R Course BL4005

Tom Reed

8 February 2017

About this Course

This R course is presented by Tom Reed (Lecturer, School of BEES, UCC) but leans heavily on a previous short course developed by Adam Kane and James Savage (postdoctoral fellows at the School of BEES, UCC). It also leans heavily on the excellent book, "Getting Started with R: An Introduction for Biologists", by Andrew P. Beckerman and Owen L. Petchey (see here: http://www.r4all.org/)

Code and further documentation relating to the course can be found on the course's github page: https://github.com/reedtom/R-course-BL4005

The aim of the course is to introduce you to working with R, but more importantly to reduce the trepidation associated with using a command-line interface for the first time. Hopefully, by the end of the course you will have enough confidence to play around with R and start applying it to your own needs. Please feel free to ask questions at any point if something is unclear or you encounter problems.

The Menu for Today:

- Getting started with R
- Getting your data into R
- Exploring your data
- Basic graphs
- Doing basic statistics in R
  - T test
  - Chi-squared test
  - Correlation
  - Regression
  - General linear models

Part 1. Getting started with R

So what is R, exactly??

- R is a programming language focused on statistical computing and graphics.

- Created by reserachers at the University of Auckland, New Zealand, based on an earlier programming language called S.
- Relative new, with the first stable beta version released in 2000.
- R is Open Source software that is freely available under the GNU General Public License, meaning that anyone can use or modify it for any purpose.

Why use R?

- It's Free!!!
- Cross-platform (works on Windows, Mac, Linux)
- A one-stop shop for data exploration, plotting and analysis
  - So no need for multiple programs (Excel, SPSS, Sigma Plot, etc.)
- Makes beautiful figures/graphs
- Reproducibility through R scripts
- By using a command-line (rather than menus), you continually learn a great deal about statistics and data analysis
- Huge range of add-on packages that expand what you can do
- Active community of developers
- Thorough and built-in documentation (help files)

Let's go ahead and open R!

If you have not already downloaded R, follow the next set of instructions.
If you have R already on your machine, start playing with it!

Downloading & Installing R and RStudio

To use R on your own computer, you will need to download R from the Comprehensive R Archive Network (CRAN: https://cran.r-project.org/). Choose the appropriate version of R for your operating system (e.g. Windows 7), then download and install the base distribution. The site will automatically give you the most recently released version of R. CRAN is also where you acquire official packages and extenstions to R.

R studio is an Integrated Development Environment (IDE) program for R, and we recommended it whenever using R for anything beyond the very basics. It makes coding easier by including a text editor that highlights R code syntax, and also provides tools for browsing data and the workspace, and giving better control of graphics and directories. RStudio is availale at https://www.rstudio.com/products/rstudio/download/ for a range of operating systems: choose the appropriate installer and follow its instructions to install.

Let's go ahead and open R!

R basically works like a big calculator

In this document, code will be formatted like this:

```
2 + 2
```

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

Enter the above into the R prompt, then press Enter to execute the code.

Expressions & Objects

At the R prompt we type expressions: commands that tell R to do something. One of the most basic things we will want R to do is to give some data a name, and then store it in memory. This is done using the <- symbol, which is known as the assignment operator. These stored, named chunks of data are called objects. For example, we can define an object called x to have the value 5.

```
x <- 5
```

Once assigned, typing just the name of an object will automatically print its value.

```
x

## [1] 5
```

The [1] before the value indicates that x is a vector (more on this later) and that its first element is 5.

Note that R allows you to overwrite any assignments you have made, and will not give you any warnings or ask for confirmation.

```
x <- 12
x

## [1] 12
```

Luckily, if you ever want to evaluate an expression again, you can press the up arrow key while in the R prompt to select a recent expression, then press Enter to redo the evaluation.

Objects will be stored in memory until they are removed by you, or until you quit the R session. In R studio, you can see what objects are currently stored using the Environment tab in the top right panel.

Calculations & Comparisons

In addition to creating objects, simple operations such as adding, multiplying, and squaring numbers can be performed easily in R using arithmetic operators. R will evaluate the expression, and auto-print the result.

```
6 + 13

## [1] 19

4 - 8

## [1] -4
```

```
345 * 2356
```

```
## [1] 812820
```

```
567 / 13
```

```
## [1] 43.61538
```

```
2^4
```

```
## [1] 16
```

```
(2+8)^2-(54/23)^3
```

```
## [1] 87.05811
```

You can also use operators on variables you have named, and store the results as objects directly. R does not auto-print the result of the expression if it has been stored as an object.

```
height <- 7
width <- 4
area <- height * width
area
```

```
## [1] 28
```

In addition to arithmetic operators, there are also relational operators that compare objects. These will return either TRUE or FALSE. * < less than * > greater than * <= less than or equal to * >= greater than or equal to * == equality * != not equals

```
height > width
```

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

```
height == 7
```

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

```
height != width
```

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

Functions

Most of the work we do with R involves applying Functions to objects or values. Functions are named commands that end with open and closed brackets (). For example, the square-root function **sqrt()**.

```
sqrt(area)
```

```
## [1] 5.291503
```

The objects or values that the function acts on are called "arguments", and go inside the function's brackets. The **sqrt()** function only has one argument, but some functions have

several. Arguments are separated by commas, and have names. For example the function seq() creates a sequence, and has arguments "from" and "to" (among others). The order in which you give the arguments is important, but this is overridden if you name them explicity.

```
seq( 1, 5 )
```

```
## [1] 1 2 3 4 5
```

```
seq( 5, 1 )
```

```
## [1] 5 4 3 2 1
```

```
seq( from = 1, to = 5 )
```

```
## [1] 1 2 3 4 5
```

```
seq( to = 5, from = 1 )
```

```
## [1] 1 2 3 4 5
```

Another super handy function is **mean()**, which...yes, you guessed it... calculates the mean, i.e. arithmetic average, of a set of numbers.

```
x<- seq( 1, 5 )
mean(x)
```

```
## [1] 3
```

We can also combine multiple functions into one, like this:

```
mean(seq(1,5))
```

```
## [1] 3
```

The above reduces two lines of code into one line of code, which can be handy when you get into more complicated programming!!

Many functions have one or two arguments that are necessary, and lots more arguments that act as options to control exactly how the function runs. If these optional arguments are not specified, the function will use default values. For example, **seq()** includes the argument "by" which controls the increment of the sequence. It has the default value of one, but this can be changed to any other number.

```
seq( from = 2, to = 20)
```

```
##  [1]  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

```
seq( from = 2, to = 20, by = 2.3 )
```

```
## [1]  2.0  4.3  6.6  8.9 11.2 13.5 15.8 18.1
```

Getting help

At this point you're probably wondering how you find out what arguments exist for each function. Thankfully, one great thing about R is the amount of built-in help that is available. If you need to know how a function works or what arguments it has, simply type "**?**"" and then the function name. You can also type **help(function)**, or **help("function")** if the function name is a special character. In RStudio, these commands will open the relevant help page in the bottom right panel.

```
?seq
help(seq)
help("^")
```

R also provides examples of how to use many common functions. Try **example(functionName)** if you are still unsure after reading the help.

```
example(seq)
```

Errors & Warning messages

Errors and Warning messages occur when problems occur as R is evaluating code. They are printed in red text and describe what went wrong, often in a way that is difficult to interpret. In the case of Errors, the evaluation of the code stops at the point the error occured. The most common error is probably when you mistakenly ask R to do something with an object that doesn't exist.

```
mass <- volume * density

## Error in eval(expr, envir, enclos): object 'volume' not found
```

Note that the error only stated that the object "volume" was not found, even though the object "density" does not exist either. The evaluation stopped at the point R hit the first error.

Warning messages, unlike error messages, do NOT stop evaluation of the code. Be careful when using any result obtained after a Warning, especially if you don't understand why the message was generated.

```
f <- sqrt(-1)

## Warning in sqrt(-1): NaNs produced

f

## [1] NaN
```

In this case R determined that the square-root of -1 was NaN ("not a number"), but still stored this result as the object f. It is possible to handle complex numbers in R, but we won't be covering them within this course.

A super handy, simple function: c()

The function **c()**, which stands for "combine", is used for combining values into a vector. Essentially, it makes a collection of things:

```
c(1,2,3,4,5)

## [1] 1 2 3 4 5

x<- 120
y<- 6
c(x,y)

## [1] 120   6

beers<- c("IPA", "Pilsner", "Stout")
beers

## [1] "IPA"     "Pilsner" "Stout"

wines<- c("red", "white", "buckfast")
wines

## [1] "red"      "white"    "buckfast"

c(beers, wines)

## [1] "IPA"      "Pilsner"  "Stout"    "red"      "white"    "buckfast"
```

Basic subsetting

Square brackets are used after an object to subset that object, i.e. extract only certain values. For example, the object we called "wines" has 3 elements, which we can see using the **length()** function

```
wines

## [1] "red"      "white"    "buckfast"

length(wines)

## [1] 3
```

This object can then be subsetted as follows:

```
wines[3]

## [1] "buckfast"

wines[1]

## [1] "red"

wines[2]
```

```
## [1] "white"
```

Using R & RStudio

Now let's switch to using R studio, which is a much handier way of keeping track of all your R code.

Once both R and Rstudio have been installed, open RStudio. You should be greeted with three panels: one on the left (Console) and two on the right (Environment/History above Files/Plots/Packages/Help/Viewer). The names above the panels are tabs: try clicking on a tab that isn't currently highlighted to change what that panel shows.

The panel on the left is the R console: at the bottom of this is the R prompt, which is where you type commands to R. You can follow the course examples by entering code into the R prompt yourself, and seeing what output R gives (generally printed in the console). Some of the outputs are not given within this document for reasons of space, so run the code yourself if you would like to see it. When entering multiple lines of code, enter them one at a time so you can easily follow what is going on.

Part 2. Getting your data into R

What we now really want to do is to get our data into R's brain, which is initially empty!

To read in our data, we will use the read.csv function, which is a way to read in "comma seperated values" files.

Here we are going to use a sample dataset, stored in a comma-seperated file called "compensation.csv". These data come from a real experiment that compared Fruit Production in some plant species between Grazed versus Ungrazed conditions. Grazing reduces above-ground biomass, and the researchers wanted to know how fruit production was affected by this reduction. They also measured Initial Root Diameter, as an additional variable of interest that might affect Fruit Production, or otherwise play a role.

First, let's look at the data file in Excel....

Now, we are going to use a script to do a bunch of things in R, including reading in the data.

It is GOOD PRACTICE to get in the habit of using **SCRIPTS** -- where you write down and save all the instructions that R uses to complete your analysis.

Using a SCRIPT gives you a *permanent, repeatable, annotated, shareable, cross-platform archive* of your analysis.

So, here we go with our first script:

```
# ------------------------------------------------------------
# R Course Sample script. Tom Reed.
# Feb 8 2017
# Analysis of compensation.csv data
```

```r
# ---------------------------------------------------------

# First, let's clear R's brain:
rm(list=ls())

# getwd (get working directory) tells you where R is currently looking
getwd()

# setwd (set working directory) tells R where to look
setwd("C:/Users/treed/Dropbox/Teaching and supervision/Fourth Years R
course")

# use getwd to confirm that R is now looking in the right place
getwd()

# Read in the data and assign it a name
# Note that we already assigne the path above using setwd()
comp.data<- read.csv("compensation.csv")

# Check the data - confirm it is what you expected
names(comp.data) # returns the names of the columns
head(comp.data)  # returns the first 6 rows
dim(comp.data)   # returns the number of rows and columns (dimensions of the
dataset)
str(comp.data)   #  a powerful compilation of the above
```

Copy and paste this whole chunk of R code into R studio, and save the whole script in your current working folder (directory). You need to save it using the extension **.R** at the end, so that your computer will subsequently recongnise this as an R studio script file.

To run each line of code in R using R studio, simply put the cursor onto the first line (anywhere in the line), and hit **control+Enter**. This sends that line of code to the R console, which is the window below the script file.

To run multiple lines of code at once, simple select them all using your mouse (or holding down **Shift** on your keyboard and then using the **down arrow** key to selet multiple lines). Then hit **control+Enter** on your keyboard to send these multiple lines of code to the R console.

To run the whole script at once, simply hit **Control+a** on your keyboard, which selects the whole script, and then hit **control+Enter** to send the whole thing to the R console.

Back to our data file:

So in the above, we used **read.csv** to read in the data. If you search in the R help files for this function, by simply typing **?read.csv** into R, you will see that there are several other arguments that can be specified, such as whether decimal places are denoted by a dot (as we do here in Ireland/the UK), or by a comma (as they do in most of the rest of Europe!!)

Summarising your data - quick version

summary() is an R function that works on many types of objects. When you specify a data farmeas an argument to **summary()**, it will return the median, mean, inter-quartile range, minimum, maximum for all numeric columns (continuous variables), and the levels and sample size for each level of all categorical columns (factor variables).

```
summary(comp.data)
```

```
##       Root            Fruit            Grazing
##  Min.   : 4.426   Min.   : 14.73   Grazed  :20
##  1st Qu.: 6.083   1st Qu.: 41.15   Ungrazed:20
##  Median : 7.123   Median : 60.88
##  Mean   : 7.181   Mean   : 59.41
##  3rd Qu.: 8.510   3rd Qu.: 76.19
##  Max.   :10.253   Max.   :116.05
```

Missing data

Missing data in R is represented by NA. You can test whether values are NA using the is.na() function In the dataset we have just called *comp.data*, there are no missing values. Don't believe me? Have a look for yourself by simply typing *comp.data* into the R console.

Many real datasets have missing data, despite the best laid plans. Let's pretend we did have some missing data by simply creating some NAs in the *comp.data* data set. We don't want to overwrite the existing data set, so let's give it a new name:

```
# Here we simply create a new object called "comp.data2", which we tell are
is exactly the same as "comp.data""
comp.data2 <- comp.data

# Now let's replace the first value of the variable Root with a missing
value, denoted NA:
comp.data2$Root[1] <- NA
```

The above line of code involves some slightly more complicated subsetting, more on that in a minute...

To convince ourselves this worked, let's look at the first 6 rows of *comp.data*, and then the first six rows of *comp.data2*

```
head(comp.data)
```

```
##    Root Fruit  Grazing
## 1 6.225 59.77 Ungrazed
## 2 6.487 60.98 Ungrazed
## 3 4.919 14.73 Ungrazed
## 4 5.130 19.28 Ungrazed
```

```
## 5 5.417 34.25 Ungrazed
## 6 5.359 35.53 Ungrazed

head(comp.data2)

##    Root Fruit  Grazing
## 1    NA 59.77 Ungrazed
## 2 6.487 60.98 Ungrazed
## 3 4.919 14.73 Ungrazed
## 4 5.130 19.28 Ungrazed
## 5 5.417 34.25 Ungrazed
## 6 5.359 35.53 Ungrazed
```

Now, let's apply the **summary()** function to the new data frame:

```
summary(comp.data2)

##       Root            Fruit           Grazing
##  Min.   : 4.426   Min.   : 14.73   Grazed  :20
##  1st Qu.: 6.059   1st Qu.: 41.15   Ungrazed:20
##  Median : 7.181   Median : 60.88
##  Mean   : 7.206   Mean   : 59.41
##  3rd Qu.: 8.511   3rd Qu.: 76.19
##  Max.   :10.253   Max.   :116.05
##  NA's   :1
```

You can now see that the variable *Root* has 1 missing value in the data frame called *comp.data2*

NAs are treated differently from 0s in a dataset: NAs will usually cause calculations to fail, unlike 0s, but many functions can also be told to strip them beforehand.

```
mean(comp.data2$Root)                  # calculation fails due to NAs (but no
error)

## [1] NA

mean(comp.data2$Root, na.rm = TRUE)   # the na.rm argument strips NAs before
calculation

## [1] 7.205667
```

In the above piece of code, you may be thoroughly confused by the bit *comp.data2$Root* -- what the hell does that mean??

Well, this is actually quite simple, even though it looks nasty.

The dollar symbol **$** is used to subset a data frame by columns. So in the data set *comp.data2*, there are three columns, corresponding to three different variables. We can see the names (i.e. column headers) of these variables using the **names()** function:

```
names(comp.data2)
```

```
## [1] "Root"    "Fruit"   "Grazing"
```

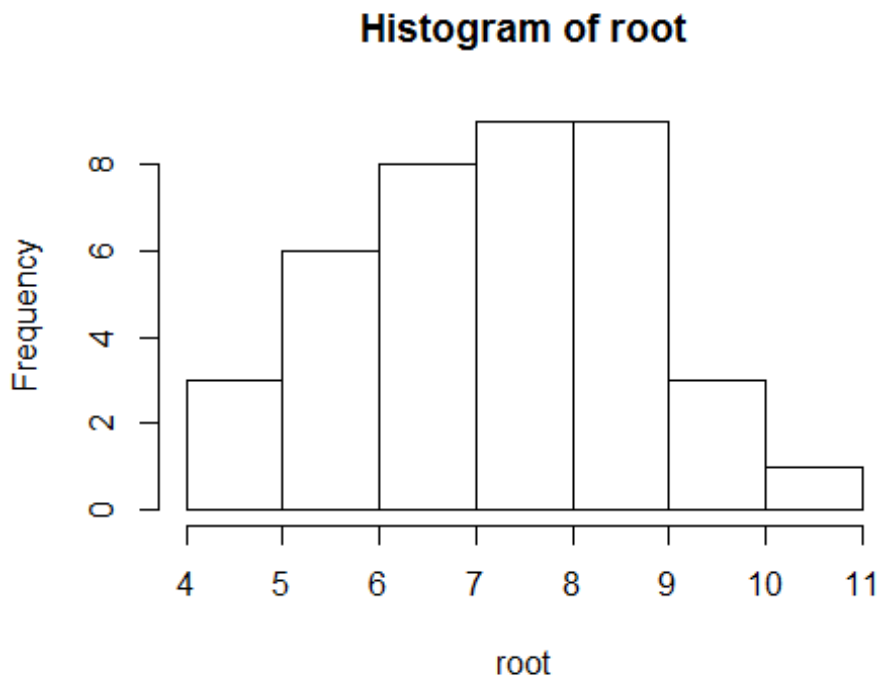
If we want to select only one column (i.e. subset the data based on one variable), we can do that as follows:

```
# Here we select only the column called Root in the data frame comp.data2,
and we give it a name (root)
root<- comp.data2$Root
root
```

```
##  [1]     NA  6.487  4.919  5.130  5.417  5.359  7.614  6.352  4.975  6.930
## [11]  6.248  5.451  6.013  5.928  6.264  7.181  7.001  4.426  7.302  5.836
## [21] 10.253  6.958  8.001  9.039  8.910  6.106  7.691  8.988  8.975  9.844
## [31]  8.508  7.354  8.643  7.916  9.351  7.066  8.158  7.382  8.515  8.530
```

We can do anything we want with this new object called root. Let's make a histogram of it using the handy, built-in function in R **hist()**

```
hist(root)
```



Notice that R has ignored the missing value in *root* when plotting this histogram

Now, we can further subset the object root by selecting only a single row within it. We do this using square brackets **[]**. Within the square brackets, we tell R what row we want to select. This is called *indexing*, and it's really important in R!!! Let's select only the first row:

```
root[1]
```

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

Now let's selected only the sixth row:

```
root[6]

## [1] 5.359
```

Now let's select rows 1 to 5:

```
root[1:5]

## [1]    NA 6.487 4.919 5.130 5.417
```

Now only rows 1, 6 and 10

```
root[c(1,6,10)]

## [1]    NA 5.359 6.930
```

Note the use of the **c()** command within the square brackets. If we didn't do that, we'd get the following error message, because we have now confused R:

```
root[1,6,10]

## Error in root[1, 6, 10]: incorrect number of dimensions
```

This brings me on to indexing objects with >1 dimension. A data frame is a 2-D object in R: there are rows going down the way, and columns going across the way.
Similarly, a matrix is a 2-D object, with rows going down the way, and columns going across the way:

```
# This generates 9 random numbers drawn from a normal distrution, and puts
them in a 3x3 matrix:
dummy<- matrix(rnorm(9), nrow=3, ncol=3)
dummy

##           [,1]      [,2]        [,3]
## [1,] -1.121868  0.990760 -0.39812242
## [2,]  1.121210 -1.723574  0.33022805
## [3,] -0.193985 -1.059968 -0.06991756

dim(dummy)

## [1] 3 3
```

To subset this matrix, we need to tell R what row we want, and what colum we want. We do this as follows:

```
# This selets the first row and the first column of dummy. Note the use of
the comma, it doesn't work without this:
dummy[1,1]

## [1] -1.121868
```

```
# This selects the second row and the second column
dummy[2,2]
```

```
## [1] -1.723574
```

```
# This selects the entire first row:
dummy[1,]
```

```
## [1] -1.1218676  0.9907600 -0.3981224
```

```
# This selects the entire third column:
dummy[,3]
```

```
## [1] -0.39812242  0.33022805 -0.06991756
```

This also works for dataframes:

```
# Here are the first 6 rows of comp.data2 again, to remind us what that data
frame looks like:
head(comp.data2)
```

```
##    Root Fruit  Grazing
## 1    NA 59.77 Ungrazed
## 2 6.487 60.98 Ungrazed
## 3 4.919 14.73 Ungrazed
## 4 5.130 19.28 Ungrazed
## 5 5.417 34.25 Ungrazed
## 6 5.359 35.53 Ungrazed
```

```
# We would get the same thing if we did this:
comp.data2[1:6,]
```

```
##    Root Fruit  Grazing
## 1    NA 59.77 Ungrazed
## 2 6.487 60.98 Ungrazed
## 3 4.919 14.73 Ungrazed
## 4 5.130 19.28 Ungrazed
## 5 5.417 34.25 Ungrazed
## 6 5.359 35.53 Ungrazed
```

```
# Here we select only the first row in the first column:
comp.data2[1,1]
```

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

```
# Remember, we set this value to be NA. If we did the same using comp.data,
we would get the original value:
comp.data[1,1]
```

```
## [1] 6.225
```

```
# Here we select the entire first column of comp.data:
comp.data[,1]
```

```
##  [1]  6.225  6.487  4.919  5.130  5.417  5.359  7.614  6.352  4.975  6.930
## [11]  6.248  5.451  6.013  5.928  6.264  7.181  7.001  4.426  7.302  5.836
## [21] 10.253  6.958  8.001  9.039  8.910  6.106  7.691  8.988  8.975  9.844
## [31]  8.508  7.354  8.643  7.916  9.351  7.066  8.158  7.382  8.515  8.530
```

```
# This gives the same result as comp.data$Root:
comp.data$Root
```

```
##  [1]  6.225  6.487  4.919  5.130  5.417  5.359  7.614  6.352  4.975  6.930
## [11]  6.248  5.451  6.013  5.928  6.264  7.181  7.001  4.426  7.302  5.836
## [21] 10.253  6.958  8.001  9.039  8.910  6.106  7.691  8.988  8.975  9.844
## [31]  8.508  7.354  8.643  7.916  9.351  7.066  8.158  7.382  8.515  8.530
```

```
# We can even ask R if these are the same thing. Note the use of the "double
equals" here, which means, "is the thing on the left exactly equal to the
thing on the right?"
comp.data[,1] == comp.data$Root
```

```
##  [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [15] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
## [29] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Don't worry if you are thoroughly confused by the above - it takes a while to get used to how all this works, but after a while, it will become second nature!!

You may also be fast realising that there are always multiple ways of doing the same thing in R.

On that note, here is another really handy way of subsetting a data frame, using the **subset()** function:

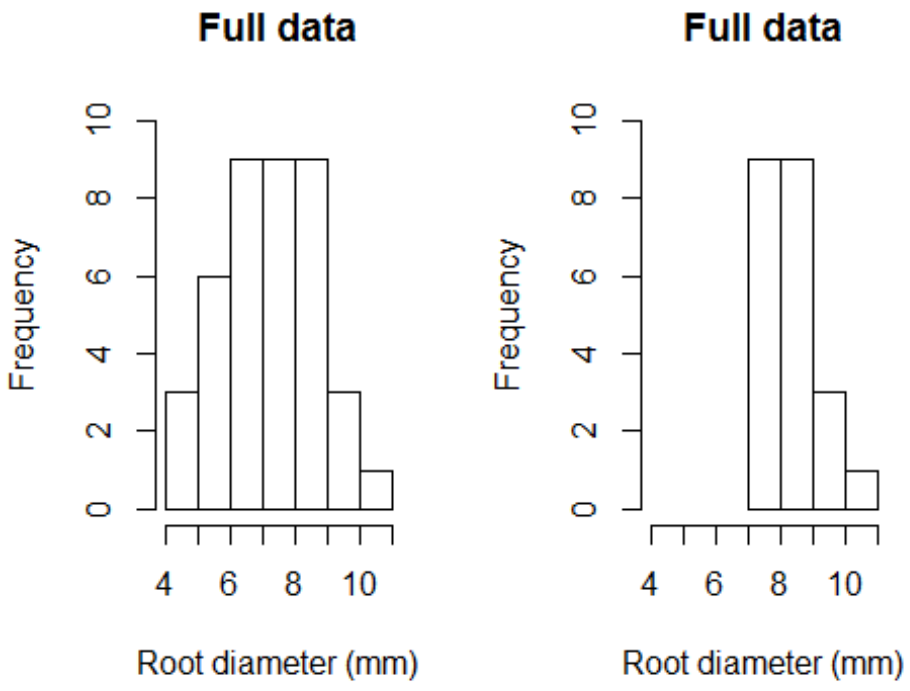
```
subset(comp.data, Root > 7)
```

```
##      Root  Fruit  Grazing
## 7   7.614  87.73 Ungrazed
## 16  7.181  73.24 Ungrazed
## 17  7.001  80.64 Ungrazed
## 19  7.302  75.49 Ungrazed
## 21 10.253 116.05   Grazed
## 23  8.001  60.77   Grazed
## 24  9.039  84.37   Grazed
## 25  8.910  70.11   Grazed
## 27  7.691  70.70   Grazed
## 28  8.988  80.31   Grazed
## 29  8.975  82.35   Grazed
## 30  9.844 105.07   Grazed
## 31  8.508  73.79   Grazed
## 32  7.354  50.08   Grazed
## 33  8.643  78.28   Grazed
## 34  7.916  41.48   Grazed
## 35  9.351  98.47   Grazed
## 36  7.066  40.15   Grazed
```

```
## 37  8.158  52.26   Grazed
## 38  7.382  46.64   Grazed
## 39  8.515  71.01   Grazed
## 40  8.530  83.03   Grazed
```

The above asks R to extract only those rows of *comp.data* where the values of the variable *Root* are greater than (denoted by **>**) 7.
Let's check that worked by plotting these subsetted data as a histogram:

```
# First let's tell R to get ready to plot two graphs, one beside the other.
par(mfrow=c(1,2))

# Now make the histogram of the full Root data:
hist(comp.data$Root, xlim=c(4,11),
     ylim=c(0,10), xlab="Root diameter (mm)",
     main="Full data")

# Now make the histogram of the subsettted Root data, which we first store in
a new object called "x":
x<- subset(comp.data, Root > 7)
hist(x$Root, xlim=c(4,11),
     ylim=c(0,10), xlab="Root diameter (mm)",
     main="Full data", breaks=4)
```



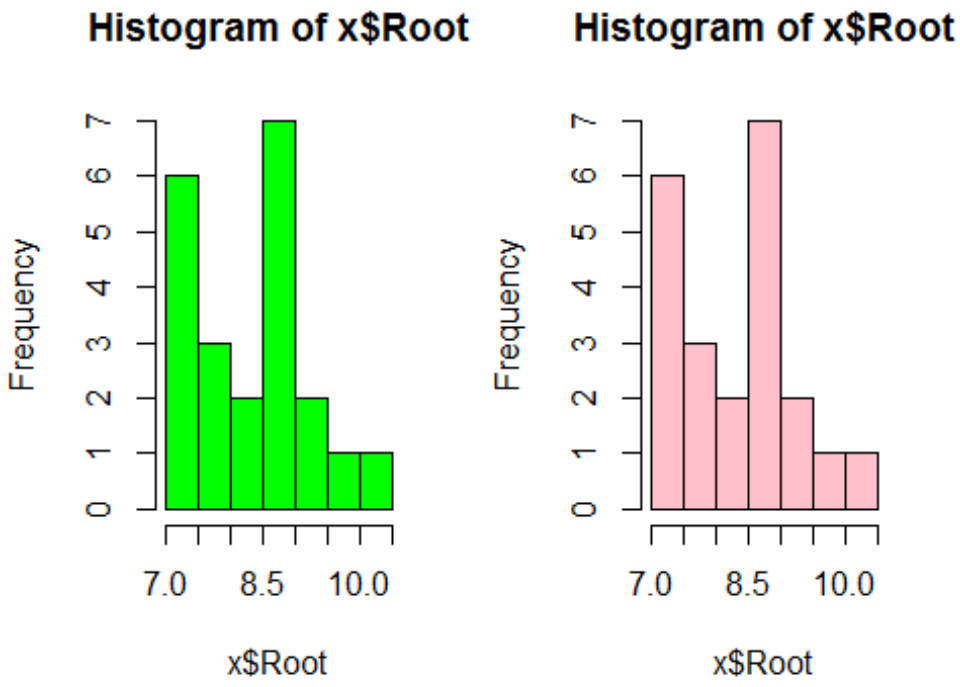
The above code for generating the histograms has got a lot more complicated! Here, we have included several additional arguments. **xlim=c(4,11)** for example tellls R to make the

X-axis run from 4 to 11 (it stands for "X axis limits"). **ylim=c(0,10)** makes the Y-axis run from 0 to 10. **main="something"** controls the main title of the histogram plot.

Note also we have used carriage returns (i.e. putting things on different lines) within the function **hist()**, just to make the R code look cleaner and easier to read. When you select the whole thing and hit **control+enter** in R studio, the whole chunk of code is sent to the console and the carriage returns are ignored by R.

There are other arguments that can be used, e.g. **col=green** would make the histogram bars green. Or use **col=pink** if that's your thing! Try it!!

```
par(mfrow=c(1,2))
hist(x$Root, col="green")
hist(x$Root, col="pink")
```



See here for a full list of possible pre-set colours in R:
http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf

We have gotten ahead of ourselves now slightly, as we have moved onto making graphs in R. We'll return to this in more detail in a moment, but first, a few more handy functions to generate custom summaries of your data

Exploring your data

A really simple way to see how many measurements you have per level of some factor is to use **table()**:

```
table(comp.data$Grazing)

##
##   Grazed Ungrazed
##       20       20
```

Note that if you just did this, R would get confused:

```
table(Grazing)

## Error in table(Grazing): object 'Grazing' not found
```

The reason is because R doesn't have an object called "Grazing" stored in it's memory, because you did not create one!! It only "knows" the data frame called *comp.data*, so you need to use the **$** symbol to tell R that the variable *Grazing* "lives" within the data frame called *comp.data*

I always use this really handy function called **with()**, which is a less clunky way of telling R to take the data frame *comp.data* and then do something with it:

```
with(comp.data,
     table(Grazing))

## Grazing
##   Grazed Ungrazed
##       20       20
```

The **aggregate** function takes, as arguments, one (or more) column of the data frame that you want to aggregate,a **list** of other columns to structure the aggregation, and a mathematical function that generates aggregated values. If you type **?aggregate** into the console, the help file will be generated by your operating system (note you need an internet connection for this).

