DAY 2. A beginners guide to solving biological problems in R

Robert Stojnić (rs550), Laurent Gatto (lg390),
Rob Foy (raf51), John Davey (jd626) and Dávid Molnár (dm516)

Course material:
http://logic.sysbiol.cam.ac.uk/teaching/Rcourse/

Original slides by Ian Roberts and Robert Stojnić

Day 2
Schedule

1. Writing scripts

2. Writing functions

3. Data analysis examples

4. Graphics

Writing custom scripts for data analysis

1

The R scripting language

Scripting

- A script is a series of instructions that when executed sequentially automates a task
  - A script is a good solution to a repetitive problem
  - The art of good script writing is
    - understanding exactly what you want to do
    - expressing the steps as concisely as possible
    - making use of error checking
    - including descriptive comments
- R is a powerful scripting language, and embodies aspects found in most standard programming environments
  - procedural statements
  - loops
  - functions
  - conditional branching
- Scripts may be written in any standard text editor, e.g. notepad, gedit, kate
  - We will use RStudio

Colony forming experiment

- We have been asked by some collaborators to analyse some trial data to see if an experiment will work.
- We are interested in the behaviour of a gene, X, which is involved in a cell proliferation pathway.
- This pathway causes cells to proliferate in the presence of a compound, Z.
- Gene X turns the pathway off, reducing cell proliferation.

- Our collaborators want to test what happens when we knock down X in the presence of Z.
- To do this, they want to grow cell colonies in the presence of Z, with or without X, and count the number of colonies that result.

Initial trial

- Our collaborators have sent us a first batch of test data, growing colonies in different concentrations of compound Z.
- Does increasing concentration of Z have an effect on colony growth?
- We want to do the following:
  - Load the data into R
  - Plot the data to inspect it
  - Calculate an Analysis of Variance to see if growth is influenced by Z concentration
  - Calculate the mean growth for each level of Z concentration, to see the direction of change
  - (We will ignore full post hoc testing)

Initial trial exercise

- The initial trial data is in the file 2.1_colony_trial.xls. This is an Excel file and the data is not in the right format for R. Enter the data into a plain text file in a data frame format, and load it into R.
- Plot the data using a formula. Recall how we did this yesterday with linear modelling:

```
plot(y~x)
```

- Calculate an analysis of variance for the data. The R function for ANOVA is aov(), which works like lm() for linear modelling – recall this from yesterday:

```
summary(lm(y~x))
```

- There are four concentrations of Z, and each concentration has been replicated three times. What is the mean colony count for each concentration? See if you can figure out a way to calculate this with what we learned yesterday. You will need to use logical indexing and you may want to use a for loop.

Importing data

- In the Excel file, the data has this format:
- But this is not a data frame format, where columns are variables and rows are observations of those variables.
- There are three variables, Z, Replicate, and Count. We need to reshape the data with these three columns.
- We can do this in Excel, and then save the file in CSV format, or we can just type up the results in a CSV file ourselves.
- Once we have a CSV file, we can load it with read.csv:

```
colony<-read.csv("2.1_colony_trial.csv”)
```

Replicate	1	2	3
No Z	150	180	223
Low Z	87	40	53
Medium Z	5	1	9
High Z	0	0	0

Z	Replicate	Count
None	1	150
None	2	180
None	3	223
Low	1	87
Low	2	40
Low	3	53
Medium	1	5
Medium	2	1
Medium	3	9
High	1	0
High	2	0
High	3	0

Plotting

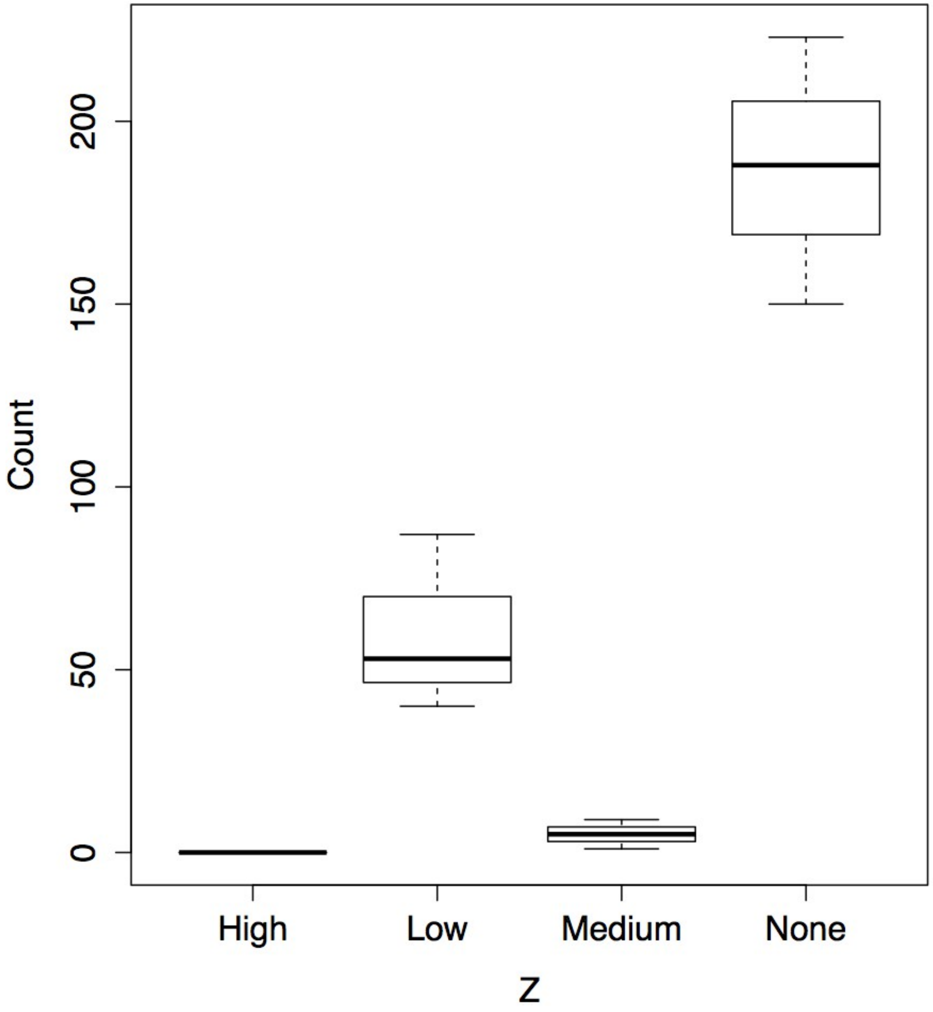
We want to plot the colony growth in response to changing Z concentration.

Z is the explanatory variable, and Count is the response variable.

We don't want to plot replicates separately here, but get R to summarise each Z concentration over all replicates.

We can call plot using the same formula syntax we learnt yesterday:

```
plot(colony$Count~colony$Z)
```



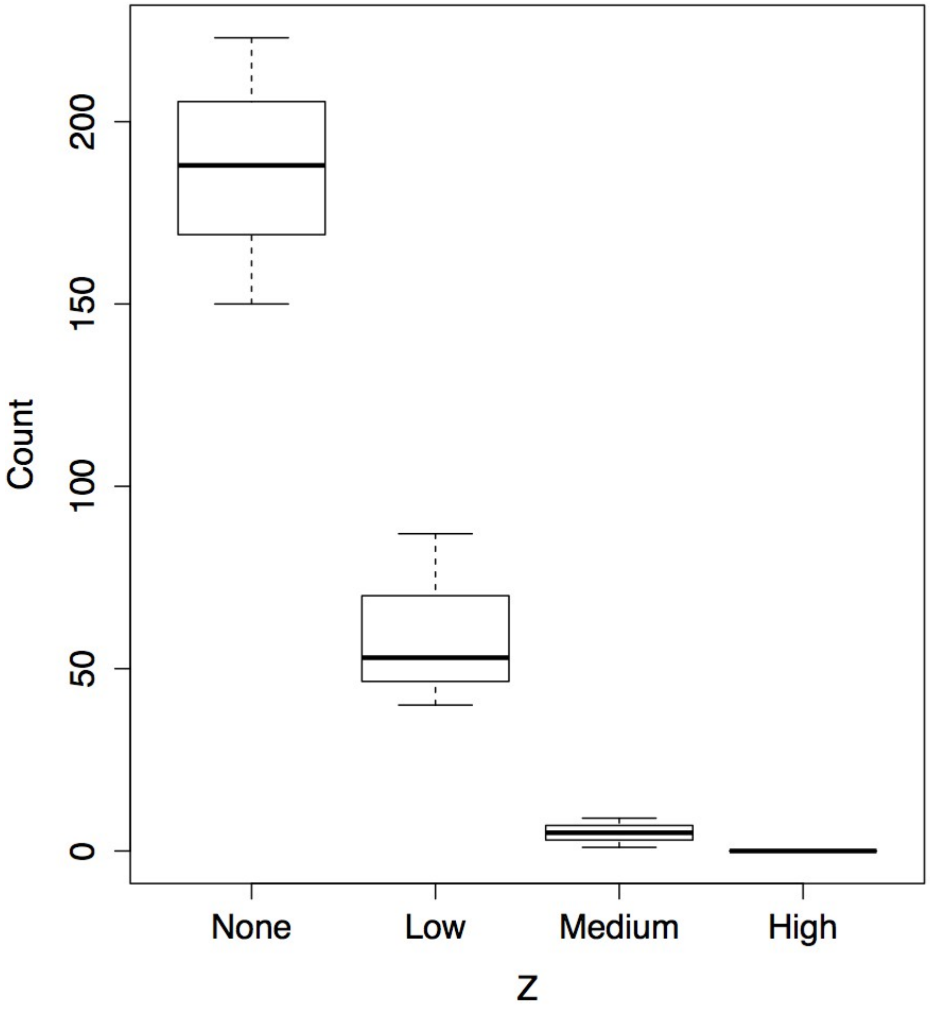
Plotting

We can improve on this. Firstly, we want to order the Z categories. Z is a factor, so we need to supply new levels to this factor in the colony data frame:

```
colony$Z<-factor(colony$Z,
   levels=c("None","Low","M
   edium","High"))
```

Second, we can tell the plot command which data frame to use, rather than using the dollar operator:

```
plot(Count~Z, data=colony)
```



Analysis of Variance

We can use the same formula syntax to calculate an analysis of variance:

```
colony.aov<-aov(Count~Z, colony)
summary(colony.aov)
            Df Sum Sq Mean Sq F value   Pr(>F)    
Z            3  68154   22718   46.89 2.02e-05 ***
Residuals    8   3876     484                     
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

This tells us what we can already see from the plot, that there is a highly significant relationship between Z concentration and colony growth.

We would like to investigate this relationship. For example, we might want to calculate the mean colony count for each concentration of Z.

Calculating group means

We can calculate a mean for a particular group like this:

```
> mean(colony[colony$Z=="None",]$Count)
[1] 187
> mean(colony[colony$Z=="Low",]$Count)
[1] 60
> mean(colony[colony$Z=="Medium",]$Count)
[1] 5
> mean(colony[colony$Z=="High",]$Count)
[1] 0
```

We could generalise this with a for loop:

```
for (z in levels(colony$Z)) {
     print(mean(colony[colony$Z==z,]$Count))
}
[1] 187
[1] 60
[1] 5
[1] 0
```

But there is a better way.

The tapply function
a brief digression

- The apply family of functions allow us to group data by variable and calculate something for each group.
- Assume we have the following data for heights of 5 males and females:

```
    data <- data.frame(gender=c("Male", "Male", "Female",
                  "Female", "Female"), height=c(6, 6.1, 5.8, 6, 5.95))
gender height
1    Male    6.00
2    Male    6.10
3 Female    5.80
4 Female    6.00
5 Female    5.95
```

- How can we get mean height of males and females separately?

  tapply() lets us do exactly this:

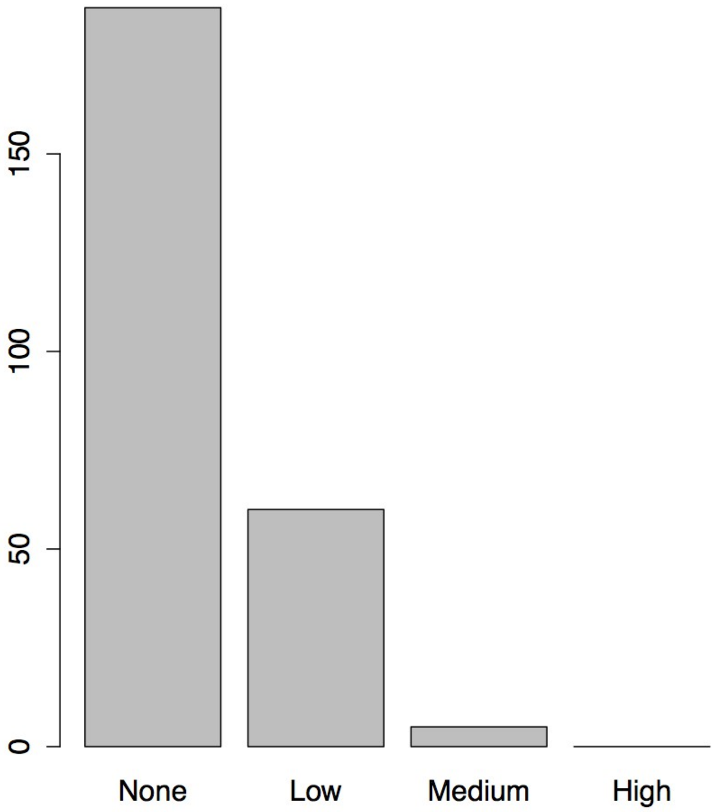
- **tapply(data$height, data$gender, mean)**

  data groups function

Using tapply on colony

- We can use tapply to calculate group means on colony like this:

```
> colony.means<-tapply( colony$Count, colony$Z, mean )
> colony.means
  None     Low Medium    High
   187      60      5       0
> barplot(colony.means)
```



A complete script

We now have a complete script to analyse this data:

```
# Load data, order Z and plot
colony<-read.csv("2.1_colony_trial.csv")
colony$Z<-factor(colony$Z,c("None","Low","Medium","High"))
plot(Count~Z,colony)

# Analysis of Variance
colony.aov<-aov(Count~Z,colony)
print(summary(colony.aov))

# Calculate group means
colony.means<-tapply(colony$Count,colony$Z,mean)
print(colony.means)
barplot(colony.means)
```

We need to print the results we want to see on screen, otherwise they will not be output.

Make sure you can source your commands (or the file 2.1_colony_1.R) from Rstudio and generate the results and plot.

Knocking down gene X: revising the script

As the trial worked, our collaborators have gone ahead with an experiment to knock down gene X in the same concentrations of Z.

On our request, they have delivered the data in a data frame format in a CSV file: 2.1_colony_Run1Counts.csv.

They want us to see if knocking down X affects colony growth.

Because we saved our analysis in a script, we can rerun the same script to analyse the data, just by changing the name of the file we are loading.

Run your script on this new data file and confirm that you can calculate an ANOVA and group means for this new data set.

Knocking down gene X: revising the script

Our current script only analyses Z, not X. We need to modify it to include X and see how both X and Z influence colony growth.

1. We need to include X and the interaction between Z and X in our formulae for plotting and for ANOVA. Look up the 'Modelling formulae' slide from Day 1 to see how to do this.

2. What does **plot** do with a formula including both X and Z? Try using **boxplot** instead. What difference does it make if you change the order of X and Z?

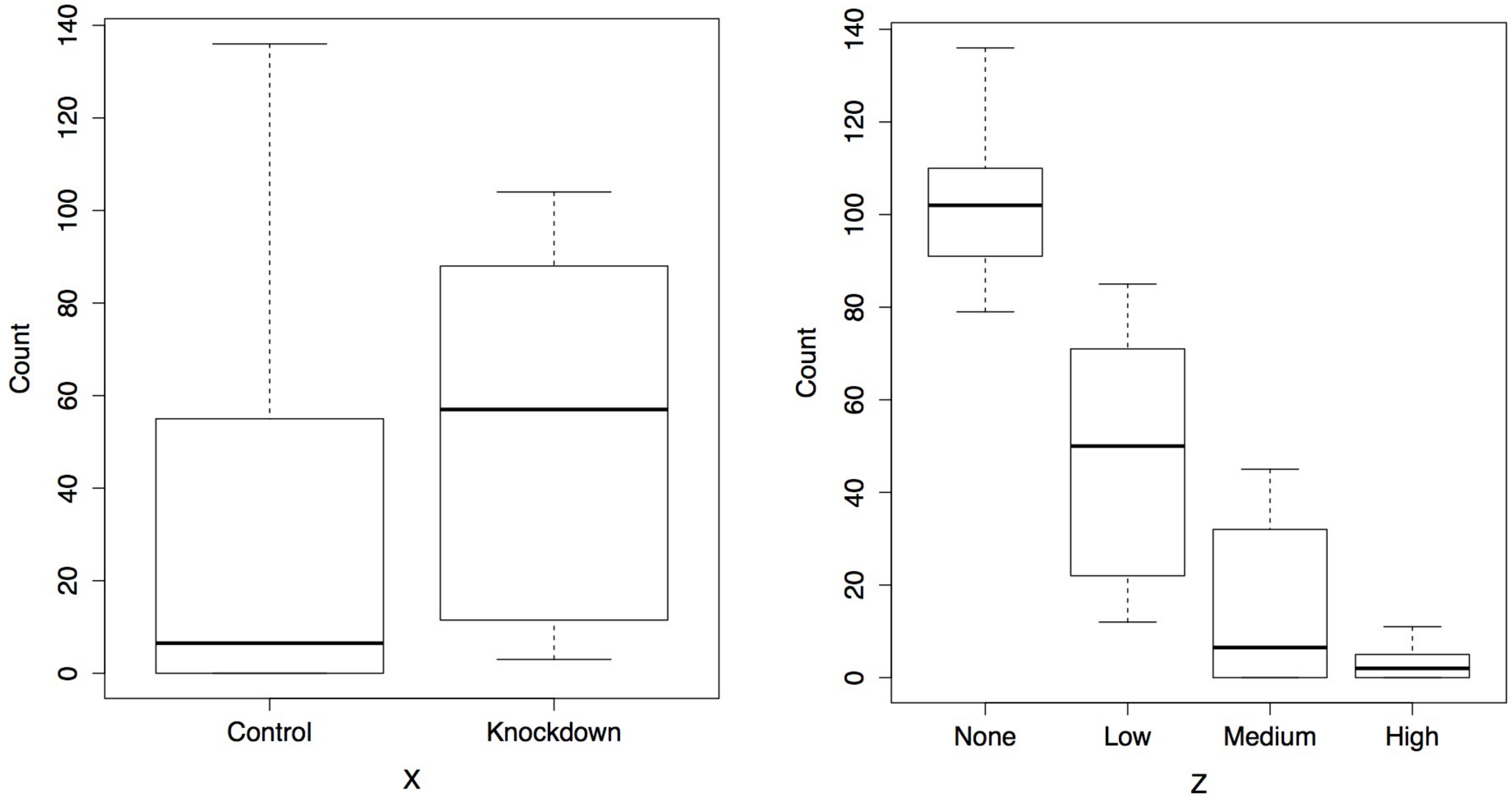
3. We need to include both X and Z in our call to **tapply**. Modify the call to **tapply** by changing the second argument, which should be a list containing the data for both X and Z.

4. Plot the group means you calculated with **tapply** using **barplot**. Plot bars for different conditions *beside* each other, not on top of each other. Check the help page for an option to do this.

Plotting interactions

Including interactions in formulae is straightforward, but **plot** doesn't show us the interaction, only the main effects:

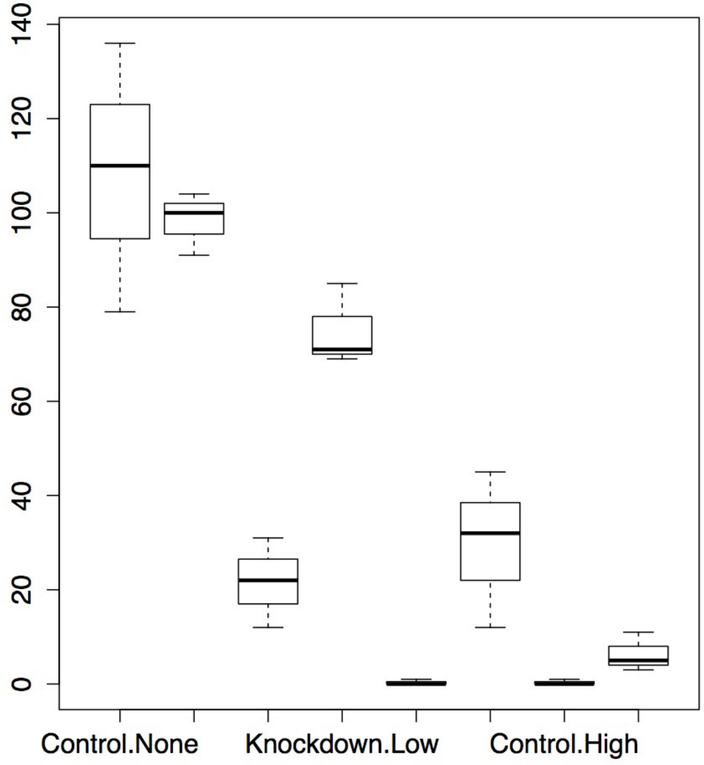
```
> plot(Count~X*Z,colony)
```



Plotting interactions

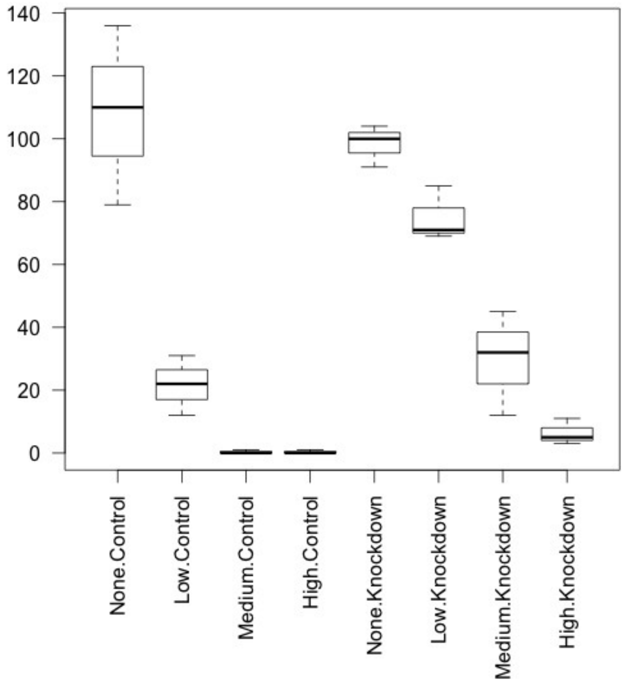
To get a sense of what's happening with the interactions, use **boxplot**:

```
> boxplot(Count~X*Z,colony)
```



To make the labels visible, we'll use some graphics commands to increase the size of the lower margin and make the x-axis labels vertical (full details on this this afternoon):

```
> par(oma=c(6,2,2,2))
> boxplot(Count~X*Z,colony,las=2)
```



It looks like knocking down X increases colony growth, except when Z is completely absent.

Analysis of variance with interactions

Including interactions in the analysis of variance is straightforward:

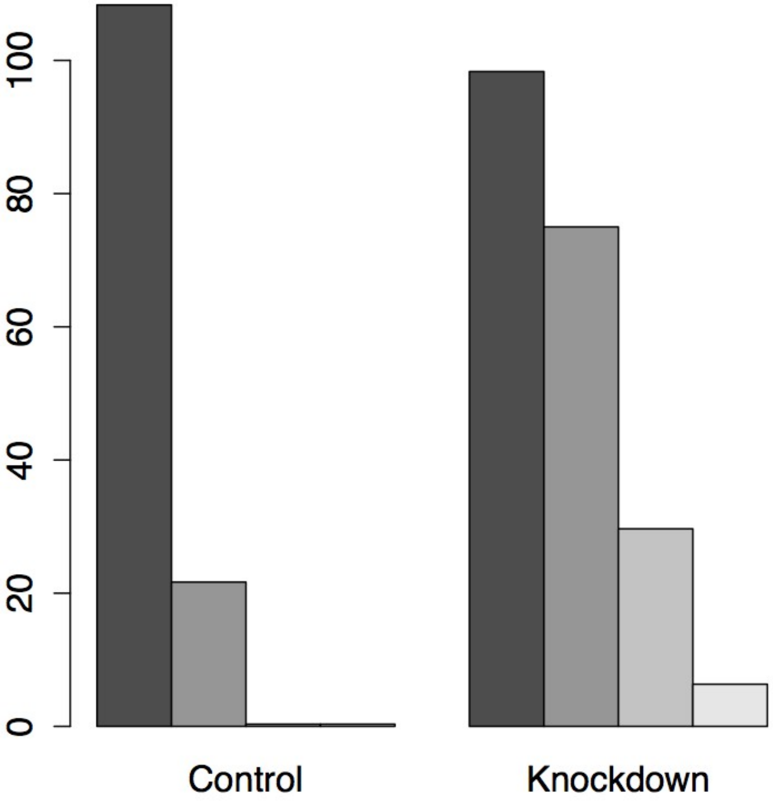
```
> colony.aov<-aov(Count~X*Z,colony)
> print(summary(colony.aov))
            Df Sum Sq Mean Sq F value    Pr(>F)
X            1   2321    2321  14.072  0.00174 **
Z            3  36150   12050  73.067 1.48e-09 ***
X:Z          3   3441    1147   6.954  0.00329 **
Residuals   16   2639     165
---
Signif. Codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Not only do X and Z have a significant effect on colony growth individually, but there is also a significant interaction between them.

tapply with multiple variables

Including Z in the call to tapply is a little fiddly, but easy when you know how. Use the **beside** option in the call to **barplot**. (What happens if you put X first in the list?)

```
> colony.means<-tapply(colony$Count,list(colony$Z,colony$X),mean)
> print(colony.means)
           Control Knockdown
None   108.3333333 98.333333
Low     21.6666667 75.000000
Medium   0.3333333 29.666667
High     0.3333333  6.333333
> barplot(colony.means,beside=TRUE)
```



Complete script: first revision

Our script now looks like this (see 2.1_colony_2.R):

```
# Load data, order Z and plot
colony<-read.csv("2.1_colony_Run1Counts.csv")
colony$Z<-factor(colony$Z,c("None","Low","Medium","High"))

par(oma=c(6,2,2,2))
boxplot(Count~Z*X,colony,las=2)

# Analysis of Variance
colony.aov<-aov(Count~X*Z,colony)
print(summary(colony.aov))

# Calculate group means
colony.means<-tapply(colony$Count,list(colony$Z,colony$X),mean)
print(colony.means)
barplot(colony.means,beside=TRUE)
```

Incorporating multiple runs: second revision

As it looks like there is an interaction between Z and X, our collaborators have repeated the experiment twice more, to increase the sample size. They have delivered two more data files, one for each extra run, in the files 2.1_colony_Run2Counts.csv and 2.1_colony_Run3Counts.csv.

