



Appunti di Programmazione

*Sabato De Maio*



# Indice

<b>1</b>	<b>Le Basi</b>	<b>1</b>
1.1	Prime Cose Da Fare . . . . .	1
1.2	Prime Cose da Sapere . . . . .	1
1.2.1	Installare e caricare pacchetti . . . . .	2
1.2.2	Imparare R in R: il pacchetto swirl . . . . .	3
1.2.3	La working directory . . . . .	3
1.3	The Name of the Game . . . . .	4
1.3.1	Classi . . . . .	6
1.4	Working Directory . . . . .	7
1.5	List . . . . .	7
1.6	Matrici . . . . .	8
1.7	Data Frame . . . . .	8
1.8	Names . . . . .	8
1.9	Factors . . . . .	9
<b>2</b>	<b>Funzioni della famiglia apply</b>	<b>11</b>
2.1	Scopo . . . . .	11
2.2	apply . . . . .	11
2.3	lapply e sapply . . . . .	13
2.4	tapply . . . . .	14
2.4.1	split . . . . .	15
2.4.2	by . . . . .	15
2.4.3	Aggregate . . . . .	18
2.5	mapply . . . . .	20
2.5.1	Introduzione alle anonymous functions . . . . .	21
2.5.2	mapply: esempi con le anonymous functions . . . . .	21
<b>3</b>	<b>Subsetting</b>	<b>23</b>
3.1	Introduzione . . . . .	23
3.2	Subsetting di Vettori . . . . .	23
3.3	Subsetting di List . . . . .	25
3.4	Operatori Logici . . . . .	26

<b>4</b>	<b>Grafici</b>	<b>27</b>
4.1	Introduzione . . . . .	27
4.2	Grafici Esplorativi . . . . .	28
4.2.1	BoxPlot . . . . .	28
4.2.2	Istogramma . . . . .	29
4.2.3	Barplot . . . . .	29
4.3	Aggiungere multidimensionalità (in modo semplice) . . . . .	29
4.4	Introduzione 2 . . . . .	30
4.4.1	Base . . . . .	30
4.4.2	lattice . . . . .	31
4.4.3	ggplot2 . . . . .	31
4.5	BASE . . . . .	31
4.6	Graphic Device . . . . .	34
	<b>Aggiornamenti</b>	<b>35</b>

# Elenco delle tabelle



# Elenco delle figure





# Capitolo 1

## Le Basi

### 1.1 Prime Cose Da Fare

La prima cosa da fare è innanzitutto installare R. Il software è disponibile a questo indirizzo, inoltre **dopo** aver installato R potete installare RStudio. Quest'ultimo è un IDE gratuito ed open-source che implementa alcune funzioni proprie, come la funzione View( ), rendendo l'uso di R più "comodo" ed orientato allo sviluppo.

Un'altra importantissima cosa fare è impostare la lingua... in inglese. Quando le cose non vanno come dovrebbero R ci informa con dei messaggi di errore che seguono alla interruzione della esecuzione del comando dato oppure con degli avvertimenti (warnings) che ci informano che sebbene il comando sia stato eseguito, qualche cosa, per i più disparati motivi, non ha funzionato come avrebbe dovuto.

Leggere ed interpretare questi messaggi, a volte criptici, può non essere facile all'inizio. Una cosa da fare potrebbe essere copiare l'errore ed inserirlo nel nostro motore di ricerca preferito per vedere se altri sfortunati utenti di R si sono imbattuti nella stessa fattispecie.

Avere R localizzato in italiano, con messaggi di errore tradotti è una sicura limitazione delle probabilità di trovare una soluzione al nostro problema. È buona regola quindi sovrascrivere l'opzione di R, che di default legge nelle preferenze di sistema del nostro computer, forzando il sistema a "parlare" in inglese.

Questo può essere ottenuto con il seguente codice in R:

---

```
1 > system("defaults write org.R-project.R force.LANG  
    en_US.UTF-8")
```

---

oppure con il seguente codice via terminale (o prompt dei comandi, o shell ecc.):

---

```
1 > defaults write org.R-project.R force.LANG en_US.UTF-8
```

---

### 1.2 Prime Cose da Sapere

Per imparare un software come R è necessario oltre che studiare, principalmente esercitarsi e mettere alla prova le proprie conoscenze. In R è presente un pacchetto con

dei data set pre caricati molto utili per esercitarsi.

Quasi sicuramente la vostra versione di R oltre ad avere già installato questo pacchetto, lo ha anche già caricato in automatico. Per verificare ciò usate la funzione `sessionInfo()` priva di argomenti.

---

```

1 > sessionInfo()
2 R version 3.1.2 (2014-10-31)
3 Platform: x86_64-apple-darwin13.4.0 (64-bit)

4 locale:
5  [1]
6      en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

7 attached base packages:
8  [1] stats      graphics  grDevices  utils      datasets  methods
9      base

10 loaded via a namespace (and not attached):
11  [1] tools_3.1.2

```

---

La linea denominata `attached base packages:` fornisce i nomi dei pacchetti caricati, all'interno della quale potete leggere `datasets`.

### 1.2.1 Installare e caricare pacchetti

La funzione del paragrafo precedente restituisce informazioni sulla nostra versione di R<sup>1</sup>. Come si è detto la linea denominata `attached base packages:` fornisce i nomi dei pacchetti caricati. Qualora `datasets` non fosse presente caricatelo con il seguente comando:

---

```

1 > library(datasets)

```

---

Qualora il pacchetto non fosse installato, e di conseguenza con il comando pretendente R vi restituisce un errore del tipo `Error in library(datasets) : there is no package called 'datasets'` procedete alla installazione manuale del pacchetto con il seguente comando, dopodiché caricatelo con il comando precedente.

---

```

1 > install.packages("datasets")

```

---

Grazie a questo comando il “vostro” R si conatterà con i databases di CRAN e scaricherà l’ultima versione di `datasets`. Questo procedimento è ovviamente valido per qualsiasi dei più di 4000 pacchetti presenti su CRAN.

Da notare l’uso delle virgolette per installare il pacchetto, mentre l’assenza di queste per caricarlo con il comando `library`. Non preoccupatevi se inizieranno a comparire diversi messaggi sulla console: R vi informerà di essersi connesso e di aver iniziato il processo di download del file richiesto.

---

<sup>1</sup>Come il nome in effetti suggerisce.

Ci sono anche altre fonti per pacchetti che necessitano una procedura leggermente diversa per l'installazione ma per ora è meglio tralasciare questo aspetto.

### 1.2.2 Imparare R in R: il pacchetto swirl

Ora che si conosce la procedura per installare e caricare un pacchetto, il primo da installare è sicuramente swirl. Questo pacchetto offre all'interno di R diverse lezioni su molti argomenti base del software che, guidando l'utente passo dopo passo, permettono di acquisire oltre a nozioni pratiche, la dimestichezza con l'interfaccia.

### 1.2.3 La working directory

Concetto fondamentale in R è la cosiddetta working directory, lo spazio lavoro dell'utente, il luogo logico del nostro computer dove R va a cercare i file da caricare ed eventualmente dove salvare i file che noi decidiamo di salvare.

Per sapere quale sia la nostra working directory usiamo il seguente comando che produce come output il percorso della cartella usata da R come working directory.

---

```
1 > getwd()
2 [1] "/Users/sabatodemaio"
```

---

Supponiamo che la cartella che R ha impostato di default non sia di nostro gradimento, ad esempio potremmo voler spostare la nostra working directory in una cartella condivisa con altri utenti, una cartella condivisa con il cloud ad esempio.

Per impostare la working directory è necessario usare il seguente comando:

---

```
1 > setwd("/Users/sabatodemaio/dropbox")
```

---

Ovviamente la cartella di destinazione deve esistere.

Per vedere all'interno della nostra working directory le sotto-cartelle ed i file presenti, molto utile è il comando seguente che restituisce come output un atomic vector di class "character" con tutti gli elementi presenti nella wd.

---

```
1 > list.files()
2 [1] "Applications"          "datasciencecoursera"
3 [3] "Desktop"               "Documenti"
4 [5] "Documents"             "Downloads"
5 [7] "Dropbox"               "GetClean"
6 [9] "knime-workspace"       "Library"
7 [11] "Movies"                "Music"
8 [13] "Pictures"              "Public"
9 [15] "Rom Lollipop"          "UCI HAR Dataset"
10 [17] "wekafiles"
```

---

Altri comandi molto utili per lavorare, settare ed esplorare la working directory sono i seguenti:

- `dir.create("<nome>")` permette di creare all'interno della working directory una cartella con il nome indicato da `<nome>`
- `file.info("<nome>")` permette di ottenere informazioni su di un file o su una cartella specificata dal `<nome>` all'interno della working directory
- `file.rename("nuovo", "vecchio")` permette di rinominare un file o una cartella
- `file.exists("<nome>")` permette di verificare se all'interno della working directory è presente un file denominato `nome`. Restituisce `TRUE` se la il file esiste e `FALSE` in caso contrario.
- `unlink("testdir2", recursive = TRUE)` permette di cancellare un file o una cartella all'interno della working directory<sup>2</sup>. L'argomento `recursive = TRUE` permette ad R di accertare che l'utente è al corrente che all'interno della cartella che si vuole eliminare sono presenti altri file. Se non specificato, R restituisce un errore al fine di avvertirci che all'interno della directory che noi abbiamo deciso di eliminare ci sono altri files o cartelle. In effetti R vuole metterci in guardia dal cancellare in modo maldestro files utili presenti all'interno della cartella che stiamo per eliminare.

