Corso introduttivo al linguaggio di programmazione R

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

Dipartimento di Scienze dell'Informazione

Università degli Studi di Milano

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Introduzione ai linguaggi di programmazione

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Caratteristiche dei linguaggi di programmazione

- Sono analoghi ai linguaggi naturali, con la differenza che vengono usati per comunicare con una macchina

Come i linguaggi naturali sono caratterizzati dalle seguenti componenti:

• Insieme di simboli (*alfabeto*) e di parole (*dizionario*) che possono essere usati per formare le frasi del linguaggio

• Insieme delle regole grammaticali (*sintassi*) per definire le frasi corrette composte dalle parole del linguaggio

• Significato (*semantica*) delle frasi del linguaggio

• Per utilizzare correttamente un linguaggio è necessario conoscerne la *pragmatica* (ad es: quali frasi è opportuno usare a seconda del contesto).

• *I linguaggi di programmazione, a differenza di linguaggi naturali, non devono essere ambigui e devono essere formalizzati* (definiti in maniera non equivocabile).

Linguaggi di programmazione: strumenti per comunicare ad una macchina come risolvere un problema.

- Strumenti per la comunicazione uomo-macchina
- Permettono di esprimere e rappresentare *programmi = algoritmi + strutture dati* comprensibili ed eseguibili da una macchina

Linguaggi macchina

- Linguaggi immediatamente comprensibili per una macchina:
  - Istruzioni e dati sono sequenze di numeri binari
  - Le istruzioni operano direttamente sull' hardware (registri, locazioni di memoria, unità fisiche di I/O del calcolatore)
  - Sono specifici per un determinato processore o famiglia di processori
  - Assumono il modello computazionale di Von Neumann

Esempio:

Calcolo della somma S di due numeri A e B

Linguaggio macchina	Linguaggio assembly
00000010101111001010	LOAD A
00000010111111001000	ADD B
00000011001110101000	STORE S

Un *linguaggio assembly* è la forma simbolica di un linguaggio macchina: si usano nomi al posto dei codici binari per le operazioni e locazioni di memoria delle macchine.

Linguaggi a basso ed alto livello

- Linguaggi assembly e macchina sono *linguaggi a basso livello*
- *I linguaggi ad alto livello* permettono di scrivere programmi con un linguaggio piu' vicino a quello naturale

Esempio:

"Stampa sullo schermo la somma fra C ed il prodotto di A e B":

Linguaggio ad alto livello (C++):

```
cout << A * B + C;
```

Linguaggio Assembly:

```
mov eax,A
mul B
add eax,C
call WriteInt
```

Linguaggio macchina:

```
A1     00000000
F7 25 00000004
03 05 00000008
E8 00500000
```

Esempio: funzione per il calcolo della media in C ed in linguaggio assembly

Linguaggio C (alto livello)

```
double mean (double* x, unsigned n)
{
  double m = 0;
  int i;
  for (i=0; i<n; i++)
     m += x[i];
  m /= n;
  return m;
}
```

Linguaggio Assembly

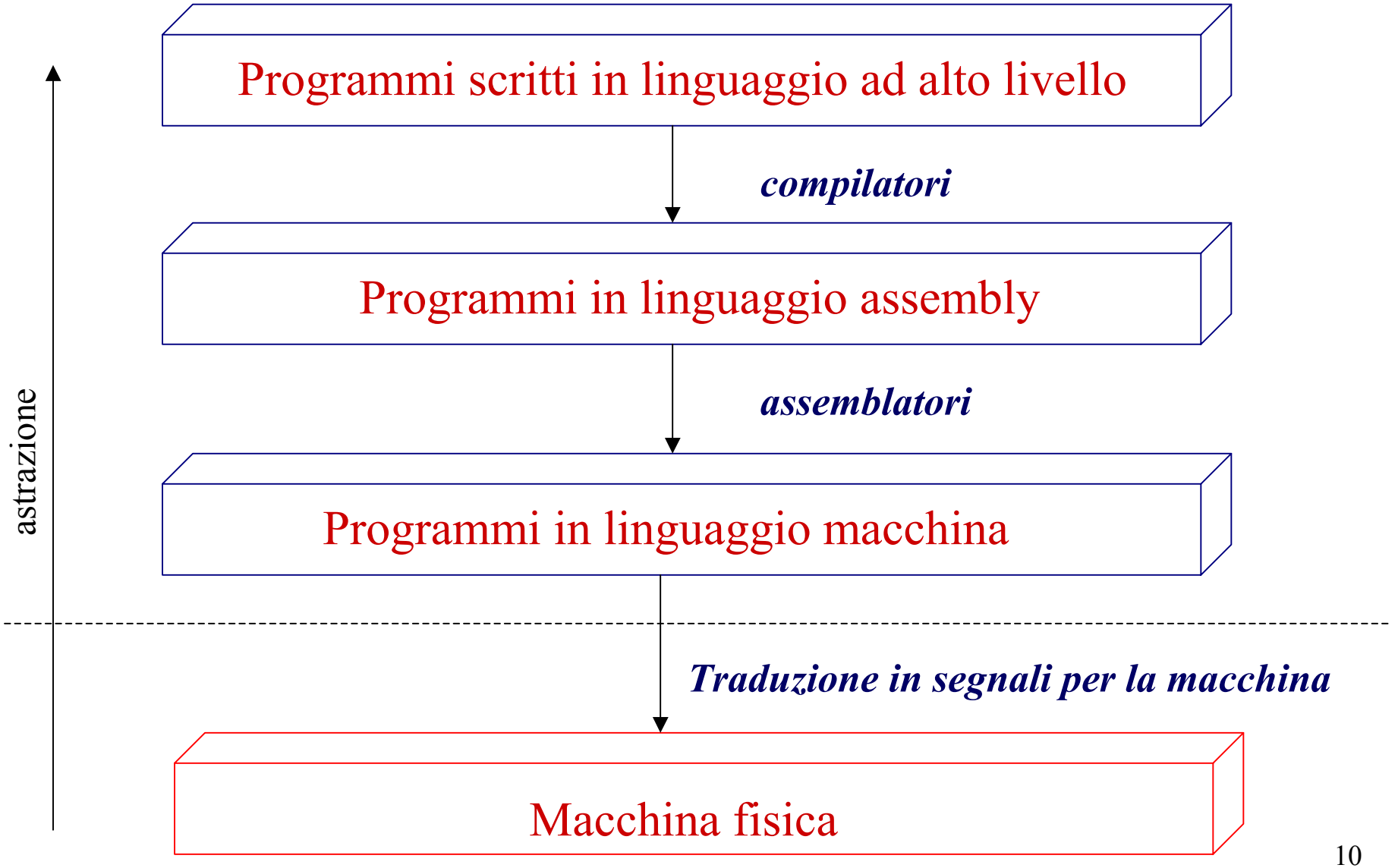
```
            .file       "qq.c"
            .text
.globl mean
            .type       mean,@function
mean:
            pushl       %ebp
            movl        %esp, %ebp
            subl        $24, %esp
            movl        $0, -8(%ebp)
            movl        $0, -4(%ebp)
            movl        $0, -12(%ebp)
.L2:
            movl        -12(%ebp), %eax
            cmpl        12(%ebp), %eax
            jb          .L5
            jmp         .L3
.L5:
            movl        -12(%ebp), %eax
            leal        0(,%eax,8), %edx
            movl        8(%ebp), %eax
            fldl        -8(%ebp)
            faddl       (%eax,%edx)
            fstpl       -8(%ebp)
            leal        -12(%ebp), %eax
            incl        (%eax)
            jmp         .L2
.L3:
            movl        12(%ebp), %eax
            movl        $0, %edx
            pushl       %edx
            pushl       %eax
            fildll      (%esp)
            leal        8(%esp), %esp
            fldl        -8(%ebp)
            fdivp       %st, %st(1)
            fstpl       -8(%ebp)
            movl        -8(%ebp), %eax
            movl        -4(%ebp), %edx
            movl        %eax, -24(%ebp)
            movl        %edx, -20(%ebp)
            fldl        -24(%ebp)
            leave
            ret
.Lfe1:
            .size       mean,.Lfe1-mean
            .ident      "GCC: (GNU) 3.2.2
```

Linguaggi ad alto livello

- Sono definiti *astraendo* rispetto alla macchina fisica
- Realizzano una *macchina virtuale* soprastante alla macchina fisica e visibile al programmatore
- Richiedono di essere implementati su un particolare sistema di calcolo tramite strumenti opportuni (*compilatori* o *interpreti*)

Esempi: *fortran, C, lisp, java, R*

Livelli di rappresentazione e macchine astratte



Linguaggi compilati e linguaggi interpretati

- I *programmi compilati* vengono tradotti completamente dalla prima all' ultima istruzione nel linguaggio macchina del sistema sottostante (netta distinzione fra compile-time e run-time).
  Es: programmi in *C*, *fortran*, *C++*
- I *programmi interpretati* vengono tradotti ed eseguiti immediatamente riga per riga (l' interprete simula una macchina astratta, no distinzione netta fra compile-time e run-time).
  Es: programmi in R
- Esistono casi "intermedi": es: *java*.
- I *compilatori* e gli *interpreti* sono i programmi che effettuano la traduzione dal linguaggio ad alto livello al linguaggio macchina

Paradigmi di programmazione

Paradigma	**Modello computaz.**	**In cosa consiste un programma**	**Esempi**
Imperativo	Stato=astrazione della memoria	Eseguire comandi	Pascal, C
Funzionale	Funzioni (definizione ed applicazione)	Valutare espressioni	Lisp
Ad oggetti	Universo di oggetti	Mandare messaggi ad oggetti	Java, C++
Logico	Predicati e deduzione logica	Soddisfare un goal	Prolog

Linguaggi di programmazione e produzione del software

Modello tradizionale “a cascata” per la produzione del sw:

1. Analisi e specificazione dei requisiti
2. Progetto (design) del sistema
3. Implementazione: Produzione del codice nel linguaggio prescelto
4. Verifica e validazione
5. Manutenzione

In realtà il processo di produzione è ciclico.

Linguaggi di programmazione e ambienti di sviluppo

Ogni fase dello sviluppo del sw può essere supportato da *ambienti di sviluppo*.

Ambienti di sviluppo per l'implementazione del sw:

- Text editor
- Compilatori
- Linker
- Librerie
- Debugger
- ...

Linguaggi di programmazione per la bioinformatica

- In linea di principio qualsiasi linguaggio ad alto livello può essere utilizzato.
- Esistono comunque *linguaggi con librerie e package specifici* specializzati per la bioinformatica:

Progetti Open Source

  - *Perl* e *BioPerl*: http://bioperl.org
  - *Python* e *Biopython*: http://biopython.org
  - *Java* e *BioJava*: http://biojava.org
  - *R* e *Bioconductor*: http://www.bioconductor.org

  - Matlab e toolbox per la bioinformatica

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Introduzione a R

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

R come linguaggio per la bioinformatica

- Linguaggio ad alto livello orientato alla analisi dei dati
- Permette di strutturare dati complessi ed eterogenei
- Dispone di un ambiente di lavoro e di sviluppo per lavorare interattivamente con i dati
- Dispone di package (librerie) specifiche per la bioinformatica
- R è il linguaggio utilizzato dal progetto internazionale open source Bioconductor per la gestione ed elaborazione di dati genomici e proteomici
- E' uno dei linguaggi maggiormente utilizzati dalla comunità internazionale dei bioinformatici

Caratteristiche di R

- Linguaggio ad alto livello *interpretato*
- Dotato di insiemi di operatori ad alto livello per *calcoli su array e matrici*
- Supporta paradigmi di programmazione *imperativa*, *object-oriented* e *funzionale*.
- Fornisce un ambiente per la *elaborazione interattiva* dei dati
- *Ambiente integrato di risorse software* per la gestione ed elaborazione di dati e la visualizzazione di grafici
- Dispone di *interfacce* verso programmi e moduli sw scritti con altri linguaggi
- Ambiente di sviluppo e package open source disponibili liberamente in internet.

Breve storia di R

- Deriva da *S*, un linguaggio ed un sistema sviluppati da *John Chambers* e collaboratori negli anni ‘80 presso i Laboratori Bell.
- S è valso l’ *ACM Software Systems Award* al suo principale progettista J. Chamber nel 1999.
- *R è un progetto Open Source* conforme per la maggior parte ad S:
  - Sviluppato inizialmente da *Ross Ihaka and Robert Gentleman* all’ Università di Auckland (Nuova Zelanda)
  - Attualmente sviluppato da una comunità internazionale di ricercatori e sviluppatori in ambito sia accademico sia industriale
  - Opera attraverso il web: www.r-project.org
  - Archivi software e documentazione: cran.r-project.org/

Dove reperire R

CRAN - the Comprehensive R Archive Network: http://cran r-project.org/ (ci sono anche mirror locali)

Sono disponibili distribuzioni binarie per :

- *Windows* 95, 98, NT e 2000
- *Macintosh* (System 8.6 - 9.1, MacOS X)
- *Linux*

L' installazione domestica sul proprio PC è semplice.

Bibliografia per R

Libri e manuali introduttivi disponibili on-line:

- W. Venables and D.M. Smith, *An Introduction to R*: http://cran.r-project.org/doc/manuals/R-intro.pdf, 2004
- J. Maindonald, *Using R for Data Analysis and Graphics*: http://wwwmaths.anu.edu.au/~johnm, 2000
- A. Mineo, *Una guida all' utilizzo dell' ambiente Statistico R*:

Libri e manuali specifici sulla definizione del linguaggio e per lo sviluppo di Package R

- R Development Core Team, *R Language Definition*: http://cran.r-project.org/doc/manuals/R-def.pdf, 2004
- R Development Core Team, *Writing R extensions*: http://cran.r-project.org/doc/manuals/R-exts.pdf, 2004

Testo di riferimento per il linguaggio S

- R. Becker, J. Chambers and A. Wilks, *The new S language*. Chapman & Hall, New York, 1988

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

L' ambiente R per Windows

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

R e sistemi “a finestre”

- E’ possibile lavorare in R con sistemi Unix, Linux, Macintosh e Windows.
- In ogni caso il medesimo codice R può girare su qualsiasi piattaforma
- In particolare si possono utilizzare sistemi a finestre per poter meglio gestire l’ ambiente R
- Tale modalità è in particolare l’ unica disponibile per sistemi Windows.

R GUI per Windows

- La finestra principale "*RGui*" rappresenta l' interfaccia grafica utente per Windows.
- Il *menu* presenta diverse voci utili per la manipolazione dei file (File), per l' editing dei dati (Edit), per caricare ed installare moduli sw (Packages), per gestire le finestre (Windows), per ottenere accedere alla documentazione e ad "aiuti" in linea (Help), per listare o rimuovere gli "oggetti" dell' ambiente corrente (Misc)
- Racchiude al suo interno un' altra finestra "*Console*" con il *prompt* per i comandi.

Utilizzare R interattivamente

I comandi per R sono inseriti interattivamente dalla console, che visualizza il prompt dei comandi:

>

Ad esempio, per chiudere R il comando è:

> q()

Alternativamente si può anche scegliere l' opzione *Exit* dal menu *File*.

Working directory: Il path per i file utilizzati durante la sessione di R fanno riferimento alla working directory.

Setting della working directory:

- Menu File/Change dir ...
- Dall' icona di R del desktop. Click con il pulsante destro del mouse: selezionare Proprietà/da

Utilizzo dell' help

R dispone di un sistema di help interattivo versatile ed articolato:

- Dal menu Help:
  - Manuals (manuali in pdf)
  - HTML help (manuali in HTML, documentazione dei package, motori di ricerca per parole chiave, link a risorse sul web)
  - ...
- Direttamente dal prompt:
  > help(xxx): apre una finestra di help sulla funzione xxx
  > ?xxx: idem
  > help.start(): apre il browser predefinito per l' help in formato HTML

Esercizi

1. Creare una directory EserciziR. All' interno di essa creare un' altra directory Es1 e settarla come working directory per R.
2. Aprire il documento pdf *Introduction to R* utilizzando il menu *Help* di R.
3. Aprire il medesimo documento in formato HTML.
4. Aprire una finestra di help per la funzione *mean*.

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Vettori ed assegnamenti in R

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Strutture dati in R

Nei linguaggi di programmazione ad alto livello non si ha accesso diretto alla memoria fisica, ma ad una sua astrazione (*struttura dati*).

Le principali strutture dati fornite da R sono:

1. Vettori
2. Array e matrici
3. Fattori
4. Liste
5. Data frame

Vettori

Rappresentano sequenze ordinate di dati omogenei.

Ad es: sequenze ordinate di numeri o di caratteri.

Esempio:

```
> c(1,4,5)  # crea un vettore di
  interi
> c("A","B","C") # crea un vettore
  di caratteri
> c("gatto", "topo", "12") # crea
  un vettore di stringhe
```

La *funzione* c(arg1, arg2, arg3, arg4) combina i suoi argomenti in vettore.

Variabili ed assegnamenti - 1

Un vettore può essere *assegnato* ad una *variabile*.

Esempio 1

```
> X <- c(1,4,5) # il vettore <1 4 5> è assegnato
  alla variabile X
> X
[1] 1 4 5
```

La variabile X rappresenta ora il vettore <1 4 5>: si può pensare come un "contenitore" del vettore stesso

Es. 2

```
> X <- c(4,7) # il vettore <4 7> è assegnato alla
  variabile X
> X
[1] 4 7
```

Un nuovo assegnamento cancella il contenuto precedente

Es.3

```
> Y <- 100  # vettore formato da 1 solo elemento
> Y
[1] 100
```

Variabili ed assegnamenti - 2

Altri modi per rappresentare l' assegnamento:

Es.1

```
> c(1,4,5) -> x
> x = c(1,4,5)
> assign(x, c(1,4,5))
```

I valori di una variabile possono essere assegnati ad altre variabili:

Es.2

```
> y <- 2
> z <- y
> z
[1] 2
```

Non possono essere assegnati valori ad una costante:

```
> 2 <- x
Error in 2 <- x : invalid (do_set) left-
  hand side to assignment
```

Concatenazione di vettori

I vettori possono essere concatenati attraverso l' operatore c di concatenazione:

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> z <- c(x,y)
> z
[1] 1 2 3 4 5 6
> w <- c(z,x,9,y)
> w
 [1] 1 2 3 4 5 6 1 2 3 9 4 5 6
```

Tipi elementari di vettori

I vettori sono sequenze ordinate i cui elementi possono essere di 3 tipi base:

- Numerici: numeri interi o in virgola mobile (floating point)
- Caratteri: singoli caratteri o stringhe (sequenze) di caratteri
- Logici: TRUE o FALSE

Operazioni con vettori aritmetici

Le operazioni usuali dell' aritmetica vengono eseguite sui vettori elemento per elemento:

Addizione e sottrazione

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> z <- x + y
> z
[1] 5 7 9
> d <- y - x
> d
[1] 3 3 3
```

Moltiplicazione e divisione

```
> x <- c(1,2,3)
> y <- c(4,5,6)
> p <- x * y
> p
[1]  4 10 18
> q <- y / x
> q
[1] 4.0 2.5 2.0
```

Funzioni matematiche

Sono disponibili diversi operatori e funzioni matematiche (che operano sempre elemento per elemento). Ad esempio:

```
> x <- c(1,2,3)
> x^3
[1]  1  8 27
> log(x)
[1] 0.0000000 0.6931472 1.0986123
> exp(x)
[1]  2.718282  7.389056 20.085537
> sin(x)
[1] 0.8414710 0.9092974 0.1411200
> sqrt(x)
[1] 1.000000 1.414214 1.732051
```

Altre funzioni di uso comune

```
> x <- c(1,2,3)
> mean(x)
[1] 2
> var(x)
[1] 1
> max(x)
[1] 3
> min(x)
[1] 1
> range(x)
[1] 1 3
> sum(x)
[1] 6
> prod(x)
[1] 6
```

```
> y <-rnorm(10) # generazione ~N(0,1)
> y
 [1] -1.5171592  0.5538263 -0.9505327
-0.6218845  0.5113505  0.7547935
 [7] -1.5403415  2.3607231  1.3177109
-1.3993465
> sort(y) # ordinamento
 [1] -1.5403415 -1.5171592 -1.3993465
-0.9505327 -0.6218845  0.5113505
 [7]  0.5538263  0.7547935  1.3177109
2.3607231
> order(y)  # indici corrispondenti
            # agli elementi ordinati
 [1]  7  1 10  3  4  5  2  6  9  8
```

La “regola del riciclo” in R (1)

Se si sommano due vettori di diversa lunghezza in R, il vettore più corto viene ripetuto tante volte fino a raggiungere la lunghezza del vettore di maggior lunghezza.

Es:

```
> x <- c(2,4)
> y <- c(3,5,7,9)
> x + y  # il vettore x viene ripetuto due
         # volte (regola del riciclo)
[1]  5  9  9 13
```

La somma precedente equivale cioè a:

```
> xx <- c(x,x)  # duplicazione del vettore x
> xx + y
[1]  5  9  9 13
```

La “regola del riciclo” in R (2)

La regola vale in generale in R, anche per altre strutture dati e per altre operazioni.

Es:

```
> x <- c(2,4)

> xx <- c(x,x)

> y <- c(3,5,7,9)

> xx * y

[1]  6 20 14 36

> x * y # il vettore x viene ripetuto due
        # volte (regola del riciclo)

[1]  6 20 14 36
```

Generazione di sequenze regolari

R dispone di diversi comandi per generare automaticamente sequenze di numeri:

```
> c(1:10)
 [1]  1  2  3  4  5  6  7  8  9 10
> c (5:1)
[1] 5 4 3 2 1
> seq (1,10)
 [1]  1  2  3  4  5  6  7  8  9 10
> seq (from=1, to=4, by=0.5)
[1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

La funzione seq() può avere 5 argomenti (si veda l' help). Un' altra funzione per generare repliche di vettori è rep():

```
> rep( c(1,2), times=4)
[1] 1 2 1 2 1 2 1 2
```

Vettori di caratteri

Gli elementi sono caratteri o stringhe di caratteri:

```
> x <- c("A","T","G")
> x
[1] "A" "T" "G"
> y <- c("ATA","TTTG","GCTCG")
> y
[1] "ATA"   "TTTG"  "GCTCG"
```

La funzione paste concatena 1 o piu' argomenti separandoli di default con degli spazi o con i caratteri specificati dall' argomento sep:

```
> paste("A","T","G")
[1] "A T G"
> paste("A","T","G", sep="")
[1] "ATG"
> paste("A","T","G", sep="C")
[1] "ACTCG"
> paste (x,y,sep="--")
[1] "A--ATA"   "T--TTTG"  "G--GCTCG"
```

Vettori logici

Sono vettori i cui elementi possono assumere valore TRUE o FALSE.

Es:

```
> x <- c(TRUE, FALSE, TRUE, FALSE)
> x
[1]  TRUE FALSE  TRUE FALSE
```

I vettori logici possono essere generati da condizioni e operazioni logiche.

Es:

```
> x <- 1:5
> x
[1] 1 2 3 4 5
> l <- x > 3  # la condizione logica è valutata
              # elemento per elemento
> l
[1] FALSE FALSE FALSE  TRUE  TRUE
```

Operatori logici

Operatori logici:

`<`, `<=`, `>`, `>=`, `==` (uguaglianza), `!=` (disuguaglianza)

Es:

```
> x<- 3:8
> x >5
[1] FALSE FALSE FALSE  TRUE  TRUE  TRUE
> x <= 5
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE
> x == 5
[1] FALSE FALSE  TRUE FALSE FALSE FALSE
> x != 5
[1]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE
> x != c(5,6) # vale la "regola del riciclo"!
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE
```

Connettivi logici

Se c1 e c2 sono espressioni logiche, allora possono essere connesse tramite:

- & (AND)
- | (OR)
- ! (NOT)

Es 1:

```
> c1 <- 5>3
> c1
[1] TRUE
> c2 <- "gatto" == "topo"
> c2
[1] FALSE
```

```
> c3 <- c1 & c2
> c3
[1] FALSE
> c3 <- c1 | c2
> c3
[1] TRUE
> c3 <- !c1
> c3
[1] FALSE
```

Es. 2: I connettivi logici operano sui vettori elemento per elemento :

```
> c1 <- c(3,4) > c(  2,6)
> c2 <- c(1,2) < c(2,8)
```

```
> c1 & c2
[1 ]  TRUE FALSE
> c1 | c2
[1] TRUE TRUE
```

Dati mancanti

- In molte situazioni reali i componenti di un vettore possono essere "non noti" o comunque non disponibili.
- In questi casi R riserva il valore speciale NA ("Not Available").
- In generale qualunque operazione che coinvolga valori NA ha come risultato NA.

Es:

```
> x <- c(1:4,NA)
> x + 2
[1]  3  4  5  6 NA
```

Si noti che NA non è un valore ma un "marcatore" di una quantità non disponibile:

```
> x == NA
[1] NA NA NA NA NA
```

Per individuare quali elementi siano effettivamente NA in un vettore si deve usare la funzione is.na():

```
> is.na(x)
[1] FALSE FALSE FALSE FALSE  TRUE
```

Vettori: selezione e accesso a sottoinsiemi di elementi

Esistono diverse modalità di accesso a singoli elementi o a sottoinsiemi di elementi di un vettore. In generale la selezione e l' accesso avviene tramite l' operatore [] (parentesi quadre) : sottoinsiemi di elementi di un vettore sono selezionati collegando al nome del vettore un vettore di indici in parentesi quadre. Esistono *4 diverse modalità di selezione/accesso*:

- Vettori di indici interi positivi
- Vettore di indici interi negativi
- Vettore di indici logici
- Vettori di indici a caratteri

Selezione ed accesso tramite vettori di indici interi positivi (1)

Gli elementi di un vettore *x* sono selezionati tramite un vettore *y* di indici positivi racchiuso fra parentesi quadre :

x[y]

i corrispondenti elementi sono selezionati e concatenati.

Es:

```
> x <- 5:10
> x[1] # selezione di un singolo elemento
[1] 5
> x[5]
[1] 9
> length(x) # lunghezza del vettore
[1] 6
> x[7] # accesso ad un elemento "fuori range"
[1] NA
```

Selezione ed accesso tramite vettori di indici interi positivi (2)

Si possono selezionare più elementi utilizzando vettori di indici positivi. Ad es:

```
> x <- 5:10
> x[2:4]
[1] 6 7 8
> x[c(1,3,5)]
[1] 5 7 9
> x[1:8]  # il vettore contiene solo 6 elementi!
[1]  5  6  7  8  9 10 NA NA
```

Un esempio un pò più complicato:

```
> s <- c("A","T")[rep(c(1,2,2,1), times=3)]
> s
 [1] "A" "T" "T" "A" "A" "T" "T" "A" "A" "T" "T" "A"
```

Si possono anche assegnare sottoinsiemi di elementi ad un vettore:

```
> y <- c("G","C","G","C")
> s[1:3] <- y[2:4]
> s
 [1] "C" "G" "C" "A" "A" "T" "T" "A" "A" "T" "T" "A"
```

Selezione ed accesso tramite vettori di indici interi negativi

Sono selezionati gli elementi di un vettore *x* che devono essere esclusi tramite un vettore *y* di indici negativi racchiuso fra parentesi quadre : x[y]
Es:

```
> y <- rep(c("G","A","T","T"), times=3)
> y
 [1] "G" "A" "T" "T" "G" "A" "T" "T" "G" "A" "T" "T"
> z <- y[-(1:5)] # selezionati tutti gli elementi di y
                 # eccetto primi 5
> z
[1] "A" "T" "T" "G" "A" "T" "T"
> z <- z[-length(z)]  # cancellazione dell' ultimo
                      # elemento di z
> z
[1] "A" "T" "T" "G" "A" "T"
```

Selezione ed accesso tramite vettori indice logici

Il vettore indice deve essere della stessa lunghezza del vettore i cui elementi devono essere selezionati. Sono selezionati gli elementi corrispondenti a TRUE nel vettore degli indici ed omessi quelli corrispondenti a FALSE.

Es:

```
> x <- c(1:5,NA,NA)
> x
[1]  1  2  3  4  5 NA NA
> i <- c(rep(TRUE,times=3),rep(FALSE,times=4))
> i # i è il vettore indice logico
[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
> x[i]  # selez.  elementi tramite vett. indice logico
[1] 1 2 3
> x[!is.na(x)] # selezione elementi che non sono NA
[1] 1 2 3 4 5
> x[!is.na(x) & x > 2]
[1] 3 4 5
```

Selezione ed accesso tramite vettori di indici a caratteri

E' applicabile quando un vettore possiede un attributo[1] names per identificare le sue componenti. In questo caso un sottovettore del vettore names può essere utilizzato per selezionare le componenti

Es:

```
> campione <- c(45,210,5.12,73.22,0.82)
> names(campione) <- c("Eta", "Conc.ciclosporina",
"Dose.giornaliera", "Conc.urea", "Conc.bilirubina")
> campione
             Eta Conc.ciclosporina  Dose.giornaliera
           45.00             210.00              5.12
Conc.urea   Conc.bilirubina
    73.22              0.82
> Eta.Dose <- campione[c("Eta","Dose.giornaliera")]
> Eta.Dose
             Eta Dose.giornaliera
           45.00              5.12
```

[1] Gli attributi dei vettori e degli oggetti in R verranno trattati nella prossima lezione

Esercizi

1. Generare un vettore contenente i primi 100 interi positivi: Calcolare la media, la varianza e la deviazione standard dei suoi elementi.
2. Ripetere il precedente esercizio con un vettore di 100 numeri random estratti da una distibuzione normale standard (si veda la funzione *rnorm*)
3. (a) Costruire una sequenza *s* costituita da 3 ripetizioni in sequenza della stringa "CGCT".
   (b) Estrarre dalla sequenza ottenuta una sottosequenza *sub* in cui compaiano tutti gli elementi di *s* eccetto la lettera C.
   (c) Aggiungere in coda alla sequenza ottenuta 3 valori NA.
   (d) Riottenere la sequenza *sub* in (b) tramite la funzione *is.na*()
4. Cosa accade se si prova a costruire un vettore eterogeneo di numeri e caratteri ? E di numeri e di valori logici ?
5. Generare un vettore *vet* di 100 elementi casuali estratti secondo la distribuzione uniforme in [0,1] (vedi *runif*). Estarre da *vett* un vettore *subvet* i cui elementi abbiano valore v>0.2 e valore v<0.6. Estrarre da *subvet* gli elementi di indice pari ed assegnarli al vettore *w*. Trasformare *w* in modo che i suoi elementi siano normalizzati rispetto al loro valore medio ed alla deviazione standard.

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Oggetti ed attributi in R

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Oggetti e attributi

- Le entità su cui R opera sono informalmente definite come oggetti.
- La nozione oggetto ha un significato tecnico preciso nei linguaggi *object-based* ed *object-oriented*. In queste slide, seguendo l' approccio informale dei corsi e manuali introduttivi su R, il termine "oggetto" si riferisce invece a qualsiasi entità definita in R, caratterizzata da proprietà definite tramite attributi.
- Da questo punto di vista sono oggetti sia i vettori (a valori numerici, a caratteri o logici), sia le liste, gli array ed anche le funzioni.
- Gli attributi degli oggetti possono essere modificati ed esistono funzioni (*attributes(object)* e *attr(object,names)*) che permettono di accedervi.
- In questa sezione vedremo solo qualche esempio di attributi utilizzabili per i vettori. Altre tipologie di attributi verano introdotte nelle prossime lezioni

Attributi

Ogni oggetto è caratterizzato da un insieme di attributi.

Es: i vettori sono costituiti da elementi base di un medesimo modo:

```
> x <- c(1,2,3,4)
> mode(x)
[1] "numeric"
> y <- x>3
> y
[1] FALSE FALSE FALSE  TRUE
> mode(y)
[1] "logical"
> x <- c("ciao", "topo")
> mode(x)
[1] "character"
```

I vettori sono caratterizzati da una determinata lunghezza:

```
> x <- c(1,2,3,4)
> length(x)
[1] 4
```

Cambio di modo

In R è spesso possibile è spesso possibile forzare *un cambio di modo degli oggetti*.

Ad es: con i vettori è possibile forzare un vettore di modo numerico ad un modo carattere:

```
> x <- 0:9
> x
 [1] 0 1 2 3 4 5 6 7 8 9
> car <- as.character(x)
> car
 [1] "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
```

E viceversa:

```
> as.integer(car)
 [1] 0 1 2 3 4 5 6 7 8 9
```

In R esistono diverse funzioni del tipo "*as.something*()" per forzare la conversione di modo o per permettere ad un oggetto di acquisire nuovi attributi

Cambiare la lunghezza di un oggetto

In R si possono costruire oggetti di lunghezza nulla:

```
> vuoto <- numeric()
> length(vuoto)
[1] 0
> mode(vuoto)
[1] "numeric"
```

E si può variare la lunghezza di un oggetto:

```
> v <- numeric()
> v[4] <- 1 # costruzione implicita di un vettore di
            # lunghezza 4
> v
[1] NA NA NA  1
> v <- v[2*1:2] # “accorciamento” implicito tramite
                #  assegnamento
> v
[1] NA  1
```

L'attributo names

L' attributo names, quando presente, etichetta gli elementi di un vettore o di una lista con una stringa di caratteri.

Es:

```
> x <- numeric(5)
> names(x) <- c("eta","altezza","press.","glicemia","temp.")
> x
     eta  altezza   press. glicemia    temp.
       0        0        0        0        0
> x[1] <- 45
> x[2] <- 175
```

L' attributo names può anche essere usato come indice:

```
> x["altezza"]
altezza
    175
> x["altezza"] <- 182
> x[2]
altezza
    182
```

Visualizzare e settare gli attributi

La funzione attributes(*oggetto*) dà una lista di tutti gli attributi non intrinseci (attributi diversi da *mode* e *length*) attualmente definiti per un determinato oggetto:

```
> x <- 1:5
> attributes(x)
NULL
> names(x) <- c(paste("c",1:5,sep=""))
> attributes(x)
$names
[1] "c1" "c2" "c3" "c4" "c5“
```

La funzione attr(*oggetto*,*nome*), può essere utilizzata per selezionare o modificare uno specifico attributo:

```
> attr(x,"names")
[1] "c1" "c2" "c3" "c4" "c5"
> attr(x,"names")[3]
[1] "c3"
> attr(x,"names")[3] <- "componente3“ # modifica
                                     # di un attributo
> attr(x,"names")
[1] "c1"   "c2" "componente3" "c4" "c5"
```

Esercizi

1. Costruire un vettore numerico *x* di 4 elementi etichettati con p,s,t,q. Utilizzare la funzione *names* per visualizzare le etichette. Assegnare alla seconda componente il valore 5 utilizzando l' attributo names come indice.
   Cambiare le etichette di *x* in primo,secondo,terzo,quarto.
2. Generare un vettore casuale *z* di 10 elementi. Allungare il vettore in modo che abbia 20 elementi. Tramite un assegnamento ridurre *z* a lunghezza 7. Verificare tramite la funzione *length* che la lunghezza sia effettivamente 7.
3. Costruire un vettore numerico i cui elementi siano i numeri da 5 a 9. Forzarne la conversione a vettore di caratteri. E' possibile forzare la conversione a vettore logico? Cosa si ottiene in tal caso? Se si forza la conversione di un vettore di stringhe a vettore di interi cosa si ottiene?

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Fattori

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Fattori

- Strutture dati per rappresentare valori che possono assumenre solo valori discreti
- In R i diversi valori che possono assumere i dati discreti sono definiti come *livelli*
- Es: dati qualitativi, o dati ordinali.
- Possono essere utili anche per “etichettare” dati memorizzati in altre strutture dati

Costruzione di fattori

I fattori sono costruiti tramite la funzione factor:

```
> trt <- factor( rep( c(" Control", "Treated"), c( 3, 4)))
> trt
[1]  Control  Control  Control Treated  Treated  Treated  Treated
Levels:  Control Treated
> levels(trt) # visualizza i livelli del fattore
[1] " Control" "Treated"
> str( trt)  # visualizza in modo succinto la struttura di
        # un oggetto R
Factor w/ 2 levels "Control"," Treated": 1 1 1 2 2 2 2
> summary( trt) # la funzione summary fornisce una tabella
          # della frequenze dei due livelli Control e Treated
Control Treated
3       4
```

I fattori si possono costruire con factor utilizzando un vettore esistente:

```
> s<-c(rep("A",3),rep("T",3),rep("G",3),rep("C",3))
> fs <- factor(s)
> fs
 [1] A A A T T T G G G C C C
Levels: A C G T
> summary(fs)/length(fs)
   A    C    G    T
0.25 0.25 0.25 0.25
```

La funzione tapply

La funzione tapply consente di applicare una funzione *f* a dati etichettati da un fattore per ognuno del livelli specificati nel fattore stesso:

```
> trt <- factor( rep( c(" Control", "Treated"), c( 3, 4)))
trt
[1]  Control  Control  Control Treated  Treated  Treated  Treated
Levels:  Control Treated
> blood.press<-c(190,180,207,178,156,205,151) # vettore dati
                                              # "etichettato" con trt
> tapply(blood.press,trt,mean) # la funzion mean è applicata ai
                       # dati blood.press per ogni livello di trt
 Control  Treated
192.3333 172.5000
```

Fattori ordinati

Un fattore ordinato è un tipo speciale di fattore in cui i livelli sono ordinati:

```
> pain <- ordered( c(" Moderate", "None", "Severe", "Severe",
   "None"),
+ levels = c(" None", "Moderate", "Severe"))
> str( pain)
Factor w/ 3 levels "None"," Moderate",..: 2 1 3 3 1
> pain
[1] Moderate None Severe Severe None
Levels: None < Moderate < Severe
> summary( pain)
None Moderate Severe
2 1 2
```

L' ordine di default è lessicografico (alfabetico):

```
> pain <- ordered( c(" Moderate", "None", "Severe", "Severe",
   "None"))
> pain
[1] Moderate None Severe Severe None
Levels: Moderate < None < Severe
```

Esercizi

1. Costruisci un fattore di 25 elementi (stringhe), caratterizzato da 3 diversi livelli
2. Costruisci un fattore ordinato di 15 elementi di livelli basso < medio < alto.
   Genera un corrispondente vettore di misurazioni di lunghezza 15 e calcola il valor medio e la deviazione standard (funzione sd) di ogni livello tramite la funzione tapply.
3. Quali strutture dati si potrebbero scegliere per modellare un data set caratterizzato da un insieme di pazienti sani e malati sottoposti a 5 diversi tipi di analisi cliniche?

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Array e matrici

Giorgio Valentini

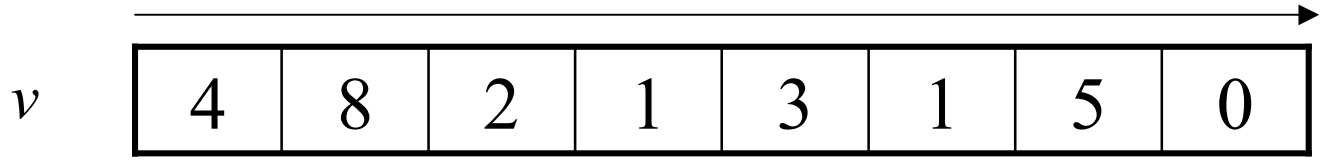
e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Array e matrici come generalizzazioni multidimensionali di vettori (1)

- I vettori sono sequenze ordinate di elementi omogenei. Un vettore *v* è rappresentabile tramite una *struttura unidimensionale*:

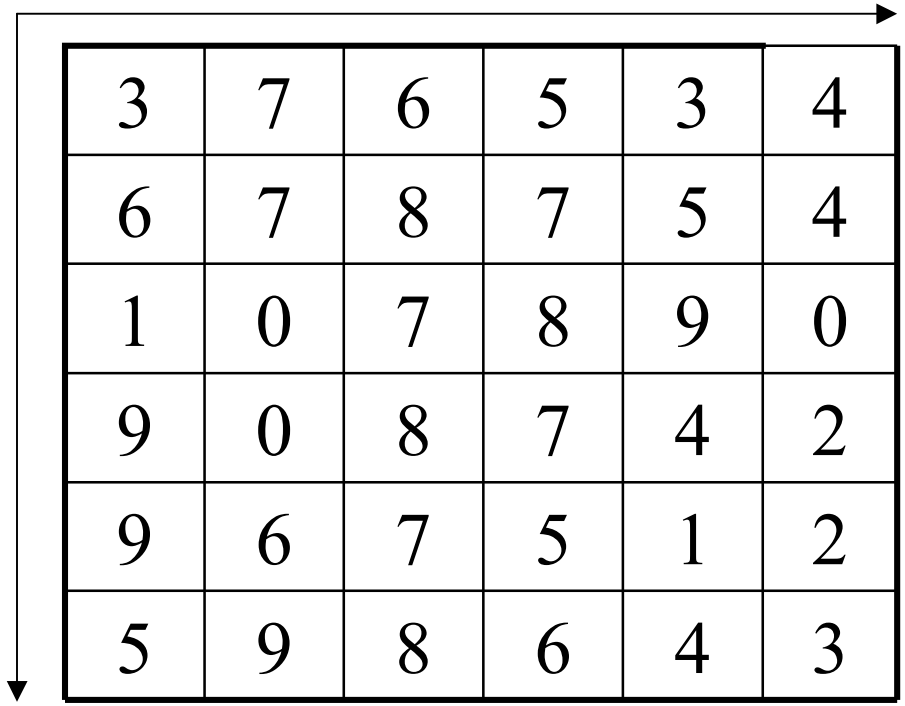


- Essendo strutture unidimensionali è possibile accedere o modificare un elemento di un vettore utilizzando un *unico indice*:

```
> v[2]
8
> v[1] <- 7
> v[1]
7
```

Array e matrici come generalizzazioni multidimensionali di vettori (2)

- In R è possibile rappresentare *estensioni bidimensionali* di vettori (matrici)



3	7	6	5	3	4
6	7	8	7	5	4
1	0	7	8	9	0
9	0	8	7	4	2
9	6	7	5	1	2
5	9	8	6	4	3

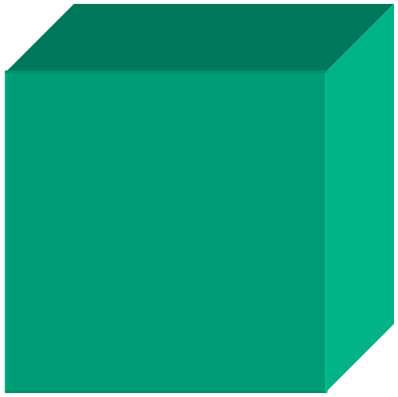
- Essendo strutture bidimensionali, è necessaria una *coppia di indici* per accedere o modificare un elemento di una matrice :

```
> m[1,3] # seleziona el. I
         # riga e III colonna

> 6

> m[2,4] <- 0
```

Array e matrici come generalizzazioni multidimensionali di vettori (3)

- In generale in R è possibile rappresentare *estensioni multidimensionali* di vettori (array):

Es: array tridimensionale



Per accedere ad un elemento sono necessari 3 indici.

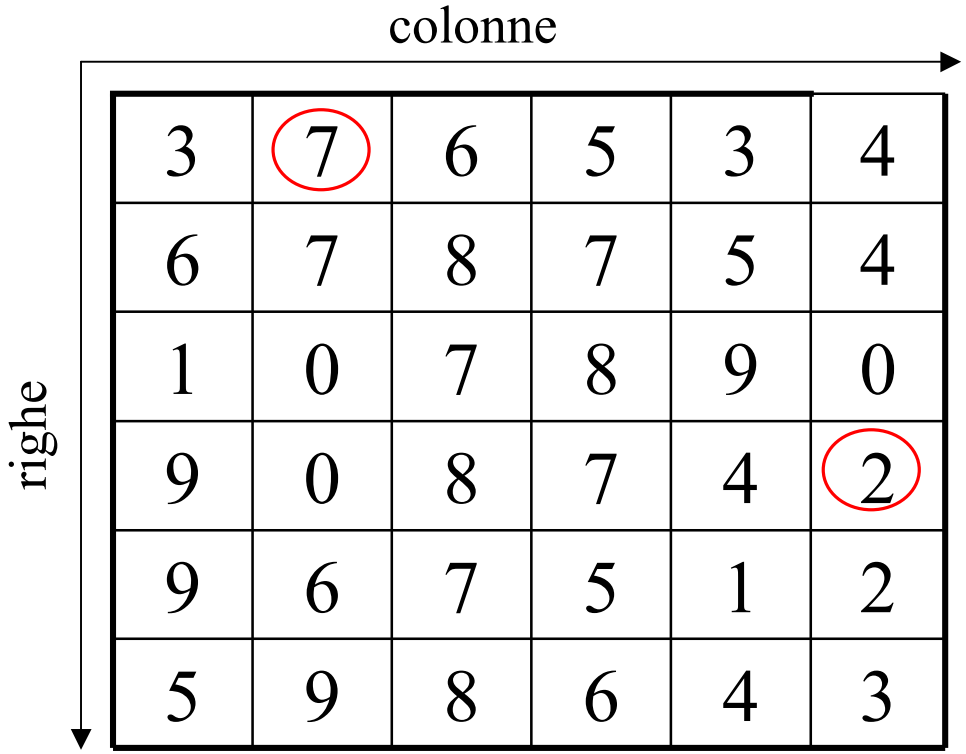
Es:

```
> a[1,3,4]
```

- In R si possono costruire array di dimensione arbitraria (limitatamente alle disponibilità di memoria)
- Le matrici sono a tutti gli effetti array bidimensionali
- Sugli array sono applicabili le medesime operazioni di accesso e modifica (estese a più dimensioni) utilizzabili per i vettori

Matrici

- In R le matrici sono array bidimensionali:



3	7	6	5	3	4
6	7	8	7	5	4
1	0	7	8	9	0
9	0	8	7	4	2
9	6	7	5	1	2
5	9	8	6	4	3

- Gli elementi di una matrice sono selezionati tramite una coppia di indici racchiusi fra parentesi quadre:

- `m[1,2]` (`m` variabile cui è associata la matrice) identifica l' elemento della riga 1 e colonna 2

- `m[4,6]` identifica l' elemento della riga 4 e colonna 6

Costruzione di matrici (1)

Le matrici possono essere costruite tramite la funzione matrix a partire da un vettore esistente

```
> x <- 1:24
> m <- matrix (x, nrow=4) # genera una matrice con 4
# righe utilizzando gli elementi del vettore x
> m  # si noti l' assegnamento dei valori "per colonne"
     [,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
> length(m)
[1] 24
> mode(m)
[1] "numeric"
> dim(m)
[1] 4 6
```

L' attributo dim mostra che la matrice *m* è un array bidimensionale 4 X 6

Costruzione di matrici (2)

La funzione matrix può avere anche altri argomenti:

```
> x <- 1:12
> m <- matrix (x, ncol=4) # si può specificare il n. delle colonne
> m
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> m <- matrix (x, ncol=4, byrow=TRUE) # inserimento el. per righe
> m
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
> m <- matrix (x, ncol=5)  # anche per le matrici si applica
                           # la "regola del riciclo"
Warning message:
data length [12] is not a sub-multiple or multiple of the number of
columns [5] in matrix
> m
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    4    7   10    1
[2,]    2    5    8   11    2
[3,]    3    6    9   12    3
```

Le funzioni cbind e rbind

Le matrici possono essere costruite anche tramite le funzioni cbind ed rbind.

cbind forma matrici legando insieme vettori o matrici "orizzontalmente":

```
> x <- 1:3
> y <- 4:6
> m <- cbind(x,y)
> m
     x y
[1,] 1 4
[2,] 2 5
[3,] 3 6
```

rbind forma matrici legando insieme vettori o matrici "verticalmente":

```
> x <- 1:3
> y <- 4:6
> m <- rbind(x,y)
> m
  [,1] [,2] [,3]
x    1    2    3
y    4    5    6
```

Se i vettori costituenti non sono della stessa lunghezza si applica la regola del riciclo

Costruzione di matrici con cbind per "giustapposizione" di matrici

```
> x<-1:12
> y<-13:24
> m1<-matrix(x,nrow=3)
> m2<-matrix(y,nrow=3)
> m <- cbind(m1,m2)
> m
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    4    7   10   13   16   19   22
[2,]    2    5    8   11   14   17   20   23
[3,]    3    6    9   12   15   18   21   24
> m2<-matrix(y,nrow=2)
> m <- cbind(m1,m2)
Error in cbind(...) : number of rows of matrices
  must match (see arg 2)
```

Operazioni con le matrici (1)

Somma e prodotto con costanti

```
> x <- 1:12
> A <- matrix(x,nrow=3)
> A + 2
     [,1] [,2] [,3] [,4]
[1,]    3    6    9   12
[2,]    4    7   10   13
[3,]    5    8   11   14
> B <- A * 2
> B
     [,1] [,2] [,3] [,4]
[1,]    2    8   14   20
[2,]    4   10   16   22
[3,]    6   12   18   24
```

Somma e prodotto "elemento per elemento"

```
> A+B
     [,1] [,2] [,3] [,4]
[1,]    3   12   21   30
[2,]    6   15   24   33
[3,]    9   18   27   36
> A*B
     [,1] [,2] [,3] [,4]
[1,]    2   32   98  200
[2,]    8   50  128  242
[3,]   18   72  162  288
```

Operazioni con le matrici (2)

Prodotto di matrici "righe X colonne": Operatore `%*%`

```
Si ricordi che A e B sono
matrici 3X4:

> A%*%B # il numero delle
# colonne di A deve essere
# uguale al numero di righe di
# B!

Error in A %*% B : non-
conformable arguments

> A%*%t(B) # t(B) indica la
           # trasposta di B

     [,1] [,2] [,3]

[1,]  332  376  420

[2,]  376  428  480

[3,]  420  480  540
```

Prodotto di matrici per vettori:

```
> z <- 1:4

> A%*%z

     [,1]

[1,]   70

[2,]   80

[3,]   90

> z%*%A

Error in z %*% A : non-
conformable arguments

> z%*%t(A)

     [,1] [,2] [,3]

[1,]   70   80   90
```

Altri operatori e funzioni per le matrici

In R esiste un repertorio amplissimo di operatori e funzioni specifiche per le matrici (si veda la documentazione):

- Ad es: la funzione *diag*, che ha un significato diverso a seconda che il suo argomento sia un intero, un vettore ed una matrice (si provi).
- La funzione *solve* consente di invertire una matrice e può essere usata per risolvere sistemi lineari. Ad es, se `A` è una matrice quadrata non singolare e `b` un vettore compatibile con `A`, allora `x<-solve(A,b)` risolve il sistema lineare `Ax=b`. (si provi ...)
- Esistono inoltre funzioni per il calcolo degli autovalori ed autovettori di matrici simmetriche, per il calcolo della svd di una matrice e molte altre.

Array

- Gli array sono generalizzazioni multidimensionali di vettori
- Consentono di costruire strutture complesse in più dimensioni
- Informalmente si possono pensare come sequenze ordinate di sottostrutture di dimensione inferiore. Ad es: una matrice può essere vista come una sequenza di elementi, di cui ognuno è un vettore; un array tridimensionale come una sequenza di matrici; un array 4-D come una sequenza di arrray 3-D.
- Si noti comunque che non esiste una gerarchia fra le dimensioni: si può accedere ad una sottostruttura scegliendo come “principale” una qualsiasi delle dimensioni.

Costruzione di array

La costruzione di array avviene tramite la funzione array:

Es. 1 (array bidimensionale):

```
> z <- array(1:6, c(2,3))
> z
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

L' array bidimensionale `z` è una matrice che si sarebbe potuto costruire equivalentemente nel modo seguente:

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

Es. 2 (array tridimensionale)

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

Sintassi della funzione array

array (*vettore di dati, vettore dimensioni, vettore del nome delle dimensioni*)

Vettore dati : un qualsiasi vettore di dati che viene utilizzato per "riempire" l' array

Vettore dimensioni: attributo *dim* dell' array, cioè un vettore di lunghezza pari al numero delle dimensioni dell' array che fornisce l' indice massimo per ogni dimensione

Vettore del nome delle dimensioni: attributo *dimnames* dell' array, cioè lista che fornisce un nome (stringa di caratteri) alle diverse dimensioni.

