

Introduzione a R

Dott. Alaimo Salvatore

Studio 32 – Blocco 2

Email: alaimos@dmi.unict.it

- Introduzione
- Concetti di base
 - Costrutti fondamentali
 - Strutture dati
 - Grafici
 - L'ambiente RStudio
 - Statistica in R
- Concetti Avanzati
 - Modelli Statistici
 - Grafica Avanzata
 - Programmazione ad Oggetti
 - Programmazione Parallela

Introduzione

Parte I

Cosa è R?

- R è una *suite integrata di strumenti software per la manipolazione dei dati, il calcolo e la visualizzazione grafica*.
- Tra le altre cose, R presenta:
 - una gestione efficace dei dati;
 - un insieme di operatori per i calcoli su array, in particolare matrici;
 - una grande raccolta di strumenti intermedi per l'analisi dei dati;
 - servizi grafici per l'analisi e visualizzazione dei dati;
 - un linguaggio di programmazione semplice ed efficace.
- R è molto più di un veicolo per lo sviluppo di nuovi metodi di analisi interattiva dei dati. Si è sviluppato rapidamente, ed è stato esteso con una *vasta collezione di pacchetti per ogni scopo*.
- Tuttavia, la maggior parte dei programmi scritti in R sono essenzialmente scritti allo scopo di analizzare dati.

Cosa è R?

- **R è un software GNU (gratuito sotto licenza GPL).**
- È disponibile on line: <http://www.r-project.org>
- Le principali repository di pacchetti sono:
 - **CRAN:** <http://cran.r-project.org/>
 - Circa 4398 pacchetti
 - **Bioconductor:** <http://www.bioconductor.org/>
 - Circa 600 pacchetti
- Ogni funzione e pacchetto è ampiamente documentato con tutte le informazioni ed esempi che ne facilitano l'utilizzo e l'apprendimento.
- Tutto ciò che serve per apprendere R è disponibile online gratuitamente (anche in italiano):
 - <http://cran.r-project.org/manuals.html>
- Un IDE per R:

- <http://www.rstudio.com/>



I tipi di dati

- R è un linguaggio **object-oriented**.
- Ogni variabile è un **oggetto** a cui è sempre associata una **classe** di appartenenza che ne definisce le proprietà.
- L'unica eccezione sono i **vettori** (gli oggetti più semplici del linguaggio) la cui classe *corrisponde al tipo di dati contenuto in esso*.
- I tipi di dati in R sono in tutto **7** (più uno):
 - *logical*
 - *numeric*
 - *integer*
 - *double o real*
 - *complex*
 - *character*
 - *raw*
 - *function*

- Ci sono 5 strutture dati fondamentali (classi predefinite)
 - **vettori** (classe **vector**)
 - **matrici** (classe **matrix**)
 - **array** (classe **array**)
 - **liste** (classe **list**)
 - **Data frame** (classe **data.frame**)
- Come si stabilisce la classe di appartenenza di un oggetto?
 - **class()**, oppure **is.list()**, **is.function()**, **is.logical()**, e così via...
 - **class()** restituisce il nome della classe di appartenenza, mentre...
 - le altre restituiscono **TRUE** o **FALSE** se l'oggetto appartiene o meno alla classe controllata dalla particolare funzione.

Eseguire istruzioni

- Le istruzioni possono essere eseguite
 - ...o dalla linea di comando digitando al prompt ogni comando (premendo INVIO per eseguirlo)
 - ...o caricando uno script .R (file di testo con la lista dei comandi)
- Per caricare uno script:

```
> source('mioscript.R')  
>
```

- Per caricare una estensione sotto forma di pacchetto, si può usare il comando **library**.

```
> library(splines)  
>
```

- Più istruzioni in una stessa riga devono essere separate da un «;».

Operatori Principali

- L'operatore di assegnamento è la freccia
 - <- oppure ->

```
> x <- 5
> x
[1] 5
> 6 -> y
> y
[1] 6
>
```

- Non è necessario dichiarare le variabili. Se al primo utilizzo una variabile non esiste un errore sarà restituito.
- **ATTENZIONE: R è case-sensitive!**

```
> X
[1] 5
> x
Errore: oggetto "x" non trovato
>
```

Operatori Principali

- Caratteri speciali:
 - Il nome di una variabile non può contenere i caratteri «!», «"», «\$», e «%»
 - Il punto «.» non è un carattere speciale e l'utilizzo è incoraggiato quando una variabile ha un nome complesso (**number.of.elements** è preferibile a **numberOfElements**)
 - Il carattere «#» è il simbolo del commento.
- Le principali parole riservate:
 - **FALSE, TRUE, Inf, NaN, NA, NULL, if, else, for, while, repeat, break, function, in, next, ...**
- Particolare attenzione deve essere posta su due «valori» speciali:
 - **NA**: rappresenta un valore ignoto (non disponibile)
 - **NaN**: rappresenta il risultato indeterminato di una computazione numerica (es. **0/0** o **Inf/0**).
- La funzione **is.nan()** restituisce **TRUE** se e solo se il suo argomento è un **NaN**.
- La funzione **is.na()** restituisce **TRUE** se il suo argomento è **NA** o **NaN**.
 - **ATTENZIONE: NA != NA e NaN != NaN**
- Molte funzioni hanno un parametro opzionale **na.rm** che esclude dalla computazione tutti i valori **NA** e **NaN** evitando errori di calcolo.

Operatori Principali

- Operatori aritmetici:
 - + (somma);
 - - (sottrazione);
 - * (moltiplicazione);
 - / (divisione);
 - ^ (potenza);
 - %/% (divisione intera tra due numeri);
 - %% (resto della divisione intera tra due numeri).
- Operatori logici:
 - & (AND logico)
 - | (OR logico)
 - ! (NOT logico)
- Operatori relazionali:
 - == (uguale), != (diverso), < (minore), > (maggiore), <= (minore uguale), >= (maggiore uguale)
 - L'applicazione degli operatori relazionali ha come risultato **TRUE** o **FALSE**.
 - Le espressioni logiche **TRUE** o **FALSE** corrispondono rispettivamente a **1** o **0**, quindi possono essere utilizzate nelle operazioni aritmetiche.

- Alcune funzioni matematiche:
 - **sum(); prod(); sqrt(); exp(); log(); sin(); cos(); max(); min(); round(); sign();**
 - **ceiling(); floor(); mean(); median(); var(); sd(); cov();**
- Il workspace rappresenta il contenuto della memoria di lavoro.
- Per visualizzare gli oggetti definiti nel workspace si può usare il comando **ls()**.
- Tramite il comando **rm()** è possibile rimuovere un oggetto.

```
> ls()
[1] "altra.variabile" "x"
> rm("altra.variabile")
> ls()
[1] "x"
>
```

- Per qualsiasi informazione su una funzione o sul suo utilizzo sono a disposizione i seguenti comandi:
 - **help(nome_funzione)**
 - **help.search("una stringa")**

Costrutti Fondamentali

Parte 2

Informazioni di base

- Le istruzioni devono essere scritte secondo l'ordine in cui verranno eseguite
- Ogni istruzione deve essere separata dalla successiva. Questo lo si può fare in due modi: **usando il «;»** o **andando a capo**

```
t<-1; sum<-0  
#Or  
product<-1  
cat(t)
```

- Attraverso parentesi graffe è possibile isolare blocchi di istruzioni:

```
repeat  
{sum<-sum+1  
  product<-product*t  
  t<-t+1  
  if(t==8) break()  
}
```

IF-THEN-ELSE

- **if** (*espressione-logica*) *istruzione o blocco* **else** *istruzione o blocco*

```
if (x == 5) print("x = 5") else print("x diverso da 5")
```

```
if ((x %% 2) == 0) {  
    print("x è pari")  
} else {  
    print("x è dispari")  
}
```

```
if ((x %% 2) == 0) {  
    print("x è divisibile per 2")  
} else if ((x %% 3) == 0) {  
    print("x è divisibile per 3")  
} else {  
    print("x non è divisibile ne per 2 ne per 3")  
}
```

```
if (!is.numeric(x)) {  
    print("x non è un numero")  
}
```

- **for** (*nome.variable in vector*) *istruzione o blocco*

```
sum  <- 0
prod <- 1
for (t in 1:10)
{
    sum  <- sum + t
    prod <- prod * t
}
```

```
sum  <- 0
prod <- 1
for (t in seq(from=1, to=10, by=2)) {
    sum  <- sum + t
    prod <- prod * t
}
```

- Il comando **seq(from=1, to=10, by=2)** genera un vettore contenente gli elementi compresi tra «**from**» e «**to**». La distanza tra due elementi consecutiva è specificata dal parametro opzionale «**by**».
- Una forma più compatta del comando è «*from:to*»

- **while** (*espressione-logica*) *istruzione o blocco*

```
sum  <- 0
prod <- 0
t    <- 1;
while (t <= 10) {
    sum  <- sum + t
    prod <- prod + t
    t    <- t + 1
}
```

```
sum  <- 0
prod <- 0
t    <- 1;
while (t <= 10) {
    sum  <- sum + t
    prod <- prod + t
    t    <- t + 2
}
```

CICLO REPEAT

- **repeat** *blocco di istruzioni*

```
sum  <- 0
prod <- 1
t    <- 1
repeat {
  sum  <- sum + t
  prod <- prod * t
  t    <- t + 1
  if (t == 10) break()
}
```

- **switch**(*espressione, lista di valori*)

```
num <- x%%2 + 1 #assegna 1 se x è pari, 2 altrimenti  
answer <- switch(num, "even", "odd");  
cat(answer, "\n")
```

```
result = switch(answer, even = {  
  # ... altri comandi qui ...  
  (x / 2) + 3  
}, odd = {  
  # ... altri comandi qui ...  
  ((x - 1) / 2) + 3  
})
```

Funzioni

- Le funzioni contengono argomenti in entrata e in uscita e permettono di utilizzare variabili interne che non interagiscono con le variabili del workspace.
- Struttura generale di una funzione:

```
Function.Name <- function (argomenti.in.input)
{
  # corpo della funzione
  output
}
```

- L'ultima espressione valutata all'interno della funzione costituisce anche il suo output.
- All'interno di una funzione si possono richiamare altre funzioni definite dall'utente purché siano state definite prima della chiamata.
- È anche possibile definire funzioni ricorsive.

Funzioni

- Per definire un argomento opzionale basta indicare dopo il nome il valore predefinito che dovrà assumere se non specificato diversamente.
- Il valore di un argomento opzionale può a sua volta essere una espressione che sarà valutata prima dell'ingresso nel corpo della funzione.

```
seq <- function(from = 1, to = 1, by = ((to - from)/(length.out - 1)),  
                length.out = NULL, along.with = NULL, ...) {  
  # ... CORPO DEL METODO ...  
}
```

- Uno speciale tipo di argomento è «...». Esso permette di specificare un elenco non definito di argomenti che potrà essere passato ad altre funzioni richiamate nel corpo del metodo.

Funzioni

```
valuta <- function (valX, div=2) {  
  if ((valX %% div) == 0) {  
    t <- 1  
  } else {  
    t <- 2  
  }  
  t  
}  
  
Main <- function (input, ...) {  
  t <- valuta(input, ...)  
  if (t == 1) {  
    cat("numero divisibile\n")  
  } else {  
    cat("numero non divisibile\n")  
  }  
}
```

Main(2)

Main(2, 3)

Main(2, div=3)

Main(input=2, div=3)

Strutture dati

Parte 3

- **5 strutture dati predefinite:**

- **vector**
- **matrix**
- **array**
- **list**
- **data.frame**

- Le funzioni **is.vector()**, **is.matrix()**, **is.array()**, **is.list()** e **is.data.frame()** ritornano **TRUE** se l'argomento è del tipo specificato altrimenti **FALSE**.
- Le funzioni **as.tipo()** permettono il *cast* da un tipo ad un altro.
 - In ambiente R, questa operazione è chiamata «**coercion**».

Vettori

- Un vettore si può definire come una *sequenza ordinata di oggetti dello stesso tipo* (es. number, character, logical).
- Il modo più comune per la costruzione di un vettore è l'utilizzo della funzione **c()**:

```
> vect<-c(1,2,3,4.5,-6) #costruzione di un vettore di cinque
#elementi
> vect[2]
[1] 2
> vect
[1] 1.0 2.0 3.0 4.5 -6.0
> vect[c(1,3,4)] #visualizza gli elementi delle posizioni 1,3,4
[1] 1.0 3.0 4.5
> vect[-c(1,3,4)] #visualizza gli elementi delle posizioni 2,5
[1] 2 -6
> vect[vect<0]+1
[1] -5
```

- Il segno «-» che si trova prima del vettore di indici indica gli *elementi che bisogna escludere* durante la selezione.
- **Qualsiasi tipologia di selezione (o qualunque tipo di operazione) che si fa su un vettore è essa stessa un vettore!**

Vettori

- Le operazioni sui vettori (così come sulle matrici e sugli array) sono eseguite su ogni elemento contenuto nel vettore (o matrice o array):

```
> vect[vect>0]  
[1] 1.0 2.0 3.0 4.5  
> vect[vect>0]+c(2,2,2,2)  
[1] 3.0 4.0 5.0 6.5
```

- l'operazione deve essere lecita!

```
> vect[vect>0]+c(2,3,4)  
[1] 3.0 5.0 7.0 6.5  
Warning message:  
In vect[vect > 0] + c(2, 3, 4) :  
  longer object length is not a multiple of shorter object length
```

- Il risultato viene restituito ugualmente, in quanto il vettore più piccolo viene ripetuto fino al raggiungimento della stessa lunghezza.

- **rep()** e **seq()** sono due funzioni, alternative, per la costruzione di un vettore:

```
> rep(c(1,2), times=3) #ripeti il vettore (1,2) 3 volte
[1] 1 2 1 2 1 2
> rep(c(1,2), c(2,3)) #ripeti il vettore "1" 2 volte e "2" 3 volte
[1] 1 1 2 2 2
> seq(3,11,by=2)
[1] 3 5 7 9 11
> seq(3,11,length.out=6)
[1] 3.0 4.6 6.2 7.8 9.4 11.0
```

- **range()** e **length()**, la prima da il range dei valori numerici contenuti nel vettore mentre la seconda restituisce la dimensione dell'oggetto che viene passato come parametro:

```
> range(vect)
[1] -6.0 4.5
> length(vect)
[1] 5
> length(vect[vect>0])
[1] 4
> length(vect[vect>0][4])
[1] 1
```

- La funzione **which()** mostra la posizione degli elementi del vettore (o di un qualsiasi altro oggetto di R) che soddisfano una particolare condizione:

```
> vect  
[1] 1.0 2.0 3.0 4.5 -6.0  
> which(vect<0)  
[1] 5  
> which.max(vect)  
[1] 4  
> which.min(vect)  
[1] 5
```

- Le funzioni **which.max()** e **which.min()** restituiscono, rispettivamente, la posizione dell'elemento massimo e quello dell'elemento minimo all'interno di un qualsiasi oggetto di R.

Sorting

- Esistono due modi per ordinare un vettore, uno attraverso la funzione **sort()** e tramite la funzione **order()**:

```
> vect
[1] 1.0 2.0 3.0 4.5 -6.0
> sort(vect) #ordina in modo crescente
[1] -6.0 1.0 2.0 3.0 4.5
> sort(vect, decreasing=TRUE) #ordina in modo decrescente
[1] 4.5 3.0 2.0 1.0 -6.0
> vect[order(vect)]
[1] -6.0 1.0 2.0 3.0 4.5
> vect[order(vect,decreasing=TRUE)]
[1] 4.5 3.0 2.0 1.0 -6.0
> vectOrd<-c(6,5,4,3,2)
> vect[order(vectOrd)] #l'ordinamento di "vect" avviene rispetto a "vectOrd"
[1] -6.0 4.5 3.0 2.0 1.0
```

- La funzione **order()**, restituendo solo la *sequenza di indici ordinata*, dà la possibilità di ordinare un vettore rispetto ad un altro.

Matrici

- Una **matrice** è una *tabella ordinata di elementi dello stesso tipo*. Ciascun elemento è univocamente localizzato tramite *una coppia di numeri interi*: l'indice di riga e quello di colonna.
- La funzione **matrix()** permette la creazione di una matrice:

```
> mtx<-matrix(1:25, ncol=5) #viene costruita una matrice con un numero di 5 cols
> mtx
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     6    11    16    21
[2,]     2     7    12    17    22
[3,]     3     8    13    18    23
[4,]     4     9    14    19    24
[5,]     5    10    15    20    25
> mtx[,3] #viene mostrata la colonna 3
[1] 11 12 13 14 15
> mtx[6,5]
Error: subscript out of bounds
> mtx[,-c(3,4,5)] #vengono selezionate le colonne 1 e 2
      [,1] [,2]
[1,]     1     6
[2,]     2     7
[3,]     3     8
[4,]     4     9
[5,]     5    10
```

- Gli elementi sono disposti *dall'alto verso il basso e da sinistra verso destra* (disposizione per colonne), a meno di non specificare il parametro «**byrow=TRUE**».

