

Introduction to Solving Biological Problems Using R - Day 1

Mark Dunning, Suraj Menon, and Aiora Zabala. Original material by Robert Stojnić, Laurent Gatto, Rob Foy John Davey, Dávid Molnár and Ian Roberts

Last modified: 06 Oct 2015

true

Course Aims

- To introduce you to the basics of R
 - Reading data
 - Perform simple analyses
 - Producing graphs
 - **How to get help!**
- Give you all the background you need to **practice** by yourselves
- Introduce tools that will help you to work in a **reproducible** manner

Day 1 Schedule

1. Introduction to R and its environment
2. Data Structures
3. Data Analysis Example
4. Plotting in R

1. Introduction to R and its environment

What's R?

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

The R-project page

<http://www.r-project.org/> (<http://www.r-project.org/>)


[\[Home\]](#)
[Download](#)
[CRAN](#)
[R Project](#)
[About R](#)
[Contributors](#)
[What's New?](#)
[Mailing Lists](#)
[Bug Tracking](#)
[Conferences](#)
[Search](#)
[R Foundation](#)
[Foundation](#)
[Board](#)
[Members](#)
[Donors](#)
[Donate](#)
[Documentation](#)
[Manuals](#)
[FAQs](#)
[The R Journal](#)
[Books](#)
[Certification](#)
[Other](#)
[Links](#)
[Bioconductor](#)
[Related Projects](#)

The R Project for Statistical Computing

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

- **R 3.2.1 (World-Famous Astronaut) prerelease versions** will appear starting June 8. Final release is scheduled for 2015-06-18.
- **R version 3.2.0** (Full of Ingredients) has been released on 2015-04-16.
- **R version 3.1.3** (Smooth Sidewalk) has been released on 2015-03-09.
- **The R Journal Volume 6/2** is available.
- **useR! 2015**, will take place at the University of Aalborg, Denmark, June 30 - July 3, 2015.
- **useR! 2014**, took place at the University of California, Los Angeles, USA June 30 - July 3, 2014.

R in the New York Times

<http://goo.gl/pww4ZO> (<http://goo.gl/pww4ZO>)

The New York Times

Business Computing

WORLD
U.S.
N.Y. / REGION
BUSINESS
TECHNOLOGY
SCIENCE
HEALTH
SPORTS
OPINION

Search Technology
Go

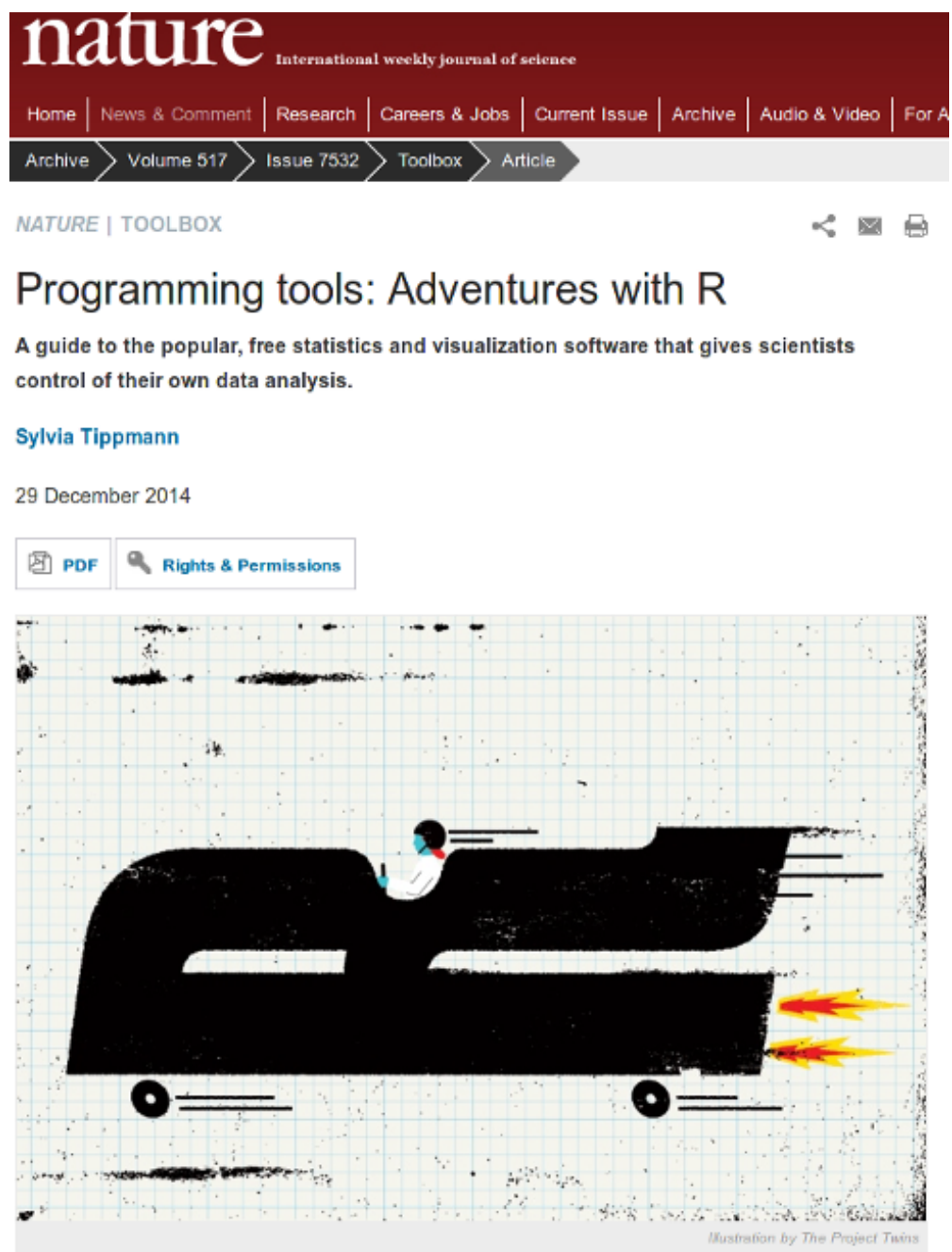
Inside Technology
[Internet](#)
[Start-Ups](#)
[Business Computing](#)
[Companies](#)

Data Analysts Captivated by R's Power

Stuart Issett for The New York Times

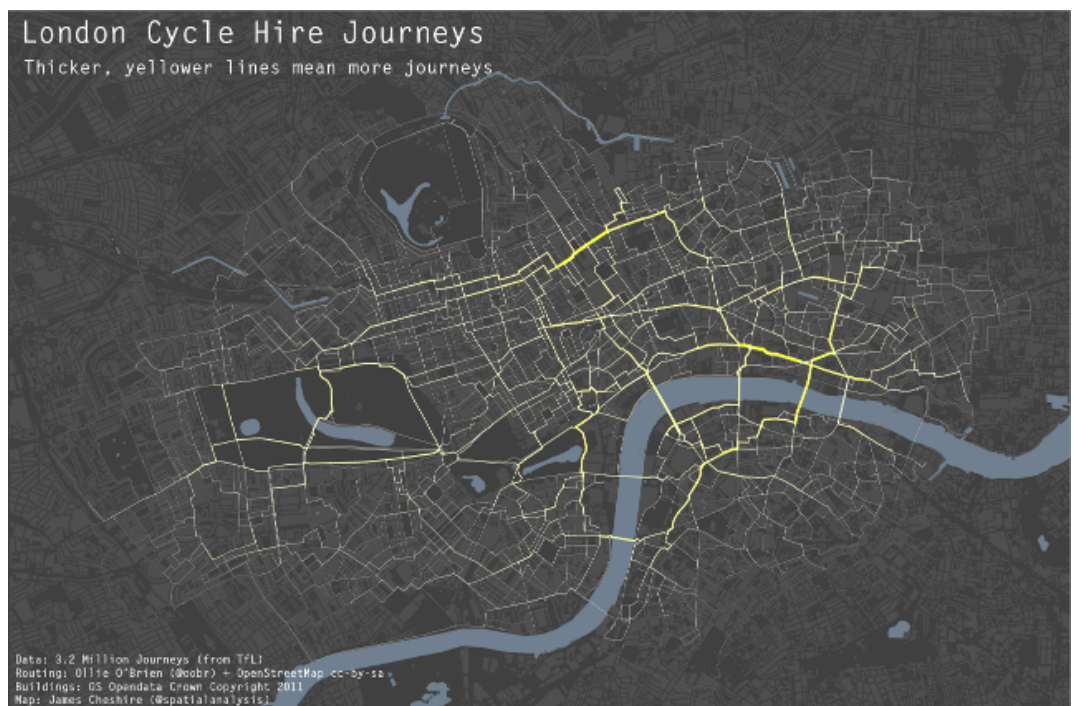
R first appeared in 1996, when the statistics professors Robert Gentleman, left, and Ross Ihaka released the code as a free software package.

R in Nature



R plotting capabilities

<http://spatial.ly/2012/02/great-maps-ggplot2/> (<http://spatial.ly/2012/02/great-maps-ggplot2/>)



R plotting capabilities

<https://www.facebook.com/notes/facebook-engineering/visualizing-friendships/469716398919> (<https://www.facebook.com/notes/facebook-engineering/visualizing-friendships/469716398919>)



Who uses R? Not just academics!

<http://www.revolutionanalytics.com/companies-using-r>
(<http://www.revolutionanalytics.com/companies-using-r>)


- Facebook
 - <http://blog.revolutionanalytics.com/2010/12/analysis-of-facebook-status-updates.html> (<http://blog.revolutionanalytics.com/2010/12/analysis-of-facebook-status-updates.html>)
- Google
 - <http://blog.revolutionanalytics.com/2009/05/google-using-r-to-analyze-effectiveness-of-tv-ads.html>
(<http://blog.revolutionanalytics.com/2009/05/google-using-r-to-analyze-effectiveness-of-tv-ads.html>)

- Microsoft
 - <http://blog.revolutionanalytics.com/2014/05/microsoft-uses-r-for-xbox-matchmaking.html> (<http://blog.revolutionanalytics.com/2014/05/microsoft-uses-r-for-xbox-matchmaking.html>)
- New York Times
 - <http://blog.revolutionanalytics.com/2011/03/how-the-new-york-times-uses-r-for-data-visualization.html> (<http://blog.revolutionanalytics.com/2011/03/how-the-new-york-times-uses-r-for-data-visualization.html>)

