Babraham Bioinformatics

Introduction to R

Version 2018-09

Licence

Table of Contents

Introduction

R is a popular language and environment that allows powerful and fast manipulation of data, offering many statistical and graphical options.

This course aims to introduce R as a tool for statistics and graphics, with the main aim being to become comfortable with the R environment. It will focus on entering and manipulating data in R and producing simple graphs. A few functions for basic statistics will be briefly introduced, but statistical functions will not be covered in detail.

Getting Started with R

What is R?

The official R project web page describes R as a ' language and environment for statistical computing and graphics'. It can be daunting if you haven't done any programming before but it is worth taking some time to familiarise yourself with the R environment as once you have grasped some of the basics it can be a very useful tool. A wide variety of statistical functions come with the default install and there are many other packages that can be installed if required.
It is very quick and easy to produce graphs with default parameters for a quick view of your data and there are all manner of parameters that can be specified to customise your graphs. R is often used to perform analysis and produce graphs for publication.

Good things about R

- It's free
- It works on all platforms
- It can deal with much larger datasets than Excel for example
- Graphs can be produced to your own specification
- It can be used to perform powerful statistical analysis
- It is well supported and documented

Bad things about R

- It can struggle to cope with extremely large datasets
- The environment can be daunting if you don't have any programming experience
- It has a rather unhelpful name when it comes to googling problems (though you can use http://www.rseek.org/ or google 'R help forum' and try that instead)

Installing R and RStudio

Instructions for downloading and installing R can be found on the R project website http://www.r-project.org/. Versions are available for Windows, Linux and Mac.

RStudio is an integrated development environment for R, available for Windows, Linux and Mac OS and like R, is free software. It offers a neat and tidy environment to work in and also provides some help with importing datasets and installing packages etc. You must have R installed in order to run RStudio. More information and instructions for download can be found at http://www.rstudio.org/.

R help

R has comprehensive help pages that are very useful once you have familiarised yourself with the layout. Information about a function (for example `read.table`) can be accessed by typing the following into the console:

```
help(read.table)
```
or
```
?read.table
```

This should include information about parameters that can be passed to the function, and at the bottom of the page should be examples that you can run which can be very useful.

If you don't know the function name that you're after, eg. for finding out the standard deviation, try

```
help.search("deviation")
```
or
```
??deviation
```

And you can always try searching the internet but remember that 'R' in a general search isn't always very good at returning relevant information so try and include as much information as possible.
Or go to http://www.rseek.org/ which will return more R specific information.

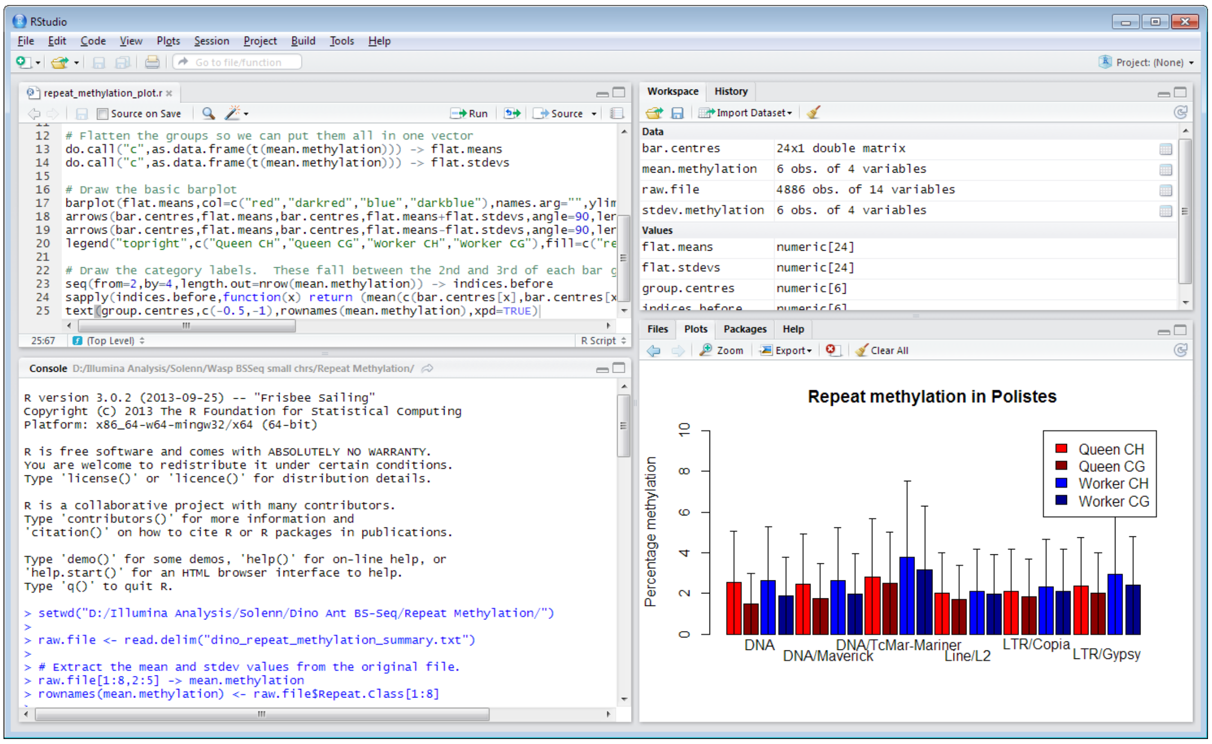
Getting familiar with the R console

R is a command line environment, this means that you type in an instruction and R interprets this and either stores the result or writes it back on the screen for you. You can run R in a very simple command shell window, but for all practical purposes it’s much easier to use a dedicated piece of software which makes it easy to work within the R environment.

There are several different R IDEs (integrated development environments) around, but by far the most common one is R studio and this is what we’re going to be using for this course.

Open RStudio. The default layout is shown below,

- Top left panel is the text editor. Commands can be sent into the console from here.
- Bottom left is the R console. You can type directly into here.
- Top right is the workspace and history. History keeps a record of the last commands entered, this is searchable. The workspace tab shows all the R objects (data structures).
- Bottom right is where graphs are plotted and help topics are shown.



```
# Flatten the groups so we can put them all in one vector
do.call("c",as.data.frame(t(mean.methylation))) -> flat.means
do.call("c",as.data.frame(t(mean.methylation))) -> flat.stdevs

# Draw the basic barplot
barplot(flat.means,col=c("red","darkred","blue","darkblue"),names.arg="",ylim
arrows(bar.centres,flat.means,bar.centres,flat.means+flat.stdevs,angle=90,ler
arrows(bar.centres,flat.means,bar.centres,flat.means-flat.stdevs,angle=90,ler
legend("topright",c("Queen CH","Queen CG","Worker CH","Worker CG"),fill=c("re

# Draw the category labels.  These fall between the 2nd and 3rd of each bar g
seq(from=2,by=4,length.out=nrow(mean.methylation)) -> indices.before
sapply(indices.before,function(x) return (mean(c(bar.centres[x],bar.centres[x
text(group.centres,c(-0.5,-1),rownames(mean.methylation),xpd=TRUE)
```

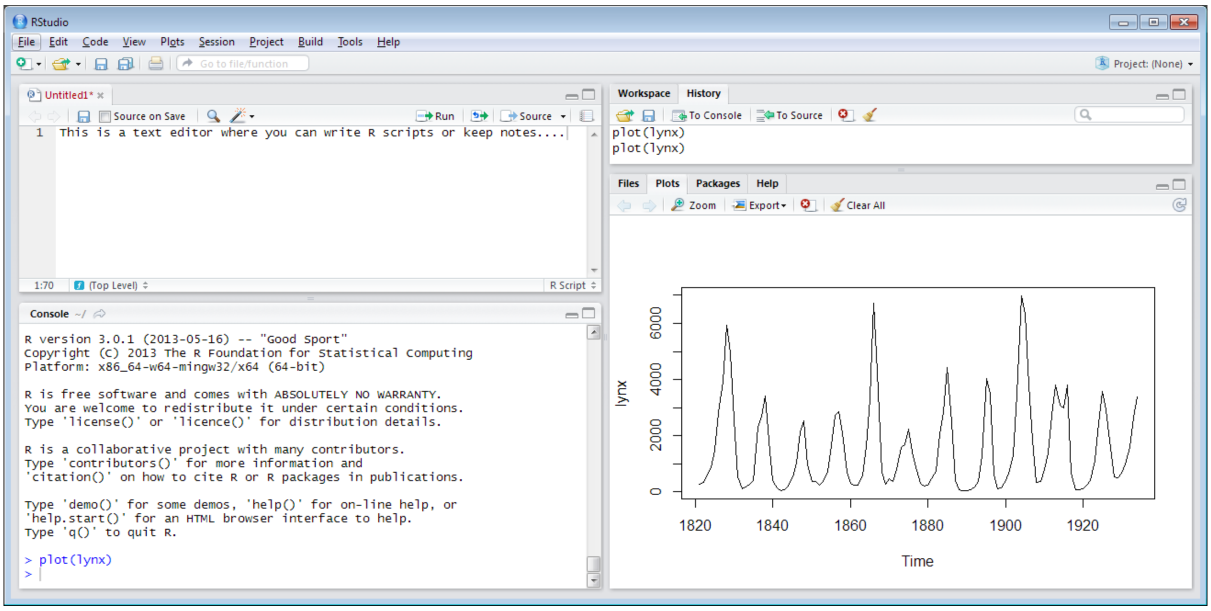
```
R version 3.0.2 (2013-09-25) -- "Frisbee Sailing"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

> setwd("D:/Illumina Analysis/Solenn/Dino Ant BS-Seq/Repeat Methylation/")
>
> raw.file <- read.delim("dino_repeat_methylation_summary.txt")
>
> # Extract the mean and stdev values from the original file.
> raw.file[1:8,2:5] -> mean.methylation
> rownames(mean.methylation) <- raw.file$Repeat.Class[1:8]
```