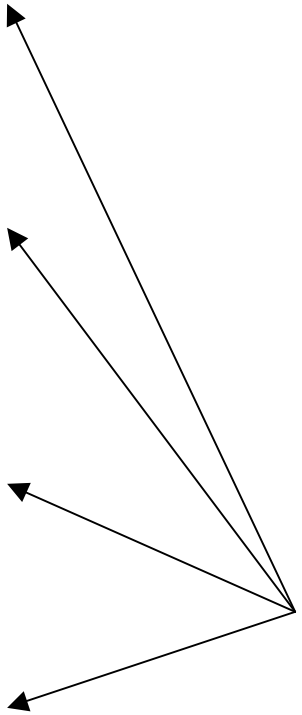
Esempi di array (1)

```
> x<-1:12  # vettore dei dati per l' array
> d<-c(2,6) # vettore delle dimensioni dell' array (attr.dim)
> z<-array(x,d) # costruzione dell' array: il III argomento
                #  dimnames è assente e per default è NULL
> z   # array bidimensionale 2X6
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
> x # x è un vettore
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> dim(x)  # come vettore non possiede l' attributo dim
[1] NULL
> dim(x)<-d  # si può assegnare una dimensione ad un vettore ...
> x  # trasformandolo in un array bidimensionale
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
> is.array(x) # la funzione is.array conferma che ora x è un array
[1] TRUE
> dim(x)<-NULL # cancellazione dell' attributo dim
> x  # x torna ad essere un vettore
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> is.array(x)
[1] FALSE
> is.vector(x)
[1] TRUE
```

Esempi di array (2)

Si noti come vengano inseriti gli elementi nell' array utilizzando l' indice più "a sinistra":

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



Essendo un array tridimensionale sono necessari 3 indici per accedere agli elementi: z[x,y,w]

Il primo elemento viene inserito in posizione z[1,1,1]; il secondo in z[2,1,1] (si muove per primo l' indice più a sinistra). Quindi si passa al II indice: z[1,2,1] e z[2,2,1].

Riempita la I matrice 2X3 con i primi 6 elementi,si muove finalmente il III indice, accedendo al I elemento della II matrice 2X6: z[1,1,2]

Quindi l' indice più a sinistra è quello che si "muove più velocemente", mentre quello più a destra corrisponde a quello più lento.

Un array tridimensionale 2X3X4 si può quindi pensare come l' "impilamento" di 4 matrici 2X3.

Accesso agli elementi di array e matrici

Le regole di accesso per array e matrici seguono quelle già viste per i vettori, considerando però l' esistenza di più indici e quindi la possibilità di utilizzare un vettore per ogni dimensione:

- Vettori di indici interi positivi
- Vettore di indici interi negativi
- Vettore di indici logici
- Vettori di indici a caratteri

Si utilizza quindi un vettore di indici *per ogni dimensione* dell' array

Esempi di accesso agli elementi di una matrice

```
> m <- matrix (1:12,nrow=2)
> m
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
> m[1,2]
[1] 3
> m [1,3:4] # el. I riga,col. 3 e 4
[1] 5 7
> m [1:2,4:6] # I e II riga, col.
              # dalla IV alla VI
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
> m[1,] #tutti gli el. della I riga
[1]  1  3  5  7  9 11
> m[,3] #tutti gli el. III col.
[1] 5 6
> m[,c(1,2,6)]
     [,1] [,2] [,3]
[1,]    1    3   11
[2,]    2    4   12
```

```
> m[,-4] # esclusione el. IV
         # colonna
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    9   11
[2,]    2    4    6   10   12
> m[m>4] # el.>4 (si ottiene un
         # vettore)
[1]  5  6  7  8  9 10 11 12

> dimnames(m)<list(c("r1","r2"),
paste("c",1:6,sep="")) # viene
# dato un nome alle componenti
> m
   c1 c2 c3 c4 c5 c6
r1  1  3  5  7  9 11
r2  2  4  6  8 10 12
> m["r1","c2"] # vettori indice
               # a caratteri
[1] 3
> m["r1",]
c1 c2 c3 c4 c5 c6
 1  3  5  7  9 11
```

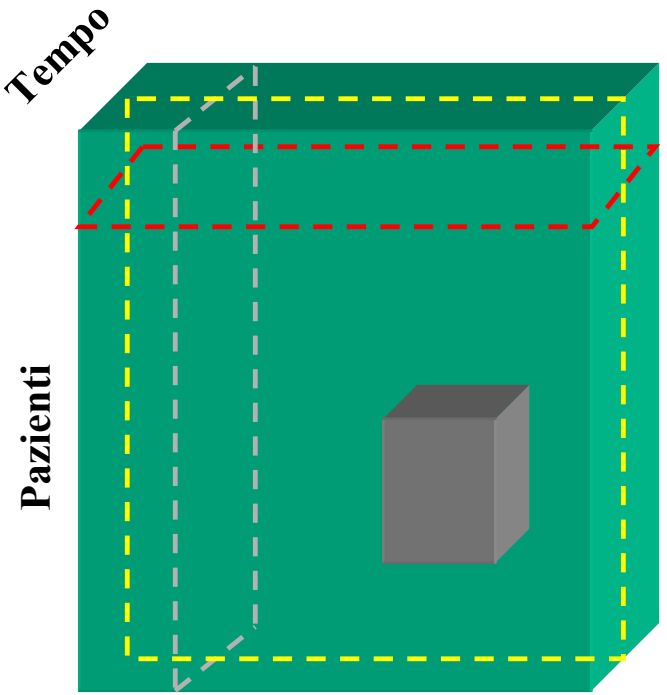
Esempi di accesso agli elementi di un array

Si consideri l' array z della slide 17:

```
> z <- array(1:24, c(2,3,4))
```

Si può accedere a "fettine" della struttura tridimensionale da ognuna delle 3 dimensioni, ottenendo 3 diversi array bidimensionali (matrici):

```
> z[1,,]
     [,1] [,2] [,3] [,4]
[1,]    1    7   13   19
[2,]    3    9   15   21
[3,]    5   11   17   23
> z[,1,]
     [,1] [,2] [,3] [,4]
[1,]    1    7   13   19
[2,]    2    8   14   20
> z[,,1]
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```



E' possibile accedere ad altre sottoparti della struttura 3D, analogamente a quanto visto nel caso bidimensionale (matrici)

Operazioni con gli array

Le operazioni sono effettuate elemento per elemento ed il risultato è un array il cui attributo *dim* è lo stesso degli operandi.

Es.:

```
> x<-1:12
> y<-x*2
> z1<-array(x,c(2,3,2))
> z2<-array(y,c(2,3,2))
> z3 <- z1*z2 # molt. el. per el.
> z1+z2 # somma el. per el.
, , 1
     [,1] [,2] [,3]
[1,]    3    9   15
[2,]    6   12   18
, , 2
     [,1] [,2] [,3]
[1,]   21   27   33
[2,]   24   30   36
```

```
> z1/z2 # div. el. per el.
, , 1
     [,1] [,2] [,3]
[1,]  0.5  0.5  0.5
[2,]  0.5  0.5  0.5
, , 2
     [,1] [,2] [,3]
[1,]  0.5  0.5  0.5
[2,]  0.5  0.5  0.5
```

Gli operandi devono avere il medesimo attributo dim:

```
> z3<-array(y,c(2,3,4))
> z1-z3
Error in z1 - z3 : non-conformable
arrays
```

Esercizi

1. Costruire una matrice 10X10 composta da numeri casuali in almeno 2 modi diversi utilizzando le funzioni matrix ed array.
2. Costruire 2 matrici di caratteri a piacere *x* e *y*, la prima di dimensione 3X4, la seconda di dimensione 5X3. Modificare con un unico assegnamento la matrice *x* in modo da sostituire le sue 2 prime colonne con le ultime 2 righe di *y*.
3. Costuire un array *a* 3X4X5 costituito dai primi 60 numeri naturali positivi. Moltiplicarlo per un array della medesima dimensione ma composto da numeri casuali. E' possibile moltiplicare l' array *a* per un vettore numerico di lunghezza 60? Se si, cosa si ottiene? Modificare infine l' array a in modo da ottenere un array 4-D 3X5X2X2. Verificare infine gli attributi di a utilizzando funzioni opportune.
4. Costruire due matrici *M* ed *N* che abbiano entrambe 5 colonne. Costruire, se possibile, tramite *rbind* una matrice di 5 colonne che abbia come righe le righe di entrambe le matrici. Utilizzando *M* ed *N*, è possibile costruire una matrice tramite *cbind*?
5. Date du matrici quadrate *A* e *B* di dimensione 3X3 costituite da numeri casuali, calcolarne il prodotto elemento per elemento e "righe per colonne". Scelto un vettore *b* di 3 elementi, risolvere il sistema lineare $Ax=b$ che ne deriva.
6. Quale struttura dati si potrebbe utilizzare per modellare un insieme di dati numerici relativi a diverse tipologie di analisi per un insieme di diversi pazienti? Se tali dati dovessero essere rilevati ad intervalli di tempo differenti, quali altre struttura dati si potrebbero utilizzare?

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Liste

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

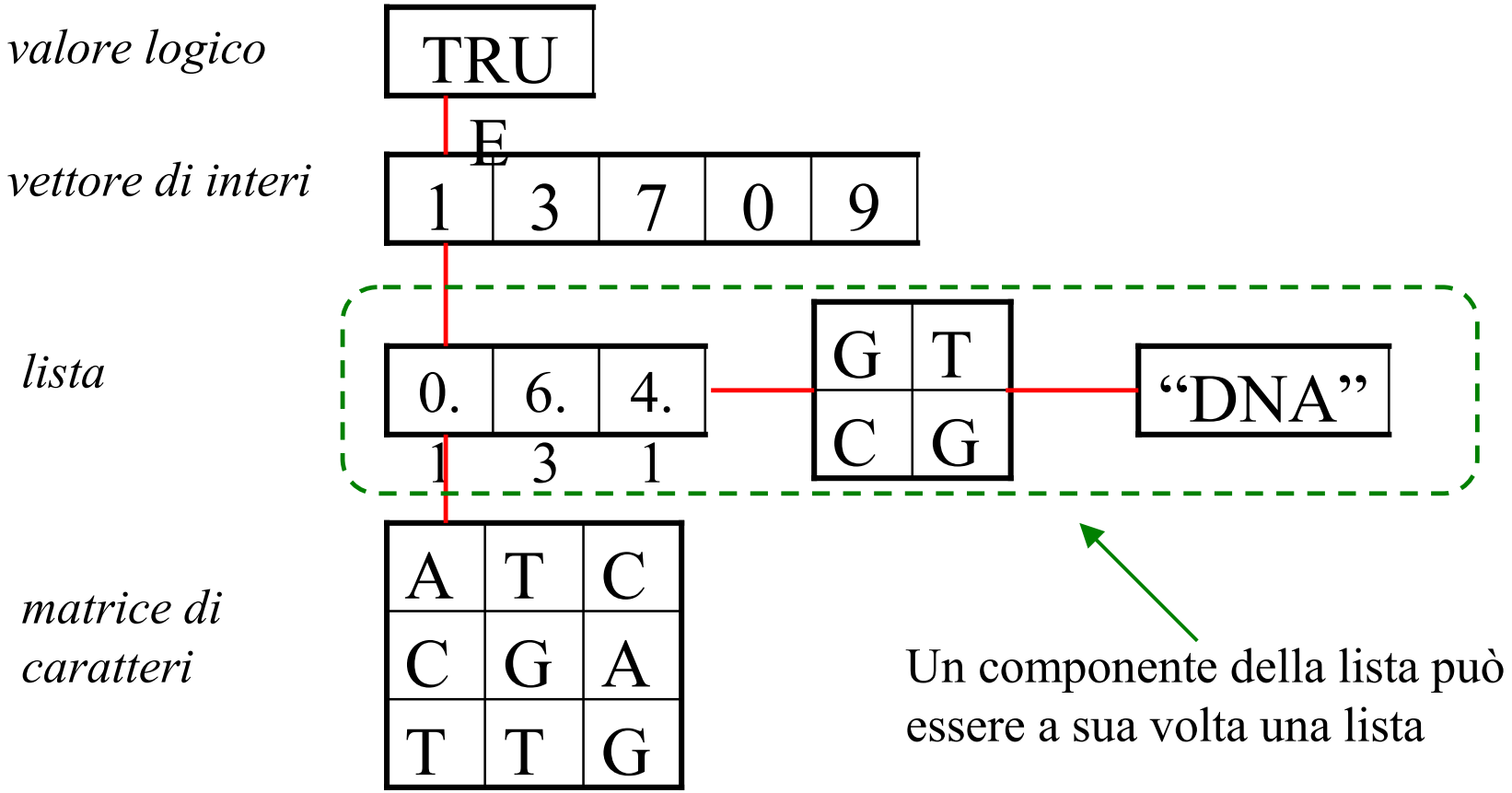
Università degli Studi di Milano

Lista come insieme ordinato di oggetti eterogenei (1)

- Le liste rappresentano un *insieme ordinato di oggetti* (componenti)
- Le componenti possono non essere dello stesso tipo o modo. Quindi le liste rappresentano *insiemi di oggetti eterogenei.*
- I componenti possono essere ad es: un vettore numerico, un valore logico, una matrice, un array di caratteri, una funzione o anche un' altra lista.
- La lista è quindi una *struttura dati ricorsiva,* poichè una sua componente può essere a sua volta una lista (e la lista componente può avere come componente un' altra lista).

Lista come insieme ordinato di oggetti eterogenei (2)

Una lista composta da oggetti eterogenei: un valore logico, un vettore di interi, un' altra lista ed una matrice di caratteri



Costruzione di una lista

Per costruire le liste si usa la funzione list:

I componenti delle liste sono sempre numerati:

```
> li <- list(TRUE, c(1,3,7,0,9))
> li
[[1]]
[1] TRUE
[[2]]
[1] 1 3 7 0 9
```

E' però possibile assegnare alle componenti un nome:

```
> li <- list(val=TRUE,
vector=c(1,3,7,0,9))
> li
$val
[1] TRUE
$vector
[1] 1 3 7 0 9
```

Accesso alle componenti di una lista

Esistono 3 modalità di accesso alle componenti di una lista:

1. Accesso tramite indice numerico
2. Accesso tramite il nome delle componenti
3. Accesso tramite indice “a caratteri”

Accesso tramite indice numerico (1)

I componenti delle liste sono numerati ed è possibile accedere ad essi tramite indice numerico racchiuso fra doppie parentesi quadre.

Es:

```
> li <- list(val=TRUE, vector=c(1,3,7,0,9),
m=matrix(1:12,nrow=2))
> li[[1]]
[1] TRUE
> li[[2]]
[1] 1 3 7 0 9
> li[[3]]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
```

Accesso tramite indice numerico (2)

E' possibile accedere anche ai singoli elementi delle componenti:

Es:

```
> li <- list(val=TRUE, vector=c(1,3,7,0,9),
m=matrix(1:12,nrow=2))
> li[[2]][2] # accesso al II el. della II componente
             # della lista (un vettore)
[1] 3
> li[[3]][1,] # accesso alla I riga della III
              # componente (una matrice)
[1]  1  3  5  7  9 11
```

Accesso tramite il nome delle componenti

Se le componenti hanno un nome è possibile accedere ad esse direttamente tramite il nome stesso

Es:

```
> li <- list(val=TRUE, vector=c(1,3,7,0,9),
m=matrix(1:12,nrow=2))
> li$val
[1] TRUE
> li$vector
[1] 1 3 7 0 9
```

Quindi `li$val` è equivalente a `li[[1]]` e `li$vector` a `li[[2]]`

Tramite la notazione `lista$nome` è possibile accedere anche ai singoli elementi delle componenti:

```
> li$vector[4]
[1] 0
> li$m[1,1]
[1] 1
```

Accesso tramite indice “a caratteri”

Se le componenti hanno un nome è possibile accedere ad esse tramite indice “a caratteri”

Es:

```
> li <- list(val=TRUE, vector=c(1,3,7,0,9),
m=matrix(1:12,nrow=2))
> li[["m"]]
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    3    5    7    9   11
[2,]    2    4    6    8   10   12
```

Questa modalità può essere utile quando il nome della componente è memorizzato in un’ altra variabile:

```
> v<-"vector";  li[[v]]
[1] 1 3 7 0 9
```

Gli operatori “[[]]” e “[]”

- L’ accesso alle componenti tramite gli operatori “[[]]” e “[]” produce risultati sostanzialmente differenti.
- “[[]]” è l’ operatore che seleziona l’ oggetto contenuto nella lista (e l’ eventuale nome associato all’ oggetto non è incluso)
- “[]” è l’ operatore che seleziona una sottolista (si riferisce quindi ad un elemento di modo “lista”, e l’ eventuale nome associato all’ oggetto viene incluso)

Gli operatori “[[]]” e “[]” - esempi

```
> li <- list(n=0.6798, s=c(rep("A",4),rep("T",4)))
> mode(li[1])
[1] "list“
> mode(li[[1]])
[1] "numeric“
> li[1]  # viene stampato come una lista
$n
[1] 0.6798
> li[[1]]  # viene stampato come un numero
 [1] 0.6798
> sqrt(li[[1]])
[1] 0.8244998
> sqrt(li[1])# non si può fare la radice quadrata di una lista
Error in sqrt(li[1]) : Non-numeric argument to mathematical
function
> names(li[2]) # il nome è associato alla sottolista
[1] "s“
> names(li[[2]])# non è associato alcun nome all’ oggetto
 NULL
```

Allungare e concatenare liste

Come qualsiasi altro oggetto accessibile tramite indici, le liste possono essere estese, aggiungendo specifiche componenti:

```
> li<-list(0.6798,
paste(c(rep("A",4),rep("T",4)),
collapse=""))
> li[3] <- list(TRUE)
> li
[[1]]
[1] 0.6798
[[2]]
[1] "AAAATTTT"
[[3]]
[1] TRUE
```

La concatenazione di liste è possibile tramite la funzione c:

```
> li1 <- list(TRUE,2)
> li2 <- list("pippo")
> li3<-list( c(1,2,3),
   c("T","A","C"))
> li123 <- c(li1,li2,li3)
> li123
[[1]]
[1] TRUE
[[2]]
[1] 2
[[3]]
[1] "pippo"
[[4]]
[1] 1 2 3
[[5]]
[1] "T" "A" "C"
```

Esercizi

1. Costruire una lista *li* composta da una matrice numerica 4X4, da un vettore di caratteri con 32 elementi, dalla stringa “topo”, e da un’ ulteriore lista composta da un vettore di 10 elementi numerici e dal valore logico FALSE.
2. Si estragga dalla lista
   `li<-list(m=matrix(rnorm(64),nrow=8),s=c(rep("T",3),rep("G",5)))`
   (a) la II colonna della matrice (b) le “G” del vettore *s*.
   Si aggiunga quindi alla lista un vettore composto da 10 numeri casuali. E’ possibile traformare la lista ottenuta in un array?
3. E’ possibile costruire un vettore di liste? In caso di risposa affermativa se ne fornisca un esempio.
4. Accedi in 3 modi diversi alla II componente della lista *li* dell’ es.2
5. Spiega la differenza fra le due diverse modalità di accesso al primo elemento della lista *li*, *li[1]* e *li[[1]]*. Tramite la funzione *diag* si estragga la diagonale della matrice memorizzata nella lista *li*.
6. Una serie di campioni è sottoposto ad una serie di analisi, i cui risultati sono espressi in modalità diverse a seconda del tipo stesso delle analisi: come valori interi, valori reali, stringhe di caratteri, un insieme di 3 valori (debole, medio, intenso), come vettore di numeri reali, e come sequenze di lunghezza variabile di numeri reali. Proponi una struttura dati R per modellare i risultati delle analisi relative ai diverse campioni, motivando la scelta effettuata.

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Data frame

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Data frame come struttura per rappresentare insiemi di dati eterogenei (1)

- Un *data frame* può essere considerato come una matrice le cui colonne rappresentano dati eterogenei:

Dati:	val.num	val.car.	val. log.	val.num	val.car.
Dato1	3.45	ATTA	TRUE	0.45	CCAT
Dato2	5.67	TGAT	FALSE	0.91	TATT
Dato3	1.45	TATA	FALSE	3.78	CCCC
Dato4	4.56	TGAG	TRUE	8.03	GAGA
Dato5	8.09	CCCG	TRUE	8.09	AGAG
Dato6	3.11	CATG	TRUE	4.56	ATAT
Dato7	1.40	TGAG	FALSE	1.80	GCTA
Dato8	7.73	GGAC	TRUE	5.90	TGAT

- Formalmente è una lista di *classe* data.frame

Data frame come struttura per rappresentare insiemi di dati eterogenei (2)

- Le colonne del data frame rappresentano *variabili* i cui modi ed attributi possono essere differenti (le matrici e gli array sono invece costituiti da elementi omogenei per modo ed attributo):

Data frame

Dati:	val.num	val.car.	val. log.	val.num	val.car.
Dato1	3.45	ATTA	TRUE	0.45	CCAT
Dato2	5.67	TGAT	FALSE	0.91	TATT
Dato3	1.45	TATA	FALSE	3.78	CCCC
Dato4	4.56	TGAG	TRUE	8.03	GAGA
Dato5	8.09	CCCG	TRUE	8.09	AGAG
Dato6	3.11	CATG	TRUE	4.56	ATAT
Dato7	1.40	TGAG	FALSE	1.80	GCTA
Dato8	7.73	GGAC	TRUE	5.90	TGAT

Matrice (array bidimensionale)

Dati:	val.num	val.num	val.num	val.num	val.num
Dato1	3.45	1.20	2.54	0.45	1.45
Dato2	5.67	2.95	5.44	0.91	0.95
Dato3	1.45	5.21	3.60	3.78	6.78
Dato4	4.56	8.00	2.12	8.03	8.73
Dato5	8.09	1.65	6.00	8.09	6.65
Dato6	3.11	2.58	3.98	4.56	3.56
Dato7	1.40	0.95	2.72	1.80	5.21
Dato8	7.73	8.82	4.43	5.90	3.18

• Un data frame può essere visualizzato come una matrice e si può accedere ai suoi elementi utilizzando indici (come per le matrici ordinarie)

Componenti dei data frame

- Formalmente i data frame sono *liste* di *classe* data.frame
- I *componenti* (colonne) del data frame possono essere costituiti da:
  - Vettori (numerici, a caratteri, logici)
  - Fattori
  - Matrici numeriche
  - Liste
  - Altri data frame

Caratteristiche dei componenti dei data frame

- I vettori numerici, logici ed i fattori sono inclusi direttamente come variabili (colonne) del data frame, mentre i vettori a caratteri sono forzati a fattori.
- Le matrici forniscono tante variabili al data frame quante sono le rispettive colonne
- Le liste forniscono tante variabili quanti sono i suoi componenti
- I data frame quanti sono i componenti

Restrizioni sulle componenti del data frame:

- I vettori componenti devono avere tutti la stessa lunghezza, mentre le matrici devono avere tutte lo stesso numero di righe
- I componenti delle liste incluse nel data frame devono rispettare le restrizioni di cui al punto precedente
- Le componenti del data frame A incluso nel data frame B devono essere conformi alle componenti del data frame B.

Costruzione dei data frame

I data frame sono costuiti tramite la funzione data.frame:

```
> x<-1:4
> y<-5:8
> z<-paste("A",1:4,sep="")
> da.fr<-data.frame(x,y,z)
> da.fr
  x y  z
1 1 5 A1
2 2 6 A2
3 3 7 A3
4 4 8 A4
```

```
 mode(da.fr)
[1] "list"
> attributes(da.fr)
$names
[1] "x" "y" "z"
$row.names
[1] "1" "2" "3" "4"
$class
[1] "data.frame"
```

I data frame possono essere costruiti con matrici

Le matrici componenti il data frame devono avere lo stesso numero di righe:

```
> m1 <-matrix(1:12,nrow=2)
> m2 <-matrix(13:18,nrow=2)
> daf<-data.frame(m1,m2)
> daf
  X1 X2 X3 X4 X5 X6 X1 X2 X3
1  1  3  5  7  9 11 13 15 17
2  2  4  6  8 10 12 14 16 18
> m3 <-matrix(1:12,nrow=4)
> daf2<-data.frame(m1,m3)
Error in data.frame(m1, m3):
arguments imply differing
number of rows: 2, 4
```

Si possono utilizzare insieme matrici e vettori, purchè il numero delle righe delle matrici sia uguale alla lunghezza dei vettori:

```
> m1 <-matrix(1:12,nrow=2)
> v <- c("A","C")
> daf3<-data.frame(m1,v)
> daf3
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
> v1<- c("A","C","G")
> daf4<-data.frame(m1,v1)
Error in data.frame(m1, v1) :
arguments imply differing
number of rows: 2, 3
```

I data frame possono essere costruiti con liste e con altri data frame

I componenti delle liste devono essere "compatibili":

```
> li <-list(a=matrix(1:12,nrow=3),
+ v=c("G","G","C"))
> m <- matrix(13:18,nrow=3)
> daf <- data.frame(li,m)
> daf
  a.1 a.2 a.3 a.4 v X1 X2
1   1   4   7  10 G 13 16
2   2   5   8  11 G 14 17
3   3   6   9  12 C 15 18
> m1 <- matrix(13:18,nrow=2)
> daf <- data.frame(li,m1)
Error in data.frame(li, m1) :
arguments imply differing number
of rows: 3, 2
```

Si possono utilizzare come componenti liste e data frame (ed anche matrici vettori e fattori), purchè compatibili:

```
> li <- list(a=matrix(1:12,nrow=3),
+ v=c("G","G","C"))
>daf.comp<-data.frame(v1=
+ c("A","B","C"),v2=c("D","E","F"))
> daf2<-data.frame(li,daf.comp)
> daf2
  a.1 a.2 a.3 a.4 v v1 v2
1   1   4   7  10 G  A  D
2   2   5   8  11 G  B  E
3   3   6   9  12 C  C  F
```

Accesso alle componenti ed agli elementi dei data frame

Esistono due modalità generali di accesso alle componenti ed agli elementi dei data frame:

1. I data frame sono liste, e quindi è possibile accedere ad essi secondo le *modalità di accesso tipiche delle liste* stesse.
2. Come classe data frame, sono definiti *operatori di accesso tramite vettori di indici*, simili a quelli utlizzati per le matrici e gli array.

Accesso alle componenti dei data frame

Essendo liste, è possibile accedere alle componenti dei data frame secondo le modalità tipiche delle liste:

1. Accesso tramite indice numerico
2. Accesso tramite il nome delle componenti
3. Accesso tramite indice "a caratteri"

Es:

```
> x<-1:4; y<-5:8
> z<-paste("A",1:4,sep="")
> da.fr<-data.frame(x,y,z)
```

1. Accesso tramite indice numerico:

```
> da.fr[[1]]
[1] 1 2 3 4
> da.fr[1]
  x
1 1
2 2
3 3
4 4
```

2. Accesso tramite il nome delle componenti:

```
> da.fr$x
[1] 1 2 3 4
```

3. Accesso tramite indice "a caratteri":

```
> da.fr["x"]
  x
1 1
2 2
3 3
4 4
> da.fr[["x"]]
[1] 1 2 3 4
```

Accesso alle componenti tramite vettori di indici

Sono definiti operatori di accesso specifici per la classe *data.frame*: si tratta di vettori di indici con una semantica simile a quella delle matrici ordinarie:

Es:

```
> x<-1:4; y<-5:8
> z<-paste("A",1:4,sep="")
> da.fr<-data.frame(x,y,z)
> da.fr
  x y  z
1 1 5 A1
2 2 6 A2
3 3 7 A3
4 4 8 A4
```

```
> da.fr[1,2]
[1] 5
> da.fr[2,2:3]
  y  z
2 6 A2
> da.fr[3,]
  x y  z
3 3 7 A3
> da.fr[2:4,1:2]
  x y
2 2 6
3 3 7
4 4 8
```

Esempi di accesso alle componenti di un data frame

```
> mat<-matrix(c(rep("A",3),rep("T",3),rep("G",3),rep("C",3)),nrow=2)
> li <- list(v1=rnorm(2),m=matrix(rnorm(6),nrow=2))
> daf <-data.frame(mat,li) # costruzione data frame
> daf
  X1 X2 X3 X4 X5 X6         v1        m.1       m.2       m.3
1  A  A  T  G  G  C -0.8058378 -0.2722994 0.5641271 2.4615146
2  A  T  T  G  C  C  1.6268044 -0.7586567 0.9504489 0.6681619
```

Esempi di accesso alle componenti:

Accesso " a modo lista"

```
> daf[2]
  X2
1  A
2  T
> daf[[2]]
[1] A T
Levels: A T
> daf["v1"]
          v1
1 -0.8058378
2  1.6268044
> daf$m.1
[1] -0.2722994 -0.7586567
```

Accesso " a modo matrice"

```
> daf[[1,2]]
[1] 1
> daf[,5]
[1] G C
Levels: C G
> daf[,7]
[1] -0.8058378  1.6268044
> daf[,7:length(daf)]
    v1        m.1        m.2       m.3
1  -0.8058378 -0.2722994 0.5641271 2.4615146
2   1.6268044 -0.7586567 0.9504489 0.6681619
> daf[1,5:7]
  X5 X6         v1
1  G  C -0.8058378
```

Estrazione “logica” di osservazioni da data frame

```
> daf[daf$m.3>1,] # estrai da daf solo le osservazioni la cui
                  # variabile m.3 > 1
  X1 X2 X3 X4 X5 X6          v1         m.1       m.2      m.3
1  A  A  T  G  G  C -0.8058378 -0.2722994 0.5641271 2.461515
```

Equivalentemente si può usare la funzione subset:

```
subset(daf,m.3>1)
  X1 X2 X3 X4 X5 X6          v1         m.1       m.2      m.3
1  A  A  T  G  G  C -0.8058378 -0.2722994 0.5641271 2.461515
```

Se si vogliono selezionare elementi da un insieme si può usare l’ operatore %in%:

```
> subset(daf,X2 %in% "A")
  X1 X2 X3 X4 X5 X6          v1         m.1       m.2      m.3
1  A  A  T  G  G  C -0.8058378 -0.2722994 0.5641271 2.461515
> subset(daf,X2 %in% c("A","T"))
  X1 X2 X3 X4 X5 X6          v1         m.1       m.2       m.3
1  A  A  T  G  G  C -0.8058378 -0.2722994 0.5641271 2.4615146
2  A  T  T  G  C  C  1.6268044 -0.7586567 0.9504489 0.6681619
```

La funzione str

La funzione str(*oggetto*) fornisce una serie minima di informazione su *oggetto*.

Es.

```
> data(iris) # caricamento di un data frame da un file
             # contenuto in un package precedentemente caricato
> mode(iris)
[1] "list"
> class(iris)
[1] "data.frame“
> iris
...
> str(iris)
`data.frame':   150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species     : Factor w/ 3 levels "setosa","versic..",..: 1 1 1 1
   1 1 1 1 1 1 ...
```

La funzione summary

La funzione summary(*oggetto*) forinisce una serie di informazioni statistiche su *oggetto*.

Es:

```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
   Species
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
   setosa    :50
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
   versicolor:50
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
   virginica :50
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
```

Una nota sull' accesso alle variabili

Un data frame è una lista di "colonne":

```
> data( Formaldehyde); str( Formaldehyde)
`data.frame':   6 obs. of  2 variables:
 $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde$optden
[1] 0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde[["optden"]]
[1] 0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde[[2]]
[1] 0.086 0.269 0.446 0.538 0.626 0.782
>  Formaldehyde[,2]
[1] 0.086 0.269 0.446 0.538 0.626 0.782
> Formaldehyde[2]
  optden
1  0.086
2  0.269
3  0.446
4  0.538
5  0.626
6  0.782
> mode(Formaldehyde[2])
[1] "list"
```

Accesso all' oggetto (vettore in questo caso) del data frame

Accesso al componente (una lista) del data frame

“Dropping” delle dimensioni

Si è visto che con:

```
>  Formaldehyde[,2]
[1] 0.086 0.269 0.446 0.538 0.626 0.782
```

si accede all’ oggetto del data frame (variabile di 6 osservazioni interpretata come vettore numerico).

Come fare a mantenere la struttura data.frame ?

```
> str(Formaldehyde[,2])  # vettore
 num [1:6] 0.086 0.269 0.446 0.538 0.626 0.782
```

si usa il parametro drop

```
> str(Formaldehyde[,2, drop=FALSE])
`data.frame':   6 obs. of  1 variable:
 $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
> dim(Formaldehyde[,2]) # un vettore non ha attributo dimensioni
NULL
> dim( Formaldehyde[,2,drop=FALSE]) # un data frame sì
[1] 6 1
```

Le funzioni attach e detach (1)

La notazione *oggetto$componente* utilizzata per liste e data frame in alcuni contesti può essere eccessivamente verbosa e poco conveniente.

La funzione attach "rende disponibili" nel cammino di ricerca corrente i nomi delle componenti come se fossero variabili "stand alone":

```
> da.fr<-data.frame(x=1:2,y=3:4,z=paste("C",1:2,sep=""))
> da.fr
  x y  z
1 1 3 C1
2 2 4 C2
> attach(da.fr)
> x # la componente x di da.fr è accessibile direttamente
[1] 1 2
> z # la componente z di da.fr è accessibile direttamente
[1] C1 C2
Levels: C1 C2
```

Assegnamenti o modifiche sulle variabili "estratte" dal data frame con detach non hanno effetto sul data frame stesso. Per modificare le componenti è necessario utilizzare la notazione *oggetto$componente* :

```
> x<-y
> da.fr # da.fr immutato
  x y  z
1 1 3 C1
2 2 4 C2
> da.fr$x<-y
> da.fr # da.fr modificato
  x y  z
1 3 3 C1
2 4 4 C2
```

Le funzioni attach e detach (2)

La funzione detach elimina dal cammino di ricerca le componenti delle liste o data frame precedentemente rese disponibili dalla funzione attach:

```
> da.fr<-data.frame(x=1:2,y=3:4,z=paste("C",1:2,sep=""))
> da.fr
  x y  z
1 1 3 C1
2 2 4 C2
> attach(da.fr)
> x # la componente x di da.fr è accessibile direttamente
[1] 1 2
> z # la componente z di da.fr è accessibile direttamente
[1] C1 C2
Levels: C1 C2
> detach(da.fr)
> x  # la variabile x non è più visibile
Error: Object "x" not found
> y # la variabile y non è più visibile
Error: Object "y" not found
```

Esercizi

1. Costruire un data frame da.fr che abbia come componenti un vettore numerico casuale v di lunghezza 20, una matrice casuale m con 4 colonne ed una lista i cui componenti siano 3 matrici a piacere.
2. Costruire una lista che abbia come componenti 3 vettori a caratteri. Trasformare la lista in un data frame tramite la funzione *as.data.frame*. Quali sono le restrizioni che si devono applicare alle liste perchè siano dei data frame?
3. Si consideri il *data frame daf della slide 12*.
   (a) Estrarre da daf l' ultima colonna
   (b) Estrarre da daf le righe la cui variabile X2 sia uguale ad "A".
   (c) Estrarre da daf un data frame composto solo dalle colonne 4,5,6 e 7.
   (d) Modificare l' ultima colonna di daf in <0,0>
   (e) Aggiungere al data frame una nuova colonna i cui valori rappresentino la somma delle colonne v1, m1, m2 ed m3.
4. Selezionare dal data set *iris* le osservazioni relative alle specie "virginica" con Petal.Length>5.890.
5. Tremite la funzione *summary* ricavare informazioni statistiche di base sulla specie "versicolor" del data set *iris*.
6. Si dispone di un insieme di dati sperimentali (ad es: dati clinici e dati bio-molecolari) da utilizzare a fini diagnostici, relativi ad un insieme di pazienti. Si discuta se ed in quali condizioni i dati siano rappresentabili tramite data frame.

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Leggere e scrivere dati da file

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Lettura e scrittura di dati da file esterni

- I dati utilizzati in bioinformatica sono usualmente di *grandi dimensioni* (ad es: file PDB che memorizzano la struttura tridimensionale delle proteine, file per la memorizzazione di dati di espressione genica, etc)
- Oggetti di grandi dimensioni sono usualmente memorizzati in *file esterni su memoria di massa*
- In R esistono diverse *funzioni di I/O* per la lettura e scrittura di file
- In questa lezione vedremo le più importanti
- Esistono anche funzioni e facility per importare/esportare dati verso altri ambienti/linguaggi di programmazione
- Per maggiori dettagli si consulti il manuale *R Data Import/Export* disponibile on-line ed installato sulle macchine del laboratorio.

Scrittura su file di data frame

La funzione write.table *memorizza un data frame in un file.*

Sintassi: write.table (*x*, file="*data*")

data è il nome del file su cui verrà scritto il data frame *x*.

La funzione write.table possiede molti altri argomenti che permettono di modularne opportunamente la semantica.

```
Esempio:
> m1 <-matrix(1:12,nrow=2); v <- c("A","C")
> daf3<-data.frame(m1,v); daf3
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
> write.table(daf3,file="data.df") # memorizza nel file
# "data.df" il data frame daf3
```

Lettura di data frame da file

La funzione read.table *legge un file memorizzato su disco, inserendo i dati direttamente in un data frame.*

Il file esterno deve essere memorizzato nel modo seguente:

- La prima riga del file deve avere un nome per ciascuna variabile del data frame
- Le righe successive del file memorizzano le osservazioni che saranno memorizzate nel data frame
- Ciascuna di queste righe può avere come primo valore l' etichetta di riga (che sarà memorizzata nel' attributo row.names del data frame)
- Ciascun valore sulla riga è separato da un blank (spazio, tabulazione, etc)
- Possono essere selezionati altri separatori
- Read.table dispone di molti altri parametri che si possono settare per esigenze particolari (vedi help).

Lettura di data frame da file: esempi

Il seguente data frame è memorizzato sul file “data.df”:

```
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
```

La lettura viene effettuta tramite la funzione read.table:

```
daf4<-read.table("data.df")
> daf4
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
```

Il file può naturalmente essere generato da altri programmi (purchè in ASCII), ad es: tramite un qualsiasi text editor, ed essere letto tramite read.table.

Lettura e scrittura di data frame : esempi

Sia read.table, sia write table possono avere altri argomenti opzionali:

```
> m1 <-matrix(1:12,nrow=2); v <- c("A","C")
> daf3<-data.frame(m1,v)
> write.table(daf3,file="data.df",col.names=paste("col",1:7,sep=""))
> read.table("data.df")
     col1 col2 col3 col4 col5 col6 col7
1    1    3    5    7    9   11    A
2    2    4    6    8   10   12    C
>  write.table(daf3,file="data.df",sep = ",") # file memorizzato
# utilizzando la virgola come separatore: controllare con un editor
>  read.table("data.df",sep=",")
  X1 X2 X3 X4 X5 X6 v
1  1  3  5  7  9 11 A
2  2  4  6  8 10 12 C
```

Per una descrizione completa degli argomenti di read.table e write.table vedi l'help in linea.

Funzioni generali per lettura/scrittura di file

- In R sono presenti diverse funzioni generali per lettura e scrittura di file in formato ASCII o binario.
- Ad es: la funzione file può aprire, creare o chiudere file e più in generale *connessioni*: ad es: file in scrittura e/o lettura, connessioni di rete tramite socket o descritte da URL.
- Ci occuperemo brevemente solo dell' insieme di funzioni per la scrittura/lettura di file.

Scrittura di file: esempio

```
>  ff <- file("ex.data", "w")  # apertura di un file
  in scrittura
>      cat("TITLE extra line", "2 3 5 7", "", "11 13
  17", file = ff, sep = "\n") # scrittura d 4 linee
  di testo
>      cat("One more line\n", file = ff)
>      close(ff) # chiude la connessione al file
>      readLines("ex.data") # lettura delle righe dal
  file
[1] "TITLE extra line" "2 3 5 7"           ""
  "11 13 17"          "One more line"
>      unlink("ex.data")  # cancella il file dal
  disco
```

Per scrivere dati su file si può usare anche la funzione write (utilizzata usualmente per scrivere matrici)

Lettura di file: esempio

```
> ff <- file("ex.data", "r") # apertura file in lettura
>  readLines(ff) # lettura di tutto il file
[1] "TITLE extra line" "2 3 5 7"          ""                  "11 13
   17"         "One more line"
> seek(ff,0) # “rewind” del file
[1] 54
>  readLines(ff,n=1) # lettura d una riga alla volta
[1] "TITLE extra line"
>  readLines(ff,n=1)
[1] "2 3 5 7"
>  readLines(ff,n=1)
[1] ""
>  readLines(ff,n=1)
[1] "11 13 17"
>  readLines(ff,n=1)
[1] "One more line"
>  readLines(ff,n=1)  # esaurite le righe del file
character(0)
> close (ff)   # chiusura file
```

La funzione scan

La funzione scan legge un file di input e memorizza i dati in un vettore o una lista.

Esempi:

A. Memorizzazione dati in un vettore

```
> x <- matrix(1:10, nrow=2)
> write (x, "data")
   # scrittura della
   matrice    # su file
> xread <- scan ("data",0)
Read 10 items
> xread
 [1]  1  2  3  4  5  6  7
  8  9 10
```

B. Memorizzazione dati in una lista

Si supponga di avere un file "data" composto dalle seguenti linee:

```
A    0.1  0.2  Q
B    0.5  0.4  M
A    1.1  1.2  Q
Q    0.3  0.9  P
> inp <-
   scan("data",list("",0,0,""))
   # lettura file e memorizzazione
   # in una lista: si noti la
   # lettura "per colonne"
Read 4 records
> inp
[[1]] "A" "B" "A" "Q"
[[2]]  0.1 0.5 1.1 0.3
[[3]]  0.2 0.4 1.2 0.9
[[4]] "Q" "M" "Q" "P"
```

Accesso a data set built-in

- Molti data set sono disponibili con R (data set built-in) ed altri sono contenuti nei package.
- Per listare i data set built-in si utilizza la funzione data().
- Per caricare un data set built-in la sintassi è:
  > data (nome-data-built-in)

Esempio:

```
> data(iris)
> iris
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
.....
```

Caricare dati da package specifici

- Molti data set sono disponibili nei package.
- Per accedere ai data set è necessario fare riferimento al package o precaricare il package stesso

Esempi:

```
> data(Puromycin, package =
"stats")
> Puromycin
   conc rate     state
1  0.02   76   treated
2  0.02   47   treated
3  0.06   97   treated
...
```

```
> library(stats)
> data(Puromycin)
>  Puromycin
   conc rate     state
1  0.02   76   treated
2  0.02   47   treated
3  0.06   97   treated
...
```

Editing dei dati

- E' possibile utilizzare la funzione edit per effettuare cambiamenti "manuali" su matrici e data frame
- E' possibile anche utilizzare la funzione edit per costruire ex novo nuove matrici e data frame
- La funzione edit fornisce un ambiente di editing simile a quelo di un foglio elettronico

Esempi:

```
> edit(iris) # editing di un data frame
# esistente

> new.data.frame <- edit (data.frame())
# creazione di un nuovo data frame
```

Esercizi

1. Costruire un data frame df1 di 5 righe con 6 variabili di cui 4 numeriche e 2 a caratteri. Memorizzare su file il data frame e quindi leggerlo, assegnandolo alla variabile df2.
2. Costruire una matrice numerica utilizzando la funzione *edit*. Scriverla su file tramite la funzione *write*. Ricaricare quindi la matrice in memoria. Si potrebbero utilizzare altre funzioni per memorizzare la matrice?
3. Scrivere su file il data frame df1 dell' es. 1 separando però gli elementi con virgole, ed omettendo il nome delle variabili.
4. Leggere un file di testo a piacere, assegnando ciascuna riga letta ad un elemnto di una lista
5. Leggere un file di testo, assegnando ogni parola ad un elemento di una lista.
6. Carica dal package *Biobase* il data set *aaMap*. A cosa si riferisce? Tramite quale struttura dati è rappresentato?

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Programmi e funzioni in R

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Programmi in R

- Un *programma* in linguaggio R è costituito da una *sequenza* di *espressioni*.
- Ogni espressione viene valutata dall' interprete e se l' espressione è sintatticamente completa viene ritornato un valore
- Il valore ritornato può essere assegnato ad una variabile.

Esempi di espressioni:

```
> 3+2

[1] 5

> x <- 3+2

> 3+2- # istruz. non
# completa

+
```

```
> data.frame(prot=c("P","Q","A"),x=1:3,y=4:6)
  prot x y
1    P 1 4
2    Q 2 5
3    A 3 6
> df <-
data.frame(prot=c("P","Q","A"),x=1:3,y=4:6)
```

Modo di esecuzione dei programmi

- I programmi (sequenze di espressioni) possono essere eseguiti :
  1. *Interattivamente*: ogni istruzione viene eseguita direttamente al prompt dei comandi
  2. *Non interattivamente*: le espressioni sono lette da un file (tramite la funzione *source*) ed eseguite dall' interprete una ad una in sequenza.
- A meno che i programmi non siano brevissimi è opportuno scrivere la sequenza delle espressioni in un file, utilizzando un text editor (ad esempio *EditPad Lite*, scaricabile gratuitamente da: http://www.EditPadLite.com)

Programmi e funzioni

- Abbiamo già visto molti esempi di funzioni disponibili in R
- Le funzioni in R possono anche definite dagli utenti
- I programmi in R sono realizzati tramite funzioni

Funzioni: sintassi

La sintassi per scrivere una funzione è:

```
function (argomenti)
  corpo_della_funzione
```

- `function` è una parola chiave di R
- `Argomenti` è una lista eventualmente vuota di *argomenti formali* separati da virgole: `(arg1, arg2, ..., argN)`
- Un *argomento formale* può essere un simbolo o un'istruzione del tipo '`simbolo=espressione`'
- Il `corpo` può essere qualsiasi espressione valida in R. Spesso è costituito da un gruppo di espressioni racchiuso fra parentesi graffe

Esempi di funzioni (1)

Es.1: Definizione di una funzione di nome `echo`:

```
> echo <- function(x) print(x)
```

Chiamata della funzione `echo`:

```
> echo(6)
[1] 6
```

Es.2 Definizione di una funzione che somma i due suoi argomenti:

```
> sum2 <- function(x,y) x+y
```

argomenti formali

```
> sum2
function(x,y) x+y
```

Chiamata della funzione `sum2` :

argomenti attuali

```
> sum2(1,2)
[1] 3
```

Esempi di funzioni (2)

```
# Funzione per il calcolo della statistica di Golub
# x,y : vettori di cui si vuole calcolare la statistica di golub
# La funzione ritorna il valore della statistica di Golub
golub <- function(x,y)  {
  mx <- mean(x);
  my <- mean(y);
  vx <- sd(x);
  vy <- sd(y);
  g <- (mx-my)/(vx+vy);
  g
}
```

La sequenza di istruzioni del corpo della funzione deve essere racchiusa fra parentesi quadre

Esempi di funzioni (3)

Utilizzo della funzione di Golub:

- La funzione golub è memorizzata nel file “golub.R” (ma potrebbe essere memorizzata in un file con nome diverso)
- Caricamento in memoria della funzione. Due possibilità:

1. `> source("golub")`
2. Dal menu File/Source R code ...

- Chiamata della funzione:

```
> x<-runif(5) # primo argomento della funzione
> x
[1] 0.6826218 0.9587295 0.4718516 0.8284525 0.2080131
> y<-runif(5) # secondo argomento della funzione
> y
[1] 0.6966353 0.0964740 0.4310154 0.1467449 0.2801970
> golub(x,y) # chiamata della funzione
[1] 0.5553528
```

Argomenti delle funzioni

- Argomenti formali e argomenti attuali
- Gli argomenti sono passati per valore
- Modalità di assegnamento degli argomenti:
  - Assegnamento posizionale
  - Assegnamento per nome
- Valori di default per gli argomenti
- L’ argomento ...
- Matching degli argomenti

Argomenti formali e attuali

x e y sono *argomenti formali*:

```
> golub <- function(x,y)  { ... }
```

Tali valori vengono sostituiti dagli *argomenti attuali* quando la funzione è chiamata:

```
> d1 <- runif(5)
> d2 <- runif(5)
```

`d1` e `d2` sono gli argomenti attuali che sostituiscono i formali e vengono effettivamente utilizzati all' interno della funzione:

```
> golub(d1,d2)
[1] 0.2218095
> d3 <- 1:5
> golub(d1,d3)
[1] -1.325527
```

Gli argomenti sono passati per valore

Le modifiche agli argomenti effettuate nel corpo delle funzioni non hanno effetto all' esterno delle funzioni stesse:

```
> fun1 <- function(x)  x <- x*2
> y<-4
> fun1(y)
> y
[1] 4
```

In altre parole i valori degli argomenti attuali sono modificabili all' interno della funzione stessa, ma non hanno alcun effetto sulla variabile dell' ambiente chiamante.

Nell' esempio precedente la copia di `x` locale alla funzione viene modificata, ma non viene modificato il valore della variabile `y` passata come argomento attuale alla funzione `fun1`

Modalità di assegnamento degli argomenti: assegnamento posizionale

Tramite questa modalità gli argomenti sono assegnati in base alla loro posizione nella lista degli argomenti:

```
> fun1 <- function (x, y, z, w) {}
> fun1(1,2,3,4)
```

L' argomento attuale 1 viene assegnato a *x*, 2 a *y*, 3 a *z* e 4 a *w*.

Altro esempio:

```
>  sub <- function (x, y) {x-y}
> sub(3,2) # x<-3 e y<-2
[1] 1
> sub(2,3) # x<-2 e y<-3
[1] -1
```

Modalità di assegnamento degli argomenti: assegnamento per nome

Tramite questa modalità gli argomenti sono assegnati in base alla loro nome nella lista degli argomenti:

```
> fun1 <- function (x, y, z, w) {}
> fun1(x=1, y=2, z=3, w=4)
```

L' argomento attuale 1 viene assegnato a *x*, 2 a *y*, 3 a *z* e 4 a *w*.

Quando gli argomenti sono assegnati per nome non è necessario rispettare l' ordine degli argomenti:

```
fun1(y=2, w=4, z=3, x=1) ≡ fun1(x=1, y=2, z=3, w=4)
```

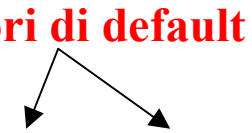
Ad esempio:

```
>  sub <- function (x, y) {x-y}
> sub(x=3,y=2) # x<-3 e y<-2
[1] 1
> sub(y=2,x=3) # x<-3 e y<-2
[1] 1
```

Valori di default per gli argomenti

E' possibile stabilire valori predefiniti per tutti o per parte degli argomenti: tali valori vengono assunti dalle variabili a meno che non vengano esplicitamente modificati nella chiamata della funzione.

Esempio:




```
> fun4 <- function (x, y, z=2, w=1) {x+y+z+w}
> fun4(1,2) # x<-1, y<-2, z<-2, w<-1
[1] 6
> fun4(1,2,5) # x<-1, y<-2, z<-5, w<-1
[1] 9
> fun4(1) # y non ha valore di default !
Error in fun4(1) : Argument "y" is missing, with
  no default
```

L’ argomento ...

L’ argomento speciale ‘...’ rappresenta un numero arbitrario di argomenti: si può quindi usare per funzioni con un numero arbitrario di argomenti.

Esempio:

```
> fun2 <- function (x, ...)  {x + c(...)}
> fun2(1,2)
[1] 3
> fun2(1,4,5)
[1] 5 6
```

Si usa spesso per passare un numero variabile di argomenti ad altre funzioni chiamate nel corpo della funzione stessa.

Matching degli argomenti

Si consideri una semplice funzione che stampa i suoi 4 argomenti:

```
print4 <- function (x=4, y=3, z=2, w=1) {
  cat("x =",x,"\t");
  cat("y =",y,"\t");
  cat("z =",z,"\t");
  cat("w =",w,"\n");
}
```

Gli argomenti attuali possono essere assegnati in modi diversi:

```
> print4(5,6)
x = 5   y = 6   z = 2   w = 1
> print4(y=6,5)
x = 5   y = 6   z = 2   w = 1
> print4(y=6,x=5,w=9)
x = 5   y = 6   z = 2   w = 9
> print4()
x = 4   y = 3   z = 2   w = 1
```

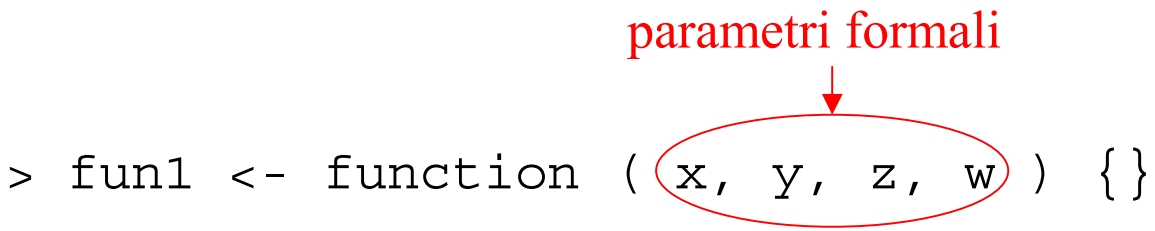
```
> print4(z=0)
x = 4   y = 3   z = 0   w = 1
> print4(1,2,3,4)
x = 1   y = 2   z = 3   w = 4
> print4(1,w=2,3,x=0)
x = 0   y = 1   z = 3   w = 2
> print4(1,2,3,4)
x = 1   y = 2   z = 3   w = 4
```

Scope

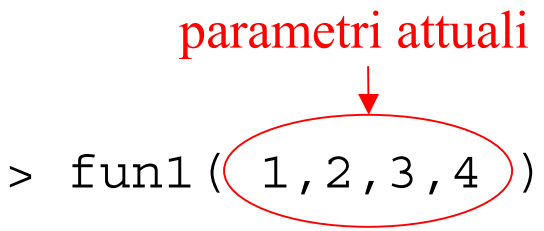
- Le regole di scope sono l' insieme delle regole mediante cui un valore viene associato ad un simbolo.
- Dal punto di vista delle regole di scope i simboli presenti nel corpo di una funzione possono essere suddivisi in 3 classi:
  1. Parametri formali
  2. Variabili locali
  3. Variabili libere

Parametri formali

Sono i parametri della lista degli argomenti presenti nella definizione della funzione:



I loro valori sono determinati dal processo di *binding* degli argomenti attuali della funzioni ai rispettivi parametri formali:

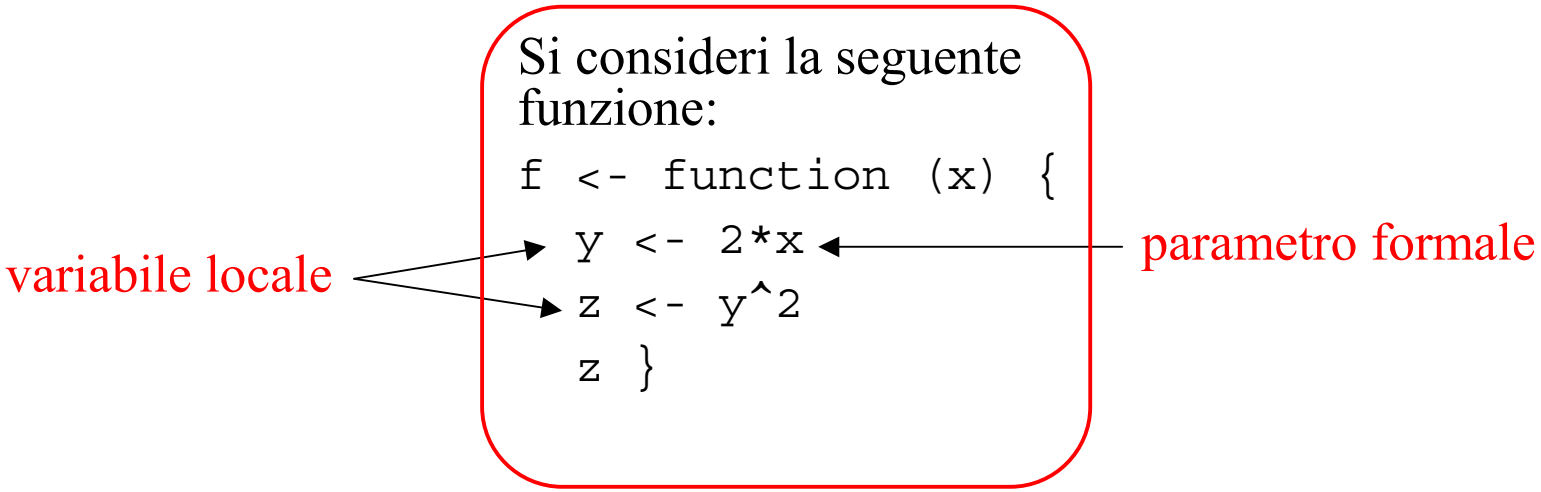


I valori dei parametri attuali sono legati ai rispettivi parametri formali:

```
x<-1, y<-2, z<-3, w<-4
```

Variabili locali

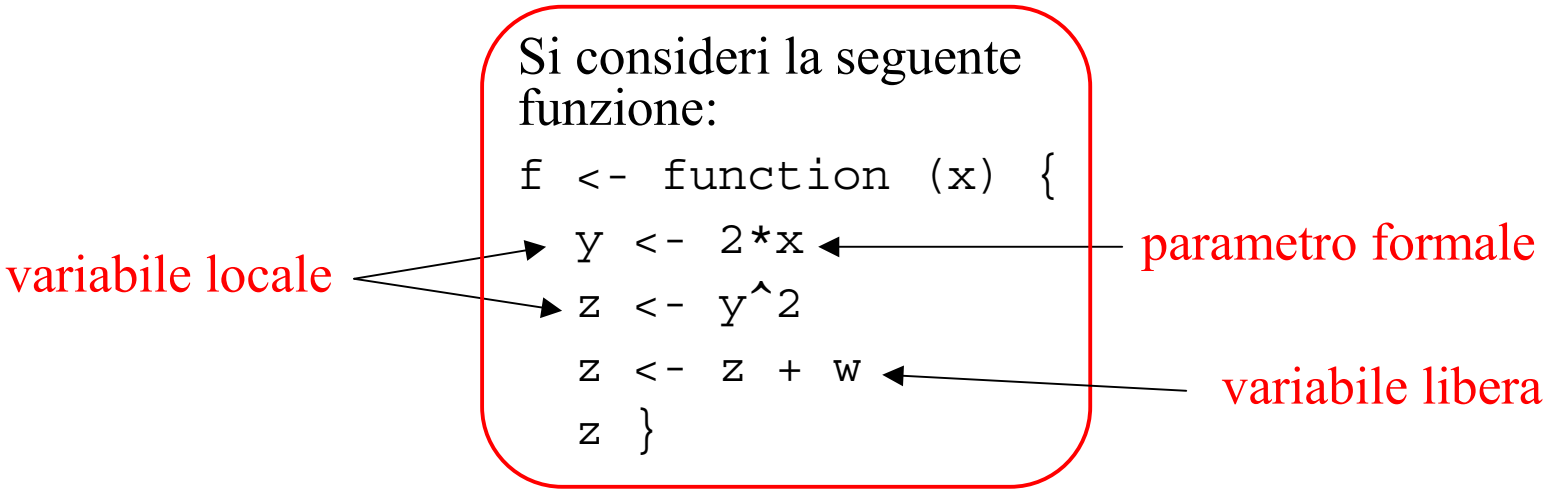
Le variabili locali *sono definite dalla valutazione di espressioni nel corpo di una funzione* ed hanno visibilità solo all' inteno della funzione stessa:




```
> f(3)
[1] 36
> y
Error: Object "y" not found
> z
Error: Object "z" not found
```

Variabili libere

Le variabili che non sono nè parametri formali e nè variabili locali sono chiamate variabili libere.
Il binding delle variabili libere viene risolto cercando la variabile nell' ambiente in cui la funzione è stata creata:




```
> f(3)
Error in f(3) : Object "w" not found
> W <- 3
> f(3)
[1] 39
```

L'operatore di "superassegnamento"

- Il passaggio dei parametri alle funzioni avviene per valore.
- Tramite l' operatore di superassegnamento '`<<-`' è però possibile modificare il valore della variabile nell' ambiente di livello superiore.
- N.B. Non si tratta di un passaggio di argomenti "by reference"

```
f <- function (x)
{
  y <- x/2;
  z <- y^2;
  x <<- z-1;
}
```

superassegnamento →

Quando la funzione `f` viene chiamata il valore della variabile `x` viene modificato:

```
> x=1; f(x)
> x
[1] -0.75
```

Se la variabile `x` non viene trovata nell' ambiente top-level, `x` viene creata e le viene assegnato il valore calcolato dalla funzione:

```
> rm(x); f(1)
> x
[1] -0.75
```

Programmazione modulare

Le funzioni R possono richiamare altre funzioni, permettendo in tal modo di strutturare i programmi in modo “gerarchico”:

```
# funzioni di "secondo livello" chiamate dalla funzioni
# P1 e P2
S1 <- function (x) {... }
S2 <- function () {... }


# funzioni di primo livello" chiamate dalla funzione
# principale
P1 <- function (x) { S1(x); ... }
P2 <- function (x) { S2(); S1(x); ... }


# funzione principale del programma R
MainProgram <- function(x,y,z) {P1(x); P2(y); P1(z) ... }
```

Programmazione top-down

- La programmazione modulare consente di affrontare i problemi “dall’ alto al basso” (approccio *top-down*), cercando cioè di partire dal problema principale definito come una funzione (MainProgram nell’ esempio precedente) con determinati ingressi (dati del problema che si vuole risolvere) ed uscite (risposte/soluzioni al problema)
- Dal problema principale si cerca poi di individuare un insieme di sottoproblemi tramite cui sia possibile risolvere il problema principale; i sottoproblemi sono risolti tramite le funzioni P1 e P2.
- A loro volta i sottoproblemi P1 e P2 si possono essere scomposti in sotto-sottoproblemi (implementati tramite le funzioni S1, S2)
- Il processo di scompozione dei problemi “dall’ alto al basso” può proseguire ancora o arrestarsi a seconda della tipologia del problema.
- In generale tale approccio non è lineare, ma richiede raffinamenti successivi
- R consente anche altri tipi di approcci al design del software (ad es: approccio bottom-up, object-oriented)

Esercizi

1. Scrivere la funzione *echo* che stampa sullo schermo i suoi argomenti.
2. Scrivere una funzione *analyze_string* che ricevuto in ingresso una stringa arbitraria calcoli la frequenza dei simboli componenti la stringa stessa
3. Scrivere una funzione che calcoli i numeri di Fibonacci
4. Scrivere un programma *analyze* che calcoli alcune semplici statistiche relative a 5 diverse tipologie di analisi. In particolare analyze deve:
   a. Leggere da un file una matrice con un numero arbitrario di righe (ogni riga rappresenta un campione) e con 5 colonne che rappresentano dati numerici relativi a 5 diverse analisi.
   b. Trasformi la matrice in un data frame con variabili var1,var2,...,var5.
   c. Memorizzi il data frame in un file
   d. Per ogni variabile calcoli media, deviazione standard.
   e. Stampi sullo schermo i valori relativi a media e deviazione standard per ogni variabile
5. Scrivere una funzione *CalcCovCor* che calcoli le matrici di covarianza e di correlazione fra *n* variabili i cui valori siano generati casualmente. La funzione deve permettere di specificare il numero delle realizzazioni (campioni) generati casualmente ed il tipo di generazione (secondo la distribuzione uniforme o gaussiana). Le matrici vanno poi memorizzate in 2 diversi file (i cui nomi devono essere specificati dall' utente).

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Strutture di controllo

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Programmi in R
come valutazione sequenziale di istruzioni

```
golub <-
  function(x,y)  {
  mx <- mean(x);
  my <- mean(y);
  vx <- sd(x);
  vy <- sd(y);
  g <- (mx-
  my)/(vx+vy);
  g
}
```

- Ogni istruzione è valutata in sequenza.
- Le singole istruzioni possono essere separate da “;” o <new line>

Controllo del flusso di esecuzione di un programma

- I programmi sono eseguiti sequenzialmente, istruzione dopo istruzione, ma in alcuni casi il *flusso di esecuzione* può scegliere vie alternative o ripetersi ciclicamente.
- In R esistono strutture di controllo specifiche per regolare il flusso di esecuzione di un programma:
  - *Blocchi di istruzioni*
  - *Istruzioni condizionali*
  - *Istruzioni di looping*

Sequenze e blocchi di istruzioni - 1

- Le sequenze di istruzioni possono essere separate da
<new line> o “;”:

```
> 3+2- # istruz. Terminata da new line

+ 1     # l’interprete avverte che l’ istruz. non è
# sintatticamente completa

[1] 4

> 3+2-;  # il “;” forza la valutazione dell’
# espressione

Error: syntax error
```

Sequenze e blocchi di istruzioni - 2

- Le istruzioni possono essere raggruppate insieme utilizzando le parentesi graffe. Una sequenza di istruzioni fra parentesi graffe costituisce un blocco.

Esempio:

```
> { x<-0;
+   y<-1;
+   x+y
+   x*y
+   z<-x*y;
+   z-1;
+ }
[1] -1
```

- Si noti che i blocchi vengono valutati solo dopo la chiusura delle parentesi graffe.
- Si può pensare ad un blocco come ad un' unica macro istruzione costitutita da una sequenza di istruzioni

Istruzioni condizionali: l' istruzione if ... else

L' istruzione if ... else permette *flussi alternativi di esecuzione* dipendenti dalla valutazione di una *condizione logica*.

Sintassi:

```
if (condizione)
     blocco1
else
     blocco2
```

Semantica:

se la condizione è vera viene eseguito il `blocco1` altrimenti viene eseguito il `blocco2` .

If..else: esempi

Es.1:

```
if (x>=0)
  print("x è positivo")
else
  print("x è negativo")
```

Es.2:

```
if (x<=0) {
  y <- x^2;
  z <- log2(1+y);
}
else
    z <-log2(x);
```

Es.3:

Il ramo else può anche essere assente:

```
if (x<0)
  x <- -x;
sqrt(x)
```

L' istruzione `sqrt(x)` viene sempre eseguita, mentre `x<--x` viene eseguita solo se `x` è negativo.

Es.4:

Se la condizione assume un valore numerico:

```
if (x)
  print("x è diverso da 0")
else
  print("x è uguale a 0")
```

Istruzione if..else innestate

Le istruzioni if..else possono essere innestate:

```
if (condizione1)
  blocco1
else if (condizione2)
  blocco2
...
else if (condizioneN)
  bloccoN
else
  bloccoN+1
```

La funzione ifelse

L' istruzione ifelse implementa una versione vettorizzata dell' istruzione if..else

Sintassi: `ifelse (condizione, a, b)`

`Condizione`, `a` e `b` sono vettori.

Semantica: ritorna un vettore i cui elementi sono `a[i]` se la `condizione[i]` è vera, altrimenti `b[i]`.

Es:

```
> v1 <- round(runif(5)*10)
> v1
[1]  5  2  1  0 10
> v2 <- round(runif(5)*10)
> v2
[1] 8 7 7 6 5
> ifelse(v1>v2,v1,v2)
[1]  8  7  7  6 10
```

La funzione switch

- La *funzione* switch consente di scegliere fra opzioni multiple.
- La sua semantica è simile a quella dell' omonima struttura di controllo di altri linguaggi di programmazione.

Sintassi:

`switch (istruzione, lista)`

Semantica:

Viene valutata `istruzione` e viene ritornato un `valore`. Se `valore` è un numero compreso fra 1 e lunghezza della lista, allora viene valutato il corrispondente elemento della lista e viene ritornato un risultato. Se `valore` è troppo grande o troppo piccolo viene ritornato NULL.

La funzione switch: esempi

```
> x <- 3
> switch(x, 2+2, mean(1:100), rnorm(3))
[1] -0.3393166  0.1595591 -0.2016252
> x <- 2
> switch(x, 2+2, mean(1:100), rnorm(3))
[1] 50.5
> x <- 5
> switch(x, 2+2, mean(1:100), rnorm(3))
NULL
```

Se in `switch (espressione, lista con nomi)`

la valutazione di `espressione` ritorna un vettore di caratteri che corrisponde al nome associato ad un elemento della lista, tale elemento viene valutato.

Esempio:

```
> y <- "frutto"
> switch(y, frutto="pera", ortaggio="cavolo",
   legume="fagiolo")
[1] "pera"
```

La funzione switch per eseguire funzioni a "scelta multipla"

```
centro <- function (x, tipo="media") {
 switch(tipo,
        media = mean(x),
        mediana = median(x),
        media_c = mean(x, trim=0.2))
}
```

```
> x<-c(1:5,seq(6,10,by=0.5)); x
 [1]  1.0  2.0  3.0  4.0  5.0  6.0  6.5  7.0  7.5  8.0  8.5  9.0
9.5 10.0
> centro(x)
[1] 6.214286
> centro(x,"mediana")
[1] 6.75
```

Istruzioni di loop

- Permettono di ripetere ciclicamente blocchi di istruzioni per un numero prefissato di volte o fino a che una determinata condizione logica viene soddisfatta
- Sono istruzioni la cui struttura sintattica è del tipo:
  loop { `blocco di istruzioni`}
- Esistono diverse forme di istruzioni di loop:
  1. `for`
  2. `while`
  3. `repeat`

Istruzione for

Sintassi:

for (*nome* in *v*)
    blocco di istruzioni

v può essere un vettore o una lista

Semantica:

Gli elementi di *v* sono assegnati ad uno ad uno alla variabile *nome* ed il *blocco di istruzioni* viene valutato ciclicamente fino a che non sono stati esauriti tutti gli elementi di *v*.

Istruzione for: esempi

```
> v = round(runif(50)*5)
> for (i in 1:5) cat(v[i],"  ")
4   4   5   2   3
> for( i in (1: 10)* 5)  cat(v[i],"  ")
3   5   2   5   2   1   1   3   4   0
> for( j in c( 3,1,4,1,5,9,2,7))  cat(v[j],"  ")
5   4   2   4   3   3   4   1
```

L' istruzione *for* può ciclare su qualsiasi tipo di sequenza:

- Es: accedere in sequenza alle componenti di un data frame

```
> for( var in names(data)) {... ; comp<- data$var; ... }
```

- Es: accedere in sequenza a funzioni diverse:

```
> x <- c(pi, pi/2, pi/4) # pi corrisponde a π
> for( f in c(sin, cos, tan)) print(f(x))
[1] 1.224606e-16 1.000000e+00 7.071068e-01
[1] -1.000000e+00  6.123032e-17  7.071068e-01
[1] -1.224606e-16  1.633178e+16  1.000000e+00
```

Istruzione while

Sintassi:

while (*condizione*)
    blocco di istruzioni

condizione è un' espressione logica

Semantica:

condizione viene valutata: se il suo valore è TRUE allora viene eseguito il *blocco di istruzioni*.
Il blocco di istruzioni continua ad essere eseguito ciclicamente se *condizione* rimane TRUE.
Quando *condizione* diventa FALSE allora si esce dal ciclo.

Istruzione while - esempi

```
f <-function(y) {
  i <- 0;
  while (y > 1) {
    y <- y/2;
    i <- i + 1;
  }
  i
}
```

```
> f(1)
[1] 0
> f(2)
[1] 1
```

```
> f(10)
[1] 4
> f(1000)
[1] 10
```

```
> i<-1; while (a[i] < 0) i <- i+1;
```

Ciclo infinito:

```
while (TRUE) {... }
```

Ricerca della prima occorrenza di “UAG” nel vettore di caratteri d:

```
> i<-1; while (d[i] != "UAG" & i <=length(d)) i <- i+1;
```

Istruzione repeat

Sintassi:

repeat

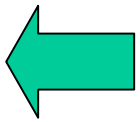
blocco di istruzioni

Semantica:

blocco di istruzioni viene eseguito ciclicamente all' infinito a meno che non venga incontrata una istruzione break che forzi l' uscita dal loop

Istruzione repeat - esempi

```
f1 <-function(y) {
  i <- 0;
  repeat {
    if (y<=1)
      break;
    y <- y/2;
    i <- i + 1;
  }
  i
}
```



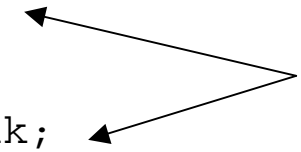
La funzione `f1` è semanticamente equivalente alla funzione `f` precedentemente vista negli esempi per l' istruzione *while*

Ciclo infinito:

```
repeat {... }
```

Si possono prevedere anche più punti di uscita da un repeat:

```
repeat {
   if (a[i]> 0.1) break;
   i<-i+1;
   ...
   if (i>length(a)) break;
   ...
}
```



punti di uscita dal loop

Istruzioni per forzare il ciclo di escuzione dei loop

- L'istruzione break forza l' uscita dai loop:

```
findcodon <-function(s,codon) {
 i<-1;
 while (s[i]!=codon) {
   if (s[i] == "UAA" | s[i] == "UAG" | s[i] == "UGA")  {
     print("Stop codon found");
     break;
   }
   i <- i+1;
 }
 cat("Codon ", s[i], " found in position ",i,"\n");
}
```

- L' istruzione next forza il flusso di esecuzione direttamente al ciclo successivo:

```
for (i in 1:length(a)) {
 if (a[i] == 0)
    next;
 b[i] <- b[i]/a[i];
}
```

Iterazioni e operazioni/funzioni vettorizzate -1

- Molte operazioni e funzioni in R sono *vettorizzate* ed operano elemento per elemento su interi oggetti.
- Utilizzare direttamente operazioni o funzioni vettorizzate *è più efficiente* che effettuare le medesime operazioni utilizzando cicli for.

Esempio: prodotto scalare di due vettori:

```
> a <-rnorm(5); b<-runif(5)
```

A. Calcolo con cicli for:

```
> for(i in 1:5) d[i]<- a[i]* b[i];
> s <-0; for(i in 1:5) s<-s+d[i];
```

B. Calcolo con funzioni vettorizzate:

```
s <- sum(a*b)
```

Iterazioni e operazioni/funzioni vettorizzate -2

- Esistono almeno due buone ragioni per rimpiazzare (dove sia possibile) i cicli for con funzioni/operazioni vettorizzate:
  1. *La velocità*: il loop `for` è molto più lento perchè deve essere valutato ogni volta dall' interprete
  2. *La chiarezza*: è molto più semplice e sintetica l' espressione `sum(a*b)` piuttosto di una serie di cicli `for`.
- Le funzioni vettorizzate includono:
  1. Gli operatori & , | , ! , + , -, * , / , ^ , %%
  2. Funzioni matematiche. Ad es: sin, cos, log, pnorm, choose
  3. Generatori di numeri casuali: rnorm, runif, rpois, ...
  4. L'istruzione ifelse per la valutazione vettorizzata di condizioni logiche.

I comandi “ciclici” della famiglia apply

- I *comandi della famiglia apply* iterano una funzione specificata su insiemi di oggetti.
- La loro sintassi generale è del tipo:
  `comando_apply(insieme_di_oggetti, f)`
  La funzione `f` viene applicata ciclicamente a ciascun oggetto contenuto nell’ `insieme_di_oggetti`.
- Sono semanticamente equivalenti ad un ciclo for del tipo:

```
for (i in insieme_di_oggetti)
    f(insieme_di_oggetti[i])
```

- In generale la loro esecuzione è più efficiente del corrispondente ciclo for.
- Ne esistono diverse varianti (si veda l’ help in linea):
  lapply ed sapply si applicano a liste; apply si applica ad array; tapply si usa con fattori.

Esercizi

1. Scrivere una funzione *find_value* che ritorni la prima occorrenza (cioè i corrispondenti indici numerici) di un valore arbitrario *val* in una matrice generica *m*.
2. Scrivere una funzione *find_all_values* che ritorni tutte le occorrenze (cioè i corrispondenti indici numerici) di un valore arbitrario *val* in una matrice generica *m*.
3. Scrivere una funzione *translate* che riceva in ingresso un vettore numerico *num* e ritorni in uscita un vettore a caratteri *out* tale che *out[i]="P"* se *num[i]>0*, altrimenti *out[i]="N"*.
4. Scrivere una funzione *op* che esegua una sola fra le operazioni di somma, sottrazione, divisione o moltiplicazione dei suoi argomenti, a seconda di un opportuno parametro scelto dall' utente.
5. Tradurre la funzione *switch* (istruzione, lista) nella corrispondente serie di istruzioni if..else. Implementare una funzione *myswitch*(istruzione,lista) semanticamente equivalente alla funzione R switch.
6. Scrivere una funzione *calc_mean_var* che calcoli la media e la varianza di una lista di vettori ricevuti in ingresso.

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

L' ambiente grafico di R

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Rappresentazioni grafiche in R

- Il linguaggio R è dotato di un ambiente grafico potente e versatile
- E' semplice produrre grafici per l' analisi esplorativa dei dati
- Si possono facilmente generare grafici di elevata qualità utilizzabili per pubblicazioni
- L' ampiente grafico di R può generare grafici in diversi formati (sono disponibili diversi *device driver*):
  - Display diretto su schermo (Linux, Windows e Macintosh)
  - postscript
  - pdf (Adobe Portable Document Format)
  - jpeg (JPEG bitmap)
  - png (PNG bitmap, simile a GIF)
  - wmf (Windows Metafile)

Tre gruppi di comandi grafici

1. Funzioni di alto livello:
   creano un nuovo grafico sul device grafico
2. Funzioni di basso livello:
   aggiungono altre parti ad un grafico esistente (ad es: nuove linee, punti o oggetti grafici)
3. Funzioni interattive:
   consentono di aggiungere o estrarre interattivamente informazioni grafiche da un grafico esistente.

Per un esempio di funzioni grafiche in R si esegua:

```
> demo(graphics)
```

Comandi di alto livello

plot	La funzione più utilizzata: permette di generare diverse tipologie di grafici (per punti, linee, grafici a barre, etc)
qqnorm, qqline, qqplot	Grafici per confrontare diverse distribuzioni
hist	Generazione di istogrammi
barplot	Grafici a “colonne”
boxplot	Box-and-whisker plot
pie	Grafici a “torta”
image, contour, persp	Comandi per la grafica 3D

Esistono molti altri comandi di alto livello: si veda ad es: dal menu Help di R: help html/packages/graphics

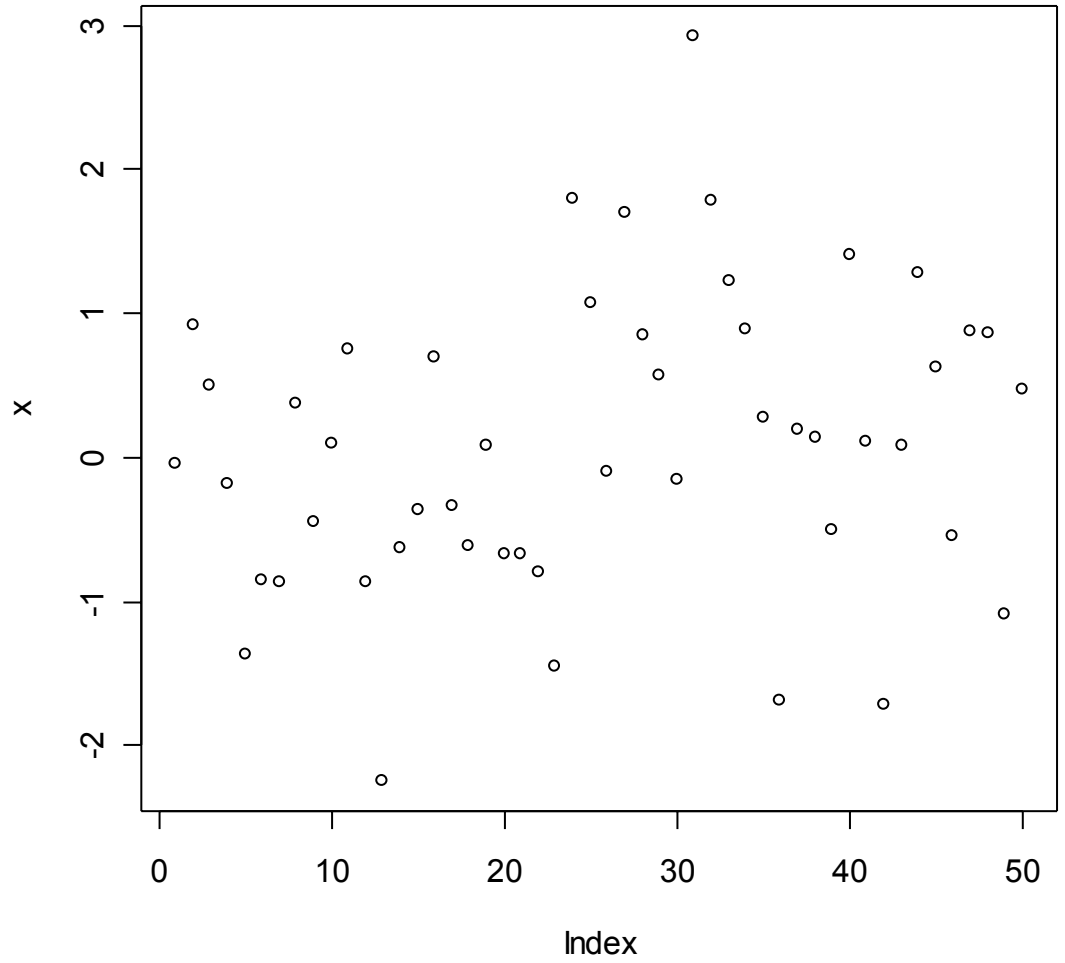
Plot

E' una funzione generica di R: il tipo di grafico generato dipende dal tipo o classe del suo argomento:
Esempi:

- plot(x,y) : se x e y sono vettori produce uno scatterplot di x verso y
- plot(X): se X è una matrice a due colonne produce uno scatterplot di una colonna rispetto all' altra
- plot(x): se x è un vettore produce un grafico dei valori del vettore rispetto agli indici
- plot(f): se f è un fattore viene prodotto un barplot
- plot(f,y): se f è un fattore ed y un vettore numerico, viene prodotto un boxplot di y per ogni livello di f
- plot (df): se df è un dataframe, produce i grafici delle distribuzioni delle variabili contenute nel data frame.

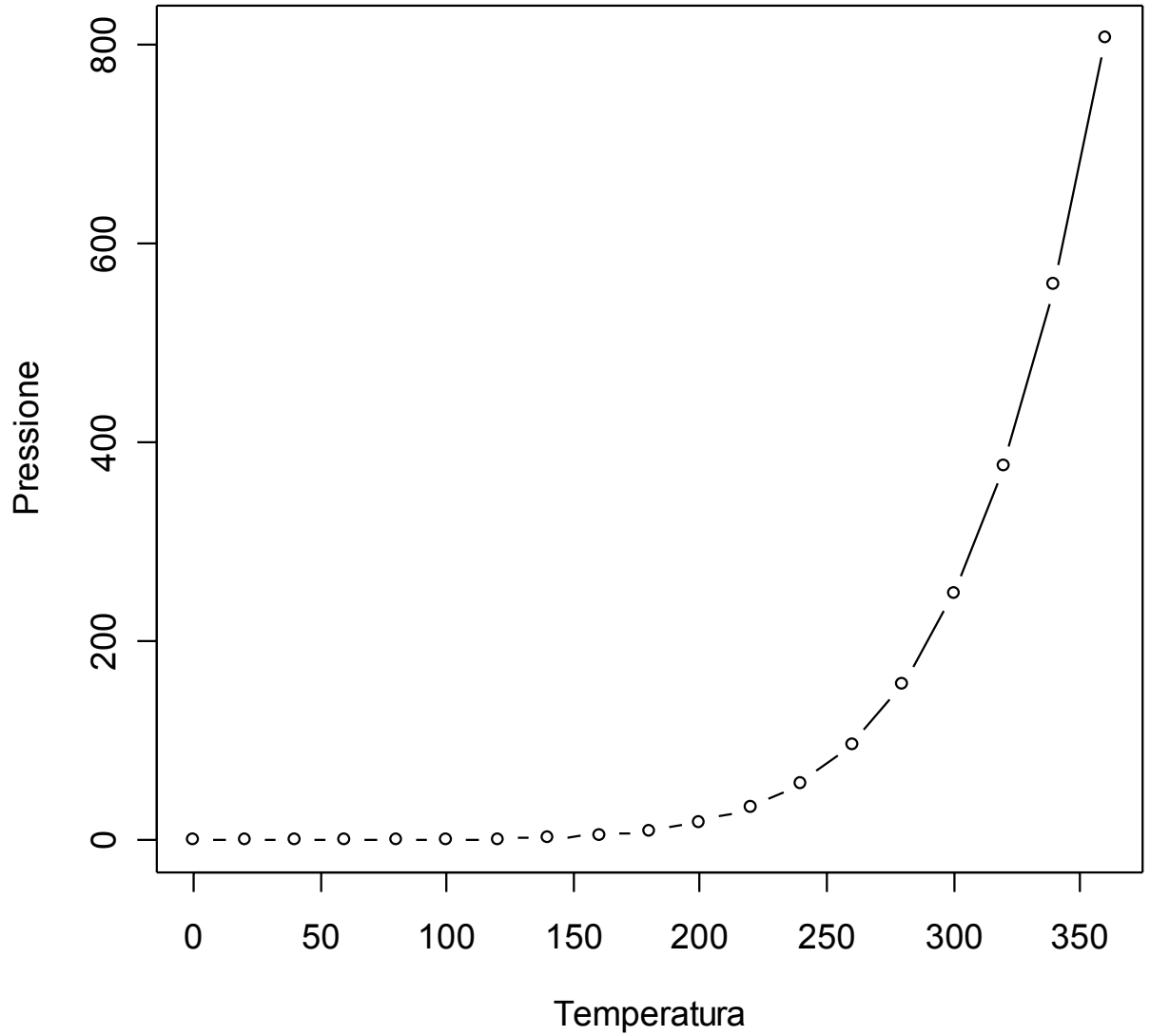
Plot dispone di diversi argomenti (si veda l'help).

Plot di un vettore numerico



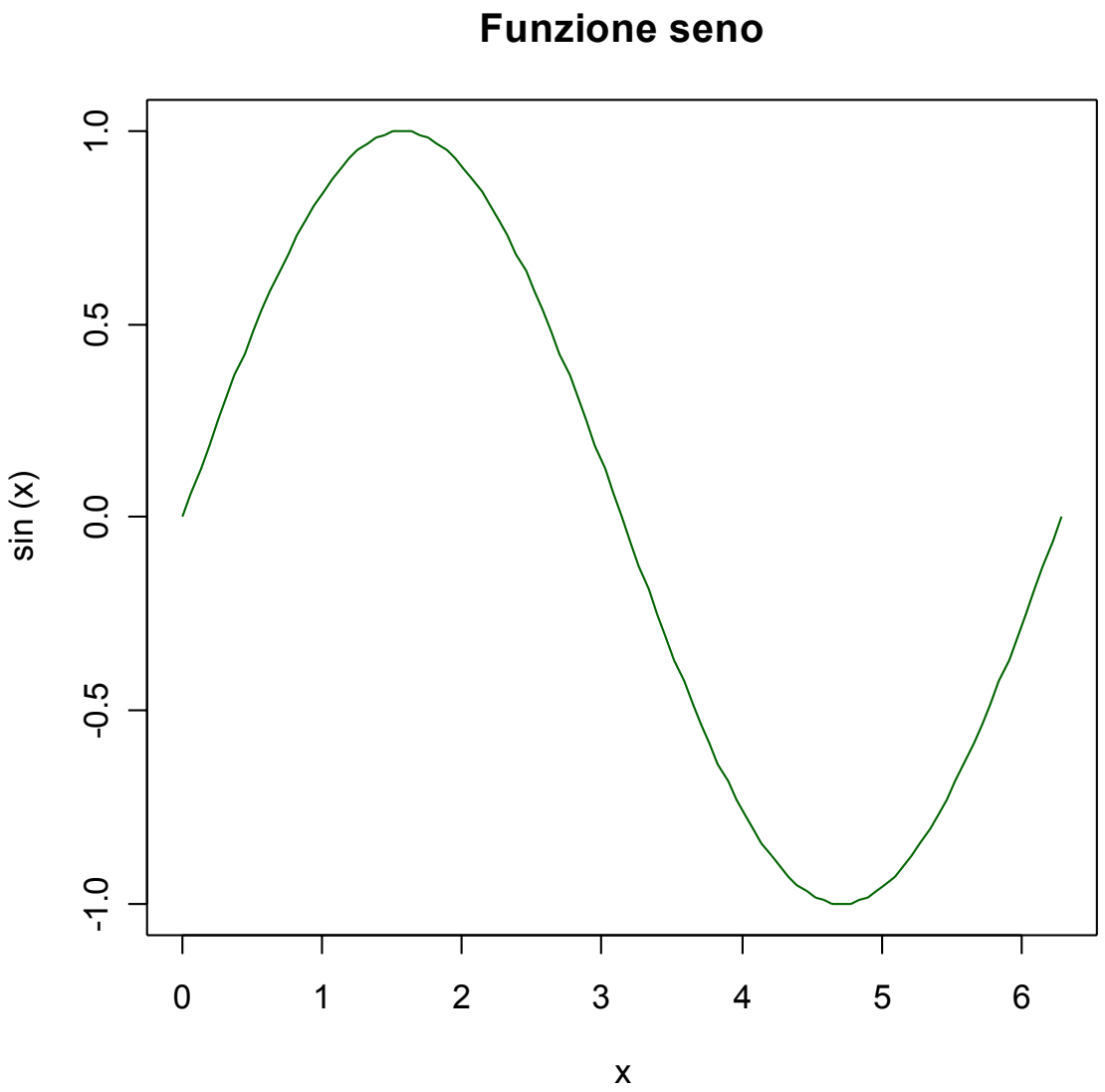

```
> x <- rnorm(50)
> plot(x)
```

Plot di un vettore numerico rispetto ad un altro



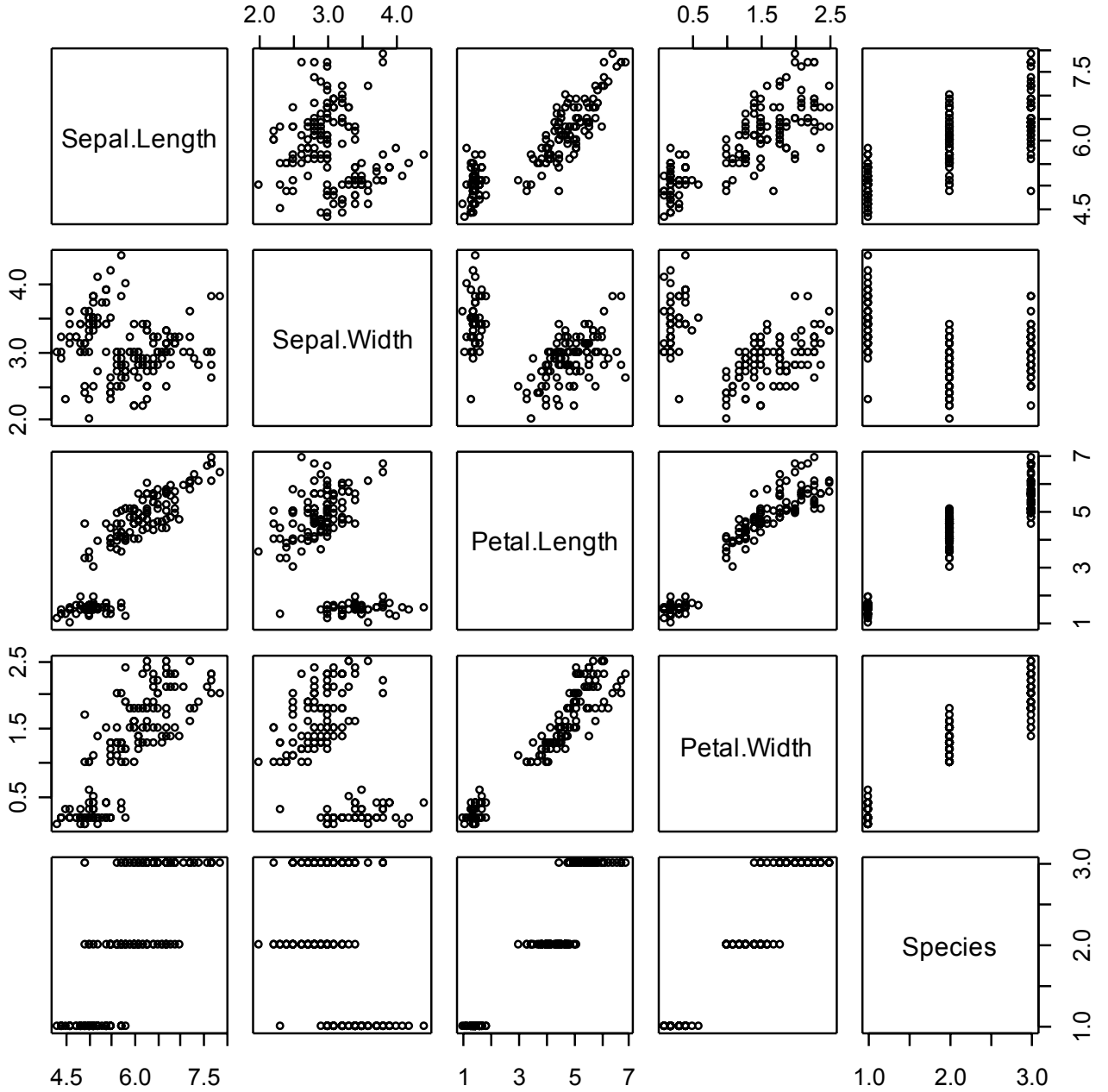

```
> plot(temperature, pressure,
+ xlab="Temperatura", ylab="Pressione", type="b")
```

Plot di una funzione




```
> plot(sin, 0, 2*pi, type="l", col="darkgreen", main="Funzione
   seno")
```

Plot di un dataframe




```
> plot(iris)
```

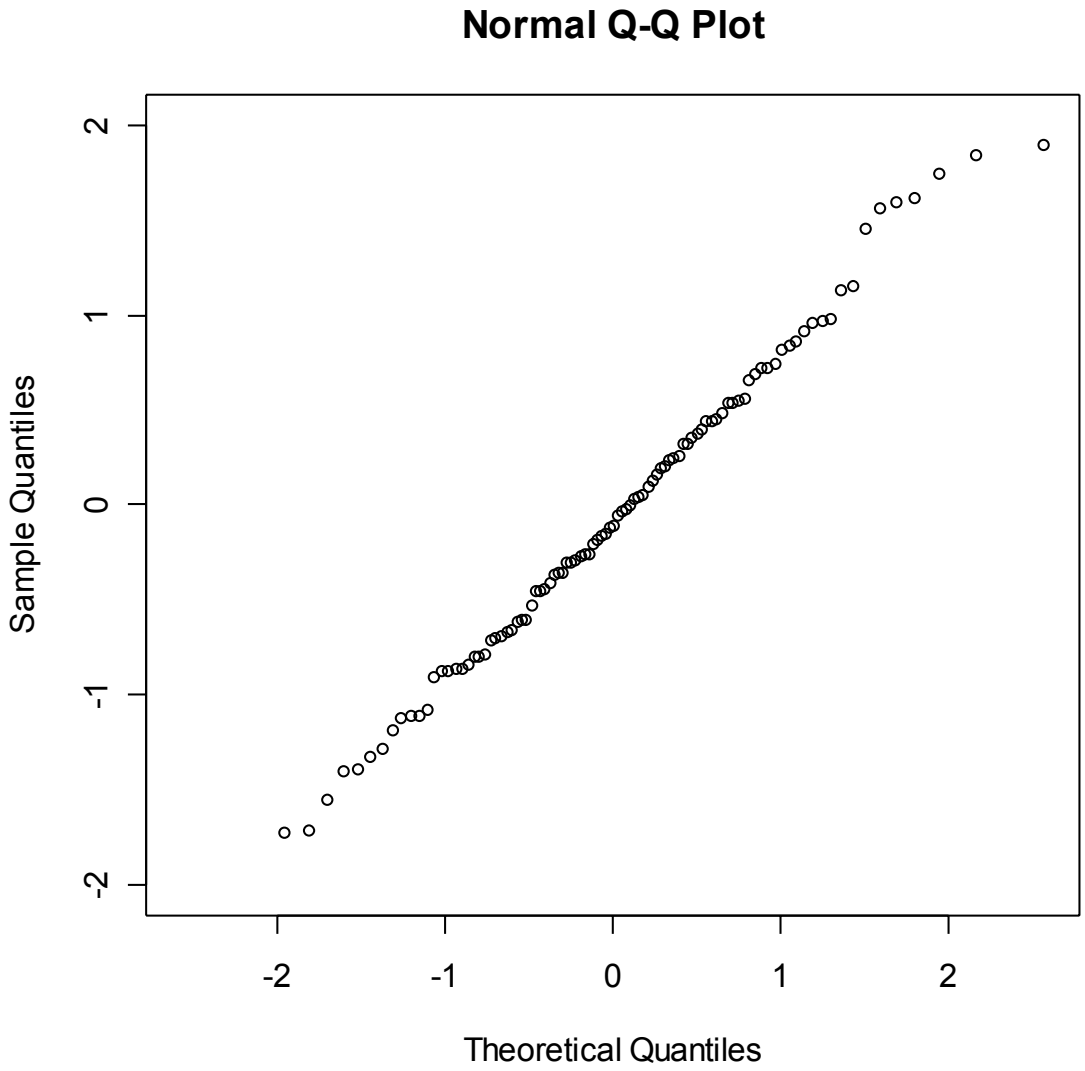
(iris è un dataframe)

qqnorm, qqline e qqplot

Funzioni per confrontare graficamente diverse distribuzioni:

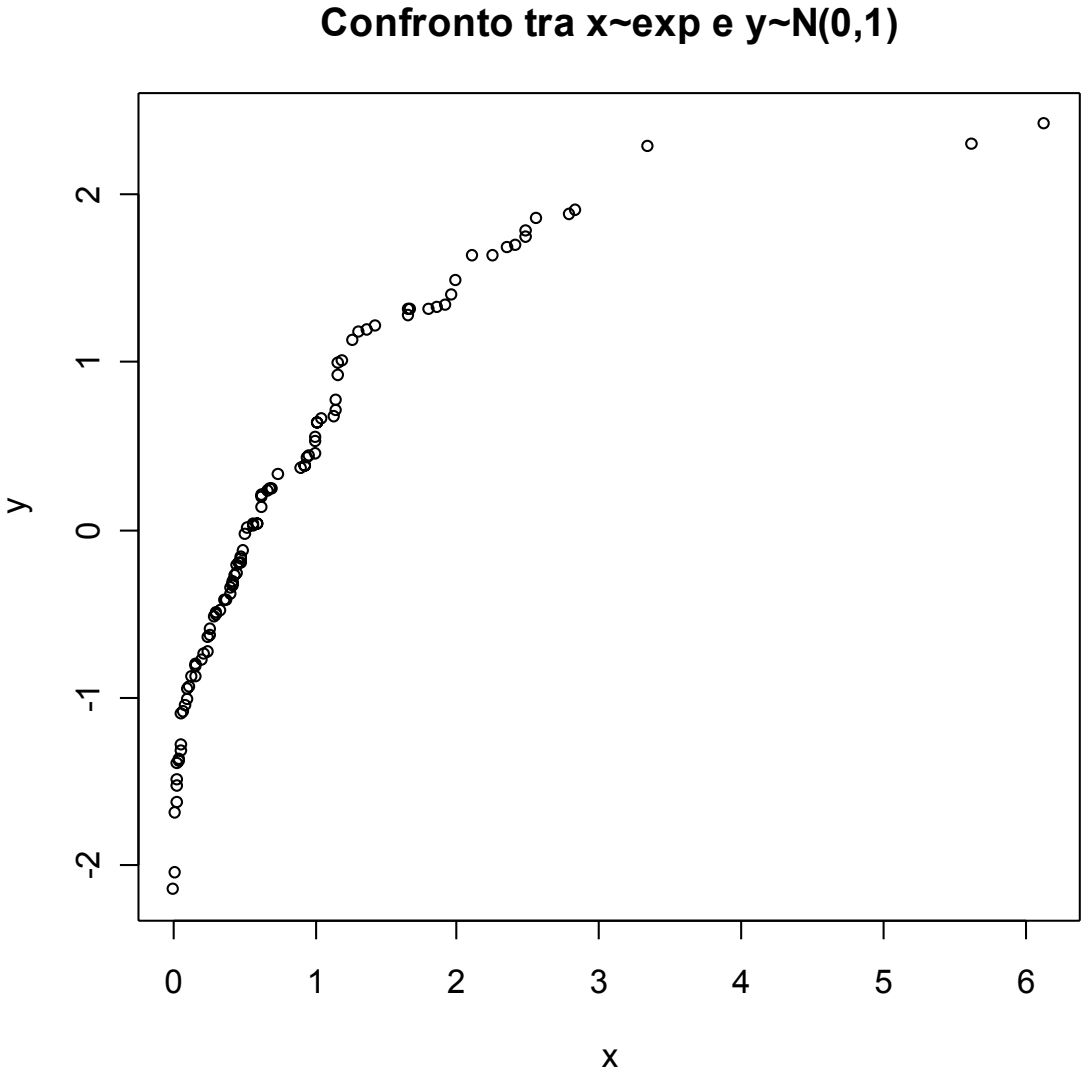
- qqnorm(x):
  produce un grafico quantile-quantile dei dati del vettore x rispetto ad una corrispondente distribuzione normale
- qqline(x):
  come qqnorm, ma viene aggiunta una linea che passa attraverso il primo e terzo quartile
- qqplot(x,y):
  produce il grafico dei quantili dei dati del vettore x rispetto ai quantili dei dati del vettore y

qqnorm: esempio




