INTERFACULTY CENTER
DATA PROCESSING & STATISTICS
CORE FACILITY

An R Primer

preparing for data manipulation and visualization

w. cools

Compiled Jun 04, 2020 (R 3.6.2)

Learning R will reward the researcher with the ability to read, manipulate, summarize and visualize data with tremendous flexibility, for free, anywhere and anytime. R gives access to an enormous range of statistical techniques made available through R packages, with a huge R community online to help out. The open source programming language R is more than just statistics; packages support reproducible research (eg., markdown), web apps (eg., shiny), databases (eg., SQL), interaction with other programming languages (eg., Python), cloud computing and more.

With huge flexibility comes difficult first steps…

Current draft aims to introduce researchers to some very basic R features, in order to ensure a minimal level of proficiency to further study data manipulation with the `dplyr` package and visualization with the `ggplot` package from the `tidyverse` ecosystem.

There are many sources online, including tutorials, Q&A, cheat sheets, collections of useful code, for example the section on Statistics and Advanced Statistics at https://www.statmethods.net/, that go well beyond the introduction here.

Our target audience is primarily the research community at VUB / UZ Brussel, those who have a keen interest to start studying R, or refresh their basic understanding, and especially those who aim to study data manipulation and visualization with `tidyverse`.

We invite you to help improve this document by sending us feedback
wilfried.cools@vub.be or anonymously at icds.be/consulting (right side, bottom)

Exemplary Analysis

Before going into any detail, a simple analysis will show you where you are heading towards.

Let's use a dataset that is included into R, the `mtcars`, and make it available in my R workspace with the `data()` function. You will read in your own data in future.

```
data(mtcars)
```

The first 6 lines of the dataset is shown to give an idea about the variables.

```
head(mtcars)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

A regression analysis of the `mpg` variable (dependent) on the `am` variable (independent) can be performed with the `lm()` function which stands for linear model. The independent variable `am` could be treated as a factor with two levels, `0` and `1` (see later) or as numeric with values 0 and 1, but because there are only two values the results will be the same. The result is assigned to the R object `myResult`.

```
myResult <- lm(mpg~am,data=mtcars)
```

To show the result, request the R object.

```
myResult
```

```
##
## Call:
## lm(formula = mpg ~ am, data = mtcars)
##
## Coefficients:
## (Intercept)           am
##      17.147        7.245
```

To request a summary, use the `summary()` function on the R object.

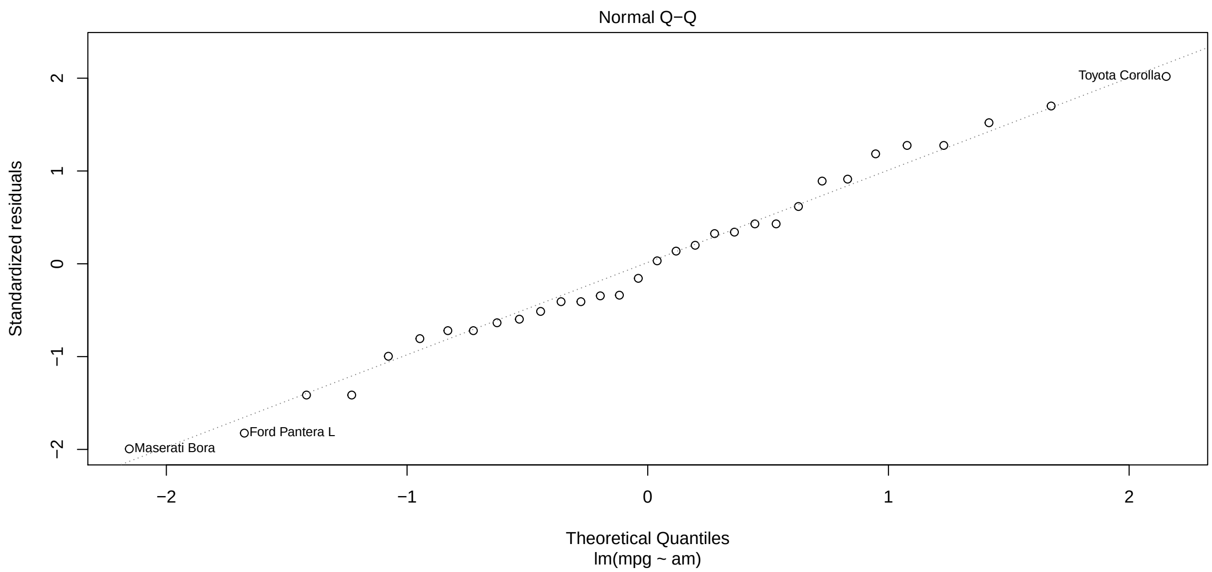
```
summary(myResult)
```

