BB852 - Data handling, visualisation and statistics

Owen R. Jones

2021-07-06

Contents

Chapter 1

Preface

This book has been written to accompany the course, *BB852 - Data Handling, Visualisation and Statistics*.

It is available as a website (https://jonesor.github.io/BB852_Book/) or as a PDF (click the link at the top of book's website). I recommend to use the website where possible because the formatting is sometimes messy on the PDF, but the PDF is useful if you want a copy for offline use.

Note: The book is a "work in progress" and will change during the course. The latest version can always be found at the website, or by downloading it again. Please let me know (jones@biology.sdu.dk) if you spot any errors, or have any suggestions for improvement.

The book is divided into three parts: data wrangling, data visualisation and statistics.

1.1 Data wrangling

The term data wrangling covers manipulation of data, for example collected from an experiment or observational study, from its raw form to a form that is ready for analysis, or summarised into tables. It includes reshaping, transforming, filtering and augmenting from other data. This book covers these processes in R mainly using the tools from the `dplyr` package.

1.2 Data visualisation

Graphing data is a crucial analytical step that can both highlight problems with the data (e.g. errors and outliers) and can inform on appropriate statistical approaches to take. This book covers the use of `ggplot2` to make high quality, publication-ready plots.

1.3 Statistics

Statistics is a *huge* field and this book does not attempt to cover more than a small fraction of it. Instead it focusses on (ordinary) linear models and generalised linear models. In a nutshell, linear models model the effects of explanatory variables on a continuous response variable with a gaussian (normal) error distribution while generalised linear models (GLMs) offer a more flexible approach that allows the response variable to have non-normal error distributions. This flexibility allows the more-appropriate modelling of phenomena including integer counts (e.g. number of individuals, or species, or events), binary (0/1) data (e.g. survived/died) or binomial count data (e.g. counts of successess and failures). It is important to realise that most commonly-used statistical methods including t-tests, ANOVA, ANCOVA, n-way ANOVA, and of course linear and multiple regression are all special cases of linear models.

My general approach with communicating these methods and ideas is to teach using examples. Therefore, the bulk of the text here consists of walk-throughs of manipulating, plotting and analysing real data. For the statistics section I focus on communicating the "gist" of the underlying mathematical machinery rather than the mathematical details. If you find yourself interested in these details then there are more specialist textbooks available.

This book accompanies the course lectures. The general idea is that there will be a lecture, followed by computer work where you work through the examples in the relevant chapter of this book. At the end of most chapters there are also exercises to test your new skills. It is very important that you do these to gradually build up your skill level and confidence.

1.4 Data sources

This book uses numerous data sets in examples, most of which are real datasets obtained from published works, or collected by me.

The data sets can be found at the following link: https://www.dropbox.com/sh/z8iv9fl9l00bm0w/AAD1WjFwcrr1ERugpunp_YH-a?dl=0

1.5 Your instructors

You are welcome to contact us with any problems/questions (but please put a little effort in first).

- Owen Jones, Associate Professor, jones@biology.sdu.dk
- John Jackson, Postdoc, johnja@biology.sdu.dk

1.6 Acknowledgements

These materials are inspired by the excellent textbook, "Getting Started With R"[1], which is the recommended textbook for BB852, and by materials for the Sheffield University course "AP 240 - Data Analysis And Statistics With R" ([https://dzchilds.github.io/stats-for-bio/]). For your convenience, the datasets for the Beckerman et al. book are available at this course's data Dropbox link (above), in a folder called "GSWR_datasets".

[1] Beckerman, Childs & Petchey (2017) *Getting Started With R.* Oxford University Press (2nd edition)

Chapter 2

Schedule

This is the schedule for the course. Please note that it is liable to change (possibly at short notice). If you find a mismatch between this schedule and the official one[1], then it is the official one that is correct.

The columns, GSWR and Course Book, refer to the relevant chapters in the recommended text book ("*Getting Started With R*") and this course book, respectively. You should aim to read and work through these chapters as the course proceeds.

The names of the topics for the **Practical** sessions corresponds to the chapter names in this website/book.

The schedule is only available on the HTML version of this document

2.1 Additional recommended reading

These can be downloaded via the link on Blackboard.

- Broman, K. W., & Woo, K. H. (2018). *Data Organization in Spreadsheets.* The American Statistician, 72(1), 2–10.
- Gotelli, N. J., & Ellison, A. M. (2013) Chapter 4, *Framing and Testing Hypotheses*, in A Primer of Ecological Statistics. Sinauer.
- Petchey, O., Beckerman, A., & Childs, D. (2009). *Shock and Awe by Statistical Software - Why R?* Bulletin of the British Ecological Society, 40(4), 55–58.
- Weissgerber, T. L., Milic, N. M., Winham, S. J., & Garovic, V. D. (2015). Beyond bar and line graphs: time for a new data presentation paradigm. PLoS Biology, 13(4), e1002128. doi:%5B10.1371/journal.pbio.1002128](https://doi.org/10.1371/journal.pbio.1002128)

[1]https://mitsdu.sdu.dk/skema/activity/N110040101/e21

- Wickham, H. (2014). *Tidy Data.* Journal of Statistical Software, 59(10), 1–23.

The following websites are also useful.

The R graph gallery: https://www.r-graph-gallery.com/

STHDA: http://www.sthda.com/english/wiki/ggplot2-essentials http://www.sthda.com/english/wiki/r-basics-quick-and-easy

Chapter 3

An R refresher

In this course we will be learning how manipulate, visualise and analyse data statistically using **R**. **R** is a programming language for data analysis and statistics. It is free and very widely used. One of its strengths is its very wide user base which means that there are hundreds of contributed packages for every conceivable type of analysis. The aim of these introductory sections is to give a basic introduction to the programming language as a tool for importing, manipulating, and exploring data. In later sections we will learn more about statistical analysis.

Before proceeding you will need to ensure you have a recent version of **R** installed on your computer (the version I am using right now is 4.0.5).

Do this: Check your R version, and/or install R on your own computer now.

In this course we will not be using **R** on its own. Instead, we will be using it with RStudio.

R and RStudio are not the same thing. It is possible to run R without RStudio, but RStudio will not work if **R** is not installed. So what is RStudio? RStudio, essentially, is a helpful piece of software that makes **R** easier to use. The three most useful features are:

- The R Console - this is where **R** runs inside RStudio. We can work *directly* with **R** by typing commands into this "console". It is also where outputs (results) from **R** are printed to the screen.
- The Code Editor - this is where you can write **R** programs (called "scripts")", which are a set of commands/instructions in the **R** language saved to a text file. It is much easier to work with scripts using RStudio than with ordinary text editors like Notepad. For example, it colour codes the text to make it easier to read and it will"auto-complete" some text to speed up your work.

- Useful “point-and-click” tools - RStudio can help with tasks like importing data, managing files, reading help files, and managing/installing packages. Doing these things is trickier in *just* **R**: RStudio just makes things easier!

You should do your coding from within RStudio.

You can download **RStudio Desktop** from https://rstudio.com/products/rstudio/download/. Select the correct version for your computer (Mac/Windows) and follow the usual instructions.

Do this: Install **RStudio Desktop** on your computer.

3.1 Getting started with R

In RStudio, create a new “R Script” file. *Scripts* are essentially programs that can be saved to allow you to return to your work in the future. They also make debugging of errors much easier.

You can use the menu to do create a new R Script (`File > New File > R Script`), but there’s also a keyboard shortcut (Windows: `Ctrl+Shift+N`; Mac: `Cmd+Shift+N`). If you save (Windows: `Ctrl+S`; Mac: `Cmd+S`), you will be prompted for a file name. Make sure it has the suffix “.R” which denotes an R script file. Save the file in a folder with a memorable name (e.g. `BB852_Work`).

When you double click on this file in future, it should automatically open in RStudio (if it doesn’t you should be able to right-click and select `Open with...`).

In RStudio you can execute commands using the “run” icon at the top of the script window, or by selecting the text and typing the shortcut `Ctrl+Enter` (Windows) or `Cmd+Enter` (Mac). Another helpful feature of RStudio is that it will colour-code the syntax that you type, making it easier to read and debug. Note that the colours you see may be different from the ones shown in this handout.

You can customise the look of RStudio using by clicking **Tools** → **Options** menu on Windows or **RStudio** → **Preferences** on a Mac. I will point out some of this in the lecture, or you can ask me to show you.

Over the next few pages I will introduce the basics of the R programming language. Try typing them into the scripting window (top left) in RStudio and ensuring that you understand what the commands are doing. It is impossible to “break” R by typing the wrong command so I encourage you to experiment and explore the R language I introduce to you here as much as possible - it really is the best way to learn!

The "look" of RStudio can be modified by changing the Preferences (**RStudio** → **Preferences** → **Appearance**). Also, there are some useful keyboard shortcuts that are worth learning, to run code, save files etc. without needing to point-and-click (**Tools** → **Keyboard Shortcuts Help**).

3.2 Getting help

R features a wealth of commands, which are more properly termed **functions**. You will learn many of these over the next few weeks. Functions often feature a several options which are specified with **arguments**. For example, the function `sum`, has the argument `...`, which is intended to be one or more **vectors** of numbers (see below), and the argument `na.rm`, which is a logical argument specifying whether or not missing values should be removed or not. Usually the arguments have default options which are used you choose not to specify them. In addition, you don't necessarily need to fully-specify the argument if they are specified in the *correct order*.

You can get **help** on R functions from within R/RStudio with the `?` and `help.search` commands. `?` requires that you know the function name while *help.search* will search all the available help files for a particular word or phrase. `??` is a synonym for *help.search*:

```
?rep
help.search("bar plot")
??"bar plot"
```

In RStudio, the help results will appear in the lower right hand area.

3.3 R as a fancy calculator

R features the usual arithmetic operations for addition, subtraction, division, multiplication:

```
4+3
```

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

```
9-12
```

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

```
6/3
```

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

```
7*3
```

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

```
(2*7)+2-0.4
```

```
## [1] 15.6
```

R also has commands for square root (`sqrt`), raising to powers (`^`), taking the absolute value (`abs`), and rounding (`round`), natural log (`log`), anti-log (`exp`), log to base-10 (`log10`):

```
sqrt(945)
```

```
## [1] 30.74085
```

```
3^5
```

```
## [1] 243
```

```
abs(-23.4)
```

```
## [1] 23.4
```

```
round(2.35425,digits=2)
```

```
## [1] 2.35
```

```
log(1.2)
```

```
## [1] 0.1823216
```

```
exp(1)
```

```
## [1] 2.718282
```

```
log10(6)
```

```
## [1] 0.7781513
```

Another thing you can do is evaluate TRUE/FALSE conditions:

```
3<10
```

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

```
5>7
```

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

```
5==5
```

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

```
6!=5
```

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

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

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

```
6 %in% c(1,2,3,4,5)
```

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

3.4 Objects in R

R is an object oriented programming language. This means that it represents concepts as **objects** that have data fields describing the object. These objects can be manipulated by **functions**. Objects can include data, but also models. Don't worry about these distinctions too much for now - all will become clear as you proceed!

Objects are assigned names in R like this. The "`<-`" command is pronounced "*gets*" so I would pronounce the following as "*x gets four*":

```
x <- 4
```

To look at any object (function or data), just type its name.

```
x
```

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

The main data object types in R are: *vectors*, *data frames*, *lists* and *matrices*. We will focus on the first two of these during this course.

A vector is simply a series of data (e.g. the sequence *1, 2, 3, 4, 5* is a vector, so is the non-numeric sequence *Male, Female, Female, Male, Male*). Each item in a vector is called an **element**. Therefore, both of these examples contain 5 elements.

There are several ways to create vectors in R. For example, you can make vectors of integers using the *colon* (`:`) function (e.g. `1:5`), or vectors of any kind of variable using the `c` function. `c` stands for *concatenate*, which means to *join (things) together in a chain or series*. Other convenient functions for making vectors are `seq`, which builds a sequence of numbers according to some rules, and `rep` which builds a vector by repeating elements a specified number of times.

Try the following:

```
A <- 1:5
B <- c(1,3,6,1,7,9)
C <- seq(1,12,2)
D <- seq(1,5,0.1)
E <- rep(c("Male","Female"),each = 3)
G <- rep(c("Male","Female"),c(2,4))
```

Try modifying the commands to make sure you know what the commands are doing.

Chapter 4

Manipulating objects

Objects can be manipulated (just like in real life). In R, we use **functions** to manipulate objects.

For example, we can use the basic arithmetic functions (`*`, `+`, `/`,`-`) on a vector:

```
B
```

```
## [1] 1 3 6 1 7 9
```

```
B*3
```

```
## [1]  3  9 18  3 21 27
```

```
B-2
```

```
## [1] -1  1  4 -1  5  7
```

You can *concatenate* entire vectors together using the `c` function. E.g. concatenating the vectors `A` and `B` from above:

```
c(A,B)
```

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

Other manipulations are also done "element-by-element". For example, here we multiply the first element of B by 1, the second by 2, the 3rd by 3 and so on…:

```
B * c(1,2,3,4,5,6)
```

```
## [1]  1  6 18  4 35 54
```

If the length of the vectors match, we can also multiply (or add/subtract/divide etc.) multiple vectors:

```
A / B
```

```
## [1] 1.0000000 0.6666667 0.5000000 4.0000000 0.7142857 0.1111111
```

4.1 Missing values, infinity and "non-numbers"

By convention, *missing values* in R are coded by the value "NA". The way that particular functions handle missing values varies: sometimes the NA values are stripped out of the data, other times the function may fail.

For example, if we asked for the mean value of a vector of numbers with an NA value, it will fail:

```
mean(c(1,3,6,1,7,9,NA))
```

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

In this case you need to specify that any NA values should be removed before calculating the mean:

```
mean(c(1,3,6,1,7,9,NA),na.rm=TRUE)
```

```
## [1] 4.5
```

Calculations can sometimes lead to answers that are plus, or minus, infinity. These values are represented in R by `Inf` or `-Inf`:

```
5/0
```

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

```
-4/0
```

```
## [1] -Inf
```

Other calculations lead to answers that are not numbers, and these are represented by `NaN` in R:

```
0/0
```

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

```
Inf-Inf
```

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

4.2 Basic information about objects

You can obtain information about most objects using the `summary` function:

```
summary(B)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    1.50    4.50    4.50    6.75    9.00
```

The functions `max`, `min`, `range`, and `length` are also useful:

```
max(B)
```

```
## [1] 9
```

```
min(B)
```

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

```
range(B)
```

```
## [1] 1 9
```

```
length(B)
```

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

4.3 Data frames

Data frames are the usual way of storing data in R. It is more-or-less the same as a worksheet in Excel. A data frame is usually made up of a number of vectors (of the same length) bound together in a single object. You can make a data frame by binding together vectors, or you can import them from outside R.

This example shows the creation of a data frame in R, from 3 vectors:

```
height <- c(173, 145, 187, 155, 179, 133)
sex <- c("Male", "Female", "Male", "Female", "Male", "Female")
age <- c(17, 22, 32, 20, 27, 30)

mydata <- data.frame(height = height, age = age, sex = sex)
mydata
```

```
##   height age    sex
## 1    173  17   Male
## 2    145  22 Female
## 3    187  32   Male
## 4    155  20 Female
## 5    179  27   Male
## 6    133  30 Female
```

Data frames can be summarised using the `summary` function (or the `str` function, which gives you a different view of the same data):

```
summary(mydata)
```

```
##      height           age            sex           
##  Min.   :133.0   Min.   :17.00   Length:6          
##  1st Qu.:147.5   1st Qu.:20.50   Class :character  
##  Median :164.0   Median :24.50   Mode  :character  
##  Mean   :162.0   Mean   :24.67                     
##  3rd Qu.:177.5   3rd Qu.:29.25                     
##  Max.   :187.0   Max.   :32.00                     
```

```
str(mydata)
```

```
## 'data.frame':	6 obs. of  3 variables:
##  $ height: num  173 145 187 155 179 133
##  $ age   : num  17 22 32 20 27 30
##  $ sex   : chr  "Male" "Female" "Male" "Female" ...
```

Data frames can be subsetted using the square brackets `[]`, or `subset` functions. With the square brackets, the first number specifies the row number, while the second number specifies the column number:

```
mydata[1,]
```

```
##   height age  sex
## 1    173  17 Male
```

```
mydata[,2]
```

```
## [1] 17 22 32 20 27 30
```

```
mydata[1,2]
```

```
## [1] 17
```

```
subset(mydata,sex == "Female")
```

```
##   height age    sex
## 2    145  22 Female
## 4    155  20 Female
## 6    133  30 Female
```

4.4 Classes in R

Every objects you create, or import into R, has a "type" called a `class`. You can ask what class an object has using the `class` function.

For example, the vectors you created above have types.

```
class(height)
```

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

```
class(sex)
```

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

```
class(mydata)
```

```
## [1] "data.frame"
```

You can find out the class of all columns in a `data.frame` by asking for a summary with `str`. For example, in this example, there are two numeric columns (`num`) and a character column (`chr`).

```
str(mydata)
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ height: num  173 145 187 155 179 133
##  $ age   : num  17 22 32 20 27 30
##  $ sex   : chr  "Male" "Female" "Male" "Female" ...
```

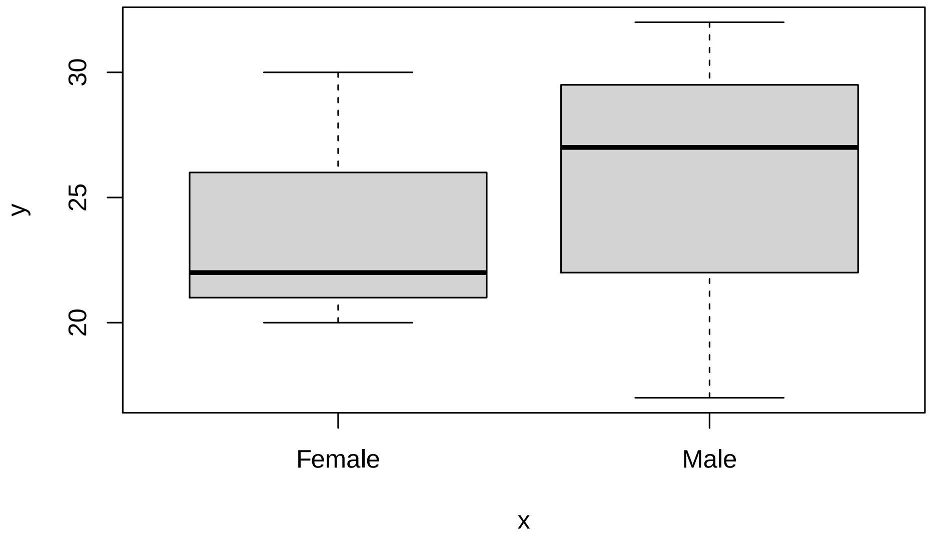
There's another special class of vector called `factor`. In the small dataset above (`mydata`), sex is registered by R to be a `character` vector. For some functionality this is perfectly fine, but for others you will need to convert the data into a factor.

For example, this code, to make a box plot, will not work:

```
plot(mydata$sex,mydata$age)
```

But this code will work fine:

```
plot(as.factor(mydata$sex),mydata$age)
```

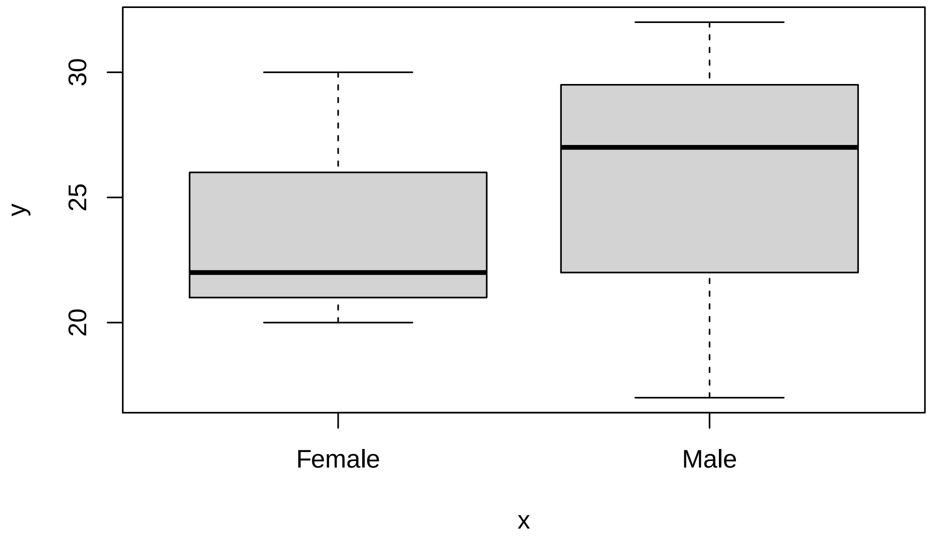


Of course it might be easier to convert it to be a factor in the data frame itself, like this:

```
mydata$sex <- as.factor(mydata$sex)
str(mydata) # You can see that it is now registered as a factor
```

```
## 'data.frame':    6 obs. of  3 variables:
##  $ height: num  173 145 187 155 179 133
##  $ age   : num  17 22 32 20 27 30
##  $ sex   : Factor w/ 2 levels "Female","Male": 2 1 2 1 2 1
```

```
plot(mydata$sex,mydata$age)
```



> If you are getting strange results from your code it is a good idea to check the structure of the data. Are the classes what they should be?

4.5 Organising your work

It would be incredibly tedious to enter real data into **R** by typing it in!

Thankfully, R can import data from a several data formats, and it understands the file structure of your computer. Thus, you can use spreadsheet software (like Excel) to enter and store your data, and you can organise your project work in a sensible way in folders (sometimes called *directories*) on your computer.

The most commonly used data format is `comma separated value (CSV)` so I will use that. You can also import from Excel, but the data must be formatted in a particular way to enable this (I'll cover this in a later class).

For this course, I suggest that you make a folder somewhere on your computer called "IntroToR". We will use this as the `working directory` for the remainder of the session. In RStudio you can set the working directory by clicking through the menu items **Session** → **Set Working Directory** → **Choose Directory**.

You can also using the `setwd` function to do this, if you know where your files are stored (the *file path*). File paths in Windows and Mac computers are expressed differently. Apple systems use the forward-slash (/) to separate folders whereas Windows can use the forward-slash (/) or double-backslash (\\). In windows you also need to define the drive (e.g. C:).

So, to set the working directory in Apple OSX you would use something like this (obviously, you need to put *your* path!):

```
setwd("/Users/orj/Desktop/IntroToR")
```

While in Windows the equivalent command would be something like this (both of the following should work):

```
setwd("C:\\Users\\orj\\Desktop\\IntroToR")
setwd("C:/Users/orj/Desktop/IntroToR")
```

Typing the path in can be annoying but there are ways to speed it up. In Windows you can copy paths from the Windows Explorer location/address bar, or you can hold down the Shift key as you right-click the file, and then choose Copy As Path.

On a Mac you can copy file paths from Finder: Select your file/folder, Right click, Press the option key (on my keyboard this is the `alt` key) and click "Copy X as Pathname"

I can check what the current working directory is using the `getwd` function:

```
getwd()
```

It is good practice to keep your files well-organised. I recommend that you create a folder in your working directory called `CourseData` (or similar). Store your data files in this folder.

I have put all the data for the course into a Dropbox folder - see the link in Chapter 1. In there you will find a file called "`carnivora.csv`". Download this to your new `CourseData` folder.

You can now import this file into R using the `read.csv` function. The specification of the argument `header = TRUE` signifies that the first row of our CSV file contains the column names. Note that your file path will be different to mine:

```
carni <- read.csv("CourseData/carnivora.csv", header = TRUE, stringsAsFactors = TRUE)
```

The `stringsAsFactors` argument tells R to treat text-type data (technically known as "character **strings**") as a special kind of data called **factors**. Essentially, factors are "categorical data" where the data can take a limited number of discrete values. For example, "treatmentA", "treatmentB", "treatmentC". Although this may seem a little esoteric right now, it is important to ensure that your data is recognised by R in the correct way. In **most** cases, your text-type data will be factor data, so it is usually safe to set `stringsAsFactors = TRUE`.

Tip: RStudio also has a point-and-click "Wizard" to help import data. Look for "Import Dataset" in the top-right pane.

4.6 Inspecting the data

We can get some basic information on your imported data (e.g. the `carni` data frame) using the `summary` function, but also the `dim` and `nrow/ncol` functions:

```
summary(carni)
```

```
dim(carni)
```

```
## [1] 112  17
```

```
nrow(carni)
```

```
## [1] 112
```

```
ncol(carni)
```

```
## [1] 17
```

We can find the names of the columns of a data frame with the `names` function:

```
names(carni)
```

```
##  [1] "Order"       "SuperFamily" "Family"      "Genus"       "Species"
##  [6] "FW"          "SW"          "FB"          "SB"          "LS"
## [11] "GL"          "BW"          "WA"          "AI"          "LY"
## [16] "AM"          "IB"
```

The first few columns are to do with the taxonomic placement of the species (Order, SuperFamily, Family, Genus and Species). There then follow several columns of life history variables: FW = Female body weight (kg), SW = Average body weight of adult male and adult female (kg), FB = Female brain weight (g), SB = Average brain weight of adult male and adult female (g), LS = Litter size, GL = Gestation length (days), BW = Birth weight (g), WA = Weaning age (days), AI = Age of independence (days), LY = Longevity (months), AM = Age of sexual maturity (days), IB = Inter-birth interval (months).

You can refer to the sub-parts of a `data.frame` (the columns) using the `$` syntax:

```
summary(carni$FW)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.050   1.245   3.400  18.099  10.363 320.000
```

4.7 “Classes” in R

I have already mentioned the different object types in R (e.g. vectors and data frames). The object types are technically known as “classes”. You can find out what “class” an object is by using the `class` function:

```
class(carni)
```

```
## [1] "data.frame"
```

In this case, the data frame is, unsurprisingly, of class "data.frame". However, the vectors that compose the data frame also have classes. There are several classes of vectors including "integer" (whole numbers), "numeric" (real numbers), "factor" (categorical variables) and "logical" (true/false values).

