you experienced?

A small introduction to R.

Patrick Meirmans,

IBED, Universiteit van Amsterdam

p.g.meirmans@uva.nl

1. Using the interface

The console

When you start R, you will be presented with the R console. This is where you will do a lot of your work with R. To work with R, you will have to enter equations and commands into the console. All output will also be written to the console. For example, you can type simple equation after the "> " prompt (but don't type the ">" itself!):

```
> 1+1
```

If you now hit "Enter", you will see the output:

```
[1] 2
```

The `[1]` indicates that this line begins with the first element of the output. A lot of different mathematical operators and functions are possible, such as:

```
> sqrt(16)
> log(22)
> exp(4)
> pi
> sin(pi)
> (4 * 3^2) / (3 - sqrt(2))
```

Note that in general, spaces don't matter as long as they are not in the middle of a name, so `sqrt(16)` is the same as `sqrt( 16 )`, or `sqrt (16)`. You can use this to improve the readability of your code. For example the last equation above is better readable with spaces than without spaces: `(4*3^2)/(3-sqrt(2))`.

R stores the history of entries you made into the console. If you want to recall something you typed earlier, you can hit the *up-arrow* key on your keyboard to see your previous entries (and the *down-arrow* to go forward again in your history). When you have selected a previous entry, just hit "*Enter*" to run it again.

Variables

When you type an equation into the console, like you did above, the results are simply printed to the console and that's it. Mostly, you will actually want to do more with the results of your calculations. Therefore, you can store the result into a variable:

```
> a = 2 + 1
```

You will notice that when you type this in, no output is printed to the console. If you want to see the value of a, you simply have to type:

```
> a
```

The variable that you defined can now further be used in equations:

```
> b = a + 1
> c = sqrt(a^2 + b^2)
```

Another way of appointing variables is to use the <- operator, this is the "official" way of doing it that is used in all books on R. There seems to be a small number of cases where <- works better than =, but I have never encountered any. Since I find = more intuitive, I will use this, but be aware that most of the R-community disagrees.

The names of variables can be anything that starts with a letter, and can also include many other characters as "." or "_", and numbers (but it cannot start with numbers). The names are case-sensitive, so variable `Patrick` is something else than variable `patrick`. The range of allowed characters for variable names depends on the operating system and language, so it is best to be conservative and use standard Latin characters. Spaces are not allowed, if you want to make your variables names more descriptive (and please do), it is common practice to separate words in the name by dots:

```
> number.of.hours <- 24
```

You also have to be careful not to use names that are in use by R for functions, such as:

```
> exp = exp(42)
```

This often works fine, but not always, and cause problems that are hard to solve.

The workspace

All variables and functions that you define are stored internally in what R calls the "workspace". You can get a view of all variables by choosing "Show workspace" in the "Workspace" menu. Equivalently, you can type `ls()` into the console. If you entered all of the above, this will give the following output:

```
[1] "a"   "b"   "c"   "exp"   "number.of.hours"
```

Another way to get an overview of what's in your workspace in by selecting "Workspace browser" from the menu. As explained above, it is bad practice to use the name `exp` since it is also the name of a standard function. Therefore, it is better to remove this variable from the workspace:

```
> rm(exp)
```

Now verify that the variable has indeed been removed.

If you want to remove all variable in the workspace select "Clear Workspace", after which you will be asked for a confirmation. It is also possible to save workspaces and reopen them later. In fact, the program will ask you whether you want to save your workspace when you quit R. I never do, and actually switched this off in the preferences, but you might find this handy.

More often than cleaning your workspace, you might want to remove all the clutter in your console (for example if you made a lot of embarrassing mistakes). For this, select "Clear console" from the "Edit" menu. If you do this, all variables you defined will still exist in the workspace, you just don't see anymore where you defined them.

R documents

Besides entering commands line by line into the console, as we have been doing so far, it is also possible to first write a whole script in a text file and then execute that in R. To create such documents, the R interface contains a text editor. You can get a new empty document by choosing "New Document" from the "File" menu. If you do this, you get a new document in which you can define variables and type some equations and commands. To execute a portion of the script, select it and choose "Execute" from the

"Edit" menu (or the shortcut Apple-Enter on a Mac). For executing the whole script, simply select it all and then choose "Execute". What happens if you execute without making a selection?

