R Course
Day 1

See also:
https://www.udemy.com/introduction-to-r/?tc=blog.rtutorial&couponCode=half-off-for-blog&utm_source=blog&utm_medium=udemyads&utm_content=post28050&utm_campaign=content-marketing-blog&xref=blog

Variables can be overwritten! Hmm...

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

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

Without naming :
```
> seq(2, 10, 2)
```

Seq is useful!
Hint: to find function **help** using ?function egg. ? seq
Can see an example using: example(func) eg. example(seq)

Also - help(function)

Vectors - numbers and strings

c() combines its arguments into a vector:
```
> x <- c(3, 4, 5, 6)
> x
[1] 3 4 5 6
```

```
> x <- 3:12   ## : specifies a range
```

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

```
> x <- seq(2, 10, 2)
> x
[1] 2 4 6 8 10
```

```
> x <- seq(2, 10, length.out = 7)
```

rep(num, repetitions) # can repeat values eg. rep(3,5) -> 3 3 3 3 3

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
```

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 ()

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

```
> x <- 1:10
> y <- x*2
> y
[1] 2 4 6 8 10 12 14 16 18 20

> z <- x^2
> z
[1] 1 4 9 16 25 36 49 64 81 100
```

Adding two vectors

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

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

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

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

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

Character (string) Vectors

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

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

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

```
> gene.expression <- c(0,3.2,1.2,-2)
> gene.expression
[1] 0.0 3.2 1.2 -2.0

> names(gene.expression)<-gene.names
> gene.expression
Pax6 Beta-actin FoxP2 Hox9
0.0 3.2 1.2 -2.0
```

We can also use the names function to get a vector of the names of an object:
> names(gene.expression)
[1] "Pax6" "Beta-actin" "FoxP2" "Hox9"

Exercise
Species: species.name <- c("Homo sapiens", "Mus musculus", "Drosophila melanogaster", "Caenorhabditis elegans", "Saccharomyces cerevisiae")

Genome size (Mb): genome.size <- c(3102, 2731, 169, 100, 12)
Protein coding genes: coding.genes <- c(20774, 23139, 13937, 20532, 6692)

Let's assume a coding gene has an average length of 1.5 kilobases. On average, how many base pairs of each genome is made of coding genes? Create a new vector to record this called coding.bases

What percentage of each genome is made up of protein coding genes? Use your coding.bases and genome.size vectors to calculate this. (See earlier slides for how to do division in R.)

How many times more bases are used for coding in the human genome compared to the yeast genome? How many times more bases are in the human genome in total compared to the yeast genome? Look up indices of your vectors to find out.

To calculate the number of coding bases, we need to use the same scale as we used for genome size: 1.5 kilobases is 0.0015 Megabases.
> coding.bases<-coding.genes*0.0015
> coding.bases
H. sapiens M. musculus D. melanogaster C. elegans S. cerevisiae
31.1610 34.7085 20.9055 30.7980 10.0380

To calculate the percentage of coding bases in each genome:
> coding.pc<-coding.bases/genome.size*100
> coding.pc
H. sapiens M. musculus D. melanogaster C. elegans S. cerevisiae
1.004545 1.270908 12.370118 30.798000 83.650000

To compare human to yeast:
> coding.bases[1]/coding.bases[5]
H. sapiens
3.104304

> genome.size[1]/genome.size[5]
H. sapiens
258.5

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

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

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

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

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

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

Data frames

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.

Character, numeric and logical data types

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

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

We can define the names using character vectors:

```
> firstName<- c("Adam", "Eve", "John", "Mary", "Peter", "Paul", "Joanna",
```

"Matthew", "David", "Sally")
> secondName<-c("Jones", "Parker", "Evans", "Davis", "Baker", "Daniels",
"Edwards", "Smith", "Roberts", "Wilson")

We also have a new type of vector, the logical vector, which only contains the values TRUE and FALSE:
> consent<-c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE)

Can shortcut any logical using T, F instead of TRUE, FALSE eg: consent<-c(T,T,FT,F,F … etc)

Factors

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

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

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

Factors in data frames

When creating a data frame, **R assumes all character vectors should be categorical variables** and converts them to factors. This is not always what we want:
> patients$firstName
[1] Adam Eve John Mary Peter Paul Joanna Matthew David Sally
Levels: Adam David Eve Joanna John Mary Matthew Paul Peter Sally

We can avoid this by asking R not to treat strings as factors, and then explicitly stating when we want a factor by using factor:
> patients<-data.frame(First_Name=firstName, Second_Name=secondName,
Full_Name=paste(firstName,secondName), Sex=**factor**(sex), Age=age,
Weight=weight, Consent=consent, **stringsAsFactors=FALSE**)

Note: Sex=**factor**(sex) ## we want to keep Sex as a **Factor!**

```
> patients$Sex
[1] Male Female Male Female Male Male Female Male Male Female
Levels: Female Male
> patients$First_Name
[1] "Adam" "Eve" "John" "Mary" "Peter" "Paul" "Joanna" "Matthew" "David" "Sally"
```

see **?data.frame** for more info

Creating a Data Frame by hand - see script "1.2_patients.R"

```
# character vector
firstName<-c("Adam","Eve","John","Mary","Peter","Paul","Joanna","Matthew","David","Sally")

# character vector
secondName<-c("Jones","Parker","Evans","Davis","Baker","Daniels","Edwards","Smith","Robert
s","Wilson")

# Factor
sex<-c("Male","Female","Male","Female","Male","Male","Female","Male","Male","Female")

# Numeric vector
age<-c(50,21,35,45,28,31,42,33,57,62)

# Double vector
weight<-c(70.8,67.9,75.3,61.9,72.4,69.9,63.5,71.5,73.2,64.8)

# Logical vector
consent<-c(TRUE,TRUE,FALSE,TRUE,FALSE,FALSE,FALSE,TRUE,FALSE,TRUE)

# Dataframe from vector objects
# Note column names are not defined in function call, hence column names will be taken

# from original vector names
patients<-data.frame(firstName, secondName, paste(firstName,secondName),factor(sex),
age, weight, consent, stringsAsFactors=FALSE)

# Note: Sex=factor(sex) ## we want to keep Sex as a Factor!

# Name columns in dataframe
# Not really required, could have be done above e.g. ... data.frame(First_Name=firstName, ...
names(patients)<-c("First_Name","Second_Name","Full_Name","Sex","Age","Weight","Consent"
)
```

```
# Display data frame
print(patients)

