BIOE286 Lab A

Introduction to R

Welcome to BIOE186/286! We are so excited to share a ton of tools and tips and tricks for programming in R. In this first lab, we'll go from the very basics (1+1) to making time-series maps.

In this course, you will work with various functions. Some are built-in (like `mean()` which calculates the average) but others were written into a package. To use those functions you first have to install the packages using `install.packages()`. Once you do this on your computer, you don't have to do it again (until you get a new computer). Once the package is installed, you must load the package (using `library()`) and only after it has been loaded can you use all the functions and datasets it contains. To start, let's load in three packages: `tidyverse` which has lots of great data manipulation tools, `readxl` which helps you input excel files into R, and `ggplot2` which helps you make pretty plots.

(Note: here I'm "commenting out" aka putting a pound sign in front of the `install.packages()` call because I've already installed these packages on my computer)

```
#install.packages("tidyverse")
#install.packages("readxl")
#install.packages("ggplot2")
library(tidyverse)
library(readxl)
library(ggplot2)
```

(PS: If your package installs aren't working, try updating R. You should be able to use `install.packages("installr"); library(installr); updateR()`).

Now that you've installed and loaded all those packages, we should be good to go. Let's start with some basic operations.

1) Basic operations

R can do basic calculations like `1+1`. If you want to store a number (e.g., 1) as a variable (e.g., t) you would type `t=1`. You can use the same approach to save the output of a calculation: `t=1+1`.

To create a vector of numbers from scratch, we have two options. First we can use the `:` symbol which says "to" (e.g., `1:10` gives you the integers one through ten). Alternatively we can use the built-in function `c()` which sticks a bunch of numbers together. The c stands for concatenate.

(Note: for coding best practice, try to make comments on many of your code lines, so you know what each line does! I'll try to role model this throughout labs by putting a comment at the end of a line [like below] or above a line. You should get in the habit of doing the same in your code).

```
1:10 # here's a numeric vector
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
c(1,1,3,5,10) #using concatenate
```

```
## [1]  1  1  3  5 10
```

Remember, the lines of code above are just printing the numbers but not actually storing them. To store those numbers as a variable, we would use:

```r
A=1:10 # here's a numeric vector
B=c(1,1,3,5,10) #using concatenate
```

Be careful! R overwrites variables without even asking you. So if instead of setting the first line as A and the second line as B, you were to run the following code...

```r
A=1:10 # here's a numeric vector
A=c(1,1,3,5,10) #using concatenate
```

...R would set A to the numbers `1:10` and then replace them with the vector of `c(1,1,3,5,10)` such that A would equal `c(1,1,3,5,10)` and `1:10` would be nowhere to be found.

To check what the variable `A` contains, you can either click on it (in the Global Environment in the top right), or type `A` or `print(A)` into the console.

Great! So we now have a variable `A` that equals `c(1,1,3,5,10)`.

2) Functions

Let's learn about some of the built-in functions that can help us calculate descriptive statistics like the sum, length, mean, and standard deviation of that vector:

```r
sum(A) #sum
```

```
## [1] 20
```

```r
length(A) #number of values
```

```
## [1] 5
```

```r
mean(A) #average
```

```
## [1] 4
```

```r
sd(A) #standard deviation
```

```
## [1] 3.741657
```

```r
max(A) #highest number
```

```
## [1] 10
```

```r
min(A) #lowest number
```

```
## [1] 1
```

Remember that you can store any of those outputs as a variable using for example `Asum = sum(A)`.

In R, it is really easy to do basic calculations from existing variables. For example, typing `A+5` adds five to every number within the vector `A`. Like Excel and other programs, + is addition, - is subtraction, * is multiplication, and / is division. In programming languages, the number 5 in `A+5` is called a "scalar"; a scalar is just a single value like a number or a name.

Other random fun built-in functions include:

- `help()` which brings up the help file / documentation for a function (e.g., try typing in `help(mean)`)
- `unique()` which gives a vector of unique scalars within an object
- `rev()` which reverses the order of something
- `seq()` which gives you a sequence of numbers. Remember, you can always check the help file / documentation if you don't know what a function needs. If you type `help(seq)`, you will see that the function requires three inputs: from, to, by. This function produces a sequence of numbers fom something, to something, by something. So if you wanted to create a vector of numbers from 1 to 10 by 0.2, you would type: `seq(from=1,to=10,by=0.2)`. This gives you 1.0, 1.2, 1.4, etc all the way up to 10.
- `rnorm()` which makes a vector of random numbers with a normal distribution. The required inputs are n, mean, sd. For example, `rnorm(n=100,mean=0,sd=.1)` will give you a vector of 100 random numbers with mean of zero and standard deviation of 0.1.

Go ahead and play around with those built-in functions. As always, let us know if you have questions!

When you feel comfortable with those functions, go ahead and run the following code to make an object called `rand` with n=100, mean=0, and sd=.1:

```r
rand=rnorm(n=100,mean=0,sd=.1)
```

You can check your global Environment and see that there is now an object called `rand` that is comprised of 100 numbers. But what if you want to make sure the code worked properly (i.e., spit out a set of numbers with a random distribution?)?

You could plot a histogram. Before giving you code to do that, I want to quickly divert to a story of the two "camps" for plotting in R.

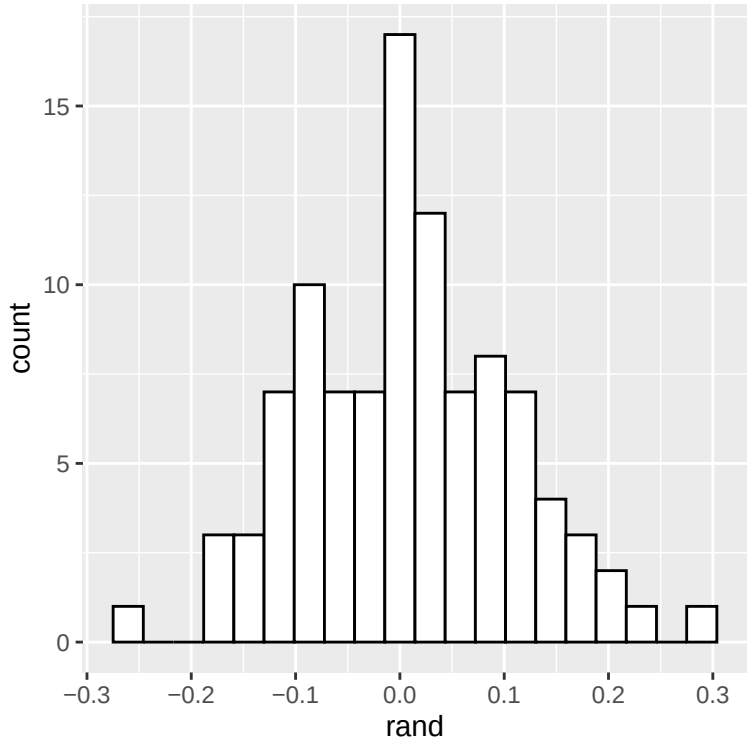
3) Plotting

Once upon a time, when R was first created, there was one way to make plots - using the built-in functions. Then someone came along and created a package called `ggplot` that contains functions for more intuitive plotting. Fast forward to today, and there are two "camps" of people who disagree ferociously about which is better. There are benefits and drawbacks to both, and we're not going to say that one is better than the other; *they are both great.*

I (Roxanne) taught myself R as an undergrad because I didn't know anyone who used it, and I taught myself base (rather than ggplot) because ggplot wasn't really mainstream yet. I have been slowly transitioning to ggplot over the last 10ish years and now mostly use ggplot. We're going to mostly use ggplot in this class (with a few exceptions, when base might be better for something). For those of you who use and love ggplot - you are very welcome to teach me new things as we go through the materials - I am still learning a lot! :)

OK, back to plotting. Just for fun, let's plot a histogram in ggplot.

```
ggplot()+aes(rand)+geom_histogram(bins=20,colour="black",fill="white") #ggplot
```




```
#don't worry too much about the aes() part of the code... we'll get there later...
```

See how we've built the ggplot using a series of + signs? Try to get in the habit of making a new line of code after each plus sign.... i.e., your code should look like this (I'm commenting out the code so it doesn't print the plot again and waste space in this PDF):

```
#ggplot()+
#  aes(rand)+
#  geom_histogram(bins=20,colour="black",fill="white") #ggplot
```

Now, let's say we want to subset some data. For a vector (which is only one-dimensional), you just need to provide one value in square brackets (not round parentheses). To pull out the second value of A, you would type `A[2]` and if you wanted to pull out the last value of A, you would type `A[length(A)]` (this basically combines two chunks of code: `length(A)` which tells you that A contains 5 values, and `A[5]` which tells you to pull out the last (5th) value of A. Like before, we can also do calculations with these numbers, like adding 2 to the 5th value of A using `A[5]+2`.

