R data types: Lists

Alberto Garfagnini

Università di Padova

R lecture 3

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`

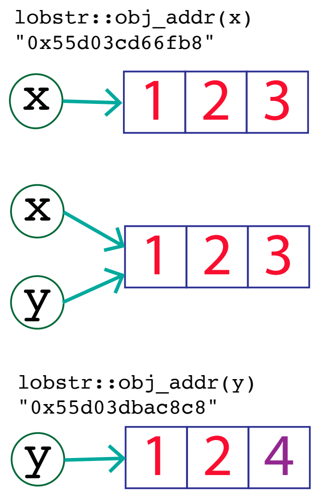
```
y[3] <- 4
```

- 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

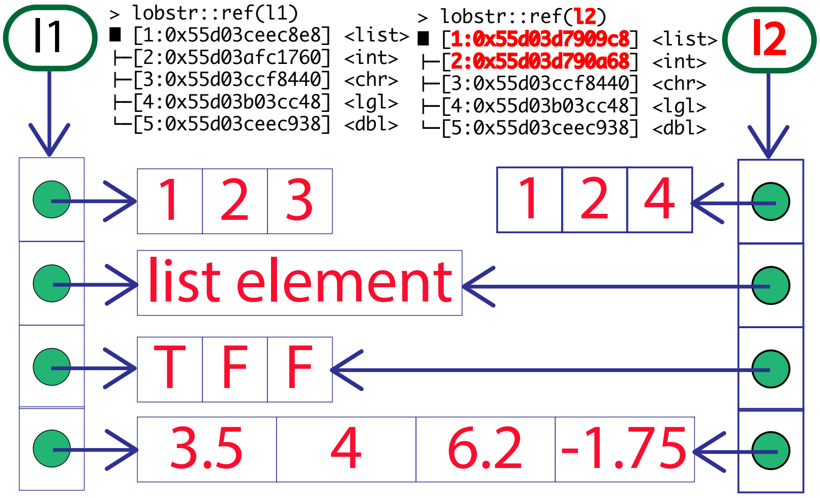
- Lists are an evolution of atomic vectors: each element can be of any type
- from the technical point of view: each element of a list is of the same type: it is a reference to another R object
- building a list:

```
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

- 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

- 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
```


The apply() collection functions

- are used to apply operations on a 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

apply()

- usage:

```
         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
```

The apply() collection functions

- sapply() takes a vector or list object and returns an object of the same type.

sapply()

- usage:

```
        sapply(X, FUN)

Arguments:
  X: A vector or an object
  FUN: Function applied to each element of X

x <- 1:10
apply(x, 1, sqrt)
#> Error in apply(x, 1, sqrt) : dim(X) must have a positive length

sapply(x, sqrt)
 [1] 1.000000 1.414214 1.732051 2.000000 2.236068
 [6] 2.449490 2.645751 2.828427 3.000000 3.162278
```


The apply() collection functions

- lapply() perfoms operations on list objects and returns a list object with the same length of the original set
- lappy() applies a function to each element of the list
- lapply() takes list, vector or data frame as input and returns a list

lapply()

- usage:

```
        lapply(X, FUN)

Arguments:

X: A vector or an object
FUN: Function applied to each element of X

box <- c("Orange", "CHErrry", "APPLE")
str(box)
#> chr [1:3] "Orange" "CHErrry" "APPLE"

lbox <- lapply(box, tolower)
str(lbox)
#> List of 3
#>  $ : chr "orange"
#>  $ : chr "cherrry"
#>  $ : chr "apple"
```

apply(), sapply() and lapply()

- apply() is used to apply functions to rows or columns of matrices or dataframes across one of the margins of a matrix
- margin=1 refers to the rows and margin=2 to the columns

```
(Y <- matrix(rbinom(20, 9, 0.45), nrow=4))
%>      [,1] [,2] [,3] [,4] [,5]
%> [1,]    6    5    4    2    5
%> [2,]    6    3    3    5    4
%> [3,]    5    2    3    4    4
%> [4,]    3    4    4    3    6
apply(Y, MARGIN=2, FUN=sum)  # apply sum() to all columns
%>[1] 20 14 14 14 19
```

