R's (basic) data objects

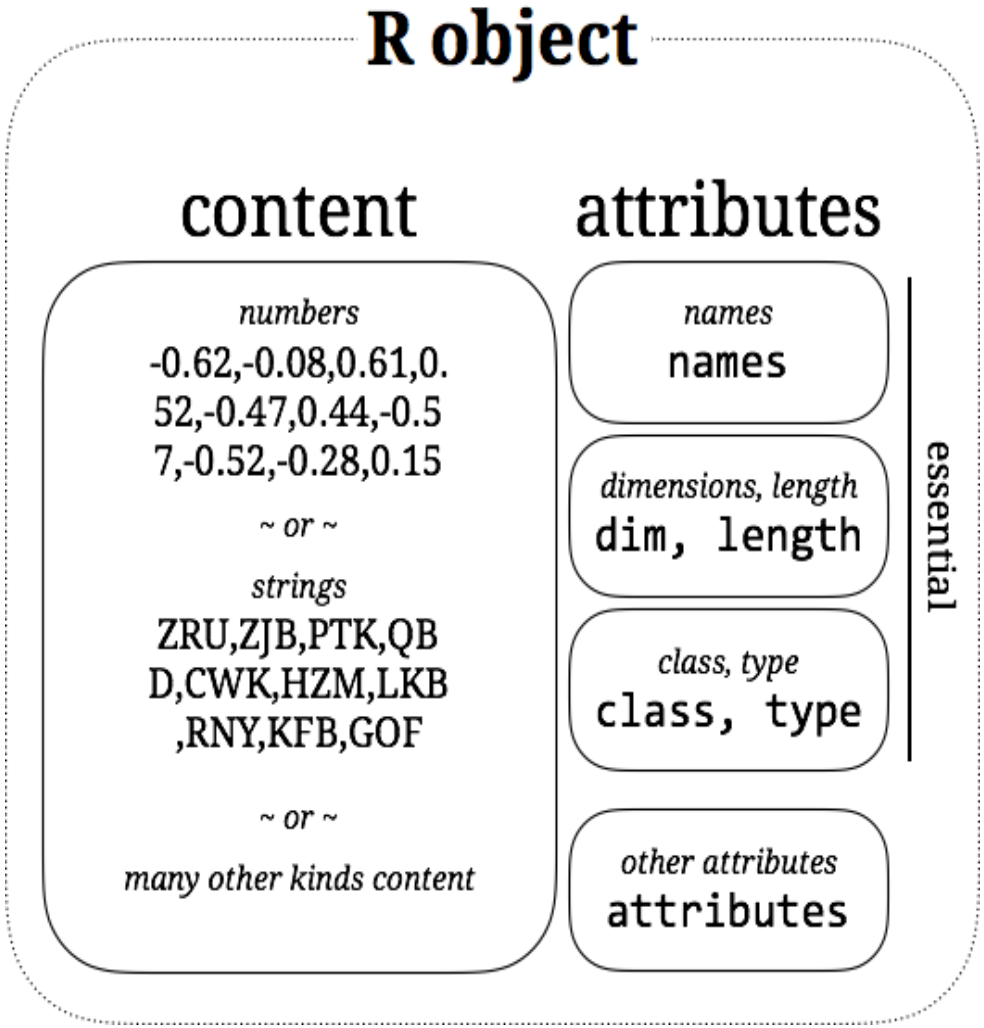
The R Bootcamp
Twitter: @therbootcamp

September 2017

Objects

> "Everything in R is an object"
>
> *John Chambers*

- R's objects are have content and attributes.
- The content can be anything from numbers or strings to functions or complex data structures.
- Attributes often encompass names, dimensions, and the class or type of the object, but other attributes are possible.
- Practically all data objects are equipped with those three essential attributes.



Data objects

- Objects either contain elements of the **same type** (homogeneous) or **different types** (heterogeneous).
- Homogeneous objects are always **flat**, i.e., contain no nested structure.
- Lists can contain anything, even lists (**recursive**), whereas data frames underly certain restrictions in terms of type and dimensions.

