DAY 1. A beginners guide to solving biological problems in R

Robert Stojnić (rs550), Laurent Gatto (lg390) and Rob Foy (raf51)

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

Slides by Ian Roberts and Robert Stojnić

Day 1 schedule

- The R environment and basics
  - Where to get R
  - Brief introduction to essential R
  - R help, scripting and packages

Morning coffee

- Objects and data types
  - Learn how to input and manipulate data

Lunch

- Introduction to essential R commands
  - Base functions
  - Read and write data

Afternoon coffee

- R for data analysis
  - Statistical tests and maths support

What’s R?

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

The environment and basics

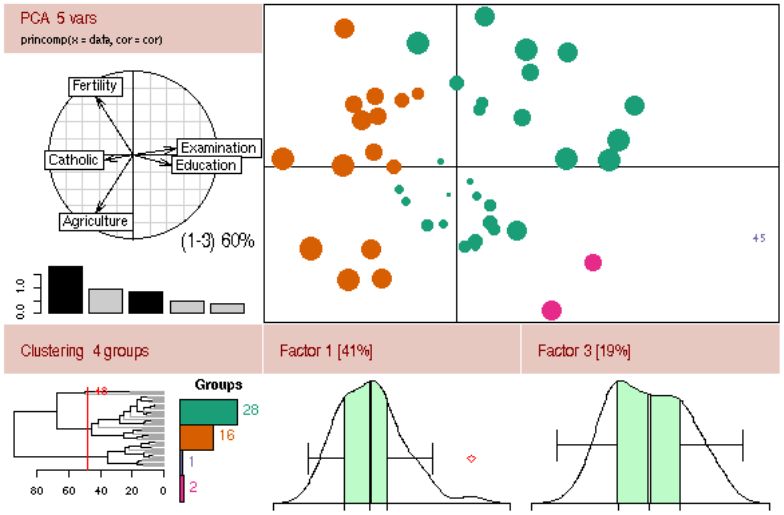
1

File Edit View History Bookmarks Tools Help

http://www.r-project.org/

Fedora Project | Ian's Web Pages | Inbox - MRC SAP P... | ScheduleWorld

The R Project for Statistical Computing



About R
- What is R?
- Contributors
- Screenshots
- What's new?

Download
- CRAN

R Project
- Foundation
- Members & Donors
- Mailing Lists
- Bug Tracking
- Developer Page
- Conferences
- Search

Documentation
- Manuals
- FAQs
- Newsletter
- Wiki
- Books
- Certification
- Other

Misc
- Bioconductor
- Related Projects
- Links

Getting Started:

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

News:

- **R 2.9.0 prerelease versions** will appear starting March 20. Final release is scheduled for April 17, 2009
- **R News 8/2** has been published on 2008-11-03.
- **DSC 2009**, The 6th workshop on Directions in Statistical Computing, will be held at the Center for Health and Society, University of Copenhagen, Denmark, July 13-14, 2009.
- **useR! 2009**, the R user conference, will be held at Agrocampus Rennes, France, July 8-10, 2009.
- **useR! 2008**, has been held at Dortmund University, Germany, August 12-14, 2008.

This server is hosted by the Department of Statistics and Mathematics of the WU Wien.

Done

www.r-project.org

Various platforms supported

- Release 2.15.0 (March 2012)
  - Base package
  - Contributed packages (general purposes extras)
  - ~4000 available packages
- Windows
  - http://www.stats.bris.ac.uk/R/bin/windows/base/R-2.11.1-win32.exe
- Mac OS (10.2 +)
  - http://cran.r-project.org/bin/macosx/
- Linux
  - http://cran.r-project.org/bin/linux/
- Executed using command line, or a graphical user interface (GUI)
  - Will demonstrate both, and use all-platform GUI, RStudio

