BIO 754 - Lecture Week 02

03-02-2017

Solutions to Homework 01

1.

Create a character vector called FRIENDS that contains the names of 5 of your friends (whose ages should range from 25 or below, to 35 and above).

```
FRIENDS = c("Ayse", "Fatma", "Ali", "Veli", "Zeki")
```

2.

Create a second numeric vector called AGES with each of the 5 individuals' ages, and check its class:

```
AGES = c(23, 30, 40, 22, 55)
AGES
```

```
## [1] 23 30 40 22 55
```

```
class(AGES)
```

```
## [1] "numeric"
```

3.

Create a character vector called FA that repeats each friend's name the same time as his/her age:

```
FA = rep(FRIENDS, AGES)
FA
```

```
##   [1] "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"
##   [9] "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"
##  [17] "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Ayse"  "Fatma"
##  [25] "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma"
##  [33] "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma"
##  [41] "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Fatma"
##  [49] "Fatma" "Fatma" "Fatma" "Fatma" "Fatma" "Ali"   "Ali"   "Ali"
##  [57] "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"
##  [65] "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"
##  [73] "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"
##  [81] "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Ali"
##  [89] "Ali"   "Ali"   "Ali"   "Ali"   "Ali"   "Veli"  "Veli"  "Veli"
##  [97] "Veli"  "Veli"  "Veli"  "Veli"  "Veli"  "Veli"  "Veli"  "Veli"
## [105] "Veli"  "Veli"  "Veli"  "Veli"  "Veli"  "Veli"  "Veli"  "Veli"
## [113] "Veli"  "Veli"  "Veli"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
## [121] "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
```

```
## [129] "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
## [137] "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
## [145] "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
## [153] "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
## [161] "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"  "Zeki"
## [169] "Zeki"  "Zeki"
```

4.

Obtain the subset of `FRIENDS` who are older than 30 years of age:

```
FRIENDS[AGES>30]
```

```
## [1] "Ali"  "Zeki"
```

Here it is important that you use the larger than `>` operator and *not* create a new vector manually (this approach can be used to subset any vector automatically).

5.

Create a vector of *even* integers ranging from 140 to -404 and call it `x`. Then check its length:

```
x = seq(140, -404, by=-2)
length( x )
```

```
## [1] 273
```

6.

Remove `x` from your environment and check the name of the objects left in your environment:

```
rm(x)
ls()
```

```
## [1] "AGES"    "FA"      "FRIENDS"
```

Logical vectors, binary and logical operators: continued

In R the following symbols are used as binary operators, which are functions to compare two elements, or elements of vectors:

`>` : greater than

`<` : less than

`>=` : greater than or equal to

`<=` : less than or equal to

`==` : exactly equal to

`!=` : not equal to

And the following logical operators:

`!`: logical NOT

`&`: logical AND

`|`: logical OR

Arithmetic operators

To see how these work, let's first create a vector `n1` that ranges from -5 to 5 and goes in steps of 2:

```
n1 = seq(-5, 5, by=2)
n1
```

```
## [1] -5 -3 -1  1  3  5
```

Create a logical vector testing equality to `3`:

```
n1 == 3
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE FALSE
```

Let's retrieve the subset of `n1` where elements are not equal to `3`:

```
n1 != 3
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE
```

```
n1[ n1 != 3 ]
```

```
## [1] -5 -3 -1  1  5
```

Let's retrieve the subset of `n1` where elements are smaller than or equal to `1`.

```
n1 <= 1  # the logical vector
```

```
## [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

```
sum(n1 <= 1)  # the number of TRUE's
```

```
## [1] 4
```

```
n1[ n1 <= 1 ]  # the subset
```

```
## [1] -5 -3 -1  1
```

We can alternatively use the function `which`, which returns the indices of `TRUE`s in a logical vector:

```
which(n1 <= 1)  # the indices
```

```
## [1] 1 2 3 4
```

```
n1[ which(n1 <= 1) ]  # the subset
```

```
## [1] -5 -3 -1  1
```

Let's retrieve the element of `n1` with value `1` and reassign it the value `5`:

```
n1 == 1
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
n1[ n1 == 1 ]  # just obtains the 4th element
```

```
## [1] 1
```

```
n1[ n1 == 1 ] = 5  # obtains the 4th element and assigns it 5
n1
```

```
## [1] -5 -3 -1  5  3  5
```

Logical operators

The `!` operator inverts `T` into `F` and vice versa:

```
T
```

```
## [1] TRUE
```

```
!T
```

```
## [1] FALSE
```

```
n1 < 1
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

```
!(n1 < 1)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
!(!(n1 < 1))
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

The `&` operator on two or more elements returns `TRUE` only if all elements are `TRUE`, otherwise `FALSE`:

```r
T & T
```

```
## [1] TRUE
```

```r
T & F
```

```
## [1] FALSE
```

```r
F & F
```

```
## [1] FALSE
```

```r
T & T & T
```

```
## [1] TRUE
```

```r
T & T & T & F
```

```
## [1] FALSE
```

When we apply it on vectors of the same length, the elements of each vector are evaluated in sequence. Let's obtain the elements of `n1` that are both positive AND larger than `3`:

```r
n1 > 0
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```r
n1 > 3
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE
```

```r
(n1 > 0) & (n1 > 3)
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE  TRUE
```

```r
n2 = 1:6
n2
```

```
## [1] 1 2 3 4 5 6
```

```r
n1
```

```
## [1] -5 -3 -1  5  3  5
```

```r
(n1 > 0) & (n2 < 6)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

The order the two vectors are written is irrelevant:

```r
(n2 < 6) & (n1 > 0)
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE FALSE
```

We can use this to subset `n1`, `n2` or any other vector of length 6:

```r
n12sub = (n2 < 6) & (n1 > 0)
n1[n12sub]
```

```
## [1] 5 3
```

```r
n2[n12sub]
```

```
## [1] 4 5
```

```r
(100:105)[n12sub]
```

```
## [1] 103 104
```

Let's obtain the elements of `n1` that are both positive AND smaller than or equal to -3:

```r
(n1 > 0) & (n1 <= -3)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
sum ( (n1 > 0) & (n1 <= -3) )
```

```
## [1] 0
```

```r
n1[(n1 > 0) & (n1 <= -3)]
```

```
## numeric(0)
```

This means "no result". We obtain an empty vector, as both conditions cannot be satisfied.

The `|` operator on two or more elements returns `TRUE` if at least one element is `TRUE`, otherwise `FALSE`:

```r
T | T
```

```
## [1] TRUE
```

```r
T | F
```

```
## [1] TRUE
```

```r
F | F
```

```
## [1] FALSE
```

```r
F | F | F | T
```

```
## [1] TRUE
```

If you apply these operators on logical vectors, each element will be evaluated in sequence:

```r
c(T,T,F) | c(T,F,F)
```

```
## [1]  TRUE  TRUE FALSE
```

```r
c(T,T,F) & c(T,F,F)
```

```
## [1]  TRUE FALSE FALSE
```

```r
! (!c(T,T,F) | !c(T,F,F))  # this is the same as above
```

```
## [1]  TRUE FALSE FALSE
```

Let's obtain the elements of `n1` that are positive OR smaller than or equal to -3:

```r
(n1 > 0) | (n1 <= -3)
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

```r
n1 [ (n1 > 0) | (n1 <= -3) ]
```

```
## [1] -5 -3  5  3  5
```

Let's obtain the elements of `n1` that are **neither** positive NOR smaller than or equal to -3:

```r
(n1 > 0) | (n1 <= -3)
```

```
## [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
```

```r
!((n1 > 0) | (n1 <= -3))
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE FALSE
```

```r
n1 [ !((n1 > 0) | (n1 <= -3)) ]
```

```
## [1] -1
```

Because `T & T` is equivalent to `!(F | F)`, inverting statements joined using `OR` is the same as inverting each statement and combining these with `AND`.

```r
n1 [ !((n1 > 0) | (n1 <= -3)) ]
```

```
## [1] -1
```

```r
n1 [ (n1 < 0) & (n1 > -3) ]
```

```
## [1] -1
```

One note about logical vectors. Just as when you combined character with numeric vectors, all elements of the resulting vector were converted into character type, when you combine logical with numeric, all elements are converted into numeric (1 for `T` and 0 for `F`):

```r
(n1 <= -3)
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE FALSE
```

```r
c( (n1 <= -3), 10, 0)
```

```
## [1]  1  1  0  0  0  0 10  0
```

You can also convert numeric data into logical, with `0` interpreted as `FALSE`.

```r
x = c( 1.1, 10, 999, 0, -Inf)
x
```

```
## [1]   1.1  10.0 999.0   0.0  -Inf
```

```r
as.logical( x )
```

```
## [1]  TRUE  TRUE  TRUE FALSE  TRUE
```

If you reconvert to numeric, you lose information:

```r
as.numeric( as.logical( x ) )
```

```
## [1] 1 1 1 0 1
```

Exercise:

Create a 4 element numeric vector called `y` containing both positive and negative values. Then convert all the negative values to positive values.

Solution:

Create vector:

```
y = c(-10, -5, 1, 3)
y
```

```
## [1] -10  -5   1   3
```

```
y < 0
```

```
## [1]  TRUE  TRUE FALSE FALSE
```

```
y[y < 0]
```

```
## [1] -10  -5
```

```
y[y < 0] = y[y < 0] * -1
y
```

```
## [1] 10  5  1  3
```

In fact this is accomplished by the built-in function `abs`:

```
y = c(-10, -5, 1, 3)
abs(y)
```

```
## [1] 10  5  1  3
```

Note: If we cannot remember the full name of a variable or function etc. use the **Tab** key. For example, write `len` and press **Tab**. It can automatically be completed to `length`.

Matrices

The function to create a matrix is `matrix`. There is no required arguments. The most commonly used arguments are `data`, `nrow` and `ncol`.

