Data Structures I

Introduction

Objects in R represent either informal data types (i.e., they do not explicitly belong to a formal class) or formal classes. In the latter case, class-specific methods can be defined.

Informal Data Types

Examples of informal data types:

- Vectors: Ordered set of elements of the same type (atomic)
- Matrices: 2-dimensional array of elements of the same type (atomic)
- Lists: ordered set of components that may belong to different classes (recursive, i.e., list components can be lists themselves)

Formal Classes

Examples of formal classes:

- Factor: Vector representing categorical values
- Data Frame: 2-dimensional array that represents a data matrix (the classic form of data in statistics where the rows represent observational units and the columns represent variables that can be of different modes)
- Objects of class `lm`: Data objects containing the results of a linear model fit
- Objects of class `glm`: Data objects containing the results of a generalized linear model fit
- . . .

Mode and storage mode

The *mode* of objects determines which values of a data element can be stored in principle. The *storage mode* indicates the form of internal representation of the data.
Examples for these concepts are presented below (focussing on vectors).

Vectors

Vectors combine elements of the same mode to a sequence of finite length. Each vector element is clearly defined by an index number.

Vector creation and description

We already saw in the *R Introduction* notebook how objects can be created using an assignment operator. If we want to create an object that is an vector, we have several possibilities, including the one we used in the aforementioned document:

```
x <- 5
```

Is this a vector object? Yes, because there are no scalars in the classical meaning, they are represented by vectors of length 1. Above, we generated the vector without declaration of the mode. If we want to know its mode, we type:

```r
mode(x)
```

```
## [1] "numeric"
```

R automatically assigned the mode `numeric` (it tried to “guess” the appropriate mode), but we could also use the following line of code to implicitely declare it:

```r
x <- numeric(1) # numeric vector of length 1
mode(x)
```

```
## [1] "numeric"
```

```r
typeof(x) # storage mode
```

```
## [1] "double"
```

Internally, the vector is stored with double precision (floating-point format). If we only need integer values, we could use the following code instead:

```r
x <- 5L
mode(x)
```

```
## [1] "numeric"
```

```r
typeof(x)
```

```
## [1] "integer"
```

We can create vectors of other modes like this:

```r
x <- logical(1) # TRUE/FALSE vector of length 1
mode(x)
```

```
## [1] "logical"
```

```r
x <- character(1) # character/string vector of length 1
mode(x)
```

```
## [1] "character"
```

As a third possibility, we could use the `vector` function:

```r
x <- vector(mode = "numeric", length = 1) # numeric vector of length 1
```

Every object contains certain meta information (apart from the already known `mode`):

```r
length(x) # Vector length
```

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

```r
is.vector(x) # TRUE
```

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

```r
is.vector(x, "numeric") # TRUE, since numeric
```

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

```r
is.vector(x, "logical") # FALSE, since numeric
```

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

```r
is.numeric(x) # TRUE, if x is an object of type 'vector' and mode 'numeric'
```

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

These characteristics are dynamic which means that they may be changed - values as well as type ("coercion") and length.

In addition, objects may have additional attributes. We can access them using `attr()` (individually) or `attributes()` (all at once):

```r
attr(x, "names") # x has no "names" attribute
```

```
## NULL
```

```r
attributes(x) # x has no attributes at all
```

```
## NULL
```

Using the `names()` function, it is possible to name the elements of a vector. By doing so, we set the "names" attribute:

```r
names(x) <- "a"
x
```

```
## a
## 0
```

```r
attr(x, "names")
```

```
## [1] "a"
```

```r
attributes(x)
```

```
## $names
## [1] "a"
```

Vectors of length > 1 are commonly created with the "combine"-function `c()`:

```r
y <- c(4, 67, 1)
y
```

```
## [1]  4 67  1
```

It is possible to use function `c()` to combine already existing vectors or append values to them:

```r
y <- c(y, 1)
y
```

```
## [1]  4 67  1  1
```

```r
z <- c(44, 33)
yz <- c(y, z)
yz
```

```
## [1]  4 67  1  1 44 33
```