I've said this a million times and I'll say it again - it is super important to keep your workspace clean. This becomes especially important later (e.g., when you are using forloops to loop through data and doing fancy calculations) but it's great to get into good data habits. For example, if we look into our Global Environments (or alternative type `ls()` to list the objects in our Global Environment) we have two objects: `A` and `rand`. We're done with `rand` for now, so we can remove it using `rm(rand)`.

You can clear the entire Global Environment using the same logic and combining the functions: `rm(list=ls())`. This says "remove everything in my Global Environment".

Cool, so you have a better idea for how we can create variables from scratch and do calculations. Now, let's import some data!

4) Working directory

File organization is arguably the most important part of coding in R. Your *working directory* is the specific folder in which you are working. You will be very happy with yourself if you make time AT THE BEGINNING to organize your files and folders.

R, like many other programs, has to be able to find the files referenced in the code. In other words, if you write an R script that loads in a dataset, and then you move that dataset, the R script will no longer work (it will create an error). Trust me, you don't want to deal with this over and over. It took me way too long to figure out that *taking shortcuts by working from files downloaded onto my desktop will only make life harder in the end*. Seriously, this is not ideal.

Before we even get started today, we recommend setting up a nice, organized structure of folders that you will use to store the materials from this class. Each lab session, you will download a file with the lab activity (.pdf or .html) as well as the raw datasets (mostly .csv) you need for the activities. We recommend creating folders for each lab, so that the lab activity, datasets, and your resulting files (R code and figures) will be saved in a single, orgainzed place.

In real life (outside of class), you will most likely be entering your raw data (or downloading your data from a tag or instrument) in Microsoft Excel which saves workbooks as .xlsx files. R doesn't like these files because they contain multiple tabs, so the best way to create files for R is to open your workbooks in excel and go to File > Save As > Comma Separated Values (.csv) files. (Note that when you convert a .xlsx to a .csv you lose your formatting and equations, so be careful with this).

As reproducible science continues to gain momentum, one of the best things you can do for yourself (and for others) is to create code and files that are clear and organized. You should be able to go from raw data (collected in the field or lab) to publication-quality figures and analyses without ever manipulating data in a program like Excel. No more arbitrary deleting of outliers, no more filtering, no more manual calculations. These things all lack transparency (i.e., no one else knows what you did to your data, and you will probably forget anyways) and reproducibility (i.e., no one can re-create a figure you've made). In R, it is relatively easy to create reproducible workflows.

After you have organized your files (outside of R), you can set your working directory to the folder that contains the data for this lab (Lab A). There are several ways you can do this:

You can type it in manually:

```
setwd("C:/Users/roxan/Documents/Teaching/Current Courses/BIOE 186 286 Exp Design and Stats/BIOE286 2025
```

Or even better, you can go to Session > Set Working Directory > Choose Directory and then click through your folders until you find the one you are looking for. Click OK, and you will see that new code (matching the structure of the code above) was added to the Console. Make sure to copy and paste this to your source code so that it is there next time you open and run the source code.

Fun fact, you can check your working directory any time using `getwd()`.

FYI, in all of these RMarkdown PDFs that I'll give you, I need to set my working directory in this same way to load in the files etc. The working directories are often super long and get cut off in the PDF that you'll get. Don't worry! You just need to replace that line of code with your own working directory, so you can ignore mine.

Also, a side note - it doesn't make sense for us to use RProjects in this class, but I (Roxanne) use them in real life, and I find them extremely useful. Google them or ask us about them if you are interested - highly recommend.

5) Importing data

AWESOME. Let's load in some data (thanks to Tanya Rodgers for creating these fake data).

```
d = read.csv("example data.csv")
```

This creates a dataframe called `d`. Whenever I import data into R, I always check it over to make sure that everything looks OK (and so that I know what I'm working with). Here are some commonly used built-in functions:

- `head(d)` to view top of data
- `tail(d)` to view bottom of data
- `str(d)` to see the data structure
- `summary(d)` to get a summary which varies depending on the Class of each column
- `nrow(d)` to see the number of rows
- `ncol(d)` to see the number of columns
- `colnames(d)` to extract the column names

Go ahead and give those functions a try. You will see that this dataset contains count data of `mussels`, `snails`, and `barnacles` across two treatments, `site` and `tideheight`.

In this class, we'll do our best to give you the natural history information you need to understand the dataset. Here, the numbers of mussels, snails, and barnacles were measured during low or high tide at three different sites (A, B, and C). You might wonder whether carnivorous snails eat barnacles, and whether barnacles and mussels compete with eachother for space in the intertidal. Imagine that different sites have different conditions (like the amount of sunlight or air temperature) and that high versus low tide might impact species distributions or behaviors.

Remember that you can refer to specific columns using the dollar sign like `d$mussels`. You can also use many of these built-in functions (e.g., `summary()`) with single columns (e.g., `summary(d$mussels)`).

You can easily add a column to the existing dataframe. Let's say you want to add the number of mussels, snails, and barnacles together into a column called `total`. You would type: `d$total=d$mussels+d$snails+d$barnacles`.

Matrices have an extra dimension (both rows and columns) as compared to vectors (single string of numbers), which means they are a bit more complicated. Nothing you can't handle. There are several ways to do the same thing (which can make learning R kind of hard - just keep in mind that whatever function you use and whichever way you decide to use it, you should *always* check every step of your code to make sure it is doing what you think it is doing).

Let's say we want to extract the data in the `mussels` column. We could do this three different ways:

- `d$mussels`
- `d[,"mussels"]`
- `d[ ,3]`

In plain English, these three bits of code say: give me the mussels column within d, give me the mussels column within d, and give me the third column. I wouldn't recommend using the third option because it is very easy to mistakenly miscount columns!

If you want to select a specific row and column, you put the row and column numbers in square brackets after the name of the dataframe (e.g., `d[1,3]` pulls out the value in the first row and the third column). Again, I would recommend using column names instead of numbers when possible, so instead of `d[1,3]` you would use `d[1,"mussels"]` (don't trust me? try it for yourself - you will get the same answer). If you want to select an entire row, you use the same notation but leave a blank space *after* the comma (e.g., `d[1,]` would extract the first row).

Finally, we could subset a dataframe based on the values. This is best done using the built-in `subset()` function, which requires you to use the "==" notation to check whether something is true. For example if we

wanted to pull out the instances where `tideheight` is low, we would use `subset(d,tideheight=="low")`. In plain English, this line of code says "subset the dataset d where the tide height is low". We can use the ampersand (&) symbol to specify that we want to subset the instances where `tideheight` is low AND `site` is A, using: `subset(d, tideheight=="low" & site=="A")`.

Instead of just asking R to give you a subset of the data `d` where `tideheight` is low, you could ask R to give you a subset of the data `d` where there are more than 6 mussels (`subset(d,mussels>6)`) or a subset of the data `d` where there are less than or equal to 4 snails (`subset(d,snails<=4)`).

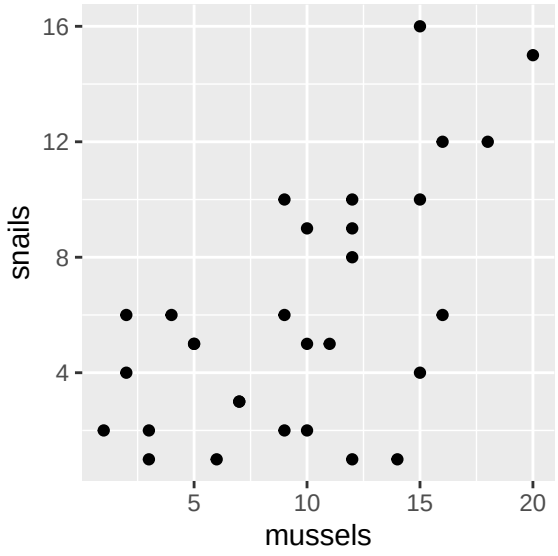
One other thing to try. In R, `==` means "is equal to" and `!=` means "is not equal to". So `subset(d,tideheight!="low")` gives you a subset of the data `d` where tide height is not equal to "low".

Try these sub-setting strategies out for yourself with some new numbers!