```
##
## Call:
## lm(formula = mpg ~ am, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -9.3923 -3.0923 -0.2974  3.2439  9.5077
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)   17.147      1.125  15.247 1.13e-15 ***
## am             7.245      1.764   4.106 0.000285 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

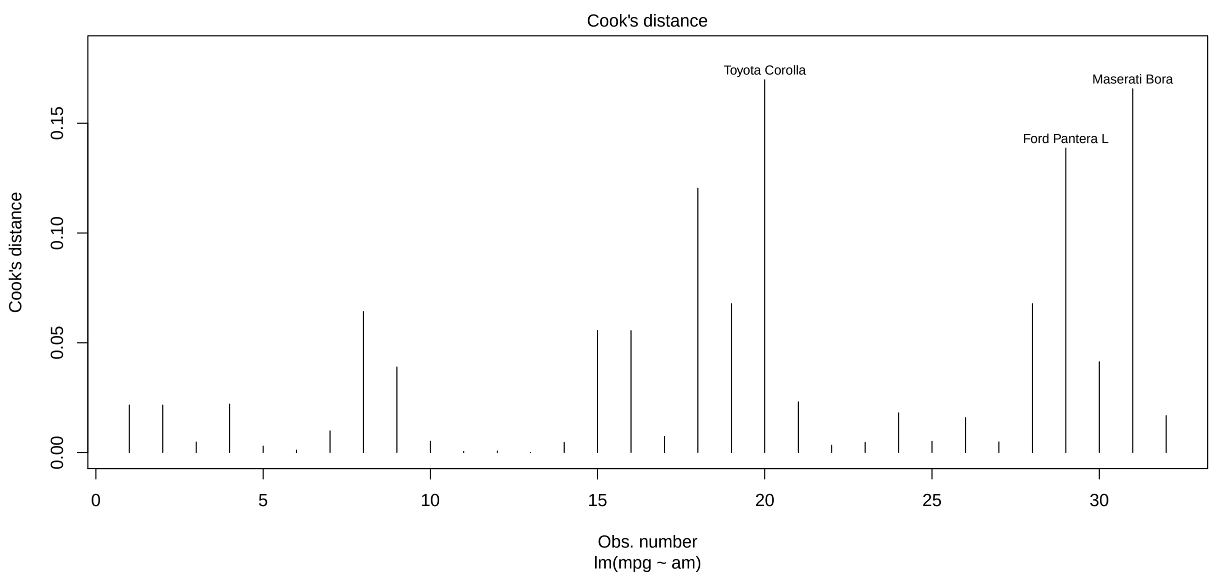
```
## Residual standard error: 4.902 on 30 degrees of freedom
## Multiple R-squared:  0.3598, Adjusted R-squared:  0.3385
## F-statistic: 16.86 on 1 and 30 DF,  p-value: 0.000285
```

Various plots are offered by default using the `plot()` function, let's consider the qqplot and the influence measured by Cook's distance.

```r
plot(myResult,2)
```




```r
plot(myResult,4)
```



The R object that represents the results of an `lm()` call (=regression) contains much more information, for example the r-squared.

```
summary(myResult)$r.squared
```

```
## [1] 0.3597989
```

