

# ggplot2

## Stats 102A

Miles Chen based on Winston Chang's R Graphics Cookbook

Department of Statistics

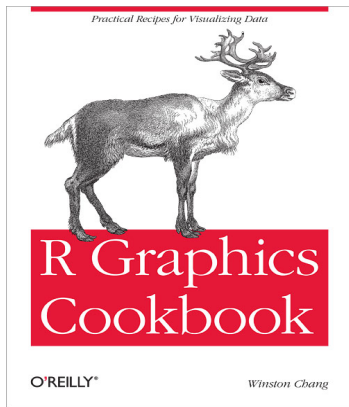
Week 5 Friday



# The R Graphics Cookbook

Today's lecture is largely derived from **The R Graphics Cookbook** by Winston Chang

<https://r-graphics.org/>



## Section 1

### introducing ggplot2

Today I will cover the usage of the package `ggplot2`, created by Hadley Wickham, and part of the tidyverse.

The `gg` in `ggplot2` stands for grammar of graphics, a system created by Leland Wilkinson. The system provides a formal and structured perspective on how to describe data graphics.

Although I am a big fan of `ggplot2`, I still frequently use base graphics.

It is often faster and easier to inspect and explore data with R's base graphics, especially when the data isn't already structured properly for use with `ggplot2`. I assume you already have some exposure to R's base graphics capabilities from Stats 20.

# Grammar of Graphics: Big Idea

In a data graphic, **we map properties of the data to visual properties of the graphic.**

Imagine we are making a scatterplot.

If we assume our data is tidy, the properties of the data are the variables stored in columns. For example, a person's height, their weight, their gender, their hair color, etc.

The visual properties of the graphic include visual properties like the x position or y position of points, the colors of the points, the symbol used for the points, and so on.

If a graphic does not map properties of the data to visual properties, it would not be a data visualization.

# Scatterplot example

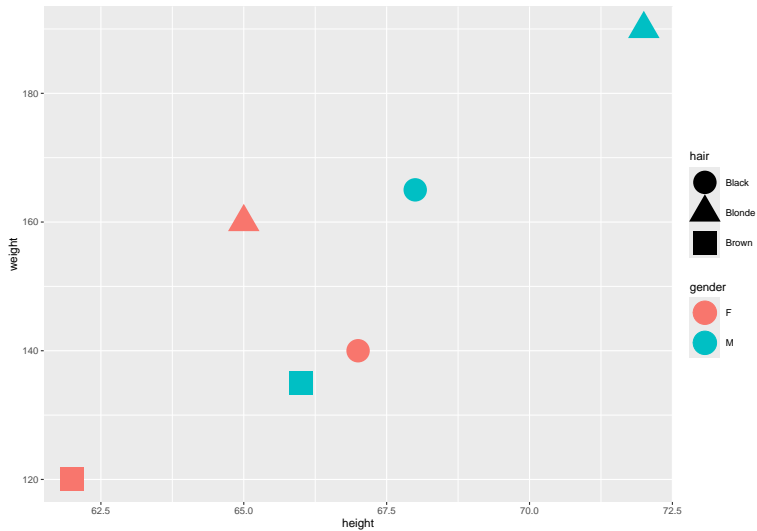
```
df <- data.frame(  
  height = c( 62,  67,  72,  68,  66,  65),  
  weight = c(120, 140, 190, 165, 135, 160),  
  gender = c("F", "F", "M", "M", "M", "F"),  
  hair = c("Brown", "Black", "Blonde", "Black", "Brown", "Blonde"))  
print(df)
```

##	height	weight	gender	hair
## 1	62	120	F	Brown
## 2	67	140	F	Black
## 3	72	190	M	Blonde
## 4	68	165	M	Black
## 5	66	135	M	Brown
## 6	65	160	F	Blonde

# Scatterplot example

```
ggplot(data = df,  
       mapping = aes(  
         x = height,  
         y = weight,  
         color = gender,  
         shape = hair  
       )) +  
geom_point(size = 10)
```

# Scatterplot example





# Data to Graphic to Brain

The power of a data visualization is that it allows our brain to utilize its visual processing abilities to think about quantities and values.

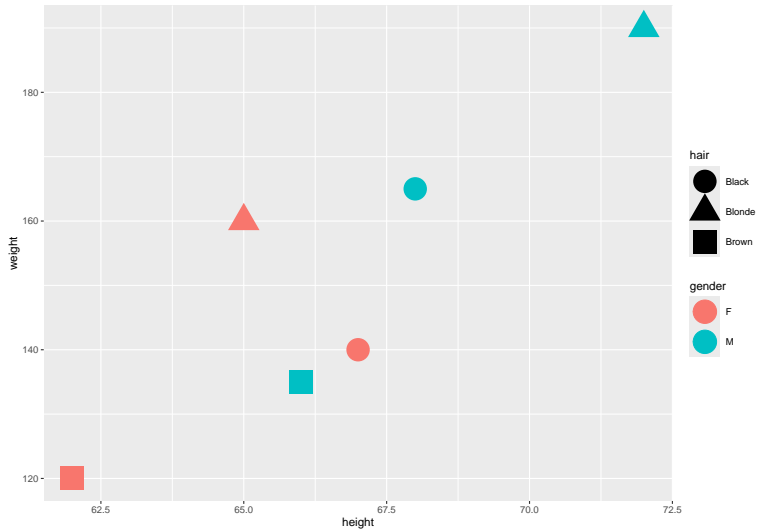
When we look only at numeric values, our brain is not able to quickly reason about them.

When we see them represented as points, our brain is able to visually process the information and identify patterns and relationships much more quickly.

The purpose of a data visualization is:

- to map properties of the data to a graphic
- **and** to assist the viewer to map the graphic back to the data in the brain

# Look at the Scatterplot again



# Mapping back to the data

By looking only at the scatterplot, we could, if we had to, recreate the entire dataset.

```
print(df)
```

```
##   height weight gender  hair
## 1     62   120      F  Brown
## 2     67   140      F  Black
## 3     72   190      M  Blonde
## 4     68   165      M  Black
## 5     66   135      M  Brown
## 6     65   160      F  Blonde
```

# Some Terminology and Theory

Before we go any further, it'll be helpful to define some of the terminology used in `ggplot2`:

- The **data** is what we want to visualize. It consists of variables, which are stored as columns in a data frame.
- **Geoms** are the geometric objects that are drawn to represent the data, such as bars, lines, and points.
- **Aesthetic attributes (aesthetics)**, are visual properties of geoms, such as x and y position, line color, point shapes, etc.
- **Mappings** are the connections between data values and aesthetics.
- **Scales** define how data values are transformed into aesthetic values (e.g., mapping data values to positions, colors, or sizes).
- **Guides** are visual aids (such as axis labels and legends) that help viewers interpret the mapping between aesthetics and data values.

