The R building blocks

March 02, 2016

The interpreter

Let's start using the interpreter to get used to the basic arithmetic operations. They all work the way you expect to.

```
1 + 1
sqrt(2)
exp(1)
2^3
```

The same happens with comparisons:

```
1 > 1
0 == log(1)
(1 + 1 == 2) | (1 + 1 == 3)
(1 + 1 == 2) & (1 + 1 == 3)
```

We can assign values to a name using the *assignment operator* `<-`. R also supports `=` for assignment but not everywhere and in any case `<-` is more widely used. For instance, we can now assign the value `1` to the variable `a`, and every time we call `a` we will get the value it is referred to:

```
a <- 1
a
```

Data types

R offers a variety of data types. The most relevant ones are: *numeric*, *integer*, *character*, and *logical*.

```
1.0 # Numeric
1 # "Integer"
"ab" # Character
TRUE # logical
```

The `#` indicates a comment, which may or may not be at the beginning of the line.

It also offers logical constants for missing value (`NA`), "Not a Number" (`NaN`) and positive and negative infinity (`Inf` and `-Inf`):

```
NA
0/0
1/0
```

You can test the type of an object using the `is.` functions (*"is `x` an integer?"*):

```
is.numeric(1.0)
is.integer(1)
is.character("ab")
is.logical(FALSE)
```

A common mistake that we all make at some point (but hopefully you will not!) is to test whether an object is `NA` by using the `==` comparison.

```
NA == NA
```

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

There are good reasons for these "odd" behavior. I like this explanation from StackOverflow:

> Think of NA as meaning "I don't know what's there". The correct answer to 3 > NA is obviously NA because we don't know if the missing value is larger than 3 or not. Well, it's the same for NA == NA. They are both missing values but the true values could be quite different, so the correct answer is "I don't know."

We have special functions to check whether an object is `NA`:

```
is.na(NA)
```

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

and the same applies for `NaN`.

Note that `is.numeric` is a *function,* and all functions work the same way in R. They have a name followed by the argument(s) enclosed in parenthesis: `any_function(x)`. We will see a lot more about them in the following sessions.

Data structures

The most common (and intuitive) data structures are `vector`, `matrix`, `data.frame` and `list`. We will see `data.frame` in the next script.

A `vector` holds data of the same type (or `NA`). Vectors can be created with the function `c`, which stands for "concatenate".

```
c(1, 2)
c(1, 2, NA)
c(1, "a") # See how the first element is casted into character!
```

There are a few functions that allows us to create vectors with particular structures. For instance, we can create a sequence using the function `seq`:

```
c(1, 2, 3, 4)
seq(1, 4)
1:4 # A shorthand for `seq`
```

See how we used a function with two (positional) arguments? It is probably a good moment to take a look at more information about `seq` using `?seq`

A `matrix` is, unsurprisingly, a matrix: a rectangular table of elements of the same type.

```r
A <- matrix(c(1, 2, 3, 4), nrow=2, ncol=2)
A
```

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

Note how we passed all data as a unique object (a `vector`). Also notice how the matrix is filled. Contrary to intuition (but it makes perfect sense, if you think about it) is that a 1-dimensional `matrix` is not a `vector`.

Operations in matrices work "kind of" the way you would expect:

```r
B <- matrix(c(4, 3, 2, 1), nrow=2)
A * B
```

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

```r
A %*% B
```

```
##      [,1] [,2]
## [1,]   13    5
## [2,]   20    8
```

See how the first operation is element-wise, while the second is the standard matrix multiplication. We also have common operations like extracting the diagonar (which is a `vector`) or calculating the transpose or the inverse:

```r
t(A)
diag(A)
solve(A) # but A^-1
```

A `list` is an ordered collection of objects, possibly of different types.

```r
list("first"=c(1, 2, 3), "second"=c("a", "b"), "third"=matrix(NA, ncol=2, nrow=2))
```

Lists play a very important role in R, as we will see later on.

Indexing

Indexing allows us to access elements of a `vector` or `matrix` by position:

```r
a <- c(1, 2, 3)
a[1]
a[1:2]
a[-3]
a[1] <- 0
```

A matrix is two dimensional, so we need two elements to indicate cells in our structure but it works using the same principles:

```r
A <- matrix(c(1, 2, 3, 4), nrow=2)
A[1, 1]
A[1, ] # Empty space is short for "everything in this dimension"
A[, 1]
A[2, 2] <- 0
A[2, ] <- 0 # Recycling rule
```

A list uses a similar notation, but with two brackets instead of one. There are other ways to access elements.

```r
my_list <- list("a"=1, "b"=2, "c"=NA)
my_list[["a"]] # Name
my_list[[1]] # Position
my_list$a <- c("Not 1 anymore")
```

Notice how we can access elements by name or position.