Universitetet i Bergen Summer 2019
STAT200 - Applied Statistics

Assignment 1 - Introduction to the R

1 General principles

This section provides a brief overview of the programming language R, introductory literature, and the most popular interfaces for using R.

1.1 Brief overview of R

The software R has been specifically designed for working in probability and statistics. The predecessor of R is S language, in contrast to which R is licensed under GPL - GNU. That is, it belongs to the family of so-called free software. R is available for all major platforms, and can be downloaded free of charge on the web site `http://cran.r-project.org/`. Apart from the software itself, you also find free literature on R in the documentation section, where the 'Contributed' part is particularly interesting. Good introductory documents for this lecture are the following:

1. 'A (very) short introduction to R' by P. Torfs & C. Brauer,
2. 'The Friendly Beginners' R Course' by T. Marthews, and
3. 'R for Beginners' by E. Paradis.

Downland these three items, you will need them later on. If you like to install R on your personal computer, simply follow the instruction of Torfs & Brauer in Section 2.1. This document will also be a major part of this first assignment.

In order to start R (e.g., under Windows), just look out for a blue 'R'. When clicking on it, a command window appears, which seems to basically provide those functionalities your would expect of a pocket calculator. Try and type something into the console window, for example a simple sum or product of two numbers.

1.2 Editors for working with R

The R console is not appropriate for working efficiently, as all input is - in principle - lost when closing R. The integrated text editor of R is not very convenient either because of its limited functionality. Fortunately, many editors are available for free, such as

- Emacs,
- Tinn-R,
- RStudio,

- Notepad++,
- ...

The probably most popular and comfortable solution. In Section 2.2 and 2.3 of document by Torfs & Brauer, you find brief explanations on RStudio, how to install it, and what the different windows show. Go through these sections, and create your first R-file as well. You may call it, e.g. 'Assignment 1.R', and save one or two lines of code of your choice. Depending on the editor your are using and your operating system, files ending on '.r' and/or on '.R' are automatically opened by RStudio.

1.3 Working directory and libraries

Working directory and libraries are rather technical subjects, but essential for working successfully with R.

Working directory: The working directory is the folder of your computer which your R session accesses by default for all basic operations, such as reading or writing data, saving figures, etc. The location of the default working directory depends on various options, some of which may be chosen during the installation of R, or pre-determined when starting up R. In order to find out where the working directory can be found on your computer, type `getwd()`, and you will be shown its location.
Section 2.4 of the document by Torfs & Brauer contains more explanations on the working directory and how to change them. Go through this section and change the working directory to different files. If you do not receive an error message, you have changed the location successfully, which you can verify by `getwd()`. Note that restrictions apply for users which are not logged in as computer administrators, hence, restrictions apply.

Libraries: Libraries, also termed with the synonym 'packages' are, roughly speaking, collections of statistical or other procedures, which extend the functionality of your basic R installation. For a beginner, the libraries already included in any R installation are sufficient, but this may change quickly. For details on this subject, read through Section 2.5 of the document by Torfs & Brauer.

1.4 Working directory and libraries

When working with R, you will most likely need help for a command, function, or other R object you get in contact with. This is easy, you only have to type `?name.of.the.command/function()` in the console. Try, e.g., `?getwd`, or another R command you already know.

On the other hand, you may be looking for help on a particular subject, but are not aware of the name of the function(s) which are available in R. In this case, the command

```
help.search("term.to.look.for")
```

may be helpful. Give it a try, e.g. with

```
help.search("linear model")
```

Note, however, that the `help.search()` function is restricted to those features of R which are available on your machine. That is, you will not get any directions to packages that are not yet installed. Section 5 of the introduction by Torfs & Brauer provides additional information on help available in R or elsewhere, go through this section.

1.5 Scripts

This part is usually not very important when starting to work with R, but addresses a technique which helps to organize longer R procedures via their execution by scripts. The idea is simple: if you use particular sequence of R commands regularly, it is not necessary to re-type and execute them each time. Instead, you save these commands in a seperate file, and execute it via the function `source(name.of.script.r)`. Naturally, R will look for the script in the current working directory.
Section 5 of Torfs' & Brauer's document contains more explanations, read this part. Then write a small script with basic commands, such as the sum of two integers. Store it in your working directory, and execute it via `source()`. Alternatively, place the script outside of your working directory, and provide the full address of the path inside the `source()` command.

