

# Introduction to R

This is *not* a systematic or comprehensive introduction to R, it's just a quick look how R works. You are going to learn R more systematically during this course, especially in the exercises.

The following books (both as pdf-files on ILIAS, folder Software\_R -> Books) offer a comprehensive more introduction to R.

- Dalgaard Peter: Introductory Statistics with R (Springer)

Peter Dalgaard is a developer of R. It's a systematic introduction to R. Recommended.

- Jim Albert, Maria Rizzo: R by Example (Springer)

As the title says, the content of this book consists entirely of examples. Recommended, although some examples are quite advanced.

- Andre de Vries, Joris Meys: R for Dummies

If you like the Dummies series, this book is for you.

## 1 What is R?

The programming language R was *specifically* developed to solve statistical problems. For information about the history of R, see

[https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/R_(programming_language))

R has the advantage that it is very widespread in the statistics community. In a way R is the state-of-the-art software for solving statistical problems. Furthermore, the software is open source and therefore available for free.

For beginners, the handling of R needs somewhat to get used to, but once that happened you'll find that R is a very powerful software.

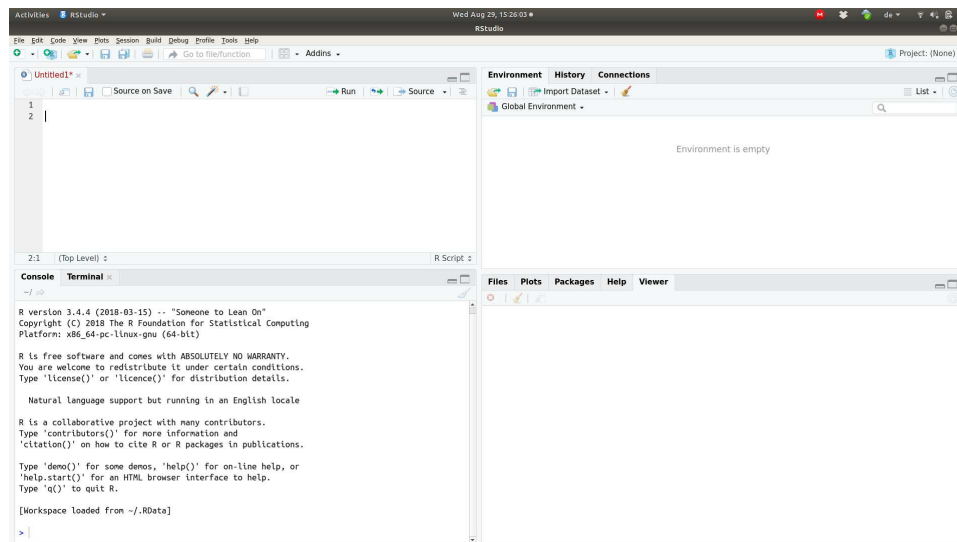
## 2 Installation

The installation instructions are on ILIAS (folder Software\_R -> Installing R and RStudio).

### 3 R with RStudio

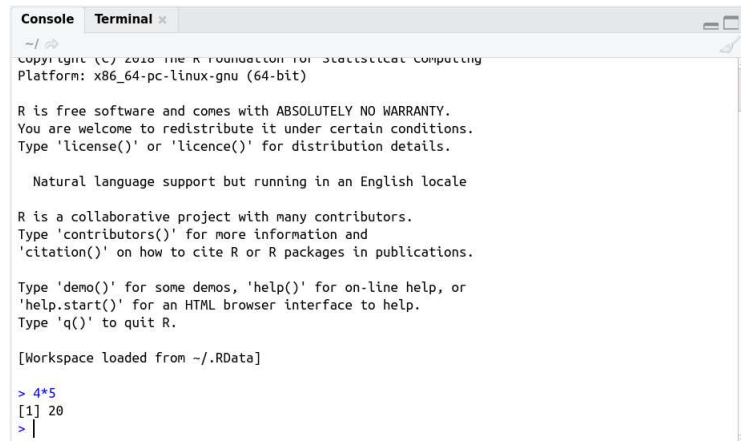
R itself has only limited possibilities for entering and handling commands via a terminal. To get around this limitation, we recommend a user interface. We'll use the interface **RStudio** which is the most widespread and popular.

If you open **RStudio**, for four panels appear.



