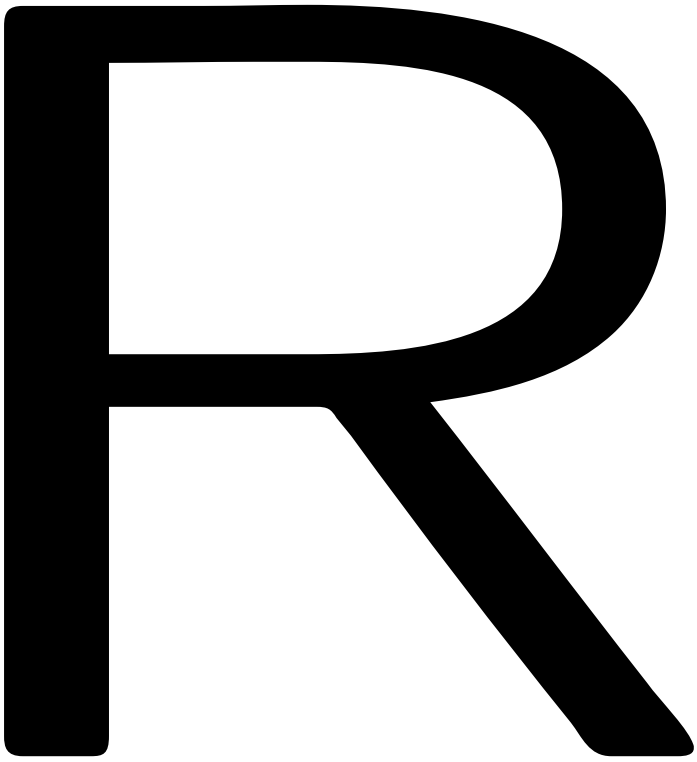

Appunti di Programmazione

Sabato De Maio

Indice

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.

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

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

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

loaded via a namespace (and not attached):
[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[1]. 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:

```
> 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.

```
> install.packages("datasets")
```

Grazie a questo comando il "vostro" R si connetterà 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.

[1] 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.

```
> getwd()
[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:

```
> 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.

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

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

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

[3] R è si 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, assegnale 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[4].

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.

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

[4]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[5] usiamo il seguente codice mediante il quale, sostanzialmente, chiediamo ad R "l'argomento della funzione is.atomic è un atomic vector?".

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

```
> second <- c(1,3,5,6,9)
> second
[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 concetti 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 da 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)
- 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)

[5] Numerico, ma per ora è meglio tralasciare questo aspetto.

1.4 Working Directory

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

```
> getwd()
[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:

```
> impiegato <- list(nome = "Mario", cognome = "Rossi",
stipendio = 100000, sindacato = TRUE)
> impiegato
$nome
[1] "Mario"

$cognome
[1] "Rossi"

$stipendio
[1] 1e+05

$sindacato
[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.

```
> names(impiegato)
[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.

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

```
> impiegato1
[[1]]
[1] "Mario"

[[2]]
[1] "Rossi"

[[3]]
[1] 1e+05

[[4]]
[1] TRUE

> names(impiegato1)
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 una 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 cone 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 contente i nomi, o anche solo il singolo nome (in questo caso ovviamente è possibile omettere la funzione c( ) necessaria per creare vettori).

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

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

```
> names(impiegato1)
NULL
> impiegato1
[[1]]
[1] "Mario"

[[2]]
[1] "Rossi"

[[3]]
[1] 1e+05

[[4]]
[1] TRUE

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

$cognome
[1] "Rossi"

$stipendio
[1] 1e+05

$sindacato
[1] TRUE
```

1.9 Factors

Factors sono usati per rappresentare dati a categorie (categorial 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.

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

```
[1] PD   FI   M5S  LN   NCD
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 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>)
```

Dove con array abbiamo un oggetto ti 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.

```
> valori1 <- c(1:5)
> valori1
[1] 1 2 3 4 5
> valori2 <- c(10:15)
> valori2
[1] 10 11 12 13 14
>
> dataF <- data.frame(valori1, valori2)
> dataF
  valori1 valori2
1       1      10
2       2      11
3       3      12
4       4      13
5       5      14
> apply(dataF, 2, sd)
  valori1   valori2
1.581139 1.581139
```

Alcune considerazioni sull'esempio precedente:

- la funzione apply si applica a data frame e array ed ha come output una lvettore 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:

```
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.

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

$valori2
[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.

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

[1] È una funzione definita internamente in C.

```
3 1.581139 1.581139
```

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[2].

Immaginiamo di avere un vettore numerico e di voler applicare una funzione a sottoinsiemi di questo vettore basati su alcuni 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
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.

[2] Ma anche di altri in effetti.

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.

```
> d1 <- split(d, d$genere)
> d1
$F
  genere reddito
3      F   32450
5      F  123000
6      F   45650

$M
  genere reddito
1      M   55000
2      M   88000
4      M   76500
```

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 funziona 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.

```
> p1 <- c(1:5)
> p2 <- c(10:14)
> mapply( "+" , p1, p2)
[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.
- 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.

```
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

```
> p1 <- c(1:5)
> p2 <- c(10:14)
> 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.

```
> mapply( function(x, y, z) (x*y)/z , p1, p2, p3)
[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>.

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

[[2]]
[1] 11 11

[[3]]
[1] -12 -12 -12

[[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.

Aggiornamenti delle edizioni