- È possibile eseguire un cast tra matrice e vettore con la funzione **as.matrix()**. Tramite la funzione **as.vector()** avviene il contrario.

```
> vect
[1] 1.0 2.0 3.0 4.5 -6.0
> as.matrix(vect)
      [,1]
[1,] 1.0
[2,] 2.0
[3,] 3.0
[4,] 4.5
[5,] -6.0
> as.vector(mtx[, -c(3,4,5)])
[1] 1 2 3 4 5 6 7 8 9 10
```

- Esistono altre tre funzioni che possono tornare utili per la costruzione di matrici : **cbind()**, **rbind()** e **diag()**.

```
> rbind(c(3,4),5:6,c(-5.5,5.5)) #disposizione per righe
      [,1] [,2]
[1,]  3.0  4.0
[2,]  5.0  6.0
[3,] -5.5  5.5
> cbind(c(3,4),5:6,c(-5.5,5.5)) #disposizione per colonne
      [,1] [,2] [,3]
[1,]    3    5 -5.5
[2,]    4    6  5.5
> diag(mtx)
[1]  1  7 13 19 25
> mtxDiag<-diag(2:5)
> mtxDiag
      [,1] [,2] [,3] [,4]
[1,]    2    0    0    0
[2,]    0    3    0    0
[3,]    0    0    4    0
[4,]    0    0    0    5
```


Matrici

- Costruire matrici vuote:

```
> mtxEmpty<-matrix(,nrow=4,ncol=3) #creazione di una matrice vuota 4x3
```

```
> mtxEmpty
```

	[,1]	[,2]	[,3]
[1,]	NA	NA	NA
[2,]	NA	NA	NA
[3,]	NA	NA	NA
[4,]	NA	NA	NA

```
> mtxEmpty[]<-12:23
```

```
> mtxEmpty
```

	[,1]	[,2]	[,3]
[1,]	12	16	20
[2,]	13	17	21
[3,]	14	18	22
[4,]	15	19	23

Matrici

- Come nel caso dei vettori, si possono utilizzare le funzioni **range()** e **length()**.
- Per sapere la dimensione della matrice basta eseguire la funzione **dim()**. Per semplicità è possibile utilizzare le funzioni **nrow()** e **ncol()** per restituire rispettivamente il numero di righe e colonne.
- La funzione **apply()** dà la possibilità di utilizzare le funzioni matematiche su righe e colonne.

```
> range(mtx)
[1] 1 25
> length(mtx)
[1] 25
> dim(mtx)
[1] 5 5
>
```

```
> mtx
      [,1] [,2] [,3] [,4] [,5]
[1,]     1     6    11    16    21
[2,]     2     7    12    17    22
[3,]     3     8    13    18    23
[4,]     4     9    14    19    24
[5,]     5    10    15    20    25
> apply(mtx,2,max)
[1]  5 10 15 20 25
> apply(mtx,1,max)
[1] 21 22 23 24 25
```

Matrici

- La funzione **t()** permette di calcolare la trasposta di una data matrice.
- Per calcolare l'inversa di una matrice bisogna utilizzare la funzione **solve()** (questa funzione permette, inoltre, di risolvere sistemi di equazioni lineari).

```
> mtxEmpty
      [,1] [,2] [,3]
[1,]  12  16  20
[2,]  13  17  21
[3,]  14  18  22
[4,]  15  19  23
> t(mtxEmpty)
      [,1] [,2] [,3] [,4]
[1,]  12  13  14  15
[2,]  16  17  18  19
[3,]  20  21  22  23
```

```
> mtxDiag
      [,1] [,2] [,3] [,4]
[1,]    2    0    0    0
[2,]    0    3    0    0
[3,]    0    0    4    0
[4,]    0    0    0    5
> invMtxDiag<-solve(mtxDiag)
> invMtxDiag
      [,1]      [,2] [,3] [,4]
[1,]  0.5 0.0000000 0.00 0.0
[2,]  0.0 0.3333333 0.00 0.0
[3,]  0.0 0.0000000 0.25 0.0
[4,]  0.0 0.0000000 0.00 0.2
```

Matrici

- Il prodotto matriciale (riga \times colonna) si effettua attraverso l'operatore «**%*%**»:

```
> invMtxDiag%*%mtxDiag
      [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

- mentre l'operatore «*****» esegue semplicemente il prodotto elemento per elemento:

```
> mtx*vect
      [,1] [,2] [,3] [,4] [,5]
[1,]    1  6.0  11  16.0  21
[2,]    4 14.0  24  34.0  44
[3,]    9 24.0  39  54.0  69
[4,]   18 40.5  63  85.5 108
[5,]  -30 -60.0 -90 -120.0 -150
>
```

- Operatore «**%in%**»: consente di vedere se un'occorrenza dei valori dell'operando di sinistra si trova nei valori dell'operando di destra. Esso ritorna un vettore logico.

Array

- Gli **array** costituiscono *un'estensione delle matrici*, allo stesso modo le matrici sono considerate un'estensione dei vettori .
- Un elemento di un array è rappresentato da un *vettore di indici*.
- Per costruire un array si utilizza la funzione **array()**:

```
> arrayEs<-array(24:49, dim=c(4,2,3)) #creazione di un "array"
> dim(arrayEs)
[1] 4 2 3
> arrayEs[, ,1]
      [,1] [,2]
[1,]    24    28
[2,]    25    29
[3,]    26    30
[4,]    27    31
> arrayEs[2, ,]
      [,1] [,2] [,3]
[1,]    25    33    41
[2,]    29    37    45
> arrayEs[2,2,2]
[1] 37
```

Indicizzazione tramite nomi

- Gli elementi in un vettore, matrice, o array possono essere **indicizzati** sia numericamente che tramite **nomi**.
- Per assegnare nomi si possono usare le funzioni **names(vettore)** (per i vettori), **rownames(matrix)** e **colnames(matrix)** per le matrici, **dimnames(object)** per gli altri oggetti.
- La funzione **unname(object)** rimuove tutti i nomi assegnati ad un oggetto
- È possibile usare il parametro «**dimnames**» dei costruttori matrix e array per assegnare direttamente i nomi.

```
> vect <- 1:4
> names(vect) <- c("elem_a", "elem_b", "elem_c", "elem_d")
> vect
elem_a elem_b elem_c elem_d
      1       2       3       4
> vect["elem_a"]
elem_a
      1
> vect[c("elem_a", "elem_d")]
elem_a elem_d
      1       4
> unname(vect)
[1] 1 2 3 4
>
```

```
> matr <- matrix(1:6, nrow=3, ncol=2, byrow=TRUE)
> rownames(matr) <- c("val_a", "val_b", "val_c")
> colnames(matr) <- c("exp_1", "exp_2")
> matr
      exp_1 exp_2
val_a     1     2
val_b     3     4
val_c     5     6
> matr[, "exp_1"]
val_a val_b val_c
      1     3     5
> matr[c("val_a", "val_b"),]
      exp_1 exp_2
val_a     1     2
val_b     3     4
>
```

Liste

- Una **lista** in R si può definire come *una raccolta di oggetti (anche di tipi diversi), comprese altre liste*.
- Per creare una lista si utilizza il comando **list()**.

```
> lista<-list(rep(1,4),c(1,2,3,4),matrix(2:9, nrow=4),c("Piano","Forte"))
> length(lista) #visualizza la dimensione della lista
[1] 4
> lista[[3]]
      [,1] [,2]
[1,]    2    6
[2,]    3    7
[3,]    4    8
[4,]    5    9
```

- Il numero di oggetti che compongono una lista ne rappresenta la sua dimensione.
- Per accedere agli elementi di una lista:
 - «`[.]`»: restituisce una lista con gli oggetti selezionati da estrarre tramite il comando **unlist()**
 - «`[[.]]`»: restituisce direttamente un oggetto della lista
 - «`lista$nome.elemento`»: restituisce direttamente un oggetto della lista

```
> length(lista[[3]])  
[1] 8  
> lista[[3]][3,2]  
[1] 8  
> lista[[4]] #viene restituito un vettore  
[1] "Piano" "Forte"  
> lista[4] #viene restituito una lista  
[[1]]  
[1] "Piano" "Forte"
```


- È possibile assegnare nomi ad ogni singolo oggetto incluso nella lista durante la sua creazione, in particolare specificando il nome all'interno della funzione **list()** (ad es. ***list(first=rep(1,4)...)...***).
- Un altro modo per nominare o rinominare una lista è quello di utilizzare la funzione **names()**.

```
> names(lista)
NULL
> names(lista)<-c("First","Second","Third","Fourth")
>
> #Estrazione del "primo" elemento
> lista$First #estrae come un vettore
[1] 1 1 1 1
> lista["First"] #estrae come una lista
$First
[1] 1 1 1 1
```

- Il **DataFrame** è l'oggetto più importante in R:
 - Simile ad un matrice, ma è usato per raffigurare dati di diversa entità.
 - Ogni riga corrisponde ad un'osservazione, mentre ogni colonna è una variabile (*quantitativa o categoriale/factor*).
 - Un DataFrame è trattato come lista:
 - Ogni elemento rappresenta una variabile statistica
 - la funzione **length()** restituisce il numero di variabili
 - La funzione **names()** restituisce/imposta i nomi delle variabili
 - La funzione **row.names()** restituisce/imposta i nomi delle osservazioni.
- Per conoscere la dimensione dell'intero dataframe: **dim()**
- Il costruttore dell'oggetto è la funzione **data.frame()**
 - Tra i parametri si può specificare il nome di ogni singola riga del dataframe (**row.names**) e il nome delle osservazioni.

Data Frame

```
> #Viene creato un "dataframe" contenente una variabile 'quantitativa' e una
> #'qualitativa':
> dataFrame<-data.frame(a=0:3, sesso=c("F","M","M","F"))
> dataFrame
  a sesso
1 0     F
2 1     M
3 2     M
4 3     F
> dim(dataFrame) #dimensione del "dataframe" ('osservazioni' e 'variabili')
[1] 4 2
> length(dataFrame)
[1] 2
> dataFrame$altezza=c("172","180","165","163") #Viene aggiunta una nuova
#variabile contenente le altezze in centimetri di ogni singolo individuo.
> dataFrame
  a sesso altezza
1 0     F    172
2 1     M    180
3 2     M    165
4 3     F    163
>
```

Data Frame

```
> names(dataFrame)
[1] "a"      "sesso"  "altezza"
> row.names(dataFrame)
[1] "1" "2" "3" "4"
> row.names(dataFrame)<-c("patient1","patient2","patient3","patient4")
> row.names(dataFrame)
[1] "patient1" "patient2" "patient3" "patient4"
> dataFrame
```

	a	sesso	altezza
patient1	0	F	172
patient2	1	M	180
patient3	2	M	165
patient4	3	F	163

Data Frame

- Esistono sostanzialmente due diversi modi per selezionare dal dataframe una variabile o un'osservazione:

```
> dataframe$sezzo
[1] F M M F
Levels: F M
> dataframe[2,3]
[1] "180"
> dataframe[2,2]
[1] M
Levels: F M
> dataframe[,2]
[1] F M M F
Levels: F M
> dataframe$altezza
[1] "172" "180" "165" "163"
```

- L'uso delle parentesi quadre dà la possibilità di selezionare le osservazioni che si desiderano. Osservate questi due esempi:

```
> #selezione dei valori della variabile "altezza" per le donne:
> dataframe[dataframe$sezzo=="F", "altezza"]
[1] "172" "163"
> #selezione dei valori della variabile "sezzo" con altezza<164
> dataframe$sezzo[dataframe$altezza<=164]
[1] F
Levels: F M
```

Operazioni su Data Frame

- Sostanzialmente sono le stesse di quelle utilizzate su un matrice. Ovviamente, devono avere un senso!

```
> dataFrame$altezza
[1] "172" "180" "165" "163"
> mean(dataFrame$altezza)
[1] NA
Warning message:
In mean.default(dataFrame$altezza) :
  argument is not numeric or logical: returning NA
> dataFrame$altezza<-NULL
> dataFrame$altezza<-c(172,180,165,163)
> mean(dataFrame$altezza)
[1] 170
```

- Per cancellare una variabile basta settarla a **NULL**.

Data Frame

- La funzione **as.data.frame()** forza una matrice di dati ad un dataframe.
- I dati possono essere inseriti in maniera semplice tramite un foglio elettronico e caricati in R usando la funzione **read.table()**.
- La funzione **write.table()** scrive un **data.frame** in un file formattato come tabella.

```
> esDataframe<-read.table(file="./Esempio.csv", #nomefile
+ header=TRUE, #Intestazione
+ sep="\t", #separatore
+ na.strings = "NA", #utile nel momento in cui un file abbia dei valori mancanti
+ dec="." #viene specificato il tipo di carattere per separare i decimali
+ )
> esDataframe
  ID Sesso Eta
1  1     M  20
2  2     F  30
3  3     F  21
4  4     F  25
5  5     M  28
+ )
> write.table(dataFrame,
+ file="EsDataframe.txt",
+ sep="\t",
+ na="NA",
+ dec=".",
+ row.names=TRUE,
+ col.names=TRUE)
```

GRAFICI

Parte 4

Creazione di un grafico

- Le funzioni principali per disegnare un grafico sono **plot()**, **point()** e **lines()**.