R documents (with extension .R) can be saved and reopened like any normal text file, which makes them ideal to store your analyses. This way, you can base any future analyses on your old ones, without having to reinvent the wheel every time. In the current document, when a bit of code is preceded by "> ", you should type it directly into the console. Otherwise, type it into a text document and execute it.

R-Studio

As an alternative to scripting in with the R text editor, you may find the interpreter *R-Studio* useful. *R-Studio* is a so-called "wrapper" for the R console described above. It offers an intuitive way to make and execute *R*-code, but there are some other benefits.

Open R-studio, press *Ctrl+Shift+N* or *Command+Shift+N* and a new documunt opens in the upper-left corner, just above the console. Save this file (*Ctrl+S / Command+S*) as e.g. "Example.R" into your desired directory. You may also set this directory as a working dir (will be useful later, once you need load some data from files or save figures): *Session – Set Working Directory – To Source File Location.* This action sends the following code to the R-console (you can actually copy the code and paste it into your script so that you save yourself some clicking next time you run the script):

```
setwd("C:/workshop/Rbasics")
```

Now type some command into the script panel (in the upper-left corner), e.g.

```
1+1
```

and press *Ctrl+Enter / Command+Enter.* If all works well, the command has been sent to the console and shows the correct result.

For executing more commands (lines) at the same time, just mark all rows with mouse and then press *Ctrl+Enter / Command+Enter*.

```
1+1
2-3
exp(4)
```

You can annotate your script by placing a hashtag # at the start of the line. Try the following in the R script panel and then again execute all lines – what happened?

```
#1+1
2-3
exp(4)  # this calculates exp
```

You can quickly *comment* and *uncomment* lines by selecting them and typing *Ctrl+Shift+C. / Command+Shift+C.*

Packages

Even though the base package of R includes many useful functions, for specialised analyses you will quickly find that you need more. Luckily there are thousands of so-called packages available which can extend the capabilities of R by providing additional functions.

R comes with several default packages, but not all are loaded ("switched on"). The idea behind this is to maximise computing power. Compare *Microsoft Office* software, where all possible functions are always loaded, which can severely slow down your computer (especially if it is older). To see which *R* packages are currently available on your computer and to see which ones are loaded, you can choose "Package Manager" from the "Packages & Data" menu. You will see a list with all available packages, each with a switch button before its name; simply switch the button to load the package. If you have a certain R document that requires that a certain package is loaded before the code can run, it may be handier to actually include some code to load that package at the first line of the script. For example, to load the "vegan" package for vegetation analysis, you can type:

```
> library(vegan)
```

If the package is not on your compter, you will get a warning (as you may have experienced if you tried the above command). Packages can be added to your computer via the CRAN website; the Comprehensive R Archive Network. The R program has a direct interface to get packages from CRAN, simply choose "Package Installer" from the "Packages & Data" menu. In the window that pops up you can click the "Get List"

button to see all packages available in the repository. Now you can select a package and click the "Install Selected" button. It may be wise to first check the "install dependencies" button. This will make sure that if the selected package relies on other packages, these will also be downloaded if they are not on your computer.

R-Studio offers a neat interface to manage and install packages. Check it out and see if you understand how it works.

Help

Though there is very extensive onscreen Help for R, finding out how to do things using this Help is rather difficult. If you want to calculate an average, and you type “average” into the search box, you will not discover that you actually have to use the function called `mean`. Instead of helping you find the functions to do what you want, the Help-files consist of a documentation that explain how the available functions work. If you want to know how the `mean` function works, you can type `mean` into the Help-search-box (or type `?mean` in the console) and you will get a –rather tersely written– overview of what the function does. So if you know the name of the function, but can’t remember exactly how it works, use Help. If you want to find out how to do something in R, use Google.

2. Strings, numbers, & vectors

Strings

A string is programmers-speak for a series of characters, so what all other people would call "text". Strings can also be stored in variables:

```
> file.name = "data nature paper.txt"
```

In the console output, strings are immediately recognisable by the quotation marks. For example, the output of the `ls()` function you used above consists of a number of strings. At the moment we will leave it with this, but later you will see that strings have important tasks as names and for factors.