Various platforms supported

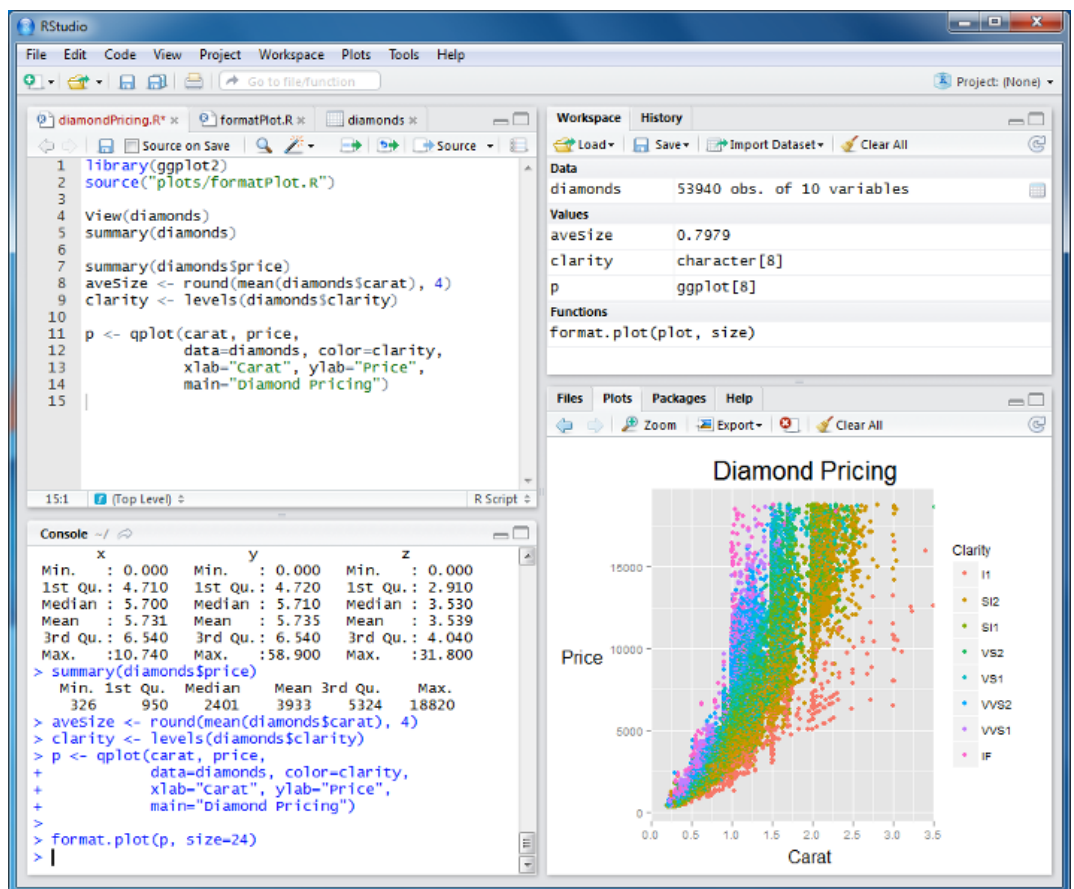
- Release 3.2.0 (April 2015)
 - Base package and Contributed packages (general purpose extras)
 - 7284 available packages as of Tue Oct 6 09:04:20 2015
- Download from <http://mirrors.ebi.ac.uk/CRAN/> (<http://mirrors.ebi.ac.uk/CRAN/>)
- Windows, Mac and Linux versions available
- Executed using command 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:
 - From the command line (particularly useful if you're quite familiar with Linux; in the console at the prompt simply type `R`)
 - As an application called  (very good for beginners)

Launching R Using RStudio

To launch RStudio, find the RStudio icon in the menu bar on the left of the screen and 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)  
myNumber
```

```
## [1] 29
```

Basic concepts in R - functions

- **Functions** in R perform operations on **arguments** (the inputs(s) to the function). We have already used:

```
sin(x)
```

this returns the sine of x . In this case the function has one argument: x . Arguments are always contained in parentheses – 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
```

Basic concepts in R - functions

- 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)
 - when testing code, it is easier and safer to name the arguments

```
seq(from = 2, to = 20, by = 4)
```

```
## [1] 2 6 10 14 18
```



```
seq(2, 20, 4)
```

```
## [1]  2  6 10 14 18
```

Basic concepts in R - vectors

- The basic data structure in R is a **vector** – an ordered collection of values.
- R treats even 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
```

- The square brackets `[]` indicate the 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
x
```

```
## [1] 3 4 5 6 7 8 9 10 11 12
```

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

```
x <- seq(2, 20, 4)
x
```

```
## [1]  2  6 10 14 18
```

```
x <- seq(2, 20, length.out=5)
x
```

```
## [1]  2.0  6.5 11.0 15.5 20.0
```

Basic concepts in R - vectors

- or we can use 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
```

Basic concepts in R - vectors

- 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
## [19] 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
```

Basic concepts in R - vectors

- 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
```

Basic concepts in R - vector arithmetic

- 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
```

```
## Warning in x + 1:3: longer object length is not a
## multiple of shorter object length
```

```
## [1] 2 4 6 5 7 9 8 10 12 11
```

Basic concepts in R - Character vectors and naming

- All the vectors we have seen so far have contained numbers, but we can also store text (“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"
## [4] "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

```
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
##      31.1610      34.7085      20.9055
##      C. elegans      S. cerevisiae
##      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
##      1.004545      1.270908      12.370118
##      C. elegans      S. cerevisiae
##      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
## [5] 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

- **This is possibly the most important slide in the whole course!?!**
- 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 possibilities:

```
??plot
```

- The **Packages** tab in the lower-right panel of RStudio will help you to locate the help pages for a particular package and its functions
 - Often there will be a user-guide / *'vignette'* too

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
- 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)**
 - Use metacran search to find functionality you need: <http://www.r-pkg.org/> (<http://www.r-pkg.org/>)
 - Or look for packages by theme: <http://cran.r-project.org/web/views/> (<http://cran.r-project.org/web/views/>)
 - **Bioconductor** specialised in genomics:
<http://www.bioconductor.org/packages/release/bioc/>
(<http://www.bioconductor.org/packages/release/bioc/>)
- Other repositories:
 - <http://r-forge.r-project.org/> (<http://r-forge.r-project.org/>)
 - github.com can also host R packages
- Bottomline: **always** first look if there is already an R package that does what you want before trying to implement it yourself

Installing packages

- CRAN packages can be installed using `install.packages`
 - or clicking on the Packages tab in RStudio

```
install.packages(name.of.my.package)
```

- Set the Bioconductor package download tool by typing:

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

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

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

Exercise: Install packages ggplot2 and DESeq

- ggplot2 is a commonly used graphics package
 - in RStudio, go to **Tools** → **Install Packages...** and type the package name
 - or use `install.packages()` function to install it:

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

- DESeq is a Bioconductor package (<http://www.bioconductor.org> (<http://www.bioconductor.org>)) for the analysis of RNA-seq data

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

2. Data structures

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.
- We will explore how to construct these data from scratch.
 - (in practice, we would usually import such data from a file)

	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

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 **class** function:

```
class(firstName)
```

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

```
class(age)
```

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

```
class(weight)
```

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

```
class(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"  
## [7] "Female" "Male" "Male" "Female"
```

```
factor(sex)
```

```
## [1] Male Female Male Female Male Male Female Male  
## [9] 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 (N.B. The **paste** function joins character vectors together)

```
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
```

Naming data frame variables

- We can access particular variables using the ‘ \$ ’ operator:

```
patients$age
```

```
## [1] 50 21 35 45 28 31 42 33 57 62
```

- R has inferred the names of our data frame variables from the names of the vectors or the commands (e.g. 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"
## [3] "Full_Name"  "Sex"
## [5] "Age"        "Weight"
## [7] "Consent"
```

Naming data frame variables

- 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"
## [3] "Full_Name"  "Sex"
## [5] "Age"        "Weight"
## [7] "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:
 - e.g. we are unlikely to be interested in the hypothesis that people called Adam are taller, so it seems a bit silly to represent this as a factor

```
patients$First_Name
```

```
## [1] Adam   Eve    John   Mary
## [5] Peter  Paul   Joanna Matthew
## [9] David  Sally
## 10 Levels: Adam David Eve ... Sally
```

Factors in data frames

- 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
## [6] Male   Female Male   Male   Female
## Levels: Female Male
```

```
patients$First_Name
```

```
## [1] "Adam"   "Eve"    "John"
## [4] "Mary"   "Peter"   "Paul"
## [7] "Joanna" "Matthew" "David"
## [10] "Sally"
```

Matrices

- Data frames are R's speciality, but R also handles matrices:
 - all columns are assumed to contain the same data type. e.g. numerical
 - matrices can be manipulated in the same fashion as data frame
 - we can easily convert between the two object types

```
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
```

- some calculations are more efficient to do on matrices. e.g.

```
rowMeans(e)
```

```
## [1] 3.5 4.5 5.5 6.5 7.5
```

Indexing data frames and matrices

- 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
## 1      Adam      Jones Adam Jones
##   Sex Age Weight Consent
## 1 Male  50   70.8    TRUE
```

Advanced indexing

- ‘Values’ in R are really vectors
- 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"
```

Advanced indexing

- We can do the logical test and indexing in the same line of R code
 - R will do the test first, and then use the vector of `TRUE` and `FALSE` values to subset the vector

```
a <- 1:5
a < 3
```

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

```
s[a < 3]
```

```
## [1] "a" "b"
```

Operators

- Operators allow us to combine multiple logical tests

- comparison operators `<`, `>`, `<=`, `>=`, `==`, `!=`
- logical operators `!`, `&`, `|`, `xor`
 - The operators for 'comparison' and 'logical' always return logical values! i.e. (`TRUE` , `FALSE`)

```
s
```

```
## [1] "a" "b" "c" "d" "e"
```

```
a
```

```
## [1] 1 2 3 4 5
```

```
s[a > 1 & a <3]
```

```
## [1] "b"
```

```
s[a == 2]
```

```
## [1] "b"
```

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,]
```

3. R for data analysis

3 steps to Basic Data Analysis

- In this short section, we show how the data manipulation steps we have just seen can be used as part of an analysis pipeline
1. Reading in data
 - `read.table()`
 - `read.csv()`, `read.delim()`
 2. Analysis
 - Manipulating & reshaping the data
 - Any maths you like
 - Plotting the outcome
 3. Writing out results
 - `write.table()`
 - `write.csv()`

A simple walkthrough

- 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)

0. Locate the data

Before we even start the analysis, we need to be sure of where the data are located on our hard drive

- Functions that import data need a file location as a character vector
- The default location is the **working directory**

```
getwd()
```

- If the file you want to read is in your working directory, you can just use the file name

```
list.files()
```

- Otherwise you need the *path* to the file
 - you can get this using `file.choose`

1. Read in the data

- The data is a tab-delimited 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 and assign it to an object (`rawdata`)

```
rawData <- read.delim("1.3_NBcountData.txt")
```

- Using `file.choose()`

```
myfile <- file.choose()
rawData <- read.delim(myfile)
```

- If the data has been comma-separated then, `sep=","` or use `read.csv` :

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

- For full list of arguments

```
?read.table
```

1b. Check the data

Always check the object to make sure the contents and dimensions are as you expect

- R will sometimes create the object without error, but the contents may be un-usable for analysis
 - if you specify an incorrect separator, R will not be able to locate the columns in your data, and you may end up with an object with just one column

```
rawData[1:10,] # View the first 10 rows to ensure import is OK
```

```
## Patient Nuclei NB_Amp NB_Nor NB_Del
## 1      1      65      0      63      2
## 2      2      51      3      43      5
## 3      3      37      2      35      0
## 4      4      37      2      35      0
## 5      5      45      2      42      1
## 6      6      46      4      41      1
## 7      7      65      1      64      0
## 8      8      59      1      54      4
## 9      9      49      0      48      1
## 10     10     46      0      45      1
```

- or use the `View` function to get a display of the data in RStudio


```
View(rawData)
```

1c. Understanding the object

Once we have read the data successfully, we can start to interact with it

- The object we have created is a data frame

```
class(rawData)
```

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

- we can query the dimensions

```
ncol(rawData)
```

```
## [1] 5
```

```
nrow(rawData)
```

```
## [1] 50
```

```
dim(rawData)
```

```
## [1] 50 5
```

1c. Understanding the object

- The names of the columns are automatically assigned

```
colnames(rawData)
```

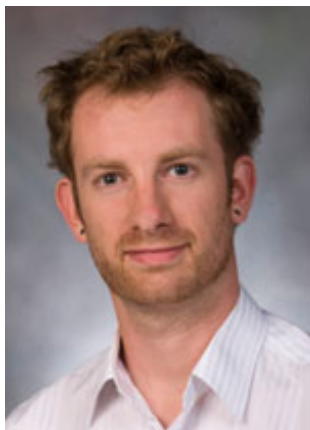
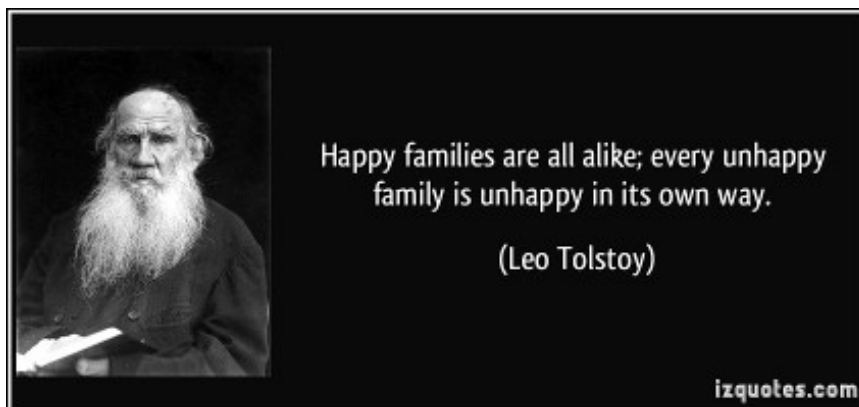
```
## [1] "Patient" "Nuclei"  "NB_Amp"  
## [4] "NB_Nor"  "NB_Del"
```

- We can use any of these names to access a particular column
 - and create a vector
 - TOP TIP: type the name of the object and hit TAB. You can select the column from the drop-down list!

