# 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 1 schedule

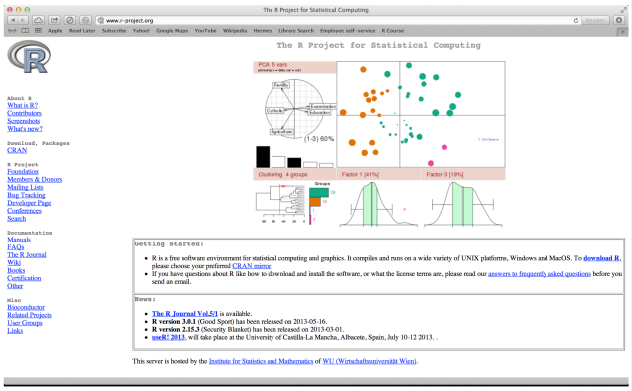
1. Introduction to R and its environment
2. Data structures
3. Data analysis example
4. Programming techniques
5. Statistics

Introduction to R and its environment

**1**

## What's R?

- A statistical programming environment
  - based on S
  - Suited to high level data analysis
- Open source & cross platform
- Extensive graphics capabilities
- Diverse range of add-on packages
- Active community of developers
- Thorough documentation




www.r-project.org

## Various platforms supported

- Release 3.1.0 (April 2014)
  - Base package
  - Contributed packages (general purposes extras)
  - >5400 available packages
- Download from http://www.stats.bris.ac.uk/R/
- Windows, Mac and Linux versions available

- Executed using command line, or a graphical user interface (GUI)
- On this course, we use the RStudio GUI (www.rstudio.com)
- Everything you need is installed on the training machines
- If you are using your own machine, download both R and RStudio

## Getting Started

- R is a program which, once installed on your system, can be launched and is immediately ready to take input directly from the user
- There are two ways to launch R:
  1) From the command line (particularly useful if you're quite familiar with Linux)
  2) As an application called RStudio (very good for beginners)

## Prepare to launch R
From command line

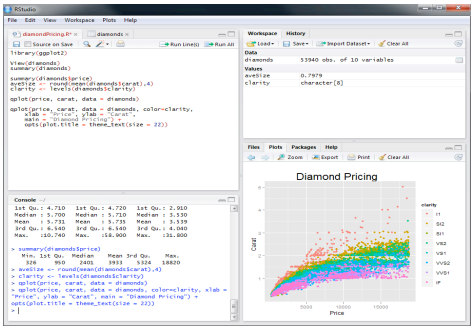
- To start R in Linux we need to enter the Linux console (also called Linux terminal and Linux shell)
- To start R, at the prompt simply type:

```
$ R
```

- If R doesn't print the welcome message, call us to help!

## Prepare to launch R
Using RStudio

- To launch RStudio, find the RStudio icon in the menu bar on the left of the screen and double-click



## The Working Directory (wd)

- Like many programs R has a concept of a working directory (wd)
- It is the place where R will look for files to execute and where it will save files, by default
- For this course we need to set the working directory to the location of the course scripts
- At the command prompt in the terminal or in RStudio console type:

```
> setwd("R_course/Day_1_scripts")
```

- Alternatively in RStudio use the mouse and browse to the directory location
- Session → Set Working Directory → Choose Directory...

## Basic concepts in R
### command line calculation

- The command line can be used as a calculator. Type:

```
> 2 + 2
[1] 4

> 20/5 - sqrt(25) + 3^2
[1] 8

> sin(pi/2)
[1] 1
```

- Note: The number in the square brackets is an indicator of the position in the output. In this case the output is a 'vector' of length 1 (i.e. a single number). More on vectors coming up...

## Basic concepts in R
### variables

- A variable is a letter or word which takes (or contains) a value. We use the assignment 'operator', <-

```
> x <- 10
> x
[1] 10
> myNumber <- 25
> myNumber
[1] 25
```

- We can perform arithmetic on variables:

```
> sqrt(myNumber)
[1] 5
```

- We can add variables together:

```
> x + myNumber
[1] 35
```

## Basic concepts in R
### variables

- We can change the value of an existing variable:

```
> x <- 21
> x
[1] 21
```

- We can set one variable to equal the value of another variable:

```
> x <- myNumber
> x
[1] 25
```

- We can modify the contents of a variable:

```
> myNumber <- myNumber + sqrt(16)
[1] 29
```

## Basic concepts in R
### functions

- **Functions** in R perform operations on **arguments** (the input(s) to the function). We have already used **sin(x)** which returns the sine of **x**. In this case the function has one argument, **x**. Arguments are *always* contained in parentheses, i.e. curved brackets **()**, separated by commas.
- Try these:

```
> sum(3, 4, 5, 6)
[1] 18
> max(3, 4, 5, 6)
[1] 6
> min(3, 4, 5, 6)
[1] 3
```

- Arguments can be named or unnamed, but if they are unnamed they must be ordered (we will see later how to find the right order).

```
> seq(from=2, to=10, by=2)
[1] 2 4 6 8 10
> seq(2, 10, 2)
[1] 2 4 6 8 10
```

## Basic concepts in R
### vectors

- The basic data structure in R is a **vector** – an ordered collection of values. R even treats single values as 1-element vectors. The function **c()** *combines* its arguments into a vector:

```
> x <- c(3, 4, 5, 6)
> x
[1] 3 4 5 6
```

- As mentioned, the square brackets **[]** indicate position within the vector (the **index**). We can extract individual elements by using the **[]** notation:

```
> x[1]
[1] 3
> x[4]
[1] 6
```

- We can even put a vector inside the square brackets (vector indexing):

```
> y <- c(2, 3)
> x[y]
[1] 4 5
```

## Basic concepts in R
### vectors

- There are a number of shortcuts to create a vector. Instead of:

```
> x <- c(3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
```

- we can write:

```
> x <- 3:12
```

- or we can use the **seq()** function, which returns a vector:

```
> x <- seq(2, 10, 2)
> x
 [1] 2 4 6 8 10
> x <- seq(2, 10, length.out = 7)
```

- 
```
> x
 [1] 2.00000 3.33333 4.66667 6.00000 7.33333 8.66667 10.00000
```

- or the **rep()** function:

```
> y <- rep(3, 5)
```

- 
```
> y
[1] 3 3 3 3 3
> y <- rep(1:3, 5)
> y
[1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
```

## Basic concepts in R
### vectors

- We have seen some ways of extracting elements of a vector. We can use these shortcuts to make things easier (or more complex!)

```
> x <- 3:12
> x[3:7]
 [1] 5 6 7 8 9
> x[seq(2, 6, 2)]
 [1] 4 6 8
> x[rep(3, 2)]
 [1] 5 5
```

- We can add an element to a vector

```
> y <- c(x, 1)
> y
[1] 3 4 5 6 7 8 9 10 11 12 1
```

- We can glue vectors together

```
> z <- c(x, y)
> z
[1] 3 4 5 6 7 8 9 10 11 12 3 4 5 6 7 8 9 10 11 12 1
```

## Basic concepts in R
### vectors

- We can remove element(s) from a vector

```
> x <- 3:12
> x[-3]
 [1] 3 4 6 7 8 9 10 11 12
> x[-(5:7)]
 [1] 3 4 5 6 10 11 12
> x[-seq(2, 6, 2)]
 [1] 3 5 7 9 10 11 12
```

- Finally, we can modify the contents of a vector

```
> x[6] <- 4
> x
[1] 3 4 5 6 7 4 9 10 11 12
> x[3:5] <- 1
> x
[1] 3 4 1 1 1 4 9 10 11 12
```

- Remember! **Square** brackets for indexing **[]**, **parentheses** for function arguments **()**.