```
matrix()
```

```
##      [,1]
## [1,]   NA
```

```
matrix(data = 1:6)
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
## [4,]    4
## [5,]    5
## [6,]    6
```

```r
matrix(data = 1:6, nrow = 3, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
matrix(1:6, 3, 2)  # this runs the same, as the default order of arguments is data, nrow and ncol
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Remember you can change the default order as long as you specify which argument you enter.

```r
matrix(1:6, 2, 3) # now we have 2 rows
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
matrix(1:6, ncol = 2, nrow = 3) # not the same
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Note that R places input data along the columns first.

Create a matrix with numbers 1 to 300, with 50 rows, and call it `me`:

```r
me = matrix(1:300,50,6)
```

You can use the functions `dim` to check dimensions, and `head` and `tail` to check the first or last rows, respectively (by default 6 rows). The function `summary` calculates some statistics on each column:

```r
dim(me)
```

```
## [1] 50  6
```

```r
head(me)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1   51  101  151  201  251
## [2,]    2   52  102  152  202  252
## [3,]    3   53  103  153  203  253
## [4,]    4   54  104  154  204  254
## [5,]    5   55  105  155  205  255
## [6,]    6   56  106  156  206  256
```

```r
head(me, 10)  # first 10 rows
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6]
##  [1,]    1   51  101  151  201  251
##  [2,]    2   52  102  152  202  252
##  [3,]    3   53  103  153  203  253
##  [4,]    4   54  104  154  204  254
##  [5,]    5   55  105  155  205  255
##  [6,]    6   56  106  156  206  256
##  [7,]    7   57  107  157  207  257
##  [8,]    8   58  108  158  208  258
##  [9,]    9   59  109  159  209  259
## [10,]   10   60  110  160  210  260
```

```r
tail(me)  # last 6 rows
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6]
## [45,]   45   95  145  195  245  295
## [46,]   46   96  146  196  246  296
## [47,]   47   97  147  197  247  297
## [48,]   48   98  148  198  248  298
## [49,]   49   99  149  199  249  299
## [50,]   50  100  150  200  250  300
```

```r
summary(me)
```

```
##        V1              V2               V3               V4
##  Min.   : 1.00   Min.   : 51.00   Min.   :101.0   Min.   :151.0
##  1st Qu.:13.25   1st Qu.: 63.25   1st Qu.:113.2   1st Qu.:163.2
##  Median :25.50   Median : 75.50   Median :125.5   Median :175.5
##  Mean   :25.50   Mean   : 75.50   Mean   :125.5   Mean   :175.5
##  3rd Qu.:37.75   3rd Qu.: 87.75   3rd Qu.:137.8   3rd Qu.:187.8
##  Max.   :50.00   Max.   :100.00   Max.   :150.0   Max.   :200.0
##        V5              V6
##  Min.   :201.0   Min.   :251.0
##  1st Qu.:213.2   1st Qu.:263.2
##  Median :225.5   Median :275.5
##  Mean   :225.5   Mean   :275.5
##  3rd Qu.:237.8   3rd Qu.:287.8
##  Max.   :250.0   Max.   :300.0
```

When subsetting matrices, we use square brackets, within which the first number or vector indicates indices of **rows**, and the second number, after the comma, indicates indices of **columns**. E.g. call the 2nd row:

```r
me[2,]
```

```
## [1]   2  52 102 152 202 252
```

```r
me[,2]  # and the second column
```

```
##  [1]  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67
## [18]  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
## [35]  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
```

Call the element in the 1st row, the 3rd column:

```
me[1,3]
```

```
## [1] 101
```

Now call the 2nd and 3rd rows simultaneously. Store the resulting object with the name `me2`, and check its dimensions and class:

```
me[2:3,]
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    2   52  102  152  202  252
## [2,]    3   53  103  153  203  253
```

```
me2 = me[2:3,]
dim(me2)
```

```
## [1] 2 6
```

```
class(me2)
```

```
## [1] "matrix"
```

```
# compare with
class(me[2,])
```

```
## [1] "integer"
```

Note that the previous subsetting commands returned single values or vectors, but `me[2:3,]` returns a `matrix` itself.

Take the subset of `me` with even number of rows, and check the dimension:

```
dim( me[seq(2,50,by=2),] )
```

```
## [1] 25  6
```

We can also use logical vectors to subset a matrix. What is critical is to be careful about not confusing rows and columns. Let's choose the columns of `me` where the 2nd row has values `>100`:

```
me[2,] > 100
```

```
## [1] FALSE FALSE  TRUE  TRUE  TRUE  TRUE
```

```r
choose1 = me[2,] > 100
class( choose1 )
```

```
## [1] "logical"
```

```r
sum( choose1 )
```

```
## [1] 4
```

```r
head( me[,choose1] )
```

```
##      [,1] [,2] [,3] [,4]
## [1,]  101  151  201  251
## [2,]  102  152  202  252
## [3,]  103  153  203  253
## [4,]  104  154  204  254
## [5,]  105  155  205  255
## [6,]  106  156  206  256
```

```r
dim( me[,choose1] )
```

```
## [1] 50  4
```

Or the subset of `me` where the 1st column has values `>45` OR `<6`:

```r
(me[,1] < 6) | (me[,1] > 45)
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
choose2 = (me[,1] < 6) | (me[,1] > 45)
class(choose2)
```

```
## [1] "logical"
```

```r
sum(choose2)
```

```
## [1] 10
```

```r
me[choose2,]
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6]
##  [1,]    1   51  101  151  201  251
##  [2,]    2   52  102  152  202  252
##  [3,]    3   53  103  153  203  253
```

```
##  [4,]    4   54  104  154  204  254
##  [5,]    5   55  105  155  205  255
##  [6,]   46   96  146  196  246  296
##  [7,]   47   97  147  197  247  297
##  [8,]   48   98  148  198  248  298
##  [9,]   49   99  149  199  249  299
## [10,]   50  100  150  200  250  300
```

You can also use the functions `ncol` and `nrow` to calculate the number of columns and rows, respectively:

```
ncol(me)
```

```
## [1] 6
```

```
nrow(me)
```

```
## [1] 50
```

Subsetting by removing an element is also possible:

```
n1[-3]  # what we learned on vectors
```

```
## [1] -5 -3  5  3  5
```

```
head(me[,-3])  # retrieve the matrix except the 3rd column
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1   51  151  201  251
## [2,]    2   52  152  202  252
## [3,]    3   53  153  203  253
## [4,]    4   54  154  204  254
## [5,]    5   55  155  205  255
## [6,]    6   56  156  206  256
```

```
head(me[,-c(1,4)])  # retrieve the matrix except the 1st and 4th columns
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   51  101  201  251
## [2,]   52  102  202  252
## [3,]   53  103  203  253
## [4,]   54  104  204  254
## [5,]   55  105  205  255
## [6,]   56  106  206  256
```

```
head(me[,c(-1,-4)]) # this also works
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   51  101  201  251
## [2,]   52  102  202  252
## [3,]   53  103  203  253
## [4,]   54  104  204  254
## [5,]   55  105  205  255
## [6,]   56  106  206  256
```

Arithmetic or logical functions on matrices apply the function on each element while retaining the matrix structure:

```r
n1 * 2
```

```
## [1] -10  -6  -2  10   6  10
```

```r
head(me - 10)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   -9   41   91  141  191  241
## [2,]   -8   42   92  142  192  242
## [3,]   -7   43   93  143  193  243
## [4,]   -6   44   94  144  194  244
## [5,]   -5   45   95  145  195  245
## [6,]   -4   46   96  146  196  246
```

```r
head(me ^ 2)
```

```
##      [,1] [,2]  [,3]  [,4]  [,5]  [,6]
## [1,]    1 2601 10201 22801 40401 63001
## [2,]    4 2704 10404 23104 40804 63504
## [3,]    9 2809 10609 23409 41209 64009
## [4,]   16 2916 10816 23716 41616 64516
## [5,]   25 3025 11025 24025 42025 65025
## [6,]   36 3136 11236 24336 42436 65536
```

```r
head(me >= 100)
```

```
##       [,1]  [,2] [,3] [,4] [,5] [,6]
## [1,] FALSE FALSE TRUE TRUE TRUE TRUE
## [2,] FALSE FALSE TRUE TRUE TRUE TRUE
## [3,] FALSE FALSE TRUE TRUE TRUE TRUE
## [4,] FALSE FALSE TRUE TRUE TRUE TRUE
## [5,] FALSE FALSE TRUE TRUE TRUE TRUE
## [6,] FALSE FALSE TRUE TRUE TRUE TRUE
```

```r
class(me >= 100)
```

```
## [1] "matrix"
```

```r
head(me[me >= 100])  # now the result is converted into a numeric vector
```

```
## [1] 100 101 102 103 104 105
```

```r
length(me[me >= 100])
```

```
## [1] 201
```

```r
class(me[me >= 100])
```

```
## [1] "integer"
```

You can also sum over matrices, numeric or logical:

```r
sum(me)
```

```
## [1] 45150
```

```r
sum(me >= 100)  # number of elements of me >= 100
```

```
## [1] 201
```

You can replace elements of matrices, just like you did for vectors:

```r
me[1,3] = -40
head(me)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1   51  -40  151  201  251
## [2,]    2   52  102  152  202  252
## [3,]    3   53  103  153  203  253
## [4,]    4   54  104  154  204  254
## [5,]    5   55  105  155  205  255
## [6,]    6   56  106  156  206  256
```

```r
me[,2] = 0
head(me)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    0  -40  151  201  251
## [2,]    2    0  102  152  202  252
## [3,]    3    0  103  153  203  253
## [4,]    4    0  104  154  204  254
## [5,]    5    0  105  155  205  255
## [6,]    6    0  106  156  206  256
```

```r
tail(me)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6]
## [45,]   45    0  145  195  245  295
## [46,]   46    0  146  196  246  296
## [47,]   47    0  147  197  247  297
## [48,]   48    0  148  198  248  298
## [49,]   49    0  149  199  249  299
## [50,]   50    0  150  200  250  300
```

Now replace the submatrix between 3rd and 4th columns, and 3rd and 4th rows, with each elements' squared values:

```r
me[3:4, 3:4]
```

```
##      [,1] [,2]
## [1,]  103  153
## [2,]  104  154
```

```r
me[3:4, 3:4]^2
```

```
##       [,1]  [,2]
## [1,] 10609 23409
## [2,] 10816 23716
```

```r
me[3:4, 3:4] = me[3:4, 3:4]^2
head(me)
```

```
##      [,1] [,2]  [,3]  [,4] [,5] [,6]
## [1,]    1    0   -40   151  201  251
## [2,]    2    0   102   152  202  252
## [3,]    3    0 10609 23409  203  253
## [4,]    4    0 10816 23716  204  254
## [5,]    5    0   105   155  205  255
## [6,]    6    0   106   156  206  256
```

Convert all the values larger than `50` into their negatives:

```r
me[me > 50] = -me[me > 50]
head(me)
```

```
##      [,1] [,2]   [,3]   [,4] [,5] [,6]
## [1,]    1    0    -40   -151 -201 -251
## [2,]    2    0   -102   -152 -202 -252
## [3,]    3    0 -10609 -23409 -203 -253
## [4,]    4    0 -10816 -23716 -204 -254
## [5,]    5    0   -105   -155 -205 -255
## [6,]    6    0   -106   -156 -206 -256
```

`rbind` and `cbind` to create/enlarge matrices

You can use the functions `rbind` and `cbind` to joing matrices and/or vectors by their rows or columns, respectively, into new matrices:

```r
rbind(1:6, 5:10)  # join 2 vectors into a matrix
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    5    6    7    8    9   10
```

```r
ne = rbind(1:6, 5:10)
dim(ne)
```

```
## [1] 2 6
```

```r
class(ne)
```

```
## [1] "matrix"
```

Joining vectors of different size is possible but R gives a warning. Here it works by recycling the shorter element:

```r
rbind(1:6, 5:11)
```

```
## Warning in rbind(1:6, 5:11): number of columns of result is not a multiple
## of vector length (arg 1)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    1    2    3    4    5    6    1
## [2,]    5    6    7    8    9   10   11
```

Add a vector `1:6` as a new bottom row to `me`:

```r
me = rbind(me, 1:6)
dim(me)  # one more row added
```

```
## [1] 51  6
```

```r
tail(me)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6]
## [46,]   46    0 -146 -196 -246 -296
## [47,]   47    0 -147 -197 -247 -297
## [48,]   48    0 -148 -198 -248 -298
## [49,]   49    0 -149 -199 -249 -299
## [50,]   50    0 -150 -200 -250 -300
## [51,]    1    2    3    4    5    6
```

However, the below code will not work (although it works with vectors):

```r
me[52,]
```

```
## Error in me[52, ]: subscript out of bounds
```

```r
me[52,] = 5:10
```

```
## Error in `[<-`(`*tmp*`, 52, , value = 5:10): subscript out of bounds
```

```
n1[10]  # vectors behave differently, you can call elements outside their range and you get NA
```

```
## [1] NA
```

Bind a vector `51:1` as the first column of `me`:

```
me = cbind(51:1, me)
dim(me)
```

```
## [1] 51  7
```

```
head(me)
```

```
##      [,1] [,2] [,3]   [,4]   [,5] [,6] [,7]
## [1,]   51    1    0    -40   -151 -201 -251
## [2,]   50    2    0   -102   -152 -202 -252
## [3,]   49    3    0 -10609 -23409 -203 -253
## [4,]   48    4    0 -10816 -23716 -204 -254
## [5,]   47    5    0   -105   -155 -205 -255
## [6,]   46    6    0   -106   -156 -206 -256
```

You can join more vectors, and also character vectors, which then produces matrices with only character elements.

