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: 27 May 2016

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 screenshot

R in the New York Times

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

New York Times, Jan 2009

R in Nature

nature International weekly journal of science

NATURE | TOOLBOX

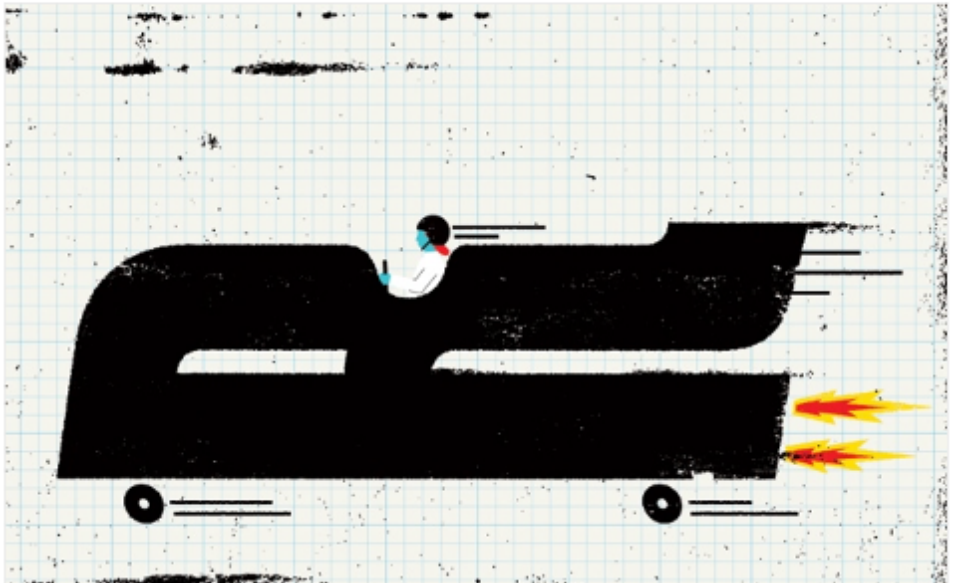
Programming tools: Adventures with R

A guide to the popular, free statistics and visualization software that gives scientists control of their own data analysis.

Sylvia Tippmann

29 December 2014

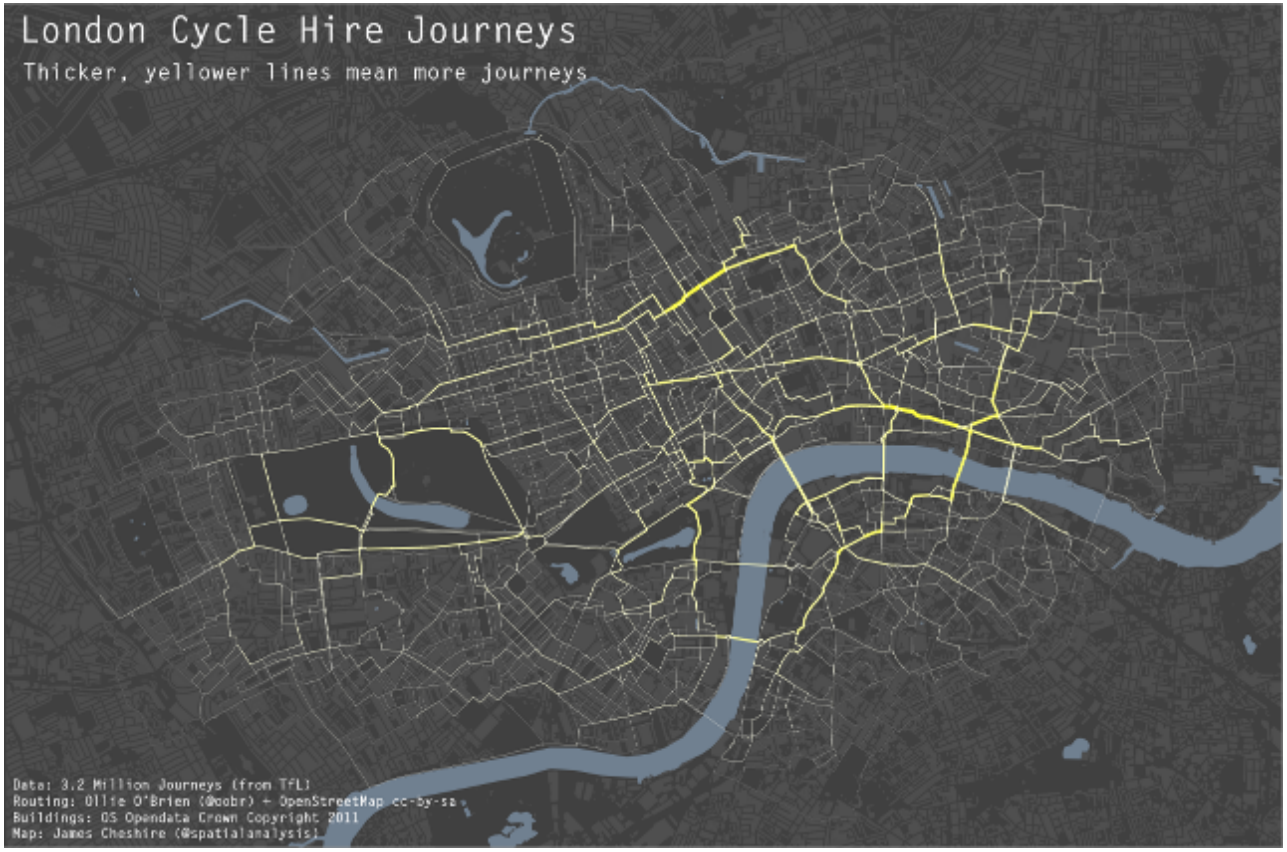
PDF Rights & Permissions



Nature, Dec 2014

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)

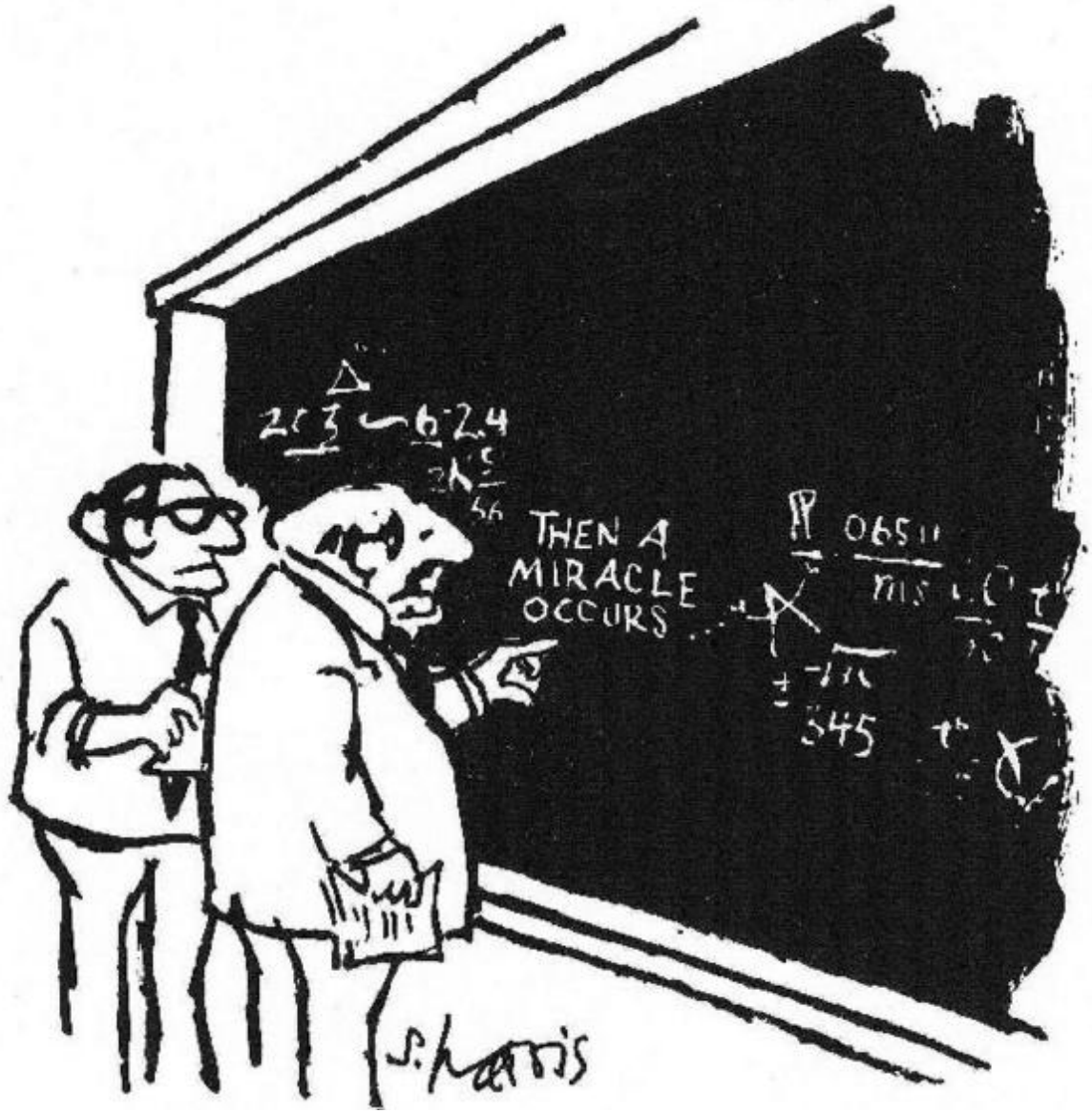
Who uses R? Not just academics!

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)
- Buzzfeed
  - http://blog.revolutionanalytics.com/2015/12/buzzfeed-uses-r-for-data-journalism.html (http://blog.revolutionanalytics.com/2015/12/buzzfeed-uses-r-for-data-journalism.html)

R can facilitate Reproducible Research


THEN A
MIRACLE
OCCURS

Sidney Harris - New York Times

It is a hot topic at the moment

- Statisticians at MD Anderson tried to reproduce results from a Duke paper and unintentionally unravelled a web of incompetence and skullduggery
    - as reported in the ***New York Times***

Submit Your

RESEARCH

75 COMMENTS

How Bright Promise in Cancer Testing Fell Apart

By GINA KOLATA JULY 7, 2011

Email

Share

Tweet

Pin

Save

More

When Juliet Jacobs found out she had lung cancer, she was terrified, but realized that her hope lay in getting the best treatment medicine could offer. So she got a second opinion, then a third. In February of 2010, she ended up at Duke University, where she entered a research study whose promise seemed stunning.

Doctors would assess her tumor cells, looking for gene patterns that would determine which drugs would best

Keith Baggerly, left, and Kevin Coombes, statisticians at M. D. Anderson Cancer Center, found flaws in research on tumors.
Michael Stravato for The New York Times

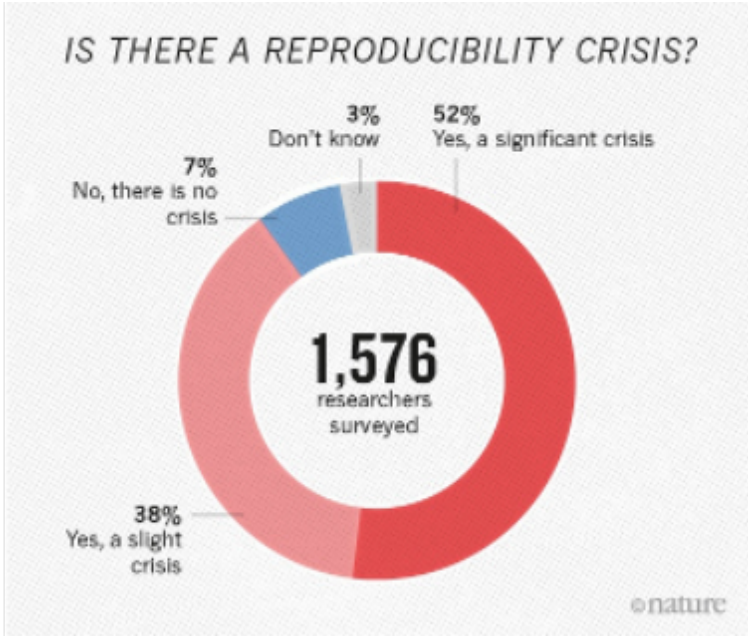
New York Times, July 2011

Hear the full account

- Very entertaining talk from Keith Baggerly in Cambridge, December 2010

The Importance of Reproducible Research in High-Throu...

Nature editorial - May 2016



rep-crisis

Reality check on reproducibility (http://www.nature.com/news/reality-check-on-reproducibility-1.19961)

1,500 scientists lift the lid on reproducibility (http://www.nature.com/news/1-500-scientists-lift-the-lid-on-reproducibility-1.19970)

Various platforms supported

- Release 3.3.0 (May 2016)
  - Base package and Contributed packages (general purpose extras)
  - 8469 available packages as of Fri May 27 11:36:23 2016
- 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)

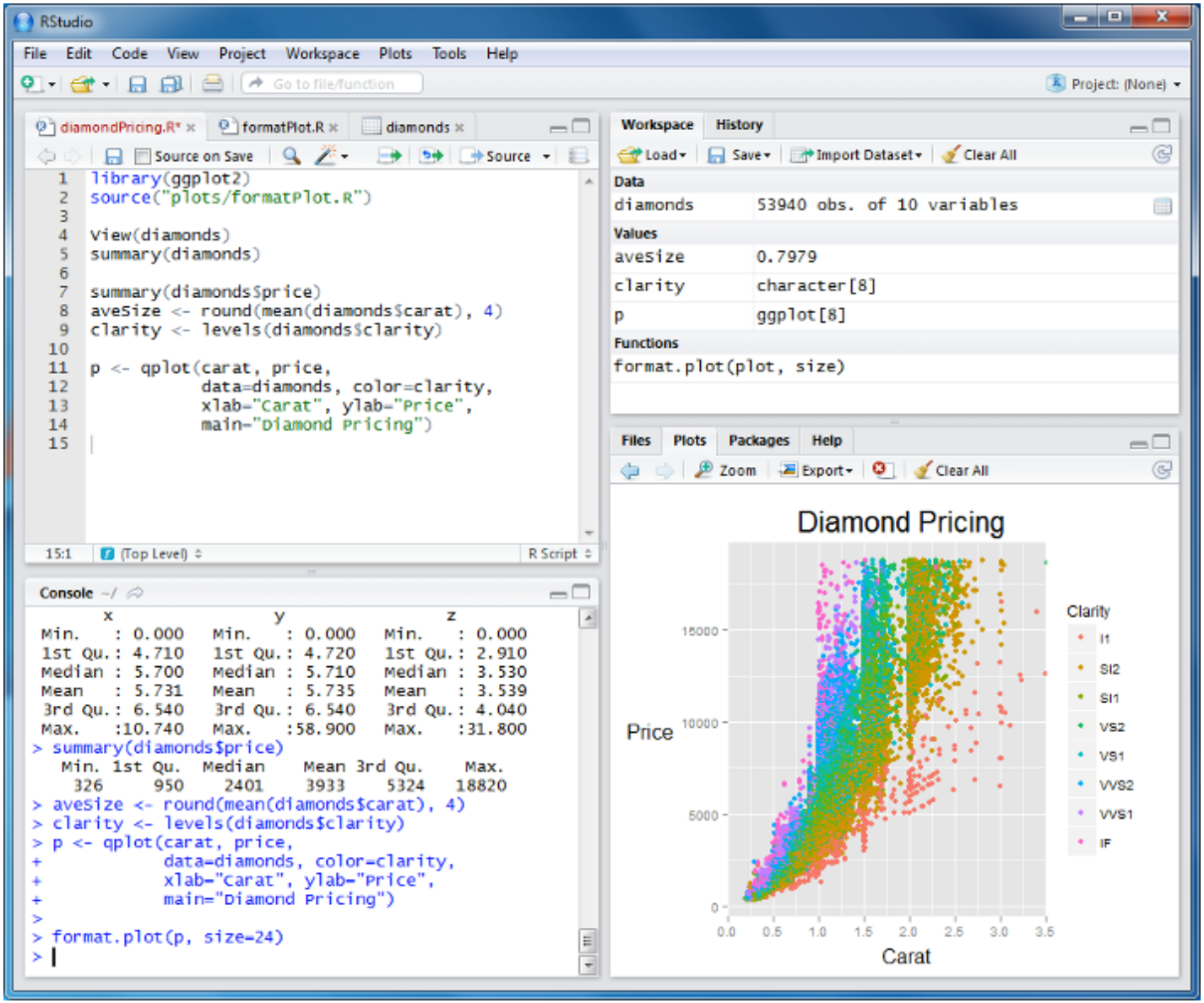
- Everything you need is installed on the training machines
- If you are using your own machine, download both R and RStudio