I expect you have heard of the first two data types, but "factor" might be puzzling. Factors are defined as variables which can take on a *limited* number of different values. They are often referred to as `categorical variables`. For example, in the carnivore dataset, the taxonomic variables are factors. The different values that a factor can take are known as `levels` and you can check on the levels of a vector with the `levels` function.

```
class(carni$Family)
```

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

```
levels(carni$Family)
```

```
## [1] "Ailuridae"   "Canidae"     "Felidae"     "Hyaenidae"   "Mustelidae"
## [6] "Procyonidae" "Ursidae"     "Viverridae"
```

4.8 Tables and summary statistics

For vectors of class "factor" you can use the `table` function to give the counts for each level:

```
table(carni$Family)
```

```
##
##   Ailuridae     Canidae     Felidae   Hyaenidae  Mustelidae Procyonidae
##           1          18          19           4          30           4
##     Ursidae  Viverridae
##           4          32
```

You can use the function `tapply` ("table apply"), to get more complex summary information. For example, I could ask what the mean female weight (FW) is in each of the families using the argument `mean`:

```
tapply(carni$FW, carni$Family, mean)
```

```
##   Ailuridae     Canidae     Felidae   Hyaenidae  Mustelidae Procyonidae
##  120.000000    9.050000   31.432105   33.540000    3.989000    3.642500
##     Ursidae  Viverridae
##  198.250000    2.672813
```

4.9 Plotting data

Basic plots can be made using the `plot` command. For example, let's have a look at the relationship between log gestation length and log female body weight (see Figure **??**, below):

```
plot(log(carni$FW), log(carni$GL))
```

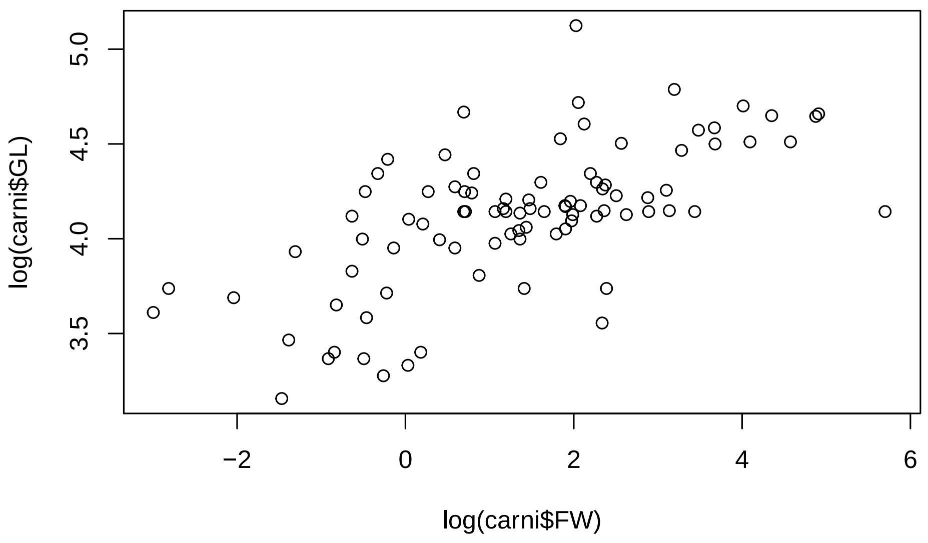


Figure 4.1: A simple scatter plot

4.10 R Packages

R packages are collections of software that add capabilities to "base R". In this course we use several packages including `dplyr`, which adds functionality for manipulating data, `ggplot2` which helps us make pretty plots and `magrittr` which adds tools to allow more "elegant" programming. Packages need to be installed using `install.packages` command before they can be used. You only need to install them once.

```
install.packages("dplyr")
install.packages("ggplot2")
install.packages("magrittr")
```

To use the packages you need to load them with the `library` command, like this:

```
library(dplyr)
library(ggplot2)
library(magrittr)
```

We will be using these packages a lot, and you will need to remember to load them every session. It is therefore useful to add those `library` commands to the top of every script you write.

4.11 Exercise: Californian bird diversity

In the 1950s-1970s there was rapid growth in the number of houses being built in California, with suburbs sprawling out into the new sites in the countryside. What effect would this have on local bird communities?

Surveys on bird abundances were carried out in several locations near Oakland, California [1]. The locations were of different ages, enabling us to investigate what changes might happen through time. Although there were no surveys before the developments, we can regard the bird abundance in the very youngest housing developments as the baseline pre-development condition.

Think about what you might expect to happen to bird species diversity through time in a newly developing suburb.

4.11.1 The data

The relevant data file is called `suburbanBirds.csv`. This file contains data on bird abundances surveyed in 1975. The columns of the data are `Name` (name of the suburb), `Year` (the year that the suburb was built), `HabitatIndex` (an index of habitat quality, related to tree height, garden maturity etc.), `nIndividuals` (number of individual birds seen in a standard survey) and `nSpecies` (number of species seen in a standard survey).

Additional surveys found an average species richness of 3.5 in nearby undisturbed habitats of grassland savanna.

4.11.2 Try the following

1. First import the data. Check that the columns look as they should (use `summary` or `str` functions).
2. What is the mean, minimum, and maximum number of species seen? (there is more than one way to do this)
3. How old are the youngest and oldest suburbs? (hint: the survey was carried out in 1975, do the math!)

[1] Vale, T. R., & Vale, G. R. (1976). Suburban bird populations in west-central California. Journal of Biogeography, 157–165.

4. Plot the relationship between `Year` and `nSpecies` as a scatter plot using base-R graphics (using the `plot` function).
5. The pattern might be easier to see if you could replace `YearBuilt` with suburb age. Create a new vector in your data frame for this variable (e.g. `df$Age <- 1975 - Year)`). Re-plot your results.
6. What do the data show? What might be the mechanisms for the patterns you see? Do they match your expectations?
7. Export your plots and paste them into a Word Document.
8. If you get this far, try plotting the other variables in the dataset.

Chapter 5

Tips and tricks

5.1 Appearance

You can modify the appearance of RStudio via the *Options* dialog: **Tools > Options** menu (**RStudio > Preferences** on a Mac).

There you will find many ways of modifying how RStudio looks and works. However, for beginners I suggest to leave the defaults for most of these. Instead, focus on the Appearance section where you can change the colour scheme or "Theme". My current favourite is "Modern", what's yours? You can also change the font and font size. When choosing a font, it is important to use only "monospace" fonts (these are fonts fixed width of characters). Why would you want to do that? Some fonts are better at distinguishing similar looking characters (e.g. is that an upper case or lower case letter "w", or is it a "1" (the number one) or a "l" (lower case L)). Good examples of monospace fonts are "LucidaConsole", and "Courier". The availability of fonts might differ between computers.

If you choose non-monospaced fonts it can cause problems with formatting your code in the scripts, so if you get any formatting weirdness, check your font!

5.2 Shortcuts

There are several useful keyboard shortcuts that can make your life easier in RStudio. You can find a full list by clicking through the menu – Tools > Keyboard Shortcuts Help – but these are the ones I find most useful:

- Auto-complete code: if you start to write function or object names, after a short pause, RStudio will offer some auto-complete the name. Use the arrow keys to choose the best option, and press Enter.
- Run current line/selection Ctrl+Enter (Cmd+Return, on a Mac)
- Run code from script beginning to current line: Ctrl+Alt+B (Cmd+Option+B)

- Comment/uncomment current line/selection: Ctrl+Shift+C (Cmd+Shift+C)
- Reflow Comment: Ctrl+Shift+/ (Cmd+Shift+/)
- Find text in Files: Ctrl+Shift+F (Cmd+Shift+F)
- Undo: Ctrl+Z (Cmd+Z)
- Cut: Ctrl+X (Cmd+X)
- Copy: Ctrl+C (Cmd+C)
- Paste: Ctrl+V (Cmd+V)
- Parentheses (brackets): Select text you want to "wrap" in parentheses and type Shift+(, or Shift+), to automatically put the brackets on both sides of text.
- Pipes (`%>%`: Ctrl+Shift+M (Cmd+Shift+M) (you'll meet these later in the course!)

5.3 The plots pane

As you make plots they appear in the plot pane, which is by default on the bottom right of RStudio. Old plots are kept in the memory and you can navigate to them using the back/forward arrows at the top of the plot pane.

You can use the *Export* button to copy a plot (for pasting into a document) after resizing it, or to save a plot as an image file.

5.4 Tables

If you have made summary tables, for example with the `dplyr` function `summarise`, you can save them as a csv file using `write.csv(xxx, file = "xxx.csv", row.names = FALSE)`. You can then open the csv in Excel and cut/paste into your Word document as a table.

5.5 Importing data from text files

There are several ways of importing data, and several ways that it can go wrong.

RStudio can import data in text files via the *Import Dataset* button (top right pane), or via the the main menu (File > Import Dataset). There are two ways to import data from text files (such as .csv or .tab or .txt). These are `base` and `readr`. They work in similar ways. During the import process you can choose what the column delimitor (separator) is (e.g. `,` or `;`) and what the decimal separator is (e.g. `.` or `,`).

When you have imported the data, you should check it using e.g. `head` or `str` functions. Make sure that you have the expected number of columns, and that they have the right names. Most of the problems with importing data comes from problems with Excel. A common problem is that the columns of the data are squashed together into a single column. This can be because (1) you have chosen the wrong delimitor (separator) or (2) because Excel has saved the csv file incorrectly.

To avoid this problems (1) check the delimitor (2) avoiding saving the file with Excel when you download it. You can also open the text file in a text editor and remove problematic characters using "Find and Replace". For example, the single-column problem is caused by Excel putting the line of data within quotation marks, which you could remove and then save the data.

5.6 Importing data from Excel

Yes, it is possible to import data directly from Excel using the *Import Dataset* dialogue (or using the `read_excel` function from the `readxl` package). This is pretty neat – BUT – the data need to be very well-arranged for this to work properly. What do I mean by "well-arranged"? Read the paper by Broman & Wu for more details [1]

5.7 Numbers

R uses scientific notation when showing you numbers. Thus very large and very small numbers are shown with exponentials so that 1,000,000 is shown as `1e+06` and `0.0005` is shown as `5e-04`, for example. This can be a bit confusing, and you can turn this behaviour off by setting the `scipen` option like this.

```
options(scipen = 999)
```

The option will be set until you re-start R, or until you turn it off with `options(scipen = 0)`. You could start your script by setting this option.

[1]Broman, K. W., & Woo, K. H. (2018). *Data Organization in Spreadsheets*. The American Statistician, 72(1), 2–10.

Part I

Data Wrangling

Chapter 6

Data wrangling with `dplyr`

This chapter focuses on using the package `dplyr`, which is designed to make working with data in R easier. The package has several key "workhorse" functions, sometimes called **verbs**. These are: `filter`, `select`, `mutate`, `arrange` and `summarise`. I covered these in the lecture, and they are also discussed in the textbook. This chapter guides you through worked examples to illustrate their use.

We will also be using **pipes** from the `magrittr` package. These are implemented using the command `%>%`.

```
library("dplyr")
library("magrittr")
```

To get to know `dplyr` and its functions we'll use a data set collected from the university campus at University of Southern Denmark (SDU) The SDU bird project follows the fate of mainly great tits (musvit) and blue tits (blåmejse) in about 100 nest boxes in the woods around the main SDU campus.

We will address two questions concerning clutch size (the number of eggs laid into the nest) -

1. How does clutch size differ between blue tits and great tits?
2. How does average clutch size vary among years?

To answer these questions we need to calculate the average clutch size (number of eggs) for each nest in each year. The data are in a file called (`sduBirds.csv`) and are raw data collected while visiting the nests. The data will need to be processed to answer those questions.

Let's import the data and take a look at it. Make sure your data looks OK before moving on. You should first set up your working directory (e.g. a folder for the course, with a sub-folder for course data etc.), and set it (with `setwd`)). See the earlier material for how to do this, or ask for help.

```
df <- read.csv("CourseData/sduBirds.csv")
str(df)
```

```
## 'data.frame':    9357 obs. of  15 variables:
##  $ Timestamp  : chr  "2013-05-14" "2013-05-03" "2013-06-25" "2013-06-18" ...
##  $ Year       : int  2013 2013 2013 2013 2013 2013 2013 2013 2013 2013 ...
##  $ Day        : int  134 123 176 169 112 112 116 183 143 107 ...
##  $ boxNumber  : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ species    : chr  "BT" "BT" "BT" "BT" ...
##  $ stage      : chr  "NL" "NL" "NE" "NE" ...
##  $ nEggs      : int  12 8 0 0 0 0 0 0 NA 0 ...
##  $ nLiveChicks: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ nDeadChicks: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ eggStatus  : chr  "WA" "CO, CV" NA NA ...
##  $ chickStatus: chr  NA NA NA NA ...
##  $ adultStatus: chr  "FN" "FN" NA NA ...
##  $ finalStatus: chr  NA NA "NE" "NE" ...
##  $ Comments   : chr  NA "MA" NA NA ...
##  $ observerID : chr  "AMK" "AMK" "AMK" "AMK" ...
```

6.1 select

From the `str` summary (above) you can see that there are many columns in the data set and we only need some of them. Let's `select` only the columns that we need for our calculations to make things a bit easier to handle. We need the `species`, `Year`, `Day`, `boxNumber` and `nEggs`:

```
df <- select(df, species, Year, Day, boxNumber, nEggs)
head(df)
```

```
##   species Year Day boxNumber nEggs
## 1      BT 2013 134         1    12
## 2      BT 2013 123         1     8
## 3      BT 2013 176         1     0
## 4      BT 2013 169         1     0
## 5      BT 2013 112         1     0
## 6      BT 2013 112         1     0
```

The output of `head` shows you the first few rows of the data set. You can see that each row represents a visit of a researcher to a particular nest. The researcher records the bird species if it is known (GT = Great tit, BT = Blue tit, NH = Nuthatch etc.), and then records the number of eggs, number of chicks, activity of the adults and so on. We need to convert this huge dataset into one which contains clutch size for each nest, for each year of the study.

The information given by `str` (above) shows that there are data on species other than our target species. We are only interested in the great tits and blue tits so we can first filter the others out using the `species` variable.

We can check what the make up of this part of the data is using the `table` function which will count up all of the entries.

```
table(df$species)
```

```
##
##   BT   GT   MT   NH   WR
##  992 4612   73   31    5
```

6.2 filter

Now let's filter this data and double check that this has worked:

```
df <- filter(df,species %in% c("GT","BT"))
table(df$species)
```

```
##
##   BT   GT
##  992 4612
```

You will notice that all the levels of the variable are retained. This is not a problem, and can usually be ignored. You can also tidy this up using the `droplevels` function, which removes all unused factor levels.

```
df <- droplevels(df)
table(df$species)
```

```
##
##   BT   GT
##  992 4612
```

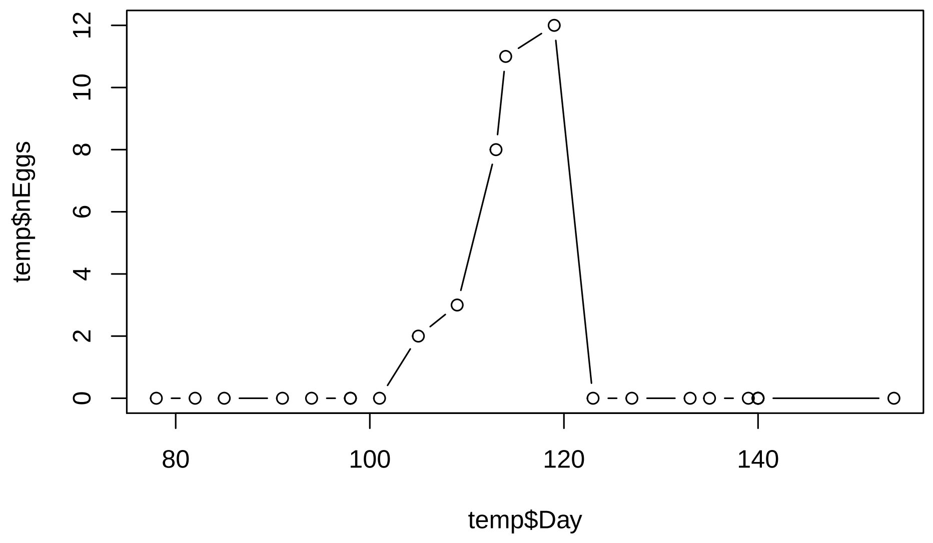
6.3 arrange

Recall that the data are records of visits to each nest a few times per week. To ensure that the data are in time order I can first `arrange` by first `Year` and then `Day`. To illustrate this we can make a temporary data set (called `temp`) to look at a particular nest in a particular year to get a record of the progress for that particular nest, and then plot it (this is an ugly plot and we will learn how to make beautiful ones soon):

```
df <- arrange(df,Year,Day)

temp <- filter(df,boxNumber == 1,Year == 2014)
max(temp$nEggs) # get the max value
```

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

```
plot(temp$Day,temp$nEggs,type="b")
```



Eggs are usually laid one per day, and the clutch size is the maximum number of eggs reached for each nest box. In this case, the clutch size is 12 eggs. The rapid decline in number of eggs after this peak value shows when the eggs have hatched and the researcher finds chicks instead of eggs!

6.4 `summarise` and `group_by`

The next part is the crucial part of our investigation. We need to get the maximum number of eggs seen at each nest. Of course we could repeatedly use `filter`, followed by `max`, for each nest-year combination but this would be incredibly tedious.

Instead, we will use the `dplyr` function, `summarise`, to do this by asking for the maximum value of `nEggs`. To make this work we need to first use the `group_by` function tell R to group the data by the variables we are interested in. If we don't do this we just get the overall maximum. We can ungroup the data using the `ungroup` function.

Because there are missing data (`NA` values) we need to specify `na.rm = TRUE` in the argument.

So first, let's get the max per species, just to illustrate how this works:

```
df <- group_by(df,species)
summarise(df,clutchSize = max(nEggs,na.rm= TRUE))
```

```
## # A tibble: 2 x 2
##   species clutchSize
##   <chr>        <int>
## 1 BT              14
## 2 GT              14
```

We can see how the data are grouped by asking for a summary:

```
summary(df)
```

```
##    species               Year           Day          boxNumber
##  Length:5604        Min.   :2013   Min.   : 60.0   Min.   :  1.00
##  Class :character   1st Qu.:2014   1st Qu.:116.0   1st Qu.: 29.75
##  Mode  :character   Median :2014   Median :133.0   Median : 57.00
##                     Mean   :2015   Mean   :133.2   Mean   : 55.83
##                     3rd Qu.:2017   3rd Qu.:148.0   3rd Qu.: 85.00
##                     Max.   :2019   Max.   :212.0   Max.   :101.00
##
##      nEggs
##  Min.   : 0.000
##  1st Qu.: 0.000
##  Median : 0.000
##  Mean   : 2.326
##  3rd Qu.: 4.000
##  Max.   :14.000
##  NA's   :613
```

We can ungroup the data again like this:

```
df <- ungroup(df)
```

So both species lay the same maximum number of eggs, but maybe this is just caused by outliers for one of the species. We'll need to dig deeper.

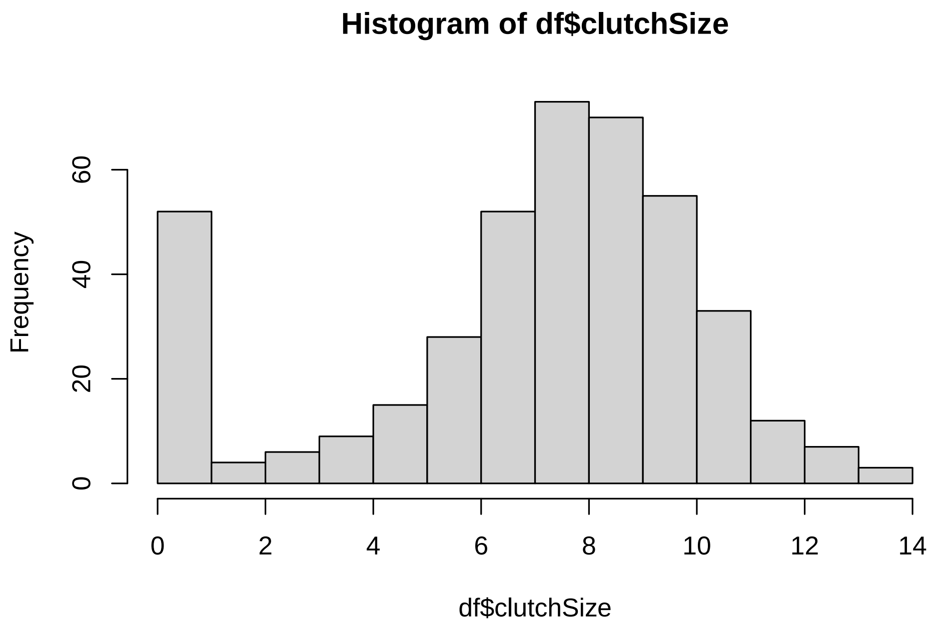
How can we calculate the average? We cannot simply ask for the `mean` because the data run through time following the development in each nest. We need to calculate the maximum `nEggs` for each nest, and then calculate the average of those. We can do this in two steps.

We first calculate the clutch size for each box for each species in each year:

```r
df <- group_by(df, species, Year, boxNumber)
df <- summarise(df, clutchSize = max(nEggs,na.rm= TRUE))
```

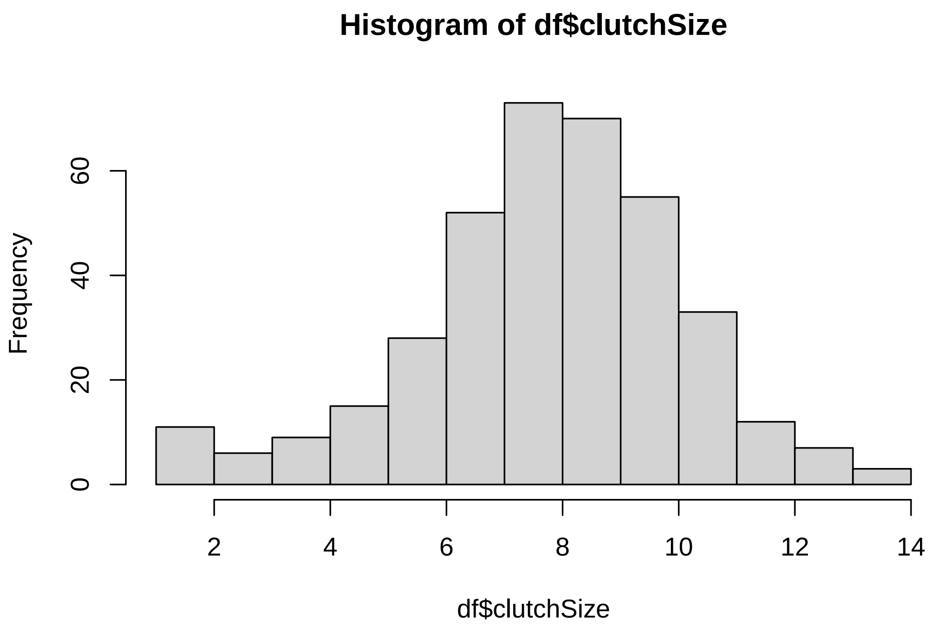
Let's first look at all the clutch size data:

```r
hist(df$clutchSize)
```



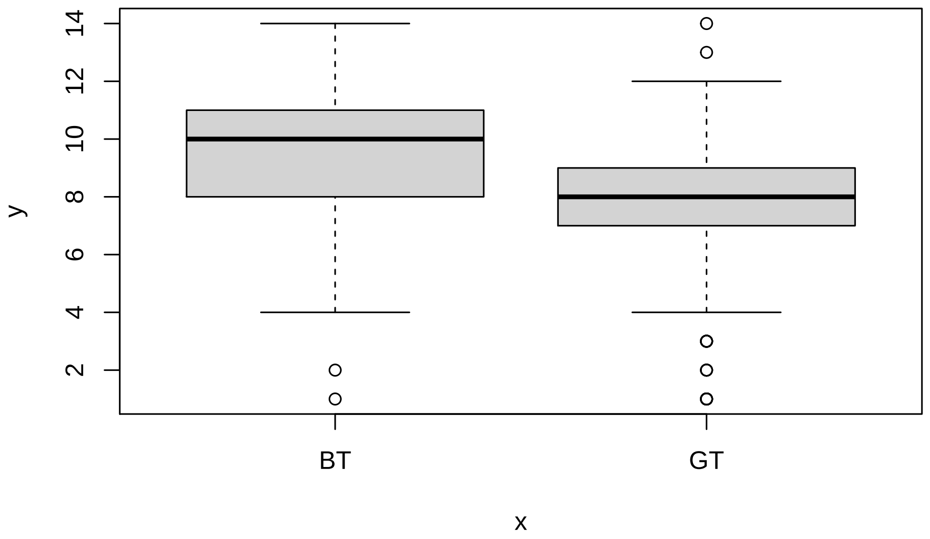
You can see here that there are a lot of zero values. This is because nests were recorded even if they did not attempt to lay eggs. We should remove these from our data using `filter` again:

```r
df <- filter(df, clutchSize > 0)
hist(df$clutchSize)
```



That looks better. Now we can plot them again but this time split apart the species (again - this plot is ugly and we'll learn to plot nicer ones soon).

```
plot(as.factor(df$species),df$clutchSize)
```



From these distributions it looks like the average clutch size is greater in the blue tit. We can use `summarise` to calculate the means.

```
df <- group_by(df,species)
summarise(df, mean = mean(clutchSize),sd = sd(clutchSize))
```

```
## # A tibble: 2 x 3
##   species  mean    sd
##   <chr>   <dbl> <dbl>
## 1 BT       9.64  2.64
## 2 GT       7.89  2.19
```

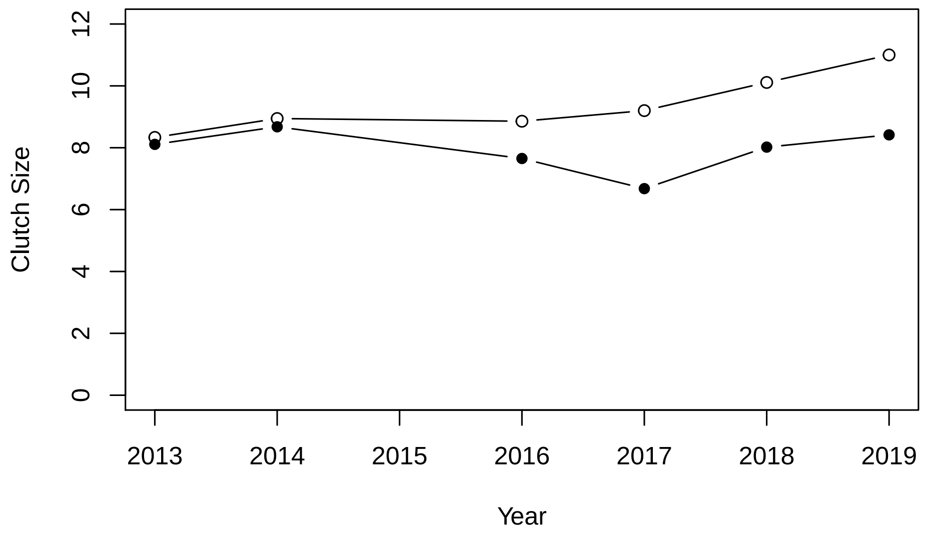
Lets now turn to the other question - how does the clutch size vary with year?

```
df <- group_by(df,species,Year)
df2 <- summarise(df, meanClutchSize = mean(clutchSize))
head(df2)
```

```
## # A tibble: 6 x 3
## # Groups:   species [1]
##   species  Year meanClutchSize
##   <chr>   <int>          <dbl>
## 1 BT       2013           8.33
## 2 BT       2014           8.94
## 3 BT       2016           8.86
## 4 BT       2017           9.2
## 5 BT       2018          10.1
## 6 BT       2019          11
```

We can plot this by first making a plot for blue tits, and then adding the points for great tits. I have used the `pch` argument to use filled circles for the great tits:

```
plot(df2$Year[df2$species == "BT"],df2$meanClutchSize[df2$species == "BT"],type="b",
     ylim=c(0,12),xlab="Year",ylab="Clutch Size")
points(df2$Year[df2$species == "GT"],df2$meanClutchSize[df2$species == "GT"],
       pch = 16, type="b")
```



So it looks like the clutch size varies a fair amount from year to year, but that generally blue tits have large clutch sizes than great tits.