```
rawData$Nuclei
```

```
## [1] 65 51 37 37 45 46 65 59 49 46 38 46
## [13] 34 59 52 66 68 49 62 64 41 53 38 35
## [25] 45 41 49 53 47 38 61 52 59 69 59 33
## [37] 44 53 62 63 70 30 39 34 60 61 70 58
## [49] 39 61
```

Word of caution



Like families, tidy datasets are all alike but every messy dataset is messy in its own way - (Hadley Wickham)

You will make your life a lot easier if you keep your data **tidy**

<http://vimeo.com/33727555> (<http://vimeo.com/33727555>)

<http://vita.had.co.nz/papers/tidy-data.pdf> (<http://vita.had.co.nz/papers/tidy-data.pdf>)

and **organised**

<http://kbroman.org/dataorg/> (<http://kbroman.org/dataorg/>)

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 NA s gracefully

```
x <- c(1,NA,3)
length(x)
```

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

Handling missing values

- However, sometimes we have to tell the functions what to do with them.
- R has some built-in functions for dealing with NA s, and functions often have their own arguments (like na.rm) for handling them

```
mean(x, na.rm=TRUE)
```

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

```
mean(na.omit(x))
```

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

2. Analysis (reshaping data and maths)

- Our analysis involves identifying patients with > 33% NB amplification
 - we can use the which function to select indices from a logical vector that are TRUE

```
# create an index of results:
prop <- rawData$NB_Amp / rawData$Nuclei
prop > 0.33
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
## [7] FALSE FALSE FALSE FALSE TRUE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE
## [19] FALSE FALSE FALSE FALSE TRUE FALSE
## [25] NA FALSE NA FALSE FALSE FALSE
## [31] FALSE FALSE FALSE FALSE FALSE FALSE
## [37] FALSE FALSE FALSE FALSE FALSE FALSE
## [43] NA FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE
```

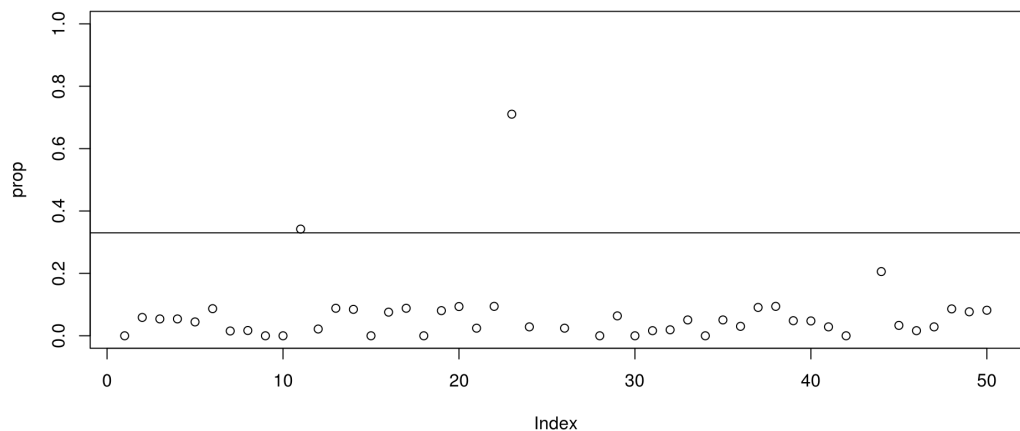
```
# Get sample names of amplified patients:
amp <- which(prop > 0.33)
amp
```

```
## [1] 11 23
```

2. Analysis (reshaping data and maths)

- We can plot a simple chart of the % NB amplification
 - note that two samples are amplified
 - plotting will be covered in detail shortly

```
plot(prop, ylim=c(0,1))
abline(h=0.33) # Add a horizontal line
```



3. Outputting the results

- We write out a data frame of results (patients > 33% NB amplification) as a 'comma separated values' text file (CSV):

```
write.csv(rawData[amp,], file="selectedSamples.csv")
```

- The output file is directly-readable by Excel
- It's often helpful to double check where the data has been saved.
 - Use the *get working directory* function:

```
getwd() # print working directory
list.files() # list files in 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

```
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")
```

4. Plotting in R

Plot basics

- As we have heard, R has extensive graphical capabilities
- but we need to start simple
- we will describe *base* graphics in R; the plots available with any standard R installation
 - other alternatives are available but require more R knowledge; e.g. `lattice`, `ggplot2`
- plotting in R is a *vast* topic
 - we cannot cover everything
 - you can tinker with plots to your hearts content
 - best to learn from examples
- ***You need to think about how best to visualise your data***
 - <http://www.bioinformatics.babraham.ac.uk/training.html#figuredesign>
(<http://www.bioinformatics.babraham.ac.uk/training.html#figuredesign>)
- R cannot prevent you from creating a plotting disaster
 - <http://www.businessinsider.com/the-27-worst-charts-of-all-time-2013-6?op=1&IR=T> (<http://www.businessinsider.com/the-27-worst-charts-of-all-time-2013-6?op=1&IR=T>)

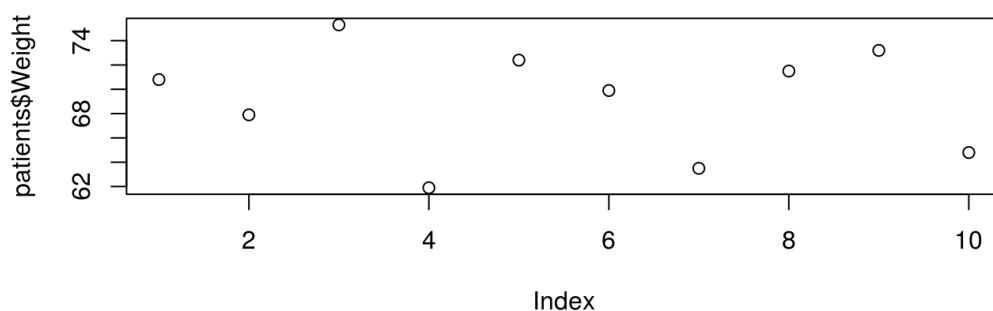
Making a Scatter Plot

- If given a single vector as an argument, `plot` will make a scatter plot with the *values* of the vector on the *y* axis, and *indices* in the *x* axis
 - e.g. it puts a point at
 - $x=1, y=70.8$
 - $x=2, y=67.9$ etc...

```
patients$Weight
```

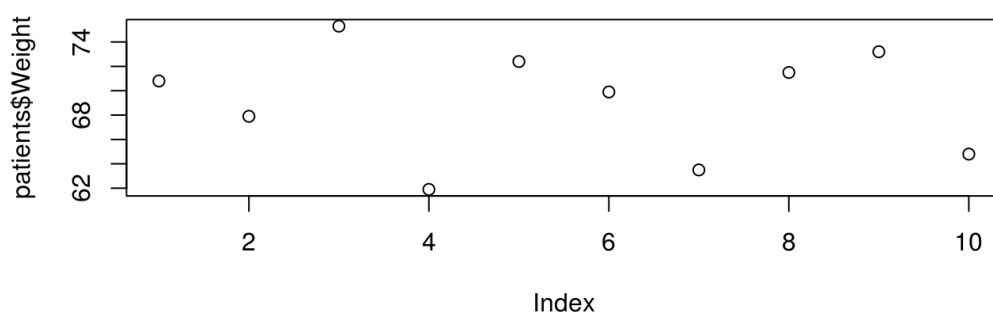
```
## [1] 70.8 67.9 75.3 61.9 72.4 69.9 63.5
## [8] 71.5 73.2 64.8
```

```
plot(patients$Weight)
```



Making a Scatter Plot

R tries to guess the most appropriate way to visualise the data



- Axis limits, labels, titles are inferred from the data
 - we can modify these as we wish

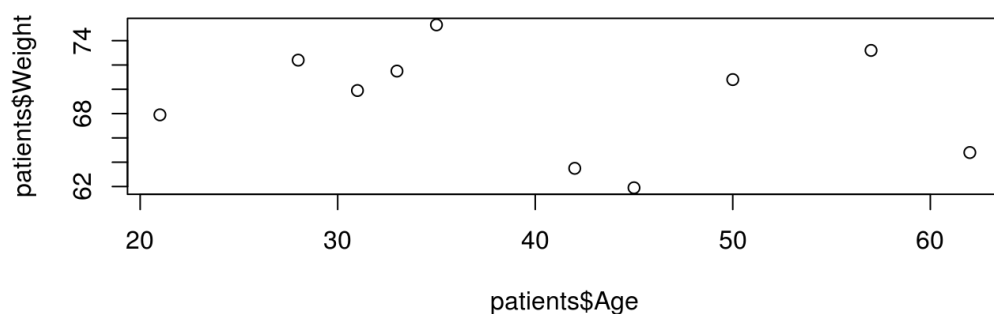
Making a Scatter Plot of two variables

- We can give two arguments to `plot`. It will put the values from the *first* argument in the *x* axis, and values from the *second* argument on the *y* axis
 - for when we want to visualise the relationship between two variables

```
patients$Age
```

```
## [1] 50 21 35 45 28 31 42 33 57 62
```

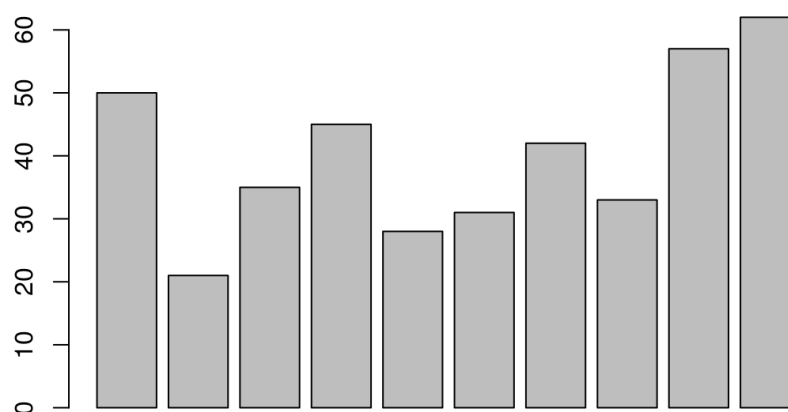
```
plot(patients$Age,patients$Weight)
```



Making a barplot

- Other types of visualisation are available
 - these are often just special cases of using the `plot` function
 - one such function is `barplot`

