



Appunti di Programmazione

Sabato De Maio

Indice

1 Le Basi 1

- 1.1 Prime Cose Da Fare 1
- 1.2 Prime Cose da Sapere 2
 - 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 5
 - 1.3.1 Vettori 5
 - 1.3.2 Avvertimenti 7
 - 1.3.3 Classi 8
- 1.4 Working Directory 8
- 1.5 List 8
- 1.6 Matrici 9
- 1.7 Data Frame 10
- 1.8 Names 10
- 1.9 Factors 11

2 Funzioni della famiglia apply 13

- 2.1 Scopo 13
- 2.2 apply 13
- 2.3 lapply e sapply 15
- 2.4 tapply 16
 - 2.4.1 split 17
 - 2.4.2 by 17
 - 2.4.3 Aggregate 20
- 2.5 mapply 22
 - 2.5.1 Introduzione alle anonymous functions 23
 - 2.5.2 mapply: esempi con le anonymous functions 23

Aggiornamenti 25

Bibliografia 27

Elenco delle tabelle

Elenco delle figure

1

Le Basi

1.1 Prime Cose Da Fare

La prima cosa da fare è innanzitutto installare R. Il software è disponibile sul sito ufficiale del progetto R accessibile cliccando [qui](#), inoltre **dopo** aver installato R potete installare RStudio cliccando [qui](#). 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 interrompono la esecuzione del comando dato oppure con degli avvertimenti (warnings) atti a comunicarci 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 programma 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

Esiste anche un altro modo per ottenere questo risultato, si veda il [1] per maggiori dettagli.

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 datasets precaricati 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
5 locale:
6 [1]
   en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
7
8 attached base packages:
9 [1] stats      graphics  grDevices  utils      datasets
   methods   base
10
11 loaded via a namespace (and not attached):
12 [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 Installare e caricare pacchetti

La funzione del paragrafo precedente restituisce informazioni sulla nostra versione di R¹. 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")
```

¹Come il nome in effetti suggerisce.

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 6000 pacchetti presenti su CRAN.

Qualora vogliate verificare il numero di pacchetti attualmente disponibili (ed altre informazioni sugli stessi), il seguente comando restituisce un oggetto matrice con tutti i pacchetti disponibili su CRAN².

```
1 packet <- available.packages()
```

Tornando alla installazione ed al caricamento di pacchetti da notare è 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 pacchetto richiesto. Se necessario, R scaricherà in automatico anche altri pacchetti necessari al corretto funzionamento del pacchetto che avete appena richiesto.

Ci sono anche altre fonti per pacchetti che necessitano una procedura leggermente diversa per l’installazione ma per ora è meglio tralasciare questo aspetto³.

1.2 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 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 gli output, i database, i grafici ed altro 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")
```

²Ce ne sono molti altri su GitHub e quasi 800 su Biocouductor.

³Tra cui Bioconductor o i tanti sviluppati ed ancora in fase beta presenti su GitHub.

1.3 Prime Cose da Sapere

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 directory⁴. 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.

⁴Potrebbe apparire più logico aver creato questo comando con il nome `dir.delete` ma probabilmente gli autori di R erano degli utenti Linux.

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 ragiona R e come sono le “parole” che usa, al fine di poter comunicare con questo software⁵

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

Sostanzialmente ogni cosa in R è un oggetto anche se spesso la definizione di oggetto può essere diversa da chi si trova a dover conoscere in profondità il software per necessità di programmazione a chi invece si trova a doverlo utilizzare. Nelle pagine seguenti le definizioni di oggetti saranno poco ortodosse prediligendo il punto di vista di un mero utilizzatore che non possiede né le capacità per entrare nei dettagli del linguaggio né ha il bisogno di doverlo fare. Per rendersi conto di questa difformità si prenda ad esempio una matrice. Una matrice potrebbe essere considerata come un oggetto a se stante. Ciò potrebbe essere anche indotto dal fatto che l'utilizzo della funzione `class` su di una matrice, ad esempio formata da numeri, produce come output “matrix”. Si provino le funzioni `typeof` e `is.atomic` sulla stessa matrice e si scoprirà che per R un vettore numerico ed una matrice sono lo stesso tipo di oggetto.

Ovviamente sebbene da un punto di vista del linguaggio di programmazione essi siano la stessa cosa, anche solo visivamente appaiono diversi. È a tal proposito che ad ogni tipo di oggetto riconosciuto dal linguaggio e dato dalla funzione `typeof`

1.3 Vettori

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 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`.

⁵R 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.

1.3 The Name of the Game

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-compatibilità⁶.

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
```

Per verificare che sia il primo codice che il secondo producono un vettore⁷ 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
```

⁶Sia 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.

⁷Numerico, ma per ora è meglio tralasciare questo aspetto.

```

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 in 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 Avvertimenti

Nelle pagine seguenti, saranno scientemente commesse alcune inesattezze al fine di rendere più pratica l'esposizione. Una su tutte ad esempio riguarda la facile classificazione di entità in R come "oggetti".

Come molti, R è un linguaggio di programmazione orientato agli oggetti e si rende quindi necessario definire cosa sono gli oggetti. In questa sede si definiscano oggetti di R entità con le quali il linguaggio entra in relazione. Ci sono diversi tipi di oggetti che probabilmente non si vedranno mai mentre ci sono altri "oggetti" che di fatto tali non sono ma sono chiamati tali nel linguaggio comune.

Questa apparente confusione potrebbe essere nata da alcuni fattori. In primo luogo R è un progetto in continua evoluzione e ad oggi, marzo 2015, un software di più di venti anni. A sua volta si basa su S che è molto più vecchio. Da ciò deriva che molte caratteristiche attuali, come l'essere orientato agli oggetti, sono state introdotte successivamente e quindi il sistema se da un lato si è adattato a nuovi sviluppi, dall'altro per motivi di retro-compatibilità ogni nuova versione si porta dietro caratteristiche delle prime versioni di fatto obsolete.