6.5 Using pipes, saving data.

I have walked you through a step-by-step data manipulation. During that process you made made (and replaced) new data sets at each step. In practice this can be done more smoothly using *pipes* (%>%) to pass the result of one function into the next, and the next, and the next…

You'll get some more practice with this as we go on. Below I show how do do this to create a clutch size data set from the raw data (note that your file path will differ from mine):

```
#Import and process data using pipes
SDUClutchSize <- read.csv("CourseData/sduBirds.csv") %>%
  filter(species %in% c("GT","BT")) %>% #include only GT and BT
  droplevels() %>% #drop unwanted factor levels
  select(species, Year, Day, boxNumber, nEggs) %>% #select columns needed
  group_by(species, Year, boxNumber) %>% #group data
  summarise(clutchSize = max(nEggs,na.rm=TRUE)) %>% #calculate clutch size (max eggs)
  filter(clutchSize > 0)

head(SDUClutchSize)
```

```
## # A tibble: 6 x 4
## # Groups:   species, Year [1]
##   species  Year boxNumber clutchSize
##   <chr>   <int>     <int>      <dbl>
## 1 BT       2013         1         12
## 2 BT       2013         5          8
## 3 BT       2013        27         10
## 4 BT       2013        31          6
## 5 BT       2013        35         10
## 6 BT       2013        37          8
```

You can save this out using the `write.csv` function. You will need to set the argument `row.names = FALSE` to stop the data including row names.

```
write.csv(x = SDUClutchSize,file = "CourseData/SDUClutchSize.csv",row.names = FALSE)
```

6.6 Exercise: Wrangling the Amniote Life History Database

In this exercise the aim is to use the "Amniote Life History Database" [1] to investigate some questions about life history evolution.

The questions are: (1) what are the records and typical life spans in different taxonomic classes? [*what is the longest, shortest and median life span in birds, mammals and reptiles?*] (2) is there a positive relationship between body mass and life span? [*do big species live longer than small ones?*]; (3) is there a trade-off between reproductive effort and life span? [*do species that reproduce a lot have short lives, so there is a negative relationship between reproduction and life span?*]; (4) is this trade-off universal across all Classes? [*does the trade-off exist in birds, reptiles and amphibians?*]

The database is in a file called `Amniote_Database_Aug_2015.csv` in the course data folder. The missing values (which are normally coded as `NA` in R) are coded as "`-999`". The easiest way to take care of this is to specify this when we import the data using the `na.strings` argument of the `read.csv` function. Thus we can import the data like this:

```r
amniote <- read.csv("CourseData/Amniote_Database_Aug_2015.csv",na.strings = "-999")
```

Let's make a start…

1. When you have imported the data, use `dim` to check the dimensions of the whole data frame (you should see that there are 36 columns and 21322 rows). Use `names` to look at the names of all columns in the data in `amniote`.

2. We are interested in longevity (lifespan) and body size and reproductive effort and how this might vary depending on the taxonomy (specifically, with Class). Use `select` to pick relevant columns of the dataset and discard the others. Call the new data frame `x`. The relevant columns are the taxonomic variables (`class`, `genus` & `species`) and `longevity_y`, `litter_or_clutch_size_n`, `litters_or_clutches_per_y`, and `adult_body_mass_g`.

3. Take a look at the first few entries in the `species` column. You will see that it is only the *epithet*, the second part of the *Genus_species* name, that is given.
Use `mutate` and `paste` to convert the `species` column to a *Genus_species* by pasting the data in `genus` and `species` together. To see how this works, try out the following command, `paste(1:3, 4:6)`. After you have created the new column, remove the `genus` column (using `select` and `-genus`).

4. What is the longest living species in the record? Use `arrange` to sort the data from longest to shortest longevity (`longevity_y`), and then look at

[1] https://esajournals.onlinelibrary.wiley.com/doi/10.1890/15-0846R.1

the top of the file using `head` to find out. (hint: you will need to use reverse sort (`-`)). Cut and paste the species name into Google to find out more!

5. Do the same thing but this time find the shortest lived species.

6. Use `summarise` and `group_by` to make a table summarising `min`, `median` and `max` life spans (`longevity_y`) for the three taxonomic classes in the database. Remember that you need to tell R to remove the `NA` values using a `rm.rm = TRUE` argument.

7. Body size is thought to be associated with life span. Let's treat that as a hypothesis and test it graphically. Sketch what would the graph would look like if the hypothesis were true, and if it was false. Plot `adult_body_mass_g` vs. `longevity_y` (using base R graphics). You should notice that this looks a bit messy.

8. Use `mutate` to create new `log`-transformed variables, `logMass` and `logLongevity`. Use these to make a "log-log" plot. You should see that makes the relationship more linear, and easier to "read".

9. Is there a trade-off between reproductive effort and life span? Think about this as a hypothesis - sketch what would the graph would look like if that were true, and if it was false. Now use the data to test that hypothesis: Use `mutate` to create a variable called `logOffspring` which is the logarithm of number of litters/clutches per year multiplied by the number of babies in each litter/clutch . Then plot `logOffspring` vs. `logLongevity`.

10. To answer the final question (differences between taxonomic classes) you could now use `filter` to subset to particular classes and repeat the plot to see whether the relationships holds universally.

Remember that if you struggle you can check back to previous work where you have used `dplyr` commands to manipulate data in a similar way. If you get truly stuck, ask for help from instructors or fellow students.

Chapter 7

Combining data sets

Handling data is often not limited to single data sets. One common task is to combine two (or more) datasets together. For example, one dataset might include a set of observations from a field study, while another might have information about the weather throughout the study period, or site-specific information. It is therefore useful to be able to combine these datasets to add the information from the second table to the first table.

R does this with the `dplyr` function `join`. In the next section you will first learn how `join` works by following an example that asks a research question that can only be answered by two data sets. After that, you will work on your own to do a similar analysis without explicit instructions (i.e. you will need to figure out how to apply the method to new data.

7.1 Using `join`

By following this example you will learn how to combine two data sets to create a new one of combined data to answer a conservation-related question: "**Does threat status vary with species' generation times?**"

This question is crucial to conservation biologists because it helps us to generalise our ideas about what drives extinction risks. In other words, if we can say "*species with slow life histories tend to be more threatened*" then this gives useful information that can help with planning. For example, imagine we have some species that have not yet been assessed (we don't know if they are threatened or not). Should we focus attention on the one with a short generation time, or long generation time?

To answer the question we will need to import two large data sets, tidy them up a bit and then combine them for analysis.

Let's start with the "Amniote Life History Database" [1], which is a good source of life history data. We have encountered this database before. Recall that the

[1]https://esajournals.onlinelibrary.wiley.com/doi/10.1890/15-0846R.1

missing values (which are normally coded as `NA` in R) are coded as "`-999`". The easiest way to take care of this is to specify this when we import the data using the `na.strings` argument of the `read.csv` function. Thus we can import the data like this:

```
amniote <- read.csv("CourseData/Amniote_Database_Aug_2015.csv",na.strings = "-999")
```

We can filter on the taxonomic `class` to subset to only mammals. Then, to address our question, we want data on generation time for mammals. Generation time is often measured as the average age at which females reproduce so we can get close to that with `female_maturity_d`. We will first `select` these columns, along with `genus` and `species`. We can combine these two taxonomic variables using `mutate` and `paste` to get our Latin binomial species name.

We have previously learned that log transforming such variables is a good thing to do, so we can use `mutate` again to do this transformation.

Finally, we can use `na.omit` to get rid of entries with missing values (which we cannot use). This is not essential, but keeps things more manageable.

```
mammal <- amniote %>%
  filter(class == "Mammalia") %>% #get the mammals only
  select(genus, species, female_maturity_d) %>% #get useful columns
  mutate(species = paste(genus, species)) %>%
  select(-genus) %>%
  mutate(logMaturity = log(female_maturity_d)) %>%
  na.omit()
```

Let's take a quick look at what we have:

```
head(mammal)
```

```
##                      species female_maturity_d logMaturity
## 20         Echinops telfairi         278.42000    5.629131
## 22    Hemicentetes nigriceps          48.57000    3.883006
## 23 Hemicentetes semispinosus          46.19892    3.832956
## 27          Microgale dobsoni        669.59200    6.506669
## 40         Microgale talazaci        639.00000    6.459904
## 47            Setifer setosus        198.00000    5.288267
```

Looks good. Now let's import the IUCN Red List data.

```
redlist <- read.csv("CourseData/MammalRedList.csv")
```

Let's take a look at that.

```
names(redlist)
```

```
##  [1] "Species.ID"              "Kingdom"
##  [3] "Phylum"                  "Class"
##  [5] "Order"                   "Family"
##  [7] "Genus"                   "Species"
##  [9] "Authority"               "Infraspecific.rank"
## [11] "Infraspecific.name"      "Infraspecific.authority"
## [13] "Stock.subpopulation"     "Synonyms"
## [15] "Common.names..Eng."      "Common.names..Fre."
## [17] "Common.names..Spa."      "Red.List.status"
## [19] "Red.List.criteria"       "Red.List.criteria.version"
## [21] "Year.assessed"           "Population.trend"
## [23] "Petitioned"
```

```
unique(redlist$Red.List.status)
```

```
## [1] "DD" "LC" "CR" "NT" "EN" "VU" "EX" "EW"
```

There's a lot of information there but what we really need is simply the Latin binomial (for which we need `genus` and `species`) and the threat status `Red.List.status`.

R treats categorical variables (`factor` variables) as alphabetical, but in this case the red list status has a meaning going from low threat (Least Concern - LC) to Critically Endangered (CR) and even Extinct in the Wild (EX) at the other end of the spectrum. We can define this ordering using `mutate` with the `factor` function.

```
redlist <- redlist %>%
  mutate(species = paste(Genus, Species))%>%
  select(species, Red.List.status) %>%
  mutate(Red.List.status = factor(Red.List.status,
                                  levels = c("LC","NT","VU","EN","CR","EW","EX")))

head(redlist)
```

```
##                species Red.List.status
## 1   Abditomys latidens            <NA>
## 2    Abeomelomys sevia              LC
## 3 Abrawayaomys ruschii              LC
## 4   Abrocoma bennettii              LC
## 5 Abrocoma boliviensis              CR
## 6      Abrocoma budini            <NA>
```

Now we can combine this with the life history data from above using `left_join`.

```
x <- left_join(mammal,redlist,by = "species")
```

Let's take a look at what we have now:

```
head(x)
```

```
##                     species female_maturity_d logMaturity Red.List.status
## 1          Echinops telfairi         278.42000    5.629131              LC
## 2     Hemicentetes nigriceps          48.57000    3.883006              LC
## 3 Hemicentetes semispinosus          46.19892    3.832956              LC
## 4          Microgale dobsoni         669.59200    6.506669              LC
## 5         Microgale talazaci         639.00000    6.459904              LC
## 6            Setifer setosus         198.00000    5.288267              LC
```

```
summary(x)
```

```
##    species          female_maturity_d  logMaturity    Red.List.status
##  Length:2000        Min.   :  23.81   Min.   :3.170   LC     :1219
##  Class :character   1st Qu.: 121.53   1st Qu.:4.800   VU     : 176
##  Mode  :character   Median : 344.12   Median :5.841   EN     : 168
##                     Mean   : 574.92   Mean   :5.745   NT     : 114
##                     3rd Qu.: 696.38   3rd Qu.:6.546   CR     :  66
##                     Max.   :6391.56   Max.   :8.763   (Other):  10
##                                                       NA's   : 247
```

You can see that there are 247 missing values for the Red List status. These are either species that have not yet been assessed, or maybe where there are mismatches in the species names between the two databases. We will ignore this problem today.

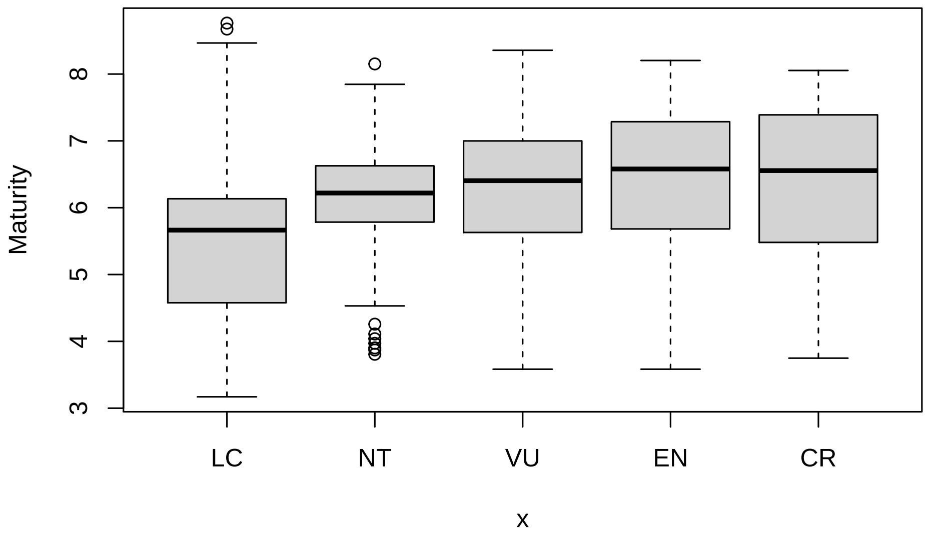
Before plotting, I will also use `filter` remove species that are extinct (status = "EX" and "EW"). To do this I use the `%in%` argument to allow me to match a vector of variables. Because I want to NOT match them I negate the match using `!`.

I then ensure that those levels are removed from the variable using `droplevels`.

```
x <- x %>%
  filter(!Red.List.status %in% c("EX","EW")) %>%
  droplevels()
```

Let's now plot the data to answer the question.

```
plot(x$Red.List.status,x$logMaturity,ylab="Maturity")
```



What can we see? If you focus on the median values, it looks like there is a weak positive relationship between this life history trait and threat status: animals with slower life histories tend to be more threatened.

7.2 Using pivot_longer

Sometimes data are not arranged in a way that make them easy to use in R. For example, data could be arranged so that the column headings are themselves data (e.g. treatments, sex of individuals).

To illustrate this I will use a data set on heights of men and women.

```
heights <- read.csv("CourseData/heights.csv")
```

Let's take a look:

```
heights
```

```
##      place heightMale heightFemale
## 1   London        170          164
## 2   London        176          158
## 3   London        179          157
## 4   London        166          158
## 5   London        177          153
## 6   London        177          155
## 7   London        173          151
## 8   London        173          155
## 9   London        173          159
## 10  London        171          158
## 11  London        173          166
## 12  London        171          156
## 13  London        172          157
## 14  London        175          159
## 15  London        179          156
## 16 Bristol        175          156
## 17 Bristol        173          156
## 18 Bristol        171          155
## 19 Bristol        172          158
## 20 Bristol        185          158
## 21 Bristol        176          153
## 22 Bristol        173          158
## 23 Bristol        173          156
## 24 Bristol        177          156
## 25 Bristol        172          159
## 26 Bristol        169          162
## 27 Bristol        177          167
## 28 Bristol        171          157
## 29 Bristol        175          166
## 30 Bristol        171          155
```

I would like to make a boxplot, but it is not possible to easily do it with the data arranged in this format. What I need to do is “unpack” or rearrange the data to add another column for sex. This will make the data frame twice as long, and less wide.

There is a convenient function called `pivot_longer` in the `tidyr` package that will do this for you. You can “pipe” data into the funtion, then tell it which columns you would like to move, and then give it the name of the new column that contains data that **was** in the column heading, and the name of the column containing the data.

```
newHeights <- heights %>%
  pivot_longer(cols = c(heightMale,heightFemale),names_to = c("Sex"),values_to = "Hei

newHeights
```

```
## # A tibble: 60 x 3
##    place  Sex          Height
##    <chr>  <chr>         <int>
##  1 London heightMale      170
##  2 London heightFemale    164
##  3 London heightMale      176
##  4 London heightFemale    158
##  5 London heightMale      179
## 6 London heightFemale    157
## 7 London heightMale      166
## 8 London heightFemale    158
## 9 London heightMale      177
## 10 London heightFemale   153
## # ... with 50 more rows
```

We are nearly done. But this is not perfect because the names in the `Sex` column are not right. We can fix this in a couple ways. Here's one easy way using the function `gsub`. The `gsub` function ("**g**eneral **sub**stution) finds text and replaces it with other text. In this case we want to find"height" and replace it with nothing ("").

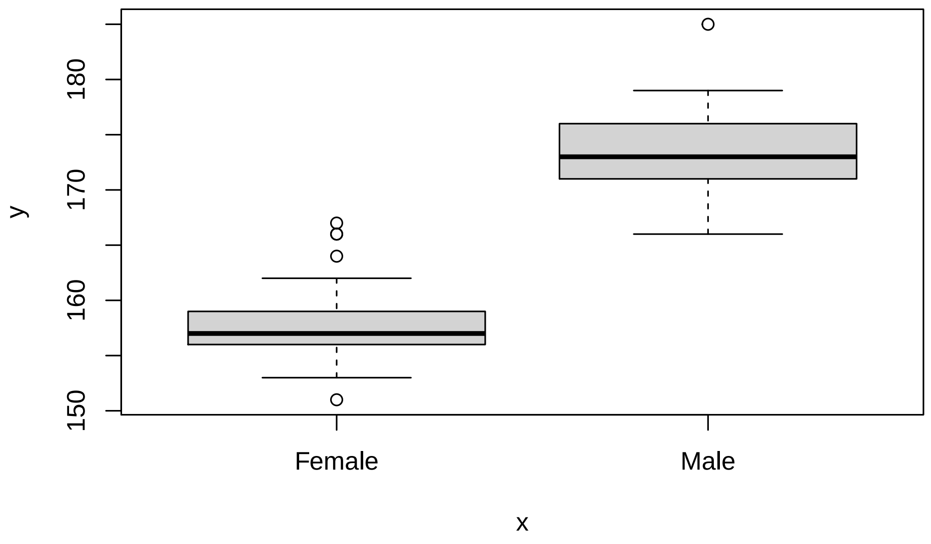
So we can now complete the job with a `mutate` command, and make sure it is recognised as a categorical variable (a `factor`, like this:

```
newHeights <- heights %>%
  pivot_longer(cols = c(heightMale,heightFemale),names_to = c("Sex"),values_to = "Height") %>%
  mutate(Sex = gsub(pattern = "height",replacement = "",x = Sex)) %>%
  mutate(Sex = as.factor(Sex))

newHeights
```

```
## # A tibble: 60 x 3
##    place  Sex    Height
##    <chr>  <fct>   <int>
##  1 London Male      170
##  2 London Female    164
##  3 London Male      176
##  4 London Female    158
##  5 London Male      179
##  6 London Female    157
##  7 London Male      166
##  8 London Female    158
##  9 London Male      177
## 10 London Female    153
## # ... with 50 more rows
```

Now we can plot those data more easily

```
plot(newHeights$Sex,newHeights$Height)
```



This data manipulation is useful surprisingly often.

7.3 Exercise: Temperature effects on egg laying dates

Data have been collected on great tits (musvit) at SDU for several years. Your task today is to analyse these data to answer the question: *is egg laying date associated with spring temperature?* The idea here is that warmer springs will lead to delayed egg laying which could have negative consequences to the population if their caterpillar food source doesn't keep pace with the change.

You are provided with two data sets: one on the birds and another on weather. You will need to process these using tools in the `dplyr` package, and combine them (using `left_join`) for analysis.

The first data set, `eggDates.csv`, is data from the SDU birds project. The data are arranged in columns where each column is a year and each row is a nest. The data in each column is the day of the year that the first egg in the nest was laid.

These data do NOT fulfill the "tidy data" standard where each variable gets a column. In this case,a single variable (first egg date) gets many columns (one for each year), and column headers are data (the years). The data will need to be processed before you can analyse it.

You will need to use `pivot_wider` to fix this issue so that you produce a version of the data with three columns - `nestNumber`, `Year` and `dayNumber`.

The second dataset, `AarslevTemperature.csv`, is a weather dataset from Årslev near Odense. This dataset includes daily temperatures records for several years. You will need to `summarise` this data to obtain a small dataset that has the weather of interest - average temperature in the months of February to April for each year.

To answer the question, you will need to join these data sets together.

1. Import the data and take a look at it with `head` or `str`.
2. Use `pivot_longer` to reformat the data. This might take a bit of trial and error - don't give up!

Maybe this will help: The first argument in the `pivot_longer` command (`cols`) tells R which columns contain the data you are interested in (in this case, these are `y2013`,`y2014` etc). Then the `names_to` argument tells R what you want to name the new column from this data (in this case, `Year`). Then, the `values_to` argument tells R what the data column should be called (e.g. `Day`). In addition, there is a useful argument called `names_prefix` that will remove the part of the column name (e.g. the `y` of `y2013`)

You should also make sure that the `Year` column is recognised as being a numeric variable rather than a character string. You can do this by adding a command using `mutate` and `as.numeric`, like this `mutate(Year = as.numeric(Year))`

You should end up with a dataset with three columns as described above.

3. Calculate the mean egg date per year using `summarise` (remember to `group_by` the year first). Take a look at the data.
4. Import the weather data and take a look at it with `head` or `str`.
5. Use `filter` subset to the months of interest (February-April) and then `summarise` the data to calculate the mean temperature in this period (remember to `group_by` year). Look at the data. You should end up with a dataset with two columns - `year` and `meanSpringTemp`.
6. Join the two datasets together using `left_join`. You should now have a dataset with columns `nestNumber`, `Year`, `dayNumber` and `meanAprilTemp`
7. plot a graph of `meanAprilTemp` on the x-axis and `dayNumber` on the y-axis.

Now you should be able to answer the question we started with: is laying date associated with spring temperatures.

Part II

Data visualisation

Chapter 8

Visualising data with ggplot

In this chapter you will be guided through using the `ggplot2` package to make some pretty plots. You will therefore need the `ggplot2` package to make this work. Remember, you can load packages like this:

```r
library(ggplot2)
```

We will use the SDU birds clutch size data that we produced at the end of the "[Data wrangling with dplyr]" chapter for these examples. You can find the data set via the Course Data Dropbox link.

> Remember to set your working directory, and start a new script. I am assuming that you have saved your data in a folder called "CourseData" inside your working directory.

```r
df <- read.csv("CourseData/SDUClutchSize.csv")
```

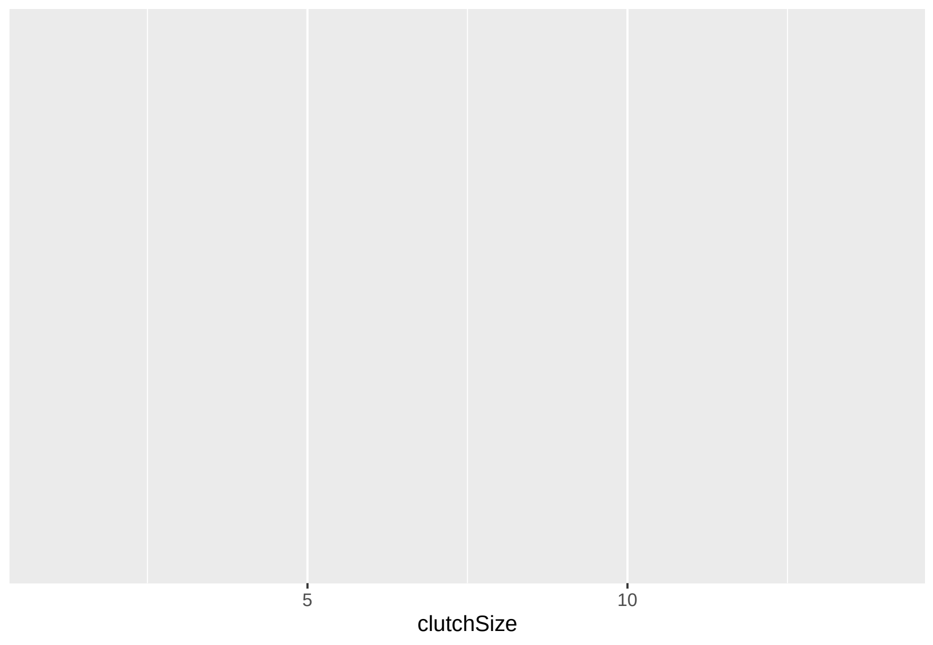
8.1 Histograms

The `ggplot` function expects two main arguments (1) the data and (2) the **aesthetics**. The aesthetics are the variables you want to plot, and associated characteristics like colours, groupings etc. The first argument is for the data, then the aesthetics are specified within the `aes(...)` argument. These usually include an argument for `x` which is normally the variable that appears on the horizontal axis, and (often) `y` which is usually the variable on the vertical axis. The details of this depend on the type of plot you are making.

After setting up the plot the graphics are added as **geometric layers** or **geoms**. There are many of these available including `geom_histogram`, `geom_line`, `geom_point` etc.