1. Upper left panel: This is our working panel, i.e. for writing commands.
2. Lower left panel: This is the heart of R. Here the commands are executed and the results displayed (except graphics).
3. Lower right panel: Used among others for displaying graphics.
4. Upper right panel: This panel lists all defined objects.

Commands can be written and executed directly in the lower left panel. For example, we type `4*5` after the `>`-prompt, press **Return** and the result is shown.



```
~/
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

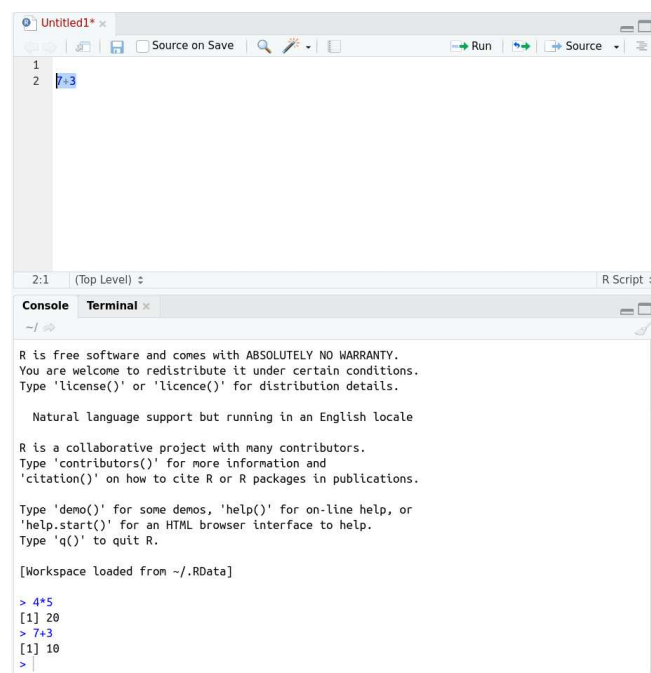
[Workspace loaded from ~/.RData]

> 4*5
[1] 20
> |
```

However, it is better to use the upper left panel to write the commands, because the content in this panel can be saved and be edited later on.

To execute commands in this panel, highlight the commands and press **ctrl+enter** (pressing **enter** alone just produces a linebreak in the editor).

The result is again displayed in the lower left panel.



```
Untitled1*
Source on Save
Run
Source

1
2 7+3

2:1 (Top Level) R Script

~/
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[Workspace loaded from ~/.RData]

> 4*5
[1] 20
> 7+3
[1] 10
> |
```

## 4 First steps

### 4.1 R as calculator

As we have just seen, we *can* use R as a calculator:

```
1+1  
  
## [1] 2
```

or

```
2*4  
  
## [1] 8  
  
exp(3)  
  
## [1] 20.08554
```

You can add comments to the source code, which is highly recommended. If you start a line with a hashtag #, then the text in this line is ignored during execution of the commands.

```
# A first example  
1+1  
  
## [1] 2
```

### 4.2 Assignments

Now it gets a little more interesting. We can assign values to variables. This is done with the assignment operator <=.

```
a <- 3  
bb <- 4  
  
a
```

```
## [1] 3

bb

## [1] 4
```

The value 3 is assigned to the variable `a` and 4 is assigned to the variable `bb`.  
Now we can use these variables for further calculations.

```
2.7*a

## [1] 8.1

# The ^-Operator is used for the calculation of powers
a^2

## [1] 9

a/bb

## [1] 0.75
```

### Remarks:

1. As we have seen, variable names can consist of several characters. They *cannot* start with a number.
2. `R` is case-sensitive, that means that the variable `a` and the variable `A` are *different*.

```
a <- 7

a

## [1] 7

A

## Error in eval(expr, envir, enclos): object 'A' not found
```

Variable `A` is not recognized by `R` because this variable hasn't been defined before.

3. We can also use the equality operator `=` for assignments.

```
a = 3
b = 4

a
## [1] 3

b
## [1] 4
```

However, this is for several reasons not recommended and we will use only the assignment operator `<-`.

