R per l'analisi statistica multivariata

Unità B: elementi di programmazione

Tommaso Rigon

Università Milano-Bicocca

Unità B

Argomenti affrontati

- Definizione di una nuova funzione
- Istruzioni di controllo: `if`, `else`
- Cicli `for`, cicli `while`
- Cenni alle funzioni `*apply`
- I pacchetti **R**

- Esercizi **R** associati: `https://tommasorigon.github.io/introR/exe/es_1.html`

Funzioni I

- Oltre ad essere un software statistico, **R** è un **linguaggio di programmazione**.
- La maggior parte degli oggetti **R** è una funzione. Infatti:

```
class(sum) # Identifica la tipologia di oggetto
# [1] "function"

class(log) # Secondo esempio
# [1] "function"
```

- È possibile quindi definire **nuove funzioni**, in aggiunta a quelle già esistenti.
- Una volta create, le funzioni possono essere usate proprio come tutte le altre.

Funzioni II

- Per creare una nuova funzione, si utilizza comando `function`.
- La **nuova funzione** "`cube`" calcola appunto il cubo del numero ricevuto come input:

```
cube <- function(x) {
  out <- x^3
  out
}
```

- L'**ultimo oggetto** (in questo caso chiamato `out`) viene restituito come risultato:

```
cube(4) # Calcola il cubo del valore 4
# [1] 64
```

- Possiamo alternativamente utilizzare anche il comando `return` per restituire il risultato:

```
cube <- function(x) {
  out <- x^3
  return(out) # Esplicita che il valore da dover restituire è out
}
cube(8) # Calcola il cubo del valore 8
# [1] 512
```

Funzioni III

- È possibile definire nuove funzioni con **molteplici argomenti**, scegliendo anche i loro eventuali **valori predefiniti**

```
power <- function(x, p = 2) {
  out <- x^p
  out
}
```

- Il valore predefinito per l'argomento `p` è il quadrato (`p = 2`), infatti:

```
power(x = 4) # Calcola il quadrato del valore 4
# [1] 16

power(4) # Sintassi alternativa: non è necessario specificare i nomi degli argomenti
# [1] 16
```

- Tuttavia, possiamo selezionare una potenza diversa nel modo seguente:

```
power(x = 4, p = 3) # Calcola il cubo del valore 4
# [1] 64

power(4, 3) # Sintassi alternativa: non è necessario specificare i nomi degli argomenti
# [1] 64
```

Istruzioni di controllo I

- L'istruzione di controllo `if` consente di svolgere una determinata operazione solamente se una certa **condizione** è verificata (`TRUE` oppure `FALSE`).
- L'istruzione di controllo (facoltativa) `else` consente di svolgere un'**operazione alternativa** se la precedente condizione non è verificata.
- Ad esempio, il seguente codice mostra a schermo (funzione `print`) una frase diversa a seconda che la condizione sia vera o falsa:

```
condizione <- pi^2 < 10 # Valore booleano (in questo caso la condizione è TRUE)

if(condizione) {
  print("La condizione è vera")
  # Alcuni comandi da eseguire
  # ...
} else {
  print("La condizione è falsa")
  # Altri comandi da eseguire
  # ...
}

# [1] "La condizione è vera"
```

Istruzioni di controllo II

- La seguente funzione `square_root` è un esempio più concreto relativo ad `if` ed `else`.
- Se il numero fornito come input è negativo, viene restituito un `NaN`:

```
square_root <- function(x) {
  if (x < 0) {
    # Messaggio di avvertimento; in realtà sarebbe più appropriato usare il comando "warning"
    print("Il valore di x deve essere positivo")
    out <- NaN # Restituisco Not A Number
  } else {
    out <- sqrt(x)
  }
  out
}

square_root(-2) # La condizione x < 0 è verificata
# [1] "Il valore di x deve essere positivo"
# [1] NaN

square_root(36) # La condizione x < 0 NON è verificata
# [1] 6
```

Cicli `while` e `for` I

- I cicli `for` ed i cicli `while` ripetono l'operazione contenuta tra le parentesi graffe {} fintanto che una determinata condizione non si è verificata.

- Cominciamo con un semplice esempio relativo alla funzione `while`:

```
i <- 5 # Partiamo con i = 25

while (i <= 25) { # Ripete l'operazione fintanto che i non è minore o uguale di 25
  print(i) # Mostra a schermo il valore di i
  i <- i + 5 # Incrementa il valore di i

  # In contesti reali, qui ovviamente ci sono altre operazioni da eseguire
  # ...
}
# [1] 5
# [1] 10
# [1] 15
# [1] 20
# [1] 25
```

Cicli `while` e `for` II

- **Nota**. Attenzione a non creare dei **loop senza fine**!

