# Getting to work in R

Compendium for the course EKOL3159

Jon Brommer

Barbara Class

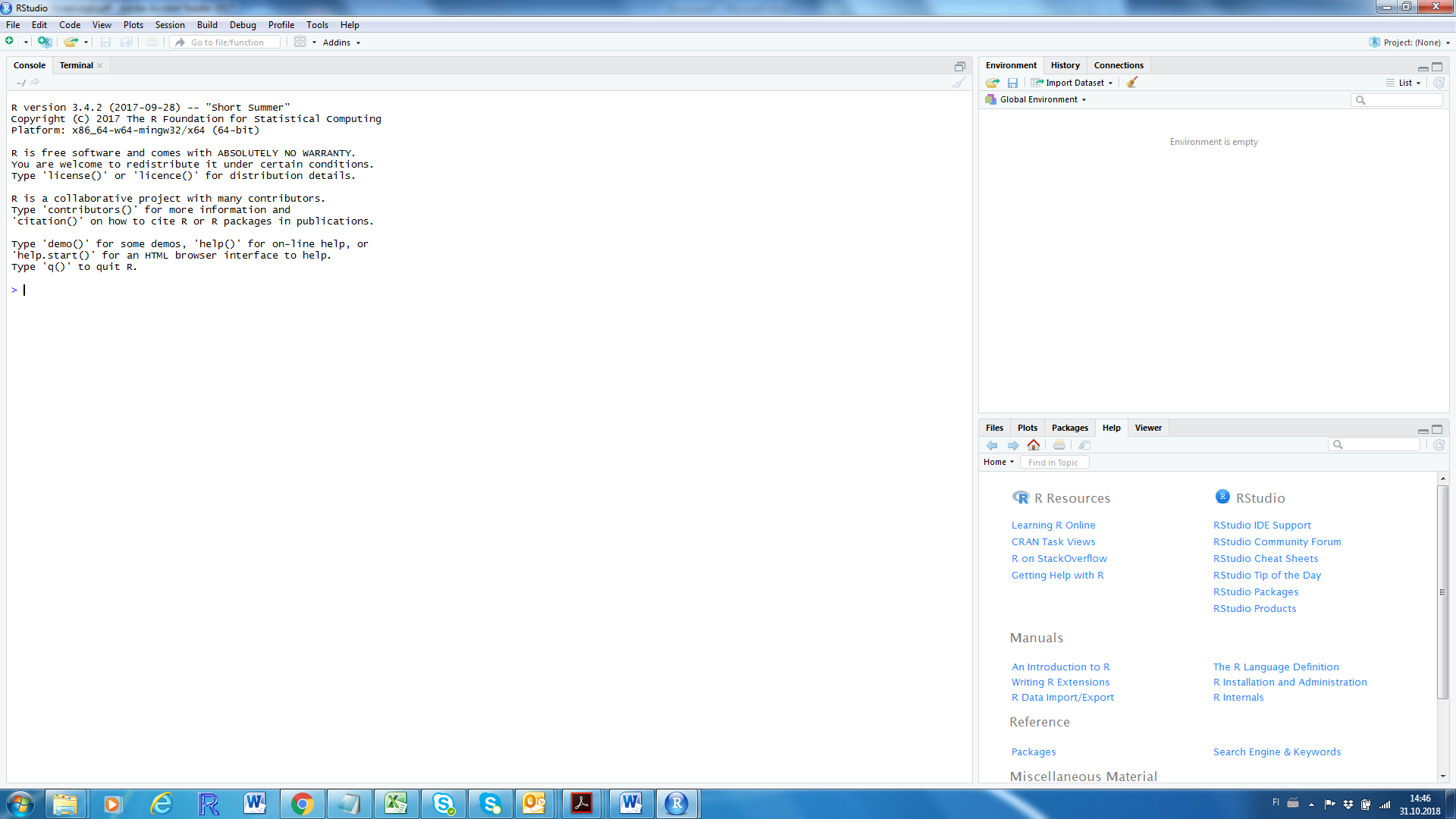
[Janni Heikkinen]

13.3.2019

# Part I: Getting started

## 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 separate program running R within itself and hence requires the program R to be installed. R Studio 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.

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 “”), 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). 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 + R

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+R 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 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.

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.

## 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.

## 1.6 Exercises chapter 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?

# 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

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):

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

Arithmetic Mean

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

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

Description

Generic function for the (trimmed) arithmetic mean.

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”

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

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.

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

References

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

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 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.

## 2.5 Exercises chapter 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

# 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 is to tell R where to look.

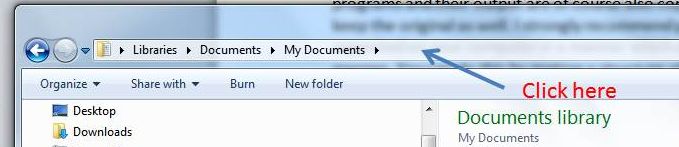
You get to know the working directory by calling the function getwd() (where wd stands for working directory) which produces, formally, a character vector with one element, or in other words a string. For example,

> getwd()

[1] "C:/Users/joegbr/Documents"

The path name may also be something like “//utuhome.utu.fi/joegbr” 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



*Standard view of your folder in windows explorer*



*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\joegbr\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\\joegbr\\Documents")

works fine. Also

> setwd("C:/Users/joegbr/Documents")

will work. Following the same logic, the windows address to the server “\\utuhome.utu.fi\joegbr “ will become the path name string by doubling each backward slash “\\\\utuhome.utu.fi\\joegbr” or equivalently by replacing backward to forward slash “//utuhome.utu.fi/joegbr”. Note that R itself will display path names using forward slashes but will accept both forward and slash (/) and double backward slash (\\) as equivalents.

> setwd("C:\\Users\\joegbr\\Documents")

> getwd()

[1] "C:/Users/joegbr/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. Because this path is a string, we can also assign it to an object

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

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.

## 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). For example,

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

You can make and save this file as a comma separated file CSV. Save the file as “MyData.csv” in your working directory.

## 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 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 options:

> 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”. Double clicking that object will open a separate tab in the top left-hand panel (same as where your script is) with your data displayed correctly. 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

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

We can see the summary statistics differ for the four variables which are named (i.e. R read the headers correctly). For individual and colour, 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.

## 3.5 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.

## 3.6 Exercises chapter 3

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

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

1.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.

2.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

H1,10

H2,12

H3,15

# 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). RStudio will in its Environment panel list data.frame objects under the heading “Data” where also e.g. objects of class matrix appear. We can use the handy function str() to investigate the structure of this object

> str(myData)

'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

From this output we see that myData is a data.frame with four variables (column headings) whose name is listed after the dollar sign $. The first two variables are both interpreted as a **factor**. A factorial variable has levels. There are 10 levels for individual: these are now listed such that their levels are listed first and then (after a second colon) we see how R codes the levels in the data. Thus individual “A” is coded 1, “B” 2, etc. Similarly for colour where there are two levels (“blue” and “red”) such that R has coded the first three cases of colour (blue, blue, red) as 1, 1, 2. 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 factorial vector

> myData$colour

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

Levels: blue red

## 4.2 Quick graphical checks: boxplot

R has extensive capacity for plotting. 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(). They 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 for simplicity. 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

*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).  … |

> boxplot(myData$mass~myData$colour)

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



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)

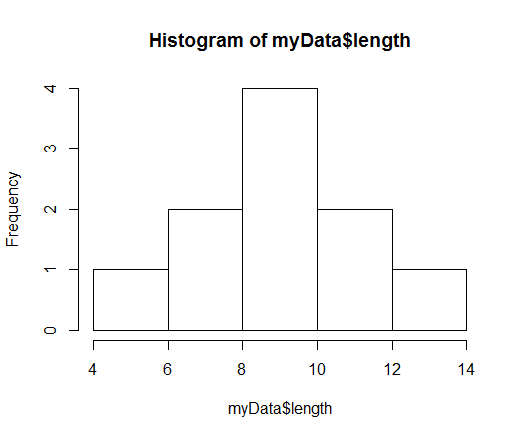


Figure 4.3. A histogram of the length of individuals in myData

When assigned to an object, the function will in addition to producing the histogram also produce 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"

## 4.4 Quick graphical checks: plot

The function plot() is the workhorse in R for producing graphical output. Here we use its default which is plotting Y against X. As for boxplot() (section 4.2) we use the function notation (the curly dash ~ ) to plot mass against length

> plot(myData$mass~myData$length)

which produces a scatterplot (Figure 4.4).

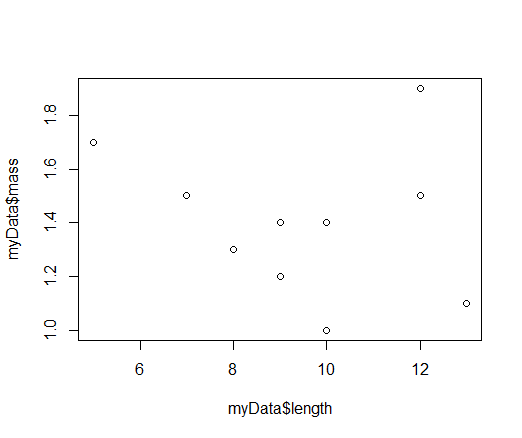


Figure 4.4. Plot of mass against length in myData

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

## 4.5 Exercises chapter 4

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)

## MY CODE: Part 1:

##PART ONE  
#Chapter 1  
  
#a) Create a suitably named object that is a vector consisting of 10  
numbers  
  
myvector <- c(1:10)  
#you don't need the "c" (beacuse it is 1-10. C stands for combine), and  
you actually don't need the brackets  
  
#b) Perform all the basic arithmetic operations with your vector  
  
myvector + 2

## [1] 3 4 5 6 7 8 9 10 11 12

myvector - 2

## [1] -1 0 1 2 3 4 5 6 7 8

myvector \* 2

## [1] 2 4 6 8 10 12 14 16 18 20

myvector / 2

## [1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0

#c) Extract the fourth and seventh elements from your vector and store these in an object. What class does this second object have?  
  
myvector2 <- myvector[c(4,7)]  
class(myvector2)

## [1] "integer"

#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?  
  
mean(myvector)

## [1] 5.5

abovemean <- myvector < mean(myvector)  
sum(myvector>mean(myvector))

## [1] 5

# This could maybe be just sum(abovemean)?  
  
  
#e) Construct a matrix out of your vector  
  
mymatrix <- matrix(myvector, 2,5)  
  
#f) Extract the values in the second row and the values in the second  
column of your matrix  
  
mymatrix[2,]

## [1] 2 4 6 8 10

mymatrix[, 2]

## [1] 3 4

#g) Extract the value of the first row and second column of your matrix  
  
mymatrix[1,2]

## [1] 3

#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?  
  
mymatrixlogicals <- mymatrix > mean(mymatrix)  
sum(mymatrix>mean(mymatrix))

## [1] 5

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
#Chapter 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.  
  
?sum()

## starting httpd help server ... done

example(sum)

##   
## sum> ## Pass a vector to sum, and it will add the elements together.  
## sum> sum(1:5)  
## [1] 15  
##   
## sum> ## Pass several numbers to sum, and it also adds the elements.  
## sum> sum(1, 2, 3, 4, 5)  
## [1] 15  
##   
## sum> ## In fact, you can pass vectors into several arguments, and everything gets added.  
## sum> sum(1:2, 3:5)  
## [1] 15  
##   
## sum> ## If there are missing values, the sum is unknown, i.e., also missing, ....  
## sum> sum(1:5, NA)  
## [1] NA  
##   
## sum> ## ... unless we exclude missing values explicitly:  
## sum> sum(1:5, NA, na.rm = TRUE)  
## [1] 15

#b) Use sum() to calculate the sum of the vector c(1,2,NA,4)  
  
sum(c(1,2,NA,4), na.rm = TRUE)

## [1] 7

#c) Use sum() to calculate the sum of myMatrix’s first row and of its  
first column  
  
sum(mymatrix[1,], mymatrix[, 1])

## [1] 28

#Now the sum is 28 when it is in reality 27 since one value gets counted  
twice  
  
#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  
  
#sum() functions "See also":  
?rowsum()  
?colSums()  
  
rowSums(mymatrix)

## [1] 25 30

colSums(mymatrix)

## [1] 3 7 11 15 19

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#Chapter 3  
  
#1.a) Create and read in the MyData file (sections 3.2 and 3.3)  
  
setwd("E:/Introduction to R")  
mydata <- read.csv2("myData.csv")  
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

#1.b) Calculate the mean mass and mean length over all individuals  
  
 # three different ways to do this  
mean(mydata$mass)

## [1] 1.4

mean(mydata$length)

## [1] 9.5

mean(mydata[,3])

## [1] 1.4

mean(mydata[,4])

## [1] 9.5

mean(mydata[,"mass"])

## [1] 1.4

mean(mydata[,"length"])

## [1] 9.5

#1.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.  
  
blue <-mydata[,2] == "blue"  
mean(mydata[blue,3])

## [1] 1.316667

#alternative way to ^  
mean(mydata$mass[blue])

## [1] 1.316667

red <-mydata[,2] == "red"  
mean(mydata[red,3])

## [1] 1.525

#alternative ways:  
c(mydata[blue[TRUE],3])

## [1] 1.0 1.2 1.4 1.5 1.5 1.3

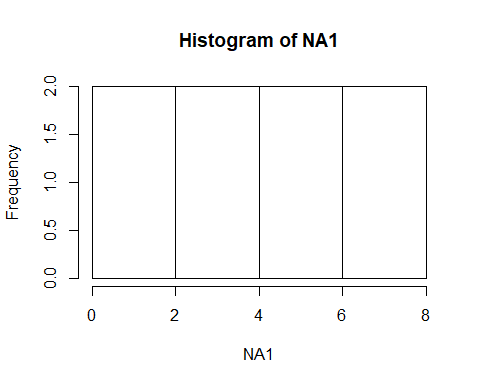
mydata[blue=="TRUE", 3]

## [1] 1.0 1.2 1.4 1.5 1.5 1.3

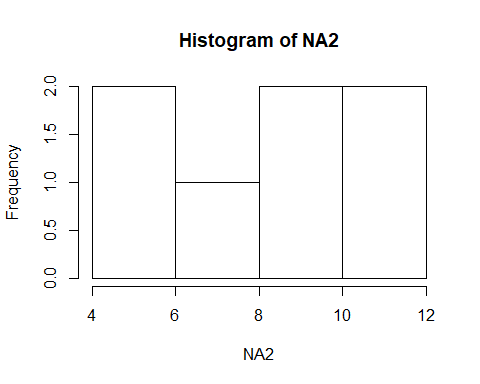
#2.a) Copy-paste the following information to a Notepad file and save it  
in your working directory. Import the data into R without changing any of the text.  
  
habitat <- read.csv("habitat\_count.txt")  
print(habitat)

## Habitat count  
## 1 H1 10  
## 2 H2 12  
## 3 H3 15

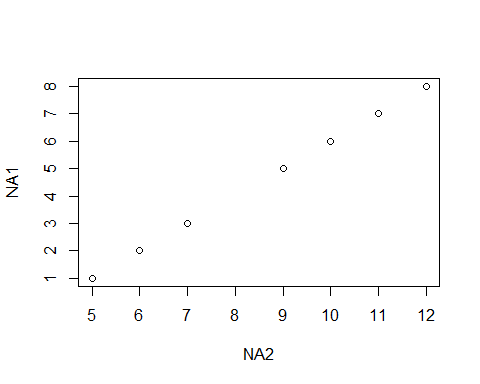
#if you want you could write the whole path to the file, but that is not needed  
  
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#Chapter 4  
  
#a) Make two vectors, with one of these containing one or more NA, and plot a histogram of each. What happens to the NA values?  
  
NA1 <- c(1:8)  
NA2 <- c(5, 6, 7, NA, 9, 10, 11, 12)  
hist(NA1)



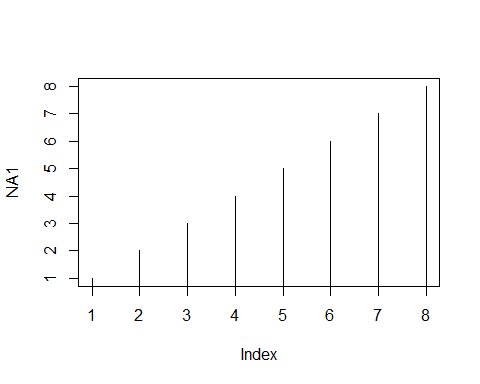
hist(NA2)



#What happens? It does not include them  
  
#b) Plot one of your vectors against the other  
  
plot(NA1 ~ NA2)



#The NA value is missing here aswell  
  
#c) Use the function plot() to plot a histogram of one of your vectors  
(study the help page)  
  
?plot  
plot(NA1, type = "h")



# Part II: R workflow: starting to do stuff with your data

In this second part, we start by explicitly introducing the R workflow (Figure 5.0).



Figure 5.0. Flow diagram of five distinct sets of operations (in different colors) which together define a workflow of many typical coding jobs. The diagram is not exhaustive, however, and both less and more operations may be required.

The workflow is perhaps the main advantage of working in R. The capacity to get your information (data) into R and then perform all subsequent steps in R, e.g. quick plotting, calculations, doing statistical analyses, making publication-quality graphs. In part I, we literally raced through an entire workflow, from getting the data (operation 1) to checking the data (operation 2) with some analyses (operation 4), albeit a very restricted workflow.

Given data, the R script allows you to keep this information in the original shape, read it into R’s brain for further processing (filtering, analyses) and prepare high-quality output. Data and script together provide a reproducible and transparent record of what was done to arrive at the final output. As long as the original data file is kept intact, you can run the script to produce the same outcome and you can share it with others.

In this second part, we look in more depth at operations 3 (data manipulation), and 4 (analysis) in the R workflow (Figure 5.0).

## 5.1 Structure your script

Some general observations on coding to start with. Your R script is a small program coded in R language. R is a platform for programming. A program is a list of instructions to be carried out by the computer. It is like a recipe for when you are cooking.

1. Mix 6 dl flower with 2.5 dl milk, some salt and 3 eggs.

2. Let the batter stand for some minutes while you heat up a frying pan

3. Bake your pancakes in butter.

The programs in R are, of course, for solving numerical problems. A program has a logical sequence for carrying out the instructions, which you can often group in sections. The next section (next set of instructions) often depends on the previous section being completed. For example, you first must read in the data before you can plot it. Annotation (use of comments in #) is a great way to clarify the different sections in your code. Actually, even before writing any R code, you can use annotations to plan the different steps your script should be doing. In general, it is always clever to annotate as if you would be writing the code for a person without a clue of what the program will be doing. This is because it is easy to forget what the lines were about even if you have spent a lot of time working out how to code it, you may not remember it after days/weeks/months. Annotation can be added prior or behind the line with code. You can visually structure your code by creating ‘blocks’ using various symbols and white-lines.

This is especially handy for larger programs that do multiple things in a row. For example

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#\*\* This code reads in data and does something complicated,

#\*\* plots and returns a value

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#----- SECTION 1 ----------

# Reading in the data #####

###########################

#data consists of the following variables

#1. Name

#2. Sex

#3. Score

…

#------ SECTION 2 ----------------------------------------

# Getting the first really difficult calculations done ###

##########################################################

# defining the variables needed

… code …

# calculating their fancy statistical thingy

… code …

#------ SECTION 3 ----------------------------------------

# Plotting to see if it all makes sense ###

##########################################################

.. code code code ..

Another aspect which helps to increase clarity is to choose variable names which have an intuitive meaning. Thus, avoid having a, b, c, d; instead these could be data, regressionModel, populationSize, output, etc. R is really specific in reading what you type in; it does not guess about your meaning and also takes into account upper and lower cases. This applies to functions, but also to the objects you make yourself. So, the object numberOfStudents is something else for R than numberofstudents. It is therefore handy to be consistent in your “style”. If you for example have a data file read into R as an object called myData, and the columns of your datafile have names, it is clever to either have them all start with a capital (or then not, as long as you are consistent). These columns can be accessed by the dollar sign $. So, you will have data$Sex and data$Name and not some kind of mix (e.g. data$Sex and data$name). This avoids having to guess and scrolling up and down to check how you defined variable names. Of course, if you use R Studio for coding in R, you will notice that R Studio will try to help you by giving suggestions, but consistency is still a clever habit.

Take home message: Think about *meaningful identifying names*, add *comments* and make a transparent *layout.* Importantly, a structured layout can help you plan the script before you start the “real” coding, as well as afterwards aid in understanding what was coded. A clearly structured script can be “read” by anyone with knowledge in R. Again, even when you are not coding the program for someone else but just for your own use, it is still very useful to have all this structure and explanation when you again look at the code some time later. It is surprising how quickly you forget what was done and why in your script.

## 5.2 More on R objects

R is an object oriented programming language. An object is anything saved in R’s brain. Easiest way to think of an object is that it is anything left of the assign operator “<-“. Objects have different classes. We have encountered some basic classes and in this section more detail is provided

## 5.2.1 Vector

A main trick in programming is learning to think in terms of *vectors* and *matrices*. A vector is a collection of numbers (R: class: “numeric”) or strings (R class “character”). It is written in text using square brackets. For example, the vector



is a row vector containing three numbers (1, 2 and 3). A vector can also be written as a column vector. For example,



In R, these are formed – for example – as c(1,2,3) for the vector *v*. R does not really work with column vectors per se, more on this later.

In R, we do not have single numbers, or scalars, or single pieces of text (strings), but all these are viewed as a vector. A matrix is, in a way, a collection of vectors stacked in rows or columns, and when you import data, you can interpret all your data for a given variable as a vector. In R *the ‘lowest level’ is always a vector*. The number (mathematicians would say, scalar) 1, for example, is in R simply a vector with only one number. It is the vector [1]. When programming, it is clever to start thinking in terms of vectors as much as possible. This is not always so easy. If we have data on individuals, and for each of these we have measured a lot of different variables, we tend to think in terms of these individuals (the rows of our datafile) rather than the data being a series of column vectors each containing the information of one particular aspect. Yet, this is how R sees it. This if why we have data$Sex and data$Name. The name of the individuals in the data is just one vector for R. Most of the basic operations which are of interest, such as filtering, plotting or calculating with your data, all involve operations on vectors. Thinking in vectors will therefore allow more efficient, but – more importantly – more transparent code.

Many built-in functions are what is called vectorized: The operation is applied to the entire vector or matrix in one go. For example,

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

> 2\*x

[1] 2 4 6 8

## 5.2.2. Factor

A factorial variable in statistics is when a variable is considered to have levels (e.g. control and treatment). These may be ordinal (there is an order: low, medium, high) or nominal (no order: black, red, blue). R will interpret all text and alphanumeric data that you read into R’s “brain” as factors. Alphanumeric refers to variables that have a character and number in them. For example, you may code males and females alphanumerically as m1, m2, f1, f2, which will be read into R as factor by default when using read.table(). Factorial variable sometimes mess up some of the coding, so it is good to be aware of these. The function str() (for structure) is useful to display whether a variable in your data is a factor or not. Consider again the myData object from Part I.

> str(myData)

'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

When reading in data, R will code factors as 1,2,3,… in alphabetic order (which may not be the order in your data or the order you want: You can of course change the levels in R; see the help for levels()). A factorial vector can be coerced to be a character or numeric vector

> as.character(myData$colour)

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

> as.numeric(myData$colour)

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

When reading in data you can also override the default to interpret strings as factors by specifying stringsAsFactors=FALSE in which case alphanumeric fields in your data will be interpreted as strings (character vector).

> myData.char=read.csv2("MyData.csv",stringsAsFactor=FALSE)

> str(myData.char)

'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

A character or numerical vectors can be turned into a factorial vector using the command as.factor().

> as.factor(c("a","b","c"))

[1] a b c

Levels: a b c

> as.factor(c(1,2,3))

[1] 1 2 3

Levels: 1 2 3

## 5.2.3 matrix

We have in Part I encountered this class. It is used for storage in two dimensions of information where all elements need to be either numeric or character. For example,

> matrix(c(1,2,3,4),2,2)

[,1] [,2]

[1,] 1 3

[2,] 2 4

stores the number 1 to 4 in a 2 x 2 matrix. It may seem redundant to specify the matrix has to be 2 x 2 given we provide a vector with 4 numbers. However, this specification is needed because R will recycle the information provided to fill the matrix to be the dimension specified. For example, storing the numbers 1 to 4 in a 2 x 3 matrix produces

> matrix(c(1,2,3,4),2,3)

[,1] [,2] [,3]

[1,] 1 3 1

[2,] 2 4 2

Warning message:

In matrix(c(1, 2, 3, 4), 2, 3) :

data length [4] is not a sub-multiple or multiple of the number of columns [3]

Note that R produces a warning, yet still performs the operation. Warnings like this are typical of R; its warns your request is odd although it will comply with it. Recycling is handy in some cases. For example, it allows easily creating a 2x2 matrix of ones

> matrix(1,2,2)

[,1] [,2]

[1,] 1 1

[2,] 1 1

A second aspect to note is that R will by default fill the matrix by columns, not by rows. Bringing up the help for matrix explains

matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,

dimnames = NULL)

as.matrix(x, ...)

## S3 method for class 'data.frame'

as.matrix(x, rownames.force = NA, ...)

is.matrix(x)

**Arguments**

|  |  |
| --- | --- |
| data | an optional data vector (including a list or [expression](http://127.0.0.1:13623/library/base/help/expression) vector). Non-atomic classed **R** objects are coerced by [as.vector](http://127.0.0.1:13623/library/base/help/as.vector) and all attributes discarded. |
| nrow | the desired number of rows. |
| ncol | the desired number of columns. |
| byrow | logical. If FALSE (the default) the matrix is filled by columns, otherwise the matrix is filled by rows. |

Thus, we need to override the default byrow=FALSE and specify byrow=TRUE in case we wish to fill the matrix by rows.

> matrix(c(1,2,3,4),2,2, byrow=TRUE)

[,1] [,2]

[1,] 1 2

[2,] 3 4

## 5.2.4 data.frame

We have seen that data is read into R as an object of class data.frame. A data.frame object can store information in 2 dimensions but can hold both numeric and characters. We can also make a data.frame using the function (surprise!) data.frame(). For example, using vectors

> students<-c("Bill", "Miranda")

> ages<-c(22,21)

> df1<-data.frame(students,ages)

> df1

students ages

1 Bill 22

2 Miranda 21

> str(df1)

'data.frame': 2 obs. of 2 variables:

$ students: Factor w/ 2 levels "Bill","Miranda": 1 2

$ ages : num 22 21

The above first used c() to make two vector objects (students and ages) and then combines these using data.frame() in a data.frame. The data.frame df1 now consists of a factorial variable and of a numeric one. The names of these variables in the data.frame (R uses the term *tag* for these “variable names”) are taken to be the names of the two vectors (students and ages). Thus, the function data.frame() has converted the character vector students to a factor. Consulting the help for data.frame explains

**Usage**

data.frame(..., row.names = NULL, check.rows = FALSE,

check.names = TRUE, fix.empty.names = TRUE,

stringsAsFactors = default.stringsAsFactors())

**Arguments**

|  |  |
| --- | --- |
| ... | these arguments are of either the form value or tag = value. Component names are created based on the tag (if present) or the deparsed argument itself. |

---

|  |  |
| --- | --- |
| stringsAsFactors | logical: should character vectors be converted to factors? The ‘factory-fresh’ default is TRUE, but this can be changed by setting [options](http://127.0.0.1:13623/library/base/help/options)(stringsAsFactors = FALSE). |

From this help info, we can see that stringsAsFactors is at play here. Clearly, by default, the function data.frame() will assume that stringsAsFactors = TRUE. To see what happens if we override the default by specifying stringsAsFactors = FALSE.

> df2<-data.frame(students,ages, stringsAsFactors=FALSE)

> df2

students ages

1 Bill 22

2 Miranda 21

> str(df2)

'data.frame': 2 obs. of 2 variables:

$ students: chr "Bill" "Miranda"

$ ages : num 22 21

Now we have created a data.frame which keeps the character vector students as a character vector.

We can index a data.frame in multiple ways as with vectors and matrices

> df1[1]

students

1 Bill

2 Miranda

> df1[,1]

[1] Bill Miranda

Levels: Bill Miranda

> df1[[1]]

[1] Bill Miranda

Levels: Bill Miranda

> df1[,"students"]

[1] Bill Miranda Laura

Levels: Bill Miranda Laura

> df1$students

[1] Bill Miranda

Levels: Bill Miranda

From the above, using the $ sign is clearly the “safest” in the sense that using the name of the column (students) is clear whereas using the number of the column is more confusing coding. On the other hand, in programming, especially if the script needs to be generic, indexing with numbers often is handy. Again, there are several ways to the same outcome.

When extracting subdata out of a data.frame (as we do above), you should be aware that the class of the extracted data is not necessarily data.frame.

> df1[,2]

[1] 22 21

> class(df1[,2])

[1] "numeric"

Thus, by extracting the second column of the data.frame we obtain a numeric vector (which is of course how it was defined in the first place). Typically, R will assume the “lowest” possible class (in this case vector). You can override this change in class using drop=FALSE in which case

> df1[,2,drop=FALSE]

ages

1 22

2 21

> class(df1[,2,drop=FALSE])

[1] "data.frame"

and the extracted column from the data.frame will stay a data.frame and not “drop” in class from data.frame to vector.

Once we have a data.frame, we can add information to it using rbind() and cbind() for adding rows and columns respectively. Consider for example the addition of an extra student to the data.frame

> df1<-rbind(df1,data.frame(students="Laura",ages=19))

> df1

students ages

1 Bill 22

2 Miranda 21

3 Laura 19

> str(df1)

'data.frame': 3 obs. of 2 variables:

$ students: Factor w/ 3 levels "Bill","Miranda",..: 1 2 3

$ ages : num 22 21 19

In the above, we have defined a new df1 to be the old df1 with an extra row. Note that for this adding an extra row to work, a data.frame with the same tags must be added to the existing data.frame. This is why data.frame(students="Laura",ages=19) is added to the existing data.frame df1. Not using the same tags (i.e. names for the variables), will produce an error.

> rbind(df1,data.frame(student="Laura",age=19))

Error in match.names(clabs, names(xi)) :

names do not match previous names

We can also add columns, perhaps a new column based on the existing information. For example, the difference in age of each student to age = 20.

> df1.ageDiff<-cbind(df1,df1$age-20)

> df1.ageDiff

students ages df1$age - 20

1 Bill 22 2

2 Miranda 21 1

3 Laura 19 -1

Note that we now also named the new data.frame differently to indicate that information was added. Renaming is, however, not required. R will happily accept the somewhat recursive statement

> df1<-cbind(df1,df1$age-20)

> df1

students ages df1$age - 20

1 Bill 22 2

2 Miranda 21 1

3 Laura 19 -1

We may also want to rename the new column number 3,

> names(df1.ageDiff)[3]<-"ageDiff"

> df1.ageDiff

students ages ageDiff

1 Bill 22 2

2 Miranda 21 1

3 Laura 19 -1

# Chapter 6. Elementary filtering of data

After reading in the data, one often needs to change some aspects of it. In data science, this is called data manipulation. To biologists, the term “data manipulation” sounds like a bad, unscientific idea. The term is used, however, to describe operations in which your dataset is altered as to become suited for analysis. Alteration include selecting, inserting, deleting and updating data. One example of data manipulation we performed was by “extracting” part of it using indexing. Data manipulation approaches are often needed before analyses are carried out. At the very least, some kind of filtering is often required.

## 6.1. ifelse() and relational operators

Conditional statement can be used to manipulate data. These are instructions like “*If* you have fresh tomatoes, boil these shortly, peel off their skin, slice them and fry them, *else* open a can of tomatoes and fry its content.” Sometimes there are more options. R uses the Booleans TRUE and FALSE to distinguish statements. For example, the above text can be translated to R using the function ifelse(), which consists of (1) a logical statement (“If you have tomatoes…”), which (2) if TRUE, performs a certain instruction (boil, peel, slice, fry), but (3) if FALSE carries out another (open a can and fry its content). For example

> ifelse(myData$colour=="blue","blue","not blue")

[1] "blue" "blue" "not blue" "blue" "blue" "not blue" "not blue" "blue"

[9] "blue" "not blue"

returns a vector of “blue” and “not blue” for blue and non-blue individuals respectively. Note the double ‘equal to’ mark == which is used by R to distinguish ‘=’ (which you can use to define an argument or an object) from ‘is equal to?’. Ask help for == to get an overview of relational operators

Box 6.1. Help page for == (shortened)

|  |  |
| --- | --- |
| Comparison {base} | R Documentation |

Relational Operators

Description

Binary operators which allow the comparison of values in atomic vectors.

Usage

x < y

x > y

x <= y

x >= y

x == y

x != y

Arguments

|  |  |
| --- | --- |
| x, y | atomic vectors, symbols, calls, or other objects for which methods have been written. |

From the above displayed help note especially the use of the exclamation mark ! to indicate the reverse (!= means “not equal to”). The use of ! as a reversal works for all logical vectors.

> c(T,T,F,F)

[1] TRUE TRUE FALSE FALSE

> !c(T,T,F,F)

[1] FALSE FALSE TRUE TRUE

The ifelse() operation can be used to recode data. For example to add an extra column where the colour information is recoded

> myData$colour.recoded<-ifelse(myData$colour=="blue","blue","not blue")

> myData$colour.recoded

[1] "blue" "blue" "not blue" "blue" "blue" "not blue" "not blue" "blue"

[9] "blue" "not blue"

Note that by doing so the factorial vector myData$colour was recoded in a vector of strings myData$colour.recoded (visible above for the observant as the “levels” are not displayed)

> class(myData$colour.recoded)

[1] "character"

If you want, this vector can be coerced into a factorial one by using the function as.factor().

The conditional statement allows us to recode columns in the data.frame in one line. For example, all individuals with mass exceeding 1.3 are to be left untouched, or if below or equal to 1.3 are coded as NA

> ifelse(myData$mass>1.3,myData$mass,NA)

[1] NA NA 1.7 1.4 1.5 NA 1.9 1.5 NA 1.4

Because a factorial vector is recoded as a character vector, the above formulation does not work for factorial vectors

> ifelse(myData$colour=="blue",myData$colour,"not blue")

[1] "1" "1" "not blue" "1" "1" "not blue" "not blue" "1"

[9] "1" "not blue"

## 6.2 subset() and logical operators

We can use indexing in combination with a relational statement to extract data on the blue individuals from the object myData

> myData[myData$colour=="blue",]

individual colour mass length colour.recoded

1 A blue 1.0 10 blue

2 B blue 1.2 9 blue

4 D blue 1.4 9 blue

5 E blue 1.5 12 blue

8 H blue 1.5 7 blue

9 I blue 1.3 8 blue

The convenience function in R for this kind of operations is subset(). The help for this function shows

The above subsetting of myData can thus be alternatively performed by introducing the conditional as the second argument (subset)

Box 6.2.1. Help page for subset (shortened)

|  |  |
| --- | --- |
| subset {base} | R Documentation |

Subsetting Vectors, Matrices and Data Frames

Description

Return subsets of vectors, matrices or data frames which meet conditions.

Usage

subset(x, ...)

## Default S3 method:

subset(x, subset, ...)

## S3 method for class 'matrix'

subset(x, subset, select, drop = FALSE, ...)

## S3 method for class 'data.frame'

subset(x, subset, select, drop = FALSE, ...)

Arguments

|  |  |
| --- | --- |
| x | object to be subsetted. |
| subset | logical expression indicating elements or rows to keep: missing values are taken as false. |
| select | expression, indicating columns to select from a data frame. |
| drop | passed on to [ indexing operator. |
| ... | further arguments to be passed to or from other methods. |

> subset(myData,colour=="blue")

individual colour mass length colour.recoded

1 A blue 1.0 10 blue

2 B blue 1.2 9 blue

4 D blue 1.4 9 blue

5 E blue 1.5 12 blue

8 H blue 1.5 7 blue

9 I blue 1.3 8 blue

We may also wish to use the third argument (select) to restrict the columns. Perhaps to only include individual, mass and length

> subset(myData,colour=="blue", c(individual,mass,length))

individual mass length

1 A 1.0 10

2 B 1.2 9

4 D 1.4 9

5 E 1.5 12

8 H 1.5 7

9 I 1.3 8

The last argument (drop) is specified to by default be FALSE. As we have seen in section 5.2.3 this means that the class of the object returned by subset will be the same as the object it is given. Thus,

> subset(myData,colour=="blue",individual)

individual

1 A

2 B

4 D

5 E

8 H

9 I

returns a data.frame, whereas

> myData[myData$colour=="blue","individual"]

[1] A B D E H I

Levels: A B C D E F G H I J

returns a vector (addition of ,drop = FALSE in the above call to subset() will of course avoid that).

Conditional statements can be combined using the function & (stands for “and”). Statements then need to be placed inside brackets.

> subset(myData,(colour=="blue")&(mass>1.3),c(individual,mass))

individual mass

4 D 1.4

5 E 1.5

8 H 1.5

or the function | (typically termed “bar”, which stands for “or”)

> subset(myData,(colour=="blue")|(mass>1.3),c(individual,colour,mass))

individual colour mass

1 A blue 1.0

2 B blue 1.2

3 C red 1.7

4 D blue 1.4

5 E blue 1.5

7 G red 1.9

8 H blue 1.5

9 I blue 1.3

10 J red 1.4

Consult the help for e.g. & to obtain a list of logical operators

|  |  |
| --- | --- |
| Box 6.2.2 Help for & (shortened)  Logic {base} | R Documentation |

Logical Operators

Description

These operators act on raw, logical and number-like vectors.

Usage

! x

x & y

x && y

x | y

x || y

xor(x, y)

---- shortened

# Chapter 7. Elementary analysis: repeated calculations for subsets

In this chapter we make a start with analysis by introducing some basic R functions for calculating descriptive statistics.

