Session 1: First steps in R and data exploration

Ralf B. Schäfer

2019-10-20

First steps

Before we begin, some notes on notation and terminology: **Bold statements** mostly refer to little exercises or questions, R functions inside the text look like this: `function()` and this is an URL that brings you to the course website. R code chunks are formatted as follows (This code does not run!):

```
object_assignment <- thisfunction(argument = does_not_work$it_only_serves_illustration)
# Comments within R code are coloured in brown.
```

If you write own R code, consider the style guide by Hadley Wickham as well as using the styler package on files or on code through the RStudio Addins. Although called "First steps", we assume that you have followed short tutorials on installing and using R/R Studio and on the very basics of R. We recommend the free datacamp course Introduction to R and the first free chapters of this Online book.

Whenever you start working with R, you should set a working directory. This is the directory where R, unless specified otherwise, will look for files to load or will save files. The working directory can be set through the R Studio Graphical User Interface (GUI): Go to Session –> Set Working Directory –> Choose Directory... . However, you can also do this from the command line using the command `setwd()`:

```
setwd("~/Gitprojects/Teaching/Data_analysis/Code")
```

If you run the script, you have to replace my file path with a path to a working directory on your local machine. To simplify the identification of your path, you can use the following function:

```
file.choose()
```

and select a file in your desired working directory. Subsequently, copy the path **without** the file reference into the `setwd()` function. Make sure to enclose the path with double quotation marks. Finally, we set an option to reduce the amount of output that is printed to save some space in this document:

```
options(max.print = 50)
```

For an overview of the options available to influence computations and displaying check the help:

```
?options
```

We start by importing a data file from the university website (obviously, you require an internet connection to successfully run the command). To store the data after import, we assign (<-) the data to an object (*data*).

```
link <- "http://www.uni-koblenz-landau.de/en/campus-landau/faculty7/environmental-sciences/landscape-ec
# link is not properly displayed in knitted document, but you can copy it from
# the Session_1.R document (which you should use anyway to run the R code yourself)
data <- read.table(link)
```

The object *data* is now shown in the *Environment pane* in R Studio. For interested students: **What happens if you run the read table function without assignment to an object?** *Hint: Inspect the Console pane in R Studio.*

Useful functions to inspect imported data are `head()` and `str()`. **Try what these functions do by running them on the imported data and by calling the related help pages as shown below:**

```
head(data)
?head
```

```
str(data)
?str
```

The imported data look mixed up. This happens when the arguments in the import function have not been set properly. We can set different arguments including:

- header: specify if the data have a header (TRUE) or not (FALSE)
- sep: specify the column separator (in this case ";")
- dec: specify the character used for the decimal point (in this case "."). In Germany and several other countries, the character for the decimal point is ",", which often leads to trouble (unless you do not exchange files with others). In academic contexts, I generally recommend to set all software products such as spreadsheet programs (e.g. Microsoft Excel, LibreOffice Calc) to locale "English (Great Britain)" or another locale related to an english-speaking country.
- row.names: specify row names, a variable that provides row names (our data set actually contains a variable *row.names*, but we ignore that) or import without row names (= NULL)

Call `?read.table` for further details and options. Misspecification of import arguments is one of the most frequent errors of beginners. We run the import function again, now with the arguments properly specified.

```
pos_dat <- read.table(url(link), dec = ".", sep = ";", header = TRUE, row.names = NULL)
# close connection after import
close(url(link))
```

For further details on how to import data refer to this tutorial.

We inspect the imported data again.

```
# show first 3 rows of dataframe
head(pos_dat)
```

```
##   row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 1        C3    1    1 Vic   m   8    94.1   60.4     89.0  36.0     74.5
## 2        C5    2    1 Vic   f   6    92.5   57.6     91.5  36.5     72.5
## 3       C10    3    1 Vic   f   6    94.0   60.0     95.5  39.0     75.4
##   earconch  eye chest belly
## 1     54.5 15.2  28.0    36
## 2     51.2 16.0  28.5    33
## 3     51.9 15.5  30.0    34
##  [ reached 'max' / getOption("max.print") -- omitted 3 rows ]
```

```
# looks ok
str(pos_dat)
```

```
## 'data.frame':    104 obs. of  15 variables:
##  $ row.names: chr  "C3" "C5" "C10" "C15" ...
##  $ case     : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ site     : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ Pop      : Factor w/ 2 levels "other","Vic": 2 2 2 2 2 2 2 2 2 2 ...
##  $ sex      : Factor w/ 2 levels "f","m": 2 1 1 1 1 1 2 1 1 1 ...
##  $ age      : int  8 6 6 6 2 1 2 6 9 6 ...
##  $ hdlngth  : num  94.1 92.5 94 93.2 91.5 93.1 95.3 94.8 93.4 91.8 ...
##  $ skullw   : num  60.4 57.6 60 57.1 56.3 54.8 58.2 57.6 56.3 58 ...
##  $ totlngth : num  89 91.5 95.5 92 85.5 90.5 89.5 91 91.5 89.5 ...
##  $ taill    : num  36 36.5 39 38 36 35.5 36 37 37 37.5 ...
##  $ footlgth : num  74.5 72.5 75.4 76.1 71 73.2 71.5 72.7 72.4 70.9 ...
##  $ earconch : num  54.5 51.2 51.9 52.2 53.2 53.6 52 53.9 52.9 53.4 ...
##  $ eye      : num  15.2 16 15.5 15.2 15.1 14.2 14.2 14.5 15.5 14.4 ...
##  $ chest    : num  28 28.5 30 28 28.5 30 30 29 28 27.5 ...
```

```
##  $ belly    : num  36 33 34 34 33 32 34.5 34 33 32 ...
```

```
# structure of object also looks meaningful now
# data.frame with 15 variables and 104 rows (obs. = observations)
# $ indicates variables with type of data (i.e. Factor: categorical, chr: character,
# int: integer = natural number, num: numeric = real number)
```

The file was taken from the R package DAAG and contains information on possums that were published in Lindenmayer et al. (1995). To access an R package, we have to load and attach a package with the function `library()`:

```
library(DAAG)
```

If you do not have the package installed, you need to install the package via:

```
# install.packages("DAAG") # (remove hashtag in this case)
```

If loaded and attached, we can subsequently load the possum data, which we have imported from a file above, quite conveniently (e.g. without the need to specify separator or decimal point) from the package:

```
data(possum)
# display first rows
head(possum)
```

```
##     case site Pop sex age hdlngth skullw totlngth taill footlgth earconch
## C3     1    1 Vic   m   8    94.1   60.4     89.0  36.0     74.5     54.5
## C5     2    1 Vic   f   6    92.5   57.6     91.5  36.5     72.5     51.2
## C10    3    1 Vic   f   6    94.0   60.0     95.5  39.0     75.4     51.9
##      eye chest belly
## C3  15.2  28.0    36
## C5  16.0  28.5    33
## C10 15.5  30.0    34
##  [ reached 'max' / getOption("max.print") -- omitted 3 rows ]
```

The metadata for data that are in a package can be called via help. **Call the help for the possum data and study the metadata.**

You can also save data that are in the session workspace for later use. For example, if you wanted to save the imported possum data for later use (ignoring that we have them in the package) on your local hard drive, execute the following code:

```
save(pos_dat, file = "Possum.Rdata")
```

Of course, you can provide a different path to the file argument in the `save()` function. **Where has the file been saved?** *Hint: If you can't find the file, run the following function:*

```
getwd()
```

If saved, you can load the data into a new R session with:

```
load("Possum.Rdata")
# Works only if file is in working directory!
```

Data handling

Now that we have properly imported the data into R, we can explore and analyse the data. However, before data exploration and analysis, the data often require (re-)organisation including joining data sets, transformation and subsetting. Here, we focus on subsetting based on conditions. But let us start with some basics. To return the column names (i.e. variable names) call:

```
names(pos_dat)
```

```
##  [1] "row.names" "case"      "site"      "Pop"       "sex"
##  [6] "age"       "hdlngth"   "skullw"    "totlngth"  "taill"
## [11] "footlgth"  "earconch"  "eye"       "chest"     "belly"
```

or:

```
colnames(pos_dat)
```

```
##  [1] "row.names" "case"      "site"      "Pop"       "sex"
##  [6] "age"       "hdlngth"   "skullw"    "totlngth"  "taill"
## [11] "footlgth"  "earconch"  "eye"       "chest"     "belly"
```

The function `colnames()`, in contrast to `names()`, also works for matrices. If we want to access variables in a dataframe, we can do this as follows:

```
pos_dat$totlngth
```

```
##  [1] 89.0 91.5 95.5 92.0 85.5 90.5 89.5 91.0 91.5 89.5 89.5 92.0 89.5 91.5
## [15] 85.5 86.0 89.5 90.0 90.5 89.0 96.5 91.0 89.0 84.0 91.5 90.0 85.0 87.0
## [29] 88.0 84.0 93.0 94.0 89.0 85.5 85.0 88.0 82.5 80.5 75.0 84.5 83.0 77.0
## [43] 81.0 76.0 81.0 84.0 89.0 85.0 85.0 88.0
##  [ reached getOption("max.print") -- omitted 54 entries ]
```

```
# Displays the data stored in the column totlngth (total length of a possum)
```

Generally, we can access parts of objects including vectors, matrices, data.frames and lists with squared brackets:

```
# Select column via name
pos_dat[ , "totlngth"]
```

```
##  [1] 89.0 91.5 95.5 92.0 85.5 90.5 89.5 91.0 91.5 89.5 89.5 92.0 89.5 91.5
## [15] 85.5 86.0 89.5 90.0 90.5 89.0 96.5 91.0 89.0 84.0 91.5 90.0 85.0 87.0
## [29] 88.0 84.0 93.0 94.0 89.0 85.5 85.0 88.0 82.5 80.5 75.0 84.5 83.0 77.0
## [43] 81.0 76.0 81.0 84.0 89.0 85.0 85.0 88.0
##  [ reached getOption("max.print") -- omitted 54 entries ]
```

If you assign the resulting data to a new object, of what class (e.g. list, vector, matrix) is the resulting object?

We can also select rows and columns via column and row numbers:

```
# Select row 1 to 3 of the column 5 and 6
pos_dat[1:3, 5:6]
```

```
##   sex age
## 1   m   8
## 2   f   6
## 3   f   6
```

```
# Select row 1, 3 and 4 of columns 7 and 9
pos_dat[c(1,3,4), c(7,9)]
```

```
##   hdlngth totlngth
## 1    94.1     89.0
## 3    94.0     95.5
## 4    93.2     92.0
```

If we want to store the selected rows and columns, we can simply assign them to a new object:

```
new_obj <- pos_dat[c(1,3,4), c(7,9)]
```

Often we want to subset data based on conditions. If we apply a condition to a vector, we obtain a logical (i.e. TRUE/FALSE) vector:

```
pos_dat$totlngth > 95
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [23] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [34] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [45] FALSE FALSE FALSE FALSE FALSE FALSE
##  [ reached getOption("max.print") -- omitted 54 entries ]
```

```
# TRUE: condition met, FALSE: condition not met
```

To use a condition to select data, we apply the logical vector to an object (e.g. vector or dataframe):

```
log_vector <- pos_dat$totlngth > 95
pos_dat$totlngth[log_vector]
```

```
## [1] 95.5 96.5 96.0
```

```
# Subset variable directly without storing logical vector in object
pos_dat$totlngth[pos_dat$totlngth > 95]
```

```
## [1] 95.5 96.5 96.0
```

```
# Different way to do this
pos_dat[pos_dat$totlngth > 95, "totlngth"]
```

```
## [1] 95.5 96.5 96.0
```

```
# Subset dataframe
pos_dat[pos_dat$totlngth > 95, ]
```

```
##    row.names case site   Pop sex age hdlngth skullw totlngth taill
## 3        C10    3    1   Vic   f   6    94.0   60.0     95.5  39.0
## 21       C54   21    1   Vic   f   3    95.9   58.1     96.5  39.5
## 59       BR6   59    4 other   m   2   102.5   62.8     96.0  40.0
##    footlgth earconch  eye chest belly
## 3      75.4     51.9 15.5    30    34
## 21     77.9     52.9 14.2    30    40
## 59     73.2     44.5 14.7    32    36
```

