More on R objects

Antonio Gasparrini

01 November 2023

Contents

This session offers an extended illustration of objects, focusing in particular on their data structures. Specifically, it is shown how more complex structures derived from vectors form the elements you are expected to manage during an analysis. These objects are factors, matrices and arrays, lists, and data frames, which are presented in this order together with information on functions to create and manipulate these objects. A final section provides a more specific definition of objects based on their *attributes*, and illustrates how objects can be transformed into different types through *coercion*.

Factors

A *factor* is a vector with elements referring to specific categories, commonly used for the discrete classification of categorical data. Factors are created by applying the function `factor()` to a vector (usually a character vector). For example:

```
sex <- c("Male","Female","Female","Male","Male")
fsex <- factor(sex)
```

The function returns a new transformed object. When the original and transformed objects are printed:

```
sex
```

```
## [1] "Male"   "Female" "Female" "Male"   "Male"
```

```
fsex
```

```
## [1] Male   Female Female Male   Male
## Levels: Female Male
```

The new object resembles the original character vector (although elements are printed as unquoted strings), but with some additional information about the *levels* (*i.e.* categories) of the factor. Actually, the mode of a

factor is `numeric`, as shown by:

```
mode(fsex)
```

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

This suggests that the original character vector `sex` is converted to numbers with levels attached as labels. Levels are assigned automatically as the unique categories in alphabetical order, stored in the object and printed with it. The function `levels()` can be used to access the levels:

```
levels(fsex)
```

```
## [1] "Female" "Male"
```

The order of the levels can be modified by the function `relevel()`. For instance:

```
fsex2 <- relevel(fsex, ref="Male")
fsex2
```

```
## [1] Male   Female Female Male   Male
## Levels: Male Female
```

The factor object `fsex2` is identical to `fsex`, but the first level (called the *reference*) is now `"Male"`. The choice of the reference level is not critical here but can be important when comparing categories, for example in regression models. Alternatively, the order of the levels can be determined through the argument `levels` in the function `factor()` when creating the object.

Sometimes categorical data are originally provided as data coded with numbers, for example, 2=Yellow, 5=Red and 8=Green. In this case, the argument `labels` can be used:

```
(fcol <- factor(c(8,2,5,5,2,8,2), labels=c("Yellow","Red","Green")))
```

```
## [1] Green  Yellow Red    Red    Yellow Green  Yellow
## Levels: Yellow Red Green
```

where the levels are assigned using labels, this time associated with numbers in increasing order.

A factor whose levels can be regarded as having an increasing or decreasing order is called *ordered factor*. These types of factors are created by setting to `TRUE` the argument `ordered` in `factors()` (with default to `FALSE`). Let's see an example:

```
factor(c(2,3,1,3,2), labels=c("low","medium","high"), ordered=TRUE)
```

```
## [1] medium high   low    high   medium
## Levels: low < medium < high
```

As shown above, in ordered factors the levels are printed showing their ordered nature. Alternatively, the function `ordered()` can be used to directly create ordered factors.

Matrices and arrays

Another type of object commonly used in R is the *matrix*, which plays a major role in statistical computations. A matrix can be defined as a rectangular array of elements of the same mode. Similarly to factors, a matrix can be created from a vector, using the function `matrix()`. For example:

```
mymat1 <- matrix(1:6, nrow=2, ncol=3)
mymat1
```

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

The arguments `nrow` and `ncol` specify the dimensions of the matrix as the number of rows and columns. Actually, one of them can be omitted here, as the other dimension is indirectly defined by the length of the vector. For example:

```
(mymat2 <- matrix(c(2,0,3,1), nrow=2))
```

```
##      [,1] [,2]
## [1,]    2    3
## [2,]    0    1
```

When a matrix is printed, in contrast to vectors, the numbers between square brackets represent the indices for rows and columns.

The elements in the vector are reordered in the matrix by column, *i.e.* by filling the first column, then the second one, and so on. Setting the argument `byrow=TRUE` causes the matrix to be filled by rows instead:

```
# if by rows:
matrix(1:6, nrow=2, ncol=3, byrow=TRUE)
```

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

The information about matrix dimensions is returned by the functions `nrow()`, `ncol()`, and `dim()`:

```
nrow(mymat1)
```

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

```
ncol(mymat1)
```

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

```
dim(mymat1)
```

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

Technically, a matrix is simply a vector with additional information about the dimensions. For example, it keeps the information about the length of the original vector:

```
length(mymat1)
```

```
## [1] 6
```

For this reason, matrices are very often acting or treated like vectors in different types of computations.