Data	
bar.centres	24x1 double matrix
mean.methylation	6 obs. of 4 variables
raw.file	4886 obs. of 14 variables
stdev.methylation	6 obs. of 4 variables
Values	
flat.means	numeric[24]
flat.stdevs	numeric[24]
group.centres	numeric[6]
indices.before	numeric[6]

The actual R session is the console window at the bottom left of the R-studio window. This is where the actual computation is done in R. All of the other windows are either keeping records of what you've done or showing you the result of a previous command.

When you type a command into the console it is evaluated by the R interpreter. If the result of this is a value then it will be printed in the console. Lots of data can be imported, manipulated and saved in an R session and though it won't always be visible on the screen, there are various ways of viewing and manipulating it.

If you create graphs these will open in a new window within the IDE as shown in the following screenshot. This also shows a text editor in which you can write anything you like including notes, though it would generally be used to create a script (i.e. lines of R commands). You can start a new text editor by selecting `File > New > RScript` from the menu bar.



```
This is a text editor where you can write R scripts or keep notes....
```

```
plot(lynx)
plot(lynx)
```

```
R version 3.0.1 (2013-05-16) -- "Good Sport"
Copyright (C) 2013 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

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

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

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

> plot(lynx)
> 
```

R Console, R Editor and R Graphics device open within the RStudio IDE

As well as issuing commands by typing directly into the console, you can also send commands to the R Console from the R Editor by selecting a command or a line of text and selecting Ctrl + Enter, or by copying and pasting.

In the console you can scroll through previous command that have been entered by using the up arrow ↑ on the keyboard.

The `>` symbol shows that R is ready for something to be entered. The console can work just like a calculator. Type `8+3` and press return. It doesn't matter whether there are spaces between the values or not.

```
> 8 + 3
[1] 11
```

The answer is printed in the console as above. We'll come on to what the `[1]` means at the end of this section.

```
> 27 / 5
[1] 5.4
```

These calculations have just produced output in the console - no values have been saved.

To save a value, it can be assigned to a data structure name. This is done by drawing an arrow, like `<-` The arrow points from the data you want to store towards the name you want to store it under. For now we'll use x, y and z as names of data structures, though more informative names can be used as discussed later in this section.

```
> x <- 8 + 3
```

If R has performed the command successfully you will not see any output, as the value of 8 + 3 has been saved to the data structure called x. You can access and use this data structure at any time and can print the value of x into the console.

```
> x
[1] 11
```

Create another data structure called y. In this case we'll draw the arrow the other way around. Functionally this is the same as the first example and it's up to you whether you do `data -> name` or `name <- data`.

```
> 3 -> y
```

Now that values have been assigned to x and y they can be used in calculations.

```
> x + y
[1] 14
> x * y
[1] 33

> x * y -> z
> z
[1] 33
```

R is case sensitive so x and X are not the same. If you try to print the value of X out into the console an error will be returned as X has not been used so far in this session.

```
> X
Error: object 'X' not found
```

To check what data structures you have created, enter `ls()` or `objects()` into the console or look at the 'workspace' tab in RStudio

If you use the same data structure name as one that you have previously used then R will overwrite the previous information with the new information.

```
> y <- 3
> y
[1] 3


> y <- 12
> y
```

```
[1] 12
```

Using an equals sign also works in most situations, eg `x = 8+3` but `<-` is generally preferred. You can also change the direction of the arrow if you want to calculate something and then send the result into a variable. The statements below are all functionally identical.

```
x<-3                   x  <-3                  x<-  3                 x <- 3

3->x                   3  ->x                  3->  x                 3 -> x
```

However, if you enter a space between the less than and minus characters that make up the assignment operator then you would be asking R a question that has a logical answer i.e. is `x` less than -5.

```
> x < - 5
[1] FALSE
```

Naming data structures

Data structures can be named anything you like (within reason), though they have to start with a letter. Having informative, descriptive names is useful and this often involves using more than one word. Providing there are no spaces between your words you can join them in various ways, using dots, underscores and capital letters though the Google R style guide recommends that names be joined with a full stop.

```
> mouse.age <- 112
> mouse.weight <- 25
> tail.length <- 54
```

These are all completely separate data structures that are not linked in any way.

Joining names by capitalising words is generally used for function names - don't worry about what these are for now - suffice to say it is not recommended to create names in the format `mouseAge`; use `mouse_age` or preferably `mouse.age`.

These names can be as long as you like, it just becomes more of a chore to type them the longer they get. RStudio helps with this in that you can press the tab key to try to complete any variable name you've started typing and it will complete it for you. Numbers can be incorporated into the name as long as the name does not begin with a number. Although you may not wish to use such convoluted names, the following are perfectly valid:

```
> tail.length.mouse.1 <- 41
> tail.length.mouse2.condition4.KO <- 35
```

Back to the strange `[1]` that appeared earlier. It happens again if we have a look at the tail length.

```
> tail.length <- 54
> tail.length
[1] 54
```

The [1] tells you that the index of the first value in the row shown is 1. This makes more sense when looking at larger data structures. R has some in-built datasets of which lynx is one.

```
> lynx
[1] 269 321 585 871 1475 2821 3928 5943 4950 2577 523 98
[13] 184 279 409 2285 2685 3409 1824 409 151 45 68 213
```

This time we've got a `[1]` and a `[13]`. There are 12 values in the first row, so on the second row of data shown above, the first value (184) is actually on the 13th row of the data structure. This can be seen more clearly by typing in `View(lynx)` which opens up a new window showing a read-only table of the data (as on the right).

This shows more clearly that the data structure is 1 column and many rows, where the value in the 13th row is 184.

Functions

Most of the work that you do in R will involve the use of functions. A function is simply a named set of code to allow you to manipulate your data in some way. There are lots of built in functions in R and you can also write your own if there isn't one which does exactly what you need.

Functions in R take the format `function.name(parameter 1, parameter 2 … parameter n)`. The brackets are always needed. Some functions are very simple and the only parameter you need to pass to the function is the data that you want the function to act upon.

For example, the function `dim(data.structure)` returns the dimensions (i.e. the numbers of rows and columns) of the data structure that you insert into the brackets, and it will not accept any additional arguments/parameters.

The square root and log2 functions can accept just one parameter as input.

```
> sqrt(245)
[1] 15.65248
```

```
> log2(15.65248)
[1] 3.968319
```

To keep it tidier the result of the square root function can be assigned to a named variable.

```
> x <- sqrt(245)
> log2(x)
[1] 3.968319
```

Alternatively, the two calculations can be performed in one expression.

```
> log2(sqrt(245))
[1] 3.968319
```

Most other functions will accept multiple parameters, such as `read.table()` for importing data.

If you're not sure of the parameters to pass to a function, or what additional parameters may be valid you can use the built in help to see this. For example to access the help for the `read.table()` function you could do:

```
?read.table
```

...and you would be taken to the help page for the function.

Working with data - Vectors

One thing which distinguishes R from other languages is that its most basic data structure is actually not a single value, but is an ordered set of values, called a vector. So, when you do:

```
> x <- 3
```

You're actually creating a vector with a length of 1. Vectors can hold many different types of data (strings, numbers, true/false values etc), but all of the values held in an individual vector must be of the same type, so it can be all numbers, or all strings but not a mix of the two.

