# Getting to work in R

## Compendium for the course BIOL3159

Jon Brommer

# Instructions

This compendium consists of seven parts, indicated by Roman numerals. Each part consists of possibly multiple chapters and finishes with 5 exercises. During the course, we work through one part each week. Your task is to

1. Work through the part of the compendium that is the topic of the week. I would recommend to run and study the code provided rather than just read it.
2. Solve the exercises of the part of the compendium that is the topic of the week. The exercises may require additional files which are provided in Moodle for your use. If you have worked through the compendium and spent some time thinking about the code, it should be possible to solve each exercise. If you cannot solve it, do not panic. There is help in the clinic! It is also not needed to solve each and every exercise, but it is recommended that you make an effort so you can follow the presentation during the weekly meeting.
3. At the meeting, there is a list on which you are to indicate which exercise of that week you are willing to present. From those willing to present, a random student is chosen. Please see more detailed instructions on how to prepare for such a presentation with the exercises for part I. The student then shows the class how he/she solved the exercise and explains what the script does in own words. This demonstration is not a test. It is meant to demonstrate to others how the exercise was dealt with and provides a starting point for discussion. To pass the course, you must have indicated that you are willing to present 3/5th of all exercises. That is, you must have willing to present 21 of the 35 exercises. If you for whatever reason do not have the possibility to indicate you are willing to present, you can submit your solution of the exercises of that week at the latest just before the meeting starts via Moodle. Each solution of an exercise that produces the correct answer will count as one “willing to present”.
4. You are to complete this compendium by including the solution to EACH exercise. At the end of the course this completed compendium will be a reference guide primarily for your own records and use. The extent to which you want to complete the compendium (the level of detail) hence depends entirely on you. In any case, to pass the course you need to file your completed compendium via Moodle. Exercises you have not solved yourself, you can learn how to solve by attending the Wednesday meetings. Alternatively, you can self-organize with fellow students e.g. a study circle to manage the completion of your compendium.

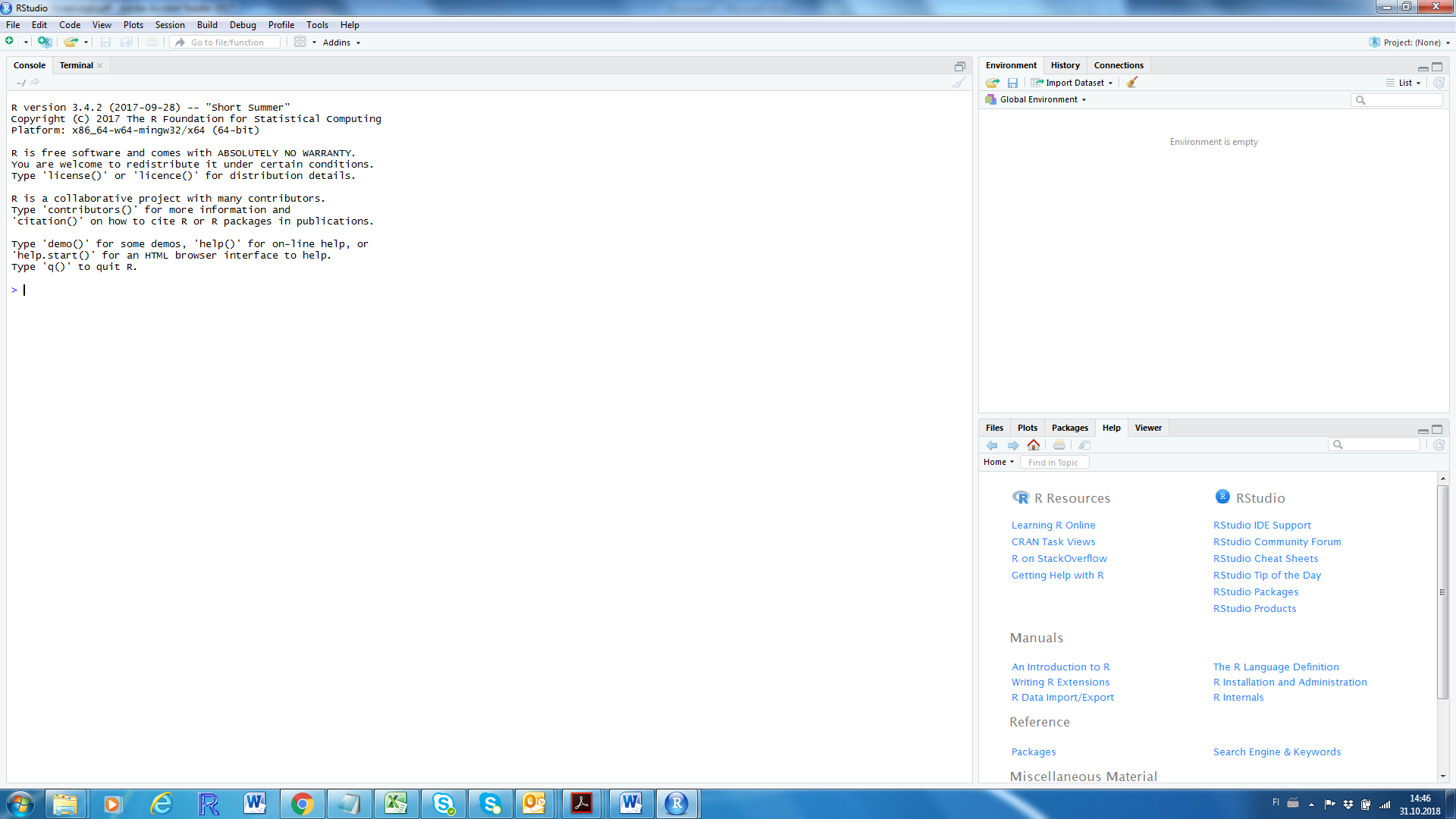
# 

# Part I: Getting started

## Chapter 1. First R session

### 1.1.1 Starting R

After installing the program R and R-studio, you can open R-studio (Fig. 1.1). R studio is a wrapper for R. That is, R Studio is a separate program running R within itself and hence requires the program R to be installed. When you open R Studio for the first time, it shows the R Console listing the version of R and its “nickname” as well as the ominous-sounding warning “R is free software and comes with ABSOLUTELY NO WARRANTY”.



*Fig. 1.1. R-studio after starting the program showing the panel with the R console on the left, and two additional panels on the right. The right-hand side panels are further subdivided into different tabs.*

There is also a cursor at the bottom blinking imposingly. Apparently waiting for you to start typing instructions. Why, you may ask, should I care to use such a program? One answer is the following line in the R console: “R is a collaborative project with many contributors”. Because R is open-source, there is a worldwide community of people continuously developing it. In particular, everyone can contribute by creating packages (also called libraries) which is a collection of programs written to solve specific tasks. It is this feature that makes R incredibly flexible. By installing specific packages you can customize R to use it e.g. as a Geographic Information System (GIS), an analyser of next-generation sequencing data, or software to carry out a specific type of statistical analyses. However, before you start on that adventure, it is worth investing time in getting to work in R on a more general level, which is what this course is about.

