Introduction to R

Dr. Federico Karagulian

Masdar Institute

13-14 August 2017

Contents

1. Downloading and installing R

The R (http://www.r-project.org/) website is not very fancy and does not reflect the power and capabilities of R. R software is available from servers all over the world (mostly hosted by universities). This network of servers is known as the *Comprehensive R Archive Network* (CRAN). The CRAN hosts both the basic R software and all the packages (now over 7,000 of them). The basic R software comes with 12 or so base packages (and a similar number of recommended packages). This basic installation is highly capable e.g. includes lots of tools for data analysis, statistics and plotting.

To download R visit the R website (http://www.r-project.org/) and on the left you will see a link to CRAN. You will then be presented with a list of locations where R is available. It doesn't really matter which you choose but scrolling down to the Denmark site makes sense.

Once chosen at the top of the page you can select which operating system you want to install R on — choose the appropriate one. You will then want to install the base system e.g. something like

```
Download R 3.2.3 for Windows (62 megabytes, 32/64 bit).
```

Once downloaded you can install the software in the usual way for your system (a .exe file for Windows, a .pkg file for Mac and the appropriate linux-flavour method).

New versions of R tend to be released every 6-12 months and it is *always* a good idea to use the most recent version.

Note, each of these installations comes with a simple GUI or command line tool, which is fine to use for many purposes. However, the preferred approach is to also install software called RStudio, described next.

2. Introduction to Rstudio (http://www.rstudio.com/)

Rstudio (http://www.rstudio.com/) is an *Integrated Development Environment* that acts as an excellent one-stop shop to do everything R-related. It is free and high quality and works on Windows, Mac OSX and linux. As a minimum it acts

as a nice “front end” to using R. You should install R first and then install RStudio (the latter should find the former).

> It is highly recommended that you learn how to use RStudio

Elements to cover in RStudio include:

- Layout of the panels, what each does
- The R Console This is where you can type (or paste) commands directly for R to perform.
- The R Source pane This is where you can record your R commands to form an analysis from start to finish.
- Installing and loading packages
- Updating packages
- Getting help
- History and Environment

2.1. Making an RStudio project

RStudio will work happily with individual R scripts (source code for a particular analysis). These are simple txt files with a .R extension. However, for ‘real world’ analyses e.g. all the analysis that might be needed for a journal article, it is useful to group all the scripts, data etc. in a single project.

3. Getting help

There are many ways of getting help with R. R itself and all packages are documented. If you want to know more about how to use a function you can type (for example) `?plot` . The RStudio team have put together a good summary of where to get help (https://support.rstudio.com/hc/en-us/articles/200552336-Getting-Help-with-R), which is worth looking at.

The best source of help though is probably *Stack Overflow* (http://stackoverflow.com/questions/tagged/r). This website allows you to ask questions about R and various R experts from around the world can answer them.

The clever approach of Stack Overflow is to allow people to ‘up-vote’ the best answers. For common questions it is usually very easy to see what the best solution is. Some of the best answers are very good and almost all are

reproducible and well-explained. It is a very good idea to think clearly about the questions you ask and provide (or generate) data so others can test the solution and show it works. Often the very act of going through how to ask a question helps to reveal the answer. More often than not a search of answers reveals solutions that already exist, as people often get stuck on similar problems.

4. General approach to data analysis using R/RStudio

4.1. 5.1 Use scripts!

Scripts are simple text files (with a .R extension) that store all your commands for a particular analysis. For example, I often have one or more scripts associated with with a paper I am writing. Here is some advice for using scripts:

- Get used to using scripts to keep a record of all your analyses.
- Scripts are reproducible. If you write them clearly you can go back to them after a year and reproduce all your analysis.
- Make sure you take the time to add comments (lots) to remind you what is going on. You can use the hash command # to do this.
- Scripts can easily be shared.
- Scripts can form the basis of developing functions to carry out repetitive tasks — and eventually form the basis of an R package.

> It is possible to save all objects in the current R session (called a workspace) as an .RData file. This can be very useful to store everything in a compact single file that can then be loaded later.

In RStudio, the the Environment tab you can view all the objects in your workspace. If you select Save, it will save all these objects in a single file (.RData) file. Saving your workspace can be useful especially if you are working on something that you keep coming back to.

