1. Programmi e Funzioni

2. Controllo del Flusso di Esecuzione

3. Funzioni I/O (Input/Output)

Marco Notaro

Programmi e funzioni in R

Modo di esecuzione dei programmi in R

• I programmi (sequenze di espressioni) possono essere eseguiti :
– *Interattivamente: ogni istruzione viene* eseguita direttamente al prompt dei comandi
– *Non interattivamente: le espressioni sono lette* da un file (tramite la funzione *source) ed* eseguite dall' interprete una ad una in sequenza.

• Usare un text editor:
-Windows: Notepad++
-Linux: Vim

Funzioni

- Nella lezione precedente avevamo già visto molti esempi di funzioni disponibili in R

- Le funzioni in R possono anche definite dagli utenti

- I programmi in R possono essere realizzati tramite funzioni

Funzioni: sintassi

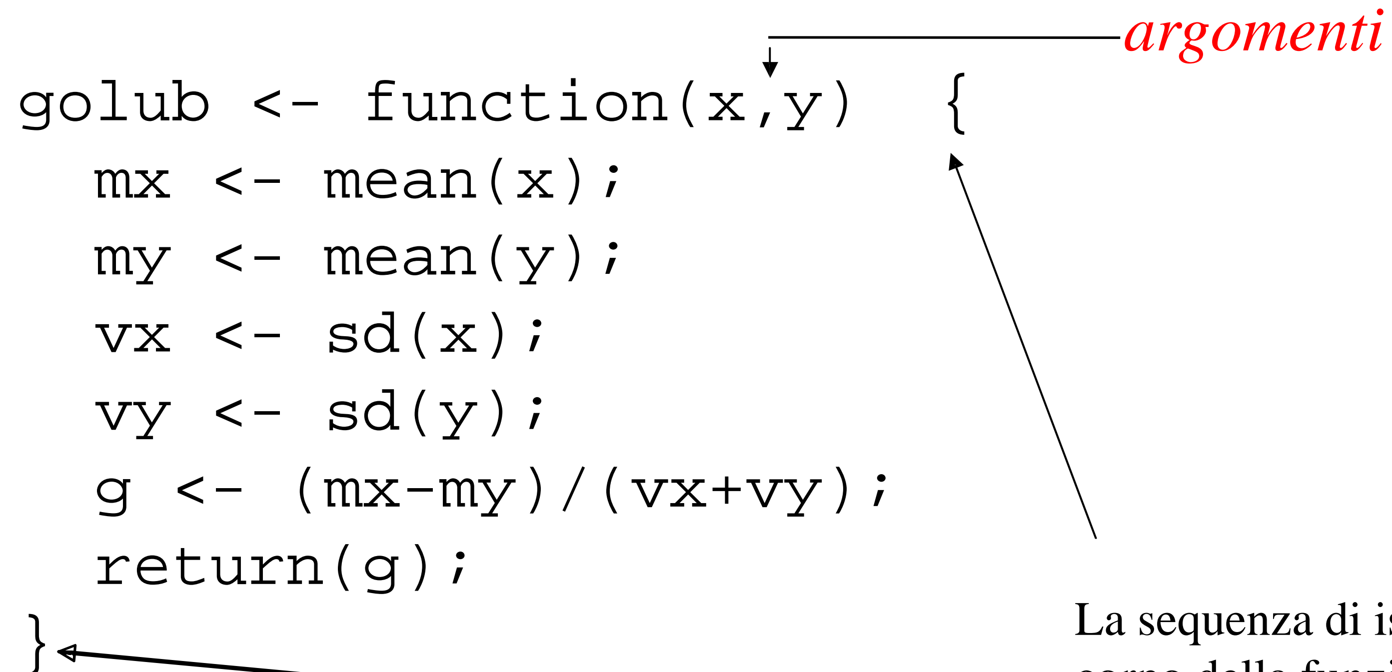
La sintassi per scrivere una funzione è:

`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

Funzioni: esempi (1)

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



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

Funzioni: esempi (2)

Utilizzo della funzione di Golub:

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

1. `> source("golub.R")1`
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 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
> funl(y)
> y
> 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)
```

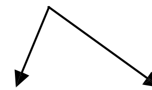
Ad esempio:

```
> 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. Ad esempio:

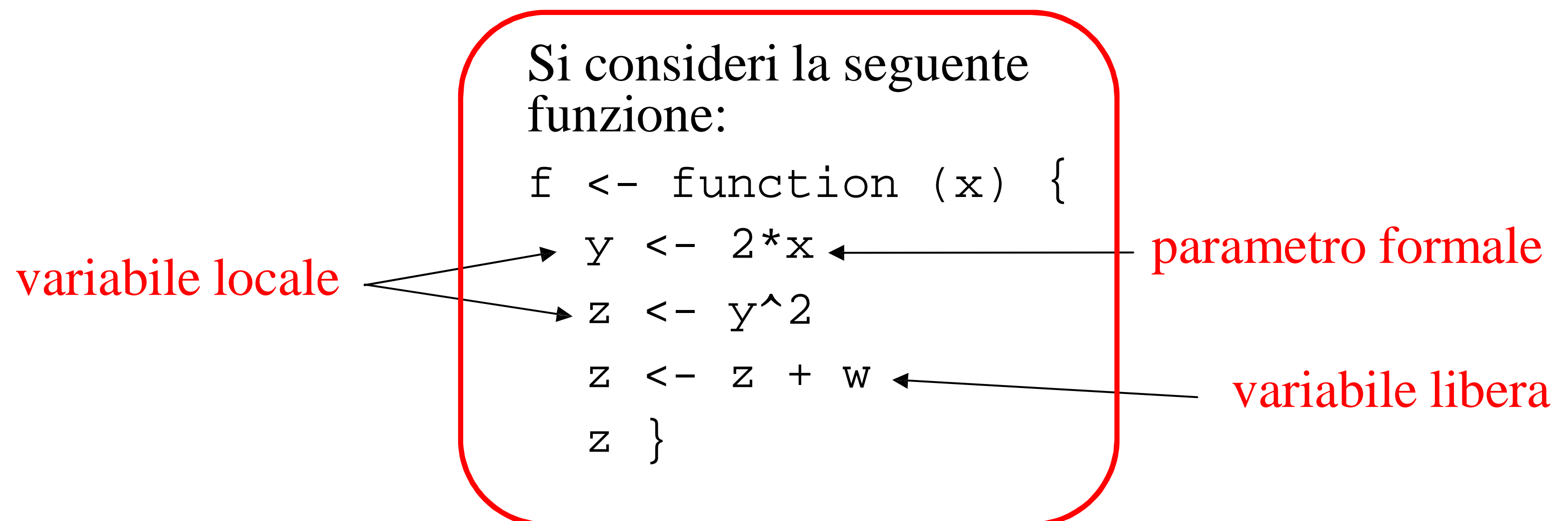
valori di default



```
> 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
```

Parametri formali, variabli locali e 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)1
Error in f(3) : Object "w" not found
> w <- 3
> f(3)1
[1] 39
```

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
# Pl e P2

Sl <- function (x) {...}

S2 <- function () {... }

S3 <- function () {... }


# funzioni di primo livello" chiamate dalla funzione
# principale

Pl <- function (x) { Sl(x); S3(); }

P2 <- function (x) { S2(); Sl(x); ... }


# funzione principale del programma R

MainProgram <- function(x,y,z) {Pl(x); P2(y); Pl(z) ...  }
```

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

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

```
Esempio:
{
  x <- runif(10);
  y <- runif(10);
  mx <- mean(x);
  my <- mean(y);
  vx <- sd(x);
  vy <- sd(y);
  g <- (mx-my)/(vx+vy);
  g;
}
```

- 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.
```

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
```

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. Le principali sono:

1. `for`
2. `while`

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*.

