# Chapter 1

# Some useful functions

## 1.1 Functions in R

Functions are always followed by brackets ( ) and within these they usually have a number of arguments. Some arguments may have a default value and usually only one or two arguments are needed. So mostly you don't have to care about the optional arguments. You can however find them using the help function e.g. **?sum** which gives information on the arguments of the function sum, the value returned from the function as well as examples and references. This help exists for all functions and some other objects.

## 1.2 Some basic functions on numerical vectors

We start by looking at **sum, mean and max**. It is obvious from the names what the functions do. There are also related functions like **min, sd, var, cumsum, range** which we don't exemplify here. They can be found in the help or simply by google e.g. "standard deviation r".

```
x<-c(1,3,5,7,7,5)
x

[1] 1 3 5 7 7 5

sum(x)

[1] 28

mean(x)

[1] 4.666667

max(x)

[1] 7
```

**Own experimentation**

First generate another vector than **x** above. Then try the functions above on the new vector. Also try the **cumsum** and **min** functions. Are there any important arguments for any of these functions? Try to find the functions for standard deviation (S) and variance (V). They are related as $S = \sqrt{V}$. Calculate S and V for a vector e.g. x above and compare the result of the two measures by using either **sqrt** or $\wedge$. What happens if you forget the ( ) when you use a function?

## 1.3 Generic functions

In R there are some generic functions where the **class** of the object determines what the function will do. The function **summary( )** is an example of a such generic function. The result from **summary** include the functions **min, max, median, mean** but it also include the function **quantile**. The result of quantile is not a single number but a vector, by default it is the limits when the data is partiotioned in 4 equal parts according to the values. It is not obvious how to calculate it, therefore there are several methods to chose among. Other examples of the use of e.g. **summary** will follow later.

```
people <- c('Lena', 'Solveig', 'Anna', 'Hans', 'Erik' )
byear <-  c(1982,      1976,      1949,      2001,   1967)
score <-  c( 22,        43,        87,        45,     60)
salary <- c(40,         47,        31,        32,     45)
df <- data.frame(Name = people, BirthYear = byear, Score = score, Salary = salary)
df

     Name BirthYear Score Salary
1    Lena      1982    22     40
2 Solveig      1976    43     47
3    Anna      1949    87     31
4    Hans      2001    45     32
5    Erik      1967    60     45

summary(score)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   22.0    43.0    45.0    51.4    60.0    87.0

summary(df)

     Name             BirthYear        Score          Salary
 Length:5           Min.   :1949   Min.   :22.0   Min.   :31
 Class :character   1st Qu.:1967   1st Qu.:43.0   1st Qu.:32
 Mode  :character   Median :1976   Median :45.0   Median :40
                    Mean   :1975   Mean   :51.4   Mean   :39
                    3rd Qu.:1982   3rd Qu.:60.0   3rd Qu.:45
                    Max.   :2001   Max.   :87.0   Max.   :47
```

## 1.4 Some functions for investigation of data frames

Here are a few of core R's other functions to investigate data frame structure.

```
names(df)

[1] "Name"      "BirthYear" "Score"     "Salary"

names(df)[3]

[1] "Score"

ncol(df)

[1] 4

nrow(df)

[1] 5

dim(df)

[1] 5 4

str(df)

'data.frame': 5 obs. of  4 variables:
 $ Name     : chr  "Lena" "Solveig" "Anna" "Hans" ...
 $ BirthYear: num  1982 1976 1949 2001 1967
 $ Score    : num  22 43 87 45 60
 $ Salary   : num  40 47 31 32 45

class(df)

[1] "data.frame"
```

**Own experimentation**

Check if the result of **summary** is correct by using the funtions **min, max, mean, median and quantile** on each vector e.g. mean(df$Salary). The function **dim** results in two numbers. What kind of object is the result?

