Lecture 3

What did we learn last time?

- matrices: create with `matrix()`.
- Address entries in an array: `a[2,2]`, `a[1:2,1:2]`, `a[,c(2,4)]`, `a[a>8]`
- Apply a function on all rows, or all column of an array:
  - `apply(a, 1, mean)` - calculate mean of all rows.
  - `apply(a, 2, mean)` - calculate mean of all columns.

All this is very useful for bootstraping!

- Functions: `function(x) { x+1}`
- Read data from file: `read.table()`

factors

Till now we saw 3 types of vectors: numbers, booleans, and strings

```r
> a=1:4
> a
[1] 1 2 3 4
> a=c("hello","there")
> a
[1] "hello" "there"
> a=(1:4)>2
> a
[1] FALSE FALSE TRUE TRUE
```

Now we’ll learn about factors. Factors are somewhere between a string and a boolean. We use them when we have values, such as strings, that have several different types

```r
> a=sample(c("summer","winter"),20,rep=T)
> a
[1] "summer" "summer" "summer" "summer" "summer" "summer" "summer" "summer" "winter"
[9] "winter" "summer" "summer" "winter" "summer" "winter" "summer" "winter"
[17] "winter" "winter" "winter" "winter"
```

This is a vector of strings. Now we want to use the fact that we have just 2 types:

```r
> a=factor(a)
> a
1 [1] summer summer summer summer summer summer summer winter winter winter summer summer winter winter winter winter winter winter winter
Levels: summer winter
```
> a[1]
[1] summer
Levels: summer winter
> as.numeric(a)
[1] 1 1 1 1 1 1 1 1 2 1 1 2 1 2 2 2 2
> levels(a)
[1] "summer" "winter"
> levels(a)=c("s","w")
> a
[1] s s s s s s w s w s w s w w w
Levels: s w
>
This has several advantages. The main one is that it is much faster to check if factors are equal to one another - you just have to compare numbers. Another comes up later, when we learn about ANOVA.

read.table

We want to read the following table: (switch to excel)

We saved it as comma separated values.

Let us first see how we can list files in a directory:

> list.files()

[1] "ab.Rdata" "a.Rdata" "data.zip"
[4] "excer1.tm" "excer1.tm-" "exer1.tm"
[7] "exercise2solution.pdf" "exercise2solution.tm" "exercise2.tm"
[10] "exercise2.tm-" "exercise3.tm" "exercise3.tm-"
[13] "index.htm" "index.old.htm" "missfont.log"
[16] "old" "Rcourse1.pdf" "Rcourse1.tm"
[19] "Rcourse2.pdf" "Rcourse2.tm" "Rcourse3.pdf"
[22] "Rcourse3.tm" "Rcourse3.tm-" "Rcourse4.html"
[25] "Rcourse4.new.pdf" "Rcourse4_no_output.tm" "Rcourse4.pdf"
[28] "Rcourse4.tm" "Rcourse4.tm-" "Rcourse5_no_output.tm"
[31] "Rcourse5.pdf" "Rcourse5.tm" "Rcourse6_no_output.tm"
[34] "Rcourse6.pdf" "Rcourse6.tm" "Rcourse7.pdf"
[37] "Rcourse7.tm" "Rcourse7.tm-" "Rcourse8_no_output.tm"
Very easy. How can we see in which directory we are?

```r
> getwd()
```

```
[1] "/home/michael/R-course-2006/lecture3"
```

And how to change it?

```r
> setwd("~/R-course-2006/lecture3")
```

These are a bit hard to remember: setwd, and getwd for set working directory, and get working directory.

```r
> a=read.table("example3.csv",head=T,row.names=i,as.is=3, sep="",")
```

```r
a
  height email  hair color age
michael   175 lachmann brown NA
sarah     180 sarah@aol.com black  20
fred      150 f.smith@msn.net black  10
  John     210 blond     25
Bilbo      90  b.bagins@shire.me black  111
Andrew    150  andy@aol.de blond  35
```

Normally, R will convert a string column that is read into categorical data. To prevent R from doing that, you should give the `as.is` argument, with the columns that should be read as strings.

If a column has values that are essentially all different, store as string. If some values are the same and you will compare between values, use categorical data (i.e. no `as.is`)

The are many ways to access the data in the table:

```r
> a$height
```

```
[1] 175 180 150 210 90 150
```

```r
> a$hei
```

```
[1] 175 180 150 210 90 150
```

```r
> a$hai
```

```
[1] brown black blond black blond
Levels: brown blond black
```
The method using the dollar sign takes a column. You can see that you can abbreviate the name.

```r
> a[,1]
[1] 175 180 150 210 90 150
> a[1:2,]

height  email  hair.color  age
michael 175    lachmann    brown NA
sarah   180    sarah@aol.com  black  20
```

```r
> a[,"email"]
[1] "lachmann" "sarah@aol.com" "f.smith@msn.net"
[4] "b.bagins@shire.me" "andy@aol.de"
```

```r
> a [,"ema"]
Error in ".data.frame"(a , "ema") : undefined columns selected
```