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


RStudio
File Edit Code View Project Workspace Plots Tools Help
diamondPricing.R* formatPlot.R diamonds
Source on Save Source
1 library(ggplot2)
2 source("plots/formatPlot.R")
3
4 View(diamonds)
5 summary(diamonds)
6
7 summary(diamonds$price)
8 aveSize <- round(mean(diamonds$carat), 4)
9 clarity <- levels(diamonds$clarity)
10
11 p <- qplot(carat, price,
12 data=diamonds, color=clarity,
13 xlab="Carat", ylab="Price",
14 main="Diamond Pricing")
15
15:1 (Top Level) R Script
Console ~/
x y z
Min. : 0.000 Min. : 0.000 Min. : 0.000
1st Qu.: 4.710 1st Qu.: 4.720 1st Qu.: 2.910
Median : 5.700 Median : 5.710 Median : 3.530
Mean : 5.731 Mean : 5.735 Mean : 3.539
3rd Qu.: 6.540 3rd Qu.: 6.540 3rd Qu.: 4.040
Max. :10.740 Max. :58.900 Max. :31.800
> summary(diamonds$price)
Min. 1st Qu. Median Mean 3rd Qu. Max.
326 950 2401 3933 5324 18820
> aveSize <- round(mean(diamonds$carat), 4)
> clarity <- levels(diamonds$clarity)
> p <- qplot(carat, price,
+ data=diamonds, color=clarity,
+ xlab="Carat", ylab="Price",
+ main="Diamond Pricing")
>
> format.plot(p, size=24)
>
Workspace History
Load Save Import Dataset Clear All
Data
diamonds 53940 obs. of 10 variables
Values
aveSize 0.7979
clarity character[8]
p ggplot[8]
Functions
format.plot(plot, size)
Files Plots Packages Help
Zoom Export Clear All
Diamond Pricing
Price
Carat
Clarity
I1
SI2
SI1
VS2
VS1
VVS2
VVS1
IF
0
5000
10000
15000
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5

RStudio screenshot

Basic concepts in R - command line calculation

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

```
2 + 2
20/5 - sqrt(25) + 3^2
sin(pi/2)
```

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

myNumber <- 25
myNumber
```

- We can perform arithmetic on variables:

```
sqrt(myNumber)
```

Basic concepts in R - variables

- We can add variables together:

```
x + myNumber
```

- We can change the value of an existing variable:

```
x <- 21
x
```

Basic concepts in R - variables

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

```
x <- myNumber
x
```

- We can modify the contents of a variable:

```
myNumber <- myNumber + sqrt(16)
myNumber
```

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)
max(3,4,5,6)
min(3,4,5,6)
```

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)

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

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

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
```

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

```
x[1]
x[4]
```

Basic concepts in R - vectors

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

```
y <- c(2,3)
x[y]
```

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
```

Basic concepts in R - vectors

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

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

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

- or we can use the **`rep()`** function:

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

```
y <- rep(1:3, 5)
y
```

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
# Extract elements from x:

x[3:7]
x[seq(2, 6, 2)]
x[rep(3, 2)]
```

Basic concepts in R - vectors

- We can add an element to a vector:

```
y <- c(x, 1)
y
```

- We can glue vectors together:

```
z <- c(x, y)
z
```

Basic concepts in R - vectors

- We can remove element(s) from a vector:

```
x <- 3:12

x[-3]
x[-(5:7)]
x[-seq(2, 6, 2)]
```

Basic concepts in R - vectors

- Finally, we can modify the contents of a vector:

```
x[6] <- 4
x

x[3:5] <- 1
x
```

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
```

```
z <- x^2
```

```
z
```

Basic concepts in R - vector arithmetic

- Adding two vectors:

```
y + z
```

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

```
x + 1:2
```

- 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")
gene.names
```

```
[1] "Pax6"       "Beta-actin" "FoxP2"
[4] "Hox9"
```

Basic concepts in R - Character vectors and naming

- 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)
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)
```

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

1. 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`**.
2. **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.)

3. How many times more bases are used for coding in the human genome compared to the yeast genome? (*S. cerevisiae*) **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
```

Answers to genome exercise

1. 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

2. 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
```

Answers to genome exercise

3. 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
```

Answers to genome exercise

- 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
```

Documenting your analysis 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.

- To create an R markdown file, Click on ***File → New File → R Markdown*** in Rstudio
  - markdown is a easy-to-read, easy-to-write text format often used to write HTML, readme files, etc.
  - a simpler (but not so informative) alternative is to use a script
- This will make our analyses ***reproducible***

Format of an R markdown file

- **Lines 8 - 10**: plain text description
- **Lines 12 - 14**: an R code 'chunk'
- **Lines 18 to 20**: another code chunk, this time producing a plot

```
 7
 8  This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents.
    For more details on using R Markdown see <http://rmarkdown.rstudio.com>.
 9
10  When you click the **Knit** button a document will be generated that includes both content as well as the output of any
    embedded R code chunks within the document. You can embed an R code chunk like this:
11
12  ```{r}
13  summary(cars)
14  ```
15
16  You can also embed plots, for example:
17
18  ```{r, echo=FALSE}
19  plot(cars)
20  ```
```

md-format

- Pressing the ***Knit HTML*** (/***Knit PDF***) button will create a report
- See `solution-exercise1.Rmd` for solution to Exercise 1
- All exercises have a markdown template that you can edit

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. The help page shows the default order of arguments. It also tells you which *package* it belongs to.
- There is typically a usage example, which you can test using the `example` function:

```
example(seq)
```

Getting help

- 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 locate the help pages for a particular package and its functions
  - Often there will be a user-guide or '*vignette*' too

Interacting with the R console

- **Important** – 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**. For example: 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/)

R packages

- Other repositories:
    - http://r-forge.r-project.org/ (http://r-forge.r-project.org/)
    - https//github.com can also host R packages, and hosts the development version of many 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")
```

Example: 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")
```

Example: Load packages ggplot2 and 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
Adam	Jones	Adam Jones	Male	50	70.8	TRUE
Eve	Parker	Eve Parker	Female	21	67.9	TRUE
John	Evans	John Evans	Male	35	75.3	FALSE
Mary	Davis	Mary Davis	Female	45	61.9	TRUE
Peter	Baker	Peter Baker	Male	28	72.4	FALSE
Paul	Daniels	Paul Daniels	Male	31	69.9	FALSE
Joanna	Edwards	Joanna Edwards	Female	42	63.5	FALSE
Matthew	Smith	Matthew Smith	Male	33	71.5	TRUE
David	Roberts	David Roberts	Male	57	73.2	FALSE
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")
```

Character, numeric and logical data types

- 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)
class(age)
class(weight)
class(consent)
```

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"
 [6] "Male"   "Female" "Male"   "Male"   "Female"
```

Factors

```
factor(sex)
```

```
 [1] Male   Female Male   Female Male   Female Male   Male
 [7] 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 (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. ...
1      Adam      Jones                    Adam Jones
2       Eve     Parker                    Eve Parker
3      John      Evans                    John Evans
4      Mary      Davis                    Mary Davis
5     Peter      Baker                   Peter Baker
6      Paul    Daniels                  Paul Daniels
7       ...
```

Naming data frame variables

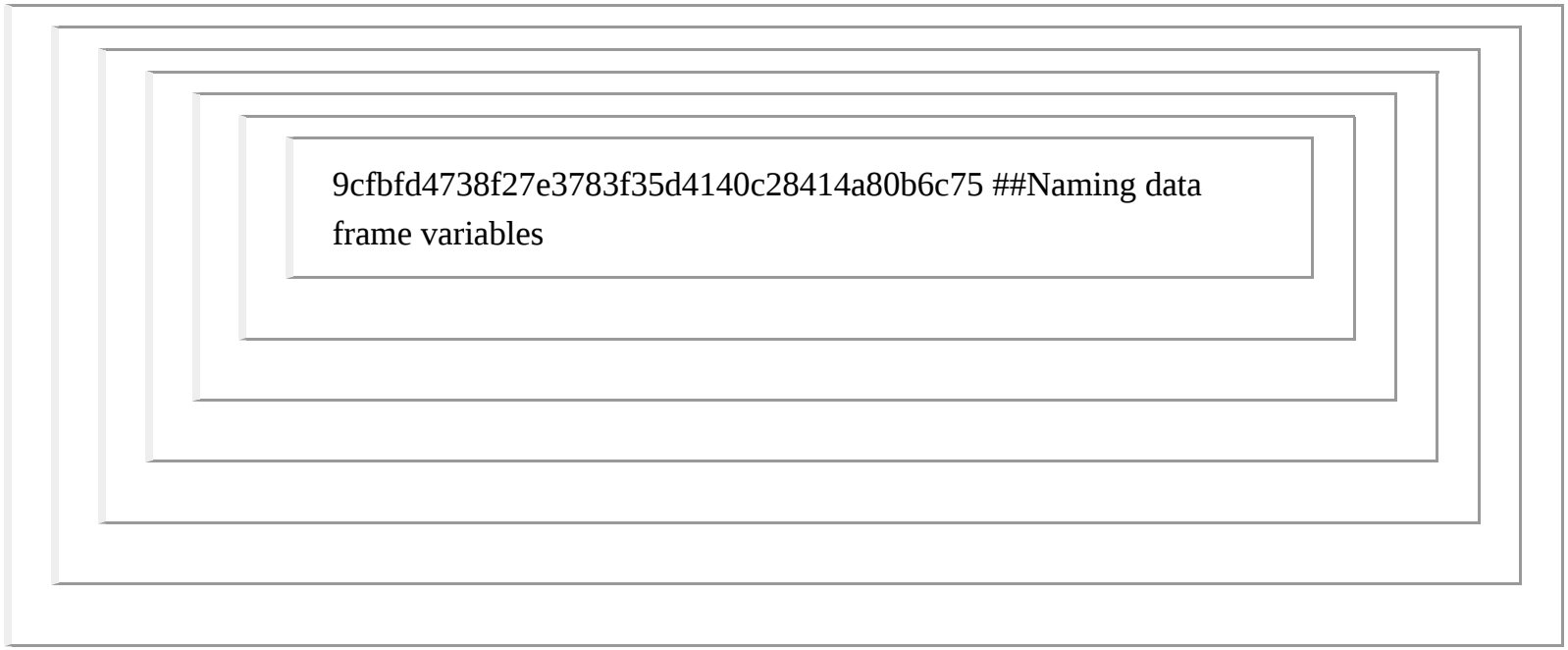
- We can access particular variables using the **'$'** *operator*:

```
patients$age
```

- 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)
```



- 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),
                       Full_Name = paste(firstName,
                                         secondName),
                       Sex = sex,
                       Age = age,
                       Weight = weight,
                       Consent = consent)
```

```
names(patients)
```

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
```

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$Sex
patients$First_Name
```

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:
- **`object[rows, colums]`**

```
e[1,2]
e[1,]
patients[1,2]
patients[1,]
```

- If you don't provide an index for either rows or columns, all of the rows or columns will be returned.

Advanced indexing

- 'Values' in R are really vectors
- Indices are actually vectors, and can be ***numeric*** or ***logical***:

```
s <- letters[1:5]
s
```

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

```
# View some of the values in s:
```

```
s[c(1,3)]
s[c(TRUE, FALSE, TRUE, FALSE, FALSE)]
```

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

# Logical tests:
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]
s[a == 2]
```

Exercise: exercise2.Rmd

- The markdown template has code to create the patients data frame from the slides
- Make a new data frame with three extra 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 from the data frame described in the slides:
  1. Patients under 40
  2. Patients who give consent to share their data
  3. Men who weigh as much or more than the average European male (70.8 kg)

Logical indexing answers: solution-exercise2.pdf

1. Patients under 40:

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

2. Patients who give consent to share their data:

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

3. 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)

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("/home/participant/Course_Materials")
```

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

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("countData.txt")
```

- Using `file.choose()` :

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

- If the data is comma-separated, then use either the argument `sep=","` or the function `read.csv()` :

```
read.csv("countData.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

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

- 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)
nrow(rawData)
dim(rawData)
```

- Or the structure of an object:
  - TIP: In RStudio, window *Environment*, click the blue arrow in the left of an object's name, in order to see the object structure

```
str(rawData)
```

```
'data.frame':   50 obs. of  5 variables:
 $ Patient: int  1 2 3 4 5 6 7 8 9 10 ...
 $ Nuclei : int  65 51 37 37 45 46 65 59 49 46 ...
 $ NB_Amp : int  0 3 2 2 2 4 1 1 0 0 ...
 $ NB_Nor : int  63 43 35 35 42 41 64 54 48 45 ...
 $ NB_Del : int  2 5 0 0 1 1 0 4 1 1 ...
```

1c. Understanding the object

- The names of the columns are automatically assigned:

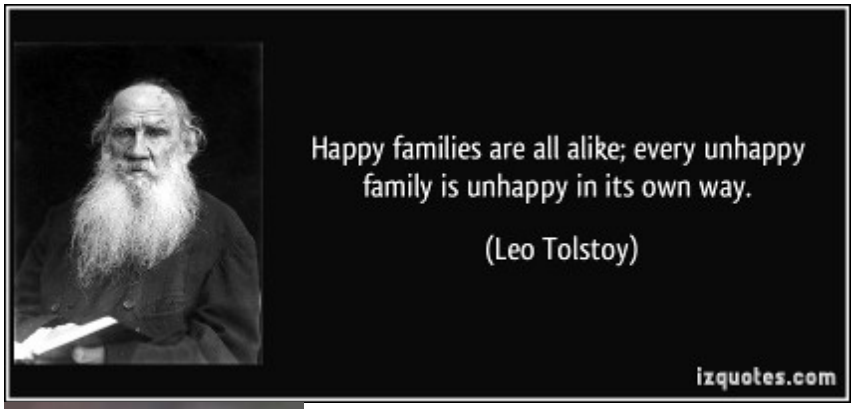
```
colnames(rawData)
```

```
[1] "Patient" "Nuclei"  "NB_Amp"  "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
```

Word of caution



> Like families, tidy datasets are all alike but every messy dataset is messy in its own way - (Hadley Wickham - RStudio chief scientist and author of dplyr, ggplot2 and others)