- we can apply() functions to the individual elements of a matrix. In this case, the margin parameter, determines only the shape of the resulting matrix

```
apply(Y, 1, sqrt)
%>          [,1]     [,2]     [,3]     [,4]
%> [1,] 2.449490 2.449490 2.236068 1.732051
%> [2,] 2.236068 1.732051 1.414214 2.000000
%> [3,] 2.000000 1.732051 1.732051 2.000000
%> [4,] 1.414214 2.236068 2.000000 1.732051
%> [5,] 2.236068 2.000000 2.000000 2.449490
apply(Y, 2, sqrt)
%>          [,1]     [,2]     [,3]     [,4]     [,5]
%> [1,] 2.449490 2.236068 2.000000 1.414214 2.236068
%> [2,] 2.449490 1.732051 1.732051 2.236068 2.000000
%> [3,] 2.236068 1.414214 1.732051 2.000000 2.000000
%> [4,] 1.732051 2.000000 2.000000 1.732051 2.449490
```


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
```

Random numbers

- random numbers from a uniform distribution $\mathcal{U}(0,1)$ are generated using `runif()`
- the random generation seed is set via `set.seed()`

```
set.seed(2019)
runif(3)
%> [1] 0.7699015 0.7128397 0.3033602
```

- resetting the seed with the same value will generate the same sequence of random numbers
- is is also possible to save the current seed and reuse it to obtain the same random numbers sub-sequence

```
current.seed <- .Random.seed  # save the current seed
runif(3)
%> [1] 0.61823636 0.05048374 0.04321880
runif(3)
%> [1] 0.820176206 0.009614496 0.102491504

current.seed -> .Random.seed # reset the previous sequence seed
runif(5)
%> [1] 0.618236361 0.050483740 0.043218804 0.820176206 0.009614496
```


sampling from a vector

- generating random numbers from probability distributions will be discussed in the next lessons
- now we want to randomize (shuffling or sampling from) the elements of a vector
- There are two ways of sampling:
  1) sampling without replacement : all the vector values will appear in output, but in a randomized sequence
  2) sampling with replacement : some vector values may be re-selected and appear more than once in the output
- using `sample()`, sampling without replacement is the default operation

```
y <- c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
sample(y) # reshuffling all vector values
%>  [1]  3  9  2  5  4  8  6  8  6  4 10  3  7 11  9
sample(y, 5) # pick up only 5 values from the original vector
%> [1] 3 8 9 4 8
sample(y, 5) # just redo it, and a different sequence may appear
%> [1] 8 3 8 4 3
```

- The option `replace=T` allows for sampling with replacement

```
sample(y, replace=T)
%>  [1]  8  3  6  8  8  4  3  7 10  9 10  9  4  4  7
```

sample()'s surprise example

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

sample(x[x>8])
%> [1] 10  9

sample(x[x>9])
%>  [1]  1 10  8  7  6  5  4  2  9  3

sample(x[x > 10])
%> integer(0)
```

sample(x, size, replace = FALSE, prob = NULL)
If 'x' has length 1, sampling takes place from '1:x'

- the first argument of sample() can be a vector of more than one element or an integer
- the resample() function is safer

```
sample(15)
%>  [1]  2  4  3 11  7 14  6  5  1 13 15 10 12  9  8

str(x[x>9])
%>  int 10

resample(x[x>8])
%> [1] 10  9
resample(x[x>9])
%> [1] 10
```

the resample() function is available in the gdata package. https://cran.r-project.org/web/packages/gdata/index.html

R subsetting

- R's subsetting operators are fast and powerful, and allow to perform complex operations in a way that few other languages can match

- there are 6 ways to subset atomic vectors

- there are 3 subsetting operators: [[, [and $

- subsetting can be combined with assignment

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

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

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

- there are two other subsetting operators:
- `[[]]` is used to extract single items
- `$` is used as a shorthand: `x$y` stands for `x[["y"]]`