2 Basic operations

In the following you will become familiar with many basic commands which are essential for working with R. At the beginning, the 'vocabulary' may seem it bit difficult, but R is as any other language: after learning a couple of terms, progress is easy. Through this first assignment, you will become familiar with the construction and operation vectors and matrices. Moreover, we introduce the concept of a list and data frame.
In the following, we will deal with different R ojects, such as vectors, matrices, lists, and data frames. Go through the various examples and exercises to get used to them. In addition to the exercises presented in this first assignment, study the remaining sections of the introduction by Torfs & Brauer as well, and carry out the exercises therein.

2.1 Creating vectors

In R, various ways exist to construct vectors. The most common ones are presented in the following.

1. **Creating numeric vectors**

```
> vec1 = c(2.8, 2.4, 2.1, 3.6, 2.8)
> vec1
[1] 2.8 2.4 2.1 3.6 2.8
```

2. **Creating string/character vectors**

```
> vec2 = c("red", "green", "red", "green", "yellow")
> vec2
[1] "red" "green" "green" "green" "yellow"
```

3. **Creating logical vectors**

```
> vec3 = c(TRUE, TRUE, FALSE, FALSE, FALSE)
> vec3
[1] TRUE TRUE FALSE FALSE FALSE
```

Note that in many documents `TRUE` and `FALSE` are abbreviated by `T` and `F`, respectively. This may be faster, but can lead to errors as well, as these abbreviations have been discontinued some years ago. Therefore, better be safe than sorry, and do not abbreviate.

4. **Creation by repetition**

```
> rep(4, 3)
[1] 4 4 4

> vec4 = rep(vec1, 2)
> vec4
[1] 2.8 2.4 2.1 3.6 2.8 2.8 2.4 2.1 3.6 2.8
```

Note that when the second argument to `rep`, contains an integer-valued vector of same length as the first argument, the following happens: the first element of the first argument is repeated as many times as indicated by the first element of the second argument; similarly the second element of the first argument is repeated as many times as suggests the second element of the second argument; etc.

```
> vec5 = rep (vec1, c (2, 1, 3, 3, 2)
> vec5
[1] 2.8 2.8 2.4 2.1 2.1 2.1 3.6 3.6 3.6 2.8 2.8
```

5. **Creation by building sequences**

```
> vec6 = 1 : 10
> vec6
[1] 1 2 3 4 5 6 7 8 9 10

> vec7 = seq(from = 3, to = 5, by = 0.2)
> vec7
[1] 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6 4.8 5.0
```

We may also construct `vec7` by writing

```
> vec7 = seq(from = 3, length = 11, by = 0.2)
```

6. **Creation component by component**

```
> vec8 = numeric() # or vec8 = c()
> vec8[1] = 41.8
> vec8[2] = -0.3
> vec8[3] = 92
> vec8
[1] 41.8 -0.3 92
```

As you will see later on, this method is primarily used within functions/loops. The same logic applies to applies to character valued (string) vectors. Then, you may alternatively start by declaring

```
> vec9 = character()
```

Or, for a logical vector, by

```
> vec10 = logical()
```

If you know the length of a vector in advance, you may also declare it directly - although this is rarely necessary. For example, the above mentioned `vec8` may also be initialized by

```
> vec8 = numeric(3)
```

7. **Accessing vector entries**
   Single or multiple entries of a vector may be accessed by typing `name.of.vec[index]`. Try, e.g., `vec5[2]` or `vec5[c(1, 3 : 5)]`.

2.2 Creating matrices

For matrices, many operations are similar to those for vectors.

1. **Generic method**
   The most straightforward way to construct a matrix is by the command `matrix`, e.g.

```
> mat1 =  matrix(vec4, ncol = 5)
> mat1
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.8  2.1  2.8  2.4  3.6
[2,]  2.4  3.6  2.8  2.1  2.8
```

   Here, by default, data is entered column by column, starting from the left. This is a direct consequence of the default value of the argument `byrow`, which is `FALSE`. Consult the help for the `matrix` function. We can obtain the same result by

```
> mat1 = matrix(vec4)
> dim(mat1) = c(2, 5)
```

   In case you prefer to fill the matrix elements line by line, the corresponding command is

```
> mat2 = matrix(vec4, ncol = 5, byrow = TRUE)
> mat2
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.8  2.4  2.1  3.6  2.8
[2,]  2.8  2.4  2.1  3.6  2.8
```

   Remember: if an argument, e.g. `byrow`, is not specified, is may take default values (such as `FALSE` or `TRUE`). Only a look into the documentation provides explanation (in most cases...).

