Tutorial 2.1 Data objects and indexing

We can specify numbers and text in R. But our data is rarely represented as single values. R has many data types for large data sets.

Vectors

A natural extension of single numeric values is a sequence of of values. These are called *vectors* in R. To define a vector we use the function c(), which stands for *concatenate*.

Notice that the class of the vector is that of the items that compose it.

```
myvector1 <- c(1, 2, 3, 4)
myvector1
```

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

```
class(myvector1)
```

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

To define a sequence of number it is shorter to use ":"

```
mysequence1 <- 1:10
mysequence1
```

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

```
class(mysequence1)
```

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

In this case the class of the sequence is "integer", rather than numeric. For the purpose of this workshop this will not cause any problems, and we can consider these two clases to be equivalent.

Vectors can also contain text:

```
mycharacters1 <- c("one", "two", "three", "four")
mycharacters1
```

```
## [1] "one"   "two"   "three" "four"
```

```
class(mycharacters1)
```

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

For statistical modelling R uses categorical variables (such as characters, T or F, or binary response) as "factor" variables. Any variable can be converted to a factor with the as.factor() function:

```
AB.response <- c("A", "B", "A", "A", "B")
AB.response
```

```
## [1] "A" "B" "A" "A" "B"
```

```
class(AB.response)
```

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

```
myfactor <- as.factor(AB.response)
myfactor
```

```
## [1] A B A A B
## Levels: A B
```

```
class(myfactor)
```

```
## [1] "factor"
```

Notice that factor variables have levels, which are the unique values. These used for statistical models, and correspond to the categories. For instance, blood type A and B in the example above.

Matrices

A table can be represented in many ways in R. The most basic form is a matrix, which can be defined as follows:

```
mymatrix1 <- matrix(data = 1:9, nrow = 3, ncol = 3)
mymatrix1
```

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

```
class(mymatrix1)
```

```
## [1] "matrix"
```

In this case we have used the function matrix() to define the matrix. We have specified several *arguments* in the function. The first is the values in the matrix (data) , the second is the number of rows (nrow), and the third is the number of columns (ncol). Also notice that we specified some of these arguments with = (inside functions we use = instead of <-, which we use to specify variables). This is not always necessary. In some functions the arguments can be specified according to their default order.

In the following example mymatrix2 is identical to mymatrix1:

```
r mymatrix2 <- matrix(1:9, 3, 3) mymatrix2
```

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

```
r class(mymatrix2)
```

```
## [1] "matrix"
```

Individual exercise: create some matrices with different numbers. Try to specify the content of the matrix as a vector using the c() function. Next, create a matrix in which the contents is character data.

Arrays

A very flexible data type in R is arrays. This is essentially a matrix with any number of dimensions.

```
myarray1 <- array(data = 1:8, dim = c(2, 2, 2))
class(myarray1)
```

```
## [1] "array"
```

Before we take a look at myarray1 we will to describe the arguments of the function. We specified the data with the *data* argument, which are the numbers from 1 to 8. We specified the dimensions with the argument *dim*, for which we used a vector. Each item of the vector is a the number of items in a dimension. We can inspect myarray1 to understand this:

```
myarray1
```

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

myarray1 can be thought of as two 2x2 matrices binded together, which would result in a cube, so our data have three dimensions.

It is possible to specify much more complex arrays with any number of dimensions:

```
myarray2 <- array(data = 1, dim = c(2, 2, 2, 2))
class(myarray2)
```

```
## [1] "array"
```

```
myarray2
```

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

myarray2 is can be thought of as a hypercube, a cube with four dimensions. Note that we specified data = 1, so all the values in the array are the number 1.

One useful application of arrays is that each dimension is a matrix, depending in the dimensions specified:

```
myarray3 <- array(data = 1:4, dim = c(2, 2))
class(myarray3)
```

```
## [1] "matrix"
```

Data frames

Perhaps one of the most powerful data classes in R is the data frame. Note that in the classes that we have learned we have not used different data types within an object. In most cases R will coerce all the items to a single data type.

```
myvector2 <- c(1, 2, 3, 4, "character1", "character2")
myvector2
```

```
## [1] "1"          "2"          "3"          "4"          "character1"
## [6] "character2"
```

In this case the numbers were converted to characters, as we can see because they are enclosed in quotes. The result will be similar for matrices and arrays.

Individual exercise: create a vector, a matrix, and an array with characters and numbers and check whether the number are converted to class character

This is where data frames are most useful. Data frames are like matrices, but each column can have a different class. We will look at this in more detail in the next tutorial. For now, we will create a data frame and verify that the object is of class data.frame:

```
myvector3 <- 1:10
myvector4 <- c(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
mycharacters2 <- c("one", "two", "three", "four", "five", "six", "seven", "eight",
    "nine", "ten")

mydataframe1 <- data.frame(myvector3, myvector4, mycharacters2, stringsAsFactors = F)

class(mydataframe1)
```

```
## [1] "data.frame"
```

```
mydataframe1
```

```
##    myvector3 myvector4 mycharacters2
## 1          1         1           one
## 2          2         3           two
## 3          3         5         three
## 4          4         7          four
## 5          5         9          five
## 6          6        11           six
## 7          7        13         seven
## 8          8        15         eight
## 9          9        17          nine
## 10        10        19           ten
```

We have specified the argument stringsAsFactors = F. This is because the default of the function converts strings to factors, so we set it as FALSE. We will see factors in more detail in statistical modelling. For now, make sure to use the argument stringsAsFactos = F when you define data frames.