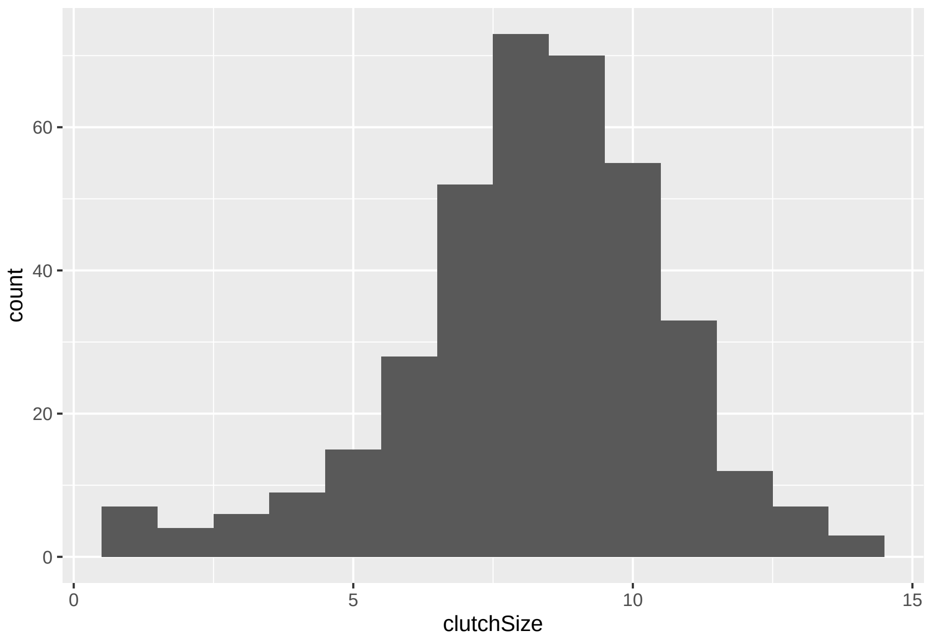
I will illustrate the construction of a simple plot by making a histogram of the clutch size of all the nests in the dataset.

```r
ggplot(df, aes(x = clutchSize))
```



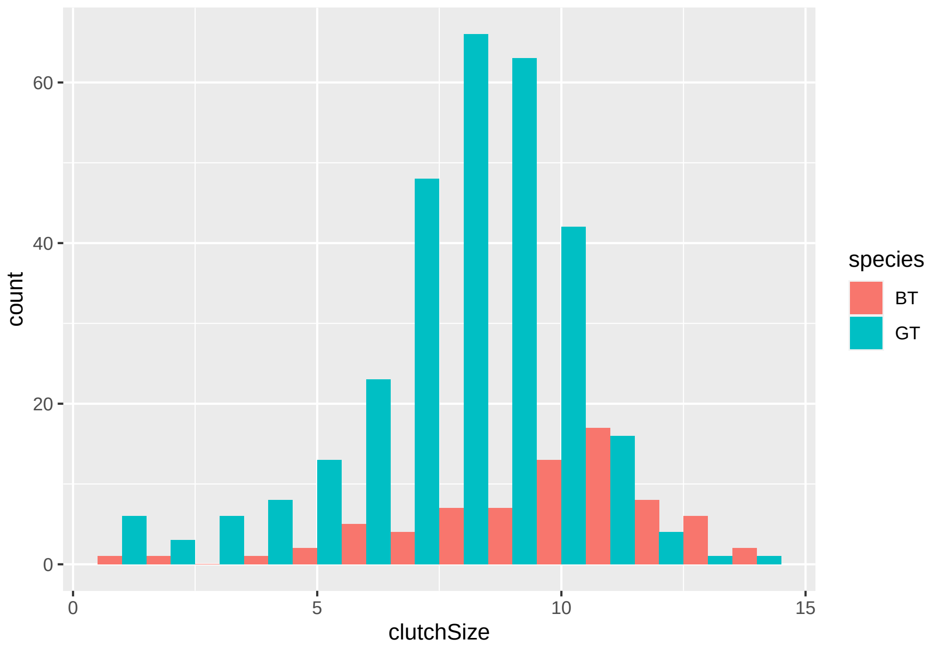
This produces an empty plot because we have not yet specified what kind of plot we want. We want a histogram, so we can add this as follows. I have set `binwidth` to be 1 because we know we are dealing with counts between just 1 and 14. Try altering the `binwidth`.

```r
ggplot(df, aes(x = clutchSize)) +
  geom_histogram(binwidth = 1)
```



We know that we have two species here and we would like to compare them. This is done within the aesthetic argument. The default is that the bars for different categories are stacked on top of each other. This is good in some cases, but probably not here.

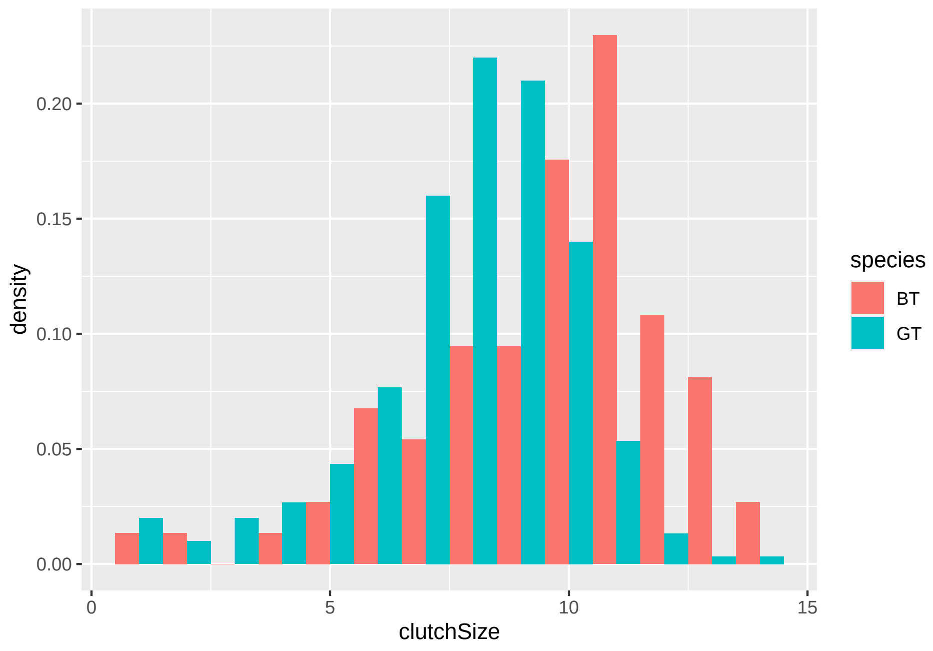
```
ggplot(df, aes(x = clutchSize,fill = species)) +
  geom_histogram(binwidth = 1,position = "dodge")
```



You can immediately see that there are far fewer blue tit nests than great tit ones. But you can also see that the center of mass for blue tits is further to the right than great tits.

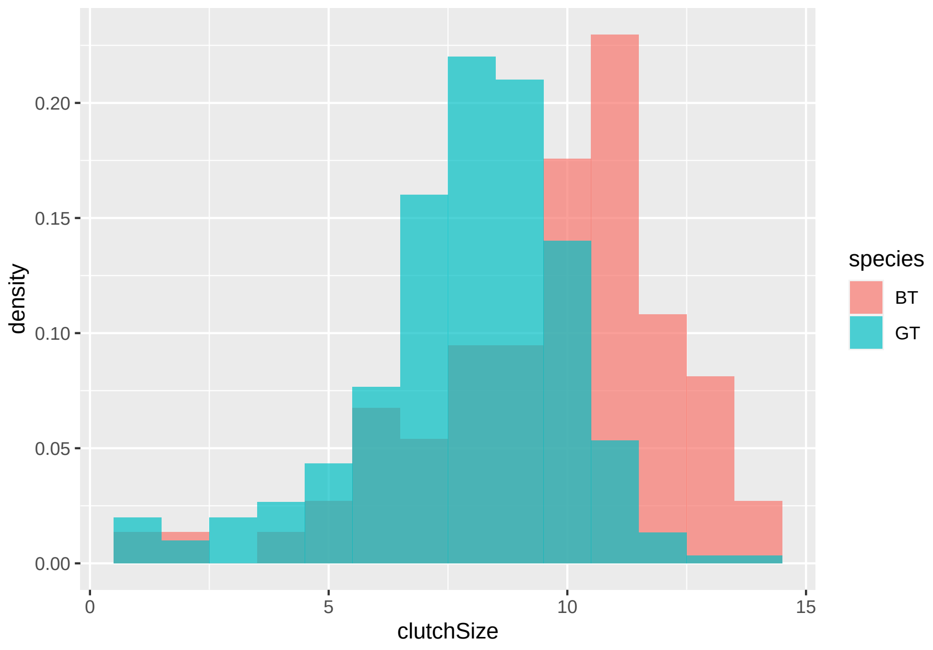
To make it easier to compare distributions with very different counts, we can put density on the y-axis instead of the default count using the argument `stat(density)`.

```
ggplot(df, aes(x = clutchSize, fill = species, stat(density))) +
  geom_histogram(binwidth = 1,position = "dodge")
```



An alternative approach would be to overlay the two sets of bars (using `position = "identity"`) and set the colours to be slightly transparent (using `alpha = 0.7`) so that you can see the overlapping region clearly.

```
ggplot(df, aes(x = clutchSize, fill = species, stat(density))) +
  geom_histogram(binwidth = 1,position = "identity",alpha=0.7)
```

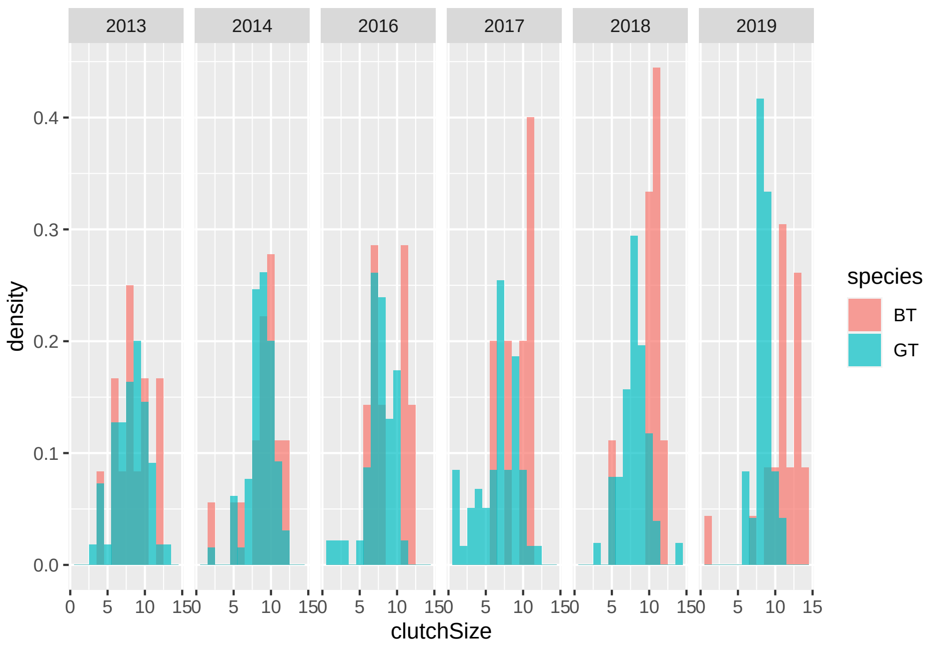


It is very clear from this plot that blue tits tend to have bigger clutch sizes than great tits. Is this difference *statistically significant*? We will look at testing this in a future class - for now we will be satisfied with our visualisation.

8.2 "Facets" - splitting data across panels

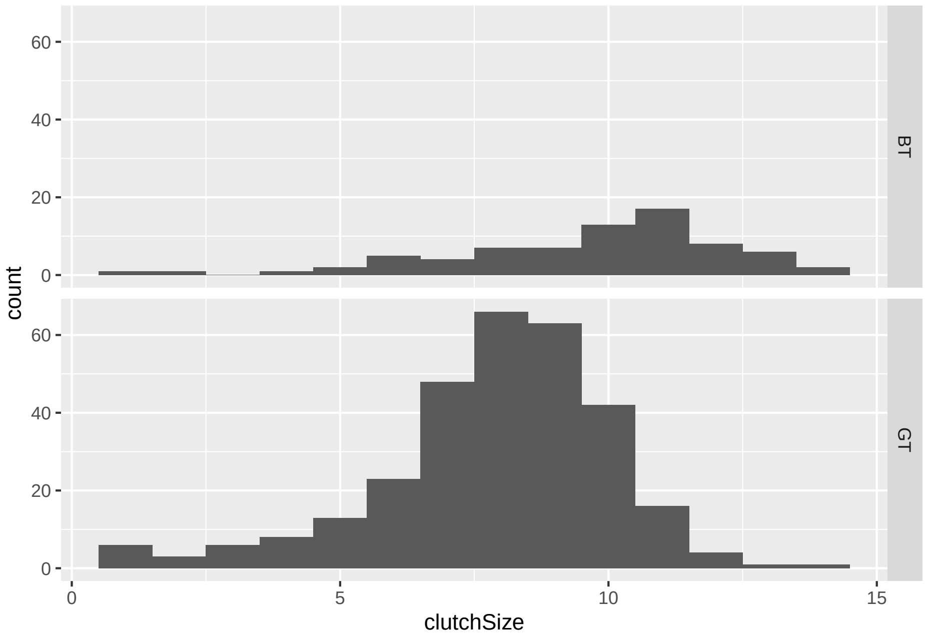
You should recall that there were several years of data represented here. `ggplot` has a very clever way of splitting up the plot to examine this.

```
ggplot(df, aes(x = clutchSize, fill = species, stat(density))) +
  geom_histogram(binwidth = 1,position = "identity",alpha=0.7) +
  facet_grid(.~Year)
```



You could split the data up by species in a similar way, as yet another way of visualising the difference between species:

```
ggplot(df, aes(x = clutchSize)) +
  geom_histogram(binwidth = 1) +
  facet_grid(species~.)
```



You can change whether the separate graphs are presented in a rows or columns by changing the order of the argument: `facet_grid(species~.)` or `facet_grid(.~species)`. Try it.

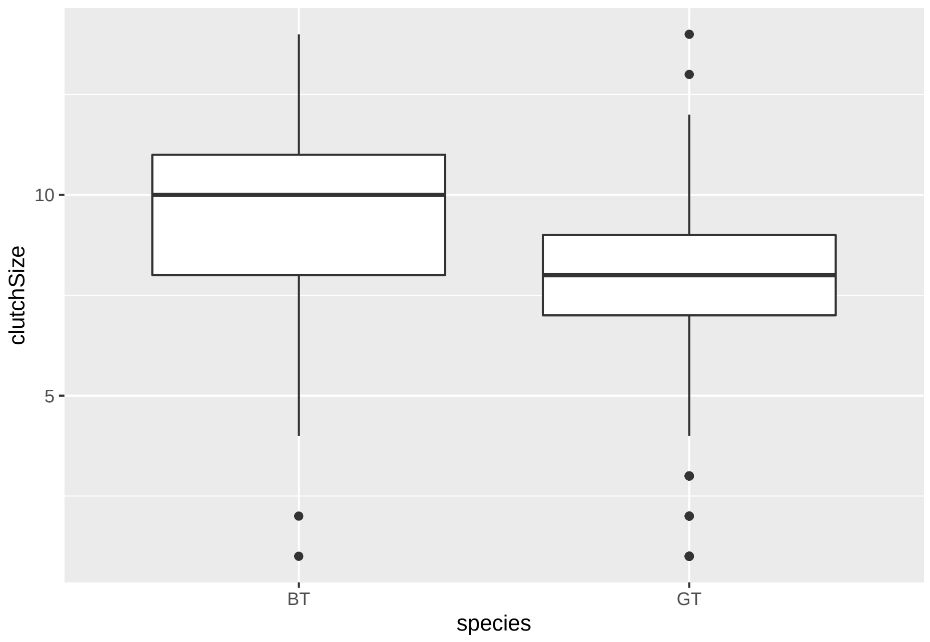
8.3 Box plots

Box plots are suitable for cases where one variable is categorical with 2+ levels, and the other is continuous. Therefore, another way to look at these distributions is to use a box plot.

In a box plot the box shows the quartiles (i.e. the 25% and 75% quantiles) within which 50% of the data are found. The horizontal line in the box is the *median*, Then the whiskers extend from the smallest to largest value *unless they are further than 1.5 times the interquartile range (the length of the box)* away from the edge of the box, in which case they are individually shown as outlier points.

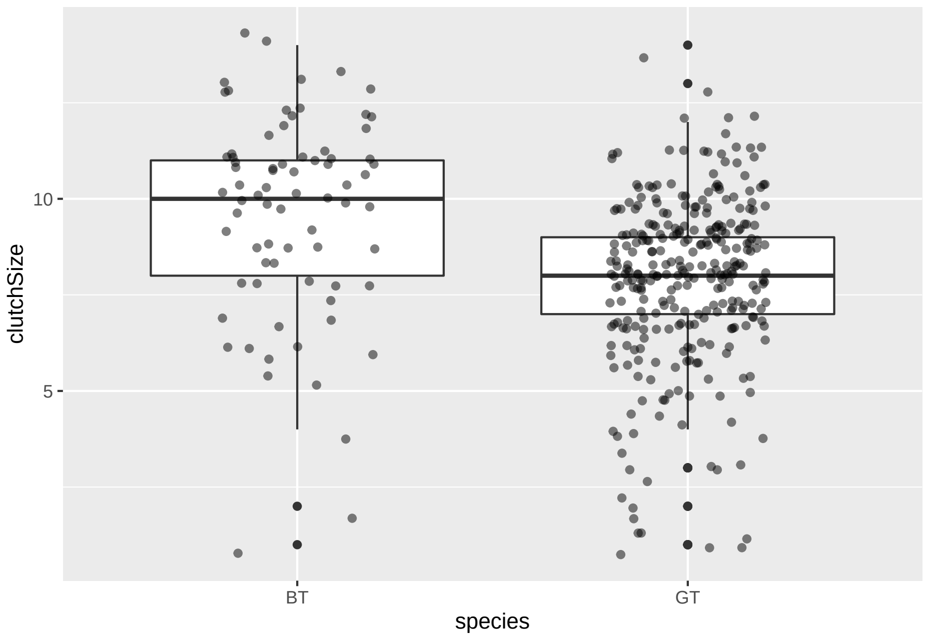
To plot them using `ggplot` you must use a `geom_boxplot` layer. The categorical variable is normally placed on the x-axis so is placed as `x` in the `aes` argument, while the continuous variable is on the `y` axis.

```
ggplot(df, aes(x = species, y = clutchSize)) +
        geom_boxplot()
```



Some researchers argue that it is a good idea to add the data as points to these plots as “full disclosure” of what the underlying data look like. These can be added with a `geom_jitter` layer (jitter is random noise added in this case to the horizontal axis). You should set `width` and `alpha` arguments to make it look nice.

```
ggplot(df, aes(x = species, y = clutchSize)) +
  geom_boxplot() +
  geom_jitter(width = .2, alpha = 0.5, colour="black",fill="black")
```



Try splitting the data into different years using `facet_grid` with the box plot.

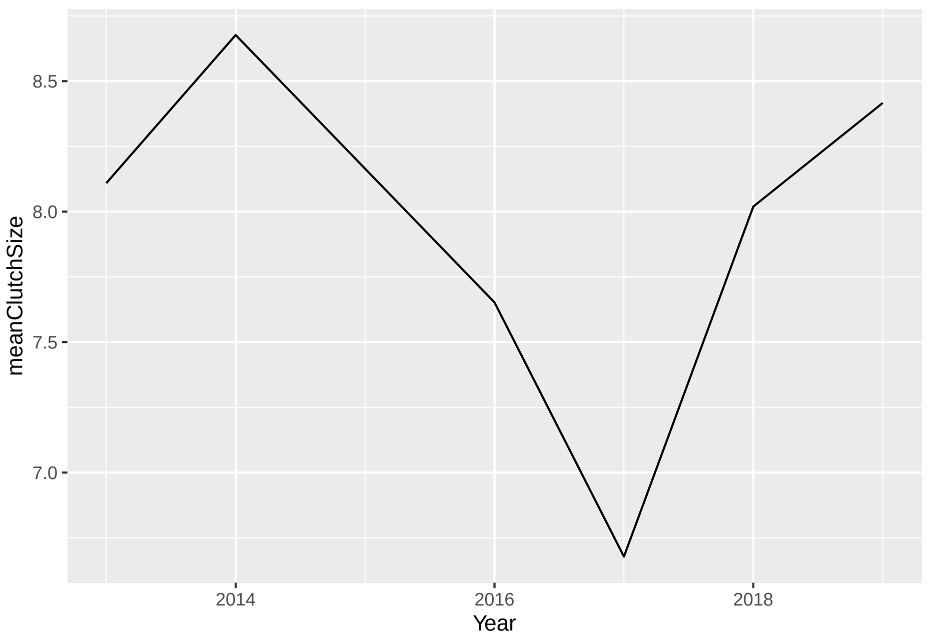
8.4 Lines and points

Perhaps not surprisingly lines and points can be added with the geoms, `geom_line` and `geom_point` respectively. To illustrate this we will make a plot showing how clutch size changes among years. First we will use `summarise` to create a dataset with the mean clutch size. We'll start simply, by looking at only great tits.

```
GTclutch <- df %>%
  filter(species == "GT") %>%
  group_by(Year) %>%
  summarise(meanClutchSize = mean(clutchSize))
```

Then you can plot this like this.

```
ggplot(GTclutch, aes(x = Year, y = meanClutchSize)) +
        geom_line()
```

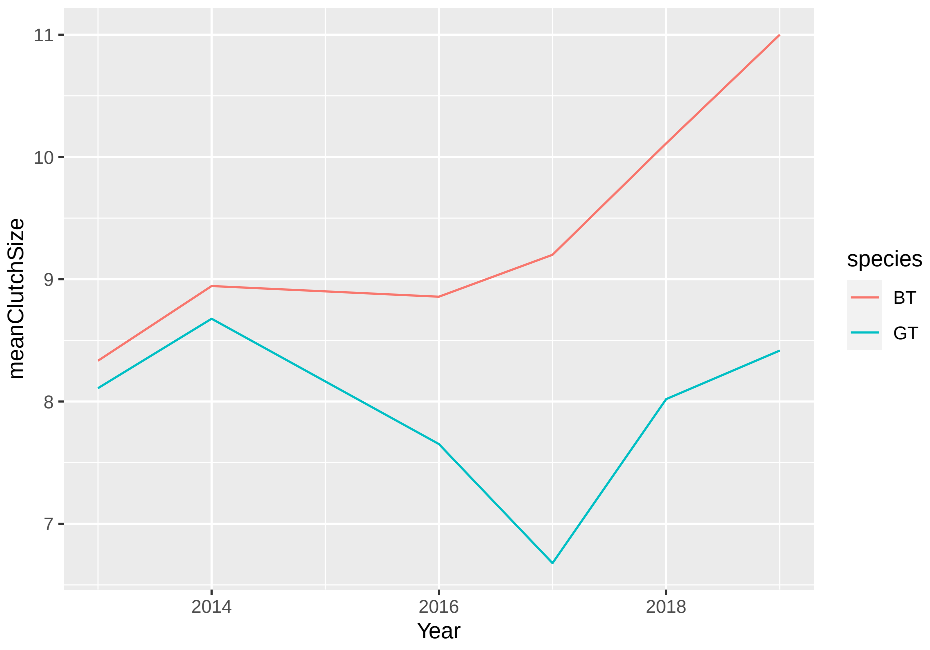


I think this looks OK, but we should add both species. I'll first need to produce a mean clutch size dataset that includes both species.

```
meanClutch <- df %>%
  group_by(species,Year) %>%
  summarise(meanClutchSize = mean(clutchSize))
```

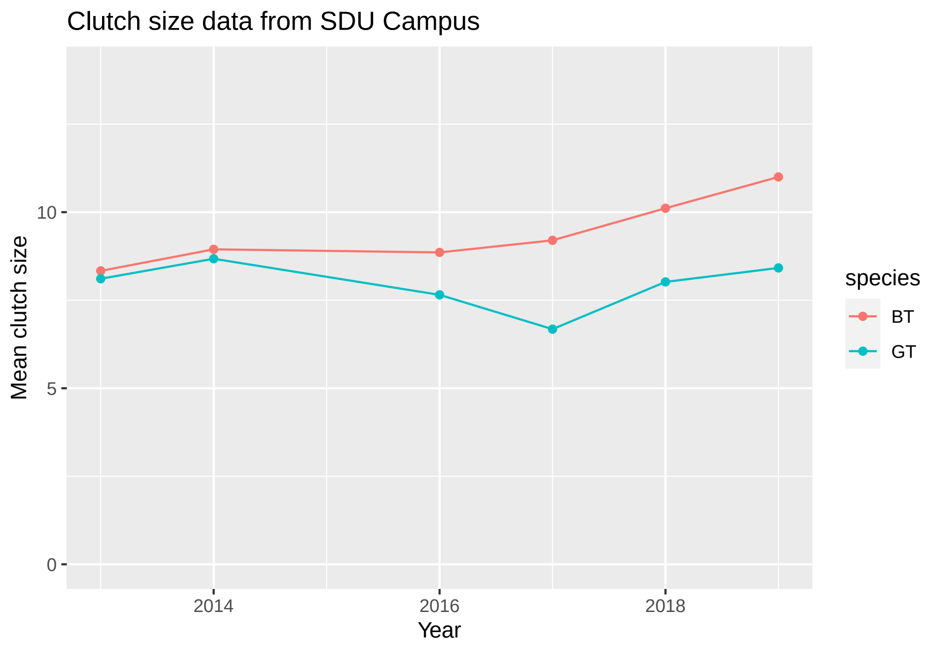
Now I can do the plot again. The only difference to the command is that I need to tell R that I want to colour the lines by species (`colour = species`).

```
ggplot(meanClutch, aes(x = Year, y = meanClutchSize, colour = species)) +
        geom_line()
```



I can improve on this by (1) changing the y axis limits (using `ylim`) so that it goes through the full range of my data (0 - 14); (2) adding points (using a `geom_point` layer) where my actual data values are; (3) adding a nicely formatted axis label (using `ylab`); adding a title (`ggtitle`)

```
ggplot(meanClutch, aes(x = Year, y = meanClutchSize, colour = species)) +
  geom_line() +
  geom_point() +
  ylim(0,14) +
  ylab("Mean clutch size") +
  ggtitle("Clutch size data from SDU Campus")
```



8.5 Scatter plots

Finally, lets make a scatter plot. The SDU bird data are not suitable for this type of plot so we'll use the data from a few days ago on suburban bird diversity.

```
df <- read.csv("CourseData/suburbanBirds.csv")
```

Take a look at the data to remind ourselves what it looks like

```
head(df)
```

```
##          Name Year HabitatIndex nIndividuals nSpecies
## 1    Alamotos 1946         10.0           48       12
## 2      Ramona 1946          9.5           30       13
## 3      Verona 1947          9.5           38       15
## 4 Valle Vista 1950          9.5           42       11
## 5    La Gonda 1955         11.0           44       13
## 6     Belgian 1956          9.0           27       14
```

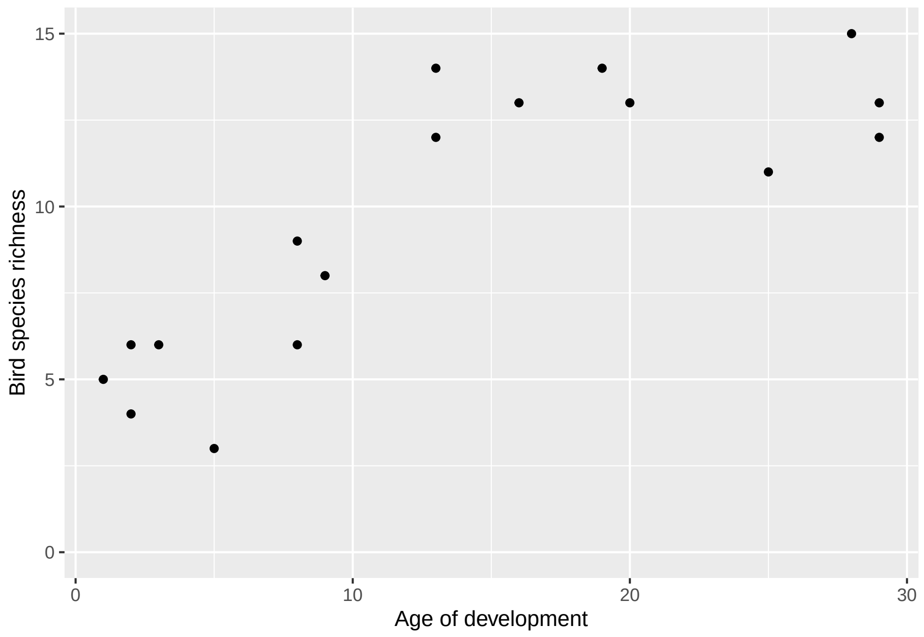
These data show the result of standardised bird surveys at housing developments of different ages in California. The surveys were carried out in 1975, and the data includes the `Year` and number of individual birds seen `nIndividuals` and number of species seen `nSpecies`. The question being addressed is "How does the age of the housing development affect the number of species?"