## 1.5 Some more useful functions for data frames

It is sometimes useful to add a data frame to the rows or columns of another data frame. Observe that for **cbind** the number of rows must agree and for **rbind** the number of columns and column names must agree.

```
a<-c(37,35,62,45,70)
h<-c(170,162,166,174,192)
df2<-data.frame(Age=a,Heigt=h)
df2

  Age Heigt
1  37   170
2  35   162
3  62   166
```

```
4  45   174
5  70   192

dfw<-cbind(df,df2)  # add the data frames by column
dfw

     Name BirthYear Score Salary Age Heigt
1    Lena      1982    22     40  37   170
2 Solveig      1976    43     47  35   162
3    Anna      1949    87     31  62   166
4    Hans      2001    45     32  45   174
5    Erik      1967    60     45  70   192

dfw24<-dfw[c(2,4),]
dfw24

     Name BirthYear Score Salary Age Heigt
2 Solveig      1976    43     47  35   162
4    Hans      2001    45     32  45   174

df.dupl<-rbind(dfw,dfw24)  # add the data frames by row
df.dupl

      Name BirthYear Score Salary Age Heigt
1     Lena      1982    22     40  37   170
2  Solveig      1976    43     47  35   162
3     Anna      1949    87     31  62   166
4     Hans      2001    45     32  45   174
5     Erik      1967    60     45  70   192
21 Solveig      1976    43     47  35   162
41    Hans      2001    45     32  45   174
```

Later on we will be looking at the tidyverse package dplyr. There are functions **bind_rows** and **bind_cols** similar to the **rbind** and **cbind** above. Another function in dplyr is shown below. It is useful for removing duplicated rows.

```
library(dplyr)
distinct(df.dupl)

     Name BirthYear Score Salary Age Heigt
1    Lena      1982    22     40  37   170
2 Solveig      1976    43     47  35   162
3    Anna      1949    87     31  62   166
4    Hans      2001    45     32  45   174
5    Erik      1967    60     45  70   192
```

**Own experimentation**

Try the functions cbind and rbind. You can e.g. use df in the example above and create sub-data.frames that you add to the original df.

## 1.6 Systematic generation of data

We here show two funtions **rep** and **seq** useful for generation of data. They can also be combined in several ways.

```r
rep(2,6)
```

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

```r
rep(c(1,5,7),times=3)
```

```
[1] 1 5 7 1 5 7 1 5 7
```

```r
rep(c(1,5,7),each=3)
```

```
[1] 1 1 1 5 5 5 7 7 7
```

```r
rep(c(1,5,7),times=2,each=3)
```

```
 [1] 1 1 1 5 5 5 7 7 7 1 1 1 5 5 5 7 7 7
```

```r
1:5
```

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

```r
seq(from=1,to=5)
```

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

```r
seq(1,5,by=1)
```

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

```r
seq(1,5,by=0.5)
```

```
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
seq(1,8,by=3)
```

```
[1] 1 4 7
```

```r
seq(4,by=3,length.out=6)
```

```
[1]  4  7 10 13 16 19
```

```r
#
#rep is gereral. Also characters and strings can be used
rep(c(412,"c","May is a nice month"),times=2)
```

```
[1] "412"                 "c"                   "May is a nice month"
[4] "412"                 "c"                   "May is a nice month"
```

**Own experimentation**

Try out the functions in different ways. Suggestion: try to use **rep** to generate the vectors below.

```
[1] "vector 1"
 [1] 8 9 8 9 8 9 8 9 8 9
[1] "vector 2"
 [1] 8 8 8 9 9 9 8 8 8 9 9 9
[1] "vector 3"
 [1] 8 8 9 9 8 8 9 9 8 8 9 9
[1] "vector 4"
[1] "2"      "monkey" "Lemon"  "2"      "monkey" "Lemon"
[1] "vector 5"
[1] "2"      "2"      "monkey" "monkey" "Lemon"  "Lemon"
[1] "vector 6"
 [1] "2"      "2"      "monkey" "monkey" "Lemon"  "Lemon"  "2"      "2"
 [9] "monkey" "monkey" "Lemon"  "Lemon"
[1] "vector 7"
[1] "2"      "2"      "monkey" "monkey" "Lemon"  "Lemon"  "2"      "2"
[9] "monkey"
```