```
ne2 = cbind(1:6, 5:10, c("a", "b", "c", "d", "e", "f"))
ne2
```

```
##      [,1] [,2] [,3]
## [1,] "1"  "5"  "a"
## [2,] "2"  "6"  "b"
## [3,] "3"  "7"  "c"
## [4,] "4"  "8"  "d"
## [5,] "5"  "9"  "e"
## [6,] "6"  "10" "f"
```

This is because by default, matrices can carry *only one class* of element, just like vectors. You cannot run arithmetic on the new object:

```
sum(ne2)
```

```
## Error in sum(ne2): invalid 'type' (character) of argument
```

Assigning names to vectors and matrices

You can assign names to vector elements, or matrix rows or columns using the function `names`. You can then use these to call the corresponding elements/rows/columns. Here we use the `letters` data object to obtain the alphabet letters:

```r
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```r
names(n1)
```

```
## NULL
```

```r
names(n1) = letters[1:6]
n1
```

```
##  a  b  c  d  e  f
## -5 -3 -1  5  3  5
```

```r
names(n1)
```

```
## [1] "a" "b" "c" "d" "e" "f"
```

```r
n1["f"]
```

```
## f
## 5
```

You can reassign names or reassign values using names:

```r
names(n1)[3] = "t"
n1
```

```
##  a  b  t  d  e  f
## -5 -3 -1  5  3  5
```

```r
n1["b"] = 2
n1
```

```
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
```

The same for matrices:

```r
rownames(me) = 1001:1051
colnames(me) = letters[1:7]
head(me)
```

```
##       a b c      d      e    f    g
## 1001 51 1 0    -40   -151 -201 -251
## 1002 50 2 0   -102   -152 -202 -252
## 1003 49 3 0 -10609 -23409 -203 -253
## 1004 48 4 0 -10816 -23716 -204 -254
## 1005 47 5 0   -105   -155 -205 -255
## 1006 46 6 0   -106   -156 -206 -256
```

```r
me[,"d"]
```

```
##   1001   1002   1003   1004   1005   1006   1007   1008   1009   1010
##    -40   -102 -10609 -10816   -105   -106   -107   -108   -109   -110
##   1011   1012   1013   1014   1015   1016   1017   1018   1019   1020
##   -111   -112   -113   -114   -115   -116   -117   -118   -119   -120
##   1021   1022   1023   1024   1025   1026   1027   1028   1029   1030
##   -121   -122   -123   -124   -125   -126   -127   -128   -129   -130
##   1031   1032   1033   1034   1035   1036   1037   1038   1039   1040
##   -131   -132   -133   -134   -135   -136   -137   -138   -139   -140
##   1041   1042   1043   1044   1045   1046   1047   1048   1049   1050
##   -141   -142   -143   -144   -145   -146   -147   -148   -149   -150
##   1051
##      3
```

Note that the row names (`1001:1051`) are now attached to the resulting vector as vector names.

```r
me[,"d"]["1020"]
```

```
## 1020
## -120
```

Data frames

Data frames look like matrices (have 2 dimensions), but unlike a matrix, their columns can contain data of different classes. You can convert a matrix into a data frame or combine vectors into data frames. One other difference is that data frames always have column names, and you can call their columns using the `$` notation:

```r
ne
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    5    6    7    8    9   10
```

```r
class(ne)
```

```
## [1] "matrix"
```

```r
ned = data.frame(ne)
ned  # column names automatically generated
```

```
##   X1 X2 X3 X4 X5 X6
## 1  1  2  3  4  5  6
## 2  5  6  7  8  9 10
```

```r
class(ned)
```

```
## [1] "data.frame"
```

```
ned$X3
```

```
## [1] 3 7
```

You can also convert the matrix `me` into data frame, and use the `$` notation to call the column named `d`:

```
med = data.frame(me)
class(med)
```

```
## [1] "data.frame"
```

```
med$d
```

```
##  [1]    -40   -102 -10609 -10816   -105   -106   -107   -108   -109   -110
## [11]   -111   -112   -113   -114   -115   -116   -117   -118   -119   -120
## [21]   -121   -122   -123   -124   -125   -126   -127   -128   -129   -130
## [31]   -131   -132   -133   -134   -135   -136   -137   -138   -139   -140
## [41]   -141   -142   -143   -144   -145   -146   -147   -148   -149   -150
## [51]      3
```

```
me$d  # this won't work
```

```
## Error in me$d: $ operator is invalid for atomic vectors
```

You can create a data frame as follows, defining the column names along with the input data, with different data types in each column:

```
d1 = data.frame(num1=1:6, num2=seq(-5,5,by=2), char1=rep(letters[1:3], 2))
d1
```

```
##   num1 num2 char1
## 1    1   -5     a
## 2    2   -3     b
## 3    3   -1     c
## 4    4    1     a
## 5    5    3     b
## 6    6    5     c
```

```
class(d1)
```

```
## [1] "data.frame"
```

```
d1$num1
```

```
## [1] 1 2 3 4 5 6
```

```r
class(d1$num1)
```

```
## [1] "integer"
```

```r
d1$num2
```

```
## [1] -5 -3 -1  1  3  5
```

```r
class(d1$num2)
```

```
## [1] "numeric"
```

```r
d1$char1
```

```
## [1] a b c a b c
## Levels: a b c
```

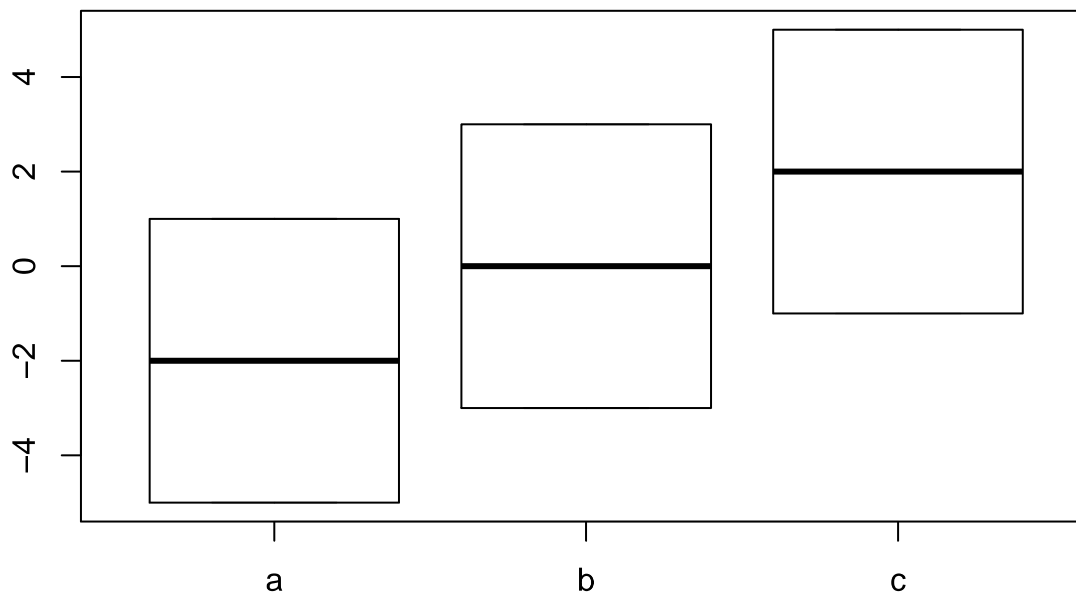
```r
class(d1$char1)
```

```
## [1] "factor"
```

As you notice, the numeric data are *not* converted into character.

You also notice that the character vector in the last column is converted into a new class, `factor`. This is a type of vector that is used to define categories, and keeps in memory the different types of unique categories (called `levels` in R) of that vector. We will learn about these later, but as an example, you can draw boxplots for data in each category level defined by the 3rd column:

```r
boxplot(d1$num2 ~ d1$char1)
```



Another example:

```
d2 = data.frame(myFriends=FRIENDS, theirAges=AGES)
d2
```

```
##   myFriends theirAges
## 1      Ayse        23
## 2     Fatma        30
## 3       Ali        40
## 4      Veli        22
## 5      Zeki        55
```

```
d2[d2$myFriends == "Fatma", ]
```

```
##   myFriends theirAges
## 2     Fatma        30
```

To check another example let's have a look at built-in R datasets. R has many small datasets that are available to play with, a list of which you can check by saying: `library(help = "datasets")`.

Here we will use a dataset from a case-control study of esophageal cancer conducted in France, where the number of cases and controls were determined and categorised based on their age, alcohol consumption and tobacco consumption. There are in total 88 combinations of category levels.

```
?esoph
```

To obtain an overview of a matrix or data frame, the `summary` function is quite useful:

```
class(esoph)
```

```
## [1] "data.frame"
```

```
head(esoph)
```

```
##   agegp     alcgp    tobgp ncases ncontrols
## 1 25-34 0-39g/day 0-9g/day      0        40
## 2 25-34 0-39g/day    10-19      0        10
## 3 25-34 0-39g/day    20-29      0         6
## 4 25-34 0-39g/day      30+      0         5
## 5 25-34     40-79 0-9g/day      0        27
## 6 25-34     40-79    10-19      0         7
```

```
tail(esoph)
```

```
##    agegp  alcgp    tobgp ncases ncontrols
## 83   75+  40-79    20-29      0         3
## 84   75+  40-79      30+      1         1
## 85   75+ 80-119 0-9g/day      1         1
## 86   75+ 80-119    10-19      1         1
## 87   75+   120+ 0-9g/day      2         2
## 88   75+   120+    10-19      1         1
```

```r
dim(esoph)
```

```
## [1] 88  5
```

```r
summary(esoph)
```

```
##    agegp          alcgp         tobgp        ncases         ncontrols
##  25-34:15   0-39g/day:23   0-9g/day:24   Min.   : 0.000   Min.   : 1.00
##  35-44:15   40-79    :23   10-19   :24   1st Qu.: 0.000   1st Qu.: 3.00
##  45-54:16   80-119   :21   20-29   :20   Median : 1.000   Median : 6.00
##  55-64:16   120+     :21   30+     :20   Mean   : 2.273   Mean   :11.08
##  65-74:15                                3rd Qu.: 4.000   3rd Qu.:14.00
##  75+  :11                                Max.   :17.000   Max.   :60.00
```

Note that summary only produces statistics of the data distribution for the last 2 columns, which are numeric. The first 3 are categorical, and therefore only their levels are listed and the corresponding frequencies.

Let's have a look at the 1st and last columns:

