Modelling in Physical Geography

Excercise 3

Konsta Happonen

Preparations

Make a new directory for this excercise and set it as your working directory. Download the directory called **sentence** from the course moodle page to that directory.

Reading data to R

R can read in text-based tables, its own binary file formats, and with the right packages, almost any kind of data (including spreadsheets). For this excercise we will focus on reading data from a text-based format called **CSV** (comma-separated values). The benefit of storing data in plain text is transferability: all software can read plain text. Below is an example table in CSV format; values on the first line are the column names, and the observations are on subsequent lines.

```
X,Y,Z
1,2,3
2,2,1
3,1,3
```

The command `read.csv()` transforms CSV files into data frames. The argument `sep` specifies, what the separator between values is. The default value is a comma: `","`. A common problem with people who work on computers set to a Finnish locale is that since the Finnish decimal separator is also a comma, the default behaviour of spreadsheet software when saving data as CSV is to use a semicolon `";"` instead. Check your separators if you ever encounter this error! The same caution applies to decimal separators as well, which are specified with the argument `dec`.

You should have downloaded from the course page a directory with several CSV files. Let us read one of them. The file contains one observation of the variables wordno and word.

```
options(stringsAsFactors = FALSE)
read.csv("sentence/word1.csv")
```

```
##   wordno word
## 1      1 This
```

Next, we want to know how many files are in the directory `sentence`. We do this with the command `list.files()`

```
list.files("sentence")
```

```
##  [1] "word10.csv" "word11.csv" "word12.csv" "word13.csv" "word14.csv"
##  [6] "word15.csv" "word16.csv" "word17.csv" "word18.csv" "word19.csv"
## [11] "word1.csv"  "word20.csv" "word21.csv" "word22.csv" "word23.csv"
## [16] "word24.csv" "word25.csv" "word26.csv" "word27.csv" "word28.csv"
## [21] "word29.csv" "word2.csv"  "word30.csv" "word31.csv" "word32.csv"
## [26] "word33.csv" "word34.csv" "word35.csv" "word3.csv"  "word4.csv"
## [31] "word5.csv"  "word6.csv"  "word7.csv"  "word8.csv"  "word9.csv"
```

The directory contains 35 CSV files, each of which contains one word of a sentence. Our task today is to read them all to R, combine them in one data frame, sort the words in correct order and paste them into one character string.

First, we need to convert the 35 files into data frames. You *could* read each file separately into its own object, but that would take forever. Luckily, there is an easier option.

Apply-family of functions

The `*apply`-family of functions is large and diverse. The common denominator is that they all take a vector of arguments, and then apply a command to each of those arguments. In this excercise we will use the command `lapply`, whose return value is a list (hence the `l` in the name).

We first read the file paths into an object. We use the argument to make `list.files()` return the location instead of the name of each file.

```r
paths <- list.files("sentence", full.names = TRUE)
str(paths)
```

```
##  chr [1:35] "sentence/word10.csv" "sentence/word11.csv" ...
```

We then give these file paths to `lapply()` and `read.csv()`. The `FUN` argument to lapply is a function name, so no brackets required. If you needed to specify further arguments to `read.csv()`, you could do after `FUN`. The ellipsis argument lets you pass named arguments on.

```r
## Apply read.csv() to all objects of paths
words <- lapply(paths, FUN = read.csv)
head(words,3) # print first three objects
```

```
## [[1]]
##   wordno   word
## 1     10 hidden
##
## [[2]]
##   wordno word
## 1     11   in
##
## [[3]]
##   wordno  word
## 1     12 their
```

```r
## the ellipsis is for passing arguments
## lapply(paths, FUN = read.csv, sep = ",")
```

We now have a list 35 data frames, and we want to combine them. Recall the command `rbind()` that is used to bind together data frames. We can use it to bind the elements of the list together…

```
rbind(words[[1]],words[[2]],words[[3]])
```

```
##   wordno   word
## 1     10 hidden
## 2     11     in
## 3     12  their
```

…but this is again slow and cumbersome. The command `do.call()` comes to our rescue. `do.call()` takes as its first argument `what` a function, and passes to it the elements of argument `args`, which should be a list.

```
words <- do.call(what = rbind, args = words) # combine data frames to one.
str(words)
```

```
## 'data.frame':    35 obs. of  2 variables:
##  $ wordno: int  10 11 12 13 14 15 16 17 18 19 ...
##  $ word  : chr  "hidden" "in" "their" "own" ...
```

