Introduction to R

Methods of Scientific Working (for Crop Sciences) (3502-440)

21 Jan 2025

Table of contents

The goal of this course is a short introduction into the R statistical package. R is a very powerful environment for data analysis, and also a programming language. It is mainly used for the reproducible analysis of data and use only a minimal set of programming instructions.

Beginners may be overwhelmed by the perceived complexity of the R package. With experience, however, this feeling will go away rapidly because the underlying principles in design and usage are easy to learn and understand.

The key advantage of using a package like R over statistical analysis programs that are based on graphical user interfaces (GUIs) is the possibility to write scripts and other types of text files that simultaneously serve as notebooks and therefore greatly contribute to the repeatability and reproducibility of the analyses.

R is an open source software. It is both a programming language and a computing environment for statistical analyses. It can be downloaded from http://www.R-project.org.

Several GUIs and editors are available for R. We will use the development environment Rstudio (free software, available at www.rstudio.org). Both R and Rstudio are available for Linux, Windows and MAC OS.

This tutorial is based on several sources:

- Original writing by members of the research group Crop Biodiversity and Breeding Informatics (in particular, Fabian Freund and Karl Schmid)
- http://cran.r-project.org/doc/contrib/Torfs+Brauer-Short-R-Intro.pdf

If you are very interested in learning R, there are excellent online materials. For example:

- https://swcarpentry.github.io/r-novice-gapminder/
- https://evolutionarygenetics.github.io/

1 RStudio: Your work environment

RStudio is a Graphical User Interface (GUI) for you to develop your R codes. RStudio has several useful features to assist your programming, such as auto-completion of unfinished codes and highlighting code blocks. Please note that ***you have to install both R and RStudio*** because RStudio is just an environment for editing your codes and it runs your codes by calling R in the background.

1.1 RStudio on your own computer

After you downloaded RStudio you ca open the program. The RStudio interface consists of several windows.

- *Bottom left:* console window (also called command window). Here you can type commands after the ">" prompt and R will then execute your command. This is the most important window, because this is where R actually does stuff.
- *Top left:* editor window (also called script window). Collections of commands (scripts) can be edited and saved. When you don't get this window, you can open it with File → New → R script Just typing a command in the editor window is not enough, it has to get into the console window before R executes the command. If you want to run a line from the script window (or the whole script), you can click Run or press CTRL+ENTER to send it to the console window.
- *Top right:* workspace / history window. In the workspace window you can see which data and values R has in its memory. You can view and edit the values by clicking on them. The history window shows what has been typed before.
- *Bottom right:* files / plots / packages / help /viewer window. Here you can open files, view plots (also previous plots), install and load packages or use the help function.

You can change the size of the windows by dragging the grey bars between the windows.

1.2 RStudio Cloud

RStudio Cloud (https://rstudio.cloud/) is a cloud-based version of RStudio that allows you to run your analysis online. You can easily sign up with your Google account or with any Email box for free.

The interface of RStudio Cloud is as same as your local RStudio.

1.3 R command line in RStudio

In Rstudio, we can either use the normal command line input for R or write scripts in the editor and run these in R. We will focus on the command line input.

Some important basic commands are:

```
#| eval: false
q()                         # quit R
?command_name               # call manual for a command; try q()
getwd()                     # show the current working directory
setwd("directory_name")  # set a working directory
```

2 Working directory

Your working directory is the folder on your computer in which you are currently working. When you ask R to open a certain file, it will look in the working directory for this file, and when you tell R to save a data file or figure, it will save it in the working directory. Before you start working, please set your working directory to where all your data and script files are or should be stored.

The working directory can also be set in Rstudio by clicking on **`Session -> Set working directory -> Choose directory`**. The working directory is important since it is the directory where all output of R will be written to.

Note that everything written after an # is not evaluated by R - we will use this for commenting.

Tip: If R reports an error message: `Error in file(file, "rt") : cannot open the connection ... No such file or directory` when trying to read data, it means you give an incorrect directory. You have to check the typing error(s) in your codes.

3 Libraries

R can do many statistical and data analyses. They are organized in so-called packages or libraries. With the standard installation, most common packages are installed. To get a list of all installed packages, go to the packages window or type `library()` in the console window. If the box in front of the package name is ticked, the package is loaded (activated) and can be used.

There are many more packages available on the R website. If you want to install and use a package (for example, the package called "geometry") you should:

- Install the package: click install packages in the packages window and type geometry or type `install.packages("geometry")` in the console window.
- Load the package: check box in front of geometry or type `library("geometry")` in the console window.

Tip: An important thing to keep in mind is that **R packages are not always available with** `install.packages()`. The default of `install.packages()` searches packages on the CRAN repository. However, in some cases, R packages may be archived on bioconductor, Github or other repositories. So, if R returns an warning message of `package is not available` when installing a package, try to search for a correct source for your installation.

Tip: Please note that you have to reload R packages with `library()` whenever you start a new R session.

First examples of R commands

4 R as calculator - Part 1

R can be used as a calculator. You can just type your equation in the command window after the > prompt:

```
> 10^2 + 36
```

and R will give the answer

```
[1] 136
```

4.1 Exercise

Compute the difference between 2022 and the year you started at this university and divide this by the difference between 2022 and the year you were born. Multiply this with 100 to get the percentage of your life you have spent at this university. Use brackets if you need them.

Note: If you use brackets and forget to add the closing bracket, the > on the command line changes into a +. The + can also mean that R is still busy with some heavy computation. If you want R to quit what it was doing and give back the >, press `ESC` (see the reference list on the last page).

5 Workspace

You can also give numbers a name. By doing so, they become so-called *variables* which can be used later. For example, you can type in the command window:

```
> a <- 4
```

You can see that a appears in the workspace window, which means that R now remembers what a is. You can also ask R what a is (just type a ENTER in the command window):

```
> a
[1] 4
```

or do calculations with a:

```
> a * 5
[1] 20
```

If you specify a again, it will forget what value it had before. You can also assign a new value to a using the old one.

```
> a <- a + 10
> a
[1] 14
```

To remove all variables from R's memory, type

```
> rm(list=ls())
```

or click `clear all` in the workspace window. You can see that RStudio then empties the workspace window. If you only want to remove the variable a, you can type `rm(a)`.

The workspace can be saved permanently after a session and reloaded to use objects defined in an earlier session. The workspace can be saved in Rstudio using the menu in the upper right corner. If you terminate R, you will always be asked whether the workspace should be saved (it is saved in the working directory).

5.1 Exercise

Repeat the previous exercise, but with several steps in between. You can give the variables any name you want, but the name has to start with a letter.

Scalars, vectors and matrices

Like other programs, R organizes numbers in

- scalars (a single number - 0-dimensional),
- vectors (a row of numbers, - 1-dimensional)
- matrices (like a table - 2-dimensional).