To investigate this we should first add a new variable for `Age` to the data set. We can do this using the `mutate` function from `dplyr`. This function creates new variables, for example by manipulating existing ones.

```
df <- mutate(df,Age = 1975 - Year)
```

When we have created this variable we can plot the data. For aesthetic reasons I also would like to set the limits on the y-axis to go extend to zero, and I would like to include proper labels on the axes.

```
ggplot(df, aes(x = Age,y = nSpecies)) +
  geom_point() +
  ylim(0,15) +
  xlab("Age of development") + ylab("Bird species richness")
```



This shows very clearly that older developments have more species, but it also appears to show that there is an asymptote around 13 species.

Compare this plot to the one you made with base graphics in a previous class.

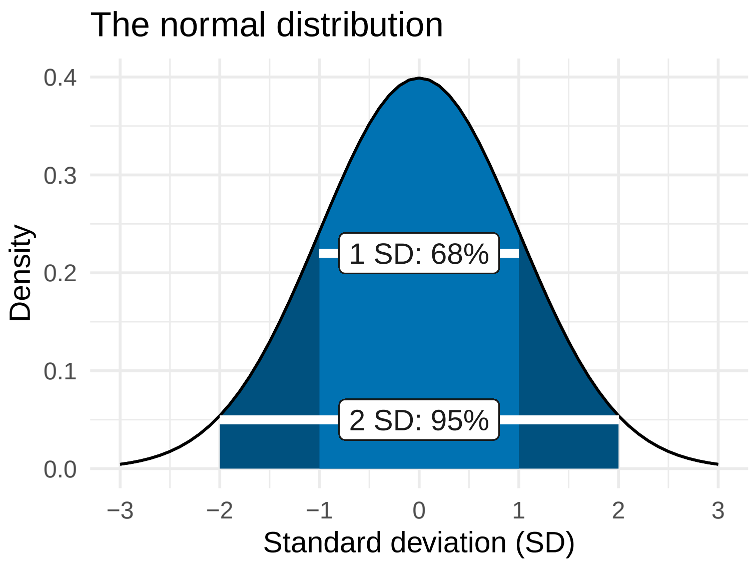
Chapter 9

Distributions and summarising data

This chapter covers two broad topics: the concept of statistical distributions and summarising data. It ends with a brief look at the "law of large numbers".

9.1 Distributions

A statistical distribution is a description of the relative number of times (the *frequency*) possible outcomes will occur if repeated samples were to be taken. They are important because (1) they are useful descriptors of data and (2) they form the basis for assumptions in some statistical approaches. For example, statistical analyses often assume a normal distribution. The normal distribution is symmetrical (centered on the mean) and 68% of observations fall within 1 standard deviation (s.d.), and 95% of observations fall within 2 s.d..



We will use R to simulate some distributions, and explore these to get a feel for them. R has functions for generating random numbers from different kinds of distributions. For example, the function `rnorm` will generate numbers from a normal distribution and `rpois` will generate numbers from a Poisson distribution.

9.2 Normal distribution

The `rnorm` function has three arguments. The first argument is simply the number of values you want to generate. Then, the second and third arguments specify the the mean and standard deviation values of the distribution (i.e. where the distribution is centered and how spread out it is).

The following command will produce 6 numbers from a distribution with a mean value of 5 and a standard deviation of 2.

```
rnorm(6,5,2)
```

```
## [1] 0.8860899 5.9511727 0.4674129 6.2696362 3.0969106 5.3058671
```

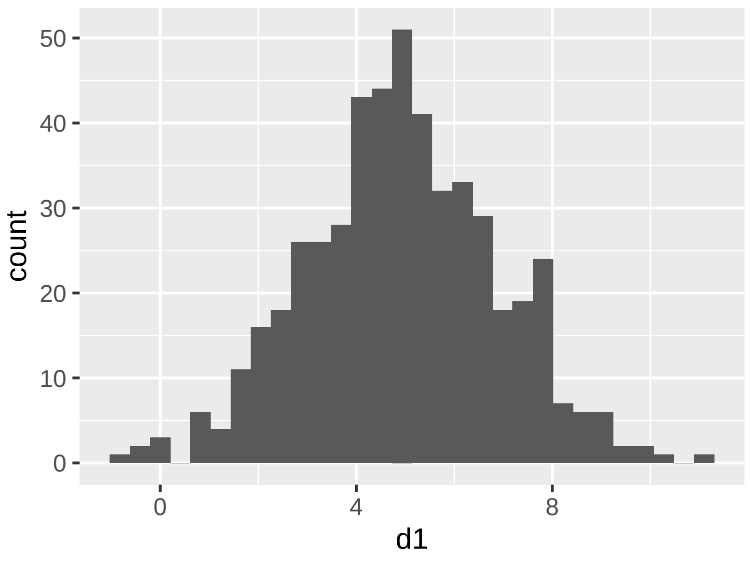
Try changing the values of the arguments to alter the number of values you generate, and to alter the mean and standard deviation.

Let's use this to generate a larger data frame, and then place markers for the various measures of "spread" onto a plot. *Note that here I put a set of parentheses around the plot code to both display the result AND save the plot as an R object called `p1`*

```
rn <- data.frame(d1 = rnorm(500,5,2))
summary(rn) #Take a look
```

```
##        d1
##  Min.   :-0.9862
##  1st Qu.: 3.6789
##  Median : 4.9241
##  Mean   : 4.9399
##  3rd Qu.: 6.2712
##  Max.   :10.9317
```

```
#Plot the data
(p1 <- ggplot(rn,aes(x=d1)) +
  geom_histogram()
  )
```



We can calculate the mean and standard deviation using `summarise` (along with other estimates of "spread"). The mean and standard deviation values will be close (but not identical) to the values you set when you generated the distribution.

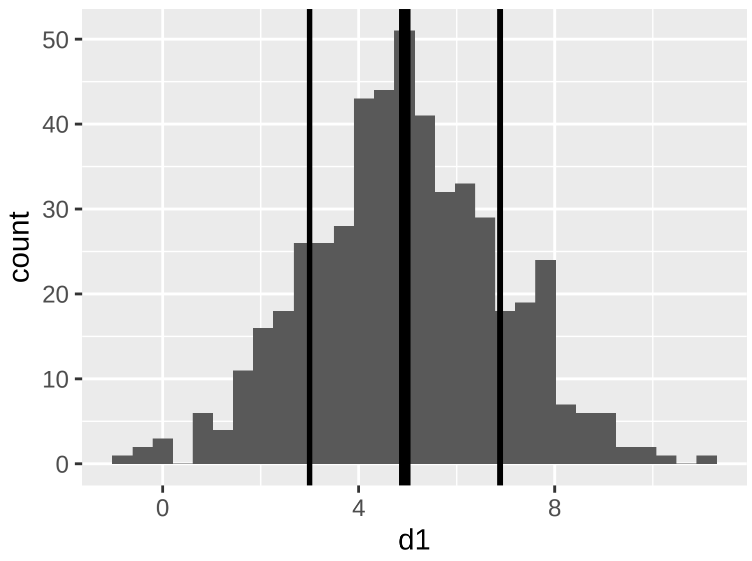
Note that here I put a set of parentheses around the code to both display the result AND save the result in an object called `sv`

```
(sv <- rn %>%
  summarise(meanEst = mean(d1),
            sdEst = sd(d1),
            varEst = var(d1),
            semEst = sd(d1)/sqrt(n()))
 )
```

```
##    meanEst    sdEst   varEst     semEst
## 1 4.939908 1.944357 3.780523 0.08695428
```

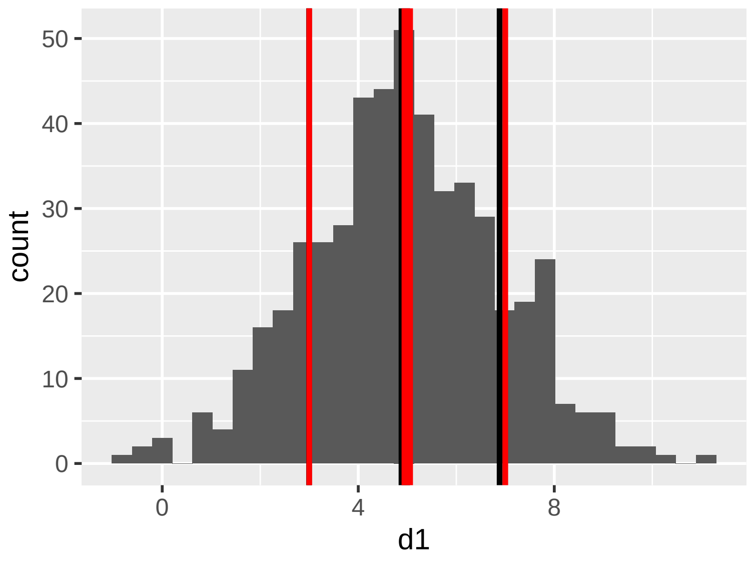
Let's use the function `geom_vline` to add some markers to the plot from above to show these values...

```
(p2 <- p1 +
  geom_vline(xintercept = sv$meanEst, size=2) + #mean
  geom_vline(xintercept = sv$meanEst+sv$sdEst, size=1) + #upperSD
  geom_vline(xintercept = sv$meanEst-sv$sdEst, size=1) #lowerSD
)
```



We can compare these with the true values (the values we set when we generated the data), by adding them to the plot in a different colour (mean=5, sd=2).

```
(p3 <- p2 +
  geom_vline(xintercept = 5, size=2, colour="red") + #mean
  geom_vline(xintercept = 5+2, size=1,colour="red") + #upperSD
  geom_vline(xintercept = 5-2, size=1,colour="red") #lowerSD
 )
```



Try repeating these plots with data that has different sample sizes. For example, use sample sizes of 5000, 250, 100, 50, 10. What do you notice? You should notice that for smaller sample sizes, the true distribution is not captured very well.

When you calculate the mean and standard deviation, you are actually fitting a simple model: the mean and standard deviation are parameters of the model, which assumes that the data follow a normal distribution.

Try adding lines for the standard error of the mean to one of your histograms.

9.3 Comparing normal distributions

Because normal distributions all have the same shape, it can be hard to grasp the effect of changing the distribution's parameters viewing them in isolation. In this section you will write some code to compare two normal distributions. This approach can be useful when considering whether a proposed experiment will successfully detect a difference between treatment groups. We'll look at this topic, known as "power analysis", in greater detail in a later class. For now we will simply use `ggplot` to get a better feel for the normal distribution.

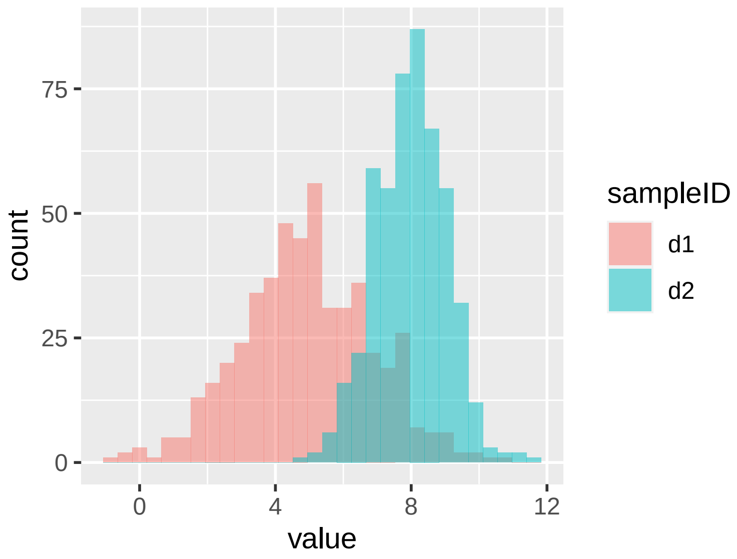
Let's use `rnorm` to generate a larger data frame with two sets of numbers from different distributions: (d1: mean = 5, sd = 2; d2: mean = 8, sd = 1).

```
rn <- data.frame(d1 = rnorm(500,5,2),d2 = rnorm(500,8,1))
summary(rn)
```

```
##        d1                d2        
##  Min.   :-0.9862   Min.   : 4.628  
##  1st Qu.: 3.6789   1st Qu.: 7.305  
##  Median : 4.9241   Median : 8.000  
##  Mean   : 4.9399   Mean   : 7.978  
##  3rd Qu.: 6.2712   3rd Qu.: 8.719  
##  Max.   :10.9317   Max.   :11.495
```

The summaries (above) show that the mean and the width of the distributions vary, but we should always plot our data. So lets make a plot in `ggplot`. In the dataset I created I have the data arranged by columns side-by-side, but `ggplot` needs the values to be arranged in a single column, and the identifier of the sample ID in a second column. I can use the function `pivot_longer` to rearrange the data into the required format.

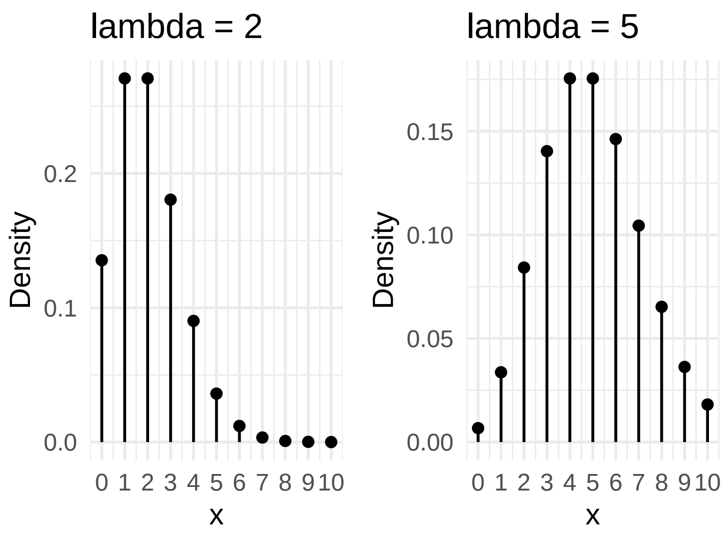
```
rn <- pivot_longer(rn,cols= c(d1,d2), names_to = "sampleID", values_to = "value") #re

#Plot histograms using "identity", and make them transparent
ggplot(rn,aes(x = value,fill=sampleID)) +
  geom_histogram(position = "identity", alpha=0.5)
```



Try changing the distributions and re-plotting them (you can change the number of samples, the mean values and the standard deviations).

9.4 Poisson distribution

The Poisson distribution is typically used when dealing with count data. The values must be whole numbers (integers) and they cannot be negative. The shape of the distributions varies with the "`lambda`" parameter. Small values of lambda give more skewed distributions.

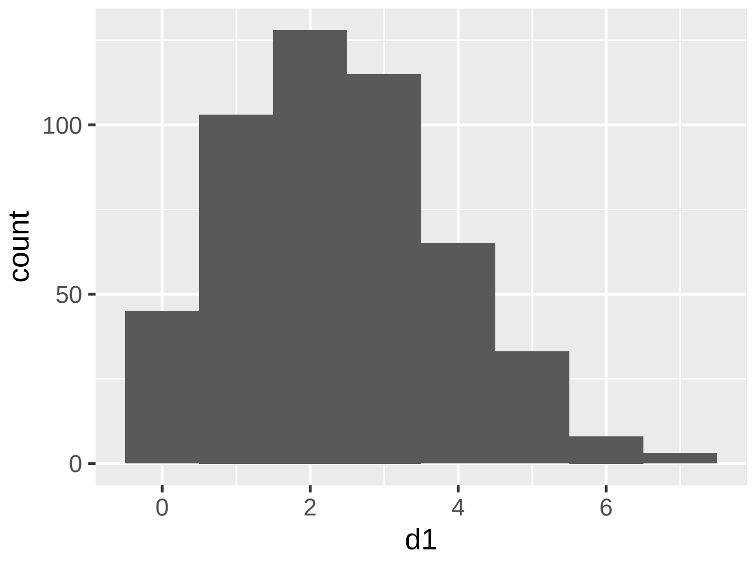


Let's generate and plot some Poisson distributed data.

```r
rp <- data.frame(d1 = rpois(500,2.4))
summary(rp) #Take a look
```

```
##        d1
##  Min.   :0.000
##  1st Qu.:1.000
##  Median :2.000
##  Mean   :2.396
##  3rd Qu.:3.000
##  Max.   :7.000
```

```r
#Plot the data
(p1 <- ggplot(rp,aes(x=d1)) +
  geom_histogram(binwidth = 1) # we know the bins will be 1
  )
```



Try changing the value of lambda and look at how the shape changes.

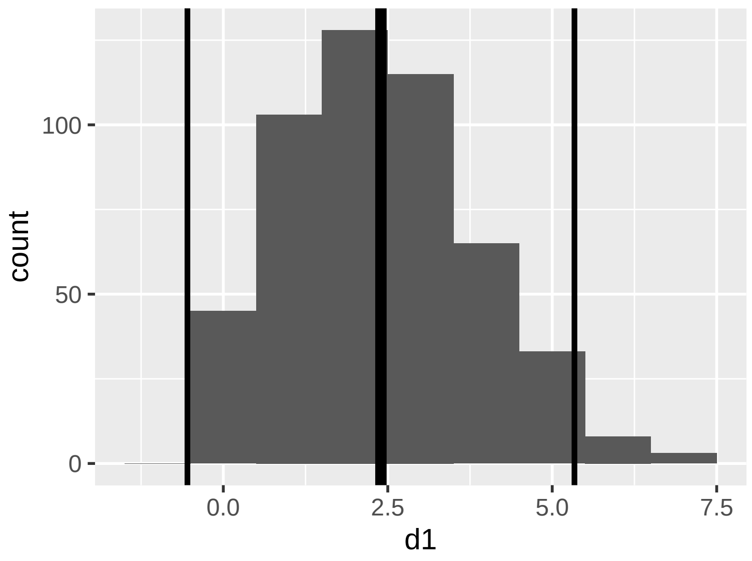
Let's calculate summary statistics of mean and standard deviation for this distribution

```
(sv <- rp %>%
  summarise(meanEst = mean(d1),
            sdEst = sd(d1))
 )
```

```
##   meanEst    sdEst
## 1   2.396 1.470888
```

Now lets plot the mean and the 2 times the standard deviation on the graph. Remember that for the normal distribution (above) that 95% of the data were within 2 times the standard deviation.

```
p1 +
  geom_vline(xintercept = sv$meanEst, size=2) + #mean
  geom_vline(xintercept = sv$meanEst+2*sv$sdEst, size=1) + #upper2SD
  geom_vline(xintercept = sv$meanEst-2*sv$sdEst, size=1) #lower2SD
```



This looks like a TERRIBLE fit: The mean is not close to the most common value in the data set and the lower limit of the standard deviation indicates we should expect some negative values - this is impossible for Poisson data. The reason for this is that mean and standard deviation, and therefore standard error, are intended for normally distributed data. When the data come from other distributions we must take another approach.

So how should we summarise this data?

One approach is to report the median as a measure of "central tendency" instead of the mean, and to report "quantiles" of the data along with the range (i.e. minimum and maximum). Quantiles are simply the cut points that divide the data into parts. For example, the 25% quantile is the point where (if the data were arranged in order) one quarter of the values would fall below; the 50% quantile would mark the middle of the data (= the median); the 75% quantile would be the point when three-quarters of the data are below. You can calculate those things using `dplyr`'s `summarise`. However, you can also simply use the base R `summary` command.

```
(sv <- rp %>%
  summarise(minVal = min(d1),
            q25 = quantile(d1,0.25),
            med = median(d1),
            q75 = quantile(d1,0.75),
            maxVal = max(d1))
 )
```

```
##   minVal q25 med q75 maxVal
## 1      0   1   2   3      7
```

```
#base R summary is just as good.
summary(rp$d1)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.000   2.000   2.396   3.000   7.000
```

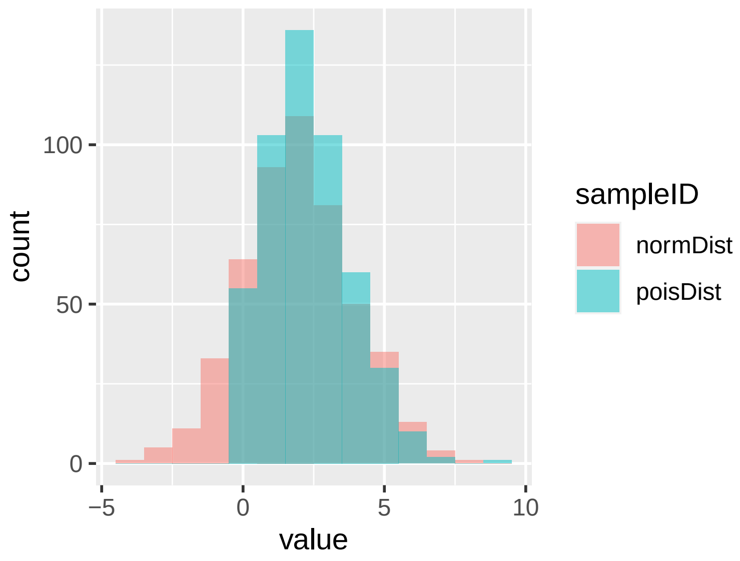
9.5 Comparing normal and Poisson distributions

To get a better feel for how these two distributions differ, lets use the same approach we used above to plot two distributions together.

```
rn <- data.frame(normDist = rnorm(500,2,2),poisDist = rpois(500,2.4))
summary(rn)
```

```
##     normDist          poisDist
##  Min.   :-3.9862   Min.   :0.000
##  1st Qu.: 0.6789   1st Qu.:1.000
##  Median : 1.9241   Median :2.000
##  Mean   : 1.9399   Mean   :2.314
##  3rd Qu.: 3.2712   3rd Qu.:3.000
##  Max.   : 7.9317   Max.   :9.000
```

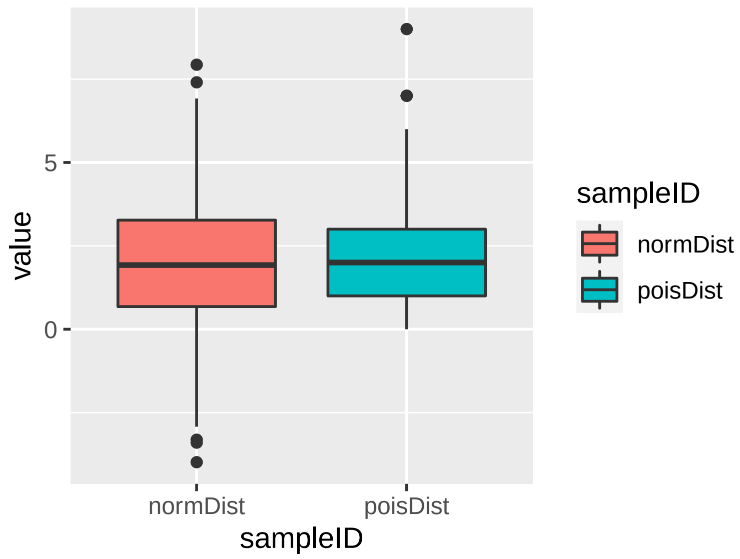
```
rn <- pivot_longer(rn,cols= c(normDist,poisDist), names_to = "sampleID", values_to =

#Plot histograms using "identity", and make them transparent
ggplot(rn,aes(x = value,fill=sampleID)) +
  geom_histogram(position = "identity", alpha=0.5,binwidth = 1)
```



Try changing the arguments in the `rnorm` and `rpois` commands to change the distributions.

Finally, let's take another view of these data and look at them using box plots. Box plots are a handy alternative to histograms and many people prefer them.

```
ggplot(rn,aes(x=sampleID, y = value,fill=sampleID)) +
  geom_boxplot()
```



You should see the main features of both distributions are captured pretty well. The normal distribution is approximately symmetrical and the Poisson distribution is skewed (one whisker longer then the other) and cannot be <0. Which graph to you prefer? (there's no right answer!)

9.6 The law of large numbers

The **law of large numbers** is one of the most important ideas in probability. It states that *As sample grows large, the sample mean converges to the population mean.* In other words, as sample size increases, you get a better idea what the true value of the mean is.

In this section you will demonstrate this law using coin tosses or dice throws. Since it is tiresome to toss coins hundreds of times it is convenient to simulate the data using **R**. Conceptually, what we are trying to do here is treat the dice rolling/coin tossing as experiments where the aim is to find the probability of getting a head/tail, or a particular number on the dice. It is useful to use dice and coins because we are pretty sure that we know what the "true" answer is: the probability of throwing a 1 on a fair dice is 1/6, while the probability of throwing a head/tail with a flipped coin is 0.5.

9.6.1 Coin flipping

Here's how to simulate a coin toss in R.

```
coinToss <- c("Heads","Tails")
sample(coinToss,1)
```

```
## [1] "Tails"
```

And here is how to simulate 6 coin tosses and make a table of the results. Note, we must use the `replace = TRUE` argument. Please ask if you don't understand why this is necessary.

```
result <- sample(coinToss,6,replace = TRUE)
table(result)
```

```
## result
## Heads Tails
##     4     2
```

We can "wrap" the `table` function with the `as.data.frame` to turn the data into a data frame that works with `ggplot` You'll probably get different results than me because this is a random process:

```
result <- data.frame(result = sample(coinToss,6,replace = TRUE))

ggplot(result,aes(x = result)) +
  geom_bar()
```

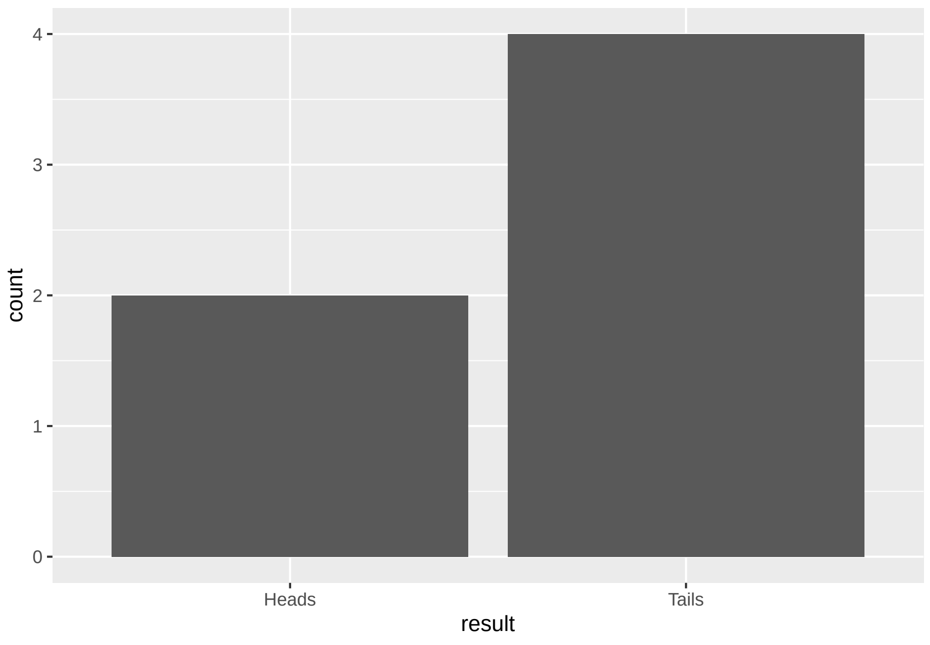


Figure 9.1: Barplot of 6 simulated coin tosses

Try this several times with small sample sizes (e.g. 4, 6, 8) and see what happens to the proportions of heads/tails. Think about what the expected outome should be. What do you notice?
Now increase the sample size (e.g. to 20, 50, 100) and see what happens to the proportions of heads/tails. What do you notice?