To query values *smaller or equal than* is done in R via `<=`, to query values *larger or equal than* is done via `>=`. Similarly, `==` means *equal*, and `!=` means *not equal*. We exemplify this by querying selected variables conditioned by the sex of the possums:

```
# Select male possums
pos_dat[pos_dat$sex == "m", c(5,7:9)]
```

```
##    sex hdlngth skullw totlngth
## 1    m    94.1   60.4     89.0
## 7    m    95.3   58.2     89.5
## 13   m    95.1   59.9     89.5
## 14   m    95.4   57.6     91.5
## 15   m    92.9   57.6     85.5
## 16   m    91.6   56.0     86.0
## 18   m    93.5   55.7     90.0
```

```
## 22   m    96.3   58.5     91.0
## 24   m    94.4   54.9     84.0
## 25   m    95.8   58.5     91.5
## 26   m    96.0   59.0     90.0
## 28   m    93.8   56.8     87.0
##  [ reached 'max' / getOption("max.print") -- omitted 49 rows ]
```

```
# Select female possums
pos_dat[pos_dat$sex == "f", c(5,7:9)]
```

```
##    sex hdlngth skullw totlngth
## 2    f    92.5   57.6     91.5
## 3    f    94.0   60.0     95.5
## 4    f    93.2   57.1     92.0
## 5    f    91.5   56.3     85.5
## 6    f    93.1   54.8     90.5
## 8    f    94.8   57.6     91.0
## 9    f    93.4   56.3     91.5
## 10   f    91.8   58.0     89.5
## 11   f    93.3   57.2     89.5
## 12   f    94.9   55.6     92.0
## 17   f    94.7   67.7     89.5
## 19   f    94.4   55.4     90.5
##  [ reached 'max' / getOption("max.print") -- omitted 31 rows ]
```

```
# Select female possums as those not male
pos_dat[pos_dat$sex != "m", c(5,7:9)]
```

```
##    sex hdlngth skullw totlngth
## 2    f    92.5   57.6     91.5
## 3    f    94.0   60.0     95.5
## 4    f    93.2   57.1     92.0
## 5    f    91.5   56.3     85.5
## 6    f    93.1   54.8     90.5
## 8    f    94.8   57.6     91.0
## 9    f    93.4   56.3     91.5
## 10   f    91.8   58.0     89.5
## 11   f    93.3   57.2     89.5
## 12   f    94.9   55.6     92.0
## 17   f    94.7   67.7     89.5
## 19   f    94.4   55.4     90.5
##  [ reached 'max' / getOption("max.print") -- omitted 31 rows ]
```

Sometimes it is necessary to know the row numbers that meet a condition. These can be queried using the `which()` function:

```
which(pos_dat$totlngth > 95)
```

```
## [1]  3 21 59
```

In every scripting or programming language, you often have multiple ways to reach a result. **Try yourself: Store the vector resulting from the 'which' function and use it as condition for subsetting the dataframe pos_dat.**

Of course, you can also combine conditions (& = logical AND, | = logical OR) to subset data:

```
# selects all possums that are male and larger than 95 (cm)
pos_dat[pos_dat$sex == "m" & pos_dat$totlngth > 95, ] # AND
```

```
##    row.names case site   Pop sex age hdlngth skullw totlngth taill
## 59       BR6   59    4 other   m   2   102.5   62.8       96    40
##    footlgth earconch  eye chest belly
## 59     73.2     44.5 14.7    32    36
```

```
# selects all possums that are male or larger than 95 (cm)
pos_dat[pos_dat$sex == "m" | pos_dat$totlngth > 95, ]# OR
```

```
##   row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 1        C3    1    1 Vic   m   8    94.1   60.4     89.0    36     74.5
## 3       C10    3    1 Vic   f   6    94.0   60.0     95.5    39     75.4
## 7       C26    7    1 Vic   m   2    95.3   58.2     89.5    36     71.5
##   earconch  eye chest belly
## 1     54.5 15.2    28  36.0
## 3     51.9 15.5    30  34.0
## 7     52.0 14.2    30  34.5
##  [ reached 'max' / getOption("max.print") -- omitted 60 rows ]
```

The combination of conditions looks complicated. The dplyr package provides functions that are intended to simplify such operations on dataframes. We only outline a few examples for application of dplyr functions, please refer to a blog post and the course material on OpenOLAT for more extensive tutorials.

```
# load dplyr library
library(dplyr)
# Select variables with the select function
# First argument is data set followed by the variable names
select(pos_dat, totlngth, sex, skullw)
```

```
##    totlngth sex skullw
## 1      89.0   m   60.4
## 2      91.5   f   57.6
## 3      95.5   f   60.0
## 4      92.0   f   57.1
## 5      85.5   f   56.3
## 6      90.5   f   54.8
## 7      89.5   m   58.2
## 8      91.0   f   57.6
## 9      91.5   f   56.3
## 10     89.5   f   58.0
## 11     89.5   f   57.2
## 12     92.0   f   55.6
## 13     89.5   m   59.9
## 14     91.5   m   57.6
## 15     85.5   m   57.6
## 16     86.0   m   56.0
##  [ reached 'max' / getOption("max.print") -- omitted 88 rows ]
```

```
# Note: No need for quotation marks.
# Select rows with the filter function
filter(pos_dat, totlngth > 95)
```

```
##   row.names case site   Pop sex age hdlngth skullw totlngth taill footlgth
## 1       C10    3    1   Vic   f   6    94.0   60.0     95.5  39.0     75.4
## 2       C54   21    1   Vic   f   3    95.9   58.1     96.5  39.5     77.9
## 3       BR6   59    4 other   m   2   102.5   62.8     96.0  40.0     73.2
##   earconch  eye chest belly
## 1     51.9 15.5    30    34
```

```
## 2     52.9 14.2    30    40
## 3     44.5 14.7    32    36
```

```
filter(pos_dat, sex == "m")
```

```
##   row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 1        C3    1    1 Vic   m   8    94.1   60.4     89.0    36     74.5
## 2       C26    7    1 Vic   m   2    95.3   58.2     89.5    36     71.5
## 3       C36   13    1 Vic   m   5    95.1   59.9     89.5    36     71.0
##   earconch  eye chest belly
## 1     54.5 15.2    28  36.0
## 2     52.0 14.2    30  34.5
## 3     49.8 15.8    27  32.0
##  [ reached 'max' / getOption("max.print") -- omitted 58 rows ]
```

```
# Combine conditions
filter(pos_dat, totlngth > 95 & sex == "m")
```

```
##   row.names case site   Pop sex age hdlngth skullw totlngth taill footlgth
## 1       BR6   59    4 other   m   2   102.5   62.8       96    40     73.2
##   earconch  eye chest belly
## 1     44.5 14.7    32    36
```

We can also combine `select()` and `filter()`:

```
select(filter(pos_dat, sex == "m"), totlngth, sex, skullw)
```

```
##    totlngth sex skullw
## 1      89.0   m   60.4
## 2      89.5   m   58.2
## 3      89.5   m   59.9
## 4      91.5   m   57.6
## 5      85.5   m   57.6
## 6      86.0   m   56.0
## 7      90.0   m   55.7
## 8      91.0   m   58.5
## 9      84.0   m   54.9
## 10     91.5   m   58.5
## 11     90.0   m   59.0
## 12     87.0   m   56.8
## 13     93.0   m   54.1
## 14     89.0   m   54.6
## 15     85.5   m   55.7
## 16     85.0   m   57.9
##  [ reached 'max' / getOption("max.print") -- omitted 45 rows ]
```

In this example, the output of `filter()` takes the position of the *data* argument in the `select()` function. A particular strength of dplyr is the use of pipelines, defined in the R context as a sequence of functions, where the output from one function feeds directly as input of the next function. This can also enhance readability. Consider for example the previous code (combination of `select()` and `filter()`) rewritten as pipe (pipe operator: %>%):

```
pos_dat %>%
  filter(sex == "m") %>%
  select(totlngth, sex, skullw)
```

```
##    totlngth sex skullw
## 1      89.0   m   60.4
```

```
## 2      89.5   m   58.2
## 3      89.5   m   59.9
## 4      91.5   m   57.6
## 5      85.5   m   57.6
## 6      86.0   m   56.0
## 7      90.0   m   55.7
## 8      91.0   m   58.5
## 9      84.0   m   54.9
## 10     91.5   m   58.5
## 11     90.0   m   59.0
## 12     87.0   m   56.8
## 13     93.0   m   54.1
## 14     89.0   m   54.6
## 15     85.5   m   55.7
## 16     85.0   m   57.9
##  [ reached 'max' / getOption("max.print") -- omitted 45 rows ]
```

dplyr also provides a useful function (`arrange()`) to sort a dataframe according to selected variables:

```
pos_dat %>%
  arrange(totlngth)
```

```
##   row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 1      BB31   39    2 Vic   f   1    84.7   51.5       75  34.0     68.7
## 2      BB41   44    2 Vic   m  NA    85.1   51.5       76  35.5     70.3
## 3      BB38   42    2 Vic   m   3    85.3   54.1       77  32.0     62.7
##   earconch  eye chest belly
## 1     53.4 13.0  25.0    25
## 2     52.6 14.4  23.0    27
## 3     51.2 13.8  25.5    33
##  [ reached 'max' / getOption("max.print") -- omitted 101 rows ]
```

```
# now in descending order
pos_dat %>%
  arrange(desc(totlngth))
```

```
##   row.names case site   Pop sex age hdlngth skullw totlngth taill footlgth
## 1       C54   21    1   Vic   f   3    95.9   58.1     96.5  39.5     77.9
## 2       BR6   59    4 other   m   2   102.5   62.8     96.0  40.0     73.2
## 3       C10    3    1   Vic   f   6    94.0   60.0     95.5  39.0     75.4
##   earconch  eye chest belly
## 1     52.9 14.2    30    40
## 2     44.5 14.7    32    36
## 3     51.9 15.5    30    34
##  [ reached 'max' / getOption("max.print") -- omitted 101 rows ]
```

Compare this to the sorting of dataframes in basic R, which is much less elegant:

```
ord_1 <- order(pos_dat[ ,"totlngth"])
pos_dat[ord_1, ]
```

```
##    row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 39      BB31   39    2 Vic   f   1    84.7   51.5       75  34.0     68.7
## 44      BB41   44    2 Vic   m  NA    85.1   51.5       76  35.5     70.3
## 42      BB38   42    2 Vic   m   3    85.3   54.1       77  32.0     62.7
##    earconch  eye chest belly
## 39     53.4 13.0  25.0    25
```

```
## 44     52.6 14.4  23.0    27
## 42     51.2 13.8  25.5    33
##  [ reached 'max' / getOption("max.print") -- omitted 101 rows ]
```

We can also easily sort by multiple columns and afterwards select a few variables:

```
pos_dat %>%
  arrange(age, desc(totlngth)) %>%
  select(age, totlngth, belly)
```

```
##    age totlngth belly
## 1    1     90.5  32.0
## 2    1     89.5  31.0
## 3    1     84.0  30.5
## 4    1     82.5  33.0
## 5    1     82.0  28.0
## 6    1     81.5  27.0
## 7    1     81.0  28.0
## 8    1     81.0  28.5
## 9    1     80.5  30.0
## 10   1     75.0  25.0
## 11   2     96.0  36.0
## 12   2     93.0  36.0
## 13   2     92.5  36.0
## 14   2     90.0  32.0
## 15   2     89.5  34.5
## 16   2     89.0  33.0
##  [ reached 'max' / getOption("max.print") -- omitted 88 rows ]
```

Another useful function is `rename()`:

```
pos_dat %>%
  rename(total_length = totlngth)
```

```
##   row.names case site Pop sex age hdlngth skullw total_length taill
## 1        C3    1    1 Vic   m   8    94.1   60.4         89.0  36.0
## 2        C5    2    1 Vic   f   6    92.5   57.6         91.5  36.5
## 3       C10    3    1 Vic   f   6    94.0   60.0         95.5  39.0
##   footlgth earconch  eye chest belly
## 1     74.5     54.5 15.2  28.0    36
## 2     72.5     51.2 16.0  28.5    33
## 3     75.4     51.9 15.5  30.0    34
##  [ reached 'max' / getOption("max.print") -- omitted 101 rows ]
```

```
# We inspect the original dataframe
head(pos_dat)
```

```
##   row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 1        C3    1    1 Vic   m   8    94.1   60.4     89.0  36.0     74.5
## 2        C5    2    1 Vic   f   6    92.5   57.6     91.5  36.5     72.5
## 3       C10    3    1 Vic   f   6    94.0   60.0     95.5  39.0     75.4
##   earconch  eye chest belly
## 1     54.5 15.2  28.0    36
## 2     51.2 16.0  28.5    33
## 3     51.9 15.5  30.0    34
##  [ reached 'max' / getOption("max.print") -- omitted 3 rows ]
```

Why is the original name still in the dataframe? **What would you need to do, to keep the changed**

name?

Finally, the mutate function allows to create new columns, for example, as a function of existing columns:

```
pos_dat %>%
  mutate(Sum_hdln_totlng = hdlngth + totlngth)
```

```
##   row.names case site Pop sex age hdlngth skullw totlngth taill footlgth
## 1        C3    1    1 Vic   m   8    94.1   60.4     89.0  36.0     74.5
## 2        C5    2    1 Vic   f   6    92.5   57.6     91.5  36.5     72.5
## 3       C10    3    1 Vic   f   6    94.0   60.0     95.5  39.0     75.4
##   earconch  eye chest belly Sum_hdln_totlng
## 1     54.5 15.2  28.0    36           183.1
## 2     51.2 16.0  28.5    33           184.0
## 3     51.9 15.5  30.0    34           189.5
##  [ reached 'max' / getOption("max.print") -- omitted 101 rows ]
```

```
# Combined with subsetting to a few columns
pos_dat %>%
  mutate(Sum_hdln_totlng = hdlngth + totlngth) %>%
  select(sex, age, hdlngth, totlngth, Sum_hdln_totlng)
```

```
##    sex age hdlngth totlngth Sum_hdln_totlng
## 1    m   8    94.1     89.0           183.1
## 2    f   6    92.5     91.5           184.0
## 3    f   6    94.0     95.5           189.5
## 4    f   6    93.2     92.0           185.2
## 5    f   2    91.5     85.5           177.0
## 6    f   1    93.1     90.5           183.6
## 7    m   2    95.3     89.5           184.8
## 8    f   6    94.8     91.0           185.8
## 9    f   9    93.4     91.5           184.9
## 10   f   6    91.8     89.5           181.3
##  [ reached 'max' / getOption("max.print") -- omitted 94 rows ]
```

Still, you would need to assign this to a new object to store the changes.

