R: an introduction to data management and analysis

Chris Beirne

2020-01-04

Contents

Chapter 1

Preface

R has changed the way we manage and analyse data. It is fast, open source and free. You can analyse data in R, produce maps, create art… I even wrote, formatted and printed this handout in R! Hopefully the next few days will be the start of your relationship with R. You'll have to work at it, you will definately go through some rough patches, but in the end I promise it will be worth it.

The information included in this book is based on my own experience and understanding of what you will require when taking your first steps into the world of R. I have tried to make it as simple and as accessible as possible. It will ideally serve as a primer for you to build on in future months and years. Where appropriate, I point you to additional resources and learning material. This booklet is by no means exhaustive, but it will, I hope, be useful. Remember, learning R is like learning a language - if you don't use it you will lose it. Keep practicing.

It is also important to note that **this is not a comprehensive R manual or an introductory statistics course**. If you are looking for those I recommend downloading and working through the following:

The R Book - Micheal Crawley

Statistics: an introduction using R - Micheal Crawley

Analysis of Biological Data - Whitlock and Schluter:

And check out this website for aweseome (free) resources: https://r-dir.com/learn/e-books.html

Want to learn R like a pirate? If the answer is YaRrr go to https://bookdown.org/ndphillips/YaRrr

Good luck, enjoy yourself, and most importantly - **if I ever see or hear of you making a graph in excel again there will be trouble!**

Chapter 2

Setup

Before we get into coding in R, we need to get a few things set up.

2.1 Notation

In the following pages you will meet several font conventions:

- Normal text white background = my comments and notes
- Grey box text = data to be entered into R by run (note #'s make text invisidble to R, but visible to you… more on that later)

```
# Print Text
print("Hello world!")
```

- White text in 'Courier new' font preceeded by the ## signs = R output

```
## [1] "Hello world!"
```

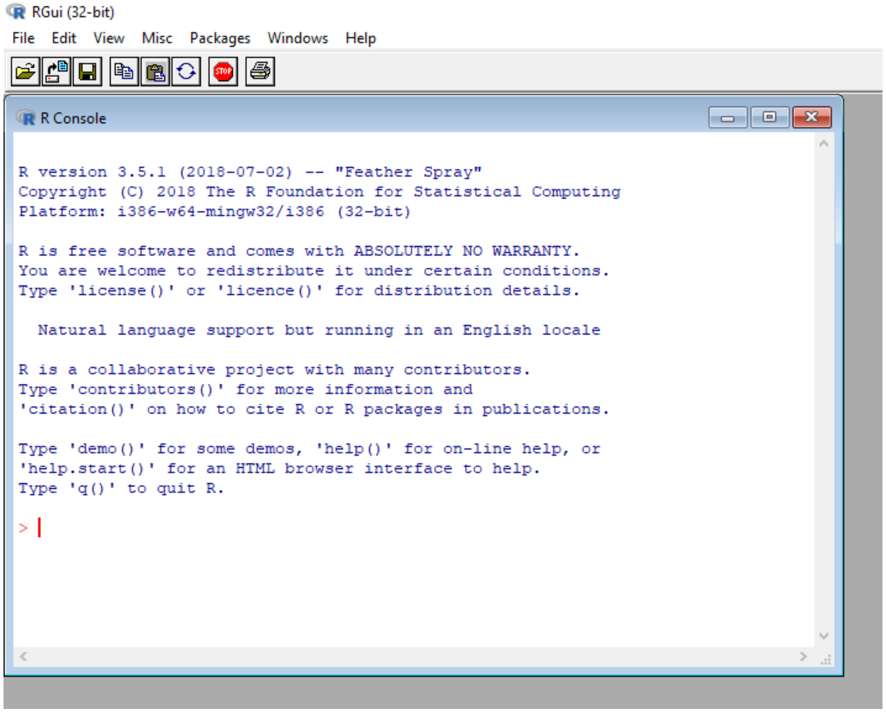
- ***Bold italic text*** = Tasks / exercises for you to complete

2.2 Download and open R

Before we get into using R studio, it is useful to see what 'base R' looks like. Think of R Studio as the fancy packaging, but R as the machine itself.

- Download and R from https://cran.cnr.berkeley.edu/
- Open R

The most important part to appreciate is the 'R console' (see below) - this is the interface between you and the inner workings of R. Everything you do in R goes through this console: your input commands, output and error messages are displayed here. I have increasingly seen people new to R ignoring the console and even having it minimized… This is not a good idea!



```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: i386-w64-mingw32/i386 (32-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

It used to be that case that all code was directly cut and pasted into the R console, after the > symbol… try it out!

```
# Send through a simple command
4+4
```

```
## [1] 8
```

However, we now have ‘R Studio’ to make our lives considerably easier!

2.3 Download and open R Studio

Head to https://www.rstudio.com/products/rstudio/download/ and download the free ‘R studio desktop edition’, install it and run it.

R Studio looks more complicated than the basic R GUI, but all of these elements are there to make your life easier. Do not worry about what these things mean, we will address each in turn below.

2.4 Setting up R Studio to be productive

We will now set up our R workspace so that we can start coding! There are lots of different work-flows in R, some better than others. I am not going to show you everyway of doing things… just my way!

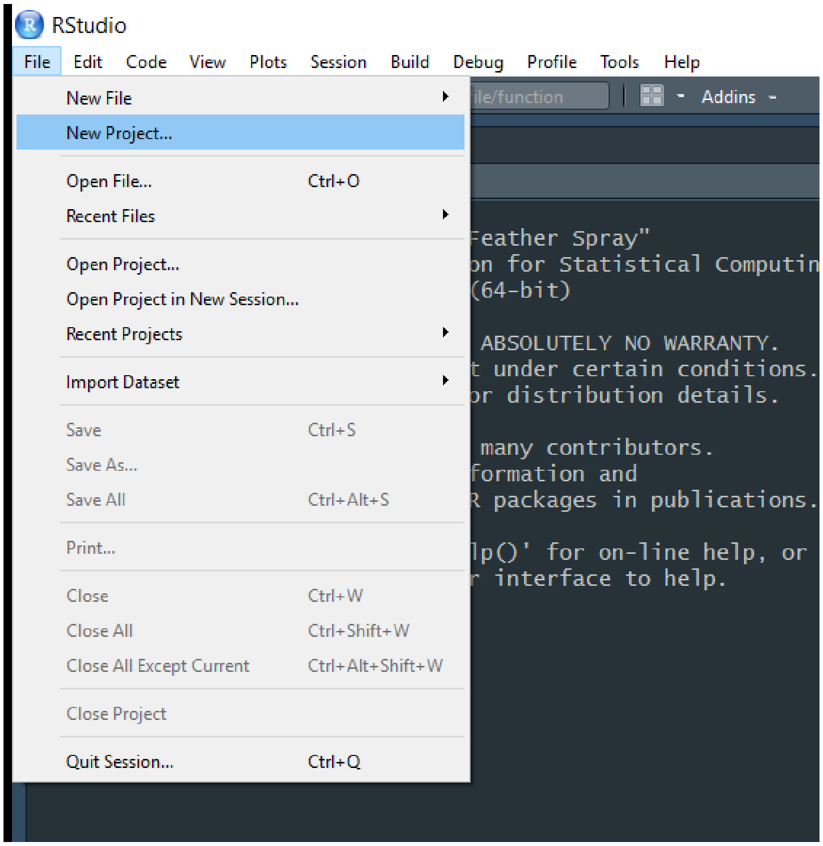
2.4.1 R Projects

I can work on multiple different projects simultaneously as I use ‘projects’ in R studio. Think of a project file as a box into which you put everything you need to perform analysis for given a paper or report.

'Projects' are useful as you can have multiple completely independent projects open at the same time - the data and workspaces are not shared.

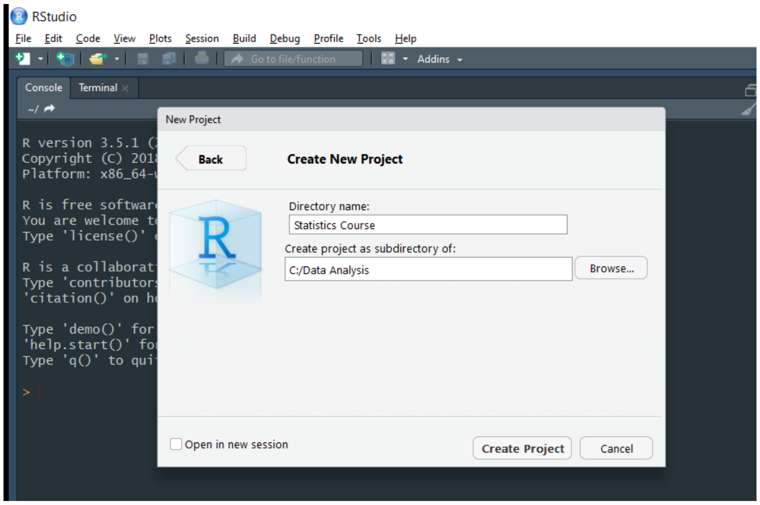
First, make a folder called "Data Analysis" inside your normal work folder.

Click on 'File' then 'New Project'



```
Feather Spray"
on for Statistical Computin
(64-bit)

 ABSOLUTELY NO WARRANTY.
t under certain conditions.
or distribution details.

 many contributors.
formation and
R packages in publications.

lp()' for on-line help, or
r interface to help.
```

Click on 'New directory' then click on 'New Project'. Call your new project something original like "Statistics course" then point the subdirectory to your newly created "Data Anlysis" folder.



```
R version 3.5.1 (
Copyright (C) 201
Platform: x86_64-

R is free softwar
You are welcome t
Type 'license()'

R is a collaborat
Type 'contributor
'citation()' on h

Type 'demo()' for
'help.start()' fo
Type 'q()' to qui

>
```

Directory name: Statistics Course

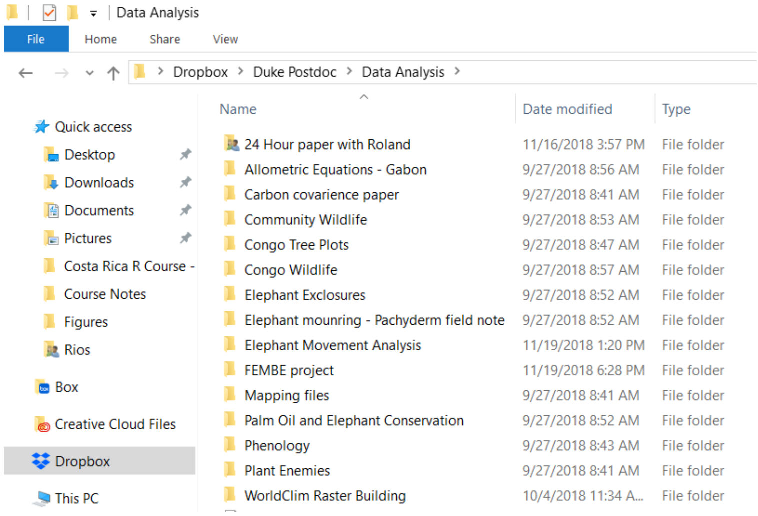
Create project as subdirectory of: C:/Data Analysis

☐ Open in new session

Then click **Create Project**.

We are going to use this 'Statistics Course' Project to save all of the code and analysis which we will perform over the next three days.

Whenever you start on a new data analysis project - you MUST make a new projects file! For example, my folder currently looks like this:

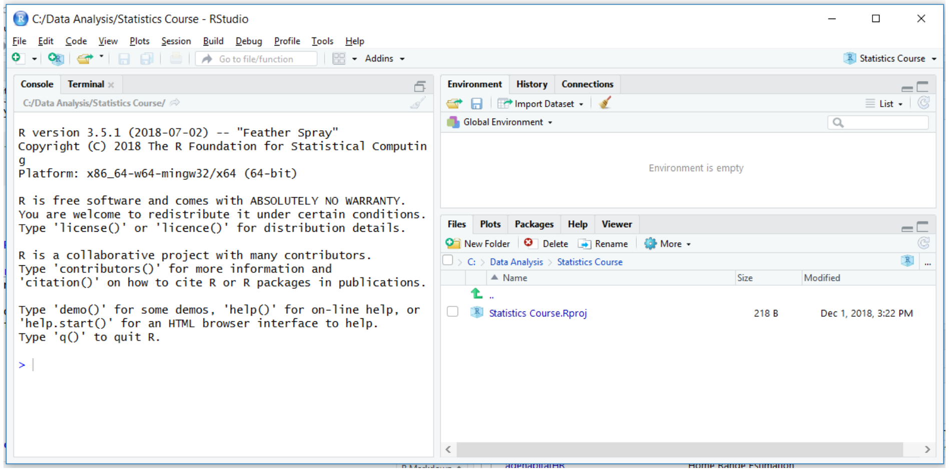


Name	Date modified	Type
24 Hour paper with Roland	11/16/2018 3:57 PM	File folder
Allometric Equations - Gabon	9/27/2018 8:56 AM	File folder
Carbon covarience paper	9/27/2018 8:41 AM	File folder
Community Wildlife	9/27/2018 8:53 AM	File folder
Congo Tree Plots	9/27/2018 8:47 AM	File folder
Congo Wildlife	9/27/2018 8:57 AM	File folder
Elephant Exclosures	9/27/2018 8:52 AM	File folder
Elephant mounring - Pachyderm field note	9/27/2018 8:52 AM	File folder
Elephant Movement Analysis	11/19/2018 1:20 PM	File folder
FEMBE project	11/19/2018 6:28 PM	File folder
Mapping files	9/27/2018 8:41 AM	File folder
Palm Oil and Elephant Conservation	9/27/2018 8:52 AM	File folder
Phenology	9/27/2018 8:43 AM	File folder
Plant Enemies	9/27/2018 8:41 AM	File folder
WorldClim Raster Building	10/4/2018 11:34 A...	File folder

Each folder contains one (or more) project files which I use to do my data analysis. I promise this will simplify your life and avoid confusion!

You can also share this folder with collaborators so that they can run all of your code without changing file paths… or share it with me if you are getting stuck on something!

Your R Studio window should now look like this:



```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computin
g
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Name	Size	Modified
..		
Statistics Course.Rproj	218 B	Dec 1, 2018, 3:22 PM

You can see the original ‘R console’ on the bottom left-hand side.

2.4.2 Create a data sheet to save your code

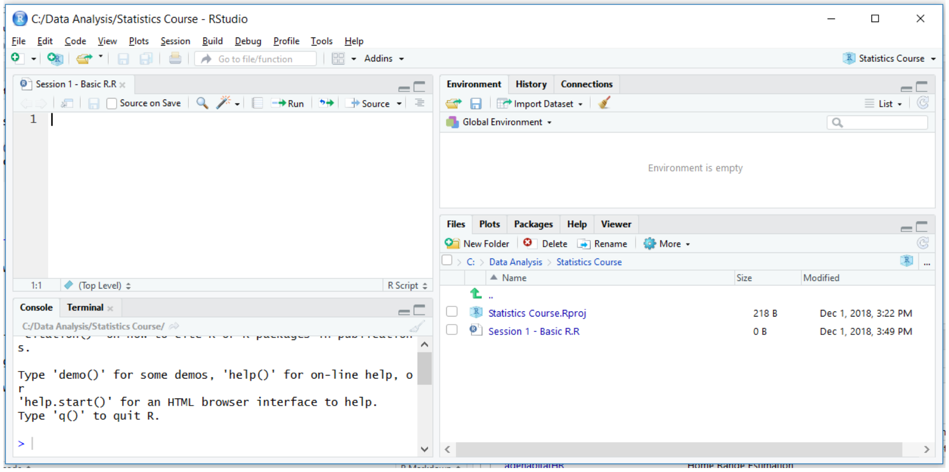
We could start by typing our data straight into the R console. But it would not easily be able to see the work we have done, or repeat the analysis in the future. A much better idea would be to save your data in a worksheet!

To do this we need to create an “.R” file.

Click on ‘File’ -> ‘New File’ -> ‘R Script’ *or* ‘Ctrl+Shift+N’

Before you do anything else… save this sheet! Call it ‘Session 1 - Basic R’

Your R studio will now look like this:



C:/Data Analysis/Statistics Course/

```
'citation()' on how to cite R or R packages in publication
s.

Type 'demo()' for some demos, 'help()' for on-line help, o
r
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> 
```

Name	Size	Modified
..		
Statistics Course.Rproj	218 B	Dec 1, 2018, 3:22 PM
Session 1 - Basic R.R	0 B	Dec 1, 2018, 3:49 PM

We are now ready to code!

2.4.3 Name your files

You should always give your code sheet a title. I follow the following convention:

```
# Title: Basic R Coding
# Date: 1st December 2018
```

The first universal rule of working in R is:

Always comment your code

The # symbol makes whatever you have written invisible to R. If you do not do this, you'll come back to you work in 6 months and have no idea what any of it means. Commenting your code will mean your past work is a valuable future resource and will make you feel less overwhelmed.

The second universal rule is:

Put spaces inbetween your code

Squashingeverythingintoonelinemakesitverydifficulytoread. Although saving space is economical, it is buch better to be clear.

Use linebreaks and spaces to make the structure and logic behind your work clear.

2.5 First Steps

Import the data set:

```
elephant <- read.csv("ClassData/Elephant.csv", header=T)
```

To run your code hit 'Ctrl+Enter' *or* click the 'Run' command in the top line of the R sheet.

You can run multiple lines of of code at one by highlighting them and pressing 'Ctrl+Enter'.

2.5.1 Veiwing Data

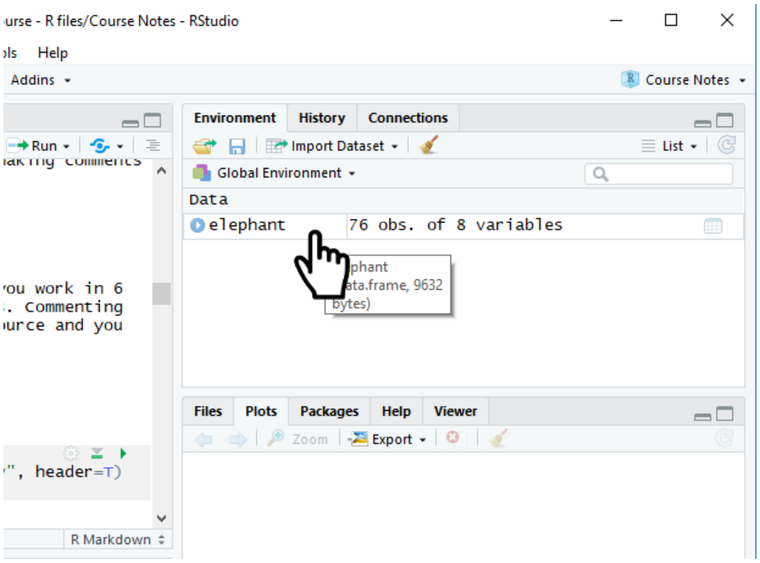
I see students opening their data in Excel to look at it *all the time*, but it is very easy in R!

We can take a quick look at our data using the `head()`. It prints the first few lines of a dataframe, and gives you a good snapshot of what is going on.

```
head(elephant)
```

```
##      Name    Sex        Region HomeRange      NDVI TuskLength GroupSize
## 1    Lisa Female Wonga Wongue    239.32 0.7264260       0.09         2
## 2    Rosa Female Wonga Wongue    184.35 0.7690821       1.14         7
## 3  Kengue   Male Wonga Wongue    478.93 0.7737994       1.47         4
## 4   Mambo   Male Wonga Wongue    279.69 0.7611887       0.30         1
## 5   David   Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 6 Mboumba   Male Wonga Wongue   1454.34 0.7614236       2.04         3
```

We can take a more in-depth, excel style, look by clicking on the ‘elephant’ dataframe in the ‘Environment’ frame (top right).



```
elephant    76 obs. of 8 variables
```

```
ou work in 6
. Commenting
urce and you
", header=T)
```

2.5.2 Data summaries

Let’s explore the `elephant` data! First, we want to know how many elephants of each sex, the largest recorded group size, and the mean tusk length in our dataset.

Enter the following into R

```
table(elephant$Sex) # How many male/female elephants?
```

```
##
## Female   Male
##     26     22
```

```
max(elephant$GroupSize) # What is the largest group size?
```

```
## [1] 12
```

```
mean(elephant$TuskLength) # What is the largest tusk length in the dataset?
```

```
## [1] 0.7
```

Let's now calculate the average tusk length for males and females. We can use the `aggregate()` function to calculate mean tusk lengths separately for each sex.

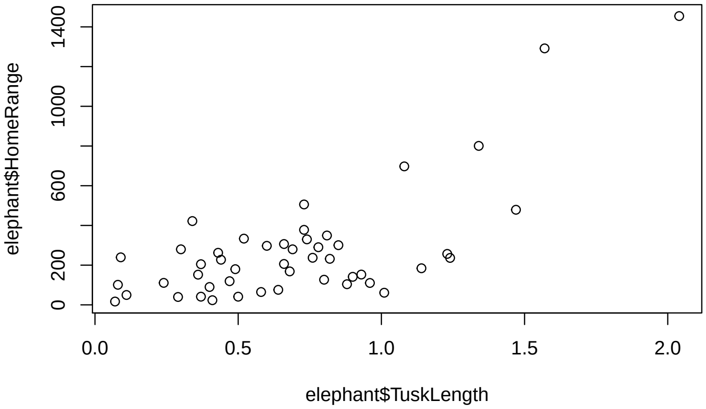
```
# Average tusk length by sex

aggregate(formula= TuskLength ~ Sex,
          data=elephant,
          FUN=mean)
```

```
##      Sex TuskLength
## 1 Female  0.5992308
## 2   Male  0.8190909
```

2.5.3 Plot your data

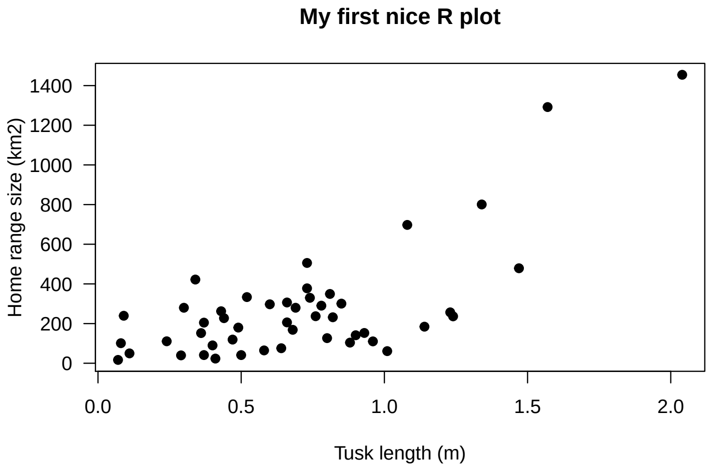
Let's make a plot of home range size versus tusk length.

```
plot(x = elephant$TuskLength, y = elephant$HomeRange) # simple r scatter plot
```



Interesting… but not very pretty. Lets clean it up.

```
plot(x = elephant$TuskLength, y = elephant$HomeRange,
     main = "My first nice R plot",     # Main heading
     xlab = "Tusk length (m)",          # X-axis label
     ylab = "Home range size (km2)",    # Y-axis label
     pch=19,                            # Filled circles
     las=1                              # Rotate y-axis labels
     ) # simple r scatter plot
```



2.5.4 Test your hypotheses

Let's do some basic hypothesis testing. First we will perform a two-sample t-test to see if male elephants and female elephants have different group sizes.

```
t.test(formula = GroupSize ~ Sex,
       data = elephant,
       alternative = 'two.sided')
```

```
##
##  Welch Two Sample t-test
##
## data:  GroupSize by Sex
## t = 0.31318, df = 44.862, p-value = 0.7556
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.139534  1.559114
## sample estimates:
## mean in group Female   mean in group Male
##             3.846154             3.636364
```

Our p-value of 0.7556 suggests that we do not have sufficent evidence to say there is a difference in group size between males and females.

