R data types: vectors

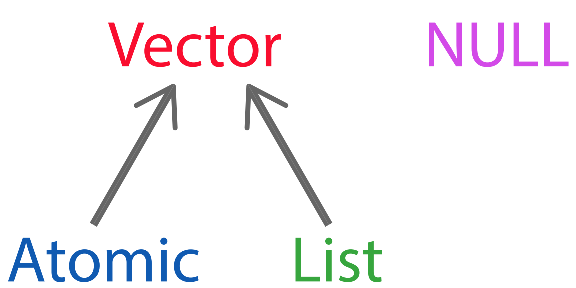
Alberto Garfagnini

Università di Padova

R lecture 2

R data types

- the most important family of data type is: `vector`
- (all other data types are known as `nodes` (i.e. functions and environments)
- vectors can be:
- atomic : all elements must have the same type
- lists : elements can be of different types
- `NULL` serves as generic zero length vector



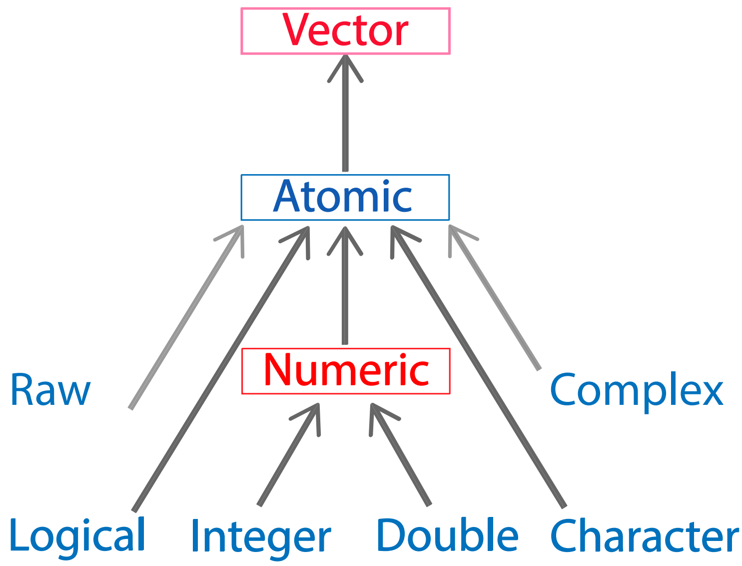
- every vector can have attributes
- two important attributes are: `dimension` and `class`
- `dimension` allows to create a matrix (`dim=2`) and array (`dim>2`)
- `class` powers the S3 object system in R

Atomic Vectors

R atomic vectors

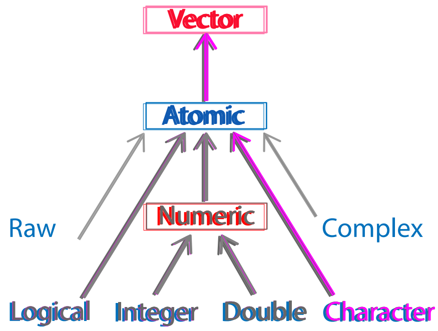
- There are 4 basic types of atomic vectors:
  - logical : `TRUE, FALSE`
    abbreviated with `T` and `F`
  - double : `2.75` (decimal), `1.23E4` (scientific) or `0xcafe` (hexadecimal)
  - integer : written similar to double, but with an `L` suffix (`123L, 1E3L` or `0xcafeL`
  - character : `"a", "a word"`. These are strings surrounded by `"` or `'`

    special characters are escaped with \
    See `?Quotes` for details
- and 2 rare types:
  - complex : `4.5 + 3i`
  - raw : (intended to hold raw bytes)