Data exploration

After we have learnt how to process data, we explore the data that will be analysed in the next session. Although graphical tools are most suitable to obtain an overview on data, the `summary()` function quickly provides information on potential outliers, missing values (*NA's*) and the range of data:

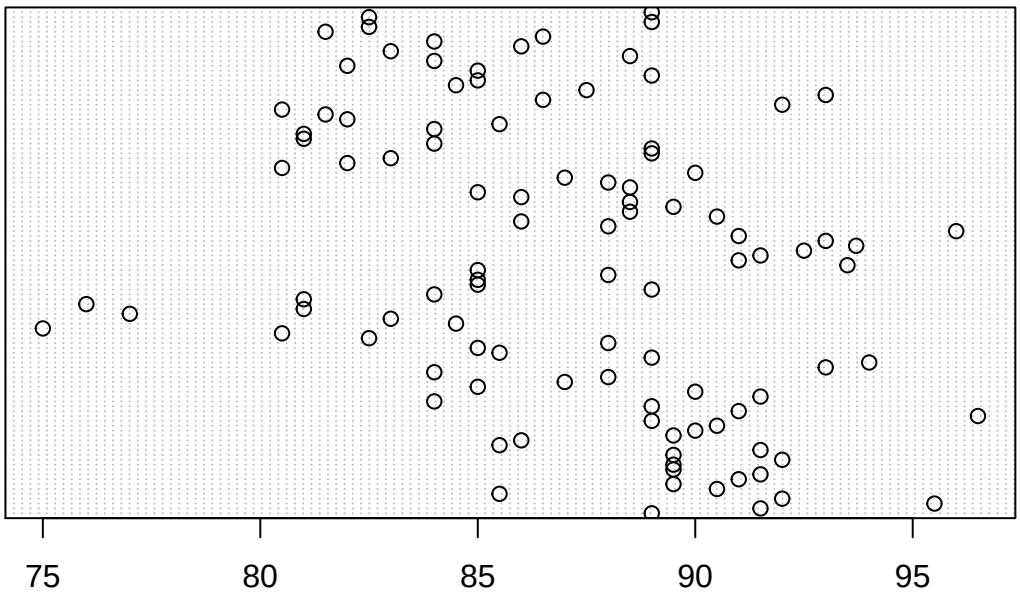
```
# Reset max.print options to 100 to avoid that information is omitted
options(max.print = 120)
summary(pos_dat)
```

```
##   row.names              case             site          Pop     sex
##  Length:104         Min.   :  1.00   Min.   :1.000   other:58   f:43
##  Class :character   1st Qu.: 26.75   1st Qu.:1.000   Vic  :46   m:61
##  Mode  :character   Median : 52.50   Median :3.000
##                     Mean   : 52.50   Mean   :3.625
##                     3rd Qu.: 78.25   3rd Qu.:6.000
##                     Max.   :104.00   Max.   :7.000
##
##       age            hdlngth           skullw         totlngth
##  Min.   :1.000   Min.   : 82.50   Min.   :50.00   Min.   :75.00
```

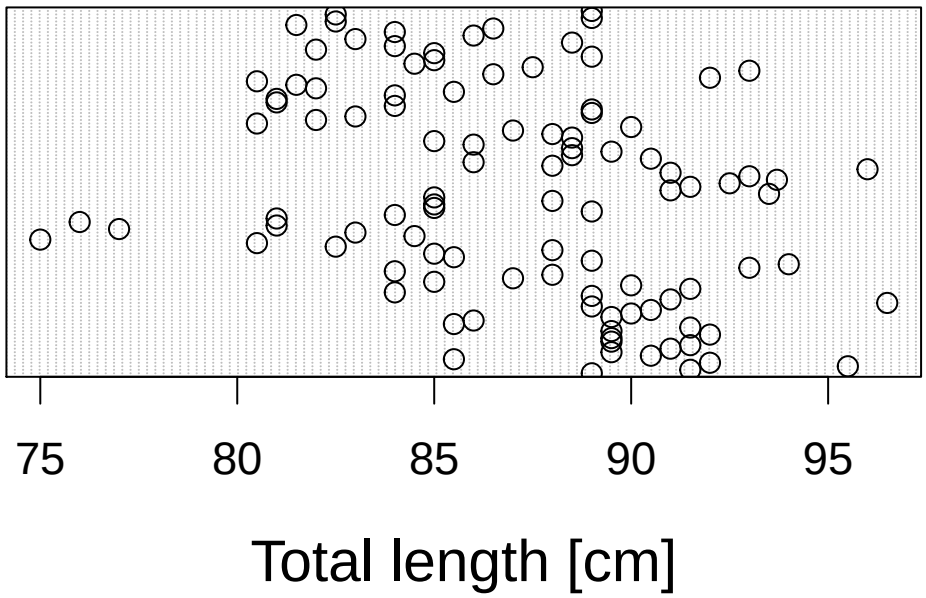
```
##  1st Qu.:2.250   1st Qu.: 90.67   1st Qu.:54.98   1st Qu.:84.00
##  Median :3.000   Median : 92.80   Median :56.35   Median :88.00
##  Mean   :3.833   Mean   : 92.60   Mean   :56.88   Mean   :87.09
##  3rd Qu.:5.000   3rd Qu.: 94.72   3rd Qu.:58.10   3rd Qu.:90.00
##  Max.   :9.000   Max.   :103.10   Max.   :68.60   Max.   :96.50
##  NA's   :2
##      taill          footlgth        earconch          eye
##  Min.   :32.00   Min.   :60.30   Min.   :40.30   Min.   :12.80
##  1st Qu.:35.88   1st Qu.:64.60   1st Qu.:44.80   1st Qu.:14.40
##  Median :37.00   Median :68.00   Median :46.80   Median :14.90
##  Mean   :37.01   Mean   :68.46   Mean   :48.13   Mean   :15.05
##  3rd Qu.:38.00   3rd Qu.:72.50   3rd Qu.:52.00   3rd Qu.:15.72
##  Max.   :43.00   Max.   :77.90   Max.   :56.20   Max.   :17.80
##                  NA's   :1
##      chest          belly
##  Min.   :22.0   Min.   :25.00
##  1st Qu.:25.5   1st Qu.:31.00
##  Median :27.0   Median :32.50
##  Mean   :27.0   Mean   :32.59
##  3rd Qu.:28.0   3rd Qu.:34.12
##  Max.   :32.0   Max.   :40.00
##
```

For the categorical variables *Pop* and *sex* the function returns the number of cases per level. For numerical variables, the minimum, maximum, quartiles, median and mean are returned. In the following we use the cleveland plot and boxplot to check for potential errors and outliers. Let us first look at a cleveland plot:

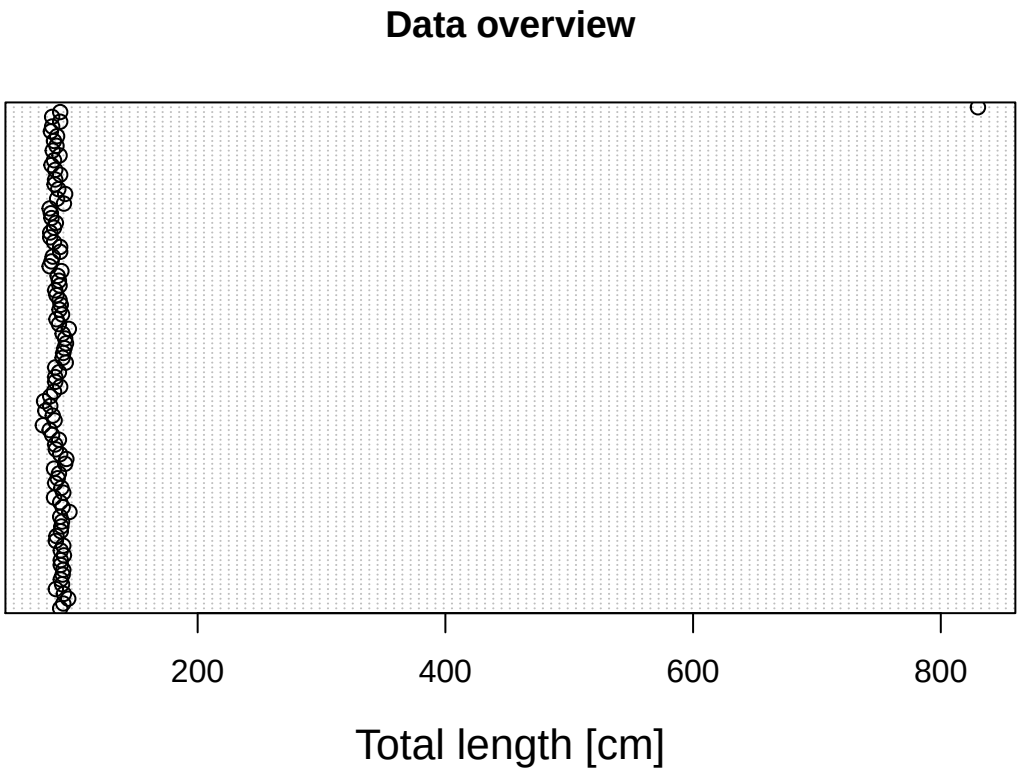
```
dotchart(pos_dat$totlngth)
```




```
# Provides an overview, but plot would benefit from polishing.
# Increase font size of labels and symbols
par(cex = 1.4)
# Check ?par for explanation and overview of other arguments
dotchart(pos_dat$totlngth,cex.lab = 1.3,
     xlab = "Total length [cm]", main = "Data overview")
```

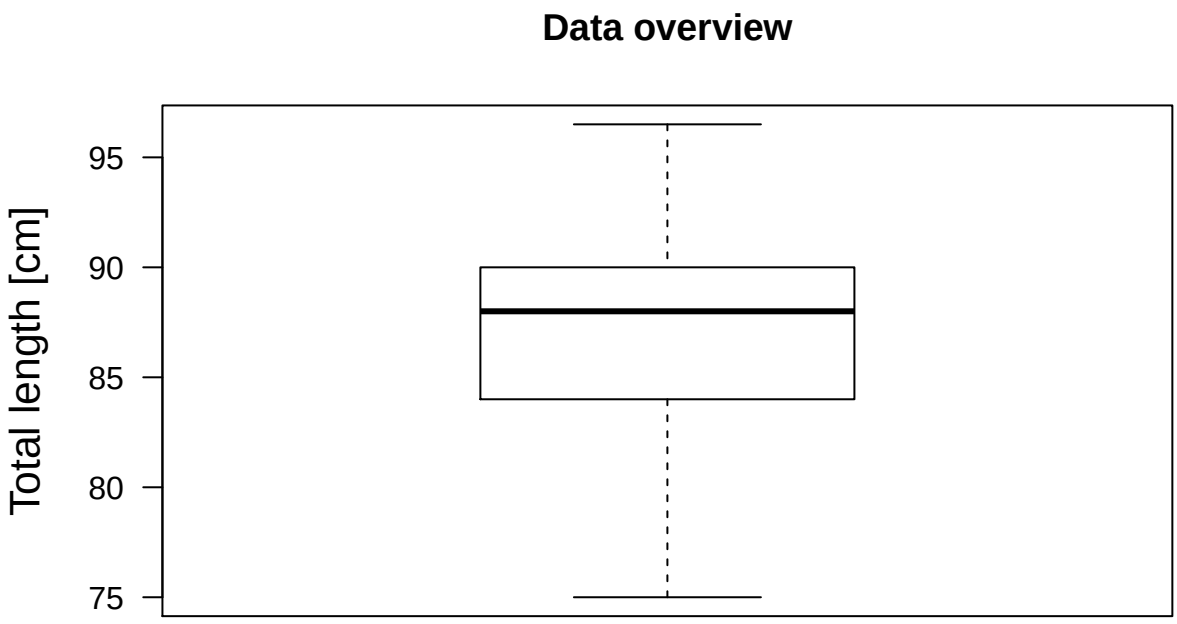


No outlier is visible. This is how an outlier would look like:

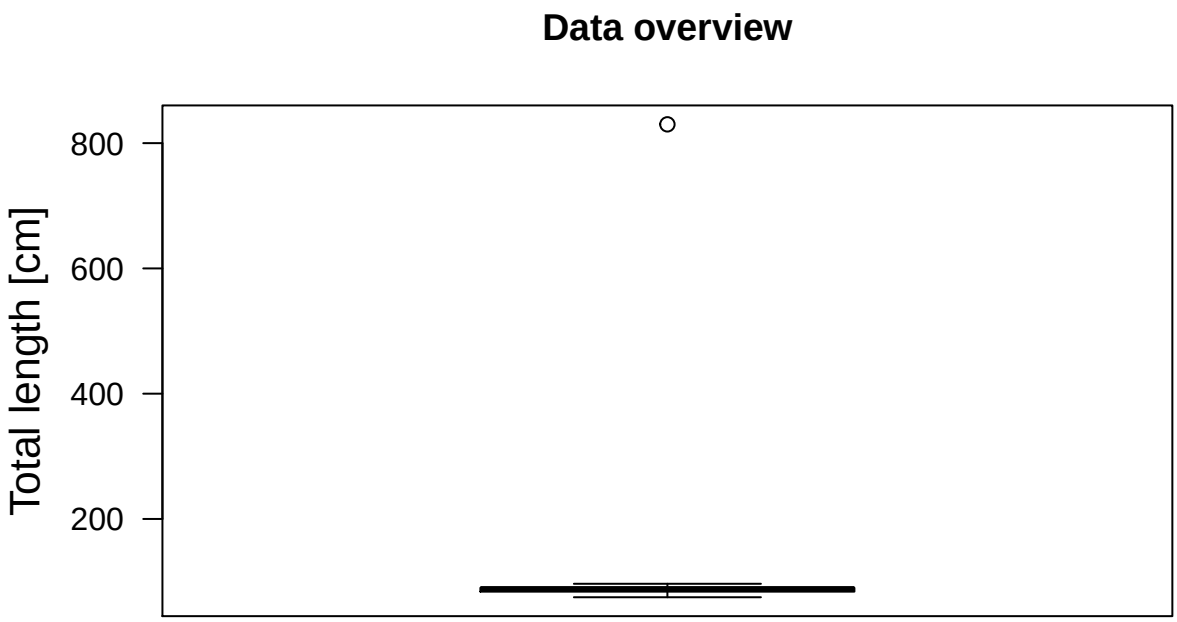


The observation at 830 is an extreme outlier. If you spot such an extreme difference to the remainder of the data, you should scrutinise the raw data, because an order of magnitude difference points to an error with a decimal point during data entry. Another useful tool that can be used for different purposes including checking for outliers is the boxplot (for brief explanation and different types visit the R graph gallery).

```
boxplot(pos_dat$totlngth, las = 1, cex.lab = 1.3,
        ylab = "Total length [cm]", main = "Data overview")
```




```
# For the same variable with an added outlier
boxplot(totlng_outl, las = 1, cex.lab = 1.3,
        ylab = "Total length [cm]", main = "Data overview")
```



You can save a figure from the graphics device using the graphical user interface (*Export* in R Studio). Direct export of a figure without plotting in R Studio can be done using specific functions such as `jpeg()`, `png()` or `pdf()`. For details see `?jpeg` and `?pdf`

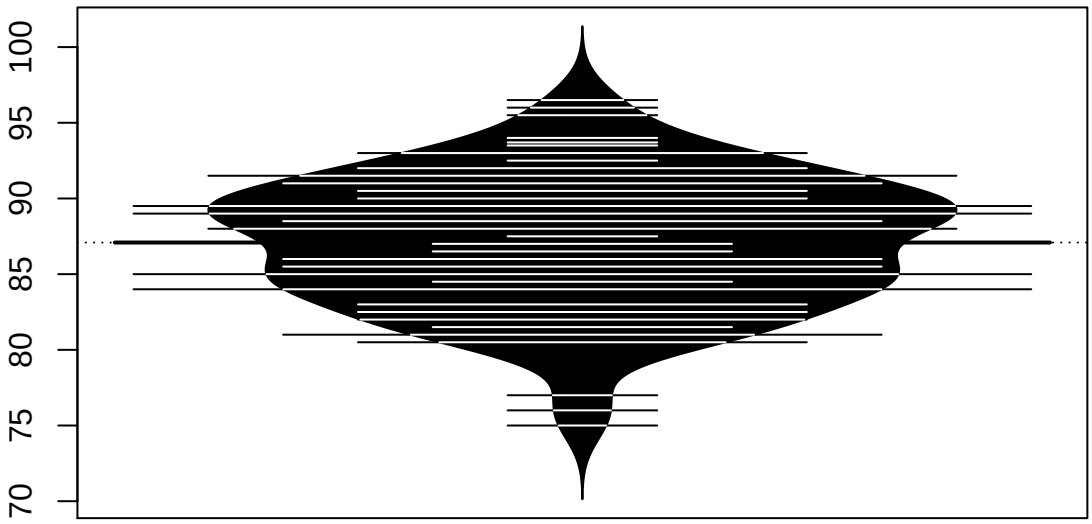
```
# example for directly exporting the boxplot to a file
jpeg("Boxplot.jpeg", quality = 100)
par(cex = 1.4)
boxplot(pos_dat$totlngth, las = 1, cex.lab = 1.3,
        ylab = "Total length [cm]", main = "Data overview")
dev.off()
# Switches off the device (here: saves content to file in working directory)
```

Although the boxplot is widely used and you should be familiar with its interpretation, interesting alternatives such as the beanplot have been introduced (Kampstra 2008). The related paper is available via open access. Refer to the paper and the lecture for explanation of the plot. We first load and attach the package:

```
library(beanplot)
```

Now we create a beanplot for the total lenght of possums

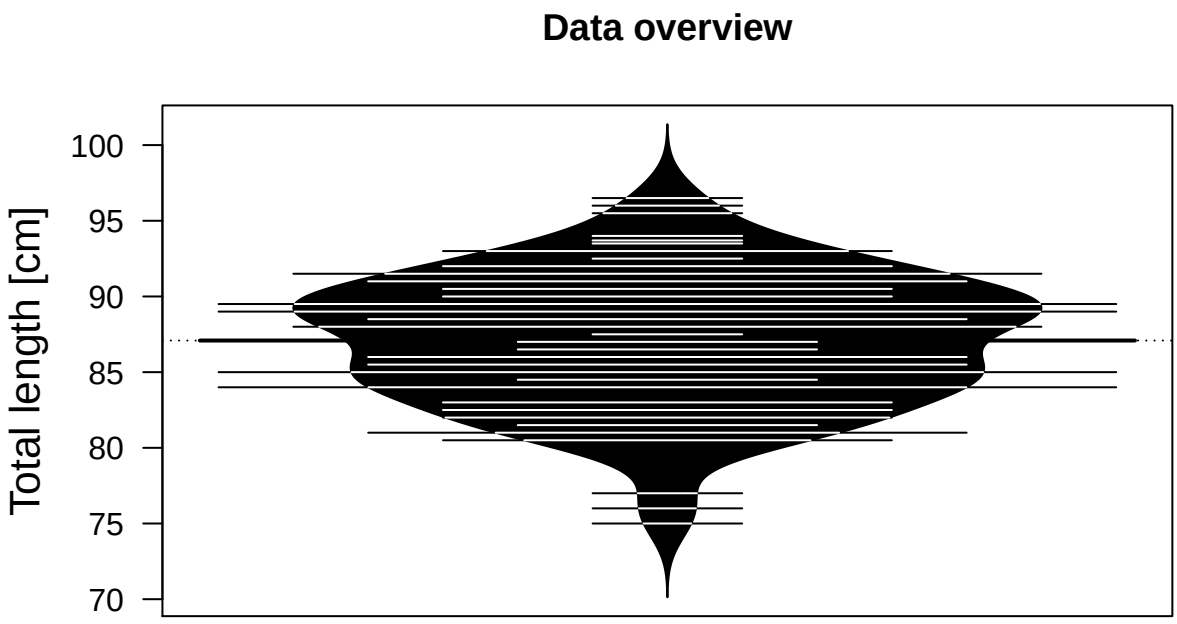
```
beanplot(pos_dat$totlngth)
```




```
# provides single observations (lines), mean (thick line) and
# displays the estimated probability density distribution (black polygon)
```

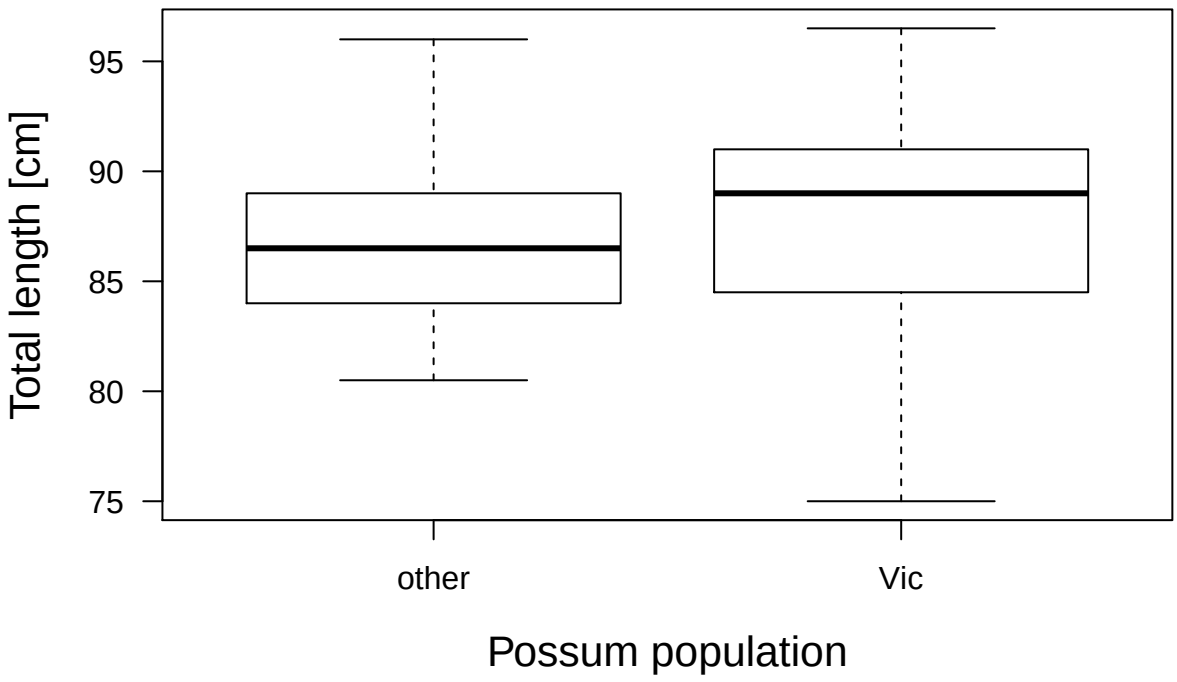
Again, a few additional arguments improve the quality of the figure. It is quite handy that these arguments are the same as for the boxplot or dotchart:

```
beanplot(pos_dat$totlngth, las = 1, cex.lab = 1.3,
         ylab = "Total length [cm]", main = "Data overview")
```



Both the boxplot and beanplot are practical tools to compare the distribution among variables or groups. Several statistical tests for between group differences (e.g. *t*-test or analysis of variance) require that the variance is homogeneous across groups. This homogeneity of variance translates to a similar spread of the data around the mean. Examplarily, we inspect the distribution of possums from Victoria and from other states with a conditional boxplot:

```
boxplot(totlngth ~ Pop, data = pos_dat, las = 1, cex.lab = 1.3,
        ylab = "Total length [cm]", xlab = "Possum population")
```



The *tilde* sign ~ is used in R to formulate statistical models, where the response variable(s) are on the left hand side and the explanatory variables/predictors on the right hand side. Here, the notation results in the plotting of the responses per factor level of the variable *Pop*.

To ease visual comparison of the spread around the median, we put both variables on the same median. This can be done with the base R function `tapply()` that applies a function to each group of data defined by the levels of a factor. We calculate the median for each group and assign it to the object *med*:

```
med <- tapply(pos_dat$totlngth, pos_dat$Pop, median)
# first argument: data, second argument: factor, third argument: function
# to be applied to each group defined by the factor
med
```

```
## other   Vic
##  86.5  89.0
```

The same calculation can be done using dplyr functions. We need two new functions `group_by()` and `summarise()` to do this elegantly. **Check what is done by calling the help for the new functions and by sequential execution of the code below (i.e. first execute the code until second %>% (not included), then until third %>% (not included)):**

```
pos_dat %>%
  group_by(Pop) %>%
  select(totlngth, Pop) %>%
  summarise(med = median(totlngth))
```

```
## # A tibble: 2 x 2
##   Pop     med
##   <fct> <dbl>
```

```
## 1 other  86.5
## 2 Vic    89
```

Note that a different type of object is produced using dplyr than when using the base R function above: a so-called tibble.

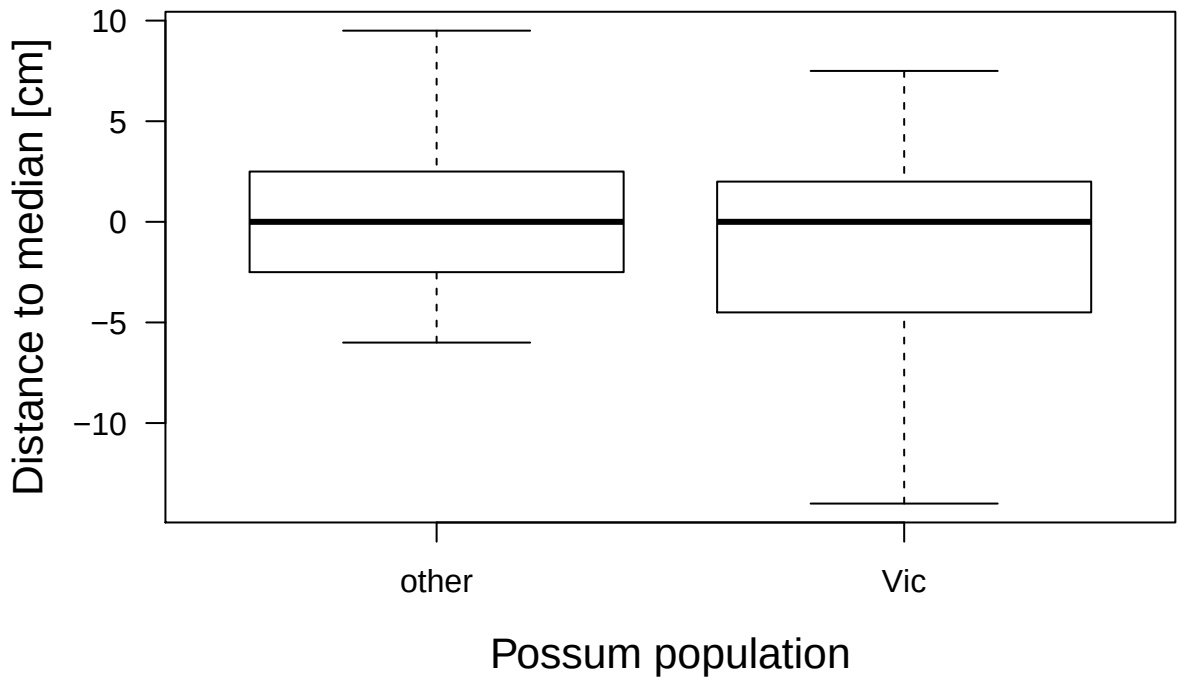
Anyway, we proceed within the framework of base R and now substract the respective median from each observation of the groups.

```
w <- pos_dat$totlngth - med[pos_dat$Pop]
```

Call the object *med* and check how it is modified by *med[pos_dat$Pop]*.

