The basics of R objects

Antonio Gasparrini

01 November 2023

Contents

This session elaborates on the notion of *objects*. A motto from the developers of S, the ancestor of R, is that in these systems *everything is an object*, highlighting that everything in R, from data to functions and expressions, can be managed as an object. Most of R objects can be seen as collections of these basic elements in different data structures. In this session, the simplest form, the *vector*, is presented. More complex data structures are composed of vectors as basic blocks. The first section provides a definition for the different objects available in R, and how objects can be transformed into each other. The next sections introduce vectors, how they can be created, and how they can be used in computations. The last section briefly summarizes special objects such as constants.

Data in objects

In R, an object can be composed by data of different sorts, depending on the internal structure of its *atomic elements*. Historically, the distinction was based on *modes* and *types*, concepts borrowed from the S language. The main modes are `numeric`, `character`, and `logical`. Other categories, such as `list` and `function` will be described later. Few others, such as `complex`, `raw`, or `symbol`, are not of interest here.

A numeric object is composed of atomic elements corresponding to real numbers, while a character object contains strings (single or double-quoted text). A logical object is made of atomic elements `TRUE` and/or `FALSE`, with shortcuts `T` or `F`, respectively. The mode of an object is displayed by:

```
mode(2)
```

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

```
mode("Robert Smith")
```

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

```
mode(FALSE)
```

```
## [1] "logical"
```

A more subtle classification is based on the *type* of data, depending on the *storage mode*. For example, a numeric vector can be stored as `double` or `integer`. The functions `typeof()` and `storage.mode()` can be used in this case (see the related help pages).

```
typeof(2)
```

```
## [1] "double"
```

Specific functions with '`is`' prefix and a suffix matching the mode or type can be used to check objects. For example:

```
is.numeric(2)
```

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

```
is.integer(5.1)
```

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

```
is.logical("Iggy Pop")
```

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

Objects of different modes can be transformed into each other through *coercion*. Specifically, logical elements `TRUE` and `FALSE` are interpreted as 1 and 0 when coerced to numeric. Conversely, any number but 0 is interpreted as `TRUE` when coerced to logical. Both numeric and logical objects can be coerced into characters, but the reverse is not always possible. Coercion is performed with specific functions with '`as`' prefix and given suffix, such as:

```
as.numeric(TRUE) ; as.numeric(FALSE)
```

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

```
## [1] 0
```

```
as.logical(-3) ; as.logical(0)
```

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

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

```
as.character(3)
```

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

In the result from the last expression above, note how the number `3` is now quoted, *i.e.* interpreted as a character value. Coercion is applied automatically by some functions and operators:

```
3 + TRUE
```

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

When the coercion is not straightforward, warnings or errors can be returned. Concepts and functions applied above to simple atomic elements can be extended to more sophisticated data structures. In the rest of this chapter, you will be introduced to vectors, while matrices/arrays, lists and data frames are presented later.

Vectors

A *vector* is a sequence of atomic elements of the same mode. It represents the basic object in R, as scalars, such as those created earlier, are taken as single-element vectors (vectors of length 1). Vectors constitute the building blocks in the creation of more complex objects, and generally in computing with R. The simplest way to create a vector is to *concatenate* elements together with the function `c()`. An example of a numeric vector:

```
(numvec1 <- c(4,2,-7,16,-2))
```

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

The elements are defined as an arbitrary number of arguments of the function `c()`. They are separated by a comma when inputted (as in any other function), and by spaces when printed, as shown above. Here is an example with a longer vector:

```
rnorm(10)
```

```
##  [1] -1.29986434 -0.81854568 -0.07836209  0.29786709 -2.93285700  1.17069395  0.60294784  2.52788634
```

The numbers between square brackets indicate the position index in the vector of the first element in each line (indexing will be discussed in greater details later). The elements concatenated by `c()` can also be vectors themselves:

```
numvec2 <- c(3,0,0,-2,14)
(numvec3 <- c(numvec1,numvec2))
```

```
##  [1]  4  2 -7 16 -2  3  0  0 -2 14
```

An example of character vector:

```
(pj <- c("Eddie","Stone","Jeff","Mike","Matt"))
```

```
## [1] "Eddie" "Stone" "Jeff"  "Mike"  "Matt"
```