R atomic vectors

- R provides a set of functions to examine an object:
  - `class()` : return the object class type
  - `typeof()` : return the the object's data type
  - `length()` : return the number of elements
  - `attributes()` : return object metadata
  - `str()` : display the internal structure of an R object




```
x <-3
class(x)
%> [1] "numeric"
typeof(x)
%> [1] "double"
length(x)
%> [1] 1
str(x)
%> num 3
```

```
y <- 3L
class(y)
%> [1] "integer"
typeof(y)
%> [1] "integer"
length(y)
%> [1] 1
str(y)
%>  int 3
```

```
z <- x>0
class(z)
%> [1] "logical"
typeof(z)
%> [1] "logical"
length(z)
%> [1] 1
str(z)
%>
logi true
```

```
w <- 'three'
class(w)
%> [1] "character"
typeof(w)
%> [1] "character"
length(w)
%> [1] 1
str(w)
%>
chr "three"
```


R vectors

- scalar types do not exist, they are considered one-element vectors

```
x <-4.7; length(x)
%> [1] 1
```

- longer vectors are usually created with the concatenate, `c()`, function
- the size of a vector is determined at creation time

```
y <- c(1, 2, 5, 8)
str(y)
%>  num [1:4] 1 2 5 8
```

- `c()` calls an be combined:

```
y <- c(y, 12, c(1, 7, 8)
str(y)
%>  num [1:8] 1 2 5 8 12 1 7 8
```

Generating sequences of numbers

- an useful way to create a vector is to generate a sequence of numbers

```
0:10 # a sequence from 0 to 10, in steps of 1
%>  [1]  0  1  2  3  4  5  6  7  8  9 10

15:5 # a sequence from 15 down to 5
%>  [1] 15 14 13 12 11 10  9  8  7  6  5
```

- the seq() function allows to generate sequences in steps other than 1

```
seq(-2, 3, 0.5)
%>  [1] -2.0 -1.5 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  2.5  3.0

seq(6, 4.2, -0.2)
%>  [1] 6.0 5.8 5.6 5.4 5.2 5.0 4.8 4.6 4.4 4.2
```

- or with a fixed vector length

```
seq(from=0.04, to=0.14, length=6)
%> [1] 0.04 0.06 0.08 0.10 0.12 0.14

seq(from=0.04, to=0.14, length=7)
%> [1] 0.04000000 0.05666667 0.07333333 0.09000000 0.10666667
%> [6] 0.12333333 0.14000000
```


Generating replicated values

- the function rep() replicates the values in a vector

```
rep (9,5) # replicate 5 times the number 9
%> [1] 9 9 9 9 9

rep(1:4, 2) # replicate twice the 1:4 sequence
%> [1] 1 2 3 4 1 2 3 4

rep(1:4, each=2) # replicate twice each sequence number
%> [1] 1 1 2 2 3 3 4 4

rep(1:4, each=2, times=3)
%>  [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4


# replicate each sequence number a different number of times
rep(1:4, 1:4)
%>  [1] 1 2 2 3 3 3 4 4 4 4

rep(c("cat","dog","mouse"), c(2,3,2))
%> [1] "cat"   "cat"   "dog"   "dog"   "dog"   "mouse" "mouse"
```

Infinity and 'not numbers'

- calculations can lead to results which go to $\pm\infty$ or are indeterminate

```
4/0
%> [1] Inf

-15/0
%> [1] -Inf
```

- but calculations involving $\pm\infty$ are properly evaluated

```
exp(-Inf)
%> [1] 0

exp(Inf)
%> [1] Inf

0/Inf
%> [1] 0

(0:3)^Inf
%> [1]   0   1 Inf Inf
```


Infinity and 'not numbers'

- some calculations may lead to results which are indeterminate, i.e. not numbers

```
0/0
%> [1] NaN

Inf - Inf
%> [1] NaN

Inf/Inf
%> [1] NaN
```

- there are functions to test weather a number is finite or infinite

```
x <- -4.5
is.finite(x)
%> [1] TRUE

is.infinite(c(-4.5, 0/0, exp(Inf)))
%> [1] FALSE FALSE  TRUE

is.nan(c(-4.5, 0/0, exp(Inf)))
%> [1] FALSE  TRUE FALSE
```

Missing or unknown values

- R represents missing or unknown values with the sentinel NA
- but most computations with NA will return NA

```
NA > 0; 2.7*NA; ! NA
%> [1] NA
%> [1] NA
%> [1] NA
```