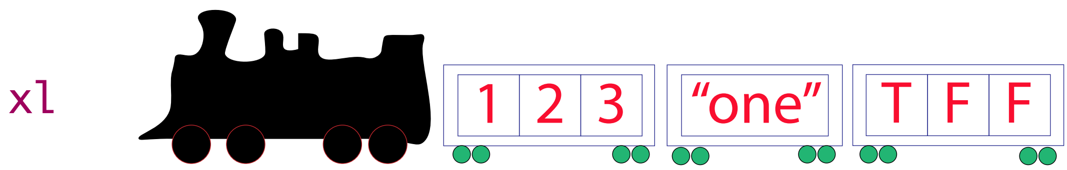
- `[[]]` is most important while working with lists: subsetting a list with single `[]` always returns a smaller list

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

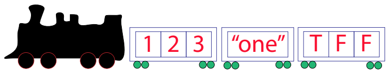
https://twitter.com/RLangTip/status/268375867468681216

- with this metaphor let's build a list

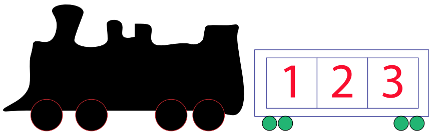
```
xl <- list(1:3, "one", c(T,F,F))
```



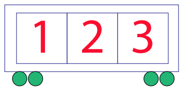
Selecting a single element

xl 

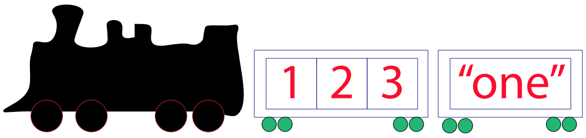
- two options are available when extracting a single element:
  - create a smaller train, with fewer cars (using `[]`)

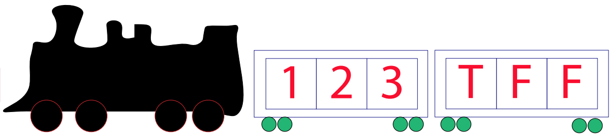
xl[1] 

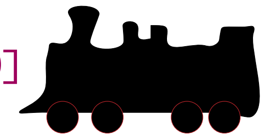
  - or extract the content of a particular car (with `[[]]`)

xl[[1]] 

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

xl[1:2] 

xl[-2] 

xl[0] 

Loops

- Let's create a function, using loops, to evaluate the factorial:
- $n! = n \cdot (n-1) \cdot (n-2) \ldots 2 \cdot 1$

```
fac1 <- function(x) {
  f <- 1
  if (x<2) return (1)
  for (i in 2:x) {
     f <- f*i
  }
  return (f)
}
sapply(1:5, fac1)
%> [1]   1   2   6  24 120
```

```
fac2 <- function(x) {
  f <- 1; t <- x
  while (t>1) {
     f <- f*t
     t <- t-1
  }
  return(f)
}
sapply(1:5, fac2)
%> [1]   1   2   6  24 120
```

```
fac3 <- function(x) {
  f <- 1; t <- x
  repeat {
     if (t<2) break
     f <- f*t
     t <- t-1
  }
  return(f)
}
sapply(1:5, fac3)
%> [1]   1   2   6  24 120
```

- But it is almost always better to use a built-in function that operates on the entire vector, removing the need of loops or repeats

```
> cumprod(1:5) # it does not work for 0
[1]   1   2   6  24 120
> fac4 <- function(x) max(cumprod(1:x))
> sapply(1:5, fac4)
[1]   1   2   6  24 120

> # R implements a factorial() function, introduced not long ago
> sapply(1:5,factorial)
[1]   1   2   6  24 120
```

Loop avoidance: use vectorized operations

- it's a good R programming practice to avoid loops wherever possible
- in many cases, using vector functions, makes it particularly straightforward

```
> y <- c(-3,4,-2,-1,8,7,9)
> y
[1] -3  4 -2 -1  8  7  9

> for (i in 1:length(y)) {if (y[i] < 0) y[i] <- 0}
> y
[1] 0 4 0 0 8 7 9
```

