

# Introduction to Ecological Analysis in **R** (Day 1)

Matthew Lau

November 4, 2008

Today, we covered lot of ground with the goal of providing the tools to input and manipulate data, conduct analyses and make plots. Below, I have detailed all of the activities that we did in R.

1. Scripting
2. Math Operations
3. Help!
4. Data Entry
5. Statistical Analyses
6. Plotting
7. Next Class

## 1 Scripting

Perhaps the most important thing we learned above everything else is to use a script editor. Do not work solely in the console command line. Please, open a new script file to work in by going to the file menu and selecting a "New Document". This document can be saved for future use, unlike the R Console, which will save what you have done, but not in a reproducible format.

When you script you can write out your code and then run it by placing the cursor on the line or highlighting all of the script you want to run and running it (Windows users press CTRL + R and Mac users press COMMAND or APPLE + ENTER). You can add notes by using the pound symbol, #, which tells R not to run any text to the right of that symbol on that line.

## 2 Math Operations

Math operations are basic and intuitive, but are essential tools for working effectively in R.

## 2.1 Basic Math Operations:

```
> 1 + 1
```

```
[1] 2
```

```
> 1 - 1
```

```
[1] 0
```

```
> 2 * 2
```

```
[1] 4
```

```
> 2/2
```

```
[1] 1
```

```
> 2^2
```

```
[1] 4
```

```
> (2 + 2) * 2
```

```
[1] 8
```

## 2.2 Math Commands

Commands (aka. functions) are the meat of R. They allow us to simplify tasks. The basic structure of a command is `command(arguments)`. Here we can see the mechanics of any command. Simply, the command tells R what you want to be done to what's in the parentheses (i.e. the *arguments*).

```
> sqrt(2)
```

```
[1] 1.414214
```

```
> log(2)
```

```
[1] 0.6931472
```

```
> cos(2)
```

```
[1] -0.4161468
```

There also several statistically relevant math commands that I'll just mention now, but we'll use them later: `mean()`, `sd()` and `length()` return the mean, standard deviation and number of elements of a vector of numbers.

### 3 HELP!

You may have already encountered some issues just with trying to do these simple operations and commands. In order to get around obstacles and improve your R knowledge-base it is important to know how to access help resources. There are many books out there including the R "bible", many of which are listed on the R website. I typically rely on two resources, the internal help files within R and internet search engines. You can reach help within R in a number of ways, but one of the easiest is to use the `help()` command. For example:

```
> help(sqrt)
```

This will bring up a help file with information pertaining to the `sqrt()` and other related commands.

### 4 Data Entry

Handling data is vital to successful analyses. Bringing data into R can be done in at least two ways. First, you can enter data manually, not unlike a calculator, and second, you get input data via file reading commands.

#### 4.1 Entering Data Manually: Making Vectors and Matrixes

R will support many different data formats. Here is how you can create vectors and matrixes.

```
> 1:10
```

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

```
> c(1.1, 25, 3.32, 4, 12, 14, 85)
```

```
[1] 1.10 25.00 3.32 4.00 12.00 14.00 85.00
```

```
> matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3,  
+       ncol = 3)
```

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

```
> array(c(1, 2, 3, 4, 5, 6, 7, 8, 9), c(3, 3))
```

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

## 4.2 Reading Data from File

More often than not, we already have our data saved as a data file. Here are two commands that you can use to bring data into R. Note that both of these commands require the data to be in a specific format. Check the help file for the commands to see how the data should be structured.

`read.table("file location")` will read a text file.