# or
patients
```

Matrices

[see ?matrix]

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

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

Useful: rowMeans(x) colMeans(x)

Lists

list(name1=obj1,name2=obj2,...

We have seen that vectors can only hold data of one type. How can we store data of multiple types? Or vectors of different lengths in one object?

We can use lists. A list can contain objects of any type:

```
one.to.ten <- 1:10
uniform.mat <- matrix(runif(100),ncol=10,nrow=10)
year.to.october <- data.frame(one.to.ten, month.name[1:10])
myList<-list( ls.obj.1=one.to.ten, ls.obj.2=uniform.mat,ls.obj.3=year.to.october)
names(myList)
```

We can use the dollar syntax to access list items (in fact, a data frame is a special type of list):
myList$ls.obj.1

We can also use **myList [[1]] # double square brackets #** to get the first item in the list.

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

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

```
> e[1,2]
[1] 6
> e[1,]
[1] 1 6

> patients[1,2]
[1] "Jones"

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

Special cases:
a[i,] i-th row : **object [row ,] # all values for row**

a[,j] j-th column : **object [, column] # all values for column**

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

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

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

[1] TRUE TRUE FALSE FALSE FALSE
> s[a<3]
[1] "a" "b"

> s[a>1 **&** a<3]
[1] "b”
> s[a**==**2]
[1] "b"
Operators
arithmetic: +, -, *, /, ^

comparison: <, >, <=, >=, ==, !=
logical: !, &, |, xor

These always return logical values ! (TRUE, FALSE)

Exercise:
Remake your data frame with three new variables: country, continent, and height. Make up the data. Make country a character vector but continent a factor.

Hint: you can check number of elements in vector using **length**(vector_name) eg. **length**(height)

Note: Works differently when applied to a **data frame** - will return the number of variables. A data frame is a list of variables, so its length is the number of variables. The length of one of the variable vectors (like Age) is the number of observations.

When the data frame is completed can access stats with: **summary**()

Use **logical indexing** to select the following patients:
– *Patients under 40* : **my**.**patients[my.patients$Age<40,] # <- note: trailing comma!**
– *Patients who give consent to share their data* : **my.patients[my.patients$Consent==TRUE,]**
– *Men who weigh as much or more than the average European male (70.8 kg)*
mean(my.patients[my.patients$Sex=="Male", "Weight"]) # to calc mean
my.patients[my.patients$Sex=="Male" & my.patients$Weight>=70.8,]

Making the data frame
# Character vectors
country<-c("ENG","SCO","IRE","SCO","ENG","US","ENG","IRE","IRE","SCO")
continent<-c("EU","EU","EU","EU","EU","US","EU","EU","EU","EU") # we'll turn this into a factor in the DF

numeric

```
height<-c(71.8,68.9,72.3,69.9,71.4,63.9,66.5,75.5,79.2,66.8)

# Dataframe from vector objects
# Note column names are not defined in function call, hence column names will be taken
# from original vector names
my.patients<-data.frame(firstName, secondName, paste(firstName,secondName),factor(sex),
country, factor(continent),age, weight, height, consent, stringsAsFactors=FALSE)

# Name columns in dataframe
# Not really required, could have be done above e.g. ... data.frame(First_Name=firstNam, ...
names(my.patients)<-c("First_Name","Second_Name","Full_Name","Sex","Country","Continent",
"Age","Weight","Height","Consent")
```

Adding and removing rows and columns

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

```
> newpatient<-c("Kate","Lawson","Kate Lawson","Female","35","62.5","TRUE")
> rbind(patients,newpatient)
```

To remove rows and columns:

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

Sort and order

Sorting a vector with sort:

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

Sorting a data frame by one variable with order:

```
> order(patients$Second_Name)
[1] 5 6 4 7 3 1 2 9 8 10
> patients[order(patients$Second_Name),]
```

also - reverse order with **rev**() eg.

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

The R workspace

The objects we have been making are created in the R workspace.

When we load a package, we are loading that package's functions and data sets

into our workspace.

You can see what is in your workspace with ls:
> ls()

You can attach data frames to your workspace and then refer to the variables directly:
> attach(patients)
> Full_Name

You can remove objects from the workspace with rm
> x<-1:5
[1] 1 2 3 4 5
> rm(x)
> x
Error: object 'x' not found

You can remove everything by giving rm a list of all the objects returned by ls:
> rm(list=ls())

Analysis in 3 Steps:
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

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

Putting it into practice:

Patient	Nuclei	NB_Amp	NB_Nor	NB_Del
1	65	0	63	2
2	51	3	43	5
3	37	2	35	0
4	37	2	35	0
5	45	2	42	1
....	....	....		

NB: If there are holes in the data, R may insert NA into the table. These can be a problem with evaluation and selecting later subsets. We can overcome this using **which**()

eg. ((rawData$NB_Amp / rawData$Nuclei) < 0.33 & rawData$NB_Del==0)
[1] FALSE FALSE TRUE TRUE FALSE FALSE TRUE FALSE FALSE FALSE FALSE FALSE NA ...

so… **which**((rawData$NB_Amp / rawData$Nuclei) < 0.33 & rawData$NB_Del==0)

This returns the rows which evaluate as TRUE.

Solution

```
#####################################################################
## 1.3_NBcountData.R                                               ##
## Exercise in reading in data, returning basic information##
## and exporting objects to spreesheets as CSV files       ##
##                                                 ##
#####################################################################
#
### START ###
## Read in tab delimitted text file of results
## Remove index column
rawData <- read.delim("1.3_NBcountData.txt")
rawData[1:10,]
rawData <- rawData[,-1] # Drop the patient index column

## Getting basics summary data
nrow(rawData)                  # number of rows
ncol(rawData)                  # number of columns
dim(rawData)                   # data frame dimensions (rows X columns)
colMeans(rawData)    # average stats

## Reorder table by decreasing nuclei count
rawDataOrder <- rawData[order(rawData[,1],decreasing=T),]
rawDataOrder

#  Identify patients with >33% NB amplification
prop <- rawData$NB_Amp / rawData$Nuclei
amp <- which(prop > 0.33)

plot(prop, ylim=c(0,1.2)) # plot a simple chart of NB amplifications

abline(h=0.33, lwd=1.5, lty=2) # Add a dotted line at 33%
```

```
# Write out results table as comma separated values file
write.csv(rawData[amp,],file="selectedSamples.csv")

#  In case you've forgotten where they've been saved
getwd() # prints working directory path, so you know where to look for the results file

### Solution to problem
### Which samples are near normal i.e. NBamp% < 0.33 & NBdel==0)
norm <- which((rawData$NB_Amp / rawData$Nuclei) < 0.33 & rawData$NB_Del==0)
write.csv(rawData[norm,],file="My_NB_output.csv")
### END ###
```

Loops

Two main styles but there are others - see also: https://www.udemy.com/blog/r-tutorial/
… and: http://www.cyclismo.org/tutorial/R/scripting.html

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

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

Example: reading in multiple files to a data frame:
```
dir(pattern="Counts.csv") # NB: pattern is regular expression
[1] "1.4_colony_Run1Counts.csv" "1.4_colony_Run2Counts.csv" "1.4_colony_Run3Counts.csv"
```

So we can load all the files using a for loop as follows:
```
colony<-data.frame()
countfiles<-dir(pattern="Counts.csv")
for (file in countfiles) {
   t<-read.csv(file)
   colony<-rbind(colony,t)
}
```

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

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

if (**condition**) {
...
do this ...
} **else** {
...
do something else ...
}

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

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

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

The stop function outputs the error message and quits.

