Sintesi di GNU R

18 maggio 2023

Indice

1 Introduzione **9**
1.1 Modalità di utilizzo . . . 9
1.2 Sintassi di base . . . 9
1.3 Librerie e pacchetti . . . 10
1.3.1 Gestione pacchetti . . . 10
1.3.1.1 Installazione . . . 10
1.3.1.2 Listing . . . 12
1.3.1.3 Aggiornamento . . . 12
1.3.1.4 Rimozione . . . 12
1.3.2 Caricamento dei pacchetti . . . 12
1.4 Environment e lexical scope . . . 12
1.4.1 Environments e search path . . . 13
1.4.2 Modifica del search path . . . 14
1.4.2.1 Aggiunta di elementi . . . 14
1.4.2.2 Rimozione di elementi . . . 15
1.4.3 Operazioni su oggetti in environments . . . 15
1.4.3.1 Creazione/assegnazione . . . 15
1.4.3.2 Listing . . . 15
1.4.3.3 Stampa . . . 16
1.4.3.4 Ricerca . . . 16
1.4.3.5 Test esistenza . . . 17
1.4.3.6 Rimozione . . . 17
1.4.3.7 Nomi presenti in più environments . . . 17
1.4.3.8 Ottenere un oggetto da un dato environment . . . 18
1.5 Interfacce con l'esterno . . . 18
1.5.1 Importare/esportare dati . . . 18
1.5.1.1 `read.table` . . . 19
1.5.1.2 `write.table` . . . 20
1.5.1.3 `dput`, `dump`, `dget` e `source` . . . 20
1.5.1.4 `saveRDS`, `save` e `save.image` . . . 22
1.5.1.5 `readRDS` e `load` . . . 22
1.5.2 Log, esecuzione script . . . 22
1.5.2.1 Log . . . 22
1.5.2.2 Script . . . 23
1.5.3 Connessioni . . . 23
1.5.4 Interfaccia col sistema operativo, directory e file . . . 25
1.6 Opzioni . . . 26
1.7 Guida in linea aiuto ecc. . . . 26

1.7.1 Sintassi . . . 26
1.7.2 Ricerca funzioni . . . 26
1.7.3 Esempi, vignette, help dei pacchetti . . . 27
1.7.4 Ricerca di errori su Internet . . . 27

2 Gli oggetti di R **29**
2.1 Classificazione degli oggetti . . . 29
2.1.1 Strutture atomiche . . . 29
2.1.1.1 Tipi di dato delle strutture atomiche . . . 30
2.1.1.2 Costanti speciali per le strutture atomiche . . . 31
2.1.1.3 Coercizione . . . 31
2.1.2 Strutture non atomiche . . . 32
2.1.3 Visualizzare una struttura dati con `str` . . . 33
2.2 Attributi degli oggetti . . . 34
2.2.1 Accesso agli attributi . . . 34
2.2.2 Attributi principali . . . 35
2.2.2.1 Classe dell'oggetto . . . 35
2.3 Vettori atomici . . . 36
2.3.1 Creazione . . . 36
2.3.2 Factor . . . 37
2.3.3 Vettorizzazione e recycling . . . 39
2.4 Liste . . . 39
2.5 Matrici ed Array . . . 40
2.5.1 Creazione mediante le funzioni `matrix` e `array` . . . 40
2.5.2 Creazione fornendo l'attributo `dim` ad una lista . . . 42
2.6 Data frames . . . 43
2.6.1 Creazione e modifica . . . 43

3 Indici e subsetting **45**
3.1 Introduzione . . . 45
3.2 Utilizzo degli indici . . . 46
3.2.1 Vettori . . . 46
3.2.2 Liste . . . 48
3.2.3 Matrici ed array . . . 49
3.2.4 Data frame . . . 51
3.2.5 S3 objects . . . 52
3.3 Simplifying vs preserving subsetting . . . 52
3.4 Subsetting e assegnamento . . . 54

4 Controllo del flusso **57**
4.1 Esecuzione condizionale . . . 57
4.1.1 `if` . . . 57
4.1.2 `switch` . . . 58
4.2 Looping . . . 59
4.2.1 `for` . . . 59
4.2.2 `while` . . . 60
4.2.3 `repeat` . . . 61
4.2.4 `break` . . . 61
4.2.5 `next` . . . 61
4.2.6 `return` . . . 62

5 Funzioni **63**
5.1 Introduzione . . . 63
5.2 Definizione . . . 64
5.2.1 Sintassi . . . 64
5.2.2 Argomenti . . . 64
5.2.2.1 L'argomento `...` e la sua gestione . . . 65
5.2.3 Corpo . . . 66
5.2.4 Ritorno della funzione . . . 66
5.2.5 L'utilizzo di `on.exit` . . . 67
5.3 Chiamata e valutazione delle funzioni . . . 68
5.3.1 Matching degli argomenti . . . 68
5.3.2 Lazy evaluation e promise . . . 68
5.4 Other tricks, good practices and misc things . . . 70
5.4.1 Infix functions . . . 70
5.4.2 Replacement functions . . . 70
5.4.3 Funzioni vettorizzate . . . 71
5.4.4 L'uso di `match.arg` . . . 72
5.4.5 `NULL` e `getOption` per parametri di default . . . 72

6 Scoping, environments, binding **75**
6.1 Scoping . . . 75
6.2 Environments . . . 76
6.2.1 Introduzione . . . 77
6.2.2 Operazioni comuni . . . 78
6.2.3 Chiamata di funzioni ed environment . . . 79
6.2.4 Reference semantics ed utilizzi connessi . . . 82
6.3 Binding . . . 83

7 Defensive programming, condition handling, debugging **85**
7.1 Programmazione difensiva . . . 85
7.2 Conditions e conditions handling . . . 85
7.2.1 Conditions . . . 86
7.2.2 Condition handling . . . 87
7.3 Debugging . . . 88
7.3.1 Funzioni utili . . . 88
7.3.2 Ispezionare lo stack delle chiamate di funzioni . . . 91
7.3.3 Debug per `warning` e `message` . . . 93

8 Functional Programming **95**
8.1 Elementi costitutivi . . . 95
8.1.1 Anonymous functions . . . 95
8.1.2 Closures . . . 96
8.1.2.1 Function factories . . . 96
8.1.2.2 Mutable states . . . 97
8.1.3 Liste di funzioni . . . 98
8.2 Functionals . . . 99
8.2.1 `lapply` . . . 99
8.2.2 `vapply` (e `sapply`) . . . 100
8.2.3 `Map` . . . 101
8.2.4 `apply` . . . 101

8.2.5 `tapply` . . . 102
8.2.6 Collassare liste con `Reduce` . . . 103
8.2.7 Funzioni per la gestione di predicati . . . 104
8.2.8 Functional e matematica . . . 104
8.3 Function operators . . . 105
8.3.1 Behavioural FOs . . . 106
8.3.2 Output FOs . . . 107
8.3.3 Input FOs . . . 108
8.3.4 Combining FOs . . . 109

9 Computing on the language **111**
9.1 Espressioni . . . 111
9.1.1 Constants . . . 113
9.1.2 Names . . . 114
9.1.3 Calls . . . 114
9.1.4 Pairlists . . . 116
9.2 Non standard evaluation . . . 116
9.2.1 Catturare le espressioni dei parametri . . . 116
9.2.2 NSE in `subset` . . . 117
9.2.3 Chiamare funzioni NSE da altre funzioni . . . 119
9.2.4 Catturare ... non valutato . . . 121

10 Programmazione ad oggetti **123**
10.1 Concetti generali . . . 123
10.2 S3 . . . 124
10.2.1 Classi . . . 124
10.2.2 Funzioni generiche . . . 125
10.2.3 Metodi . . . 128
10.3 S4 . . . 130
10.3.1 Funzioni generiche e metodi in S4 . . . 130
10.4 Esaminare il codice di funzioni S4 . . . 131
10.4.1 Definizione di classi . . . 132
10.4.2 Definizione di metodi . . . 132
10.4.3 Instanziazione di oggetti . . . 133

11 Performance **135**
11.1 Esecuzione parallela . . . 135
11.1.1 Setup e chiusura . . . 135
11.1.2 Funzioni utili per l'esecuzione parallela . . . 136
11.2 Analisi automatica del codice . . . 136
11.3 Timing . . . 137
11.4 Code profiling . . . 137
11.5 Interfaccia con il linguaggio C . . . 139
11.5.1 Codice compilato in pacchetti . . . 139
11.5.1.1 Setup del pacchetto . . . 139
11.5.1.2 L'utilizzo di `.C` . . . 139
11.5.1.3 Api di R . . . 141
11.5.1.4 L'utilizzo di `.Call` . . . 142
11.5.1.5 Aspetti salienti di programmazione interna . . . 142
11.5.2 Uso di `inline` . . . 147

12 Data management **149**
12.1 Sorting . . . 149
12.2 Sampling . . . 151
12.3 Splitting . . . 152
12.4 Subset . . . 152
12.5 Merging . . . 153
12.6 Binding . . . 154
12.7 Suddivisione in classi . . . 156
12.8 Stringhe . . . 158
12.8.1 Manipolazione di stringhe e vettori di caratteri . . . 158
12.8.2 Esecuzione di comandi contenuti in stringhe . . . 159
12.8.3 Espressioni regolari ed elaborazione del testo . . . 159
12.8.3.1 `grep` e `grepl` . . . 160
12.8.4 `regexpr` e `gregexpr` . . . 161
12.8.5 `sub` e `gsub` . . . 162
12.8.6 `regexec` . . . 162
12.8.7 Wildcards . . . 164
12.8.8 Matching fuzzy . . . 164
12.9 Date e tempi (*timestamp*) . . . 164
12.9.1 Funzioni utili . . . 165
12.9.2 Creare date . . . 166
12.9.2.1 Date/Tempi a partire da stringhe . . . 166
12.9.2.2 Data/Tempo da un vettore di giorni/secondi . . . 167
12.9.3 Date/Tempi a partire da molteplici vettori . . . 167
12.9.4 Estrarre informazioni dalle date . . . 168
12.9.5 Elaborazioni su date . . . 168
12.10Eliminazione record doppi . . . 168
12.11Reshape . . . 169

13 Grafici **171**
13.1 Introduzione . . . 171
13.1.1 Il sistema grafico di R . . . 171
13.1.2 Funzioni grafiche e procedura di creazione dell'immagine . . . 171
13.1.3 Una guida alla scelta tra `graphics` e `grid` . . . 172
13.2 I devices grafici . . . 172
13.2.1 Attivazione e chiusura di un device . . . 173
13.2.2 Gestione di più device . . . 173
13.2.3 Riutilizzo del contenuto di un device grafico . . . 174
13.3 L'uso del pacchetto `graphics` . . . 175
13.3.1 Opzioni . . . 176
13.3.2 Funzioni di basso livello per la modifica del grafico . . . 178
13.3.3 Altre funzioni utili . . . 178
13.3.4 Come ottenere grafici multipli . . . 179
13.3.5 Come inserire formule matematiche . . . 179
13.4 L'uso del pacchetto `ggplot2` . . . 180
13.4.1 Costruzione base . . . 180
13.4.2 Layer molteplici . . . 183
13.4.3 Faceting . . . 185
13.4.4 Trasformazioni . . . 187
13.4.5 Coordinate . . . 189

13.5 Colori 191
13.6 Note di Teoria dei Grafici 195

14 Misc e sandbox **199**
14.1 Interfaccia con database 199
14.1.1 Access 199
14.1.1.1 Interagire con Access ed altri DMBS 199
14.2 Effettuare confronti 200
14.3 Inserire un elemento (vettore) in una certa posizione di una lista (`data.frame`) 200
14.4 Eliminare i livelli non utili dei factor 201
14.5 Modificare le singole variabili di un dataset serialmente 202
14.6 Creare barre di avanzamento 202

Capitolo 1

Introduzione

1.1 Modalità di utilizzo

R può esser utilizzato in modalità

- **interattiva**, avviandolo mediante il comando `R` al prompt;
- in modalità **batch** mediante

  ```
  R CMD BATCH file.input file.output
  ```

  Se `file.output` non è specificato, l'output verrà rediretto a un file con estensione `.Rout`;
- come **linguaggio di scripting**, creando un file avente intestazione

  ```
  #!/usr/bin/env Rscript
  ```

  e a seguire le istruzioni (come se fosse un file `.R` normale). Dandogli i permessi di esecuzione e ponendolo nel path dei binari sarà possibile eseguirlo comandando il nome del file.

1.2 Sintassi di base

I comandi impartibili ad R possono essere separati da un punto e virgola o andando a capo; i commenti si fanno con `#`. I comandi impartibili ad R si suddividono in:

- *espressioni*: valutate e il risultato stampato a video (ma non salvato):

  ```
  1+2

  ## [1] 3
  ```

  Gli operatori servono per costruire le espressioni del linguaggio. Possono esser *unari* o *binari*, a seconda che richiedano uno o due oggetti nell'espressione che li interessa. Un elenco di operatori di R si trova in tabella 1.1.

- *assegnazioni*: salva il valore di una espressione valutata in una variabile, senza stampar il primo a video[1]

```
a <- 1+2
```

La sintassi prevede un *identificatore* a sinistra, il valore/espressione a destra e una freccia in mezzo rivolta a sinistra in mezzo. L'identificatore è una sequenza di lettere/numeri:

  - non può iniziare con una cifra, con `_` o con un punto seguito da una cifra;
  - gli identificatori che iniziano con un punto sono nascosti e non sono rilevati da `ls()`;
  - non può coincidere con una delle parole *riservate* del linguaggio

```
if else repeat while function for in next break
TRUE FALSE NULL Inf NaN
NA NA_integer_ NA_real_ NA_complex_ NA_character_ ...  ..1 ..2
```

  - essendo R *case sensitive*, gli identificatori `gatto` e `Gatto` indicano oggetti differenti.

1.3 Librerie e pacchetti

Una libreria è una directory all'interno della quale sono presenti i pacchetti installati; la libreria principale è `R_HOME/library` (su Linux `/usr/lib/R/library`). La funzione `.libPaths()`, se invocata senza argomenti restituisce le librerie presenti sul sistema.

1.3.1 Gestione pacchetti

I pacchetti possono esser distribuiti in forma di sorgente o in forma compilata; installare i primi richiede tutta una serie di tool. Nell'installazione è generalmente necessario specificare (anche implicitamente) la libreria che accoglierà i pacchetti (nel caso in cui ci siano più librerie sul sistema.

1.3.1.1 Installazione

Per installare un pacchetto si utilizzi in primis il sistema di pacchettizzazione del sistema operativo: se vi sono pacchetti di interesse non ancora pacchettizzati, si avvii R con i privilegi di root, dopodiché:

```
install.packages("nome_pacchetto", dependencies = TRUE)
```

Come primo argomento si può specificare anche un vettore di char che includano i nomi (case sensitive) dei pacchetti da installare; questo comando basterà per scaricare dal CRAN il pacchetto (eventualmente le dipendenze) ed installarlo. Se non si specifica l'argomento `lib` inerente la scelta della libreria di installazione verrà utilizzata la prima directory di `.libPaths()`.

[1] Se si vuole stampare anche il valore a video, occorre porre l'assegnazione fra parentesi tonde.

`-`	sottrazione, unario o binario
`+`	addizione, unario o binario
`*`	moltiplicazione, binario
`/`	divisione, binario
`^`	potenza, binario
`%/%`	divisione intera, binario
`%%`	modulo (resto divisione), binario
`<`	minore, binario
`>`	maggiore, binario
`==`	uguale a, binario
`>=`	maggiore o uguale, binario
`<=`	minore o uguale, binario
`&`	`and`, binario, *vettorizzato* (confronta tutti gli elementi di un vettore)
`&&`	`and`, binario, *non vettorizzato* (confronta solamente i primi elementi di un vettore)
`!`	`not` unario
`|`	`or`, binario, *vettorizzato*
`||`	`or`, binario, *non vettorizzato*
`<-`	Assegnamento, binario
`$`	Subset di lista, binario
`~`	tilde, utilizzata nelle formule, può esser unaria o binaria
`?`	help
`:`	sequenza, binario (nelle formule: interazione)
`%x%`	operatore binario speciale, x può esser rimpiazzato da qualsiasi nome valido
`%*%`	prodotto riga per colonna (matrici), binario
`%in%`	Operatore di *matching*, binario (nelle formule dei modelli: *nesting*)

Tabella 1.1: Operatori di R

1.3.1.2 Listing

Per vedere quali pacchetti sono installati:

```
library()
```

1.3.1.3 Aggiornamento

Per

- controllare lo status dei pacchetti installati:

  ```
  packageStatus()
  ```

- aggiornare tutti i pacchetti installati:

  ```
  update.packages()
  ```

1.3.1.4 Rimozione

Per rimuovere un pacchetto o agire mediante il sistema di pacchettizzazione del sistema operativo o mediante `remove.packages`:

```
remove.packages("nome_pacchetto")
```

Il primo argomento è un vettore di nomi di pacchetti; si può specificare il parametro `lib` (libreria dove si trova il pacchetto).

1.3.2 Caricamento dei pacchetti

Per caricare un determinato pacchetto utilizzare `library` o `require`. Entrambe caricano il pacchetto, ma *require* è disegnata per l'utilizzo all'interno di altre funzioni: ritorna `FALSE` e da un `Warning` se il pacchetto non esiste, mentre `library` da `Error`.

```
library(MASS)
# code using MASS stuff

if (require(MASS)){
    # code using MASS stuff
}
```

1.4 Environment e lexical scope

In ogni linguaggio di programmazione è necessario in runtime che avvenga un processo per ottenere il valore associato ad un dato identificatore.
Ad esempio, se chiediamo al computer di valutare `x+5`, o di eseguire `mean(3,4)`

vi deve essere un meccanismo per determinare il valore rappresentato da `x`, o quale codice usare per `mean`.
Le *scoping rules* di un linguaggio sono l'insieme di regole utilizzate per ottenere un generico valore (funzione, dataset, variabile ecc) a partire dal suo identificatore.

1.4.1 Environments e search path

Quando R cerca un oggetto (un dataset, una funzione o che altro), tale ricerca avviene in una serie di *environments*:

- un environment è una collezione di coppie (simbolo, valore). Ad esempio `x` è un simbolo e `3.14` potrebbe essere il suo valore. Il global environment, o *workspace*, è un set di coppie simbolo-valore, derivanti dalle assegnazioni dell'utente. Allo stesso modo ogni pacchetto ha un namespace che assomiglia concettualmente ad un environment (con coppie simboli-valori);
- ogni environment ha un "parent" (genitore) environment; è altresì possibile che un environment abbia più environment "figli";
- l'unico environment che non ha un genitore è l'environment vuoto.

Quando si lavora a linea di comando, R cerca nella serie di environment che formano il *search path*, visualizzabile mediante il comando `search`

```
search()

##  [1] ".GlobalEnv"        "package:knitr"     "tex2pdf"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "package:methods"
## [10] "Autoloads"         "package:base"
```

Pertanto cerca in sequenza:

1. inizialmente tra gli oggetti memorizzati nel workspace, o *global environment*, identificato da `.GlobalEnv`;
2. nelle librerie caricate, nell'ordine specificato da `search`.

Il global environment (o user workspace) è sempre il primo elemento del search path e il pacchetto `base` è sempre l'ultimo. Appena trova un identificatore con un nome uguale a quello richiesto, R otterrà il valore di quel simbolo e lo restituirà[2]. Pertanto se definiamo una nostra funzione con lo stesso nome di una (ad esempio del pacchetto `base`, nel seguito andiamo ad usare quella:

```
mean <- function(x) {x+x}  #wrong
mean(1)

## [1] 2
```

[2]Va detto che vi sono namespace separati per funzioni e per non funzioni, quindi è possibile avere un oggetto di nome `c` e al contempo una funzione che si chiami `c`; tuttavia non è buona prassi sfruttare ciò.

1.4.2 Modifica del search path

L'ordine dei pacchetti nel search path è rilevante ai fini della ricerca; l'utente può scegliere con quale ordine vengono caricati i pacchetti.

1.4.2.1 Aggiunta di elementi

Si utilizzano `library`, `require` e `attach`. In tutti i casi l'oggetto viene posto al secondo posto:

- mediante `library` (o `require`) si carica un pacchetto

```
search()

##  [1] ".GlobalEnv"        "package:knitr"     "tex2pdf"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "package:methods"
## [10] "Autoloads"         "package:base"

library(lbmisc)
search()

##  [1] ".GlobalEnv"        "package:lbmisc"    "package:knitr"
##  [4] "tex2pdf"           "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"
```

- mediante `attach` possono esser aggiunti al search path gli oggetti contenuti in altri oggetti (liste, data.frame, environments). `attach` effettua una copia nel search path; se l'oggetto originale cambia, la copia non sarà toccata in seguito alla variazione

```
adf <- data.frame("x" = 1:3)
attach(adf)
search()

##  [1] ".GlobalEnv"        "adf"               "package:lbmisc"
##  [4] "package:knitr"     "tex2pdf"           "package:stats"
##  [7] "package:graphics"  "package:grDevices" "package:utils"
## [10] "package:datasets"  "package:methods"   "Autoloads"
## [13] "package:base"

x               # prendo quello nel search path (posto 2), dato che

## [1] 1 2 3

                # in globalenv non vi è un elemento con nome x
adf$x <- 2      # modifico l'originale (nel globalenv)
x               # riprendo quello nel search path

## [1] 1 2 3
```

1.4.2.2 Rimozione di elementi

Si utilizzi `detach` specificando l'environment da rimuovere fra parentesi (di default è il secondo nel search path):

```
detach(adf)
detach(package:lbmisc)
```

1.4.3 Operazioni su oggetti in environments

1.4.3.1 Creazione/assegnazione

Vi sono due modi per creare oggetti:

- gli oggetti creati (mediante assegnazione standard con `<-`) durante una sessione interattiva sono memorizzati nel workspace;
- per salvare in altri environments si utilizza `assign`, specificando per `pos` l'environment in cui effettuare l'assegnazione:

```
attach(adf)
assign('y', letters[1:3], pos = 2)      # secondo elemento del search.path (adf)
assign('z', c("x", "y"),  pos = "adf") # stessa cosa
ls()

## [1] "a"   "adf"

y                                       # R lo trova nell'environment adf

## [1] "a" "b" "c"

z                                       # idem

## [1] "x" "y"
```

1.4.3.2 Listing

Per elencare gli oggetti presenti in un dato environment si usa `ls`:

- senza ulteriori specifiche lista gli oggetti del workspace; per elencare anche gli oggetti nascosti (quelli che iniziano con `.`) specificare `all=TRUE`;

```
a <- 1
.b <- 2
ls()

## [1] "a"   "adf"

ls(all = TRUE)

## [1] ".b"  "a"   "adf"
```

- se come primo argomento (`name`) si specifica il numero progressivo dell'environment nel search path o il suo nome si effettuerà il listing di quell'environment

```
search()

##  [1] ".GlobalEnv"        "adf"               "package:knitr"
##  [4] "tex2pdf"           "package:stats"     "package:graphics"
##  [7] "package:grDevices" "package:utils"     "package:datasets"
## [10] "package:methods"   "Autoloads"         "package:base"

head(ls('package:stats'))

## [1] "acf"        "acf2AR"     "add.scope"  "add1"       "addmargins"
## [6] "aggregate"

head(ls(pos = 2))

## [1] "x" "y" "z"
```

1.4.3.3 Stampa

Generalmente è possibile conoscere i valori assunti da un oggetto chiamandolo (*auto-printing* dei risultati di una espressione) o utilizzando la funzione print (*explicit printing*)

```
a

## [1] 1

print(a) # esempio con oggetti del workspace

## [1] 1
```

Se l'oggetto è di dimensioni consistenti (prime linee non visualizzate) diviene utile usare `page` o, se matrice/data.frame, `View`

```
page(Indometh, meth = "print")
View(Indometh)
```

1.4.3.4 Ricerca

Si utilizzi `find` per trovare l'environment di un dato identificatore:

```
find("anova")

## [1] "package:stats"
```

1.4.3.5 Test esistenza

`exists` testa se un oggetto esiste all'interno dei environment:

```
exists("anova", where = "package:stats")

## [1] TRUE
```

1.4.3.6 Rimozione

Si usa `rm`; se necessario specificare l'environment, mediante l'opzione `pos`:

```
rm(x,y,z)                  # <- elimina x, y, z
rm(list = ls())            # <- elimina tutti gli oggetti
rm(list = ls(all = TRUE)) # <- idem, con anche quelli nascosti
```

```
# Specifica dell'environment
names(Indometh)

## [1] "Subject" "time"    "conc"

attach(Indometh)
search()

##  [1] ".GlobalEnv"        "Indometh"          "adf"
##  [4] "package:knitr"     "tex2pdf"           "package:stats"
##  [7] "package:graphics"  "package:grDevices" "package:utils"
## [10] "package:datasets"  "package:methods"   "Autoloads"
## [13] "package:base"

rm(conc, pos = "Indometh")
ls(pos = 2)

## [1] "Subject" "time"

## rimozione di oggetti da pacchetti non è possibile per ovvi motivi
rm(anova, pos = "package:stats")

## Error in rm(anova, pos = "package:stats"):  non è possibile rimuovere
i binding da un ambiente bloccato
```

1.4.3.7 Nomi presenti in più environments

La funzione `conflicts` individua gli oggetti presenti in più di un environment

```
library(survival)
library(boot)
```

```
##
## Caricamento pacchetto:  'boot'
## Il seguente oggetto è mascherato da 'package:survival':
##
##     aml

conflicts()

## [1] "aml"       "body<-"    "kronecker" "plot"

find("aml")

## [1] "package:boot"     "package:survival"
```

1.4.3.8 Ottenere un oggetto da un dato environment

Se più oggetti esistono sotto lo stesso nome, solitamente ad eccezione di quello residente nel primo environment, gli altri non saranno accessibili. Per ottenere una copia di questo è necessaria `get`.
Ad esempio:

```
aml1 <- get("aml", pos = 'package:boot')
aml2 <- get("aml", pos = 'package:survival')
```

1.5 Interfacce con l'esterno

1.5.1 Importare/esportare dati

Le funzioni principali per l'importazione dati sono:

- `read.table` per leggere dati in formato testuale (`read.csv` o `read.delim` sono wrapper di `read.table`);
- `readLines` per leggere righe di un (qualsiasi) file testuale, memorizzate in un vettore di caratteri;
- `readRDS` per la lettura di file binario (prodotto da R) contenente un singolo oggetto;
- `load` per il caricamento di un file binario (prodotto da R) contenente un insieme di oggetti

La *scrittura di file* presenta funzioni speculari:

- `write.table` per scrivere dati in formato tabellare
- `writeLines` per la scrittura di un vettore di stringhe come righe di un file di testo
- `saveRDS` per il salvataggio di singoli oggetti R[3]

[3] saveRDS permette anche la scrittura di output testuale ASCII

- `save` o `save.image` per il salvataggio di un set di oggetti R salvati su un formato binario R
- `write.foreign`, del pacchetto `foreign`, per la scrittura di file testuali facilmente importabili da package come Stata, SAS o SPSS

1.5.1.1 `read.table`

Il modo più generico per per importare dati da un file di testo a formato tabulare è utilizzare `read.table`, la quale legge dati da un file e crea un data.frame.
La funzione ha alcuni importanti argomenti:

- `file`: nome del file o della connessione
- `header`: valore logico che indica che il file nella prima riga presenta il nome delle variabili
- `col.names`: un vettore dove si può specificare il nome assunto dalle variabili (colonne). Se non presente, e il file non ha una prima riga che le specifichi, i nomi sono generati automaticamente dal sistema
- `colClasses`: un vettore che può specificare la classe di ogni colonna importata dal dataset
- `sep`: serve per specificare il separatore delle colonne
- `dec`: serve per specificare il separatore dei numeri decimali
- `skip`: numero di linee da saltare, dall'inizio, prima di iniziare a importare dati. A volte all'inizio del file vi sono metadati o altre informazioni non strettamente necessarie all'importazione, che possono esser saltate specificando questo parametro
- `nrows`: numero di righe dal leggere, al massimo
- `stringsAsFactors`: un valore logico, di default `TRUE`, che specifica se i character devono essere trasformati automaticamente in factor
- `comment.char`: una stringa che specifica il carattere di commento (quando trovato in una linea, si ignora la rimanente parte della stessa nell'importazione e si passa direttamente alla successiva). Di default è impostato a `"#"`, ma impostandolo a `""` (se non vi sono commenti) incrementa le prestazioni in lettura (poichè disabilita la ricerca di commenti e velocizza l'esecuzione)

Ad esempio per leggere un file csv, il seguente comando salverà in un dataframe di nome `db` il contenuto di `file.csv` (residente nella working directory di R)

```
db <- read.table(file = "file.csv", sep = ",", dec  = "." )
```

Alcune **ottimizzazioni in fase di lettura**, utili per file di dimensioni ingenti:

- impostare comment.char a `""`

- utilizzare `colClasses`: specificandolo si guadagna molto in efficienza perchè si disattiva il default di R che cerca di capire il tipo di dato da importare (se ad esempio tutte le colonne sono numeriche basta specificare `colClasses="numeric"` e tale vettore verrà riciclato).
  Si noti che colClasses è specificato per colonna (non per variabile) e quindi deve essere specificato anche per la colonna dei row.names, se presente).
  Valori possibili da specificare nel vettore di `colClasses` sono i seguenti; ad eccezione di NA, vanno tutti posti tra virgolette (sono un char, anche `NULL`):
  - `NA`, il default che lascia ad R il cercare di convertire sulla base dei dati
  - `NULL` (specificando anche le virgolette) la colonna è skippata dall'importazione
  - una delle classi di oggetti atomici (`logical`, `integer`, `numeric`, `complex`, `character`, `raw`)
  - `factor`
  - `Date` o `POSIXct`
  - un formato che sia supportato da una funzione della famiglia `as.`
- utilizzare `nrows` può esser utile per evitare di importare un intero dataset; anche importando tutto il dataset, specificare il numero esatto di linee da importare incrementa le prestazioni (si usi `wc -l` su sistemi Unix)

Ad esempio un modo quick'n dirty di importare un dataset efficientemente può essere

```
initial <- read.table("datatable.txt", nrows=100)
classes <- sapply(initial, class)
final.db <- read.table("datatable.txt", colClasses=classes)
rm(initial, classes)
```

1.5.1.2 `write.table`

`write.table` è la funzione generica per l'output in formato testuale; ha opzioni simili a read.table (nel caso vedere l'help).
Ad esempio, volendo salvare un `data.frame` (o una matrice o qualcosa che sia coercibile al primo) di nome `sei` nel file `sei.csv`, semplicemente comandiamo

```
write.table(sei             ,   # l'oggetto da esportare
            file = "sei.csv",   # il nome del file in cui salvarlo
            sep  = ","      ,   # il separatore e' una virgola
            dec  = "."      )   # per decimale si utilizza il punto
```

1.5.1.3 `dput`, `dump`, `dget` e `source`

Vi sono altri formati testuali nei quali si può salvare i dati, oltre alla versione tabellare; questi formati sono sempre testuali, ma si differenziano da quelli prodotti da `write.table`.

Le due funzioni principali che svolgono questo compito sono `dump` e `dput`, utilissime ad esempio per inviare dati via mail nelle mailing list di supporto, per dare riproducibilità alla richiesta.
Tipicamente questi formati contengono più metadati (es il tipo di classe di ogni oggetto, es classi delle colonne di un dataframe) rispetto a quelli di write.table (anche se sono intelleggibili direttamente solamente entro R), che non vengono persi tra esportazione/importazione, e che quindi non sono da specificare una seconda volta.

La funzione `dput` prende in input un singolo oggetto R e **crea codice R in grado di ricreare l'oggetto** in R. Ad esempio applichiamolo ad un dataframe

```
dput(Formaldehyde)

## structure(list(carb = c(0.1, 0.3, 0.5, 0.6, 0.7, 0.9), optden = c(0.086,
## 0.269, 0.446, 0.538, 0.626, 0.782)), class = "data.frame", row.names = c("1",
## "2", "3", "4", "5", "6"))
```

Tutti i metadati come rownames, names e class sono conservati. Solitamente si vuole salvare su file il risultato quindi si procederà come segue

```
dput(Formaldehyde, file = "formal.R")
```

In seguito per importare i dati si utilizzerà la funzione gemella `dget`

```
(my.formal <- dget(file = "formal.R"))

##   carb optden
## 1  0.1  0.086
## 2  0.3  0.269
## 3  0.5  0.446
## 4  0.6  0.538
## 5  0.7  0.626
## 6  0.9  0.782
```

La funzione `dump` è molto simile a `dget`, ad eccezione che:

- può esportare oggetti multipli
- esporta anche il/ nome/i dell'oggetto/ oltre alla sua rappresentazione in codice R, e non sarà possibile "rinominarlo" direttamente in importazione
- l'importazione è effettuata mediante `source`

```
x <- "foo"
y <- data.frame(a=1, b="a")
dump(c("x","y"), file="data.R")
rm(x,y)
source("data.R")
x
```

```
## [1] "foo"

y

##   a b
## 1 1 a
```

1.5.1.4 `saveRDS`, `save` e `save.image`

`saveRDS`, `save` e `save.image` servono per esportare nei formati binari di R: `.rds`, `.rda` (uguale, per applicazione, all'estensione `.Rdata`).
`saveRDS` salva un singolo oggetto in un file binario, con la possibilità, in importazione di assegnargli un nuovo nome

```
saveRDS(Indometh, file="Indo.rds")
```

`save` salva uno o più oggetti in un medesimo file

```
# Un oggetto
save(asd, file="asd.rda")
# Più oggetti nel medesimo file
save(asd1, asd2, file = "real.rda")
save(list = c("asd1","asd2"), file="asds.rda")
```

Infine `save.image` serve per salvare l'intero workspace:

```
save.image("salvataggio.rda")
```

1.5.1.5 `readRDS` e `load`

Per l'importazione di `rds` si usa `readRDS`:

```
my.indo <- readRDS(file="Indo.rds")
```

Per caricare file `rda` or `Rdata` bisogna utilizzare indistintamente `load`. Questi a contrario della precedente non permettono un'assegnazione, e il nome è quello impiegato nella procedura di salvataggio. Un esempio:

```
load("prova.Rdata")
load("miei_dataset/ciao.rda")
```

1.5.2 Log, esecuzione script

1.5.2.1 Log

Per registrare **sia comandi che output** delle elaborazioni preparare un file sorgente ed eseguirlo in modalità batch; il file di output conterrà entrambi.
Per registrare il **solo output delle elaborazioni** su un file comandare `sink("path_file")`, per terminarla semplicemente `sink()`. Ad esempio

```
sink("mio.log")
cor(sei)
sink()
```

Può presentarsi la necessità di salvare gli **output specifici** di alcuni comandi (ad esempio è utile se non abbiamo necessità di guardare tutto un file `sink`) : `capture.output` permette di stampare su un file l'output di una determinata espressione R.

```
capture.output(cor(sei), file = "prova.txt")
```

1.5.2.2 Script

Si è visto in precedenza l'utilizzo di source per importare strutture di dati. In generale `source` esegue codice presente su un file nel punto in cui viene chiamata. Generalmente è silenziosa; si aggiunga l'opzione `echo=T` se si vuole che i calcoli e i risultati vengano stampati a video

```
source("/home/l/test.R", echo = TRUE)
```

1.5.3 Connessioni

Vi sono diverse funzioni che R ci mette a disposizione per aprire *connessioni* con il mondo esterno. Vi sono diverse tipologie di funzioni/connessioni, le principali sono:

- `file` è la funzione standard per aprire connessioni ad un file standard non compresso (es testuali)
- `gzfile`, usata per connessioni a file compressi con l'algoritmo `gzip` (file di estensione `.gz`)
- `bzfile`, per file compressi con l'algoritmo `bzip2` (file di estensione `.bz2`)
- `url`: quando si apre una connessione si può leggere dati da quella pagina specificata .
- `pipe`: se si vuole prendere in input o dare in output ad un programma esterno (eseguito in una shell) determinati comandi
- `socketConnection`: per l'apertura di socket

Una connessione può essere aperta per varie finalità, che vengono specificate nelle funzioni attraverso l'argomento `open`: Non tutte le modalità sono applicabili a tutte le connessioni; ad esempio le url possono esser aperte solo in lettura, mentre file e socket possono esser aperti in lettura e scrittura.