Getting Started

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

Prepare to launch R

From command line

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

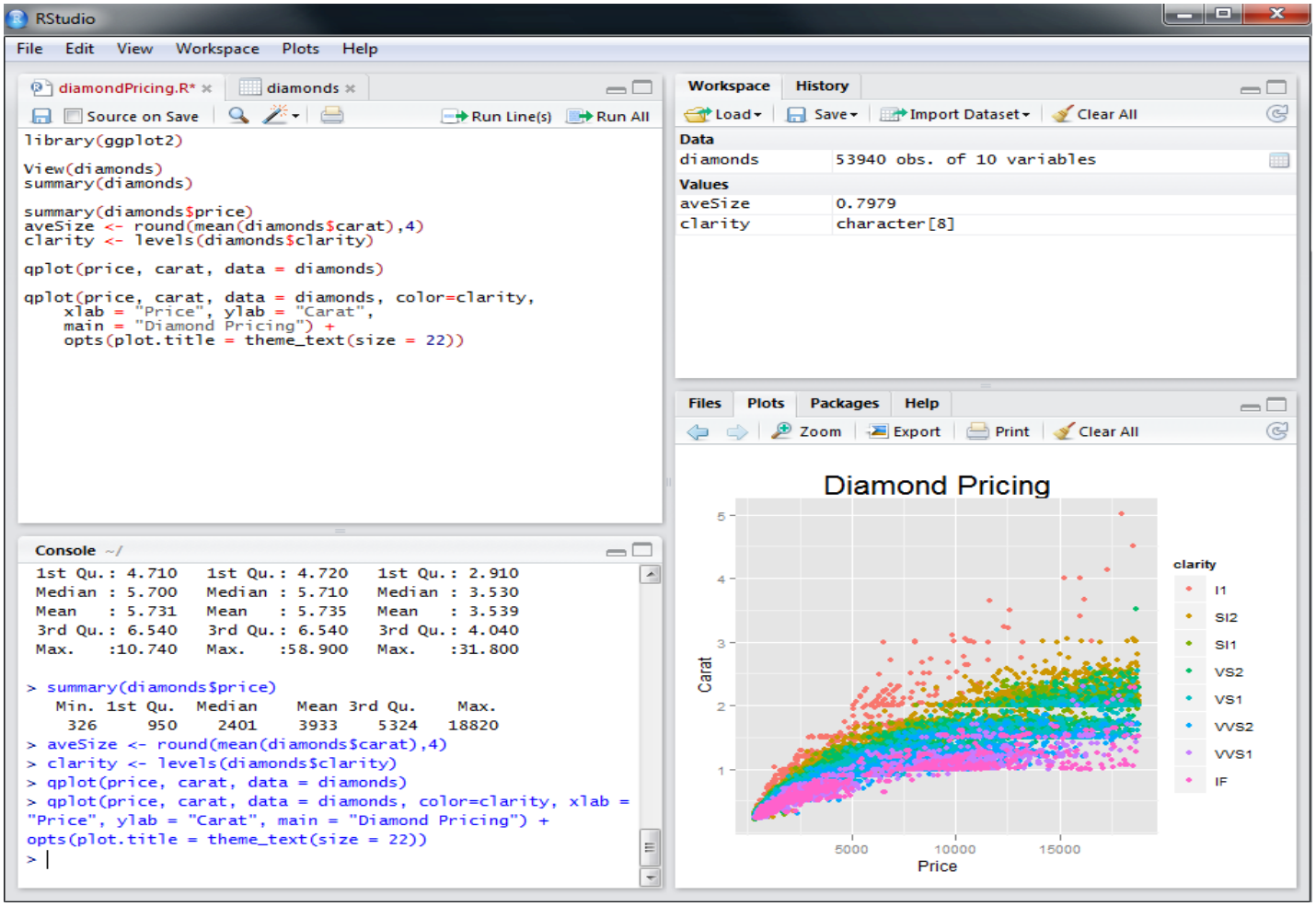
```
$ R
```

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

Prepare to launch R

Using RStudio

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



The Working Directory (wd)

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

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

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

Basic concepts in R
command line calculation

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

```
> 2 + 2
[1] 4

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

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

- Note: The number in the square brackets is an indicator of the position in the ouptut. 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 modify the contents of a variable:

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

Basic concepts in R
functions

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

```
> floor(3.142)
[1] 3
> sum(3, 4, 5, 6)
[1] 18
> max(3, 4, 5, 6)
[1] 6
> mean(3, 4, 5, 6)
[1] 3
```

- Something has gone wrong with the last function. We need to understand more about vectors...

Basic concepts in R
vectors

- The function **c()** *combines* its arguments into a **vector**

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

- As mentioned, the square brackets **[]** indicate position within the vector. 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

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

- We can now solve the problem from the previous slide

```
> mean(x)
[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)
```

- Write

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

- Using the **seq()** function...

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

- `> x`

```
 [1] 2.00000 3.33333 4.66667 6.00000 7.33333 8.66667 10.00000
```

- or the **rep()** function

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

- `> y`

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

Basic concepts in R
vectors

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

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

- We can add an element to a vector

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

- We can glue vectors together

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

Basic concepts in R
vectors

- We can remove element(s) from a vector

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

- Finally, we can modify the contents of a vector

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

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

Basic concepts in R
vector arithmetic

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

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

- Adding two vectors

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

- Vectors don't have to be the same length (what's this?)...

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

- but that doesn't always work

```
> x + 1:3  (...?)
```

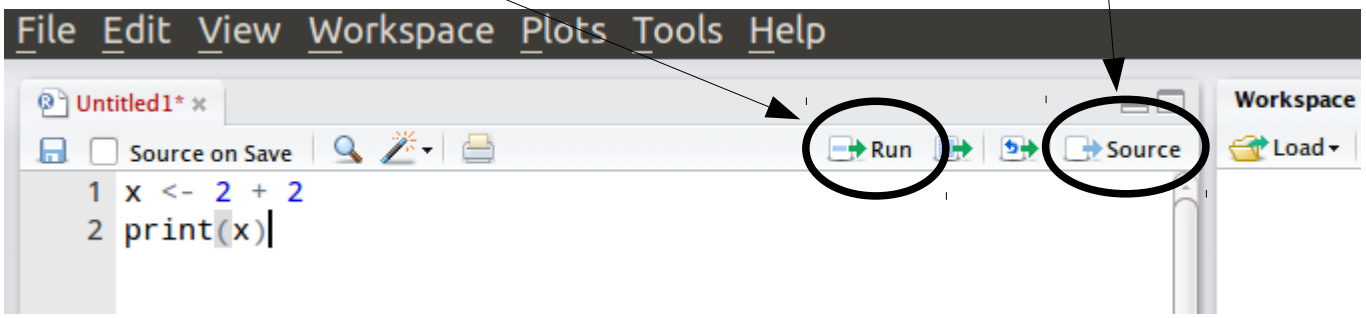
Writing scripts with Rstudio

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

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

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

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



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

Getting Help

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

  ```
  > ?seq
  ```

- This retrieves the syntax and arguments for the function. It also tells you which **package** it belongs to. There will typically be example usage
- If you can't remember the exact name type `??` followed by your guess. R will return a list of possibles
- ```
  > ??rint
  ```
  ```

Interacting with the R console

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

R packages

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

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

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

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

R packages

- 3900+ packages on CRAN:
    - Use CRAN search to find functionality you need:

    http://cran.r-project.org/search.html
    - Or, look for packages by theme:

    http://cran.r-project.org/web/views/
- 550+ packages in Bioconductor:
    - Specialised in genomics:

    http://www.bioconductor.org/packages/release/bioc/
- **Other repositiories:**
- 1000+ projects on R-forge:
    - http://r-forge.r-project.org/
- R graphical manual:
    - http://bg9.imslab.co.jp/Rhelp/

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

Exercise: Install Packages Matrix and aCGH

- Matrix is a CRAN extras package
  - Use `install.packages()` function...

```
install.packages("Matrix")
```

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

```
biocLite("aCGH")
```

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

```
library("Matrix") # loads matrix functions
library("aCGH") # loads aCGH functions
```

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

Time for ...

