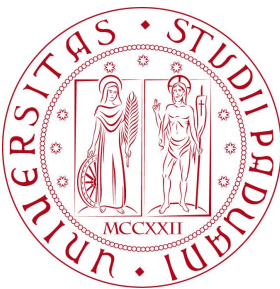


R Lists and Data Frames

Alberto Garfagnini

Università di Padova

AA 2023/2024 - R lecture 4



R internals: variables and objects creation

- We create a vector with three values and assign it to a reference variable, `x`

```
x <- c(1, 2, 3)
```

- we now copy `x` to another variable `y`:

```
y <- x
```

- and modify one element of `y`

```
y[3] <- 4
```

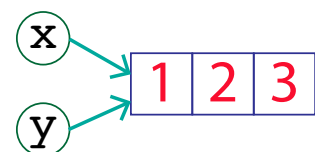
- did we modify also `x`?

No, they refer to two different objects:

```
str(x)
# num [1:3] 1 2 3
str(y)
# num [1:3] 1 2 4
```

- the behavior is called **copy-on-modify**
- all R objects are **immutable**

```
lobstr::obj_addr(x)
"0x55d03cd66fb8"
```



```
lobstr::obj_addr(y)
"0x55d03dbac8c8"
```



The `lobstr` package allows to visualize R data structures: it shows memory location and size of objects.

URL: <https://github.com/r-lib/lobstr>

R Lists

- **List** are important objects in R each element of the list can be of different type
- from the technical point of view: each element of a list is of the same type: it is a reference to another R object
- building a list:

```
l1 <- list( 1:3,
            "one_list_element",
            rep(c(T,F,T), 1:3),
            c(3.5, 4, 6.2, -1.75) )

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

# [[2]]
# [1] "one list element"

# [[3]]
# [1] TRUE FALSE FALSE TRUE TRUE TRUE

# [[4]]
# [1] 3.50 4.00 6.20 -1.75

str(l1)
# List of 4
# $ : int [1:3] 1 2 3
# $ : chr "one list element"
# $ : logi [1:6] TRUE FALSE FALSE TRUE TRUE TRUE
# $ : num [1:4] 3.5 4 6.2 -1.75
```

Indexing Lists

- subscripts on **vectors**, **matrices**, **arrays**, and **dataframes** have one set of square brackets
- subscripts on **lists** use double square brackets:

```
l1 <- list( 1:3, "list_element",
            c(TRUE, FALSE, FALSE),
            c(3.5, 4, 6.2, -1.75) )
```

- double square brackets **[[]]** return the element in the list with its type

```
l1[[4]]
# [1] 3.50 4.00 6.20 -1.75

storage.mode(l1[[4]])
# [1] "double"

str( l1[[4]] )
# num [1:4] 3.5 4 6.2 -1.75

l1[[4]][3]
# [1] 6.2
```

→ single square brackets **[]** always return a list

```
str( l1[4] )
# List of 1
# $ : num [1:4] 3.5 4 6.2 -1.75
```

R named Lists

- list elements can be given a name at creation time:

```
l1 <- list(index = 1:3, text = "list_element",  
           test = c(TRUE, FALSE, FALSE),  
           speed = c(3.5, 4, 6.2, -1.75) )
```

```
str(l1)  
# List of 4  
# $ index: int [1:3] 1 2 3  
# $ text : chr "list element"  
# $ test : logi [1:3] TRUE FALSE FALSE  
# $ speed: num [1:4] 3.5 4 6.2 -1.75
```

- and this allows to extract elements by name
- these are all equivalent:

```
l1$speed  
# [1] 3.50 4.00 6.20 -1.75  
  
l1[["speed"]]  
# [1] 3.50 4.00 6.20 -1.75  
  
l1[[4]]  
# [1] 3.50 4.00 6.20 -1.75
```