- Il codice seguente richiede l'interruzione forzata della sessione di **R** oppure, nella peggiore delle ipotesi, il riavvio del computer:

```
# NON ESEGUIRE IL SEGUENTE CODICE!
#
# La condizione i <= 25 è sempre vera perché i non viene aggiornato
i <- 5
while (i <= 25) {
  print(i)

  # Altre operazioni da eseguire
  # ...
}
```

Cicli `while` e `for` III

- In alternativa ai cicli `while`, si può usare la sintassi più esplicita dei cicli `for`.
- Il **ciclo** esegue il contenuto delle parantesi graffe considerando di volta in volta i valori contenuti ad esempio in un vettore:

```
values <- seq(from = 5, to = 25, by = 5)
values
# [1]  5 10 15 20 25

for (i in values) {
  print(i + 2) # Mostra a schermo il valore di i + 2

  # Altre operazioni da eseguire
  # ...
}
# [1] 7
# [1] 12
# [1] 17
# [1] 22
# [1] 27
```

Esercizio riassuntivo I

- Si calcolino gli elementi di una matrice quadrata **D** di dimensione $n \times n$, i cui elementi sono pari

$$d_{ij} = (x_i - x_j)^2, \qquad i, j \in \{1, \ldots, n\},$$

dove $\mathbf{x} = (x_1, \ldots, x_n)^\intercal$ è un generico vettore in $\mathbb{R}^n$.

- In altri termini, si definisca una funzione `distances(x)` che a partire da un generico vettore `x` restituisca una matrice `D`.

- **Traccia dello svolgimento**. Si crei innanzitutto una matrice `D` i cui elementi sono tutti pari a `0`, usando il comando `matrix`.

- Quindi, si usino due cicli `for` "annidati", ovvero uno all'interno dell'altro, per calcolare ciascuno dei valori `D[i, j]`.

Soluzione esercizio riassuntivo I

```r
distances <- function(x) {
  n <- length(x) # Ottengo la lunghezza del vettore x
  D <- matrix(0, nrow = n, ncol = n) # Creazione matrice vuota

  for (i in 1:n) {
    for (j in 1:n) {
      D[i, j] <- (x[i] - x[j])^2
    }
  }
  D # Valore da restituire
}

x <- c(5, 2, 1, 24) # Esempio per verificare che sia corretto
distances(x)
#      [,1] [,2] [,3] [,4]
# [1,]    0    9   16  361
# [2,]    9    0    1  484
# [3,]   16    1    0  529
# [4,]  361  484  529    0
```

- **Nota**. Esistono molti modi diversi (ma corretti) di implementare questa funzione. Per inciso, questa soluzione non è affatto il modo più efficiente.

Esercizio riassuntivo II

- **Fizzbuzz** è un semplice esercizio di programmazione spesso usato nei colloqui di lavoro per verificare le conoscenze di programmazione di base.

- Il compito è il seguente: per tutti i numeri da 1 a 100 si stampi a schermo
  - la parola **fizz** se il numero è un multiplo di 3,
  - la parola **buzz** se è multiplo di 5,
  - la parola **fizzbuzz** se il numero è un multiplo sia di 3 che di 5,
  - il numero stesso altrimenti.

- **Suggerimento**. Si usi la funzione `print` e la funzione resto %% di **R**.

- **Bonus**. Per rendere il codice più snello è possibile usare l'istruzione di controllo chiamata `else if`.

Soluzione esercizio riassuntivo II

- La soluzione riportata fa uso del costrutto `else if`. Vengono riportati solamente i primi 6 valori dei 100 che vengono restituiti.

```
for (i in 1:100) {
  condA <- (i %% 3) == 0 # Il numero è un multiplo di 3?
  condB <- (i %% 5) == 0 # Il numero è un multiplot di 5?

  if (condA & condB) {
    print("fizzbuzz")
  } else if (condA) {
    print("fizz")
  } else if (condB) {
    print("buzz")
  } else {
    print(i)
  }
}
# [1] 1
# [1] 2
# [1] "fizz"
# [1] 4
# [1] "buzz"
# [1] "fizz"
```

La famiglia di funzioni `*apply`

- Ove possibile, sarebbe meglio **evitare** l'utilizzo dei **cicli** `for`, perchè questi tendono ad essere lenti in **R** (a differenza di altri linguaggi, come C++).
- In alcuni casi, questo è possibile tramite la **famiglia di funzioni** `*apply`, ovvero: `apply`, `tapply`, `sapply`, `mapply`, `lapply`.
- Incontreremo le funzioni `*apply` varie volte durante il corso.
- La più semplice, ovvero `apply`, esegue una determinata funzione per ciascuna riga / colonna di una matrice.
- **Esercizio**. Si consulti la documentazione delle funzioni `*apply` per avere una prima idea del loro funzionamento.

I pacchetti **R**

- Come menzionato nella lezione introduttiva, **R** è organizzato in **pacchetti**.
- Se vogliamo utilizzare le funzioni di un pacchetto, questo può essere richiamato usando la funzione `library`, ovvero:

```
library(MASS) # Carica in memoria il pacchetto MASS
library(knitr) # Carica in memoria il pacchetto knitr
```

- Se un pacchetto non è presente nel computer è necessario installarlo.
- Si può usare il comando seguente oppure usare la finestra "Packages" presente in RStudio.

```
install.packages("knitr") # Installa il pacchetto knitr
```