The resulting vector w gives the distance of each observation to the median of its group. This is equivalent to setting the median of both data sets to 0. We plot again:

```
boxplot(w ~ Pop, data = pos_dat, las = 1, cex.lab = 1.3,
        ylab = "Distance to median [cm]", xlab = "Possum population")
```



The variance looks slightly higher in Victoria, but such a small difference would be irrelevant (i.e. not violate the assumption of same variance of statistical tests). If this was a serious analysis, we would also need to account for the sex of possums as this is known to influence the body length. Ignoring the sex in the analysis could lead to flawed inference if for example the proportion of females differs between the two sample populations *Pop* (see the Chapter *Sample and Population* in the document *Key terms and concepts*, in particular the explanation of the *Simpson's paradox*). Indeed, the proportion of females differs strongly between possums measured in Victoria (*Vic*) and other states (*other*):

```
pos_dat %>%
  select(sex, Pop) %>%
```

```
  group_by(Pop, sex) %>%
  count
```

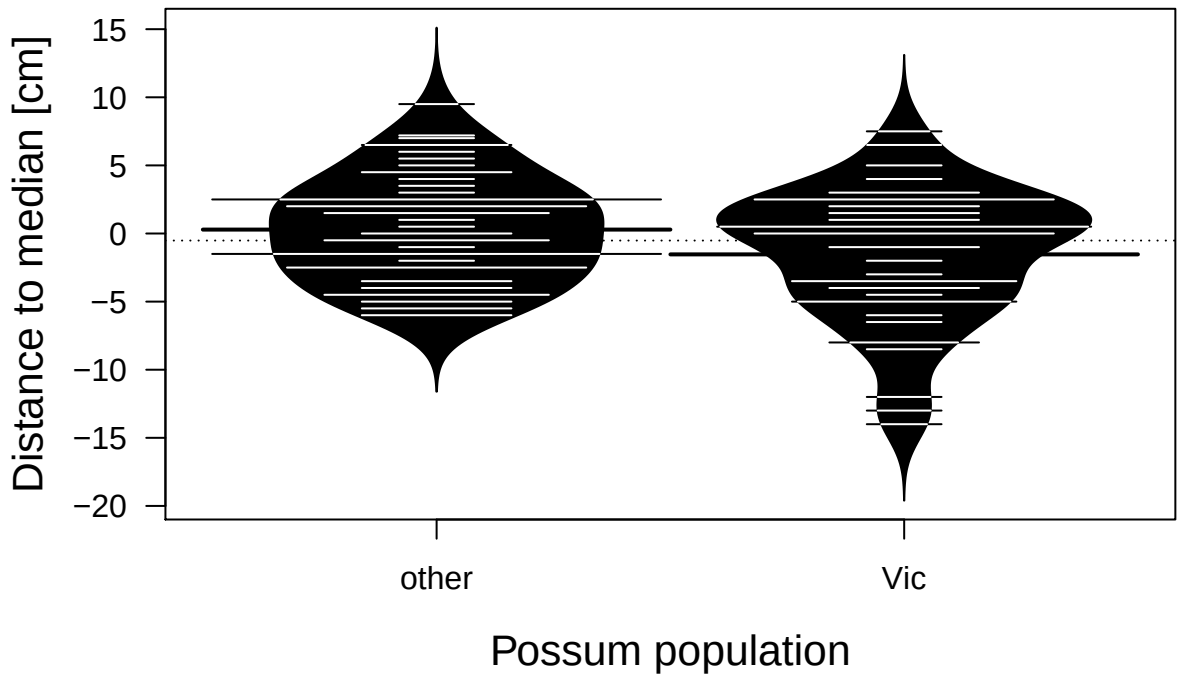
```
## # A tibble: 4 x 3
## # Groups:   Pop, sex [4]
##   Pop   sex       n
##   <fct> <fct> <int>
## 1 other f        19
## 2 other m        39
## 3 Vic   f        24
## 4 Vic   m        22
```

```
# more than 50% females in Victoria, but only 1/3 in other states.
```

A question you may still have is: *When should we be concerned about a difference in the variance?* We discuss this question in the context of the respective method as there is no general answer.

We can also use the beanplot for comparison:

```
beanplot(w ~ Pop, data = pos_dat, las = 1, cex.lab = 1.3,
        ylab = "Distance to median [cm]", xlab = "Possum population")
```

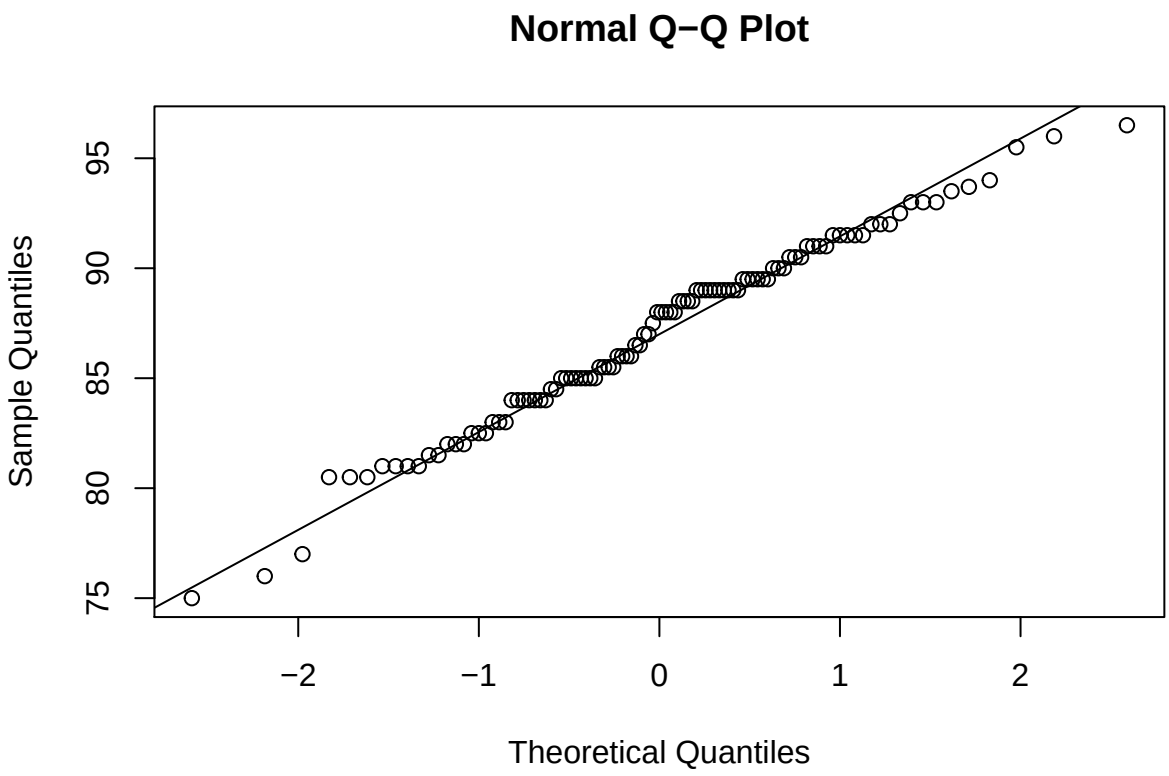


The Victorian population is less symmetric than the population from other states, presumably due to the higher proportion of females in the population (superimposition of two distributions). Note that we can again use the same arguments for the boxplot and beanplot function. Quite convenient, isn't it? However, the beanplot displays the mean instead of median. **Try to produce the same plot with the mean substracted from each observation!** *Hint: the function to calculate the mean is* `mean()`.

Another assumption of several data analysis tools is that the data is normally distributed. This can be

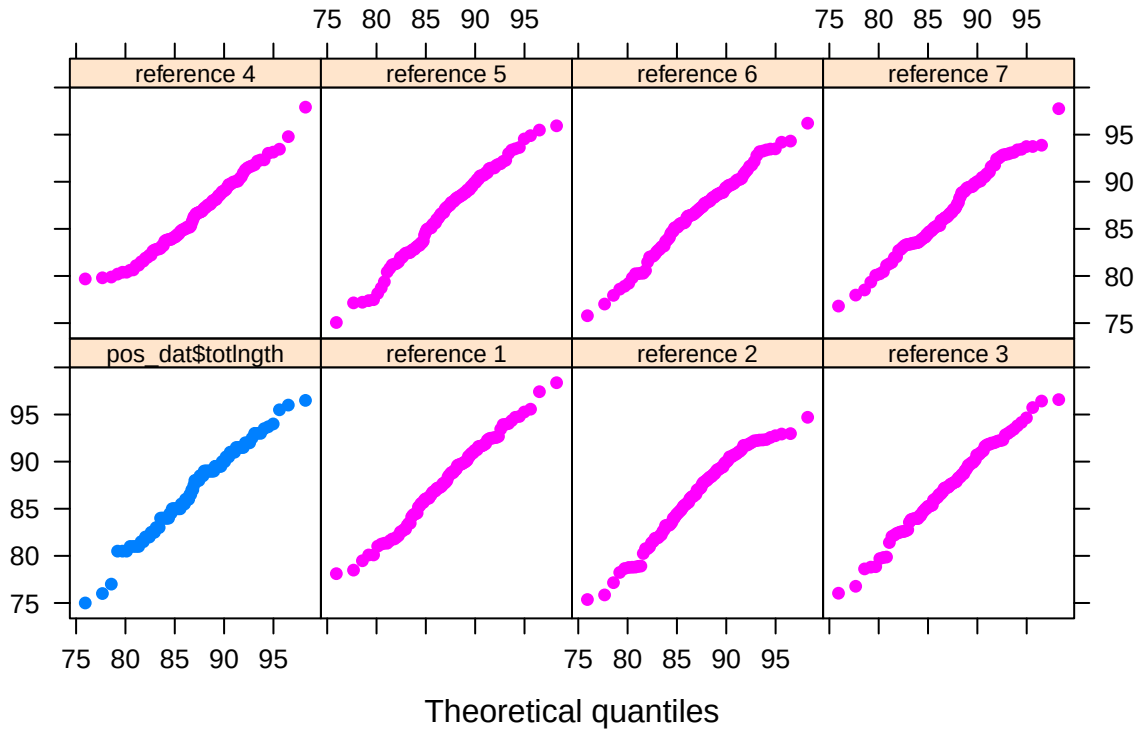
checked using the so-called *QQ-plot*, which plots theoretical quantiles from a normal distribution against the sample Quantiles (the definition and interpretation of quantiles is explained in the lecture). If the data originate from a normal distribution, the points should approximately fall on a 1:1 line. For statistical tests focusing on between-group differences, the assumption would need to be checked for each group. Here, we ignore any potential grouping structure of the data and exemplify the QQ-plot for the variable total length:

```
# Quantile-quantile plot
qqnorm(pos_dat$totlngth)
# We add a line that goes through the first and third quartiles,
# which helps to spot deviations.
qqline(pos_dat$totlngth)
```



The deviations here are minor and can be ignored. Again, you may ask: *When should we be concerned about a deviation?* A helpful function in this context is `qreference()` provided in the package DAAG, which relates to the book by Maindonald & Braun (2010). It produces reference plots to aid in the evaluation whether the data are normally distributed. The reference plots are based on sample quantiles from data that has been generated through random sampling from a normal distribution with the same parameters (sample mean and sample variance) and sample size as the data under evaluation.

```
library(DAAG)
qreference(pos_dat$totlngth, nrep = 8, xlab = "Theoretical quantiles")
```




```
# nrep controls the number of reference plots
```

The reference plots give an idea how data randomly sampled from a normal distribution can deviate from the theoretical normal distribution. Clearly, the data (blue) does not look conspicuous when compared to the reference plots (purple).