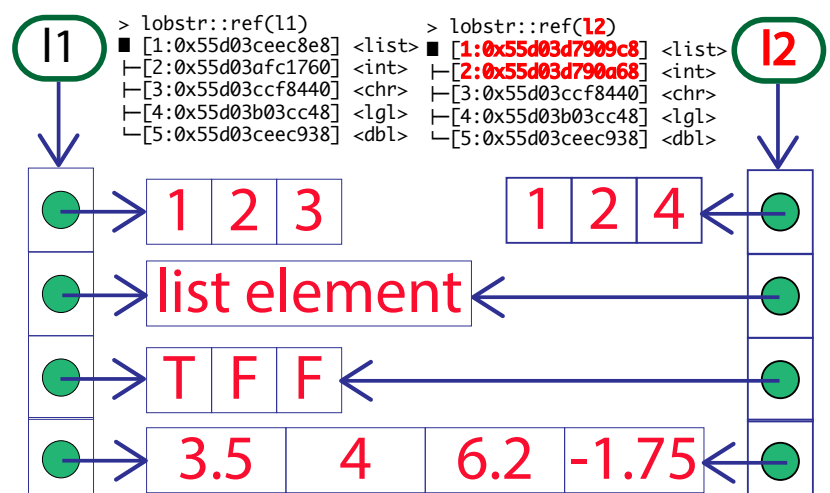
R Lists storage

- Lists** are an evolution of atomic vectors: each element can be of any type
- from the technical point of view: each element of a list is of the same type: it is a reference to another R object
- building a list:

```
l1 <- list( 1:3,  
           "list_element",  
           c(TRUE, FALSE, FALSE),  
           c(3.5, 4, 6.2, -1.75)  
)  
typeof(l1)  
# [1] "list"
```

- we copy to a new list and modify one element

```
l2 <- l1  
l2[[1]] <- c(1L, 2L, 4L)
```



R data.frames

- two important S3 vectors built on top of lists are `data.frames`, `tibbles`, and `data.table`,
- a data frame is like a matrix, with a 2-dim rows-and-columns structure
- it's a **named list of vectors**, with attributes for columns and rows names, (`names`, `row.names`), belonging to the `data.frame` class
- technically, a data frame is a list with all equal length vectors

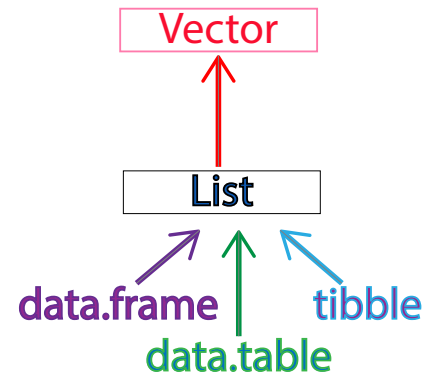
```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
# [1] "list"
```

```
attributes(df1)
# $names
# [1] "x" "y"

# $class
# [1] "data.frame"

# $row.names
# [1] 1 2 3
```

```
str(df1)
# 'data.frame':      3 obs. of  2 variables:
#  $ x: int  1 2 3
#  $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```



R data.frames : examples

- we have a table with the results of two exams for the student of an hipotetical course, and we want to **import them** in a `data.frame`

Exam ₁	Exam ₂	Channel
27	25	A-L
28	30	M-Z
...		
27	27	M-Z
25	28	A-L

```
exam1 <- c(27,28,24,24,30,26,23,23,24,28,27,25)
exam2 <- c(25,30,26,24,30,30,25,25,30,28,27,28)
channel <- c("AL","MZ","MZ","MZ","MZ","MZ","MZ","MZ","MZ","AL","AL","MZ","AL")
```

```
dc <- data.frame(exam1, exam2, channel)
head(dc, n=2) # extract the first two lines of the data frame
#   exam1 exam2 channel
# 1    27    25     AL
# 2    28    30     MZ
```

```
dc1 <- data.frame(exam1, exam2, channel,
                  stringsAsFactors = TRUE)
```

