# An introduction to R using the tidyverse

***Bernd Klaus***

**12 Mai 2017**

## Contents

# 1 Required packages and other preparations

```
library("TeachingDemos")
library("openxlsx")
library("multtest")
library("Biobase")
library("tidyverse")
library("cowplot")
```

# 2 Introduction and getting help

R is a software language for carrying out complicated (and simple) statistical analyses. It includes routines for data summary and exploration, graphical presentation and data modeling. The aim of this lab is to provide you with a basic fluency in the language. When you work in R you create objects that are stored in the current workspace. Each object created remains in the image unless you explicitly delete it. At the end of the session the workspace will be lost unless you save it.

You can get your current working directory via `getwd()` and set it with `setwd()`. By default, it is usually your home directory.

Commands written in R are saved in memory throughout the session. You can scroll back to previous commands typed by using the "up" arrow key (and "down" to scroll back again). You finish an R session by typing `q()` at which point you will also be prompted as to whether or not you want to save the current workspace into your working directory. If you do not want to, it will be lost. Remember the ways to get help:

- Just ask!
- `help.start()` and the HTML help button in the Windows GUI.
- `help` and `?`: `help("data.frame")` or `?help`.
- `help.search()`, `apropos()`
- `browseVignettes("package")`
- rseek.org
- use tab–completion in RStudio, this will also display help–snippets

In this tutorial we will make use of packages from the tidyverse and the tutorial itself was written using rmarkdown. The tidyverse is a set of R packages that try to make your life easier when working with data in R. They improve the basic R experience tremendously and are designed to foster the human understanding of programming code.

# 3 Elementary data types and arithmetics

The elementary unit in R is an object and the simplest objects are scalars, vectors and matrices. R is designed with interactivity in mind, so you can get started by simply typing:

```
4 + 6
   [1] 10
```

What does R do? It sums up the two numbers and returns the scalar value 10. In fact, R returns a vector of length 1 - hence the [1] denoting first element of the vector. We can assign objects values for subsequent use. For example:

```
x <- 6
y <- 4
z <- x + y
z
   [1] 10
```

does the same calculation as above, storing the result in an object called `z`. We can look at the contents of the object by simply typing its name. At any time we can list the objects which we have created:

```
ls()
    [1] "a"             "binary_s"      "bodyfat"       "d"
    [5] "days"          "embl_colors"   "h"             "hex_grid"
    [9] "i"             "inch_to_m"     "L"             "pat"
   [13] "pat_tiny"      "pat_xls"       "pb_to_kg"      "readError"
   [17] "readError2"    "robust_z"      "s"             "sample_vector"
   [21] "Shop2"         "test"          "Tuesday"       "w"
   [25] "x"             "y"             "y_mat"         "y_tibble"
   [29] "z"
```

Notice that `ls` is actually an object itself. Typing `ls` would result in a display of the contents of this object, in this case, the commands of the function. The use of parentheses, `ls()`, ensures that the function is executed and its result — in this case, a list of the objects in the current environment — displayed. More commonly, a function will operate on an object, for example

```
sqrt(16)
   [1] 4
```

calculates the square root of 16. Objects can be removed from the current workspace with the function `rm()`. There are many standard functions available in R, and it is also possible to create new ones. Vectors can be created in R in a number of ways. For example, we can list all of the elements:

```
z <- c(5, 9, 1, 0)
```

Note the use of the function `c` to concatenate or "glue together" individual elements. This function can be used much more widely, for example

```
x <- c(5, 9)
y <- c(1, 0)
z <- c(x, y)
```

would lead to the same result by gluing together two vectors to create a single vector. Sequences can be generated as follows:

```
seq(1, 9, by = 2)
   [1] 1 3 5 7 9
seq(8, 20, length = 6)
   [1]  8.0 10.4 12.8 15.2 17.6 20.0
```

These examples illustrate that many functions in R have optional arguments, in this case, either the step length or the total length of the sequence (it doesn't make sense to use both). If you leave out both of these options, R will make its own default choice, in this case assuming a step length of 1. So, for example,

```
x <- seq(1, 10)
```

also generates a vector of integers from 1 to 10. At this point it's worth mentioning the help facility again. If you don't know how to use a function, or don't know what the options or default values are, type `help(functionname)` or simply `?functionname` where `functionname` is the name of the function you are interested in. This will usually help and will often include examples to make things even clearer. Another useful function for building vectors is the `rep` command for repeating things: the first command will repeat the vector 1, 2, 3 six times, will the second one will repeat each element six times.

```
rep(1:3, 6)
    [1] 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3 1 2 3
rep(1:3, times = c(6, 6, 6))
    [1] 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3
```

R will often adapt to the objects it is asked to work on. An example is the vectorized arithmetic used in R:

```
x <- 1:5
y <- 5:1
x + y
   [1] 6 6 6 6 6
x^2
   [1]  1  4  9 16 25
x * y
   [1] 5 8 9 8 5
```

showing that R uses component-wise arithmetic on vectors. R will also try to make sense of a statement if objects are mixed. For example:

```
x <- c(6, 8 , 9 )
x + 2
   [1]  8 10 11
```

Two particularly useful functions worth remembering are `length`, which returns the length of a vector (i.e. the number of elements it contains) and `sum` which calculates the sum of the elements of a vector. R also has basic calculator capabilities:

- `a+b`, `a-b`, `a\*b`, `a\*\*b` (a to the power of b)
- additionally: `sqrt(a)`, `sin(a)` ...

and some simple statistics:

- `mean(a)`
- `summary(a)`
- `var(a)`
- `min(a,b)`, `max(a,b)`

**Exercise: Simple R operations**

Define

- x <- c(4, 2, 6)

and

- y <- c(1, 0, -1)

Decide what the result will be of the following:

- length(x)
- sum(x)
- sum(x^2)
- x + y
- x * y
- x - 2
- x^2

Use R to check your answers.