Word of caution

- 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)
```

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)

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

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
```

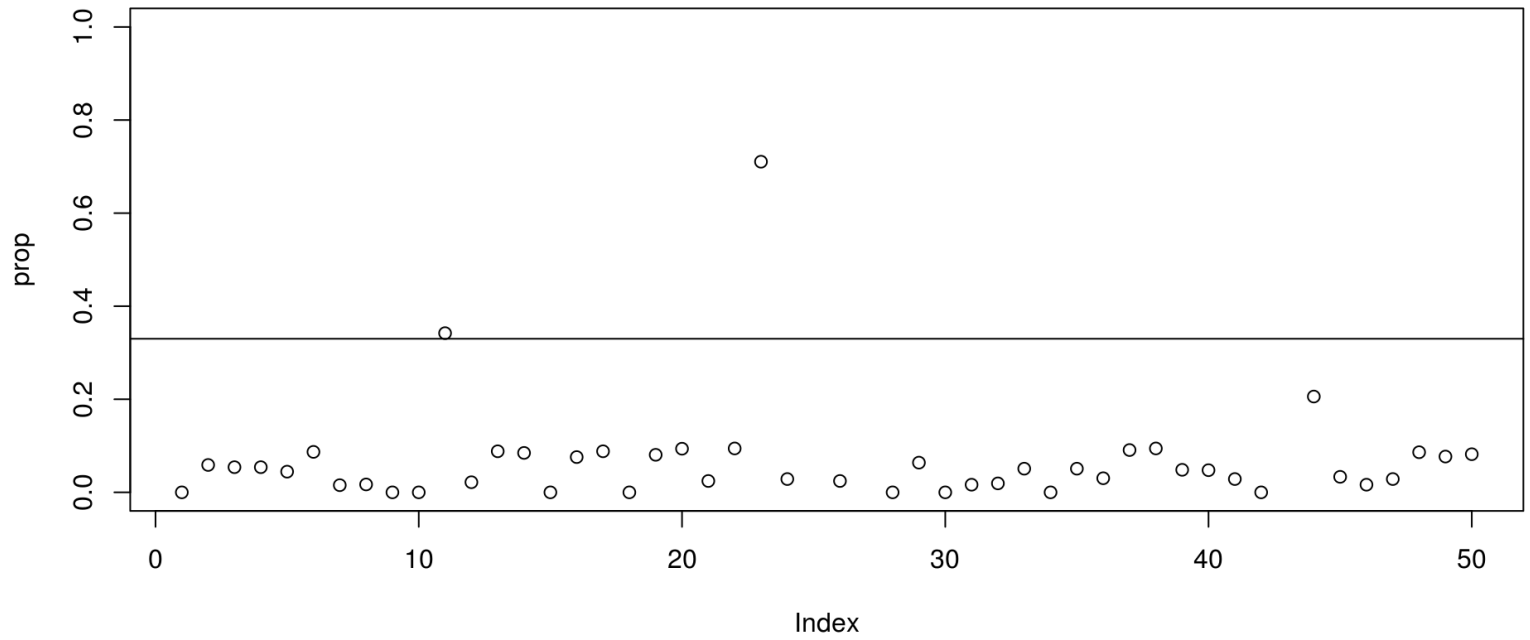
```
# Get sample names of amplified patients:
amp <- which(prop > 0.33)
```

```
amp
```

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))
# Add a horizonal line:
abline(h=0.33)
```



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: exercise3.Rmd

- 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: solution-exercise3.pdf

```
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 more advanced alternatives are, e.g., `lattice` , `ggplot2`
    - See our intermediate R course (http://training.csx.cam.ac.uk/bioinformatics/event/1800066) for fancy graphics
- 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)

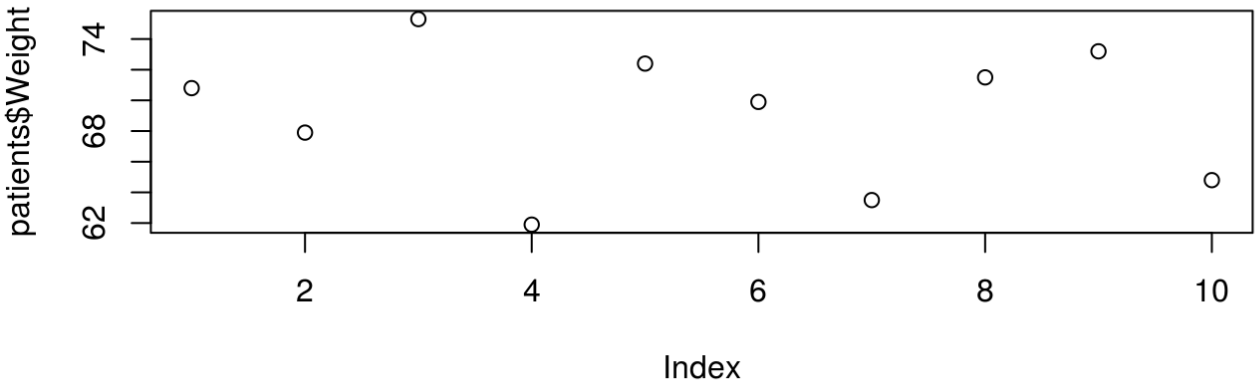
Making a Scatter Plot

- If given a single vector as an argument, the function **`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
```

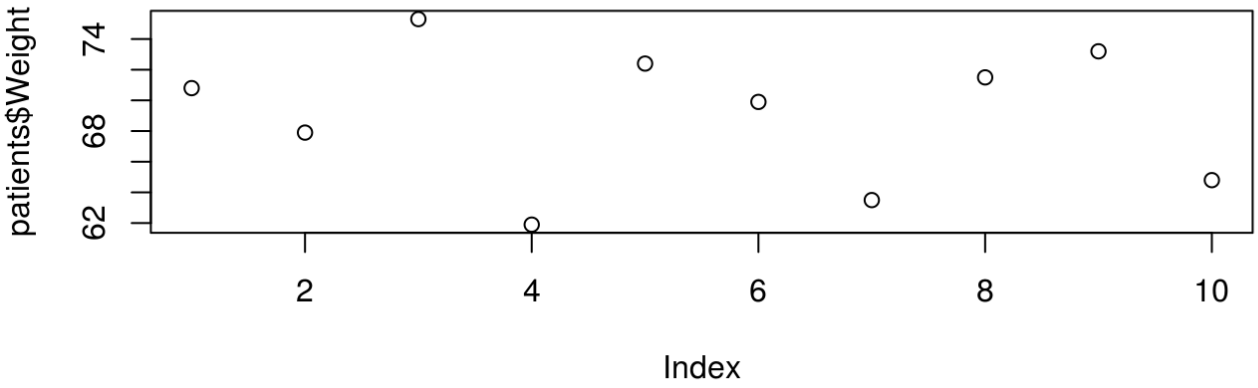
```
 [1] 70.8 67.9 75.3 61.9 72.4 69.9 63.5 71.5 73.2 64.8
```

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



Making a Scatter Plot

- R tries to guess the most appropriate way to visualise the data, according to the type and dimensions of the object(s) provided



- Axis limits, labels, titles are inferred from the data
  - We can modify these as we wish, by specifying ***arguments***

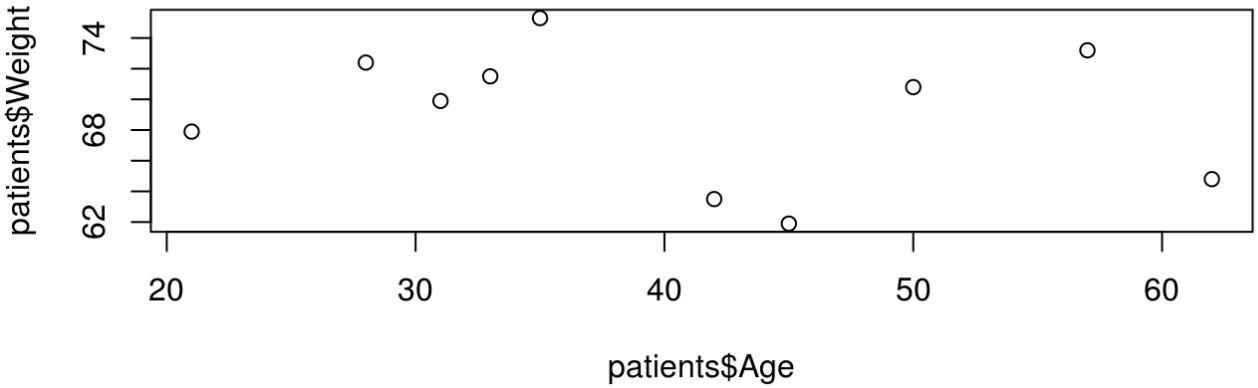
Making a Scatter Plot of two variables

- We can give two arguments to `plot()`:
  - In order to visualise the relationship between two variables
  - It will put the values from the *first* argument in the *x* axis, and values from the *second* argument on the *y* axis

```
patients$Age
```

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

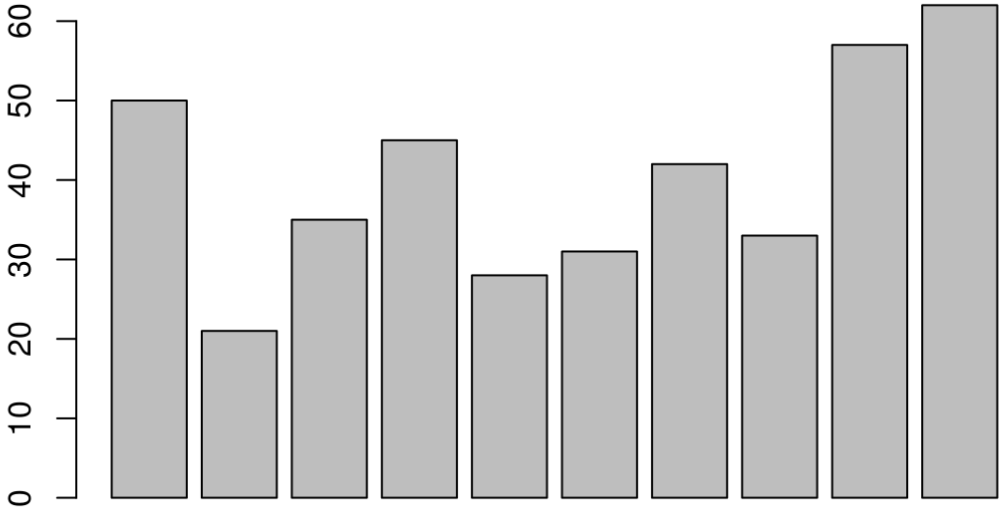
```
plot(patients$Age, patients$Weight)
```



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()`

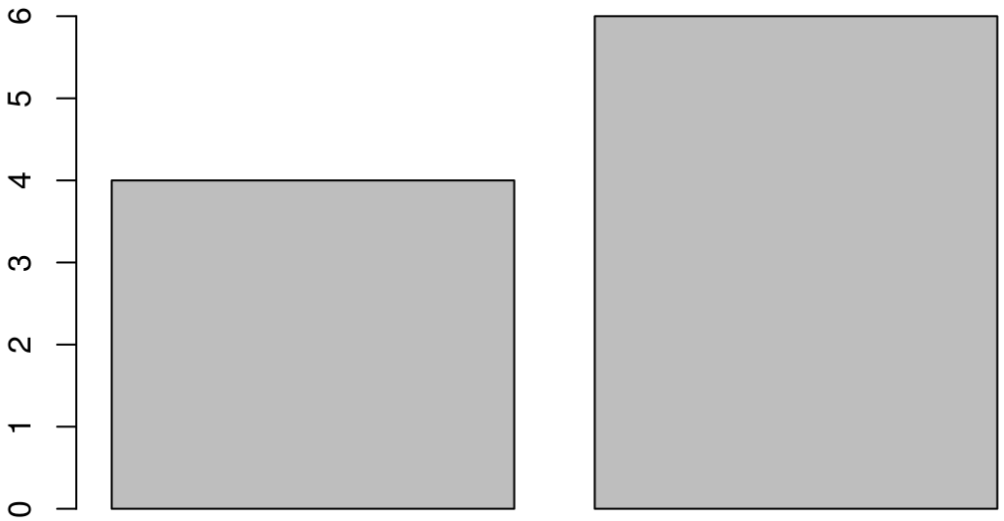
```
barplot(patients$Age)
```



Making a barplot

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

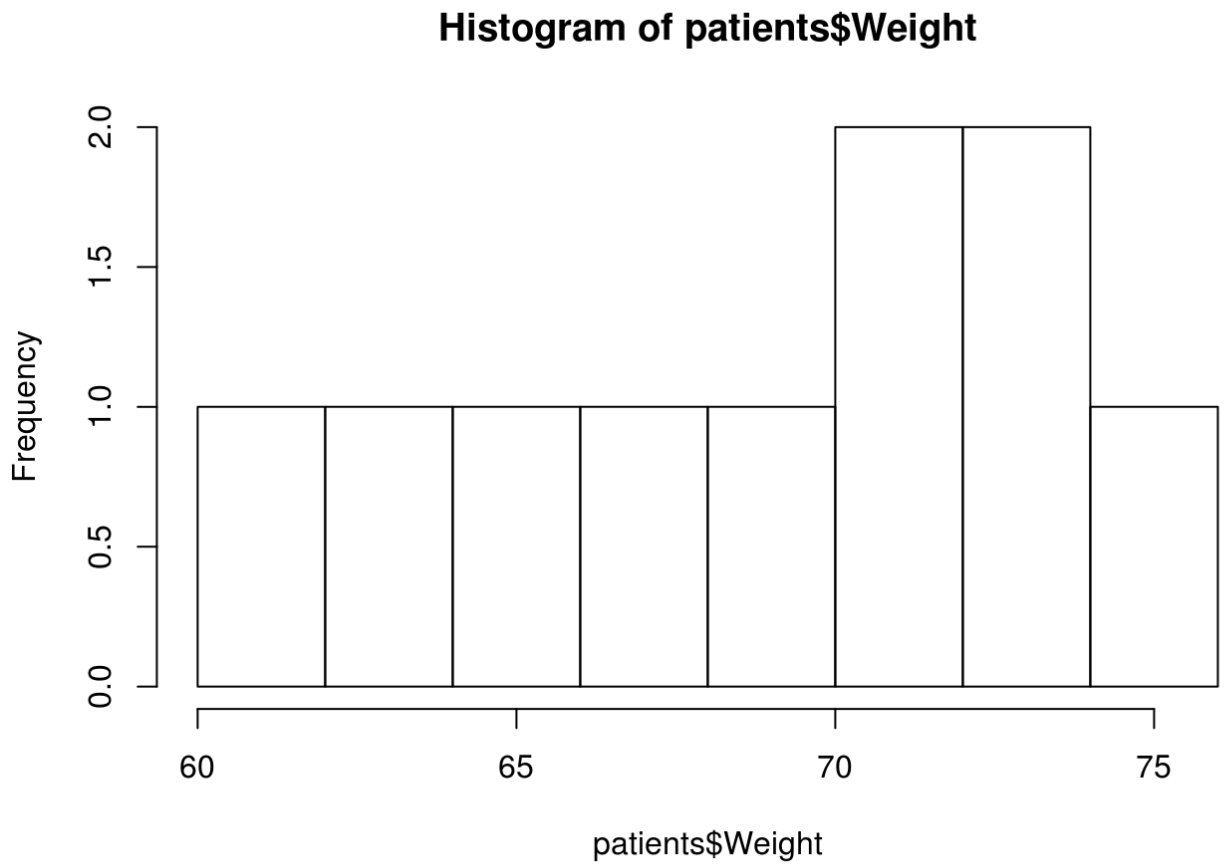
```
barplot(summary(patients$Sex))
```



Plotting a distribution: Histogram

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

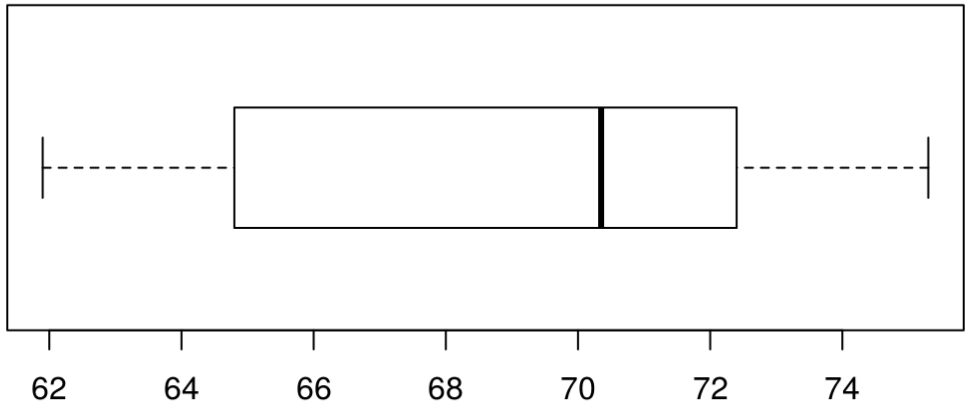
```
hist(patients$Weight)
```



Plotting a distribution: Boxplot

- The boxplot is commonly used in statistics to visualise a distribution:

```
boxplot(patients$Weight, horizontal = TRUE)
```

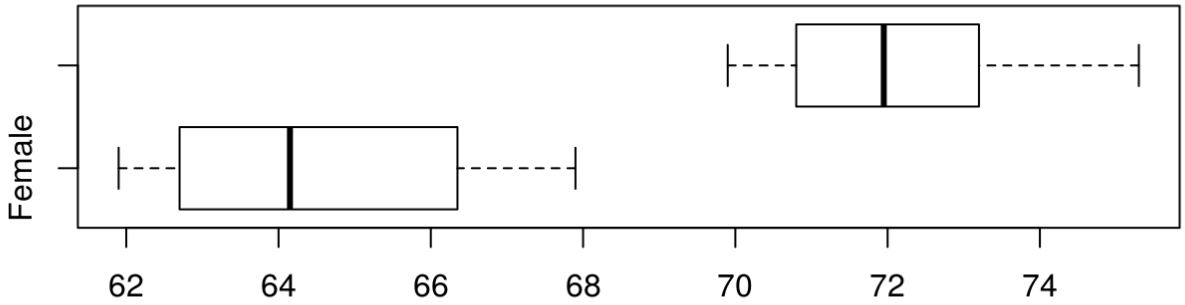


- The black solid line is the ***median***
- The top and bottom of the box are the 75th and 25th percentiles
    - Hence, the distance between these is a reflection of the *spread* of the data; the Inter-Quartile Range (***IQR***)
- Whiskers are drawn at 1.5 x IQR and -1.5 x IQR

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 ~ x` means put continuous variable `y` on the *y* axis and categorical `x` on the x axis

```
boxplot(patients$Weight ~ patients$Sex, horizontal = T)
```



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

Exercise: exercise4a.Rmd

- In the course 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)

1. Import these data into R
2. 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

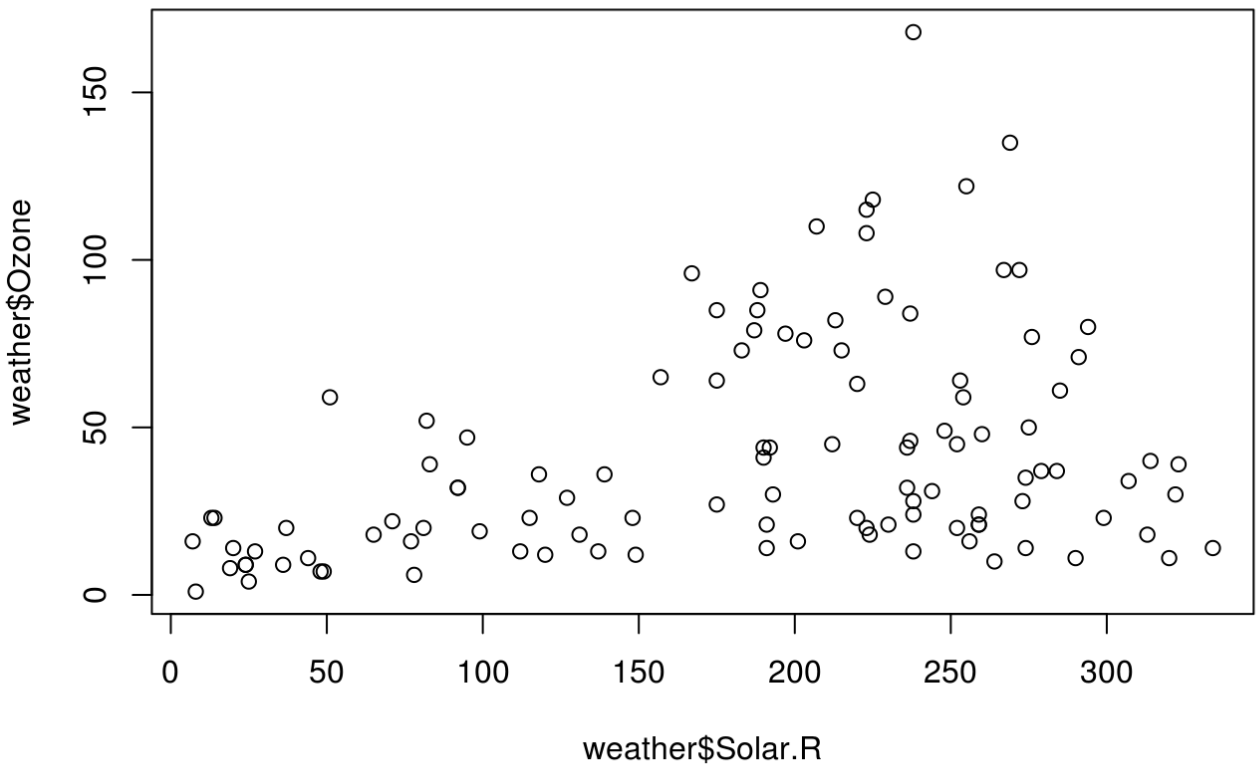
Suggestions: solution-exercise4a.pdf