# Our scatterplot example

In our scatterplot example,

- Our data is a small sample of people
- We select **point** as our **geoms** (geometric objects) to represent each observation.
- We **map** each variable to an **aesthetic attribute** such as x-position, y-position, point color, and point shape.

```
ggplot(data = df,  
       mapping = aes(  
         x = height,  
         y = weight,  
         color = gender,  
         shape = hair  
       )) +  
  geom_point(size = 10)
```

# Our scatterplot example

On the plot itself:

- The **scales** were determined automatically by ggplot.
  - ▶ It chose to make the lower left corner something around (59.5, 118) and the upper right corner something around (72.5, 192)
  - ▶ It automatically chose the colors salmon and teal to represent genders, and the shapes circle, triangle, and square to represent hair colors.
- The **guides** were also automatically created by ggplot.
  - ▶ it creates a legend to the right of the graphic showing the shapes and colors and their meanings
  - ▶ it put tick marks and variable names on the x and y axes

## Section 2

### Aesthetic Mappings

# Aesthetic Mappings

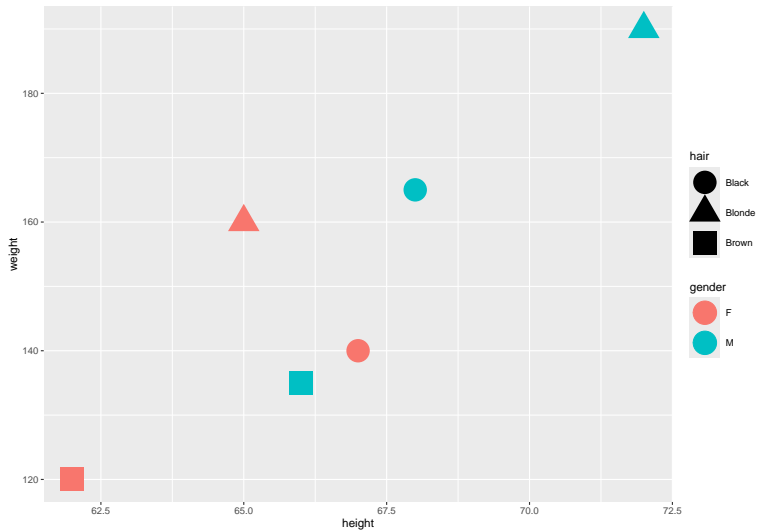
The central idea in `ggplot` is that each variable is mapped to an aesthetic attribute.

In our earlier scatterplots, we mapped *height* to *x*, *weight* to *y*, *gender* to *point color* and *hair color* to *point shape*

```
ggplot(data = df,  
       mapping = aes(  
         x = height,  
         y = weight,  
         color = gender,  
         shape = hair  
       )) +  
geom_point(size = 10)
```



# Original scatterplot

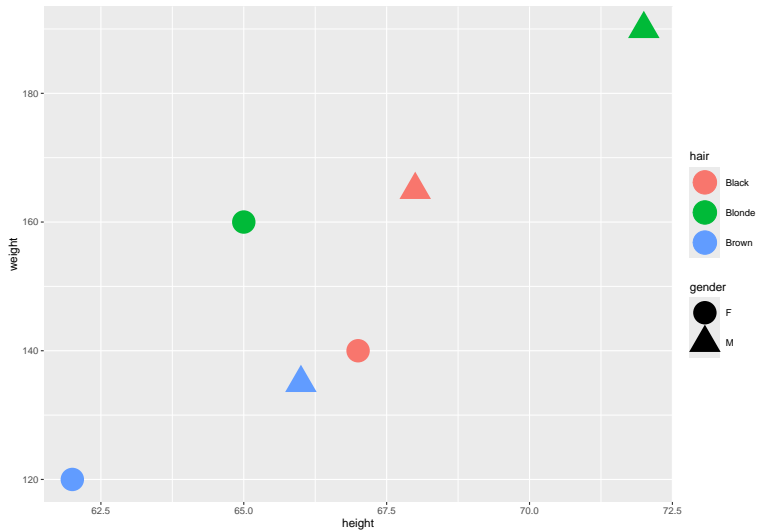


# Changing the Aesthetic mapping

We can change the aesthetic mappings. This time we map hair color to the point color, and gender to the point shape.

```
ggplot(data = df,  
       mapping = aes(  
         x = height,  
         y = weight,  
         color = hair,  
         shape = gender  
       )) +  
geom_point(size = 10)
```

# Changing the Aesthetic mapping



# Recommendations for Aesthetic Mappings

Our eyes generally notice color before they notice shape.

- Use color to represent the more important categorical variable.
- Use shape for the less important categorical variable.

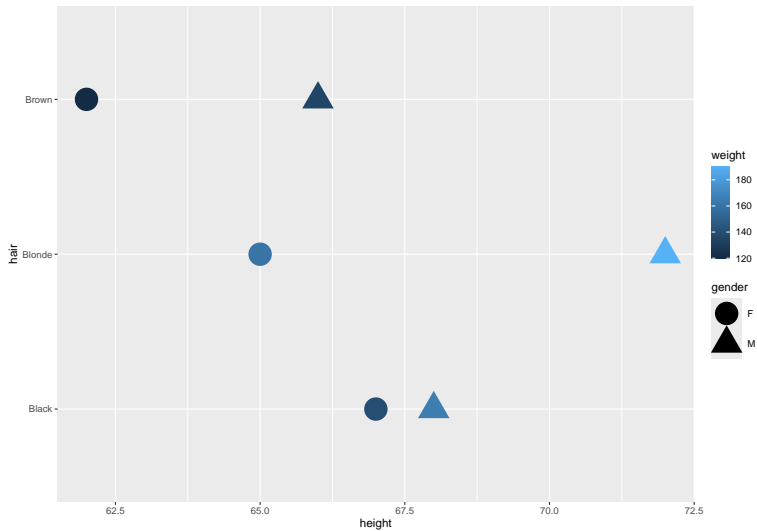
# “Unusual” Mappings

What would happen if I map a categorical variable like “hair” onto the y-axis?

What would happen if I map a numeric variable like “weight” onto the color aesthetic?

```
ggplot(data = df,  
       mapping = aes(  
         x = height,  
         y = hair,  
         color = weight,  
         shape = gender  
       )) +  
geom_point(size = 10)
```

# “Strange” Mappings



ggplot does its best with the mappings we provide.

It puts each category of hair color on the y-axis - a position for “brown”, “blonde”, and “black”.

It also colors the values accordingly, creating a continuous color scale. Darker shades of blue represent smaller numbers, and lighter shades represent larger numbers.

**On the surface, representing a number with an y coordinate may seem very different from representing a number with a color of a point, but at an abstract level, they are the same.**

# Variable Types Determine the Aesthetics

Some aesthetics can only work with categorical variables, such as the shape of a point: triangles, circles, squares, etc.

Some aesthetics work with categorical or continuous variables, such as x or y position, and as we saw even color.

For a bar graph, the variable mapped to the x-axis is usually categorical.

For a scatter plot, the variable mapped to the x and y-axes are usually numeric.



## Section 3

### ggplot vs base graphics

# A quick comparison between ggplot and base graphics

We could create a comparable graphic with base graphics.

However, it is a bit more work to achieve this.