In this case, with a continuous dependent variable and a categorical independent an equivalent analysis would be ANOVA which can be performed with the `aov()` function in which the `am` is automatically treated as a factor.

```
summary(aov(mpg~am,data=mtcars))
```

```
##             Df Sum Sq Mean Sq F value   Pr(>F)
## am           1  405.2   405.2   16.86 0.000285 ***
## Residuals   30  720.9    24.0
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

The t-value when squared gives an F-value, so you can verify, both the ANOVA and regression offer exactly the same evidence for group differences in `mpg` for `am` equal to `0` or `1`.

Interaction with R

R commands can be entered in the R console for interpretation.

- Enter commands after the prompt, typically `>` to start and `+` to proceed.

- `enter` to make R interpret your command. `ESC` to exit a command.
- `Arrow Up` (Down) to request previous (next) command. The function `history()` shows earlier commands.

- `Tab` to complete the name of an R object if uniquely identifiable. `Tab` again to list possible names of R object if not uniquely identifiable.

R scripts combine commands for current and future use, they can be flushed to the R console.

- from the script window in **RGui** (basic)
- from source code editors, eg., **Notepad++** (general purpose)
- integrated development environments: eg., **RStudio** (standard / recommended)

R workspace

An R workspace is a working environment for a user to interact with, that includes all R objects a user makes or imports.

An R workspace is linked to a directory on your system, the working directory, for its input and output. Retrieve the working directory:

```
getwd()
```

Check what is in that working directory already:

```
dir()
```

Set a working directory by its directory path, use forward slashes (or double backward \\ because \ is an escape character).

```
setwd('C:/Users/.../Documents/')
```

From the working directory it is straightforward to include:

- objects from an R workspace using `load()` (eg., `load(mydta.RData)`)
- R code with variables and/or functions to execute using `source()` (eg., `source('myprog.r')`)
- functions from installed R packages using `library()` or `require()` (see below)
- data from text or other files (see below)

An R workspace offers over 1000 functions and operators combined into the package `base`.
Include dedicated functions by loading in appropriate additional packages when necessary.

To include all functions related to the `tidyverse` package, at least once install the packages of interest, and occasionally update them.

```
install.packages('tidyverse')
```

Every time a workspace is opened, all relevant packages should be included.

```
library(tidyverse)
```

Check which packages are loaded:

```
search()
```

```
##  [1] ".GlobalEnv"        "package:readxl"    "package:forcats"   "package:stringr"
##  [5] "package:dplyr"     "package:purrr"     "package:readr"     "package:tidyr"
##  [9] "package:tibble"    "package:ggplot2"   "package:tidyverse" "package:knitr"
## [13] "package:stats"     "package:graphics"  "package:grDevices" "package:utils"
## [17] "package:datasets"  "package:methods"   "Autoloads"         "package:base"
```

To get help on how to use functions, eg., `read.delim()`, call them with `?read_delim`.
To get help on how to use packages, eg., `tidyr`, call `help(package='tidyr')`.

R objects

An R workspace contains R objects which can be data as well as methods. Each object is of a certain class which defines the object and how it is handled.

Check the objects currently in your workspace:

```
ls()
```

```
## [1] "mtcars"   "myResult"
```

Check the class of an object.

```
class(mtcars)
```

```
## [1] "data.frame"
```

The data are represented by a dataframe.

```
class(myResult)
```

```
## [1] "lm"
```

The results of an `lm` analysis are represented by an lm-object.

```r
class(plot)
```

```
## [1] "function"
```

The plot function is represented as a function-object.

Objects are defined by various attributes and methods. The content of an object, its structure, can be extracted. This `str()` function is very very convenient, just to make sure what the object is.

Check the structure of an object.