Next, lets see if there is a correlation between the tusk length of an elephant and its home range size.

```
cor.test(formula = ~ TuskLength + HomeRange,
         data = elephant)
```

```
##
##  Pearson's product-moment correlation
##
## data:  TuskLength and HomeRange
## t = 6.6534, df = 46, p-value = 3.021e-08
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.5195249 0.8210520
## sample estimates:
##       cor
## 0.7002868
```

We got a p-value of 3.021e-08, which is P < 0.00000003. Which means that it is very unlikely that the positive relationship between tusk length and home range size came about by chance. Tusk length predicts home range size.

2.5.5 Regression analysis

Finally, lets run a full regression analysis to see if home range size is infleuced by sex, tusk length and an elephant group size.

```r
# Fit the model
reg.model <- lm(formula = HomeRange ~ Sex + TuskLength + GroupSize,
                    data = elephant)

#Show the summary statistics
summary(reg.model)
```

```
##
## Call:
## lm(formula = HomeRange ~ Sex + TuskLength + GroupSize, data = elephant)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -318.82 -157.82   12.18  107.21  578.80
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -52.027     78.994  -0.659    0.514
## SexMale       87.325     61.249   1.426    0.161
## TuskLength   455.194     75.454   6.033 3.02e-07 ***
## GroupSize     -9.269     12.959  -0.715    0.478
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 203.4 on 44 degrees of freedom
## Multiple R-squared:  0.5191, Adjusted R-squared:  0.4863
## F-statistic: 15.83 on 3 and 44 DF,  p-value: 4.006e-07
```

The only strong predictor of elephant home range size is tusk length!

2.5.6 Congratulations

You are an R programmer!

We just imported a data set, explored it and did some basic analysis in no time at all! R is amazing… if you know what you are doing.

Let’s take a few steps back and start learning the basics.

Chapter 3

R Basics

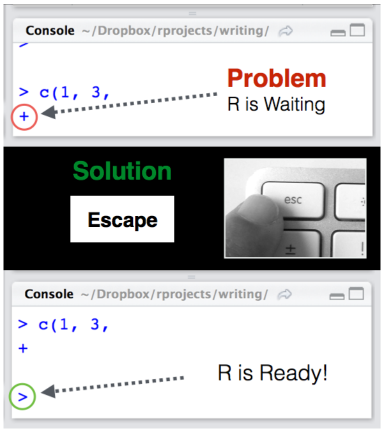
R has some quirks which you should understand before we continue:

- Anything that follows a # on the command line is taken as a comment and ignored by R. Comments can be included almost anywhere. In R Studio when you insert a # symbol the line changes colour (default = green) so that you know it is a comment.
- R is a case sensitive language. i.e. 'A' is not the same as 'a' and can be used to name different variables

Try running the following

```
A <- "Be careful with captials"
a
```

- Separate commands should be separated by a new line (but you can also use a semicolon ;)
- A very common problem is that you try to run something and nothing happens. A frequent cause if this is that you did not finish your statement, and R is waiting for additional information. When this happens a continuation prompt + will appear as your command is not complete. Usually you have forgotten to close a bracket! So R has two states: **ready** (>) or **wating** (+). If you have this issue, the best thing to do it hit escape, try to correct the problem, and try again.


Console ~/Dropbox/rprojects/writing/
Problem
R is Waiting
Solution
Escape
esc
Console ~/Dropbox/rprojects/writing/
R is Ready!

```
> c(1, 3,
+
```

```
> c(1, 3,
+
>
```

- Always troubleshoot your code. Never assume anything has worked… **always check it!**

3.1 Quick Glossary

- `<-` is the 'gets function'
- (and) are 'brackets'
- [and] are 'square brackets'
- `""` are 'quotation marks'
- `folder/folder/Your data.csv` is a 'filepath'

3.2 Objects and functions

R deals with two key elements: objects and functions:

- **Objects** are packets of data (numbers, text, dataframes etc) stored in the 'global environment'.

```
object <- c(3,4,6,7)
```

- **Functions** are proceedures which you apply to objects to return a new, altered, object. For example `mean()` will take an object containing lots of values, take the average, and retun it for you to see or assign to a new object.

```
mean(object)
```

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

As you get more comfortable in R, the functions you apply will get more and more complex, however the fundamentals remain the same: take an object -> apply a function -> store/interpret the results.

3.3 R as a calculator

The easiest way to use R is as a calculator. Write the following into your R script file:

```
# Addition (ALWAYS COMMENT YOUR CODE!!!)
2+2
```

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

TASK 1 Try out the following functions: multiplication `*`, subtraction `-`, division `/`, logarithm `log()`, exponent `exp()` and square `^`

3.4 Assigning single values (a.k.a. scalars)

What if we want to store the results of our calculations? We have completed our calculations, but none of them have been saved in the workspace. Look at the "Environment" in the top right… It only has the `elephants` dataset set from earlier. Where are our results?

To assign a single value to a variable or 'object' we use the 'gets' operator `<-`

```
# The gets operator
b <- 4    # Literally object 'b' gets 4
```

Look at the global environment now! It has something in it: `b`. Objects with a single value are called 'scalars'. Check what information the scalar, `b`, contains:

```
b
```

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

Lets increase the value of `b` by two

```
# Remember b = 4
b + 2
```

```
## [1] 6
```

We get the answer 6… Great. Let's double check what b is…

```
b
```

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

b is still four because we didnt update it! **Remember** to change an object you **MUST UPDATE IT WITH THE GETS FUNCTION!**. The correct way to do this is:

```
b <- b + 2 # The proper way to update an object
b
```

```
## [1] 6
```

You can create objects with expressions too:

```
c <- log(8)
c
```

```
## [1] 2.079442
```

Once you have objects in your workspace (i.e. `b` and `c`), you can do arithmatic with them… more on this below:

```
b + c
```

```
## [1] 8.079442
```

3.5 Assigning multiple values

3.5.1 Concatenation (a.k.a. vectors)

This is one of the most important operators: `c()` . I means to concatenate or link together. It allows you to link multipe values into a vector. Vectors are objects with multiple values.

```
w <-c(2,3,1,6,4,3,3,7) #creates a vector with these numbers
w
```

```
## [1] 2 3 1 6 4 3 3 7
```

If you perform an operation on a vector (multiple value object) with a scalar (single value object), R will cleverly apply the scalar to each element in the vector. For example, if you have a vector and want to add 100 to each element, just add the vector and scalar objects. Let's add 100 the `w` variable we just created:

```
w + 100 # vector + scalar
```

```
## [1] 102 103 101 106 104 103 103 107
```

3.5.2 Sequences

Sometimes you will want to generate repeated sequences of number or words. There are several ways of doing this.

1. A regular sequence of integer values :

```
d <- c(1:10)    # create a vector of whole numbers from 1 to 10
d
```

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

2. A sequence of non-integer values `seq(from, to, by, length.out)`

```
e <-seq(from = 1,
        to = 5,
        by = 0.5)   # create a sequence from 1 to 5 in 0.5 steps
e                 # Note : length out controls the number of elements in your sequence
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

3. A series of repeated elements `rep(X, times, each, length.out)`

```
e <-rep(x=2,times=10)   # repeats x 2, 10 times
e
```

```
##  [1] 2 2 2 2 2 2 2 2 2 2
```

4. Non-numeric sequences can also be produced

```
f <- rep("Osa Conservation",times=3)    # repeats abc 3 times
f
```

```
## [1] "Osa Conservation" "Osa Conservation" "Osa Conservation"
```

5. Repeat a series of repeats of a sequence

```
g <-rep(c(1:5),times=3) # repeats the series 1 to 5, 3 times
g
```

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

6. Repeat the individual elements of a series

```
h <-rep(1:5,each=3) # repeats each element of the series 3 times
h
```

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

Summary table of the functions to create vectors

Function	Example	Result
c(a,b,...)	c(1, 5, 9)	1, 5, 9
a:b	1:5	1, 2, 3, 4, 5
seq(from, to, by, length.out)	seq(from = 0, to = 6, by = 2)	0, 2, 4, 6
rep(x, times, each, length.out)	rep(c(7, 8), times = 2, each = 2)	7, 7, 8, 8, 7, 7, 8, 8

TASK 2 Sequence challenges

3.5.3 Vector Arithmatic

Vector arithmatic very important to get the hang of - you can perform the same function to multiple values simulteneously!

```
x <- c(1,2,3,4)
y <- c(5,6,7,8)

x*y
```

```
## [1]  5 12 21 32
```

```
x-y
```

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

3.5.4 Vector Summaries

Some typical functions used with vectors include `mean()`,`var()`, `sd()`,`range()`,`length()`,`max()`,`min()`, `summary()`. I use `summary()` and `length()` very frequently.

Task 3 Try out all of the data summary functions listed above on the following dataset... remember to write notes for each example!

```
i <- c(1,2,2,3,4,5,6,6,6,7)

#Example
mean(i) # gives the average
```

```
## [1] 4.2
```

3.5.5 Ordering and manipulating vectors

If you want to become a proficcient R user, knowing how to manipulate data is **essential!** You will use all of the following commands frequently… be sure to practise them.

3.5.5.1 Extracting values

You will frequently need to extract particular elements from a vector. In order to extract a single element use square brackets, `[ ]`, containing the position, or index, of the element. For example:

```
i[3] # Takes the third value out of the "i" datastring defined above
```

```
## [1] 2
```

To extract a sequential range of values:

```
i[c(3:9)]   # extracts the values of elements 3 to 9
```

```
## [1] 2 3 4 5 6 6 6
```

To extract a non sequential range of values:

```
i[c(2,4,6,8,10)]    # extracts the values for elements 2,4,6,8,10
```

```
## [1] 2 3 5 6 7
```

3.5.5.2 Extracting values using logic

The importance of this can not be overstated! R gives you the power to subset data using logical statements.

```
i[i>4] # extracts all values which are greater than 4
```

```
## [1] 5 6 6 6 7
```

```
i[i==6] # extracts all values which are equal to six
```

```
## [1] 6 6 6
```

The key logic arguements are listed below:

==	equal
!=	not equal
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
|	or
!	not
%in%	in the set

*** Task 4 Try to run an example of each of the commands listed above***

3.5.5.3 Sorting values

Vectors can be ordered using the functions `sort()` and `rev()`. Some examples are given below:

```
y <- c(4,2,5,6,4,3,5,6,7,4,3)
sort(y) # Places all values in ascending order
```

```
##  [1] 2 3 3 4 4 4 5 5 6 6 7
```

```
rev(sort(y)) # Places all values in descending order.  rev() = reverse
```

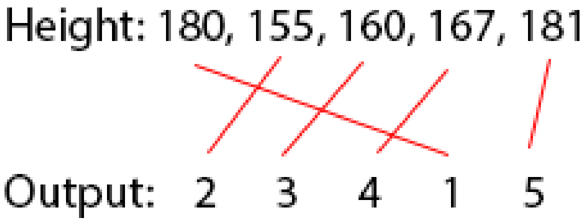
```
##  [1] 7 6 6 5 5 4 4 4 3 3 2
```

To be honest, I very rarely use `sort()`, I find `order()` to be much more useful. Have a look at the output below:

```
height <- c(180,155,160,167,181)
order(height)
```

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

To interpret this, lets start with the `order(height)` output. The first value, 2, (remember ignore [1]) should be read as 'the smallest value of height is the second element of height'. If we check this by looking at height, you can see that element 2 has a value of 155, which is the smallest value. The second smallest value in height is the 3rd element of height, which when we check is 160. The largest value of height is element 5 which is 181.



Now suppose the variable height is the height (in cm) of five different women. We know the names of these women and can store their names in a variable called w.names.

```
names <-c ("Joanna","Charlotte","Helen","Karen","Amy")
```

Now we can order the names of the women according to their height

```
height.ord <- order(height) # creates a variable of ordered height
names[height.ord]
```

```
## [1] "Charlotte" "Helen"     "Karen"     "Joanna"    "Amy"
```

You are probably thinking 'what's the use of this?' Well, imagine you have a dataset which contains two columns of data and you want to sort each column. If you just use `sort()` to sort each column separately, the values of each column will become uncoupled from each other. By ordering one column and then ordering the other column based on the value of the first column you will keep the correct association of values. More on this later!

3.5.5.4 Subsetting elements in a vector

There are times when your variables may be too long, or you want to create a new column. `substr()` command is very useful to know. It works as `substr(data, starting character, ending character)`:

```
# substring
substr(names,1,3)
```

```
## [1] "Joa" "Cha" "Hel" "Kar" "Amy"
```

3.5.5.5 Switching to upper/lower case

There are some very useful commands to standardise variables in a dataset. Try out the following:

```
# Switch 'names' to lower case
tolower(names)
```

```
## [1] "joanna"    "charlotte" "helen"     "karen"     "amy"
```

```
# Switch 'names' to upper case
toupper(names)
```

```
## [1] "JOANNA"    "CHARLOTTE" "HELEN"     "KAREN"     "AMY"
```

3.5.5.6 Removing whitespace

Sometime entering data into excel, you create variables with whitespaces in. They can be very difficult to detect, and R will treat them as different. For example, "Johanna", "Johanna" and " Johanna" will be treated as different factor levels by R. You can remove these with `trimws()`:

```
# Remake the 'names data frame with white spaces
names <- c("Johanna", "Johanna ", " Johanna", "Jo Hannah")
table(names)
```

```
## names
##   Johanna Jo Hannah   Johanna  Johanna
##         1         1         1         1
```

```
# Correct the problems with trimws(), note trimws() only deals with spaces at the start
# or the end!
names <- trimws(names)
table(names)
```

```
## names
## Jo Hannah   Johanna
##         1         3
```

3.5.6 Removing objects from the workspace

Through out this session we have been adding things to our workspace… Check out the global environment in the top right! What if we want to remove things to keep things tidy?

```
rm(P) # Removes a single object
```

```
## Warning in rm(P): object 'P' not found
```

```
rm(b,c,Q) # remove multiple objects using a comma.
```

```
## Warning in rm(b, c, Q): object 'Q' not found
```

Task 5 Practise creating sequences of information and subsetting them

3.6 Quitting R

When you are finished with R, save your worksheet and close R. R Studio will ask you if you want to save your workspace image. This saves all of the files in the 'Global environment' (on the right) for when you open the project next time. If you do not do this, you will have to re-run all of your code each time you load R.

3.7 Packages

More often than not, you will need to look outside of 'base R' in order to perform certain functions or analyses. To do this we need to download a 'package' into R.

If you want to see the (ever expanding) list of packages available in R, type in:

```
CRAN.packages()
```

3.7.1 Install your first package

We will new install 'dplyr' - a package which will use later. Do not worry about what it does for now!

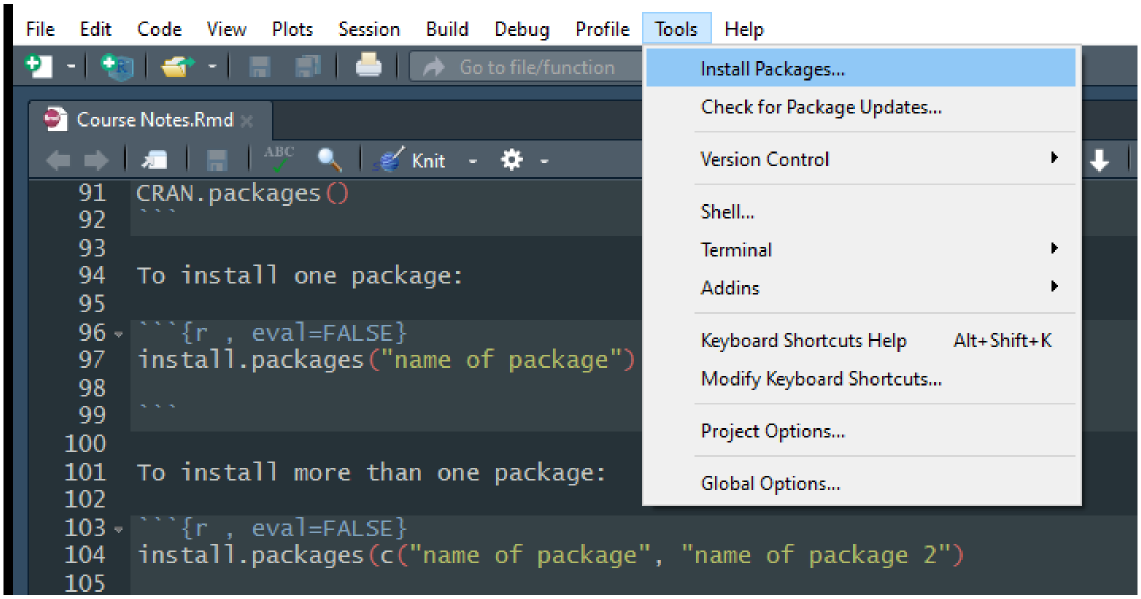
To install a package:

```
install.packages("dplyr")
```

Packages are updated frequently, to update yours type:

```
update.packages()
```

Or you can install packges through the 'Tools' heading:



~~~~
CRAN.packages()
```

To install one package:

```{r , eval=FALSE}
install.packages("name of package")

```

To install more than one package:

```{r , eval=FALSE}
install.packages(c("name of package", "name of package 2")

~~~~

3.7.2 Loading a package

You will realise that just because you have installed a package does not mean you can actually use it. This is to prevent R taking up lots of computer memory with packages you are not currently using. To load a package into the workspace (and therefore make it available) you must run the following:

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

It is now unpacked and ready to use!

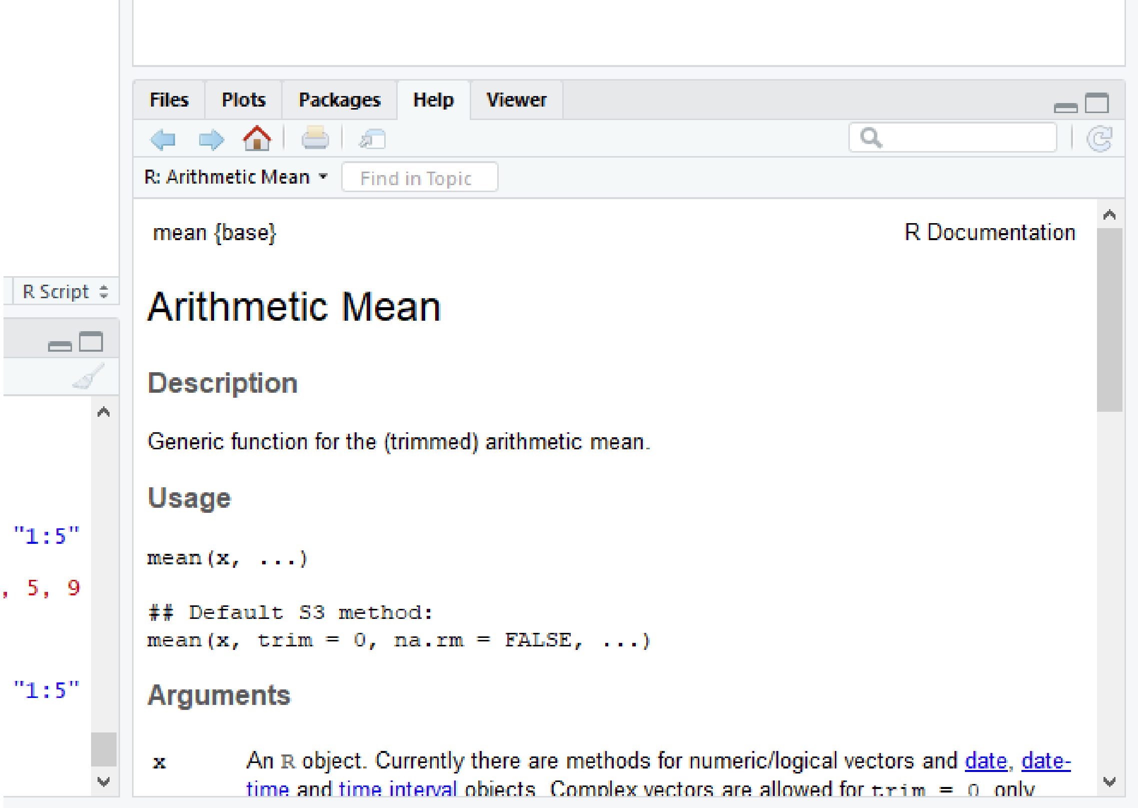
To see a list of active packages, click the ‘Packages’ tab in the bottom right-hand corner.

3.8 Getting help

Use the question mark.

```
?plot
```

This opens an interactive display in the bottom right hand corner of R studio.



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only

Although daunting at first, this information is actually very useful:

Description: gives a brief description of the function.

Usage: gives the name of the arguments associated with the function and possible default values (options).

Arguments: provides more detail regarding each argument.

Details: gives further details of the function.

Value: if applicable, gives the type of object returned by the function or the operator.

See Also: provides information on other help pages with similar or related content.

Examples: gives some examples of using the function. You can also access examples at any time by using the example() function (i.e. example(plot))

I most frequently veiw the **Arguements** and **Examples** tabs.

If that does not help… **google it!**

Chapter 4

Advanced coding

So far we have been dealing with single value scalars or multiple value vectors created in R. But what if you want to start importing and analysing more complicated datasets collected in the field? We first need to learn how to import and analyse our data.

4.1 Classes of data

Data in R come in two principal flavours: numbers (usually integer or continuous) and strings (uausally characters or factors). They each behave in slightly different ways. It is very important to know when and where to utilise them, and how to convert between them.

4.1.1 Numbers

A numeric object is any integer (e.g. 7 or -4) or real number (e.g. 3.14). You do not have to do anything special to create a numeric object, if a column contains only numbers it will be classed as numeric.

