Labelling and indexing in R

Antonio Gasparrini

01 November 2023

Contents

This session describes methods in R for labelling and indexing objects. The former provides a way to assign names to elements of the object, a process which simplifies several computations. Indexing is the main methods used in R for selecting parts of objects, offering a simple and intuitive way to manipulate them. Indexing, in particular, is a key feature in the R language and software, and plays a fundamental role in many computations. However, the method is based on simple and general rules which apply to all the types of objects in R. The concepts and methods illustrated in this session are commonly applied to perform more advanced computational tasks in R. The first section introduces labelling, while the rest of the sections deals with indexing methods for different types of objects, from vectors to data frames.

Labelling

The different objects created in R may include labels for their elements, usually called *names*. The elements can be named directly when creating the object by assigning *tags* through the operator '='. We start with the simplest object in R, the vector, where tags can be assigned to each argument of the function `c()`:

```
yearfirstlp <- c(Muse=1999, "Neil Young"=1968, "Stone Roses"=1989)
yearfirstlp
```

```
##        Muse  Neil Young Stone Roses
##        1999        1968        1989
```

This process directly assigns names to the elements. As you can see above, names in tags with spaces must be quoted. Names are stored as an attribute of the object, and the function `names()` can be used to extract them:

```
attributes(yearfirstlp)
```

```
## $names
## [1] "Muse"        "Neil Young"  "Stone Roses"
```

```
names(yearfirstlp)
```

```
## [1] "Muse"        "Neil Young"  "Stone Roses"
```

The same labelling process applies to more complex objects, although we need to account for their different data structure. For instance, matrix objects have two dimensions, and names can be assigned both to rows and columns using the argument `dimnames`, which accepts a list of two vectors with corresponding names. Here we create a matrix `mymat` and assign names to both rows and columns:

```
mymat <- matrix(1:6, nrow=2, ncol=3, dimnames=list(c("r1","r2"), c("c1","c2","c3")))
mymat
```

```
##    c1 c2 c3
## r1  1  3  5
## r2  2  4  6
```

The corresponding function `dimnames()`, or alternative the more specific functions `rownames()` and `colnames()`, can be used to extract the names from an existing matrix:

```
rownames(mymat)
```

```
## [1] "r1" "r2"
```

```
colnames(mymat)
```

```
## [1] "c1" "c2" "c3"
```

The process of tagging also works for list objects, which can be seen as extensions of vectors where elements can be objects themselves different from scalar elements. For example, we can create an object `mylist` and name its components:

```
mylist <- list(vector=1:5, matrix=mymat)
mylist
```

```
## $vector
## [1] 1 2 3 4 5
##
## $matrix
##    c1 c2 c3
## r1  1  3  5
## r2  2  4  6
```

Again, similarly to vectors, names can be extracted (or replaced) using the function `names()`:

```
names(mylist)
```

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

Data frames share properties with both matrices and lists, and therefore we can label them using either names for rows and columns or directly by tagging. Specifically, `rownames()` and `colnames()` refer to the names of the records and variables, while `names()` returns/assigns the names of the variables alone, as they are considered elements of a list. As an example, we can first create a data frame `bands` using tags for naming the variables:

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

and then extract the names corresponding to rows or columns/variables:

```
rownames(bands)
```

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

```
colnames(bands)
```

```
## [1] "name"    "year"    "country"
```

```
names(bands)
```

```
## [1] "name"    "year"    "country"
```

As shown above, in data frames the row names are set by default to row numbers if not specified.

Indexing vectors

A very common task in data manipulation is to select data. In R, this task is performed through *indexing*. The idea behind indexing is to select data in an object referring to the sequence or names of its elements. The elements are selected through the use of square brackets '`[]`' in so-called *extractor expressions*. These brackets actually represent a proper operator acting on different objects (see `?'['` or `?Extract`).

Here we first address the issue with simple unidimensional vectors, and then we extend the computation to more complex objects in the next sections. The selection of elements of a vector is performed by appending to its name an *index vector* between square brackets. There are four types of index vectors.