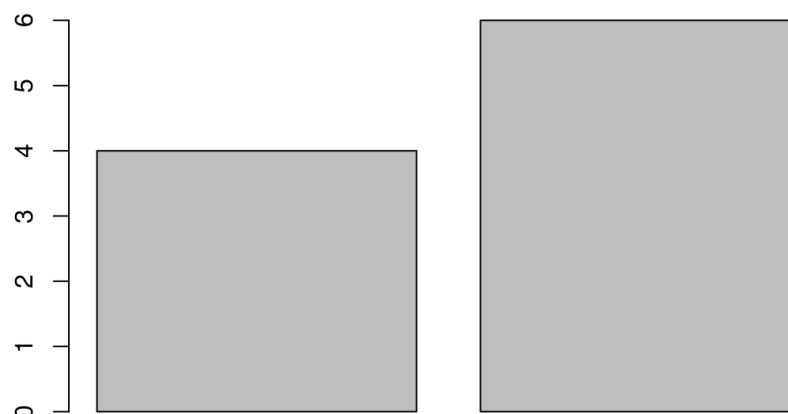
```
barplot(patients$Age)
```



Making a barplot

- It is more usual to display count data in a barplot
 - e.g. the counts of a particular *category* variable

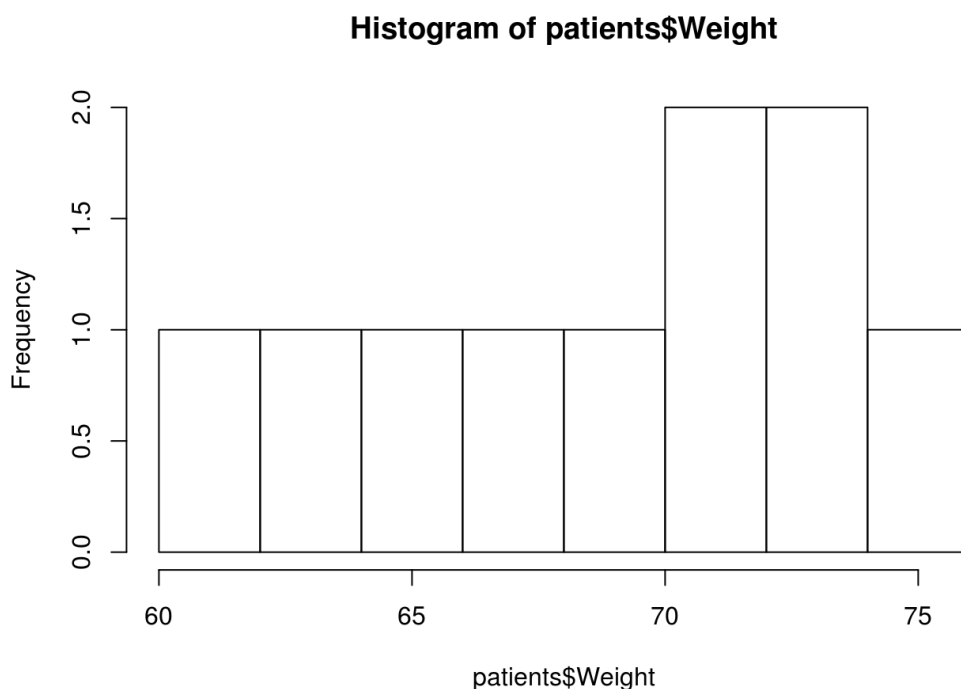
```
barplot(summary(patients$Sex))
```



Plotting a distribution: Histogram

- A histogram is a popular way of visualising a distribution of *continuous* data
 - can change the width of bins
 - y-axis can be either frequency or density

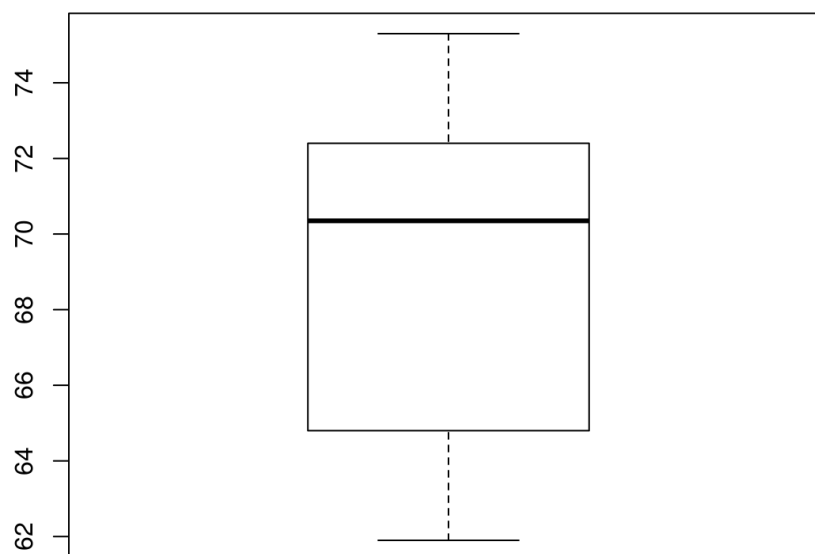
```
hist(patients$Weight)
```



Plotting a distribution: Boxplot

- The boxplot is commonly-used in statistics to visualise a distribution
 - The black solid line is the *median*
 - The top and bottom of the box are the 75th and 25th percentiles
 - Hence, the distance between these is a reflection of the *spread* of the data; the Inter-Quartile Range (*IQR*)
 - Whiskers are drawn at $1.5 \times \text{IQR}$ and $-1.5 \times \text{IQR}$

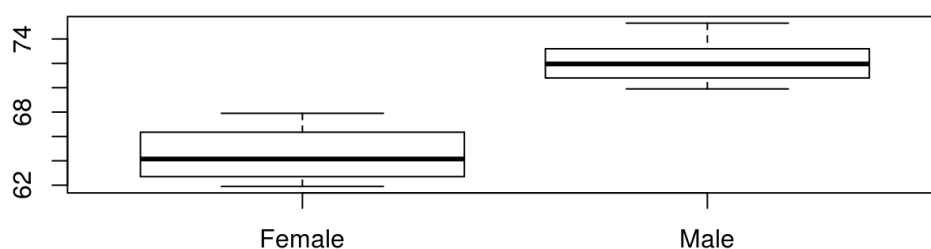
```
boxplot(patients$Weight)
```

Plotting a distribution: Boxplot

- Sometimes we want to compare distributions between different categories in our data
- For this we need to use the *'formula'* syntax
 - For now, $y \sim x$ means put continuous variable y on the y axis and categorical x on the x axis

```
boxplot(patients$Weight~patients$Sex)
```



- Other alternatives to consider
 - `example(dotchart)`
 - `example(stripchart)`
 - `example(vioplots)` ### From vioplots library
 - `example(beeswarm)` ## From beeswarm library

Exercise: Data exploration

- In the Day_1_scripts folder you will find the file `ozone.csv`
 - Data describing weather conditions in New York City in 1973, obtained from the supplementary data (<http://faculty.washington.edu/heagerty/Books/Biostatistics/index-chapter.html>) to *Biostatistics: A Methodology for the Health Sciences*
 - Full description here

<http://faculty.washington.edu/heagerty/Books/Biostatistics/DATA/ozonedoc.txt>
 (http://faculty.washington.edu/heagerty/Books/Biostatistics/DATA/ozonedoc.txt)

- Import these data into R
- What data types are present? Try to think of ways to create the following plots from the data
 - scatter plot two variables. e.g. Solar Radiation against Ozone
 - a histogram. e.g. Temperature
 - boxplot of a continuous variable against a categorical variable. e.g. Ozone level per month

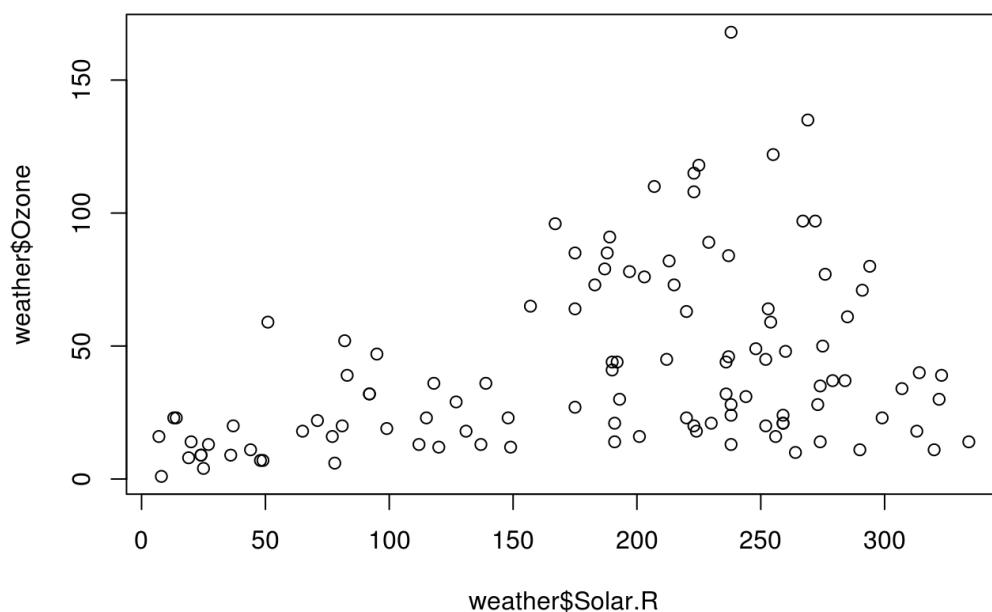
Suggestions

```
weather <- read.csv("ozone.csv")
View(weather)
```

Ozone	Solar.R	Wind	Temp	Month	Day
41	190	7.4	67	5	1
36	118	8.0	72	5	2
12	149	12.6	74	5	3
18	313	11.5	62	5	4
NA	NA	14.3	56	5	5
28	NA	14.9	66	5	6

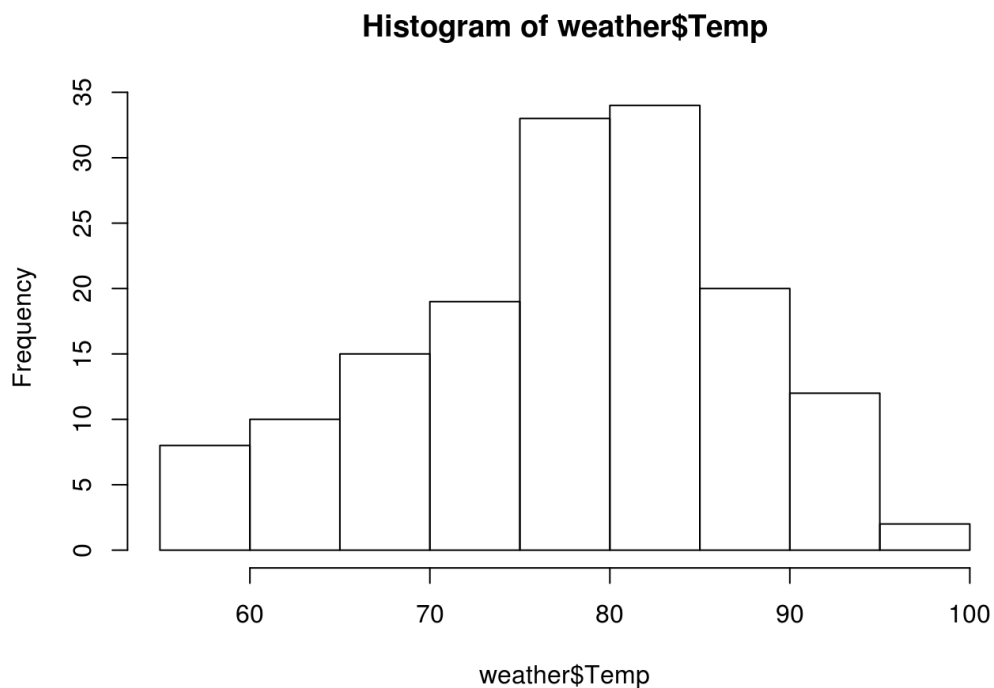
Suggestions

```
plot(weather$Solar.R,weather$Ozone)
```