We can start by writing a first command by going to the console (click your mouse there) and write a command and R will execute it after you press ENTER

> print(“I will get to work in R”)

[1] “I will get to work in R”

Throughout, code as it appears in the Console panel is printed as above. The R command is behind the symbol > which is in default R-Studio and here printed in blue, and the line(s) without > denote what R prints on the Console (in black) in response to the R command.

You see from this first command that print() makes R print your text – what you wrote between quotation marks “” inside the brackets (), but R also places [1] in front of it. This denotes that R views your piece of text (called a string) as a vector.

You can make a vector of numbers using the function c() for *combine* by typing

> c(1,2,3)

[1] 1 2 3

The [1] is the indexing. If you type a very long series of numbers so that it would extend to a second row, R will show the index of the first element of the vector on the second row. We can make a vector of 40 by using the colon

> c(1:40)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

Here, the second row starts with element number 26 in the vector and R tells this to us by printing [26] on the second row. Where the row end will depend on the size of the window pane so you may get a different division than the one above.

Coding like this, directly in the command window, is not a good way forward. Although we can use the arrow pointing up on our keyboard to repeat the last commands, we would quickly get lost in the code. You can bundle your lines of code in a script. A script is simply a text file. By selecting with your mouse *File* 🡪 *New script* 🡪 *R script*, an additional panel opens on the left side where you can write your code. You can send your code to the command window by pressing CTRL + ENTER (in stand-alone R this is CTRL + R). There are 2 ways:

1. Place the cursor on a line in the script and press CTRL+R, which runs the entire line of code and moves the cursor to the next line with code
2. Highlight the code you want to run using your mouse and press CTRL+R, which runs only the selected piece of code, be it a subset of one line, the whole line or several lines

Try it by writing in your new script c(1:40) and running the whole line (option 1), and by running only a highlighted 1:40 to get in the R console the output

> c(1:40)

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

> 1:40

[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40

We can complete this first script by adding the print command listed above and annotate it. Annotation in R scripts means that you add comments after the hashtag symbol # which R will ignore. R Studio will skip lines that are annotated when pressing CTRL+ENTER. Although the level of details in your annotation is a personal decision, it is often better to annotate abundantly, especially when getting started. Throughout, R script is here shown with an indent and with annotations in red and code in black (without the > in front) with strings in green. This is to separate script (what is in the text file) from what is printed on the console after running lines in your script. Your script could look something like

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# first R script

# NAME

# DATE

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# R can print on the command window by using print

# I can send the line below to the command window by pressing CTRL + ENTER

print(“I will get to work in R”)

# A vector in R is made using c()

c(1,2,3)

# I can also make a vector using :

1:40

You can save this script. For example naming it MyFirstScript. RStudio will automatically give it the extension .R so its full name will be “MyFirstScript.R”. Close your script, and open it again. The script will be the same and it will produce the same result when you run the command lines again. That is, the script presents a reproducible and complete record of a set of instructions, a short program.

### 1.1.2 Objects

Let’s make another vector consisting of the number 1 to 4, c(1,2,3,4) or 1:4 and send it to R’s console by pressing CTRL+ENTER. We get

> c(1,2,3,4)

[1] 1 2 3 4

but, this vector is not stored into R’s “brain”. To store information, you need to assign it to an object. The assign operator is <-. For example,

> myVector<-c(1,2,3,4)

The vector you created will not be printed in the command window. R only shows the command you just ran and will now simply wait for the next command showing > followed by a blinking cursor. Second, R now remembers the object you just created and named myVector. This name is arbitrary and can be anything. You should, however, name your objects as informative as possible. So not var1, var2, or similar whose meaning you are likely to forget rapidly.

Any object stored in R’s brain, can be called back, try this by running the name of the object as the command

> myVector

[1] 1 2 3 4

R then returns the vector you defined previously in this script. In RStudio, all objects are listed in the top-right hand panel (under the tab “Environment”).

Let’s make a second object by typing in the script and then running the lines

> myMatrix<-matrix(c(1,2,3,4),2,2)

> myMatrix

[,1] [,2]

[1,] 1 3

[2,] 2 4

Note that R Studio displays this matrix in the Environment (top right-hand panel) under a different heading than the vector created earlier. R Studio places matrix under “data”, and a vector under “values”. You can click on any object listed under “data” and R-studio will show it to you in a separate tab. You can also call the object in R (e.g. by using your mouse to highlighting “myMatrix” in your script and press CTRL+ENTER as explained above). R returns:

[,1] [,2]

[1,] 1 3

[2,] 2 4

Again, R here shows the indexing in square brackets, but because this is a matrix it shows the indices of both rows and columns. Thus R indexes [row,column]. Clearly, R has constructed the matrix column-wise: Numbers 1 and 2 in the first column [,1] and the numbers 3 and 4 in the second one [,2].

A vector or matrix need not be numerical. It can contain strings. A string is denoted between quotes “”. For example, “a1” would denote a string consisting of the letter “a” and the number “1”. To make a vector of strings, called a character vector, you can use the combine function c() as for numerical vectors.

> myCharacterVector<-c(“a”,”b”,”c”,”d”)

A vector and matrix must be EITHER numerical OR character, not both. To see this, try

> c(“a”,2)

[1] “a” “2”

R will thus coerce (force) the number 2 into character “2” as seen from the quotation marks and the resulting vector will hence be a character vector (as evidenced by the quotation marks).

R allows you to assign names instead of the numbers to the indexing of objects, using the function names(), or (for matrix()) colnames() and rownames().

> names(myCharacterVector)<-c(“letter.1”,”letter.2”,”letter.3”,”letter.4”)

> myCharacterVector

letter.1 letter.2 letter.3 letter.4

“a” “b” “c” “d”

> colnames(myMatrix)<-c(“col1”,”col2”)

> rownames(myMatrix)<-c(“row1”,”row2”)

> myMatrix

col1 col2

row1 1 3

row2 2 4

Naming the elements allows you therefore an alternative to numbers for the “coordinates” within each object. In the next section, we explore how we can extract information from objects.

### 1.1.3 Indexing

Because R now has the objects in its memory, you can access the object, as you saw. But, you can also access elements within each object. For the vector and matrix, you use the index numbers which you place between square brackets []. For example the command

> myVector[1]

[1] 1

returns the first element in the object MyVector

You can use a vector to index multiple elements in this vector (REMEMBER: multiple elements in R always are combined in a vector using the c() function)

> myVector[c(1,4)]

[1] 1 4

Similarly, you can index a matrix, but now you need to index both row and column

> myMatrix[1,2]

[1] 3

returns the matrix element in the first row and second column. Names can also be used for the indexing. The above command is equivalent to

> myMatrix[”row1”,”col2”]

[1] 3

Although indexing using numbers and names are in the above example practically equivalent, using names will in many cases be more intuitive.

By using the minus sign some part of the object can be removed. For example

> myMatrix[-2,]

[1] 2 4

### 1.2 Calculate something

You store objects in R’s memory because you want to use the information. All standard calculations you can do work of course in R, such as, for example, multiplication (\*), division (/), squaring (^2) and square root (sqrt)

> myVector\*3

[1] 3 6 9 12

> myMatrix\*3

col1 col2

row1 3 9

row2 6 12

> myVector/2

[1] 0.5 1.0 1.5 2.0

> myVector^2

[1] 1 4 9 16

> sqrt(myMatrix)

col1 col2

row1 1.000000 1.732051

row2 1.414214 2.000000

Clearly, R applies the desired operation on every element of the vector or matrix.

