Data structures

Tim Riffe etal. with slight edits by Jonas Schöley

September 13, 2017

Contents

Introduction

R offers several possibilities to store data in specific kinds of objects. The most important, and the ones we will use later, are:

- vector
- matrix and array
- data frame
- list

We will leave other objects to you as exercises, as appropriate.

Vectors

Vectors are the basic building blocks in R. As we saw in Module 1, a vector consists of a set of elements of the same data type. For example, a numeric vector is a set of numbers. There are several ways to construct a numeric vector. We have seen that the function `c()` can be used to collect elements together into a vector:

```
a <- c(1, 2, 3, 4, 5)
b <- c(6, 7, 8, 9, 10)
```

We can also join the two vectors/objects `a` and `b` using `c()` again:

```
ab <- c(a, b)
ab
```

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

We obtain a different result if we change the order:

```
ba <- c(b, a)
ba
```

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

An oft-required vector is a vector consisting of a regular sequence of numbers. It is possible to construct such a vector by using the operator `:` or the `seq()` function. See different options to build up a sequence of number from 1 to 10 by 1:

```
1:10
seq(from = 1, to = 10, by = 1)
seq(1, 10, 1)
```

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

Note how we can skip the names of the arguments as long as we use them in the right order. It's clear that the first option is faster, whether the second and third are clearer and more flexible. Next examples present some capacities of `seq()`:

```
seq(1, 10, 2)
seq(10, 1, -3)
seq(1, 5, 0.5)
seq(1, 10, length = 4)
seq(1, by = 2, length = 5)
```

```
## [1] 1 3 5 7 9
## [1] 10  7  4  1
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
## [1]  1  4  7 10
## [1] 1 3 5 7 9
```

In the last two lines we do not gives the arguments in their default order, therefore we need to explicitly call them.

Besides sequences, repetitions are often used. R provides here the `rep(x, times)` function, which repeats the value `x` as often as indicated by the integer `times`. In the next examples we also show how functions can be used simultaneously:

```
rep(1, 5)
rep(c(1, 2, 3), 2)
rep(seq(4, 1, -1), 3)
rep(1:3, 1:3)
```

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

In the last example each value in the first argument was repeated the associated value in the second argument.

Likewise `seq()`, the function `rep()` offers additional arguments such as

```
rep(1:3, each = 3)
rep(1:3, length = 10)
```

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

We can extract elements from vectors using square brackets. Let's take the 11th elements of the following vector which contains also missing cases:

```
x <- c(seq(1, 10, 2), NA, seq(5, 15, 2), NA, 100:95)
x
x[11]
```

```
##  [1]   1   3   5   7   9  NA   5   7   9  11  13  15  NA 100  99  98  97
## [18]  96  95
```

```
## [1] 13
```

Of course several elements can be extracted too using within the square brackets vectors which specify the needed positions. We can either internally define the vector of positions:

```
# take 2nd, 5th and 11th elements of x
x[c(2, 5, 11)]
```

```
## [1]  3  9 13
```

or previously define it:

```
# take all elements of x between the 5th and 9th positons
pos <- 5:9
x[pos]
```

```
## [1]  9 NA  5  7  9
```

You can also be more selective. Using the logical operators presented in Module 1, we take all the elements of `x` which are bigger than 50:

```
x[x > 50]
```

```
## [1]  NA  NA 100  99  98  97  96  95
```

Here R chooses those elements of the vector `x` for which the condition `element > 50` is `TRUE`:

```
x > 50
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE    NA FALSE FALSE FALSE FALSE FALSE
## [12] FALSE    NA  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Other two examples with different relational operators:

```
x[x == 15]
x[x != 5]
```

```
## [1] NA 15 NA
##  [1]   1   3   7   9  NA   7   9  11  13  15  NA 100  99  98  97  96  95
```

Note that in this cases, R extracts also `NA` elements from the vector since the logical operator is not able to discriminate between `TRUE` and `FALSE`.

It can be useful to operate with multiple relationships. In the next line we select all the elements of `x` which are equal to 1, 5, 7, or 20:

```
pos1 <- x %in% c(1, 5, 7, 20)
x[pos1]
```

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

Note that obviously R did not find any element equal to 20, (but you're not warned), and `NA` values are not taken into account.

Negative indices can be used to skip certain elements. For example, we can select all but the second element of `x` like this:

```
x[-2]
```

```
##  [1]   1   5   7   9  NA   5   7   9  11  13  15  NA 100  99  98  97  96
## [18]  95
```

The fifth through eleventh elements of `x` can be skipped as follows:

```
x[-(5:11)]
```

```
##  [1]   1   3   5   7  15  NA 100  99  98  97  96  95
```

Here do not forget to use parentheses otherwise you would search for negative positions (try it, see what happens!).

The function `is.na` presented on Module 1, can be used to check for missing cases and, applied its results and a logical negation as index, it can be used to skip `NA` cases:

```
x1 <- x[!is.na(x)]
x1
```

```
##  [1]   1   3   5   7   9   5   7   9  11  13  15 100  99  98  97  96  95
```

Arithmetic can be done on R vectors. For example, we can simply multiply all elements of `x1` by 3, elementwise:

```
x1 * 3
```

```
##  [1]   3   9  15  21  27  15  21  27  33  39  45 300 297 294 291 288 285
```

or add 7 to each element:

```
x1 + 7
```

```
##  [1]   8  10  12  14  16  12  14  16  18  20  22 107 106 105 104 103 102
```

The above examples show how binary arithmetic operator can be used with vectors and constants. In general, the binary operators also work elementwise when applied to pairs of vectors. For example, we aim to compute:

$$y_i^{x_i} \qquad \text{for} \quad i = 1, 2, 3 \qquad \Rightarrow \qquad (y_1^{x_1}, y_2^{x_2}, y_3^{x_3}) \,,$$

as follows with newly created vectors:

```
y <- 5:7
x <- 2:4
y^x
```

```
## [1]   25  216 2401
```

which is equivalent to:

```
c(y[1]^x[1], y[2]^x[2], y[3]^x[3])  # c(5^2, 6^3, 7^4)
```

```
## [1]   25  216 2401
```

Particular attention is needed when the lengths of the two vectors are not equal

Another example is taken from demography. In the following we present a way of computing the Crude Death Rates (CDR) for different populations. In general, for a single year, CDR is defined as

$$CDR = \frac{D[t, t+1)}{PY[t, t+1)}$$

where $D[t, t+1)$ are the number of deaths between the start of year t and the start of year $t+1$ (practically, Jan. 1 and Dec. 31 of the same year), and $PY[t, t+1)$ denotes the person-years lived in the same period, which one might approximate as follows:

$$PY[t, t+1] = \frac{N(t+1) - N(t)}{\ln\left[\frac{N(t+1)}{N(t)}\right]}$$

with $N(t+1)$ and $N(t)$ are the number of persons alive on Jan. 1 and Dec. 31, respectively. More details in Chapter 1 of Preston, Heuveline, and Guillot (2001).

Here we compute the CDR for four female populations in 2008. First we create three vectors with $D[t, t+1)$, $N(t)$ and $N(t+1)$ where t and t+1 are years 2008 and 2009 (both Jan. 1), for Sweden, USA, France and Japan, respectively:

```
D01 <- c(47389, 1245787, 260434, 533696)
N0 <- c(4618852, 153833535, 32048040, 64563312)
N1 <- c(4652510, 155243324, 32217830, 64512583)
```

Then we compute the person-years:

```
PY01 <- (N1 - N0)/log(N1/N0)
```

Note the in R natural logarithms are computed with function `log()`.

Finally CDR (per thousand) can be calculated for all three populations:

```
CDR <- D01/PY01
CDR * 1000
```

```
## [1] 10.222707  8.061397  8.104912  8.269492
```

As we saw in Module 1, R allows also to use vectors which consist of characters or strings and, for instance, function `rep()` can also be applied to them

```
country <- c("Sweden", "USA", "France", "Japan")
rep(country, 2)
rep(country, each = 2)
```

```
## [1] "Sweden" "USA"    "France" "Japan"  "Sweden" "USA"    "France" "Japan"
## [1] "Sweden" "Sweden" "USA"    "USA"    "France" "France" "Japan"  "Japan"
```

The following example shows the flexibility of R in operating with different objects. Given two new objects `sex` and `age` associated each other, we want to compute the mean completed age for a given sex:

```
sex <- factor(c(1, 2, 2, 1, 2, 1, 1, 2))
age <- c(20, 34, 56, 33, 78, 35, 12, 92)
age1 <- age[sex == 1]
mean(age1)
```

```
## [1] 25
```

The vector `sex` was defined as factor: although not necessary for this example, this operation is more natural when dealing with categorical values.

Matrices and Arrays

Matrices

Matrices are two-dimensional structures containing rows and columns. A matrix can be regarded as a collection of vectors of the same length (i.e. the same number of elements) and of the same data type. There are different ways to construct a matrix. Here, we address only two possibilities:

1. using the function `matrix(data,nrow,ncol)` in which `data` is the element we aim to arrange in a matrix, followed by the number of rows and columns for the matrix.
2. "gluing" vectors together. Either one for each column (`cbind`) or for each row (`rbind`).

Let's give some examples for the first possibility:

```
M1 <- matrix(1:20, 4, 5)
M1
M2 <- matrix(1:20, 5, 4)
M2
```

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

Note that the order of the arguments is crucial, and see how the elements are stored internally: down the first column, then down the second and so on. This is known as column-major storage order. Of course we could use already-created objects (e.g. vectors) and arrange them in a matrix:

```
M3 <- matrix(c(age, sex), 8)
M3
```

```
##      [,1] [,2]
## [1,]   20    1
## [2,]   34    2
## [3,]   56    2
## [4,]   33    1
## [5,]   78    2
## [6,]   35    1
## [7,]   12    1
## [8,]   92    2
```

In this case we did not specify the number of columns: given the length of `data`, R will automatically fill up a matrix. This automatic behavior could lead to some issues:

```
M4 <- matrix(c(age, sex), 2)
M4
M5 <- matrix(c(age, sex), 3)
```

```
## Warning in matrix(c(age, sex), 3): data length [16] is not a sub-multiple
## or multiple of the number of rows [3]
```

```
M5
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]   20   56   78   12    1    2    2    1
## [2,]   34   33   35   92    2    1    1    2
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]   20   33   12    2    2    2
## [2,]   34   78   92    2    1   20
## [3,]   56   35    1    1    1   34
```

In `M4` we arrange our data in two rows, but since we work in a column-major storage order, the outcome is not an error, but it is also not necessarily what we would like to have. Then, arranging 16 elements in a 3-row matrix produces the object (`M5`), but also a warning message: the data length (16) is not a sub-multiple or multiple of the number of rows (3). So R creates a matrix but when it finishes the data, it starts recycling

them for filling up the matrix. Bear in mind that, for example, if we fill a 9×9 matrix using a vector with 27 elements, R does not produce any warning: $9 \times 9 = 81 = 27 * 3$, i.e. there is no remainder after recycling $n = 3$ times the vector.

The matrix `M3` can be constructed by gluing/binding together `age` and `sex` by column, but if we aim to have one observation for each column we can bind `age` and `sex` by row:

```
M6 <- cbind(age, sex)
M6
M7 <- rbind(age, sex)
M7
```

```
##      age sex
## [1,]  20   1
## [2,]  34   2
## [3,]  56   2
## [4,]  33   1
## [5,]  78   2
## [6,]  35   1
## [7,]  12   1
## [8,]  92   2
##     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## age   20   34   56   33   78   35   12   92
## sex    1    2    2    1    2    1    1    2
```

Note that these functions preserve the names of the vectors to which they are applied, and this information is preserved in the final matrix as column/row names.

It is noteworthy to point out that both `cbind()` and `rbind()` can be used to bind several matrices. Such matrices must have the same number of columns (rows) when we employ `rbind()` (`cbind()`).

Having data defined as a matrix facilitates analysis in several ways. For example, in a regression setting you can enter a matrix of covariates at once without referring to each single vector/variable. Moreover matrix algebra and associated population analysis can be easily performed with several R functions (covered in another module).

Referencing elements in matrices works similar to referencing elements of vectors. The only thing we have to consider is that matrices are two-dimensional (in contrast to one-dimensional vectors). Consequently two indices are needed. Specifically we refer to an element of a matrix in the following way: `matrix[row_index,column_index]`. As example:

```
M6[5, 1]
```

```
## age
##  78
```

Always, remember: rows first, columns second! Somewhat confusing, R also allows a matrix to be indexed as a vector, using just one value:

```
M6[12]
```

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

To understand what we have selected remember how R fills a matrix by columns. `M6` is a (8×2) matrix, therefore the 12th element refers to the 4th element of the second column.

Whole rows or columns of matrices may be selected by leaving the corresponding index blank:

```
M6[3, ]
M6[, 2]
```

```
## age sex
##  56   2
## [1] 1 2 2 1 2 1 1 2
```

As with vectors, negative referencing excludes specific rows/columns:

```
M1[, -c(3, 5)]
M1[-c(1, 3), ]
```

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

Arrays

Matrices and arrays can be considered special cases of vectors. A matrix is a stored as a vector with an extra attribute, which is its dimension. An array is just the same as a matrix, except it can have more than two dimensions. An example of three-dimensional array could be a collection of data, where death rates are arranged by period and age (i.e.~Lexis coordinates) for several countries.

Arrays can be constructed with `array(data, dimension_vector)` function. An example should clarify this approach. 24 uniform random numbers are arranged into a three-dimensional array with 2 rows, 4 columns and 3 layers:

```
x <- runif(24)
A <- array(x, c(2, 4, 3))
A
```

```
## , , 1
##
##           [,1]      [,2]        [,3]      [,4]
## [1,] 0.2443563 0.2895363 0.124153171 0.7798179
## [2,] 0.1914247 0.4834397 0.006162035 0.7857669
##
## , , 2
##
##           [,1]       [,2]      [,3]      [,4]
## [1,] 0.2760544 0.58517872 0.1070466 0.9675255
## [2,] 0.5662241 0.04557764 0.2879073 0.8295934
##
## , , 3
##
##           [,1]      [,2]      [,3]      [,4]
## [1,] 0.1479960 0.5936895 0.8981055 0.1079660
## [2,] 0.8907068 0.5879948 0.9690538 0.5329825
```

While matrices need two indices to reference elements, arrays need as many indices as there are dimensions. Figure 1 gives a schematic example of the array `A`, with three dimensions. The first index refers to the row, the second to the column and the third to the layer of the array. For instance, the "upper-right" element on the second "layer" could be referenced as follows:

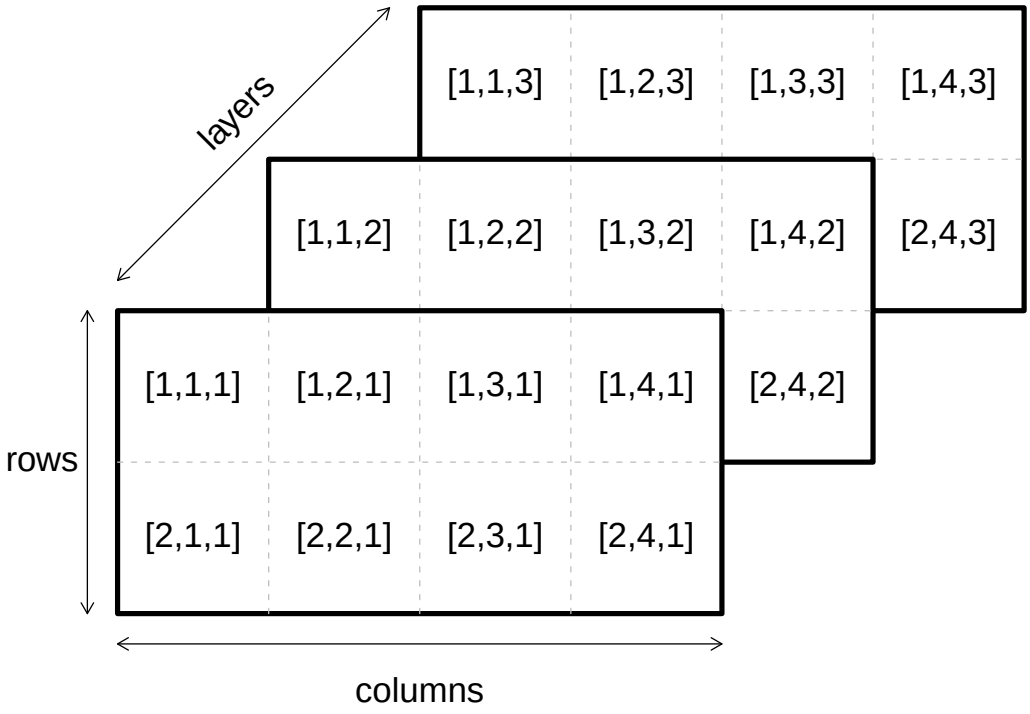

Figure 1: Schematic representation of an array.

