R syntax in a nutshell – Part II

1.6 Arrays, matrices, data frames

Several kinds of table-like objects exist in R. **Data frames** are data objects to be processed by statistics, with "observations" as columns (elements/oxides in geochemistry) and "cases" (samples) in rows. They can contain columns of any mode, even mixed modes; thus they are not meant for matrix operations.

On the other hand, all elements of a **matrix** can only be of a single mode (numeric, most commonly). **Arrays** are generalized matrices: they must have a single mode but can have any number of dimensions. Although superficially similar, these three types of objects must not be confused.

matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE)

This command defines a matrix of `nrow` rows and `ncol` columns, filled by the data (if `data` have several elements, they will be used down the columns, unless an extra parameter `byrow=TRUE` is provided). For instance:

In [1]:
```r
x <- matrix(1:12,3,4)
print(x)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
```

In [2]:
```r
x <- matrix(1:12,3,4,byrow=TRUE)
print(x)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```

As a default, filling a matrix with data — as well as matrix division by a vector — proceeds along columns, not rows!

array(data = NA, dim = length(data))

Defines a new data array and fills it with `data`. The argument `dim` is a vector of length one or more, giving maximal dimensions in each of the directions.

1.6.1 Matrix/data frame operations

Matrices can be subject to scalar operations using the common operators (`+-*/^`). Similar to vectors, the shorter component is recycled as appropriate. Useful functions for matrix/data frame manipulations are summarized in the following Table:

Function	**Meaning**
`nrow(x)`	number of rows
`ncol(x)`	number of columns
`rownames(x)`	row names
`colnames(x)`	column names
`rbind(x,y)`	binds two objects (matrices or data frames) of the same `ncol` (or vectors of the same length) as rows
`cbind(x,y)`	binds two objects (matrices or data frames) of the same `nrow` (or vectors of the same length) as columns
`t(x)`	matrix transposition
`apply(X,MARGIN,FUN)`	applies function `FUN` (for vector manipulations) along the rows (`MARGIN` = 1) or columns (`MARGIN` = 2) of a data matrix `X`
`x%*%y`	matrix multiplication (does not work on a data frame!)
`solve(A)`	matrix inversion
`dix(x)`	diagonal elements of a matrix

It is worth noting that matrix multiplication is performed using the `%*%` operator.

The function `solve` can serve to solving a set of linear argebraic equations. Here is an example; we shal try to solve the following set of linear equations:

$x + 2y - 0.7z = 21$

$3x + 0.2y - z = 24$

$0.9x + 7y - 2z = 27$

Let's first specify the matrix of coeffients to the individual variables:

In [3]:
```
A <- matrix(c(1,3,0.9,2,0.2,7,-0.7,-1,-2),3,3)
print(A)
```

```
     [,1] [,2] [,3]
[1,]  1.0  2.0 -0.7
[2,]  3.0  0.2 -1.0
[3,]  0.9  7.0 -2.0
```

And then the vector of right hand sides:

In [4]:
```
y <- c(21,24,27)
y
```

21 · 24 · 27

In [5]:
```
x <- solve(A,y)
```

```
x
```

30 · 20 · 70

In [6]:
```
# checking the result, back-calculating y
print(A%*%x)
```

```
     [,1]
[1,]   21
[2,]   24
[3,]   27
```

Of the functions presented in the table, some explanation is required for `apply`:

apply(X, MARGIN, FUN,...)

If `X` is a matrix, it is split into vectors along rows (if `MARGIN` = 1) or columns (if `MARGIN` = 2). To each of these vectors is applied the function `FUN` with optional parameters `...` passed to it. For instance, we can calculate row sums of a matrix:

In [7]:
```
x <- matrix(1:12,3,4,byrow=TRUE)
print(x)
apply(x,1,sum)
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
```
10 · 26 · 42

1.7 Building simple databases: indexing/subsetting of vectors, arrays and data frames

In real life, one often needs to select some elements of a vector or a matrix, fulfilling certain criteria. This can be achieved using logical conditions or logical variables placed in **square brackets** after the object name. Subsets can be also used on the left hand side of the assignments when replacement of selected elements by certain values is desired.

1.7.1 Vectors

Subsets of a vector may be selected by appending to the name of the vector an index vector in square brackets. For example, we can first create a named vector:

In [8]:
```
x <- c(1,12,15,NA,16,13,0,NA,NA)
names(x) <- c("Pl","Bt","Mu","Q","Kfs","Ky","Ol","Px","C")
print(x)
```

```
 Pl  Bt  Mu   Q Kfs  Ky  Ol  Px   C 
  1  12  15  NA  16  13   0  NA  NA 
```

Index vectors can be of several types: logical, numeric (with positive or negative values), and character:

1. Logical vector