To manually create a vector you can use the `c()` (short for combine) function which can take an arbitrary number of arguments and will combine them into a single vector. The only parameters that need to be passed to `c()` here are the data values that we want to combine.

```
> mouse.weights <- c(19, 22, 24, 18)
```

If words are used as data values they must be surrounded by quotes (either single or double - there's no difference in function, though pairs of quotes must be of the same type).

```
> mouse.strains <- c('castaneus', "black6", 'molossinus', '129sv')
```

If quotes are not used, R will try and find the data structure that you have referred to.

```
> mouse.strains <- c(castaneus, 'black6', 'molossinus', '129sv')
Error: object 'castaneus' not found
```

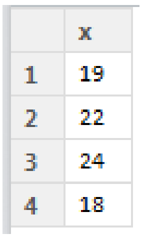
To access the whole data structure just type the name of it as before:

```
> mouse.weights
[1] 19 22 24 18

> mouse.strains
[1] "castaneus"  "black6"  "molossinus"  "129sv"
```

Or use:
`head(mouse.weights)`
To see just the first few lines of the vector.

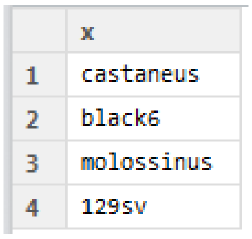
For a more graphical view you can also use the View function. In RStudio, the data is displayed in the text editor window. Both of these datasets are 4 element vectors.

```
> View(mouse.weights)
```



	x
1	19
2	22
3	24
4	18

```
> View(mouse.strains)
```



	x
1	castaneus
2	black6
3	molossinus
4	129sv

Data types in vectors

Within a vector all of the values must be of the same 'type'. There are four basic data types in R:

- Numeric - An integer or floating point number
- Character - Any amount of text from single letter to whole essay
- Logical - TRUE or FALSE values
- Factor - A categorised set of character values

Internally R distinguishes between integers (whole numbers) and floating point numbers (fractional numbers) but they are stored the same way.

Factors are the default way which R stores many pieces of text. They are used when grouping data for statistical or plotting operations and in many cases are interchangeable with characters, but there are differences, especially when merging or sorting data which can cause problems if you use the wrong type for this kind of data.

To see what type of data you're storing in a vector you can either look in the workspace tab of RStudio or you can use the class function.

```
> class(mixed.frame$numbers)
[1] "character"

> class(c(1,2,3))
[1] "numeric"

> class(c("a","b","c"))
[1] "character"

> class(c(TRUE,TRUE,FALSE))
[1] "logical"
```

Functions for making vectors

Although you can make vectors manually using the `c()` function, there are also some specialised functions for making vectors. These provide a quick and easy way to make up commonly used series of values.

The `seq()` function can be used to make up arithmetic series of values. You can specify either a start, end and increment (`by`) value, or a start, increment (`by`) and length (`length.out`) and the function will make up an appropriate vector for you.

```
> seq(from=5,to=10,by=0.5)
 [1]  5.0  5.5  6.0  6.5  7.0  7.5  8.0  8.5  9.0  9.5 10.0

> seq(from=1,by=2,length.out=10)
 [1]  1  3  5  7  9 11 13 15 17 19
```

The `rep()` function simply repeats a value a specified number of times.

```
> rep("hello",5)
[1] "hello" "hello" "hello" "hello" "hello"
```

Finally, there is a special operator for creating vectors of sequential integers. You simply separate the lower and higher values by a colon to generate a vector of the intervening values.

```
> 10:20
 [1] 10 11 12 13 14 15 16 17 18 19 20
```

You can also combine these functions with `c()` to make up more complicated vectors

```
c(rep(1,3),rep(2,3),rep(3,3))
[1] 1 1 1 2 2 2 3 3 3
```

Accessing Vector Subsets

To access specific positions in a vector you can put square brackets after it, and then use a vector of the index positions you want to retrieve. Note that unlike most other programming languages index counts in R start at 1 and not 0. To view the 2nd value in the data structure:

```
> mouse.strains[2]
[1] "black6"
```

Note, that in this instance the number `2` in the above expression is actually just a shortcut for `c(2)`, so what we're pulling out are a vector of index positions. We can therefore also use the automated ways to make vectors of integers to easily pull out larger subsets. To view a range of values we could use the lower:higher notation we saw above:

```
> mouse.strains[2:4]
[1] "black6"  "molossinus"  "129sv"
```

We should think of the statement above as two separate operations. We use `2:4` to make a vector with 2,3,4 in it, and then we put that into square brackets to select the corresponding values from `mouse.strains`. To view or select non-adjacent values the `c()` function can be used again. To view the 2nd and the 4th values:

```
> mouse.strains[c(2,4)]
[1] "black6"   "129sv"
```

Remember that to create a vector you need to use the c function. If you leave this out (which is really easy to do) you'll get an error:

```
> mouse.strains[2,4]

Error in mouse.strains[2, 4] : incorrect number of dimensions
```

Accessing vectors using names

In all R data structures you have the option of assigning names to numeric positions so that you can use the name to access the data instead of the position. For vectors you read and assign names using the names function. When you first create a vector there won't be any names associated with it, but you can assign some and then use them in the places where you would otherwise use the positions – using the same square bracket notation.

```
> c(180,190,170) -> people.heights

> names(people.heights)

NULL
```

We assign names by treating the function as a variable and assigning data to it. This style of assignment is very common in R.

```
> names(people.heights) <- c("laura","simon","anne")
```

We can now see that the names are associated with the values in the vector.

```
> people.heights
laura simon  anne
  180   190   170
```

We can also use the names to retrieve the corresponding values. Even though names are assigned we can still use index positions too.

```
> people.heights["simon"]
simon
  190
> people.heights[c("simon","laura")]
simon laura
  190   180
> people.heights[c(2,1)]
simon laura
  190   180
```

Vectorised Operations

The other big difference between R and other programming languages is that normal operations are designed to be applied to whole vectors rather than individual values. This means that you can very quickly and easily apply changes to whole sets of data without having to write complex code to loop through individual values.

If we wanted to log transform a whole dataset for example then we can do this using a single operation.

```
> data.to.log <- c(1,10,100,1000,10000,100000)
> log10(data.to.log)
[1] 0 1 2 3 4 5
```

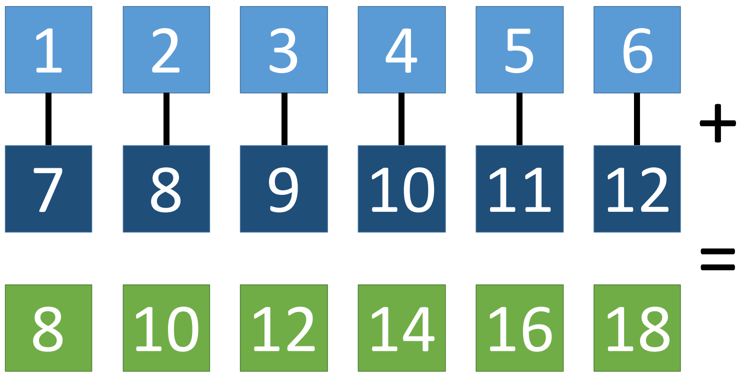
Here we passed a vector to the log10 function and we received back a vector of the log10 transformed versions of all of the elements in that vector without having to do anything else. We can use this for other operations too:

```
> data.to.log + 1
[1]      2     11    101   1001  10001 100001
> (data.to.log +1) * 2
[1]      4     22    202   2002  20002 200002
```

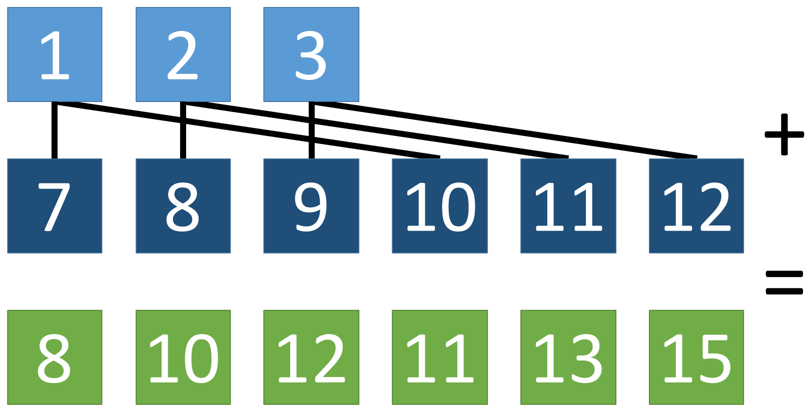
You can also use two vectors in any mathematical operation and the calculation will be performed on the equivalent positions between the two vectors. If one vector is shorter than the other then the calculation will 'wrap round' and start again at the beginning (but you will get a warning if the longer vector's length isn't a multiple of the shorter vector's length).

```
> x <- 1:10
> y <- 21:30
> x+y
 [1] 22 24 26 28 30 32 34 36 38 40

> j <- c(1,2)
> k <- 1:20
> k*j
 [1]  1  4  3  8  5 12  7 16  9 20 11 24 13 28 15 32 17 36 19 40
```

It is important to understand how operations involving two vectors work since this ends up being a critical aspect of many parts of R. The basic rules are that if two vectors are the same length, then equivalent indices are paired together.