```
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

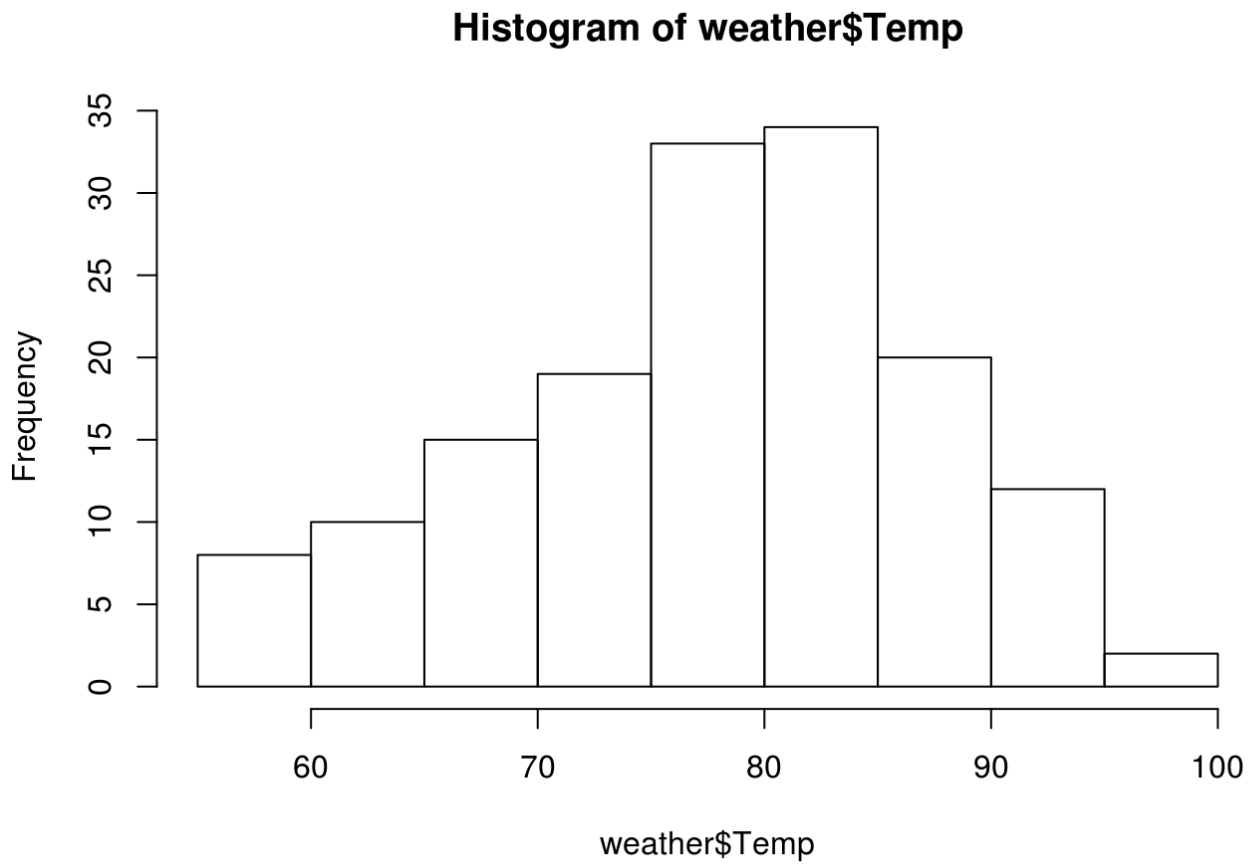
Suggestions

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



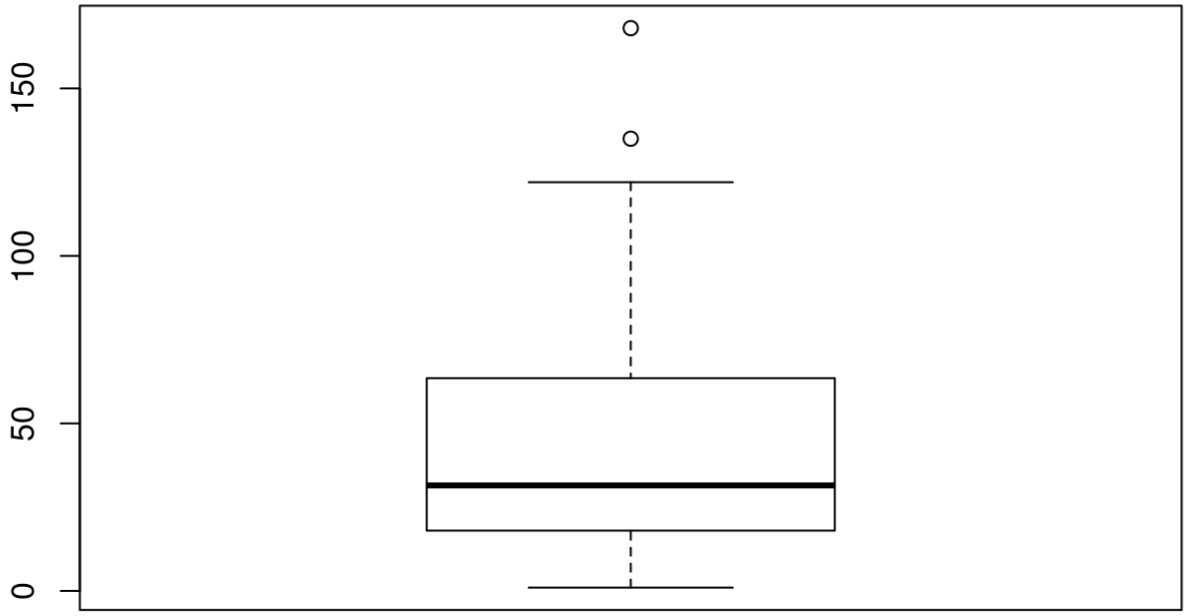
Suggestions

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



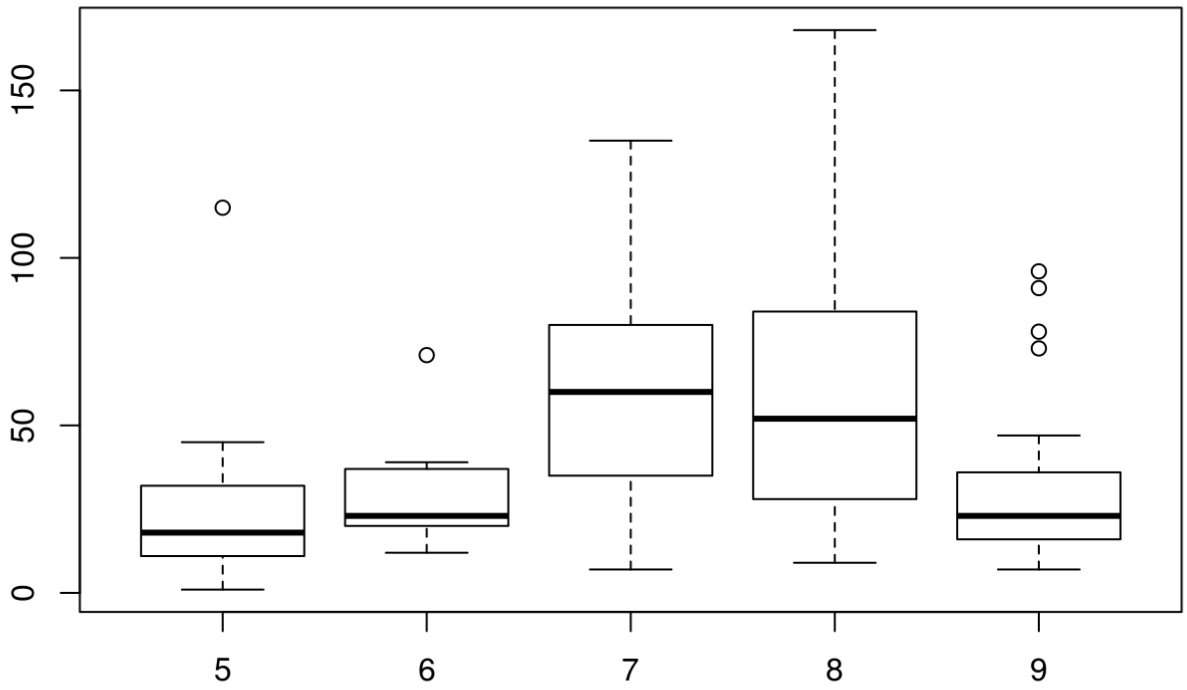
Suggestions

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



Suggestions

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

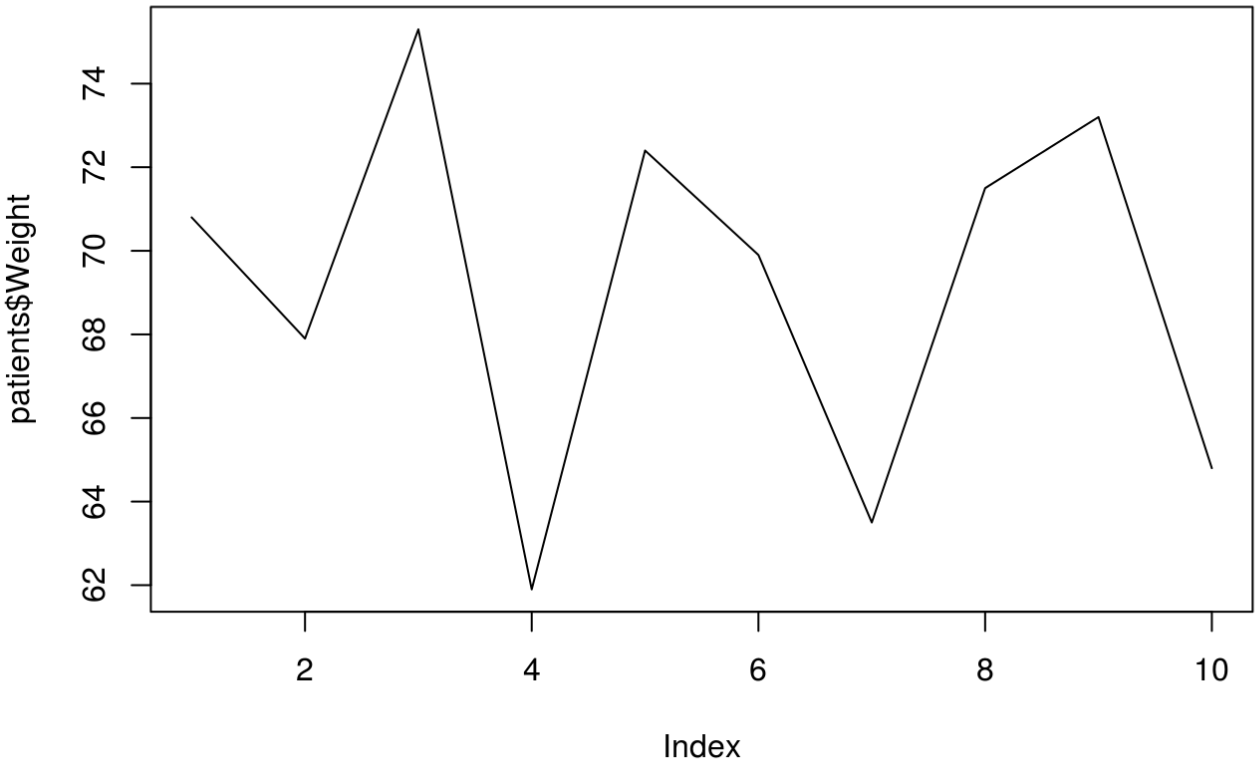


Simple customisations

- `plot()` comes with a large collection 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 remains the same

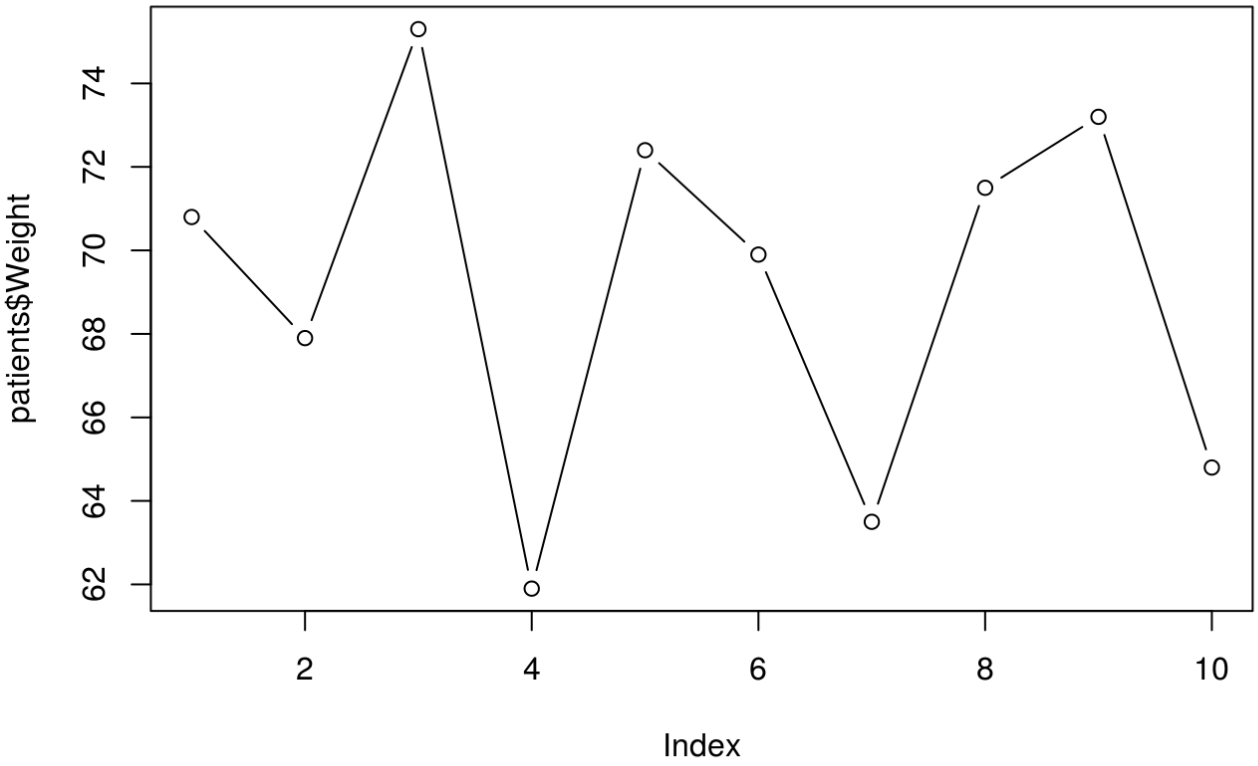
```
plot(patients$Weight, type = "l")
```



Simple customisations

- We can also have both lines and points:

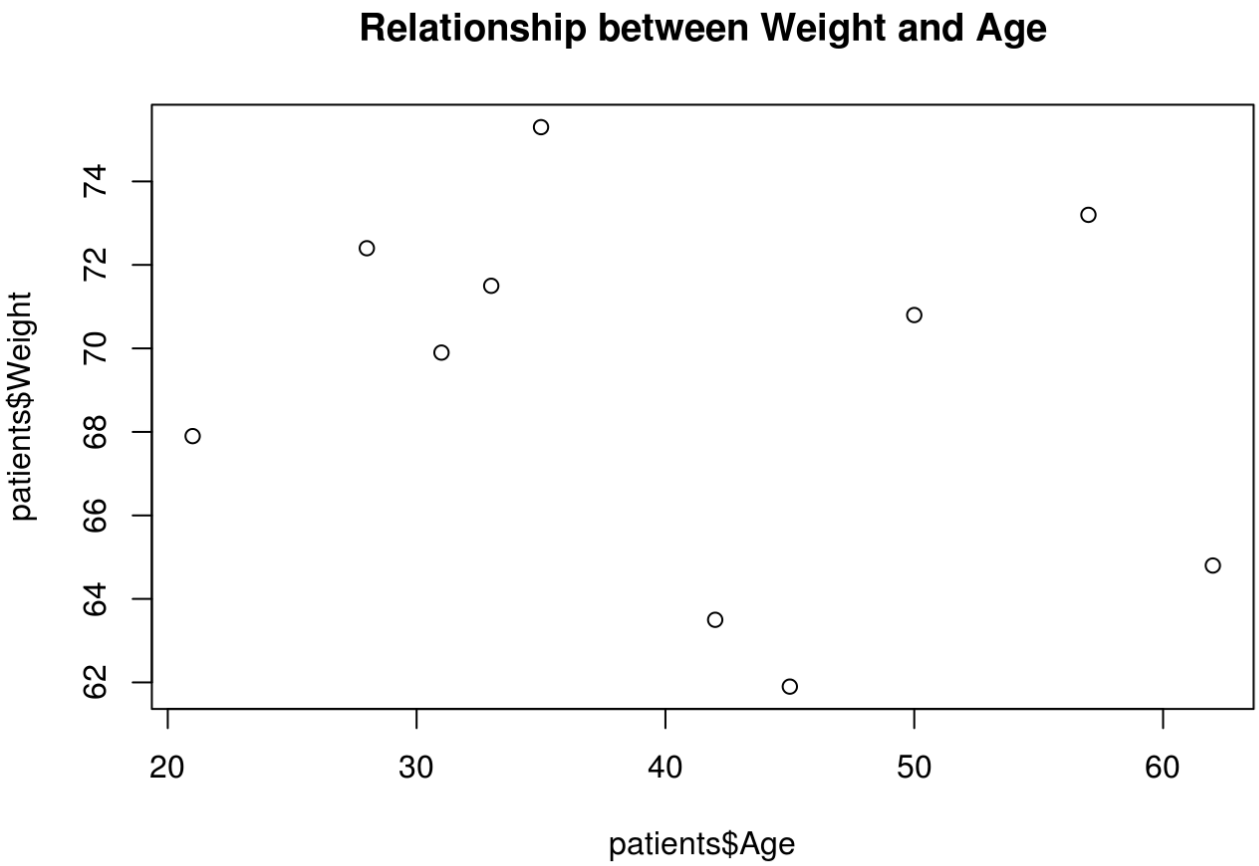
```
plot(patients$Weight, type = "b")
```



Simple customisations

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

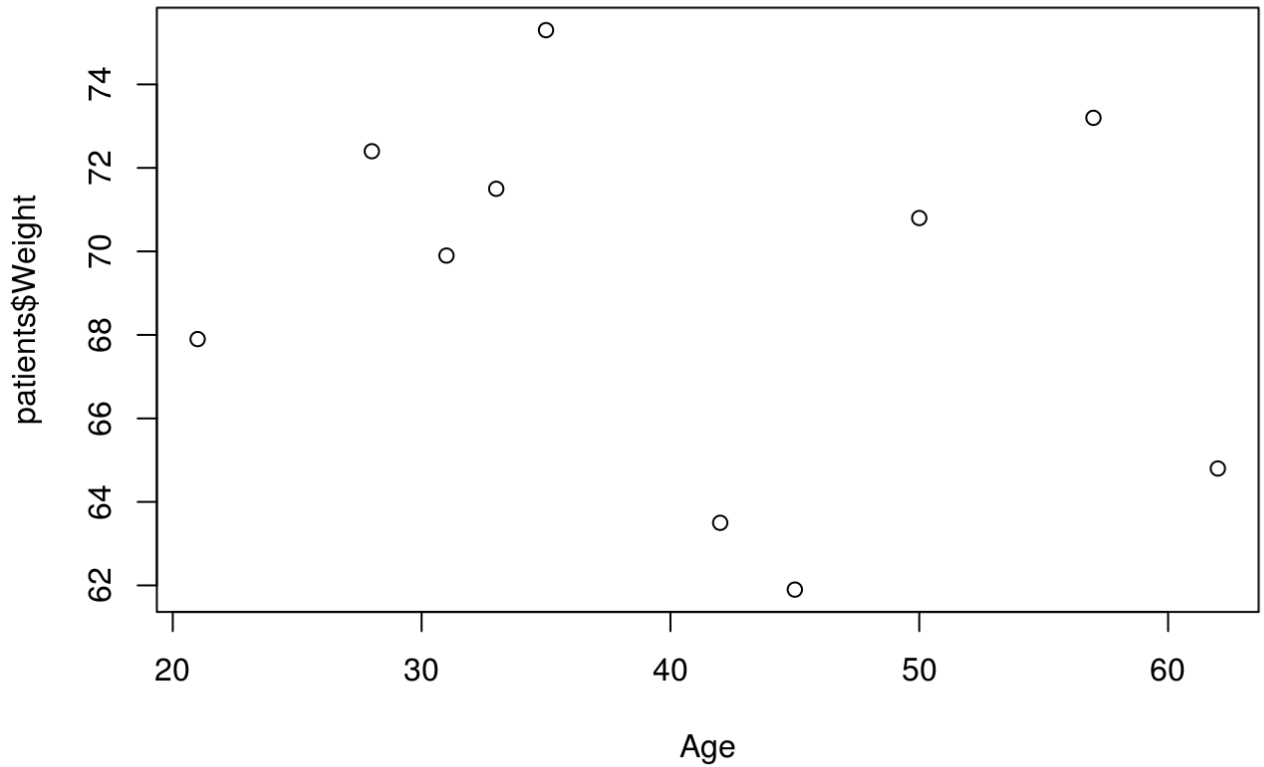
```
plot(patients$Age, patients$Weight,
     main = "Relationship between Weight and Age")
```



Simple customisations

- Adding the x-axis label:

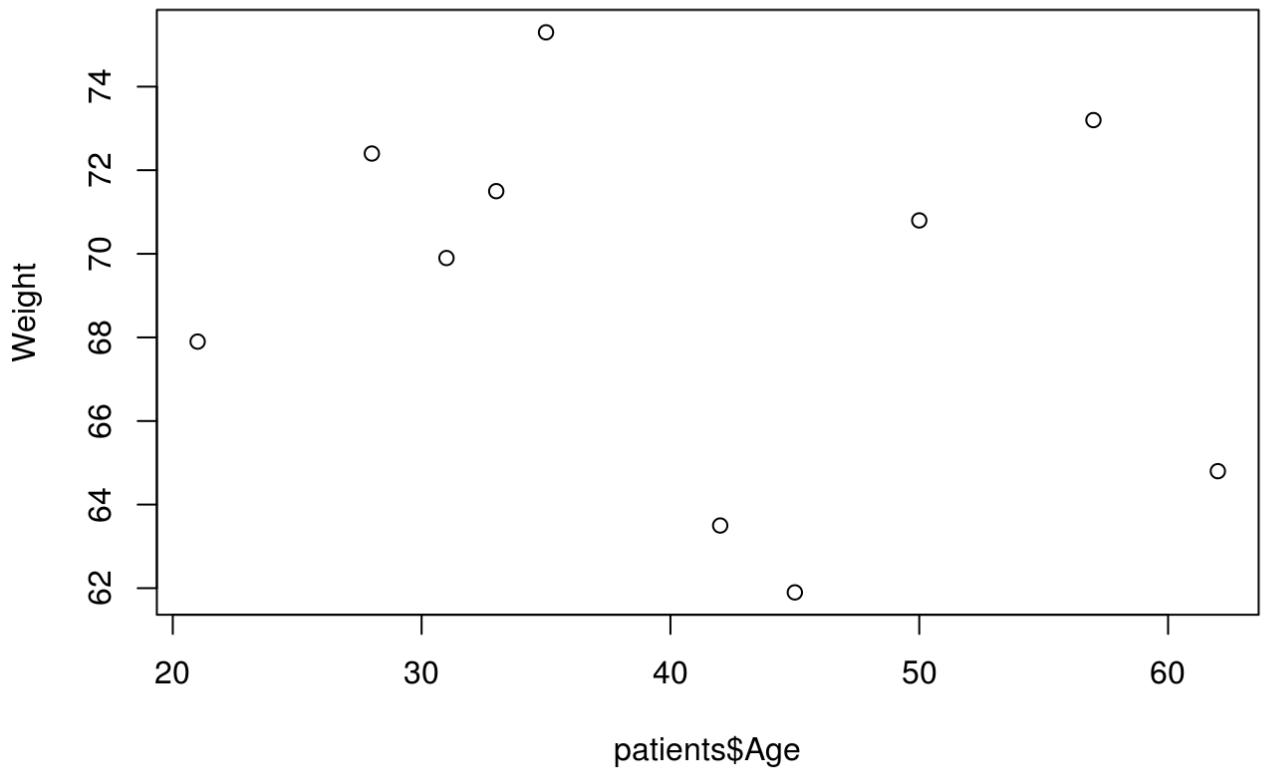
```
plot(patients$Age, patients$Weight, xlab = "Age")
```