Decide what the following sequences are and use R to check your answers:

- 7:11
- seq(2, 9)
- seq(4, 10, by=2)
- seq(3, 30, length=10)
- seq(6, -4, by=-2)

Determine what the result will be of the following R expressions, and then use R to check whether you are right:

- rep(2, 4)
- rep(c(1, 2), 4)
- rep(c(1, 2), c(4, 4))
- rep(1:4, 4)
- rep(1:4, rep(3, 4))

Use the `rep` function to define simply the following vectors in R.

- (6, 6, 6, 6, 6, 6)
- (5, 8, 5, 8, 5, 8, 5, 8)
- (5, 5, 5, 5, 8, 8, 8, 8)

**Exercise: R as a calculator** Calculate the following expression, where `x` and `y` have values `-0.25` and `2` respectively.

Then store the result in a new variable and print its content.

```
x + cos(pi/y)
```

# 4 Summaries, subscripting and useful vector functions

Let's suppose we've collected some data from an experiment and stored them in an object `x`. Some simple summary statistics of these data can be produced:

```
x <- c(7.5, 8.2, 3.1, 5.6, 8.2, 9.3, 6.5, 7.0, 9.3, 1.2, 14.5, 6.2)
mean(x)
   [1] 7.22
var(x)
   [1] 11
summary(x)
      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1.20    6.05    7.25    7.22    8.47   14.50
```

It may be, however, that we subsequently learn that the first 6 data points correspond to measurements made in one experiment, and the second six on another experiment. This might suggest summarizing the two sets of data separately, so we would need to extract from `x` the two relevant subvectors. This is achieved by subscripting:

```
x[1:6]
   [1] 7.5 8.2 3.1 5.6 8.2 9.3
x[7:12]
   [1]  6.5  7.0  9.3  1.2 14.5  6.2
summary(x[1:6])
      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      3.10    6.07    7.85    6.98    8.20    9.30
summary(x[7:12])
      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1.20    6.28    6.75    7.45    8.73   14.50
```

You simply put the indexes of the element you want to access in square brackets. Note that R starts counting from 1 onwards. Other subsets can be created in the obvious way. Putting a minus in front, excludes the elements:

```
x[c(2, 4, 9)]
   [1] 8.2 5.6 9.3
x[-(1:6)]
   [1]  6.5  7.0  9.3  1.2 14.5  6.2
head(x)
   [1] 7.5 8.2 3.1 5.6 8.2 9.3
```

The function `head` provides a preview of the vector. There are also useful functions to order and sort vectors:

- `sort`: sort in increasing order
- `order`: orders the indexes is such a way that the elements of the vector are sorted, i.e `sort(v) = v[order(v)]`
- `rank`: gives the ranks of the elements of a vector, different options for handling *ties* are available.

```
x <- c(1.3, 3.5, 2.7, 6.3, 6.3)
sort(x)
   [1] 1.3 2.7 3.5 6.3 6.3
order(x)
   [1] 1 3 2 4 5
x[order(x)]
   [1] 1.3 2.7 3.5 6.3 6.3
```

```
rank(x)
   [1] 1.0 3.0 2.0 4.5 4.5
```

**Exercise: Milk sales and summaries**

- Define x <- c(5, 9, 2, 3, 4, 6, 7, 0, 8, 12, 2, 9)

Decide what the result will be of the following:

- x[2]
- x[2:4]
- x[c(2, 3, 6)]
- x[c(1:5, 10:12)]
- x[-(10:12)]

Use R to check your answers.

- The vector y <- c(33, 44, 29, 16, 25, 45, 33, 19, 54, 22, 21, 49, 11, 24, 56) contains sales of milk in liters for 5 days in three different shops (the first 3 values are for shops 1, 2 and 3 on Monday, etc.). Produce a statistical summary of the sales for each day of the week and also for each shop.

# 5 Classes, modes and types of objects

R is an object-oriented language, so every data item is an object in R. As in other programming languages, objects are instances of "blue-prints" called classes. There are the following elementary types or ("modes"):

- numeric: real number
- character: chain of characters, text
- factor: categorical data that takes a fixed set of values
- logical: TRUE, FALSE
- special values: NA (missing value), NULL ("empty object"), Inf, -Inf (infinity), NaN (not a number)

We haven't met factors yet: They are designed to represent categorical data that can take a fixed set of possible values. Factors are built on top of integers, and have a levels attribute:

```
x <- factor(c("wt", "wt", "mut", "mut"), levels = c("wt", "mut"))
x
   [1] wt  wt  mut mut
   Levels: wt mut
```

Data storage types includes matrices, lists, data frames (tibbles), which will be introduced in the next section. Certain types can have different subtypes, e.g. numeric can be further subdivided into the integer, single and double types. Types can be checked by the `is.*` and changed ("casted") by the `as.*` functions. Furthermore, the function `str` is very useful in order to obtain an overview of an (possibly complex) object at hand. The following examples will make this clear. We first assign the value `9` to an object and then perform various operations on it.

```
a <- 9
# is a a string?
```

```
is.character(a)
   [1] FALSE
# is a a number?
is.numeric(a)
   [1] TRUE
# What's its type?
typeof(a)
   [1] "double"
# now turn it into a factor
a <- as.factor(a)
# Is it a factor?
is.factor(a)
   [1] TRUE
# assign an string to a:
a <- "NAME"
# what's a?
class(a)
   [1] "character"
str(a)
    chr "NAME"
```

# 6 Matrices, lists, data frames and basic data handling

## 6.1 Matrices

Matrices are two–dimensional vectors and can be created in R in a variety of ways. Perhaps the simplest is to create the columns and then glue them together with the command `cbind`. For example:

```
x <- c(5, 7 , 9)
y <- c(6, 3 , 4)
z <- cbind(x, y)
z
        x y
   [1,] 5 6
   [2,] 7 3
   [3,] 9 4
dim(z)
   [1] 3 2
```

We can also use the function `matrix()` directly to create a matrix.

```
z <- matrix(c(5, 7, 9, 6, 3, 4), nrow = 3)
```

There is a similar command, `rbind`, for building matrices by gluing rows together. The functions `cbind` and `rbind` can also be applied to matrices themselves (provided the dimensions match) to form larger matrices.