1. Modify your script to load in the two new files. You may wish to use the looping code you wrote yesterday. Combine all three data sets into one **colony** data frame.

2. Now we have an additional variable, the **Run** each observation came from. Create a **Run** vector to record this, and add it to your colony data frame with **cbind**. You will need to use the **rep** function with its **each** argument – look it up to see what it does.

3. Modify your boxplot and analysis of variance to include **Run**. How does this data set look? Would you trust an analysis of it?

Loading multiple files

This should look familiar from yesterday:

```
# Load data
colony.files<-dir(pattern="Counts.csv")
colony<-data.frame()
for (cf in colony.files) {
  colony<-rbind(colony,read.csv(cf))
}
```

We *could* just call **read.csv** three times, but this would be very brittle. What if we were given fifty more files? What if the filenames changed? This code will handle these cases – we would only have to change the pattern in the first line, not every call to **read.csv**.

Creating the Run variable

If we inspect the data files, we can see there are 24 observations in each file. So we can hardcode a Run variable like this:

```
Run<-rep(1:3,each=24)
colony<-cbind(Run,colony)
```

But this is brittle in the same way as calling **read.csv** three times is brittle. It would be better to get R to calculate **Run** from what it knows about the data.

We can count the observations in **colony** with **nrow**, and the number of runs (number of files) by getting the **length** of **colony.files**.

```
nruns<-length(colony.files)
Run<-rep(1:nruns,each=nrow(colony)/nruns)
colony<-cbind(Run,colony)
```

Analysis of variance with Run variable

Adding the Run variable to our analysis of variance is easy:

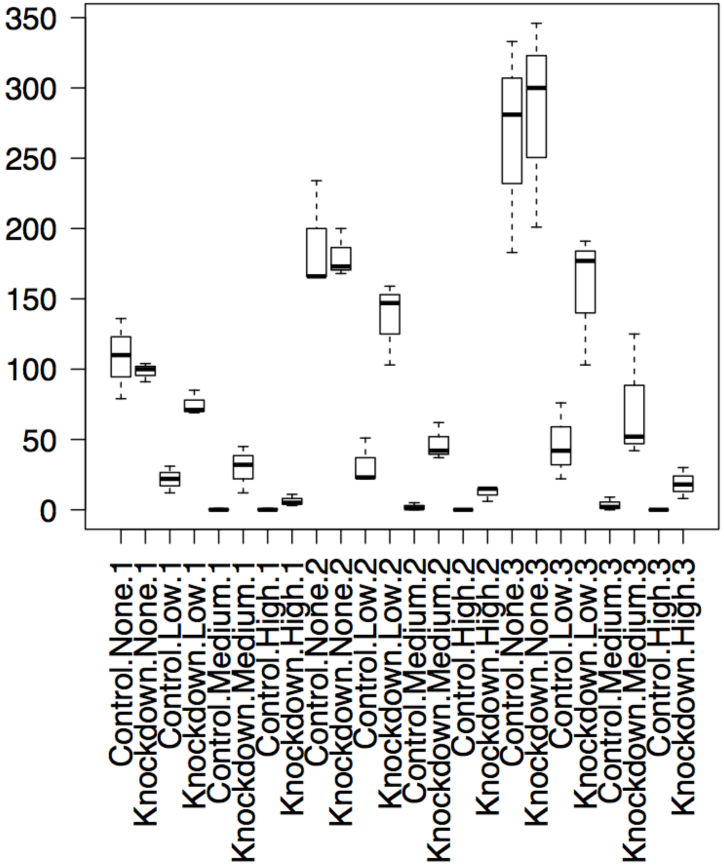
```
> colony.aov<-aov(Count~X*Z*Run,colony)
> print(summary(colony.aov))
            Df Sum Sq Mean Sq F value    Pr(>F)    
X            1  24939   24939  32.454 4.71e-07 ***
Z            3 355524  118508 154.221  < 2e-16 ***
Run          1  48197   48197  62.721 1.05e-10 ***
X:Z          3  21967    7322   9.529 3.51e-05 ***
X:Run        1   3485    3485   4.535   0.0376 *  
Z:Run        3  49513   16504  21.478 2.19e-09 ***
X:Z:Run      3    805     268   0.349   0.7897    
Residuals   56  43032     768                     
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

But the result is troubling – why should Run be correlated with colony growth? Let's have a look at the plot.

Plotting the Run variable

It is also easy to add Run to our boxplot:

```
boxplot(Count~X*Z*Run,colony,las=2)
```



Unfortunately, this shows that something has gone wrong during the experiments. The later runs have much higher colony growth than the first.

We probably can't rely on this data to tell us something about X and Z. At the very least, we will have to control for Run during the analysis.

Scripting summary

You can find the final version of the script in 2.1_colony_3.R.

We have seen the value of writing a reusable script, and how we can modify scripts as new data comes in.

We have also seen how easy it is to incorporate new variables into R, and to repeat complicated analyses with single commands.

Finally, we have learnt a little about generalising our code to cope with variations in current and future input. Now we'll go on to learn how to define our own functions in R, to help us to generalise even further.

User functions

2

Introducing ...
User functions

- All R commands are function calls.

- Functions take some input, perform calculations on that input, and return some output.

- EG **sqrt** is a function that takes a value, calculates the square root of the value, and returns the square root.
- **aov** takes a formula referring to some data, calculates the analysis of variance for that data, and returns the model it calculated.

- We can define our own functions. User functions extend the capabilities of R by adapting or creating new tasks that are tailored to your specific requirements.

- User functions are objects, just like vectors and data frames. This has a few useful implications.

Defining a new function

- A function has a name, arguments, procedural steps, and a return value.

```
sqXplusX <- function(x){
    x^2 + x
}
```

- `sqXplusX` is the function name
- `x` is the single argument to this function and it exists only within the function
- everything between brackets { } are procedural steps
- the **last** calculated value is the function return value. We can call **return** explicitly:

```
sqXplusX <- function(x){
    return(x^2 + x)
}
```

- After defining the function, we can use it:

```
> sqXplusX(10)
[1] 110
```

Named and default arguments

- We can generalise our function by adding a second argument.

```
powXplusX <- function(x, power=2){
    x^power + x
}
```

- The power argument has a default value of 2; if we don't supply a power when we call the function, x will be squared.
- Arguments without default value are required, those with default values are optional.

```
> powXplusX(10)
[1] 110
> powXplusX(10, 3)
[1] 1010
```

arguments matched based on **position**

```
> powXplusX(x=10, power=3)
[1] 1010
```

arguments matched based on **name**

Calculation with user functions

User functions can be used wherever a built in function can be used:

```
a <- matrix(1:100, ncol=10, byrow=T)  # make some dummy data
sqXplusX(a)
```

functions are R objects, just like a vector or a data frame, and exist in our workspace:

```
> sqXplusX
function(x) x^2+x
> ls()
```

Just like we can create anonymous vectors, we can create anonymous functions:

```
> 1:10
 [1]  1  2  3  4  5  6  7  8  9 10
> function(x) x^2+x
function(x) x^2+x
```

But why would we want to do this?

Assigned or anonymous ...
User functions

Anonymous functions are really useful for `apply()` functions. We'll use the **colony** data frame again. Let's say we wanted to know the percentage of plates in each condition with no growth at all. Recall how we were using **tapply**:

```
tapply(colony$Count,list(colony$Z,colony$X,colony$Run),mean)
```

We can write a function to calculate our percentage directly into the call to **tapply**:

```
tapply(colony$Count,list(colony$Z,colony$X,colony$Run),
        function(x) length(x[x==0])/length(x)*100)
```

```
, , 1

        Control Knockdown
None    0.00000         0
Low     0.00000         0
Medium 66.66667         0
High   66.66667         0
...
```

Variable scope

Objects created in functions are not available to the global environment unless returned. They are limited to the *scope* of the function.

```
> addone<-function(x) {x<-x+1; x}
> x<-1
> addone(x)
[1] 2
> x
[1] 1
```

The **x** in the global environment has nothing to do with the **x** declared in the function, and is unchanged by the call to the function. To update the global **x**, we would need to assign the return value of the function:

```
> x<-addone(x)
```

A function can only return one object, but that object can be a list, so if you have many objects to return, package them up into a list first.

Script / function tips
User functions

- If your script repeats the same command with different values more than twice, you should consider writing a function to generalise that command.

- Writing functions makes your code more easily understandable because they encapsulate a procedure into a well-defined boundary with consistent input/output

- Functions should only do one thing. If a function is doing multiple tasks, try to split it up into multiple functions. This rule of thumb means functions tend to be short, not more than around one or two screens of code.

- Look at other functions to get ideas for how to write your own …
    - Display function code by entering the function’s name without brackets.

Checking input and reporting errors

- A function should fail gracefully if it does not receive valid input when it is called. We can use **if** statements to check for appropriate input.
- R has two useful commands to tell the user something is wrong. **warning** prints a message and continues to run the function. **stop** ends the function after printing the message.
- For example, we might rewrite our **powXplusX** function to check that the power argument is a whole number:

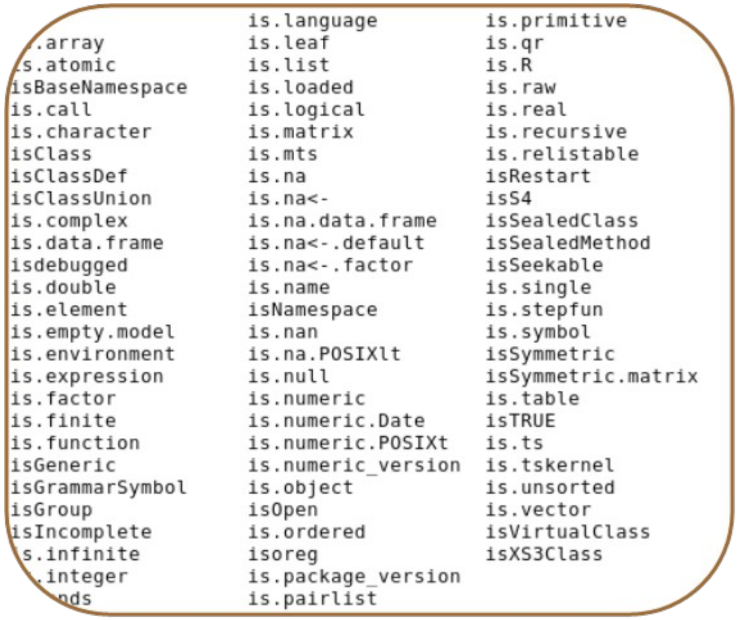
```
powXplusX<-function(x,power=2) {
   if (power %% 1 != 0) stop("Power should be a whole number")
   x^power+x
}
```

```
> powXplusX(10,3)
[1] 1010
> powXplusX(10,3.5)
Error in powXplusX(10, 3.5) : Power should be a whole number
```

Checking input and reporting errors

R has a very useful set of functions called the **is** family, which check the type of input values. For example:

```
sqXplusX <- function(x){
    if (is.numeric(x)) {
        x^2 + x
    } else {
        stop("Input should be numeric")
    }
}

> sqXplusX(10)
[1] 110
> sqXplusX("ten")
Error in sqXplusX("ten") : Input should be numeric
```



```
.array
s.atomic
isBaseNamespace
is.call
is.character
isClass
isClassDef
isClassUnion
is.complex
is.data.frame
isdebugged
is.double
is.element
is.empty.model
is.environment
is.expression
is.factor
is.finite
is.function
isGeneric
isGrammarSymbol
isGroup
isIncomplete
s.infinite
.integer
nds
is.language
is.leaf
is.list
is.loaded
is.logical
is.matrix
is.mts
is.na
is.na<-
is.na.data.frame
is.na<-.default
is.na<-.factor
is.name
isNamespace
is.nan
is.na.POSIXlt
is.null
is.numeric
is.numeric.Date
is.numeric.POSIXt
is.numeric_version
is.object
isOpen
is.ordered
isoreg
is.package_version
is.pairlist
is.primitive
is.qr
is.R
is.raw
is.real
is.recursive
is.relistable
isRestart
isS4
isSealedClass
isSealedMethod
isSeekable
is.single
is.stepfun
is.symbol
isSymmetric
isSymmetric.matrix
is.table
isTRUE
is.ts
is.tskernel
is.unsorted
is.vector
isVirtualClass
isXS3Class
```

The `is.family`

Here's another, more concise way to do the same thing:

```
sqXplusX <- function(x){
    if (!is.numeric(x)) stop ("Input should be numeric")
    x^2 + x
}
```

Checking input and reporting errors

Here's another, more concise way to do the same thing:

```
sqXplusX <- function(x){
    if (!is.numeric(x)) stop ("Input should be numeric")
    x^2 + x
}
```

This is not only shorter, but it also gets all the error checking out of the way before the main processing steps.

You may also find the **%in%** command useful, which checks to see if the elements of one vector are present in another:

```
> levels(colony$Z)
[1] "None"   "Low"    "Medium" "High"
> "Low" %in% colony$Z
[1] TRUE
> "Zero" %in% colony$Z
[1] FALSE
> c("None","Low") %in% colony$Z
[1] TRUE TRUE
```

Temperature conversion exercise
User functions

Centigrade to Fahrenheit conversion is given by F = 9/5 * C +32.

Write a function that converts between temperatures.

The function should take two named arguments:

temperature `(t)` *is numeric*

units `(unit)` *is character*

Both arguments should have appropriate default values.

The function should report an appropriate error if inappropriate values are given.