2. **Other methods**
   You may also create a matrix with k columns by 'joining' k vectors of identical length.

```
> mat3 = cbind(vec1, 3 : 7)
> mat3
     vec1
[1,]  2.8 3
[2,]  2.4 4
[3,]  2.1 5
[4,]  3.6 6
[5,]  2.8 7
```

The function `rbind` works in a similar way.

3. **Extract a sub-matrix, column, or line**

```
> mat1 [, c(2, 4, 5)]
```

or

```
> mat1[, c(FALSE, TRUE, FALSE, TRUE, TRUE)]
     [,1] [,2] [,3]
[1,]  2.1  2.4  3.6
[2,]  3.6  2.1  2.8
```

retains only the columns 2, 4, and 5.

```
> mat3 [c(1, 4), ]
```

or

```
> mat3 [c(TRUE, FALSE, FALSE, TRUE, FALSE), ]
     vec1
[1,]  2.8 3
[2,]  3.6 6
```

returns a matrix consisting of lines 1 and 4. Attention: if a single column or line is extracted, the result is a vector the components of which are the values found in the respective column or line.

```
> mat1[3]
[1] 2.8 2.8
```

4. **Size of a matrix**
The function `dim()` not only fixes/changes the dimensions of a matrix, but also serves for knowing the size of a matrix.

```
> dim(mat1)
[1] 2 5
```

2.3 Creation of lists

Lists are a relatively special object type. They elements of lists may contain, in principle, R objects of any type, such as matrices or vectors, but also lists again.

1. **Directe method**

```
> list1 = list(vec1, c("red", "blue"), mat1)
> list1
[[1]]
[1] 2.8 2.4 2.1 3.6 2.8

[[2]]
[1] "red" "blue"

[[3]]
     [,1] [,2] [,3] [,4] [,5]
[1,]  2.8  2.1  2.8  2.4  3.6
[2,]  2.4  3.6  2.8  2.1  2.8
```

2. **Component by component** The same result as above can be obtained by

```
> list1 = list()
> list1[[1]] = vec1
> list1[[2]] = c("red", "blue")
> list1[[3]] =  mat1
```

   As before, this technique is mostly used inside functions or loops.

2.4 More vector operations

1. **Copying, extending, extracting, shortening, replacing, naming elements, knowing and changing the type**
   You have already seen above how to access different components (entries) of a vector. Many more operations exist, e.g., simple copying by

```
> vec11 = vec1
> vec13 =  vec12 =  vec11
```

   or extending a vector

```
> c(vec1, c(3.9, 2.7))
[1] 2.8 2.4 2.1 3.6 2.8 3.9 2.7
```

   You may also try

```
> c("white", vec2)
```

What happens, and why? Extracting the elements of a vectore is, as seen, straightforward:

```
> vec1[2]
> vec1[c(2, 4)]
> vec1[2 : 4]
> vec1[vec3]
```

The last technique is very important. Since the two vectors `vec1` and `vec3` have the same length, the components of `vec1` are extracted according to the entries of the logical vector `vec3`, which serves as index in a certain sense.
Shortening vectors is also easy, try

```
> vec11[-c(2)]
> vec12[-c(2,4)]
> vec13[-(2:4)]
```

The same holds true for replacting single elements of the vector, which is done by

```
> vec5
> vec5[3 : 5] = c(1034, 238, -99)
> vec5
```

The different elements of a vector may also be given names, e.g.,

```
> names(vec1) = c("julie", "paul", "solveigh", "valentin", "elsa")
[1] julie paul solveigh valentin elsa
      2.8  2.4      2.1      3.6  2.8
```

Then, you may extract the second element of `vec1` by its name:

```
> vec1["paul"]
```

The type of a vector can be determined by

```
> mode(vec5)
[1] "numeric"
```

or, if a specific type should be checked, by

```
> is.numeric(vec5)
[1] TRUE
```

