R - prima lezione

1 Alcuni fatti elementari

- per uscire da R chiudendo la sessione di lavoro:
  `q()`
- per valutare un'espressione: invio
- per inserire un commento:
  #questo e' un commento e non viene valutato
- aritmetica banale:

```
> 1+2*3
[1] 7
> 2**3
[1] 8
> 2^3
[1] 8
> sqrt(2)
[1] 1.414214
> sin(3.14159)    #  sin(Pi greco) e' zero
[1] 2.65359e-06   #  quasi zero
```

- In R esiste anche la costante Pi greco:

```
> sin(pi)        #  piu' vicino a  zero:
[1] 1.224606e-16
```

- Alcune funzioni:

  `sqrt(x)` radice quadrata

`abs(x)` valore assoluto

`sin(x) cos(x) tan(x)` funzioni trigonometriche

`asin(x) acos(x) atan(x)`funzioni trigonometriche inverse

`exp(x) log(x)` esponenziale e logaritmo naturale

`gamma(x)` funzione Gamma

`beta(a,b)` funzione beta Beta(a,b) = (Gamma(a)Gamma(b))/(Gamma(a+b)).

`choose(n,k)` coefficiente binomiale n su k

...

Oss: le funzioni possono essere annidate:

```
> log(exp(2*pi))
[1] 6.283185
```

2 Costanti

2.1 Assegnazione di valori numerici

Un valore viene assegnato ad un oggetto con nome a nostra scelta tramite l'operatore di assegnazione `<-` oppure `_`.
Si noti che l'uso dell'operatore `=` è sconsigliato per le assegnazioni in quanto non sempre valido; a partire dalla versione 6 di R, al primo utilizzo dell'operatore `_` si riceve un messaggio di avvertimento (unico per tutto il corso della sessione), che ne sconsiglia l'uso in quanto, sebbene più rapido, genera codici che possono essere non facilmente leggibili.

`x<- -2*sqrt(3)`

assegna all'oggetto x il valore `-2*sqrt(3)`

NB: il risultato non viene visualizzato; per mostrare il valore numerico di x:

```
> x
[1] -3.464102
```

x viene trattato come un valore numerico, sul quale possiamo effettuare altre operazioni:

```
> x^2
[1] 12
```

oppure, volendo assegnare questo valore a un altro oggetto:

```
> y<- x^2
> y
[1] 12
```

Tutti gli oggetti vengono memorizzati e conservati per tutto il corso della sessione di lavoro; per visualizzare l'elenco degli oggetti che abbiamo memorizzato, e quanto eventualmente disponibile nella directory di lavoro:

```
> ls()
[1] "x" "y"
```

Per eliminare un oggetto, ad esempio x:

```
> rm(x)
> x
Error: Object "x" not found
```

2.2 Assegnazione di valori logici

R permette di eseguire operazioni logiche e di definire variabili logiche:

```
> y
[1] 12
> y<10
[1] FALSE
> #   l'espressione di sopra ha il valore logico di falso (F)
> y!=10     # != indica "diverso da"
[1] TRUE
```

costruisce una variabile logica e la assegna a x:

```
> x<- y!=10
> x
[1] TRUE
```

un altro tipo di operazione logica:

```
> y
[1] 12
> ifelse(y>10,'vero','falso') #restituisce un valore alfanumerico
[1] "vero"
> ifelse(y>10,1,0) #restituisce un valore numerico
[1] 1
```

R contempla anche la funzione logica `if`. Riferendoci all'esempio precedente possiamo scrivere

```
if(y>10) {print("vero")} else {print("falso")}
```

Si notino il differente uso delle parentesi e la presenza delle azioni relative alle due condizioni alternative, ciascuna delle quali nell'esempio è racchiusa tra parentesi graffe. Queste ultime possono anche essere omesse, ma sono sempre necessarie quando si utilizzino *gruppi* di azioni condizionali.

Come in altri linguaggi di programmazione, non è necessario che il comando `if` sia seguito dal comando `else`, che può essere omesso. Ad esempio nell'ambito di un programma più ampio si potrebbe inserire un `if` per eseguire un comando "condizionatamente", ossia solo se una data condizione risulta verificata. Nel caso dell'esempio precedente, possiamo voler escludere che in un insieme di calcoli successivi vengano utilizzati input non validi. Possiamo ad esempio controllare preventivamente che y sia un numero strettamente positivo: scriveremo quindi

```
if(!y>0) print("errore: numero inserito <=0 ! Controllare l'input")
#oppure
if(y<=0) print("errore: numero inserito <=0 ! Controllare l'input")
```

ad esempio volendo calcolare all'interno di una funzione la radice quadrata di un numero utilizziamo il costrutto if ... else :
`if(!y>0) {print("errore: valore<=0!")} else {y <- sqrt(y)}`.
Per verificare l'effettivo funzionamento del controllo, si assegni a y un valore non valido (ad es. -2) e successivamente uno valido (ad es. 4):

```
> y <- -2
> if(!y>0) {print("errore: valore<=0!")} else {y<- sqrt(y)}
[1] "errore: valore<=0!"
> y <- 4
> if(!y>0) {print("errore: valore<=0!")} else {y<- sqrt(y)}
> y
[1] 2
```

