Module 2: Exploratory Data Analysis and Plotting

The purpose of this handout is to introduce you to working with and manipulating data in R, exploring your data and to creating figures from the ground up.

1 The Resource-by-Predation dataset

In this workshop, we will use data from an experiment I conducted in Gamboa, Panama in 2010 and which was published in the journal *Ecology* in 2013 (`https://www.jstor.org/stable/23436298`). The experiment was part of an NSF funded project to Drs. Karen Warkentin (Boston University) and James Vonesh (Virginia Commonwealth University) studying the effects of flexible hatching timing by red-eyed treefrog (*Agalychnis callidryas*) embryos on interactions with predators and food levels and subsequent phenotype development of tadpoles. You should have been emailed a .csv file titled 'RxP.csv.' The data are called by the short name 'RxP', which stands for 'Resource-by-Predation', which was the nature of the experiment (we were studying the interaction of resources and predators). This brings up a small but important point. Since R is entirely based on typing commands by hand, you should give your datasets and variable short names so that they are quick and easy to type.

First, let's get a handle on what the data are.

- The experiment consisted of 96 400 L mesocosm tanks arrayed in an open field in the northwest corner of Gamboa. The mesocosms were spatially arranged in 8 blocks of 12 tanks each. Each block consisted of 1 tank from each of 12 unique treatment combinations. Each tank began with 50 tadpoles and the experiment ended when all tadpoles reached metamorphosis or had died. Experimental treatments were as follows:
  - Hatching age (Early [4 days post-oviposition] or Late [6 days post-oviposition])
  - Predators (Control, Nonlethal [caged] dragonfly larvae or Lethal [free-swimming] dragonfly larvae)
  - Resources (Low [0.75 g] or High [1.5 g] food level, added every 5 days)
- We wanted to know how when a frog embryo hatches might affect its development to metamorphosis under various combinations of predators and resources. At metamorphosis, Touchon et al. measured the following response variables:
  - Age at hatching, both in terms of time since eggs were oviposited and time since emergence began (definied as Day 1)
  - Snout-vent length at emergence
  - Tail length at emergence
  - Snout-vent length at completion of tail resorption
  - Mass at completion of tail resorption
  - Number of days needed for each metamorph to fully resorb the tail
- Metamorphosis is a process that takes time. This process is generally defined as the time from when the froglets arms erupt from the body (they develop under the skin) until when the tail is

fully resorbed into the body. The froglet may choose to leave the water early or late during that process. Thus, several measurements were taken when the froglets first left the water and several more when the tail was fully resorbed.

- During the course of the experiment disease broke out in 18 of the mesocosms containing Nonlethal predators and thus those tanks have been removed from the dataset.
- Full citation for the article: Touchon, J.C., McCoy, M.W., Vonesh, J.R., and Warkentin, K.W. 2013. Effects of hatching plasticity carry over through metamorphosis in red-eyed treefrogs. *Ecology*. 94(4): 850-860.

2 Reading in a Data File

There are several ways/options for reading in a data file, depending on what type of computer you are using and if you are using `RStudio` or standard `R`.

2.1 Reading the file from using `read.csv`

If you are loading a data file from somewhere on your computer, you will read it into the active workspace with the command `read.csv` or `read.table` (I recommend `read.csv`). If the data are in your working directory, you would simply do the following (and remember, you should be working in a script window!):

```
> RxP<-read.csv("RxP.csv")
```

2.1.1 Mac vs PC

If your data are not in your working directory, you will need to specify *exactly* where to find the file on your computer. On a Mac this is easy, as you can drag the icon of the file from a Finder window into your script window and your computer will paste in the exact address of the file for you. On a PC, it is not as simple. You can copy the address from the file and paste it into your script window, but know that for whatever reason it pastes in the address with the backslashes (/) in the wrong direction, as forward slashes (\) and you will have to change that manually.

2.2 Using the web version of `RStudio`

If you are using the Vassar web version of `RStudio`, it is very easy to read in your file, but it is a 2-step process. First, in the Files window (lower right corner) click on Upload and use the window that opens to select the .csv file on your computer. This uploads the file to your working directly. Next, in your script window or at the command line use `read.csv` as described above.

2.3 Using the desktop version of `RStudio`

Under the File menu, there is an option to select titled 'Import Dataset' and you can choose to import the data from your .csv file. However, this is a bit annoying because instead of reading in the file as a *data frame*, `RStudio` reads it in as a format called a *tbl_df*, which is different. This is a newer format preferred by one of the lead designers of `RStudio`, but the annoying thing is that the format does not work well with most of `R` built in functions. Evidently, it is faster at reading in very huge files.

3 Data Exploration and Error Checking

The first step whenever you start working with a dataset is to check it for errors. Questions you should ask yourself include:

- Did the data import correctly?
- Are the column names correct?
- Are the types of data appropriate? (e.g., factor vs numerical)
- Are the numbers of columns and rows appropriate?
- Are there typos?

If, for example, a column that is supossed to be numerical shows up as a factor, that likely indicates a typo where you accidentally have text in place of a number (remember, each column in a data frame is a vector, and vectors can only have one mode, so a vector with both numbers and characters is treated as if it is all characters). Similarly, if you have a factor that should have 3 categories, but imports with 4, you likely have a typo (e.g., "predator" vs "predtaor"), and the misspelled version is showing up as a separate category. These sorts of mistakes are very common.

Because this dataset has been thoroughly examined (very thoroughly!), these types of errors are not present. However, you might want to change the names of columns or remove outliers, which we will cover in the subsequent sections.

3.1 Data structure

Begin by examining the structure of the data frame.

```
> str(RxP)
'data.frame':  2502 obs. of  14 variables:
 $ Ind              : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Block            : int  5 5 5 5 5 5 5 2 2 1 ...
 $ Tank             : int  7 4 4 7 10 4 4 5 4 1 ...
 $ Tank.Unique      : int  55 52 52 55 58 52 52 17 16 1 ...
 $ Hatch            : Factor w/ 2 levels "E","L": 1 2 2 1 2 2 2 1 2 2 ...
 $ Pred             : Factor w/ 3 levels "C","L","NL": 3 1 1 3 2 1 1 2 1 3 ...
 $ Res              : Factor w/ 2 levels "Hi","Lo": 1 1 1 1 1 1 1 1 1 1 ...
 $ Age.DPO          : int  35 35 35 35 36 36 36 39 39 39 ...
 $ Age.FromEmergence: int  1 1 1 1 2 2 2 5 5 5 ...
 $ SVL.initial      : num  18 17.7 18.1 16.8 18.7 17.5 17.3 19.6 16.5 17.5 ...
 $ Tail.initial     : num  5.4 1.1 5 6.4 6.3 4.4 1.3 1.5 2 5.1 ...
 $ SVL.final        : num  17 18 17.8 17.1 19.3 17.8 17.9 19.6 17.7 19.5 ...
 $ Mass.final       : num  0.38 0.35 0.41 0.3 0.46 0.3 0.42 0.5 0.33 0.46 ...
 $ Resorb.days      : int  3 3 3 3 3 4 2 2 2 3 ...
```

We can see that our dataset has 2502 observations of 14 different variables, some of which are integers, some are factors and some are numerical. Things to notice:

1. Several variables are listed twice but are coded in different ways. For example, there is a column titled `Tank` and one titled `Tank.Unique`. As stated earlier, there are 12 tanks in each of 12 blocks. The varible `Tank` lists what number a tank is (1-12) in a given block, whereas `Tank.Unique` gives each tank a unique number out of the entire 96.
2. Similarly, we have the columns `Age.DPO` and `Age.FromEmergence`. The first one, `Age.DPO` is the age at emergence from the water in terms of days post-oviposition (DPO), whereas `Age.FromEmergence` counts the day the first animal crawled out of the water as Day 1, and so the age of animals is recorded in terms of days after emergence began. Sometimes it can be useful to view the same data in two different manners.
3. We have three categorical predictors (factors): Hatching age, Predator treatment and Resource level.
4. We have several response variables (e.g., SVL or Mass) that were measured at the initial point when froglets left the water, or at the finale of metamorphosis, when the tail was fully resorbed, or both.

3.2 Data exploration

Begin by plotting the data to check for errors. The default `plot` function will create a simple graphic based on the type of data you provide it. In order to access a named variable within a data frame, we use the "$" operator, as in `data frame$variable`. The data frame always goes first, then the $, then the name of the variable you are interested in. For example, you might type in the following to start looking at your data.

```
> plot(RxP$SVL.initial)
> plot(RxP$Tail.initial)
> plot(RxP$Mass.final)
> plot(RxP$Pred)
```

What did that tell us?

1. SVL generally varies between 14 and 24 mm, but there is one animal that is much smaller.
2. Tail length at emergence varied from about 0 to 15 mm in most individuals, but there were three froglets with tails longer than 15 mm.
3. Mass varied from about 0.2 g to over 1 g.
4. The number of individuals surviving to metamorphosis in the three Predator treatments varied considerably, from over 1200 froglets in the Control group to approximately 500 in the Nonlethal group.

Notice that the style of plot changed depending on if we plotted a continuous variable or a factor. For the continuous variable, the default is to plot the data in order, from the first row to the last. In the case of a factor, the default is plot the number of observations in each group.

