R Data Frames and Lists

Alberto Garfagnini

Università di Padova

AA 2021/2022 - 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`

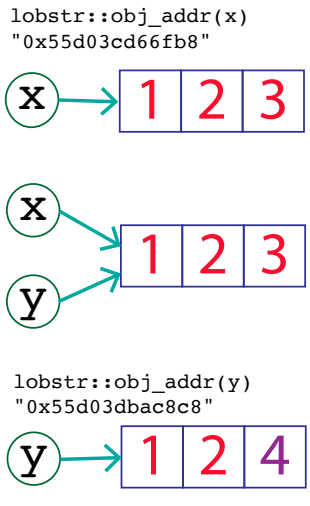
```
y[3] <- 4
```

- 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`



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

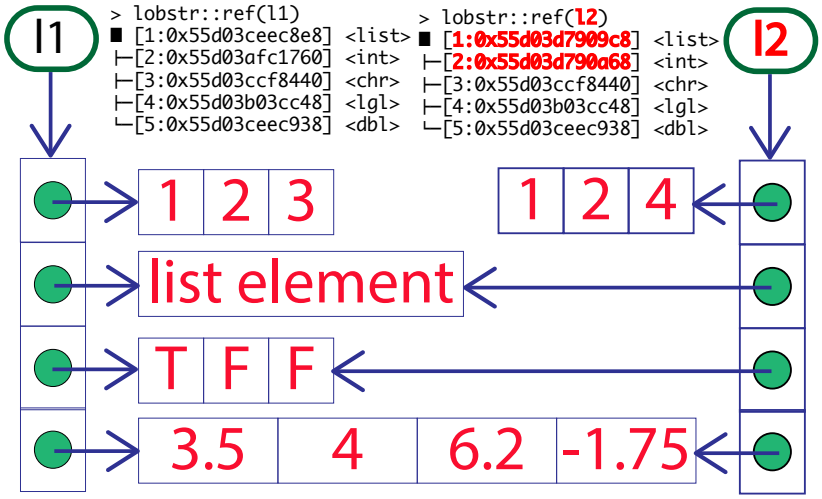
- Lists are an evolution of atomic vectors: each element can be of any type
- from the technical point of view: each element of a list is of the same type: it is a reference to another R object
- building a list:

```
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 matrices (1)

- a matrix is a 2-dimensional object
- the first way of creating a matrix is by calling the `matrix()` object constructor

```
X <- matrix(c(1,0,0,0,1,0,0,0,1), nrow=3)   ;   X
#      [,1] [,2] [,3]
# [1,]    1    0    0
# [2,]    0    1    0
# [3,]    0    0    1

class(X)
# [1] "matrix"
attributes(X)
# $dim
# [1] 3 3
str(X)
# num [1:3, 1:3] 1 0 0 0 1 0 0 0 1
```

- another way is to transform a vector in a matrix: data can be arranged by rows (`byrow=T`) or columns (`byrow=F`)

```
vct <- c(1,2,3,4,4,3,2,1)
V <- matrix(vct,byrow=T,nrow=2)
V
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    3    4
# [2,]    4    3    2    1
```

```
V <- matrix(vct,byrow=F,nrow=2)
V
#      [,1] [,2] [,3] [,4]
# [1,]    1    3    4    2
# [2,]    2    4    3    1
```

R matrices (2)

- another possibility is to convert the vector to a matrix by specifying the new dimensions (rows and columns), using the `dim` function

```
vct <- c(1,2,3,4,4,3,2,1)
vct
# [1] 1 2 3 4 4 3 2 1

dim(vct) <- c(4,2)
is.matrix(vct)
# [1] TRUE

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

- we can then transform the matrix:

```
tvct <- t(vct) # transpose the matrix
tvct
#      [,1] [,2] [,3] [,4]
# [1,]    1    2    3    4
# [2,]    4    3    2    1
```


Accessing or operating on matrix rows or columns

- Let's create a matrix with $n = 20$ entries sampled from a Poisson distribution with $\lambda = 1.5$

```
X <- matrix(rpois(n=20,lambda=1.5), nrow=4)
X
#      [,1] [,2] [,3] [,4] [,5]
# [1,]    1    1    1    2    4
# [2,]    1    1    3    3    2
# [3,]    1    3    5    0    1
# [4,]    2    1    1    2    2

X[3,3] # return element in row 3 and column 3
# [1] 5
X[4,] # return row 4
# [1] 2 1 1 2 2
X[,5] # return column 5
# [1] 4 2 1 2
```