MORNING COFFEE

Objects

2

R stores different types of data

- Types of data
  - Logical, integer, character, floating point
- Data is stored in objects
  - Vectors, data frames, matrices, arrays, lists
  - Vector is the most basic object

- Most appropriate type and value is determined by R syntax

```
a <- 10      # takes the value of number 10
a <- "10"    # takes the value of characters "10"
a <- b       # takes the value of variable b
a <- "b"     # takes the value of character "b"
```

Data types and storage modes

- R creates appropriately sized object variables to hold data
- Objects are vectors … they have a length dimension …
  - Vectors support vector arithmetic, which is R's big thing
- The ‘mode of storage’ used is determined by the ‘data type’
- *Mode* is one of
  - logical, numeric, or character
- Standard data *types*
  - **Logical**
    - (TRUE or FALSE)
  - **Integer**
    - (e.g. whole numbers -2 / +2 Billion)
  - **Double / floating point**
    - (e.g. fractions, scientific expressions)
  - **Character string - always in quotes**!
    - (e.g. 4 is a number, "4" or "abc" are characters)

If vectors are generated to hold numbers, and type isn’t specified, they will be held as mode numeric, type double.

```
> typeof(…)
> mode(…)
```

Exercise
Data types & storage modes

- Create vectors `i, l, s` and `d` of length 20 for each *data type* integer, logical, character, and double
- Examine their storage mode & confirm data type
  - `mode(...)`
  - `typeof(...)`
- Tips:
  - you can do this by manually assigning values, or by generating them (see the functions below)
  - `sample(20)` will create a vector of 20 random integer values
  - R special character object `letters` is 'builtin', you can subset it
  - `runif(20)` will generate 20 random uniformly distributed 'double' values between 0 and 1
  - relationships are 'tests' and return logical values (e.g. 2>1 is TRUE, but 5<1 is FALSE)
    - `l <- i >= 10 # test if 'i' is greater or equal to 10`

Solution

Example code:
04_objects.R script

```
> i <- sample(20)
> i
[1]  6  1 14  8  5 10 18 12  2 11 16  3  4 20  9  7 19 13 15 17
> s <- letters[i]
> s
[1] "f" "a" "n" "h" "e" "j" "r" "l" "b" "k" "p" "c" "d" "t" "i" "g" "s" "m" "o"
[20] "q"
> l <- i >= 10
> l
[1] FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE
[13] FALSE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> d <- runif(20)
> d
 [1] 0.292480127 0.586057481 0.812791710 0.189530051 0.634724719 0.882981055
 [7] 0.358151866 0.003088843 0.900198085 0.905947217 0.032231749 0.330297215
[13] 0.398237377 0.832774598 0.048503020 0.965920822 0.357567181 0.688554482
[19] 0.728918707 0.477420883
> mode(i)
[1] "numeric"
> typeof(i)
[1] "integer"
> mode(l)
[1] "logical"
> typeof(l)
[1] "logical"
> mode(s)
[1] "character"
> typeof(s)
[1] "character"
> mode(d)
[1] "numeric"
> typeof(d)
[1] "double"
```

l is mode logical , type logical

t is mode character, type character
(note the quotes in object t)

d is a mode numeric,type double

Storage modes & data types

- Data types - why care?
  - May get an undesired result if calculations are between numbers stored as different types
  - R will coerce data types when calculations between differing types are forced
    - If the operation is inappropriate, the calculation will fail.

      e.g.

      ```
      > 2 + "2"
      ```

      will fail as we cannot add a character string to integer!

R Objects - vectors

c(...) or vector(..., mode=…, length=…)

- Vectors
  - one-dimensional sequence of data items of only one data type

```
> a <- c(1, 2, 3, 4, 5)
> typeof(a)
[1] "double"

> aa <- c(1, "2", 3)
> typeof(aa)

[1] "character"
```

Vector arithmetic is a fundamental R concept

```
> a <- 1:10 ; b <- 101:110 ; c <- a + b
> c
[1] 102 104 106 108 110 112 114 116 118 120
```

```
a:   1   2   3   4   5   6   7   8   9  10
b: 101 102 103 104 105 106 107 108 109 110
C: 102 104 106 108 110 112 114 116 118 120
```

Arithmetic operation occurs between corresponding items of the two vectors. If vectors are different lengths, shortest is recycled

R Objects

data.frame(var1=…, var2=…, etc.)

- Data frames
  - multiple columns of vectors that form a table
  - columns can be of different data types

```
> a <- 1:10                    # shortcut for numbers 1 to 10
> b <- letters[1:10]           # shortcut for letters
> c <- month.name[1:10]        # shortcut for month names
> d <- data.frame(a, b, c, stringsAsFactors=FALSE)  # data frame assignment
> d
      a    b   c
1     1    a   January
2     2    b  February
3     3    c     March
4     4    d     April
5     5    e       May
6     6    f      June
7     7    g      July
8     8    h    August
9     9    i September
10    10   j   October
```

  - Column names can changed

    `colnames(d) <- c("a", "b", "c")`
  - Address columns with $ notation
    - `d$a # returns only the column a (as a vector)`

R Objects

matrix(..., ncol=…, nrow=…)

- Matrices
  - Like data frames, but for a single data type.

```
> 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
```

  - Matrices are usually associated with numeric data, and usual operations +, -, *, ... work on whole matrices as well

Indexing data frames, matrices & arrays

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

- Works just like vectors, only in 2 dimensions

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

```
a <- data.frame(1:10, letters[1:10], month.name[1:10], stringsAsFactors=FALSE)
names(a) <- c("numbers","letters","months")
a
   numbers letters    months
1        1       a   January
2        2       b  February
3        3       c     March
4        4       d     April
5        5       e       May
6        6       f      June
7        7       g      July
8        8       h    August
9        9       i September
10      10       j   October
```

Direct reference row & column

```
a[1,1]
[1] 1
```

Direct reference of rows and columns, range notation

```
a[1:4,1:2]
  numbers letters
1       1       a
2       2       b
3       3       c
4       4       d
```

Omit a value to get all of them, e.g. full first column

```
a[, 1]
[1] 1 2 3 4 5 6 7 8 9 10
```