### 1.3 The Name of the Game

Cercare un punto da cui partire per introdurre un argomento come R, che come vedremo è vastissimo, non è cosa facile. Un primo punto da cui partire è capire innanzitutto come ragione R e come sono le “parole” che usa, al fine di poter comunicare con questo software<sup>3</sup>

La prima cosa da da capire sono i tipi di oggetti, con cui possiamo è possibile lavorare in R.

Sostanzialmente ogni cosa in R è un oggetto.

Il primo tipo è costituito da “atomic vector”, chiamati così in quanto sono l'unità fondamentale alla base di R. Proviamo ad esempio a creare e a stampare, semplicemente scrivendo il suo nome, un elemento che chiameremo `first`, costituito dal solo numero 1.

---

```

1 > first <- c(1)
2 > first
3 [1] 1

```

---

Per quanto semplice e banale possa sembrare l'esempio precedente esso introduce sicuramente alcune fondamentali caratteristiche che analizzeremo e potrebbe far sorgere anche qualche interessante domanda. Andiamo per ordine.

Per prima cosa notiamo che il prompt di R (o la shell di R) restituisce sempre un segno `>` quando diamo dei comandi, mentre restituisce il numero del primo elemento

---

<sup>2</sup>Potrebbe apparire più logico aver creato questo comando con il nome `dir.delete` ma probabilmente gli autori di R erano degli utenti Linux.

<sup>3</sup>R è sì un software ma è anche un linguaggio derivato a sua volta da altri linguaggi di più basso livello, S ed S-Plus. Per ora, al fine di non introdurre troppi concetti si passi questa definizione superficiale.

della riga quando “stampiamo” a video un oggetto. In questo caso abbiamo un elemento di nome `first` che abbiamo stampato, semplicemente scrivendo il suo nome, ed il numero uno tra parentesi quadre ci indica appunto che quella riga inizia con il primo elemento di `first`. Da notare che R non produce nessun output con la prima stringa di codice.

Tornando leggermente indietro, analizziamo il codice che ci ha permesso di creare l'elemento `first`.

Il primo importante operatore che abbiamo usato è stato `<-` composto da un segno minore ed un segno meno. Importante è non inserire alcuno spazio tra questi due simboli. Questo operatore permette di definire oggetti, attribuire loro determinati valori in senso lato (quindi anche funzioni, matrici, ecc.).

Un modo semplice per comprendere concettualmente questo operatore è quello di interpretarlo come una freccia (cosa che da un punto di vista grafico ricorda molto) che punta verso sinistra. In pratica è come se dicessimo ad R “vedi questa stringa che ho digitato, `first`, assegna il valore che è alla destra della freccia”.

Infine a volte si potrebbe essere tentati di usare `=` in luogo di `<-` ma ciò oltre ad essere sconsigliato potrebbe causare problemi. Di fatto l'utilizzo del simbolo di uguaglianza è stato ritenuto per meri motivi di retro-compatibilità<sup>4</sup>.

Il secondo, ed importantissimo, elemento introdotto è `c( )`. Questa è una funzione che permette di combinare elementi e di generare, dalla combinazione di questi, un vettore.

Ecco ora la domanda che, si spera, a qualcuno sia balenata nella mente: “Un vettore con un solo elemento, perché non uno scalare?”. La risposta è semplice: R non ha molta simpatia per gli scalari, non li conosce e non è in grado di riconoscerli; semplicemente per lui non esistono.

Da questa importante caratteristica ne deriva che anche un solo elemento, in questo caso numerico, è considerato come un vettore di lunghezza uno ma pur sempre un vettore. Questa caratteristica ha, come si vedrà, importanti effetti.

Ricapitolando:

- con l'operatore `<-` abbiamo creato un oggetto chiamato `first`, alla sinistra dell'operatore
- al quale abbiamo attribuito valore 1
- mediante la funzione `c( )`

Trattandosi un vettore costituito da un solo elemento avremmo potuto usare anche il seguente codice, che si è preferito trascurare per presentare al lettore la funzione `c( )`, che come detto tornerà utile praticamente sempre nell'uso di R.

---

```
1 > first1 <- 1
2 > first1
3 [1] 1
```

---

<sup>4</sup>Sia le linee guida di Google che su diversi forum proibiscono o sconsigliano l'uso semplice di un `=`. Alcuni link=1 sembra che le due cose abbiano funzioni diverse anche se provando l'esempio proposto in Rentrambi funzionano.

Per verificare che sia il primo codice che il secondo producono un vettore<sup>5</sup> usiamo il seguente codice mediante il quale, sostanzialmente, chiediamo ad R “l’argomento della funzione `is.atomic` è un atomic vector?”.

---

```
1 > is.atomic(first)
2 [1] TRUE
3 > is.atomic(first1)
4 [1] TRUE
```

---

Proviamo ora a creare e stampare a video un elemento, di nome `second`, composto dai numeri 1, 3, 5, 6, 9, con il codice seguente.

---

```
1 > second <- c(1,3,5,6,9)
2 > second
3 [1] 1 3 5 6 9
```

---

In questo caso avendo un numero di elementi maggiore di uno l’uso della funzione `c()` si rende necessario. Senza di essi infatti avremmo un errore.

Un’altra fondamentale caratteristica è che un vettore in R può contenere solo oggetti appartenenti alla stessa classe. Per ora si tralasci il concetto di classe che sarà introdotto nel prosieguo e si rifletta sul semplice fatto che sia abbastanza logico aspettarsi che un software tratti un maniera diversa un numero, rispetto ad una lettera o rispetto ad un carattere o ancora rispetto ad un condizione di vero o falso e così via.

Tuttavia se creo un vettore con oggetti appartenenti a classi diverse in realtà R non mi dà errore ma forza la creazione del vettore stesso modificando gli oggetti in esso contenuti in modo tale da farli appartenere alla stessa classe. C’è una precisa gerarchia con cui R fa questa operazione.

### 1.3.1 Classi

La classe di un oggetto è sostanzialmente un attributo di quest’ultimo. Essa serve ad R per capire come relazionarsi con gli altri oggetti e a definire il comportamento dell’oggetto stesso. Per ora si considerino le seguenti classi di oggetti, ognuna delle quali ha fortissime implicazioni nel modo in cui R tratta gli oggetti:

- `character` (stringhe di caratteri)
- `numeric` (numeri reali)
- `integer` (che in realtà non sono numeri interi, ma “livelli”), utile per descrivere “mutabili” e non “variabili”
- `complex` (numeri complessi)
- `logical` (vero/falso)

---

<sup>5</sup>Numerico, ma per ora è meglio tralasciare questo aspetto.

## 1.4 Working Directory

Una delle prime cose da impostare e capire in R è la working directory. Per vedere in quale cartella del nostro computer stiamo lavorando è necessario usare il codice:

---

```
1 > getwd()  
2 [1] "/Users/sabatodemaio"
```

---

## 1.5 List

Un altro tipo di oggetti usati in R sono le list. Le list sostanzialmente sono dei vettori, ma non degli atomic vectors. Le list sono definite recursive vectors perché sostanzialmente sono delle collezioni di oggetti che possono contenere elementi appartenenti a più classi diverse.

Creiamo ora, con il comando `list ( )` una lista chiamata impiegato, con alcune caratteristiche di quest'ultimo:

---

```
1 > impiegato <- list(nome = "Mario", cognome = "Rossi",  
2 stipendio = 100000, sindacato = TRUE)  
3 > impiegato  
4 $nome  
5 [1] "Mario"  
  
7 $cognome  
8 [1] "Rossi"  
  
10 $stipendio  
11 [1] 1e+05  
  
13 $sindacato  
14 [1] TRUE
```

---

Da notare che abbiamo usato per ogni caratteristica dei nomi, rappresentati dalle parole alla sinistra di ogni simbolo `=`; ciò può essere visto con il comando `names( )` che ci restituisce i nomi degli oggetti che compongono la lista.