```
str(dc1)
# 'data.frame': 12 obs. of  3 variables:
#  $ exam1 : num  27 28 24 24 30 26 23 23 24 28 ...
#  $ exam2 : num  25 30 26 24 30 30 25 25 30 28 ...
#  $ channel: Factor w/ 2 levels "AL","MZ": 1 2 2 2 2 2 2 2 2 1 1 ...
```

From R 4.0
`stringsAsFactors = FALSE`
by default

- Data frames are list of vectors, therefore **copy-on-modify** has important consequences

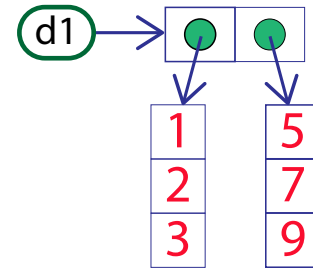
```
d1 <- data.frame(x = c(1, 2, 3),
                 y = c(5, 7, 9))
```

```
d1
#   x y
# 1 1 5
# 2 2 7
# 3 3 9
```

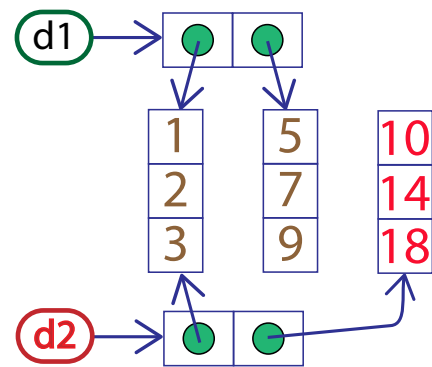
- if we **modify a column** → only the reference to the new column will be updated

```
d2 <- d1
d2[, 2] <- d2[, 2] * 2
d2
#   x   y
# 1 1 10
# 2 2 14
# 3 3 18
```

```
> lobstr::ref(d1)
# [1:0x55905d24e9e8] <df[,2]>
└─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e564e68] <dbl>
```



```
> lobstr::ref(d1)
# [1:0x55905d24e9e8] <df[,2]>
└─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e564e68] <dbl>
```



```
> lobstr::ref(d2)
# [1:0x55905c9f1628] <df[,2]>
└─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e92cd78] <dbl>
```

R `data.frames` objects creation

(2)

- Data frames are list of vectors, therefore **copy-on-modify** has important consequences

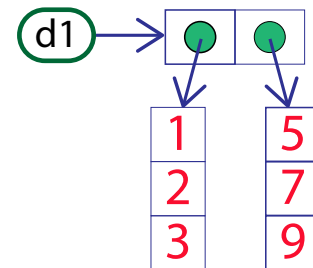
```
d1 <- data.frame(x = c(1, 2, 3),
                 y = c(5, 7, 9))
```

```
d1
#   x y
# 1 1 5
# 2 2 7
# 3 3 9
```

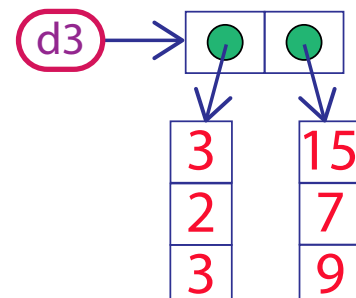
- but if **any row is modified** → every column is **modified** because every column must be copied

```
d3 <- d1
d3[1, ] <- d3[1, ] * 3
d3
#   x   y
# 1 3 15
# 2 2  7
# 3 3  9
```

```
> lobstr::ref(d1)
# [1:0x55905d24e9e8] <df[,2]>
└─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e564e68] <dbl>
```

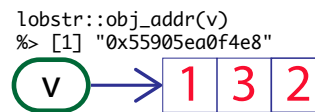


```
> lobstr::ref(d3)
# [1:0x55905e6f1058] <df[,2]>
└─x = [2:0x55905ea0c238] <dbl>
└─y = [3:0x55905ea0c1e8] <dbl>
```

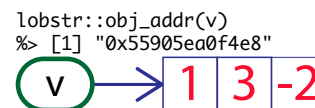


- Modifying an R object usually **creates a copy**
- but there are **2 exceptions**:
 - **objects with single binding** get a special performance optimization
 - **environments**, a special type of object, **are always modified in place**