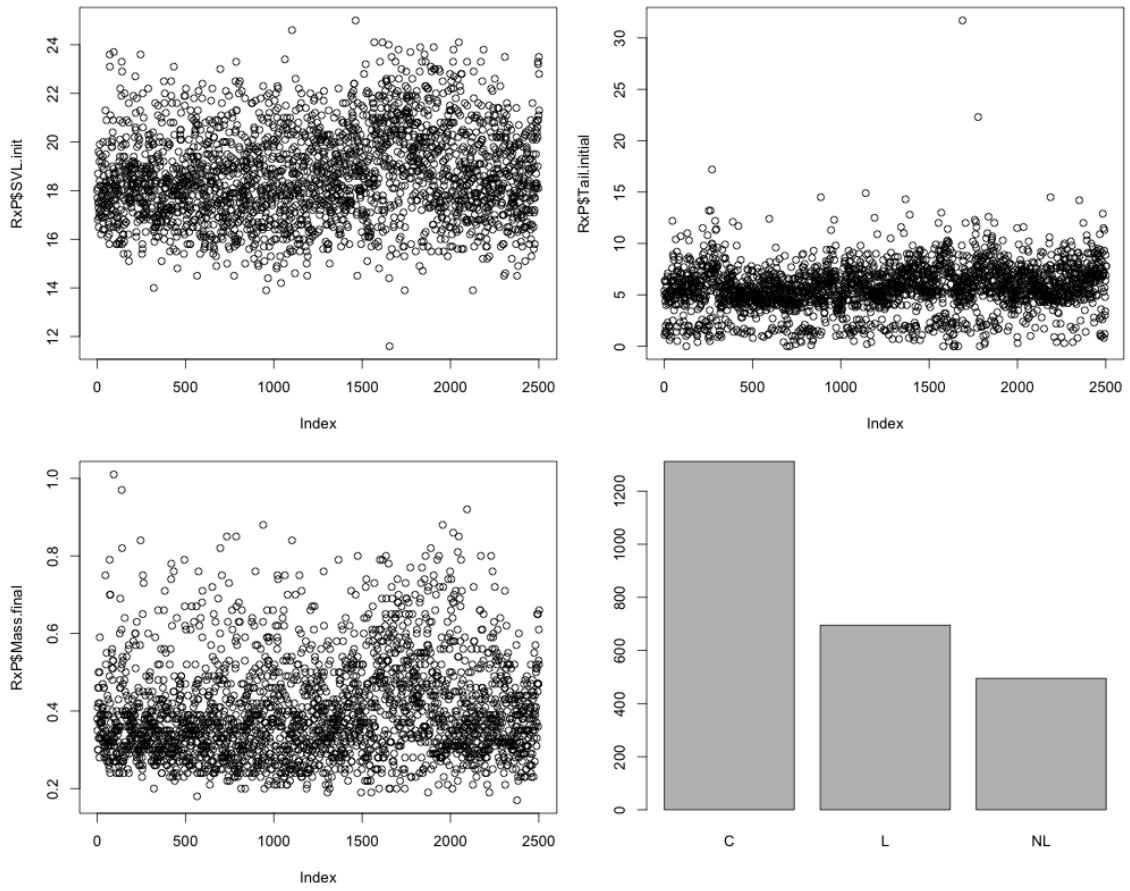

Figure 1: Plots of various response variables

3.3 Further data exploration and Indentifying mistakes

Plotting data by itself can be useful, for example if we want to check for outliers or find typos (which might make a numeric variable plot as a factor, for example). However, it is often more useful to plot response data against an explanatory variable. For example, maybe we want to know how final mass of metamorphs varies across predator treatment. Here, we will use the ~ to separate our response, or dependent, varible (`Mass.final`) from our predictor, or independent, variable (`Pred`).

```
> plot(Mass.final~Pred, data=RxP)
```

Here are some important things to take note of. First, by providing a categorical variable as our predictor, `R` automatically knew to make a box-and-whisker plot (aka 'boxplot'). There are not that many instances when `R` will think for you, but this is one. Looking at Figure 2, there are several things to know about how `R` draws a boxplot. First, the top and bottom of each box represents the *interquartile range*, i.e., the middle 50% of your data. Thus, 25% of metamorphs in each predator treatment are larger than the top of their respective box, and 25% are smaller than the bottom. Secondly, the heavy

dark line in the middle of the box is the *median*, not the mean as many folks might initially think. The extremes of the 'whiskers' are either 1) the max or min value of the data or 2) 1.5 times the interquartile range. In the event of option #2, R will plot all the points that fall beyond the 1.5 times the IQR. So what does that mean in practice? If you look at Figure 2, you can see that the bottom whiskers are all just that, a whisker. That means they have been plotted to the smallest value in the dataset and that that value falls within 1.5 times the IQR. The upper whiskers have a whole lot of points above them, meaning that the whiskers extend to the 1.5 times the IQR mark, and the points plotted above the whisker fall outside that range.

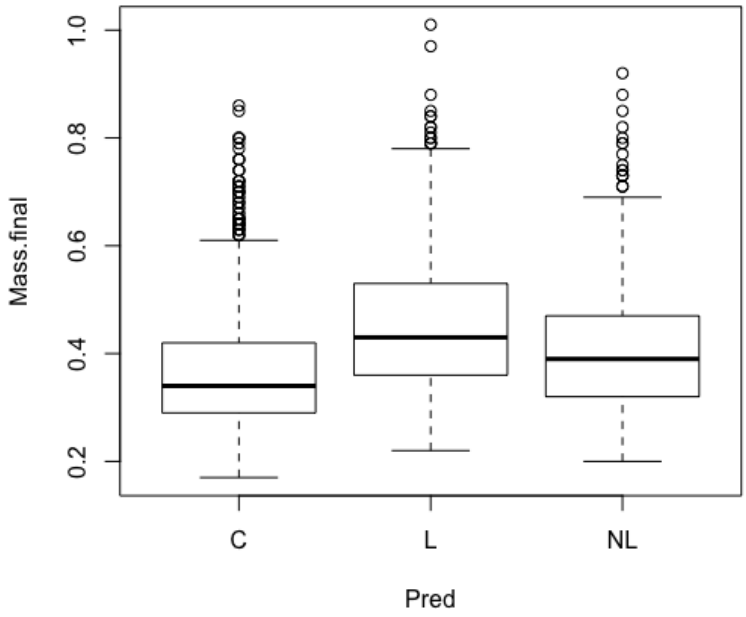

Figure 2: Relationship between Predator treatment and Mass at the end of *A. callidryas* metamorphosis

What happens if we plot two continuous variables against one another, instead of a continuous vs a categorical predictor? Maybe we want to see if there is a relationship between mass at the end of metamorphosis and SVL at the end of metamorphosis. Since we have provided two continuous variables, R will know to automatically make a scatterplot.

```
> plot(Mass.final~SVL.final, data=RxP)
```

This figure tells us several things.

- There appear to be several outliers. These are either individuals that have a very small SVL but a large Mass, or vice-versa. These almost certainly represent mistakes during data entry and should be removed.
- The relationship between SVL and Mass is not linear. Longer frogs seem to have disproportionately larger masses. This is to be expected in many length-to-mass relationships in nature, and perhaps plotting the data on log-log axes would make this relationship linear.

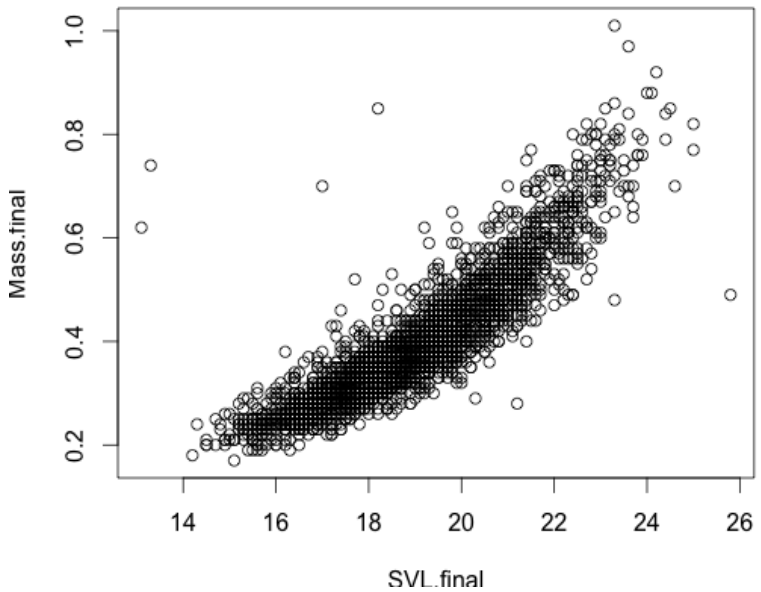

Figure 3: Relationship between SVL and Mass at the end of *A. callidryas* metamorphosis

- In addition, notice that we have introduced a new syntax. We can use the ~ to denote a relationship between two vectors, usually written as `response~predictor`. This structure will be used later for defining statistical models and can be expanded to encorporate multiple predictors. e.g., `response~predictor1 + predictor2 + etc.`.

First, let's see if plotting the figure on log-log axes makes the length-to-mass relationship linear. There are easy two ways to accomplish this.