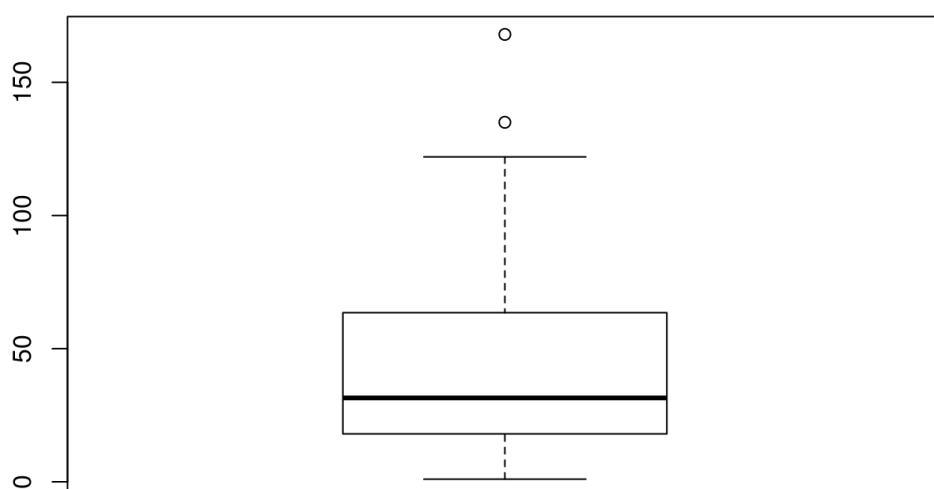
Suggestions

```
hist(weather$Temp)
```



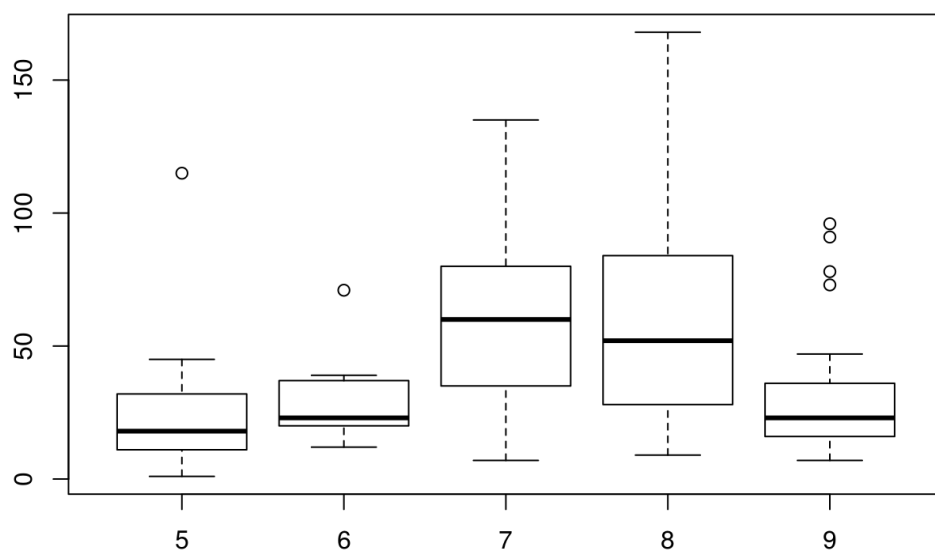
Suggestions

```
boxplot(weather$Ozone)
```



Suggestions

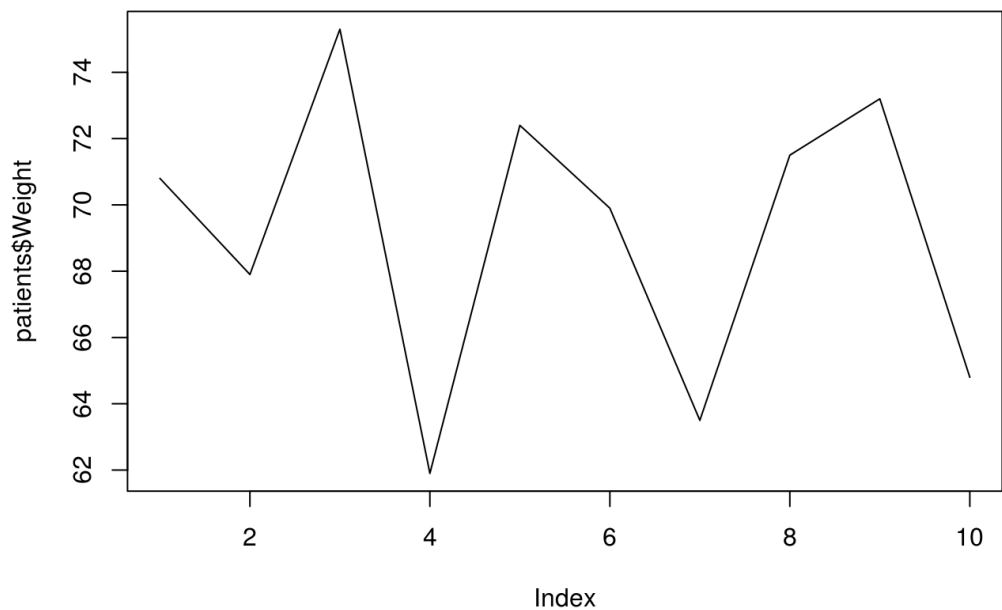
```
boxplot(weather$Ozone~weather$Month)
```



Simple customisations

- `plot` comes with a whole array of arguments that can be set when we call the function
 - see `?plot` and `?par`
- Recall that unless specified, arguments have a default value
- We can choose to draw lines on the plot rather than points
 - the rest of the plot is the same as before

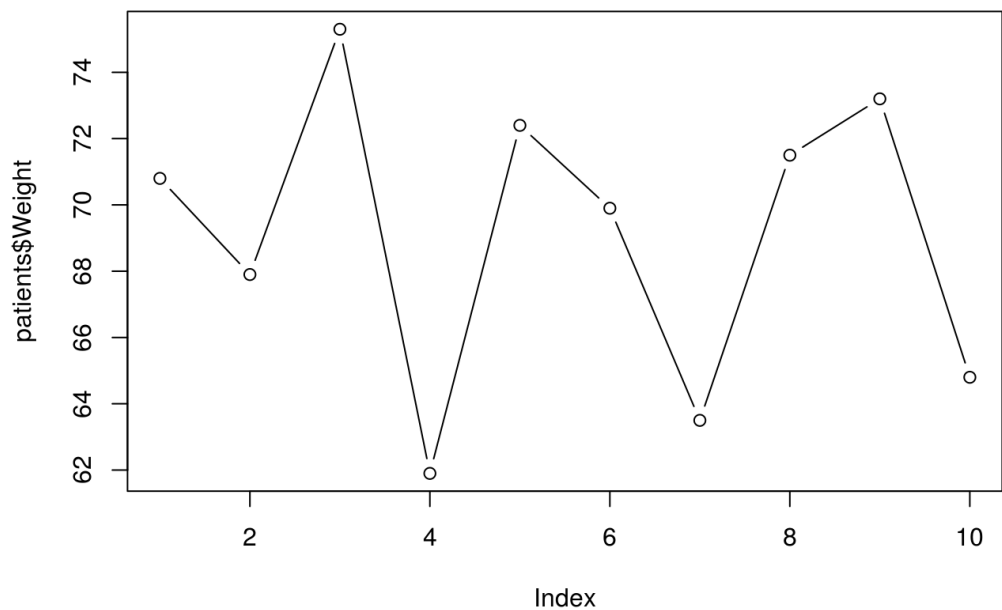
```
plot(patients$Weight,type="l")
```



Simple customisations

- We can also have both lines and points

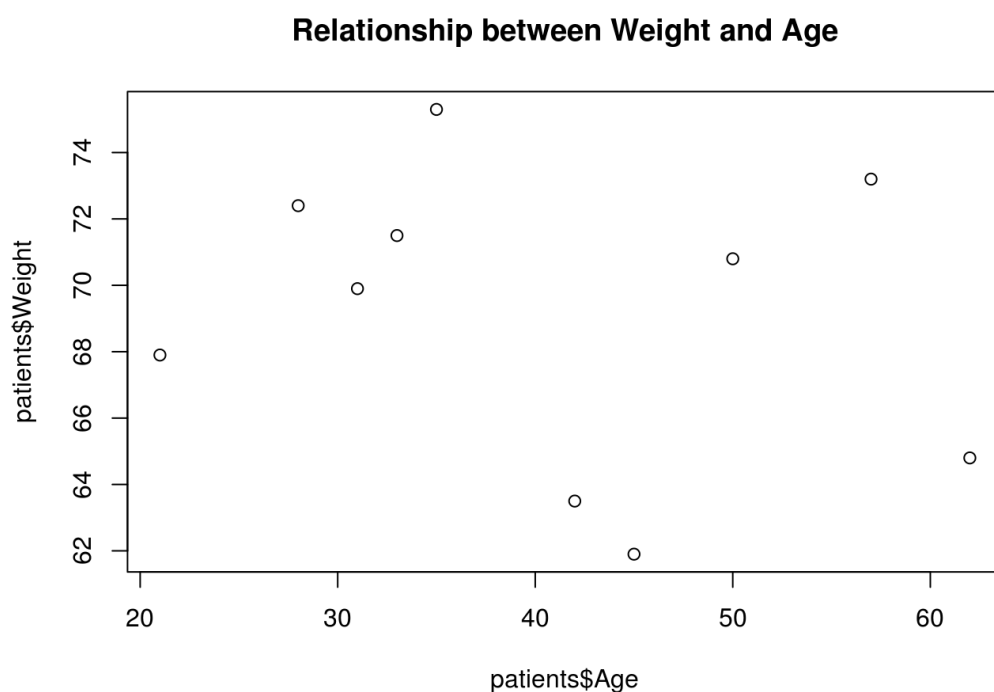
```
plot(patients$Weight, type="b")
```



Simple customisations

- Add an informative title to the plot using the `main` argument

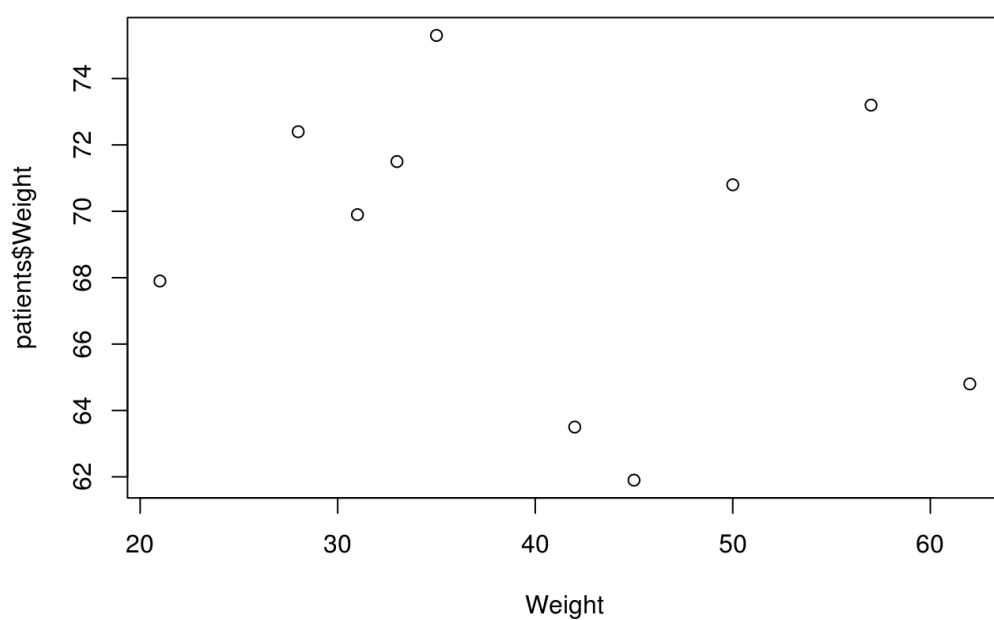
```
plot(patients$Age,patients$Weight,main="Relationship between Weight and Age")
```