The example below will use the function **aggregate** to generate the mean, standard deviation and sample size of *Fruit Proudction* in our *comp.data* dat set. *comp.data* has 3 columns. Here we use **aggregate** to get the mean *Fruit Production*, in each *Grazing* level:

```
aggregate(comp.data$Fruit, list(comp.data$Grazing), mean)

##    Group.1       x
## 1   Grazed 67.9405
## 2 Ungrazed 50.8805

# Or better yet, use this:
with(comp.data,  aggregate(Fruit, list(Grazing), mean))

##    Group.1       x
## 1   Grazed 67.9405
## 2 Ungrazed 50.8805
```

This says to R, FIRST get the Fruit column from *comp.data*, THEN divide it up by the *Grazing* treatment levels (Grazed and Ungrazed) and FINALLY calculate the mean Fruit

Production per Grazing level. The aggregate function returns a new data frame with the first column corresponding to the levels of *Grazing* and a second column corresponding to mean Fruit Production. Of course, if you want to use the values returned by **aggregate**, you must use the <- symbol and assign the result of aggregate to a new object. Perhaps you'll call it mean.fruit.

tapply is another super handy function thar returns the same values as **aggregate** does, using a very similar structure, but instead of returing a data frame, it return a matrix. We will see in the next section on graphing why this can be so very handy. Use **tapply** as follows:

```
tapply(comp.data$Fruit, list(comp.data$Grazing), mean)
```

```
##   Grazed Ungrazed
##  67.9405  50.8805
```

Graphing your data

We are now ready to start generating some kick-ass graphs.

Here is some solid advice: NEVER start an analysis with statistical analysis. ALWAYS start an analysis with a picture. You need to actually look at your data!!! Also, if you have done a nice experiment, or collected some nice observational data, it is highly likely that some theoretical relationship underpinened your research effort. You have in your head an EXPECTED pattern in your data

So plot your data first. Make the picture that should tell you the answer - make the axes correspond to the theory. If you can do this, AND see the pattern you EXPECTED, you are in great shape. You will possibly know the answer!!

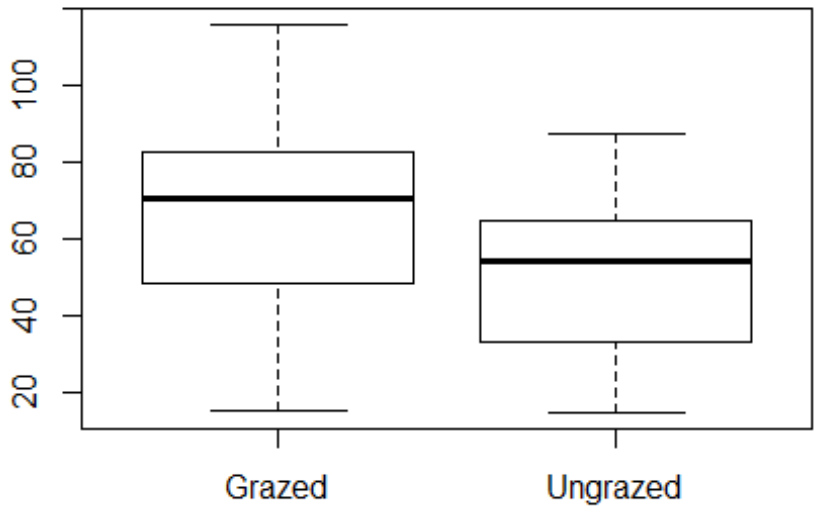
Making a picture - bar plots

When your response variable (the thing you are interested in) is continuous and your explanatory variable (the thing you suspect is related to, or explains some of the variation in, your respones variable) is categorical, the the appropriate type of graph to make would be either a box plot, or a bar plot.

Both are easy in R!!

Here is how to make a simple box-plot:

```
with(comp.data,
     boxplot(Fruit ~ Grazing))
```



Here's a nice web resource on boxplots in R: https://www.r-bloggers.com/box-plot-with-r-tutorial/

In fact, if you ever want help with something in R, just type something like "boxplots in R" into Google, and you'll be showered with useful results!

While box plots are useful for data exploration, they don't make for very nice graphs, e.g. for you thesis or for a publication, and hence many people prefer bar plots.

Making bar plots in R is easy, but it does require a few steps. We're going to make bar plots with standard error bars on them. Remember, standard error = standard deviation divided by the square root of the sample size.

First, we need to calculate the means, standard deviations and the sample sizes. We're going to use **tapply** for this, becaues the **barplot** function in R LOVES **tapply**

To keep things all in order, we'll produce a fully annotated script, which you can save as whatever name you like on your computer (e.g. "barplot_example.R")

```
# ------------------------------------------------------------
# Bar plot sample script Tom Reed.
# Feb 8 2017
# Making a bar plot using the compensation.csv data
# ------------------------------------------------------------

# First, let's clear R's brain:
rm(list=ls())
```

```r
# getwd (get working directory) tells you where R is currently looking
getwd()

# setwd (set working directory) tells R where to look
setwd("C:/Users/treed/Dropbox/Teaching and supervision/Fourth Years R
course")

# use getwd to confirm that R is now looking in the right place
getwd()

# Read in the data and assign it a name
# Note that we already assigne the path above using setwd()
comp.data<- read.csv("compensation.csv")

# Check the data - confirm it is what you expected
names(comp.data) # returns the names of the columns
head(comp.data)  # returns the first 6 rows
dim(comp.data)   # returns the number of rows and columns (dimensions of the
dataset)
str(comp.data)   #  a powerful compilation of the above

# Calculate mean of each group using tapply, which returns a matrix. As good
practice, we'll tell R to ignore any missing values (na.rm=T) if they exist.
Here there are no missing values, but there might be in other data sets!!
mean.fruit <- with(comp.data, tapply(Fruit, list(Grazing), mean, na.rm=T))

# Calculate standard deviation of each group:
sd.fruit <- with(comp.data, tapply(Fruit, list(Grazing), sd, na.rm=T))

# Calculate sample size (n) of each group:
n.fruit <- with(comp.data, tapply(Fruit, list(Grazing), length))

# Now calculate the standard errors:
se.fruit <- sd.fruit/ sqrt(n.fruit)
```

We still haven't made the barplot, but let's first just have a look at what we've produced so far:

```r
mean.fruit
```

```
##   Grazed Ungrazed
##  67.9405  50.8805
```

```r
sd.fruit
```

```
##   Grazed Ungrazed
## 24.96081 21.75897
```

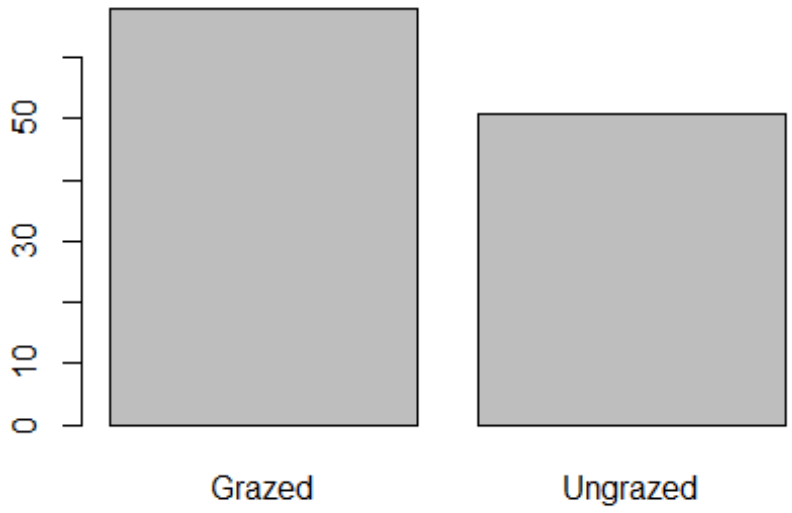
```r
n.fruit
```

```
##   Grazed Ungrazed
##       20       20

se.fruit

##   Grazed Ungrazed
## 5.581406 4.865455
```

You will notice that all of these objects have the same structure and dimensions.

Making the bar plot couldn't be easier:

```
barplot(mean.fruit)
```

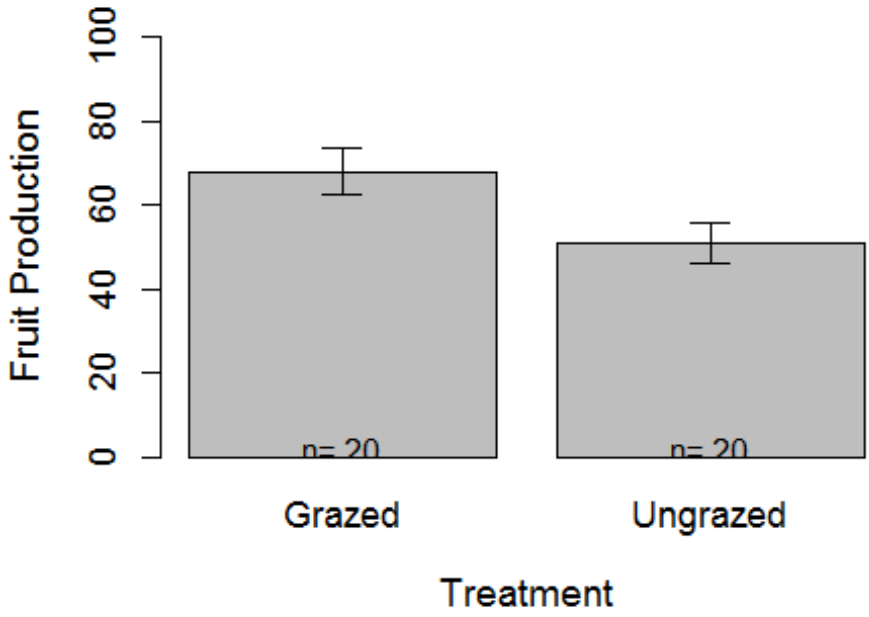


However, there are no error bars in this bar plot, and in general it could do with some nicer formatting. Take a deep breath...