The a you defined before was a scalar. To define a vector with the numbers 3, 4 and 5, you need the function `c`, which is short for *concatenate* (paste together).

```
b <- c(3,4,5)
```

Matrices and other 2-dimensional structures will be introduced below.

6 R Style guide

You may wonder whether it is better to write `rm(list=ls())` or `rm(list = ls())`, or to write `b <- c(3,4,5)` or `b <- c(3, 4, 5)`. All of these variants work and their use is a matter of taste.

Simple rule can be applied to decide on the writing:

- As long as you write for yourself, decide on what you like
- But be consistent throughout your code, it makes reading easier

- If you write for others. Follow a generally accepted *style guide*. A widely used style guide for R is here: https://style.tidyverse.org/

The assignment operator

Historically, R used `<-` as an assignment operator, but = can be used as well because it does the same thing. `<-` consists of two characters, < and -, and represents an arrow pointing at the object receiving the value of the expression. In this introduction, we use `<-` as assignment operator.

Objects, values and classes

When using R, your data, functions, results etc. are stored in the active memory of the PC in the form of **objects** of different **classes** to which you assign **names**.

Here is a summary of basic classes:

- `integer`: numbers that can be written without a decimal component
- `numeric` (or `double`): any number; can be positive or negative
- `character`: any text, including symbols and numbers (***numbers of `character` class CANNOT be used in calculation***)
- `logical`: value of `TRUE` or `FALSE`

The data of basic classes are building blocks of your objects:

Object	Class
vector	numeric, integer, character, logical
factor	numeric, integer, character
array	numeric, integer, character, logical
matrix	numeric, integer, character, logical
data frame	numeric, integer, character, logical
list	numeric, character, logical, function, expression...

Note: `factor` is the text with categorical information (***so it CAN be used in statistical models***)

First let us focus on **numerical objects**. To create numerical objects we need to write our result to a variable. A variable is an **object** to which we have given a **name** and assign a **value**.

Here is an example:

```
x <- 1
y <- 2
z <- 10
Peter <- x + y
Bernard <- y - x
Rabbit <- z * Peter
```

Now we can display the **values** of these variables simply by typing their **name**.

```
x
```

etc.

Vectors and functions

Often, we will not only deal with single objects, but with several objects at once. For example, various measurements may have been taken from the same plant. All measurements combined in one variable are a *vector* (an ordered set) of entries.

7 Vectors

For example, the vector (1,2,3,4) can be defined by

```
v1 <- c(1,2,3,4)
```

A vector consists of objects of the same class and that by using `c()` on vectors, you can concatenate vectors. Try using the command `str` instead of mode to get further information about an object.

```
v2 <- c("a", 1, 2, 3)
mode(v2)
str(v2)
v3 <- c(v1, v1, v1)
v3
```

Instead of typing all entries by hand, vectors consisting of copies of the same element or of equidistant values can be defined by the following commands:

```
rep(4, 6)       # The first argument gives the object to repeat,
                # the second argument the number of repetitions
seq(2, 4, 0.5)  # Consists of values with distance 0.5 from 2 to 4 (including 2 and 4)
1:7             # A vector consisting of 1,...,7
seq(along = v2) # Vector (1,...,length(v2))
```

Note that `length(v1)` gives the length of the vector `v1`.

7.1 Exercise

Construct a vector of length 300 consisting of 50 copies of `1,2,3,4,5,6`.

8 Working with vectors

Let `v1` be a vector with numerical entries, for example

```
v1 <- 1:5
mode(v1)
```

A certain entry, say the$i$th entry of `v1` can be accessed by `v1[i]`. Note that you can also access a sub vector by specifying all of the entries you want to access.

```
v1[1] # The first entry of the vector`
v1[c(1, 2)] # The first two entries of the vector`
v1[-c(1, 2)] # All entries of the vector apart from the first two`
v1[v1<3] # All entries smaller than 3`
```

Question: What kind of an object is `v1<3`?

In the last example, we introduced a vector consisting of objects of class `logical`. Let's look at it:

~~~
```{r, results='show'}
v1 < 3
```
~~~

Such operations (a vector, a comparison operator and a object the vector is compared to) produces a vector giving the result of the comparison for each vector entry.

Here is a list of logical comparison operators:

- ==, !=: equal, unequal
- >, >=: greater, greater or equal
- <, <=: smaller, smaller or equal
- &, |: and, or (to combine logical expressions, each expression has to be put into ())
- `!(`logical condition`)`: negation of a logical condition

```
logicv1 <- (v1 > 1) & (v1 < 4)
v1[logicv1]
```

So far, accessing entries of a vector resulted in the output of the accessed entries, discarding the information at which position of the original vector the entries are placed. This information can be retrieved by `which`.

```
v4 <- -7:3  # Defines v4 as the vector (-7, -6, ..., 1, 2, 3). Note the blank!
which(v4 > 0)   # Gives the positions of all entries of v4 bigger than
```

8.1 Exercise

In the previous exercise, you constructed a vector of length 300 consisting of 50 copies of `1,2,3,4,5,6`. How would you extract the values between 2 and 5 from this vector?

Note that to get the position of a minimal or maximal entry of a numerical vector `v1`, you can use `which.min(v1)` and `which.max(v1)` (if there are several entries tied for minimum or maximum, the entry with lowest position is shown). To overwrite entries in a vector, you just have to assign a new value to the entry:

```
v1[1] <- 100
v1[1]
v1
```

Here's a list of some useful functions for a numerical vector `v`:

- `max(v)`, `min(v)`: Gives the maximal/minimal value of a vector
- `sum(v)`: Sums the entries of `v`
- `mean(v)`: arithmetic mean of `v`
- `sort(v)`: sort entries of `v` in increasing order. To sort in decreasing order, add the second argument `decreasing=TRUE`

For a logical vector `v`, the following commands might be useful:

- `any(v)`: Is at least one entry of `v` TRUE?
- `all(v)`: Are all entries of `v` TRUE?

9 Functions

If you would like to compute the mean of all the elements in the vector `b` from the example above, you could type

```
> (3 + 4 + 5) / 3
```

But when the vector is very long, this is very boring and time-consuming work. This is why things you do often are automated in so-called functions. Some functions are standard in R or in one of the packages. You can also program your own functions. When you use a function to compute a mean, you will type:

```
> mean(x = b)
```

Within the brackets you specify the *arguments*. Arguments give extra information to the function. In this case, the argument `x` says of which set of numbers (vector) the mean should computed (namely of `b`?). Sometimes, the name of the argument is not necessary: mean(b) works as well.

9.1 Exercise

Compute the sum of 4, 5, 8 and 11 by first combining them into a vector and then using the function `sum`.