La **procedura per l'uso di una connessione** richiede la sua apertura, il suo utilizzo, la sua chiusura ed eliminazione. Vediamo un esempio con `file`

mode	descrizione
`r`	Open for reading in text mode
`w`	Open for writing in text mode
`a`	Open for appending in text mode
`rb`	Open for reading in binary mode
`wb`	Open for writing in binary mode
`ab`	Open for appending in binary mode
`r+`	Open for reading and writing
`w+`	Open for reading and writing, truncating file initially
`a+`	Open for reading and appending

Tabella 1.2: Opzioni di apertura delle connessioni

```
con <- file("foo.txt",open="r")
data <- read.csv(con)
close(con)
rm(con)
```

Molte funzioni si preoccupano autonomamente di gestire le connessioni in maniera automatica (es read.table lo fa). Ma a volte gestire la connessione direttamente ci permette maggiore flessibilità. Un esempio è il caso seguente dove vogliamo leggere 10 linee al massimo da un file (esempio con `gzfile`)

```
con <- gzfile("r-base_3.0.2-1.diff.gz")
rdiff <- readLines(con,10)
close(con)
head(rdiff, n=2)
```

Un esempio con `url`

```
> site <- url("http://bragliozzo.altervista.org")
> site
                        description
"http://bragliozzo.altervista.org"
                              class
                              "url"
                               mode
                                "r"
                               text
                             "text"
                             opened
                           "closed"
                           can read
                              "yes"
                          can write
                               "no"
> site.index <- readLines(site)
> head(site.index) # stampa la prima riga dell'html
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01...>"
[2] "<html>"
```

```
[3] "<head>"
[4] "<title>Luca Braglia's HomePage</title>"
[5] "<meta name=\"author\" content=\"Luca Braglia\">"
[6] "<meta name=\"keywords\" content=\"Luca, Braglia,Homepage\">"
```

Infile alcuni esempi con `pipe`:

```
## Esempio in lettura da un comando di sorting della ls della directory
sorted.ls <- pipe("ls /tmp | sort")
head(readLines(sorted.ls))
## [1] "06_22.c"       "a.out"         "ArteLaTeX.pdf"
## [4] "asd"           "asd.csv"       "biostat"

## Esempio in scrittura
con <- pipe("touch")
writeLines(c("foo","bar","baz"), con) # cosi crea i file foo,bar,baz
```

Per maggiori info vedere `?connections`

1.5.4 Interfaccia col sistema operativo, directory e file

Per impartire comandi al sistema operativo utilizzare il comando `system2()`

```
system2("ls")
```

A differenza di creare un pipe, non ci da la possibilità di interagire con il programma creato; ad esempio non possiamo importarne l'output (ciò che riusciamo a salvare è il valore di uscita restituito dal programma) o fornirgli input:

```
> (a <- system("ls Downloads"))
[1] 0
> (b <- system("ls CartellaInesistente"))
[1] 2
```

Alcune funzioni utili riguardanti le cartelle:

Funzione	Significato
`list.dirs`	elenca le directory presenti in una dir
`setwd`	imposta working directory
`getwd`	stampa a video il nome la directory corrente
`dir`	lista il contenuto di una cartella
`dir.create`	crea una cartella
`dirname`	estrae un path alla cartella di residenza da quello del file

Alcune funzioni utili riguardanti i file:

Funzione	Significato
`list.files`	elenca i file in una dir
`file.create`	crea un file
`file.remove`	elimina un file o directory vuota[4]
`file.rename`	rinomina un file
`file.exists`	verifica l'esistenza del file
`file.access`	verifica l'accessibilità al file
`file.append`	copia in coda il secondo file specificato al primo
`file.show`	apre un pager su un file
`file.path`	costruisce il path (tipicamente relativo di un file a partire dai suoi elementi
`file.copy`	copia un file
`file.symlink`	crea un link
`file.info`	fornisce informazioni sul file
`download.file`	scarica un file da internet

1.6 Opzioni

Il comando `options()` permette di stampare o impostare le opzioni globali di R.

```
## options() ## too long, print all options
options("width") ## read settings

## $width
## [1] 80

options("width" = 100) ## set new values
```

Impostare le opzioni in runtime fa si che esse si perdano al riavvio di R. Per evitare questo si possono specificare le opzioni nel file di inizializzazione (`$HOME/.Rprofile`), un file che viene eseguito prima dell'avvvio del programma.

1.7 Guida in linea aiuto ecc.

1.7.1 Sintassi

La guida in linea ad un comando semplicemente si ottiene digitando quest'ultimo dopo un punto interrogativo; stessa cosa fa `help("nome_comando")`. Per l'aiuto su parole riservate o comandi non alfanumerici utilizzare le virgolette.

```
?mean
help("mean")
?"while"
?"@"
```

1.7.2 Ricerca funzioni

Se non si conosce il nome del comando di cui si ha bisogno, si utilizzi `help.search()` o `apropos()`:

- `apropos` evidenzia tutte le funzioni che contengono la chiave cercata e che possono al momento esser utilizzate

```
apropos('mean')

##  [1] ".colMeans"     ".rowMeans"     "colMeans"      "kmeans"         "mean"           "m
##  [7] "mean.default"  "mean.difftime" "mean.POSIXct"  "mean.POSIXlt"   "rowMeans"       "w
```

- `help.search` cerca la parola chiave in tutto il manuale del comando, anche nei pacchetti non caricati.

```
help.search("mean")
```

- alternativamente `sos::findFn`

1.7.3 Esempi, vignette, help dei pacchetti

- `example` per avere esempi sull'utilizzo di un comando usare

```
example("mean")
```

- `library(help="nome")` per accedere alla guida in linea di una pacchetto, che mostra le funzioni in esso contenute nonché una breve descrizione, utilizzare

```
library(help="stats")
```

- `vignette` per documenti più discorsivi

```
vignette()          ## elenca tutti i nomi es "arima"
vignette("arima")   ## visualizza la vignetta scelta
```

1.7.4 Ricerca di errori su Internet

Per ricercare errori su Internet, effettuare questo setup (per ottenere l'errore in inglese) ed eseguire il codice che genera errore. Copiare ed incollare su google

```
Sys.setenv(LANGUAGE = "en")
```

Capitolo 2

Gli oggetti di R

La materia prima su cui R lavora sono gli **oggetti**. In R

- **tutto è un oggetto**: dati, risultati intermedi e finali sono memorizzati in oggetti, per esser poi riutilizzati in seguito;
- **ogni oggetto ha una classe**: la classe dell'oggetto descrive cosa in esso è contenuto e quali funzioni (metodi) si possono ad esso applicare

2.1 Classificazione degli oggetti

Gli oggetti che costituiscono la materia prima delle elaborazioni sono *vettori*, *matrici*, *array liste* e *data frame*.[1].
Questi possono esser classificati (tabella 2.1) in base alla loro **dimensionalità** (1, 2 n dimensioni) e in base all'**omogeneità** del loro contenuto (ovvero se contengono elementi dello stesso tipo o no). Pressochè tutti gli altri oggetti si fondano su questi oggetti di base.

2.1.1 Strutture atomiche

La caratteristica comune delle strutture atomiche consiste nell'essere composte da *dati aventi lo stesso tipo*; sono pertanto strutture atomiche i vettori atomici, le matrici e gli array.
L'atomicità di un oggetto può essere testata con `is.atomic`

[1]Per la creazione di questi oggetti si usa principalmente `c()` per i vettori, si usa `matrix()` per le matrici, `array()` per gli array, `list()` per le liste, `data.frame()` per i data frame.

	Omogeneo	Eterogeneo
1d	Vettore atomico	Lista
2d	Matrice	Data frame
nd	Array	

Tabella 2.1: Classificazione strutture dati di base in R

2.1.1.1 Tipi di dato delle strutture atomiche

Vi sono 6 **tipi** di dato di cui può essere composta una struttura atomica:

- `logical` (valori logici, es `TRUE` o `FALSE` [2])
- `integer` (numeri interi relativi, es `1L`, `-2L`)
- `double` (numeri con virgola, es `1`, `0.1`, `.2`[3])
- `character` (stringhe, es `"ciao"`)
- `complex` (numeri complessi: `1+3i`)
- `raw` (numeri in notazione esadecimale, es `0a`)

Il tipo di dato di una struttura atomica può essere interrogato mediante la funzione `typeof()`, fornendo come argomento la struttura stessa

```
typeof(1)

## [1] "double"

typeof(1:3)

## [1] "integer"

typeof(matrix(letters, 2))

## [1] "character"
```

Esistono altresì una serie di funzioni specializzate che testano (restituendo un valore logico `TRUE` o `FALSE`) il tipo di dato della struttura come `is.logical`, `is.integer`, `is.character`, eccetera

```
is.double(1)

## [1] TRUE

is.integer(1:3)

## [1] TRUE

is.character(matrix(letters, 2))

## [1] TRUE
```

`is.numeric` è un test generico che restituisce vero se viene applicato ad un oggetto di tipo `integer` o `double`.

[2] È anche possibile utilizzare `T` ed `F`, sono meglio gli standard `TRUE` e `FALSE`
[3] Bisogna dunque specificare il postfisso `L` ad un numero se si vuole l'intero

2.1.1.2 Costanti speciali per le strutture atomiche

A comporre una struttura atomica, oltre al dato, vi possono essere **quattro costanti speciali**:

- `NULL`: valore **vuoto**
- `Inf`: **infinito** (risultato di $1/0$)
- `NA`: valore **mancante** (sta per Not Available[4]). Calcoli numerici e logici che hanno a che fare con `NA`, ritornano solitamente `NA`.
- `NaN`: non un numero (es risultato di $0/0$). I valori `NaN` sono incomparabili, quindi test di uguaglianza o altro con `NaN`, restituiranno `NA`.

Anche per l'individuazione di queste costanti all'interno di una struttura atomica vi sono apposite funzioni della famiglia `is.*` che restituiscono una struttura logica equivalente a quella testata con il test per singolo elemento. Vediamo un esempio con un vettore

```
x <- c(NULL, 1, NaN, NA, 4, -Inf)
x

## [1]    1  NaN   NA    4 -Inf

## il dato null viene eliminato, è vuoto
is.infinite(x)

## [1] FALSE FALSE FALSE FALSE  TRUE

is.na(x)

## [1] FALSE  TRUE  TRUE FALSE FALSE

is.nan(x)

## [1] FALSE  TRUE FALSE FALSE FALSE
```

In genere un `NaN` è considerato un `NA` (ovvero un dato mancante), ma il contrario non è necessariamente vero (ovvero un `NA`, non necessariamente è un `NaN`)

2.1.1.3 Coercizione

Nel caso cercassimo di creare/modificare una struttura atomica in maniera tale da contenere dati di tipo differente, non viene segnalato errore, bensì avviene una **coercizione** automatica dal tipo meno flessibile a quello più flessibile. Nello specifico, l'ordine di trasformazione è `logical -> integer -> double -> character`. Vediamo alcuni esempio attraverso i vettori

[4] Il valore mancante `NA` è diverso dalla stringa `"NA"`

```
typeof(c(TRUE, 1L))

## [1] "integer"

typeof(c(1L, 2))

## [1] "double"

typeof(c(2, "a"))

## [1] "character"

typeof(c("a", TRUE))

## [1] "character"
```

La coercizione di cui sopra è detta **implicita** o automatica. Nel caso di trasformazione da logico a intero/double, `TRUE` diventa 1 menttre `FALSE` diventa 0.
Un altro tipo di coercizione è quella **esplicita** operata dall'utente che vuole trasformare un tipo di dato in un altro mediante funzioni della famiglia `as.*` (es `as.logical`, `as.integer`, `as.double`, `as.character`).
La coercizione spesso avviene automaticamente; la maggior parte delle funzioni matematiche converte gli `integer` in `double` e gli operatori logici cercheranno di convertire a logico, come ad esempio in

```
1 || FALSE

## [1] TRUE
```

La coercizione in generale può far si che si perda informazione (e venga emesso un `Warning`) se passiamo da un tipo di dato più flessibile (es un `character`) ad uno meno (es `double`), che si può manifestare mediante valori mancanti, ad esempio nel caso

```
as.logical(c("TRUE", "a"))

## [1] TRUE   NA
```

o minor precisione del dato immagazzinato, come ad esempio in

```
as.integer(c(1.9, 2.9))

## [1] 1 2
```

2.1.2 Strutture non atomiche

Appartengono alle strutture non atomiche quelle strutture di dati utili per immagazzinare dati di tipologia differente e in ultima analisi *sono composte da differenti strutture atomiche*; nello specifico abbiamo

- `data.frame`, che memorizzano al proprio interno vettori atomici aventi tipo anche differente, ma della stessa lunghezza;
- `liste`, che memorizzano al proprio interno altre strutture, di tipo e dimensione differente.

2.1.3 Visualizzare una struttura dati con `str`

Dato un oggetto, il modo migliore di capire che struttura di dati abbia è utilizzare la funzione **`str`** (abbreviato di *structure*), la quale si occupa di indicare la *struttura*, stampare una *parte di dati* e listare *attributi* dell'oggetto.
Per ciò che riguarda le strutture di dati più complesse, in generale le riconduce alle strutture base e plotta approssimativamente una linea per oggetto "base".
Vediamo applicata ad alcune strutture dati di base

```
## Vettore di caratteri
str(letters)

##  chr [1:26] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"

## Vettore di interi
x <- 1:100
str(x)

##  int [1:100] 1 2 3 4 5 6 7 8 9 10 ...

## Vettore di double
x <- rnorm(100)
str(x)

##  num [1:100] 0.00468 0.54266 -1.49875 0.3867 0.25192 ...

## Factor
x <- gl(2, 3, labels = c("Control", "Treat"))
str(x)

##  Factor w/ 2 levels "Control","Treat": 1 1 1 2 2 2

## Matrice
x <- matrix(rnorm(100), 10, 10)
str(x)

##  num [1:10, 1:10] 1.63 -1.019 0.774 0.683 0.641 ...

## Array
x <- array(rnorm(24), dim = 2:4)
str(x)

##  num [1:2, 1:3, 1:4] -0.486 0.153 0.248 -0.664 -0.38 ...

## Data Frame
str(Formaldehyde)
```

```
## 'data.frame': 6 obs. of  2 variables:
##  $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
##  $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782

## Lista
x <- list(letters, 1:5, Formaldehyde)
str(x)

## List of 3
##  $ : chr [1:26] "a" "b" "c" "d" ...
##  $ : int [1:5] 1 2 3 4 5
##  $ :'data.frame': 6 obs. of  2 variables:
##   ..$ carb  : num [1:6] 0.1 0.3 0.5 0.6 0.7 0.9
##   ..$ optden: num [1:6] 0.086 0.269 0.446 0.538 0.626 0.782
```

`str` mostra anche tutti gli attributi di un oggetto

```
comment(Formaldehyde) <- "myComment"
str(Formaldehyde)

## 'data.frame': 6 obs. of  2 variables:
##  $ carb  : num  0.1 0.3 0.5 0.6 0.7 0.9
##  $ optden: num  0.086 0.269 0.446 0.538 0.626 0.782
##  - attr(*, "comment")= chr "myComment"
```

2.2 Attributi degli oggetti

Gli oggetti possono avere uno o più attributi a loro connessi; questi sono **metadati** che

- regolano il funzionamento delle operazioni sugli oggetti stessi o
- descrivono maggiormente la struttura dati considerata

2.2.1 Accesso agli attributi

Per accedere in lettura/scrittura agli attributi di un oggetto si può utilizzare `attributes()` o `attr()`:

attributes accede a tutti gli attributi dell'oggetto considerato come una lista

```
## Lettura
attributes(Formaldehyde)

## $names
## [1] "carb"   "optden"
##
## $class
## [1] "data.frame"
```

```
##
## $row.names
## [1] "1" "2" "3" "4" "5" "6"
##
## $comment
## [1] "myComment"

## Scrittura
attributes(Formaldehyde)$names <- c("Foo", "Bar")
```

attr accede a un singolo attributo

```
## Lettura
attr(Formaldehyde, "names")

## [1] "Foo" "Bar"

## Scrittura
attr(Formaldehyde, "names") <- c("carb", "optden")
```

La sintassi di scrittura può essere usata anche per creare **nuovi attributi**, non solo modificare quelli esistenti, es per memorizzare informazioni aggiuntive/metadati.

```
attr(Formaldehyde, "description") <- "Determination of Formaldehyde"
```

Infine, per l'accesso agli *attributi principali*, esistono funzioni ad hoc che hanno il medesimo nome dell'attributo di nostro interesse, le quali possono essere utilizzate sia in lettura che in scrittura

```
names(Formaldehyde) <- c("Asd","Qwe")
```

2.2.2 Attributi principali

Gli attributi più importanti di un oggetto sono:

- la **classe** (`class`) dell'oggetto;
- i **nomi** vari delle componenti dell'oggetto (`names`, `col.names`, `row.names` e `dimnames`);
- **dimensioni** (`dim`) dell'oggetto (ovvero numero di righe/colonne ecc per matrici, array ecc).

2.2.2.1 Classe dell'oggetto

La classe di un oggetto, cui è possibile accedere mediante l'omonima funzione `class`, determina le operazioni che si possono svolgere sullo stesso e come viene gestita all'interno del sistema quel tipo di oggetto (es. come viene plottato, come viene stampato a video ecc).

Nomi dell'oggetto I nomi dell'oggetto sono un attributo fondamentale poiché:

- utile per scrivere codice leggibile, poiché se un oggetto ha nomi, ci si può riferire ad alcune sue componenti senza la notazione numerica degli indici;
- se presente viene utilizzato come etichetta nelle stampe a video dell'oggetto interessato.

`names` può essere utilizzato per tutti gli oggetti. A seconda dell'oggetto cui si applica avrà una diversa interpretazione.
In strutture tabellari di dati (es matrici, data.frame) `colnames` riguarda le colonne dell'oggetto, `rownames` le righe.
`dimnames` può esser utilizzato, oltre a matrici e a dataframe, anche in liste per ottenere i nomi associati alle varie dimensioni.

Dimensioni L'attributo dimensione (cui è possibile accedere mediante `dim`) specifica la struttura (es numero righe, colonne ecc) che hanno gli oggetti con almeno due dimensioni (matrici, array e dataframe).

2.3 Vettori atomici

I **vettori in generale** sono la struttura dati di base di R. Ne abbiamo di **due tipi**: i **vettori atomici** e le **liste**.
Vettori atomici e liste:

- si accomunano per essere una struttura dati ad una dimensione che ha una determinata **lunghezza** (utilizzare `length()`) che rappresenta quanti elementi contengono;
- si differenziano per l'omogeneità in quanto i vettori atomici contengono elementi dello stesso tipo mentre le liste possono contenere oggetti di tipo diverso.

Nel prosieguo *ci concentriamo sui vettori atomici* e vediamo le liste nella prossima sezione.

2.3.1 Creazione

I **vettori atomici** sono, appunto, una struttura di dati atomica, quindi possono contenere informazioni strettamente del medesimo tipo ed ereditano tutte le proprietà già viste delle strutture atomiche.
La **creazione** di vettori atomici:

- avviene tipicamente attraverso la funzione `c()` (diminutivo di *combine*), listando tutti gli elementi tra parentesi separati da virgole.

```
a <- c(TRUE, FALSE, FALSE)
b <- c(1, 4, 2)
c <- c(1L, 4L, 2L)
d <- c("a","x","y")
```

- nel caso di un vettore di `integer`, si può operare specificando gli estremi (con step unitari) ed utilizzando l'operatore `:`

```
e <- 1:15
```

- alcune *funzioni utili* nella creazione di vettori sono `seq` (per sequenze numeriche) e `rep` (per la ripetizione di vettori forniti) .

2.3.2 Factor

Il factor è un particolare tipo di dato immagazzinabile in un vettore, che in R serve per la rappresentazione delle variabili categoriali (*unordered* oppure *ordered*); effettivamente consiste in un vettore di `integer` cui sono associate etichette (*labels*).
Utilizzare i factor indica a molte funzioni di R che questa è una variabile categoriale, e pertanto viene trattata in maniera particolare.

I factor vengon creati mediante l'omonima funzione:

```
## --------------------
## Non ordinato: genere
## --------------------
gender <- factor(c("m","f","f","m"),
                 levels = c("m","f"),
                 labels = c("Maschio", "Femmina"))
gender

## [1] Maschio Femmina Femmina Maschio
## Levels: Maschio Femmina

class(gender)

## [1] "factor"

typeof(gender)

## [1] "integer"

as.integer(gender)

## [1] 1 2 2 1

## ----------------------------
## Ordinato, gradi di istruzione
## ----------------------------
school <- factor(c(1, 2, 1, 1, 3),
                 levels=1:3,
                 ordered=TRUE,
                 labels=c("Primaria", "Secondaria", "Terziaria"))
school
```

```
## [1] Primaria   Secondaria Primaria   Primaria   Terziaria
## Levels: Primaria < Secondaria < Terziaria

class(school)

## [1] "ordered" "factor"

typeof(school)

## [1] "integer"
```

Peculiarità dei factor è l'attributo `levels`, che specifica le categorie del dato: esso è accessibile mediante l'omonima funzione, che se utilizzata come *lvalue* permette di modificare le etichette del factor.

```
attributes(school)

## $levels
## [1] "Primaria"   "Secondaria" "Terziaria"
##
## $class
## [1] "ordered" "factor"

levels(school)

## [1] "Primaria"   "Secondaria" "Terziaria"

levels(school) <- c("Prim.", "Sec.", "Terz.")
school

## [1] Prim. Sec.  Prim. Prim. Terz.
## Levels: Prim. < Sec. < Terz.
```

Anche per un factor non ordinato è in qualche modo importante l'ordinamento dei levels, poichè come gruppo di confronto nelle procedure di stima è impiegato (di default, almeno) il primo gruppo del factor (`maschio` e `Primaria`, nei casi di sopra).
Altre funzioni utili sono:

- `nlevels` restituisce il numero di modalità di un factor.

  ```
  nlevels(school)

  ## [1] 3
  ```

- `relevel` serve per specificare ad un factor non ordinato, ma già formato, che d'ora in poi utilizzi un'altro gruppo specificato come gruppo base:

  ```
  (gender.mod <- relevel(gender, ref = "Femmina"))

  ## [1] Maschio Femmina Femmina Maschio
  ## Levels: Femmina Maschio
  ```

2.3.3 Vettorizzazione e recycling

La vettorizzazione è una caratteristica di R che rende facile l'utilizzo dalla linea di comando e in generale la scrittura di codice compatto (che non necessiti di loop per operazioni comuni, una cosa utile per un linguaggio computazionale). Supponendo di avere due due vettori

```
x <- 1:4
y <- 6:9
```

volendo aggiungerli e avendo la stessa dimensione la somma viene fatta elemento per elemento dall'operatore + (**vettorizzazione**)

```
x + y

## [1]  7  9 11 13
```

Se vogliamo confrontare invece un vettore con una costante in altri linguaggi dovrebbero implementare un loop, mentre in R la scrittura è molto più semplice:

```
x > 2

## [1] FALSE FALSE  TRUE  TRUE
```

In generale, i vettori di una medesima espressione, non necessariamente devono avere la stessa lunghezza; nel caso il valore dell'espressione è un vettore con lunghezza pari a quella del vettore più lungo coinvolto nell'espressione.
I vettori piu piccoli vengono **riciclati**, anche in parte, fino a che non matchano la lunghezza del vettore più lungo. In particolare una costante è semplicemente ripetuta.

```
x <- 1:3
y / x

## Warning in y/x:  la lunghezza più lunga dell'oggetto non è un multiplo
della lunghezza più corta dell'oggetto

## [1] 6.000000 3.500000 2.666667 9.000000
```

In quest'ultimo esempio, il vettore `x` di dimensione inferiore viene riciclato per giungere ad un vettore lungo 4 da utilizzare nella divisione (il quarto elemento coincide col primo).

2.4 Liste

Le liste sono oggetti costituiti da un insieme ordinato di componenti, che possono avere differente `typeof` o `class`. Le liste vengono utilizzate per costruire i tipi di dati complissi all'interno di R.
Le liste si costituiscono mediante la funzione `list`, indicando i componenti separati da virgola con la sintassi `contenuto, contenuto, ...` o `nome=contenuto,`

`nome=contenuto, ...` . La differenza è che nel primo modo gli elementi della lista non hanno names, nel secondo sì:

```
x <- list(1, "a", TRUE)
x

## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE

pp <- list(name = "Peter", surname = "Pan", age = 15)
pp

## $name
## [1] "Peter"
##
## $surname
## [1] "Pan"
##
## $age
## [1] 15
```

Le liste hanno **due set di indici**, uno per il n-esimo componente della lista (che può essere un nome fornito durante la chiamata a `list`, o un numero progressivo circondato da due quadre in sua assenza) e l'indice riguardande il singolo elemento all'interno dell'n-esimo componente della lista.

Per ampliare una lista è necessario usare gli indici qualora non abbia nomi, o concatenarla mediante `c` nel caso rimanente

```
x[4] <- FALSE
pp <- c(pp, list(girlfriend = "Wendy"))
```

La funzione `length` applicata alle liste restituisce il numero dei componenti.

2.5 Matrici ed Array

Matrici ed array sono, come i vettori, strutture di dati atomici; sono tuttavia caratterizzate dalla presenza di una attributo di **dimensione**.
Matrici ed array si differenziano tra loro per il fatto che la matrice ha due dimensioni, mentre un array ne può avere a piacere.

2.5.1 Creazione mediante le funzioni `matrix` e `array`

Le funzioni `matrix` e `array` costituiscono il modo più semplice per costituire array a 2 o n dimensioni rispettivamente:

```r
## Matrice
m <- matrix(1:6, nrow = 2, ncol = 3)
m

##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6

class(m)

## [1] "matrix" "array"

attributes(m)

## $dim
## [1] 2 3

## Array
a <- array(1:24, dim = c(4, 3, 2))
a

## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]   13   17   21
## [2,]   14   18   22
## [3,]   15   19   23
## [4,]   16   20   24

class(a)

## [1] "array"

attributes(a)

## $dim
## [1] 4 3 2
```

Le **dimensioni di matrici ed array** sono rappresentate da un *vettore di interi non negativi*: la lunghezza di questo vettore costituisce il numero di dimensioni della matrice/array.
Per il "riempimento" dei dati nella struttura, la matrice/array è costituita "per colonna" (*column-wise*), ovvero facendo muovere un determinato indice (es righe

nelle matrici) più velocemente di quello alla sua destra (colonne). Per le matrici questo comportamento di default può esser cambiato specificando `by.row=TRUE`.
Se non avessimo specificato i dati da inserire, ma solo la struttura, la matrice/array sarebbe stata creata con valori `NA`.
Sia per `matrix` che per `array` è possibile dare un nome alle dimensioni, in sede di creazione, mediante l'opzione `dimnames`(non mostrato) o in seguito mediante l'omonima funzione:

```
dimnames(a) <- list("maiusc" = LETTERS[1:4],
                    "num" = 1:3,
                    "incognita" = c("x","y"))
a
```

```
## , , incognita = x
##
##       num
## maiusc 1 2  3
##      A 1 5  9
##      B 2 6 10
##      C 3 7 11
##      D 4 8 12
##
## , , incognita = y
##
##       num
## maiusc  1  2  3
##      A 13 17 21
##      B 14 18 22
##      C 15 19 23
##      D 16 20 24
```

I nomi delle dimensioni sono interrogabili e modificabili mediante la funzione omonima, `dimnames`, applicata all'oggetto.

2.5.2 Creazione fornendo l'attributo `dim` ad una lista

Alternativamente una matrice/array può esser creata a partire da un vettore o una lista, assegnando a questo l'attributo di dimensione.
Le dimensioni di un oggetto sono ottenibili/modificabili mediante la funzione `dim`:

```
a <- 1:24
b <- a
dim(a) <- c(6,4)
dim(b) <- c(4,3,2)
class(a)
```

```
## [1] "matrix" "array"
```

```
class(b)
```

```
## [1] "array"
```

E' anche possibile creare matrici o array impostando le dimensioni ad una lista; otteniamo una struttura di dati un po' esotica ma alle volte può essere utile per organizzare oggetti complessi in strutture matriciali

```
l <- list(1:3, "a", TRUE, 1.0)
dim(l) <-  c(2, 2)
l

##      [,1]      [,2]
## [1,] integer,3 TRUE
## [2,] "a"       1
```

2.6 Data frames

Sono una "particolare" lista, con classe `data.frame`, in cui tutti gli elementi della lista devono avere la medesima lunghezza, possono avere differenti classi (come nelle liste).

2.6.1 Creazione e modifica

La creazione avviene attraverso la funzione `data.frame`:

```
educ <- c( 5, 10, 13, 10, 20, 15)
age  <- c(30, 24, 45, 50, 90, 70)
inc  <- c(23, 12, 45, 56, 67, 53)
male <- c( 1,  0,  1,  1,  0,  0)
name <- c("gianni", "elisa", "mario", "franco", "giulia", "giada")
(sei <- data.frame(educ, age, inc, male, name))

##   educ age inc male   name
## 1    5  30  23    1 gianni
## 2   10  24  12    0  elisa
## 3   13  45  45    1  mario
## 4   10  50  56    1 franco
## 5   20  90  67    0 giulia
## 6   15  70  53    0  giada
```

Bisogna di solito fare attenzione al comportamento di default di `data.frame`, che trasforma ogni stringa in factor (in questo caso la variabile name); tale comportamento di default può essere disabilitato con il parametro `stringAsFactors = FALSE`.
Funzioni utili sono `nrow` e `ncol`, che restituiscono il numero di righe e colonne rispettivamente del dataframe

Aggiunta di casi o di variabili Aggiungiamo al data frame la variabile single, che andiamo a creare; aggiungiamo in seguito una nuova osservazione:

```
sei$single <- c(0, 0, 1, 1, 0, 1)
sei[7,] <- data.frame(12, 40, 50, 1, 'asd', 1)
sei

##   educ age inc male   name single
## 1    5  30  23    1 gianni      0
## 2   10  24  12    0  elisa      0
## 3   13  45  45    1  mario      1
## 4   10  50  56    1 franco      1
## 5   20  90  67    0 giulia      0
## 6   15  70  53    0  giada      1
## 7   12  40  50    1    asd      1
```

Eliminazione di casi o variabili Per eliminare casi o variabili dal data frame si utilizzi `NULL` o gli indici

```
sei[3] <- NULL ##<---- elimino la terza variabile
sei

##   educ age male   name single
## 1    5  30    1 gianni      0
## 2   10  24    0  elisa      0
## 3   13  45    1  mario      1
## 4   10  50    1 franco      1
## 5   20  90    0 giulia      0
## 6   15  70    0  giada      1
## 7   12  40    1    asd      1

sei <- sei[-2,]  ##<---- in questo modo elimino la seconda unita'
sei

##   educ age male   name single
## 1    5  30    1 gianni      0
## 3   13  45    1  mario      1
## 4   10  50    1 franco      1
## 5   20  90    0 giulia      0
## 6   15  70    0  giada      1
## 7   12  40    1    asd      1
```

Capitolo 3

Indici e subsetting

3.1 Introduzione

L'utilizzo degli operatori di estrazione (tra cui gli indici) permette l'accesso, per lettura o modifica, ad elementi interni ad una struttura di dati.
R ha quattro operatori che permettono l'estrazione di dati

```
x[i]       x[[i]]       x$a       x@a
```

I primi due sono gli indici veri e propri:

`[` ritorna sempre un oggetto della stessa classe dell'originale (es se lo si usa con un vettore, si avrà un vettore, se con una lista, si otterrà una lista); può esser usato per selezionare più di un oggetto (sebbene vi sia una eccezione);

`[[` (per estrarre elementi di liste o `data.frame`); può essere utilizzato per estrarre solamente un unico elemento e la classe dell'oggetto ritornato non necessariamente sarà una lista o un `data.frame` (es si pensi all'estrazione di un vettore da una lista);

`$` (per estrarre elementi di una lista o data.frame a partire dal loro `name`; allo stesso modo di `[[`, non necessariamente ritorna un oggetto della stessa classe

`@` serve per l'estrazione di componenti da oggetti S4

L'estrazione in lettura avviene valutando l'espressione che contiene indici: in generale se come indice viene fornito:

- un valore mancante: verrà restituito un valore mancante
- nulla: verrà restituito l'elemento nel suo complesso (senza operare selezioni)

L'accesso per la modifica di alcuni pezzi della strettura di dati considerata avviene valutando l'espressione con indici nella parte sinistra di un assegnazione. Nella parte destra vi saranno i valori di modifica.
Nel seguito si evidenzieranno tipicamente operazioni di estrazione. Per maggiori info si veda `?Extract`.

3.2 Utilizzo degli indici

3.2.1 Vettori

Per un un generico vettore come indici è possibile utilizzare:

- vettori di interi
- vettori logici
- vettori stringa

Vettore di interi Se utilizziamo un vettore di numeri interi, i valori positivi selezionano i valori della cella corrispondente, i negativi escludono la cella corrispondente dall'output. È necessario che tutti gli elementi dell'indice presentino lo stesso segno.

```
x <- 1:10 * 2
x[1]        ## estraggo il primo valore

## [1] 2

x[c(1, 1)]  ## estraggo due volte il primo valore

## [1] 2 2

x[c(1, NA)] ## se si fornisce un indice mancante viene restituito missing

## [1]  2 NA

x[1.2]      ## i numeri reali sono silenziosamente troncati ad intero

## [1] 2

x[-1]       ## tutti ad eccezione del primo

## [1]  4  6  8 10 12 14 16 18 20

x[1:3]      ## dal primo al terzo

## [1] 2 4 6

x[-(1:3)]   ## tutti ad eccezione di 1,2,3

## [1]  8 10 12 14 16 18 20
```

Se l'indice è positivo ed eccede la lunghezza del vettore, viene dato `NA`; nel caso negativo, non viene tolto nessun elemento.

```
x[30]

## [1] NA

x[-30]

##  [1]  2  4  6  8 10 12 14 16 18 20
```

Se si assegna un valore ad un indice oltre la lunghezza del vettore, si ottiene l'effetto di allungare la struttura sino a comprendere i dati assegnati.

```
x[15] <- 1
x

##  [1]  2  4  6  8 10 12 14 16 18 20 NA NA NA NA  1
```

Vettore logico Nel caso si fornisca come indice vettore logico, sono restituiti i valori per cui tale risulta vera (sono omessi i `FALSE`).

```
x <- c(1:9, NA) * 2
x > 10

##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE    NA

!(x > 10)

##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE    NA

x[x > 10]

## [1] 12 14 16 18 NA

x[ !(x > 10)]

## [1]  2  4  6  8 10 NA
```

Se il vettore logico non ha lunghezza pari a quella del vettore sul quale è applicato come indice, verrà effettuato del recycling

```
x[c(TRUE, FALSE)]

## [1]  2  6 10 14 18
```

Vettore di stringhe Se il vettore presenta l'attributo `names`, è possibile accedere ai suoi dati anche specificando una vettore di stringhe che selezionino i nomi:

```
x <- c(1:9, NA) * 2
names(x) <- letters[1:10]
x

##  a  b  c  d  e  f  g  h  i  j
##  2  4  6  8 10 12 14 16 18 NA

x[c("a", "e")]

##  a  e
##  2 10
```

I nomi devono matchare esattamente.

3.2.2 Liste

Considerando una lista del tipo

```
(pp <- list(name = "Peter",
            surname = "Pan",
            age = 15))

## $name
## [1] "Peter"
##
## $surname
## [1] "Pan"
##
## $age
## [1] 15
```

Possiamo accedere a sue parti mediante l'operatore `[`, `[[` e `$`. Utilizzare il primo ritorna una lista, mentre i rimanenti due estraggono e restituiscono il contenuto dell'elemento della lista

```
## [ ritorna un elemento che è dello stesso tipo
## dell'originale: una lista, composta da un singolo elemento
pp[1]

## $name
## [1] "Peter"

## [[ ritorna un risultato diverso dall'origine, un vettore
pp[[1]]

## [1] "Peter"

## se la lista ha nomi, possiamo utilizzarli insieme a [; ritorna
## in maniera simile a pp[1]
pp["name"]

## $name
## [1] "Peter"

## se la lista ha nomi, possiamo utilizzarli insieme a [[; ritorna
## in maniera simile a pp[[1]]
pp[["name"]]

## [1] "Peter"

## se la lista ha nomi, possiamo utilizzarli insieme a $; ritorna
## in maniera simile a pp[1]
pp$name

## [1] "Peter"
```

Se si vogliono estrarre più elementi da una lista allora bisogna utilizzare `[` (non si possono utilizzare `[[` o `$`):

```
pp[c(1, 3)]

## $name
## [1] "Peter"
##
## $age
## [1] 15

pp[c("surname", "age")] ## nel caso la lista sia dotata di nomi

## $surname
## [1] "Pan"
##
## $age
## [1] 15
```

3.2.3 Matrici ed array

E' possibile fare il subset di una struttura a più dimensioni in tre modi:

- mediante molteplici vettori
- mediante un singolo vettore
- mediante una matrice

Nel seguito ci focalizziamo sulle matrici, l'utilizzo negli array è analogo.