---

```
1 > names(impiegato)  
2 [1] "nome"      "cognome"    "stipendio"  "sindacato"
```

---

Spesso per pratiche applicazione nominare gli elementi di una lista è molto utile. Il risultato comunque sarebbe stato lo stesso anche se non li avessimo usati; il seguente codice non utilizza nomi ma produce lo stesso risultato tranne che, ovviamente, per l'assenza dei nomi che R non restituisce non essendo stati specificati dall'utente.

---

```
1 > impiegato1 <- list("Mario", "Rossi", 100000, TRUE)
```

---

```

2 > impiegato1
3 [[1]]
4 [1] "Mario"

6 [[2]]
7 [1] "Rossi"

9 [[3]]
10 [1] 1e+05

12 [[4]]
13 [1] TRUE

15 > names(impiegato1)
16 NULL

```

---

## 1.6 Matrici

In R una matrice è sostanzialmente un vettore con due attributi dimensionali: il numero delle righe ed il numero delle colonne. Gli attributi dimensionali a loro volta possono essere conservati in un altro vettore di interi.

Essendo la matrice una particolare forma di vettore ne deriva che anche nelle matrici possono essere conservati dati appartenenti ad una sola classe.

## 1.7 Data Frame

Da un punto di vista concettuale i data frame possono essere considerati come delle matrici. Da un punto di vista più tecnico un data frame è una list i cui elementi sono dei vettori di uguale dimensione.

Ne consegue che, per definizione di list, all'interno di un data frame possono essere conservati oggetti appartenenti a diverse classi; cosa non possibile con le matrici.

## 1.8 Names

Dare dei nomi agli oggetti spesso può risultare molto utile. Sostanzialmente tutti gli oggetti in R possono avere nomi. È possibile attribuire il nome con funzione `names()`. Nell'esempio, `<oggetto>` è un qualsiasi oggetto R che vogliamo rinominare, mentre `<vettore>` è un vettore contenente i nomi, o anche solo il singolo nome (in questo caso ovviamente è possibile omettere la funzione `c()` necessaria per creare vettori).

---

```

1 names(<oggetto>) <- <vettore>

```

---



Il seguente esempio può sicuramente aiutare a meglio comprendere l'uso di questa funzione.

---

```
1 > names(impiegato1)
2 NULL
3 > impiegato1
4 [[1]]
5 [1] "Mario"

7 [[2]]
8 [1] "Rossi"

10 [[3]]
11 [1] 1e+05

13 [[4]]
14 [1] TRUE

16 > nomi <- c("nome", "cognome", "stipendio", "sindacato")
17 > names(impiegato1) <- nomi
18 > impiegato1
19 $nome
20 [1] "Mario"

22 $cognome
23 [1] "Rossi"

25 $stipendio
26 [1] 1e+05

28 $sindacato
29 [1] TRUE
```

---

## 1.9 Factors

Factors sono usati per rappresentare dati a categorie (categorical data), caratteristiche non misurabili, o come spesso chiamati in statistica, “mutabili” quindi proprietà di unità statistiche che misurano caratteristiche di queste ultime piuttosto che valori variabili. Esempi possono essere il sesso, l’orientamento politico ecc.

I factors possono essere pensati come un vettore che possiede alcune informazioni aggiuntive riguardo i valori che il vettore può assumere. Questi valori sono definiti livelli.

---

```
1 > partiti <- factor(c("PD", "FI", "M5S", "LN", "NCD"))
2 > partiti
```

<sup>3</sup> [1] PD FI M5S LN NCD  
<sup>4</sup> Levels: FI LN M5S NCD PD

---

## Capitolo 2

# Funzioni della famiglia `apply`

### 2.1 Scopo

Molto spesso nelle pratiche applicazioni al fine di condurre nel migliore dei modi la nostra analisi si rende necessario applicare una stessa funzione a una serie di elementi.

Il modo principale per ottenere ciò è attraverso la creazione di un apposito loop (una ripetizione consecutiva un numero di volte) con i costrutti di R a questo scopo preposti come `for`, `if`, eccetera.

Spesso però, con operazioni abbastanza frequenti (come la somma di tutti i valori di una colonna ad esempio) lo stesso risultato si può ottenere con una famiglia di funzioni appositamente costruita per sfruttare le caratteristiche proprie degli oggetti a cui si applicano ed ottenere lo stesso risultato di un loop ma con meno codice.

Queste funzioni appartengono alla cosiddetta famiglia `apply`.

### 2.2 `apply`

La prima di queste funzioni è `apply`. Questa funzione opera in maniera molto semplice permettendo con poco codice di applicare un'altra funzione, da specificare all'interno di `apply`, su una dimensione di un oggetto array e quindi anche sulla forma più basilare di array, un array bidimensionale o più volgarmente, una matrice.

Nella esecuzione non è più veloce di un loop (lo era nelle prime versioni del linguaggio S), è però molto più pratica in quanto come si è detto, richiede molto meno codice.

La sintassi è la seguente:

---

```
1 apply(<array>, <dimensione>, <funzione>)
```

---

Dove con `array` abbiamo un oggetto di tipo array anche multidimensionale ma anche un data frame, con `funzione` abbiamo il nome e soltanto il nome di una funzione da applicare e con `dimensione` abbiamo una delle dimensioni dell'oggetto.

La particolarità da ricordare onde evitare di incorrere in errori, è che l'oggetto `<funzione>` all'interno di `lapply` va inserito senza alcuna parentesi. In pratica va scritto

solo il nome della funzione usata. Ad esempio se vogliamo la media, dobbiamo usare la funzione `mean` e scrivere solo il nome di quest'ultima.

Fondamentale è anche capire il concetto di dimensione. Banalmente una matrice (che è un array bidimensionale), ha appunto due dimensioni: righe e colonne. Ne consegue che quindi specificando 1 la funzione `<funzione>` all'interno di `<array>` sarà applicata alle righe mentre con 2 sarà applicata alle colonne.

Vediamo un esempio in cui creiamo un data frame usando due vettori di valori e successivamente applichiamo a quest'ultimo la funzione `sd` al fine di trovare la deviazione standard di ogni colonna.

---

```

1 > valori1 <- c(1:5)
2 > valori1
3 [1] 1 2 3 4 5
4 > valori2 <- c(10:15)
5 > valori2
6 [1] 10 11 12 13 14
7 >
8 > dataF <- data.frame(valori1, valori2)
9 > dataF
10   valori1 valori2
11 1         1      10
12 2         2      11
13 3         3      12
14 4         4      13
15 5         5      14
16 > apply(dataF, 2, sd)
17   valori1  valori2
18 1.581139 1.581139

```

---

Alcune considerazioni sull'esempio precedente:

- la funzione `apply` si applica a data frame e array ed ha come output un vettore di elementi; come si vedrà non sempre l'output è un oggetto vettore.
- è stato utilizzato l'operatore `colon` dato dai due puntini `< : >` per creare una sequenza di numeri da 1 a 5 e da 10 a 14, quindi per ottenere due vettori di 5 elementi.
- la lunghezza dei vettori da unire nel data frame deve essere uguale altrimenti R avrebbe restituito un errore.
- il numero 2 usato come secondo elemento della funzione `apply` sta ad indicare la seconda dimensione del nostro oggetto e cioè le colonne. Sarebbe stato possibile calcolare la deviazione standard per ogni riga specificando la dimensione 1 e cioè righe.

## 2.3 lapply e sapply

La seconda di queste funzioni è **lapply**<sup>1</sup>, diminutivo di list apply. L'idea alla base di questa funzione è che data una lista di oggetti **lapply** applica una determinata funzione a tutti gli elementi di quella lista, quindi ad ogni singolo oggetto.

Questo oltre ad essere molto utile per oggetti che sono liste in senso stretto ha anche altre implicazioni. A ben ricordare un data frame non è altro che una list composta da vettori di eguali dimensioni.

La funzione **lapply** quindi prende ogni colonna del data frame, che in realtà non è altro che un elemento vettore a sua volta parte del data frame che è una list, e vi applica una specifica funzione. Ne consegue che è come se avessimo creato un loop per tutte le colonne (ma anche le righe) di un data frame.

La sintassi è molto semplice:

---

```
1 lapply(<dataframe>, <funzione>)
```

---

Ecco un esempio in cui creiamo un data frame usando due vettori di valori e successivamente applichiamo a quest'ultimo la funzione **sd** al fine di trovare la deviazione standard di ogni colonna.

---

```
1 > dataF
2   valori1 valori2
3 1         1     10
4 2         2     11
5 3         3     12
6 4         4     13
7 5         5     14
8 >
9 > lapply(dataF, sd)
10 $valori1
11 [1] 1.581139

13 $valori2
14 [1] 1.581139
```

---

Come già detto, con **lapply** si ottiene come output un oggetto di classe list. Ciò a volte potrebbe non essere desiderabile e per ottenere un oggetto di tipo vettore, in R è implementata un'altra funzione che lavora esattamente come **lapply** ma semplifica il risultato offrendo come output un vettore. Questa funzione è **sapply**, che sta per simplified lapply.