And now example of logical vector:

```
(logvec <- c(T,T,F,T,F))
```

```
## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

Logical vectors are also created by *logical expressions* that are true or false. For example, using the numeric vectors created above:

```
numvec2 == -2
```

```
## [1] FALSE FALSE FALSE  TRUE FALSE
```

```
pj != "Stone"
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

```
numvec1 > 3
```

```
## [1]  TRUE FALSE FALSE  TRUE FALSE
```

```
! numvec1 >= 2
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE
```

In these expressions defining conditions, note the use of the *relational operators* such as '==' (equality), '`!=`' (inequality), '>' (higher than), '>=' (higher than or equal to), as well as *logical operators* such as '`!`' (logical negation). See the help pages using `help('==')`. These operators are very useful for selecting data through indexing, as discussed in other sessions.

A vector is primarily defined by its mode, as seen earlier, and its *length*. The length is returned by:

```
length(numvec1)
```

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

You can easily check which objects are present in the global environment, and which mode and length they have, by looking at the Environment window in RStudio.

Regular sequences

Vectors can also be produced as *regular sequences*. The easiest way is through the colon operator ':', which is often applied to produce sequences of integers. See the result of the expression:

```
3:10
```

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

while `10:3` produces the same vector with a reverse order. The operator above is a shortcut for a specific application of the more flexible function `seq()`, which creates different sequences of numbers depending on its arguments. Some example:

```
seq(from=1, to=2, by=0.2)
```

```
## [1] 1.0 1.2 1.4 1.6 1.8 2.0
```

```
seq(from=3, to=6, length.out=5)
```

```
## [1] 3.00 3.75 4.50 5.25 6.00
```

The first two arguments `from` and `to` specify the beginning and end of the sequence, while `by` sets the step of the increment, and `length.out` specifies the length (*i.e.*, number of elements) of the vector. See `?seq` for other examples and alternative usage.

Different regular sequences can be created with the function `rep()`. The default behaviour is to replicate a given vector for a number of times:

```
rep(1:4, 2)
```

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

Alternatively, you can input a vector of the same length as the second argument `times` (not named above), specifying how many times each element must be replicated:

```
rep(1:3, times=1:3)
```

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

Using the alternative argument `each` with a single integer number, the repetition is performed by element for the same number of times:

```
rep(1:4, each=2)
```

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

Note also how `rep()` works with character vectors as well:

```
rep(c("a","b","c"), 2)
```

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

Computing with vectors

Simple arithmetic computations involving arithmetic operators such as '+', '-', '*', '/', or '^' can be extended to vectors. For example:

```
numvec1 * numvec2
```

```
## [1]  12   0   0 -32 -28
```

Such simple operations between vectors are performed element by element and are an example of *vectorized* computations. When the length of vectors is different, the *recycling rule* (introduced in a different session) usually applies.

Similarly, *relational* operators and *logical* (or Boolean) operators can be used for vector computations. We have already met some of them in the previous sections. For instance:

```
1:3 == 3:1
```

```
## [1] FALSE  TRUE FALSE
```

```
c(T,F,T) & c(F,F,T)
```

```
## [1] FALSE FALSE  TRUE
```

```
numvec1 > 3 | numvec1 < 0
```

```
## [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

See in particular the operators '&' (logical AND) and '|' (logical OR).

The precedence in expressions involving multiple operators is as follows. The operator '^' is evaluated first, followed by the colon operator ':'. Then, the other arithmetic operators with the usual rules ('*' and '/' first, then '+' and '-'). Relational operators are evaluated next ('>'-type and '=='-type), followed by logical operators ('!', '&', and then '|'). The assignment operator '<-' is the last to be evaluated (see `?Syntax` for a more detailed description). An example:

```
3+2^2:10
```

```
## [1]  7  8  9 10 11 12 13
```

Here the exponential is evaluated first, producing `4`, which is then used as the first argument to create a sequence of integers up to `10`. Finally, the whole vector is summed to `3`. Round brackets '`()`' can be applied to force the order of precedence. For example, the expression:

```
(3+2)^2:10
```

```
##  [1] 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10
```

returns different results: here the content within round brackets is evaluated first. The use of round brackets is always recommended for complex expressions with multiple operators, to make the code clearer and avoid errors.

Although operators are useful for basic calculations, more complex computations with vectors usually involve the use of proper functions. An example is the calculation of mean and standard deviation of a numeric vector, such as:

```
mean(numvec1)
```

```
## [1] 2.6
```

```
sd(numvec2)
```

```
## [1] 6.403124
```

Some functions can return more complex objects, such as `summary()`, which returns a vector instead:

```
summary(numvec1)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    -7.0    -2.0     2.0     2.6     4.0    16.0
```

Other functions for mathematical calculations are `max()` and `min()`, `range()` and `sum()`, and `quantile()` (for computing percentiles). Rounding is obtained with `round()` and other functions (see `?round` for a complete list). See the related help pages about their usage.

Similarly to operators, also functions can be vectorized, meaning they can work element by element. An example is the arithmetic functions `log()` and `abs()` to compute the logarithm and absolute value:

```
log(1:5)
```

```
## [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379
```

```
abs(-2:2)
```

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

Many other functions accept vectors as arguments, and produce a wide range of computations. Some useful functions deal with the order of the elements in a vector, whatever its mode. For example `rev()` reverses the vector, while `sort()` rearranges the elements in ascending (or descending) order:

```
pj
```

```
## [1] "Eddie" "Stone" "Jeff"  "Mike"  "Matt"
```

```
rev(pj)
```

```
## [1] "Matt"  "Mike"  "Jeff"  "Stone" "Eddie"
```

```
sort(pj)
```

```
## [1] "Eddie" "Jeff"  "Matt"  "Mike"  "Stone"
```

The function `order()` returns the ascending (or descending) ordering permutation of the indices in numeric or character vectors:

```
numvec1
```

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

```
order(numvec1)
```

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

Other functions are applied to select elements. For example, `which()` and `which.max()`-`which.min()` returns the indices of the elements satisfying a statement and the maximum/minimum element, respectively:

```
which(numvec1>0)
```

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

```
which.max(numvec1)
```

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

More developed selections are provided by `match()` and the operator '`%in%`':

```
match(c("Mike","Stone"), pj)
```

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

```
pj %in% c("Stone","Chris","Layne","Mike")
```

```
## [1] FALSE  TRUE FALSE  TRUE FALSE
```

The former returns a vector with the indices of the elements in the second vector matching those in the first one. The latter returns a logical vector indicating which elements in the first vector match those in the second one. These functions are often applied in indexing for selecting subsets of data.

Constants and reserved words

Other objects with special features are available in R. An important object is the missing value indicator, which is defined by the logical constant `NA` (meaning 'Not Available'). For example, the vector:

```
c(2,NA,4,1)
```

```
## [1]  2 NA  4  1
```

has the second element missing. Note how the text is not quoted, thus meaning that `NA` is a reference to an object, in this case a pre-defined constant. A more detailed summary of how to deal with missing values will be provided in another session. A related value (but rarely important in everyday use) is `NaN` (meaning 'Not a Number'), which is a numeric constant often returned from a computation involving numeric values (such as `log(-2)` seen earlier.

Another important example is the null object, defined by `NULL`. This is often returned by expressions or functions whose value is undefined. Be careful that the `NULL` and `NA` objects refer to two different situations: in the former, the value is not defined, while in the latter it is missing. Let's see an example:

```
c(2,NA,3) ; c(2,NULL,3)
```

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

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

The concatenation works as expected for missing values, which are correctly included in the vector. Instead, the `NULL` values are excluded, as they are undefined.

The names chosen for the objects above are examples of `reserved words`, *i.e.* names that are retained for specific objects and cannot be used in the assignment of new objects (otherwise an error is returned). Other objects defined by reserved words are `TRUE` and `FALSE` for logical objects described earlier (but not the shortcuts `T` and `F`), or the numeric constant `Inf` for infinite values (returned for example by the expression `1/0`). Other examples of built-in constants are:

```
pi
```

```
## [1] 3.141593
```

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
```

The first one returns the Greek π (the ratio of the circumference of a circle to its diameter), while the last one produces the vector with the 26 letters of the Roman alphabet (see `?letters` for other examples). Other reserved words are used for control flow constructs (*e.g.*, `if` or `for`, described in other sessions.