Simple customisations

- Adding the x-axis label

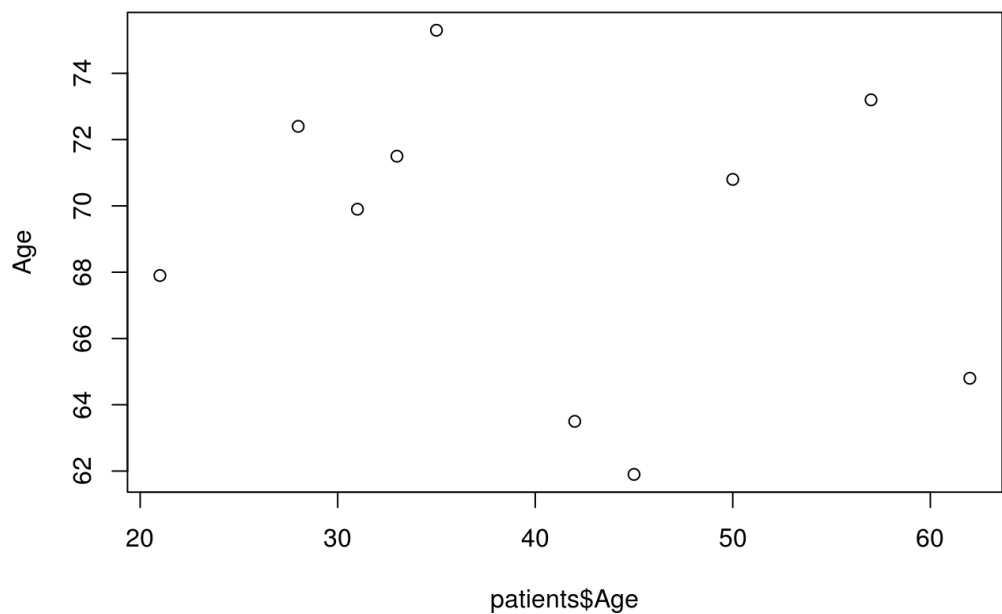
```
plot(patients$Age,patients$Weight,xlab="Weight")
```



Simple customisations

- Adding the y-axis label

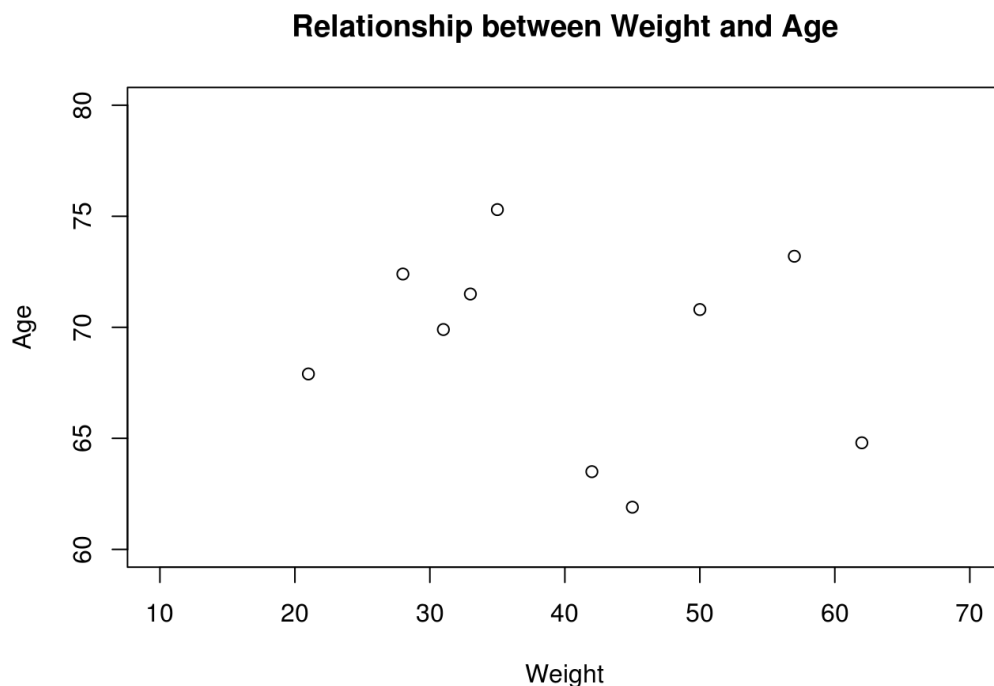
```
plot(patients$Age,patients$Weight,ylab="Age")
```



Simple customisations

- We can specify multiple arguments at once
 - here `ylim` and `xlim` are used to specify axis limits

```
plot(patients$Age,patients$Weight,  
     ylab="Age",  
     xlab="Weight",  
     main="Relationship between Weight and Age",  
     xlim=c(10,70),  
     ylim=c(60,80))
```



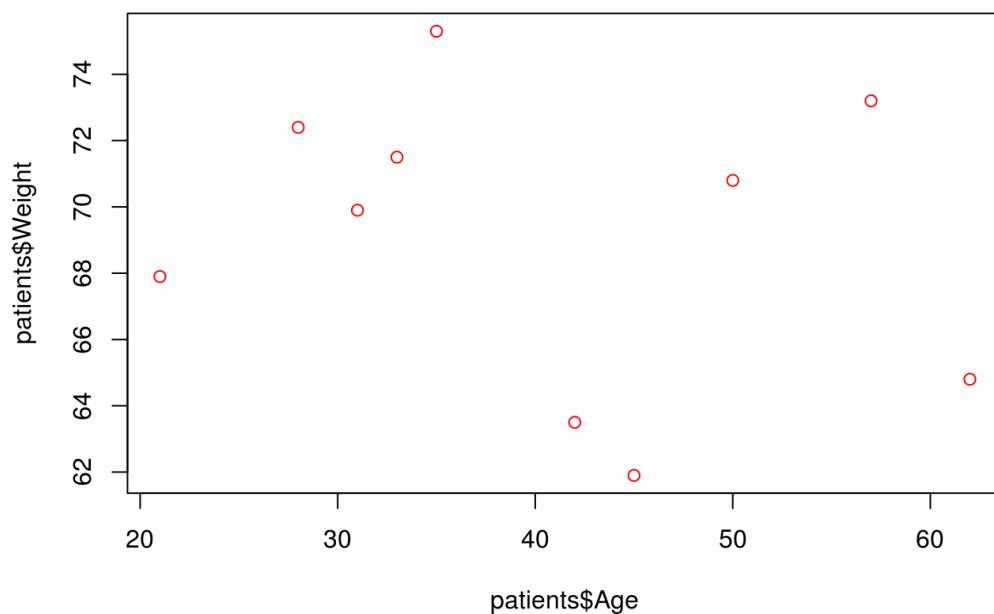
Defining a colour

- R can recognise various strings "red", "orange", "green", "blue", "yellow"
- Or more exotic ones maroon4, lightcyan1, green2, lightsalmon, bisque2, sienna1, grey63, darkslategray2..... See `colours()` .
- See <http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf> (<http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf>)
- Can also use **Red** **Green** **Blue** , hexadecimal, values

Use of colours

Changing the `col` argument to `plot` changes the colour that the points are plotted in

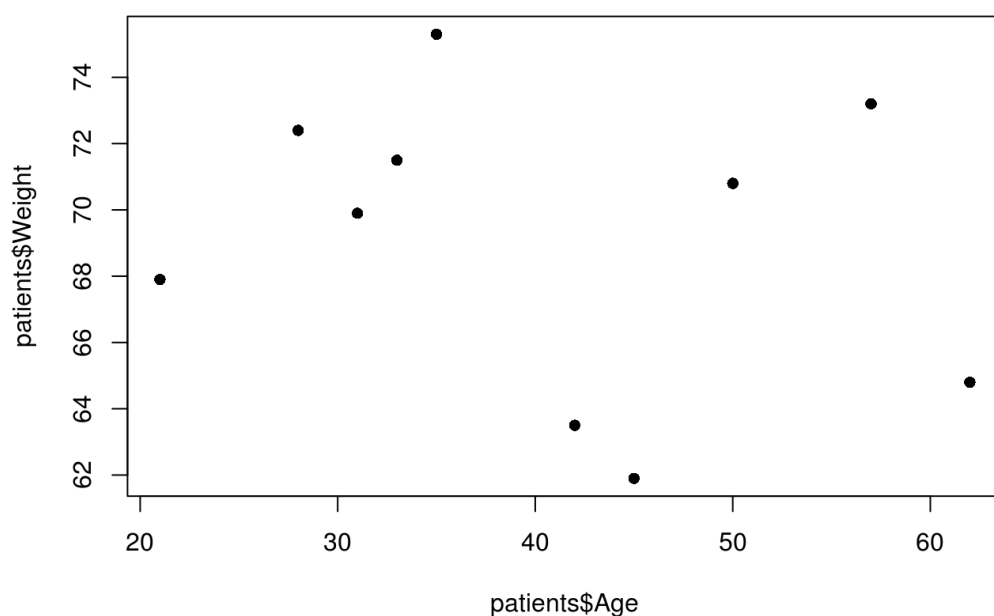
```
plot(patients$Age,patients$Weight,col="red")
```

Plotting characters

- R can use a variety of *plotting characters*
- Each of which has a numeric *code*

```
plot(patients$Age,patients$Weight, pch=16)
```



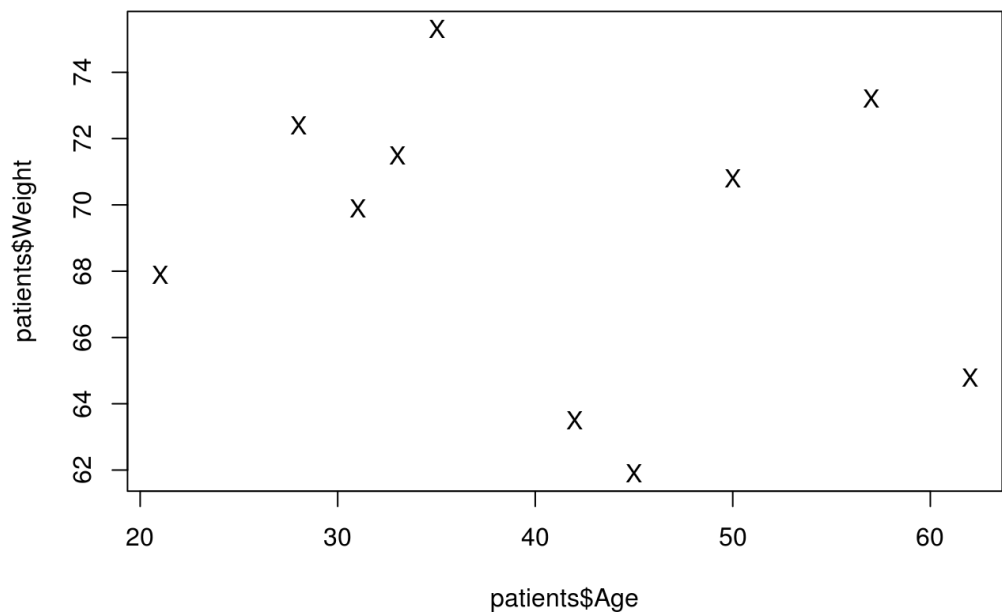
Plotting characters

◇	⊕	■	•	▽
5	10	15	20	25
×	⊕	▣	●	△
4	9	14	19	24
+	*	⊗	◆	◇
3	8	13	18	23
△	⊗	⊞	▲	□
2	7	12	17	22
○	▽	⊗	●	○
1	6	11	16	21