Ecco un esempio usando il data frame creato in precedenza.

---

```
1 > sapply(dataF, sd)
2   valori1   valori2
```

---

<sup>1</sup>È una funzione definita internamente in C.

3 1.581139 1.581139

Questa funzione opera dietro le quinte una semplificazione del risultato ottenuto, ma per far ciò deve cercare in qualche modo di capire quale potrebbe essere un risultato semplificato.

Questo processo che **sapply** esegue in background, con basi di dati di grosse dimensioni potrebbe essere abbastanza lento per i normali tempi di esecuzione e a volte anche per l'utente che potrebbe avvertire il lungo tempo di esecuzione.

Per ovviare a ciò potrebbe essere utile usare la funzione **vapply** che richiede un terzo argomento obbligatorio che indica il formato di output desiderato. In questo modo la computazione è molto più veloce perché la funzione sa già cosa deve restituire. Tuttavia R fa esattamente ciò che l'utente dice di fare a meno che non si tratti di qualcosa di impossibile, è bene quindi prestare attenzione a non richiedere un formato di output impossibile per i dati che **vapply** deve processare.

## 2.4 tapply

La funzione **tapply** è usata per applicare una funzione su sottoinsiemi di un oggetto matrice o data frame o array, divisi secondo particolari istruzioni definite da un altro sottoinsieme dello stesso oggetto<sup>2</sup>.

Immaginiamo di avere un vettore numerico e di voler applicare una funzione a sottoinsiemi di questo vettore basati su alcune caratteristiche definite da un altro vettore (che quasi sempre è un factor). In pratica **tapply** divide un vettore in pezzi più piccoli ed applica la funzione ad ognuno di questi.

È molto utile vedere un esempio. Supponiamo di misurare per un gruppo di sei persone, il loro sesso e il loro reddito. Al fine di condurre una ricerca su una eventuale differenza di reddito tra uomini e donne sarebbe utile con i nostri dati calcolare la media del reddito per ogni gruppo: uomini e donne.

Per prima cosa creiamo un data base e successivamente usiamo la funzione **tapply** per applicare la funzione **mean** (media), sulla colonna reddito suddivisa però sulla base dei due livelli presenti nella colonna genere, quindi per sesso maschile e femminile.

```
1 > d <- data.frame( list( genere = c("M", "M", "F", "M", "F", "F"),
2   reddito = c(55000, 88000, 32450, 76500, 123000, 45650) ) )
3 > d
4   genere reddito
5 1      M   55000
6 2      M   88000
7 3      F   32450
8 4      M   76500
9 5      F  123000
10 6      F   45650
```

<sup>2</sup>Ma anche di altri in effetti.

```

11 > tapply(d$reddito, d$genere, mean)
12      F      M
13 67033.33 73166.67

```

---

L'output è composto dalle due medie della colonna reddito, una calcolata sulla base del fattore M e l'altra sulla base del fattore F.

### 2.4.1 split

Come abbiamo visto, **tapply** divide l'oggetto in gruppi ai quali applica separatamente una data funzione invocata dall'utente. La funzione **split** sostanzialmente si ferma al primo passo.

La funzione **split** divide un oggetto R sulla base delle indicazioni dell'utente e ha come output un oggetto list.

Nell'esempio sottostante si utilizza il data frame precedente, per ottenere due data frame entrambi elementi di un oggetto di livello superiore di class list.

```

1 > d1 <- split(d, d$genere)
2 > d1
3 $F
4   genere reddito
5 3      F   32450
6 5      F  123000
7 6      F   45650

9 $M
10  genere reddito
11 1      M   55000
12 2      M   88000
13 4      M   76500

```

---

### 2.4.2 by

Da un punto di vista concettuale la funzione **by** è esattamente come **tapply** con una piccola ma fondamentale differenza che la rende uno strumento potentissimo. Come si è visto la funzione **tapply** ha bisogno di due vettori per poter funzionare, dove il primo rappresenta i dati sui quali applicare la funzione ed il secondo i gruppi in base ai quali suddividere il primo vettore.

La funzione **by** permette di applicare una funzione ma non richiede che il primo elemento sia un vettore di uguali dimensioni ma accetta anche matrici, arrays e data frame. Per meglio comprendere le implicazioni di questa differenza ci serviremo di un esempio usando il database iris presente nel pacchetto **datasets**.

Supponiamo di voler calcolare per ogni fattore della colonna Species, le statistiche descrittive relative alla colonna Sepal.Width usando la utilissima funzione **summary**. Per

questo semplice compito è possibile utilizzare sia la funzione `by` che `tapply`. Il codice seguente mostra l'uso di entrambe le funzioni, si noti che i risultati sono stati salvati in oggetti ai quali è stata applicata successivamente la funzione `class`.

Si può notare come entrambi gli oggetti non siano atomic vectors ma strutture più complesse di classi differenti. Questa è la prima differenza.

---

```

1 > ct <- tapply(iris$Sepal.Width, iris$Species, summary )
2 > class(ct)
3 [1] "array"
4 > is.atomic(ct)
5 [1] FALSE
6 > cb <- by(iris$Sepal.Width, iris$Species, summary )
7 > class(cb)
8 [1] "by"
9 > is.atomic(cb)
10 [1] FALSE

```

---

Stampando a video gli oggetti `ct` e `cb` che contengono i nostri output si può notare come, nonostante la classe differente, i risultati siano sostanzialmente gli stessi e, in questo caso, l'uso dell'una o dell'altra funzione può essere indifferente.

---

```

1 > ct
2 $setosa
3   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
4  2.300   3.200   3.400   3.428   3.675   4.400

6 $versicolor
7   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
8  2.000   2.525   2.800   2.770   3.000   3.400

10 $virginica
11   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
12  2.200   2.800   3.000   2.974   3.175   3.800

14 > cb
15 iris$Species: setosa
16   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
17  2.300   3.200   3.400   3.428   3.675   4.400

18
19 iris$Species: versicolor
20   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
21  2.000   2.525   2.800   2.770   3.000   3.400

22
23 iris$Species: virginica
24   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.

```

---



```

25 2.200 2.800 3.000 2.974 3.175 3.800
26 >

```

La vera differenza tra **tapply** e **by** entra in gioco quando l'elemento lungo il quale applicare la funzione non è più un vettore ma un data frame (ma anche array o matrice).

Il seguente codice interrompe la sua esecuzione e restituisce un messaggio di errore che ci informa che la dimensione dell'indice e dei dati sui quali applicare la funzione hanno dimensioni differenti e quindi R non sa come comportarsi.

```

1 > tapply(iris, iris$Species, summary)
2 Error in tapply(iris, iris$Species, summary) :
3   arguments must have same length

```

Proviamo invece esattamente la stessa sintassi utilizzando però la funzione **by** al posto di **tapply**.

```

1 > by(iris, iris$Species, summary)
2 iris$Species: setosa
3   Sepal.Length   Sepal.Width   Petal.Length   Petal.Width
4   Min.      :4.300   Min.      :2.300   Min.      :1.000   Min.      :0.100
5   1st Qu.:4.800   1st Qu.:3.200   1st Qu.:1.400   1st Qu.:0.200
6   Median :5.000   Median :3.400   Median :1.500   Median :0.200
7   Mean    :5.006   Mean    :3.428   Mean    :1.462   Mean    :0.246
8   3rd Qu.:5.200   3rd Qu.:3.675   3rd Qu.:1.575   3rd Qu.:0.300
9   Max.    :5.800   Max.    :4.400   Max.    :1.900   Max.    :0.600
10
11 iris$Species: versicolor
12   Sepal.Length   Sepal.Width   Petal.Length   Petal.Width
13   Min.      :4.900   Min.      :2.000   Min.      :3.000   Min.      :1.000
14   1st Qu.:5.600   1st Qu.:2.525   1st Qu.:4.000   1st Qu.:1.200
15   Median :5.900   Median :2.800   Median :4.350   Median :1.300
16   Mean    :5.936   Mean    :2.770   Mean    :4.260   Mean    :1.326
17   3rd Qu.:6.300   3rd Qu.:3.000   3rd Qu.:4.600   3rd Qu.:1.500
18   Max.    :7.000   Max.    :3.400   Max.    :5.100   Max.    :1.800
19
20 iris$Species: virginica

```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
21	Species			
22	Min. :4.900	Min. :2.200	Min. :4.500	Min. :1.400
	setosa : 0			
23	1st Qu.:6.225	1st Qu.:2.800	1st Qu.:5.100	1st Qu.:1.800
	versicolor: 0			
24	Median :6.500	Median :3.000	Median :5.550	Median :2.000
	virginica :50			
25	Mean :6.588	Mean :2.974	Mean :5.552	Mean :2.026
26	3rd Qu.:6.900	3rd Qu.:3.175	3rd Qu.:5.875	3rd Qu.:2.300
27	Max. :7.900	Max. :3.800	Max. :6.900	Max. :2.500

Il risultato è ben diverso. La funzione `by` ha applicato la funzione `summary` a tutto il data frame iris dividendolo secondo i fattori presenti nella colonna da noi indicata, nell'esempio `Species`, restituendo un oggetto sempre di classe `by` che contiene le statistiche descrittive fornite da `summary` per tutte le colonne di iris.