We already got to know object modes "logical", "integer", "double", and "character". There are two other types ("raw" for hexadecimal numbers and "complex" for complex numbers) that we do not use here. These modes are hierarchichally structured like this:

"raw" < "logical" < "integer" < "double" < "complex" < "string"

If we try to combine vectors of different modes, all of them will automatically be coerced to the highest level:

```r
c(y, TRUE, FALSE) # TRUE and FALSE are represented by 1 and 0, respectively
```

```
## [1]  4 67  1  1  1  0
```

```r
c(y, pi)
```

```
## [1]  4.000000 67.000000  1.000000  1.000000  3.141593
```

```r
c(y, "xyz")
```

```
## [1] "4"   "67"  "1"   "1"   "xyz"
```

We can also directly change the mode of a vector:

```r
a <- c("f", "h")
mode(z) <- mode(a)
z
```

```
## [1] "44" "33"
```

The same is possible for the vector length:

```r
length(y) <- length(z)
y # only the first two values are retained
```

```
## [1]  4 67
```

If an object is being interpretable as numbers, a conversion to type “numeric” is possible:

```r
as.numeric(z) # temporary coercion
```

```
## [1] 44 33
```

```r
z
```

```
## [1] "44" "33"
```

```r
z <- as.numeric(z) # permanent coercion
z
```

```
## [1] 44 33
```

Otherwise, we get a warning:

```r
a <- as.numeric(a)
```

```
## Warning: NAs introduced by coercion
```

```r
a
```

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

Warnings mean that the operation is performed, but the result may not be the intended one. Here, NAs were created. (Warnings contrast *errors* that indicate that it was **not** possible to perform the operation.)

```r
a <- as.character(a)
a
```

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

It is not possible to backtransform, the original information is lost!

Vector arithmetic

It is possible to use vectors in arithmetic expressions, e.g.:

```r
x <- c(33, 44)
y <- c(55, 66)
x + y
```

```
## [1]  88 110
```

We see that the addition is performed element by element, i.e., `44 + 55` and `33 + 66`, respectively. The vectors do not need to have the same length. If one of the vectors is shorter, it is *recycled* until it matches the length of the longer vector ("recycling rule"), e.g.:

```
x <- c(33, 44, 55, 66)
x + y # 33 + 55, 44 + 66, 55 + 55, 66 + 66
```

```
## [1]  88 110 110 132
```

The recycling rule is also applied if the length of the longer vector is not a multiple of the length of the shorter vector, but we get a warning message:

```
x <- c(33, 44, 55)
x + y # 33 + 55, 44 + 66, 55 + 55
```

```
## Warning in x + y: longer object length is not a multiple of shorter object
## length
```

```
## [1]  88 110 110
```

Sequences and replications

The simplest way to generate a sequence is by usage of the colon operator `:`, e.g.:

```
1:5
```

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

```
5:1
```

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

```
1.1:5.1
```

```
## [1] 1.1 2.1 3.1 4.1 5.1
```

It can only be used for steps of `1` or `-1`. For more complex sequences, we use the `seq()`-function. We only need to set 3 of the 4 arguments defining the sequence:

```
seq(from = 0, to = 1, by = 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
seq(from = 0, to = 1, length.out = 11)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
seq(to = 1, by = 0.1, length = 11)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
seq(from = 0, by = 0.1, length = 11)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

```
seq(0, 1, 0.1)
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

`rep()` is used to get replications of vectors:

```
rep(x = 1, times = 10)
```

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

```
rep(1:5, 5)
```

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

```
rep(1:5, 1:5)
```

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

```
rep(1:5, 5:1)
```

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

Vector arithmetic revisited:

```
1:5 + 2 # 1 + 2, 2 + 2, 3 + 2 etc.
```

```
## [1] 3 4 5 6 7
```

```
1:5 * 2 # 1 * 2, 2 * 2, 3 * 2 etc.
```

```
## [1]  2  4  6  8 10
```

Exponent has as stronger tie than the colon operator:

```
1:5 ^ 2 # 1:25
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