Plotting characters

- Or you can specify a character

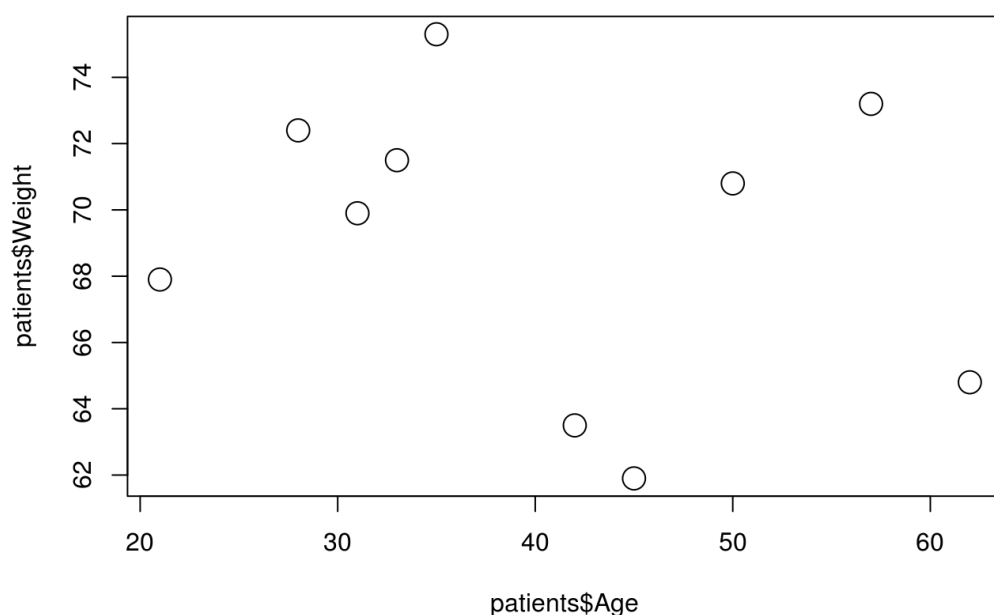
```
plot(patients$Age,patients$Weight, pch="X")
```



Size of points

Character **expansion**

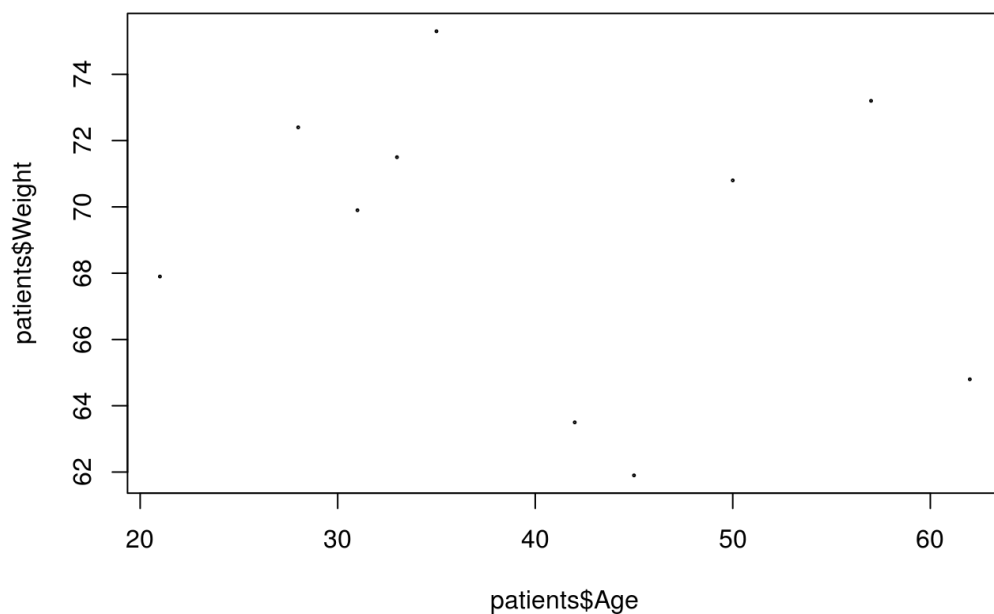
```
plot(patients$Age,patients$Weight, cex=2)
```



Size of points

Character expansion

```
plot(patients$Age,patients$Weight, cex=0.2)
```

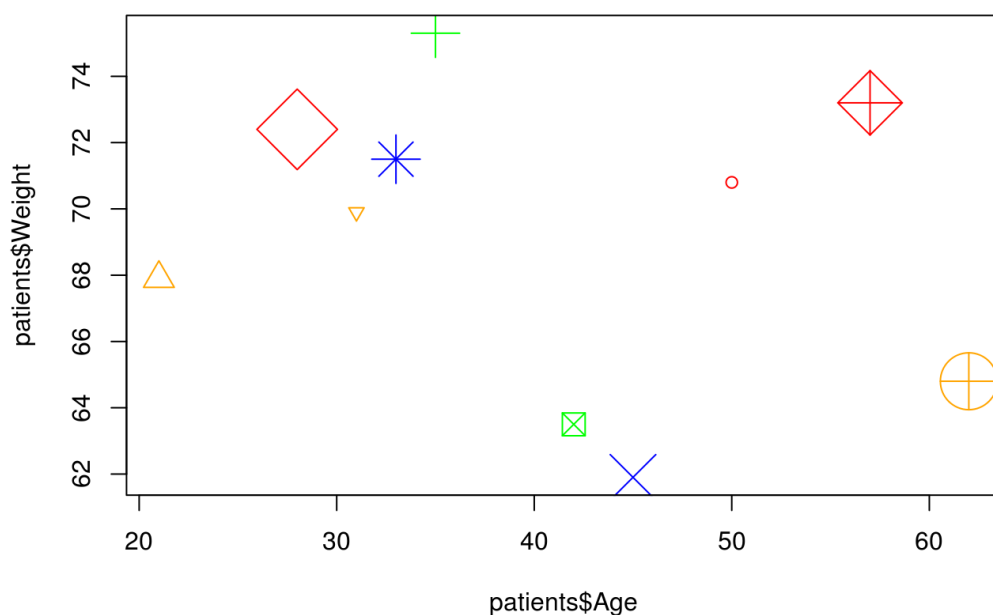


Colours and characters as vectors

- Previously we have used a *vector* of length 1 as our value of colour and character
- We can use a vector of any length
 - the values will get *recycled* (re-used) so that each point gets assigned a value

- We can use a pre-defined **colour palette** (see later)

```
plot(patients$Age,patients$Weight,
     pch = 1:10,
     col=c("red","orange","green","blue"),
     cex = 1:5)
```

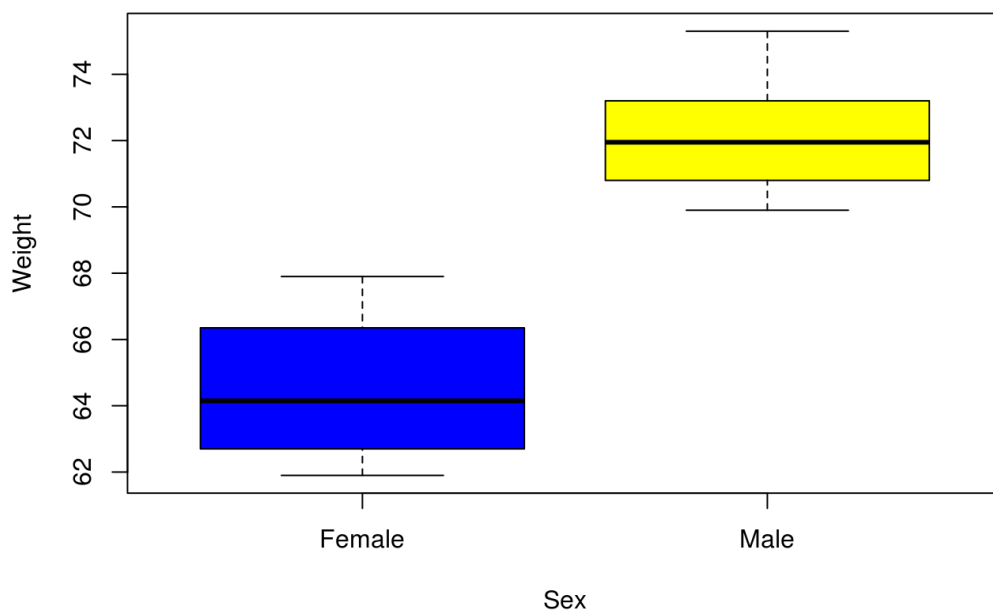


Other plots use the same arguments

- Other plotting functions use the same arguments as `plot`
 - technical explanation: the arguments are *'inherited'*

```
boxplot(patients$Weight~patients$Sex,
        xlab="Sex",
        ylab="Weight",
        main="Relationship between Weight and Gender",
        col=c("blue","yellow"))
```

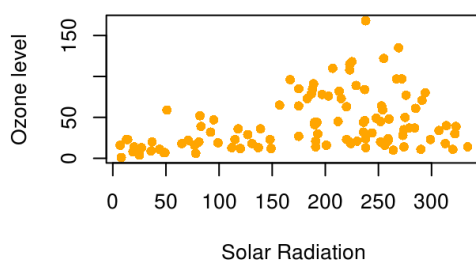
Relationship between Weight and Gender



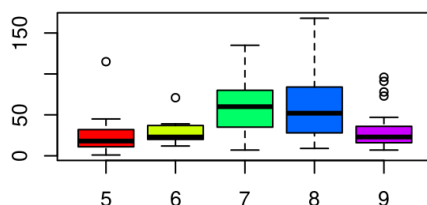
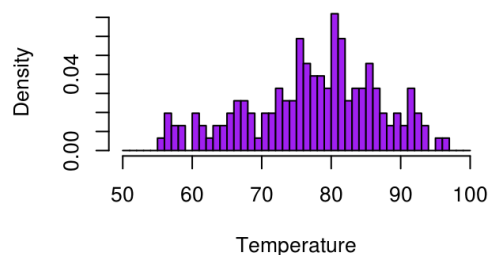
Exercise

- Can you re-create the following plots?
- HINT
 - see the `breaks` and `freq` arguments to `hist()`
 - for third plot, see the `rainbow` function
 - don't worry too much about getting the colours exactly correct