Questa differenza fa sì che in molte applicazioni, soprattutto in analisi che comportano l'applicazione di modelli stocastici, la funzione `by` sia a volte da preferire ma spesso "l'unica" via possibile per ottenere il risultato desiderato<sup>3</sup>.

### 2.4.3 Aggregate

Alla funzione `aggregate` è dedicato questo sotto-paragrafo che è tale non per una minore importanza dell'argomento, ma semplicemente perché `aggregate` può essere interpretata da un punto di vista concettuale come una sorta di tappi potenziata. Sebbene il suffisso `apply` non sia presente nel nome di questa funzione, essa è da considerarsi a pieno titolo appartenente a tale famiglia perché al suo interno applica `tapply`. Ne consegue che `aggregate` potrebbe essere usata in luogo di `tapply`. I risultati sarebbero gli stessi ma il formato di output sarebbe leggermente diverso.

Si osservi il seguente esempio costruito usando il database iris presente di default in R nel pacchetto `datasets`<sup>4</sup>

```

1 > prova <- tapply(iris$Sepal.Length, iris$Species, mean)
2 > prova
3      setosa versicolor virginica
4      5.006      5.936      6.588
5 >
6 >

```

<sup>3</sup>Si notino le virgolette; Per quanto non dimostrabile come affermazione è palese fin dai primi utilizzi che in R il detto che tutte le strade portano a Roma è vero più che mai. È molto difficile se non quasi impossibile non trovare un modo alternativo di ottenere ciò che si vuole utilizzando altre funzioni o strumenti di R. Si rammenti inoltre la vastissima e mondiale comunità di sviluppatori, appassionati, statistici ecc. che scrivono pacchetti.

<sup>4</sup>Si usi `data()` per vedere tutti dataset preinstallati in R.

```

7 > prova1 <- aggregate(iris$Sepal.Length, list(iris$Species),
  mean)
8 > prova1
9      Group.1      x
10 1      setosa 5.006
11 2 versicolor 5.936
12 3  virginica 6.588
13 >
14 >
15 > is.atomic(prova)
16 [1] TRUE
17 > is.atomic(prova1)
18 [1] FALSE
19 >
20 >
21 > class(prova)
22 [1] "array"
23 > class(prova1)
24 [1] "data.frame"

```

---

Come si può notare dal codice le due funzioni presentano gli stessi risultati sotto forma diversa. `tapply` restituisce un array mentre `aggregate` un data frame. Tuttavia la vera differenza, che si spera non sia sfuggita ad un lettore attento, è negli argomenti delle due funzioni.

La funzione `aggregate` ha come secondo argomento, l'indice il base al quale suddividere che abbiamo visto in `tapply`, un oggetto di tipo `list`. Da questa caratteristica deriva la vera potenza di `aggregate`: poter applicare una determinata funzione non già su una determinata caratteristica ma bensì su di una lista di caratteristiche. Vediamo un esempio.

Supponiamo di voler calcolare la media di `Sepal.Length` sulla base della specie, dalla colonna `Species` ed anche sulla base della variabile `Petal.Width`. Il codice seguente permette di fare questo. L'unico elemento necessario è quello di usare la funzione `list` per indicare per quali elementi l'oggetto della nostra funzione, la colonna `Sepal.Length`, deve essere suddiviso.

```

1 > prova5 <- aggregate(iris$Sepal.Length, list(iris$Species,
  iris$Petal.Width), mean)
2 > head(prova5)
3      Group.1 Group.2      x
4 1   setosa      0.1 4.820000
5 2   setosa      0.2 4.972414
6 3   setosa      0.3 4.971429
7 4   setosa      0.4 5.300000
8 5   setosa      0.5 5.100000
9 6   setosa      0.6 5.000000

```

```

10 > tail(prova5)
11      Group.1 Group.2      x
12 22 virginica      2.0 6.650000
13 23 virginica      2.1 6.916667
14 24 virginica      2.2 6.866667
15 25 virginica      2.3 6.912500
16 26 virginica      2.4 6.266667
17 27 virginica      2.5 6.733333
18 >
19 >
20 > class(prova5)
21 [1] "data.frame"

```

---

Piccola postilla in conclusione di questo paragrafo. Lo stesso risultato in verità si potrebbe ottenere con la stessa funzione `tapply`, tuttavia il risultato sarebbe stato un oggetto poco pratico da usare. Si provi il seguente codice.

---

```

1 prova3 <- tapply(iris$Sepal.Length, list(iris$Species,
      iris$Petal.Width), mean)

```

---

## 2.5 mapply

Da un punto di vista concettuale `mapply` potrebbe essere quella più tricky. La forza di `mapply` sta nel fatto di poter essere applicata ad elementi multipli. Il funzionamento è il seguente: dati due (o più) oggetti R, ad esempio due vettori di uguale lunghezza, `mapply` estrae il primo elemento di ognuno e vi applica la funzione specificata, dopodiché estrae il secondo elemento di ogni oggetto e vi applica la funzione, dopodiché estrae il terzo elemento di ogni oggetto e vi applica la funzione... e così via fino a che tutti gli elementi di tutti gli oggetti non sono stati trattati.

Come al solito un esempio può essere chiarificatore.

Supponiamo di avere due vettori `p1` e `p2` e di voler sommare il primo elemento di `p1` con il primo di `p2`, il secondo di `p1` con il secondo di `p2` ecc.

---

```

1 > p1 <- c(1:5)
2 > p2 <- c(10:14)
3 > mapply( "+", p1, p2)
4 [1] 11 13 15 17 19

```

---

Considerazioni sull'esempio:

- l'esempio è volutamente banale; lo stesso risultato si sarebbe potuto avere semplicemente con `> p1 + p2`, alcuni esempi più complessi si avranno nei successivi paragrafi<sup>5</sup>.

---

<sup>5</sup>Oltre che banale questo esempio in realtà è anche, volutamente e per fini didattici, sbagliato. Esso

- si noti l'uso delle virgolette nello specificare la funzione somma.
- il semplice operatore di addizione, il simbolo "+", è in realtà una funzione.

### 2.5.1 Introduzione alle anonymous functions

Al fine di meglio comprendere la funzione `mapply` fornendo esempi più complessi, si introducono brevemente le anonymous functions molto usate dalle funzioni della famiglia `apply`. Le anonymous functions sono funzioni che non hanno un nome perché definite solo all'interno della funzione (`mapply`, ma anche `apply`, `sapply` ed altre) che di esse si serve ed usate solo al suo interno. Una volta eseguite, sostanzialmente spariscono.

Un esempio semplice è rappresentato dal seguente codice che sostanzialmente istruisce R su una funzione composta da due argomenti `x` ed `y` che restituisce come output un valore dato dalla potenza del primo, `x`, elevato alla seconda, `y`.

---

```
1 function(x, y) x^y
```

---

### 2.5.2 mapply: esempi con le anonymous functions

Ora che brevemente sono state introdotte le anonymous functions è possibile sfruttarle per costruire qualche esempio più complesso su come utilizzare la funzione `mapply`.

Supponiamo di avere tre vettori di eguale lunghezza

---

```
1 > p1 <- c(1:5)
2 > p2 <- c(10:14)
3 > p3 <- c(-1, 1, -1, 2, 1)
```

---

Usiamo questi vettori e la funzione `mapply` per ottenere quel valore dato dalla quoziente avente al numeratore il prodotto del primo elemento di `p1`, moltiplicato il primo elemento di `p2` e come denominatore il primo elemento di `p3`.

---

```
1 > mapply(function(x, y, z) (x*y)/z, p1, p2, p3)
2 [1] -10 22 -36 26 70
```

---

Una simile funzione in R non esiste ecco perché si è reso necessario definirla all'interno della funzione `mapply`.

Un altro esempio illustrativo potrebbe essere il seguente che fa uso della funzione `rep(<valore>, <volte>)`, che sostanzialmente ripete il `<valore>` un numero di `<volte>`.