- exception: when some identity holds for all possible inputs

```
NA ^ 0
%> [1] 1
NA | TRUE
%> [1] TRUE
NA & FALSE
%> [1] FALSE
```

- but, how do we check for NA values ?

```
y <- c(4, NA, -8)
y == NA   # it does not work, sets all to NA
%> [1] NA NA NA

y == "NA" # this does not work, either
%> [1] FALSE    NA FALSE

is.na(y) # this is the proper way
%> [1] FALSE  TRUE FALSE

y[! is.na(y)] # produce a vector with NAs removed
%> [1]  4 -8
```


Missing values: NA

- some built-in functions allow to skip NAs from computations

```
y <- c(4, NA, -8) ; mean(y)
%> [1] NA
mean(y, na.rm=TRUE)
%> [1] -2
```

- how to we find the locations of NA values within a vector ?

```
vmv <- c(1:6, NA, NA, 8:12)
# Get the index of the values
seq(along=vmv)
%> [1]  1  2  3  4  5  6  7  8  9 10 11 12 13

seq(along=vmv)[is.na(vmv)] # and now of the NAs
%>[1] 7 8

which(is.na(vmv)) # a simpler way exists
%>[1] 7 8
```

- if NAs are 'zero-count' values, we may want to replace them with 'zeros'

```
vmv[is.na(vmv)] <- 0 ; vmw
%> [1]  1  2  3  4  5  6  0  0  8  9 10 11 12

vmv[which(is.na(vmv))] <- 0 ; vmw
%> [1]  1  2  3  4  5  6  0  0  8  9 10 11 12

ifelse(is.na(vmv), 0, vmv) # use the 'vectorized' ifelse function
%> [1]  1  2  3  4  5  6  0  0  8  9 10 11 12
```

Vector indexing

- the advantage of vector-based language is that it is simple to make computation involving all values in the vector

```
probe <- c(4, 7, 6, 5, 6, 7)
length(probe)
%> [1] 6
mean(probe)
%> [1] 5.833333
min(probe)
%> [1] 4
max(probe)
%> [1] 7
```

- subscripting is done through square brackets `[]` (indexing starts at '1')

```
index <- c(1, 3, 4, 6) # a vector of selected indexes
probe[index]
%> [1] 4 6 5 7
probe[c(1, 3, 4, 6)] # this is also valid
%> [1] 4 6 5 7
```


Vector indexing

- unwanted values can be dropped using negative indexes

```
probe <- c(4, 7, 6, 5, 6, 7) ; probe
%> [1] 4 7 6 5 6 7

probe[-1] # remove the first element
%> [1] 7 6 5 6 7

probe[-length(probe)] # remove the last element
%> [1] 4 7 6 5 6
```

- write a function to remove the smallest two values (with index 1 and 2) and largest two values (which will have subscripts `length(x)` and `length(x)-1`)

```
trim <- function(x) sort(x)[-c(1,2,length(x)-1,length(x))]
trim(probe)
%> [1] 6 6
```

- sequences can be used to extract values

```
probe[1:3]
%> [1] 4 7 6
probe[seq(1,length(probe),2)]
%> [1] 4 6 6
probe[seq(1,length(probe),2)] # get odd indexes values
%> [1] 4 6 6
probe[seq(2,length(probe),2)] # get even indexes values
%> [1] 7 5 7
```

Vector attributes

- more complicated data structures, like matrices, arrays, factors and datetimes are built on top of vector by adding attributes
- attributes are also used to create user-defined S3 classes
- attributes are name/value pairs
- they can be retrieved/modified with `attr()` or retrieved en masse with `attributes()`

```
counts <- c(25,12,7,4,6,2,1,0,2)

attr(counts, "nx") <- "count1"
attr(counts, "ny") <- "events"

attr(counts, "nx")
%> [1] "count1"

attributes(counts)
%> $nx
%> [1] "count1"

%> $ny
%> [1] "events"
```

- most attributes are lost during operations, unless they are part of an S3 class
- only `names` and `dim` attributes are preserved