Numbers & vectors

In R, vectors are rows of concatenated numbers. Vectors are everywhere in R. In fact, they are so pervasive that single numbers are actually equivalent to vectors of length one. If you understand how to work with vectors, you know the most important thing there is to learn about R. The simplest way to create a vector is as follows:

```
> pH = c(4.3, 4.7, 4.2, 4.5)
```

The function name `c` is short for "concatenate". If you now check the value of the variable `pH`, you will see that the console returns:

```
[1] 4.3 4.7 4.2 4.5
```

Also here, variable names can be used to combine vectors. For example, this will replace the previous vector `pH` with a slightly longer one:

```
> pH = c(pH, 7.0, 6.7)
```

There are many ways to create vectors. Try to figure out what the following commands do:

```
> b = 1:10
> d = seq(0, 1, 0.1)
> e = rep(b, 5)
```

```
> f <- rep(b, each=5)
> g = runif(12, min=10, max=20)
```

Vector operations

Since R is a statistical framework, it contains many functions for calculating summary statistics on data, such as the mean, variance, etc. For many such functions, vectors are the main input:

```
> pH = c(4.3, 4.7, 4.2, 4.5, 7.0, 6.7, 7.2, 7.5)
> mean(pH)
> var(pH)
> sd(pH)
> length(pH)
> max(pH)
> sort(pH)
```

Of course, you can also make a vector out of these results. How do you make a vector containing the average, maximum and minimum pH?

In contrast to the functions above, many other functions are performed on an element-by-element basis:

```
> a = c(1, 2, 3, 4)
> sqrt(a)
```

Gives the output:

```
[1] 1.000000 1.414214 1.732051 2
```

When performing mathematical operations using a combination of vectors of the same length, they are also performed on an element-by-element basis:

```
> b = c(4, 5, 6, 7)
> a+b
```

Produces:

```
[1]  5  7  9 11
```

When two vectors of unequal length are used, the shorter one is repeated to match the length of the longer one.

```
> d = c(1,2)
> a*d
```

Gives the output

```
[1] 1 4 3 8
```

A single number is the same as a vector of length one, so therefore:

```
> 2*a
```

gives

```
[1] 2 4 6 8
```

What happens when the length of the longest vector is not an exact multiple of that of the smaller one?

Accessing vector elements

Often it is necessary to access individual elements of a vector. For this, the name of the vector is followed by two square brackets, with in between them the index number of the element that you want to retrieve. So to obtain the third element of vector `pH`, you write:

```
> pH[3]
```

It is not possible to go beyond the length of the vector, so `pH[9]` gives an error: `NA`, which stands for Not Available, which is the code for missing data.

It is also possible to get multiple elements at a time, in which case the returned value is, of course, a vector.

```
> thirty = round( runif(30,0,100) ,0 )
> first.ten = thirty[1:10]
> every.second = thirty[seq(2,30,2)]
```

What did the `runif` function do?

Can you now make a vector that contains every fifth element?

Logic

Sometimes you may want to filter datasets based on certain conditions. For example, you may have two vectors, but only want to have the elements of the first vector for which the corresponding elements in the second vector are above a certain threshold. For this, logic operations are used.

```
> pH = c(4.3, 4.7, 4.2, 4.5, 7.0, 6.7, 7.2, 7.5)
> acid = pH < 7
> acid
```

Produces:

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

So, this is a vector of whether the condition is true or false. You can now use this vector to access the elements of another vector, where only the ones that are `TRUE` here are returned.

```
> num.species = c(12, 15, 11, 13, 21, 11, 19, 18)
> num.species[acid]
```

This produces:

```
[1] 12 15 11 13 11
```

Of course, this result can also be obtained at once:

```
> num.species[pH < 7]
```

Conditions can also be combined to get more powerful filtering. For a change, type the following into a new, empty text-document (for example in R-Studio) rather than into the console and then execute the lines:

```
soil = c("peat","peat","peat","peat","sand","sand","sand","sand")
pH = c(4.3, 4.7, 4.2, 4.5, 7.0, 6.7, 7.2, 7.5)
```

```
num.species = c(12, 15, 11, 13, 21, 11, 19, 18)
```