---

```
1 > mapply(function(x, y, z){ rep(y, x) * z }, p1, p2, p3)
2 [[1]]
3 [1] -10
```

---



---

infatti non sfrutta la cosiddetta vettorizzazione delle operazioni cosa che per grandi quantità di dati è decisamente più efficiente da un punto di vista del tempo necessario ad effettuare le operazioni. Non si sottovaluti questo aspetto nell'era dei big data.

```
5  [[2]]
6  [1] 11 11

8  [[3]]
9  [1] -12 -12 -12

11 [[4]]
12 [1] 26 26 26 26

14 [[5]]
15 [1] 14 14 14 14 14
```

---

Come prima la funzione anonima `function` ha tre elementi. Essa è istruita per prendere il primo elemento di `p1`, ripeterlo un numero di volte pari al numero del primo elemento `p2` e moltiplicare questo output per il primo elemento di `p3`. Così per tutti gli elementi  $i$ -esimi dei tre vettori di eguale lunghezza.

## Capitolo 3

# Subsetting

### 3.1 Introduzione

Nelle pratiche applicazioni mai (o quasi) i dati con i quali ci si trova a lavorare sono quelli necessari alla nostra analisi. Spesso sarebbe opportuno avere altri dati oppure, al fine di rispondere a determinate domande, si rende necessario estrarre dai dati in nostro possesso un sotto insieme di questi. La operazione di estrazione di sottoinsiemi di dati da oggetti in R (che quindi da un punto di vista concettuale diventano insiemi di livello superiore, che quindi contengono i primi) va sotto di subsetting.

Estrarre un sottoinsieme di dati da una base di dati di maggiori dimensioni può diventare una operazione abbastanza frustrante se non si conoscono le corrette procedure per raggiungere il nostro obiettivo se non ci comprende a fondo come lavorano gli strumenti utilizzati per il subsetting ma soprattutto se non si sa bene con che tipo di oggetto si ha a che fare perché ogni tipo di oggetto richiede un approccio diverso.

### 3.2 Subsetting di Vettori

I primi strumenti da utilizzare per il subsetting sono le funzioni date dalla coppia di parentesi quadre e dalla doppia coppia di parentesi quadre: [`<valore>`] e [[`<valore>`]].

Prima di tutto è bene ricordare e tenere sempre a mente che, sebbene composte da uno ed un solo simbolo, le parentesi quadre sono delle funzioni in R. Ciò potrà tornare utile nelle pratiche applicazioni. Queste funzioni hanno la seguente sintassi:

---

<sup>1</sup> > `<oggetto>[<posizione>]`

---

ed estraggono all'`<oggetto>` il valore corrispondente alla `<posizione>`. La principale differenza è che con la funzione di doppia parentesi quadra è possibile estrarre un solo valore mentre con una singola coppia, come si vedrà, è possibile estrarre più valori, ma anche più valori consecutivi, un intervallo di valori ma anche escludere valori. Alcuni esempi.

Si consideri il seguente codice che crea un vettore di nome **vettore**<sup>1</sup> composto da 100 osservazioni di numeri casali. Si tralasci per ora l'utilizzo della funzione **rnorm** che serve a generare numero pseudo casuali, anche se il codice è abbastanza chiaro<sup>2</sup>.

---

```
1 > vettore <- rnorm(100)
```

---

Sappiamo che **vettore** contiene 100 osservazioni, supponiamo ora di voler estrarre il valore corrispondente alla settantesima osservazione<sup>3</sup>. La funzione **[** è ciò che fa al caso nostro. La sintassi è molto semplice, dall'oggetto **vettore** dal quale vogliamo estrarre (che si scrive per primo) si richiama la funzione **[** che prende come argomento il numero della posizione che vogliamo estrarre.

---

```
1 > vettore[70]
2 [1] -0.4074543
```

---

Lo stesso risultato del precedente codice si può ottenere utilizzando la funzione **[[** che si differenzia dalla semplice **[** in quanto permette di estrarre uno ed uno solo elemento da un oggetto R. Ne consegue che tutti gli esempi seguenti non saranno fattibili con la funzione **[[**.

Supponiamo ora di voler estrarre i valori delle osservazioni che vanno dalla cinquantesima alla sessantesima. La funzione **[** in questo caso avrà due valori separati dalla funzione **:** che serve per generare sequenze. In questo caso è come se si stesse dicendo ad R “dammi i valori di **vettore** che sono dalla posizione cinquanta alla sessanta”.

---

```
1 > vettore[50:60]
2 [1] -0.9688182 -0.1123738  0.1217248 -1.8646036
3 [5] -0.2812464 -0.6120908 -0.6647192 -1.5550553
4 [9] -0.1804152 -0.5585904 -0.5825612
```

---

Si noti che lo stesso risultato si sarebbe potuto ottenere con il seguente codice che seppure più lungo, noioso, nonché palesemente inefficiente introduce l'utilizzo della funzione **c** all'interno della funzione **[**.

---

```
1 > vettore[c(50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60)]
2 [1] -0.9688182 -0.1123738  0.1217248 -1.8646036
3 [5] -0.2812464 -0.6120908 -0.6647192 -1.5550553
4 [9] -0.1804152 -0.5585904 -0.5825612
```

---

Con la funzione **c** all'interno di **[** è possibile selezionare più valori non consecutivi, dove evidentemente l'utilizzo della funzione **:** restituirebbe ciò che a noi serve. Il seguente esempio permette di estrarre utilizzando sia **c** che **[** i valori delle osservazioni che sono al, ad esempio, dodicesimo, venticinquesimo ed ottantunesimo posto nel nostro oggetto.

---

```
1 > vettore[c(12, 25, 81)]
2 [1] -1.2779710  1.8429035  0.4637161
```

---

<sup>1</sup>Che fantasia! ecco perché Keynes chiamava l'Economia (e gli economisti) la scienza triste...

<sup>2</sup>Di fatto la **r** sta per random e **norm** sta per normal.

<sup>3</sup>Il numero settanta non ha alcun significato, è usato ora a mero titolo di esempio



Supponiamo ora di voler estrarre dal nostro vettore tutti i valori escludendo però i valori dal ventesimo al quarantesimo. Per far ciò ci serviremo della funzione `:` e dell'operatore `-` che serve per escludere i valori che lo seguono. Si faccia attenzione al costrutto logico: voler ottenere un sottoinsieme di **vettore** costituito da tutti i valori **esclusi** i valori che vanno dal ventesimo al quarantesimo.

Quest'ultima precisazione è per dissuadere il lettore ad utilizzare il seguente codice `> vettore[-20:40]` che restituirebbe un errore<sup>4</sup>. Il codice corretto è il seguente e per essere tale si serve anche della funzione `c` e non già soltanto dell'operatore `-`.

---

```
1 > vettore[-c(20:40)]
```

---

Si noti che lo stesso risultato si sarebbe potuto ottenere anche con il seguente codice<sup>5</sup>

---

```
1 > prova2 <- vettore[-20:-40]
```

---

### 3.3 Subsetting di List

Il subsetting di oggetti list può a volte essere leggermente più complesso. È bene sapere che le funzioni viste nel precedente paragrafo non perdono la loro validità ma anzi sono usate. Particolare rilievo ha la funzione `[[` quando si tratta di liste e si ricordi che questa funzione permette di estrarre un solo elemento da un oggetto R.

Per illustrare il subsetting da un list si osservi il seguente codice con il quale creiamo una sequenza di numeri da uno a dieci e la salviamo un vettore di nome **numeri**, un vettore di caratteri formato dalle prime dieci lettere dell'alfabeto e successivamente creiamo con la funzione `list` una lista di oggetti che comprende il vettore `lettere`, il vettore `numeri` ed un terzo oggetto `NA`.

---

```
1 > numeri <- 1:10
2 > lettere <- letters[1:10]
3 > lista <- list(lettere, numeri, NA)
```

---

Prima di procedere si osservi che le lettere sono state create estraendo dall'oggetto `letters` le prime dieci. Questo oggetto che contiene le lettere minuscole dell'alfabeto è già presente nella nostra workspace in quanto rappresenta una delle costanti presenti di default in R<sup>6</sup>.

Procediamo ora al subsetting di elementi (ma anche di oggetti in questo caso) da un oggetto di tipo list. Nessuna funzione nuova è introdotta, in sostanza il subsetting di oggetti list si ottiene con una combinazione delle funzioni già viste in precedenza.

Supponiamo di voler estrarre il primo elemento della lista, la sintassi da usare è la stessa di quella usata per un vettore.

---

<sup>4</sup>Di fatto con questo codice si dice ad R di estrarre la sequenza di valori compresi tra la posizione -20 e la posizione 40

<sup>5</sup>Per esercizio si salvino in due oggetti diversi i subsetting fatti con entrambi i metodi e si verifichi che essi siano esattamente uguali utilizzando la funzione `identical(<oggetto1>, <oggetto2>)` che ha una sintassi abbastanza eloquente.