4. Like most programming languages, R doesn't understand the expression `3a`.

```
3a
## Error: <text>:1:2: unexpected symbol
## 1: 3a
##      ^
```

For multiplications, we *have to* use the multiplication operator `*` (`3*a`).

```
3*a
## [1] 9
```

## 4.3 Vectors

### 4.3.1 Generate vectors

Vectors are very important in R. A vector<sup>1</sup> is a so-called *onedimensional array*. In more familiar terms, a vector is simply a list<sup>2</sup> of objects. Vectors are generated with the command `c(...)` (concatenate).

```
z_1 <- c(3, 4, 4.5, -2, 7)

z_1
```

---

<sup>1</sup>The expression "vector" is in R more general than the one used in mathematics, where the components consist generally just of numbers. See the vector `mixed` in the example above.

<sup>2</sup>We have to be careful here as the expression "list" is also used in R and has a different, more general meaning. But in the beginning, you can think of vectors as lists.

```
## [1] 3.0 4.0 4.5 -2.0 7.0

z_2 <- c(1, 2, 3, 4, 5)

z_2

## [1] 1 2 3 4 5

mixed <- c(2, 'sad', 5.3, 'YES', "beautiful", "sad")

mixed

## [1] "2" "sad" "5.3" "YES" "beautiful" "sad"
```

### Remarks:

1. Each entry in a vector is called a *component* of this vector. In this document, we'll use component and entry interchangeably.
2. As we have seen with the vector `mixed`, vectors can be very general. The entries doesn't have to be numbers and the same entries can occur several times and
3. There is no difference between `'...'` and `"..."`.
4. The order of the components in a vector is important. In the following example the vectors `a` and `b` are *not* the same because the entries 1 and 2 are in a different order.

```
a <- c(1, 2)
b <- c(2, 1)

# The command "identical" checks whether two object are equal
# (result "true") or not (result "false")

identical(a, b)

## [1] FALSE
```

5. Vectors like `z_2`, where the components are successive numbers occur quite often in R. There is a shortcut command for this kind of vectors.

```
z_2 <- 1:5

z_2
```

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

a <- -3:4

a

## [1] -3 -2 -1 0 1 2 3 4
```

6. An important note. If you look at examples on the web, you'll often see the variable names like `z.1`, i.e. dots are used within variable names. This is perfectly acceptable. It *is* a convention in `R` to use dots within variable names.

However, using dots within variable names in Python are *not* acceptable because the dot has a special use in Python.

To avoid any confusion with Python codes, it's better to use the subscript dash `_` within variable names in `R`.

### 4.3.2 Operations with vectors

#### Simple operations

The components of the vectors `z_1` and `z_2` are numbers (which is often, but certainly not always the case in `R`) and we can perform the usual arithmetical operations with these vectors.

```
z_1 + z_2

## [1] 4.0 6.0 7.5 2.0 12.0

4*z_1

## [1] 12 16 18 -8 28

z_1*z_2

## [1] 3.0 8.0 13.5 -8.0 35.0
```

The result of the last multiplication of these two vectors is a vector<sup>3</sup>. The first component of the resulting vector is the product of the first components of the vectors, the second component is the product of the second components and so on.

---

<sup>3</sup>I.e. this product is not to be mistaken with the scalar product.



Of course, a multiplication with the vector `mixed` doesn't make any sense and R produces an error message:

```
3*mixed

## Error in 3 * mixed: non-numeric argument to binary operator
```

### First statistical operations

As an example for a different operation on vectors, we wish to calculate the average (or mean) of the values in the vector `z_1`. This is done with the R-command `mean(...)`.

```
mean(z_1)

## [1] 3.3
```

Or we can calculate the sum of the values of the vector `z_1` (or the minimum or the maximum).

```
sum(z_1)

## [1] 16.5

min(z_1)

## [1] -2

max(z_1)

## [1] 7
```

Of course, we could have done this by just taking a look at the vectors as there are just 5 components. It's a completely different story, if a vector has 100 000 entries.

The command `length(...)` computes the number of components in a vector.

```
length(mixed)

## [1] 6
```

The vector `mixed` has 6 entries.

It is often useful to access a specific component of a vector. This is done with square brackets after the variable with a number for the corresponding component within the brackets.

```
mixed[2]

## [1] "sad"
```

The second entry of the vector `mixed` is `sad`.

Now, we want to know what the first and the third entries are. How can this be done? The obvious, but wrong way, is as follows:

```
mixed[1, 3]

## Error in mixed[1, 3]: incorrect number of dimensions
```

Because a vector is a onedimensional object, the components are accessed just by *one* number in the square brackets. If we want to access several components at the same time, we exchange the *one* number in the square brackets with *one* vector, in this case `c(1, 3)`:

```
mixed[c(1, 3)]

## [1] "2" "5.3"
```

If we want to access components 2, 3, 4, 5, we can use, as we have seen, a shortcut command: `2:5`.