If they are not the same length then the shorter vector is recycled, but it is expected that the length of the longer vector will be a multiple of the length of the shorter vector (you’ll get a warning if it isn’t).



Working with data - Lists

Vectors are a core part of R and are extremely useful on their own, but they are limited by the fact that they can only hold a single set of values and that all values must be of the same type. To build more complex data structures we need to look at some of the other data structures R provides.

The next level of organisation you can have in R is a list. A list is simply a collection of vectors, sort of like a vector of vectors but with some extra features. A list is created from a set of vectors, and whilst each vector is a fixed type, different vectors within the same list can hold different types of data.

Once a list is created the vectors in it can be accessed using the position at which they were added. You can also assign a name to each position in the list and then access the vector using that name rather than its position, which can make your code easier to read and more robust if you choose to add more vectors to the list at a later date.

As an example, the two data structures `mouse.strains` and `mouse.weights` can be combined into a list using the `list()` function.

```
> mouse.data <- list(weight=mouse.weights,strain=mouse.strains)
```

If you now view this list you will see that the two sets of data are now stored together and have the names weight and strain associated with them.

```
> mouse.data
$weight
[1] 19 22 24 18

$strain
[1] "castaneus"  "black6"     "molossinus" "129sv"
```

To access the data stored in a list you can use either the index position of each stored vector or its name. You can use the name either by putting it into square brackets, as you would with an index, or by using the special `$` notation after the name of the list (which is generally the nicer way to do this).

```
> mouse.data[[1]]
[1] 19 22 24 18

> mouse.data[["weight"]]
[1] 19 22 24 18

> mouse.data$weight
[1] 19 22 24 18
```

In this case, because we’re accessing a position in a list rather than a vector we use double square brackets.

Once you’re retrieved the vector from the list you can use conventional square bracket notation to pull out individual values.

```
> mouse.data$weight[2:3]
[1] 22 24
```

Lists can be useful but they are really just a collection of vectors, and there is no linkage between positions in the different vectors stored within a list (`mouse.data$weight[1]` has no connection to `mouse.data$strain[1]`), in fact the vectors stored in a list don't even have to be the same length. This is perfectly valid.

```
> names <- list(first=c("Bob","Dave"),last=c("Smith","Jones","Baker"))
> names
$first
[1] "Bob"  "Dave"

$last
[1] "Smith" "Jones" "Baker"
```

Since most real data occurs in a more regular table structure we need a different data structure to represent this.

Working with data – Data frames

By far the most common data structure used for real data sets in R is the data frame. In essence a data frame is just a list, but with one important difference which is that all of the vectors stored in a data frame are required to have the same length, so you create a 2D table type structure, where all of the columns store the same type of data.

Because of this guarantee a data frame can then add in some functionality to access data from within this 2D structure in a convenient manner and makes it easy to perform operations which use the whole of the data structure.

Data frames are created in the same way as lists, with columns being either named or simply accessed by position.

```
> mouse.dataframe <- data.frame(weight=mouse.weights,strain=mouse.strains)

> mouse.dataframe
  weight     strain
1     19  castaneus
2     22     black6
3     24 molossinus
4     18      129sv

> mouse.dataframe[[1]]
[1] 19 22 24 18

> mouse.dataframe$weight
[1] 19 22 24 18
```

However, data frames have some extra functionality which allows you to extract subsets of data directly from within the 2D structure. Specifically the normal bracket notation for extracting data from a vector has been extended so that you can supply two values to specify which rows and columns you want to extract from a data frame. You can do something simple such as:

```
> mouse.dataframe[2,1]
[1] 22
```

You can choose to select all values from one dimension by leaving out the index value:

```
> mouse.dataframe[,1]
[1] 19 22 24 18
```

You can also use the same range selectors we saw before but in two dimensions:

```
> mouse.dataframe[2:4,1]
[1] 22 24 18
```

There are also some extra functions you can use with data frames. You can use the `dim()` function to find out how many rows and columns your data frame has, and you can use `nrow()` and `ncol()` to get these values individually.

```
> dim(mouse.dataframe)
[1] 4 2

> nrow(mouse.dataframe)
[1] 4

> ncol(mouse.dataframe)
[1] 2
```

We saw before that you can set a list or a data frame up with named columns and use these to access the vectors which are stored there. For data frames you can also define named rows and use these names in a similar way. You can then use `rownames()` and `colnames()` to either access or set the row and column names.

```
> test.frame <- data.frame(0:5,100:105,200:205)
> test.frame
  X0.5 X100.105 X200.205
1    0      100      200
2    1      101      201
3    2      102      202
4    3      103      203
5    4      104      204
6    5      105      205

> colnames(test.frame) <- c("ColA","ColB","ColC")
> rownames(test.frame) <- c("RowA","RowB","RowC","RowD","RowE","RowF")

> test.frame
     ColA ColB ColC
RowA    0  100  200
RowB    1  101  201
RowC    2  102  202
RowD    3  103  203
RowE    4  104  204
RowF    5  105  205

> test.frame["RowD","ColB"]
[1] 103
```

Reading and Writing data from files

Getting and setting the working directory

If you don't want to write out the full path file each time that you want to read in and write out files, you can set the directory that you're working in. To check which directory you're working in at the moment:

```
> getwd()
 [1] "m:/"
```

To set your working directory:

```
setwd("the/place/I/want/to/read/and/write/files")
```

Once you have started to type the path you can again use the tab key to get RStudio to show you possible completions so you don't have to type the whole thing.

Alternatively, RStudio allows you to set the working directory through the menu options.

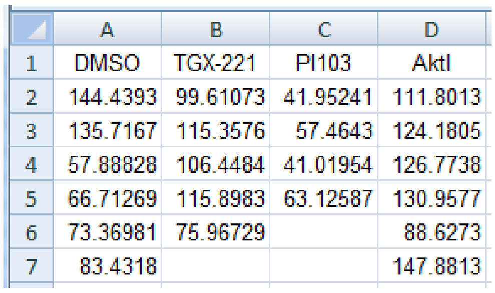
```
Session > Set Working Directory > Choose Directory
```

Importing data

R has a few different functions for importing data files. We will use `read.table()` and variations of this.

`read.table()` reads in a table of data from a file and creates a data structure from it. There are various options that can be set to make sure that the data is imported correctly and that it is in the format that you require.

The file neutrophils.csv contains data on angiogenesis under different conditions. Shown below is a section of the dataset in Excel.



	A	B	C	D
1	DMSO	TGX-221	PI103	Aktl
2	144.4393	99.61073	41.95241	111.8013
3	135.7167	115.3576	57.4643	124.1805
4	57.88828	106.4484	41.01954	126.7738
5	66.71269	115.8983	63.12587	130.9577
6	73.36981	75.96729		88.6273
7	83.4318			147.8813

If you open neutrophils.csv in a simple text editor such as notepad you can see that the values are separated by commas.

When using `read.table()` R has to be told what the separating characters are by using the parameter `sep`.

`sep = "\t"` for tab delimited files
`sep = ","` for comma separated files

We can see in the Excel and Notepad screenshots that the data has column headings. The default setting in `read.table()` is `header = FALSE` i.e. that there are no column headings. We therefore need to set `header = TRUE` so that the column names are not read in as part of the dataset values.
To read in the file neutrophils.csv which, as we've established is comma separated:

```
> neutrophils <- read.table("neutrophils.csv", sep=",", header = TRUE)
```

This will only read in the neutrophils.csv file if the file is in your current working directory. Remember to use `getwd()` to check which directory you're currently working in and `setwd()` to change it. You can still access files that are saved in a different location to your current working directory but you will need to include the file path in the command.

```
> neutrophils <-  read.table("neutrophils.csv", sep=",", header = TRUE)
```

If the `sep` parameter is left blank it will default to spaces. You can see the default parameters and more information about these by looking at the help page `?read.table`.