## Basic concepts in R
### vector arithmetic

- When applying all standard arithmetic operations to vectors, application is element-wise

```
> x <- 1:10
> y <- x*2
> y
[1] 2 4 6 8 10 12 14 16 18 20
> z <- x^2
> z
[1] 1 4 9 16 25 36 49 64 81 100
```

- Adding two vectors

```
> y + z
[1] 3 8 15 24 35 48 63 80 99 120
```

- If vectors are not the same length, the shorter one will be recycled:

```
> x + 1:2
[1] 2 4 4 6 6 8 8 10 10 12
```

- But be careful if the vector lengths aren't factors of each other:

```
> x + 1:3
```

## Basic concepts in R
### Character vectors and naming

- All the vectors we have seen so far have contained numbers, but we can also store strings in vectors – this is called a **character** vector.

```
> gene.names <- c("Pax6","Beta-actin","FoxP2","Hox9")
```

- We can name elements of vectors using the **names** function, which can be useful to keep track of the meaning of our data:

```
> gene.expression <- c(0,3.2,1.2,-2)
> gene.expression
[1]  0.0  3.2  1.2 -2.0
> names(gene.expression)<-gene.names
> gene.expression
      Pax6 Beta-actin      FoxP2       Hox9
       0.0        3.2        1.2       -2.0
```

- We can also use the **names** function to get a vector of the names of an object:

```
> names(gene.expression)
[1] "Pax6"       "Beta-actin" "FoxP2"      "Hox9"
```

## Exercise: genes and genomes

- Let's try some vector arithmetic. Here are the genome lengths and number of protein coding genes for several model organisms:

| Species | Genome size (Mb) | Protein coding genes |
|---|---|---|
| *Homo sapiens* | 3,102 | 20,774 |
| *Mus musculus* | 2,731 | 23,139 |
| *Drosophila melanogaster* | 169 | 13,937 |
| *Caenorhabditis elegans* | 100 | 20,532 |
| *Saccharomyces cerevisiae* | 12 | 6,692 |

- Create **genome.size** and **coding.genes** vectors to hold the data in each column using the **c** function. Create a **species.name** vector and use this vector to name the values in the other two vectors.

## Exercise: genes and genomes

- Let's assume a coding gene has an average length of 1.5 kilobases. On average, how many base pairs of each genome is made of coding genes? Create a new vector to record this called **coding.bases**.
- What percentage of each genome is made up of protein coding genes? Use your **coding.bases** and **genome.size** vectors to calculate this. (See earlier slides for how to do division in R.)
- How many times more bases are used for coding in the human genome compared to the yeast genome? How many times more bases are in the human genome in total compared to the yeast genome? Look up indices of your vectors to find out.

## Answers to genome exercise

- Creating vectors:

```
> genome.size<-c(3102,2731,169,100,12)
> coding.genes<-c(20774,23139,13937,20532,6692)
> species.name<-c("H. sapiens","M. musculus","D. melanogaster","C. elegans","S. cerevisiae")
> names(genome.size)<-species.name
> names(coding.genes)<-species.name
```

- To calculate the number of coding bases, we need to use the same scale as we used for genome size: 1.5 kilobases is 0.0015 Megabases.

```
> coding.bases<-coding.genes*0.0015
> coding.bases
     H. sapiens     M. musculus D. melanogaster      C. elegans   S. cerevisiae
        31.1610         34.7085         20.9055         30.7980         10.0380
```

## Answers to genome exercise

- To calculate the percentage of coding bases in each genome:

```
> coding.pc<-coding.bases/genome.size*100
> coding.pc
     H. sapiens     M. musculus D. melanogaster      C. elegans   S. cerevisiae
       1.004545        1.270908       12.370118       30.798000       83.650000
```

- To compare human to yeast:

```
> coding.bases[1]/coding.bases[5]
H. sapiens
  3.104304
> genome.size[1]/genome.size[5]
H. sapiens
     258.5
```

- Note that if a new vector is created using a named vector, the names are usually carried across to the new vector. Sometimes this is what we want (as for **coding.pc**) but sometimes it is not (when we are comparing human to yeast). We can remove names by setting them to the special NULL value:

```
> names(coding.pc)<-NULL
> coding.pc
[1]  1.004545  1.270908 12.370118 30.798000 83.650000
```

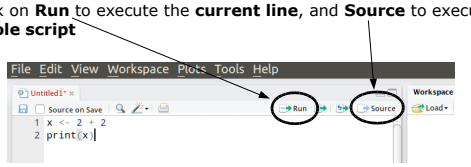
## Writing scripts with Rstudio

Typing lots of commands directly to R can be tedious. A better way is to write the commands to a file and then load it into R.

- Click on **File -> New** in Rstudio
- Type in some R code, e.g.

```
x <- 2 + 2
print(x)
```

- Click on **Run** to execute the **current line**, and **Source** to execute the **whole script**




Sourcing can also be performed manually with `source("myScript.R")`

## Getting Help

- To get help on any R function, type `?` followed by the function name. For example:

```
> ?seq
```

- This retrieves the syntax and arguments for the function. You can see the default order of arguments here. The help page also tells you which **package** it belongs to.
- There will typically be example usage, which you can test using the **example** function:

```
> example(seq)
```

- If you can't remember the exact name type `??` followed by your guess. R will return a list of possibles

```
> ??plot
```

## Interacting with the R console

- R console symbols
  - ; end of line
    - Enables multiple commands to be placed on one line of text
  - # comment
    - indicates text is a comment and not executed
  - + command line wrap
    - R is waiting for you to complete an expression
- Ctrl-c or escape to clear input line and try again
- Ctrl-l to clear window
- Press q to leave help (using R from the terminal)
- Use the TAB key for command auto completion
- Use up and down arrows to scroll through the command history

## R packages

- R comes ready loaded with various libraries of functions called **packages**. e.g. the function **sum()** is in the **base** package and **sd()**, which calculates the standard deviation of a vector, is in the **stats** package
- There are 1000s of additional packages provided by third parties, and the packages can be found in numerous server locations on the web called **repositories**
- The two repositories you will come across the most are
  - **The Comprehensive R Archive Network (CRAN)**
  - **Bioconductor**
- CRAN is mirrored in many locations. Set your local mirror in RStudio using Tools → Options, and choose a CRAN mirror
- Set the Bioconductor package download tool by typing:

```
> source("http://bioconductor.org/biocLite.R")
```

- Bioconductor packages are then loaded with the biocLite() function:

```
> biocLite("PackageName")
```

## R packages

- 5400+ packages on CRAN:
  - Use CRAN search to find functionality you need:
    http://cran.r-project.org/search.html
  - Or, look for packages by theme:
    http://cran.r-project.org/web/views/
- 750 packages in Bioconductor:
  - Specialised in genomics:
    http://www.bioconductor.org/packages/release/bioc/
- **Other repositiories:**
- 1700+ projects on R-forge:
  - http://r-forge.r-project.org/
- R graphical manual:
  - http://rgm3.lab.nig.ac.jp/RGM

Bottomline: **always** first look if there is already an R package that does what you want before trying to implement it yourself

## Exercise: Install Packages ggplot and DESeq

- ggplot2 is a commonly used graphics package (we will try it tomorrow).
  - Use `install.packages()` function...

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

  - or in RStudio goto Tools → Install Packages... and type the package name
- DESeq is a BioConductor package (www.bioconductor.org)
  - Use `biocLite()` function

```
biocLite("DESeq")
```

- R needs to be told to use the new functions from the installed packages
  - Use `library(…)` function to load the newly installed features

```
library(ggplot2) # loads ggplot functions
library(DESeq) # loads DESeq functions
```

  - `library()`
    - Lists all the packages you've got installed locally

Data structures

# 2

## R is designed to handle experimental data

- Although the basic unit of R is a vector, we usually handle data in **data frames**.
- A data frame is a set of observations of a set of variables – in other words, the outcome of an experiment.
- For example, we might want to analyse information about a set of patients. To start with, let's say we have ten patients and for each one we know their name, sex, age, weight and whether they give consent for their data to be made public.

