R Data Types - part I

Alberto Garfagnini

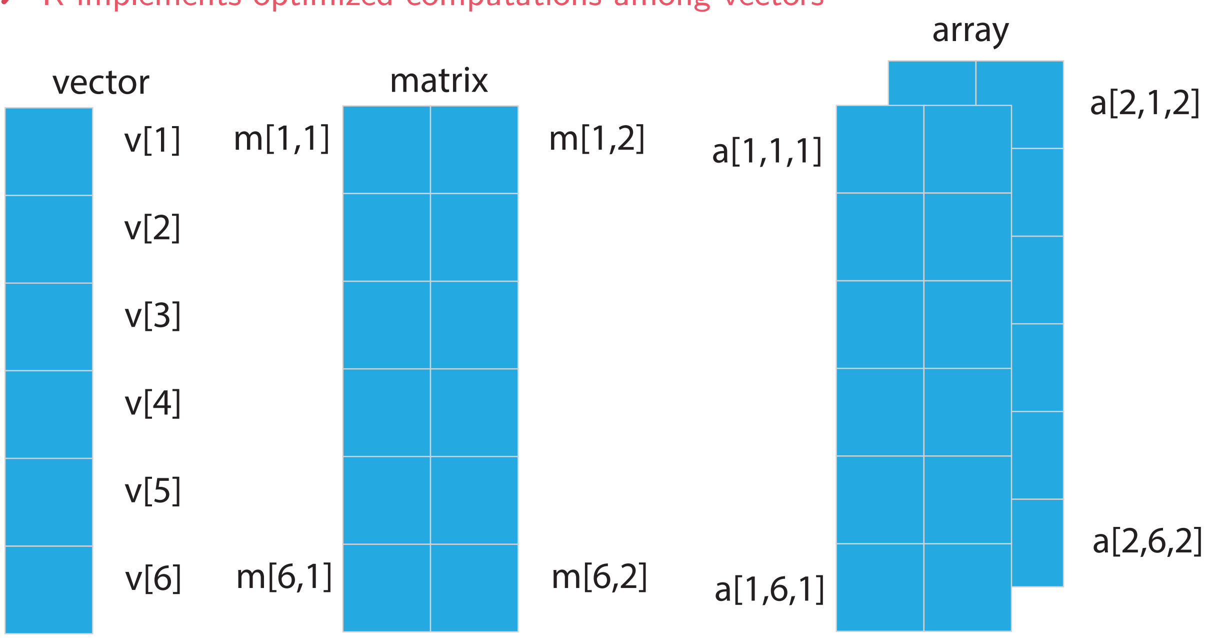
Università di Padova

AA 2023/2024 - R lecture 2

R data types: `atomic vectors`

- the basic data structure is a `vector` : a sequence of values stored in contiguous memory areas
- the `matrix` is a 2-dim representation of a vector
- the `array` is a multi-dimensional vector (with $\dim > 2$)
- `vector`, `matrix` and `array` are atomic types : all elements must be of same type

➜ R implements optimized computations among vectors



R is dynamically-typed language

- R is a dynamically-typed language

```
x <- 1L
y <- 2
x + y
# [1] 3

y <- 5.6
x + y
# [1] 6.6
```

➜ Dynamic typing allows to assign a value of a different data type to the same variable at any time

```
y <- "Test"

x + y
# Error in x + y : non-numeric argument to binary operator
```

➜ But the operation is acknowledged with an error if the data types of the two variables are not compatible

All variables do not need to be declared

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 can be combined:

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

Useful functions to inspect data types and storage

class()

- tell us what sort of data we have

```
x <- c(3, 7, 9)
class(x)
# [1] "numeric"
```

typeof() / storage.mode()

- get or set the mode (i.e. the type), or the storage mode of an R object

```
typeof(x)
# [1] "double"
```

length()

- returns the number of element in the object

```
length(3)
# [1] 3
```

str()

- Compactly display the internal structure of an R object

```
str(x)
# num [1:3] 3 7 9
```


Vector Maths

- several Maths functions are available for vectors

```
(x <- 3:1)
# [1] 3 2 1

str(x)
# int [1:3] 3 2 1

f1 <- 2.5
x*f1
# [1] 7.5 5.0 2.5


f2 <- c(4, 2)
x*f2
# [1] 12  4  4
# Warning message:
# In x * f2 : longer object length is not a multiple
#             of shorter object length


f3 <- c(4, 2, 1)
x*f3
# [1] 12  4  1
```

The shorter vector, f2, is duplicated to cover the length of x

Vector indexing - (1)