- in the example below, a loop can be replaced by logical subscripts

```
> y <- c(-3,4,-2,-1,8,7,9)

> y[y<0] <- 0

> y
[1] 0 4 0 0 8 7 9
```


the ifelse() vectorized function

- ifelse() allow to work on an entire vector without using loops
-

```
> y <- log(rpois(20,1.5))
> y
 [1]      -Inf 1.0986123 0.0000000 0.0000000 0.0000000 0.6931472
 [7] 0.6931472      -Inf 1.3862944 0.6931472 1.3862944      -Inf
[13]      -Inf 0.0000000 0.0000000 1.0986123 0.0000000 0.0000000
[19]      -Inf 1.0986123

> mean(y)
[1] -Inf

> (y <- ifelse(y<0, NA, y))
 [1]        NA 1.0986123 0.0000000 0.0000000 0.0000000 0.6931472
 [7] 0.6931472        NA 1.3862944 0.6931472 1.3862944        NA
[13]        NA 0.0000000 0.0000000 1.0986123 0.0000000 0.0000000
[19]        NA 1.0986123

> mean(y, na.rm=TRUE)
[1] 0.5431911
```

Loops are slow, compared to vectorized operations

- let's generate $5 \cdot 10^7$ events according to an uniform distribution, $\mathcal{U}(0, 1)$
- we want to search for the maximum value in the vector using the vectorized function `max()` e by using conventional loops

```
x <- runif(50000000)

system.time(max(x))
%>    user  system elapsed
%>   0.106   0.000   0.106

pc <- proc.time()
cmax <- x[1]
for (i in 2:length(x)) { if(x[i]>cmax) cmax <- x[i] }

proc.time()-pc
%>    user  system elapsed
%>   2.061   0.071   2.133
```

- `system.time()` and `proc.time()` produce a vector of three numbers, showing the user, system and total elapsed time in seconds

Good/Bad practice in building vectors

- we want to build a vector containing 10^n elements in the sequence 1:10^n
- three ways are analyzed

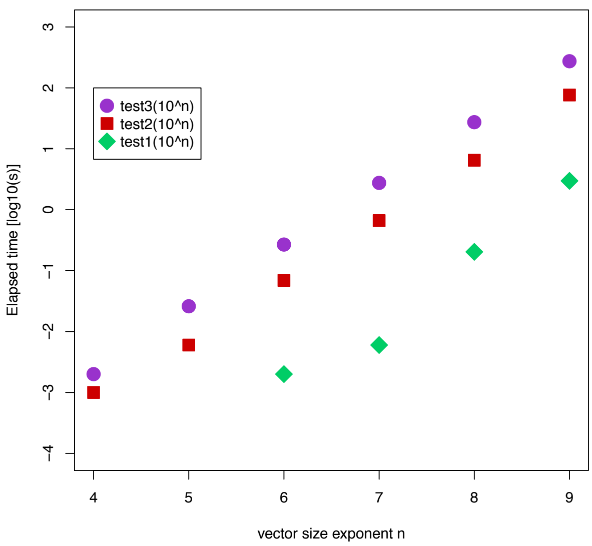
```
test1 <- function(n){
    y <- 1:n
}
```

```
test2 <- function(n){
    y <- numeric(n)
    for (i in 1:n)
        y[i] <- i
}
```

```
test3 <- function(n){
    y <- NULL
    for (i in 1:n)
        y <- c(y,i)
}
```

```
> system.time(test1(10000000))
  user  system elapsed
 0.006   0.000   0.006
> system.time(test2(10000000))
  user  system elapsed
 0.622   0.011   0.633
> system.time(test3(10000000))
  user  system elapsed
 2.755   0.003   2.758
```

- the first method (`test1`) is the best
- the loop using a pre-determined vector length is reasonably fast
- the last method (`test3`) is the slowest
- Moral: never grow vectors by repeated concatenation