9.6.2 Virtual dice

Let's try the same kind of thing with the roll of (virtual) dice.

Here's how to do one roll of the dice:

```
diceRoll <- 1:6
sample(diceRoll,1)
```

```
[1] 3
```

Here's how to do 6 rolls of the dice:

```
result <- sample(diceRoll,6,replace = TRUE)
table(result)
```

```
## result
## 2 3 6
## 2 2 2
```

Your table will probably look different to this, because it is a random process. You may notice that some numbers in the table are missing if some numbers were never rolled by our virtual dice.

Now I try doing 90 rolls of the dice. I can set a value for `n` (the number of rolls), and I can also plot a horizontal line at the theoretically expected value of n/6; in this case that is 15.

Before proceeding, can you see why the expected value is n/6?:

```
n <- 90
result <- data.frame(result = sample(diceRoll,n,replace = TRUE))

ggplot(result,aes(x = result)) +
  geom_bar() +
  geom_abline(intercept = n/6,slope = 0)
```

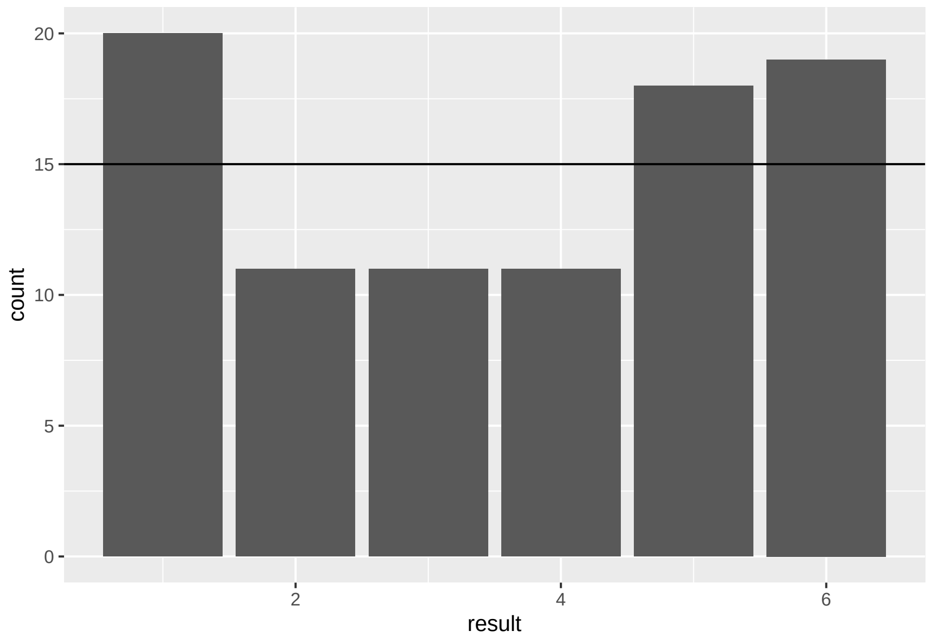


Figure 9.2: Barplot of 90 simulated dice throws

Try adjusting the code to simulate dice rolls with small (say, 30) and large (say, 600, or 6000, or 9000) samples. Observe what happens to the proportions, and compare them to the expected value.
You will notice that what would be considered a good sample size for the coin flipping (i.e. it recovers the true probability of 0.5 reasonably well) is not adequate for getting a good estimate of the probabilities for the dice. This is because of the different number of possibilities: as the range of possible outcomes increases, the sample size requirements increase.

Chapter 10

Pimping your plots

In this chapter you will learn, by following examples, how to customise plots made with `ggplot` to improve "readability", or just for aesthetic reasons.

We will cover the following:

1. Modifying axes (log transform, different tick marks/ranges etc.).
2. Colour schemes.
3. *Themes* - built-in sets of styles.
4. Multiple sub-plots in a plot.
5. Saving your plots.

For these examples I will use the dataset on animal life history, `Anage`.

```
x <- read.csv("CourseData/anage_data.csv")
```

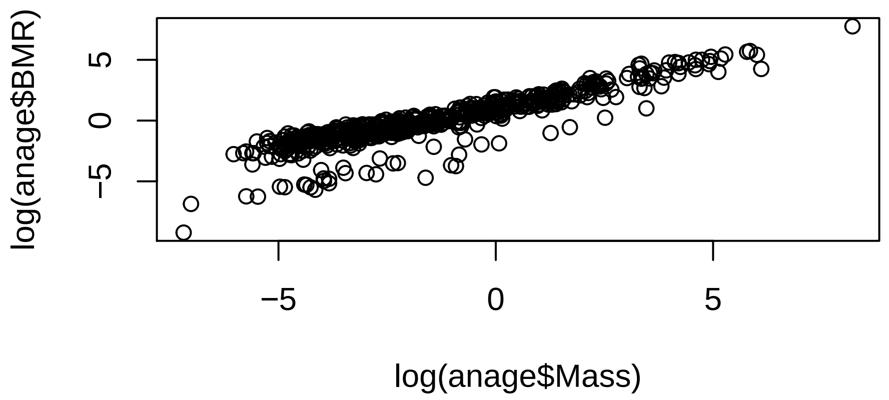
You can remind yourself what this data looks like using commands like `summary`, `str` and `names`.

I will process the data a bit to make it easier to work with. One of the commands might be new to you - `rename`. This is simply a way of renaming columns, in this case to make them more "user friendly" (e.g. I want to rename the column "Metabolic.rate..W." to "BMR" (for basal metabolic rate)).

I will also use `mutate` to (1) convert the Mass from grams to kilograms and (2) to make a new variable called "BMRperKg" which standardises metabolic rate by expressing it as rate per kilogram.

```
anage <- x %>%
  mutate(Species = paste(Genus,Species)) %>%
  rename(Longevity = "Maximum.longevity..yrs.",
         Mass = "Body.mass..g." ,
         BMR = "Metabolic.rate..W.") %>%
  select(Class, Order, Species,Mass,Longevity,BMR) %>%
  filter(Class %in% c("Aves","Amphibia","Mammalia","Reptilia")) %>%
  droplevels() %>% #this removes unused "factor levels" e.g. "Insecta"
  mutate(Mass = Mass/1000,
         BMRperKg = BMR/Mass)

summary(anage)
```

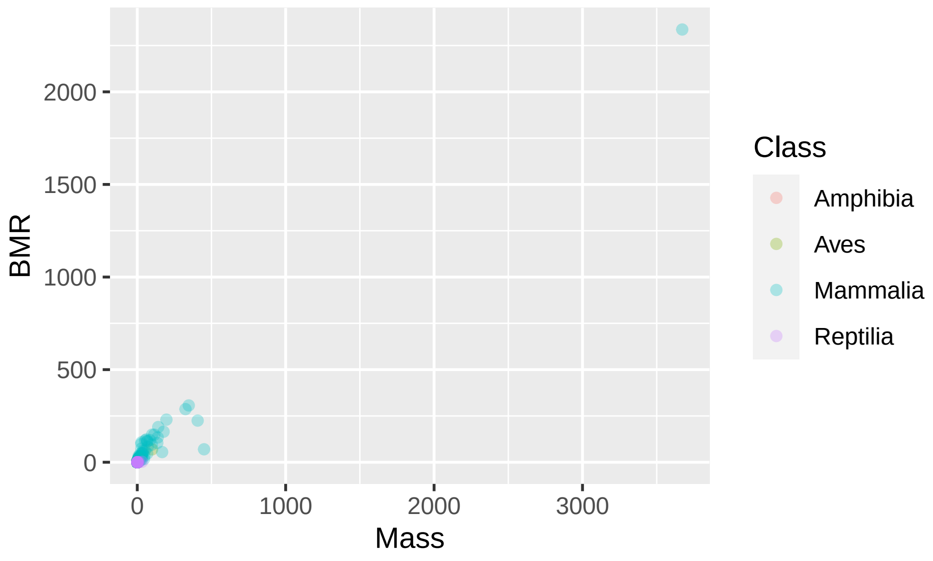
```
##     Class               Order              Species               Mass
##  Length:3231        Length:3231        Length:3231        Min.   :   0.001
##  Class :character   Class :character   Class :character   1st Qu.:   0.026
##  Mode  :character   Mode  :character   Mode  :character   Median :   0.131
##                                                           Mean   :  13.188
##                                                           3rd Qu.:   1.111
##                                                           Max.   :3672.000
##                                                           NA's   :2604
##    Longevity            BMR               BMRperKg
##  Min.   :  0.40   Min.   :   0.0001   Min.   : 0.0454
##  1st Qu.: 10.20   1st Qu.:   0.2655   1st Qu.: 2.2191
##  Median : 16.20   Median :   0.7050   Median : 4.5745
##  Mean   : 19.37   Mean   :  11.8309   Mean   : 7.0439
##  3rd Qu.: 24.45   3rd Qu.:   3.1370   3rd Qu.: 9.8686
##  Max.   :211.00   Max.   :2336.5000   Max.   :45.7692
##  NA's   :432      NA's   :2604        NA's   :2604
```



10.1 A basic plot

Now lets start with a basic plot. You will see a warning about removing rows with missing values. This is just a warning to let you know that there are missing (`NA`) values in the data you are plotting.

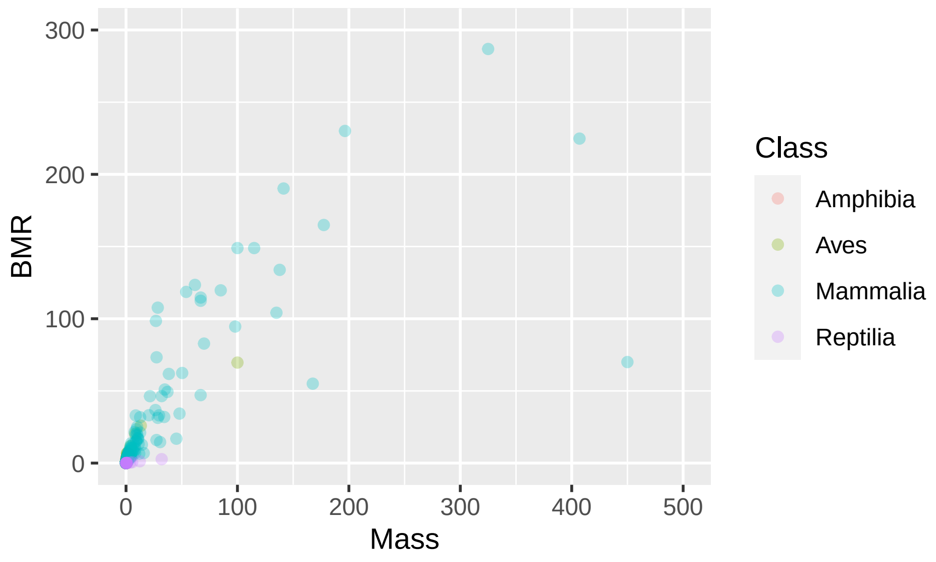
```
(p1 <- ggplot(anage,aes(x = Mass, y = BMR, colour = Class)) +
  geom_point(alpha=0.3)) #use alpha argument to make points transparent
```



10.2 Axis limits

These points are really spread out. One option to deal with this might be to set the range over which the axes are allowed to go using `xlim` and `ylim`.

```
p1 +
  xlim(0,500) +
  ylim(0,300)
```

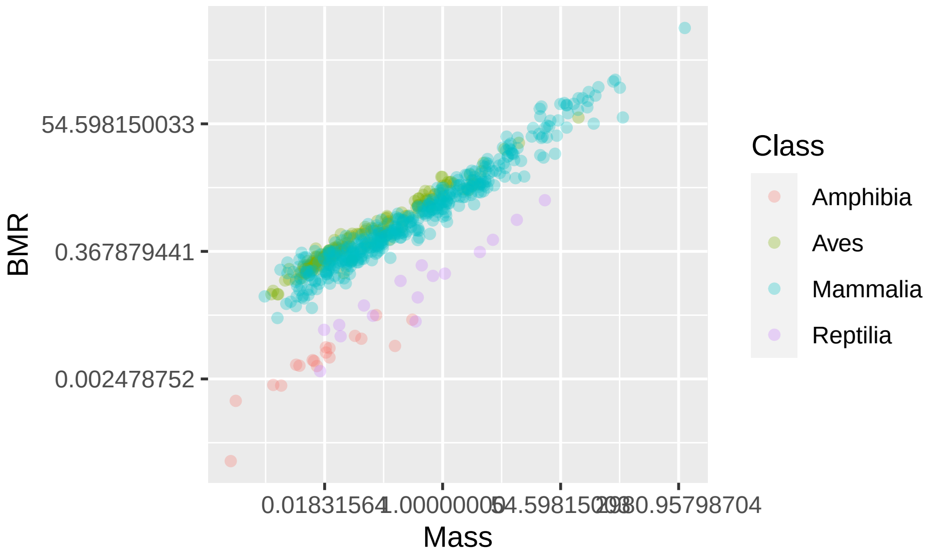


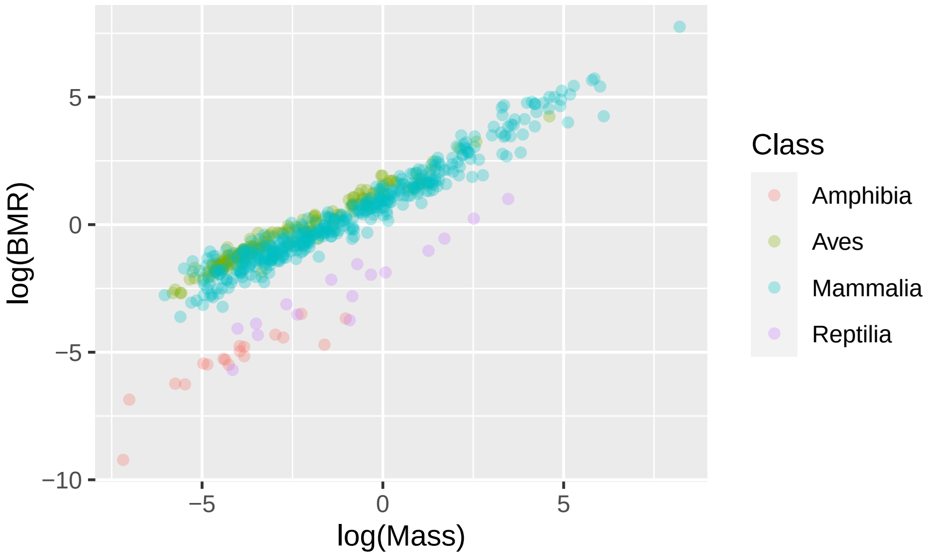
10.3 Transforming the axis (log scale)

In this particular case though use of a log scale would be best because even after focusing on a smaller part of the range of values you can see that the points are still concentrated at smaller values. In a moment, you will also see that log-transforming the data makes the cloud of points pleasingly linear.

You can set a log scale by using the commands `scale_x_continuous(trans = "log")` and `scale_y_continuous(trans = "log")`.

```
(p2 <- p1 +
  scale_x_continuous(trans = "log") +
  scale_y_continuous(trans = "log"))
```

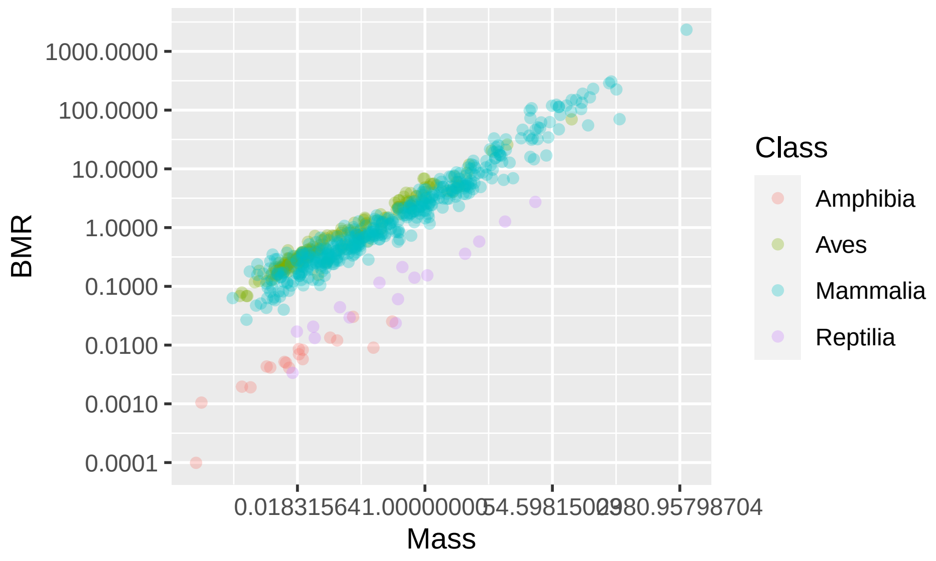




10.4 Changing the axis tick marks

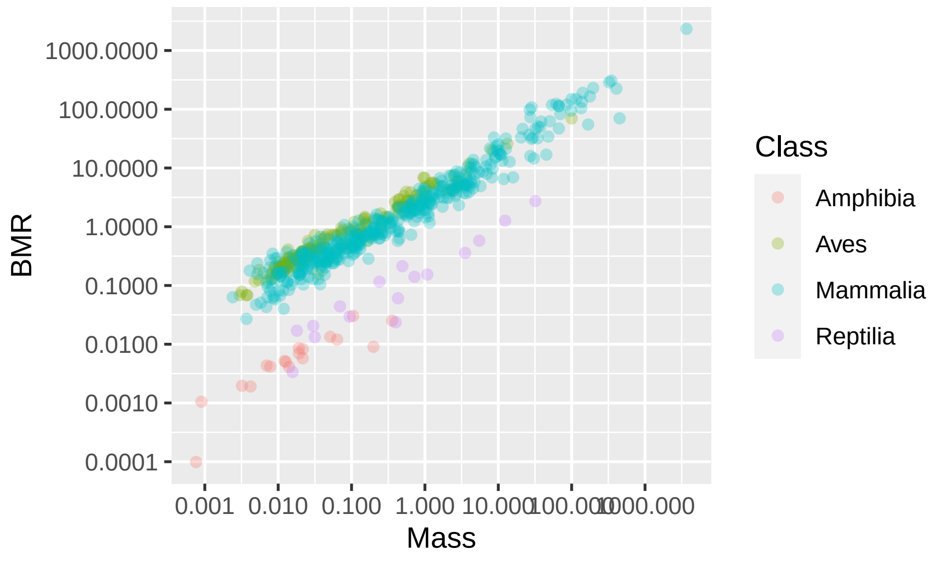
This looks nice. But the numbers on the axis are not very nice. Using `summary(anage$BMR)` tells us that the range of data is from 0.0001 to 2336.5. We could place tick marks anywhere on this axis, but let's try 0.0001, 0.001,0.1, 1,10, 100, 1000.

```
(p2 <- p1 +
  scale_x_continuous(trans = "log") +
  scale_y_continuous(trans = "log", breaks = c(0.0001,0.001,0.01,0.1,1,10,100,1000)))
```



Using `summary(anage$Mass)` tells us that the range of data is from 0.001 to 3672. We could place tick marks anywhere on this axis, but let's try 0.001,0.1, 1,10, 100, 1000.

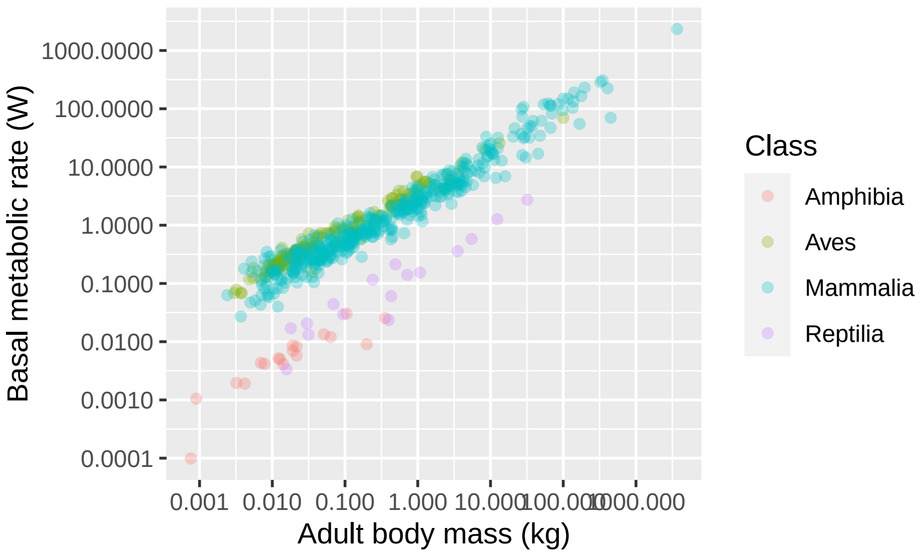
```
(p2 <- p1 +
  scale_x_continuous(trans = "log",
                     breaks = c(0.001,0.01,0.1,1,10,100,1000)) +
  scale_y_continuous(trans = "log",
                     breaks = c(0.0001,0.001,0.01,0.1,1,10,100,1000))
 )
```



10.5 Axis labels

Now, let's think about the axis labels. The labels in the plots so far have no units indicated, and might not be easy to interpret for the reader. Let's add units, and also spell out more fully what "BMR" and "Mass" means (the axes is basal metabolic rate in Watts and adult body mass in kg).

```
(p3 <- p2 +
  xlab("Adult body mass (kg)") +
  ylab("Basal metabolic rate (W)")
)
```



10.6 Colours

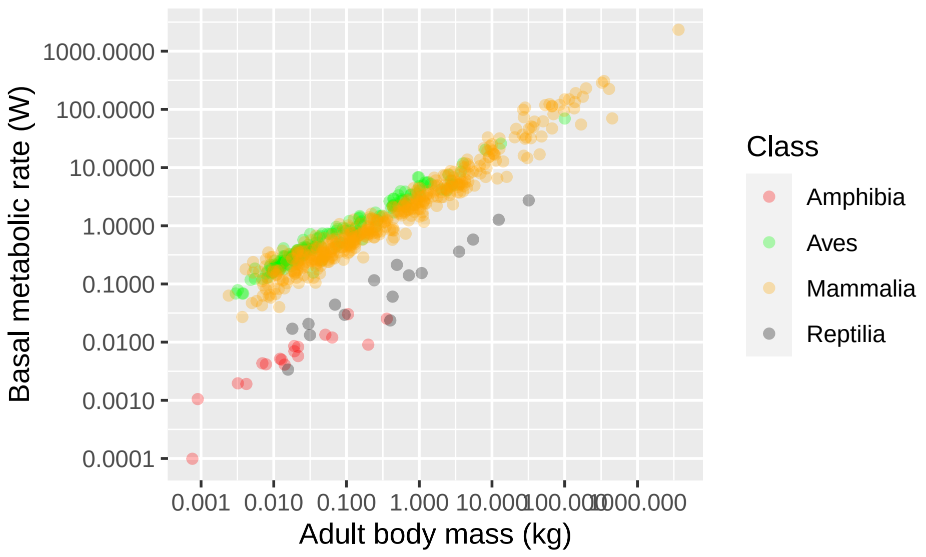
What about those colours? The `ggplot` package uses some default colours that are OK, but sometimes you will want to make a change.

You can "manually" adjust colours using the `scale_colour_manual` function. You can either name individual colours (e.g. "red","green","orange","black")[1], or you can find their so-called "hex-codes" from a site like http://colorbrewer2.org/ or https://htmlcolorcodes.com/color-picker/. You can add a two digit number after the hex code to set the "opaqueness" of the colour. For example "#FF000075" is red, with 75% opacity.

With colour names...

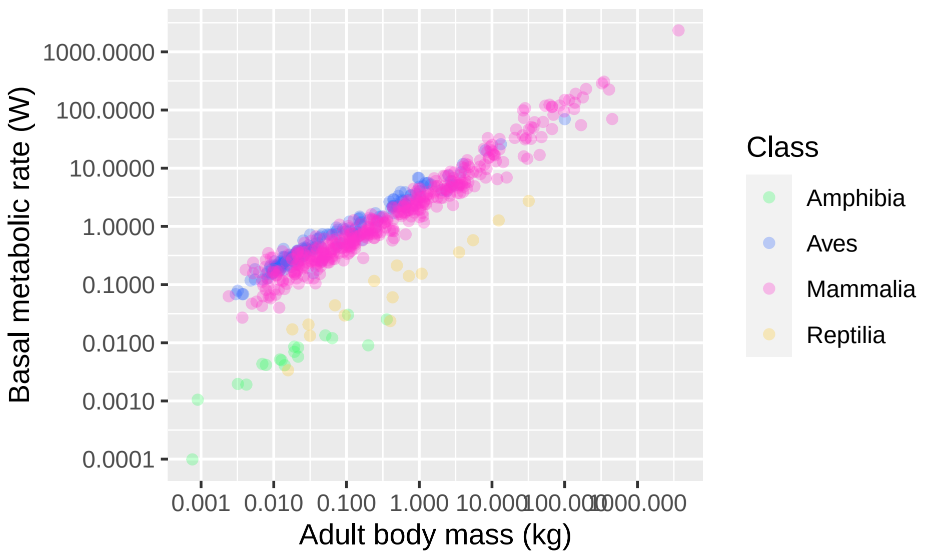
[1]See https://www.r-graph-gallery.com/42-colors-names.html

```
p3 +
  scale_colour_manual(values = c("red","green","orange","black"))
```



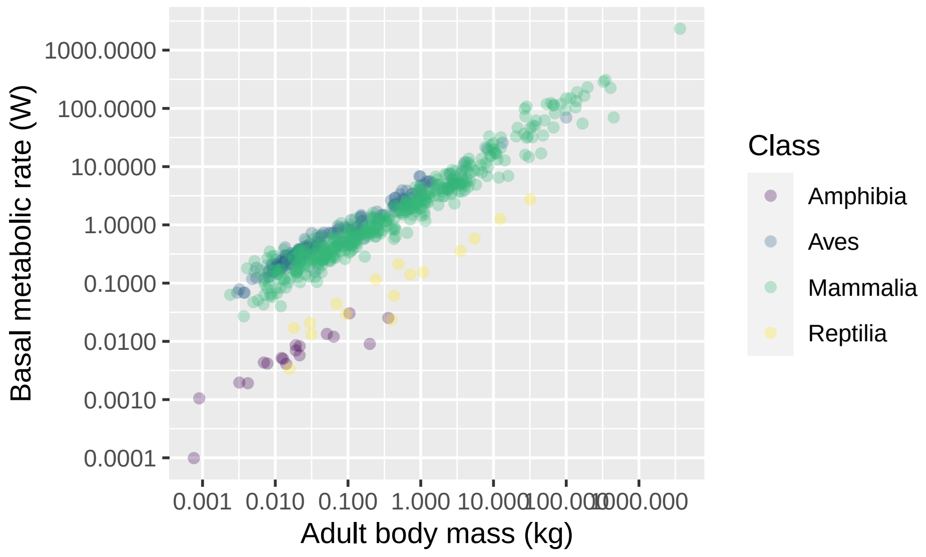
And with some hex codes…

```
p3 +
  scale_colour_manual(values = c("#33FF6475","#3368FF75","#FF33CE75","#FFCA3375"))
```



Another alternative is to use some of `ggplot`'s built in "palettes" of colour combinations. For example, there are several palettes called "`viridis`".