Also, **while** works too - in a similar way to Perl. Check it out...

Exercise

1. Output the patients data frame, with the patients sorted in order of age, oldest first. (You may need the rev function.)
2. Load in the colony data frame using a for loop. Three of the data files are in the Day_1_scripts folder. Load all three files into colony using the for loop in the slides.
3. How many observations do you have in the colony data frame?
Find out by counting the number of rows in colony using the nrow function.
4. Suppose a power analysis of your data shows that you only need 48 observations to robustly test your hypothesis. This means we can stop loading files when we have loaded at least 48 observations. Modify your for loop so it will only load files if the colony data frame has less than 48 observations in it.

Solutions:

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

See also ?**order**() - may not need rev as **order**() can be given attribute **'decreasing = TRUE'**

```
colony<-data.frame() # initialise a new data frame
countfiles<-dir(pattern="Counts.csv") # pattern match for files with 'Counts.csv'

# set loop to read in files
n <- 48
if (length(countfiles) == 0) {
  stop("No Counts.csv files found!") # throw error if no files found
} else {
  for (file in countfiles ) {
          t<-read.csv(file) # read the file
          print(file)
          colony<-rbind(colony,t) # add to data frame
          if (nrow(colony) >= n) {
          message(paste("We have enough rows:", nrow(colony)))
          break
          }
  }
}
# get some stats for colony
summary(colony)

# find number of rows in 'colony' data frame
nrow(colony)
```

Alternative code:

```
for (file in countfiles) {
   if (nrow(colony) < 48) {
      t<-read.csv(file)
      colony<-rbind(colony,t)
   }
}
```

Statistics

Distributions : help(Distributions) or ?distributions

Density, cumulative distribution function, quantile function and random variate generation for many standard probability distributions are available in the stats package.

For every distribution there are four commands. The commands for each distribution are prepended with a letter to indicate the functionality:

“d” returns the height of the probability density function
“p” returns the cumulative density function
“q” returns the inverse cumulative density function (quantiles)
“r” returns randomly generated numbers

The Normal Distribution
In probability theory, the normal (or Gaussian) distribution is a very commonly occurring continuous probability distribution—a function that tells the probability that any real observation will fall between any two real limits or real numbers, as the curve approaches zero on either side.

The Gaussian distribution is sometimes informally called the bell curve. However, many other distributions are bell-shaped (such as Cauchy's, Student's, and logistic).

There are four functions that can be used to generate the values associated with the normal distribution.
dnorm, pnorm, qnorm, rnorm

You can get a full list of them and their options using the help command:

> help(Normal)

Binomial Distribution
In probability theory and statistics, the binomial distribution is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p. Therewith the probability of an event is defined by its binomial distribution.

The **binomial distribution** is a discrete probability distribution. It describes the outcome of *n*independent trials in an experiment. Each trial is assumed to have only two outcomes, either success or failure. If the probability of a successful trial is *p*, then the probability of having *x*successful outcomes in an experiment of *n* independent trials is as follows.

$$f(x) = \binom{n}{x} p^x (1-p)^{(n-x)} \quad where\ x = 0, 1, 2, ..., n$$

Problem

Suppose there are twelve multiple choice questions in an English class quiz. Each question has five possible answers, and only one of them is correct. Find the probability of having four or less correct answers if a student attempts to answer every question at random.

Solution

Since only one out of five possible answers is correct, the probability of answering a question correctly by random is 1/5=0.2. We can find the probability of having exactly 4 correct answers by random attempts as follows.

```
> dbinom(4, size=12, prob=0.2)

[1] 0.1329
```

To find the probability of having four or less correct answers by random attempts, we apply the function dbinom with $x = 0,\ldots,4$.

```
> dbinom(0, size=12, prob=0.2) +

+ dbinom(1, size=12, prob=0.2) +

+ dbinom(2, size=12, prob=0.2) +

+ dbinom(3, size=12, prob=0.2) +

+ dbinom(4, size=12, prob=0.2)

[1] 0.9274
```

Alternatively, we can use the cumulative probability function for binomial distribution pbinom.

```
> pbinom(4, size=12, prob=0.2)

[1] 0.92744
```

Answer

The probability of four or less questions answered correctly by random in a twelve question multiple choice quiz is 92.7%.

Exercise

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

1) We can model a coin toss using the binomial distribution. Use the **rbinom** function to generate a sample of 100 coin tosses. Look up the binomial distribution help page to find out what arguments this function needs.

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

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

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

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

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

> summary(factor(coin.toss))
0 1
50 50
```

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

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

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

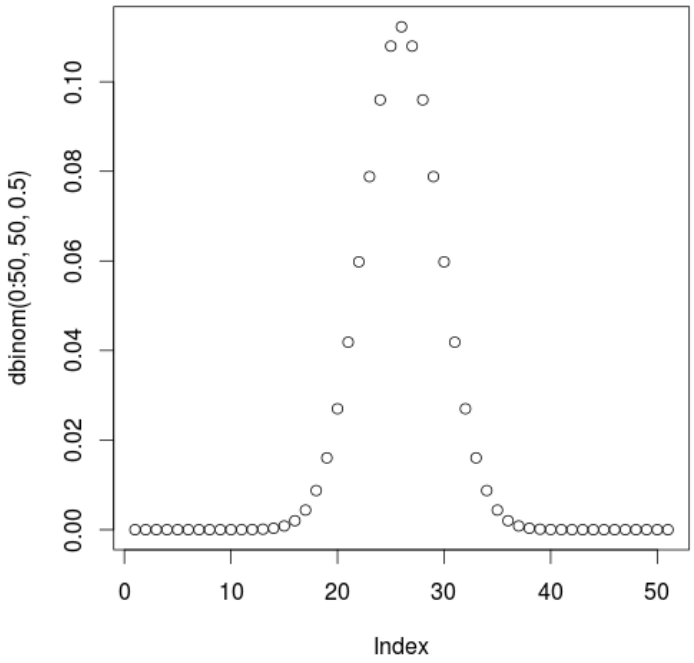
```
> pbinom(25, 50, 0.5)
```

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

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

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

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



Produces normal distribution where index is:
0 - getting no heads
50 - getting all heads

Two sample tests: Basic data analysis

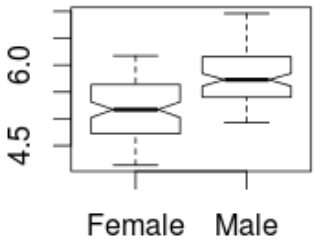
- Comparing 2 variances
    - Fisher's F test: var.test()
- Comparing 2 sample means with normal errors
    - Student's t test: t.test()
- Comparing 2 means with non-normal errors
    - Wilcoxon's rank test: wilcox.test()
- Comparing 2 proportions
    - Binomial test: prop.test()
- Correlating 2 variables
    - 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()

Exercise
Men, on average, are taller than women.

The steps

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