	Homogeneous	Heterogeneous

1d	**atomic vectors**
	`c(), numeric()`
	`character(), etc.`
		lists
		`list()`

2d	**matrices**
	`matrix(), cbind()`
		data frames
		`data.frame(),`
		`tibble()`

nd	**arrays** `matrix(), cbind()`	- -

Vectors

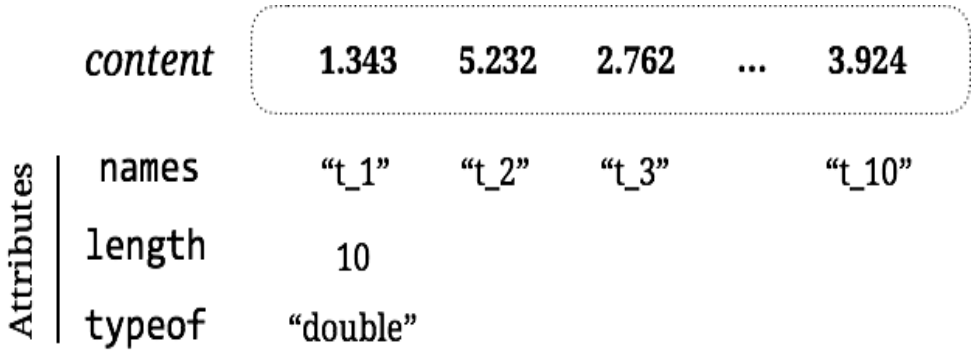
R's most **basic (and smallest) data format** - even single values are implmented as vectors.

```
# creating a vector (incl. names)
my_vec <- c(t_1 = 1.343, t_2 = 5.232)

# vectors are always flat
my_vec <- c(1.343, c(5.232, 2.762))

# naming vectors
my_vec <- c(t_1 = 1.343, t_2 = 5.232)
names(my_vec) = c("t_1","t_2")

# evaluting inherent attributes
names(my_vec)
length(my_vec)
typeof(my_vec)
```



Types

A vector contains elements of one of four **basic types**: `integer`, `double`, `numeric`, and `character`. You can **test** the type using `typeof()` or the type-specific `is.*()`, e.g., `is.integer()`.

```
# numeric vectors
my_vec <- c(1.343, 5.232)
typeof(my_vec) ; is.integer(my_vec)
```

```
## [1] "double"

## [1] FALSE
```

```
# integer vectors (overruled by R)
my_vec <- c(1, 7, 2)
typeof(my_vec) ; is.integer(my_vec)
```

```
## [1] "double"

## [1] FALSE
```

```
# logical vectors
my_vec <- c(TRUE, FALSE)
typeof(my_vec) ; is.logical(my_vec)
```

```
## [1] "logical"

## [1] TRUE
```

```
# character vectors
my_vec <- c('a', 'hello', 'world')
typeof(my_vec) ; is.character(my_vec)
```

```
## [1] "character"

## [1] TRUE
```

Coercion

R allows you to **flexibly change types** into another using `as.*()`, which cam however lead to a **loss of information!** Often **coercion occurs automatically**. Mathematical functions (+, log, abs, etc.) will coerce to a double or integer, logical operations (&, |, any, etc) will coerce to a logical.

```
# double to integer
my_vec <- as.integer(c(1, 7, 2))
is.integer(my_vec)
```

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

```
# double to logical (and back - > info loss
my_vec <- as.logical(c(1.21, 7.24, 0))
as.logical(my_vec) ; as.numeric(as.logical(
```

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

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

```
# logical operation -> logical type
c(1, 7, 2) > 3
```

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

```
# mathematical operation -> numeric type
c(TRUE, FALSE, TRUE) + 3
```

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

```
# anything can be coerced to character
as.character(c(TRUE, FALSE, 0))
```