- we must convert the categorical variable gender into a vector of colors. With only two categories, it can be done easily with `ifelse()`
- we must convert the categorical variable hair color into a vector of shapes (R calls these pch values). To do this, I convert `df$hair` to an integer vector, and then use those values to 'subset' the vector of desire shape values.
- we call `plot`
- we must add the legends, which itself can become a time consuming process

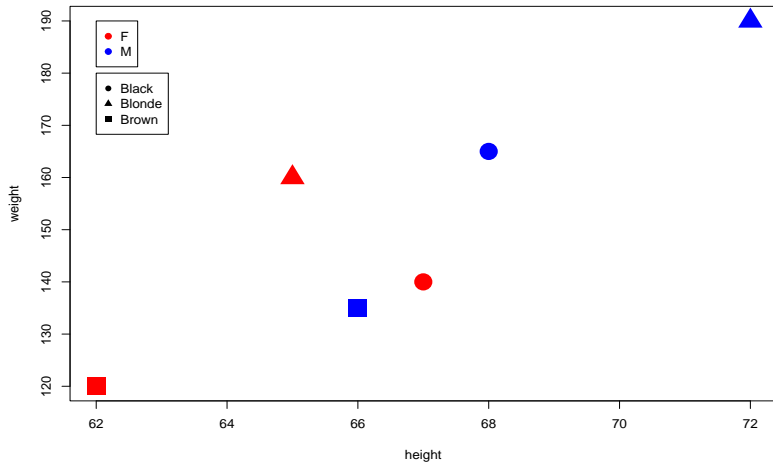
## Comparison: scatterplot commands with basegraphics

```
colors = ifelse(df$gender == "F", "Red", "Blue")

ints = as.integer(factor(df$hair)) # becomes c(3, 1, 2, 1, 3, 2)
shapes = c(16, 17, 15)[ints]

plot(df$height, df$weight, col = colors, pch = shapes, cex = 3, xlab = "height")
legend(62, 190, c("F", "M"), col = c("Red", "Blue"), pch = 19)
legend(62, 180, c("Black", "Blonde", "Brown"), pch = c(16, 17, 15))
```

# Comparison: The scatterplot with basegraphics

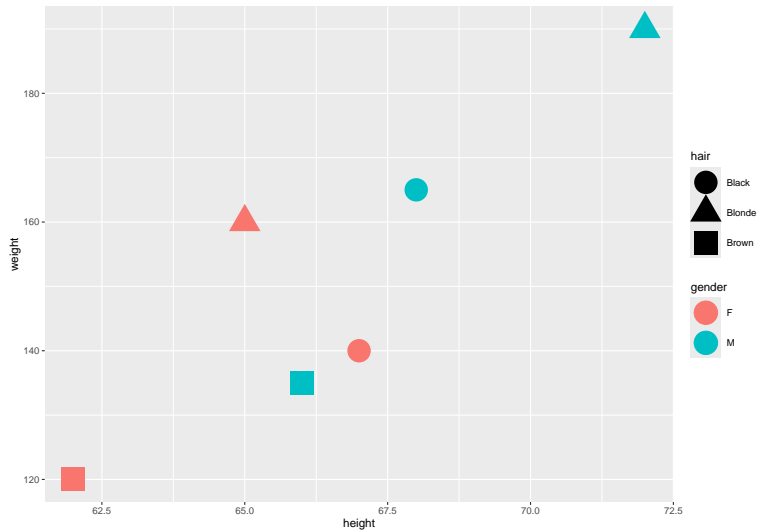


## Comparison: ggplot commands

Compare the base graphics command to the ggplot command. Here I simply provide the data, how the variables should be mapped to aesthetic attributes, and tell ggplot that I want them represented with points.

```
ggplot(data = df,  
       mapping = aes(  
         x = height,  
         y = weight,  
         color = gender,  
         shape = hair  
       )) +  
geom_point(size = 5)
```

# Comparison: The scatterplot with ggplot



## Section 4

### Starting from scratch

# Building a Simple Graph

Ggplot2 requires that data is tidy:

- they must be stored in data frames
- each type of variable that is mapped to an aesthetic must be stored in its own column

```
df <- data.frame(  
  height = c( 62,  67,  72,  68,  66,  65),  
  weight = c(120, 140, 190, 165, 135, 160),  
  gender = c("F", "F", "M", "M", "M", "F"),  
  hair = c("Brown", "Black", "Blonde", "Black", "Brown", "Blonde"))  
print(df)
```

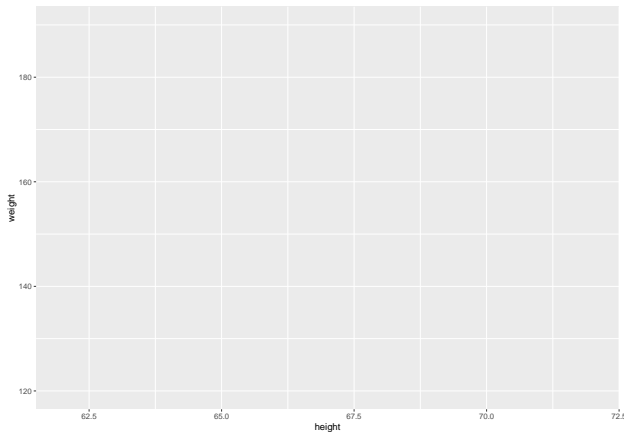
##	height	weight	gender	hair
## 1	62	120	F	Brown
## 2	67	140	F	Black
## 3	72	190	M	Blonde
## 4	68	165	M	Black
## 5	66	135	M	Brown
## 6	65	160	F	Blonde



# A Graph without Geoms

A basic ggplot() specification looks like this:

```
ggplot(df, aes(x = height, y = weight))
```



## ggplot without any Geoms

The previous command creates a ggplot object using the data frame `dat`. It also specifies default aesthetic mappings within `aes()`:

- `x = height` maps the column height to the x position.
- `y = weight` maps the column weight to the y position.

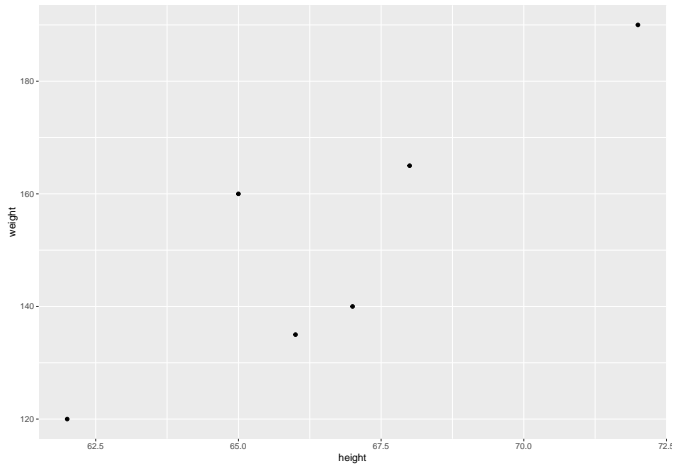
After we've given `ggplot()` the data frame and the aesthetic mappings, there's one more critical component: we need to tell it what geometric objects to put there.

At this point, `ggplot2` doesn't know if we want bars, lines, points, or something else to be drawn on the graph.

We'll add `geom_point()` to draw points, resulting in a scatter plot.