R Objects

`list(name1=obj1, name2=obj2, …)`

- Lists
  - Store collections of R objects.
    - List members (items) can be of any R object, or data type.

```
a <- 1:10
b <- matrix(runif(100),ncol=10,nrow=10)
c <- data.frame(a,month.name[1:10])

myList<-list( ls.obj.1=a, ls.obj.2=b,ls.obj.3=c )
summary(myList)
names(myList)
```

  - Using dollar notation `($)`, we can address list items directly, by name, as with columns of a data frame

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

  - Alternatively, `myList[[1]]` to get first item in the list

R Objects

`factor(obj, levels=…, labels=…)`

- Factors
  - Factors store categorical data
  - categorical data is usually expressed in levels.
    - They are especially useful where repetition is found
    - Exercise may take place on any day of the week

```
The factor is exercise, and levels are week day names
```

    - A gene may be deleted, lost, normal, gained or amplified

```
The factor is gene copy number, and levels are -2, -1, 0, 1, 2
```

  - Factors require a good understanding of data dependency
  - Experiments often have multiple explanatory variables, and it is interesting to observe interactions between them

Factors are similar to 'enumerated data types' in other languages

Operators

- arithmetic

  +, -, *, /, ^

(equal to, not equal to)

- comparison

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

- logical

  !, &, |, xor

these always return logical values ! (TRUE, FALSE)

([!] not, [&] and, [bar, shift-back slash] or, [xor]exclusive or)

Advanced indexing / subscripting vectors

- Indexing / subscripting
  - The process of referencing a particular data value stored in an R object
- Indexing can be achieved either with numbers or logicals, e.g.:

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

Or, with expression resulting in either numbers or logicals:

```
> a <- 1:5

> s[a[1:4]] # range expression, answer is a to d

> s[a<3] # first 2 items, returns a, b

> s[a>1 & a<3] # answer is 2, returns b
```

Exercise: objects & indexing
Make a phonebook with 10 listings

Construct your phonebook from 5 vectors (make up the data).

First name

Character vector: `firstName`

Second name

Character vector: `secondName`

Telephone number

Numeric vector: `telNumber`

Listed in directory

Logical vector: `notListed`

`i.e. TRUE or FALSE`

Full name

Character vector: `fullName`

```
fullName <- paste(firstName,secondName)
```

Combine vectors into a single `phoneBook` dataframe

Use your phone book to extract the following data items

1. Telephone numbers of people with second names beginning with letters below M

```
phoneBook$secondName < "M"
```

2. Telephone numbers of people not listed in a directory

```
phoneBook$notListed == TRUE
```

3. Telephone numbers of all odd row people

```
seq(1,10,2)
```

Some tips for the indexing expressions are given.

One solution

"+" Line wrap indicator

```
> firstName<-c("Adam","Eve","John","Mary","Peter","Paul","Luke",
+     "Matthew","David","Sally")
> length(firstName)
[1] 10
> secondName<-c("Tiny","Large","Small","Davis","Thumb","Daniels",
+     "Edwards","Smith","Howkins","Dutch")
> length(secondName)
[1] 10
> telNumber<-c(111111,222222,333333,444444,555555,666666,777777,888888,121212,232323)
> length(telNumber)
[1] 10
> notListed<-c(TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE,TRUE,FALSE)
> length(notListed)
[1] 10
> phoneBook<-data.frame(firstName,secondName,paste(firstName,secondName),telNumber,notListed,
                        stringsAsFactors=FALSE)
> names(phoneBook)<-c("First_Name","Second_Name","Full_Name","Tel_Number","Not_Listed")
> phoneBook
   First_Name Second_Name     Full_Name Tel_Number Listed
1        Adam        Tiny     Adam Tiny     111111   TRUE
2         Eve       Large     Eve Large     222222  FALSE
3        John       Small    John Small     333333   TRUE
4        Mary       Davis    Mary Davis     444444  FALSE
5       Peter       Thumb   Peter Thumb     555555   TRUE
6        Paul     Daniels  Paul Daniels     666666  FALSE
7        Luke     Edwards  Luke Edwards     777777   TRUE
8     Matthew       Smith Matthew Smith     888888  FALSE
9       David     Howkins David Howkins     121212   TRUE
10      Sally       Dutch   Sally Dutch     232323  FALSE
```

```
> typeof(firstName)
[1] "character"
> typeof(secondName)
[1] "character"
> typeof(telNumber)
[1] "double"
> typeof(notListed)
[1] "logical"
```

```
> phoneBook$Second_Name<"M"
 [1] FALSE  TRUE FALSE  TRUE FALSE
      TRUE  TRUE FALSE  TRUE  TRUE
> phoneBook[phoneBook$Second_Name<"M",4]
[1] 222222 444444 666666
      777777 121212 232323

> phoneBook[phoneBook$Not_Listed,4]
[1] 111111 333333 555555 777777 121212

> phoneBook[seq(1,10,2),4]
[1] 222222 444444 666666 888888 232323
```

Odd

05_phoneBook.R script

Time for

LUNCH
(RESUME 1:30 PM)

R Commands & flow control

3

Basic R 'Built-in' functions for working with variables

- list & remove objects

```
ls(), rm()
rm(list=ls()) # get rid of everything
```

- Add rows or columns to a data frame, ***df***. Row bind, column bind

```
rbind(df,...), cbind(df,...)
```

- Remove a row, or column, from a data frame.