1. `plot(log(Mass.final)~log(SVL.final), data=RxP)`

   This will take the log of each variable and plot them against one another. Thus, the values on the axes will be in terms of the logarithm of either SVL or Mass.

2. `plot(Mass.final~SVL.final, data=RxP, log="xy")`

   This will plot the figure with normal numeric axes but the scale of the axes will increase at a log rate. Note that by denoting `log="x"` or `log="y"` you can just log transform one axis.

Notice that the two figures are identical except in their axes. In the figure on the left, the axis is linear in its increments but the values are log-transformed. In the figure on the right, the units correspond to the raw values that were measured, but they increase non-linearly. Comparing these figures with Figure 2, it indeed appears that log-transformation made the data more linear.

3.4 Fixing mistakes in the data

How do we go about fixing mistakes that we have found in the data file? One option would be to go back to your original .csv file or use Microsoft Excel (or whatever program you use to organize and input

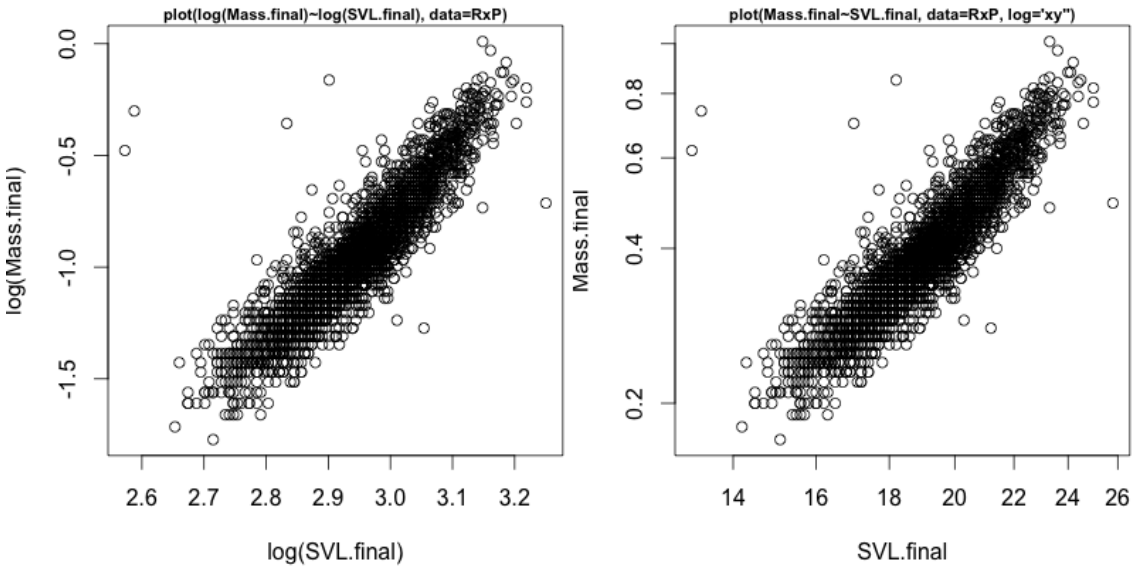

Figure 4: Relationship between log(SVL) and log(Mass) at the end of *A. callidryas* metamorphosis

your data) and change it there. You should certainly do that! But this dataset has over 2500 individuals and trying to figure out which individuals are outliers could be very tedious.

First, we can see that there are four individuals that are recorded as having an SVL below 19 mm, and a Mass above 0.6 g. Thus, one way to identify where in the data file those mistakes are located is to search for any froglets with a Mass above 0.6 and an SVL below 19. We can do this using the []'s, which allow us to subset the data based on some criterion or criteria.

3.4.1 A note about indexing

Using square brackets ([]'s) to navigate a data frame (or other type of object) is one of the single most important things you can learn to do in R. If you think about a data frame as a two-dimensional object with rows and columns, every item (i.e., cell) in the data frame has a location in terms of its row and column. []'s allow you to navigate the data in a simple and elegant manner. You did a tiny bit of this in Module 1, but here it is explained more fully. Square brackets allow you to find the location of any object in terms of `[row, column]`. Thus, if you want the object in the 5th row and the 3rd column in our data frame RxP, you would type the following:

```
> RxP[5,3]
[1] 10
```

You can also []'s to identify a range of things, like the first four rows and the first three columns.

```
> RxP[1:4,1:3]
  Ind Block Tank
1   1     5    7
2   2     5    4
3   3     5    4
```

```
4   4     5    7
```

Most importantly, you can enter in logical statements within the []'s. For example, if you just want to return all of the rows that are from the Low resource treatment, you can type the code below. Note that the column part of the [row, column] notation is left blank, which indicates to return every column in the data frame. Also, note the code to define what you want to return. The code below states "return every row of RxP where the Resource column is equal to Lo."

```
> RxP[RxP$Res=="Lo",]
      Ind Block Tank Tank.Unique Hatch Pred Res Age.DPO Age.FromEmergence SVL.initial
16     16     1   12          12     E    L  Lo      39                 5        19.5
18     18     5    3          51     E    L  Lo      37                 3        18.2
22     22     3    9          33     E    C  Lo      38                 4        16.2
23     23     3    9          33     E    C  Lo      38                 4        17.8
55     55     1    8           8     E    C  Lo      40                 6        17.1
70     70     3    3          27     E   NL  Lo      40                 6        18.4
73     73     3    3          27     E   NL  Lo      40                 6        18.1
74     74     1   10          10     L    C  Lo      41                 7        16.5
78     78     1   10          10     L    C  Lo      41                 7        17.0
91     91     1    4           4     L    L  Lo      41                 7        16.5
102   102     7    2          74     E    L  Lo      38                 4        17.6
113   113     5    3          51     E    L  Lo      39                 5        16.5
115   115     5    3          51     E    L  Lo      39                 5        18.4
119   119     5   12          60     L   NL  Lo      39                 5        17.9
122   122     5    3          51     E    L  Lo      39                 5        19.0
...
```

Think about how easy it is use this to create new subsets of your data, if you wanted to. You can couple indexing with the assign function (the little arrow, <-) to create new objects. For example, if you wanted to create separate objects that contained the Lo and Hi resource individuals, you could type the following:

```
> RxP.Lo<-RxP[RxP$Res=="Lo",]
> RxP.Hi<-RxP[RxP$Res=="Hi",]
```

With all of that in mind, let's use indexing to find our errors and remove them from the data frame. For example, if we want to identify every froglet that has a final SVL of less than 19, we can type the following:

```
> RxP[RxP$SVL.final<19,]
  Ind Block Tank Tank.Unique Hatch Pred Res Age.DPO Age.FromEmergence
1   1     5    7          55     E   NL  Hi      35                 1
2   2     5    4          52     L    C  Hi      35                 1
3   3     5    4          52     L    C  Hi      35                 1
4   4     5    7          55     E   NL  Hi      35                 1
6   6     5    4          52     L    C  Hi      36                 2
7   7     5    4          52     L    C  Hi      36                 2
  SVL.initial Tail.initial SVL.final Mass.final Resorb.days
```

```
1        18.0          5.4      17.0       0.38          3
2        17.7          1.1      18.0       0.35          3
3        18.1          5.0      17.8       0.41          3
4        16.8          6.4      17.1       0.30          3
6        17.5          4.4      17.8       0.30          4
7        17.3          1.3      17.9       0.42          2
...
```

This tells R to give us every row of RxP where RxP$SVL.final is smaller than 19. Remember that the convention within in the square brackets is [rows,columns]. The comma after 19 tells R to return every column. As you can see, this returns a lot of individuals (only the first 6 rows are shown above)! The same logic applies if we want to subset the data based on multiple criteria. We can string criteria together using the & logical command. For example, we want to find every froglet with an SVL smaller than 19 mm and a mass greater than 0.6 g.

```
> RxP[RxP$SVL.final<19 & RxP$Mass.final>0.6,]
      Ind Block Tank Tank.Unique Hatch Pred Res Age.DPO Age.FromEmergence
734   734     2    7          19     E   NL  Hi      51                17
1024 1024     8    7          91     E    L  Lo      48                14
1078 1078     6   11          71     E    C  Lo      54                20
1284 1284     1    2           2     E    C  Hi      62                28
     SVL.initial Tail.initial SVL.final Mass.final Resorb.days
734         18.0          0.6      18.2       0.85           1
1024        22.2          8.3      13.3       0.74           6
1078        16.2          1.5      17.0       0.70           3
1284        22.1          9.1      13.1       0.62           6
```

This command returned just four individuals, which correspond to the four points plotted in the upper left corner of the plots in Figures 2 and 3. Looking at Figures 2 and 3, there appears to be one other obvious point that has a suspiciously low Mass despite the largest SVL.

```
> RxP[RxP$SVL.final>24 & RxP$Mass.final<0.6,]
      Ind Block Tank Tank.Unique Hatch Pred Res Age.DPO Age.FromEmergence
1127 1127     1    3           3     L    C  Hi      58                24
     SVL.initial Tail.initial  SVL.final  Mass.final Resorb.days
1127        20.3          4.7       25.8        0.49           3
```

Now we have identified five individuals that look like they are the result of bad data entry. This is likely to happen in any large experiment, so it is important to clean up the data before analyzing it.