## Chapter 7.1 aggregate()

Suppose we want to calculate mean of mass of blue and for red individuals separately. We could use subset (or indexing) to extract information on these two groups and calculate the means

> mean(subset(myData,colour=="blue",mass)[,"mass"])

[1] 1.316667

> mean(subset(myData,colour=="red",mass)[,"mass"])

[1] 1.525

In the above lines, we first subset the data to be only the mass of blue individuals (which creates a data.frame) and then also index this data.frame for column [,”mass”] to provide the function mean with the data as a vector. The alternative, as discussed in section 6.2 is to specify drop as TRUE

> mean(subset(myData,colour=="blue",mass,drop=TRUE))

[1] 1.316667

> mean(subset(myData,colour=="red",mass,drop=TRUE))

[1] 1.525

It is handy to use one of R’s functions for repeating analysis for different subsets. For example, the function aggregate()(Box 7.1)

> aggregate(mass~colour, data=myData,FUN=mean)

colour mass

1 blue 1.316667

2 red 1.525000

applies the function (argument FUN) mean() to data on mass for each subset of the data specified by colour. In the above, we use the so-called “formula” implementation (the wavy dash ~; y~x indicates y as a function of x) as well as specifying in detail the second and third argument. The above can be shortened to

> aggregate(mass~colour, myData,mean)

colour mass

1 blue 1.316667

2 red 1.525000

The above work because, as shown in the help for aggregate() (Box 7.1), under the formula method, aggregate() expects the first argument to be the formula (mass~colour), the second argument to be the data.frame (myData) and the third argument to be the function (mean). The help also shows that aggregate() can take arguments in different ways. You can take some time to try these various approaches; they differ in subtle ways in their output. For simplicity, we here stick to the formula way of using this function.

Box 7.1. Help for aggregate (shortened)

|  |  |
| --- | --- |
| aggregate {stats} | R Documentation |

Compute Summary Statistics of Data Subsets

Description

Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

Usage

aggregate(x, ...)

## Default S3 method:

aggregate(x, ...)

## S3 method for class 'data.frame'

aggregate(x, by, FUN, ..., simplify = TRUE, drop = TRUE)

## S3 method for class 'formula'

aggregate(formula, data, FUN, ...,

subset, na.action = na.omit)

## S3 method for class 'ts'

aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,

ts.eps = getOption("ts.eps"), ...)

Arguments

|  |  |
| --- | --- |
| x | an R object. |
| by | a list of grouping elements, each as long as the variables in the data frame x. The elements are coerced to factors before use. |
| FUN | a function to compute the summary statistics which can be applied to all data subsets. |
| simplify | a logical indicating whether results should be simplified to a vector or matrix if possible. |
| drop | a logical indicating whether to drop unused combinations of grouping values. The non-default case drop=FALSE has been amended for **R** 3.5.0 to drop unused combinations. |
| formula | a [formula](http://127.0.0.1:40914/help/library/stats/help/formula), such as y ~ x or cbind(y1, y2) ~ x1 + x2, where the y variables are numeric data to be split into groups according to the grouping x variables (usually factors). |
| data | a data frame (or list) from which the variables in formula should be taken. |
| subset | an optional vector specifying a subset of observations to be used. |

Suppose we have an additional categorical variable, habitat, which we add

> myData$habitat<-c(rep("sunny",5),rep("shady",5))

> myData

individual colour mass length colour.recoded habitat

1 A blue 1.0 10 blue sunny

2 B blue 1.2 9 blue sunny

3 C red 1.7 5 not blue sunny

4 D blue 1.4 9 blue sunny

5 E blue 1.5 12 blue sunny

6 F red 1.1 13 not blue shady

7 G red 1.9 12 not blue shady

8 H blue 1.5 7 blue shady

9 I blue 1.3 8 blue shady

10 J red 1.4 10 not blue shady

We then can calculate means in mass for all combinations of these both categories

> aggregate(mass~colour+habitat, myData,mean)

colour habitat mass

1 blue shady 1.400000

2 red shady 1.466667

3 blue sunny 1.275000

4 red sunny 1.700000

where + is used to add the second category in the formula (Box 7.1).

## 7.2 tapply()

Another useful function to perform similar operations as aggregate() is tapply(), pronounced as t-apply; it is one function in the so-called apply() family to which also aggregate() belongs. Investigation of its help (Box 7.2) shows that we cannot use the formula syntax for tapply() but have to pass it vectors

> tapply(myData$mass,myData$colour,mean)

blue red

1.316667 1.525000

> tapply(myData[,"mass"],myData$colour,mean)

blue red

1.316667 1.525000

The help page for tapply() is rather technical (Box 7.2). It includes the term “ragged array”. We will later get more familiar objects of class array. For the present, it is sufficient to realise that tapply() is a fairly general function which will produce an object of class array

> means.per.colour<-tapply(myData[,"mass"],myData$colour,mean)

> class(means.per.colour)

[1] "array"

Box 7.2. Help page for tapply (shortened)

|  |  |
| --- | --- |
| tapply {base} | R Documentation |

## Apply a Function Over a Ragged Array

### Description

Apply a function to each cell of a ragged array, that is to each (non-empty) group of values given by a unique combination of the levels of certain factors.

### Usage

tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)

### Arguments

|  |  |
| --- | --- |
| X | an **R** object for which a [split](http://127.0.0.1:40914/help/library/base/help/split) method exists. Typically vector-like, allowing subsetting with [[](http://127.0.0.1:40914/help/library/base/help/%5b). |
| INDEX | a [list](http://127.0.0.1:40914/help/library/base/help/list) of one or more [factor](http://127.0.0.1:40914/help/library/base/help/factor)s, each of same length as X. The elements are coerced to factors by [as.factor](http://127.0.0.1:40914/help/library/base/help/as.factor). |
| FUN | the function to be applied, or NULL. In the case of functions like +, %\*%, etc., the function name must be backquoted or quoted. If FUN is NULL, tapply returns a vector which can be used to subscript the multi-way array tapply normally produces. |

## 7.3 Different statistics per subset

In the above examples we only considered one descriptive statistic, the average as provided by the function mean(). Various other functions can of course be used to obtain other descriptive statistics. by specifying the function argument to aggregate() or tapply(). For example, the standard deviation SD,

> sd.per.colour<-tapply(myData[,"mass"],myData$colour,sd)

> sd.per.colour

blue red

0.194079 0.35000

Again, internally tapply() and aggregate() construct subset specific vectors of the variable you pass to these functions. By using length() as the function, you can count the number of elements in each of these vectors which is your sample size (except in the case of missing values).

> n.per.colour

blue red

6 4

## 7.3 Exercises part II

II.1

a) Calculate the mean, SD and sample size of length for each colour in myData

b) Use aggregate() to calculate the means of both mass and length in myData for each colour in one command line

c) Code the construction of the following data.frame

Trial Experiment Measure

1 1 control 1

2 2 control 3

3 3 control 4

4 4 control 3

5 5 control 3

6 6 treatment 4

7 7 treatment 5

8 8 treatment 8

9 9 treatment 8

10 10 treatment 9

and use tapply() to calculate the mean, SD and sample size for each experimental group.

II.2

Write a script which

1. imports the datafile “ExperimentalData.txt” which you can find in Moodle
2. Performs some checks the data import was successful
3. Recodes the 4 groups to:

group Feeding treatment Parasite treatment

1 not fed removed

2 not fed retained

3 fed removed

4 fed retained

(d) Extracts all data where the parasite was removed

(e) Calculates the variance for all 4 experimental groups

(f) Calculates the mean for “fed” and “not fed” treatment groups

II.3

Use the data in "Fruit\_tasting\_data\_2018.txt" from Moodle

1. Create four additional variables: score1.apples and score2.apples which hold score1 and score2 respectively only for apples, and score1.strawberies and score2.strawberries which contain score1 and score2 respectively but only for strawberries.
2. Calculate the mean for each of these four variables
3. Create two datasets. One for apples and one for strawberries
4. Make an object of class numeric containing score1 values for apples
5. Make an object of class character containing the name of each fruit tasted in the dataset

II.4

Write a script which

1. defines an object that is a matrix with 10 columns containing the numbers 1 to 100
2. defines another object that is a matrix with 10 columns containing the numbers 1 to 100 such that the numbers increment by 1 for each row across the column (i.e. row 1 is 1, 2, 3, etc)
3. names the rows and columns of these two objects
4. Extracts the third row of each object
5. Calculates the mean of each column in each object (TIP: Find functions related to the function mean() from R’s help)

II.5

1. Create the following data.frame

Plot Species SoilType CationValue

P1 Acer Wet 10

P2.2 Acer Wet 12

P2.3 Quercus Dry 4

P3 Quercus Dry 3

P10 Acer Dry 1

P11 Quercus Wet 7

P12 Quercus Unknown 2

Use subset() to

1. Make an object containing only information for Quercus
2. Make an object containing only information for Quercus in dry soil types
3. Make an object containing only information for Quercus in dry and wet soil types
4. Make an object containing only information for Quercus in dry and Acer in dry and wet soil types
5. Make an object containing only the soil type and cation values for Acer
6. Make a vector of cation values for dry soil type

## MY CODE: Part 2:

setwd("E:/Introduction to R")  
mydata <- read.csv2("myData.csv")  
  
#II.1  
#a) Calculate the mean, SD and sample size of length for each colour in myData  
  
#two ways:  
 #using vectors  
tapply(mydata$length, mydata$colour, mean)

## blue red   
## 9.166667 10.000000

tapply(mydata$length, mydata$colour, sd)

## blue red   
## 1.722401 3.559026

tapply(mydata$length, mydata$colour,length)

## blue red   
## 6 4

#using formula  
aggregate(length~colour, mydata, mean)

## colour length  
## 1 blue 9.166667  
## 2 red 10.000000

aggregate(length~colour, mydata, sd)

## colour length  
## 1 blue 1.722401  
## 2 red 3.559026

aggregate(length~colour, mydata, length)

## colour length  
## 1 blue 6  
## 2 red 4

#Way longer routes to the same results:  
 #Two alternative ways to extract the specific colours from the matrix  
blue <- subset(mydata,colour=="blue")  
red <- subset(mydata, colour == "red")  
  
red <- mydata[mydata$colour == "red", ]  
blue <- mydata[mydata$colour == "blue", ]  
  
mean(blue$mass)

## [1] 1.316667

mean(red$mass)

## [1] 1.525

sd(blue$mass)

## [1] 0.194079

sd(red$mass)

## [1] 0.35

length(blue$mass)

## [1] 6

length(red$mass)

## [1] 4

#b) Use aggregate() to calculate the means of both mass and length in  
myData for each colour in one command line  
  
aggregate(cbind(mass, length)~colour, mydata, mean)

## colour mass length  
## 1 blue 1.316667 9.166667  
## 2 red 1.525000 10.000000

#in the help page under formula: "y ~ x or cbind(y1, y2) ~ x1 + x2"  
  
  
#c) Code the construction of the following data.frame (10rows: 1:10.  
3columns: Trial, Experiment, Measure)  
  
data.frame <- data.frame(Trial=1:10, Experminet=c(rep("control", 5), rep("treatment", 5)), Measure = c(1,3,4,3,3,4,5,8,8,9))  
   
 #alternatively:  
Trial <- 1:10  
Experiment <- c(rep("control", 5), rep("treatment", 5))  
Measure <- c(1,3,4,3,3,4,5,8,8,9)  
dataframe <- data.frame(Trial, Experiment, Measure)  
  
#and use tapply() to calculate the mean, SD and sample size for each  
experimental group.  
  
tapply(dataframe$Measure, dataframe$Experiment, mean)

## control treatment   
## 2.8 6.8

tapply(dataframe$Measure, dataframe$Experiment, sd)

## control treatment   
## 1.095445 2.167948

tapply(dataframe$Measure, dataframe$Experiment, length)

## control treatment   
## 5 5

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#II.2   
#Write a script which   
#(a) imports the datafile “ExperimentalData.txt” which you can find in  
Moodle  
  
expdata <- read.csv("ExperimentalData.txt")  
  
  
#(b) Performs some checks the data import was successful  
  
View(expdata)  
summary(expdata)

## group size id   
## Min. :1.00 Min. : 2.032 Min. : 1.0   
## 1st Qu.:1.75 1st Qu.: 8.158 1st Qu.: 250.8   
## Median :2.50 Median : 9.837 Median : 500.5   
## Mean :2.50 Mean :10.057 Mean : 500.5   
## 3rd Qu.:3.25 3rd Qu.:11.701 3rd Qu.: 750.2   
## Max. :4.00 Max. :22.262 Max. :1000.0

names(expdata)

## [1] "group" "size" "id"

head(expdata)

## group size id  
## 1 1 7.627971 1  
## 2 1 6.993920 2  
## 3 1 8.057788 3  
## 4 1 10.422756 4  
## 5 1 10.576236 5  
## 6 1 6.131685 6

tail(expdata)

## group size id  
## 3995 4 9.314158 995  
## 3996 4 10.306628 996  
## 3997 4 15.241052 997  
## 3998 4 15.320831 998  
## 3999 4 10.698038 999  
## 4000 4 13.144118 1000

length(expdata)

## [1] 3

dim(expdata)

## [1] 4000 3

#(c) Recodes the 4 groups to:  
# group Feeding treatment Parasite treatment  
# 1 not fed removed  
# 2 not fed retained  
# 3 fed removed  
# 4 fed retained  
  
expdata$Feeding\_treatment <- ifelse(expdata$group <= 2, "not fed", "fed")  
expdata$Parasite\_treatment <- ifelse(expdata$group == 1 | expdata$group == 3, "removed", "retained")  
  
  
#(d) Extracts all data where the parasite was removed  
  
 #two ways:  
removed <- subset(expdata, Parasite\_treatment == "removed")  
removed <- expdata[expdata$Parasite\_treatment == "removed",]  
  
  
#(e) Calculates the variance for all 4 experimental groups  
  
aggregate(size~group, data=expdata,FUN=var)

## group size  
## 1 1 2.348075  
## 2 2 3.970370  
## 3 3 6.445510  
## 4 4 9.728437

#(f) Calculates the mean for “fed” and “not fed” treatment groups  
  
aggregate(size~Feeding\_treatment, data=expdata, FUN=mean)

## Feeding\_treatment size  
## 1 fed 11.124401  
## 2 not fed 8.989453

#Part2.2, create a script  
  
 #reading in data:  
expdata <- read.csv("ExperimentalData.txt")  
 #checks for successfull data import:  
expdata

## group size id  
## 1 1 7.627971 1  
## 2 1 6.993920 2  
## 3 1 8.057788 3  
## 4 1 10.422756 4  
## 5 1 10.576236 5  
## 6 1 6.131685 6  
#..... (Note 9.3: I removed the rest of the expdata listed from this  
compendium)

summary(expdata)

## group size id   
## Min. :1.00 Min. : 2.032 Min. : 1.0   
## 1st Qu.:1.75 1st Qu.: 8.158 1st Qu.: 250.8   
## Median :2.50 Median : 9.837 Median : 500.5   
## Mean :2.50 Mean :10.057 Mean : 500.5   
## 3rd Qu.:3.25 3rd Qu.:11.701 3rd Qu.: 750.2   
## Max. :4.00 Max. :22.262 Max. :1000.0

names(expdata)

## [1] "group" "size" "id"

head(expdata)

## group size id  
## 1 1 7.627971 1  
## 2 1 6.993920 2  
## 3 1 8.057788 3  
## 4 1 10.422756 4  
## 5 1 10.576236 5  
## 6 1 6.131685 6

tail(expdata)

## group size id  
## 3995 4 9.314158 995  
## 3996 4 10.306628 996  
## 3997 4 15.241052 997  
## 3998 4 15.320831 998  
## 3999 4 10.698038 999  
## 4000 4 13.144118 1000

#Creating two new columns for Feeding treatment and Parasite treatment:  
 The first part of the code creates a new column to the dataframe,  
 the second part defines which rows in the new columns should be  
 fed/not fed or removed/retained.  
expdata$Feeding\_treatment <- ifelse(expdata$group <= 2, "not fed", "fed")  
expdata$Parasite\_treatment <- ifelse(expdata$group == 1 | expdata$group == 3, "removed", "retained")  
  
 #Extracts the data where the parasites were removed:  
removed <- subset(expdata, Parasite\_treatment == "removed")  
  
 #variance for the experimental groups:  
aggregate(size~group, data=expdata,FUN=var)

## group size  
## 1 1 2.348075  
## 2 2 3.970370  
## 3 3 6.445510  
## 4 4 9.728437

#Mean for fed/not fed:  
aggregate(size~Feeding\_treatment, data=expdata, FUN=mean)

## Feeding\_treatment size  
## 1 fed 11.124401  
## 2 not fed 8.989453

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#II.3   
#Use the data in "Fruit\_tasting\_data\_2018.txt" from Moodle  
  
fruits <- read.table("Fruit\_tasting\_data\_2018.txt")  
  
  
#a) Create four additional variables: score1.apples and score2.apples  
which hold score1 and score2 respectively only for apples, and  
score1.strawberies and score2.strawberries which contain score1 and score2 respectively but only for strawberries.  
  
fruits$"score1.apples" <- ifelse(fruits$Fruit == "apple",  
 fruits$Score1, NA )  
fruits$"score2.apples" <- ifelse(fruits$Fruit == "apple",  
 fruits$Score2, NA )  
fruits$"score1.strawberries" <- ifelse(fruits$Fruit == "strawberry",  
 fruits$Score1, NA )  
fruits$"score2.strawberries" <- ifelse(fruits$Fruit == "strawberry",  
 fruits$Score2, NA )  
  
  
#b) Calculate the mean for each of these four variables  
   
mean(fruits$score1.apples, na.rm = TRUE)

## [1] 3.11

mean(fruits$score2.apples, na.rm = TRUE)

## [1] 3.7

mean(fruits$score1.strawberries, na.rm = TRUE)

## [1] 3.9125

mean(fruits$score2.strawberries, na.rm = TRUE)

## [1] 3.8125

#Two alternative ways  
colMeans(fruits[,5:8], na.rm = TRUE)

## score1.apples score2.apples score1.strawberries   
## 3.1100 3.7000 3.9125   
## score2.strawberries   
## 3.8125

colMeans(fruits[,c("score1.apples","score2.apples","score1.strawberries","score2.strawberries")], na.rm = TRUE)

## score1.apples score2.apples score1.strawberries   
## 3.1100 3.7000 3.9125   
## score2.strawberries   
## 3.8125

#c) Create two datasets. One for apples and one for strawberries  
  
 #two ways of extractiong the data  
   
apples <- fruits[fruits$Fruit == "apple", ]  
apples <- subset(fruits, Fruit=="apple")  
 #if you want to exclude some columns, you could use "select", like this:  
apples <- subset(fruits, Fruit=="apple", select = c(Variety, Fruit,Score1,Score2,score1.apples,score2.apples))  
  
strawberries <- fruits[fruits$Fruit == "strawberry",]  
strawberries <- subset(fruits, Fruit=="strawberry")  
  
#d) Make an object of class numeric containing score1 values for apples  
  
as.numeric(applescore1 <- fruits$score1.apples)

## [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA 2.1  
## [18] 3.7 3.1 3.8 4.2 3.2 2.3 1.3 3.0 4.4

#alternative + "drop = TRUE", which gives the values in a row/vector :  
applescore1 <- subset(fruits, Fruit == "apple", select = Score1,  
 drop=TRUE)  
class(applescore1)

## [1] "numeric"

#e) Make an object of class character containing the name of each fruit tasted in the dataset  
  
as.character(fruitname <- fruits$Variety)

## [1] "Bounty" "Sonata" "Polka" "Malwina"   
## [5] "Salsa" "Rumba" "Allstar" "Florence"   
## [9] "Mae" "Latestar" "Lowanna" "Mira"   
## [13] "Monophylla" "Northeastern" "Sara" "Maradesbois"   
## [17] "Lobo" "Amorosa" "Santana" "Discovery"   
## [21] "Rubinola" "Zari" "Jaspi" "Fuji"   
## [25] "Golden" "Pinklady"

#if not changed, factorial  
  
  
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#II.4  
#Write a script which  
#a) defines an object that is a matrix with 10 columns containing the numbers 1 to 100  
  
matrix100 <- matrix(1:100, ncol = 10)  
  
#b) defines another object that is a matrix with 10 columns containing the numbers 1 to 100 such that the numbers increment by 1 for each row across the column (i.e. row 1 is 1, 2, 3, etc)  
  
matrix100\_2 <- matrix(1:100, ncol = 10, byrow = TRUE)  
  
  
#c) names the rows and columns of these two objects  
  
colnames(matrix100)<- colnames(matrix100, do.NULL=FALSE, prefix = "row")  
rownames(matrix100)<- rownames(matrix100, do.NULL=FALSE, prefix = "row")  
  
rownames(matrix100\_2)<- colnames(matrix100, do.NULL=FALSE, prefix = "row")  
colnames(matrix100\_2)<- rownames(matrix100, do.NULL=FALSE, prefix = "row")  
  
  
#d) Extracts the third row of each object  
  
matrix100[3,]

## row1 row2 row3 row4 row5 row6 row7 row8 row9 row10   
## 3 13 23 33 43 53 63 73 83 93

matrix100\_2[3,]

## row1 row2 row3 row4 row5 row6 row7 row8 row9 row10   
## 21 22 23 24 25 26 27 28 29 30

#a) Calculates the mean of each column in each object (TIP: Find functions related to the function mean() from R’s help)  
  
colMeans(matrix100)

## row1 row2 row3 row4 row5 row6 row7 row8 row9 row10   
## 5.5 15.5 25.5 35.5 45.5 55.5 65.5 75.5 85.5 95.5

colMeans(matrix100\_2)

## row1 row2 row3 row4 row5 row6 row7 row8 row9 row10   
## 46 47 48 49 50 51 52 53 54 55

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#II.5  
#b) Create the following data.frame  
  
#I saved the text in a textfile "Secies.txt", and imported it.  
Species <- read.table("Species.txt", header = TRUE)  
Species

## Plot Species SoilType CationValue  
## 1 P1 Acer Wet 10  
## 2 P2.2 Acer Wet 12  
## 3 P2.3 Quercus Dry 4  
## 4 P3 Quercus Dry 3  
## 5 P10 Acer Dry 1  
## 6 P11 Quercus Wet 7  
## 7 P12 Quercus Unknown 2

#Use subset() to   
#c) Make an object containing only information for Quercus  
  
Quercus <- subset(Species, Species == "Quercus")  
  
#d) Make an object containing only information for Quercus in dry soil  
types  
  
Quercus\_dry <- subset(Species, Species == "Quercus" & SoilType == "Dry")  
  
#e) Make an object containing only information for Quercus in dry and wet soil types  
  
Quercus\_drywet <- subset(Species, Species == "Quercus" & SoilType == "Dry" | Species == "Quercus" & SoilType == "Wet")  
  
 #alternative: != means "not equal to". Th code below works as long as  
 the only remaining soil types are wet & dry  
Quercus\_drywet <- subset(Species, Species == "Quercus" & SoilType != "Unknown")  
  
#f) Make an object containing only information for Quercus in dry and Acer in dry and wet soil types  
  
QuercusAcer <- subset(Species, Species == "Quercus" & SoilType == "Dry" | Species == "Acer" & SoilType == "Dry" | Species == "Acer" & SoilType == "Wet" )  
 #The code above gives an unnecessary specification since there is no  
 other soiltypes than wet & dry  
  
 #So, alternative way:  
QuercusAcer <- subset(Species, Species == "Quercus" & SoilType == "Dry" | Species == "Acer")  
  
  
#g) Make an object containing only the soil type and cation values for  
Acer  
  
Acer\_S\_C <-subset(Species, Species == "Acer", c(SoilType, CationValue) )  
  
  
#h) Make a vector of cation values for dry soil type  
  
DryCations <- subset(Species, SoilType == "Dry", c(CationValue),  
 drop = TRUE)  
 #drop = TRUE cancels the default which will print the values as a list  
 instead of a vertical vector (row)

# Part III

In this part we focus on workflow operations 3a (data manipulation) and 5a (plotting a high-quality graph) and 5b (getting descriptive statistics out of R). In the previous parts, the help page was printed as an aid in various situations. From here on, this information is shown less frequently. While working through the compendium, it is a good idea to check the help pages.

## Chapter 8. First “pretty plot” of analysis

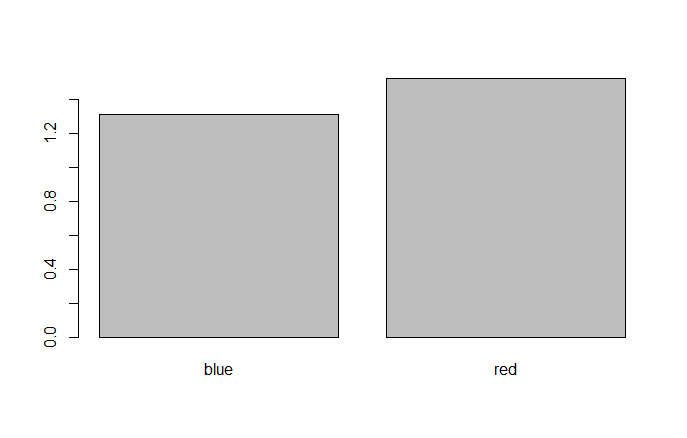
Producing high-quality graphs of your data is a major incentive to stick to the R workflow (Figure 5.0). One such workflow includes (operation 1) reading in the data, (operation 2) checking the data, (operation 3) filtering of data, (operation 4) calculating statistics from these data. All these are introduced in the previous chapters. We here focus on (operation 5a) plotting the descriptive statistics in a high-quality plot.

## 8.1 barplot()

To illustrate the plotting operation, the function barplot() is used, but as we will see in further chapters many of the coding translates to other plotting functions. As seen from its help page (Box 8.1), the minimal requirement for this function is information on the height of the bars to plot. We can, e.g. readily plot the means of differently coloured individuals by using the information on mean mass calculated using tapply() in section 7.2 above,

> barplot(means.per.colour)

which produces a fairly dull looking barplot (Fig. 8.1). Most importantly, the barplot lacks a label for the axes, although it has named the bars correctly. Study of the help page shows that to specify the axes labels xlab and ylab are used for X and Y axes respectively. It may also be good to increase the font size of the names and numbering of the axes (cex.axis, cex.names). The argument cex works as a multiplication factor of the default font size such that specification of cex>1 increases the font size. By running

Figure 8.1.1 Mean mass of blue and red individuals plotted using the default setting of barplot()

> barplot(means.per.colour,xlab=list("Colour",cex=1.5), ylab = list("Mass (g)",cex=1.5),cex.names=1.2,cex=1.2)

a better plot is obtained (Figure 8.1.2). In the above line of code, we specify that the font size of the labels of the X and Y axes are to be increased by 50% by providing the labels as an object of class list. We will look at list in more detail later on, but for now it is sufficient to realise that this list object contains both the name of the axes as a string, and the cex scaling parameter

> list("Colour",cex=1.5)

[[1]]

[1] "Colour"

$cex

[1] 1.5

Box 8.1. Help page for barplot (shortened)

|  |  |
| --- | --- |
| barplot {graphics} | R Documentation |

## Bar Plots

### Description

Creates a bar plot with vertical or horizontal bars.

### Usage

barplot(height, ...)

## Default S3 method:

barplot(height, width = 1, space = NULL,

names.arg = NULL, legend.text = NULL, beside = FALSE,

horiz = FALSE, density = NULL, angle = 45,

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

main = NULL, sub = NULL, xlab = NULL, ylab = NULL,

xlim = NULL, ylim = NULL, xpd = TRUE, log = "",

axes = TRUE, axisnames = TRUE,

cex.axis = par("cex.axis"), cex.names = par("cex.axis"),

inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,

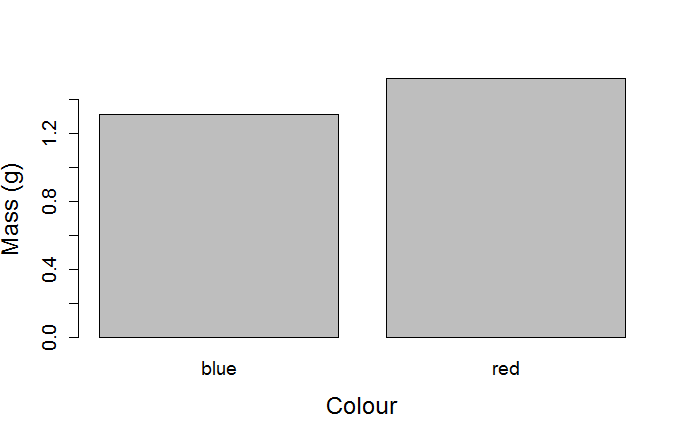
add = FALSE, args.legend = NULL, ...)

### Arguments

|  |  |
| --- | --- |
| height | either a vector or matrix of values describing the bars which make up the plot. If height is a vector, the plot consists of a sequence of rectangular bars with heights given by the values in the vector. If height is a matrix and beside is FALSE then each bar of the plot corresponds to a column of height, with the values in the column giving the heights of stacked sub-bars making up the bar. If height is a matrix and beside is TRUE, then the values in each column are juxtaposed rather than stacked. |
| width | optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will have no visible effect unless xlim is specified. |

----

The resulting bar plot (Figure 8.1.2) is now intelligible, but still looks quite grey. We can introduce colour into the plot by the argument col. We also computed more descriptive statistics; the standard deviation of differently coloured individuals as well as the sample sizes of these two subsets. We can add these pieces of information by using two additional plotting functions, segments() and text(). These functions place a line, respectively text in the X-Y plotting space defined by barplot(). However, the X-Y mapping is not clear from the figure: While we can more or less interpret the numbering of the Y-axis, we do not see the numbering of the X-axis. We can extract that information if we assign the barplot() to an object. The following line

Figure 8.1.2 Barplot of the mean mass of differently coloured individuals produced by overriding the default settings of barplot()

> barplot.X<- barplot(means.per.colour,xlab=list("Colour",cex=1.5), ylab = list("Mass (g)",cex=1.5),cex.names=1.2,cex=1.2)

> barplot.X

[,1]

[1,] 0.7

[2,] 1.9

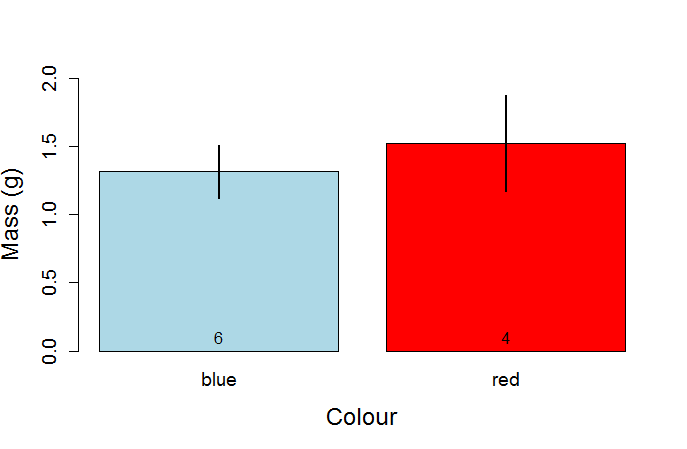
will produce the barplot (Figure 8.1.2) and also an object with the X coordinates of the middle of the bars plotted. That is, the middle of the bar for “blue” is plotted at X = 0.7, with the height of the bar being 1.32, i.e. the mean mass of blue individuals. To show the extent of the SD in relation to the mean, a line can be plotted using segments() through the middle of the bar going from the mean mass minus 1 SD to the mean mass + 1 SD for each subset. We increase the width of this line by passing the argument lwd=2 to segments(). We write the sample size in the bar using text(). Because the mean + 1 SD exceeds the maximal value of the Y axis as automatically determined by barplot(), new limits of the Y axis need to be provided using the argument ylim to barplot(). While we are at it, we also add some colour using the argument col to barplot(). The drawing of the “pimped” barplot (Figure 8.1.3) now takes three lines of coding

> barplot.X<-barplot(means.per.colour,xlab=list("Colour",cex=1.5), ylab = list("Mass (g)",cex=1.5),cex.names=1.2,cex=1.2,ylim=c(0,2), col=c("lightblue","red"))

> segments(barplot.X,means.per.colour- sd.per.colour, barplot.X, means.per.colour+sd.per.colour,lwd=2)

> text(barplot.X,0.1,n.per.colour)

We have now coded the plotting of a nice-looking graph. In particular, note that the plot was constructed in steps; lines and text were added to a barplot. This way of constructing a plot is a general aspect of plotting in R. Further adjustments can of course be made to the barplot as we have by no means introduced all possible modifications.

Figure 8.1.3 Barplot of mean mass of blue and red individuals with the standard deviation indicated by the line (+/1 1SD) and sample size of each subset printed above the X axis

We have now performed an entire workflow, from reading and checking the data (Part I) to the calculation of the descriptive statistics (Part II) to plotting these in high-quality graphs (this chapter). All these operations were done in R. While “coding a plot” may seem cumbersome to you, and likely represent initial investment in learning, remember that the coding is general. It will work, with no or minor adjustments, for other data to produce similarly good-quality figures. The use of a program dedicated to drawing figures will typically require also input by the user, quite often much “manual” input involving mouse clicking which does not leave a trace and therefore has to be done again when making new plots.

## 8.2 plot()

The function plot() was introduced in 4.4 as a quick graphical check or exploration of your data. Here we make the plot look nicer. A key aspect is that many of the aspects introduced in the previous section also apply to plot().

As in the barplot example, we want to use colours to separate the subsets. This was straightforward for the barplot as there were two bars and two colours. For plotting each data point, however, we need a vector containing the colour to use for each data point. In this case, we of course have this information in myData, the vector myData$colour, but in general we may not have written the subsets as colours (e.g. “control”, “treatment”) or we may want to use different colours. We can make use of the fact that myData$colour is a factorial vector which is, as we have seen, internally coded using 1’s and 2’s. We can check which is what by

> levels(myData$colour)

[1] "blue" "red"

Thus, blue individuals are coded as 1 and red individuals as 2. Remember that R will code the factors in alphabetic order (see 3.4) so this coding is as it should be. The vector

> myData$colour

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

Levels: blue red

therefore becomes a vector of 1’s and 2’s when interpreted as a number

> as.numeric(myData$colour)

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

as was also seen in str(myData) (section 4.1). Because of this, we can construct a vector cl of the colours we want to use and use myData$colour for indexing this vector, where each element in cl is the colour we want to plot each point.

> cl=c("lightblue","red")

> cl[myData$colour]

[1] "lightblue" "lightblue" "red" "lightblue" "lightblue" "red" "red" "lightblue"

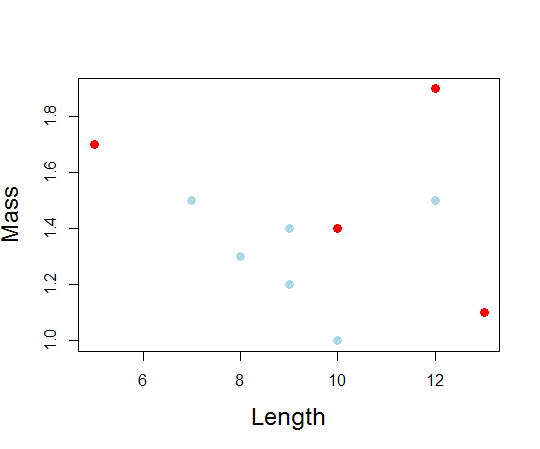
[9] "lightblue" "red"

Note in the above lines of code that we do not need to coerce the factorial vector myData$colour to a numerical one, but R automatically interprets this vector as 1’s and 2’s.

We then include cl[myData$colour] as the argument col to plot().

> plot(myData$mass~myData$length,col=cl[myData$colour],pch=19, cex=1.3, xlab=list("Length",cex=1.5), ylab=list("Mass",cex=1.5))

The above produces a plot of the two subsets distinguished by colour (Figure 8.2). It uses the pch argument to draw filled circles and the cex argument to increase the size of the filled circles. As before, xlab and ylab arguments define the labels and their font size.

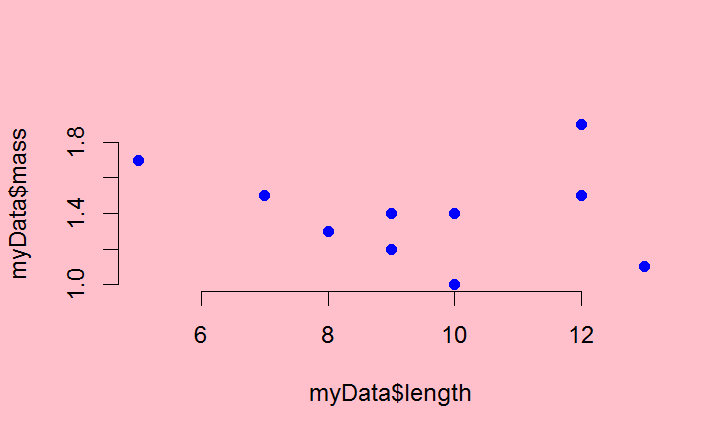
Figure 8.2 Mass plotted against length for individuals of blue and red colour as indicated by the respective coloration of the plotted dots.

## 8.3 Graphical parameters: find your way in the maze of options

Plots in R are highly adjustable. The help page for par() contains lots of information that can help you improve the quality of your plots considerably. The function par() itself is used to set graphical parameters, and there are many! The idea is that by setting these graphical parameters you can personalize (i.e. change the default) of plots. For example, I can decide I want all my plots to have increased font size by 50%, a filled dot, a pink background, plot in blue and not draw the square box around the plot. I can set these graphical parameters and the same plot() command for my “quick and dirty” plot (Figure 4.4) will look very different (Figure 8.3).