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

- Sort a data frame. There are three functions, ***rank***, ***sort*** and ***order***. Order is the hardest one to understand, but is best suited to reordering data frames.

```
phoneBook[order(phoneBook$secondName) , ]
```

- Missing values, uses ***is*** family of functions
- (NA is different from "want to record missing values")

```
is.na(…)
```

- Names of objects

```
names(…)
colnames(...)
rownames(...)
```

- Return length of an object, number of rows or columns of a dataframe or matrix

```
length(…)
nrow(…)
ncol(…)
```

A note on data sort ...

```
A<-1:7
B<-c("Mon","Tue","Wed","Thur","Fri","Sat","Sun")
names(A)<-B

sort(A) ; sort(names(A))
order(A); order(names(A))
```

sort() returns the sorted data. order() returns a permutation vector showing how to reorder the data
Use sort() to sort a single vector, order() to sort multiple dependent vectors (e.g. columns in a dataframe)

Basic R 'Built-in' functions for working with matrices

- Adding rows and columns to matrix or data frame
  - Make a sample matrix from vector

```
x <- matrix(x, ncol=4) # converts vector X to matrix, with 4 columns and 5 rows
y <- matrix(y, ncol=4) # converts vector Y to matrix, with 4 columns and 5 rows
```

- Addition of rows with rbind

```
xx <- rbind(x,y)
```

- Addition of columns with cbind

```
yy <- cbind(x,y)
```

- Examine the output

```
xx
yy
```

- Arithmetic with matrices
  - Means & medians of values by row

```
rowMeans(xx) ; rowMedians(xx)
```

  - Means & medians of values by column

```
colMeans(yy) ; colMedians(yy)
```

  - Name rows and columns

```
rownames(xx) <- c(...)
colnames(yy) <- c(...)
```

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

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

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

  - Sum of values in a series

```
sum(x)
```

  - Average estimates (mean / median)

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

  - Range of values in a series

```
range(x)
```

  - Correlation, variance & covariance, of series (e.g. heights & yields) of vectors

```
var(x)    # variance of x
cor(x,y)  # correlation of x and y
cov(x,y)  # covariance of x and y
```

- Arithmetic with vectors
  - Rank ordering

```
rank(x) # returns positions (placement) of
   elements
```

  - Quantiles

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

  - Tukey's 5 number summary

```
fivenum(x)
```

  - Square Roots

```
sqrt(x)
```

  - Standard deviations

```
sd(x)
```

  - Median average distance

```
mad(x)
```

  - Trigonometry functions

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

```
x <- sample(10000,20)/rnorm(20,5)
```

```
y <- sample(10000,20)/rnorm(20,7)
```

Useful vector functions
Commands & flow control

- repeat: generates a vector of repeating data units

```
rep(data, number of repeats)
```

```
> rep(1:5, 5)
[1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Useful to create empty vectors, e.g.

```
> rep(0, 10)
[1] 0 0 0 0 0 0 0 0 0 0
```

- sequence: generates a vector of a data sequence

```
seq(from, to, by)
```

```
> seq(1,10,2)
[1] 1 3 5 7 9
> seq(2,10,2)
[1]  2  4  6  8 10
```

Looping - informal introduction

- Consider a problem where we need to execute a certain command many times
- e.g. we want to print numbers 1, 2, .. 10 one-by-one

one solution:

```
print(1)
print(2)
...
print(9)
print(10)
```

a better solution:

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

Both give exactly the same output!
The **for** loop is just more compact.

Output:

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

R language elements
Commands & flow control

- Looping
  - Iterate over a set of values (**for** loop)
  - or while a condition is met (**while** loop)

• Remember that all operations in R are vectorized. No need to use loops to e.g. make a sum of two vectors.

• Loops are multi-line commands. R will execute them only after the whole loop has been typed in. Use Rstudio editor to type it all in, don't do it in R console!

LOOPS
Commands & flow control

- For loops - iterate over a set of values, in this case 1..10

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

- While loops - iterate while a condition is met

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

when this condition is false the loop stops

Loops with breaks
Commands & flow control

Any loop can be prematurely broken with **break**

```
for (f in 1:10) {
      if (f==6) break
      print( f )
}
```

stops when f == 6

Output:

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

What would happen if we switched lines 2 and 3 (i.e. break after print)?

Loop exercise
Commands & flow control

1. Write a for loop that prints the letters of the alphabet [a to z]

2. Write a while loop that does the same, in reverse [z to a]

3. Afterwards, add a break statement to stop the loops when letter 'n' is reached

- Hints :
  - think about how would you solve this problem without using loops
  - base your solution on the for/while examples from previous slides
  - try to modify them so they print letters of the alphabet (instead of numbers), use the built-in vector **letters** to get individual letters

Solution to loop problem
Commands & flow control

For loop

```
for (f in 1:26){
   print(letters[f])
}
```

While loop

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

Add this (→) to make it break

```
if (letters[f]=="n") break
```

Example code:
06_loopExercise.R

Code formatting avoids bugs!

- Code formatting is crucial for readability of loops

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

BAD!!!

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

GOOD!

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

Conditional branching
Commands & flow control

- Some commands might need to be executed only if a condition is met.

- `if` allows different outcomes to be selected based up on a calculation result within brackets.

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

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

Exercise: functions, loops & branches

- Write a function that computes the first *i* elements of the Fibonacci series
  - The Fibonacci series is simply a vector in which the proceeding number is an addition of the two prior items
    - 1, 1, 2, 3, 5, 8, 13, 21 ...
    - think vector arithmetic x[***i***] = x[***i***-2] + x[***i***-1] ; where ***i*** is the index number
- Solve the problem as follows:
  - Work out the steps needed to compute the first 10 items of the series
    - Steps
      - Define a vector to hold answer
      - Use a for loop to iterate 10 times
      - Use an if branch to do something different for index numbers <2 and >2
  - Later we will embody the steps in a function that takes ***i*** as an argument

Fibonacci problem: procedural solution

07_fibonacci.R
(solution1, procedure)

i is a For loop counter that iterates 10 times

if ***i*** is less than 2, then Fibonacci can only be 1, Otherwise the Fibonacci equation is computed . The result is stored in the ***i*** -th element of **fib**

fibResult will store the Fibonacci series, as it is computed

```
fib <- rep(0, 10)

for (i in 1:10){
        if (i>2){
            fib[i] <- fib[i-2] + fib[i-1]
        } else {
            fib[1] <- 1
            fib[2] <- 1
        }
}
```

There are 2 sets of procedural braces in the if … else statement, and these are nested inside the For loop { } braces.

{ } → if is TRUE procedure; ***i*** counter is greater than 2
{ } → if is FALSE procedure

It is important to ensure there are equal numbers of opening and closing procedural braces!

Q. The if else isn't needed. Show why?
A. See code sheet.

Defining user functions
Commands & flow control

- User functions are objects, they are assigned like any other R object!