Simple customisations

- Adding the y-axis label:

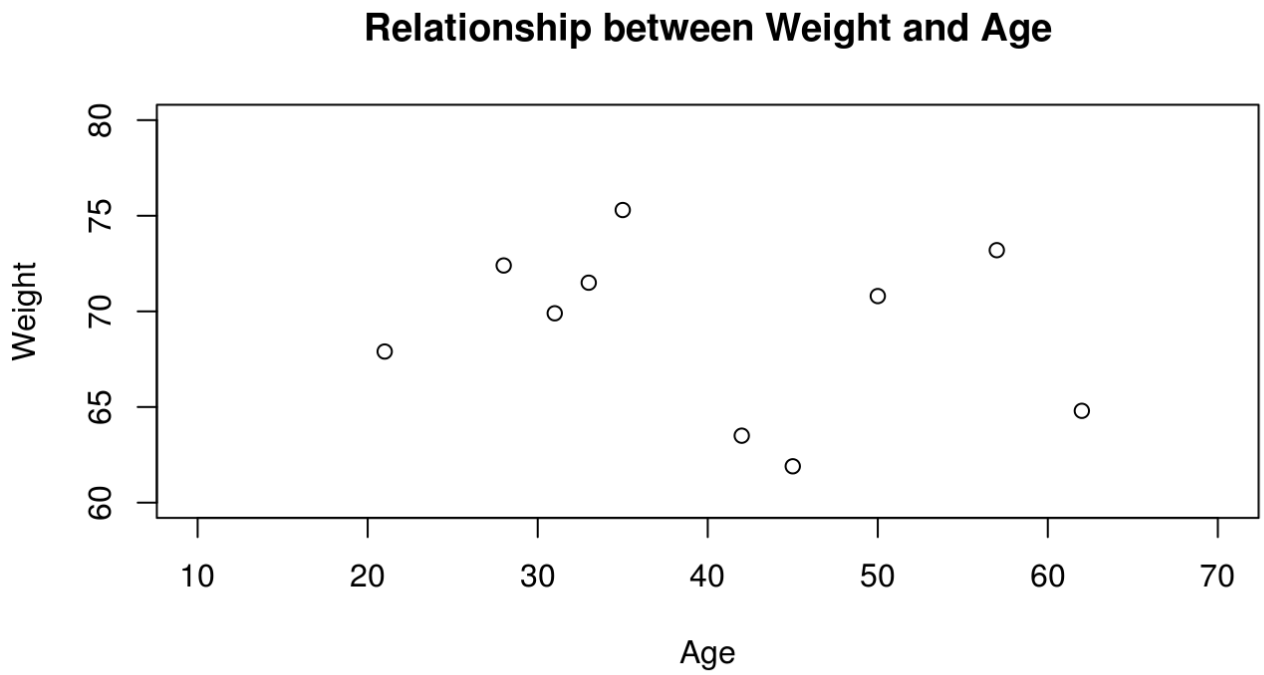
```
plot(patients$Age, patients$Weight, ylab = "Weight")
```



Simple customisations

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

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



Defining a colour

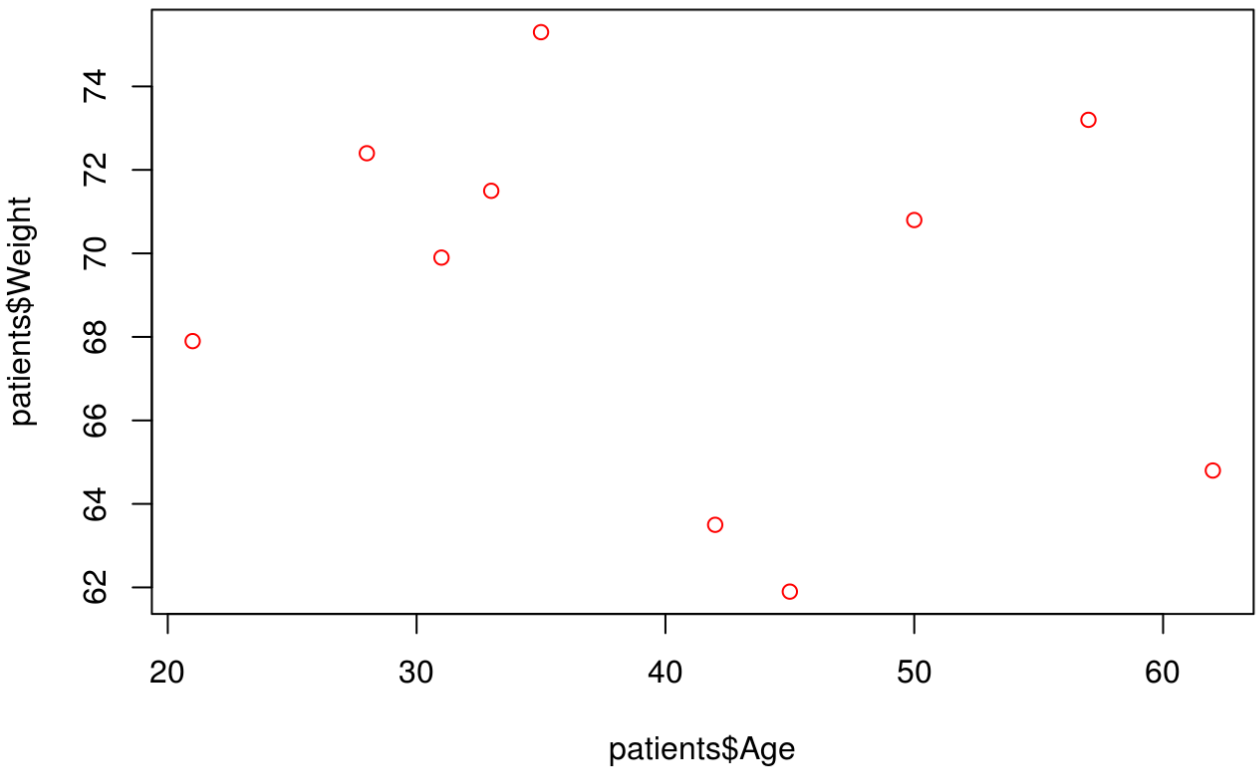
- R can recognise various strings, such as `"red"`, `"orange"`, `"green"`, `"blue"`, `"yellow"` ...
- Or more exotic ones like `grey34, gray65, grey32, rosybrown1, azure2, bisque2, gray63, tan` ...
  - See `colours()`
- See http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf (http://www.stat.columbia.edu/~tzheng/files/Rcolor.pdf)
- Can also use **R**ed **G**reen **Blue** and hexadecimal values:
  - `rgb(0.7, 0.7, 0.7)` → A light grey in RGB format`
  - `"#B3B3B3"` → The same light grey in hexadecimal

- `"#0000FF88"` → A semi-transparent blue, in hexadecimal
  - The hexadecimal system is the native colour system for screen visualisation (e.g. webs). It indicates the intensity of Red, Green and Blue by using two digits for each colour, in a scale from 0-9 and A-F (0 meaning no intensity and F meaning most intense)

Use of colours

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

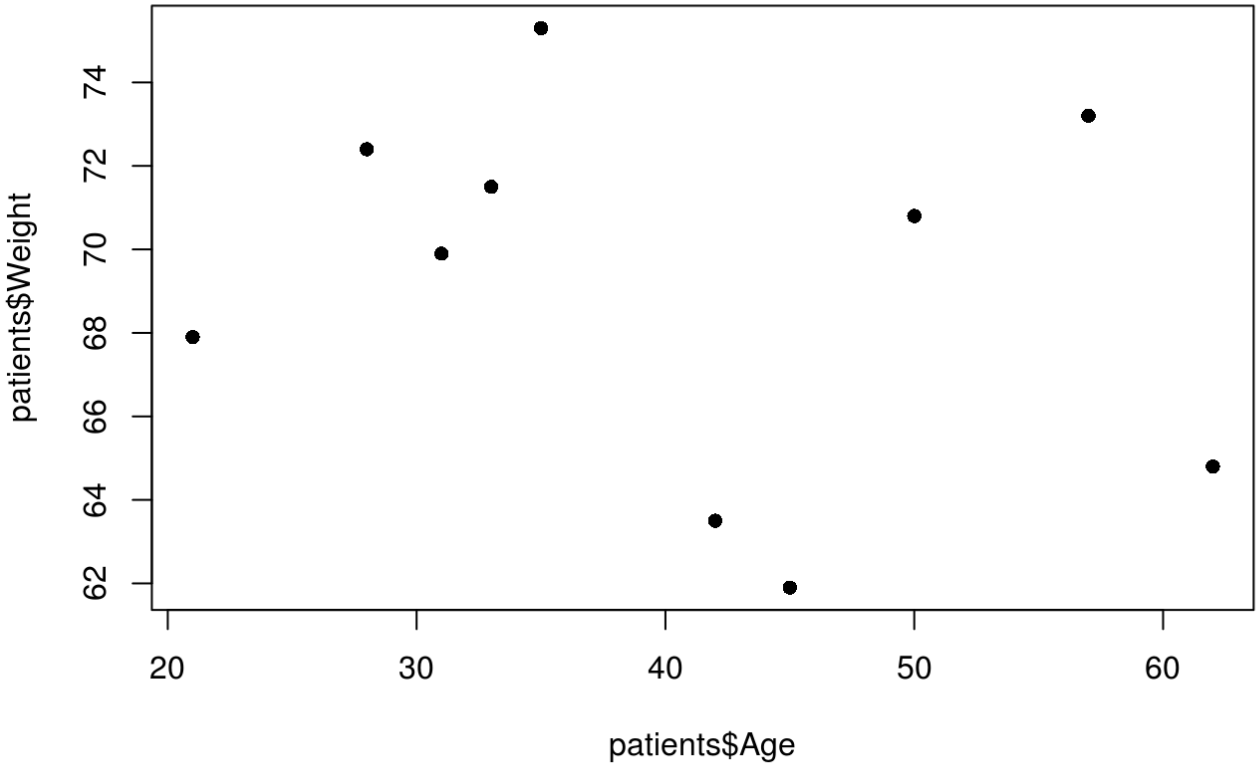
```
plot(patients$Age, patients$Weight, col = "red")
```



Plotting characters

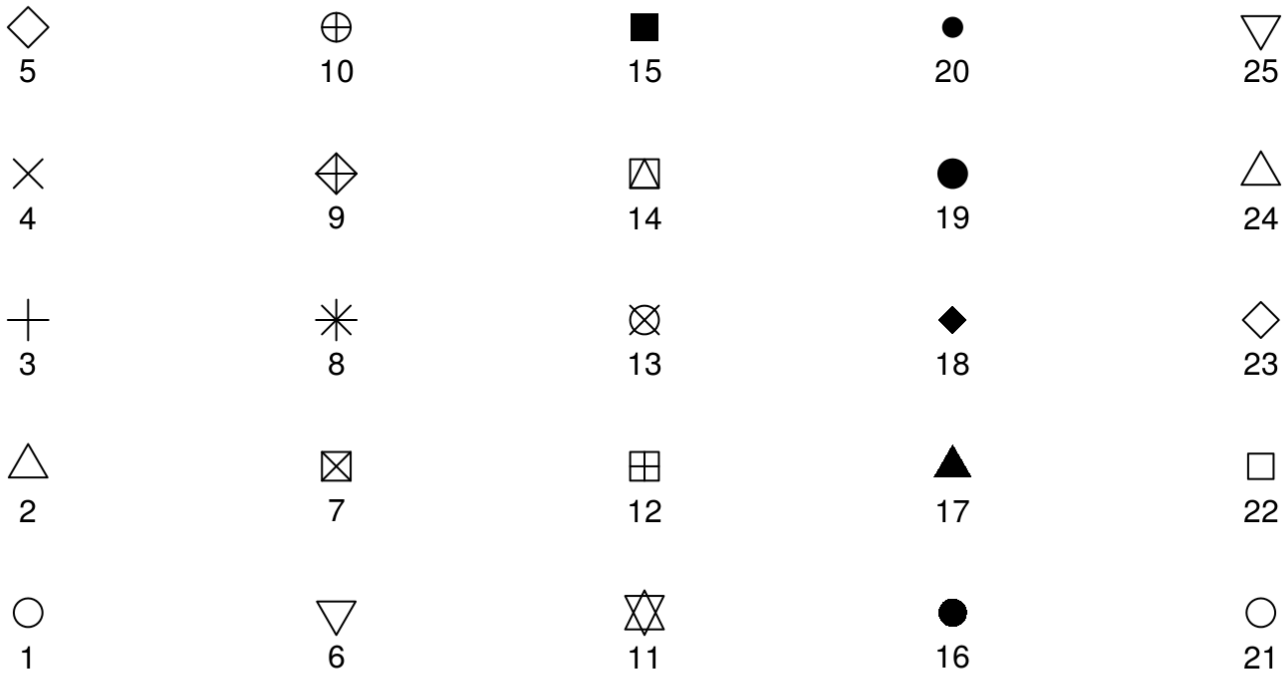
- R can use a variety of **p**lotting **ch**aracters
- Each of which has a numeric *code*

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



Plotting characters

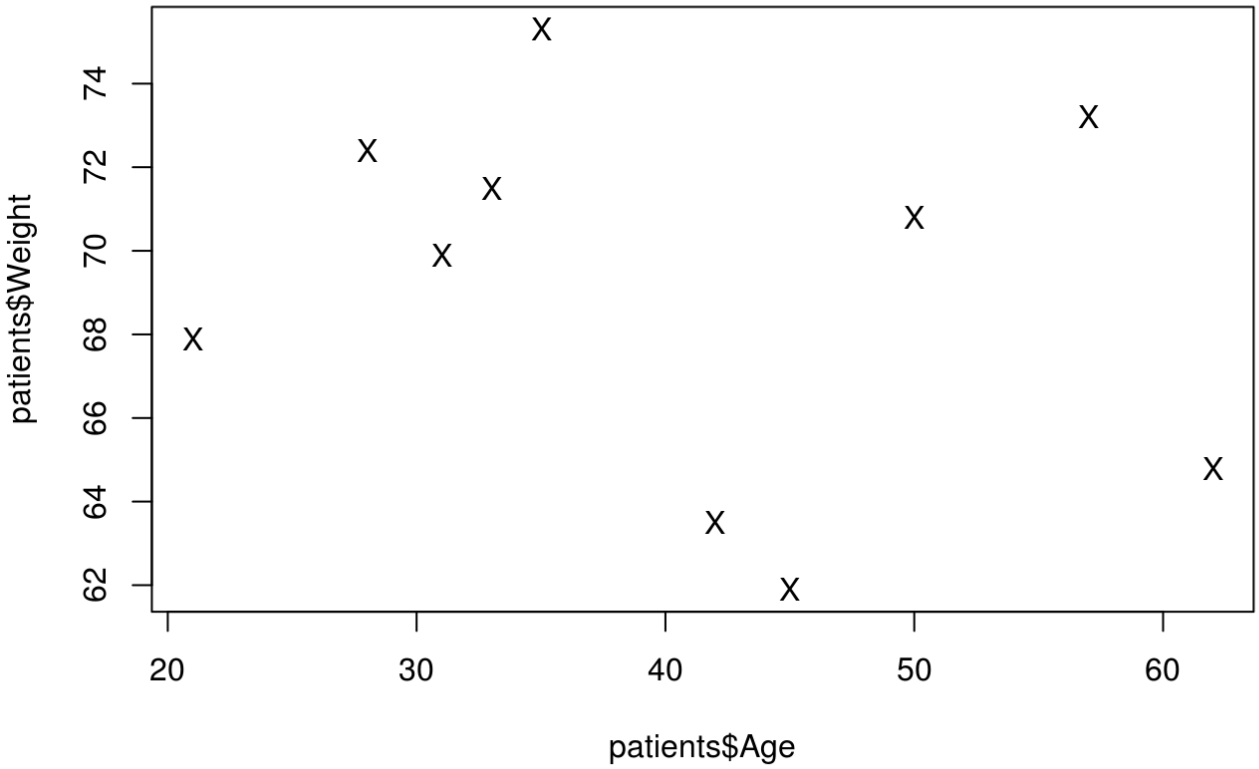
```
Warning in plot.xy(xy.coords(x, y), type = type, ...):
unimplemented pch value '26'
```