6) More plots

Let's say we want to plot the number of snails as a function of the number of mussels. Because they are both continuous variables, we would use:

```
ggplot(data = d,mapping = aes(x = mussels, y = snails)) +
  geom_point()
```

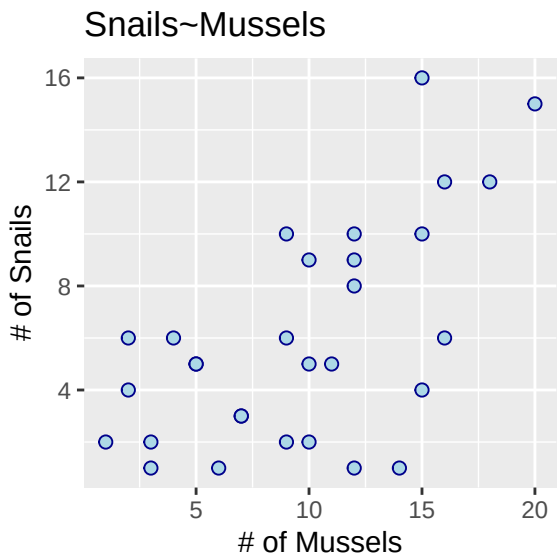


Here's an example of how to add lots of things to your base plot to make it prettier (yes, this is very important):

One quick note... R has a lot of data point shapes (circles, squares, triangles, etc). Search Google for "r pch" and you'll see what we mean. The default is pch=20, which is a solid circle with just a single color, so if you don't specify a pch, it'll just assume you want that one. If you choose some of the other shapes, like pch=21 (a circle that has BOTH an outline color and a fill color), you'll be able to separately set colors for the outline and the fill, like I've done below.

```
ggplot(data = d, mapping = aes(x = mussels, y = snails)) +
  geom_point(pch=21, #sets the point type to "21" which is circles with outlines
             fill="lightblue",col="darkblue",#change fill and outline colors of points
```

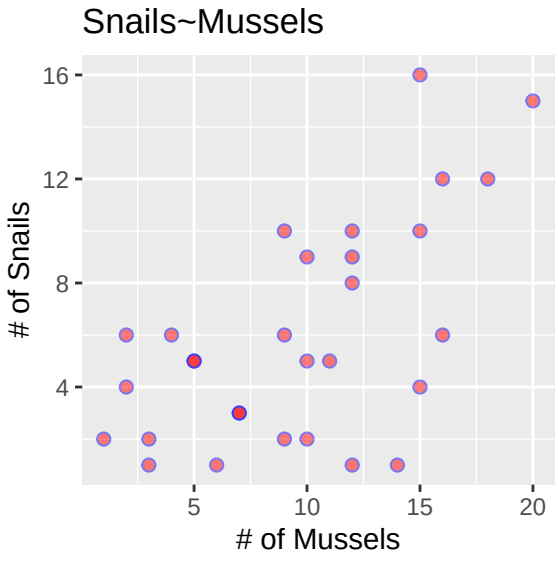
```
            size=2)+ #change size of points
  labs(x="# of Mussels",y="# of Snails",title="Snails~Mussels") #change labels
```



Here you see I've used simple built-in colors like "light blue" and "dark blue". Instead, you can use the built-in function `rgb()` which gives you a color based on the intensity of the 3 primary colors: red, green and blue (in that order). Check the help file `help(rgb)` to see a list of the arguments (inputs to the function) required. Long story short, the function requires values for red, green, and blue. Each number should be a decimal between 0 and 1. If you wanted to create the color red, you would type `rgb(1,0,0)` (which basically says give me a color with 100% red, 0% green, and 0% blue). The color white is `rgb(1,1,1)` and the color black is `rgb(0,0,0)`. You can literally create any color of the rainbow. Remember back to elementary school when you learned that the color brown is basically half green and half red? `rgb(.5,.5,0)`. Boom.

If you want to be super extra fancy, you can add a fourth argument to the `rgb()` function for transparency (again, decimal number between 0 and 1, but this time 0 is completely transparent and 1 is completely opaque). So, if you wanted to create a half-way see-through red point you would use `rgb(1,0,0,0.5)`. Another way to write this same thing (in a less efficient but easier to understand way) is `rgb(red=1,green=0,blue=0,alpha=.5)`. Let's try it out!

```
ggplot(data = d, mapping = aes(x = mussels, y = snails)) +
  geom_point(pch=21,#sets the point type to "21" which is circles with outlines
             fill=rgb(1,0,0,.5), #fill is half transparent red
             col=rgb(0,0,1,.5), #outline color is half transparent blue
             size=2)+ #change size of points
  labs(x="# of Mussels",y="# of Snails",title="Snails~Mussels") #change labels
```

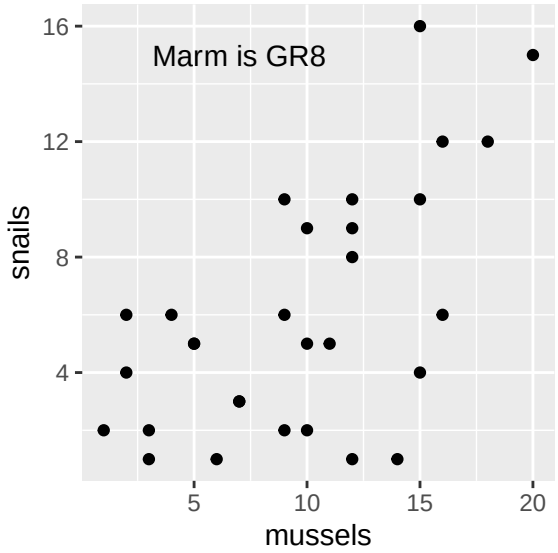


See how a couple of the points are darker red than the others? Any idea why? If not, phone a friend!

Note that the alpha settings for transparency are super handy when you have a ton of overlapping points, but not so much in the graph of snails~mussels. In later labs, I will show you how to use color palettes (pre-determined groups of colors).

One more thing I want to show you ... how to add text annotations! You can set the label text manually:

```
ggplot(data = d,mapping = aes(x = mussels, y = snails)) +
  geom_point()+
  annotate("text",x=7,y=15,label="Marm is GR8")
```

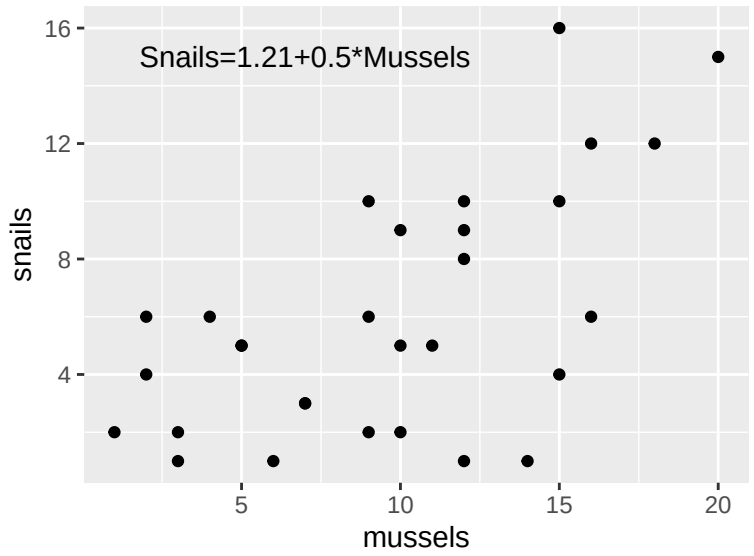


You can also, of course, add text from an object. Let's say that we want to fit a linear model to this dataset. We haven't taught you about assumptions, etc yet so do not read too much into this linear model example (like, literally please ignore it).... I just want to show you a proof of concept for how you would pull out the equation from a line of best fit and then overlay it onto a plot.

```
res=summary(lm(snails~mussels,data=d))
```

Here's the syntax to make the same plot in ggplot, including the annotation:

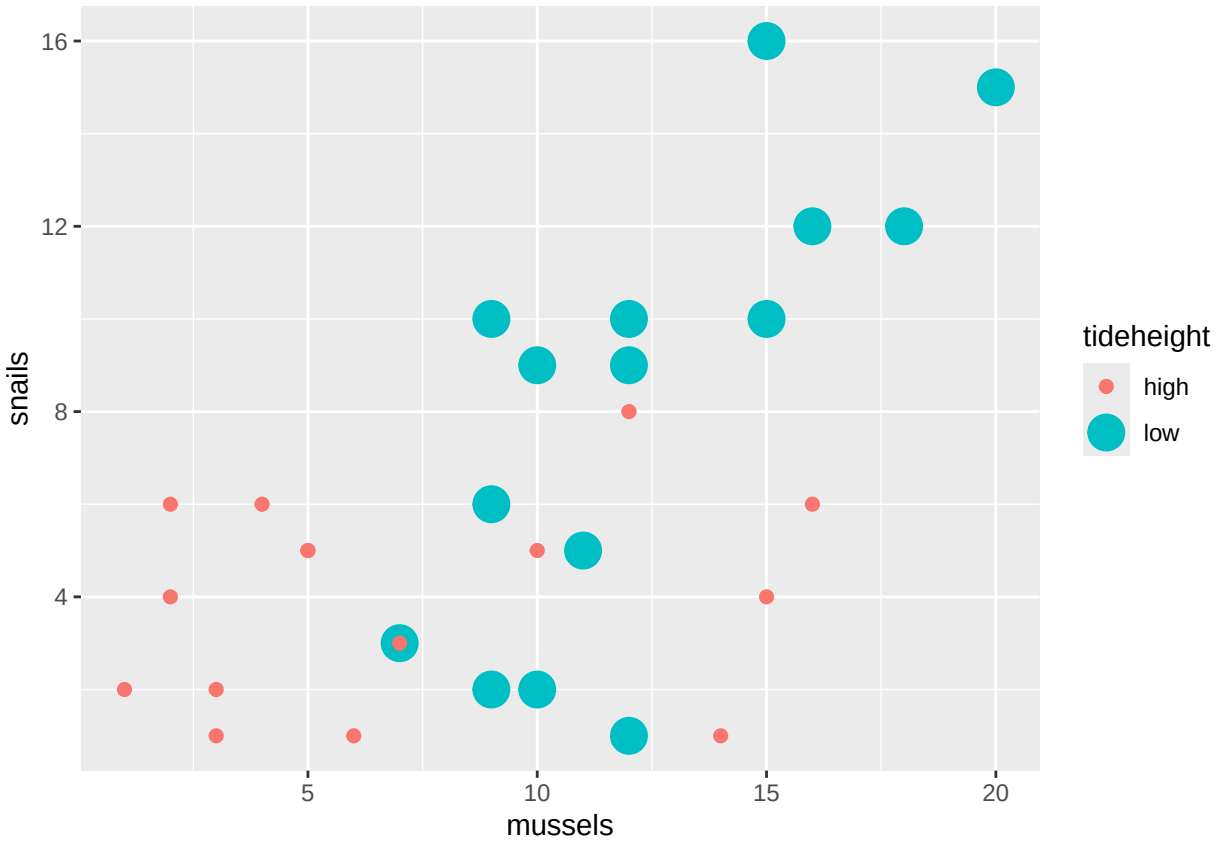
```
ggplot(data = d,mapping = aes(x = mussels, y = snails)) +
  geom_point()+
  annotate("text",x=7,y=15,label=paste0("Snails=",round(coef(res)[1],digits=2),"+",
                            round(coef(res)[2],digits=2),"*Mussels"))
```



Don't panic! I know the `label=...` line is kind of nuts. Throughout the quarter we'll help you break down complex lines of code. In general, you can decompose chunks of code, starting in the inside, and working your way out. Let's take for example the part of that line that says "`round(coef(res)[2],digits=2)`". What does this do? It takes `res` (the summary linear model output), finds the second coefficient (using the `coef()` function and then the square bracket for subsetting the second element), then rounds it to two digits. You can run pieces of the code, like `coef(res)`, in your console to see what comes out. You'll see that the beginning of that line of code has the `paste0()` function which takes stuff (separated by commas) and pastes it together with no (zero) separation. So, that line of code says "make an object called `mylabel` that is a pasted together character string of Snails=Intercept+Slope*Mussels, except the intercept and slope values come from the model fit". Make sense? Ask if that is unclear!

The ggplot package makes it really easy to add in other pieces of information from the dataset. For example, we can color and size the points by tide height:

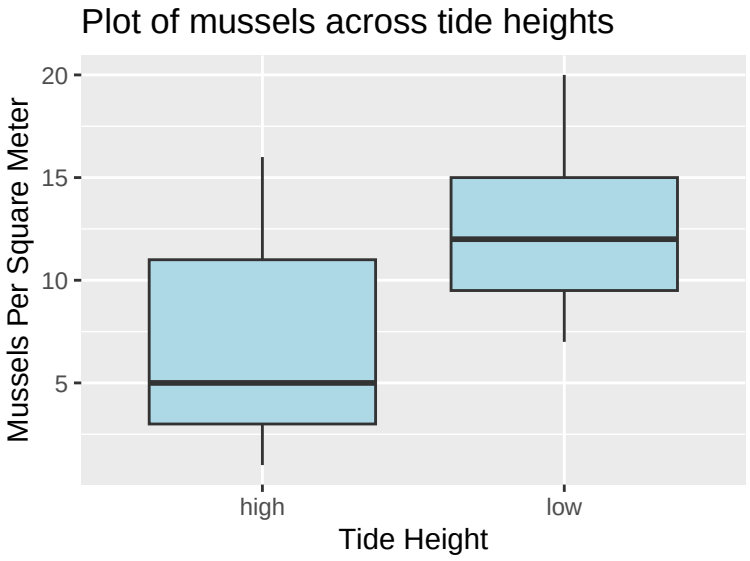
```
ggplot(data = d,mapping = aes(x = mussels, y = snails)) +
  geom_point(aes(color=tideheight,
                 size=tideheight))
```



That sort of thing would be much less efficient to code in base R.

All of these plot characteristics are the same across most plot types in R. Let's says we want to create a boxplot of `mussel` counts (an integer value) as a function of `tideheight` (a factor). We would use the `geom_boxplot()` function instead of `geom_point()`.

```
ggplot(data=d, aes(x=tideheight, y=mussels)) +
  geom_boxplot(fill="light blue")+
  labs(title="Plot of mussels across tide heights",
       x="Tide Height",
       y = "Mussels Per Square Meter")
```



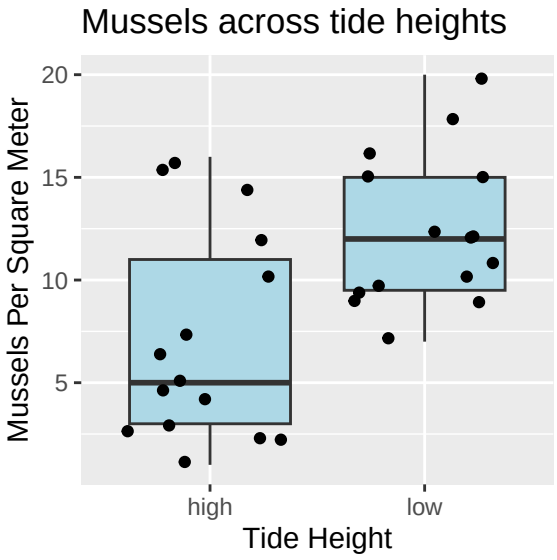
According to Marm, "boxplots are one of the dumbest things ever invented". **Any idea why he thinks that?**

(For **bold** questions, we'd like you to put the answer in your R source code that you submit).

Try looking at the help files for base R (`help(boxplot)`) and ggplot (`help(geom_boxplot)`). What do the "whiskers" represent in each? What about the boxes? And the middle line? This can be confusing and inconsistent. But perhaps the most important question you should ask is: where are the raw data?????

In ggplot, it is pretty easy to layer the raw data on top of the boxplot using the `geom_jitter()` function. YOU SHOULD LITERALLY ALWAYS DO THIS. Here's how:

```
ggplot(data=d, aes(x=tideheight, y=mussels)) +
  geom_boxplot(fill="light blue")+
  geom_jitter()+
  labs(title="Mussels across tide heights",
       x="Tide Height",
       y = "Mussels Per Square Meter")
```



(hot tip: ggplot layers things from top to bottom. so if you want something to be layered on top, put it lower down in your code... here, the geom_jitter points are added on top of the geom_boxplot boxes).

7) Data aggregation

In R, the syntax for plots and models is `(y~x,data=d)`. (THIS IS REALLY IMPORTANT AND YOU'LL NEED TO KNOW IT A MILLION TIMES IN THIS CLASS SO CONSIDER WRITING IT DOWN SOMEWHERE YOU CAN SEE IT ALL THE TIME!) The tilde (~) basically says "as a function of".