R has a plethora of functions to calculate properties of your objects. For example, the function mean().

> mean(myVector)

[1] 2.5

> mean(myMatrix)

[1] 2.5

Again, just because you asked R to calculate something does not mean R remembers it. For R to remember the mean value of myMatrix you have to assign it to an object.

> meanMyMatrix<-mean(myMatrix)

This object now holds the mean of all the values in myMatrix. To get this value, run the command

> meanMyMatrix

[1] 2.5

Alternatively, check the Environment in the top right-hand panel which shows this value.

R uses the brackets () exclusively in functions. Between the brackets you may include what is called an **argument** to the function. If you do not give an argument, you may get an error message

> mean()

Error in mean.default() : argument “x” is missing, with no default

In order to distinguish functions from objects, we here always include the brackets. Hence, mean() denotes the function and mean a (not-so-cleverly-named) object.

### 1.3 Class

Everything on the left-hand side of the assign operator <- is an object. You can easily create objects, as demonstrated above. These objects have a certain class. You can ask R about the class of your object using class().

> class(myVector)

[1] “numeric”

> class(myCharacterVector)

[1] “character”

> class(myMatrix)

[1] “matrix”

From the above, you can see that R distinguishes two types of vectors: numeric and character vectors.

A function in R is also an object (of class “function”).

> class(mean)

[1] “function”

R is an object oriented programming language. That is, for many purposes, the class of the object defines what R does when giving a command. Later on there are examples illustrating this aspect. For now, it is sufficient to remember that objects can be of different class.

### 1.4 Logicals

R uses logicals TRUE and FALSE. For example, the logical vector stating for each element in myVector whether it is true or false that the respective element is larger than 2.

> myVector

[1] 1 2 3 4

> myVector>2

[1] FALSE FALSE TRUE TRUE

That is, it is TRUE only for elements 3 and 4 in myVector that these are larger than 2. Similarly, for a matrix

> myMatrix

col1 col2

row1 1 3

row2 2 4

> myMatrix>2

col1 col2

row1 FALSE TRUE

row2 FALSE TRUE

Internally, logicals are viewed by R as binary with TRUE equal to 1 and FALSE equal to 0. Because of this we can perform slightly counter-intuitive calculations using logicals, such as summation

> sum(myVector>2)

[1] 2

> sum(myMatrix>2)

[1] 2

The above thus shows that there are two elements in both myVector and myMatrix for which it is TRUE that they are larger than 2.

### 1.5 NA, NaN, NULL

Apart from numbers and strings, R recognizes missing values as Not Available (NA). For example, a vector with the third element a missing value

> vector.OneNA<-c(1,2,NA,4)

> vector.OneNA

[1] 1 2 NA 4

Whenever you calculate with an NA value, the result itself is - of course - NA

> vector.OneNA[2]\*2

[1] 4

> vector.OneNA[3]\*2

[1] NA

NA (missing value) should not be confused with NaN (Not a Number)

> sqrt(-1)

[1] NaN

Warning message:

In sqrt(-1) : NaNs produced

Note that R here generates a warning, while still giving an answer.

Again, if an element is NA, a logical “answer” to a question cannot be produced, and hence the “answer” is NA

> vector.OneNA>2

[1] FALSE FALSE NA TRUE

In addition to NA, R also recognizes NULL as something which is not defined. NULL is not TRUE nor FALSE, and NULL is fundamentally different from a missing value (NA). NULL is ignored by R when constructing a vector

> c(1,2,3,NA)

[1] 1 2 3 NA

> c(1,2,3,NULL)

[1] 1 2 3

We have in section 1.1.2 named the elements in myCharacterVector but not in myVector. When we ask R to give us the names of these vectors we get

> names(myCharacterVector)

[1] “letter.1” “letter.2” “letter.3” “letter.4”

> names(myVector)

NULL

For the latter case, the names are not defined, hence NULL.

## Chapter 2. Getting help in using R

### 2.1 R’s own help on functions

R contains many functions with many details; most people’s brains cannot remember all details. To get help we can use the function ?

> ?mean

In R Studio, the help page will appear in the bottom right-hand corner. You can also use the search function inside this help tab in R Studio. All help files in R follow the same structure and logic. R’s own help is not always easy to follow and it takes time and experience getting used to. However, “decoding” the help files is a crucial skill in better understanding R.

Let’s go through this one in some detail

|  |  |
| --- | --- |
| mean {base} | R Documentation |

**Arithmetic Mean**

The top part tells us what we are looking at: The name of the function (mean) and the package it belongs to inside curly brackets (base), as well as a short description of what the function does (calculation of the arithmetic mean):

This is followed by a description explaining shortly what the function is for

**Description**

Generic function for the (trimmed) arithmetic mean.

Then comes an important part, the usage of the function.

**Usage**

mean(x, ...)

## Default S3 method:

mean(x, trim = 0, na.rm = FALSE, ...)

What is mentioned under “Usage” between brackets () are the ***arguments***. The first line under “Usage” lists two arguments, x and … Then it continuous with a technically-sounding Default S3 method listing 4 arguments (x, trim, na.rm and …). We can ignore the “S3” aspect for the time being and focus on the two arguments defined under “Usage” with an equal to “=” (i.e., trim=0 and na.rm=FALSE). What is given here are the ***default*** values of these two arguments. That is, if the user does not specify these arguments, R will assume the default values given here are valid. The next section explain in more detail what these arguments are

**Arguments**

|  |  |
| --- | --- |
| x | An **R** object. Currently there are methods for numeric/logical vectors and [date](http://127.0.0.1:19885/library/base/help/Dates), [date-time](http://127.0.0.1:19885/library/base/help/date-time) and [time interval](http://127.0.0.1:19885/library/base/help/time%20interval) objects. Complex vectors are allowed for trim = 0, only. |
| trim | the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint. |
| na.rm | a logical value indicating whether NA values should be stripped before the computation proceeds. |
| ... | further arguments passed to or from other methods. |

Clearly, the first argument passed to the function mean() is an object containing the information for which the arithmetic mean is to be computed. The other arguments allow the user to some extent filter the information prior to calculation of the mean. This filtering is done either by “trimming” the extreme values, or by removing the NA values.

The help continuous explaining more about the arguments, under the heading “Value”

**Value**

If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA\_real\_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.

From this “Value” section we thus learn more about the use of trim. The help page continues with a reference

**References**

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) The New S Language. Wadsworth & Brooks/Cole.

Most of the base functions are based on the S language out of which R evolved. The help page suggests, if relevant, other functions, which may be of interest as they are similar. These other functions are hyperlinks and clicking on them brings you to the respective help page

**See Also**

[weighted.mean](http://127.0.0.1:19885/library/base/help/weighted.mean), [mean.POSIXct](http://127.0.0.1:19885/library/base/help/mean.POSIXct), [colMeans](http://127.0.0.1:19885/library/base/help/colMeans) for row and column means.

Lastly, each help page gives an example of the function in use.