```
if( !is.numeric(t) ) { .... }
if( !(unit %in% c("c","f")) ){...}
```

The function should print out the temperature in Fahrenheit if given in Centigrade, and vice versa.

Building the solution

- It is difficult to write large chunks of code. Instead, start with something that works and build upon it.
- E.g. to solve the temperature conversion exercise:
  - write a skeleton function definition (eg just a name and brackets)
  - add appropriate argument names and defaults
  - write code to convert Centigrade into Fahrenheit and check it works
  - write code to convert Fahrenheit to Centigrade and check it works
  - add error checking code, including the checks from the previous slide, and any others you can think of
  - write a set of test calls to confirm that your function handles correct *and incorrect* input
- If you get stuck, call us to help you!

Temperature conversion exercise script

```r
convTemp<-function(t=0,unit="c"){ # convTemp is defined as a new user function requiring two arguments, t and unit, the default values are 0 and "c", respectively.

    if ( !is.numeric(t) ) stop("Non numeric temperature entered")

    if ( !(unit %in% c("c","f"))){
        stop("Unrecognized temperature unit. Enter (c)entigrade or (f)ahrenheit.")
    }

    converted<-0

    # Conversion for centigrade
    if ( unit=="c" ) {
        converted <- 9/5 * t + 32
    }

    # Conversion for Fahrenheit
    if(unit=="f"){
        converted <- 5/9 * (t-32)
    }

    converted
}
```

```
> convTemp(t=-273,unit="c")
[1] -459.4
```

Example code:
2.2_convtemp.R

Temperature conversion application

An American colleague is moving to Cambridge and wants to know what the average minimum and maximum temperatures are for each season, in Fahrenheit.

The file **2.2_cambridge_temps.tsv** contains minimum and maximum temperatures per month.

1. Load this file into a data frame in R with **read.delim**.
2. Add a season variable to the data frame (define the seasons however you like).
3. Use **tapply**, **mean** and **convTemp** to calculate average minimum and maximum temperatures for each season.

Temperature conversion application

Here's one way of doing this:

```
camtemps<-read.delim("2.2_cambridge_temps.tsv")

camtemps<-cbind(camtemps, Season=c("Winter", "Winter", "Spring", "Spring",
"Spring", "Summer", "Summer", "Summer", "Autumn", "Autumn", "Autumn",
"Winter"))

tapply(camtemps$MinimumTemp, camtemps$Season, function(x) convTemp(mean(x)))
tapply(camtemps$MaximumTemp ,camtemps$Season, function(x) convTemp(mean(x)))
```

But because R functions are so flexible, we can do the conversion at any point in the calculation. These alternatives will also work:

```
> tapply(camtemps$MinimumTemp, camtemps$Season,
         function(x) convTemp(mean(x)))


> convTemp(tapply(camtemps$MinimumTemp, camtemps$Season, mean))

> tapply(convTemp(camtemps$MinimumTemp), camtemps$Season, mean)
```

Advanced data processing

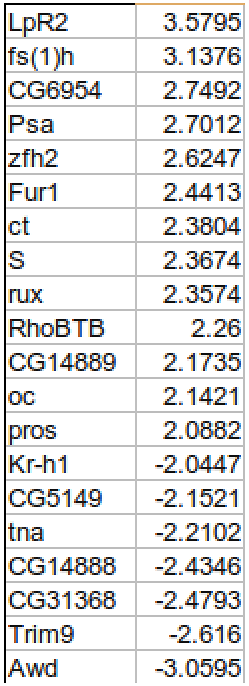
3

Combining data from multiple sources
Gene clustering example

- R has powerful functions to combine heterogeneous data into a single data set
- Gene clustering example data:
  - five sets of differentially expressed genes from various experimental conditions
  - file with names of experimentally verified genes
- Gene clustering exercise:
  1. combine this dataset into a single table and cluster to see which conditions are similar
  2. repeat the clustering but only on a subset of experimentally verified genes

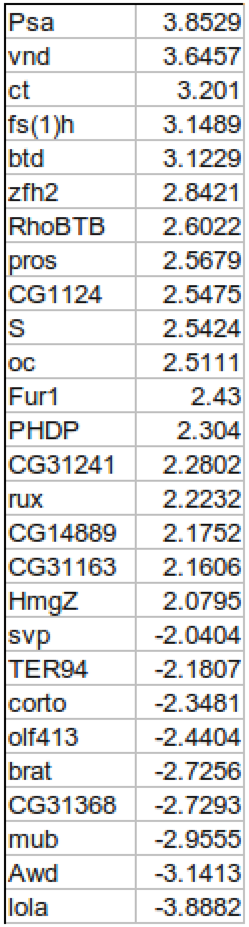
Combining gene tables

- input files have two columns: gene names and fold change
- we want to combine all five tables into a single table, with 0 for missing values



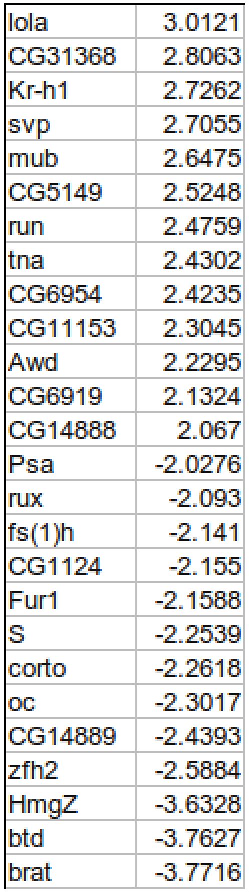
LpR2	3.5795
fs(1)h	3.1376
CG6954	2.7492
Psa	2.7012
zfh2	2.6247
Fur1	2.4413
ct	2.3804
S	2.3674
rux	2.3574
RhoBTB	2.26
CG14889	2.1735
oc	2.1421
pros	2.0882
Kr-h1	-2.0447
CG5149	-2.1521
tna	-2.2102
CG14888	-2.4346
CG31368	-2.4793
Trim9	-2.616
Awd	-3.0595

+



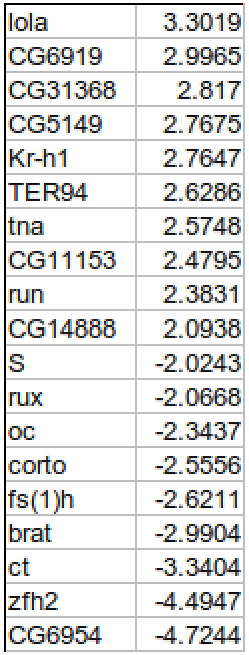
Psa	3.8529
vnd	3.6457
ct	3.201
fs(1)h	3.1489
btd	3.1229
zfh2	2.8421
RhoBTB	2.6022
pros	2.5679
CG1124	2.5475
S	2.5424
oc	2.5111
Fur1	2.43
PHDP	2.304
CG31241	2.2802
rux	2.2232
CG14889	2.1752
CG31163	2.1606
HmgZ	2.0795
svp	-2.0404
TER94	-2.1807
corto	-2.3481
olf413	-2.4404
brat	-2.7256
CG31368	-2.7293
mub	-2.9555
Awd	-3.1413
lola	-3.8882

+



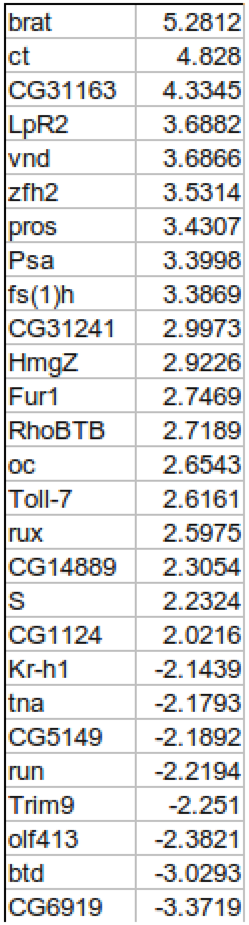
lola	3.0121
CG31368	2.8063
Kr-h1	2.7262
svp	2.7055
mub	2.6475
CG5149	2.5248
run	2.4759
tna	2.4302
CG6954	2.4235
CG11153	2.3045
Awd	2.2295
CG6919	2.1324
CG14888	2.067
Psa	-2.0276
rux	-2.093
fs(1)h	-2.141
CG1124	-2.155
Fur1	-2.1588
S	-2.2539
corto	-2.2618
oc	-2.3017
CG14889	-2.4393
zfh2	-2.5884
HmgZ	-3.6328
btd	-3.7627
brat	-3.7716

+



lola	3.3019
CG6919	2.9965
CG31368	2.817
CG5149	2.7675
Kr-h1	2.7647
TER94	2.6286
tna	2.5748
CG11153	2.4795
run	2.3831
CG14888	2.0938
S	-2.0243
rux	-2.0668
oc	-2.3437
corto	-2.5556
fs(1)h	-2.6211
brat	-2.9904
ct	-3.3404
zfh2	-4.4947
CG6954	-4.7244

+



brat	5.2812
ct	4.828
CG31163	4.3345
LpR2	3.6882
vnd	3.6866
zfh2	3.5314
pros	3.4307
Psa	3.3998
fs(1)h	3.3869
CG31241	2.9973
HmgZ	2.9226
Fur1	2.7469
RhoBTB	2.7189
oc	2.6543
Toll-7	2.6161
rux	2.5975
CG14889	2.3054
S	2.2324
CG1124	2.0216
Kr-h1	-2.1439
tna	-2.1793
CG5149	-2.1892
run	-2.2194
Trim9	-2.251
olf413	-2.3821
btd	-3.0293
CG6919	-3.3719

Gene clustering

Script walkthrough 1

- To make the big table we first need to find out all the genes present in at least one of the files
- Make sure not to use factors in read.delim()

```
# start with en empty collection of genes
genes <- c()
for( fileNum in 1:5 ){
   # load in files 2.3_DiffGenes1.tsv ...
   t <- read.delim(paste("2.3_DiffGenes", fileNum, ".tsv", sep=""),
                  as.is=TRUE, header=FALSE)
   # label the input columns to help code readability
   names(t) <- c("gene", "expression")
   genes <- union(genes, t$gene)
}

# for tidiness order our genes by name
genes <- sort(genes)

genes # show all genes
```

when loading in character data use **as.is=T** to prevent it being converted to factors!

union() is a set operation, combines two vectors by eliminating duplicates. There are also intersect() and setdiff()

Example code:
2.3_geneClustering.R

Gene clustering

Script walkthrough 2

- Using the complete list of genes, we can create the big table and fill in the values:

```
# make the destination table [rows = unique genes, cols = file numbers]
values <- matrix(0, nrow=length(genes), ncol=5)
rownames(values) <- genes # name the rows with the complete gene names

for(fileNum in 1:5){
  # read in the file again
     t <- read.delim(paste("2.3_DiffGenes", fileNum, ".tsv", sep=""),
             as.is=T, header=F)
     names(t) <- c("gene", "expression")

     # match the names of the genes to the rows in our big table
     index <- match(t$gene, rownames(values))
     # copy the expression levels
     values[index,fileNum] <- t$expression
}
```

match() returns the index of first argument in the second, i.e. index of input file genes in the big table

we use index to pick the rows in such way that they match the gene order in the input file

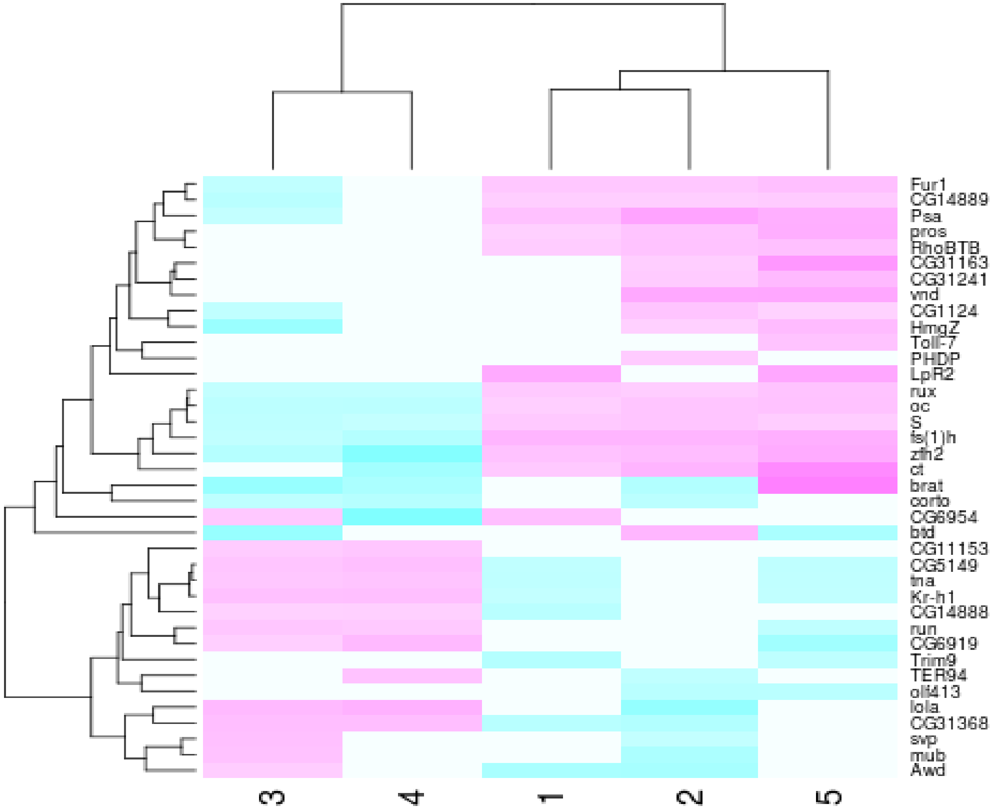
Gene clustering

Script walkthrough 3

- Now we can do hierarchical clustering:

```
heatmap(values, scale="none", col = cm.colors(256))
```

Values from the matrix are colour-coded. Rows and columns are re-arranged according to similarity



Gene clustering

Script walkthrough 4

- In a second part of our analysis, we want to produce the same heatmap but only based on a list of experimentally verified genes

- The problem is data is not formatted in the most convenient way:

genes	citation
oc,run,RhoBTB,CG5149,CG11153,S,Fur1	Segal et al, Development 2001
tna,Kr-h1,rux	Krejci et al, Development 2002

Gene clustering

Script walkthrough 5

- We load in this table, and only extract the gene names, then we use them to select a subset of values matrix

```
# load in the tab-delimited file with genes and citations
t.exp <- read.delim("2.3_ExperimentalGenes.tsv", as.is=T)
# split all gene names by "," and then flatten it out into a single vector
experim.genes <- unlist( strsplit(t.exp$genes, ",") )
```

unlist() flattens out a nested list into a single vector

strsplit() splits a vector of strings by a custom split character (","). The result is a list of split values for each element of the input vector

```
# redo the heatmap by using just the genes in the experimentally verified set
is.experimental <- rownames(values) %in% experim.genes
heatmap(values[ is.experimental, ], scale="none", col = cm.colors(256))
```

Gene clustering review

- We load in the five tables twice - first to collect gene names, then to load expression values
- Based on expression table (values) we construct a clustered heatmap first on the whole set of genes, then on a selected subset
- Go through the code, try it out it and understand it
- Try answering the following questions:
  - what is rownames(values) ?
  - why is rownames(values)[index] and t$gene giving the same output?
  - what is the difference between rownames(values) %in% experim.genes and experim.genes %in% rownames(values)

Example code:
2.3_geneClustering.R

Graphics

4

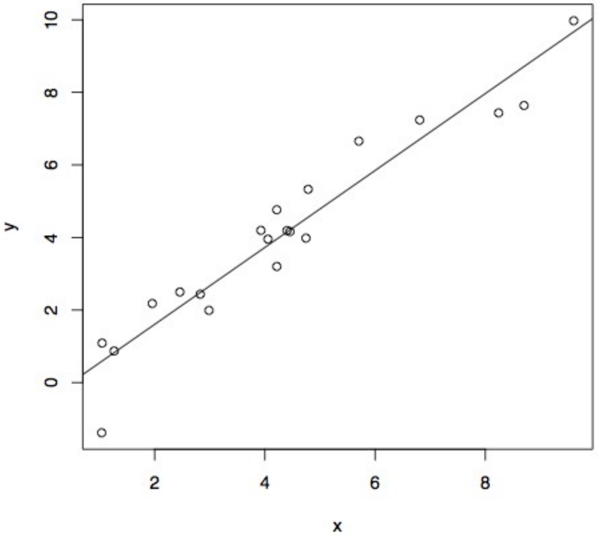
Starting out with R graphics
Graphics

We have already used several functions from R's built in **graphics** package. Now we will explain the fundamental principles of this package, and how R handles graphics devices (printing to the screen or to different file types).

R's traditional (or 'base') graphics functions are still widely used, but they are old and often difficult to use. Many alternative graphics packages are available. We will briefly introduce a commonly used alternative, **ggplot**.

We will start by creating a random plot:

```
> x <- runif(20, min = 1, max = 10)
> y <- x + rnorm(20, mean = 0, sd = 1)
> plot(x, y)
> abline(lm(y ~ x))
```



Starting out with R graphics
Graphics

If you run these commands in Rstudio, the plot will appear in the Plot window. If you run them from the standard R Gui, or from the console, the plot appears in a new window. These windows are graphics *devices*.

A device is something R can pipe graphics output to. PDF, JPEG and PNG files can also be graphics devices.

Some devices (such as PDFs) can have multiple *pages* of output, but others (like screens) can only have one.

plot is an example of a *high-level graphics function*; it creates a new plot on a new page of the current device.

abline is an example of a *low-level graphics function*; it adds something to an existing plot on the current page of the current device. If we tried to call **abline** before calling **plot**, it would fail, because there would be nothing to draw upon.

Graphics devices

R maintains a list of open graphics devices on a stack, and will write to whatever is the *current* device:

```
> dev.cur()
RStudioGD
        2
```

Each open device has a number associated with it; device number 1 is the *null* device, which is used when there is no other device open.

There are several functions available to create devices of various types. For example, **jpeg** creates a new JPEG file device on the top of the device stack:

```
> jpeg("xyplot_example.jpg")
> dev.cur()
jpeg
   4
> plot(x,y)
> dev.off()
RStudioGD
        2
```

dev.off() closes the current graphics device and switches to the next device on the stack, in this case the Rstudio window we already had open.

High level graphics functions

The box below lists the high level plotting functions provided by the **graphics** package. All of these functions will draw a new plot on the next available plotting region.

The first argument to a high level graphics function is always the data to plot. This can often come in different forms; for example, we have seen **plot** take a vector, a data frame or a formula as its data. Sometimes more than one object can be passed in as data.

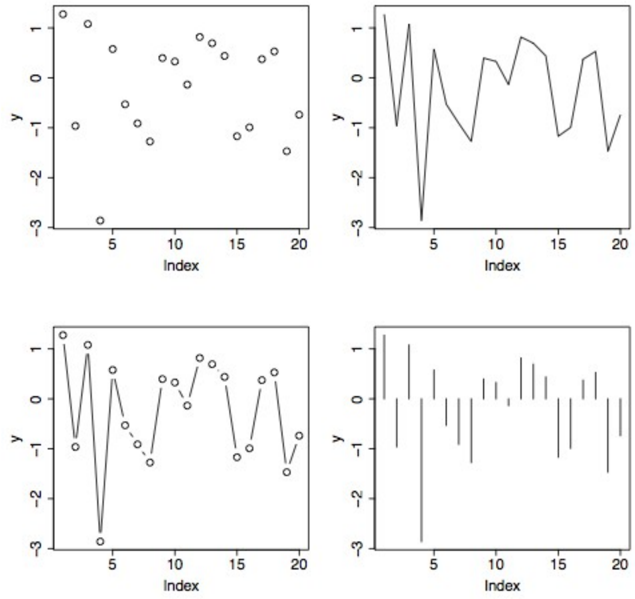
```
> plot(litters)
> plot(1:10)
> plot(y~x)
> plot(x, y)
```

```
assocplot
barplot
boxplot
bxp
cdplot
contour
coplot
curve
dotchart
filled.contour
fourfoldplot
hist
image
matplot
mosaicplot
pairs
persp
pie
plot
spineplot
stars
stem
stripchart
sunflowerplot
xspline
```

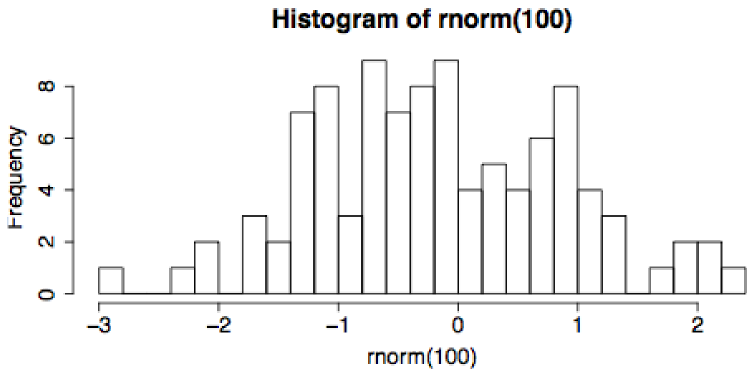
High level graphics functions

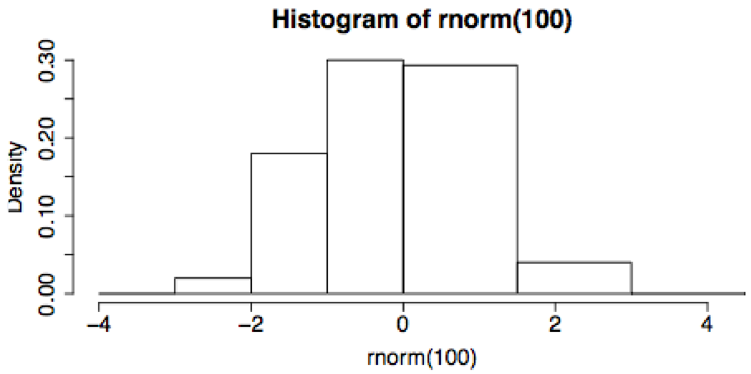
High level functions usually have many arguments for modifying the plots, most of which have very terse and opaque options:

```
> y <- rnorm(20)
> plot(y, type = "p")
> plot(y, type = "l")
> plot(y, type = "b")
> plot(y, type = "h")
```



```
> hist(rnorm(100), breaks = 20)
> hist(rnorm(100), breaks = c(-4, -3, -2, -1, 0, 1.5, 3, 4.5))
```



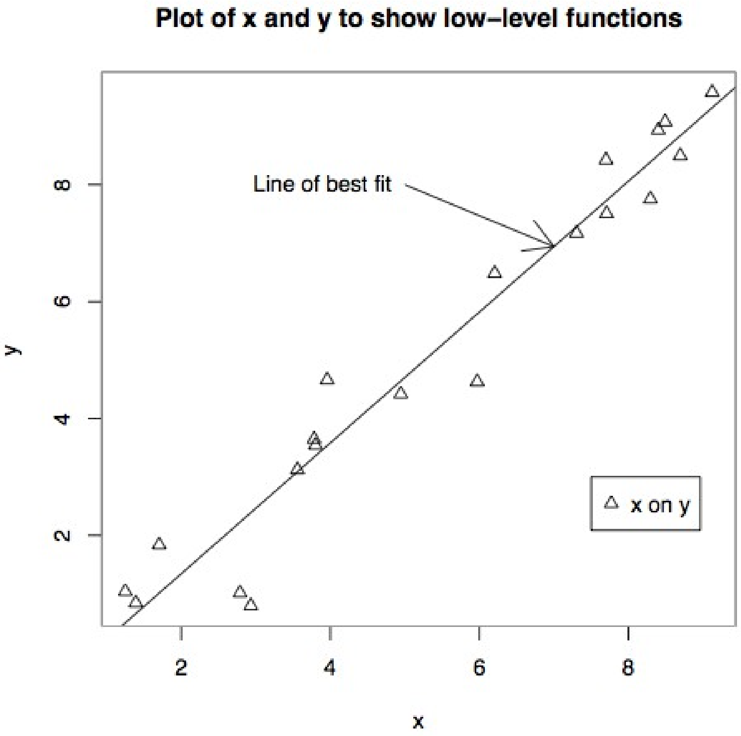


Low level graphics functions

The box on the right lists some of the low level functions provided by the **graphics** package. Here's how we might use some of these to create a complex plot:

abline
arrows
axis
box
grid
legend
lines
mtext
points
polygon
rect
rug
segments
symbols
text
title

```
> x <- runif(20, min = 1, max = 10)
> y <- x + rnorm(20, mean = 0, sd = 1)

> plot(x, y, pch = 2, main = "Plot of x
and y to show low-level functions")

> lmfit <- lm(y ~ x)

> abline(lmfit)

> box(col = "grey")

> arrows(5, 8, 7, predict(lmfit,
data.frame(x = 7)))
> text(5, 8, "Line of best fit", pos = 2)

> legend(7.5, 3, c("x on y"), pch = 2)
```



Low level graphics functions

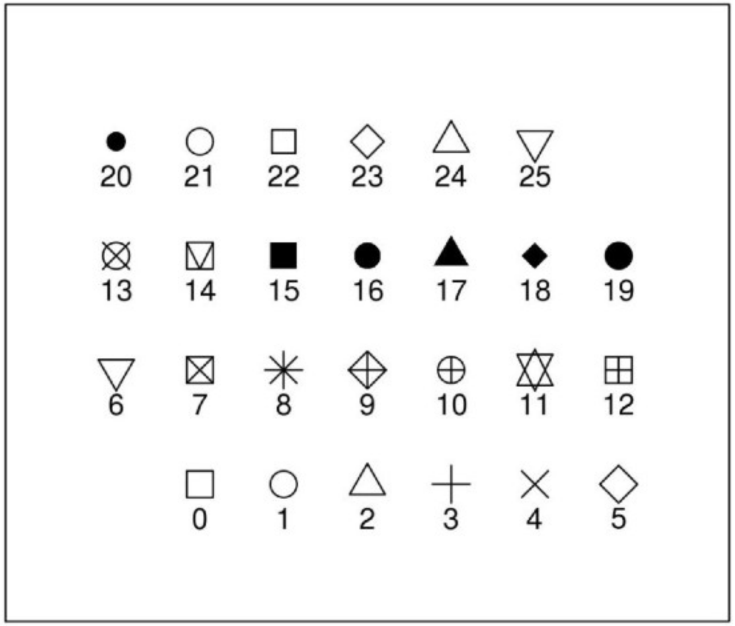
R's traditional graphics uses a *painter* model – once something is drawn on a device, it cannot be erased. If you make a mistake with a low level function, you will have to start drawing the plot again from scratch.