```
A[1, 4, 2]
```

```
## [1] 0.9675255
```

Data frame

Data frames represent data in a two-dimensional way like matrices. They are the closest representation of data in R as you may be familiar with from spreadsheets or from software like SPSS or STATA. That means that one row typically represents the values of several variables for one individual/unit. In addition, it is also possible to assign names to each row, and to have different data type for each variable/column. See the following example:

```
DataMort <- data.frame(country = country, PY = PY01, CDR = CDR)
DataMort
```

```
##   country        PY         CDR
## 1  Sweden   4635661 0.010222707
## 2     USA 154537358 0.008061397
## 3  France  32132860 0.008104912
## 4   Japan  64537944 0.008269492
```

The command `merge()` joins two data frames by common columns, also with different order, but with exact name. Here we add to the previous dataframe, values of life expectancy and the infant mortality rate:

```
country1 <- c("USA", "Japan", "Sweden", "France")
e0 <- c(80.69, 86.04, 83.12, 84.36)
m0 <- c(0.0062, 0.0025, 0.0025, 0.0034)
DataMort.extra <- data.frame(country = country1, e0 = e0, IMR = m0)
DataMort.extra
DataMort1 <- merge(DataMort, DataMort.extra, by = "country")
DataMort1
```

```
##   country    e0    IMR
## 1     USA 80.69 0.0062
## 2   Japan 86.04 0.0025
## 3  Sweden 83.12 0.0025
## 4  France 84.36 0.0034
##   country        PY         CDR    e0    IMR
## 1  France  32132860 0.008104912 84.36 0.0034
## 2   Japan  64537944 0.008269492 86.04 0.0025
## 3  Sweden   4635661 0.010222707 83.12 0.0025
## 4     USA 154537358 0.008061397 80.69 0.0062
```

Data are alphabetically reordered based on the variable given to the argument `by()`.

Referencing vectors and elements of a data frame works analogously to referencing vectors and elements in a matrix:

```
DataMort1[-2, ]
DataMort1[2, 3]
```

```
##   country        PY         CDR    e0    IMR
## 1  France  32132860 0.008104912 84.36 0.0034
## 3  Sweden   4635661 0.010222707 83.12 0.0025
## 4     USA 154537358 0.008061397 80.69 0.0062
## [1] 0.008269492
```

There are also other ways to reference columns. Using the `$` sign or the brackets `[[ ]]`, variables can be selected:

```
DataMort1$PY
DataMort1[[2]]
DataMort1[["PY"]]
DataMort1[, 2]
DataMort1[, "PY"]
```

```
## [1]  32132860  64537944   4635661 154537358
## [1]  32132860  64537944   4635661 154537358
## [1]  32132860  64537944   4635661 154537358
## [1]  32132860  64537944   4635661 154537358
## [1]  32132860  64537944   4635661 154537358
```

The selection of rows in a data frame can be done by using brackets and a condition:

```
DataMort1[DataMort1$e0 > 84, ]
```

```
##   country       PY         CDR    e0    IMR
## 1  France 32132860 0.008104912 84.36 0.0034
## 2   Japan 64537944 0.008269492 86.04 0.0025
```

Another possiblity to select values in a data frame is to use the command `subset()`. Let's select all the countries with CDR greater then 0.009:

```
subset(DataMort1, CDR > 0.009)
```

```
##   country      PY        CDR    e0    IMR
## 3  Sweden 4635661 0.01022271 83.12 0.0025
```

We can also use multiple selection taking only a specific variable. For instance, we take the infant mortality rates for all the countries with life expectancy greater than 83 and CDR smaller then 0.01:

```
subset(DataMort1, (e0 > 83) & (CDR < 0.01), IMR)
```

```
##      IMR
## 1 0.0034
## 2 0.0025
```

Note that logical expressions can be combined by using the logical operators `&` (logical "and") and `|` (logical "or").

If you are not sure what the variable names of your data frame are, you can simply ask for them:

```
names(DataMort1)
```

```
## [1] "country" "PY"      "CDR"     "e0"      "IMR"
```

Or for more complete information, try:

```
str(DataMort1)
```

```
## 'data.frame':    4 obs. of  5 variables:
##  $ country: Factor w/ 4 levels "France","Japan",..: 1 2 3 4
##  $ PY     : num  3.21e+07 6.45e+07 4.64e+06 1.55e+08
##  $ CDR    : num  0.0081 0.00827 0.01022 0.00806
##  $ e0     : num  84.4 86 83.1 80.7
##  $ IMR    : num  0.0034 0.0025 0.0025 0.0062
```

This is actually valid for any and all objects, but it's useful for `data.frame` objects because you might not know ahead of time what the column classes are.

Lists

The most flexible way to store values/data in R is a list. It does not put any restrictions on length or type of the data. The elements of a list can be vectors, matrices, data frames, other lists, and anything else in general. If you want to collect all relevant information of one entity in one object the construction of a list is useful.[1]

Let's continue with the demographic example previously presented. We include in the same list mortality data from the dataframe `DataMort1`, we collect in matrix "Total Fertility Rates" and "Mean Age at Childbearing" for the same countries (source: Human Fertility Database (2015)), we specify the data source with a string, and we set the reference date for all these data.

```
TFR <- c(1.991, 1.36, 1.917, 2.074)
MAC <- c(29.89, 30.31, 30.58, 28.02)
DataFert <- rbind(TFR, MAC)
Source <- "Human Mortality and adn Human Fertility Databases"
date <- as.Date("2008/06/30")
DemoList <- list(mortality = DataMort1, fertility = DataFert, Source = Source,
    date = date)
DemoList
```

```
## $mortality
##   country        PY         CDR    e0    IMR
## 1  France  32132860 0.008104912 84.36 0.0034
## 2   Japan  64537944 0.008269492 86.04 0.0025
## 3  Sweden   4635661 0.010222707 83.12 0.0025
## 4     USA 154537358 0.008061397 80.69 0.0062
##
## $fertility
##       [,1]  [,2]   [,3]   [,4]
## TFR  1.991  1.36  1.917  2.074
## MAC 29.890 30.31 30.580 28.020
##
## $Source
## [1] "Human Mortality and adn Human Fertility Databases"
##
## $date
## [1] "2008-06-30"
```

The left indentation of the arguments to `list()`, for example, `fertility =...` is not compulsory, but it helps to structure your coding more tidily.

Referencing elements of a list works analogously to referencing elements in a data frame:

```
DemoList$fertility
DemoList$mortality$IMR
DemoList[[4]]
DemoList[["Source"]]
names(DemoList)
```

```
##       [,1]  [,2]   [,3]   [,4]
## TFR  1.991  1.36  1.917  2.074
## MAC 29.890 30.31 30.580 28.020
## [1] 0.0034 0.0025 0.0025 0.0062
## [1] "2008-06-30"
## [1] "Human Mortality and adn Human Fertility Databases"
```

[1] It turns out that `data.frame` objects are stored as lists where each element is of the same length, such that it can be treated in a tabular way.

```
## [1] "mortality" "fertility" "Source"    "date"
```

A selection of values in a list can be done using the covered selection methods:

```
DemoList$fertility[2, 3]
subset(DemoList$mortality, DemoList$mortality$e0 > 84)
```