To view the first few rows of the data, type `head(neutrophils)`. This is very useful when dealing with large datasets when you wouldn't want the whole dataset printed onto the screen. By default this will show the first 6 rows of the object, though more can be viewed by specifying a number.

```
> head(neutrophils,10)
        DMSO   TGX.221    PI103      Akt1
1  144.43930  99.61073 41.95241 111.80130
2  135.71670 115.35760 57.46430 124.18050
3   57.88828 106.44840 41.01954 126.77380
4   66.71269 115.89830 63.12587 130.95770
5   73.36981  75.96729       NA  88.62730
6   83.43180        NA       NA 147.88130
7   97.41048        NA       NA  72.68707
8   97.91444        NA       NA  82.73766
9  107.97630        NA       NA  49.60179
10 113.44100        NA       NA        NA
```

Similarly, `tail()` can be used to view the last few rows.

```
> tail(neutrophils)
        DMSO TGX.221 PI103 Akt1
12 132.62770      NA    NA   NA
13  75.49360      NA    NA   NA
14 103.82920      NA    NA   NA
15 104.77120      NA    NA   NA
16  95.34444      NA    NA   NA
17 113.07590      NA    NA   NA
```

You can see that R has inserted NA values into the blank fields.
We've already used `dim()` to find out the dimensions of a data structure. To see some basic statistics on the data use `summary()` .

```
> summary(neutrophils)
  DMSO             TGX.221            PI103            Akt1
 Min.   : 57.89   Min.   : 75.97   Min.   :41.02   Min.   : 49.60
 1st Qu.: 83.43   1st Qu.: 99.61   1st Qu.:41.72   1st Qu.: 82.74
 Median :103.83   Median :106.45   Median :49.71   Median :111.80
 Mean   :100.80   Mean   :102.66   Mean   :50.89   Mean   :103.92
 3rd Qu.:113.08   3rd Qu.:115.36   3rd Qu.:58.88   3rd Qu.:126.77
 Max.   :144.44   Max.   :115.90   Max.   :63.13   Max.   :147.88
                  NA's   : 12.00   NA's   :13.00   NA's   :  8.00
```

This shows the minimum, lower quartile, median, mean, upper quartile and maximum values for each column.

Using read.delim or read.csv instead of read.table

Whilst `read.table` is the core function to read data from a file there are a couple of shortcuts which make your life easier. If you look in the documentation for `read.table` you will see that there are functions called `read.delim` and `read.csv`. These both call `read.table`, but with a bunch of options already set for you. Specifically they set up the delimiter to be tab or comma, and they set the header to `TRUE` so that you don't have to do this each time.

Most of the time it's therefore easier to use one of these functions instead of using `read.table` directly.

Notes about reading files

Be aware that if you have spaces in row or column names and haven't changed the default column separator (either by setting `sep=` or by using `read.delim` or `read.csv`) then R will read the words as separate entities and this will probably cause your data to be read in incorrectly.

As well as typing in the file paths you can also select files or folders interactively by using `file.choose()`, `choose.files()` or `dir.choose()`.

```
> x <- file.choose()
```

This brings up a file navigator window to choose a file from.

```
> x
[1] "M:\\R_course_files\\neutrophils.txt"

> data <- read.table(x, sep = "\t", header = T)
> head(data)

       DMSO   TGX.221    PI103     Akt1
1 144.43930  99.61073 41.95241 111.8013
2 135.71670 115.35760 57.46430 124.1805
3  57.88828 106.44840 41.01954 126.7738
4  66.71269 115.89830 63.12587 130.9577
5  73.36981  75.96729       NA  88.6273
6  83.43180        NA       NA 147.8813
```

Another problem which is fairly common is that you have a text file which somewhere contains an unmatched quote character, something like:

```
      Name [tab] Organisation  [tab] Age
      Bob  [tab] Smith et al.  [tab] 35
      John [tab] "Acme devices [tab] 26
```

In this case you'll end up with huge chunks of your file stuck together in the same field as the program doesn't realise that the second item on John's line has finished. If you have a file which isn't quoting internal fields then it's safer to add `quote=""` to the read options to avoid trying to interpret these types of mis-matched quotes.

An alternative for small datasets - copy and paste

If you've got a small amount of data that you're looking at in Excel that you want to insert into R, you can effectively copy and paste. Select the data, copy it, go into R and enter the text below into the console. You may need to set `header = FALSE`.

```
> my.data <- read.table("clipboard", sep = "\t")
```

This will also work if you copy data from a text based application, though be careful about line endings if you're copying data from somewhere like notepad.

Writing data

In the same way as you can use `read.table` to read data from a file into a data frame you can use the corresponding `write.table` function to write the data from a data frame back into a text file. To make sure your headers and data line up there are generally two forms of the command you will use.

If you don't have any row names set up for your data frame use the command below. This will not write out the numeric row names and will just include your data.

```
write.table(data,file="out.txt",sep="\t",quote=FALSE,row.names=FALSE)
```

If you do have row names then use the command below which will keep the row names and will move the column names so they line up correctly with the data.

```
write.table(data,file="out.txt",sep="\t",quote=FALSE,col.names=NA)
```

Logical Tests and Filtering

Data subset access recap

We've seen in the earlier chapters that there are a number of ways to access data stored in vectors or data frames using the square bracket notation.

```
> a <- 101:110 # Set up a simple vector
> a[3]         # Access a single value
[1] 103
> a[3:5]       # Access a continuous range of values
[1] 103 104 105
> a[c(1,7,4)]  # Access a discontinuous range of values
[1] 101 107 104
```

There is however another way to access values, and it's one we can use to create logical data subsets. One of the data types you can store within R are logical values (TRUE or FALSE), and you can access subsets of a vector using a vector of logical values to say whether you want to include each position.

```
> test <- 1:5
> logical.values <- c(TRUE,FALSE,FALSE,TRUE,TRUE)
> test[logical.values]
[1] 1 4 5
```

On its own this wouldn't be hugely useful, but we can use this in combination with logical tests to filter our data.

Logical tests

A logical test is simply a mathematical expression which can be applied to a value to produce either a true or false result. Simple examples would be testing if one value equals another value, or if one value is higher than another value. Each of these types of test will return a logical vector in R.

```
> test <- 1:10
> test == 5
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE

> test > 4
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE

> test <= 8
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

Note that the test for equality is == (with two equals signs). Using a single equals will assign the value of 5 to test (which is another reason why using -> for assignment is a good idea).

The real power of these types of test comes from using the resulting logical vectors to pull out the associated data values.

```
> test <- 1:10
> logical.array <- test <= 7
> test[logical.array]
[1] 1 2 3 4 5 6 7
```

This method unnecessarily creates a named logical vector. It's much more common to simply put the logical test inside the square brackets

```
> test[test<=7]
[1] 1 2 3 4 5 6 7
```

We can extend the same principle to data frames where this approach is even more powerful and useful.

```
> expr.data
             ExprA      ExprB pValue
Mia1      5.832890  3.2442384  1e-01
Snrpa     8.592457  5.0218460  1e-03
Itpkc     8.487840  6.1648040  4e-02
Adck4     7.693487  6.4141630  2e-01
Numbl     8.366323  6.8119226  1e-01
Ltbp4     6.960445 10.4294070  1e-03
Shkbp1    7.569856  5.8292007  1e-01
Spnb4    10.653741  9.3793970  2e-01
Blvrb     7.321928  5.2948647  5e-02
Pgam1     0.000000  0.2848804  5e-01
Sertad3   8.129283  3.0218458  1e-04
```

In this case the gene names are used as the row names for the data frame. If we wanted to pull out the names of genes with a significant change we therefore need to get a subset of the output of `rownames()`

```
> rownames(expr.data)[expr.data$pValue <= 0.01]
[1] "Snrpa"   "Ltbp4"   "Sertad3" "Sertad1"
```

We can also subset the entire dataset. In this example we'll find the absolute difference between the two expression values and then filter this to produce a logical vector which we can use to subset the whole data frame.

Note that we need a trailing comma in the square brackets since we're only filtering the rows of the data frame. Leaving the columns set blank will mean that we keep all of the original columns when we generate the subset, but we will have removed some of the rows.