```
(1:5) ^ 2 # 1 ^ 2, 2 ^ 2, 3 ^ 2 etc.
```

```
## [1]  1  4  9 16 25
```

```
2 ^ 1:5
```

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

```
(2 ^ 1):5
```

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

```
2 ^ (1:5) # 2 ^ 1, 2 ^ 2, 2 ^ 3 etc.
```

```
## [1]  2  4  8 16 32
```

`sample()` generates vectors of random numbers:

```
sample(0:1, 20, replace = TRUE)
```

```
##  [1] 0 1 1 0 1 1 1 1 1 0 1 1 1 0 0 0 0 1 1 0
```

```
sample(0:1, 20, replace = TRUE)
```

```
##  [1] 0 0 0 0 0 1 0 1 1 1 0 1 0 0 0 0 1 0 0 0
```

```
sample(0:1, 20, replace = TRUE)
```

```
##  [1] 0 1 0 0 0 0 1 0 0 0 1 0 1 1 1 0 1 1 0 0
```

```
sample(0:1, 20)
```

```
## Error in sample.int(length(x), size, replace, prob): cannot take a sample larger than the population
```

```
# error, sampling without replacement is only possible two times here!
```

The numbers are not really random (of course), but pseudo-random. If we know the starting value, the algorithm gives us the same values again:

```r
set.seed(1)
sample(0:1, 20, replace = TRUE)
```

```
##  [1] 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0
```

```r
set.seed(1)
sample(0:1, 20, replace = TRUE)
```

```
##  [1] 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 1 1 1 1 0
```

Set operations

If we have sets of values (given as vectors), we can apply built-in set operations to find out about set unions, intersections etc.:

```r
(x <- 1:10)
```

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

```r
(y <- c(3, 2, 7, 4, 6))
```

```
## [1] 3 2 7 4 6
```

```r
setdiff(x, y) # select elements that are part of x, but not y
```

```
## [1]  1  5  8  9 10
```

```r
setdiff(y, x)
```

```
## numeric(0)
```

```r
union(x, y) # union of all elements in x and y (no duplicates!)
```

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

```r
(z <- 5:15)
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

```r
union(x, z)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

```r
intersect(x, y)
```

```
## [1] 2 3 4 6 7
```

```r
# which elements are part of x and y? each element of x is compared
setequal(x, y)  # are all elements of x and y the same?
```

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

```r
(a <- 5:15)
```

```
##  [1]  5  6  7  8  9 10 11 12 13 14 15
```

```r
setequal(z, a)
```

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

```r
(b <- 15:5)
```

```
##  [1] 15 14 13 12 11 10  9  8  7  6  5
```

```r
setequal(z, b)
```

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

```r
is.element(x, y) # are the single elements of x contained in y?
```

```
##  [1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE
```

```r
x %in% y # the same
```

```
##  [1] FALSE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE
```

```r
which(x %in% y) # which elements of x are contained in y?
```

```
## [1] 2 3 4 6 7
```

Sorting

We use `sort()` to sort a vector:

```r
set.seed(1)
x <- sample(1:100, 20)
x
```

```
##  [1] 68 39  1 34 87 43 14 82 59 51 85 21 54 74  7 73 79 37 83 97
```

```r
sort(x)
```

```
##  [1]  1  7 14 21 34 37 39 43 51 54 59 68 73 74 79 82 83 85 87 97
```

```r
sort(x, decreasing = TRUE) # sort in reverse order
```

```
##  [1] 97 87 85 83 82 79 74 73 68 59 54 51 43 39 37 34 21 14  7  1
```

```r
y <- sample(letters, 10) # sample 10 lower-case letters
y
```

```
##  [1] "j" "y" "l" "o" "a" "t" "c" "f" "z" "r"
```

```r
sort(y)
```

```
##  [1] "a" "c" "f" "j" "l" "o" "r" "t" "y" "z"
```

`order()` returns an order permutation, i.e., provides information about the indices of the lowest, second-lowest, . . . , highest values in the original vector:

```r
set.seed(1)
x
```

```
##  [1] 68 39  1 34 87 43 14 82 59 51 85 21 54 74  7 73 79 37 83 97
```

```r
order(x)
```

```
##  [1]  3 15  7 12  4 18  2  6 10 13  9  1 16 14 17  8 19 11  5 20
```

The lowest value (1) is found on position number 3 of `x`, the second-lowest (7) on position number 15 etc. `order()` can also be used to do multiple sorting:

```r
x <- c(0, 0, 1, 1)
y <- c(1, 0, 1, 0)
order(x, y) # (0, 0), (0, 1), (1, 0), (1, 1)
```

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

`rank()` gives us the ranks of the values in a vector. If there are no bindings (equal values of two or more elements), `order` and `rank` give us the same results, e.g.:

```r
z <- 4:1
order(z)
```

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

```r
rank(z)
```

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

With bindings present, we get:

```r
x
```

```
## [1] 0 0 1 1
```

```r
order(x)
```

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

```r
rank(x)
```

```
## [1] 1.5 1.5 3.5 3.5
```

Per default, the ranks are the means of the ties (`(1 + 2) / 2` and `(3 + 4) / 2`, respectively).

Subsetting

If we want to extract certain values of a vector, we use the `[` operator:

```r
x <- c(5, 2, 7)
x[3] # Third element of x
```

```
## [1] 7
```

```r
x[1:2] # First and second element of x
```

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

```r
x[-1] # all elements of x except the first one
```

```
## [1] 2 7
```

```r
x[-c(1:2)] # all elements of x except the first two
```

```
## [1] 7
```

```r
x[-1:2] # negative index mixed with positive indices -> error!
```

```
## Error in x[-1:2]: only 0's may be mixed with negative subscripts
```

```r
y <- 1:2 # index vector
x[y]
```

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

```r
x[-y]
```

```
## [1] 7
```

```r
y <- c(TRUE, TRUE, FALSE) # Boolean index vector
x[y]
```

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

```
x[-y] # caution!
```

```
## [1] 2 7
```

```
-y
```

```
## [1] -1 -1  0
```

```
z <- 1:5
z[c(TRUE, FALSE)] # recycling rule
```

```
## [1] 1 3 5
```

Combining the `[` operator with the assigment operator `<-`, we can replace vector elements:

```
x[3] <- 1000 # replacement of the third element
x
```

```
## [1]    5    2 1000
```

```
x[-1] <- c(1000, 2000) # replacement of two elements
x
```

```
## [1]    5 1000 2000
```

```
x[1:3] <- c(1000, 2000) # recycling rule also applies here
```

```
## Warning in x[1:3] <- c(1000, 2000): number of items to replace is not a multiple
## of replacement length
```

```
x
```

```
## [1] 1000 2000 1000
```

Usually, Boolean values are not directly entered, but result from the usage of logical operators (see below):

```
x <- 10:1
x == 3 # element of x equals 3 (T/F)?
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
```

```
x > 3 # element of x is greater 3 (T/F)?
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
```

```
x <= 3 # element of x is less or equal 3?
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
```

```
x[x == 3] # return x where x == 3?
```

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

```
y <- x == 3
x[y] # the same
```

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

```
x[x == 3] <- 30
x
```

```
##  [1] 10  9  8  7  6  5  4 30  2  1
```

```
x[x > 3] # only elements that show TRUE are returned
```

```
## [1] 10  9  8  7  6  5  4 30
```

Index vectors

There are a number of functions that can be used to get index vectors for certain values of interest:

```
which.min(x) # index of the lowest value
```

```
## [1] 10
```

```
which.max(x) # index of the highest value
```

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

```
which(x > 3)
```

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

```
x[which(x > 3)] # x where x greater 3
```

```
## [1] 10  9  8  7  6  5  4 30
```

```
x[which.min(x)] # lowest value of x
```

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

```
min(x) # the same
```

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

```
x[which.max(x)] # highest value of x
```

```
## [1] 30
```

```
max(x) # the same
```

```
## [1] 30
```

Logical operators

Apart from the already known operators (`==`, `>` etc.), this ones are of special importance:

```
x <- 1:10
# not equal; ! indicates NOT (logical negation):
x != 3
```

```
##  [1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```
# logical AND -> both conditions are TRUE (TRUE, TRUE):
x > 3 & x < 9
```

```
##  [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

```
# logical OR -> at least one of the conditions is TRUE
# (TRUE, TRUE or TRUE, FALSE or FALSE, TRUE):
x > 3 | x < 9
```

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

```
y <- 6:15
# exactly one of the conditions is TRUE (TRUE, FALSE or FALSE, TRUE):
xor(x > 4, y < 8)
```

```
##  [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Caution: `==` and `!=` do not allow for a finite representation of fractions, nor for rounding error. Because of that, surprising answers may occur:

```
x1 <- 0.5 - 0.3
x2 <- 0.3 - 0.1
x1 == x2 # FALSE on most machines
```

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

Using `all.equal()` and/or `identical()` is almost always preferable in this case, depending on the scope of the test for equality (exact equality or near equality -> see the help pages for details).

Moreover, `==` and `!=` also shouldn't be used in `if` expressions. These ask for a single `TRUE` or `FALSE`, but the mentioned operators may either return vectors that are not of length 1 or NAs.

Misc. functions

Differences between vector elements:

```
x <- 1:10
diff(x)
```

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

Sums and products of elements:

```
sum(x)
```

```
## [1] 55
```

```
prod(x)
```

```
## [1] 3628800
```

Cumulated sum:

```
cumsum(x)
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```
diff(cumsum(x))
```

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

```
cumsum(diff(x))
```

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

Rounding:

```
pi
```

```
## [1] 3.141593
```

```
round(pi) # zero decimal places
```

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

```
round(pi, 2) # two decimal places
```

```
## [1] 3.14
```

```
x <- 11111
round(x, -1)
```

```
## [1] 11110
```

```r
round(x, -3)
```

```
## [1] 11000
```

```r
y <- -0.99
round(y)
```

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

```r
ceiling(y)
```

```
## [1] 0
```

```r
floor(y)
```

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

```r
trunc(y)
```

```
## [1] 0
```

String operations

Number of characters for each element:

```r
x <- c("John Doe", "Jane Doe")
nchar(x)
```

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

Concatenate vectors:

```r
paste("My name is", x)
```

```
## [1] "My name is John Doe" "My name is Jane Doe"
```

```r
age <- c(25, 30)
y <- paste(x, "is", age, "years old.")
y
```

```
## [1] "John Doe is 25 years old." "Jane Doe is 30 years old."
```

```r
length(y)
```

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

```r
z <- paste0(y, collapse = " ")
z
```

```
## [1] "John Doe is 25 years old. Jane Doe is 30 years old."
```

```r
length(z)
```

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

Substring:

```r
substring(x, 1, 4) # first element, length 4
```

```
## [1] "John" "Jane"
```

Abbreviation (defaults to four characters):

```
abbreviate(x)
```

```
## John Doe Jane Doe
##   "JhnD"   "JanD"
```

Factors

Factors are used to code categorical data (e.g., country of birth, place of residence, gender, educational status, grade etc.). We generate a vector of 20 pupils' grades:

```
grades <- sample(1:5, 20, replace = TRUE)
grades
```

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

We use `factor()` to encode this object as a categorical variable:

```
str(grades)
```

```
##  int [1:20] 1 4 1 2 5 3 2 3 3 1 ...
```

```
grades <- factor(grades)
grades
```

```
##  [1] 1 4 1 2 5 3 2 3 3 1 5 5 2 2 1 5 5 1 1 5
## Levels: 1 2 3 4 5
```

We can assign labels to get a more meaningful representation:

```
grades <- factor(grades, labels = c("Sehr gut", "Gut", "Befriedigend",
                                    "Genügend", "Nicht genügend"))
grades
```