```r
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

```r
str(plot)
```

```
## function (x, y, ...)
```

Some objects consist of many different pieces of information. Extract the names of the attributes.

```r
attr(myResult,'names')
```

```
##  [1] "coefficients"  "residuals"     "effects"       "rank"          "fitted.values"
##  [6] "assign"        "qr"            "df.residual"   "xlevels"       "call"
## [11] "terms"         "model"
```

Check a particular piece of information in an object using these names following the `$` sign (to extract a slot from a list).

```r
myResult$call
```

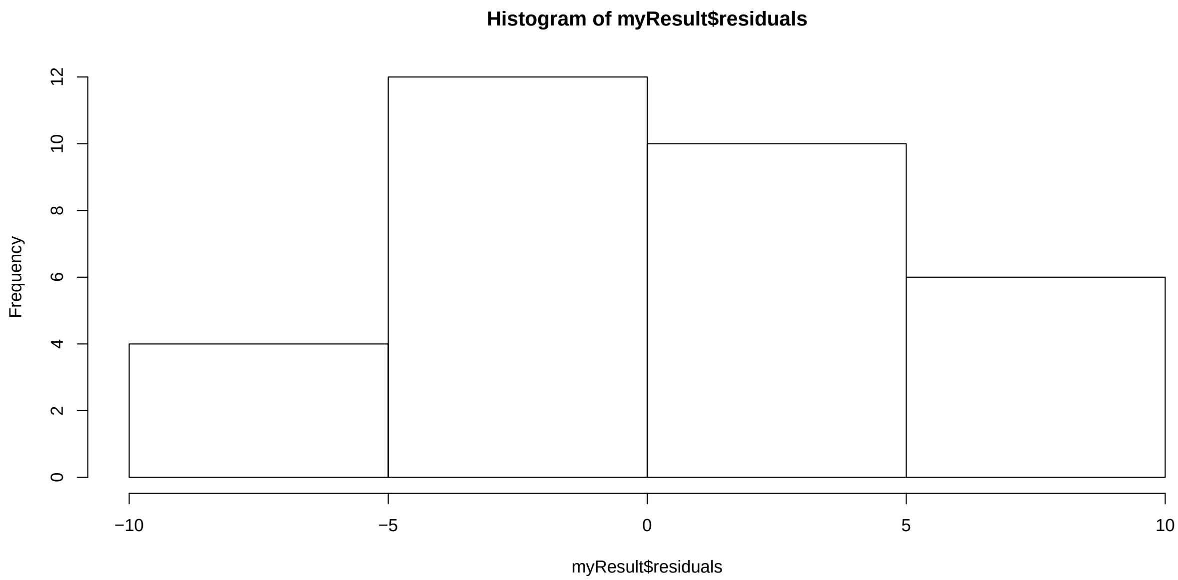
```
## lm(formula = mpg ~ am, data = mtcars)
```

```r
myResult$contrasts
```

```
## NULL
```

Or simply use the content directly, for example making a histogram with `hist()` of the residuals that remain after a regression analysis.

```r
hist(myResult$residuals)
```



Create R objects

R objects are created by assigning values using the `<-` assignment operator. Equivalently, the `assign()` function can be used (eg., assign('myNewObject',c(1:10)))

Represents the directory path with a string, a character variable named myDirPath. (Be aware that `=` is equivalent to `<-`, equality is denoted `==`.)

```
myDirPath <- 'C:/Users/.../Documents/'
class(myDirPath)
```

```
## [1] "character"
```

Once established, the R object can be used by any function able to deal with that type of object.

```
setwd(myDirPath) # set working directory to the given directory path
```

This way also the `myResult` object was created, with class lm.

An R object can be removed, which is especially of interest when it uses a lot of memory.

```
rm(myDirPath)
```

A few R objects are created to exemplify. To show the result of an assignment is should either be called explicitly, or the assignment should be made in brackets.

```
a <- c('a vector of 2',1)
a
```

```
## [1] "a vector of 2" "1"
```

```
class(a)     # a character vector (number is turned into character)
```

```
## [1] "character"
```

```
(b <- c(2,NA,3))
```

```
## [1]  2 NA  3
```

```
class(b)     # a numeric vector
```

```
## [1] "numeric"
```

```
(mx <- matrix(1:12,nrow=3)) # a matrix
```