Relationship between ozone level and solar radiation

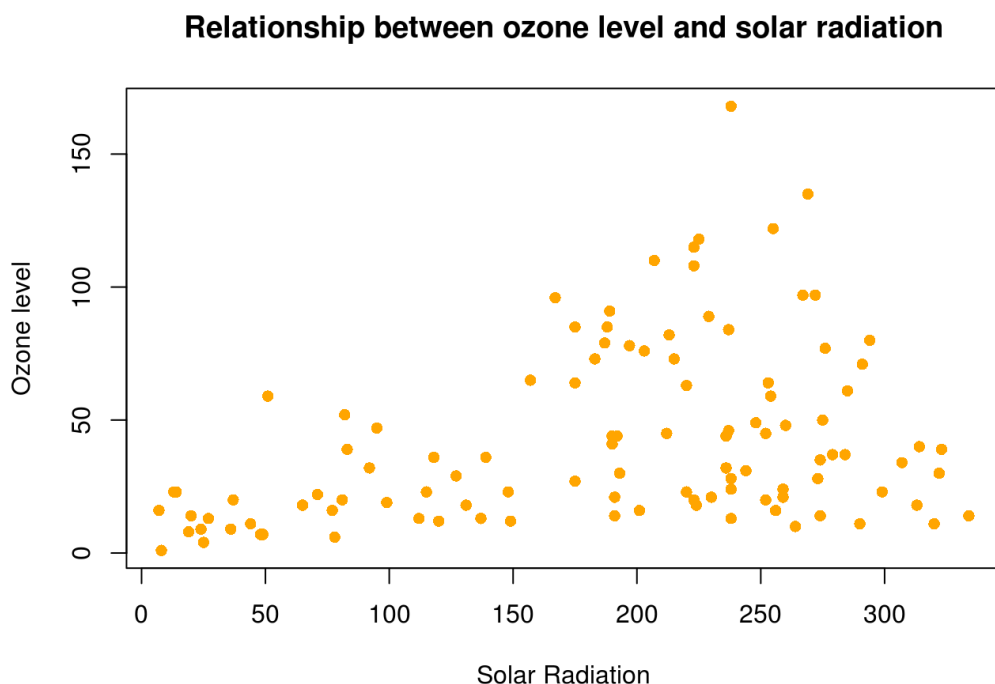


Distribution of Temperature



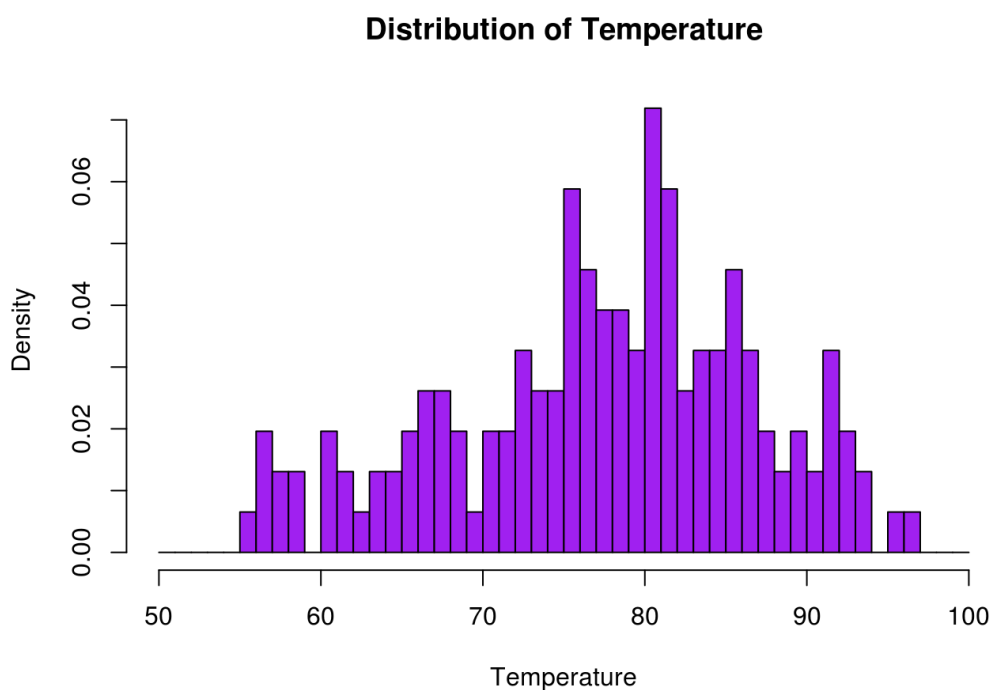
Solutions

```
plot(weather$Solar.R,weather$Ozone,col="orange",pch=16,  
      ylab="Ozone level",xlab="Solar Radiation",  
      main="Relationship between ozone level and solar radiation  
")
```



Solutions

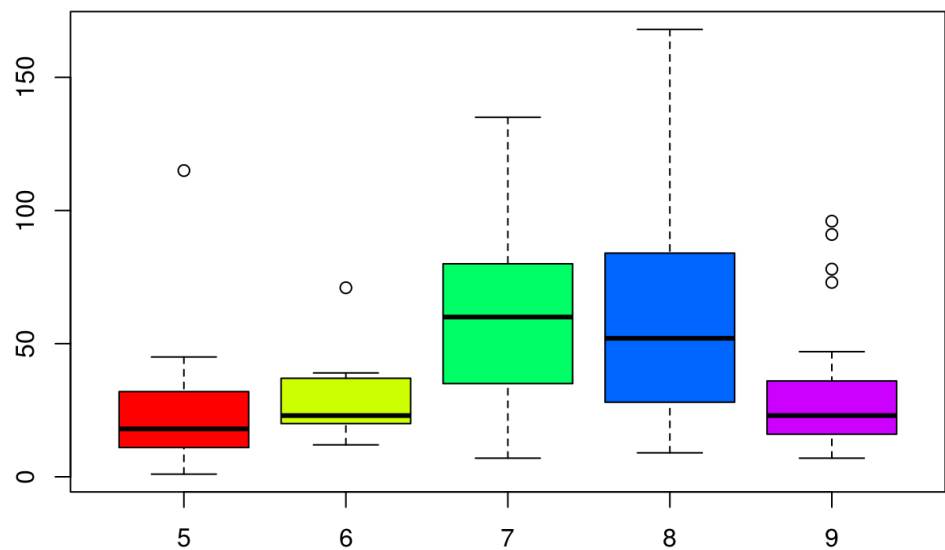
```
hist(weather$Temp,col="purple",xlab="Temperature",  
      main="Distribution of Temperature",breaks = 50:100,freq=FA  
LSE)
```



Solutions

- The `rainbow` function is used to create a vector of colours for the boxplot; in other words a *palette*
 - Red, Orange, Yellow, Green, Blue, Indigo, Violet....etc

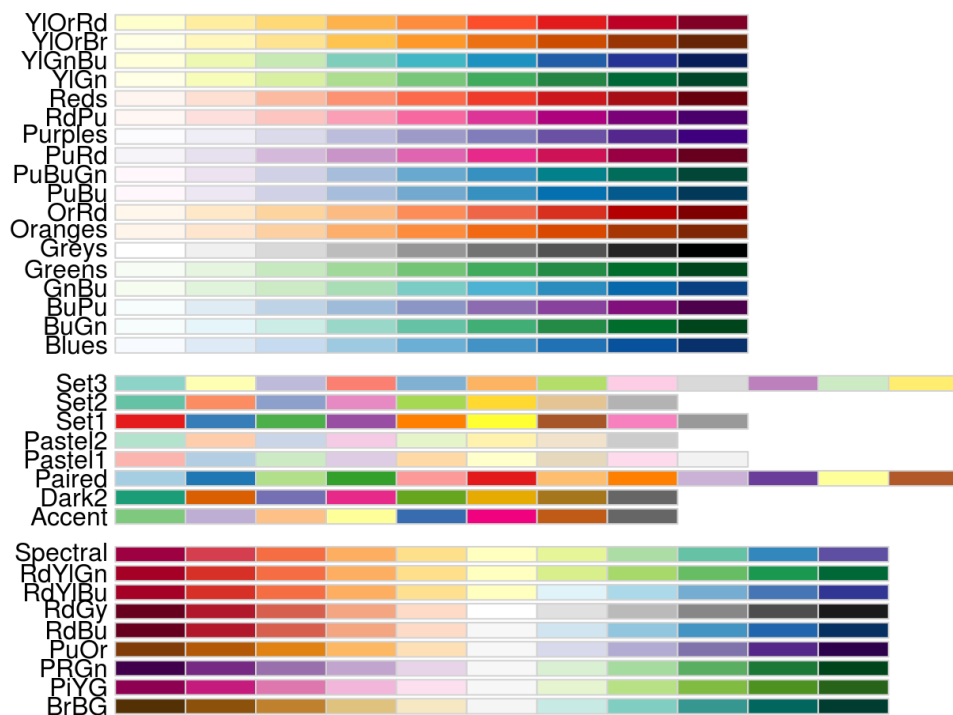
```
boxplot(weather$Ozone~weather$Month,col=rainbow(5))
```



Solutions

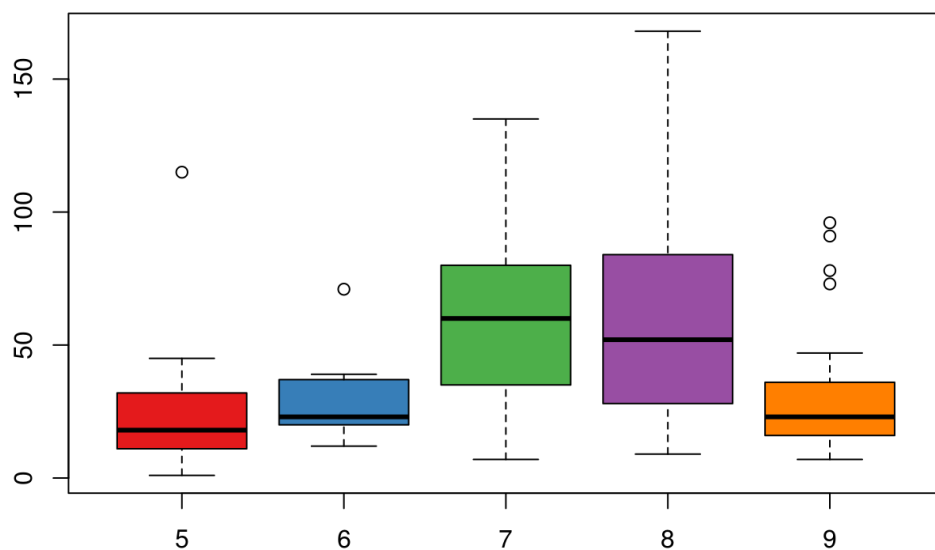
- More aesthetically-pleasing palettes are provided by the `RColorBrewer` package

```
library(RColorBrewer)  
display.brewer.all()
```



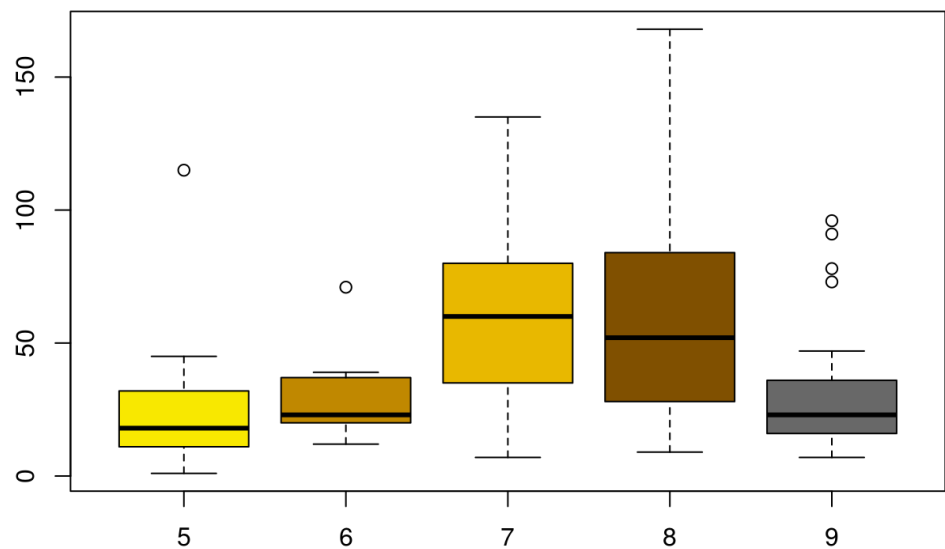
Solutions

```
boxplot(weather$Ozone~weather$Month,col=brewer.pal(5,"Set1"))
```



And finally..

```
library(palettetown)
boxplot(weather$Ozone~weather$Month,col=pokepal("Pikachu",5))
```

End of Day 1

To come tomorrow.....

- More customisation of plots
- Statistics
- Further manipulation of data
- Report writing