The function `rnorm`, as another example, is a standard R function which creates random samples from a normal distribution. Hit the `ENTER` key and you will see 10 random numbers as:

```
> rnorm(10)
[1] -0.949  1.342 -0.474 0.403
[5] -0.091 -0.379  1.015  0.740
[9] -0.639  0.950
```

- Line 1 contains the command: `rnorm` is the function and the 10 is an argument specifying how many random numbers you want - in this case 10 numbers (typing `n=10` instead of just 10 would also work).
- Lines 2-4 contain the results: 10 random numbers organised in a vector with length 10.

Entering the same command again produces 10 new random numbers. Instead of typing the same text again, you can also press the upward arrow key to access previous commands. If you want 10 random numbers out of normal distribution with mean 1.2 and standard deviation 3.4 you can type

```
> rnorm(10, mean = 1.2, sd = 3.4)
```

showing that the same function (`rnorm`) may have different interfaces and that R has so called named arguments (in this case mean and sd). By the way, the spaces around the “,” and “=” do not matter, but the use of spaces is frequently recommended to improve the readbility of the code.

Comparing this example to the previous one also shows that for the function rnorm only the first argument (the number 10) is compulsory, and that R gives default values to the other so-called optional arguments.

Tip: RStudio has a nice feature: when you type `rnorm(` in the command window and press TAB, RStudio will show the possible arguments.

10 R as calculator - Part 2

The R console can be used as pocket calculator and basic arithmetic calculations `+`, `-`, `*` and `/` are easy to carry out. Many more complex mathematical operations are available, for example:

- `^` : power
- `sqrt(x)` : square root of `x`
- `log(x)`, `exp(x)` : natural logarithm and exponential function of `x`, respectively
- `sin(x)`, `cos(x)`, `tan(x)` : trigonometric functions of `x`
- `abs(x)` : absolute value `|x|` of `x`

Other object classes

So far, we have used one type of object classes, `numeric`. Four other important object classes are `character`, `logical`, `integer` and `function`.

The class `character` consists of objects that are strings of symbols (e.g. words but also something like `A?7Fd`).

The class `logical` is reserved for the logical expressions `TRUE` and `FALSE`. Such objects are often useful in programming.

`integer` is a class which only allows integer-valued entries.

`NA` is the class reserved for missing values. `NA` is not really an object class, but gets it class information from what object class the value is missing from. If no such information is available, `NA` is treated as logical.

```r
word <- "hello"
word2 <- "A?7Fd"
mv <- NA
Bool <- TRUE
```

To display a `class` of an object one can use a function like `mode()`

```r
mode(x)
mode(sin)
mode(word)
mode(Peter)
```

Plots

R can make graphs. This is a very simple example:

```r
x <- rnorm(100)
plot(x)
```

- In the first line, 100 random numbers are assigned to the variable x, which becomes a vector by this operation.
- In the second line, all these values are plotted in the plots window.

10.1 Exercise

Plot 100 normal random numbers.

Help and documentation

There is a large amount of free documentation and help available. Some help is automatically installed. Typing in the console window the command

```
help(rnorm)
```

gives help on the `rnorm` function. It gives a description of the function, possible arguments and the values that are used as default for optional arguments. Typing

```
example(rnorm)
```

gives some examples of how the function can be used. An HTML-based global help can be called with:

```
help.start()
```

or by going to the help window.

The following links can also be useful:

- https://cran.r-project.org/doc/manuals/R-intro.pdf – A full manual.
- http://cran.r-project.org/doc/contrib/Short-refcard.pdf – A short reference card.
- www.statmethods.net – Also called Quick-R. Gives very productive direct help. Also for users coming from other programming languages.
- mathesaurus.sourceforge.net – Dictionary for programming languages (e.g. R for Matlab users).
- Just using Google (type e.g. “R rnorm” in the search field) can also be very productive.

10.2 Exercise

Find help for the `sqrt` function.

Scripts

R is an interpreter that uses a command line based environment. This means that you have to type commands, rather than use the mouse and menus. This has the advantage that you do not always have to retype all commands and are less likely to get complaints of arms, neck and shoulders.

You can store your commands in files, the so-called scripts. These scripts have typically file names with the extension .R, e.g. `foo.R`. You can open an editor window to edit these files by clicking `File` and `New` or `Open file...`, where also the options `Save` and ‘Save as are available.

You can run (send to the console window) part of the code by selecting lines and pressing `CTRL+ENTER` or click `Run` in the editor window. If you do not select anything, R will run the line your cursor is on. You can always run the whole script with the console command source, so e.g. for the script in the file `foo.R` you type:

```
source("foo.R")
```

You can also click `Run all` in the editor window or type `CTRL+SHIFT+S` to run the whole script at once.

10.3 Exercise

Make a file called `firstscript.R` containing R-code that generates 100 random numbers and plots them, and run this script several times.

Note: You can add comments to the script explaining what the code does. Type these comments behind a # sign, so is not evaluated as code by R.

Data structures

If you are unfamiliar with R, it makes sense to just retype the commands listed in this section. Maybe you will not need all these structures in the beginning, but it is always good to have at least a first glimpse of the terminology and possible applications.

11 Vectors

Vectors were already introduced, but they can do more:

```
> vec1 <- c(1, 4, 6, 8, 10)
> vec1
[1]  1  4  6  8 10
> vec1[5]
[1] 10
> vec1[3] <- 12
> vec1
[1] 1 4 12 8 10
> vec2 <- seq(from = 0, to = 1, by = 0.25)
> vec2
[1] 0.00 0.25 0.50 0.75 1.00
> sum(vec1)
[1] 35
> vec1 + vec2
[1] 1.00 4.25 12.50 8.75 11.00
```

- In line 1, a vector `vec1` is explicitly constructed by the concatenation function `c()`, which was introduced before. Elements in vectors can be addressed by standard `[i]` indexing, as shown in lines 4-5.
- In line 6, one of the elements is replaced with a new number. The result is shown in line 8.
- Line 9 demonstrates another useful way of constructing a vector: the `seq()` (sequence) function.
- Lines 10-15 show some typical vector oriented calculations. If you add two vectors of the same length, the first elements of both vectors are summed, and the second elements, etc., leading to a new vector of length 5 (just like in regular vector calculus). Note that the function sum sums up the elements within a vector, leading to one number (a scalar).

12 Matrices

Matrices are nothing more than 2-dimensional vectors. To define a matrix, use the function matrix:

```
> mat <- matrix(data = c(9,2,3,4,5,6), ncol = 3)
> mat
     [,1] [,2] [,3]
[1,]    9    3    5
[2,]    2    4    6
```

The argument data specifies which numbers should be in the matrix. Use either ncol to specify the number of columns or nrow to specify the number of rows.