A vector type may also be changed, e.g. from numeric to character:

```
> vec14 = as.character(vec4)
> vec14
[1] "2.8" "2.4" "2.1" "3.6" "2.8" " 2.8" "2.4" "2.1" "3.6" "2.8"
```

2. **Arithmetic operations on numeric vectors**
   Various arithmetic operations may be carried out on numneric vectors, simple ones such as

```
> vec1 + 4
[1] 6.8 6.4 6.1 7.6 6.8
```

Try:

```
> 2 * vec1 - c(1,2)
```

Attention: When two vectors do not have the same length, the shortest is replicated until it reaches the length of the bigger one. Moreover, when lengths of both vectors are not multiples of each other, a warning is given.

3. **Application of a function**
   Two cases have to be distinguised: on the one hand, a application of a function to a vector may cause an evaluation element by element, and return a vector of same length as the argument as result. On the other hand, a function may take the entire vector as one argument, and return only one value. Besides, other less common cases exist, they are described in the following.

   (a) **Vector of length k $\xrightarrow{f}$ vector of length k**
   Examples for such a function are `abs`, `log`, `log10`, `sqrt`, `exp`, `sin`, `cos`, `tan`, `acos`, `atan`, `asin`, `cosh`, `sinh`, `tanh`, `gamma`, `lgamma`, and many others. Applying of these functions to a vector results in a vector of identical length, the i^{th} component of which is the result of applying the function to the i^{th} component of the initial vector.

   ```
   > sin(vec1)
   ```

   The functions `round`, `trunc`, `floor`, `ceiling` carry out rounding, truncating, and similar operations. Try

   ```
   > round(c(9.238, -1.34222), 2)
   ```

   The functions `sort` and `order` do, well, what their name says: sorting a vector in incrasing or decreasing order,

   ```
   > sort(vec1)
   [1] 2.1 2.4 2.8 2.8 3.6
   > order(vec1)
   [1] 3 2 1 5 4
   ```

   and returning the index of the entries of a vector for rearranging it in decreasing or increasing order.

   (b) **Vector of length k $\xrightarrow{f}$ one numerical value**
   Generate numerical and logical vectors, and test how the results of the functions `length`, `sum`, `cumsum`, `prod`, `cumprod`, `min`, and `max` look like.
   Then, test some statistical functions, for example, `mean`, `var`, and `sd`. Note that the `var` function calculates the unbiased sample variance given by

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2$$

   Reproduce the result obtained by the `var` function by a single line of code.

(c) **Vector of length $k \xrightarrow{f}$ two numerical values**
This is a rather rare case, an example is the function `range()`, which corresponds to `c(min(x), max(x))`.

(d) **Other functions**
There are many other functions, which calculate various quantities and return different output - and might require input of a particular format as well. Simple examples are `summary()`, or `cor()`. Have a look on their help, and test both.

4. **Operations on character vectors**
Treating character vectors is not much different to the numerical case, one may, e.g., paste them together by

```
> paste("jules", "jim")
[1] "jules jim"
```

Specifying the seperator argument (`sep`) allows to remove the white space between the two vectors, or inserting a different seperator.

```
> paste("jules", "jim", sep = "")
[1] "julesjim"
```

The `paste()` function also allows to paste vectors of different length, and more than two vectors:

```
> paste("X", 1 : 4, sep = "")
[1] "X1" "X2" "X3" "X4"

> paste("X", 1 : 4, c("s", "o"), sep = "")
[1] "X1s" "X2o" "X3s" "X4o"
```

5. **Logical operators**
Logical operators allow for different tasks. One may, for example, check for equality with `==`, (strict) inferiority or superiority by `<`, `>`, and `<=`, `>=`, respectively, of inequality with `!=`.
Attention: Consider two vectors a and b given (numeric or character). Then, two cases should be considered.

- **First case: a and b are of identical length**. The result of comparison a and b is a logical vector having the same length as a and b, since each its components is the result of comparing the i^{th} component of of a (`a[i]`) and b (`b[i]`). Example:

  ```
  > c(1, 4, -2, 5) < c(2, 4, -3, 6)
  [1] TRUE FALSE FALSE TRUE
  ```

- **Second case: a et b are of different length**. Without loss of generality, let the length of a be greater than the length of b. When comparing these two vectors, the shorter vector is replicated in a vector b', until it attains the same length as a. Then, the result of the comparison is a logical vector of the same length as a, resulting from the element-wise comparison of a and b', i.e., all `a[i]` and `b[i]`. Example:

```
> a = c(3, 2, 8, 3, -3)
> b = c(1, 2)
> a <= b
[1] FALSE TRUE FALSE FALSE TRUE
```

Effectively, here R carries out the following comparison:
`c(3, 2, 8, 3, -3) <= c(1, 2, 1, 2, 1)`.