Plotting characters

- Or you can specify a character:

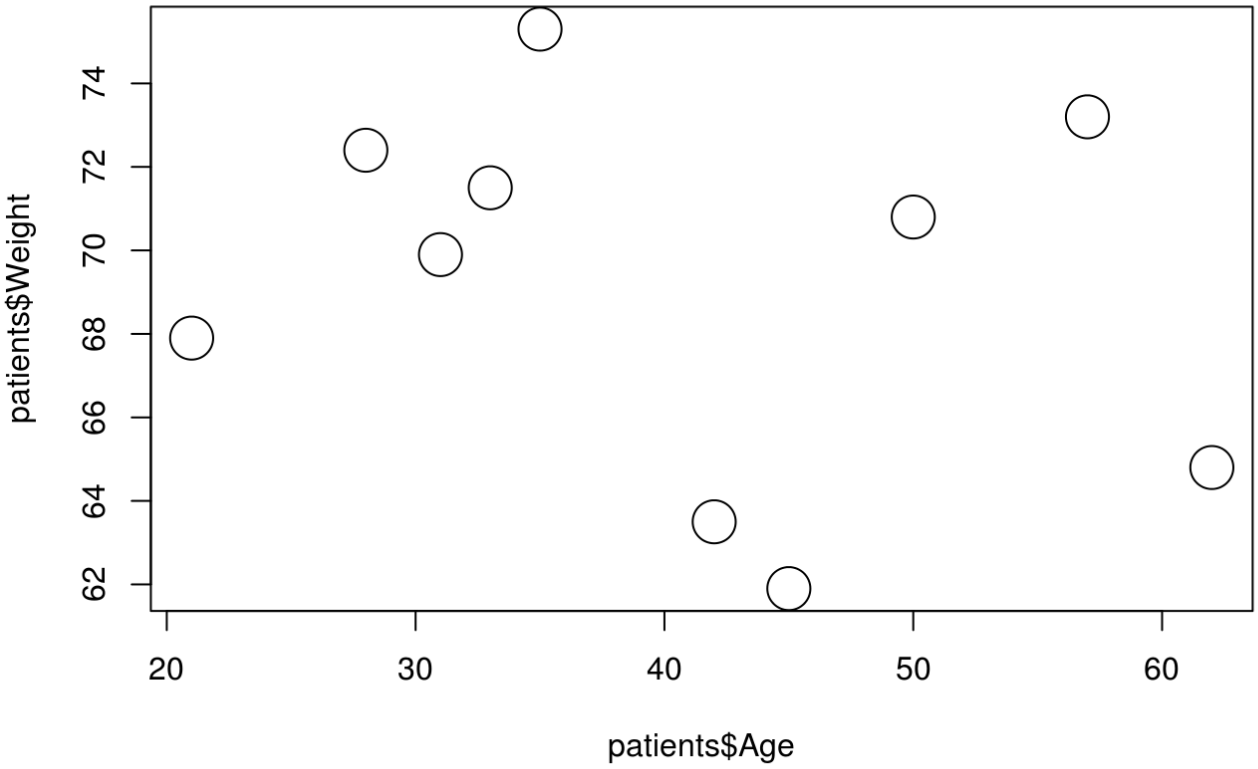
```
plot(patients$Age, patients$Weight, pch = "X")
```



Size of points

Character **ex**pansion:

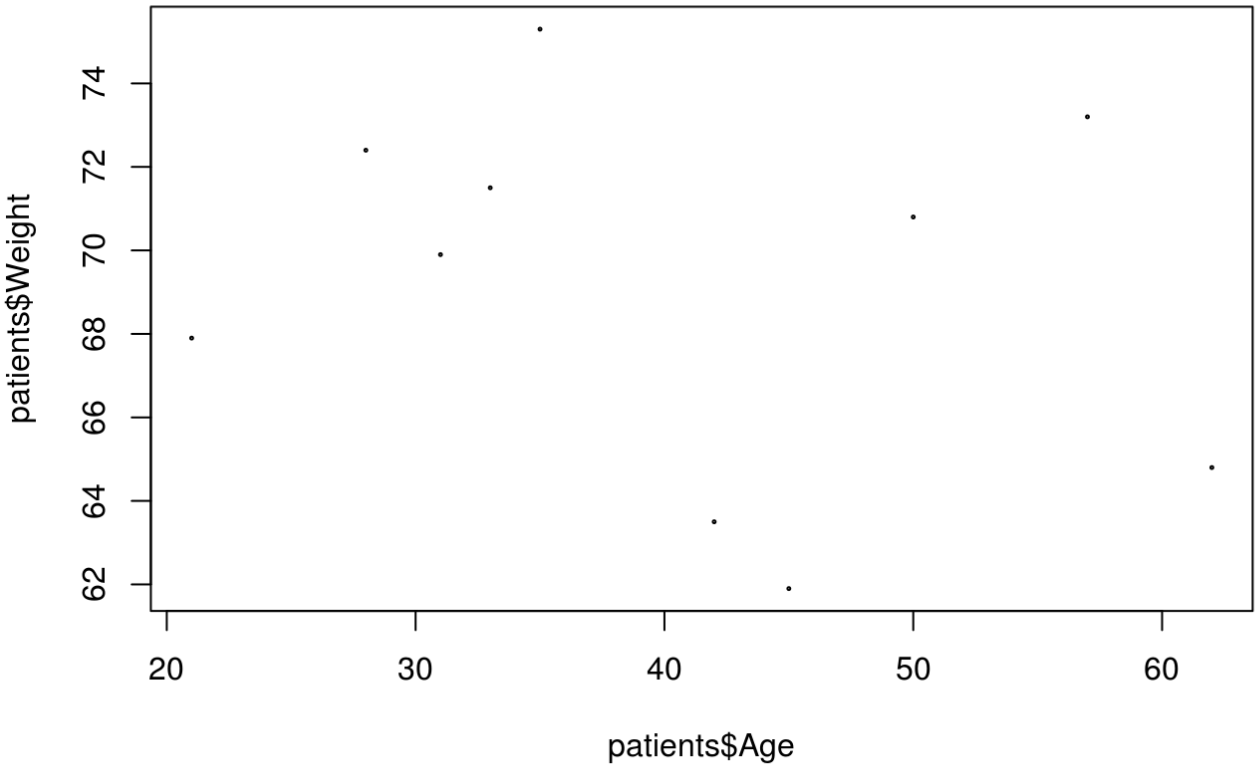
```
plot(patients$Age, patients$Weight, cex = 3)
```



Size of points

Character **ex**pansion:

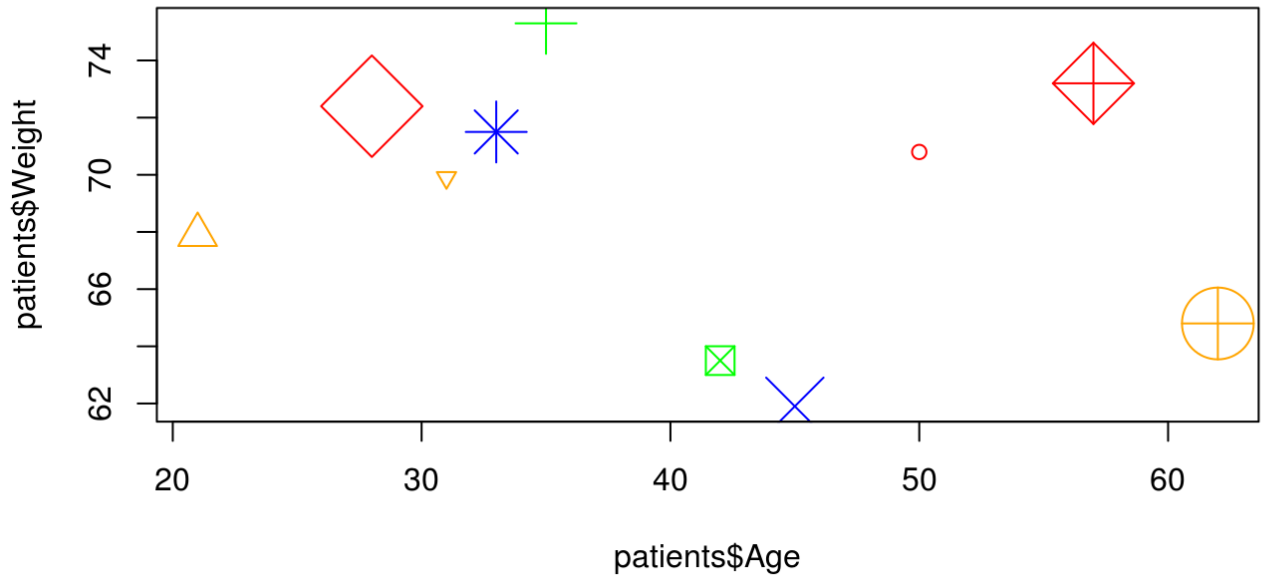
```
plot(patients$Age, patients$Weight, cex = 0.2)
```



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)

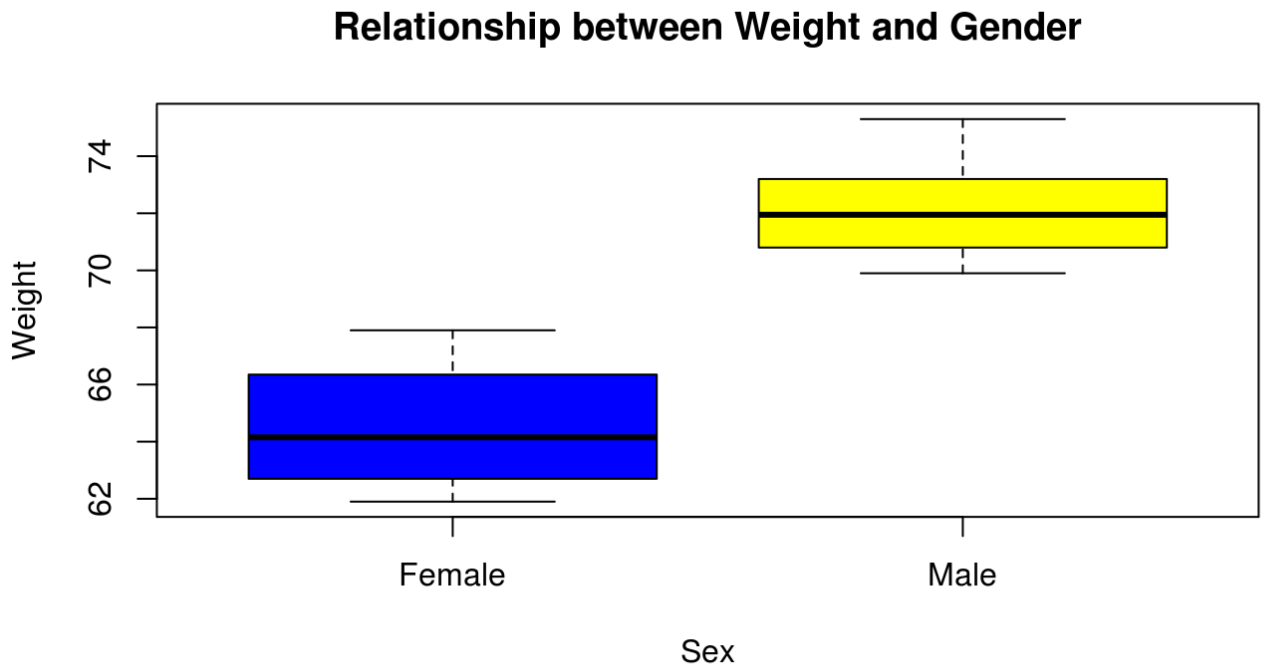
```
plot(patients$Age, patients$Weight,
     pch = 1:10, cex = 1:5,
     col = c("red", "orange", "green", "blue"))
```



Other plots use the same arguments

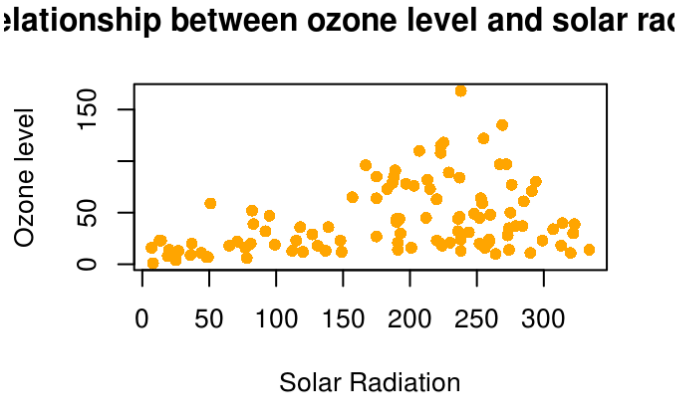
- Other plotting functions use the same arguments as `plot()`
  - technical explanation: the arguments are '*inherited*'

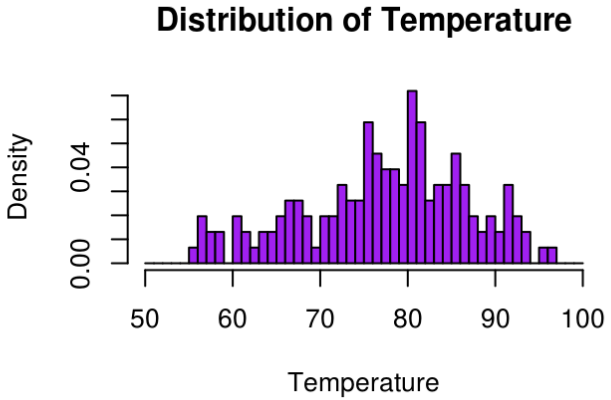
```
boxplot(patients$Weight~patients$Sex,
        xlab = "Sex",
        ylab = "Weight",
        main = "Relationship between Weight and Gender",
        col  = c("blue","yellow"))
```