However, I tend not to save my workspace and rely instead on using scripts to recreate everything I need, as this approach is more reliable/reproducible. There are a few exceptions though. If you work with large data sets or data that takes a long time to create using scripts it can be a good idea to save

your workspace. For these sorts of situation it can be more useful to save specific data objects e.g.

```
save(bigdata1, bigdata2, file = "./processed/MyBigData.RData")
```

where `bigdata1` and `bigdata2` are large data sets. It is then possible when you want to work on these data sets to load them:

```
load("./processed/MyBigData.RData")
```

Another advantage of saving data like this is that the data objects are compressed and can be much smaller than csv, Excel, txt files etc.

4.2. Leave the data alone (as much as you can)

This is a very important issue. Often data can be acquired from instruments, the Internet, co-workers etc. and it can be tempting to try and manually process the file(s) in various ways e.g. in Excel. Sometimes it is very difficult not to edit the data manually in some way. However, this can be dangerous and changes the original data.

> It is much better to keep data in its original format and do all the processing in R. By using scripts to do this, each step is entirely reproducible. It also means if new data becomes available you will not need to manually adjust it — just run the script (or modify it)

4.3. Use relative paths

Information on computers (particularly networked servers) can be stored in a very complex hierarchy of directories e.g. “C:/Users/David/Research/Projects/Denmark/Course/R/2015/data/…”.

It is of course possible to always refer to the full path for the location of some information such as that above. However, it is far better to use relative paths (and get used to how to use them).

In RStudio you can make a new project and set the *working directory* to the location of the project. What I tend to do is at the beginning of a script is set the working directory e.g.

```
setwd("~/Dropbox/masdar_data/")
mydata <- read.csv("mydata.csv")
```

It means I can refer to that location without typing in the file path. For example I can typing any path at all…

> The tilde character ~ refers to your ***home directory***. On Windows machines for example it might be something like "C:/Users/David/", on Mac OS it might be something like /Users/David/

If we need to read some data from the sub directory of "AarhusCourse" (data) we could refer to it in several ways e.g.

```
"C:/Users/David/Dropbox/masdar/data/"
"~/Dropbox/masdar/data/"
```

However, having set the working directory to "~/Dropbox/masdar/" we can make use of the *relative path* using the dot `.` . The dot means relative to the current directory:

```
"./data/"
```

So to read in a csv file at that location the command will become something like `mydata <- read.csv("./data/mydata.csv")` .

This approach has a couple of key advantages:

- It makes your work much more transferable to other systems. For example if you move your work to another computer or shared it with a colleague. In this case you only need to set the new working directory and all sub-directories will have the same relative paths.
- It makes the code much less verbose and easier to read.

One further recommendation is that it is a good idea to get into the habit of saving file names (and directories) without spaces.

5. Simple R and vectors

5.1. Basic R use

First, a few examples of working with single numbers in the console.

```
# addition
4 + 4
```

```
## [1] 8
```

```
# subtraction
8 - 4
```

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

```
# division
4 / 3
```

```
## [1] 1.333333
```

```
# multiplication
5 * 10
```

```
## [1] 50
```

```
# exponentiation
5 ^ 5
```

```
## [1] 3125
```

```
# modulo - remainder
5 %% 3
```

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

5.2. Variable assignment

Can assign different quantities to variables. Note that traditionally the assignment operator `<-` is used in R, which is recommended. However, the `=` is also acceptable in almost all situations.

```
x <- 5
y <- 6
x + y
```

```
## [1] 11
```

Note! R is case sensitive, so `x` not `X` etc.

5.3. Logical operators

Useful for many purposes of course, but sub-setting data in particular

```
< # less than
<= # less than or equal to
> # more than
>= # more than or equal to
== # exactly
equal to != #
not equal to
!x # not x
x | y # x
OR y x &
y # x AND
y
```

5.4. Vectors in R

A vector is a simple array of numbers or characters, such as the measurements of a single variable. R makes it easy to carry out mathematical operations and functions to all the values in a vector at once. This is one of the most important concepts in R.

Take a sequence of numbers `1, 2, 5, 8` . How do we represent these numbers by inputting then directly in R?

It is necessary to use the `c` function (concatenate) to combine them into a single vector:

```
x <- c(1, 2, 5, 8) # define x
x # prints values of x to the screen
```

```
## [1] 1 2 5 8
```

How can we add 5 to each of them? In most programming languages we need to loop through them, something like:

```
x <- c(1, 2, 5, 8) # define x
for (i = 1 to 4)
  x[i] = x[i] + 5
next
```

In R we can just work with all of x at once

```
x <- c(1, 2, 5, 8) # define x
x <- x + 5 # add 5 to all elements of x
```

```
## [1]  6  7 10 13
```

So 5 is added to all elements of x in a simple way. This might seem odd because x is 4 integers and the 5 is only one integer. We are adding a vector of length 4 to a vector of length 1. However, in R, the 5 is *recycled* for the length of x. This can seem strange compared with other programming languages but it makes for rapid coding — and
coding that is easy to read. Think of it in terms of adding a value to a 'whole' vector. It also mean transforming and working with variables is more akin to general algebra rather than computer code (e.g. we don't need a for loop to work with a vector every time we want to do something with it).