```
mixed[2:5]

## [1] "sad" "5.3" "YES" "beautiful"
```

### 4.3.3 R -commands with options

To illustrate a command with an option in `R`, we create the following vector:

```
z_3 <- c(5, 2, NA, 4)
```

The “value” `NA` (not available) occurs often in data science. These are “values”, which for some reason are unknown. For example, in a survey some persons didn’t want to give their age (these are the `NA`’s).

If we want to determine the minimum of the vector `z_3`, `R` produces the following result:

```
min(z_3)
```

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

`R` tries to find the minimum value in this vector, but it doesn’t know what to do with the `NA`, so it returns `NA`.

But we can still find the minimum value of the *existing* numbers in this vector. This is done with a so-called *option* which we put additionally within the brackets of the function call.

```
min(z_3, na.rm = TRUE)
```

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

In this case the option is `na.rm = ...` and means *na remove*. If we set this option `TRUE`, then all `NA`’s are removed for the calculation of the minimum, if `FALSE` then the original vector is used, including the `NA`.

All options have a default setting. In our example, the default setting is `na.rm = FALSE` (or simply `F`). That means, if we don’t add the option to the command, `R` sets `na.rm = FALSE`.

```
min(z_3)
```

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

```
min(z_3, na.rm = FALSE)
```

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

Most of the time, commands have several options. We can check which options a command has, including the default setting, the following way (for our example the minimum)

```
?min
```

## 4.4 Data sets (twodimensional)

A lot of data are given in the form of twodimensional data sets (tables). The following table consist of fictional temperatures in several Swiss cities during several month.

	Luzern	Basel	Chur	Zuerich
Jan	2	5	-3	4
Feb	5	6	1	0
Mar	10	11	13	8
Apr	16	12	14	17
Mai	21	23	21	20
Jun	25	21	23	27

Such data sets are usually saved in files. In this case the file is called **weather.csv** (on ILIAS), which you can download and save in a suitable folder<sup>4</sup>.

A huge source for data sets is:

<https://www.kaggle.com>

### 4.4.1 Loading (importing) data sets

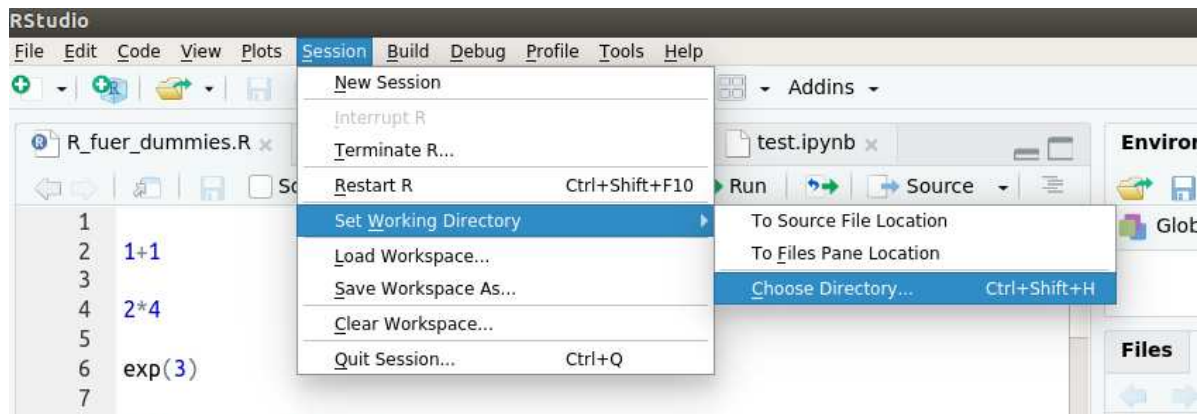
To load data sets, it's recommended that you set your working directory first. This can be done within **RStudio** or manually. We highly recommend that you do this with **RStudio**, especially for Windows user.

#### **RStudio**

Go to (see figure below) **Session, Set Working Directory, Choose directory**, then click your way through to folder you want to work with, then **Open**. **RStudio** will set the working directory automatically.

---

<sup>4</sup>The folder path should't contain empty spaces or umlauts ä, ö or ü. **R** handles these badly.



## Manually

With the `setwd(...)`-command (“set working directory”), we specify the working directory in which the file `weather.csv` has been saved. This is in *my* case `/home/euler/Dropbox/Statistics/Software_R_Python/R`.

