Chapter 1

R objects - part 2

1.1 Factors

Factors are "vectors" of categoreical data. In practice categorical usually means that there are more than one observation of each category. However, since a R vector is defined so that all elements are of the same type and this is a more general type so **a factor is not a vector**. A factor also includes more information than a vector. Most important are the levels. It can also include labels. There is a function **is.factor** for checking if the object is a factor or not. It returns TRUE or FALSE.

```
x<-c(1,2,3,4,1,3)
x
```

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

```
str(x)
```

```
 num [1:6] 1 2 3 4 1 3
```

```
y<-as.factor(x)
str(y)
```

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

```
is.vector(x)
```

```
[1] TRUE
```

```
is.vector(y)
```

```
[1] FALSE
```

```
is.factor(x)
```

```
[1] FALSE
```

```
is.factor(y)
```

```
[1] TRUE
```

```
x<-factor(c("f","b","c","f","c","b","b")) # levels are stored in alphabetical order
x
```

```
[1] f b c f c b b
Levels: b c f
```

```
x<-factor(c(1,7.1,"a","Low",14,1,"a","high","low",14,1,"Low"))
x

 [1] 1    7.1  a    Low  14   1    a    high low  14   1    Low
Levels: 1 14 7.1 a high low Low

levels(x)   # if we only want the levels

[1] "1"    "14"   "7.1"  "a"    "high" "low"  "Low"

x[c(2,4)]   # all levels are printed

[1] 7.1 Low
Levels: 1 14 7.1 a high low Low
```

The levels of factors are by default listed alphabetically (numbers are included and come before letters). Sometimes the first level is used as a **reference**. We can change it.

```
y<-relevel(x,ref="high")
y

 [1] 1    7.1  a    Low  14   1    a    high low  14   1    Low
Levels: high 1 14 7.1 a low Low
```

Own experimentation

Create one or two own factors. Choose only few different values which are repeated several times.

1.2 Lists

A list is a general object which include e.g. data frame. It is a common result object from many functions.

```
x1<-data.frame(measurement=c(12.7,6.4,5.7,14,2.5),
               gender=factor(c("male","male","female","male","female")))
x2<-c(2.5,14)
x3<-5
x4<-"This is the minimum and maximum values"

x1

  measurement gender
1        12.7   male
2         6.4   male
3         5.7 female
4        14.0   male
5         2.5 female

typeof(x1)

[1] "list"

str(x1)
```

```
'data.frame': 5 obs. of  2 variables:
 $ measurement: num  12.7 6.4 5.7 14 2.5
 $ gender     : Factor w/ 2 levels "female","male": 2 2 1 2 1

is.list(x1)

[1] TRUE

y<-list(what=x4,min.max=x2,N=x3)
y

$what
[1] "This is the minimum and maximum values"

$min.max
[1]  2.5 14.0

$N
[1] 5

str(y)    # like a vector but elements of different type

List of 3
 $ what   : chr "This is the minimum and maximum values"
 $ min.max: num [1:2] 2.5 14
 $ N      : num 5
```

To access elements from a list we can use name or number as follows.

```
y$N

[1] 5

y[[2]]

[1]  2.5 14.0

z<-list(oldlist=y,data=x1)
z

$oldlist
$oldlist$what
[1] "This is the minimum and maximum values"

$oldlist$min.max
[1]  2.5 14.0

$oldlist$N
[1] 5


$data
  measurement gender
1        12.7   male
2         6.4   male
3         5.7 female
4        14.0   male
5         2.5 female

str(z)
```

```
List of 2
 $ oldlist:List of 3
  ..$ what   : chr "This is the minimum and maximum values"
  ..$ min.max: num [1:2] 2.5 14
  ..$ N      : num 5
 $ data   :'data.frame': 5 obs. of  2 variables:
  ..$ measurement: num [1:5] 12.7 6.4 5.7 14 2.5
  ..$ gender     : Factor w/ 2 levels "female","male": 2 2 1 2 1

z$oldlist

$what
[1] "This is the minimum and maximum values"

$min.max
[1]  2.5 14.0

$N
[1] 5

z[[1]]              # same as z$oldlist

$what
[1] "This is the minimum and maximum values"

$min.max
[1]  2.5 14.0

$N
[1] 5

z$oldlist[2]        # second element

$min.max
[1]  2.5 14.0

z[[1]][2]

$min.max
[1]  2.5 14.0

z[[2]][,2]          # second column of the second list element

[1] male   male   female male   female
Levels: female male
```

Own experimentation

Play around a little by creating lists. Try to include different elements like; vectors, text string, number or even data frames.

