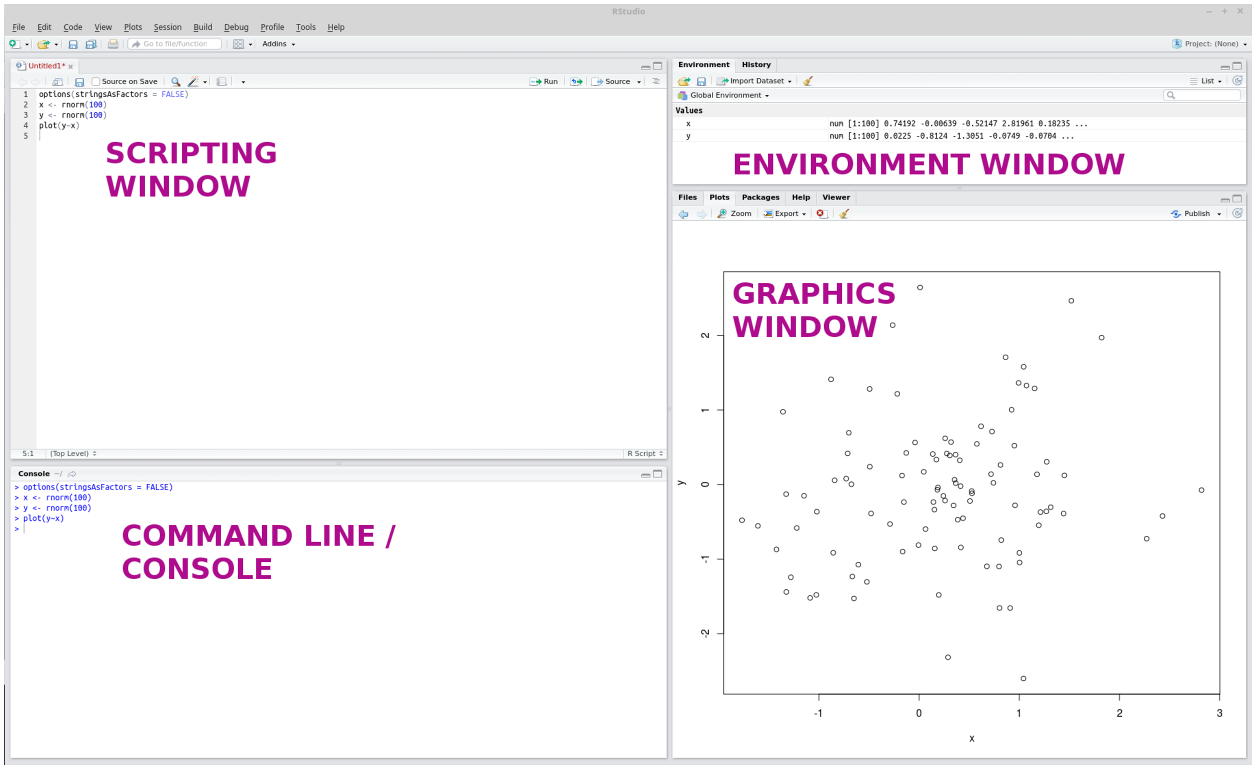
Modelling in Physical Geography

Excercise 1

Konsta Happonen

Getting to know R and RStudio

We use the graphical user interface RStudio to interact with R. The default interface is divided into four windows: scripting window, command line, environment window and graphics window (figure 1). We write our code into the scripting window. When we execute the code, it is sent to the command line/console. The console is also where R prints any messages or errors your commands may invoke. The environment window lists all objects you might have created with your code. Finally, the graphics window is where the results of plotting commands are displayed by default.



```
options(stringsAsFactors = FALSE)
x <- rnorm(100)
y <- rnorm(100)
plot(y~x)
```

```
Values
x    num [1:100] 0.74192 -0.00639 -0.52147 2.81961 0.18235 ...
y    num [1:100] 0.0225 -0.8124 -1.3051 -0.0749 -0.0704 ...
```

```
> options(stringsAsFactors = FALSE)
> x <- rnorm(100)
> y <- rnorm(100)
> plot(y~x)
>
```

Figure 1: The RStudio default interface is divided into 4 windows. If the scripting window is not visible, click the green-white plus sign in the left corner.

