R_programming_DataStructures

Frank Fichtenmueller

31 October 2016

R Working with basic data structures in R

In R there are basic data types:

- **Datatypes**
  - character
  - numeric (real number)
  - integer
  - complex
  - logical / boolean (TRUE/FALSE)

```
x <- c(1,3,4,5,6)
class(x)
```

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

```
# Numeric
x
```

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

```
# The logical
as.logical(x)
```

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

```
class(as.logical(x))
```

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

```
# The charcter
as.character(x)
```

```
## [1] "1" "3" "4" "5" "6"
```

```
class(as.character(x))
```

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

```
# The complex number
as.complex(x)
```

```
## [1] 1+0i 3+0i 4+0i 5+0i 6+0i
```

```
class(as.complex(x))
```

```
## [1] "complex"
```

```r
#
```

There are different datastructures to hold combinations of these datatypes.

- **Datatypes**
  - The vector
  - The matrix
  - The list
  - The Factor Levels

Lets look at the basic vector first

It can only **hold ony type of data** and will therefore find coerce the data, to find a common minimum representation. As this is done automatically, it can lead to problems if not taken into account.

```r
# It can hold only a specific type of data, so conversion happens under the hood
l <- c('a', 1, 4+0i, TRUE)
l
```

```
## [1] "a"    "1"    "4+0i" "TRUE"
```

```r
# Its class is set to be able to accompany all given inputs
class(l)
```

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

```r
class(l[4])
```

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

```r
# It can be indexed, therefore it remembers position
l[1]
```

```
## [1] "a"
```

```r
l[1:3]
```

```
## [1] "a"    "1"    "4+0i"
```

```r
#The following mixes can occur
y <- c(1.7, 'a') #Character
y <- c(TRUE, 2)  #numeric
y <- c(1, 5+0i)  #complex
```

Attributes

Datatypes carry **5 basic kind** of attributes that sets them appart and influences their specific behaviour.

- names, dimnames
- dimensions (matrices, arrays)
- class (e.g. integer, numeric, etc..)
- length
- other user-defined attributes/metadata

These can be accessed using the `attributes()` function call. If the R object does not contain attributes the function call will return `NULL`.

Even though an object might not return `attributes`, we can still use the following functions to adress attributes directly.

- `dim()` will return the dimensions of the object (Null for Vector)
- `length()` returns the number of entries in the object

```
x
```

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

```
attributes(x)
```

```
## NULL
```

```
length(x)
```

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

```
dim(x)
```

```
## NULL
```

Matrices

Matrices are vectors with a dimension attribute. The dimension is itself an integer vector of length 2(number of rows, number of columns)

```
m <- matrix(nrow=2, ncol=3)
dim(m)
```

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

```
attributes(m)
```

```
## $dim
## [1] 2 3
```

They are constructed column wise by combing single column vectors along the n-row dimension. Given a `vector of length n` they will split it according to the `nrow` attribute.

```r
matrix(1:6, nrow=3, ncol=2)
```

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

```r
#The function call will infer the missing dimension automatically
matrix(1:10, nrow=5)
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

```r
# Or you recombine them by adding a dimension to a vector
v <- c(1:10)
dim(v) <- c(2,5)
v
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

They can be combined out of single vectors trough **column-wise or row-wise** binding. * `cbind` to combine the vectors column-wise * `rbind` to combine row-wise

```r
x <- 1:3
y <- 10:12
# Column wise
cbind(x,y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```r
# Row wise
rbind(x,y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Lists

They are a special type of vector that can cointain elements of different classes. They are a very important data type. Especially in combination with the various `apply()` functions, they are a powerful combination.

They can be directly created using the `list()` function, which takes an arbitrary number of arguments to concatinate.

```r
l <- list(1,2,3, 'a', 'b', 'c', TRUE, FALSE, TRUE)
l
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] "a"
##
## [[5]]
## [1] "b"
##
## [[6]]
## [1] "c"
##
## [[7]]
## [1] TRUE
##
## [[8]]
## [1] FALSE
##
## [[9]]
## [1] TRUE
```

If we want to create a `list` with a preset length to be filled e.g. by a function call we can specify this with the `vector('list', length=??)` command call.