Notice that the dimension of the matrix is determined by the size of the vector and the requirement that the number of rows is 3 in the example above, as specified by the argument `nrow = 3`. As an alternative we could have specified the number of columns with the argument `ncol = 2` (obviously, it is unnecessary to give both). Notice that the matrix is "filled up" column-wise. If instead you wish to fill up row-wise, add the option `byrow=TRUE`.

```
z <- matrix(c(5, 7 , 9 , 6 , 3 , 4), nrow = 3, byrow = TRUE)
z
     [,1] [,2]
[1,]    5    7
[2,]    9    6
[3,]    3    4
```

R will try to interpret operations on matrices in a natural way. For example, with z as above, and y defined below we get:

```
y <- matrix(c(1, 3, 0, 9, 5, -1), nrow = 3, byrow = TRUE)
y
     [,1] [,2]
[1,]    1    3
[2,]    0    9
[3,]    5   -1
y + z
     [,1] [,2]
[1,]    6   10
[2,]    9   15
[3,]    8    3
y * z
     [,1] [,2]
[1,]    5   21
[2,]    0   54
[3,]   15   -4
```

Notice that multiplication here is component–wise. As with vectors it is useful to be able to extract sub-components of matrices. In this case, we may wish to pick out individual elements, rows or columns. As before, the `[ ]` notation is used to subscript. The following examples illustrate this:

```
z[1, 1]
[1] 5
z[, 2]
[1] 7 6 4
z[1:2, ]
     [,1] [,2]
[1,]    5    7
[2,]    9    6
z[-1, ]
     [,1] [,2]
[1,]    9    6
[2,]    3    4
z[-c(1, 2), ]
[1] 3 4
```

So, in particular, it is necessary to specify which rows and columns are required, whilst omitting the index for either dimension implies that every element in that dimension is selected.

## 6.2 Data frames (tibbles) and lists

A data frame is a matrix where the columns can have different data types. As such, it is usually used to represent a whole data set, where the rows represent the samples and columns the variables. Essentially, you can think of a data frame as an excel table.

Here, we will meet the first tidyverse member, namely the *tibble* package, which improves the conventional R `data.frame` class. A tibble is a `data.frame` which a lot of tweaks and more sensible defaults that make your life easier. For details on the tweaks, see the help on tibble: `?tibble` so that you never have to use a standard data frame anymore.

Let's illustrate this by the small data set saved in comma–separated-format (csv) — `patients`. We load it in from a website using the function `read_csv`, which is used to import a data file in *comma separated format — csv* into R. In a .csv–file the data are stored row–wise, and the entries in each row are separated by commas.

The function `read_csv` is from the *readr* package and will give us a tibble as the result. The function `glimpse()` gives a nice summary of a tibble.

```
pat <- read_csv("http://www-huber.embl.de/users/klaus/BasicR/Patients.csv")
   Parsed with column specification:
   cols(
     PatientId = col_character(),
     Height = col_double(),
     Weight = col_double(),
     Gender = col_character()
   )
pat
   # A tibble: 3 × 4
     PatientId Height Weight Gender
         <chr>  <dbl>  <dbl>  <chr>
   1        P1   1.65     75      f
   2        P2   1.90     NA      m
   3        P3   1.60     50      f
glimpse(pat)
   Observations: 3
   Variables: 4
   $ PatientId <chr> "P1", "P2", "P3"
   $ Height    <dbl> 1.65, 1.90, 1.60
   $ Weight    <dbl> 75, NA, 50
   $ Gender    <chr> "f", "m", "f"
```

It has weight, height and gender of three people. We can also use the function `read.xlsx` from the *openxlsx* package to import the data from an excel sheet. Here, we have to use the function `as_tibble` to turn the data.frame into an equivalent tibble.

```
pat_xls <- as_tibble(read.xlsx("Patients.xlsx"))
pat_xls
   # A tibble: 3 × 4
```

```
  PatientId Height Weight Gender
*     <chr>  <chr>  <chr>  <chr>
1        P1   1.65   75.0      f
2        P2   1.90   <NA>      m
3        P3   1.60   50.0      f
str(pat_xls)
  Classes 'tbl_df', 'tbl' and 'data.frame':    3 obs. of  4 variables:
   $ PatientId: chr  "P1" "P2" "P3"
   $ Height   : chr  "1.65" "1.90" "1.60"
   $ Weight   : chr  "75.0" NA "50.0"
   $ Gender   : chr  "f" "m" "f"
```

## 6.3 Accessing data in data frames

Now that we have imported the small data set, you might be wondering how to actually access the data. For this the functions `filter` and `select` from the *dplyr* package of the *tidyverse* are useful. `filter` will select certain rows (observations), while `select` will subset the columns (variables of the data). In the following command, we get all the patients that are less tall than 1.5 and select their Height and Gender as well as their Id:

```
pat_tiny <- filter(pat, Height < 1.7)
select(pat_tiny, PatientId,  Height, Gender)
  # A tibble: 2 × 3
    PatientId Height Gender
        <chr>  <dbl>  <chr>
1        P1   1.65      f
2        P3   1.60      f
```

There are a couple of operators useful for comparisons:

- `Variable == value`: equal
- `Variable != value`: un–equal
- `Variable < value`: less
- `Variable > value`: greater
- `&`: and
- `|` or
- `!`: negation
- `%in%`: is element?