## The patients data frame

We are going to create a data frame called 'patients', which will have ten rows (observations) and seven columns (variables). The columns must all be equal lengths.

| | First_Name | Second_Name | Full_Name | Sex | Age | Weight | Consent |
|---|---|---|---|---|---|---|---|
| 1 | Adam | Jones | Adam Jones | Male | 50 | 70.8 | TRUE |
| 2 | Eve | Parker | Eve Parker | Female | 21 | 67.9 | TRUE |
| 3 | John | Evans | John Evans | Male | 35 | 75.3 | FALSE |
| 4 | Mary | Davis | Mary Davis | Female | 45 | 61.9 | TRUE |
| 5 | Peter | Baker | Peter Baker | Male | 28 | 72.4 | FALSE |
| 6 | Paul | Daniels | Paul Daniels | Male | 31 | 69.9 | FALSE |
| 7 | Joanna | Edwards | Joanna Edwards | Female | 42 | 63.5 | FALSE |
| 8 | Matthew | Smith | Matthew Smith | Male | 33 | 71.5 | TRUE |
| 9 | David | Roberts | David Roberts | Male | 57 | 73.2 | FALSE |
| 10 | Sally | Wilson | Sally Wilson | Female | 62 | 64.8 | TRUE |

Let's see how we can construct this from scratch.

## Character, numeric and logical data types

- Each column is a vector, like previous vectors we have seen, for example:

```
> age<-c(50, 21, 35, 45, 28, 31, 42, 33, 57, 62)
> weight<-c(70.8, 67.9, 75.3, 61.9, 72.4, 69.9, 63.5, 71.5, 73.2, 64.8)
```

- We can define the names using character vectors:

```
> firstName<- c("Adam", "Eve", "John", "Mary", "Peter", "Paul", "Joanna",
"Matthew", "David", "Sally")
> secondName<-c("Jones", "Parker", "Evans", "Davis", "Baker", "Daniels",
"Edwards", "Smith", "Roberts", "Wilson")
```

- We also have a new type of vector, the **logical** vector, which only contains the values TRUE and FALSE:

```
> consent<-c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE)
```

## Character, numeric and logical data types

- Vectors can only contain one type of data; we cannot mix numbers, characters and logical values in the same vector. If we try this, R will convert everything to characters:

```
> c(20, "a string", TRUE)
[1] "20"       "a string" "TRUE"
```

- We can see the type of a particular vector using the **mode** function:

```
> mode(firstName)
[1] "character"
> mode(age)
[1] "numeric"
> mode(weight)
[1] "numeric"
> mode(consent)
[1] "logical"
```

## Factors

- Character vectors are fine for some variables, like names.
- But sometimes we have categorical data and we want R to recognize this.
- A factor is R's data structure for categorical data.

```
> sex<-c("Male", "Female", "Male", "Female", "Male", "Male", "Female",
"Male", "Male", "Female")
> sex
 [1] "Male"   "Female" "Male"   "Female" "Male"   "Male"   "Female" "Male"
"Male"   "Female"
> factor(sex)
 [1] Male   Female Male   Female Male   Male   Female Male   Female
Levels: Female Male
```

- R has converted the strings of the sex character vector into two **levels**, which are the categories in the data.
- Note the values of this factor are not character strings, but levels.
- We can use this factor to compare data for males and females.

## Creating a data frame (first attempt)

- We can construct a data frame from other objects:

```
> patients<-data.frame(firstName, secondName, paste(firstName,secondName),
    sex, age, weight, consent)
> patients
   firstName secondName paste.firstName..secondName.    sex age weight consent
1       Adam      Jones                  Adam Jones   Male  50   70.8    TRUE
2        Eve     Parker                 Eve Parker Female  21   67.9    TRUE
3       John      Evans                 John Evans   Male  35   75.3   FALSE
4       Mary      Davis                 Mary Davis Female  45   61.9    TRUE
5      Peter      Baker                Peter Baker   Male  28   72.4   FALSE
6       Paul    Daniels               Paul Daniels   Male  31   69.9   FALSE
7     Joanna    Edwards             Joanna Edwards Female  42   63.5   FALSE
8    Matthew      Smith              Matthew Smith   Male  33   71.5    TRUE
9      David    Roberts              David Roberts   Male  57   73.2   FALSE
10     Sally     Wilson               Sally Wilson Female  62   64.8    TRUE
```

- The **paste** function joins character vectors together.
- We can access particular variables using the **dollar** operator:

```
> patients$age
 [1] 50 21 35 45 28 31 42 33 57 62
```

## Naming data frame variables

- R has inferred the names of our data frame variables from the names of the vectors or the commands (eg the paste command).
- We can name the variables after we have created a data frame using the **names** function, and we can use the same function to see the names:

```
> names(patients)<-c("First_Name", "Second_Name", "Full_Name", "Sex",
"Age", "Weight", "Consent")
> names(patients)
 [1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
"Weight"      "Consent"
```

- Or we can name the variables when we define the data frame:

```
> patients<-data.frame(First_Name=firstName, Second_Name=secondName,
Full_Name=paste(firstName,secondName), Sex=sex, Age=age, Weight=weight,
Consent=consent)
> names(patients)
 [1] "First_Name"  "Second_Name" "Full_Name"   "Sex"         "Age"
"Weight"      "Consent"
```

## Factors in data frames

- When creating a data frame, R assumes all character vectors should be categorical variables and converts them to factors. This is not always what we want:

```
> patients$firstName
 [1] Adam    Eve     John    Mary    Peter   Paul    Joanna  Matthew David   Sally
Levels: Adam David Eve Joanna John Mary Matthew Paul Peter Sally
```

- We can avoid this by asking R not to treat strings as factors, and then explicitly stating when we want a factor by using **factor**:

```
> patients<-data.frame(First_Name=firstName, Second_Name=secondName,
Full_Name=paste(firstName,secondName), Sex=factor(sex), Age=age,
Weight=weight, Consent=consent, stringsAsFactors=FALSE)
> patients$Sex
 [1] Male   Female Male   Female Male   Female Male   Male   Female
Levels: Female Male
> patients$First_Name
 [1] "Adam"    "Eve"     "John"    "Mary"    "Peter"   "Paul"    "Joanna"
"Matthew" "David"   "Sally"
```

## Matrices

**`matrix(..., ncol=…, nrow=…)`**

- Data frames are R's speciality, but R also handles matrices:

```
> e <- matrix(1:10, nrow=5, ncol=2)
> e
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> f <- matrix(1:10, nrow=2, ncol=5)
> f
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8   10
> f %*% e
     [,1] [,2]
[1,]   95  220
[2,]  110  260
```

The `%*%` operator is the matrix multiplication operator, not the standard multiplication operator.

## Lists

**`list(name1=obj1, name2=obj2, …)`**

- We have seen that vectors can only hold data of one type. How can we store data of multiple types? Or vectors of different lengths in one object?
- We can use lists. A list can contain objects of any type.

```
one.to.ten <- 1:10
uniform.mat <- matrix(runif(100),ncol=10,nrow=10)
year.to.october <- data.frame(a, month.name[1:10])

myList<-list( ls.obj.1=one.to.ten, ls.obj.2=uniform.mat,
ls.obj.3=year.to.october )
names(myList)
```

- We can use the dollar syntax to access list items (in fact, a data frame is a special type of list):

```
myList$ls.obj.1
```

- We can also use **`myList[[1]]`** to get the first item in the list.
- (For the curious: this double indexing is necessary because lists are in fact just like vectors – they can only contain one type of object. But one of the types they can contain is a list. So any list like the above is actually a list of lists; the first element `myList[1]` is a list containing a vector, and so we need double indexing to actually get the vector.)