Now try to use **seq** to generate the vectors below.

```
[1] "vector 1"
[1] 11 12 13 14
[1] "vector 2"
[1]  7 10 13 16 19
[1] "vector 3; the odd numers up to 39"
 [1]  1  3  5  7  9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
```

## 1.7 Sorting

Let us first generate a data frame. The function **sort** sorts a vector by deafault ascending but it can be changed descending. To sort a data frame due to one column (vector) the easiest way is to give the ordered row numbers as a vector (see the example). The function **order** gives the row numbers of the original vector sorted. This can be generalised to sort data frames due to more than one column.

```
df<-data.frame(var1=c(1,3,5,3,5,7),var2=rep(c(2,5,3)),var3=rep(c("A","O"),3),
  var4=rep(c("a",4,"Lars")),var5=seq(10,7,length.out=6))

df

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
2    3    5    O    4  9.4
3    5    3    A Lars  8.8
4    3    2    O    a  8.2
5    5    5    A    4  7.6
6    7    3    O Lars  7.0

sort(df$var2)              # Sorting

[1] 2 2 3 3 5 5

sort(df$var2,decreasing=T)   #change to decreasing order
```

```
[1] 5 5 3 3 2 2

df

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
2    3    5    O    4  9.4
3    5    3    A Lars  8.8
4    3    2    O    a  8.2
5    5    5    A    4  7.6
6    7    3    O Lars  7.0

df[c(1,4,3,6,2,5),] # manual sorting by var2

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
4    3    2    O    a  8.2
3    5    3    A Lars  8.8
6    7    3    O Lars  7.0
2    3    5    O    4  9.4
5    5    5    A    4  7.6

df$var2

[1] 2 5 3 2 5 3

order(df$var2) # Sorting using positions(row numbers) of var2.

[1] 1 4 3 6 2 5

                  # It is the same vector as we used for sorting above.
df[order(df$var2),] # so therfore this will also be the dataframe sorted by var2

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
4    3    2    O    a  8.2
3    5    3    A Lars  8.8
6    7    3    O Lars  7.0
2    3    5    O    4  9.4
5    5    5    A    4  7.6

df[order(df$var2,decreasing=T),] # same but decreasing

  var1 var2 var3 var4 var5
2    3    5    O    4  9.4
5    5    5    A    4  7.6
3    5    3    A Lars  8.8
6    7    3    O Lars  7.0
1    1    2    A    a 10.0
4    3    2    O    a  8.2

df[order(df$var3),] # we can also sort on characters or factors

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
3    5    3    A Lars  8.8
5    5    5    A    4  7.6
2    3    5    O    4  9.4
4    3    2    O    a  8.2
6    7    3    O Lars  7.0
```

```
df[order(df$var3,df$var2),] # sorted by var3 and then var2

  var1 var2 var3 var4 var5
1    1    2    A    a 10.0
3    5    3    A Lars  8.8
5    5    5    A    4  7.6
4    3    2    O    a  8.2
6    7    3    O Lars  7.0
2    3    5    O    4  9.4
```

**Own experimentation**

Suggestion: Try to sort the data frame above by var1, by var5, by var5 and var1, by var1(increasing) and var2(decreasing). You can also try to sort it in the order 1,6,2,5,3,4 of the rows in df.

## 1.8 Filtering subsets of data frames

By giving a logical expression in the row we can get a subset of a vector or data frame. However, we have to watch out for missing data which can give an incorrect result. As an example, assume we want to select females below the age of 40 with a height over 175.

```
df<-data.frame(sex=factor(c("male","male","female","female","female","female")),
               age=c(25,42,38,39,44,24),height=c(172,184,177,152,171,180))
df

     sex age height
1   male  25    172
2   male  42    184
3 female  38    177
4 female  39    152
5 female  44    171
6 female  24    180

df[df$sex=="female" & df$age<40 & df$height>175,]

     sex age height
3 female  38    177
6 female  24    180
```