<sup>6</sup>Si utilizzi `?letters` per maggiori informazioni.

```
1 > lista[1]
2 [[1]]
3 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

---

Qualora invece volessimo estrarre il terzo elemento del primo elemento della lista, in altre parole il terzo elemento del vettore che a sua volta è il primo elemento dell'oggetto lista, la sintassi da utilizzare è la seguente, che altro non è che una combinazione delle due funzioni viste finora.

---

```
1 > lista[[1]][3]
2 [1] "c"
```

---

Si osservi che se avessimo usato `[` in luogo di `[[` nel primo elemento del subsetting avremmo ottenuto un valore `NULL`.

---

```
1 > lista[1][3]
2 [[1]]
3 NULL
```

---

### 3.4 Operatori Logici

# Capitolo 4

## Grafici

### 4.1 Introduzione

Nella rappresentazione grafica di dati ci sono delle regole generali da seguire nella costruzione al fine di meglio spiegare cosa i dati contengono, quali informazioni possiedono ecc.

Alcuni punti fondamentali da seguire sono i seguenti:

**show comparisons** È sempre utilissimo, se non vitale<sup>1</sup>, presentare dati comparati, grandezze relative ad altre grandezze in modo da poter fare paragoni. Ogni ipotesi è sempre relativa rispetto ad un'altra, quindi chiedersi sempre “questi dati rappresentati a cosa sono comparati?”.

**show causality, mechanism, explanation, systematic structure** includere una anche probabile sovrastruttura, al fine di chiedersi quale è il contesto generale, il framework, all'interno del quale questi dati si collocano. Includere relazioni di cause e/o effetto, ma anche semplici relazioni al fine sia di arricchire l'analisi ma anche eliminare, qualora le relazioni fossero sbagliate, possibili fonti di errore che apporterebbero sicuramente più danni all'analisi se scoperte in una fase successiva.

**show multivariate data** dati multivariati perché la realtà è complessa!

**integration of evidence** integrare il più possibile parole, numeri, immagini, diagrammi; fare uso di diversi modi di presentare i dati anche gli stessi; non lasciare che siano gli strumenti a guidare l'analisi. Combinare diversi elementi. Se sai fare solo una tabella non limitarti solo a quello.

**Describe and document the evidence** Un grafico dovrebbe poter raccontare una storia, un fatto in modo completo ma soprattutto credibile. Tutto ciò si raggiunge con descrizioni, unità di misura usate, scale di valori e soprattutto fonti.

**content is king** la prima cosa, la più importante è comunque la storia da raccontare, la research question che si vuole soddisfare. Poi si pensa a rappresentarli.

---

<sup>1</sup>Soprattutto in alcune materie come l'economia

## 4.2 Grafici Esplorativi

All'inizio di una analisi di dati è utile costruire dei grafici esplorativi che siano di rapida esecuzione, richiedano poco codice e che aiutino a far capire cosa c'è al loro interno. Spesso i grafici di questa fase sono molto ampi, includendo molti dati e presentano caratteristiche estetiche che o sono assenti o sono limitate ad una mera informazione anche se è bene ricordare che di fatto spesso, nelle fasi finali, si tramutano in potenti mezzi per comunicare con i grafici.

Al di là di questo, gli scopi principali di rappresentare i dati in forma grafica possono essere molteplici tra cui:

- capire le proprietà dei dati
- trovare regolarità
- suggerire strategie per applicare modelli
- effettuare il debug di una analisi poco efficace
- ma anche comunicare risultati

Nei successivi paragrafi si analizzeranno i primi comandi per creare grafici. Per far ciò ci si servirà di alcuni dei data sets già installati in R forniti col pacchetto **datasets**. Per informazioni su questo pacchetto si veda il paragrafo XXX su come caricarlo e/o installarlo. Per avere una panoramica sui dataset presenti in questo pacchetto e sul loro contenuto, si utilizzi il seguente comando.

---

```
1 > library(help = "datasets")
```

---

### 4.2.1 BoxPlot

È possibile fare un boxplot con il seguente comando:

---

```
1 > boxplot(pollution$pm25, col = "blue")
```

---

dove l'argomento opzionale `col` indica il colore da usare per colorare la nostra scatola. È possibile aggiungere una linea fissata al valore che ai fini della nostra analisi può essere utile, con il seguente comando:

---

```
1 abline(h = 12)
```

---

in questo modo appare evidente graficamente quanti valori sono al di sotto e al di sopra del nostro valore di riferimento.

### 4.2.2 Istogramma

Per costruire un istogramma si usa il comando `hist`. Ecco nel codice seguente un esempio, nel quale come nell'esempio del boxplot, si utilizza l'elemento opzionale `col`.

---

```
1 > hist(pollution$pm25, col = "green")
```

---

L'istogramma può essere personalizzato aggiungendo i dati stessi sotto forma di “tappeto” con il comando `rug`, molto utile soprattutto per verificare e constatare anche visivamente la presenza di valori eccezionali.

---

```
1 rug(pollution$pm25)
```

---

Posso cambiare i breaks, ossia il numero delle classi in cui dividere le nostre osservazioni con l'argomento opzionale `breaks = 100`. Anche all'istogramma si possono aggiungere linee. Il comando seguente aggiunge una linea verticale, per questo il primo argomento è `v` (fosse stata orizzontale l'argomento sarebbe stato `h`) di spessore (`line width`) 2.

---

```
1 > abline(v = 12, lwd = 2, col = "magenta")
```

---

### 4.2.3 Barplot

È possibile ottenere un barplot con il seguente comando:

---

```
1 barplot(table(pollution$region), col = "wheat", main = "Number  
  of Counties in Each Region")
```

---

Così mi vedo dove sono più numerosi i valori se nella east o west coast.

## 4.3 Aggiungere multidimensionalità (in modo semplice)

grafici bidimensionali - multiple/overlayed plots fatti con lattice o ggplot2 - scatterplot, smooth scatterplot

maggiori di due dimensioni spinning plot, 3d , colori ecc ma spesso è molto utile magari avere comunque un grafico 2d ed usare colori, font, etichette ecc per renderlo multivariato ed aggiungerci diverse cose.

---

```
1 boxplot(pm25 ~ region, data = pollution, col = "red")
```

---

Con questo in effetti mi diventa già a due dimensioni perché ho sull'asse x la costa (east ed west) mentre sul y ho la concentrazione di pm10

Lo stesso si può fare anche con gli istogrammi:

---

```
1 par(mfrow = c(2,1), mar = c(4, 4, 2, 1))
2 hist(subset(pollution, region == "east")$pm25, col = "green")
3 hist(subset(pollution, region == "west")$pm25, col = "green")
```

---

Dove con `par` serve per impostare i parametri dei grafici.  
 lo `scatterplot`:

---

```
1 with(pollution , plot(latitude , pm25))
2 abline( h = 12, lwd = 2, lty = 2)
```

---

dove la funzione è `plot`, che senza specificazione particolari mi stampa uno grafico scatter secondo le impostazioni di `plot.default` posso anche colorare diversamente le regioni in modo da avere una visione più chiara:

---

```
1 with(pollution , plot(latitude , pm25, col =region))
2 abline( h = 12, lwd = 2, lty = 2)
```

---

tutti questi grafici iniziali sono Exploratory plots are "quick and dirty" nel senso che è preferibile non perdere tempo per formattare assi, colori, legende ecc. Uno degli scopi finali è quello di farsi una idea dei dati e soprattutto cercare di capire come approcciare i dati e magari trarre spunti su quale modello utilizzare ecc.

## 4.4 Introduzione 2

### 4.4.1 Base

In R ci sono tre diversi sistemi di disegnare grafici ognuno dei quali con le proprie caratteristiche utili per raggiungere risultati diversi. Unica cosa molto importante da ricordare e da considerare quando si inizia l'analisi di dati è che bisogna scegliere uno ed uno solo e creare grafici seguendo quello, non si possono mischiare le funzioni di uno con quelle di un altro altrimenti il grafico viene (se viene?) in modo confusionale.

Il primo metodo è "base", è il vecchio sistema che era presente in R. Concettualmente è come la tela bianca di un artista. Uno alla volta si aggiungono elementi per comporre il grafico finale aggiungendo assi, text, lines, points axis eccetera. È utile come fase iniziale. le ultime funzioni sono cosiddette annotations invece altre servono proprio per generare il grafico.

Questo sistema è pratico, intuitivo e facile ma non si può andare indietro ad esempio aggiustando i margini. Il "disegnare" si compone da una serie di comandi di R e quindi sostanzialmente non c'è un vero e proprio linguaggio grafico sottostante che permetta ad esempio di creare nuovi tipi di grafici.

Un esempio di un grafico disegnato con base potrebbe essere questo

---