Remember that []'s also let us access specific cells within a data frame, and coordinates of cells are [row,column]. Our problem individuals are 734, 1024, 1078, 1127, 1284, which is indicated by the row numbers shown on the left side of the output. We want to get rid of these rows entirely, which we can do by using the logical command !=, which means "not equal to". Lastly, let's create a new data frame called temp that excludes the data we don't want.

```
> temp<-RxP[RxP$Ind!=734 & RxP$Ind!=1024 & RxP$Ind!=1078 & RxP$Ind!=1127 &
  RxP$Ind!=1284,]
```

The command above essentially subsets the `RxP` data frame and returns every column when the Individual is not equal to 734, 1024, 1078, 1127, or 1284, and then assigns that resulting data frame to the object `temp`. `R` does not provide any warning messages if you write over data. *It is very easy to accidentally write over your data!* That is one reason why we made a new data frame. Lastly, we should confirm that it worked by plotting the data.

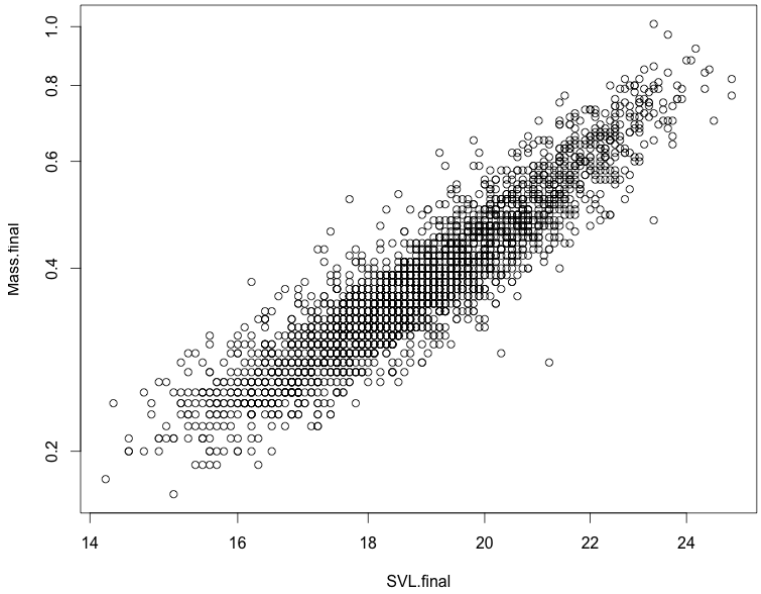

Figure 5: Relationship between log(SVL) and log(Mass) at the end of *A. callidryas* metamorphosis, with instances of obvious data entry mistakes removed

3.5 Outliers?

Identifying outlier data is often difficult and subjective. What one researcher might see as an obvious outlier, another researcher may not. Such is life. One reason to remove outliers, defined here as infrequent and extreme values, is that they can have tremendous influence on your statistical analyses.

A good way to see if outliers have undue influence on your model is to run the model with and without the data in question, and see if it changes the result. If removing the one or two data points does not affect anything, then you can probably feel comfortable leaving them in. But if those data points change the result, then you have to decide what to do. Perhaps, as in the example of the RxP dataset, you have a lot of data (here we have more than 2500 individuals). If one or two individuals change the results for the other 2500 froglets, then they are certainly exerting too much influence and our ability to estimate the ecological relationships of interest will be faulty.

By plotting various response variables against one another, we can further explore the data and search for more errors or potentially find genuine outliers. Some examples of this might include the following (Note that we are now working with our modified data frame `temp`):

```
> plot(SVL.initial~Age.DPO, data=temp)
```

```
> plot(Mass.final~Age.DPO, data=temp)
> plot(Resorb.days~Tail.initial, data=temp)
> plot(SVL.final~SVL.initial, data=temp)
```

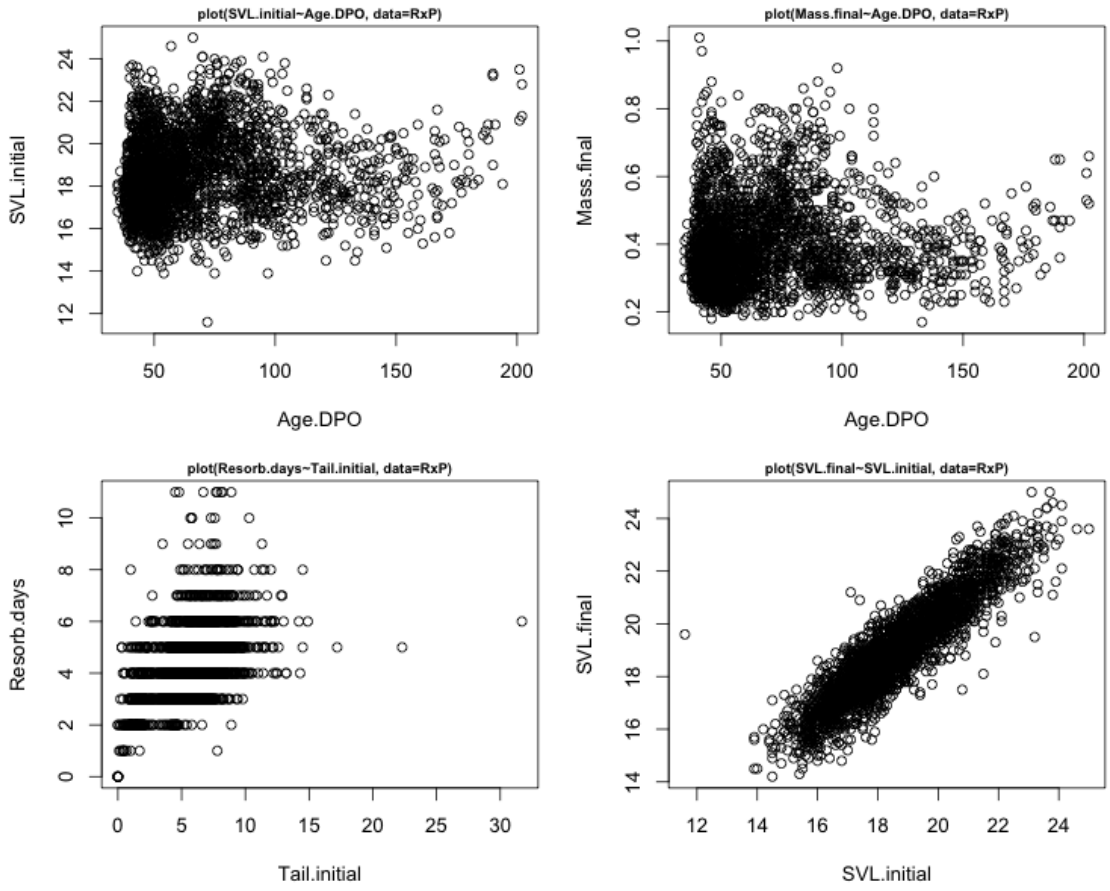

Figure 6: Various response variables plotted against one another in order to look for outlier data

Those figures tell us a lot of things. First, there is one froglet that has a recorded initial SVL that is much smaller (less than 12 mm) than any other froglet. This seems to clearly be an error, especially given that the final SVL of that same individual is about 19 mm. Secondly, although nearly all froglets had tails between 0 and 15 mm, three had tails longer than this. These are likely real data points but they 1) are not representative of the overall population and 2) have a lot of potential to skew our results (which they do).

First, let's identify the froglet with the incorrect very small initial SVL and remove it by assigning `temp` without the problem individual to the same object `temp`. This is essentially like writing over a file with a new version of itself.

```
> temp[temp$SVL.initial<12,]
```

```
      Ind Block Tank Tank.Unique Hatch Pred Res Age.DPO Age.FromEmergence
1655 1655     6    8          68     E    C  Hi      72                38
     SVL.initial Tail.initial SVL.final Mass.final Resorb.days
1655        11.6          5.3      19.6       0.43           5
> temp<-temp[temp$Ind!=1655,]
```

Next we can identify the three individuals with the very long tails and remove them.

```
> temp[temp$Tail.initial>15,]
      Ind Block Tank Tank.Unique Hatch Pred Res Age.DPO Age.FromEmergence
271   271     1    2           2     E    C  Hi      44                10
1689 1689     2   10          22     E    C  Hi      76                42
1777 1777     7    8          80     E    C  Lo      76                42
     SVL.initial Tail.initial SVL.final Mass.final Resorb.days
271         18.8         17.2      18.9       0.36           5
1689        23.3         31.7      23.7       0.64           6
1777        15.8         22.3      15.6       0.23           5
> temp<-temp[temp$Tail.initial<15,]
```

Notice that we did not have to specify the three individuals with the long tails in order to remove them. Instead, since we saw that indeed they all had tails longer than 15 mm, we just redefined `temp` as everything with tails smaller than 15 mm. Now our dataset is ready to analyze! For good measure, let's rename our dataset so that it is clear that we are using the cleaned up data with the outliers removed.

```
> RxP.clean<-temp
```

3.6 Data exploration using `ggplot2`

All of the plotting you have done until now was done in what is known as 'base graphics', i.e., the graphics functions that are built in to `R`. One problem with base graphics is that the figures produced are relatively utilitarian and ugly (at in many people's view). There is a whole universe of functions and arguments you can use to make them look wonderful, but in their basic version they are kind of ugly. A relatively recently designed package called `ggplot2` makes making very nice looking figures very easy. However, the syntax for coding in `ggplot2` is a little different than base graphics. There are two workhorse functions in `ggplot2`: the first is `qplot()` (which stands for 'quick plot') and the other is `ggplot()`. We will cover `ggplot()` at a later date. For now, let's explore `qplot()`.