```
> expr.data[abs(expr.data$ExprA-expr.data$ExprB)>2,]
           ExprA     ExprB pValue
Mia1    5.832890  3.244238  1e-01
Snrpa   8.592457  5.021846  1e-03
Itpkc   8.487840  6.164804  4e-02
Ltbp4   6.960445 10.429407  1e-03
Blvrb   7.321928  5.294865  5e-02
Sertad3 8.129283  3.021846  1e-04
Sertad1 7.693487  4.343774  1e-02
```

Summary of logical test operators

==	equal to	**!=**	not equal to
&	and	**|**	or
<	less than	**>**	greater than
<=	less than or equal to	**>=**	greater than or equal to

Filtering using the subset() function

Using logical vectors can be a very effective way of filtering data. Complex commands can be constructed by combining multiple logical tests, allowing powerful filtering of datasets.

In cases where only fairly simple filtering is required, the `subset()` function can be used instead. This uses a relatively simple syntax and so may be preferable to the method described in the previous section.

In the previous example we found the absolute difference between the two expression values and then filtered this to produce a logical vector which was then used to subset the whole data frame.

We can use the subset function to perform the same filtering. The first argument to the subset function is the entire dataset, the 2nd argument is the subset (a logical expression indicating elements or rows to keep: missing values are taken as false)

```
> subset(expr.data, abs(ExprA-ExprB)>2)
           ExprA     ExprB pValue
Mia1    5.832890  3.244238  1e-01
Snrpa   8.592457  5.021846  1e-03
Itpkc   8.487840  6.164804  4e-02
Ltbp4   6.960445 10.429407  1e-03
Blvrb   7.321928  5.294865  5e-02
Sertad3 8.129283  3.021846  1e-04
Sertad1 7.693487  4.343774  1e-02
```

To filter the whole expr.data dataset by p-value:

```
> subset(expr.data, pValue <=0.01)
           ExprA     ExprB pValue
Snrpa   8.592457  5.021846  1e-03
Ltbp4   6.960445 10.429407  1e-03
Sertad3 8.129283  3.021846  1e-04
Sertad1 7.693487  4.343774  1e-02
```

In these examples, the results are only being printed to the screen, the subset function does not modify the `expr.data` dataset itself. To save a filtered version of the dataset we would need to assign a name to it.

```
> expr.filtered.by.pval <- subset(expr.data, pValue <=0.01)
> expr.filtered.by.pval
           ExprA     ExprB pValue
Snrpa   8.592457  5.021846  1e-03
Ltbp4   6.960445 10.429407  1e-03
Sertad3 8.129283  3.021846  1e-04
Sertad1 7.693487  4.343774  1e-02
```

Graphs

One of the most useful functionalities in R is its ability to generate publication quality graphs. R contains a number of powerful and flexible tools for graphing which either allow you to build simple graphs quickly, or to have the ability, with a bit of code, to generate complex multi-layer graph.

The ability to generate figures and graphs is a core part of the R language and you achieve this by running functions. Up until now all of the functions we've used have generated data as their output, but some functions generate graphs. In all other respects these functions are the same as the ones we've used before. They have help pages, they take arguments, and the arguments they take will alter the way in which the function works. There's actually nothing new in terms of R syntax to learn to be able to draw figures and graphs.

Boxplots

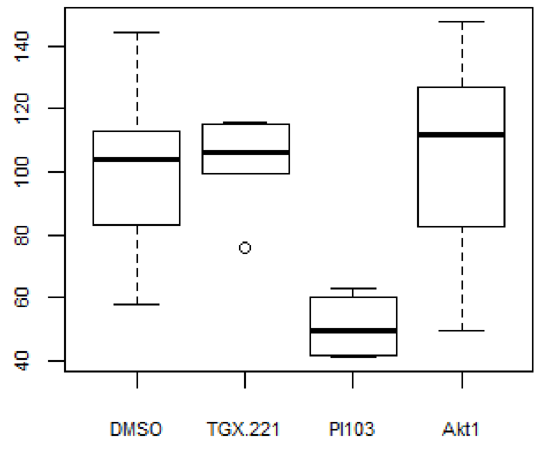
Boxplots or box and whisker diagrams are useful for viewing a summary of the distribution of data values. By default R will plot a box with the horizontal lines representing versions of the 25th percentile, median, and 75th percentile. Whiskers extend from the box to the lowest and highest values, though any values that are considered by R to be outliers are not included in the whiskers but plotted as separate points.

The default parameters can be overridden to change the representation of outliers.

For a simple boxplot:

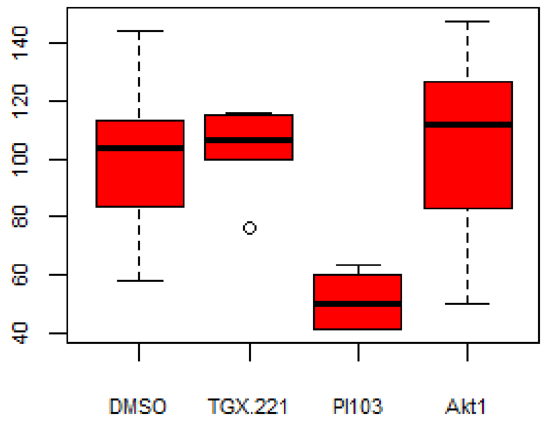
```
> boxplot(neutrophils)
```

This is using all default parameters but to customise the graph parameters can be changed. We'll just set the colours for now.



To make each of the boxes the same colour, pass one colour to the `col` parameter.

```
> boxplot(neutrophils, col = 'red')
```

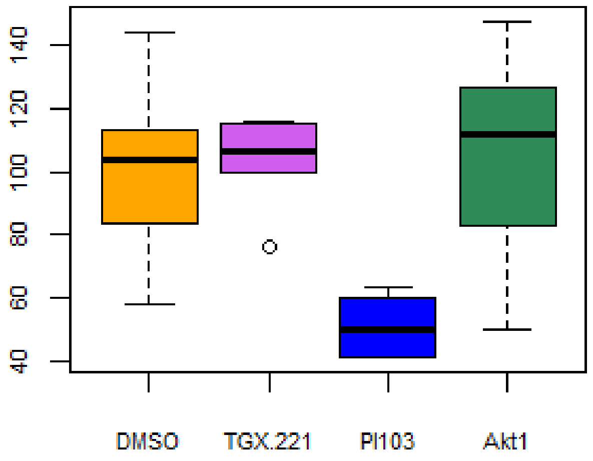


To make each box a different colour, we need to pass 4 colours to the `col` parameter. Colours can be specified in several different ways. One way is used above, by specifying a colour name (e.g. 'red' or 'blue'). A list of colour names can be accessed by typing `colours()` or `colors()` into the console. If we want to create a data structure called `colour.names` of 4 colour names:

```
> colour.names <- c('orange', 'mediumorchid2', 'blue', 'seagreen')
```

(There are quite a range of colours available.)

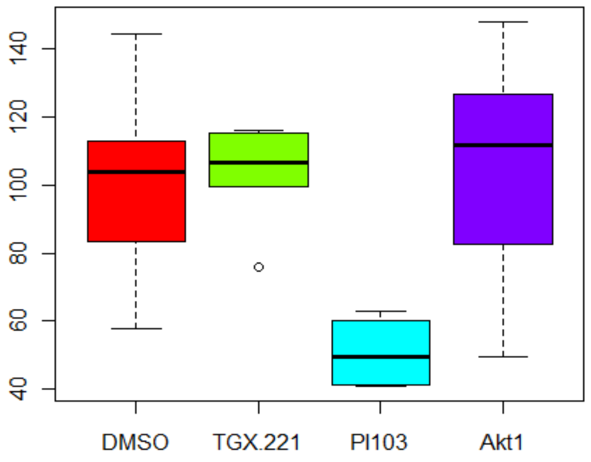
```
> boxplot(neutrophils, col =
      colour.names)
```



R also supplies various colour palettes; details of these can be viewed by entering ?rainbow into the console.

These can be a quick and easy way of accessing many colours, you just have to specify how many colours you want. So, for rainbow colours:

```
> rainbow.colours <- rainbow(4)
> boxplot(neutrophils, col =
rainbow.colours)
```



The following command would produce exactly the same output as the command above:

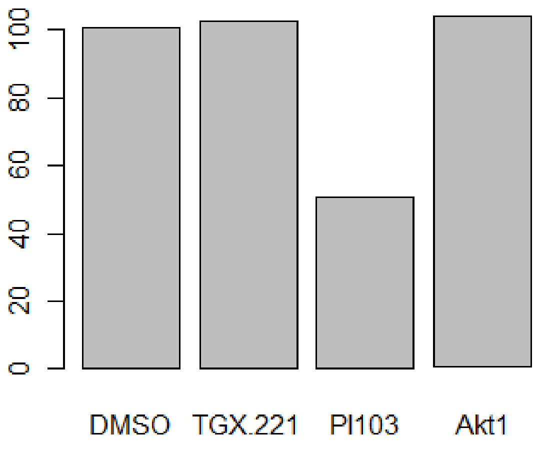
```
> boxplot(neutrophils, col = rainbow(4))
```

Barplot

To create a simple barplot we need a single value to plot for each bar so we will calculate the means of the neutrophils dataset. We calculated the mean of a column earlier, now we can calculate the means of all the columns at once.

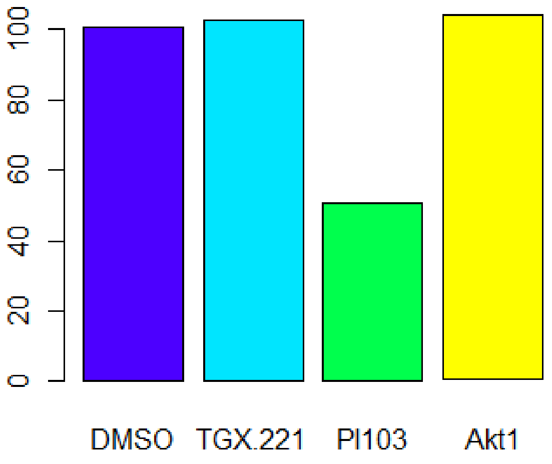
`colMeans()` is a quick way to calculate the mean for one or more columns.