The words in the data frame are still scrambled. We can use the command `order` to arrange the data frame acording to the variable `wordno`.

```
## Think about what the order command does.
## data.frame(order(words$wordno),words$wordno)
words <- words[order(words$wordno),] # sort data frame according to wordno
```

Data frames are not the best for reading long sentences. Print the text to your console with `cat()`

```
## cat(words$word)
```

Writing/saving data

We can now save our data. The recommended form for saving tables is CSV with the command `write.csv()`.

```
## We set row.names to FALSE, because we do not want to save them.
write.csv(words, file = "words.csv", row.names = FALSE)
```

Logical tests

Logical tests or comparisons are expressions that return either `TRUE` or `FALSE`. The table below lists some logical operators.

Expression	Meaning
`x == y`	x is equal to y
`x != y`	x is not equal to y
`x > y`	x is greater than y
`x >= y`	x is greater than or equal to y
`x < y`	x is smaller than y
`x <= y`	x is smaller than or equal to y
`x %in% y`	x is a member of the set y
`is.na(x)`	x is missing data

Try them now.

```r
1 == 2
```

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

```r
1 != 2
```

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

```r
1 > 1
```

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

```r
1 >= 1
```

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

```r
1 <= 1
```

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

```r
1 <= 1
```

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

```r
1 %in% 1:5
```

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

```r
1 %in% 2:5
```

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

```r
is.na(1)
```

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

```r
is.na(NA)
```

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

Note that the singular `=` is equal to `<-`, so always remember to use `==`.

Logical expressions and subsetting

Logical expressions can be used to separate data of interest from a larger data frame. To demonstrate this, we again use the `brambles` dataset. Giving a logical vector as a subsetting argument returns those rows which get the value `TRUE`.

```r
brambles <- boot::brambles
brambles[FALSE,] # the FALSE value is rotated, returns an empty data frame.
```

```
## [1] x   y   age
## <0 rows> (or 0-length row.names)
```

```r
head(brambles[,c(TRUE,FALSE,TRUE)]) # select the first and third column.
```

```
##       x age
## 1 0.677   0
## 2 0.676   0
## 3 0.681   0
## 4 0.683   0
## 5 0.776   0
## 6 0.794   0
```

Now, this is not as easy as using integers to directly identify the rows we want. However, we can use logical tests to create logical vectors, so that we can pick certain kinds of observations from our data.

```r
logi <- brambles$x > 0.5 # Which observations have an x-coordinate greater than 0.5?
str(brambles[logi,]) # return only those observations
```

```
## 'data.frame':    461 obs. of  3 variables:
##  $ x  : num  0.677 0.676 0.681 0.683 0.776 0.794 0.944 0.948 0.983 0.986 ...
##  $ y  : num  0.001 0.022 0.031 0.038 0.028 0.033 0.011 0.01 0.077 0.084 ...
##  $ age: num  0 0 0 0 0 0 0 0 0 0 ...
```

```r
str(brambles[brambles$age %in% c(0,2),]) # Only ages 0 and 2
```

```
## 'data.frame':    438 obs. of  3 variables:
##  $ x  : num  0.677 0.676 0.681 0.683 0.776 0.794 0.944 0.948 0.983 0.986 ...
##  $ y  : num  0.001 0.022 0.031 0.038 0.028 0.033 0.011 0.01 0.077 0.084 ...
##  $ age: num  0 0 0 0 0 0 0 0 0 0 ...
```

Logical values can be reversed with the operator `!`.

```r
str(brambles[!brambles$age %in% c(0,2),]) # Only ages that are not 0 or 2
```

```
## 'data.frame':    385 obs. of  3 variables:
##  $ x  : num  0.27 0.306 0.787 0.81 0.81 0.817 0.97 0.937 0.804 0.934 ...
##  $ y  : num  0.423 0.368 0.11 0.112 0.082 0.108 0.047 0.066 0.1 0.074 ...
##  $ age: num  1 1 1 1 1 1 1 1 1 1 ...
```

Subsetting is a very important skill to master; commit it to heart to avoid much suffering in the future.