```
##   MAC
## 30.58
##   country       PY         CDR    e0    IMR
## 1  France 32132860 0.008104912 84.36 0.0034
## 2   Japan 64537944 0.008269492 86.04 0.0025
```

Basic functions

Getting Information about Data

R has various built-in functions to extract information about objects. These functions are crucial when we load external datasets and/or we aim to work with outcomes of internal R functions. It is always a good habit to ask beforehand the type of object and its dimensions for understanding compatibility with other objects.

The length of an object can be obtained by the function `length()`. Whereas in case of a vector it returns just the number of elements of the vector, in conjunction with a matrix/array gives the total number of elements the matrix/array. The same function applied to a data frame or a list provides the number of columns/variables and the number of list elements, respectively.

The dimension of an object can be retrieved via the function `dim()`. Whereas for a vector and list the dimension is `NULL`, we obtain the number of rows, columns and, possibly, layers for matrices, dataframes, and arrays.

As we have seen in Module 1, `mode()` gives the storage mode of the object. Moreover the function `class()` shows the data type of an object. For example the mode for a matrix might be numeric and its class is matrix itself.

In order to represent the outcomes of these functions for different R objects we have previously built during Module 2, we provide a table in which several examples are simultaneously shown.

Table 1: Example of functions for requiring information from an R object.

Object	`length()`	`dim()`	`mode()`	`class()`
x	24	NULL	numeric	numeric
M1	20	4,5	numeric	matrix
A	24	2,4,3	numeric	array
DataMort1	5	4,5	list	data.frame
DemoList	4	NULL	list	list

Furthermore, remember that if we consider a single element of a list or dataframe, their storage modes and classes are not necessary the same:

```
mode(DemoList)
mode(DemoList$fertility)
```

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

Sometimes a given object needs to be coerced to another, if feasible, R offers several function for converting a specific class to another one. For example if we have a dataframe (always check using `class()`), but we need its element as a matrix:

```
as.matrix(DataMort)
```

```
##      country  PY          CDR
## [1,] "Sweden" "  4635661" "0.010222707"
## [2,] "USA"    "154537358" "0.008061397"
## [3,] "France" " 32132860" "0.008104912"
## [4,] "Japan"  " 64537944" "0.008269492"
```

Of course in this instance all elements will be coerced to strings since a matrix (like a vector) must contain elements of the same kind. Other options for forceing specific classes are: `as.vector()`, `as.matrix()`, `as.data.frame()`, `as.array()`, `as.list()`, (in general `as.<newformat>()`).

An important function for getting a first glance at an R object is `summary()`. It shows a lot of relevant information about an object at once. For example, if applied to a vector, we obtain its median, mean, range and first and third quartile:

```
summary(CDR)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## 0.008061 0.008094 0.008187 0.008665 0.008758 0.010223
```

The same information are obtained for each column/variable when we ask for the summary of a dataframe (or a matrix):

```
summary(DataMort)
```

```
##    country        PY                 CDR
##  France:1   Min.   :  4635661   Min.   :0.008061
##  Japan :1   1st Qu.: 25258560   1st Qu.:0.008094
##  Sweden:1   Median : 48335402   Median :0.008187
##  USA   :1   Mean   : 63960956   Mean   :0.008665
##             3rd Qu.: 87037798   3rd Qu.:0.008758
##             Max.   :154537358   Max.   :0.010223
```

Different outcomes are obtained when the function is applied to a list:

```
summary(DemoList)
```

```
##           Length Class      Mode
## mortality 5      data.frame list
## fertility 8      -none-     numeric
## Source    1      -none-     character
## date      1      Date       numeric
```

Compare with the results of `str()` on the same objects.

Sorting data

Vectors can be sorted by using the function `sort()`. The sorting is ascending by default. For descending, specify `decreasing = TRUE` in the argument list of the `sort`-function. Let's try with the life expectancy:

```
e0
sort(e0)
sort(e0, decreasing = TRUE)
```

```
## [1] 80.69 86.04 83.12 84.36
## [1] 80.69 83.12 84.36 86.04
## [1] 86.04 84.36 83.12 80.69
```

The rows of a data frame can be sorted according to one or more variables using `order()`. This function returns a permutation which rearranges the original vector in ascending or descending order (`decreasing = TRUE`). Let's see what this means using the object `e0`:

```
order(e0)
```

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

The lowest values in `e0` are the first, then the third, the fourth, and the second.

The function `order()` can be used to rearrange the vector `e0` according to `order(ages)` resulting to a sorted vector:

```
e0[order(e0)]
```

```
## [1] 80.69 83.12 84.36 86.04
```

Using this property we are able to sort a data frame according to one variable as follows:

```
DataMort1[order(DataMort1$e0), ]
```

```
##   country        PY         CDR    e0    IMR
## 4     USA 154537358 0.008061397 80.69 0.0062
## 3  Sweden   4635661 0.010222707 83.12 0.0025
## 1  France  32132860 0.008104912 84.36 0.0034
## 2   Japan  64537944 0.008269492 86.04 0.0025
```

If there are more than one variables after which a data frame should be ordered, the sorting procedure can be realized like in the following example. First we construct a new dataframe:

```
weight <- c(100, 120, 100, 120, 120, 80, 100, 120, 100, 100)
height <- c(178, 165, 174, 173, 185, 196, 195, 170, 170, 177)
sex <- c(1, 1, 2, 2, 1, 1, 1, 2, 2, 1)
DF <- as.data.frame(cbind(height, weight, sex))
DF
```

```
##    height weight sex
## 1     178    100   1
## 2     165    120   1
## 3     174    100   2
## 4     173    120   2
## 5     185    120   1
## 6     196     80   1
## 7     195    100   1
## 8     170    120   2
## 9     170    100   2
## 10    177    100   1
```

Now we sort the data frame first by sex and then by weight (within sex):

```
DF[order(DF$sex, DF$weight), ]
```

```
##    height weight sex
## 6     196     80   1
## 1     178    100   1
## 7     195    100   1
## 10    177    100   1
```

```
## 2     165    120   1
## 5     185    120   1
## 3     174    100   2
## 9     170    100   2
## 4     173    120   2
## 8     170    120   2
```

Note that our final output presents all the observations where `sex` equals to 1, and within these, observations are sorted according to their weight in ascending order.

Exercises

Exercise 1

First, construct the following objects:

1. `A`: a vector of 20 numbers from 15 to 250
2. `B`: a vector of length 50 where the integers from 1 to 10 are repeated 5 times

Then compute:

1. the logarithm of base 2 for `A`
2. the logarithm of base 10 for `B`
3. the natural logarithm for `A`

Remember that the function `log(x)` calculates the logarithm of `x`, for a given base. See its documentation.

Exercise 2

We have a population of 10 individuals. We have information on their sex, height, education.

Table 2: Simple data set

ID	sex	height (cm)	education
1	M	178	primary
2	M	168	secondary
3	F	169	primary
4	M	170	tertiary
5	F	157	tertiary
6	M	185	primary
7	F	178	tertiary
8	M	173	secondary
9	F	172	primary
10	M	180	secondary

Create vectors containing the information for this population (`ID`, `sex`, `height`, `educ`), and calculate:

1. the median heights for the males
2. the variance of the heights for the females
3. the mean of the heights for the individuals with tertiary education
4. which individuals are higher than 170 cm

Exercise 3

The Human Mortality Database (2015)} provides detailed mortality and population data. At present the database contains detailed data for a collection of over 35 countries or areas. Some mortality data on Austria can be found in the data directory of this course. The names of the required data files are `popAustria.txt` and `deathsAustria.txt`. The first data set contains the the Austrian population by age (ages range from 0 to 110+ years) for the period 2000-2005. The second data set contains the corresponding number of deaths in Austria.

1. Read the data in.
2. Calculate the total sum of people who were living in Austria in the year 2005.
3. Calculate the total sum of people who died in Austria in the year 2005.
4. What's the crude death rate for Austria in 2005?
5. Select the last three years of the data set (i.e.~2003, 2004, 2005) and calculate the mean age of death for these three years together.
6. Merge the population counts and the death counts for the year 2001 into one data set.
7. Merge the population counts and the death counts for all the available years into one data set.

Exercise 4

1. Create the following data frame `Dat`:

ID	time	age	drug	censor
1	9	46	0	1
1	6	49	1	0
2	10	23	0	1
3	5.5	19	0	1
3	6	29.5	0	1
3	15	28.5	1	1
4	3	34	0	0
5	13	28	1	0

2. Create out of `Dat` a new data frame `Dat2´ that only contains the variables`IDand`drug`‘.
3. Create out of `Dat` a new data frame `Dat3` that only contains the values of the individuals with an `ID` that equals to 1 or 3.
4. Sort the data set `Dat` both descending and ascending according to `time`.

Bibliography

Human Fertility Database. 2015. “Max Planck Institute for Demographic Research (Germany) and Vienna Institute of Demography (Austria).”

Human Mortality Database. 2015. “University of California, Berkeley (USA) and Max Planck Institute for Demographic Research (Germany).”

Preston, Samuel H., Patrick Heuveline, and Michel Guillot. 2001. *Demography: Measuring and Modeling Population Processes*. Oxford: Blackwell.