# A Basic Scatterplot

```
ggplot(df, aes(x = height, y = weight)) + geom_point()
```



## Add Another Aesthetic Mapping

If you're going to reuse some of these components, you can store them in variables. We can save the ggplot object in `p`, and then add `geom_point()` to it. This has the same effect as the preceding code:

```
p <- ggplot(df, aes(x = height, y = weight))
```

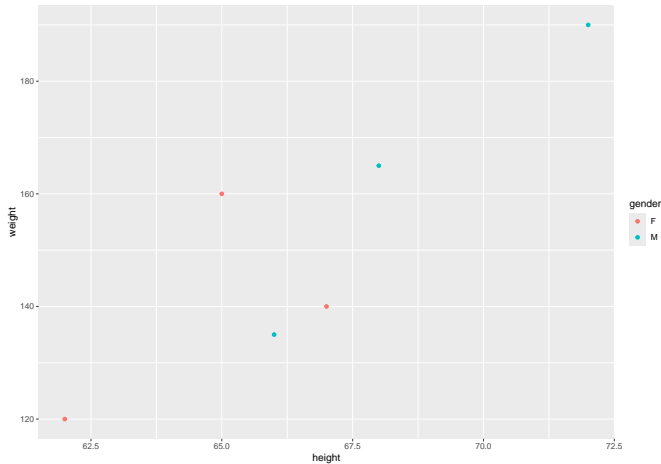
```
p + geom_point()
```

We can also map the variable `group` to the color of the points, by putting `aes()` inside the call to `geom_point()`, and specifying `color=group`:

```
p + geom_point(aes(color = group))
```

# Add Another Aesthetic Mapping

```
p + geom_point(aes(color = gender))
```



# Aesthetic Mappings vs Settings

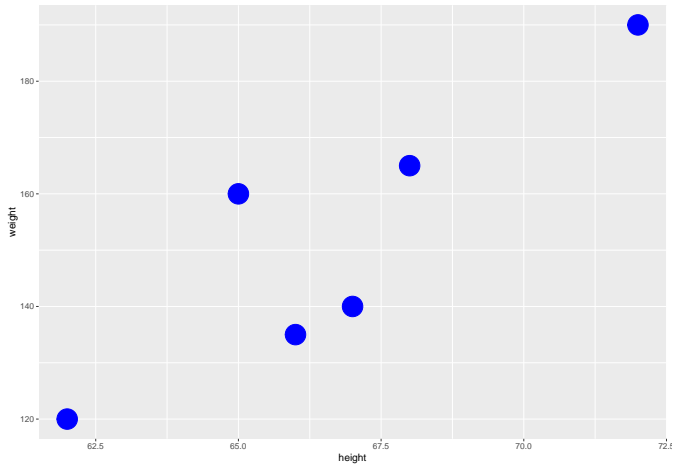
This doesn't alter the default aesthetic mappings that we defined previously, inside of `ggplot(...)`. What it does is add an aesthetic mapping for this particular geom, `geom_point()`. If we added other geoms, this mapping would not apply to them.

Contrast this aesthetic mapping with aesthetic setting. This time, we won't use `aes()`; we'll just set the value of color and size directly:

```
p + geom_point(color = "blue", size = 10)
```

# An Aesthetic Setting

```
p + geom_point(color = "blue", size = 10)
```



## Changing the scale

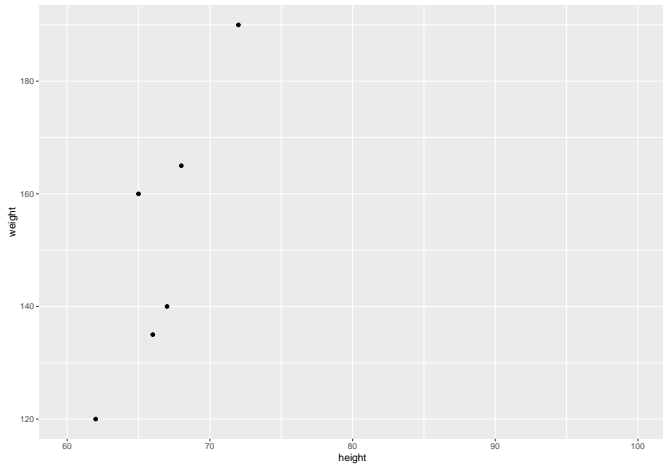
We can also modify the scales; that is, the mappings from data to visual attributes. Here, we'll change the x scale so that it has a larger range:

```
p + geom_point() + scale_x_continuous(limits = c(60,100))
```



# Changing a scale

```
p + geom_point() + scale_x_continuous(limits = c(60,100))
```



# Changing a scale

If we use the `color = gender` mapping, we can also modify the color scale.

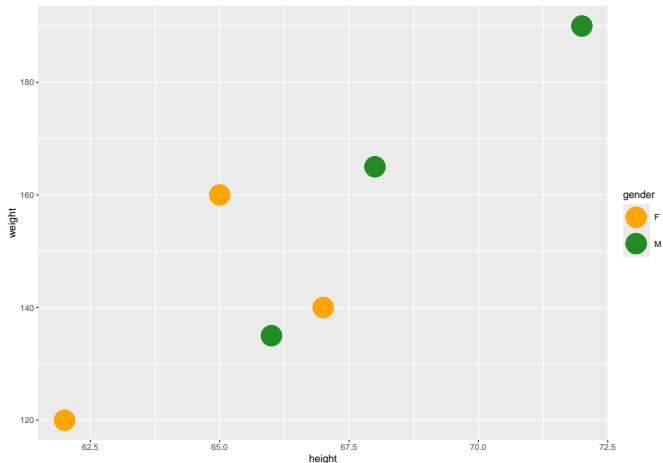
Note that `color = gender` is a mapping and goes inside `aes()`. `size = 10` is a setting for all points, and remains outside of `aes()`

```
p + geom_point(aes(color = gender), size = 10) +  
  scale_color_manual(values=c("orange", "forestgreen"))
```

When you modify the scale, the guide also changes. With the `x` scale, the guide is the markings along the `x`-axis. With the color scale, the guide is the legend.

# Changing a scale

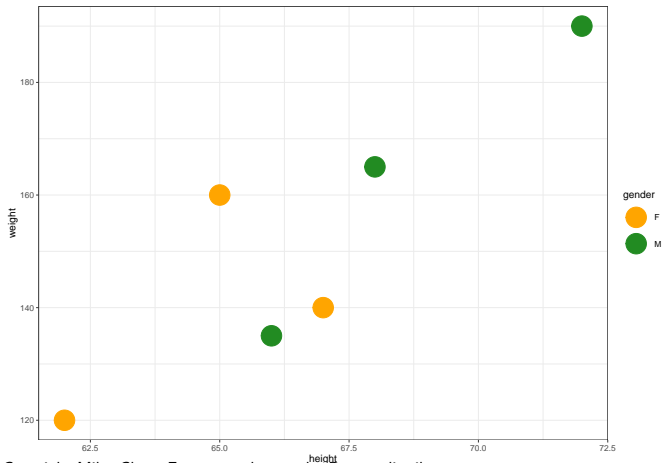
```
p2 <- p + geom_point(aes(color = gender), size = 10) +  
  scale_color_manual(values=c("orange", "forestgreen"))  
print(p2)
```



# Using themes

You can change the color of the background and style of the plot with themes. A common choice is `theme_bw()` which gets rid of the gray background.

```
p2 + theme_bw()
```



In R's base graphics, the graphing functions tell R to draw graphs to the output device (the screen or a file). Ggplot2 is a little different. The commands don't directly draw to the output device.