Molteplici vettori Il modo più comune è fornire un vettore per ogni dimensione;

- i vettori di selezione non debbono essere dello stesso tipo
- l'interpretazione da dare agli indici nell'effettuare la selezione è la stessa dei vettori
- nel caso per una dimensione non venga fornito alcun vettore su quella dimensione non si effettuano selezioni

```
a <- matrix(1:9, nrow = 3)
colnames(a) <- LETTERS[1:3]
a[1:2, ]

##      A B C
## [1,] 1 4 7
## [2,] 2 5 8

a[c(T, F, T), c("B", "A")]
```

```
##      B A
## [1,] 4 1
## [2,] 6 3

a[0, -2]

##      A C
```

Di default `[` semplifica il risultato alla minor dimensionalità possibe ovvero ad esempio nel seguente caso vien ritornato un vettore di lunghezza 1, piuttosto che una matrice 1×1:

```
a[1,2]

## B
## 4

is.matrix(a[1,2])    ## singolo indice in ogni dimensione

## [1] FALSE
```

Questo è quello che si vuole di solito, ma se specifichiamo `drop = FALSE` dopo l'ultimo indice, non viene elimnata l'attributo `dim`, come si vedrà.

Un vettore Dato che matrici e array sono implementati come vettori con attributi speciali, si può fare un subset con singolo vettore (come nei vettori veri e propri), e nel caso si comporteranno come un vettore. Matrici ed array sono memorizzati in ordine di colonna

```
(vals <- outer(1:5, 1:5, FUN = "paste", sep = ","))

##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
## [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
## [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
## [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
## [5,] "5,1" "5,2" "5,3" "5,4" "5,5"

vals[c(4, 15)]

## [1] "4,1" "5,3"
```

Utilizzo di matrice E' possibile anche fare subset di dati multidimensionali con una matrice di interi o se la struttura di dati ha nomi con matrice di caratteri. Ogni riga nella matrice specifica la locazione di un valore mentre ogni colonna corrisponde ad una dimensione nell'array subsettato, Questo significa che si usa una matrice a due colonne per subsettare una matrice, una matrice a tre per un array 3d e così via. Il valore restituito è un vettore:

```
select <- matrix(c(1, 1,
                   3, 1,
                   2, 4),
                 ncol = 2,
                 byrow = TRUE)
vals[select]

## [1] "1,1" "3,1" "2,4"
```

3.2.4 Data frame

I `data.frame` posseggono sia le caratteristiche di liste e di matrici:

- se si subsetta con un singolo vettore si comportano come liste
- se si subsetta con due vettori si comportano come matrici

La selezione delle singole variabili avviene mediante `df$nome_var`, ma nel seguito ci concentriamo sul subsetting vero e proprio:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
## Indici interi
df[3, ]

##   x y z
## 3 3 1 c

df[, 3]

## [1] "a" "b" "c"

df[3]    ## list-like

##   z
## 1 a
## 2 b
## 3 c

df[3, 3]

## [1] "c"

## Indici carattere
df[c("x", "y")]   ## list-like

##   x y
## 1 1 3
## 2 2 2
## 3 3 1

df[, c("x", "y")] ## matrix-like
```

```
##   x y
## 1 1 3
## 2 2 2
## 3 3 1

## Indici logici non mostrati
```

Vi è una umportante differenza se si seleziona una singola colonna: la selezione matrix-like semplifica per default mentre quella list-like non lo fa

```
str(df["x"])   # list-like

## 'data.frame': 3 obs. of  1 variable:
##  $ x: int  1 2 3

str(df[, "x"]) # matrix-like

##  int [1:3] 1 2 3
```

Infine, dato che i data frame sono liste di colonne, si può usare `[[` per estrarre colonne

```
df[["x"]]

## [1] 1 2 3

df[[2]]

## [1] 3 2 1
```

3.2.5 S3 objects

GLi oggetti S3 sono costituiti semplicemente da vettori atomici, array e liste, pertanto si può sempre procedere ad estrazione di elementi utilizzando le tecniche descritte in precedenza e le informazioni desumibili da `str`.

3.3 Simplifying vs preserving subsetting

E' importante capre la distinzione tra subsetting semplificante e preservante. Il subsetting:

Preservante preserva la struttura dell'output analoga a quella dell'input e solitamente è meglio nella programmazione, dato che il risultato sarà del medesimo tipo;

Semplificante ritorna la struttura di dati più semplice atta a rappresentare il risultato, e l'output varia lievemente in relazione all'input fornito. Costituisce solitamente il risultato che si desidera in esecuzione interattiva;

Sfortunatamente come switchare da un tipo di subset all'altro dipende dalla tipologia di input, come riassunto in tabella 3.1. Inoltre, come detto, l'effetto del subsetting semplificante dipende dal tipo di dato come mostrato in seguito

	Simplifying	Preserving
Vettore	`x[[1]]`	`x[1]`
Lista	`x[[1]]`	`x[1]`
Factor	`x[1:4, drop = T]`	`x[1:4]`
Array	`x[1, ]` oppure `x[, 1]`	`x[1, , drop = F]` oppure `x[, 1, drop = F]`
Data frame	`x[, 1]` oppure `x[[1]]`	`x[, 1, drop = F]` oppure `x[1]`

Tabella 3.1: Tipi di subsetting, oggetti e implementazione

Vettori atomici In questo il subsetting semplificante rimuove i nomi

```
x <- c(a = 1, b = 2)
x[1]

## a
## 1

x[[1]]

## [1] 1
```

Liste Ritorna l'oggetto all'interno della lista, non una lista da un singolo elemento

```
y <- list(a = 1, b = 2)
str(y[1])

## List of 1
##  $ a: num 1

str(y[[1]])

##  num 1
```

Factor Si eliminano eventuali livelli non utilizzati

```
z <- factor(c("a", "b"))
z[1]

## [1] a
## Levels: a b

z[1, drop = TRUE]

## [1] a
## Levels: a
```

Matrici o array Se vi sono dimensioni di lunghezza 1, elimina quella dimensione

```
a <- matrix(1:4, nrow = 2)
a[1, , drop = FALSE]

##      [,1] [,2]
## [1,]    1    3

a[1, ]

## [1] 1 3
```

Data Frame S l'output è una colonna singola, il subsetting semplificante restituisce un vettore anzichè un data frame

```
df <- data.frame(a = 1:2, b = 1:2)
str(df[1])

## 'data.frame': 2 obs. of  1 variable:
##  $ a: int  1 2

str(df[[1]])

##  int [1:2] 1 2

str(df[, "a", drop = FALSE])

## 'data.frame': 2 obs. of  1 variable:
##  $ a: int  1 2

str(df[, "a"])

##  int [1:2] 1 2
```

3.4 Subsetting e assegnamento

Tutti gli operatori di subset possono esser combinati con assegnamenti al fine di modificare i valori selezionati:

- la struttura del LHS deve matchare con quella del RHS
- non vi è check per indici duplicati, ad es

```
x <- 1:5
x[c(1,1)] <- 2:3
x

## [1] 3 2 3 4 5
```

- non si possono combinare indici interi con `NA`
- si possono combinare indici logici con `NA`; in questo caso `NA` è trattato come se fosse `FALSE`

```
x[c(T,F,NA)] <- 1
x

## [1] 1 2 3 1 5
```

Il subset con niente può essere utile con le assegnazioni perchè preserva la classe/struttura originale del dato; ad esempio nel seguente caso

```
mtcars[] <- lapply(mtcars, as.integer)
mtcars <- lapply(mtcars, as.integer)
```

nella prima assegnazione `mtcars` rimarrà un data frame mentre nel secondo `mtcars` diviene una lista.
Con le liste e derivati si può utilizzare subsetting più `<- NULL` per eliminare componenti.

Capitolo 4

Controllo del flusso

Le *control structures* (come in altri linguaggi di programmazione) permettono di controllare il flusso dell'esecuzione del programma in base a condizioni in runtime. Le strutture in R più comuni sono:

- `if, else`: per testare una condizione
- `for`: per eseguire un loop un determinato numero di volte
- `while`: eseguire un loop finché una condizione è vera
- `repeat`: eseguire un loop infinito
- `break`: stoppare l'esecuzione di un loop
- `next`: skippare una iterazione di un loop
- `return`: uscire da una funzione

Tipicamente si usano queste strutture all'interno di una funzione (meno nell'esecuzione interattiva).

4.1 Esecuzione condizionale

4.1.1 `if`

La sintassi preferibile è

```
if (condizione) {
    comandi
} else if (condizione) {
    comandi
}
else {
    comandi
}
```

Sia `else if` che `else` non sono obbligatorie da inserire in una struttura `if`; `else if` può esser specificata quante volte si vuole (per gestire diverse situazioni) mentre vi può essere una sola `else`.
Vi sono differenti modi di utilizzare il costrutto `if` in assegnazione (in maniera valida):

```
## x deve essere uno scalare nei casi seguenti, non un vettore
## Primo modo, abbastanza standard.
x <- 1
if (x > 3){
    y <- 10
} else {
    y <- 0
}
y

## [1] 0

## Secondo modo, specifico di R
x <- 4
y <- if(x > 3){
         10
     } else {
         0
     }
```

4.1.2 `switch`

La funzione `switch` fornisce una alternativa compatta a `if`; la sintassi è;

`switch(EXPR, ...)`

Con:

- `EXPR` espressione che verrà valutata
- `...` a specificare le alternative separate da virgola: gli argomenti nella forma `valore = espressione` specificano l'espressione valutata (es chiamate o altro) che deve esser tenuto nel caso in cui la valutazione di `EXPR` risulti `valore`.
  I `valore` (indipendentemente dal fatto che l'esito del test sia di caratteri o numerico) vanno inclusi tra virgolette.
  Nel caso in cui non vi sia un'esito della valutazione previsto, viene restituito `NULL`, a meno che non vi sia un argomento ulteriore che viene eseguito.
  Se l'`azione` associata ad un determinato `valore` non viene specificata viene eseguita la prossima azione specificata

```
switch(1,
       "1" = "a",
       "2" = "b")
```

```
## [1] "a"

# utilizzo con espressione/azione di default se non vi è match
# se non specificata, in assenza di match è restituito NULL
switch("foo",
       "1" = "a",
       "2" = "b",
       "default")

## [1] "default"

## per associare molteplici esiti ad una sola espressione: funziona solo se
## EXPR è stringa
switch("1",
       "1" = , "2" = "1 oppure 2",
       "3" = , "4" = "3 oppure 4",
       "non so")

## [1] "1 oppure 2"
```

4.2 Looping

R presenta tre costrutti utili per creare looping di valutazione:

- `for`
- `while`
- `repeat`

`next` e `break` forniscono controllo addizionale su tali loop.

4.2.1 `for`

La sintassi di `for` é:

```
for (iteratore in sequenza) espressione
```

Se vi sono più istruzioni da eseguire in ogni ciclo, `espressione` deve essere sostituita da un blocco di codice, come segue:

```
for (iteratore in sequenza){
   comandi
}
```

Il funzionamento è il seguente:

- `for` prende una variabile `iteratore` e gli assegna, all'inizio di ogni ciclo, il valore dettato da una `sequenza` (che può esser costituita da un vettore, un array, una lista, o un data.frame). Ad ogni ciclo viene assegnato il valore susseguente. Bisogna prestare particolare attenzione a non sovrascrivere/modificare il valore dell'iteratore nell'espressione o nel blocco di comandi.

- la variabile `iteratore` esiste anche dopo che il loop si è concluso, e presenta il valore dell'ultimo elemento del vettore valutato nel loop.

Nella programmazione di un `for` alcune funzioni primitive molto efficienti possono essere utili:

- `seq_len` crea un vettore di lunghezza predeterminata
- `seq_along` crea un vettore lungo quanto un altro composto di numeri progressivi

```
seq_len(3)

## [1] 1 2 3

month.abb

##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

seq_along(month.abb)

##  [1]  1  2  3  4  5  6  7  8  9 10 11 12
```

4.2.2 `while`

La sintassi è:

```
while (condizione) espressione
```

Anche qui una singola espressione può esser sostituita da un blocco

```
while (condizione) {
   comandi
}
```

`condizione` è una qualsiasi espressione che produce un valore logico (se un vettore di logici verrà considerato solo il primo; se `FALSE` si passa al comando successivo; se `TRUE` viene eseguita l'espressione/blocco, e la condizione viene rivalutata al termine dello stesso. Il comando sarà così rieseguito fino a quando la condizione rimanga `TRUE`.
Un esempio di utilizzo (fittizio):

```
count <- 0
while(count < 10) {
    count <- count + 1
}
count

## [1] 10
```

4.2.3 `repeat`

`repeat` fa si che la valutazione ed esecuzione del comando che lo segue sia continuata sino a che un `break` non sia dato. Il modello è:

```
repeat expr
```

oppure quando (come spesso è) vi sono più istruzioni si sostituisce con un blocco:

```
repeat {
   comandi ...
}
```

4.2.4 `break`

`break` provoca l'uscita dal loop attualmente in esecuzione (viene posto come uno dei suoi statements).
Vediamo un esempio di funzione che conta da 0 fino al numero dato in argomento (che deve esser positivo):

```
counter <- 0
repeat {
    cat(counter," ")
    if (counter < 10) {
        counter <- counter + 1
    } else {
        break
    }
}

## 0  1  2  3  4  5  6  7  8  9  10
```

4.2.5 `next`

`next` fa saltare direttamente alla fase successiva della valutazione degli statements, in un ciclo `for`, `while` o `repeat`.
Nessun comando posto dopo `next` e all'interno del medesimo statements, viene eseguito/valutato.

```
for (i in seq_len(10)) {
    ## Se minore o uguale a 5 salta,
    ## ovvero skippa le prime 5 iterazioni
    if (i <= 5) {
        next
    }
    ## Altrimenti stampa il valore
    cat(i, " ")
}

## 6  7  8  9  10
```

4.2.6 `return`

`return` è funzione che può esser utilizzata per uscire da un loop; è primariamente utilizzata per uscire da una funzione e ritornare un valore specificato tra parentesi alla funzione chiamante.

Capitolo 5

Funzioni

5.1 Introduzione

Le funzioni sono le componenti fondamentali di elaborazione all'interno di R; incapsulano codice per il riutilizzo e costituiscono il fondamento per tecniche più avanzate.
Le funzioni hanno generalmente **tre componenti**[1]:

- il `body()`, il **codice** dentro la funzione
- i `formals()`, la lista di **argomenti formali** (o *formals*) che controlla la chiamata di funzione e specificati in sede di definizione; differiscono dagli **argomenti attuali** (o *current*) che sono specificati dall'utilizzatore in sede di chiamata
- l'`environment()`, mappa della locazione delle variabili della funzione

Insieme agli oggetti costituiscono gli elementi essenziali del linguaggio tanto che per capire il funzionamento di R due slogan sono utili (John Chambers):

- ogni cosa esistente è un oggetto
- ogni cosa che accade è una chiamata di funzione

Ogni operazione in R è una chiamata di funzione, che gli assomigli o meno:

```
x <- 10; y <- 5
`+`

## function (e1, e2)  .Primitive("+")

`+`(x, y)

## [1] 15

`for`(i, 1:2, print(i))
```

[1] Vi fanno eccezione le funzioni primitive, che chiamano codice C attraverso `.Primitive` e non contengono codice R (ad esempio `sum`); queste funzioni hanno `NULL` le tre componenti

```
## [1] 1
## [1] 2

`if`(i == 1, print("Yes"), print("No"))

## [1] "No"

x[3]

## [1] NA

`[`(x, 3)

## [1] NA

{print(1); print(2)}

## [1] 1
## [1] 2

`{`(print(1), print(2)) ## notare che qui mettiamo la virgola

## [1] 1
## [1] 2
```

È anche possibile redefinire queste funzioni, anche se nella maggior parte dei casi costituisce una cattiva idea.

5.2 Definizione

5.2.1 Sintassi

`function` è la parola chiave votata alla creazione di funzioni:

```
nome_funzione <- function(argomenti){
  ## ...
  ## codice R
  ## ...
}
```

Se `nome_funzione` include caratteri alternativamente non accettati come identificatore, è possibile porla tra backticks (ottenibili in Linux mediante `[Alt Gr] + [']`.

5.2.2 Argomenti

In sede di definizione, il singolo argomento può essere specificato come:

- identificatore (e basta)

  ```
  argomento1
  ```

- identificatore con default;

  `argomento2 = default`

  qui `default` è espressione che verrà utilizzato qualore non venga specificato altro nella chiamata della funzione;

- la forma speciale

  `...`

  in questo caso la funzione accetta argomenti ulteriori di quantità non stabilita a priori.

Qualora una funzione necessiti di più parametri debbono essere suddivisi da virgola, come in

`(argomento1, argomento2 = default2, ...)`

Dato che vige la *lazy evaluation* (cfr sez 5.3.2), i valori di default possono essere definiti in termini di altri argomenti:

```
g <- function(a = 1, b = a * 2){
  c(a, b)
}
g()

## [1] 1 2

g(10)

## [1] 10 20
```

5.2.2.1 L'argomento ... e la sua gestione

L'argomento `...` è utilizzato per indicare un numero variabile di argomenti, e in sede di chiamata necessita di *named argument* per i parametri che la seguono (si veda la sezione 5.3.1).
In sede di definizione, `...` è utile:

- quando il numero di argomenti passato in sede di chiamata non può esser conosciuto (in sede di definizione di funzione), come ad esempio

  ```
  str(paste)

  ## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
  ```

- quando si vuole estendere o modificare i default di un'altra funzione, senza copiare l'intera lista degli argomenti della funzione originaria

```r
myplot <- function(x, y, type = "l", ...) {
  plot(x, y, type = type, ...)
}
```

La gestione di `...` all'interno del corpo della funzione, dipende dal fatto che la lista di parametri che memorizza debba essere elaborata o meno. Se:

- deve essere elaborata, un modo comune per gestire gli argomenti `...` è wrapparlo in una lista, come segue:

```r
## Una versione semplice
dots1 <- function(...) list(...)
dots1(a = 1, "x")

## $a
## [1] 1
##
## [[2]]
## [1] "x"
```

- alternativamente se la lista di parametri deve esser utilizzata come input per una funzione interna glie la si può passare direttamente come si è visto in precedenza per la funzione `myplot`.
  Altri, a questo approccio, preferiscono richiedere una lista esplicita di parametri, sotto forma di lista.

5.2.3 Corpo

Il *corpo della funzione* è codice R valido; se composto da più statement, questi sono da contenere tra parentesi graffe.

5.2.4 Ritorno della funzione

La **funzione ritorna** come output un valore alla chiamante, che consiste alternativamente:

1. nel valore dell'ultima espressione valutata (*ritorno implicito*), che non deve essere un assegnamento

2. nell'espressione contenuta nella funzione `return`, che può essere posta a piacere all'interno del corpo della funzione (*ritorno esplicito*)([2])

Un esempio di funzione (che risolve una equazione di secondo grado) che usa entrambi i modi per ritornare:

[2] A livello stilistico hadley preferisce l'uso di default del ritorno implicito, lasciando quello esplicito a condizioni particolari o a uscita veloce dalla funzione

```
eqt2 <- function(a, b, c)
{
  det <- b^2 - 4*a*c
  if (det < 0) {
    ## qui ritorno esplicito
    return(rep(NA, 2))
  } else {
    ## qui ritorno implicito
    c( (- b + sqrt(det)) / (2*a),
       (- b - sqrt(det)) / (2*a))
  }
}

## x^2 + 4x + 4 = 0, (x + 2)^2,  x_{1,2} = -2
eqt2(1, 4, 4)

## [1] -2 -2
```

Qualora una funzione debba restituire più di un risultato, è comune incapsulare il tutto in liste.

5.2.5 L'utilizzo di `on.exit`

Una funzione puo inserire nel suo corpo una chiamata a `on.exit` in qualsiasi punto del proprio corpo, la quale permette di eseguire del codice all'uscita della funzione

```
on.exit(expr)
```

`expr` viene eseguito sia che la funzione esca normalmente che per esito di un errore.
Un'applicazione tipica è cambiare alcuni paramentri/opzioni che si erano modificati durante l'esecuzione della funzione chiamante, per riportarli alla normalità:

```
oldpar <- par(...)
on.exit(par(oldpar))
```

Se in una funzione si fa uso più volte `on.exit`, a partire dalla seconda bisogna aggiungere il parametro `add = TRUE`, per evitare che l'ultima chiamata non sovrascriva le precedenti espressioni da valutare. L'ordine di chiamata delle varie espressioni corrisponde all'ordine con cui sono state impostate:

```
f1 <- function(){
  on.exit(print(getwd()))
  on.exit(print("b"), add = TRUE)
  1
}
f1()

## [1] "/home/l/.sintesi/sintesi_cs/r"
## [1] "b"
## [1] 1
```

5.3 Chiamata e valutazione delle funzioni

5.3.1 Matching degli argomenti

Quando una funzione viene chiamata inizia un processo di valutazione: gli argomenti della funzione vengono incrociati con gli argomenti forniti dall'utente e il corpo della funzione viene valutato sequenzialmente.
La **valutazione** avviene come segue: si parte dall'analisi degli argomenti:

1. per ogni argomento fornito di un nome (es `anni = 4`) il programma ricerca nella lista degli argomenti disponibili se ce ne è uno che coincide perfettamente, e ad esso gli assegna il valore indicato. Questo è il **matching by name esatto**.
2. gli argomenti che non sono andati via al colpo precedente vengono esaminati mediante il matching parziale: se il nome dell'argomento fornito nella chiamata coincide con la prima parte di uno presente nella definizione, allora il valore viene assegnato. Questo è il **matching by name parziale** ed è consigliabile soprattutto per le chiamate in un ambiente interattivo (non in sede di programmazione).
3. ogni valore rimanente viene accoppiato in ordine di presentazione nella chiamata, agli argomenti non denominati nella definizione di funzione. Si chiama **matching by position** o *positional matching*
4. gli argomenti rimanenti vengono assegnati ad `...`, se presente

In sede di chiamata di funzione qualsiasi **argomento che compare dopo `...`** deve essere *named argument*, ovvero nella forma `argom = valore` (deve esser nominato esplicitamente, senza contare sul matching posizionale, ne su quello posizionale), poichè non vi è modo per R di sapere se stiamo passando a `...` o stiamo passando ad un diverso argomento

```
str(paste)

## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)

paste("a", "b", sep=":")  ## a e b vanno in ... sep matcha con sep

## [1] "a:b"

paste("a", "b", se=":")   ## neanche il partial matching funziona!

## [1] "a b :"

paste("a", "b", ":")      ## anche questo ovviamente è sbagliato

## [1] "a b :"
```

5.3.2 Lazy evaluation e promise

Gli argomenti di una funzione vengono valutati solo se/quando necessario. Il principio di **lazy evaluation** fa si che al momento della chiamata di funzione,

le espressioni fornite come argomento vengano solamente *parsate* (ossia ne sia controllata la correttezza sintattica) ma non vengano valutate; questo avverrà solo quando all'interno del corpo della funzione sarà necessario[3].

Formalmente un argomento di funzione viene memorizzato in una **promise**, che può essere pensata come una lista di due componenti:

- l'espressione che da origine al parametro (alla quale si può accedere mediante `substitute`);
- l'environment dove l'espressione deve deve essere valutata

La prima volta che nel corpo della funzione si fa uso del parametro l'espressione della promise viene effettivamente valutata nell'environment appropriato; in merito a questo secondo vi sono alcune sottigliezze da considerare:

- se il parametro è stato specificato dall'utente l'environment di valutazione è l'environment *della chiamante*
- se invece consiste in un valore di default, l'environment per la valutazione coincide con l'environment *della chiamata*

Uno snippet permette di chiarire il punto:

```
x <- 1
f <- function(y = x) {
  x <- 2
  print(y)
}

## Uso del parametro di default: environment di valutazione dell'argomento
## è l'environment della chiamata (dove abbiamo appena definito x <- 2)
f()

## [1] 2

## Uso del parametro specificato dall'utente: environment di valutazione è
## quello da dove è partita la chiamata di funzione, quindi in questo caso
## il globalenv
f(x)

## [1] 1
```

In seguito alla valutazione questo valore è memorizzato in memoria in maniera tale che accessi successivi non provochino una rivalutazione. Tuttavia l'espressione originale continua ad essere accessibile mediante `substitute`.

Come chiaro esempio di lazy evaluation, si consideri:

[3] All'interno della funzione, se si desidera forzare la valutazione del parametro `x`, o lo si valuta come primo o si utilizza l'equivalente `force(x)` (che fa la stessa cosa ma è più esplicito in merito).

```
f <- function(a, b){
    a^2
}
f(a = 2)

## [1] 4
```

Nella sua definizione, non utilizza l'argomento `b`, pertanto la chiamata che si è effettuato non produrrà un errore poichè si passa solamente `a`.
Dalla lazy evaluation derivano quindi che si possono specificare argomenti con espressioni che coinvolgono variabili che debbono ancora esser create (e che vengono create all'interno della funzione, prima che si renda necessaria la valutazione dell'espressione considerata).

5.4 Other tricks, good practices and misc things

5.4.1 Infix functions

La maggior parte delle funzioni sono *prefisse* nel senso che in sede di chiamata, il loro nome viene prima di quello dei parametri. Vi sono però numerose funzioni (es operatori aritmetici) il cui il nome viene posto, in sede di chiamata tra i parametri, ad esempio

```
1:3 %in% 1

## [1]  TRUE FALSE FALSE

`%in%`(1:3, 1)

## [1]  TRUE FALSE FALSE
```

Per definire queste funzioni:

- il nome della funzione in generale deve essere racchiuso tra due %;
- la funzione deve avere due parametri; il primo posto viene interpretato come parametro a sinistra della funzione, mentre il secondo come parametro a destra

```
`%without%` <- function (x, y) x[!(x %in% y)]
```

5.4.2 Replacement functions

Le funzioni di rimpiazzo sono quelle, poste a sinistra di una assegnazione, che permettono di modificare il contenuto dell'oggetto passato come argomento[4].
Sono funzioni ordinarie caratterizzate da:

[4]Si noti che alcune funzioni di rimpiazzo presentano una omonima funzione per la semplice stampa dei valori, come `rownames` e `rownames<-` (ma non sono la stessa funzione).

- nome termina con il token di assegnamento `<-` ed è contenuto fra backticks;
- tipicamente hanno due parametri, `x` e `value`; il primo indica l'oggetto che viene modificato, gli altri vengon utilizzati all'interno della funzione;
- debbono ritornare il valore modificato.

Un esempio in cui i valori superiori a un cutoff, li impostiamo ad `Inf`

```
`cutoff<-` <- function(x, value){
    x[x > value] <- Inf
    x
}
y <- 1:4
cutoff(y) <- 2
y

## [1]   1   2 Inf Inf
```

Un esempio con tre parametri dei quali ne usiamo 2 è una funzione che mette a posto dei missing non impostati a `NA`

```
`modify<-` <- function(x, position, value){
  x[position] <- value
  x
}
x <- 1:10
## la chiamata si modifica in questo modo
modify(x, 1) <- 10
x

##  [1] 10  2  3  4  5  6  7  8  9 10
```

5.4.3 Funzioni vettorizzate

Sono quelle funzioni che, se ricevono come argomento un vettore, restituiscono, dopo l'elaborazione, un vettore della medesima lunghezza (es `sin`); nell'implementazione di funzioni generiche bisognerebbe perseguire questa convenzione. Facciamo un esempio scrivendo una funzione che restiuisce 1 se il valore passatogli (scalare o vettore) è maggiore di 0, 0 se uguale a 0, e -1 altrimenti;

```
cmp0 <- function(value)
{
  if( any(great <- value > 0 ) )
    value[great] <- 1
  if( any(equal <- value == 0 ) )
    value[equal] <- 0
  if( any(less <- value < 0 ) )
    value[less] <- -1
```

```
  value
}
cmp0(2)

## [1] 1

cmp0(c(1,2,0,-4))

## [1]  1  1  0 -1
```

5.4.4 L'uso di `match.arg`

Spesso utile avere una struttura che permetta all'utente di scegliere, mediante matching parziale, fra una serie di opzioni predefinite, e che eviti che questo specifichi opzioni sbagliate. Un template è il seguente:

```
which.side <- function(sides = c("mine", "yours", "the.truth"))
{
  side <- match.arg(sides)
  side
}
which.side()      ## di default viene matchato il primo elemento

## [1] "mine"

which.side("mi")  ## matcha mine

## [1] "mine"

which.side("the") ## match the.truth

## [1] "the.truth"

which.side("in")  ## non matcha niente, da errore e un suggerimento

## Error in match.arg(sides):  'arg' should be one of "mine", "yours",
"the.truth"
```

5.4.5 `NULL` e `getOption` per parametri di default

Due idee:

- se specifichiamo un parametro come di default a `NULL` invece che lasciarlo vuoto; a livello di logica equivale a dire che non si specifica nulla di default, ma facendo così possiamo utilizzare funzioni come `is.null` per indirizzare l'elaborazione nel caso l'utente non specifichi nulla[5].

[5] Alternativamente si può lasciarlo senza default e utilizzare `missing` ma il tutto risulta meno compatto/intelleggibile

```
f <- function(a, b = NULL){
  ## ... R code
}
```

- in sede di programmazione (soprattutto di pacchetti) è possibile lasciare spazio alla configurazione dell'utente di alcuni parametri di default attraverso le opzioni globali, e ciò è ottenuto mediante `getOption` (che consiste in un wrapper ad `option`). Un esempio viene fornito per l'argomento `table.placement` della funzione `xtable`

  ```
  xtable(...,
         table.placement = getOption("xtable.table.placement", "ht"),
         ...)
  ```

  `getOption` va a vedere se l'utente ha una option corrispondente a `xtable.table.placement`: se cosi è restituisce tale valore, alternativamente fornisce quello dato come default, in questo caso `ht`.

Capitolo 6

Scoping, environments, binding

6.1 Scoping

Lo scoping consiste nell'insieme di regole che gestiscono come il valore associato ad un nome venga reperito durante l'elaborazione. Comprendere lo scoping permette di:

- costruire strumenti attraverso la composizione di funzioni (programmazione funzionale)
- bypassare le regole di valutazione usuali e fare non-standard evaluation

Ad esempio consideriamo la seguente funzione:

```
f <- function(x) {
    y <- 2 * x
    print(x)
    print(y)
    print(z)
}
```

In questa:

- x è un **parametro**; in sede di esecuzione il suo valore è dato dal parametro specificato in chiamata di funzione;
- y è una **variabile locale**, ovvero il suo valore è determinato dalla valutazione di un'espressione all'interno della funzione;
- z è una **free variable**, considerabili come rimanenti rispetto alle altre due

In generale si possono identificare differenti scoping rules, che si differenziano soprattutto in relazione alla gestione delle free variables:

- **dynamic scope**: il valore associato a una free variable è determinato dalla ricerca progressiva negli environment delle funzioni che hanno chiamato la presente

- **lexical scope**: il valore di una free variable è ricercato nell'environment nel quale la funzione è stata definita

R utilizza prevalentemente il *lexical scoping* (similmente a Scheme, Perl, Python, Common Lisp); pertanto la ricerca dei valori associati a simboli è basata su come le funzioni erano nested al momento della definizione, non a quello della chiamata. Non è necessario conoscere come la funzione è stata chiamata per capire dove risieda il valore di una free variable: basta guardare la rispettiva definizione. Ad esempio:

```
x <- 1
h <- function(){
  y <- 2
  i <- function(){
    z <- 3
    c(x, y, z)
  }
  i()
}
h()

## [1] 1 2 3
```

Le medesime regole si applicano alle closures, ovvero funzioni create da altre funzioni (si vedranno in seguito, qui approfondiamo solo come interagiscono con lo scoping). Ad esempio la seguente funzione `j` ritorna una funzione:

```
j <- function(x) {
  y <- 2
  function(){
    c(x, y)
  }
}
k <- j(1)
k()

## [1] 1 2
```

Questo funziona perchè a `k` viene assegnata una funzione ma viene al contempo preservato l'environment nel quale essa è stata definita, e tale environment include il valore di `y`.

6.2 Environments

Gli environments sono le strutture di dati che, nel linguaggio, rendono possibile lo scoping. Possono essere utili anche di per se per specifiche applicazioni dato che hanno reference semantics: quando si modifica un binding di un environment, questo non viene copiato ma modificato direttamente

6.2.1 Introduzione

Il compito di un environment è di fornire un associazione di nomi/identificatori a valori (residenti da qualche parte in memoria).

In linea di massima un environment è simile ad una lista, con alcune eccezioni:

- gli oggetti di un environment hanno un **nome univoco** (due oggetti non possono avere lo stesso nome) e **non sono ordinati** (quindi ad esempio non hanno senso scritture che fanno uso di indici numerici)
- gli environment (ad eccezione dell'empty environment) hanno un genitore (**parent environment**). Il parent environment è utilizzato per implementare il **lexical scoping**[1].
- gli environment hanno **reference semantics**

Più tecnicamente l'environment è composto da **due elementi**:

- il **frame**: contiene il binding name-object[2]. E' possibile elencare i binding disponibili in un dato momento utilizzando `ls`.
- il **parent environment**. E' indagabile con la funzione `parent.env`.

Vi sono quattro environment speciali (e relative funzioni):

- `globalenv()`, il global environment, è il workspace dove l'utente generalmente lavora. Il parent dell'environment globale è l'ultimo oggetto di cui si è fatto l'attach (solitamente un pacchetto, attraverso le funzioni `library` o `require`)
- `baseenv()`, o base environment, è l'environment del pacchetto base. Il suo genitore è l'empty environment
- `emptyenv()`, l'empty environment, è il progenitore finale di tutti gli environment creati e non ha binding nel proprio frame
- `environment()` è l'environment corrente

La funzione `search` lista tutta la progenie del global environment; questo è chiamato search path perchè gli oggetti richiesti dal global environment possono esser trovati ispezionando in ordine i frame di questi environment[3].