- 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
```

- sub-scripting is done through square brackets [] (indexing starts at '1')

```
probe[2]
# [1] 7
```

- sub-scripting can be done with vectors

```
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
```


Generating sequences - :,seq()

- R provides and easy way 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

10:-5 # a sequence from 15, down to -5
# [1] 10  9  8  7  6  5  4  3  2  1  0 -1 -2 -3 -4 -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
```

Vector indexing - (2)

- 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
```


Vectors and logical subscripts

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

sum(x)
# [1] 55
```

- What is the result of `sum( x < 5 )` ?

```
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 sub-scripting to perform the desired sum

```
x[x<5]
# [1] 0 1 2 3 4

sum(x[x<5])
# [1] 10
```

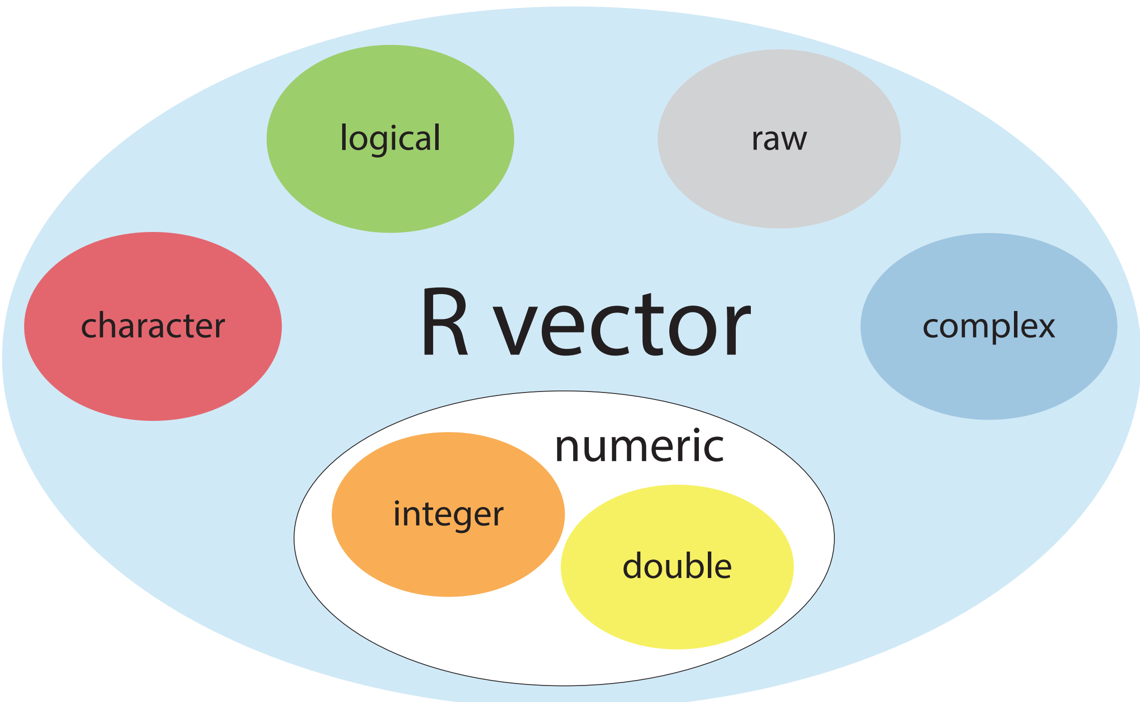
Generating replicated values - `rep()`

- 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"
```


R vector types

- 4 standard vector types are commonly used: `double` and `integer` (under the `numeric` class), `character`, and `logical`
- plus two rare types: `complex` and `raw`



Vector - numeric

- double is the default type for all numbers

```
n <- c(1, 3, 12)

class(n)
[1] "numeric"

typeof(n)
# [1] "double"

str(n)
# num [1:3] 1 3 12
```

- integer is assumed for all sequences

  or for numbers with the L suffix

```
n <- c(1L, 3L, 12L)

class(n)
# [1] "integer"

typeof(n)
# [1] "integer"

str(n)
# int [1:3] 1 3 12
```


Vector - logical

- logical vectors can assume two possible values:

  TRUE or FALSE
- it can be abbreviated with T and F

```
x <- -5:5
# Select the positive numbers in the vector
l.positive <- x > 0

# Select all even numbers
l.even <- x %% 2 == 0

# and the odd numbers
l.odd <- x %% 2 > 0

# Get all odd, positive numbers
x[l.odd & l.positive]
# [1] 1 3 5

# and all even, positive numbers
x[l.even & l.positive]
# [1] 2 4
```

