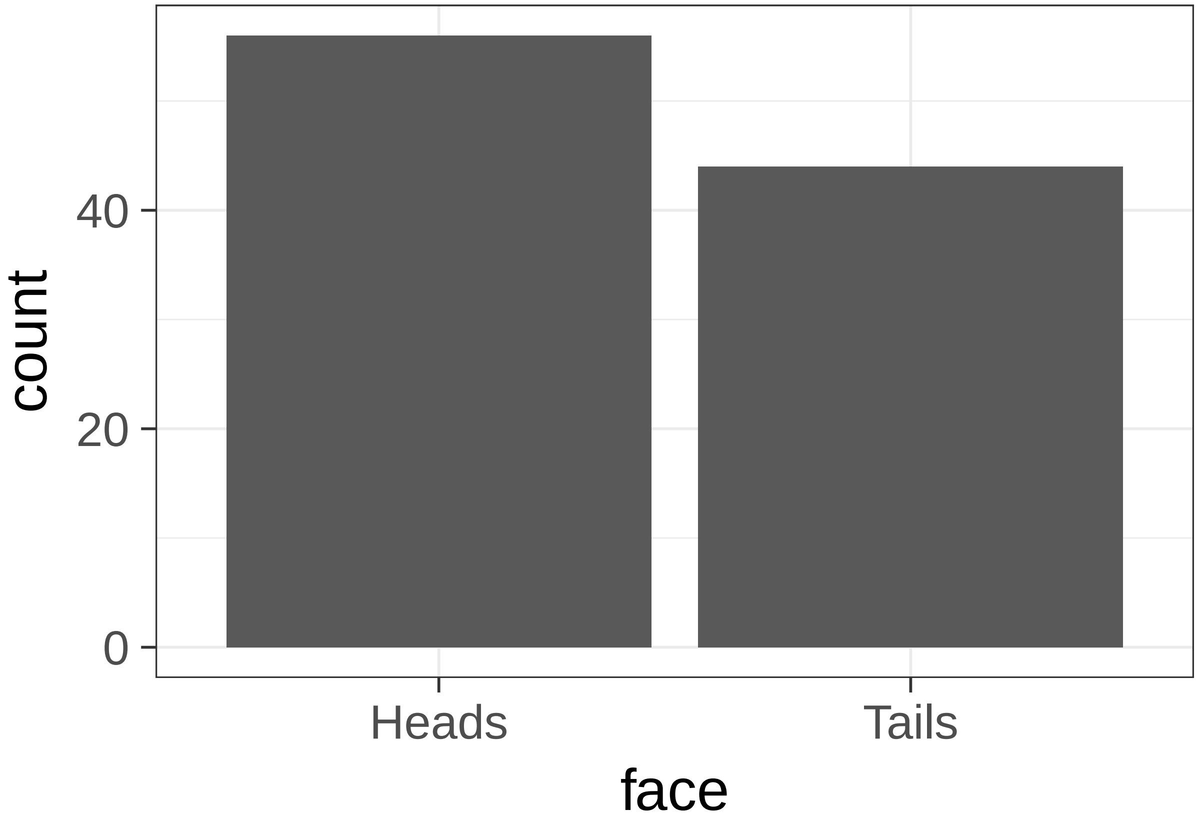
```
## Create datasets to demonstrate stat bin/identity for geom_bar()
```

```r
coin.flip <- as.data.frame(rbinom(100, 1, 0.5))
colnames(coin.flip)[1] <- "result"
coin.flip$face <- ifelse(coin.flip$result == 1, "Heads", "Tails")

library(dplyr)
coin.flip.summary <- data.frame(count(coin.flip, face))
```