```
setwd("~/Course_Materials/Day_1_scripts")
> heightData<-read.csv("1.5_heightData.csv")
> attach(heightData) # make variables available with having to always use dataframe index
> var.test(Female,Male) #

        F test to compare two variances

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

> t.test(Female,Male, alternative="less") # Performs one and two sample t-tests on vectors of data.

        Welch Two Sample t-test

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

> boxplot(heightData, notch=T) # visualise
```



Regression (Linear) Analysis

In statistics, linear regression is an approach for modeling the relationship between a scalar dependent variable y and one or more explanatory variables denoted X. The case of one explanatory variable is called simple linear regression.

Define data -variables

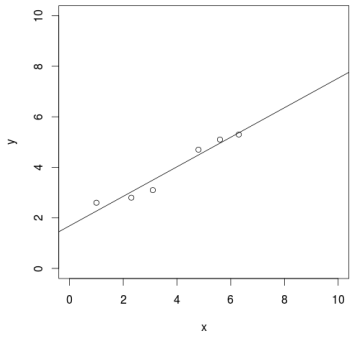
```
x<-c(1, 2.3, 3.1, 4.8, 5.6, 6.3)
y<-c(2.6, 2.8, 3.1, 4.7, 5.1, 5.3)
```

Plot relationship

```
plot(y~x, xlim=c(0,10),ylim=c(0,10))
```

Define model and add gradient (slope) through points

```
myModel<-lm(y~x) # Note formula notation ( y is given by x )
abline(myModel)
```



Summary stats

Get the coefficients of the fit from:

```
summary.lm(myModel)
        coef(myModel)
```

resid(myModel)
fitted(myModel)

Get QC of fit from plot(myModel)
Find out about the fit data from names(myModel)

Modelling formulae
R has a very powerful formula syntax for describing statistical models.
Suppose we had two explanatory variables x and z and one response variable y.

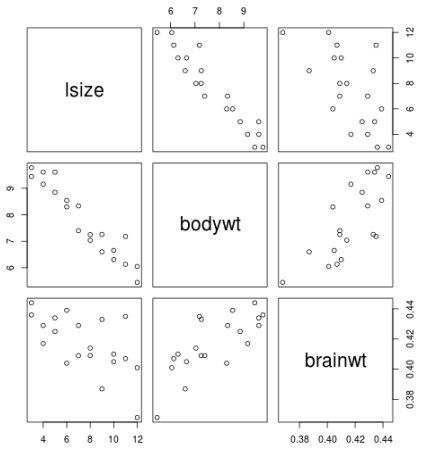
We can describe a relationship between, say, y and x using a tilde (~), placing the response variable on the left of the tilde and the explanatory variables on the right:
> y~x

It is very easy to extend this syntax to do multiple regressions, ANOVAs, to include interactions, and to do many other common modelling tasks. For example:
> y~x # If x is continuous, this is linear regression
> y~x # If x is categorical, this is ANOVA
> y~x+z # If x and z are continuous, this is multiple regression
> y~x+z # If x and z are categorical this is a two-way ANOVA
> y~x+z+x:z # : is the symbol for the interaction term
> y~x*z # * is a shorthand for x+z+x:z

Exercise
Mice have varying numbers of babies in each litter. Does the size of the litter affect the average brain weight of the offspring? We can use linear modelling to find out. (This example is taken from John Maindonald and John Braun's book Data Analysis and Graphics Using R (CUP, 2003), p140-143.)
1) Install and load the DAAG package. The litters data frame is part of this package. Take a look at it. How many variables and observations does it have? Does summary tell you anything useful? What about plot ?
2) Are any of the variables correlated? Look up the cor.test function and use it to test for relationships.
3) Use lm to calculate the regression of brain weight on litter size, brain weight on body weight, and brain weight on litter size and body weight together.
4) Look at the coefficients in your models. How is brain weight related to litter size on its own? What about in the multiple regression? How would you interpret this result?

Solutions
In Rstudio - under packages -> install -> DAAG. Select the package to load it.
The litters data frame is part of this package
> summary(litters)

> plot(litter) # shows plots of variables against each other



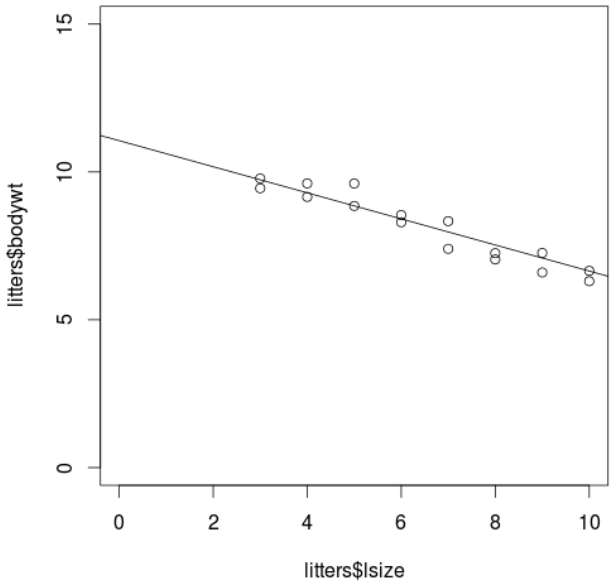
? cor.test: Test for association between paired samples, using one of Pearson's product moment correlation coefficient, Kendall's tau or Spearman's rho.

Usage

```
cor.test(x, ...)