```r
vector('list', length=3)
```

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

Factors & Factor Levels

To represent categorial variables, we can use the `Factor` type in R. This can be unordered or ordered. Once can think of a factor as an integer vector where each integer has a label.

Using a labeled factor is better, as labels are self describing, instead of using dummy variables (e.g. 0|1)

```r
x <- factor(c('yes', 'no', 'yes', 'yes', 'no', 'no'))
x
```

```
## [1] yes no  yes yes no  no
## Levels: no yes
```

The `factor()` call automatically creates the factor levels by identifying the number of unique elements associated. Now we can apply more advanced summary descriptions to the data

```r
table(x)
```

```
## x
##  no yes
##   3   3
```

```r
# And use the unclass() command to identfy the underlying structure of the data
unclass(x)
```

```
## [1] 2 1 2 2 1 1
## attr(,"levels")
## [1] "no"  "yes"
```

Often factors will be automatically created when importing a dataset using e.g. `read.table()` command. As this will create factor levels that are ordered by ascending first letters, we might want to give them a more indicative structure manually.

This can be done useing the `levels` attribute to the `factor()` call.

```r
x <- factor(c('yes', 'no', 'yes', 'no', 'no'))
x # Levels are now in alphabetical ordering
```

```
## [1] yes no  yes no  no
## Levels: no yes
```

```r
# We can reassign a new ordering of the factors by directly assigning a vector
levels(x) <- c('yes', 'no')
x
```

```
## [1] no  yes no  yes yes
## Levels: yes no
```

```r
# Or take care of the ordering in the function call itself
x <- factor(c('yes', 'no', 'yes', 'yes'), levels = c('yes', 'no'))
x
```

```
## [1] yes no  yes yes
## Levels: yes no
```

Missing Values / NANs

Indicated by NA or NaN.

- `is.na()` is used to test object if they are NAs
- `is.nan()` is used to test for NaN
- NA values have a clas, so therea are ineger NA, character NA, etc.
- A NaN value is also NA but the converse does not hold true

```r
# Create a vector with NA in it
x <- c(2,3, NA, 10, 3)

# Function returns a boolean vector
is.na(x)
```

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

```r
# This can be used for indexing to select NAs out
x_clean <- x[!is.na(x)]
x_clean
```

```
## [1]  2  3 10  3
```

```r
# We can impute the missing data with an average measurement e.g.

x[is.na(x)] <- 10
x
```

```
## [1]  2  3 10 10  3
```

DataFrames

Used to store tabular data in R. They are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of theach element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. Matices must have every elemt be the same class. Data fames have a special attribute called `row.names` which indicate information about each row of teh data frame.

They are usually not created manually, but read into R using

- `read.table()`
- or `read.csv()`

However, data frames can also be created explicitly with the `data.frame()` function or they can be coerced from other types of objects, like lists.

For some applications, especially in machine learning, which mostly relies on linear algebra and matrix algebra applications, we have to work with the data inside a table as a matrix. To this end we can use the `data.matrix()` or the `as.matrix()` even though mostly you will need the result of the previous.

Lets's look at an example:

```r
# First we create a data frame manually
x <- data.frame(foo = 1:4, bar=c(T,T,F,F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```r
# With the attributes dimension
dim(x)
```

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

```r
nrow(x)
```

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

```r
ncol(x)
```

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

DataTables can be indexed trough positional attributes, just as matrices. And they can aditionally be indexed using the row & column names.

```r
x[1] # Extracts the first column
```

```
##   foo
## 1   1
## 2   2
## 3   3
## 4   4
```

```r
x[,1] # Extracts the complete first row
```

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

```r
x$foo # Indexes the first column
```

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

```r
x$bar # Indexes the second column
```

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

Names

As R is a statistical programming language, it is layed out to work closely with procedures used in everyday statistical analysis. Therefore the use of **named variables** in handy and easily accessible.

```
x <- 1:3
names(x)
```

```
## NULL
```

```
# We can just assign names as a string vector to the names attribute of the variable
names(x) <- c('New York', 'Washington DC', 'Los Angeles')
x
```

```
##      New York Washington DC   Los Angeles
##             1             2             3
```

In the same way we can assign names to **list** objects, which is very usefull when indexing or applying summary functions on them.

```
x <- list('Los Angeles' = 1, 'Boston' = 2, 'New York City' = 3)
x
```

```
## $`Los Angeles`
## [1] 1
##
## $Boston
## [1] 2
##
## $`New York City`
## [1] 3
```

```
names(x)
```

```
## [1] "Los Angeles"   "Boston"        "New York City"
```

matrices can have row and dim names

```
m <- matrix(1:4, nrow=2, ncol=2)
dimnames(m) <- list(c('a', 'b'), c('c', 'd'))
m
```

```
##   c d
## a 1 3
## b 2 4
```

```
# Or we can again assign them individually
rownames(m) <- c('new', 'old')
colnames(m) <- c('first', 'last')
m
```

```
##     first last
## new     1    3
## old     2    4
```

NOTE: There is a difference in setting col & row names in matrices and dataFrames

Object | Set column names | set row names