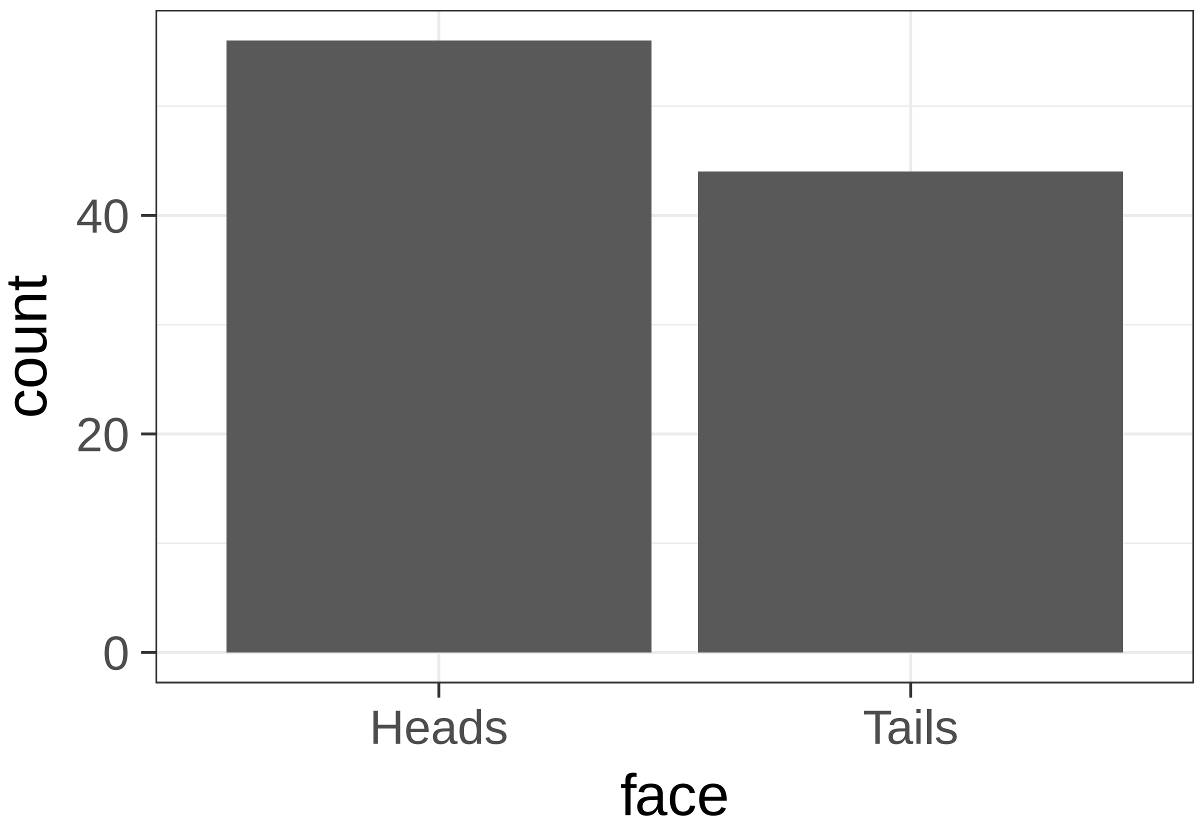
```r
# Here is data from 100 coin tosses
head(coin.flip)
```

```
##   result  face
## 1      1 Heads
## 2      0 Tails
## 3      1 Heads
## 4      0 Tails
## 5      1 Heads
## 6      1 Heads
```

```r
# Bar plot
# Notice how only the x value is defined
bar.plot1 <- ggplot(data= coin.flip, aes(x= face))

bar.plot1 + geom_bar(stat= "count") + theme_bw(base_size= 22)
```



```r
# Notice when stat is not specified it defaults to count
bar.plot1 + geom_bar() + theme_bw(base_size= 22)
```