This makes for clean code but also other powerful possibilities.

> Practice typing some vectors into R, adding, multiplying them etc. to get a feel for how they work

5.5. Selecting subsets of vectors

```
x <-  c(1, 4, 5, 18, 22, 3, 10, 33, -2, 0)
# to select the 4th element
# use the square brackets [] to
subsample: x[4]
```

```
## [1] 18
```

```
# to select the 3rd to 6th
integers: x[3:6]
```

```
## [1]  5 18 22  3
```

```
# to select everything except the first and second value
# elements can be omitted using the  sign
x[c(-1, -2)]
```

```
 ## [1]  5 18 22 3 10 33 -2  0
```

6. Some useful R functions to use frequently

6.1. the c function

Combines elements of a vector and is very important in R

```
x <- 1:10
y <- 1:5
c(x, y)
```

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

Has uses everywhere. For example, setting the x-axis limits of a plot. Rather than saying something like `xmin = 0` , `xmax = 10` we can use `xlim = c(0, 10)`

6.2. which function

Useful for subsetting data in many situations - returns the id (index) of matching criterion

```
x <- 1:10 # sequence of numbers from 1 to 10
which(x > 5)
```

```
## [1]  6  7  8  9 10
```

```
# sometimes useful to read into a variable id
e.g. id <- which(x > 5)
x[id]
```

```
## [1]  6  7  8  9 10
```

6.3. seq and rep

`seq` and `rep` make sequences of numbers or replicates numbers. They are very useful in a whole range of situations.

Here are a few examples of `seq` :

```
# make a sequence of integers from 1
to 10 seq(from = 1, to = 10, by = 1)
```

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

```
# easier integer sequence,
use : 1:10
```

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

```
# or step up in twos
seq(from = 1, to = 10, by = 2)
```

```
## [1] 1 3 5 7 9
```

```
# often we need a sequence of a certain length, 15 in this
case seq(from = 0, to = 15, length.out = 15)
```

```
##  [1]  0.000000  1.071429  2.142857  3.214286  4.285714  5.357143  6.
428571
##  [8]  7.500000  8.571429  9.642857 10.714286 11.785714 12.857143 13.
928571
## [15] 15.000000
```

The use of `rep` is straightforward and very powerful…

```
# repeat '1' 5
times rep(1,
times = 5)
```

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

```
# repeat a vector (1, 3) five times i.e. 1, 3, 1, 3, ...
rep(c(1, 3), times = 5)
```

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

```
# argument each is very
useful rep(c(1, 3), each
= 5)
```

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

Practice a few sequences and repetitions and see if you can work out how to make a vector of hours of the day for a full week (assume hours on each day go from 0 to 23)

6.4. paste function

The `paste` function combines several character strings. The default separation in doing so is a space e.g.

```
paste("this", "that", "other")
```

```
## [1] "this that other"
```

```
# use the sep option
paste("this", "that", "other", sep = ", ")
```

```
## [1] "this, that, other"
```

```
# use values of a variable, in this case the
date paste("Today is", Sys.Date())
```

```
## [1] "Today is 2016-07-06"
```

```
# what about a vector of
values? paste("number",
1:5)
```

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

In the latter case we are pasting a single item `"number"` with a sequence of integers (1 to 5). In this case "number" is *recycled* 5 times to match the length of the integers. This again is one of the unusual aspects of the R language but it makes it very easy to work with data in a compact way.

There is also a `paste0` function that assumes no separation and can be slightly more efficient and 'tidy' to use e.g.

```
paste0("19/", "1/", "2015")
```

```
## [1] "19/1/2015"
```

`paste` and `paste0` are very useful in numerous situations. One final example is when trying to combine various date and time elements to form a

single date/time. For example, a data set may have a date in one column and the time in another.