```
myFunction <- function(args=...){
    ...code...
    return(...)
}
```

- User functions may pass any number of named or unnamed arguments, with or without default values
- User functions may only return a single object
  - They automatically return the last assigned object. Hence, a return statement is not required unless the object you want to return isn't the last object referenced

Fibonacci problem revistited: User function solution

- Modify your Fibonacci code as follows

07_fibonacci.R
(solution1, function)

```
fibonacci <- function(n=10){
        fib <- rep(0, n)

        for (i in 1:n){
            if (i>2){
                fib[i] <- fib[i-2] + fib[i-1]
            } else {
                fib[1] <- 1
                fib[2] <- 1
            }
      }

        return(fib)
}
```

Now type ***fibonacci()***

Add an argument ***fibonacci(25)***

n is the named argument passed to your user defined **fibonacci** function. It specifies the upper index number to which Fibonnaci will be calculated. We define the function with a default value 10, which will be automatically passed If the user doesn't enter any arguments

Fibonacci problem: Streamlined function

- This version of the function computes the value at position ***n***, not up to position ***n***
- It is more efficient than previous versions because results are not incrementally stored

```
fibonacciAtPosition <- function(n=10){
      nextVal <- 1
      prevVal <- 0
      while(n > 1){
          currentVal <- nextVal
          nextVal <- nextVal + prevVal
          prevVal <- currentVal
          n <- n-1
      }
      return(nextVal)
}
```

To get a range of values, it's necessary to use the `sapply()` function:

```
sapply(1:10,fibonacciAtPosition)
```

07_fibonacci.R (solution2, function)

prevVal and **nextVal** are lower and upper elements of the Fibonacci sequence.
n counts down. Each decrement of **n** sees the values of **prev** and **next** swap, via **current**. Prior to the swap, **next** is assigned the value of **next + prev**, which is equivalent to the Fibonacci formula described earlier.

Time for

AFTERNOON COFFEE (RESUME 20 MIN)

R for data analysis ... Or ...

Doing stuff with the R you've learnt today

4

3 steps to
Basic data analysis

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

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

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

A simple walkthrough
Exemplifies 3 steps to R analysis

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

Step 1.
Read in the data

Patient	Nuclei	NB_Amp	NB_Nor	NB_Del
1	42	0	34	8
2	40	3	30	7
3	56	6	50	0
4	42	5	37	0
5	32	1	30	1
6	70	10	53	7
7	65	3	58	4
8	40	4	31	5
9	60	0	54	6
10	61	0	57	4
11	43	13	29	1

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

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

```
rawData <- read.delim("08_NBcountData.txt")
rawData[1:10,]     # View the first 10 rows to ensure import is OK
                   # Note data frame contains a patient index column
```

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

```
read.csv("08_NBcountData.csv")
```

`?read.table` for a full list of arguments

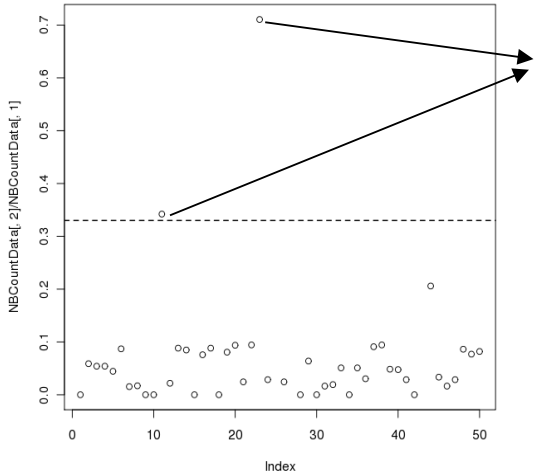
08_NBcountData.R
(script commands)

08_NBcountData.txt
(data file)

Step 2.
Analysis (reshaping data & maths)

- Our analysis involves identifying patients with > 33% NB amplification
  - `prop <- rawData$NB_Amp / rawData$Nuclei # create an index of results`
  - `amp <- which(prop > 0.33) # Get sample names of amplified patients`

- We can plot a simple chart of the % NB amplification
  - `plot(prop, ylim=c(0,1.2))`
  - `abline(h=0.33,lwd=1.5,lty=2)`



These 2 samples are amplified (11 & 23)

Step 3.
Outputting the results

- We write out a data frame of results (patients > 33% NB amplification) as a 'comma separated values' text file
  - `write.csv(rawData[amp,],file="selectedSamples.csv") # Export table, file name = selectedSamples.csv`
    - Files are directly readable by Excel and Calc

- Its often helpful to double check where the data has been saved
  - Use get working directory function
    - `getwd() # print working directory`

Data analysis exercise:
Which samples are near normal?

- Patients are near normal if:

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

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

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

08_NBcountData.R
(script commands)

Solution to NB normality test

Basic data analysis

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

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

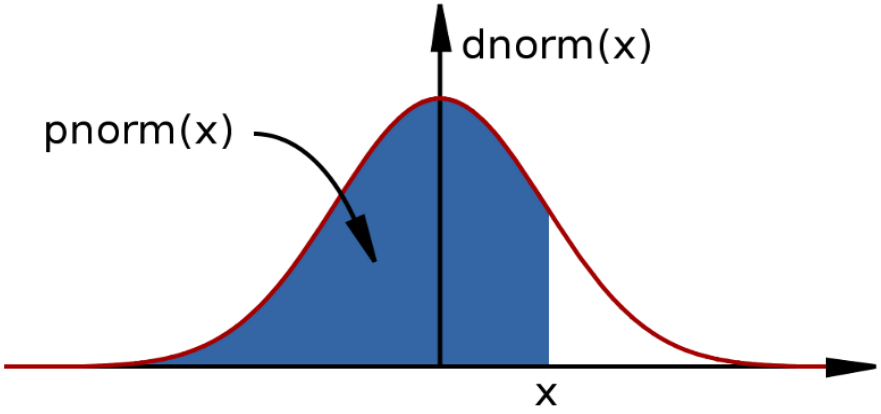

> write.csv(rawData[norm,],"My_NB_output.csv")
```

R mathematics support
Basic data analysis

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

Pseudo-random numbers and distributions

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



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

Classical tests
Basic data analysis

- Tests for normality
  - Quantile-quantile plot
    - Simply plots your distribution against the same number of randomly distributed events. Looking for a straight line.

```
qqnorm(...)        # generates qq plot
qqline(...)    # intersects qq plot
```

  - shapiro.test()
    - Simple normality test. Significant pValue (<0.05) when test fails. I.e. distribution is not normally spread.