Pimp my barplot

```
# First, we need to create a new object called "mids" (note: you can call it
whatever you like!), that stores the x-axis midpoints of the bars - the
location of the bars on the x-axis.
mids<- barplot(mean.fruit,
               xlab="Treatment",
               ylab="Fruit Production",
               ylim=c(0,100),
               cex.names=1.1,
               cex.lab=1.2,
               cex.axis=1.1)
```

```r
# Now, we can use the Arrows function to put error bars on the plot.  Have a
look at the arrrows function in R help to see what argumeents this take, and
have a look at example(arrows) to understand more about how this function
works
arrows(x0=mids, y0=mean.fruit - se.fruit, x1=mids, y1=mean.fruit + se.fruit,
       code = 3,angle = 90, length=0.1)

# We can even add text on the bars themselves to indicate the sample sizes,
like this:
text(x=mids, y=2, labels=paste("n=", n.fruit))
```



If you want to save this plot as a pdf (always high resolution), you can do so easily like this:

```r
# Check what folder you are currently working in. This is where the pdf will
be saved.
getwd()
# Open the PDF device - specify a file name and a paper size
pdf(file="Barplot.pdf", paper="a4")

# Do the plotting, as before:
mids<- barplot(mean.fruit,
               xlab="Treatment",
               ylab="Fruit Production",
               ylim=c(0,100),
               cex.names=1.1,
               cex.lab=1.2,
```

```
              cex.axis=1.1)

arrows(x0=mids, y0=mean.fruit - se.fruit, x1=mids, y1=mean.fruit + se.fruit,
      code = 3,angle = 90, length=0.1)

text(x=mids, y=2, labels=paste("n=", n.fruit))

# Now simply close the connection to your hard drive like this:
dev.off()

# The resulting pdf should now be in your harddrive in your working folder!
```

If you want to covert your pdf into a high quality JPEG, here is a great website for this: http://pdftojpg.me/

You can then insert that JPEG file into a Microsoft Word doc (e.g. your thesis!), and maintain the high resolution. Unfortunately Word does not allow you to insert PDFs directly into your word doc, at least not without totally blurring the image.

Making a picture - scatter plots

When you want to plot one continuous variable against another continuous variable, this would be a scatter plot. These are easy in R too.

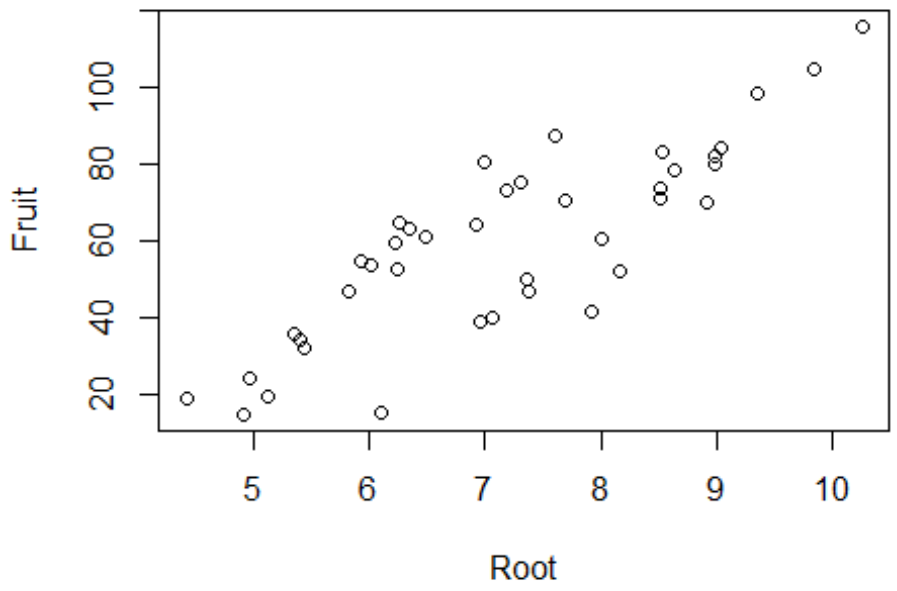
The syntax is simply this:

```
plot(y ~ x, data = dataset)
```

This says "plot the y variables as a function of the x-variable, where these variables live in the dataset called blah"

We'll use the *comp.data* data set again as an example, and we'll plot *Fruit* against *Root*:

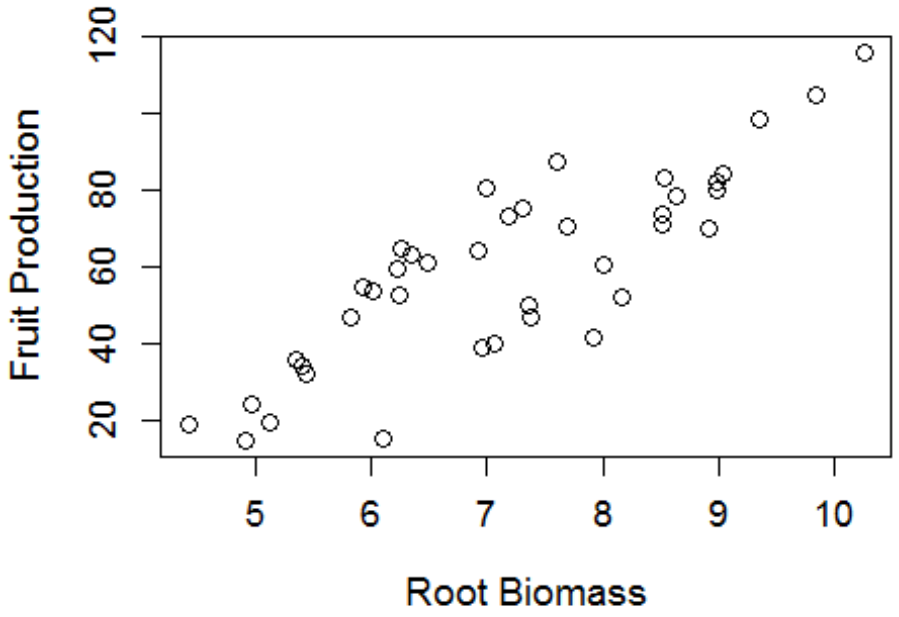
```
plot(Fruit ~ Root, data=comp.data)
```



That actually looks pretty good already, but we can make it nicer:

Pimp my scatterplot: axis labels

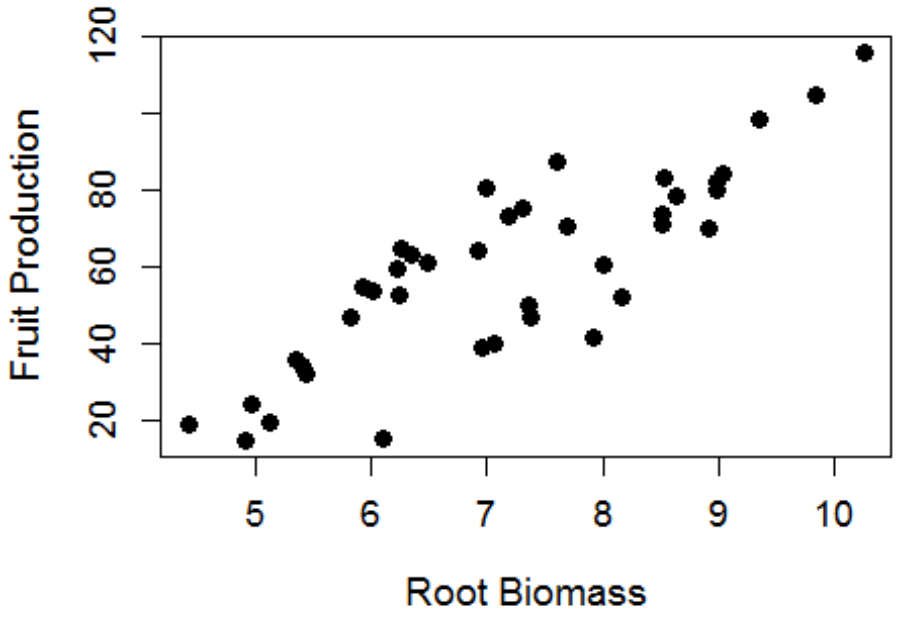
```
plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1.2,
     cex.axis=1.1,
     cex= 1.2)
```



The **cex=1.2** bit tells R to make the data points (here circles) 1.2 times bigger than normal

We can change these to another type of symbol if we like, e.g. filled black circles, using pch=16

```
plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1.2,
     cex.axis=1.1,
     cex= 1.2,
     pch=16)
```



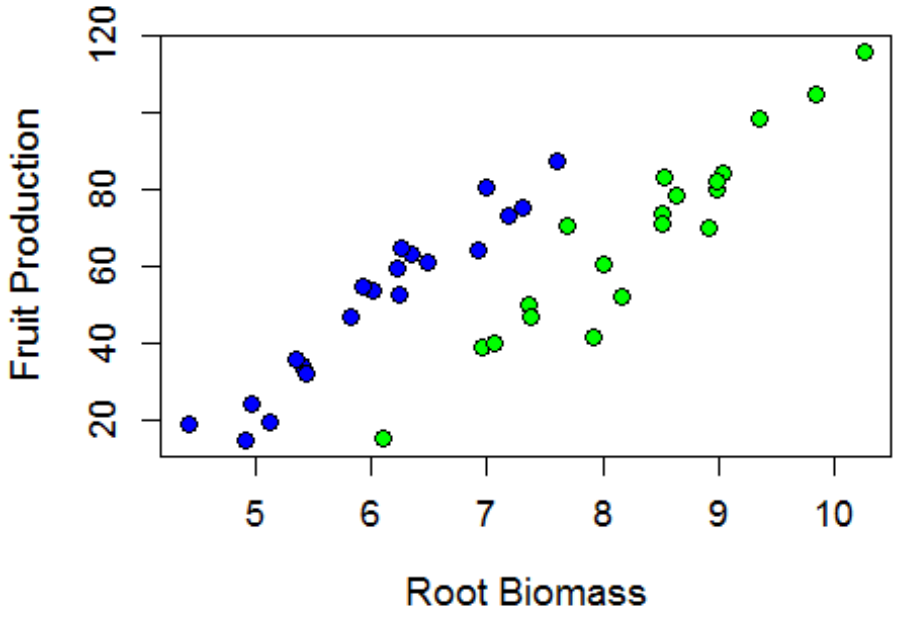
Play around with pch to discover other types of symbols. For more info on this, see here: http://www.statmethods.net/advgraphs/parameters.html

If you look closely at this scatter-plot, you will notice two groups of data. These actually correspond to the two levels of the variable *Grazing*: Grazed versus Ungrazed.