```
1 library(datasets)
2 data(cars)
3 with(cars , plot(speed , dis))
```

---

È semplice ma posso aggiungere e fare qualsiasi cosa... un poco alla volta..

### 4.4.2 lattice

Il secondo sistema per creare grafici è `lattice`. Si attiva con il pacchetto `lattice` e l'idea di fondo è che piuttosto che aggiungere i pezzi uno alla volta al grafico, lo si crea con una singola funzione composta da diversi argomenti.

Utilizzare `lattice` è molto utile per grafici condizionati, inoltre i margini, gli spazi e molti parametri sono impostati automaticamente quindi il risultato oltre ad avere un aspetto più gradevole, richiede meno tempo per aggiustamenti. Questo è utile soprattutto quando si devono esportare i grafici come si vedrà.

Il lato negativo è che spesso è difficile specificare in una sola funzione tutto un grafico, è poco intuitivo annotare, non si può aggiungere nulla al grafico una volta creato ma bisogna ricrearlo.

Un esempio con `lattice` che è relativamente semplice ma che avrebbe richiesto molto più codice per essere fatto in base.

---

```
1 library(lattice)
2 state <- data.frame(state.x77, region = state.region)
3 xyplot(Life.Exp ~ Income | region, data = state, layout = c(4,
  1))
```

---

### 4.4.3 ggplot2

Sostanzialmente il merito di questo pacchetto è di aver creato una sorta di grammatica o addirittura linguaggio per i grafici. Mischia concetti di entrambi i sistemi (base e `lattice`), permette di costruire il grafico un poco alla volta come il sistema base però permette anche di fare in modo automatico i calcoli per i margini, spazi ecc.

Molto utile come `lattice` per fare grafici condizionati. Ha un sacco di parametri impostati su default quindi se l'utente non vuole può lasciare tutto come sta. Esempio:

---

```
1 library(ggplot2)
2 data(mpg)
3 qplot(displ, hwy, data = mpg)
```

---

## 4.5 BASE

Il motore grafico di “base” per creare grafici è contenuto all'interno di due pacchetti che sono `graphics` (che contiene le funzioni di base per creare i grafici) e `grDevices` (che contiene il codice per generare PDF, postscript, ma anche per graphic devices come Quartz, per X11 ed altri).

Il motore `lattice` usa il pacchetto `lattice` che contiene le funzioni per generare grafici che sono indipendenti dal sistema “base” infatti esso si serve del pacchetto `grid` che implementa un sistema base alternativo a “base”.

Prima di costruire un grafico è importante fare alcune considerazioni sul “dove”, inteso come supporto, il grafico andrà ad essere rappresentato. Sarà un grafico mostrato solo a

schermo? oppure in una presentazione? o stampato su carta in una relazione? mi serve poterlo ridimensionare? c'è un grande ammontare di dati oppure solo pochi punti?

Con il sistema base per creare grafici a due dimensioni ci sono due fasi sostanzialmente. La prima è quella di inizializzare un grafico (sostanzialmente se non c'è un device che mi riproduce il grafico alcune funzioni come `plot` o `hist` fanno sì che una finestra con il grafico appaia) la seconda è quella di aggiungere elementi ad un grafico esistente.

Ci sono diversi parametri che sono unici per tutte le funzioni di “base”. L'elenco seguente, senza presunzione di completezza, ne riporta i principali:

**pch** i simboli da utilizzare (di default sono cerchi non colorati)

**lty** tipo di linea da utilizzare piena, tratteggiata, puntini eccetera (di default è una linea piena)

**lwd** spessore della linea

**col** specifica il colore; la funzione `colors()` restituisce i nomi da poter usare nello spazio nomi, oppure è possibile specificare il colore usando la codifica hex.

**xlab** etichetta asse x

**ylab** etichetta asse y

La funzione **par** serve per specificare parametri globali, quindi parametri che saranno a tutti i grafici che nella sessione di lavoro saranno generati. L'elenco seguente ne riporta alcuni parametri:

**las** orientamento

**bg** sfondo

**mar** dimensioni dei margini

**oma** margine esterno dimensioni (default è zero)

**mfrow** e **mfcol** numero di grafici per riga e colonna; entrambi utili quando si hanno grafici multipli come ad esempio una matrice di grafici, servono per controllare quanti grafici ci sono per riga e o colonna

Per ognuno di questi parametri è possibile visionare il valore di default con il seguente codice che non necessita spiegazioni.

---

```
1 par("<nome_parametro>")
```

---

Alcune funzioni base di “base”:

**plot** riproduce come output uno scattergramma ma, in base al tipo di dati che deve rappresentare, anche altri tipi di grafici

**lines** aggiunge linee ad un grafico



**points** aggiunge punti

**text** aggiunge etichette di testo usando specifiche coordinate

**title** aggiunge annotazioni agli assi, etichette assi, sottotitoli eccetera

**mtext** aggiunge testo ai margini (text margin)

**axis** modifica assi (etichette e spessore)

Nelle pagine successive saranno proposti alcuni esempi.

---

```
1 library(datasets)
2 with(airquality, plot(Wind, Ozone))
3 title(main = "Ozone and Wind in New York City")
```

---

Con questi invece aggiungo punti colorati al grafico precedente:

---

```
1 with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in
  New York City"))
2 with(subset(airquality, Month == 5), points(Wind, Ozone, col =
  "blue"))
```

---

con questo aggiungo una linea di regressione:

---

```
1 with(airquality, plot(Wind, Ozone, main = "Ozone and Wind in
  New York City",
2                               pch = 20))
3 model <- lm(Ozone ~ Wind, airquality)
4 abline(model, lwd = 2)
```

---

aggiungo due grafici nello stesso grafico con il par(mfrow = c(1, 2))

---

```
1 par(mfrow = c(1, 2))
2 with(airquality, {
3   plot(Wind, Ozone, main = "Ozone and Wind")
4   plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
5 })
```

---

Con questo invece ne creo tre, e con mtext creo una etichetta generale per tutte e tre i grafici:

---

```
1 par(mfrow = c(1, 3), mar = c(4, 4, 2, 1), oma = c(0, 0, 2, 0))
2 with(airquality, {
3   plot(Wind, Ozone, main = "Ozone and Wind")
4   plot(Solar.R, Ozone, main = "Ozone and Solar Radiation")
5   plot(Temp, Ozone, main = "Ozone and Temperature")
6   mtext("Ozone and Weather in New York City", outer = TRUE)
7 })
```

---

## 4.6 Graphic Device

Un graphic devices è una entità, sia fisica (monitor) che virtuale (un file) dove si può far apparire un grafico.

Ogni volta che R disegna un grafico sulla base di nostre indicazioni deve necessariamente inviarlo ad un graphic device.

Lo schermo fisico del computer ad esempio è un graphic device che in base al sistema operativo utilizzato assume diversi nomi e conseguentemente può essere invocato con diverse funzioni. Sui sistemi Mac OS si attiva con la funzione `quartz` mentre sui sistemi Windows è presente l'apposita funzione `windows`, sui sistemi Linux\Unix infine è presente la funzione `x11` su Linux/Unix

È importante considerare per prima cosa dove il grafico andrà a finire, in altre parole quale è il graphic device che lo riceve? Per capire ciò basta è `?Devices`.

Il primo modo per creare un grafico è creare una funzione di plotting come `plot` `xyplot` o `qplot` ecc. Il grafico appare e poi lo puoi annotare ecc.

Un altro modo è quello di richiamare in modo esplicito un graphic device.

I graphic file device sono di diversi formati organizzati in due classi: vettoriali e bitmap. Vettoriali:pdf (buono ma con molti punti può essere lento), svg basato su xml molto utile per il web supporta anche animazioni e cose interattive, win.metafile solo per windows, postscript un formato un poco più vecchio

bitmap: png ha una compressione senza perdita come il GIF, jpeg per le linee non va bene perché si può vedere aliasing , tiff bmp

È possibile avere diversi graphic devices contemporaneamente ma si può disegnare solo uno alla volta ed in quello attivo. Quello attivo si può sapere con la funzione `dev.cur()`

Ogni graphic devices è un numero intero tra 2 ecc. si può cambiare con `dev.set(<numero intero>)` dove intero è il numero associato a quel graphic device.

`dev.copy` permette di copiare un grafico da un devices (spesso lo schermo), verso un altro

`dev.copy2pdf` lo fa come prima ma direttamente verso un pdf.

Copiare non è un lavoro perfetto, spesso ci sono degli aggiustamenti ecc.

---

```

1 library(datasets)
2 with(faithful , plot(eruptions , waiting)) ## Create plot on
   screen device
3 title(main = "Old Faithful Geyser data") ## Add a main title
4 dev.copy(png, file = "geyserplot.png") ## Copy my plot to a
   PNG file
5 dev.off()
```

---

# Aggiornamenti delle edizioni