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

Factors

Factors are a special case of vector that can contain only **predifined values** defined in the attribute `levels`. Factors are rarely useful and sometimes **dangerous**, yet several R functions will coerce `character` vectors to `factor`.

```
# create a factor
my_fact <- factor(c('A','B','C'))
levels(my_fact)
```

```
## [1] "A" "B" "C"
```

```
# test type
typeof(my_fact)
```

```
## [1] "integer"
```

```
# add value at 4th position
my_fact[4] <- 'A'
# my_fact[4] <- 'D' # leads to error
```

```
# dangerous behavior of factors pt. 1
my_fact <- factor(c('A','B','C'))
mean(as.integer(my_fact))
```

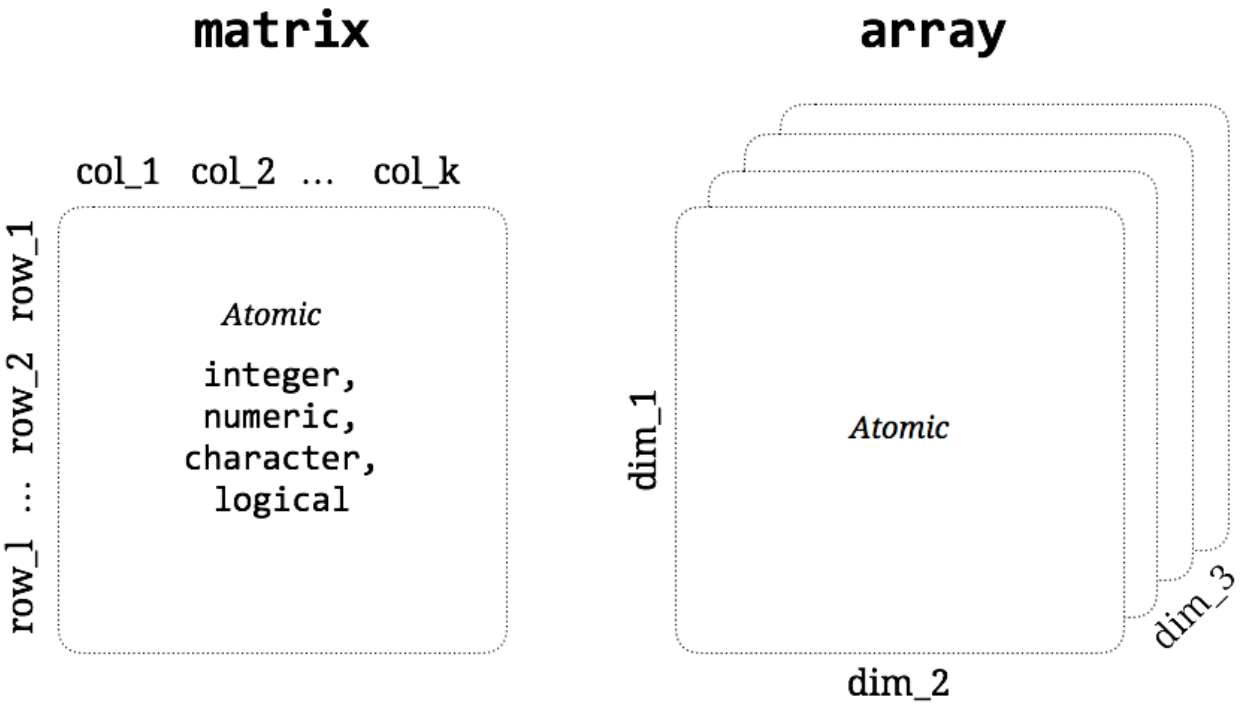
```
## [1] 2
```

```
# dangerous behavior of factors pt. 2
my_fact <- factor(c(1.32,4.52,.23))
as.double(my_fact) # ranks
```

