Unit 2: Visualizing Data

Data should ALWAYS be plotted FIRST – before you do anything else! All statistics software, including R, will very happily run analyses on your data whether it makes sense or not. You are responsible for figuring out what makes sense and the only way to do that is through exploring the data. The easiest and best way to do this is by visualizing the data in some way.

There are three major graphing methods in R: base graphics (no package required), lattice graphics, and ggplot2 (both are packages that must be installed and loaded). Lattice is a package that allows more complicated layouts of multivariate data, but we will not talk about it further here. The package ggplot2 is gaining in popularity and we will also touch on how to graph using this package.

Graphic devices

The plot area in RStudio is frequently inadequate and also frustrating if you are needing to change your pane sizes frequently. You can remedy this by putting your plot in its own, separate window. R refers to these windows (or as we'll see later, a file type you wish to plot to) as 'graphic devices'. To start a device you simply use the appropriate function with the arguments width and height to indicate size. This is one of the only places where the function name depends on whether you are using a Mac or a PC. On a Mac computer the function is `quartz()`. On a Windows machine, the function is `windows()`.

Another key piece of information before we start: remember the command **`graphics.off()`**

This function takes no arguments. It sets your graphing parameters back to default values. Whenever you are doing graphs with modifications you will want to use this command at the beginning the set of code to draw your graph. But also try using it anytime you find your graphs are not what you expect.

Base Graphics

There are three types of plotting commands in base graphics:

1. High-level – create new plots
2. Low-level – add more information to existing plots (e.g. points, lines, text)
3. Interactive – interactively add or subtract information from plots.

High-level Graph Functions

`plot()` – one of most frequently used plotting functions in R

The `plot()` function has may different type options:
Use type=

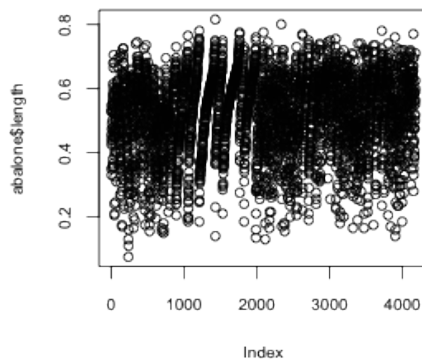
- "p" for **p**oints,
- "l" for **l**ines,
- "b" for **b**oth,
- "c" for the lines part alone of "b",
- "o" for both '**o**verplotted',
- "h" for '**h**istogram' like (or 'high-density') vertical lines,
- "n" for no plotting.

For this overview of graphs, we'll use the data called "abalone.csv"

Plot 1 variable

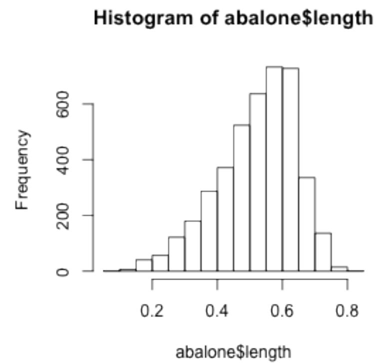
Scatter plot

```
plot(abalone$length)
```