```
a <- c(1,2,3) # generate a vector of pure numbers
class(a) # check the class of the data
```

```
## [1] "numeric"
```

Beware! If there are non-numbers in the data to, it will be treated differntly by R. This is a common issue when you import a dataset which contains data entry errors. The column you expected to be classed as numeric is classed as a 'character' object.

```
a <- c(1,"f",3) # generate a variable with some data
class(a) # check the class of the data
```

```
## [1] "character"
```

4.1.2 Strings

Strings can take several forms, but the dominant types are characters and factors.

4.1.3 Characters

Characters are strings of letters and numbers and can take any unique combinations. They are very flexible, easy to deal with and always require quotation marks "".

```
b <- c("Dave", "Brian", "John")
b
```

```
## [1] "Dave"  "Brian" "John"
```

```
class(b)
```

```
## [1] "character"
```

A very common reason for code failing is forgetting the "" quotation marks when referencing a character string. When you do this, R will look for the word you have typed in the global environment - it thinks it is an object! If it isn't there, it will retun an error:

```
c <- c(Dave, Brian, John)
```

Error: object 'Dave' not found

4.1.4 Factors

Factors are superficially similar to character strings, but rather than being flexible, they have ordered levels hard-wired into their data structure.

```
c <- factor(c("Dave", "Brian", "John"))
c # Note the 'Levels:' output which tells you what values the different factors can take.
```

```
## [1] Dave  Brian John
## Levels: Brian Dave John
```

```
class(c)
```

```
## [1] "factor"
```

You can also find out what the different levels are inside a factor using 'levels()'

```
levels(c)
```

```
## [1] "Brian" "Dave"  "John"
```

The way to think about factors is as a set of labelled drawers holding the data. The draws are references one to the number of drawers, and if you remove all the data from a given draw, the draw remains in the dataset to be filled again.

Let me show you this by changing "Brian" to "Dave":

```
c[c=="Brian"] <- "Dave"
c
```

```
## [1] Dave Dave John
## Levels: Brian Dave John
```

We no longer have a “Brian” in the dataset, but the level “Dave” remains to be used. We can remove an empty level in a set of factors using the `factor()` commond:

```
c <- factor(c)
c
```

```
## [1] Dave Dave John
## Levels: Dave John
```

4.1.5 Changing data structure

There comes a time in every coders life when you need to switch the nature of your data from one type to another. This is easily accomplished by using the ‘as.character()’, ‘as.factor()’, and ‘as.numeric()’ commands.

```
# Create a numeric vector
d <- c(2,6,8,5,5,5,6,7,8,5)
d # all looks normal
```

```
##  [1] 2 6 8 5 5 5 6 7 8 5
```

```
class(d)
```

```
## [1] "numeric"
```

As expected… it is numeric. Let’s change it to a character string.

```
# Create a numeric vector
d <- as.character(d)
d # Note how there are now ""'s around the numbers.
```

```
##  [1] "2" "6" "8" "5" "5" "5" "6" "7" "8" "5"
```

```
class(d)
```

```
## [1] "character"
```

Change it to a factor.

```
# Create a numeric vector
d <- as.factor(d)
d # Note how now we have levels!
```

```
##  [1] 2 6 8 5 5 5 6 7 8 5
## Levels: 2 5 6 7 8
```

```
class(d)
```

```
## [1] "factor"
```

Complete the circle, and change it back to numeric.

```
# Create a numeric vector
d <- as.numeric(d)
d # WHAT!!!???!! The numbers have changed.
```

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

Never change a factor to numeric like that. The numbers the values take are the “levels” contained within the factor. **THIS IS WHY YOU SHOULD ALWAYS TROUBLESHOOT YOUR CODE**. If you do need to go from a factor to numeric, always use ‘as.character()’ as an intermediate command first.

4.2 Dataframes

Dataframes sound a little dull… but they are actually powerful two dimensional vector holding structures! They will be the most common way you store data in R. Other ways of storing data exist (e.g. lists, matrices), but I will not discuss them here - just be aware they exist! Dataframes are composed of rows and columns and can contain data of different types (e.g. numeric, character and factor), whereas vectors must always be the same data type.

4.2.1 Exploring dataframes

If you want to become proficient data analysis, you need to become a pro at the exploration and manipulation of dataframes. To do this properly, you will need master a range of different R techniques, from data summarisation to plotting. Tomorrow we will do this on a real dataset. Today, we will keep working on our elephant dataframe.

Let’s import and examine the structure of the elephant dataframe:

```
elephant <- read.csv("ClassData/Elephant.csv", header=T)
head(elephant)
```

```
##      Name    Sex        Region HomeRange      NDVI TuskLength GroupSize
## 1    Lisa Female Wonga Wongue    239.32 0.7264260       0.09         2
## 2    Rosa Female Wonga Wongue    184.35 0.7690821       1.14         7
## 3  Kengue   Male Wonga Wongue    478.93 0.7737994       1.47         4
## 4   Mambo   Male Wonga Wongue    279.69 0.7611887       0.30         1
## 5   David   Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 6 Mboumba   Male Wonga Wongue   1454.34 0.7614236       2.04         3
```

The dataset above contains the results from one year of tracking elephants with GPS collars in Gabon. The dataframe has 7 variables (columns) and each row represents data from an individual elephant.The variables `Name`, `Sex` and `Region` are catagorical. `HomeRange`, `NDVI`, `TuskLength` and ‘GroupSize“ are continuous.

4.2.2 Numerical Summaries

The first thing we should do is examine the data structure using 'str()'

```
str(elephant)
```

```
## 'data.frame':    48 obs. of  7 variables:
##  $ Name      : Factor w/ 48 levels "Abu","Amelia",..: 21 41 17 23 7 26 25 30 31 35 ...
##  $ Sex       : Factor w/ 2 levels "Female","Male": 1 1 2 2 2 2 2 1 1 2 ...
##  $ Region    : Factor w/ 4 levels "Ivindo","Loango",..: 4 4 4 4 4 4 4 4 4 4 ...
##  $ HomeRange : num  239 184 479 280 1292 ...
##  $ NDVI      : num  0.726 0.769 0.774 0.761 0.795 ...
##  $ TuskLength: num  0.09 1.14 1.47 0.3 1.57 2.04 0.93 0.68 0.8 0.73 ...
##  $ GroupSize : int  2 7 4 1 4 3 4 2 3 5 ...
```

When we imported the dataframe, R automatically defined the catagorical variables as 'Factors' and the continuous variables as numeric ('num') or integer ('int'). There is no practical difference between numeric and integer, just that integers do not have decimal places.

The 'str()' command gives you the number of observations (rows) and variables (columns). We can also extract this information with:

```
nrow(elephant) # number of rows
```

```
## [1] 48
```

```
ncol(elephant) # Number of columns
```

```
## [1] 7
```

If you want a summary of all of the information in your dataframe, you can use the 'summary()' function. For continuous variables you get the mean, median and range of values, for catagorical variables you get the frequency within each category:

```
summary(elephant)
```

```
##            Name        Sex                    Region     HomeRange
##  Abu         : 1   Female:26   Ivindo          : 6   Min.   :  16.78
##  Amelia      : 1   Male  :22   Loango          :15   1st Qu.: 108.67
##  Annabelle_ww: 1               Moukalaba Doudou: 6   Median : 216.58
##  Beti        : 1               Wonga Wongue    :21   Mean   : 271.87
##  Boniface    : 1                                     3rd Qu.: 301.99
##  BraBrou     : 1                                     Max.   :1454.34
##  (Other)     :42
##       NDVI          TuskLength       GroupSize
##  Min.   :0.7053   Min.   :0.0700   Min.   : 1.00
##  1st Qu.:0.7748   1st Qu.:0.4075   1st Qu.: 2.00
##  Median :0.7878   Median :0.6700   Median : 3.00
##  Mean   :0.7892   Mean   :0.7000   Mean   : 3.75
##  3rd Qu.:0.8073   3rd Qu.:0.8850   3rd Qu.: 5.00
##  Max.   :0.8493   Max.   :2.0400   Max.   :12.00
##  NA's   :1
```

4.2.3 Selecting entries in a dataframe

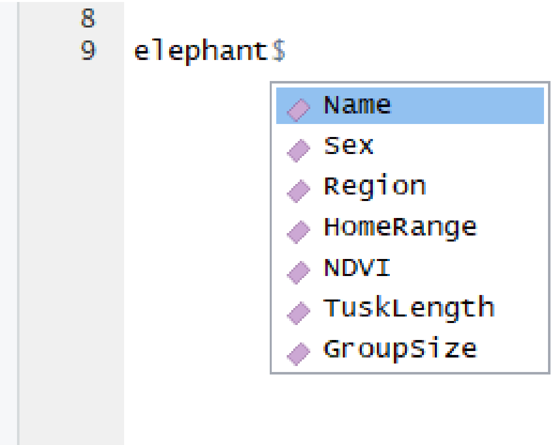
4.2.3.1 Columns

To select a single column you must provide its **full address**. The name of the data frame is the 'city' (in this case `elephant`), the column name is the street. You link them with the '$' sign.

```
elephant$Name
```

```
##  [1] Lisa         Rosa         Kengue       Mambo        David
##  [6] Mboumba      Mba          Nana         Ndeka        Nze
## [11] Wongo        Stam         Junior       Ponogo       Larouille
## [16] Nongo        Orembo       Tonda        BraBrou      Malaika
## [21] Boniface     Ekare        Marijo       Janice       Rose_Loango
## [26] Abu          Mpemba       Keva         Ta_a         Tonnere
## [31] Kigali       Onero        Izouwa       Nzamba       Annabelle_ww
## [36] Ngele        Moukalaba    Dedi         Penelope     Megane
## [41] Beti         Stephanie    Doudou       Dumbo        Kalie
## [46] Program      Estell       Amelia
## 48 Levels: Abu Amelia Annabelle_ww Beti Boniface BraBrou David ... Wongo
```

R studio is very helpful and provides you with column title hints after you have typed the '$' sign, but even so, you should always keep your column titles small in order to save time and keep your code looking clean.



```
elephant$
```

4.2.3.2 The cordinate method

You can also select elements from dataframes using their coordinates (also known as indexing or subscripting). To do this you use the [ ] square brackets. Again you must specify the address (the data frame), then the rows (x) and/or column (y) coordinates. **dataframe**[**row.number, column.number**]

For example, to extract the 2nd row's 3rd element:

```
elephant[2,3] #coordinate method
```

```
## [1] Wonga Wongue
## Levels: Ivindo Loango Moukalaba Doudou Wonga Wongue
```

You can select the whole row by leaving the column argument blank.

```
elephant[2,] #just the 2nd row
```

```
##   Name    Sex       Region HomeRange      NDVI TuskLength GroupSize
## 2 Rosa Female Wonga Wongue    184.35 0.7690821       1.14         7
```

Or select the whole column by leaving the row argument blank.

```
elephant[,3] # just the third column
```

```
##  [1] Wonga Wongue     Wonga Wongue     Wonga Wongue     Wonga Wongue    
##  [5] Wonga Wongue     Wonga Wongue     Wonga Wongue     Wonga Wongue    
##  [9] Wonga Wongue     Wonga Wongue     Wonga Wongue     Wonga Wongue    
## [13] Loango           Loango           Ivindo           Wonga Wongue    
## [17] Loango           Loango           Wonga Wongue     Wonga Wongue    
## [21] Loango           Loango           Ivindo           Ivindo          
## [25] Loango           Loango           Loango           Loango          
## [29] Ivindo           Wonga Wongue     Wonga Wongue     Loango          
## [33] Loango           Ivindo           Wonga Wongue     Wonga Wongue    
## [37] Moukalaba Doudou Moukalaba Doudou Moukalaba Doudou Moukalaba Doudou
## [41] Wonga Wongue     Wonga Wongue     Moukalaba Doudou Loango          
## [45] Loango           Moukalaba Doudou Loango           Ivindo          
## Levels: Ivindo Loango Moukalaba Doudou Wonga Wongue
```

If you want to extract more than one column and row you can use supply vectors:

```
elephant[c(1:3),c(1,3,5)]
```

```
##     Name       Region      NDVI
## 1   Lisa Wonga Wongue 0.7264260
## 2   Rosa Wonga Wongue 0.7690821
## 3 Kengue Wonga Wongue 0.7737994
```

4.2.3.3 The logic method

One of the most powerful ways to extract data in R is to use logical statements. For example, let's subset the data to only include males:

```
elephant[elephant$Sex=="Male",] # subset to just males. Remember that leaving a blank
```

```
##         Name  Sex       Region HomeRange      NDVI TuskLength GroupSize
## 3     Kengue Male Wonga Wongue    478.93 0.7737994       1.47         4
## 4      Mambo Male Wonga Wongue    279.69 0.7611887       0.30         1
## 5      David Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 6    Mboumba Male Wonga Wongue   1454.34 0.7614236       2.04         3
## 7        Mba Male Wonga Wongue    152.61 0.7627158       0.93         4
## 10       Nze Male Wonga Wongue    505.67        NA       0.73         5
## 11     Wongo Male Wonga Wongue    297.24 0.8040024       0.60         7
## 13    Junior Male       Loango     64.70 0.8300000       0.58         3
## 15 Larouille Male       Ivindo    101.05 0.7856077       0.08         6
## 17    Orembo Male       Loango    329.86 0.8085495       0.74         2
## 19   BraBrou Male Wonga Wongue    227.24 0.7614054       0.44         9
## 21  Boniface Male       Loango    231.81 0.7876408       0.82         2
## 22     Ekare Male       Loango     39.71 0.7936707       0.29         1
## 26       Abu Male       Loango    697.46 0.7877630       1.08         2
## 29      Ta_a Male       Ivindo     75.67 0.7052599       0.64         8
## 30   Tonnere Male Wonga Wongue    800.60 0.8131276       1.34         2
## 31    Kigali Male Wonga Wongue    237.10 0.8070265       0.76         3
## 32     Onero Male       Loango     89.94 0.8205471       0.40         2
## 34    Nzamba Male       Ivindo    256.60 0.7803319       1.23         6
## 41      Beti Male Wonga Wongue    300.54 0.7418243       0.85         1
## 44     Dumbo Male       Loango    205.92 0.7766205       0.66         3
## 45     Kalie Male       Loango    119.24 0.8008448       0.47         2
```

```
                                # space after the , returns all columns
```

Let's subset the data to only include elephants with home ranges greater than 500km2:

```
elephant[elephant$HomeRange>500,] # Home range greater than 500
```

```
##       Name  Sex       Region HomeRange      NDVI TuskLength GroupSize
## 5    David Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 6  Mboumba Male Wonga Wongue   1454.34 0.7614236       2.04         3
## 10     Nze Male Wonga Wongue    505.67        NA       0.73         5
## 26     Abu Male       Loango    697.46 0.7877630       1.08         2
## 30 Tonnere Male Wonga Wongue    800.60 0.8131276       1.34         2
```

You can link multiple logic statements together with the '&' sign. Extract elephants with homeranges greater than 500km2 *AND* tusks over 1.5m:

```
# Home range greater than 500 and tusk length  great than 1.5
elephant[elephant$HomeRange>500 & elephant$TuskLength>1.5,]
```

```
##      Name  Sex       Region HomeRange      NDVI TuskLength GroupSize
## 5   David Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 6 Mboumba Male Wonga Wongue   1454.34 0.7614236       2.04         3
```

If we want to extract elephants with tusks great than 2 meters **or** less than 0.5 meters? Easy! Use the or '|' operation:

```
# elephants with tusks greate than two meters, or less than 0.5 meters.
elephant[elephant$TuskLength<0.2 | elephant$TuskLength>1.5,]
```

```
##             Name    Sex       Region HomeRange      NDVI TuskLength
## 1           Lisa Female Wonga Wongue    239.32 0.7264260       0.09
## 5          David   Male Wonga Wongue   1291.67 0.7954598       1.57
## 6        Mboumba   Male Wonga Wongue   1454.34 0.7614236       2.04
## 15     Larouille   Male       Ivindo    101.05 0.7856077       0.08
## 28          Keva Female       Loango     49.69 0.8300000       0.11
## 35 Annabelle_ww Female Wonga Wongue     16.78 0.8076506       0.07
##    GroupSize
## 1          2
## 5          4
## 6          3
## 15         6
## 28         3
## 35         2
```

The '%in%' operation helps you to easily extract data based on multiple OR arguments. Imagine you have a list of elephants that you know were poached, and you want to subset the data to only view those.

```
poached <- c("David", "BraBrou", "Ekare") # create a vector of poached elephants
elephant[elephant$Name %in% poached,] # subset using %in%
```

```
##       Name  Sex       Region HomeRange      NDVI TuskLength GroupSize
## 5    David Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 19 BraBrou Male Wonga Wongue    227.24 0.7614054       0.44         9
## 22   Ekare Male       Loango     39.71 0.7936707       0.29         1
```

```
# Return all rows where elephant name matches the names in poached
```

TASK 6 Extract the data for female elephants in Wonga wongue.

4.2.4 dplyr and data summaries

So far we have just used functions contained in base R, but the package `dplyr` within the `tidyverse` (more on this later) has a lot to offer.

```
library(dplyr) # Load the dplyr package
```

First we will summarise home range and tusk length by park using the summarize function:

```
summarize(group_by(elephant, Region), # this says calculate the following using the
                                      # elephants dataframe, split by region
          Individuals=n(),            # give the number of individuals
          Avg.Area=mean(HomeRange),   # average Home range
          Avg.Tusk=mean(TuskLength))  # average tusk length
```

```
## # A tibble: 4 x 4
##   Region           Individuals Avg.Area Avg.Tusk
##   <fct>                  <int>    <dbl>    <dbl>
## 1 Ivindo                     6     158.    0.587
## 2 Loango                    15     198.    0.589
## 3 Moukalaba Doudou           6     252.    0.732
## 4 Wonga Wongue              21     363.    0.802
```

We can add increase the number of summary subdivisions easily too:

```
summarize(group_by(elephant, Region, Sex), # this says calculate the following using the
                                           # elephants dataframe, split by region
          Observations=n(),#give the number of Observations in each group
          Avg.Area=mean(HomeRange), #average Home range
          Avg.Tusk=mean(TuskLength)) #average tusk length
```

```
## # A tibble: 7 x 5
## # Groups:   Region [4]
##   Region           Sex    Observations Avg.Area Avg.Tusk
##   <fct>            <fct>         <int>    <dbl>    <dbl>
## 1 Ivindo           Female            3     171.    0.523
## 2 Ivindo           Male              3     144.    0.65
## 3 Loango           Female            7     169.    0.543
## 4 Loango           Male              8     222.    0.63
## 5 Moukalaba Doudou Female            6     252.    0.732
## 6 Wonga Wongue     Female           10     160.    0.582
## 7 Wonga Wongue     Male             11     548.    1.00
```

###TASK 7### ###Summarize group size by region###

4.2.4.1 Dealing with NA's

Try to find the average NDVI score across all elephants:

```
mean(elephant$NDVI)
```

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

Whaaat?! **R is broken. . .** Not really. One of our elephants is missing an NDVI score. This is very common in real world data where the complexities of life sometimes get in the way of data collection. NA's often sneak under the radar in datasets (particularly if you analse data with 1000's of observations).

You can easily adjust the `mean()` command to ignore NA's:

```
mean(elephant$NDVI, na.rm=T)
```

```
## [1] 0.7891777
```

However, what if you want to find the problem and correct it? Or remove it completely? We can try to use logic like in the previous examples:

```
elephant[elephant$NDVI=="NA",] # Subset the data where the NDVI score is NA
```

```
##    Name  Sex Region HomeRange NDVI TuskLength GroupSize
## NA <NA> <NA>   <NA>        NA   NA         NA        NA
```

That didn't work. NA's do not behave like normal data. You need to use the logical statement 'is.na()' to evaluate NA's. First lets look at what is.na() does:

```
is.na(elephant$NDVI) # what is.na does
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE
```

It produces a vector of true and false statements, if NDVI is not an NA it returns 'FALSE', if NDVI is an NA it returns 'TRUE'. We can use this information to make our selection!

```
elephant[is.na(elephant$NDVI)==TRUE,]
```

```
##    Name  Sex       Region HomeRange NDVI TuskLength GroupSize
## 10  Nze Male Wonga Wongue    505.67   NA       0.73         5
```

Now we see the real data!

Remove the NA from the dataset:

```
elephant <- elephant[is.na(elephant$NDVI)==FALSE,] # subset the data where is.na() is false.
```

In summary, NA's behave strangely, but dont get flustered by them. You just need to be vigilent and know how to deal with them.

4.2.5 Merging dataframes

Taking data from one source and merging it with another source is a fundamental step in performing data analysis. For example, you have collected lots of biodiversity data in the field and you want to merge it with some covariate data exported from ArcGIS.

In this case, your elephant project collaborators have just provided you with data on the disease status of the individuals in your data set, and you need to add it to your dataframe.

***Import the `disease` dataframe.

```
ele.dis <- read.csv("ClassData/Elephant.disease.csv", header=T)
```

###TASK 8#### *** Explore the new dataframe with the `str()` and `head()` functions. What can you tell me about it?***

In the past you might have opened the two files in excel and copied the data over... but what if the data are in different orders, have repeated individuals, or there are 1000's of rows? It would be a nightmare! We can do something much simplier in the `dplyr` package - a left join. `Left_join()` takes two datasheets and merges them using a common key (in this case `Name`), to add any columns from the right hand side (`ele.dis`) not present on the left hand side (`elephant`).

```
# First load the package
library(dplyr)
new.ele <- left_join(elephant, ele.dis, by="Name") # by specifies the key
head(new.ele)
```

```
##      Name    Sex       Region HomeRange      NDVI TuskLength GroupSize
## 1    Lisa Female Wonga Wongue    239.32 0.7264260       0.09         2
## 2    Rosa Female Wonga Wongue    184.35 0.7690821       1.14         7
## 3  Kengue   Male Wonga Wongue    478.93 0.7737994       1.47         4
## 4   Mambo   Male Wonga Wongue    279.69 0.7611887       0.30         1
## 5   David   Male Wonga Wongue   1291.67 0.7954598       1.57         4
## 6 Mboumba   Male Wonga Wongue   1454.34 0.7614236       2.04         3
##   Nematodes Abcess  Injury Disease
## 1    absent absent present    TRUE
## 2    absent absent  absent   FALSE
## 3    absent absent  absent   FALSE
## 4    absent absent present    TRUE
## 5    absent absent present    TRUE
## 6    absent absent present    TRUE
```

Always check the output is sensible

Magic! There are other types of joins you can use, but this is the one which I use most frequently. For more ideas check out http://stat545.com/bit001_dplyr-cheatsheet.html

4.2.6 Exporting dataframes

Take the updated elephant dataset you have just produced and use the `write.csv()` command. The arguments contained are `write.csv(object, "name.csv", row.names=F)`

If you do not use a file in your write csv() the file will turn up in the project directory. However you can write it to any folder on your computer using a file path.