```
> neutrophil.means <-
colMeans(neutrophils, na.rm = T)


> neutrophil.means
   DMSO  TGX.221    PI103     Akt1
 100.80   102.66    50.89   103.92

> barplot(neutrophil.means)
```



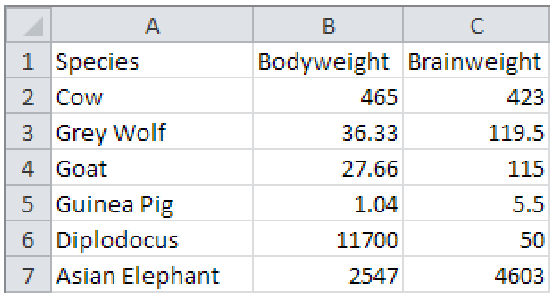
Colours in barplot work in the same way as for the boxplot. topo.colors() is another colour palette like rainbow.

```
> my.colours <- topo.colors(4)

> barplot(neutrophil.means, col =
           my.colours)
```



Scatterplots

Scatterplots can be useful for looking at correlations between variables. This time we'll read in a file on brain and bodyweights of 27 animals.



	A	B	C
1	Species	Bodyweight	Brainweight
2	Cow	465	423
3	Grey Wolf	36.33	119.5
4	Goat	27.66	115
5	Guinea Pig	1.04	5.5
6	Diplodocus	11700	50
7	Asian Elephant	2547	4603

As you can see in the snapshot of the dataset in Excel, there are 3 columns; the first column being the names of the species.

If we want the names of the species to be row names in the data structure in R, the parameter `row.names` can be set to `row.names = 1`, where 1 is the column number (in the original data file) that we want to use for the row names.

```
> brain.bodyweight <- read.delim("brain_bodyweight.txt", row.names = 1)

> head(brain.bodyweight)
                Bodyweight Brainweight
Cow                 465.00       423.0
Grey Wolf            36.33       119.5
Goat                 27.66       115.0
Guinea Pig            1.04         5.5
Diplodocus        11700.00        50.0
Asian Elephant     2547.00      4603.0

> dim(brain.bodyweight)
[1] 27  2
```

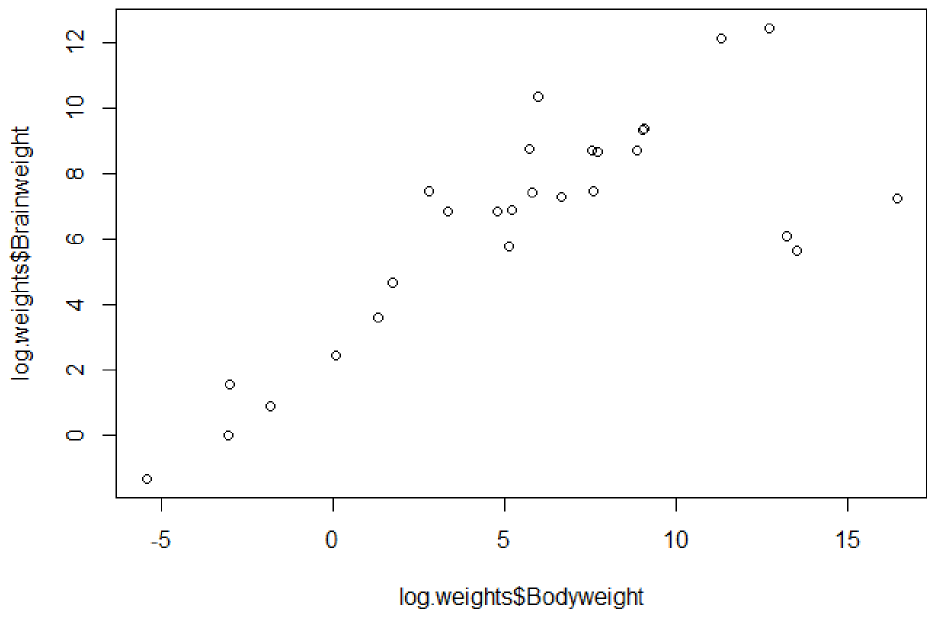
So our data structure is 27 rows by 2 columns; the row names are an additional part of the structure.

To make sense of this data we're going to need to log transform both the brain and bodyweight data. Since we're transforming the whole of the data frame we can do this in a single operation.

```
> log.weights <-   log2(brain.bodyweight)
```

The input to the `plot` function can be a single data structure if it's in the right format i.e. x values in the first column and y values in the 2nd column, or you can specifically pass the two datasets.

```
> plot(log.weights$Bodyweight, log.weights$Brainweight)
```



This plot uses default parameters; it can be customised by adjusting a wide range of parameters, the details of which are not covered in this course, though they can be found in the help files for `plot` and `par`.

Line Graphs

Line graphs are a variation of the scatterplot function.
This time we'll use a different dataset which has zscores for an ABL1 dataset at 100,000 bp intervals across a chromosome.

```
> chr.data <- read.delim("chr_data.txt")
> head(chr.data)
  chr position        ABL1
1   8        1 -0.04661052
2   8   100001  0.25605089
3   8   200001  0.25605089
4   8   300001  0.86137348
5   8   400001  0.86137348
6   8   500001  0.86137348

> dim(chr.data)
[1] 974   3
```

The chromosome column (column 1) is irrelevant for what we're looking at so we can remove this. We can do this by selecting the columns we want to keep and then saving over the original dataset.

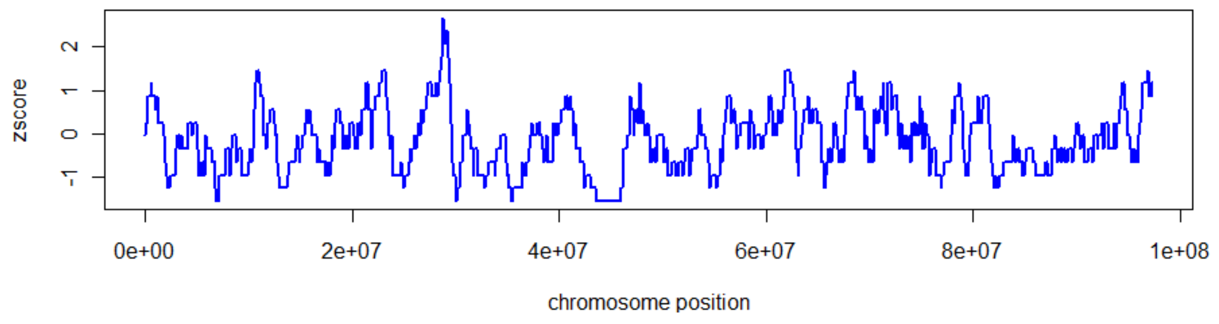
```
> chr.data <- chr.data[,2:3]
```

Check the adjusted dataset.

```
> head(chr.data)
  position        ABL1
1        1 -0.04661052
2   100001  0.25605089
3   200001  0.25605089
4   300001  0.86137348
5   400001  0.86137348
6   500001  0.86137348
```

We want to plot the chromosome position along the x axis and the zscores for the second column (ABL1) on the y axis. To produce a line graph we need to set the plot `type = 'l'` (that's a lower case L not a 1). `lwd` sets the line width.