**Examples**

x <- c(0:10, 50)

xm <- mean(x)

c(xm, mean(x, trim = 0.10))

The bottom of the help page displays the R version you are using, and which package the function belongs. The hyperlink “Index” brings you to a help page listing of all functions in that package allowing browsing which is often useful to visit

[Package base version 3.4.0 [Index](http://127.0.0.1:19885/library/base/html/00Index.html)]

### 2.2 Running the example on the help page

R has the function example() to allow you to quickly run the lines of code listed under the “Examples” heading on the Help page. For example, we can run the lines given in the help of the function mean() by using example().

> example(mean)

mean> x <- c(0:10, 50)

mean> xm <- mean(x)

mean> c(xm, mean(x, trim = 0.10))

[1] 8.75 5.50

In the above, R signals it is running the example of mean() by displaying mean> instead of > at the start of each line. Illustrated here are use of the function mean() with the default value of trim (i.e. mean(x)) and the use of this function overriding the default by specifying the argument trim = 0.10.

### 2.3 Interpreting the help page further

We have now studied the help page. To calculate the arithmetic mean of myVector we can thus give the command

> mean(myVector)

[1] 2.5

A more complicated situation arises when we try the calculate the mean of a vector with a missing value

> Vector.OneNA<-c(1,2,NA,4)

> mean(Vector.OneNA)

[1] NA

Intuitively, this makes sense. As there is one value which is missing (unknown), R tells us the mean is unknown. However, we likely also want to calculate the mean of the value ignoring the missing values. Consulting the help for mean() lists under “Arguments”

|  |  |
| --- | --- |
| na.rm | a logical value indicating whether NA values should be stripped before the computation proceeds. |

We furthermore see from “Usage” the default of this argument

mean(x, trim = 0, na.rm = FALSE, ...)

Thus, calling the function mean() will by default not strip the NA values before computation. Consequently, we are to override the default by specifying the alternative na.rm = TRUE (na.rm stands for NA remove and is an argument also used in other functions).

> mean(Vector.OneNA,na.rm=TRUE)

[1] 2.333333

### 2.4 Search for help on topic

Direct searching for a function requires that you know what you need. A wider search for help in stand-alone R is requested by two question marks. For example ??mean will bring up a page with lots of different help pages with the search term in them. Items are displayed as hyperlinks to the specific page and a short description. Depending on the R installation the help on a topic will differ, but it likely contains

|  |  |  |
| --- | --- | --- |
| [base::colSums](http://127.0.0.1:19885/library/base/html/colSums.html) |  | Form Row and Column Sums and Means |
| [base::Date](http://127.0.0.1:19885/library/base/html/Date.html) |  | Date Class |
| [base::DateTimeClasses](http://127.0.0.1:19885/library/base/html/DateTimeClasses.html) |  | Date-Time Classes |
| [base::difftime](http://127.0.0.1:19885/library/base/html/difftime.html) |  | Time Intervals / Differences |
| [base::mean](http://127.0.0.1:19885/library/base/html/mean.html) |  | Arithmetic Mean |
| [boot::sunspot](http://127.0.0.1:19885/library/boot/html/sunspot.html) |  | Annual Mean Sunspot Numbers |

where for each item found first the package is listed (base or boot) followed by double colon and then the name of the function in that package. General help on a topic is also obtained by searching in R-Studio in the “Help” panel (bottom-right panel). However, the above example does not translate as it will produce the help for the function mean(). However, a search for the term “linear” will be equivalent to the command ??linear in the console.

Because R is open source software, there is plenty of help on the internet. One good search engine is RSeek (<https://rseek.org/>) which is a customised Google search to deliver only content that relates to R. Try, for example, a search in RSeek for “mean”. It will produce a wealth of hits including tutorials, blogs and other content where the search term is displayed.

A short note on internet help is perhaps at place here. Open source software and help for using it is great. A downside is that there is often so much information that you can spend a long time reading and not necessarily getting anywhere. I recommend spending time with the help in R first. Carefully reading and trying to understand the R help. Study the examples. Try your problem and then if it does not work or is not the same use the information in help to figure out why, as exemplified above for the function mean() when there are missing values. As you use R more you get better in understanding the help R gives. You can complement the R help with R-seek in case there are details you need more info on.

A second point is that you can do a lot with the standard (base) installation of R. As mentioned above, R’s flexibility relies on the user being able to customize R by loading on top of the base R installation different additional packages, each containing many different functions and other items. You will find many internet pages starting with loading the author’s favorite package for a particular problem even when the base installation of R has sufficient functions for that problem. This may be fine for the person authoring the page, but will require especially a novice user to spend time acquainting with different packages. A good basis in R and understanding of R’s own help pages will avoid this. You can do much more in base R than most internet help pages suggest. For this reason, all exercises can be solved using base R unless stated otherwise.

## Chapter 3. Getting a data file into R’s brain

### 3.1 Getting and setting your working directory

At this point, you have opened a new script and written some commands (section 1.1.1). This script is now saved somewhere on your computer. Your script is stored in a folder (also called directory) on your computer or a server. This directory may or may not be where R would look by default. R has a default path (address) to its so-called working directory. The path is the address to the place on your computer or on a server where R will read and write information from and to. For example, if you want to import data, R will look for this data file in its working directory. Hence, the first thing to do is to tell R where to look.

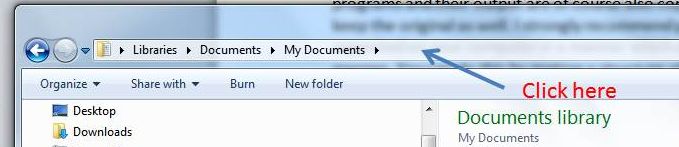
You get to know the working directory by calling the function getwd() (where wd stands for working directory) which produces, formally, a character vector with one element, or in other words a string. For example,

> getwd()

[1] “C:/Users/jonbro/Documents”

The path name may also be something like “//utuhome.utu.fi/jonbro” which would be a server address on my home university. In R, this is the *working directory* in which you are at present. You can change the working directory using setwd() which requires a path name as a string similarly as the one obtained above from getwd().

There are a number of ways to get the path name of directory you want use as working directory. The one I use: When you in Windows explorer go to the folder you want to go to and click on the bar in the top, it shows the path, which you can copy-paste into your script



*Figure 3.1.1: Standard view of your folder in windows explorer*



*Figure 3.1.2: After clicking in the top bar, it shows the path to your directory*

You can then use the R command setwd() to change the directory. There is one complication, which is that R read “\..” as a special character. For example, “\t” is ASCII code for a tab. Thus, the above path “C:\Users\joegbr\Documents” will not be understood by R. You get an error message when you try

> setwd(« C:\Users\jonbro\Documents »)

Error: ‘\U’ used without hex digits in character string starting “”C:\U”

You either need to change the “\” to “\\” or to change it to “/”. That is

> setwd(“C:\\Users\\jonbro\\Documents”)

works fine. Also

> setwd(“C:/Users/jonbro/Documents”)

will work. Following the same logic, a location on a server can be specified. For example, at my work place, the address to the server ”[\\utuhome.utu.fi\jonbro“ will become the path name string by doubling each backward slash ”\\\\utuhome.utu.fi\\jonbr0”](file:///\\utuhome.utu.fi\jonbro%20) or equivalently by replacing backward to forward slash “//utuhome.utu.fi/jonbro”. Note that R itself will display path names using forward slashes but will accept both forward slash (/) and double backward slash (\\) as equivalents.