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

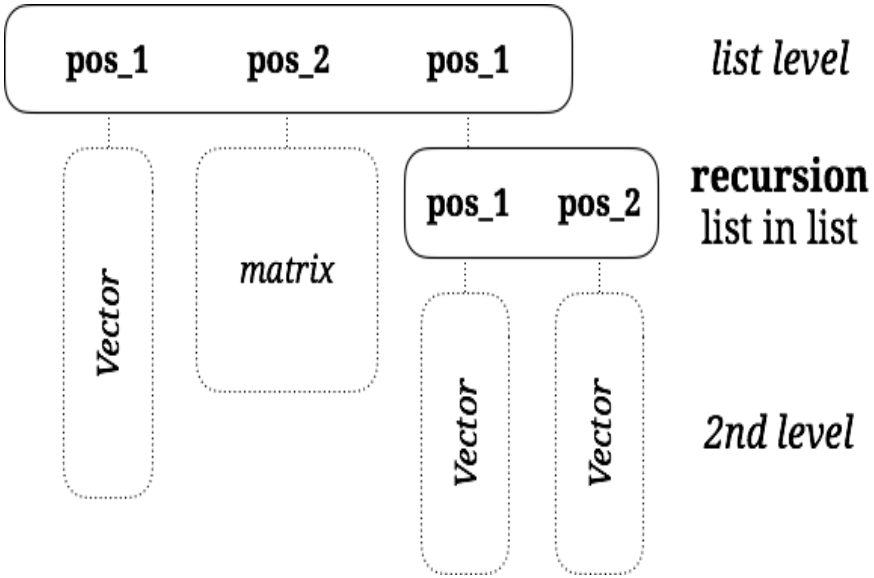
Matrices & Arrays

Matrices and arrays are **straightforward extensions** of vectors with 2 (matrix) or *n* dimensions. Both are **atomic** (carry only one type), have names (col-, row-, and dimnames) and dimension attributes Compared to vectors, lists, and data frames, **they usually play a lesser role in most applications**.



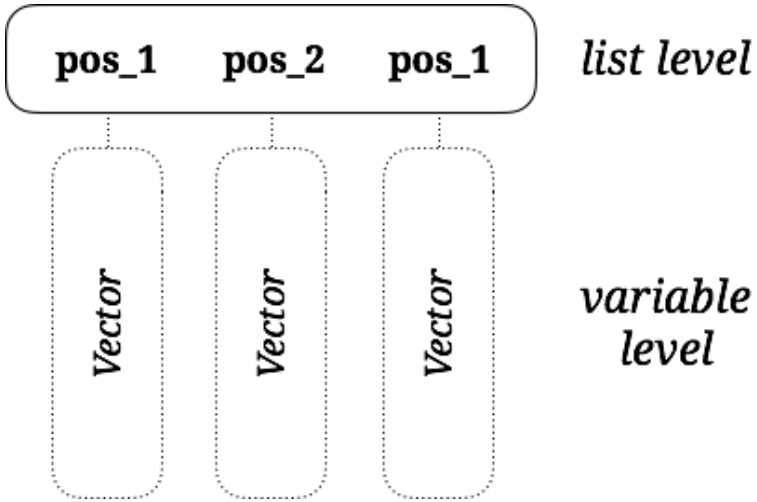
lists

- Lists are **R's swiss army knife**. They often are used for outputs of statistical functions e.g., `lm()`.
- Lists have **non-flat** structures that take **any object type**, inluding lists, which renders lists **recursive**.
- Intuitively lists can be understood as a meta-vector that on top of its content contains an overarching **organizational layer**.
- To create a list use `list()` or `as.list()`



`data.frames`

- Data frames have been **R's main data format** for data representation.
- **Data frames are lists** with specific **requirements**:
  - Every element must be a vector.
  - The lengths of the vectors must be equal or multiples of another.
- To create a data frame use `data.frame()` and `as.data.frame`.