If you now want to obtain the average number of species from sand-grounds with a pH of at least 7.0, you can write in one time:

```
> mean(num.species[soil == "sand" & pH >= 7.0])
```

There are two things to note here. First, to assess equality, == is used, and not = since the latter was already used to assign values to variables. Errors in this distinction are very common, even among experienced users, and often hard to find. In our example here an error-message would be produced when such a mistake is made, but this is certainly not always the case. Second, the logical operator & is used to combine logical statements. Its complement operator is |, which stands for a logical "or". So to get the numbers of species for all samples with a pH lower than 4.5 and higher than 7.0, you write:

```
> num.species[pH < 4.5 | pH > 7.0]
```

Another useful operator is !=, which stands for "not equal to". So the above mean could also have been obtained with:

```
> mean(num.species[soil != "peat" & pH >= 7.0])
```

How do you get the average pH for all plots with more than 15 species?

3. Arrays & matrices

Creating a matrix

If you have several variables measured for the same samples, it is possible to combine these into a single data matrix, rather than to put all variables separately into vectors. So if for the above samples you would not only have measured pH, and the number of species, but also nitrogen levels:

```
nitrogen = c(12.5, 10.1, 9.8, 11.6, 13.0, 12.4, 11.6, 8.4)
```

Now you can combine these three variables into a single matrix, using the `cbind` command, where the "`c`" stands for "column". Later you will learn how to immediately read data matrices from files, as you would not normally type your data into your R-script.

```
eco.data = cbind(pH, num.species, nitrogen)
```

You now have a data matrix that contains your data. If you now print the matrix to the console, you will see that the output is indeed nicely tabular and that R nicely used the original variable names as the column names. You can now do several operations on your matrix, for example:

```
> dim(eco.data)
> t(eco.data)
```

There are also other ways in which you can create a matrix (by the way, R calls them arrays). Next to the above-used `cbind` function, there is also an `rbind` function which will combine vectors as rows into a matrix. For example, you can create an array and fill them with data using the `array` command.

```
> empty.matrix = array(1, dim = c(5, 8))
```

Another method is to take a vector and give it dimensions. Note that arrays are not limited to two dimensions:

```
> g = 1:56
> dim(g) = c(7,4,2)
```

Can you now create an array of 5 columns and 16 rows, filled with random numbers?

Array calculations

Performing mathematical operations with arrays works in the same way as for vectors, as calculations are performed element-by-element (if you want to do matrix algebra, you have to use a special syntax, which is beyond the scope of the introduction). When a calculation is done using both an array and a vector, the elements of the vector will be repeated to match the size of the matrix. In this case the calculations are first performed by going over rows, then over columns.

```
# create a matrix of 7 rows, 3 columns
a = 1:21
dim(a) = c(7,3)
# now do some operations with it
a/4
sqrt(a)
a * 1:7
a * 1:3
```

Do you understand the output of the lower two examples? How do you multiply the first column by one, the second column by two, and the third column by three?

Accessing array elements

Just like vector-elements, array elements can be accessed using square brackets, where comma's are used to separate elements from the different dimensions. For a two-dimensional matrix the syntax is: `array[row, column]`. So, to get the element at the third row of the second column you type:

```
> eco.data[3, 2]
```

If the column or row index is left blank, all elements are returned. So to get all the data for the third sample:

```
> eco.data[3,]
```

Also here logic operators can be used, so to get only the samples from peat:

```
> eco.data[soil == "peat",]
```

To get a vector with the means for all columns in the matrix, you could of course write:

```
> all.means = c( mean(eco.data[,1]), mean(eco.data[,2]),
     mean(eco.data[,3]) )
```

However, this quickly gets a nuisance if you have a matrix with a very large number of columns. Here the `apply` function comes to the rescue, which can apply a specified function to the rows of columns of an array. The use of this function is a bit more complex than what you have seen before, as it requires three different arguments. The first argument is the array on which you want to apply the function, the second is a number specifying the `margin`: a `1` indicates that the function is applied over rows, and `2` indicates columns. The third argument is the function that is to be used. So to calculate the mean for every column, you can simply write:

```
> all.means = apply(eco.data, 2, mean)
```

How can you quickly calculate the mean for only the samples from peat soils?

