Docente: **Matteo Re**

UNIVERSITÀ DEGLI STUDI DI MILANO

Insegnamento: Informatica e statistica

C.d.l. **BIOTECNOLOGIA – curriculum farmaceutico**

Programmi e Funzioni in R

Matteo Re

mail: matteo.*re@unimi.it*

http://homes.di.unimi.it/re

DI - Dipartimento di Informatica

Università degli Studi di Milano

Programmi in R

Strutture dati + Algoritmi = Programmi

vettori
fattori
array
liste
…

Linguaggio R

- Strutture + Funzioni
- Classi + metodi

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 (ad esempio *Notepad*++, scaricabile gratuitamente dalla rete)

Esempio di script R

```
# Functional classification of yeast genes using gene expression data
library(yeastCC);
library(e1071);
data(spYCCES);
source("yeastGO.R");

# data preparation
Yeast.specific.TAS <- Get.yeast.GO.specific.classes(evidence="TAS");
Yeast.general.TAS <- Get.yeast.GO.all.classes(Yeast.specific.TAS);
load("Yeast.general.classes.TAS.object");
l <- Count.examples.per.class(Yeast.general.classes.TAS);
cl1.genes <- Extract.class(Yeast.general.classes.TAS,"GO:0000902"); cl2.genes <- Extract.class(Yeast.general.classes.TAS,"GO:0006092");
paste("GO:0000902", ":", get.Term.Definition("GO:0000902")); paste("GO:0006092", ":", get.Term.Definition("GO:0006092"));
exprs.cl1 <- exprs(spYCCES)[as.character(cl1.genes$gene.names),]; exprs.cl2 <- exprs(spYCCES)[as.character(cl2.genes$gene.names),];
exprs.cl1[is.na(exprs.cl1)] <- 0; exprs.cl2[is.na(exprs.cl2)] <- 0;

# classification of yeast genes
n1.test<-round(nrow(exprs.cl1)/3); n1.train<-nrow(exprs.cl1)-n1.test;
n2.test<-round(nrow(exprs.cl2)/3); n2.train<-nrow(exprs.cl2)-n2.test;
Xtrain<-rbind(exprs.cl1[1:n1.train,],exprs.cl2[1:n2.train,]);
Xtest<-rbind(exprs.cl1[(n1.train+1):nrow(exprs.cl1),],exprs.cl2[(n2.train+1):nrow(exprs.cl2),]);
ytrain<-as.factor(c(rep(1,n1.train),rep(2,n2.train)));
ytest<-as.factor(c(rep(1,n1.test),rep(2,n2.test)));
model <- svm(as.matrix(Xtrain),ytrain,type="C-classification", kernel="linear", cost=1, gamma=1, degree=2, coef0=1);
prediction <- predict(model,Xtest);
conf.matrix <- table(prediction,ytest);
sensitivity <- conf.matrix[1,1]/(conf.matrix[1,1]+conf.matrix[2,1]);
specificity <- conf.matrix[2,2]/(conf.matrix[2,2]+conf.matrix[1,2]);
accuracy <- (conf.matrix[1,1] + conf.matrix[2,2])/length(ytest);
```

Funzioni

- Abbiamo già visto molti esempi di funzioni disponibili in R
- Le funzioni in R possono anche definirle gli utenti
- I programmi in R possono essere realizzati tramite funzioni

Funzioni: sintassi

La sintassi per scrivere una funzione è:

`function (argomenti)  corpo_della_funzione`

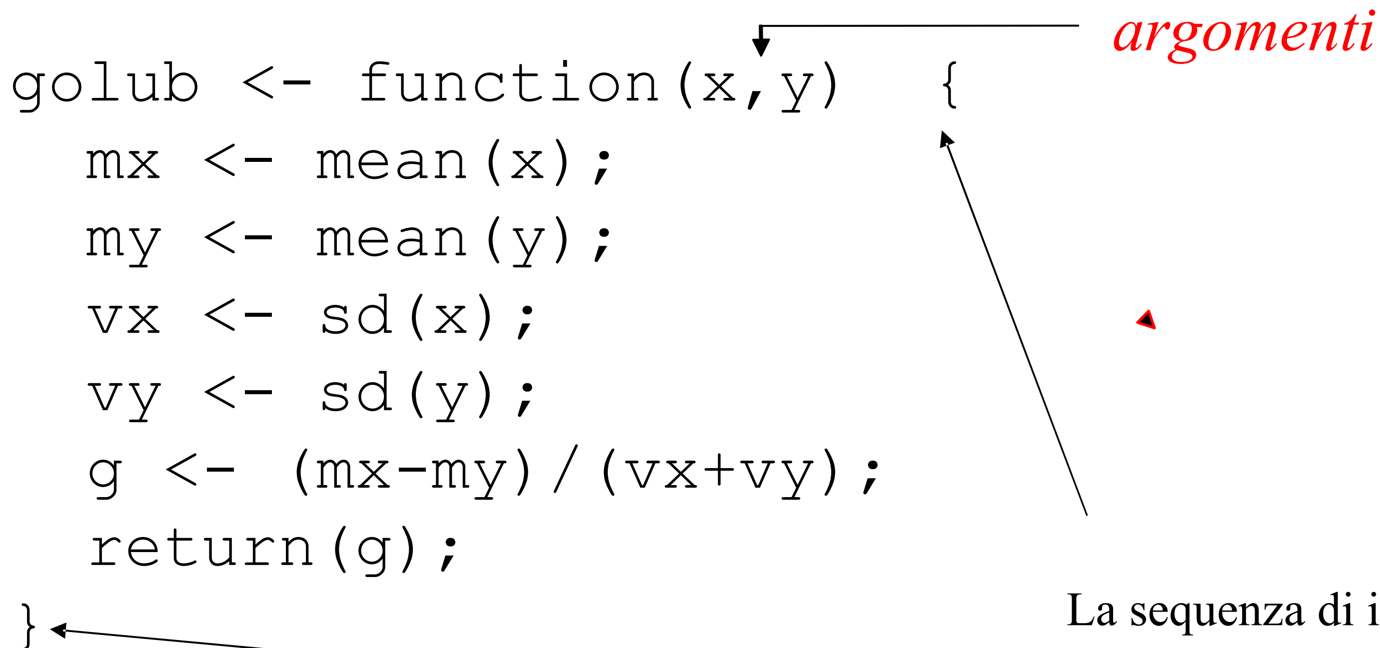
- `function` è una **parola chiave** di R
- `Argomenti` è una lista eventualmente vuota di *argomenti formali* separati da virgole:
  `(arg1, arg2, ..., argN)`