`read.csv("file location")` will read a .csv file. This is most useful when you have your data organized as spreadsheets, which many people do. For example:

```
> read.csv("/Users/artemis/Desktop/rock_lichen_data.csv")
```

	moth	canopy	A
1	2	14.3	7.2
2	2	28.5	2.8
3	2	34.9	4.6
4	2	43.6	5.2
5	2	169.9	25.4
6	2	38.0	0.0
7	2	102.4	9.7
8	2	64.8	9.9
9	2	39.2	1.8
10	2	39.2	4.7
11	1	320.0	39.9
12	1	206.4	58.0
13	1	194.9	39.1
14	1	135.2	18.3
15	1	89.8	12.4
16	1	417.7	44.1
17	1	164.5	24.1
18	1	155.7	34.9
19	1	155.1	43.1
20	1	169.0	19.0

This is the standard formatting for the `read.csv` command. Data are organized into columns with the columns headed by their appropriate names. Column names should not contain spaces or math operations and should not start with numbers.

## 4.3 Making Objects

Here you may have already noticed a missing piece of the puzzle. Although we can create these nifty vectors and matrixes, R simply spits them out and that's it. It would be very useful to be able to call and use them again. We can use objects to do this for us. Just as we can use symbols in math to represent numbers (e.g.  $a + b = c$ ),

we can use different symbols to represent or "store" our data. This is done by using either the equals sign "=" or by making a left arrow with the less-than sign "<" and a dash "-". For instance:

```
> a = 10
> a
```

```
[1] 10
```

```
> a <- 10
> a
```

```
[1] 10
```

You can see that the first line above tells R that "a" is the number 10, because when we enter "a" R returns the number 10. Note that object names: a) cannot start with a number, b) are case sensitive and c) cannot contain math operators (e.g. + , - , / , etc.).

We can also make much more complex objects with vectors and matrixes, as well as data frames, data tables and lists. For example, we can create an object from a data matrix read in from a file:

```
> data <- read.csv("/Users/artemis/Desktop/rock_lichen_data.csv")
> data
```

	moth	canopy	A
1	2	14.3	7.2
2	2	28.5	2.8
3	2	34.9	4.6
4	2	43.6	5.2
5	2	169.9	25.4
6	2	38.0	0.0
7	2	102.4	9.7
8	2	64.8	9.9
9	2	39.2	1.8
10	2	39.2	4.7
11	1	320.0	39.9
12	1	206.4	58.0
13	1	194.9	39.1
14	1	135.2	18.3
15	1	89.8	12.4
16	1	417.7	44.1
17	1	164.5	24.1
18	1	155.7	34.9
19	1	155.1	43.1
20	1	169.0	19.0

## 5 Statistical Analyses

### 5.1 t-test

A t-test is a very common parametric analysis of ecological data, when we have one or two samples that we would like to analyze. Here we looked at an example where we were interested in testing whether or not habitat restoration had improved the abundance of tigers in a preserve with a set of tiger counts of tigers taken once a month for six months. We did a t-test in two different ways. The first way we used a command that comes in the R base package, `t.test()`:

```
> tigers <- c(10, 14, 11, 12, 10, 18)
> t.test(tigers)
```

```
One Sample t-test
```

```
data: tigers
t = 9.934, df = 5, p-value = 0.0001765
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 9.265422 15.734578
sample estimates:
mean of x
 12.5
```

This conducts a t-test using the default settings (see the help file) of the population mean different from zero. One problem is that it is a two-tail test, which we don't want (counts can never be less than zero). So we can re-run it specifying the argument "alternative" as "greater" for a one-tail test of the mean greater than zero:

```
> t.test(tigers, alternative = "greater")
```

```
One Sample t-test
```

```
data: tigers
t = 9.934, df = 5, p-value = 8.823e-05
alternative hypothesis: true mean is greater than 0
95 percent confidence interval:
 9.964453      Inf
sample estimates:
mean of x
 12.5
```

Notice that "greater" is inside of quotes. This tells R that it is text, not an object.

