

---

# A beginners guide to solving biological problems in R

Slides by: 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. Writing functions

---

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



## The R Project for Statistical Computing

### About R

[What is R?](#)  
[Contributors](#)  
[Screenshots](#)  
[What's new?](#)

### Download, Packages

[CRAN](#)

### R Project

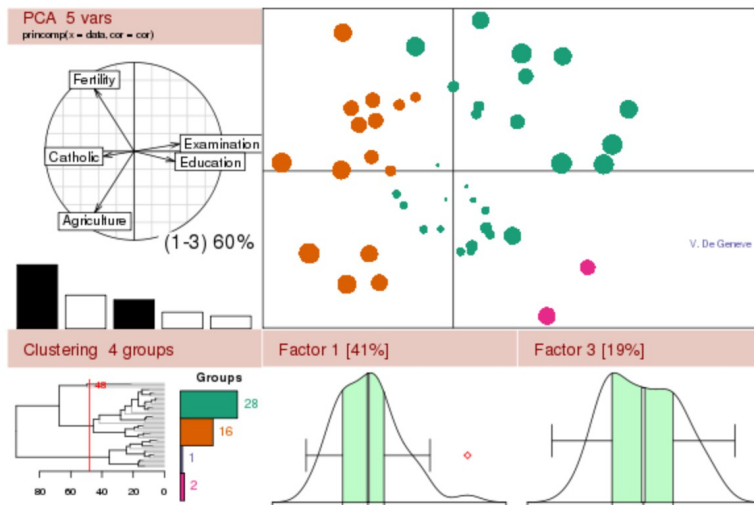
[Foundation](#)  
[Members & Donors](#)  
[Mailing Lists](#)  
[Bug Tracking](#)  
[Developer Page](#)  
[Conferences](#)  
[Search](#)

### Documentation

[Manuals](#)  
[FAQs](#)  
[The R Journal](#)  
[Wiki](#)  
[Books](#)  
[Certification](#)  
[Other](#)

### Misc

[Bioconductor](#)  
[Related Projects](#)  
[User Groups](#)  
[Links](#)



### Getting Started:

- R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred [CRAN mirror](#).
- If you have questions about R like how to download and install the software, or what the license terms are, please read our [answers to frequently asked questions](#) before you send an email.

### News:

- [The R Journal Vol.5/1](#) is available.
- **R version 3.0.1** (Good Sport) has been released on 2013-05-16.
- **R version 2.15.3** (Security Blanket) has been released on 2013-03-01.
- [useR! 2013](#), will take place at the University of Castilla-La Mancha, Albacete, Spain, July 10-12 2013..

This server is hosted by the [Institute for Statistics and Mathematics](#) of [WU \(Wirtschaftsuniversität Wien\)](#).

# 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

# Prepare to launch R

## From command line

---

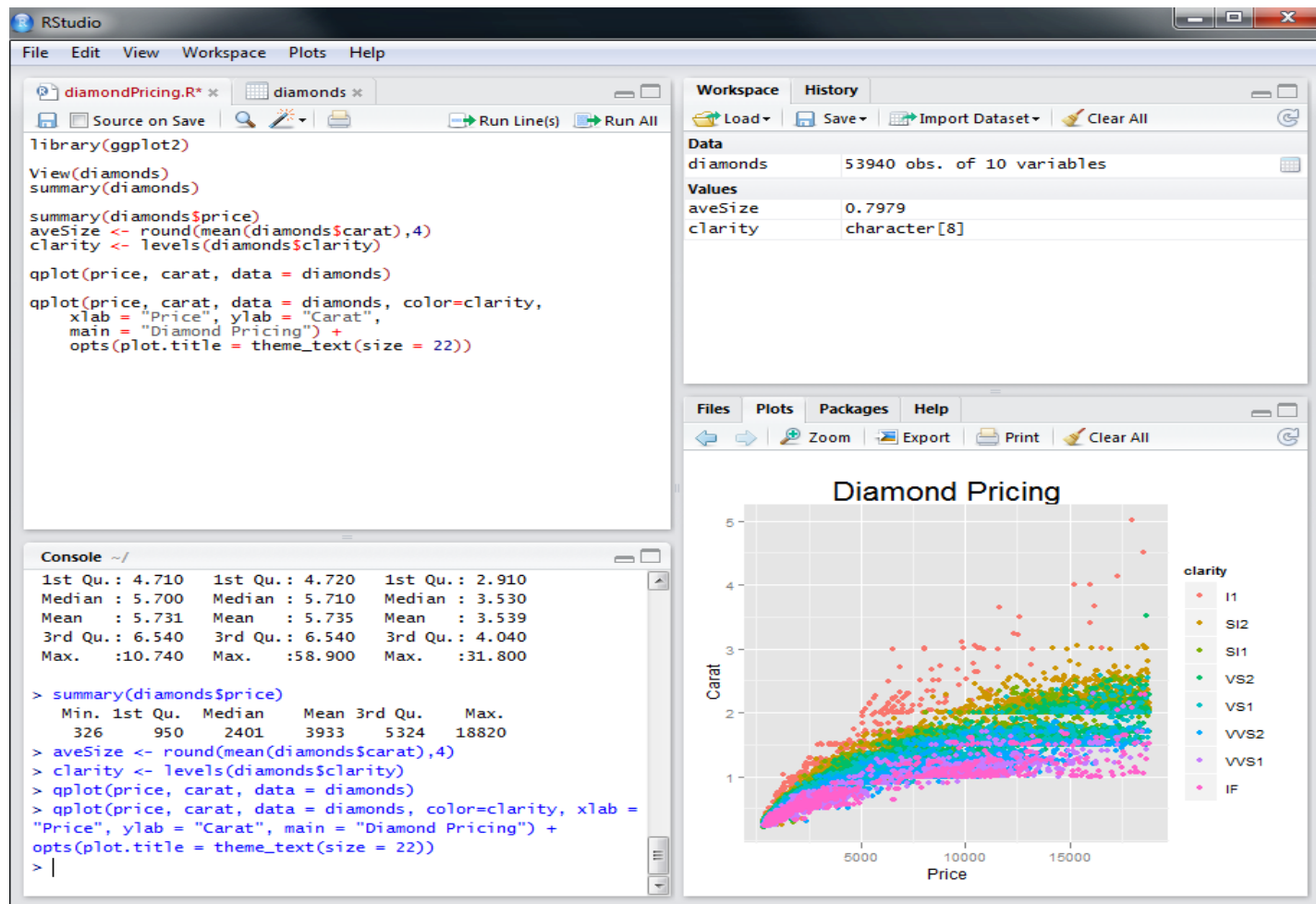
- To start R in Linux we need to enter the Linux console (also called Linux terminal and Linux shell)
- To start R, at the prompt simply type:

**\$ R**

# Prepare to launch R

## Using RStudio

- To launch RStudio, find the RStudio icon 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/Practical_1_scripts")
```

- Alternatively in RStudio use the mouse and browse to the directory location
- Tools → 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
<i>Homo sapiens</i>	3,102	20,774
<i>Mus musculus</i>	2,731	23,139
<i>Drosophila melanogaster</i>	169	13,937
<i>Caenorhabditis elegans</i>	100	20,532
<i>Saccharomyces cerevisiae</i>	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

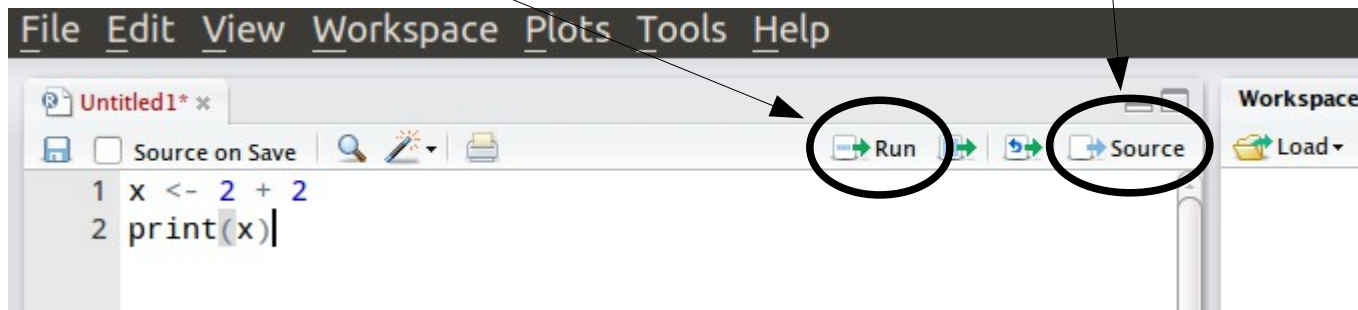
---

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  
`> ??rint`

# 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-I** 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

---

- 4700+ 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/>
- 670+ packages in Bioconductor:
  - Specialised in genomics:  
<http://www.bioconductor.org/packages/release/bioc/>
- **Other repositories:**
- 1600+ 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

---

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     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    Male    Female  Male    Male    Female
Levels: Female Male

> patients$First_Name
[1] "Adam"    "Eve"      "John"     "Mary"     "Peter"    "Paul"    "Joanna"
"Matthew" "David"    "Sally"
```

# Indexing data frames

Special cases:

a[*i*, ] i-th row

a[ ,*j*] j-th column

- You can index multidimensional data structures like 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 ]
```

```
> 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

---

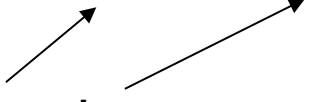
- arithmetic

+, -, \*, /, ^

- comparison

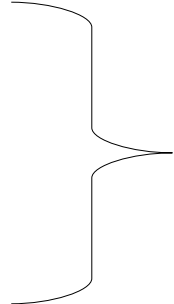
<, >, <=, >=, ==, !=

(equal to, not equal to)



- logical

!, &, |, xor



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("05_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 weight 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	42	0	34	8
2	40	3	30	7
3	56	6	50	0
4	42	5	37	0
5	32	1	30	1
6	70	10	53	7
7	65	3	58	4
8	40	4	31	5
9	60	0	54	6
10	61	0	57	4
11	43	13	29	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("08_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("08_NBcountData.csv")
?read.table for a full list of arguments
```

08\_NBcountData.R  
(script commands)

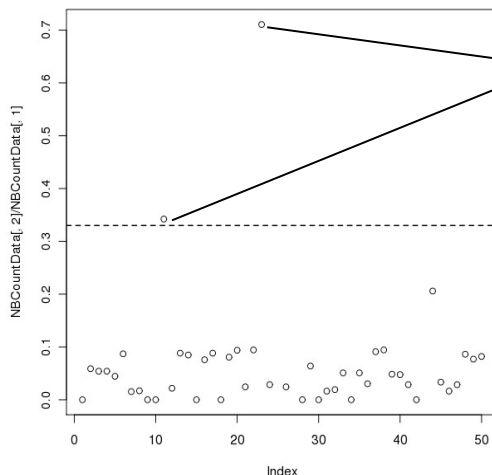
08\_NBcountData.txt  
(data file)

## 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(0,1.2))`
  - `abline(h=0.33,lwd=1.5,lty=2)`



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

# 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")
```