if-else -structures

The words `if` and `else` are used to control the flow of your script. The commands following an `if`-statement are executed only if the argument to `if` returns `TRUE`. Expressions following an `if`-statement are typically wrapped in curly brackets, which signify that the `if` statement applies to all the lines of code between the brackets.

```
## Returns 1 if 1 == 2
if(1 == 2){
    1
}

## Returns 2 if 1 == 1
if(1 == 1){
    2
}
```

```
## [1] 2
```

The first expression does not return anything, because one is not equal to two. The argument of the second expression on the other hand evaluates to `TRUE` (because one is indeed one), and so the contents of the curly brackets are evaluated.

The word `else` is a handy way to specify that something should be done if the previous `if`-statement did not get executed.

```
## If the data set is less than 10 rows, it is small
## Otherwise it is large
if(NROW(brambles) < 10){
    cat("The dataset is small")
} else {
    cat("The dataset is large")
}
```

```
## The dataset is large
```

You can also bind together several `if`-statements with `else`.

```
## 0-9 rows: small dataset
## 10-999 rows: intermediate dataset
## 1000+ rows: large dataset
if(NROW(brambles) < 10){
    cat("The dataset is small")
} else if(NROW(brambles) < 1000){
    cat("The dataset is intermediate")
} else {
    cat("The dataset is large")
}
```

```
## The dataset is intermediate
```

For-loops

Sometimes you need to do some operation multiple times, for example to many objects. Sometimes the operation is so complex that simple `*apply` commands do not suffice. An easy-to-understand alternative is the `for`-loop. A demonstration follows:

```
## Print the number of each iteration of the loop
for(i in 1:6) {
    cat(" Round", i)
}
```

```
##  Round 1 Round 2 Round 3 Round 4 Round 5 Round 6
```

The `i` in the previous code is an index vector of length one. The loop goes through the values `1:6` one at a time. In each iteration, it executes the expression between the curly brackets but replaces the `i` with one of the members in `1:6`.

The index vector can have any name, and does not need to hold integers.

```
names <- c("Juha", "Konsta", "Annina", "Miska")
for(nimi in names){
    print(paste0(nimi, ", step forward!"))
}
```

```
## [1] "Juha, step forward!"
## [1] "Konsta, step forward!"
## [1] "Annina, step forward!"
## [1] "Miska, step forward!"
```

For-loops are especially useful when combined with if-else statements.

```
for(i in names){
    if(i %in% c("Konsta", "Juha")){
        print("There you are teacher!")
    } else {
        print("You are not the teacher today!")
    }
}
```

```
## [1] "There you are teacher!"
## [1] "There you are teacher!"
## [1] "You are not the teacher today!"
## [1] "You are not the teacher today!"
```

The additional words `break` and `next` can be used to control the flow of the loop in more detail. The command `break` terminates the loop, while `next` skips the remaining expressions in the current iteration and jumps to the next one.

```
for(i in names){
    if(i == "Konsta"){
        next # skips this iteration
    } else if(i == "Juha"){
        print("There you are Juha!")
    } else if(i == "Annina") {
        print("There you are Annina!")
        break
    } else {
        print("Hello professor!")
    }
}
```

```
## [1] "There you are Juha!"
## [1] "There you are Annina!"
```

For-loops can be used the same way as apply commands, although one must remember a few more steps.

First, one must create a container vector for the results of the calculcation. The index vector of the for-loop should be a vector of integers from one to the numer of observations. Then,

```
## Create the container: a numeric vector with length equaling
## the number of operations we need to conduct.
## Here we calculate the mean for each column of brambles
result <- numeric(NCOL(brambles))

for(i in 1:NCOL(brambles)){
    result[i] <- mean(brambles[,i])
}

result
```

```
## [1] 0.5478202 0.5042539 0.6597813
```

Often for-loops could be replaced with simpler constructs. Compare for example the following statements:

```
sapply(brambles,mean)
```

```
##         x         y       age
## 0.5478202 0.5042539 0.6597813
```

```
colMeans(brambles)
```

```
##         x         y       age
## 0.5478202 0.5042539 0.6597813
```

None the less, for loops are often useful more complex tasks, and a fundamental programming skill.