Histogram

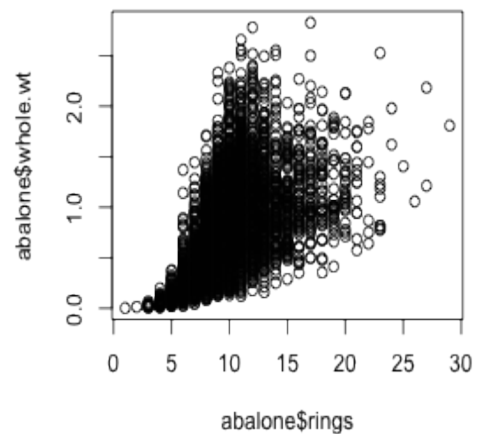
```
hist(abalone$length)
```



Plot 2 Variables

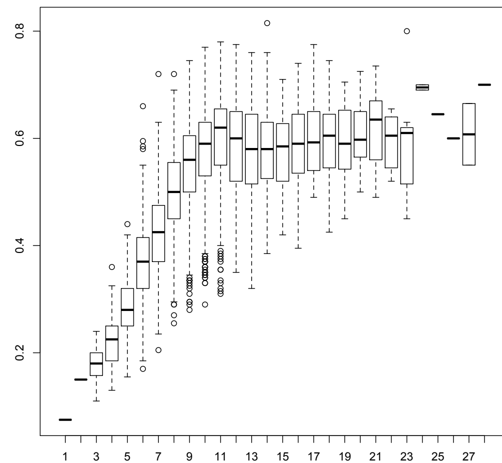
Scatter plot

```
plot(abalone$rings, abalone$whole.wt)  or
plot(abalone$whole.wt ~ abalone$rings)
```



Box plot

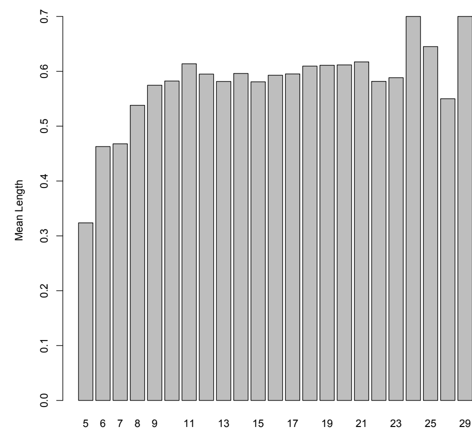
```
boxplot(abalone$length ~ abalone$rings)
```



Bar chart

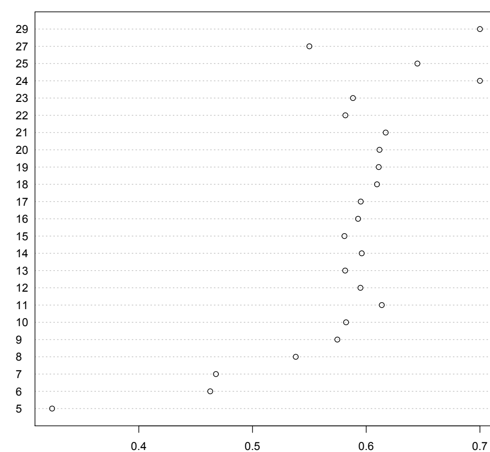
For the bar chart we'll use mean length values for each ring number; we can calculate these numbers easily from our original dataset, but for now, you can use the prepared dataset.

```
barplot(females$mean.length),
  names.arg=females$rings, ylab="Mean
  Length")
```



Dot chart

```
dotchart(females$mean.length, labels =
  females$rings)
```

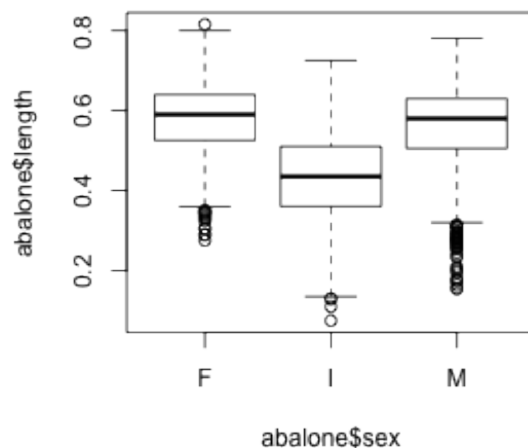


Generic Functions

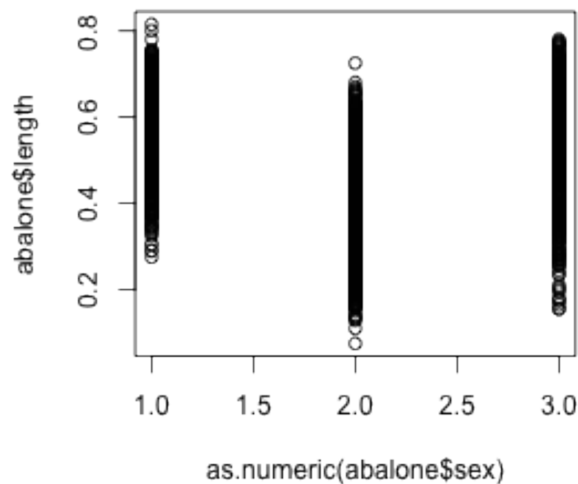
Some functions in R are what is called 'generic functions'. The output of these functions depend on the *class* of the first *argument* (those things in brackets after the function name) you provide. `plot()` is one such function and a good way to really get a feel for what this means. Test it out for yourself – the plot you get will depend on what type of data you give the function.