The function `filter` allows us to combine multiple conditions easily, if you specify multiple of them, they will automatically concatenated via a `&`. For example, we can easily get light and female patients via:

```
filter(pat, Height < 1.5, Gender == "f")
  # A tibble: 0 × 4
  # ... with 4 variables: PatientId <chr>, Height <dbl>, Weight <dbl>,
  #   Gender <chr>
```

We can also retrieve small OR female patients via

```
filter(pat, (Height < 1.5) | (Gender == "f"))
   # A tibble: 2 × 4
     PatientId Height Weight Gender
         <chr>  <dbl>  <dbl>  <chr>
   1        P1   1.65     75      f
   2        P3   1.60     50      f
```

## 6.4 Vectors with arbitrary contents: Lists

Lists can be viewed as vectors that contain not only elementary objects such as number or strings but can potentially arbitrary objects. The following example will make this clear. The list that we create contains a number, two vectors and a string that is itself part of a list.

```
L <- list(one = 1, two = c(1, 2), five = seq(1, 4, length = 5),
          list(string = "Hello World"))
L
   $one
   [1] 1

   $two
   [1] 1 2

   $five
   [1] 1.00 1.75 2.50 3.25 4.00

   [[4]]
   [[4]]$string
   [1] "Hello World"
```

Lists are the most general data type in R. In fact, data frames (tibbles) are lists with elements of equal lengths. List elements can either be accessed by their name using the dollar sign `$` or via their position via a double bracket operator `[[]]`.

```
names(L)
   [1] "one"  "two"  "five" ""
L$five + 10
   [1] 11.0 11.8 12.5 13.2 14.0
L[[3]] + 10
   [1] 11.0 11.8 12.5 13.2 14.0
```

Using only a single bracket (`[]`) will extract a sublist, so the result will always be a list, while the dollar sign `$` or the double bracket operator `[[]]` removes a level of the list hierarchy. Thus, in order to access the string, we would first have to extract the sublist containing the string from `L` and then get the actual string from the sublist, `[[` drills down into the list while `[` returns a new, smaller list.

```
L[[4]]$string
   [1] "Hello World"
L[2]
   $two
```

```
   [1] 1 2
```

Since data frames are just a special kind of lists, they can actually be accessed in the same way.

```
pat$Height
   [1] 1.65 1.90 1.60
pat[[2]]
   [1] 1.65 1.90 1.60
pat[["Gender"]]
   [1] "f" "m" "f"
```

More on lists can be found in the respective chapter of "R for data science" here.

## 6.5 Summary: data access in R

We prape a simple vector to illustrate the access options again:

```
sample_vector <- c("Alice" = 5.4, "Bob" = 3.7, "Claire" = 8.8)
sample_vector
    Alice    Bob Claire
      5.4    3.7    8.8
```

### 6.5.1 Access by index

The simplest way to access the elements in a vector is via their indices. Specifically, you provide a vector of indices to say which elements from the vector you want to retrieve. A minus sign excludes the respective positions

```
sample_vector[1:2]
   Alice   Bob
     5.4   3.7
sample_vector[-(1:2)]
   Claire
      8.8
```

### 6.5.2 Access by boolean

If you generate a boolean vector the same size as your actual vector you can use the positions of the true values to pull out certain positions from the full set. You can also use smaller boolean vectors and they will be concatenated to match all of the positions in the vector, but this is less common.

```
sample_vector[c(TRUE, FALSE, TRUE)]
    Alice Claire
      5.4    8.8
```

This can also be used in conjunction with logical tests which generate a boolean result. Boolean vectors can be combined with logical operators to create more complex filters.

```
sample_vector[sample_vector < 6]
   Alice   Bob
     5.4   3.7
```

### 6.5.3 Access by name

if there are names such as column names present (note that rowname are not preserved in the tidyverse), you can access by name as well:

```
sample_vector[c("Alice", "Claire")]
    Alice Claire
      5.4    8.8
```

## 6.6 Applying a function to elements of a data structure

R encourages the use of functions for programming. Instead of e.g. looping through a vector or data frame, you can use specialized functions that apply another function to each element of your data. These kinds of functions are called apply functions. Here, we will use the `map` family of functions from the *purrr* package instead of the base R functions. An apply / map call applies a function to a vector or list and returns the result in another vector/list. Thus, each step consists of "mapping" a list value to a result.

We will introduce the `map` functions by looking at a typical data set in a tabular format, where the rows represent the samples and the columns the variables measured. The data set `bodyfat` contains various body measures for 252 men. We turn it into a tibble by using the function `as_tibble()`.

Let's inspect it a bit. The first thing we notice is that tibbles prints only the first 10 rows by default. Tibbles are designed so that you don't accidentally overwhelm your console when you print large data frames. Additionally, we get a nice summary of the variables available in our data set.

```
load(url("http://www-huber.embl.de/users/klaus/BasicR/bodyfat.rda"))
bodyfat <- as_tibble(bodyfat)
bodyfat
   # A tibble: 252 × 15
      density percent.fat   age weight height neck.circum chest.circum
        <dbl>       <dbl> <int>  <dbl>  <dbl>       <dbl>        <dbl>
   1     1.07        12.3    23    154   67.8        36.2         93.1
   2     1.09         6.1    22    173   72.2        38.5         93.6
   3     1.04        25.3    22    154   66.2        34.0         95.8
   4     1.08        10.4    26    185   72.2        37.4        101.8
   5     1.03        28.7    24    184   71.2        34.4         97.3
   6     1.05        20.9    24    210   74.8        39.0        104.5
   7     1.05        19.2    26    181   69.8        36.4        105.1
   8     1.07        12.4    25    176   72.5        37.8         99.6
   9     1.09         4.1    25    191   74.0        38.1        100.9
   10    1.07        11.7    23    198   73.5        42.1         99.6
   # ... with 242 more rows, and 8 more variables: abdomen.circum <dbl>,
   #   hip.circum <dbl>, thigh.circum <dbl>, knee.circum <dbl>,
```

```
#   ankle.circum <dbl>, bicep.circum <dbl>, forearm.circum <dbl>,
#   wrist.circum <dbl>
```

As data frames are just a special kind of list, namely a list that is composed of vectors of equal length, we can use a map function to compute the mean value for every variable in our data set.

```
head(map_dbl(bodyfat, mean))
      density percent.fat         age      weight      height neck.circum
         1.06       19.15       44.88      178.92       70.15       37.99
```

Here we use map_dbl, to ensure that we get a double value back. There are specialized mapping functions for many data types, but you can always use the default `map()` function as a fallback when there is no specialized equivalent available.