The first type is composed of a sequence of positive numbers, corresponding to the indices of the elements to be extracted. For instance, we create and then index a vector with this first type:

```
numvec <- c(4,2,-7,16,-2)
numvec
```

```
## [1]  4  2 -7 16 -2
```

```
numvec[3]
```

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

```
numvec[3:5]
```

```
## [1] -7 16 -2
```

The last two expressions above tell R to extract the third element of the vector, and then the elements from the third to the fifth. In this method of indexing, the value returned is a vector of length equal to the index vector. The same element can be extracted more than once:

```
numvec[rep(3:4, each=3)]
```

```
## [1] -7 -7 -7 16 16 16
```

The second type of index vectors is specified as a sequence of negative numbers, corresponding to the indices of the elements *not* to be extracted. For example, the following expression:

```
numvec[-3]
```

```
## [1]  4  2 16 -2
```

selects all the elements of `numvec` but the third one. A vector of negative numbers can be used to select multiple elements to be excluded from extraction, such as:

```
numvec[-c(1:3)]
```

```
## [1] 16 -2
```

In this case, the first three elements are not extracted. Be careful about using the `c()` function after the minus sign, otherwise the expression `-1:3` will produce a mixture of negative and positive integers which are not accepted by the extractor operator (and an error is returned).

The third type of index vector is a logical vector with the same length as the vector whose elements need to be extracted. In this case, only the elements with indices corresponding to `TRUE` will be selected. The expression:

```
numvec[c(T,T,F,T,F)]
```

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

extracts only three elements of the vector `numvec`. The method above can be used with logical operators to select elements of a vector given their values:

```
numvec[numvec>3]
```

```
## [1]  4 16
```

In the example above, only the elements of `numvec` higher than 3 are selected. Here, R first creates a logical vector and then applies it to select the elements. Selection based on the values of the elements also applies to character vectors or factors.

The fourth and last type of index vector is composed of character strings, corresponding to the names of the elements to be selected. We provide an example using the vector `yearfirstlp` created above and applying the selection:

```
yearfirstlp[c("Muse","Stone Roses")]
```

```
##        Muse Stone Roses
##        1999        1989
```

In this last option, the selection can only be applied to vectors with named elements. The advantage is that often the names of the elements are easier to remember and identify if compared to their indices. This method is very useful when indexing more complex data structures such as lists or data frames, as shown later.

Indexing matrices or arrays

The indexing methods illustrated above can be extended to matrices. The simpler method is to define two index vectors corresponding to the dimensions of rows and columns, respectively. All the types of index vectors illustrated in the previous section may be applied. Such index vectors are usually specified in square brackets separated by commas. For example, the command:

```
mymat[2, c(1,3)]
```

```
## c1 c3
##  2  6
```

selects a vector with elements corresponding to the second row, and the first and third columns of `mymat`. Leaving empty one of the two positions within the square brackets causes all the elements for that dimension to be selected. For instance:

```
mymat[, -3]
```

```
##    c1 c2
## r1  1  3
## r2  2  4
```

extracts a matrix with all the rows and all but the third columns of `mymat`. In the first example above, the result is transformed in a vector if a single index is specified in one of the dimensions, *i.e.* the returned value is coerced to the lowest possible dimension. In contrast, the object returned in the second example is still a matrix. The argument `drop` prevents this action:

```
mymat[2, c(1,3), drop=FALSE]
```

```
##    c1 c3
## r2  2  6
```

In this case, the returned object is a matrix with just one row. Row and/or column names are kept, when possible.

Logical vectors, or character vectors corresponding to row or column names (if any), can also be specified. An example:

```
mymat[c(T,F), c("c1","c3")]
```

```
## c1 c3
##  1  5
```

The expression above extracts the elements of the first row and first and third columns.

Other methods are available, although more rarely applied. For instance, we can use a single index vector compatible with the length of the matrix (equal to the product of the two dimensions):

```
mymat[c(T,F,F,T,F,F)]
```

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

Alternatively, we can define an *index matrix*. These can be of two types. First, a logical matrix with the same dimensions where `TRUE`/`FALSE` correspond to elements (not) to be kept:

```
(indexmat1 <- matrix(c(T,T,F,F,F,T), nrow=2))
```

```
##      [,1]  [,2]  [,3]
## [1,] TRUE FALSE FALSE
## [2,] TRUE FALSE  TRUE
```

```
mymat[indexmat1]
```

```
## [1] 1 2 6
```

Finally, we can define a two-column index matrix where each row specifies the indices of each element to be selected:

```
(indexmat2 <- matrix(c(1,2,1,3), nrow=2, byrow=TRUE))
```

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

```
mymat[indexmat2]
```

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

returns a vector with elements corresponding to the first row and second column, and the first row and third column.

The indexing of arrays follows the same lines, and multiple indexing vectors separated by commas can be specified between the square brackets. The argument `drop` can again be used to prevent the coercion to the lowest possible dimension, such as matrices or vectors.