Close any scripts that might open automatically. If a new scripting window is not visible, press the green-white plus-sign in the left corner. Some people have a habit of polluting the working environment by saving objects to te global envoronment. If the environment window lists any objects at all, type this command to the console and run it: `rm(list=ls())`. It deletes all objects.

To see how this layout works, write these few lines of code into the scripting window. DO NOT COPY AND PASTE. Typing is a much more effective way of learning how to code.

```
### Modelling in Physical Geography
### Week 1 - excercise 1
### Learn how to use R

## A few lines of code to demonstrate the layout of RStudio
a <- c(1,2,3,4,5) # assign a vector of values 1 through 5 to an object named a
b <- 1:5 # same for b
plot(x=a,y=b) # plot the relationship of a and b
```

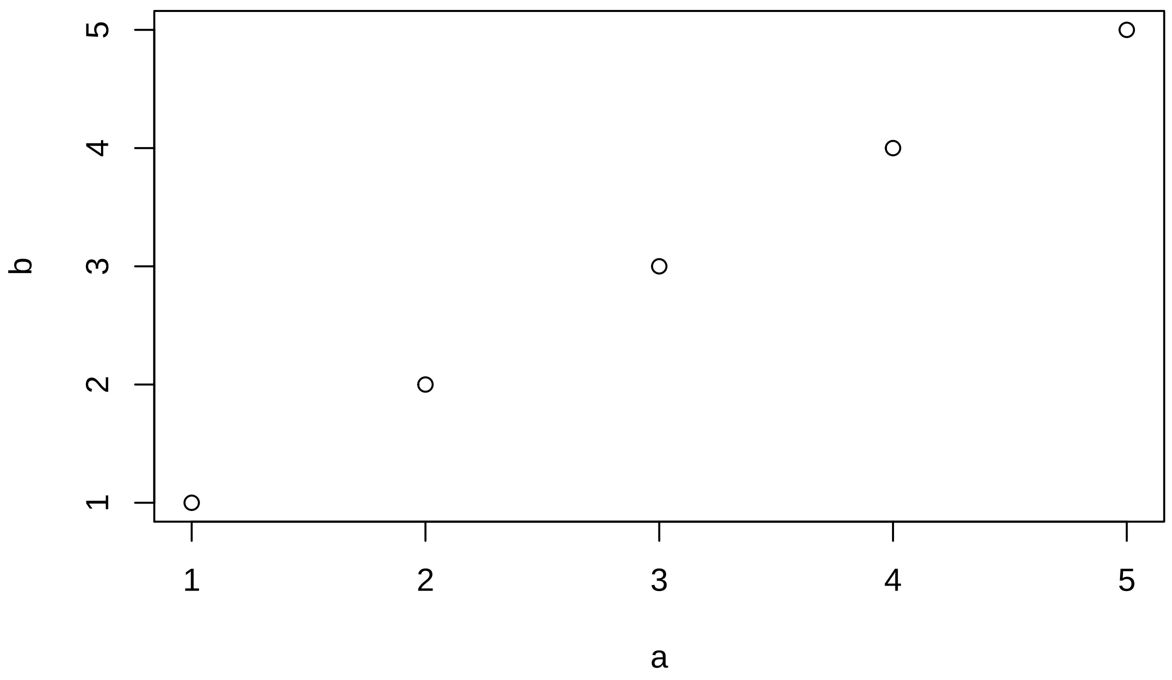


Figure 2: Output of the command `plot`

You execute the code line-by-line with Ctrl+Return. You can also highlight the entire script and execute it with Ctrl+Return. You should see the code being sent to the command line, and a plot similar to above appearing in the graphics window. The above code also demonstrates that there are multiple ways to perform even simple operations. A vector of digits 1–5 can be created both with the commad `1:5` and `c(1,2,3,4,5)`. In addition, the command `seq(from=1,to=5)` would work as well. Just as you can convey the same message in many different ways using natural language, you can achieve the same analysis result with very different lines of code. This flexibility makes the programmatic approach to data analysis very powerful, but also makes it crucial that you arrange and document your code properly.