The second case also contains a particularly interesting comparison. If the length of the shorter vector is only one, the result is obvious. Example:

```
> a > 2.5
[1] TRUE FALSE TRUE TRUE FALSE
```

By a simple expression, the following expression allows to select only those elements of a vector satisfying certain conditions:

```
> vec1[vec1 > 2.5]
[1] 2.8 3.6 2.8
```

Apart from the above, other operators exist for working specifically with logical vectors: they can be evaluated by means of a logical 'or' (`|`), and 'and' (`&`), a negation (`!`). In addition, the operator `any` returns the value `TRUE` if at least one of the components contains the value `TRUE`, and `all` returns the value `TRUE` if all components of a vector are equal to `TRUE`. Example:

```
> a = c(3, 2, 8, 3, -3)
> b = c(1, 2)
> any(a <= b)
[1] TRUE

> all(a <= b)
[1] FALSE
```

2.5 Operations on lists

List operations follow, in principal, the same logic as those for vectors - although there are much less operations allowed.

1. **Naming list elements**
   There are two different methods for giving names to the elements of a list. If a list has already been constructed, names can be given (or changed) by
   `> names(list1) = c("weight", "colour", "matrix")`
   Alternatively, the names may also be determined when creating the list:
   `> list2 = list(weight = vec1, colour = c("red", "blue"), matrix = mat1)`
   Try

   ```
   > list1

   > list2
   ```

and check if the result is as expected.

2. **Extracting elements of a list**
   This is a rather simple procedure, try

   ```
   > list1$couleur
   ```

   ```
   > list1[[2]]
   ```

2.6 Matrix operations

As before, the most important matrix operations are described.

1. **Naming lines and columns**
   Naming both lines and columns is carried out by

   ```
   > dimnames(mat1) =  list(c("jules", "jim"), paste("X", 1 : 5, sep = ""))
   > mat
         X1  X2  X3  X4  X5
   jules 2.8 2.1 2.8 2.4 3.6
   jim   2.4 3.6 2.8 2.1 2.8

   > mat1["jim", "X3"]
   [1] 2.8
   ```

   Moreover, the functions `colnames()` and `rownames()` allow to extract (and change) the names of columns and rows, respectively, separately.

2. **Arithmetical operations**
   These work in a similar fashion as the respective operations on vectors. Try
   `> mat1 + 3`

3. **Specific operations**
   Moreover, matrix-specific operations exist, e.g., for calculating the product of two matrices

   ```
   A %*% B
   ```

   or transposing a matrix

   ```
   t(A)
   ```

   May other commands exist: `solve()` for inverting a matrix, `diag()` for extracting or changing the diagonal of a square matrix,...

2.7 Missing and absent values

The absence of a value in R is indicated by `NULL`:

```
> dimnames(mat1) = list(NULL, paste("X", 1 : 5, sep = ""))
```

On the other hand, missing values in an object containing data is indicated by the value `NA`:

```
> vec14 =  c(31, 43, NA, 33)
```

For checking the presence of missing values, the command `is.na()` should be used.

```
> is.na(vec14)
[1] FALSE FALSE TRUE FALSE
```

2.8 Structures for storing data: `data.frame` and `ts`

In statistics, data are often presented in matrix like form. Lines represent the individuals observed, whereas columns stand for to the variables studied. For most analyses carried out with R, it is useful to store data in a data structure called data frame.