## Default S3 method:
cor.test(x, y,
        alternative = c("two.sided", "less", "greater"),
        method = c("pearson", "kendall", "spearman"),
        exact = NULL, conf.level = 0.95, continuity = FALSE, ...)

## S3 method for class 'formula'
cor.test(formula, data, subset, na.action, ...)
```

> cor.test(litters$bodywt, litters$lsize, method = "kendall", alternative = "greater")
> plot(litters$bodywt~litters$lsize, xlim=c(0,10),ylim=c(0,15))

> litterModel<-lm(litters$bodywt~litters$lsize)
> abline(litterModel)




```
> cor.test(litters$bodywt, litters$lsize, method = "spearm", alternative = "l", exact = FALSE)

        Spearman's rank correlation rho

data:  litters$bodywt and litters$lsize
S = 2594.764, p-value = 6.528e-11
alternative hypothesis: true rho is less than 0
sample estimates:
      rho
-0.9509502
```

Now… Litter size vs. brain weight

```
> plot(litters$brainwt~litters$lsize, xlim=c(0,0.5),ylim=c(0,10))
> bwModel<-lm(litters$brainwt~litters$lsize)
> abline(bwModel)
```

Correlation

```
> cor.test(litters$brainwt, litters$lsize, method = "spearm", alternative = "l", exact = FALSE)

        Spearman's rank correlation rho

data:  litters$brainwt and litters$lsize
S = 2149.702, p-value = 0.001903
alternative hypothesis: true rho is less than 0
```

```
sample estimates:
      rho
-0.616317

> lm(litters$brainwt~litters$lsize)

Call:
lm(formula = litters$brainwt ~ litters$lsize)

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

Alternative solutions

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

Linear modelling answers

To calculate the linear models:

```
> lm(brainwt~lsize)

Call: lm(formula = brainwt ~ lsize)
Coefficients:
(Intercept) lsize
0.447000 -0.004033

> lm(brainwt~bodywt)
Call:
lm(formula = brainwt ~ bodywt)

Coefficients:
(Intercept) bodywt
0.33555 0.01048

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

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

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

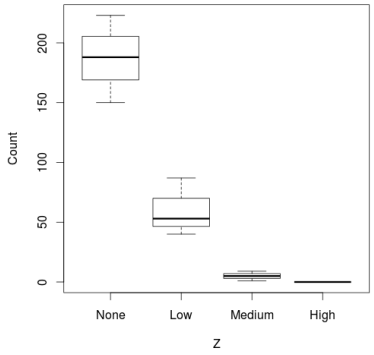
Day 2

Into practice…
Colony forming experiment
We have been asked by some collaborators to analyse some trial data to see if an experiment will work. We are interested in the behaviour of a gene, X, which is involved in a cell proliferation pathway. This pathway causes cells to proliferate in the presence of a compound, Z. Gene X turns the pathway off, reducing cell proliferation.
Our collaborators want to test what happens when we knock down X in the presence of Z.
To do this, they want to grow cell colonies in the presence of Z, with or without X, and count the number of colonies that result.

Our collaborators have sent us a first batch of test data, growing colonies in different concentrations of compound Z, and replicating each Z concentration three times.
Does increasing concentration of Z have an effect on colony growth?

We want to do the following:
– Load the data into R
– Plot the data to inspect it
– Calculate an Analysis of Variance to see if growth is influenced by Z concentration
– Calculate the mean growth for each level of Z concentration, to see the direction of change
(We will ignore full post hoc testing)

```
# Load data, order Z and plot
colony<-read.csv("2.1_colony_trial.csv")
colony$Z<-factor(colony$Z,levels=c("None","Low","Medium","High"))
attach(colony)
plot(Count~Z)
```




```
# Analysis of Variance
colony.aov<-aov(Count~Z)
print(summary(colony.aov))
```

```
          Df Sum Sq Mean Sq F value   Pr(>F)
Z          3  68154   22718   46.89 2.02e-05 ***
Residuals  8   3876     484
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

This tells us what we can already see from the plot, that there is a highly significant relationship between Z concentration and colony growth.
We would like to investigate this relationship. For example, we might want to calculate the mean colony count for each concentration of Z.

We can calculate a mean for a particular group like this:

```
> mean(colony[Z=="None",]$Count)
[1] 187
> mean(colony[Z=="Low",]$Count)
[1] 60
> mean(colony[Z=="Medium",]$Count)
[1] 5
> mean(colony[Z=="High",]$Count)
[1] 0
```

We could generalise this with a for loop:

```
for (z in levels(Z)) {
  print(mean(colony[Z==z,]$Count))
}
```

```
[1] 187
[1] 60
[1] 5
[1] 0
```

NB: But there is a better way - **tapply** [**useful - look into this!**]
The **apply** family of functions allow us to **group data by variable** and calculate **something for each group**.
See ?apply and ?tapply, ?lapply

We can use tapply to calculate group means on colony like this:

```
> colony.means<-tapply( Count, Z, mean )
> colony.means
None Low Medium High
187    60    5          0
```

```
print(colony.means)
barplot(colony.means)
detach(colony)
```

Phase II - modifying the script for more parameters
As the trial worked, our collaborators have gone ahead with an experiment to knock down gene X in the same concentrations of Z.
The new data is in the file 2.1_colony_run.csv

They want us to see if knocking down X affects colony growth.
Because we saved our analysis in a script, we can rerun the same script to analyse the data, just by changing the name of the file we are loading. Run your script on this new data file and confirm that you can calculate an ANOVA and group means for this new data set.

Our current script only analyses Z, not X. We need to modify it to include X and see how both X and Z influence colony growth.

1. We need to include X and the interaction between Z and X in our formulae for plotting and for ANOVA. Look up the 'Modelling formulae' slide from Day 1 to see how to do this.

2. What does plot do with a formula including both X and Z? Try using boxplot instead. What difference does it make if you change the order of X and Z?

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

4. Plot the group means you calculated with tapply using barplot.
Plot bars for different conditions beside each other, not on top of each other. Check the help page for an option to do this.