Commenting

The first thing to learn is that always, ALWAYS comment your scripts. Look at the previous code block. The first row begins with a `#`. This is a sign to the command line not to execute anything on this line. This is called a comment. You can also start a comment mid-line; this is demonstrated on the last three lines.

The aim of commenting is to make the code human-readable. It is much easier to return back to your code when you've written comments that explain what each command is doing. There is no universally accepted right amount of commenting, but too many comments is better than too few. Take a habit of commenting everything you code until you know what each command does by heart.

It is a good idea to use the first few lines of a new script to describe what the script does, in case you forget or if sombody else needs to read your code.

Anatomy of R commands

Most R commands are called by writing the function name followed by brackets. If the function takes any *arguments*, they are written between the brackets. For example, in the command `plot(x=a,y=b)`, the objects `a` and `b` are given as arguments to the function `plot()`. To see what arguments a function can take, you can type `?function_name` on the command line, for example `?plot`.

A major expetion is the assignment arrow `<-`. It is used to create new objects. On the left-hand side you write the name of the object you are creating (or updating), and on the right-hand side you put what ever you want to put in your new object. We already used the assignment arrow twice in the above piece of code (`a <- c(1,2,3,4,5)`).

Working directory

The directory R runs in is called the working directory. This is where R searches for and saves any files you read or write by default. It is a good idea to run each project you start in their own directory. That way any results and figures you create will automatically be associated with the code it was made with.

Let us now create a new directory, and set it as our working directory. You can check your current working directory with the command `getwd()`. Then, create a new subdirectory called `"excercise_1"`. The quotes are important, otherwise R would search for an object named `excercise_1`. Next, set the new subdirectory as your working directory with `setwd()`.

```
getwd() # what is your current working directory?
dir.create("excercise_1") # create directory excercise_1, remember the quotes!
setwd("excercise_1") # set working directory.
```

Data types

We will now go over the most important data types in R: integer, numeric, character, logical, and factor.

Integer and numeric

We have already created one numeric and one integer object. You can check the type and structure of objects with the command `str()`

```
str(a) # print the structure of a
```

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

```
str(b)
```

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

The only meaningful difference between the numeric and integer types is that integers cannot have decimals, and thus consume less memory.

Character

Character objects are used to store text. The text must be quoted, otherwise R will search for an object with that name. You can use the command `paste()` to bind pieces of text together into one character string. Try it now.

```
a
```

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

```
"a"
```

```
## [1] "a"
```

```
str("a")
```

```
##  chr "a"
```

```
text1 <- "Hello"
text2 <- "world!"
paste(text1,text2)
```

```
## [1] "Hello world!"
```

```
paste(text1,2,"u",2)
```

```
## [1] "Hello 2 u 2"
```

Factors

Because R and its predecessors were designed in a time when memory was limited, the default behaviour of R is to convert character-type data to *factors*. This is a rather inconvenient, although memory efficient data type that is unintuitive to handle. This is why I recommend you turn factor conversion off at the beginning of all your scripts.

```
fac_a <- factor(a) # create a factor
str(fac_a)
```

```
##  Factor w/ 5 levels "1","2","3","4",..: 1 2 3 4 5
```

```
options(stringsAsFactors = FALSE) # Turn automatic factor conversion off
```

Logical

Logical data has two possible values: `TRUE` and `FALSE`. They usually arise from logical comparisons.

```r
1 == 1 # is 1 equal to 1?
```

```
## [1] TRUE
```

```r
1 >= 2 # is 1 equal to or greater than 2?
```

```
## [1] FALSE
```

```r
a == b # is each of the elements of a equal to those of b?
```

```
## [1] TRUE TRUE TRUE TRUE TRUE
```

```r
logi <- c(TRUE,FALSE) # assign a logical vector
str(logi) # print the structure of logi
```

```
##  logi [1:2] TRUE FALSE
```

```r
as.numeric(logi) # logical data can be converted to numeric.
```

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

Logical data is useful in subsetting data and as control switches, as we shall later see.