Notice that the syntax for the data frame is different from that of matrices. An other difference is that the columns have names. We can use this simplify data manipulation.

To find the names of the columns of data frames use the code bellow. Note that this can also be used for matrices when row or column names are specified.

```
colnames(mydataframe1)  # If we had specified row names, we would use the function rownames().
```

```
## [1] "myvector3"     "myvector4"     "mycharacters2"
```

The column names are useful because we can extract each column by using its name and the $ sign. The result is a an objet with the same class as the column.

```
column1 <- mydataframe1$myvector3
class(column1)
```

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

```
column1
```

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

Converting classes of objects

In some cases it is possible to change the class of an object, like we did for characters and factors in a previous example. This may be necessary for some packages and some functions. In particular, data frames and matrices can be converted with the functions as.data.frame() and as.matrix():

```
as.matrix(mydataframe1)
```

```
##       myvector3 myvector4 mycharacters2
##  [1,] " 1"      " 1"      "one"
##  [2,] " 2"      " 3"      "two"
##  [3,] " 3"      " 5"      "three"
##  [4,] " 4"      " 7"      "four"
##  [5,] " 5"      " 9"      "five"
##  [6,] " 6"      "11"      "six"
##  [7,] " 7"      "13"      "seven"
##  [8,] " 8"      "15"      "eight"
##  [9,] " 9"      "17"      "nine"
## [10,] "10"      "19"      "ten"
```

```
as.data.frame(mymatrix1)
```

```
##   V1 V2 V3
## 1  1  4  7
## 2  2  5  8
## 3  3  6  9
```

Note that data frames need names for the columns. When these are not provided, as in the case above, the function as.matrix assigns the default of V1, V2, etc...

Individual exercise: Save the objects above and check their class to ensure that the object type conversion works.

There are other similar functions, such as as.numeric, as.character, and others.

Tutorial 2.2 Indexing

Indexing is crucial for any data analysis and for manipulating data. The index refers to the position of an item in an object, such as the first item in a vector or a matrix. Slicing is the extraction of sections of data. We will start with a vector.

We use the square brackets as follows:

```
myvector <- 1:10
myvector
```

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

```
# To get the first item we use:
myvector[1]
```

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

```
# To get the fifth item:
myvector[5]
```

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

We can also *slice* the vector to get a section of items. This can be obtained by specifying a vector inside []

```
myvector[1:3]
```

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

```
myvector[c(1, 4)]
```

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

In the case of a matrix, we need to specify the index in both dimensions. The first is the row item and the second is the column:

```
mymatrix <- matrix(1:9, 3, 3)
mymatrix
```

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

```
# To get the first item use:
mymatrix[1, 1]
```

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

```
# To get the first item in the second row:
mymatrix[2, 1]
```

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

```
# To get the entire first row leave the space for the column blank, but do
# not forget to use the ','
mymatrix[1, ]
```

```
## [1] 1 4 7
```

```
# The same applies for the any of the columns. In this case we will extract
# the second column
mymatrix[, 2]
```

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

An other useful option is to remove one section of a vector of a matrix, using the - sign:

```
# To remove the first row of the matrix:
mymatrix[-1, ]
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
## [2,]    3    6    9
```

```
# To remove item 3 from the vector:

myvector[-3]
```

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

Individual exercise: create a matrix and create two vectors which are rows one and two

In data frames, the syntax for indexing is the same as it is for matrices

```
mydataframe1[4, 2]
```

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

However, note that data frames have names for the columns. These can be used with the $ symbol to obtain the values for the respective column. We can extract the names only with the function colnames() or names()

```
colnames(mydataframe1)
```

```
## [1] "myvector3"     "myvector4"     "mycharacters2"
```

```
names(mydataframe1)  # the result is the same as above
```

```
## [1] "myvector3"     "myvector4"     "mycharacters2"
```

```
# To get the column myvector3:

mydataframe1$myvector3
```

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

```
# To get the column mycharacters2

mydataframe1$mycharacters2
```

```
##  [1] "one"   "two"   "three" "four"  "five"  "six"   "seven" "eight"
##  [9] "nine"  "ten"
```

Individual exercise: Create a data frame, extract one of the columns and items 1 to 3, and store them in a vector

Tutorial 2.3 Sorting

It is ofter useful to sort data. This can be done for vectors as follows:

```
# We create two objects with unsorted number and letters
myvector5 <- c(2, 3, 1, 4, 5, 10)
mycharacters3 <- c("c", "a", "b", "d")
```

The function sort() sorts the vector from the smallest, to the largest value. In the case of characters, it uses the alphabetical order.

```
sort(myvector5)
```

```
## [1]  1  2  3  4  5 10
```

```
sort(mycharacters3)
```

```
## [1] "a" "b" "c" "d"
```

The function order() returns the order of the items in the object. Note that it does not return the actual values, unlike sort()

```
order(myvector5)
```

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

```
order(mycharacters3)
```

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

These two functions have a parameter called *decreasing*. The default is decreasing=T, and it can be set to decreasing=F to sort or order the items from largest to smallest (or from the last to the first letters in the alphabet)