Note that if we give `plot()` data of class 'factor' as an x-variable, plot automatically produces a boxplot. We can force it back into a scatter plot, using the function `as.numeric()` to modify our x variable:

e.g., `plot(abalone$length ~ abalone$sex)`



versus `plot(abalone$length ~ as.numeric(abalone$sex))`

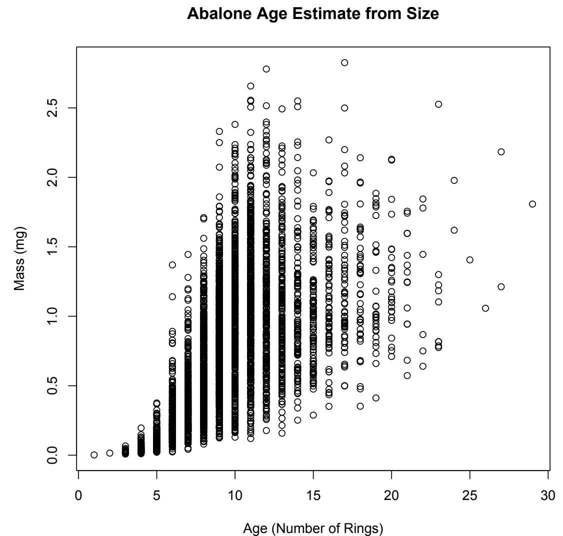


Low-level Plotting Functions (Modifying Existing Plots)

When it comes to fully customizing graphs, you will use these a lot, as the best way to customize your figure is to plot each element separately. Each of these functions only works *after* you've used one of the high-level plotting functions.

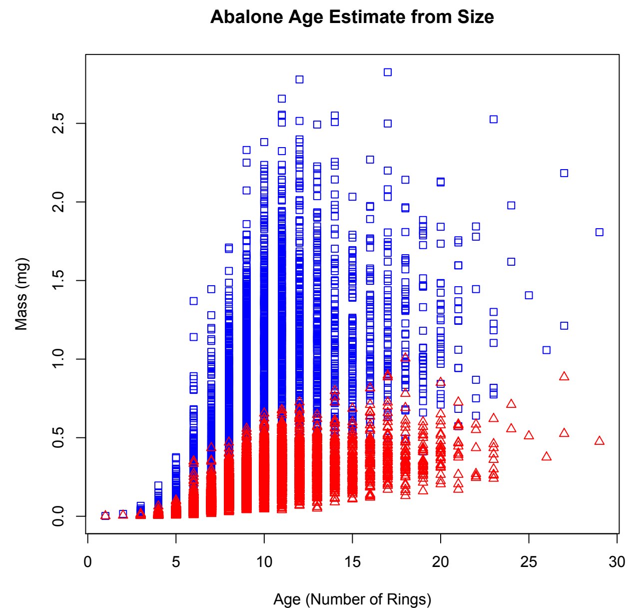
1. Add titles: **`title()`**

```
plot(abalone$rings, abalone$whole.wt, ann = FALSE)
title(main = "Abalone Age Estimate from Size", xlab = "Age (Number of Rings)" ylab = ("Length (mm)"))
```