R Lists example

- let's create a more complex list with etherogeneous object types

```
apples  <- c(4, 4.5, 5, 3.9)
oranges <- c(TRUE, FALSE, TRUE)
chalk   <- c("limestone", "marl", "ooline", "CaCO3")
pears   <- c(3.2-4.5i, 12.8+2.2i)

items <- list(apples, oranges, chalk, pears)
items
%> [[1]]
%> [1] 4.0 4.5 5.0 3.9
%>
%> [[2]]
%> [1]  TRUE FALSE  TRUE
%>
%> [[3]]
%> [1] "limestone" "marl"      "ooline"    "CaCO3"
%>
%> [[4]]
%> [1]  3.2-4.5i 12.8+2.2i
```


R List example: element access

- vectors, matrices and arrays subscripts have one set of square brackets [6], [3,4] or [2,3,2,1]
- lists subscripts have double square brackets [[2]] or [[i,j]]

```
items[[3]]
%> [1] "limestone" "marl"      "ooline"    "CaCO3"

items[[3]][1]
%> [1] "limestone"
```

- if the list elements have names, it is possible to use the operator $ for list indexing

```
names(items) <- c("apples", "oranges", "chalk", "pears")

str(items)
%> List of 4
%>  $ apples : num [1:4] 4 4.5 5 3.9
%>  $ oranges: logi [1:3] TRUE FALSE TRUE
%>  $ chalk  : chr [1:4] "limestone" "marl" "ooline" "CaCO3"
%>  $ pears  : cplx [1:2] 3.2-4.5i 12.8+2.2i

items$pears
%> [1]  3.2-4.5i 12.8+2.2i
```

R list example: Applying functions

- the length of the list is the number of items on the list. To get the length of the individual vectors we use the `lapply()` function

```
length(items)
%> [1] 4

lapply(items, length)
%> $apples
%> [1] 4

%> $oranges
%> [1] 3

%> $chalk
%> [1] 4

%> $pears
%> [1] 2
```

```
class(items)
%> [1] "list"

lapply(items, class)
%> $apples
%> [1] "numeric"

%> $oranges
%> [1] "logical"

%> $chalk
%> [1] "character"

%> $pears
%> [1] "complex"
```


R Lists : 3

- applying numeric functions to the list, will only work for objects of class `numeric` or `complex`

```
mean(items)
%> [1] NA
%> Warning message:
%> In mean.default(items) :
%>   argument is not numeric or logical: returning NA

lapply(items, mean)
%> $apples
%> [1] 4.35

%> $oranges
%> [1] 0.6666667

%> $chalk
%> [1] NA

%> $pears
%> [1] 8-1.15i
%> Warning message:
%> In mean.default(X[[i]], ...) :
%>   argument is not numeric or logical: returning NA
```

- a warning message points out that the third vector cannot be coerced to a number (it is not numeric, complex or logical), therefore NA appears in the output

R Lists : 4

- The summary() function works for lists, but the most useful overview of a list content is given by str(), the structure function:

```
summary(items)
%>          Length Class  Mode
%> apples  4      -none- numeric
%> oranges 3      -none- logical
%> chalk   4      -none- character
%> <NA>    2      -none- complex

str(items)
%> List of 4
%>  $ apples : num [1:4] 4 4.5 5 3.9
%>  $ oranges: logi [1:3] TRUE FALSE TRUE
%>  $ chalk  : chr [1:4] "limestone" "marl" "ooline" "CaCO3"
%>  $ NA     : cplx [1:2] 3.2-4.5i 12.8+2.2i
```


Question Time

- What is the effect of [[1:2]] on a list ?

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

% xl[[1:2]] is equivalent to xl[[1]][[2]]
xl[[1:2]]
[1] 2
xl[[1]][[2]]
%> [1] 2
% i.e. it extracts the element stored in the list at position 1, and
%      then it gets the second element

% This fails:
 xl[[1:3]]
%> Error in xl[[1:3]] : recursive indexing failed at level 2
% and it is equivalent to:
> (a1 <- xl[[1]])
[1] 1 2 3
> (a2 <- a1[[2]])
[1] 2
> (a3 <- a2[[3]])
Error in a2[[3]] : subscript out of bounds
```