Let us have a look at the data frame in the above example. The selection of female (df$sex=="female") actually gives a logic vector. Which means that we select the rows with result equal to TRUE. We can see that the only rows whith all three conditions TRUE are the 3rd and 6th row which were selected in the former example. Below we combine the three selection criteria.

```
v1<-df$sex=="female"
v2<-df$age<40
v3<-df$height>175
vall<-v1 & v2 & v3
data.frame(v1,v2,v3,vall)

     v1    v2    v3  vall
1 FALSE  TRUE FALSE FALSE
2 FALSE FALSE  TRUE FALSE
```

```
3  TRUE  TRUE  TRUE  TRUE
4  TRUE  TRUE FALSE FALSE
5  TRUE FALSE FALSE FALSE
6  TRUE  TRUE  TRUE  TRUE

df[vall,]

     sex age height
3 female  38    177
6 female  24    180
```

## 1.9 Missing data

Now what happens if there is a missing in any of the variables sex, age and height? A missing appears as a **NA** (not available). We need to be somewhat careful when handling missing values. There are some functions to help with this. Let us look at the same example but with some missing data.

```
df.missing<-data.frame(sex=factor(c("male","male",NA,"female","female","female")),
                       age=c(25,42,38,39,NA,24),height=c(172,NA,177,152,NA,180))
df.missing

     sex age height
1   male  25    172
2   male  42     NA
3   <NA>  38    177
4 female  39    152
5 female  NA     NA
6 female  24    180

# we now filter the data as above and get
df.missing[df.missing$sex=="female"  & df.missing$age<40 & df.missing$height>175,]

        sex age height
NA     <NA>  NA     NA
NA.1   <NA>  NA     NA
6    female  24    180

df.missing[6,]  # this is probably what we actually wanted

     sex age height
6 female  24    180

df.missing$height

[1] 172  NA 177 152  NA 180

df.missing$sex    # missing (NA) is not seen as a factor level

[1] male   male   <NA>   female female female
Levels: female male

# WARNING in the following example NA is interpreted as a text category
factor(c("male","male","NA","female","female","female"))

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

```
v1<-df.missing$sex=="female"  # check the three conditions
v2<-df.missing$age<40
v3<-df.missing$height>175
vall<-v1 & v2 & v3
data.frame(sex=v1,age=v2,height=v3,total=vall)  # vall = all conditions fulfilled

    sex   age height total
1 FALSE  TRUE  FALSE FALSE
2 FALSE FALSE     NA FALSE
3    NA  TRUE   TRUE    NA
4  TRUE  TRUE  FALSE FALSE
5  TRUE    NA     NA    NA
6  TRUE  TRUE   TRUE  TRUE

df.missing[vall,] # the same as we got above

        sex age height
NA     <NA>  NA     NA
NA.1   <NA>  NA     NA
6    female  24    180

# also rows where the logical expression cannot be evaluated (NA) are included
nrow(df.missing[vall,])

[1] 3
```

Let us for a moment only focus on the selection on sex.

```
is.na(df.missing$sex)    # check if missing

[1] FALSE FALSE  TRUE FALSE FALSE FALSE

!is.na(df.missing$sex)  # not missing

[1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE

df.missing[df.missing$sex=="female" ,]

      sex age height
NA   <NA>  NA     NA
4  female  39    152
5  female  NA     NA
6  female  24    180

df.missing[df.missing$sex=="female" & !is.na(df.missing$sex) ,] # this is better,

     sex age height
4 female  39    152
5 female  NA     NA
6 female  24    180

# To make a correct selection to avoid problems with missing data
df.missing[df.missing$sex=="female" & !is.na(df.missing$sex) & df.missing$age<40 &
             !is.na(df.missing$age)& df.missing$height>175& !is.na(df.missing$height),]

     sex age height
6 female  24    180
```

### 1.9.1 Some functions when dealing with missing data

The functions **subset** and **na.omit** may be useful.