- 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




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

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

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 nome diverso)
- Caricamento in memoria della funzione. Due possibilità:

1. `< source("golub.R")`
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
< 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.

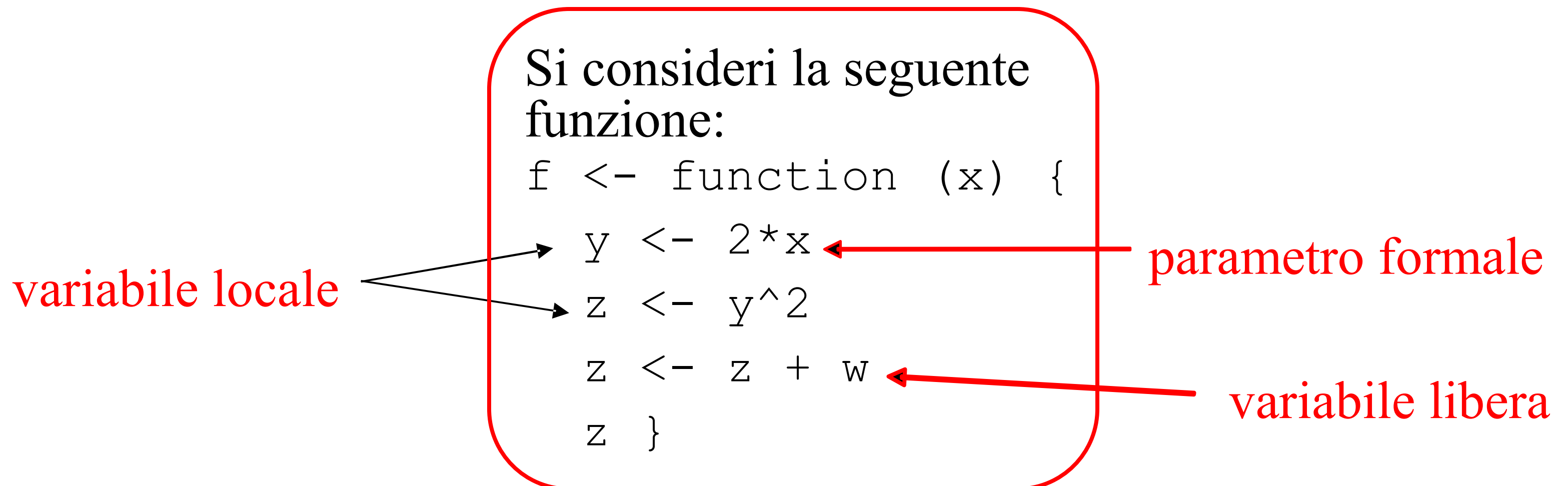
Esempio:

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(
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

```
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 () {... }

S3 <- function () {... }


# funzioni di primo livello" chiamate dalla funzione
# principale

P1 <- function (x) { S1(x); S3(); }

P2 <- function (x) { S2(); S1(x); ... }


# funzione principale del programma R

MainProgram <- function(x,y,z) {P1(x); P2(y); P1(z) ... }
```

Main program

P1

S1

S3

P2

S2

S1

• • •

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, S3)
- 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 (I)

1. Scrivere una funzione *compute.mean.var* che, avendo come argomento una lista di vettori numerici, calcoli la media e la varianza per ciascun elemento della lista.
2. Scrivere una funzione find.stop.codon che, ricevuto come argomento un vettore di "triplette" del codice genetico ritorni un messaggio "codon di stop trovato" o "codon di stop non trovato" a seconda che una delle triplette "UAA", "UAG" o "UGA" sia presente o meno nel vettore di ingresso.
3. Scrivere una funzione find.codon che, ricevuto in ingresso un vettore di "triplette" del codice genetico ed una tripletta codon arbitraria, stampi sullo schermo un messaggio di codon trovato e la sua posizione, o un messaggio di codon non trovato.
   In caso pero' incontri prima un codon di stop deve stampare un messaggio di stop codon trovato, la sua posizione e terminare.

Esercizi (II)

4. Scrivere una funzione *analyze_string* che ricevuto in ingresso una stringa arbitraria calcoli la frequenza dei simboli componenti la stringa stessa
5. Scrivere una funzione che calcoli i numeri di Fibonacci
6. 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
7. 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).