We also programmed our own t-test from scratch using what we new about the mechanics of the t-test. First we calculated our observed t, and then we computed a p-value for that t:

```
> tobs <- (mean(tigers) - 0)/(sd(tigers)/sqrt(length(tigers)))
> pt(tobs, df = length(tigers) - 1, lower.tail = FALSE)

[1] 8.82312e-05
```

Here the `mean()` and `sd()` commands are computing our mean and standard deviations for us. The `length()` command yields the number of elements in our data vector "tigers" which is also our sample size (i.e.  $n$ ). The `pt()` command gives us our p-value. The "lower.tail" argument specification gives us the area under the  $t$  distribution to the right of our observed  $t$ .

We see that we get both an observed  $t$  and p-value exactly equal to the results from our command, `t.test`.

## 5.2 Checking Assumptions

There are two assumptions of a  $t$ -test. The first, that the sample is representative of the population, is perhaps the most important but hard to establish. The second, that the population is normally distributed, can easily be checked with several simple diagnostics: the Shapiro-Wilks test, quantile-quantile (QQ) plotting and histogram. Here is the code to get these three tools:

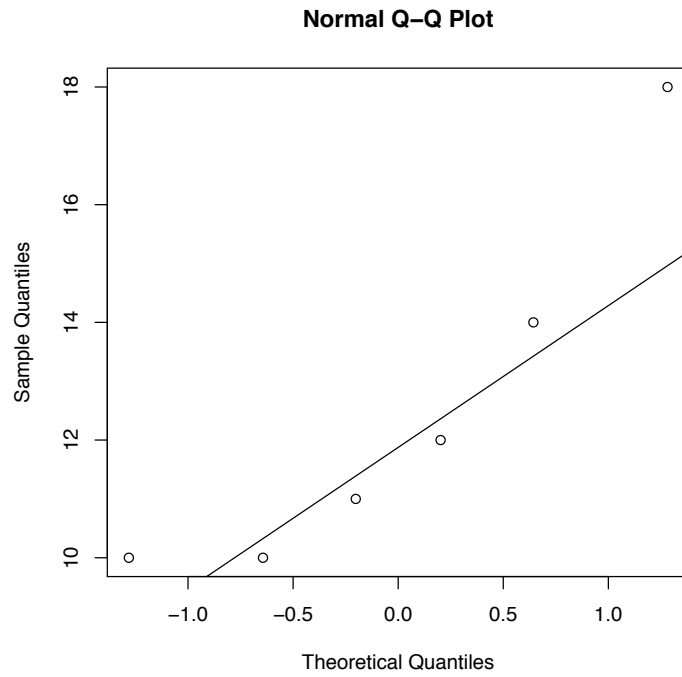
```
> shapiro.test(tigers)

      Shapiro-Wilk normality test

data:  tigers
W = 0.8502, p-value = 0.158
```

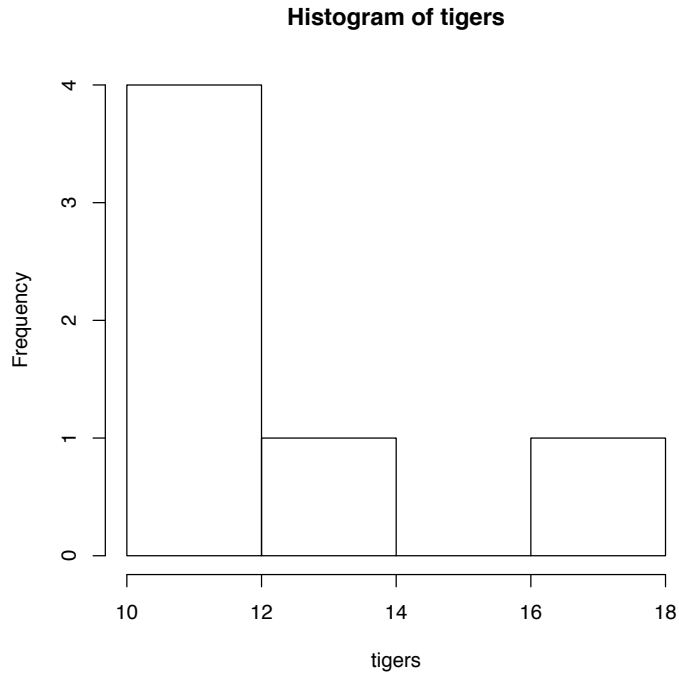
This command, `shapiro.test`, runs a Shapiro-Wilks test with the null-hypothesis that the data were drawn from a normal distribution.

```
> qqnorm(tigers)
> qqline(tigers)
```