1.3 Dates

Functions for dates is useful both for reading data where format on date can be written in different formats but also for the handling of dates. For example we may only want to use the calendar year or calculate the difference in time between two dates. Also other functions is possible e.g. min and max.

```
x.t<-Sys.time()
x.t
```

```
[1] "2021-06-08 08:28:00 CEST"
```

```
str(x.t) #returns an object of class "POSIXct"
```

```
 POSIXct[1:1], format: "2021-06-08 08:28:00"
```

```
x.d<-Sys.Date()
x.d
```

```
[1] "2021-06-08"
```

```
str(x.d) # returns an object of class "Date"
```

```
 Date[1:1], format: "2021-06-08"
```

```
format(x.d,format="%Y-%m-%d")
```

```
[1] "2021-06-08"
```

```
# some alternatives
format(x.d,format="%d-%m-%Y")
```

```
[1] "08-06-2021"
```

```
format(x.d,format="%Y/%m/%d")
```

```
[1] "2021/06/08"
```

```
format(x.d,format="%Y")
```

```
[1] "2021"
```

```
x<-c("2017,02,16")
x
```

```
[1] "2017,02,16"
```

```
x.d<-as.Date(x,format="%Y,%m,%d")
x.d
```

```
[1] "2017-02-16"
```

```
format(x.d,format="%m/%Y")
```

```
[1] "02/2017"
```

```
y<-c("20190414")
y.d<-as.Date(y,format="%Y%m%d")
y.d
```

```
[1] "2019-04-14"
```

```
y.d-x.d
```

```
Time difference of 787 days
```

```
as.numeric(y.d-x.d)/365.25  # difference in years
```

```
[1] 2.154689
```

```
both<-c(x.d,y.d)
both
```

```
[1] "2017-02-16" "2019-04-14"
```

```
max(both)
```

```
[1] "2019-04-14"
```

1.4 Functions

We have seen that we can generate objects like vectors, data frames and lists. Other important objects are functions which are not used for storage of data but for e.g calculations or plots. Even if we can create our own functions, which we will look at later, most common is to use the large numbers of functions available in R. Functions (and sometimes data sets) are stored in **packages**. Some of these are available per default in core R.
We have already seen some examples of functions. Some more examples of functions are shown below.

```
x<-c(1,4,9,25,10,2)
length(x)

[1] 6

sqrt(x)

[1] 1.000000 2.000000 3.000000 5.000000 3.162278 1.414214

round(sqrt(x),2)

[1] 1.00 2.00 3.00 5.00 3.16 1.41

sort(x)

[1]  1  2  4  9 10 25

log(x)

[1] 0.0000000 1.3862944 2.1972246 3.2188758 2.3025851 0.6931472

sample(x,3)          # random sample from x

[1] 1 2 9

sample(x,10)

Error in sample.int(length(x), size, replace, prob):  cannot take a sample larger than the
population when 'replace = FALSE'

# it does not work because default is without replacement
# i.e. max sample size is length(x)

sample(x,10,replace=TRUE)  # but this works (with replacement)

 [1]  1  4  1  9  4  4  2  4 25 10
```

We have already seen that functions may be generic, i.e. what the function does is depending on the class of arguments. Later on we will learn how to make our own functions.

Own experimentation

Try the presented functions using own data. There are also alternatives to the function **round** if if you want e.g. the number 2.6 to be 2 or 2.4 to be 3. Can you find them by looking at help for round?

1.4.1 Installation of a package

To install a package the easiest way is to use one of the menus in RStudio [Tools->Install Packlages] above the **Edit pane** or [Packages->Install] in the **Help pane**.

1.4.2 Loading a package

If a a package is not available per default it has to installed. This mean that the files are downloaded to the computer. However, to make them acessable in R you need to use the **library** function to **load** the package. The installation only need to be made once but the loading has to be made for every new R session.

The istalled packages can be found in RStudio in the **Help pane**. Click on the package name and you will get a description.

With the function **search()** you can see the loaded packages together with the work space (.GlobalEnv). Some packages are loaded by default, see **options()** below.

```
options()$defaultPackages      # packages loaded by default

[1] "datasets"  "utils"     "grDevices" "graphics"  "stats"     "methods"

search()                       # all loaded packages

 [1] ".GlobalEnv"        "package:knitr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"

library(Epi)
search()

 [1] ".GlobalEnv"        "package:Epi"       "package:knitr"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"

detach("package:Epi")  # remove a specific package/data base from the search path
search()

 [1] ".GlobalEnv"        "package:knitr"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

Own experimentation

Load a package e.g. Epi. In which search position is it put? Then use deatch() without specification. Result? Which position of the search path was affected?

1.4.3 Masking

Because R is developed by an open source community, it may happen that multiple packages may use the same name for a function or dataset. If you load packages that use the same name for an object,

R will warn that certain object(s) have been "masked" by the new package. In the example below the function sqrt is masked. The original function is still there but cannot be reached since the new definition in the work space has the first search position.

```
rm(list=ls())       # just for cleaning up a little

x<-9
sqrt(x)             # square root of x

[1] 3

sqrt<-log
ls()                # ls() list the objects in work space

[1] "sqrt" "x"

sqrt(x)             # sqrt now is the log function

[1] 2.197225

base::sqrt(x)       # the solution if we have masking

[1] 3

rm(sqrt)
ls()

[1] "x"

sqrt(x)             # not masked anymore

[1] 3
```