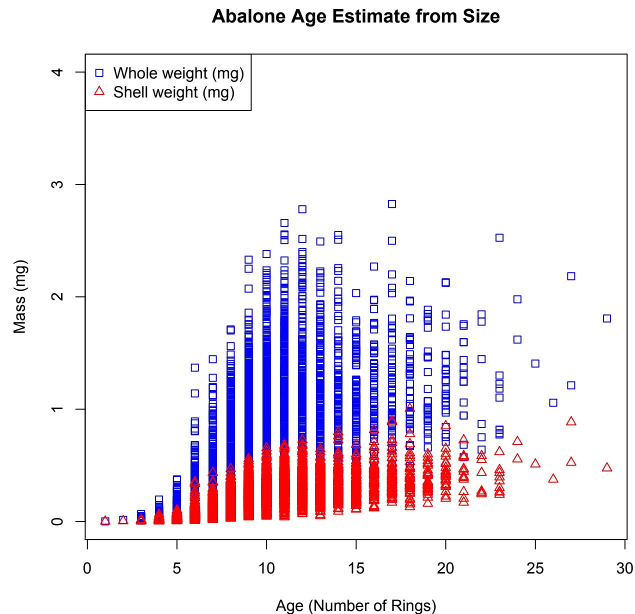
2. Add more data points: **`points()`**

```
points(abalone$rings, abalone$shell.wt,pch=2, col="red") #plot a second set of points on the same x,y scale; customize symbols and colour
```



3. Add legend: **legend()**

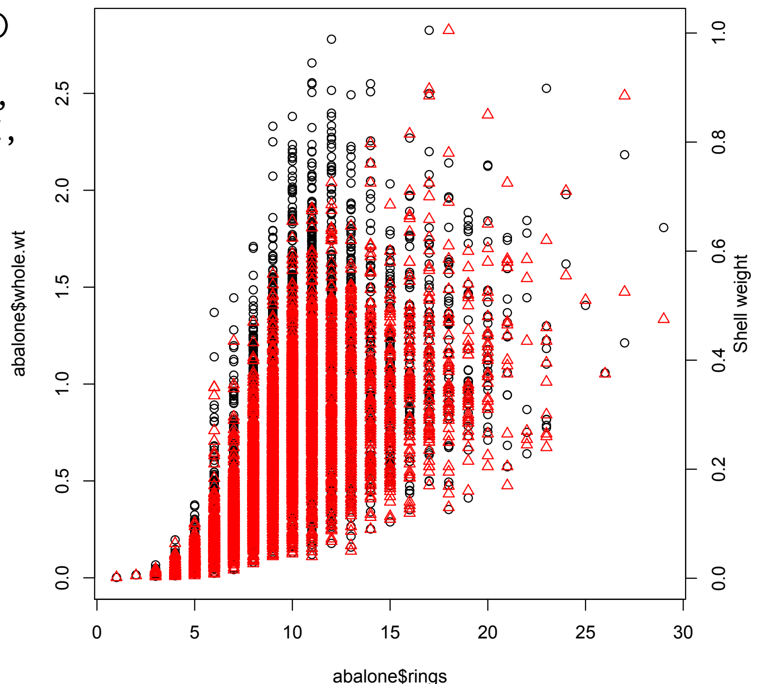
```
plot(abalone$rings, abalone$whole.wt, ann = FALSE, pch=0, col="blue", ylim
    = c(0, 4)) #change y-axis to make room for legend
title(main = "Abalone Age Estimate from Size", xlab = "Age (Number of
    Rings)", ylab = ("Mass (mg)"))
points(abalone$rings, abalone$shell.wt, pch = 2, col = "red")
legend("topleft", legend=c("Whole weight (mg)", "Shell weight (mg)"),
    pch=c(0,2), col=c("blue","red"))  # draw legend to identify different
    points
```



4. Add another axis: **axis()** add axis to plot; placement of axis on plot is coded as 1 (bottom), 2 (left), 3 (top), 4 (right)