3.6.1 Boxplots

Earlier you made a boxplot using base graphics. The syntax for this was conveniently the same that we will use to define statistical models (`response predictor`). `ggplot2` does things differently. Instead, you explicitly specify what variable you want on the x or y axes, and you have to specify the type of plot you want using the `geom` argument. The code below creates both plots. Don't forget that in order to use `qplot()` you first need to load the `ggplot2` library. You have to do this each time you restart `R`.

```
> library(ggplot2)
> #Make a boxplot using base graphics
> plot(Mass.final~Pred, data=RxP.clean)
```

```
> #Make a boxplot using base graphics
> qplot(y=Mass.final, x=Pred, data=RxP.clean, geom="boxplot")
```

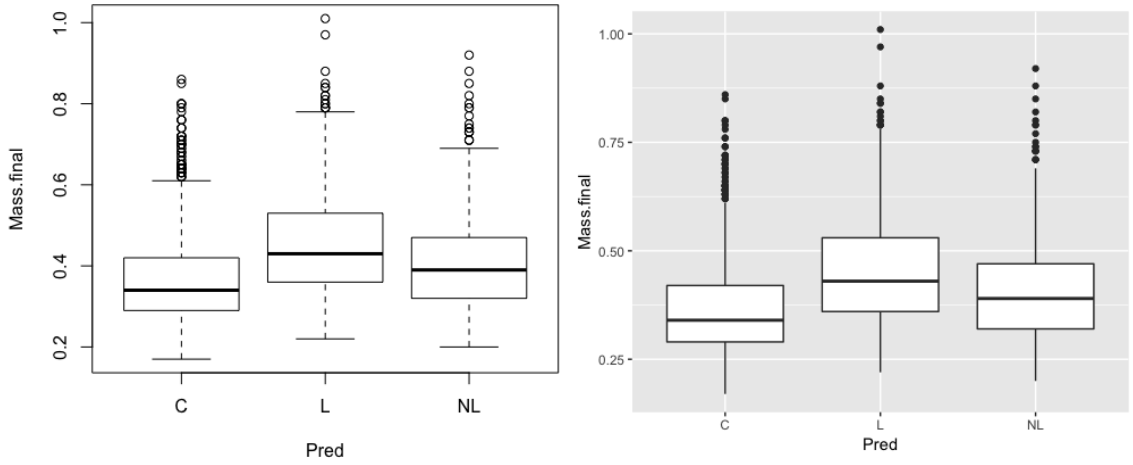

Figure 7: Two boxplots of the same data, the final mass of *A. callidryas* metamorphs plotted against predator treatment. The plot on the left was made with base graphics and the plot on the right with `qplot()`.

The figure made with `ggplot2` is nicer looking (to many people), but this is not why `ggplot2` is so useful. Where `ggplot2` really shines is in its ability to add colors and to plot data across many different variables at once. For example, if you want to add colors to your figures, you can use either the `col` or `fill` arguments. In the case of a boxplot, `col` will change the color of the outline of the boxes whereas `fill` changes the color inside each one. Notice that we define the colors as one of the variables in our dataset. `R` will know how many categories you have, and therefore how many colors to plot, and `qplot()` will even add a legend for you.

```
> #Change the outline color of the boxes and whiskers
> qplot(y=Mass.final, x=Pred, data=RxP.clean, geom="boxplot", col=Pred)
> #Change the color that fills each of the boxes
> qplot(y=Mass.final, x=Pred, data=RxP.clean, geom="boxplot", fill=Pred)
```

3.6.2 Faceting

Okay, so you can add colors really easily. What if we want to plot our mass data across multiple predictors, to more fully explore how the data fall out across different groups? We can easily do this by adding in the `facets` argument. Faceting allows you to easily split your plot window in an intuitive manner. The syntax for `facets` is 'rows columns'. In other words, you specify the variable you want in each row and what you want in each column. If you only want to use one facet, you need to place a '.' on the other side of the ~.

```
> #Plot the mass data by all three categorical predictors by using facets
> qplot(y=Mass.final, x=Pred, data=RxP.clean, geom="boxplot", fill=Pred, facets=Hatch~Res)
```

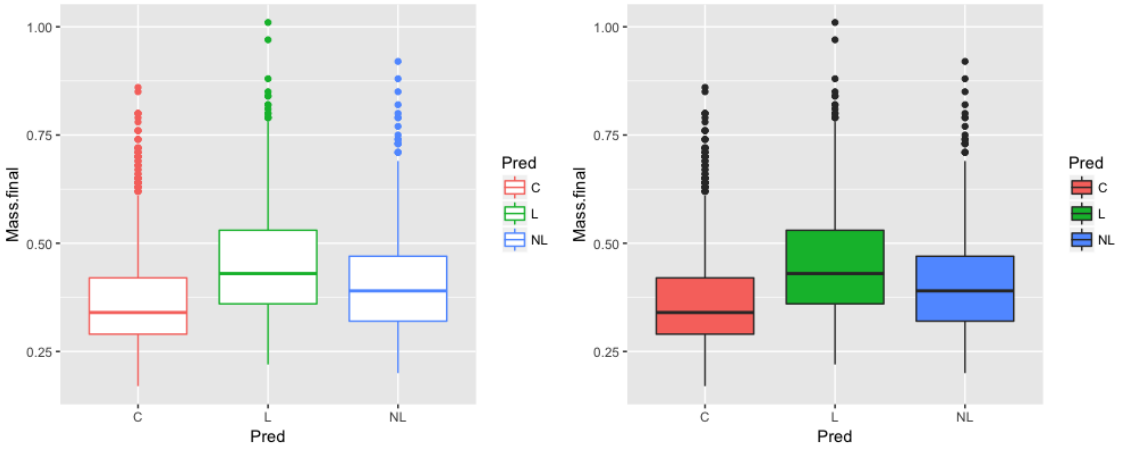

Figure 8: It's really easy to add nice looking colors with `qplot()`!

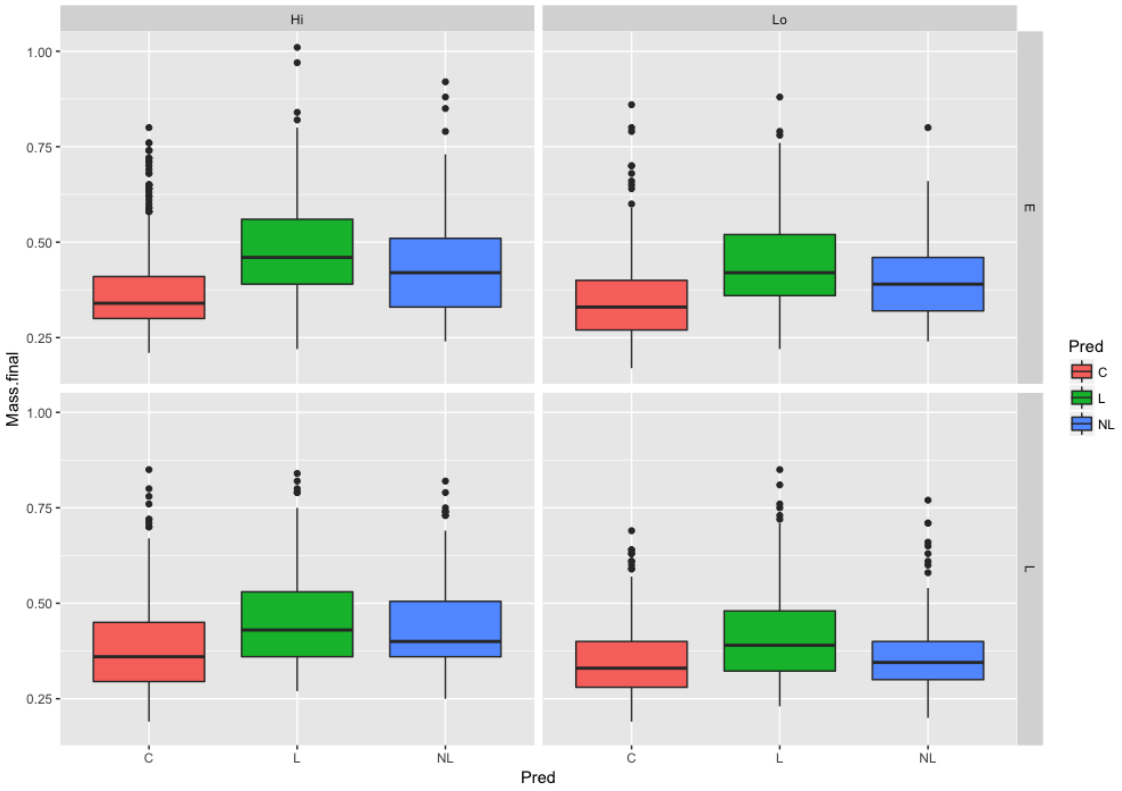

Figure 9: Whoa!! It was so easy to break our data apart in to subplots with `qplot()`!