```
v <- c(1, 3, 2)
lobstr::obj_addr(v)
# [1] "0x55905ea0f4e8"
```



```
v[3] <- -2
lobstr::obj_addr(v)
# [1] "0x55905ea0f4e8"
```



- but **it is very difficult to predict** when R applies this **optimization**
- concerning object binding, **R only counts 0, 1 or MANY**
- it means that if an object has 2 bindings (i.e. many), and one gets deleted, the reference does not go back to 1 (many - 1 = many)
- when a function is called, it makes a reference to the object → it is very difficult to predict whether or not a copy will occur
- cfr: <https://developer.r-project.org/Refcnt.html>

Accessing `data.frames` elements

- a data frame is a list, therefore we can access them via **component index** value `[[j]]` or via **component names**

```
str(dc1)
'data.frame':   12 obs. of  3 variables:
 $ exam1  : num  27 28 24 24 30 26 23 23 24 28 ...
 $ exam2  : num  25 30 26 24 30 30 25 25 30 28 ...
 $ channel: Factor w/ 2 levels "AL","MZ": 1 2 2 2 2 2 2 2 1 1 ...
```

```
dc1[[1]] # access by component index
# [1] 27 28 24 24 30 26 23 23 24 28 27 25
```

```
dc1$exam1 # access by component name
# [1] 27 28 24 24 30 26 23 23 24 28 27 25
```

- but a **data.frame** can be treated in a matrix-like fashion, as well

```
dc1[,1] # select column 1
# [1] 27 28 24 24 30 26 23 23 24 28 27 25
```

```
dc1[1,1] # and access the single element, as well
# [1] 27
```

- `dc1[2:4,]` # Select only rows 2:4

```
# exam1 exam2 channel
# 2      28      30      MZ
# 3      24      26      MZ
# 4      24      24      MZ
```

```
dc1[-(2:10),] # drop rows 2:10
```

```
#      exam1 exam2 channel
# 1         27      25      AL
# 11        27      27      MZ
# 12        25      28      AL
```

- with the `sample` function , data can be selected at random

```
dc1[sample(1:12,3),] # select 3 rows at random
```

```
#      exam1 exam2 channel
# 10        28      28      AL
# 5         30      30      MZ
# 1         27      25      AL
```

```
dc1[sample(1:12,3),] # select 3 rows at random
```

```
#      exam1 exam2 channel
# 2         28      30      MZ
# 4         24      24      MZ
# 7         23      25      MZ
```

Advanced data frames : data selection

- suppose we want to extract all columns that contain numbers, rather than characters or logicals, from a data frame

```
dc[,sapply(dc1,is.numeric)]
```

```
#      exam1 exam2
# 1         27      25
# 2         28      30
# 3         24      26
# 4         24      24
# 5         30      30
# 6         26      30
# 7         23      25
# 8         23      25
# 9         24      30
# 10        28      28
# 11        27      27
# 12        25      28
```

```
dc <- data.frame(exam1, exam2, channel )
str(dc)
'data.frame':  12 obs. of  3 variables:
 $ exam1 : num  27 28 ...
 $ exam2 : num  25 30 ...
 $ channel: Fact w/ 2 lvl "AL","MZ": 1 2 ...
```

- and now we want to get only factors (and remove numerics)

```
dc[,sapply(dc,is.factor)]
```

```
# [1] AL MZ MZ MZ MZ MZ MZ MZ AL AL MZ AL
# Levels: AL MZ
```

Summary of data selection in data frames

- given a data frame called `data`, we assume `n` is a row number, and `m` is one of the column.
- the syntax `[n,]` selects all the columns given row `n`, while `[,m]` selects all the rows with column `m`

command	meaning
<code>data[n,]</code>	select all of the columns from row <code>n</code> of the data frame
<code>data[-n,]</code>	drop the whole of row <code>n</code> from the data frame
<code>data[1:n,]</code>	select all of the columns from rows 1 to <code>n</code> of the data frame
<code>data[-(1:n),]</code>	drop all of the columns from rows 1 to <code>n</code> of the data frame
<code>data[c(i,j,k),]</code>	select all of the columns from rows <code>i</code> , <code>j</code> , and <code>k</code> of the data frame
<code>data[x > y,]</code>	use a logical test ($x > y$) to select all columns from certain rows
<code>data[,m]</code>	select all of the rows from column <code>m</code> of the data frame
<code>data[, -m]</code>	drop the whole of column <code>m</code> from the data frame
<code>data[, 1:m]</code>	select all of the rows from columns 1 to <code>m</code> of the data frame
<code>data[, -(1:m)]</code>	drop all of the rows from columns 1 to <code>m</code> of the data frame
<code>data[, c(i,j,k)]</code>	select all of the rows from columns <code>i</code> , <code>j</code> , and <code>k</code> of the data frame
<code>data[, x > y]</code>	use a logical test ($x > y$) to select all rows from certain columns
<code>data[, c(1:m,i,j,k)]</code>	add duplicate copies of columns <code>i</code> , <code>j</code> , and <code>k</code> to the data frame
<code>data[x > y, a != b]</code>	extract certain rows ($x > y$) and certain columns ($a \neq b$)
<code>data[c(1:n,i,j,k),]</code>	add duplicate copies of rows <code>i</code> , <code>j</code> , and <code>k</code> to the data frame

Subsetting atomic vectors (1)