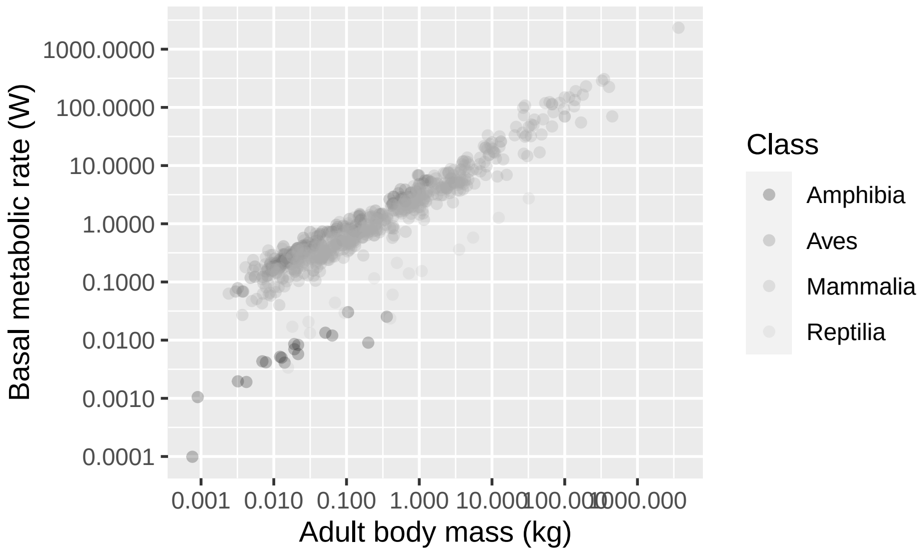
```
p3 +
  scale_colour_viridis_d(option = "D")
```



Try using other `option` arguments `A`, `B`, `C` and `E`. Try also adding an argument for transparency `alpha = 0.5`.

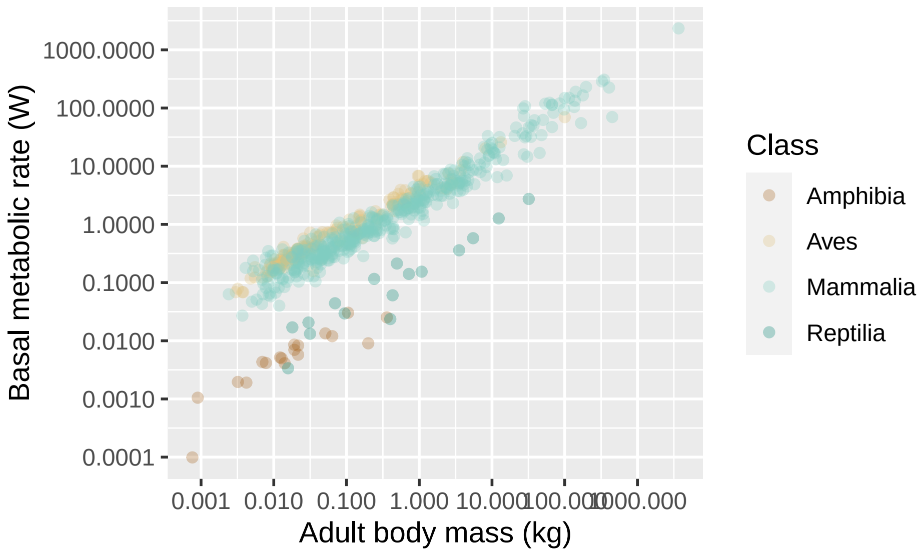
Here's a couple more palettes. There's one for shades of grey...

```
p3 +
  scale_colour_grey()
```



There's another one for various colour schemes, called "colour brewer". Try using "RdGy", "RdYlBu" and "Spectral" see `?scale_colour_brewer` for more options.

```
p3 +
  scale_colour_brewer(palette = "BrBG")
```

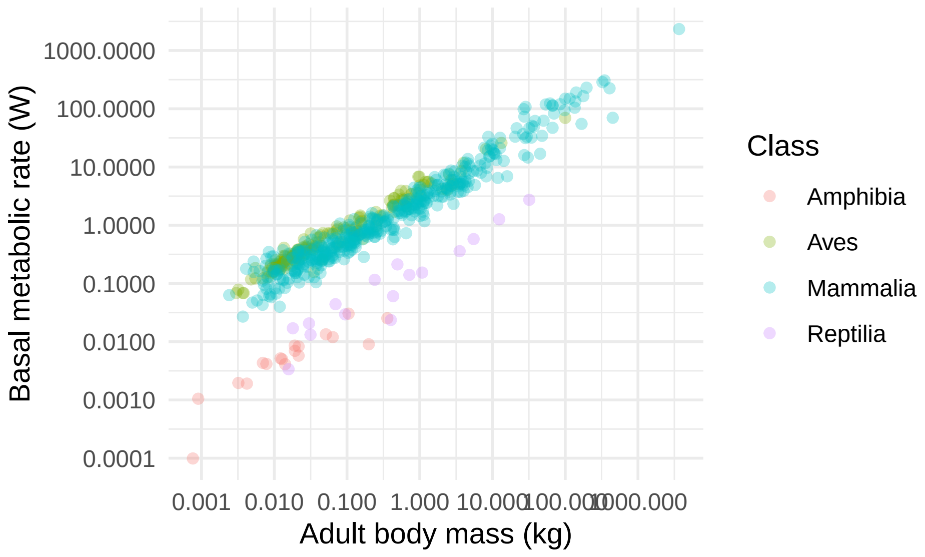


10.7 Themes

Finally, `ggplot` includes the option to set a **theme** for the plots. "Themes"" make adjustments to the "look" of the plot. It is possible to write your own themes, but I recommend to use some ready-made ones. You can implement them by adding them as you would any other addition to the `ggplot` command (e.g. `+ theme_light()`.

There are several themes included with `ggplot`. Try my favourite, `theme_minimal()`. Then try `theme_classic()` and `theme_dark()`.

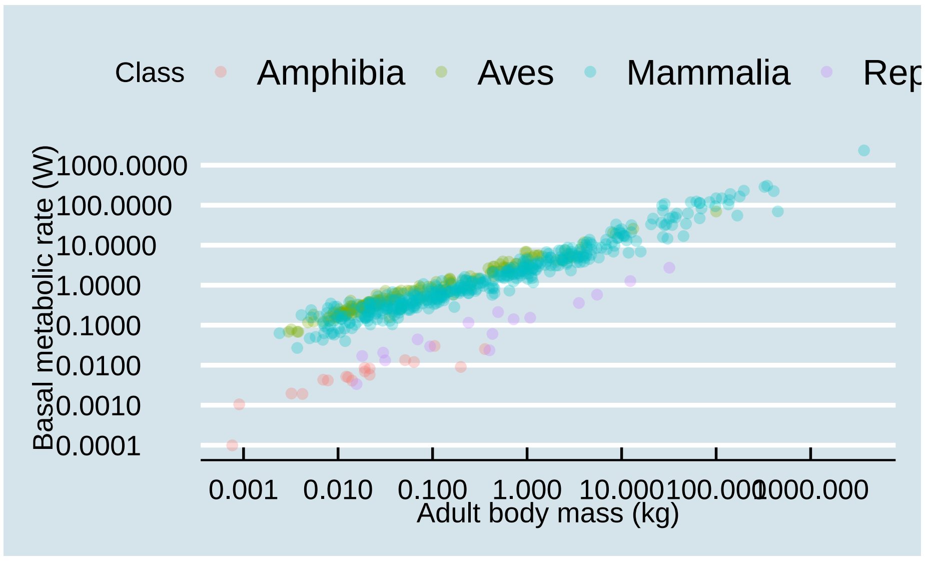
```
(p4 <- p3 +
  theme_minimal()
)
```



For more theme fun, you can install packages that include more themes. The best one is called `ggthemes` (remember that you only need to install the package once). Try `theme_economist()`, `theme_tufte()` and (ugh!) `theme_excel()`. You can see what other themes there in this package at https://jrnold.github.io/ggthemes/reference/index.html (some of them are really ugly in my opinion!).

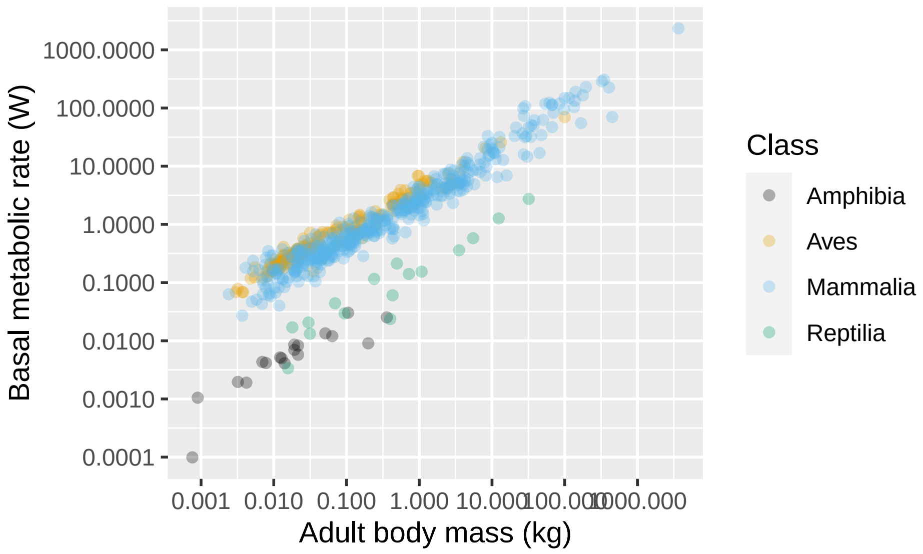
```
install.packages("ggthemes")
```

```
library(ggthemes)
p3 +
  theme_economist()
```



This package also includes some useful colour scales, including some for colour blind people.

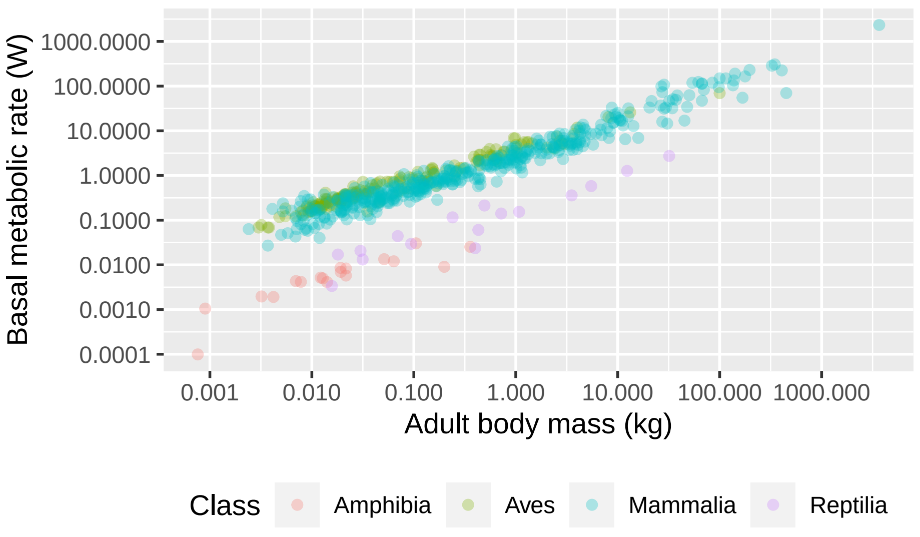
```
p3 +
  scale_color_colorblind()
```



10.8 Moving the legend

By default, the legend is placed on the right. You can move it around by adding a `theme` argument to your plot commands. It can also be placed on the "`top`", "`bottom`", or "`left`". You can also remove the legend altogether by using `legend.position = "none"`. You might also want to remove the legend title using the theme argument `legend.title = element_blank()`.

```
p3 +
  theme(legend.position  = "bottom")
```



10.9 Combining multiple plots

It is often useful to combine two or more plots into a single figure. For example, many journals have strict limits on the number of plots so it is useful to combine plots into "Figure 1A and B" etc.

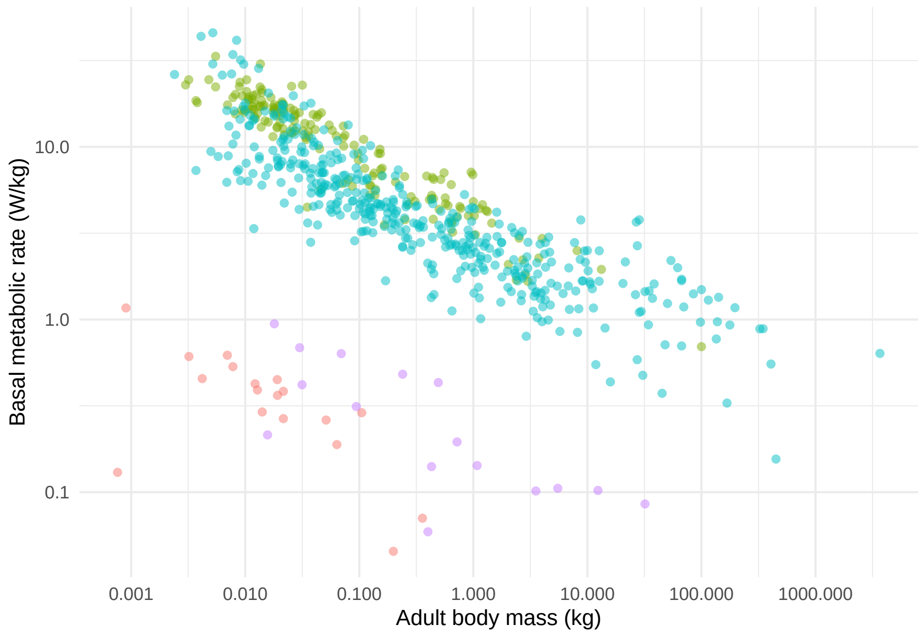
There are several R packages that can do this and my favourite is called `patchwork`.

```
install.packages("patchwork") #only need to do this once
```

```
library(patchwork)
```

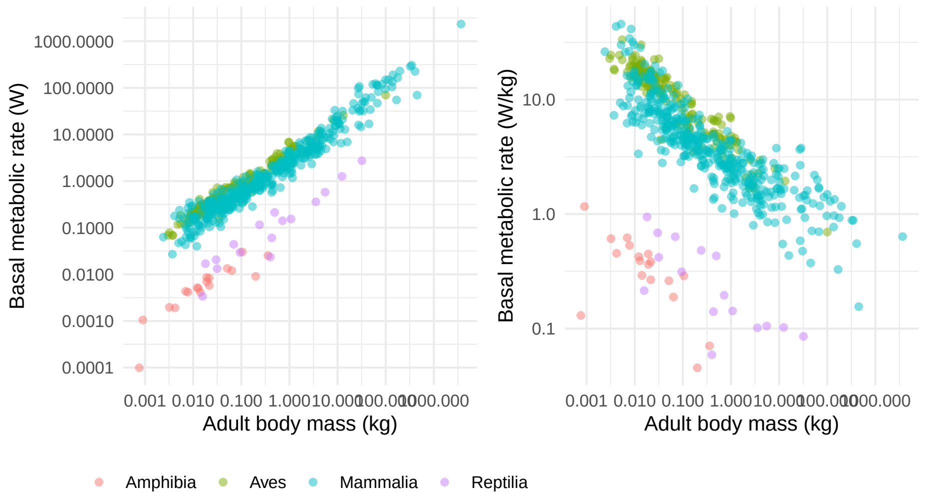
I will illustrate it by first making another plot, this time showing the relationship between body mass and *standardised* BMR (BMR per kg). Because I am combining the plots into a smaller space I have decided to remove the figure legend (I could put it in the figure caption instead).

```
#PlotA (this is what you have already created above)
plotA <- ggplot(anage,aes(x = Mass, y = BMR, colour = Class)) +
  geom_point(alpha=0.5) +
  scale_x_continuous(trans = "log",breaks =c(0.001,0.01,0.1,1,10,100,1000)) +
  scale_y_continuous(trans = "log",breaks =c(0.0001,0.001,0.01,0.1,1,10,100,1000)) +
  xlab("Adult body mass (kg)") +
  ylab("Basal metabolic rate (W)") +
  theme_minimal() +
  theme(legend.position  = "bottom",
        legend.title = element_blank())

(plotB <- ggplot(anage,aes(x = Mass, y = BMRperKg, colour = Class)) +
    geom_point(alpha=0.5) +
    scale_x_continuous(trans = "log",breaks =c(0.001,0.01,0.1,1,10,100,1000)) +
    scale_y_continuous(trans = "log",breaks =c(0.0001,0.001,0.01,0.1,1,10)) +
    xlab("Adult body mass (kg)") +
    ylab("Basal metabolic rate (W/kg)") +
    theme_minimal() +
    theme(legend.position  = "none")
) #This one is wrapped in brackets so that R shows it
```



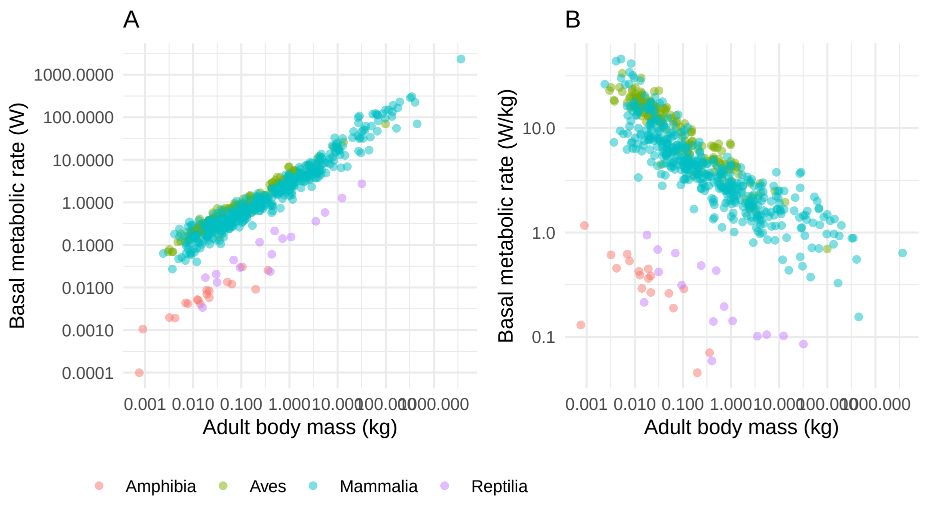
Now I can combine these using the very simple syntax like this:

```
plotA + plotB
```



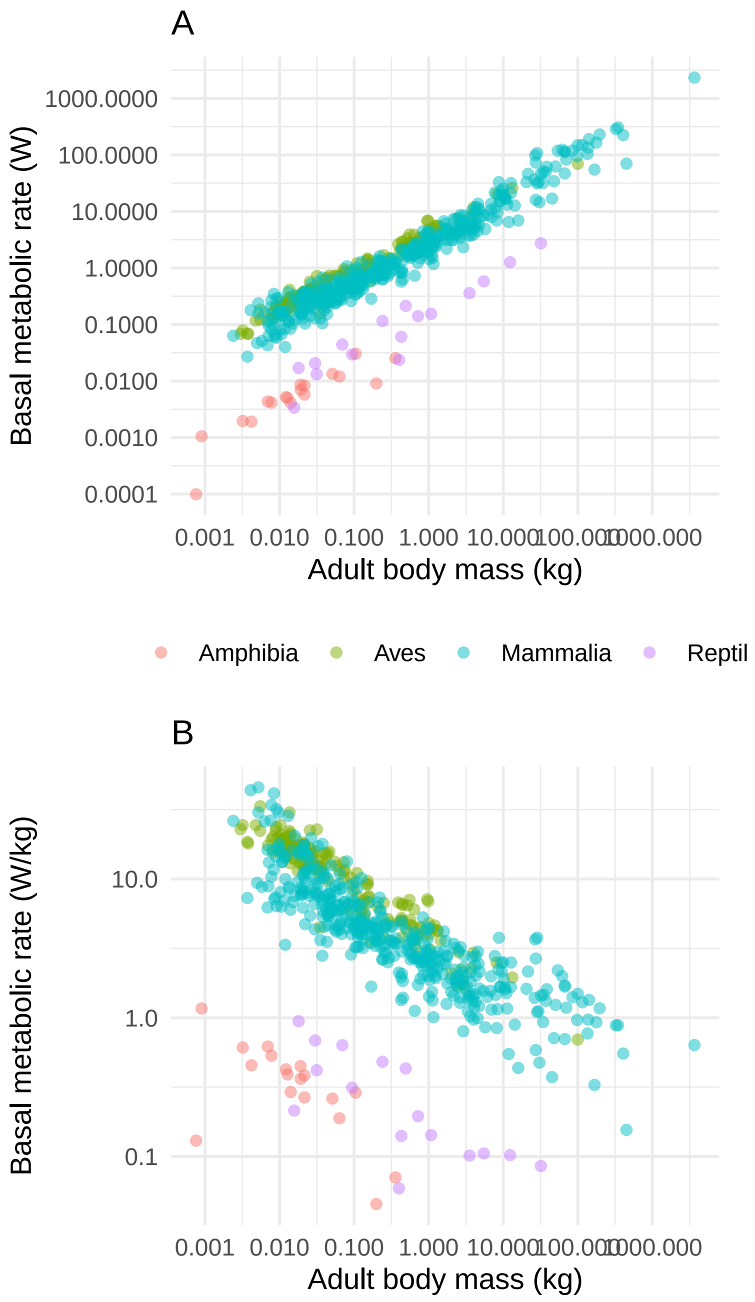
I can add titles using the `ggtitle` command like this.

```
plotA + ggtitle("A") +  plotB + ggtitle("B")
```



You could place the sub-plots on top of each other like this.

```
(plotA + ggtitle("A")) / (plotB + ggtitle("B"))
```



10.10 Saving your plot

You should, I think, already know about using the “Export” button in RStudio to save out your plot. This is useful and easy, but you should know that you can also save the plots using a typed command (`ggsave`) in your script. This command is handy because it allows you to automatically set the size, and file name of your plot.

The default setting for `ggsave` is that it will save the last plot that was printed to your computer screen to a file name that you specify. Therefore easiest way to use the command is to simply place the `ggsave` command immediately after your `ggplot` command. You should set the width and height of the plot and the units (the default is inches). It usually takes a few attempts and a bit of trial-and-error to choose the dimensions so that the plot looks nice.

```
ggsave("MySavedPlot1.png", width = 18, height=10, units = "cm")
```

The command can save to various file types including `png`, `jpeg`, `pdf` (see the `ggplot` help file for more). R knows what file file type is chosen by checking the file extension in the file name (e.g. `.png`). I advise to use `png`.

10.11 Final word on plots

We have covered a lot of ground here. There is a lot to learn, but don't feel like you have to remember all of these commands (I don't). Mostly it is simply a case of remembering that it is *possible* to do these things, and knowing where to look up the commands. Obvious starting points are this course book, and the text book (including the online version!). You can also usually find help by Googling "ggplot" followed by what you are trying to do (e.g. "ggplot change axis ticks"). One of my frequently used web sites is this one http://www.sthda.com/english/ which has an extensive section on `ggplot` (http://www.sthda.com/english/wiki/ggplot2-essentials).

Even though we have covered a lot of ground we have still only gotten a taster of what `ggplot` is capable of. I encourage you to learn more. A useful resource for learning is the online R graph gallery" at https://www.r-graph-gallery.com/, which shows you how to make and modify many types of plot.

Part III

Statistics

Chapter 11

Randomisation Tests

Simple experiments testing for a difference in mean values between two groups usually have the null hypothesis that there is no difference. The alternative hypothesis varies. Sometimes it is simply that the two groups are different (and that the difference could be wither positive or negative). In other cases the alternative hypothesis is that the mean of Group A is less then the mean of Group B (or that it is greater).

Randomisation tests are an intuitive, but computationally intensive way of testing these hypotheses. They have a long history and were first proposed by R.A. Fisher in the 1930s. However they only became convenient when computers became sufficiently fast to do the calculations.

Carrying out a test in R requires that you put your `dplyr` skills to the test. Here you will be guided through an example.

11.1 Randomisation test in R

A new drug has been developed that is supposed to reduce cholesterol levels in men. An experiment has been carried out where 12 human test subjects have been assigned randomly to two groups: "Control" and "Drug". The pharmaceutical company is hoping that the "Drug" group will have lower cholesterol than the "Control" group. The aim here is to do a randomisation test to check that.

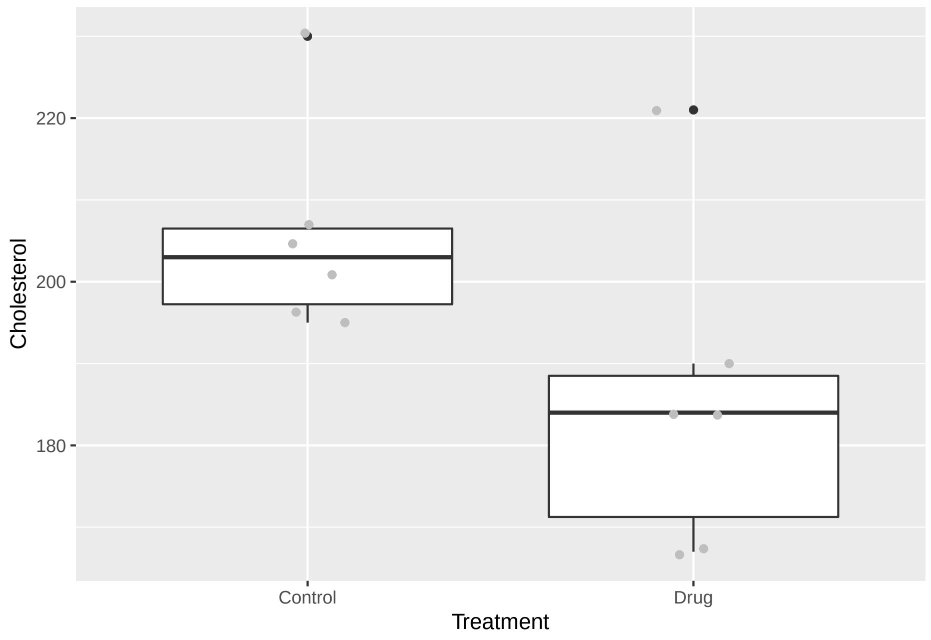
Remember to load the `dplyr`, `magrittr` and `ggplot` packages, and to set your working directory correctly.