Let's colour in these differently. There are at least 2 different ways of doing this:

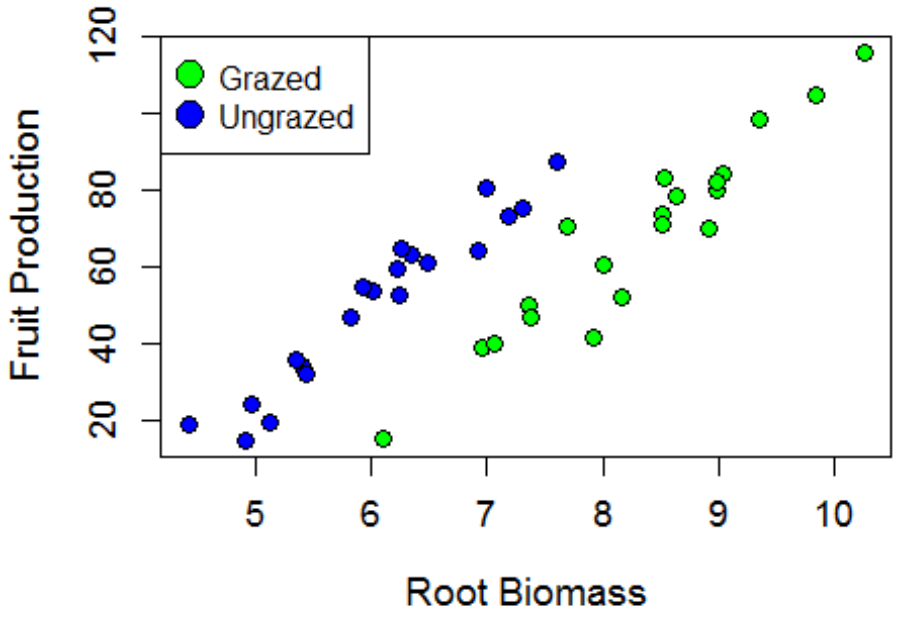
```
culr<- ifelse(comp.data$Grazing == "Grazed", "green", "blue")

plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1.2,
     cex.axis=1.1,
     cex= 1.2,
     pch=21,
     bg=culr)

# Or we could do it this way:

culr <- c("green", "blue")

plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1.2,
```

```
     cex.axis=1.1,
     cex= 1.2,
     pch=21,
     bg=culr[Grazing])
```



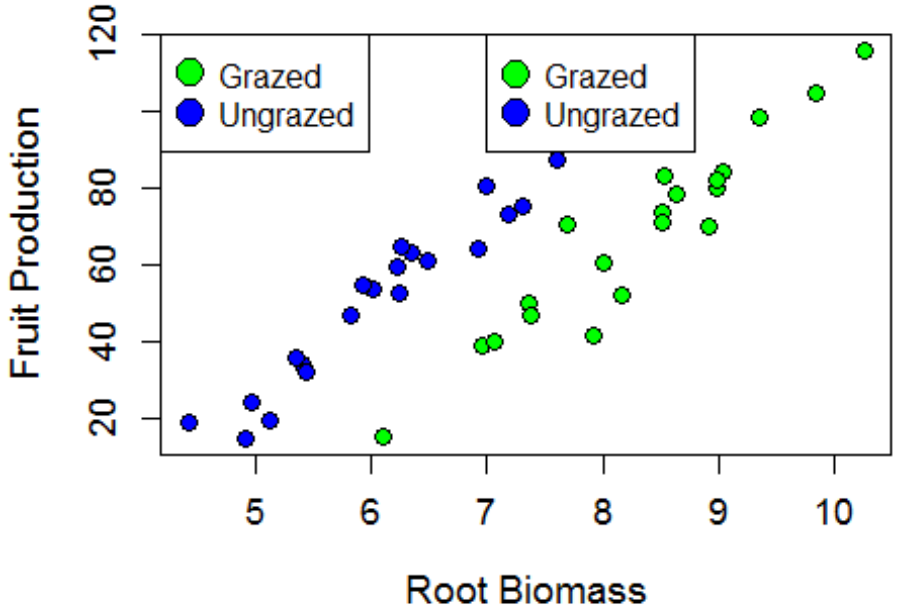
We can also add a legend to the graph. Type in ?legend into the console to learn more about how this works

```
legend("topleft", legend = c("Grazed", "Ungrazed"),
       pch=21, pt.bg = c("Green", "Blue"), pt.cex=2)
```



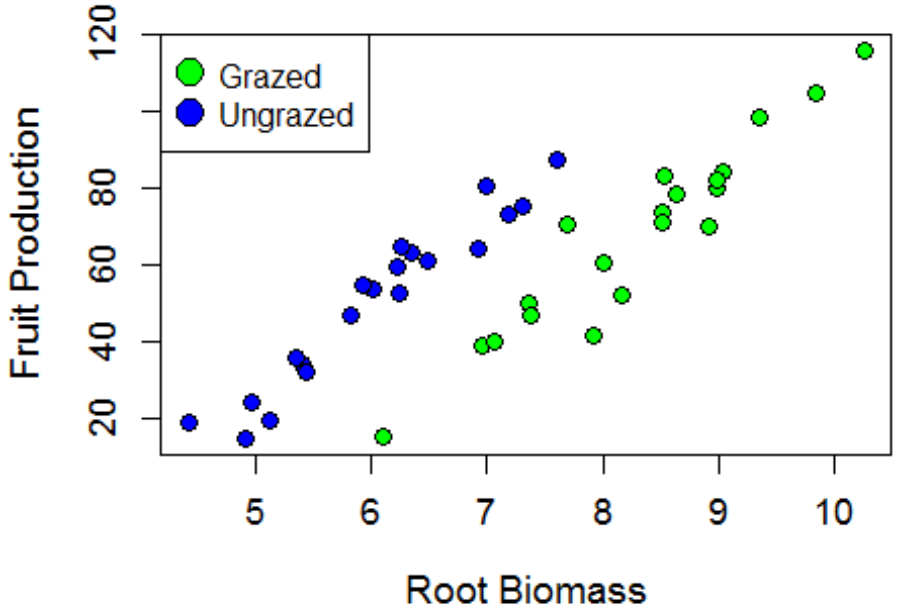
OR we can specify where to put the legend in the plot, using the coordinates of the 2 axis (X axis first, then Y axis):

```
legend(x=7, y=120, legend = c("Grazed", "Ungrazed"),
       pch=21, pt.bg = c("Green", "Blue"), pt.cex=2)
```



You now have 2 legends in this plot, which you don't want! To just keep one, you simply edit your script so that you first call the function **plot**, and then you call the function **legend** (using whichever format you prefer):

```
culr <- c("green", "blue")

plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1.2,
     cex.axis=1.1,
     cex= 1.2,
     pch=21,
     bg=culr[Grazing])

legend("topleft", legend = c("Grazed", "Ungrazed"),
       pch=21, pt.bg = c("Green", "Blue"), pt.cex=2)
```



Now let's say we wanted to save BOTH the barplot we made above AND the scatterplot to a pdf file. Easy!

```
# Check what folder you are currently working in. This is where the pdf will
be saved.
getwd()

# Open the PDF device - specify a file name and a paper size
pdf(file="Barplot and scatterplot.pdf", paper="a4")

# NB***:  Here set the layout of the plotting
# In this case, we specify a 1 x 2 plot and place the figures side by side:
The "oma="" argument controls the amount of blank space in the outer margins
of the graphing space, in ther order bottom, left, top, right. Play around
with it to understand it!!
par(mfrow=c(1,2), oma=c(5,0,5,0))


# Do the plotting, first the bar plot from before:
mids<- barplot(mean.fruit,
               xlab="Treatment",
               ylab="Fruit Production",
               ylim=c(0,100),
               cex.names=1,
               cex.lab=1,
               cex.axis=1)
```

```
arrows(x0=mids, y0=mean.fruit - se.fruit, x1=mids, y1=mean.fruit + se.fruit,
      code = 3,angle = 90, length=0.1)

text(x=mids, y=5, labels=paste("n=", n.fruit))


# And now the second plot, here the scatter plot we've already made:
culr <- c("green", "blue")

plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1,
     cex.axis=1,
     cex= 1.1,
     pch=21,
     bg=culr[Grazing])

legend("topleft", legend = c("Grazed", "Ungrazed"),
       pch=21, pt.bg = c("Green", "Blue"), pt.cex=1.5)

# Now simply close the connection to your hard drive like this:
dev.off()

# The resulting pdf should now be in your harddrive in your working folder!
```

Ok that's enough on plotting for one day. But the take-home messages:

- The sky is the limit with R!
- Build your scripts sequentially, and annotate them nicely with comments
- Use the help function in R to understand what things to, or Google it!
- Keep looking at what you have created, by simply typing the name of the object into the console. That way you can keep track of any mistakes
- Play around with the functions and the arguments they take - it's the only way to learn more!!

Doing basic statistics in R

If you can think of a statistical test, R can do it! It just takes a while to learn the syntax of how to actually run these yourself. Here we will only take a very cursory look at a few common ones:

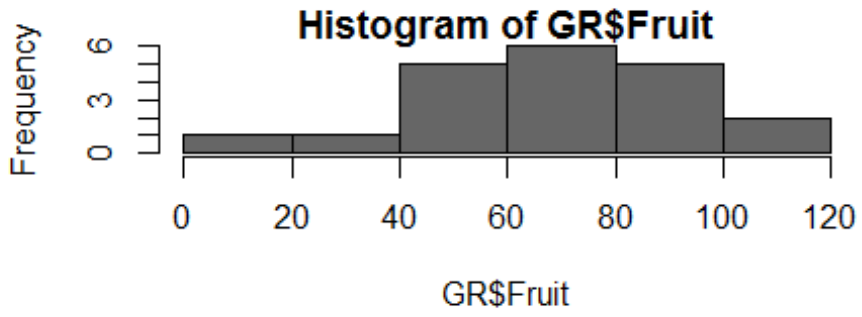
T tests.

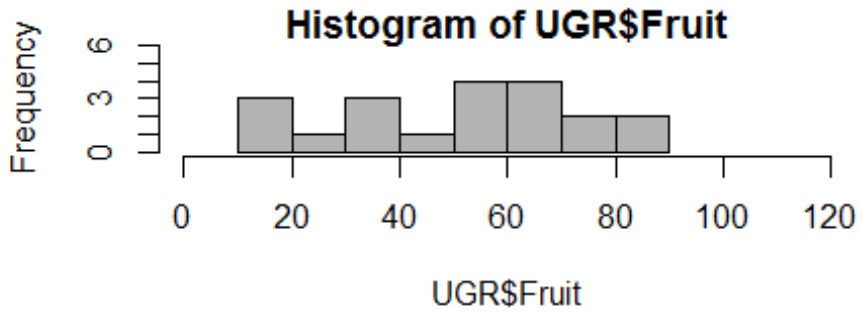
The two sample t-test is a comparison of the means of two groups. It is appropriate when sample sizes in each group are small, but it does assume that the data being analysed are normally distributed and their variances are similar.