> setwd(« C:\\Users\\jonbro\\Documents »)

> getwd()

[1] C:/Users/jonbro/Documents

The argument to setwd()is a character vector with one element (that is, it is given within quotation marks “”) and recognized by R as a piece of text. We can therefore also assign it to an object

> MyDirectory <- “//utuhome.utu.fi/jonbro”

and use the command setwd(MyDirectory) to set the working directory. In this case we give the object MyDirectory which contains the string with the path and thus do not write setwd(“MyDirectory”). In the latter case, R would interpret the path to be “MyDirectory” which will produce an error.

Working from a cloud server is typically a clever idea. If you work from an own computer, you can install a client to the cloud server on your computer. The client will synchronize a directory on your computer to the cloud server directory and vice versa (you can typically choose which directories are to be synchronized). For example, suppose I want to use Dropbox. Using a Dropbox client I can create on my own computer a folder to be kept up-to-date with my Dropbox account. This folder is called, say, “C:\Users\jonbro\Dropbox” and I create a folder “GettingToWorkInR” on my Dropbox. I can now access this directory from R by setting the path

> setwd(« C:\\Users\\jonbro\\Dropbox\\GettingToWorkInR »)

As this is a directory on a cloud server, I can access it from anywhere and from multiple devices (e.g. using a Dropbox app on my mobile phone). Moreover, I can share this cloud server directory with other person(s) and – if I give them editing rights – these other persons can both read, run and edit these files.

### 3.2 Create a data file in your working directory

Let’s make a data file and save it. R is good for many things, except perhaps data entering. Open Excel or your favorite spreadsheet program and type in some data. Let’s say we have (in columns) individual A,B,C, ... J , with color either red or blue (about half), mass (varying around 1-2 , with 0.1 accuracy), length (varying around 10, in 1 accuracy). You can make and save this file as a CSV file (which stands for “comma separated values” file). Save the data as “MyData.csv” in your working directory.

|  |  |  |  |
| --- | --- | --- | --- |
| individual | colour | Mass | Length |
| A | blue | 1.0 | 10 |
| B | blue | 1.2 | 9 |
| C | red | 1.7 | 5 |
| D | blue | 1.4 | 9 |
| E | blue | 1.5 | 12 |
| F | red | 1.1 | 13 |
| G | red | 1.9 | 12 |
| H | blue | 1.5 | 7 |
| I | blue | 1.3 | 8 |
| J | red | 1.4 | 10 |

### 3.3 Read in the data file from your working directory

We start a new script that reads in the data-file we have made into R. Remember that for R to remember anything, you have to assign it to a named object, e.g. myData. Further, remember to annotate. Thus, the start could look like:

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# R script for reading in datafile

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# switching to the directory my data is in

# the line below requires your directory

setwd(“include-your-directory-here”)

# read in the data

myData<-

We now need to find out how to “import data” or how to “read a data file”. At our disposal we have, for example, the R reference card (<https://cran.r-project.org/doc/contrib/Short-refcard.pdf>) or Rseek.org or we can run a general search in R (??import or ??read). You should try these options. They should result into your finding the function read.table(). The “Usage” part of the help for this functions shows

**Usage**

read.table(file, header = FALSE, sep = "", quote = "\"'",

dec = ".", numerals = c("allow.loss", "warn.loss", "no.loss"),

row.names, col.names, as.is = !stringsAsFactors,

na.strings = "NA", colClasses = NA, nrows = -1,

skip = 0, check.names = TRUE, fill = !blank.lines.skip,

strip.white = FALSE, blank.lines.skip = TRUE,

comment.char = "#",

allowEscapes = FALSE, flush = FALSE,

stringsAsFactors = default.stringsAsFactors(),

fileEncoding = "", encoding = "unknown", text, skipNul = FALSE)

read.csv(file, header = TRUE, sep = ",", quote = "\"",

dec = ".", fill = TRUE, comment.char = "", ...)

read.csv2(file, header = TRUE, sep = ";", quote = "\"",

dec = ",", fill = TRUE, comment.char = "", ...)

read.delim(file, header = TRUE, sep = "\t", quote = "\"",

dec = ".", fill = TRUE, comment.char = "", ...)

read.delim2(file, header = TRUE, sep = "\t", quote = "\"",

dec = ",", fill = TRUE, comment.char = "", ...)

This help page lists a number of functions which are related under the same heading. This is another, fairly typical, feature of help in R. Let’s focus on the first three arguments (file, header, sep) and look at what the “Arguments” section of this help page says about these three arguments:

**Arguments**

|  |  |
| --- | --- |
| file | the name of the file which the data are to be read from. Each row of the table appears as one line of the file. If it does not contain an absolute path, the file name is relative to the current working directory, [getwd](http://127.0.0.1:19885/library/utils/help/getwd)(). Tilde-expansion is performed where supported. This can be a compressed file (see [file](http://127.0.0.1:19885/library/utils/help/file)).  Alternatively, file can be a readable text-mode [connection](http://127.0.0.1:19885/library/utils/help/connection) (which will be opened for reading if necessary, and if so [close](http://127.0.0.1:19885/library/utils/help/close)d (and hence destroyed) at the end of the function call). (If [stdin](http://127.0.0.1:19885/library/utils/help/stdin)() is used, the prompts for lines may be somewhat confusing. Terminate input with a blank line or an EOF signal, Ctrl-D on Unix and Ctrl-Z on Windows. Any pushback on stdin() will be cleared before return.)  file can also be a complete URL. (For the supported URL schemes, see the ‘URLs’ section of the help for [url](http://127.0.0.1:19885/library/utils/help/url).) |
| header | a logical value indicating whether the file contains the names of the variables as its first line. If missing, the value is determined from the file format: header is set to TRUE if and only if the first row contains one fewer field than the number of columns. |
| sep | the field separator character. Values on each line of the file are separated by this character. If sep = "" (the default for read.table) the separator is ‘white space’, that is one or more spaces, tabs, newlines or carriage returns. |

The first argument file is clearly the name of the file. It can either contain the whole path or the path to the file is assumed to be the path to your working directory. This argument would hence be the filename “MyData.csv” for each of the variants of read.table().

The second argument header is a logical, whose default clearly is FALSE for read.table() (i.e. first line in the datafile is not taken to be the header) but has default value TRUE for the other variants.

The third argument specified the character used to separate field, or – in other words - the columns in the datafile. This now depends on your locale and your settings. In a comma-separated file (CSV), fields can be separated by commas (,) which is the default for the function read.csv(), but alternatively the separator is a semicolon (;) which is the default for read.csv2().

Further hints are provided by realizing that the argument dec (character used for decimal points) is a dot (.) for read.csv() and a comma (,) for read.csv2(). Clearly, if your Excel uses decimals commas, it must use something else as a separator. To be really sure, you can open the comma-separated file in a text editor which shows the separator. In windows, you can e.g. do this by right-clicking the file in Windows Explorer and selecting “Open with” and then Notepad or similar.