Si può verificare facilmente che il valore di `y` memorizzato non provoca incongruenze. Modificando il valore di `y` otteniamo invece il messaggio di errore:

```
> y<- -2
> if(y<=0) print("errore: numero inserito <=0 ! Controllare l'input")
[1] "errore: numero inserito <=0 ! Controllare l'input"
```

R lavora con costanti come visto finora, ma gestisce anche vettori, matrici ed altri tipi di oggetti (data frame e liste) che vedremo piu' avanti.

3 Vettori e matrici

3.1 Creazione di vettori

Particolari sequenze:

```
> -2:3
[1] -2 -1  0  1  2  3
# a:b genera una sequenza-vettore di valori
tra a e b di passo 1
> -2.3:1
[1] -2.3 -1.3 -0.3  0.7
> 10:1
 [1] 10  9  8  7  6  5  4  3  2  1
```

Per utilizzare un passo differente:

```
> seq(-1,1,0.5)
[1] -1.0 -0.5  0.0  0.5  1.0
```

o equivalentemente:

```
> seq(-1,1,by=0.5)
[1] -1.0 -0.5  0.0  0.5  1.0
```

Possiamo fissare il numero di valori in output omettendo il passo:

```
> seq(-1,1,length=21)
[1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4
[16]  0.5  0.6  0.7  0.8  0.9  1.0
```

Possiamo generare replicazioni dello stesso valore x k volte: `rep(x,k)`

```
> rep(2.1,3)
[1] 2.1 2.1 2.1
```

Per memorizzare dati a nostra scelta: funzione `c()`:

```
> x <- c(1,2,3,5,8,13)
> x
[1]  1  2  3  5  8 13
```

Oss: La funzione `c()` serve per assegnare ad un insieme di valori la modalita' "vettore".

Oss: Possiamo combinare i comandi visti finora:

```
> x<- c(rep(0.1,2),4,seq(-1,2,0.5),4:2)
> x
 [1]  0.1  0.1  4.0 -1.0 -0.5  0.0  0.5  1.0  1.5  2.0  4.0  3.0  2.0
```

NB: i vettori sono dotati di una dimensione (la lunghezza); per visualizzarne il numero di componenti:

```
> length(x)
[1] 13
```

Se il vettore e' lungo, per scriverne i valori puo' essere piu' conveniente la funzione scan():

```
> x<- scan()
#una volta dato l'invio, R e' pronto a ricevere e memorizzare i dati:
1: 1
2: 2
3: 3
4: 5
5: 8
6: 13
7:
Read 6 items
> x
[1]  1  2  3  5  8 13
```

NB: La funzione scan() serve anche per leggere un vettore da miofile.dat, localizzato in miadir:

```
> miovec <- scan("c:\miadir\miofile.dat")
```

NB: specificare il percorso corretto per poter accedere al file.

```
> seq(-1,1,length=21)
[1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2 -0.1  0.0  0.1  0.2  0.3  0.4
[16]  0.5  0.6  0.7  0.8  0.9  1.0
```

3.1.1 Operazioni sui vettori:

R applica ai vettori l'aritmetica degli scalari, operando *componente per componente*:

```
> x<- 0:5
> y<- 1:6
> x
[1] 0 1 2 3 4 5
> x+1
[1] 1 2 3 4 5 6
> y
[1] 1 2 3 4 5 6
> x+y
[1]  1  3  5  7  9 11
> (x+y)^2
[1]   1   9  25  49  81 121
> x*y
[1]  0  2  6 12 20 30
```

Quindi R lavora per componenti.
Possono essere eseguite operazioni logiche anche sui vettori

```
> x<3.2
[1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

Per effettuare il prodotto scalare x'y:

```
> t(x)%*%y #t() indica l'operatore di trasposizione
     [,1]
[1,]   70
```

ma anche, trattandosi di vettori