```
write.csv(new.ele, "C:/Program Files/Home Range and Disease.csv", row.names=F)
```

4.3 Plots

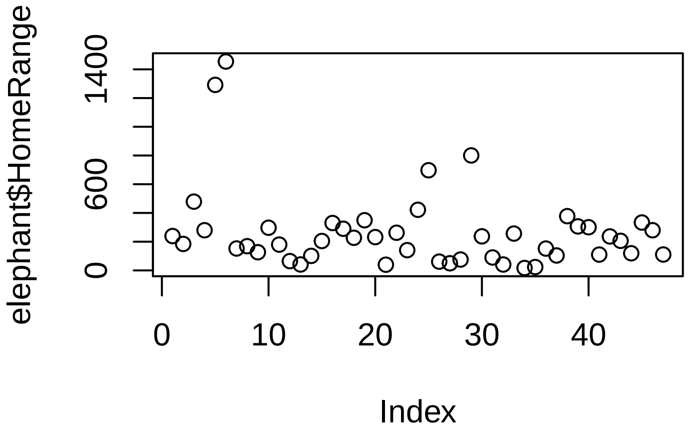
Summarising your data, either numerically or graphically, is an important (if often overlooked) component of any data analyses. Fortunately, R has excellent graphics capabilities! It can be used when you want to produce plots for initial data exploration, model validation or highly complex publication quality graphs.

If you can imagine a graph… you can plot it in R! Consequently it would be impossible to go through all possible combinations here. This will simply serve as a primer.

4.3.1 Scatter plots

The most common function used to produce graphs in R is the plot() function. The simplest plot is a single variable plot - e.g. elephant home ranges:

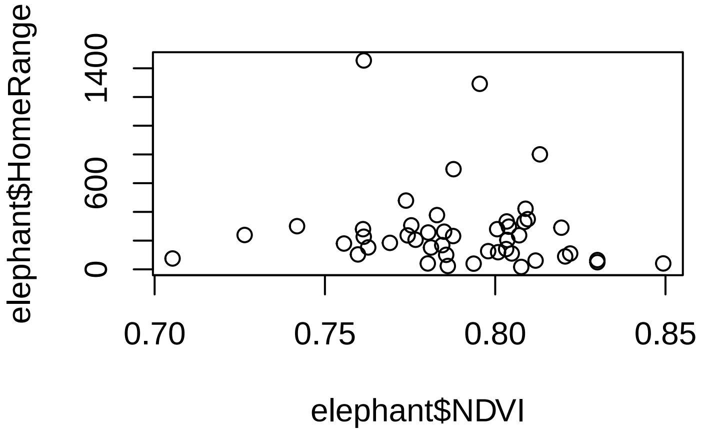
```
plot(elephant$HomeRange)
```



Here, R has plotted home range size on the y axis and an index (1 to the length of the dataframe) on the x axis, plotted in the order at which they occur in the data.

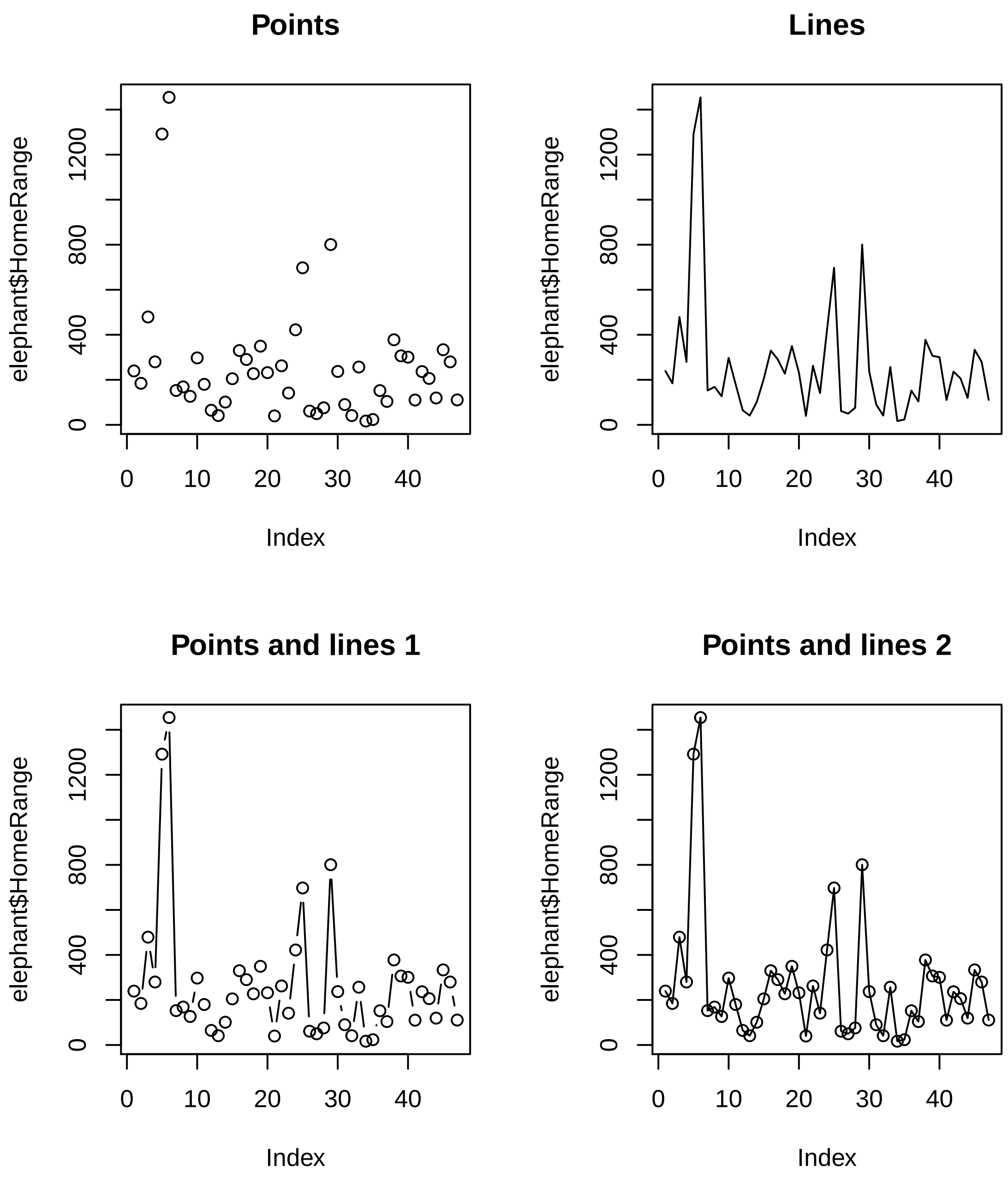
We can provide a second continuous variable (a scatter plot) with the following:

```
# You can use plot(x,y) or plot(y~x), they produce the same result.
#plot(elephant$HomeRange~ elephant$NDVI) # Will give the same result
plot(elephant$NDVI,elephant$HomeRange) # Scatter plot of two continuous variables
```



You can also specify the type of graph you wish to plot using the option type="". For example, you can plot just the points (type="p"), lines (type="l"), both points and lines connected (type="b") and both points and lines with the lines running through the points (type="o").

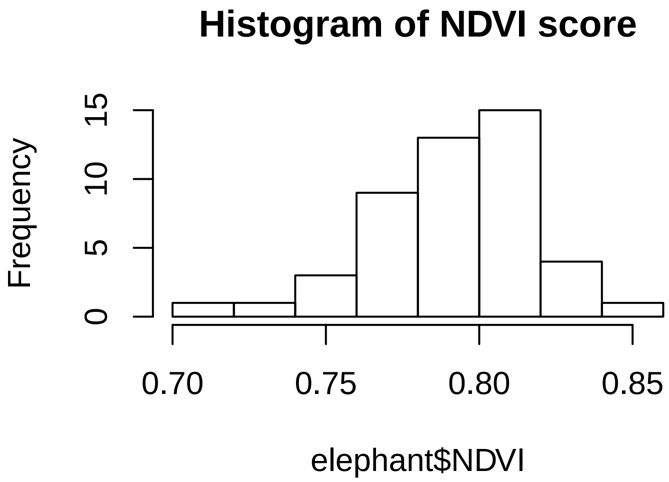
```
par(mfrow=c(2,2)) # allow a plot with four windows
plot(elephant$HomeRange, type="p", main="Points")
plot(elephant$HomeRange, type="l", main="Lines")
plot(elephant$HomeRange, type="b", main="Points and lines 1")
plot(elephant$HomeRange, type="o", main="Points and lines 2")
```



4.3.1.1 Histograms

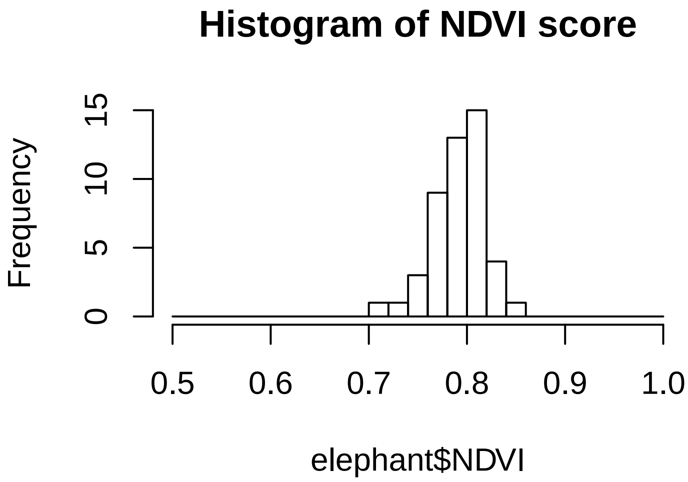
The hist() function allows you to draw a histogram of a variable in order to gain an impression of its frequency distribution. To plot a histogram of height.

```
hist(elephant$NDVI, main= "Histogram of NDVI score") #default histogram
```



You can manually change the number of breaks in a histogram by supplying a sequence spanning the range of the variable of interest.

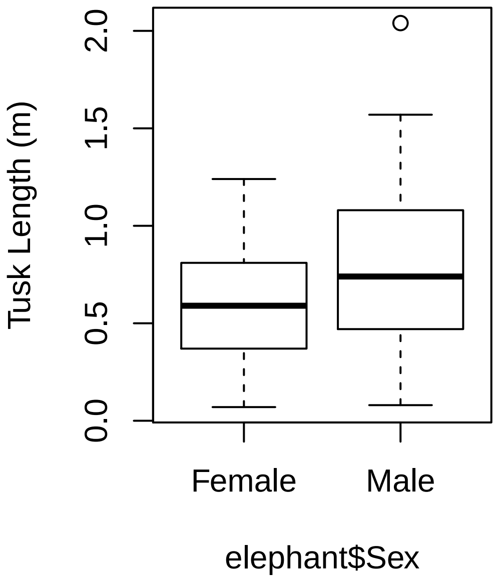
```
brks <- seq(0.5,1,0.02) # This sets the break width
hist(elephant$NDVI, breaks=brks,
     main= "Histogram of NDVI score")
```



4.3.1.2 Boxplots

Boxplots are useful for exploring how a continuous covariate is related to a catagorical covariates. They are specified as ‘boxplot(continuous~catagorical)’. **Note* you have to use the ~ with boxplots or you get weird results.

```
boxplot(elephant$TuskLength~elephant$Sex, ylab="Tusk Length (m)") # ylab controls the axis labels
```



To interpret a box plot: The cental black bar = median, the upper and lower edges are the 25% and 75% quartiles, the whiskers are 1.5 the interquartile range, circles beyond the whiskers denote values > 1.5 times the interquartile range from the median (sometimes callled outliers).

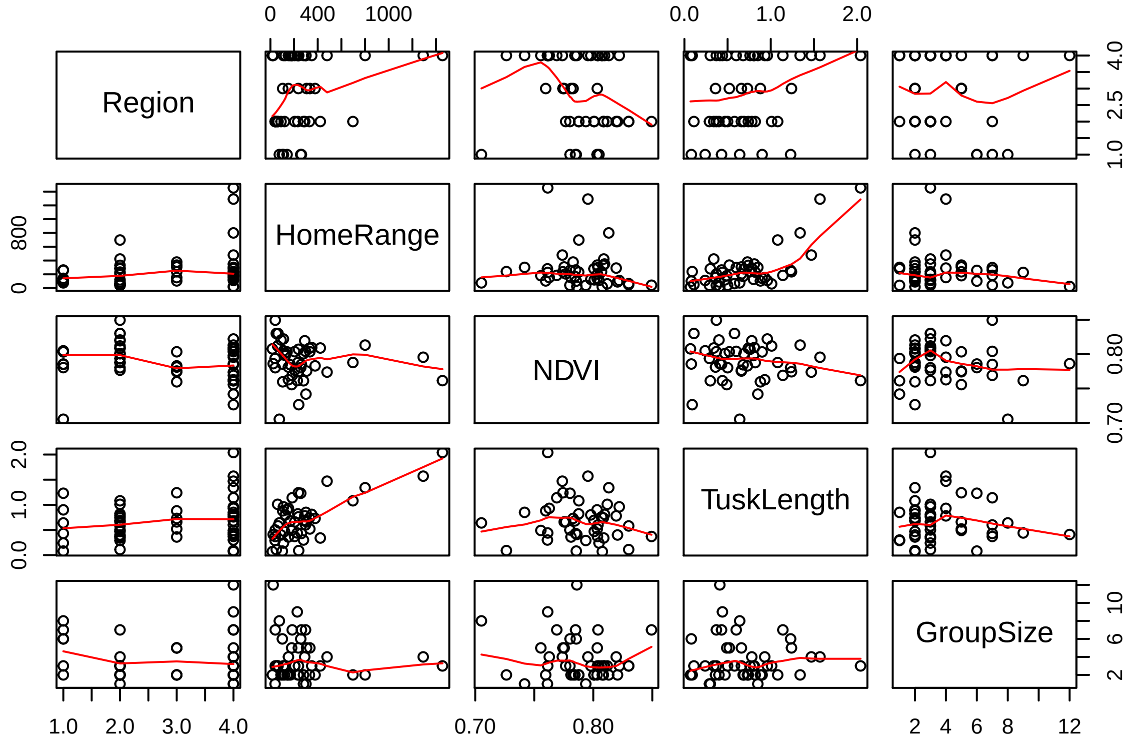
Use '?boxplot' to see all of the different boxplot cutomisation options.

```
?boxplot # use the help call
#wow there are lots of things to change!
```

4.3.1.3 Correlation plots

When analysing data sets with large numbers of continuous variables, it is often important to determine whether any of them are related to one another. Plotting multivariate data in excel is a nightmare, but it is very easy to do in R using the 'pairs()' function.

```
pairs(elephant[,3:7], # specifies the panels you wish to plot
      panel = panel.smooth) # adds a smoothed average line to
```



```
# help you understand the relationship
```

These plots take a while to get used to but are very useful! You can see home range and NDVI are positively correlated. Note - catagorical covariates are treated as numeric (look at region), and are therefore of limited use in these plots.

When we get into linear modelling we will see that we cannot include two variables in a model if they are correlated, as one of the assumptions of linear modelling is that all explanatory variables are independent. For this reason, correlation plots are very important.

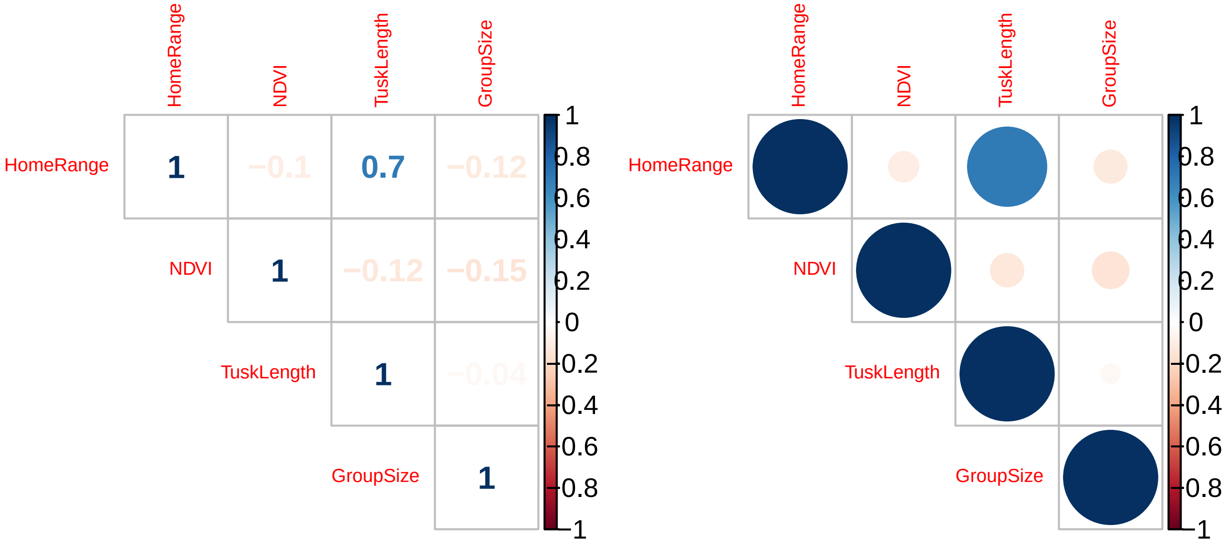
To explore this we can use the 'corrplot' package. In this plot, correlation coefficients are calculated and colored according to the degree of correlation (red=negative corellation, blue=positive correlation, white = uncorrelated).

```
library(corrplot) # Install corrplot

# First you need to determine the correlations between the variables
correlations <- cor(elephant[,4:7])

par(mfrow=c(1,2))
# make the plot - Number method
corrplot(correlations,
         type="upper",
         method="number")

corrplot(correlations,
         type="upper",
         method="circle")
```

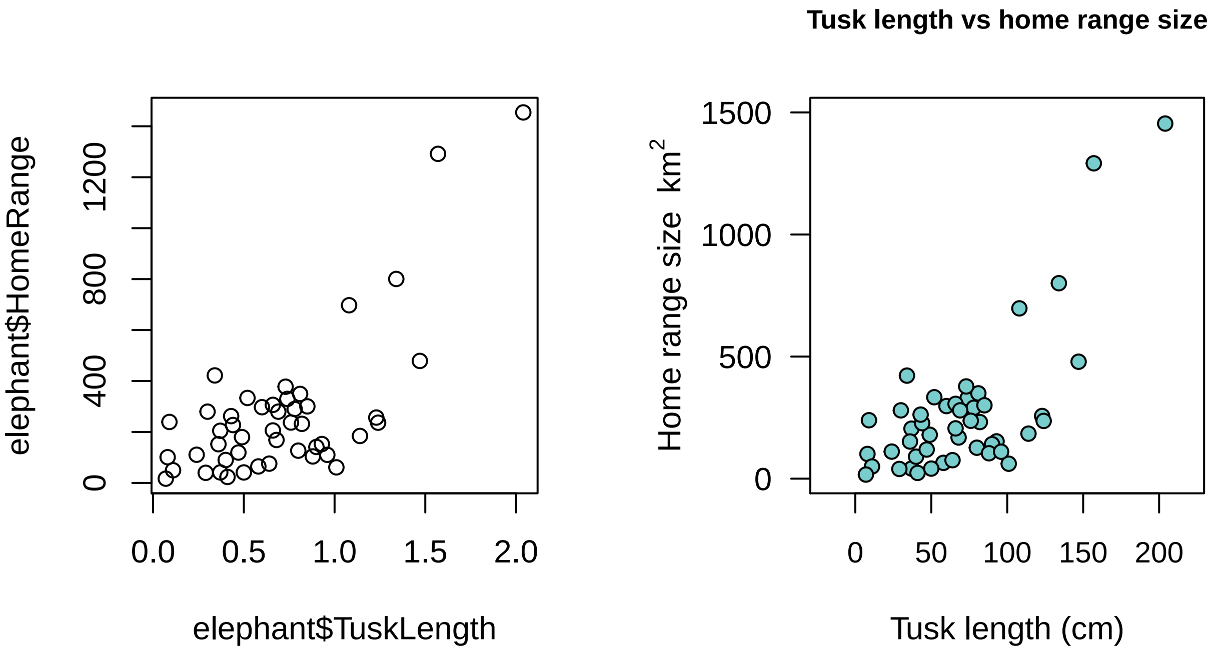


Checkout http://www.sthda.com/english/wiki/visualize-correlation-matrix-using-correlogram for more options!

4.3.1.4 Re-formatting plots like a pro

All the graphs presented so far are suitable for data exploration. If however, you would like to make them a little prettier (for publications and reports) you can change many of the default settings to get them exactly the way you want.

Lets revisit the default home range vs. tusk length plot (left), and compare it to a nicely formatted graph (right).



Lets break down the plot command step by step:

```
plot(elephant$HomeRange~elephant$TuskLength, # Your data
     main="Nice plot",                       # main = allows you to add a title
     pch=19,                                 # Changes the plotting symbols
     cex=1.2,                                # Changes the size of points
     col ="black"                            # The border colour
     bg="darkslategray3",                    # The fill colour
     ylab="Home range km2",                  # ylab = y axis label
     xlab="Tusk length (cm)",                # xlab controls the x axis label
     xaxt="n",                               # xaxt="n", suppresses the xaxis - so we can
                                             #                       plot a new one
     las=1,                                  # Rotates The Y axis labels

     ylim=c(0,1500),                         # Changes the y axis limits
     xlim=c(-0.2,2.2),                       # Changes the x-axis limits