```
x <- c(2.1, 4, 6.7, 1.75)
```

- **positive integers** return elements at a specified position

```
x[c(1,3)]  
# [1] 2.1 6.7
```

```
% Duplicate indices will duplicate values  
x[c(1,1,3,3)]  
# [1] 2.1 2.1 6.7 6.7
```

```
% Real numbers are truncated to integers  
x[sort(x)]  
# [1] 2.10 4.00 1.75 NA
```

- **negative integers** exclude elements

```
x[-c(1,3)]  
# [1] 4.00 1.75
```

```
% NB negative and positive ints cannot be mixed  
x[c(-1,3)]  
# Error in x[c(-1, 3)]: only 0's may be mixed with negative subscripts
```



```
x <- c(2.1, 4, 6.7, 1.75)
```

- **logical vectors** select elements where the logical value is **TRUE**

```
x[c(T, T, F, T)]
# [1] 2.10 4.00 1.75
```

```
x[x>2]
# [1] 2.1 4.0 6.7
```

- if in `x[sel]`, `length(sel) != length(x)` the **recycling rules** are used: the shorter vector is recycled to the length of the longer

```
x[c(TRUE, FALSE)]
# [1] 2.1 6.7
```

```
%# is equivalent to:
x[c(TRUE, FALSE, TRUE, FALSE)]
# [1] 2.1 6.7
```

- **nothing** returns the original vector

```
x[]
# [1] 2.10 4.00 6.70 1.75
```

Subsetting atomic vectors

```
x <- c(2.1, 4, 6.7, 1.75)
```

- **zero** returns a zero-length vector (it can be helpful to generate test data)

```
x[0]
# numeric(0)
```

- **named vectors** can be accessed with **character vectors**

```
y <- setNames(x, LETTERS[1:length(x)])
y
#      A      B      C      D
# 2.10 4.00 6.70 1.75
y["A"]
#      A
# 2.1
```

```
y[c('A', 'A', 'D')]
#      A      A      D
# 2.10 2.10 1.75
```

- **WARNING:** subsetting with factors will use the underlying integer vector, not the character levels. → **Avoid subsetting with factors**

```
y[factor("B")]
#      A
# 2.1
```

- subsetting a matrix or a list works in a similar way as subsetting atomic vectors

```
S <- matrix(1:9, nrow = 3)
# [1,] 1 4 7
# [2,] 2 5 8
# [3,] 3 6 9
```

- using `[]` always returns a list
- `[[]]` and `$` allows to pull out elements from the list
- the common rule to subset a matrix (2D) and an array (nD, $n > 2$) is to supply a 1D vector for each dimension, separated by a comma
- blank subsetting allows to keep all data for the corresponding dimension

```
%# Get rows 1 and 3 and all columns
S[c(1,3), ]
#      [,1] [,2] [,3]
# [1,]    1    4    7
# [2,]    3    6    9

colnames(S) <- c("S1", "S2", "S3")
S[c(T, F, T), c("S1", "S3")]
#      S1 S3
# [1,]  1  7
# [2,]  3  9
```

Subsetting matrices

- matrices and arrays are just vectors with special attributes, therefore they can be subset with a single vector, as if they were a 1D vector

