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 <- 1:10;  
  y <- rep(2, 10);  
  z <- seq(-1, 1, length=10);  
  w <- x + y + z  
}
```

- 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 costituita 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: Cosa accade se viene valutata un varabile non di “mode” logical?

```
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
```

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 .
- In genere v compare nelle istruzioni contenute nel blocco, ma non necessariamente

Istruzione for: esempi

```
> v = sample(1:100, 10);
```

```
> v
```

```
[1] 45 66 46 29 60 44 15 61 59 17
```

```
> for (i in 1:5) cat(v[i], " ")
```

```
45 66 46 29 60
```

```
> for( j in c(3,1,4,1,5,9,2,7)) cat(v[j], " ")
```

```
46 45 29 45 60 59 66 15
```

- 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  $\pi$ 
```

```
> 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

```
> a <- sample(1:20, 20) # permutazione
degli interi da 1 a 20
> a
[1]  7  8  1  3 18  2 14  6 10 12 20
5 19 13  4 17 16 15  9 11
> i<-1;
> while (a[i] < 10) i <- i+1;
> i
[1] 5
> a[i]
[1] 18
```

```
> i <- 0
> y<-10
> while (y>1) {
      y<-y/2;
      i<-i+1
    }
> i
[1] 3
```

- **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

Esempio:

```
> repeat{ # estrazione casuale di un numero pari <= 100
  b<- sample(1:100, 1)
  if(b%% 2 == 0) break; }
> b
[1] 58
```

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:

```
> x<-sample(1:10^6, 10000)/10^5;y<-sample(1:10^6, 10000)/10^5
```

A. Calcolo con cicli for:

```
> z <- 0;
> system.time(for (i in 1:10000) z <- z + x[i]*y[i])
user system elapsed
0.008 0.000 0.007
```

B. Calcolo con funzioni vettorizzate:

```
> system.time(x%*%y) (oppure sum(x*y))
user system elapsed
0 0 0
```

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)** o **a%*%b** piuttosto di una serie di cicli for. Vedi anche **rowSums** e **colSums** per somma delle righe(colonne) di una matrice
- 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** , ... le vedremo
 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;

I comandi “ciclici” della famiglia apply

- **Esempio.**

```
> y<-sample(1:10^3, 10000, replace=TRUE)
```

```
> system.time(b<-lapply(y, function(el){if(el%%2==0)  
                                return(TRUE)else return(FALSE)} ) )
```

```
user system elapsed  
0.012  0.000  0.011
```

- Con il ciclo **for**

```
> res<-rep(0, length(y))
```

```
> system.time(for(i in 1:length(y))  
              {if(y[i]%%2==0) res[i]<-TRUE else res[i] <-FALSE})
```

```
user system elapsed  
0.016  0.000  0.018
```

Funzioni e Script R

- In R ci sono due modi per modulare l'esecuzione di codice:
 - **Funzioni**
 - **Script**
- Permettono di riutilizzare del codice precedentemente scritto a cui abbiamo “attribuito un nome”, oppure di eseguire una sequenza di comandi R “automatizzata”
- Molto utili per riutilizzare codice scritto precedentemente
- Abbiamo già visto molti esempi di funzioni disponibili in R
- Le funzioni in R possono anche definite dagli utenti

Funzioni

- Funzioni: sintassi

function (argomenti) corpo_della_funzione

- **function** è una parola chiave di R

- **argomenti** è una sequenza 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 determinare se un numero n è pari

```
pari <- function (n) {  
  return(n%%2==0)  
}
```

```
> pari(3)  
[1] FALSE  
> pari(2)  
[1] TRUE
```

- Una funzione può essere scritta direttamente da linea di comando, oppure scritta in un file di testo (con un qualsiasi editor di testo) e caricata con il comando **source**

- Supponiamo di scrivere il codice a sinistra in un file *prova_libreria.R*

```
> rm(list=ls())  
> source("prova_libreria.R")  
> pari(50)  
[1] TRUE
```

Argomenti formali e attuali

- Nella funzione pari n è detto parametro formale, serve cioè a descrivere il comportamento della funzione, ma non ha un valore “attuale”
- n viene sostituito dall'argomento attuale quando la funzione è chiamata:
 - > d <- 23;
 - > res <- pari(d)
- d è l'argomento *attuale*, cioè quello il cui valore è usato al posto di n nella definizione della funzione

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, y) { x <- x + y^2 }  
> a <- 2; b <- 3  
> fun1(a, b)  
> a  
[1] 2
```
- 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 a 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
```

Parametri formali, variabili 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
- Si sconsiglia comunque l'uso di variabili libere

Si consideri la seguente funzione:

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

```
> 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;  
}
```

- 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
```

Script R

- Uno *script* R è una sequenza di comandi R contenuti in file di testo
- Uno script è eseguibile mediante il comando *source*
- Gli script possono contenere ogni comando R, quindi anche la definizione di funzioni utente

Scriviamo i seguenti comandi nel file di testo primoScript.R

```
rm(list=ls())
pari <- function (n) {
  return(n%%2==0)
}
A <- matrix(sample(1:50, 25), nrow =5)
num_pari <- sum(apply(A, c(1,2), FUN=pari))
```

Quindi digitiamo dalla shell R:

```
>source("primoScript.R")
> ls()
[1]"A" "num_pari" "pari"
> num_pari
[1] 14
```

Esercizi

1. Scrivere delle funzioni R, e salvarle nel file *mia_libreria.R*, che svolgano le seguenti operazioni
 - a) Calcolo della media campionaria dei dati contenuti in un vettore, escludendo eventuali valori NA
 - b) Calcolo della varianza campionaria dei dati contenuti in un vettore, escludendo eventuali valori NA
 - c) Calcolo della deviazione standard campionaria dei dati contenuti in un vettore, escludendo eventuali valori NA
 - d) Calcolo della deviazione standard campionaria dei dati contenuti in un vettore, escludendo eventuali valori NA
 - e) Calcolo della mediana campionaria dei dati contenuti in un vettore, escludendo eventuali valori NA
 - f) Calcolo del percentile k -mo dei dati un vettore. k deve essere un parametro della funzione

Esercizi

2. Scrivere delle funzioni R, e salvarle nel file *mia_libreria.R*, che
- g) Calcolino la frequenza assoluta dei dati contenuti in un vettore (non usate *table*). Si usi opportunamente il comando *unique*
 - h) Calcolino la frequenza relative dei dati contenuti in un vettore (non usate *table*)
 - i) Calcolino la moda dei dati contenuti in un vettore (non usate *table*)
 - j) Calcolino l'indice di eterogeneità di Gini dei dati contenuti in un vettore
 - k) Calcolino l'entropia dei dati contenuti in un vettore