Having studied the help this far, it is clear that there are a number of options. These are:

1. Use read.table() and override the defaults to fit your need
2. Use the variants listed below read-table() on the help page which has the default specification you need

For example, for a Finnish local where the separator is a semicolon and decimal commas are used, the above translates into the following two options that are equivalent:

> myData<-read.table(“MyData.csv”, header=TRUE, sep=”;”,dec=”,”)

> myData<-read.csv2(“MyData.csv”)

Clearly, depending on how your text file is constructed, the above may need to be adjusted. For example, your text file may be comma separated and decimal dots are used in which case

> myData<-read.table(“MyData.csv”, header=TRUE, sep=”,”,dec=”.”)

> myData<-read.csv (“MyData.csv”)

would be correct. Nevertheless, the “long” option of using read.table() and its shorter variant as listed on the help page are equivalent. They both result in reading in the data and storing it in R’s brain. Which option you choose to use is up to you.

### 3.4 Quick check that the data import was successful

We should now have the data read into R; our updated script is

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# R script for reading in datafile

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# switching to the directory my data is in

# the line below requires your directory

setwd(“include-your-directory-here”)

# read in the data

# below the “long” version is used for a Finnish locale

# and the line below may need to be altered for your locale (see 3.3)

myData<- read.table(“MyData.csv”, header=TRUE, sep=”;”,dec=”,”)

# quickly check the data is read correctly

It is always a good idea to check the import of information was carried out successfully so we can be sure that any issues arising later on in the workflow are not due to erroneously reading in the data. The object myData should now appear in the Environment (top-right hand panel) in R Studio. It should state “10 obs. of 4 variables” because the data has 4 variables (individual, colour, mass and length) and for each of these there are 10 data points (i.e. 10 lines of data). Double clicking object myData in the Environment will open a separate tab in the top left-hand panel (same as where your script is) with your data displayed correctly. This is clearly a quick way to check that your data import has been successful. You can also call the object and check it in the command panel, as we did before

> myData

individual colour mass length

1 A blue 1.0 10

2 B blue 1.2 9

3 C red 1.7 5

4 D blue 1.4 9

5 E blue 1.5 12

6 F red 1.1 13

7 G red 1.9 12

8 H blue 1.5 7

9 I blue 1.3 8

10 J red 1.4 10

The function summary() provides a convenient summary

> summary(myData)

individual colour mass length

Length:10 Length:10 Min. :1.000 Min. : 5.00

Class :character Class :character 1st Qu.:1.225 1st Qu.: 8.25

Mode :character Mode :character Median :1.400 Median : 9.50

Mean :1.400 Mean : 9.50

3rd Qu.:1.500 3rd Qu.:11.50

Max. :1.900 Max. :13.00

We can see the summary statistics differ for the four variables which are named; hence, R has read the headers correctly. For individual and colour, summary() shows that R interprets these string variables as characters (i.e. text – “blue”, “blue”, “red”, …). For mass and length, summary() provides counts of the different entries and for mass and length a summary statistic. A quick check of minimum, maximum and mean values should convince you the reading of the data was done correctly.

Another convenient approach is to ask for the column headings

> names(myData)

[1] “individual” “colour” “mass” “length”

which returns a character vector we can copy-paste into the script as an annotation which aids in remembering the column headings in the data we read. Our script now looks something like

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# R script for reading in datafile

#¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤¤

# switching to the directory my data is in

# the line below requires your directory

setwd(“include-your-directory-here”)

# read in the data

# below the “long” version is used for a Finnish locale

# and the line below may need to be altered for your locale (see 3.3)

myData<- read.table(“MyData.csv”, header=TRUE, sep=”;”,dec=”,”)

# quickly check the data is read correctly

summary(myData)

names(myData)

#[1] “individual” “colour” “mass” “length”

Further useful functions are head() and tail() which provide the first and last 6 lines in the data respectively. These are of course handy when the data file is large.

### 3.5 Reading data with strings coded as factors

From R version 4 onwards, the default when reading in data is that alphanumeric data (data containing at least one character) are interpreted as characters. For example, the variables individual and colour in MyData are read as characters. For some analyses, including several statistical applications, your alphanumeric data likely refers to a factorial variable. You can already inform R that these variables are factors by specifying stringsAsFactor=TRUE when reading in your data. For example,

> myDataF<-read.table(“MyData.csv”, header=TRUE, sep=”,”,dec=”.”, stringsAsFactors = TRUE)

> summary(myDataF)

individual colour mass length

A :1 blue:6 Min. :1.000 Min. : 5.00

B :1 red :4 1st Qu.:1.225 1st Qu.: 8.25

C :1 Median :1.400 Median : 9.50

D :1 Mean :1.400 Mean : 9.50

E :1 3rd Qu.:1.500 3rd Qu.:11.50

F :1 Max. :1.900 Max. :13.00

(Other):4

Now, the variables individual and colour are interpreted as factors. The summary() shows for a factorial variable how many times each level occurs in the data. Thus, individual A, B,C,..F each occur one time in the data. For colour, the reported output states 6 cases of blue and 4 of red. Here, again, is a good way to check the data; do the reported numbers make sense to you. In particular, any typos are quick to spot. For example, perhaps you have mistakenly typed the colour for one case as “blu”. It would then show up in the summary output as a third level which would be clearly erroneous.

Similarly, the description provided by str()

> str(myDataF)

'data.frame': 10 obs. of 4 variables:

$ individual: Factor w/ 10 levels "A","B","C","D",..: 1 2 3 4 5 6 7 8 9 10

$ colour : Factor w/ 2 levels "blue","red": 1 1 2 1 1 2 2 1 1 2

$ mass : num 1 1.2 1.7 1.4 1.5 1.1 1.9 1.5 1.3 1.4

$ length : int 10 9 5 9 12 13 12 7 8 10

shows that individual and colour are interpreted as a factor. There are 10 levels for individual: these are now listed such that their level is listed first and then (after a second colon) we here see how R codes the data. Thus individual “A” is coded 1, “B” 2, etc. Similarly for colour there are two levels (“blue” and “red”) and R has coded the first three cases of colour (blue, blue, red) as 1, 1, 2.

### 3.6 Alternative ways to read in data; some aspects of coding to keep in mind

Reading in data into R is a pretty basic feature. The above illustrated that using code this can already be accomplished in different, although highly related manners. Different scripts accomplishing the same result is a feature of programming and of R. The more complicated the problem, the likely it is there are many different ways to solve it. At this point, your first priority is to get to work in R, to get a working script, a script you understand what it does and which you can adjust for other challenges and work further with. As such, therefore, there is no single approach which is superior. Rather, solutions will likely be individual. Once you have a solid base, it becomes interesting to read other people’s script and solutions, learn and take onboard other approaches.

R Studio allows you to import data using its drop-down menu (*File -> Import dataset* or “Import dataset” from the Environment panel). R Studio can import data from multiple formats, including Excel. Doing so, however, leaves no “trace” in your script. It will make your script reliant on manual actions, and not completely reproducible. You have to, for example, remember exactly which file contains your data. Manual data importing will also undermine the reproducibility of your script. A script is reproducible it running your script in its entirety produces the same outcome on each machine without further input. A fully reproducible script will run after opening the script, pressing CTRL+A (i.e. select all) and CTRL+ENTER (running each line of code). To have the “best of both” you can import data using R Studio’s built-in “import dataset”, which will produce the code in the command window and you can then copy-paste this line to your script to make it reproducible. R Studio will include the complete path as the file name.