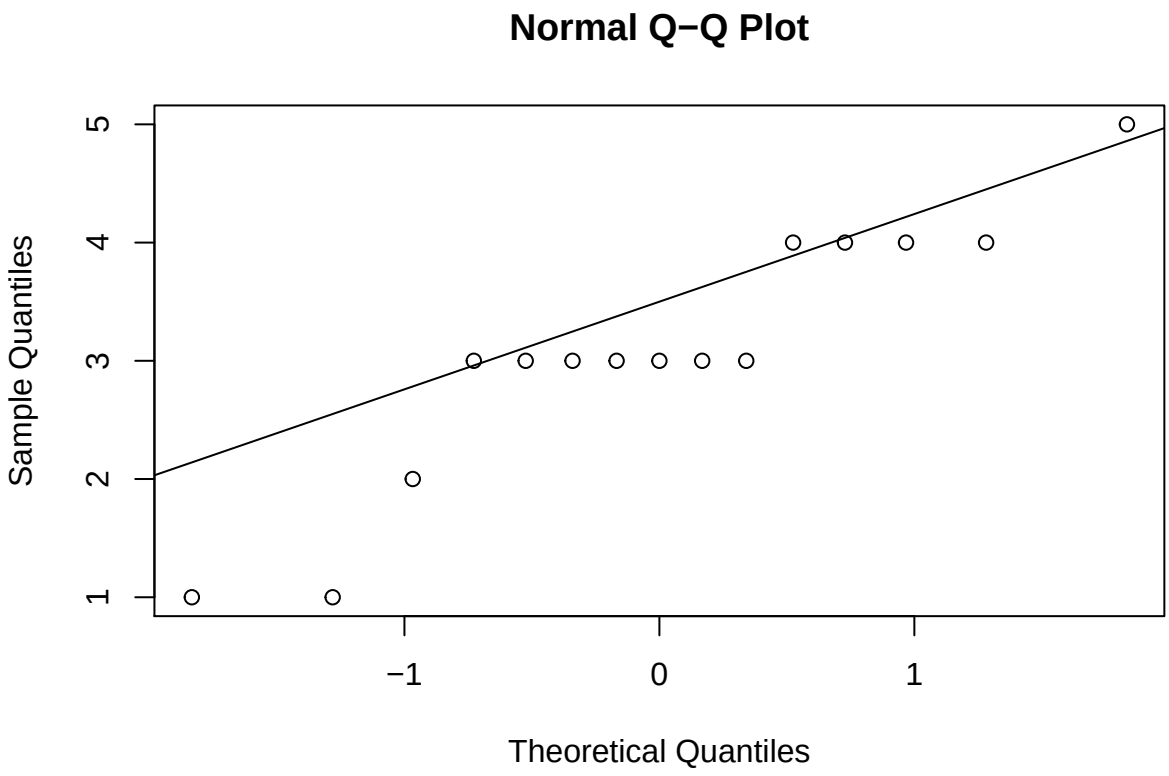
A similar approach is to plot the QQ-plot for the data among reference QQ-plots without indication in the figure, which of the QQ-plots relates to the data. Unless the data deviate strongly from a normal distribution, the QQ-plot related to the data is presumably indistinguishable from the reference plots. Hence, if you cannot identify the QQ-plot related to the data, there is no need for concern regarding a potential deviation from the normal distribution. The code for this approach is provided in a blog.

How would a strong deviation look? We discuss two examples. First, we draw samples from a binomial distribution:

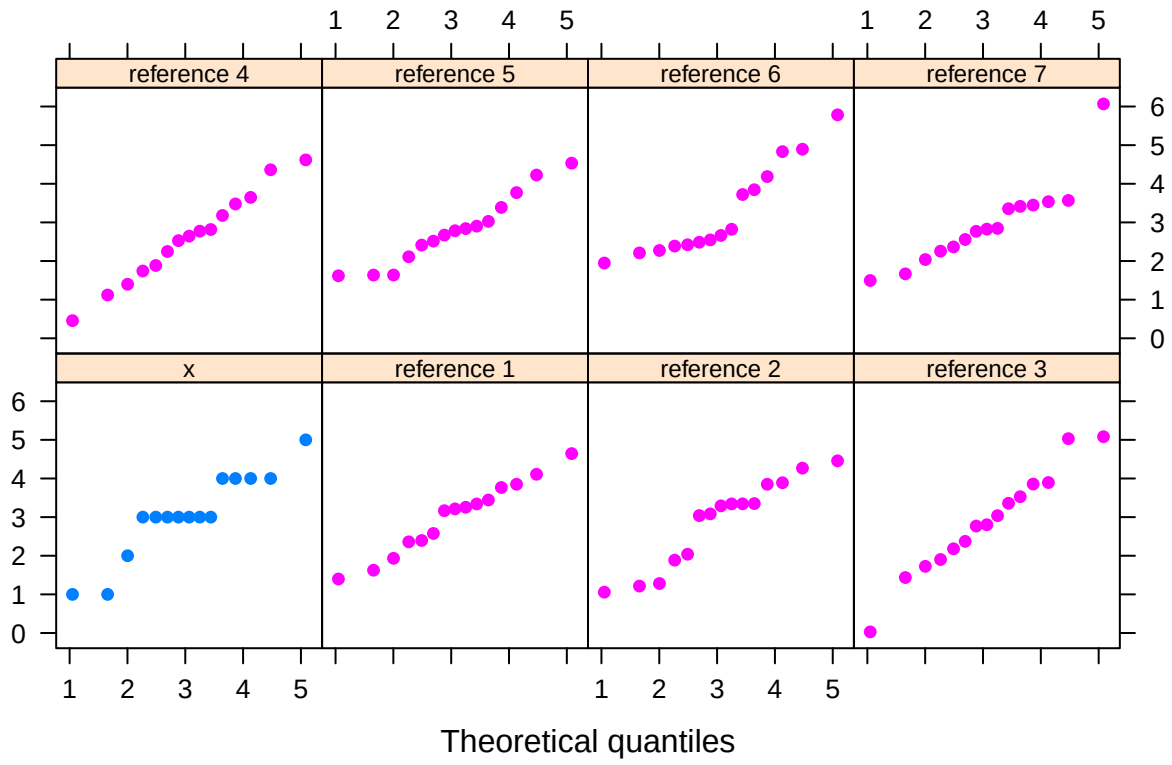
```
set.seed(2018)
# set.seed is used for reproducibility
# i.e. the function returns the same random numbers
x <- rbinom(n = 15, size = 5, p = 0.6)
# n = sample size, size = number of trials
# p = probability of success
```

We use the QQ plot to evaluate for normal distribution:

```
qqnorm(x)
# Strong deviation
qqline(x)
```




```
# The deviation is particularly obvious when compared to reference plots
qreference(x, nrep = 8, xlab = "Theoretical quantiles")
```

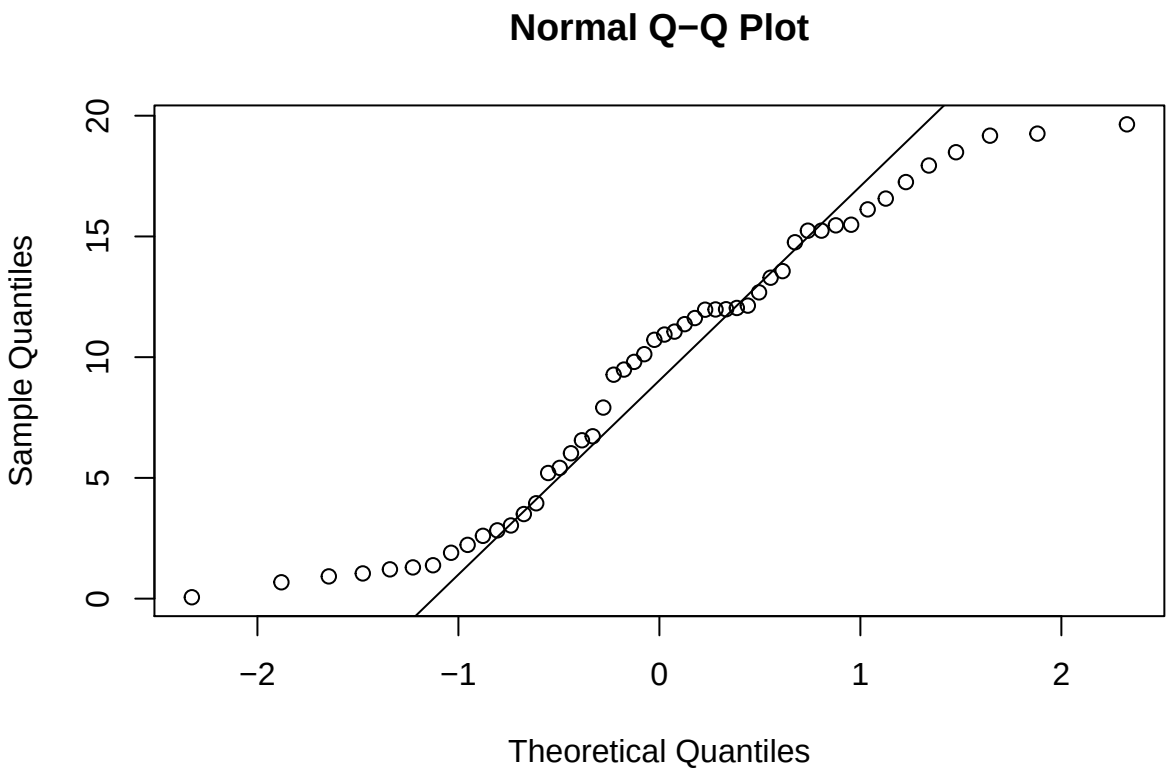


Even if the data would deviate less from the 1:1 line, the data clearly comes from a discrete distribution, whereas the normal distribution is continuous. In the second example, we draw samples from a uniform distribution.

```
set.seed(2018)
y <- runif(50, min = 0, max = 20)
```

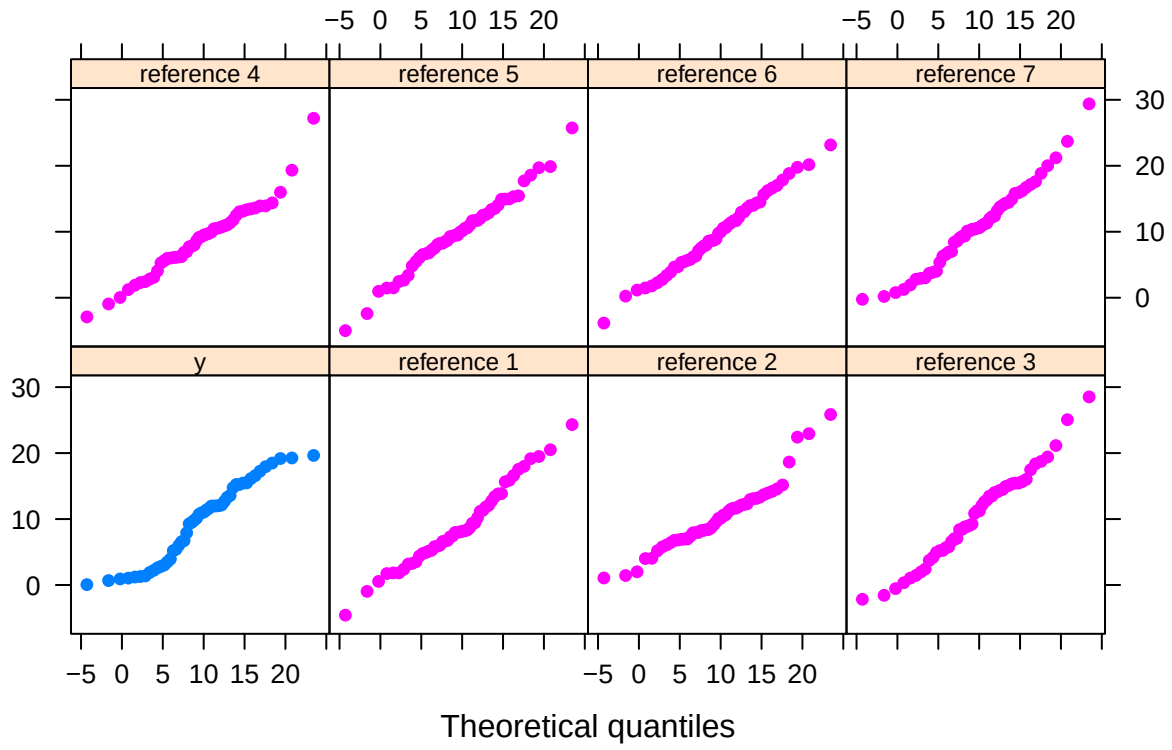
Again, we use the QQ plot to evaluate for normal distribution:

```
qqnorm(y)
qqline(y)
```