## Indexing data frames and matrices

Special cases:
a[*i*, ] i-th row
a[ ,*j*] j-th column

- You can index multidimensional data structures like matrices and data frames using commas. If you don't provide an index for either rows or columns, all of the rows or columns will be returned.

```
object [ rows , columns ]
```

```
> e[1,2]
[1] 6
> e[1,]
[1] 1 6
> patients[1,2]
[1] "Jones"
> patients[1,]
  First_Name Second_Name  Full_Name  Sex Age Weight Consent
1       Adam       Jones Adam Jones Male  50   70.8    TRUE
```

## Advanced indexing

- As values in R are really vectors, so indices are actually vectors, and can be numeric or logical:

```
> s <- letters[1:5]
> s[c(1,3)]
[1] "a" "c"
> s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
[1] "a" "c"
> a<-1:5
> a<3
[1]  TRUE  TRUE FALSE FALSE FALSE
> s[a<3]
[1] "a" "b"
> s[a>1 & a<3]
[1] "b"
> s[a==2]
[1] "b"
```

## Operators

- arithmetic
  +, -, *, /, ^
- comparison
  <, >, <=, >=, ==, !=
- logical
  !, &, |, xor

(equal to, not equal to)

these always return logical values ! (TRUE, FALSE)

## Exercise

- Create a data frame called **my.patients** using the instructions in the slides. Change the data if you like.
- Check you have created the data frame correctly by loading the original version from this file in the *Day_1_scripts* folder using **source**:

```
> source("1.2_patients.R")
```

- Remake your data frame with three new variables: country, continent, and height. Make up the data. Make country a character vector but continent a factor.
- Try the **summary** function on your data frame. What does it do? How does it treat vectors (numeric, character, logical) and factors? (What does it do for matrices?)
- Use logical indexing to select the following patients:
  - Patients under 40
  - Patients who give consent to share their data
  - Men who weigh as much or more than the average European male (70.8 kg)

## Logical indexing answers

- Patients under 40:

```
> patients[patients$Age<40,]
```

- Patients who give consent to share their data:

```
> patients[patients$Consent==TRUE,]
```

- Men who weigh as much or more than the average European male (70.8 kg):

```
> patients[patients$Sex=="Male" & patients$Weight>=70.8,]
```

R for data analysis

# 3

## 3 steps to Basic data analysis

1. Reading in data
   - read.table()
   - read.csv(), read.delim()

2. Analysis
   - Manipulating & reshaping the data
   - Any maths you like
   - Plotting the outcome
     - High level plotting functions (covered tomorrow)

3. Writing out results
   - write.table()
   - write.csv()

## A simple walkthrough
## Exemplifies 3 steps to R analysis

- 50 neuroblastoma patients were tested for NMYC gene copy number by interphase nuclei FISH
  - Amplification of NMYC correlates with worse prognosis
  - We have count data
    - Numbers of cells per patient assayed
      - For each we have NMYC copy number relative to base ploidy
- We need to determine which patients have amplifications
  - (i.e >33% of cells show NMYC amplification)

## Step 1.
## Read in the data

| Patient | Nuclei | NB_Amp | NB_Nor | NB_Del |
|---|---|---|---|---|
| 1 | 44 | 0 | 41 | 3 |
| 2 | 67 | 3 | 58 | 6 |
| 3 | 33 | 7 | 26 | 0 |
| 4 | 36 | 6 | 30 | 0 |
| 5 | 51 | 5 | 45 | 1 |
| 6 | 43 | 0 | 38 | 5 |
| 7 | 64 | 1 | 56 | 7 |
| 8 | 58 | 2 | 49 | 7 |
| 9 | 56 | 0 | 53 | 3 |
| 10 | 66 | 0 | 56 | 10 |
| 11 | 33 | 13 | 19 | 1 |

This data is a tab delimited text file
Each row is a record, each column is a field
Columns are separated by tabs in the text.

We need to read in the results table and assign it to an object (rawData)

```
rawData <- read.delim("1.3_NBcountData.txt")
rawData[1:10,]    # View the first 10 rows to ensure import is OK
                  # Note data frame contains a patient index column
```

If the data had been comma separated values, then sep=","

```
read.csv("1.3_NBcountData.csv")
```

?read.table for a full list of arguments

1.3_NBcountData.R (script commands)

1.3_NBcountData.txt (data file)

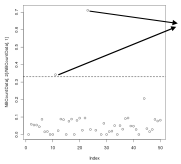
## Handling missing values

- The data frame contains some 'NA' values, which means the values are missing – a common occurrence in real data collection.
- NA is a special value that can be present in objects of any type (logical, character, numeric etc).
- NA is not the same as NULL. NULL is an empty R object. NA is one missing value within an R object (like a data frame or a vector).
- Often R functions will handle NAs gracefully, but sometimes we have to tell the functions what to do with them. R has some built-in functions for dealing with NAs, and functions often have their own arguments (like na.rm) for handling them.

```
> x<-c(1,NA,3)
> mean(x)
[1] NA
> mean(x,na.rm=TRUE)
[1] 2
> mean(na.omit(x))
[1] 2
> is.na(x)
[1] FALSE  TRUE FALSE
```

## Step 2.
## Analysis (reshaping data & maths)

- Our analysis involves identifying patients with > 33% NB amplification
  - `prop <- rawData$NB_Amp / rawData$Nuclei # create an index of results`
  - `amp <- which(prop > 0.33) # Get sample names of amplified patients`
- We can plot a simple chart of the % NB amplification
  - `plot(prop, ylim=c(0,1))`
  - `abline(h=0.33)`



These 2 samples are amplified (11 & 23)

## Step 3.
## Outputting the results

- We write out a data frame of results (patients > 33% NB amplification) as a 'comma separated values' text file
  - `write.csv(rawData[amp,],file="selectedSamples.csv") # Export table, file name = selectedSamples.csv`
    - Files are directly readable by Excel and Calc
- Its often helpful to double check where the data has been saved
  - Use get working directory function
    - `getwd() # print working directory`

## Data analysis exercise:
## Which samples are near normal?

• Patients are near normal if:

(NB_Amp/Nuclei <0.33 & NB_Del ==0)

• Modify the condition in our previous code to find these patients

• Write out a results file of the samples that match these criteria, and open it in a spreadsheet program

1.3_NBcountData.R
(script commands)

## Solution to NB normality test
### Basic data analysis

```
> norm <- which( prop < 0.33 & rawData$NB_Del==0)
> norm

[1] 3  4  7 15 20 24 36 37 42 47


> write.csv(rawData[norm,],"My_NB_output.csv")
```

R programming techniques

**4**

## Basic R 'Built-in' functions for working with objects

- R has many built-in functions for doing simple calculations on objects. Start with a random sample of 15 numbers from 1 to 100 and try the functions below.

```
> x<-sample(100,15)
```

- Arithmetic with vectors
  - Min / Max value number in a series

```
min(x) ; max(x)
```

  - Sum of values in a series

```
sum(x)
```

  - Average estimates (mean / median)

```
mean(x) ; median(x)
```

  - Range of values in a series

```
range(x)
```

  - Variance

```
var(x)
```

- Arithmetic with vectors
  - Rank ordering

```
rank(x)
```

  - Quantiles

```
quantile(x); boxplot(x)
```

  - Square Root

```
sqrt(x)
```

  - Standard deviation

```
sd(x)
```

  - Trigonometry functions

```
tan(x) ; cos(x) ; sin(x)
```

## Basic R 'Built-in' functions for working with data frames

- We have seen before how we can get the **names** of our variables, but for dataframes and matrices we can also get these names with **colnames**, and the names of the rows with **rownames**:

```
> names(patients)
[1] "First_Name"  "Second_Name" "Full_Name"   "Sex"     "Age"     "Weight"      "Consent"
> colnames(patients)
[1] "First_Name"  "Second_Name" "Full_Name"   "Sex"     "Age"     "Weight"      "Consent"
> rownames(patients)
[1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

We can get the numbers of rows or columns with **nrow** and **ncol**:

```
> nrow(patients)
[1] 10
> ncol(patients)
[1] 7
```

We can also find the length of a vector or a list with **length**, although this may give confusing results for some structures, like data frames:

```
> length(c(1,2,3,4,5))
[1] 5
> length(patients)
[1] 7
> length(patients$Age)
[1] 10
```

Remember, a data frame is a list of variables, so its length is the number of variables. The length of one of the variable vectors (like Age) is the number of observations.

## Basic R 'Built-in' functions for working with data frames

We can add rows or columns to a data frame using **rbind** and **cbind**:

```
> newpatient<-c("Kate","Lawson","Kate Lawson","Female","35","62.5","TRUE")
> rbind(patients,newpatient)
   First_Name Second_Name      Full_Name    Sex Age Weight Consent
1        Adam       Jones     Adam Jones   Male  50   70.8    TRUE
2         Eve      Parker     Eve Parker Female  21   67.9    TRUE
3        John       Evans     John Evans   Male  35   75.3   FALSE
4        Mary       Davis     Mary Davis Female  45   61.9    TRUE
5       Peter       Baker    Peter Baker   Male  28   72.4   FALSE
6        Paul     Daniels   Paul Daniels   Male  31   69.9   FALSE
7      Joanna     Edwards Joanna Edwards Female  42   63.5   FALSE
8     Matthew       Smith  Matthew Smith   Male  33   71.5    TRUE
9       David     Roberts  David Roberts   Male  57   73.2   FALSE
10      Sally      Wilson   Sally Wilson Female  62   64.8    TRUE
11       Kate      Lawson    Kate Lawson Female  35   62.5    TRUE
> cbind(patients,10:1)
```

- We can also remove rows and columns:

```
patients[-1,] # remove first row
patients[,-1] # remove first column
```

## Basic R 'Built-in' functions for working with data frames

Sorting a vector with **sort**:

```
> sort(patients$Second_Name)
[1] "Baker"   "Daniels" "Davis"   "Edwards" "Evans"   "Jones"   "Parker"  "Roberts" "Smith"
 "Wilson"
```

Sorting a data frame by one variable with **order**:

```
> order(patients$Second_Name)
[1]  5  6  4  7  3  1  2  9  8 10
> patients[order(patients$Second_Name),]
   First_Name Second_Name      Full_Name    Sex Age Weight Consent
5       Peter       Baker    Peter Baker   Male  28   72.4   FALSE
6        Paul     Daniels   Paul Daniels   Male  31   69.9   FALSE
4        Mary       Davis     Mary Davis Female  45   61.9    TRUE
7      Joanna     Edwards Joanna Edwards Female  42   63.5   FALSE
3        John       Evans     John Evans   Male  35   75.3   FALSE
1        Adam       Jones     Adam Jones   Male  50   70.8    TRUE
2         Eve      Parker     Eve Parker Female  21   67.9    TRUE
9       David     Roberts  David Roberts   Male  57   73.2   FALSE
8     Matthew       Smith  Matthew Smith   Male  33   71.5    TRUE
10      Sally      Wilson   Sally Wilson Female  62   64.8    TRUE
```

## The R workspace

- The objects we have been making are created in the R workspace.
- When we load a package, we are loading that package's functions and data sets into our workspace.
- You can see what is in your workspace with **ls**:

```
> ls()
```

- You can attach data frames to your workspace and then refer to the variables directly:

```
> attach(patients)
> Full_Name
```

- You can remove objects from the workspace with **rm**:

```
> x<-1:5
[1] 1 2 3 4 5
> rm(x)
> x
Error: object 'x' not found
```

- You can remove everything by giving **rm** a list of all the objects returned by **ls**:

```
> rm(list=ls())
```

## The R workspace

- Your workspace is like an unsaved Word document.
- When you quit R, it will usually save your workspace to a hidden file called '.Rdata' in your current directory. This workspace will be loaded again if you open R in the same directory.
- This file is a binary, computer-readable file, not a human-readable file, which you have to open with R (like a Word document in Office).
- It is safer to explicitly save your workspace using **save.image**:

```
> save.image("phd.chapter.1.R")
```

- This way, if you are working on several different projects, you can make sure the objects for each project are saved to named files, rather than trying to remember which directory you were working in, or risking overwriting some objects you forgot about and need later.
- To load a particular image, use **load**:

```
> load("phd.chapter.1.R")
```

## Packages in the R workspace

- You can see which packages are loaded into your workspace with **search**:

```
> search()
 [1] ".GlobalEnv"        "tools:rstudio"     "package:stats"     "package:graphics"
 [5] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"
 [9] "Autoloads"         "package:base"
```

- **.GlobalEnv** is where all the objects you create are stored.
- Most of the core functions are in **stats**, **utils**, **methods** and **base**.
- We will cover **graphics** and **grDevices** tomorrow afternoon.
- **Search** shows the search path R runs through whenever you use an object or function name. It will first look in your global environment, then in the **Rstudio** tools (if using Rstudio), then in the **stats** package and so on.
- When loading packages, you will often see warnings about some objects or functions being 'masked'. This means that the newly loaded package contains an object with the same name as some object in a package that is already loaded. R will use the object in the new package whenever it comes across the name, because the new package will be earlier in the search path.

## Introducing loops

- Many programming languages have ways of doing the same thing many times, perhaps changing some variable each time. This is called **looping**.
- Loops are not used in R so often, because we can usually achieve the same thing using vector calculations.
- For example, to add two vectors together, we do not need to add each pair of elements one by one, we can just add the vectors.
- But there are some situations where R functions can not take vectors as input. For example, **read.csv** will only load one file at a time.
- What if we had ten files to load in, all ending in the same extension (like *.csv*)?

## Introducing loops

- We could do this:

```
> colony<-data.frame()    # Start with empty data frame

> colony1<-read.csv("11_CFA_Run1Counts.csv")
> colony2<-read.csv("11_CFA_Run2Counts.csv")
> colony3<-read.csv("11_CFA_Run3Counts.csv")
...
> colony10<-read.csv("11_CFA_Run10Counts.csv")

> colony<-rbind(colony1,colony2,colony3,...,colony10)
```

  But this will be boring to type, difficult to change, and prone to error.

- As we are doing the same thing 10 times, but with a different file name each time, we can use a **loop** instead.

## LOOPS
### Commands & flow control

- R has two basic types of loop:
  **for** loop: run some code on every value in a vector
  **while** loop: run some code while some condition is true
- Here are two simple examples of these loops:

```
for (f in 1:10) {
   print(f)
}

i <- 1
while ( i <= 10 ) {
   print(i)
   i <- i + 1
}
```

when this condition is false the loop stops

## LOOPS
### Commands & flow control

- Here's how we might use a **for** loop to load in our CSV files.
- If the data files are in your current working directory, we can look up files containing a particular substring in their name using the **dir** function:

```
dir(pattern="Counts.csv")
```

```
[1] "1.4_colony_Run1Counts.csv" "1.4_colony_Run2Counts.csv"
  "1.4_colony_Run3Counts.csv"
```

- So we can load all the files using a **for** loop as follows:

```
colony<-data.frame()
countfiles<-dir(pattern="Counts.csv")
for (file in countfiles) {
   t<-read.csv(file)
   colony<-rbind(colony,t)
}
```

- Here, we use a temporary variable **t** to store the data in each file, and then add that data to the main **colony** data frame.

## Conditional branching
### Commands & flow control

- Use an **if** statement for any kind of condition testing.
- Different outcomes can be selected based on a condition within brackets.

```
if (condition) {
… do this …
} else {
… do something else …
}
```

  - condition is any logical value, and can contain multiple conditions
    - e.g. `(a==2 & b <5)`, this is a compound conditional argument

## Conditional branching
### Commands & flow control

For example, if we were writing a script to load an unknown set of files, using the **for** loop we wrote before, we might want to warn the user if we can't find any files with the pattern we are searching for. Here's how we can use an **if** statement to test for this:

```
colony<-data.frame()
countfiles<-dir(pattern="Counts.csv")

if (length(countfiles) == 0) {
    stop("No Counts.csv files found!")
} else {
    for (file in countfiles) {
        t<-read.csv(file)
        colony<-rbind(colony,t)
    }
}
```

The **stop** function outputs the error message and quits.

## Code formatting avoids bugs!

- Code formatting is crucial for readability of loops

```
f<-26
while(f!=0){
print(letters[f])
f<-f-1 }
```

BAD!!!