4. Factors, lists & data frames

Factors

Above, you used the vector "soil" to specify a classification of our data. While it's fine to use vectors for this, there is also a specialised data type available in R, the so-called factor. Factors are mostly used for formulating statistical models, e.g. for an ANOVA or a regression analysis. The easiest way to create a factor is to call the `factor` function on a vector.

```
> country= c("Ned", "Ned", "Bel", "Bel", "Ned", "Ned", "Bel", "Bel")
> country.fact = factor(country)
```

One difference between a normal vector and a factor (apart from the subtle difference in pronunciation) is the way their output is presented. When a factor is printed to the console, it also prints the levels that it contains (go ahead and give it a try....). If you only want to know the levels of a factor, you can use the `levels` function:

```
> levels(country.fact)
```

produces

```
[1] "Bel" "Ned"
```

You will undoubtedly have noticed that the levels of the factor are by default returned in alphabetical order. A factor does not necessarily have to contain strings, it can also contain numerical data, in which case the levels will be returned sorted from smallest to largest. In some cases, this is not desired. For the above example, you might want to always have the Netherlands first and Belgium last. In that case, you can use a so-called ordered factor, which you can create using the `ordered` function.

Lists

A list is a collection of multiple items, which can be assessed by their names. These items do not have to be of the same type or size. Lists are therefore often used by complex functions to return different types of results.

```
> eco.combined = list(name="nature data", substrate=soil,
    country=country.fact, data=eco.data)
```

Here `name`, `substrate`, `country`, and `data` are the names of the three elements that are put into the list `eco.combined`. You can see that the data are of different types: `name` is a single string, `substrate` a vector of strings, `country` a factor, and `data` an array. You can now access the elements by their names:

```
> eco.combined$substrate
> eco.combined$name
```

If you are lazy, the names of the elements of a list can also be abbreviated to the least number of characters necessary to distinguish them:

```
> eco.combined$s
> eco.combined$n
```

Finally, the elements of a list are numbered automatically, so you can also access them by number using a double pair of square brackets:

```
> eco.combined[[2]]
> eco.combined[[1]]
```

However, I generally advise against those latter two methods, because it decreases the readability of your code. It is better to type a few more characters so that if you reread your code a few months later, you still know what you did. If you use R-studio, it shows autocomplete options, which is really helpful to avoid typos and save time. If you ever need to know which names were used for constructing a list, use:

```
> names(eco.combined)
```

Data frames

Data frames are multi-dimensional data sets, which are a bit of a cross between lists and arrays. In an array, all columns had to contain the same type of data, whereas a data frame can consist of a mixture of data types as long as they are all of the same length. So you can have a data frame where some of the columns contain strings or factors and others contain numerical data. Like a list, the variables can get a name.

```
> eco.frame = data.frame(soil=factor(soil), country=country.fact,
    pH=pH, num.species=num.species, nitrogen=nitrogen)
```

What happens if you don't supply names for the variables in your data frame?

As for lists, the variables (columns) can be retrieved using that name, or using the double square brackets. However, it is also possible to use the same syntax as for arrays to get to the individual elements. So the following are all equivalent:

```
> eco.frame$soil
> eco.frame[[1]]
> eco.frame[,1]
```

Of course, you can also get to the individual elements using the same notation as for arrays. So the pH for the third sample can be obtained using:

```
> eco.frame[3,3]
```

Can you now also filter the matrix to only get the samples from Belgium? Can you think of different ways to do this?

5. Working with files

The working directory

When accessing files, R has the concept of a working directory: the location in the file system that is used as a reference point when looking for or creating files. The easiest way to work with R is to make a folder with all your data in it, and then use that as a working directory in R. You can change the working directory using the "Change Working Directory..." command from the "Misc" menu. Or use `setwd()` as we introduced above in the section on R-Studio. If you want to know the current working directory, just look at the upper left corner of your console window or type `getwd()`.