```
> asseX<-seq(0,10,0.5)  
> fun1<-sin(asseX)  
> fun2<-cos(asseX)  
> plot(asseX,fun1,type="b",col=2,pch="*",ylab="f(asseX)",xlab="time", ylim=c(-1,1))
```

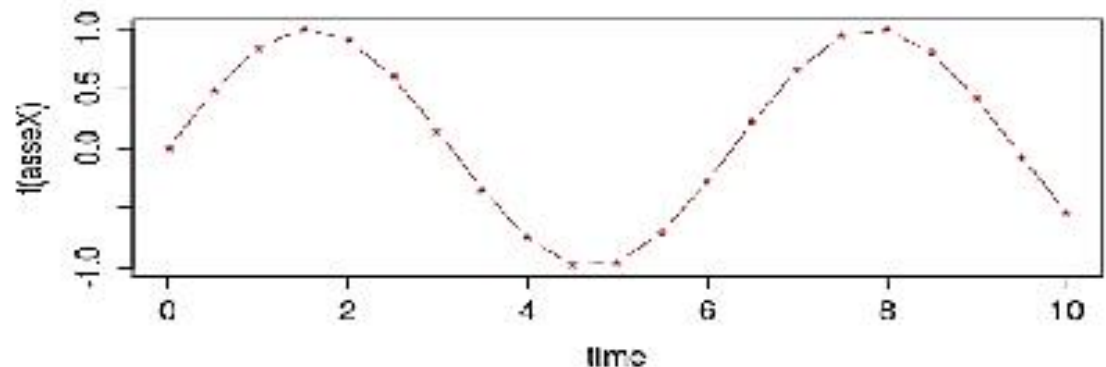
Tipologia grafico:
"b"=both
"l"=lines
"p"=points (default)

Etichette

Range degli assi

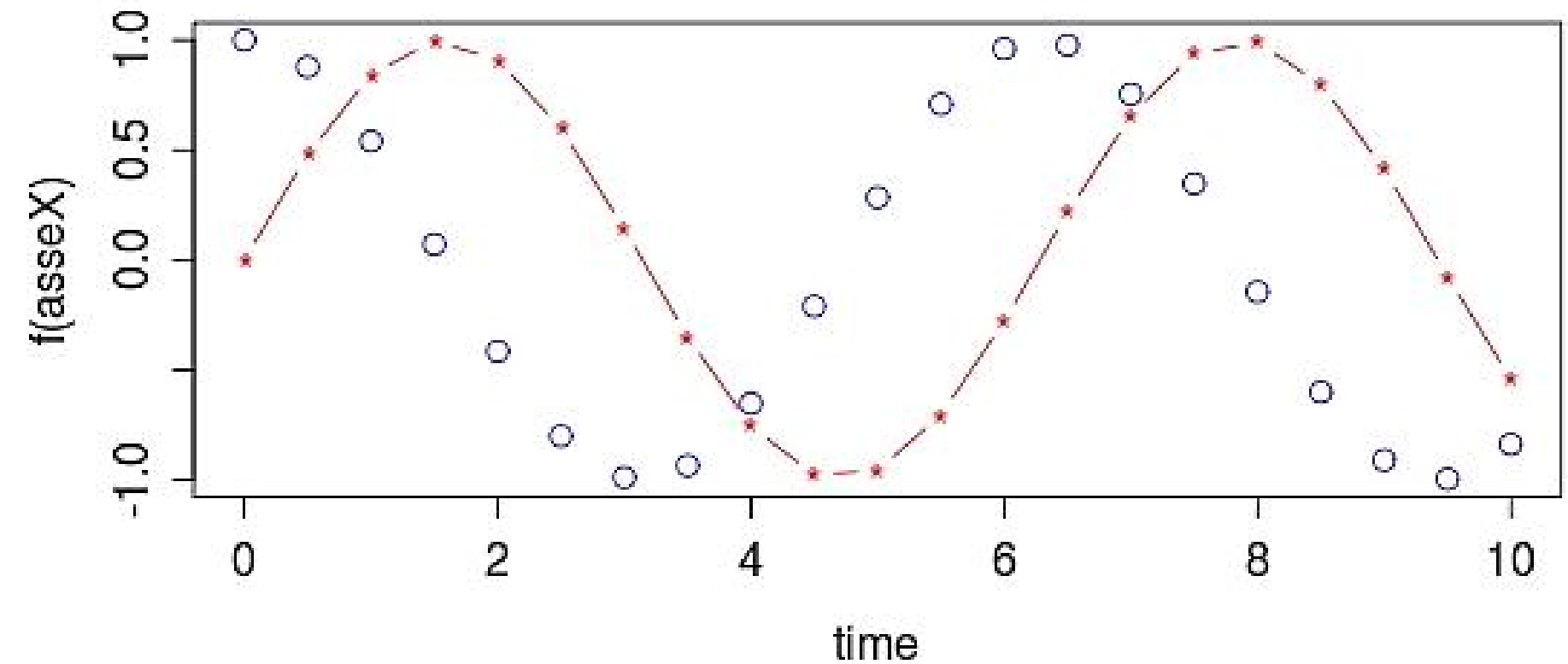
Colore:
default 1= nero

Tipo di punto:
"o" default



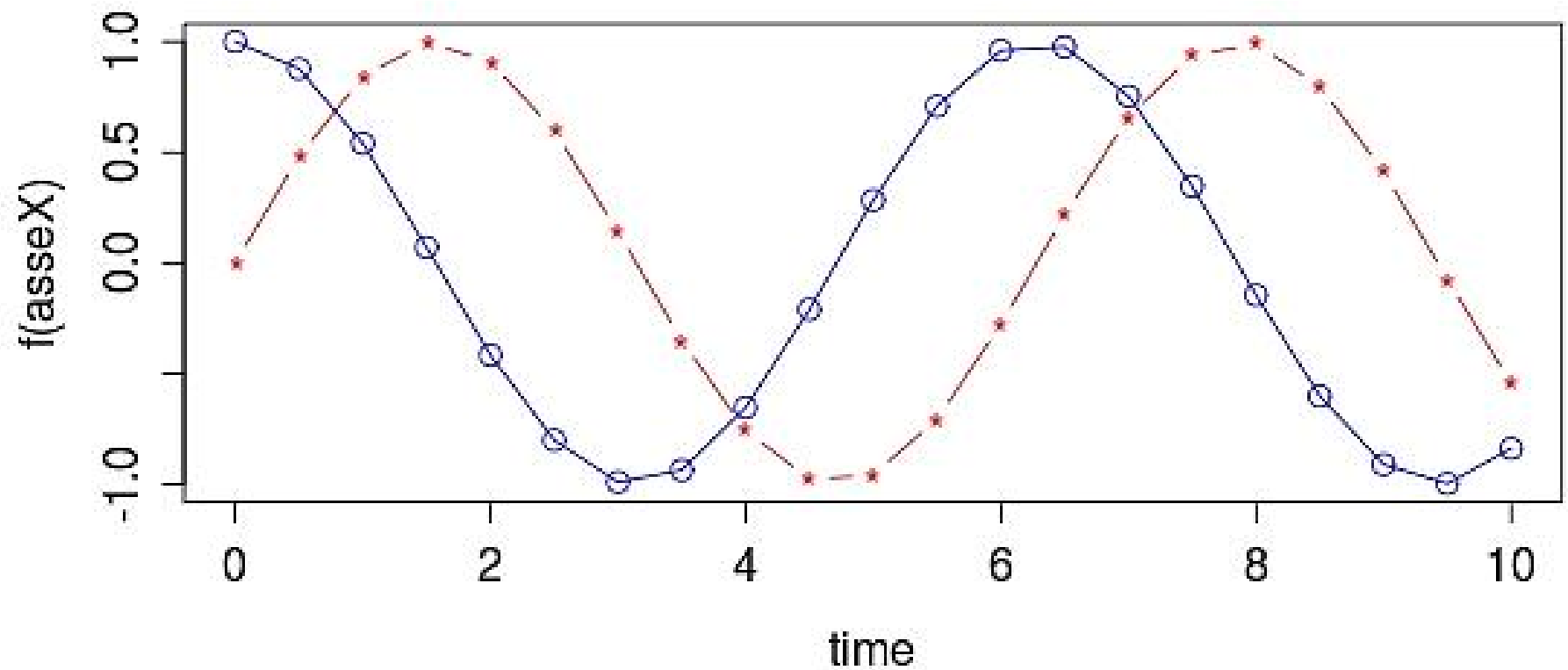
Creazione di un grafico

```
> points(asseX, fun2, col=4)
```



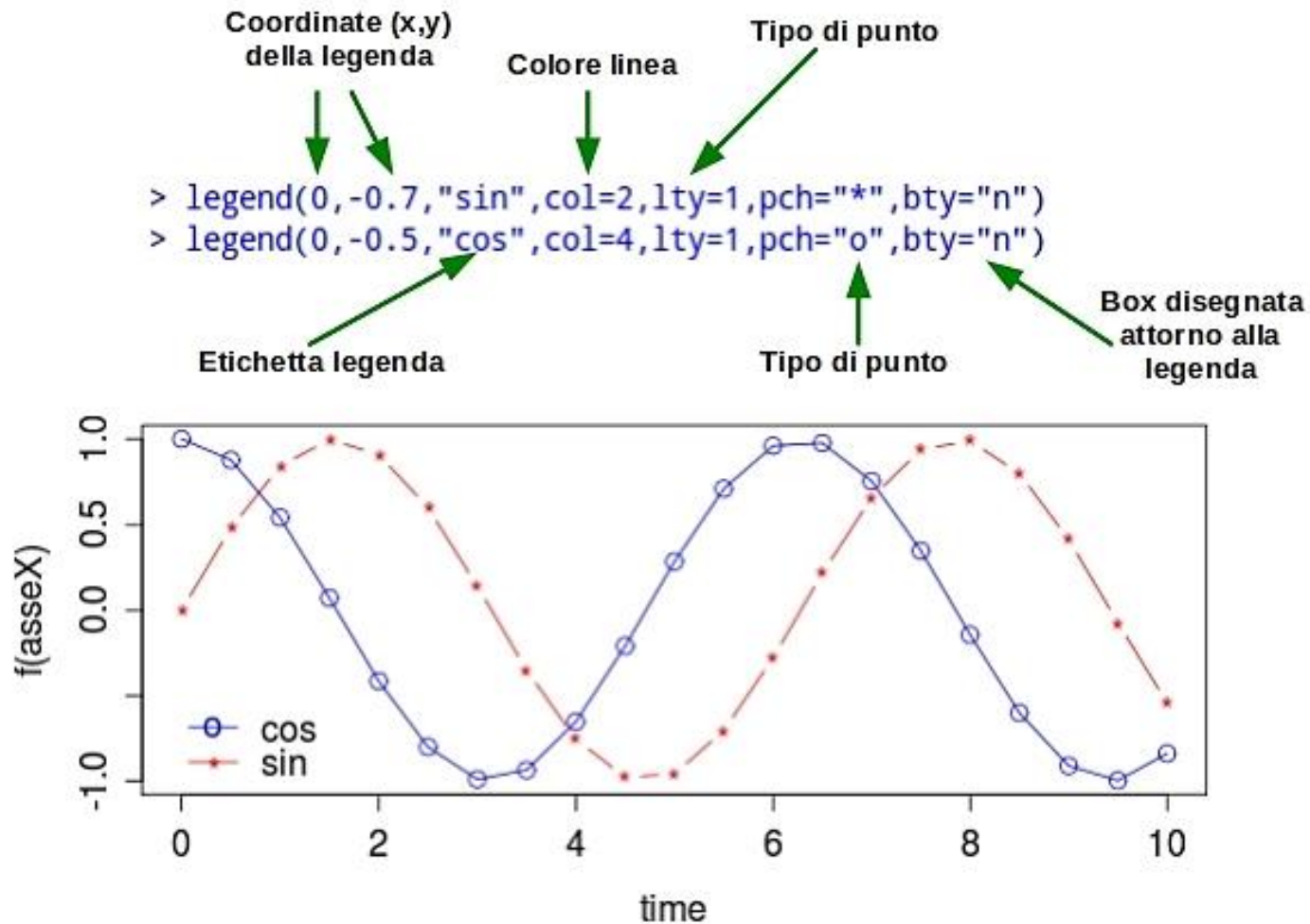
Creazione di un grafico

```
> lines(asseX, fun2, col=4)
```



Legenda

- Per aggiungere una legenda al grafico, c'è la funzione **legend()**.



Salvare un grafico

- Per salvare un grafico invece di visualizzarlo si può usare una delle seguenti funzioni:

Funzione	Salva in
<code>pdf("mygraph.pdf")</code>	pdf file
<code>png("mygraph.png")</code>	png file
<code>jpeg("mygraph.jpg")</code>	jpeg file
<code>bmp("mygraph.bmp")</code>	bmp file
<code>postscript("mygraph.ps")</code>	postscript file

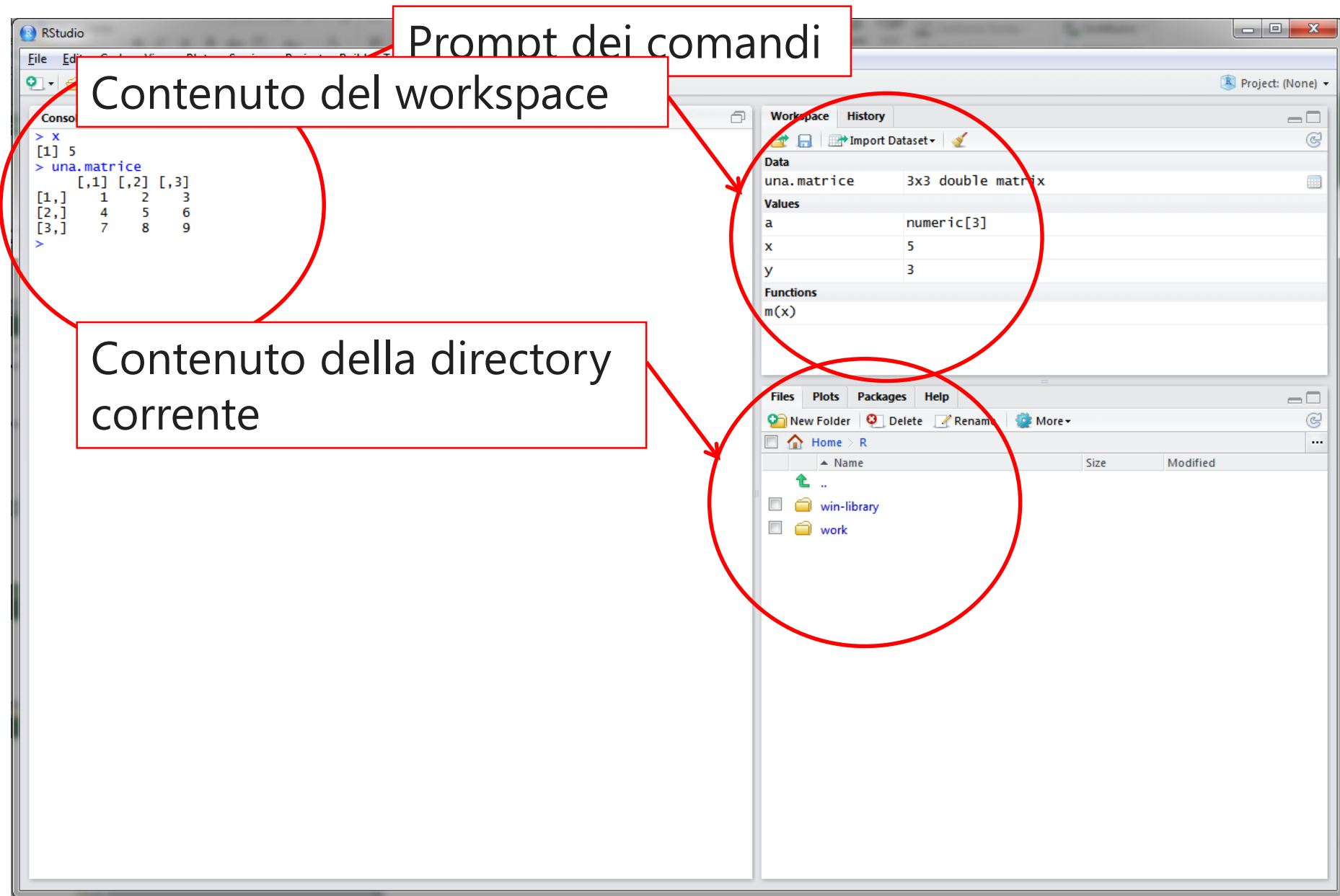
- Esempio:

```
jpeg("myplot.jpg")  
plot(x)  
dev.off()
```

L'ambiente RStudio

Parte 5

Uno sguardo a RStudio



Uno sguardo a RStudio

The image shows the RStudio application window. On the left is the Console pane with the following R code and output:

```
> x  
[1] 5  
> una.matrice  
[1,] 5 23 5 23  
[2,]  
[3,]  
>
```

On the right is the Environment pane, which shows the current workspace. It contains a table with the following data:

Object	Class	Attributes
una.matrice	3x3 double matrix	
values		
a	numeric[3]	
		5
		3

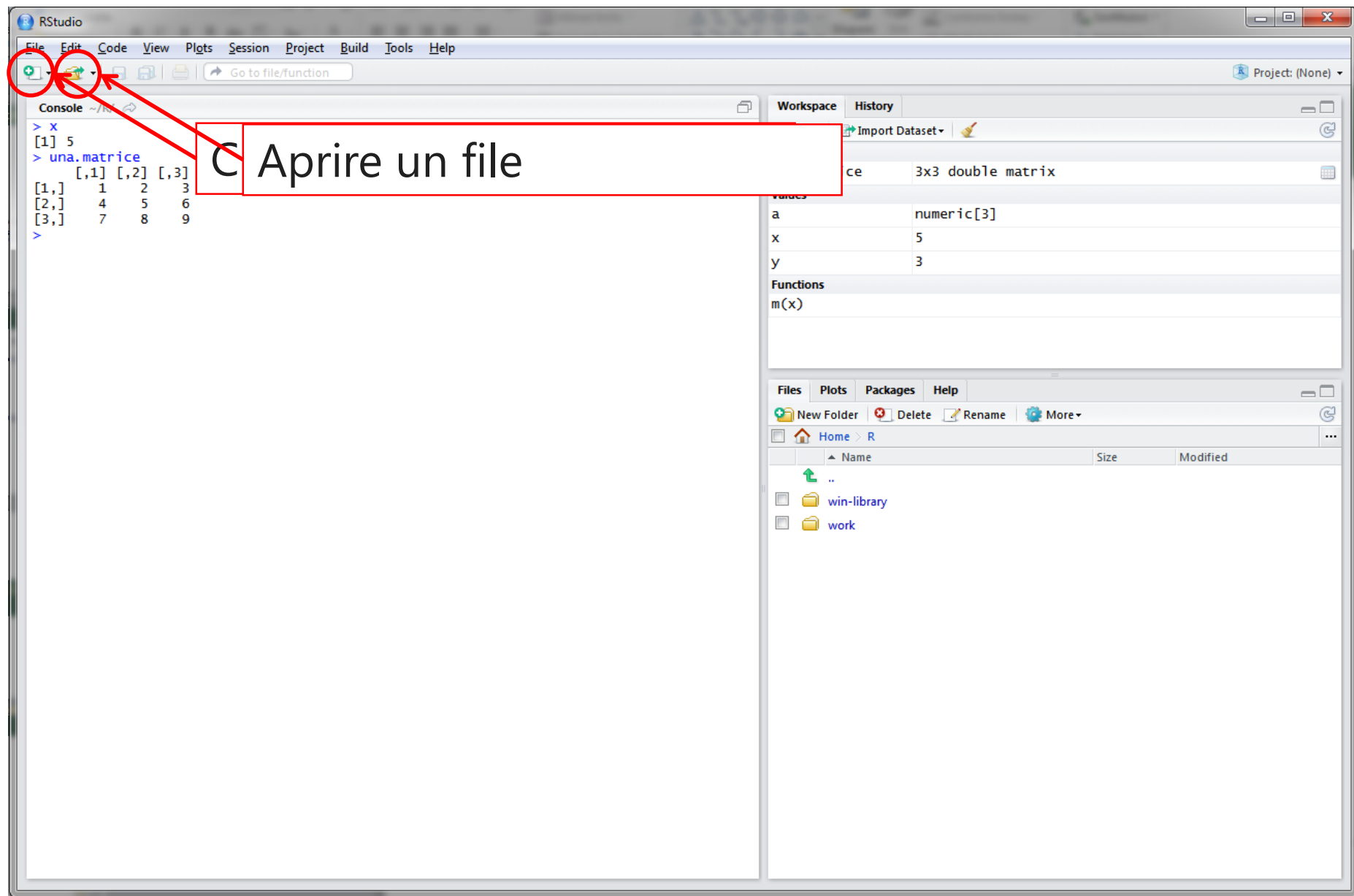
Below the Environment pane is the Files pane, which shows the file explorer. It displays the following structure:

- Home > R
 - ..
 - win-library
 - work

Annotations in red circles and arrows highlight the following features:

- A red box labeled "Salvare un workspace" points to the "Save Workspace" icon (a floppy disk) in the top-left corner of the Environment pane.
- A red box labeled "Importare dati salvati in file" points to the "Import Dataset" button (a green arrow) in the top-left corner of the Environment pane.
- A red box labeled "di Ripulire il workspace corrente." points to the "Clear Environment" button (a trash can) in the top-left corner of the Environment pane.
- A red box labeled "sito web." points to the "Help" button (a question mark) in the top-right corner of the Environment pane.

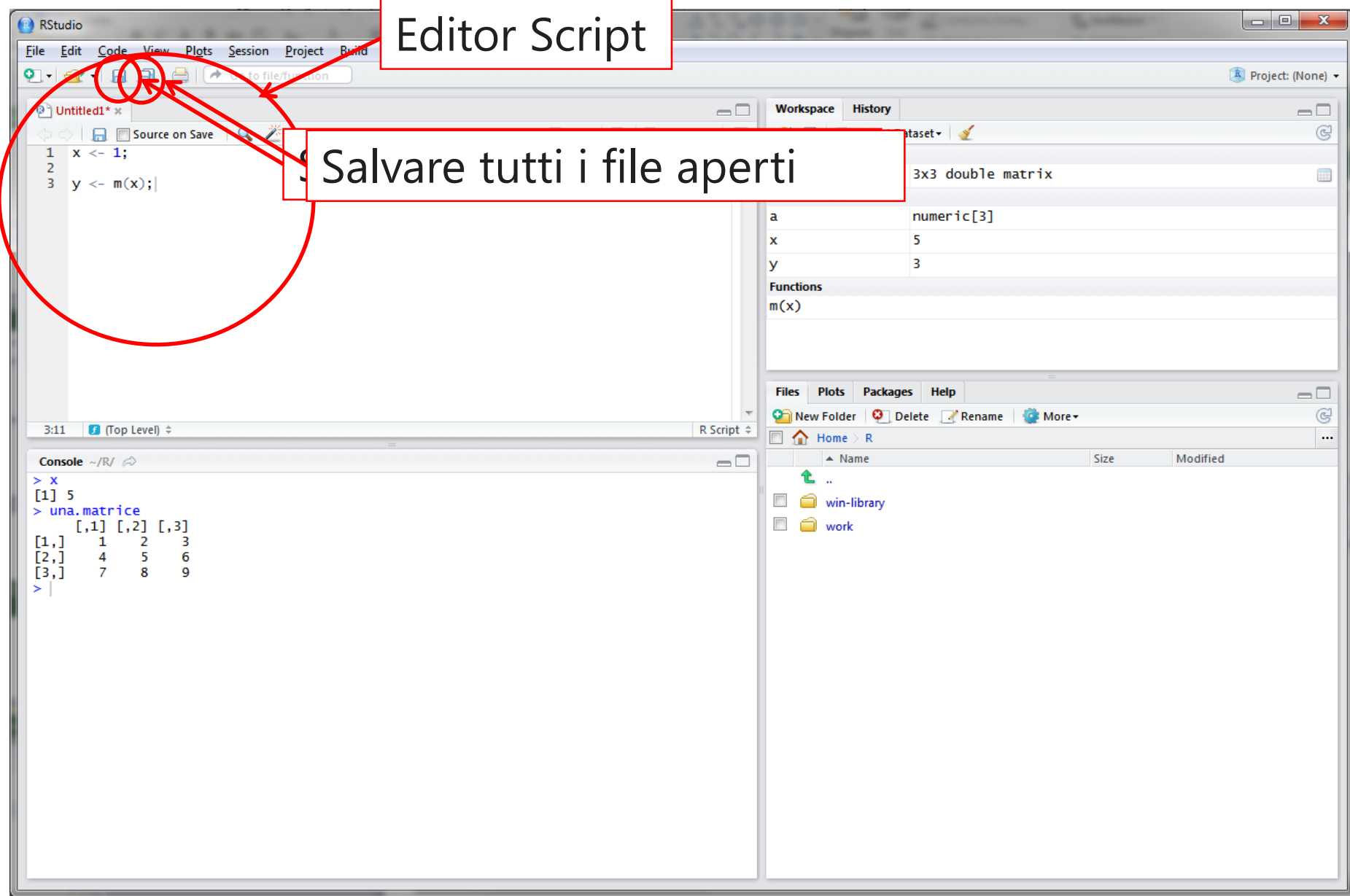
Uno sguardo a RStudio



Uno sguardo a RStudio

Editor Script

Salvare tutti i file aperti



Uno sguardo a RStudio

Storico comandi

Lista dei pacchetti installati
Attivazione/Disattivazione
Pacchetto

The screenshot shows the RStudio environment with several panels. A red circle highlights the 'History' tab in the top right, which contains a list of executed R commands. A red arrow points from the 'Storico comandi' label to this circle. Another red circle highlights the 'Packages' tab in the bottom right, which displays a list of installed and available packages. A red arrow points from the 'Lista dei pacchetti installati' label to this circle. A third red arrow points from the 'Attivazione/Disattivazione Pacchetto' label to the same 'Packages' tab. In the top left, a data frame named 'una.matrice' is displayed. In the bottom left, the R console shows the command 'view(una.matrice)' being executed.

Workspace History

```
ls()  
setwd("~/R")  
x <- 5  
y <- 3  
a <- c(1, 2, 3)  
m <- function(x) { x; }  
una.matrice <- matrix(c(1,2,3,4,5,6,7,8,9), nrow=3, ncol=3, byrow=T)  
x  
una.matrice  
m  
x  
una.matrice  
view(una.matrice)
```

Files Plots Packages Help

Package	Description	Version	Status
affxparser	Affymetrix File Parsing SDK	1.28.1	✖
affyio	Tools for parsing Affymetrix data files	1.24.0	✖
annotate	Annotation for microarrays	1.34.1	✖
AnnotationDbi	Annotation Database Interface	1.18.4	✖
Biobase	Biobase: Base functions for Bioconductor	2.16.0	✖
BiocGenerics	Generic functions for Bioconductor	0.2.0	✖
BiocInstaller	Install/Update Bioconductor and CRAN Packages	1.4.9	✖
biomart	Biological and chemical data networks	1.0	✖
Biostrings	String objects representing biological sequences, and matching algorithms	2.24.1	✖
bit	A class for vectors of 1-bit booleans	1.1-9	✖
bitops	Functions for Bitwise operations	1.0-4.1	✖
boot	Bootstrap Functions (originally by Angelo Canty for S)	1.3-7	✖
car	Tools: moving window statistics, GIF, Base64, ROC AUC, etc.	1.13	✖
class	Functions for Classification	7.3-5	✖
cluster	Cluster Analysis Extended Rousseeuw et al.	1.14.3	✖
codetools	Code Analysis Tools for R	0.2-8	✖

[1,] [1,] [1,] [1,]
[1,] 1 2 3
[2,] 4 5 6
[3,] 7 8 9
> view(una.matrice)
> |

Uno sguardo a RStudio

Visualizzatore variabili

The screenshot shows the RStudio interface with the 'una.matrice' variable selected in the workspace. A red circle highlights the variable viewer window, which displays the data in a table format. The table has 3 rows and 3 columns, labeled V1, V2, and V3. The data values are as follows:

	V1	V2	V3
1	1	2	3
2	4	5	6
3	7	8	9

The console shows the following commands and output:

```
> x  
[1] 5  
> una.matrice  
      [,1] [,2] [,3]  
[1,]    1    2    3  
[2,]    4    5    6  
[3,]    7    8    9  
> view(una.matrice)  
>
```

The workspace pane on the right shows the variable 'una.matrice' as a '3x3 double matrix'. The 'Values' section lists 'a' as 'numeric[3]', 'x' as '5', and 'y' as '3'. The 'Functions' section lists 'm(x)'. The file explorer at the bottom right shows the directory structure: Home > R, with subfolders 'win-library' and 'work'.

Uno sguardo a RStudio

The screenshot displays the RStudio environment with the following components:

- Menu Bar:** File, Edit, Code, View, Plots, Session, Project, Build, Tools, Help.
- Toolbar:** Includes icons for opening files, saving, and navigating.
- Source Editor:** Shows a file named 'una.matrice' with 3 observations of 3 variables. The data is displayed in a table:

	V1	V2	V3
1	1	2	3
2	4	5	6
3	7	8	9

Visualizzatore help

Workspace: Shows the 'una.matrice' object as a 3x3 double matrix. The 'Values' section lists 'a' as numeric[3], 'x' as 5, and 'y' as 3. The 'Functions' section lists 'm(x)'.

Console: Displays the following R commands and output:

```
> x  
[1] 5  
> una.matrice  
      [,1] [,2] [,3]  
[1,]    1    2    3  
[2,]    4    5    6  
[3,]    7    8    9  
> view(una.matrice)  
>
```

Help Pane: The 'Statistical Data Analysis' link is circled in red. Other visible links include 'Manuals', 'Reference', 'Miscellaneous Material', 'About R', 'Authors', 'Resources', 'The R Language Definition', 'R Installation and Administration', 'R Internals', 'Search Engine & Keywords', 'Packages', and 'Frequently Asked Questions'.

Statistica in R

Parte 6

- R fornisce un gran numero di funzioni per calcoli statistici.
- I più importanti indici descrittivi:
 - **mean(x)**: calcola la media aritmetica di un vettore;
 - **var(x,y=NULL)**: calcola la varianza campionaria di un vettore (o la matrice di varianza, o la varianza tra le colonne di due matrici);
 - **cov(x,y=NULL)**: calcola la matrice di covarianza di una matrice (o la covarianza tra due vettori, o la covarianza tra le colonne di due vettori);
 - **cor(x,y=NULL)**: calcola il coefficiente di correlazione lineare tra due vettori (o la matrice di correlazione, o la correlazione tra le colonne di due vettori);
 - **sd(x)**: calcola la deviazione standard di un vettore (se l'input è una matrice o un dataframe, l'output è il vettore delle deviazioni standard delle colonne dell'input);
 - **median(x)**: calcola la mediana di un vettore in input;
 - **quantile(x)**: calcola i quantili di un vettore in input
- Tutti i metodi hanno il parametro opzionale «**na.rm**» che se settato a **TRUE** rimuove tutti i valori *NA* o *NaN* prima di eseguire la computazione.

Distribuzioni di probabilità

- Il calcolo delle funzioni associate alle principali distribuzioni di probabilità è estremamente facile in R.
- È, inoltre, possibile generare osservazioni pseudo-casuali con distribuzione diversa da quella uniforme.
- Tutte queste funzioni sono disponibili nel pacchetto «**stats**» tipicamente caricato di default all'avvio dell'ambiente.
- La struttura di tutte le funzioni che implementano le principali distribuzioni è la stessa:
 - **Densità di probabilità**: si antepone **d** al nome della distribuzione;
 - **Funzione di ripartizione**: si antepone **p** al nome della distribuzione;
 - **Quantili**: si antepone **q** al nome della distribuzione;
 - **Osservazioni pseudo-casuali**: si antepone **r** al nome della distribuzione;

Distribuzioni di probabilità

Distribuzione	Nome in R	Parametri
Beta	beta	shape1, shape2, ncp
Binomiale	binom	size, prob
Binomiale negativa	nbinom	size, prob
Cauchy	cauchy	location, scale
Chi-quadro	chisq	df, ncp
Esponenziale	exp	rate
F	f	df1, df2, ncp
Gamma	gamma	shape, scale
Geometrica	geom	prob
Ipergeometrica	hyper	m, n, k
Log-normale	lnorm	meanlog, sdlog
Logistica	logis	location, scale
Normale	norm	mean, sd
Poisson	pois	lambda
T-student	t	df, ncp
Uniforme	unif	min, max
Weibull	weibull	shape, scale
Wilcoxon	wilcox	m, n

- Parametri comuni delle funzioni di ripartizione:
 - **lower.tail**: se **FALSE** calcola $P(X > x)$, altrimenti (default) $P(X \leq x)$
 - **log.p**: se **TRUE** calcola il logaritmo della probabilità
- Parametri comuni delle funzioni densità di probabilità:
 - **log**: se **TRUE** calcola il logaritmo della probabilità
 - ...utile per lavorare con prodotti e divisioni di numeri molto piccoli.
- Parametri comuni dei generatori di numeri casuali:
 - **n**: indica il numero di osservazioni pseudo-casuali da generare

Test statistici

- Molti tra i test statistici principali sono disponibili nel pacchetto «**stats**».
- Alcuni tra i più comuni sono:
 - **chisq.test(x)**: esegue il test χ^2 su una tabella di contingenza;
 - **binom.test(x,n,p=0.5)**: esegue il test di ipotesi su una serie di esperimenti di Bernulli;
 - **cor.test(x,y)**: esegue il test di correlazione tra due coppie di input;
 - **fisher.test(x)**: esegue il test di Fisher su una tabella di contingenza;
 - **ks.test(x,y)**: esegue il test di Kolmogorov-Smirnov su uno o due coppie di vettori in input;
 - **t.test(x,y=NULL)**: esegue il T-test su uno o due coppie di vettori in input;
 - **wilcox.test(x,y=NULL)**: esegue il test di Wilcoxon su uno o due campioni.
- L'output di questi test statistici è oggetto classe «**htest**» utilizzabile come una lista.
- Comune a tutti i risultati è la componente «**p.value**» che indica la probabilità che l'ipotesi nulla sia vera sulla base delle condizioni del test.

Introduzione a R

Dott. Alaimo Salvatore

Studio 32 – Blocco 2

Email: alaimos@dmi.unict.it

- Introduzione
- Concetti di base
 - Costrutti fondamentali
 - Strutture dati
 - Grafici
 - L'ambiente RStudio
 - Statistica in R
- Concetti Avanzati
 - Modelli Statistici
 - Grafica Avanzata
 - Programmazione ad Oggetti
 - Programmazione Parallela

Come installare pacchetti

- Installare un pacchetto in R è estremamente semplice:
 - Se il pacchetto è nella repository **CRAN**:
 - **install.packages(c("nomepacchetto", ...), dependencies=TRUE)**
 - Se il pacchetto è nella repository **BioConductor**:
 - **source(<http://bioconductor.org/biocLite.R>)**
biocLite("nomepacchetto")

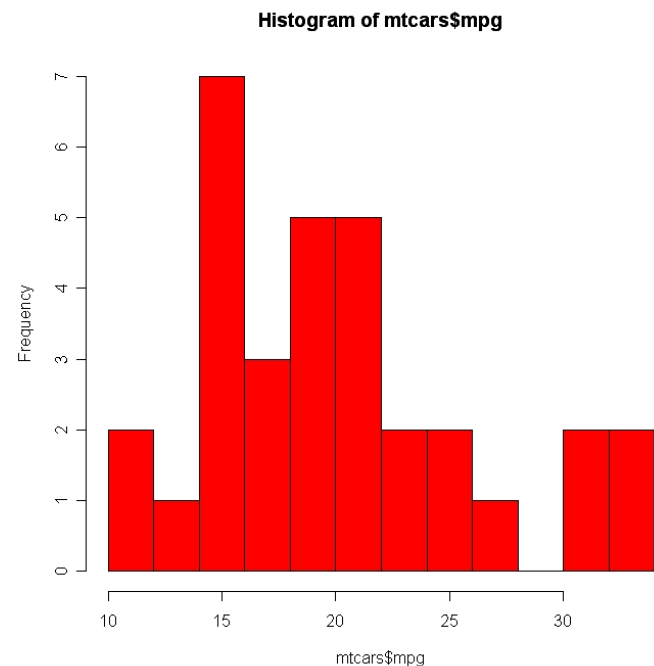
Grafica Avanzata

Parte 7

Istogramma

- Dato un vettore di valori, per creare un istogramma si può usare la funzione «**hist**».
- Sintassi:
 - **hist**(*x*, **breaks**=*"Sturges"*, ...)
 - **x**: è un vettore di valori
 - **breaks**: può essere un numero per indicare il numero di bins, un vettore con i punti di taglio, o il nome di un algoritmo.
- Esempio:

```
hist(mtcars$mpg, breaks=12, col="red")
```



Dot Plot

- È possibile

- Sintassi

- dotplot

-

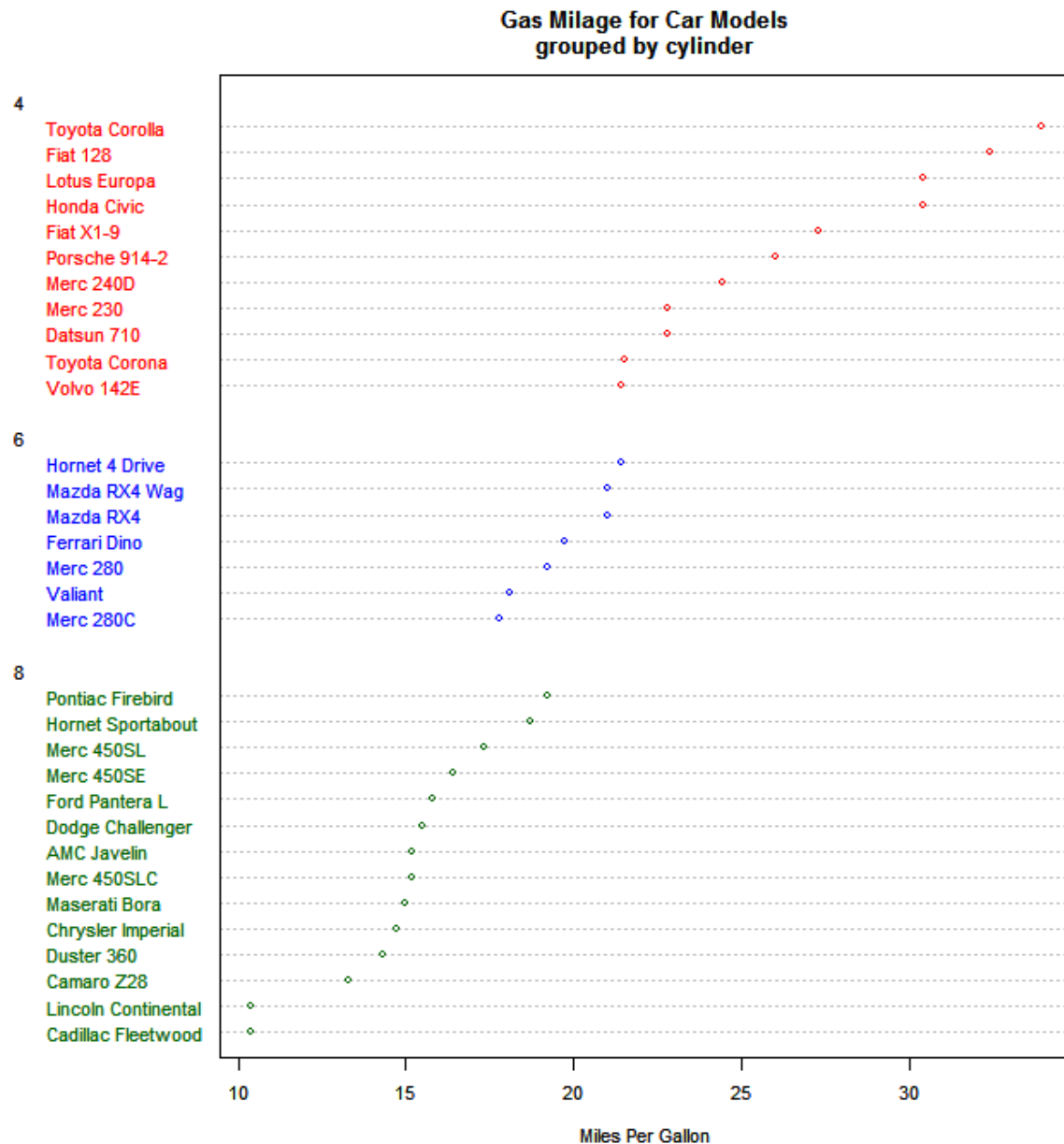
-

-

-

- Esempio

```
x <- mtcars
x$cyl
x$color
x$color
x$color
x$color
dotchart(x$cyl,
```

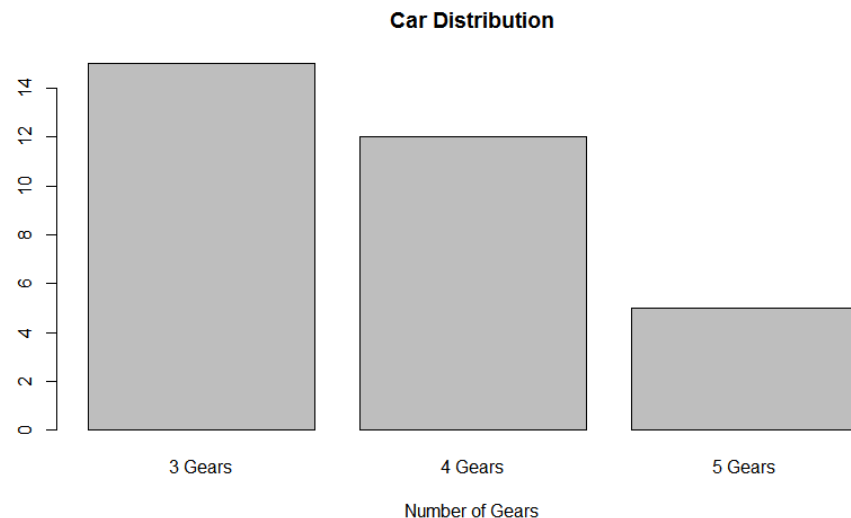


Grafici a barre

- È possibile creare grafici a barre con il comando «**barplot**»
- Sintatti:
 - **barplot**(*height*, **names.arg**=NULL, **horiz**=FALSE, **xlab**=NULL, **ylab**=NULL, ...)
 - **height**: è un vettore o matrice che descrive le barre del grafico
 - **names.arg**: è un vettore di nomi da mostrare per ogni barra
 - **horiz**: è un booleano che indica l'orientamento del grafico
 - **xlab**, **ylab**: sono le etichette da mostrare sull'asse x o y rispettivamente

- Esempio:

```
counts <- table(mtcars$gear)
barplot(counts, main="Car Distribution",
        names.arg=c("3 Gears", "4 Gears", "5 Gears"),
        xlab="Number of Gears")
```

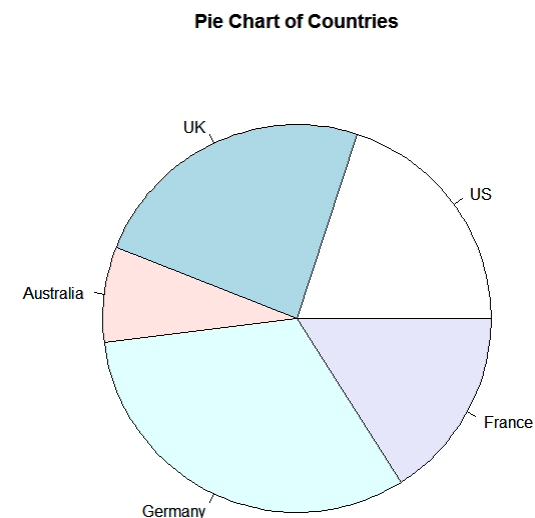


Grafici a torta

- È possibile creare grafici a torta con il comando «**pie**»
- Sintassi:
 - **pie**(*slices*, **labels**=NULL, **main**=NULL, ...)
 - **slices**: è un vettore che descrive le porzioni del grafico
 - **labels**: è un vettore di nomi per ogni porzione del grafico

- Esempio:

```
# Simple Pie Chart  
slices <- c(10, 12, 4, 16, 8)  
lbls <- c("US", "UK", "Australia", "Germany", "France")  
pie(slices, labels = lbls, main="Pie Chart of Countries")
```



BoxPlot

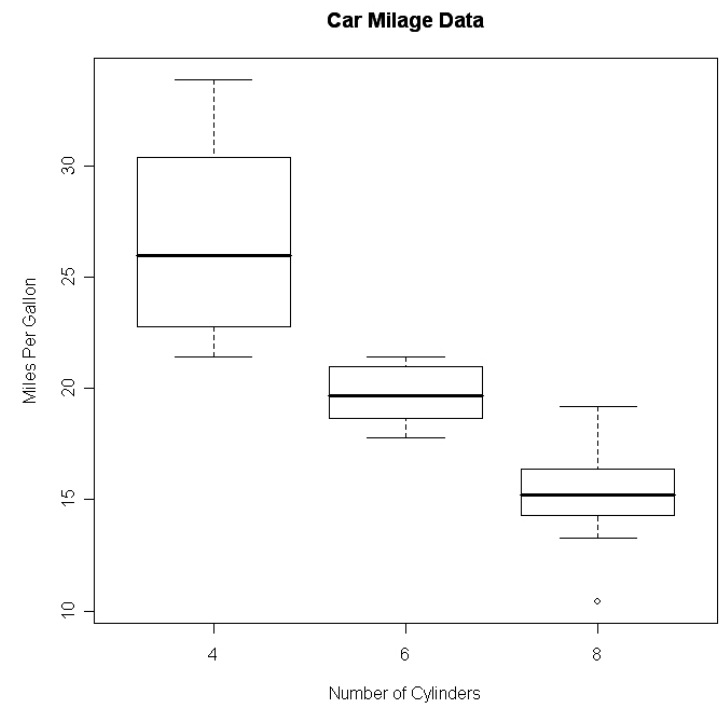
- Sintassi:

- **boxplot**(*formula*, **data**=NULL, ...)

- **formula**: un modello a partire dal quale definire l'insieme dei dati
 - **data**: il dataframe contenente i dati da elaborare

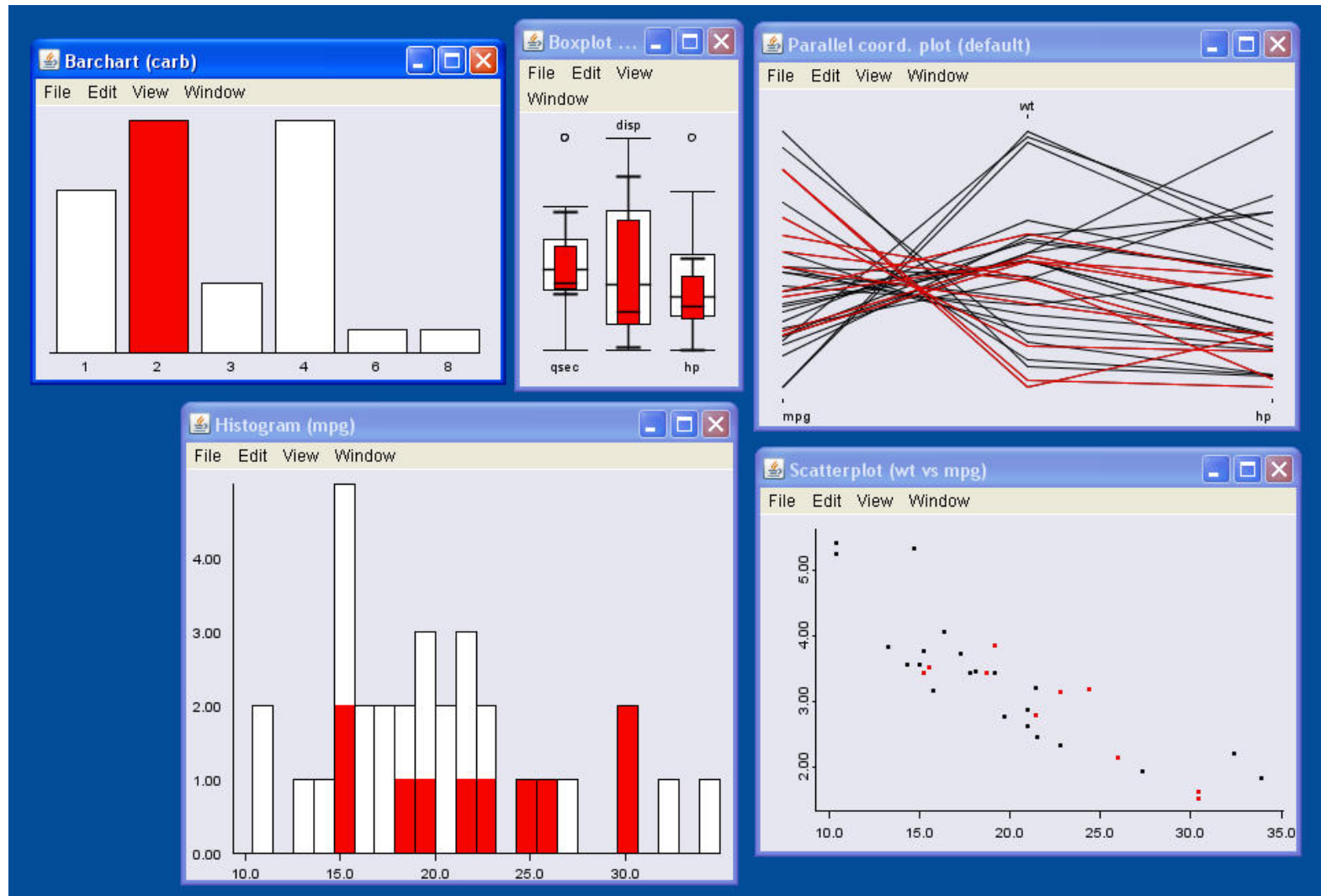
- Esempio:

```
# Boxplot of MPG by Car cylinders
boxplot(mpg~cyl,data=mtcars, main="Car Milage Data",
        xlab="Number of Cylinders", ylab="Miles Per Gallon")
```



Plot Interattivi

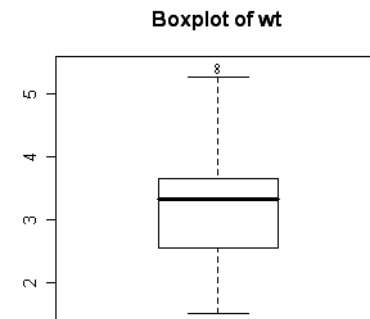
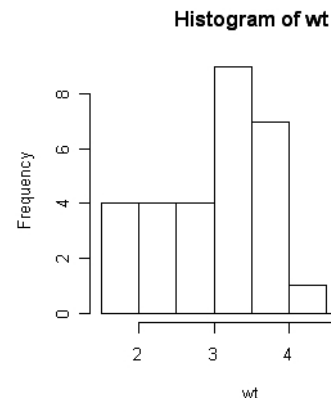
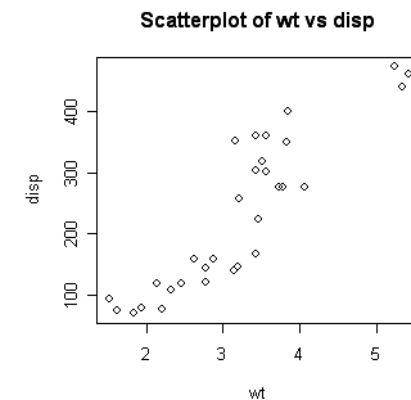
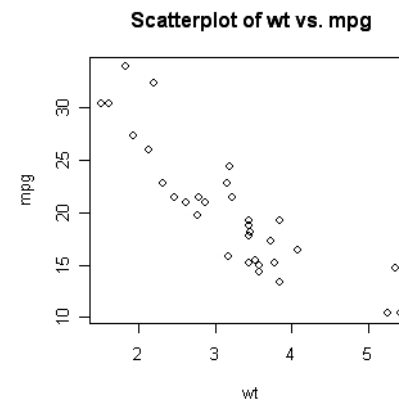
- Tramite il pacchetto «**iplots**» è possibile realizzare plot interattivi.



Combinare grafici

- Per combinare più grafici in una singola immagine si può impostare il parametro «**mfrow**» del sottosistema grafico tramite il comando «**par**». Il parametro è un vettore di due elementi che rappresentano rispettivamente il numero di righe e colonne del grafico.
- Esempio:

```
# 4 figures arranged in 2 rows and 2 columns
attach(mtcars)
par(mfrow=c(2,2))
plot(wt,mpg, main="Scatterplot of wt vs. mpg")
plot(wt,disp, main="Scatterplot of wt vs disp")
hist(wt, main="Histogram of wt")
boxplot(wt, main="Boxplot of wt")
```



Modelli Statistici

Parte 8

- R offre una serie di strumenti altamente versatili e potenti per la rappresentazione di modelli statistici
- Inoltre, esistono molti pacchetti che estendono gli strumenti di base forniti dal linguaggio
 - ...le potenzialità sono quindi infinite
- Ma prima di iniziare...Cos'è un modello statistico?
 - Un modello statistico è la formalizzazione delle relazioni tra variabili sotto forma di equazioni matematiche.
 - In pratica descrive come una o più variabili random sono correlate ad altre variabili.
 - Il modello si dice statistico perché le variabili non sono deterministiche, ma si suppone che ci sia una specifica distribuzione di probabilità che genera i dati osservati

- R fornisce tutti gli strumenti necessari a:
 - Definire semplicemente un modello (Formulae)
 - A partire da dati: (LM, GLM, NLM)
 - calcolare i parametri del modello che meglio li approssimano
 - eseguire delle predizioni attraverso un modello pre-calcolato
 - aggiornare i parametri di un vecchio modello
 - Eseguire test che consentono di stabilire la bontà di un modello (ANOVA, Kolmogorov-Smirnov)
- LM
$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \epsilon$$
- GLM
$$\psi(Y) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \epsilon$$

Formulae

- Permettono di rappresentare simbolicamente modelli statistici di ogni tipo
- La notazione può essere estesa da pacchetti che rendono disponibili nuove caratteristiche
- L'operatore che definisce una formula è «~»
- Come ogni altro oggetto di R anche le formulae hanno una classe di appartenenza: **formula**
 - Il cast da un oggetto (tipicamente stringa) ad una formula si può eseguire tramite il comando «**as.formula**»
- Sintassi:
 - ***response ~ op1 term1 op2 term2 op3 term3 ...***
 - «**response**» definisce le variabili dipendenti del modello
 - «**op**» è un operatore che implica l'inclusione o l'esclusione di una variabile dal modello
 - «**term**» definisce una variabile indipendenti del modello

- Operatori:
 - «+» inclusione di una variabile
 - «-» esclusione di una variabile
 - «:» interazione tra due variabili
 - «*» inclusione di due variabili e della loro interazione
 - « $\wedge n$ » inclusione di tutte le interazioni fino all' n -esimo ordine
 - «**I()**» definisce una porzione del modello in cui gli operatori sono usati nel loro senso aritmetico
- Termini:
 - vettori, matrici, espressioni, factor
 - formulae
 - 1, 0 (nei modelli lineari rappresenta il coefficiente β_0)
- La notazione qui descritta è tipica dei modelli lineari. Per i modelli non lineari quanto scritto nella formula è interpretato direttamente come una equazione.

- Esempi:

- $\text{peso} \sim \text{altezza} + \text{età} - \text{genere}$
- $\text{peso} \sim \text{altezza} + \text{età} * \text{genere}$
 - $\rightarrow \text{peso} \sim \text{altezza} + \text{età} + \text{genere} + \text{età} : \text{genere}$
- $\text{peso} \sim (\text{altezza} + \text{età} + \text{genere}) ^ 2 - \text{altezza} : \text{età}$
 - $\rightarrow \text{peso} \sim (\text{altezza} + \text{età} + \text{genere}) * (\text{altezza} + \text{età} + \text{genere}) - \text{altezza} : \text{età}$
 - $\rightarrow \text{peso} \sim \text{altezza} + \text{età} + \text{genere} + \text{altezza} : \text{genere} + \text{età} : \text{genere}$

- *model* <- **lm**(*formula*, *data*, *subset*, *weights*, *na.action*)
 - Addestra un modello lineare sui dati in input attraverso la descrizione data sotto forma di una formula;
 - Input:
 - «**formula**» una *formula* che descrive il modello da usare per i dati
 - «**data**» un *dataframe* o una *lista* contenente le variabili del modello
 - «**subset**» un *vettore* opzionale che elenca il sottoinsieme di dati da usare per l'addestramento del modello
 - «**weight**» un *vettore* opzionale che fornisce un insieme di pesi iniziale
 - «**na.action**» una stringa opzionale che indica cosa fare se tra i dati ci sono valori NA
 - Valori possibili: «**na.fail**», «**na.omit**», «**na.pass**»
 - L'output consiste in una lista di classe «**lm**» che fornisce le informazioni sull'adattamento del modello ai dati:
 - «**coefficients**» un *vettore* con i coefficienti calcolati dalla funzione
 - «**fitted.values**» un *vettore* con i valori delle variabili indipendenti calcolati tramite il modello
 - «**residuals**» un *vettore* con gli errori di fitting
 - «**rank**» il rango del modello

Modelli Lineari Generalizzati

- ***model <- glm(formula, family, data, subset, weights, na.action)***
 - Addestra un modello lineare generalizzato sui dati in input attraverso la descrizione data sotto forma di una formula;
 - Input:
 - «**formula**» una *formula* che descrive il modello da usare per i dati
 - «**family**» la *link function* usata nel modello
 - binomial, gaussian, Gamma, poisson
 - «**data**» un *dataframe* o una *lista* contenente le variabili del modello
 - «**subset**» un *vettore* opzionale che elenca il sottoinsieme di dati da usare per l'addestramento del modello
 - «**weight**» un *vettore* opzionale che fornisce un insieme di pesi iniziale
 - «**na.action**» una stringa opzionale che indica cosa fare se tra i dati ci sono valori NA
 - valori possibili: «**na.fail**», «**na.omit**», «**na.pass**»
 - L'output consiste in una lista di classe «**glm**» che fornisce le informazioni sull'adattamento del modello ai dati:
 - «**coefficients**» un *vettore* con i coefficienti calcolati dalla funzione
 - «**fitted.values**» un *vettore* con i valori delle variabili indipendenti calcolati tramite il modello
 - «**residuals**» un *vettore* con gli errori di fitting
 - «**rank**» il rango del modello

- ***model <- nls(formula, data, subset, weights, na.action)***
 - Addestra un modello non lineare sui dati in input usando la forma funzionale descritta dalla formula;
 - Input:
 - «**formula**» una *formula* che descrive l'equazione del modello da usare per i dati
 - «**data**» un *dataframe* o una *lista* contenente le variabili del modello
 - «**subset**» un *vettore* opzionale che elenca il sottoinsieme di dati da usare per l'addestramento del modello
 - «**weight**» un *vettore* opzionale che fornisce un insieme di pesi iniziale
 - «**na.action**» una stringa opzionale che indica cosa fare se tra i dati ci sono valori NA
 - valori possibili: «**na.fail**», «**na.omit**», «**na.pass**»
 - L'output consiste in una lista che tra gli elementi ne incorpora uno chiamato «**m**» che consiste in una lista di classe «**nlsModel**» che fornisce le informazioni sull'adattamento del modello ai dati:
 - «**getPars()**» un *vettore* con i coefficienti calcolati dalla funzione
 - «**fitted()**» un *vettore* con i valori delle variabili indipendenti calcolati tramite il modello
 - «**resid()**» un *vettore* con gli errori di fitting

- Tutti i metodi elencati restituiscono oggetti su cui possono essere usate le seguenti funzioni:
 - **`predict(model, data)`**
 - esegue una predizione per nuovi dati usando il modello fornito
 - **`coefficients(model)`**
 - restituisce i coefficienti calcolati per il modello fornito
 - **`summary(model)`**
 - calcola e restituisce un elenco di statistiche riassuntive del modello in input

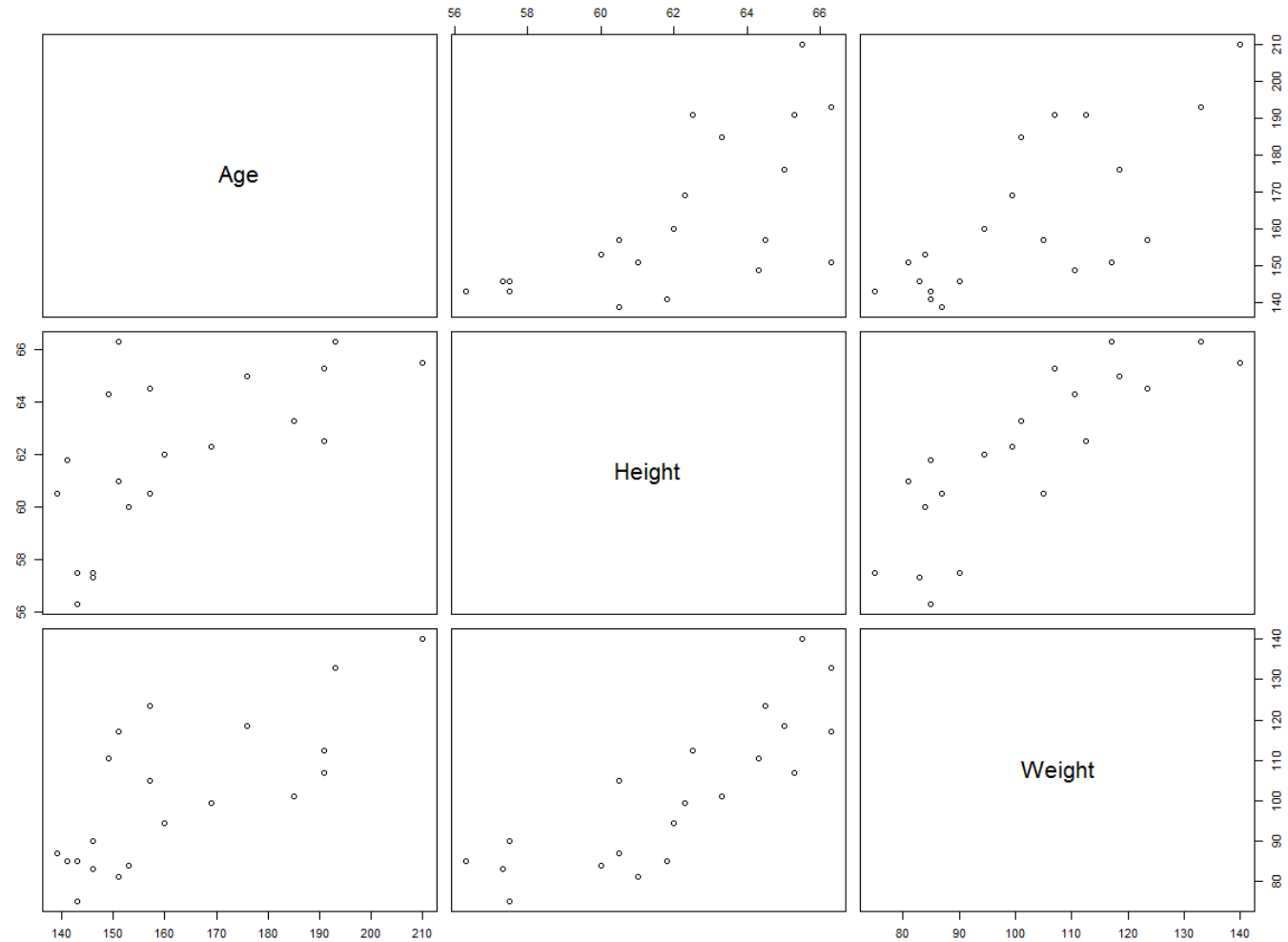
Esempio di Modello Lineare

```
> pop
```

	Age	Height	weight
1	143	56.3	85.0
2	191	62.5	112.5
3	160	62.0	94.5
4	157	64.5	123.5
5	191	65.3	107.0
6	141	61.8	85.0
7	185	63.3	101.0
8	210	65.5	140.0
9	149	64.3	110.5
10	169	62.3	99.5
11	157	60.5	105.0
12	139	60.5	87.0
13	146	57.5	90.0
14	151	66.3	117.0
15	153	60.0	84.0
16	176	65.0	118.5
17	146	57.3	83.0
18	151	61.0	81.0
19	193	66.3	133.0
20	143	57.5	75.0

```
> plot(pop)
```

```
>
```



Esempio di Modello Lineare

```
# Creazione del modello
```

```
> model <- lm(weight ~ Age + Height, data=pop)
> summary(model)
```

Call:

```
lm(formula = weight ~ Age + Height, data = pop)
```

Residuals:

Min	1Q	Median	3Q	Max
-15.073	-6.688	1.223	7.420	14.675

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)	
(Intercept)	-168.1133	45.2984	-3.711	0.00173	**
Age	0.3061	0.1348	2.271	0.03642	*
Height	3.5484	0.9062	3.916	0.00111	**

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 9.477 on 17 degrees of freedom

Multiple R-squared: 0.763, Adjusted R-squared: 0.7351

F-statistic: 27.37 on 2 and 17 DF, p-value: 4.843e-06

```
> coefficients(model)
```

(Intercept)	Age	Height
-168.113277	0.306143	3.548435