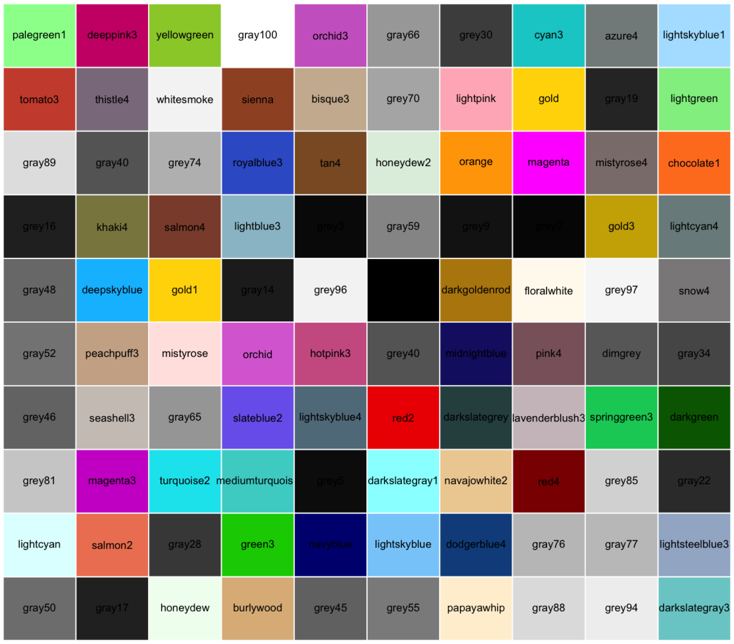
     )

axis(1, at=seq(0,2,0.5), labels=seq(0,200,50))# Adds back in the x-axis with the
                                              #  corrected labels
```

There is a lot going… lets break it down further!

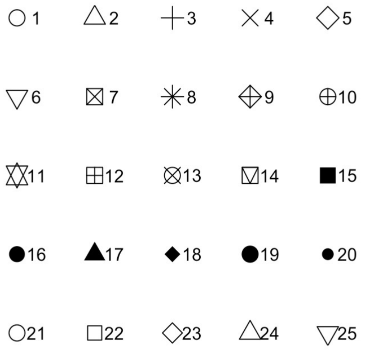
4.3.1.4.1 Colours and shapes

Most plotting functions have a color argument (usually `col()`) that allows you to specify the color of whatever your plotting. There are many ways to specify colors in R, but easiest way to specify a color is to enter its name as a string. For example `col = "red"` is R's red color. Of course, all the basic colors are there, but R also has tons of quirky colors like `"snow"`, `"papayawhip"` and `"lawngreen"`. There are some examples below:



I my personal fave is `"darkslategray3"`.

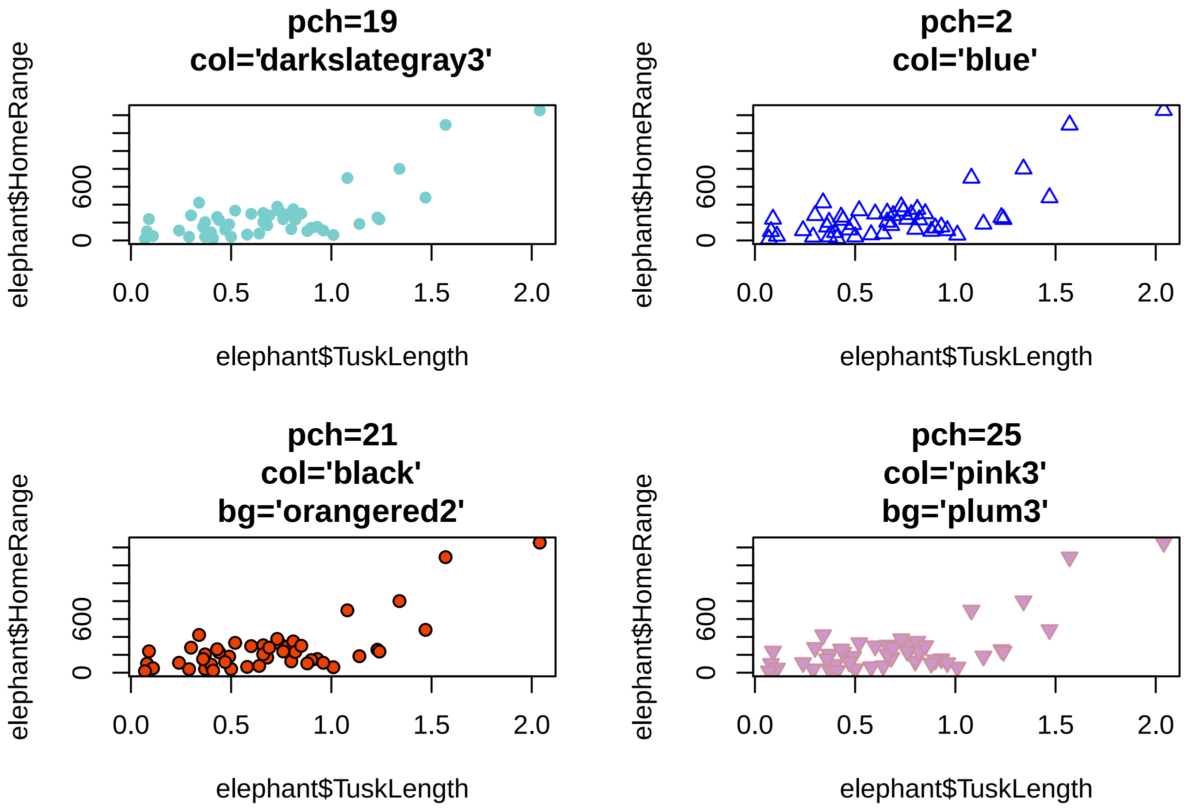
The way R controls plotting symbols is with the `pch=` command. There are a of different options to choose from depending on what you want to do.



My go to is `pch=19`. You get a set of nice filled circles... but you might be different! Play with some variation below. Try your own!

```
# Example code
plot(elephant$HomeRange~elephant$TuskLength,
     pch=16,
     col="darkslategray3")
```

TASK 9 Make your own customised graphs

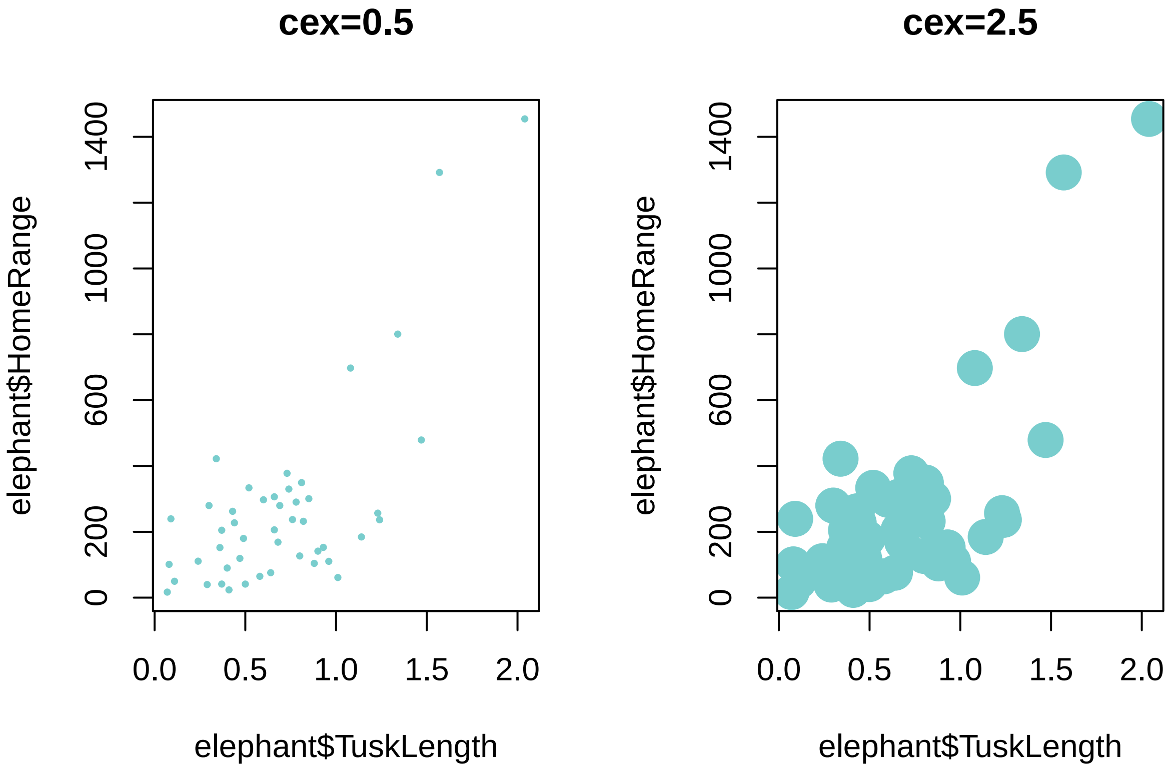


Do you want more control? Check out https://data.library.virginia.edu/setting-up-color-palettes-in-r/

4.3.1.5 Size

The size of the symbols is controlled by `cex=`. The size you choose is highly dependent on the the amount of data you have - the more data you have the smaller your symbols need to be.

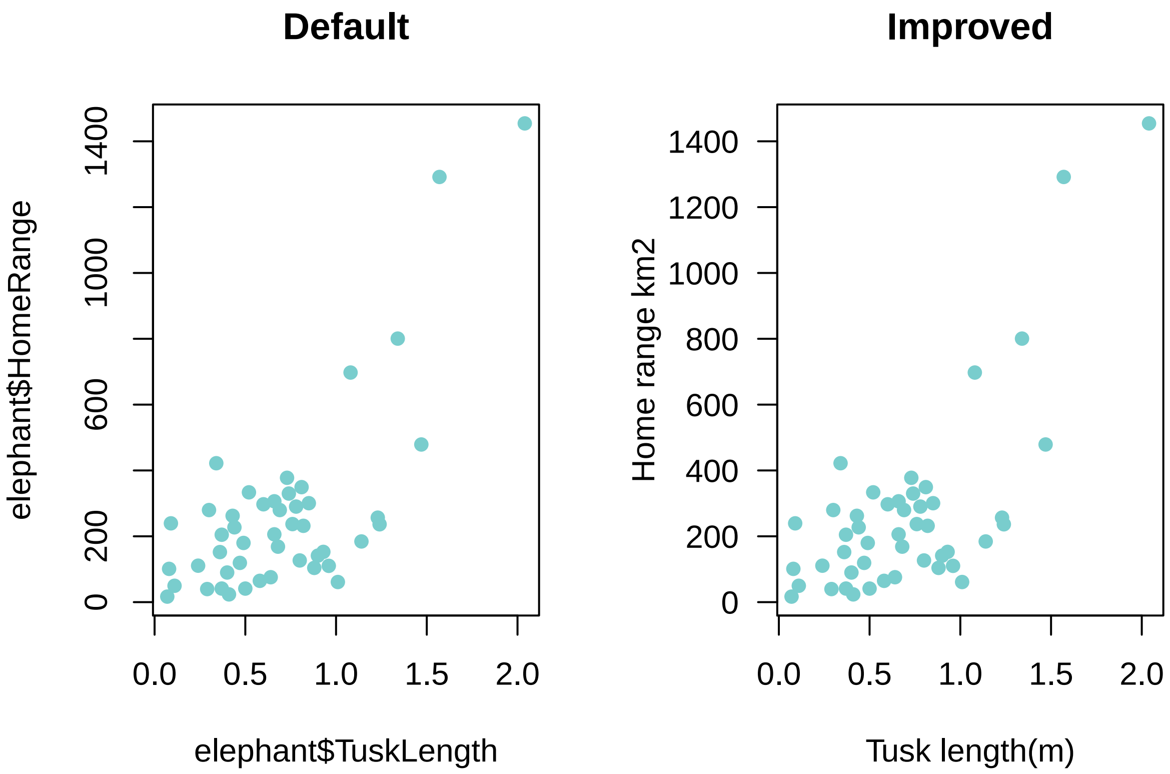
```
par(mfrow=c(1,2))
plot(elephant$HomeRange~elephant$TuskLength,pch=16,col="darkslategray3",
     cex=0.5, main="cex=0.5")
plot(elephant$HomeRange~elephant$TuskLength,pch=16,col="darkslategray3",
     cex=2.5, main="cex=2.5")
```



4.3.1.6 Labels and axes

The default plot in R leaves you with ugly axis labels. I use a few basic commands to improve these plots. The first of which is `las=1` which rotates the y-axis labels to horizontal. `ylab=` and `xlab=` always need changing to something sensible. `main=` can be used to add a plot title.

```r
par(mfrow=c(1,2))
plot(elephant$HomeRange~elephant$TuskLength,pch=16,col="darkslategray3", main= "Default")
plot(elephant$HomeRange~elephant$TuskLength,pch=16,col="darkslategray3", main= "Improved",
     las=1,
     xlab="Tusk length(m)",
     ylab="Home range km2" )
```



Finally, there are times when you might need to control the format of the axes themselves - such as where you place the 'tick marks' and what units the axes are in. In the initial example, I converted tusk length from meters in centimeters. Here is how:

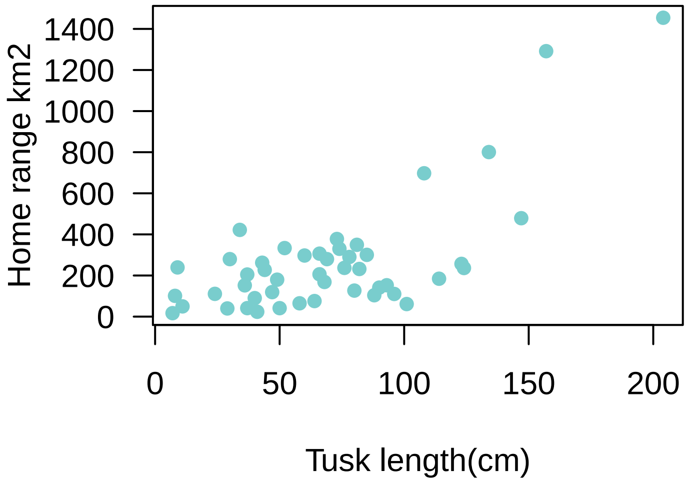
First you must supress the x-axis with `xaxt="n"`, then add back in the axis with the `axis()` function. The key commands it contains are:

`axis(side, at=, labels=)`.

Command	Description
side	an integer indicating the side of the graph to draw the axis (1=bottom, 2=left, 3=top, 4=right)
at	a numeric vector indicating where tic marks should be drawn
labels	a character vector of labels to be placed at the tickmarks (if NULL, the at values will be used)

I am editing the x axis (1), and I want the tick marks to be at 0, 0.5, 1, 1.5, 2 -> `seq(0,2,0.5)` and finally I want to change those to be labels of 0, 50, 100, 150, 200 -> `seq(0,200,50)`

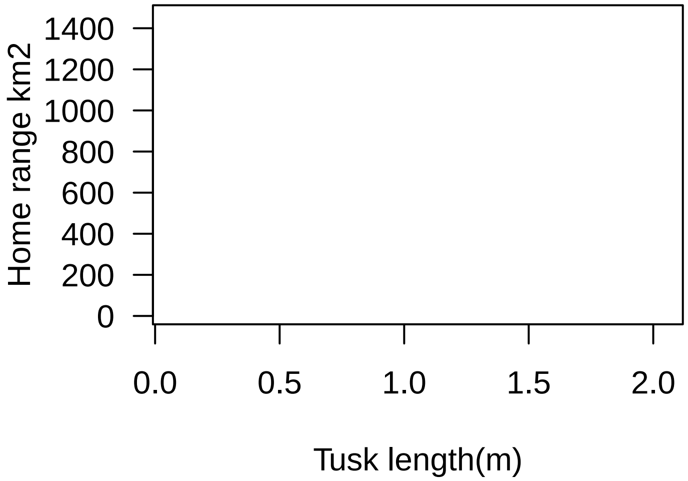
```r
plot(elephant$HomeRange~elephant$TuskLength,pch=16,col="darkslategray3",
     main= "", las=1, xlab="Tusk length(cm)", ylab="Home range km2",
     xaxt="n") # Supress the x!

axis(1,                        # x-axis
     at=seq(0,2,0.5),          # Where I want my ticks
     labels=seq(0,200,50))     # What I want the labels to be
```



4.3.1.7 Starting from scratch

Eventually you will want to make a plot so complicated you will have to start from scratch (a blank plot). To create a blank plot you should use the `type="n"` command in your plot call.

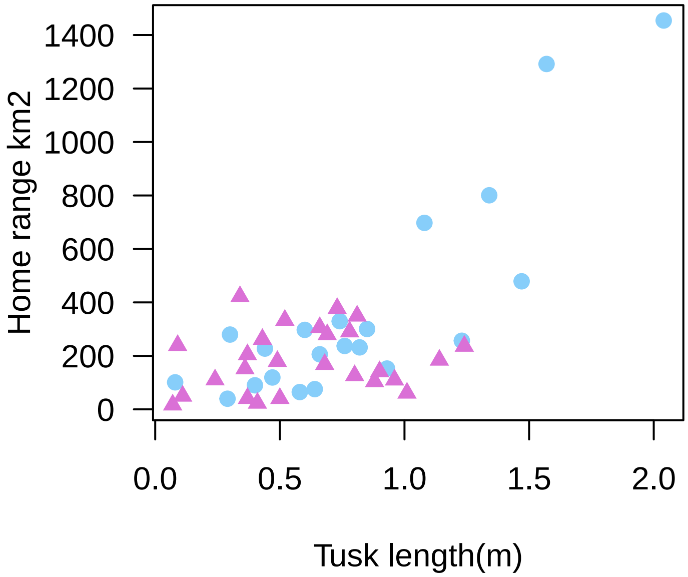
```
plot(elephant$HomeRange~elephant$TuskLength,
     pch=16,col="darkslategray3", las=1, xlab="Tusk length(m)", ylab="Home range km2",
     type="n")     # This makes the plot blank!
```



Scary huh!

Now you must use your newly aquired subsetting and plot colouring skills to add in the data points for males in blue, and females in red. You can add datapoints using the `points()` command.

```
# First, subset the data
males   <- elephant[elephant$Sex=="Male",] # Return all male rows
females <- elephant[elephant$Sex=="Female",] # return all female rows

# Add the males points
points(males$HomeRange, males$TuskLength,
       pch=19,
       col="lightskyblue")

points(females$HomeRange, females$TuskLength,
       pch=17,
       col="orchid")
```

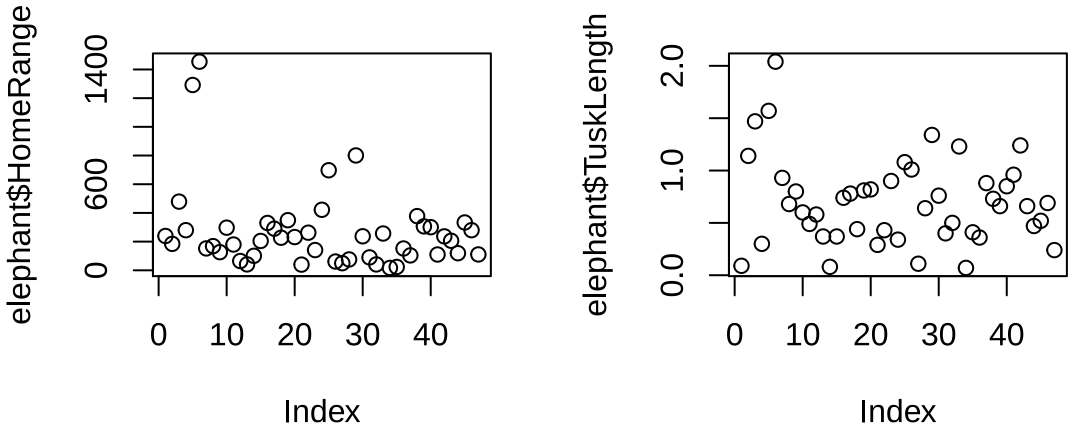


TASK 10 Repeat the above graph but change the point type based on elephant region

4.3.1.8 Multiple graphs

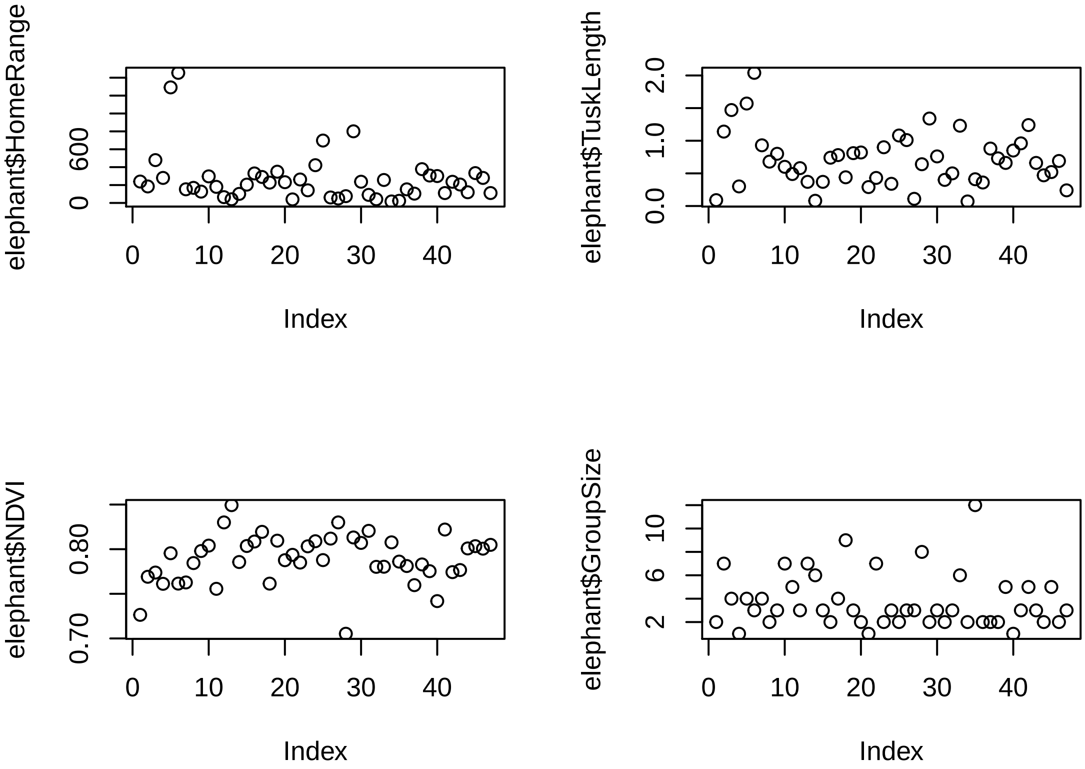
The key to publishable quality graphs is the ability to plot multiple graphs in one window. To start doing this you need to master the `par(mfrow=c(rows,columns))` command. With this method, you have to specify the number of rows and columns you would like. For example, to plot two graphs side by side then you would use:

```
par(mfrow=c(1,2)) #1 row, 2 columns
plot(elephant$HomeRange)
plot(elephant$TuskLength)
```



For a 2 x 2 block of graphs:

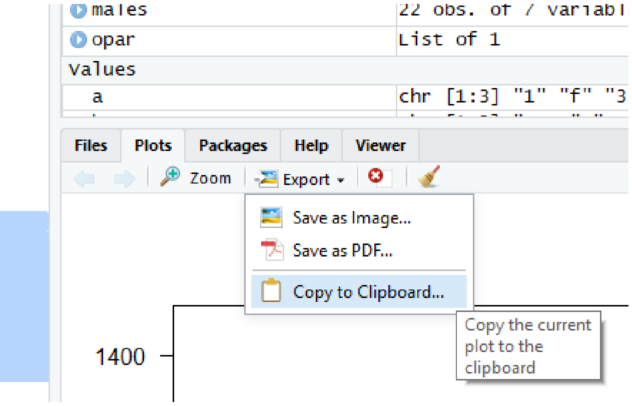
```
par(mfrow=c(2,2)) # Two rows two columns
plot(elephant$HomeRange)
plot(elephant$TuskLength)
plot(elephant$NDVI)
plot(elephant$GroupSize)
```



We have just breifly touched on the possibilites of changing the `mar()` command here. If you want to know more check out

4.3.2 Exporting Graphs

The easiest way to export a graph from R to word or powerpoint is using the 'Export' button in the 'Plots' tab.



For the most professional way of exporting graphs (using pre-defined dimensions) see https://www.datamentor.io/r-programming/saving-plot/

4.4 Merging columns

Sometimes you will want to merge the data from two columns, to make a new column - e.g. if you want to create a unique ID column from sites and plot numbers. Lets make a quick fake dataset and use the `paste0()` function.