Writing data

If you want to write data to disk, the easiest way is the `write.table` function. It can take many possible arguments, which can quite drastically change the way the data is organised in the output. If you ultimately want to get your data back into Excel, the following command is useful to save a tab-delimited text file that Excel can read:

```
> write.table(eco.frame, file="example R output.txt", quote=FALSE,
    sep="\t")
```

Reading files

The most straightforward way of getting your data into R is via a tab-delimited text file, for example if your data currently exists in Excel. If you save that file to your working directory, you can read it into R with the `read.table` function:

```
> eco.from.file = read.table("example R output.txt", header=TRUE)
```

The function returns a data frame with the data in it, and it automatically guesses whether a column contains numerical data or a factor. The `read.table` function can take many possible arguments (see the corresponding Help file), that I will not discuss here, but you may need `dec=""` if you have a system that uses commas instead of periods before decimals. If you want to know what the `header` argument does, just see what happens when you leave it out (or set it to `FALSE`, which is the default).

6. Functions

Arguments

Up to now, you mostly have been using very simple functions, that take only a single argument (`sqrt`, `mean`, `var`). However, in the last examples, you got a taste of more flexible functions (`read.table`, `write.table`) that can take many different arguments. So now let's take a closer look at functions and how they work.

The arguments that can be given to a function can be roughly classified into two classes: required and optional. Required arguments are those that are really needed to perform the function. For example, the `mean` function has one required argument, namely the variable from which the mean is calculated. Optional arguments can be omitted (so not specified when the function is called), and in that case they will take a default value. For example, above you saw that the `read.table` function has an optional `header` argument that takes a default value of `FALSE`. You also saw that failing to recognise that a function has some optional arguments can lead to errors, in this case in reading the data correctly.

You can tell from the definition of a function which arguments are required and which are optional. The function definition, its arguments, and their meaning can be found in the help files. For example, the function `mean` can actually take two optional arguments besides the single required argument. The function is defined as:

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

According to the Help files the arguments are defined as:

`x` An R object. Currently there are methods for numeric/logical vectors and data, date-time, and time interval objects, and for data frames all of whose columns have a method. 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.

Here, the arguments `trim` and `na.rm` are optional which you can tell because they both have a default value defined (`0` and `FALSE`, respectively). The argument `x` is required because it does not have a default value defined. In this case, the ellipsis argument (`...`) simply means that any further arguments provided by the user will be ignored. Knowing this, how can you calculate the average pH for the `eco.from.file` data?

Writing your own functions

If you want to do something over and over again, it may be handy to write your own function for it. A function is defined by giving it a name, followed by the word `function` and then the arguments of the function between brackets. The `expression`, the "body" of the function, then follows in curly brackets behind the definition:

```
name <- function(arg_1, arg_2, ...){ expression }
```

However, before you start writing your own functions, it is useful to know something about ways to loop and repeat calculations. Often when writing a function you will need to repeat certain calculations a number of times or until a certain condition is met. One important way to do this is using a "loop". The most common type of loop is the so-called `for`-loop which is found in almost all programming languages. The basic syntax is:

```
for( variable in vector) { expression }
```

The loop iterates over the elements of `vector` one by one, and at every step of the iteration puts the corresponding value into `variable`. So to one-by-one print the values of a vector of random numbers to the console:

```
rn = runif(10)
for(i in rn){
     print(i)
}
```

Note that here we use the `print` function to output something to the console, since simply typing the variable name does not always work within loops (and functions). Another type of loop is the `while`-loop, here the expression is performed as long as a certain condition is true.

```
while( condition ) { expression }
```

So let's say we want to roll a virtual dice until we rolled a six (type this into a script so you can run it several times).

```
eyes = 0
while(eyes != 6){
     eyes = round(runif(1, 1, 6),0)
     print(eyes)
}
```

Another way to develop your algorithm when you are writing your own function is using `if-else` statements:

```
if( condition ) { expression } else { expression }
```

Note that the `else` part of this statement is optional, so an `if` statement can be used to only do something if the condition is true. As an example of the use of an `if-else` statement, let's write a small function that tests whether a single number is odd:

```
is.odd <- function(x){
     if(x %% 2 == 1){
          return(TRUE)
     } else {
          return(FALSE)
     }
}
```

The `return` command that is used here, gives the output of a function back to the point where the function was called. To use the function, you have to call the function and specify a value of x. So `is.odd(5)` will tell you whether 5 is an odd number or not.