```
> plot(x, y, pch = 2, main = "Plot of x and y to show low-level
functions")
```

The **pch** argument stands for plotting character. The box on the bottom right shows the built-in plotting characters and their numbers.

```
> arrows(5, 8, 7,
         predict(lmfit, data.frame(x = 7)))
> text(5, 8, "Line of best fit", pos = 2)
> legend(7.5, 3, c("x on y"), pch = 2)
```

These commands take pairs of x and y coordinates. They use the scales of the data. The arrow begins at x=5, y=8, and ends at x=7, y=the predicted y value for x=7 based on the linear regression.



Graphics state and **par**

We can use high and low level functions to change graphics parameters like colour and shape, but these are not listed as arguments on the help pages for these functions.

Also, R set most of these parameters itself – we didn't specify fonts, or axis labels, for example.

Each graphics device has a *state*; a set of parameters with current values, which R will refer to whenever it draws anything on that device.

We can view and change the values of these parameters using the **par** function. The **par** help page describes all the parameters. Try this to view the current values of the graphics state:

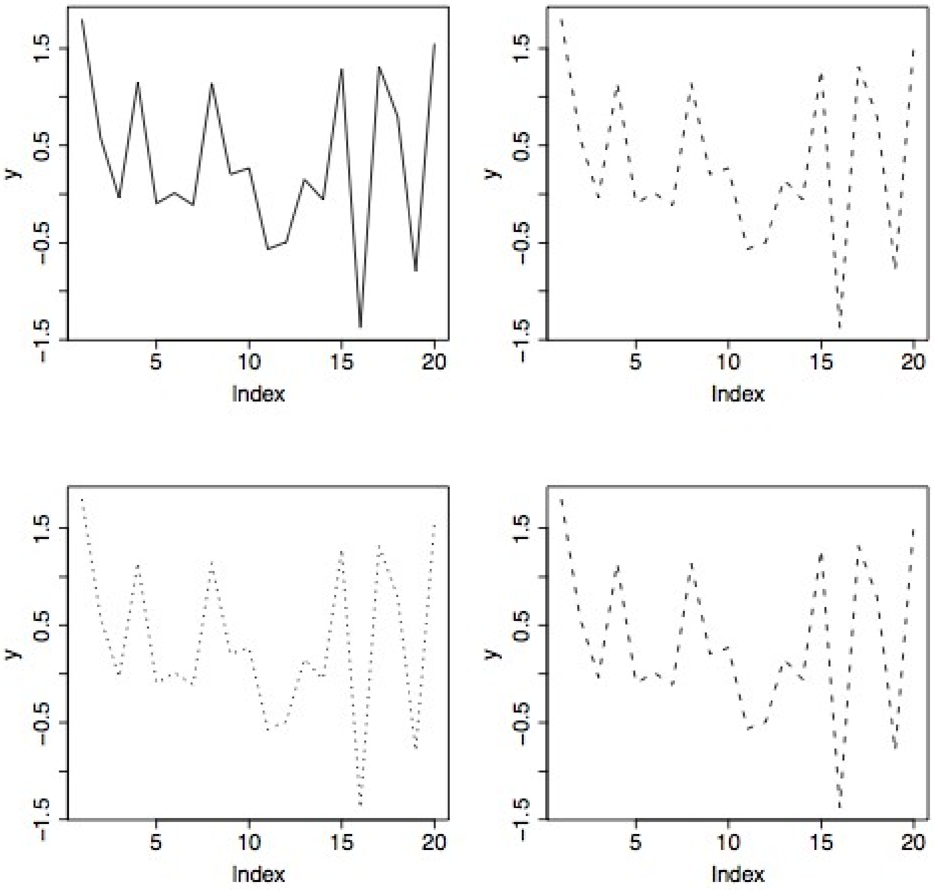
```
> par()
```

When we set a graphics parameter in a call to a high or low level graphics function, we only change its value for that call alone. But we can also change a graphics parameter with par, which changes it persistently, until the graphics device is closed.

Graphics state and **par**

Here's an example of setting graphics parameters persistently using **par**, and temporarily in a call to a high level function:

```
> y <- rnorm(20)
> par("lty")
[1] "solid"
> plot(y, type="l")

> par(lty="dashed")
> par("lty")
[1] "dashed"
> plot(y, type="l")

> plot(y, type="l", lty="dotted")

> plot(y, type="l")
> par("lty")
[1] "dashed"
```



Page layout

A graphics *page* has *outer margins* and an *inner region*.

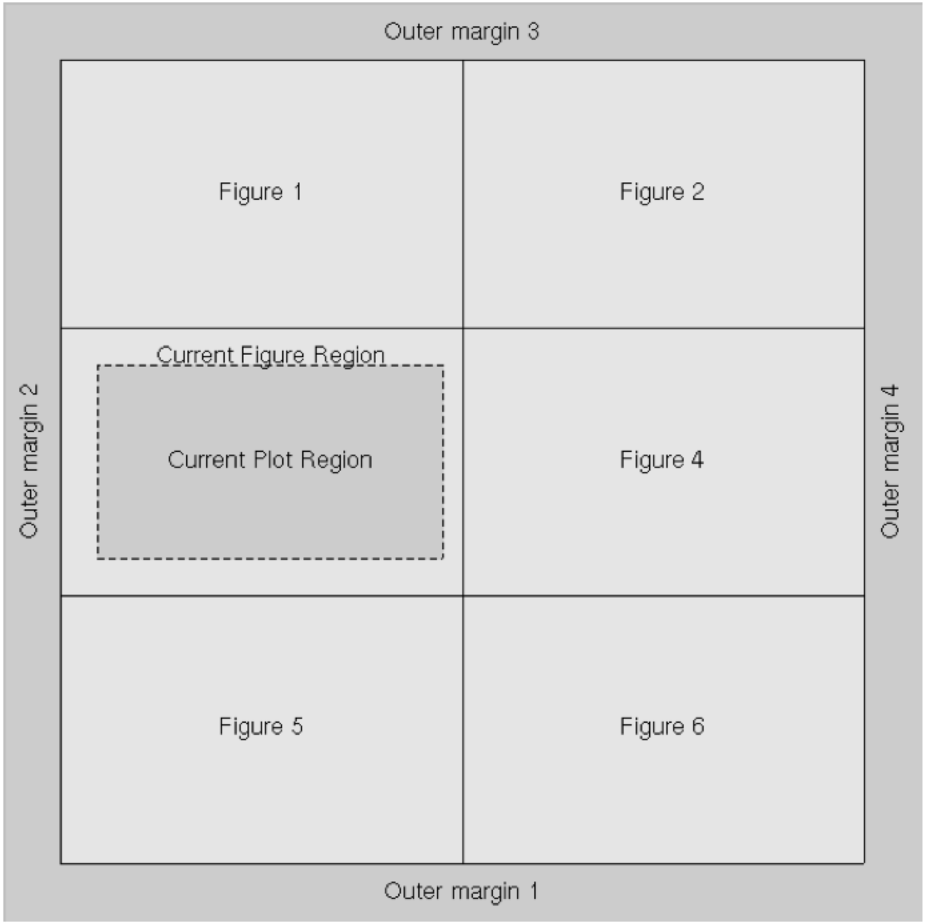
The inner region is split into one or more *figure regions*, which typically contains a *plot region*. The plot region contains data points, but not titles, axes and so on.

To create multiple figures on one page, we can use the **mfrow** parameter:

```
> par(mfrow=c(3,2))
```

This will create 3 rows and 2 columns on the current page, like the figure on the right. We can also change the size of the outer margins and figure margins:

```
> par(oma=c(6,2,2,2))
> par(mar=c(5,4,4,2))
```



From Paul Murrell, R Graphics, 2005

Bar plot exercise

We already used the **barplot** function this morning, and tried out its **beside** argument. Let's explore this function in more detail. Generate a random matrix and plot it with **barplot**:

```
> y<-matrix(sort(rnorm(50)), c(5,10))
> barplot(y)
```

- add the **beside** argument.

- open the **barplot** help page and find the argument to plot bars horizontally, not vertically.

- find the argument that defines the colours of the bars. What happens if you use the **palette()** or **colours()** functions as input to this argument?

- call **colours()** on its own. Find the five shades of red in the output, and use only these shades in your bar plot.

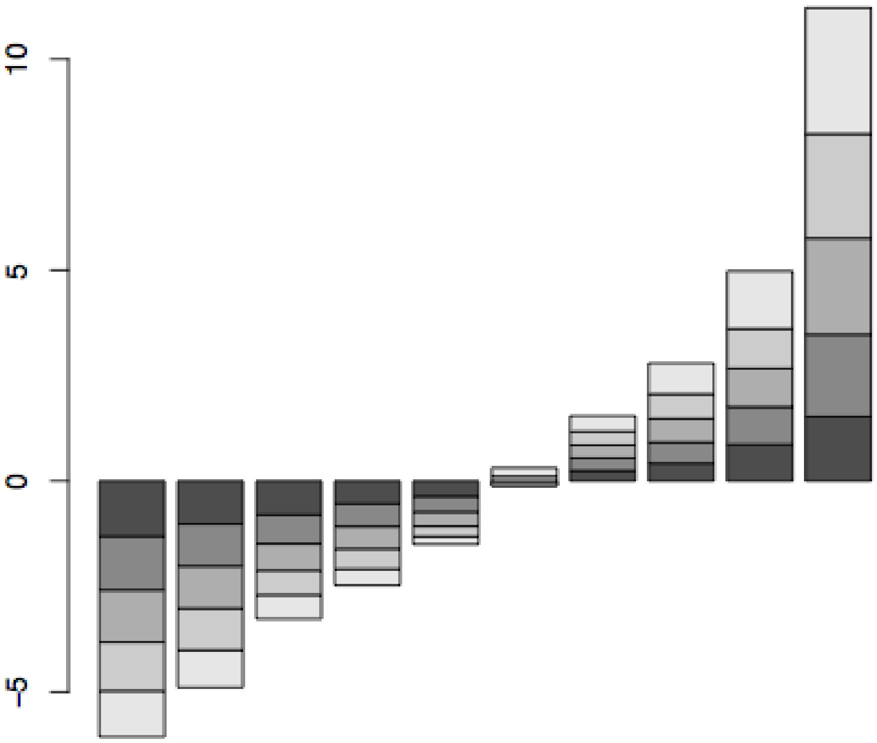
- the x-axis doesn't cover the range of values. Find the argument to change this and extend the x-axis as needed.

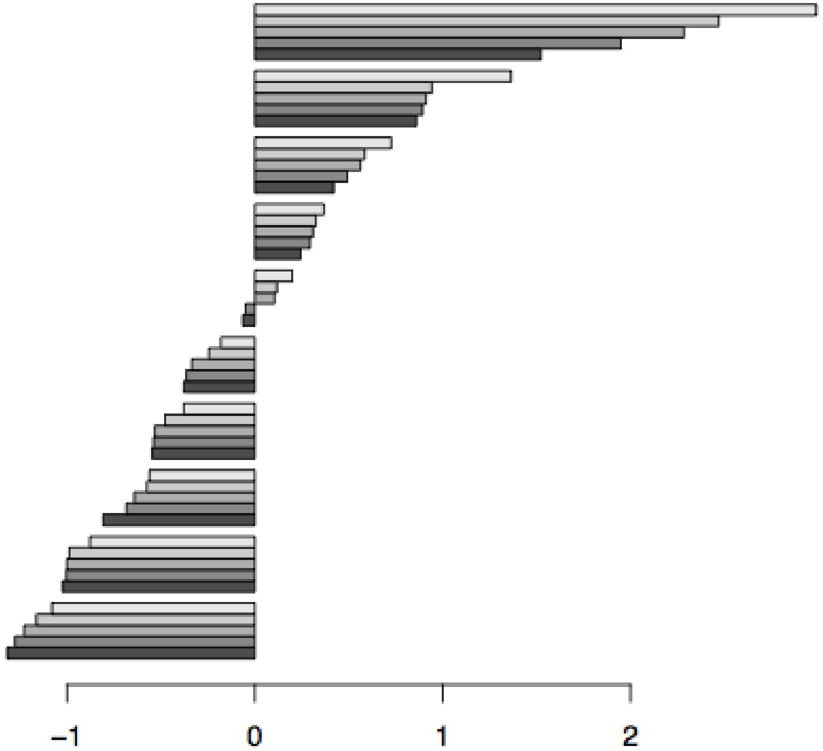
- add an appropriate title and x-axis label to the plot.

- save your plot to a PDF 15 inches high by 15 inches wide.

Bar plot exercise: positions

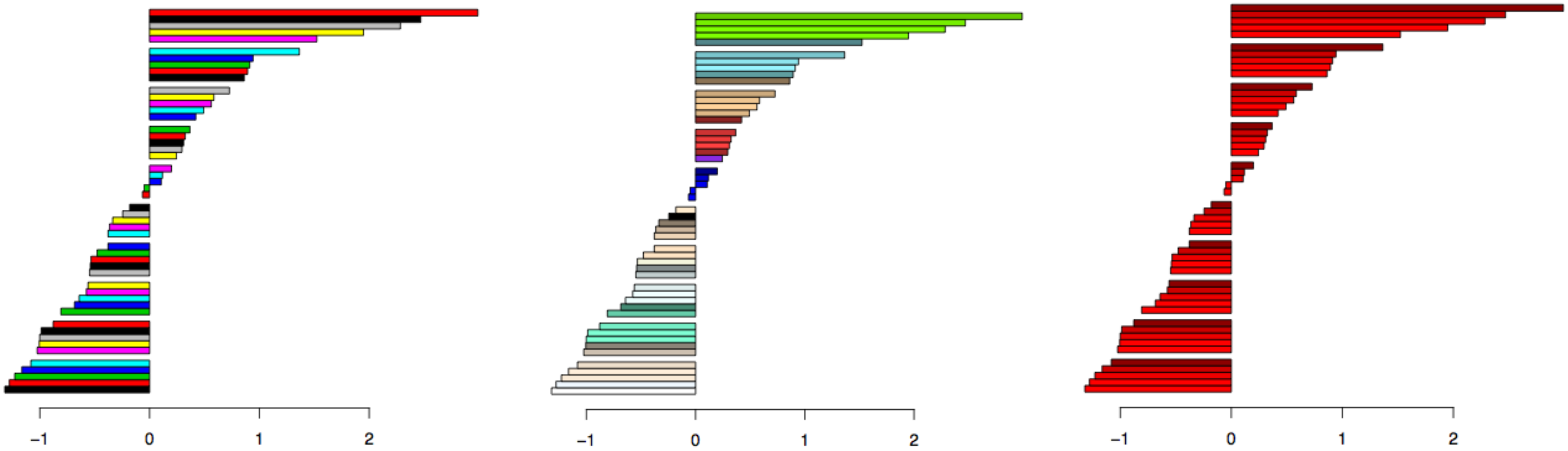
```
> y<-matrix(sort(rnorm(50)), c(5,10))
> barplot(y)
> barplot(y, horiz=TRUE, beside=TRUE)
```





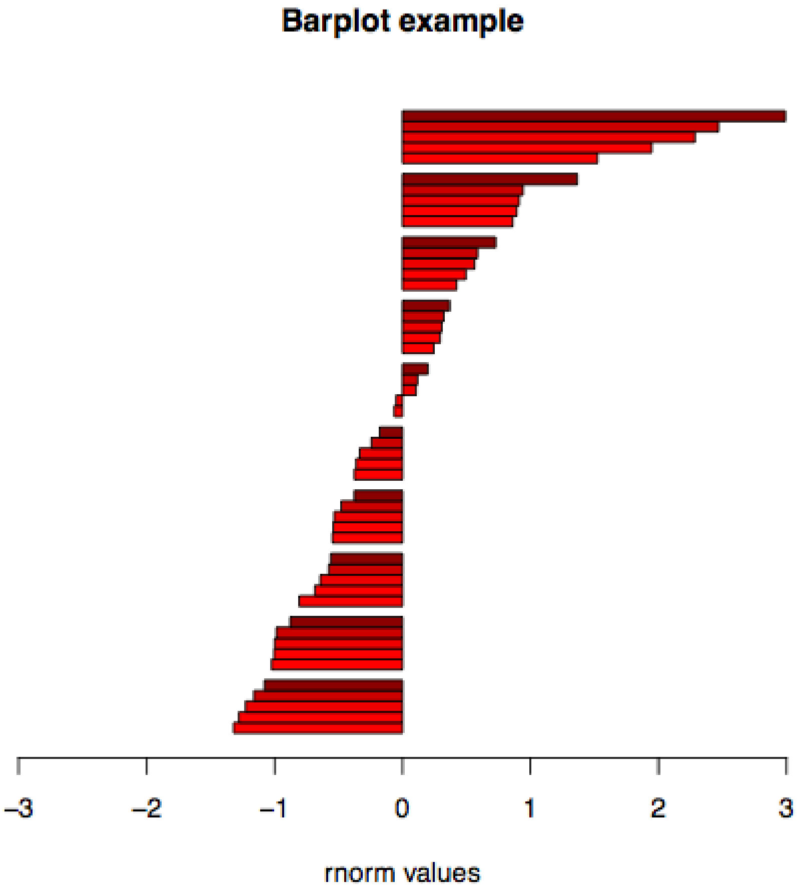
Bar plot exercise: colours

