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 bootstrapping!

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

`read.table`

We want to read the following table:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>name</td>
<td>height</td>
<td>email</td>
<td>hair.color</td>
<td>age</td>
</tr>
<tr>
<td>2</td>
<td>michael</td>
<td>175</td>
<td>lachmann</td>
<td>brown</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>sarah</td>
<td>180</td>
<td><a href="mailto:sarah@aol.com">sarah@aol.com</a></td>
<td>black</td>
<td>20</td>
</tr>
<tr>
<td>4</td>
<td>fred</td>
<td>150</td>
<td><a href="mailto:f.smith@msn.net">f.smith@msn.net</a></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>John</td>
<td>210</td>
<td>blond</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Bilbo</td>
<td>90</td>
<td><a href="mailto:b.bagins@shire.me">b.bagins@shire.me</a></td>
<td>black</td>
<td>111</td>
</tr>
<tr>
<td>7</td>
<td>Andrew</td>
<td>150</td>
<td><a href="mailto:andy@aol.de">andy@aol.de</a></td>
<td>blond</td>
<td>35</td>
</tr>
</tbody>
</table>

We saved it as comma separated values.

```r
> a=read.table("example2.csv",head=T,row.names=1,as.is=3, sep="","
> 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
```

Normally, R will convert a string column that is read in 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:
The method using the dollar sign takes a column. You can see that you can abbreviate the name.

```
> 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
```

You can not abbreviate 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.

```
> 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
```

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

So a will be sorted by height if we write first the 5th row, then the 3rd, then the 6th, then the 1st, and so on. we can do that like this:

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

In R, almost everything can have names:

> a=1:4
> a
  [1] 1 2 3 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.

An easier way to do the same:

> x=c(first=3,second=4)
> x
  first second
       3       4

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

> a=rnorm(100)
> summary(a)
                 Min.   1st Qu.    Median     Mean   3rd Qu.     Max. 
-2.20700   -0.69920    0.20734    0.15384    0.97010    2.63700

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

```r
> 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:

```r
> 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
```

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:

```r
> x=matrix(1:6,2,3)
> x
     [,1] [,2] [,3]
[1,]  1  3  5
[2,]  2  4  6
> x[2,2]="four"
```
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 4
2 2 5
3 3 6
> x[2,2] = "five"
> x
   a  b
1 1  4
2 2  five
3 3  6
> 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
```
So, a function is very easy to define: We just say 
function(x) followed by an expression.

\[
square = \text{function}(z) \quad z^2
\]

\[
square(3)
\]

\[
[1] \quad 9
\]

\[
square(1:5)
\]

\[
[1] \quad 1 \quad 4 \quad 9 \quad 16 \quad 25
\]

We can see that functions can simply be stored in variables, like numbers or strings.

Functions can also take several arguments:

\[
mult = \text{function}(x,y) \quad x*y
\]

\[
mult(2,3)
\]

\[
[1] \quad 6
\]

\[
mult(1:4,2:5)
\]

\[
[1] \quad 2 \quad 6 \quad 12 \quad 20
\]

Applying a function to a vector

\[
a=1:10
\]

\[
f=\text{function}(x) \quad x+1
\]

\[
sapply(a,f)
\]

\[
[1] \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9 \quad 10 \quad 11
\]

\[
sapply() \quad \text{applies the function to each element in the vector, and returns the result.}
\]

If the result is a vector, then we get a matrix:

\[
g=\text{function}(x) \quad c(x,x+1)
\]

\[
g(2)
\]

\[
[1] \quad 2 \quad 3
\]

\[
g(5)
\]
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
```
```r
> a=1
> for(i in 1:10) a = a * i
> a

[1] 3628800
```

`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.