```r
esoph[,1]
```

```
##  [1] 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34
## [12] 25-34 25-34 25-34 25-34 35-44 35-44 35-44 35-44 35-44 35-44 35-44
## [23] 35-44 35-44 35-44 35-44 35-44 35-44 35-44 35-44 45-54 45-54 45-54
## [34] 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54
## [45] 45-54 45-54 55-64 55-64 55-64 55-64 55-64 55-64 55-64 55-64 55-64
## [56] 55-64 55-64 55-64 55-64 55-64 55-64 55-64 65-74 65-74 65-74 65-74
## [67] 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74
## [78] 75+   75+   75+   75+   75+   75+   75+   75+   75+   75+   75+
## Levels: 25-34 < 35-44 < 45-54 < 55-64 < 65-74 < 75+
```

```r
esoph$agegp  # again the 1st column, called by its name
```

```
##  [1] 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34 25-34
## [12] 25-34 25-34 25-34 25-34 35-44 35-44 35-44 35-44 35-44 35-44 35-44
## [23] 35-44 35-44 35-44 35-44 35-44 35-44 35-44 35-44 45-54 45-54 45-54
## [34] 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54 45-54
## [45] 45-54 45-54 55-64 55-64 55-64 55-64 55-64 55-64 55-64 55-64 55-64
## [56] 55-64 55-64 55-64 55-64 55-64 55-64 55-64 65-74 65-74 65-74 65-74
## [67] 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74 65-74
## [78] 75+   75+   75+   75+   75+   75+   75+   75+   75+   75+   75+
## Levels: 25-34 < 35-44 < 45-54 < 55-64 < 65-74 < 75+
```

```r
class(esoph$agegp)
```

```
## [1] "ordered" "factor"
```

```r
esoph[,ncol(esoph)]  # a way to call the last column
```

```
##  [1] 40 10  6  5 27  7  4  7  2  1  2  1  1  1  2 60 14  7  8 35 23 14  8
## [24] 11  6  2  1  3  3  4 46 18 10  4 38 21 15  7 16 14  5  4  4  4  3  4
## [47] 49 22 12  6 40 21 17  6 18 15  6  4 10  7  3  6 48 14  7  2 34 10  9
## [70] 13 12  3  1  4  2  1  1 18  6  3  5  3  3  1  1  1  2  1
```

```r
esoph$ncontrols
```

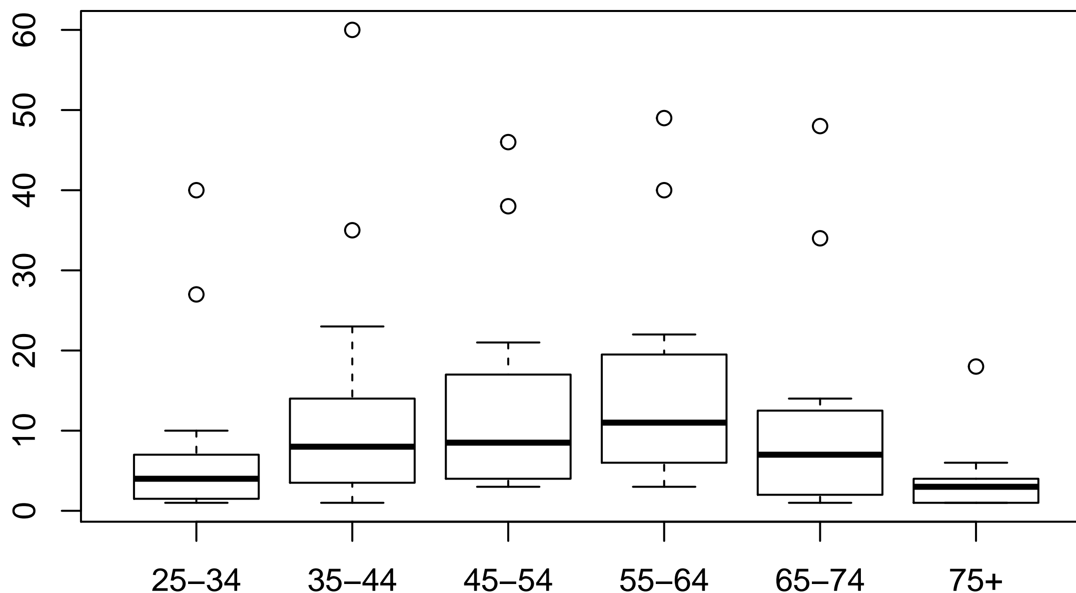
```
##  [1] 40 10  6  5 27  7  4  7  2  1  2  1  1  1  2 60 14  7  8 35 23 14  8
## [24] 11  6  2  1  3  3  4 46 18 10  4 38 21 15  7 16 14  5  4  4  4  3  4
## [47] 49 22 12  6 40 21 17  6 18 15  6  4 10  7  3  6 48 14  7  2 34 10  9
## [70] 13 12  3  1  4  2  1  1 18  6  3  5  3  3  1  1  1  2  1
```

```r
class(esoph$ncontrols)
```

```
## [1] "numeric"
```

As the age ranges of patients are defined as `factor`, we can conveniently plot the number of controls depending on age class. The `boxplot` function automatically sorts the numeric data corresponding to each class, and plots them:

```r
boxplot( esoph$ncontrols ~ esoph$agegp )
```



We will learn more on this shortly.

List objects

The final type of object we will learn is `list`. Just as data frames are flexible versions of matrices that can contain different types of variables, lists are flexible versions of vectors, which can contain elements of different classes, and dimensions.

To create a list, use the function `list`:

```r
l1 = list(n1, letters[1:10], me[1:4,], 1:6)
l1
```

```
## [[1]]
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
##
## [[2]]
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
##
## [[3]]
##       a b c      d      e    f    g
## 1001 51 1 0    -40   -151 -201 -251
## 1002 50 2 0   -102   -152 -202 -252
## 1003 49 3 0 -10609 -23409 -203 -253
## 1004 48 4 0 -10816 -23716 -204 -254
##
## [[4]]
## [1] 1 2 3 4 5 6
```

```r
length(l1)
```

```
## [1] 4
```

```r
dim(l1)
```

```
## NULL
```

Note that the first element of our list is a vector, second element is a character, third one is a matrix and the last one is again a vector. Also note that the list is not 2D, but only has length.

There are two ways to subset a list. If you use single squared brackets you retrieve the subset still as a list:

```r
l1[1]
```

```
## [[1]]
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
```

```r
class(l1[1])
```

```
## [1] "list"
```

```r
l1[3:4]
```

```
## [[1]]
##       a b c      d      e    f    g
## 1001 51 1 0    -40   -151 -201 -251
## 1002 50 2 0   -102   -152 -202 -252
## 1003 49 3 0 -10609 -23409 -203 -253
## 1004 48 4 0 -10816 -23716 -204 -254
##
## [[2]]
## [1] 1 2 3 4 5 6
```

```r
class(l1[3:4])
```

```
## [1] "list"
```

```r
l1[c(1,4)]
```

```
## [[1]]
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
##
## [[2]]
## [1] 1 2 3 4 5 6
```

In all cases, the result is a list. To retrieve specific elements of a list in their original format (e.g. as vectors or matrices), you use double square brackets `[[ ]]`.

```r
l1[[1]]
```

```
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
```

```r
class(l1[[1]])
```

```
## [1] "numeric"
```

```r
l1[[2]]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```r
class(l1[[2]])
```

```
## [1] "character"
```

```r
l1[[3]]
```

```
##       a b c      d      e    f    g
## 1001 51 1 0    -40   -151 -201 -251
## 1002 50 2 0   -102   -152 -202 -252
## 1003 49 3 0 -10609 -23409 -203 -253
## 1004 48 4 0 -10816 -23716 -204 -254
```

```r
l1[[2:3]]  # this may not return what you wished
```

```
## [1] "c"
```

The latter code doesn't return you the 2nd and 3rd elements of the list, but the 3rd element of the list's 2nd element.

So this will work, as you retrieve a vector and run an arithmetic operation on it:

```
l1[[1]] + 5
```

```
##  a  b  t  d  e  f
##  0  7  4 10  8 10
```

But not this one, because `l1[1]` is still a `list`:

```
l1[1] + 5
```

```
## Error in l1[1] + 5: non-numeric argument to binary operator
```

You can retrieve subsets of subsets:

```
l1[[1]][5]
```

```
## e
## 3
```

```
l1[[2]][3]
```

```
## [1] "c"
```

```
l1[[3]][1,2]
```

```
## [1] 1
```

You can name lists as well:

```
names(l1) = c("ayse", "fatma", "emine", "hatice")
l1
```

```
## $ayse
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
##
## $fatma
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
##
## $emine
##       a b c      d      e    f    g
## 1001 51 1 0    -40   -151 -201 -251
## 1002 50 2 0   -102   -152 -202 -252
## 1003 49 3 0 -10609 -23409 -203 -253
## 1004 48 4 0 -10816 -23716 -204 -254
##
## $hatice
## [1] 1 2 3 4 5 6
```

```r
l1$ayse
```

```
##  a  b  t  d  e  f
## -5  2 -1  5  3  5
```

```r
l1$ayse["e"]
```

```
## e
## 3
```

An alternative is naming lists while creating them:

```r
l2 = list(myFriends=FRIENDS, theirAges=AGES)
l2
```

```
## $myFriends
## [1] "Ayse"  "Fatma" "Ali"   "Veli"  "Zeki"
##
## $theirAges
## [1] 23 30 40 22 55
```

```r
l2$myFriends[2]
```

```
## [1] "Fatma"
```

```r
l2$theirAges[2]
```

```
## [1] 30
```

The random number generator and `runif`

In many types of statistical analyses we use random simulations. These can be used to:

- Create null distributions to compare with the observed data and estimate statistical significance,
- Estimate unknown population parameters, by comparing the observed data with data simulated across a range of parameter values,
- To make predictions of outcomes in stochastic models (with uncertainty).

To run random simulations we need to produce random data, for which R has a built-in **random number generator (RNG)** function. The functions produce random-like data. The simplest version draws random data from the uniform distribution, between `0` and `1`. Its argument defines how many values it should draw:

```r
runif(1)  # everyone should be obtaining different numbers
```

```
## [1] 0.5527402
```

```r
runif(1)
```

```
## [1] 0.3966875
```

```r
runif(10)
```

```
##  [1] 0.8517917 0.6086973 0.9120233 0.3785479 0.6719675 0.8918534 0.2856304
##  [8] 0.8242832 0.4914492 0.7593137
```

In fact, the data is not really random but product of an algorithm that creates **pseudo-random** numbers, i.e. data that do *not* follow any obvious pattern, but are actually produced by a deterministic algorithm.

The function starts from a number, called the **seed**, and recursively creates a chain of numbers using a specific algoritm (see https://en.wikipedia.org/wiki/Pseudorandom_number_generator). If we set the seed to the same number, we will obtain the same set of values:

```r
set.seed(1); runif(10)
```

```
##  [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968
##  [7] 0.94467527 0.66079779 0.62911404 0.06178627
```

```r
set.seed(1); runif(10)
```

```
##  [1] 0.26550866 0.37212390 0.57285336 0.90820779 0.20168193 0.89838968
##  [7] 0.94467527 0.66079779 0.62911404 0.06178627
```

```r
set.seed(-15789544); runif(10)
```

```
##  [1] 0.17720190 0.14120542 0.81872758 0.46575383 0.17011815 0.74524598
##  [7] 0.24398009 0.04199975 0.34042047 0.03052069
```