R is a very powerful tool for aggregating data(grouping it and then calculating something). The `aggregate()` function is great for this. The syntax is `y~x,data=d` with one more argument FUN that specifies which function (e.g., mean, sd, max, min, length, etc) you want to use. Let's say you want to calculate the *median* number of `mussels` in each `tideheight` category. You would type:

(for those of you who use tidyverse for data manipulation, don't worry - we'll go over that in another lab later... let's just keep it simple for now so we can get everyone up to a foundational level!)

```
aggregate(mussels~tideheight, data=d, FUN=median)
```

```
##   tideheight mussels
## 1       high       5
## 2        low      12
```

Compare that with your boxplot above; do the values seem to match? Try calculating the max, min, or length by replacing `median` above. If you want, you can also add in an additional explanatory variable using the notation (y~x1+x2,data=d,FUN=median). For example if you wanted to calculate the median number of mussels for each `tideheight` and `site` combination, you would use:

```
aggregate(mussels~tideheight+site, data=d, FUN=median)
```

```
##   tideheight site mussels
## 1       high    A       5
## 2        low    A      10
## 3       high    B       3
## 4        low    B      12
## 5       high    C      14
## 6        low    C      15
```

Remember this code prints the result from `aggregate()` but doesn't actually store it. If you wanted to refer to these results later (e.g., if you wanted to make them into a figure), you would assign the results to a variable (e.g., `agg`) using: `agg=aggregate(mussels~tideheight+site, data=d, FUN=mean)`.

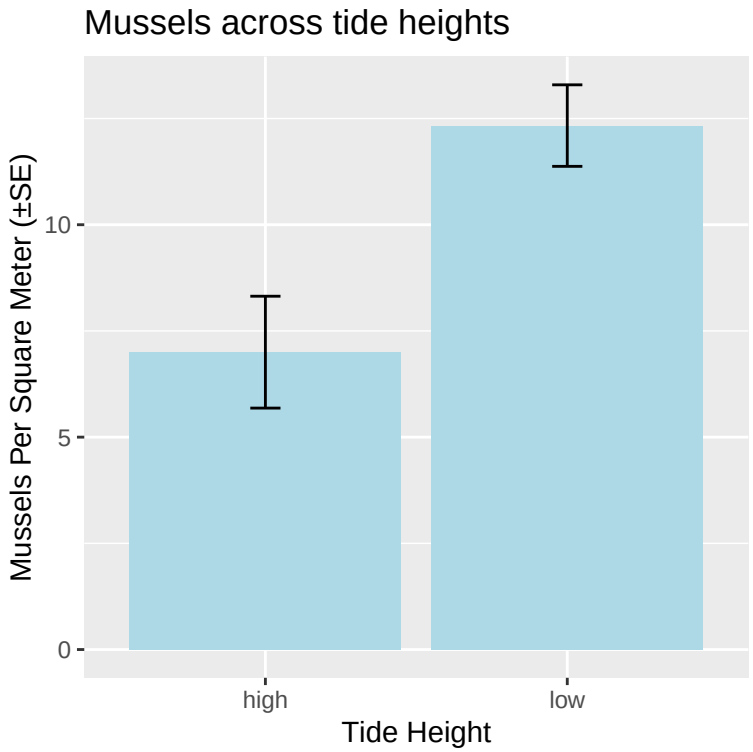
8) Fancy plotting

Just for fun (and your future reference / copy-pasting abilities), here is some code to make a relatively easy bargraph that shows Standard Error bars. Note this requires us to use tidyverse to calculate mean, standard deviation, and standard error of mussel #s for each tide height. Again, don't stress about this code - we'll have a whole lab on tidyverse calculations.

```
agg = d %>% #take the d dataset
  group_by(tideheight) %>% #for each tideheight category
  summarise(mean = mean(mussels), #calculate the mean # of mussels
            sd = sd(mussels), #and the sd of mussels
            se=sd/sqrt(n())) #and the se of mussels
```

Now that we have our aggregated dataset `agg`, we can make a plot from those data.

```
ggplot(data=agg, aes(x=tideheight, y=mean)) +
  geom_bar(stat="identity", fill="light blue") +
  geom_errorbar(aes(ymin=mean-se, ymax=mean+se),
                width=.1)+
  labs(title="Mussels across tide heights",
       x="Tide Height",
       y = "Mussels Per Square Meter (±SE)")
```



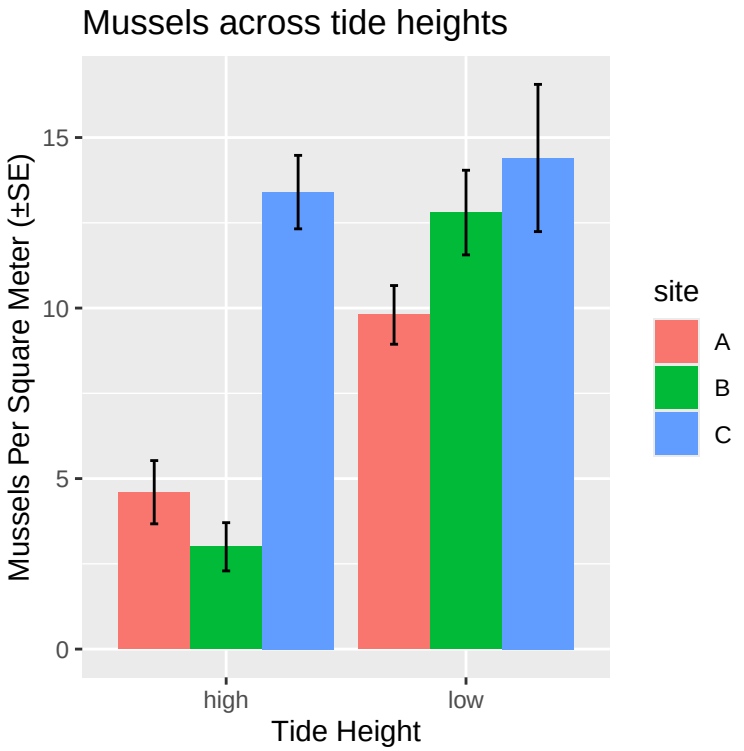
See how I've indicated on the y-axis label that we are showing ±SE? Please get in the habit of doing this in all your plots!

We can also group by site if we want... this is the same code that we used above to create `agg`, but this time we're adding `site` as a grouping variable and calling the resulting table `aggbysite`.

```
aggbysite = d %>% #take the d dataset
  group_by(tideheight,site) %>% #for each tideheight category AND site
  summarise(mean = mean(mussels), #calculate the mean # of mussels
            sd = sd(mussels), #and the sd of mussels
            se=sd/sqrt(n())) #and the se of mussels
```

Now we can revise the barplot code we wrote above. Now, we've added `fill=site` to tell ggplot that we want different colors for each site (by default, a legend is added) and some `position_dodge()` information to tell the bars and error bars to match eachother.

```
ggplot(data=aggbysite, aes(x=tideheight, y=mean,fill=site)) +
  geom_bar(stat="identity",
           position=position_dodge(width=.9)) +
  geom_errorbar(aes(ymin=mean-se, ymax=mean+se),
                width=.1,
                position=position_dodge(width=.9))+
  labs(title="Mussels across tide heights",
       x="Tide Height",
       y = "Mussels Per Square Meter (±SE)")
```



This plot is still super ugly... later in the class we'll get deep into changing theme and color options. Stay tuned!

Now, let's practice...

9) EXAMPLE: sea star

Open the data file `Size and age of seastars.csv`.

```
dat=read.csv("Size and age of seastars.csv")
```

(if the line above doesn't work, try checking your working directory to make sure the file is there.)