More formally defined, a matrix is a rectangular scheme of $n \cdot m$ values, where n is the number of rows and m is the number of columns. A matrix can be defined by listing all entries in a vector, specifying the number of rows and stating whether the entries are ordered by rows or columns. The $n \times n$ identity matrix can be defined by `diag(n)`.

```
matrix1 <-  matrix(c(1, 2, 3, 4), nrow = 2, byrow = TRUE)   # Ordered by rows
matrix2 <-  matrix(c(1, 2, 3, 4), nrow = 2, byrow = FALSE)  # Ordered by columns
D <-  diag(2)
```

Note that since the first argument of `matrix` is a vector, you can use the commands written down in the chapter about vectors to easily built matrices with specific patterns in their entries. To access an entry of a matrix, you have to specify its row and column. As in the case of vectors, you can also access several entries at once.

```
matrix1[1,2]    # Accesses the entry in the 1st row, 2nd column
matrix1[ ,1]    # Accesses the first column
matrix1[1, ]    # Accesses the first
```

There are many operations available to manipulate matrices. The following list shows some important commands for matrices:

```
v3 <- c(1, 1)           # Defines a 2-dimensional vector
t(matrix1)              # Transposes matrix1 (switches rows with columns)
matrix1 %*% matrix2     # Multiplies matrix1 with matrix 2
matrix1 %*% v3          # Multiplies matrix with vector
eigen(matrix1)          # Computes eigenvalues and eigenvectors
solve(matrix1)          # Inverts the matrix
solve(matrix1, v3)      # Solves the system of linear equation matrix1*x=v3
cbind(matrix1, v3)      # Adds v3 as a new column (works also adding matrices)
rbind(matrix1, v3)      # Adds v3 as a new row (works also adding matrices)
```

An expansion of the concept of matrix for more than two dimensions is `array`.

12.1 Exercise

Put the numbers 31 to 60 in a vector named `P` and in a matrix with 6 rows and 5 columns named `Q`. Tip: use the function `seq`. Look at the different ways scalars, vectors and matrices are denoted in the workspace window.

Matrix operations are similar to vector operations:

```
> mat[1,2]
[1] 3
> mat[2,]
[1] 2 4 6
> mean(mat)
[1] 4.8333
```

- Elements of a matrix can be addressed in the usual way: `[row,column]` (line 1).
- Line 3: When you want to select a whole row, you leave the spot for the column number empty (the other way around for columns of course).
- Line 5 shows that many functions also work with matrices as argument.

13 Data frames

Time series are often ordered in data frames. A data frame is a matrix with names above the columns. This is nice, because you can call and use one of the columns without knowing in which position it is.

```
> t <- data.frame(x = c(11,12,14), y = c(19,20,21), z = c(10,9,7))
> t
  x  y  z
1 11 19 10
2 12 20 9
3 14 21 7
> mean(t$z)

[1] 8.666667
> mean(t[["z"]])
[1] 8.666667
```

- In lines 1-2 a typical data frame called `t` is constructed. Its columns have the names `x`, `y` and `z`.
- Line 8-11 show two ways of how you can select the column called `z` from the data frame called `t`.

Normally, we will deal with data collected from different individuals. This data can be seen as a scheme with rows and columns (similar to a matrix), where the rows stand for the individuals and the columns stand for each measured variable or some information about **experimental factors**. Note here that in contrast to a matrix, the columns may have any object as entry (but the type of object is equal for all rows/individuals). This type of data structure is called a **data frame** in R. It can be defined by the command `data.frame` as follows:

```
data1 <- data.frame(height = c(3, 4, 5, 3),
                    earlength = c(5, 5, 4, 2),
                    treatment = c("c", "a", "a", "b"))
                    # Note that the third column is referred to as a factor
str(data1)
```

If we compare defining a data frame to defining a matrix, we see that we enter each column as a separate vector and we can name the columns (similar naming is possible for matrices and vectors). Non-numerical values are mostly experimental conditions in a data frame and will be treated as factors. To access entries, there are two possibilities: We can do as with matrices or directly address the columns.

```
data1[1, 2]               # Accesses the entry in the 1st row, second column
data1$earlength[1]      # Does the same
data1$earlength             # The column earlength
data1[[1]]              # The first column
```

The benefit of using the column names is that you don't have to memorize the exact structure of the data frame, but just the column names (thus, use reasonable column names). For programming though, it's often easier to address the columns by number and not by name. If you are about to work with one data frame a lot, you can use `attach()` to add the data frame to the search path of R. This means that R knows that if you type in a column name, it's from said data frame. You can detach by using `detach()`

```
#| eval: false
data1$height
height            # Doesn't work
attach(data1)
height            # Works now
data1$height
detach(data1)
```

We know how to access different columns of a data frame. However, we will often be interested to work with a subset of data, for example only data from individuals/rows under a certain experimental condition (e.g., a certain factor level of an experimental factor). Such subsets of data can be accessed by `subset`.

```
subset(data1, treatment == "a") # Chooses all rows with treatment a
data_sub <- subset(data1, treatment %in% c("a", "b"))   # Chooses all rows with treatment a or b
str(data_sub)
table(data_sub$treatment) # Unused factor levels are kept by subset
data_sub2 <- droplevels(data_sub$treatment) # Kicks out unused factor levels
table(data_sub2)
subset(data1, treatment == "a", select = height) # Shows the height values for all individuals \# wi
```

As with matrices, `rbind` and `cbind` can be used to glue data sets together. A more flexible command is `merge` (which we don't cover here), to learn about it type `?merge`. A similar, but more flexible class for such lists is `list`. All defined objects are displayed in a list in the upper right corner in Rstudio.

13.1 Exercise

Make a script file which constructs three random normal vectors of length 100. Call these vectors `x1`, `x2` and `x3`. Make a data frame called `t` with three columns (called `a`, `b` and `c`) containing respectively `x1`, `x1+x2` and `x1+x2+x3`. Call `plot(t)` for this data frame. Can you understand the results? Re-run this script a few times.

14 Lists

Another basic structure in R is a list. The main advantage of lists is that the "columns" (they're not really ordered in columns any more, but are more a collection of vectors) don't have to be of the same length, unlike matrices and data frames.

```
> L <- list(one = 1, two = c(1,2), five=seq(0, 1, length = 5))
> L
$one
[1] 1
$two
[1] 1 2
$five
[1] 0.00 0.25 0.50 0.75 1.00
> names(L)
[1] "one"  "two"  "five"
> L$five + 10
[1] 10.00 10.25 10.50 10.75 11.00
```

- Lines 1-2 construct a list with names and values. The list also appears in the workspace window.
- Lines 3-9 show a typical printing (after pressing `L ENTER`).
- Line 10 illustrates how to find out what is in the list.
- Line 12 shows how to use the numbers.