The map functions are really useful for applying your custom functions, for example we can compute a robust z–score by subtracting the median and dividing by the mean absolute deviation for each variable.

This will bring all the variables in the data set to a common scale and make them directly comparable. These kinds of transformations are often performed before clustering or dimensionality reduction.

You can create your own functions very easily by adhering to the following template:

```
function_name <- function(argument_1, argument_2,
                          optional_argument = defautl_value )
{
  return(...)
}
```

As you can see, the source code of the function has to be in curly brackets, while the arguments are defined in the parentheses. Arguments without a default value are mandatory, and default value are specified by equality signs.

By default R returns the result of the last computation performed within the curly brackets (often, this will be the last line of the function). However, you can always specify the return value directly with `return()`. If you want to return multiple values, you can return a list.

We can now easily define our function and apply it to the data set.

```
robust_z <- function(x){
  (x - median(x)) / mad(x)
}

head(map_df(bodyfat, robust_z), 3)
   # A tibble: 3 × 15
     density percent.fat   age weight height neck.circum chest.circum
       <dbl>       <dbl> <dbl>  <dbl>  <dbl>       <dbl>        <dbl>
   1   0.763      -0.745 -1.69 -0.775 -0.759      -0.759       -0.782
   2   1.459      -1.414 -1.77 -0.113  0.759       0.211       -0.722
   3  -0.648       0.658 -1.77 -0.783 -1.265      -1.686       -0.460
   # ... with 8 more variables: abdomen.circum <dbl>, hip.circum <dbl>,
   #   thigh.circum <dbl>, knee.circum <dbl>, ankle.circum <dbl>,
   #   bicep.circum <dbl>, forearm.circum <dbl>, wrist.circum <dbl>
```

Here, we used the function `map_df` to make sure that we get a data frame back. There is an even simpler way to achieve the same goal. Using a tilde (~) to create an R formula, the map functions allow you to define anonymous functions with a default argument `.x`.

With this, we do not need to define our robust z–score function explicitly.

```
head(map_df(bodyfat, ~  (.x - median(.x)) / mad(.x)), 3)
   # A tibble: 3 × 15
     density percent.fat   age weight height neck.circum chest.circum
       <dbl>       <dbl> <dbl>  <dbl>  <dbl>       <dbl>        <dbl>
   1   0.763      -0.745 -1.69 -0.775 -0.759      -0.759       -0.782
   2   1.459      -1.414 -1.77 -0.113  0.759       0.211       -0.722
   3  -0.648       0.658 -1.77 -0.783 -1.265      -1.686       -0.460
   # ... with 8 more variables: abdomen.circum <dbl>, hip.circum <dbl>,
   #   thigh.circum <dbl>, knee.circum <dbl>, ankle.circum <dbl>,
   #   bicep.circum <dbl>, forearm.circum <dbl>, wrist.circum <dbl>
```

## 6.7 Computing variables from existing ones and predicate functions

Often, we want to use variables stored in our data set to compute derived quantities. For example, we might be interest in the weight in kilograms instead of pounds and the height in meters instead of inches. The function `mutate` allows us to do this.

```
pb_to_kg <- 1/2.2046
inch_to_m <- 0.0254

bodyfat <- mutate(bodyfat, height_m = height * inch_to_m,
                           weight_kg = weight * pb_to_kg)

select(bodyfat, height,  height_m, weight, weight_kg)
   # A tibble: 252 × 4
      height height_m weight weight_kg
       <dbl>    <dbl>  <dbl>     <dbl>
   1    67.8     1.72    154      70.0
   2    72.2     1.84    173      78.6
   3    66.2     1.68    154      69.9
   4    72.2     1.84    185      83.8
   5    71.2     1.81    184      83.6
   6    74.8     1.90    210      95.4
   7    69.8     1.77    181      82.1
   8    72.5     1.84    176      79.8
   9    74.0     1.88    191      86.6
   10   73.5     1.87    198      89.9
   # ... with 242 more rows
```

We often want to apply our function only to variables in the data set that are of a specific type, e.g. numeric, we can use simple predicate functions that return `TRUE` or `FALSE` in combination with `discard` or `keep` to perform appropriate selections. For example, we can exclude the id column of the patients data set, before computing the variable–wise means.

```
keep(pat, is_double)
   # A tibble: 3 × 2
     Height Weight
      <dbl>  <dbl>
   1   1.65     75
   2   1.90     NA
   3   1.60     50
map_dbl(discard(pat, is_character), mean, na.rm = TRUE)
   Height Weight
     1.72  62.50
```

Note that we specified `na.rm = TRUE` as an additional argument to the map function. This will be directly passed on to the argument list of `mean` and excludes any missing values before computing the means.

**Exercise: Handling a small data set**

- Import the data set `Patients.csv` from the website

http://www-huber.embl.de/users/klaus/BasicR/Patients.csv

- Which variables are stored in the data frame and what are their values?
- Is there a missing weight value? If yes, replace it by the mean of the other weight values.
- Calculate the mean weight and height of all the patients.
- Calculate the **BMI = Weight / Height^2** of all the patients.

# 7 Simple plotting in R: qplot of *ggplot2*

The package *ggplot2* allows very flexible plotting in R, but takes a while to get acquainted with the underlying grammar of graphics. Thus, we will use its function `qplot()` for "quick plotting", which requires no knowledge of the underlying advanced features and behaves much like R's default `plot` function. However, it offers advanced options like faceting or coloring by condition as well.

```
qplot(x, y = NULL, ..., data, facets = NULL,
  NA), ylim = c(NA, NA), log = "", main = NULL,
  xlab = , ylab = )
```

The arguments are:

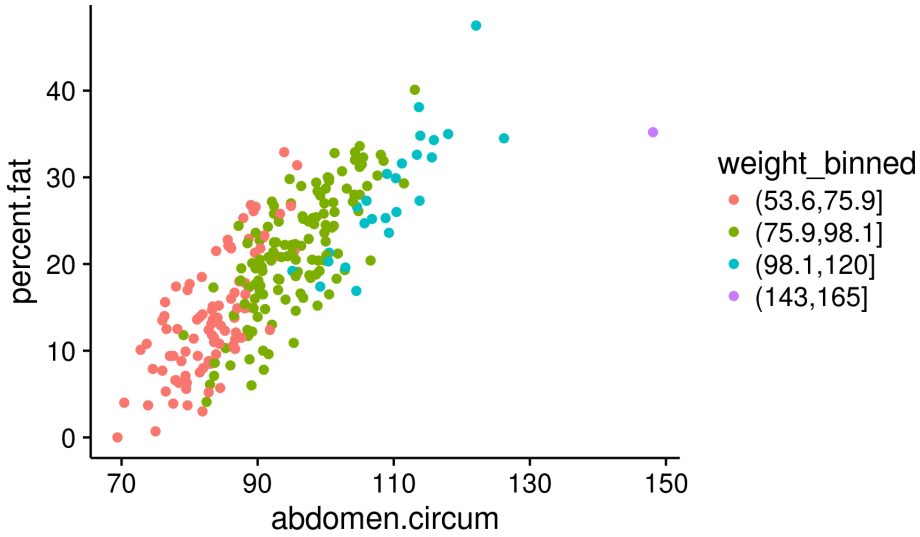
- `x:` x–axis data
- `y:` y–axis data (may be missing)
- `data:` `data.frame` containing the variables used in the plot
- `facets` split the plot into facets, use a formula like . ~split to do wrapped splitting and row ~ columns to split by rows and columns
- `main:` plot heading

- `color, fill` set to factor/string in the data set in order to color the plot depending on that factor. Use `I("colorname")` to use a specific color.
- `geom` specify a "geometry" to be used in the plots, examples include point, line, boxplot, histogram etc.
- `xlab, ylab, xlim, ylim` set the x–/y–axis parameters

As an example, we create a plot of `perc.fat` against abdomen circumference and color it by weight. For this we bin the weight vector into 5 discrete categories using the `cut` function.

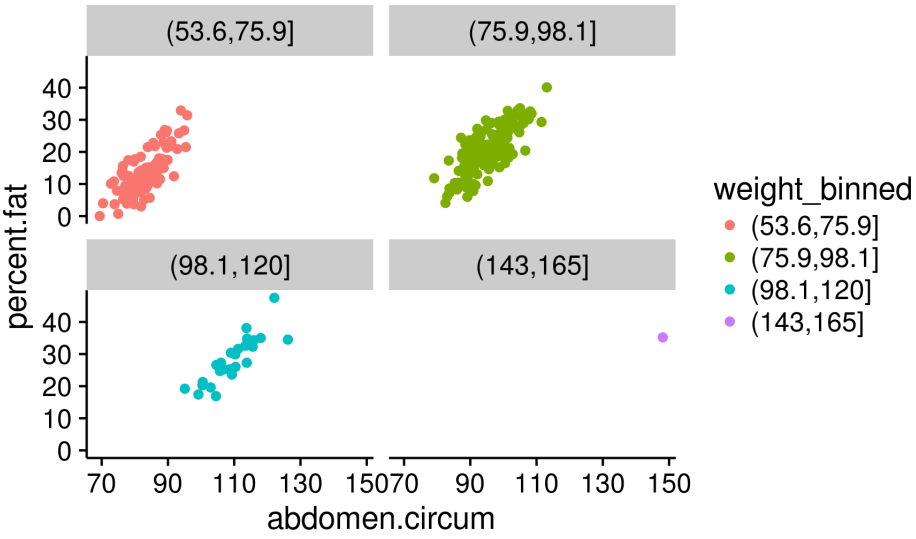
```
bodyfat <- mutate(bodyfat, weight_binned = cut(weight_kg, 5))

qplot(abdomen.circum, percent.fat,
      color = weight_binned, data = bodyfat)
```




We can (unsurprisingly) see that abdomen circumference, weight and bodyfat are highly correlated to each other. We can also produce a faceted plot split by weight.

```
qplot(abdomen.circum, percent.fat,
      color = weight_binned, data = bodyfat,
      facets = ~weight_binned)
```

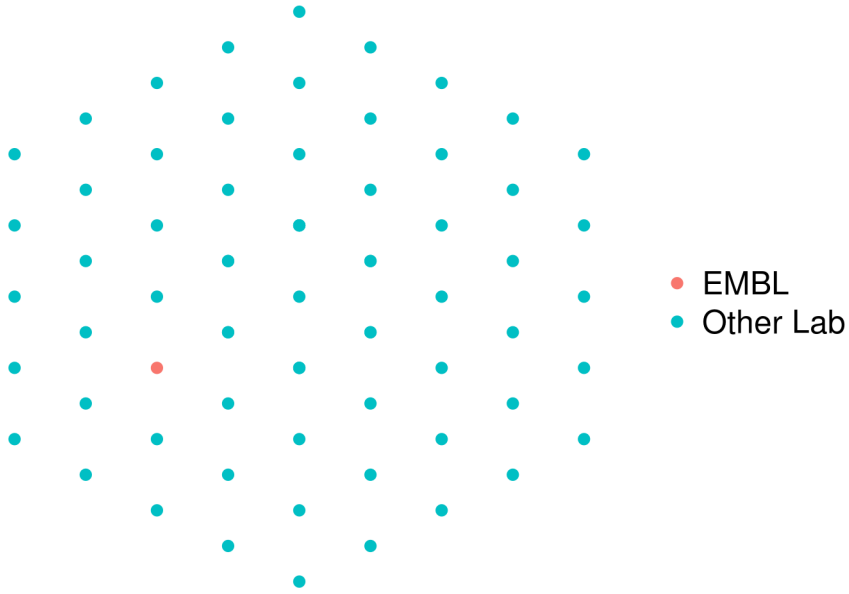



**Exercise: Plotting the EMBL logo**

The code below plots the embl logo. The plus sign adds additional "layers" to the ggplot object modifying any given plot and we use it here to make the axes disappear.