Vector attribute: `names`

- a vector can be given a name in three ways

```
% When creating it
x <- c(a = 1, b = 2, c = 3)

% By assiging a character vector to names()
x <- 1:3
names(x) <- c("a", "b", "c")

% Inline, with the function setNames()
x<- setNames(1:3, c("a", "b", "c"))

x
%> a b c
%> 1 2 3
```

- vector names can be retrieved with the function `names()`

```
names(x)
%> [1] "a" "b" "c"
```

- and removed with `uname()`, or setting `names(x) <- NULL`

```
unname(x)
[1] 1 2 3
names(x) <- NULL
x
%> [1] 1 2 3
```

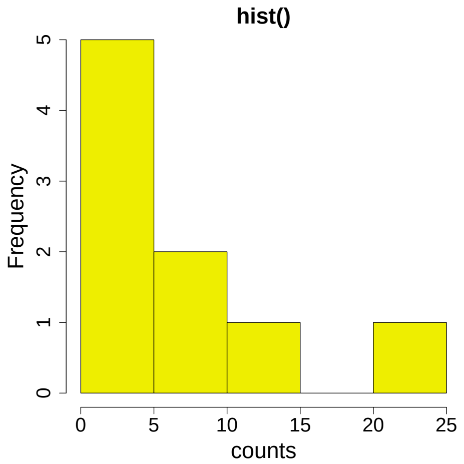
Example : naming vector elements

- sometimes is is useful to have values in a vector labelled
- for instance, we have a vector of counts occurrence of 0, 1, 2, . . .

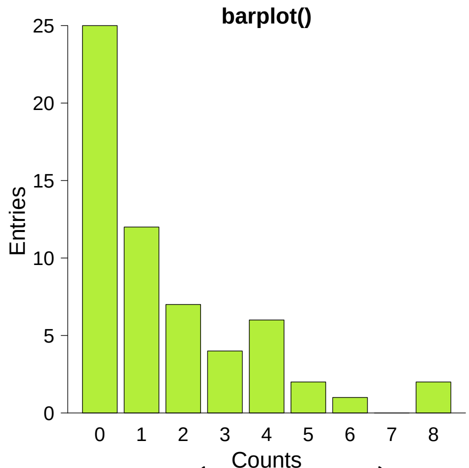
```
counts <- c(25,12,7,4,6,2,1,0,2)

names(counts) <- 0:(length(counts)-1)
str(counts)
%> Named num [1:9] 25 12 7 4 6 2 1 0 2
%> - attr(*, "names")= chr [1:9] "0" "1" "2" "3" ...

names(counts) <- NULL  # names can be easily removed
str(counts)
%> num [1:9] 25 12 7 4 6 2 1 0 2
```



hist(counts, ...)



barplot(counts, ...)

Vector attribute: dimensions

- adding a dim attribute to a vector, changes it behavior to

\- a 2D matrix dim = c(nrow, ncol)

```
v1 <- c(1:20)   ;   v1
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20

class(v1)
%> [1] "integer"

str(v1)
%> int [1:20] 1 2 3 4 5 6 7 8 9 10 ...

%% We transfor the vector to a matrix 4 x 5:
dim(v1) <- c(4,5)

class(v1)
%> [1] "matrix"

str(v1)
%> int [1:4, 1:5] 1 2 3 4 5 6 7 8 9 10 ...

v1
%>       [,1] [,2] [,3] [,4] [,5]
%> [1,]     1    5    9   13   17
%> [2,]     2    6   10   14   18
%> [3,]     3    7   11   15   19
%> [4,]     4    8   12   16   20
```

Vector attribute: dimensions

- adding a dim attribute to a vector, changes it behavior to
- a multi-dimensional array dim = c(dim1, dim2, ... dimn)

```
v1 <- c(1:20)

dim(v1) <- c(2,5,2)

class(v1)
%> [1] "array"

str(v1)
%> int [1:2, 1:5, 1:2] 1 2 3 4 5 6 7 8 9 10 ...

v1
%> , , 1
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    3    5    7    9
%> [2,]    2    4    6    8   10
%>
%> , , 2
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]   11   13   15   17   19
%> [2,]   12   14   16   18   20
```