As a more complex example of a function, let's write a function to calculate the harmonic mean, since R does not have a build-in function for this.

```
harm.mean <- function(x){
     if(is.numeric(x) == FALSE){
          return( warning("The data must be numerical") )
     }
     insum = 0
     for(i in x){
          insum = insum + 1/i
     }
     insum = length(x) / insum
     return(insum)
}
```

After the function definition, a test is performed whether the data is actually suitable for calculating a mean. Then a new variable called `insum` is declared and given a value of 0. This variable is only valid within the scope of the function, so it will not be available outside it. Then a `for`-loop is used to go over the elements of the vector one by one and add its inverse to `insum`. The function then uses `return` to give the resulting value back. Now give it a try:

```
> harm.mean( eco.from.file$num.species )
```

or

```
> harm.mean.output = harm.mean( eco.from.file$num.species)
```

Do you understand the difference between these two ways of calling the function?

Though loops can be very handy, R is notoriously slow at performing them, so in general it is better to use vector operations if this is possible. In the `harm.mean` function I used a `for`-loop for pedagogical purposes. Can you think of a way to calculate the harmonic mean without a loop?

7. Graphics

Simple plots

R has very extensive graphical capabilities, though these are not always very straightforward to use. So doing graphics in R requires quite a bit of Googling and reading Help-files, but the results can be rewarding, especially if you want to automate the analysis and visualisation of data.

The most basic function to produce graphics is `plot`, for which the type of plot produced depends on the type of data. For example, for two continuous variables, it produces what is known as XY-plot or a scatterplot.

```
> plot(eco.from.file$nitrogen,eco.from.file$pH)
```

The graph pops up in a new window, but is very crude and often has incorrect titles. If you want to change any aspect of the graph, you will have to change the command and execute it again (use the up-arrow to go back in history and then modify the command), overwriting the old plot. Plotting is (perhaps) more intuitive as it does not overwrite old plots, and allows leaving back and forth between all the graphics produced within a session. This is (perhaps) more intuitive, and useful to explore how we can improve some basic aspects of the plot.

To give the plot more appropriate titles:

```
plot(eco.from.file$nitrogen,eco.from.file$pH, xlab="Nitrogen",
     ylab="pH", main="Nitrogen vs. pH")
```

You can change the colour of the symbols using the `col` argument, and the shape through the `pch` argument. When the latter is given a value between 1 and 25, it will use a variety of symbols, but what happens when you use values higher than 25? Try a range of values including some higher and lower than 25.

```
plot(eco.from.file$nitrogen,eco.from.file$pH, xlab="Nitrogen",
     ylab="pH", main="Nitrogen vs. pH", pch=19, col="red")
```

To change the symbol size, use the `cex` argument, where you can give a value by which the default size is multiplied. Likewise, the size of the axis scale can be changed using `cex.axis`, and the label text using `cex.lab`:

```
plot(eco.from.file$nitrogen,eco.from.file$pH, xlab="Nitrogen",
     ylab="pH", main="Nitrogen vs. pH", pch=19, col="red", cex =
     1.5, cex.axis=0.8, cex.lab=1.2)
```

When you want to add more points to this graph, you can call the `points` function. So if you want to label the peat and sand soils differently, you can split the data in two, then plot the two groups separately:

```
sandy = eco.from.file[eco.from.file$soil == "sand",]
peaty = eco.from.file[eco.from.file$soil == "peat",]
plot(sandy$nitrogen,sandy$pH, xlab="Nitrogen", ylab="pH",
     main="Nitrogen vs. pH", pch=19, col="red")
points(peaty$nitrogen,peaty$pH, col="blue")
```

Why are there only four points on the graph?

If you want to change the scale of the axes of a graph, you should use the `ylim` and `xlim` arguments. Both take as a value a vector of length two. So to set the scale of the x-axis from 0 to 100 you can add the argument xlim=c(0,100). Now fix the problem of the disappearing points.

Other types of graphs

I am not planning to go into detail about all different types of graphs that are possible with R, but here I will simply give some examples. If you want to explore a bit further, just look at the Help-files (or Google) to see which arguments they take and what you can all change about them.

```
> rand = rnorm(1000)
> hist(rand, freq=FALSE, ylim=c(0,0.4))
> curve(dnorm, add=TRUE, col="red")
> qqnorm(rand)
> boxplot(rand)
```