> par(cex=1.5, pch=19, bg="pink", col="blue",bty="n")

> plot(myData$mass~myData$length)

Figure 8.3 Quick graphical check plot of chapter 4.4 drawn after overriding the default grpahical parameters using par()

Because I have now set these parameters, they will affect the way all my plots will look. For example, also my quick boxplot() (Figure 4.3) will now be plotted on a pink background. This may not be what you want. It is typically more useful to specify (most) graphical parameters when executing the plot command. Luckily, except for a few, almost all graphical parameters in this long list can be used directly in the various plotting functions and you do not need to specify them prior to plotting using par(). Nevertheless, you can find information on how to “pimp your plot” from the help page of par(). For example, the arguments cex, pch, lwd used above are all in the help on par(). Remember that you can in R Studio’s also search within the help page (“Find Topic”). For example, you can search for “dashed” on the help page of par() to find out how to create a dashed line. The R Reference Sheet is also helpful for this purpose.

Similarly, the help page for the function points() has an overview of how you can change the default open circle to various symbols (argument pch). This function allows you to overlay a second plot on top of a first plot. Again, even when you do not use this function, you can use the pch argument in plot() as above.

The function colours() provides a list of all named colours when you run this command. Using a named colour is the easiest way to modify the colour in your plot. Note that multiple colours have a number after their name (e.g. “red”, “red2”, “red3”, …), where higher numbers mean a darker colour. In contrast, the grey scale works such that higher numbers refer to lighter grey (“grey80” is lighter than “grey20”). The help page for this function (or the “Color Specification” section at the bottom of the help for par() details how you can obtain even more options.

As you become more familiar with working in R, you will quickly become more accustomed to the many graphical parameters and various options available.

## 8.4 Exporting figures from R Studio

When you have made a plot, it shows in the plot tab in the left-hand bottom panel in R Studio. Above the plot, R Studio has a couple of buttons. You can move back and forward using the arrows and use the button “Export” to easily export your figure as a graphical file (multiple options are available from the drop-down menu), as a PDF or copy-paste the figure into e.g. a Word document.

You can also use the function pdf() to generate a PDF file of your figure. The PDF format is highly flexible (see its help page), and works across platforms. You can print Fig. 8.2 to your working directory as a PDF file

> pdf(file="Fig8\_3.pdf",paper="a4")

> plot(myData$mass~myData$length,col=cl[myData$colour],pch=19, cex=1.3, xlab=list("Length",cex=1.5), ylab=list("Mass",cex=1.5))

> dev.off()

null device

1

The call to function pdf() opens a named file (or creates this file if it does not exist) in your working directory. The argument paper specifies what kind of paper size you use (handy if you want to print it; here “a4”, but “letter” could be used in USA locale). You then execute the plot() code (can be multiple lines of code) that creates your plot. You must end with the function dev.off() which terminates the connection with the file (without this command, you cannot open the file).

## Chapter 9. Coding an own function

## 9.1.1 function()

A function carries out a specific set of instructions and returns the result. For example, sum() and colSum() are functions (to sum all up all elements or to sum up only part of the elements, respectively). You can program a function yourself in R using function(). For example, you can write a function to Z-standardize a data vector called Zstand: In script notation

Zstand<-function(x) {

mean.x<-mean(x)

sd.x<-sd(x)

z.x<-(x-mean.x)/sd.x

return(z.x) #writing return is needed if an object is defined

} #function(x)

where R Studio will automatically indent the code within curly brackets {…}. Note that I here annotate what the closing curly bracket is closing (#function(x)). This is of course not needed, but a habit which may be useful when code gets more complicated. When you place your cursor on any kind of bracket, R Studio will highlight its corresponding bracket (e.g. curly bracket close } and open {).

Running these lines produces in the Console

> Zstand<-function(x) {

+ mean.x<-mean(x)

+ sd.x<-sd(x)

+ z.x<-(x-mean.x)/sd.x

+ return(z.x) #writing return is needed if an object is defined

+ } #function(x)

> class(Zstand)

[1] "function")

where the +’s indicate that the Console is waiting for further lines of code.

By the way, if you ever get stuck in this situation where R is waiting (showing +) whereas you expect it to be ready (showing >), move to the Console (place your cursor there and mouse click) and press “Esc” (escape) or press the “Stop” button in R Studio’s Console panel if available.

The code above starts by specifying we want to make an object called Zstand() which is a function of x (the input variable). What is written between the curly brackets ‘{ … }’ is a little program which specifies what Zstand() does, and is here ended with the function return() specifying what the object Zstand() will return when called; in this case z.x which is the Z-standardised value of x. For a vector v, calling Zstand(v) thus returns the Z-standardised values of v.

> Zstand(c(1,2,3,4))

[1] -1.1618950 -0.3872983 0.3872983 1.1618950

where the Z-standardised values are printed in the Console. For R to remember the values it should be assigned to an object.

> z.x-Zstand(c(1,2,3,4))

As indicated in the annotation above, we do not need to include a the return() function, which would make the script look like

Zstand<-function(x) {

mean.x<-mean(x)

sd.x<-sd(x)

(x-mean.x)/sd.x

} #function(x)

## 9.1.2 Note on coding

When starting with coding, a good rule is “if it works, it works”. Note, however, that the function Zstand() as defined above contains many redundant temporary vectors. Vectors mean.x and sd.x are defined explicitly, but used only once. The coding of this function can thus be reduced to be more compact by not writing out these “help” vectors. That is, we can equivalently code

Zstand<-function(x) {

z.x<-(x- mean(x))/ sd(x)

return(z.x)

} #function(x)

which can be reduced even further by removal of return()

Zstand<-function(x) {

(x- mean(x))/ sd(x)

} #function(x)

This situation is fairly typical when coding. The first version of Zstand() (in 9.1.1) has the advantage that it breaks down the code into its smallest possible steps: (1) define the mean, (2) define the standard deviation, (3) calculate the input vector minus mean and divide by standard deviation. The second and last version of Zstand() (in 9.1.2) performs these steps in one go. The last version is clearly very compact and still very intelligible given that we use functions with intuitive names like mean() and sd(). Compact coding, and especially avoiding to define redundant objects are important in advanced applications involving either many calculations and/or Big Data or similar. For many applications, however, the compactness of your code is largely irrelevant. When starting out, you should primarily strive to code in a way that is intelligible and natural to you, so you understand what your code does. As you proceed in coding your solutions, and gain confidence you likely find that your code is becoming more compact.

## 9.2 Default arguments in a function

To better understand how R uses default values in functions, it is instructive to make an own function which contains a default. The function below returns the value(s) given minus a desired value. We can when setting up the function create a default value to use as “minus value”, which we define as myMinusValue=1. This means that unless the user specifies another value, this default value will be used. The default is specified within function().

> minusValue<-function(x,myMinusValue=1) {

+ minus.x<-x-myMinusValue

+ return(minus.x)

+ } #function(x,myMinusValue=1)

Thus, if we do not specify the argument myMinusValue, our function will assume that myMinusValue=1.

> minusValue(2)

[1] 1

We have to override the default if we want the function minusValue() to use another value than 1

> minusValue(2,2)

[1] 0

Another version of the above coded more compactly is

> minusValue<-function(x,myMinusValue=1) {

+ x-myMinusValue

+ } #function(x,myMinusValue=1)

## 9.3 Example: coding a function to calculate sample size with missing values

There are a number of handy basic functions to get at properties of your data. One such function is unique(), which works both for numbers and strings. It returns which entries in a vector are unique (i.e. it ignores the repeats). For example,

> vec<-c(1,2,2,3,3,4,4,5,5,6,7,7,8,8,9,10)

> unique(vec)

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

> vec.s<-c("a","b","b","c","b","a","c")

> unique(vec.s)

[1] "a" "b" "c"

You can count how many entries there are in a vector, by using length(). Thus,

> length(vec)

[1] 16

> length(unique(vec))

[1] 10

where the former gives the total number of elements in the vector and the latter the number without the repeats.

Note that the use of length() means you are counting the number of elements in a vector. As explained above, NA (missing values) are counted as elements (because they are!). Thus

> length(c(1,2,3))

[1] 3

> length(c(1,NA,3))

[1] 3

In the latter case, the sample size is clearly not 3 and thus using length() to obtain your sample size (as in chapter 7) produces erroneous results. When there are missing values, you need to remove these if you want to calculate sample size. Sadly, na.rm=TRUE which we encountered for the function mean() is not an argument to length(). There are multiple ways.

One is to use the fact that logical vectors (vectors consisting of TRUE and/or FALSE) when summed produce a sum of all the elements which are TRUE. That is, a logical vector is – from this perspective – viewed as a binary vector of 1 (TRUE) and 0 (FALSE). To get the logical vector saying whether each element is NA or not, a number of approaches can be used. For example, complete.cases() (which creates a logical vector stating whether there is an NA or not), or is.na() which creates a logical vector stating whether each element is NA (TRUE) or not (FALSE) which then of course needs to be reversed using !

> complete.cases(c(1,NA,3))

[1] TRUE FALSE TRUE

> sum(complete.cases(c(1,NA,3)))

[1] 2

> !is.na(c(1,NA,3))

[1] TRUE FALSE TRUE

> sum(!is.na(c(1,NA,3)))

[1] 2

Clearly, is.na() can be used to obtain the number of missing values

> sum(is.na(c(1,NA,3)))

[1] 1

We now have all the ingredients to code our own function to calculate sample size of any vector irrespectively of whether it holds NA or not. Such a function could look like

> sampleSize<-function(x) {

+ # function to calculate sample size stripping away missing values

+ sum(complete.cases(x))

+ } #function(x)

> sampleSize(c(1,2,NA,4))

[1] 3

To put the function into use we can consider the following data

> df.withNA<-data.frame(Trial=1:10, Experiment=c(rep("control",5),rep("treatment",5)),Measure=c(1,3,4,NA,3,4,5,NA,8,9))

> df.withNA

Trial Experiment Measure

1 1 control 1

2 2 control 3

3 3 control 4

4 4 control NA

5 5 control 3

6 6 treatment 4

7 7 treatment 5

8 8 treatment NA

9 9 treatment 8

10 10 treatment 9

where in two trials the measure failed resulting in a missing value (NA). To obtain the sample size per experimental subset, we use our new function within the tapply() function

> tapply(df.withNA$Measure,df.withNA$Experiment,sampleSize)

control treatment

4 4

This clearly counts the sample size correctly, as opposed to using length().

> tapply(df.withNA$Measure,df.withNA$Experiment,length)

control treatment

5 5

## 9.4 Anonymous function

The functions aggregate() and tapply() (and other members of the apply() family) are great tools. These functions apply a function on a subset of your data. As we saw above, the user can define the function as a named object of class function. Alternatively, if the function is really short and simple, you can create the function within the data extraction function itself. It has, in that case, no name and is therefore considered an *anonymous function*. For example,

> tapply(df.withNA$Measure,df.withNA$Experiment,function(x) sum(complete.cases(x)))

control treatment

4 4

where instead of giving within tapply()the argument sampleSize(), this function is written out. In doing so, the curly brackets are not needed.

Anonymous functions are not necessary in the sense that you can achieve the above equally well by first specifying the function explicitly (i.e. assign it to an object of class function, and then use this function in the call to aggregate()). Nevertheless, using anonymous functions allows for clearer coding (after some practice), as it is clear from the call to tapply() what is done without the need to go back in the script to see what sampleSize() does. In particular, anonymous functions are handy if two or perhaps three functions are to be combined. With two functions, sum() and complete.cases() we could obtain sample sizes excluding missing values. Another way is to combine the functions length() and na.omit() where the latter strips NA from an object. In the end, it is a matter of taste and style how you implement functions in these kind of situations.

## Chapter 10 Make a table and export it

In this chapter we focus on workflow operations 5b. In Part II and this Part, we have started exploring R’s powerful functions for applying functions over subsets (apply family) which can be used to compute descriptive statistics, as well as how to write purpose-specific functions. So far, this has only been printed on the Console, or then in R’s brain, perhaps even as different objects. Just as we can export our plots, we likely also want to combine the descriptive stats in a table and export this information. The two basic ways to convey information are, after all, figures and tables.

## 10.1 Table of descriptive statistics

Before doing more advanced data analysis, it is a good habit to explore your data. One aspect is to present descriptive statistics. As a simple approach, we can construct a data.frame which we fill with descriptive statistics on subsets of df.withNA. The function aggregate() is well suited for this purpose. Let’s start with the mean

> descriptiveStats.df.withNA<-aggregate(Measure~Experiment, data=df.withNA,mean)

> descriptiveStats.df.withNA

Experiment Measure

1 control 2.75

2 treatment 6.50

Notice that this call to aggregate() produces the mean measure despite there being NA values. The help page (Box 7.1) explains why: when using the formula method, aggregate has as default additional argument na.action=na.omit. Intuitively, this default means that the NA values are omitted prior to applying the function specified to the different subsets (performed by the function na.omit(); see also 9.4). What is made here is a data.frame

> str(descriptiveStats.df.withNA)

'data.frame': 2 obs. of 2 variables:

$ Experiment: Factor w/ 2 levels "control","treatment": 1 2

$ Measure : num 2.75 6.5

Thus, we can add columns (see 5.2.4) to this data.frame with more descriptive statistics

> #add n

> descriptiveStats.df.withNA<-data.frame(descriptiveStats.df.withNA, n=aggregate(Measure~Experiment, data=df.withNA,length)$Measure)

> #add sd

> descriptiveStats.df.withNA<-data.frame(descriptiveStats.df.withNA, sd=aggregate(Measure~Experiment, data=df.withNA,sd)[,2])

> names(descriptiveStats.df.withNA)[2]<-"mean"

> descriptiveStats.df.withNA

Experiment mean n sd

1 control 2.75 4 1.258306

2 treatment 6.50 4 2.380476

Note that each call to aggregate() will create a data.frame with the same structure; it has two columns whose tags are $Experiment and $Measure. We only want to add the descriptive stats of the measure ($Measure or [,2] as illustrated in lines above for n and sd respectively). Clearly we can add additional descriptive stats following the same logic.

We have seen that aggregate() can apply a function to multiple variables (using cbind(), see Box 7.1). Now, you may wonder whether we cannot also execute multiple functions at the same time. To do so we have to – as in chapter 9 – code our own function. We can combine the computation of multiple descriptive statistics:

> descriptiveStats.df.withNA.multi<-aggregate(Measure~Experiment, data=df.withNA,function(x) c(mean(x),length(x), sd(x)))

> #what did we make?

> str(descriptiveStats.df.withNA.multi)

'data.frame': 2 obs. of 2 variables:

$ Experiment: Factor w/ 2 levels "control","treatment": 1 2

$ Measure : num [1:2, 1:3] 2.75 6.5 4 4 1.26 ...

> descriptiveStats.df.withNA.multi

Experiment Measure.1 Measure.2 Measure.3

1 control 2.750000 4.000000 1.258306

2 treatment 6.500000 4.000000 2.380476

which clearly produces a rather odd object: A data.frame with its second “column” a 2 x 3 matrix which when printed to the Console, however, does produce the numbers we are after. While it is certainly possible to work further with this object (e.g. coerce the matrix to a data.frame) or with this approach in general, my preference is for the “line-by-line” option outlined above. It is tractable and produces the correct tags.

## 10.2 Export our summary table as a text file

We read text formatted data using read.table(), and can use the function write.table() to do the reverse. As a minimum, it requires as arguments the object to be written and the name of the file it is written to

> write.table(descriptiveStats.df.withNA,"DescriptiveStats.txt")

As a rule, I provide the extension “.txt”. The code above produces a text file which looks like

"Experiment" "Mean" "n" "sd"

"1" "control" 2.75 4 1.25830573921179

"2" "treatment" 6.5 4 2.38047614284762

(remember you can easily open text files in your R (Studio) editor). Overriding some of the defaults is a good idea. In particular, we can avoid the quotations marks by specifying quote=FALSE and omit the row numbers that mess up the alignment by specifying row.names=FALSE. We can also specify we want the fields to be separated by a semi-colon using sep.

> write.table(descriptiveStats.df.withNA,"DescriptiveStats.txt", quote=F,row.names = F,sep=";")

which produces

Experiment;mean;n;sd

control;2.75;4;1.25830573921179

treatment;6.5;4;2.38047614284762

When you copy-paste the above text into Word, you can convert it to a Table (Highlight the text and select *Insert*->*Table*->*Convert Text to Table*; select “Semicolons” under “Separate Text at”). The Table will then in its basic form looks like:

|  |  |  |  |
| --- | --- | --- | --- |
| Experiment | mean | n | sd |
| control | 2.75 | 4 | 1.25830573921179 |
| treatment | 6.5 | 4 | 2.38047614284762 |

This basic table can then be formatted further in your document.

## 10.3 Is this handy?

The approach (10.1) to create a data.frame by “picking” from objects what you want to have, adding these one by one can of course be applied to collect also other statistics. Results from statistical analyses, for example. The creation of this kind of tables by coding in R often feels quite cumbersome, especially when you want to combine many different statistics. In my experience, however, it is common to realise a (hopefully small) mistake was made at some point in the workflow. If the construction of your table was coded, you can then quickly update your table, at least compared to manually extracting the statistics you wanted to report. As a consequence, time spent in coding may be paid back to you with interest.

Another important aspect to remember is that even if the “line-by-line” coding quickly looks like a massive block of code, it is usually constructed of many repeats of the same line of code with only small alteration. For example in the above (10.1), we first create a start (with the mean per subset), and after that repeat the same procedure 2 times where only the name of the function is changed, but the lines of code are otherwise identical. Clearly, copy-pasting and carefully updating the name of the function allows one to efficiently create this kind of repetitive blocks of code.

## 10.4 Exercises part III

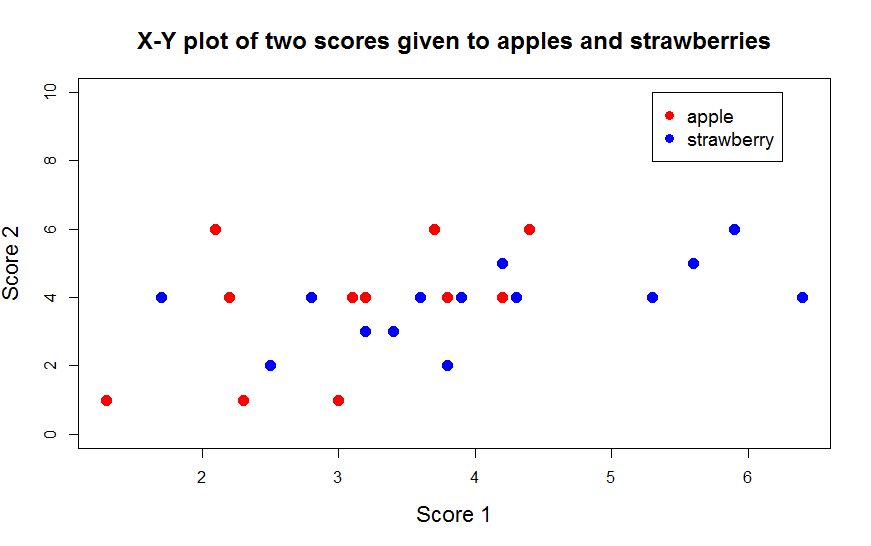
III.1

Use the data in "Fruit\_tasting\_data\_2018.txt" from Moodle

a.

* Plot “score 2” as a function of “score 1”
* Rename the axes
* Change the symbol plotted
* Increase point size of the symbol plotted
* Increase font size for the axes
* Change the y axis so that the minimum value is 0 and the maximum is 10
* Change the color of the points so that they are different for apples and for strawberries
* Add a text on top of the figure
* Add a legend

Resulting figure should look something like this:

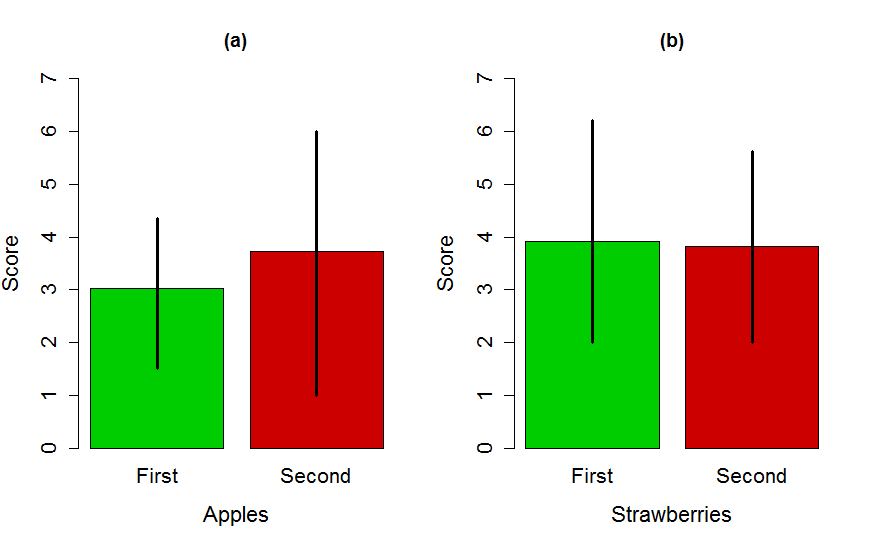


b. You obtain the extra scores 2.2 and 4 for the apple Gyllenkroks Astrakan. Add this to the data object in R. Redraw the plot

III.2

Use the data in "Fruit\_tasting\_data\_2018.txt" from Moodle

1. Calculate the mean and 95% confidence interval for score1 and score2 for apples and strawberries. A 95% confidence interval consists of 2 values. A lower and upper value. With 95% probability the data lies between these two values. The function quantile() is useful here.
2. Create a barplot of the mean score1 for apples and strawberries with a line indicating the 95% confidence interval (i.e. the line goes from the lower confidence to the upper confidence value, TIP: read carefully section 8.1 for this exercise). Make the plot look as good as you can.
3. Use par() to plot two panels in one graph and make a 2-panel plot showing the average score1 and score2 of apples and strawberries with their 95% confidence interval respectively. That is, one panel with a barplot of score1 and score2 of apples and the second panel showing barplot of score1 and score2 of strawberries. Something like this:



III.3

You learn from Wikipedia (<https://en.wikipedia.org/wiki/Standard_error>) that the standard error is defined as the standard deviation divided by the square root of the sample size.

1. Code a function called calc.SE() to calculate the standard error of any numerical vector passed to this function. By default, the function should not strip away NA elements in the vector (like in the function mean), but it should also be possible to override the default and keep the NA values.

Check that it works: You should obtain the following output

> calc.SE(1:10)

[1] 0.9574271

> calc.SE(c(1:10,NA))

[1] NA

> calc.SE(c(1:10,NA),na.rm = TRUE)

[1] 0.9574271

1. The following lines make a small data.frame

df.year<-data.frame(Year=c(rep(1,5),rep(2,5))

, Response=c(4.256898, 7.103097, 5.345175, 3.259264, 4.327387, 9.995211, 8.621862, 10.499859,10.032349, 8.821877))

Create this data.frame. Calculate the sample size, mean and SE of Response in both years in using both aggregate and tapply. Each of these calculations is ONE line of code (i.e. one line of code with aggregate to generate sample sizes per year, one to generate mean, etc..)

1. Some additional data is added which contains NA’s in your data.frame.

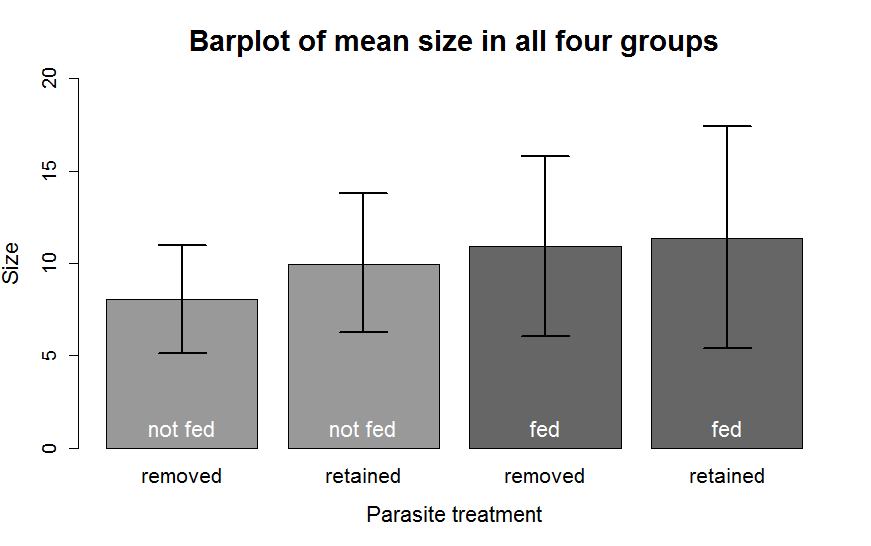
df.year<-rbind(df.year, data.frame(Year=c(rep(1,2),rep(2,2))

, Response=c(4.33, NA, 10.1, NA)))

Run these lines to add this data, and calculate the sample size, mean and SE per year using both aggregate and tapply. Each of these calculations is ONE line of code.

III.4

1. Extend your script of exercise II.2 to produce the following barplot of mean size +/- the 95% confidence interval in each of the experimental groups in greyscale (You can find the definition of 95% confidence interval in III.1). Make the graph look close to the graph below (note the way confidence intervals are plotted here; check R reference card for arrows()).



III.5

The following lines of code generate a small dataset

df.withNA<-data.frame(Trial=1:10, Experiment=c(rep("control",5),rep("treatment",5)),Measure=c(1,3,4,NA,3,4,5,NA,8,9))

df.withNA<- data.frame(df.withNA,Measure2=c(2,3,2,3,4,5,4,5,6,6),block=rep(c("B1","B2"),5))

which when printed looks like:

Trial Experiment Measure Measure2 block

1 1 control 1 2 B1

2 2 control 3 3 B2

3 3 control 4 2 B1

4 4 control NA 3 B2

5 5 control 3 4 B1

6 6 treatment 4 5 B2

7 7 treatment 5 4 B1

8 8 treatment NA 5 B2

9 9 treatment 8 6 B1

10 10 treatment 9 6 B2

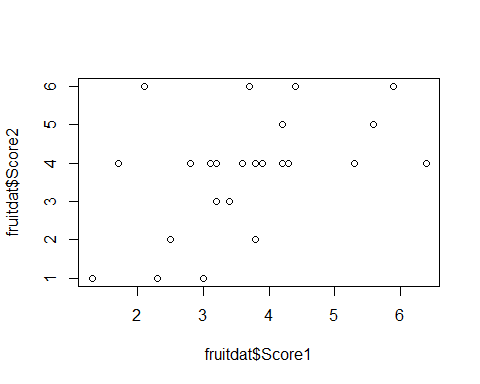
1. Use one of line of code with aggregate to compute how many different blocks there were per treatment group
2. Code the construction of a data.frame with the descriptive statistics sample size, mean and standard deviation for “Measure” for each of the four possible combinations of “Experiment” and “block”.
3. Code the construction of a data.frame with the descriptive statistics sample size, mean and standard deviation for “Measure2” for each of the four possible combinations of “Experiment” and “block”.
4. Code the construction of a data.frame with the descriptive statistics sample size, mean and standard deviation for “Measure” and “Measure2” using a single call to aggregate() for each of the four possible combinations of “Experiment” and “block”.
5. Are there differences between what you get in a,b,c? If so, why?

## MY CODE: Part 3:

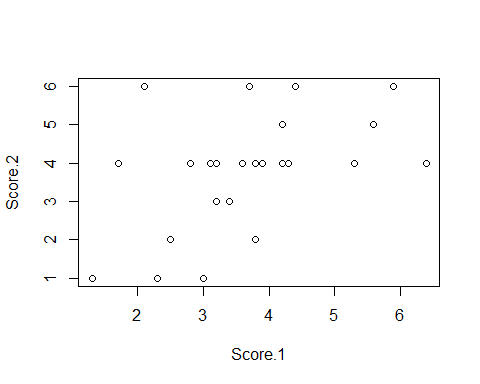
#III.1  
#Use the data in "Fruit\_tasting\_data\_2018.txt" from Moodle  
  
setwd("E:/Introduction to R")  
fruitdat <- read.table("Fruit\_tasting\_data\_2018.txt")  
head(fruitdat)

## Variety Fruit Score1 Score2  
## 1 Bounty strawberry 4.3 4  
## 2 Sonata strawberry 2.8 4  
## 3 Polka strawberry 3.6 4  
## 4 Malwina strawberry 3.9 4  
## 5 Salsa strawberry 3.2 3  
## 6 Rumba strawberry 3.2 3

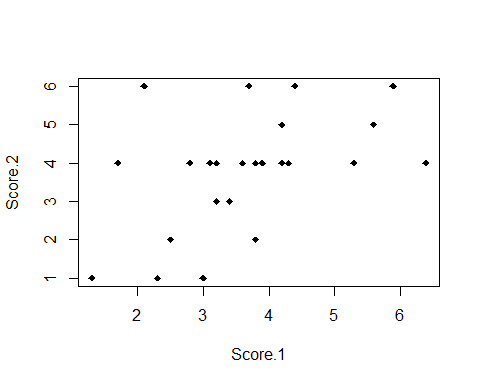
#a. Plot "score 2" as a function of "score 1"   
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2)



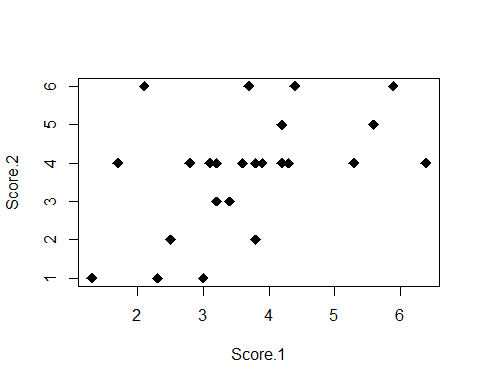
#Rename the axes  
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = "Score.1", ylab = "Score.2")



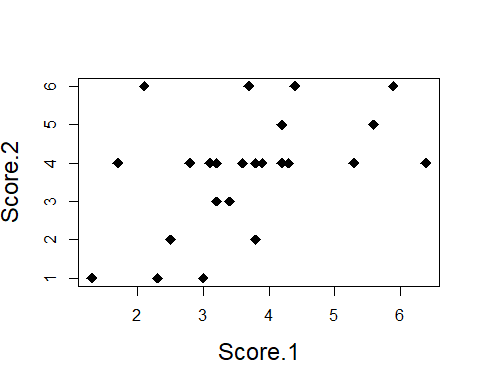
#Change the symbol plotted  
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = "Score.1", ylab = "Score.2",  
 pch = 18)



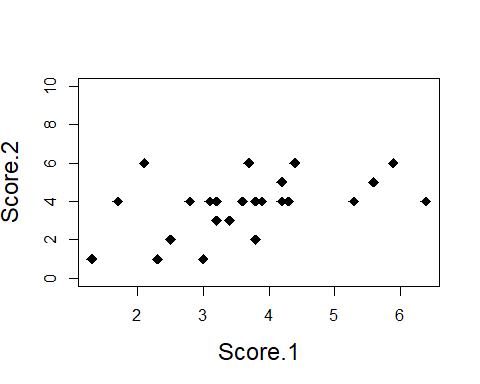
#Increase point size of the symbol plotted  
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = "Score.1", ylab = "Score.2",  
 pch = 18, cex=1.5)



#Increase font size for the axes  
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = list("Score.1", cex = 1.5),   
 ylab = list("Score.2", cex = 1.5),  
 pch = 18, cex=1.5)



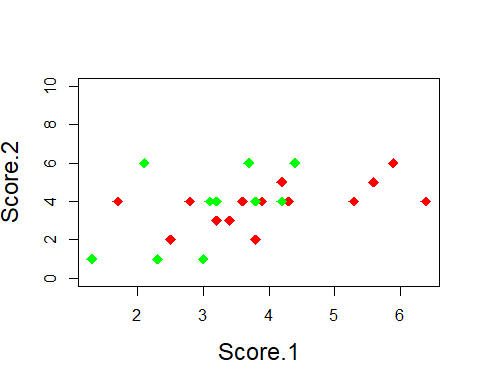
#Change the y axis so that the minimum value is 0 and the maximum is 10  
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = list("Score.1", cex = 1.5),   
 ylab = list("Score.2", cex = 1.5),  
 pch = 18, cex=1.5, ylim = c(0,10))



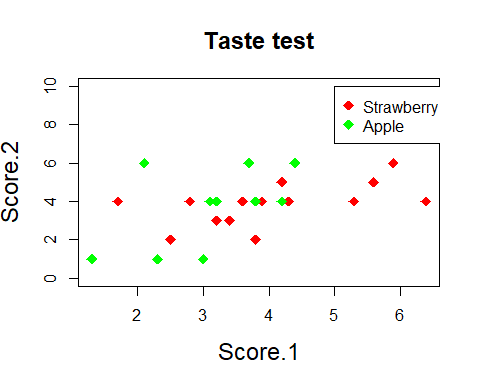
#Change the color of the points so that they are different for apples and for strawberries  
  
  
 #creating a vector with the wanted colours  
colors.vector <- c("green","red")  
  
 #applying that vector to the binary values -> it aplies the right colour to the right values!  
colors.vector[fruitdat$Fruit]

## [1] "red" "red" "red" "red" "red" "red" "red" "red"   
## [9] "red" "red" "red" "red" "red" "red" "red" "red"   
## [17] "green" "green" "green" "green" "green" "green" "green" "green"  
## [25] "green" "green"

#adding the colour argument as above ^  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = list("Score.1", cex = 1.5),   
 ylab = list("Score.2", cex = 1.5),  
 pch = 18, cex=1.5, ylim = c(0,10),  
 col = colors.vector[fruitdat$Fruit])



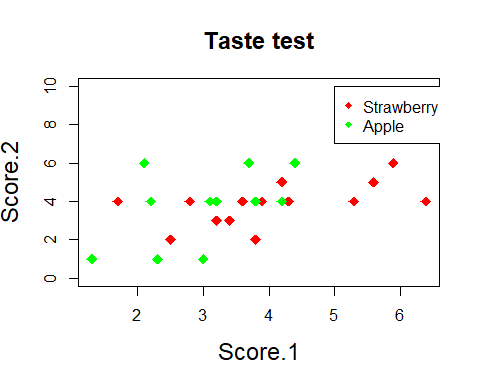
#you could also do it directly like this:  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = list("Score.1", cex = 1.5),   
 ylab = list("Score.2", cex = 1.5),  
 pch = 18, cex=1.5, ylim = c(0,10),  
 col = c("green","red")[fruitdat$Fruit])  
  
#Add a text on top of the figure  
  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = list("Score.1", cex = 1.5),   
 ylab = list("Score.2", cex = 1.5),  
 pch = 18, cex=1.5, ylim = c(0,10),  
 col = colors.vector[fruitdat$Fruit],   
 main = list("Taste test", cex = 1.5))  
  
#Add a legend  
  
legend(5, 10, c("Strawberry", "Apple"), pch = 18, col = c("red", "green"), pt.cex = 1.5)



#How would you increase the size of the symbols in the legend?  
 Answer in class: pt.cex = 1.5, found in the legend help page  
 pch = list(18, cex = 1.5) does not increase the size, but changes the  
 symbol for apples.  
  
  
#b. You obtain the extra scores 2.2 and 4 for the apple Gyllenkroks Astrakan. Add this to the data object in R. Redraw the plot   
  
fruitdat<-rbind(fruitdat,  
 data.frame(Variety="Gyllenkroks Astrakan",  
 Fruit="apple",Score1=2.2,Score2 =4))  
  
 #alternative:   
as.character(fruitdat$Variety)

## [1] "Bounty" "Sonata" "Polka"   
## [4] "Malwina" "Salsa" "Rumba"   
## [7] "Allstar" "Florence" "Mae"   
## [10] "Latestar" "Lowanna" "Mira"   
## [13] "Monophylla" "Northeastern" "Sara"   
## [16] "Maradesbois" "Lobo" "Amorosa"   
## [19] "Santana" "Discovery" "Rubinola"   
## [22] "Zari" "Jaspi" "Fuji"   
## [25] "Golden" "Pinklady" "Gyllenkroks Astrakan"

fruitdat<-rbind(fruitdat,c("Gyllenkroks Astrakan","apple",  
 Score1=2.2,Score2 =4))  
  
 #redrawing the plot  
myplot <- plot(fruitdat$Score1, fruitdat$Score2,  
 xlab = list("Score.1", cex = 1.5),   
 ylab = list("Score.2", cex = 1.5),  
 pch = 18, cex=1.5, ylim = c(0,10),  
 col = colors.vector[fruitdat$Fruit],   
 main = list("Taste test", cex = 1.5))  
legend(5, 10, c("Strawberry", "Apple"), col = c("red", "green"),  
 pch = c(18))



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#III.2  
#Use the data in "Fruit\_tasting\_data\_2018.txt" from Moodle   
  
fruitdat <- read.table("Fruit\_tasting\_data\_2018.txt")  
  
#(a) Calculate the mean and 95% confidence interval for score1 and  
score2 for apples and strawberries. A 95% confidence interval consists of 2 values. A lower and upper value. With 95% probability the data lies  
between these two values. The function quantile() is useful here.   
  
 #means  
aggregate(cbind(Score1, Score2) ~ Fruit , data=fruitdat, FUN=mean)

## Fruit Score1 Score2  
## 1 apple 3.1100 3.7000  
## 2 strawberry 3.9125 3.8125

#HOW WE DID IT IN CLASS THE RIGHT WAY, confindence interval:  
tapply(fruitdat$Score1, fruitdat$Fruit, quantile, probs=c(0.025, 0.975))

## $apple  
## 2.5% 97.5%   
## 1.480 4.355   
##   
## $strawberry  
## 2.5% 97.5%   
## 2.0000 6.2125

tapply(fruitdat$Score2, fruitdat$Fruit, quantile, probs=c(0.025, 0.975))

## $apple  
## 2.5% 97.5%   
## 1 6   
##   
## $strawberry  
## 2.5% 97.5%   
## 2.000 5.625

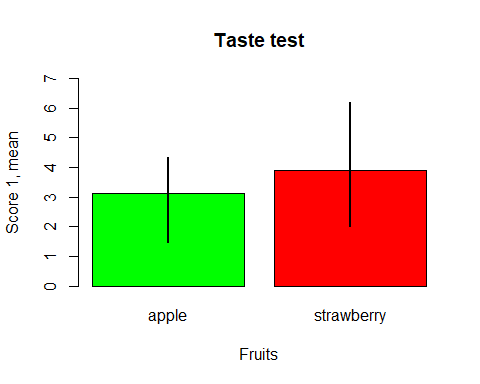
#(b) Create a barplot of the mean score1 for apples and strawberries  
with a line indicating the 95% confidence interval (i.e. the line goes  
from the lower confidence to the upper confidence value, TIP: read  
carefully section 8.1 for this exercise). Make the plot look as good as  
you can.  
  