Instead, the functions build plot objects, and the graphs aren't drawn until you use the `print()` function, as in `print(object)`.

It might feel strange because we have not explicitly asked R to `print()` any of our graphs.

In R, when you issue a command at the prompt, it really does two things: first it runs the command, then it runs `print()` with the returned result of that command.

The behavior at the interactive R prompt is different from when you run a script or function. In scripts, commands aren't automatically printed. So in scripts, after building your ggplot object, you'll want to run `print()`

Sometimes your data must be transformed or summarized before it is mapped to an aesthetic.

This is true, for example, with a histogram, where the samples are grouped into bins and counted. The counts for each bin are then used to specify the height of a bar. Some geoms, like `geom_histogram()`, automatically do this for you, but sometimes you'll want to do this yourself, using various `stat_xx()` functions.

## Section 5

geoms

# The Basic ggplot Geometries

The basics:

- `geom_point()` [https://ggplot2.tidyverse.org/reference/geom\\_point.html](https://ggplot2.tidyverse.org/reference/geom_point.html)
- `geom_line()` and `geom_path()` [https://ggplot2.tidyverse.org/reference/geom\\_path.html](https://ggplot2.tidyverse.org/reference/geom_path.html)
- `geom_col()` and `geom_bar()` [https://ggplot2.tidyverse.org/reference/geom\\_bar.html](https://ggplot2.tidyverse.org/reference/geom_bar.html)

Some useful ones that incorporate automatic statistical summaries and stuff

- `geom_histogram()` - this is really a `geom_bar()` applied to the data after using stats to bin the data
- `geom_boxplot()`



Full list: <https://ggplot2.tidyverse.org/reference/index.html#section-geoms>

Also get this cheat sheet:

<https://github.com/rstudio/cheatsheets/blob/main/data-visualization.pdf>

## Section 6

### Bar / Column Graphs

## From the documentation:

There are two types of bar charts: `geom_bar()` and `geom_col()`.

`geom_bar()` makes the height of the bar proportional to the number of cases in each group (or if the weight aesthetic is supplied, the sum of the weights).

If you want the heights of the bars to represent values in the data, use `geom_col()` instead.

# Bar Graphs

Data format:

- One column (categorical or numeric) represents the x position of each bar (column),
- One column (numeric) represents the vertical (y) height of each bar (column).

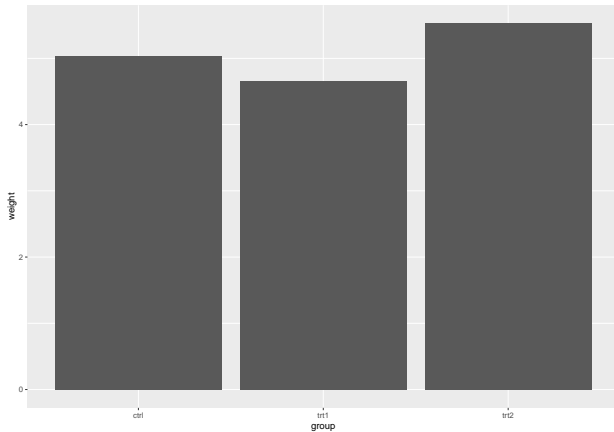
```
pg_mean
```

```
##   group weight
## 1  ctrl  5.032
## 2  trt1  4.661
## 3  trt2  5.526
```

Use `ggplot()` with `geom_col()` and specify what variables you want on the x- and y-axes:

# Bar Graphs

```
ggplot(pg_mean, aes(x = group, y = weight)) + geom_col()
```



## Create side-by-side barplots

To create a side-by-side barplot, map a variable to fill, and use `geom_bar(position="dodge")`. In this case, the data has another column that is categorical that can be mapped to the fill.

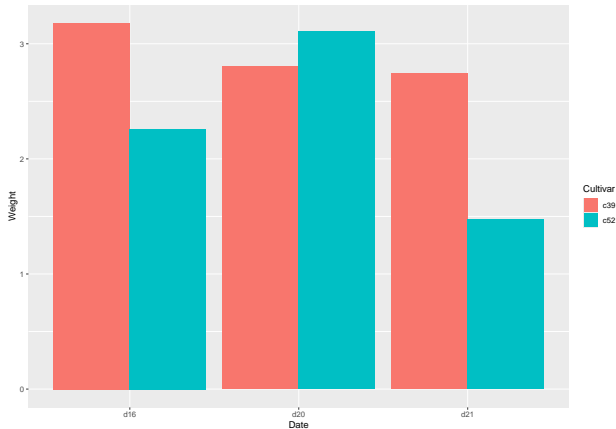
```
cabbage_exp
```

##	Cultivar	Date	Weight	sd	n	se
## 1	c39	d16	3.18	0.9566144	10	0.30250803
## 2	c39	d20	2.80	0.2788867	10	0.08819171
## 3	c39	d21	2.74	0.9834181	10	0.31098410
## 4	c52	d16	2.26	0.4452215	10	0.14079141
## 5	c52	d20	3.11	0.7908505	10	0.25008887
## 6	c52	d21	1.47	0.2110819	10	0.06674995

```
ggplot(cabbage_exp, aes(x=Date, y=Weight, fill=Cultivar)) +  
  geom_col(position="dodge")
```

# Side-by-side barplots

```
ggplot(cabbage_exp, aes(x = Date, y = Weight, fill = Cultivar)) +  
  geom_col(position="dodge")
```



# Barplot to summarize counts

This time, you have a large dataset, and would like to summarize a categorical variable by graphing the counts of each category.

Use `geom_bar()` without mapping anything to `y` and without specifying `stat = identity`. `diamonds` data has 53940 rows.