```
v <- outer(1:5, 1:5, FUN="paste", sep=",")
v
#      [,1] [,2] [,3] [,4] [,5]
# [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
# [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
# [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
# [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
# [5,] "5,1" "5,2" "5,3" "5,4" "5,5"

v[seq(3, 23, 5)]
# [1] "3,1" "3,2" "3,3" "3,4" "3,5"
```

- to preserve the original matrix dimension, use `drop = FALSE`

```
(S <- matrix(1:6, nrow = 2))
#      [,1] [,2] [,3]
# [1,]    1    3    5
# [2,]    2    4    6

S[1, ]
# [1] 1 3 5

S[1, , drop = FALSE]
#      [,1] [,2] [,3]
# [1,]    1    3    5
```

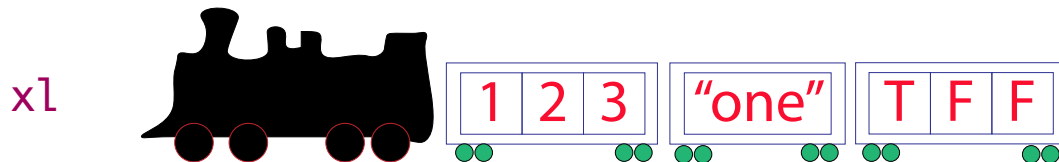
- there are two other subsetting operators:
 - `[]` is used to extract single items
 - `$` is used as a shorthand: `x$y` stands for `x[["y"]]`
- `[]` is most important while working with lists: subsetting a list with single `[]` always returns a smaller list

If list `xl` is a train carrying objects, then `xl[[5]]` is the object in car 5; `xl[4:6]` is a train of cars 4-6

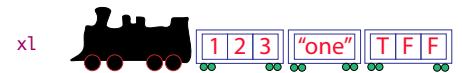
<https://twitter.com/RLangTip/status/268375867468681216>

- with this metaphor let's build a list

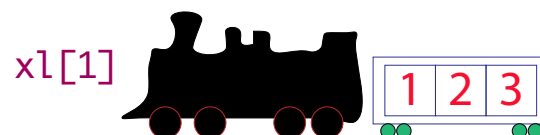
```
xl <- list(1:3, "one", c(T,F,F))
```



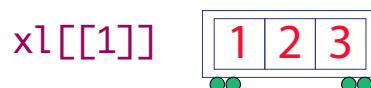
Selecting Lists a single elements



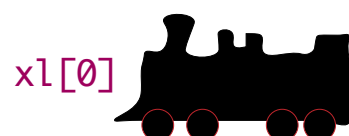
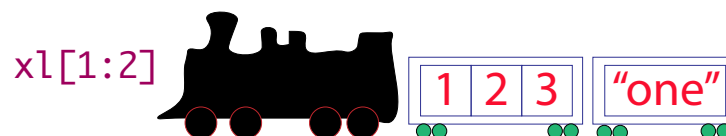
- two options are available when extracting a single element:
 - create a smaller train, with fewer cars (using `[]`)



- or extract the content of a particular car (with `[]`)



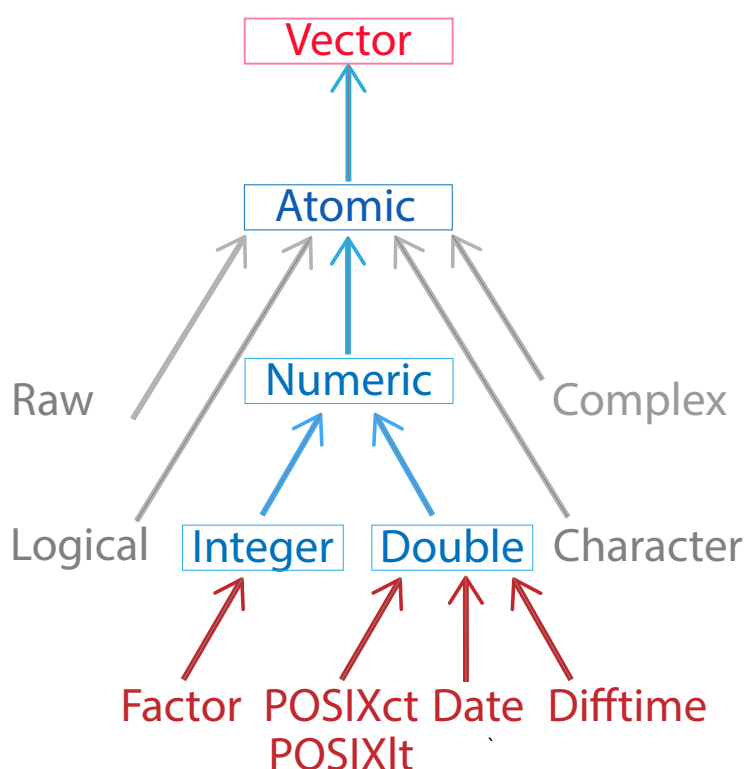
- extracting multiple (or zero) elements, we have to build a smaller train



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

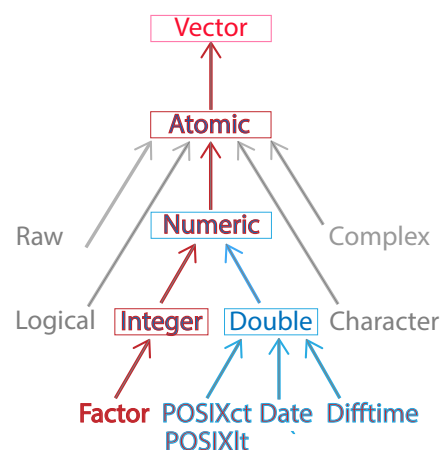
- a **factor** is a vector that contains only predefined values
- it is used to store categorical data