Let's use the *comp.data* data set to compare the mean Fruit Production in Grazed versus Ungrazed treatments:

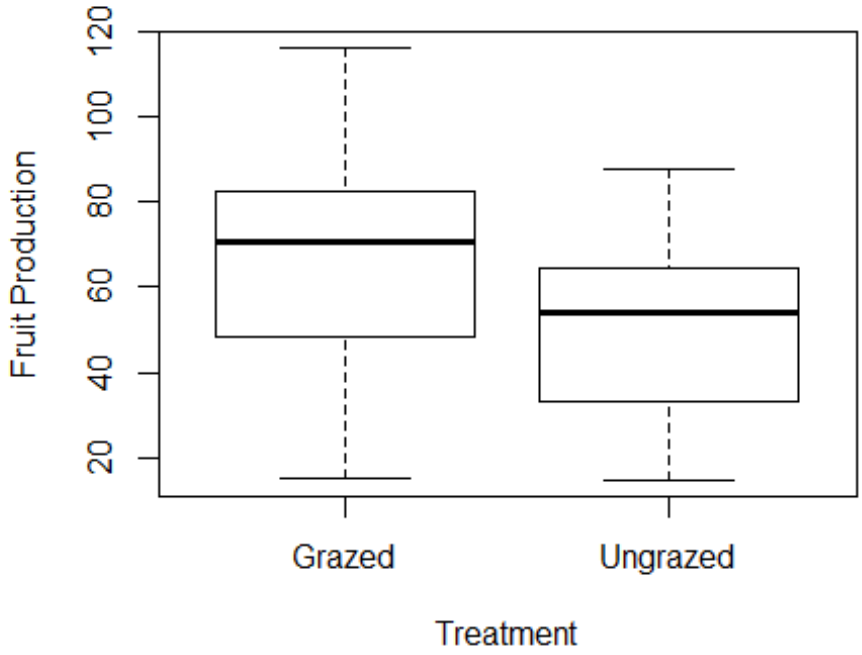
First, as always, let's plot the data:

```
# Subset the data:

GR <- subset(comp.data, Grazing=="Grazed")
UGR <- subset(comp.data, Grazing=="Ungrazed")

# Summarise these sub datasets:
summary(GR)

##       Root            Fruit            Grazing
##  Min.   : 6.106   Min.   : 14.95   Grazed  :20
##  1st Qu.: 7.614   1st Qu.: 49.22   Ungrazed: 0
##  Median : 8.511   Median : 70.86
##  Mean   : 8.309   Mean   : 67.94
##  3rd Qu.: 8.978   3rd Qu.: 82.52
##  Max.   :10.253   Max.   :116.05

summary(UGR)

##       Root           Fruit           Grazing
##  Min.   :4.426   Min.   :14.73   Grazed  : 0
##  1st Qu.:5.402   1st Qu.:33.77   Ungrazed:20
##  Median :6.119   Median :54.23
##  Mean   :6.053   Mean   :50.88
##  3rd Qu.:6.598   3rd Qu.:64.46
##  Max.   :7.614   Max.   :87.73

# Make a sinle figure wit h2 plots, one below the other:
par(mfrow=c(2,1), oma=c(1,1,1,1), mar=c(5,4,1,1))
hist(GR$Fruit, xlim=c(0,120), ylim=c(0,6), col="grey40")
hist(UGR$Fruit, xlim=c(0,120), ylim=c(0,6), col="grey70")
```






```r
# Or just make a boxplot:
par(mfrow=c(1,1), oma=c(1,1,1,1), mar=c(5,4,1,1))
with(comp.data, boxplot(Fruit ~ Grazing, ylab="Fruit Production",
xlab="Treatment"))
```




```
# Now do your t-test!

t.test(GR$Fruit, UGR$Fruit)

##
##  Welch Two Sample t-test
##
## data:  GR$Fruit and UGR$Fruit
## t = 2.304, df = 37.306, p-value = 0.02689
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##   2.061464 32.058536
## sample estimates:
## mean of x mean of y
##   67.9405   50.8805

# To make sure that the above t-test was valid, let's check for "homogeneity
of variances", i.e. that the spread of data in each group was not
significantly different:

bartlett.test(comp.data$Fruit, comp.data$Grazing)

##
##  Bartlett test of homogeneity of variances
##
## data:  comp.data$Fruit and comp.data$Grazing
## Bartlett's K-squared = 0.3478, df = 1, p-value = 0.5554
```

Chi-squared test

Let's say we collected data on the colour of peppered moths in industrial verus rural areas, that looked like this:

```
numbers<- c(115, 30, 85, 70)
env<- c("Industrial", "Rural", "Industrial", "Rural")
col <- c("Black", "Black", "White", "White")

moths<- data.frame(numbers,env,col)

with(moths, table(env, col))*numbers
```

```
##             col
## env          Black White
##   Industrial   115    85
##   Rural         30    70
```

We then want to know if there are more black moths in industrial areas, and white moths in rural areas, than we would expect by chance. This is where you need to use a chi-squared test!

```
# First, we have to get the data into the right format (here a matrix format)
moth.data<- matrix(c(115, 30, 85, 70), 2, 2, byrow=TRUE)

# Then one more line of code is all we need:
chisq.test(moth.data)
```

```
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  moth.data
## X-squared = 19.103, df = 1, p-value = 1.239e-05
```

Accoring to the above, there is a very small probability that the pattern we foiund arose by chance; in other words, we have strong evidence for a non-randon association between moth colour and habitat (industrial versus rural)

Here's how you'd retrive the expected values:

```
my.chi <- chisq.test(moth.data)
names(my.chi)
```

```
## [1] "statistic" "parameter" "p.value"   "method"    "data.name" "observed"
## [7] "expected"  "residuals" "stdres"
```

```
my.chi$expected
```

```
##           [,1]     [,2]
## [1,]  96.66667 48.33333
## [2,] 103.33333 51.66667
```

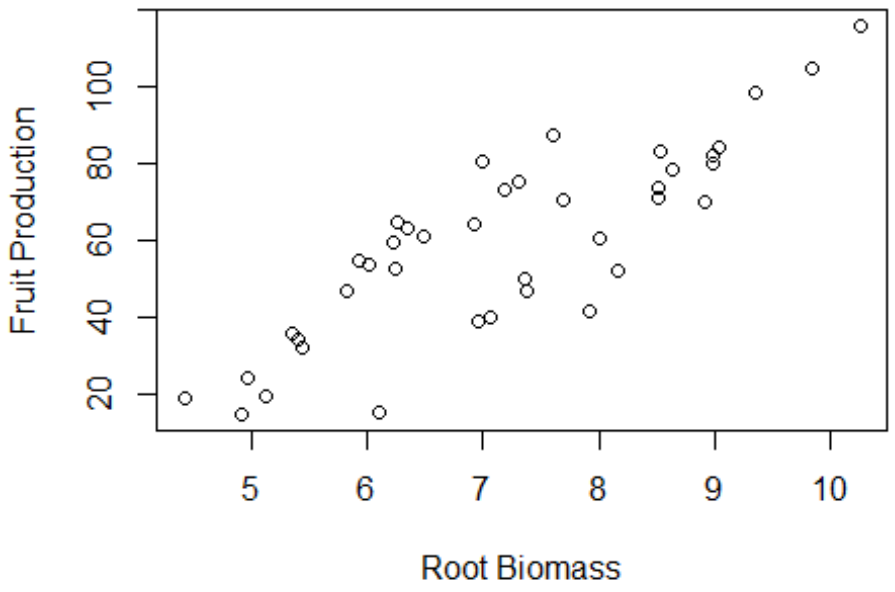
Correlation

This is a simple test of whether two continuous variables (in the case of the Pearson product-moment correlation) are associated with each other statistically.

Easy-peasy in R.

Let's use the Fruit Production and Root data in *comp.data*, to see if they are significantly correlated:

```
# Here's the graph again:
plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production")
```




```
# And here's thee correlation test:
with(comp.data, cor.test(x=Root, y=Fruit))

##
##  Pearson's product-moment correlation
##
## data:  Root and Fruit
## t = 9.5835, df = 38, p-value = 1.099e-11
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7175066 0.9132757
## sample estimates:
```

```
##      cor
## 0.841036
```

They sure are! Which is kind of obvious from the graph.

Regression

Related to correlation is regression, but with regression there is a definite response variable and a definite explanatory variable (both continuous)

With correlation, it doesn't matter which way round you specify the variables:

```
with(comp.data, cor.test(x=Root, y=Fruit))

##
##  Pearson's product-moment correlation
##
## data:  Root and Fruit
## t = 9.5835, df = 38, p-value = 1.099e-11
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7175066 0.9132757
## sample estimates:
##      cor
## 0.841036

with(comp.data, cor.test(x=Fruit, y=Root))

##
##  Pearson's product-moment correlation
##
## data:  Fruit and Root
## t = 9.5835, df = 38, p-value = 1.099e-11
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.7175066 0.9132757
## sample estimates:
##      cor
## 0.841036
```

But it does with regression. Here's how you do it in R, using the *lm* function (which stands for linear model:

```
reg.model<- lm(Fruit ~ Root, data=comp.data)

# We just stored the results in a new object called reg.model, now let's see
a summary of the actual results:
summary(reg.model)

##
## Call:
```

```
## lm(formula = Fruit ~ Root, data = comp.data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -29.3844 -10.4447  -0.7574  10.7606  23.7556
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -41.286     10.723  -3.850 0.000439 ***
## Root          14.022      1.463   9.584  1.1e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 13.52 on 38 degrees of freedom
## Multiple R-squared:  0.7073, Adjusted R-squared:  0.6996
## F-statistic: 91.84 on 1 and 38 DF,  p-value: 1.099e-11
```

```
# Or you can see the output as an ANOVA table:
anova(reg.model)
```

```
## Analysis of Variance Table
##
## Response: Fruit
##           Df  Sum Sq Mean Sq F value    Pr(>F)
## Root       1 16795.0 16795.0  91.844 1.099e-11 ***
## Residuals 38  6948.8   182.9
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Again, highly significant, as expected. Note that you get a different result if you do it the other way around, i.e. test for an effect on Fruit (x-axis) on Root (y-axis):

reg.model<- lm(Fruit ~ Root, data=comp.data)

We just stored the results in a new object called reg.model, now let's see a summary of the actual results:

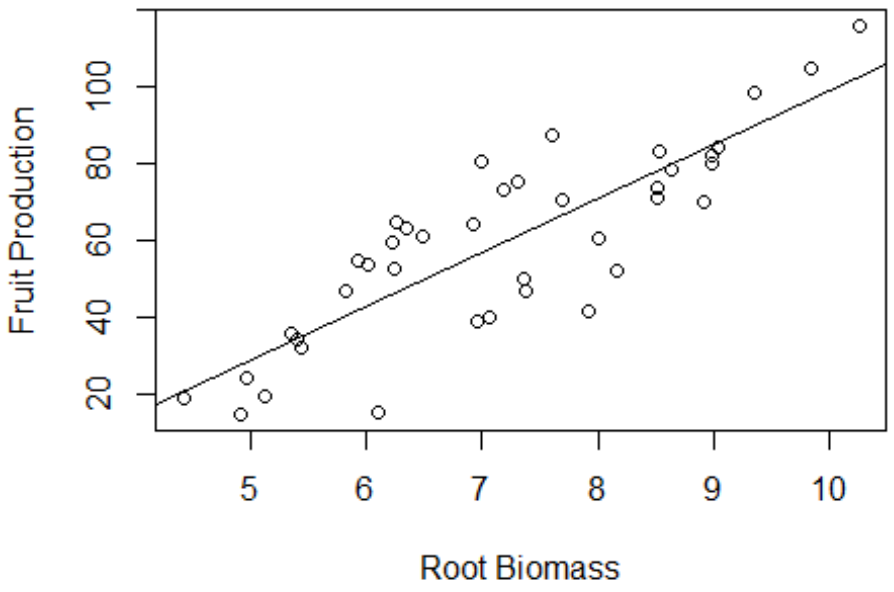
```
summary(lm(Root ~ Fruit, data=comp.data))
```