 #creating an object with the means to use in the barplot() function  
means.per.fruit <- tapply(fruitdat$Score1,fruitdat$Fruit, mean)  
means.per.fruit

## apple strawberry   
## 3.1100 3.9125

mybarplot <- barplot(means.per.fruit, col = c("green", "red"),  
 main = "Taste test", xlab = "Fruits",  
 ylab = "Score 1, mean", ylim = c(0,7))  
  
 #crating y-coordinates (the lower and upper values of the interval) for the beginning and end of the lines to use in the segments() function  
  
lower.score1 <- tapply(fruitdat$Score1, fruitdat$Fruit, quantile,  
 probs=0.025)  
upper.score1 <- tapply(fruitdat$Score1, fruitdat$Fruit, quantile,  
 probs=0.975)   
  
 #(alternatively to get the quantiles for score1 per fruit, in one line  
 of code)  
  
tapply(fruitdat$Score1, fruitdat$Fruit, quantile, probs=c(0.025, 0.975))

## $apple  
## 2.5% 97.5%   
## 1.480 4.355   
##   
## $strawberry  
## 2.5% 97.5%   
## 2.0000 6.2125

#adding the line of the confidence intervals of 95%  
segments(mybarplot, lower.score1, mybarplot, upper.score1, lwd = 2)



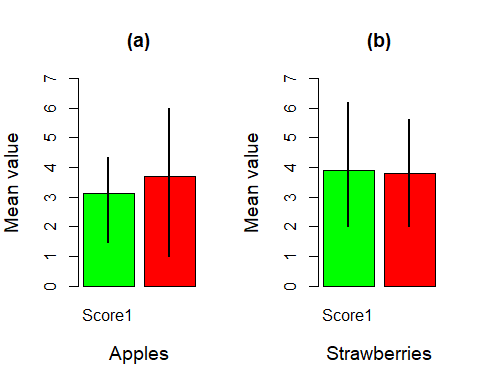
#(c) Use par() to plot two panels in one graph and make a 2-panel plot showing the average score1 and score2 of apples and strawberries with their 95% confidence interval respectively. That is, one panel with a barplot of score1 and score2 of apples and the second panel showing barplot of score1 and score2 of strawberries. Something like this:  
  
lower.score2 <- tapply(fruitdat$Score2, fruitdat$Fruit, quantile,  
 probs=0.025)  
upper.score2 <- tapply(fruitdat$Score2, fruitdat$Fruit, quantile,  
 probs=0.975)   
  
 #Confidence intervalls - the coordinates for the segments() function  
lower.apple <- c(lower.score1[1], lower.score2[1])  
upper.apple <- c(upper.score1[1], upper.score2[1])  
  
lower.straw <- c(lower.score1[2], lower.score2[2])  
upper.straw <- c(upper.score1[2], upper.score2[2])  
  
  
 #Getting the means of Score1 & Score2 for both fruits seperatly for the barplot() function  
dat.apples <- subset(fruitdat, Fruit == "apple",  
 select = c(Score1, Score2))  
  
apple\_means <- colMeans(dat.apples)  
apple\_means

## Score1 Score2   
## 3.11 3.70

dat.strawberries <- subset(fruitdat, Fruit == "strawberry",  
 select = c(Score1, Score2))  
  
strawberry\_means <- colMeans((dat.strawberries))  
strawberry\_means

## Score1 Score2   
## 3.9125 3.8125

#the key to getting two plots. The vector tells how many "rows" (1) and "columns" (2) of barplot I want.  
par(mfrow = c(1,2))  
  
 #Plotting Apples  
barplot.apple <- barplot(apple\_means, col = c("green", "red"),  
 main = "(a)", xlab = list("Apples", cex=1.2),  
 ylab = list("Mean value", cex=1.2), ylim = c(0,7))  
  
segments(barplot.apple, lower.apple, barplot.apple, upper.apple, lwd = 2)  
  
  
 #Plotting Strawberries  
barplot.strawberry <- barplot(strawberry\_means, col = c("green", "red"),  
 main = "(b)", xlab = list("Strawberries", cex=1.2),  
 ylab = list("Mean value", cex=1.2), ylim = c(0,7))  
  
segments(barplot.strawberry, lower.straw, barplot.strawberry,  
 upper.straw, lwd = 2)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
#III.3  
  
#a) Code a function called calc.SE() to calculate the standard error of any numerical vector passed to this function. By default, the function should not strip away NA elements in the vector (like in the function mean), but it should also be possible to override the default and keep the NA values.   
   
  
calc.SE <- function(x, na.rm = FALSE){  
 (sd(x, na.rm = na.rm))/(sqrt(length(x)))  
} #function(x, na.rm = FALSE)  
 #na.rm argument can be used since sd() has it built in. sd(x, na.rm =  
 na.rm) is like that, beacuase we want to functions na.rm to be na.rm in  
 our function.  
  
 #testing it out  
calc.SE(c(NA, 1:10))

## [1] NA

calc.SE(c(NA, 1:10), na.rm = TRUE)

## [1] 0.9128709

#b) The following lines make a small data.frame   
  
df.year<-data.frame(Year=c(rep(1,5), rep(2,5)),  
 Response=c(4.256898, 7.103097, 5.345175, 3.259264,  
 4.327387, 9.995211, 8.621862, 10.499859,  
 10.032349, 8.821877))  
  
#Create this data.frame. Calculate the sample size, mean and SE of Response in both years in using both aggregate and tapply. Each of these calculations is ONE line of code (i.e. one line of code with aggregate to generate sample sizes per year, one to generate mean, etc..)  
  
dataframe\_length <- tapply(df.year$Response, df.year$Year, length)  
dataframe\_mean <-tapply(df.year$Response, df.year$Year, mean)  
dataframe\_SE <-tapply(df.year$Response, df.year$Year, calc.SE)  
  
df.year2 <- cbind(dataframe\_length, dataframe\_mean, dataframe\_SE)  
df.year

## Year Response  
## 1 1 4.256898  
## 2 1 7.103097  
## 3 1 5.345175  
## 4 1 3.259264  
## 5 1 4.327387  
## 6 2 9.995211  
## 7 2 8.621862  
## 8 2 10.499859  
## 9 2 10.032349  
## 10 2 8.821877

dataframe\_length2<- aggregate(Response ~ Year, data=df.year, FUN = length)  
dataframe\_mean2<- aggregate(Response ~ Year, data=df.year, FUN = mean)  
dataframe\_SE2<- aggregate(Response ~ Year, data=df.year, FUN = calc.SE)  
  
df.year3<- cbind(dataframe\_length2, dataframe\_mean2, dataframe\_SE2)  
df.year3

## Year Response Year Response Year Response  
## 1 1 5 1 4.858364 1 0.6510234  
## 2 2 5 2 9.594232 2 0.3684377

#c) Some additional data is added which contains NA’s in your data.frame.   
  
df.year<-rbind(df.year, data.frame(Year=c(rep(1,2),rep(2,2))  
 , Response=c(4.33, NA, 10.1, NA)))   
  
#Run these lines to add this data, and calculate the sample size, mean and SE per year using both aggregate and tapply. Each of these calculations is ONE line of code.  
  
aggregate(Response ~ Year, data=df.year, FUN = length)

## Year Response  
## 1 1 6  
## 2 2 6

aggregate(Response ~ Year, data=df.year, FUN = mean, na.rm = TRUE)

## Year Response  
## 1 1 4.770303  
## 2 2 9.678526

aggregate(Response ~ Year, data=df.year, FUN = calc.SE, na.rm = TRUE)

## Year Response  
## 1 1 0.5388033  
## 2 2 0.3124150

tapply(df.year$Response, df.year$Year,function(x) length(na.omit((x))))

## 1 2   
## 6 6

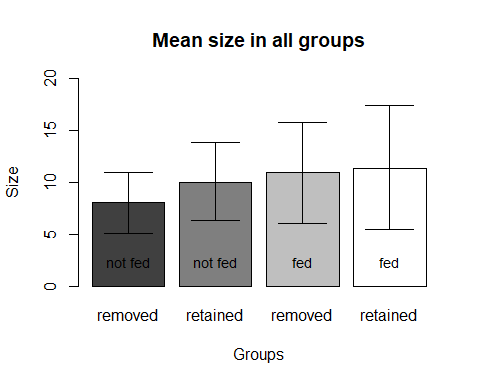
tapply(df.year$Response, df.year$Year, mean, na.rm = TRUE)

## 1 2   
## 4.770303 9.678526

tapply(df.year$Response, df.year$Year, calc.SE, na.rm = TRUE)

## 1 2   
## 0.4988349 0.2892401

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#III.4  
#a) Extend your script of exercise II.2 to produce the following barplot  
of mean size +/- the 95% confidence interval in each of the experimental   
groups in greyscale (You can find the definition of 95% confidence   
interval in III.1). Make the graph look close to the graph below (note the way confidence intervals are plotted here; check R reference card for   
arrows()).  
  
 #reading in data (and the extra columns)  
expdata <- read.csv("ExperimentalData.txt")  
expdata$Feeding\_treatment <- ifelse(expdata$group <= 2, "not fed", "fed")  
expdata$Parasite\_treatment <- ifelse(expdata$group == 1 | expdata$group == 3, "removed", "retained")  
  
 #coordinates for the arrows function  
lower\_expdata <- tapply(expdata$size, expdata$group,  
 quantile, probs=c(0.025))   
upper\_expdata <- tapply(expdata$size, expdata$group,  
 quantile, probs=c(0.975))   
  
 #group means  
means.per.group <- tapply(expdata$size, expdata$group, mean)  
  
 #One-panel view:  
par(mfrow = c(1,1))  
  
 #Barplot:  
barplot\_expdata <- barplot(means.per.group, col = c("gray25", "gray50",  
 "gray75", "gray100"),  
 main = "Mean size in all groups",  
 xlab = "Groups", ylab = "Size",  
 ylim = c(0,20),  
 names.arg=c("removed", "retained",  
 "removed", "retained"))  
  
 #Adding the confidence interval lines to the plot:  
arrows(barplot\_expdata, lower\_expdata, barplot\_expdata, upper\_expdata,  
 code = 3, angle = 90, cex = 0.7)  
  
 #Adding not fed/fed text to the bars  
BarNamesFed <- c("not fed", "not fed", "fed", "fed")  
text(barplot\_expdata,0.9, BarNamesFed, cex=0.9, pos = 3)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#III.5   
#The following lines of code generate a small dataset  
  
df.withNA<-data.frame(Trial=1:10, Experiment=c(rep("control",5),  
 rep("treatment",5)),  
 Measure=c(1,3,4,NA,3,4,5,NA,8,9))  
df.withNA<- data.frame(df.withNA,Measure2=c(2,3,2,3,4,5,4,5,6,6),  
 block=rep(c("B1","B2"),5))  
  
#(a) Use one of line of code with aggregate to compute how many different blocks there were per treatment group  
  
aggregate(block ~ Experiment, data = df.withNA,  
 function(x) length(unique(x)))

## Experiment block  
## 1 control 2  
## 2 treatment 2

#(b) Code the construction of a data.frame with the descriptive statistics sample size, mean and standard deviation for "Measure" for each of the four possible combinations of "Experiment" and "block".   
  
descriptiveStats.df.withNA<-aggregate(Measure~Experiment+block,  
 data=df.withNA, FUN =mean)  
descriptiveStats.df.withNA

## Experiment block Measure  
## 1 control B1 2.666667  
## 2 treatment B1 6.500000  
## 3 control B2 3.000000  
## 4 treatment B2 6.500000

#Changing the name "Measure" to "mean":  
names(descriptiveStats.df.withNA)[3]<-"mean"  
descriptiveStats.df.withNA

## Experiment block mean  
## 1 control B1 2.666667  
## 2 treatment B1 6.500000  
## 3 control B2 3.000000  
## 4 treatment B2 6.500000

#add n  
descriptiveStats.df.withNA<-data.frame(descriptiveStats.df.withNA,  
 n=aggregate(Measure~Experiment+block,  
 data=df.withNA,length)$Measure)  
descriptiveStats.df.withNA

## Experiment block mean n  
## 1 control B1 2.666667 3  
## 2 treatment B1 6.500000 2  
## 3 control B2 3.000000 1  
## 4 treatment B2 6.500000 2

#add sd  
descriptiveStats.df.withNA<-data.frame(descriptiveStats.df.withNA,  
 sd=aggregate(Measure~Experiment+block,  
 data=df.withNA,sd)[,3])  
descriptiveStats.df.withNA

## Experiment block mean n sd  
## 1 control B1 2.666667 3 1.527525  
## 2 treatment B1 6.500000 2 2.121320  
## 3 control B2 3.000000 1 NA  
## 4 treatment B2 6.500000 2 3.535534

#(c) Code the construction of a data.frame with the descriptive statistics sample size, mean and standard deviation for "Measure2" for each of   
the four possible combinations of "Experiment" and "block".   
  
descriptiveStats.df.withNA2<-aggregate(Measure2~Experiment+block,  
 data=df.withNA, FUN =mean)  
descriptiveStats.df.withNA2

## Experiment block Measure2  
## 1 control B1 2.666667  
## 2 treatment B1 5.000000  
## 3 control B2 3.000000  
## 4 treatment B2 5.333333

#Changing the name "Measure" to "mean"  
names(descriptiveStats.df.withNA2)[3]<-"mean"  
descriptiveStats.df.withNA2

## Experiment block mean  
## 1 control B1 2.666667  
## 2 treatment B1 5.000000  
## 3 control B2 3.000000  
## 4 treatment B2 5.333333

#add n  
descriptiveStats.df.withNA2<-data.frame(descriptiveStats.df.withNA2,  
 n=aggregate(Measure~Experiment+block,   
 data=df.withNA,length)$Measure)  
descriptiveStats.df.withNA2

## Experiment block mean n  
## 1 control B1 2.666667 3  
## 2 treatment B1 5.000000 2  
## 3 control B2 3.000000 1  
## 4 treatment B2 5.333333 2

#add sd  
descriptiveStats.df.withNA2<-data.frame(descriptiveStats.df.withNA2,  
 sd=aggregate(Measure~Experiment+block,  
 data=df.withNA,sd)[,3])  
descriptiveStats.df.withNA2

## Experiment block mean n sd  
## 1 control B1 2.666667 3 1.527525  
## 2 treatment B1 5.000000 2 2.121320  
## 3 control B2 3.000000 1 NA  
## 4 treatment B2 5.333333 2 3.535534

#(d) Code the construction of a data.frame with the descriptive statistics sample size, mean and standard deviation for "Measure" and "Measure2" using a single call to aggregate() for each of the four possible combinations of "Experiment" and "block".   
  
descriptiveStats.df.withNA3<-aggregate(cbind(Measure,Measure2)~Experiment+block, data=df.withNA, FUN =mean)  
  
descriptiveStats.df.withNA3

## Experiment block Measure Measure2  
## 1 control B1 2.666667 2.666667  
## 2 treatment B1 6.500000 5.000000  
## 3 control B2 3.000000 3.000000  
## 4 treatment B2 6.500000 5.500000

#Changing the name "Measure" to "mean"  
names(descriptiveStats.df.withNA3)[3]<-"mean1"  
names(descriptiveStats.df.withNA3)[4]<-"mean2"  
descriptiveStats.df.withNA3

## Experiment block mean1 mean2  
## 1 control B1 2.666667 2.666667  
## 2 treatment B1 6.500000 5.000000  
## 3 control B2 3.000000 3.000000  
## 4 treatment B2 6.500000 5.500000

#add n  
descriptiveStats.df.withNA3<-data.frame(descriptiveStats.df.withNA3,  
 n=(aggregate(Measure+Measure2~Experiment+block, data=df.withNA,length)[3]))  
descriptiveStats.df.withNA3

## Experiment block mean1 mean2 Measure...Measure2  
## 1 control B1 2.666667 2.666667 3  
## 2 treatment B1 6.500000 5.000000 2  
## 3 control B2 3.000000 3.000000 1  
## 4 treatment B2 6.500000 5.500000 2

#add sd  
descriptiveStats.df.withNA3<-data.frame(descriptiveStats.df.withNA3,  
 sd=aggregate(Measure~Experiment+block,  
 data=df.withNA,sd)[,3])  
descriptiveStats.df.withNA3

## Experiment block mean1 mean2 Measure...Measure2 sd  
## 1 control B1 2.666667 2.666667 3 1.527525  
## 2 treatment B1 6.500000 5.000000 2 2.121320  
## 3 control B2 3.000000 3.000000 1 NA  
## 4 treatment B2 6.500000 5.500000 2 3.535534

#For some reason the code doesn't name the column “n” right …  
  
  
#(e) Are there differences between what you get in a,b,c? If so, why?   
  
 #A&B are different since there are two different values  
 #The last way shows it wrong, since the NAs cause trouble

# Part IV. Combining multiple datasets

In the previous chapters, we have worked on writing programs that read information from a datafile and calculate something or plot something. These have been (unrealistically) simple in a number of ways. In this chapter, we consider the type of data and applications you are more likely to encounter. We start with some general reflections on databases from the perspective of constructing your own database and working with this. In addition, it is worth to keep in mind that for many analyses one would combine data from different sources. In the workflow, these operations fall under 3b. The underlying drive of this part (and of the course in general) is: Keep your datafiles in the original shape and use R to carry out all the work. In doing so, you have a clear record of what was done to get from your data to the output you obtained which means you can go back, adjust or repeat it whenever needed.

## Chapter 10 Database operations

## 10.1 Relational database: what and why

Connecting datafiles is a general issue which you encounter in most practical applications of programming. For example, when you carry out statistical analyses on data you collected you typically have to first put together the information you need for the analysis, often from multiple source files. Another application is when you wish to connect information in your data files to publicly available datasets, which is common in bio-informatics (e.g. matching genetic sequence data to genes), spatial analyses (e.g. map information to other spatial explicit information) and ecological analyses (e.g. climate data to population counts).

For most practical applications, we need to have more than one file. It is typically not possible, or extremely messy, to have all information in one file. This is because the structure of our data is typically fairly complex and different measures are made on different levels. As an example, consider the following: Suppose we have surveyed 30 meadows each for three times during the flowering season and counted how many of different species of flowers there were in 30 1m2 randomly placed squares on these meadows. We now have data on different levels, which are hierarchically nested.

**1. Meadow**

This includes characteristics of each meadow which are ‘fixed’. Its location, its orientation, its slope, its size, etc.

**2. Visit to each meadow**

Each visit has different characteristics, such as a different date, the weather conditions were probably different and perhaps also who did the survey on that day, etc.

**3. Findings within the squares placed out during each visit to each meadow.**

This is the lowest level of the data, and consists of counts of different species, made for each of the 30 squares.

To place all information in one file requires always that each data line concerns at least the lowest level observed in our data. Thus, for the above example, we could construct a single file with for each meadow, 3 times 30 rows of data on each square. This would then include the same information for the meadow characteristics (which are ’fixed’) for each of these rows. It would also have to have all the species one could possibly encounter in the columns. In short, this would be a very large datafile, with lots of repeats (for the levels meadows and visits) and probably lots of missing values (not all species observed in each square). Because the basic data is still fairly restricted in size, it would work, but it would be messy and inconvenient to work with. In entering the data, you would be prone to make a mistake and finding such mistakes would be difficult. Furthermore, when you want to plot or analyze the data, you would fairly quickly start to construct ‘sub-files’. This is where things get even more messy and it is possible to rapidly lose track of what and how you handled the data to arrive at a specific answer. This will not do, your findings must be repeatable and transparent!

One alternative is to design a database structure where the information is entered for each level, but where the files can be linked. This is called a relational database. In practice, this requires adding some identification code (ID), which links the levels, whenever relevant. For example, the files could be structured as displayed in Figure 10.1.



*Figure 10.1. Potential construction of a relational data base consisting of three files, where information on each level is denoted only once. The identifiers which link the different files are graphically displayed by arrows.*

**The objective of any relational database is that each piece of information is entered only once**. Specific files which you may need to address a specific problem or analysis are constructed from the database. The database files themselves are not altered, unless a mistake appears. Any mistake is corrected in the relevant database file, and the data you need for the analysis are then created again by database operations. This may sound cumbersome, but the advantages of working with a database are:

(1) Entering the information is easier, because redundancy is minimal. The only information which is repeated when you go one level lower is the ID number that links to the higher level. For example, in the “Meadow data” file, each meadow has one row with one “Meadow ID” number. In the “Visit data” file, the “Meadow ID” number needs to be repeated for each of the three rows with different “Visit ID” numbers.

(2) You avoid having several copies of your information in different files which are constructed to allow doing analyses on different levels. The latter is familiar to anyone who has analyzed data where the data manipulation are carried out manually to arrive at specific files for specific analyses. It is amazing how fast you accumulate files!

(3) Because each essential piece of information is entered only once, any mistake found requires only one correction to be made and this correction is done in the relevant database file, not in several files. This makes it much easier to keep track of your data and to be sure it is as correct as possible at any given time.

(4) Because the database is not used for analysis, it does not have to be constructed in a way to allow analyses and is thus much more flexible. For example, the “Census data” file in Fig. 10.1. is fundamentally different from one which you would otherwise need to construct for a statistical analysis. In particular, it could be constructed such that squares in which none of the focal species were found are simply not listed, which may drastically lower the amount of work you need for entering the data, and thus reduces the risk of making a mistake.

At this point, a familiar feeling is probably coming to you. One of the points of working in R which was made previously is the potential to concentrate your entire workflow to R. That is, reading in data, extracting the information you need and doing “stuff” with it such as statistics and plotting the results (for example). Clearly, therefore, it is attractive to read in multiple datafiles in R and write a program to link these, extract the information you need to construct the dataset you need for the analysis you want to do, run the analysis, plot the output or tabulate the result etc; all in one script which is annotated, and kept safe. This means other users can follow what is going on, you can go back to check what is done and in general the workflow will be much improved.

Obviously, you can use a database program (e.g. Microsoft Access, which is installed on your course computer) to do the database work. These programs are nice and clickable. But, a database program does not analyze anything for you, so you will always need to export it and import it in another program. In fact, my experience when working with ecological data is that database programs do not even work particularly well in extracting the file you need for your analysis. A main reason is that database programs are not designed well for handling missing values in a flexible manner. Of course, this may be a matter of skill or experience, but it is noteworthy that more demanding database problems require you to program it (in SQL or similar). In any case, one clear advantage of using R is that you do not need to switch programs and do a lot of clicking, exporting and importing after having made the initial investment of programming. If you find a mistake in the data, you correct the original database. If you find a mistake in what your coding did, you modify the script. In both cases, two key stroke combinations (CTRL+A, CTRL+ENTER) in the R-script editor will run all of it again and redo all of your analyses and plotting. It is important to realize that re-doing analyses because you spotted something odd is really common! Time you lose by sweating on constructing the database and in terms of writing your script to format the purpose-specific file for your analysis, you will gain back, with rent. The more your code, the better you get and the faster you will be able to code the next data extraction challenge. This is perhaps a main reason to stick to R and not confuse yourself with learning many different programs at least in the initial stage.

## 10.2 Using merge() to connect multiple data.frame objects

The function merge() merges data. Selection from the help on merge shows the most important arguments

**Usage**

merge(x, y, ...)

## Default S3 method:

merge(x, y, ...)

## S3 method for class 'data.frame'

merge(x, y, by = intersect(names(x), names(y)),

by.x = by, by.y = by, all = FALSE, all.x = all, all.y = all,

sort = TRUE, suffixes = c(".x",".y"),

incomparables = NULL, ...)

**Arguments**

|  |  |
| --- | --- |
| x, y | data frames, or objects to be coerced to one. |
| by, by.x, by.y | specifications of the columns used for merging. See ‘Details’. |
| all | logical; all = L is shorthand for all.x = L and all.y = L, where L is either [TRUE](http://127.0.0.1:13623/library/base/help/TRUE) or FALSE. |
| all.x | logical; if TRUE, then extra rows will be added to the output, one for each row in x that has no matching row in y. These rows will have NAs in those columns that are usually filled with values from y. The default is FALSE, so that only rows with data from both x and y are included in the output. |
| all.y | logical; analogous to all.x. |

Thus, by default merge() will try to merge two datasets by finding the tag the datasets have in common. This may work smoothly. For example,

> d1<-data.frame(students=c("Jukka", "Harald", "Sven", "Bitte"), nationality=c("F","N","S","D"))

> d2<-data.frame(ages=c(22,22,41,25),students=c("Harald", "Jukka", "Sven", "Bitte"))

> #merge the data.frames

> merge(d1,d2)

students nationality ages

1 Bitte D 25

2 Harald N 22

3 Jukka F 22

4 Sven S 41

In the above, the tag “students” was there in both data.frames and R merged them correctly.

In some cases, the naming is perhaps a bit different requiring specification of how merge() should do the trick. For example,

> #merge but different name

> d3<-data.frame(ages=c(22,22,41,25),studs=c("Harald", "Jukka", "Sven", "Bitte"))

> merge(d1,d3, by.x="students",by.y="studs")

students nationality ages

1 Bitte D 25

2 Harald N 22

3 Jukka F 22

4 Sven S 41

Here, the first argument is x and the second y (as specified in the help above).

It is important to note that merge() will assume the ID really is unique. For example, if we have another student called “Harald”

> d2.b<-rbind(d2,data.frame(ages=21, students="Harald"))

merge() will not understand the difference between these two Haralds

> merge(d1,d2.b)

students nationality ages

1 Bitte D 25

2 Harald N 22

3 Harald N 21

4 Jukka F 22

5 Sven S 41

Here, both Haralds are automatically assumed to have the nationality “N” although the nationality is specified only for one Harald and we in fact do not know the nationality of the second Harald.

The converse happens rather easily when merging. That is, when information is provided not for all the IDs, merge will by default simply reduce the data. Thus, missing information is by default omitted. In most practical applications, this is a very common situation due to missing values in the data. For example, perhaps we did not have the information for age for student Bitte

> d3.b<-d3[1:3,] #no age for Bitte

> d3.b

ages studs

1 22 Harald

2 22 Jukka

3 41 Sven

> #standard merge (Bitte falls off)

> merge(d1,d3.b, by.x="students", by.y="studs")

students nationality ages

1 Harald N 22

2 Jukka F 22

3 Sven S 41

Thus, student “Bitte” has disappeared after merging, and hence the information we had on her nationality. We need to specify all.x=TRUE as argument to merge() and override the default (see the help above). This makes sure we have all the information in the first data.frame (argument x) also after the merge operation. Thus,

> merge(d1,d3.b, by.x="students", by.y="studs", all.x=TRUE)

students nationality ages

1 Bitte D NA

2 Harald N 22

3 Jukka F 22

4 Sven S 41

In this case, the merged data.frame contains NA (missing value: see below) for the student for which we did not have the age in d3.b. Similar procedure works for the second argument in merge() (by specifiying all.y=TRUE) such that it is straightforward to merge two data files which both contain “loose” cases which are unique to either data file. As a rule, it is important to check that cases in your data are not lost after merging. An easy way to check this is either using dim() to obtain the dimension of the data.frame or nrow() to obtain the number of rows. Note that length() when applied to a data.frame will provide you the number of columns.

**10.3 Exercises part IV**

**Exercise IV.1**

Database files which correspond to the structure given in Fig. 10.1 can be found in Moodle. It consists of three “;” separated files called “MeadowsData.txt”, “VisitsData.txt” and “CensusData.txt”. [Note that I tend to avoid the .csv extension because Excel believes it can open them]

This data is made up. As detailed above, the data consists of 30 meadows, visited each 3 times. During each visit 30 squares were placed randomly and a count was made of how many of each of 35 species were present in the squares. The information on the different levels (meadow, visit, census) are linked through ID numbers).

Import these three text files in your workspace.

Use the database operations to first link those files you need and then extract the relevant information allowing you to produce a scatter plot of the slope of a meadow (on X) vs. the average temperature on that meadow (on Y) recorded during the three visits. Make the plot look as good (publication-ready) as you can in terms of axes labels, font sizes, plotting symbol, etc. This exercise requires to link information from two database files and couple information from two levels to each other.

**Exercise IV.2**

Import the meadows database as described above.

In the meadow database, the lowest level datafile “CensusData.txt” deals with the count of the species found in the squares. There were always 30 squares randomly placed during each visit. If no species were found, the square was not included in the data. That is, zero counts are not included.

a) Produce a histogram using hist() of how many squares were not empty (i.e. had one or more of the focal species in it) per meadow and per visit. That is, you should work out how you count how many squares there are in the census data file per visit (per meadow). Thus, get the columns

Meadow Visit Non-empty.squares.per.meadow.n.visit

1 101 21

1 102 14

1 103 18

2 201 20

---etc

This is, anyway, what I get ….

Add a column where you calculate the number of *empty squares* per visit per meadow, and a column where you calculate the *proportion* of all squares per visit per meadow, which were not empty.

b) Produce a scatter plot of the average temperature during the 3 visits in a meadow (on X) against the average proportion of non-empty squares during these three visits (on Y). Make the plot look as good (publication-ready) as you can in terms of axes labels, font sizes, plotting symbol, etc.

**Exercise IV.3**

In a second survey of the meadows, the thermometer used to measure the temperature was not always working. Thus, some visits had missing data for temperature. This data is in the file “VisitsDataWithNA.txt” which can be found in your Moodle. In this exercise, this file replaces the “VisitsData” file in your database.

Import this file.

1. Calculate the average temperature per open and closed meadow and the sample size used to calculate this average
2. One approach to deal with missing data in certain analyses is to assume the missing value is equal to the overall mean. Add a new column to the data where the missing value for temperature has been replaced with the overall mean temperature. Calculate the average temperature per open and closed meadow again and compare to previous estimates.
3. You are worried that one of the observers had real bad luck (or was really sloppy) with the measurement of the temperature. You want to test this. Calculate how many times each observer made a visit, how many times each observer had a missing value for the temperature and what the proportion of missing temperatures out of the total visits for each observer was.
4. Calculate per meadow how many times the temperature was measured and how many times it was missing.

**Exercise IV.4**

Somebody heard about your meadow project and came up with the following interesting information describing for each meadow whether it was “open” or “closed” 20 years ago. The piece of paper she gave your reads:

Mdw Closed?

1. yes
2. no
3. yes
4. yes
5. no
6. no
7. no
8. no
9. no
10. yes
11. no
12. no
13. no
14. yes
15. no
16. no
17. no
18. no
19. yes
20. no
21. no
22. yes
23. no
24. no
25. no
26. yes
27. yes
28. yes
29. no
30. yes
31. Add this data to your database.
32. Produce a single bar plot depicting the number of meadows which were open or closed 20 years ago and open or closed now. That is, the numbers of open-open; open-closed; closed-open; closed-closed.

**Exercise IV.5**

The following code produces two data frames containing information on countries and their area.

The first data frame contains country names, their ranking (relative to the other countries in the same data set) and area (information found on <https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_area> ).The second contains country names, continents, and their ranking within each continent.

Countries.area<-data.frame("Country"=c("Brasil", "Argentina", "Peru", "South-Africa", "Colombia", "Ethiopia" ,"France" ,"Madagascar", "Kenya", "Morocco", "Germany", "Italy", "Ecuador" ,"Iceland", "Latvia"),"Rank"=c(1:15), "Area"= c(8515767, 2780400 ,1285216, 1221037, 1141748, 1104300, 672051, 587041, 580367, 446550, 357021, 301336, 256369, 103000, 64559))

Countries.Continents<-data.frame("country"=c("Brasil", "Argentina", "Peru", "South-Africa", "Colombia", "Ethiopia" ,"Madagascar", "Kenya", "Morocco", "Germany", "Finland","Italy", "Ecuador" ,"Iceland", "Latvia"),"Continent"=c("South-America","South-America","South-America","Africa","South-America","Africa","Africa","Africa","Africa","Europe", "Europe","Europe", "South-America","Europe","Europe"),"Rank"=c(1, 2, 3, 1, 4, 2, 3, 4, 5, 1, 2, 3, 5, 4, 5))

1. Calculate the average area of the first ranked countries of each continent.
2. Two countries actually have incomplete information. Complete your already existing table (using the Wikipedia page) and recalculate a) on the corrected table.

**MY CODE: Part 4:**

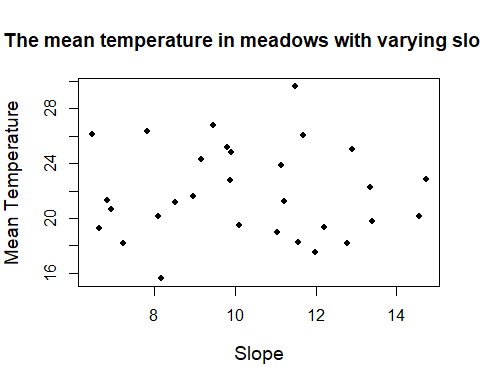
#Exercise IV.1   
#Use the database operations to first link those files you need and then   
extract the relevant information allowing you to produce a scatter plot of the slope of a meadow (on X) vs. the average temperature on that meadow (on Y) recorded during the three visits. Make the plot look as good (publication-ready) as you can in terms of axes labels, font sizes, plotting symbol, etc. This exercise requires to link information from two database files and couple information from two levels to each other.  
  
setwd("E:/Introduction to R")  
Meadowsdata <- read.csv2("MeadowsData.txt")  
Visitdata <- read.csv2("VisitsData.txt")  
Censusdata <- read.csv2("CensusData.txt")  
  
 #merging the datasets  
MeadowVisitMerge <- merge(Meadowsdata, Visitdata)  
  
 #Converting the data from factor to numeric for the mean function  
MeadowVisitMerge$Temperature <- as.numeric(levels(MeadowVisitMerge$Temperature))[MeadowVisitMerge$Temperature]  
MeadowVisitMerge$Slope <- as.numeric(levels(MeadowVisitMerge$Slope))[MeadowVisitMerge$Slope]  
class(MeadowVisitMerge$Temperature)

## [1] "numeric"

class(MeadowVisitMerge$Slope)

## [1] "numeric"

#Making a dataframe with the slope and the mean temperature. The mean  
 temperature is calculated with tapply in the code below:  
plotdata <- data.frame(Meadow.ID = unique(MeadowVisitMerge$Meadow.ID),  
 Slope = unique(MeadowVisitMerge$Slope),  
 Temperature.mean = tapply(MeadowVisitMerge$Temperature,  
 MeadowVisitMerge$Meadow.ID,  
 mean))  
  
 #plotting the slope vs. mean temperature  
plot(plotdata$Temperature.mean ~ plotdata$Slope,   
 xlab = list("Slope", cex = 1.2),  
 ylab = list("Mean Temperature", cex = 1.2),  
 main = "The mean temperature in meadows with varying slopes",  
 pch = 16, cex = 0.8)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#IV.2  
#In the meadow database, the lowest level datafile "CensusData.txt" deals with the count of the species found in the squares. There were always 30  
squares randomly placed during each visit. If no species were found, the   
square was not included in the data. That is, zero counts are not  
included.   
  
#a) Produce a histogram using hist() of how many squares were not empty   
(i.e. had one or more of the focal species in it) per meadow and per   
visit.That is, you should work out how you count how many squares there   
are in the census data file per visit (per meadow).  
  
 #Making a data frame with the desired data  
VisitCensusmerge <- merge(Visitdata, Censusdata)  
histdata <- data.frame(Meadow.ID = VisitCensusmerge$Meadow.ID,  
 Visit.ID = VisitCensusmerge$Visit.ID,  
 Square = VisitCensusmerge$Square)  
head(histdata)

## Meadow.ID Visit.ID Square  
## 1 1 101 1  
## 2 1 101 1  
## 3 1 101 1  
## 4 1 101 2  
## 5 1 101 2  
## 6 1 101 2

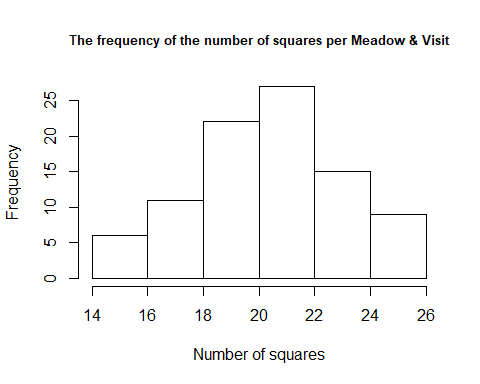
#Making a data frame without the dublicates. Adding a column for   
"non-empty" squares -> when aggregate is used, it only uses the unique   
sample size of "Square" -> we get how many non-empty squares/visit there   
are  
  
histdata\_UniqueRows <- unique(histdata)  
head(histdata\_UniqueRows)

## Meadow.ID Visit.ID Square  
## 1 1 101 1  
## 4 1 101 2  
## 12 1 101 3  
## 16 1 101 4  
## 17 1 101 6  
## 19 1 101 7

#Making a data frame for the histogram  
histdata\_UniqueSquares <- aggregate(Square ~ (Meadow.ID + Visit.ID), data = histdata\_UniqueRows, FUN = length)  
head(histdata\_UniqueSquares)

## Meadow.ID Visit.ID Square  
## 1 1 101 21  
## 2 1 102 24  
## 3 1 103 18  
## 4 2 201 20  
## 5 2 202 19  
## 6 2 203 20

#The histogram  
hist(histdata\_UniqueSquares$Square,  
 xlab = "Number of squares",  
 main = list("The frequency of the number of squares per Meadow & Visit",  
 cex = 0.8))



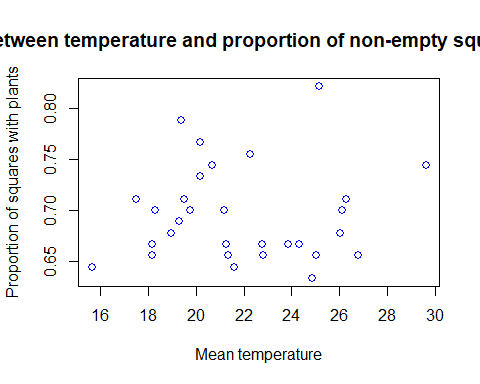
#Add a column where you calculate the number of empty squares per visit   
per meadow, and a column where you calculate the proportion of all squares per visit per meadow, which were not empty.  
  