Creating your own functions

Sometimes the existing functions of R do not quite do what you would want them to do. Often a handy solution is to define your own functions. Functions are created like all other objects: with the assignment operator. After the arrow, you write `function(arguments)`, where `arguments` are any arguments your function needs to function.

```
## A simple function definition
my_function <- function(x) {print(x)}
my_function(3)
```

```
## [1] 3
```

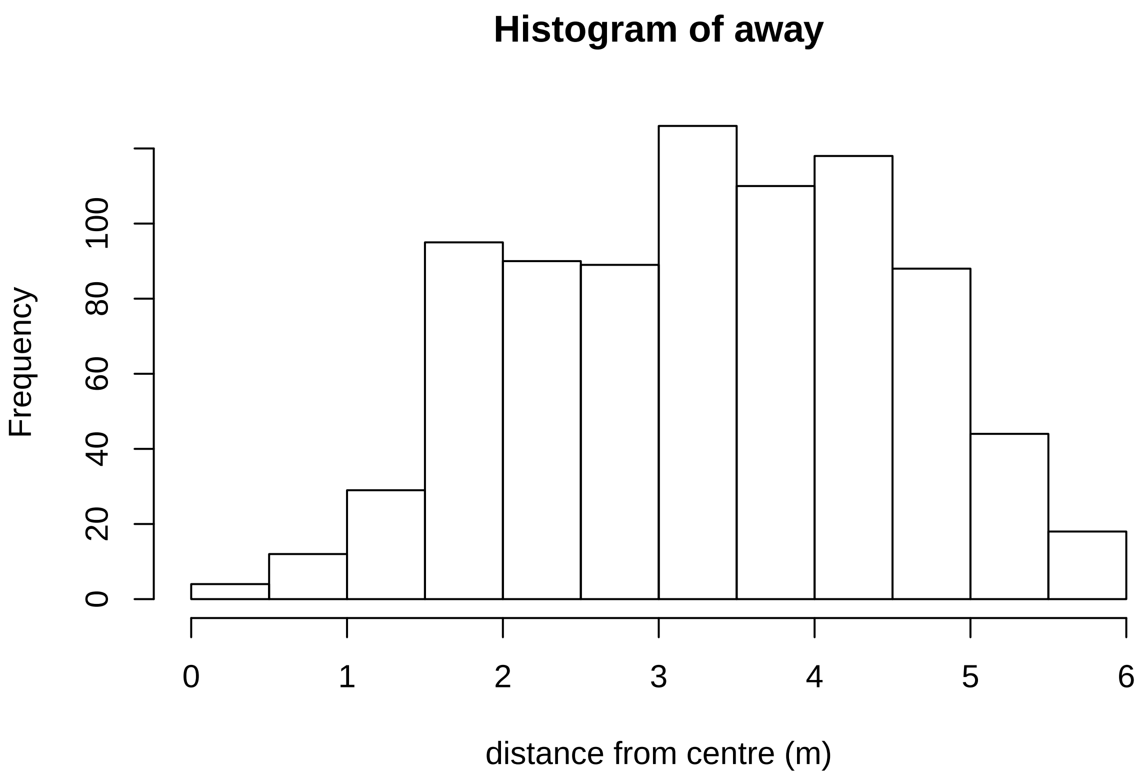
Let us make a function to calculate how far our bramble observations are from the centre of our study plot. The centre coordinate of our study plot was (0.5,0.5), but we include it as an argument so that our function would be more general. We also include a scale parameter, since the coordinates of our data were originally from 9 m × 9 m squares. This allows us to convert the distances to metres.

```
away_from_centre <- function(x, y, centre = 0.5, scale = 9){
    distance <- sqrt((x-centre)^2 + (y-centre)^2)
    distance <- distance*scale
    return(distance)
}
```

The syntax of a function definition is similar to that of for-loops and if-else statements: the curly brackets delimit which expressions belong to the function. The last expression is the command `return()`. This defines which results are returned when the function is run.

We can now run the function on our `brambles`.

```
## centre and scale are defined in the function definition
## We do not need to explicitly set them if we are fine with the defaults
away <- away_from_centre(x=brambles$x, y = brambles$y)
hist(away, xlab = "distance from centre (m)") # distribution of distances from the centre
```


Histogram of away
Frequency
distance from centre (m)