The `qqnorm` command produces a QQ plot and the command `qqline` draws a one to one line on top of the points. Note that you must first produce the QQ plot before drawing the QQ line.

```
> hist(tigers)
```



This command is self-explanatory. It produces a histogram of the frequency of various data values.

The Shapiro-Wilks test does not refute the hypothesis of normality, both the QQ plot and the histogram show deviations from normality. In this case, since we have such a small sample size and likely have low power to reject the null hypothesis of normality in the Shapiro-Wilks test, we should go with the evidence from the QQ plot and the histogram that suggest non-normality.

### 5.3 Non-parametric Test: Wilcoxon Signed-Rank Test

Although the t-test has been shown to be robust to non-normality, we know that count data are not usually normally distributed. Typically they follow a poisson distribution. So, regardless of the conclusions we drew from our normality investigations, it would be prudent to use a test that does not make an assumption of the underlying population distribution. In this case, where we are conducting inference about a single sample, we can use the Wilcoxon Signed-Rank Test. This is easily done by using the `wilcox.test` command from the R base package.

```
> wilcox.test(tigers, alternative = "greater")

Wilcoxon signed rank test with continuity
correction
```

```
data: tigers
V = 21, p-value = 0.01776
alternative hypothesis: true location is greater than 0
```

Here we see that the results, although different, yield the same answer as our t-tests.

## 5.4 Regression

In R there are two steps to running a regression. First we need to fit a model. Here we can use the `lm` command to fit a least squares regression model to the data that we loaded earlier:

```
> data
> attach(data)

> lm(A ~ canopy)

Call:
lm(formula = A ~ canopy)

Coefficients:
(Intercept)      canopy
      2.5353         0.1368

> fit <- lm(A ~ canopy)
```

Notice that we used the `attach` command. This tells R to make objects from the columns. This object consists of a vector named by the column name. This allows us to use these objects in our `lm` command. Notice the structure of the argument here. This is our model specification. It consists of the response variable (A), a tilde `~` signifying regression and our predictor variable (canopy). The `lm` command fits the model and generates the typical output that you need to conduct the regression analysis. Thus, we save this output to a new object, `fit`. This will contain different components within it: such as the fitted values and the residuals. We can access these components using a dollar sign `$`. For instance, we can pull our residuals out of this output to check the assumption of the residuals being normally distributed:

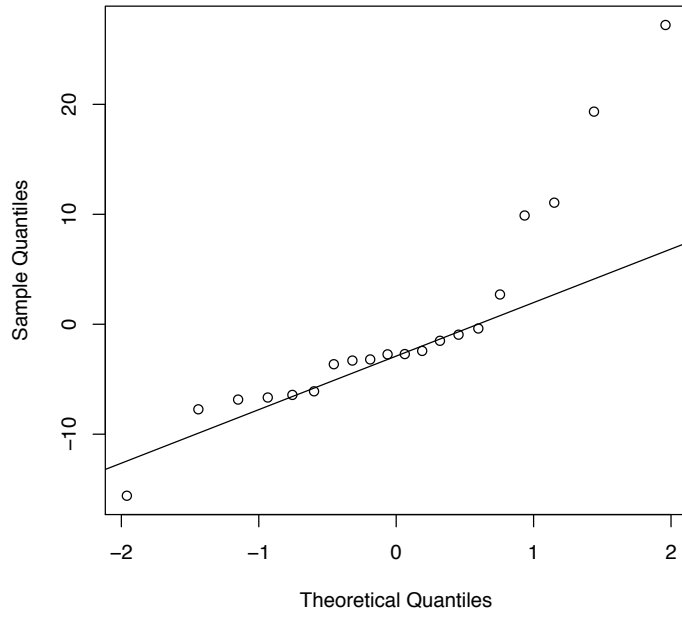
```
> residuals <- fit$residuals
> shapiro.test(residuals)

      Shapiro-Wilk normality test

data: residuals
W = 0.843, p-value = 0.004082

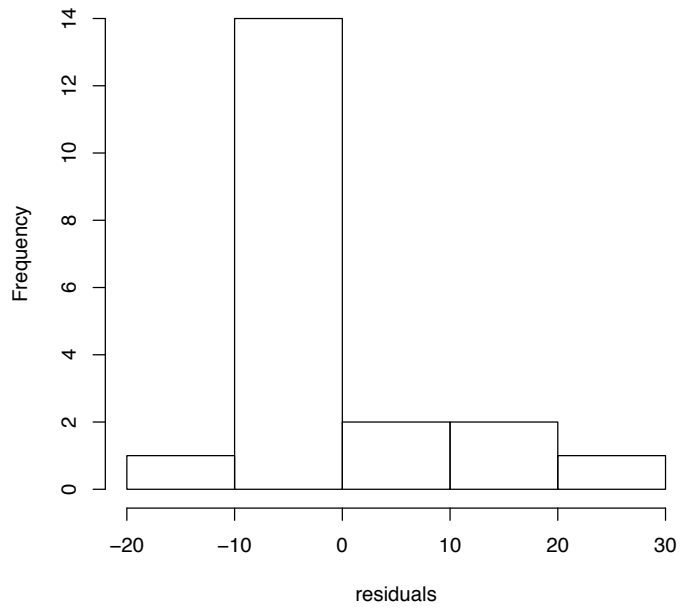
> qqnorm(residuals)
> qqline(residuals)
```

**Normal Q-Q Plot**



```
> hist(residuals)
```

**Histogram of residuals**



Now we can now use this information to conduct a regression analysis (i.e. a hypothesis test for the slope different from zero), using the summary command:

```
> summary(fit)

Call:
lm(formula = A ~ canopy)

Residuals:
    Min       1Q   Median       3Q      Max
-15.5971  -6.1815  -2.7243   0.3875  27.2191

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  2.53526     3.69067   0.687   0.501
canopy        0.13685     0.02246   6.092 9.33e-06 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 10.2 on 18 degrees of freedom
Multiple R-squared:  0.6734,    Adjusted R-squared:  0.6553
F-statistic: 37.12 on 1 and 18 DF,  p-value: 9.332e-06
```

This is our standard F-table, which we can use to conduct our hypothesis test.

## 5.5 ANOVA

Analysis of Variance (ANOVA) is theoretically very similar to regression. It makes sense then that in R conducting an ANOVA is nearly identical to regression. We do only two things differently. First, we need to make sure that we specify our predictor variable, in this case "moth", as categorical. We can do this with the factor command. Second, we can use the anova command to produce an F-table that is more useful for conducting an ANOVA. Overall, the process is still very much the same:

```
> fit.anova <- lm(A ~ factor(moth))
> anova(fit.anova)

Analysis of Variance Table

Response: A
      Df Sum Sq Mean Sq F value    Pr(>F)
factor(moth)  1 3421.7  3421.7  26.599 6.612e-05 ***
Residuals    18 2315.6   128.6
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

One note here is that we could have made a new object for "moth" as a factor and then used the new object in our `lm` command. It's merely a matter of personal preference, R doesn't care:

```
> moth <- factor(moth)
> fit.anova <- lm(A ~ moth)
> anova(fit.anova)
```

Analysis of Variance Table

Response: A

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
moth	1	3421.7	3421.7	26.599	6.612e-05 ***
Residuals	18	2315.6	128.6		

---

Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

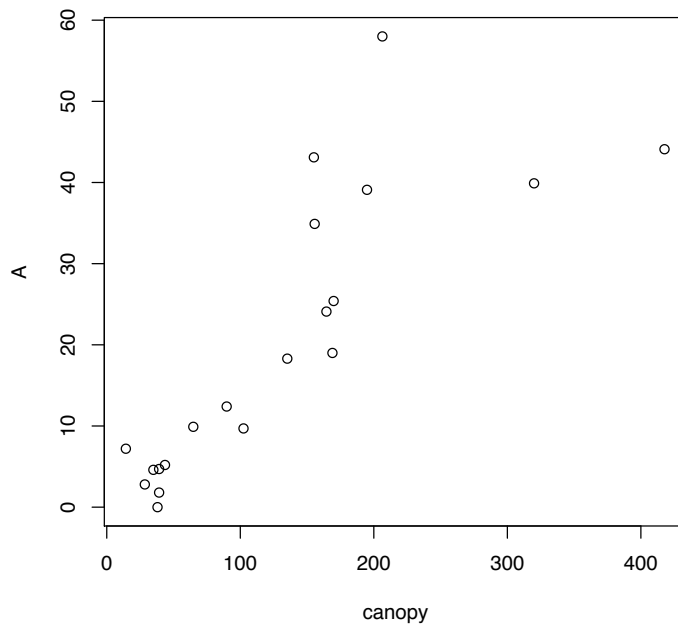
## 6 Plotting

Visualizing data to look for trends is an important aspect of analysis. We can use one command, `plot`, to create great plots that complement both regression and ANOVA.

### 6.1 Scatterplots

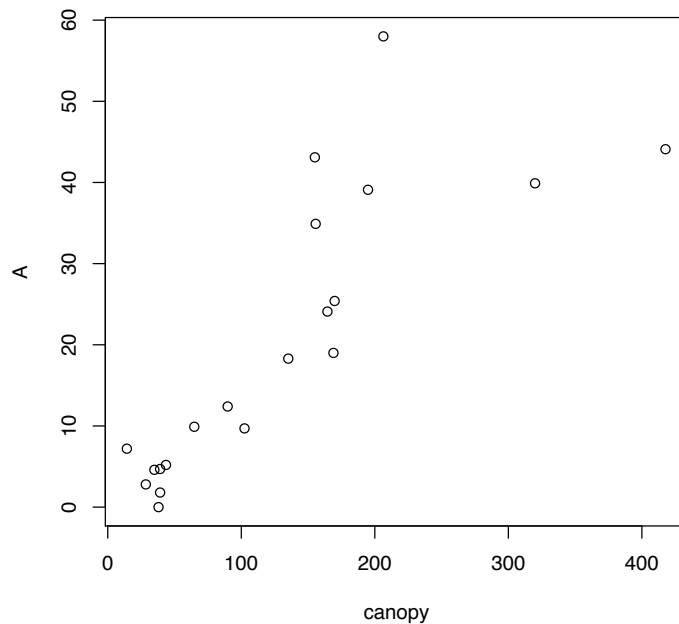
To make a scatterplot, first we simply use the `plot` command:

```
> plot(canopy, A)
```



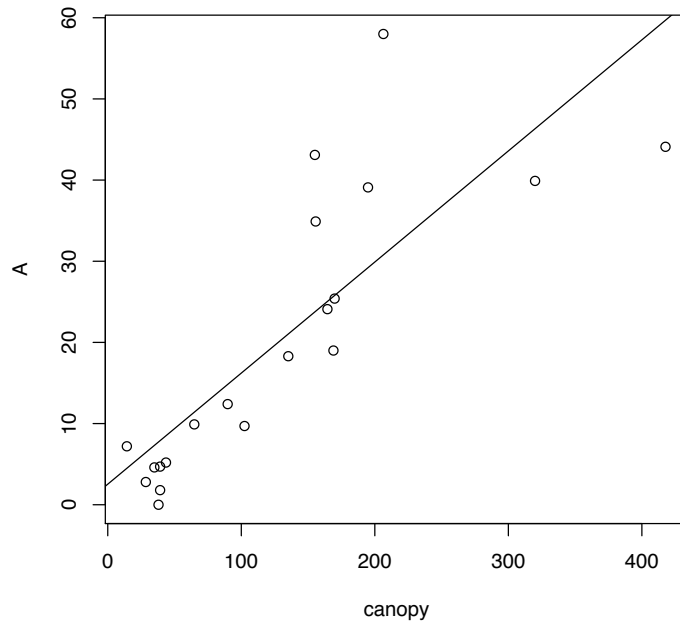
The plot command is expecting the "x" variable first and then the "y" variable second. We could also specify the input as a formula:

```
> plot(A ~ canopy)
```



Either way is valid. I prefer the formula specification, because the text is the same as the model specification in the `lm` command. Now, say we want to add a regression line. We can do this with another command, `abline`:

```
> fit <- lm(A ~ canopy)
> plot(A ~ canopy)
> abline(fit)
```

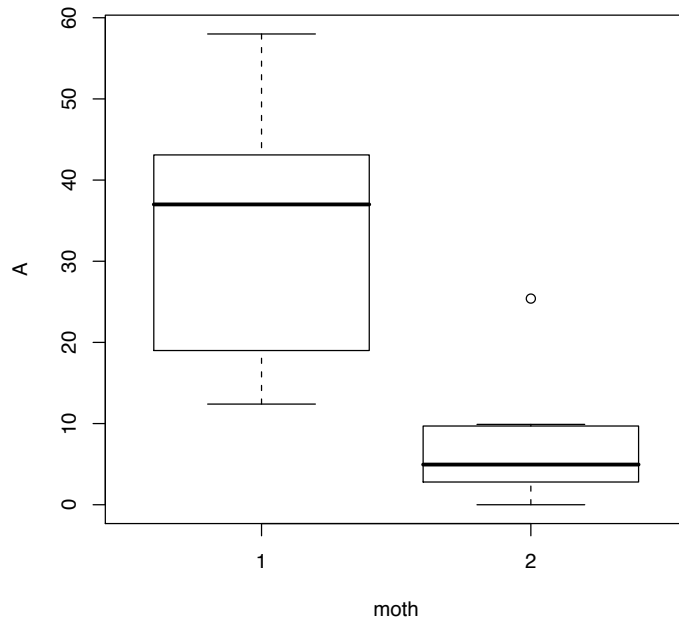


This uses our fitted model, "fit", to draw the regression line.

## 6.2 Box-and-Whisker and Bar Plots

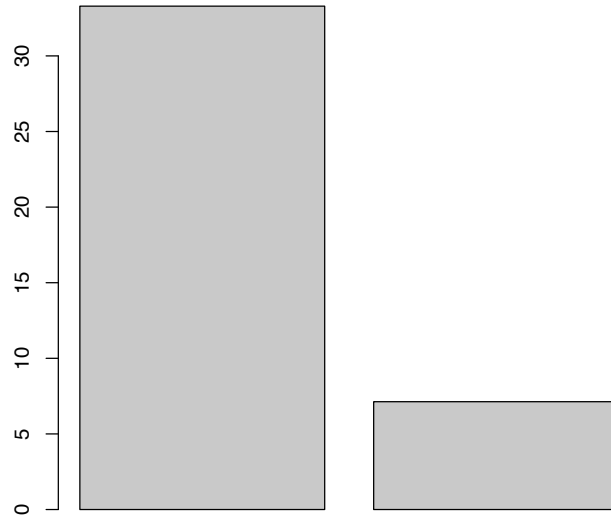
Making a box-and-whisker plot is very much the same process:

```
> moth <- factor(moth)
> plot(A ~ moth)
```



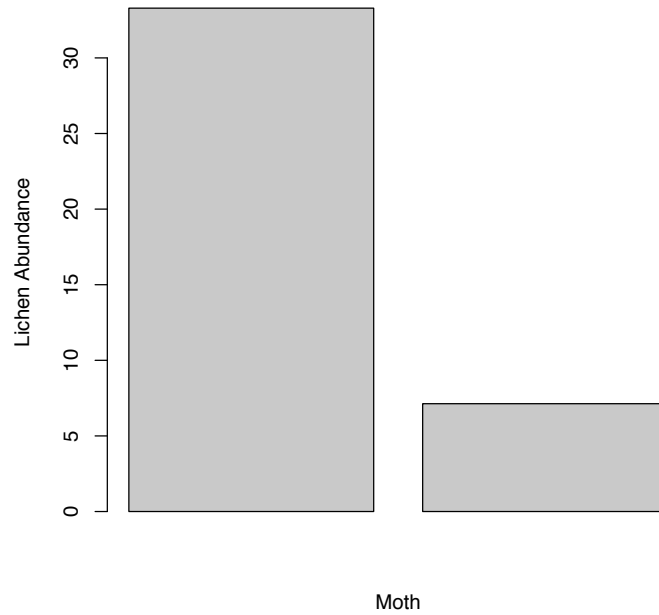
Notice that the plot for "moth" as the predictor is a box-and-whisker plot. If we want a standard barplot we use the `barplot` command, however, we first need the means for each of the levels of "moth"

```
> A.R <- data[11:20, 3]
> A.S <- data[1:10, 3]
> mean.R <- mean(A.R)
> mean.S <- mean(A.S)
> barplot(c(mean.R, mean.S))
```



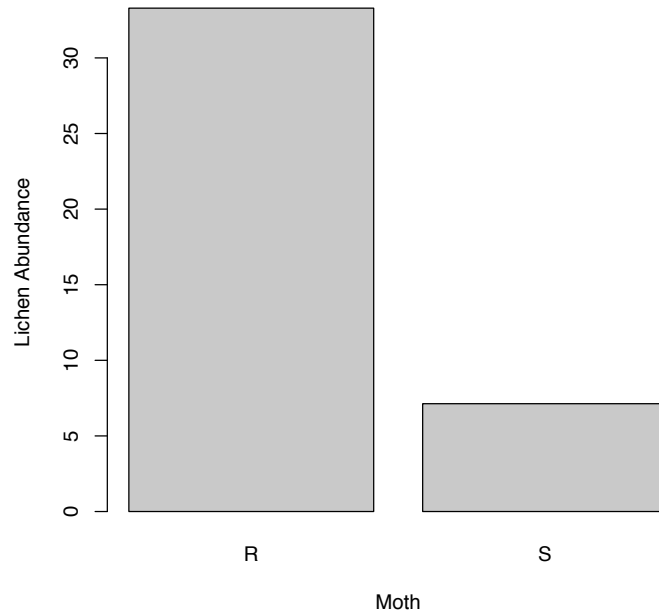
This plot looks pretty poor at this point. There is a whole suite of arguments that can be specified to alter how the plot looks. For instance, you'll notice that the axes have no names. We can add (or change default) axis names by specifying the *xlab* and *ylab* arguments:

```
> barplot(c(mean.R, mean.S), xlab = "Moth", ylab = "Lichen Abundance")
```



We can also add names to the two levels of "moth" by specifying the *names* argument:

```
> barplot(c(mean.R, mean.S), xlab = "Moth", ylab = "Lichen Abundance",  
+         names = c("R", "S"))
```



We could continue tweaking all of the various specifications so that we get the plot exactly like we want.

## 7 Next class:

Next week, we'll go over:

- Improving analysis efficiency
- Changing the working directory
- Analysis of Co-Variance (ANCOVA)
- Making better bar plots
- Begin analyzing community data