However the colors are not quite right. Can you fix that? Check out the ggplot2 docs or try googeling!

```
load("hex_grid.Rdata")
embl_colors <- c("#E2001A", "#6FAA46")
qplot(x, y, data = hex_grid, color = lab, asp = 1) +
    theme(panel.border = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          axis.text = element_blank(),
          line = element_blank(),
          title = element_blank())
```




# 8 Programming statements

R offers the typical options for flow–control known from many other languages.

The most important one, the **if–statement** is used when certain computations should only be performed if a certain condition is met (and maybe something else should be performed when the condition is not met):

```
w <- 3
  if (w < 5) {
      d <- 2
    } else {
      d <- 10
    }
d
   [1] 2
```

If you want perform a computation for every entry of a list, you usually do the computations for one time step and then for the next and the next, etc. Because nobody wants to type the same commands over and over again, these computations are automated in **for–loops**.

```
h <- seq(from = 1, to = 8)
s <- numeric() # create empty  vector
    for (i in 1:8)
    {
    s[i] <- h[i] * 10
    }
s
   [1] 10 20 30 40 50 60 70 80
```

Note however, that you should typically resort to `map` function for this purpose as this leads to more readable code:

```
map_dbl(h, ~.x*10)
   [1] 10 20 30 40 50 60 70 80
```

Another useful command is the **ifelse**–command, it replaces elements of a vector based on the evaluation of another logical vector of the same size. This is useful to replace missing values, or to binarize a vector.

```
s <- seq(from = 1, to = 10)
binary_s <- ifelse(s > 5, "high", "low")
binary_s
    [1] "low"  "low"  "low"  "low"  "low"  "high" "high" "high" "high" "high"
```

**Exercise: Base calling errors**

The function `readError(noBases)` simulates the base calling errors of a sequencing machine. The parameter **noBases** represents the number of positions in the genome sequenced and the function will return a vector, which has the entry "error" if a base calling error occurs at a certain position and "correct" if the base is read correctly. It can be obtained with the command

```
source("http://www-huber.embl.de/users/klaus/BasicR/readError.R")
```

- Let the sequencer read a thousand positions and try to infer a base calling error rate from this simulation HINT: The functions `table` and `prop.table` could be handy for this!
- Let us assume the technology improves and the machine is less prone to errors. Change the function accordingly!

# 9 Answers to exercises

**Exercise: Simple R operations**

Use the rep function to define simply the following vectors in R.

- (6, 6, 6, 6, 6, 6)
- (5, 8, 5, 8, 5, 8, 5, 8)
- (5, 5, 5, 5, 8, 8, 8, 8)

**Solution: Simple R operations**

```
rep(6, 6)
rep(c(5, 8), 4)
c(rep(5, 4), rep(8, 4))
```

**Exercise: R as a calculator** Calculate the following expression, where `x` and `y` have values `-0.25` and `2` respectively. Then store the result in a new variable and print its content.

```
x + cos(pi/y)
   Warning in cos(pi/y): NaNs produced
```

**Solution: R as a calculator**

```
x <-  -0.25
y <- 2
x + cos(pi/y)
```

**Exercise: Milk sales**

The vector y<-c(33, 44, 29, 16, 25, 45, 33, 19, 54, 22, 21, 49, 11, 24, 56) contains sales of milk in liters for 5 days in three different shops (the first 3 values are for shops 1, 2 and 3 on Monday, etc.). Produce a statistical summary of the sales for each day of the week and also for each shop.

**Solution: Milk sales**

```
y <- c(33, 44, 29, 16, 25, 45, 33, 19, 54, 22, 21, 49, 11, 24, 56)
# day of the week summary, example: Tuesday
Tuesday <- y[ (1:3) + 3 ]
summary(Tuesday)
## Shop 2 summary
Shop2 <- y[ seq(2, length(y), by = 3 )  ]
summary(Shop2)

## alternative: turn into a tibble and use map functions

y_mat <- t(matrix(y, nrow = 3, ncol = 5, byrow = FALSE))
y_tibble <- as_tibble(y_mat)
names(y_tibble) <- c("Shop_1", "Shop_2", "Shop_3")
map(y_tibble, summary)

# summary by days

days <- matrix(y, nrow = 3, ncol = 5, byrow = FALSE)
days <- as_tibble(days)
names(days) <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday")
map(days, summary)
```

**Exercise: Handling a small data set**

- Import the data set `Patients.csv` from the website

http://www-huber.embl.de/users/klaus/BasicR/Patients.csv

- Which variables are stored in the data frame and what are their values?
- Is there a missing weight value? If yes, replace it by the mean of the other weight values.
- Calculate the mean weight and height across all the patients.
- Calculate the **BMI = Weight / Height^2** of all the patients.

**Solution: Handling a small data set**

```
pat <- read_csv("http://www-huber.embl.de/users/klaus/BasicR/Patients.csv")
pat

pat$Weight[is.na(pat$Weight)] <- mean(pat$Weight, na.rm = TRUE)
```

```
pat

map_dbl(keep(pat, is_double), mean, na.rm = TRUE)

pat <- mutate(pat, BMI = Weight / Height^2)
```