```
> x%*%y #quando opera con vettori, R li considera riga
o colonna in modo da ottenere un prodotto valido.
     [,1]
[1,]   70
```

3.2 Assegnazione di matrici

Per creare una matrice a partire da due (o piu') vettori: `cbind(x,y)`

```
> x
[1] 0 1 2 3 4 5
> y
[1] 1 2 3 4 5 6
> xx<- cbind(x,2*y)
> xx
     x
[1,] 0  2 [2,] 1  4 [3,] 2  6 [4,] 3  8 [5,] 4 10 [6,] 5 12
```

Oss: `rbind(x,y)` opera in modo analogo, ma agisce sulle righe.

Per creare una matrice, esiste anche il comando

`matrix(vettore, nrow=numrighe, ncol=numcolonne, byrow=F)`.

Tale comando scrive gli elementi di `vettore` per colonna su `numrighe` e/o `numcolonne` specificate.
Dato il numero di elementi della matrice, e' sufficiente indicare un solo altro argomento, il numero di righe o il numero di colonne.
Per default, l'opzione `byrow` e' impostata su `False` ; se si vuole che la creazione della matrice avvenga per riga anziche' per colonna, bisogna indicare l'opzione `byrow=T`.

Esempio:

```
> x1<- matrix(1:9,ncol=3)
> x1
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> x1<- matrix(1:9,ncol=3,byrow=T)
> x1
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
```

3.2.1 Operazioni tra matrici:

Prodotto tra la matrice xx trasposta e y

```
> xx
     [,1] [,2]
[1,]    0    2
[2,]    1    4
[3,]    2    6
[4,]    3    8
[5,]    4   10
[6,]    5   12
> dim(xx) #mi indica le dimensioni della matrice xx
[1] 6 2
> t(xx) #trasposta di xx
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    0    1    2    3    4    5
[2,]    2    4    6    8   10   12
> dim(t(xx))
[1] 2 6
> t(xx)%*%y #prodotto matriciale
     [,1]
[1,]   70
[2,]  182
```

Per l'inversa di una matrice: `solve(x)`; `det(x)`indica il determinante di x.

Esempio: Creo due matrici x1, x2 e ne inverto il prodotto x1*x2.

```
> x1<- rbind(c(0,2,4),c(1,3,5))
> x1
     [,1] [,2] [,3]
[1,]    0    2    4
[2,]    1    3    5
> x2<- matrix(seq(0,10,2),nrow=3)
> x2
     [,1] [,2]
[1,]    0    6
[2,]    2    8
[3,]    4   10
> det(x1%*%x2)
[1] 144
```

```
> solve(x1%*%x2)
           [,1]       [,2]
[1,]  0.5555556 -0.3888889
[2,] -0.1805556  0.1388889
```

`diag(x)` crea una matrice diagonale a partire da un vettore:

```
> x<- 1:3
> x
[1] 1 2 3
> diag(x)
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    2    0
[3,]    0    0    3
```

4 Estrarre elementi da vettori o matrici

Si usano le parentesi *quadrate*. Notare che le parentesi *tonde* si utilizzano per indicare gli *argomenti delle funzioni*.

Esempio: Elimino la terza componente di x:

```
> x
[1] 0 1 2 3 4 5
> x[3]
[1] 2
```

l'elemento (1,2) di xx:

```
> xx
     [,1] [,2]
[1,]    0    2
[2,]    1    4
[3,]    2    6
[4,]    3    8
[5,]    4   10
[6,]    5   12
> xx[1,2]
[1] 2
```

la prima colonna di xx:

```
> xx[,1]
[1] 0 1 2 3 4 5
```

la prima riga di xx:

```
> xx[1,]
[1] 0 2
```

Si possono estrarre anche sottoinsiemi di elementi, utilizzando la funzione `c()` per elencarli:

```
> xx
     [,1] [,2]
[1,]    0    2
[2,]    1    4
[3,]    2    6
[4,]    3    8
[5,]    4   10
[6,]    5   12
> xx[c(2,3,5),1]
[1] 1 2 4
> xx[1:3,2]
[1] 2 4 6
> x
[1] 0 1 2 3 4 5
> x[c(1:2,4:6)]
[1] 0 1 3 4 5
> x[c(1:2,6,6,2:6)]
[1] 0 1 5 5 1 2 3 4 5
```

eccetera
Di un vettore o di una matrice si possono selezionare i soli elementi che soddisfano determinate condizioni:

```
> xx
     [,1] [,2]
[1,]    0    2
[2,]    1    4
[3,]    2    6
[4,]    3    8
[5,]    4   10
```