Vector - `character`

- logical vectors can assume two possible values:

```
k <- c("one", "two", "three", "one")
class(k)
# [1] "character"

typeof(k)
# [1] "character"

str(k)
# chr [1:4] "one" "two" "three" "one"
```


Infinity - `Inf`

- 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
```

Not a Number: NaN

- 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

```
v1 <- c(-4.5, 0/0, exp(Inf))

is.finite(v1)
# [1] TRUE FALSE FALSE

is.infinite(v1)
# [1] FALSE FALSE  TRUE

is.nan(v1)
# [1] FALSE  TRUE FALSE
```


Missing or unknown values: NA

- 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
```

Factors

- Factors are categorical variables that have a fixed number of levels

```
gender <- c("Male", "Female", "Male", "Male", "Female", "Female")
class(gender)
# [1] "character"

mode(gender)
# [1] "character"

str(gender)
# chr [1:6] "Male" "Female" "Male" "Male" "Female" "Female"
```

- Factors are stored internally as numbers

```
gender <- factor(gender)
class(gender)
# [1] "factor"

mode(gender)
# [1] "numeric"

str(gender)
Factor w/ 2 levels "Female","Male": 2 1 2 2 1 1

levels(gender)
# [1] "Female" "Male"
```


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

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 assigning 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")))
# a b c
# 1 2 3
```

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

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

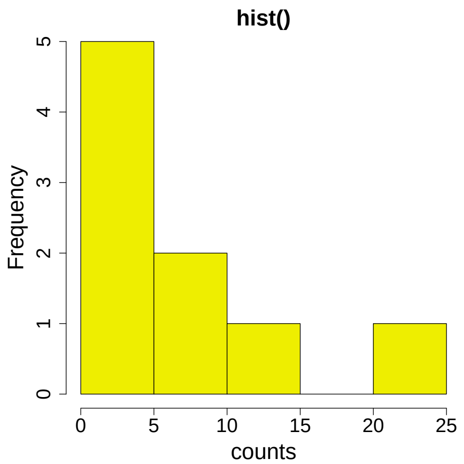
- and removed with `uname()`, or setting `names(x) <- NULL`

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

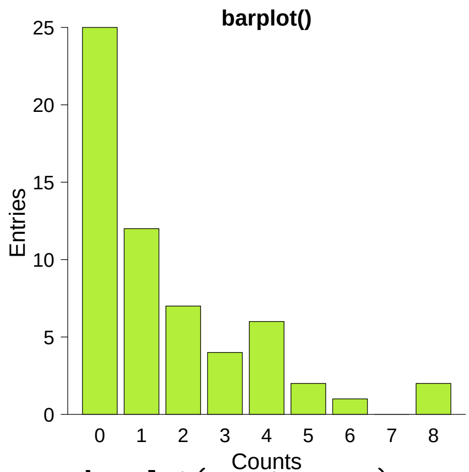

Example : naming vector elements

- sometimes is is useful to have values in a vector labeled
- 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 transform 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
```

Matrix

- An `matrix` is a bi-dimensional array where all the entries have the same class

```
(m1 <- matrix(1:12, nrow=3))
#      [,1] [,2] [,3] [,4]
# [1,]    1    4    7   10
# [2,]    2    5    8   11
# [3,]    3    6    9   12
```

- the default filling (by column) can be changed (`byrow = TRUE`)

```
(m2 <- matrix(1:12, nrow=4, byrow=T))
#      [,1] [,2] [,3]
# [1,]    1    2    3
# [2,]    4    5    6
# [3,]    7    8    9
# [4,]   10   11   12

class(m2)
# [1] "matrix" "array"
attributes(m2)
# $dim
# [1] 4 3
```

- we can "rearrange" the matrix by changing the number of rows and columns

```
dim(m2) <- c(2,6)
m2
#      [,1] [,2] [,3] [,4] [,5] [,6]
# [1,]    1    7    2    8    3    9
# [2,]    4   10    5   11    6   12
```


Matrix element access

- A `matrix` element is accessed with the syntax:

`A[row, col]`

```
(m1 <- matrix(1:12, nrow=3))
#      [,1] [,2] [,3] [,4]
# [1,]    1    4    7   10
# [2,]    2    5    8   11
# [3,]    3    6    9   12

print(m1[1,3])
# [1] 7
```