```
>  z<-rnorm(100);  qqnorm(z)
```

qqplot: esempio




```
> x <- rexp(100); y <- rnorm(100)
> qqplot(x,y, main="Confronto tra x~exp e y~N(0,1)")
```

Hist

hist genera istogrammi utilizzando un vettore numerico.

Esempi:

- hist(x):
  genera un istogramma utilizzando il vettore numerico x
- hist(x, nclass=n):
  genera un isotgramma con un numero n di classi
- hist(x, breaks=b, ...):
  i punti di break degli intervalli dei valori di x che delimitano le classi sono esplicitamente elencati con il parametro breaks
- hist(x, probability=TRUE)
  le colonne rappresentano frequenze relative invece che assolute

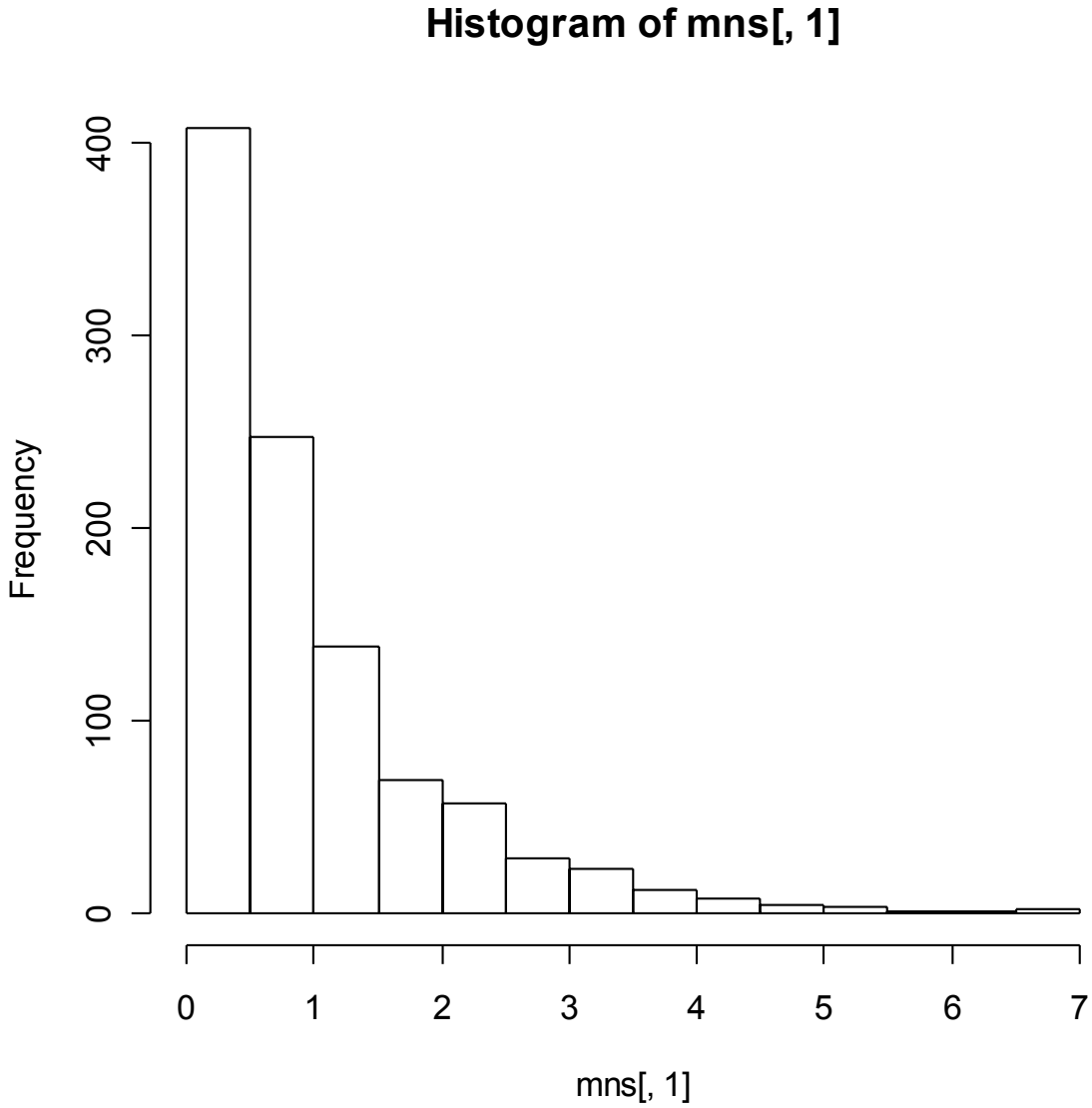
Visualizzazione di distribuzioni tramite istogrammi

In accordo con il teorema del limite centrale si può far vedere come la distribuzione delle medie di campioni estratti da una distribuzione asimmetrica converga ad una normale, al crescere della cardinalità dei campioni:

```
> # generazione dell matrice dei dati estratti in accordo ad
> # una distribuzione esponenziale negativa
> data <- matrix( rexp( 1000 * 32), nrow = 32)
> mns <- cbind( data[ 1, ], # media dei campioni di 1
+ apply( data[ 1: 4, ], 2, mean), # media dei campioni di 4
+ apply( data[ 1: 32, ], 2, mean)) # media dei campioni di 32
```

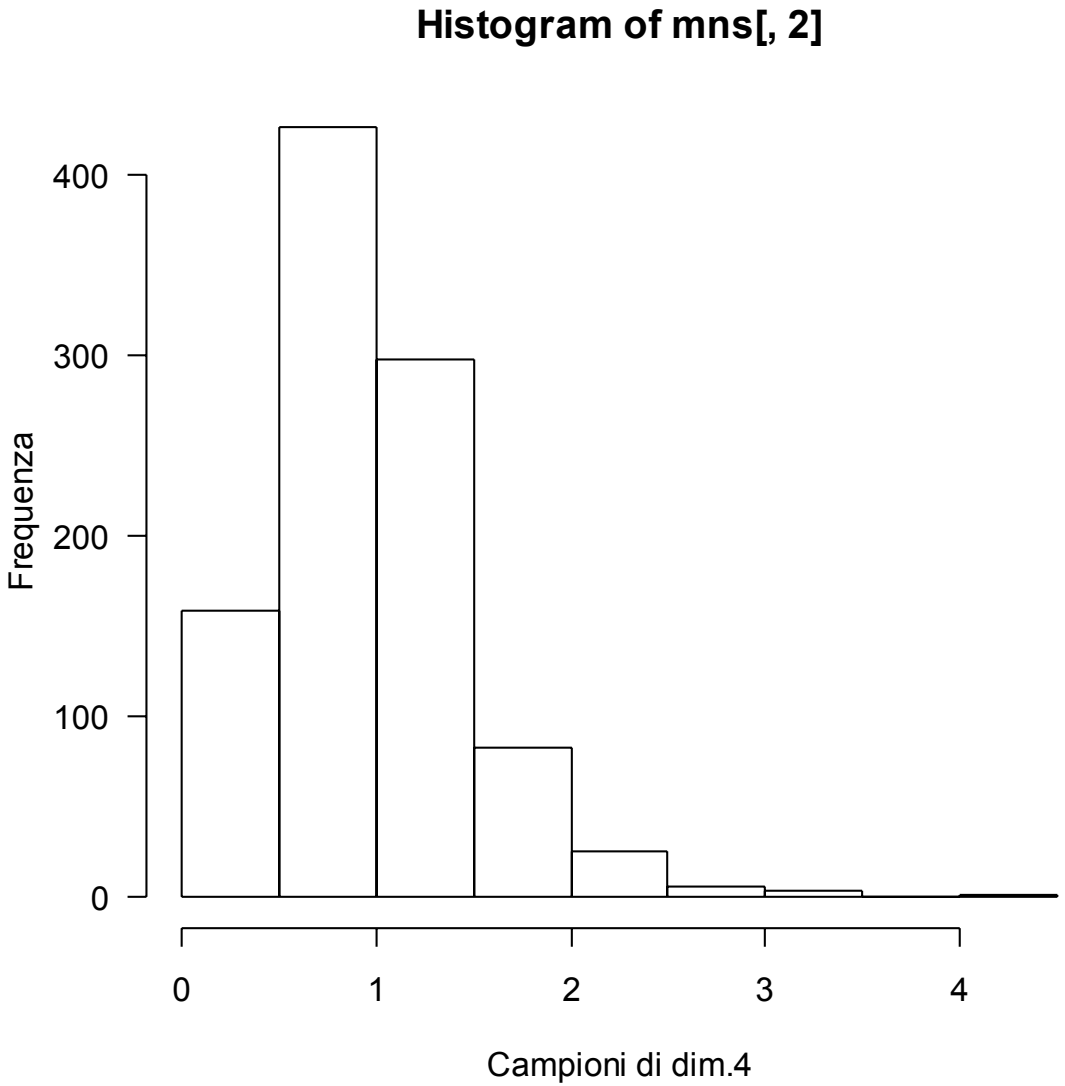
Le distribuzioni possono essere visualizzate con istogrammi (vedi slide successive)

Istogramma delle distribuzione delle medie con n=1



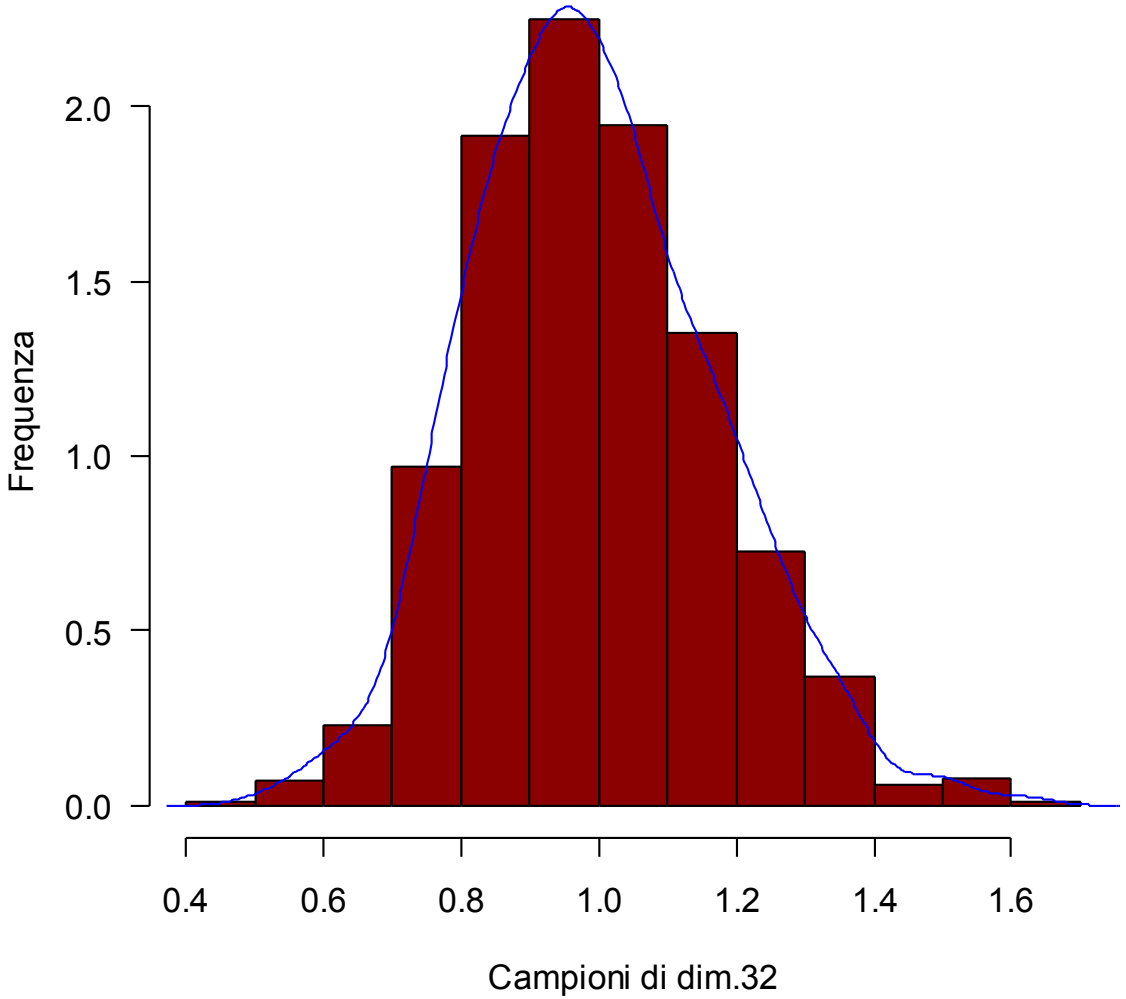

```
> hist(mns[,1])
```

Istogramma delle distribuzione delle medie con n=4




```
> hist( mns[, 2], xlab = "Campioni di dim.4", las = 1)
```

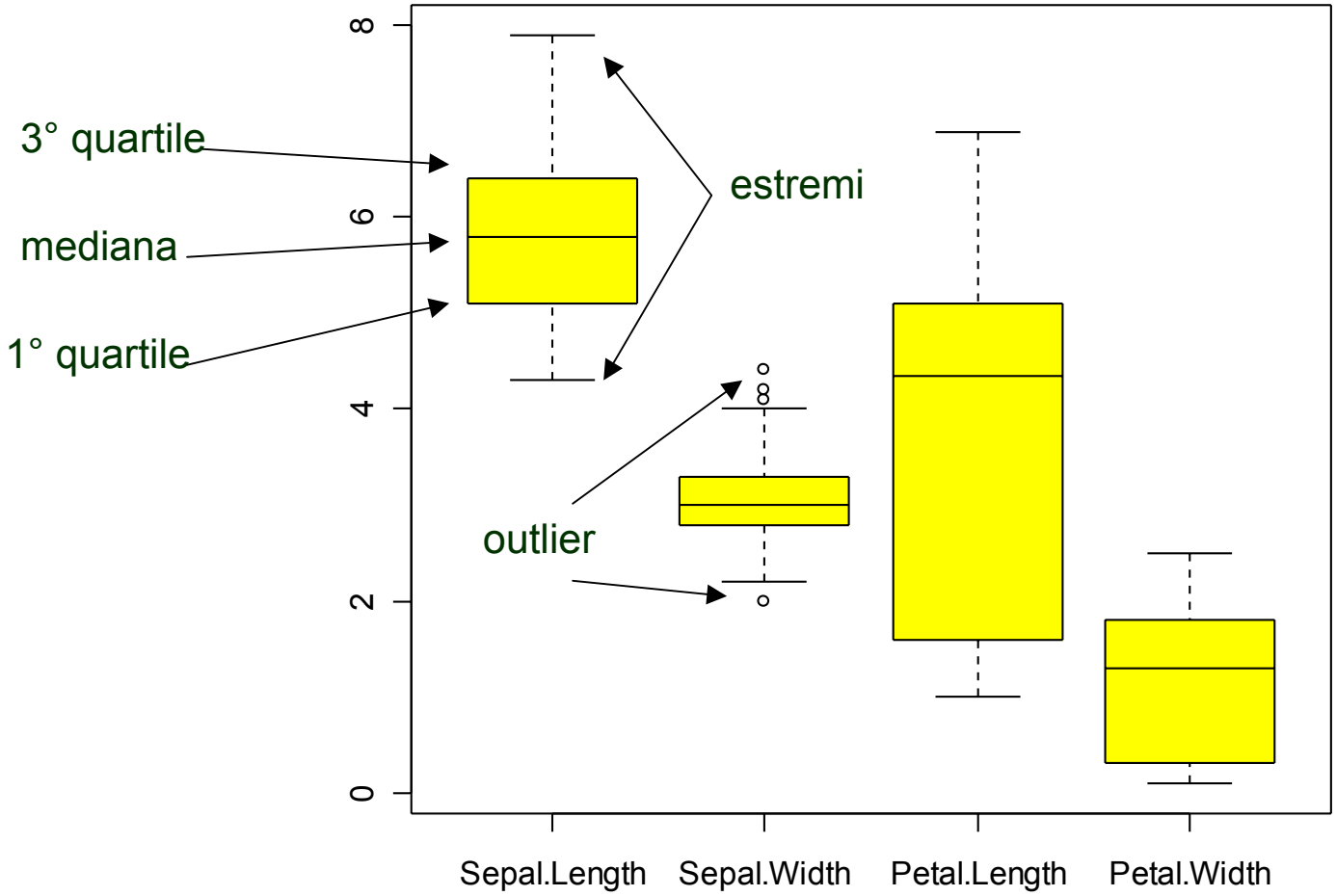
Istogramma delle distribuzione delle medie con n=32




```
> hist( mns[, 3], main="", ylab = "Frequenza", xlab =
+ "Campioni di dim.32", las = 1, col = "darkred", freq=FALSE)
> lines(density( mns[, 3]), col = "blue")
```

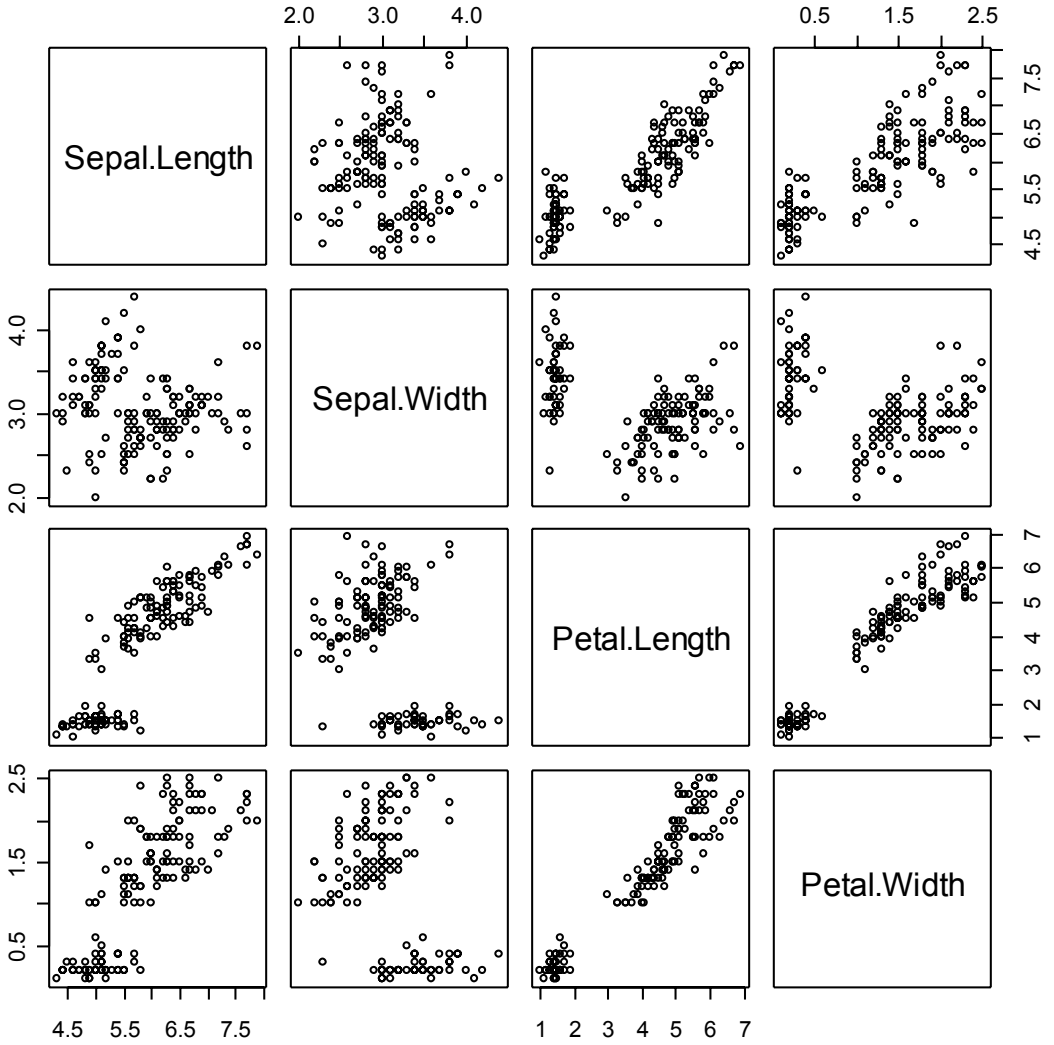
Boxplot

• Forniscono una descrizione grafica sintetica di un insieme di dati utilizzando semplici statistiche.



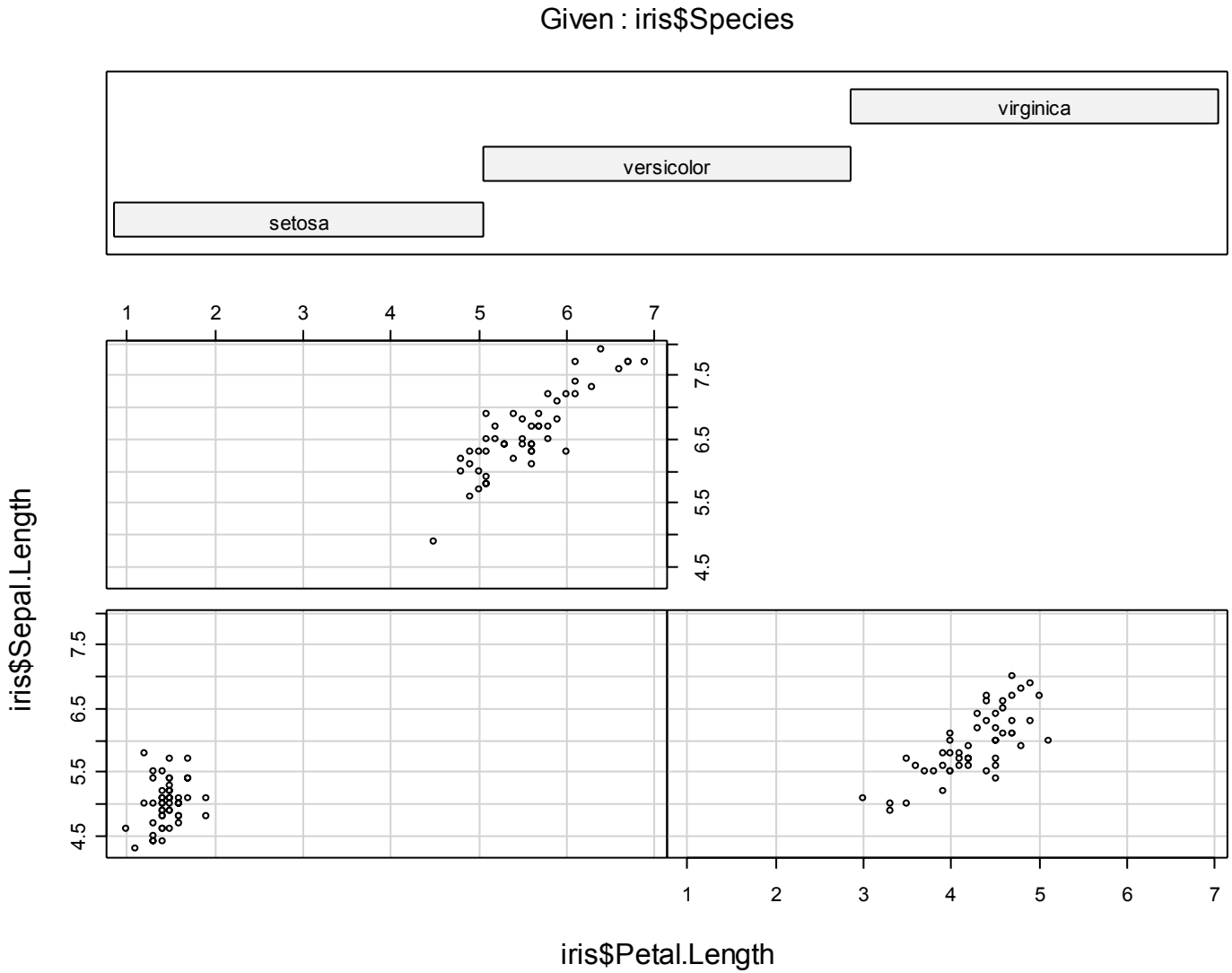