The manageable and uncomplicated computational facilities for matrices are one of the advantages of using R. For example, matrix objects can be bound by columns, by:

```
cbind(mymat1, mymat2)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    2    3
## [2,]    2    4    6    0    1
```

with the requirement that the two (or multiple) matrices have the same number of rows. The function `rbind()` does the corresponding operation by rows. Simple vectors can be bound in the same way.

Common arithmetic, relational or logical operators similarly work with matrices:

```
mymat3 <- matrix(4:1, 2)
mymat2 + mymat3
```

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

```
mymat2 > mymat3
```

```
##       [,1]  [,2]
## [1,] FALSE  TRUE
## [2,] FALSE FALSE
```

Again, dimensions are expected to be consistent.

The operator '$*$' performs element by element multiplication. Matrix multiplication is instead expressed through the operator '%*%'. An example:

```
mymat2 %*% mymat1
```

```
##      [,1] [,2] [,3]
## [1,]    8   18   28
## [2,]    2    4    6
```

As usual, dimensions need to be consistent. Other algebraic operations involving matrices are carried out by specific functions, such as `solve()` (inversion), `t()` (transposition), `chol()` (Cholesky decomposition). See the related help pages for some examples. The function `diag()` returns different values depending on the first argument:

```
diag(mymat2)
```

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

```
diag(1:3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    2    0
## [3,]    0    0    3
```

```
diag(2)
```

```
##      [,1] [,2]
## [1,]    1    0
## [2,]    0    1
```

Specifically, it extracts the diagonal from a square matrix or creates a diagonal matrix from a vector. With the third expression where the first argument is a numeric scalar (a vector of length 1), it constructs an identity matrix with dimension given by the scalar.

Other functions available for numeric matrices are `colSums()` and `colMeans()`, which return a vector with the sum or mean of a matrix, respectively. For instance:

```
colSums(mymat1)
```

```
## [1]  3  7 11
```

```
colMeans(mymat1)
```

```
## [1] 1.5 3.5 5.5
```

The functions `rowSums()` and `rowMeans()` apply the same computations to rows.

Vectors and matrices can be generalized to array objects, created by the function `array()`. An example:

```
array(1:8, dim=c(2,2,2))
```

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

where the argument `dim` specifies the dimensions. In this case, the object is printed as matrices in different *layers*, defining the third dimension. Arrays generalize the concept of vector and matrix, being defined as sequences of elements structured in 1 or more dimensions.

Lists

The objects illustrated so far (vectors, factors, matrices/arrays) are composed of elements with the same mode. A more flexible object in R is the *list*, basically a collection of different objects, optionally with different modes. Lists are created by the function with the same name:

```
mylist1 <- list(1:5, fsex, mymat2)
mylist1
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
## [1] Male   Female Female Male   Male
## Levels: Female Male
##
## [[3]]
##      [,1] [,2]
## [1,]    2    3
## [2,]    0    1
```

where elements of the list, called *components*, are included as an arbitrary number of arguments of `list()`. When printed, the numbers in the double square brackets '`[[]]`' identify the index of each component in the list (which can be any object), while the single square brackets '`[]`' refer to the elements of the component itself as seen for vectors or matrices. Lists resemble vectors in that they are sequences of elements, although these elements can be more structured objects of different types. Similarly to vectors, the function `c()` can be used to concatenate lists, in this case appended component by component. Let's have a look at the properties of this object:

```
mode(mylist1)
```

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

```
length(mylist1)
```

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

Lists can include other lists, allowing the user to construct arbitrary data structures. Their mode is `list`,

highlighting this peculiarity. Given their flexibility, they are often used as the output of complex functions, storing information in objects of various types, such as regression functions.