```
df.missing

     sex age height
1   male  25    172
2   male  42     NA
3   <NA>  38    177
4 female  39    152
5 female  NA     NA
6 female  24    180

# This is a function that make things easier
# Here is a difference - missing values are taken as false
subset(df.missing,sex=="female" & age<40 & height>175)

     sex age height
6 female  24    180

# subset also makes the filtering easier since you don't have to
# repeat the name of the data frame
# which is especially helpful if it has a long name

dataframe.with.a.long.name<-df # No missing here
dataframe.with.a.long.name[dataframe.with.a.long.name$sex=="female"
                          & dataframe.with.a.long.name$age<40
                          & dataframe.with.a.long.name$height>175,]

     sex age height
3 female  38    177
6 female  24    180

subset(dataframe.with.a.long.name,sex=="female" & age<40 & height>175)

     sex age height
3 female  38    177
6 female  24    180

# You may want to remove all missing, so called removing listwise
df.clean<-na.omit(df.missing)
df.clean

     sex age height
1   male  25    172
4 female  39    152
6 female  24    180

df.clean[df.clean$sex=="female"  & df.clean$age<40 & df.clean$height>175,]

     sex age height
6 female  24    180
```

### 1.9.2 Arguments in functions to handle missing data

The **na.rm** is an argument you can found in many functions. Another alternative is **use** (used in `var()` and `cor()`). It is a good idea to check the help.

```
#
x<-c(1,3,5,NA,7,5,14,NA,6)
x

[1]  1  3  5 NA  7  5 14 NA  6

sum(x) #does not work

[1] NA

sum(x,na.rm=T) # this works

[1] 41

mean(x)

[1] NA

mean(x,na.rm=T)

[1] 5.857143
```

Below are some examples of handling counts in tables when there are missing data.

```
set.seed(221)
df<-data.frame(size=sample(c("big","small"),size=20,replace=T),
               time=sample(c("before","middle","after"),size=20,replace=T))
dfm<-df
dfm[c(6,16),1]<-NA
dfm[c(4),2]<-NA
dfm            # dfm has 3 missing

    size   time
1  small middle
2  small before
3    big  after
4    big   <NA>
5    big  after
6   <NA> middle
7    big before
8  small middle
9  small middle
10   big middle
11   big before
12   big  after
13 small  after
14 small  after
15   big before
16  <NA>  after
17   big middle
18 small middle
19 small middle
20   big middle

table(df$time,df$size)

         big small
  after    4     2
  before   3     1
  middle   5     5
```

```
table(dfm$time,dfm$size)  # we cannot see in the table that we have missing data

         big small
  after    3     2
  before   3     1
  middle   3     5

table(dfm$time,dfm$size,useNA = "ifany")

         big small <NA>
  after    3     2    1
  before   3     1    0
  middle   3     5    1
  <NA>     1     0    0

xtabs(~time+size,data=dfm,addNA = T)

        size
time     big small <NA>
  after    3     2    1
  before   3     1    0
  middle   3     5    1
  <NA>     1     0    0
```

## 1.10 The function cut

The function cut is used to split a numerical continuous variable into categories.

```
x<-seq(0,10)
x

 [1]  0  1  2  3  4  5  6  7  8  9 10

gr<-cut(x,breaks=c(0,2,5,Inf))

gr

 [1] <NA>    (0,2]   (0,2]   (2,5]   (2,5]   (2,5]   (5,Inf] (5,Inf] (5,Inf]
[10] (5,Inf] (5,Inf]
Levels: (0,2] (2,5] (5,Inf]

class(gr)