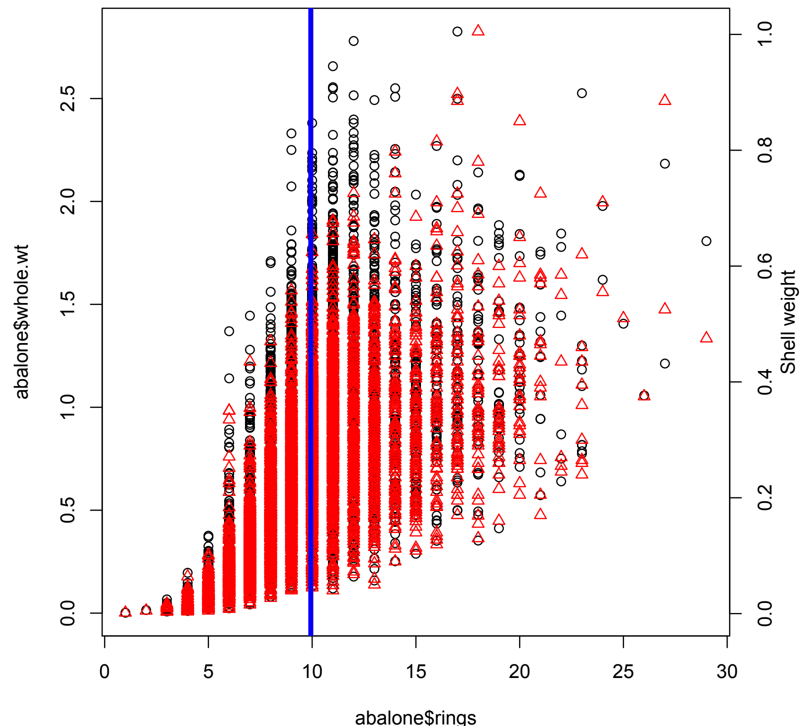
```
plot(abalone$rings, abalone$whole.wt)
par(new = TRUE)
plot(abalone$rings, abalone$shell.wt,
    pch = 2, col = "red", ann = FALSE,
    axes = FALSE)
axis(4)
mtext("Shell weight",4, line=2)
```

for mtext, give it the side and how many lines out on that side you want to place text (line=0 will print it right at the axis line)



5. Add lines:
   a. **lines()** - connect points
   b. **segments()** – add a straight line between two points; takes values (x0,y0,x1,y1) to draw line from (x0,y0) to (x1,y1)
   c. **abline()** – add straight line; takes intercept & slope of line; use h=value for horizontal line at y=value and v=value for vertical line at x=value
   d. **arrows()** - this function draws arrows between pairs of points. You will use it often as it works to add in error bars. We'll come back to that in the next section.

   e.g. `abline(v=mean(abalone$rings),col="blue",lwd=4)`



> *To plot line of best fit:*
>
> *Use either*
>
> `abline(lm(y~x))` *OR*
>
> `curve(intercept + b*x, add=TRUE)` *# curve is high-level plotting functio*

6. Add text: **text()** – add text in plotting area; **mtext()** – add text to one of the margins (as we saw above in #4).

   `mtext("Shell Weight", side = 4, line = 3)`

The trick with adding text to your graph is remembering that text coordinates are always in the same units as the graph axes. Use the graph axes to identify where you wish to place text.

Interactive Plotting Functions

A handy interactive plotting function that you are likely to use is `identify()`.

This function lets you use your mouse to click on a point (or several points) on a graph have it labelled.

Customizing Plots: Graphical Parameters

`par()` – use this command with no arguments to see what the global settings are for your graphical parameters. Also get used to looking this up in the help regularly, as it spells out all the possible parameter options there.

There are many, many graphical parameters. Some you will set separately, using the `par()` function and sending arguments to it, others you will add in as arguments to some other functions.

e.g. Suppose you want to put 4 plots on a single page (in a single window) – you can do this by giving the par() option mfrow a matrix: `par(mfrow = c(2,2))` before using your plot() commands

e.g. Alternatively, when you are customizing the plotting symbol ('pch' option in `par()`) you are likely to put it specifically into the function `points()`. If you set it both globally (outside of any other function) and within a function, the value within the function will override the global value for that instance only.

We will spend much more time talking about these in a future session of the workshop!

par() options that can be used directly in plot():

ann = (annotation); TRUE (include plot annotations) or FALSE (do not include)

cex = (relative size of text and symbols); default =1, takes a number

col = (plotting colour); R accepts color names, html color codes
Use `colors()` *to get info on built-in R colors.*

font = (font style); 1 = plain, 2 = bold, 3 = italic, 4 = bold italic

las = (text direction); 0 = always parallel to axis, 1 = always horizontal, 2 = always perpendicular to axis , 3 = always vertical

lty = (line type); 0 = blank, 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, etc.

lwd = (line width); default = 1, takes a number

mar = (margins); takes a vector of 4 numbers - c(bottom, left, top, right) - indicating the size of margins (in lines)

pch = (symbol type); use ?pch to get info on which numbers of this argument give which symbol

Practice. Spend some time playing around with some of these. Load the airquality data, using `data(airquality)`.