The important variables in this worksheet are age in years (`Age..years.`) and diameter in milimeters (`Diameter..m.`). Note that the reason these column names contain periods is that in R, spaces aren't

allowed in the names of the columns or in the variable names. (But they are allowed in the code, such that `A=2` is the same thing as `A = 2`).If you import data with a space in any of the column headers, R will replace the spaces with periods. *Best practice: look at your data before you save it in Excel to upload into R. Change header names to something easy (think lowercase, no spaces or characters, easy to spell and remember what the columns contain).*

Now that it is already loaded into R, we can change the names of the columns using:

```
colnames(dat)=c("Age","Diameter")
head(dat) #to check that it worked properly...
```

```
##   Age Diameter
## 1   1       35
## 2   2       65
## 3   3       92
## 4   4      116
## 5   5      138
## 6   6      158
```

By the way, these data are from ochre stars, also known as *Pisaster ochraceus*:

Figure 1: a happy, healthy ochre star

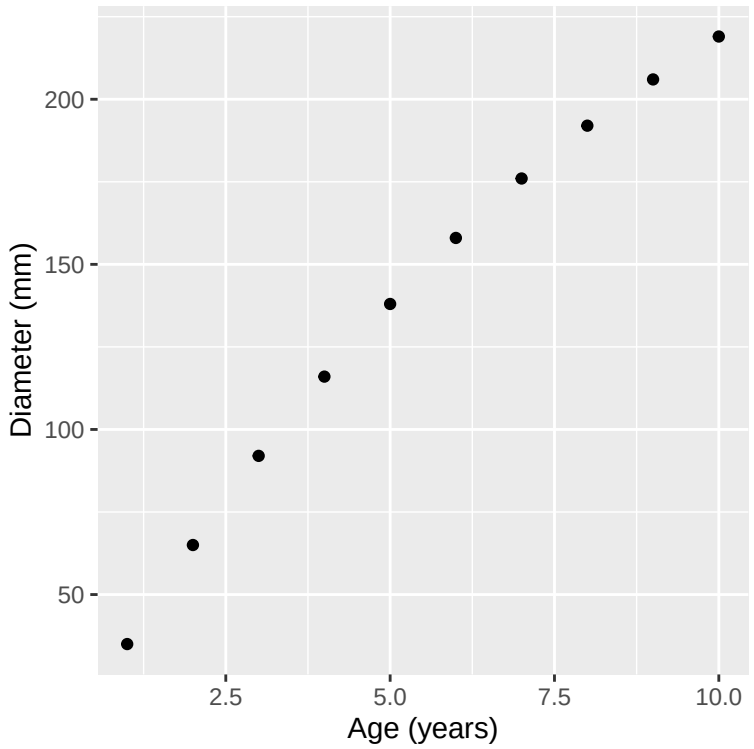
Let's build a plot! Assume you want to look at the relationship between size and age of these seastars. Ask yourself three questions:

- What type of data are these?
- What sort of graph do you want to make?
- Which variable goes on which axis?

Here, the data are both continuous numeric so you'd probably want to create a scatterplot with diameter on the y-axis and age on the x-axis (you are probably asking whether diameter depends on age, not the other way around).

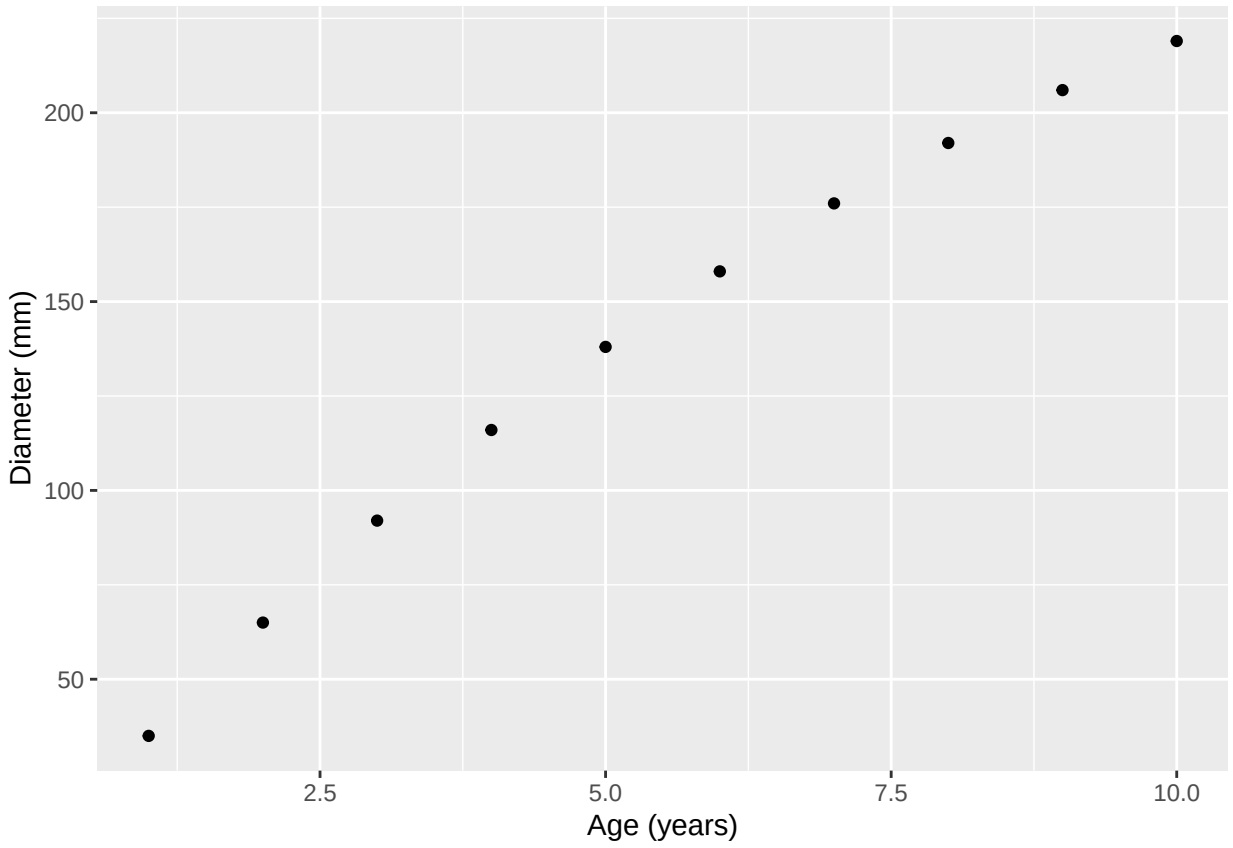
You should use ggplot, which I've shown above, to plot these data. Consider making some modifications to color, size, labels, etc.

```
ggplot(data=dat, aes(x=Age,y=Diameter))+
  geom_point()+
  labs(x="Age (years)",y="Diameter (mm)")
```



Aside from changing the x- and y-axis labels to something meaningful (e.g., with units), saving high resolution figures is one of the easiest ways to make your R plots look great. There is a built-in function called `ggsave()` that saves plots with the settings and filenames you specify. The way you use the function is through two lines of code: 1) the code to make the plot, and 2) the `ggsave()` function with all the specifications (use `help(ggsave)`) Here's an example:

```
ggplot(data=dat, aes(x=Age,y=Diameter))+
  geom_point()+
  labs(x="Age (years)",y="Diameter (mm)")
```




```
ggsave(filename='Diameter~Age.png',width=4,height=4,units="in",dpi=300)
```

This chunk of code basically says "create a png file of the last plot (the Diameter~Age one) that is 4 inches wide and 4 inches high with resolution 300 inches. Remember that the plot will be saved in your working directory - go ahead and open up that folder and see if there is a beautiful new plot! (Pro tip, you can create a .pdf instead of a .png using the `pdf()` function in the same way (just specify .pdf at the end of the file name instead of .png).

10) EXAMPLE: abalone landings

Let's start fresh with `rm(list=ls())` and open a new dataset called `abalonelandings.csv`. These are measured in lbs.

```
dat=read.csv("abalonelandings.csv")
```

Go ahead and use the tips and tricks we provided above to check out this dataset.

> **What is the relationship you would like to depict? What are the x and y axes? What sort of graph type is most appropriate?**

Create the graph of your choice and interpret the results!

> **Are the axis settings right? If not, ask us, and we can show you how to change them.**

Wanna go above and beyond? Try using `scale_color_manual()` (Google how to use it!) to change the point colors to match the colors of the species.

Export the figure as a high resolution .png file.

11) EXAMPLE: VO_2 max

Now try to create the appropriate graph for the relationship between time to run a mile and oxygen consumption, using the dataset `VO2 max vs runtime.csv`.