```
[6,]    5   12
> xx[xx>=5] #gli elementi di xx superiori o = a 5
[1]  5  6  8 10 12
> xx[xx[,1]>=4] #gli elementi di xx la cui prima coordinata e' >=4
[1]  4  5 10 12
```

NB: notare la differenza nell'output che si ottiene:

```
> xx[xx[,1]>=4,]
     x
[1,] 4 10
[2,] 5 12
```

Ovviamente, sottoinsiemi di elementi possono essere salvati in nuovi vettori.
Possiamo *eliminare* elementi, indicare tra le parentesi quadre un numero *negativo*: l'elemento corrispondente viene omesso dal vettore risultante:

```
> x
[1] 0 1 2 3 4 5
> x[-1] #elimina la prima componente del vettore
[1] 1 2 3 4 5
> xx
     x
[1,] 0  2
[2,] 1  4
[3,] 2  6
[4,] 3  8
[5,] 4 10
[6,] 5 12
> xx[-1,] #elimina la prima RIGA
     x
[1,] 1  4
[2,] 2  6
[3,] 3  8
[4,] 4 10
[5,] 5 12
```

creiamo una nuova matrice a tre colonne

```
> z<- cbind(xx, c(7,4,9,1,12,3))
> z
```

```
     x
[1,] 0  2  7
[2,] 1  4  4
[3,] 2  6  9
[4,] 3  8  1
[5,] 4 10 12
[6,] 5 12  3
> z[,-2] #elimina la seconda COLONNA (restituisce ancora una matrice)
     x
[1,] 0  7
[2,] 1  4
[3,] 2  9
[4,] 3  1
[5,] 4 12
[6,] 5  3
```

NB: gli elementi delle matrici assumono un numero d'ordine progressivo assegnato prima per riga e poi per colonna.

Se vogliamo eliminare il valore 10, essendo l'elemento numero 11 di z,

```
> z[11]
[1] 10
> z[-11]
 [1]  0  1  2  3  4  5  2  4  6  8 12  7  4  9  1 12  3
```

Possiamo eliminare gli elementi che soddisfano una determinata condizione:

```
> indici<- which(z>=8)
# which(condizione) identifica gli indici degli elementi che soddisfano condizione
> indici
[1] 10 11 12 15 17
> z[-indici] #elimina da z gli elementi >=8.
 [1] 0 1 2 3 4 5 2 4 6 7 4 1 3
```

Possiamo *modificare* elementi che soddisfano certe condizioni:

Esempio:

```
> x<- seq(-2,3,pi/4)
> x
[1] -2.0000000 -1.2146018 -0.4292037  0.3561945  1.1415927
1.9269908  2.7123890
```

selezioniamo gli elementi tra -1 e 0

```
> x[x<0&x>(-1)]
[1] -0.4292037
```

ora vogliamo assegnare a questi elementi il loro valore assoluto:

```
> a<- x[x<0&x>(-1)]
> x[x<0&x>(-1)]<- abs(a)
> x
[1] -2.0000000 -1.2146018  0.4292037  0.3561945  1.1415927  1.9269908
[7]  2.7123890
```

analogamente potevamo operare con `which`:

```
> a<- which(x<0&x>(-1))
> a
[1] 3
> x[a]<- abs(x[a])
> x
[1] -2.0000000 -1.2146018  0.4292037  0.3561945  1.1415927  1.9269908
[7]  2.7123890
```

il risultato e' lo stesso.

4.1 Alcune funzioni utili per la manipolazione di vettori e matrici

```
> length(x)         #...il numero di elementi
[1] 24
> max(x)            #...il massimo
[1] 26
> min(x)            #...il minimo
[1] 3
> sum(x)            #...la somma dei valori in x
[1] 348
> prod(x)           #...il prodotto dei valori in x
[1] 2.016457e+26
```

Alcune operazioni
Prendiamo un vettore di 100 numeri estratti da una distr. Un(0,4)*: `x<-`runif(100)*4 ; ovviamente avremo

*Per l'estrazione di numeri casuali si rimanda alla seconda lezione

```
> length(x)
[1] 100         --------->il numero di elementi
> min(x)
[1] 0.04556106  --------->il minimo
> max(x)
[1] 3.998025    --------->il massimo
> sum(x)
[1] 204.6429    --------->la somma degli elementi
```

NB: per le matrici:

```
> z
     x
[1,] 0  2  7
[2,] 1  4  4
[3,] 2  6  9
[4,] 3  8  1
[5,] 4 10 12
[6,] 5 12  3
> max(z)        #--------->massimo "globale"
[1] 12
> max(z[,1])    #--------->massimo sulla prima colonna
[1] 5
> max(z[,2])    #--------->massimo sulla seconda colonna
[1] 12
> sum(z)        #--------->somma "globale"
[1] 93
> sum(z[,1])    #--------->somma degli elementi della prima colonna
[1] 15
> prod(z)       #--------->prodotto "globale"
[1] 0
> prod(z[2,])   #--------->prodotto degli elementi della seconda riga
[1] 16
```

...eccetera
Inoltre per il nostro vettore di realizzazioni da Un(0,4)

```
> mean(x)           #--------->media aritmetica
[1] 2.046429
> sum(x)/length(x)  #--------->sempre la media aritmetica
[1] 2.046429
```

```
> var(x)            #--------->varianza corretta
[1] 1.201118
```

la varianza campionaria:

```
> sum(x^2)/length(x)-mean(x)^2
[1] 1.189106
```

oppure

```
> var(x)*(length(x)-1)/length(x)
[1] 1.189106
```

5 Creazione di funzioni personalizzate

In R e' anche possibile creare una *funzione* che risponde a nostre specifiche esigenze.
Ad esempio, possiamo voler costruire una funzione che calcoli la media "k-trimmed", o "$\alpha-$trimmed, con $k/n = \alpha$ e $k \in \{1, ..., n\}$, cioe' la media aritmetica delle osservazioni "centrali", escludendo le k piu' alte e le k piu' basse, in modo da eliminare l'effetto di valori anomali.
Oltre a calcolare la media "trimmed" sul vettore che ci interessa, in R possiamo costruire una funzione che *per ogni vettore e per ogni scelta di k* ci restituisca la media trimmed di ordine k.

Una funzione e' un oggetto caratterizzato da parametri (due nel nostro caso) e una volta caricata in R rimarra' a disposizione per tutta la sessione di lavoro o, laddove salvata in memoria, fino a quando non la elimineremo (comando `rm()`).

Per costruire la nostra funzione, operiamo nel modo seguente: ricordiamo in primo luogo che la definizione di media k-trimmed e'

$$\bar{x}_k = \frac{1}{n-2k}(X_{(k+1)} + \cdots + X_{(n-k)}) = \frac{1}{n-2k}\sum_{i=k+1}^{n-k} X_{(i)},$$

dove $X <_{(i)}$ indica l'i-ma statistica d'ordine.
Per ottenere le statistiche d'ordine e' sufficiente ordinare il nostro campione. Supponiamo che il nostro campione sia estratto da una distribuzione Student(19)[†]:

```
> x<- rt(100,19)
```

Il nostro vettore puo' essere ordinato con il comando `sort()` :

[†]Per la generazione di numeri casuali, si rimanda alla lezione successiva

```
> xsort<- sort(x)
```

Sul blocco note scriviamo la funzione che chiamiamo "trimean":

```
 trimean<- function(kappa,vec)
# in parentesi tonde mettiamo i parametri della nostra funzione
 {
vec<- sort(vec)
enne<- length(vec)
kappa1<- enne-kappa+1
vec<- vec[-c((1:kappa),(kappa1:enne))]
res<- mean(vec)
return(res)
}
```

Copiamo questa funzione in R (comandi copia e incolla). R immagazzina l'informazione che gli abbiamo passato. In presenza di errori di sintassi...reagisce.
Infine, applichiamo la nostra funzione al vettore x generato precedentemente, scegliendo k=5:

```
> trimean(5,x)
[1] -0.02810358
> mean(x)
[1] -0.04269589
```

6 Loop

Anche se si tratta di una procedura non completamente efficiente, in R esiste la possibiltà di eseguire loop. Si vedrà in seguito che è preferibile utilizzare comandi alternativi, quali `apply` e `sapply`, che consentono di evitare il costrutto `for` e di ridurre i tempi di esecuzione. Per calcoli non eccessivamente elaborati e numero di iterazioni non molto elevato si può comunque fare ricorso al comando `for`. Come argomenti del comando `for`, la sintassi prevede di indicare l'indice (`i`) e il suo campo di variazione (es.: `1:100`), identificato dal sottocomando `in`. Volendo calcolare la somma dei primi 10 numeri naturali scriviamo ad esempio:

```
a<- 0
x<- 1:10
for(i in x) a<- a+i
a
```

oppure

```
a<- 0
x<- 1:10
for(i in x){
    a<- a+i
    print(a)
}
```

Si noti che è equivalente scrivere `for (i in 1:10)`, oppure assegnare ad x la sequenza $(1, \ldots, 10)$, `x<- 1:10` e successivamente scrivere `for(i in x)`.