Data types in R

Alla Tambovtseva

Data types

There are four basic data types in R:

- integer
- numeric
- character
- logical

An integer type corresponds to integers, a numeric type corresponds to all real numbers (integer, non-integer, rational, irrational), a character type stands for strings, i.e. text variables, and a logical type refers to boolean variables that can take only two values TRUE or FALSE.

Let's create a new numeric variable x1 (do not forget that in R we can use only a point as a decimal separator):

```r
x1 <- 9.5
```

Now we can check whether x1 belongs to every type mentioned above. It can be easily done with a set of commands that start with the prefix is.:

```r
is.numeric(x1) # TRUE
```

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

```r
is.integer(x1)
```

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

```r
is.character(x1)
```

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

However, usually there is no need to check all the types. We can use a special function class() so as to return namely the type of a variable:

```r
class(x1)
```

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

Type conversion

Imagine the following situation. We open a file and see that due to some mistakes of a data collector columns with indices have the wrong types: character instead of numeric. So as to fix it, we may apply commands starting with the as. prefix used for type conversion.

If a text in quotes consists of valid numeric symbols (digits and a point), the conversion is very simple:

```r
as.integer('9') # we get number 9
```

```
## [1] 9
```

```r
as.numeric('9')
```

```
## [1] 9
```

```
as.numeric('9.5') # we get number 9.5
```

```
## [1] 9.5
```

However, if there are inappropriate characters like letters, spaces, commas, underscores, the conversion will certainly fail returning an NA value (empty value, stems from Not Applicable) and printing a warning:

```
as.numeric('9,5')
```

```
## Warning: в результате преобразования созданы NA
```

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

So as to convert such character variables to numeric ones, we will need a replacement function that can substitute "," by "." (we will discuss it later). However, if we work with large data tables obtained from files, there is no need to replace a decimal separator and make every incorrect column numeric. While uploading a file, we can specify the decimal separator and make R understand commas instead of points correctly.

Numeric variables: some details

With numeric variables we can perform the same operations as with numbers: addition, substraction, division, rasing to powers, etc.

```
a <- 25
b <- 15
```

Let's save the sum of the variables:

```
sum_ab <- a + b
sum_ab
```

```
## [1] 40
```

And now let's raise a to the power of b:

```
a ^ b
```

```
## [1] 9.313226e+20
```

The result above looks strange. The letter e here stands for 10, not for the exponent $e \approx 2.7$. This is so called scientific or exponential or computer notation for a number. Here we should multiply 9.313226 by 10^{20} and this will be the result in the usual form. If, vice versa, the result was very small and close to zero, the e would be raised to the negative power:

```
2/237899 # the same as 0.000008406929
```

```
## [1] 8.406929e-06
```

Character variables: some details

What can we do with character variables? Different things, for instance, change the case of letters, transliterate texts, split texts into pieces by a particular symbol, etc. However, the basic operation usually needed in data analysis is replacement. Sometimes we want to get rid of spaces in variable names or substitute inconvenient symbols in texts.

For example, we have the character variable group with group names and we want to change # to -.

```
group <- "group#1 group#2 group#3"
```

To do so, we can apply the function gsub that substitutes all the occurences of a particular symbol (gsub can be read as global substitute or general substitute). In R there is also the sub function, it is used for replacement as well, but it changes only the first occurence of a symbol.

```
gsub("#", "-", group)
```

```
## [1] "group-1 group-2 group-3"
```

Note that in the code above we just look at the possible changes and do not save these transformations. If we print the value of group, we will get the same results as before:

```
group
```

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

That is because all basic objects in R are immutable, i.e. they cannot be changed in any other way than assigning new values to old variables. Thus, to save changes, we have to execute the following:

```
group <- gsub("#", "-", group)
group # it changed
```

```
## [1] "group-1 group-2 group-3"
```

Logical operations and testing conditions

Logical operations are used to formulate conditions or to check whether they hold true. Let's create two numeric variables and compare them.

```
x <- 2
y <- 10
```

Usual operations:

```
x > y
```

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

```
x < y
```

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

Non-strict inequalities (no space between two operators):

```
x <= y
```

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

```
x >= y
```

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

As for testing equality, in programming we have to use a double = instead of a single =. That is because a single = implies value assignment while a double one corresponds to condition testing.

```
x == y
```

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

The opposite (non-equality) is tested via != operator that replaces $\neq$ in programming:

```
x != y
```

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

We can formulate complex conditions using logical conjunction (AND) and logical disjunction (OR, inclusive OR).

```
x & y < 5 # x < 5 and y < 5 hold true at the same time
```

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

```
x | y < 5 # x < 5 or x < 5, at least one is correct (one or both)
```

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

In terms of set theory these operations correspond to intersection ($\cap$) and union ($\cup$).