- there are special functions for calculating summary statistics on a matrix:

```
rowSums(X)         # use colSums(X) for columns
# [1]  9 10 10  8
rowMeans(X)        # use colMeans(X) for columns
# [1] 1.8 2.0 2.0 1.6
```

Adding rows and columns to a matrix

- given a matrix, we would like to add a row, at the bottom, showing the column means, and a column at the right showing the row variances:

```
vct <- matrix(c(1,0,2,5,1,1,3,1,3,1,0,2,1,0,2,1), byrow=T, nrow=4)
vct
#      [,1] [,2] [,3] [,4]
# [1,]    1    0    2    5
# [2,]    1    1    3    1
# [3,]    3    1    0    2
# [4,]    1    0    2    1

vct <- rbind(vct, apply(vct, 2, mean))
vct <- cbind(vct, apply(vct, 1, var))

colnames(vct) <- c(1:4, "variance")
rownames(vct) <- c(1:4, "mean")

vct
#         1   2    3    4  variance
# 1     1.0 0.0 2.00 5.00 4.6666667
# 2     1.0 1.0 3.00 1.00 1.0000000
# 3     3.0 1.0 0.00 2.00 1.6666667
# 4     1.0 0.0 2.00 1.00 0.6666667
# mean  1.5 0.5 1.75 2.25 0.5416667
```


R Data frames

- two important S3 vectors built on top of lists are `data frames` and `tibbles`
- 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

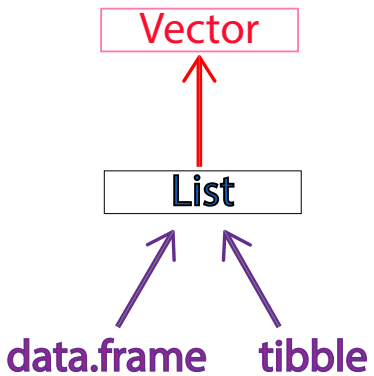
```
df1 <- data.frame(x = 1:3, y = letters[1:3])
typeof(df1)
# [1] "list"


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`

$Exam_1$	$Exam_2$	Gender
27	25	M
28	30	F
...		
27	27	M
25	28	F

```
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)
gender <- c("M","F","M","M","M","M","M","M","F","F","M","F")

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

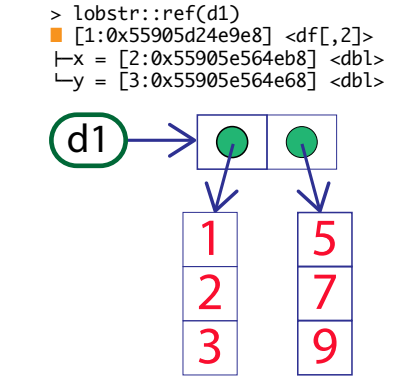
From R 4.0 `stringsAsFactors = FALSE` by default

```
dc1 <- data.frame(exam1, exam2, gender,
                  stringsAsFactors = FALSE)
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 ...
# $ gender: chr  "M" "F" "M" "M" ...
```


R Data Frames objects creation (1)

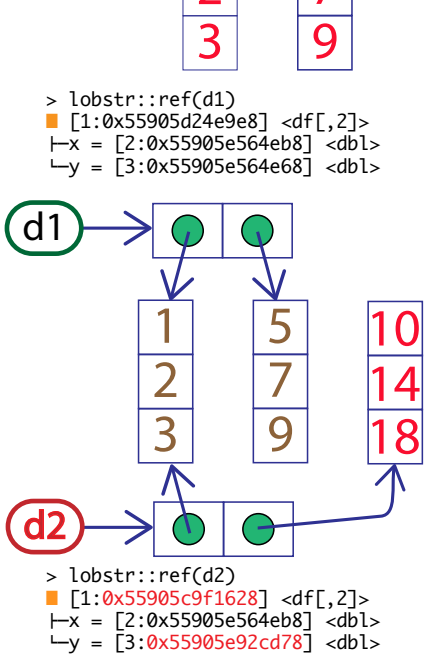
- Data frames are list of vectors, therefore copy-on-modify has important consequences

```
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
```