Functions

To understand computations in R, two slogans are helpful:

- Everything that exists is an object.
- Everything that happens is a function call.

```
v2 <- c("a", 1, 2, 3)
str(v2)
```

So if we go back to this example from before we now know that `v2` is an object. `str()` on the other hand is a function that provides us with some information on `v2`.

How to make a function?

All R functions have three parts:

- the `body`, the code inside the function.
- the `formals`, the list of arguments which controls how you can call the function.
- the `environment`, the map of the location of the functions variables.

Here we will focus on the first two components.

To define a new function, we have to specify the arguments of the function, a function name and the function itself. For example, we can define the function that calculates *2*b + 2* by:

```
a <- function(b) {
    2 * b + 2
}
formals(a)
body(a)
mode(a) # shows the mode of a
a(4) # computes the value of the function a for an input number 4
```

A function may have more than one argument, and the arguments don't necessarily have to be objects of the class `numeric`. For example R can also plot mathematical functions by using the function `curve`. `curve` has many possible arguments. Type `?curve` to get an overview. Note that some arguments have a predefined **default value**, meaning that if you don't specify a value for such an argument, the default value is used. For starters, we will focus on the arguments:

- `expr`: The function which to plot
- `from`: Lower bound of the x-coordinate of the plot
- `to`: Upper bound of the x-coordinate of the plot
- `xlab`: Label of the x-axis, can either be written in text ("text") or as a mathematical expression (using `expression()`)
- `ylab`: Label of the y-axis, , can either be written in text ("text") or as a mathematical expression (using `expression()`)

Here's the command to let R plot the function `sin2x` from -2π to 2π (with labelled axes).

```
#| eval: false
sin2x <- function(x) {
    sin(2**x)
}
curve(sin2x, from = -2 * pi, to = 2 * pi, xlab = "x ",
      ylab = expression(sin(2 * x)))
```

Note that R doesn't keep track of objects defined in a function unless you force it to return their values. By default, just the last evaluated expression is returned (as seen). Using `return` at the end of your function, you can specify the return values. Here's an example:

```
testf <- function(x) {y <- 2*x; x+y}
testf(2) # returns only the value of the function, y is not returned
testf <- function(x) {y <- 2*x; z <- x+y; return(c(y, z))}
testf(2)  #  y and z are returned
```

14.1 Exercises:

A) Plot a function `x*x` or `x^2` (both will of course give the same result) for `x` from -100 to 100.

B) Write a function that computes the mean of all negative and the mean of all positive values of a vector.

C) Consider an very big (infinite) population of diploid individuals. A locus with alleles A_1 and A_2 is in Hardy-Weinberg equilibrium if the genotype frequencies are

Genotype	A_1A_1	A_1A_2	A_2A_2
Frequency	p^2	$2pq$	q^2

where p is the frequency of A_1 and $q = 1 - p$ is the frequency of A_2.

Create a R script and write a function `HWfreq` that returns the genotype frequencies if you use the A_1 allele frequency p.

Graphics

One of the main strengths of R comes from its strong graphical possibilities. Here we will just learn the basics of the plotting functions while it is encouraged to look into various online plotting tutorials if you want to learn more:

- https://www.statmethods.net/graphs/

The following lines show a simple plot

```
plot(rnorm(100), type = "l", col = "gold")
```

Hundred random numbers are plotted by connecting the points by lines (the symbol between quotes after the `type=`, is the letter l, not the number 1) in gold.

Another very simple example is the classical histogram plot, generated by the simple command

```
hist(rnorm(100))
```

The following few lines create a plot using the data frame `t` constructed in the previous exercise:

```
plot(t$a, type = "l", ylim = range(t), lwd = 3, col = rgb(1, 0, 0, 0.3))
lines(t$b, type="s", lwd=2, col = rgb(0.3, 0.4, 0.3, 0.9))
points(t$c, pch = 20, cex = 4, col = rgb(0, 0, 1, 0.3))
```

Note that with plot you get a new plot window while points and lines add to the previous plot.

Add these lines to the script file of the previous section. Try to find out, either by experimenting or by using the help, what the meaning is of `rgb`, the last argument of `rgb`, `lwd`, `pch`, `cex`.

To learn more about formatting plots, search for `par` in the R help. Google “R color chart” for a pdf file with a wealth of color options.

To copy your plot to a document, go to the plots window, click the “Export” button, choose the nicest width and height and click `Copy` or `Save`.

15 More advanced plotting

```r
#| eval: false
data(iris) # loading a plant dataset already existing in R
class(iris) # lets see what type of data this is
summary(iris) # summary of the dataset
plot(iris$Sepal.Length,iris$Sepal.Width) # plots the length and width of the plants
?plot # shows us all the options we can use with the plot function
plot(iris$Petal.Length, iris$Petal.Width, pch=21,
    bg=c("red","green3","blue")[unclass(iris$Species)],
    main="Iris_Data",
    xlab="Petal_length",ylab="Petal_Width")
```

16 Basic linear regression models

To fit a linear model to a data set, we just have to specify the linear model we want to use and then plot the data using the `plot()` function.

```r
#| eval: false
x <- c(1,2,3,4,5)
y <- c(1.6,4,6.5,7.5,10)
plot(x,y)
```

Here, you can again add graphical arguments to plot. Now we want to add a regression line. We define the regression of y on x as:

```r
#| eval: false
reg <- lm(y~x)
reg
str(reg)
abline(reg) # draws the regression line into the plot
```

Function `abline` just draws a line with a certain$y$-axis intercept and slope (here given by the object `reg`). See `?abline` for detail. The function `lm` uses the given formula to fit a linear model. This model can have several independent variables (we used one, namely `x`) and may also include interactions between the independent variables (nested designs are also possible). Let's go through a little example. Denote again with y the dependent variable and with `x1,x2,x3` the independent variables.

If you want to include

- no interactions, the model is `y ~ x1 + x2 + x3`
- only all interactions of two variables, the model is `y ~ (x1 + x2 + x3)^2`
- all interactions, the model is `y ~ x1 * x2 * x3`

To get more information, type `?lm`. A good analysis of the `iris` dataset using linear models (`lm`) can be found here: https://warwick.ac.uk/fac/sci/moac/people/students/peter_cock/r/iris_lm/

Reading and writing data files

Importing data from external sources such as tables from spreadsheet programs like Excel, text files of measurement, output data from other programs, or data from big databases is a frequent task in data analysis.

We start first with a data set in format `.txt`, which we will create using the built-in editor in Rstudio. Open a new `.txt`-file by clicking on the button marked with + in the upper right corner in the GUI and choosing a new text file.