Solutions:

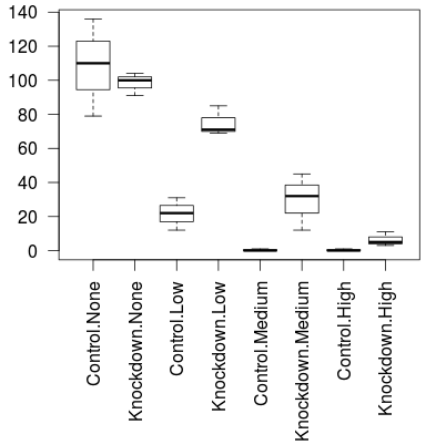
```
# Load data, order Z and plot
colony<-read.csv("2.1_colony_run.csv")
colony$Z<-factor(colony$Z,c("None","Low","Medium","High"))
attach(colony)

# just with Z
plot(Count~Z)
```

[*To make the labels visible, we'll use some graphics commands to increase the size of the lower margin and make the x- axis labels vertical (full details on this later):*
> par(oma=c(6,2,2,2))

> *boxplot(Count~X*Z,las=2)*
…]

# Analysis of Variance - just using Z
colony.aov<-aov(Count~Z)
print(summary(colony.aov))

# now with X & Z
par(oma=c(6,2,2,2))
plot(Count~X*Z)
boxplot(Count~X*Z,las=2)



Analysis of Variance - with X & Z
> colony.aov<-aov(Count~X*Z)
> print(summary(colony.aov))

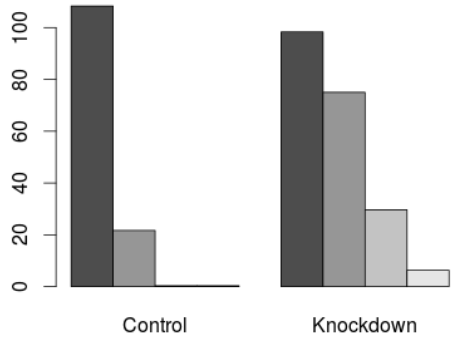
```
            Df Sum Sq Mean Sq F value   Pr(>F)
X            1   2321    2321  14.072  0.00174 **
Z            3  36150   12050  73.067 1.48e-09 ***
X:Z          3   3441    1147   6.954  0.00329 **
Residuals   16   2639     165
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

tapply with multiple variables
Including Z in the call to tapply is a little fiddly, but easy when you know how.
> colony.means<-tapply(Count,list(Z,X),mean) # Calculate group means

> print(colony.means)

```
         Control Knockdown
None   108.3333333 98.333333
Low     21.6666667 75.000000
Medium   0.3333333 29.666667
High     0.3333333  6.333333
```

# Use the **beside** option in the call to barplot (What happens if you put X first in the list?)
barplot(colony.means,**beside**=TRUE)



> detach(colony) # clear colony data frame from web space

Functions

One of the great strengths of **R** is the user's ability to add functions. In fact, many of the functions in **R** are actually functions of functions. The structure of a function is given below.

```
myfunction <- function(arg1, arg2, ... ){

statements

return(object)

}
```

Objects in the function are local to the function. The object returned can be any data type. Here is an example.

```
# function example - get measures of central tendency
# and spread for a numeric vector x. The user has a
# choice of measures and whether the results are printed.

mysummary <- function(x,npar=TRUE,print=TRUE) {
  if (!npar) {
    center <- mean(x); spread <- sd(x)
  } else {
    center <- median(x); spread <- mad(x)
  }
  if (print & !npar) {
    cat("Mean=", center, "\n", "SD=", spread, "\n")
  } else if (print & npar) {
    cat("Median=", center, "\n", "MAD=", spread, "\n")
  }
  result <- list(center=center,spread=spread)
  return(result)
}

# invoking the function
set.seed(1234)
x <- rpois(500, 4)
y <- mysummary(x)
Median= 4
MAD= 1.4826
# y$center is the median (4)
# y$spread is the median absolute deviation (1.4826)
```

```
y <- mysummary(x, npar=FALSE, print=FALSE)

# no output

# y$center is the mean (4.052)

# y$spread is the standard deviation (2.01927)
```

It can be instructive to look at the code of a function. In **R**, you can view a function's code by typing the function name without the ( ). Finally, you may want to store your own functions, and have them available in every session. You can customize the R enviroment to load your functions at start-up.

Exercise
Centigrade to Fahrenheit conversion is given by F = 9/5 * C +32.
Write a function that converts between temperatures.
The function should take two named arguments:
temperature (t) is numeric
units (unit) is character
Both arguments should have appropriate default values.
The function should report an appropriate error if inappropriate values are given.
Note: Conditional tests can be negated using **! eg. !**is.numeric(t)
if(!is.numeric(t)) { .... }

You may also find the %in% command useful, which checks to see if the elements of one vector are present in another:

```
> levels(colony$Z)
[1] "None" "Low" "Medium" "High"

> "Low" %in% colony$Z
[1] TRUE

> "Zero" %in% colony$Z
[1] FALSE

> c("None","Low") %in% colony$Z # multiple values
[1] TRUE TRUE
```

eg. if(!(unit %in% c("c","f"))){...}

The function should print out the temperature in Fahrenheit if given in Centigrade, and vice versa.
Solution

```
convert.temp <- function(t, unit="c") {
    if (!is.numeric(t)) stop ("Input should be numeric")
    if( !(unit %in% c("c","f")) ) stop ("Oops! You need to set the temperature scale: degrees C
    (c) or Fahrenheit (f)")

  converted<-0

  # Centigrade Conversion
  if ( unit=="c" ) {
   converted <- 9/5 * t + 32
  }
  # Fahrenheit Conversion
  if (unit=="f"){
   converted <- 5/9 * (t-32)
  }
  converted # this is needed - doesn't return last executed like Perl!
}
```

Or… Returning multiple values as a list:

```
convert.temp <- function(t, unit="c") {
  if (!is.numeric(t)) stop ("Input should be numeric")
  if( !(unit %in% c("c","f")) ) stop ("Oops! You need to set the temperature scale: degrees C (c) or
Fahrenheit (f)")

  converted<-0
  scale<-""

  # Conversion for centigrade
  if ( unit=="c" ) {
    converted <- 9/5 * t + 32
    scale = "Centigrade"
  }
  # Conversion for Fahrenheit
  if(unit=="f"){
    converted <- 5/9 * (t-32)
    scale = "Fahrenheit"
  }

  result <- list(startTemp=t,unit=scale,newTemp=converted)
  return(result)
```

HINT: You can check if a function is called with no arguments using **'missing'**

missing can be used to test whether a value was specified as an argument to a function.
eg. if(missing (x)) stop ("You must supply an argument, x") … etc

Advanved Data Procesing: Gene Clustering Exercise
See Basic_R_Day_2_slides

Solutions

```
#########################################################################
## 2.3_geneClustering.R                                  ##
## Exercise in writing scripts with multiple steps          ##
## Script assigns several factor objects.  Reads in 3 data     ##
## files.  Undertakes some analysis and generates results      ##
##                                                  ##
## Robert Stojnic. rs550@cam.ac.uk                        ##
#########################################################################
##
##
### START ###
#################################

########################################
## Part 1.  Build a list of all genes ##
########################################