```
setwd("/home/euler/Dropbox/Statistics/Software_R_Python/R")
```

## Important for Windows user

Windows uses backslashes `\` instead of slashes `/` in their pathes. So your path looks something like

```
C:\Users\Documents\folder\R
```

This path doesn’t work in `R`. To get around this, you have to double the backslashes:

```
setwd("C:\\Users\\Documents\\folder\\R")
```

## Loading data sets

Once we have set the working directory, we can load (import) the data set with the command `read.csv(...)`.

```
temp <- read.csv("weather.csv")
```

```
temp
```

```
##      Luzern Basel Chur Zurich
## Jan      2      5    -3      4
## Feb      5      6      1      0
## Mar     10     11     13      8
## Apr     16     12     14     17
## May     21     23     21     20
## Jun     25     21     23     27
```

We can also add the file path directly to the file name:

```
temp <- read.csv("/home/euler/Dropbox/Statistics/Software_R_Python/R/weather.csv")

temp
```

In both cases the content of the table was assigned to the variable `temp` (for temperature). The assignment of data sets to variables is very useful as we will see shortly.

### Remarks:

1. Once the table is imported, the table is then referred in R-terminology as *data frame*. That's the reason, why data frames are sometimes assigned to the variable `df`.

```
df <- ...
```

You see this quite often in blogs or newsgroups. It's important keep in mind that `df` is a variable and *not* a command.

2. There exist several commands for loading files. By now, a lot of data sets are available in the .csv-format, so we can use the command `read.csv(...)`.
3. There is a more general command `read.table(...)`, which can load other file formats as well. Let's try this out for our file `weather.csv`:

```
temp <- read.table("weather.csv")

temp

##      V1      V2
## 1 Luzern , "Basel", "Chur", "Zurich"
## 2 Jan      , 2, 5, -3, 4
## 3 Feb      , 5, 6, 1, 0
## 4 Mar      , 10, 11, 13, 8
## 5 Apr      , 16, 12, 14, 17
## 6 May      , 21, 23, 21, 20
## 7 Jun      , 25, 21, 23, 27
```

This doesn't look right, something must have gone wrong. The table doesn't look "nice". There are two reasons for this behaviour:

- a) The column names are `V1` and `V2`, which is not correct. These should be `Luzern`, `Basel`, `Chur` and `Zurich`.

We have to force `read.table(...)` to accept the first line as column names. This is done with the option `header = T`

```
temp <- read.table("weather.csv", header = T)
```

```
temp
```

```
##   Luzern X..Basel...Chur...Zurich.
## 1   Jan                      ,2,5,-3,4
## 2   Feb                      ,5,6,1,0
## 3   Mar                      ,10,11,13,8
## 4   Apr                      ,16,12,14,17
## 5   May                      ,21,23,21,20
## 6   Jun                      ,25,21,23,27
```

- b) The table still doesn't look right. The entries appear to be separated by commas, but `read.table(...)` expects by default that the entries in the table are separated by empty spaces.

However, it is possible to load files of data sets which are separated by commas. This is done with the option `sep = ","`.

```
temp <- read.table("weather.csv", sep = ",")
```

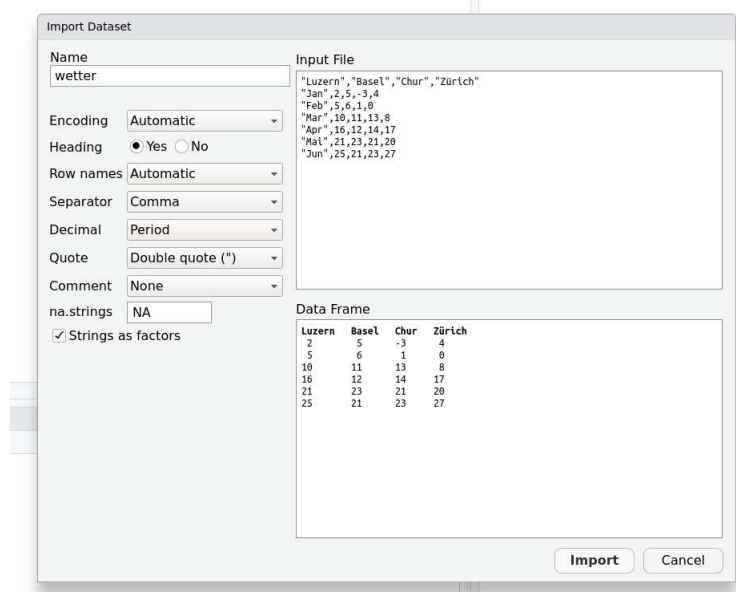
```
temp
```