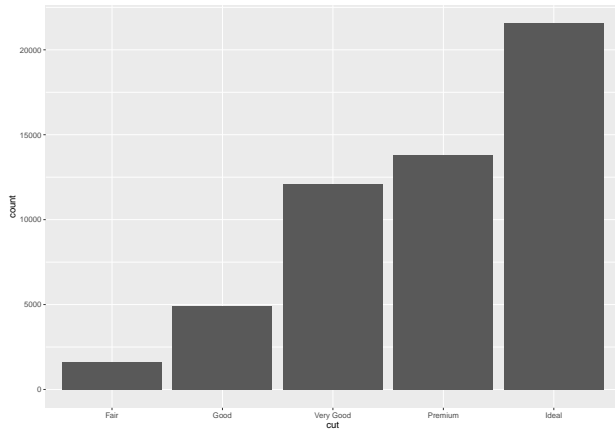
```
print(diamonds, n = 8)
```

```
## # A tibble: 53,940 x 10
##   carat cut      color clarity depth table price      x      y      z
##   <dbl> <ord>    <ord> <ord>  <dbl> <dbl> <int> <dbl> <dbl> <dbl>
## 1  0.23 Ideal      E      SI2    61.5    55   326   3.95   3.98   2.43
## 2  0.21 Premium    E      SI1    59.8    61   326   3.89   3.84   2.31
## 3  0.23 Good       E      VS1    56.9    65   327   4.05   4.07   2.31
## 4  0.29 Premium    I      VS2    62.4    58   334   4.2    4.23   2.63
## 5  0.31 Good       J      SI2    63.3    58   335   4.34   4.35   2.75
## 6  0.24 Very Good  J      VVS2    62.8    57   336   3.94   3.96   2.48
## 7  0.24 Very Good  I      VVS1    62.3    57   336   3.95   3.98   2.47
## 8  0.26 Very Good  H      SI1    61.9    55   337   4.07   4.11   2.53
## # i 53,932 more rows
```



# Barplot to summarize counts

```
ggplot(diamonds, aes(x = cut)) + geom_bar()
```



# Mapping a variable to the fill

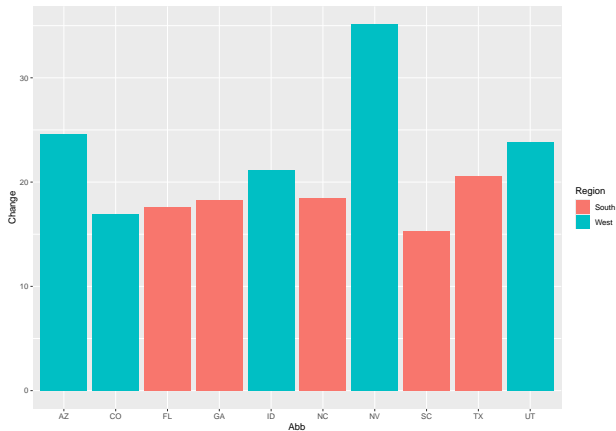
You can display another variable by mapping it to the aesthetic attribute fill.

```
upc <- subset(uspchange, rank(Change) > 40) # top 10 pop changes
upc
```

##	State	Abb	Region	Change
## 3	Arizona	AZ	West	24.6
## 6	Colorado	CO	West	16.9
## 10	Florida	FL	South	17.6
## 11	Georgia	GA	South	18.3
## 13	Idaho	ID	West	21.1
## 29	Nevada	NV	West	35.1
## 34	North Carolina	NC	South	18.5
## 41	South Carolina	SC	South	15.3
## 44	Texas	TX	South	20.6
## 45	Utah	UT	West	23.8

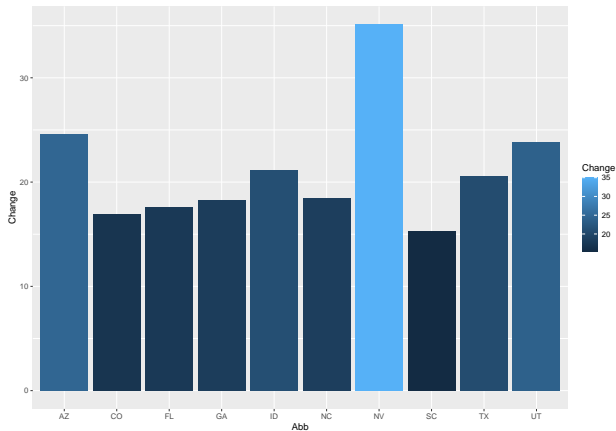
# Mapping a categorical variable to color

```
ggplot(upc, aes(x = Abb, y = Change, fill = Region)) +  
  geom_col()
```



# Mapping a categorical variable to color

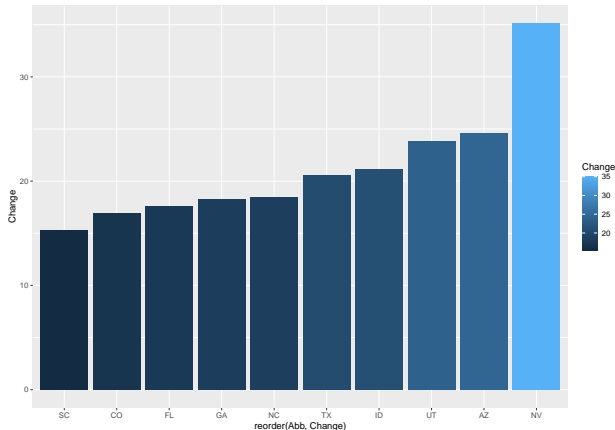
```
ggplot(upc, aes(x = Abb, y = Change, fill = Change)) +  
  geom_col()
```



# Reordering bars along the x axis

In the aes, use `x = reorder(x-variable, variable to arrange by)`

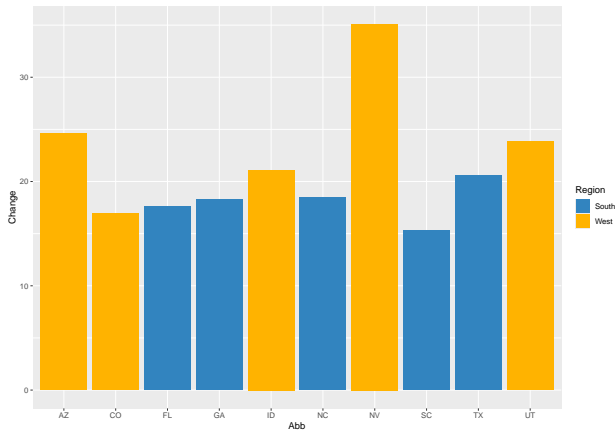
```
ggplot(upc, aes(x = reorder(Abb, Change), y = Change, fill = Change)) +  
  geom_col()
```



# Custom colors for categorical scale

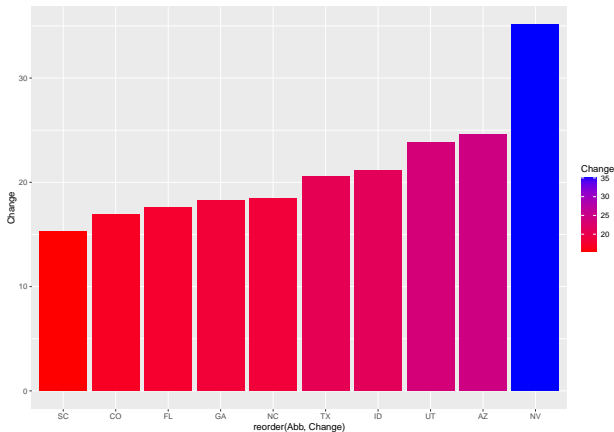
For and aesthetic mapping, you can control the colors with `scale_fill_manual`

```
ggplot(upc, aes(x = Abb, y = Change, fill = Region)) +  
  geom_col() +  
  scale_fill_manual(values = c("#3284BF", "#FFb300"))
```



# Custom Colors for continuous scale

```
ggplot(upc, aes(x = reorder(Abb, Change), y = Change, fill = Change)) +  
  geom_col() +  
  scale_fill_gradient(low = "red", high = "blue")
```