```
> predict(model, list(Height=70, Age=220))
```

```
1
147.6286
```

Programmazione ad Oggetti

Parte 9

Classi e Metodi

- Alcune ragioni per cui programmare ad oggetti è una buona pratica:
 - aumento della produttività
 - facilita il mantenimento del codice
 - codice riutilizzabile
 - incapsula la rappresentazione degli oggetti
 - specializza comportamento delle funzioni allo specifico oggetto
- È importante ricordare che:
 - tutto in R è un **oggetto** ed ha, quindi, una **classe** di appartenenza
 - Una **classe** rappresenta la definizione di un oggetto (metodi e proprietà)
 - Un **metodo** è una funzione che esegue una qualche computazione su un oggetto di una specifica classe.
- In R una **funzione generica** è un particolare tipo di funzione che determina la classe dei suoi argomenti e seleziona automaticamente il metodo più appropriato da eseguire (una funzione associata ad una collezione di metodi).

Le classi in R

- A differenza di altri linguaggi OOP, in R ci sono 3 tipi di classi e quindi tre stili di rappresentazione:
 - **Classi S3**: vecchio stile, veloci ma molto informali (Non li studieremo)
 - **Classi S4**: nuovo stile, rigoroso e formale
 - **ReferenceClasses**: classi in stile Java o C++ in contrasto con lo stile funzionale di R
- L'approccio S3 è molto semplice da implementare e non ha requisiti formali sulla struttura degli oggetti. Basta assegnare il nome di una classe ad un qualunque oggetto R per definirla.
- L'approccio S4 richiede invece che di una classe ne sia prima data una definizione rigorosa. Ogni oggetto valido di una classe S4 ne dovrà soddisfare obbligatoriamente tutti i requisiti.
- S4 sta lentamente sostituendo S3 anche se quest'ultimo è ancora molto usato
 - Repository come bioconductor obbligano tutti i nuovi pacchetti ad usare classi di tipo S4 o ReferenceClasses
- Tutto ciò che vedremo richiede che il pacchetto «**methods**» sia caricato.

Classi S4

- Le informazioni nelle classi S4 sono organizzate in «**slot**». Ogni slot ha un nome e un preciso tipo (classe).
- Gli slot sono uno dei principali vantaggi delle classi S4.
 - La classe dei dati nello slot deve corrispondere a quella nella definizione.
 - Non è quindi possibile avere un numero in uno slot che ha tipo stringa.

Function	Description
setClass()	Crea una nuova classe
setMethod()	Crea un nuovo metodo
setGeneric()	Crea una funzione generica
new()	Costruisce un nuovo oggetto
getClass()	Restituisce la definizione di una classe
getMethod()	Restituisce la definizione di un metodo
getSlots()	Restituisce il nome e la classe di ogni slot
@	Restituisce o rimpiazza il contenuto di uno slot
validObject()	Controlla se un oggetto è valido

Creare/Instanziare una classe

- Sintassi:

- **setClass**(*Class, slots, contains, prototype, validity*)

- «**Class**»: il nome della classe da creare
 - «**slots**»: una lista con i nomi e le classi degli slot
 - «**contains**»: un vettore con i nomi delle superclassi da ereditare
 - «**prototype**»: una lista con i valori di default per ogni slot
 - «**validity**»: una funzione che restituisce TRUE se l'input è un oggetto valido

- *my.object* <- **new**(*Class, ...*)

- «Class»: il nome della classe da istanziare
 - «...»: i parametri che richiede il costruttore o una lista degli slot con i relativi valori

Aggiungere metodi

- Sintassi:
 - **setMethod**(*f, signature, definition*)
 - «**f**»: una funzione generica o il nome di una qualsiasi funzione
 - «**signature**»: un vettore con le classi a cui associare il metodo
 - «**definition**»: la funzione che sarà richiamata
 - **setGeneric**(*name, def, valueClass*)
 - «**name**»: Il nome della funzione generica da creare
 - «**def**»: una funzione opzionale che definisce la funzione di default
 - «**valueClass**»: un vettore opzionale di nomi di classi che costituiscono l'output del generico
 - **callNextMethod()**
 - Chiama il metodo di una delle classi ereditate. Per essere più precisi, il metodo scelto corrisponde a quello che sarebbe stato eseguito se quello corrente non fosse stato definito.

- Alcuni esempi di metodi definibili nelle classi:
 - `setMethod(f="initialize", signature=..., definition=function (.Object, ...) {
 Object <- callNextMethod(.Object, ...)

 ...
 return (Object)
})`
 - `setMethod(f="show", signature=..., definition=function (object) { ... })`
 - `setMethod(f="[, signature=..., definition=function (x,i,j) { ... })`
 - `setMethod(f="[<=", signature=..., definition=function (x,i,j,value) { ... })`
- Per rendere lecito un cast da un tipo a un altro:
 - **setAs**(*from*, *to*, *def*)
 - «**from**»: il tipo di partenza
 - «**to**»: il tipo a cui è possibile fare il cast
 - «**def**»: la funzione che esegue la conversione degli oggetti

Reference Classes

- Sono classi i cui oggetti hanno campi che possono essere acceduti per riferimento nello stile dei linguaggi OOP classici quali Java e C++
- Le computazioni con questi oggetti invocano metodi e/o impostano/restituiscono campi che sono presenti direttamente all'interno dell'oggetto stesso.
- La computazione può potenzialmente modificare un oggetto a differenza di quanto accade con gli oggetti di tipo S4.
- A differenza degli oggetti S4, le Reference Classes non rispettano il modello di programmazione funzionale adottato da R.

Function	Description
setRefClass()	Definisce una classe e ne restituisce la funzione generatrice
getRefClass()	Restituisce la funzione generatrice di una classe
\$	Simbolo usato per accedere ai metodi o ai campi della classe
\$callSuper()	« super. » (interno)
\$copy()	Crea una copia dell'oggetto
\$field()	Accede o Setta un campo
\$export()	Esegue un cast sull'oggetto
\$import()	Importa campi da un altro oggetto
\$show()	« toString » dell'oggetto
\$help()	Visualizza l'help
\$accessors()	Definisce automaticamente uno o più metodi get/set
.self	Riferimento a se stesso (interno)
.refClassDef	Riferimento alla classe (interno)

Creare/Instanziare una classe

- Sintassi:

- `myClass <- setRefClass(Class, fields, contains, methods)`

- «**Class**»: il nome della classe da creare

- «**fields**»: una lista di campi con i rispettivi tipi

- «**contains**»: un vettore con i nomi delle superclassi da ereditare

- «**methods**»: una lista con tutti i metodi dell'oggetto

- `my.object <- myClass(...)`

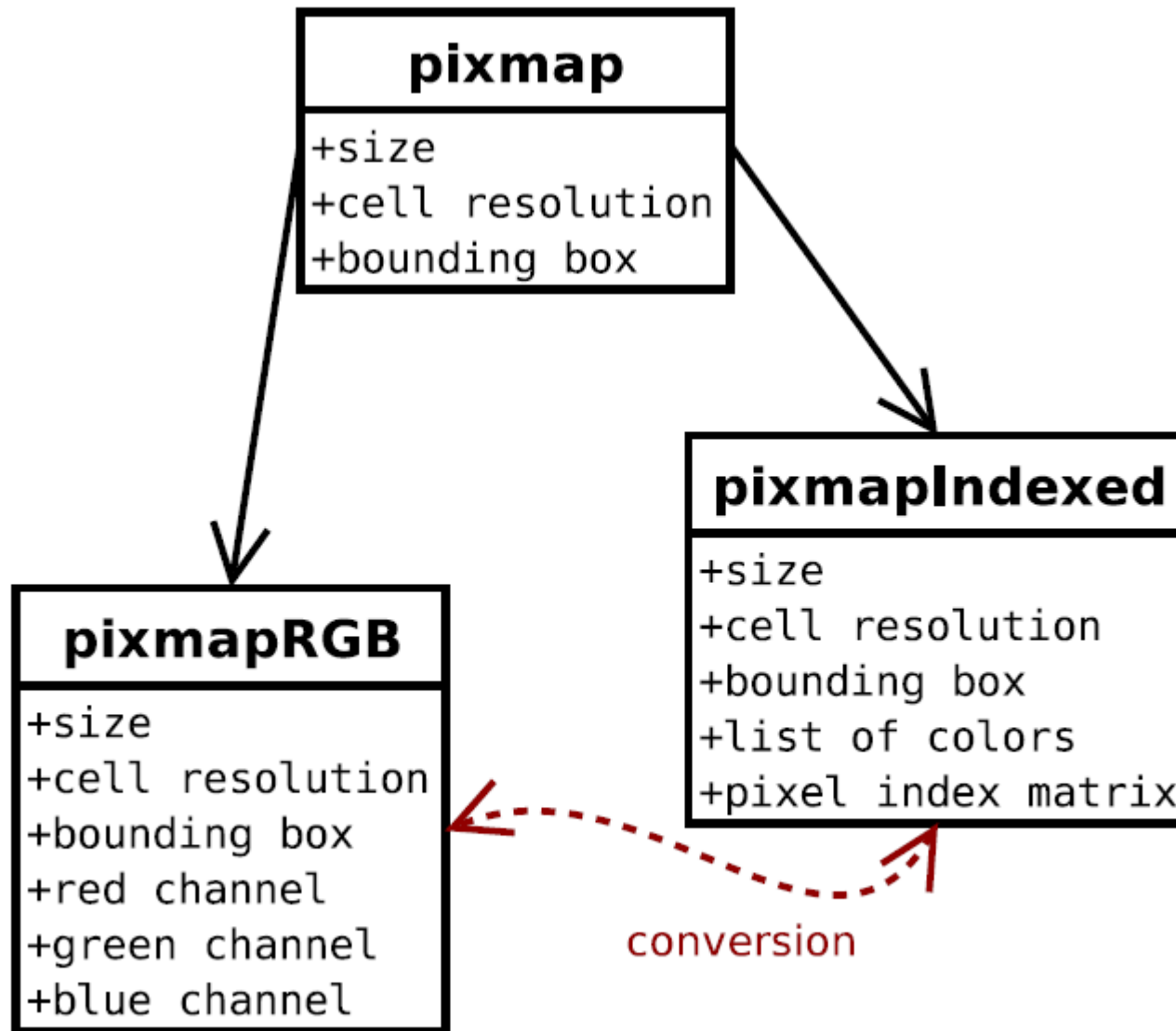
- «**myClass**»: la funzione generatrice della classe da istanziare

- «...»: i parametri che richiede il costruttore o una lista dei campi con i relativi valori

Segnatura dei metodi predefiniti

- *object***\$initialize**(...) – Costruttore della classe
- *object***\$callSuper**(...) – Interno – Chiama il metodo ereditato dalla superclasse
- *object***\$copy**(shallow=FALSE) – Crea una copia dell'oggetto. Se shallow è TRUE non copia gli oggetti contenuti all'interno
- *object***\$field**(name, value) – Con un solo argomento restituisce il valore di un campo; Con due imposta il nuovo valore di un campo
- *object***\$export**(Class) – Esegue e restituisce un cast sull'oggetto corrente
- *object***\$import**(value) – Importa «**value**» nell'oggetto corrente
- *object***\$show**() – Stampa l'oggetto corrente
- *class***\$help**(topic) – Visualizza l'help su un metodo della classe
- *class***\$accessors**(...) – Costruisce automaticamente i getter e i setter per una lista di campi
- *class***\$methods**(...) – Senza parametri restituisce la lista dei metodi; Se richiamato con una lista di metodi, li aggiunge alla definizione della classe

Un semplice esempio – Immagini con Mappa di Pixel



Un semplice esempio – Immagini con Mappa di Pixel

```
setClass("pixmap",
  slots=list(size="integer",
             cellres="numeric",
             bbox="numeric"),
  prototype=list(size=integer(2),
                 cellres=c(1,1),
                 bbox=numeric(4)))

setGeneric("plot")

setMethod("show", "pixmap",
  function (object) {
    cat("Pixmap image \n")
    cat("  Type:", class(object), "\n")
    cat("  Size:", paste(object@size, collapse="x"), "\n")
    cat("  Resolution:", paste(object@cellres, collapse="x"), "\n")
    cat("  Bounding box:", object@bbox, "\n")
    cat("\n")
  })
```

Un semplice esempio – Immagini con Mappa di Pixel

```
setClass("pixmapRGB",
  slots=list(red="matrix",
             green="matrix",
             blue="matrix"),
  contains="pixmap")

setClass("pixmapIndexed",
  slots=list(index="matrix",
             col="character"),
  contains="pixmap")

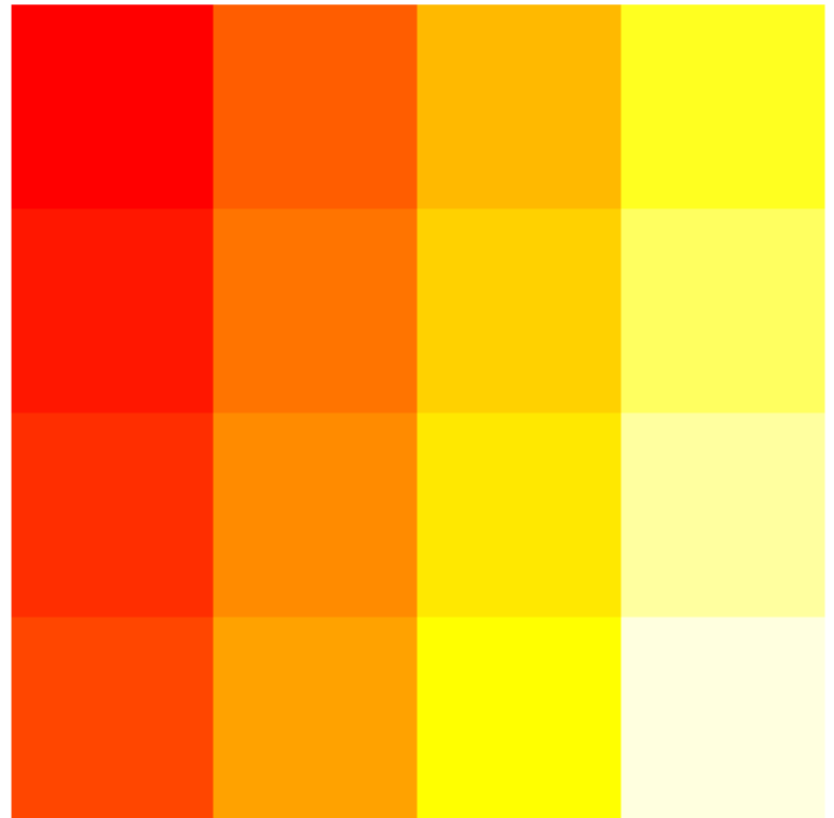
setMethod("plot", "pixmapIndexed",
  function (x, y, xlab="", ylab="", axes=FALSE, asp=1, ...) {
    image(t(x@index[x@size[1]:1,]), col=x@col,
          xlab=xlab, ylab=ylab, axes=axes, asp=asp, ...)
  })

setAs("pixmapRGB", "pixmapIndexed", function(from, to) {
  x  <- rgb(from@red, from@green, from@blue)
  col <- unique(x)
  x  <- match(x, col)
  x  <- matrix(x, nrow=from@size[1], ncol=from@size[2])
  return (new("pixmapIndexed", size=from@size, index=x, col=col))
})

setMethod("plot", "pixmapRGB",
  function (x, y, ...) {
    plot(as(x, "pixmapIndexed"), ...)
  })
```

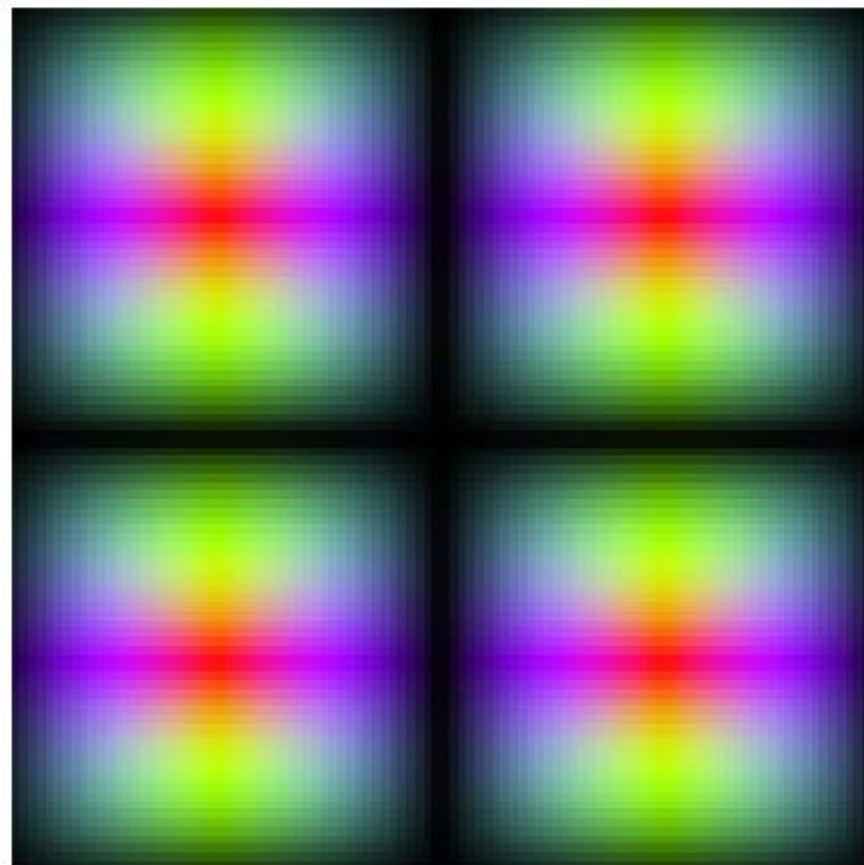
Un semplice esempio – Immagini con Mappa di Pixel

```
> mypic <- new("pixmapIndexed")
> mypic@index <- matrix(1:16, 4)
> mypic@col <- heat.colors(16)
> mypic@size <- dim(mypic@index)
>
> print(mypic)
Pixmap image
Type: pixmapIndexed
Size: 4x4
Resolution: 1x1
Bounding box: 0 0 0 0
>
> plot(mypic)
```



Un semplice esempio – Immagini con Mappa di Pixel

```
> x <- seq(-3, 3, length=100)
> z1 <- outer(x, x, function(x, y) abs(sin(x)*sin(y)))
> z2 <- outer(x, x, function(x, y) abs(sin(2*x)*sin(y)))
> z3 <- outer(x, x, function(x, y) abs(sin(x)*sin(2*y)))
>
> mypic <- new("pixmapRGB",size=dim(z1),red=z1,green=z2,blue=z3)
>
> print(mypic)
Pixmap image
  Type: pixmapRGB
  Size: 100x100
  Resolution: 1x1
  Bounding box: 0 0 0 0
>
> plot(mypic)
```



Un semplice esempio – Immagini con Mappa di Pixel

```
pixmap <- setRefClass("pixmap", fields=list(
  size="integer", cellres="numeric", bbox="numeric"
), methods=list(
  initialize=function(size=integer(2),cellres=c(1,1),bbox=numeric(4), ...) {
    callSuper(size=size, cellres=cellres, bbox=bbox, ...)
  },
  show=function() {
    cat("Pixmap image \n")
    cat("  Type:", class(.self), "\n")
    cat("  Size:", paste(size, collapse="x"), "\n")
    cat("  Resolution:", paste(cellres, collapse="x"), "\n")
    cat("  Bounding box:", bbox, "\n")
    cat("\n")
  }
))

pixmap$accessors("size", "cellres", "bbox")
```

Un semplice esempio – Immagini con Mappa di Pixel