Normality test example
Basic data analysis

```
y<-rnorm(1000) # normally distributed data
yy<-exp(rnorm(1000)) # exponential data

yScale<-c(min(y),max(yy)) # y axis scale
qqnorm(y, ylim=yScale) # plot normal series

par(new=T)
qqnorm(yy, ylim=yScale, col="blue")
qqline(y, lty=2, col="black")
qqline(yy, lty=2, col="blue")
```

```
> shapiro.test(y)

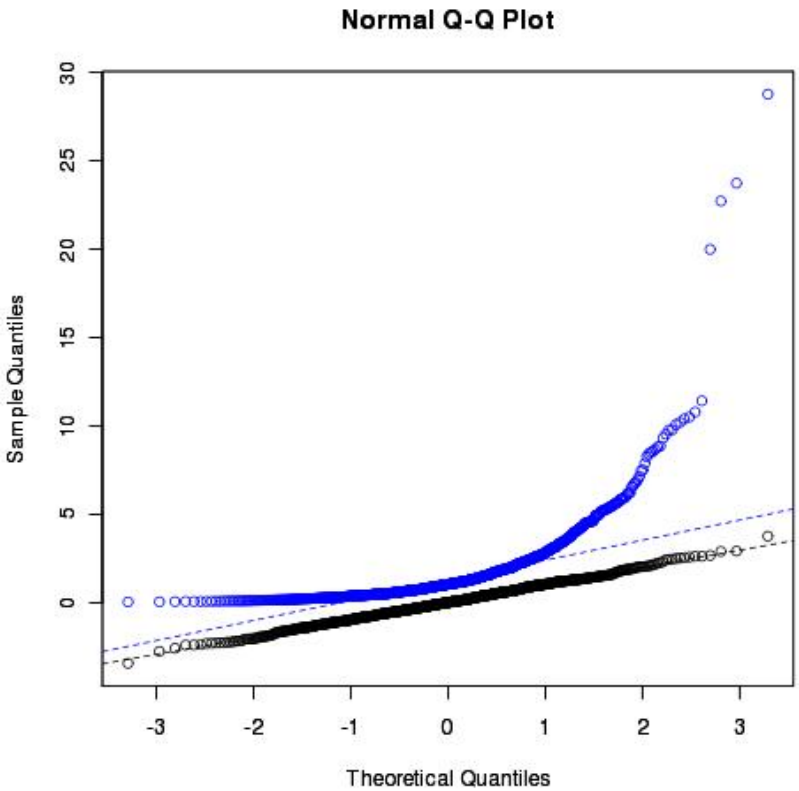
        Shapiro-Wilk normality
test

data:  y
W = 0.9988, p-value = 0.7452
```

```
> shapiro.test(yy)

        Shapiro-Wilk normality
test

data:  yy
W = 0.5884, p-value < 2.2e-16
```



Example code:
09_normalTests.R

Two sample tests
Basic data analysis

- Comparing 2 variances
  - Fisher's F test

`var.test()`

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

`t.test()`

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

`wilcox.test()`

- Comparing 2 proportions
  - Binomial test

`prop.test()`

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

`cor.test()`

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

`chisq.test()`

  - Fisher's exact test

`fisher.test()`

Comparison of 2 data sets example
Basic data analysis

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

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

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

```
heightData<-read.csv("10_heightData.csv",header=T)
```

- Test the variance, calculate F statistic

```
var(heightData$Male) ; var(heightData$Female)
var.height <- var(heightData$Female) / var(heightData$Male)
var.height   # F ~ 1, no significant difference, but how do we know?
```

- Determining what a significant difference is?
  - Critical value of F

```
qf(0.975,99,99) # p=(1-α/2) ; α=0.05
```

- Critical F for 99 d.f = 1.49, at p~0.05
- Our F (~1) < Critical F (1.49). Therefore no difference in variance
  - Probability associated with our F test (two tailed test)

```
2*(1-pf(1,99,99))
```

  - 1 i.e, every single time!!!

var.test(...) does this all in 1 go!

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

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

- Student's t test is appropriate for comparing the difference in mean height in our data.
  - Remember a t test = $\frac{\text{difference in two sample means}}{\text{standard error of the difference of the means}}$
- In this example we have 198 degrees of freedom (losing 1df per estimate of mean height) and we will accept a significant results if p <= 0.05
  - Hence ***critical t*** for this test is
    - `qt(0.975,198)` # 1.97
      - Test statistic needs to be bigger than 1.97 if we are to reject the null hypothesis that the samples means are equal
    - We calculate the t statistic long hand …

```
mean(heightData$Male-heightData$Female)/sqrt((var(heightData$Male)/100)+
       (var(heightData$Female)/100))
[1] 8.45  # Result is significant, but by how much?
2 * pt(8.45,198, lower.tail=FALSE)
[1] 6.2e-15 # two-tailed
```

t.test(...) does this all in 1 go!