  
 #Adding a column for empty squares  
histdata\_UniqueSquares$EmptySquares<- 30-histdata\_UniqueSquares$Square  
head(histdata\_UniqueSquares)

## Meadow.ID Visit.ID Square EmptySquares  
## 1 1 101 21 9  
## 2 1 102 24 6  
## 3 1 103 18 12  
## 4 2 201 20 10  
## 5 2 202 19 11  
## 6 2 203 20 10

#Adding a column for proportion of non-empty squares of all squares  
histdata\_UniqueSquares$ProportionOfNonEmptySquares <- histdata\_UniqueSquares$Square/30  
  
head(histdata\_UniqueSquares)

## Meadow.ID Visit.ID Square EmptySquares ProportionOfNonEmptySquares  
## 1 1 101 21 9 0.7000000  
## 2 1 102 24 6 0.8000000  
## 3 1 103 18 12 0.6000000  
## 4 2 201 20 10 0.6666667  
## 5 2 202 19 11 0.6333333  
## 6 2 203 20 10 0.6666667

# b) Produce a scatter plot of the average temperature during the 3 visitsin a meadow (on X) against the average proportion of non-empty squares   
during these three visits (on Y). Make the plot look as good (publication-ready) as you can in terms of axes labels, font sizes, plotting symbol,   
etc.  
  
 #Calculating the means of the proportions per meandow  
ProportionMean <- aggregate(ProportionOfNonEmptySquares~Meadow.ID,  
 data = histdata\_UniqueSquares,  
 FUN = mean)  
  
plot(plotdata$Temperature.mean,ProportionMean$ProportionOfNonEmptySquares,  
 cex = 1.1, xlab = "Mean temperature",  
 ylab = "Proportion of squares with plants",  
 main = list("Correlation between temperature and proportion of non-empty squares in meadows", cex= 1.2),  
 col = "blue")



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#Exercise IV.3  
  
#In a second survey of the meadows, the thermometer used to measure the temperature was not always working. Thus, some visits had missing data for temperature. This data is in the file "VisitsDataWithNA.txt" which can be found in your Moodle. In this exercise, this file replaces the "VisitsData" file in your database.  
  
VisitDataWithNA <- read.csv2("VisitsDataWithNA.txt")  
  
#(i) Calculate the average temperature per open and closed meadow and  
 the sample size used to calculate this average  
  
 #Merging the two data frames  
VisitNA.Meadow.merge <- merge(VisitDataWithNA, Meadowsdata)  
  
 #Converting the data from factor to numeric for the mean function  
VisitNA.Meadow.merge$Temperature <- as.numeric(levels(VisitNA.Meadow.merge$Temperature))[VisitNA.Meadow.merge$Temperature]  
class(VisitNA.Meadow.merge$Temperature)

## [1] "numeric"

#The mean temperature and sample size per open and closed meadow (how   
the question was interpreted in class)  
  
aggregate(Temperature ~ Openness, data = VisitNA.Meadow.merge, FUN = mean)

## Openness Temperature  
## 1 Closed 21.16081  
## 2 Open 22.28424

aggregate(Temperature ~ Openness, data = VisitNA.Meadow.merge,  
 FUN = length)

## Openness Temperature  
## 1 Closed 31  
## 2 Open 51

#(ii) One approach to deal with missing data in certain analyses is to   
assume the missing value is equal to the overall mean. Add a new column to the data where the missing value for temperature has been replaced with   
the overall mean temperature. Calculate the average temperature per open   
and closed meadow again and compare to previous estimates.  
  
 #Making a copy of the Temperature column  
VisitNA.Meadow.merge$Temperature2 <- VisitNA.Meadow.merge$Temperature  
  
 #Replacing the NA-values with the mean  
VisitNA.Meadow.merge$Temperature2[is.na(VisitNA.Meadow.merge$Temperature2)] <- mean(VisitNA.Meadow.merge$Temperature2, na.rm = TRUE)  
  
 #The new mean temp:  
aggregate(Temperature2 ~ Openness, data = VisitNA.Meadow.merge, FUN= mean)

## Openness Temperature2  
## 1 Closed 21.25785  
## 2 Open 22.26064

#.. compared to the old one: (quite close!)  
aggregate(Temperature ~ Openness, data = VisitNA.Meadow.merge, FUN = mean)

## Openness Temperature  
## 1 Closed 21.16081  
## 2 Open 22.28424

#(How we did it in class:)  
#ifelse(is.na(VisitDataWithNA$Temperature) == TRUE, mean(VisitDataWithNA$Temperature, na.rm = TRUE), VisitDataWithNA$Temperature)  
  
  
#(iii) You are worried that one of the observers had real bad luck (or   
was really sloppy) with the measurement of the temperature. You want to   
test this. Calculate how many times each observer made a visit, how many   
times each observer had a missing value for the temperature and what the   
proportion of missing temperatures out of the total visits for each   
observer was.  
  
 #How many times each observer made a visit:  
total.visits <-aggregate(Visit.ID ~ Observer, data = VisitNA.Meadow.merge,  
 length)  
  
 #OR  
total.visits2 <-tapply(VisitNA.Meadow.merge$Temperature,   
 VisitNA.Meadow.merge$Observer, length)  
  
 #How many times each observer had a missing value:  
total.visits.without.NA <-aggregate(Temperature~Observer,  
 data = VisitNA.Meadow.merge,  
 FUN = length)  
  
 #How many NA  
total.visits.with.NA <- total.visits[,2] - total.visits.without.NA[,2]  
 #Bob had no NAs, Harriet had 8 NAs  
  
 #Proportion of successfull measurements (The first value is Bob, second   
 Harriet)  
total.visits.without.NA[,2]/total.visits[,2]

## [1] 1.0000000 0.8857143

#(iv) Calculate per meadow how many times the temperature was measured and how many times it was missing.  
  
 #The number of visits per meadow  
a1 <- aggregate(is.na(Temperature)~Meadow.ID, data = VisitDataWithNA,  
 FUN = length)  
head(a1)

## Meadow.ID is.na(Temperature)  
## 1 1 3  
## 2 2 3  
## 3 3 3  
## 4 4 3  
## 5 5 3  
## 6 6 3

#How many times the temperature was measured  
a2 <- aggregate(Temperature~Meadow.ID, data = VisitDataWithNA,  
 FUN = length)  
head(a2)

## Meadow.ID Temperature  
## 1 1 3  
## 2 2 2  
## 3 3 3  
## 4 4 1  
## 5 5 3  
## 6 6 3

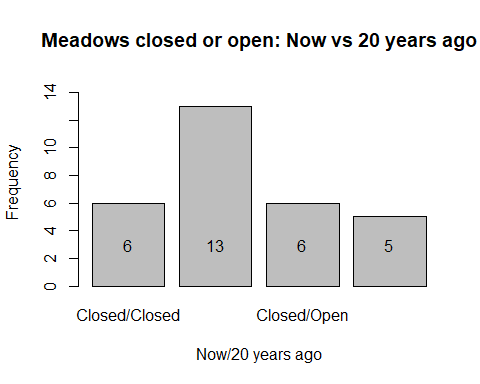
#How many times the temperature was not measured  
TempNotMeasured <- a1[,2]-a2[,2]  
a3 <- data.frame(Meadow.ID = a1$Meadow.ID,  
 TempNotMeasured = TempNotMeasured)  
head(a3)

## Meadow.ID TempNotMeasured  
## 1 1 0  
## 2 2 1  
## 3 3 0  
## 4 4 2  
## 5 5 0  
## 6 6 0

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#Exercise IV.4  
#Somebody heard about your meadow project and came up with the following   
interesting information describing for each meadow whether it was "open"   
or "closed" 20 years ago. The piece of paper she gave your reads:  
  
 #(I saved the provided information in a text file in notebook)  
  
#(i) Add this data to your database.  
  
Open20YearsAgoData <- read.table("Openness20YearsAgo.txt")  
colnames(Open20YearsAgoData) <- c("Meadow.ID", "Openness20YearsAgo")  
  
  
#(ii) Produce a single bar plot depicting the number of meadows which   
were open or closed 20 years ago and open or closed now. That is, the   
numbers of open-open; open-closed; closed-open; closed-closed.  
  
 #changing the labels so that they are the same for the "new data" and   
 "old data"  
Open20YearsAgoData$Openness20YearsAgo <- ifelse(Open20YearsAgoData$Openness20YearsAgo == "yes", "Open", "Closed")  
  
 #Merging the new piece of information to the Meadowsdata  
MeadowOpen20YearsAgoMerge <- merge(Meadowsdata, Open20YearsAgoData)  
  
 #Aggregating data to use in the barplot  
ClosedOpenData <- aggregate(Meadow.ID ~ Openness + Openness20YearsAgo,  
 data= MeadowOpen20YearsAgoMerge,  
 FUN = length)  
  
colnames(ClosedOpenData) <- c("OpennessNow", "Openness20YearsAgo",  
 "Frequency")  
ClosedOpenData

## OpennessNow Openness20YearsAgo Frequency  
## 1 Closed Closed 6  
## 2 Open Closed 13  
## 3 Closed Open 6  
## 4 Open Open 5

barplotOpenness <- barplot(ClosedOpenData$Frequency,  
 xlab = "Now/20 years ago",  
 ylab = "Frequency",  
 main = "Meadows closed or open: Now vs 20 years ago",  
 ylim = c(0,15), names.arg = c("Closed/Closed",  
 "Open/Closed",  
 "Closed/Open",  
 "Open/Open"))  
  
 #Adding the sample sizes of each group:  
text(barplotOpenness, 3, ClosedOpenData$Frequency )



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#Exercise IV.5  
#The following code produces two data frames containing information on countries and their area.  
  
Countries.area<-data.frame("Country"=c("Brasil", "Argentina", "Peru", "South-Africa", "Colombia", "Ethiopia" ,"France" ,"Madagascar", "Kenya", "Morocco", "Germany", "Italy", "Ecuador" ,"Iceland", "Latvia"),"Rank"=c(1:15), "Area"= c(8515767, 2780400 ,1285216, 1221037, 1141748, 1104300, 672051, 587041, 580367, 446550, 357021, 301336, 256369, 103000, 64559))  
  
Countries.Continents<-data.frame("country"=c("Brasil", "Argentina", "Peru", "South-Africa", "Colombia", "Ethiopia" ,"Madagascar", "Kenya", "Morocco", "Germany", "Finland","Italy", "Ecuador" ,"Iceland", "Latvia"),"Continent"=c("South-America","South-America","South-America","Africa","South-America","Africa","Africa","Africa","Africa","Europe", "Europe","Europe", "South-America","Europe","Europe"),"Rank"=c(1, 2, 3, 1, 4, 2, 3, 4, 5, 1, 2, 3, 5, 4, 5))  
  
#a) Calculate the average area of the first ranked countries of each continent.  
  
 #Merging the two data frames  
CountriesMerge <- merge(Countries.area, Countries.Continents,  
 by.x="Country", by.y="country",  
 all.x=TRUE, all.y = TRUE)  
  
 #Giving proper names to the columns:  
colnames(CountriesMerge) <- c("Country", "Rank.Area",  
 "Area", "Continent",  
 "Rank.Area.Continents")  
CountriesMerge

## Country Rank.Area Area Continent Rank.Area.Continents  
## 1 Argentina 2 2780400 South-America 2  
## 2 Brasil 1 8515767 South-America 1  
## 3 Colombia 5 1141748 South-America 4  
## 4 Ecuador 13 256369 South-America 5  
## 5 Ethiopia 6 1104300 Africa 2  
## 6 France 7 672051 <NA> NA  
## 7 Germany 11 357021 Europe 1  
## 8 Iceland 14 103000 Europe 4  
## 9 Italy 12 301336 Europe 3  
## 10 Kenya 9 580367 Africa 4  
## 11 Latvia 15 64559 Europe 5  
## 12 Madagascar 8 587041 Africa 3  
## 13 Morocco 10 446550 Africa 5  
## 14 Peru 3 1285216 South-America 3  
## 15 South-Africa 4 1221037 Africa 1  
## 16 Finland NA NA Europe 2

#Getting the avarage for each rank in each continent  
MeanAreaByContinentRanking <- aggregate(Area ~ Continent +   
 Rank.Area.Continents,  
 data = CountriesMerge, FUN = mean)  
head(MeanAreaByContinentRanking)

## Continent Rank.Area.Continents Area  
## 1 Africa 1 1221037  
## 2 Europe 1 357021  
## 3 South-America 1 8515767  
## 4 Africa 2 1104300  
## 5 South-America 2 2780400  
## 6 Africa 3 587041

#Getting only the values for first ranked countires  
subset(MeanAreaByContinentRanking,  
 MeanAreaByContinentRanking$Rank.Area.Continents == 1)

## Continent Rank.Area.Continents Area  
## 1 Africa 1 1221037  
## 2 Europe 1 357021  
## 3 South-America 1 8515767

#b) Two countries actually have incomplete information. Complete your   
already existing table (using the Wikipedia page) and recalculate a) on   
the corrected table.  
  
 #Fixing the continent of France  
CountriesMerge[6,4] <- "Europe"  
  
 #If we add France's true ranking (3) without alterations first; there   
will be two countries ranked the same, so therefore: Fixing the overall   
ranking before France is inserted (+1):  
  
CountriesMerge$Rank.Area.Continents  
 <- ifelse(CountriesMerge$Continent == "Europe" &  
 CountriesMerge$Rank.Area.Continents >= 3,  
 CountriesMerge$Rank.Area.Continents+1,  
 CountriesMerge$Rank.Area.Continents)  
 #Fixing the ranking of France  
CountriesMerge[6,5] <- 3  
  
 #Fixing the area of Finland  
CountriesMerge[16,3] <- 338145  
  
 #If we add Finlands true ranking (11) without alterations first; there   
will be two countries ranked the same, so therefore: Fixing the overall   
ranking before Finland is inserted (+1):  
  
CountriesMerge$Rank.Area <- ifelse(CountriesMerge$Rank.Area >= 11,  
 CountriesMerge$Rank.Area+1,  
 CountriesMerge$Rank.Area)  
  
 #Fixing the ranking of Finland  
CountriesMerge[16,2] <- 11  
CountriesMerge

## Country Rank.Area Area Continent Rank.Area.Continents  
## 1 Argentina 2 2780400 South-America 2  
## 2 Brasil 1 8515767 South-America 1  
## 3 Colombia 5 1141748 South-America 4  
## 4 Ecuador 14 256369 South-America 5  
## 5 Ethiopia 6 1104300 Africa 2  
## 6 France 7 672051 Europe 3  
## 7 Germany 12 357021 Europe 1  
## 8 Iceland 15 103000 Europe 5  
## 9 Italy 13 301336 Europe 4  
## 10 Kenya 9 580367 Africa 4  
## 11 Latvia 16 64559 Europe 6  
## 12 Madagascar 8 587041 Africa 3  
## 13 Morocco 10 446550 Africa 5  
## 14 Peru 3 1285216 South-America 3  
## 15 South-Africa 4 1221037 Africa 1  
## 16 Finland

# Part V Repeatedly carrying out the same set of instructions

In the previous parts, our focus has been on working through the workflow (Figure 5.0) linearly. That is, each operation is performed only once and then the program moves to the next operation. In this part, we start using R to repeatedly carry out one or more operations. Knowing some coding to handle this kind of automation is of course a clear advantage as you will otherwise find yourself repeating the same procedure many times, with clear risk for error.

## Chapter 11. Objects of class list

## 11.1 list

We have encountered a list when improving the axes of barplot() and plot() in chapter 8. A list is very flexible in storing information, even more flexible than a data.frame. It can contain various pieces of information. For example, a vector, matrix and data.frame

> myList<-list(

+ myVec=c(1,2,3)

+ , myMat=matrix(c(1,2,3,4),2,2)

+ , myDf=data.frame(students=c("Bill","Miranda","Laura"), ages=c(22,21,19)))

> myList

$`myVec`

[1] 1 2 3

$myMat

[,1] [,2]

[1,] 1 3

[2,] 2 4

$myDf

students ages

1 Bill 22

2 Miranda 21

3 Laura 19

> str(myList)

List of 3

$ myVec: num [1:3] 1 2 3

$ myMat: num [1:2, 1:2] 1 2 3 4

$ myDf :'data.frame': 3 obs. of 2 variables:

..$ students: Factor w/ 3 levels "Bill","Miranda",..: 1 2 3

..$ ages : num [1:3] 22 21 19

We can index a list by using $ and the tag name or by using double square brackets [[ ]] instead of single square brackets []), but otherwise following the same logic as for other classes

> myList$myMat

[,1] [,2]

[1,] 1 3

[2,] 2 4

> myList[["myMat"]]

[,1] [,2]

[1,] 1 3

[2,] 2 4

> myList[[2]]

[,1] [,2]

[1,] 1 3

[2,] 2 4

A list can be extended readily

> myList$extra<-letters

> myList

$myVec

[1] 1 2 3

$myMat

[,1] [,2]

[1,] 1 3

[2,] 2 4

$myDf

students ages

1 Bill 22

2 Miranda 21

3 Laura 19

$extra

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"

[20] "t" "u" "v" "w" "x" "y" "z"

Or by vector indexing which will mean the list element will not have a tag

> myList[[5]]<-TRUE

> myList

--- some info not displayed ---

$extra

[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"

[20] "t" "u" "v" "w" "x" "y" "z"

[[5]]

[1] TRUE

> names(myList)

[1] "myVec" "myMat" "myDf" "extra" ""

A list can also be recursive (list within a list). For example, we can add a list containing the letters in the alphabet and a number as a list (note that 1:length(letters)creates a vector of the number 1,2, .. length of the vector letters which is thus a counter)

> myList$mySecondList<-list(letter=letters, count=1:length(letters))

> str(myList)

List of 6

$ myVec : num [1:3] 1 2 3

$ myMat : num [1:2, 1:2] 1 2 3 4

$ myDf :'data.frame': 3 obs. of 2 variables:

..$ students: Factor w/ 3 levels "Bill","Miranda",..: 1 2 3

..$ ages : num [1:3] 22 21 19

$ extra : chr [1:26] "a" "b" "c" "d" ...

$ : logi TRUE

$ mySecondList:List of 2

..$ letter: chr [1:26] "a" "b" "c" "d" ...

..$ count : int [1:26] 1 2 3 4 5 6 7 8 9 10 ...

> myList[["mySecondList"]]

$letter

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

$count

[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

Note from the above that the list now created is considered to contain 6 elements. We can verify this by asking the length of the list

> length(myList)

[1] 6

Thus, the list now is composed of 6 main branches (in the structure which str() produces, these are listed behind the “first” $ indented to the far left; myVec, myMat, myDf, extra, "", mySecondList). Two of the “branches” (myDf, MySecondList) split up into further sub-branches, where the former are the tags for the data.frame and the latter the one for the list. We can index those by further indexing using the same rules as always. For example, we can further index mySecondList using square brackets

> myList[["mySecondList"]][[1]]

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

or dollar signs

> myList$mySecondList$letter

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

A list is an important class in R. There is in general little need to create lots of different objects for temporary storage of all kinds of information of various classes. It is handier and more compact to store all needed information in one list and access the information through indexing. For example, the above myList holds many different pieces of information yet we can easily obtain each part of it through indexing. Because they are so versatile, lists are common in R. For example, most statistical tests will return the results as a list.

## 11.2 Converting a list to object of another class

List and data.frame are closely related. In fact, if a list is of the appropriate format, it can be easily made into a data.frame

> class(myList[["mySecondList"]])

[1] "list"

> data.frame(myList[["mySecondList"]])

letter count

1 a 1

2 b 2

3 c 3

--- (shortened) ----

The function unlist() will do its best to create a vector out of a list

> class(myList[["mySecondList"]])

[1] "list"

> unlist(myList[["mySecondList"]])

letter1 letter2 letter3 letter4 letter5 letter6 letter7 letter8

"a" "b" "c" "d" "e" "f" "g" "h"

letter9 letter10 letter11 letter12 letter13 letter14 letter15 letter16

"i" "j" "k" "l" "m" "n" "o" "p"

letter17 letter18 letter19 letter20 letter21 letter22 letter23 letter24

"q" "r" "s" "t" "u" "v" "w" "x"

letter25 letter26 count1 count2 count3 count4 count5 count6

"y" "z" "1" "2" "3" "4" "5" "6"

count7 count8 count9 count10 count11 count12 count13 count14

"7" "8" "9" "10" "11" "12" "13" "14"

count15 count16 count17 count18 count19 count20 count21 count22

"15" "16" "17" "18" "19" "20" "21" "22"

count23 count24 count25 count26

"23" "24" "25" "26"

## Chapter 12 Working with strings and more on logical operators

## 12.1 String manipulation

A string is an element in a character vector and is interpreted as “text”. When reading in data, R will by default interpret anything with a letter in it as a factor. Remember that you can override this default when reading in the data by specifying stringsAsFactor = FALSE. Other option is to coerce the factor into a character vector using as.character(). Also numerical variable can be coerced to characters. Once the information is in string format, it is amenable to manipulation. Let’s consider some examples. This first one uses strsplit() (string split) to recode a variable which contains the date)

> df2<-data.frame(name=c("Bob","Jenny","John"), birthday=c("01.01.2010", "02.02.2010","03.03.2010"))

> df2

name birthday

1 Bob 01.01.2010

2 Jenny 02.02.2010

3 John 03.03.2010

> df2[,"birthday"]<-as.character(df2[1,"birthday"])

> df2 #note the appearance on the screen does not change…

name birthday

1 Bob 01.01.2010

2 Jenny 01.01.2010

3 John 01.01.2010

> strsplit(df2[,"birthday"],".",fixed=T)

[[1]]

[1] "01" "01" "2010"

[[2]]

[1] "01" "01" "2010"

[[3]]

[1] "01" "01" "2010"

A number of aspects are noteworthy in the above. First, the function strsplit does not work on factors, only on character vectors, hence the coercion to character. Second, the function strsplit splits the string according to so-called *regular expressions* only. For more information on regular expressions, check the help on regexp in R. Bottom line for this case is that the dot “.” is a regular expression that stands for “any character”.

Thus,

> strsplit(df2[,"birthday"],".")

[[1]]

[1] "" "" "" "" "" "" "" "" "" ""

[[2]]

[1] "" "" "" "" "" "" "" "" "" ""

[[3]]

[1] "" "" "" "" "" "" "" "" "" ""

which is not helpful. In this example, the fixed = TRUE addition is used which specifies the match needs to be exact which overrides the default interpretation of “.” as “any character”. Third, strsplit produces a list. Remember that you can coerce the output into a vector by using unlist()

> unlist(strsplit(df2[,"birthday"],".",fixed=T))

[1] "01" "01" "2010" "01" "01" "2010" "01" "01" "2010"

The above thus allow separating a string containing “day.month.year” into day, month, year (some more work is needed of course).

The reverse of strsplit() is the function paste()

> paste("this allows also to paste number", as.character(9), "as a character")

[1] "this allows also to paste number 9 as a character"

Other useful functions are based on finding a pattern and include grep() and sub() and functions related to these. The former states where a text pattern is found and the second allows finding and replacing a pattern within a string. For example,

> MyFiles<-(c("data\_020217","data\_010117"))

> grep("0101",MyFiles)

[1] 2

> sub("0101","0102",MyFiles)

[1] "data\_020217" "data\_010217"

## 12.2 Sorting

One may wish to sort data. One basic function for this is sort()

> x=c(1,2,5,4)

> x

[1] 1 2 5 4

> sort(x)

[1] 1 2 4 5

which gives the values of the vector x sorted from low to high (see help for changing defaults). The second function order() gives the indices of x in the order low to high

> order(x)

[1] 1 2 4 3

Note that none of these operation actually changes x

> x

[1] 1 2 5 4

If we want to re-define the object x such that it is sorted, we need to code this as

> x

[1] 1 2 5 4

> x<-sort(x)

> x

[1] 1 2 4 5

or

> x=c(1,2,5,4)

> x

[1] 1 2 5 4

> x<-x[order(x)]

> x

[1] 1 2 4 5

## 12.3 More on logical operators

We encountered the basic logical operator (==, >, <, etc). While these are handy, there are a number of operators allowing the construction of more precise “questions”. For example, we may want to select only a few specific cases out of a vector. Here the function %in% typically is handy

> vec=c(1,1,2,2,2,3,4,5,5,6,6,6,7,8,9)

> vec%in%c(2,8)

[1] FALSE FALSE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE

> vec[vec%in%c(2,8)]

[1] 2 2 2 8

> vec[!vec%in%c(2,8)]

[1] 1 1 3 4 5 5 6 6 6 7 9

R furthermore has a number of functions for finding different specific conditions. For example, the function which() allows you to find where a specific condition is TRUE. For example,

> which(vec==3)

[1] 6

> which.max(vec)

[1] 15

> which.min(vec)

[1] 1

The above hence returns which element in vec equals 3, the maximal and minimal value respectively.

## Chapter 14 for loops

A for loop is a classic programming structure to repeatedly carry out the same set of instructions. In this chapter we take a look at this feature.

**14.1 Initialization**

When you read a data file into R, or when you extract information from database files, you essentially create a new data.frame. By further extracting data from a data.frame, you will automatically create another data.frame (remember that you may need to include drop=FALSE). These options we have become familiar with in previous chapters. In some cases, however, you may want to construct a data.frame from scratch, one part at a time, starting with nothing. R complains when you directly define an object, such as a data.frame, and at the same time want to ‘fill’ it with information. For example, suppose you want to place information on a variable v1 in a data.frame called dat.

> dat$v1<-1

Error in dat$v1 <- 1 : object 'dat' not found

R complains: dat is not defined. You therefore first need to define (initiate) this data.frame as an empty data.frame. For example,

> dat<-data.frame(v1=numeric(),v2=character())

> dat

[1] v1 v2

<0 rows> (or 0-length row.names)

The above tells R that dat is an empty data.frame consisting of the numerical variable v1 and, in this case, also a character variable v2. You can then use, as before, the functions rbind() or cbind() to add data to this empty data.frame. For example,

> dat<-rbind(dat,data.frame(v1=c(1,2,3),v2=c("Bob","Bob","Harriet")))

> dat

v1 v2

1 1 Bob

2 2 Bob

3 3 Harriet

Remember that only a data.frame with the correct tags can be added to a data.frame.

The above also works without specifying v1 and v2 when you define your data.frame first as empty

> dat2=data.frame()

> dat2

data frame with 0 columns and 0 rows

> rbind(dat2,data.frame(v1=c(1,2,3),v2=c("Bob","Bob","Harriet")))

v1 v2

1 1 Bob

2 2 Bob

3 3 Harriet

To initialize, you may also specify an empty list and first include all your information into this list and afterwards coerce the list to a data.frame.

> dat3=list()

> dat3

list()

> dat3$v1=c(1,2,3)

> dat3$v2=c("Bob","Bob","Harriet")

> dat3

$v1

[1] 1 2 3

$v2

[1] "Bob" "Bob" "Harriet"

> dat3<-as.data.frame(dat2)

> dat3

v1 v2

1 1 Bob

2 2 Bob

3 3 Harriet

The flexibility of adding information to a list is a bit greater than when you define an empty data.frame. For practical applications, it depends on what you want to do which option to use (or some other options).

**14.2 For loop**

Often you want to repeat the same operation a number of times. The classical programming structure for doing this is a so-called for loop. A for loop will re-iterate the code written between the curly brackets ({}) for as many times as is specified in the for command. There are multiple ways of constructing for loops.

Let’s assume we have data on 100 sites and we want to calculate some statistics, say the mean. Suppose we have a data.frame dat

> summary(dat)

site X Y

A1 : 20 Min. : -0.3927 Min. :-2.768

A10 : 20 1st Qu.: 25.5230 1st Qu.: 4.598

A100 : 20 Median : 50.3463 Median : 6.920

A11 : 20 Mean : 50.4666 Mean : 6.667

A12 : 20 3rd Qu.: 75.7369 3rd Qu.: 8.824

A13 : 20 Max. :102.5700 Max. :15.711

(Other):1880

As we are repeating this calculation of the mean for each site, one by one, we need to store each value. One alternative to storing your data in a data.frame is to initialize sites.means as an empty data.frame, and then fill it with the information needed using rbind. Thus, you use rbind within the for loop to add rows to the data.frame. Something like

> # for loop filling empty data.frame

> # which are the sites. This needs to be defined beforehand

> sites<-unique(dat$site)

> # you need to define an empty data.frame to hold the data

> sites.means=data.frame()

> for (i in 1:length(sites)) {

+ MySubset<-subset(dat,site==sites[i]) #only date on the i-th site

+ sites.means<-rbind(sites.means,colMeans(MySubset[,c("X","Y")]))

+ } #for i

> names(sites.means)=c("mean.X","mean.Y")

> #the above does not have the site name, but since this was in order of

> # sites we can cbind this column

> sites.means$site<-sites

> head(sites.means)

mean.X mean.Y site

1 1.095090 1.218722 A1

2 1.957245 1.604291 A2

3 2.784740 1.324065 A3

4 4.082257 1.602554 A4

5 5.191368 2.296375 A5

6 5.619714 2.254710 A6

The above uses a loop for (i in 1:length(sites)) { CODE } to repeat what is the “CODE” for each value of i. The value of i is stated to be “in” the vector 1:length(sites) which is the vector 1:100 as there are 100 sites (i.e. 1,2,3, …, 100). Note further that I annotated behind the closing curly bracket “} #for i” what is closed.

How does the for loop work? In the first round, i = 1, each line of code between the curly brackets is processed until the closing curly bracket and then the program starts again from the first line after the opening curly bracket (on the line with the for command) for the second round. In the second round i = 2, etc. etc. After the 100th time i = 100 and now the for loop stops and the program continues with the next line after the closing curly bracket “} #for i”.

Above I use the for function in a “classic” fashion (for (i in 1:length(sites))) in which an index number, arbitrarily called i , is allowed to vary from 1 to as many sites as there are. The identity of the site is then identified by using the index number, thus specifying sites[i], meaning that in the first round we are considering sites[1] and in the second round of the loop, we are considering sites[2], etc. Such a construction is not necessary in R. This is because the for function already implies to R that we have a vector and that we assign the consecutive values of this vector to i. In the above code, the vector is 1:length(sites), and i thus is assigned the value 1,2,3,…. Alternatively, we can use the vector sites, and assign its consecutive values to i; in which case i is assigned “A1”, “A2”, “A3”, … Using this formulation, we would not need to identify the specific site as sites[i], but would use i instead. I illustrate the approach below, but note that to avoid confusion with the example above I do not use i, but assign to s the consecutive entries in the vector sites. Script would look like

> sites.means=data.frame()

> for (s in sites) {

+ MySubset<-subset(dat,site==s)

+ sites.means<-rbind(sites.means,colMeans(MySubset[,c("X","Y")]))

+ } #for s

These two implementations work otherwise the same and it is a matter of taste how you do it.

This example – as you well know by now – is simply meant to illustrate a for() loop. It codes what can be easily done in a single line using e.g. aggregate().

> sites.means.fast<-aggregate(dat[,c("X","Y")],by=dat["site"],mean)

> head(sites.means.fast)

site X Y

1 A1 1.09509 1.218722

2 A10 10.25517 3.609373

3 A100 99.98842 9.838488

4 A11 10.93560 2.937553

5 A12 12.17747 3.348020

6 A13 13.04435 3.111437

In the next section, we look at exactly what is the difference between apply() family functions and for loops.

## Chapter 15 apply() family revisited

**15.1 apply()**

One of the strengths of R is the functions within the so-called apply() family (apply, tapply, lapply, sapply, mapply, aggregate). These functions are all meant to “apply” a function repeatedly to an object. We have introduced and worked with some of these apply() family function. The function to be applied repeatedly can be an R function or a function coded for a specific purpose by yourself including an anonymous function. Because of the repeated nature, using an apply() function often is more effective than a for() loop as we saw in chapter 14. With more effective I mean that there is less need/temptation to create additional objects (which you easily do when coding for loops) and hence clearer coding. apply() may or may not be faster than a for() loop. For large data-sets (images, geographical information, genetic data from NGS, Big Data), for() loops may start to become rather time consuming. Moreover, specific approaches in R and packages to carry out such analyses typically include purpose-specific adaptations of the basic apply() functions. Hence, the logic of using apply() will typically translate also to more specific approached. For example, for carrying out spatial analyses in the raster package, there are apply() operations applicable to raster objects.

The function apply() follows the similar logic as tapply(). However, instead of specifying as the second argument for which subsets a function is to be applied, apply() asks for the dimension (1 = row, 2 = column, etc) over which a function is to be applied. For example, we want to calculate the number of NAs per variable in a matrix with 10 columns.

> datm<-matrix(rnorm(1000),100,10) #this creates a matrix of 100 rows and 10 columns filled with random numbers

> datm[datm<0]<-NA #numbers smaller than zero are set as NA

> apply(datm,2,function(m) sum(is.na(m))) #returns n of NA values per col

[1] 50 53 52 45 53 51 43 46 58 54

Here, an anonymous function is applied over datm ‘s second dimension (i.e. its columns) giving the number of NAs for each column.

Remember: Working with apply() requires writing functions. In the above example, we used an anonymous function (a function defined within the call to apply which is not assigned to an object). Clearly we could also have written

> HowManyNAs<-function(m) {sum(is.na(m))}

> apply(datm,2,HowManyNAs)

[1] 50 53 52 45 53 51 43 46 58 54

As we have seen before, all functions in the apply() family allow one to include additional arguments that are passed to the function called within apply. Consider the following examples

> apply(datm,2,mean) #will produce NAs

[1] NA NA NA NA NA NA NA NA NA NA

> apply(datm,2,function(m) mean(m,na.rm=T)) #anonymous function

[1] 0.7422612 0.9544298 0.8616230 0.8880168 0.9509261 0.7516653 0.6780077 0.6956566 0.7709907 0.7978700

> apply(datm,2,mean,na.rm=T) #pass the removal of NA values along to function mean

[1] 0.7422612 0.9544298 0.8616230 0.8880168 0.9509261 0.7516653 0.6780077 0.6956566 0.7709907 0.7978700

## 15.2 sapply() and lapply()

The functions sapply() (for simplify apply) and lapply() (list apply) are highly similar. The function sapply() will (try to) produce a vector and lapply() will produce a list. Like other members of the apply() family functions, sapply()and lapply() are a versatile tool for rapidly “running through” data. These functions typically are used when one does not work along pre-defined rows and columns or elements in a list, as in the above examples or with subsets as in tapply(). More flexible approaches can also be implemented. For example, perhaps we have data on a two factorial design

> treatments = c("control","treated")

> datf<-data.frame(data = rnorm(1000), treatment =treatments[rbinom(1000,1,0.5)+1])

> head(datf)

data treatment

1 0.4028300 control

2 -0.4383698 control

3 -2.0057118 treated

4 0.1910955 treated

5 -0.2685560 treated

6 -0.5539900 treated

We then want to calculate some statistic for each level of the treatment using e.g. sapply. We can do so by passing the different treatments

> sapply(treatments,function(m) mean(datf[datf[,"treatment"]==m,1]))

control treated

0.0001870327 -0.0538581376

The two levels of the treatment (“control” and “treatment”) are here passed, each in turn, as “m” into the anonymous function and then used to filter the data within the function mean. The above code assumes implicitly the data (object datf) is defined. We can make this aspect more explicit by passing the object datf along as a special argument to the anonymous function defined within sapply

> sapply(treatments,function(m,dat.to.use) mean(datf[datf[,"treatment"]==m,1]),dat.to.use=datf)

control treated

0.0001870327 -0.0538581376

Note that these examples are meant to illustrate the possibilities of the apply family. The last example can of course be coded effectively using tapply or aggregate (another member of the apply family).

> aggregate(datf$data, list(datf$treatment),mean)

Group.1 x

1 control 0.0001870327

2 treated -0.0538581376

Another example would be to use sapply() to produce repeated plots.

> datGroups<-data.frame(x=c(1,2,2,3,2,2,3,4,3,4,4,5,6), y=c(2,2,3,2,4,3,5,4,4,5,6,7,8), group=c(rep(1,6),rep(2,7)))

> datGroups

x y group

1 1 2 1

2 2 2 1

3 2 3 1

4 3 2 1

5 2 4 1

6 2 3 1

7 3 5 2

8 4 4 2

9 3 4 2

10 4 5 2

11 4 6 2

12 5 7 2

13 6 8 2

> layout(matrix(1:2,1,2))

> sapply(1:2,function(m) plot(y~x,subset(datGroups,group==m)))

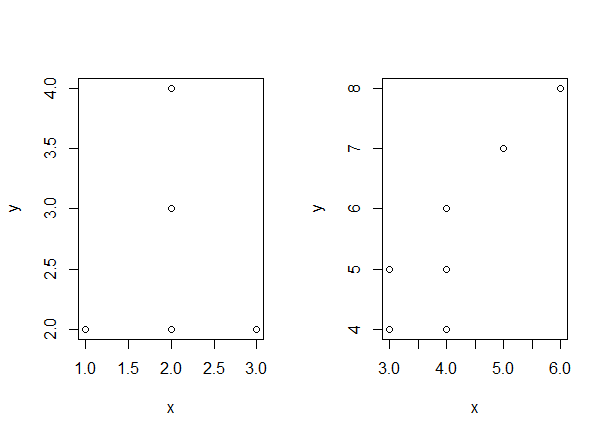
[[1]]

NULL

[[2]]

NULL

produces the following two-panel quick plots



As in the example above, an alternative and potentially more transparent coding would be

> sapply(1:2,function(m,dat) plot(y~x,subset(dat,group==m)),dat=datGroups)

or then to write out the function

> mySubsetPlot<-function(mySubset,dat) {

+ plot(y~x,subset(dat,group==mySubset))

+ }

> sapply(1:2,mySubsetPlot,dat=datGroups)

where the vector 1:2 is taken to be the first argument in the function mySubsetPlot() and the second argument is provided as the third argument to sapply() similar as we encountered before for tapply() and aggregate(). Of course this function could be easily expanded upon with additional lines of code, e.g. to produce more fancy plots adding several addition layers or calculation of additional statistics.