Matrices and Arrays

- matrices and arrays can be created with the functions matrix() and array

```
v1 <- c(1:20)
matrix(v1, nrow=4, ncol=5)
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    5    9   13   17
%> [2,]    2    6   10   14   18
%> [3,]    3    7   11   15   19
%> [4,]    4    8   12   16   20

array(v1, c(2,5,2))
%> , , 1
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    1    3    5    7    9
%> [2,]    2    4    6    8   10
%>
%> , , 2
%>
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]   11   13   15   17   19
%> [2,]   12   14   16   18   20
```

Vectors and logical subscripts

```
(x <- 0:10)
%> [1]  0  1  2  3  4  5  6  7  8  9 10

sum(x)
%>[1] 55

sum(x<5)
%>[1] 5
```

- the first `sum()` call sums up all the numbers in the vector
- the second call does not return the sum of the values which are lower than five

```
x<5
%> [1] TRUE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

- `x<5` is a vector of logicals, but summing it up R converts logical `TRUE` to 1 and `FALSE` to 0
- we need vector subscripting to perform the desired sum

```
x[x<5]
%> [1] 0 1 2 3 4

sum(x[x<5])
%> [1] 10
```

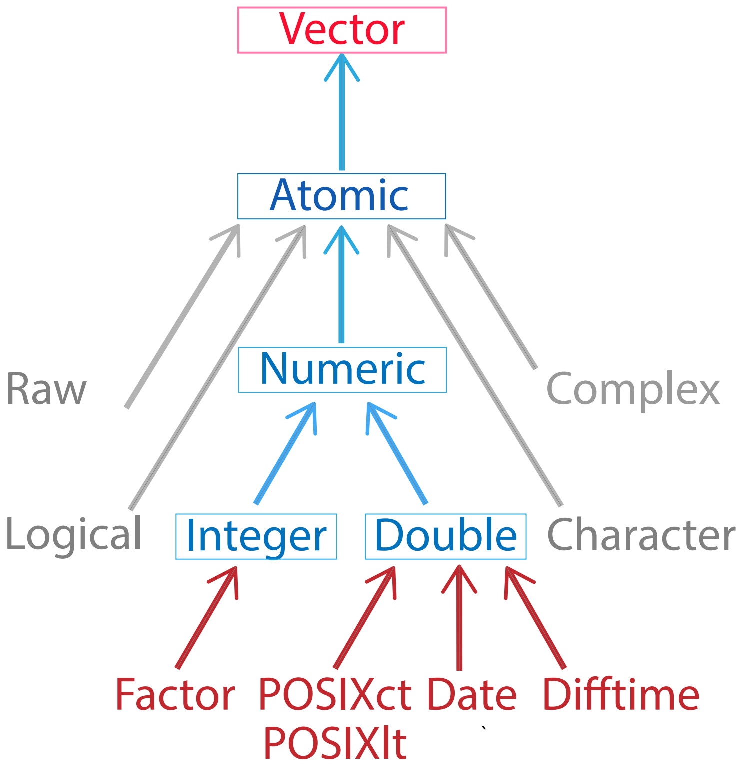

R vector functions

Function	Description
`max(x)`	the maximum value in `x`
`min(x)`	the minimum value in `x`
`sum(x)`	the sum of all values in `x`
`mean(x)`	arithmetic average of the values in `x`
`median(x)`	median value in `x`
`range(x)`	a vector with `min(x)` and `max(x)`
`var(x)`	sample variance of `x`
`cor(x,y)`	correlation between `x` and `y` vectors
`sort(x)`	a sorted version of `x`
`rank(x)`	a vector with the ranks of the `x` values
`order(x)`	a vector with the permutations to sort `x` in asc order
`quantile(x)`	a vector with: minimum, lower quantile, median, upper quantile and maximum of `x`
`cumsum(x)`	a running sum of the vector elements
`cumprod(x)`	a running product of the vector elements
`cummax(x)`	a vector of non-decreasing numbers with the cumulative maxima
`cummin(x)`	a vector of non-decreasing numbers with the cumulative minima
`pmax(x, y, z)`	vector containing the maximum of `x, y` or `z` for each position
`pmin(x, y, z)`	vector containing the minimum of`x, y` or `z` for each position
`colMeans(x)`	column means of a dataframe or matrix
`colSums(x)`	column sums of a dataframe or matrix
`rowMeans(x)`	row means of a dataframe or matrix
`rowSums(x)`	row sums of a dataframe or matrix

S3 Atomic Vectors

S3 atomic vectors

- S3 is the basic object system in R.
- an object is turned into an S3 object with a `class` attribute
- some important S3 vectors used in R are
  - `factor` vectors : used to store categorical data, as a fixed set of levels
  - `Date` vectors, for time object with day resolution
  - `POSIXct`/`POSIXlt` vectors, for time object with second (or sub-second) resolution
  - `difftime` vectors, for storing time durations

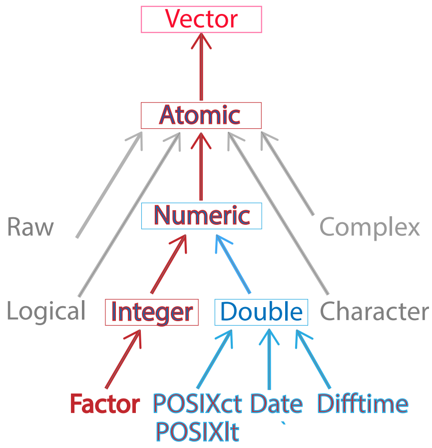


S3 atomic vectors : factors

- a `factor` is a vector that contains only predefined values
- it is used to store categorical data