```r
set.seed(-15789544); runif(10)
```

```
##  [1] 0.17720190 0.14120542 0.81872758 0.46575383 0.17011815 0.74524598
##  [7] 0.24398009 0.04199975 0.34042047 0.03052069
```

By default, the seed is set using system time and process ID, which should be unique in each R session, ensuring that no two sessions produce the same numbers. But being able to set the seed yourself is useful as it allows to produce reproducible "random simulations".

Testing the law of large numbers using `runif`

Exercise

If the data comes from a uniform distribution, we'd expect 50% of the results to be `<0.5`. Produce 1000 values using `runif` and the proportion of values that are `<0.5`:

Solution

We need to use the binary operator, and the `sum` function (not `length`). Let's first write code to calculate the sum:

```r
length( runif(1000) < 0.5 )  # this counts all instances, not TRUEs
```

```
## [1] 1000
```

```r
sum( runif(1000) < 0.5 )  # this counts only TRUEs
```

```
## [1] 511
```

```r
sum( runif(1000) < 0.5 )  # a different from the above as it's an independent run
```

```
## [1] 516
```

```r
# and now the proportion
sum( runif(1000) < 0.5 )/1000
```

```
## [1] 0.498
```

```r
sum( runif(1000) < 0.5 )/1000
```

```
## [1] 0.503
```

If these are random numbers, do you expect the relative **deviation from the expectation** to increase or to decrease when we draw *more numbers*?

To address this question, let's compare the proportions of values `<0.5` when you draw just `100` values, and when you draw `1000000` values. Run the algorithm 3 times for each, and calculate the absolute difference from your expectation `0.5`. We can make this simpler by defining a variable `N` that denotes the number of draws. Because you use this parameter value multiple times in your command, and you will change its value from `100` to `1000000` (after by copying and pasting your earlier command), it is safer to define it as a separate variable and recycle:

```r
N = 100
sum( runif(N) < 0.5 )/N
```

```
## [1] 0.47
```

```r
sum( runif(N) < 0.5 )/N
```

```
## [1] 0.55
```

```r
sum( runif(N) < 0.5 )/N
```

```
## [1] 0.46
```

```r
N = 1000000
sum( runif(N) < 0.5 )/N
```

```
## [1] 0.500582
```

```r
sum( runif(N) < 0.5 )/N
```

```
## [1] 0.50001
```

```r
sum( runif(N) < 0.5 )/N
```

```
## [1] 0.499384
```

As we increase the sample size, the sample average approaches expected value `0.5`, the population average, because random deviations cancel each other out. This behaviour of larger samples converging to the expected value is called the **"law of large numbers"**.

Simulating normal distributions with `rnorm`

In R there are built-in functions to generate random data from many fundamental statistical distributions, e.g. normal, Poisson, exponential, gamma, F, etc. We will now learn to use the `rnorm` function to create data from normal distributions, also called the Gaussian distribution, after Carl Friedrich Gauss (1777-1855).

Why is the normal distribution so useful? Many biological variables (e.g. human height) are complex, i.e. affected by a multitude of factors with small effect (thousands of alleles, such as variants in growth factor genes (dx.doi.org/10.1038/ng.3097), and past and present environmental effects, such as nutrients and stressors). In each observation (e.g. height of individual X) some of these small factors contribute, some do not.

Variables representing such complex phenotypes tend to be normally distributed. The theoretical normal distribution has ~2/3 of observations within 2 standard deviations from the mean (both ways), and 95% of observations within 4 standard deviations from the mean.

The reason such distributions look normal is related to the **"central limit theorem"**: if sample sizes are large enough, distributions of **sample means** (from whatever original distribution) will follow a normal distribution.

The random combination of small effects on a phenotype is like a random sample - so the sample means (phenotype values) will be normally distributed. For human height, you can imagine each small genetic or environmental effect as a coin toss (does not have to be fair), and the final height as the average of thousands of coin tosses. As it is unlikely to get all heads in thousand coin tosses, it is unlikely to be very short or very tall (assuming no major effects), and most individuals are close to the average.

Measurement errors are also frequently normally distributed. The reason is similar: A multitude of small factors have an effect and in each measurement, a different combination of these influences each measurement.

We will study the central limit theorem later. Let's first learn how to use `rnorm`. The function draws random data from normal distributions. As you should know, normal distributions are defined by 2 parameters, the **mean** and the **standard deviation**. If you supply `rnorm` with only 1 argument, the function returns as many random numbers from the **standard normal distribution** (SND, with mean = 0 and s.d. = 1).

So let's choose 20 numbers from the SND:

```r
rnorm(20)
```

```
##  [1]  0.1960064  2.4462407  1.2203606 -1.5758791  0.1634141  1.9938730
##  [7]  0.7899663 -0.2952133 -0.1442775 -0.9280124 -0.4966993  1.2359051
## [13] -0.5253713  0.6012178 -0.2272293  1.0421560 -0.2643906 -0.4702039
## [19]  1.6389710 -0.4964167
```

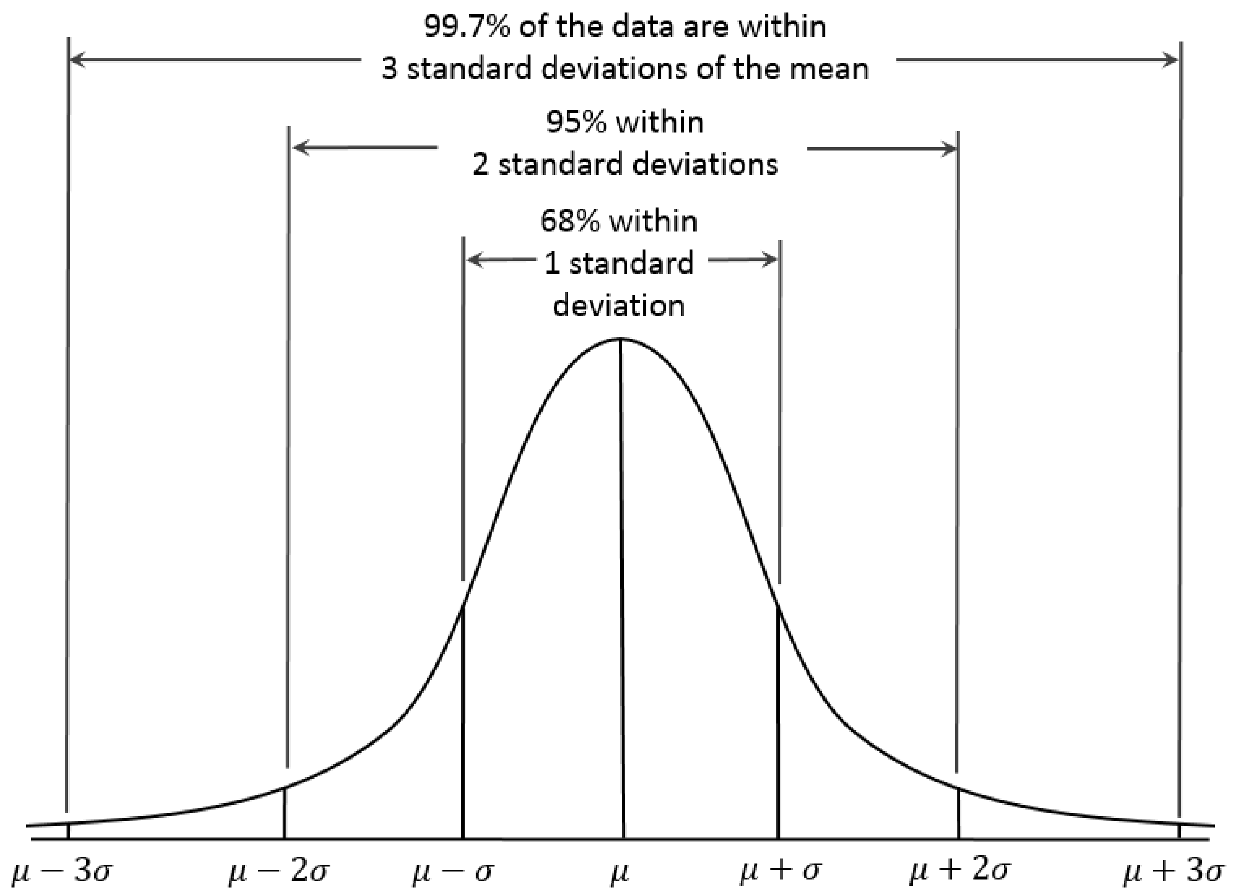


Figure 1: A normal distribution

```r
rnorm(20)
```

```
##  [1] -0.73032477  0.08736836 -0.06085858  1.13799785  1.01622393
##  [6]  1.13151791 -1.06599820  0.69408775 -0.29057987  1.56224305
## [11] -2.10660044 -0.56598083 -1.86690294 -0.35567099 -0.08163022
## [16] -1.73400495  0.25477189  0.34600433 -1.35483447  0.59658465
```

The data should be derived from a distribution with mean of 0 and an s.d. of 1. How can we check that? We can use the built-in functions `mean` and `sd`. Try them on vectors you create using `rnorm`:

```r
x = rnorm(20)
mean(x)  # not super close to the expectation - 0
```

```
## [1] -0.09583255
```

```r
sd(x)
```

```
## [1] 0.9017521
```

```r
# so let's try a larger sample (remember the law of large numbers)
x2 = rnorm(1000000)
mean(x2)
```

```
## [1] 8.266849e-05
```

```r
sd(x2)
```

```
## [1] 0.9999958
```

We can change the parameter values to obtain data from any normal distribution, which we can specify using the functions' arguments `mean` and `sd`.

Exercise:

Check the help function of `rnorm`. Then simulate a distribution of the heights of 1000 humans, with mean = 170 and variance = 400 cm, and store the results in a vector called `heights`. Check if it worked using `mean` and `sd`. Then calculate the proportion of people who are higher than 2 meters.

```r
?rnorm
```

Solution:

```r
set.seed(1)
heights = rnorm(1000, mean = 170, sd = 400^0.5)  # sd = square root of variance
mean(heights)
```

```
## [1] 169.767
```

```r
sd(heights)
```

```
## [1] 20.69832
```

```r
sum(heights > 200)/1000  # proportion of individuals > 2m
```

```
## [1] 0.076
```

Would the proportion increase or decrease if we increased the variance to `800`?