These examples are meant to give an overview of how function-based coding works. In function-based coding the objective is to code a function (explicit or anonymous) to carry out a specific procedure and use a member of the apply()family to repeat this function. You can potentially code extensive repeated operations in this way especially when writing out the function. The alternative is of course to use a for() loop. The above as a for() loop would be

> datGroups<-data.frame(x=c(1,2,2,3,2,2,3,4,3,4,4,5,6), y=c(2,2,3,2,4,3,5,4,4,5,6,7,8), group=c(rep(1,6),rep(2,7)))

> layout(matrix(1:2,1,2))

> for (g in 1:2) {

+ plot(y~x,subset(datGroups,group==g))

+ }

The potential advantage of function-based coding is (potentially) increased speed, as well as the potential to write more general procedures. That is, the line sapply(1:2,mySubsetPlot,dat=datGroups) can also take another data.frame as the third argument dat (provided this data.frame has a subset called group). Whether you find this a handy way of coding compared to using a for() loop (i.e. not function-based coding) is perhaps a matter of taste at this point.

## 15.3 apply() or for() loop

To some extent, whether you use for() loops or apply() family functions is a matter of taste. However, apply() cannot replace all for() loops (but any apply() job can be coded as a for() loop). The main difference between an apply() operation and a for() loop is that in a call to apply() the order of the iteration (the repeat “job” to carry out) cannot be important.

In section 5.1, for example, we could shuffle the order of the columns around in the matrix and apply the same operation to get the same answer. The order of the columns is not important. A for() loop is the only option whenever the order is important. A typical example is any Markovian process where the value of the next step depends on the outcome of the previous one. For example, in iterating population dynamics: That is, to calculate the population size at time step 10, we need to know the population size at time step 9, etc… We cannot calculate such a process where the order of the time step would be shuffled around. Thus, we must use a for() loop in such a case.

In deciding whether to use apply() or a for() loop, we are talking about differences in coding “styles”. If the order in which you handle the subsets is not important, you can write a function, even a very long one, and then use apply() rather than a for() loop. Again, which way to solve the coding problem is at this point largely up to what you find most intuitive.

## 15.4 Exercises part V

V.1 The space separated file “birds.txt” contains info on ringed individuals, their sex and size

(a) read in the data

(b) check that the coding of the rings were typed consistently and correct if needed

(c) fix the date so it becomes a numeric for day, month and year (3 columns)

(c) how many birds were processed on each day? (TIP: a robust way to handle dates is to turn them into a counter (days since some arbitrary start date), e.g. Julian date)

(d) the file “birds\_observer” contains for each bird the ID of the observer. Read in and link this file to the above data file

(e) What is the mean size of birds recorded per observer?

(f) What is the mean size of male and female birds recorded per observer?

(f) How many male and female birds were recorded per observer?

(g) How many birds were measured by “you” and “I”?

(h) Produce a barplot of the mean size of male and female birds measured by the two observers (i.e. 2 x 2 categories = 4 bars) with the sample size printed in the bars just above the X-axis

V.2 The file “dynamics.txt” which you can find in the Moodle page of this course contains results of 1000 simulations (rows) of population size over 100 years (in columns)

(a) Read in this information into R

(b) calculate the expected population size (i.e. the mean) for each of the 100 time steps

(c) plot the trajectory of the mean population size over time as good as you can

(d) use the function quantile to calculate the lower and upper 95% interval over all 1000 simulations for each of the 100 time steps

(e) use the function polygon to plot both the “envelope” in which 95% of simulated values lie as well as the mean, as in the following plot (please make the plot look better!)

TIP: A number of functions draw or add things to your plot (like arrows, polygon, lines, text, segments, axis, …), but they all require that the plotting space is defined first using a plot command. However, you do not actually need to plot anything using plot…

V.3

The file “XY.txt” contains data.

1. Read in this data as the object myDat
2. The following snippet of code runs a linear regression analysis where the results are stored in the objectmyStats

myStats<-lm(y~x, data=myDat)

1. Explore the object myStats. Typically, object produced by statistical analyses are of a class specific to that analysis (the function lm produces an object of class lm). However, is this a matrix, data.frame or list type of object and why?
2. If you print the object myStats you see

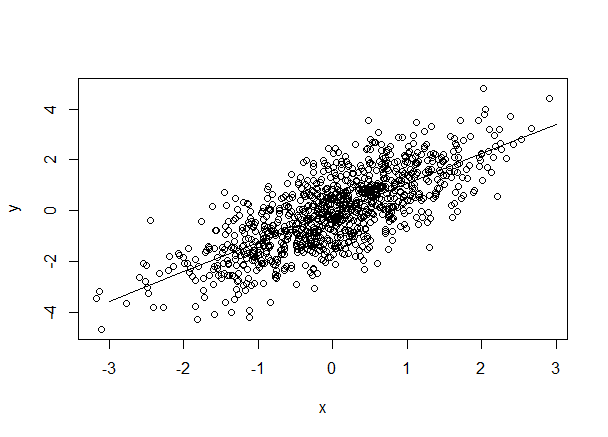
Coefficients:

(Intercept) x

-0.07829 1.16187

where the first number is the Intercept and the second the slope such that the regression models tells you y = -0.07829 + 1.16187. Find the intercept and slope within the object myStats so you can access them

1. Produce a scatterplot of y against x and draw the regression line (as stated in (e) above) through the plot. the “raw” plot should look like

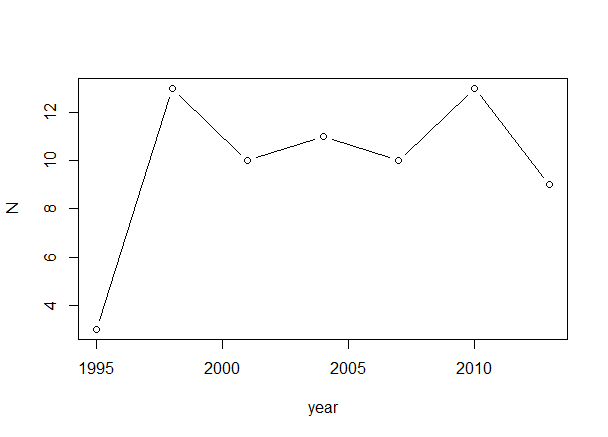


1. Make the plot look nicer (name the axes, increase font size, plots filled dots,..)
2. Produce a 2 panel plot containing in (a) fitted values against y and in (b) residuals against x. All the information needed for plotting is to be taken from the object myStats (it is all in there…)
3. Add to myStats a new tag called name which holds your name

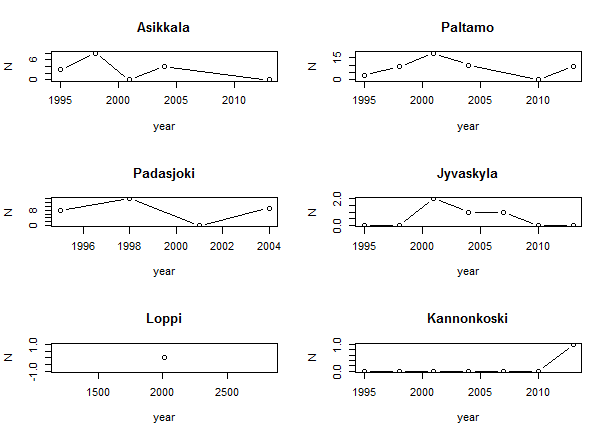
V.4

Read in the datafile “PopData” which contains counts of winter nests of beavers in Finland per municipality. The counts are made every third year

1. make sure there is only one count per municipality per year in this data
2. how many counts are there per municipality? make a histogram of the number of counts
3. plot the dynamics as a line with dots for the municipality Virrat



1. write a for loop to plot the dynamics of any six municipalities in a single plot



1. as in d but using an apply() family function

V.5

Calculating descriptive stats of many files

The zipped folder “spectroData” contains files. Each file contains data on reflectance measured with a photospectrometer. The first column contains the wavelengths (in nm), column 2-4 are 3 measures of reflectance obtained on one subject. The subject measured is given in the file name and the header of the columns 2:4.

Unzip all these files in a separate directory.

Write a script that reads in all files into R and calculates the following parameters for each measure on each subject:

* Brightness: The average reflectance in the interval 300-700 nm
* Hue: Maximal reflectance in the interval 300-700 nm

The following data.frame needs to be produced

Subject ID measure Brightness Hue

1015V 1 .. ..

1015V 2 .. ..

-- etc

TIP: As always. Approach the solution in steps. Start by coding when reading in one file, and calculating the desired parameters. Then figure out how to add one more to this, and finally automate. dir() will produce a vector with all file names in your working directory.

## MY CODE: Part 5:

#V.1  
#The space separated file "birds.txt" contains info on ringed individuals, their sex and size  
#(a) read in the data  
  
setwd("E:/Introduction to R")  
Birdsdata <- read.table("birds.txt", header = TRUE,  
 stringsAsFactor = FALSE)  
  
  
#(b) check that the coding of the rings were typed consistently and   
correct if needed  
  
head(Birdsdata)

## date rings sex size  
## 1 02.01.2017 1001v female 10.419545  
## 2 02.02.2017 1002V male 8.475233  
## 3 02.01.2017 1003v female 8.109559  
## 4 01.01.2017 1004v male 9.050083  
## 5 02.02.2017 1005v female 4.869759  
## 6 01.01.2017 1006v male 10.980671

#The typing of the letter "V" is inconsistent, let's fix it:  
Birdsdata$rings <- sub("v", "V", Birdsdata$rings)  
head(Birdsdata)

## date rings sex size  
## 1 02.01.2017 1001V female 10.419545  
## 2 02.02.2017 1002V male 8.475233  
## 3 02.01.2017 1003V female 8.109559  
## 4 01.01.2017 1004V male 9.050083  
## 5 02.02.2017 1005V female 4.869759  
## 6 01.01.2017 1006V male 10.980671

#doing the same with a for loop: (Done in class)  
Birdsdata$rings <- as.character(Birdsdata$rings)  
a <- 1  
for (i in Birdsdata$rings) {  
 substr(i, 5,5) <- "V"  
 print(i)  
 Birdsdata$rings[a]<-i  
 a<-a+1  
}

## [1] "1001V"  
## [1] "1002V"  
## [1] "1003V"  
## [1] "1004V"  
## [1] "1005V"  
## [1] "1006V"  
## [1] "1007V"  
 #... I removed the rest of the print for this compendium

#(c) fix the date so it becomes a numeric for day, month and year (3   
columns)  
  
 #How to seperate the elements in the column and unlist them to a   
 continuous vector  
DaysVectors <- unlist(strsplit(Birdsdata$date, ".", fixed = TRUE))  
  
 #Making a matrix out of the vector:  
DaysMatrix <- matrix(data = DaysVectors, nrow = 1000, byrow = TRUE)  
  
 #Making the matrix into a dataframe so it can be bound to the Birdsdata   
 later and naming the columns  
DaysDataframe <- as.data.frame(DaysMatrix)  
colnames(DaysDataframe) <- c("Day", "Month", "Year")  
  
 #Binding the columns of our dataframe with days, months,  
 years to the Birds data  
Birdsdata <- cbind(Birdsdata, DaysDataframe)  
head(Birdsdata)

## date rings sex size Day Month Year  
## 1 02.01.2017 1001V female 10.419545 02 01 2017  
## 2 02.02.2017 1002V male 8.475233 02 02 2017  
## 3 02.01.2017 1003V female 8.109559 02 01 2017  
## 4 01.01.2017 1004V male 9.050083 01 01 2017  
## 5 02.02.2017 1005V female 4.869759 02 02 2017  
## 6 01.01.2017 1006V male 10.980671 01 01 2017

#transforming the columns to numeric  
Birdsdata$Day <- as.numeric(Birdsdata$Day)  
Birdsdata$Month <- as.numeric(Birdsdata$Month)  
Birdsdata$Year <- as.numeric(as.character(Birdsdata$Year))  
  
class(Birdsdata$Day)

## [1] "numeric"

class(Birdsdata$Month)

## [1] "numeric"

class(Birdsdata$Year)

## [1] "numeric"

View(Birdsdata)  
  
 #or do it with a for loop (in class)  
a<-1  
for (i in 1:length(Birdsdata$date)) {  
 c<-strsplit(as.character(Birdsdata$date[i]), "[.]")  
 print(i)  
 Birdsdata$Day[i]<- as.numeric(c[[1]][[1]])  
 Birdsdata$Month[i]<- as.numeric(c[[1]][[2]])  
 Birdsdata$Year[i]<- as.numeric(c[[1]][[3]])  
 a<-a+1  
 print(a)  
   
}

## [1] 1  
## [1] 2  
## [1] 2  
## [1] 3  
## [1] 3  
## [1] 4  
## [1] 4  
## [1] 5  
## [1] 5  
## [1] 6  
## [1] 6  
## [1] 7  
## [1] 7  
 #...

#(c) how many birds were processed on each day? (TIP: a robust way to   
handle dates is to turn them into a counter (days since some arbitrary   
start date), e.g. Julian date)  
  
aggregate(size ~ date, data = Birdsdata, FUN = length)

## date size  
## 1 01.01.2016 113  
## 2 01.01.2017 144  
## 3 01.02.2016 137  
## 4 01.02.2017 116  
## 5 02.01.2016 107  
## 6 02.01.2017 129  
## 7 02.02.2016 138  
## 8 02.02.2017 116

#(d) the file "birds\_observer" contains for each bird the ID of the   
observer. Read in and link this file to the above data file  
  
Birds\_observer\_Data <- read.table("birds\_observer.txt")  
Merge1 <- merge(Birdsdata, Birds\_observer\_Data, by.x = "rings",  
 by.y = "V1")  
  
 #I renamed the columns so that "obeserver" was included as a column name  
colnames(Merge1) <- c("rings", "date", "sex", "size", "Day",  
 "Month", "Year", "observer")  
head(Merge1)

## rings date sex size Day Month Year observer  
## 1 1001V 02.01.2017 female 10.419545 2 1 2017 you  
## 2 1002V 02.02.2017 male 8.475233 2 2 2017 you  
## 3 1003V 02.01.2017 female 8.109559 2 1 2017 you  
## 4 1004V 01.01.2017 male 9.050083 1 1 2017 you  
## 5 1005V 02.02.2017 female 4.869759 2 2 2017 you  
## 6 1006V 01.01.2017 male 10.980671 1 1 2017 I

#(e) What is the mean size of birds recorded per observer?   
  
aggregate(size ~ observer, data = Merge1, FUN = mean)

## observer size  
## 1 I 10.001742  
## 2 you 9.960281

#(f) What is the mean size of male and female birds recorded per observer?  
  
aggregate(size ~ observer+sex, data = Merge1, FUN = mean)

## observer sex size  
## 1 I female 9.915172  
## 2 you female 9.974466  
## 3 I male 10.091531  
## 4 you male 9.943601

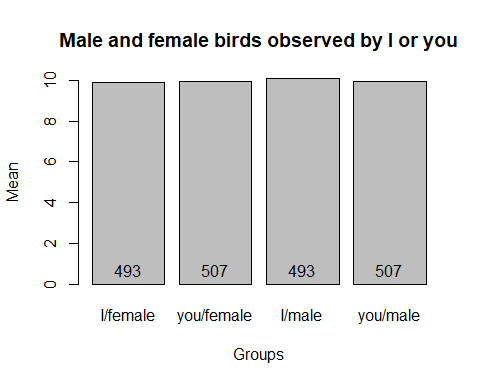
#(f) How many male and female birds were recorded per observer?  
  
aggregate(size ~ observer+sex, data = Merge1, FUN = length)

## observer sex size  
## 1 I female 251  
## 2 you female 274  
## 3 I male 242  
## 4 you male 233

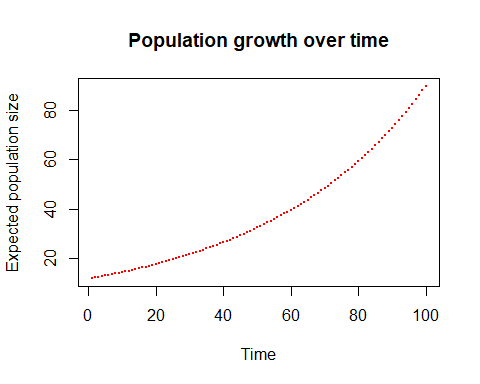
#(g) How many birds were measured by "you"and "I"?  
  
Length\_Sex\_Observer <- aggregate(size ~ observer, data = Merge1,  
 FUN = length)  
Length\_Sex\_Observer

## observer size  
## 1 I 493  
## 2 you 507

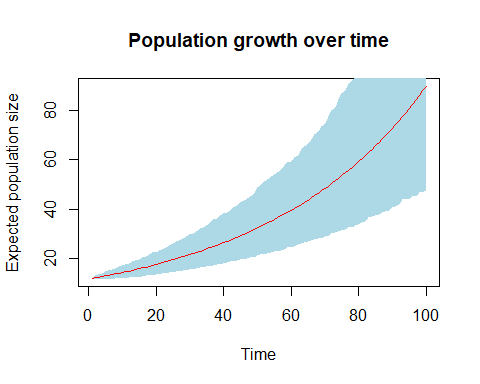
#(h) Produce a barplot of the mean size of male and female birds measured by the two observers (i.e. 2 x 2 categories = 4 bars) with the sample size printed in the bars just above the X-axis  
  
Means\_Sex.Observer <- aggregate(size ~ observer+sex, data = Merge1,  
 FUN = mean)  
  
Barplot1 <- barplot(Means\_Sex.Observer$size, xlab = "Groups",  
 ylab = "Mean",  
 main = "Male and female birds observed by I or you",   
 names.arg = c("I/female", "you/female", "I/male",  
 "you/male"))  
  
 #adding the sample size above the x-axis  
text(Barplot1, 0.7, Length\_Sex\_Observer$size)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#V.2 The file "dynamics.txt" which you can find in the Moodle page of this course contains results of 1000 simulations (rows) of population size over 100 years (in columns)   
  
#(a) Read in this information into R  
  
DynamicsData <- read.table("dynamics.txt")  
  
  
#(b) calculate the expected population size (i.e. the mean) for each of   
the 100 time steps  
  
 #Counting the mean of the columns. The "2" indicates the "second   
 dimension" = columns.   
MeanColumns <- apply(DynamicsData,2,mean)  
  
  
#(c) plot the trajectory of the mean population size over time as good as you can  
  
plot(MeanColumns, xlab = "Time", ylab = "Expected population size",  
 col = "red", cex = 0.2, pch = 16,  
 main = "Population growth over time" )



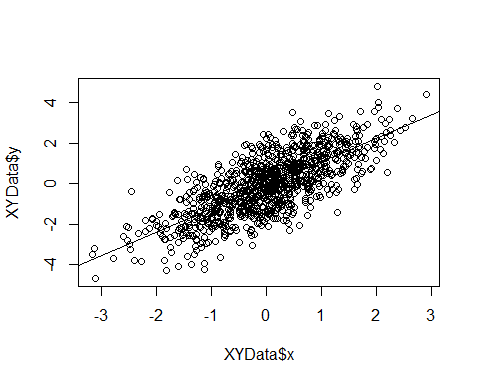
#(d) use the function quantile to calculate the lower and upper 95%   
interval over all 1000 simulations for each of the 100 time steps  
  
QuantileColumns <- apply(DynamicsData,2,quantile, probs = c(0.025, 0.975))  
  
  
#(e) use the function polygon to plot both the "envelope" in which 95% of simulated values lie as well as the mean, as in the following plot (please make the plot look better!)   
#TIP: A number of functions draw or add things to your plot (like arrows, polygon, lines, text, segments, axis, ..), but they all require that the   
plotting space is defined first using a plot command. However, you do not actually need to plot anything using plot  
  
 #creating coordinates for the polygon  
x <- c(1:100,100:1)  
y <- c(QuantileColumns[1,], sort(QuantileColumns[2,], decreasing = TRUE))  
 #alternative way to flip the vector y  
y <- c(QuantileColumns[1,], QuantileColumns[2,100:1])  
  
 #plottin with no dots/lines (type), done in class  
plot(MeanColumns, xlab = "Time", ylab = "Expected population size",  
 col = "red", cex = 0.2, pch = 16,  
 main = "Population growth over time",  
 type = "n")  
  
polygon(x, y, col = "lightblue", border = NA)  
lines(MeanColumns, col = "red")



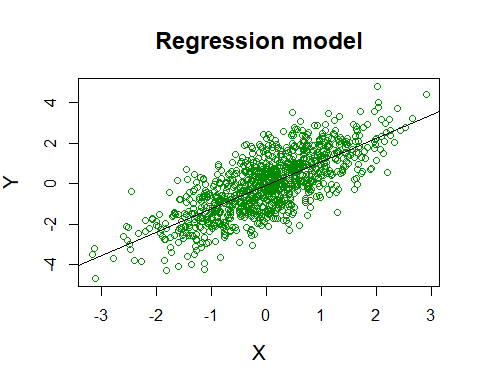
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#V.3   
#The file "XY.txt" contains data.   
  
#(a) Read in this data as the object myDat  
  
 #(XYData = myData in this exercise)  
XYData <- read.table("XY.txt", header = TRUE)  
  
#(b) The following snippet of code runs a linear regression analysis   
where the results are stored in the object myStats  
  
myStats<-lm(y~x, data=XYData)  
  
#a) Explore the object myStats. Typically, object produced by statistical analyses are of a class specific to that analysis (the function lm   
produces an object of class lm). However, is this a matrix, data.frame or list type of object and why?  
  
#View(myStats)  
  
 #myStats is a list. If you View it says under type that it is a list con   
 taining 12 objects of varying types.   
  
#b) If you print the object myStats you see  
  
print(myStats)

##   
## Call:  
## lm(formula = y ~ x, data = XYData)  
##   
## Coefficients:  
## (Intercept) x   
## -0.07829 1.16187

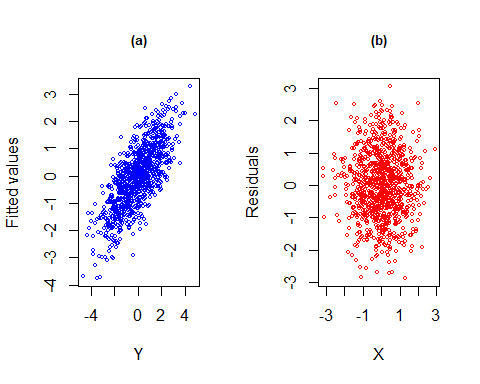
#where the first number is the Intercept and the second the slope such   
that the regression models tells you y = -0.07829 + 1.16187. Find the   
intercept and slope within the object myStats so you can access them  
  
Intercept <- myStats$coefficients[1]  
Slope <- myStats$coefficients[2]  
  
  
#c) Produce a scatterplot of y against x and draw the regression line (as stated in (b) above) through the plot. The "raw" plot should look like  
  
plot(XYData$x, XYData$y)  
   
abline(lm(y~x, data=XYData))  
 #this makes the same line:  
abline(myStats$coefficients)



#d) Make the plot look nicer (name the axes, increase font size, plots   
filled dots,..)  
  
plot(XYData$x, XYData$y, xlab = list("X", cex = 1.3),  
 ylab = list("Y", cex = 1.3),  
 col = "green4",  
 main = list("Regression model", cex = 1.5))  
abline(myStats$coefficients)



#e) Produce a 2 panel plot containing in (a) fitted values against y and   
in (b) residuals against x. All the information needed for plotting is to be taken from the object myStats (it is all in there!)  
  
 #The key to getting two panels  
par(mfrow = c(1,2))  
  
plot(myStats$model$y, myStats$fitted.values,  
 main = list("(a)", cex = 0.8),  
 xlab ="Y" , ylab = "Fitted values", cex = 0.5, col = "blue")  
  
plot(myStats$model$x, myStats$residuals, main = list("(b)", cex = 0.8),  
 xlab = "X", ylab = "Residuals", cex = 0.5, col = "red")



#f) Add to myStats a new tag called name which holds your name   
  
myStats$name <- "Janni"  
myStats$name

## [1] "Janni"  
  
  
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#V.4  
#Read in the datafile "PopData" which contains counts of winter nests of   
beavers in Finland per municipality. The counts are made every third year  
  
PopData <-read.csv2("PopData.csv")  
head(PopData)

## Municipality year N  
## 1 Asikkala 2001 0  
## 2 Padasjoki 2001 0  
## 3 Nastola 2004 0  
## 4 Asikkala 2013 0  
## 5 Hollola 2013 0  
## 6 Humppila 2013 0

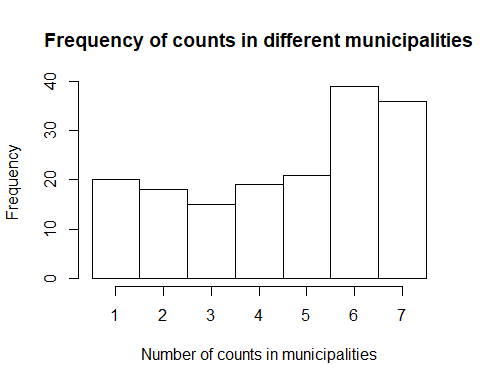
tail(PopData)

## Municipality year N  
## 763 Mikkeli 1998 233  
## 764 Savonlinna 2007 256  
## 765 Lieksa 2001 264  
## 766 Mikkeli 2001 280  
## 767 Savonlinna 2001 317  
## 768 Savonlinna 1998 400

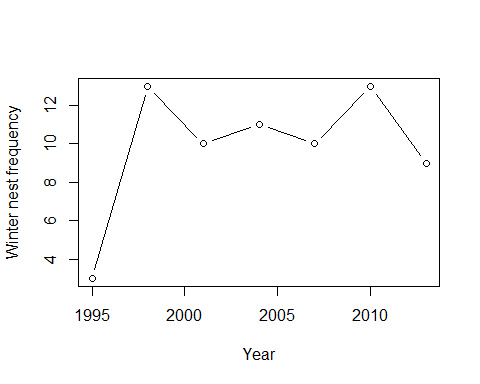
#(a) make sure there is only one count per municipality per year in this data  
  
 #aggregating the sample size of each municipality and year  
CountsYearMunicipality <- aggregate(N ~ Municipality + year, data = PopData, FUN = length)  
  
 #checking that the vector we just created only has the number one in   
 it's "N"-column  
  
length(which(!CountsYearMunicipality$N==1))

## [1] 0

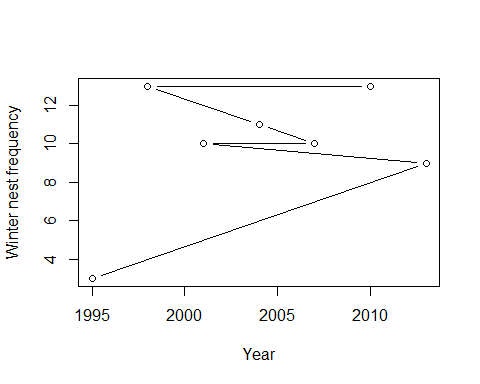
#The number of values in the column with the sample size per muncipality   
 per years which is not = 1 is 0!  
  
  
#(b) How many counts are there per municipality? make a histogram of the number of counts  
  
CountsMunicipality <- aggregate(N ~ Municipality, data = PopData,  
 FUN = length)  
  
 #changing the parameters for plot so that there is room for only one  
 plot  
par(mfrow = c(1,1))  
  
hist(CountsMunicipality$N, xlab = "Number of counts in municipalities",  
 main = "Frequency of counts in different municipalities",  
 breaks = seq(0.5,7.5, 1))  
 #breaks = c(1:7)



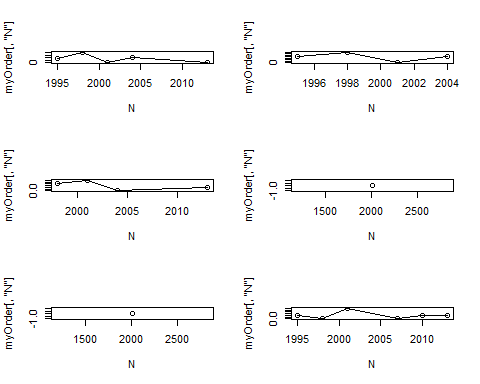
#(c) plot the dynamics as a line with dots for the municipality Virrat  
  
 #Crating a dataframe with only the Virrat-rows  
VirratData <- subset(PopData, PopData$Municipality == "Virrat")  
  
 #Changing the data frames order so it is orderd accoring to the year -   
 otherwise it will dras the line horisontally according to the   
 frequency!   
VirratData <- VirratData[order(VirratData$year),]  
  
plot(VirratData$year, VirratData$N, type = "b",  
 xlab = "Year", ylab = "Winter nest frequency")



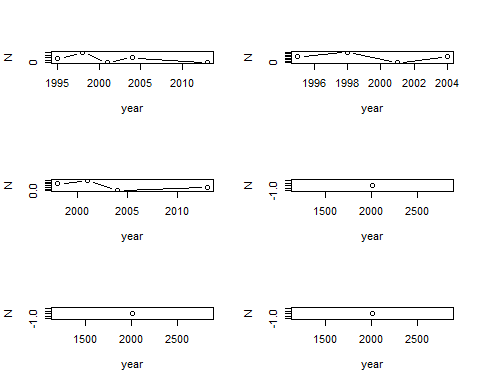
#otherwise it will look like this:  
VirratData <- subset(PopData, PopData$Municipality == "Virrat")  
plot(VirratData$year, VirratData$N, type = "b",  
 xlab = "Year", ylab = "Winter nest frequency")



#a) write a for loop to plot the dynamics of any six municipalities in a   
single plot  
  
 #How we did it in class:  
municipal <- unique((PopData$Municipality))  
par(mfrow=c(3,2))  
  
for (i in c(1:5,10)) {  
 mysubset <- subset(PopData, Municipality ==as.character(municipal[i]))  
 myOrder <- mysubset[order(mysubset[,"year"]),]  
   
 plot(myOrder[,"year"], myOrder[,"N"], xlab = "N")  
 lines(myOrder[,"year"], myOrder[,"N"])  
   
}



#b) as in d but using an apply() family function  
  
 #How we did it in class:  
PopData <- PopData[order(PopData$year),]  
sapply(1:6, function(m) plot(N~year,  
 subset(PopData, Municipality==as.character(municipal[m])),  
 type = "b"))



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#V.5   
#Calculating descriptive stats of many files   
  
#The zipped folder "spectroData" contains files. Each file contains data   
on reflectance measured with a photospectrometer. The first column contains the wavelengths (in nm), column 2-4 are 3 measures of reflectance obtained on one subject. The subject measured is given in the file name and   
the header of the columns 2:4.  
  
#Unzip all these files in a separate directory.  
#Write a script that reads in all files into R and calculates the   
following parameters for each measure on each subject:   
  
#Brightness: The average reflectance in the interval 300-700 nm  
  
#Hue: Maximal reflectance in the interval 300-700 nm  
  
#The following data.frame needs to be produced  
#Subject ID measure Brightness Hue  
#1015V 1 .. ..  
#1015V 2 .. ..  
#-- etc  
  
 #How we did it in class:  
  
setwd("E:/Introduction to R/SpectroData")  
namesFiles <- dir()  
  
result <- data.frame()  
  
for (n in 1:615) {  
 file <- read.csv(namesFiles[n], sep = ";")  
 validWaveLengths <- subset(file, file[,1] >= 300 & file[,1] <= 700)  
 meansColumns <- colMeans(validWaveLengths)  
   
   
   
 subjectmeasureBrightHue<- data.frame(SubjectID = strsplit(namesFiles[n], "[.]")[[1]][1], measure = c(1,2,3), brightness = colMeans(validWaveLengths[,2:4]), hue = apply(validWaveLengths[,2:4],2,max))  
   
 result <- rbind(result, subjectmeasureBrightHue)  
}  
  
head(result)

## SubjectID measure brightness hue  
## X1001V 1001V 1 3.651486 6.059817  
## X1001V.1 1001V 2 6.358666 9.988772  
## X1001V.2 1001V 3 6.120168 9.737166  
## X1002V 1002V 1 3.158160 6.322204  
## X1002V.1 1002V 2 4.207818 7.589168  
## X1002V.2 1002V 3 4.428078 7.768416

tail(result)

## SubjectID measure brightness hue  
## X1614V 1614V 1 6.305754 10.506393  
## X1614V.1 1614V 2 6.302629 10.286879  
## X1614V.2 1614V 3 5.412998 9.413704  
## X1615V 1615V 1 1.569529 3.492700  
## X1615V.1 1615V 2 3.504584 6.991194  
## X1615V.2 1615V 3 3.700666 6.852716



# Part VI:

**Chapter 16 Array and tabulation**

## 16.1 Array

We have come across the class array implicitly, because tapply() produces an array. As we have seen, a vector has one dimension, and a matrix has two dimensions. An array is a general extension of this same concept to *n*-dimensions. For example a 3-dimensional array can be thought of as matrices stacked on top of each other (rows and columns stacked). In >3 dimensions, my imagination does not accommodate visualization anymore, but R is happy to accept it. For example, an array consisting of NA values in 4 dimensions

> array(NA, dim=c(2,2,2,2))

, , 1, 1

[,1] [,2]

[1,] NA NA

[2,] NA NA

, , 2, 1

[,1] [,2]

[1,] NA NA

[2,] NA NA

, , 1, 2

[,1] [,2]

[1,] NA NA

[2,] NA NA

, , 2, 2

[,1] [,2]

[1,] NA NA

[2,] NA NA

In the above, the first two dimensions are row and column (2 rows and 2 columns). The third and fourth dimension of this array each also hold two “layers”. Similar to vector and matrix, an array is either filled with numbers or with characters, but not a combination of the two. One way to fill the array is by indexing

> myArray<-array(NA, dim=c(2,3,2))

> myArray[,,1]<-matrix(c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray[,,2]<-matrix(6+c(1,2,3,4,5,6),2,3,byrow=TRUE)

> myArray

, , 1

[,1] [,2] [,3]

[1,] 1 2 3

[2,] 4 5 6

, , 2

[,1] [,2] [,3]

[1,] 7 8 9

[2,] 10 11 12

The above thus first defines an empty array and then fills it with two matrices “stacked” in the third dimension. Just like a matrix or vector, the “slots” in an array can be named. Suppose we construct an array with information (e.g. presence (1) or absence (0) for several X and Y coordinates for two species.

> mySpeciesArray<-array(NA, dim=c(2,3,2),dimnames=list(c("x1","x2"),c("y1","y2","y3"),c("species1","species2")))

> mySpeciesArray[,,1]<-matrix(c(0,1,1,1,0,1),2,3,byrow=TRUE)

> mySpeciesArray[,,2]<-matrix(c(1,0,0,0,1,0),2,3,byrow=TRUE)

> mySpeciesArray

, , species1

y1 y2 y3

x1 0 1 1

x2 1 0 1

, , species2

y1 y2 y3

x1 1 0 0

x2 0 1 0

The usual functions which we have introduced before can also be used for arrays. For example, functions calculating means and sums over rows and columns (which are optimized to be as fast as possible).

> colMeans(myArray,dims=2)

[1] 3.5 9.5

The above calculate the mean and sum for values in each “matrix stack” in the third dimension of the array. To see this we can calculate the mean of each matrix

> mean(matrix(c(1,2,3,4,5,6)))

[1] 3.5

> mean(matrix(6+c(1,2,3,4,5,6)))

[1] 9.5

We can also use rowMeans() to calculate the mean of values “slicing” through the third dimension

> rowMeans(myArray,dims=2)

[,1] [,2] [,3]

[1,] 4 5 6

[2,] 7 8 9

Similarly, the apply() function is of course highly useful here. The above can be coded as

> apply(myArray,3,mean)

[1] 3.5 9.5

> apply(myArray,c(1,2),mean)

[,1] [,2] [,3]

[1,] 4 5 6

[2,] 7 8 9

which may or may not be more intuitive than using colMeans() and rowMeans().

## 16.2 Factorial data: Tabulation

Data often is factorial, meaning there are categories (which may be ordinal or nominal). We have already looked at tapply() and aggregate() as examples of functions used to analyze this type of data when summarizing information per level of a factor. In statistical applications, tabulation is a basic tool for processing data prior to contingency testing (chi-square and higher dimensional contingency tables). Tabulation of multiple factorial variables is also often used for investigating sample sizes for each cross-categorical level in your data (examples below). Data exploration is an important aspect of analyses. The functions table(), xtabs() and split() are useful to this end.