Special (NA,NaN and NULL)

R has reserved words for missing data (**NA**, Not Available), numeric things that are not numbers (**NaN**, Not a Number), and nothingness (**NULL**). NaN values arise for example when one tries to divide something by zero. Some functions require you to specify what to do to NA- and NaN-values. The null object represents nothingness in the R-world, and can for example be used to delete contents of certain objects.

```r
n <- c(1,NA,NaN,3) # missing data
sum(n) # arithmetic operations involving NA return NA
```

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

```r
sum(n, na.rm = TRUE) # the argument na.rm = TRUE removes NA- and NaN- values before summing.
```

```
## [1] 4
```

Vectors, lists, matrices, data frames and arrays.

To do analysis you need a way to arrange data points into larger collections. When deciding what kind of object to store your data in, you mostly need to consider two things:

1. How many dimensions does your data have?
2. Are all your data of the same type?

	One-dimensional	Two-dimensional	N-dimensional
Data of same type	`c()`	`matrix()`	`array()`
Data of different types	`list()`	`data.frame()`	`list()`

Table 1: A lookup table for choosing the right container class for your data.

Vectors `c()`

The most basic container is a one-dimensional vector. All members of a vector must be of the same basic type, so that you cannot mix character and numeric data. Singular data points are actually vectors of length one. To create longer vectors, you concatenate them together with the command `c()`. To access specific members of the vector, use square brackets. For example, `x[3]` prints the value of the third member in the vector x.

```
vec1 <- c("a",2)
str(vec1) # numeric data was converted to character data
```

```
##  chr [1:2] "a" "2"
```

```
vec2 <- c("a","3")
vec3 <- c(vec1,vec2) # combine two vectors
length(vec3) # vec3 is of length 4
```

```
## [1] 4
```

```
vec3[3] # the third member of vec3 is "a"
```

```
## [1] "a"
```

```
vec3[1] <- "EKA" # replace the first member of vec3 with the character string "EKA"
vec3
```

```
## [1] "EKA" "2"   "a"   "3"
```

Lists `list()`

Lists are a flexible class of objects that can store data of different types. Each member of a list is itself a list. The practical implication is that instead of using singular square brackets to access the contents of a list, you use double square brackets, e.g. `x[[1]]`. Lists can have an any number of dimensions, and the members of lists do not need to be of the same length.

```
list1 <- list("a",2)
str(list1) # both members retained their original types
```

```
## List of 2
##  $ : chr "a"
##  $ : num 2
```

```r
str(list1[2]) # members of lists are also lists
```

```
## List of 1
##  $ : num 2
```

```r
str(list1[[2]]) # double square brackets access the contents of the list
```

```
##  num 2
```

```r
list2 <- list("b","c", c(3,4,5)) # Lists can hold objects of different lengths
list2
```

```
## [[1]]
## [1] "b"
##
## [[2]]
## [1] "c"
##
## [[3]]
## [1] 3 4 5
```

```r
list2[[3]][2] # Access the third list member's second value.
```

```
## [1] 4
```

```r
c(list1,list2) # Lists can be concatenated
```

```
## [[1]]
## [1] "a"
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] "b"
##
## [[4]]
## [1] "c"
##
## [[5]]
## [1] 3 4 5
```

Data frames `data.frame()`

Data frames are the most important container class in R. They are two-dimensional, and thus have rows and columns. All members of one column must be of the same data type, and all columns must be of equal length. Rows of the data frame thus represent individual observations, while columns represent identifiers or measured variables. When creating data frames, you specify names for your variables: `data.frame(variable1=c(1,2),variable2=c("a","b"))`.