```r
heights = rnorm(1000, mean = 170, sd = 800^0.5)
mean(heights)
```

```
## [1] 169.54
```

```r
sd(heights)
```

```
## [1] 29.41512
```

```r
sum(heights > 200)/1000  # proportion of individuals > 2m
```

```
## [1] 0.153
```

This should be intuitive - as variance increases, the tails grow fatter, and you find "extreme" values become more likely.

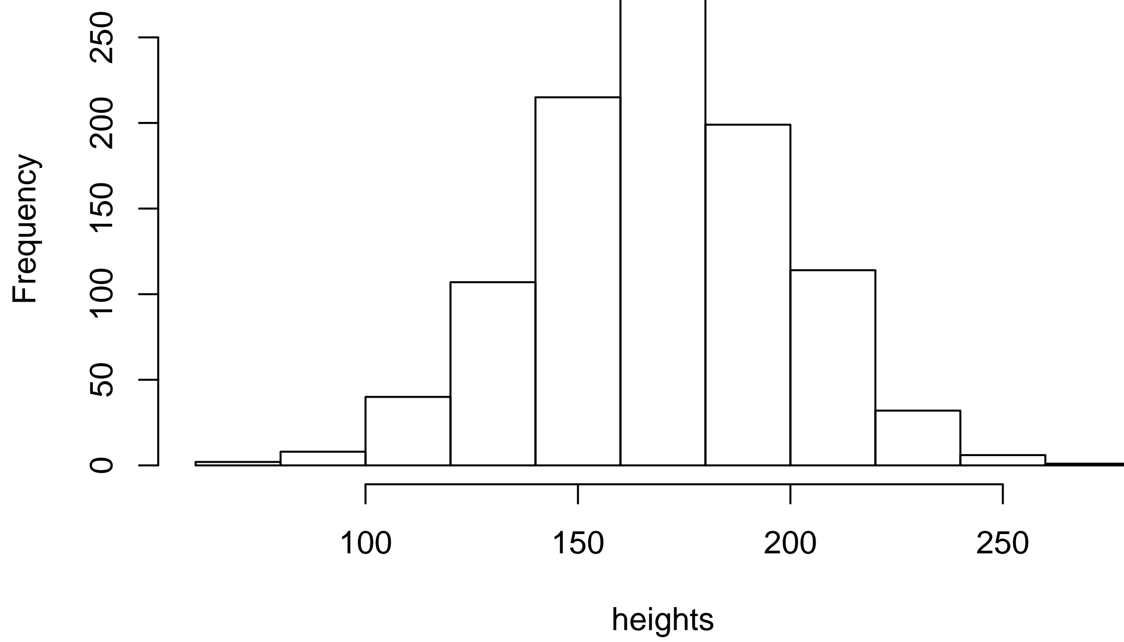
Plotting functions: hist, boxplot, barplot, plot

We will now learn basic plotting functions in R. There are many others, and you can start exploring these here: http://whitlockschluter.zoology.ubc.ca/r-code/rcode02.

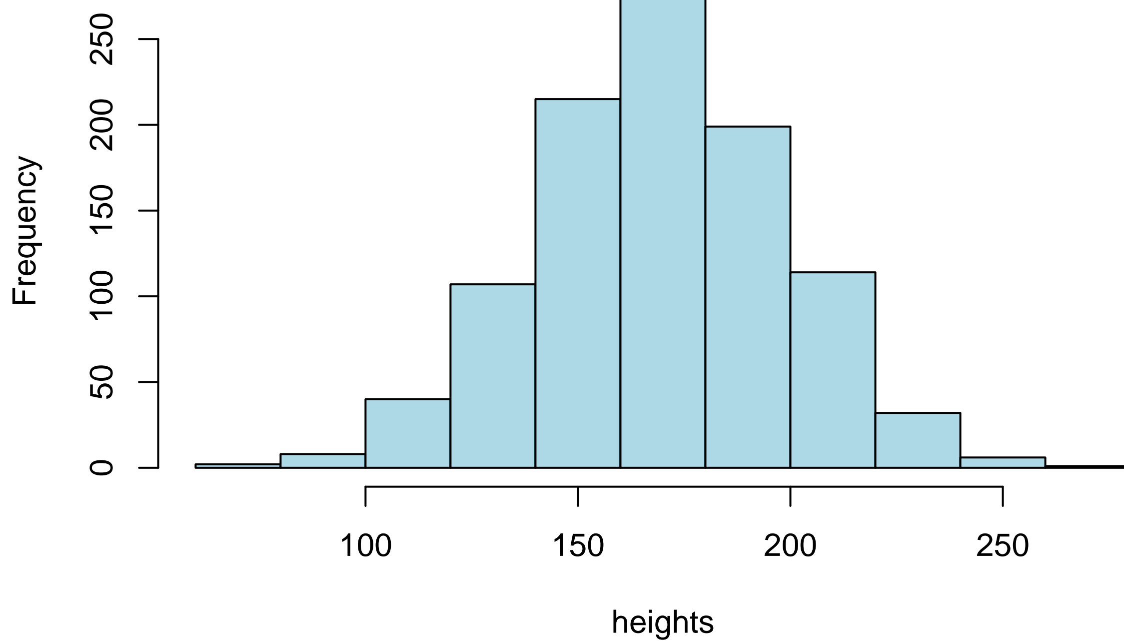
Histogram

The first is a histogram, a distribution of frequencies. To create a histogram from given values, use the `hist` function. We can use the arguments `col` to specify color and `breaks` to specify the number of breakpoints. For other arguments, please check `?hist`.

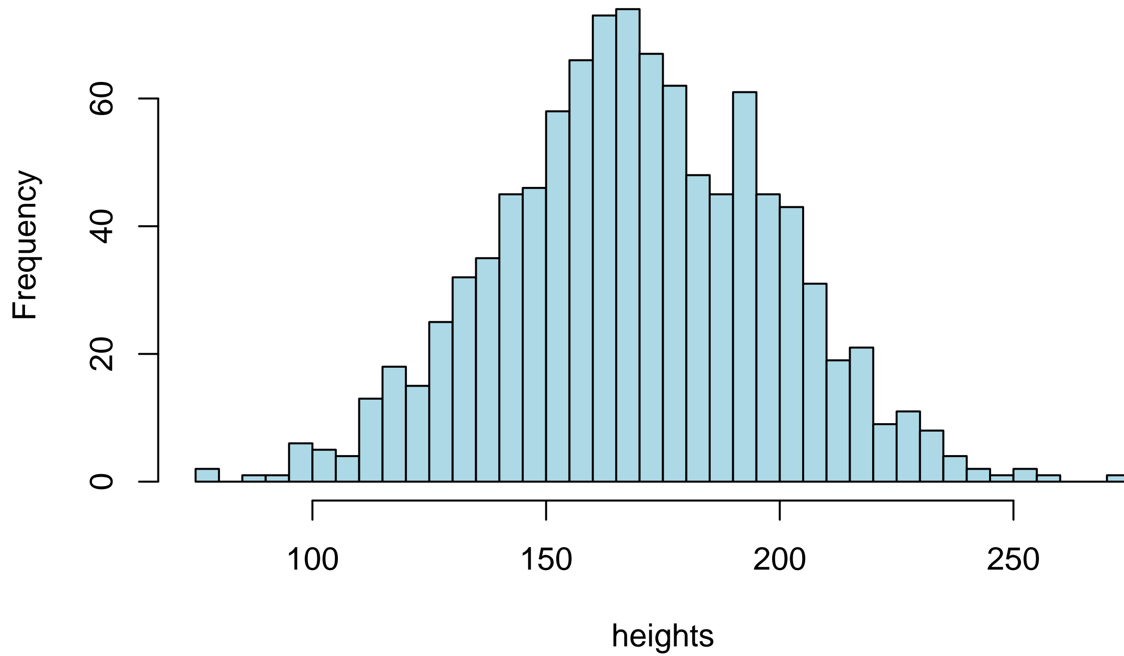
```r
hist(heights)
```



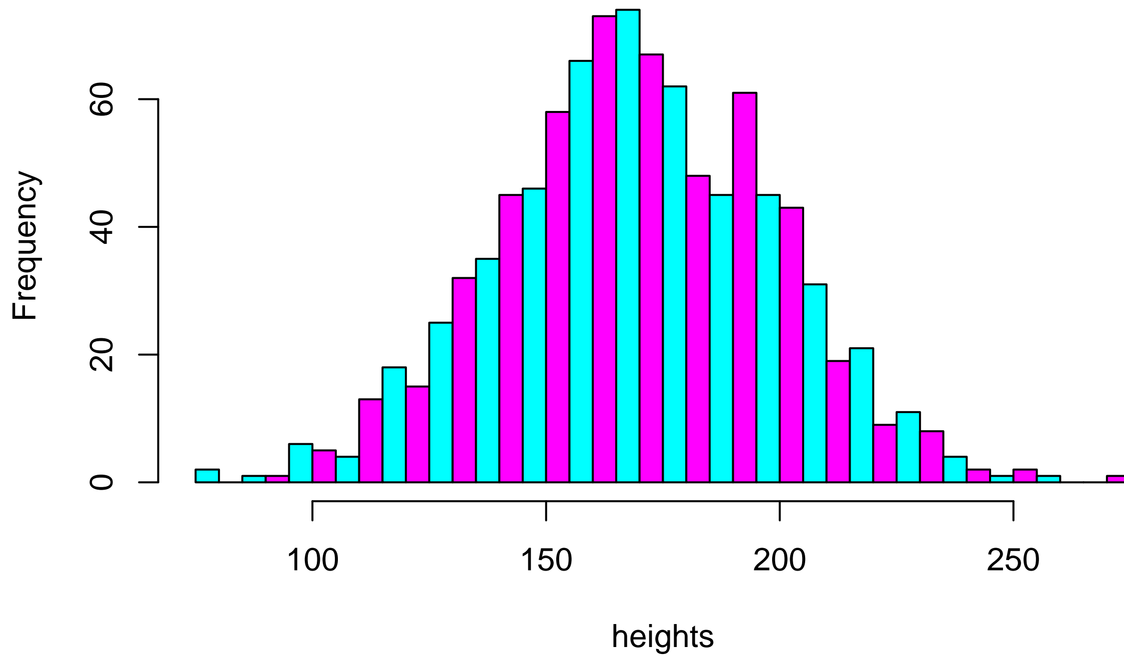

```r
hist(heights, col="light blue")
```



```r
hist(heights, col="light blue", breaks = 40)
```



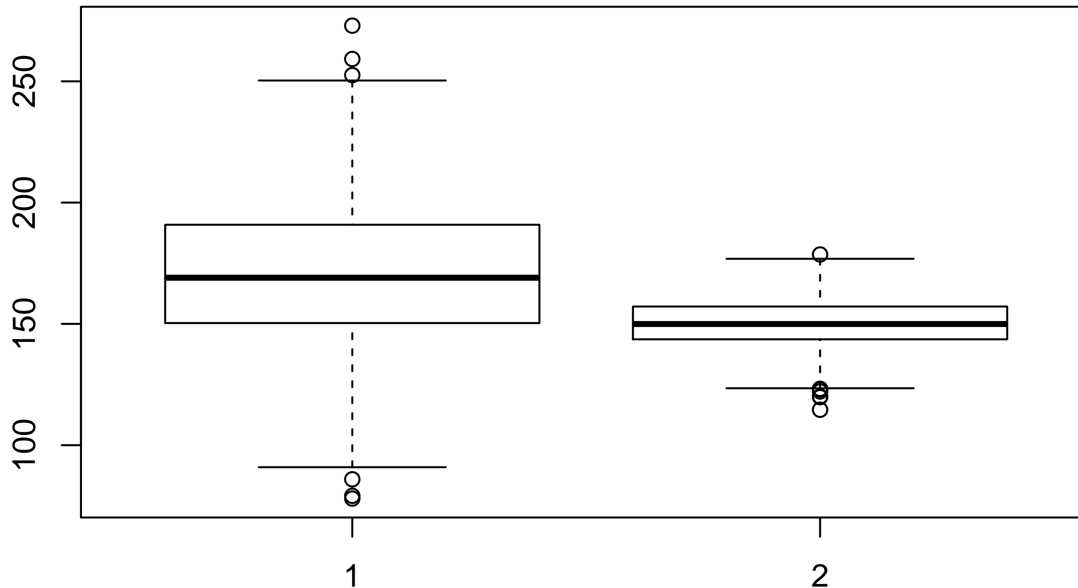

