

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

1

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)
```

2

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>)

3

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

4

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

5

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
```

```
> sum2
```

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

Chiamata della funzione `sum2` :

```
> sum2(1,2)
```

```
[1] 3
```

6

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

7

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
```

8

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

9

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
```

10

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`.

11

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
```

12

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
```

13

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:

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
```

14

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.

15

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:

<pre>> 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</pre>	<pre>> 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</pre>
---	--

16

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

17

Parametri formali

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

parametri formali

```
> fun1 <- function (x, y, z, w) {}
```

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

parametri attuali

```
> fun1 (1, 2, 3, 4)
```

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

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

18

Variabili locali

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

Si consideri la seguente funzione:

```
f <- function (x) {  
  y <- 2*x  
  z <- y^2  
  z }  
}
```

variabile locale → y <- 2*x
→ z <- y^2
parametro formale ← x
variabile libera ← z

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

19

Variabili libere

Le variabili che non sono né parametri formali e né 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:

Si consideri la seguente funzione:

```
f <- function (x) {  
  y <- 2*x  
  z <- y^2  
  z <- z + w  
  z }  
}
```

variabile locale → y <- 2*x
→ z <- y^2
parametro formale ← x
variabile libera ← z

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

20

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
```

21

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) ... }
```

22

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 sottosottoproblemi (implementati tramite le funzioni S1, S2)
- Il processo di scomposizione 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)

23

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:
 1. 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.
 2. Trasformi la matrice in un data frame con variabili *var1,var2,...,var5*.
 3. Memorizzi il data frame in un file
 4. Per ogni variabile calcoli media, deviazione standard.
 5. 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).

24