```
##      Luzern Basel Chur Zurich
## Jan       2     5   -3      4
## Feb       5     6    1      0
## Mar      10    11   13      8
## Apr      16    12   14     17
## May      21    23   21     20
## Jun      25    21   23     27
```

4. There is another possibility to import (load) files, which doesn't work properly for non-.csv-files. In `RStudio` click on in the upper right panel `Import Dataset`, then click on the first entry (might look different for different operating systems).



Then click through to your folder and click on the desired file. The following window appears.



On the upper right side, the original table appears (Input File). The table on lower right side (Data Frame) looks as it should be. If that's not the case, you have to change the parameters on the left hand side (change for example [Separator](#) and see what happens).

#### 4.4.2 Accessing columns and rows

An important remark at the beginning: the first row and first column are *not* part of a data frame. These are used to access of the individual data, whole (or parts of) rows and columns.



A very useful command is the `head(...)`-command. The output are by default the first 6 rows (in our example a bit useless because the table has only 6 rows). With this command we can check whether the file has been correctly imported.

```
head(temp)

##      Luzern Basel Chur Zurich
## Jan      2    5   -3      4
## Feb      5    6    1      0
## Mar     10   11   13      8
## Apr     16   12   14     17
## May     21   23   21     20
## Jun     25   21   23     27
```

The size of a data set is often unknown. If we want to know the number of rows and columns in a data set, we use the command `dim(...)`

```
dim(temp)

## [1] 6 4
```

*Important:* The first value is the number of rows, the second value the number of columns. In the example above, we have 6 rows and 4 columns.

We obtain the column names with the command `colnames(...)` and the row names `rownames(...)`

```
colnames(temp)

## [1] "Luzern" "Basel"  "Chur"   "Zurich"

rownames(temp)

## [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun"
```

For a first overview of the data, we use the command `summary(...)`:

```
summary(temp)
```

##	Luzern	Basel	Chur	Zurich
## Min.	: 2.00	Min. : 5.00	Min. : -3.00	Min. : 0.00
## 1st Qu.:	6.25	1st Qu.: 7.25	1st Qu.: 4.00	1st Qu.: 5.00
## Median	:13.00	Median :11.50	Median :13.50	Median :12.50
## Mean	:13.17	Mean :13.00	Mean :11.50	Mean :12.67
## 3rd Qu.:	19.75	3rd Qu.:18.75	3rd Qu.:19.25	3rd Qu.:19.25
## Max.	:25.00	Max. :23.00	Max. :23.00	Max. :27.00

**Min.** is the minimal value, **1st Qu.** the first (or lower) quartile (see lecture notes), **Median** the median (see lecture notes), **Mean** the mean (average), **3rd Qu.** the third or upper quartile (see lecture notes) and **Max.** the maximal value of the corresponding columns.

Now, we want to know for example what the temperature in February in Zurich was. This can be done in different ways.

```
temp[2, 4]
```

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

We access the value in the 2nd row and the 4th column.

*Important:* The first number *always* correspond to the row(s) and the second to the column(s)<sup>5</sup>.

However, for large data frames this method is not practicable. It's much easier to access the values with their row name and column name:

```
temp["May", "Basel"]
```

```
## [1] 23
```

This is the temperature in May in Basel.

We can also read whole rows or columns:

```
temp["Jun", ]
```

```
##      Luzern Basel Chur Zurich
## Jun      25    21   23     27
```

```
temp[, "Basel"]
```

```
## [1]  5  6 11 12 23 21
```

---

<sup>5</sup>This convention is also used for entries in matrices.

For columns, the following command is possible

```
temp["Basel"]  
  
##      Basel  
## Jan      5  
## Feb      6  
## Mar     11  
## Apr     12  
## May     23  
## Jun     21
```

We can calculate the mean of a column:

```
mean(temp[, "Luzern"])  
  
## [1] 13.16667
```

An important task is to save data sets or part of it. This can be done with the command `write.csv(...)` for .csv-files.