```
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
```

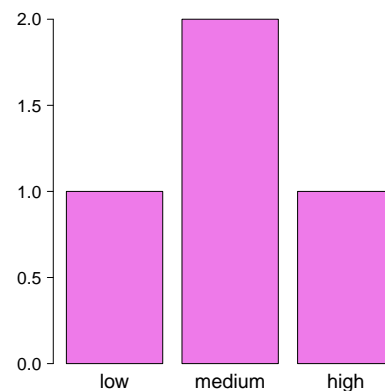
```
table(coord)
# coord
# North Est South West
#      1    2     0    1
```



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

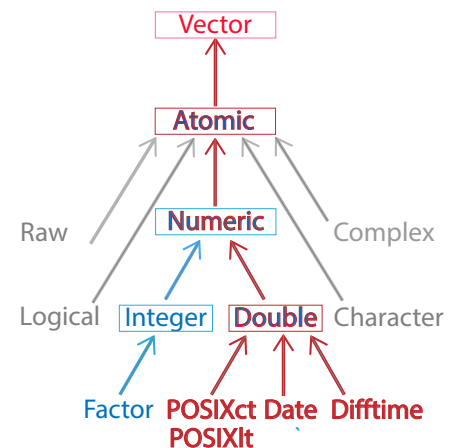
```
today <- Sys.Date() ; today
# [1] "2020-03-15"
typeof(today)
# [1] "double"
class(today)
# [1] "Date"

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(), tzzone="CET")
now_ct
# [1] "2022-03-15 14:22:41 UTC"
```

```
r20bday_ct <- as.POSIXct("2022-02-29 12:00", tzzone="CET")
now_ct - r20bday_ct
# Time difference of 15.14075 days
```

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

```
structure(now_ct, tzzone="Europe/Rome")
# [1] "2022-03-15 15:30:53 CET"
structure(now_ct, tzzone="Europe/Moscow")
# [1] "2022-03-15 17:30:53 MSK"
structure(now_ct, tzzone="Asia/Chongqing")
# [1] "2022-03-15 22:30:53 CST"
```

- durations represent the time difference between two pair of dates or date-times
- they are stored in `difftime`
- 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, tzzone="Europe/Rome")
# [1] "2022-03-22 16:04:58 CET"

fourty_min <- as.difftime(40, units="mins")
later <- today + fourty_min
later
# [1] "2022-03-15 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 only 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"
```