Indexing lists

Indexing is more sophisticated with lists, as it involves the use of multiple operators and a more complex syntax. The simplest approach is to select the components using the same methods as in vectors, *i.e.* by specifying an index vector. To illustrate the process, we first create a list with two components, and then define the indexing expression:

```
mylist <- list(numbers=1:3, names=c("Roger","Nick","Richard","Syd","David"))
mylist[2]
```

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

This reduces the list to its second component only. All four types of indexing vectors can be used, including a vector of negative values, a logical vector, or a character vector of names (for named lists) to select one or multiple components (even repeatedly).

You can notice, however, that the syntax above returns an object which is again a list. The length of such a list is equal to that of the index vector, in the example above a list of length 1. Instead, if we want to directly extract the object corresponding to a specific component of the list, we need to use a different operator defined by double square brackets '`[[]]`':

```
mylist[[2]]
```

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

In this case, the object returned is the vector originally included as the second component in `mylist`. You can notice that the same result can be achieved in named lists by using the dollar operator '`$`':

```
mylist$names
```

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

Multiple indexing can be performed as well referring separately to nested levels of lists:

```
mylist[[2]][3:4]
```

```
## [1] "Richard" "Syd"
```

Here, the first index refers to a specific component of the list, while the second index vector specifies the indices of the elements. The command above extracts the third and fourth elements of the second components of `mylist`.

Indexing data frames

Selecting data within datasets is an important aspect of statistical analysis. Although in R the computation may involve multiple objects, the focus of the analysis of real data is often on a single dataset stored in a data frame. Similarly to labelling, indexing methods for both matrices and lists can be used for data frames.

The list-wise indexing is commonly applied to extract a selection of variables from the dataset. Using again the object `bands` created above, the expression:

```
bands[c(1,3)]
```

```
##          name country
## 1   The Cure      UK
## 2       dEUS Belgium
## 3  Pearl Jam     USA
## 4 Pink Floyd      UK
```

returns the dataset with only the first and third variable, taken as first and third components of a list. Similarly:

```
bands[1]
```

```
##          name
## 1   The Cure
## 2       dEUS
## 3  Pearl Jam
## 4 Pink Floyd
```

returns a data frame with just one variable, corresponding to the first component/variable `name`. The same results can also be obtained with `bands["name"]` using a character index vector. You may note how the object is still a data frame, although with just one variable. Instead:

```
bands[[1]]
```

```
## [1] "The Cure"   "dEUS"       "Pearl Jam"  "Pink Floyd"
```

selects the same variable, but returns the object itself, in this case the character vector `name`. Exactly the same results can also be obtained using the dollar operator in `bands$name`. In all the examples above, `bands` is treated as a list. The last selection can also be obtained using matrix-wise indexing, with the expression `bands[,1]`.

The same matrix-wise indexing can be used to select observations corresponding to specific rows, obtaining similar results as with the function `head()`. For example, the first two rows can be extracted by:

```
bands[1:2,]
```

```
##       name year country
## 1 The Cure 1976      UK
## 2     dEUS 1991 Belgium
```

Indexing can also apply simultaneously to two dimensions. The two following expressions select by both rows and columns:

```
bands[c(1,3), 2]
```

```
## [1] 1976 1990
```

```
bands[1, 1:2]
```

```
##       name year
## 1 The Cure 1976
```

You may note that similarly to matrices, the first command selects only one variable and the returned value is coerced to a vector. However, when only a row is selected in the second example, the row dimension is not dropped and the returned object is still a data frame, although with a single observation.

Similarly, index vectors of negative numbers, logical values, or row and column names can be used for selecting both observations and variables. The most common approach, however, is to use a logical vector to select observations, based on the value of one or more variables, and a character vector of column names to select variables. An example will clarify the issue. Let's index the dataset `bands` with this expression:

```
bands[bands$year>1970 & bands$country=="UK", c("name","year")]
```

```
##       name year
## 1 The Cure 1976
```

This code extracts the names and year of formation (selection by column) of UK bands formed after 1970 (selection by row). Although the same results could have been obtained by the use of different index vectors, this approach reproduces closely the logical statement used for selecting the observations, and furthermore lists the name of the variables, providing expressions which are easier to interpret and less prone to errors.

As shown in the examples in this section, selecting subsets of data sometimes involves the use of complex expressions, that can be nonetheless simplified to some simple general rules. In particular, the statements used for selection need to be translated into one or more indexing vectors. Different expressions may return exactly the same result, and the choice often depends on the selection criteria, the type of data and your personal preference. This offers a general and flexible computing alternative to specific subsetting functions available in different packages.