```
x <- factor(c("a", "b", "b", "c"))
str(x)
%> Factor w/ 3 levels "a","b","c": 1 2 2 3

typeof(x)
%> [1] "integer"
attributes(x)
%> $levels
%> [1] "a" "b" "c"
%>
%> $class
[%> 1] "factor"
```



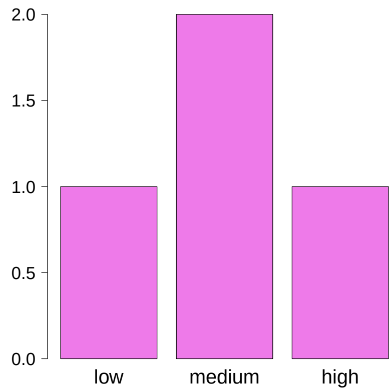

```
coord <- factor(c("Est", "West", "Est", "North"),
                levels = c("North", "Est", "South", "West")) ; coord
%> [1] Est   West  Est   North
%> Levels: North Est South West

> table(coord)
%> coord
%> North   Est South  West
%>     1     2     0     1
```


S3 atomic vectors : ordered factors

- they behave like factors, but the order of the levels is meaningful

```
grade <- ordered(c("high", "low", "medium",
                   "medium"),
                 levels = c("low","medium","high"))
str(grade)
%> Ord.factor w/ 3 levels
%>                 "low"<"medium"<..: 3 1 2 2
summary(grade)
%>    low medium   high
%>      1      2      1
barplot(table(grade), color="orchid2")
```



Note

- in base R factors are encountered very frequently:
- many base R functions (`read.csv()`, `data.frame()`) automatically convert character vectors to factors
- to suppress this behavior use `stringsAsFactors = FALSE`
- factors are built on top of integers, be careful when treating them like strings

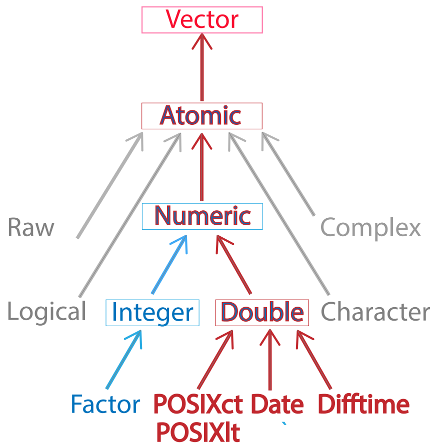
S3 atomic vectors : Dates

Date vectors are built on top of double vectors