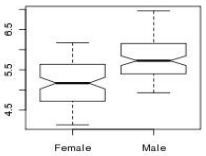
```
t.test(Female,Male)
```

3. Comparison of 2 data sets
Review findings

```
> t.test(heightData$Male,heightData$Female)

    Welch Two Sample t-test

data:  Male and Female
t = 8.4508, df = 197.997,
p-value = 6.217e-15
alternative hypothesis: true difference in
means is not equal to 0
95 percent confidence interval:
 0.4841288 0.7788497
sample estimates:
mean of x mean of y
 5.800214  5.168725
```



Notched boxplots are a good way to review t.tests

```
heightData<-heightData[,-1]
boxplot(heightData,notch=T)
```

Review findings:

1. There is no difference in the location of variance between Males and Females
2. Males are taller than Females, on average (5.8 ft vs. 5.2 ft, pValue=6.2e-15)
3. Notched boxplot demonstrates difference in mean heights

Exercise
Work through the previous tests

You should:

1) Undertake the steps of the variance and t.test 'height exercise'
2) Make sure you are able to derive the F and t statistics, with probabilities
3) Generate the simple box plot
4) Access the help and arguments information for each function used

help("t.test")

Shortcut ... ?t.test

args(t.test)

Linear regression
Basic data analysis

- Linear modeling is supported by the function **`lm()`**
  - **`example(lm)`** # the output assumes you know a fair bit about the subject

- lm is really useful for plotting lines of best fit to XY data in order to determine, intercept, gradient & Pearson's correlation coefficient
  - This is very easy in R

- Three steps to plotting with a best fit line
  - Plot XY scatter-plot data
  - Fit a linear model
  - Add bestfit line data to plot with abline() function

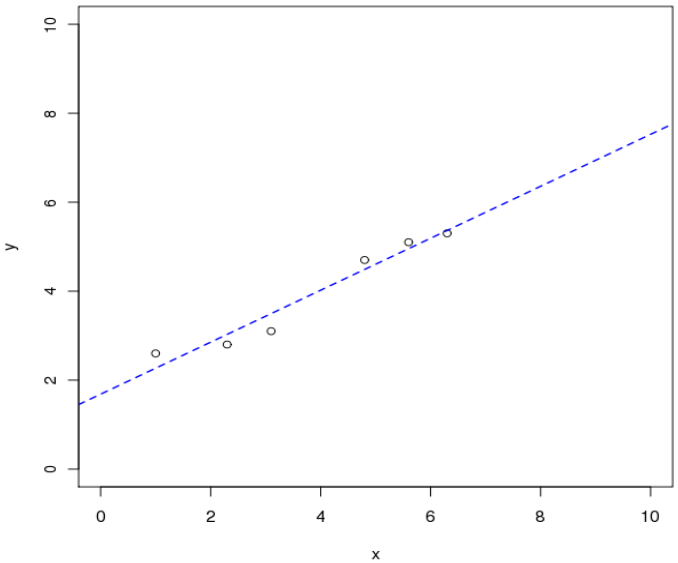
Typical linear regression analysis
Basic data analysis

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

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

Note formula notation
(y is given by x)

```
> myModel<-lm(y~x)
> abline(mvModel,lty=2,lwd=1.5,col="blue")
```



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

Get QC of fit from
`plot(myModel)`

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

The linear model object
Basic data analysis

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

```
> summary.lm(myModel)

Call:
lm(formula = y ~ x)

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

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

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

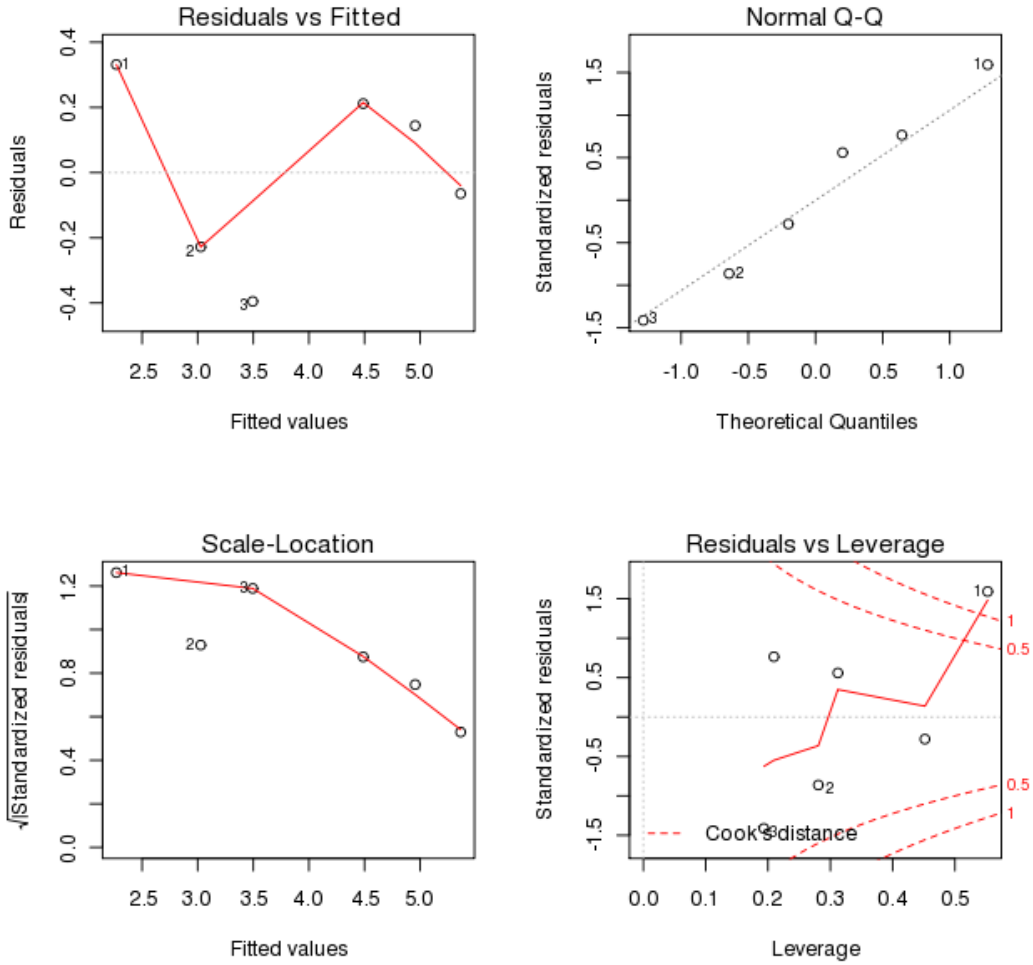
Y intercept

Gradient

Good fit: would happen 1 in 1000 by chance

R^2 , with pValue

QC plots of regression analysis
Basic data analysis




```
plot(myModel)
```

End of Day 1