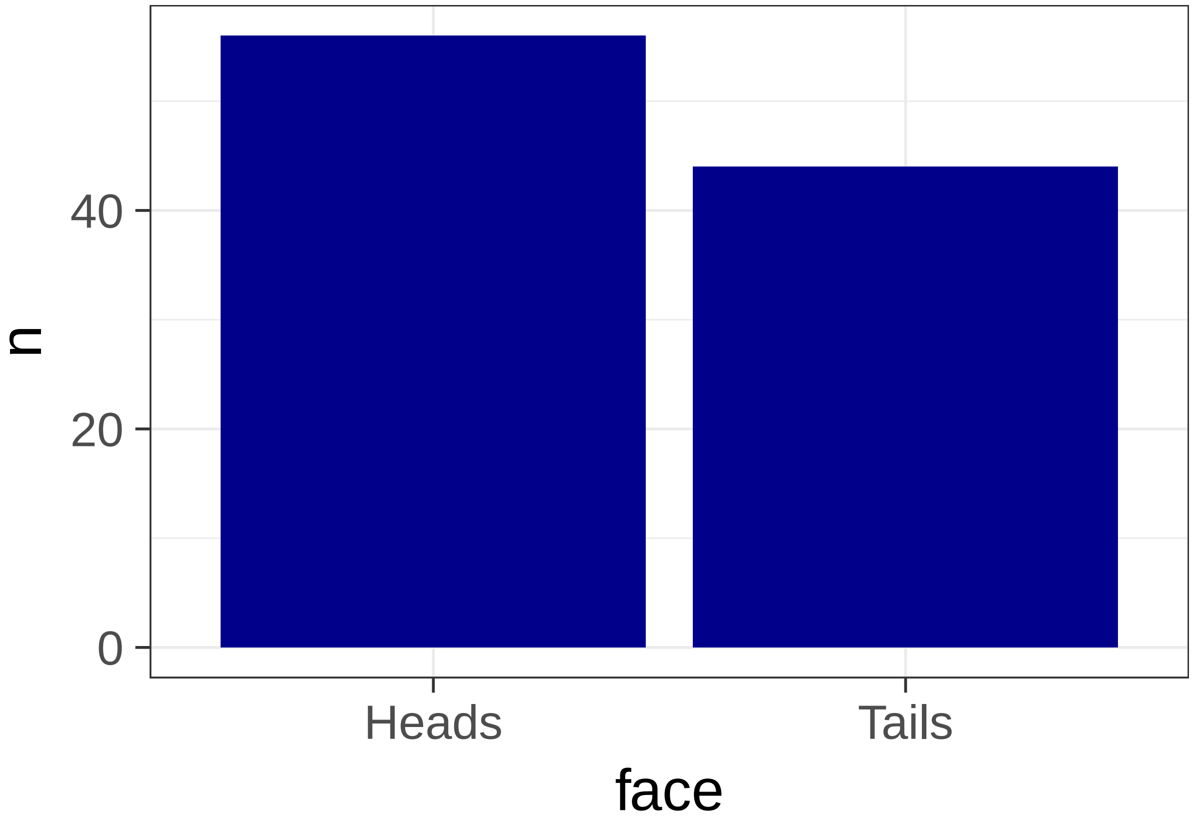


In other situations we may know the count of the data, and we want to feed it directly to ggplot, in that case we also define a y value