```
> barplot(y, horiz = TRUE, beside = TRUE, col = palette())
> barplot(y, horiz = TRUE, beside = TRUE, col = colours())
> barplot(y, horiz = TRUE, beside = TRUE, col = c("red", "red1", "red2",
  "red3", "red4"))

> palette()
[1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
"gray"
```



Bar plot exercise: axis and labels

```
> barplot(y, horiz = TRUE, beside = TRUE,
          col = c("red", "red1", "red2", "red3", "red4"),
          xlim = c(-3, 3),
          main = "Barplot example", xlab = "rnorm values")
```



Bar plot exercise: writing to a PDF

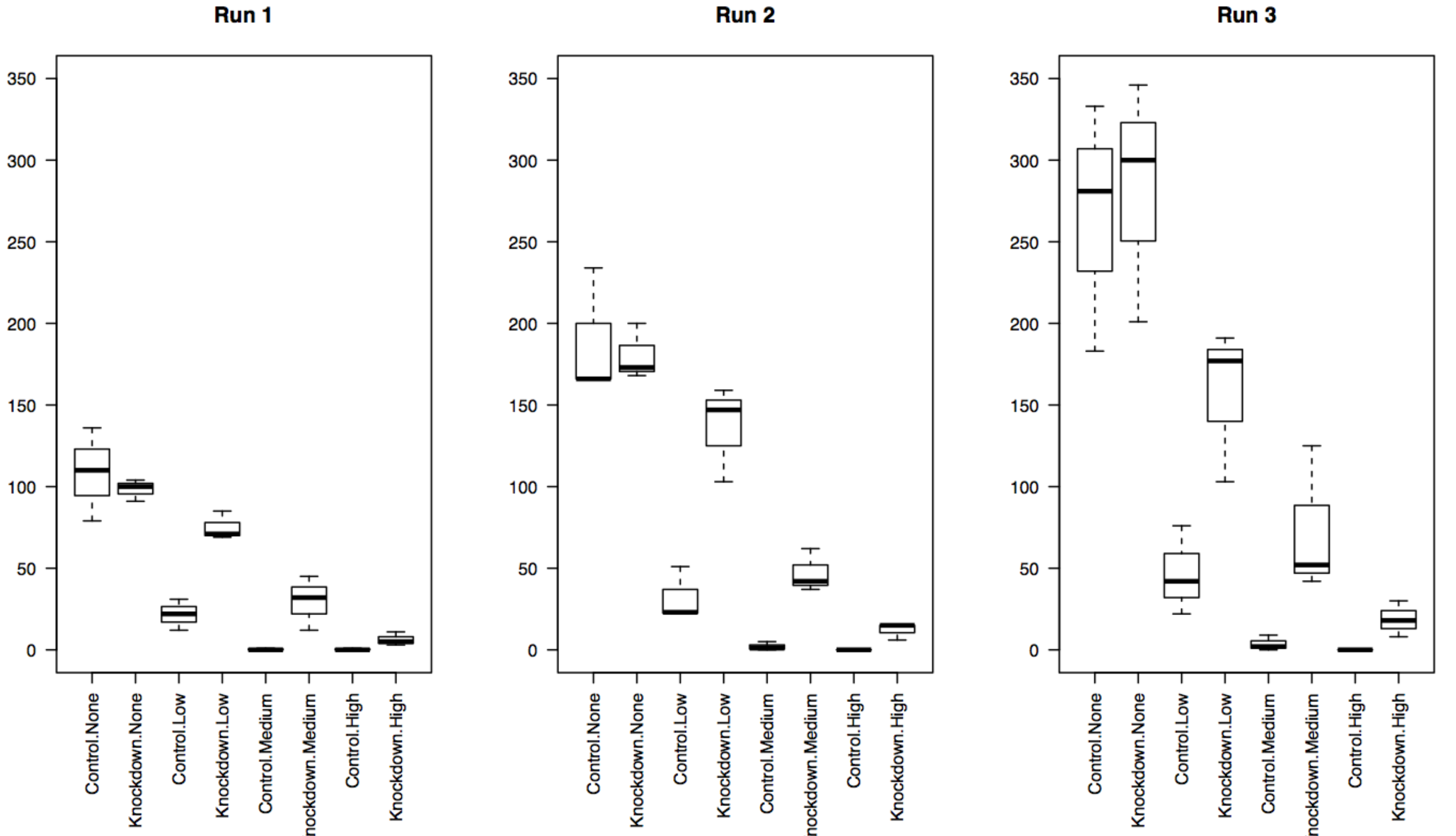
There is a **pdf()** function to create a PDF file as an R graphics device. This function (in common with most graphics device functions) has width and height options. For **pdf**, the default units of these options are inches, but other devices differ (for example, **png** and **jpeg** default to pixels).

Don't forget to call **dev.off** after finishing your plotting, or your PDF file may not be written correctly and may not open.

```
> pdf("barplot.exercise.pdf", width=15,height=15)
> barplot(y, horiz = TRUE, beside = TRUE,
           col = c("red", "red1", "red2", "red3", "red4"),
           xlim = c(-3, 3),
           main = "Barplot example", xlab = "rnorm values")
> dev.off()
```

Colony plot exercise

Can you replicate this plot of the **colony** data from this morning?



ggplot

The traditional graphics functions are still widely used, but show their age. Remembering not only the odd names for parameters (**pch**) but also the numbers for a particular symbol is very inconvenient, and things like drawing legends manually is extra work we would rather not have to do.

ggplot is one package that solves a lot of these problems. Here, we will just demonstrate the kind of output you can produce with ggplot, and compare it to the traditional graphics.

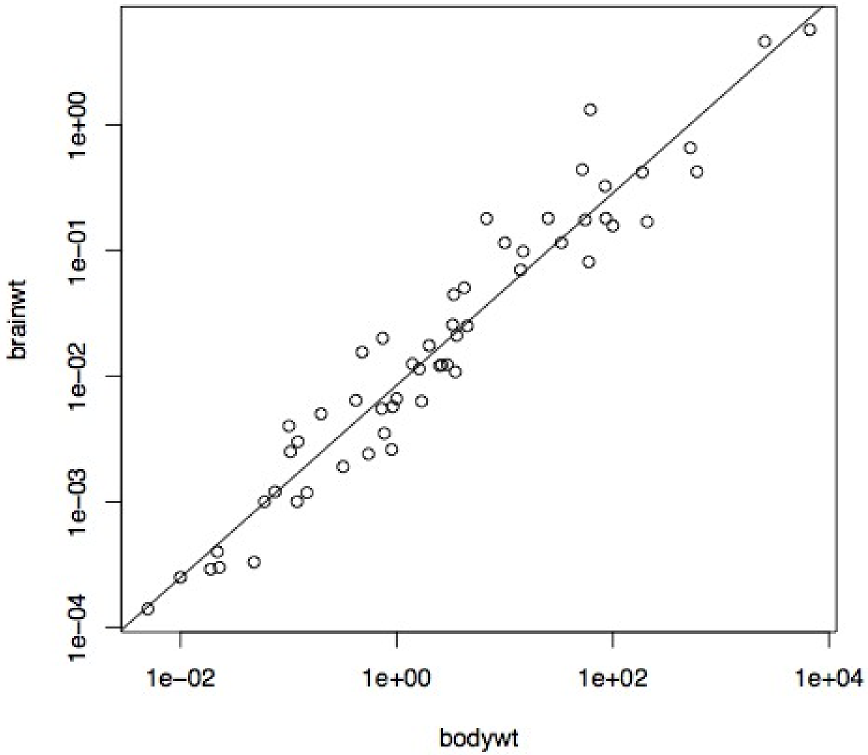
Load the **ggplot** library. We'll use one of the R built-in datasets as a demonstration.

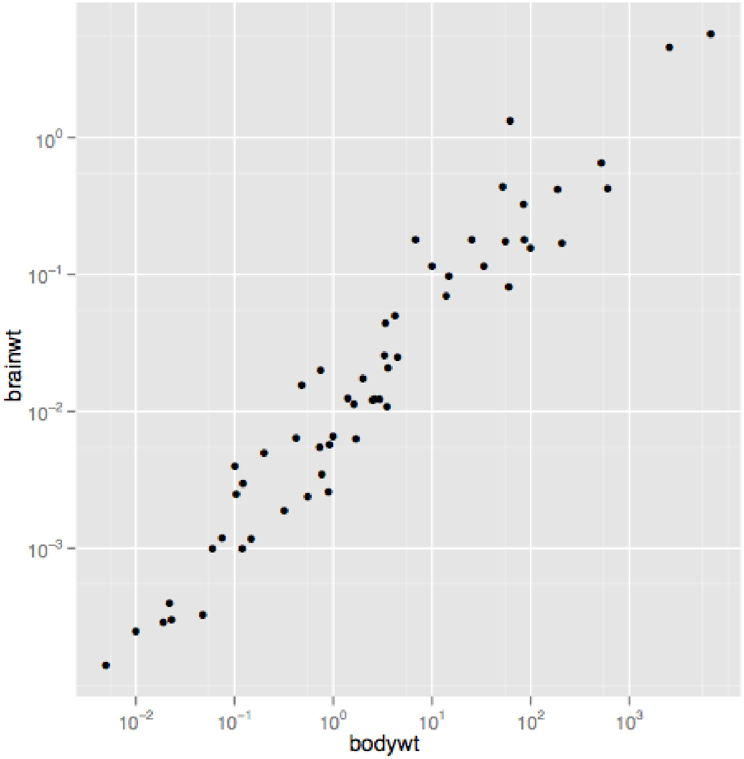
```
> install.packages("ggplot2") # if ggplot is not installed
> library(ggplot2)
> attach(msleep)
```

ggplot

Here's a linear regression plot with traditional graphics, and with ggplot:

```
> plot(bodywt, brainwt, log = "xy")
> abline(lm(log10(brainwt) ~ log10(bodywt)))
> qplot(bodywt, brainwt, log = "xy")
```

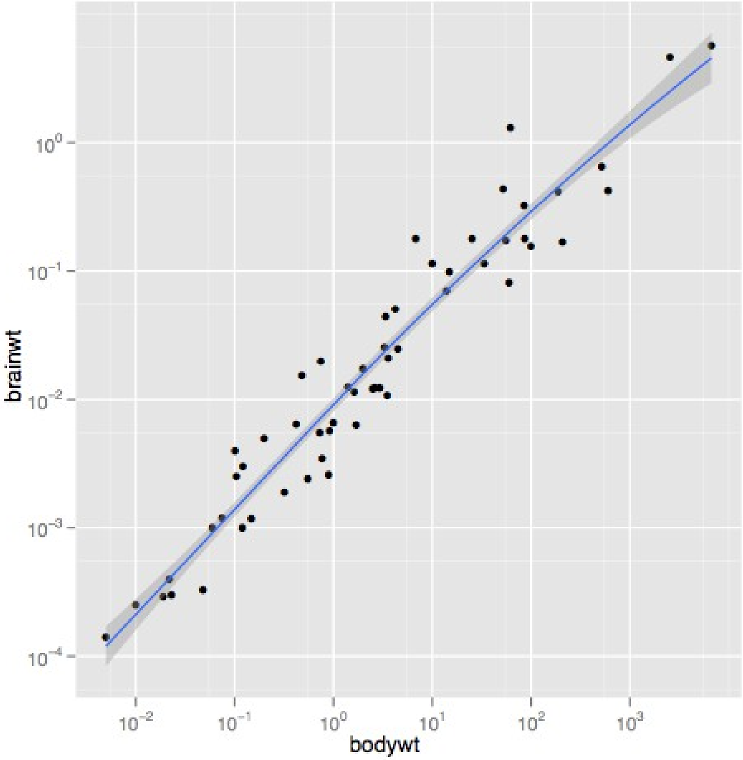


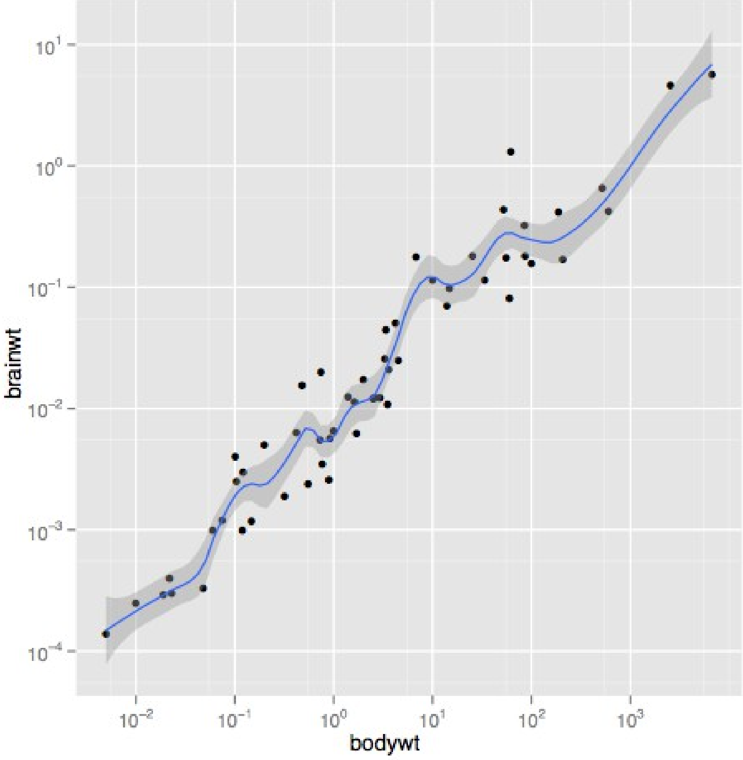


ggplot

Plot types in ggplot are known as *geoms*. We can add geoms easily, and do sophisticated things that aren't possible in traditional graphics:

```
> qplot(bodywt, brainwt, log = "xy", geom = c("point", "smooth"), span = 1)
> qplot(bodywt, brainwt, log = "xy", geom = c("point", "smooth"), span = 0.2)
```





ggplot

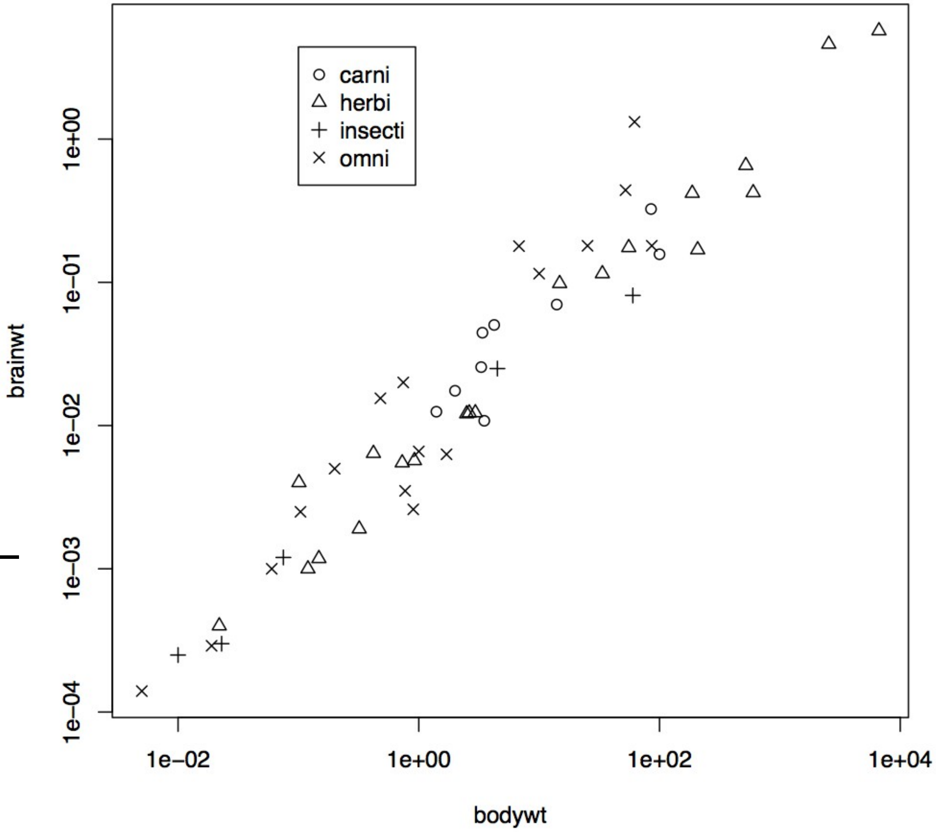
When we make plots, we want to focus on our data, and not be distracted by the drawing commands. The traditional graphics functions can be very distracting:

```
> plot(bodywt, brainwt, log = "xy", pch = as.numeric(vore))
> legend(0.1, 4.4, levels(vore), pch = 1:4)
```

Here, we are manually converting the **vore** factor to numbers, in order to select four symbols for **pch** to use for each level of the factor.

Then, when we draw the legend, we have to specify the same four numbers and give the levels of the factor again.

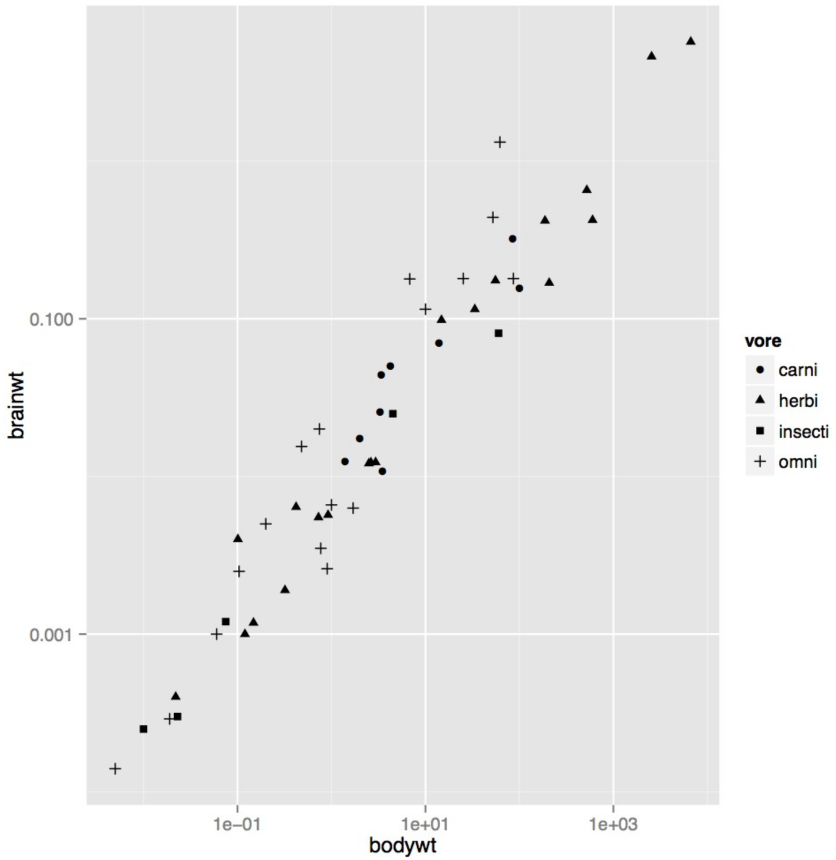
Note there is no connection between the call to **plot** and the call to **legend** – the inputs to **legend** are completely separate, which means we have to do fiddly, unnecessary work twice.

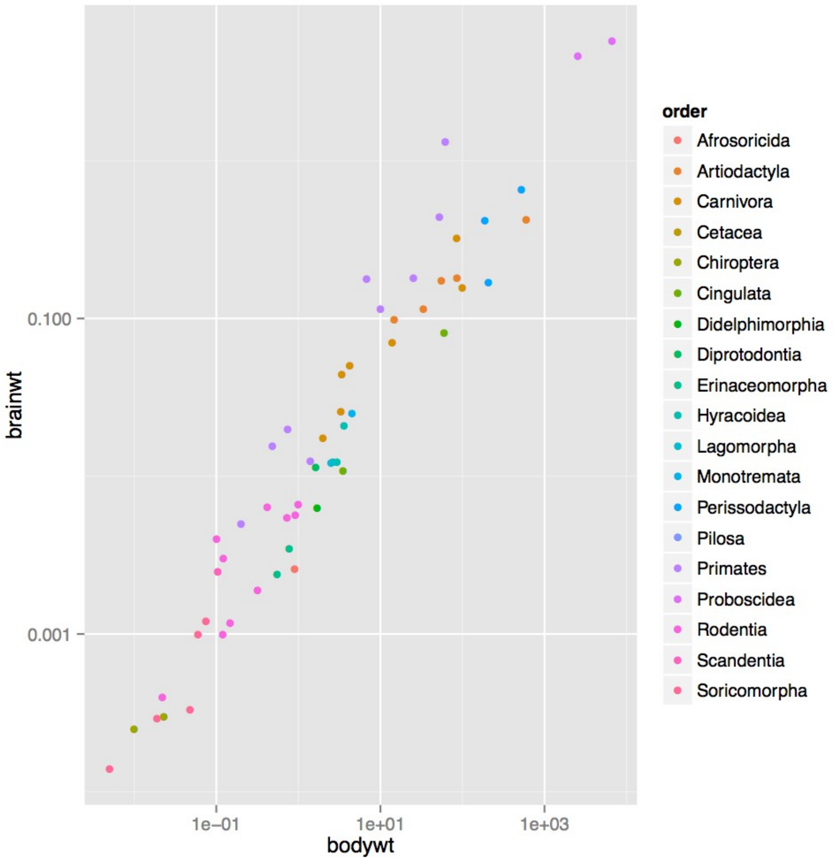


ggplot

With ggplot, we can just do this, and it takes care of choosing symbols and creating a legend:

```
> qplot(bodywt, brainwt, log = "xy", shape = vore)
> qplot(bodywt, brainwt, log = "xy", colour = order)
```





References

- Official documentation on:
  - http://cran.r-project.org/manuals.html
- A good repository of R recipes:
  - Quick-R: http://www.statmethods.net/
- Don't forget that many packages come with tutorials (vignettes)
- Website of this course:
  - http://logic.sysbiol.cam.ac.uk/teaching/Rcourse/
- R forums (stackoverflow & official):
  - http://stackoverflow.com/questions/tagged/r
  - http://news.gmane.org/gmane.comp.lang.r.general
- Plenty of textbooks to choose from, comprehensive list + reviews:
  - http://www.r-project.org/doc/bib/R-books.html

Thanks for your attention!

END OF COURSE