Esempi Istruzione for

```
> 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:*
Ad Esempio accedere in sequenza a funzioni diverse:

```
> x <- c(pi, pi/2, pi/4) # pi corrisponde a p
> 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:

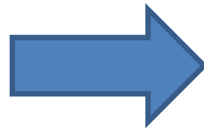
condizione viene valutata: se il suo valore è TRUE allora viene eseguito il *blocco di istruzioni.*

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;
```

Loop *vs* Funzioni Vettorizzate
Who will be the winner?
And…Why?

Esperimento:

1. Implementare la funzione *sum.func*. Tale funzione riceve in input un vettore numerico e ritorna la somma di tutti gli elementi presenti nel vettore
2. Ottenere lo stesso risultato utilizzando la funzione sum
3. Confrontare il tempo di calcolo con system.time

TIPS:

1. Implementare la funzione *sum.fun* utilizzando un piccolo numero di esempi (e.g., x <- sample(10))
1. Verificare che la funzione *sum.func* restituisca un risultato corretto
2. Ripetere l'esperimento utilizzando questa volta un numero elevato di esempi (e.g., x <- round(runif(10000000),4))

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/matrice; tapply si usa con fattori.

Funzioni I/O

Lettura e scrittura di dati da file

- 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 (noi vedremo le principali)

- Per maggiori dettagli si consulti il manuale *R Data Import/Export disponibile on-line*

Caricare e salvare oggetti in formato binario

- Caricare e salvare oggetti arbitrari in formato binario:

– Salvare oggetti in formato binario:

```
> x <- runif(20);
> y <- list(a = 1, b = TRUE, c = "oops");
> save(x, y, file = "xy.rda");
```

– Caricare oggetti in formato binario

```
> load("xy.Rdata");
> ls()
[1] "x" "y"
```

- Caricare e salvare oggetti relativi ad un’intera sessione di lavoro:

```
> save.image();
> load(".RData");
```

Scrittura su file di data frame/matrice

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

Sintassi:

`write.table (`*`x, file="data.txt")`*

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 (si veda l'help…)

Esempio:

```
> m <- matrix(runif(1000),ncol=10)
> colnames(m) <- paste(rep("col",ncol(m)),1:ncol(m),sep="")
> rownames(m) <-paste(rep("row",nrow(m)),1:nrow(m),sep="")
> write.table(m,file="matrix.txt")
```

Lettura di data frame/matrice da file

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

Esempio:

```
> read.table("matrix.txt")
```

read.table dispone di molti altri parametri che si possono settare per esigenze particolari (vedi help)

2. read.delim is almost the same as read.table, except the field separator is *tab* by default…

```
> write.table(m,file="matrix2.txt",sep="\t")
> read.delim("matrix2.txt")

> write.table(m,file="matrix3.txt",sep=";")
> read.delim("matrix3.txt",sep=";")
```

Da notare il campo separatore!!

Importare ed Esportare File in Excel

A. Usare write.table e read.table e le “funzioni di conversione” di Excel

```
> write.table(iris,"iris.txt",sep="\t")
```

Aprire “iris.txt” con Excel ed utilizzare le conversioni formato ....

Per aprire il file da R:

```
> read.table("iris.txt",sep="\t")
```

B. Leggere e scrivere direttamente file in Excel

```
> data(iris)
> write.csv2(iris,"iris.csv")
> read.csv2("iris.csv")
```

csv: Comma-Separated Values

Note sul data set *Iris*:

1. data set built-in in R;
2. `data()`: per listare i data set

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

```
> a <- c(20, 26, 25, 26, 28, 24, 30, 27, 25, 23, 28, 22, 20, 25, 25, 21,
22, 24, 22, 29, 29, 23, 27, 25, 21, 24, 24, 24, 27, 19, 24)
> write(a,"voti.txt")
>  marks <- scan("voti.txt")
Read 31 items
> marks
 [1] 20 26 25 26 28 24 30 27 25 23 28 22 20 25 25 21 22 24 22 29 29 23
27 25 21 24 24 24 27 19 24
```

Le funzioni di I/O si possono usare anche per il download di file dalla rete

```
read.table("nome_percorso_file", sep=";", header=F)
```