# start out with an empty collection of genes
genes <- c()
for(fileNum in 1:5){
        # load in files 2.3_DiffGenes1.tsv ...
        t <- read.delim(paste("2.3_DiffGenes", fileNum, ".tsv", sep=""), as.is=T, header=F)
        # label the input columns to help code readability
  # When loading in character data use as.is=TRUE to
  # prevent it being converted to factors!
        names(t) <- c("gene", "expression")
        genes <- union(genes, t$gene)   # union() is a set operation
  # It combines two vectors by eliminating duplicates.
  # There are also intersect() and setdiff()
}

# for tidiness order our genes by name
genes <- sort(genes)
genes

################################################
```

```
## Part 2.  Combine data into a single table ##
################################################

# make the destination table with size n(genes) rows by n(files) columns
values <- matrix(0, nrow=length(genes), ncol=5) # populate matrix with zeros

# row are going to the complete set of genes
rownames(values) <- genes

for(fileNum in 1:5){
        # read in the file again
        t <- read.delim(paste("2.3_DiffGenes", fileNum, ".tsv", sep=""), as.is=T, header=F)
        names(t) <- c("gene", "expression")

        # match the names of the genes to the rows in our big table
        index <- match(t$gene, rownames(values))
        # use the matched indices to copy the expression levels
        values[index,fileNum] <- t$expression
}

# make the heat map with hierarchical clustering
png("geneClusters.png") # will open a file for writing
heatmap(values, scale="none", col = cm.colors(256))
dev.off() # close file

#####################################################
## Part 3.  Redo the heatmap on subset of genes ##
#####################################################

# load in the experimentally verified genes
t.exp <- read.delim("2.3_ExperimentalGenes.tsv", as.is=T)

# split all gene names by "," and then flatten it out into a single vector
experim.genes <- unlist( strsplit(t.exp$genes, ",") )
# unlist() flattens out a nested list into a single vector
# strsplit() splits a vector of strings by a custom split character (",").
# The result is a list of split values for each element of the input vector

# redo the heatmap by using just the genes in the experimentally verified set
is.experimental <- rownames(values) %in% experim.genes
#png("geneClusterExp.png")
x11() # make a new window
heatmap(values[ is.experimental, ], scale="none", col = cm.colors(256))
```

#dev.off() **# uncomment to close new window**

Graphics

> x <- 1:21
> y <- 5:25

simple
> **plot**(x,y)

more parameters
plot(x,y, xlab="X data", ylab="Ydata", xlim=c(0,10), ylim=c(0,10),main="Our title")
xlab, ylab - axes labels
xlim … axis length

points() is used to add points to an existing plot
lines() is used to add lines to an existing plot

plot(c(0, 5), c(0, 5), type="l") # draw as line from (0,0) to (5,5)
points(1, 3) # add a point at 1,3

Ex 2:
x <- seq(-2, 2, 0.1)
y <- sin(x)
plot(x,y)
plot(x~y, col=”red”, pch=”16”)

col=”red” - sets colour
pch=”16” ?

See also: boxplot, hist, barplot
data <- c("2000"=0, "2001"=20, "2002"=50, "2003"=100)
barplot(data, main="Number of R developers")

data1 <- rnorm(1000, mean=0)
data2 <- rnorm(1000, mean=1)
boxplot(data1, data2)

data <- rnorm(1000)
hist(data)

par Function

par can be used to set or query graphical parameters. Parameters can be set by specifying them as arguments to **par** in tag = value form, or by passing them as a list of tagged values.

Top level graphics function
parameter specifies various page settings. These are inherited by subordinate functions, if no other styles are set. Specific colours and styles may be set globally with par, but changed ad hoc in plotting commands

The global setting will remain unchanged, and reused in future plotting calls: par sets the size of page and figure margins: margin spacing is in 'lines'
par is responsible for controlling the number of figures that areplotted on a page
par may set global colouring of axes, text, background, foreground, line styles (solid/dashed), if figures should be boxed or open etc. etc

mfrow - allows plots side by side etc eg.

par(mfrow=c(1,1)) # one figure on page
par(oma=c(2,2,2,2)) # equal outer margins
par(mar=c(5,4,4,2)) # Sets space for x & y labels, a main title, and a thin margin on the right

par(mfrow=c(2,2)) # 2 x 2 figures per page
par(oma=c(1,0,1,0) # 1 line spacing top and bottom
par(mar=c(4,2,4,2)) # 4 lines at bottom & top 2 lines left & right
par(bg="lightblue",fg="darkgrey") # light blue background, dark grey spots
par(pch=16,cex=1.4) # Large circles for spots, Execute 4 times with different colors:
plot(1:10)
box("figure",lty=3,col="blue") # Draw a blue dashed line around plot
box("outer",lty=1,lwd=3,col="green") # Draw a green solid line around figure

Also
pch= ... Sets one of the 26 standard plotting character used. Can also use characters, such as "."

cex= ... Character expansion. Sets the scaling factor of the printing character

las= ... Axes label style. 1 normal, 2 rotated
90º - 4 styles (0-3)

Worked example - building the plot

par(mfrow=c(1,1))
par(bg="white",fg="black",cex=1)

```
par(oma=c(1,1,1,1))
par(mar=c(5,4,4,2)+0.1)

plot(1:10,main="The plot title",sub="A subtitle", xlab="Numbers",ylab="More numbers")

mtext(c("Bottom", "Left", "Top","Right"), c(1,2,3,4),line=.5)

text(2,10,"Text at X=2,Y=10")
legend(locator(1),"Some Legend",fill="red")
```

Adding a second Y Axis

```
x1<-1:20
y1<-sample(1000,20)
y2<-runif(20)
y2axis<-seq(0,1,.2) # demo data
par(mar=c(4,4,4,4)) # Set up equivalent figure margins

# Plot and label first Y series -
plot(x1,y1,type="p",pch=10,cex=2,col="red",
  main="A second axis example",
  ylab="Big values",ylim=c(0,1100),
  xlab="Ordered units")
points(x1,y1,type="l",lty=3,lwd=2,col="green") # Connect dots with a line
par(new=TRUE) # Overlay a second plot region
plot(x1,y2,type="p",pch=20,cex=2,col="black",axes=FALSE,bty="n",xlab="",ylab="")
points(x1,y2,type="l",lty=2,lwd=2,col="grey") # Plot second Y series, but suppress labels
axis(side=4,at=pretty(y2axis))
mtext("Little values",side=4,line=2.5) # Anotate second Y axis
# Add legend, note X,Y is on second Y axis scale
legend(15,0.2,c("Big Y","Little Y"),lty=1,lwd=2,col=c("green","grey"))
```

Saving plots to files

Unless specified, R plots all graphics to the screen. To send plots to a file, you need to set up an appropriate graphics device …