```r
hist(heights, breaks = 40, col = c("cyan", "magenta"))
```



Boxplot

What if we wish to compare two distributions. Say we have data from another population who have mean=150 and sd=10. We can use the function `boxplot` to compare the two:

```
heights2 = rnorm(1000, mean=150, sd=10)
boxplot(heights, heights2)
```

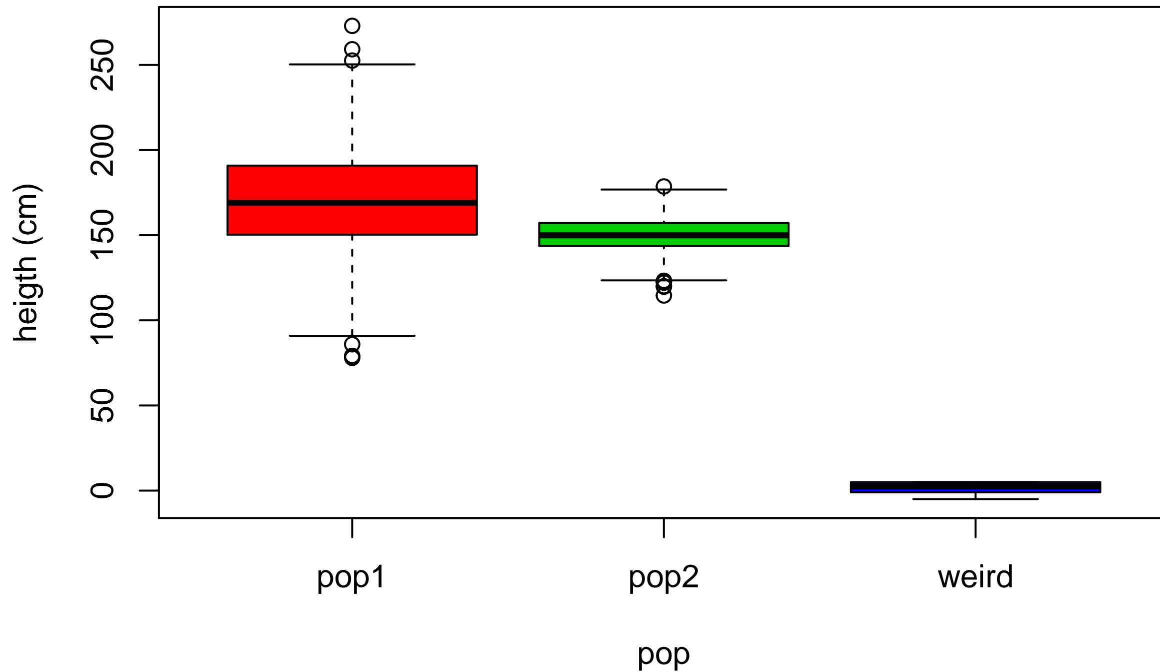


Please check what the boxplot is displaying using `?boxplot`.

The plot displays differences both in location (median) and spread (the interquantile range) between the two data.

Just for fun we can include the `n1` vector as comparison. Below is an example of defining colors, y and x axis labels, the title, and the x axis names.

```
boxplot(heights, heights2, n1,
        names = c("pop1", "pop2", "weird"),
        col = 2:4, ylab = "heigth (cm)",
        xlab="pop", main="my first boxplot")
```



Note that integers correspond to specific colors, defined by the `palette()` object (which you can redefine):

```
palette()
```

```
## [1] "black"   "red"     "green3"  "blue"    "cyan"    "magenta" "yellow"
## [8] "gray"
```

The boxplot can also be called using a formula where the response (dependent) and the explanatory (independent) variables are defined in the format:

responseVariable ~ explanatoryVariable

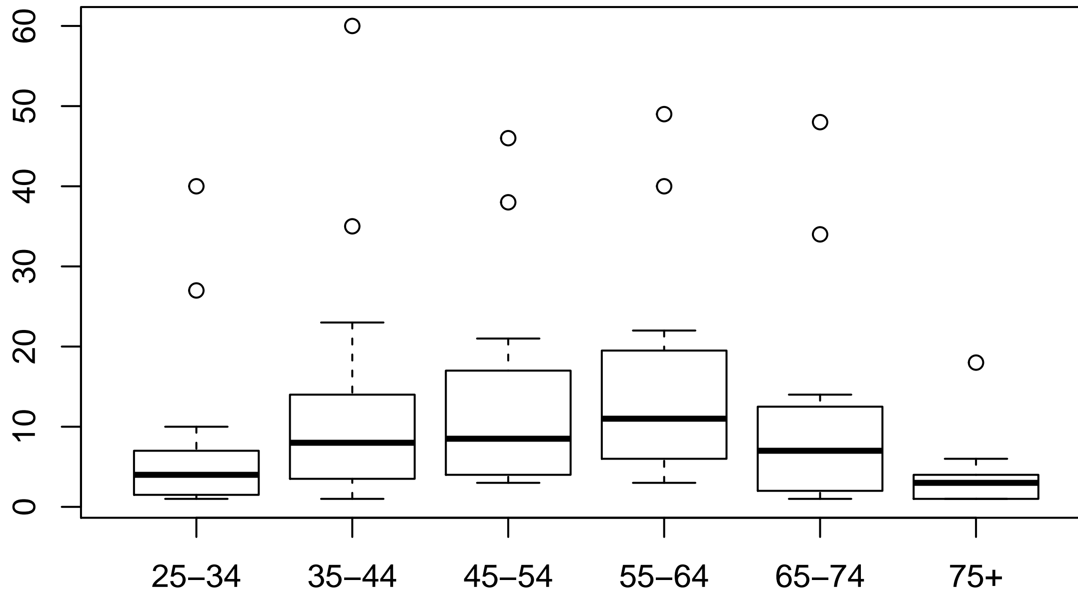
In boxplots we usually have one numeric response variable, while the explanatory variable is a factor (categorical). In this context, the formula indicates that the response variable should be grouped according to the explanatory variable.

When the response and explanatory variables are in `data.frame` format, the formula can be written using the column names of the data frame object. Let's repeat this example with the `esoph` dataset, where we will plot `ncontrols` (number of individuals in the control group) according to each age range, `agegp`:

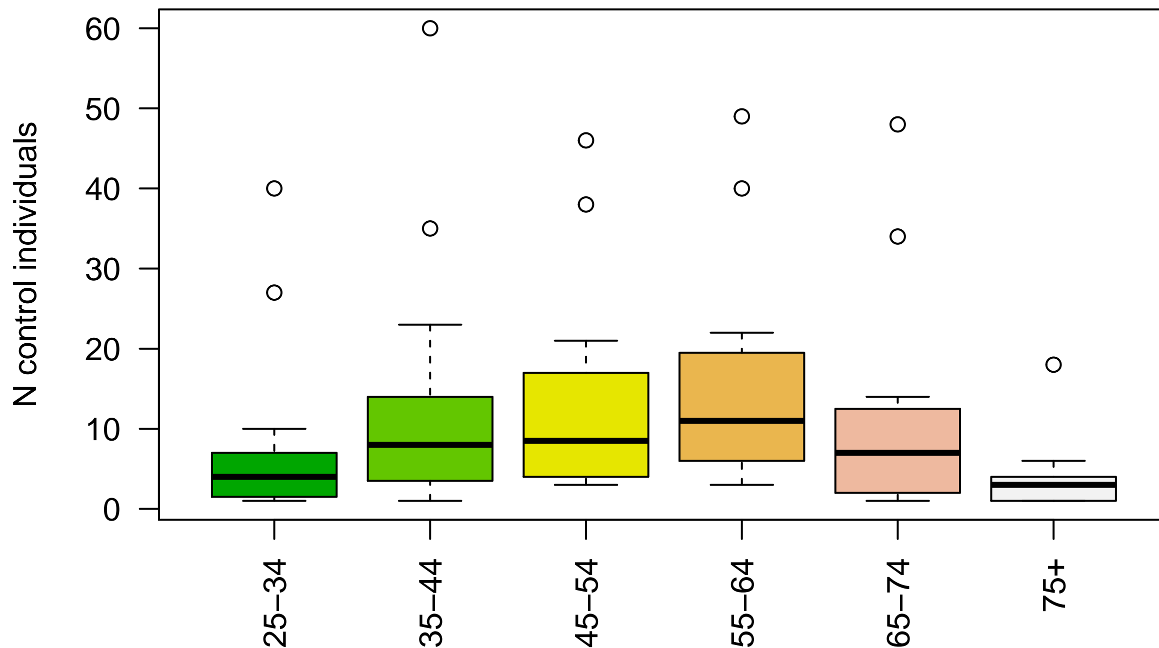
```
head(esoph)
```

```
##   agegp     alcgp    tobgp ncases ncontrols
## 1 25-34 0-39g/day 0-9g/day      0        40
## 2 25-34 0-39g/day    10-19      0        10
## 3 25-34 0-39g/day    20-29      0         6
## 4 25-34 0-39g/day      30+      0         5
## 5 25-34     40-79 0-9g/day      0        27
## 6 25-34     40-79    10-19      0         7
```