Type in:

height	earlength	experiment
150	15	a
120	11	b
135	11	c

Note here that we have a heading containing the column names and semicolons which separate different values (imagine the data coming from measuring the traits in crop plants grown under different experimental conditions). Save the file as`data2.txt` in your working directory. We will now import this data set as a data frame in R. This can be either done by clicking on `Import Dataset` in Rstudio in the upper right corner (and specify the header, the separation symbol etc.). The same can be done by using the command `read.csv` with suitable parameters.

```
#| eval: false
data2 <- read.csv("data2.txt", sep = ";")
```

The argument `header` has the default value `TRUE` meaning that the program reads in the first row of the text as the header containing column names (For numbers, `read.csv` expects the decimal symbol `.`). The same command also works for excel files if you export the files from Excel as csv files.

A similar command is `read.table`, but it has different default values for the arguments.

Data frames can be saved as text files with `write.table`, which has the same arguments and default values as `read.table`. You only have to specify the name of the output file:

```
#| eval: false
write.table(data1, "data1.txt", sep = ";")  # Separation symbol ;
```

You can view output file in the built-in editor of Rstudio.

Another command for writing to text files is `write.csv`. It allows no control on arguments to enable problem-free export to Excel.

To save a R-object as a R-object, which is not a text file but a binary object, you can use the command `save`. Hdere is an example to write R objects `v1` and `v2` to a file with `save` by writing all objects into a file called `vector.RData`. The ending `.RData` is not mandatory but used to indicate that the file is a binary file containing R objects, which can be imported into R with `load`.

```
#| eval: false
save(v1, v2, file = "vector")
v1 <- 0 # Change v1
v2 <- 0 # Change v2
load("vector") # Load the previous definitions of the vectors
v1
v2
```

The workspace including all defined variables can be saved by `save.image("file_name")` and loaded by `load("file_name")`.

16.1 Exercise

Make a file called `tst1.txt` in a `Text File` window of RStudio similar to the above text file and store it in your working directory. Write a script to read it, to multiply the column called `g` by 5 and to store it as `tst2.txt`.

Not available (or missing) data

As already mentioned, missing data should be coded as `NA`. One way to exclude missing data is to only keep data rows that are complete for all variables. This can be done by `na.exclude`. For vectors and data frames, it marks all missing values to be ignored in further analyses. To show the positions of `NA`, use `is.na`. It gives the same object format with logical values indicating whether there is a missing value (`TRUE`) or not (`FALSE`). To check whether there is any missing data at all, use `any(is.na())`

```r
v_na <- c(NA, 2, 4, NA)
data_na <- data.frame(b1 = 1:4, b2 = c(NA, 3, 3, NA))
data_na
is.na(v_na)
any(is.na(v_na))
is.na(data_na)
data_good <- na.exclude(data_na)    # Remove all rows with NA
data_good
str(data_good)
mean(na.exclude(v_na)) # Compute the mean of present values, no permanent change in v_na
```

16.2 Exercise

Compute the mean of the square root of a vector of 100 random numbers. What happens?

When you work with real data, you will encounter missing values because instrumentation failed or because you didn't want to measure in the weekend. When a data point is not available, you write `NA` instead of a number.

```r
j <- c(1,2,NA)
```

Calculating statistics of incomplete data sets is strictly speaking not possible. Maybe the largest value occurred during the weekend when you didn't measure. Therefore, R will say that it doesn't know what the largest value of j is:

```
> max(j)
[1] NA
```

If you don't mind about the missing data and want to compute the statistics anyway, you can add the argument `na.rm = TRUE` (Should I remove the `NA`s? Yes!).

```
> max(j, na.rm=TRUE)
[1] 2
```

Classes

The exercises you did before were nearly all with numbers. Sometimes you want to specify something which is not a number, for example the name of a measurement station or data file. In that case you want the variable to be a character string instead of a number.

An object in R can have several so-called classes. The most important three are numeric, character and POSIX (date-time combinations). You can ask R what class a certain variable is by typing `class(...)`.

17 Characters

To tell R that something is a character string, you should type the text between apostrophes, otherwise R will start looking for a defined variable with the same name:

```
> m <- "apples"
> m
[1] "apples"
> n <- pears
Error: object ‘pears’ not found
```

Of course, you cannot do computations with character strings:

```
> m + 2
Error in m + 2 : non-numeric argument to binary operator
```

18 Dates

Dates and times are complicated. R has to know that 3 o’clock comes after 2:59 and that February has 29 days in some years. The easiest way to tell R that something is a date-time combination is with the function strptime:

```
> date1 <- strptime( c("20170225230000",
+ "20170226000000", "20170226010000"),
+ format="%Y%m%d%H%M%S")
>
> date1
[1] "2017-02-25 23:00:00"
[2] "2017-02-26 00:00:00"
[3] "2017-02-26 01:00:00"
```

- In lines 1-2 you create a vector with `c(...)`. The numbers in the vectors are between apostrophes because the function strptime needs character strings as input.
- In line 3 the argument format specifies how the character string should be read. In this case the year is denoted first `(%Y)`, then the month `(%m)`, day `(%d)`, hour `(%H)`, minute `(%M)` and second `(%S)`. You don’t have to specify all of them, as long as the format corresponds to the character string.

18.1 Exercise

Make a graph with on the x-axis: today, the next end-of-year day and your next birthday and on the y-axis the number of presents you expect on each of these days. Tip: make two vectors first.

Programming tools

When you are building a larger program than in the examples above or if you're using someone else's scripts, you may encounter some programming statements. In this Section we describe a few tips and tricks.

To write complex functions or to use commands repeatedly in an automatic fashion, it is necessary to use R as a programming language. The two most important ingredients of programming are conditions (`if`, `else`) and loops (repetition of commands).

19 Conditional execution of commands

The `if`-statement is used when certain computations should only be done when a certain condition is met (and maybe something else should be done when the condition is not met). An example:

```
> w <- 3
> if( w < 5 ){
+   d = 2
+   } else {
+     d = 10
+ }
> d
[1] 2
```

- In line 2 a condition is specified: w should be less than 5.
- If the condition is met, R will execute what is between the first brackets in line 4.
- If the condition is not met, R will execute what is between the second brackets, after the else in line 6. You can leave the else{…}-part out if you don't need it.
- In this case, the condition is met and d has been assigned the value 2 (lines 8-9).

The syntax of the commands is

```
if (logical condition) {command} else {commands} # The else part is optional
```

If the logical condition equals `TRUE`, then the the commands in the first brackets are used. If `else` is used, the second set of commands will be used if the logical condition equals `FALSE`. We will now define a function which states whether a numerical value is positive.

```
#| eval: false
pos <- function(x){if (x > 0){print("positive")} else {print("not positive")}
```

Try out the new defined function `pos`!