1. **Construction of a data frame**
   Consider the following example. Suppose you interview Jules, Jim, and Elsa, and note their age, gender, and if they like a certain music style. The resulting data could be stored in a data frame called `survey`, which looks as follows

   ```
   > enquete
         age sex music
   Jules 24  m   TRUE
   Jim   26  m   FALSE
   Elsa  22  f   TRUE
   ```

   How can we construct a data frame? Several possibilities exist:

   - A straightforward way to obtain a data frame is the transformation of a matrix by

     ```
     > dat1 = as.data.frame(mat1)
     ```

     Note that this way is not suitable for our survey, because it contains columns of different nature, numeric and character.
   - Alternatively, it is possible to first create the different variables, and then storing them jointly in a data frame:

     ```
     > age = c(24, 26, 22)
     > sex = c("m", "m", "f")
     > music = c(TRUE, FALSE, TRUE)
     > survey = data.frame(age, sex, music)
     ```

Sometimes, it columns contain variables which should be considered factors, e.g., in the context of an analysis of variance (which you will see later). In this case, one has to declare explicitly that the respective variable should be considered as factor.

```
> sex = factor(c("m", "m", "f"))
```

The different levels of a factor can be shown by

```
> levels(sex)
[1] "f" "m"
```

Note that the levels are ordered alphabetically by default.

Afterwards, one may attribute names to the columns, or change the column names. This works similar to lists:

```
> names(survey)
> names(survey) = c("the.age", "the.sex", "the.music")
```

Note that R assigns the names of the objects of which the data frame is constructed as column names by default. Moreover, line names may be attributed as well by

```
> row.names(enquete) = c("Jules", "Jim", "Elsa")
```

An alternative, quick way for constructing a data frame is by

```
> survey = data.frame(age = c(24, 26, 22),
                      sex = c("m", "m", "f"),
                      music = c(TRUE, FALSE, TRUE))
```

Remark: A data frame can be transformed into a matrix by the function `as.matrix()`. This is in particular recommended for data frames containing only numerical entries, since all arithmetic operations are carried out faster (or at all) on matrices.

2. **Working with data frames**
Storing data in a data frame object is very helpful for a large numer of procedures in R, from graphical representation to regression, analysis of variance, or generalized linear models, etc. Many operations on data frames are similar to matrices. For example, extraction of a 'sub data frame' works by

```
> survey[1 : 2, 2 : 3]
```

Moreover, when working with only one data frame, it is comfortable to use the `attach()` function. This procedure allows to access the columns of a data frame by their names.

```
> attach(survey)
> age
```

Attention: after carrying out the `attach()` command, the objects addressed by the column names are not the columns of the data frame, but local copies. Therefore, changing entries of, e.g., the `age` variable will not affect data stored in the data frame, and vice versa. Moreover, do not forget to `detach` the data frame if you do not need the functionality of `attach` anymore.

```
> detach(survey)
```

Remark: Alternatively, columns may also be addressed by

```
> survey$age
```

3. **Construction of contingency tables from a data frame**
   Inside the data frame `survey`, the variables `sex` and `music` are factors, each having two levels. Contingency tables can be easily obtained by

```
> attach(survey)
> table(sex)
F  H
1  2
```

Or, if both variables should be taken account simultaneously:

```
> table(sex, music)
  F T
F 0 1
H 1 1
```

4. **Construction of time series (`ts`) object**
   Time series are a particular type of data with a chronological order. Often, the data are recorded in discrete, equidistant steps, such as one observation per minute/day/month/etc. Try

```
> x = c(1, 3, 2, 4, 4, 3, 5, 2, 3, 4, 1, 8,
        1, 3, 2, 4, 5, 2, 3, 2, 2, 2, 4, 3)
> ts(x, start = c(2002, 3), freq = 12)
```

and

```
> ts(x, start = c(2002, 3), freq = 4)
```

In both cases, you obtain a time series as result.

2.9 Reading and writing data

When working with larger data sets, these are usually available in form of a file, and entering or copying them manually is not a very pleasant option. Several functions for reading different types of data exist in R, and in the following you learn how to work with the most important ones.

1. **The function `scan()`**
   The scan function is offers basic data reading procedures. Have a look on the data in the text file `data.txt`, and store it in your working directory. You may then read it into R by

```
> vec15 = scan("data.txt")
```

Inspect the result: you obtain a vector with all observations from the file, read line by line. Looking at the original file, however, you realize that the data are not a single vector, but more of form of a matrix:

```
22.4  96.2  88.5
20.1 103.0 100.4
22.2 104.7  97.1
 .     .     .
 .     .     .
 .     .     .
 .     .     .
```

Therefore, you may read the data into a matrix by

```
> dat1 = matrix(scan("data.txt"), ncol = 3, byrow = TRUE)
```

Then, you may also give names to the lines and/or columns, and start your analysis.
Remark: pay attention to the type of data you read. If a single column contains only one non-numerical entry, all entries of the entire matrix are considered characters. Moreover, recall that missing values must be of the form `NA` in order to be recognized by R as such.