Components in the list can be named, by giving *tags* to arguments of `list()` through the operator '=':

```
mylist2 <- list(numbers=1:3, names=c("Roger","Nick","Richard","Syd","David"))
mylist2
```

```
## $numbers
## [1] 1 2 3
##
## $names
## [1] "Roger"   "Nick"    "Richard" "Syd"     "David"
```

Single objects representing components within a named list can then be retrieved by the dollar-sign operator '$':

```
mylist2$names
```

```
## [1] "Roger"   "Nick"    "Richard" "Syd"     "David"
```

The expression above can be translated as 'the component `names` of the list `mylist2`'.

Data frames

A *data frame* is the R object which is more similar to the rectangular array traditionally used to store data in statistical programs, with observations for different records (the rows) collected for different variables (the columns). A data frame is created by the function `data.frame()`, with usage similar to `list()`. An example:

```
bands <- data.frame(name=c("The Cure","dEUS","Pearl Jam","Pink Floyd"),
  year=c(1976,1991,1990,1965), country=c("UK","Belgium","USA","UK"))
bands
```

```
##         name year country
## 1   The Cure 1976      UK
## 2       dEUS 1991 Belgium
## 3  Pearl Jam 1990     USA
## 4 Pink Floyd 1965      UK
```

In this case, the arguments of `data.frame()` specify vectors of equal length which are interpreted as variables of the data frame. The variables can also be existing vectors. The arguments can be tagged, as in the example above, or otherwise, the variables will be given the names of the objects, if any, or default names.

Data frames share properties with both matrices and lists, although their structure is more similar to the latter. Similarly to matrices, they are printed as rectangular arrays with rows and columns. However, the data in different columns are allowed to be of different types. In addition, their mode and length are those of a list:

```
mode(bands)
```

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

```
length(bands)
```

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

Similarly to lists, they can be seen as collections of objects (the single variables in columns), although all these objects must be consistent with the number of rows. The objects included in a data frame are usually vectors or factors of the same length (as in the example above), but similarly to lists, they can also be

matrices with the same number of rows. Also, these objects, interpretable as components of a list, can be retrieved with the dollar operator '$':

```
bands$country
```

```
## [1] "UK"      "Belgium" "USA"     "UK"
```

Given their similarity to matrices, functions seen earlier are applicable with data frames as well:

```
nrow(bands)
```

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

```
ncol(bands)
```

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

Functions `cbind()` and `rbind()` also work with data frames, although the former requires the two objects having the same variable names. If all the variables are numeric, `colSums()` and similar functions can also be used.

Attributes and object types

As mentioned earlier, the R objects described in this chapter are created by assembling simpler objects such as vectors. The information about this additional structure and complexity can be stored in *attributes*, which help to define the object itself. The *intrinsic* attributes, common to vectors and all other objects, are mode and length. Other intrinsic attributes may be defined for some objects. Some attributes are required in the definition of specific objects, while others are just optional.

Non-intrinsic attributes are returned by the function `attributes()`. Let's see the attributes of `mymat2`:

```
# attributes (intrinsic and non-intrinsic)
attributes(mymat2)
```

```
## $dim
## [1] 2 2
```

The function returns a list with all the attributes. In this case, a list of length 1 with a single component `dim`, storing the information about the dimensions of the matrix. Such attribute can be accessed with `attributes(mymat2)\$dim`, given it is in a list, or by the function `attr()`, which accepts as arguments the object and the name of the attribute:

```
attr(mymat2, "dim")
```

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

Another attribute can be the vector with names of the elements or components. Let's check with the list object `mylist2`:

```
attributes(mylist2)
```

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

In R, there are functions with prefix `is` to test if an object is of a given type, for instance `is.factor()`, `is.matrix()`, `is.list()` and `is.data.frame()`. Similary, there are related functions with prefix `as` to transform an object in another type, a process called *coercion*. For example, the command:

```
as.data.frame(mymat2)
```

```
##   V1 V2
```

```
## 1  2  3
## 2  0  1
```

coerces the matrix `mymat2` to a data frame, as proved by the default names given to columns. Let's see another example with factors, which can naturally be coerced to numeric or character vectors:

```
as.character(fsex)
```

```
## [1] "Male"   "Female" "Female" "Male"   "Male"
```

```
as.numeric(fcol)
```

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

The coercion to a numeric vector is straightforward, given the mode of a factor is numeric, as shown above. However, interestingly, you can appreciate how the factor `fcol` is now composed of a different set of numbers (by default integers starting from 1) than the original numeric vector from which it has been created.