By browsing the web, you likely will come across snippets of R code allowing you to directly import an Excel worksheet. Thus, one does not need to save data as a comma-separated file as we did in the above. In general, however, storing data in a simple, text-based file format (such as CSV) is to be preferred. CSV and other text files will work across platforms as they can be read by all software without ambiguity. Long-term data storage is always best in simple formats such as text files.

## Chapter 4. Checking your data

### 4.1 Object data.frame

When you read in data, R stores it as an object of class data.frame which is a 2 dimensional (rows and columns) collection of numbers **and/or** character strings **and/or** factorial variables. That is, a data.frame can contain a mix of different types of variables. This makes it useful for classic data tables because a matrix or a vector can only contain one type of variable (either a number OR a character) (section 1.1.2). R Studio will in its Environment panel list data.frame objects under the heading “Data” where also e.g. objects of class matrix appear. By clicking the data.frame object in R Studio it will be show the data.frame in a new tab in the same window as your script. You can then visually check the data is as you expect it to be, which works well for small datasets.

We can use the handy function str() to investigate the structure of this object

> str(myData)

'data.frame': 10 obs. of 4 variables:

$ individual: chr "A" "B" "C" "D" ...

$ colour : chr "blue" "blue" "red" "blue" ...

$ mass : num 1 1.2 1.7 1.4 1.5 1.1 1.9 1.5 1.3 1.4

$ length : int 10 9 5 9 12 13 12 7 8 10

From this output we see that myData is a data.frame with four variables (column headings) whose name is listed after the dollar sign $. For the first two variables, it says they are character chr. The variable mass is considered as num (numerical) and length as int (integer).

The dollar signs are printed for a reason; we can use the dollar signs to index a data.frame.

> myData$mass

[1] 1.0 1.2 1.7 1.4 1.5 1.1 1.9 1.5 1.3 1.4

R will consider this a vector, in this case a numerical vector

> class(myData$mass)

[1] “numeric”

and in the case of colour it is a character vector

> myData$colour

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

Other useful functions to perform quick checks are summary(), head(), tail() (chapter 3.4), which summarize the data in each of your variables, give the first 6 rows and last 6 rows respectively.

Critical aspects to check here are that these summary statistics really adhere to what you know about your data. Small typos can result in reading in the data incorrectly. In most cases you have some prior knowledge of what to expect in the data. Check that all the variables are present and the correct data is listed under the correct variable. Important is that variables are indeed interpreted by R in the correct way. In the above, both str() and summary() show that individual and colour are interpreted as factors: the function str() states this and summary() provides numbers of observations per level of a factor, but provides minimum, maximum, quantiles and mean for numeric variables. It is also clever to take some time to from the summary deduce whether the values you obtain are as expected. For example, no negative values or extreme values outside the range you expect. For complicated and large datasets you may need to run additional checks. Bottom line is that whenever you import data and assign it to an object, then this object is a data.frame and you should quickly check it contains the information you expect.

### 4.2 Quick graphical checks: boxplot

We can paraphrase a popular saying to “A picture tells a 1000 numbers”. R has extensive capacity for plotting, and we can make some “quick and dirty” plots to graphically explore that reading in the data was successful as well as visually investigate basic properties of the data. The function boxplot() draws a so-called box-and-whiskers plot which is useful to compare numerical variables across different factorial levels. In doing so we use the functional notation ~ as an argument to inform boxplot() which vectors we want plotted against each other. The boxplot() help page (Box 4.1) looks daunting: there are a lot of arguments and apparently also two ways to use boxplot().

*Box 4.1: R’s help page for boxplot(). Only the first part shown*

**Box Plots**

**Description**

Produce box-and-whisker plot(s) of the given (grouped) values.

**Usage**

boxplot(x, ...)

## S3 method for class 'formula'

boxplot(formula, data = NULL, ..., subset, na.action = NULL,

drop = FALSE, sep = ".", lex.order = FALSE)

## Default S3 method:

boxplot(x, ..., range = 1.5, width = NULL, varwidth = FALSE,

notch = FALSE, outline = TRUE, names, plot = TRUE,

border = par("fg"), col = NULL, log = "",

pars = list(boxwex = 0.8, staplewex = 0.5, outwex = 0.5),

horizontal = FALSE, add = FALSE, at = NULL)

**Arguments**

|  |  |
| --- | --- |
| formula | a formula, such as y ~ grp, where y is a numeric vector of data values  to be split into groups according to the grouping variable grp (usually a factor).  … |

The key aspect to remember is that there are a lot of defaults. Many of these relate to refining your plots which we discuss later on, but here ignore. The key aspect to take from the help page is that we can give as first argument a so-called formula of two vectors, numerical vector y and grouping vector grp. In the previous section we saw how to extract such vectors from the data.frame holding out data

> boxplot(myData$mass~myData$colour)

which produces the boxplot (Fig. 4.2) plotted under the “Plots” tab in the right-hand bottom panel of R Studio.

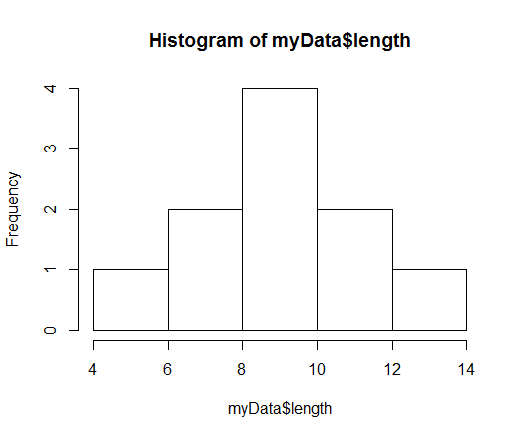
*Figure 4.2: Default box-and-whiskers plot of mass by colour in* myData *produced by* boxplot()

### 4.3 Quick graphical checks: histogram

The function hist() will draw a histogram of a vector. When using its default values, the function will automatically determine much of the settings defining the histogram plotted.

> hist(myData$length)

produces the histogram (Fig. 4.3)



*Figure 4.3: A histogram of the length of individuals in* myData

When assigned to an object, the function will in addition to producing the histogram also produce the information on its “settings” and a breakdown of the numbers plotted.

> descriptive.stats<-hist(myData$length)

> descriptive.stats

$‘breaks‘

[1] 4 6 8 10 12 14

$counts

[1] 1 2 4 2 1

$density

[1] 0.05 0.10 0.20 0.10 0.05

$mids

[1] 5 7 9 11 13

$xname

[1] “myData$length”

$equidist

[1] TRUE

attr(,”class”)

[1] “histogram”

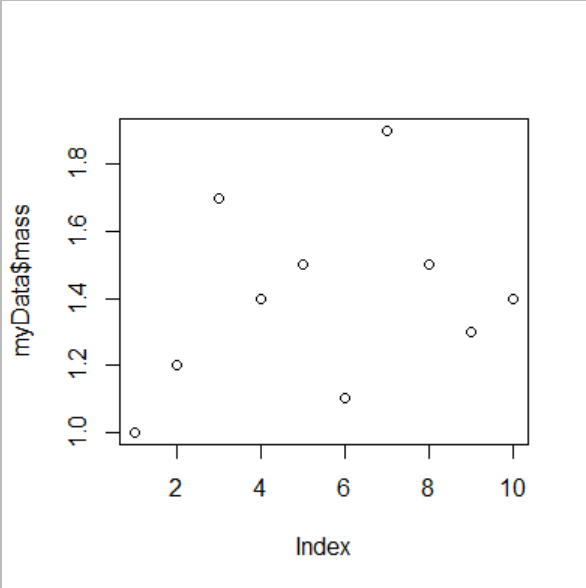
Here, the ‘breaks’ tell us where in the histogram the plotted bins start and end. For example, the first bin counts the number of cases where the data is between 4 and 6. The count is presented in count.