[1] "factor"

summary(gr)   # similar as table() for factors

  (0,2]   (2,5] (5,Inf]    NA's
      2       3       5       1

gr.lab<-cut(x,breaks=c(0,2,5,Inf),labels=c("small","medium","large"))
gr.lab

 [1] <NA>   small  small  medium medium medium large  large  large  large
[11] large
Levels: small medium large

summary(gr.lab)
```

```
 small medium  large   NA's
     2      3      5      1

table(gr.lab,useNA="ifany") # alt for table

gr.lab
 small medium  large   <NA>
     2      3      5      1

xtabs(~gr.lab,addNA=T)       # alt for xtabs

gr.lab
 small medium  large   <NA>
     2      3      5      1
```

Observe e.g. (2,5] means that the intervals are open to the left (2 is not included but 50 is). This is possible to change. See help for info (cut(right))

**Own experimentation**

Try to change the cut-off points and the values in the original vector x.
Why is there one missing (NA) in the variable gr above? Can you make so all values from 0 to 2 belong to a category?

## 1.11 The function apply

Apply is a function for calculations made by rows or by columns in a data frame. Calculations by row is not always meaningful.

```
x<-data.frame(bmi.gr1=c(21.6,24.3,29.2,26.1,23.6),bmi.gr2=c(26.7,28.3,25.2,28.1,27.1),
              bmi.gr3=c(24.3,23.2,24.8,23.5,25.0))
x

  bmi.gr1 bmi.gr2 bmi.gr3
1    21.6    26.7    24.3
2    24.3    28.3    23.2
3    29.2    25.2    24.8
4    26.1    28.1    23.5
5    23.6    27.1    25.0

apply(x,1,mean)    # mean by row

[1] 24.20000 25.26667 26.40000 25.90000 25.23333

apply(x,2,mean)    # mean by column (variable names are kept)

bmi.gr1 bmi.gr2 bmi.gr3
  24.96   27.08   24.16

apply(x,2,sd)

  bmi.gr1   bmi.gr2   bmi.gr3
2.8658332 1.2457929 0.7893035

apply(x,2,sort)    # the function does not need to produce one numeric result
```

```
     bmi.gr1 bmi.gr2 bmi.gr3
[1,]    21.6    25.2    23.2
[2,]    23.6    26.7    23.5
[3,]    24.3    27.1    24.3
[4,]    26.1    28.1    24.8
[5,]    29.2    28.3    25.0
```

## 1.12 Extension of apply to tapply

The function **tapply** is similar to **apply** but more general. Instead of applying a function to a row or column it is applied to one or more categorical variables e.g. sex and age.

```
library(haven)
norsjo86 <- read_sav("../data/norsjo86.sav")
norsjo86<-as.data.frame(norsjo86)
tapply(norsjo86$height,INDEX=list(factor(norsjo86$sex)),FUN=max,na.rm=T)

  1   2
191 178

tapply(norsjo86$sbp,INDEX=list(factor(norsjo86$sex),factor(norsjo86$agegrp)),
       FUN=mean,na.rm=T)

        30       40       50       60
1 119.0000 121.4839 127.4444 134.1212
2 109.6667 117.0556 132.1143 133.5938
```

## 1.13 The head and tail functions

If you have a data frame with many rows it is not practical to print all of it. With these functions yoy can look at the beginning or the end of the data frame.

```
x<-data.frame(x=seq(1,10,length.out=4000),y=seq(-628,10,length.out=4000),
              z=seq(0,180,length.out=4000))
nrow(x)

[1] 4000

head(x)

         x         y          z
1 1.000000 -628.0000 0.00000000
2 1.002251 -627.8405 0.04501125
3 1.004501 -627.6809 0.09002251
4 1.006752 -627.5214 0.13503376
5 1.009002 -627.3618 0.18004501
6 1.011253 -627.2023 0.22505626

tail(x,10)

             x         y        z
3991  9.979745  8.564141 179.5949
3992  9.981995  8.723681 179.6399
```

```
3993  9.984246  8.883221 179.6849
3994  9.986497  9.042761 179.7299
3995  9.988747  9.202301 179.7749
3996  9.990998  9.361840 179.8200
3997  9.993248  9.521380 179.8650
3998  9.995499  9.680920 179.9100
3999  9.997749  9.840460 179.9550
4000 10.000000 10.000000 180.0000
```

**Own experimentation**

Take a look at the data frame x in this example - not very practical to print by typing x. There may even be a maximum number of rows for the output. Try to convert it to a tibble using **as_tibble** and print it. Any difference?