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

```
class(mx)    # matrix of numbers
```

```
## [1] "matrix"
```

```
(mydataframe <- data.frame(y=runif(10),x=rep(c(1,2),each=5)))
```

```
##            y x
## 1  0.3418390 1
## 2  0.7129452 1
## 3  0.8780568 1
## 4  0.1293030 1
## 5  0.9996421 1
## 6  0.8101368 2
## 7  0.5557797 2
## 8  0.2854117 2
## 9  0.8653330 2
## 10 0.3435603 2
```

```
class(mydataframe)
```

```
## [1] "data.frame"
```

Dataframes: R data object for analysis

Typically data are stored as a dataframe in R, which is a list of equally sized vectors. A vector is a combination of elements of a particular type.

Dataframes are similar to matrices, but more flexible because each vector can be of a different type. Roughly speaking, the first column could consist of characters, the second of numbers, the third of booleans, … all of the same size.

An exemplary dataframe is created from vectors of size 4:

```
mydataframe <- data.frame(a=1:4,b=c(T,FALSE,T,TRUE),c=c('a','b','a','a'),d=c(1.2,NA,1.5,.2),
e=c(1.2,T,.3,NA),f=c(1.2,'>5',NA,.2),g=c(T,FALSE,'true','?'))
```

Notice that the dataframe consists of column a to g.

```
mydataframe
```

```
##   a     b c   d   e    f     g
## 1 1  TRUE a 1.2 1.2  1.2  TRUE
```

```
## 2 2 FALSE b  NA 1.0   >5 FALSE
## 3 3  TRUE a 1.5 0.3 <NA>  true
## 4 4  TRUE a 0.2  NA  0.2     ?
```

```r
str(mydataframe)
```

```
## 'data.frame':    4 obs. of  7 variables:
##  $ a: int  1 2 3 4
##  $ b: logi  TRUE FALSE TRUE TRUE
##  $ c: Factor w/ 2 levels "a","b": 1 2 1 1
##  $ d: num  1.2 NA 1.5 0.2
##  $ e: num  1.2 1 0.3 NA
##  $ f: Factor w/ 3 levels ">5","0.2","1.2": 3 1 NA 2
##  $ g: Factor w/ 4 levels "?","FALSE","true",..: 4 2 3 1
```

The different vectors within the dataframe:

- a: integer (int)
- b: logical (with TRUE or FALSE), T and F are converted into booleans TRUE and FALSE
- c: factor, character type labels are converted into a factor with two levels
- d: numeric, double values with NA for a missing value which does not change the type
- e: numeric, the T is interpreted as TRUE which is a boolean which is converted into a 1
- f: factor, the >5 can not be interpreted as numeric/boolean, all values are turned into characters and converted to factors
- g: factor, the ? and true can not be interpreted as numeric/boolean, all values are turned into characters and converted to factors

To avoid characters converted into factors, add the stringsAsFactors=FALSE argument:

```r
mydataframe2 <- data.frame(c=c('a','b','a','a'),f=c(1.2,'>5',NA,.2),g=c(T,FALSE,'true','?'),
    stringsAsFactors=FALSE)
str(mydataframe2)
```

```
## 'data.frame':    4 obs. of  3 variables:
##  $ c: chr  "a" "b" "a" "a"
##  $ f: chr  "1.2" ">5" NA "0.2"
##  $ g: chr  "TRUE" "FALSE" "true" "?"
```

Notice, characters are quoted.

Extract the vector from a dataframe, using the `$` operator and a name, the list selector `[[ ]]` with either a name or a number for the position.

```r
mydataframe$a    # select vector named `a`
```

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

```r
mydataframe[['a']]  # select vector named `a`
```

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

```r
mydataframe[[1]]     # select vector at position 1
```

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

```r
mydataframe[,1] # treats dataframe as if matrix, select column 1
```

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

```r
mydataframe[,'a']   # treats dataframe as if matrix, select column 'a'
```

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

The class is now of type vector, in this case numeric:

```r
class(mydataframe$a)
```

```
## [1] "integer"
```

Extract a row from a dataframe, using either the row name or a number of the position.