6.5. The %in% (match) functions

Very useful when trying to match more than one thing. Note! == will only test for an exact match

```
Vars <- c("this", "that", "other")
which(Vars %in% c("this", "other"))
```

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

```
## not this:
which(Vars == c("this", "other"))
```

```
## Warning in Vars == c("this", "other"): longer object length is not a
## multiple of shorter object length
```

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

6.6. The grep function

`grep` is a very powerful character matching function.

Imagine a list of species we want to select from by matching particular characters

```
species <- c("CO", "NO2", "ClNO2", "H2", "CH4")
# select species that contain the string
'NO2' grep(pattern = "NO2", species,
ignore.case = TRUE)
```

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

This basic idea can be used to select in many powerful ways using *regular expressions* Also useful in data frames e.g. selecting particular columns in big data sets

6.7. Finding out about data in your R session

New users to R find the prompt (>) off-putting, where is my data and how can I find out about it!?

The easiest way to start is to work with RStudio and look at the *environment* tab. This contains information on all objects in your current R session.

There are a few common functions that I find invaluable when exploring data. These include `ls()` , `head` , `tail` , `nrow` and `str` . We will use these functions throughout the course to highlight their use.

7. Getting data into R

7.1. Many possibilities…

- Read from simple .txt or .csv files (`read.table` and `read.csv`)
- Make connections with databases
  - including MySQL, MS SQL Server, Postgres, sqlite and many more Use R's own data format (.RData files)
  - these are efficient (compressed)
- Dedicated functions to import into R directly
  - e.g. `importAURN` , `importKCL` , functions in openair
- Excel workbooks
  - Many packages can be used here and I sometimes use `XLConnect`
- Many ways of reading data from websites using packages such as `RCurl` XML, JSON etc.
  -

7.2. Brief introduction to data frames

When you read data into R using the methods above, the data are read into a *data frame*. The data frame is the most commonly used way of storing and working with data in R – it is also the way openair works with data. There are other data formats such as matrices, arrays and lists but these will not be covered in this course in a comprehensive way.

You can think of a data frame as being analogous to a spreadsheet in that data are stored in a rectangular grid and each column could be a different type of data e.g. numeric, integer or character.

Unlike spreadsheets a single column of data in a data frame must be of the same type e.g. integer. Spreadsheets on the other hand can be a mix of all sorts of data types (which actually makes spreadsheets less reliable to work with).

7.3. Generate a data frame in R

Most of the time a data frame in R will be the result of importing data from somewhere. However, it is also possible (and sometimes necessary) to generate data 'on the fly' using the `data.frame` function. For example, here is a simple data frame:

```
thedata <- data.frame(x = c(1, 2, 3), y = c("low", "medium",
                        "high"), z = c(0.5, 1, 10))
# show
the data
thedata
```

```
##   x      y    z
## 1 1low  0.5
## 2 2 medium  1.0
## 3 3   high 10.0
```

> Using some of the previous commands above e.g. `seq` and `rep` try making a data frame showing one year of hourly data that shows hour of the day and day of the week for 2015

7.4. Subsetting data frames

You can refer to a column of data in various ways. Going back to our simple data called `thedata`

The columns of the data frame are the vectors (representing variables).
Access them by name using the `$` symbol.

```
# the 'x'
column
thedata$x
```

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

Often it is useful to use the name of a variable directly, such as:

```
thedata[ , "x"]
```

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

One reason is you could have a variable representing that variable e.g.

```
myvar <- "x"

# now select it
thedata[ , myvar]
```

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

Or, access variables using square brackets that include a comma. Integers before the comma refer to rows, integers after the comma indicate columns: mydata[rows, columns].

```
thedata[2, 3] # 2nd row, 3rd column contents of data frame
```

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

```
thedata$y[2]  # 2nd element of species vector
```

```
## [1] medium
## Levels: high low medium
```

```
thedata[2, 3] # 2nd element of 3rd column vector
```

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

You can easily transform the value of a variable in a data frame. For example, to double the value of `x` :

```
thedata$x <- thedata$x * 2
# show all
the data
thedata
```

```
##   x      y    z
## 1 2low  0.5
## 2 4 medium  1.0
## 3 6   high 10.0
```

Or change a specific value:

```
# find all values in z that exactly equal 10 and set
to 20 thedata$z[thedata$z == 10] <- 20
thedata
```

```
##   x      y    z
## 1 2low  0.5
## 2 4 medium  1.0
## 3 6   high 20.0
```

We can make a new column (variable) based on the value of existing variables

```
# make new column that is x + z
thedata$alpha <- thedata$x +
thedata$z

# the transform function can make this a bit tidier...
thedata <- transform(thedata, beta = alpha * z)
```

```
thedata
```

```
##    x       y    z alpha   beta
## 1 2      low  0.5   2.5  1.25
## 2 4  medium  1.0   5.0  5.00
## 3 6    high 20.0  26.0 520.00
```