```
today <- Sys.Date()  ;  today
%> [1] "2020-03-15"
typeof(today)
%> [1] "double"
class(today)
%> [1] "Date"

yesterday <- as.Date("2020-03-14")
yesterday
%> [1] "2020-03-14"

delta <- today - yesterday  ;  delta
%> Time difference of 1 days
class(delta)
%> [1] "difftime"
```



they are represented as number of days since 1970/01/01

```
days_since_1970_01_01 <- unclass(today)
days_since_1970_01_01
%> [1] 18336
class(days_since_1970_01_01)
%> numeric
```


S3 atomic vectors : Date-times

- baseR provides two ways of storing date-time information:

- `POSIXct`

  `ct` = calendar time (the `time_t` type in C)

- `POSIXlt`

  `lt` = local time (the `struct tm` type in C)

* `POSIXct` vectors are built on top of double vectors, and time is represented as seconds since 1970/01/01

```
now_ct <- as.POSIXct(Sys.time(), tzone="CET")
now_ct
%> [1] "2020-03-15␣14:22:41␣UTC"

r20bday_ct <- as.POSIXct("2020-02-29␣12:00", tzone= "CET")
now_ct - r20bday_ct
%> Time difference of 15.14075 days
```

* the `tzone` attribute controls only how date-time is formatted, not how it is represented

```
structure(now_ct, tzone="Europe/Rome")
%> [1] "2020-03-15␣15:30:53␣CET"
structure(now_ct, tzone="Europe/Moscow")
%> [1] "2020-03-15␣17:30:53␣MSK"
structure(now_ct, tzone="Asia/Chongqing")
%> [1] "2020-03-15␣22:30:53␣CST"
```

Time duration

- durations represent the time difference between two pair of dates or date-times
- they are stored in `difftimes`
- this S3 class has a `unit` attribute that determines how the difference should be interpreted

```
one_week <- as.difftime(1, units="weeks")
attributes(one_week)
%> $class
%> [1] "difftime"
%>
%> $units
%> [1] "weeks"

today <- Sys.time()
next_sunday <- today + one_week

structure(next_sunday, tzone="Europe/Rome")
%> [1] "2020-03-22 16:04:58 CET"

fourty_min <- as.difftime(40, units="mins")
later <- today + fourty_min
later
%> [1] "2020-03-15 15:44:58 UTC"
```


`unique` and `duplicated` for vectors

- with the function `table()` we can inspect how many times each name appears
- the function `unique()` extracts the unique values in a vector, in the order in which the values are encountered in the vector

```
names <- c("John", "John", "Jim", "Anna", "Beatrix", "Anna")
table(names)
%> names
%>     Anna  Beatrix      Jim     John
%>        2        1        1        2

unique(names)
%> [1] "John"    "Jim"     "Anna"    "Beatrix"
```

- the function `duplicated` creates a vector of logical values which is TRUE if that name has already appeared in the vector

```
duplicated(names)
%> [1] FALSE  TRUE FALSE FALSE FALSE  TRUE

names[!duplicated(names)]
%> [1] "John"    "Jim"     "Anna"    "Beatrix"
```

Operating on sets: union, intersect and setdiff

- given two sets, the union() function gives a set with all elements, but counting only once those common to both sets

```
setA <- c ("a", "b", "c", "d", "e")
setB <- c ("d", "e", "f", "g")

union(setA,setB)
%> [1] "a" "b" "c" "d" "e" "f" "g"
```

- intersection() gives ony the elements they have in common

```
intersect(setA, setB)
%> [1] "d" "e"
```

- the difference between the two sets is order-dependent

```
setdiff(setA, setB)
%> [1] "a" "b" "c"
setdiff(setB, setA)
%> [1] "f" "g"

setequal(setA, setA) # compare if the sets are equal
%> [1] TRUE
setequal(setA, setB)
%> [1] FALSE

setA %in% setB
%> [1] FALSE FALSE FALSE  TRUE  TRUE
setA[setA %in% setB]   # equal to intersect(setA, setB)
%> [1] "d" "e"
```