The row names can be consulted.

```r
row.names(mydataframe)
```

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

These row names can also be altered.

```r
row.names(mydataframe) <- c('row.1','row.2','row.3','row.4')
```

To extract a row it is necessary to treat it like a matrix, with a position or name in front of the comma (after the comma is for columns).

```r
mydataframe[4,] # treats dataframe as if matrix, select row 4
```

```
##       a    b c   d  e   f g
## row.4 4 TRUE a 0.2 NA 0.2 ?
```

```r
mydataframe[c('row.1','row.4'),] # treats dataframe as if matrix, select rows by name
```

```
##       a    b c   d   e   f    g
## row.1 1 TRUE a 1.2 1.2 1.2 TRUE
## row.4 4 TRUE a 0.2  NA 0.2    ?
```

Read dataframes

Various ways exist to read data into R.

In base R, read in a tab-delimited text file with header:

```r
dta <- read.table('RealData_clean.txt',sep='\t',header=T)
class(dta)
```

```
## [1] "data.frame"
```

Instead of tabs as separator (sep=‘’), other symbols can be used, like comma’s (sep=‘,’). For other options consult the help file:

```r
?read.table
```

Packages with dedicated read-and-write functions, like the `foreign` package, provide more functions for reading in data from SAS, Stata, SPSS, and more.

```r
library(foreign)         # load in functions in package foreign
help(package='foreign') # consult the help file for package foreign
read.spss("RealData_clean.sav",to.data.frame=T) # read sav file
```

Excel is notoriously error prone but popular. A package dedicated to reading in data from Excel is `readxl`.

```
library(readxl)
# read xlsx file and assign it to mydtaXls object
mydtaXls <- read_excel("RealData_clean.xlsx",sheet="datafile")
class(mydtaXls) # consult class of mydtaXls object
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

The `tidyverse` ecosystem includes the `readr` package, also good for reading in data. Check out the arguments with `?read_delim`. Different types of files can be read in with different delimiters. Note that also a copy-paste is possible, an example in `readr` is included.

```
# library(readr) # or tidyverse
mydtaCsv <- read_delim('RealData_clean.csv',delim=';')
mydtaTxt <- read_delim('RealData_clean.txt',delim='\t')
mydtaClp <- read_delim(clipboard(),delim='\t')
```

For each function, many arguments are possible.
More complex and versatile tools exist, also for XML, relational databases, unstructured data, …

Data types and structures in R

Dataframes are lists of vectors with a particular type that should be assigned appropriately.

From the `mydtaXls` file the 5th to 16th row is selected (before the comma in brackets), for the columns 1 to 3, 6, 7, 10 and 11 (after the comma in brackets).

```
rows <- c(1:3,6,8,10,186:189)
(dta <- mydtaXls[rows,c(1:3,6,7,10:11)])
```

```
## # A tibble: 10 x 7
##    Pte          Dx         P `Size mm (FMT,VZ, QRT)`   `FMT (mm)` `Measurement in end~ CL
##    <chr>        <chr> <dbl> <chr>                          <dbl> <chr>                <chr>
##  1 1            TB        3 VZ 24 (vierling)                  NA Y                    ?
##  2 2            TB        1 FMT 13                            13 Y/N                  ?
##  3 3            FLIN      1 VZ 23                             NA Y                    1
##  4 6            TB        3 FMT 6                              6 N                    ?
##  5 8            TB        2 FMT 8                              8 Y                    1
##  6 10           TB        4 VZ 45                             NA Y                    1
##  7 186          PRR       1 infected heterotopic abo~         NA ?                    0
##  8 186 non ev~ <NA>      NA <NA>                              NA <NA>                 <NA>
##  9 PUL evol     <NA>      NA <NA>                              NA <NA>                 <NA>
## 10 1            PRR       NA 0                                 NA ?                    ?
```

The extremely useful `str()` function extracts the structure of the R object. Use it!!