```
> data(iris); boxplot(iris[,1:4], col="yellow")
```

Rappresentazione di dati multivariati -1




```
> pairs(iris[,1:4])
```

Rappresentazione di dati multivariati -2



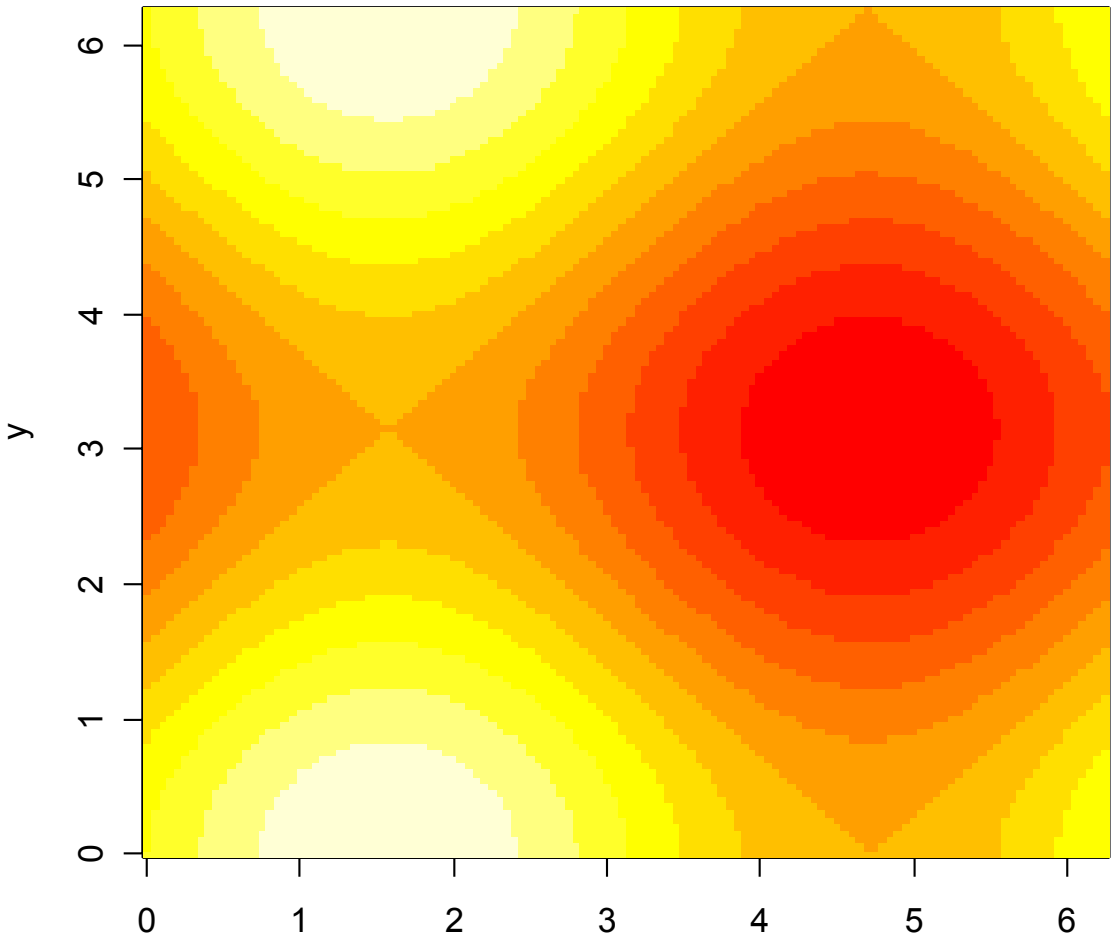

```
> coplot(iris$Sepal.Length ~ iris$Petal.Length | iris$Species)
```

Funzioni per la grafica 3D

- image:
  permette di visualizzare grafici 3D, come immagini 2D, utilizzando diversi toni di colore per le altezze
- persp:
  permette di visualizzare superfici wireframe o a faccette piene
- contour:
  rappresenta una superficie 3D tramite curve di livello

Ognuna di queste funzioni è dotata di diversi parametri che permettono diverse modalità di visualizzazione (v. help)

Image: esempio

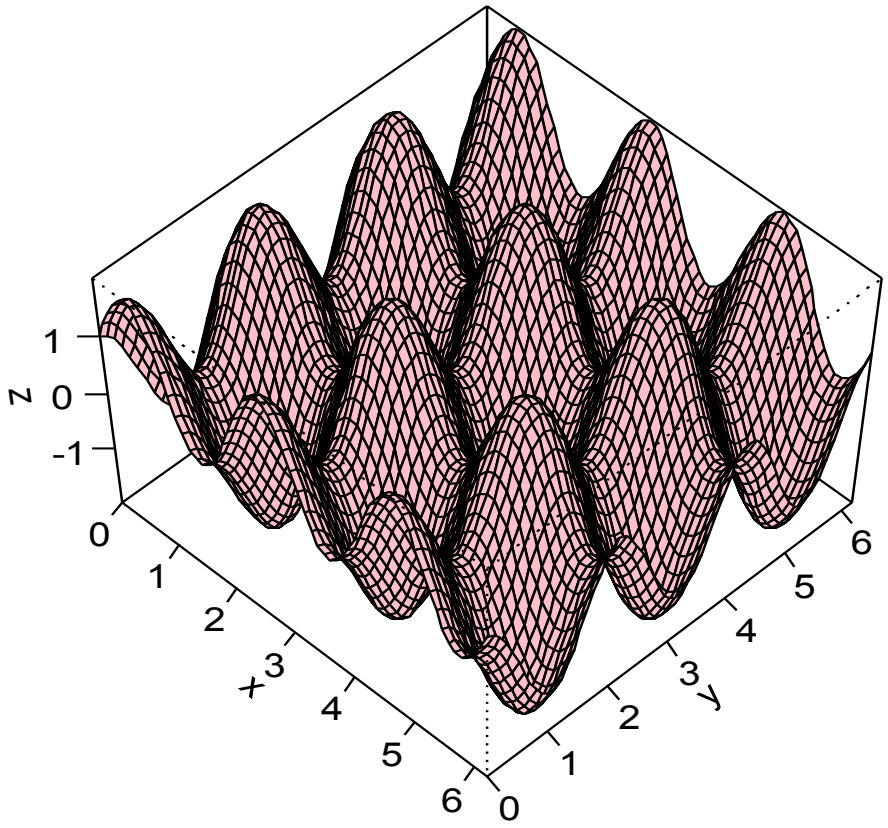


Rappresentazione della funzione:

f(x,y)=sin(x)+cos(y)

```
> x<-y<-seq(0,2*pi,by=0.05)
> z <- outer(sin(x),cos(y),"+") # crea la matrice z delle altezze
> image(x,y,z)
```

persp: esempio

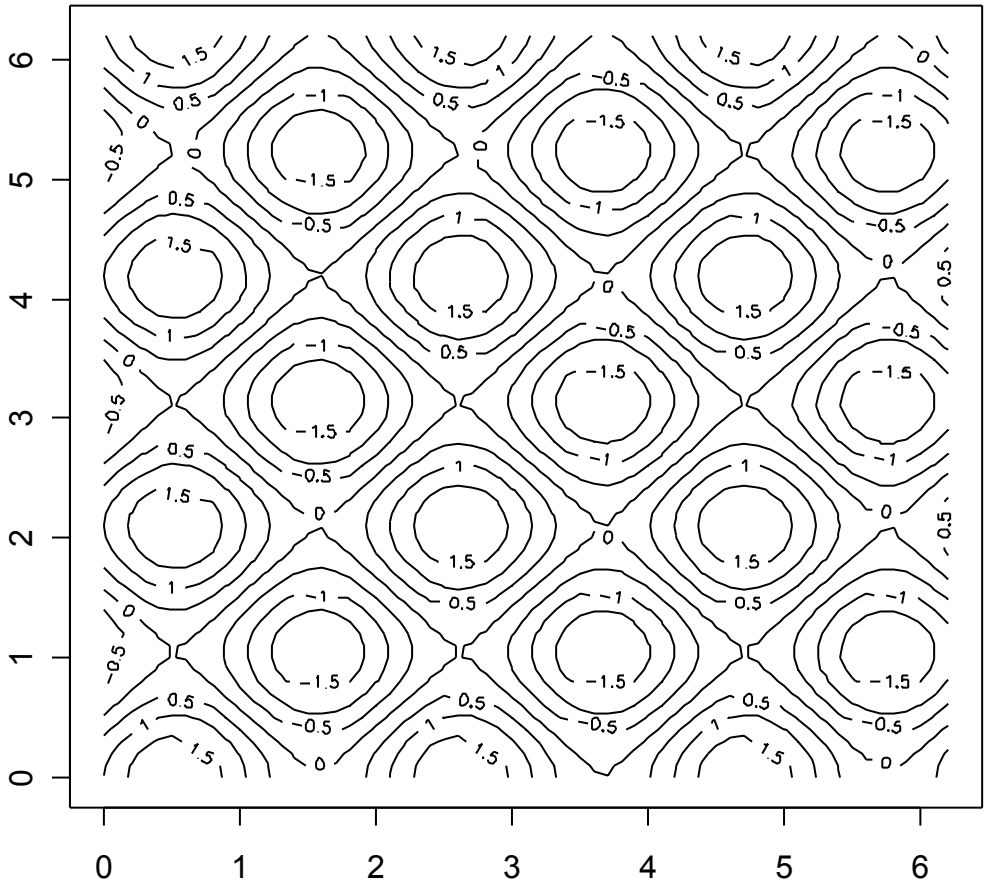


Rappresentazione della funzione:

f(x,y)=sin(3x)+cos(3y)

```
> x<-y<-seq(0,2*pi,by=0.1)
> z <- outer(sin(3*x),cos(3*y),"+") # crea la matrice z delle altezze
>  persp(x,y,z,phi=60,theta=45,d=10,col="pink",ticktype="detailed")
```

contour: esempio



Rappresentazione della funzione:

f(x,y)=sin(3x)+cos(3y)

```
> x<-y<-seq(0,2*pi,by=0.1)
> z <- outer(sin(3*x),cos(3*y),"+") # crea la matrice z delle altezze
>  contour(x,y,z)
```

Argomenti per le funzioni di alto livello

- Si possono passare diversi argomenti aggiuntivi alle funzioni di alto livello:

add=TRUE	Forza la funzione ad agire sul grafico corrente, aggiungendo nuove componenti al grafico
axes=FALSE	Sopprime la generazione automatica degli assi
log="x", log="y", log="xy"	Assi logaritmici
type	Controlla il tipo di plot prodotto: type="p" plot per punti; type="l" plot di linee; type="b" plot di punti connessi da linee; type="h" plot di linne verticali dai punti all' asse. (v. help per descrizione completa delle opzioni disponibili)
xlab=stringa ylab=stringa	Etichette associate agli assi
main=string	Titolo del grafico
…	Esistono diversi altri argomenti specifici o meno per ogni funzione (v. help)

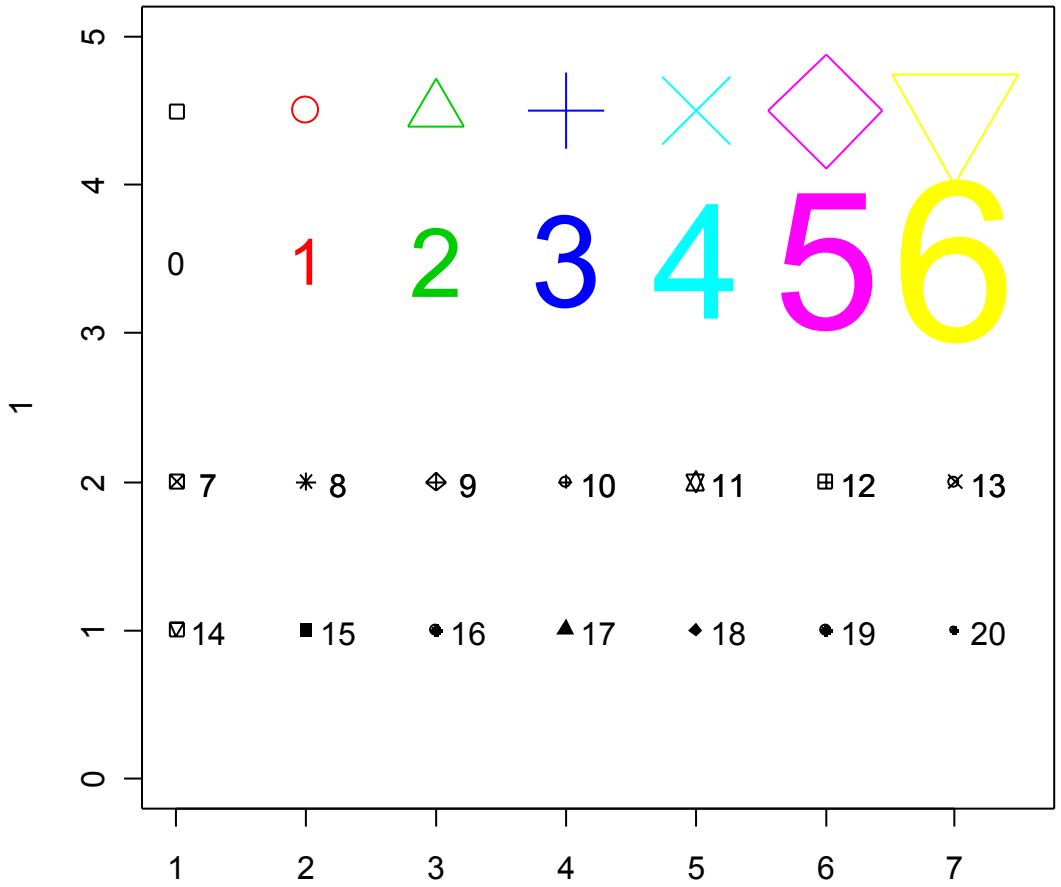
Funzioni grafiche di basso livello

- E' possibile modificare un grafico generato con funzioni ad alto livello con funzioni di basso livello, che possono aggiungere ad es: punti, linee o testo ad un grafico esistente. Alcuni esempi (ma esistono molti altri comandi) sono:

points(x,y)	Aggiunge un punto al grafico corrente (in posizione x,y)
lines(x,y)	Aggiunge una linea al grafico corrente
text(x,y,label)	Aggiunge la stringa di testo label in posizione x,y
abline(a,b)	Aggiunge una linea di inclinazione a ed intercetta b
polygon(x,y,z, ...)	Disegna un poligono i cui vertici (ordinati) sono elencati come argomenti
legend(x,y,legend)	Aggiunge una legenda in posizione x,y
title(main.sub)	Aggiunge il titolo main ed opzionalmente un sottotitolo sub
axis(side,...)	Aggiunge gli assi nelle posizioni specificate da side

Le coordinate sono fornite in termini di *coordinate utente*, definite da precedenti comandi di alto livello

Esempio di funzioni grafiche di basso livello




```
> plot(1, 1, xlim=c(1, 7.5), ylim=c(0,5), type="n")
> points(1:7, rep(4.5, 7), cex=1:7, col=1:7, pch=0:6)
> text(1:7,rep(3.5, 7), labels=paste(0:6), cex=1:7, col=1:7)
> points(1:7,rep(2,7), pch=(0:6)+7)
> text((1:7)+0.25, rep(2,7), paste((0:6)+7))
> points(1:7,rep(1,7), pch=(0:6)+14)
> text((1:7)+0.25, rep(1,7),paste((0:6)+14))
```

Utilizzo dei parametri grafici

- Come abbiamo già visto è possibile modificare il comportamento delle funzioni grafiche
- In R è possibile modificare i parametri grafici secondo due modalità:
  1. Cambiamenti temporanei: cioè tramite il passaggio esplicito di argomenti alle funzioni grafiche (come già visto nelle slide preceenti)
  2. Cambiamenti permanenti: la funzione par permette di accedere e modificare permantemete i parametri del device grafico corrente, fino a che non viene nuovamente chiamata par.

Device driver

- R può generare grafici (a diversi livelli di qualità) per diversi display o dispositivi di stampa.
- Il ruolo dei device driver è di tradurre le istruzioni grafiche di R in una forma “comprensibile” per un particolare dispositivo di visualizzazione
- Per divenire attivo, ogni tipo di device driver dispone di una sua propria funzione di inizializzazione (vedi help(“Devices”) per una lista dei device driver disponibili).
- Successivamente l’ output delle funzioni grafiche viene indirizzato al dispositivo selezionato.

Device driver: esempi

1. Apertura di una finestra in ambiente Windows (device aperto di default sul sistema utilizzato in laboratorio):

```
> windows() #
  l' output delle funzioni grafiche è indirizzato alla
   finestra corrente
> windows() # viene aperta un' altra finestra
  l' output delle funzioni grafiche è indirizzato alla
   nuova finestra
```

2. Creazione di un file grafico postscript

```
> postscript("grafico.ps") # apertura device postscript
> plot(y) # l'output grafico è indirizzato sul file ps
> dev.off() # chiusura del device postscript corrente
```

3. Creazione di un file grafico in formato jpeg

```
> jpeg("grafico.jpg")
> plot(y)
> dev.off()
```

Grafica interattiva

- Con la funzione locator è possibile sia ottenere le coordinate grafiche di un punto sul grafico, sia aggiungere un oggetto grafico in una posizione specifica tramite un click del mouse.
  Per esempio, si provino ad eseguire i seguenti comandi:

```
> y <-rnorm(100)
> plot(y)
> locator(5)
```

```
> y <-rnorm(100)
> plot(y)
> text(locator(1), "Punto critico")
```

- Con la funzione identify è possibile identificare punti particolari di un grafico tramite un click del mouse: per ogni click viene indicato l' indice del punto più vicino al click del mouse.
  Per esempio, si provino ad eseguire i seguenti comandi:

```
> y <-rnorm(100)
> plot(y)
> identify(y)
```

Grafica dinamica

- Attualmente R non ha funzioni grafiche built-in per supportare la grafica dinamica (ad es: rotazione di nuvole di punti, trasformazioni metriche o simili di grafici)
- Esiste comunque il *package R* xgobi che permette di accedere alle funzioni di grafica dinamica disponibili nel sistema Xgobi di Swayne, Cook e Buja: http://www.research.att.com/areas/stat/xgobi

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Debugging

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Debugging in R

- Gli strumenti per il debugging permettono di “congelare” l’ esecuzione in particolari punti del codice e di controllare lo stato dell’ esecuzione, eseguire il codice istruzione per istruzione e di visualizzare il contentuo delle variabili e dello stack.
- Esistono diversi strumenti per effettuare il debugging di programmi scritti in R:
  1. Funzione browser
  2. Funzione debug
  3. Funzione trace/untrace

Browser

- Una chiamata alla funzione browser all' interno di una funzione R ferma l' esecuzione nel punto della chiamata e fornisce all' utente un prompt da cui può eseguire comandi per verificare lo stato dell' esecuzione.
- Alcuni comandi eseguibili dal prompt del browser:
  - <RET> opp. 'c' : continua l' esecuzione
  - 'n' : esegue l'istuzione successiva
  - get("v") : ritorna il valore della variabile v
  - 'where' : ritorna lo stack delle chiamate
  - 'Q' : termina l' esecuzione e salta al prompt di R

Utilizzo della funzione browser: esempio

```
golub <-
function(x,y)  {
  browser();
  mx <- mean(x);
  my <- mean(y);
  vx <- sd(x);
  vy <- sd(y);
  g <- (mx-
my)/(vx+vy);
  g
}
```

```
> golub(x,y)
Called from: golub(x, y)
Browse[1]> c
[1] 0.408709
> golub(x,y)
Called from: golub(x, y)
Browse[1]> n
debug: mx <- mean(x)
Browse[1]> n
debug: my <- mean(y)
Browse[1]> n
debug: vx <- sd(x)
Browse[1]> get("my")
[1] 0.1161415
Browse[1]> where
where 1: golub(x, y)
Browse[1]> Q
>
```

Debug

- Il comando debug(fun) invoca il debugger sulla funzione fun. Le successive chiamate alla funzione fun lanciano il debugger
- Il debugger consente la valutazione delle istruzioni nel corpo della funzione sottoposta a debug
- I comandi eseguibili dal prompt sono simili a quelli della funzione browser.
- Il debugging viene inattivato dalla funzione undebug(fun)

Utilizzo della funzione debug: esempio

```
> debug(golub)
> golub(x,y)
debugging in: golub(x, y)
debug: {
    mx <- mean(x)
    my <- mean(y)
    vx <- sd(x)
    vy <- sd(y)
    g <- (mx - my)/(vx + vy)
    g
}
Browse[1]>
debug: mx <- mean(x)
Browse[1]>
debug: my <- mean(y)
Browse[1]>
debug: vx <- sd(x)
```

```
Browse[1]> mx
[1] 0.9413439
Browse[1]> get("mx")
[1] 0.9413439
Browse[1]> where
where 1: golub(x, y)

Browse[1]> c
exiting from: golub(x, y)
[1] 0.408709

> undebug(golub)
> golub(x,y)
[1] 0.408709
```

Trace/untrace

- Il comando trace(fun) fa sì che ogni volta che la funzione fun venga valutata, la corrispondente chiamata venga stampata sullo schermo.
- *trace* consente quindi di seguire l'esecuzione delle funzioni specificate chiamate all' interno di un determinato programma.
- Il comando untrace(fun) disattiva tale meccanismo di "tracciamento" della funzione.

Esempio:

```
> trace(golub)
> golub(x,y)
trace: golub(x, y)
[1] 0.408709
> untrace(golub)
> golub(x,y)
[1] 0.408709
```

Università degli Studi di Milano

Laurea Specialistica in Genomica Funzionale e Bioinformatica

Corso di Linguaggi di Programmazione per la Bioinformatica

Package

Giorgio Valentini

e –mail: *valentini@dsi.unimi.it*

DSI – Dipartimento di Scienze dell' Informazione

Università degli Studi di Milano

Package in R

I package forniscono uno strumento semplice ed efficiente per gestire collezioni di funzioni e di dati (librerie) e la relativa documentazione.

I package R forniscono librerie liberamente disponibili scritte da sviluppatori esperti in diversi domini applicativi.

Caratteristiche dei package R:

- Caricamento dinamico in memoria: vengono caricati in memoria quando necessario e possono essere scaricati in qualsiasi momento.
- Facili da installare ed aggiornare: le funzioni, dati e documentazione sono installati con un singolo comando che può essere eseguito dall' interno o dall' esterno di R.
- Estendibili ed adattabili alle esigenze degli utenti: gli utenti possono creare propri package (si veda il manuale disponibile on line "*Writing R extensions*")

Installare ed aggiornare package

- I package possono essere installati:
  1. direttamente dal sito CRAN: cran.r-project.org. selezionando dal menu *Packages* della finestra *RGui* l'opzione *Install packages from CRAN*
  2. da file locali compressi tramite il programma *zip* (file "zippati") selezionando dal menu *Packages* della finestra *RGui* l'opzione *Install packages from local zip files*
- Analogamente l' aggiornamento di package già installati può essere effettuato selezionando dal menu *Packages* l'opzione *Update packages from CRAN.*

Caricare in memoria i package

- Le funzioni ed i dati dei package devono essere cariacati in memoria perchè possono essere utilizzabili.
- Per caricare un package si può scegliere uno dei modi seguenti:
  - Dal menu *Packages* selezionare l'opzione *Load package*
  - Dal prompt digitare: *library*(*nome_package*)

Esempio:

```
> library() # elenca i package correntemente installati

> library(cluster) # package per il clustering

> library(marray)  # package per il preprocessing di
                     dati di cDNA microarray
```

Documentazione sui package

- Per avere accesso alla documentazione sui package installati nel sistema si può selezionare dal menu *Help* l' opzione *HTML help*. Selezionando *Packages* dalla pagina del browser si può scegliere dalla lista dei package il package desiderato ed accedere alle informazioni dettagliate in formato HTML.
- Dal prompt si può anche digitare *help*(*package=xxx*) per avere informazioni sul package *xxx*
- Spesso i package dispongono della medesima documentazione in formato pdf.
- Ogni funzione, insieme di dati, classe ed in generale ogni oggetto disponibile nei package *dispone di una propria documentazione specifica*.

Struttura tipo di una pagina di documentazione

Indipendentemente dal formato della documentazione (HTML, pdf, etc), nella pagina di documentazione di una funzione si trovano solitamente le seguente sezioni:

- Nome della funzione e package nella quale è contenuta
- Description: descrizione sintetica delle caratteristiche generali della funzione
- Usage: sintassi (formato della chiamata della funzione)
- Arguments: elenco e descrizione degli argomenti della funzione
- Value: valore riotrnato dalla funzione
- Details: descrizione più dettagliata della funzione (se necessaria)
- References: riferimenti bibliografici utili
- See also: link ad altre funzioni, oggetti correlati alla funzione
- Examples: alcuni esempi di utilizzo della funzione

I package di Bioconductor

- Bioconductor è un progetto software internazionale il cui obiettivo consiste nel fornire strumenti software di alta qualità per attività professionali e di ricerca nell'ambito della bioinformatica
- Nell' ambito del progetto, sono stati sviluppati e sono tuttora in sviluppo diversi package R per la bioinformatica, liberamente scaricabili dal sito di Bioconductor: www.bioconductor.org
- Nelle prossime lezioni vedremo in particolare alcuni package per l' *analisi dei dati di espressione genica* e per *l'analisi di biosequenze*.