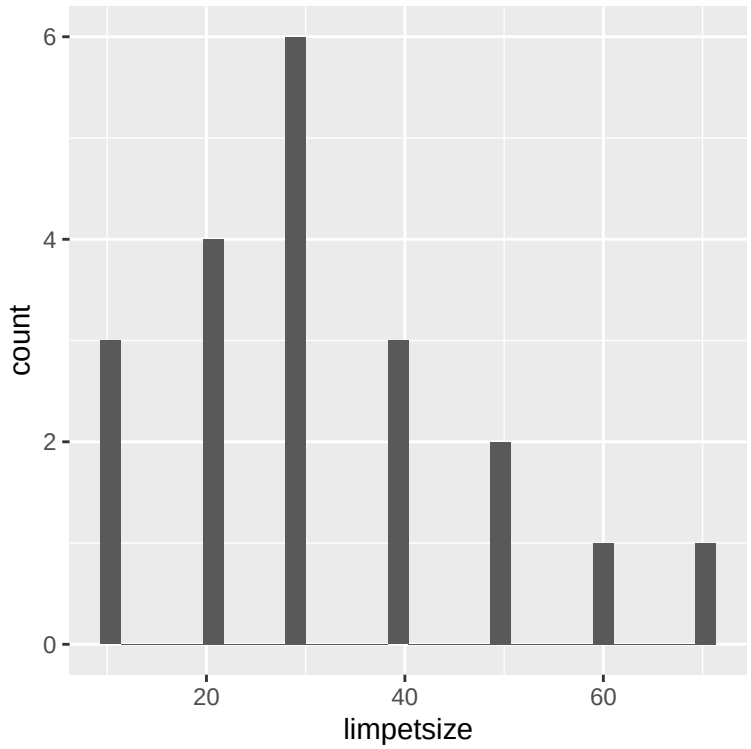
What type of graph would you use?

12) EXAMPLE: Limpet size

Now you are going to plot the distribution of limpet sizes (a size frequency distribution) using the dataset `Limpet size.csv`. This type of distribution is the most informative way to represent datasets that summarize counts of something (like the size or age frequencies of populations).

You can start with a simple histogram in ggplot (note here, for histograms, you only have to specify what goes on the x axis (limpetsize), not what goes on the y-axis, because ggplot does that for you):

```
dat=read.csv("Limpet size.csv")
ggplot(data=dat,aes(x=limpetsize))+
  geom_histogram()
```

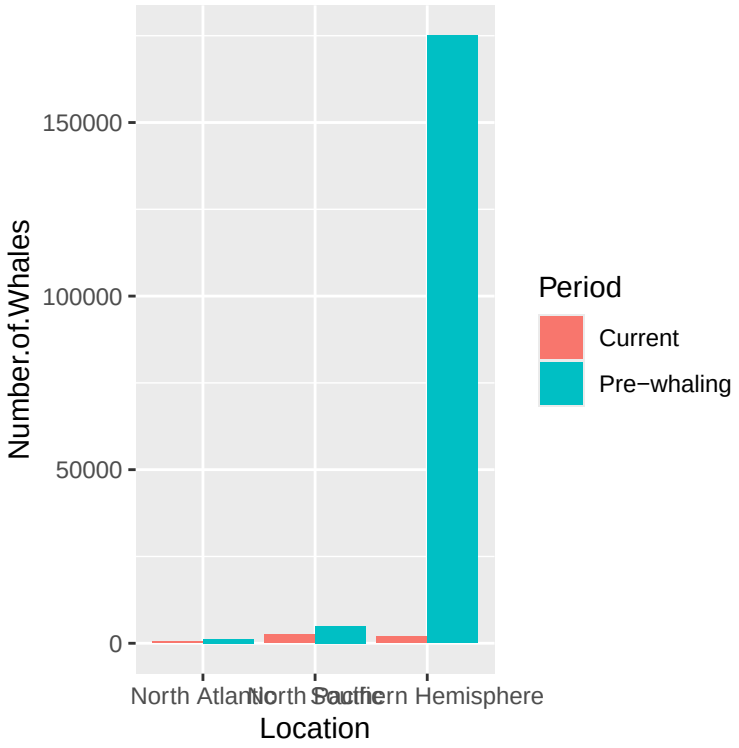


Beautiful. You might want to change the x-axis labels and main title...

13) EXAMPLE: Whales

Now, open the dataset `Blue whale abundance.csv`. Here you are looking at the relationship between number of whales, location of harvest, and time period. Let's say you want to compare the number of whales harvested across both location and time period.

```
dat=read.csv("Blue whale abundance.csv")
ggplot(dat,aes(x=Location,y=Number.of.Whales,fill=Period))+
  geom_bar(stat="identity",position=position_dodge())
```



If you don't include the `position=position_dodge()` part, the barplot will be stacked instead of side by side.

See how the x-axis labels are overlapping? You can rotate each label 90 degrees using the following code: `+theme(axis.text.x=element_text(angle=90))`. I'll add it to the plot down below so you can try it out.

This looks great, but we probably want to re-order the time periods to be pre-whaling and then current (i.e., swap the order of the legends). Let's check the structure of the `Period` data and then change it to a factor with levels in the order we want:

```
str(dat$Period)
```

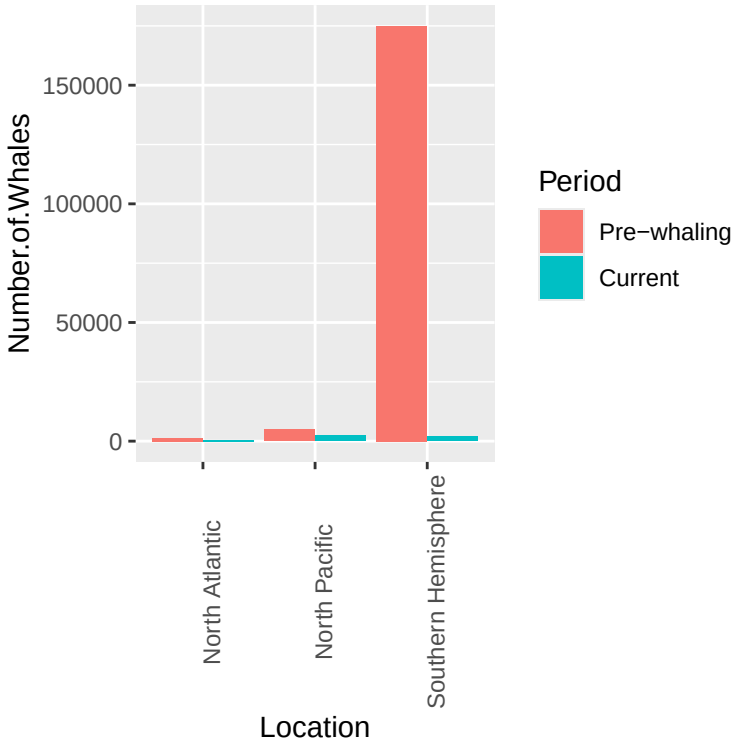
```
##  chr [1:6] "Pre-whaling" "Current" "Pre-whaling" "Current" "Pre-whaling" ...
```

Looks like it is currently a character - you can tell because the output in the R console says `chr`.

```
dat$Period=factor(dat$Period,levels=c("Pre-whaling","Current"))
```

That line of code says "order the factor by Pre-whaling then Current". Now, make the plot again:

```
ggplot(dat,aes(x=Location,y=Number.of.Whales,fill=Period))+
  geom_bar(stat="identity",position=position_dodge())+
  theme(axis.text.x=element_text(angle=90))
```



Make sure to save the graph. Now, onto the final example!

14) EXAMPLE: Urchin/kelp map

Let's make a map overlaid with data using the dataset `Urchin kelp Lat Long.csv`.

```
dat=read.csv("Urchin kelp Lat Long.csv")
```

Maps are a little bit complicated to plot in R but they are gorgeous. Make sure you have the packages `ggmap` and `mapdata` installed.

```
#install.packages("ggmap")
library(ggmap)
library(mapdata)

ylim=range(dat$LAT) #find latitude limits of data to inform map
xlim=range(dat$LONG) #find longitude limits of data to inform map
```