```
search()
```

```
##  [1] ".GlobalEnv"        "package:boot"      "package:survival"  "adf"
##  [5] "package:knitr"     "tex2pdf"           "package:stats"     "package:graphics"
##  [9] "package:grDevices" "package:utils"     "package:datasets"  "package:methods"
## [13] "Autoloads"         "package:base"
```

[1] Se un nome non è trovato in un environment, viene ricercato nel parent environment, poi (se non trovato) nel parent del parent e così via

[2] Sfortunatamente il termine frame è utilizzato incoerentemente qua e la in R: ad esempio `parent.frame` non fornisce il frame del genitore di un environment ma restituisce l'environment che ha fatto la chiamata che ha creato il corrente environment

[3] `Autoloads` è un environment particolare che serve per risparmiare memoria e viene utilizzato per caricare oggetti di grandi dimensioni dai pacchetti (ad esempio dataset) solo quando sono necessari

6.2.2 Operazioni comuni

Per **creare un environment** si utilizza la funzione `new.env`, per **assegnare dei binding** si procede mediante assegnazione similmente a quanto avviene alle liste con nomi

```
e <- new.env()
e$a <- FALSE
e$b <- "a"
e$c <- 2.3
e$d <- 1:3
```

Per **elencare gli elementi** di un environment si possono utilizzare `ls` ed in particolare `ls.str`

```
ls(e)

## [1] "a" "b" "c" "d"

ls.str(e)

## a :  logi FALSE
## b :  chr "a"
## c :  num 2.3
## d :  int [1:3] 1 2 3
```

Dato un nome, è possibile **estrarre il valore** con `$`, `[[` o `get`:

- i primi due ritornano `NULL` se quel nome non ha associazione
- `get` (di default, attraverso il parametro `inherits = TRUE`) usa le regole di scoping, risalendo negli environments parents fino a che non trova un valore associato a tale nome

```
a <- new.env()
a$b <- 1
a$c <- 2
x <- 3

a$b

## [1] 1

a[["x"]]

## NULL

get("x", envir = a)

## [1] 3
```

Per **eliminare oggetti** da un environment, a contrario di una lista dove gli si può assegnare `NULL`, occorre utilizzare `rm`

```
rm("c", envir = a)
ls(a)

## [1] "b"
```

Per **determinare se un binding esiste** usare `exists`

```
exists("c", envir = a, inherits = FALSE)

## [1] FALSE
```

Si è utilizzato `inherits = FALSE` perchè alternativamente avrebbe cercato anche nei parent environments, trovando la funzione `c` nel environment base. Infine per `comparare degli environments` bisogna utilizzare `identical`, non `==`

```
identical(environment(), globalenv())

## [1] FALSE
```

6.2.3 Chiamata di funzioni ed environment

La maggior parte degli environment non viene creata mediante `new.env`, ma ciò avviene in automatico come conseguenza della chiamata di funzioni. Questa proprietà non si estende ad una qualsiasi blocco

```
a <- 1
b <- 2
environment()

## <environment: 0x556a78e2ab40>

{
  b <- 1
  print(environment())
}

## <environment: 0x556a78e2ab40>

b

## [1] 1
```

Vi sono **quattro tipi di environment** associati ad una funzione: enclosing, binding, execution e calling.

Enclosing environment è l'environment nel quale la funzione è stata creata/definita. E' l'unico environment che, a contrario degli altri, è unico; ovvero per ogni funzione vi è uno e un solo enclosing environment (mentre vi possono esser 0, 1 o n binding execution e calling environment).

Per conoscere l'enclosing environment di una funzione si passa alla funzione environment il nome della funzione

```
f <- function() 1
environment(f)

## <environment: 0x556a78e2ab40>

environment(sd)

## <environment: namespace:stats>
```

Binding environments E' l'insieme di environment che hanno un binding alla funzione. Nel caso di `f` prima l'enclosing environment coincide con il binding environment, ma ciò non avviene se si assegna la funzione in un environment differente

```
e <- new.env()
e$g <- function() print(parent.env(environment()))

g

## Error in eval(expr, envir, enclos):  oggetto 'g' non trovato

with(e, g())

## <environment: 0x556a78e2ab40>
```

In questo caso binding environment è `g` mentre enclosing environment è `globalenv`. Nel primo caso si ha errore perchè cerchiamo di chiamare la funzione da globalenv che non è binding environment; al contrario nel secondo chiamiamo la funzione dal corretto environment (il binding) quindi il tutto funziona (e la funzione stampa il parent environment, che è globalenv).
La **distinzione tra binding ed enclosing** environment è rilevante per i namespace dei pacchetti che sono implementati facendo uso degli environment, beneficiando del fatto che una funzione non deve necessariamente risiedere (ovvero avere binding) nel proprio enclosing environment.
Nello specifico ogni pacchetto ha due environment associati:

- il **package environment**: contiene i binding per ogni funzione esportata ed è piazzato nel search path
- il **namespace environment**: contiene i binding per qualsiasi funzione (esportata o interna) ed ha come parent environment uno special *import environment* che contiene tutti i binding che le funzioni del pacchetto necessitano (funzioni di altri pacchetti)

Ogni funzione esportata da un pacchetto è bindata/chiamabile nel package environment (che si trova nel search path) ma ha come enclosing environment il namespace environment: la ricerca di una free variable di un pacchetto avviene prima facendo riferimento al namespace environment e poi al global environment e da li a tutto il search path. In questo modo, attraverso il namespace environment, si riesce a preservare il funzionamento del pacchetto.

Execution environment Consiste nell'environment che viene creato ad ogni chiamata e che contiene tutte le variabili locali (es parametri); il parent (ovvero dove si andrà a cercare eventuali free variable) è l'enclosing environment della funzione chiamata (ovvero dove questa è stata definita).
Al termine dell'esecuzione l'execution environment è solitamente eliminato.

Quando però si crea/restituisce una funzione dall'interno di un'altra, l'enclosing environment della funzione restituita è l'execution environment della funzione più esterna e questo ultimo non viene cancellato quando la chiamata termina perchè può servire in seguito per le chiamate della funzione restituita.
Questa peculiarità serve per creare **function factory** ovvero funzioni generali che creano funzioni più specifiche: questo apre nuove possibilità di programmazione, che esemplifichiamo con il caso seguente

```
make_power <- function(n) {
  function(x) x^n
}

cube <- make_power(n=3)
square <- make_power(n=2)
cube

## function(x) x^n
## <environment: 0x556a7fd041b0>

square

## function(x) x^n
## <environment: 0x556a7fd53270>

cube(3)

## [1] 27

square(3)

## [1] 9
```

`n` è una free variable per `cube` o `square`; tuttavia è definita nell'execution environment di `make_power` (a seconda di cosa gli verrà passato come argomento) che costituisce parent environment per cube o square.
Visto che `n` non è trovata nell'execution environment di cube o square (dove è free variable), la sua ricerca proseguirà nel parent environment.
Quindi di fatto abbiamo definito una funzione `make_power` che è di fatto in grado di costruire tanti altri tipi di funzioni

Calling environment Consiste nell'environment dal quale la funzione è stata chiamata. Possiamo avere accesso a questo environment utilizzando la mal denominata funzione `parent.frame` (che appunto ritorna l'environment nel quale la funzione è stata chiamata).

6.2.4 Reference semantics ed utilizzi connessi

A parte essere la struttura di dati sulla quale è basato lo scoping, gli environment possono essere utili come strutture in sè in quanto sono caratterizzate da reference semantics: quando sono passate ad una funzione non viene fatta una copia ma viene fornito un puntatore con il quale modificare l'oggetto passato (e l'attività di passaggio alla funzione chiamata risulta più efficiente).

Una dimostrazione della reference semantics:

```
f <- function(x) x$a <- 1
e <- new.env(parent = emptyenv())
f(e)
ls.str(e)

## a :  num 1
```

Si nota dunque che l'environment `e` passato alla funzione ne esce con l'elemento assegnato.
Nel caso si voglia usare environment per il passaggio a funzioni (ad esempio al posto di liste) è opportuno impostare il suo parent all'empty environment, come è stato fatto nell'esempio (questo ci assicura che non ereditiamo dal current environment o qualche progenitore dei binding, ad esempio se utilizziamo la funzione `get`). Un esempio:

```
x <- 1
e1 <- new.env()
get("x", envir = e1)

## [1] 1

e2 <- new.env(parent = emptyenv())
get("x", envir = e1)

## [1] 1
```

Gli environment sono utili per risolvere tre problemi comuni:

- evitare copie di dati di grandi dimensioni
- gestire lo stato in un pacchetto
- avere una struttura assimilabile ad una hash table che permette la memorizzazione e ottenimento rapido di coppie key - value

Approfondiamo il secondo in quanto segue.

Stati di un pacchetto Generalmente gli oggetti di un pacchetto sono bloccati nel senso che non è possibile modificarli direttamente Tuttavia se in un pacchetto è necessario mantenere uno stato tra chiamate di funzione si può fare qualcosa del genere:

```
my_env <- new.env(parent = emptyenv())
my_env$a <- 1

get_a <- function(){
  my_env$a
}

set_a <- function(value) {

  old <- my_env$a
  my_env$a <- value
  invisible(old)

}
```

Ritornare il valore vecchio è una buona pratica perchè rende piu semplice risettare al valore precedente assieme ad `on.exit`.

6.3 Binding

Il binding è l'attività di creare una associazione tra un nome e un valore; R è un linguaggio estremamente ricco da punto di vista della possibilità di cosa di può bindare (oggetti, funzioni, espressioni ecc) e in quale environment. Il binding si effettua tipicamente mediante

`<-` l'assegnamento standard, effettua il binding nell'environment corrente;

`«-` non local assignment: l'assegnamento non crea mai una variabile nel corrente environment bensì modifica una variabile esistente risalendo progressivamente nei parent environments[4]. Se non trova un nome che corrisponda crea una nuova variabile nel globalenv. L'operatore è solitamente utilizzato con una closure

[4] Di fatto `name «- value` equivale a `assign("name", value, inherits = TRUE)`.

Capitolo 7

Defensive programming, condition handling, debugging

Questa parte tratta di come evitare problemi comuni prima che occorrano (defensive programming), di come segnalare e gestire situazioni inattese (condition e condition handling) e come mettere a posto i bug (debugging).

7.1 Programmazione difensiva

Nella scrittura di funzioni, la programmazione difensiva prescrive di fallire presto e in maniera rumorosa; alcuni accorgimenti per realizzarla in R:

- essere restrittivi sull'input accettato dalla funzione; implementabile mediante `stopifnot` che verifica un'asserzione ovvero una affermazione che deve essere necessariamente vera
- evitare funzioni che usano non standard evaluation (come `transform`, `subset` e `with`). Queste funzioni permettono di risparmiare tempo nell'utilizzo interattivo, ma dato che fanno assunzioni in maniera tale da ridurre la diteggiatura, quando falliscono, spesso falliscono con messaggi d'errore poco informativi
- evitare funzioni che ritornano tipi differenti di output in base all'input fornito. I due rischi maggiori stanno nell'uso di `[` e `sapply`:
  - quando si fa il subset di un `data.frame` in una funzione bisognera sempre porre, per continuare ad avere un `data.frame`
  - invece di `sapply` utilizzare la più restrittiva `vapply` che da errore se l'input è di tipo incorretto e ritorna il tipo corretto di output anche nel caso di input di lunghezza nulla

7.2 Conditions e conditions handling

In generale, R ha alcuni modi per riportare informazioni sull'esecuzione delle funzioni: errori, warning e messaggi.

Queste tre situazioni sono considerate **conditions**, il quale è un concetto generico per indicare che qualcosa (di solitamente inatteso) è occorso.

7.2.1 Conditions

Errori L'*error* è solitamente un errore di più "grave" entità, per cui la funzione si blocca e non completa la propria esecuzione. Sono creabili mediante la funzione `stop`; questa termina l'azione della funzione, e solleva errore ritornando il controllo al livello superiore.
Ad esempio un modo comune per comunicare all'utente che deve inserire obbligatoriamente alcuni argomenti

```
f1 <- function(n, x = stop("x deve esser specificato")){
  n + x
}
f1(n = 1)

## Error in f1(n = 1):  x deve esser specificato
```

Warning Il *warning* è una indicazione che qualcosa di inusuale, ma non necessariamente fatale, è successo: comunque la funzione è in grado di portarre a termine il proprio compito pertanto l'esecuzione continua. Un esempio

```
log(-1:2)

## Warning in log(-1:2):  Si è prodotto un NaN

## [1]       NaN      -Inf 0.0000000 0.6931472
```

I warning sono generati dalla funzione `warning` omonima, e vengono mostrati al termine/uscita dalla funzione; warning non stoppa l'esecuzione della funzione. Un wrapper attorno alla funzione `mean` come esempio:

```
mean2 <- function(x) {
  if (any(is.na(x))) {
    warning("Missing values in x; na.rm = TRUE below")
    mean(x, na.rm = TRUE)
  } else {
    mean(x)
  }
}

x <- c(1,2,NA)
mean2(x)

## Warning in mean2(x):  Missing values in x; na.rm = TRUE below

## [1] 1.5

x <- 1:4
mean2(x)

## [1] 2.5
```

La funzione `warning`, si comporta differentemente a seconda dell'opzione warn (`options("warn")`): per maggiore controllo sulla funzione si veda `?warning`.
Per **silenziare i warning** derivanti da una espressione si usi `suppressWarnings`:

```
suppressWarnings(expr)
```

Messaggi Un *message* è una notificazione/diagnostica generica prodotta dalla funzione `message`; l'esecuzione della funzione continua.
Per **silenziare i messages** derivanti da una espressione si usi `suppressMessages`:

```
suppressMessages(expr)
```

7.2.2 Condition handling

Le conditions possono esser gestite: le due funzioni che permettono di fare questa cosa sono `try` e la più generale `tryCatch`[1].

- `try` permette di continuare l'esecuzione anche qualora occorra un errore; è implementata utilizzando `tryCatch`
- `tryCatch` permette di specificare/definire funzioni (dette handler) che vengano chiamate qualora si rilevino le conditions prescelte. Nel seguito approfondiamo questa

`tryCatch` è uno strumento generico per la gestione delle conditions: oltre ad errors warnings e messages può gestire gli interrupts (quando l'utente pigia `Ctrl + C`). Attraverso `tryCatch` si mappa condizioni a relativi handlers:

- si definiscono funzioni con il nome della condition cui deve rispondere; i più standard sono `error`, `warning`, `message`, `interrupt` e `condition` (questa viene attivata per qualsiasi condition).
  Ad esempio nel seguito

```
show_conditions <- function(code) {
  tryCatch(code,
           error   = function(x) c(y = "Errore", x),
           warning = function(x) c(y = "Warning", x),
           message = function(x) c(y = "Message", x))
}
```

  Come unico argomento delle funzioni viene passata la condition, gestita come lista di due elementi (messaggio di errore o altro più la chiamata che l'ha generato; spesso ci interessa soprattutto il primo)

- se una condition è segnalata, `tryCatch` chiamerà l'handler il cui nome matcha

```
show_conditions(arcsin(2))
```

[1] `withCallingHadlers` è un'altra funzione ma probabilmente meno utile, cfr Hadley's Advanced Programming a pag 165.

```
## $y
## [1] "Errore"
##
## $message
## [1] "non trovo la funzione \"arcsin\""
##
## $call
## arcsin(2)

show_conditions(log(-1))

## $y
## [1] "Warning"
##
## $message
## [1] "Si è prodotto un NaN"
##
## $call
## log(-1)

show_conditions(message("Hello World!"))

## $y
## [1] "Message"
##
## $message
## [1] "Hello World!\n"
##
## $call
## message("Hello World!")
```

- se invece tutto va bene `tryCatch` restituisce il valore valutato

```
show_conditions(1+2)

## [1] 3
```

Gli handler possono fare di tutto ma tipicamente ritornano un valore o creano un messaggio di errore maggiormente informativo (elaborando/arricchendo il `message` disponibile).

7.3 Debugging

7.3.1 Funzioni utili

Nel caso siano necessarie modifiche al codice di una funzione poichè non produce risultati sperati, R mette a disposizione alcune funzioni per il debug;

- `traceback` individua quale funzione ha dato *error* stampando il call stack della funzione dopo che è accaduto un errore. Se non vi è errore non fa nulla
- `browser` sospende l'esecuzione di una funzione nel punto in cui è chiamata e pone l'esecuzione in modalità debug (eseguendo una linea alla volta)
- `debug` flagga una funzione per la modalità debug, facendo si che qualunque volta sia eseguita, venga eseguita interamente una linea alla volta
- `trace` permette di inserire debugging code in una specifica posizione di una funzione, senza editare la funzione stessa (utile se si sta debuggando codice di qualcun'altro)
- `recover` permette di modificare il comportamento in caso di errore (di default viene stampato il messaggio di errore e si esce dalla funzione) se utilizzato come error handler. Fa si che qualsiasi volta accada un errore nella funzione, piuttosto che ritornare alla console l'interprete interromperà l'esecuzione proprio dove è accaduto l'errore, printerà il call stack (in maniera tale da capire quanto siamo profondi nello stesso) e poi sarà possibile muoversi nel call stack per esaminare gli environment.

traceback In caso di *error* ricevuto da una funzione può tornare utile la funzione `traceback` la quale, stampando lo stack delle chiamate di funzioni, permette di individuare qual'è la funzione che ha avuto problemi

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)

## Error in "a" + d:  argomento non numerico passato ad un operatore
binario
```

Usando traceback immediatamente dopo si ottiene

```
> traceback()
4: i(c) at #1
3: h(b) at #1
2: g(a) at #1
1: f(10)
```

Un altro esempio:

```
a <- mean(rnorm(runif()))

## Error in runif():  argomento "n" assente, senza valore predefinito
```

```
> traceback()
3: runif()
2: rnorm(runif())
1: mean(rnorm(runif()))
```

Comando	Interpretazione
`n`	Next: esegui la prossima linea[3]
`s`	Step into: come n, ma se una funzione entra e debugga anche quella funzione
`f`	Finish: termina l'esecuzione del loop o della funzione quella funzione
`c`	Continue: continua l'esecuzione sino al termine della funzione
`where`	mostra il *call-stack* (simile a `traceback`)
`Q`	Stop: stoppa il debugging ed esci

Tabella 7.1: Comandi speciali per `browser`.

In generale la funzione che ha restituito *error* è quella in cima allo stack (l'ultima chiamata) delle chiamate di funzioni rappresentato da `traceback`.
È necessario chiamare traceback immediatamente dopo che l'errore si è verificato: se si esegue altro codice, non funziona perchè la prossima funzione impiegata risetta la call stack a zero.
`traceback` è utile perchè permette di individuare subito il punto di errore ma non spiega il perchè dell'errore stesso, per il quale tornano utili altri strumenti .

browser Serve per creare una sorta di breakpoint all'interno di una funzione: una chiamata a `browser` interrompe l'esecuzione della funzione nel punto specificato e permette l'ispezione dell'environment da cui viene chiamato.

```
func <- function(...) {
    ## qui il codice è eseguito senza interruzione
    browser()
    ## qui il codice è eseguito a step
}
```

Può esser utile soprattutto se si sa a priori il punto in cui una funzione può presentare dei problemi; alternativamente impostare

```
options(error = browser)
```

che fa si di bloccare l'esecuzione ed evocare browser laddove si verifica un errore[2].
Una volta chiamata, viene attivato un prompt speciale:

```
Browse[1]>
```

Da questo prompt, l'utente può inserire *qualsiasi codice R* (stampa valori assegnati, creazione nuovi oggetti, altri comandi R) o alcuni *comandi specifici* (tab 7.1.

debug Se si desidera eseguire passo passo una funzione, entrandone nell'environment per controllarne più direttamente il funzionamento occorre utilizzare `debug`, che prende in input la funzione (ad esempio `debug(funz)`): equivale a chiamare `browser` nella prima linea della funzione.
Un altro utilizzo utile di debug è per finalità didattiche/esplorative, es `debug(lm)`.
Per evitare che, all'interno della stessa sessione, ogni volta che si chiami la funzione essa entri in modalità debug, occorre una volta finito comandare `undebug`

[2] `options(error = NULL)` per de-impostare l'handler della situazione di errore

sulla funzione. In alternativa per debuggare solamente la prima esecuzione della funzione utilizzare `debugonce`. `debug` può anche esser chiamato dall'interno di una funzione, per debuggare ad esempio una seconda funzione da questa chiamata.

recover È possibile impostare questa funzione come handler di errore default a livello globale mediante

```
options(error = recover) # per impostarla
options(error = NULL)    # per tornare a opzioni di default
```

Se impostata, quando avviene un errore ci viene stampato la call stack e ci viene fornito un menu dal quale scegliere

```
> read.csv("nosuchfile") ### <--- file inesistente
Warning in file(file, "rt") :
  cannot open file 'nosuchfile': File o directory non esistente
Errore in file(file, "rt") : non posso aprire questa connessione

Enter a frame number, or 0 to exit

1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote =
3: file(file, "rt")
```

Scegliendo 1, 2 o 3 si può andare in Browse nell'environment di quella funzione per esaminare gli oggetti. Si può così analizzare tutti gli oggetti presenti nell'environment del call stack.
La stampa dello stack avviene in maniera invertita rispetto a quella di traceback, quindi l'environment dal quale si è generato l'errore è l'ultimo tra quelli listati.

Se si desidera procedere con recover anche nel caso di warning, non solo di error, occorre modificare l'opzione `warn` impostandola a 2 (a 1 di default)

```
options(warn = 2, error = recover) # warning come error gestiti con recover
options(warn = 1)                  # ritorno a impostazioni di default
```

In seguito anche un eventuale `warning` verrà gestito con `recover`

dump.frames `dump.frames` è una equivalente a recover per situazioni non interattive: crea `last.dump.rda` file nella working directory attuale. In seguito si può caricare il file e usare debugger per entrare in debug mode analogamente a `recover` (Advanced R, pag 157).

7.3.2 Ispezionare lo stack delle chiamate di funzioni

Una volta che siamo in debug mode, quali funzioni possono essere utili per esplorare dove siamo, come ci siamo arrivati e cosa c'è? In larga parte si applicano le funzioni rivolte agli environments.
Una volta localizzato dove si presume sia l'errore (ad esempio mediante `traceback`) si può porre `browse` (o equivalente) prima della linea incriminata per bloccare la valutazione e poter ispezionare lo stack

```
f <- function(){
  x <- 2
  g()
}

g <- function() {
  x <- 1
  h()
}

h <- function(){
  x <-3
  browser()
  stop()
}

g()
```

Ad ogni environment facente parte dello stack viene fornito un progressivo numerico nell'ordine di creazione (`.Globalenv` è indicato con il numero 0). Il numero di environment nel corrente stack è dato da `sys.nframe`.
Una volta bloccata la valutazione abbiamo a disposizione alcune funzioni che forniscono accesso agli environment;

- `sys.calls` fornisce la lista delle chiamate effettuate (parametri specificati compresi) sino a quella attuale. `sys.call` una chiamata (di default l'attuale):

```
!Browse[2]> sys.calls()
 [[1]]
 f()

 [[2]]
 g()

 [[3]]
 h()
```

- `sys.frames` fornisce i rispettivi environments sempre nell'in ordine. A contrario `sys.frame` ne fornisce solamente uno (fornendogli il progressivo numerico)

```
!Browse[2]> sys.frames()
 [[1]]
 <environment: 0x2c2e0e8>
# <- environment creato per la chiamata di f

 [[2]]
 <environment: 0x2c2e190>
 # <- environment creato per la chiamata di g

 [[3]]
 <environment: 0x2c2e238>
  # <- environment creato per la chiamata di h
```

- `sys.function` serve per listare il codice delle funzioni con indice nullo (codice della funzione del presente environment) o negativo (ad esempio `-1` codice della funzione che ha chiamato il presente environment

```
!Browse[2]> sys.function(-1)
 function() {
   x <- 1
   h()
 }
```

Per listare genericamente il contenuto di tutti gli environment possiamo comandare

```
lapply(sys.frames(), ls.str)
```

Per ispezioni più specifiche su un singolo environment la cosa più semplice è spostarvicisi, comandando `recover` e scegliendo l'environment desiderato.

7.3.3 Debug per `warning` e `message`

Le funzioni presentate in precedenza sono orientate prevalentemente ad intervenire in caso di errore. Vi sono però altri modi in cui una funzione può fallire:

- una funzione potrebbe generare un warning inatteso. Il modo più semplice di gestirli in sede di debug è di convertirli in errore mediante

  ```
  options(warn = 2)
  ```

  ed utilizzare i metodi di debugging gia visti (si può ignorare in questo caso alcune chiamate extra presenti nello stack, come `doWithOneRestart withOneRestart withRestart` e `.signalSimpleWarning`)

- una funzione potrebbe generare un messaggio inatteso. Per gestire la cosa (e trasformarlo in errore) possiamo fare una cosa del genere

```
message2error <- function(code){
  withCallingHandlers(code, message = function(e) stop(e))
}

f <- function() g()
g <- function() message("Hi")
g()

## Hi

message2error(g())

## Hi
```

  così da poter proseguire dopo con `traceback` e localizzare i punti problematici.

Capitolo 8

Functional Programming

R è al suo cuore un **linguaggio di programmazione funzionale**; ciò significa che fornisce molti strumenti per la creazione e manipolazione di funzioni.
In particolare in R le funzioni sono **first-class objects**; vi si può fare le stesse cose che è possibile fare con altri oggetti, ovvero ad esempio

- assegnarle a variabili
- memorizzarle in liste
- passarle come argomento ad altre funzioni
- crearle all'interno di altre funzioni (funzioni nested)
- restituirle come risultato di un'altra funzione

8.1 Elementi costitutivi

8.1.1 Anonymous functions

L'unità più semplice alla base di tutta la programmazione funzionale è la funzione. In R le funzioni sono oggetti di per se; non necessariamente sono associati ad un nome, ne R ha una sintassi speciale per farlo. Una funzione alla quale non è stato associato un nome è detta **funzione anonima**; si creano utilizzando `function` ma senza effettuare una assegnazione

```
sapply(mtcars, function(x) length(unique(x)))
```

```
##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   25    3   27   22   22   29   30    2    2    3    6
```

Un esempio di utilizzo diretto di una funzione anonima è il seguente (calcolo *naive* della media): nella prima parentesi abbiamo la definizione della funzione, attraverso seconda effettuiamo la chiamata e forniamo i parametri.

```
( function(x) sum(x)/length(x) )(1:10)
```

```
## [1] 5.5
```

Le applicazioni maggiori delle funzioni anonime si hanno con tutte le tecniche rese disponibili dalla programmazione funzionale, ovvero

- **closures**: funzioni che restituiscono funzioni
- **functionals** funzioni che ricevono in input funzioni e le applicano a qualcosa
- **function operators**: funzioni che prendono in input funzioni e restituiscono funzioni

8.1.2 Closures

Le closures sono funzioni associate ad un environment, dal quale possono accedere a dati[1]. In R quasi tutte le funzioni sono closures: vi fanno eccezione le funzioni primitive che chiamano codice C e non hanno un environment associato.
Nell'ambito della programmazione funzionale, una closure si ottiene tipicamente dalla chiamata di una funzione che restituisce un'altra funzione (la quale memorizza l'execution environment della chiamante). Le closures sono utili per

- creare function factories (funzioni che creano funzioni)
- gestire stati mutabili associati ad una funzione

8.1.2.1 Function factories

Riprendendo la funzione `make_power`, essa è una function factory ovvero una funzione che serve per creare funzioni. In questo contesto abbiamo concettualmente due livelli di parametri da gestire:

- quelli della funzione chiamante: servono tipicamente per settare il comportamento della restituita
- quelli della funzione restituita: servono per lasciare la personalizzazione all'utente in sede di chiamata finale (ad esempio per fornire i dati da processare)

```
make_power <- function(n) {
  function(x) x^n
}
cube <- make_power(n = 3)
square <- make_power(n = 2)
```

Quando si stampano le relative closures non si hanno informazioni illuminanti

```
cube

## function(x) x^n
## <environment: 0x556a7dc599f8>
```

[1] Da questo il detto che *An object is data with functions; a closure is a function with data*

```
square

## function(x) x^n
## <environment: 0x556a7dca6c08>
```

Quello che cambia è l'enclosing environment che è l'environment associato alla singola chiamata che ha restituito la funzione memorizzata (rispettivamente `cube` e `square`). Un modo per conoscere il contenuto dell'environment è coercirlo a lista:

```
as.list(environment(square))

## $n
## [1] 2

as.list(environment(cube))

## $n
## [1] 3
```

Le function factories sono particolarmente utili nei problemi di massima verosimiglianza.

8.1.2.2 Mutable states

Avere variabili a due livelli permette di mantenere delle variabili di stato che siano utili tra invocazioni differenti della medesima funzione: questo è possibile perchè, se l'execution environment è rinfrescato ad ogni chiamata, il parent environment resta fisso e possiamo modificare i suoi valori attraverso `«-`.
Un esempio per la creazione di contatori

```
make_counter <- function(){
  i <- 0
  function(){
    i <<- i + 1
    i
  }
}

a_counter <- make_counter()
another_counter <- make_counter()

a_counter()

## [1] 1

a_counter()

## [1] 2

a_counter()
```

```
## [1] 3

another_counter()

## [1] 1
```

E' fondamentale l'utilizzo di «- nella funzione restituita: alternativamente (usando <-) l'assegnamento creerebbe una variabile locale e complessivamente la funzione restituirebbe sempre 1 (non fungendo allo scopo per il quale è stata creata).

8.1.3 Liste di funzioni

Altro elemento base della programmazione funzionale è la possibilità di memorizzare funzioni come elementi di una lista

```
descr <- list(mean = mean,
              sd = sd,
              median = median,
              iqr = IQR)
x <- 1:100
```

Possiamo innanzitutto essere interessati ad applicare tutte le funzioni memorizzate in una lista. Un template utile per farlo è:

```
lapply(descr, function(f) f(x))

## $mean
## [1] 50.5
##
## $sd
## [1] 29.01149
##
## $median
## [1] 50.5
##
## $iqr
## [1] 49.5
```

nel quale si passa come argomento la funzione e, all'interno dell'anonymous function, si applica tale funzione ad `x` che viene ricercato nel parent environment. Il template può esser ulteriormente esteso; ad esempio se vogliamo procedere a temporizzare le funzioni di una lista un modo potrebbe essere il seguente

```
lapply(descr, function(f) system.time(replicate(1e03, f(x))))

## $mean
##    utente   sistema trascorso
##     0.005     0.000     0.005
##
```

```
## $sd
##    utente   sistema trascorso
##     0.018     0.000     0.018
##
## $median
##    utente   sistema trascorso
##     0.035     0.000     0.035
##
## $iqr
##    utente   sistema trascorso
##     0.086     0.000     0.086
```

8.2 Functionals

Le functionals sono il complemento delle closures: ovvero sono funzioni che prendono in input una funzione (e un set di dati) e restituiscono un valore derivante dall'applicazione della funzione sui dati.
Tutte le funzioni presentate nel seguito fungono in qualche modo da rimpiazzo compatto ed intelleggible a iterazioni facenti uso di `for`; ogni functional è però disegnata specificamente per un determinato compito e si classifica a seconda del dato di input che si attende e di output che restituisce. Se la fattispecie di functional che serve a noi non è implementata, una volta studiato il nucleo comune possiamo procedere alla sua scrittura.

8.2.1 `lapply`

`lapply` prende in input tre argomenti:

- una lista `X` o qualcosa che vi possa esser coercito;
- una funzione `FUN`;
- altri argomenti via `...`, che costituiscono i parametri da passare a `FUN`

`lapply` applica la funzione ad ogni elemento di `X` e **restituisce sempre una lista**, indipendentemente dalla class degli input. Il looping vero e proprio viene effettuato internamente in C, per efficienza (quindi non è visibile); tuttavia una funzione equivalente in puro R aiuta a focalizzare il pattern di questa tipologia di funzioni :

```
lapply

## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x556a765d6cc8>
## <environment: namespace:base>
```

```
lapply2 <- function(x, f, ...) {
  out <- vector("list", length(x))
  for (i in seq_along(x)){
    out[[i]] <- f(x[[i]], ...)
  }
  out
}
```

I passi svolti da una functional sono quindi:

1. allocare una struttura per l'output;
2. applicare la funzione ad ogni pezzo di input;
3. salvarne il risultato nella struttura d'output.

8.2.2 `vapply` (e `sapply`)

`sapply` e `vapply`, similmente a `lapply` prendono in input qualcosa coercibile a lista, ma cercano di semplificare l'output (a vettore) se possibile. Le differenze:

- `sapply` cerca di indovinare il tipo di output più opportuno; nello specifico è un wrapper di lapply che effettua detta trasformazione nelle fasi finali.[2] `vapply` si attende un parametro aggiuntivo che specifichi la tipologia di vettore (per ogni elemento esito dell'applicazione della funzione);
- `vapply` fornisce errori più informativi e non fallisce mai silenziosamente

Per questi motivi `vapply` è più sicura ed orientata alla programmazione, mentre `sapply` è più rivolta all'utilizzo interattivo.
Un esempio comparativo per indagare le colonne numeriche di un dataset

```
sapply(Indometh, is.numeric)

## Subject    time    conc
##   FALSE    TRUE    TRUE

vapply(Indometh, is.numeric, logical(1))

## Subject    time    conc
##   FALSE    TRUE    TRUE
```

Il beneficio è che con `vapply` abbiamo più controllo in programmazione sul tipo di output restituito e meno sorprese.
Una implementazione in puro R dell'essenza due funzioni segue:

[2] In particolare

- se il risultato è una lista con ogni elemento di lunghezza 1, ritorna un vettore (con nome)
- se il risultato è una lista con ogni elemento consistente in un vettore (della stessa lunghezza maggiore di 1), viene ritornata una matrice
- altrimenti viene ritornata una lista

```
sapply2 <- function(x, f, ...) {
  res <- lapply2(x, f, ...)
  simplify2array(res)
}

vapply2 <- function(x, f, f_value, ...) {
  out <- matrix(rep(f_value, length(x)), nrow = length(x))
  for (i in seq_along(x)) {
    res <- f(x[i], ...)
    stopifnot(
      length(res) == length(f_value),
      typeof(res) == typeof(f_value)
      )
    out[i, ] <- res
  }
  out
}
```

8.2.3 `Map`

In `lapply` solamente un argomento cambia di iterazione in iterazione, mentre gli altri rimangono fissi.
Con `Map`[3] la funzione è sempre unica tuttavia tutti i parametri forniti vengon passati in maniera vettorizzata:

```
a <- list(1:2, 3:4, 5:6,     7:8)
b <- list(1:2, 3:4, c(5, NA)    )
unlist(Map(sum, a, b))

## Warning in mapply(FUN = f, ..., SIMPLIFY = FALSE): l'argomento più
lungo non è un multiplo della lunghezza del più piccolo

## [1]  6 14 NA 18
```

Nel caso si voglia che alcuni parametri sian fissi/costanti si deve applicare una funzione anonima:

```
unlist(Map( function(x, y) sum(x, y, na.rm = TRUE), a, b) )

## Warning in mapply(FUN = f, ..., SIMPLIFY = FALSE): l'argomento più
lungo non è un multiplo della lunghezza del più piccolo

## [1]  6 14 16 18
```

8.2.4 `apply`

È una functional che opera su array, con sintassi:

[3] mapply è simile ma meno intuitiva

```
str(apply)

## function (X, MARGIN, FUN, ..., simplify = TRUE)
```

- un array `X`
- un intero `MARGIN` che indica quale margine dovrebbe esser "mantenuto" (il rimanente viene collassato mediante l'applicazione della funzione). Considerando una matrice `1` indica le righe, `2` le colonne ecc
- la funzione `FUN` da applicare
- `...` per gli argomenti da utilizzare con `FUN`

Un esempio con una matrice di valori numerici[4]:

```
x <- matrix(rnorm(50), nrow = 10, ncol = 5)
## Somme di riga
apply(x, 1, sum)

##  [1]  2.64938500 -0.09624178 -3.82686151  1.04416076 -0.03650136  2.25101608  2.08
##  [9] -0.85371949  2.53269338

## Medie di colonna
apply(x, 2, mean)

## [1]  0.16665367 -0.33785972  0.31414694  0.06993394  0.14798026

## Specificazione di parametri
apply(x, 2, quantile, probs = c(.25, .75))

##           [,1]       [,2]       [,3]       [,4]       [,5]
## 25% -0.4560911 -0.8363155 0.06107801 -0.3349207 -0.4263069
## 75%  0.8154583 -0.1202923 0.94058158  0.7087662  1.1096540
```

Una criticità di `apply` e che non ha un argomento `simplify`; pertanto non si è mai completamente sicuri di ciò che ritorna e l'utilizzo all'interno di funzioni deve essere accompagnato da rigoroso check dell'input.