The function table() will tabulate the numbers of observation for factorial levels in the data. For example,

> df3<-data.frame(individual=c("ID1","ID1","ID2","ID2","ID3"), col=c("blue","blue","blue","green","green"),habitat=c("bush","field","field","bush","field"))

> df3

individual col habitat

1 ID1 blue bush

2 ID1 blue field

3 ID2 blue field

4 ID2 green bush

5 ID3 green field

> df3.tab<-table(df3)

> df3.tab

, , habitat = bush

col

individual blue green

ID1 1 0

ID2 0 1

ID3 0 0

, , habitat = field

col

individual blue green

ID1 1 0

ID2 1 0

ID3 0 1

> class(df3.tab)

[1] "table"

The df3.tab object resembles an array (but technically is of class table), and can be processed further. For example, to get the sample sizes break-down for the cross-classified categories

> colSums(df3.tab)

habitat

col bush field

blue 1 2

green 1 1

The above is equivalent to using apply to sum all cells *per* 2nd **and** 3rd dimension (color and habitat respectively), i.e. c(2,3). That is,

> apply(df3.tab,c(2,3),sum)

habitat

col bush field

blue 1 2

green 1 1

The function xtabs() for cross tabulation uses the formula object to specify which factors are to be cross tabulated. This is a clever function for cross-tabulation. A formula object includes always the ~ and one then needs to specify for which data the required formula is to be used. Many functions in R accept arguments as a formula object, familiar by now from the function aggregate(). For example, we may want to know per individual what habitat they were recorded in

> xtabs(~individual+habitat,data=df3)

habitat

individual bush field

ID1 1 1

ID2 1 1

ID3 0 1

For another example we can suppose we have data on parasites of the individuals which is added to our data.

> df3$parasites<-c(1,2,5,0,6)

> xtabs(parasites~col+habitat,data=df3)

habitat

col bush field

blue 1 7

green 0 6

Cross tabulation, because of its formula, presents a perhaps more intuitive approach to quickly sum cases belonging to specific categories, which is typically handy to investigate how e.g. sample sizes or other variables (which can be summed) break down across different categories. As we have seen, also tapply() and aggregate() or other functions can be used to sum. The difference is that table(), xtabs() and tapply() produce an “array-like” object (potentially multidimensional)

> tapply(df3$parasites,list(df3$col, df3$habitat),sum)

bush field

blue 1 7

green 0 6

whereas aggregate() produce a 2-dimensional “data.frame-like” object. Consider the above breakdown using aggregate()

> aggregate(parasites~col+habitat,data=df3,sum)

col habitat parasites

1 blue bush 1

2 green bush 0

3 blue field 7

4 green field 6

There are hence multiple solutions possible in R allowing you as user to choose the format most useful to your need.

To use xtabs() to get a break-down of sample sizes per cross-classified factorial variables, you may be tempted to try the following:

> xtabs(individual~col+habitat,data=df3) #will not work

Error in Summary.factor(1L, na.rm = FALSE) :

sum not meaningful for factors

But that does not work since individual is not considered something to sum up (it is a factorial variable). One options I sometimes use is to add a column of 1 (1 for each individual in this case) and sum up over that

> df3$one<-1

> xtabs(one~col+habitat,data=df3)

habitat

col bush field

blue 1 2

green 1 1

Again, this is just one way. You can compare with the options above where we used table() and summed the columns, or the use of aggregate() and tapply() for calculating sample size as discussed in earlier parts.

The take-home message is that there is an arsenal of possibilities for tabulation and calculating sums and means and other simple descriptive stats of your data. These are powerful tools to investigate your data (especially when it is large) and to calculate sample sizes and statistics for different categories.

**Chapter 17. Simulation**

## 17.1 Simulation

A simulation is a common application of programming. With a simulation, you can explore the theoretical aspects of a certain idea in a general sense. Simulations are especially useful when this idea cannot be studied in terms of analytical mathematics, either because it is too complicated, because you want to verify that your results indeed are correct, or because your mathematical skills are simply not that developed to actually solve the problem analytically. Lack of mathematical skill in e.g. algebra is nothing to be overly depressed about when you have a computer which can perform huge amounts of calculations! Of course, programming a simulation requires another form of mathematical skills. In particular, many simulations are based on random processes. Thus, understanding probability processes is a major part in coding simulations. The main reason for simulating stochastic (i.e. probabilistic) processes is that such processes quickly become difficult to track analytically. They are, however, relatively straightforward when simulated numerically thanks to the immense speed of calculations in modern computers.

The dark side of simulations is that, in order to be as general as possible, you typically need to consider many different possibilities and simulations quickly become time consuming, even on a fast computer. Optimizing the speed of your code has in the previous chapters not been an issue. For the kind of problems we considered thusfar, the main criterium is that it works. This is because the majority of problems we considered thusfar require reaching a solution only once or a few times. For example, when you need to link two datafiles together and extract information, the program needs to do this only once and then it is ready. In a simulation, the same procedure is repeated many times and thus coding this procedure in the most time-efficient manner will rapidly start to pay off in reducing the overall computational time. Coding more demanding simulations, therefore, starts to make demands on your skills as a programmer. Still, it is good to keep in mind that even an inefficient algorithm will provide you with the answer; you just need to take longer coffee-break or let the computer run over night / during your holiday. That’s what I do!

## 17.2 Statistical distributions

R is very much a platform for statistics. It therefore holds all the usual statistical distributions, which often are handy for executing simulations. Most distributions are available with the letters d, p, q and r in front of them, which stands for density, probability, quantile, and random, respectively. For example, the normal distribution (norm). From the help we learn

---------------------------------------------------------------------------------------------------------------------------------

**Description**

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to mean and standard deviation equal to sd.

**Usage**

dnorm(x, mean = 0, sd = 1, log = FALSE)

pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)

qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)

rnorm(n, mean = 0, sd = 1)

-----------------------------------------------------------------------------------------------------------------------------------

Similar modifications are made of the binomial distribution (binom), Poisson distribution (pois), uniform distribution (unif) and others. For example, a single random number drawn from a normal distribution with mean 0 and standard deviation (sd) of 1 – the default values for these arguments – is simply

> rnorm(1)

[1] -0.7074996

Likewise, the probability to obtain the value of 1.96 or lower from a normal distribution with mean 0 and sd = 1 is

> pnorm(1.96)

[1] 0.9750021

The function sample() is useful for making random draws from a known set of numbers, either with or without replacement. For example, I have 10 observations and I may want to “shuffle” these observations in a random order, but I do not want to use the same observation twice (once a value is drawn, it is not replaced in the set of numbers to draw from, i.e. no replacement). One realization of the indices of these 10 observations in random order would be

> sample(10,10)

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

Sampling with replacement would generate

> sample(10,10,replace=T)

[1] 10 2 3 10 5 10 7 3 9 4

## 17.3 What is a simulation? Some examples

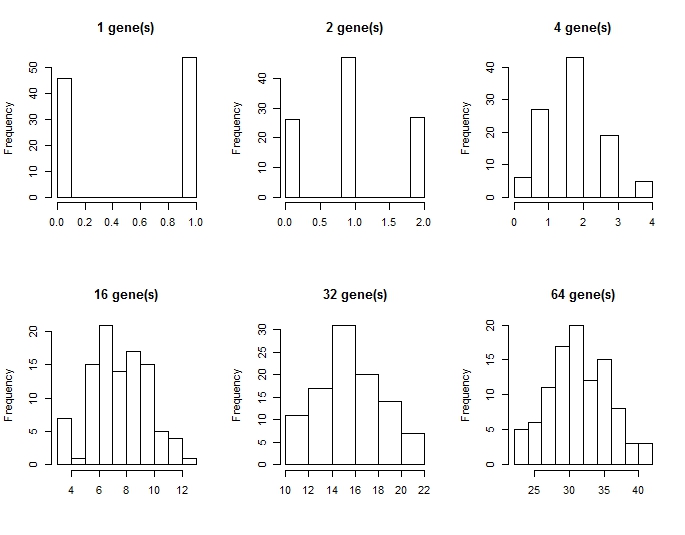
Simulations in a wide sense entail basically any programming task where you want to obtain some understanding of an interesting process using numerical calculations. For example, when Mendelian genetics was rediscovered in the early 1900s, people were very skeptical of evolution through natural selection. This was because the Mendelian experiments concerned discrete states (pea flowers were white or red) whereas most variation in nature was normally distributed including lots of intermediate variants between the two extremes. Scientists could not reconcile genetics with the observation of natural variation. One part of the Modern Synthesis was when RA Fisher solved this apparent paradox. His solution was to assume that traits were affected by multiple genes, where each gene only has a small effect. Mind, this was decades before the discovery of DNA. One way to understand it is to simulate Fisher’s theory. We assume there are *N* genes. Each of these genes either increases the mean of a trait by 1 or it has no effect (i.e. the effect sizes are +1 and 0). For any genotype, whether it is turned on is for each of these *N* genes a stochastic process. Because there are only two directions (‘off’ or ‘on’), it is a random binomial process. If we draw the expected frequency distribution of a trait for 100 random genotypes, assuming the trait is determined by 1, 2, 4, 16, 32, or 64 genes we go from a discrete distribution (red or white) to obtain a normal distribution.

> ngenes<-c(1,2,4,16,32,64)

> par(mfrow=c(2,3))

> for (i in 1:length(ngenes)){

+ hist(replicate(100,rbinom(1,ngenes[i],0.5)),main=paste(as.character(ngenes[i]), "gene(s)"),xlab=" ") }



Another useful application of simulation is data generation which can be performed before planning experiments or collecting data or after performing statistical analyses to check whether you can be confident about your results (remember that absence of evidence is not evidence of absence).

A simulation allows you to put together a dataset with known properties. You can then analyze many simulated datasets to verify that a certain code you have written to calculate these properties indeed works. For example, you write code to simulate mean temperature for a number of meadows. You simulate data with temperatures measured 10 times in 5 meadows and for which you assume the mean increase by one degree for each of these five meadows

> Data.meadows<-data.frame("Meadow.ID"=rep(seq(1:5),10), "Temperature"= NA)

> for ( i in 1:5){

+ Data.meadows[Data.meadows$Meadow.ID==i,2]<-rnorm(10,i,0.1)

+ }

> mean.temp<-aggregate(Temperature~Meadow.ID, Data.meadows, mean)

> mean.temp

Meadow.ID Temperature

1 1 1.016489

2 2 1.971542

3 3 2.994640

4 4 4.021103

5 5 4.997875

This example is of course rather contrived, as you would not typically be that concerned with testing such an obvious function as the calculation of the mean. The key point, however, is that data with a known property is simulated to test the function for the calculation of this property (in this case, the mean).

## 17.4 Storing information

A simulation could also concern a statistical test. The simulation in such cases could, for example, be a randomization process which concerns your data. Here, instead of assuming a statistical distribution with certain properties (e.g. the normal distribution), you use randomization to perform your statistical test. One of the advantages of randomization tests is that they are less reliant on the statistical distribution and can work also when a statistical distribution is not obvious. Clearly, when you perform this type of analyses, you end up with lots of values. Thus, experience with storing data in R is useful prerequisite. In R, you can store values in objects of various classes, for example in a matrix, array or data.frame for further analysis or export it to become a Table. We have already encountered a number of useful functions, mainly belonging to the apply family (apply, tapply, lapply, sapply, aggregate). These form a very versatile “backbone” for calculating descriptive statistics (means, sample sizes) and properties of your data.

## 17.5 Correlation example

Suppose that variable Y and X are measured 20 times in 100 different sites. The data is included in a data.frame called dat which in its columns has information on 1. site (a factor), 2. Y and 3. X. We can make this data by simulation

> sites=vector("character",length=0) #initialize the vector

> X=vector("numeric",length=0) #initialize the vector

> Y=vector("numeric",length=0) #initialize the vector

> for (s in 1:100) {

+ sites<-c(sites, rep(paste("A",s,sep=""),20))

+ X<-c(X,rnorm(20,s,1) )

+ Y<-c(Y,rnorm(20,s^.5,2) )

+}

> dat=data.frame(sites,X,Y)

> names(dat)=c("site","X","Y")

> summary(dat)

site X Y

A1 : 20 Min. : -0.3927 Min. :-2.768

A10 : 20 1st Qu.: 25.5230 1st Qu.: 4.598

A100 : 20 Median : 50.3463 Median : 6.920

A11 : 20 Mean : 50.4666 Mean : 6.667

A12 : 20 3rd Qu.: 75.7369 3rd Qu.: 8.824

A13 : 20 Max. :102.5700 Max. :15.711

(Other):1880

R has the function by for this sort of procedure. For example,

> by(dat, dat$site, function(m) cor.test(~X+Y,m)$estimate)

dat$site: A1

[1] 0.1393261

------------------------------------------------------------

dat$site: A10

[1] -0.3753485

------------------------------------------------------------

dat$site: A100

[1] 0.2004982

------------------------------------------------------------.

.

.

Here the correlation is calculated for each site and printed on the screen. You of course want to capture this output in an object and store it in a data.frame. Here, sapply() is handy

> sapply(as.character(unique(dat$site)), function(m) cor.test(~X+Y,data=subset(dat,site==m))$estimate)

A1.cor A2.cor A3.cor A4.cor A5.cor A6.cor A7.cor A8.cor

-0.093652740 -0.106637265 0.277442326 -0.293219903 -0.161676232 0.354503678 0.186317221 -0.013437900

A9.cor A10.cor A11.cor A12.cor A13.cor A14.cor A15.cor A16.cor

-0.084801428 -0.123946225 0.011772327 -0.235175733 0.303227859 0.033135281 0.367818472 -0.159603447

A17.cor A18.cor A19.cor A20.cor A21.cor A22.cor A23.cor A24.cor

-0.418181208 0.353193343 -0.165017518 0.351212353 -0.008100790 -0.066809312 -0.126526975 -0.105642485

...

What is going on here? You can check the help for cor.test(). The first part (~X+Y) specifies in the so-called “function” format which variables cor.test() will calculate the correlation for. The data is a subset of dat, specifying to only consider each site. The coercion of site to be a character vector (as.character) is needed here because site is a factor. The function cor.test() produces an object with the following structure (use the function str to see this)

> str(cor.test(~X+Y,data=subset(dat,site==as.character(sites[1])))

List of 9

$ statistic : Named num -0.141

..- attr(\*, "names")= chr "t"

$ parameter : Named int 31

..- attr(\*, "names")= chr "df"

$ p.value : num 0.889

$ estimate : Named num -0.0253

..- attr(\*, "names")= chr "cor"

$ null.value : Named num 0

..- attr(\*, "names")= chr "correlation"

$ alternative: chr "two.sided"

$ method : chr "Pearson's product-moment correlation"

$ data.name : chr "X and Y"

$ conf.int : atomic [1:2] -0.365 0.321

..- attr(\*, "conf.level")= num 0.95

- attr(\*, "class")= chr "htest"

Because we are here interested in the correlation coefficient, which is called estimate. We access this directly by using the $ sign. Hence

> cor.test(~X+Y,data=subset(dat,site== as.character(sites[1])))$estimate

cor

0.2004982

gives the correlation between Y and X for site i. Similarly, the $conf.int gives the confidence interval. Clearly, we could have included other pieces of information (such as the P value) in a similar fashion.

One alternative to storing your data in a data.frame is to use a for() loop. First define sites.corr as an empty list and fill it with the information needed and then convert it to a data.frame. In this case, you need to use rbind() within the for() loop to add rows. Something like

> #list

> # which are the sites

> sites<-unique(dat$site)

> # empty list to hold the data. The elements of the list need names

> # note the use of the vector function here

> sites.corr=list(

+ site=NULL

+ ,corr=NULL

+ ,CI=NULL

+ )

> for (i in 1:length(sites)) {

+ sites.corr$site=rbind(sites.corr$site,as.character(sites[i]))

+ sites.corr$corr=rbind(sites.corr$corr

+ ,cor.test(~X+Y,data=subset(dat

+ ,site==as.character(sites[i])) )$estimate)

+ sites.corr$CI=rbind(sites.corr$CI

+ ,cor.test(~X+Y,data=subset(dat

+ ,site==as.character(sites[i])) )$conf.int)

+ } #for

> sites.corr.frame<-data.frame(sites.corr)

> sites.corr.frame

site cor CI.1 CI.2

1 A1 0.1393261 -0.3231164 0.5480566

2 A2 0.002020341 -0.4408946 0.444144

3 A3 -0.01128978 -0.4515546 0.4333962

...

The advantage of using the list format is that it deals automatically with storing 2 parameters. For example, the CI in the above example becomes after conversion to a data.frame CI.1 (lower CI) and CI.2 (upper CI). Using names() the headers of the data.frame can of course be renamed to more intelligible names if so desired.

## 17.6 Automating functions

In the above example, we wanted to calculate a statistic for subsets (of rows) of the data. In other cases, it may be needed to calculate a statistics for each of a number of variables. For example, we may want to explore what happens to the relationship between Y and X if we add noise to X. We can do this by appending extra columns to dat to consist not only Y and X, but also X2, X3, X4, X5, stored in columns 4:7 of dat. These extra columns are the value of X with added random numbers drawn from a normal distribution with increasing standard deviation.

> dat$X2<-dat$X+rnorm(length(dat$X),0,10)

> dat$X3<-dat$X+rnorm(length(dat$X),0,30)

> dat$X4<-dat$X+rnorm(length(dat$X),0,60)

> dat$X5<-dat$X+rnorm(length(dat$X),0,100)

> head(dat)

site X Y X2 X3 X4 X5

1 A1 -0.54824543 1.092888 3.491997 -31.392790 53.335874 154.47870

2 A1 2.53309615 -1.488752 5.260736 5.546408 79.135956 -45.28982

3 A1 2.20298908 3.779022 13.656302 -8.263379 -65.086277 -25.82910

4 A1 1.44612986 -1.865175 17.136321 -35.222721 111.813535 -133.41463

5 A1 -0.02184114 2.918691 8.006858 -11.029653 8.370312 140.84532

6 A1 2.12115603 -1.108438 8.828050 -9.435287 73.919308 -58.90706

Now, we want to calculate the regression coefficients (intercept and slope) for each of these variables against Y. This is of course a job for sapply().

> sapply(c(2,4:7),function(m) lm(dat$Y~dat[,m])$coef)

[,1] [,2] [,3] [,4] [,5]

(Intercept) 2.73132895 3.14207806 4.8578194 5.93474709 6.459233646

dat[, m] 0.07905026 0.07064563 0.0369711 0.01517776 0.005595341

where the first row denotes the value of the intercept and the second row the value for the slope of Y against each of the explanatory variables (listed in columns X to X5). These are the coefficients of the linear regression given by the call lm(dat$Y~dat[,m])$coef. We can see that the slope of the regression declines to zero as we add noise to X (called “attenuation” in statistics). Clearly, the function which sapply() uses (which is now anonymous) can be expanded to report a vector of also other statistics. This would require writing out the function.

An alternative approach is to construct a for() loop and make use of the possibility to construct a statistical formula based on a character vector using as.formula(). For example,

> Xcoef=list(var=vector("list"), coef=vector("list")) #for storing

> for (col in c(2,4:7)) {

+ mod<-lm(as.formula(paste("Y~", names(dat)[col])), data= dat)

+ Xcoef$var<-rbind(Xcoef$var, names(dat)[col])

+ Xcoef$coef<-rbind(Xcoef$coef,mod$coef)

+ } #for (col

> Xcoef.frame=data.frame(Xcoef)

> Xcoef.frame

var coef..Intercept. coef.X

1 X 2.731329 0.07905026

2 X2 3.142078 0.07064563

3 X3 4.857819 0.0369711

4 X4 5.934747 0.01517776

5 X5 6.459234 0.005595341

In the above the as.formula() function is used to specify formula Y~X, Y~X2, Y~X3, … The end result is an object which holds (in rows) the name of the variable and the intercept and slope of the regression of Y on each of these variables. Note that in this loop a new linear regression model is made each ‘round’. If more info on the regression is needed then this needs to be stored explicitly which requires extending what is stored in Xcoef.

## 17.7 Exercises belonging to Part VI

**VI.1** Using data “FishData.txt”

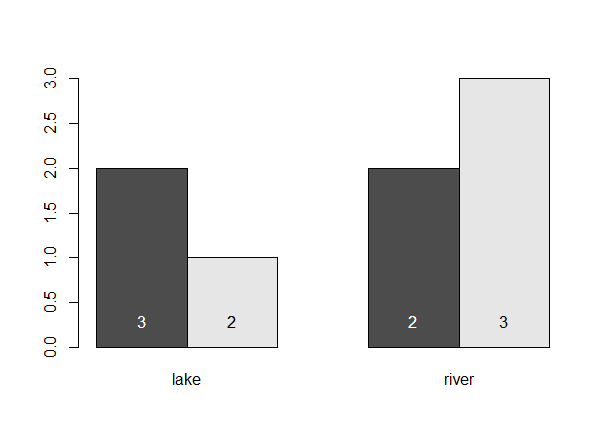
calculate

(a) number of infected and uninfected per tank

(b) number of infected and uninfected per origin

(c) 3D array-like object with per tank (3rd dimension), whether each individual (row) was infected or not (column)

(d) barplot of the number of infected individuals per origin and tank with the total number of fish printed above the X axis. Basic plot like this (but yours should look better)



**VI.2**

The following data is a very small part of the two first breeding bird atlases in Finland. The data is available online from the Zoological Museum. Breeding birds were censused in 10km x 10km squares covering the whole of Finland. The data can be interpreted to give the occupancy status for each bird species in Finland. Hence, each atlas is a map of the national distribution of breeding birds. Thus, the atlas allows calculating various properties of these distributions.

In the data below, species are anonymized, coordinates of cells are given using ordered northerlies and easterlies (i.e. numbers), such that cell (1,2) is east of (1,1) etc. The data otherwise follows the format it has been made available. We here only use this small part of the data to be able to check our results easily but the coding to be done will work also on the real dataset; it could easily be performed on the entire dataset (>3800 cells per atlas, > 300 bird species)

species N E status1 status2 occupied1 occupied2

a 1 1 1 0 TRUE FALSE

a 4 3 1 0 TRUE FALSE

b 3 2 1 1 TRUE TRUE

b 2 3 1 0 TRUE FALSE

b 4 1 1 0 TRUE FALSE

c 1 1 0 1 FALSE TRUE

c 4 3 1 0 TRUE FALSE

d 1 3 1 0 TRUE FALSE

d 2 1 1 1 TRUE TRUE

d 3 2 1 1 TRUE TRUE

e 5 2 1 1 TRUE TRUE

e 1 1 1 1 TRUE TRUE

e 3 3 1 1 TRUE TRUE

1. Get the data into R
2. Create an array that represents the data for atlas 1 (status1 / occupied1). The array thus has occupancy for each northerly N in the first dimension, easterly E in 2nd dimension and species in third dimension. The dimensions are named. Thus,

> atlas1[,,”a”]

1 2 3

1 TRUE NA NA

2 NA NA NA

3 NA NA NA

4 NA NA TRUE

5 NA NA NA

or something similar like

E

N 1 2 3

1 1 0 0

2 0 0 0

3 0 0 0

4 0 0 1

5 0 0 0

depending on how you did this

The logic here is that we get this matrix because species “a” occupied cell (1,1) and (4,1) (N gives the cell coordinate row, E the cell coordinate column)

TIP: there are a number of ways to do this, but remember that this is a form of tabulation….

1. Create an array AtlasData that follows the logic of the above assignment (b) but also holds the second atlas in the array’s fourth dimension
2. In one apply command, you can now obtain the sum of cells each species occupies in each of the two atlases (i.e. its “range size”). What is this one line? Approach the solution by
3. first figuring out how to do this on the atlas1 array created in (b),
4. and then extend this to the AtlasData array

**VI.3:** Simulate a dataset

You are studying a wild population of birds and you would like to investigate the effect of tarsus length on longevity. One way to do it is to run a mark-recapture analysis which is a model that allows you to calculate the probability of survival of individuals from one year to the other as a function of different covariates. Although you have not received yet the data from your supervisor, you want to practice running such analysis for a toy dataset. You know that this population has been monitored at every breeding season for 5 years by two field assistants and that the population size is about 200 individuals. The mark-recapture model uses the encounter history of each individual (whether it was caught or not each year). The encounter history is a character string stating ”1” when the birds was caught (= encountered) and “0” when not. For example, “01100” means the birds was caught in the second and third occasions. Each individual in the data has one encounter history, which therefore looks like:

Ring history

1001V 01100

1002V 10101

1003V 10110

etc.

1. Use a binomial distribution to generate encounter histories as described above for 200 individuals whose Ring follows the above logic (1001V, 1002V, …). The probability of encounter is 0.6
2. Each bird’s tarsus length has been measured on the first time it was caught. Each bird was measured by one out of two observers, but one observer consistently gave higher measures than the other. Create a dataset ready to be analyzed: it should in addition to the previous one (individual identities + encounter history), also contain the covariates observer and tarsus length.
3. The calipers used to measure the birds’ tarsus, which are very sensitive to humidity, did not work on ten occasions because of heavy rainfalls and the birds were not measured on subsequent breeding seasons. Change the tarsus length measurement to NA for ten randomly chosen birds.

**VI.4**: Simulate population dynamics

The expected size *N* of a population which is growing stochastically (the growth rate changes at every time step) can be predicted at any time *t* (t = 0, 1, 2, … tmax) given its initial population size (*N0*) using the following formula:

*Nt+1=Nt\*exp(rt)* where *rt*follows a normal distribution with mean *r* and standard deviation *sdr*

1. Simulate the growth of a population over 30 time steps for *N0*=100, mean *r*= 0.1 and *sdr*=0.05 and plot it. (TIP: consider carefully your first value of *t* in thissimulation).
2. Using a for loop, repeat the operation 30 times and plot each growth curve on the same graph. You should store the simulated data at each iteration.
3. Plot the average population growth and its 95%CI envelope as you did in Part V.

**VI.5:** Power analysis, function and loop

You are planning an experiment, in which you want to test whether product “substance X” has an effect on the growth of saplings. You know the growth of saplings is on average 100 (arbitrary units), with a standard deviation of 5. You expect that “substance X” will lead to a 10% increase in growth, without changing the standard deviation. You have at your disposal limited facilities and want to write a simulation exploring how many saplings you should include in the experiment. The basic test you want to do is a t-test comparing control (untreated) saplings to treated (“substance X” treated) saplings, and the number of saplings (N) are of the same size in both these groups.

1. Create a data.frame with random numbers of growth for 10 untreated and 10 treated saplings
2. Run a t-test on the data you created using t.test()
3. Create a function which does a) and b) in one run for any expected % of increase (inc) and for any sample size in both groups (n). The function should return the probability of the growth of control and treated saplings being equal which is given by the p-value of the t-test.

The function should start like:

My\_fun<-function(n, inc){

…

}

1. Write a script that runs n.sim number of simulations of your growth experiment on N saplings and returns the vectors of p-values for each round of the simulation.
2. Can you apply this program to inform you about how many saplings you should have in your experiment?

## MY CODE: Part 6:

#VI.1 Using data "FishData.txt"  
#calculate  
  
setwd("E:/Introduction to R")  
FishData <- read.table("FishData.txt", header = TRUE)  
  
#(a) number of infected and uninfected per tank  
  
aggregate(individual ~ infected + tank, data = FishData, FUN = length)

## infected tank individual  
## 1 no 1 1  
## 2 yes 1 4  
## 3 no 2 1  
## 4 yes 2 4

#OR  
xtabs(~tank+infected, data = FishData)

## infected  
## tank no yes  
## 1 1 4  
## 2 1 4

#(b) number of infected and uninfected per origin  
  
aggregate(individual ~ infected + origin, data = FishData, FUN = length)

## infected origin individual  
## 1 no lake 2  
## 2 yes lake 3  
## 3 yes river 5

#OR  
xtabs(~origin+infected, data = FishData)

## infected  
## origin no yes  
## lake 2 3  
## river 0 5

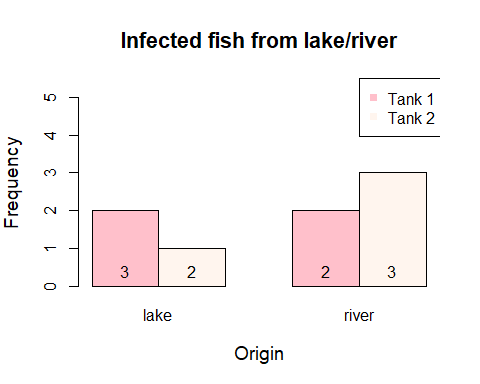
#OR  
table(FishData$tank, FishData$infected)

##   
## no yes  
## 1 1 4  
## 2 1 4

#(c) 3D array-like object with per tank (3rd dimension), whether  
#each individual (row) was infected or not (column)   
  
mysubset <- data.frame(individual = FishData$individual,  
 infected = FishData$infected,  
 tank = FishData$tank)  
myarray <- table(mysubset)  
myarray

## , , tank = 1  
##   
## infected  
## individual no yes  
## ID1 0 1  
## ID10 0 0  
## ID2 0 0  
## ID3 0 1  
## ID4 0 0  
## ID5 1 0  
## ID6 0 0  
## ID7 0 1  
## ID8 0 0  
## ID9 0 1  
##   
## , , tank = 2  
##   
## infected  
## individual no yes  
## ID1 0 0  
## ID10 0 1  
## ID2 0 1  
## ID3 0 0  
## ID4 1 0  
## ID5 0 0  
## ID6 0 1  
## ID7 0 0  
## ID8 0 1  
## ID9 0 0

#(d) barplot of the number of infected individuals per origin and tank   
with the total number of fish printed above the X axis. Basic plot like   
this (but yours should look better)  
  
 #Subseting the rows that have infected individuals  
subsetinfected <- subset(FishData, infected == "yes")  
  
 #Creating the sata for the barplot  
barplotdata<- tapply(subsetinfected$infected,  
 list(subsetinfected $tank,subsetinfected $origin),  
 length)  
  
 #barplot  
mybarplot <- barplot(barplotdata, beside = TRUE, ylim = c(0,5.5),  
 main = list("Infected fish from lake/river",  
 cex = 1.4),  
 xlab = list("Origin", cex = 1.2),  
 ylab = list("Frequency", cex = 1.2),  
 col = c("pink", "seashell"))  
   
 #Making the data for the total number of fish, to print on the bars  
totalfish <- tapply(FishData$infected,  
 list(FishData$tank,  
 FishData$origin), length)  
  
 #adding the total sample size to the bars  
text(mybarplot, 0.4, totalfish)  
legend(5, 5.5, c("Tank 1", "Tank 2"), pch = 15,  
 col = c("pink", "seashell"), cex = 1)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VI.2   
  
#(a) Get the data into R  
  
atlasdata <- read.table("atlas.txt", header = TRUE)  
  
  
#(b) Create an array that represents the data for atlas 1 (status1 /  
 occupied1). The array thus has occupancy for each northerly N in the   
first dimension, easterly E in 2nd dimension and species in third   
dimension. The dimensions are named.   
  
Species.N.E <- xtabs(occupied1~N+E+species, data = atlasdata)  
Species.N.E

## , , species = a  
##   
## E  
## N 1 2 3  
## 1 1 0 0  
## 2 0 0 0  
## 3 0 0 0  
## 4 0 0 1  
## 5 0 0 0  
##   
## , , species = b  
##   
## E  
## N 1 2 3  
## 1 0 0 0  
## 2 0 0 1  
## 3 0 1 0  
## 4 1 0 0  
## 5 0 0 0  
##   
## , , species = c  
##   
## E  
## N 1 2 3  
## 1 0 0 0  
## 2 0 0 0  
## 3 0 0 0  
## 4 0 0 1  
## 5 0 0 0  
##   
## , , species = d  
##   
## E  
## N 1 2 3  
## 1 0 0 1  
## 2 1 0 0  
## 3 0 1 0  
## 4 0 0 0  
## 5 0 0 0  
##   
## , , species = e  
##   
## E  
## N 1 2 3  
## 1 1 0 0  
## 2 0 0 0  
## 3 0 0 1  
## 4 0 0 0  
## 5 0 1 0

#The logic here is that we get this matrix because species "a" occupied   
cell (1,1) and (4,1) (N gives the cell coordinate row, E the cell   
coordinate column)  
#TIP: there are a number of ways to do this, but remember that this is a  
#form of tabulation!  
  
#(c) Create an array AtlasData that follows the logic of the above  
#assignment (b) but also holds the second atlas in the array's fourth   
dimension  
  
 #Creating the array of atlas 2:  
Species.N.E.2 <- xtabs(occupied2~N+E+species, data = atlasdata)  
  
 #Creating an ampty array  
newarray <- array(NA, dim = c(5, 3, 5, 2))  
  
 #Filling that arrays fourth dimension with the two atlases  
newarray[,,,1] <- Species.N.E  
newarray[,,,2] <- Species.N.E.2  
  
 #Naming the dimensions  
dimnames(newarray) <- list(1:5, 1:3, c("a","b","c","d","e"),  
 c("Atlas1", "Atlas2"))  
  
  
#(d) In one apply command, you can now obtain the sum of cells each  
species occupies in each of the two atlases (i.e. its "range size").   
What is this one line? Approach the solution by   
  
#(i) first figuring out how to do this on the atlas1 array created in (b),   
  
 #Array with atlas1 = Species.N.E. The summary is done on the third  
 dimension  
apply(Species.N.E, 3, sum)

## a b c d e   
## 2 3 1 3 3

#(ii) and then extend this to the AtlasData array   
  
 #Array with atlas 1 and 2 = newarray. You can list several dimensions!   
Species are the third dimension, and the atlases are the fourth dimension (first is rows, second is columns)  
  
apply(newarray, c(3,4), sum)

## Atlas1 Atlas2  
## a 2 0  
## b 3 1  
## c 1 1  
## d 3 2  
## e 3 3

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VI.3: Simulate a dataset  
  
#(a) Use a binomial distribution to generate encounter histories as   
described above for 200 individuals whose Ring follows the above logic   
(1001V, 1002V, ...). The probability of encounter is 0.6   
  
 #Creating the "Ring" vector  
Ring <- paste(1001:1200, "V", sep = "")  
head(Ring)

## [1] "1001V" "1002V" "1003V" "1004V" "1005V" "1006V"

#Creating a 200x5 matrix with out of a vector with ones and zeros  
created with the probability of 0.6 for 1   
binom.matrix <- matrix(data = sample(c(0,1), size = 1000, replace = TRUE,  
 prob = c(0.4, 0.6)),  
 nrow = 200, ncol = 5,  
 byrow = TRUE,  
 dimnames = list(1:200, 1:5))  
  
 #combining the contents of the columns in the matrix to make up the  
 "history" column -> one column with 5-digit "codes" comprised of  
 different combinations of 0 and 1.   
history <- paste(binom.matrix[,1], binom.matrix[,2], binom.matrix[,3],  
 binom.matrix[,4], binom.matrix[,5])  
  
RingedBirdsData <- data.frame(Ring, history)  
head(RingedBirdsData)

## Ring history  
## 1 1001V 1 1 1 0 1  
## 2 1002V 0 0 1 1 1  
## 3 1003V 0 1 0 1 1  
## 4 1004V 0 1 0 0 1  
## 5 1005V 1 0 1 1 1  
## 6 1006V 1 1 1 0 0

#How to do it with a for loop (done in class)  
nbirds <- 200  
history <- c()  
  
for (i in 1:nbirds){  
 x<-rbinom(5, 1, 0.6) #The middle argument: How many times will it roll  
 the odds  
 history[i]<-paste(x, collapse = "")  
   
}  
  
vector <- c(1001:1200)  
rings <- paste(vector, "V", sep = "")  
Birdsdata <- data.frame(rings, history)  
head(Birdsdata)

## rings history  
## 1 1001V 11111  
## 2 1002V 01011  
## 3 1003V 10111  
## 4 1004V 11111  
## 5 1005V 01111  
## 6 1006V 01010

#(b) Each bird's tarsus length has been measured on the first time it   
was caught. Each bird was measured by one out of two observers, but one   
observer consistently gave higher measures than the other. Create a   
dataset ready to be analyzed: it should in addition to the previous one   
(individual identities + encounter history), also contain the covariates   
observer and tarsus length.   
  
 #Making a vector for Observers  
Observer <- sample(c("Observer1","Observer2"), size = 200, replace = TRUE)  
  
 #Making a data frame with the Observer column included  
RingedBirdsData2 <- data.frame(Ring, history, Observer)  
  
 #Filling a column for torso length with values which are higher for  
 Observer 2  
 #Two ways to do this. Like this:   
  
RingedBirdsData2$Torso <- ifelse(RingedBirdsData2$Observer == "Observer1",  
 sample(seq(20, 25, 0.01)),  
 sample(seq(23, 28, 0.01)))  
  
#.. or like this:  
RingedBirdsData2$Torso <- ifelse(RingedBirdsData2$Observer == "Observer1",  
 runif(200, min = 20, max = 25),  
 runif(200, min = 22, max = 27))  
  
head(RingedBirdsData2)

## Ring history Observer Torso  
## 1 1001V 11111 Observer1 23.52069  
## 2 1002V 01011 Observer1 23.40119  
## 3 1003V 10111 Observer1 21.22074  
## 4 1004V 11111 Observer2 23.80930  
## 5 1005V 01111 Observer2 24.02672  
## 6 1006V 01010 Observer2 23.81567

#IN CLASS:  
ID <- rbinom(200, 1, 0.5)  
Birdsdata <- cbind(Birdsdata, ID)  
  
tarsus <- c()  
for (i in 1:nbirds) {  
 tarsus[i]<-ifelse(Birdsdata$ID[i]=="1", rnorm(1,20,2), rnorm(1, 18, 2))  
   
}  
  
Birdsdata <- cbind(Birdsdata, tarsus)  
head(Birdsdata)

## rings history ID tarsus  
## 1 1001V 11111 1 18.37678  
## 2 1002V 01011 1 20.09403  
## 3 1003V 10111 1 19.54369  
## 4 1004V 11111 0 15.60312  
## 5 1005V 01111 1 17.62132  
## 6 1006V 01010 1 18.34348

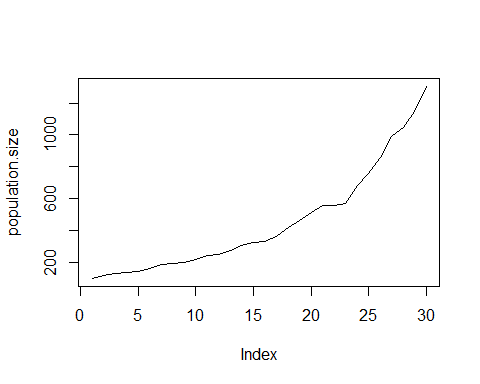
#(c) The calipers used to measure the birds' tarsus, which are very   
sensitive to humidity, did not work on ten occasions because of heavy   
rainfalls and the birds were not measured on subsequent breeding seasons. Change the tarsus length measurement to NA for ten randomly chosen birds.  
  