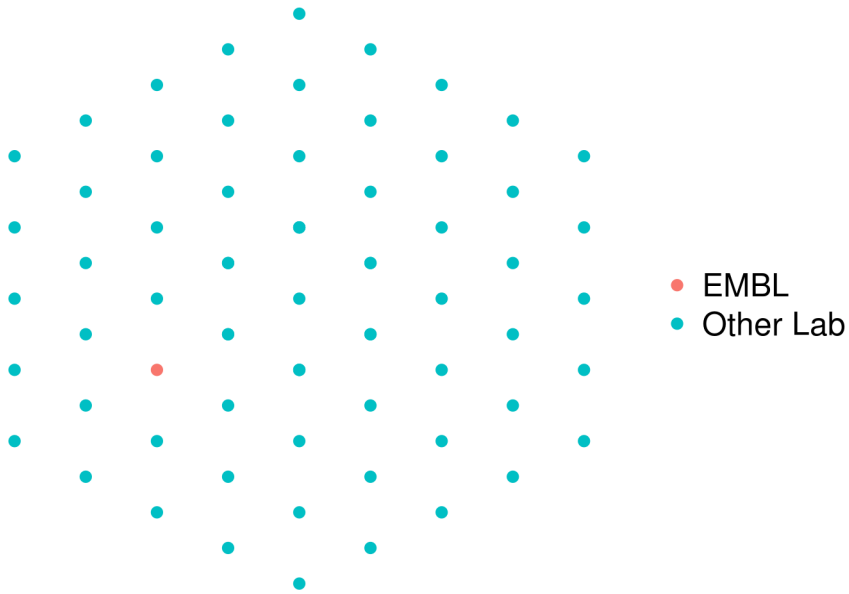
**Exercise: Plotting the EMBL logo**

The code below plots the embl logo. Note that the plus sign adds additional layers to the ggplot object. This allows you to modify any given plot. However the colors are not quite right. Can you fix that? Check out the ggplot2 docs or try googeling!

```
load("hex_grid.Rdata")

embl_colors <- c("EMBL" = "#E2001A", "Other Lab" = "#6FAA46")

qplot(x, y, data = hex_grid, color = lab, asp = 1) +
    theme(panel.border = element_blank(),
          panel.grid.major = element_blank(),
          panel.grid.minor = element_blank(),
          axis.text = element_blank(),
          line = element_blank(),
          title = element_blank())
```




**Solution: Plotting the EMBL logo**

```
load("hex_grid.Rdata")
embl_colors <- c("EMBL" = "#E2001A", "Other Lab" = "#6FAA46")

qplot(x, y, data = hex_grid, color = lab, asp = 1) +
    theme(panel.border = element_blank(),
          panel.grid.major = element_blank(),
```

```
            panel.grid.minor = element_blank(),
            axis.text = element_blank(),
            line = element_blank(),
            title = element_blank()) +
      scale_colour_manual(values = embl_colors )
```

**Exercise: Base calling errors** The function `readError(noBases)` simulates the base calling errors of a sequencing machine. The parameter `noBases` represents the number of positions in the genome sequenced and the function will return a vector, which has the entry "error" if a base calling error occurs at a certain position and "correct" if the base is read correctly. It can be obtained with the command

```
source("http://www-huber.embl.de/users/klaus/BasicR/readError.R")
```

- Let the sequencer read a thousand positions and try to infer a base calling error rate from this simulation HINT: The functions `table` and `prop.table` could be handy for this!
- Let us assume the technology improves and the machine is less prone to errors. Change the function accordingly!

**Solution: Base calling errors**

```
source("http://www-huber.embl.de/users/klaus/BasicR/readError.R")

test <- readError(1000)
## number of errors
sum(test == "error")
##  error probability
sum(test == "error") / 1000

prop.table(table(test))


readError2 <- function(noBases){

  positions <- integer(noBases) ## initialize vector
  for (i in 1:noBases ) {
    positions[i] <- rbinom(n = 1, size = 1, prob = 0.05)
  }
    return(ifelse(positions, "correct", "error"))
  }

### equivalent function
rbinom(n = 1000, size = 1, prob = 0.05)
```

```
sessionInfo()
   R version 3.4.0 (2017-04-21)
   Platform: x86_64-pc-linux-gnu (64-bit)
   Running under: Ubuntu 16.04.2 LTS

   Matrix products: default
   BLAS: /usr/lib/libblas/libblas.so.3.6.0
```

```
          LAPACK: /usr/lib/lapack/liblapack.so.3.6.0

          locale:
           [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
           [3] LC_TIME=de_DE.UTF-8        LC_COLLATE=en_US.UTF-8
           [5] LC_MONETARY=de_DE.UTF-8    LC_MESSAGES=en_US.UTF-8
           [7] LC_PAPER=de_DE.UTF-8       LC_NAME=C
           [9] LC_ADDRESS=C               LC_TELEPHONE=C
          [11] LC_MEASUREMENT=de_DE.UTF-8 LC_IDENTIFICATION=C

          attached base packages:
          [1] parallel  stats     graphics  grDevices utils     datasets  methods
          [8] base

          other attached packages:
           [1] cowplot_0.7.0       dplyr_0.5.0         purrr_0.2.2
           [4] readr_1.1.0         tidyr_0.6.1         tibble_1.3.0
           [7] ggplot2_2.2.1       tidyverse_1.1.1     multtest_2.31.0
          [10] Biobase_2.35.1      BiocGenerics_0.21.3 openxlsx_4.0.17
          [13] TeachingDemos_2.10  knitr_1.15.1        BiocStyle_2.3.32

          loaded via a namespace (and not attached):
           [1] reshape2_1.4.2   splines_3.4.0    haven_1.0.0      lattice_0.20-35
           [5] colorspace_1.3-2 htmltools_0.3.6  stats4_3.4.0     yaml_2.1.14
           [9] survival_2.41-3  foreign_0.8-68   DBI_0.6-1        modelr_0.1.0
          [13] readxl_1.0.0     plyr_1.8.4       stringr_1.2.0    munsell_0.4.3
          [17] gtable_0.2.0     cellranger_1.1.0 rvest_0.3.2      codetools_0.2-15
          [21] psych_1.7.3.21   evaluate_0.10    labeling_0.3     forcats_0.2.0
          [25] curl_2.5         broom_0.4.2      Rcpp_0.12.10     backports_1.0.5
          [29] scales_0.4.1     jsonlite_1.4     mnormt_1.5-5     hms_0.3
          [33] digest_0.6.12    stringi_1.1.5    bookdown_0.3     grid_3.4.0
          [37] rprojroot_1.2    tools_3.4.0      magrittr_1.5     lazyeval_0.2.0
          [41] MASS_7.3-47      Matrix_1.2-8     xml2_1.1.1       lubridate_1.6.0
          [45] assertthat_0.2.0 rmarkdown_1.5    httr_1.2.1       R6_2.2.0
          [49] nlme_3.1-131     compiler_3.4.0
```