```
##
## Call:
## lm(formula = Root ~ Fruit, data = comp.data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.25105 -0.69970 -0.01755  0.66982  1.63933
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 4.184256   0.337987  12.380  6.6e-15 ***
```

```
## Fruit       0.050444   0.005264   9.584  1.1e-11 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8111 on 38 degrees of freedom
## Multiple R-squared:  0.7073, Adjusted R-squared:  0.6996
## F-statistic: 91.84 on 1 and 38 DF,  p-value: 1.099e-11
```

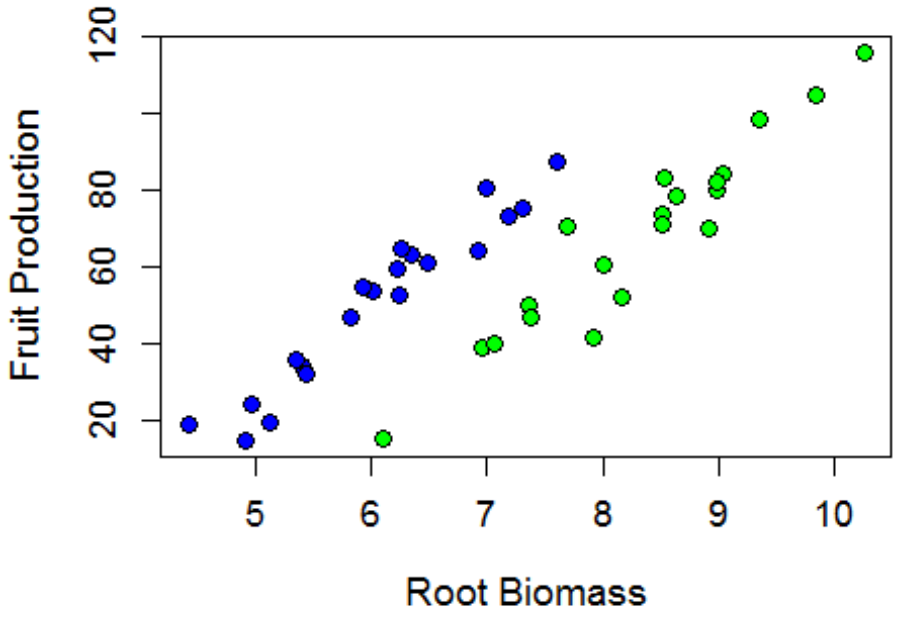
It is really easy to tell R to plot the resulting linear regression best-fit line through the data (going back to having *Root* on the x-axis):

```
# Here's the graph again:
plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production")

reg.model<- lm(Fruit ~ Root, data=comp.data)
abline(reg.model)
```



General linear models

You will recall that there are actually different patterns going on here for Grazed versus Ungrazed treatments:



In this case, we might want to test for the effets of BOTH *Root* and *Grazing* in the same model. This is called a general linear model, and one can include multiple explanatory variables, which can be continuous or categorical.

This also couldn't be simpler in R:

```
# We simply include both Root and Grazing as separate terms on the right-hand
side of the ~ in the lm() function, seperated by a + sign:
gen.lin.model<- lm(Fruit ~ Root + Grazing, data=comp.data)

# We just stored the results in a new object called gen.lin.model, now let's
see a summary of the actual results:
summary(gen.lin.model)

##
## Call:
## lm(formula = Fruit ~ Root + Grazing, data = comp.data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -17.1920  -2.8224   0.3223   3.9144  17.3290
##
## Coefficients:
##                 Estimate Std. Error t value Pr(>|t|)
## (Intercept)     -127.829      9.664  -13.23 1.35e-15 ***
## Root              23.560      1.149   20.51  < 2e-16 ***
## GrazingUngrazed   36.103      3.357   10.75 6.11e-13 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.747 on 37 degrees of freedom
## Multiple R-squared:  0.9291, Adjusted R-squared:  0.9252
## F-statistic: 242.3 on 2 and 37 DF,  p-value: < 2.2e-16

# Or you can see the output as an ANOVA table:
anova(gen.lin.model)

## Analysis of Variance Table
##
## Response: Fruit
##           Df  Sum Sq Mean Sq F value    Pr(>F)
## Root       1 16795.0 16795.0  368.91 < 2.2e-16 ***
## Grazing    1  5264.4  5264.4  115.63 6.107e-13 ***
## Residuals 37  1684.5    45.5
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

So both *Root* and *Grazing* are highly significant. What about an interaction - i.e. is the linear effect of Root Biomass on Fruit Production different in Grazed versus Ungrazed treatments?

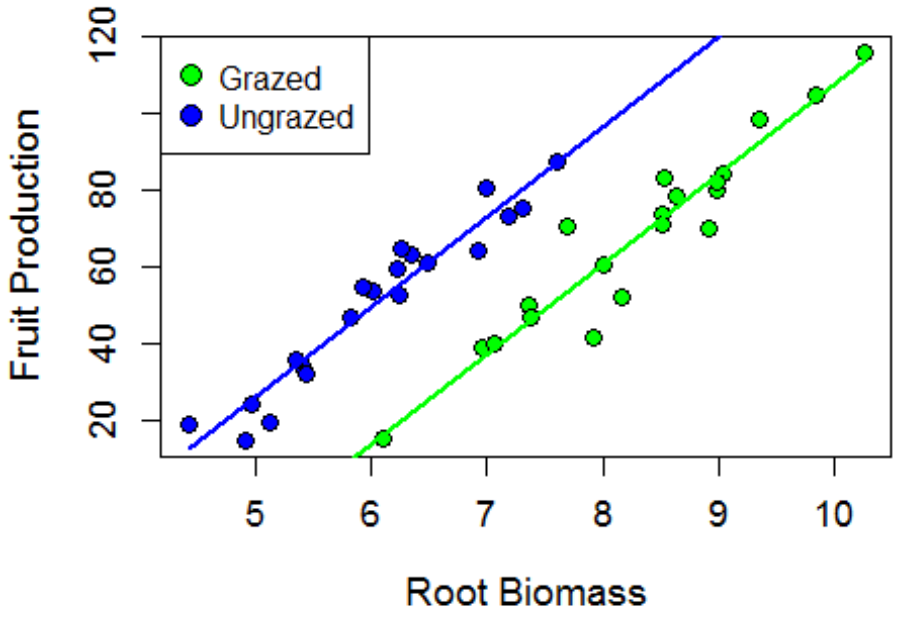
```
# To test for an interaction between Root and Grazing, we simply specify the
model like this, using the * symbol between the terms, which stands for
"please kindly test for an interaction there R, like a good lad"
gen.lin.model2<- lm(Fruit ~ Root * Grazing, data=comp.data)

# We just stored the results in a new object called gen.lin.2, now let's see
a summary of the actual results:
summary(gen.lin.model2)

##
## Call:
## lm(formula = Fruit ~ Root * Grazing, data = comp.data)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -17.3177  -2.8320   0.1247   3.8511  17.1313
##
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)          -125.173     12.811  -9.771 1.15e-11 ***
## Root                   23.240      1.531  15.182  < 2e-16 ***
## GrazingUngrazed        30.806     16.842   1.829   0.0757 .
## Root:GrazingUngrazed    0.756      2.354   0.321   0.7500
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.831 on 36 degrees of freedom
```

```
## Multiple R-squared:  0.9293, Adjusted R-squared:  0.9234
## F-statistic: 157.6 on 3 and 36 DF,  p-value: < 2.2e-16

# Or you can see the output as an ANOVA table:
anova(gen.lin.model2)

## Analysis of Variance Table
##
## Response: Fruit
##              Df  Sum Sq Mean Sq  F value    Pr(>F)
## Root          1 16795.0 16795.0 359.9681 < 2.2e-16 ***
## Grazing       1  5264.4  5264.4 112.8316 1.209e-12 ***
## Root:Grazing  1     4.8     4.8   0.1031      0.75
## Residuals    36  1679.6    46.7
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

So it turns out that there is no interaction here, that is, the effect of Root Biomass on Fruit Production is the same in both treatments.

We can show that graphically by plotting seperate lines through the data points for each treatment level:

```
culr<- ifelse(comp.data$Grazing == "Grazed", "green", "blue")

plot(Fruit ~ Root, data=comp.data,
     xlab="Root Biomass",
     ylab="Fruit Production",
     cex.lab=1.2,
     cex.axis=1.1,
     cex= 1.2,
     pch=21,
     bg=culr)

legend("topleft", legend = c("Grazed", "Ungrazed"),
       pch=21, pt.bg = c("Green", "Blue"), pt.cex=1.5)

xv<- seq(min(comp.data$Root), max(comp.data$Root), length.out=2)
GR<- coef(gen.lin.model)[1] + coef(gen.lin.model)[2]*xv
UGR<- coef(gen.lin.model)[1] + coef(gen.lin.model)[2]*xv +
coef(gen.lin.model)[3]

lines(xv, GR, col="green", lwd=2)
lines(xv, UGR, col="blue", lwd=2)
```



That's all folks, remember, practice practice practice with R, it's the only way!!!