 #How I did it (very complicated.., just use the indexing as we did in   
 class). %in% ia a slightly different version of ==   
RingedBirdsData2$Torso <- ifelse(RingedBirdsData2$Torso %in% sample(RingedBirdsData2$Torso,10, replace = TRUE), NA, RingedBirdsData2$Torso)  
  
sum(is.na(RingedBirdsData2$Torso))

## [1] 10

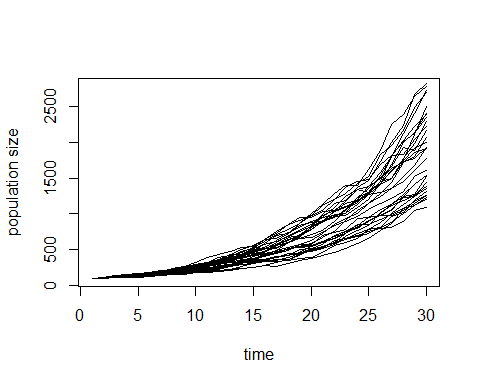
#IN CLASS: (MUCH BETTER WAY)  
  
Random10 <- sample(200,10)  
Birdsdata$tarsus[Random10] <- NA  
head(Birdsdata)

## rings history ID tarsus  
## 1 1001V 11111 1 18.37678  
## 2 1002V 01011 1 20.09403  
## 3 1003V 10111 1 19.54369  
## 4 1004V 11111 0 15.60312  
## 5 1005V 01111 1 NA  
## 6 1006V 01010 1 18.34348

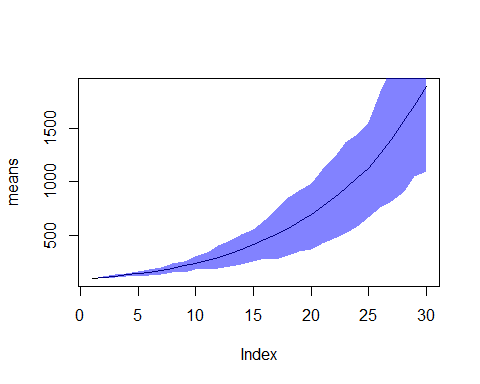
#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VI.4: Simulate population dynamics  
  
#The expected size N of a population which is growing stochastically (the growth rate changes at #every time step) can be predicted at any time t   
(t = 0, 1, 2, ... tmax) given its initial population #size (N0) using the following formula:  
 #Nt+1=Nt\*exp(rt) where rt follows a normal distribution with mean r and standard deviation sdr  
  
#i) Simulate the growth of a population over 30 time steps for N0=100,   
mean r= 0.1 and sdr=0.05 and plot it. (TIP: consider carefully your first value of t in this simulation).  
  
population.size <- numeric(length = 30)  
population.size[1] <- 100 #sets initial value  
  
for (i in 2:30) { # starting from to beacause 1 is the initial value  
 rt <- rnorm(n=1, mean =0.1, sd=0.05) #rt as normal distibution   
 population.size[i] <- population.size[i-1]\*exp(rt) #fills vectors   
 according to given  
 formula  
   
}  
   
plot(population.size, type = "l")



#ii) Using a for loop, repeat the operation 30 times and plot each   
growth curve on the same graph. You should store the simulated data at   
each iteration.  
  
pop.size2 <- matrix(NA, 30, 30)   
  
for (j in 1:30) {  
 pop.size2[1,j] <- 100 #initial value for evey columns (for evey repeat)  
   
 #inner loop  
 for (i in 2:30) { #again starting from 2  
 rt <- rnorm(n=1, mean = 0.1, sd = 0.05)  
 pop.size2[i,j] <- pop.size2[i-1,j]\*exp(rt)  
   
 }  
   
}  
  
plot(pop.size2[,1], type = "l", xlab = "time", ylab = "population size")  
  
for (j in 2:30) {  
 lines(pop.size2[,j], type = "l") #adds lines of next 29 columns to   
 exsistent plot  
   
}



#iii) Plot the average population growth and its 95%CI envelope as you  
 did in Part V.  
  
means <- apply(pop.size2,1,mean)  
quantiles <- apply(pop.size2,1,quantile, probs = c(0.0025, 0.975))  
  
plot(means, type = "l")  
  
 #envelope:  
x <- c(1:30, 30:1)  
y <- c(quantiles[1,1:30], quantiles[2,30:1])  
  
mycol <- rgb(0,0, 255, max = 255, alpha = 125)  
polygon(x,y, col=mycol, border = NA)



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VI.5: Power analysis, function and loop  
#You are planning an experiment, in which you want to test whether product "substance X" has an effect on the growth of saplings. You know the growth of saplings is on average 100 (arbitrary units), with a standard deviation of 5. You expect that "substance X"will lead to a 10% increase in growth, without changing the standard deviation. You have at your disposal limited facilities and want to write a simulation exploring how many saplings you should include in the experiment. The basic test you want to do is a   
t-test comparing control (untreated) saplings to treated ("substance   
X"treated) saplings, and the number of saplings (N) are of the same size   
in both these groups.  
  
#a) Create a data.frame with random numbers of growth for 10 untreated and 10 treated saplings  
  
untreated <- rnorm(10, mean = 100, sd = 5)  
treated <- rnorm(10, mean = (100\*1.1), sd = 5)  
  
saplings <- data.frame(untreated, treated)  
  
  
#b) Run a t-test on the data you created using t.test()  
  
t.test(saplings$treated, saplings$untreated, mu = 0.1, paired = TRUE)

##   
## Paired t-test  
##   
## data: saplings$treated and saplings$untreated  
## t = 4.3714, df = 9, p-value = 0.001793  
## alternative hypothesis: true difference in means is not equal to 0.1  
## 95 percent confidence interval:  
## 7.567554 23.585478  
## sample estimates:  
## mean of the differences   
## 15.57652

#c) Create a function which does a) and b) in one run for any expected %   
of increase (inc) and for any sample size in both groups (n). The function should return the probability of the growth of control and treated saplings being equal which is given by the p-value of the t-test.  
#The function should start like:  
  
My\_fun <- function(n, inc){  
 saplingsFun <- data.frame(untreated = rnorm(n, mean = 100, sd = 5),  
 treated = rnorm(n, mean = (100 + inc), sd = 5))  
 Ttest <- t.test(saplingsFun$untreated, saplingsFun$treated, paired = T)  
 return(Ttest$p.value)  
}  
   
My\_fun(20,5)

## [1] 0.006322212

#d) Write a script that runs n.sim number of simulations of your growth   
experiment on N saplings and returns the vectors of p-values for each   
round of the simulation.  
  
My\_fun\_randomized <- function(n.sim, n, inc = 10){  
 sapply(1:n.sim, function(x) My\_fun(n,inc))  
   
}  
   
 #with for loop:  
  
n.sim <- 100  
N <- 50  
incre <-10  
p.values<-c()  
  
for (i in 1:n.sim) {  
 p.values[i] <- My\_fun(N, incre)  
   
}  
  
head(p.values)

## [1] 2.560113e-12 1.572445e-10 1.801590e-14 3.208419e-15 1.434023e-13  
## [6] 1.817380e-15

#e) Can you apply this program to inform you about how many saplings you  
should have in your experiment?  
  
n.sim <- 1000  
 # -> use it for power calculations  
 #plot it with an array of the p-values

# Part VII

In this part, we look at how R can be customized to become an engine for specialized operations to fit your need as an analyst.

## Chapter 18. Discover R packages and access public information

## 18.1 What is a package?

Packages are collections of R functions, data, and compiled code in a well-defined format. The directory where packages are stored is called the library. R comes with a standard set of packages. Others are available for download and installation. Once installed, they have to be loaded into the session to be used. There are countless R packages available that are more or less useful or easy to use depending on what you would like to do as well as there are countless R packages that are just silly (check the “CatterPlots” package below)! For regular updates about R packages and tutorials, check out the [R-Bloggers website](https://www.r-bloggers.com/) (they also have a Facebook page). Packages allow you to modify R into whatever you want to do. This can be anything from sending Email and tweets, to accessing public data, mounting data-exploration apps on the web, to running most up-to-date statistical analyses.

## 18.2 How to install and load packages

Let’s assume you want to load ggplot2. For most packages, the basic command to install is:

> install.packages("ggplot2")

To then load this package, you have two possibilities:

> library(ggplot2)

> require(ggplot2)

They are both equivalent but require() is designed for use inside other functions. You can also install packages using drop-down control in R Studio.

When you start to install, R will then ask you to select a CRAN repository. Double-click on one of them (the one closest to your location for instance) and the package will be installed. Some packages are not available from CRAN repositories but can be installed from GitHub using the install\_github function from the devtools package. This means you first need to install the package devtools.

> install.packages("devtools")

> library(devtools)

> install\_github("Gibbsdavidl/CatterPlots")

Finally, some packages can be installed from zip files. This can be the case for packages that allow R to run other softwares that are licensed. For example, to install the licensed software Asreml 3.0 (VSN International)

> install.packages("asreml\_3.0.zip", repos = NULL, type = "win.binary")

## 18.3 Setting your repository

Every time you install an R package, the default is to ask you which CRAN repository R should use. To set the repository and avoid having to specify this at every package install, simply:

* create a file .Rprofile in your home area.
* Add the following piece of code to it:

cat(".Rprofile: Setting UK repository")  
r = getOption("repos") # hard code the UK repo for CRAN  
r["CRAN"] = "<http://cran.uk.r-project.org>"  
options(repos = r)  
rm(r)

## 18.4 Installing packages without root access

First, you need to designate a directory where you will store the downloaded packages. On my machine, I use the directory /data/Rpackages/ After creating a package directory, to install a package, use the command:

> install.packages("ggplot2", lib="/data/Rpackages/")  
> library(ggplot2, lib.loc="/data/Rpackages/")

It’s a bit of a pain having to type /data/Rpackages/ all the time. To avoid this burden, we create a file .Renviron in our home area, and add the line R\_LIBS=/data/Rpackages/ to it. This means that whenever you start R, the directory /data/Rpackages/ is added to the list of places to look for R packages and so:

> install.packages("ggplot2")  
> library(ggplot2)

## 18.5. Updating and removing

R and all its packages are continuously developing. When your R version is getting outdated, R will remind you when loading a recently updated package by warning that your R version is not as recent as the R version under which the package was built. For example,

>library(sommer)

Warning message:

package ‘sommer’ was built under R version 3.4.0

For outdated packages, there is no warning. You can update your packages

> update.packages(ask=FALSE, checkBuilt = TRUE)

Since it is R, there is of course package to help you automate this process. Package installr is useful for an automated approach to updating both your R and all packages to the latest version.

If you find you do not need a package anymore, you can remove it

> remove.packages(“ggplot2”)

## 18.6. Exercises belonging to part VII

VII.1

**Exercise 6.1:** Get data andPlot using a package

Go to Moodle for this course, and open the DataFile file “MyData.csv” we using during the start of this course in Excel and save it as an Excel worksheet (i.e. xls or xlsx file) in your working directory

Use package XLConnect to read in this Excel worksheet into R.

Using this data, produce a plot with two panels (a) and (b) with each panel a box-and-whiskers plot of the mass and the length for blue and red individuals respectively. You produce these plots using the standard packages installed in R (i.e. not installing anything. In every plot, boxes for blue and red individuals should be filled in blue and red. An example of the box plot:



**VII.2:** Use RgoogleMaps  
**(NOTE! We did not do this one for the course, we were given a replacement for this. It is included below after VII.5!)**

You can use R to make reasonable nice-looking maps plotting e.g. your study area with some details, but without getting into details of using R as a GIS. One of the easiest approaches is based on Google Maps, the package RgoogleMaps allows you to extract maps from google so you can plot on it, looking like the maps you typically get in your browser. Real basic info you can find here:

<https://pakillo.github.io/R-GIS-tutorial/#rgooglemaps>

The file “coordinates.txt” is a tab-delimited text file which contains the coordinates of 60 points in your study area.

1. Figure out how to produce the map below using **ONLY** RgoogleMaps (the internet is full of various options). The map plots the 60 coordinates using standard R plotting controls for points, see the map below.

[TIP: Carefully reading the help files and studying the overview of functions in RgoogleMaps gets you further than getting lost on the internet]

You can play a bit around and

1. produce different map backgrounds, e.g. satellite
2. produce a map with the number of the cluster (1:60, column in “coordinates.txt”) plotted at next to the symbol showing the location.

**VII.3:** Understanding XML

Public data comes in different forms. For example, XML. Check out this blog:

<http://www.r-bloggers.com/r-and-the-web-for-beginners-part-ii-xml-in-r/>

a. Implement the exercise described in this blog, and learn also from other sources about the logic of XML as a data format so you understand and can explain what is going on. Note that you have to download and install the XML package.

b. Produce a plot showing the frequency distribution of the plants’ light conditions for growing.

**VII.4:** Extract weather data

I wrote a simple function called “get\_met\_data.R”, which allows you to download public data (in XML format) from the Finnish Meteorological Institute (FMI), based on their Open data agreement. See <http://en.ilmatieteenlaitos.fi/open-data-manual> and related pages. You find the function from the following URL <https://github.com/JonBrommer/Meteo-data>

(i) Study the function and details. Get your API key which allows you to download climatic data from FMI. The data is in XML format. Get the function to work for you for the following url

url.kajaani.85=[http://data.fmi.fi/fmi-apikey/**INSERT-YOUR-KEY-HERE**/wfs?request=getFeature&storedquery\_id=fmi::observations::weather::daily::timevaluepair& fmisid=101725&starttime=1985-01-01T12:00:00Z&endtime=1985-12-31T12:00:00Z&parameters=tday&](http://data.fmi.fi/fmi-apikey/INSERT-YOUR-KEY-HERE/wfs?request=getFeature&storedquery_id=fmi::observations::weather::daily::timevaluepair&%20fmisid=101725&starttime=1985-01-01T12:00:00Z&endtime=1985-12-31T12:00:00Z&parameters=tday&)

url.kajaani.85="http://data.fmi.fi/fmi-apikey/de2a72a1-d237-40f1-a5e3-37dd68ee7b60/wfs?request=getFeature&storedquery\_id=fmi::observations::weather::daily::timevaluepair&fmisid=101725&starttime=1985-01-01T12:00:00Z&endtime=1985-12-31T12:00:00Z&parameters=tday"

which is a request to download data on daily temperatures (“parameters=tday&”) from station in Kajaani (based on the FMI station ID, “fmisid=101725”), for the time period starting 1 January 1985 and ending 31.12.1985 (starttime and end time are given in the format year-month-day).

Insert your API key at the correct place and copy-paste the URL in your web browser. It produces the data file (in XML). You can check out the structure. Note that this XML is more complicated than the previous example.

1. Climate warming started in the end of the 1970s. Warming is often asymmetric. Write a routine that downloads the April and May daily temperatures from five weather stations for the period 1974-2018, covering as much of the latitudinal range we have in Finland (south to north).
2. Plot the trend in warming, arbitrarily defined as the deviation of the monthly temperature to the mean temperature of 1974-1984, against year for each station for the months April and May separately in one multi-panel figure. Plot the positive deviations (warmer than the mean) in red and the negative deviations (colder than the mean) in blue.

**VII.5** Image analysis

When you take a picture, you make an array. Each pixel in your picture has an intensity value for Red, Green or Blue (RGB). The layers for RGB for an array of values. This means that in R if you understand how to manipulate matrices and arrays, you can manipulate pictures.

Matloff’s book home page offers chapter 3 for free download. Go to the URL: <http://nostarch.com/artofr.htm> and download this chapter. Study section 3.2.3 on image manipulation.

(i) Repeat what Matloff does on pages 64 and 65, but using a different image file. Here is the start of your script:

#playing with an image as an example of matrix indexing in R

#####################################

#section 1: organising things

#load the pixmap package into R

library(pixmap)

#get the image we really love...

R.picture <- read.pnm(system.file("pictures/logo.ppm", package="pixmap")[1])

#look at it

plot(R.picture)

#we want it in black-and-white to make life simpler

R.picture.grey<-as(R.picture, "pixmapGrey") #this is a generic way to coerce

#look at it once more, now b/w

plot(R.picture.grey)

#check out its structure

str(R.picture.grey)

#####################################################

#section 2: gettin ready to punch a hole in R

#we use R's locator function to find the coordinates of two corners in the image

coord<-locator(2) #this allows you to select 2 points by clicking on the image

coord #this provides you with the coordinates

## you take it from here

Add to the code the required operation to punch a hole in your image. You use the locator() function (implemented in this script) to select coordinates that are the opposite corners of a rectangular ‘hole’, as Matloff does in his image in the book chapter.

Note that pixmap objects use the ‘@’ sign instead of the ‘$’ sign. Technically, pixmap is based on S4 class objects whereas we have previously dealt with S3 class objects. The difference is that the rules for S4 are a bit more restrictive in terms of manipulation, but for the present exercise you just use @ instead of $ and it should be fine

**Replacement for Exercise VII.2:** Make word clouds with R

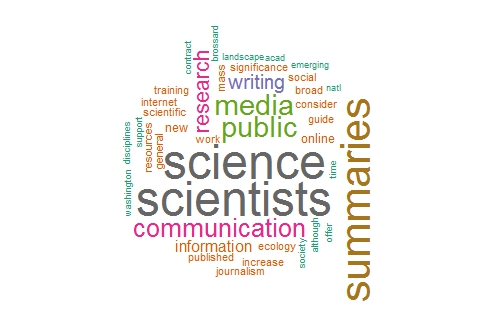
A word cloud is “an image composed of words used in a particular text or subject, in which the size of each word indicates its frequency or importance” (Oxford dictionary). In this exercise, you will create a word cloud based on a pdf file ("PNAS-2015-Kuehne-science communication.pdf") and try to make it look as good as possible. Note that sometimes the same word can appear in different forms (e.g. “summaries” and “summary”) or some words are always associated (e.g. “lay summary”).

Some useful tutorials can be found at these addresses:

<https://www.r-bloggers.com/building-wordclouds-in-r/>

<http://www.sthda.com/english/wiki/text-mining-and-word-cloud-fundamentals-in-r-5-simple-steps-you-should-know>

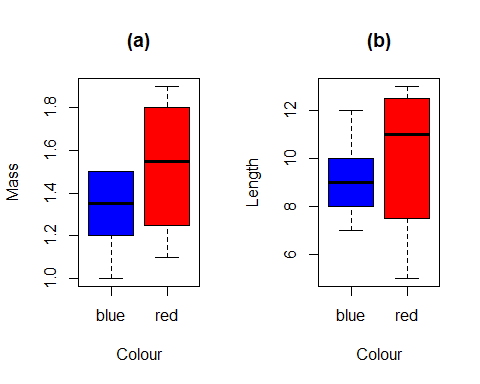
<http://riccardo-hertel.com/wp/index.php/2015/09/03/using-r-to-create-a-word-cloud-from-a-pdf-document/>

Below is an example of a word cloud based on this pdf:

## MY CODE: Part 7:

#VII.1  
#Exercise 6.1: Get data and Plot using a package  
  
#Go to Moodle for this course, and open the DataFile file "MyData.csv" we using during the start of this course in Excel and save it as an Excel   
worksheet (i.e. xls or xlsx file) in your working directory  
  
#Use package XLConnect to read in this Excel worksheet into R.   
  
setwd("E:/Introduction to R")  
  
*install.packages("XLConnect")*  
library(XLConnect)

MyData <- loadWorkbook("MyData.xlsx")  
MyData <- readWorksheet(MyData, "myData", header = TRUE)   
  
#Using this data, produce a plot with two panels (a) and (b) with each   
panel a box-and-whiskers plot of the mass and the length for blue and red individuals respectively. You produce these plots using the standard   
packages installed in R (i.e. not installing anything. In every plot,   
boxes for blue and red individuals should be filled in blue and red. An   
example of the box plot:  
  
 #getting the double panel view:  
par(mfrow = c(1,2))  
  
boxplot(mass ~ colour, data = MyData,  
 col = c("blue", "red"), main = "(a)",  
 xlab = "Colour", ylab = "Mass")  
  
boxplot(length ~ colour, data = MyData,  
 col = c("blue", "red"), main = "(b)",  
 xlab = "Colour", ylab = "Length")



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#Exercise VII.2 (replacement for the original exercise)  
#Make word clouds with R  
  
#In this exercise, you will create a word cloud based on a pdf file   
("PNAS-2015-Kuehne-science communication.pdf") and try to make it look as good as possible. Note that sometimes the same word can appear in   
different forms (e.g. "summaries" and "summary" or some words are always  
 associated (e.g. "lay summary").  
  
 #instructions direcly from one of the blog posts:  
  
*install.packages*("tm")# for text mining  
*install.packages*("SnowballC")# for text stemming  
*install.packages*("wordcloud")# word-cloud generator  
*install.packages*("RColorBrewer")# color palettes  
*install.packages*("pdftools")  
# Load  
library("tm")

## Loading required package: NLP

library("SnowballC")  
library("wordcloud")

## Loading required package: RColorBrewer

library("RColorBrewer")  
library(pdftools)  
  
my\_pdf <- readPDF(control=list(text="-layout"))(elem=list(uri="PNAS-2015-Kuehne-science communication.pdf"), language="en")  
  
text\_raw <- my\_pdf$content  
  
text\_corpus <- Corpus(VectorSource(text\_raw))  
  
corpus\_clean <- tm\_map(text\_corpus, stripWhitespace)

corpus\_clean <- tm\_map(corpus\_clean, removeNumbers)

corpus\_clean <- tm\_map(corpus\_clean, content\_transformer(tolower))

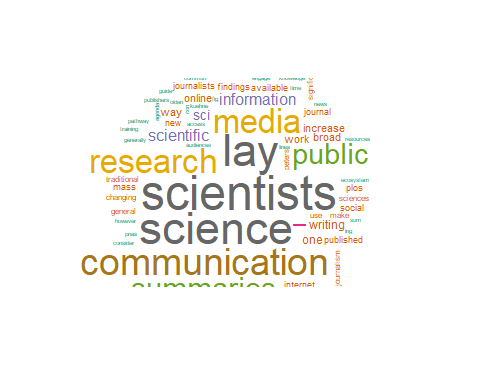
print(stopwords("en"))

## [1] "i" "me" "my" "myself" "we"   
## [6] "our" "ours" "ourselves" "you" "your"   
## [11] "yours" "yourself" "yourselves" "he" "him"   
## [16] "his" "himself" "she" "her" "hers"   
## [21] "herself" "it" "its" "itself" "they"   
##.... I removed the rest of them

corpus\_clean <- tm\_map(corpus\_clean, removeWords, stopwords("english"))

#corpus\_clean <- tm\_map(corpus\_clean, removeWords, my\_stopwords)  
corpus\_clean <- tm\_map(corpus\_clean, removePunctuation)

par(mfrow = c(1,1))  
  
wordcloud(corpus\_clean, max.words=Inf, random.order=FALSE,  
 scale= c(3, 0.1), colors=brewer.pal(8,"Dark2"))



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VII.3: Understanding XML  
  
#a. Implement the exercise described in this blog, and learn also from   
other sources about the logic of XML as a data format so you understand   
and can explain what is going on. Note that you have to download and   
install the XML package.   
  
*install.packages("XML")*  
library(XML)  
  
xml.url <- "http://www.w3schools.com/xml/plant\_catalog.xml"  
  
 #This didn't work:   
#xmlfile <- xmlTreeParse(xml.url)  
  
 #Did this instead  
*install.packages*("httr")  
library(httr)

xmlfile <-xmlTreeParse(rawToChar(GET(xml.url)$content))  
  
# the xml file is now saved as an object you can easily work with in R:  
class(xmlfile)

## [1] "XMLDocument" "XMLAbstractDocument"

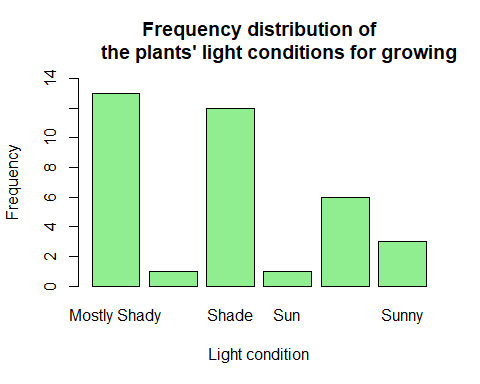
# Use the xmlRoot-function to access the top node  
xmltop = xmlRoot(xmlfile)  
# have a look at the XML-code of the first subnodes:  
print(xmltop)[1:2]

## <CATALOG>  
## <PLANT>  
## <COMMON>Bloodroot</COMMON>  
## <BOTANICAL>Sanguinaria canadensis</BOTANICAL>  
## <ZONE>4</ZONE>  
## <LIGHT>Mostly Shady</LIGHT>  
## <PRICE>$2.44</PRICE>  
## <AVAILABILITY>031599</AVAILABILITY>  
## </PLANT>  
## <PLANT>  
## <COMMON>Columbine</COMMON>  
## <BOTANICAL>Aquilegia canadensis</BOTANICAL>  
## <ZONE>3</ZONE>  
## <LIGHT>Mostly Shady</LIGHT>  
## <PRICE>$9.37</PRICE>  
## <AVAILABILITY>030699</AVAILABILITY>  
## </PLANT>  
## <PLANT>  
## <COMMON>Marsh Marigold</COMMON>  
## <BOTANICAL>Caltha palustris</BOTANICAL>  
## <ZONE>4</ZONE>  
## <LIGHT>Mostly Sunny</LIGHT>  
## <PRICE>$6.81</PRICE>  
## <AVAILABILITY>051799</AVAILABILITY>  
#... I removed the rest

# To extract the XML-values from the document, use xmlSApply:  
plantcat <- xmlSApply(xmltop, function(x) xmlSApply(x, xmlValue))  
  
# Finally, get the data in a data-frame and have a look at the first rows and columns  
plantcat\_df <- data.frame(t(plantcat),row.names=NULL)  
plantcat\_df[1:5,1:4]

## COMMON BOTANICAL ZONE LIGHT  
## 1 Bloodroot Sanguinaria canadensis 4 Mostly Shady  
## 2 Columbine Aquilegia canadensis 3 Mostly Shady  
## 3 Marsh Marigold Caltha palustris 4 Mostly Sunny  
## 4 Cowslip Caltha palustris 4 Mostly Shady  
## 5 Dutchman's-Breeches Dicentra cucullaria 3 Mostly Shady

#b. Produce a plot showing the frequency distribution of the plants' light conditions for growing.  
  
lightcondtions <- table(plantcat\_df$LIGHT)  
barplot(lightcondtions, xlab = "Light condition", ylab = "Frequency",  
 ylim = c(0,14), col = "lightgreen",  
 main = "Frequency distribution of  
 the plants' light conditions for growing")



#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VII.4: Extract weather data  
  
#(i) Study the function and details. Get your API key which allows you to download climatic data from FMI. The data is in XML format. Get the   
function to work for you for the following url  
  
get.met.data<-function(url.address) {  
 require(XML)  
 r<-xmlRoot(xmlTreeParse(url.address))  
 #below goes directly to "results" node under the root  
 metcat<-xmlSApply(r[[1]][[1]][[7]][[1]],  
 function(x) xmlSApply(x,xmlValue))  
 #splits the vector with "Z" and "T" which is the last and first  
 character in the time stamp   
 metdat<-data.frame(matrix(  
 unlist(strsplit(unlist(strsplit(metcat,"Z")),"T"))  
 ,xmlSize(r[[1]][[1]][[7]][[1]]),  
 ,byrow=T))  
 year.month.day<-data.frame(matrix(  
 unlist(strsplit(as.character(metdat[,1]),"-"))  
 ,xmlSize(r[[1]][[1]][[7]][[1]]),  
 ,byrow=T))  
 met.data.set<-  
 cbind(year.month.day,metdat[,3])  
 names(met.data.set)<-c("year","month","day","met.data")  
 #convert it to numeric  
 met.data.set$year<-as.numeric(as.vector(met.data.set$year))  
 met.data.set$month<-as.numeric(as.vector(met.data.set$month))  
 met.data.set$day<-as.numeric(as.vector(met.data.set$day))  
 met.data.set$met.data<-as.numeric(as.vector(met.data.set$met.data))  
 return(met.data.set)  
} #end of function  
# to load using dget, a white line at the end is needed  
  
 #my new api-key: de2a72a1-d237-40f1-a5e3-37dd68ee7b60  
 #MY WORKING URL:  
url.kajaani.85="http://data.fmi.fi/fmi-apikey/de2a72a1-d237-40f1-a5e3-37dd68ee7b60/wfs?request=getFeature&storedquery\_id=fmi::observations::weather::daily::timevaluepair&fmisid=101725&starttime=1985-01-01T12:00:00Z&endtime=1985-12-31T12:00:00Z&parameters=tday"  
  
 #does it work? – yes it does!  
head(get.met.data(url.kajaani.85))

## year month day met.data  
## 1 1985 1 2 -14.2  
## 2 1985 1 3 -27.4  
## 3 1985 1 4 -30.9  
## 4 1985 1 5 -37.6  
## 5 1985 1 6 -39.7  
## 6 1985 1 7 -40.3

tail(get.met.data(url.kajaani.85))

## year month day met.data  
## 359 1985 12 26 -10.0  
## 360 1985 12 27 -18.0  
## 361 1985 12 28 -20.5  
## 362 1985 12 29 -13.0  
## 363 1985 12 30 -16.0  
## 364 1985 12 31 -16.7

#(ii) Climate warming started in the end of the 1970s. Warming is often asymmetric. Write a routine that downloads the April and May daily   
temperatures from five weather stations for the period 1974-2018, covering as much of the latitudinal range we have in Finland (south to north).   
  
 #NOTE: my own api-key refused to give me all five stations in this for   
loop (only gave me data from three of the stations??), so I use another   
one here than in the exercise above (i)..  
  
require(XML)  
id.vec<-c(100932,101315,101776,101952,102035)  
start.time.string=c("-04-01T00:00:00Z")  
end.time.string=c("-05-31T23:50:00Z")  
met.data<-data.frame()  
  
for (i in 1:length(id.vec)) {  
 for (y in 1974:2018) {  
 # create URL address  
   
 start.time.string.new <- paste(as.character(y), start.time.string, sep = "")  
   
 end.time.string.new <- paste(as.character(y), end.time.string, sep = "")  
   
 url.addr<-  
 paste("http://data.fmi.fi/fmi-apikey/168c9e89-8d09-4d56-be04-8d93d8a759f7/wfs?request=getFeature&storedquery\_id=fmi::observations::weather::daily::timevaluepair&fmisid="  
 , id.vec[i]  
 , "&starttime="  
 , start.time.string.new  
 , "&endtime="  
 , end.time.string.new  
 ,"&parameters=tday"  
 , sep="")  
 #adds the temperature data to the list  
 mdat<-get.met.data(url.addr) #data  
   
 #first rbind the data of the institute to the dataframe and then add a   
 new column with the FMISID  
 met.data<-rbind(met.data, cbind(id.vec[i],mdat))  
 #met.data<-cbind(FMISID= c(rep(id.vec[i],length(mdat$year))), met.data)  
   
 } #for (y  
} #for (id  
  
#viewing the result:  
head(met.data)

## id.vec[i] year month day met.data  
## 1 100932 1974 4 1 2.4  
## 2 100932 1974 4 2 2.4  
## 3 100932 1974 4 3 2.8  
## 4 100932 1974 4 4 2.9  
## 5 100932 1974 4 5 3.5  
## 6 100932 1974 4 6 4.7

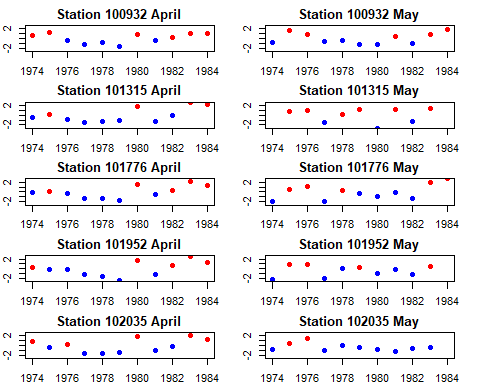
tail(met.data)

## id.vec[i] year month day met.data  
## 13719 102035 2018 5 26 8.4  
## 13720 102035 2018 5 27 7.0  
## 13721 102035 2018 5 28 13.9  
## 13722 102035 2018 5 29 9.1  
## 13723 102035 2018 5 30 5.9  
## 13724 102035 2018 5 31 8.9

names(met.data)[1] <- "ID"  
names(met.data)[5] <- "temp"  
  
 #checking that all the stations are in there:  
aggregate(temp~ID,met.data, length)

## ID temp  
## 1 100932 2745  
## 2 101315 2744  
## 3 101776 2745  
## 4 101952 2745  
## 5 102035 2745

#(iii) Plot the trend in warming, arbitrarily defined as the deviation of the monthly temperature to the mean temperature of 1974-1984, against year for each station for the months April and May separately in one multi-  
panel figure. Plot the positive deviations (warmer than the mean) in red   
and the negative deviations (colder than the mean) in blue.  
  
 #subseting the years we want:  
Tempsforyears<- subset(met.data, year>= 1974 & year<= 1984)  
  
 #getting the mean temperatures for each month and year:  
MeanMonth<- aggregate(temp~month+year+ID, data= Tempsforyears, mean)  
  
 #getting the mean temperature for each station:  
Meanfor10yearsperID<- aggregate(temp~month+ID, data=MeanMonth, mean)  
  
 #merging the data to be used in the for loop  
TempData<- merge(MeanMonth, Meanfor10yearsperID, by=c("ID","month"))  
  
 #adding a new column for: the monthly mean minus the mean for 10 years  
TempData$deviation<- TempData$temp.x- TempData$temp.y  
  
  
#Now we plot using a for loop to make new data frames for each plot.  
  
 #Setting the available "slots" for plots:  
par(mfrow=c(5,2))  
  
 #Since there is a error message "figure marings too large" unless we   
 alter the margins, we need to reduce them. This means that it cuts out   
 the axis labels though. You could just use the parameters:   
 par(mfrow=c(1,2)), and then it would make five equally big set of plots   
 (1x2).. this was not an issue everyone had..  
par(mar = c(2, 2, 2, 2))  
  
  
for (i in 1:length(unique(TempData$ID))) {  
   
 #get data for each station  
 SubsetTempData<- subset(TempData, ID==unique(TempData$ID)[i])  
 #get data for both months separately  
 April<- subset(SubsetTempData, month== "4")  
 May<-subset(SubsetTempData, month== "5")  
   
 plot(April$deviation~April$year,  
 main= paste("Station", SubsetTempData$ID[1],  
 "April", sep= " "),  
 xlab="Year",  
 ylab="Deviation from mean temperature, °C",  
 pch= 19, ylim= c(-2.5,2.5),  
 col=ifelse(April$deviation >= 0, "red", "blue"))  
   
 plot(May$deviation~May$year,  
 main = paste("Station" ,SubsetTempData$ID[1],  
 "May", sep= " "),  
 xlab="Year", ylab="Deviation from mean temperature",  
 pch= 19, ylim= c(-2.5,2.5),  
 col=ifelse(May$deviation >= 0, "red", "blue"))  
}



(The plots look much nicer in real life, *knitr* only pastes the direct output without “zooming them”)

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
#VII.5 Image analysis  
  
#(i) Repeat what Matloff does on pages 64 and 65, but using a different   
image file. Here is the start of your script:  
  
par(mfrow = c(1,1))  
  
 #section 1: organising things   
 #load the pixmap package into R  
#install.packages("pixmap")  
library(pixmap)  
  
 #get the image we really love...  
R.picture <- read.pnm(system.file("pictures/logo.ppm",  
 package="pixmap")[1])

#look at it  
plot(R.picture)



#we want it in black-and-white to make life simpler  
R.picture.grey<-as(R.picture, "pixmapGrey") #this is a generic way to  
 coerce  
  
 #look at it once more, now b/w  
plot(R.picture.grey)

  
(I pasted this here since *knitr* didn’t do it for some reason?)  
  
 #check out its structure  
str(R.picture.grey)

## Formal class 'pixmapGrey' [package "pixmap"] with 6 slots  
## ..@ grey : num [1:77, 1:101] 1 1 0.999 0.999 1 ...  
## ..@ channels: chr "grey"  
## ..@ size : int [1:2] 77 101  
## ..@ cellres : num [1:2] 1 1  
## ..@ bbox : num [1:4] 0 0 101 77  
## ..@ bbcent : logi FALSE

#section 2: gettin ready to punch a hole in R  
#we use R's locator function to find the coordinates of two corners in the image  
  
coord<-locator(2) #this allows you to select 2 points by clicking on the   
 image



*\*click\*, \*click\**

#row numbers in pixmap objects increase from the top of the picture to  
 the bottom, the opposite of the numbering used by locator() --> the   
 columns are correct, but we need to modify the rows (y-values).   
 Otherwise it wil puch the hole in the wring place.  
  
nrows <- nrow(R.picture.grey@grey)  
R.picture.grey@grey[(nrows-coord$y[1]):(nrows-coord$y[2]),  
 coord$x[1]:coord$x[2]] <-1  
plot(R.picture.grey)



(Now there should be a hole in the image based on where you clicked, *knitr* skips the locator bit of code – and therefore we now have no coordinates for the hole. This is what it should look like (pasted separately from R:s output):



# Getting further with R

An ending note for this course.

The main idea of this course is to get you started in R. The entire workflow including data import, data extraction from public sources, data manipulation, and merging of different datasets producing the information you need is done in R. This part of the workflow has been one focal point of this course. After you have prepared the data, you can analyse the data in R as well (which we have not covered in detail). Next steps concern the extracting of key parameters from the analysis (thereby building your results table) and plotting of the results in R as introduced in this course.

There are many books on using R as well as blogs and online courses and these are a great asset. In the end, however, the best way to develop your skills is by getting to work in R.