

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()`

factors

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

```
> 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

```
> 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:

```
> a=factor(a)
> a
[1] summer summer summer summer summer summer summer winter winter summer
[11] summer winter summer winter summer 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 2 2 1 1 2 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 s w w s s w s w s w 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"

```

```
[40] "Rcourse8_no_output.tm~" "Rcourse8.pdf"          "Rcourse8.tm"
[43] "Rcourse8.tm~"           "Rcourse9.pdf"          "Rcourse9.tm"
[46] "R-data.pdf"            "rdebuts_en.pdf"       "ROracle.pdf"
[49] "solutions_excer1.pdf"  "solutions_excer1.tm"  "solutions_excer3.tm"
[52] "speeding.csv"          "strange.txt"          "style.css"
[55] "test.txt"              "tm_r"                  "transparent.gif"
[58] "usingR.pdf"           "Verzani-SimpleR.pdf"  "whole_session.Rdata"
```

Very easy. How can we see in which directory we are?

```
> getwd()
[1] "/home/michael/R-course-2006/lecture3"
```

And how to change it?

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

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

```
> a=read.table("example3.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=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=rnorm(100)
```

```
> summary(a)
```

```
      Min.   1st Qu.   Median     Mean   3rd Qu.     Max.
-2.44200 -0.47890  0.05365  0.03399  0.55690  2.31200
```

>

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.

```

> a
[1] -0.4537234085  0.2035398095  1.7588079362 -0.6350240268  0.0488713617
[6]  1.2813721351 -0.9256706566 -0.5337963296  0.2853847796 -0.0274561132
[11]  1.2784024867  0.6267928062 -0.9468052261 -0.6470745748  0.7105299717
[16]  0.4231621606  0.3594478596  0.0943223408  0.8148872209 -0.0669757749
[21] -1.7208055002  0.0820803030 -1.9929795587 -0.5633375196  0.1195273252
[26]  0.3659968602 -1.7163781670 -0.5779157421  0.0892933089  1.1566806357
[31]  0.0584212821 -0.1986729584  2.3121756933 -1.4706790483  0.4594501613
[36] -0.3360009659  0.5372507378 -0.0001737655  0.1040321510 -0.4363763243
[41] -0.8119655010 -0.7563158986 -0.2510567464 -2.4424935064  1.5112430304
[46]  0.3980827411 -0.1311156548  0.0756224658 -1.0187524709 -0.0796116363
[51] -0.0702842942 -2.2365236026 -0.2041454850 -0.4585398634 -0.7820663211
[56] -0.4606226632  1.3373184596  0.2333585851  0.3594622850  0.4003287392
[61]  2.1509508962  1.9616156493 -1.3261345898 -0.4550934939 -0.2667490071
[66] -0.6746374258  0.2568645989  0.9229310214  2.1248761735  1.2962657347
[71]  1.9618145608 -2.1831479082  0.7342159210  0.0088940067 -0.2183922710
[76] -0.2907933668 -0.3459344679 -0.9749733957 -0.3951760294  0.4937933112
[81]  0.6157331049  0.3058743228 -0.2343949332 -1.3815314838  1.1766248811
[86]  1.4235062669 -1.8261141447  0.2092363487 -0.7860781107  0.6317808849
[91]  0.8433853363 -1.8430391672  0.1755283530  0.1209463590  0.2127138408
[96] -0.0594683835  0.9916309911  1.5356038708  2.2528051869 -0.2795650909

```