### 4.4 Quick graphical checks: plot

The function plot() is the workhorse in R for producing graphical output. We can obtain a simple plot of the data by calling

> plot(myData$mass)

which plots each value in the vector myData$mass against the “Index” (Figure 4.4). Here, the “Index” is simply a counter going from 1 (first number in the vector) to 10 (10th and last number in the vector. We can thus graphically explore the data, such as minimum and maximum and easily spot any unexpected or extreme values.

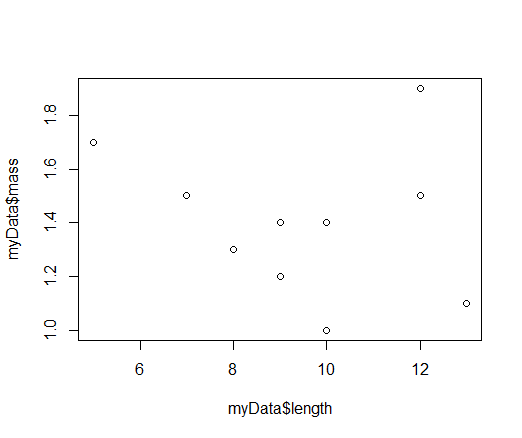


*Figure 4.4: Plot of mass in* myData

We can also plot “Y against X”. As for boxplot() (section 4.2) we use the function notation (the wavy dash ~ ) to plot mass against length

> plot(myData$mass~myData$length)

which produces a scatterplot (Figure 4.5).

*Figure 4.5: Plot of mass against length in* myData

Together the various graphical functions provide a nice way to graphically explore properties of data.

### 4.5. Export your plot

R Studio collects your plots in the bottom right-hand pane. If you go to the “Plots” tab, you can use the arrows to browse through your plots. The plot that is showing can be copied using the “Export” button. Selecting “Copy to Clipboard” is a quick option. Some adjustments can then be made in a separate pane and the result can be copied and then pasted. Using “Export” also allows you to easily export your figure as a graphical file (multiple options are available from the drop-down menu), or as a PDF.

### 4.6 Saving your session results

When quitting R or R Studio, the program will prompt you to save your workspace image. Doing so will save everything that is in R’s memory placed there during the session. Doing so in R Studio will automatically restore R Studio to where you quitted last time. You can execute R saving the entire session by

> save.image()

which will create a file .RData in your working directory. Alternatively, you can name the file. For example

> save.image(“SessionPart1.RData”)

where it is helpful to remember to provide the extension .RData.

It is also possible to save only specific objects using the function save() to a .RData workspace. For example

> save(myData,file=“myData.RData”)

will only save the data.frame myData, and

> save(myData, myMatrix, file=“myData.RData”)

will save both objects myData and myMatrix. You can restore your workspace from a previous session by using load(). For example, we can remove all the objects in the current workspace using the command

> rm(list=ls())

We can then load only the object myData

> load(“myData.RData”)

or all objects

> load(“SessionPart1.RData”)

If there are objects in the current workspace, the function load() will add all the content in the .RData file to your current workspace, but it will not take away anything from R’s memory. Thus, if you have painstakingly created an object in one R session, you can save that object (or objects) and quickly load them in another session using the .RData format.

### 4.7 Exercises Part I

Prepare the exercises such that you can run the script in R Studio to present it to the class. Each exercise is thus a separate script, or part of a longer script. Remember to annotate the script clearly. It is a good idea to use annotation to remind yourself what the question or sub-question was before you provide the solution. You can also annotate any additional notes to yourself. To present your solution script, it should run from the teacher’s computer (the one linked to the projector). There are multiple ways to achieve the running of the script. For example,

1. You log into the teacher’s computer. You run the script from your own “utuhome” directory. The script and any files it may need to run should be on this directory, and should in principle not require modification of the script.
2. Your run the script from a USB memory stick. Your script and any files it needs to run are on the memory stick. You may need to adjust the working directory (drive).
3. Retrieve it from a cloud server and copy it to the teacher computer. Typically requires modification of the directories.

In practice, it is a good idea to test running your script on one of the university computers before the meeting so you can see what is a method that works for you.

When preparing your final compendium, it is sufficient to paste a working script in this compendium. It principle, it can be one long script in the end (sort of a minimum effort solution). If you want to have a record of what the script does, it is better to paste it after each exercise and include the output. This is easiest done by running the entire script of that exercise in R Studio, and pasting what is printed in the console. You can also include any figures. My experience is that you quickly forgot what you did, so the latter, more detailed, option may help you remember what you learned.

**Exercise I.1**

a) Create a suitably named object that is a vector consisting of 10 numbers

b) Perform all the basic arithmetic operations with your vector

c) Extract the fourth and seventh elements from your vector and store these in an object. What class does this second object have?

d) Calculate the mean value of your vector. Make a logical vector stating for each element whether it smaller than the vector’s mean value or not. How many values are below the mean?

e) Construct a matrix out of your vector

f) Extract the values in the second row and the values in the second column of your matrix

g) Extract the value of the first row and second column of your matrix

h) Turn your matrix of values into a matrix of logicals stating for all values in your matrix whether they are larger than the mean value or not. How many values are below the mean?

**Exercise I.2**

a) Get help for the function sum() which we encountered in chapter 1. Review the sections of the help page and run the examples.

b) Use sum() to calculate the sum of the vector c(1,2,NA,4)

c) Use sum() to calculate the sum of myMatrix’s first row and of its first column

d) Use the help page to find a related function that allows you to sum rows and columns of a matrix and use it for myMatrix

**Exercise I.3**

a) Create and read in the MyData file (sections 3.2 and 3.3)

b) Calculate the mean mass and mean length over all individuals

c) Use logicals and indexing to calculate the mean mass of blue and of red individuals separately.

TIP. approach this problem in steps. First work out how to use logicals to get a vector where the blue individuals are TRUE, then how to use this logical vector and indexing to obtain a vector of mass of blue individuals and lastly to calculate the mean of this vector.

**Exercise I.4**

a) Copy-paste the following information to a Notepad file and save it in your working directory. Import the data into R without changing any of the text.

Habitat, count

H3,10

H2,12

H3,15

B5,8

H1,11

H1,13

B5,10

H3,15

H2,10

b) Read the data into R in such a way that habitat is coded as a factor. Investigate the differences in how the variables are interpreted using some of the functions introduced in this part to explore your data.

**Exercise I.5**

a) Make two vectors, with one of these containing one or more NA, and plot a histogram of each. What happens to the NA values?

b) Plot one of your vectors against the other

c) Use the function plot() to plot a “histogram” of one of your vectors (study the help page). What does it do?