```
temp1 <- temp[c("Jan", "May"), c("Chur", "Zurich")]  
  
temp1  
  
##      Chur Zurich  
## Jan    -3      4  
## May    21     20  
  
write.csv(temp1, file = "weather1.csv")
```

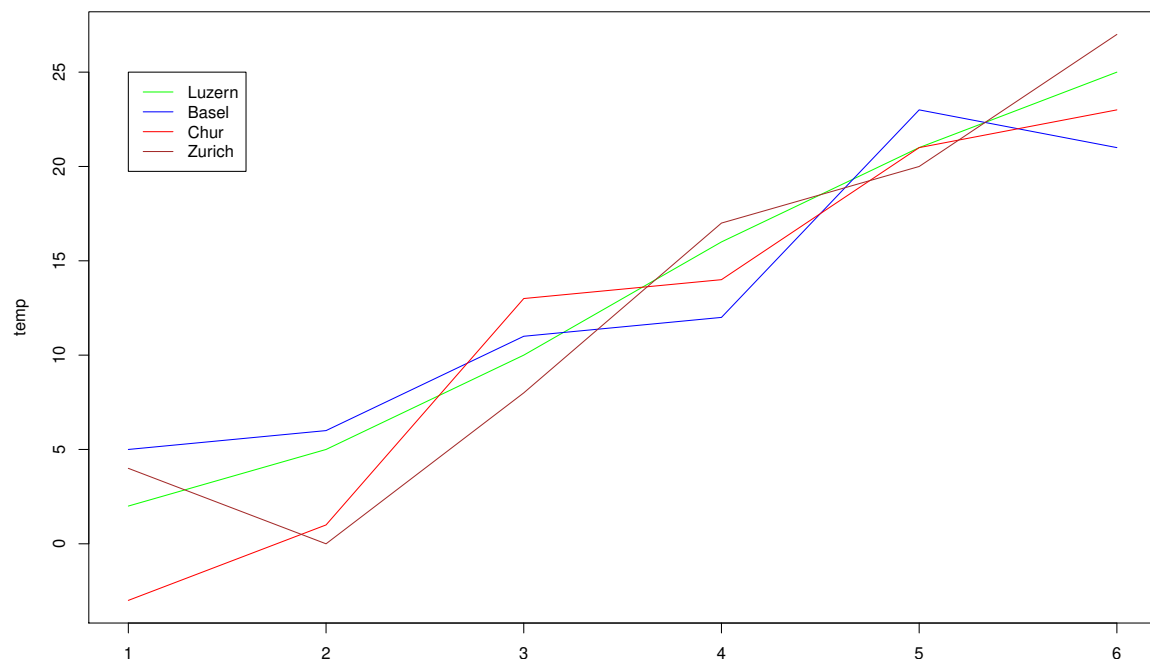
Are we interested in a specific separation of the values in the data set, then this is done with the appropriate option `sep = ....`. By default, entries are separated with commas. For other `write`-command the default setting is generally different. To be on the safe side, always set `sep = ","`.

```
write.csv(temp1, file = "weather1.csv", sep = ",")
```

### 4.4.3 A example for a plot

An important tool in statistics is the graphical representation of the data. Here we give just a simple example, you'll learn many more applications of plots during this course.

```
color <- c('green', 'blue', 'red', 'brown')  
  
matplot(temp, col = color, type = "l", lty = 1)  
  
legend(x = 1, y = 25, legend = colnames(temp), col = color, lty = 1)
```



The first command

```
farben <- c('green', 'blue', 'red', 'brown')
```

defines a vector with colours as entries. We'll use this vector in the 2nd and 3th command.

The second command

```
matplot(temp, col = color, type = "l", lty = 1)
```

plots the graphs of the temperature in the columns. The option `col = ...` stands for *colors* which means that the graphs are colored according the vector `col = color`. The vector `color` has been defined above.

The options `type = "..."` and `lty = ...` are explained in the exercises.

The third command

```
legend(x = 1, y = 25, legend = colnames(temp), col = color, lty = 1)
```

put the legend in upper left corner. The first two arguments 1 and 25 are the coordinates of the position of the legend. The option `legend = colnames(temp)` designate the names for the lines, which in this case are obviously the column names of the data frame.