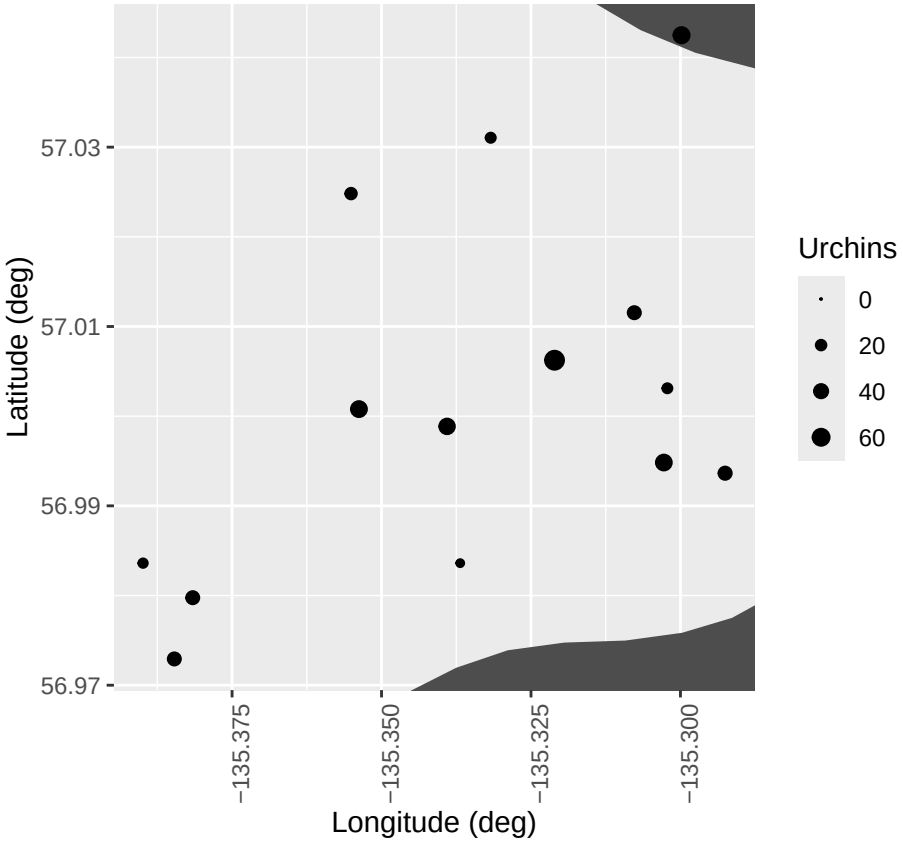
FYI, `range()` is a built in function. Try typing `help(range)` into the R console to see what it does.

Now let's continue making the map plot!

```
#world map data
w=map_data("worldHires",ylim=ylim,xlim=xlim) #extract map data

#make the plot
z=ggplot()+ #make an empty plot
  labs(y= "Latitude (deg)", x = "Longitude (deg)")+
geom_polygon(data=w,aes(x=long,y=lat,group=group),fill="grey30")+
coord_fixed(1.5,xlim=xlim,ylim=ylim)+
geom_point(dat,mapping=aes(x=LONG,y=LAT,size=Urchins))+
scale_size(range = c(0, 3))+
theme(axis.text.x=element_text(angle=90))

#display the plot
z
```



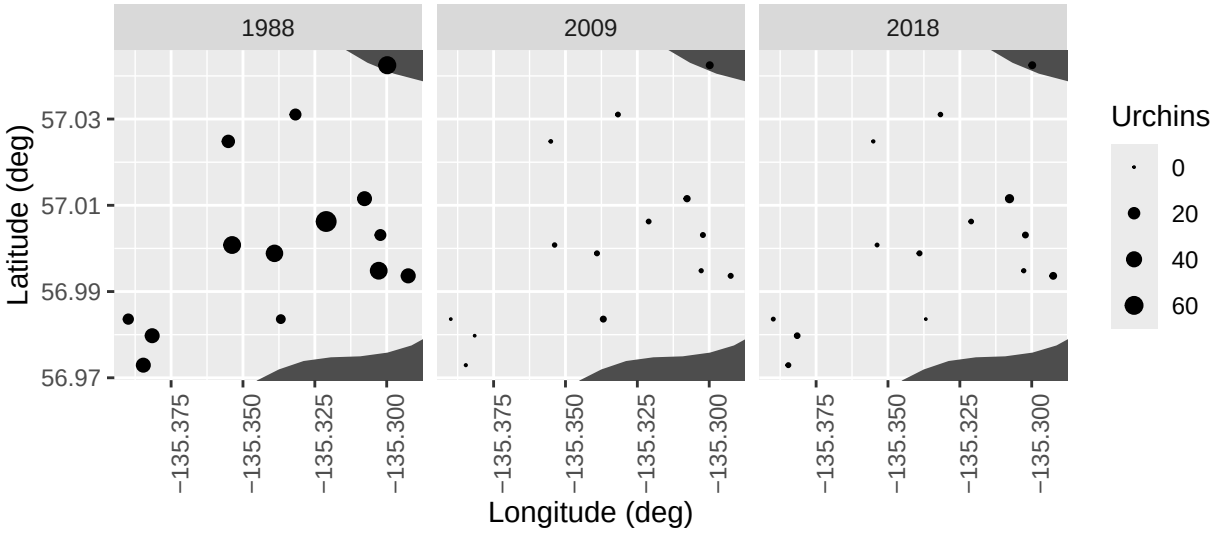
These data were collected in Sitka Alaska in a relatively small area so the dark grey polygons you see are small slivers of the coast. In this plot, the size of the points indicates the abundance of urchins at the site.

To save ggplots, you use `ggsave()` with just the filename in parentheses. The function basically takes the last plot you viewed and saves it. So after running the code above you would type something like `ggsave("SitkaUrchinsAllYears.png")`. Go ahead and try it!

If you looked at the data (using `head(dat)` or `colnames(dat)`) you would see that there is a column called `Year` that contains three unique values `unique(dat$Year)`: 1988 (before sea otters), 2009 (after sea otters colonized), and 2018 (after hunting for otters resumed near Sitka).

We can plot the data by year in three panels using the `facet_grid` function:

```
z+facet_grid(.~Year)
```



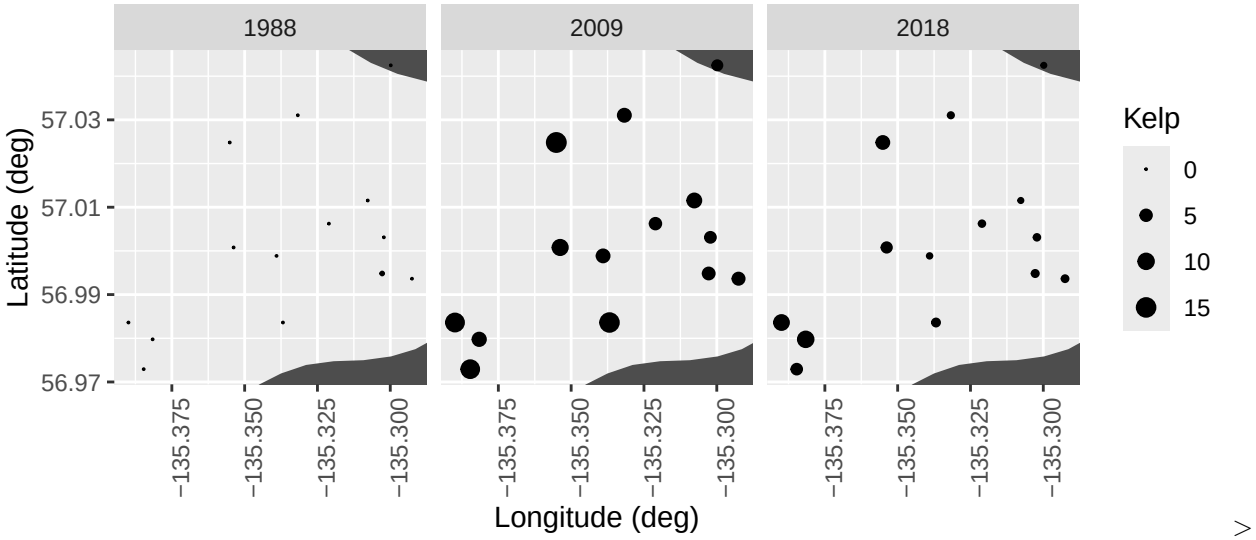
Yay, you made your first faceted plot! These are INCREDIBLY useful for plotting lots of stuff without repeating axes and/or for making separate plots for groups (like year).

Want to know how the faceting was specified in the plot above? Check out the help page by typing `help(facet_grid)` into the Console. You'll see that I specified that each variable should be a column by including `.~Variable`. The opposite (each variable as a row) would be specified using `Variable~.`. Try it out for yourself. Weird syntax, but it works.

What pattern do you see?

Let's make the same plot for kelp:

```
z=ggplot()+
 labs(y= "Latitude (deg)", x = "Longitude (deg)")+
 geom_polygon(data=w,aes(x=long,y=lat,group=group),fill="grey30")+
 coord_fixed(1.5,xlim=xlim,ylim=ylim)+
 geom_point(dat,mapping=aes(x=LONG,y=LAT,size=Kelp))+
 scale_size(range = c(0, 3))+
 theme(axis.text.x=element_text(angle=90))
z+facet_grid(.~ Year)
```



What is the pattern? > How do the graphs help you understand the patterns?

Congratulations! You've made it through your first BIOE186/286 lab. Hopefully your brain isn't too tired. You've done a lot of really important work today. Please let us know if you have any questions :)