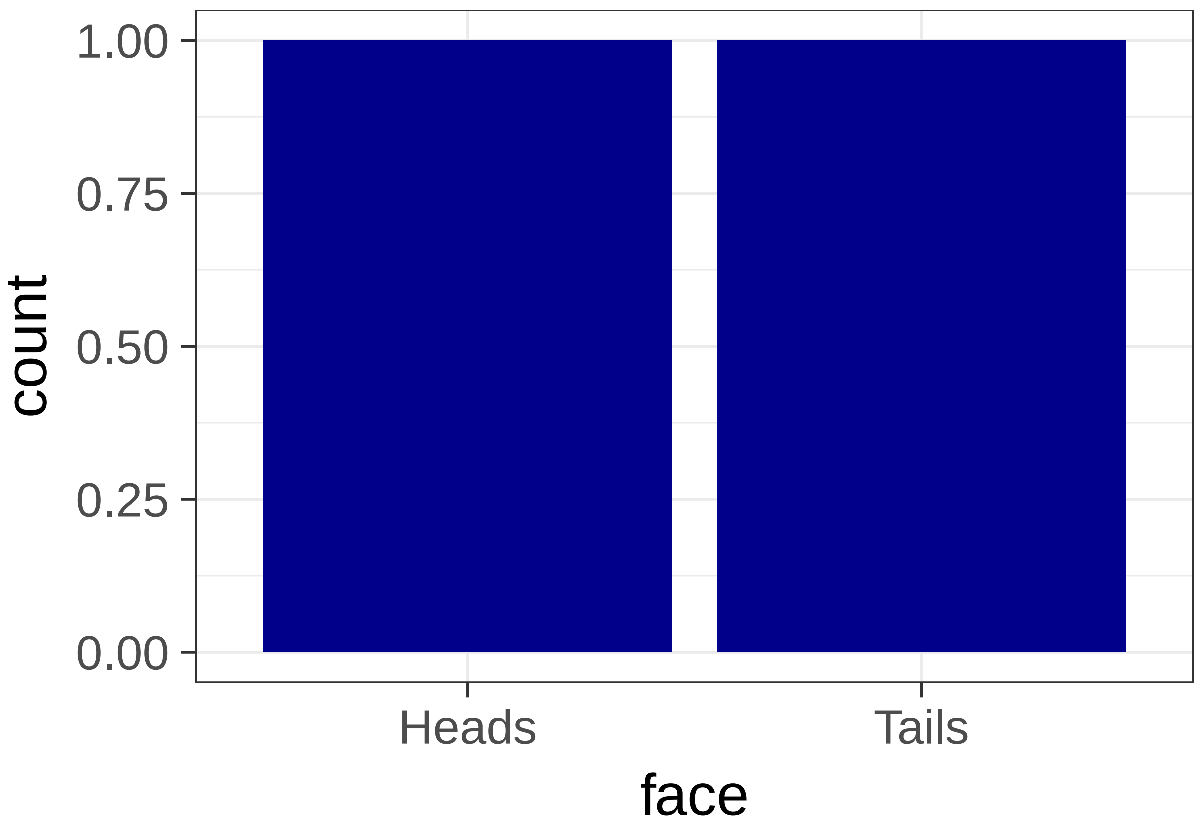
```
head(coin.flip.summary)
```

```
##    face  n
## 1 Heads 56
## 2 Tails 44
```

```
# Stat= identity
bar.plot2 <- ggplot(data= coin.flip.summary, aes(x= face, y= n))

bar.plot2 +
  geom_bar(stat= "identity", fill= "dark blue") +
  theme_bw(base_size= 22)
```



```
#If we specify stat=count, then the value is 1,
# because there is only one count of each level of the "face" variable
bar.plot3 <- ggplot(data= coin.flip.summary, aes(x= face))

bar.plot3 +
  geom_bar(stat= "count", fill= "dark blue") +
  theme_bw(base_size= 22)
```



Task 7 To explore `geom_bar()`, we will use the data set wpl, which has counts of the number of telephones in 7 regions of the world from 1951 to 1961. Spend the next few minutes generating a plot similar to the one below with:

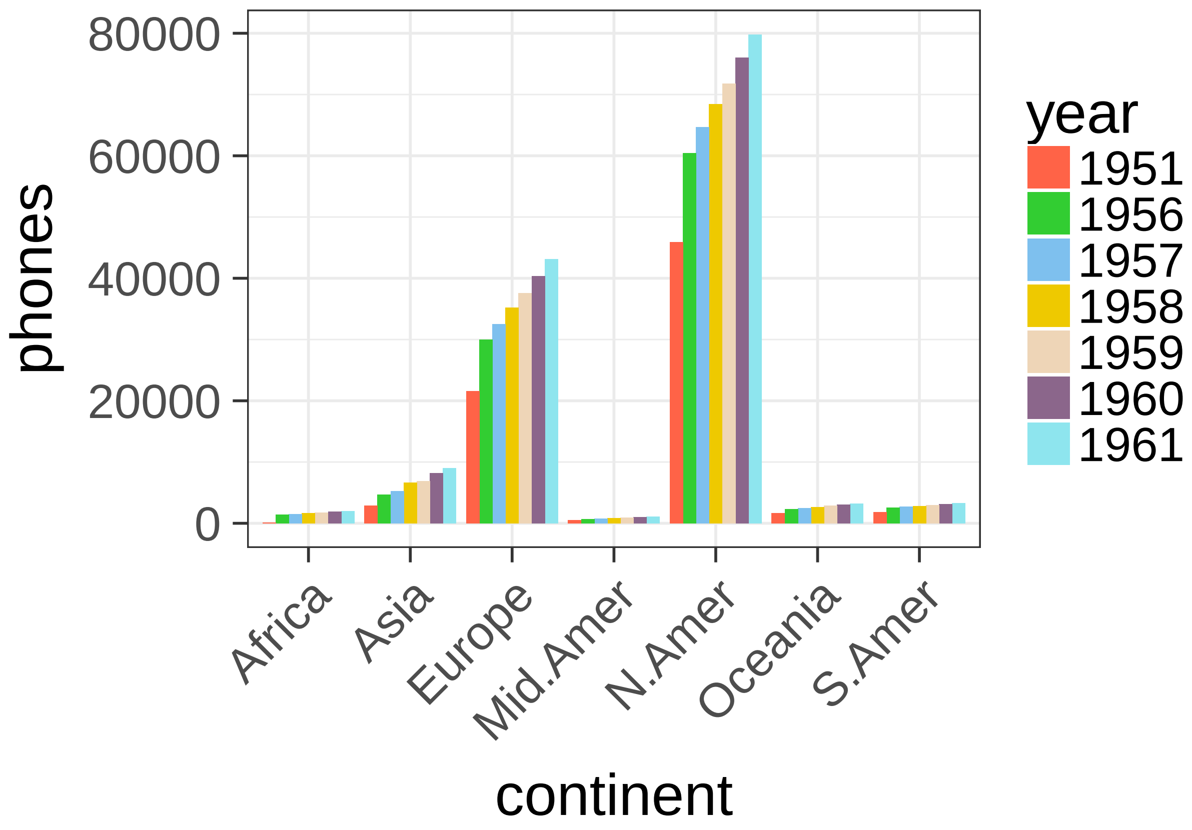
- Continent on the x-axis and telephone counts on the y-axis
- Each year in a different color
- Create a custom color scheme for each year (i.e. non-default colors)
- Experiment making plots with different values for the `position` argument
  - `position` takes values of `"stacked"`, `"dodge"`, `"fill"`
- Rotate the x-axis labels using `theme(axis.text.x = element_text(angle= ?), hjust= ?, vjust = ?)`

```
# Dataset for bar plots
# WorldPhones dataset is a base R data set.
library(tidyr)
wp <- as.data.frame(WorldPhones)
wp$year <- row.names(wp)
wpl <- gather(wp, key= continent, value= phones, N.Amer:Mid.Amer)
head(wpl)
```

```
##   year continent phones
## 1 1951    N.Amer  45939
## 2 1956    N.Amer  60423
## 3 1957    N.Amer  64721
## 4 1958    N.Amer  68484
## 5 1959    N.Amer  71799
## 6 1960    N.Amer  76036
```

```
fill.colors <- c("tomato", "limegreen", "skyblue2", "gold2", "bisque2", "plum4", "cadetblue2")
```

```r
wpl.bar.plot <- ggplot(data= wpl, aes(x= continent, y= phones))

wpl.bar.plot +
  geom_bar(aes(fill= year), stat= "identity", position= "dodge") +
  scale_fill_manual(values= fill.colors) +
  theme_bw(base_size = 22) +
  theme(axis.text.x = element_text(angle= 45, vjust= 1, hjust= 1))
```