- it is possible to access a single row

`A[row, ]`

```
m1[1,]
# [1]  1  4  7 10
```

- and column

`A[, col]`

```
m1[,2]
# [1] 4 5 6
```

- and a subset of the full matrix (rows 1 2 and columns 2 and 3)

```
m1[1:2,2:3]
#      [,1] [,2]
# [1,]    4    7
# [2,]    5    8
```

Matrix Maths - `product`

- Let's define two matrices

```
(A <- matrix (c (1, 0, 4, 2, -1, 1), nrow = 3))
#      [,1] [,2]
# [1,]    1    2
# [2,]    0   -1
# [3,]    4    1

(B <- matrix (c (1, -1, 2, 1, 1, 0), ncol = 3))
#      [,1] [,2] [,3]
# [1,]    1    2    1
# [2,]   -1    1    0
```

- we can multiply them since (`nrows(A) = ncol(B)`)

```
A %*% B
#      [,1] [,2] [,3]
# [1,]   -1    4    1
# [2,]    1   -1    0
# [3,]    3    9    4

B %*% A
#      [,1] [,2]
# [1,]    5    1
# [2,]   -1   -3
```


Square Matrix Maths - `diag() / det() / solve()`

- we can define a 3×3 diagonal matrix

```
(I <- diag (1, nrow = 3, ncol = 3))
#      [,1] [,2] [,3]
# [1,]    1    0    0
# [2,]    0    1    0
# [3,]    0    0    1
```

- and compute the determinant of a square matrix

```
C <- matrix (c (1, 2, 4, 2, 1, 1, 3, 1, 2), nrow = 3)
det(C)
# [1] -5
```

- since the determinant is $\neq 0$ we can compute the inverse, C^{-1}

```
Cinv <- solve(C)
C %*% Cinv
#      [,1]         [,2]          [,3]
# [1,]    1 8.881784e-16 -2.220446e-16
# [2,]    0 1.000000e+00 -1.110223e-16
# [3,]    0 0.000000e+00  1.000000e+00
```

Solving system of linear equations with matrices

- Let's suppose we have the following linear system:

$$\begin{cases} 3x + 4y & = & 12 \\ x + 2y & = & 8 \end{cases}$$

```
(A <- matrix (c (3, 1, 4, 2), nrow = 2))
#       [,1] [,2]
# [1,]     3    4
# [2,]     1    2
(k <- matrix (c (12, 8), nrow = 2))
#       [,1]
# [1,]    12
# [2,]     8

v <- solve(A, k)
#        [,1]
# [1,]    -4
# [2,]     6
```

- the solution is $x = -4$ and $y = 6$

```
A %*% v
#       [,1]
# [1,]    12
# [2,]     8
```


Array

- An array is a multi-dimensional object where all the entries have the same class
- The dimensions of an array are specified by its `dim` argument

```
(ar <- array (1:24, dim = c (2, 4, 3)))
#
# , , 1
#       [,1] [,2] [,3] [,4]
# [1,]     1    3    5    7
# [2,]     2    4    6    8
#
# , , 2
#       [,1] [,2] [,3] [,4]
# [1,]     9   11   13   15
# [2,]    10   12   14   16
#
# , , 3
#       [,1] [,2] [,3] [,4]
# [1,]    17   19   21   23
# [2,]    18   20   22   24
```

- we can change it to a matrix by modifying the `dim` attribute

```
dim(ar) <- c(4,6)
ar
#       [,1] [,2] [,3] [,4] [,5] [,6]
# [1,]     1    5    9   13   17   21
# [2,]     2    6   10   14   18   22
# [3,]     3    7   11   15   19   23
# [4,]     4    8   12   16   20   24
```

Data Types testing and coercing

- we can always test whether objects are a particular type
- and also coerce them to a different type

```
lv <- c (T, F, T)

is.logical(lv)
# [1] TRUE

as.numeric(lv)
# [1] 1 0 1

as.character(lv)
# [1] "TRUE"  "FALSE" "TRUE"
```

Type	Testing	Coercing
Array	`is.array()`	as.array()
Character	`as.character()`	as.character()
Complex	`is.complex()`	as.complex()
Dataframe	`is.data.frame()`	as.data.frame()
Double	`is.double()`	as.double()
Factor	`is.factor()`	as.factor()
List	`is.list()`	as.list()
Logical	`is.logical()`	as.logical()
Matrix	`is.matrix()`	as.matrix()
Numeric	`is.numeric()`	as.numeric()
Raw	`is.raw()`	as.raw()
Time series	`is.ts()`	as.ts()
Vector	`as.vector()`	as.vector()