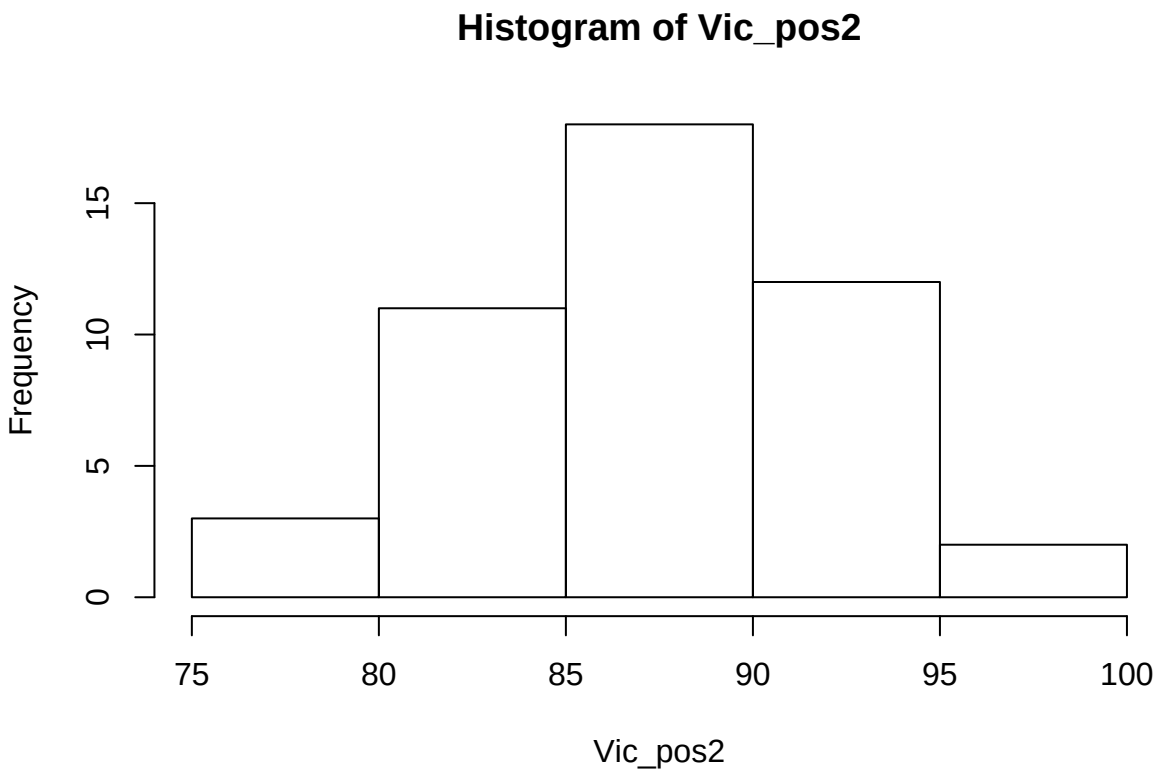
A strong deviation is visible, particularly for the lower and upper quantiles. This impression is confirmed when comparing the QQ-plot for the data to reference QQ-plots. A much stronger curvature is visible.

```
qreference(y, nrep = 8, xlab = "Theoretical quantiles")
```



Another useful tool is the histogram. It can be used to check normality of the data, symmetry and whether the data is bi- or multi-modal. Typically, the histogram displays the frequency with which values of the data fall into, typically same-sized, intervals. For example, we plot a histogram for the total length of Victorian possum populations:

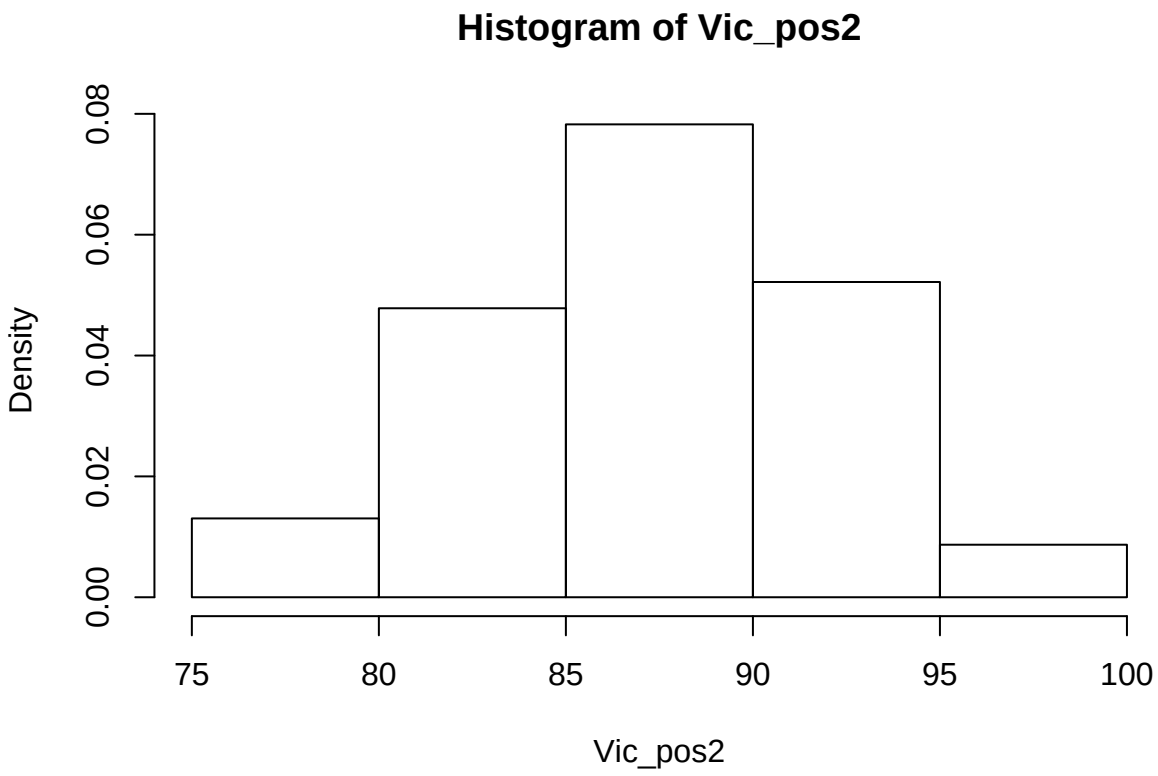
```
# extract Victorian possum data for variable total length
Vic_pos <- pos_dat %>%
              filter(Pop == "Vic") %>%
              select(totlngth)
# convert to vector, otherwise histogram function throws an error
Vic_pos2 <- as.vector(t(Vic_pos))
# create histogram
hist(Vic_pos2)
```




```
# Intervals have a width of 5
```

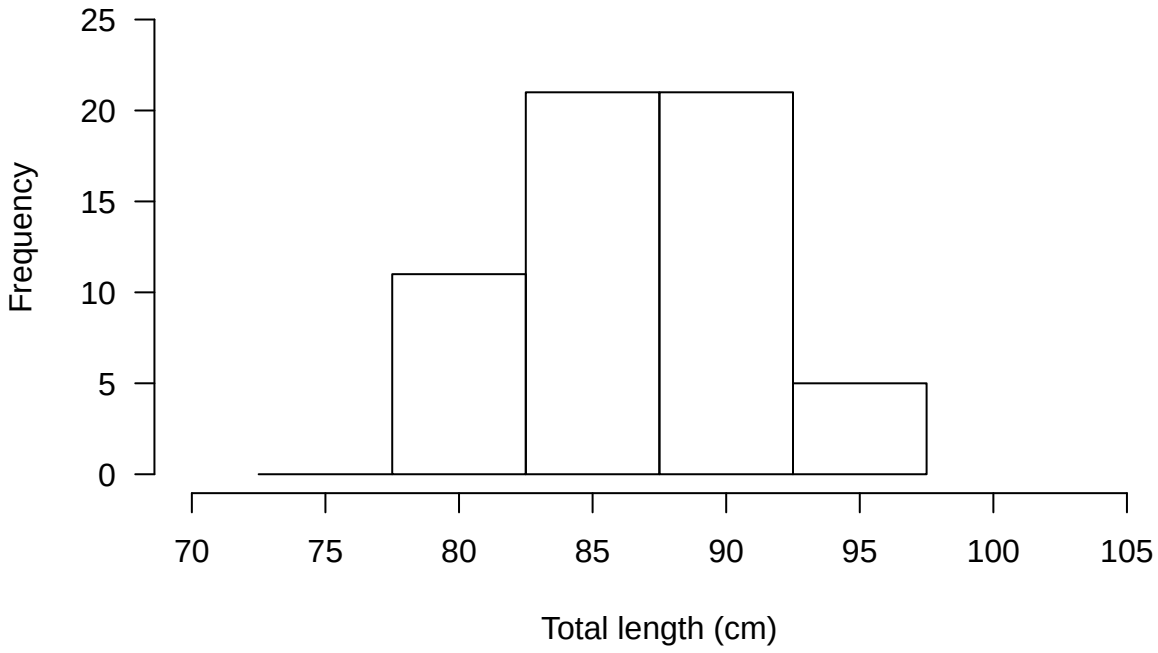
The histogram shows that more than 15 possums had a lenght between 85 and 90 cm, whereas 11 and 12 possums had a lenght between 80 and 85 cm and between 90 and 95 cm, respectively. Instead of absolute frequencies, the histogram can also be used to display relative frequencies, i.e. the probability densities:

```
hist(Vic_pos2, probability = TRUE)
```

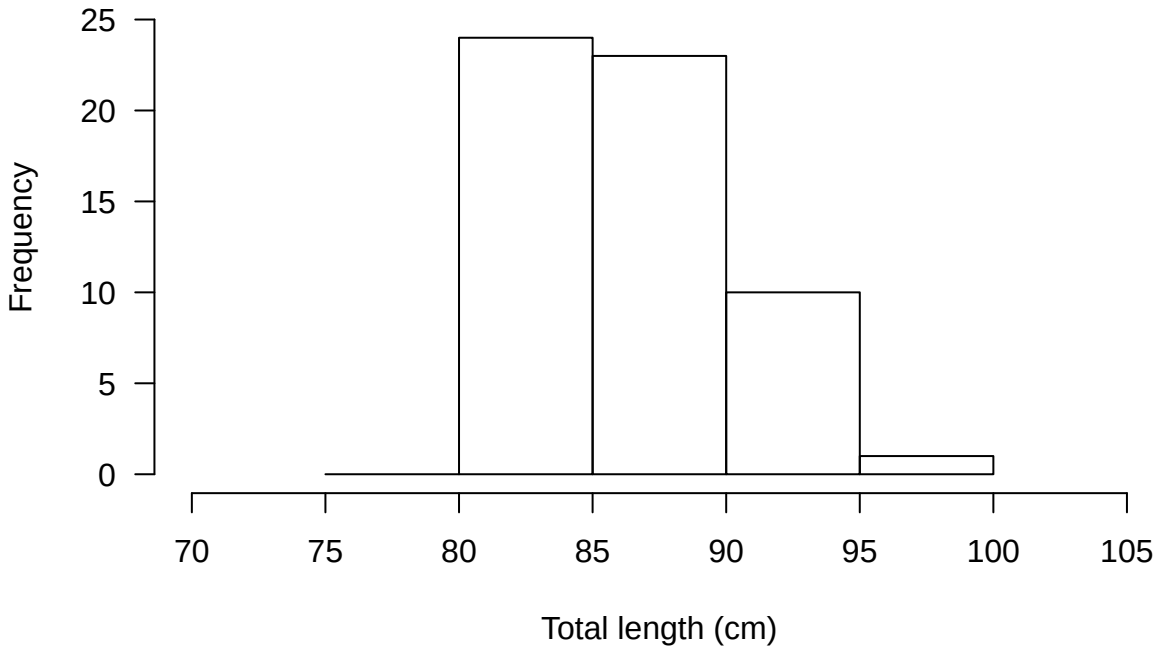


Note that different intervals affect the outlook and potentially the interpretation of a histogram:

```
# We use the data from the non-Victorian possum population.
nVic_pos <- pos_dat %>%
  filter(Pop == "other") %>%
  select(totlngth)
# convert to vector
nVic_pos2 <- as.vector(t(nVic_pos))
# plot histogram, manually provide breaks
hist(nVic_pos2, breaks = 72.5 + (0:5) * 5,
     xlim = c(70, 105), ylim = c(0, 26),
     xlab = "Total length (cm)", main = "", las = 1)
```



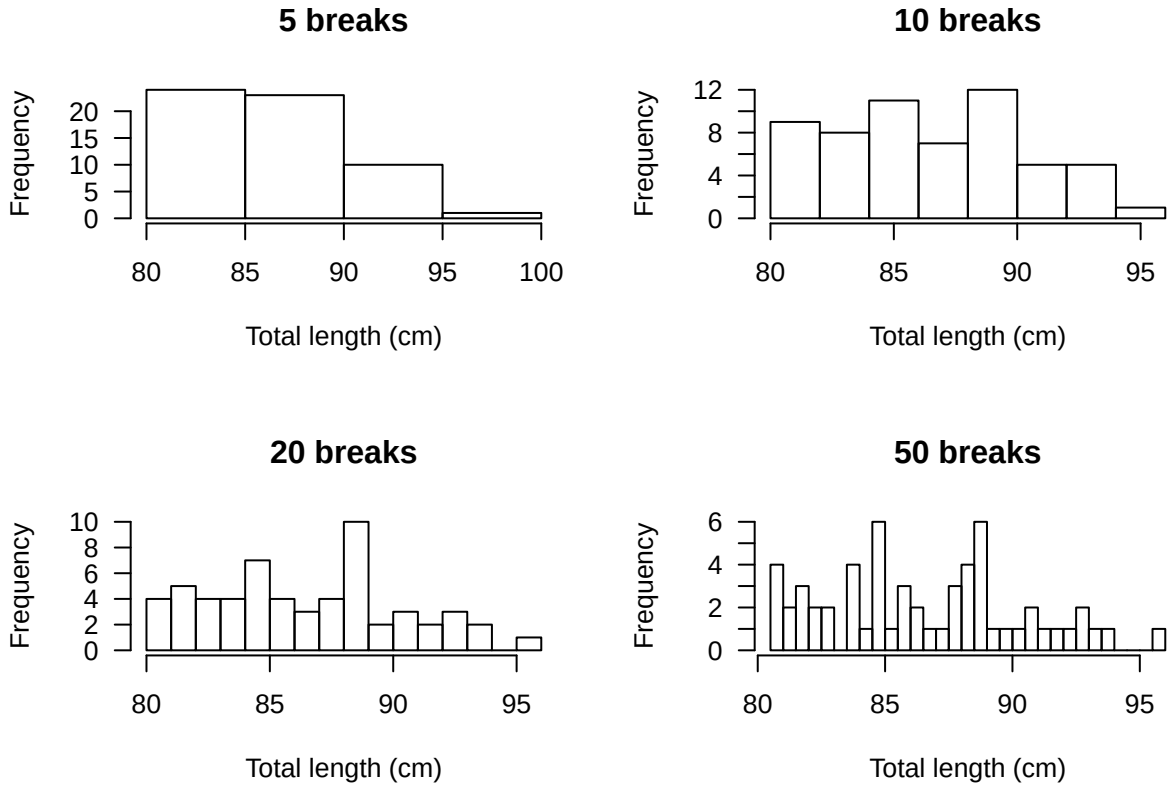

```
# Plot suggests a normal distribution
# Provide different set of breaks
hist(nVic_pos2, breaks = 75 + (0:5) * 5,
     xlim = c(70, 105), ylim = c(0, 26),
     xlab = "Total length (cm)", main = "", las = 1)
```




```
# With different break points between the intervals, the data look rather skewed
```