8.2.5 `tapply`

`tapply` è una funzione che applica un'altra funzione a dei subsets di un vettore individuati da uno o più fattori di classificazione. Ai fini pratici può esser vista come la combinazione di `split` (applicato ad un vettore) e `sapply`.
La sintassi è:

[4]Per alcune applicazioni comuni (somme e medie) di array (tipicamente matrici) abbiamo shorcuts altamente ottimizzati come `rowSums`, `rowMeans`, `colSums`, `colMeans`, che sono tipicamente molto più veloci di apply (la differenza si nota soprattutto lavorando con matrici di grandi dimensioni).

```
str(tapply)

## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- `X` è il vettore a cui applicare la funzione
- `INDEX` è un factor o una lista di factor; costituisce il fattore di stratificazione per creare i gruppi
- `FUN` è la funzione da applicare
- `...` contiene gli argomenti opzionali da dare a `FUN`
- `simplify`

8.2.6 Collassare liste con `Reduce`

Reduce riduce un vettore/lista `x` ad un singolo valore applicando ricorsivamente una funzione `f` ai suoi membri, presi a due a due (combina i primi due elementi attraverso f, poi combina il risultato di questi con il terzo, e così via).
Ad esempio chiamare `Reduce(f, 1:3)` è equivalente a chiamare `f(f(1,2), 3)`.
L'essenza di Reduce può essere espressa mediante

```
Reduce2 <- function(f, x){
  out <- x[[1]]
  for(i in seq(2, length(x))){
    out <- f(out, x[[i]])
  }
}
```

Ad esempio può essere utile per fare dei merge ricorsivi, basati sempre sullo stesso id:

```
dat <- data.frame(id = rep(1:2, 3),
                  x = letters[1:6])
(dat.split <- split(dat, gl(3,2) ))

## $`1`
##   id x
## 1  1 a
## 2  2 b
##
## $`2`
##   id x
## 3  1 c
## 4  2 d
##
## $`3`
##   id x
## 5  1 e
## 6  2 f
```

```
## Poniamo il dataset master nel primo elemento della lista e le tabelle di
## lookup nei successivi
Reduce(function(x, y) merge(x = x, y = y, by = "id", all.x = TRUE),
       dat.split)

##   id x.x x.y x
## 1  1   a   c e
## 2  2   b   d f
```

8.2.7 Funzioni per la gestione di predicati

Un predicato è una funzione che restituisce un singolo `TRUE` o `FALSE`. Vi sono tre functionals dedicate ai predicati, nel senso che applicano ricorsivamente i predicati ad elementi della lista ottenendo i valori di verità/falsità, sui quali poi effettuano semplici manipolazioni aggiuntive:

- `Filter` seleziona solamente gli elementi che matchano il predicato (ritornano `TRUE`)
- `Find` ritorna il primo elemento che matcha il predicato (o l'ultimo se `right = TRUE`)
- `Position` ritorna la posizione del primo elemento che matcha il predicato (o l'ultimo se `right = TRUE`)

Un esempio triviale

```
a <- list(2, 4, 6, 8, 10)
f <- function(x) x >= 5
unlist(Filter(f, a))

## [1]  6  8 10

unlist(Find(f, a))

## [1] 6

unlist(Position(f, a))

## [1] 3
```

8.2.8 Functional e matematica

Le functional sono comuni in matematica, laddove si necessita di strumenti generici per processare funzioni matematiche. Esempi ne sono:

- `integrate`, che calcola l'area sotto la curva definita dalla funzione `f`
- `uniroot` trova dove la funzione `f` raggiunge lo 0
- `optimise` trova per quale valore di `x` si ha il minimo di `f(x)` (per il massimo di f(x) basta calcolare il minimo di -f(x))

- `optim` è una generalizzazione di `optimise` che lavoro a più di una dimensione

Vedi advanced R, pag 222, per esempi e approfondimenti.

8.3 Function operators

Abbiamo visto funzioni che restituiscono una funzione (function factory) e funzioni che prendono in input una funzione (functionals). In questa sezione approfondiamo i function operators (FO), che consistono in funzioni che

- prendono una o più funzioni in input e
- restituiscono una funzione come output

Sono quindi function factory che servono tipicamente per wrappare una funzione esistente; qui si mostrano quattro tipi di FO:

- **behavioural** FO: consistono in FO che cambiano il comportamento di una funzione
- **input** FO: FO che cambiano l'input fornito ad una funzione
- **output** FO: FO che cambiano l'output fornito ad una funzione
- **combining** FO: FO che combinano più funzioni attraverso composizione ed operatori logici

Il template base comune di queste funzioni è il seguente:

```
FOtemplate <- function(f){
  force(f)
  function(...){
    ## eventuale preprocessing input qua ...
    rval <- f(...)
    ## eventualmente post-processing output qua ...
    ## ... e restitusci il valore finale
    rval
  }
}
```

Il `force` serve per valutare la funzione ed evitare problemi connessi alla lazy evaluation; può infatti essere che, soprattutto in loop, `f` cambi dal momento in cui `FOtemplate` è chiamata per creare la funzione utilizzata e quello in cui vi è effettivamente la chiamata di quest'ultima. Se non ponessimo `force` la funzione sarebbe valutata solo al momento della chiamata della funzione generata, andando incontro ad effetti probabilmente indesiderati.
Una volta definito il template opportuno (che si differenziano, appunto, a seconda del pre/post processing che vogliamo imporre alle funzioni) possiamo utilizzare il template:

- come function factory classica, quindi creando assegnando la funzione creata

- in maniera diretta come segue

```
FOtemplate(mean)(1:10)

## [1] 5.5
```

Nell'esempio il template non aggiunge niente ma si limita ad eseguire la funzione passatagli

8.3.1 Behavioural FOs

Queste FO lasciano input ed output della funzione non cambiata ma aggiungono qualche attività prima o dopo. L'esempio portato è creare una funzione che sia utile per il download di file multipli da un server, modificando opportunamente `download.file` come segue

```
download_file <- function(url, ...){
  download.file(url, basename(url), ...)
}
```

e aggiungendovi esecuzione posticipata e un semplice indicatore di avanzamento.

Esecuzione posticipata Per l'esecuzione posticipata possiamo utilizzare `Sys.sleep` e impostare un FO come segue

```
delay_by <- function(delay, f){
  force(f)
  function(...){
    Sys.sleep(delay)
    f(...)
  }
}
```

Indicatore di avanzamento Per un semplice indicatore di avanzamento possiamo definire una template che stampi un punto ogni tot chiamate della funzione. Occorre implementare un contatore, e lo facciamo mediante:

```
dot_every <- function(n, f){
  i <- 1
  function(...){
    if (i %% n == 0) cat(".")
    i <<- i + 1
    f(...)
  }
}
```

Chiamata complessiva Definite le due funzioni di cui sopra possiamo utilizzare il tutto come

```
lapply(urls, dot_every(10, delay_by(1, download_file)))
```

Si noti che nella definizione di `dot_every` e `delay_by` si è posto il parametro della funzione come ultimo; alternativamente (ponendolo come primo) avremmo avuto come chiamata complessiva

```
lapply(urls, dot_every(delay_by(download_file, 1), 10))
```

che è forse meno leggibile.

8.3.2 Output FOs

Queste FO modificano l'output prodotto da una funzione. Le modifiche possono essere di tipo cosmetico o sostanziale: ne vediamo alcuni esempi più semplici per dare l'idea della tipologia di funzione da definire.

Negate Pensata per essere applicata ad una funzione predicato, `Negate` ritorna una funzione che afferma restituisce il contrario dell'originale

```
Negate <- function(f){
  force(f)
  function(...) !f(...)
}
Negate(is.null)(NULL)

## [1] FALSE
```

Può essere utile quando quello che ci serve è il contrario di quello restituito da una funzione; ad esempio se vogliamo filtrare gli elementi di una lista che non sono nulli si può definire

```
compact <- function(x) Filter(Negate(is.null), x)
```

failwith trasforma una funzione che manda un error in una funzione che ritorna un valore default in caso di errore, o il valore normale in caso di esecuzione corretta. L'essenza è semplicemente un wrapper di `try`:

```
failwith <- function(default = NULL, f, quiet = FALSE) {
  force(f)
  function(...) {
    out <- default
    try(out <- f(...), silent = quiet)
    out
  }
}
log("a")

## Error in log("a"):  argomento non-numerico per una funzione matematica
```

```
failwith(NA, log, quiet = TRUE)("a")

## [1] NA
```

`failwith` è molto utile se utilizzata assieme a functionals: invece di terminare tutto il loop a causa dell'errore si può completare l'iterazione e in seguito verificare quali sono stati i problemi. Ad esempio nel fittare una serie di GLM ad un data frame, qualora un modello fallisca nella convergenza si può sempre proseguire al prossimo modello e più tardi verificare quali sono stati i modelli problematici

```
# If any model fails, all models fail to fit:
models <- lapply(datasets, glm, formula = y ~ x1 + x2 * x3)
# If a model fails, it will get a NULL value
models <- lapply(datasets, failwith(NULL, glm),
                 formula = y ~ x1 + x2 * x3)
# remove failed models (NULLs) with compact
ok_models <- compact(models)
# extract the datasets corresponding to failed models
failed_data <- datasets[vapply(models, is.null, logical(1))]
```

8.3.3 Input FOs

Anche per queste FOs le modifiche possono esser marginali o più sostanziali. Alcune esempi di builtin o di come realizzare questa tipologia di funzioni seguono.

Cambio di tipologia di input Alcune funzioni già esistenti servono allo scopo: `base::Vectorize` converte una funzione scalare in una vettoriale, ciclando sull'argomento `vectorize.args`. Un esempio con `sample` permette di generare campioni multipli in una chiamata; si adotta `SIMPLIFY = FALSE` per assicurarsi che la nuova funzione vettorizzata restituisca sempre una lista:

```
sample2 <- Vectorize(sample, "size", SIMPLIFY = FALSE)
str(sample2(1:5, 5:3))

## List of 3
##  $ : int [1:5] 1 3 2 5 4
##  $ : int [1:4] 3 5 1 2
##  $ : int [1:3] 3 4 5
```

Un altro esempio consiste nel convertire una funzione che prende in input una serie di parametri in una che prende una lista

```
splat <- function(f){
  force(f)
  function(args) {
    do.call(f, args)
  }
}
```

che può tornare utile se si vuole invocare una funzione con argomenti che variano

```
x <- c(NA, runif(100), 1000)
args <- list(list(x),
             list(x, na.rm = TRUE),
             list(x, na.rm = TRUE, trim = 0.1))
lapply(args, splat(mean))

## [[1]]
## [1] NA
##
## [[2]]
## [1] 10.42719
##
## [[3]]
## [1] 0.536593
```

8.3.4 Combining FOs

In questa sezione vediamo alcune FOs che servono per combinare funzione, nello specifico composizione di funzioni e connessione di predicati mediante equivalente di operatori logici

Composizione di funzioni In base alla definizione matematica $f \circ g = f(g(x))$; per darne una implementazione in R possiamo usare operatore binario definito come

```
`%O%` <- function(f, g) {
  force(f)
  force(g)
  function(...) f(g(...))
}
```

In seguito possiamo adottare

```
vapply(mtcars, length %O% unique, integer(1))

##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   25    3   27   22   22   29   30    2    2    3    6
```

mentre l'alternativa con funzione anonima sarebbe

```
vapply(mtcars, function(x) length(unique(x)), integer(1))

##  mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
##   25    3   27   22   22   29   30    2    2    3    6
```

Con questa tipologia di operatore è l'ultima funzione ad essere applicata per prima (e poi si procede a ritroso); per un approccio speculare bisogna utilizzare operatori di pipe

Combining FOs per predicati logici Quando si usa `Filter` o altre functionals che lavorano con predicati logici può essere utile definire degli FOs che permettano di combinare i risultati di diversi predicati mediante i classici operatori logici. Possiamo dunque definire

```
and <- function(f1, f2) {
  force(f1); force(f2)
  function(...) {
    f1(...) && f2(...)
  }
}

or <- function(f1, f2) {
  force(f1); force(f2)
  function(...) {
    f1(...) || f2(...)
    }
}

not <- function(f) {
  force(f)
  function(...) {
    !f(...)
  }
}
```

e utilizzarle ad esempio mediante

```
sapply(iris, class)

## Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
##    "numeric"    "numeric"    "numeric"    "numeric"     "factor"

names(Filter(or(is.character, is.factor), iris))

## [1] "Species"

names(Filter(not(is.numeric), iris))

## [1] "Species"
```

Capitolo 9

Computing on the language

9.1 Espressioni

Alla base del linguaggio vi sono **espressioni** che vengono valutate: una espressione è un *oggetto* che rappresenta una azione che può essere eseguita in R[1].
Generalmente diteggiate, le espressioni (in quanto oggetti) possono anche essere generate/modificate in maniera programmatica.
Alcuni **strumenti utili** sono:

- `quote` serve per catturare una espressione diteggiata e restituirla non valutata

```
x <- 2
a <- quote(x + 1)
str(a)

##  language x + 1
```

- `eval` serve per valutare una espressione passatagli come parametro

```
eval(a)

## [1] 3
```

  Permette di specificare l'environment (anche lista o data.frame) dove questo deve avvenire

```
z <- data.frame(x = 1:4, y = sin(1:4))
eval(a, envir = z)

## [1] 2 3 4 5
```

[1] Sfortunatamente `expression` non ritorna una espressione in questo senso, ma una sorta di lista di espressioni.

Un errore comune con `eval` è di dimenticarsi di utilizzare una espressione quotata;

```
eval(x, z)

## [1] 2

eval(quote(x), z)

## [1] 1 2 3 4

eval(y, z)

##   a b
## 1 1 a

eval(quote(y), z)

## [1]  0.8414710  0.9092974  0.1411200 -0.7568025
```

- `parse` torna utile nel caso più raro il codice sia memorizzato in stringhe: attraverso essa è possibile convertire una stringa in una espressione
- al contrario fa `deparse` che prende una espressione e la trasforma in stringa

Una espressione di fatto può essere pensata come una lista; ogni sua componente è modificabile mediante assegnamento (questo comportamento deriva dalle proprietà delle call, componente fondamentale delle espressioni, che sono direttamente assimilabili a liste, e dal fatto che in R qualsiasi cosa che viene eseguita è una call); questo modo di programmare si chiama **metaprogrammazione**, o *computing on the language*.

```
a

## x + 1

length(a)

## [1] 3

as.list(a)

## [[1]]
## `+`
##
## [[2]]
## x
##
## [[3]]
## [1] 1
```

```
a[[1]] <- `-`
a[[2]] <- as.name("y")
a[[3]] <- 3
a

## .Primitive("-")(y, 3)

eval(a, z)

## [1] -2.158529 -2.090703 -2.858880 -3.756802
```

Una espressione è anche chiamata *abstract syntax tree* perchè può essere rappresentata mediante una struttura ad albero di operazioni successive.
La funione `pryr::ast` permette di stampare l'**albero dell'espressione** (qui non posta tra `quote`):

```
library(pryr)

## Error in library(pryr):  non c'è alcun pacchetto chiamato 'pryr'

ast(y <- x * 10)

## Error in ast(y <- x * 10):  non trovo la funzione "ast"
```

Ad ogni livello dell'albero, il primo elemento è la funzione da applicare mentre i rimanenti sono gli argomenti; l'albero è ricorsivo.

Vi sono 4 componenti possibili di una espressione, che si vanno ad approfondire in seguito:

1. constants
2. names
3. calls
4. pairlists

9.1.1 Constants

Le costanti sono vettori atomici di lunghezza unitaria. `ast` li stampa tout court;

```
ast("a")

## Error in ast("a"):  non trovo la funzione "ast"

ast(1L)

## Error in ast(1L): non trovo la funzione "ast"
```

9.1.2 Names

I nomi, anche chiamati simboli, rappresentano il nome di un oggetto. `ast` li rappresenta premettendo un backticks

```
ast(asd)

## Error in ast(asd):  non trovo la funzione "ast"

ast(mean)

## Error in ast(mean):  non trovo la funzione "ast"

ast(`an unusual name`)

## Error in ast(‘an unusual name‘):  non trovo la funzione "ast"
```

Tipicamente si usa `quote` per catturare dei nomi o si usa `as.name` per convertire da un carattere a un nome. `is.name` serve per testare se un elemento è un nome

```
quote(a)

## a

as.name("a")

## a

is.name(quote(a))

## [1] TRUE
```

I nomi sono chiamati anche symbols; `as.symbol` e `is.symbol` sono identici a `as.name` e `is.name`.

9.1.3 Calls

Le calls rappresentano una chiamata di funzione. Nella visualizzazione, `ast` stampa `()` e sotto stampa gli elementi figli: il primo figlio è il nome della funzione chiamata, i rimanenti sono gli argomenti

```
ast(f())

## Error in ast(f()):  non trovo la funzione "ast"

ast(f(g(), h(a = 1,b = 2)))

## Error in ast(f(g(), h(a = 1, b = 2))):  non trovo la funzione "ast"
```

Sono simili alle liste in quanto

- hanno una lunghezza, ispezionabile mediante `length`, pari al numero di argomenti più uno (nome della funzione da applicare

- si può accedere in lettura e scrittura ai rispettivi argomenti mediante `[[` e `[`
- sono ricorsive: possono contenere a loro volta constants, names, calls e pairlist.

Come abbiamo detto i membri di una call sono accessibili in lettura e scrittura:

```
a <- quote(mean(1:10, trim = 0.05))
length(a)

## [1] 3

as.list(a)

## [[1]]
## mean
##
## [[2]]
## 1:10
##
## $trim
## [1] 0.05

a$trim <- NULL
eval(a)

## [1] 5.5
```

Generalmente è preferibile la modifica per nome che per indice di lista delle chiamate, dato che l'ordine di chiamata dei parametri può variare.

Per **creare una call** da zero possiamo utilizzare `call` (fornendo come primo argomento una stringa con il nome della funzione da chiamare e in seguito i parametri separati da virgole o alternativamente `as.call` (fornendo una lista)

```
a <- call(":", 1, 10)
eval(a)

##  [1]  1  2  3  4  5  6  7  8  9 10

b <- call("mean", quote(1:10), na.rm = TRUE)
eval(b)

## [1] 5.5
```

Per **accedere alla call corrente**, ovvero all'espressione che ha fatto partire l'esecuzione della funzione nella quale ci troviamo, abbiamo due strumenti:

- `sys.call` ritorna esattamente quello che ha ditteggiato l'utente
- `match.call` ritorna una chiamata equivalente che fa utilizzo solamente di parametri con nome

```
f <- function(abc = 1, def = 2, ghi = 3) {
  list(sys = sys.call(), match = match.call())
}
f(d = 2, d)

## $sys
## f(d = 2, d)
##
## $match
## f(abc = d, def = 2)
```

In sede di programmazione se si vuole modificare la chiamata effettuata dall'utente e valutare questa versione, è preferibile e più semplice utilizzare `match.call` e procedere, in seguito alla modifica dell'oggetto ottenuto, alla sua valutazione mediante `eval`.

9.1.4 Pairlists

sono eredità del passato di R e sono utilizzate solo negli argomenti formali delle funzioni (sono state rimpiazzate dalle liste in ogni dove). `ast` stampa `[]` al livello più alto della pairlist

```
ast(function(x = 1, y = 2) x + y)

## Error in ast(function(x = 1, y = 2) x + y):  non trovo la funzione
"ast"
```

Per approfondimenti vedi Advanced R pag 296.

9.2 Non standard evaluation

In R, a contrario di altri linguaggi, si può accedere non solo al valore assunto da un parametro ma anche dalle espressioni che nell'environment chiamante sono state fornite come parametro.
Nello specifico gli argomenti delle funzioni sono rappresentati da un tipo speciale di oggetti, chiamato **promise**. Queste consistono in una espressione (necessaria per calcolare il valore) ed un environment nel quale valutarla (di solito non si ha a che fare con le promise perchè la prima volta che il parametro viene utilizzato, l'espressione viene valutata nell'environment di riferimento).

Questo rende possibile valutare il codice in maniera non standard, ovvero di fare uso della cosiddetta non standard evaluation, o NSE; la NSE è particolarmente utile per funzioni quando si fa uso prevalentemente interattivo perchè permette di ridurre drasticamente la ditteggiatura per le chiamate.

9.2.1 Catturare le espressioni dei parametri

La funzione `substitute` sta alla base della NSE: se applicata ad un parametro di una funzione restituisce l'espressione impostata nella chiamante

```
f <- function(x) substitute(x)
f(asd)

## asd
```

`substitute` di solito viene accompagnato con `deparse` se si desidera trasformare in stringa l'espressione; questo può essere utile ad esempio:

- per creare etichette nei grafici che fanno uso di espressioni digitate in sede di chiamata
- per evitare di dover digitare le virgolette (come ad esempio in library)

9.2.2 NSE in `subset`

`subset` costituisce un esempio di funzione che performa NSE; votata prevalentemente all'utilizzo interattivo, permette il notevole risparmio di digitazione e rende il codice complessivamente più leggibile (soprattutto quando i criteri di scelta aumentano):

```
sample_df <- data.frame(a = 1:5,
                        b = 5:1,
                        c = c(5, 3, 1, 4, 1))
## Versioni classica
sample_df[sample_df$a >= 4, ]

##   a b c
## 4 4 2 4
## 5 5 1 1

## Versione con subset
subset(sample_df, a >= 4 )

##   a b c
## 4 4 2 4
## 5 5 1 1
```

`subset` fa si che che l'espressione sia valutata non nell'ambito non dell'environment chiamante (potrebbero non esistere anche i nomi a e b nel global environment) ma viene applicato ad un altro environment.
Volendo cercare di riprodurne il comportamento, dobbiamo far si che `a` e `b` vengano valutate come `sample_df$a` e `sample_df$b` invece che `globalenv()$x`; per farlo facciamo semplicemente uso di `eval`

```
subset2 <- function(x, condition){
  condition_call <- substitute(condition)
  r <- eval(condition_call, envir = x)
  x[r, ]
}
subset2(sample_df, a >= 4 )
```

```
##   a b c
## 4 4 2 4
## 5 5 1 1
```

Questa prima versione funziona ma vi possono essere problemi connessi allo scoping delle variabili:

```
y <- 4
x <- 4
condition <- 4
condition_call <- 4

## vorremmo che le chiamate seguenti restituissero tutte la stessa cosa
subset2(sample_df, a == 4)

##   a b c
## 4 4 2 4

subset2(sample_df, a == y)

##   a b c
## 4 4 2 4

subset2(sample_df, a == x)

##       a  b  c
## 1     1  5  5
## 2     2  4  3
## 3     3  3  1
## 4     4  2  4
## 5     5  1  1
## NA   NA NA NA
## NA.1 NA NA NA

subset2(sample_df, a == condition)

## [1] a b c
## <0 righe> (o 0-length row.names)

subset2(sample_df, a == condition_call)

## [1] a b c
## <0 righe> (o 0-length row.names)
```

Si nota che `x`, `y`, `condition` e `condition_call` sono tutte variabili locali nella funzione. Il problema è che `eval`, qualora non trovi i nomi nell'environment specificato, guarda nell'environment di `subset2`[2], che è ovviamente non quello che vogliamo: abbiamo bisogno di un modo per dire a `eval` dove trovare i nomi

[2]Questo avviene perchè fa uso del parametro di default di eval, che valuta parent.frame, ma la valuta (essendo un valore di default) nell'environment della funzione chiamata, non della chiamante

necessari, qualora non risiedano nell'environment specificato. Il parametro che ci serve è il terzo, `enclos` che dobbiamo settare a `parent.frame()`. (a parte questo alcune migliorie generali per la funzione)

```
subset3 <- function(x, condition){
  condition_call <- substitute(condition)
  r <- eval(condition_call, envir = x, enclos = parent.frame())
  x[r, ]
}

subset3(sample_df, a == 4)

##   a b c
## 4 4 2 4

subset3(sample_df, a == y)

##   a b c
## 4 4 2 4

subset3(sample_df, a == x)

##   a b c
## 4 4 2 4

subset3(sample_df, a == condition)

##   a b c
## 4 4 2 4

subset3(sample_df, a == condition_call)

##   a b c
## 4 4 2 4
```

Quello che abbiamo fatto è un esempio di **dynamic scoping**, ovvero di ricerca dei nomi mancanti nell'environment chiamante, non in quello ove è stata definita la funzione.

9.2.3 Chiamare funzioni NSE da altre funzioni

Vi possono essere problemi nella NSE se la nostra funzione è chiamata da altre funzioni. Ipotizziamo di voler definire una funzione `subscramble` che permuti un sottoinsieme dei dati originali.

```
subscramble <- function(x, condition) {
  scramble(subset3(x, condition))
}

scramble <- function(x) x[sample(nrow(x)), ]
```

```
subset3 <- function(x, condition) {
  condition_call <- substitute(condition)
  r <- eval(condition_call, x, parent.frame())
  x[r, ]
}

subscramble(sample_df, a >= 4)

## Error in eval(expr, envir, enclos):  oggetto 'a' non trovato
```

Il fatto è che quando `subset3` è chiamata da dentro `subscramble`, il valore di `condition_call` è il simbolo `condition` piuttosto che la chiamata `a >= 4`. La chiamata ad `eval` in `subset3` cerca il nome `condition` inizialmente in `envir = x` (il `data.frame sample_df`). Non trovandolo li continua la ricerca in `enclos = parent.frame()` dove effettivamente trova un oggetto di nome `condition`, che viene dunque valutato. Tuttavia questo oggetto è una promise, la cui espressione è `a >= 4` e il cui environment di valutazione è `.GlobalEnv`. Nel valutare si fa si che, a meno che un oggetto di nome `a` sia trovato in `.GlobalEnv` (e da li nel search path), la valutazione della promise fallisca con il messaggio osservato.
E se un oggetto di nome `a` esiste succedono casini

```
a <- c(4, 1, 2)
subscramble(sample_df, a >= 4)

##   a b c
## 4 4 2 4
## 1 1 5 5
```

Questo è un esempio della tensione generale tra funzioni che sono disegnate per l'utilizzo interattivo e funzioni sicure in ambito di programmazione: una funzione che usi `substitute` può ridurre la digitazione ma **può essere difficile/pericolosa da chiamare da un'altra funzione**.

Come sviluppatori, bisognerebbe fornire come workaround due versioni della stessa funzione, una che usa la standard evaluation e una che usa NSE. In questo caso possiamo scrivere una versione di `subset3` che considera una espressione già quoted (per questo il `_q` come suffisso del nome)

```
subset4_q <- function(x, condition){
  r <- eval(condition, x, parent.frame())
  x[r, ]
}
```

Tipicamente di questa funzione l'utente finale non se ne servirà, quindi il nome potrebbe essere anche piu lungo ed esplicativo.
Dopodichè possiamo riscrivere `subset4` e `subscramble` affinchè facciano uso di `subset4_q`; sono queste due le funzioni dedicate all'utente

```
subset4 <- function(x, condition){
  subset4_q(x, substitute(condition))
}
```

```
subscramble2 <- function(x, condition){
  condition <- substitute(condition)
  scramble(subset4_q(x, condition))
}

subset4(sample_df, a >= 3)

##   a b c
## 3 3 3 1
## 4 4 2 4
## 5 5 1 1

subscramble2(sample_df, a >= 3)

##   a b c
## 5 5 1 1
## 3 3 3 1
## 4 4 2 4
```

9.2.4 Catturare . . . non valutato

Per catturare le espressioni non valutate residenti in . . . vi è una tecnica che funziona bene in diverse situazioni

```
unev_dots <- function(...){
  eval(substitute(alist(...)))
}

x <- 1
y <- 2
(res <- unev_dots(a = x, b = y))

## $a
## x
##
## $b
## y

sapply(res, class)

##      a      b
## "name" "name"
```

Si fa uso della funzione `alist` che semplicemente cattura tutti i suoi argomenti.

Capitolo 10

Programmazione ad oggetti

Classi e metodi sono un sistema per effettuare programmazione ad oggetti in R. Classi e metodi permettono la creazione di nuovi oggetti e le funzioni che servono per manipolarli.
La programmazione ad oggetti in R è un po' differente da quella degli altri linguaggi di OOP (object oriented programming): R è un functional language e invece che avere *oggetti* che hanno metodi, le *funzioni* hanno metodi che si comportano in maniera differente a seconda della classe di dato che trattano.
Vi sono due stili di classi e metodi in R:

- **classi/metodi S3**; inclusi a partire dalla versione 3 di S. Sono più informali (una nuova classe di dati non ha una definizione formale), a volte chiamati *old style* classes/methods.
- **classi metodi S4**; più formali e rigorosi, introdotti a partire da R 1.4.0, chiamati new-style classes/methods

Per ora e per il futuro più prossimo metodi/classi S3 ed S4 sono sistemi separati (possono esser mischiati fino a un certo tot); ogni sistema può esser utilizzato in maniera indipendente, ma gli sviluppatori sono incoraggiati ad utilizzare S4 (S3 è ancora utilizzato poichè quick and dirty).
Il codice per implementare metodi e classi S3 è builtin, quello di S4 si trova nel pacchetto `methods` (di solito caricato di default).

10.1 Concetti generali

classe è la descrizione di una cosa.

oggetto è una istanza di una classe

generic function è una funzione che dispaccia i metodi; tipicamente incapsula un conetto generico (es `plot`,`mean`, `predict`, possono dover avere comportamente differenti a seconda dell'oggetto cui vengono applicate). Le funzioni generiche S3 ed S4 sono differenti a livello di codice, ma concettualmente svolgono lo stesso ruolo.

metodo è l'implementazione di una funzione generica per un oggetto di una particolare classe. Esso è una funzione che opera solamente su una certa classe di oggetti

Quando si programma si possono scrivere nuovi metodi per una funzione generica già esistente o si possono definire nuove funzioni generics e i metodi loro associati. Ovviamente se un tipo di dato necessario non esiste in R si può sempre definire una nuova classe insieme a generics e metodi che vanno con essa.

10.2 S3

10.2.1 Classi

Ottenere la classe di un oggetto Si ottiene mediante la funzione `class`

```
class(Indometh)

## [1] "nfnGroupedData" "nfGroupedData"  "groupedData"    "data.frame"
```

La **classe di un oggetto** è un *vettore carattere* che può essere *anche di lunghezza superiore ad 1*, permettendo forme di ereditarietà.

Definizione di un oggetto appartenente ad una classe La differenza principale tra S3 e S4 è che in S3 non vi è un modo per definire una classe formalmente (ovvero impostarne a priori i contenuti possibili).
Semplicemente per dichiarare un determinato oggetto come facente parte di una classe dobbiamo impostare l'attributo `class`. Ad esempio un modo tipico è creare una lista

```
a <- list(surname=c("Braglia", "Braglia"), name=c("Luca", "Tequila"))
class(a) <- "famiglia"
a

## $surname
## [1] "Braglia" "Braglia"
##
## $name
## [1] "Luca"    "Tequila"
##
## attr(,"class")
## [1] "famiglia"
```

Alternativamente se:

- vogliamo sfruttare alcuni metodi gia esistenti, ad esempio per la classe `data.frame`
- vogliamo altresì fornire all'utente una funzione che definisca una classe nel senso di S3, effettuando preliminarmente un check dei dati forniti

potremo scrivere qualcosa del genere

```
famiglia <- function(surname, name) {
```

```
    stopifnot(is.character(surname))
    stopifnot(is.character(name))
    if (length(surname) != length(name)) {
        stop("surname e name devono avere la stessa lunghezza")
    }

    my.object <- data.frame("surname" = surname, "name" = name)
    class(my.object) <- c("famiglia", "data.frame")
    my.object
}
```

In seguito per la creazione di un oggetto di classe famiglia potremo fare

```
mia.fam <- famiglia(surname=c("Braglia", "Braglia"), name=c("Luca", "Tequila"))
mia.fam

##   surname    name
## 1 Braglia    Luca
## 2 Braglia Tequila
```

La funzione `unclass` rimuove l'attributo classe all'oggeto

10.2.2 Funzioni generiche

Una volta che si è definita una struttura di dati con una determinata classe è necessario tipicamente aggiungere metodi per le funzioni generiche già esistenti.

```
print

## function (x, ...)
## UseMethod("print")
## <bytecode: 0x556a791a44c0>
## <environment: namespace:base>

summary

## function (object, ...)
## UseMethod("summary")
## <bytecode: 0x556a817a91b8>
## <environment: namespace:base>

plot

## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x556a81f4cd98>
## <environment: namespace:base>
```

`print` è chiamata di default quando il nome di un oggetto è chiamato al prompt, `summary` serve per fornire alcune misure di sintesi complessiva dell'oggetto e `plot` per rappresentare graficamente l'oggetto.

Creazione di una funzione generica Di solito una funzione generica è una funzione semplice che utilizza `UseMethod` o `NextMethod` per indirizzare al metodo corretto.
Nel creare una nuova funzione generica va tenuto in mente che la lista di argomenti inclusa costituirà il set massimo di argomenti per i metodi che da essa dipartono, inclusi quelli scritti molti anni piu tardi. Pertanto scegliere un buon set di argomenti potrebbe essere un problema di design rilevante e ci vogliono buone ragioni per non includere un `ldots`
Se viene posto `...` bisogna pensare a dove porlo. Di solito è posto come ultimo argomento della generica ma non necessariamente è la scelta giusta. Si rammenti che se ... è fornito, gli argomenti che lo seguono devono essere nominati esplicitamente in sede di chiamata, al fine di non essere inghiottiti dallo stesso (quelli prima vanno anche di partial matching senza problemi)

Infine a volte si vuole rendere generica una funzione di base di R che non lo è. Per evitare di chiedere modifiche al codice di R, un modo per definire una funzione `foo` di R generica è il seguente:

```
foo <- function(object, ...) UseMethod("foo")
foo.default <- function(object, ...) base::foo(object)
```

Variabili di ambiente disponibili all'interno di un metodo In aggiunta ai parametri all'interno dell'ambiente di un metodo saranno disponibili alcuni nomi aggiuntivi come `.Class`, `.Method`, `.Generic`

Dispatching con `UseMethod` Tipicamente le funzioni generiche utilizzano

```
UseMethod(generic, object)
```

Il **method dispatch** avviene sulla base della classe/i del:

- di default il primo argomento della funzione generica in cui `UseMethod` o `NextMethod` vengono chiamata
- alternativamente l'`object` fornito come argomento alle funzioni stesse. Ad esempio

  ```
  UseMethod("loglm1", data)
  ```

  chiamata da `loglm` farà partire il metodo sulla base della classe di data, non del primo argomento della generica

Considerando che la classe di un oggetto può essere un vettore di lunghezza superiore all'unità (es `c("first","second")`, quando una funzione generica `fun` chiama `UseMethod("fun")`:

- il sistema cerca inizialmente la funzione `fun.first` e se la trova, la applica all'oggetto

- se non la trova prova a cercare `fun.second` allo stesso modo e in seguito ad applicarla
- se anche questa manca prova con `fun.default` se esiste
- se anche `fun.default` non esiste viene restituito un errore

Il funzionamento di UseMethod

- crea una nuova funzione con argomenti analoghi alla generica, e lascia i parametri non valutati,
- ma qualsiasi variabile locale definita prima di `UseMethod` sarà però disponibile anche nel metodo (a differenza di S)
- qualsiasi statement dopo UseMethod nella generica non sarà valutata e `UseMethod` non ritorna nulla

Per maggiori informazioni si veda `?UseMethod`.

Dispatching con `NextMethod` Sebbene sia a volte deprecato (Venables and Ripley) e a volte visto come standard (Writing Extension - Cap 7 Generic functions and methods), la funzione `NextMethod` può essere chiamata all'interno di un metodo, per farne partire un altro. Ad esempio nel metodo per `data.frame` di `t`,

```
t.data.frame

## function (x)
## {
##     x <- as.matrix(x)
##     NextMethod("t")
## }
## <bytecode: 0x556a80b4e160>
## <environment: namespace:base>
```

è abbastanza usuale per un metodo di effettuare alcune modifiche all'oggetto considerare e di dispacciare al prossimo metodo.

La funzione `NextMethod`

- guarda alla lista di classi dell'oggetto a partire da quella dopo quella che ha fatto partire il corrente metodo e
- fa si il metodo per la seconda (ad esempio) venga invocata
- se questo non è presente si procede con la terza, e così via
- se non trova nessun metodo specifico prova quello di default
- se non si trova la default non si da necessariamente errore
- in aggiunta nell'ambiente creato da una `NextMethod` sarà disponibile un attributo `previous` di `.Class`

Listare le funzioni generiche disponibili Per individuare tutte le funzioni generiche nel search path, si può utilizzare questo trick

```
Filter(length,
       sapply(search(), function(x) {
                  Filter(isGeneric,ls(all.names=TRUE,env = as.environment(x)))
          } ))
```

```
## $`package:stats`
## [1] "cov2cor"  "toeplitz" "update"
##
## $`package:graphics`
## [1] "image"
##
## $`package:utils`
## [1] ".DollarNames" "head"         "prompt"       "tail"
##
## $`package:methods`
##  [1] "addNextMethod" "Arith"         "body<-"        "cbind2"        "coerce"
##  [7] "Compare"       "Complex"       "initialize"    "kronecker"     "loadMethod"
## [13] "Math"          "Math2"         "Ops"           "rbind2"        "show"
## [19] "Summary"
##
## $`package:base`
##   [1] "-"           "!"           "!="          "["           "[[<-"        "[<-"
##   [7] "*"           "/"           "&"           "%*%"         "%/%"         "%%"
##  [13] "^"           "+"           "<"           "<="          "=="          ">"
##  [19] ">="          "|"           "$"           "$<-"         "abs"         "acos"
##  [25] "acosh"       "all"         "all.equal"   "any"         "anyNA"       "as.ar
##  [31] "as.double"   "as.integer"  "as.logical"  "as.matrix"   "as.numeric"  "as.ve
##  [37] "asin"        "asinh"       "atan"        "atanh"       "body<-"      "ceili
##  [43] "chol"        "chol2inv"    "colMeans"    "colSums"     "cos"         "cosh"
##  [49] "cospi"       "crossprod"   "cummax"      "cummin"      "cumprod"     "cumsu
##  [55] "determinant" "diag"        "diag<-"      "diff"        "digamma"     "dim"
##  [61] "dim<-"       "dimnames"    "dimnames<-"  "drop"        "exp"         "expm1
##  [67] "floor"       "formals<-"   "format"      "gamma"       "is.finite"   "is.in
##  [73] "is.na"       "isSymmetric" "kronecker"   "length"      "lgamma"      "log"
##  [79] "log10"       "log1p"       "log2"        "max"         "mean"        "min"
##  [85] "norm"        "print"       "prod"        "qr"          "qr.coef"     "qr.fi
##  [91] "qr.Q"        "qr.qty"      "qr.qy"       "qr.R"        "qr.resid"    "range
##  [97] "rcond"       "rep"         "round"       "rowMeans"    "rowSums"     "sign"
## [103] "signif"      "sin"         "sinh"        "sinpi"       "solve"       "sqrt"
## [109] "sum"         "summary"     "t"           "tan"         "tanh"        "tanpi
## [115] "tcrossprod"  "trigamma"    "trunc"       "unname"      "which"       "zapsm
```

10.2.3 Metodi

Ottenere i metodi disponibili per una classe Per conoscere i **metodi disponibili** per una funzione generica, nel sistema S3, si utilizza la funzione

`methods`:

```
library(xtable)
methods(xtable)

##  [1] xtable.anova*           xtable.aov*             xtable.aovlist*         xtable.coxph*
##  [5] xtable.data.frame*      xtable.glm*             xtable.lm*              xtable.matrix*
##  [9] xtable.prcomp*          xtable.summary.aov*     xtable.summary.aovlist* xtable.summary.
## [13] xtable.summary.lm*      xtable.summary.prcomp*  xtable.table*           xtable.ts*
## [17] xtable.zoo*
## see '?methods' for accessing help and source code
```

Si vede che il metodo specifico è memorizzata con un nome che parte dalla funzione generica di riferimento, separa con un punto e in seguito pone la classe dell'oggetto su cui il metodo opera (formato `generic.class`.
Per esaminare il codice non sempre è possibile stampare la funzione chiamandola perché diversi metodi sono *nascosti* (se chiamiamo in S3 methods per stampare i metodi, un metodo nascosto è stampato con a fianco un asterisco).
Per ottenere il **codice di un metodo S3** si usi la funzione `getS3method`, la chiamata è del tipo

```
getS3method(<generic>, <class>)
```

ad esempio

```
# print.xtabs ... NON FUNZIONA (hidden method)
getS3method("print","xtabs")

## function (x, na.print = "", ...)
## {
##     ox <- x
##     attr(x, "call") <- NULL
##     print.table(x, na.print = na.print, ...)
##     invisible(ox)
## }
## <bytecode: 0x556a809f6c78>
## <environment: namespace:stats>
```

Da notare che **sebbene alcuni metodi siano visibili, è meglio non chiamarli direttamente ma affidarsi alla dispatcher**.

Creare un nuovo metodo E' necessario semplicemente creare una funzione con gli stessi parametri e default della generica che abbia nome specificato dalla sintassi `generica.metodo`. Le regole da seguire sono:

- A method must have all the arguments of the generic, including ... if the generic does.
- A method must have arguments in exactly the same order as the generic.
- If the generic specifies defaults, all methods should use the same defaults.

Un modo semplice per rispettare tutte queste regole è **mantenere sempre la generica molto semplice** tipo:

```
scale <- function(x, ...) UseMethod("scale")
```

Bisogna eventualmente aggiungere parametri e default nella generica, se hanno un senso in tutti i possibili metodi che la implementano.

10.3 S4

I file help nel pacchetto `methods` sono la fonte primaria di **documentazione** sulla programmazione ad oggetti. Si veda:

- `?Classes`, `?Methods`
- `?setClass`, `?setMethod`, `?setGeneric`

Alcune sono tecniche ma vale la pena perseverare insieme alla pratica (sono indirizzate a programmatori, gli utilizzatori principali di classi e metodi).
In S4, attraverso pacchetto `methods`:

- una classe può esser definita con `setClass`.
- gli oggetti sono creati utilizzando `new`.

10.3.1 Funzioni generiche e metodi in S4

In S4 la funzione `show` è un esempio di funzione generica (sarebbe l'equivalente di print)

```
show

## standardGeneric for "show" defined from package "methods"
##
## function (object)
## standardGeneric("show")
## <bytecode: 0x556a7898cf38>
## <environment: 0x556a771f1f70>
## Methods may be defined for arguments: object
## Use  showMethods(show)  for currently available ones.
## (This generic function excludes non-simple inheritance; see ?setIs)
```

come dispatcher si usa la funzione `standardGeneric`. La funzione è diversa da `UseMethod` ma la funzione è la stessa.
Per vedere quali metodi sono disponibili per la funzione genrica si deve utilizzare `showMethods`

```
showMethods("show")
```

```
## Function: show (package methods)
## object="abIndex"
## object="ANY"
## object="BunchKaufman"
## object="C++Class"
## object="C++Function"
## object="C++Object"
## object="classGeneratorFunction"
## object="classRepresentation"
## object="denseMatrix"
## object="diagonalMatrix"
## object="dsyMatrix"
## object="dtrMatrix"
## object="envRefClass"
## object="externalRefMethod"
## object="function"
##     (inherited from: object="ANY")
## object="genericFunction"
## object="genericFunctionWithTrace"
## object="MatrixFactorization"
## object="MethodDefinition"
## object="MethodDefinitionWithTrace"
## object="MethodSelectionReport"
## object="MethodWithNext"
## object="MethodWithNextWithTrace"
## object="Module"
## object="namedList"
## object="ObjectsWithPackage"
## object="oldClass"
## object="pBunchKaufman"
## object="refClassRepresentation"
## object="refMethodDef"
## object="refObjectGenerator"
## object="rleDiff"
## object="signature"
## object="sourceEnvironment"
## object="sparseMatrix"
## object="sparseVector"
## object="standardGeneric"
##     (inherited from: object="genericFunction")
## object="traceable"
```

10.4 Esaminare il codice di funzioni S4

Per il codice di un metodo S4 la funzione è `getMethod`, la sintassi

`getMethod(<generic>, <signature>)`

Della signature parliamo a momenti.

10.4.1 Definizione di classi

Una **nuova classe** può esser definita esplicitamente in S4 utilizzando `setClass`:

- bisogna specificare al minimo il nome della classe
- si può inoltre specificare gli elementi di dati che la compongono, chiamati **slots**

definiamo una classe polygon mediante

```
setClass("polygon",
         representation(x = "numeric",
                        y = "numeric"))
```

Gli slots di questa classe sono x ed y (entrambi oggetti numerici, destinati a memorizzare vettori di coordinate x e y). Si può avere accesso agli slots di una classe S4 mediante l'operatore `@`.
Informazioni sulla definizione di una classe possono esser ottenute mediante `showClass`.

10.4.2 Definizione di metodi

In seguito alla definizione è possibile **definire metodi** per quella classe attraverso `setMethod`:

- è necessario specificare una *funzione generica*
- è necessario specificare una `signature`: la signature è un vettore di caratteri che indica la calsse di oggetti che sono accettati e processati dal metodo in questione

Nel caso precedente, voglio definire un metodo per la visualizzazione (funzione generica `plot`) dei poligoni (oggetto che costituisce la `signature`)

```
setMethod("plot", "polygon",
          function(x, y, ...) {
              ## plot... essendo argomenti numerici viene usato il default
              ## ovvero non creo un loop infinito
              ## impostiamo però solo la finestra
              plot(x@x, x@y, type="n", ...)
              ## ora aggiungo le figure
              xp <- c(x@x, x@x[1])
              yp <- c(x@y, x@y[1])
              lines(xp, yp)
          })
```

Si noti che:

- nella definizione della funzione gli argomenti devono matchare esattamente quelli della generica plot (che è appunto `plot(x,y,...)`
- all'interno della funzione accediamo agli elementi attraverso `@`, ovvero il primo x è l'oggetto cui ci si riferisce, il secondo x il suo slot

In generale dall'impostazione di classi e metodi, non viene stampato del gran output (a meno che non vi siano errori); vi è però il side effect che verranno create classi e metodi (che esistono in memoria sino alla chiusura del programma, quindi sarà necessaria una ri-definizione).

Dopo che abbiamo creato un metodo, esso sarà aggiunto alla lista dei metodi per la generica

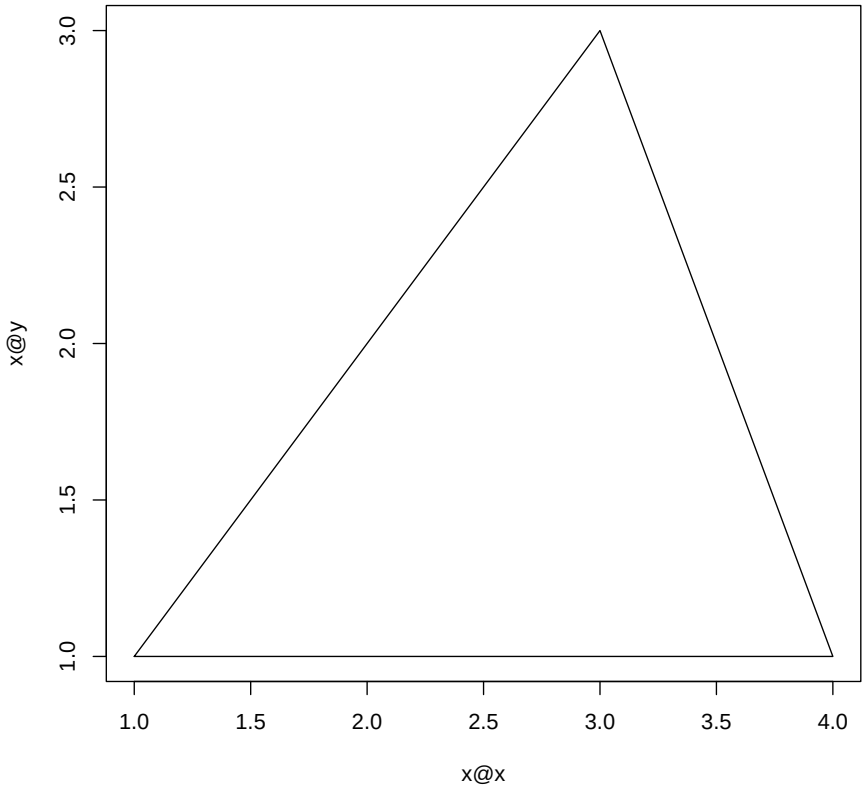
```
showMethods("plot")

## Function: plot (package base)
## x="ANY"
## x="polygon"
```

Vi sono due metodi: il primo è `ANY` (il default method in S4), il secondo è il nostro polygon.

10.4.3 Instanziazione di oggetti

Per creare un nuovo oggetto utilizziamo `new`

```
p <- new("polygon", x=c(1,2,3,4), y=c(1,2,3,1))
plot(p)
```


x@y
3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
x@x

Capitolo 11

Performance

11.1 Esecuzione parallela

L'esecuzione di task in parallelo si effettua mediante il pacchetto `parallel`. Nel seguito si vedono alcuni template di elaborazioni e le funzioni principali del pacchetto

11.1.1 Setup e chiusura

Il setup tipico (e portabile anche in Windows) è il seguente:

```
library(parallel)
nc <- detectCores()
cl <- makeCluster(nc)
```

Abbiamo creato un cluster di nodi in base alla disponibilità di processori della macchina. In seguito utilizziamo le funzioni del pacchetto fornendo come cluster la variabile `cl` creata.

Il cluster è composto da processi di R assolutamente isolati ed indipendenti, che in particolar modo non ereditano dal processo che li ha creati librerie caricate e dati disponibili nel workspace. Per questi motivi:

- eventuali librerie dovranno essere specificate nel codice (ad esempio mediante `library` o `require`) passato ai cluster,
- mentre per l'impostazione dell'ambiente e dei dati disponibili (se ad esempio i cluster devono effettuare elaborazioni su dati che non vengono passati come parametro di funzione, sfruttando il lexical scope) è necessario esportarle mediante

  ```
  nome_var <- 1
  clusterExport(cl, 'nome_var')
  ```

Una volta terminato è necessario **stoppare il cluster** mediante

```
stopCluster(cl)
```

11.1.2 Funzioni utili per l'esecuzione parallela

`parLapply` È un equivalente di `lapply`

```
parLapply(cl, X = 4:7, fun = function(x) mean(rnorm(10^x)))
```

`clusterMap` È un equivalente di `Map`; ad esempio per calcolare delle statistiche descrittive in parallelo sul medesimo vettore casuale possiamo fare:

```
clusterMap(cl, function(f, y) {
    library(stats)
    f(y)
},
list(mean, sd, median),
list(rnorm(100000))[rep(1,3)]
)
```

Settaggio di semi per la generazione di numeri casuali Per settare il seme per generare numeri casuali in parallelo si usa `clusterSetRNGStream`:

```
f <- function(x) {x; mean(rnorm(100))}

clusterSetRNGStream(cl, 1234)
unlist(parLapply(cl, X = 1:4, fun = f))

clusterSetRNGStream(cl, 1234)
unlist(parLapply(cl, X = 1:4, fun = f))
```

11.2 Analisi automatica del codice

Il pacchetto `codetools` mette a disposizione strumenti per l'analisi del codice di una funzione per trovare eventuali problemi. Un esempio

```
library(codetools)
## Funzione per effettuare controlli di default su una funzione
checkUsage(lbsurv::km, all = TRUE, name = "g")

## g: Error while checking: non c'è alcun pacchetto chiamato 'lbsurv'

## Funzione per guardare i nomi del namespace globale chiamati da
## una funzione
findGlobals(lbsurv::km)[1:30]
```

```
## Error in loadNamespace(x):  non c'è alcun pacchetto chiamato 'lbsurv'
```

```
## Funzione per controllare un intero pacchetto
library(lbmisc)
checkUsagePackage("lbmisc")

## compare_columns: no visible binding for global variable 'first'
## compare_columns: no visible binding for global variable 'second'
## plot_pfun: local variable 'z' assigned but may not be used
## preprocess_data: local variable 'mr_pos' assigned but may not be used
## preprocess_data: local variable 'unique' assigned but may not be used
## show_col : showCols.graphics: local variable 'n' assigned but may not be used
## show_col : showCols.grid: local variable 'n' assigned but may not be used
```

11.3 Timing

Il **timing** delle procedure puó esser effettuato:

- mediante `system.time`; questa è una utile funzione per confrontare, sotto il profilo dell'efficienza, procedure alternative utilizzabili per svolgere il medesimo compito

  ```
  system.time(mean(rnorm(1e+08)))
  ```

-

mediante il pacchetto `microbenchmark` fornisce una comoda interfaccia per il confronto di procedure in *competizione* fra loro. Un esempio:

```
library(microbenchmark)
a <- function(x) sum(rnorm(x))
b <- function(x) {
    res <- c()
    for (i in 1:x){
        res <- c(res, rnorm(1))
    }
}
microbenchmark(a(100), b(100), times = 1000)
```

11.4 Code profiling

Fare *profiling* del codice significa determinare quanto tempo di esecuzione un programma impiega in differenti sezioni del codice. Questo serve per individuare quali parti dell'algoritmo costituiscono un *bottleneck*.

Le statistiche sul tempo di esecuzione di una determinate procedure possono esser disaggregate a livello di singola funzione in esso impiegata.

Il confronto di due funzioni può esser utile per presentare i concetti. Le due funzioni non hanno un significato pratico, ma evidentemente la seconda impiega più tempo di esecuzione della prima.

```
## Define two (competing) functions
fun1 <- function(x) mean(rnorm(x))
fun2 <- function(x) mean(rnorm(x*2))
```

Ora impostiamo la simulazione: eseguiamo le due funzioni all'interno di due chiamate ad `Rprof`. Nella prima chiamata si specifica il file dove verranno salvate le statistiche di esecuzione.
Cosa succede in R, all'interno del *setting* di `Rprof`? Il Rprof salva nei log specificati ogni 20 msec, quale funzione di R è in utilizzo[1]. Occorrerà in una seconda fase sintetizzare questi dati.
Si noti che, nell'esempio a seguire, prima di ogni misurazione si esegue `gc` per far si che le procedure per liberare memoria non avvengano verosimilmente quando si temporizza il codice (sporcando la stima).

```
reps <- 1000000

gc()
Rprof("profTest1.out")
fun1(reps)
Rprof(NULL)

gc()
Rprof("profTest2.out")
fun2(reps)
Rprof(NULL)
```

Ora abbiamo eseguito le due funzioni e salvato le statistiche nei due file; per avere alcune sintesi utili vi sono due strade. Da linea di comando

```
R CMD Rprof profTest1.out > funct1stat.txt
R CMD Rprof profTest2.out > funct2stat.txt
```

Questo primo modo necessita di Perl ma funziona più efficientemente per file .out lunghi (es lunghe procedure, o a parità di tempo se si imposta un monitoraggio ad intervalli più stretti). L'interpretazione dei file creati è immediata.

Alternativamente, dall'interno di R si può utilizzare `summaryRprof` (alternativa più portabile, non necessita di Perl):

```
test1 <- summaryRprof("profTest1.out")
test2 <- summaryRprof("profTest2.out")
```

Nel seguito commentiamo questi due oggetti:

[1] Il monitoraggio del codice in se comporta l'utilizzo di tempo macchina, quindi i tempi di esecuzione complessivi sono meno affidabili di quelli dati da `system.time`.

```
## Tempo complessivo impiegato per le due procedure
test1$sampling.time
test2$sampling.time

## Distribuzione del tempo per funzione che lo ha
## utilizzato
test1$by.self
test2$by.self
```

nello specifico occorre guardare a `self.time` (secondi) `self.pct`(%).

11.5 Interfaccia con il linguaggio C

Ci poniamo nell'ottica di scrivere codice in C/C++:

- da utilizzare nell'ambito di un pacchetto;
- da definire e utilizzare on the fly.

11.5.1 Codice compilato in pacchetti

11.5.1.1 Setup del pacchetto

Utilizzeremo le funzioni `.C` e `.Call`; indipendentemente da ciò, per far funzionare il tutto

- poniamo i nostri file `.c` in `src`;
- nell'ipotesi di usare `roxygen2`, aggiungiamo la linea `#' useDynLib nomePacchetto` al file `nomePacchetto-package.R`, all'interno della directory `R`.

La compilazione funziona come segue:

- se in `src` non è presente un `Makefile`, `R CMD INSTALL` farà di suo la compilazione (utilizzando il `Makevars` eventualmente presente). Questa è la strada **consigliata**.
- se invece è presente un `Makefile` sarà utilizzato `make` con le configurazioni di (in ordine) `/usr/lib/R/etc/Makeconf`, `src/Makefile` e `Makevars`.[2]

11.5.1.2 L'utilizzo di `.C`

Tradizionalmente la chiamata di procedure C è stata affrontata mediante la funzione `.C`; questa è sufficiente se dobbiamo solamente manipolare vettore e non dobbiamo effettuare chiamate ad R.

[2] È meglio utilizzare la prima strada; `make` dovrebbe esser necessario di rado (e in tal caso il `Makefile` deve essere scritto in maniera saggiamente portabile).

Tipo R	Tipo C
`logical`	`int *`
`integer`	`int *`
`double`	`double *`
`character`	`char **`

Tabella 11.1: Tipi in C e in R

Scrittura del codice C Le funzioni che scriviamo in C devono avere le seguenti proprietà:

- le funzioni devono ritornare `void`, e ritornano qualcosa al livello di R tramite un parametro apposito passato come argomento per memorizzare i risultati;
- tutti gli argomenti passati al C, sono passati per *valore*, quindi sono copie che non intaccano l'originale dato in argomento;
- ogni file `.c` che deve esser chiamato da R deve contenere l'inclusione dell'header `<R.h>`.

Un esempio segue

```
#include <R.h>
void convolve(double *a, int *na, double *b, int *nb, double *ab)
{
   int i, j, nab = *na + *nb - 1;
   for(i = 0; i < nab; i++)
     ab[i] = 0.0;
   for(i = 0; i < *na; i++)
     for(j = 0; j < *nb; j++)
       ab[i + j] += a[i] * b[j];
}
```

Scrittura del wrapper R Per utilizzare il codice C, dobbiamo creare un wrapper di .C all'interno di R:

```
.C(.NAME, ..., NAOK = FALSE, PACKAGE)
```

Il primo argomento di `.C` è la stringa che consiste nel nome della funzione da chiamare; dopo si possono porre fino a 65 argomenti che verranno presi, nello stesso ordine, all'interno della funzione C.
Se agli argomenti vengono dati nomi, questi vengono utilizzati solo in R (per memorizzare i risultati quando la computazione ritorna), non all'interno del C. La tabella 11.1 mappa alcuni tipi di R gli equivalenti che si manipolerà all'interno della funzione C. Dopo i parametri passati alla funzione C si può controllare il comportamento mediante alcuni parametri aggiuntivi:

- se il parametro `NAOK` è a `FALSE` (di default), la presenza di `NA`, `NaN` ecc genera errori; se a `TRUE`, passa tutto inosservato;
- il parametro `PACKAGE` specifica in quale codice oggetto senza estensione (es per `foo.so` sarà `"foo"`) cercare la funzione; è vivamente consigliato il suo utilizzo;

Un esempio segue:

```
conv <- function(a, b)
       .C("convolve",
          as.double(a),
          as.integer(length(a)),
          as.double(b),
          as.integer(length(b)),
          ab = double(length(a) + length(b) - 1))$ab
```

In generale:

- se si passano vettori bisogna passarne anche la lunghezza perché il C non lo sa a priori;
- bisogna coercire esplicitamente allo R *storage type* equivalente, per evitare errori;
- ciò che viene ritornato dalla funzione C è una lista che contiene gli argomenti passati al codice compilato, eventualmente cambiati all'interno di esso; di quella lista dobbiamo tenere il vettore o valore di risultato tramite un indice di lista (con nome o senza, dipende se lo abbiamo posto nella chiamata di `.C`), in questo caso mediante `$ab`;
- occorre in sede di chiamata generare un vettore che sarà destinato a contenere i risultati della computazione perché il meccanismo non funziona con `return` del C.

Per ciò che riguarda il trattamento delle stringhe, queste possono essere accorciate, ma non aumentate oltre alla lunghezza della stringa originale che include il risultato finale. La funzione `.Call` è migliore per la gestione delle stringhe.

11.5.1.3 Api di R

Vi si ha accesso mediante l'inclusione di `R.h`. Vediamo alcune macro o funzioni che possono tornare utili.

Il booleano è definito in `R_ext/Boolean.h` come

```
typedef enum { FALSE = 0, TRUE /*, MAYBE */ } Rboolean;
```

Cose che possono essere utili da `R_ext/Error.h`

```
void NORET Rf_error(const char *, ...);
void    Rf_warning(const char *, ...);
void    R_ShowMessage(const char *s);
```

Da `R_ext/Memory.h`

```
char*   R_alloc(size_t, int);
```

Da `R_ext/Print.h`

```
void Rprintf(const char *, ...);
void REprintf(const char *, ...);
```

11.5.1.4 L'utilizzo di `.Call`

Se si desidera scrivere codice C che usi le strutture dati interne di R bisogna utilizzare `.Call` o `.External`; questo però impone una programmazione più complessa rispetto a `.C`.
Sia `.Call` che `.External` possono essere utilizzate, oltre che per il C, anche per il C++; qui ci si concentra su `.Call`, che ha un funzionamento lievemente più semplice, mentre `.External` è lievemente più generale.
Va detto che sia `.Call` che `.External` usano una **chiamata per riferimento**, il che è più efficiente, ma per sicurezza **bisognerebbe trattare gli argomenti che si ricevono solamente in lettura**.

In questa sezione vediamo l'impostazione minimale per `.Call`; a partire dal prossimo ci concentriamo sulle strutture dati e tutto ciò che è necessario per programmare in C usando `.Call`.

Scrittura del codice C Un lato C minimale:

```
#include <R.h>
#include <Rinternals.h>

SEXP foo(SEXP x)
{
    return x;
}
```

Scrittura del wrapper R L'interfaccia dal lato R è

```
.Call(.NAME, ..., PACKAGE)
```

dove i parametri assumono lo stesso significato visto in precedenza per `.C`. Un esempio minimale segue

```
foo <- function(x) .Call("foo", x)
```

11.5.1.5 Aspetti salienti di programmazione interna

Oggetti R a livello C Tutti gli oggetti con i quali si ha a che fare, sono rappresentati internamente come `SEXP` ovvero un puntatore (opaco) ad una struttura `SEXPREC` che contiene le informazioni e i dati sull'oggetto.
`SEXPREC` è una struttura definita come:

```
typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
        struct primsxp_struct primsxp;
        struct symsxp_struct symsxp;
        struct listsxp_struct listsxp;
        struct envsxp_struct envsxp;
        struct closxp_struct closxp;
        struct promsxp_struct promsxp;
    } u;
} SEXPREC, *SEXP;
```

SEXPTYPE	R equivalent
LGLSXP	logical
INTSXP	integer
REALSXP	numeric with storage mode double
CPLXSXP	complex
STRSXP	character (vettore di stringhe)
CHARSXP	stringa scalare
VECSXP	list (generic vector)
CLOSXP	function or function closure
ENVSXP	environment
LISTSXP	pairlist
DOTSXP	a ... object
NILSXP	NULL
SYMSXP	name/symbol
FREESXP	nodo rilasciato dal GC
FUNSXP	closure o builtin

Tabella 11.2: SEXPTYPEs

Vediamo che ha un header definito come

```
struct sxpinfo_struct sxpinfo; \
struct SEXPREC *attrib; \
struct SEXPREC *gengc_next_node, *gengc_prev_node
```

sxpinfo è una struttura che contiene informazioni sull'oggetto stiamo manipolando, attrib è un puntatore ad un'altra SEXPREC che contiene gli attributi dell'oggetto, mentre gengc_next_node e gengc_prev_node sembrano i nodi ad altre strutture collegate alla presente (tipo a-la lista double linked). La parte di dati è una union che contiene diverse strutture a seconda del tipo di oggetto considerato; a parte primsxp che è un intero, gli altri sono gruppi di puntatori sxpinfo' definita come:

```
struct sxpinfo_struct {
    SEXPTYPE type      :  TYPE_BITS; /* attualmente 5 bits*/
    unsigned int obj   :  1;
    unsigned int named :  2;
    unsigned int gp    : 16;
    unsigned int mark  :  1;
    unsigned int debug :  1;  /* utilizzato per closures/environment */
    unsigned int trace :  1;  /* functions and memory tracing */
    unsigned int spare :  1;  /* currently unused */
    unsigned int gcgen :  1;  /* old generation number */
    unsigned int gccls :  3;  /* node class */
}; /*               Tot: 32 */
```

A specificare la tipologia di oggetto è un intero chiamato SEXPTYPE. I tipi più comuni sono riportati in tabella 11.2. Si noti che la struttura interna di SEXPREC non è resa disponibile: piuttosto SEXP è un puntatore opaco e i dati interni possono essere acceduti solamente dalle funzioni fornite.

Vi sono anche forme alternative utilizzate per i vettori, ovvero VECSXP (equivalente di SEXP) che puntano a strutture VECTOR_SEXPREC, definite come:

```
typedef struct VECTOR_SEXPREC {
    SEXPREC_HEADER;
    struct vecsxp_struct vecsxp;
} VECTOR_SEXPREC, *VECSEXP;
```

Quindi l'header è il medesimo e si differenzia per la struttura dati (che sono semplicemente due interi che definiscono la lunghezza del vettore); per muoversi nel vettore si utilizzerà i puntatori di lista linkata.

```
The vector types are RAWSXP, CHARSXP, LGLSXP, INTSXP, REALSXP, CPLXSXP,
STRSXP, VECSXP, EXPRSXP and WEAKREFSXP. Remember that such types are a
VECTOR_SEXPREC, which again consists of the header and the same three
pointers, but followed by two integers giving the length and 'true length'3
of the vector, and then followed by the data
               ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Manipolazione a livello C Gli oggetti di R debbono essere manipolati in C attraverso un set di funzioni e di macro (per la parte *pubblica* risedenti in `Rinternals.h`[3], che sono le stesse funzioni e macro sono quelle utilizzate per implementare le parti core di R.
Una gran parte di R e di pacchetti standard è implementata usando le funzioni e le macro che si descriveranno in seguito.

Garbage collection e creazione di oggetti La memoria allocata all'interno di una funzione C non viene de-allocata dall'utente, ma dal **garbage collector** a fasi cicliche.
Dopo aver creato un oggetto R all'interno del codice C, bisogna applicare la macro `PROTECT` su un puntatore all'oggetto; questo fa in modo si dice a R che l'oggetto è in uso e che non lo distrugga durante la garbage collection. La protezione non è necessaria per oggetti che R sa di star utilizzando, es argomenti di funzioni.
Appena non si ha più bisogno di un oggetto occorre chiamare la macro `UNPROTECT` che prende come argomento un `int` che indica il numero di oggetti da un-proteggere (funziona a stack, quindi LIFO, ovvero gli ultimi ad essere protetti saranno i primi ad essere un-protetti).
Le chiamate a `PROTECT` e `UNPROTECT` debbono essere bilanciate quando la funzione ritorna, alternativamente verrà dato warning. (`stack imbalance`).

Per la **creazione di oggetti** dobbiamo dichiarare l'oggetto e in seguito **allocare memoria**; per l'allocazione di memoria sono disponibili diverse macro dal nome `allocXxxx` che si possono trovare in `Rinternals.h`. Una comune è ad esempio `allocVector` che alloca un vettore di tipo e dimensione specificata (e inizializza nel caso liste, espressioni e vettori di caratteri)

Un esempio di dichiarazione, allocazione e protezione di un vettore di 2 valori `double`, fatto all'interno del C, è:

```
SEXP ab;
...
ab = PROTECT(allocVector(REALSXP, 2));
```

Per allocare oggetti di C (non oggetti R) utilizzare `R_alloc` (si veda le API di R)

[3] L'inclusione di `R.h` rimane necessaria

Lettura e scrittura di vettori Si può usare `length` su un vettore che restituisce un `int` con la sua lunghezza.

Vi è una helper function per ogni vettore atomico che permette di accedere all'array C che memorizza i dati; occorre utilizzare `REAL()`, `INTEGER()`, `LOGICAL()`. Quello che queste macro restituiscono sono puntatori, quindi le possiamo usare sia in lettura che scrittura. Un esempio segue

```
#include <R.h>
#include <Rinternals.h>

SEXP calltest1_slave(SEXP x){

    int n = length(x);
    SEXP out = PROTECT(allocVector(INTSXP, n));
    int *pout = INTEGER(out);

    for(int i = 0; i < n; i++)
*(pout + i) = INTEGER(x)[i] + 1;

    UNPROTECT(1);
    return out;

}
```

Valori missing e simili Ogni vettore atomico ha associato una costante speciale per ottenere o impostare valore mancanti:

- `INTSXP: NA_INTEGER`
- `LGLSXP: NA_LOGICAL`
- `STRSXP: NA_STRING`

I missing per `REALXSP` sono più complessi: utilizzare `ISNA(x)`, `ISNAN(x)` o `!R_FINITE()` per controllare NA, NaN o valori infiniti. Usare le costanti `NA_REAL`, `R_NaN`, `R_PosInf` e `R_NegInf` per impostare tali valori.

Vettori di caratteri e liste Stringhe e liste sono più complicate di vettori numerici perché i singoli elementi sono essi stessi delle `SEXP`.

Ogni elemento di una `STRSXP` è un `CHARSXP`, un oggetto immutabile che contiene un puntatore ad una stringa C memorizzata in un pool globale:

- utilizzando `STRING_ELT(x, i)` si estrae il `CHARSXP`, e mediante `CHAR(STRING_ELT(x, i))` per ottenere il `const char *`: le stringhe vanno pertanto considerate solamente per la lettura.
- i valori si assegnano mediante `SET_STRING_ELT(x, i, value)`
- utilizzare `mkChar()` per trasformare una stringa C in un `CHARSXP` (usato per creare una stringa e inserirla in un vettore)
- utilizzare `mkString()` per trasformare una stringa C in un `STRSXP` (usato per creare un nuovo vettore stringa di lunghezza 1

Gli elementi di una lista possono essere SEXP d qualsiasi tipo:

- `VECTOR_ELT(x, i)` serve per accedere agli elementi di un vettore
- `SET_VECTOR_ELT(x, i, value)` serve per modificare

Modifica degli input La modifica dei dati passati in input è sconsigliata. Se si desidera farlo occorre creare una copia all'interno del C mediante `duplicate`

```
SEXP foo(SEXP x){

   SEXP x_copy = PROTECT(duplicate(x));
   ...
}
```

dopodiché lavorare sulla copia

Test di oggetti Per effettuare il **test** che un oggetto sia di un tipo appropriato:

- esistono le funzioni `isType` (es `isReal`) che restituiscono un valore booleano.
- si può usare la macro `TYPEOF` (che ritorna costanti pari ai valori di `LGLSXP`, `INTSXP` eccetera)

Coercizione di vettori Qualora:

- un vettore non sia del tipo desiderato[4]
- si desideri assicurarsi che il vettore input sia valido

si può **coercire** per sicurezza mediante `coerceVector`[5] .
Ad esempio se abbiamo che un `SEXP` è di tipo `INTEGER`, ma si desidera avere un oggetto `REAL` object, lo si crea mediante:

```
SEXP newSexp = PROTECT(coerceVector(oldSexp, REALSXP));
```

La protezione è resa necessaria perché un nuovo oggetto viene creato

Coercizione di scalari Vi sono alcun funzioni helper che trasformano un vettore R di lunghezza unitaria in una variabile scalare C

- `asLogical(x):  LGLSXP -> int`
- `asInteger(x):  INTSXP -> int`
- `asReal(x):  REALSXP -> double`
- `CHAR(asChar(x)):  STRSXP -> const char*`

E viceversa

[4] In questo caso alternativamente alla coercizione si può generare un **errore** mediante la funzione `error`

[5] Vedere `Type Coercions of all kinds` in `Rinternals`

- `ScalarLogical(x): int -> LGLSXP`
- `ScalarInteger(x): int -> INTSXP`
- `ScalarReal(x): double -> REALSXP`
- `mkString(x): const char* -> STRSXP`

Questi creano tutti oggetti di R, quindi debbono essere protetti.

Altre funzioni/macro utili Vediamo alcune macro/funzioni utili e la loro definizione/dichiarazione

```
#define TYPEOF(x)       ((x)->sxpinfo.type)
#define ATTRIB(x)       ((x)->attrib)
# define LENGTH(x)      (((VECSEXP) (x))->vecsxp.length)
# define IS_SCALAR(x, type) (TYPEOF(x) == (type) && LENGTH(x) == 1)
```

11.5.2 Uso di `inline`

Il pacchetto `inline` permette di compilare, caricare ed eseguire codice C, C++ e Fortran.