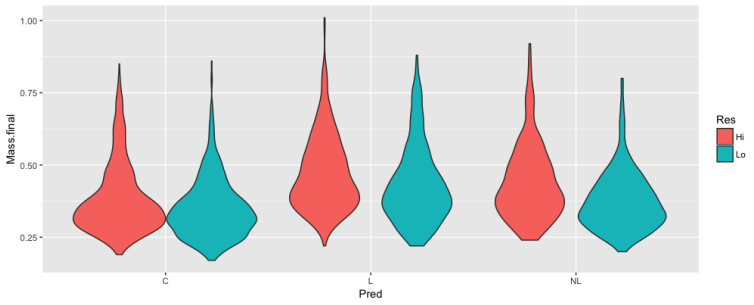

Figure 10: Violin plots of final mass of metamorphs across resource and predator treatments. The violin plots make it easy to see where the preponderance of data are, and we can see things that in the Nonlethal treatment, the Low resource treatment has a slightly downshifted distribution compared to the High resource treatment.

3.6.3 Violin plots

Box plots are great for seeing some basic info about the spread of data you have, but they can be somewhat misleading because they do mask where the preponderance of data might fall. One alternative is known as a violin plot. The coding is the same as for a boxplot, but the `geom` has a different definition (`geom="violin"`). The code below also exemplifies one other great thing about \texttt{ggplot2}. By de argument as a variable (not already defined as our x-axis), we can automatically plot the various categories within the treatment. So convenient!

```
> #Plot the mass data by all three categorical predictors by using facets
> qplot(y=Mass.final, x=Pred, data=RxP.clean, geom="violin", fill=Res)
```

3.6.4 Histograms and Density plots

By default, if you plot just a single continuous variable using `qplot()`, `R` will plot a histogram. Histograms are extremely useful for seeing the distribution of your data. Do they look normally distributed? Are they skewed to one side or the other? There is no need to specify a `geom` here, but if you want to specify `geom="hist"`, you can.

```
> #Make a basic histogram
> qplot(x=Mass.final, data=RxP.clean)
```

The same principle of using the color or fill arguments as a way to view your data apply to histograms, but with one caveat. If you add a fill or color argument to a histogram in `qplot()`, `R` will make a *stacked* histogram. It can be more useful to see the data distributions overlayed on one another. This is best achieved with a *density plot*, which is similar to a histogram but instead plots a smoothed line that shows the shape of the data. Note that you should use the `col` argument in the density plot instead of the `fill` argument. What happens if you do not?

```
> #Make a stacked histogram
> qplot(x=Mass.final, data=RxP.clean, fill=Pred)
> #Make overlayed density plots
> qplot(x=Mass.final, data=RxP.clean, geom="density", col=Pred)
```

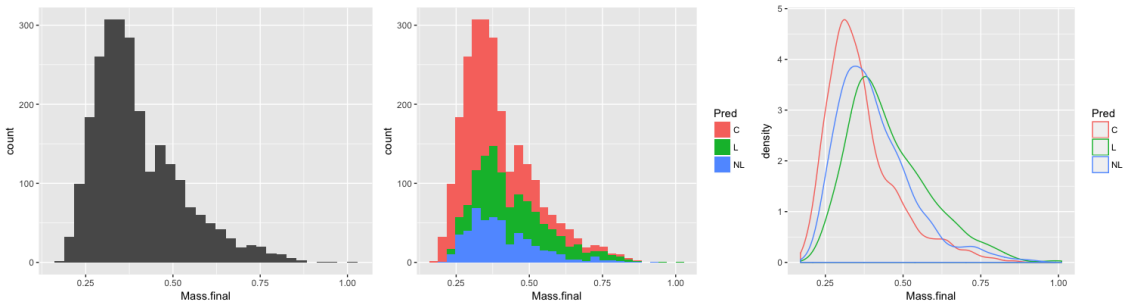

Figure 11: Three ways to view the distributions of the data, the final mass of *A. callidryas* metamorphs, made with `qplot()`. The plot on the left is standard histogram. The middle plot is a stacked histogram, showing which data contribute to the overall shape of the histogram. The plot one on right is a density plot.

3.6.5 Scatterplots

The same principle works for continuous response variables. Above, we defined our x-axis as a categorical variable, but if we instead use a continuous variable `R` will plot a scatterplot. Similarly, we can use facets or colors to visualize the variation in our data, which is extremely useful. For example, in the code below I've filled the points based on the Resource treatment, and faceted the data based on the predator treatment. Imagine the possibilities!!

```
> #Make a series of scatter plots
> qplot(x=log(SVL.final), y=log(Mass.final), data=RxP.clean, col=Res, facets=.~Pred)
```

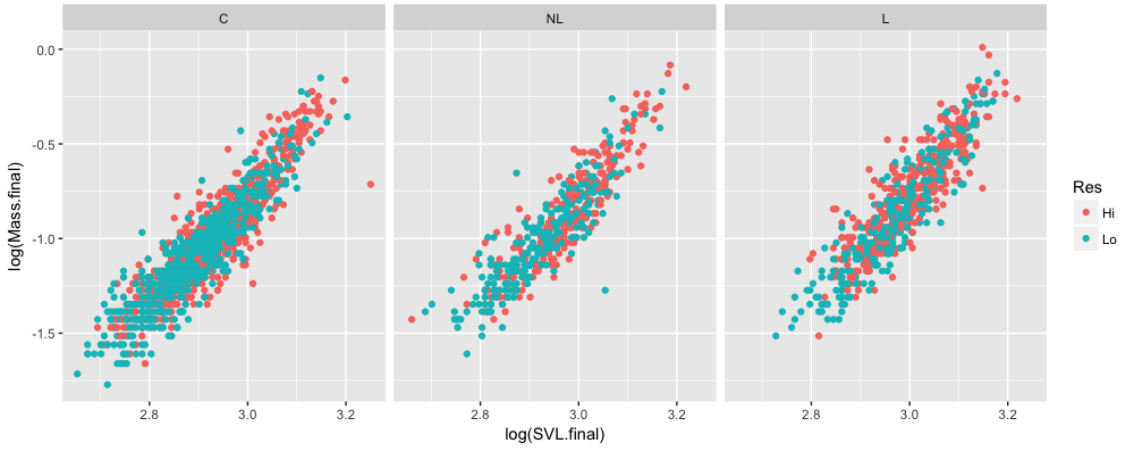

Figure 12: The log of the final mass of metamorphs plotted against the log of their SVLs, plotted with `qplot()`. The data are faceted by predator treatment and points are colored by resource treatment.

4 Summarizing and Manipulating Your Data

There are many tools in R to help you summarize your data efficiently. One old function that is useful to know about is `tapply()`, however the functions `aggregate()` and `summarise()` are what you are more likely to use.

`tapply()` is a simple function for calculating the mean or standard deviation of a group of data. The name of the function is short for 'table apply', as you are *applying* a function to a *table* of data. `tapply()` requires three arguments: 1) the data you are summarizing, 2) the category you want to summarize the data by, and 3) the function to apply to those data. For example, if you want to calculate the average size of metamorphs in each of the predator treatments, you can type:

```
> tapply(RxP.clean$SVL.initial, RxP.clean$Pred, mean)
       C        L       NL
18.13149 19.43098 18.90486
```

The main disadvantage of using `tapply()` is that it returns a list, which is much more difficult to work with than a data frame. `tapply()` can be useful for very simple tasks, but is not that great in general.

`aggregate()` is a similar function but the output is slightly different. It is better for summarizing data across multiple groups. `aggregate()` can be coded in two different ways. 1) It can take the same three arguments as `tapply()` but the second argument is in the form of a list, which allows you to easily include multiple factors. More useful is 2) you can formulate the expression of how you want your data summarized *in the exact same format* you use for specifying plots or models (which we will cover in Module 3). Another advantage of `aggregate()` is that the output is a data frame, which makes it easy to then use the output for plotting figures or other purposes. For example, if you want to calculate the average size of metamorphs at emergence across all combinations of Food and Resource treatments, you can type either of the two methods below (I highly recommend the second form, which is shorter to type, more intuitive, and provides a better output).

```
> aggregate(RxP.clean$SVL.initial, list(RxP.clean$Pred, RxP.clean$Res), mean)
  Group.1 Group.2        x
1       C      Hi 18.37231
2       L      Hi 19.64221
3      NL      Hi 19.39076
4       C      Lo 17.87029
5       L      Lo 19.14696
6      NL      Lo 18.41102

> aggregate(SVL.initial~Pred*Res, mean, data=RxP)
  Pred Res SVL.initial
1    C  Hi    18.37851
2    L  Hi    19.64221
3   NL  Hi    19.38520
4    C  Lo    17.86433
5    L  Lo    19.15724
6   NL  Lo    18.41102
```

Lastly, the package `dplyr` has many useful functions for data wrangling. By using some of these in combination, you can easily summarize yoru data. The utility of this may not be completely evident now, but it will be later, I promise. The downside of `dplyr` is that, much like `ggplot2`, it has it's own lexicon that is fairly distinct from the rest of `R`, meaning that you have to learn a whole different set of commands. So be it. It's pretty great once you learn the coding.