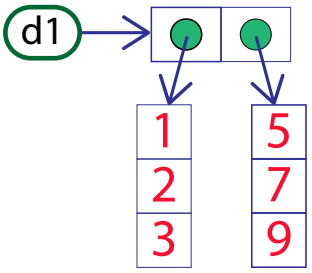


R Data Frames objects creation (2)

- 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
```

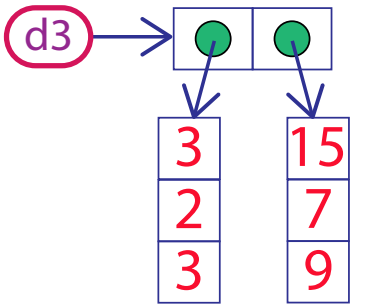
```
> lobstr::ref(d1)
■ [1:0x55905d24e9e8] <df[,2]>
├─x = [2:0x55905e564eb8] <dbl>
└─y = [3:0x55905e564e68] <dbl>
```



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

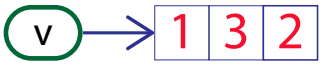


Modify-in-place

- Modifying an R object usually creats 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

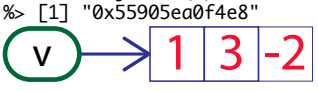
```
v <- c(1, 3, 2)
lobstr::obj_addr(v)
# [1] "0x55905ea0f4e8"
```

```
lobstr::obj_addr(v)
%> [1] "0x55905ea0f4e8"
```




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

```
lobstr::obj_addr(v)
%> [1] "0x55905ea0f4e8"
```



- but it is very difficult to predicts 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 funtion is called, it makes a reference to the object ➜ it is very difficult to predict weather 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(dc)
# '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 ...
#  $ gender: Factor w/ 2 levels "F","M": 2 1 2 2 2 2 2 2 1 1 ...

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

dc$exam1 # access by component name
#  [1] 27 28 24 24 30 26 23 23 24 28 27 25
# Levels: F M
```

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

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

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


Advanced data frames : data selection (1)

-
```
dc[2:4,]  # Select only rows 2:4
#   exam1 exam2 gender
# 2    28    30      F
# 3    24    26      M
# 4    24    24      M

dc[-(2:10),] # drop rows 2:10
#    exam1 exam2 gender
# 1     27    25      M
# 11    27    27      M
# 12    25    28      F
```

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

```
dc[sample(1:12,3),]  # select 3 rows at random
#   exam1 exam2 gender
# 8    23    25      M
# 9    24    30      F
# 6    26    30      M

dc[sample(1:12,3),]  # select 3 rows at random
#    exam1 exam2 gender
# 1     27    25      M
# 10    28    28      F
# 2     28    30      F
```

Advanced `data frames` : data selection (2)

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

```
dc[,sapply(dc,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, gender)
str(dc)
'data.frame':  12 obs. of  3 variables:
$ exam1 : num  27 28 ...
$ exam2 : num  25 30 ...
$ gender: Fact w/ 2 levels "F","M": 2 1
```

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

```
dc[,sapply(dc,is.factor)]
#  [1] M F M M M M M M F F M F
# Levels: F M
```


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

The apply() collection functions

- are used to perform operations on all the elements of a complex object (vector, list, data.frame, ...) avoiding the use of loops
- apply() is the most basic and can be used over a matrix or array
- the following functions are members of the same family:
- lapply(), sapply(), tapply() and mapply()

Function: apply(X, MARGIN, FUN)

```
with

  X: an array or matrix

  MARGIN:  a value or range between 1 and 2
           to define where to apply the function:
  MARGIN=1: the manipulation is performed on rows
  MARGIN=2: the manipulation is performed on columns
  MARGIN=c(1,2) the manipulation is performed
                on rows and columns

  FUN: which function to apply.
       Built functions like mean, median, sum, min, max
       and user-defined functions can be applied
```


apply() example on data.frame

Function: apply(X, MARGIN, FUN)