```
In [9]: #x[x>10]      # all elements of x higher than 10 (or NA)
        x[!is.na(x)] # all elements of x that are available
```

Pl: 1 **Bt:** 12 **Mu:** 15 **Kfs:** 16 **Ky:** 13 **Ol:** 0

2. Numeric vector with positive values

```
In [10]: x[1:5]      # the first five elements
         x[c(1,5,7)] # 1st, 5th and 7th elements
```

Pl: 1 **Bt:** 12 **Mu:** 15 **Q:** <NA> **Kfs:** 16

Pl: 1 **Kfs:** 16 **Ol:** 0

3. Numeric vector with negative values (specifies elements to be excluded)

```
In [11]: x
         x[-(1:5)] # all elements except for the first five
```

Pl: 1 **Bt:** 12 **Mu:** 15 **Q:** <NA> **Kfs:** 16 **Ky:** 13 **Ol:** 0 **Px:** <NA> **C:** <NA>

Ky: 13 **Ol:** 0 **Px:** <NA> **C:** <NA>

4. Character vector (referring to the element names)

```
In [12]: x[c("Q","Bt","Mu")]
```

Q: <NA> **Bt:** 12 **Mu:** 15

1.7.2 Matrices/data frames

Elements of a matrix are presented in the order `[row,column]`. If nothing is given for a `row` or `column`, it means no restriction. For instance:

```
x[1,]          # (all columns) of the first row
x[,c(1,3)]     # (all rows) of the first and third columns
x[1:3,-2]      # all columns (apart from the 2nd) of rows 1-3
```

If the result is a single row or column, it is automatically converted to a vector. To prevent such a behaviour, one can supply an optional parameter `drop=FALSE`, e.g.:

```
x[1,,drop=FALSE]   # (all columns) of the 1st row, keep as matrix
```

Moreover, matrices can be manipulated using index arrays. This concept is best explained on an example. Let's assume a matrix defined as:

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

If the elements `[1,3]`, `[2,2]` and `[3,1]` in `x` are to be replaced by zeroes, we can create an index array `i` containing the element coordinates:

```
In [13]: x <- matrix(1:20,4,5)
         print(x)
```

```
     [,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
```

In [14]:
```
print(x[1,])
print(x[1,,drop=FALSE])
```

```
[1]  1  5  9 13 17
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    9   13   17
```

In [15]:
```
print(x[,c(1,3)])
```

```
     [,1] [,2]
[1,]    1    9
[2,]    2   10
[3,]    3   11
[4,]    4   12
```

In [16]:
```
print(x[1:3,-2])
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    9   13   17
[2,]    2   10   14   18
[3,]    3   11   15   19
```

In [17]:
```
x <- matrix(1:20,4,5)
print(x)
i <- matrix(c(1,2,3,3,2,1),3,2)
print(i)
```

```
     [,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]
[1,]    1    3
[2,]    2    2
[3,]    3    1
```

In [18]:
```
x[i]
```

9 · 6 · 3

In [19]:
```
x[i] <- 0
print(x)
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    5    0   13   17
[2,]    2    0   10   14   18
[3,]    0    7   11   15   19
[4,]    4    8   12   16   20
```

The situation for multidimensional arrays is analogous — just the relevant number of dimensions is higher than two.

R involves numerous built in datasets that can be used to demonstrate its capabilities. The object `islands` is a vector with areas of landmasses (i.e. islands and continents) exceeding 10 000 sq. miles. Before we can start working, we need to attach the data object using the command:

In [20]:
```
data(islands)
```

To-do list:

- Display the whole vector. What is the area of Luzon?
- What is the average value of the whole vector?
- Which continent is the largest and what is its area?
- Which continents/islands have an area larger than 5000 sq. miles?
- Display the names of 15 smallest and largest continents/islands.
- Assuming that these are British miles, recalculate the data to km2 (1 sq mi = 2.59 km2)

In [21]:
```r
head(islands) # print just a few observations
islands["Luzon"]
```

Africa: 11506 **Antarctica:** 5500 **Asia:** 16988 **Australia:** 2968 **Axel Heiberg:** 16 **Baffin:** 184

Luzon: 42

In [22]:
```r
mean(islands)
```

1252.72916666667

In [23]:
```r
max(islands)
which(islands==max(islands))
```

16988
Asia: 3

In [24]:
```r
names(islands)[islands>5000]
```

'Africa' · 'Antarctica' · 'Asia' · 'North America' · 'South America'

In [25]:
```r
z <- sort(islands)
print(z[1:15])
```

```
       Vancouver            Hainan   Prince of Wales             Timor
              12                13                13                13
          Kyushu            Taiwan       New Britain       Spitsbergen
              14                14                15                15
    Axel Heiberg          Melville       Southampton Tierra del Fuego
              16                16                16                19
           Devon             Banks            Celon
              21                23                25
```

In [26]:
```r
out <- z[(length(z)-14):length(z)]
print(out)
```