By using a few choice functions, such as `group_by()` and `summarise()`, we can easily say we want to calculate the means of whatever variable (or variables) we are interested in. Note of course that you can use different functions besides `mean()`. The other important thing to note in the code below is the use of the pipe command, %>% which designates the output of one line to be the input of the next.

The code below does several things all at once, so let's walk through it. The first line defines what data frame we are going to be using, then 'pipes' it to the next line. The second line says we are going to be doing *something* and we are going to be grouping the data by the resource and predator treatments, which are 'piped' to the next line. Lastly, the third line says that we will calculate a new column and we will call it `SVL.mean`, and it will be calculated as the mean (which is of course a function) of the data in the column called `SVL.initial`. Since the `summarise()` function has inherited the lines above it, it knows to find the data in the `RxP.clean` data frame and to calculate means of all the combinations of our `Res` and `Pred` groups. Don't forget to load the `dplyr` package!

```
> library(dplyr)
> temp<-RxP.clean %>%
        group_by(Res, Pred) %>%
        summarise(SVL.mean = mean(SVL.initial))
> temp
# A tibble: 6 x 3
# Groups:   Res [?]
     Res   Pred SVL.mean
  <fctr> <fctr>    <dbl>
1     Hi      C 18.37515
2     Hi      L 19.64221
3     Hi     NL 19.39076
4     Lo      C 17.87029
4     Lo      L 19.14696
5     Lo     NL 18.41102
```

A few other things to note. The object that was created by `summarise()` is called a 'tibble', as in the pronounced version of the abbreviation 'tbl'. It is just like a data frame, but instead shows you the type of data in each column. You can turn it in to a standard data frame by using the function `as.data.frame()` to redefine it. This is not only useful, but necessary, in some situations.

```
> str(temp)
Classes grouped_df, tbl_df, tbl and 'data.frame': 6 obs. of  3 variables:
 $ Res     : Factor w/ 2 levels "Hi","Lo": 1 1 1 2 2 2
 $ Pred    : Factor w/ 3 levels "C","NL","L": 1 2 3 1 2 3
 $ SVL.mean: num  18.4 19.4 19.6 17.9 18.4 ...
 - attr(*, "vars")= chr "Res"
 - attr(*, "drop")= logi TRUE
> temp<-as.data.frame(temp)
```

```
> str(temp)
'data.frame': 6 obs. of  3 variables:
 $ Res     : Factor w/ 2 levels "Hi","Lo": 1 1 1 2 2 2
 $ Pred    : Factor w/ 3 levels "C","NL","L": 1 2 3 1 2 3
 $ SVL.mean: num  18.4 19.4 19.6 17.9 18.4 ...
```

4.1 Reordering your data

We are going to use `aggregate()` to create a bar graph of mean metamorph size. However, you might have noticed by now that R orders the levels of a factor alphabetically, so that "C" comes before "L", which comes before "NL". For plotting purposes, it would probably make more sense to have them ordered logically (Control, Non-lethal, Lethal, or vice-versa). To see the current ordering of the factor levels, type:

```
> levels(RxP.clean$Pred)
[1] "C" "L"  "NL"
```

As with most things in R, there multiple ways to reorder the factor levels. Here are two examples.

1. One way would be to use the function `relevel()`. Using `relevel()` we can specify a new first level and all other levels will shift to the right. Thus, to get the ordering "C, NL, L", from the current order we have to first specify "NL" as the new first level (so it is ranked ahead of L), then repeat the command to specify "C" as the first level (moving it ahead of both NL and L).

   ```
   > RxP.clean$Pred<-relevel(RxP.clean$Pred, "NL")
   > RxP.clean$Pred<-relevel(RxP.clean$Pred, "C")
   ```

   Note that we did not change the data at all, just the way the levels of the factor are considered.

2. A second way to have changed the order would be use the function `factor()` to recode all the levels of the factor at once. `factor()` is a generic function which could be used, for example, to turn a vector of text or of 0's and 1's into factor levels. Here, we are essentially writing over the current factor with a new factor that is the same, but has a different order for the factor levels.

   ```
   > RxP.clean$Pred<-factor(RxP.clean$Pred, c("C","NL","L"))
   ```

Why would you choose to use one method over another? The first method, using the `relevel()` function is more cumbersome, but ensures you won't make mistakes because it only allows you to change one thing at a time. The second technique, using the `factor()` function, allows you to change the order to how you want it with only a single command, but has the possibility to introduce mistakes if you make a typo. Any mistakes will just become a new level in the data, even if that level does not exist in the data.

4.2 Calculating means and standard errors

Now that our Pred factor is in the order we want, let's use `dplyr` to make a new data frame which contains the means and standard errors of the data. There is no built in function to calculate the standard error in `R` (there are multiple definitions of "standard error" and the `R` guru's could never agree on a single definition of what exactly a standard error is), so we will have to calculate it ourselves. For this exercise, we will use the most common and widely accepted definition: standard deviation/square root of N.

In the code below, notice that I've begun annotating my code using the #. In R, the # is a way to make comments to yourself. Anything that follows a # will not be executed by the program and so it is a great way to make notes to yourself about what you are doing and why you are doing it. When you come back to your analyses in a week, a month or a year, you need to have notes to remind yourself of what you were doing. You should have noticed this earlier as well. Always leave notes in your script file for your future self.

The other important thing to notice is the use of a new function, `mutate()`, which is how you create new variables in the world of `dplyr`. Similarly, if we want to access the values we have just calculated for plotting in any function besides `ggplot2`, we will need to convert the tibble into a regular data frame.

```
> met.plot<-RxP.clean %>% #Define what dataset we are using
  group_by(Res, Pred) %>% #Set the groups
  summarise(SVL.mean = mean(SVL.initial), SVL.sd = sd(SVL.initial),
  SVL.n = length(SVL.initial)) %>% #Calculate the mean, sd and N of SVL
  mutate(SVL.se = SVL.sd/sqrt(SVL.n)) #Calculate the SE

> met.plot
# A tibble: 6 x 6
# Groups:   Res [2]
     Res   Pred SVL.mean    SVL.sd SVL.n    SVL.se
  <fctr> <fctr>    <dbl>     <dbl> <int>     <dbl>
1     Hi      C 18.38518 0.6963069    16 0.1740767
2     Hi     NL 19.11539 1.0212486     7 0.3859957
3     Hi      L 20.08653 1.1728881    16 0.2932220
4     Lo      C 17.86738 0.7565341    16 0.1891335
5     Lo     NL 18.46814 0.8001960     7 0.3024457
6     Lo      L 19.86672 1.4539000    16 0.3634750

> met.plot<-as.data.frame(met.plot) #Convert the tibble into a standard data frame
```

5 Plotting Your Data

Now that we have those values we can create a nice looking bar graph. The most basic function to make a bar graph is `barplot()`. There are many, many arguments that can be passed to `barplot()`, which can be viewed in the help file (remember how to get to the help: `?barplot`). A slightly improved version is the function `barplot2()`, which makes plotting error bars much simpler. `barplot2()` is found in the `gplots` package. We will cover making barplots in `ggplot2` in a later module (hint: it's not as easy as you might hope).

Let's start by simply plotting the bars in their most basic form.

```
> barplot2(met.plot$SVL.mean)
```

We have passed to `barplot2()` the variable of mean tadpole sizes, which have been plotted in order from top to bottom. However, this figure by itself is quite boring. It has no axes titles, color, error bars, etc.

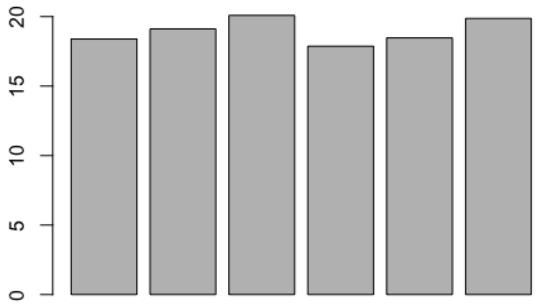

Figure 13: The most boring barplot of metamorph SVL imaginable

We can add a label to the y-axis with the argument `ylab="axis label goes here"` (short for "y label"). Similarly, we can use the command `ylim=c(min,max)` (short for "y limits") to change the y-axis to be between 15 and 20 in order to better see the differences between our treatments .

Since the first three bars are the High Resource tadpoles, and the second three bars are the Low Resource tadpoles, those sets of bars should probably be grouped together. We can do this by changing the spacing between them, with the `space=c()` argument. By default, each bar is 1 unit wide with 0.2 units between them. You can look at the bars and see that there is even spacing before the first bar and between each bar. The spacing starts at the y-axis, so our vector of spaces needs to include the space before the first bar. What you pass to the `space` argument is a vector of numbers that defines how much space goes in between each bar. Since there are 6 bars, there are 5 gaps between them, plus the gap before the first bar, so we need a vector of 6 values.

```
> barplot(met.plot[,3], ylim=c(15,20), ylab="Mean SVL of metamorphs (mm)",
space=c(0,0,0,0.5,0,0))
```

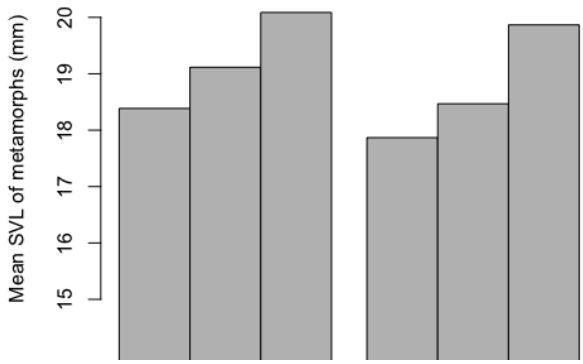

Figure 14: A barplot of metamorph SVL, with bars grouped by Resource treatment

Uh oh! We changed the y-axis, but R is still trying to plot the bars all the way to 0. We can fix that by add the argument `xpd=F`, which will make the bars stop at the x-axis. Also, let's add some color. The two sets of three bars are the Control, Non-lethal and Lethal Predator treatments, from left to right. We can add color using the `col=c()` argument. Colors can be defined in many ways in R.