```

1 > typeof(get("for"))
2 [1] "special"
3 > typeof(get("break"))
4 [1] "special"

```

1.5 Working Directory

1.3 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. La classe è alla base delle funzionalità definite “generiche” di R. 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)
- numerico (numeri reali)
- integer (che in realtà non sono numeri interi, ma “livelli”), utile per descrivere “mutabili” e non “variabili”
- numeri complessi
- logical (vero/falso)

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.

1.8 Data Frame

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  
3 [1] PD FI M5S LN NCD  
4 Levels: FI LN M5S NCD PD
```


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 appartengo 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>)
```

2.2 apply

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`¹, 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

¹È una funzione definita internamente in C.

2.4 tapply

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
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 oggetto².

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
```

²Ma anche di altri in effetti.


```

6 2      M    88000
7 3      F    32450
8 4      M    76500
9 5      F   123000
10 6      F    45650
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 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
8
9 $M
10  genere reddito
11 1      M    55000
12 2      M    88000
13 4      M    76500

```

2.4 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

2.4 tapply

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
5
6 $versicolor
7   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
8   2.000   2.525   2.800   2.770   3.000   3.400
9
10 $virginica
11   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
12   2.200   2.800   3.000   2.974   3.175   3.800
13
14 > cb
15 iris$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
4   Petal.Width      Species
5   Min.    :4.300    Min.    :2.300    Min.    :1.000    Min.
6   :0.100    setosa    :50
7   1st Qu.:4.800    1st Qu.:3.200    1st Qu.:1.400    1st
8   Qu.:0.200    versicolor: 0
9   Median :5.000    Median :3.400    Median :1.500    Median
10  :0.200    virginica : 0
11  Mean    :5.006    Mean    :3.428    Mean    :1.462    Mean
12  :0.246
13  3rd Qu.:5.200    3rd Qu.:3.675    3rd Qu.:1.575    3rd
14  Qu.:0.300
15  Max.    :5.800    Max.    :4.400    Max.    :1.900    Max.
16  :0.600
17 -----
18 iris$Species: versicolor
19   Sepal.Length    Sepal.Width    Petal.Length    Petal.Width
20   Species
21   Min.    :4.900    Min.    :2.000    Min.    :3.00    Min.
22   :1.000    setosa    : 0
23   1st Qu.:5.600    1st Qu.:2.525    1st Qu.:4.00    1st
24   Qu.:1.200    versicolor:50
25   Median :5.900    Median :2.800    Median :4.35    Median
26   :1.300    virginica : 0

```

2.4 tapply

```
16 Mean      :5.936      Mean      :2.770      Mean      :4.26      Mean      :1.326
17 3rd Qu.:6.300      3rd Qu.:3.000      3rd Qu.:4.60      3rd Qu.:1.500
18 Max.      :7.000      Max.      :3.400      Max.      :5.10      Max.      :1.800
19 -----
20 iris$Species: virginica
21 Sepal.Length Sepal.Width Petal.Length
22 Petal.Width Species
23 Min.      :4.900      Min.      :2.200      Min.      :4.500      Min.
24 :1.400      setosa      : 0
25 1st Qu.:6.225      1st Qu.:2.800      1st Qu.:5.100      1st
26 Qu.:1.800      versicolor: 0
27 Median :6.500      Median :3.000      Median :5.550      Median
28 :2.000      virginica :50
29 Mean      :6.588      Mean      :2.974      Mean      :5.552      Mean
30 :2.026
31 3rd Qu.:6.900      3rd Qu.:3.175      3rd Qu.:5.875      3rd
32 Qu.:2.300
33 Max.      :7.900      Max.      :3.800      Max.      :6.900      Max.
34 :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 desiderato³.

2.4 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 `tapply` 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`⁴

³Si 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.

⁴Si usi `data()` per vedere tutti dataset preinstallati in R.

```

1 > prova <- tapply(iris$Sepal.Length, iris$Species, mean)
2 > prova
3   setosa versicolor  virginica
4   5.006    5.936    6.588
5 >
6 >
7 > prova1 <- aggregate(iris$Sepal.Length, list(iris$Species),
8   mean)
9 > prova1
10   Group.1      x
11 1   setosa 5.006
12 2 versicolor 5.936
13 3  virginica 6.588
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,
2   iris$Petal.Width), mean)
3 > head(prova5)
4   Group.1 Group.2      x
5 1   setosa    0.1 4.820000

```

2.5 mapply

```
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 paragrafi⁵.
- 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 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 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>`.

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

2.5 mapply

```
1 > mapply( function(x,y,z){ rep(y, x) * z } , p1, p2, p3)
2 [[1]]
3 [1] -10
4
5 [[2]]
6 [1] 11 11
7
8 [[3]]
9 [1] -12 -12 -12
10
11 [[4]]
12 [1] 26 26 26 26
13
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.

Aggiornamenti delle edizioni

Bibliografia

- [1] R Core Team. *R Installation and Administration*. Version 3.1.3 (2015-03-09). 2015.