You can not abreviate when you use `[`,

We want to sort the table by height. How do we do that?
Here the function `order()` helps. It gives us the order of elements.

```r
> a

height  email  hair.color  age
michael 175    lachmann    brown NA
sarah   180    sarah@aol.com  black  20
fred    150    f.smith@msn.net  10
John    210    blond  25
Bilbo   90     b.bagins@shire.me  black 111
Andrew  150    andy@aol.de  blond 35
```

```r
> order(a$height)
[1]  5  3  6  1  2  4
```

```r
> order(a$height)
> i
> i
[1]  5  3  6  1  2  4
> a[i,]

height  email  hair.color  age
Bilbo   90     b.bagins@shire.me  black 111
fred    150    f.smith@msn.net  10
Andrew  150    andy@aol.de  blond 35
michael 175    lachmann    brown NA
sarah   180    sarah@aol.com  black  20
John    210    blond  25
```

```r
> a[ order( a$height ) , ]
```
height  email  hair.color  age
Bilbo   90  b.bagins@shire.me  black  111
fred    150 f.smith@msn.net  10
Andrew  150 andy@aol.de  blond  35
michael 175 lachmann  brown NA
sarah   180 sarah@aol.com  black  20
John     210  blond  25

Names

In R, almost everything can have names:

> a=c(2,3,4)
> a
[1] 2 3 4
> a=c(x=2,y=3,z=4)
> a
 x y z
2 3 4
> names(a)
[1] "x" "y" "z"
> a=1:4
> names(a)
NULL
> names(a)=c("a","b","c","d")
> a
 a b c d
1 2 3 4
> names(a)
[1] "a" "b" "c" "d"
>
Now we see that each entry has a name.

This is useful when functions return values, because it tells us what each value is:

> a=runif(100)
> summary(a)

                     Min. 1st Qu.  Median    Mean  3rd Qu.    Max.  
-2.442000  -0.47890  0.05365   0.03399 0.556900  2.312000

> summary is a function, and the values that it returns have names:

> sum.a=summary(a)
> names(sum.a)
[1] "Min." "1st Qu." "Median" "Mean" "3rd Qu." "Max."
> sum.a["Mean"]
  Mean
  0.1538
> sum.a["Max."
  Max.
  2.637
>
matrices and data.frames have names for the rows and columns:

> a=matrix(1:6,2,3)
> a
 [,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
> rownames(a)=c("first row","second row")
> colnames(a)=c("a","b","c")
> a
   a b c
first row 1 3 5
second row 2 4 6
> a[,"a"]
   first row second row
          1       2
> a[,"b"]
   first row second row
          3       4

Row names are especially important when we want to use data from different sources: for example when we have results of one experiment for some genes, and more results of from a second experiment. The row names allow us then to quickly connect the results for the same genes.

**What is the difference between matrices and data.frames?**

A matrix basically is a vector, and as such can hold only one type of data. This can cause strange results sometime:

> x=matrix(1:6,2,3)
> x
 [,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
> x[2,2]="four"
> x
 [,1] [,2] [,3]
[1,] "1" "3" "5"
[2,] "2" "four" "6"
by changing a single entry in x, the whole matrix was changed to strings.

```r
> x <- data.frame(a=1:3, b=4:6)
> x
     a b
  1 1 1
  2 2 2
  3 3 3
> x[2,2]
[1] 5
> x[2,2] <- "five"
> x
     a b
  1 1 1
  2 2 five
  3 3 3
> x[,2]
[1] "4" "five" "6"
> x[,1]
[1] 1 2 3
>
You can see that only one column of x was changed to strings. A data.frame is a list of vectors (or matrices), all of which have the same length, and can thus be addressed as an array.

```r
> x[1]
     a
  1 1
  2 2
  3 3
> x[2]
     b
  1 4
  2 five
  3 6
>
Functions

Last time we talked a bit about functions. Now some more.

Let us define a simple function:

```r
> f <- function(x) x + 1
> f(4)
[1] 5
> f(9.3)
```
[1] 10.3
> x=17
> f(2)
[1] 3
> x
[1] 17
>
So, a function is very easy to define. We just say
function(x) followed by an expression.

> square= function(z) z^2
> square(3)
[1] 9
> square(1:5)
[1]  1  4  9 16 25
> square
  function(z) z^2

We can see that functions can simply be stored in variables, like numbers or strings.
Functions can also take several arguments:

> mult = function(x,y) x*y
> mult(2,3)
[1] 6
> mult(1:4,2:5)
[1]  2  6 12 20
>
Applying a function to a vector

> a=1:10
> f=function(x) x+1
> sapply(a,f)
[1]  2  3  4  5  6  7  8  9 10 11
>
sapply() applies the function to each element in the vector, and returns the result.
If the result is a vector, then we get a matrix:

> g=function(x) c(x,x+1)
> g(2)
[1] 2 3
> g(5)
[1] 5 6
> sapply(1:4,g)
\[
[1,] \ [2,] \ [3,] \ [4,]
[1,] \ 1 \ 2 \ 3 \ 4
[2,] \ 2 \ 3 \ 4 \ 5
\]

Sometimes we need more than one expression for the calculation. In that case we can enclose the calculations with \{\}. The result will be the last expression.

```r
> die.roll.6=function(){ x=sample(1:6,1); x==6 }
> die.roll.6()
[1] FALSE
>
Assignments in functions do not affect the outside:

```r
> x=100
> die.roll.6()
[1] FALSE
> x
[1] 100
>
To see what is happening in a function, we can add print statements:

```r
> die.roll.6=function(){ x=sample(1:6,1); print(x); x==6 }
> die.roll.6()
[1] 2
[1] FALSE
>
R also has conditional statements, and loops:

**Conditionals**

```r
> f = function(x) { if( x > 3) 4 else 5}
> f(2)
[1] 5
> f(6)
[1] 4
>
**Loops**

```r
> for(i in 1:10) print(i)
[1] 1
[1] 2
[1] 3
[1] 4
```
> a=1
> for(i in 1:10) a = a * i
> a

for has the following structure: you give a variable that will iterate over a vector or list of things.

The expression will be called with each i

> a

[1]  5  6  7  8  9 10

> a=1
> for(i in 1:10) a = a * i
> a

[1] 3628800