MARGIN=1: manipulate rows

MARGIN=2: manipulate columns

```
df <- data.frame(exam1 = c(27,28,24,24,30,26),
                 exam2 = c(25,30,26,24,30,30),
                 labrep = c(29,30,29,27,29,30))
str(df)
# 'data.frame': 6 obs. of  3 variables:
#  $ exam1 : num  27 28 24 24 30 26
#  $ exam2 : num  25 30 26 24 30 30
#  $ labrep: num  29 30 29 27 29 30
```

we compute the average mark over exam2 and labrep:

```
row_avg <- apply(X = df[, 2:3], MARGIN = 1, FUN = mean)

# [1] 27.0 30.0 27.5 25.5 29.5 30.0
```

and the mean mark over exam2 and labrep:

```
exam_avg <- apply(X = df[, 2:3], MARGIN = 2, FUN = mean)

# exam2 labrep
#  27.5   29.0
```

lapply() example on list

Function: lapply(X, FUN, ...)

```
plants <- list(height = runif(10, min = 10, max = 20),
               mass = runif(10, min = 5, max = 10),
               flowers = sample(1:10, 10))
str(plants)
# List of 3
#  $ height : num [1:10] 16.8 11.7 13.8 20 15.1 ...
#  $ mass   : num [1:10] 7.53 8.53 9.36 9.76 9.67 ...
#  $ flowers: int [1:10] 9 2 8 5 4 10 6 3 1 7
```

we need to compute the average for each list element

```
lapply(plants, mean)
# $height
# [1] 15.63694
# $mass
# [1] 8.008216
# $flowers
# [1] 5.5
```

the output of lapply() is also a list

with unlist() we can "cast-it-down" to a simple named vector

```
unlist(lapply(plants, mean))
#    height      mass   flowers
# 15.636939  8.008216  5.500000
```


apply(), sapply() and lapply()

- it is also possible to apply an anonymous, user defined, function

```
apply(Y, 1, function(x) x^2+x)  # compute x^2 + x for each element
#     [,1] [,2] [,3] [,4]
# [1,]   42   42   30   12
# [2,]   30   12    6   20
# [3,]   20   12   12   20
# [4,]    6   30   20   12
# [5,]   30   20   20   42
```

- in case you need to apply a function to a vector, rather than to the margin of a matrix, use sapply()

```
sapply(12:14, seq) # generate a list of seq, from 1:12 to 1:14
# [[1]]
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12
#
# [[2]]
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13
#
# [[3]]
#  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

apply() short summary

Overview

- `lapply()` returns a list
  - its may require second step collapsing list
- `sapply()` is a simplified version
  - it returns, in general, a vector
  - it is more specific, and less general
- more specialized functions exist:
  - `vapply()` can specify a return object
  - `mapply()` can use multiple arguments
  - `rapply()` can be used recursively

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
```

Subsetting atomic vectors (2)

```
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 (3)

```
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 matrices (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 (2)

- 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
```

Selecting a single element (1)

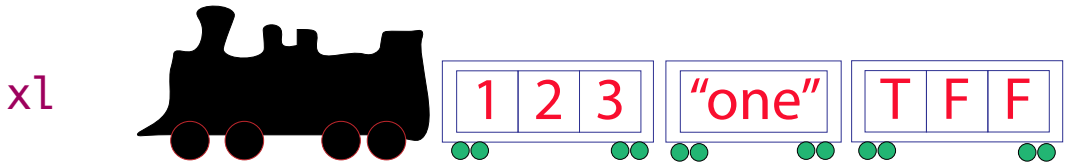
- there are two other subsetting operators:
  - `[[]]` is used to extract single items
  - `$` is used as a shorthand: `x$y` stands for `x[["y"]]`
- `[[]]` is most important while working with lists: subsetting a list with single `[]` always returns a smaller list

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

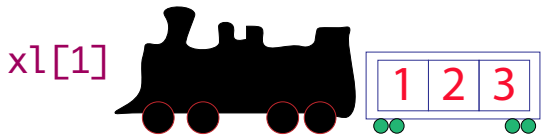
- with this metaphor let's build a list

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

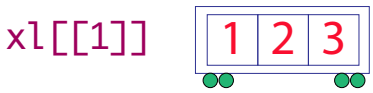


Selecting a single element

- 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