```r
boxplot(ncontrols ~ agegp, data = esoph)  # separate the numeric
```




```r
boxplot(ncontrols ~ agegp, data = esoph, las = 2, col=terrain.colors(6), ylab="N control individuals")
```

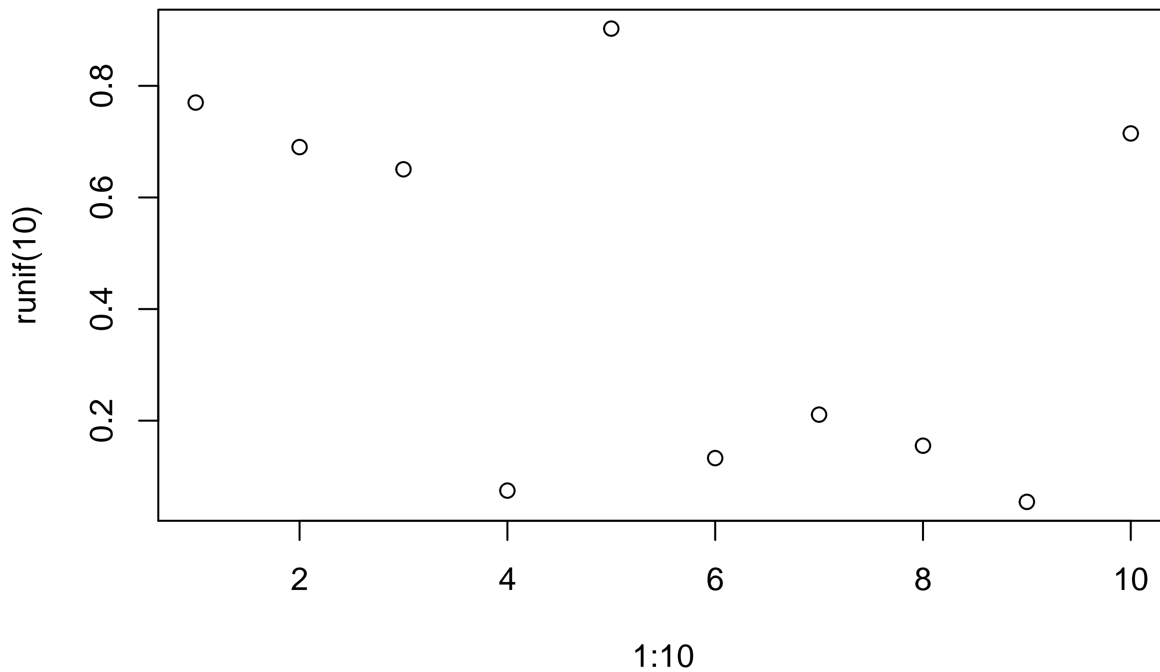


The option `las=2` dictates that the x axis names are written vertically.

Scatter plot

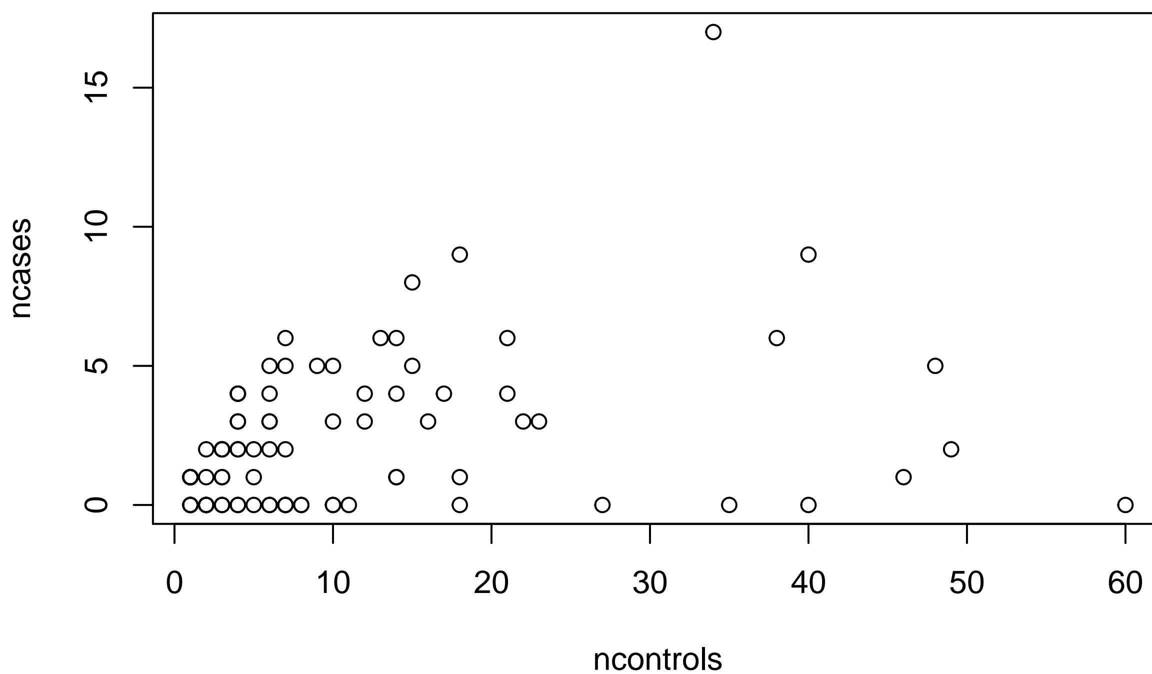
Scatter plots are drawn using the `plot` function, which plot two numeric variables against another. The first input is by default the x values, and the second is the y.

```r
plot(1:10, runif(10))
```

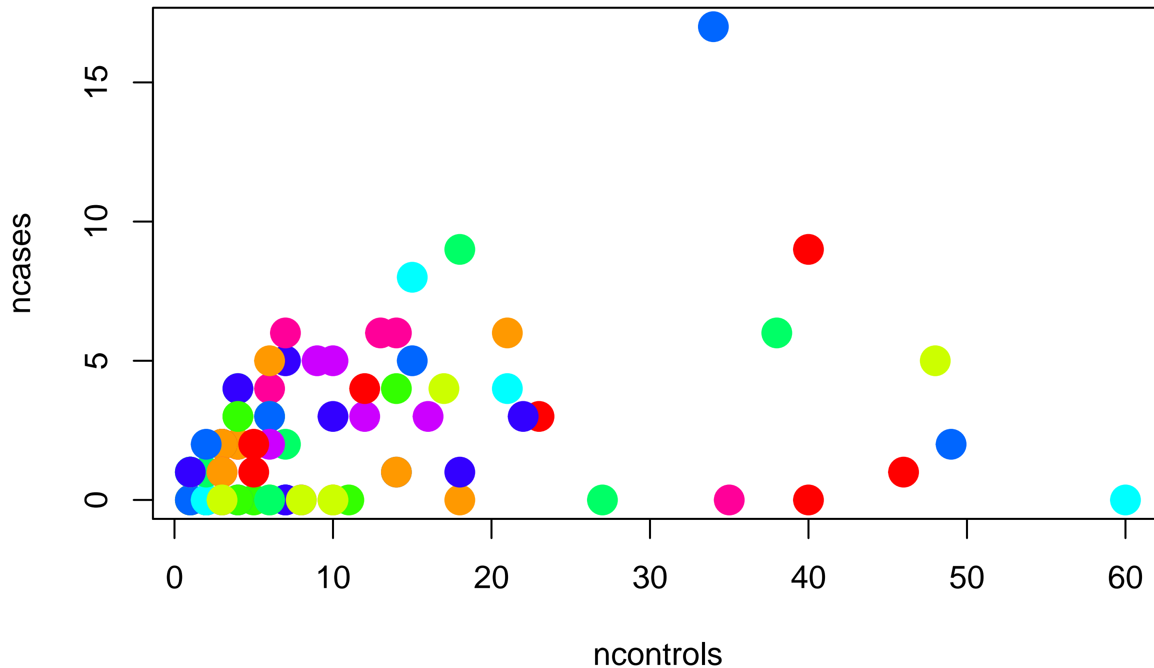


You can also use the formula notation when using data frames. The variable before the tilda sign is automatically considered the "response" and plotted on the y axis.

```r
plot(ncases ~ ncontrols, esoph)
```




```r
plot(ncases ~ ncontrols, esoph,
     pch = 19, # pch=19 means fill the circles
     col=rainbow(10), cex=2)  # cex=2 increases font size
```

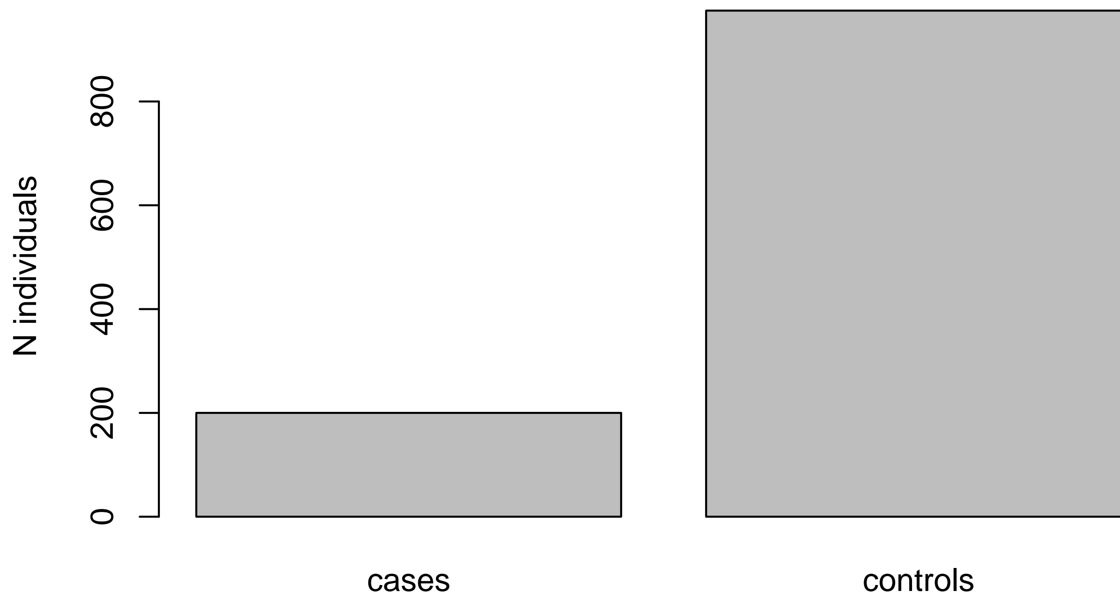


Learn more about plotting parameters here: http://www.statmethods.net/advgraphs/parameters.html

Bar plot

The final type of plot we'll learn is the bar plot, used to display frequencies of categorical variables, using the function `barplot`. The input is either a vector or a matrix. The function simply draws bars with height defined by the numbers. A vector input simply draws bars side by side. A matrix input draws bars that are grouped, either stacked on top of each other, or beside on another:

```
barplot(c(sum(esoph$ncases), sum(esoph$ncontrols)),
        ylab="N individuals", names = c("cases", "controls"))
```



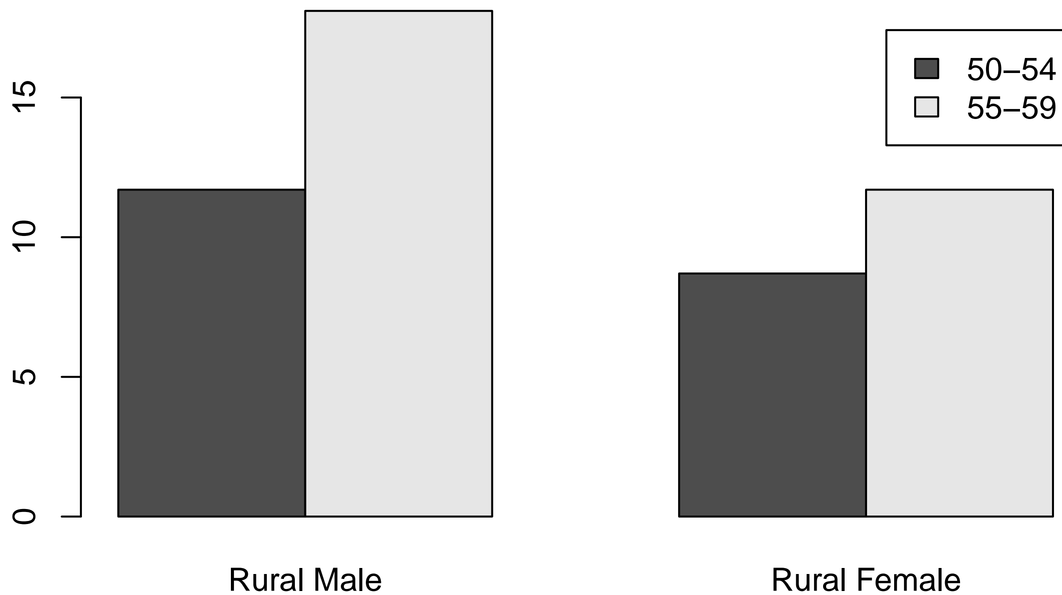
We can also try a matrix. This is a dataset of death rates per 1000 in Virginia:

```r
VADeaths[1:2,1:2]
```

```
##       Rural Male Rural Female
## 50-54       11.7          8.7
## 55-59       18.1         11.7
```

```r
barplot(VADeaths[1:2,1:2])
```

```r
barplot(VADeaths[1:2,1:2], beside = T,
        legend.text = rownames(VADeaths[1:2,]))
```



You can check example barplots using the following:

```r
example(barplot)
```