```
# Make a data frame
newdat <- data.frame(Site=rep(c("A","B","C"), each=3), Plot=rep(1:3, times=3), Height=rnorm(9))

# Merge the site and the plot columns, separate them with a dot, and make a new "unique" column.
newdat$Unique <- paste0(newdat$Site,".", newdat$Plot)
head(newdat)
```

```
##   Site Plot    Height Unique
## 1    A    1 0.0211645    A.1
## 2    A    2 1.5676161    A.2
## 3    A    3 0.9619536    A.3
## 4    B    1 1.5489223    B.1
## 5    B    2 0.3027338    B.2
## 6    B    3 0.1558021    B.3
```

Chapter 5

Real world data

We have now been through the basics of R and R studio. Today we apply it to a real world example - a camera trapping dataset collected at Osa Conservation. You will be confronted with real issues involved in preparing, exploring and analysing ecological data in R.

5.1 Data organisation

The more time I spend doing research, the more I realise that the majority of the problems we have performing data analysis can be solved with good organisation. When you are building your own data analysis data frame you should stick to my **the 10 database commandments**:

5.1.1 The 10 database commandments

- **The survey data collected with the same protocol should never be split by site/person collecting it/month of the year.** Never ever split up datasets, it is a nightmare to fix. Have a column in the dataset that contains site ID/observer ID/Month instead.
- **One line = one statistical unit**. Do not pool data independent data into one line. In ecology, one statistical unit usually means one observation (i.e. spotting a species on a transect) - if you see two different species or two groups of the same species, use two lines. If you pool this data you going to waste time and losing important information.
- **Never encode catagorical variables as 0 or 1**. You will need a key to figure out what they mean in the future! Just use the actual levels - ie. yes/no or inside/outside (e.g. protected areas)
- **Your database should only include data - no empty filler cells to look nice!** Some people ‘pretty up’ their spreadhseet by including spaces… No thanks!
- **Headers should only span one row**. R can not handle more… listen to R.
- **Be consistent!** R treats the following as different: “Man”, “man”, " man“,”man ". This is a hassle to clean up!
- **Include units in the column headings**. There is nothing worse than characters included in a numerical column (e.g. 3 years) -> R will read it as a character string!
- **Don’t record coordinates in degrees, minutes, seconds.** Decimal degrees are king. We are in the digital age - get out of the middle ages!

- **Headings should be consise, but still understandable**. No-one likes to type elephant$NameOfTheElephant every time they want to plot a graph.
- **Never use commas when recording data!**

5.1.2 My database setup

I like to have four spreadhseets for each project: "data", "effort", "covariates", and "species info". Each sheet has at least one 'key' column to link it to data in other columns.

- **Data** is a database of the observations or captures of a species and the additional information you record when they are observed (e.g body size, group size etc).
- **Covariates** is a database containing the locations of your survey points, and any additional site-level information about those sites for the analysis (altitude, proximity to roads etc.)
- **Effort** is a database recording the sampling events. If you were doing observational transects you would record a unique id for each time you walked a transect. For camera trapping it would be each time you deployed a trap in the same location.
- **Species info** is the sheet to link species codes in the main database to data about the species (e.g. latin names)

5.2 Osa data

We will explore the OsaGrid dataset. I'll need your help to tell me basic summary information about this project and start the data exploration. I will (occasionally) give you some helpful hints. The idea is you apply some of the tools you learnt in the first two sessions to real world data!

The data are from a terrestrial camera trapping survey conducted in four different habitats: primary, secondary, plantation and agricultural land.

- OsaCTgrid_dat # Keys = 'Code' and 'Station'
- OsaCTgrid_covariates # Keys = 'Station'
- OsaCTgrid_effort # Keys = 'Station'
- OsaCTgrid_speciesinfo # Keys = 'Code'

5.3 Importing a dataframe into R from excel

The only time I **ever** use excel is to enter my data. After today, you will hopefully be the same! Luckily, somebody has already entered this data for you. Check out the OsaCTgrid files, three of them are in .csv format, one of them, OsaCTgrid_data, is an excel file. R doesnt like excel files. We need to convert it first.

Open the excel documents 'OsaCTgrid_data.xls'.

In excel, select File | Save as.. from the menu and navigate to the folder where you wish the file to be saved. Enter the file name (keep it the same for simplicity) in the 'File name:' dialogue box. In the 'Save as Type:' dialogue box click on the down arrow to open the drop down menu and select 'csv (Comma delimited)' as your file type. Select your R Project folder, the 'ClassData' folder and click Ok to save the file. Your file will now be saved as *OsaCTgrid_data.csv*.

This file can now be read directly into R using the `read.csv()` function.

```
data <- read.csv("ClassData/OsaCTgrid_data.csv", header =T) # header = T tells R
                                                            # that the data has column titles
```

TASK 11 Read in the other files. call them `effort`, `covariates` and `sp.info`

REMEMBER TO MAKE NOTES

5.4 Effort exploration

The first thing I like to do with a dataset is figure out how many surveys were performed. This information is stored in our `effort` datasheet.

TASK 12 Explore the structure of the `effort` sheet. What can you tell me about it? How many camera traps were working when collected ('Status' column)? What are the other categories? *Hint: use `table()`*

```
##   Station   date_set date_collected  Status
## 1   TCT01 13/05/2017     14/08/2017 working
## 2   TCT02 26/05/2017     14/08/2017 working
## 3   TCT03 18/05/2017     18/08/2017 working
## 4   TCT04 18/05/2017     16/08/2017 working
## 5   TCT05 18/05/2017     17/08/2017 working
## 6   TCT06   2/6/2017     17/08/2017 working
```

```
##
##      camera broken camera setup error     SD not working
##                  1                  1                  2
##            working
##                 56
```

5.4.1 Dealing with dates

We need to find out how many nights each camera was running. We have a start date (`date_set`) and an end date (`date_collected`). What is the easiest way to find out the effort?

R has an amazing `lubridate` package to make working with dates quick and easy. Read more about it here: https://lubridate.tidyverse.org/

Install the lubridate package

```
library(lubridate)
```

The way that lubridate works is it automatically detects the format of your date, as long as you tell it the order of the days, months and years. If your date object was 'y/m/d' you would use the `ymd()` command, if it is 'd/m/y' you use the `dmy()` command.

```
effort$start <- dmy(effort$date_set) # extract the date in d/m/y format
```

TASK 13 Repeat the same for the date collected, call it `effort$end`. Use the `head()` function to check it worked If it work it should look like this:

```
head(effort)
```

```
##   Station   date_set date_collected  Status      start        end
## 1   TCT01 13/05/2017     14/08/2017 working 2017-05-13 2017-08-14
## 2   TCT02 26/05/2017     14/08/2017 working 2017-05-26 2017-08-14
## 3   TCT03 18/05/2017     18/08/2017 working 2017-05-18 2017-08-18
## 4   TCT04 18/05/2017     16/08/2017 working 2017-05-18 2017-08-16
## 5   TCT05 18/05/2017     17/08/2017 working 2017-05-18 2017-08-17
## 6   TCT06   2/6/2017     17/08/2017 working 2017-06-02 2017-08-17
```

Now to calculate the number of days the camera was functioning, we use the `interval()` command. The interval command works as `interval(start, end)/desired.unit(1)`

```
interval(effort$start, effort$end)/days(1) # days
interval(effort$start, effort$end)/weeks(1) # weeks
interval(effort$start, effort$end)/years(1) # weeks
interval(effort$start, effort$end)/seconds(1) # seconds!
```

Great! The result in days is the most useful. Hang on… what are we forgetting?

TASK 13 Assign the result to a new column

TASK 14 What is the mean amount of time a camera was active? What is the maximum and the minimum? Hint: use summary()

```
summary(effort$days)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   33.00   92.75  106.00   99.27  108.00  110.00
```

5.4.2 Remove cameras that did not function

For this analysis we only want to analyse the camera stations which worked fully.

TASK 15 Remove the cameras which were not working by until the end of the project * Hint: use the [,]'s with a logic statement.

Check that your subset worked. You should now have only 56 cameras, all of which have "working" in the 'Status' column

```
table(effort$Status)
```

```
##
##       camera broken camera setup error     SD not working
##                   0                  0                  0
##             working
##                  56
```

```
# Whenever you remove factors it is a good idea to "reset the levels"

effort$Status <- factor(effort$Status)
```

TASK 14 Plot a boxplot of the amount of camera effort in the remaining camera stations *Hint: use boxpolot()*

5.5 Covariate data

We now need to see how the study is designed. The first thing to do is to explore the structure of the covariate dataframe. First, let's remove the cameras that did not work.

TASK 15 Subset the covariates data frame to only include camera stations in the updated effort dataframe. *Hint: Use [,] with a `%in%` logical statment.*

If your command worked, you should now only have 56 camera stations in you `covariates` file.

TASK 16 Explore the covariates database using `str()` What can you tell me about the covariate data frame? 1) How many columns do we have? 2) What format are the gps locations in? 3) How many cameras are located on/off trails? How many different habitats do we have?

```
## [1] 14
```

```
## [1] N08°24'36.1" N08°24'20.7" N08°25'14.7" N08°24'34.2" N08°24'37.4"
## [6] N08°24'48.4"
## 59 Levels: N08°23'47.8" N08°23'53.8" N08°24'00.2" ... N08°25'44.5"
```

```
##
## off  on
##  45  11
```

Now lets check out the habitat types

```
table(covariates$habitat_type)
```

```
##
## agricultural matrix agricultural_matrix          plantation
##                   1                   9                  10
##         plantation              primary            primary
##                   1                  17                   3
##           secondary          secondary
##                  11                   4
```

WHAT! 8 categories...! But you said that there were only 4!?!?! We have two different problems here! Agricultural matrix is spelt with and without an '_' and there are some blank spaces after several of the plantation, primary and secondary catagories. R sees all.

TASK 17 Enter the following code

```
# Dealing with the missing underscore - use logic to correct it

# Replace "agricultural matrix" with "agricultural_matrix"
covariates$habitat_type[covariates$habitat_type=="agricultural matrix"] <-
                                                                   "agricultural_matrix"

# White space issues are very common,
# there is a little bit of code to deal with it trimws()!
covariates$habitat_type <- trimws(covariates$habitat_type)

# Check that it worked
table(covariates$habitat_type)
```

```
##
## agricultural_matrix          plantation              primary
##                  10                  11                   20
##           secondary
##                  15
```

```
# Hooray!
```

5.5.1 Plotting your survey sites - Substr()

Plotting GPS coordinates in R is exactly the same as doing a normal scatter plot. However, the first thing you need to do is ensure that your data are in decimal degrees. Unfornuately ours are in degrees, minutes seconds.

The calculation to convert DMS to decimal degress is relatively simple:

Decimal Degrees = (Seconds/3600) + (Minutes/60) + Degrees.

However to extract the data from our colums we will have to master the substring command `substr()`. First extract the first row of the `covariates$latitude` column and examine its structure.

```
covariates$latitude[1]
```

```
## [1] N08°24'36.1"
## 59 Levels: N08°23'47.8" N08°23'53.8" N08°24'00.2" ... N08°25'44.5"
```

If you count the number of latters from the first, the degree data are stored in the 2nd and 3rd characters, minutes in the 5th and 6th, and seconds data in the 8th-11th slots. `substr()` can extract these values `substr(data, start position, end position)`.

```
# example of substring
substr(covariates$latitude[1], 2,3) # extract the second and third values (degrees)
```

```
## [1] "08"
```

TASK 18 What is the problem? The number is actually a character string as it is surrounded by "" quatation marks. ***Convert it to numeric***

Now we just plug the vlues into the equation (remembering to convert the output to `as.numeric`).

Enter the following

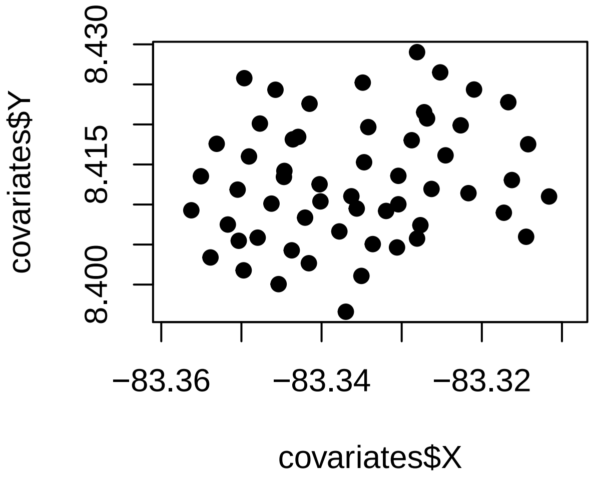
```
covariates$Y = (as.numeric(substr(covariates$latitude, 8,11)) / 3600) + # seconds +
               (as.numeric(substr(covariates$latitude, 5,6 ))  / 60) +  # minutes +
                as.numeric(substr(covariates$latitude, 2,3 ))           # degrees
```

TASK 19 Do the same for the longitude data. assign it to column `X` *HINT: Be careful, the character positions change! And multiple by -1 as it is a "west" coordinate.*

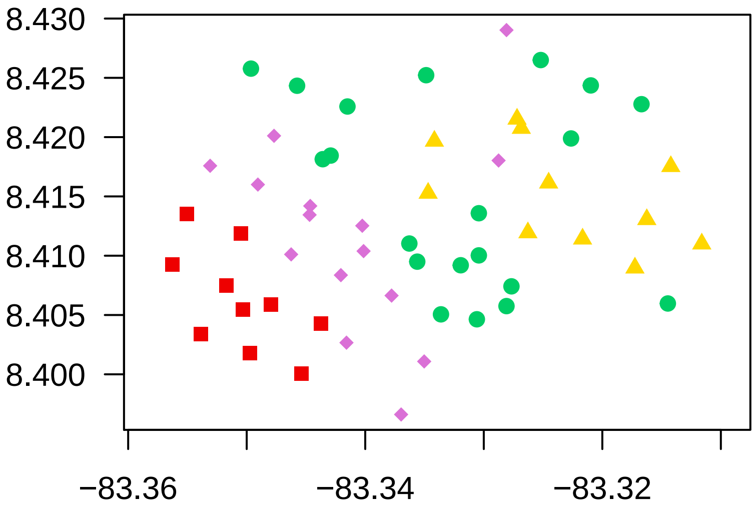
```
## [1] W083°19'49.5"
## 59 Levels: W083°18'41.8" W083°18'51.2" W083°18'52.1" ... W083°21.11.1
```

Now we can do a standard scatter plot of our data. . . . with one difference. Whenever you map things in R you should lock the aspect ratio to 1 (`asp=1`) or the map will look strange!

```
plot(covariates$Y~covariates$X, pch=19,asp=1)
```



TASK 20 Smarten up your graph! Try adding separate shapes for each different habitat *Hint: start with a blank plot (`type="n"`), then sequentially add the points for the different habitats using `points()` and the logic statement covariates$Habitat_type==*
Try to make something like this:



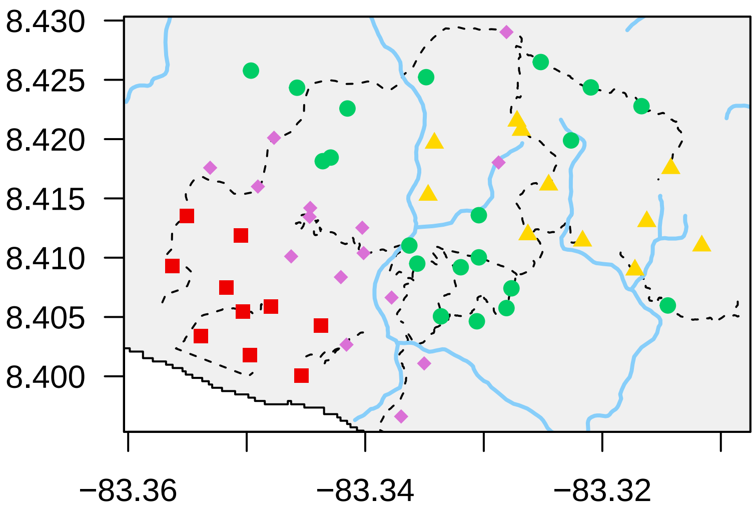
It is relatively simple to add shape files to plots like these using the "Simple Features" package. All the clever ways to use the Simple Features would be a couse in itself. Now lets just use it to make a nice graph.

The first step is to load the package and the shapefiles

```
library(sf)
trails    <- st_read("ClassData/Map Files/Trails.shp", stringsAsFactors = F)
roads     <- st_read("ClassData/Map Files/Roads.shp", stringsAsFactors = F)
rivers    <- st_read("ClassData/Map Files/Rivers.shp", stringsAsFactors = F)
country   <- st_read("ClassData/Map Files/Country.shp", stringsAsFactors = F)
```

When making maps, it is best to start the plot with the survey sites, then add the layers on top. This is the esiest way to ensure that the plot window is appropriately sized.

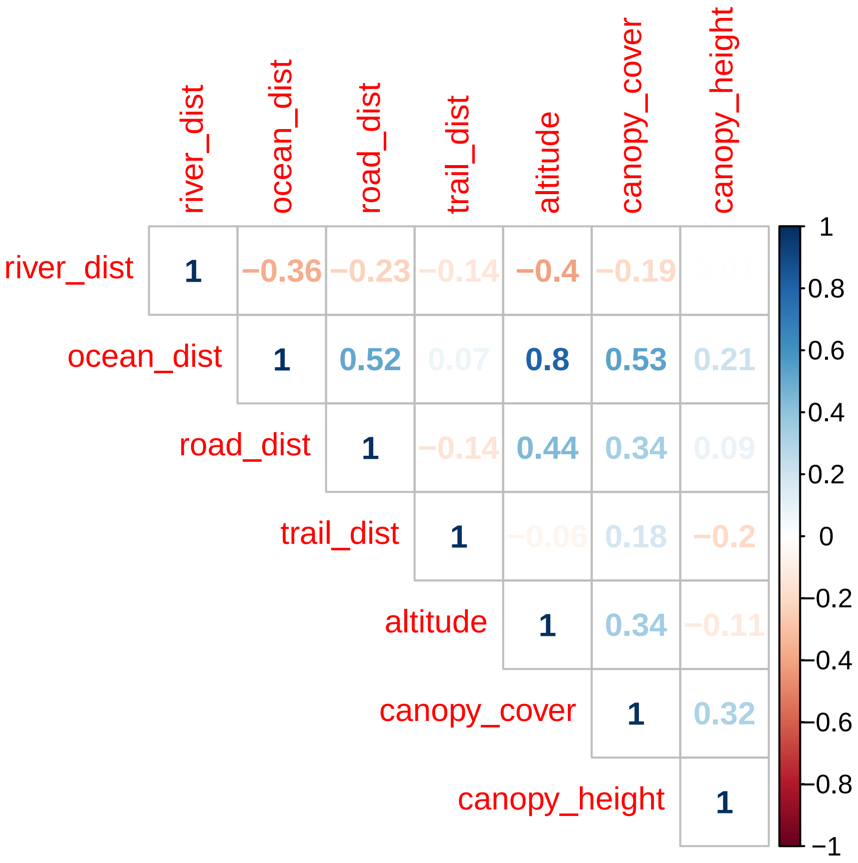
To plot Simple Features shapefiles you need to specify that you want to plot the geometry of the object: st_geometry. See below:



5.5.2 Explore your covariates

We have 7 covariates in this data which could explain some variation in mammal abundance detected on camera traps. We should explore them.

TASK 21 *** How are the covariates river dist, ocean_dist, road_dist, altitude, canopy_cover, and canopy_height related to one another?*** *Hint: Use the `corrplot` package to calculate the covariance between the explanatory variables.* What do you think about the correlations? Are we worrid about anything?



5.6 Camera trap data

We will now clean and look at the camera trap data.

TASK 22 Remove the data from the cameras that did not work until the end

*** Remove the data from the cameras which did not survive until the end of the study.***

TASK 23* Have a look at the structure of the `data` dataframe and tell me what you see. How many observations do we have in the camera trap dataframe? How many different species classifications do we have? *Hint* use `nrow()` to find out how many rows are in `data`.

A useful comman to know is `as.dataframe()` as it can turn a table output, into a more easily observable dataframe.

TASK 24 Run the following code

```
det.freq <- as.data.frame(table(data$Code)) # Make a table into a dataframe
```

You dataframe should look like this (but longer):

```
##                     Var1 Freq
## 1                 agouti 2490
## 2                 Agouti   20
## 3              ants_army    1
## 4  armadillo_ninebanded  142
## 5                  bike    6
## 6        bird_large_uid    1
## 7        bird_small_uid  635
## 8                broken   10
## 9   capuchin_whitefaced   17
## 10     caracara_crested    1
## 11                coati  335
## 12                Coati    0
```

Doing this we can see that we have some issues. Agouti appears twice ("agouti" and "Agouti"), and coati appears twice as ("Coati" and "coati"). Luckily there is a nice command to deal with this `tolower()`!

TASK 25 Convert the errors to lowercase

```
data$Code <- tolower(data$Code)
```

```
# Always check you code has worked!
as.data.frame(table(data$Code))[1:10,]
length(unique(data$Code)) # 53 species classification
```

5.6.1 Subsetting you data to only include species of interest.

As you have seen, we have 54 different species classifications, but some of those are birds, humans, even 'bike'. This is where the `sp.info` dataframe comes in.

TASK 26 Explore the structure of the sp.info dataframe

```
##                   Code Mammal Native              Species
## 1               agouti    yes    yes  Dasyprocta punctata
## 2            ants_army     no    yes                 <NA>
## 3 armadillo_ninebanded    yes    yes Dasypus novemcinctus
## 4                 bike     no     no                 <NA>
## 5       bird_large_uid     no    yes                 <NA>
## 6       bird_small_uid     no    yes                 <NA>
```

```
##  [1] "agouti"                   "bird_small_uid"
##  [3] "capuchin_whitefaced"      "coati"
##  [5] "heron_barethroatedtiger" "nothing"
##  [7] "opossum_foureyed"         "opossum_uid"
##  [9] "opossum_water"            "paca"
## [11] "people"                   "raccoon"
## [13] "setup"                    "tamandua"
## [15] "uid"                      "curassow_great"
## [17] "grison"                   "horse"
## [19] "mammal_small_uid"         "opossum_common"
## [21] "peccary_collared"         "puma"
## [23] "skunk_striped"            "woodrail_greynecked"
## [25] "bird_large_uid"           "hawk_roadside"
## [27] "armadillo_ninebanded"     "tinamou_great"
## [29] "cow"                      "egret_cattle"
## [31] "ocelot"                   "squirrel_uid"
## [33] "tapir"                    "caracara_crested"
## [35] "ibis_white"               "vulture_black"
## [37] "lizard_uid"              "jaguarundi"
## [39] "tayra"                   "dog"
## [41] "iguana_green"            "ants_army"
## [43] "peccary_whitelipped"     "tinamou_little"
## [45] "margay"                  "motmot_bluecrowned"
## [47] "jaguar"                  "guan_crested"
## [49] "forestfalcon_collared"   "broken"
## [51] "scorpion_bark"           "bike"
## [53] "horse_ridden"
```

For this analysis, we want to subset the data to just mammals, which are native, and can be identified to genus level.

TASK 27 Write a line of code which allows us to subset `sp.info` to a new dataframe called `focal.info` based off the above criteria. *Hint: use `[,]` and multiple logic statements linked by the '&' sign. Remember, to deal with NA's you will need to do the 'is.na()' command.*

```
## [1] 21
```

If you have performed your operation correctly, your sp.focal dataset should contain 21 species.

*** Now subset the `data` dataframe to only contain the species in your '`focal.info` database. Assign it to a new dataset - final.data ***

```
##                              Name Station                Code
## 47 2017-07-02 12.55.24_00047.AVI    CT02              agouti
## 48 2017-08-11 09.07.34_00048.AVI    CT02              agouti
## 49 2017-08-13 09.25.20_00049.AVI    CT02              agouti
## 50 2017-08-13 13.01.38_00050.AVI    CT02              agouti
## 51 2017-08-14 08.57.04_00051.AVI    CT02              agouti
## 53 2017-06-23 13.28.04_00053.AVI    CT02 capuchin_whitefaced
```

5.6.2 Extracting time and date information

When ever you have observatiosn you should record the times and dates. We dont currently have that information in the `final.data` dataframe.

In the `effort` sheet we extracted date information using `lubridate()`. We can do exactly the same thing here as the dates and times are embedded in the filenames column `final.data$Name`.