```
f <- 26
while( f != 0 ){
   print(letters[f])
   f <- f-1
}
```

GOOD!

- The code between brackets {} **always** is indented, this clearly separates what is executed once, and what is run multiple times
- Trailing bracket } always alone on the line at the same indentation level as the initial bracket {
- Use white spaces to divide the horizontal space between units of your code, e.g. around assignments, comparisons

## Exercise

1. Output the patients data frame, with the patients sorted in order of age, oldest first. (You may need the **rev** function.)

2. Load in the **colony** data frame using a for loop. Three of the data files are in the *Day_1_scripts* folder. Load all three files into **colony** using the for loop in the slides.

3. How many observations do you have in the **colony** data frame? Find out by counting the number of rows in **colony** using the **nrow** function.

4. Suppose a power analysis of your data shows that you only need 48 observations to robustly test your hypothesis. This means we can stop loading files when we have loaded at least 48 observations. Modify your **for** loop so it will only load files if the **colony** data frame has less than 48 observations in it.

## Answers to exercise

1. To order the patients by decreasing age:

```
patients[rev(order(patients$Age)),]
```

3. To find the number of rows in the **colony** data frame:

```
nrow(colony)
```

4. To stop loading files after at least 48 observations have been found, use the code from the first **for** loop slide with a new **if** statement:

```
colony<-data.frame()
countfiles<-dir(pattern="Counts.csv")
for (file in countfiles) {
    if ( nrow(colony) < 48 ) {
        t<-read.csv(file)
        colony<-rbind(colony,t)
    }
}
```

Statistics

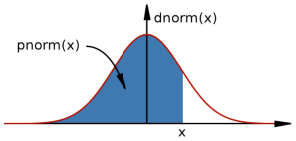
# 5

## Built-in support for statistics

- R is a statistical programming language
  - Classical statistical tests are built-in
  - Statistical modeling functions are built-in
  - Regression analysis is fully supported
  - Additional mathematical packages are available
    - MASS, Waves, sparse matrices, etc

## Distribution functions

- mostly commonly used distributions are built-in, functions have stereotypical names, e.g. for normal distribution:
  - pnorm - cumulative distribution for x
  - qnorm - inverse of pnorm (from probability gives x)
  - dnorm - distribution density
  - rnorm - random number from normal distribution




- available for variety of distributions: punif (uniform), pbinom (binomial), pnbinom (negative binomial), ppois (poisson), pgeom (geometric), phyper (hyper-geometric), pt (T distribution), pf (F distribution) ...

## Distribution functions

- 10 random values from the Normal distribution with mean 10 and standard deviation 5:

```
rnorm(10,mean=10,sd=5)
```

- The probability of drawing 10 from this distribution:

```
dnorm(10,mean=10,sd=5)
[1] 0.07978846
dnorm(100,mean=10,sd=5)
[1] 3.517499e-72
```

- The probability of drawing a value smaller than 10:

```
pnorm(10,mean=10,sd=5)
[1] 0.5
```

- The inverse of **pnorm**:

```
qnorm(0.5,mean=10,sd=5)
[1] 10
```

- How many standard deviations for statistical significance?

```
qnorm(0.95,mean=0,sd=1)
[1] 1.644854
```

## Two sample tests
### Basic data analysis

- Comparing 2 variances
  - Fisher's F test

`var.test()`

- Comparing 2 sample means with normal errors
  - Student's t test

`t.test()`

- Comparing 2 means with non-normal errors
  - Wilcoxon's rank test

`wilcox.test()`

- Comparing 2 proportions
  - Binomial test

`prop.test()`

- Correlating 2 variables
  - Pearson's / Spearman's rank correlation

`cor.test()`

- Testing for independence of 2 variables in a contingency table
  - Chi-squared

`chisq.test()`

  - Fisher's exact test

`fisher.test()`

## Comparison of 2 data sets example
### Basic data analysis

- Men, on average, are taller than women.
  - The steps
    1. Determine whether variances in each data series are different
       - Variance is a measure of sampling dispersion, a first estimate in determining the degree of difference
         - *Fisher's F test*
    2. Comparison of the mean heights.
       - Determine probability that mean heights really are drawn from different sample populations
         - *Student's t test, Wilcoxon's rank sum test*

## 1. Comparison of 2 data sets
### Fisher's F test

- Read in the data file into a new object, heightData

```
heightData<-read.csv("1.5_heightData.csv")
```

- **attach** the data frame so we don't have to refer to it by name all the time:

```
attach(heightData)
```

- Do the two sexes have the same variance?

```
var.test(Female,Male)
```

```
F test to compare two variances

data:  Female and Male
F = 1.0073, num df = 99, denom df = 99, p-value = 0.9714
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 0.6777266 1.4970241
sample estimates:
ratio of variances
           1.00726
```

## 2. Comparison of 2 data sets
### Student's t test

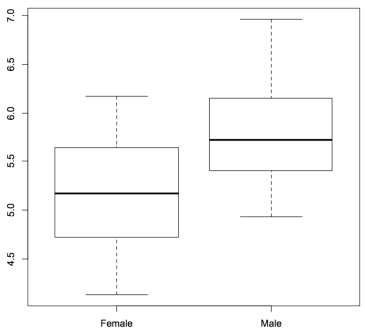
- Student's t test is appropriate for comparing the difference in mean height in our data. We need a one-tailed test.
  - Remember a t test = $\frac{\text{difference in two sample means}}{\text{standard error of the difference of the means}}$

```
t.test(Female,Male, alternative="less")
```

```
Welch Two Sample t-test

data:  Female and Male
t = -8.4508, df = 197.997, p-value = 3.109e-15
alternative hypothesis: true difference in means is less than 0
95 percent confidence interval:
       -Inf -0.5079986
sample estimates:
mean of x mean of y
 5.168725  5.800214
```

## 3. Comparison of 2 data sets
### Review findings

```
> boxplot(heightData)
```



## Linear regression
### Basic data analysis

- Linear modeling is supported by the function `lm()`
  - `example(lm)` # the output assumes you know a fair bit about the subject
- lm is really useful for plotting lines of best fit to XY data in order to determine intercept, gradient & Pearson's correlation coefficient
  - This is very easy in R
- Three steps to plotting with a best fit line
  - Plot XY scatter-plot data
  - Fit a linear model
  - Add bestfit line data to plot with abline() function

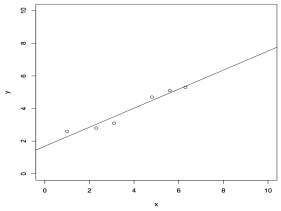
## Typical linear regression analysis
### Basic data analysis

| X | Y |
|---|---|
| 1.0 | 2.6 |
| 2.3 | 2.8 |
| 3.1 | 3.1 |
| 4.8 | 4.7 |
| 5.6 | 5.1 |
| 6.3 | 5.3 |

```
> x<-c(1, 2.3, 3.1, 4.8, 5.6, 6.3)
> y<-c(2.6, 2.8, 3.1, 4.7, 5.1, 5.3)
> plot(y~x, xlim=c(0,10),ylim=c(0,10))
```

Note formula notation
( y is given by x )

```
> myModel<-lm(y~x)
> abline(myModel)
```




Get the coefficients of the fit from:
`summary.lm(myModel)` **and**
`coef(myModel)`
`resid(myModel)`
`fitted(myModel)`

Get QC of fit from
`plot(myModel)`

Find out about the fit data from
`names(myModel)`

## Modelling formulae

- R has a very powerful formula syntax for describing statistical models.
- Suppose we had two explanatory variables **x** and **z** and one response variable **y**.
- We can describe a relationship between, say, **y** and **x** using a tilde **~**, placing the response variable on the left of the tilde and the explanatory variables on the right:

```
> y~x
```

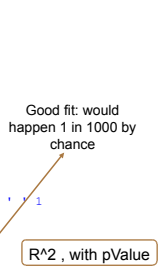
- It is very easy to extend this syntax to do multiple regressions, ANOVAs, to include interactions, and to do many other common modelling tasks. For example:

```
> y~x       # If x is continuous, this is linear regression
> y~x       # If x is categorical, this is ANOVA
> y~x+z     # If x and z are continuous, this is multiple regression
> y~x+z     # If x and z are categorical this is a two-way ANOVA
> y~x+z+x:z # : is the symbol for the interaction term
> y~x*z     # * is a shorthand for x+z+x:z
```

## The linear model object
### Basic data analysis

- Summary data describing the linear fit is given by
  - summary(myModel)

```
> summary(myModel)

Call:
lm(formula = y ~ x)

Residuals:
       1        2        3        4        5        6
 0.33159 -0.22785 -0.39520  0.21169  0.14434 -0.06458