Selecting from data frames is very similar to that for vectors, except there are two dimensions. For example, to select the first two columns and row 2:

```
# this says "row 2" and "columns 1
to 2" thedata[2, 1:2]
```

```
##   x       y
## 2 4  medium
```

```
# if you want to select all rows, leave the first element
blank thedata[ , 2:3]
```

```
#      y     z
## 1  low  0.5
## 2  medium  1.0
## 3  high 20.0
```

More complex queries can be made either using the square brackets as above, or more clearly using the `subset` command.

```
# select all values of x >=2 and y = 'medium'
thedata[thedata$x >= 2 & thedata$y == "medium",]
```

```
##   xy z alpha beta
## 2 4 medium 1      5    5
```

```
# same as
subset(thedata, x >=2 & y == "medium")
```

```
##   xy z alpha beta
## 2 4 medium 1      5    5
```

> Try making your own data frame and experiment with selecting different parts, setting different values and adding new variables. Try experimenting with `%in%` and (maybe) `grep`
>
> ...

7.5. Changing the variable names

Often it is necessary to change the variable i.e. column names of a data frame. How do we do this? The names can be accessed using `names` or `colnames` functions,

which gives the vector of names. Being a vector we can do all the things described above.

```
names(thedata)
```

```
## [1] "x"     "y"     "z"     "alpha" "beta"
```

```
# refer to the second
name names(thedata)[2]
```

```
## [1] "y"
```

```
# the 2nd and 3rd
names
names(thedata)[2:3]
```

```
## [1] "y" "z"
```

So to change the names of columns 2 and 3 (y and z):

```
names(thedata)[2:3] <- c("a", "b")
```

> Can you work out how to use `which` or `grep` to find and change column names? Both are very useful and powerful, particularly for complex data sets

7.6. Example using read.csv

csv files are very common and are useful to be able to work with. The attraction of these files is that they are simple 'rectangular' format with each column separated by a comma (although in some countries a semi colon is used). They are also free of any formatting, multiple sheets etc.

Most of the time these files contain a header line at line 1 followed by data. However, this is not always the case and it is useful to know how to deal with different formats. Below is an example of reading in a simple file called 'ExampleDataLong.csv'. This file is a copy of the data that comes with the openair package but in csv format. Have a look at the file in Excel.

In R the `read.csv` function is the function to use to read in csv files (which actually uses `read.table` with different defaults e.g. the column separation is a comma)

```
# read 'ExampleDataLong.csv' file
dat <- read.csv("data/ExampleDataLong.csv", header = TRUE,
                stringsAsFactors = FALSE)

# look at the first few
lines head(dat)
```

```
##                date   ws  wd nox no2 o3 pm10    so2      co pm25
## 1 01/01/1998  00:00 0.60 280 285  39  1   29 4.7225  3.3725   NA
## 2 01/01/1998 01:00  2.16 230  NA  NA NA   37     NA      NA   NA
## 3 01/01/1998 02:00  2.76 190  NA  NA  3   34 6.8300  9.6025   NA
## 4 01/01/1998  03:00 2.16 170 493  52  3   35 7.6625 10.2175   NA
## 5  01/01/1998 04:00 2.40 180 468  78  2   34 8.0700  8.9125   NA
## 6  01/01/1998 05:00 3.00 190 264  42  0   16 5.5050  3.0525   NA
```

The following commands are useful for viewing aspects of a data frame. Also, in RStudio the *Environment* tab lists all the objects in the current R session. If you click on `dat` it will open it up so you can look through the data — this is a very useful aspect of RStudio but it is well worth familiarising yourself with the command line options below.

```
head(dat)     # prints the first few rows
tail(dat)     # prints the last few rows
names(dat)    # see the variable names
str(dat)      # check the variable types
rownames(dat) # view row names (numbers, if you haven't assigned names)
```

Some other useful data frame functions and operations are:

```
is.data.frame(dat)           # TRUE or FALSE
ncol(dat)                    # number of columns in data frame
nrow(dat)                    # number of rows
names(dat)                   # variable names
names(mydata)[4] <- c("NOx")   # change 1st variable name to quad
```

7.7. Formatting dates and times correctly

Robust treatment of dates and times in any software can be complex with issues related to time zones and daylight saving time (DST).