```r
final.data$Name[1]
```

```
## [1] 2017-07-02 12.55.24_00047.AVI
## 11906 Levels: 2017-04-03 14.38.28_01513.AVI ... 2017-08-20 06.52.14_12082.AVI
```

Examining the first element in the dataframe you can see that the first 19 characters contain the the date and time information in the format 'Y-m-d H.M.S'.

TASK 28 *** Write a command to extract the first 19 characters from `final.data$Name`. *Hint: use substr()*

Next, convert that object to an R date column called `final.data$Date` using lubridate - `ymd_hms()`

If you have run the code correctly the top of you dataset should now look like this:

```
head(final.data)
```

```
##                                Name Station               Code
## 47 2017-07-02 12.55.24_00047.AVI    CT02              agouti
## 48 2017-08-11 09.07.34_00048.AVI    CT02              agouti
## 49 2017-08-13 09.25.20_00049.AVI    CT02              agouti
## 50 2017-08-13 13.01.38_00050.AVI    CT02              agouti
## 51 2017-08-14 08.57.04_00051.AVI    CT02              agouti
## 53 2017-06-23 13.28.04_00053.AVI    CT02 capuchin_whitefaced
##                    Date
## 47 2017-07-02 12:55:24
## 48 2017-08-11 09:07:34
## 49 2017-08-13 09:25:20
## 50 2017-08-13 13:01:38
## 51 2017-08-14 08:57:04
## 53 2017-06-23 13:28:04
```

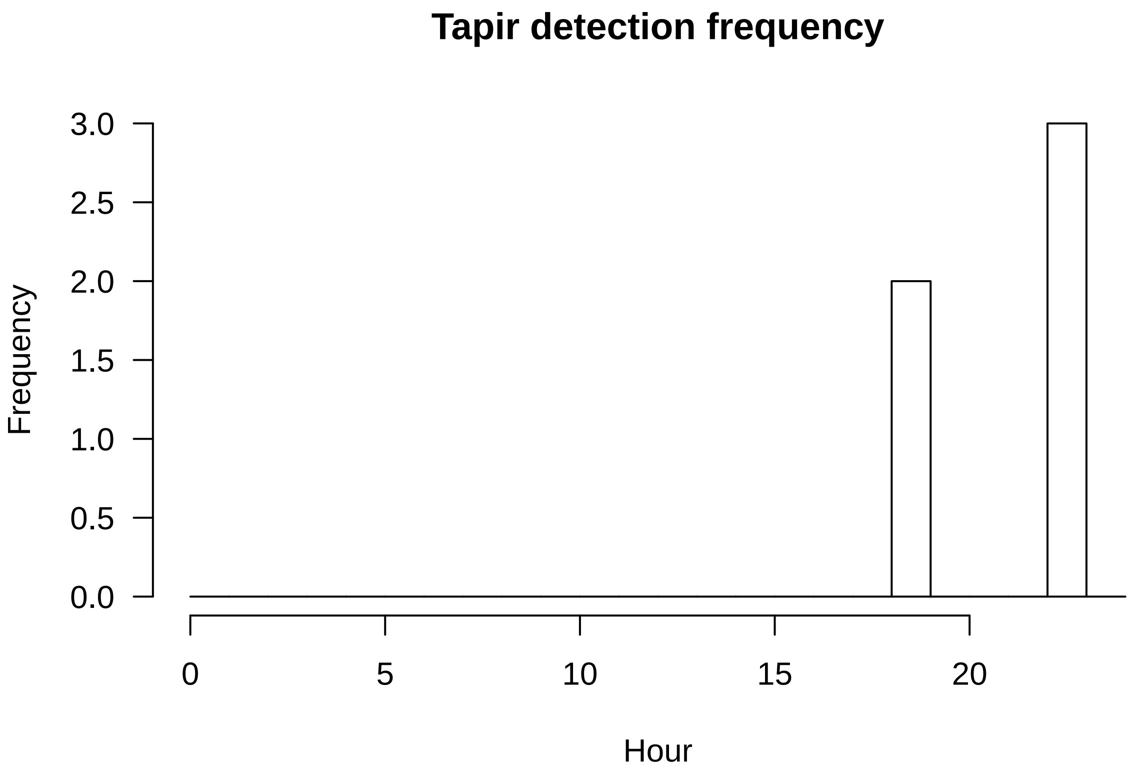
We can use this information to explore animal activity patterns. To do that we need to extract the an “hour” column from the `final.data$Date` information. Lubridate makes this nice and easy.

```
final.data$Hour <- hour(final.data$Date) + minute(final.data$Date)/60
```

TASK 29 Plot a histogram of of the times recorded in the hour category and specify hourly breaks. What can you tell me about when animals are usually captured? *Hint: use* ***`hist`*** *for the plot and* ***`seq()`*** *to determine the breaks.*

TASK 30 *** Repeat this graph using the data for two species of your choice. Try a diurnal mammal and a nocturnal mammal (paca, possum_foureyed. Share your graphs with the class** Hint: subset the data using the [,] and a logic statement e.g. final.data$Hour[final.data$Code==“tapir”]. Remember to change the title! Edit the code below.

```
hist(final.data$Hour[final.data$Code=="tapir"],  # Specify the data
     breaks=seq(0,24,1),                         # Specify the breaks
     main = "Tapir detection frequency", # change the title
     las=1,                                      # Rotate the y axis labels (essential!!!!)
     xlab="Hour")                                # Label the x-axis
```



5.7 Building analysis dataframes

So far we have explored our four different datasheets, but now we need to build the final analysis dataframes. To do that we need to decide what we want to study, and then define what out statistical unit is.

TASK 31 Come up with some questions to ask

5.7.1 Relative abundance

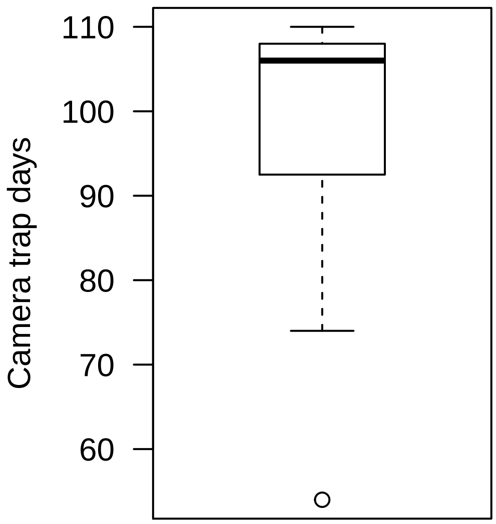
we could hypothesise that the more degraded a habitat is, the fewer animals you will find there. We have four habitat types so we might think that:

Primary > Secondary > Plantation > agricultural_matrix

As we are interested in the abundance of mammals in the different habitats, I could just add up all of the detections of a given species in a habitat, and compare those between habitat types. ***Is there a problem with that?***

Remember the camera effort boxplot from earlier:

```
boxplot(effort$days,ylab="Camera trap days", las=1)
```



```
summary(effort$days)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   54.00   92.75  106.00   99.91  108.00  110.00
```

Some cameras were active for 110 days, th rest for less. If we were to just add up the number of captures, the cameras with half the amount of effort will have half as many captures, regardless of that habitat they are in.

Consequently, we need to standardise our counts to **relative abundance** of mammals: the number of detections per unit time (usually 100 camera trap days). Fortunately, this is relatively simple:

For each camera -> sum(all detections) / effort.days * 100

The steps we need to take are:

1) Summarise the `data` file to count every observation and store it in a new dataframe
2) Merge the count data with the `effort` file - so we can calculate the relative abundance index
3) Merge the new count data file with the `covariates` file so we can explore how our covariates influence mammal abundance.

In this case the "statistical unit" we are interested in (remember point 2 of the 9 commandments) is the camera. So we want to summarise this data so that one row = one camera station.

5.7.2 Step 1: Summarise

There are some very simple functions in the `dplyr` to summarise data. the `summarize()` function is a good one to know.

The `summarize()` function works by summarize(group_by(data, group 1, group 2 etc), "column name" = the action you want to perform)

We will produce the summary for each species:

```
library(reshape2)
final.data$Code<- factor(final.data$Code)

count.sp <- melt(table(final.data$Station, final.data$Code))
colnames(count.sp) <- c("Station", "Code", "Count")
head(count.sp)
```

```
##   Station    Code Count
## 1    CT02 agouti     5
## 2    CT03 agouti     1
## 3    CT04 agouti     1
## 4    CT06 agouti    45
## 5    CT07 agouti   154
## 6    CT08 agouti     0
```

If you performed this operation correctly it should look like the following:

```
head(count.sp)
```

```
##   Station    Code Count
## 1    CT02 agouti     5
## 2    CT03 agouti     1
## 3    CT04 agouti     1
## 4    CT06 agouti    45
## 5    CT07 agouti   154
## 6    CT08 agouti     0
```

5.7.3 Step 2: Merge with effort

Next we need to merge our data with the effort data. To merge with the effort sheet we can use the `Station` key and the `left_join()` command. We dont want to use all the columns in the `effort` sheets as things will get messy. Lets subset it first:

```
# First subset the effort file to just the columns you want to merge - 'Station' and 'days'
effort[, c("Station", "days")] # literally, give me all rows,
#but just the station and days columns

# Join your reduced data frame with the count data
count.sp <- left_join(count.sp, effort[, c("Station", "days")], by="Station")
```

```
##    Station days
## 1    TCT01   93
## 2    TCT02   80
## 3    TCT03   92
## 4    TCT04   90
## 5    TCT05   91
## 6    TCT06   76
## 7    TCT07   75
## 8    TCT08   74
## 9    TCT09   76
## 11    CT02  108
```

```
## Warning: Column `Station` joining factors with different levels, coercing
## to character vector
```

Your final job is to calculate the **R**elative **A**bundance **I**ndex or **RAI**.

For `count.sp`, we need to divide the counts by the number of trapping days and times it by 100 (to get the number of captures per 100 days).

```
count.sp$RAI <- count.sp$Count/count.sp$days * 100
# all of those decimal places are annoying... remove them using round()
count.sp$RAI <- round(count.sp$RAI,2)
```

5.7.4 Step 3: Merge the covariates

So we now have a dataset on relative abundance index which we want to analyse, but we dont have any information about the sites themselves. We need to add the covariate data. To use column names on all of these would take a long time, we can use numbers instead. The columns we want are: 1 = station, then 4:12 (the covariates: habitat_type:canopy height).

```
# Check the covariate subset
covariates[,c(1,4:12)] # I have all the columns I want
```

TASK 35 Perform the `left_join` but with covariate data.

```
## Warning: Column `Station` joining character vector and factor, coercing
## into character vector
```

All being well your dataset should now look like this:

```
head(count.sp)
```

```
##   Station    Code Count days    RAI         habitat_type trail river_dist
## 1    CT02 agouti     5  108   4.63 agricultural_matrix   off       1631
## 2    CT03 agouti     1  110   0.91 agricultural_matrix   off        644
## 3    CT04 agouti     1  110   0.91 agricultural_matrix   off       1148
## 4    CT06 agouti    45   85  52.94           secondary   off        219
## 5    CT07 agouti   154  108 142.59 agricultural_matrix   off       1589
## 6    CT08 agouti     0  106   0.00 agricultural_matrix   off       1378
##   ocean_dist road_dist trail_dist altitude canopy_cover canopy_height
## 1        252       582        104       12        70.25        10.435
## 2        201        96        160        1        22.90         5.697
## 3        230       133        165        6         6.44         9.953
## 4        135       499        116       14        38.71         7.239
## 5        787       767         46       13        15.62         9.746
## 6        760       256        194       19        20.41         7.065
##          Y         X
## 1 8.403389 -83.35386
## 2 8.400056 -83.34536
## 3 8.401778 -83.34972
## 4 8.396611 -83.33697
## 5 8.409278 -83.35625
## 6 8.407500 -83.35169
```

We will also make a site*species matrix so we can explore relationships between multiple different species.

```
count.mat <- dcast(count.sp, Station  ~ Code,value.var ="Count", fun.aggregate=sum, drop=FALSE)
```

Save the count.sp dataset for later.

```
write.csv(count.sp, "SpeciesRAIdata.csv", row.names=F)
write.csv(count.mat, "SpeciesMATRIXdata.csv", row.names=F)
```

We are now ready for some proper data exploration

5.8 Real data exploration

Team Task I have never analysed this data, I have no idea what it will tell us.

Break into groups and summarise the data in these datasets.

Questions:

- What are the most commonly detected species?

Hints: - Scatterplots of continuous variables with RAI (`plot()`) - Boxplots of catagorical variables with RAI (`boxplot()`) - Boxplots of catagorical covariates with Habitat_type (`boxplot`)

Chapter 6

Introduction to Modelling

Make a new R script called - "Introduction to Modelling", give it a title and import the data we will use.

```r
# Title: Linear modelling in R

# Read in the covariates dataframe
covariates <- read.csv("ClassData/OsaCTgrid_covariates.csv", header =T)
```

6.1 Is canopy height related to habitat type at Osa?

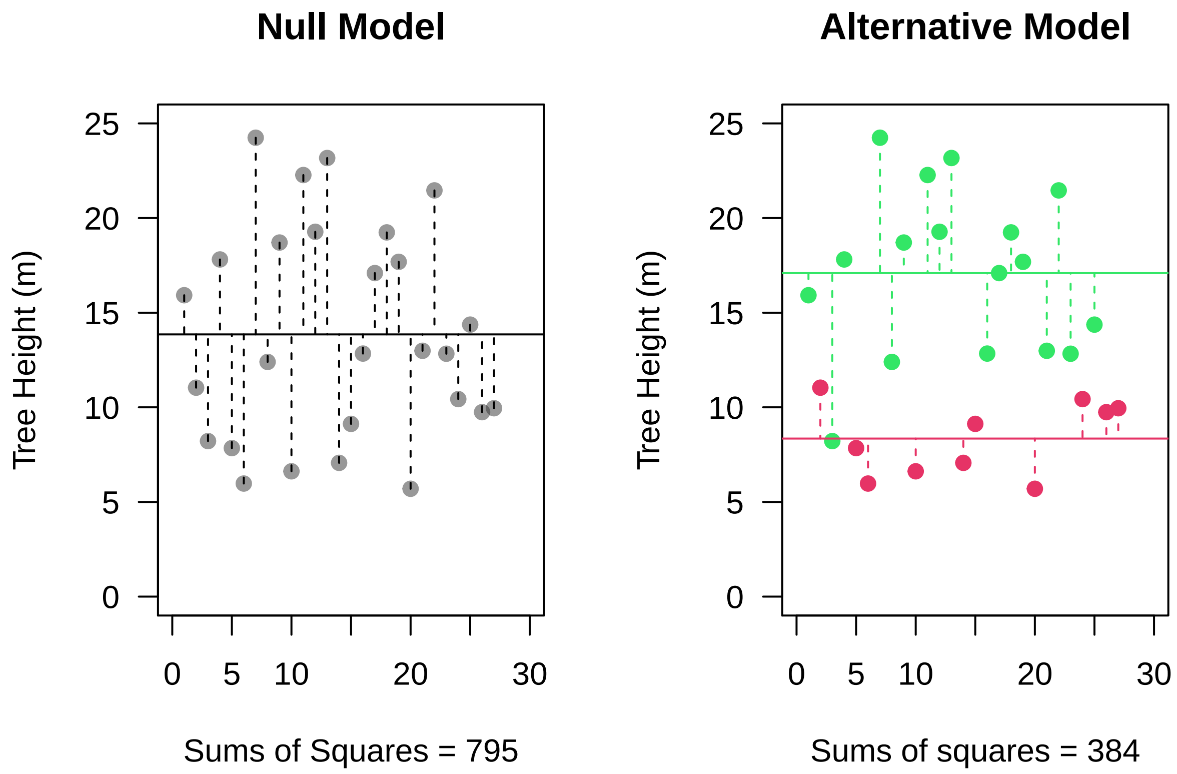
Subset the 'covariate' data to just "primary" and "habitat_type":

```r
# Subset the data to just primary and agricultural_matric habitats
hdat <- covariates[covariates$habitat_type=="primary" |
                     covariates$habitat_type=="agricultural_matrix", ]

# Reset the factor levels:
hdat$habitat_type <- factor(hdat$habitat_type)
```

6.1.1 What is a model

When we are "modelling" we are fitting functions to the data in order to explain them. A good model will always explain more information than a bad model, but will not be so complicated that we cannot interperate it. *Remember, we are not trying to recreate reality, we are trying to explain reality.*

On the left is an example of a null model, canopy height does not vary by habitat type (a single intercept), and on the right is a model where the intercept varies by habitat type (an alternate model - more on that later). We determine how much variation a model explains by measuring the 'sums of squares' - the squared distance (dashed lines) from our model (line) to the actual data (points). The more complicated model has a lower sums of squares, so it explains more information!



6.1.2 The P-value

The problem with just looking at sums of squares is that the more complicated a model is... the more variation it explains. So instead we sometimes use p-values to compare the two alternative models. P-values measure the probability the difference between the null model and the alternative model came about by chance, but only after paying a penalty for how complex the models are (via degrees of freedom - don't worry about this right now).

Lets run a linear model and see what we get:

```
mod1 <- lm(canopy_height~habitat_type,data=hdat)
summary(mod1)
```

```
##
## Call:
## lm(formula = canopy_height ~ habitat_type, data = hdat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -8.8738 -2.5147  0.5972  2.1187  7.1542
##
## Coefficients:
##                     Estimate Std. Error t value Pr(>|t|)
## (Intercept)            8.350      1.173   7.119 1.84e-07 ***
## habitat_typeprimary    8.740      1.478   5.913 3.59e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3.709 on 25 degrees of freedom
## Multiple R-squared:  0.5831, Adjusted R-squared:  0.5664
## F-statistic: 34.97 on 1 and 25 DF,  p-value: 3.592e-06
```