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  1.68422    0.29056   5.796   0.0044 **
x            0.58418    0.06786   8.608   0.0010 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.3114 on 4 degrees of freedom
Multiple R-squared: 0.9488,   Adjusted R-squared: 0.936
F-statistic:  74.1 on 1 and 4 DF,  p-value: 0.001001
```




Good fit: would happen 1 in 1000 by chance

R^2 , with pValue

## Exercise
### The coin toss

To learn how the distribution functions work, try simulating tossing a fair coin 100 times and then show that it is fair.

1) We can model a coin toss using the binomial distribution. Use the **rbinom** function to generate a sample of 100 coin tosses. Look up the binomial distribution help page to find out what arguments this function needs.
2) How many heads or tails were there in your sample? You can do this in two ways; either select the number of successes using indices, or convert your sample to a factor and get a summary of the factor.
3) If we toss a coin 50 times, what is the probability that we get exactly 25 heads? What about 25 heads or less? Use **dbinom** and **pbinom** to find out.
4) The argument to **dbinom** is a vector, so try calculating the probabilities for getting any number of coin tosses from 0 to 50 in fifty trials using **dbinom**. Plot these probabilities using **plot**. Does this plot remind you of anything?

## Coin toss answers

- To simulate a coin toss, give **rbinom** a number of observations, the number of trials for each observation, and a probability of success:

```
> coin.toss<-rbinom(100, 1, 0.5)
```

- Because we only specified one trial per observation, we either have an outcome of 0 or 1 successes. To get the number of successes, use indices or a factor to look up the number of 1s in the **coin.toss** vector (your numbers will vary):

```
> length(coin.toss[coin.toss==1])
[1] 50
> summary(factor(coin.toss))
 0  1
50 50
```

## Coin toss answers

The probability of getting exactly 25 heads from 50 observations of a fair coin:

```
> dbinom(25, 50, 0.5)
```

The probability of getting 25 heads or less from 50 observations of a fair coin:

```
> pbinom(25, 50, 0.5)
```

The probabilities for getting all numbers of coin tosses from 0 to 50 in fifty trials:

```
> dbinom(0:50, 50, 0.5)
```

To plot this distribution, which should resemble a normal distribution:

```
> plot(dbinom(0:50, 50, 0.5))
```

## Exercise
### Linear modelling example

Mice have varying numbers of babies in each litter. Does the size of the litter affect the average brain weight of the offspring? We can use linear modelling to find out. (This example is taken from John Maindonald and John Braun's book *Data Analysis and Graphics Using R* (CUP, 2003), p140-143.)

1) Install and load the **DAAG** package. The **litters** data frame is part of this package. Take a look at it. How many variables and observations does it have? Does **summary** tell you anything useful? What about **plot**?
2) Are any of the variables correlated? Look up the **cor.test** function and use it to test for relationships.
3) Use **lm** to calculate the regression of brain weight on litter size, brain weight on body weight, and brain weight on litter size and body weight together.
4) Look at the coefficients in your models. How is brain weight related to litter size on its own? What about in the multiple regression? How would you interpret this result?

## Linear modelling answers

- To install and load the package and look at **litters**:

```
> install.packages("DAAG")
> library(DAAG)
> litters
> summary(litters)
> plot(litters)
```

- To calculate correlations between variables:

```
> attach(litters)
> cor.test(brainwt, lsize)
> cor.test(bodywt, lsize)
> cor.test(brainwt, bodywt)
```

## Linear modelling answers

- To calculate the linear models:

```
> lm(brainwt~lsize)

Call:
lm(formula = brainwt ~ lsize)

Coefficients:
(Intercept)        lsize
   0.447000    -0.004033
```

```
> lm(brainwt~bodywt)

Call:
lm(formula = brainwt ~ bodywt)

Coefficients:
(Intercept)       bodywt
    0.33555      0.01048
```

```
> lm(brainwt~lsize+bodywt)

Call:
lm(formula = brainwt ~ lsize + bodywt)

Coefficients:
(Intercept)        lsize       bodywt
    0.17825      0.00669      0.02431
```

Interpretation: brain weight decreases as litter size increases, but brain weight increases proportional to body weight (when bodywt is held constant, the lsize coefficient is positive: 0.00669). This is called 'brain sparing'; although the offspring get smaller as litter size increases, the brain does not shrink as much as the body.

End of Day 1

# 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

## 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

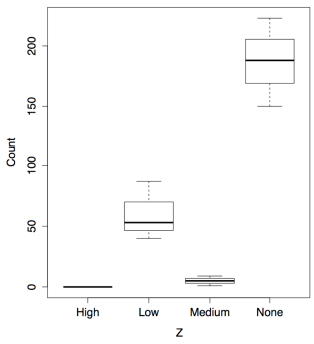
We want to plot the colony growth in response to changing Z concentration.

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)
```



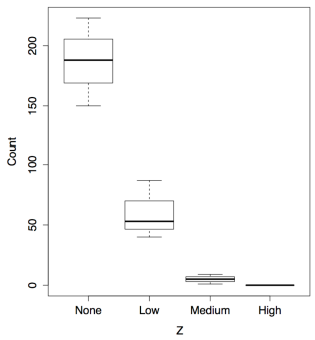
## Plotting

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:

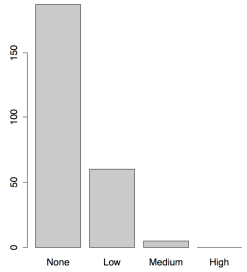
```
    data <- data.frame(gender=c("Male", "Male", "Female",
           "Female", "Female"), height=c(6, 6.1, 5.8, 6, 5.95))
gender height
1   Male   6.00
2   Male   6.10
3 Female   5.80
4 Female   6.00
5 Female   5.95
```

- 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?

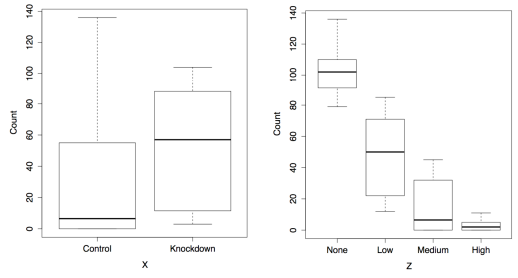
3. We need to include both X and Z in our call to **tapply**. Modify the call to **tapply** by changing the second argument, which should be a list containing the data for both X and Z.

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:

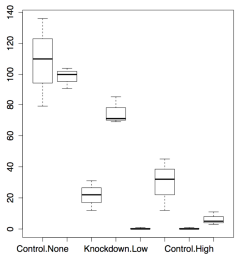
```
> plot(Count~X*Z,colony)
```



## Plotting interactions

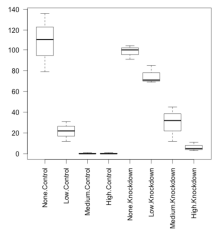
To get a sense of what's happening with the interactions, use **boxplot**:

```
> 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:

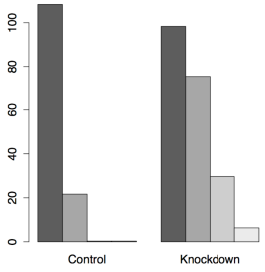
```
> colony.aov<-aov(Count~X*Z,colony)
> print(summary(colony.aov))
            Df Sum Sq Mean Sq F value   Pr(>F)
X            1   2321    2321  14.072  0.00174 **
Z            3  36150   12050  73.067 1.48e-09 ***
X:Z          3   3441    1147   6.954  0.00329 **
Residuals   16   2639     165
---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

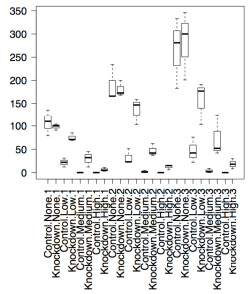
```
> colony.aov<-aov(Count~X*Z*Run,colony)
> print(summary(colony.aov))
             Df Sum Sq Mean Sq F value   Pr(>F)
X             1  24939   24939  32.454 4.71e-07 ***
Z             3 355524  118508 154.221  < 2e-16 ***
Run           1  48197   48197  62.721 1.05e-10 ***
X:Z           3  21967    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
> powXplusX(x=10, power=3)
[1] 1010
```

arguments matched based on **position**

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
```


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