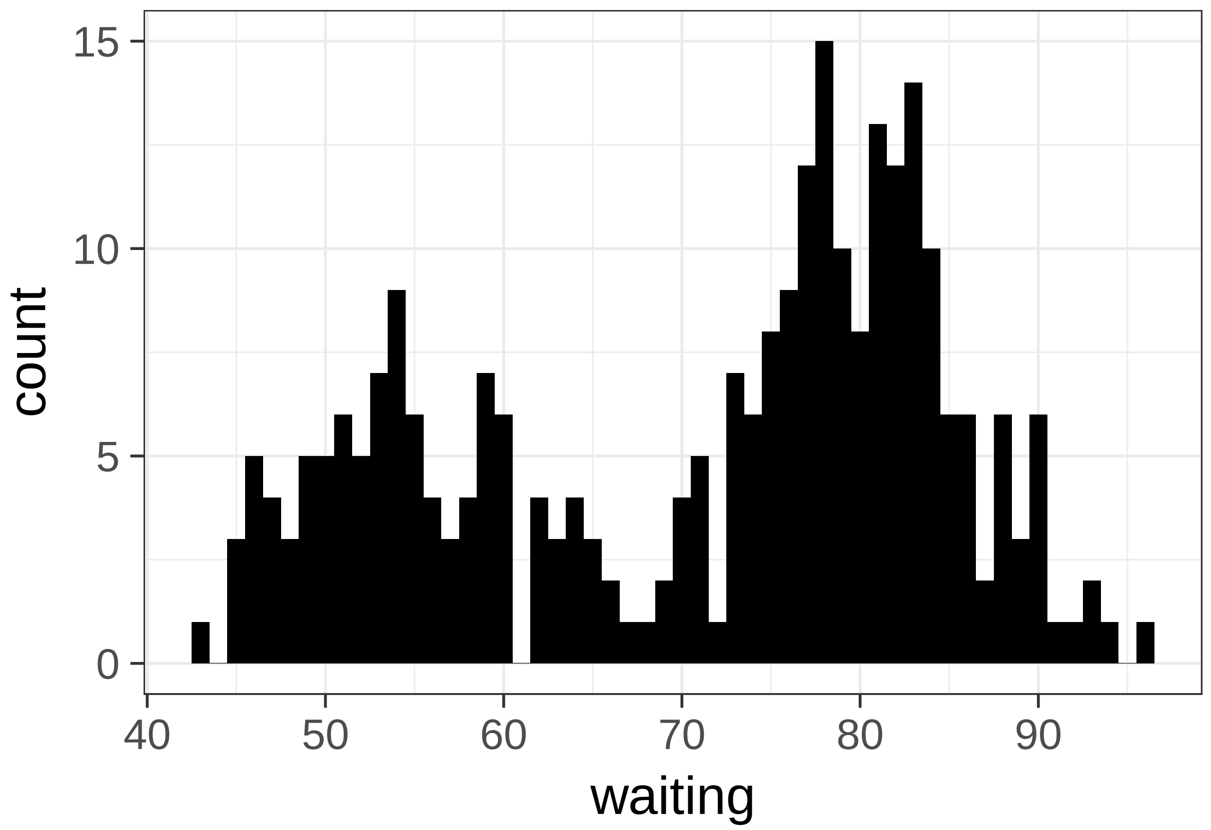


Histograms can be made with `geom_histogram` defining only x-axis variable in `aes()`. The x-axis variable must be continuous, otherwise for discrete variables use `geom_bar(stat= "count")`.

Task 8

- Using the data set `faithful`, make a histogram of the wait times (`waiting`) between eruptions of the Old Faithful geyser in Yellowstone.
- adjust the `binwidth` argument in `geom_histogram()`
- customize the plot as you see fit

```r
ggplot(data= faithful, aes(x= waiting)) +
  geom_histogram(stat= "bin", fill= "black", binwidth= 1) +
  theme_bw(base_size= 20)
```



Plotting Errorbars

Error bars can be plotted with `geom_errorbar`. First you need to have a data frame where the min and max of the error bar are already calculated. In this example we will be plotting the standard error of chicken weights across 4 different diets.