Import the data, called `cholesterol.csv`.

```
ch <-read.csv("CourseData/cholesterol.csv")
```

Let's first take a look at the data by plotting it. I will first plot a boxplot first, and add the jittered points for clarity.

```
ggplot(ch,aes(x=Treatment,y=Cholesterol)) +
      geom_boxplot()+
    geom_jitter(colour="grey",width=.1)
```



It looks like there might be a difference between the groups. Now let's consider our test statistic and our hypotheses. Our test statistic is the difference in mean cholesterol levels between the two groups: mean of control group minus the mean of the drug group. The *null hypothesis* is that there is no difference between these two groups (i.e. the difference should be close to 0) The *alternative hypothesis* is that the mean of the drug group should be less than the mean of the drug group. (i.e. mean of control group minus the mean of the drug group should be negative).

11.1.1 Calculate the observed difference

There are a few ways of doing this. In base-R you can use the function `tapply` ("table apply"), followed by `diff` ("difference").

```
tapply(ch$Cholesterol,ch$Treatment,mean)
```

```
##  Control     Drug
## 205.6667 185.5000
```

```
diff(tapply(ch$Cholesterol,ch$Treatment,mean))
```

```
##      Drug
## -20.16667
```

Because we are focusing on learning `dplyr`, you can also calculate the means like like this:

```
ch %>% # ch is the cholesterol data
  group_by(Treatment) %>% # group the data by treatment
  summarise(mean = mean(Cholesterol)) # calculate means
```

```
## # A tibble: 2 x 2
##   Treatment  mean
##   <chr>     <dbl>
## 1 Control    206.
## 2 Drug       186.
```

Here the pipes (%>%) are passing the result of each function on as input to the next. You can use further commands, `pull` to get the `mean` vector from the summary table, and then use `diff` to calculate the difference between the groups, before passing that to a value called "`observedDiff`".

```
observedDiff <- ch %>%
  group_by(Treatment) %>% # group the data by treatment
  summarise(mean = mean(Cholesterol)) %>% # calculate means
  pull(mean) %>% # extract the mean vector
  diff()
```

This is a complicated set of commands. To make sure that you understand it, try running it bit-by-bit to see what is going on.

11.1.2 Null distribution

Now we ask, what would the world look like if our null hypothesis was *true*. To do this we can disassociate the treatment group variable from the measured cholesterol values. We do this using by using the `mutate` function to replace the `Treatment` variable with a shuffled version of itself with the `sample` function.

Let's try that one time:

```
ch %>%
  mutate(Treatment = sample(Treatment)) %>% #shuffle the Treatment data
  group_by(Treatment) %>%
  summarise(mean = mean(Cholesterol)) %>%
  pull(mean) %>%
  diff()
```

```
## [1] -10.16667
```

In this instance, the difference with the shuffled `Treatment` values is 0.833, which is rather different from our observed difference of -20.1666667.

Doing this one time is not much help though - we need to repeat this many times. I suggest that you do it 1000 times here, but some statisticians would suggest 5000 or even 10000 replicates.

We can do this easily in R using the function `replicate` which simply a kind of wrapper that tells R to repeat a command `n` times and then pass the result to a vector. Let's try it first 10 times to see how it works:

```
replicate(10,
            ch %>%
            mutate(Treatment = sample(Treatment)) %>%
            group_by(Treatment) %>%
            summarise(mean = mean(Cholesterol)) %>%
            pull(mean) %>%
            diff()
)
```

```
##  [1]  11.500000  -6.500000 -19.166667  -1.833333  -7.833333  21.166667
##  [7] -11.500000 -21.833333  28.833333  -5.833333
```

You can see that the `replicate` command simply does the sampling-recalculation of the mean 10 times.

In the commands below I create 1000 replicates of the shuffled differences. I want to put them in a dataframe to make it easy to plot. Therefore, I first create a `data.frame` called `shuffledData`. This data frame initially has a variable called `rep` which consists of the numbers 1-1000. I then use `mutate` to add the 1000 shuffled differences.

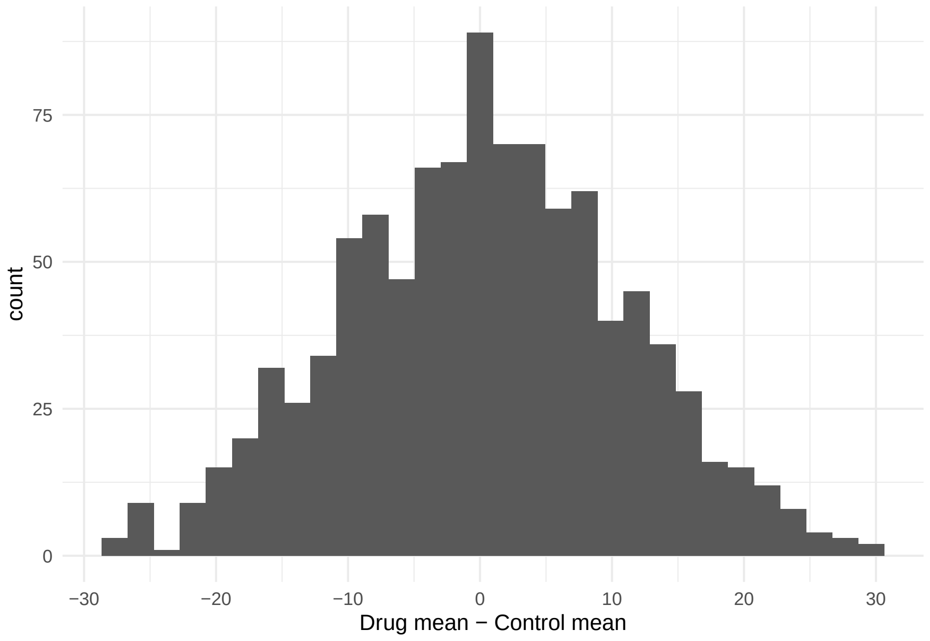
```
shuffledData <- data.frame(rep = 1:1000) %>%
  mutate(shuffledDiffs = replicate(1000,
            ch %>%
            mutate(Treatment = sample(Treatment)) %>%
            group_by(Treatment) %>%
            summarise(mean = mean(Cholesterol)) %>%
            pull(mean) %>%
            diff()
            ))
```

When you use `summarise`, R will give you a message like this:
`summarise() ungrouping output (override with .groups argument)`
This can be annoying, particularly if you are using randomisation tests and summarising hundreds of times. Thankfully, you can turn off this behaviour by setting one of the `dplyr` options like this:
`options(dplyr.summarise.inform = FALSE)`
I suggest to put this code at the beginning of your script if the messages annoy you!

11.1.3 Testing significance

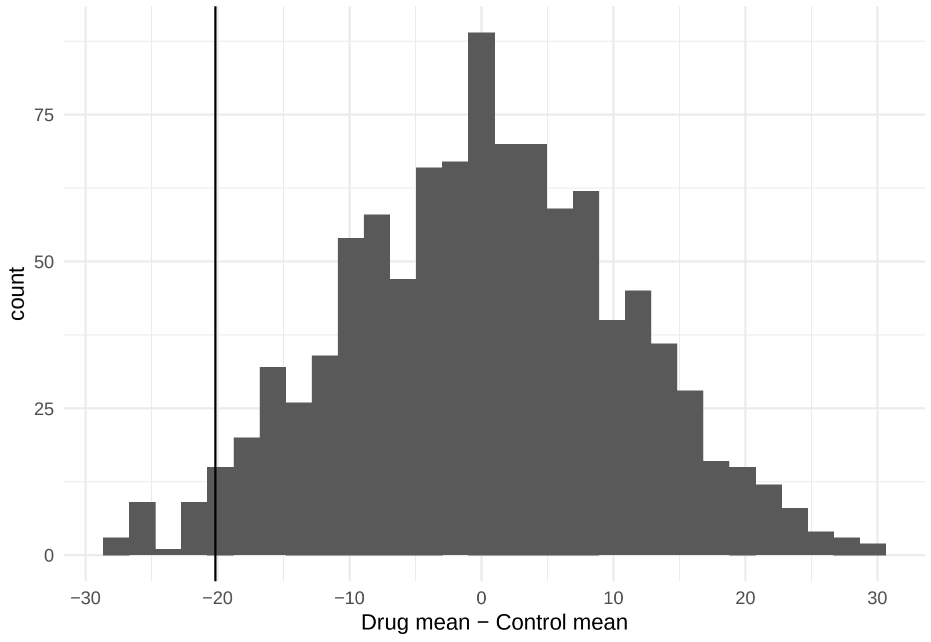
Before formally testing the hypothesis it is useful to visualise what we have created in a histogram. I can use `ggplot` to do this, to create a plot called `p1`. Note that by putting the command in brackets R will both create the plot object, and print it to the screen. Note that because the shuffling of the data is random process your graph will look slightly different to mine.

```
(p1<-ggplot(shuffledData,aes(x=shuffledDiffs)) +
  geom_histogram() +
  theme_minimal()+
  xlab("Drug mean - Control mean"))
```



You can now add your observed difference (calculated above) to this plot like this:

```r
p1 + geom_vline(xintercept = observedDiff)
```



11.1.4 Testing the hypothesis

Recall that the alternative hypothesis is that the observed difference (control mean-drug mean) will be less than 0. You can see that there are few of the null distribution sample that are as extreme as the observed difference. To calculate a p-value we can simply count these values and express them as a proportion. Note that because the shuffling of the data is random process your result will probably be slightly different to mine.

```r
table(shuffledData$shuffledDiffs<=observedDiff)
```

```
##
## FALSE  TRUE
##   977    23
```

So that is 23 of the shuffled values that are equal to or less than the observed difference. The p-value is then simply $23/1000 = 0.023$.

Therefore we can say that the drug appears to be effective at reducing cholesterol.

11.1.5 Writing it up

We can report our findings something like this:

"To test whether effect of the drug at reducing cholesterol level is statistically significant I did a 1000 replicate randomisation test with the null hypothesis being that there is no difference between the group means and the alternative hypothesis that the mean for the drug treatment is lower than the control treatment. I compared the observed difference to this null distribution to calculate a p-value in a one-sided test.

The observed mean values of the control and treatment groups 205.667 and 185.500 respectively and the difference between them is therefore -20.167 (drug mean - control mean). Only 25 of the 1000 null distribution replicates were as low or lower than my observed difference value. I conclude that the observed difference between the means of the two treatment groups is statistically significant (p = 0.025)"

11.2 Paired Randomisation Tests

The paired randomisation test is a one-sample randomisation test where the distribution is tested against a value of 0 (i.e. where there is no difference between the two groups). Often, this distribution is the **difference** in measurements between two sets of measurements taken from the same individuals (or study sites) before and after some treatment has been applied.

I will illustrate this with an example from Everitt (1994) who looked at using cognitive behaviour therapy as a treatment for anorexia. Everitt collected data on weights of people before and after therapy. These data are in the file `anorexiaCBT.csv`

```
#Remember to set your working directory first
an <- read.csv("CourseData/anorexiaCBT.csv")
head(an)
```

```
##   Subject Week01 Week08
## 1       1   80.5   82.2
## 2       2   84.9   85.6
## 3       3   81.5   81.4
## 4       4   82.6   81.9
## 5       5   79.9   76.4
## 6       6   88.7  103.6
```

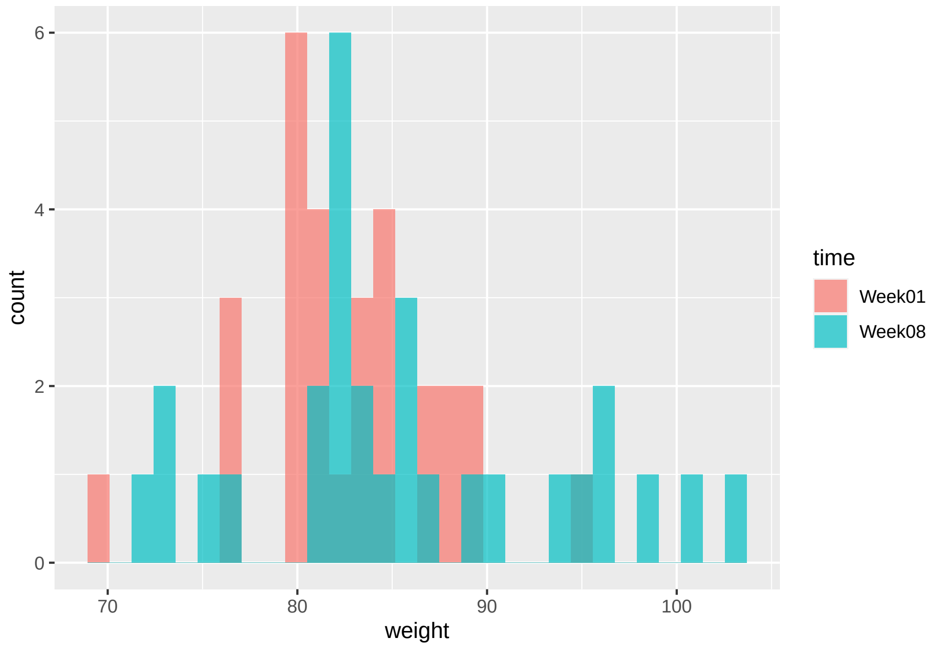
These data are arranged in a so-called "wide" format. To make plotting and analysis data need to be rearranged into a tidy "long" format so that each observation is on a row. We can do this using the `pivot_longer` function:

```
an <- an %>%
  pivot_longer(cols = starts_with("Week"),names_to = "time",values_to = "weight")
head(an)
```

```
## # A tibble: 6 x 3
##   Subject time   weight
##     <int> <chr>   <dbl>
## 1       1 Week01   80.5
## 2       1 Week08   82.2
## 3       2 Week01   84.9
## 4       2 Week08   85.6
## 5       3 Week01   81.5
## 6       3 Week08   81.4
```

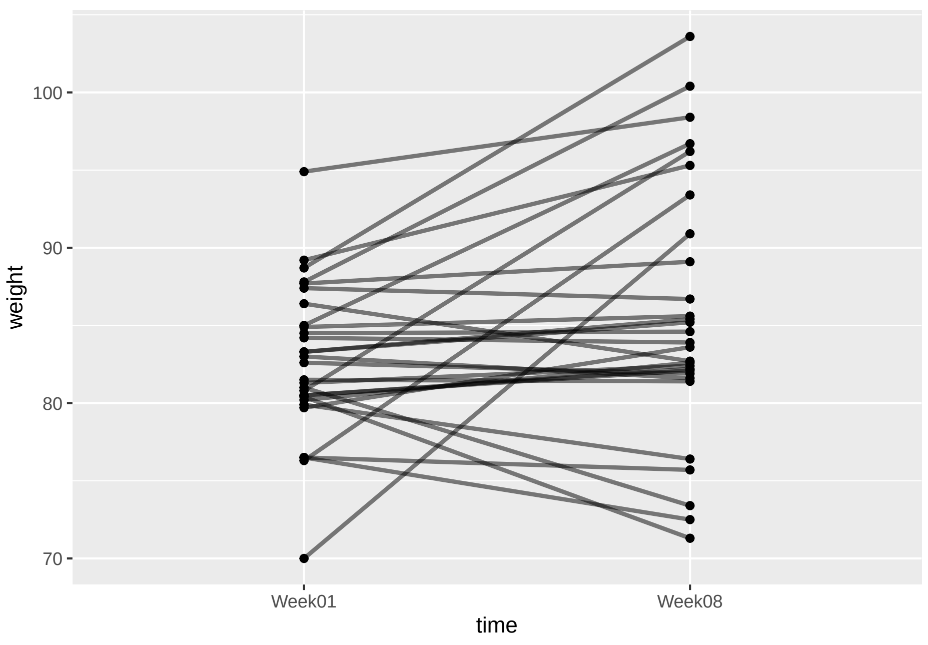
We should *always* plot the data. So here goes.

```
(p1<-ggplot(an,aes(x=weight,fill=time)) +
   geom_histogram(position = "identity", alpha=.7)
 )
```



Another useful way to plot this data is to use an **interaction plot**. In these plots the matched pairs (grouped by `Subject`) are joined together with lines. You can plot one like this:

```
(p2<-ggplot(an,aes(x=time,y=weight,group=Subject)) +
   geom_point() +
   geom_line(size=1, alpha=0.5)
 )
```



What we are interested in is whether there has been a change in weight of the subjects after CBT. The null hypothesis is that there is zero change in weight. The alternative hypothesis is that weight has increased.

The starting point for the analysis is to calculate the observed change in weight.

```
an <- an %>%
  group_by(Subject) %>%
  summarise(change = diff(weight))
```

You have created a dataset that looks like this:

```
head(an)
```

```
## # A tibble: 6 x 2
##   Subject change
##     <int>  <dbl>
## 1       1  1.70
## 2       2  0.700
## 3       3 -0.100
## 4       4 -0.700
## 5       5 -3.5
## 6       6 14.9
```

And you can calculate the observed change like this:

```
obsChange <- mean(an$change)
obsChange
```

```
## [1] 3.006897
```

11.2.1 The randomisation test

The logic of this test is that if the experimental treatment has no effect on weight, then the *Before* weight is just as likely to be larger than the *After* weight as it is to be smaller.

Therefore, to carry out this test, we can permute the SIGN of the change in weight (i.e. we randomly flip values from positive to negative and vice versa). We can do this by multiplying by 1 or -1, randomly.

```
head(an)
```

```
## # A tibble: 6 x 2
##   Subject change
##     <int>  <dbl>
## 1       1  1.70
## 2       2  0.700
## 3       3 -0.100
## 4       4 -0.700
## 5       5 -3.5
## 6       6 14.9
```

```
anShuffled <- an %>%
  mutate(sign = sample(c(1,-1),size = nrow(an),replace = TRUE)) %>%
  mutate(shuffledChange = change * sign)
```

Let's take a look at this new shuffled dataset:

```
head(anShuffled)
```

```
## # A tibble: 6 x 4
##   Subject change  sign shuffledChange
##     <int>  <dbl> <dbl>          <dbl>
## 1       1  1.70      1          1.70
## 2       2  0.700     1          0.700
## 3       3 -0.100     1         -0.100
## 4       4 -0.700    -1          0.700
## 5       5 -3.5       1         -3.5
## 6       6 14.9      -1        -14.9
```

We need to calculate the mean of this shuffled vector. We can do this by `pull` to get the vector, and then `mean`.

```
an %>%
  mutate(sign = sample(c(1,-1),size = nrow(an),replace = TRUE)) %>%
  mutate(shuffledChange = change * sign) %>%
  pull(shuffledChange) %>%
  mean()
```

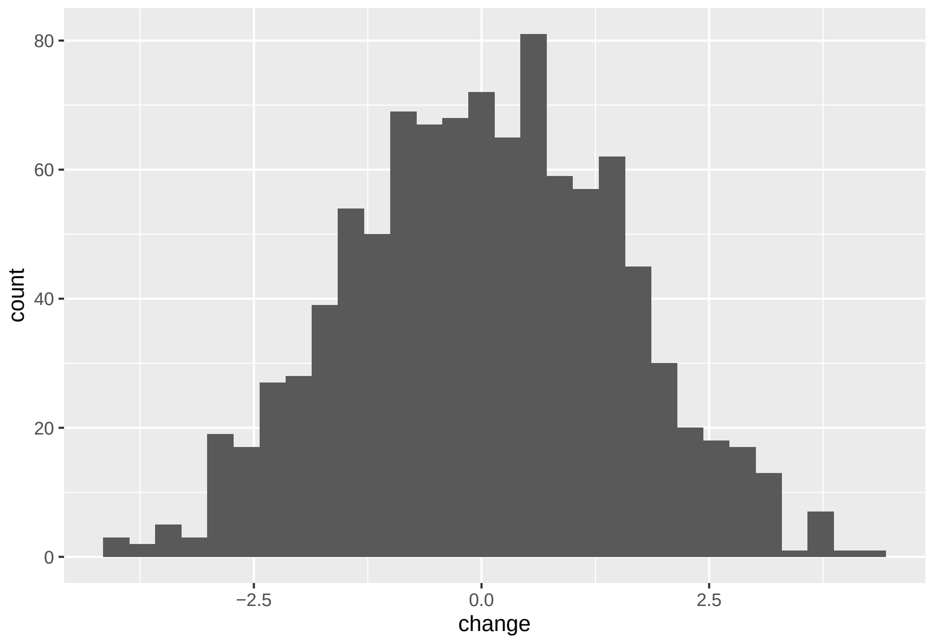
```
## [1] 0.1517241
```

Now we will build a null distribution of changes in weight by repeating this 1000 times. We can do this using the `replicate` function to "wrap" around the function, passing the result into a data frame. We can then compare this null distribution to the observed change.

```
nullDist = data.frame(change =
                        replicate(1000,an %>%
                                    mutate(sign = sample(c(1,-1),size = nrow(an),replace = TRUE
                                    mutate(shuffledChange = change * sign) %>%
                                    pull(shuffledChange) %>%
                                    mean()))
```

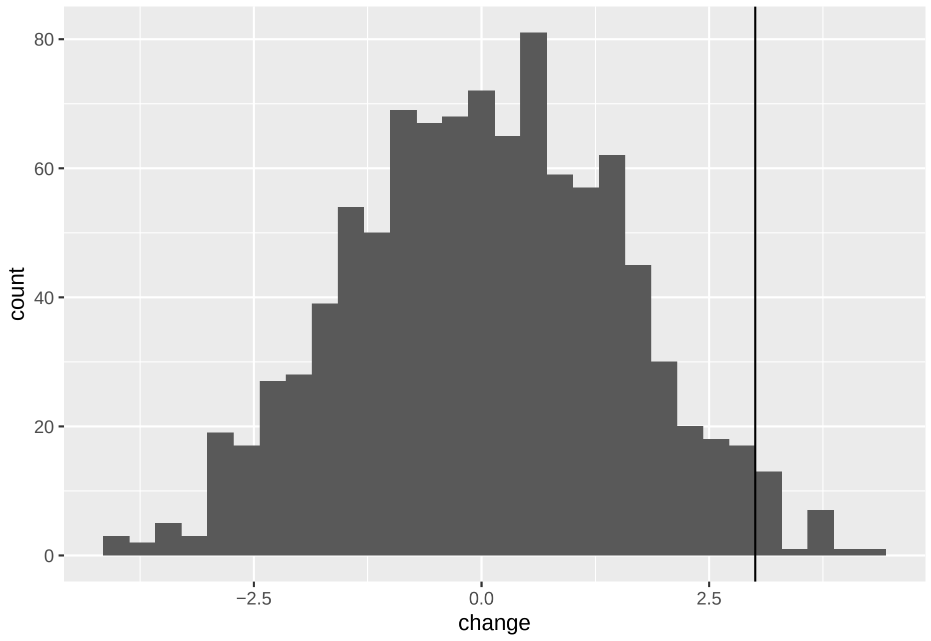
11.2.2 Null distribution

```
(nullDistPlot <- ggplot(nullDist,aes(x=change)) +
  geom_histogram())
```



We can add the observed change as a line to this:

```
nullDistPlot + geom_vline(xintercept = obsChange)
```



11.2.3 The formal hypothesis test

The formal test of significance then works by asking how many of the null distribution replicates are as extreme as the observed change.

```
table(nullDist$change>=obsChange)
```

```
##
## FALSE  TRUE
##   977    23
```

So we can see that 23 of 1000 replicates were greater than or equal to the observed change. This translates to a p-value of 0.023. We can therefore say that the observed change in weight after CBT was significantly greater than what we would expect from chance.

11.3 Exercise: Sexual selection in Hercules beetles

A Hercules beetle is a large rainforest species from South America. Researchers suspect that sexual selection has been operating on the species so that the males are significantly larger than the females. You are given data[1] on width measurements in cm of a small sample of 20 individuals of each sex. Can you use your skills to report whether males are significantly larger than females.

The data are called `herculesBeetle.csv` and can be found via the Dropbox link on Blackboard.

Follow the following prompts to get to your answer:

1. What is your null hypothesis?
2. What is your alternative hypothesis?
3. Import the data.
4. Calculate the mean for each sex (either using `tapply` or using `dplyr` tools)
5. Plot the data as a histogram.
6. Add vertical lines to the plot to indicate the mean values.
7. Now calculate the difference between the mean values using `dplyr` tools, or `tapply`.
8. Use `sample` to randomise the sex column of the data, and recalculate the difference between the mean.

[1]This example is from: https://uoftcoders.github.io/rcourse/lec09-Randomization-tests.html

9. Use `replicate` to repeat this 10 times (to ensure that you code works).
10. When your code is working, use `replicate` again, but this time with 1000 replicates and pass the results into a data frame.
11. Use `ggplot` to plot the null distribution you have just created, and add the observed difference.
12. Obtain the p-value for the hypothesis test described above. (1) how many of the observed differences are greater than or equal to the shuffled differences in the null distribution. (2) what is this expressed as a proportion of the number of replicates.
13. Summarise your result as in a report. Describe the method, followed by the result and conclusion.

Chapter 12

Comparing two means with a t-test

We will cover the following:

- One-sample t-test
- Paired t-test
- Two-sample t-test ("Welch t-test")

12.1 Some theory

In this theory section I focus on the one-sample t-test, but the concepts apply to the other types of t-test.

The one-sample t-test is used to compare the mean of a sample to some fixed value. For example, we might want to compare pollution levels (e.g. in mg/m^3) in a sample to some acceptable threshold value to help us decide whether we need to take action to prevent or clean up pollution.

One of the assumptions of t-tests (and many other tests/models) is that the distribution of values in the **sample** of data can be described by a normal distribution. If this assumption is true, you can use these data to estimate the parameters of this sample's normal distribution: the mean and standard error of the mean.

The mean gives an estimate of location, and the standard error of the mean (which is calculated as $s/\sqrt{n}$, where s = standard deviation and n = sample size) gives an estimate of precision of this estimate (i.e. how certain is it that the mean value is really where you think it is?)

The t-test then works by comparing your estimated distribution with some fixed value. Sometimes you are asking "is my mean *different* from the value?", other times you are asking "is my mean less than/greater than the value?". This depends on the hypothesis. The default that R-uses is that it tests whether the