1. You can use the default colors 1-8 (in order: black, red, green, blue, cyan, magenta, yellow and grey).
2. You can define custom colors using the function `rgb()`. This is great because you can define any color you want based on the RGB color system, which allows you to match colors between your figures and (for example) a Powerpoint or Keynote color scheme (if you are so inclined).
3. You can define custom colors using the hex color system. One advantage of using hex colors is that you can make colors translucent, which can be useful in certain scatterplots of venn diagrams, for example.
4. You can used predefined and named colors viewable by typing `colors()`. If you are curious as to what these are, just type `colors()` into the console and you will get a huge list of the prenamed colors. A google search can help you figure out what they look like.

For this exercise, we will use prenamed colors built in to R. Since these treatments represent varying types of predator treatments, we can make them increasing versions of the same basic color. Why not choose green? Note that since the progression of colors is the same in the first three bars as it is in the second three bars, we only have to define our set of colors once and they will recycle for the second set of three bars.

```
> barplot2(met.plot$SVL.mean, ylab="Mean SVL of metamorphs (mm)",
  space=c(0,0,0,0.5,0,0), las=1, col=c("light green","green","dark green),
  ylim=c(15,20), xpd=F)
```

Now we should think about adding error bars. Error bars are a pain no matter what. In base R they are added with the `arrows()` function. In `barplot2` we can add them inside the plot command by saying `plot.ci=T`, as in "Plot confidence intervals? True", and then defining the location of the top and bottom of the error bar (simply the mean plus SE and the mean minus SE).

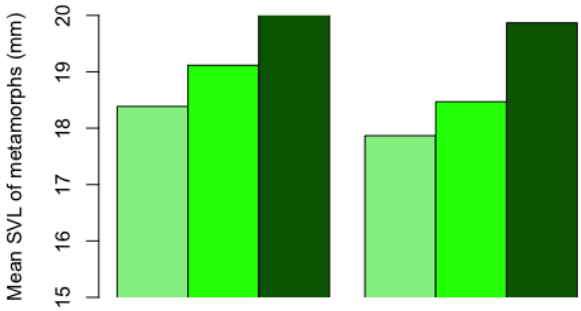

Figure 15: A barplot of metamorph sizes, with bars grouped by Resource treatment and colors indicating the Predator treatment

```
> barplot2(met.plot$SVL.mean, ylim=c(15,20), ylab="Mean SVL of metamorphs  SE (mm)",
  space=c(0,0,0,0.5,0,0), xpd=F, col=c("light green","green","dark green"), plot.ci=T,
  ci.u=met.plot$SVL.mean+met.plot$SVL.se, ci.l=met.plot$SVL.mean-met.plot$SVL.se)
```

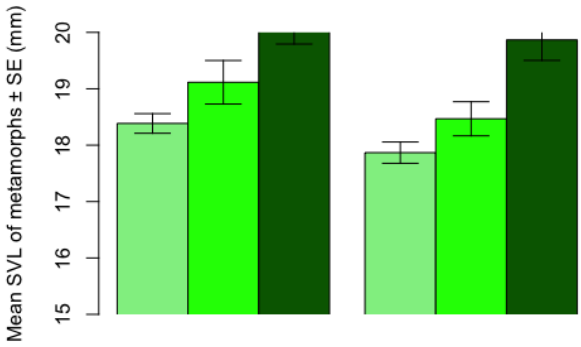

Figure 16: A barplot of metamorph SVL with standard error bars. Bars are grouped by Resource treatment and colors indicate the Predator treatment

Oh no! We successfully added error bars but they extend beyond our plot window. In the next iteration, we need to increase our y-limit to 21. In addition, let's go ahead and add some labels to the x-axis using the `axis()` function. Once again, this is a separate function from `barplot()`. Like `arrows`, `axis()` has many different arguments that can be passed to it.

```
> axis(side=1, at=c(1.5, 5), labels=c("High resource diet", "Low resource diet"))
```

Within the function `axis()` we have passed arguments to define 1) which side we want the axis on (1=bottom), 2) where on the x-axis we want the labels to appear (at 1.5 and 5) and 3) what we want the labels to say.

Lastly, we can add a legend to specify what the different colored bars indicate. Once again, this uses a separate function `legend()`.

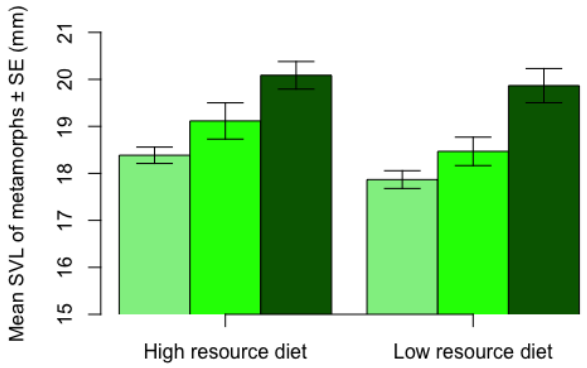

Figure 17: A barplot of metamorph SVL with standard error bars. Bars are grouped by Resource treatment and colors indicate the Predator treatment

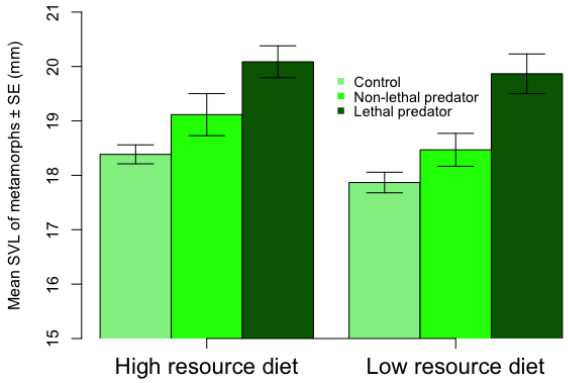

Figure 18: A very complete barplot of mean metamorph SVL with standard error bars. Bars are grouped by Resource treatment and colors indicate the Predator treatment

```
> legend(x=3.5, y=20, legend=c("Control","Non-lethal predator","Lethal predator"),
col=c("light green","green","dark green"), bty="n", pch=15)
```

The `legend()` command above contains arguments to define 1) where we want the legend in x and y coordinates, 2) what we want the legend to say, 3) that we want to put squares next to the next (the `pch=16` part), 4) the colors of the squares (the same as our bars) and 5) that we do not want to put a box around the legend (the `bty="n"` part). It often takes a bit of trial and error to find the perfect spot for the legend.

Thus, in the end, we need four functions to make this figure. `barplot()`, `arrows()`, `axis()`, and `legend()`. That may seem cumbersome, but once you have made one, it is very easy to modify the code to make others. And you can easily change the data (the tadpole.mean and tadpole.SE objects) and remake the figure instantly.

Here is the final set of code to make the figure.

```
> barplot2(met.plot$SVL.mean, ylim=c(15,21), ylab="Mean SVL of metamorphs  SE
(mm)", space=c(0,0,0,0.5,0,0), xpd=F, col=c("light green","green","dark green"),
plot.ci=T, ci.u=met.plot$SVL.mean+met.plot$SVL.se, ci.l=met.plot$SVL.mean-
met.plot$SVL.se)
> axis(side=1, at=c(1.5, 5), labels=c("High resource diet", "Low resource diet"))
> legend(x=3.5, y=20, legend=c("Control","Non-lethal predator","Lethal predator"),
col=c("light green","green","dark green"), bty="n", pch=15)
```

Lastly, here are a couple of other useful tips when plotting.

- Typing the function `box()` at the command prompt will draw a box around your plot.
- You can add hatching to your bars with the `density=` and `angle=` arguments. `density` tells R how close together to make the lines and `angle` determines the angle to plot the lines at.
- You can shortcut the placement of the legend by just assigning the x-coordinate to be (for example) `"topright"` or `"bottomleft"`. Choose the basic location based on the part of your plot with the most whitespace. Look at the help function for `legend` for all the built in positions.

6 Assignment!

Here are some things to do on your own.

1. Use `qplot()` to make a 2x3 faceted plot of age at metamorphosis against final metamorph SVL, adding informative axes labels (hint: the `ylab` argument also works in `qplot()`).
2. Make a barplot like above, but instead of using the initial SVL data, make it from a) the final metamorph mass data and b) have it grouped by predator treatment and hatching age. Make sure it has correct error bars and give it some creative colors!

Turn in to me (via Moodle) a word doc or pdf with both figures. The assignment is due the evening of Wednesday 2/7/2018.