Members of the data frame are accessed with — surprise, surprise — square brackets. But because data frames are two-dimensional, you have to specify the dimensions. This is done with a comma `x[1,2]`. The first number refers to rows, and the second to columns. If either number is left blank, then it is taken to mean all data on that dimension. For example, `x[,c(2,3)]` refers to all rows but only the second and third columns of data frame `x`.

You can access specific variables of a data frame by using the `$`-operator: `x$variable1`. This returns a one-dimensional vector, so a comma is no longer needed in further subsetting.

Data frames can be joined together vertically with `rbind()` (bind rows) and horizontally by `cbind()` (bind columns).

```
## create a new data frame with variables 'name' and 'race'
creatures <- data.frame(name = c("Frodo","Arwen","Shelob"),
                        race = c("hobbit", "elf", "spawn of Ungoliant"))

## dividing long expressions on multiple rows helps maintain the readability of your code.
creatures$legs <- c(2,2,8) # add a new column
creatures$exists <- FALSE # vectors of length 1 are rotated to fill the column
creatures
```

```
##     name               race legs exists
## 1  Frodo             hobbit    2  FALSE
## 2  Arwen                elf    2  FALSE
## 3 Shelob spawn of Ungoliant    8  FALSE
```

```
str(creatures)
```

```
## 'data.frame':    3 obs. of  4 variables:
##  $ name  : chr  "Frodo" "Arwen" "Shelob"
##  $ race  : chr  "hobbit" "elf" "spawn of Ungoliant"
##  $ legs  : num  2 2 8
##  $ exists: logi  FALSE FALSE FALSE
```

```
creatures[2,] # second row
```

```
##    name race legs exists
## 2 Arwen  elf    2  FALSE
```

```
creatures[3,c(3,4)] # third row, third and fourth columns
```

```
##   legs exists
## 3    8  FALSE
```

```
creatures$name[3] # third name in the data frame
```

```
## [1] "Shelob"
```

```
str(rbind(creatures,creatures)) # combine data frames
```

```
## 'data.frame':    6 obs. of  4 variables:
##  $ name  : chr  "Frodo" "Arwen" "Shelob" "Frodo" ...
##  $ race  : chr  "hobbit" "elf" "spawn of Ungoliant" "hobbit" ...
##  $ legs  : num  2 2 8 2 2 8
##  $ exists: logi  FALSE FALSE FALSE FALSE FALSE FALSE
```

```r
str(cbind(creatures,creatures))
```

```
## 'data.frame':    3 obs. of  8 variables:
##  $ name  : chr  "Frodo" "Arwen" "Shelob"
##  $ race  : chr  "hobbit" "elf" "spawn of Ungoliant"
##  $ legs  : num  2 2 8
##  $ exists: logi  FALSE FALSE FALSE
##  $ name  : chr  "Frodo" "Arwen" "Shelob"
##  $ race  : chr  "hobbit" "elf" "spawn of Ungoliant"
##  $ legs  : num  2 2 8
##  $ exists: logi  FALSE FALSE FALSE
```

```r
names(creatures) # Access the names of your variables
```

```
## [1] "name"   "race"   "legs"   "exists"
```

```r
names(creatures)[2] <- "species" # Biologically more accurate
```

Matrices `matrix()`

Matrices are like data frames, but can only hold data of a single type. Matrices are needed for efficient calculation of large datasets. They are created with the command `matrix()`. Matrices can also be combined with `rbind` and `cbind`.

```r
dada <- 1:12 # some data to fill the matrix
matrix(dada, ncol = 4) # create a matrix with 4 columns
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```r
matrix(dada, nrow = 3) # create a matrix with 3 rows
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

```r
mat <- matrix(dada, nrow = 3, byrow = TRUE) # insert data row-by-row
mat <- t(mat) # transpose (flip) matrix
mat
```

```
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

```
colnames(mat) <- c("Neo","Morpheus","Trinity") # Name the columns of the matrix
rbind(mat[,"Trinity"],mat[,3])
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    9   10   11   12
## [2,]    9   10   11   12
```