`if`-conditions can be nested, for example we can give the sign of a numerical value b programming

```
#| eval: false
sign2 <- function(x){
  if (x > 0){
    print("positive")
    }
  else{
    if (x == 0){
      print("value is 0")
    }
    else{
      print("negative")
    }
  }
}
```

There is a (somewhat) similar function to `if` in R called `switch`.

To get a subset of points in a vector for which a certain condition holds, you can use a shorter method:

```
> a <- c(1,2,3,4)
> b <- c(5,6,7,8)
> f <- a[b == 5 | b == 8] > f
[1] 1 4
```

- In line 1 and 2 two vectors are made.
- In line 3 you say that f is composed of those elements of vector `a` for which `b` equals 5 or 8. Note the double = in the condition. Other conditions (also called logical or Boolean operators) are <, >, !=, <= and >= >=. To test more than one condition in one if-statement, use `&` if both conditions have to be met ("and") and `|` if at least one of the conditions has to be met ("or").

20 Repeated execution of commands

Repeated use of a command can be achieved by using one of the following commands:

- `repeat`{commands}:repeats the commands until it hits `break`
- `while`(condition){commands}: Repeats the commands as long as the condition is fulfilled
- `for` (range) {commands}: Repeats the commands for as many times as specified by range

These commands are manipulating loops:

- `next`: jumps right to the begin of the loop
- `break`: ends the loop

The exact syntax of these loop commands will be shown for the example of summing up all entries of a vector v.

With `repeat`:

```
 v <- 1:10          # An arbitrary vector
sumv <- 0
i <- 0
repeat {
i <- i + 1
sumv <- sumv + v[i]
if (i < length(v)) {next}
print(sumv)
break}
```

With `while`:

```
sumv <- 0 # Reset value
i <- 0 # Reset value value
while (i < length(v)) {
  i <- i + 1
  sumv <- sumv + v[i]
}
print(sumv)
```

With a `for` loop:

```
sumv <- 0                   # Set back value
for (i in seq(along = v)){sumv <-sumv + v[i]}
print(sumv)
```

For the `for`-loop, we used `seq(along = v)` instead of `1:length(v)` because of possible problems if `length(v)` would be zero. Summing up a vector is already implemented in R, just type `sum(v)`.

20.1 Exercise

For any numerical vector `v`, write a function/R-script that computes the sum of all positive and all negative values and shows the results on the screen. Try to use loops and/or conditions. Test it with several vectors. Do you have to use loops and/or conditions?

Remark: An output consisting of text and variables can be displayed by using `cat`. Try `?cat`. A small example:

```
x <- 42
cat("The answer is", x, "\n")
```

If you want to model a time series, you usually do the computations for one time step and then for the next and the next, etc. Because nobody wants to type the same commands over and over again, these computations are automated in for-loops.

In a for-loop you specify what has to be done and how many times. To tell "how many times", you specify a so-called counter. An example:

```
> h <- seq(from = 1, to = 8)
> s <- c()
> for (i in 2:10)
   {
     s[i] = h[i] * 10
   }
> s
[1] NA 20 30 40 50 60 70 80 NA NA
```

- First the vector h is made.
- In line 2 an empty vector (s) is created. This is necessary because when you introduce a variable within the for-loop, R will not remember it when it has gotten out of the for-loop.
- In line 3 the `for`-loop starts. In this case, `i` is the counter and runs from 2 to 10.
- Everything between the curly brackets (line 5) is processed 9 times. The first time `i=2`, the second element of h is multiplied with 10 and placed in the second position of the vector `s`. The second time `i = 3`, etc. In the last two runs, the 9th and 10th elements of h are requested, which do not exist. Note that these statements are evaluated without any explicit error messages.

20.2 Exercise

Make a vector from 1 to 100. Make a for-loop which runs through the whole vector. Multiply the elements which are smaller than 5 and larger than 90 with 10 and the other elements with 0.1.

21 Writing your own functions

Functions you program yourself work in the same way as pre-programmed R functions.

```
fun1 <- function(arg1, arg2 )
   {
   w = arg1 ^ 2
   return(arg2 + w)
   }
mod <- fun1(arg1 = 3, arg2 = 5)
mod
```

- In line 1 the function name (fun1) and its arguments (arg1 and arg2) are defined.
- Lines 2-5 specify what the function should do if it is called. The return value (arg2+w) is given as output. - In line 6 the function is called with arguments 3 and 5 and the answer is shown on the screen.
- In line 8 the answer is stored in the variable mod.

21.1 Exercise

Write a function for the previous exercise, so that you can feed it any vector you like (as argument). Use a for-loop in the function to do the computation with each element. Use the standard R function length in the specification of the counter.

22 OPTIONAL: Vector-oriented programming

If you want to apply a `function` on each entry of a vector, this can either be done by writing a loop as described in the last section or by using `sapply` (the latter being faster in most cases). For example, we want to square each entry of a numerical vector.

```
v <- 1:10
square <- function(x) {x^2}      # Define the square function
sapply(v, square)                # Arguments: vector/data frame, function
```

If used wth a data frame, the function is applied to each column of the data frame. To apply the same function to either rows or columns of a matrix,use `apply`. We will search for the maximum value in each row and each column of a matrix.

```
matrix3 <- matrix(1:25, nrow = 5)
matrix3                 # Computes the maximal values in each row
apply(matrix3, 1, max)
apply(matrix3, 2, max) # Computes the maximal values in each column
```

The second argument of `apply` specifies whether the function is applied to rows or columns. Often, we want to apply a function to a data frame, but only to all individuals/ rows that carry a certain experimental condition (= column value). This can be done with `tapply`. Let's imagine an experiment with conditions a and b. We want to compute the mean (command `mean`) for the values of each condition.

```
data3 <- data.frame(values = 1:10, condition = c(rep("a", 5), rep("b", 5)))
data3
attach(data3)
tapply(values, condition, mean)
detach(data3)
```

Reference section: Useful commands and functions in R

23 Data creation

- `read.table`: read a table from file. Arguments: `header=TRUE`: read first line as titles of the columns; `sep=","`: numbers are separated by commas; `skip = n`: don't read the first n lines.
- `write.table`: write a table to file
- `c`: paste numbers together to create a vector
- `array`: create a vector, Arguments: dim: length - matrix: create a matrix, Arguments: ncol and/or nrow: number of rows/columns
- `data.frame`: create a data frame
- `list`: create a list
- `rbind` and `cbind`: combine vectors into a matrix by row or column

24 Extracting data

- `x[n]`: the `n`-th element of a vector
- `x[m:n]`: the `m`-th to nth element
- `x[c(k,m,n)]`: specific elements
- `x[x>m & x<n]`: elements between `m` and `n - xn`: element of list or data frame named n - x[["n"]]: idem
- `[i,j]`: element at `i`-th row and `j`-th column - `[i,]`: row `i` in a matrix