```
aov(mod1)
```

```
## Call:
##    aov(formula = mod1)
##
## Terms:
##                 habitat_type Residuals
## Sum of Squares      480.9726  343.8819
## Deg. of Freedom            1        25
##
## Residual standard error: 3.70881
## Estimated effects may be unbalanced
```

If you examine the P-value in the table for the `habitat_typeprimary` category you will see that it is very small, which suggests that the two habitats do indeed have different canopy heights! Great!

Next we need to check if our models fit the assumptions of a liner model, those assumptions are:

- **Residuals are independent of each other** Each sample sould be an independent from all the other samples (the result of one should not influence the result of another).

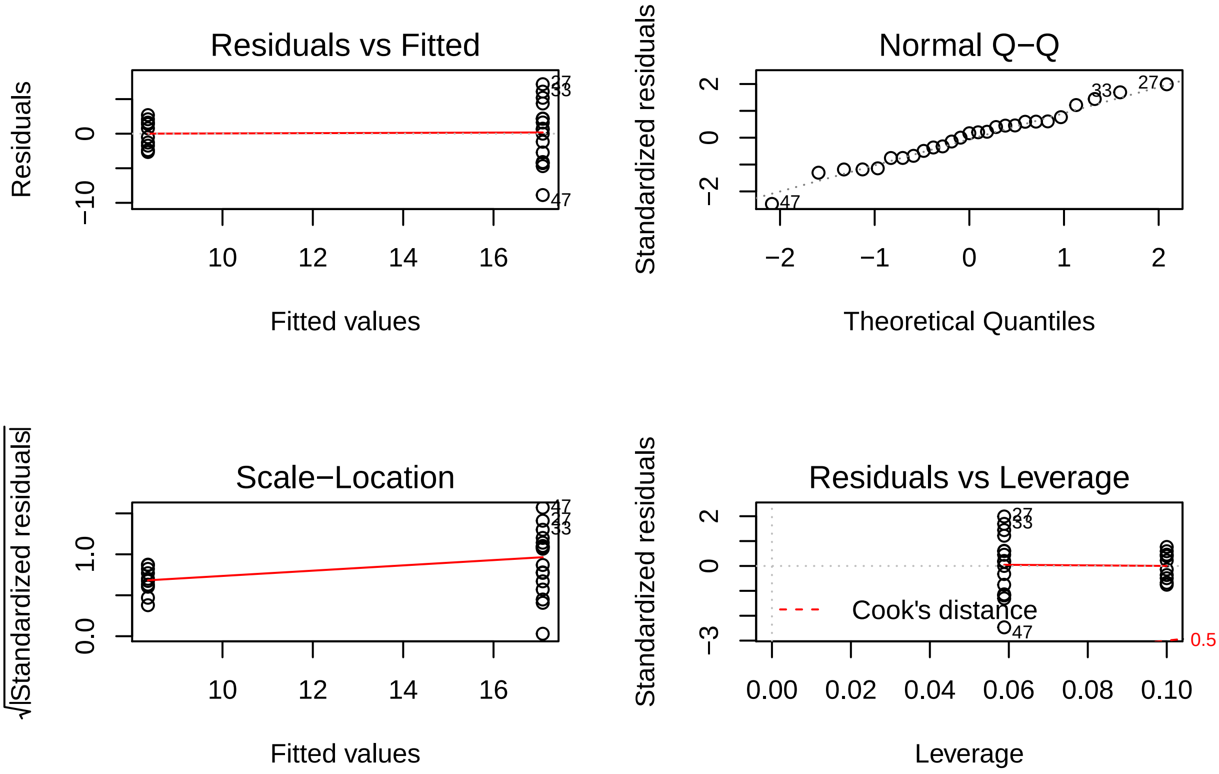
- **The explanatory variable X is measured without error** Your measure of the explanatory variable should be exact, in reality this is very difficult to do!

- **The residuals of the model are normal** For any given value of X, the sampled Y values normally distributed errors arround the model fit.

- **The resudials have constant variability** The model fits the data equally well across the full span of your explanatory variables (homoscedascity = constand variability; heteroscedacity = varying heteroscedacity).

The first two are difficult to test, but the secon two can be check through plotting th model object:

```
par(mfrow=c(2,2)) # chnage to plot window to allow 4 plots
plot(mod1)
```



Look at the first two plots. The top left plot shows that the model fits each category equally well (the red line is flat), the top right plot shows that the residuals are normal (the points follow the dashed line). The bottom two plots help with outlier identification, don't worry about these for now!

So our model looks reasonable. Let's do another example.

6.2 Are Coatis more abundant in close proximity to the ocean

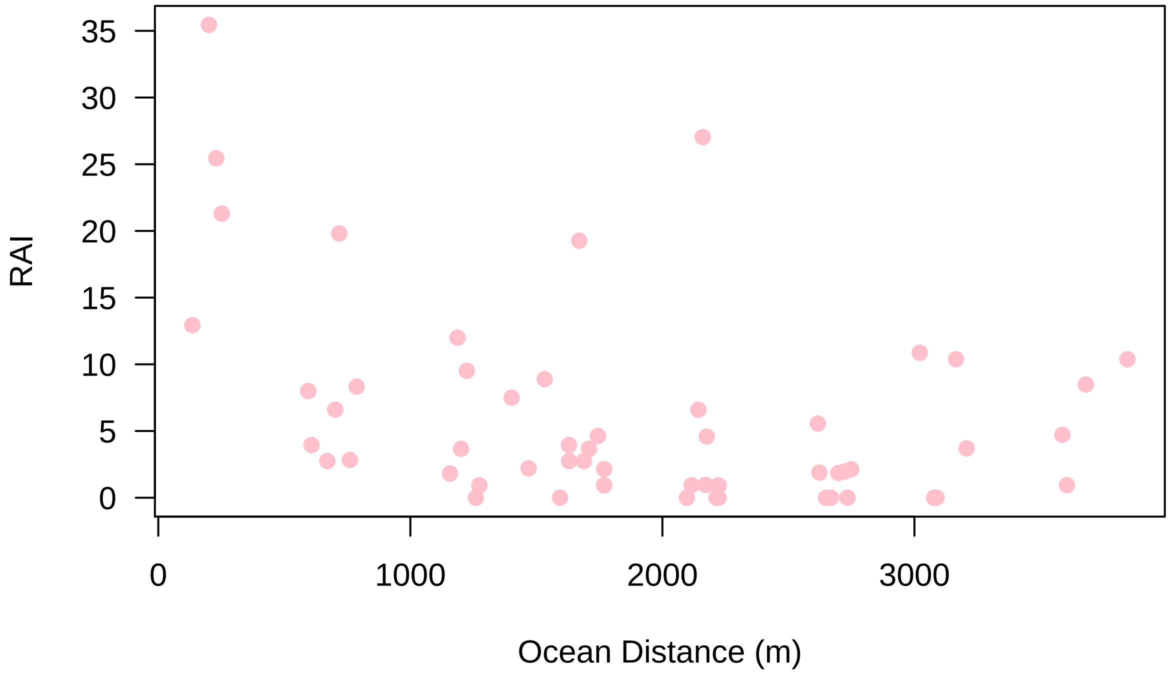
Now lets fit a different model using the capture data from the Osa grid. Coatis are well known to raid turtles nests for food, for this reason we might expect that coatis are more frequently detected close to the ocean. Let's see if this is the case.

First, let's import our relative abundance index dataset from earlier ()

```
# Read in the dataframe
count.sp <- read.csv("SpeciesRAIdata.csv", header=T)

# Subset the data to just coatis
coati <- count.sp[count.sp$Code=="coati",]
```

First lets plot the data:

```
par(mfrow=c(1,1))
plot(coati$RAI~coati$ocean_dist, pch=19, col="pink", ylab="RAI",
     xlab= "Ocean Distance (m)", las=1)
```



Fit the linear models, m1 = capture rate varies by ocean distance:

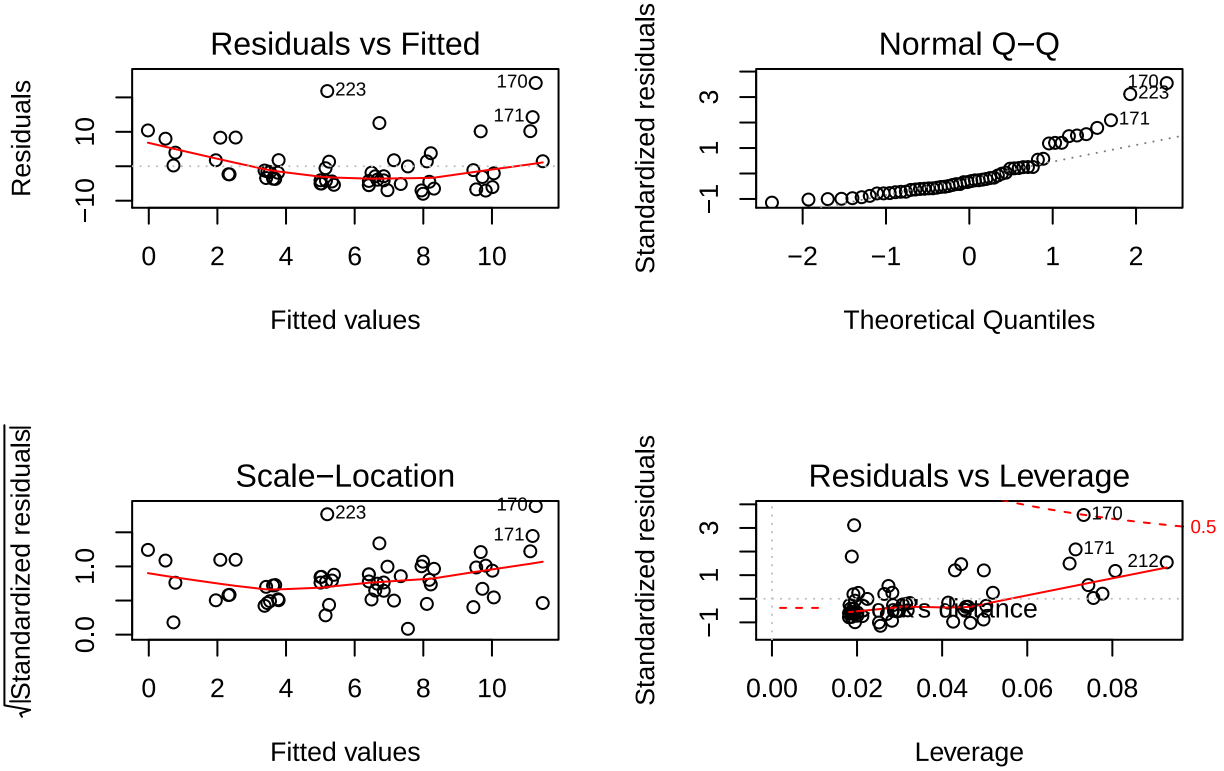
```
m1 <- lm(RAI~ocean_dist, data=coati)
summary(m1)
```

```
##
## Call:
## lm(formula = RAI ~ ocean_dist, data = coati)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -7.992 -4.431 -2.188  1.745 24.175
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 11.8985846  2.0908781   5.691 5.32e-07 ***
## ocean_dist  -0.0031008  0.0009889  -3.136  0.00277 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.078 on 54 degrees of freedom
## Multiple R-squared:  0.154,  Adjusted R-squared:  0.1384
## F-statistic: 9.833 on 1 and 54 DF,  p-value: 0.002773
```

If we examine the p-value for `ocean_dist` in the m1 model, you can see it is significant (0.00277), suggesting it performs better than the null model (capture rate does not vary by `ocean_dist`).

Let's check how the model fits the data.

```
par(mfrow=c(2,2))
plot(m1)
```



The top left plot suggests that the model does not fit very well. It is under estimating coati abundance in close proximity to the ocean, over estimating it at intermediate distances, and underestimating it again at far distances.

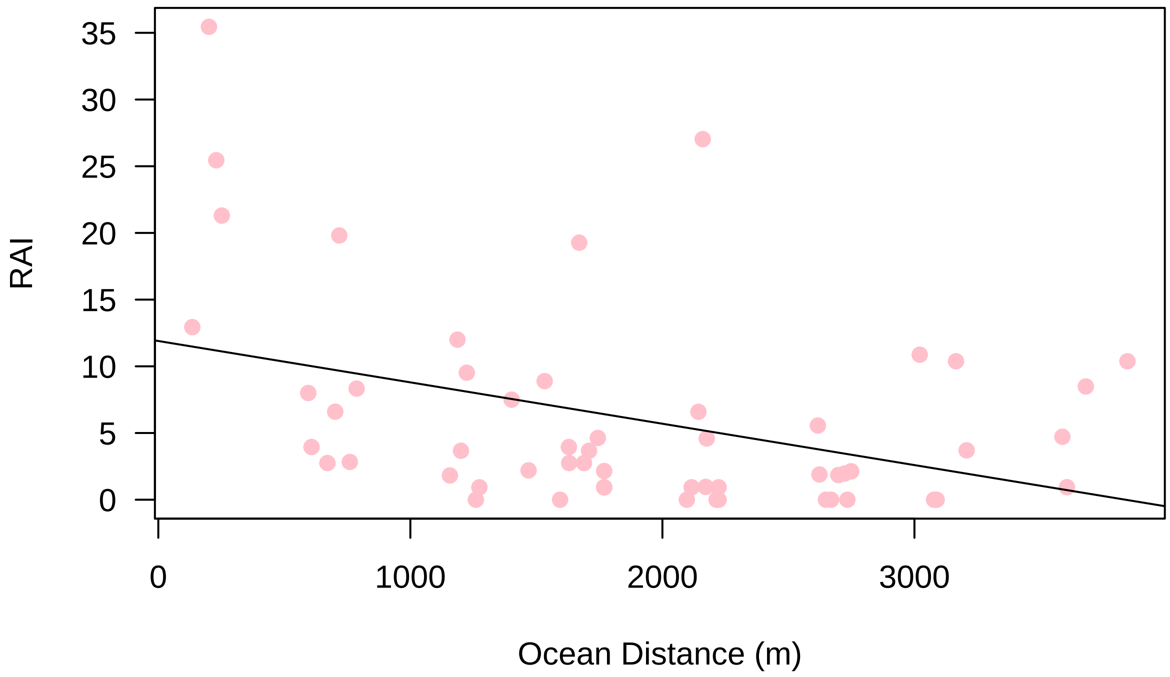
The top right plot also suggests that our residuals are not normally distributed.

Let's plot the model prediction to see if this is the case:

```
coef(m1) # Pulls out the intercept and slope parameters for the model.
```

```
## (Intercept)  ocean_dist
## 11.89858459 -0.00310083
```

```
par(mfrow=c(1,1))
plot(coati$RAI~coati$ocean_dist, pch=19, col="pink", ylab="RAI",
     xlab= "Ocean Distance (m)", las=1)

abline(coef(m1), col="black") # plots the line
```



Our fitted model fits the description above - it isnt performing very well across the whole range of ocean distance.

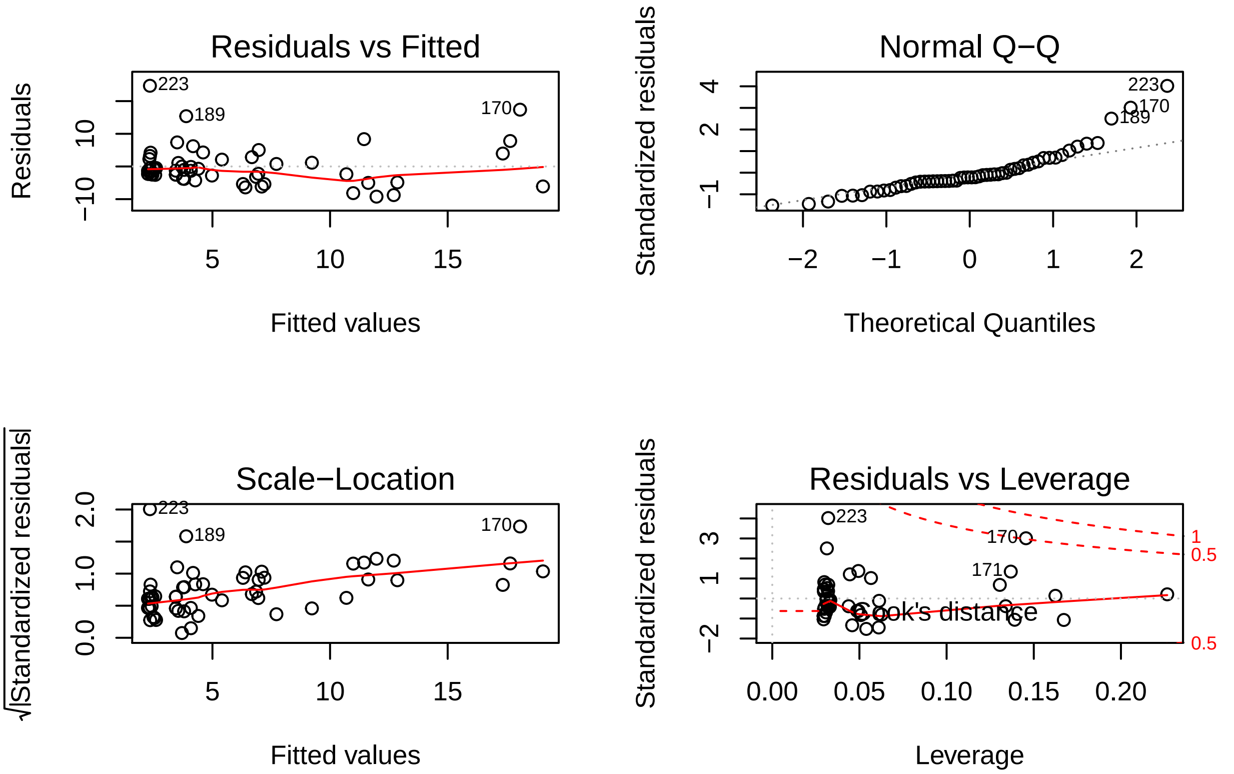
Lets try a quadratic shape for the line:

```
m2 <- lm(RAI~ocean_dist + I(ocean_dist^2), data=coati)
summary(m2)
```

```
##
## Call:
## lm(formula = RAI ~ ocean_dist + I(ocean_dist^2), data = coati)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -9.224 -2.881 -1.331  2.140 24.687
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)      2.113e+01  2.932e+00   7.209 2.08e-09 ***
## ocean_dist      -1.588e-02  3.272e-03  -4.853 1.11e-05 ***
## I(ocean_dist^2)  3.325e-06  8.205e-07   4.052 0.000167 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.242 on 53 degrees of freedom
## Multiple R-squared:  0.3542, Adjusted R-squared:  0.3298
## F-statistic: 14.53 on 2 and 53 DF,  p-value: 9.299e-06
```

The quadratic term is also significant! Interesting. Lets check the model assumptions.

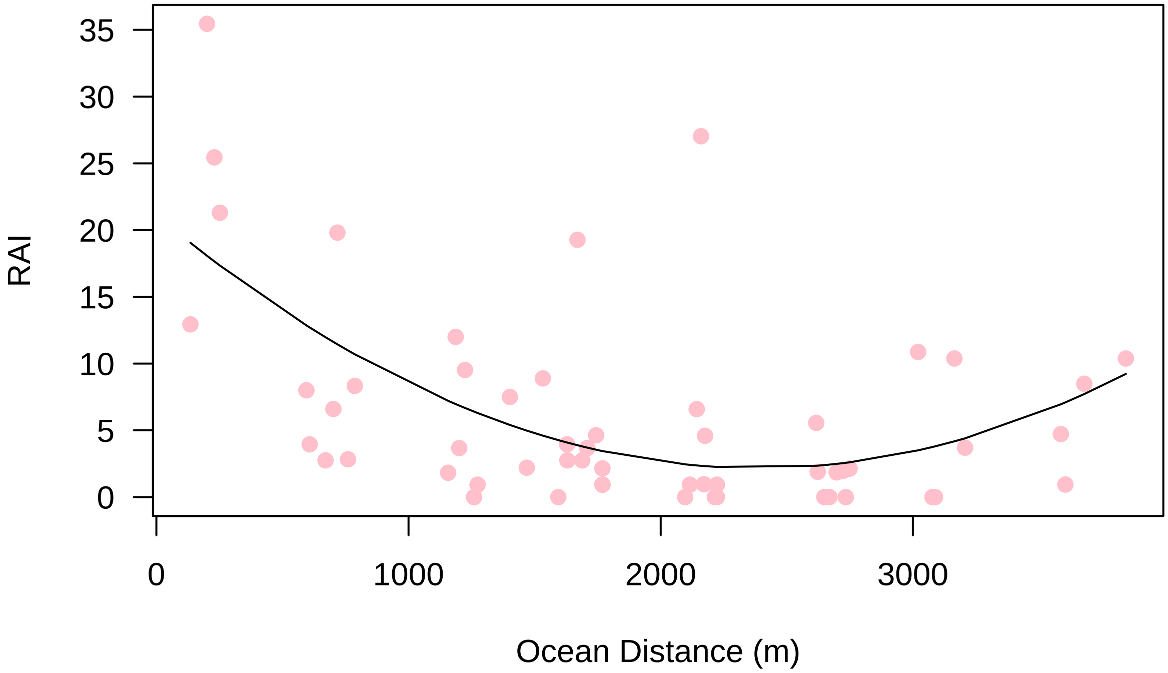
```
par(mfrow=c(2,2))
plot(m2)
```



They look better than before, the top left plot looks much more like a “sky at night”.

Now lets plot the model predictions. the `abline()` command we used earlier can only handle straight lines, so our code will have to be more complex:

```
par(mfrow=c(1,1))
plot(coati$RAI~coati$ocean_dist, pch=19, col="pink", ylab="RAI",
     xlab= "Ocean Distance (m)", las=1)
lines(predict.lm(m2)[order(coati$ocean_dist)]~ coati$ocean_dist[order(coati$ocean_dist)],
      type="l", col="black")
```



That looks great! But now the question becomes… is it better than model 1? Both have significant variables in, but you can't compare the p-values. Instead we will use information criterion, or AIC. when doesnt an AIC comparison the **lower** number is always better. The lower the number, the more information a model explains (after paying a cost for how complex it is).

```
AIC(m1,m2)
```

```
##    df      AIC
## m1  3 382.0666
## m2  4 368.9519
```

The m2 model has a lower AIC - by a long way (many people assume AICs>2 to reflect a 95% difference).

Congratulations - you have performed your first round of model selection!

6.2.1 Comparing multiple predictors of Coati detections

We have several variables which may influence Coati detects. What if we want to found out this is the most important? Again we can use AIC to perform this. The potential predictors we have identified are: river distance, ocean distance, road distance, trail distance, altitude and canopy cover. First fit every model individually, then use the MuMIn package to compare the results:

```
n1 <- lm(RAI~river_dist, data=coati)
n2 <- lm(RAI~ocean_dist, data=coati)
n3 <- lm(RAI~road_dist, data=coati)
n4 <- lm(RAI~trail_dist, data=coati)
```

```
n5 <- lm(RAI~altitude, data=coati)
n6 <- lm(RAI~canopy_cover, data=coati)

library(MuMIn)

model.sel(n1,n2,n3,n4,n5,n6)
```

```
## Model selection table
##     (Int) rvr_dst   ocn_dst  rod_dst   trl_dst      alt  cnp_cvr df
## n2 11.900         -0.003101                                       3
## n6 10.020                                               -0.08841  3
## n1  3.773  0.0041                                                 3
## n5  7.510                                      -0.01449           3
## n3  7.720                  -0.00331                               3
## n4  7.002                           -0.007115                     3
##      logLik  AICc delta weight
## n2 -188.033 382.5  0.00  0.834
## n6 -190.562 387.6  5.06  0.066
## n1 -191.085 388.6  6.10  0.039
## n5 -191.526 389.5  6.98  0.025
## n3 -191.782 390.0  7.50  0.020
## n4 -192.015 390.5  7.96  0.016
## Models ranked by AICc(x)
```

So the ocean distance variable is the best supported predictor! Phew!

Task: Try this process with a different animal. Do you get the same result?

That is all for now. For more information check out:

Statistics: an introduction using R - Micheal Crawley

The R Book - Micheal Crawley