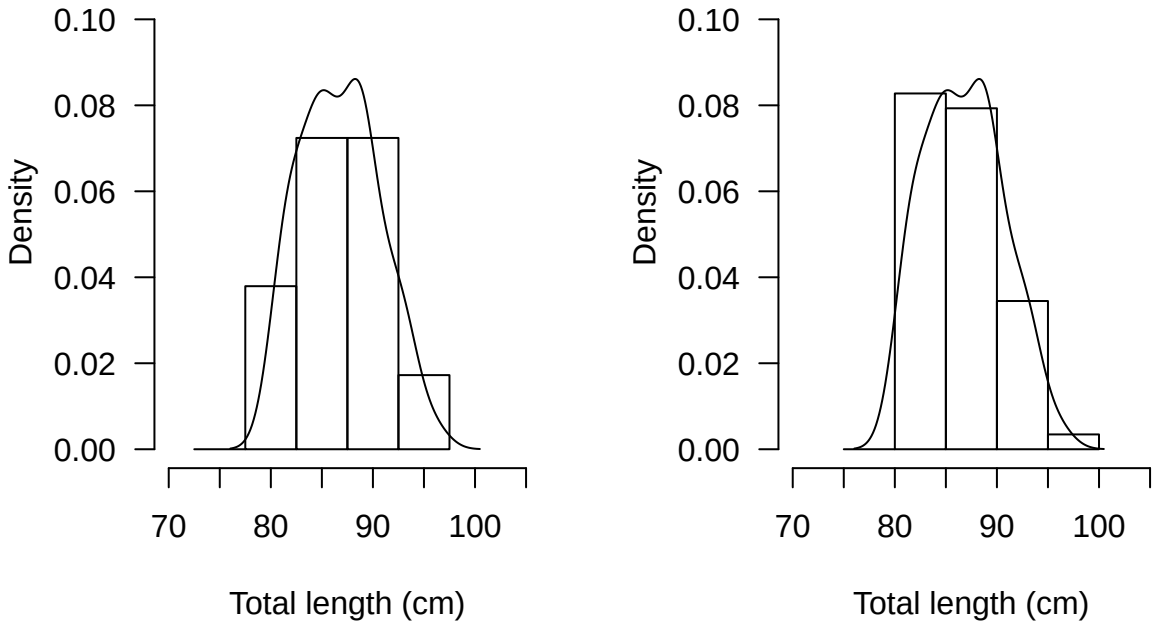
The outlook and interpretation is also affected by the number of breaks:

```
par(mfrow = c(2,2), las = 1)
hist(nVic_pos2, breaks = 5, main = "5 breaks", xlab = "Total length (cm)")
hist(nVic_pos2, breaks = 10, main = "10 breaks", xlab = "Total length (cm)")
hist(nVic_pos2, breaks = 20, main = "20 breaks", xlab = "Total length (cm)")
hist(nVic_pos2, breaks = 50, main = "50 breaks", xlab = "Total length (cm)")
```



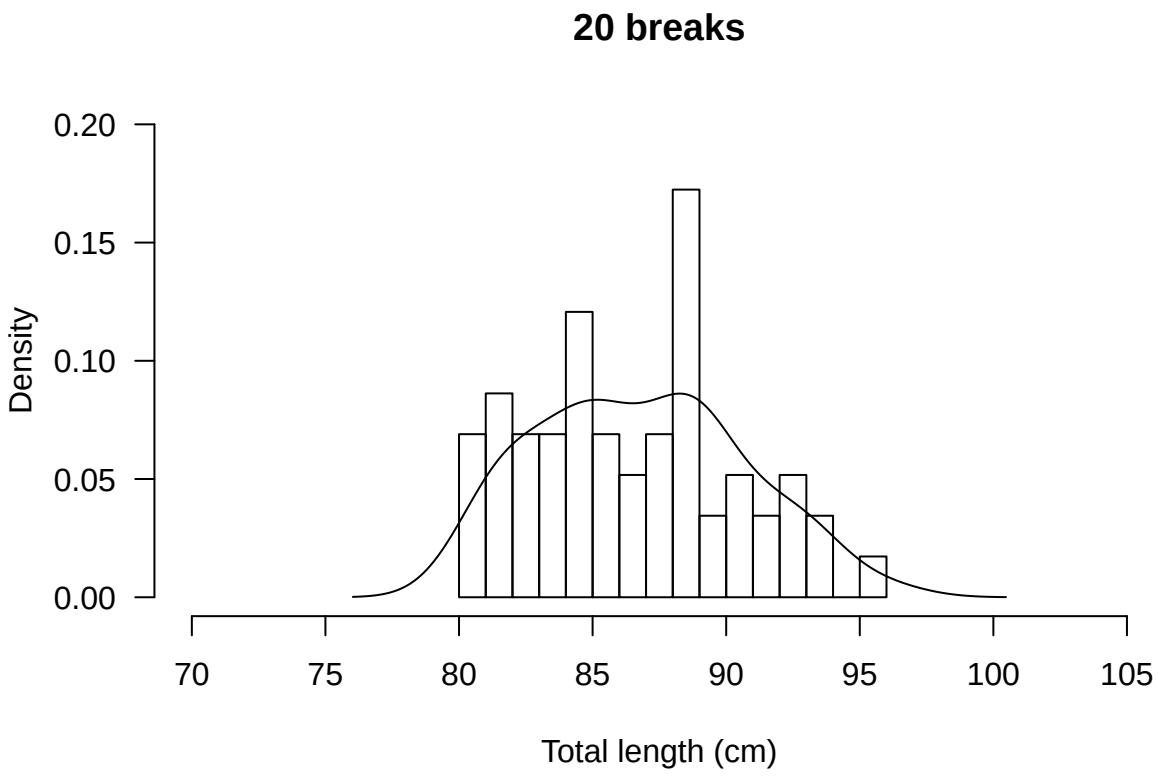
Adding density lines to histograms aids in reducing the effect of the number of breaks and of the position of the break points on the interpretation. The density lines are derived from the empirical density distribution of the data.

```
# Derive density line
dens <- density(nVic_pos2)
# add to first histogram that suggested normal distribution
# only works for relative frequency histogram, which requires
# to set the argument probability = TRUE
par(mfrow = c(1,2), las = 1)
hist(nVic_pos2, breaks = 72.5 + (0:5) * 5,
     xlim = c(70, 105), ylim = c(0, 0.11), probability = TRUE,
     xlab = "Total length (cm)", main = "")
lines(dens)
# add to histogram for which data look rather skewed
hist(nVic_pos2, breaks = 75 + (0:5) * 5,
     xlim = c(70, 105), ylim = c(0, 0.11), probability = TRUE,
     xlab = "Total length (cm)", main = "")
lines(dens)
```



The density lines are the same and confirm the approximate normal distribution of the data. Similarly, the density lines can alleviate the effect of the number of breaks on the interpretation, where too few or too many breaks may result in flawed interpretations.

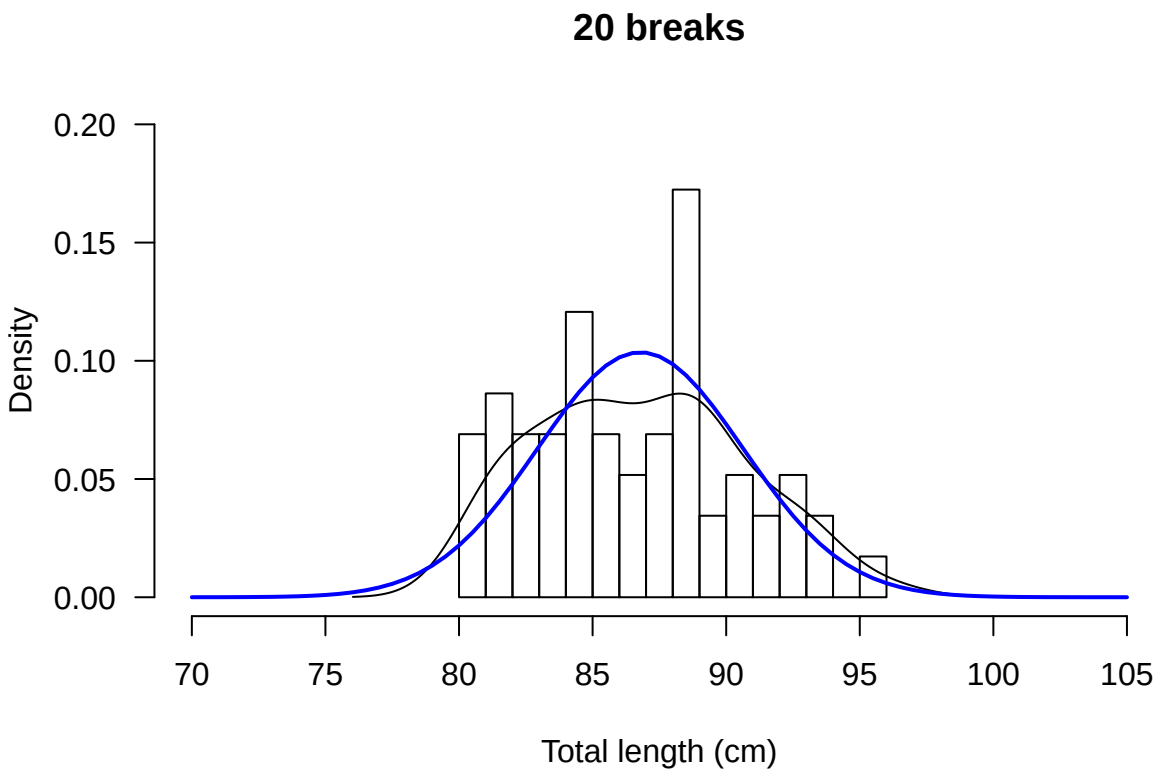
```
par(mfrow = c(1,1), las = 1)
hist(nVic_pos2, breaks = 20,
     xlim = c(70, 105), ylim = c(0, 0.2), probability = TRUE,
     xlab = "Total length (cm)", main = "20 breaks")
lines(dens)
```



Setting break points manually or a high number of breaks is particularly useful when the aim is to assess the frequency of distinct values such as zeros.

To evaluate normality of data in a histogram can be done by overlaying a density line from a theoretical normal probability distribution. To do this, we generate a normal distribution with the parameters (i.e. mean and variance) taken from the sample data.

```
# calculate sample mean
mean_samp <- mean(nVic_pos2)
# calculate sample standard deviation
sd_samp <- sd(nVic_pos2)
# derive densities for normal distribution
dens_norm <- dnorm(seq(70, 105, by=.5), mean = mean_samp, sd = sd_samp)
# add to plot
hist(nVic_pos2, breaks = 20,
     xlim = c(70, 105), ylim = c(0, 0.2), probability = TRUE,
     xlab = "Total length (cm)", main = "20 breaks", las = 1)
lines(dens)
lines(seq(70, 105, by=.5), dens_norm, col="blue", lwd = 2)
```



The blue line displays the normal distribution. A slight deviation of the sample distribution is visible, which is presumably due to ignoring the influence of sex on the length of possums.

A range of plots that illustrate different deviations from normality are available on the histogram Wikipedia page: right skewed distribution, left skewed distribution, bimodal distribution and multimodal distribution.

Several other tools for exploratory analysis that have been mentioned in the lecture will be used and introduced in the context of specific methods of data analysis later in the course.

You can render the Rmarkdown document related to this pdf by executing the following function:

```
# rmarkdown::render("/Users/ralfs/Gitprojects/Teaching/Data_analysis/Code/Session_1.R")
# You need to replace the file location with the path to your file location.
```

References

Kampstra, Peter. 2008. "Beanplot: A Boxplot Alternative for Visual Comparison of Distribution." *Journal of Statistical Software* 28: 1–9.

Lindenmayer, DB, KL Viggers, RB Cunningham, and CF Donnelly. 1995. "Morphological Variation Among Populations of the Mountain Brushtail Possum, Trichosurus-Caninus Ogilby (Phalangeridae, Marsupialia)." *Australian Journal of Zoology* 43 (5): 449–58.

Maindonald, J. H., and John Braun. 2010. *Data Analysis and Graphics Using R: An Example-Based Approach*. 3rd ed. Cambridge Series in Statistical and Probabilistic Mathematics 10. Cambridge ; New York: Cambridge University Press.