```
postscript(file="a_name.ps", ...)
pdf(file="...pdf", ...)
jpeg(file=" ...jpg", ...)
png(file=" ....png", ...)
```

Each graphics device will have a specific set of arguments that dictate characteristics of the outputted file:

height=, width=, horizontal=, res=, paper=
Top tip: jpg, A4 @ 300 dpi, portrait, size in pixels
jpg(file="my_Figure.jpg", height=3510, width=2490, res=300)
Postscript & pdf work in inches by default, A4 = 8.3" x 11.7"

Graphics devices need closing when printing is finished
dev.off()

for example:
png("tenPoints.png", width=300, height=300)
plot(1:10)
dev.off()

Thoughts when plotting Graphics to a file
Its very tempting to send all graphical output to a pdf file. Caution!
For high resolution publication quality images you need postscript. Set up postscript file capture with the following function
postscript("a_file.ps",paper="a4")

postscript images can be converted to JPEG using ghostscript (free to download) for low resolution lab book photos and talks. PDF images will grow too large for acrobat to render if plots contain many data points (e.g. Affymetrix MA plots)
Automatically send multiple page outputs to separate image files using ...
file="somename%02d.jpg"
Don't forget to close graphics devices (i.e. the file) by using
dev.off()

For info on manipulating chart colour palette - see the Day2 slides...

Expert stuff…

- Make a full A4 page figure comprising of 6 plots: 2 each of XY plot (**plot**()), barchart (**barplot**()) and box plots (**boxplot**())
- The two version of each plots should consistent of: the default plot and a customised plot (change for instance colours, range, captions...)
- Output the completed 6-panel figure to: screen, jpeg, postscript and pdf file

Solution
##

```
## 19_6PanelPlotscript.R                                    ##
## Graphics exercise.  Generate some high level plots. Display ##
## In various ways.  Get user to write a function             ##
##                                                  ##
## Ian Roberts.  ir210@cam.ac.uk                          ##
######################################################################
##
##
### START ###

#Make a function to draw the plots
plotXYBarBox <- function(){
        #Some arbitrary data for plotting

        # for x/y plot
        x <- 1:20
        y <- sample(1000, 20) # randomly pick y values from 1:1000

        # for bar plots, a vector of values - 10 random numbers from 1:100
        bar.data <- sample(100, 10)

        # for box plots a range of values - normal values with different means
        box.data <- replicate(5, rnorm(100, mean=runif(1)*5))

        # set up the plot with 6 panels
        par(mfrow=c(3,2))

        #Plot x/y plots

        #Default
        plot(x,y)

        #Annotated
        plot(x,y, xlab="Scatter plot X values", ylab="Y values",
                main="Scatter plot", xlim=c(0,10), ylim=c(0,1500), col="blue")
        mtext("This is \n margin text",side=4, cex=0.7, line=1)
        legend("topright",legend="random values",fill="blue")

        #Plot barplots
        #Default
        barplot(bar.data)
        #Annotated
        barplot(bar.data, xlab="Bar plot X values", ylab="Y values",
```

```
            main="Bar plot",col=rainbow(5))
        mtext("This is \n margin text",side=4, cex=0.7, line=1)

        #Plot boxplots
        #Default
        boxplot(box.data)
        #Annotated
        boxplot(box.data, xlab="Box plot X categories", ylab="Y values",
            main="Box plot", ylim=c(0,15),col=rainbow(5))
        mtext("This is \n margin text",side=4, cex=0.7, line=1)
        legend("topright",legend=1:5,fill=rainbow(5),ncol=3)
}

#Make the plots
#Plot to screen

plotXYBarBox()

#plot to postscript
postscript(file="sixPanels.ps", paper="a4", width=21, height=27,horizontal=F)
plotXYBarBox()
dev.off()

#plot to jpeg
jpeg(file="sixPanels.jpg", height=800, width=560,res=150)
par(oma=c(1,1,1,1))
plotXYBarBox()
dev.off()

#plot to pdf
pdf(file="sixPanels.pdf",paper="a4", width=21, height=27)
par(oma=c(1,1,1,1))
plotXYBarBox()
dev.off()

####################END SCRIPT########################################
```

Error Handling
To handle errors, you can use warning() or message()

But, also consider **try**() [- **look into this further**]

try is a wrapper to run an expression that might fail and allow the user's code to handle error-recovery. See ?try() for more info.
try(expr, silent = FALSE)

```
options(show.error.messages = FALSE)
try(log("a"))
print(.Last.value)
options(show.error.messages = TRUE)

## alternatively,
print(try(log("a"), TRUE))
```

More on Lists - tutorial

A list is a generic vector containing other objects.
For example, the following variable x is a list containing copies of three vectors n, s, b, and a numeric value 3.

```
n = c(2, 3, 5)
s = c("aa", "bb", "cc", "dd", "ee")
b = c(TRUE, FALSE, TRUE, FALSE, FALSE)
x = list(n, s, b, 3)   # x contains copies of n, s, b
```

List Slicing

We retrieve a list slice with the single square bracket "[]" operator. The following is a slice containing the second member of x, which is a copy of s.

```
> x[2]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"
```

With an index vector, we can retrieve a slice with multiple members. Here a slice containing the second and fourth members of x.

```
> x[c(2, 4)]
[[1]]
[1] "aa" "bb" "cc" "dd" "ee"

[[2]]
[1] 3
```

Member Reference

In order to reference a list member directly, we have to use the double square bracket "[[]]" operator. The following object x[[2]] is the second member of x. In other words, x[[2]] is a copy of s, but is not a slice containing s or its copy.

```
> x[[2]]
[1] "aa" "bb" "cc" "dd" "ee"
```

We can modify its content directly.

```
> x[[2]][1] = "ta"
> x[[2]]
[1] "ta" "bb" "cc" "dd" "ee"
> s
[1] "aa" "bb" "cc" "dd" "ee"   # s is unaffected
```

Visualisation

ggbio - http://www.bioconductor.org/packages/release/bioc/html/ggbio.html
The ggbio package extends and specializes the grammar of graphics for biological data. The graphics are designed to answer common scientific questions, in particular those often asked of high throughput genomics data.

gviz - http://www.bioconductor.org/packages/release/bioc/html/Gviz.html
Gviz uses the biomaRt and the rtracklayer packages to perform live annotation queries to Ensembl and UCSC and translates this to e.g. gene/transcript structures in viewports of the grid graphics package. This results in genomic information plotted together with your data.