

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:

	A	B	C	D	E
1	name	height	email	hair color	age
2	michael	175	lachmann	brown	
3	sarah	180	sarah@aol.com	black	20
4	fred	150	f.smith@msn.net		10
5	John	210		blond	25
6	Bilbo	90	b.bagins@shire.me	black	111
7	Andrew	150	andy@aol.de	blond	35

We saved it as comma separated values.

```
> 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`)

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

```

> a$height
[1] 175 180 150 210 90 150
> a$hei
[1] 175 180 150 210 90 150
> a$hai
[1] brown black      blond black blond
Levels: black blond brown

```

>

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

```

```

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

```

```

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

```

>

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,]

```

	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

```
> 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=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.
-2.2070 -0.6992  0.2073  0.1538  0.9701  2.6370
```

```
>
```

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.

```
> x=data.frame(a=1:3,b=4:6)
```

```
> x
      a b
1 1 4
2 2 5
3 3 6
```

```
> x[2,2]
```

```
[1] 5
```

```
> 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.

```
> 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:

```
> 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.

```

> die.roll.6=function(){ x=sample(1:6,1); x==6 }
> die.roll.6()
[1] FALSE
>

```

Assignments in functions do not affect the outside:

```

> x=100
> die.roll.6()
[1] FALSE
> x
[1] 100
>

```

To see what is happening in a function, we can add print statements:

```

> 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

```

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

```

Loops

```

> for(i in 1:10) print(i)
[1] 1
[1] 2
[1] 3

```

```
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
> 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`.