Accessing & changing **atomic** objects pt. 1

To access and change atomic data objects use **brackets** `[` or **names**. When using `[` be sure to use as many indices as there are dimensions, e.g., `[3]` for vectors and `[1,3]` for matrices. Omittence of an index means for all elements.

```
# retrieve second element from vector
my_vec <- c('A','B','C')
my_vec[2]
```

```
## [1] "B"
```

```
# change the second element
my_vec[2] <- 'D' ; my_vec
```

```
## [1] "A" "D" "C"
```

```
# change beyond length(my_vec)
my_vec[7] <- 1 ; my_vec
```

```
## [1] "A" "D" "C" NA  NA  NA  "1"
```

```
# create matrix
my_mat <- matrix(c(1:6), nrow=2)
my_mat
```

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

```
# retrieve second row from matrix
my_mat[2, ] ; my_mat[2, 1]
```

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

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

Accessing & changing **atomic** objects pt. 2

Provided the object is equipped with a **name attribute**, indexing can also be accomplished using the element's name.

```
# retrieve element 'a' from vector
my_vec <- c(a = 1, b = 4, c = 5)
my_vec['a']
```

```
## a
## 1
```

```
# change the element
my_vec['c'] <- 'D'

# change beyond length(my_vec)
my_vec['d'] <- 1 ; my_vec
```

```
##   a   b   c   d
## "1" "4" "D" "1"
```

```
# create matrix
my_mat <- matrix(c(1:6), nrow=2)
colnames(my_mat) <- c('v_1','v_2','v_3')
rownames(my_mat) <- c('c_1','c_2')

# retrieve second row from matrix
my_mat['c_1', ] ; my_mat['c_1', 'v_2']
```

```
## v_1 v_2 v_3
##   1   3   5

## [1] 3
```

Accessing & changing **complex** objects pt. 1

In accessing and changing complex objects the additional `list`-layer needs to be taken into account. Single brackets `[` will select elements within the list, not the object behind those elements. To select the object behind the element use **double brackets** '[['. Additionaly, complex objects can be conveniently accessed using the **dollor operator $**. In order to further descend into the `list`'s structur append **multiple select operators**, e.g., `my_list[[1]][[2]]`.

```
# retrieve elements from list
my_list <- list('A'=c('A','B'),
                'B'=list(c(1,2,3),
                         c(TRUE,FALSE,TRUE)
my_list[1] ; my_list[[1]] ; my_list[['A']]
```

```
## $A
## [1] "A" "B"

## [1] "A" "B"

## [1] "A" "B"
```

```
# retrieve deep elements in list
my_list <- list('A'=c('A','B'),
                'B'=list(c(1,2,3),
                         c(TRUE,FALSE,TRUE)
my_list[[2]][1] ; my_list[[2]][[1]] # etc
```

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

## [1] 1 2 3
```

Accessing & changing **complex** objects pt. 2

Data frames can be accessed **exactly like lists**. In addition, data frames allow for a matrix-like access using **single bracket** `[`. Note however that selecting rows using single bracket returns a data frame, whereas for selecting columns returns a vector.

```
# retrieve elements from list
my_df <- data.frame('v_1'=c('A','B','C'),
                    'v_2'=c(1,2,3))
my_df[1] ; my_df[[1]] ; my_df[['v_1']]
```

```
##   v_1
## 1   A
## 2   B
## 3   C

## [1] A B C
## Levels: A B C

## [1] A B C
## Levels: A B C
```

```
# retrieve elements from list
my_df <- data.frame('v_1'=c('A','B','C'),
                    'v_2'=c(1,2,3))
my_df[1,] ; my_df[,1] ; my_df[1,2]
```

```
##   v_1 v_2
## 1   A   1

## [1] A B C
## Levels: A B C

## [1] 1
```

Object algebra

R has implementations of most operations of vector and matrix algebra. As R is otherwise a slow language, it is often desirable to make use of them.
-

```
# create objects
my_mat <- matrix(1:9, ncol=3)
my_vec <- c(1:3)

# object times scale (also a vector)
my_mat * 5 ; my_vec * 5
```

```
##      [,1] [,2] [,3]
## [1,]    5   20   35
## [2,]   10   25   40
## [3,]   15   30   45

## [1]  5 10 15
```

```
# create objects
my_mat <- matrix(1:9, ncol=3)
my_vec <- c(1:3)

# matrix multiplication
my_vec %*% my_mat
```

```
##      [,1] [,2] [,3]
## [1,]   14   32   50
```

Practical

Link to practical