In data frames date-times are usually stored in `POSIXct` format in the form
e.g. 2016-07-06 15:39:22

It is much easier to work with data in GMT (i.e. UTC) as many potential pitfalls can be avoided related to different time zones and daylight savings time

You need to tell R what the original date format is and then format it as `POSIXct`

For example, say we have a file with dates in the format dd/mm/YYYY HH:MM and they were originally made in UTC:

```
myDate <- "20/6/2014 13:00"
as.POSIXct(strptime(myDate, format = "%d/%m/%Y %H:%M", tz = "GMT"))
```

```
## [1] "2014-06-20 13:00:00 GMT"
```

`strptime` converts the character string into an R representation of the date (note the setting of the time zone `tz`)

`as.POSIXct` converts it into a format that works well in data frames.

> Have a look at the help file for `strptime` and see if you can format the following dates

```
"2005-01-27"

"1/Feb/2009 14:00"

"31.12.1999 15"

"20150119"
```

While using base R functions to format dates and times is straightforward (once you know how to do it), many prefer using a package such as `lubridate` , which specialises in dealing with dates and times. You can install the package in the usual way, then load it:

```
library(lubridate)
```

```
##
## Attaching package: 'lubridate'
```

```
## The following object is masked from 'package:plyr':

##     here
```

```
## The following object is masked from 'package:base':

##     date
```

Have a look at functions such as `ymd` , `dmy_hm` , `parse_date_time`

```
dmy_hm(myDate)
```

```
## [1] "2014-06-20 13:00:00 UTC"
```

> Practice formatting dates and times with the `lubridate` package. Furthermore, see if you can use the `paste` function mentioned earlier to think about how to make a date/time out of a separate date and time based on the data below…

```
## make an example data set with date and time separately
mydat <- data.frame(date = c("14/12/2015", "15/12/2015"),
                    time = c("12:00", "12:00"))
```

> **Using `read.csv` and the `lubridate` functions, see if you can load the 'ExampleDataLong.csv' file that is in the course data directory. Check the format of the date once converted…

More often than not we want to import more than one file at a time. We can download individual files as described above — but it is much better to automat ethe process.

7.8. Save a data frame as a csv file

Sometimes it is useful to save (export) a data frame as a csv file, which can be done using the `write.csv` function:

```
write.csv(dat, file = "~/temp/testDat.csv", row.names = FALSE)
```

In this case the option `row.names = FALSE` is used. The default is `TRUE` and will add the row names (which will just be the line number in most cases) — which are generally not needed.

8. Cleaning up data

This is a potentially vast topic but often it is necessary to manipulate data in certain ways before it is analysed in more detail. Often it is possible to do this when importing data e.g. setting values to missing based on strings such as `-999` . However, there is a more general need to deal with data cleaning.

Here are a few examples. In all these cases it is important to ensure that potentially useful data are not 'thrown away' - or that some sort of bias in introduced.

8.1. Replace a number and set to missing

Imagine a case where an instrument reports '0' when it should be set as missing `NA` .

```
# make some fake data
test <- data.frame(x = c(1, 2, 0, 0, 4, 5),
                   y = c(0, 0, 12, 11, 10, 0))
test
```

```
##   x  y
## 1 1  0
## 2 2  0
## 3 0 12
## 4 0 11
## 5 4 10
## 6 5  0
```

To deal with one variable (say `x`) it is easy to find out which values are zero and then set them to missing:

```
# id of numbers equal to zero
id <- which(test$x == 0)

# now set to missing
test$x[id] <- NA
test
```

```
##    x  y
## 1  1  0
## 2  2  0
## 3 NA 12
## 4 NA 11
## 5  4 10
## 6  5  0
```

What if you had 10s of columns? It is possible to do this for an entire data frame. However, care must be taken!

```
test[test == 0] <- NA
test
```

```
##    x  y
## 1  1 NA
## 2  2 NA
## 3 NA 12
## 4 NA 11
## 5  4 10
## 6  5 NA
```

9. Working with Air Quality data from MOCCE

Once we will get familiar with the R environment we will use pre-built script to be used explore Air Quality data.

What we would like to achieve in the next training sessions is:

- **From Excel to R** (how to convert Excel spreadsheet from MOCCAE into a database format to be used for air quality analysis in R or any web-linked software)
- **Time-series** (static and dynamic)
- **Interactive** time-series for long data sets
- Summary **statistics** plots (daily and annual averages)
- Air Quality Indexes (**AQI**) (this will be explored once all participants are familiar with R).