```
      Britain        Honshu       Sumatra        Baffin    Madagascar
           84            89           183           184           227
       Borneo    New Guinea     Greenland     Australia        Europe
          280           306           840          2968          3745
   Antarctica South America North America        Africa          Asia
         5500          6795          9390         11506         16988
```

In [27]:
```r
# Or, alternatively:
out <- rev(sort(islands))[1:15]
print(out)
```

```
         Asia        Africa North America South America    Antarctica 
        16988         11506          9390          6795          5500 
       Europe     Australia     Greenland    New Guinea        Borneo 
         3745          2968           840           306           280 
   Madagascar        Baffin       Sumatra        Honshu       Britain 
          227           184           183            89            84 
```

In [28]:
```
print(islands*2.59) #  recalculate the data to km2 (1 sq mi = 2.59 km2)
```

```
           Africa        Antarctica              Asia         Australia 
         29800.54          14245.00          43998.92           7687.12 
     Axel Heiberg            Baffin             Banks            Borneo 
            41.44            476.56             59.57            725.20 
          Britain           Celebes             Celon              Cuba 
           217.56            189.07             64.75            111.37 
            Devon         Ellesmere            Europe         Greenland 
            54.39            212.38           9699.55           2175.60 
           Hainan        Hispaniola          Hokkaido            Honshu 
            33.67             77.70             77.70            230.51 
          Iceland           Ireland              Java            Kyushu 
           103.60             85.47            126.91             36.26 
            Luzon        Madagascar          Melville          Mindanao 
           108.78            587.93             41.44             93.24 
         Moluccas       New Britain        New Guinea  New Zealand (N) 
            75.11             38.85            792.54            113.96 
  New Zealand (S)      Newfoundland     North America    Novaya Zemlya 
           150.22            111.37          24320.10             82.88 
  Prince of Wales          Sakhalin     South America       Southampton 
            33.67             75.11          17599.05             41.44 
      Spitsbergen           Sumatra            Taiwan          Tasmania 
            38.85            473.97             36.26             67.34 
Tierra del Fuego             Timor         Vancouver          Victoria 
            49.21             33.67             31.08            212.38 
```

1.8 Lists

Lists are ordered collections of other objects, known as components, which do not have to be of the same mode or type. Thus lists can be viewed as very loose groupings of R objects, involving various types of vectors, data frames, arrays, functions and even other lists.

Components are numbered and hence can be addressed using their sequence number given in double square brackets, `x[[3]]`. Moreover, components may be named and referenced using an expression of the form `list_name$component_name`. Subsetting is similar to that of other objects, described above.

list.name <- list(component_name_1=, component_name_2=...)

Builds a list with the given components.

Here is a simple real-life example of a list definition:

In [29]:
```
x1 <- c("Luckovice","9 km E of Blatna","disused quarry")
x2 <- "melamonzonite"
x3 <- c(47.31,1.05,14.94,7.01,8.46,10.33)
names(x3) <- c("SiO2","TiO2","Al2O3","FeO","MgO","CaO")
luc <- list(ID="Gbl-4",Locality=x1,Rock=x2,major=x3)
print(luc)
```

```
$ID
[1] "Gbl-4"

$Locality
[1] "Luckovice"        "9 km E of Blatna" "disused quarry"

$Rock
[1] "melamonzonite"

$major
 SiO2  TiO2 Al2O3   FeO   MgO   CaO 
47.31  1.05 14.94  7.01  8.46 10.33 
```

As well as some examples of subsetting:

In [30]:
```
luc[[1]]
```

'Gbl-4'

In [31]:
```
luc$major # or luc[[3]]
```

SiO2: 47.31 **TiO2:** 1.05 **Al2O3:** 14.94 **FeO:** 7.01 **MgO:** 8.46 **CaO:** 10.33

In [32]:
```
luc[[2]][3]
```

'disused quarry'

In [33]:
```
luc$major[c("SiO2","Al2O3")]
```

SiO2: 47.31 **Al2O3:** 14.94

1.9 Coercion of individual object types

R is generally reasonably good at dealing seamlessly with data types, converting them on the fly when needed. When necessary, there are a series of functions for testing the mode (or type) of an object: `is.numeric(x)`, `is.character(x)`, `is.logical(x)`, `is.matrix(x)`, `is.data.frame(x)`

At times there is a need to explicitly convert between data types/modes using functions such as: `as.numeric(x)`, `as.character(x)`, `as.expression(x)`.

Less straightforward are: `as.matrix(x)`, `as.data.frame(x)` which attempt to convert an object `x` to a matrix or data frame, respectively.

A more user-friendly way of converting data frames to matrices is provided by the function `data.matrix` that converts all the variables in a data frame `x` to numeric mode and then binds them together as the columns of a matrix.

In [34]:
```
as.numeric("1.5444")*2
```

3.0888