1. Create a new variable in the data set that will let you plot ozone over time & make this plot.
2. Add lines to the plot.
3. Add temperature to the plot, using a second axis with its own scale.
4. Put the temperature axis on the right-hand side of the plot and label it.
5. Re-plot showing temperature only as a line plot, with the line coloured red.

ggplot2

A graphics package based on the principles of a grammar of graphics – principles or rules that we can apply to plots, analogous to the grammar rules of a language.

This is a flexible graphing package that makes it easier to produce good plots without having to program every single detail. But I think in the end that is a little less flexible than base graphics.

Using ggplot2 does require a bit of a mental shift in the way you think about making plots, including getting familiar with some terminology:

- Data – what we want to visualize; for ggplot() data *must* be in a data.frame in a *long* format
- Geoms – geometric objects that represent data such as points, bars, lines
- Aesthetics –visual properties of geoms, e.g (x,y) position, symbol
- Scales – scaling of graph aesthetics
- Mappings – connect data values to aesthetic

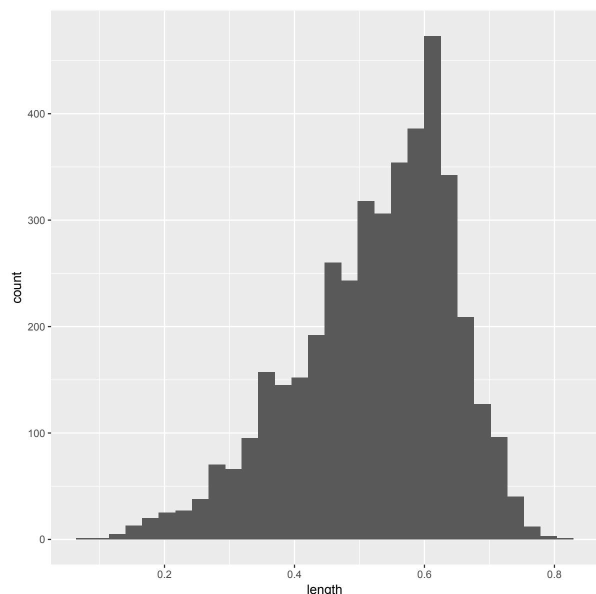
You can get more information about using ggplot2 from the website: ggplot.org

An even better source of help for learning gglplot2 is the book *R Graphics Cookbook*; see the website: www.cookbook-r.com/Graphs/

We'll use the same data as above with the base graphics to produce the same types of plots using ggplot2.

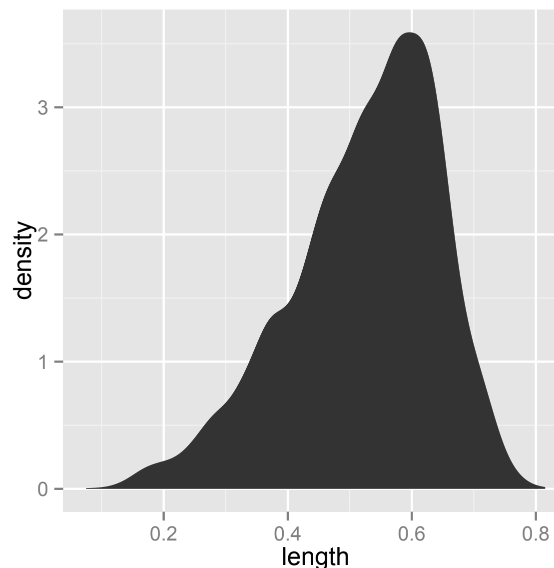
Plot histogram

```
ggplot(abalone, aes(x=length)) +
geom_histogram()
```