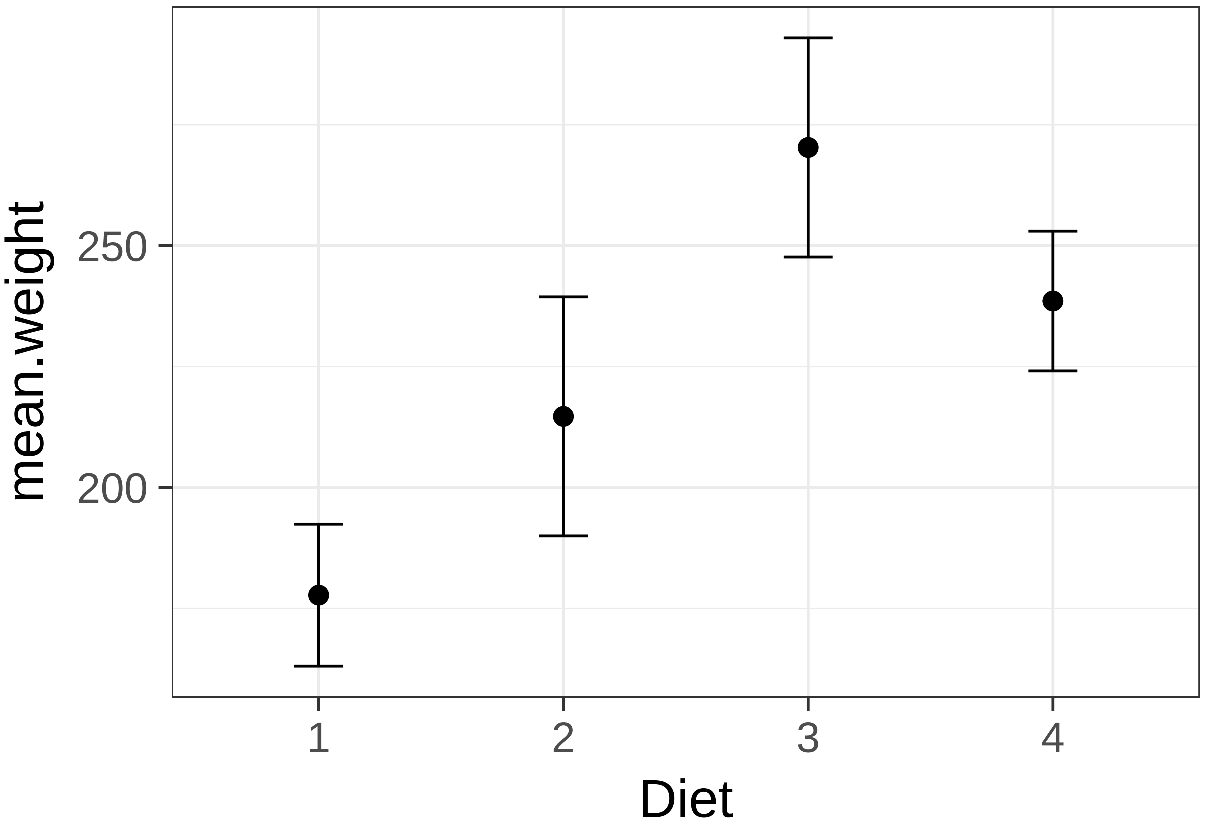
```
## Average chicken weight at day 21
library(plyr)
chick.wt <- ddply(ChickWeight[which(ChickWeight$Time == 21), ], .(Diet), summarize,
             N = length(weight),
             mean.weight = mean(weight),
             se.weight = sd(weight) / sqrt(N))
```

```
head(chick.wt)
```

```
##   Diet  N mean.weight se.weight
## 1    1 16    177.7500  14.67552
## 2    2 10    214.7000  24.70944
## 3    3 10    270.3000  22.64904
## 4    4  9    238.5556  14.44925
```

To use `geom_error bar` first plot the mean value using a different `geom_xx` layer. Then you specify the range of the errorbars within the `geom_errorbar(aes(ymin= ?, ymax= ?))`layer.

```
errorbar.plot <- ggplot(data= chick.wt, aes(x= Diet, y= mean.weight))

errorbar.plot +
  geom_point(size= 3) +
  geom_errorbar(aes(ymin= mean.weight - se.weight,
                    ymax= mean.weight + se.weight), width= 0.2) +
  theme_bw(base_size = 20)
```



Conclusion

This should give you a strong foundation to make plots using ggplot. You are now equipped to read the help documentation and explore additional ggplot capabilities.