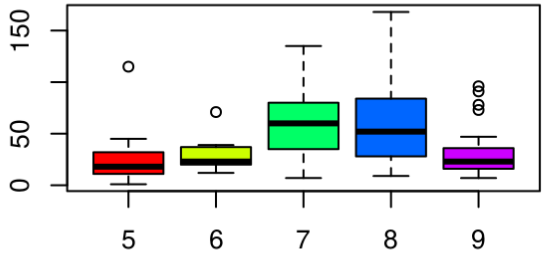


Exercise: exercise4b.Rmd

- Can you re-create the following plots? Hint:
  - See the `breaks` and `freq` arguments to hist (`?hist`) to create 50 bins and display density rather than frequency
  - For third plot, see the rainbow function (`?rainbow`)
  - Don’t worry too much about getting the colours exactly correct

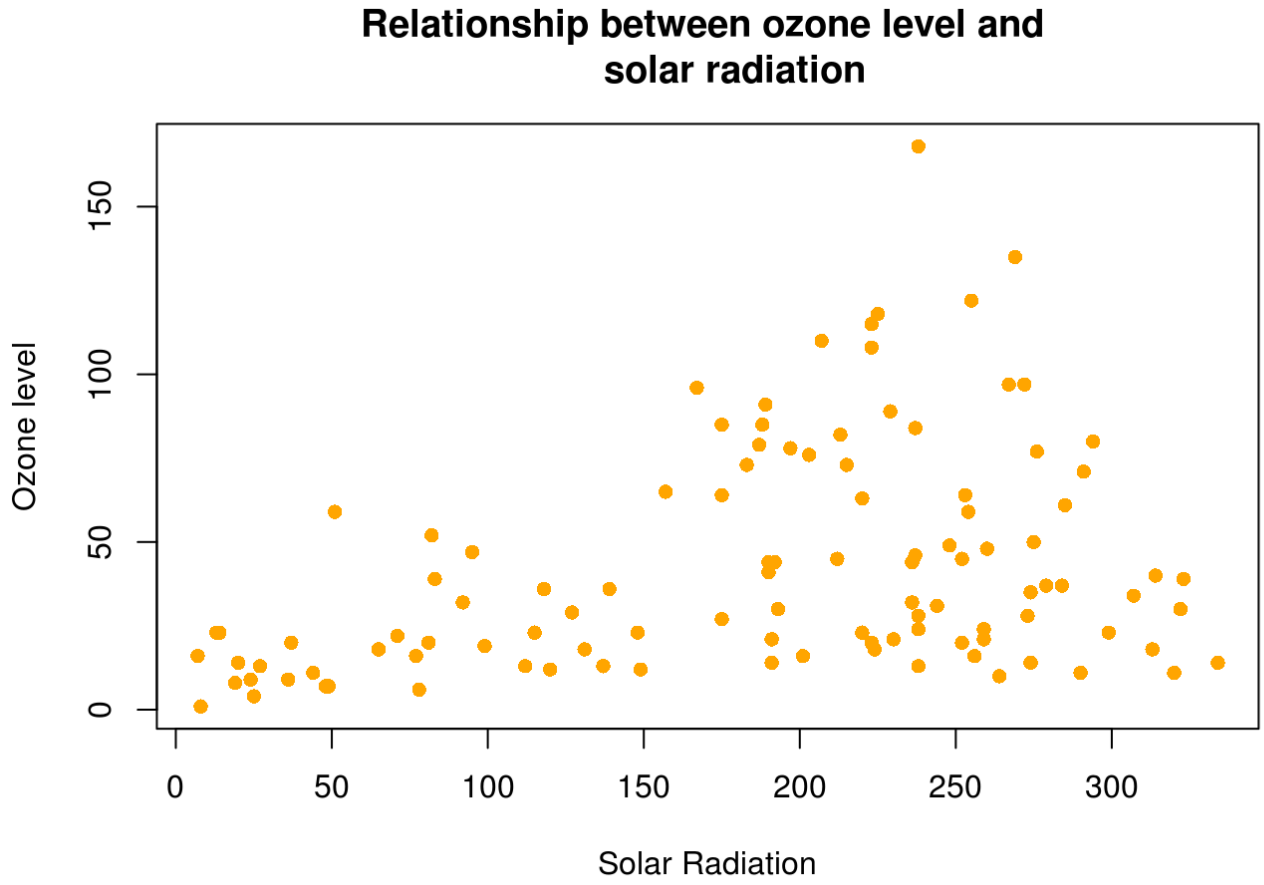






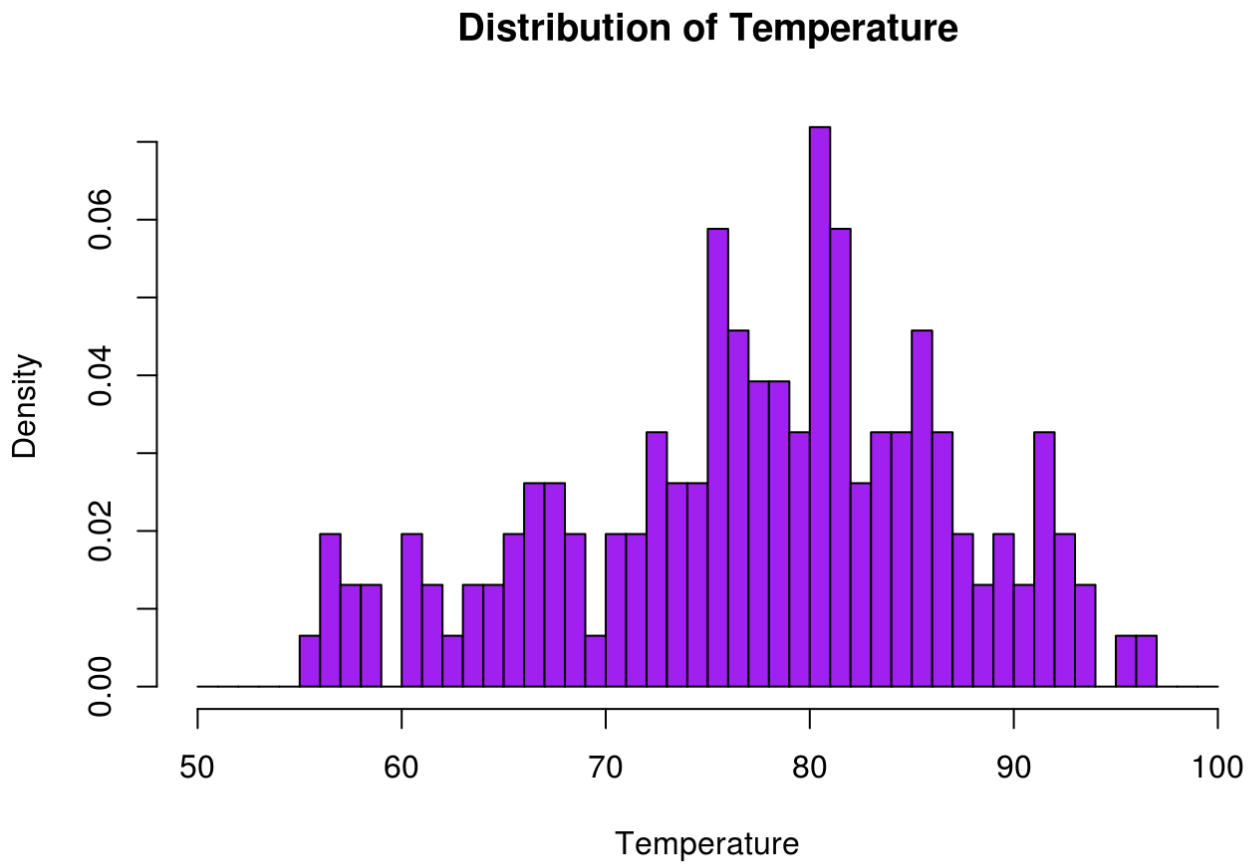
Solutions: solution-exercise4b.pdf

```
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=FALSE)
```

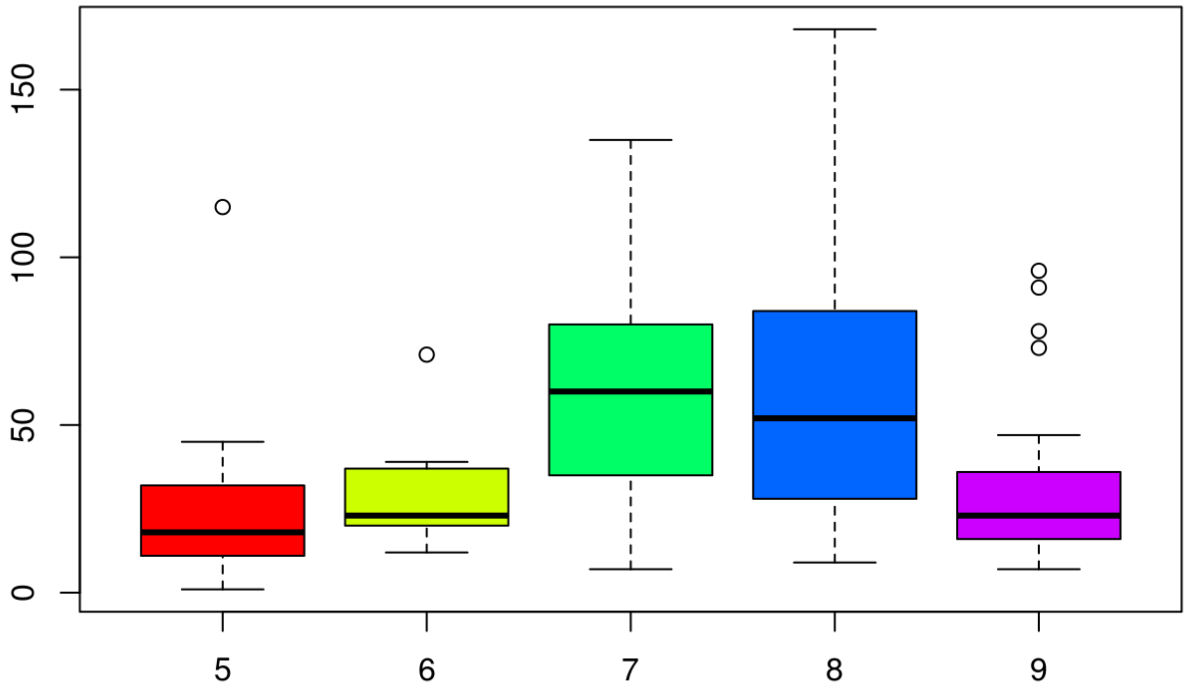


Solutions

- 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.
  - Other palette functions available: `heat.colors()`, `terrain.colors()`, `topo.colors()`, `cm.colors()`

- Red, Orange, Yellow, Green, Blue, Indigo, Violet….etc

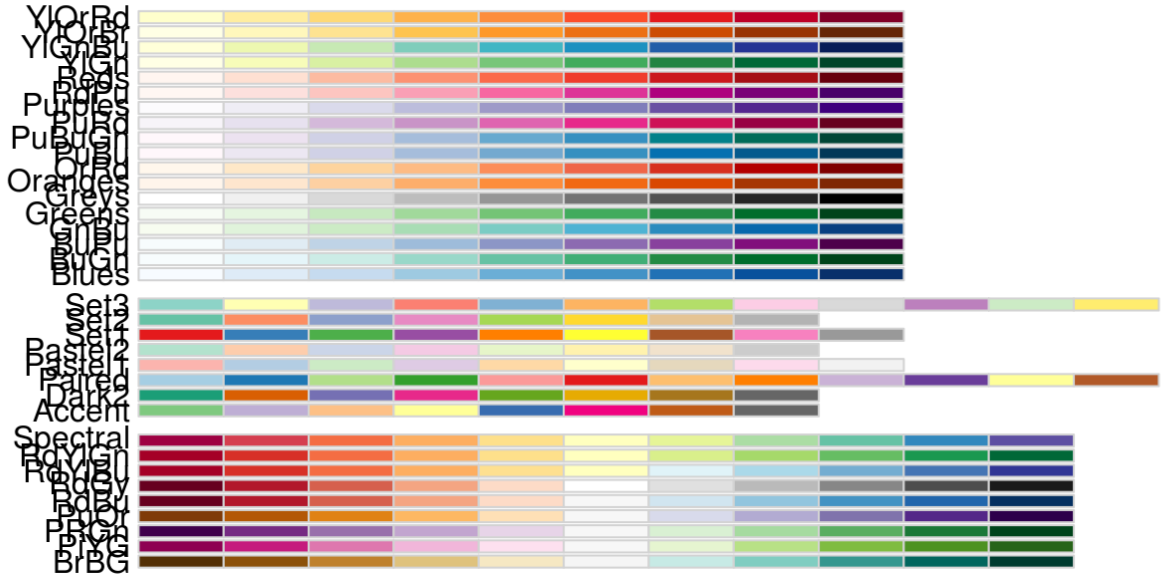
```
boxplot(weather$Ozone ~ weather$Month, col=rainbow(5))
```



Solutions

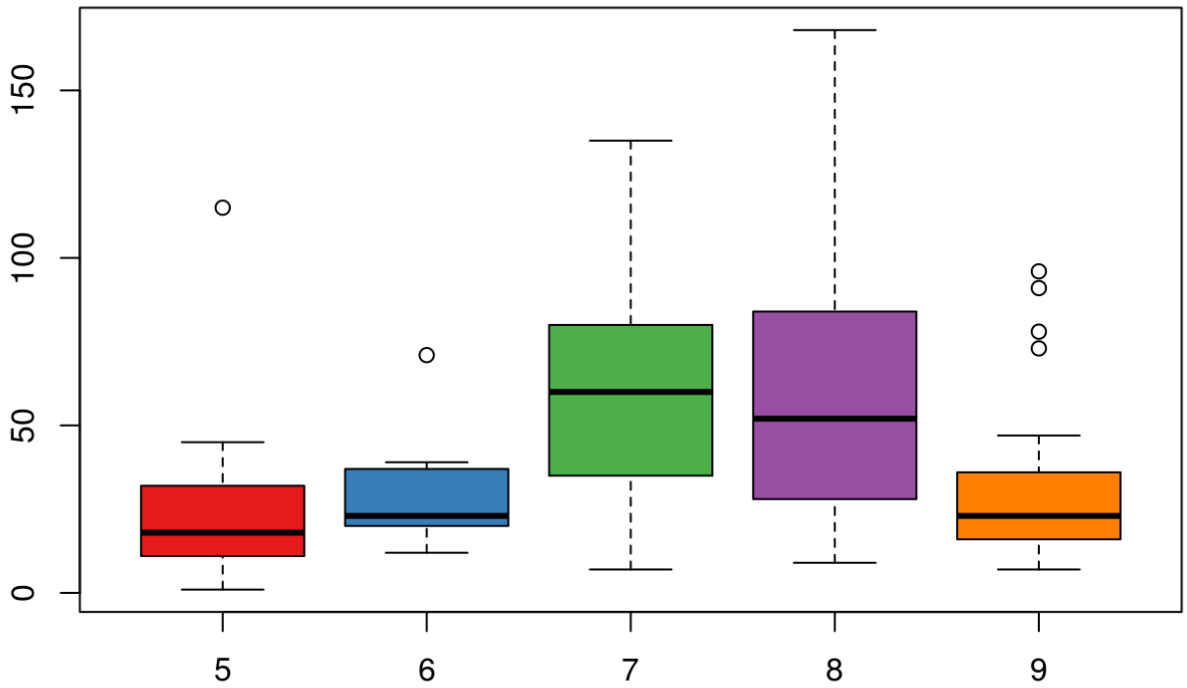
- More aesthetically-pleasing palettes are provided by the **`RColorBrewer`** package:

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



Solutions

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



End of Day 1

To come tomorrow…

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