```
pixmapIndexed <- setRefClass("pixmapIndexed", fields=list(
  index="matrix", col="character"
), contains="pixmap", methods=list(
  plot=function(xlab="", ylab="", axes=FALSE, asp=1, ...) {
    image(t(index[size[1]:1,]), col=col,
          xlab=xlab, ylab=ylab, axes=axes, asp=asp, ...)
  }
))
```

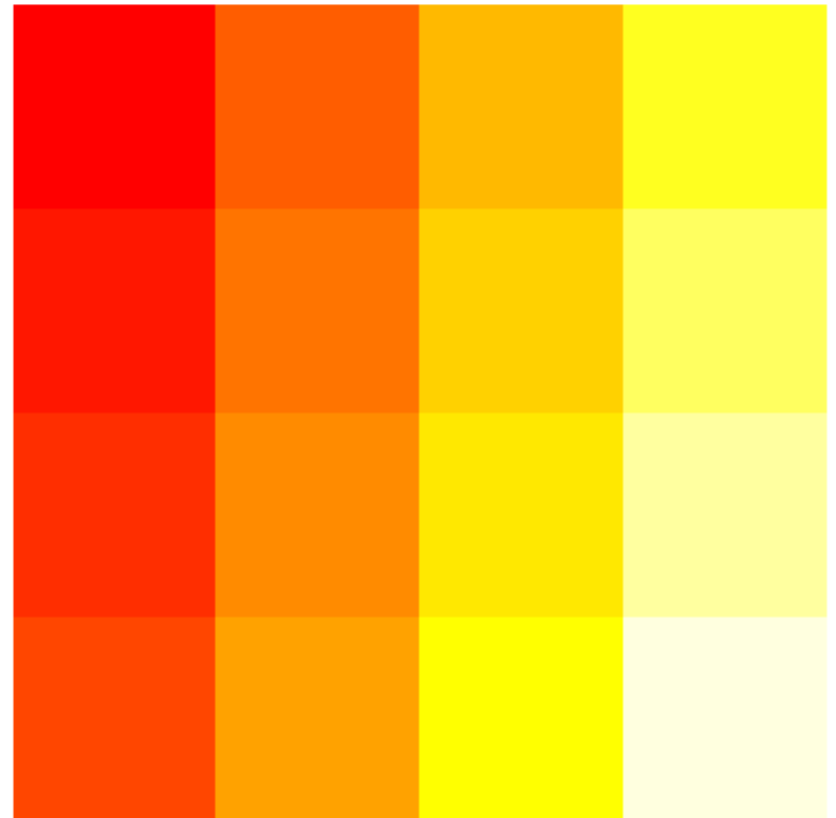
```
pixmapIndexed$accessors("index", "col")
```

```
pixmapRGB <- setRefClass("pixmapRGB", fields=list(
  red="matrix", green="matrix", blue="matrix"
), contains="pixmap", methods=list(
  export=function(Class) {
    if (Class == "pixmapIndexed") {
      x <- rgb(.self$red, .self$green, .self$blue)
      col <- unique(x)
      x <- match(x, col)
      x <- matrix(x, nrow=size[1], ncol=size[2])
      return (pixmapIndexed(size=size, index=x, col=col))
    }
    callSuper(Class)
  },
  plot=function(x, y, ...) {
    export("pixmapIndexed")$plot()
  }
))
```

```
pixmapRGB$accessors("red", "green", "blue")
```

Un semplice esempio – Immagini con Mappa di Pixel

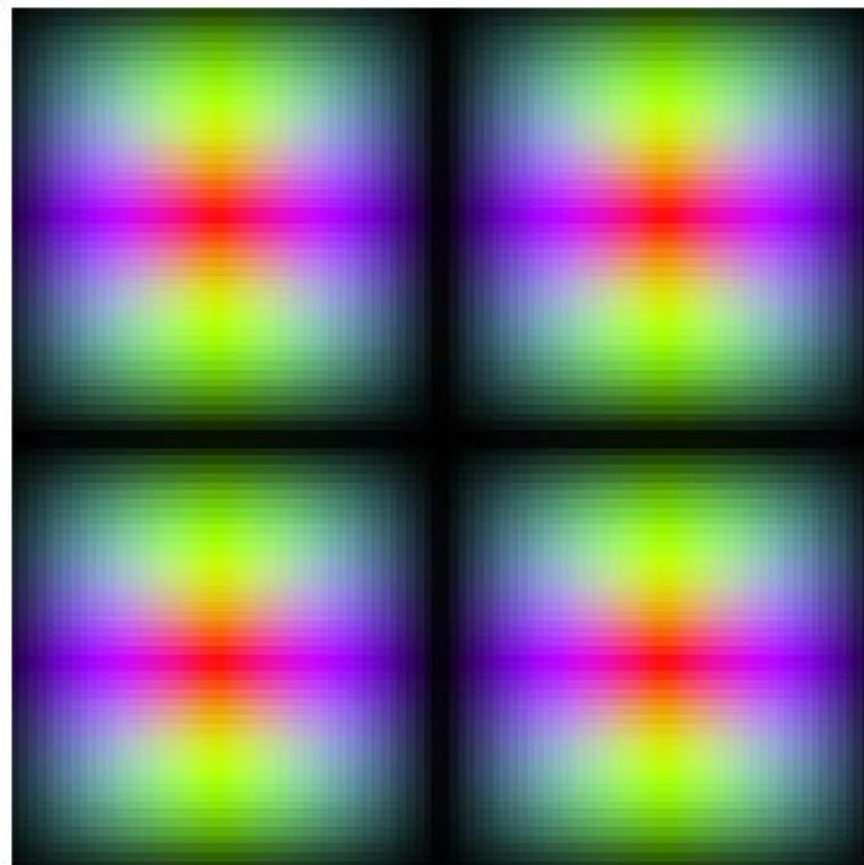
```
> mypic <- pixmapIndexed(index=matrix(1:16, 4),  
+                         col=heat.colors(16))  
> mypic$setSize(dim(mypic$getIndex()))  
>  
> print(mypic)  
Pixmap image  
  Type: pixmapIndexed  
  Size: 4x4  
  Resolution: 1x1  
  Bounding box: 0 0 0 0  
  
>  
> mypic$plot()  
>
```



Un semplice esempio – Immagini con Mappa di Pixel

```
> x <- seq(-3, 3, length=100)
> z1 <- outer(x, x, function(x, y) abs(sin(x)*sin(y)))
> z2 <- outer(x, x, function(x, y) abs(sin(2*x)*sin(y)))
> z3 <- outer(x, x, function(x, y) abs(sin(x)*sin(2*y)))
>
> mypic <- pixmapRGB(size=dim(z1),red=z1,green=z2,blue=z3)
>
> print(mypic)
Pixmap image
  Type: pixmapRGB
  Size: 100x100
  Resolution: 1x1
  Bounding box: 0 0 0 0

>
> mypic$plot()
>
> print(mypic$export("pixmapIndexed"))
Pixmap image
  Type: pixmapIndexed
  Size: 100x100
  Resolution: 1x1
  Bounding box: 0 0 0 0
```



Programmazione Parallela

Parte 10

- Prima di poter parlare di **programmazione parallela**, è necessario parlare di **vettorizzazione** del codice.
- Infatti lo scopo principale della parallelizzazione è lo speed-up del codice in modo da sfruttare al massimo tutte le capacità del sistema di calcolo.
- In un ambiente come R, però, un primo speed-up significativo si ottiene eseguendo una **vettorizzazione** del codice

Prima di iniziare

- Un tipico codice non vettorizzato che esegue calcoli su matrici è:

```
for (i in ...) {  
  for (j in ...) {  
    my.matrix <- func(mymatric, i, j)  
  }  
}
```

- Il principale svantaggio di questo codice è che la maggior parte del tempo si perde nelle iterazioni e nel copiare le variabili
 - ... piuttosto che fare calcoli
- Infatti in R non esiste il passaggio per riferimento (paradigma funzionale) e ogni operazione che modifica una qualche variabile richiede che questa venga prima copiata e poi modificata
 - ...l'unica eccezione sono le Reference Classes

- Quando si sviluppa codice R occorre seguire alcune semplici regole:
 - Evitare cicli che processano un elemento per iterazione
 - Usare funzioni che processano intere strutture dati in una singola chiamata, eliminando i cicli e il processo di copia
 - Processare vettori interi piuttosto che singoli elementi (vettorizzazione)
 - Usare le funzioni della serie «**apply**» per semplificare la vettorizzazione, ovvero il processing di intere righe, colonne o liste
 - Programmazione Funzionale!!

Le funzioni «apply»

- La più semplice funzione per vettorizzare il codice in R è la funzione «**apply**» che opera sulle matrici:

- **apply(M,1,fun)** – applica fun alle righe di M
- **apply(M,2,fun)** – applica fun alle colonne di M

- Sintassi:

- *result* <- **apply**(*X*, *MARGIN*, *FUN*, ...)

- «**x**»: un array o una matrice o un dataframe (da convertire in matrice)
- «**MARGIN**»: la dimensione a cui applicare FUN (es. 1 righe, 2 colonne per una matrice)
- «**FUN**»: la funzione da applicare
- «...»: lista opzionale di argomenti da passare alla funzione

- Esempio:

```
> M <- matrix(runif(20), nrow=5, ncol=4)
> apply(M, 1, median)
[1] 0.4681438 0.2906495 0.2682414 0.6058773 0.3938608
> apply(M, 2, median)
[1] 0.4269077 0.2745305 0.5677378 0.3987907
> |
```

Altre funzioni «apply»

- «**apply**» non funziona sulle liste per questo è disponibile la funzione «**lapply**»
- Sintassi:
 - *result <- lapply(X, FUN, ..., simplify, USE.NAMES)*
 - «**X**»: un vettore o una lista o un oggetto che può essere convertito in lista
 - «**FUN**»: la funzione da applicare ad ogni elemento di X
 - «...»: una lista opzionale di argomenti da passare alla funzione
 - «**simplify**»: se TRUE cerca di trasformare l'output in un vettore o una matrice altrimenti restituisce una lista
 - «**USE.NAMES**»: se TRUE e X è un vettore di stringhe usa X come vettore dei nomi per la lista dei risultati
 - Esempio:

```
> x <- list(a = 1:10, beta = exp(-3:3))
> lapply(x, mean)
$a
[1] 5.5

$beta
[1] 4.535125
```

Altre funzioni «apply»

- Per il caso speciale «**lapply(..., simplify=TRUE, USE.NAMES=TRUE)**» è stata definita la funzione «**sapply**»

- Sintassi:

- *result* <- **sapply**(*X*, *FUN*, ...)

- «**X**»: un vettore o una lista o un oggetto che può essere convertito in lista
- «**FUN**»: la funzione da applicare ad ogni elemento di X
- «...»: una lista opzionale di argomenti da passare alla funzione

- Esempio:

```
> x <- list(a = 1:10, beta = exp(-3:3))
> x
$a
 [1]  1  2  3  4  5  6  7  8  9 10

$beta
 [1] 0.04978707 0.13533528 0.36787944 1.00000000 2.71828183 7.38905610 20.08553692

> y <- sapply(x, mean)
> y
      a      beta
5.500000 4.535125
```

Altre funzioni «apply»

- Altre due funzioni che permettono di vettorizzare facilmente il codice sono:
 - *result* <- **mapply**(*FUN*, ...)
 - Applica parallelamente «**FUN**» a più liste
 - *result* <- **tapply**(*X*, *INDEX*, *FUN*, ...)
 - Applica «**FUN**» ad un vettore o lista o oggetto «**X**» suddividendolo in gruppi specificati dalla variabile categoriale «**INDEX**»
- Esempi:

```
> mapply(rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4
```

```
> groups
[1] 15 11 11 13 19 12 9 13 15 14
Levels: 9 11 12 13 14 15 19
> tapply(groups, groups, length)
 9 11 12 13 14 15 19
1  2  1  2  1  2  1
```

Parallelizzazione del codice

- Per parallelizzare un codice vettorizzato, tutte le funzioni necessarie sono disponibili nel pacchetto «**parallel**»
- Tutte le funzioni *apply* hanno un corrispettivo nel pacchetto *parallel*:
 - `apply(X, MARGIN, FUN, ...)` → `parApply(cl, X, MARGIN, FUN, ...)`
 - `apply(X, 1, FUN, ...)` → `parRapply(cl, X, FUN, ...)`
 - `apply(X, 2, FUN, ...)` → `parCapply(cl, X, FUN, ...)`
 - `lapply(X, FUN, ...)` → `parLapply(cl, X, fun, ...)`
 - `sapply(X, FUN, ...)` → `parSapply(cl, X, fun, ...)`
 - `mapply(FUN, ...)` → `clusterMap(cl, fun, ..., SIMPLIFY=FALSE)`
- Hanno la stessa sintassi della corrispondente versione non parallela con l'aggiunta di un parametro «cl» che rappresenta un oggetto che descrive il cluster computazionale

Oggetto cluster

- Per costruire un oggetto cluster si utilizza la funzione «**makeCluster**»
 - la sintassi completa è disponibile nell'help del pacchetto parallel
 - per una macchina con processore multi-core:
 - **cl <- makeCluster(detectCores())**
- Al termine delle operazioni è necessario distruggere l'oggetto cluster con la funzione «**stopCluster**»:
 - **stopCluster(cl)**
- Se un cluster usa il generatore di numeri random è necessario inizializzarlo con il comando «**clusterSetRNGStream(cl, iseed)**»
 - **clusterSetRNGStream(cl, 123)** # imposta un seed riproducibile
 - **clusterSetRNGStream(cl, NULL)** # imposta un seed casuale

Esempio

```
> cl <- makeCluster(detectCores())
>
> M <- matrix(runif(20), nrow=5, ncol=4)
> x <- list(a = 1:10, beta = exp(-3:3))
>
> parApply(cl=cl, M, 1, median)
[1] 0.4681438 0.2906495 0.2682414 0.6058773 0.3938608
> parRapply(cl=cl, M, median)
[1] 0.4681438 0.2906495 0.2682414 0.6058773 0.3938608
>
> parApply(cl=cl, M, 2, median)
[1] 0.4269077 0.2745305 0.5677378 0.3987907
> parCapply(cl=cl, M, median)
[1] 0.4269077 0.2745305 0.5677378 0.3987907
>
> parLapply(cl=cl, x, mean)
$a
[1] 5.5

$beta
[1] 4.535125

> parSapply(cl=cl, x, mean)
      a      beta
5.500000 4.535125
>
> clusterMap(cl, rep, 1:4, 4:1)
[[1]]
[1] 1 1 1 1

[[2]]
[1] 2 2 2

[[3]]
[1] 3 3

[[4]]
[1] 4

>
> stopCluster(cl)
```


- Un'altra strategia per evitare cicli ed ottenere uno speed-up significativo del codice consiste nell'usare a pieno le caratteristiche della programmazione funzionale.
- Supponiamo di definire un predicato (funzione) ***f***, che restituisce ***TRUE*** per gli elementi che si desidera elaborare, ***FALSE*** altrimenti.
- Filter(***f***, ***x***) – restituisce gli elementi del vettore ***x*** per cui ***f*** è ***TRUE***.
 - Il parametro opzionale «***right***» se ***TRUE*** impone alla funzione di analizzare gli elementi dall'ultimo al primo, se ***FALSE*** (default) dal primo all'ultimo.

Programmazione Funzionale

- Supponendo di avere una funzione **$f(x,y)$** qualsiasi
- `Reduce(f, x)` – è un modo per iterare una lista o vettore **x** applicando una funzione **f** a risultati successivi.
- Supponiamo che $x = x_1, x_2, x_3, \dots$
- `Reduce(f,x)` calcola:
 - $f(x_1, x_2)$
 - $f(f(x_1, x_2), x_3)$
 - $f(f(f(x_1, x_2), x_3), x_4)$
- Il risultato di *default* è il valore dell'ultima chiamata a **f** , ma impostando il parametro «***accumulate=TRUE***» tutti i risultati intermedi saranno restituiti.

Programmazione Funzionale

```
x <- as.double(1:1000000)
```

```
f1 <- function(x) {  
  s <- 0  
  for (i in 1:length(x)) {  
    s <- s + x[i]  
  }  
  print(s)  
}
```

```
f2 <- function(x) {  
  s <- Reduce("+", x)  
  print(s)  
}
```

```
|
```

```
print(system.time(f1(x)))
```

```
print(system.time(f2(x)))
```

```
> source('D:/[redacted]  
[1] 500000500000  
   user  system elapsed  
   0.51    0.00    0.55  
[1] 500000500000  
   user  system elapsed  
   0.25    0.00    0.25  
> |
```

- **Find**(f , x) – trova il **primo** elemento di x che soddisfa il predicato f
- **Find**(f , x , right=TRUE) – trova l'**ultimo** elemento di x che soddisfa il predicato f
- **Position**(f , x) – trova la posizione del **primo** elemento di x che soddisfa il predicato f
- **Position**(f , x , right=TRUE) – trova la posizione dell'ultimo elemento di x che soddisfa il predicato f
- **Negate**(f) – calcola la negazione di un predicato f
- **Map**(f , ...) – un wrapper di mapply con «**simplify=FALSE**»

THE END