```
> plot(x = chr.data[,1], y = chr.data[,2], type = 'l', col= 'blue', lwd= 2,
         xlab = 'chromosome position', ylab = 'zscore')
```

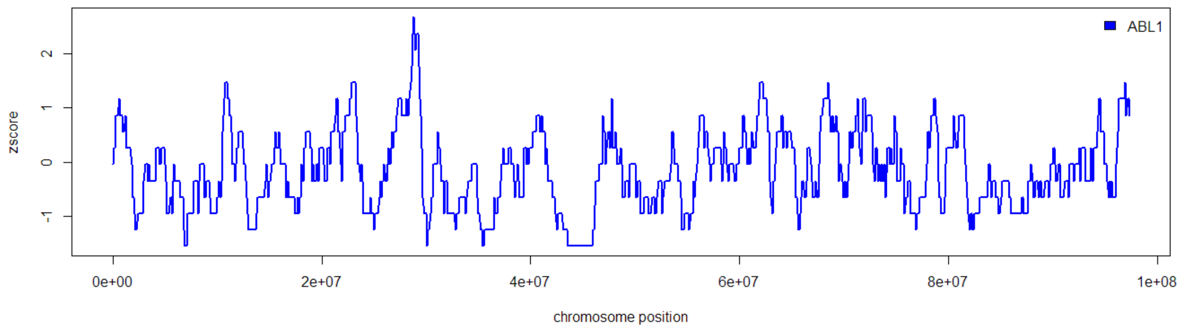


After drawing the initial plot we can then add more information on top of the existing data. This could be other datasets, titles, text, shapes or legends. More information about these types of complex graphs is covered in our advanced plotting course.

To add a legend to the plot area use the `legend()` function. You can either specify x and y coordinates that are used for the top left corner of the legend box, or use a keyword such as `"bottomleft"`, `"topright"`, `"center"` etc.

```
> legend("topright","ABL1",fill="blue",bty="n")
```

fill refers to the colour boxes that the legend creates. The colours can be set up as a data structure as we've done with the legend text, or it can just be created in the main command. bty refers to the box type around the legend, bty = "n" removes the box.

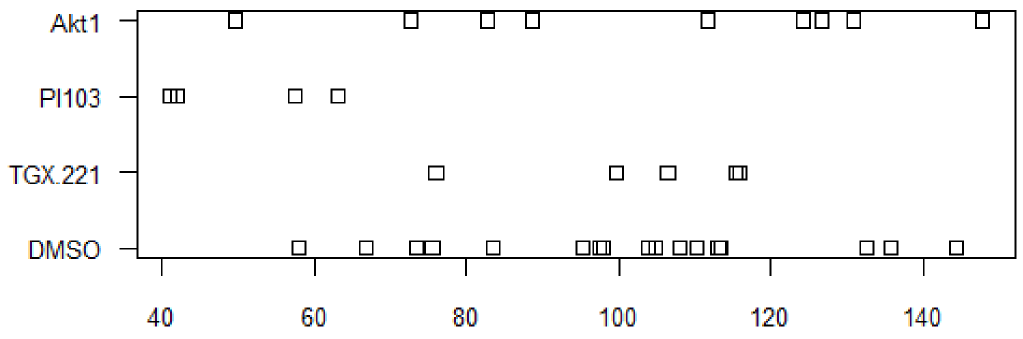


You can play around with the location of the legend, change the box around it, change the colour and size of the text etc. For more details see the legend help page.

Stripchart

This is useful way of plotting data if the datasets aren't too large.

```
> stripchart(neutrophils)
```

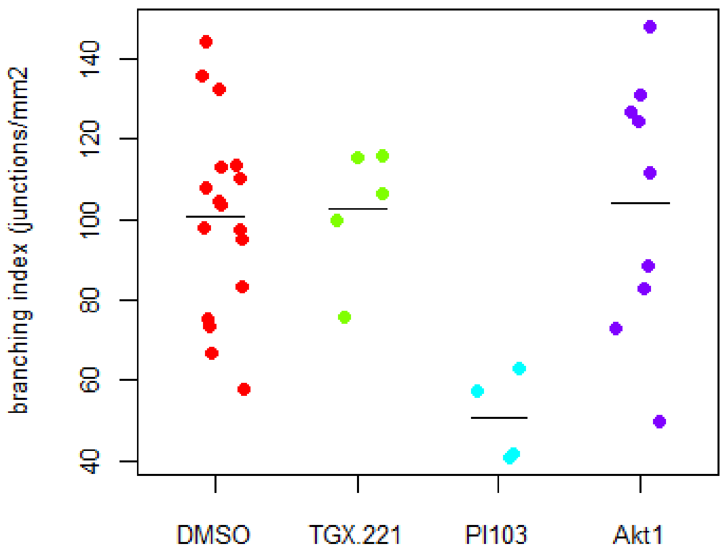


The default parameters don't produce a pretty graph, but with some adjustments it can be made to look very different. Check the stripchart help page for more information.

```
> stripchart(neutrophils, vertical = T, col = rainbow(4), pch = 16, method
        = 'jitter', jitter = 0.2, ylab = 'Branching index (junctions/mm2)')
```

To add a line for the mean the function segments() can be used. segments() works in the same way as arrows but joins points up with lines instead of arrows.

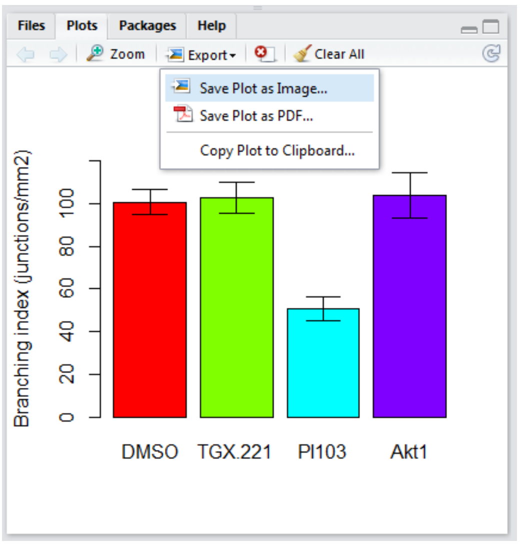
```
> segments(x0 = (1:4)-0.2, y0 = colMeans(neutrophils,na.rm = T),
        x1 = (1:4)+0.2)
```



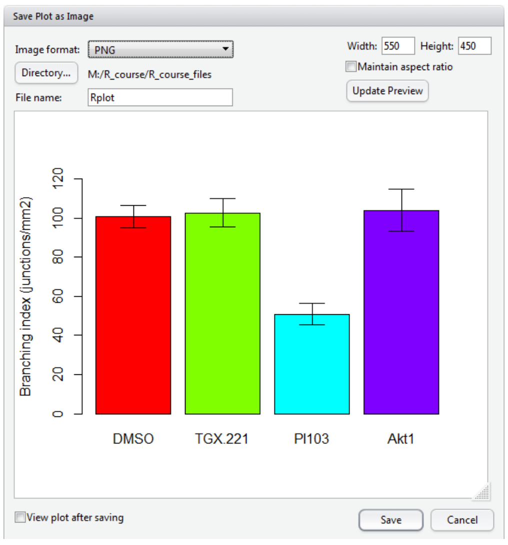
Exporting graphs

Exporting graphs from RStudio

Graphs can be exported from RStudio using the item Export from the menu bar in the graphics window. The graph can be saved as a pdf or as a range of image formats. The size of the graph can be altered.



Exporting a graph from RStudio 1



Selecting image format, adjusting size, and choosing where to save the graph to.

Exporting graphs directly from the console

Rather than creating graphs that appear in graphical windows as we have done so far, graphs can be printed directly into a pdf or image file.

```
> pdf(file = 'graph_name.pdf')
> boxplot(neutrophils, col = rainbow(4))
> dev.off()
```

The pdf file will be saved in your working directory – remember to use `getwd()` to check and `setwd()` to change your working directory. Alternatively, a full path name for the file can be used, just like when reading files in. For example:

```
> pdf(file = 'D:/projects/graphs/boxplot1.pdf')
```

The width and height of the graphics region can be adjusted with the `width` and `height` parameters. Check `?pdf` for more details.

png, jpeg, tiff and bmp files can all be created in a similar way

```
> png(file = 'graph_name.png')
> boxplot(neutrophils, col = rainbow(4))
> dev.off()
```

See `?png` for more details.

Further information

This course has only touched upon the many functions of R. For further information see the R project website http://www.r-project.org/ which has various manuals available; these are useful for understanding R in more detail.

Following on from this course we have a number of other R courses which delve into more detail on different aspects of R. All of the course material can be found at https://www.bioinformatics.babraham.ac.uk/training.html

Follow on courses which might be of interest are:

- Advanced R – looks into more aspects of the core language focussing on robustly developing larger scripts and using automation and looping.
- Plotting with R – covers the usage of the core R plotting libraries to generate complex figures
- Using ggplot – looks at an add-in graphics package for R which supplements the graphical capabilities in the core language.
- Statistics with R – looks at how you can use the statistical capabilities within R to analyse data.

Most importantly.....

If you have a specific problem don't spend hours getting frustrated. There are many additional online resources available (try googling ‘R help forum’) and you can also contact us at babraham.bioinformatics@babraham.ac.uk and we will be happy to try to help out.