25 Information on variables

- `length`: length of a vector
- `ncol` or `nrow`: number of columns or rows in a matrix
- `class`: class of a variable
- `names`: names of objects in a list
- `print`: show variable or character string on the screen (used in scripts or for-loops)
- `return`: show variable on the screen (used in functions)
- `is.na`: test if variable is `NA`
- `as.numeric` or `as.character`: change class to number or character string
- `strptime`: change class from character to datetime (POSIX)

26 Plotting

- `plot(x)`: plot `x` (y-axis) versus index number (x-axis) in a new window
- `plot(x,y)`: plot `y` (y-axis) versus `x` (x-axis) in a new window
- `image(x,y,z)`: plot `z` (color scale) versus `x` (x-axis) and `y` (y-axis) in a new window
- `lines` or `points`: add lines or points to a previous plot
- `hist`: plot histogram of the numbers in a vector
- `barplot`: bar plot of vector or data frame
- `contour(x,y,z)`: contour plot
- `abline`: draw line (segment). Arguments: `a,b` for intercept `a` and slope `b`; or `h = y` for horizontal line at `y`; or `v = x` for vertical line at `x`.
- `curve`: add function to plot. Needs to have an `x` in the expression. Example: `curve(x^2)`
- `legend`: add legend with given symbols (`lty` or `pch` and `col`) and text (`legend`) at location (`x = "topright"`)
- `axis`: add axis. Arguments: side – `1` =bottom, `2` = left, `3` = top, `4` = right
- `mtext`: add text on axis. Arguments: text (character string) and side
- `grid`: add grid
- `par`: plotting parameters to be specified before the plots. Arguments: e.g. `mfrow=c(1,3))`: number of figures per page (1 row, 3 columns); `new = TRUE`: draw plot over previous plot.

27 Plotting parameters

These can be added as arguments to plot, lines, image, etc. For help see par.

- `type:` "l"=lines, "p"=points, etc.
- `col`: color – "blue", "red", etc
- `lty`: line type – 1=solid, 2=dashed, etc.

- `pch`: point type – 1=circle, 2=triangle, etc.
- `main`: title character string
- `xlab` and `ylab`: axis labels – character string
- `xlim` and `ylim`: range of axes – e.g. c(1,10)
- `log`: logarithmic axis – “x”, “y” or “xy”

28 Statistics

- `sum`: sum of a vector (or matrix)
- `mean`: mean of a vector
- `sd`: standard deviation of a vector
- `max` or `min`: largest or smallest element
- `rowSums` (or `rowMeans`, `colSums` and `colMeans`): sums (or means) of all numbers in each row (or column) of a matrix. The result is a vector.
- `quantile(x,c(0.1,0.5))`: sample the 0.1 and 0.5th quantiles of vector `x`

29 Data processing

- `seq`: create a vector with equal steps between the numbers
- `rnorm`: create a vector with random numbers with normal distribution (other distributions are also available)
- `sort`: sort elements in increasing order
- `t`: transpose a matrix
- `aggregate(x,by=ls(y),FUN="mean")`: split data set `x` into subsets (defined by `y`) and computes means of the subsets. Result: a new list.
- `na.approx`: interpolate (in `zoo` package). Argument: vector with NAs. Result: vector without NAs.
- `cumsum`: cumulative sum. Result is a vector.
- `rollmean`: moving average (in the `zoo` package) - paste: paste character strings together
- `substr`: extract part of a character string

30 Fitting

- `lm(v1 ~ v2)`: linear fit (regression line) between vector `v1` on the y-axis and `v2` on the x-axis
- `nls(v1 ~ a + b * v2, start = ls(a = 1, b = 0))`: nonlinear fit. Should contain equation with variables (here `v1` and `v2` and parameters (here `a` and `b`) with starting values
- `coef`: returns coefficients from a fit
- `summary`: returns all results from a fit

31 Programming

- `function (arglist) {expr}`: function definition: do expr with list of arguments arglist
- `if (cond) {expr1} else {expr2}` : if-statement: if cond is true, then expr1, else expr2
- `for (var in vec) {expr}`: for-loop: the counter var runs through the vector vec and does expr each run
- `while (cond) {expr}`: while-loop: while cond is true, do expr each run

32 Keyboard shortcuts

There are several useful keyboard shortcuts for RStudio (see Help → Keyboard Shortcuts):

- `CRL+ENTER`: send commands from script window to command window
- `$\uparrow$` or ↓ in command window: previous or next command
- `CTRL+1`, `CTRL+2`, etc.: change between the windows

Not R-specific, but very useful keyboard shortcuts:

- `CTRL+C`, `CTRL+X` and `CTRL+V`: copy, cut and paste
- `ALT+TAB`: change to another program window - ↑ ↓ ← or → movecursor
- `HOME` or `END`: move cursor to begin or end of line
- `Page Up` or `Page Down`: move cursor one page up or down
- `SHIFT+↑/↓/←/→/HOME/END/PgUp/PgDn`: select

33 Error messages

- `No such file or directory` or `Cannot change working directory`: Make sure the working directory and file names are correct.
- `Object 'x' not found`: The variable x has not been defined yet. Define x or write apostrophes if x should be a character string.
- `Argument 'x' is missing without default`: You didn't specify the compulsory argument x.
- `+`: R is still busy with something or you forgot closing brackets. Wait, type } or) or press ESC.
- `Unexpected ')' in ")"` or `Unexpected '}' in "}"`: The opposite of the previous. You try to close something which hasn't been opened yet. Add opening brackets.
- `Unexpected 'else' in "else"`: Put the else of an if-statement on the same line as the last bracket of the "then"-part: `}else{`.
- `Missing value where TRUE/FALSE needed`: Something goes wrong in the condition-part (`if(x==1)`) of an if-statement. Is x `NA`?
- `The condition has length > 1 and only the first element will be used` In the condition-part (`if(x==1)`) of an if-statement, a vector is compared with a scalar. Is x a vector? Did you mean `x[i]`?
- `Non-numeric argument to binary operator`: You are trying to do computations with something which is not a number. Use `class(...)` to find out what went wrong or use `as.numeric(...)` to transform the variable to a number.
- `Argument is of length zero` or `Replacement is of length zero`: The variable in question is `NULL`, which means that it is empty, for example created by `c()`. Check the definition of the variable.

Further literature

There's an extensive list of books about R (with book descriptions) available at http://www.r-project.org/doc/bib/R-books.html. From our lab, the recommendations of books would be

- M. J. Crawley, "The R Book", Wiley %- M. J. Crawley, "Statistics: An Introduction using R", Wiley
- U. Ligges, "Programmieren mit R", Springer (in German, this course is based on it)

There are also some manuals available at the R-project website http://cran.r-project.org/.