Ipotizzando di dover utilizzare una funzione da chiamare con `.Call` occorre creare la funzione compilata mediante la funzione `cfunction` dove si specifica nel primo parametro (`signature`, che non è necessario se la funzione non prende input) un vettore di argomenti (nome e tipo) e nel secondo il codice che costituisce il corpo della funzione (`body`)

```
library(inline)

## Error in library(inline): non c'è alcun pacchetto chiamato 'inline'

sig <- c(a = "integer", b = "integer")
body <- "  SEXP result = PROTECT(allocVector(REALSXP, 1));
           REAL(result)[0] = asReal(a) + asReal(b);
           UNPROTECT(1);
           return result;"
add <- cfunction(sig = sig, body = body)

## Error in cfunction(sig = sig, body = body): non trovo la funzione
"cfunction"

add(1,3)

## Error in add(1, 3): non trovo la funzione "add"
```

Capitolo 12

Data management

12.1 Sorting

Per ordinare gli oggetti sulla base di qualche parametro si possono utilizzare le funzioni `sort` o `order`: sort ha un utilizzo più limitato e sorta diretto, order fornisce l'ordine degli indici dell'oggetto di partenza che garantisce che questo sia ordinato rispetto alla variabile considerata. Partiamo dal caso del vettore:

```
x <- c(2, 5, 1, 4)
sort(x)

## [1] 1 2 4 5

order(x)

## [1] 3 1 4 2
```

È possibile **ordinare** anche in maniera **decrescente** ("artigianalmente" mediante `rev` o specificando l'apposita opzione di funzione):

```
rev(sort(x))

## [1] 5 4 2 1

sort(x, decreasing = TRUE)

## [1] 5 4 2 1

rev(order(x))

## [1] 2 4 1 3

order(x,decreasing=T)

## [1] 2 4 1 3
```

Per cio' che riguarda i **data.frame**, si potrebbe voler *ordinare il df rispetto ad una o più variabili, in maniera crescente o decrescente*. In questo torna utile `order`, perchè fornisce gli indici

```
sei
```

```
##   educ age inc male   name
## 1    5  30  23    1 gianni
## 2   10  24  12    0  elisa
## 3   13  45  45    1  mario
## 4   10  50  56    1 franco
## 5   20  90  67    0 giulia
## 6   15  70  53    0  giada
```

```
order(sei$age)
```

```
## [1] 2 1 3 4 6 5
```

```
sei[order(sei$age),]
```

```
##   educ age inc male   name
## 2   10  24  12    0  elisa
## 1    5  30  23    1 gianni
## 3   13  45  45    1  mario
## 4   10  50  56    1 franco
## 6   15  70  53    0  giada
## 5   20  90  67    0 giulia
```

Sulla base di due variabili, età e sesso, la prima crescente la seconda decrescente (si noti il - prima di `sei$male`, per aver l'ordinamento inverso); in questi casi la seconda variabile (e le altre eventualmente a venire) è utilizzata nel caso in cui non si riesca ad avere un ordinamento solo sulla base della prima (o delle precedenti).

```
sei[order(sei$age,-sei$male),]
```

```
##   educ age inc male   name
## 2   10  24  12    0  elisa
## 1    5  30  23    1 gianni
## 3   13  45  45    1  mario
## 4   10  50  56    1 franco
## 6   15  70  53    0  giada
## 5   20  90  67    0 giulia
```

E' importante anche vedere **come il sorting tocchi i valori mancanti** (`NA`): creiamone un paio

```
sei$educ[2]<-NA
sei$inc[4]<-NA
sei
```

```
##   educ age inc male   name
## 1    5  30  23    1 gianni
## 2   NA  24  12    0  elisa
## 3   13  45  45    1  mario
## 4   10  50  NA    1 franco
## 5   20  90  67    0 giulia
## 6   15  70  53    0  giada
```

Nella funzione order, il valore di `na.last` è `TRUE`: quindi i `NA` vengono posti alla fine:

```
sei[order(sei$educ),]

##   educ age inc male   name
## 1    5  30  23    1 gianni
## 4   10  50  NA    1 franco
## 3   13  45  45    1  mario
## 6   15  70  53    0  giada
## 5   20  90  67    0 giulia
## 2   NA  24  12    0  elisa

# Possiamo dire di mettere gli \texttt{NA} all'inizio
sei[order(sei$educ, na.last=F),]

##   educ age inc male   name
## 2   NA  24  12    0  elisa
## 1    5  30  23    1 gianni
## 4   10  50  NA    1 franco
## 3   13  45  45    1  mario
## 6   15  70  53    0  giada
## 5   20  90  67    0 giulia

# Oppure di non elencarli proprio
sei[order(sei$educ, na.last=NA),]

##   educ age inc male   name
## 1    5  30  23    1 gianni
## 4   10  50  NA    1 franco
## 3   13  45  45    1  mario
## 6   15  70  53    0  giada
## 5   20  90  67    0 giulia
```

12.2 Sampling

Se vogliamo estrarre un campione da un data.frame; di base l'estrazione casuale viene fatta mediante `sample`. Combinando sample e l'utilizzo degli indici arriviamo a ciò che è richiesto. Ad esempio per estrarre dal database `sei` un campione di 3 soggetti basta comandare

```
sei[sample(1:nrow(sei),3),]

##   educ age inc male  name
## 2   NA  24  12    0 elisa
## 6   15  70  53    0 giada
## 3   13  45  45    1 mario
```

si vede che i soggetti non sono ordinati (perchè la funzione `sample` ha restituito `1 5 3` che sono finiti pari pari ad indice); per evitare ciò basta sortare il sample

```
sei[sort(sample(1:nrow(sei),3)),]

##   educ age inc male   name
## 1    5  30  23    1 gianni
## 2   NA  24  12    0  elisa
## 5   20  90  67    0 giulia
```

12.3 Splitting

Si può voler separare un dataframe (o un vettore) in due sulla base di una determinata caratteristica : la funzione per farlo e' `split`. Ad esempio vogliamo dividere `sei` in due dataset, uno per i maschi e l'altro per le donne.

```
split(sei,sei$male)

## $`0`
##   educ age inc male   name
## 2   NA  24  12    0  elisa
## 5   20  90  67    0 giulia
## 6   15  70  53    0  giada
##
## $`1`
##   educ age inc male   name
## 1    5  30  23    1 gianni
## 3   13  45  45    1  mario
## 4   10  50  NA    1 franco
```

Si crea una lista accessibile mediante il `$gruppo`. Pertanto per creare due dataframe sulla base del risultato precedente possiamo procedere come segue:

```
sei.maschi  <- split(sei, sei$male)[["1"]]
sei.femmine <- split(sei, sei$male)[["0"]]
```

Lo stesso è possibile fare con semplici vettori.

12.4 Subset

Oltre all'utilizzo degli indici, per selezionare un sottoinsieme di un dataset l'utente ha a disposizione la funzione `subset`; considerando `sei`

```
subset(sei, inc > 30)

##   educ age inc male   name
## 3   13  45  45    1  mario
## 5   20  90  67    0 giulia
## 6   15  70  53    0  giada

subset(sei, inc > 30, male)

##   male
## 3    1
## 5    0
## 6    0
```

Ma anche introdurre **selezione condizionale**:

```
subset(sei, (inc > 30) & (educ < 15), c(male, age))

##   male age
## 3    1  45

subset(sei, (inc > 30) & (educ < 15),-male)

##   educ age inc  name
## 3   13  45  45 mario
```

12.5 Merging

Vogliamo studiare le relazioni tra eta e reddito dei soggetti inclusi nei seguenti dataframe

```
(a <- data.frame("nome" = c("Anna", "Luca", "Massimo", "Giuseppe", "Piero"),
                 "eta" = c(18, 23, 32, 17, 34)))

##       nome eta
## 1     Anna  18
## 2     Luca  23
## 3  Massimo  32
## 4 Giuseppe  17
## 5    Piero  34

(b <- data.frame("nome" = c("Piero", "Paolo", "Anna", "Nadia", "Stefano"),
                 "reddito" = c(150, 241, 120, 100, 133)))

##      nome reddito
## 1   Piero     150
## 2   Paolo     241
## 3    Anna     120
## 4   Nadia     100
## 5 Stefano     133
```

È necessario associare ad ogni soggetto il suo reddito (se questo è presente nel data.frame b), anche se si nota che i due database presentano dei buchi, se considerati congiuntamente (es di Paolo non si sa l'età). Lo si fa mediante la funzione `merge`; di base restituisce solo le unita' che presentano valori in entrambi i database

```
merge(a, b)

##    nome eta reddito
## 1  Anna  18     120
## 2 Piero  34     150

merge(a, b, all.x = TRUE)

##       nome eta reddito
## 1     Anna  18     120
## 2 Giuseppe  17      NA
## 3     Luca  23      NA
## 4  Massimo  32      NA
## 5    Piero  34     150

merge(a, b, all.y = TRUE)

##      nome eta reddito
## 1    Anna  18     120
## 2   Nadia  NA     100
## 3   Paolo  NA     241
## 4   Piero  34     150
## 5 Stefano  NA     133

merge(a, b, all = TRUE)

##       nome eta reddito
## 1     Anna  18     120
## 2 Giuseppe  17      NA
## 3     Luca  23      NA
## 4  Massimo  32      NA
## 5    Nadia  NA     100
## 6    Paolo  NA     241
## 7    Piero  34     150
## 8  Stefano  NA     133
```

12.6 Binding

Questo e' possibile farlo mediante due comandi come `cbind` (che unisce per le colonne, ma i due dataset devono avere lo stesso numero di righe) e `rbind` (che unisce per le righe, ma i due dataset devono avere le stesso numero di colonne). Questi prendono vettori, matrici o data.frame e li combinano per colonne o righe rispettivamente; suddivido il solito dataset in due parti per poi utilizzare `*bind`

```
(sei1 <- sei[1:2])

##   educ age
## 1    5  30
## 2   NA  24
## 3   13  45
## 4   10  50
## 5   20  90
## 6   15  70

(sei2 <- sei[3:4])

##   inc male
## 1  23    1
## 2  12    0
## 3  45    1
## 4  NA    1
## 5  67    0
## 6  53    0
```

Unione per colonne se si vuol maneggiare con variabili differenti delle stesse unita' statistiche: seconda colonna del primo con la prima del secondo

```
cbind(sei1[2], sei2[1])

##   age inc
## 1  30  23
## 2  24  12
## 3  45  45
## 4  50  NA
## 5  90  67
## 6  70  53
```

unione per righe: utile se si desiderano aggiungere nuove unità

```
> rbind(sei1,sei2)
Errore in match.names(clabs, names(xi)) : names do not match previous names:
inc, male
```

R è "furbo" e ci dice che stiamo mettendo sotto la stessa variabile cose diverse.
Lo eludiamo mediante

```
> names(sei1) <- names(sei2)
> rbind(sei1,sei2)
   inc male
1    5   30
2   10   24
3   13   45
4   10   50
5   20   90
6   15   70
7   23    1
```

```
8   12    0
9   45    1
10  56    1
11  67    0
12  53    0
```

12.7 Suddivisione in classi

Per i dati numerici, puo' esser utile la suddivisione in classi. Puo' essere effettuata mediante la funzione cut(), la quale crea una variabile categoriale (factor) dividendo la variabile numerica che figura come suo argomento in intervalli. Un esempio con classi di età

```
> eta <- c(2,29,18,13,4,35,46,21,98,54,76,43)
> cleta<-data.frame(eta)
> cleta
   eta
1    2
2   29
3   18
4   13
5    4
6   35
7   46
8   21
9   98
10  54
11  76
12  43

> breaks=c(0,13,20,60,eta[which.max(eta)]+1)
> cleta$cleta1 <- cut(eta,breaks)

> cleta
   eta  cleta1
1    2  (0,13]
2   29 (20,60]
3   18 (13,20]
4   13  (0,13]
5    4  (0,13]
6   35 (20,60]
7   46 (20,60]
8   21 (20,60]
9   98 (60,99]
10  54 (20,60]
11  76 (60,99]
12  43 (20,60]

> attributes(cleta$cleta1)$class
[1] "factor"
```

```
> cleta$cleta1b <- cut(eta,breaks,labels=c("kid","teen","man","elder"))

> cleta
   eta  cleta1 cleta1b
1    2  (0,13]     kid
2   29 (20,60]     man
3   18 (13,20]    teen
4   13  (0,13]     kid
5    4  (0,13]     kid
6   35 (20,60]     man
7   46 (20,60]     man
8   21 (20,60]     man
9   98 (60,99]   elder
10  54 (20,60]     man
11  76 (60,99]   elder
12  43 (20,60]     man
```

L'`as.numeric` ci restituisce la classe di appartenenza.

```
> cleta$ncleta1 <- as.numeric(cleta$cleta1b)
> cleta
   eta  cleta1 cleta1b ncleta1
1    2  (0,13]     kid       1
2   29 (20,60]     man       3
3   18 (13,20]    teen       2
4   13  (0,13]     kid       1
5    4  (0,13]     kid       1
6   35 (20,60]     man       3
7   46 (20,60]     man       3
8   21 (20,60]     man       3
9   98 (60,99]   elder       4
10  54 (20,60]     man       3
11  76 (60,99]   elder       4
12  43 (20,60]     man       3
```

Un esempio con **break inferiore** incluso nell'intervallo

```
> cleta$cleta2 <- cut(eta,breaks,right=F)
> cleta$ncleta2 <- as.numeric(cleta$cleta2)
> cleta$areequal <- cleta$ncleta1==cleta$ncleta2
> cleta
   eta  cleta1 cleta1b ncleta1  cleta2 ncleta2 areequal
1    2  (0,13]     kid       1  [0,13)       1     TRUE
2   29 (20,60]     man       3 [20,60)       3     TRUE
3   18 (13,20]    teen       2 [13,20)       2     TRUE
4   13  (0,13]     kid       1 [13,20)       2    FALSE
5    4  (0,13]     kid       1  [0,13)       1     TRUE
6   35 (20,60]     man       3 [20,60)       3     TRUE
7   46 (20,60]     man       3 [20,60)       3     TRUE
8   21 (20,60]     man       3 [20,60)       3     TRUE
9   98 (60,99]   elder       4 [60,99)       4     TRUE
```

```
10  54 (20,60]     man       3 [20,60)       3     TRUE
11  76 (60,99]   elder       4 [60,99)       4     TRUE
12  43 (20,60]     man       3 [20,60)       3     TRUE
```

12.8 Stringhe

12.8.1 Manipolazione di stringhe e vettori di caratteri

Vi sono numerose funzioni che possono tornare utili per il trattamento delle stringhe: `paste` coercizza a vettore di caratteri ciò che gli viene dato in argomento, dopodichè **unisce** elemento per elemento (eventualmente riciclando il vettore più corto, per quanto possibile

```
> paste(c("a","b","c"),c("x","y","z"))
[1] "a x" "b y" "c z"

> paste(c("a","b","c","d"), 1:2)
[1] "a 1" "b 2" "c 1" "d 2"

> paste(c("a","b","c","d"), 1:3)
[1] "a 1" "b 2" "c 3" "d 1"
```

Di default i caratteri pastati vengono distanziati da uno spazio; questo è evitabile specificando l'opzione `sep` (che più generalmente specifica il separatore tra i caratteri pastati.

```
> paste(c("a","b"),c("x","y"), sep = "")
[1] "ax" "by"
> paste(c("a","b"),c("x","y"), sep = "+")
[1] "a+x" "b+y"
```

L'inserzione dell'opzione `collapse` fa si che invece di un vettore di stringhe venga tutto concatenato in una unica stringa, con il separatore specificato da collapse, appunto:

```
> paste(c("a","b"), c("x","y"), sep = "", collapse=" + ")
[1] "ax + by"
```

Per **estrarre** uno o più frammenti di stringa si può utilizzare `substring`, specificare il carattere di partenza e quello di arrivo

```
> substring(c("mamma","papa") , 1 , 2)
[1] "ma" "pa"
```

`nchar` serve per misurare la lunghezza delle stringhe contenute in vettori di caratteri

```
> nchar("supercalifragilistichespiralidoso")
[1] 33
> nchar(c("asd","qwerty"))
[1] 3 6
```

Per **abbreviare** stringhe lunghe si può utilizzare `abbreviate`

```
> data(women)
> names(women)
[1] "height" "weight"
> names(women) <- abbreviate(names(women))
> names(women)
[1] "hght" "wght"
```

Per **cambiare da minuscole a maiuscole** utilizzare `tolower` e `toupper`; per un controllo più fine `chartr`

```
> tolower("CIAO")
[1] "ciao"
> chartr("a","A","ciao")
[1] "ciAo"
```

12.8.2 Esecuzione di comandi contenuti in stringhe

Una stringa può esser eseguita come comando, facendone il parsing e poi valutandola:

```
eval(parse(text = "mean(asd)"))
```

12.8.3 Espressioni regolari ed elaborazione del testo

Le funzioni principali di R per maneggiare espressioni regolari sono:

- `grep`, `grepl`: cercano il match di una espressione regolare/pattern in un vettore di caratteri; ritornano gli indici degli elementi che matchano (`grep`) o un vettore logico che indica i valori che matchano (`grepl`)
- `regexpr`, `gregexpr`: cercano il match e restituiscono gli indici, all'interno di ogni *stringa*, dove il match inizia e la lunghezza dei caratteri matchati. Molto utile è l'utilizzo combinato con la funzione `regmatches`.
- `sub`, `gsub`: ricercano in un vettore una espressione regolare e rimpiazzano il match con un'altra stringa
- `regexec`: più facile da mostrare che da spiegare. Anche con questa è utile `regmatches`.

Il dataset utilizzato in seguito Si tratta di un file html che include informazioni su omicidi avvenuti a baltimora. La prima linea è

```
39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class="address">3400 Clifton Ave.<br />Baltimore,
MD 21216</dd><dd>black male, 17 years old</dd><dd>Found on
January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'
```

Abbiamo innanzitutto latitudine e longitudine (`39.311024, -76.674227`), si ha anche un indirizzo dove è avvenuto l'omicidio (`3400 Clifton Ave.`), la causa (`shooting`), la data di rinvenimento del corpo (`Found on January 1, 2007`) e informazioni sulla persona (come nome, età e razza).

```
h <- readLines("dataset/homicides.txt")
length(h)

## [1] 1250
```

12.8.3.1 `grep` e `grepl`

`grep` **ricerca** il pattern in un vettore di stringhe e restituisce gli indici dei valori che matchano:

```
grep("^New", state.name)

## [1] 29 30 31 32
```

se si setta `value=TRUE`, restituisce gli elementi che matchano

```
grep("^New", state.name, value=TRUE)

## [1] "New Hampshire" "New Jersey"    "New Mexico"    "New York"
```

`grepl` è una variazione che restituisce invece un vettore di logici che indicano quali elementi matchano

```
grepl("^New", state.name)

##  [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [17] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
## [33] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [49] FALSE FALSE
```

Per applicarlo all'esempio degli omicidi possiamo voler contare o estrarre quanti sono stati uccisi mediante sparo; possiamo cercare due tipi di informazioni, `iconHomicideShooting` e `Cause`:

```
length(grep("iconHomicideShooting", h))

## [1] 228

length(grep("iconHomicideShooting|icon_homicide_shooting", h))

## [1] 1003

length(grep("Cause: shooting", h))

## [1] 228

## Shooting a volte maiuscolo e avolte minuscolo, quindi uso una
## caracter class
length(grep("Cause: [Ss]hooting", h))

## [1] 1003
```

Il testo non è formattato strettamente (ad esempio a volte si scrive `iconHomicideShooting`, altre `icon_homicide_shooting`.

12.8.4 `regexpr` e `gregexpr`

Grep non ci dice esattamente dove il match avviene all'interno della stringa (utile soprattutto su espressioni regolari più complicate delle literals semplici). Per questo si usa `regexpr` (se non trova restituisce `-1`), la quale fornisce anche la lunghezza del match.
`gregexpr` termina la propria ricerca al primo match all'interno della stringa (leggendo da sinistra a destra). `gregexpr` fornisce invece tutti i match in una stringa.
Se ad esempio volessimo usare `regexpr` sull'estrazione della data usiamo il pattern

```
<dd>Found(.*?)</dd>
```

Togliamo la greediness di * mediante ?, per far si che si fermi a prima del primo `</dd>` incontrato dopo `<dd>`

```
regexpr("<dd>Found(.*?)</dd>", h[1:10])

##  [1] 177 178 188 189 178 182 178 187 182 183
## attr(,"match.length")
##  [1] 33 33 33 33 33 33 33 33 33 33
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Il primo vettore restituito è il punto dove inizia il match (es al 177-esimo carattere del primo elemento), il secondo è la lunghezza. Se vogliamo vedere il match nel primo elemento possiamo comandare

```
substr(h[1], 177, 177+33-1)

## [1] "<dd>Found on January 1, 2007</dd>"
```

Una funzione utile è `regmatches` ce estrae i match in una stringa senza la necessità di utilizzare `substr`

```
r <- regexpr("<dd>Found(.*?)</dd>", h[1:5])
regmatches( h[1:5], r)

## [1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>"
## [3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>"
## [5] "<dd>Found on January 5, 2007</dd>"
```

12.8.5 sub e gsub

`sub` e `gsub` **rimpiazzano** il match effettuato da un'espressione regolare

```
> sub("g","G","aggeggio")
[1] "aGgeggio"
> gsub("g","G","aggeggio")
[1] "aGGeGGio"
```

Per l'applicazione al caso degli omicidi, abbiamo ancora una versione grezza della data e vorremmo togliere i margini html `dd`

```
r <- regexpr("<dd>Found(.*?)</dd>", h[1:5])
m <- regmatches(h[1:5], r)
```

Vogliamo togliere sia il pre, fino ad on, che il post

```
sub("<dd>[F|f]ound on |</dd>", "", m)

## [1] "January 1, 2007</dd>" "January 2, 2007</dd>" "January 2, 2007</dd>" "January
## [5] "January 5, 2007</dd>"

## ma sub si ferma al primo match, usiamo gsub
(d <- gsub("<dd>[F|f]ound on |</dd>", "", m))

## [1] "January 1, 2007" "January 2, 2007" "January 2, 2007" "January 3, 2007" "Janua

## e quindi finalmente possiamo trasformarlo a data con as.Date,
## prima di aver momentaneamnete cambiato il locale ad americano
## (dato che i mesi son scritti in inglese)
Sys.setlocale("LC_TIME", "en_US.UTF-8")

## Warning in Sys.setlocale("LC_TIME", "en_US.UTF-8"): Il sistema
operativo indica che la richiesta di cambiamento localizzazione a "en_US.UTF-8"
non può essere onorata

## [1] ""

Sys.getlocale("LC_TIME")

## [1] "it_IT.UTF-8"

as.Date(d, "%B %d, %Y")

## [1] NA NA NA NA NA
```

12.8.6 regexec

La funzione `regexec` funziona come `regexpr` ad eccezione che fornisce gli indici per le subexpression entro parentesi

```
regexec("<dd>[F|f]ound on (.*?)</dd>", h[1] )

## [[1]]
## [1] 177 190
## attr(,"match.length")
## [1] 33 15
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Restituisce un elemento in cui il primo elemento 177 è dove inizia il match beccato e 33 è la lunghezza del match. Il secondo elemento dei vettori è l'inizio della parte tra parentesi nel match, mentre 15 è la lunghezza del match tra parentesi.
Nei casi precedenti potevamo anche non usare la parentesi; regexec sarebbe cambiata così

```
regexec("<dd>[F|f]ound on .*?</dd>", h[1] )

## [[1]]
## [1] 177
## attr(,"match.length")
## [1] 33
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Possiamo sfruttare questa cosa con **substr** o con **regmatches**. Seguiamo questo secondo approccio

```
r <- regexec("<dd>[F|f]ound on (.*?)</dd>", h[1:2])
regmatches( h[1:2], r)

## [[1]]
## [1] "<dd>Found on January 1, 2007</dd>" "January 1, 2007"
##
## [[2]]
## [1] "<dd>Found on January 2, 2007</dd>" "January 2, 2007"
```

ogni elemento della lista ritornata è un vettore composto prima dal match completo e poi dal match entro parentesi. Possiamo in seguito usare lapply o sapply per estrarre gli elementi

```
regm <- regmatches( h[1:2], r)
dates <- sapply(regm, function(x) x[2])
dates <- as.Date(dates, "%B %d, %Y")
dates

## [1] NA NA
```

12.8.7 Wildcards

Per le **wildcards** si utilizza `glob2rx` (che trasforma in espressione regolare, da porre in grep, il pattern specificato)

```
> grep(glob2rx("p*"), emilia)
[1] 1 2
```

12.8.8 Matching fuzzy

Per il **matching fuzzy** si utilizza `agrep`

```
> grep("res",emilia)
integer(0)
> agrep("res",emilia)
[1] 3
```

12.9 Date e tempi (*timestamp*)

R ha un modo peculiare per rappresentare date e tempi

- le date sono rappresentate dalla classe `Date`. Le date sono memorizzate internamente come il numero di giorni a partire dal 1/1/1970.
- i tempi, o *timestamp*, sono rappresentati mediante `POSIXct` e `POSIXlt`.
  Nella classe `POSIXct`, i dati sono rappresentati come un vettore di secondi a partire dal dal 1/1/1970; utili ad esempio per memorizzare delle date all'interno di `data.frame`.
  Nella classe `POSIXlt`, i dati sono rappresentati come una *lista*, che memorizza tra l'altro anche informazioni aggiuntive utili (come giorno della settimana, giorno dell'anno, mese, giorno del mese).

In generale un tempo può esser trasformato in data e viceversa mediante le funzioni della famiglia `as.`.
Nel passaggio da una data ad un *timestamp*, ovviamente si perderanno le informazioni dall'ora in giu.
Se si necessita solo di una data è meglio utilizzare `Date`.
Alcune dimostrazioni delle proprietà delle date

```
## DATE
## Lo 0 è il Primo gennaio del 70, nella classe Date
unclass(as.Date("1970-01-01"))

## [1] 0

## e i giorni sono memorizzati come intero di distanza in giorni...
unclass(as.Date("1970-01-02"))

## [1] 1

Sys.Date()
```

```
## [1] "2023-05-18"

as.numeric(Sys.Date())

## [1] 19495

## POSIXct
(x <- Sys.time())

## [1] "2023-05-18 10:59:47 CEST"

class(x)

## [1] "POSIXct" "POSIXt"

unclass(x) # numero di secondi

## [1] 1684400387

## POSIXlt
p <- as.POSIXlt(x)
names(unclass(p)) # lista

##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"   "isdst"  "zone"
## [11] "gmtoff"

p$sec

## [1] 47.21243
```

12.9.1 Funzioni utili

Alcune funzioni che possono tornare utili applicabili a date così come a tempi sono:

- `weekdays` fornisce il nome del giorno della settimana
- `month` il nome de giorno del mese
- `quarters`: fornisce il trimestre

```
weekdays(Sys.Date())

## [1] "giovedì"

months(Sys.time())

## [1] "maggio"

quarters(Sys.Date())

## [1] "Q2"
```

12.9.2 Creare date

Per far sì che R capisca che un vettore è una data dobbiamo comportarci differentemente a seconda che le informazioni di partenza siano:

- stringhe
- un vettore di giorni/secondi a partire da un dato tempo
- vettori differenti contenenti rispettivamente anni, mesi, giorni ecc

12.9.2.1 Date/Tempi a partire da stringhe

Nel primo caso utilizziamo

- `as.Date` se desideriamo avere una data
- `strptime` se vogliamo un timestamp

Con `as.Date`, a meno che il formato della stringa rispetti lo standard ISO delle date (ovvero sia in forma `YYYY-MM-DD`) è necessario specificare mediante `format` il formato della stringa:

```
## Formato standard, non c'e` bisogno di specificare format
c <- "1983-11-04"
(d <- as.Date(c))

## [1] "1983-11-04"

class(d)

## [1] "Date"

## Formato classico italiano
 a <- "04/11/1983"
(b <- as.Date(a, format="%d/%m/%Y"))

## [1] "1983-11-04"

class(b)

## [1] "Date"
```

Le opzioni più utili come formato sono riportate in tabella 12.1.

`strptime` estrae da una stringa attraverso `format` e restituisce un *timestamp*.

```
datestring <- c("Gennaio 10, 2012 10:40", "Dicembre 9, 2011 09:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
```

Qualora la stringa si codificata in un altro locale, prima di usare `as.Date` cambiare momentaneamente il locale

Formato	Significato
%Y	Anno a quattro cifre
%m	Mese (01-12)
%d	Giorno (01-31)
%H	Ora (00-23)
%M	Minuto (00-59)

Tabella 12.1: Opzioni Format per date e tempi

```
dat <- "January 10, 2012 10:40"
Sys.setlocale("LC_TIME", "en_US.UTF-8")

## Warning in Sys.setlocale("LC_TIME", "en_US.UTF-8"): Il sistema
operativo indica che la richiesta di cambiamento localizzazione a "en_US.UTF-8"
non può essere onorata

## [1] ""

as.Date(dat, "%B %d, %Y")

## [1] NA

Sys.setlocale("LC_TIME", "it_IT.UTF-8")

## [1] "it_IT.UTF-8"
```

12.9.2.2 Data/Tempo da un vettore di giorni/secondi

Nel caso in cui una data/tempo sia memorizzata come numero di giorni/secondi a partire da una data epoca, per ottenere tale data/tempo occorre utilizzare `as.Date` e `as.POSIXct`, specificando per quest'ultimo il giorno di inizio del conteggio dei secondi.
Un esempio di secondo giorno a partire dal primo gennaio dell'1983

```
d <- as.Date(2, origin="1983-01-01")
d

## [1] "1983-01-03"
```

Per un esempio di `as.POSIXct`, effettuiamo una importazione di date salvate da SPSS (che vengono memorizzate, come secondi a partire dal 15 ottobre 1582:

```
timestamp.R <- as.POSIXct(data.spss, origin="1582-10-15")
```

12.9.3 Date/Tempi a partire da molteplici vettori

Si può partire dall'utilizzo di `ISOdatetime` (per generare un timestamp) o `ISOdate` (sempre un timestamp) ed eventualmente passare ad `as.Date` in seguito se ci serve una data:

```
(now <- ISOdatetime(2011,1,23,12,23,00)) #crea un timestamp POSIXct
as.Date(now)                             #coercizione a Date
```

12.9.4 Estrarre informazioni dalle date

Per *estrarre informazioni* è in generale meglio convertire le date nella classe `POSIXlt` (una lista, con nomi, delle singole componenti, es anno):

```
> (adesso <- Sys.time())
[1] "2011-01-23 11:47:54 CET"
> lt.adesso <- as.POSIXlt(adesso)
> lt.adesso$
lt.adesso$sec    lt.adesso$mday   lt.adesso$wday
lt.adesso$min    lt.adesso$mon    lt.adesso$yday
lt.adesso$hour   lt.adesso$year   lt.adesso$isdst
```

Da cui, ad esempio

```
> lt.adesso$sec
[1] 54.5034
> lt.adesso$min
[1] 47
> lt.adesso$hour
[1] 11
> lt.adesso$mday
[1] 23
```

Si presti attenzione però al fatto che, per ottenere info corrette:

```
> lt.adesso$mon+1
[1] 1
> lt.adesso$year+1900
[1] 2011
```

Perchè lo standard posix specifica che il numerico dei mesi vada da 0 a 11, e gli anni partano a 0 da 1900.

12.9.5 Elaborazioni su date

Per il **confronto di date** si possono utilizzare i normali operatori, ad esempio:

```
data1 > data2
```

restituisce un logical, per tutte

Per la differenza di date si utilizzi `difftime` per la classe `POSIXct` e l'operatore di sottrazione per `Date`

```
difftime(data1, data2, unit="day")    # POSIXct
data2 - data 1                        # Date
```

12.10 Eliminazione record doppi

In un dataframe (o in matrice o in vettore) si utilizzi `unique` per eliminare i record duplicati:

```
> asd
  V1 V2
1  4  9
2  2  7
3  5 10
4  4  9
5  5 10
6  2  1
> unique(asd)
  V1 V2
1  4  9
2  2  7
3  5 10
6  2  1
```

Altrimenti si può utilizzare `asd[!duplicated(asd), ]`

12.11 Reshape

Consideriamo il dataset con id personale id e numero progressive evento `num_prog`

```
> foo
  id num_prog a b
1  1        1 3 7
2  1        2 1 5
3  2        1 2 6
```

Esso è in forma *long*, ovvero ad un'unità statistica possono corrispondere più record (il che appunto allunga, verso il basso il dataset).
La funzione `reshape` permette di porlo in maniera *wide* (o longitudinale): ovvero un record per ogni unità statistica

```
> bar <- reshape(foo, idvar="id", timevar="num_prog", direction="wide")
> bar
  id a.1 b.1 a.2 b.2
1  1   3   7   1   5
3  2   2   6  NA  NA
```

Gli `NA` si generano dove non avevamo opportune dati. Per riportarlo alla forma precedente si comandi:

```
> foo2 <- reshape(bar, idvar="id", direction="long")
> foo2
    id num_prog a.1 b.1
1.1  1        1   3   7
2.1  2        1   2   6
1.2  1        2   1   5
2.2  2        2  NA  NA
```

Vediamo però che l'algoritmo, non riuscendo a discernere gli `NA` generati da un reshape dai dati normali procede tranquillamente. Vogliamo eliminare l'ultima riga. Guardando alle rownames dell'ultimo data frame notiamo che sono semplicemente nella forma "`id.numero_prog`" quindi

```
> paste(foo$id,foo$num_prog, sep=".")
[1] "1.1" "1.2" "2.1"
```

Utilizzando gli indici e il nome delle righe possiamo arrivare al dataset di partenza mediante

```
> foo2<-foo2[paste(foo$id,foo$num_prog, sep="."),]
> foo2
    id num_prog a.1 b.1
1.1  1        1   3   7
1.2  1        2   1   5
2.1  2        1   2   6
```

Capitolo 13

Grafici

13.1 Introduzione

13.1.1 Il sistema grafico di R

Al cuore del sistema grafico di R sta il pacchetto `grDevices`, che può essere appellato come il motore grafico (*graphics engine*). Questo fornisce l'infrastruttura fondamentale per i grafici in R, come selezione dei colori, dei font e i *devices* per la redirezione dell'output grafico su diversi supporti (monitor, file ecc)[1].
Due pacchetti si fondano direttamente sul motore grafico e forniscono due sistemi di plotting *largamente incompatibili fra loro* all'interno di R, dividendo di fatto la funzionalità grafica del software in due *mondi separati*:

- il pacchetto `graphics`: è il pacchetto grafico tradizionale di R, e fornisce funzioni per la produzione di grafico completo[2], cosi come funzione per l'aggiunta di output grafico a plot già esistenti;
- il pacchetto `grid`: è il sistema grafico di ultima generazione nell'ambito di R. Fornisce un set di tool grafici di base, *non* fornisce funzioni per il disegno di plot completi ed è raramente utilizzato in via diretta.

Molti altri pacchetti grafici si basano o su `graphics` o su `grid`: si fondano su quest'ultimo due pacchetti popolari ed alternativi a `graphics` per la produzione di grafici completi, `ggplot2` e `lattice`. Vi sono poi pacchetti specifici per la produzione di grafici settoriali/specializzati (es `maps`).
Vi sono poi pacchetti che forniscono una interfaccia tra R e sistemi grafici di terze parti come `rgl` (immagini 3d sofisticate), `rggobi` ed `iplots` (sistemi di visualizzazione grafica dinamica e interattiva).

13.1.2 Funzioni grafiche e procedura di creazione dell'immagine

Le funzioni grafiche possono esser classificate in.

[1] Oltre al motore grafico di base fornito da `grDevices`, su CRAN vi sono diversi pacchetti che forniscono devices aggiuntivi ove redirigere l'output grafico; si citano tra gli altri `tikzDevice` che permette la creazione di codice tikz per l'inserimento all'interno di documenti LaTeX

[2] Ad esempio `graphics` fornisce `plot` e `hist`

- *funzioni che producono output grafico*
  - funzioni di **alto livello** per la produzione di grafici completi
  - funzioni di **basso livello**, che aggiungono ulteriore output ad un grafico già creato e/o lo modificano
- funzioni funzioni che permettono di *gestire l'output grafico* (es indirizzarlo verso un devices ecc)

Tipicamente la *procedura di creazione di un grafico* si alterna

1. gestendo innanzitutto dove fare pervenire l'output grafico (su quale device? schermo o file? se file quale formato?)
2. passando poi all'utilizzo di funzioni di alto livello per la produzione del grafico;
3. e infine modificando l'output prodotto con funzioni di basso livello.

13.1.3 Una guida alla scelta tra `graphics` e `grid`

L'esistenza di due sistemi grafici distinti e largamente incompatibili come `grid` e `graphics` pone il dubbio su quale utilizzare:

- se è necessario produrre un plot nuovo, generalmente entrambi i sistemi forniscono alternative altrettanto utilizzabili (`grid` via `lattice` e `ggplot2`), quindi la scelta dipende se l'implementazione di uno o dell'altro ci soddisfa. Va detto che nella progettazione i sistemi `lattice` e `ggplot2` sono stati ottimizzati sulla percezione grafica umana. Inoltre forniscono tool per la visualizzazione di dati multidimensionali più efficaci. L'unico contro è che hanno una curva di apprendimento più ripida rispetto al sistema tradizionale.
- se è necessario modificare un plot dopo che lo si è creato (es aggiungere ulteriore output grafico), la cosa da sapere è quale sistema grafico ha creato il grafico base; in generale lo stesso sistema grafico dovrebbe essere utilizzato (o meglio le funzioni di basso livello che esso fornisce[3])
- se è necessario creare una funzione che produca un output grafico per il quale nessuno ha scritto ancora niente, allora implementarla usando `grid`
- i sistemi basati su `grid` sono lievemente più lenti del sistema graphics tradizionale (anche se nella maggior parte dei casi tale differenza di elaborazione è irrilevante)

13.2 I devices grafici

La prima scelta da effettuare quando si sta per creare un grafico è scegliere su quale device redirigere l'output dei comandi grafici.
Attivare un device vuol dire comunicare ad R di utilizzarlo come destinazione cui indirizzare l'output grafico; se *disattiviamo* il device, R non invia più dati allo stesso.
I principali device su cui R può scrivere sono (tra gli altri):

[3] Il pacchetto `gridBase` fornisce un modo di raggirare questa limitazione

- schermo: l'attivazione è automatica alla chiamata di una funzione di alto livello, se non si è specificato un device alternativo
- file `pdf`: grafica vettoriale cross-platform
- file `emf`: grafica vettoriale in sistema windows
- file `png`: formato bitmap compresso buono per linee (anti-aliasing) o immagini a colori, adotta una compressione senza perdita di qualità
- `jpeg`: grafica bitmap cross platform, ottimo per il plotting di molti dati, perde molto con il ridimensionamento, pessimo per il plotting di linee (no anti-aliasing)
-

I formati bitmap rispetto a quelli vettoriali possono avere un'applicazione utile quando si plottano molti punti perchè la compressione fa si che le dimensioni del file rimangano contenute. Tendono

13.2.1 Attivazione e chiusura di un device

L'attivazione di un device varia a seconda della sua tipologia: il *device "schermo"* viene attivato di default ad una chiamata di funzione grafica di alto livello[4]. *Device file* possono esser attivati mediante la funzione con nome (spesso) coincidente con l'estensione del file. Ad esempio:

```
> pdf("file.pdf")
> png("file.ps")
```

Una volta aperto, la scrittura su device avverrà semplicemente passando "comandi grafici" ad R (ad esempio `plot(rnorm(10))`). La **chiusura di un device** si ottiene mediante:

```
> dev.off()
```

13.2.2 Gestione di più device

Come si possono gestire più device contemporaneamente? Ad esempio abbiamo attivato il device schermo e abbiamo effettuato alcune prove; ora però vogliamo salvare un grafico ben riuscito, e magari poter tornare al device schermo poi per effettuare ulteriori prove.
Per farlo possiamo pertanto il secondo device, senza chiudere il primo. Ad esempio mediante `postscript("file.ps")`: ora abbiamo attivo questo device e tutto l'output grafico sarà inviato al file postscript. Quindi possiamo inviare il grafico "fatto bene".
Se vogliamo tornare al vecchio device schermo per fare prove, e più in generale **muoverci tra un device e l'altro**, dobbiamo comandare:

```
> dev.set(id_device)
```

[4]Per l'apertura esplicita bisogna utilizzare le funzioni `x11` sotto linux, `windows` in windows e `quartz` in mac

dove `id_device` è il **numero di identificazione** del device il quale **può esser ottenuto mediante**:

```
> dev.list()
```

che ci fornisce tipo e id di ogni device attivo; ad esempio, con due file PNG, un PDF e un device schermo `dev.list()` mi da:

```
> dev.list()
X11 pdf PNG PNG
  2   3   4   5
```

Il device 1 (null device) è riservato al sistema e non può esser utilizzato. Per **sapere quale device è attivo** si può comandare:

```
> dev.cur()
PNG
  5
```

Se vogliamo pertanto riaprire il device-schermo, pertanto:

```
> dev.set(2)
```

Per la chiusura di device, `dev.off()` accetta come argomento l'id del device; se non specificato dev.off chiuderà il device corrente, dopodichè attiverà il device successivo nella lista.
Altri comandi che possono tornare utili sono:

```
dev.prev()
dev.next()
```

che restituiscono il **numero di device precedente o successivo** a quello dato in argomento (che di default è quello attivo attualmente). Il comando per **chiudere tutti i device grafici attivi** (sia file che schermo) è

```
> graphics.off()
```

13.2.3 Riutilizzo del contenuto di un device grafico

Copiare un plot da un device ad un altro può essere utile poichè alcuni plot richiedono molto codice e può esser comodo evitare di dover copiare incollare (per mantenere) il codice nella sezione di un altro device. Abbiamo due strategie possibili: quella mediante `recordPlot` e quella mediante `dev.copy`.

`recordPlot()`, memorizza l'output presente sul device attivo in una variabile, per poterlo riutilizzare in seguito.
Alcune applicazioni sono:

- salvare uno stesso grafico in più formati
- salvare un grafico in seguito ad esplorazioni a linea di comando in un file

Dopo aver creato il grafico di interesse, semplicemente:

```
> gr1 <-recordPlot()
```

Una volta salvati i grafici che ci servono, aprire il device-file e salvare chiamando la variabile mediante, alternativamente

```
> gr1
> replayPlot(gr1)
```

Alternativamente si può ricorrere, tenendo i due device aperti contemporaneamente

- `dev.copy` *copia il contenuto del corrente device* in uno specificato mediante l'opzione `which` dopodichè si sposta in quest'ultimo device.
- `dev.copy2pdf` salva il contenuto del corrente device in un file pdf

Bisogna comunque specificare che **copiare un plot non è un'operazione esatta** (ovvero quello che vi è sullo schermo può non essere perfettamente uguale a quello che viene memorizzato nel pdf, poichè alcune approssimazioni avvengono per assecondare la differenza di basso livello insite nei diversi devices).

13.3 L'uso del pacchetto `graphics`

Il sistema grafico tradizionale è utile soprattutto per i grafici 2D. Nel sistema grafico tradizionale un'immagine è creata chiamndo innanzitutto una funzione di alto livello che crea un plot completo, e in seguito chiamando funzioni di basso livello che aggiungono ulteriori elementi, se necessari.
`plot` è la funzione di alto livello più importante. Si tratta di una funzione *generica*, ovvero fornendogli dati in input di diversa natura produrrà differenti tipologie di immagini.
Si può passargli due vettori `x` e `y` e in tal caso produce uno scatterplot[5] oppure sotto `x` una struttura di dati più complessa ed essa invocherà una funzione ad hoc per il plotting dell'oggetto.