## Section 7

### Line Graphs



# Line graph basics

Use `ggplot()` with `geom_line()`, and specify what variables you mapped to x and y

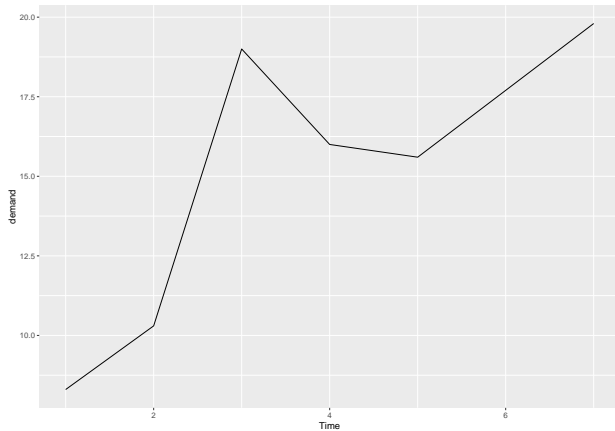
BOD

```
##   Time demand
## 1     1    8.3
## 2     2   10.3
## 3     3   19.0
## 4     4   16.0
## 5     5   15.6
## 6     7   19.8
```

```
ggplot(BOD, aes(x = Time, y = demand)) + geom_line()
```

# Line Graphs

```
ggplot(BOD, aes(x=Time, y=demand)) + geom_line()
```



# Draw multiple lines

We will summarise some tooth growth data from Guinea Pigs who were given Orange Juice or Vitamin C. We will draw one line for OJ, and one line for VC

```
tg <- ToothGrowth %>% group_by(supp,dose) %>% summarise(length = mean(len))
```

```
## `summarise()` has grouped output by 'supp'. You can override using the  
## `.groups` argument.
```

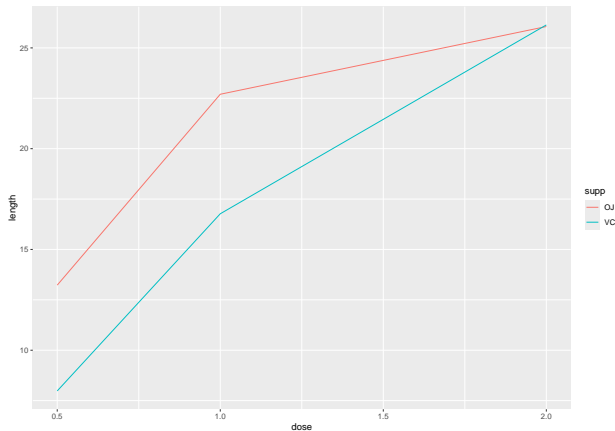
```
tg
```

```
## # A tibble: 6 x 3  
## # Groups:   supp [2]  
##   supp   dose length  
##   <fct> <dbl> <dbl>  
## 1 OJ     0.5  13.2  
## 2 OJ     1    22.7  
## 3 OJ     2    26.1  
## 4 VC     0.5   7.98  
## 5 VC     1    16.8  
## 6 VC     2    26.1
```

# Draw multiple lines

We map the supplement type to the color of the line.

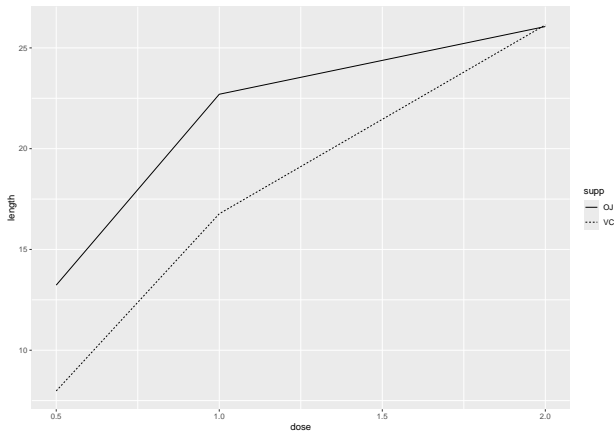
```
ggplot(tg, aes(x = dose, y = length, color = supp)) + geom_line()
```



# Draw multiple lines

We can also map the supplement type to the line type.

```
ggplot(tg, aes(x = dose, y = length, linetype = supp)) + geom_line()
```



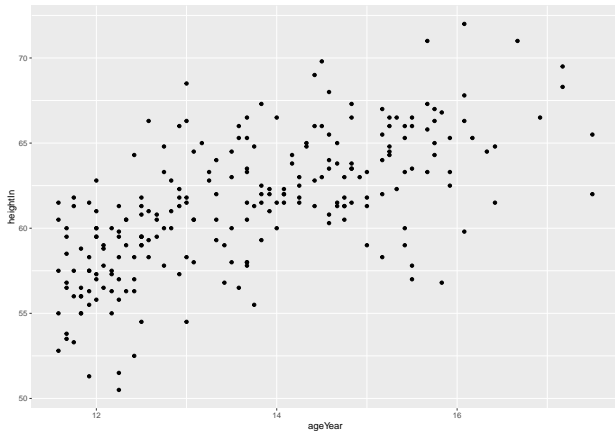
## Section 8

# Scatterplots

# Scatterplot basics

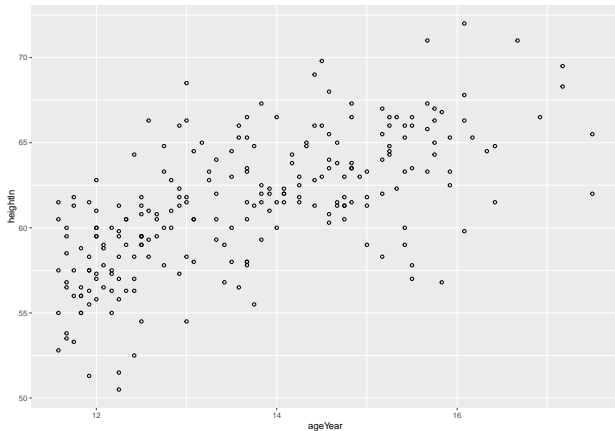
Map the x to x, the y to y, and add `geom_point()`

```
ggplot(heightweight, aes(x = ageYear, y = heightIn)) + geom_point()
```



# Different shapes (as a setting)

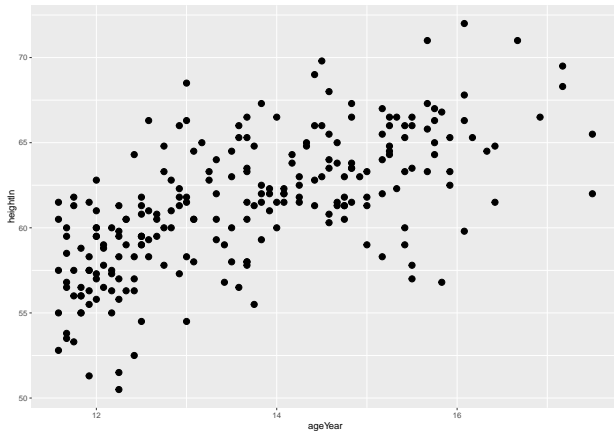
```
ggplot(heightweight, aes(x = ageYear, y = heightIn)) + geom_point(shape = 21)
```