```
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)
```

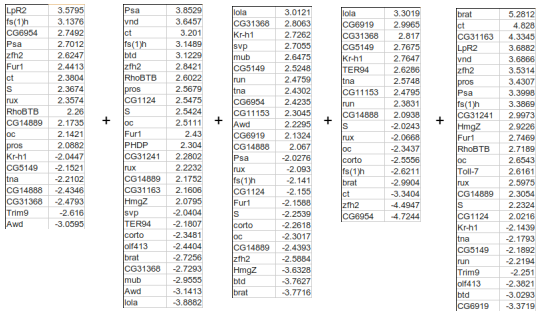
Advanced data processing

## 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 | -2.1607 |
| 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 | 2.3631 |
| 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

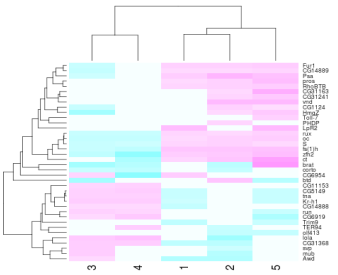
we use index to pick the rows in such way that they match the gene order in the input file

## 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**

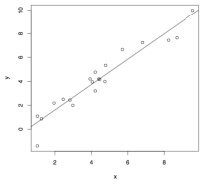
## Starting out with R graphics
### Graphics

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.

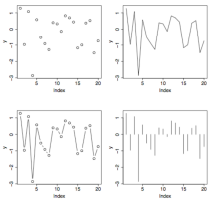
```
> plot(litters)
> plot(1:10)
> plot(y~x)
> plot(x, y)
```

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

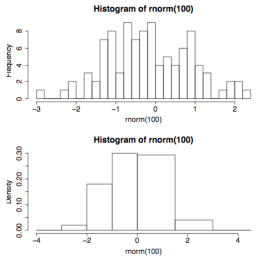
## High level graphics functions

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))
```

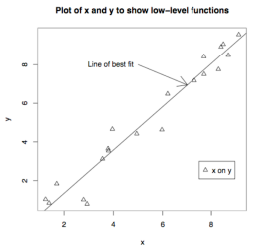



## Low level graphics functions

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

```
> 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

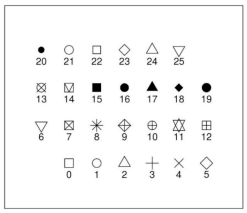
R's traditional graphics uses a *painter* model – once something is drawn on a device, it cannot be erased. If you make a mistake with a low level function, you will have to start drawing the plot again from scratch.

```
> 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:

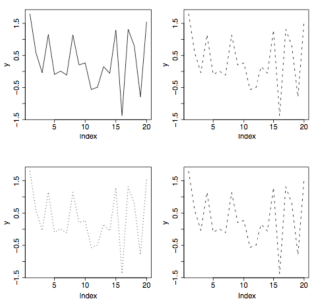
```
> par()
```

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*.

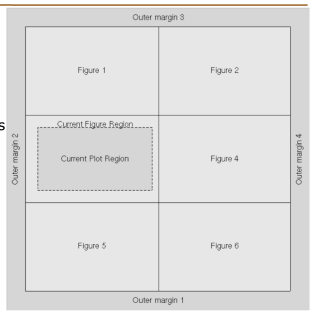
The inner region is split into one or more *figure regions*, which typically contains a *plot region*. The plot region contains data points, but not titles, axes and so on.

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

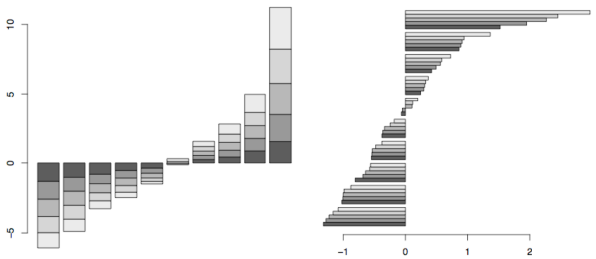
We already used the **barplot** function this morning, and tried out its **beside** argument. Let's explore this function in more detail. Generate a random matrix and plot it with **barplot**:

```
> 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.

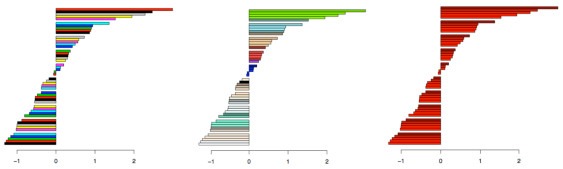
## Bar plot exercise: positions

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



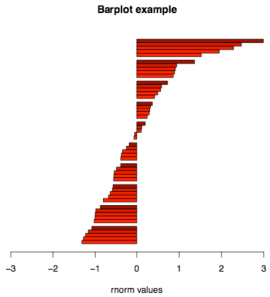
## Bar plot exercise: colours

```
> 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

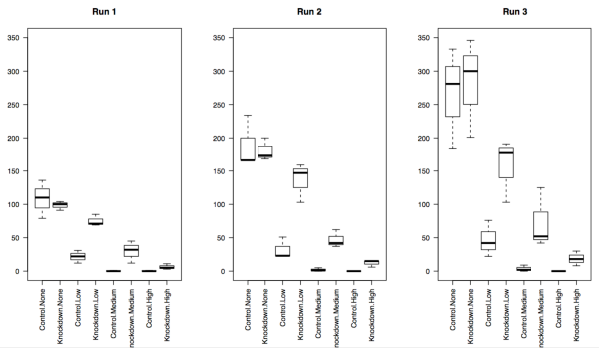
There is a **pdf()** function to create a PDF file as an R graphics device. This function (in common with most graphics device functions) has width and height options. For **pdf**, the default units of these options are inches, but other devices differ (for example, **png** and **jpeg** default to pixels).

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.

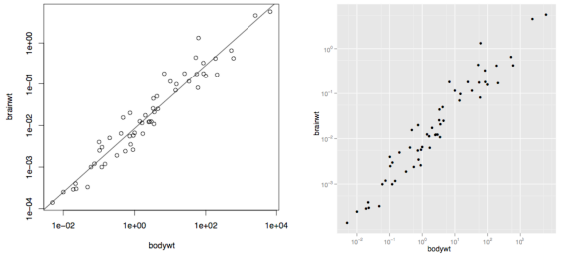
Load the **ggplot** library. We'll use one of the R built-in datasets as a demonstration.

```
> 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:

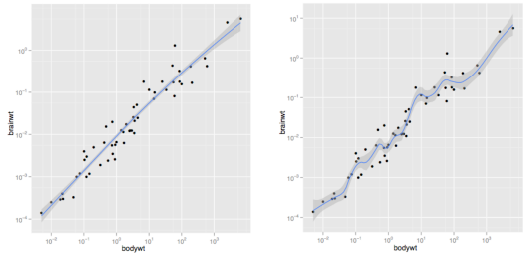
```
> plot(bodywt, brainwt, log = "xy")
> abline(lm(log10(brainwt) ~ log10(bodywt)))
> qplot(bodywt, brainwt, log = "xy")
```



# 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

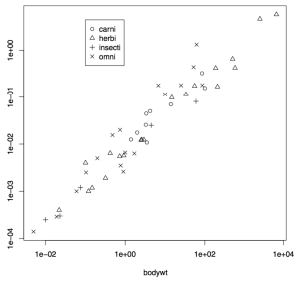
When we make plots, we want to focus on our data, and not be distracted by the drawing commands. The traditional graphics functions can be very distracting:

```
> 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.

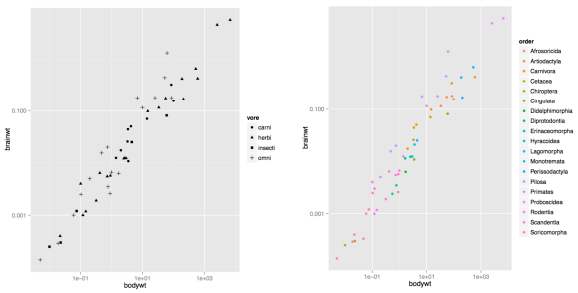
Note there is no connection between the call to **plot** and the call to **legend** – the inputs to **legend** are completely separate, which means we have to do fiddly, unnecessary work twice.




# 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