2. **The `read.table()` function**
   A more powerful alternative to `scan` is the function `read.table()`. If data are read by this function, the resulting object is a data frame. Therefore, numerical columns remain numerical, even if other coloumns with character entries are present in the data set.
   Have a look into the help, then read the data in the file 'data.txt' by means of `read.table()`. A couple of aspects are noteworthy. First, one possible argument is `header`, and allows to determine if the first line contains already data, or just the names of the respective column. Secondly, related functions exist, such as `read.csv()` and `read.csv2()`, which allow to read comma seperated value files of American and European format, respectively. Last but not least, observe the large number of options allowing for reading data of more complex format as well. For example, seperators of columns and decimals may be specified for uncommon data formats.

3. **Writing data from R into the working directory**
   The functions for writing data are closely related to those for reading data. Have a look on the help for the functions `write()`, which exports matrices in general, or `write.table()` and `write.csv()` for exporting data frames. Later on, you may also need to work with Excel files. For this tasks, specific packages exists, e.g. the `xlsx` package, which contains a collection of helpful functions.
   Moreover, R objects can be saved (and restored) by the functions `dput()` (and `dput()`). Note, however, that they are saved uncompressed as plain text file. Therefore, this is not a recommendable solution for large files.

2.10 Some more complex functions

The number of functions available in R is vast, and it is nearly impossible to know all of them. Nevertheless, a couple of helpful functions for facilitating the work with data exist, and some of them are described in the following.

1. **Application of a function to lines (coloums) of a matrix:** `apply()`

```
> apply(mat1, 1, mean)
[1] 2.74 2.74
```

The result is a vector of length equal to the number of rows of tt mat1, and whose entries are arithmetic mean of each line of the matrix. Alternatively,

```
> apply(mat1, 2, mean)
[1] 2.60 2.85 2.80 2.25 3.20
```

provides as result a vector of length equal to number of columns of the matrix `mat1`. The entries of the vector correspond to the mean of each column.

2. **Application of a function to the elements of a list:** `sapply()` **et** `lapply()`
The functions `sapply()` et `lapply()` work in a similar way as the `apply()` function, but are designed for operating on lists. Try

```
> my.list = list(c(1, 2), c(3, 1, 2))
> sapply(my.list, mean)
[1] 1.5 2.0
```

The result is a vector of length equal to the number of elements of the list. The value of each component of the vector is the average of list element. Alternatively,

```
> lapply(my.list, sd)
[[1]]
[1] 0.7071068

[[2]]
[1] 1
```

returns a list as result instead of a vector. In this example, the resulting liste has teh same length (number of elements) as the list to which the function `lapply()` was applied. The value of each list element is the standard deviation, calculated by `sd`, of each element of the original list.

3. **Application of a function to a grouped vector:** `tapply()`
The `tapply()` function is helpful to carry out a procedure on a variable which is subject to a grouping, e.g., calculation of the average age of a sample divided into men and women:

```
> tapply(age, sex, mean)
 f  m
22 25
```

The result is the average of the variable `age`, evaluated for the individuals belonging to the levels `f` and `m`, respectively, of the `sex` variable.
Note that it is also possible to treat groupings with respect to the levels of several variables. In our case this does not make a lot of sense, but is possible:

```
> tapply(age, list(sex, music), mean)
  FALSE TRUE
f    NA   22
m    26   24
```

returns the average age for each combination the different levels of the variables age and sex. In the example above, the `mean()` function was used. Naturally, it can be replaced by any R function which can be applied to a vector and returns a single value.

4. **Function for constructing lists:** `split`
The function tt split builds a list from two arguments, the first being a vector and the second being a vector of factor type (or a numeric/character vector, which can be converted to a factor). The number of components of the resulting list corresponds to the number of levels of the factor. Each component of the list contains a vector, containing all these values of the first vector for which the factor takes the value attributed to the respective list element.

```
> split(age, sex)
$f
[1] 22

$m
[1] 24 26
```