```
str(dta)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    10 obs. of  7 variables:
##  $ Pte                    : chr  "1" "2" "3" "6" ...
##  $ Dx                     : chr  "TB" "TB" "FLIN" "TB" ...
##  $ P                      : num  3 1 1 3 2 4 1 NA NA NA
##  $ Size mm (FMT,VZ, QRT): chr  "VZ 24 (vierling)" "FMT 13" "VZ 23" "FMT 6" ...
##  $ FMT (mm)               : num  NA 13 NA 6 8 NA NA NA NA NA
```

```
##  $ Measurement in endom : chr  "Y" "Y/N" "Y" "N" ...
##  $ CL                   : chr  "?" "?" "1" "?" ...
```

The classes in this case are tbl_df, tbl, data.frame. When `tidyverse` is loaded, dataframes are automatically assigned the extra class `tbl_df` and `tbl` with extra functionality (see future).

In the example the numeric `P` is reduced by 2, and then turned into a factor.
The `Dx` variable which is of type character is made a factor, which has levels.
The `CL` variable is made an ordered factor, with levels 'yes', 'maybe' and 'no' using the data 1, ? and 0.
The `CL` variable which is now a factor is turned into a numeric again, first making it a character vector (tricky issue: is to avoid obtaining the level ranks) and then making it a number.
The `FMT (mm)` variable is then assigned to the not yet existing `fmt` variable. Notice the `‘, these quotes are required because the variable name contains a space. Finally, the`FMT (mm)‘ is removed, simply by assigning it the NULL value.

```
dta$P <- factor(dta$P-2)
levels(dta$P)
```

```
## [1] "-1" "0"  "1"  "2"
```

```
dta$Dx <- factor(dta$Dx)
levels(dta$Dx)
```

```
## [1] "FLIN" "PRR"  "TB"
```

```
dta$newCL <- factor(dta[,'CL'],ordered=TRUE,levels=c('1','?','0'),labels=c('yes','maybe','no'))
dta[,'CL'] <- factor(dta[,'CL'],levels=c('1','0'))
dta$numCL <- as.numeric(as.character(dta$CL))
dta$fmt <- dta$`FMT (mm)`
dta$`FMT (mm)` <- NULL
```

The resulting structure is:

```
str(dta[,c('P','Dx','newCL','CL','numCL','fmt')])
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    10 obs. of  6 variables:
##  $ P    : Factor w/ 4 levels "-1","0","1","2": 3 1 1 3 2 4 1 NA NA NA
##  $ Dx   : Factor w/ 3 levels "FLIN","PRR","TB": 3 3 1 3 3 3 2 NA NA 2
##  $ newCL: Ord.factor w/ 3 levels "yes"<"maybe"<..: NA NA NA NA NA NA NA NA NA NA
##  $ CL   : Factor w/ 2 levels "1","0": NA NA NA NA NA NA NA NA NA NA
##  $ numCL: num  NA NA NA NA NA NA NA NA NA NA
##  $ fmt  : num  NA 13 NA 6 8 NA NA NA NA NA
```

ADVICE: first always check structure, change where necessary

Applying functions

Functions consist of code that you can execute, to process your data for example. Technically, R functions are also R objects.

A few simple functions are `summary()` and `table()`.

```
summary(dta$P)
```

```
##   -1    0    1    2 NA's
##    3    1    2    1    3
```

```
summary(as.numeric(dta$P))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
##   1.000   1.000   2.000   2.143   3.000   4.000       3
```

```
summary(as.numeric(as.character(dta$P)))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's 
## -1.0000 -1.0000  0.0000  0.1429  1.0000  2.0000       3
```