Plot density curve

```
ggplot(abalone, aes(x=length)) + stat_density()
```



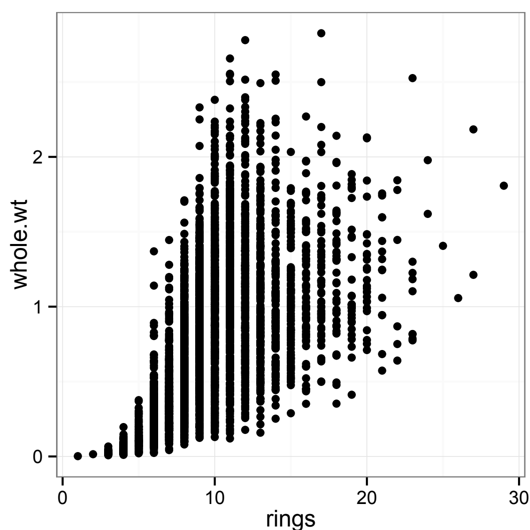
While this code may seem very awkward at first,
one of the benefits can be seen immediately in the ability to assign ggplot() output to an object that is then easily modified, e.g.

```
p <- ggplot(abalone, aes(x=length))
p + geom_histogram()  # to produce histogram
p + stat_density() # to produce density plot
```

Plot 2 Variables

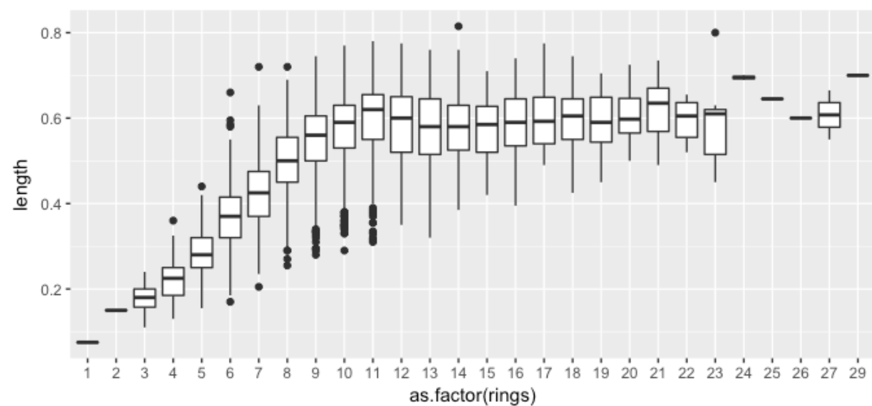
Scatterplot

```
ggplot(abalone, aes(x = rings, y = whole.wt)) + geom_point()
```



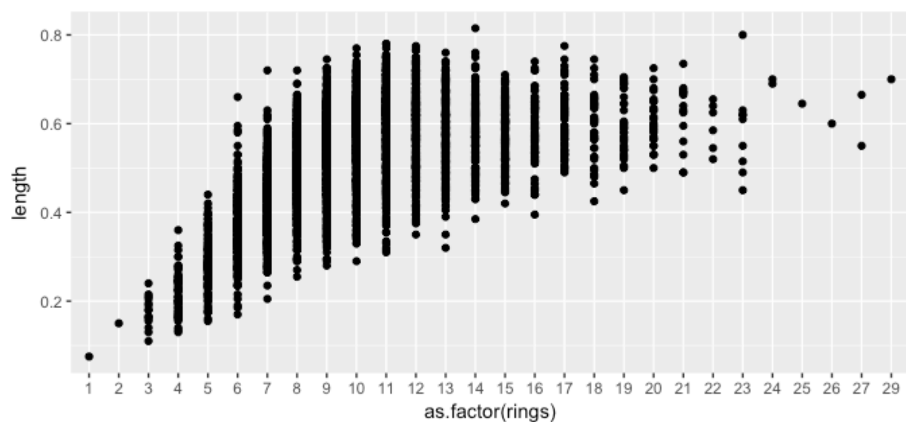
Boxplot

```
p <- ggplot(abalone,(aes(x = as.factor(rings, y = length)))
p + geom_boxplot()
```



Force scatter plot

```
p + geom_point()
```



Practice 2. Let's practice putting together some of the tools we've learned. Start by creating a dataframe.

```
dat <- data.frame("xval" = 1:4, "yval" = c(3,5,6,9),
"group"=c("A","B","A","B"))
```

1. Use ggplot() to plot points.
2. Plot points so each group has its own color.