```
##  [1] Sehr gut       Genügend       Sehr gut       Gut            Nicht genügend
##  [6] Befriedigend   Gut            Befriedigend   Befriedigend   Sehr gut
## [11] Nicht genügend Nicht genügend Gut            Gut            Sehr gut
## [16] Nicht genügend Nicht genügend Sehr gut       Sehr gut       Nicht genügend
## Levels: Sehr gut Gut Befriedigend Genügend Nicht genügend
```

Internally, the levels are still coded by numbers, which allows for a more efficient way of storing character data. This also allows for a simple back-transformation to numeric values:

```
str(grades)
```

```
##  Factor w/ 5 levels "Sehr gut","Gut",..: 1 4 1 2 5 3 2 3 3 1 ...
```

```
as.numeric(grades)
```

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

But be careful:

```
## Numeric values are sometimes accidentally converted to factors.
## Converting them back to numeric is trickier than you'd expect.
f <- factor(5:10)
f
```

```
## [1] 5  6  7  8  9  10
## Levels: 5 6 7 8 9 10
```

```
# not what you might expect, probably not what you want:
as.numeric(f)
```

```
## [1] 1 2 3 4 5 6
```

```r
# what you typically meant and want:
as.numeric(as.character(f))
```

```
## [1]  5  6  7  8  9 10
```

We could also generate factors from character vectors:

```r
# e.g., grades of four pupils:
grades.ch <- c("Sehr gut", "Gut", "Befriedigend", "Gut")
grades.ch
```

```
## [1] "Sehr gut"     "Gut"          "Befriedigend" "Gut"
```

```r
str(grades.ch)
```

```
##  chr [1:4] "Sehr gut" "Gut" "Befriedigend" "Gut"
```

```r
grades <- factor(grades.ch)
grades
```

```
## [1] Sehr gut     Gut          Befriedigend Gut
## Levels: Befriedigend Gut Sehr gut
```

This leads to two problems: First of all, it is neither possible to use levels that were not defined nor to combine factors:

```r
grades[3] <- "Genügend"
```

```
## Warning in `[<-.factor`(`*tmp*`, 3, value = "Genügend"): invalid factor level,
## NA generated
```

```r
# Problem: levels "Genügend" and "Nicht genügend"
# were not part of the data, and thus, are no valid entries!

# grade of a fifth pupil:
fifth <- factor("Genügend")
# numeric representations of both vectors, but "Genügend"
# also got value '1' because it is the only level of factor 'fifth':
c(grades, fifth)
```

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

Better: Define all possible levels (or at least check if all of them are there when before creating the factor):

```r
# e.g., grades of four pupils:
grades2 <- c("Sehr gut", "Gut", "Befriedigend", "Gut")
grades2 <- factor(grades2, levels = c("Sehr gut", "Gut", "Befriedigend",
                                      "Genügend", "Nicht genügend"))
grades2
```

```
## [1] Sehr gut     Gut          Befriedigend Gut
## Levels: Sehr gut Gut Befriedigend Genügend Nicht genügend
```

Secondly, if the order of the levels is not specified, R automatically proceeds alphabetically:

```r
str(grades)
```

```
##  Factor w/ 3 levels "Befriedigend",..: 3 2 NA 2
```

Here, this leads to results that do not make sense: `Befriedigend` is represented by `1`, `Gut` by `2`, and `Sehr gut` by `3`, which would lead to wrong results when, e.g., computing grade averages! The `ordered()` function can be used to explicitely generate ordered factors:

```
grades2 <- ordered(grades.ch, levels = c("Sehr gut", "Gut", "Befriedigend",
                                          "Genügend", "Nicht genügend"))
grades2
```

```
## [1] Sehr gut     Gut          Befriedigend Gut
## Levels: Sehr gut < Gut < Befriedigend < Genügend < Nicht genügend
```

Count occurences of categories:

```
table(grades.ch) # character vector
```

```
## grades.ch
## Befriedigend          Gut     Sehr gut
##            1            2            1
```

```
table(grades2)    # factor
```

```
## grades2
##       Sehr gut            Gut   Befriedigend       Genügend Nicht genügend
##              1              2              1              0              0
```

Levels that do not occur are dropped when applying `factor()` to an object that already is a factor:

```
grades3 <- factor(grades2)
grades3
```

```
## [1] Sehr gut     Gut          Befriedigend Gut
## Levels: Sehr gut < Gut < Befriedigend
```

Command `cut()` is used to convert numeric vectors to factors by cutting. We put each of 30 individuals aged 20 to 80 into one of two age groups and let R choose the cutoff value:

```
set.seed(1)
age <- sample(20:80, 30)
cut(age, breaks = 2)
```

```
##  [1] (50,80.1] (19.9,50] (50,80.1] (19.9,50] (50,80.1] (19.9,50] (50,80.1]
##  [8] (19.9,50] (19.9,50] (50,80.1] (50,80.1] (19.9,50] (50,80.1] (50,80.1]
## [15] (50,80.1] (19.9,50] (19.9,50] (19.9,50] (19.9,50] (50,80.1] (50,80.1]
## [22] (19.9,50] (50,80.1] (50,80.1] (50,80.1] (50,80.1] (50,80.1] (50,80.1]
## [29] (19.9,50] (19.9,50]
## Levels: (19.9,50] (50,80.1]
```

If we want to determine the cutoff values manually, we have to set a vector of cutoff values (3 for 2 categories, 4 for 3 categories etc.). It is also possible to construct labels directly:

```
age.cat <- cut(age, breaks = c(-Inf, 60, Inf), labels = c("young", "old"))
age.cat
```

```
##  [1] old   young young young young young old   young young old   young young
## [13] old   old   old   young young young young old   young young young old
## [25] old   young old   old   young young
## Levels: young old
```

```
str(age.cat)
```

```
##  Factor w/ 2 levels "young","old": 2 1 1 1 1 1 2 1 1 2 ...
```

Missing values, finite, infinite and NaN numbers

Let us assume that a person is asked about her monthly income and does not want to answer this question. This is a typical example of a *missing value*. There are several reasons why values are missing (unreadable, forgot about, interview partner didn't want to give an answer etc.). In either case, we must flag such values, it is not admissible to substitute them with another value (e.g., a 0). In R, `NA` is used as special symbol that indicates a missing value (similar to a `NULL` value in relational databases, i.e., with respect to basic vectors, `NA` behaves like `NULL` in other languages. `NULL` also exists in R, but will likely not be what you want.). `NA` is available for all data types.

```
x <- 1:10
x[c(1, 7, 8)] <- NA # Insert missing values on positions 1, 3 and 8
x # x with NAs
```

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

```
na.omit(x) # x is returned without NAs; additionally: Position of the missings
```

```
## [1]  2  3  4  5  6  9 10
## attr(,"na.action")
## [1] 1 7 8
## attr(,"class")
## [1] "omit"
```

```
na.pass(x) # x is returned without changes
```

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

```
na.fail(x) # error if x includes NAs (x is not returned)
```

```
## Error in na.fail.default(x): missing values in object
```

```
sum(x) # Statistical functions in R require that all values are nonmissing!
```

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

```
mean(x)
```

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

```
sum(x, na.rm = TRUE) # Remove missing values before the computation
```

```
## [1] 39
```

```
is.na(x) # logical condition for missing values in x
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
```

```
any(is.na(x) == TRUE) # Any missing values in x?
```

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

```
which(is.na(x) == TRUE) # Position of the missing values in x
```

```
## [1] 1 7 8
```

```
1 / 0 # returned value is infinite (Inf)
```

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

```
0 / 0 # returned value in not a number (NaN)
```

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

```
x <- x[1:5]
x[2] <- 0 / 0
x[5] <- 1 / 0
x
```

```
## [1]  NA NaN   3   4 Inf
```

```
is.na(x) # no distinction between NA and NaN
```

```
## [1]  TRUE  TRUE FALSE FALSE FALSE
```

```
# to distinguish 'ordinary' NAs from missing values that result from
# certain calculations:
is.nan(x)
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE
```

```
is.finite(x)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```
is.infinite(x)
```

```
## [1] FALSE FALSE FALSE FALSE  TRUE
```