Arrays `array()`

Arrays are multidimensional matrices. Otherwise they follow the same logic. Subsetting operations for N-dimensional arrays have N-1 commas.

```
arr <- array(dada, dim = c(3,2,2)) # array has 3 rows, two columns and two "layers"
arr
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
```

```
arr[1,2,] # first row, second column, entire third dimension.
```

```
## [1]  4 10
```

Attaching packages to gain new functionality

A major advantage of using R is that its functionality can be increased by installing contributed packages. These are collections of new functions made by programmers and scholars around the world. Packages are hosted on a multitude of sites, the most important of which is The **C**omprehensive **R** **A**rchive **N**etwork, or CRAN. By default all you have to do is type `install.packages("package_name")` and *Voilà*, your R installation has just been improved! In this excercise we will not be installing any packages due to university cybersecurity policies, but try it on your own computer!

After installation, the package still has to be loaded before its functions become available to the user. For this, use the command `library(package_name)` (no quotes needed).

You can also access specific functions from packages by prefixing te function name with the package name followed by two colons: `package::function()`

```
## install.packages("boot") # install the package if needed
## logit(0.9) # Throws an error because the function "logit"  is not available.
library(boot) # load the package "boot"
logit(0.9) # Logit works!
```

```
## [1] 2.197225
```

```
boot::logit(0.9) # This would have worked withoud loading the package
```

```
## [1] 2.197225
```

Running simple calculations in R

R also functions as a calculator with the mathematical operators `+`, `-`, `*`, `/`, `^`, `log()`, `exp()` and `%%` (modulus/jakojäännös). These operations are *vectorized*, meaning that you can perform the calculations on objects with more than one member.

```
a
```

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

```
a - a
```

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

```
a + 1
```

```
## [1] 2 3 4 5 6
```

```
a / 10
```

```
## [1] 0.1 0.2 0.3 0.4 0.5
```

```
a ^ a
```

```
## [1]    1    4   27  256 3125
```

```
log(a)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

```
exp(log(a))
```

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

```r
a%%4
```

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

```r
a-1*10^2 # R honours the order of operations
```

```
## [1] -99 -98 -97 -96 -95
```

A particularily important statistic for a collection of numbers is the mean. There are multiple ways of calcucating it for different types of objects.

```r
mean(creatures$legs)
```

```
## [1] 4
```

```r
sum(creatures$legs)/length(creatures$legs)
```

```
## [1] 4
```

```r
rowMeans(mat)
```

```
## [1] 5 6 7 8
```

```r
colMeans(mat)
```

```
##      Neo Morpheus  Trinity
##      2.5      6.5     10.5
```

```r
rowSums(mat)/NCOL(mat)
```

```
## [1] 5 6 7 8
```

```r
colSums(mat)/NROW(mat)
```

```
##      Neo Morpheus  Trinity
##      2.5      6.5     10.5
```

Removing objects and saving your script

Once you are done with an object, you can remove it with `rm()`

```r
rm(mat) # Destroy the matrix
```

That concludes the excercises for today. You can then save your script by clicking the save icon. Give it an informative name, like `mpg-excercise1-myname.R`. R scripts always end in the suffix `.R`.

Where to find help

Prefixing a function name with a question mark (`?function_name()`) opens the help page for that function, which describes the functions arguments, return values and usage. Using double question marks (`??search_term`) tries to match your search term to function names and descriptions, which is handy if you are not entirely sure what you are looking for. Google is the best friend of even experienced programmers, look especially for threads on question-answer sites like Stack Overflow. Also remember to ask. Ask your friends, ask your teachers, your supervisors and family. Many data analysis tasks are more about problem-solving and logic than finding the specific function that does what you need.

PS

When you close your R session, you are prompted to save your workspace. This means that objects you created are saved and available next time you start R. NEVER SAVE YOUR WORKSPACE. The idea of programmatic data-analysis is that you can rerun your scripts to reproduce your results. This cannot be guaranteed if you have rogue objects polluting the workspace. Also, old objects consume memory and can slow down your computer.