```
table(dta$P,useNA='always')
```

```
## 
##   -1    0    1    2 <NA> 
##    3    1    2    1    3
```

```
table(as.numeric(dta$P),useNA='always')
```

```
## 
##    1    2    3    4 <NA> 
##    3    1    2    1    3
```

A summary taken from a factor offers a frequency table.
A summary of a factor that is converted to numeric offers a few statistics (extrema, mean, …) calculated on the level ranks. This is probably not what is intended.
A summary of a factor that is converted first to a character, then to numeric will correctly offer the statistics of the actual values.
A table would offer a frequency table. Always be cautious when working with factors, they behave in sometimes complex ways.

ADVICE: first always check summaries and tables, change where necessary in order to detect anomalies

Almost all statistical analyses in R will use functions others created for you to execute on your data. Examples have been the `lm()`, `aov()`, and `plot()`.

Creating functions

Instead of using existing functions, it is possible to create your own. For simple data analysis this is rarely necessary but some basic understanding will probably help you understand how to use functions better.

A function in R has the structure `function(arglist) {body}`.

The body contains a set of instructions that are executed each time the function is called. The list of arguments bring in the information required for the function to execute.

A function that returns a random standard normal value could be:

```
myRandomNormalValue <- function(){ return(rnorm(1)) }
myRandomNormalValue()
```

```
## [1] 1.215895
```

```
myRandomNormalValue()
```

```
## [1] -1.121834
```

Each call of the function will execute it.

A function that returns the value to the power 2 could be:

```
myValueToPower2 <- function(val){ return(val^2) }
myValueToPower2(4)
```

```
## [1] 16
```

```
myValueToPower2(6)
```

```
## [1] 36
```

An assignment of a value to the argument sets the default value.

```
myValueToPower2 <- function(val=10){ return(val^2) }
myValueToPower2(4)
```

```
## [1] 16
```

```
myValueToPower2()
```

```
## [1] 100
```

It is possible to use the functions as if they are the return value.

```
myValueToPower2() + myValueToPower2(8) / myValueToPower2(2)
```

```
## [1] 116
```

Functions will become more important when chunks of code are repeated many times.

Control Structures

It is possible to execute code conditionally. For simple data analysis this is rarely necessary but some basic understanding will probably help you understand other researcher's code.

Execute code dependent on a condition being true.

```
if(rnorm(1)>0){ cat('the generated value was above 0\n') }
```

Execute code dependent on a condition being true and other code when false.

```
ifelse(rnorm(1)>0,'above','below')
```

```
## [1] "above"
```

Execute code multiple times, using an indicator variable.

```
for(it in 1:10){
    if(rnorm(1)>0){ cat('for it equal to',it,'the generated value was above 0\n') }
}
```

```
## for it equal to 1 the generated value was above 0
## for it equal to 3 the generated value was above 0
## for it equal to 4 the generated value was above 0
## for it equal to 6 the generated value was above 0
## for it equal to 7 the generated value was above 0
## for it equal to 8 the generated value was above 0
```

Execute a code as long as a condition holds.

```
it <- 3
while(it > 0){
    cat('it =',it,'\n')
    it <- it-1
}
```

```
## it = 3
## it = 2
## it = 1
```

Still other control structures exist.

R help files

The `?` should give basic information on functions and how to use them.

To open the R help file, use the `?` operator and the name of the function without brackets.

```
?paste
```

A help file in R consists of:

- reference to package it belongs to (eg., base, which is always loaded automatically)
- description
- usage with required arguments and their default value if any
- arguments with additional information on the arguments
- details that could be of interest
- value with information on the result
- references for further information
- see also for highlighting similar, or related functions
- examples to show its use

From the help file for `paste()`:

- paste works on r objects that can be converted to character
- a separator is put in between, by default a single space.
- these values by default are not collapsed

For example, with a first object a numeric vector of 4 elements and a second object a character vector of 2 elements.

```
paste(c(1:4),c("a","b"))     # turn into characters combinations with default separator
```

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

```
paste(c(1:4),c("a","b"),sep='-')     # turn into characters combinations with dash separator
```

```
## [1] "1-a" "2-b" "3-a" "4-b"
```

```
paste(c(1:4),c("a","b"),sep='-',collapse='/')    # ~ and collapse with slash into one
```

```
## [1] "1-a/2-b/3-a/4-b"
```

More more examples are, in this case, provided at the help page.

wilfried.cools@vub.be