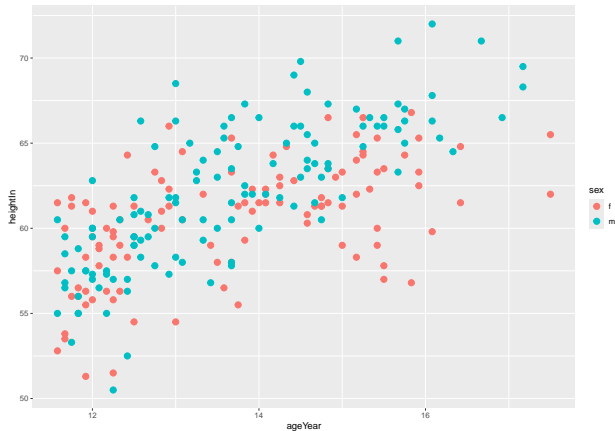
# Different size as a setting

```
ggplot(heightweight, aes(x = ageYear, y = heightIn)) + geom_point(size = 3)
```



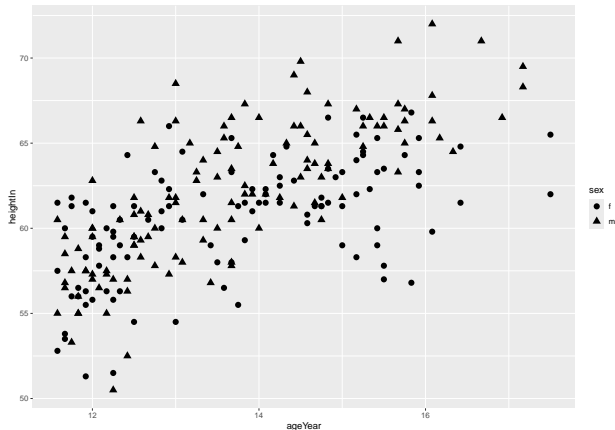
# Mapping a color to a variable

```
ggplot(heightweight, aes(x = ageYear, y = heightIn, color = sex)) +  
  geom_point(size = 3)
```



# Mapping a shape to a variable

```
ggplot(heightweight, aes(x = ageYear, y = heightIn, shape = sex)) +  
  geom_point(size = 3)
```



## Section 9

### Plotting info from multiple data frames

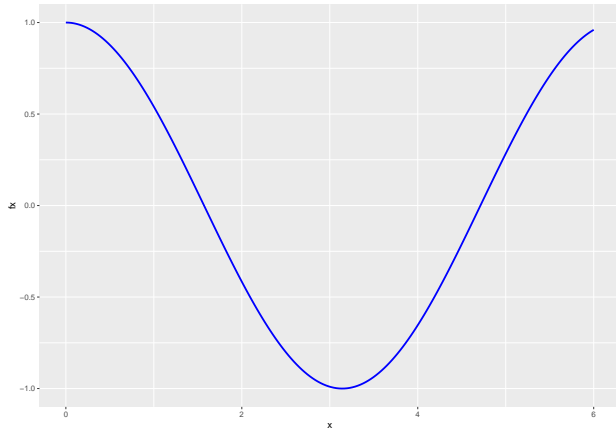
## Start with a simple line graph of $\sin(x)$

```
f1 <- function(x) cos(x)
x <- seq(0, 6, length.out = 200)
fx <- rep(NA, length(x))
for (i in seq_along(x)) {
  fx[i] <- f1(x[i])
}
function_data <- data.frame(x, fx) # the data frame for the function

p <- ggplot(function_data, aes(x = x, y = fx)) +
  geom_line(color = "blue", linewidth = 1)
```

# Simple graph

```
print(p)
```



## add some data points

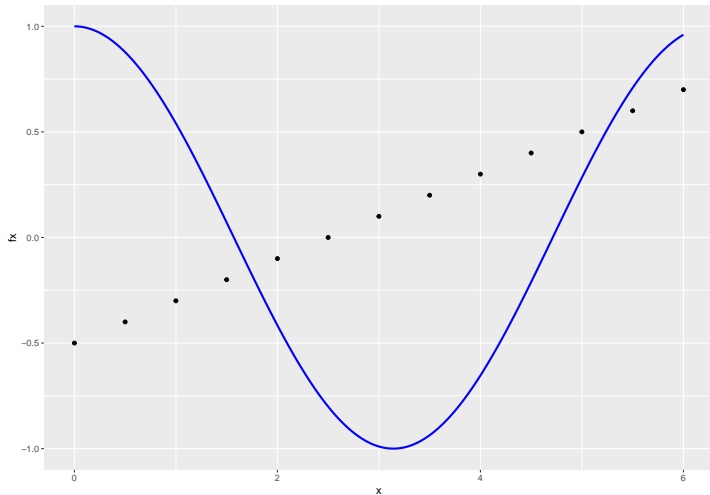
```
xp <- seq(0, 6, by = 0.5)
yp <- seq(-.5, by = 0.1, length.out = length(xp))
df_points <- data.frame(x = xp, y = yp) # the data frame for the points
print(df_points)
```

```
##      x      y
## 1  0.0 -0.5
## 2  0.5 -0.4
## 3  1.0 -0.3
## 4  1.5 -0.2
## 5  2.0 -0.1
## 6  2.5  0.0
## 7  3.0  0.1
## 8  3.5  0.2
## 9  4.0  0.3
## 10 4.5  0.4
## 11 5.0  0.5
## 12 5.5  0.6
## 13 6.0  0.7
```

```
# We have already established a plot p. We'll simply add the point geometries
p <- p + geom_point(aes(x = x, y = y), data = df_points)
```

# sine curve plus points

```
print(p)
```





# add line segments

```
# the data.frame for line segments has 4 columns for each segment:  
# a pair of x,y coords for the start and a pair for the end.  
# I'll make vertical segments by setting x2 = x1  
n <- 8  
x1 <- seq(1.25, by = 0.5, length.out = n)  
y1 <- rnorm(n, mean = 0.5, 0.2)  
x2 <- x1  
y2 <- rnorm(n, mean = -0.5, 0.1)  
df_segs <- data.frame(x1, y1, x2, y2) # the data frame for the line segments  
print(df_segs)
```

```
##      x1      y1      x2      y2  
## 1 1.25 0.5552856 1.25 -0.6077173  
## 2 1.75 0.5622886 1.75 -0.6603362  
## 3 2.25 0.3350857 2.25 -0.6262979  
## 4 2.75 0.2633777 2.75 -0.5902904  
## 5 3.25 0.3640732 3.25 -0.2174549  
## 6 3.75 0.6843514 3.75 -0.4145263  
## 7 4.25 0.6811593 4.25 -0.3756806  
## 8 4.75 0.4602919 4.75 -0.5333152
```

## sine curve plus points plus segments

```
# we add additional geometries to the existing plot (which has the sin curve and points)  
p <- p + geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2),  
                      data = df_segs)
```

# sine curve plus points plus segments

```
print(p)
```