```
methods(plot)[1:20]
```

```
##  [1] "plot,ANY-method"     "plot,polygon-method" "plot.aareg"          "plot.acf"
##  [5] "plot.boot"           "plot.cox.zph"        "plot.data.frame"     "plot.decomposed.ts"
##  [9] "plot.default"        "plot.dendrogram"     "plot.density"        "plot.ecdf"
## [13] "plot.factor"         "plot.formula"        "plot.function"       "plot.hclust"
## [17] "plot.histogram"      "plot.HoltWinters"    "plot.isoreg"         "plot.lm"
```

Ad esempio per conoscere di più per la tipologia di plot effettuato se gli si passa in input un data.frame, si può consultare `?plot.data.frame`.
In molti casi pacchetti disponibili definiscono nuovi metodi di plotting degli oggetti di cui si occupano, definendo una funzione `plot.tipoOggetto`.
Sempre nel pacchetto `graphics`, poi, esistono altre funzioni per plot più specifici come `barplot`, `hist` e `boxplot` (grafici omonimi).
Funzioni high-level per il plotting di

- una singola variabile: Murrell, R Graphics, pag 33

[5] La funzione chiamerà il metodo `plot.default`.

- due variabili: Murrell, R Graphics, pag 35
- più di due variabili: Murrell, R Graphics, pag 37

13.3.1 Opzioni

Spesso la rappresentazione di default della funzione di alto livello non è soddisfacente, pertanto occorre effettuare personalizzazioni del grafico, impostando alcune opzioni.
In R le opzioni grafiche possono essere:

- settate globalmente per tutti i grafici
- impostate per uno specifico grafico (se la sua funzione accetta tali parametri come argomento della chiamata)

Parametri globali L'interrogazione/setting delle opzioni grafiche globali/di default avviene attraverso la funzione `par`.
R tiene in memoria un set di parametri per ogni device grafico attivo.
Una semplice invocazione restituisce tutti i parametri attuali; se si desiderano parametri in particolare occorre specificarli in un vettore di caratteri;

```
## vediamo quali parametri sono impostati per col e lty
par(c("col", "lty"))
## imposto alcuni parametri ad un nuovo valore
par(col="red", lty="dashed")
```

Una modifica di parametri grafici via `par` ha inizio a partire dal grafico successivo e fino a che non viene cambiata nuovamente l'impostazione; le opzioni globali non vengono considerate se si specificano opzioni ad-hoc nel contesto di una chiamata di funzione grafica.
Una buona abitudine può essere in alcuni salvare i parametri di default prima di effettuare le modifiche, al fine di poterli reimpostare una volta cha abbiamo terminato i grafici con impostazioni particolari.
La sintassi per farlo

```
old.graph.parameters <- par(no.readonly = TRUE) #backup
...
chiamate a par globali
...
par(old.graph.parameters) # <- torno alla situazione precedente
```

Parametri locali ad un grafico La maggior parte dei parametri di `par` possono essere anche impartiti come argomenti opzionali di una specifica chiamata di funzione grafica (che lo preveda). In tal caso varranno solamente per il grafico in questione, non per i successivi.

Elenco parametri I parametri grafici possono esser classificati in

- parametri impostabili sia in funzione che via par

Parametro	Significato
cex	dimensione font (moltiplicatore rispetto a default)
cex.axis	dimensione font *tick assi* (moltiplicatore rispetto a cex)
cex.lab	dimensione font *etichetta assi* (moltiplicatore rispetto a cex)
cex.main	dimensione font *titolo* (moltiplicatore rispetto a cex)
cex.sub	dimensione font *sottotitolo* (moltiplicatore rispetto a cex)
bg	colore dello sfondo
fg	colore del *foreground*
col	colore delle linee/dei simboli
col.axis	colore tick assi
col.lab	colore legenda assi
col.main	colore titolo
col.sub	colore sottotitolo
adj	come il testo viene giustificato
las	rotazione del testo negli assi/margini
font	grassetto/corsivo testo
font.axis	grassetto/corsivo testo tick assi
font.lab	grassetto/corsivo testo etichetta assi
font.main	grassetto/corsivo testo titolo
fond.sub	grassetto/corsivo testo sottotitolo
type	tipo di plot (linea, punti, entrambi)
pch	simbolino dei plot (pag 69 Murrell)
lty	tipo linea (continua, tratteggiata) (pag 69 Murrell)
lwd	spessore linea

Tabella 13.1: Opzioni settabili mediante par e come argomenti della funzioni grafiche.

- parametri impostabili solo mediante par, a livello globale
- parametri read-only (non possono esser modificati)

In tabella 13.1 sono elencati i parametri impostabili mediante par o accettati dalla maggior parte delle funzioni di alto livello.
Altri parametri (es mfrow) possono esser specificati solamente attraverso par. Un elenco è presentato in tabella 13.2.
Infine, alcuni parametri sono tipicamente specificati ad ogni chiamata funzione di alto livello, poichè si riferiscono ad un singolo grafico. Le opzioni si impostano con la sintassi

```
parametro=valore
```

Parametro	Significato
mfrow	numero di plot per riga, colonna (plot riempiti per riga)
mfcol	numero di plot per riga, colonna (plot riempiti per colonna)
mar	ampiezza dei margini
oma	ampiezza margini esterni

Tabella 13.2: Opzioni (alcune) settabili solamente mediante par. Vedi Murrell, pag 56

I parametri più comuni sono:

`main` accetta come valore una stringa che specifica il titolo del grafico

`sub` stringa, sottotitolo del grafico

`xlab-ylab` accettano una stringa come valore, impostano l'etichetta dell'asse delle x e delle y

`xlim-ylim` accettano una vettore di due valori numerici, specificando il range di plotting (rispettivamente per x e y)

13.3.2 Funzioni di basso livello per la modifica del grafico

Altre funzioni utili per l'aggiunta di output ad un grafico già esistente:

- `lines` aggiunge linee ad un plot, forniti i valori in coordinate x-y di punti che la funzione connette
- `points` aggiunge punti ad un plot
- `text` stampa la stringa alle coordinate specificate
- `title` aggiunge un titolo all'asse x, y, titolo, sottotitolo o margine esterno
- `mtext` aggiunge testo ai margini interni o esterni del plot
- `axis` aggiunge ticks e labels all'asse
- `segments` per disegnare un segmento da $(6, 1)$ a $(2, 8)$, `segments(6,1,2,8)`;
- `arrows` per disegnare un vettore da $(0, 0)$ a $(1, 1)$, `arrows(0,0,1,1)`;
- `rect` serve per disegnare un rettangolo; un esempio di rettangolo da (0,1) a (2,3) è `rect(0,1,2,3)`

13.3.3 Altre funzioni utili

- `grid` aggiunge la griglia al diagramma stampato
- `curve` per la rappresentazione di funzioni matematiche; ad esempio può esser utilizzato per aggiungere una distribuzione ad un istogramma.
- `locator` identifica le coordinate di dove si clicca col mouse (utile ad esempio per piazzare del testo in un punto specifico). Cliccare col sinistro sui punti che si vogliono identificare (possono essere anche più di uno); dopodichè cliccare col destro per terminare e ottenere i valori stampati a video;
- `abline` aggiunge una linea al grafico, date pendenza ed intercetta. Si possono anche stampare linee verticali specificando come parametro `v=3` (equivale all'equazione $x = 3$), linee orizzontali mediante `h=2` (che equivale a $y = 2$. Come applicazione prende in input un modello lm bivariato e ne stampa la retta corrispondente.

13.3.4 Come ottenere grafici multipli

Per impostare grafici multipli su una medesima figura si possono seguire due strade:

- `mfrow` e `mfcol` parametri settati mediante `par`;
- mediante la funzione di più alto livello `layout`

la seconda, a differenza dei primi permette di creare regioni multiple di dimensioni differenti. Il contenuto della funzione è una matrice che specifica la struttura della pagina, il contenuto della matrice l'ordine di plot. Il layout può poi essere ispezionato mediante un'opportuna funzione

```
> a <- layout(matrix(1:6,byrow=T, ncol=2))
> layout.show(a)
```

Per creare un pdf con layout

```
pdf("foo.pdf")
layout(matrix(1:4,byrow=T, ncol=2))
plot(...)
...
dev.off()
```

Per vedere possibilità di personalizzazione di `layout`, R Graphics, da pagina 78

13.3.5 Come inserire formule matematiche

Si ottengono mediante la funzione `plotmath`; si veda `demo(plotmath)`. È possibile aggiungere formule:

- sintassi similare a latex (rendering peggiore)
- i simboli matematici sono expression in R, ed è necessario utilizzare la funzione `expression` per includerli in un grafico
- una lista di simboli ammessi è documentata in `?plotmath`

L'idea è che invece di passare una etichetta di puro testo ad un label di qualcosa passiamo una `expression`. Ad esempio

```
plot(0,0, main= expression(theta==0),
          xlab=expression(sum(x[i]*y[i], i==1, n)))
```

E' possibile anche concatenare stringhe mediante l'asterisco, del tipo

```
...
xlab = expressione("The mean (" * bar(x) * ")is " * sum(x[i]/n, i==1,n))
...
```

Nel caso volessimo utilizzare un valore computato nell'annotazione matematica, bisogna usare la `substitute` function, ad esempio

```
x <- rnorm(100)
plot(x, xlab=substitute( bar(x) == k, list(k=mean(x))  ))
```

13.4 L'uso del pacchetto ggplot2

Costituisce una implementazione della *grammar of graphics* una descrizione di come i grafici possano essere scomposti in elementi di base combinabili a piacere.

```
library(ggplot2)
```

13.4.1 Costruzione base

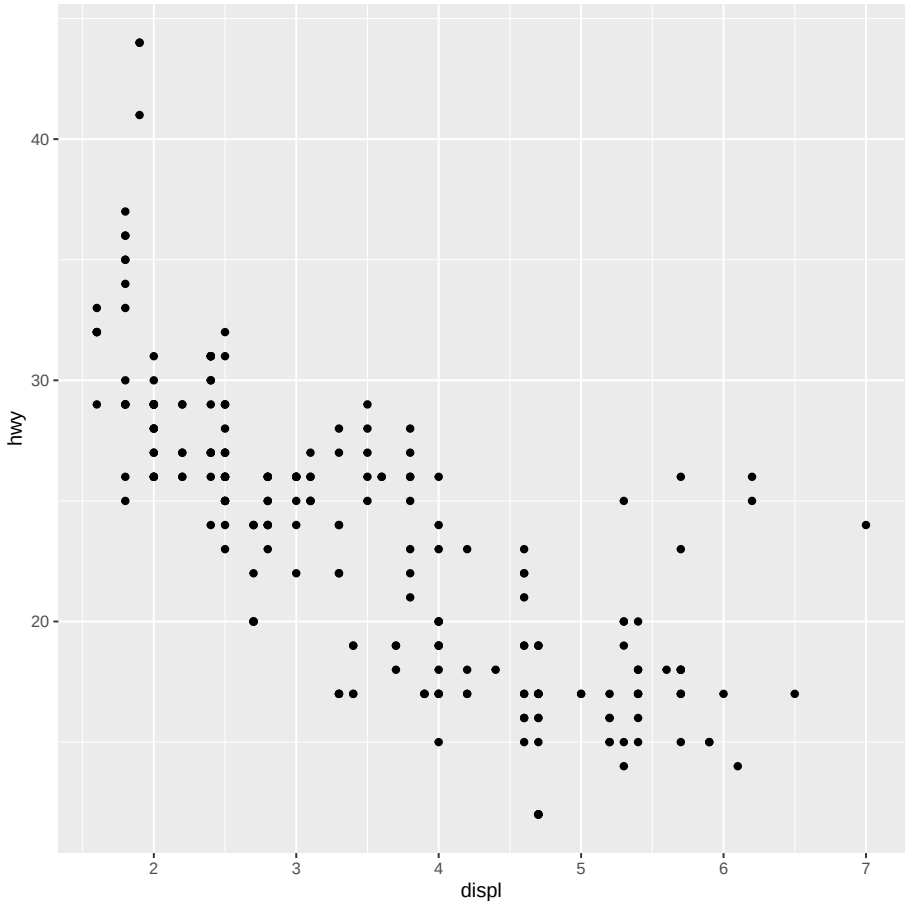
`ggplot2` costruisce immagini attraverso layer:

- si inizia specificando mediante `ggplot` il dataset dal quale prendere i dati
- si aggiunge a questo i layer di interesse mediante le funzioni `geom_` e l'operatore `+`:

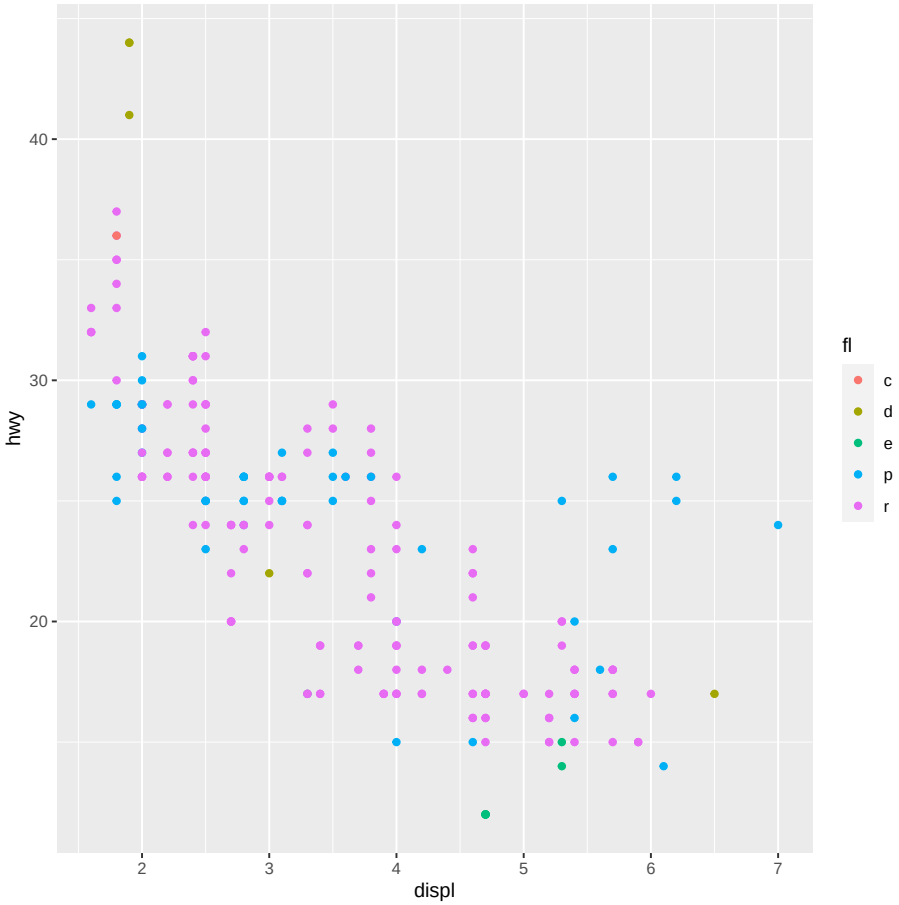
Un template tipico per la costruzione di un grafico:

```
ggplot(data = <DATA>) +  <GEOM_FUNCTION>(mapping = aes(<MAPPINGS>)) + ...
```

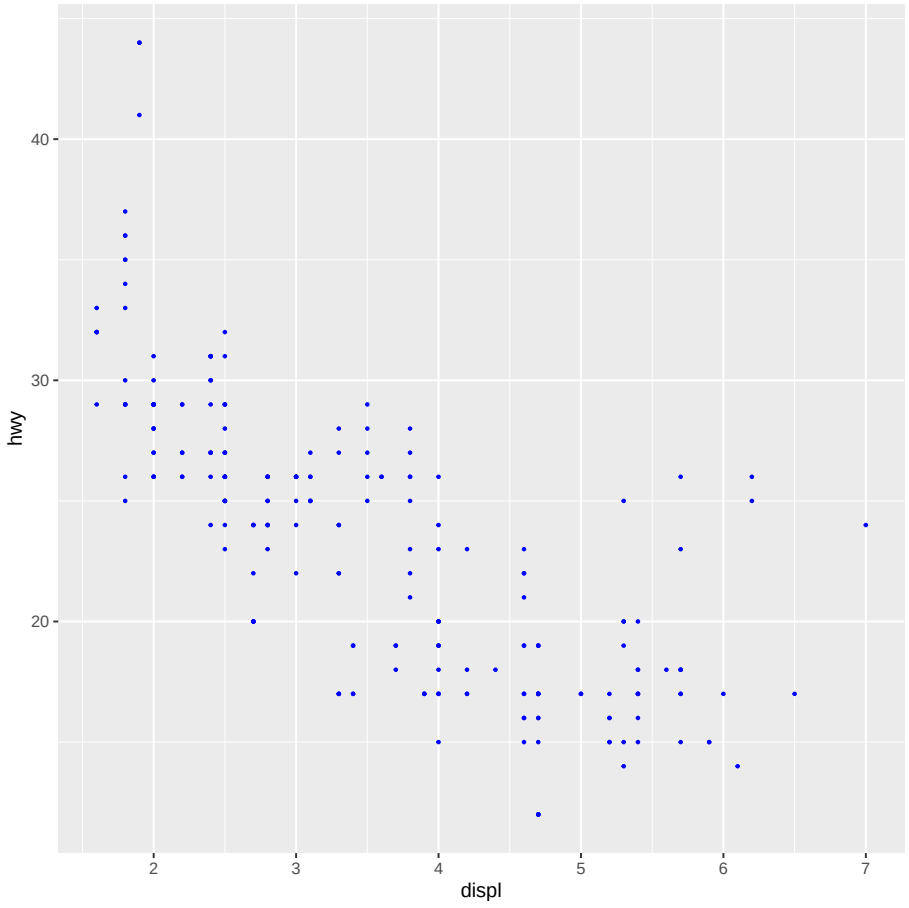
```
## vediamo il dataset macchine
mpg

## # A tibble: 234 x 11
##    manufacturer model      displ  year   cyl trans      drv     cty   hwy fl    cl
##    <chr>        <chr>      <dbl> <int> <int> <chr>      <chr> <int> <int> <chr> <c
##  1 audi         a4           1.8  1999     4 auto(l5)   f        18    29 p     co
##  2 audi         a4           1.8  1999     4 manual(m5) f        21    29 p     co
##  3 audi         a4           2    2008     4 manual(m6) f        20    31 p     co
##  4 audi         a4           2    2008     4 auto(av)   f        21    30 p     co
##  5 audi         a4           2.8  1999     6 auto(l5)   f        16    26 p     co
##  6 audi         a4           2.8  1999     6 manual(m5) f        18    26 p     co
##  7 audi         a4           3.1  2008     6 auto(av)   f        18    27 p     co
##  8 audi         a4 quattro   1.8  1999     4 manual(m5) 4        18    26 p     co
##  9 audi         a4 quattro   1.8  1999     4 auto(l5)   4        16    25 p     co
## 10 audi         a4 quattro   2    2008     4 manual(m6) 4        20    28 p     co
## # ... with 224 more rows

## plottiamo la relazione
b <- ggplot(data = mpg)

b + geom_point(mapping = aes(x = displ, y = hwy))
```




```
# colori dipendenti dal dato vanno in aes che mappa i dati agli estetici
b + geom_point(mapping = aes(x = displ, y = hwy, color = fl))
```




```
# viceversa parametri fissi/validi per tutti i plot vanno fuori
b + geom_point(mapping = aes(x = displ, y = hwy), color = 'blue', size = 0.5)
```

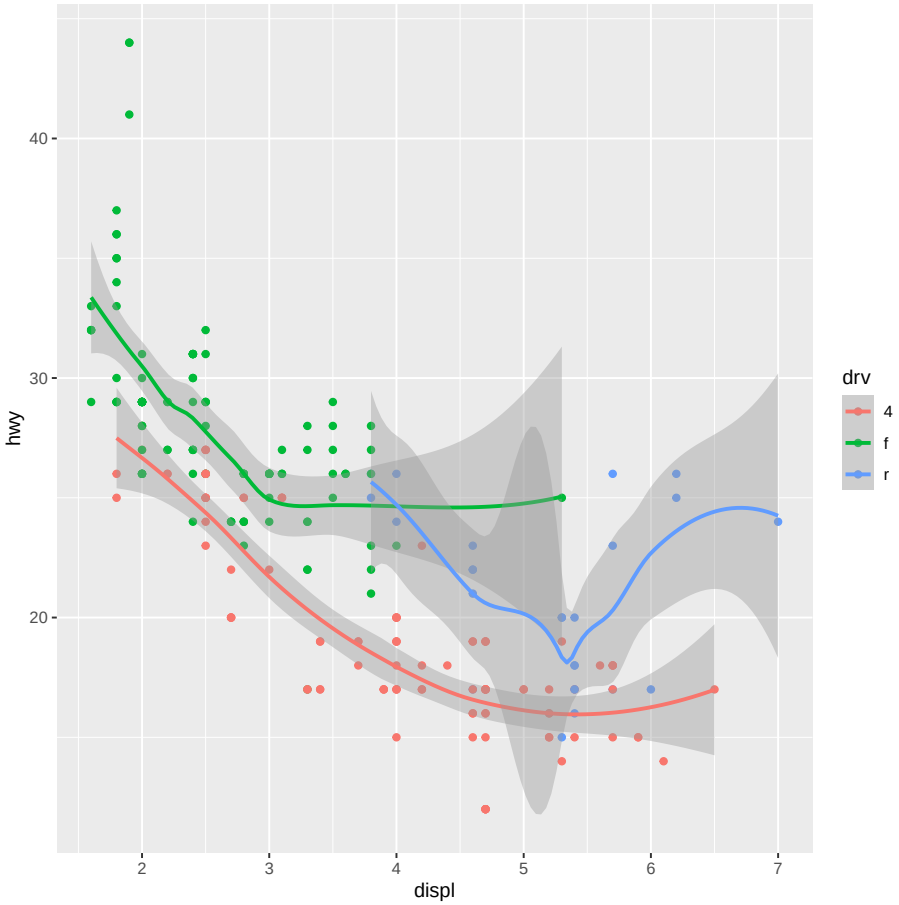


Gli `aes` accettati dipendono dalla funzione `geom` utilizzata, quindi riferirsi alla relativa documentazione.

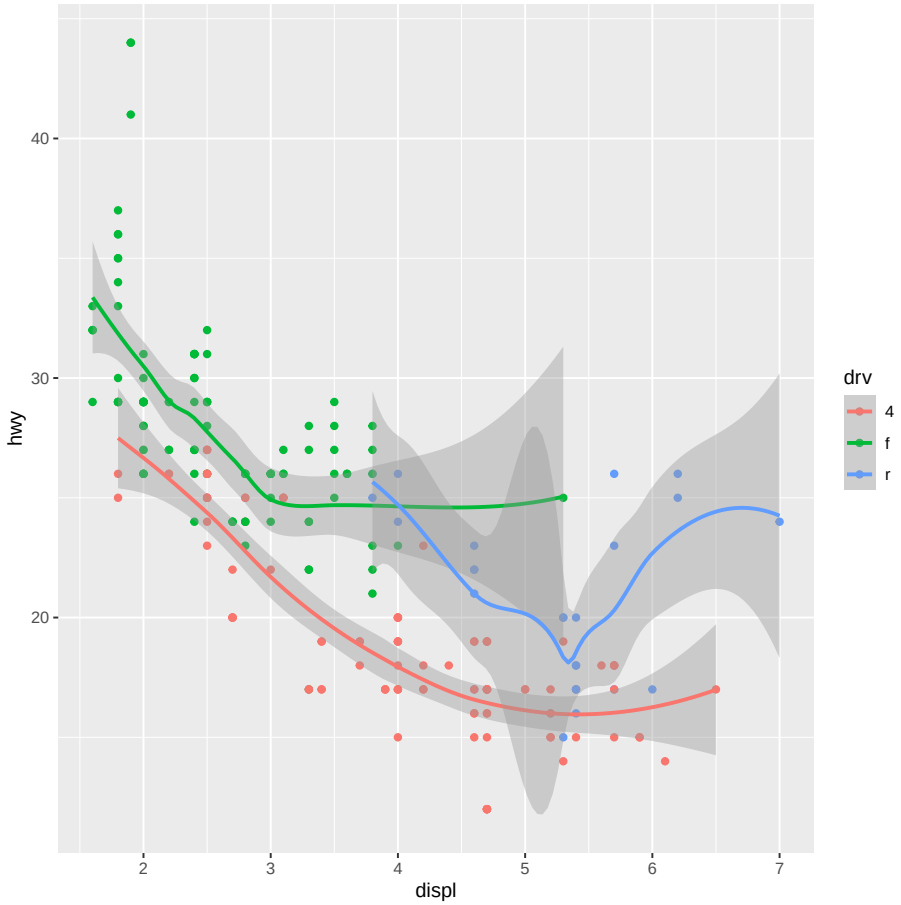
13.4.2 Layer molteplici

L'aggiunta di layer può portare a duplicazioni che meglio evitare; specificando il mappaggio nella chiamata a `ggplot` tali parametri rimarranno fissi per tutte le chiamate

```
# aggiunta di altri layer: aggiungiamo un loess, diverse ripetizioni
b + geom_point(mapping = aes(x = displ, y = hwy, col = drv)) +
    geom_smooth(mapping = aes(x = displ, y = hwy, group = drv, col = drv))

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



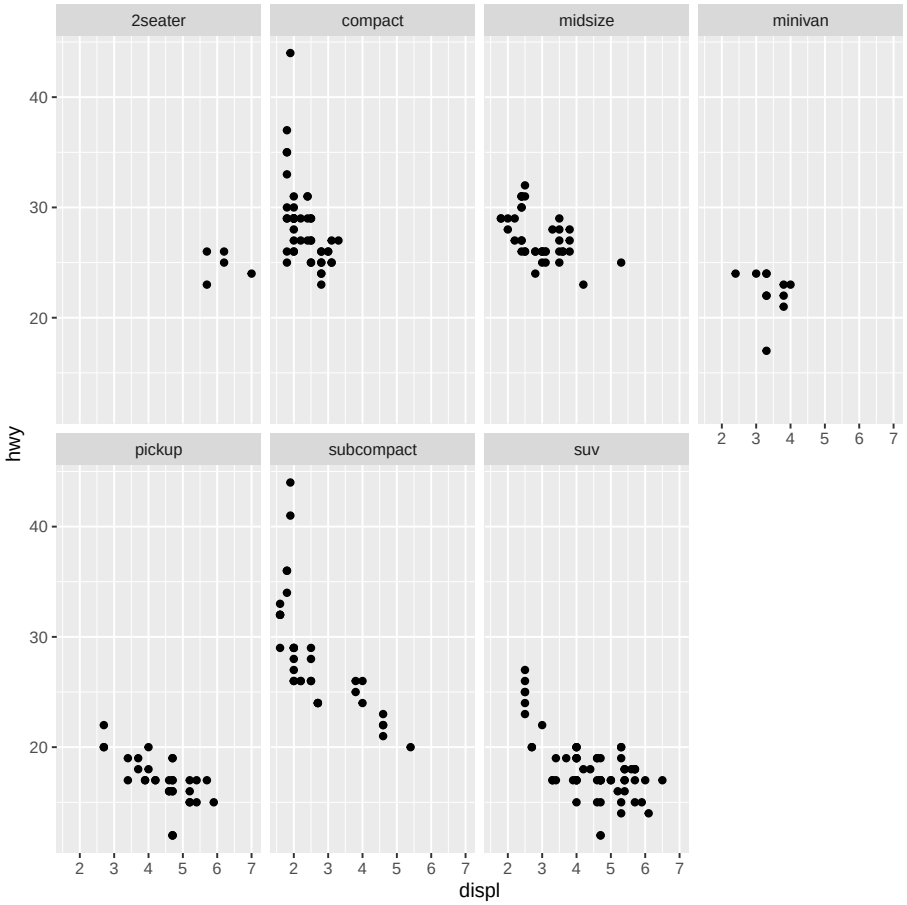

```
# meglio allora fare così
ggplot(data = mpg, mapping = aes(x = displ, y = hwy, col = drv)) +
    geom_point() +
    geom_smooth(mapping = aes(group = drv))

## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



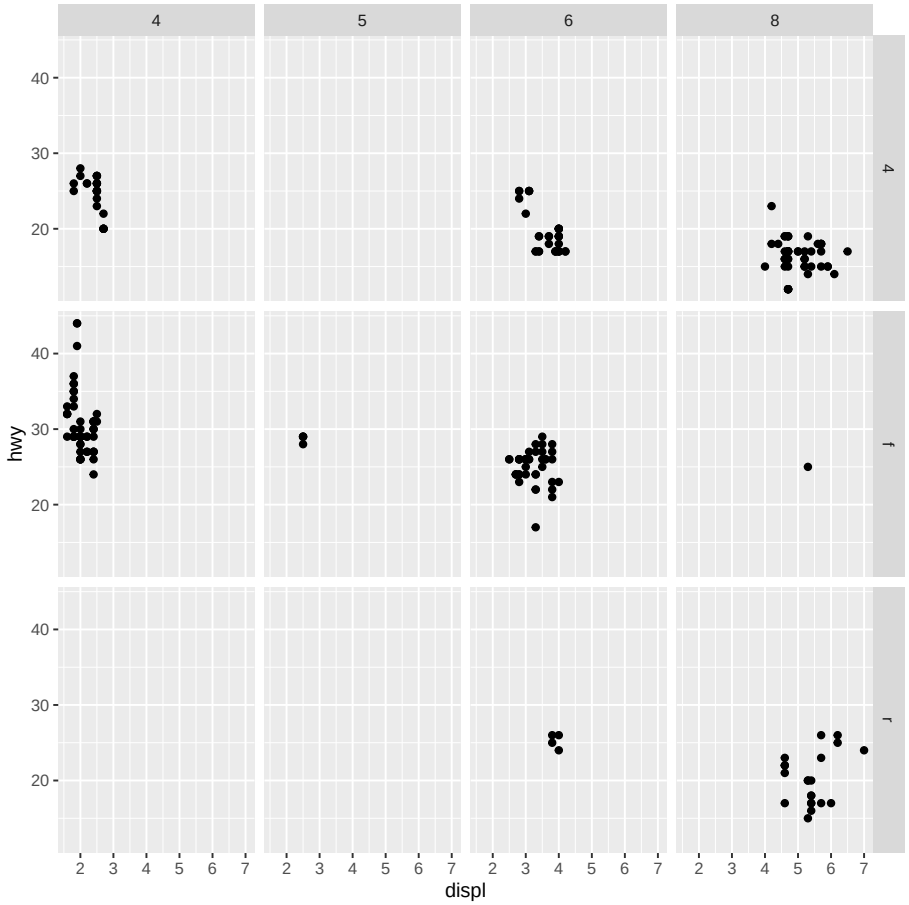
13.4.3 Faceting

Consiste nello stratificare l'immagine per una o più variabili e può esser fatto mediante `facet_wrap` (utile soprattutto se il fattore di stratificazione è uno) o `facet_grid` (utile per due fattori di stratificazione

```
c <- b + geom_point(mapping = aes(x = displ, y = hwy))
c + facet_wrap(~ class, nrow = 2)
```



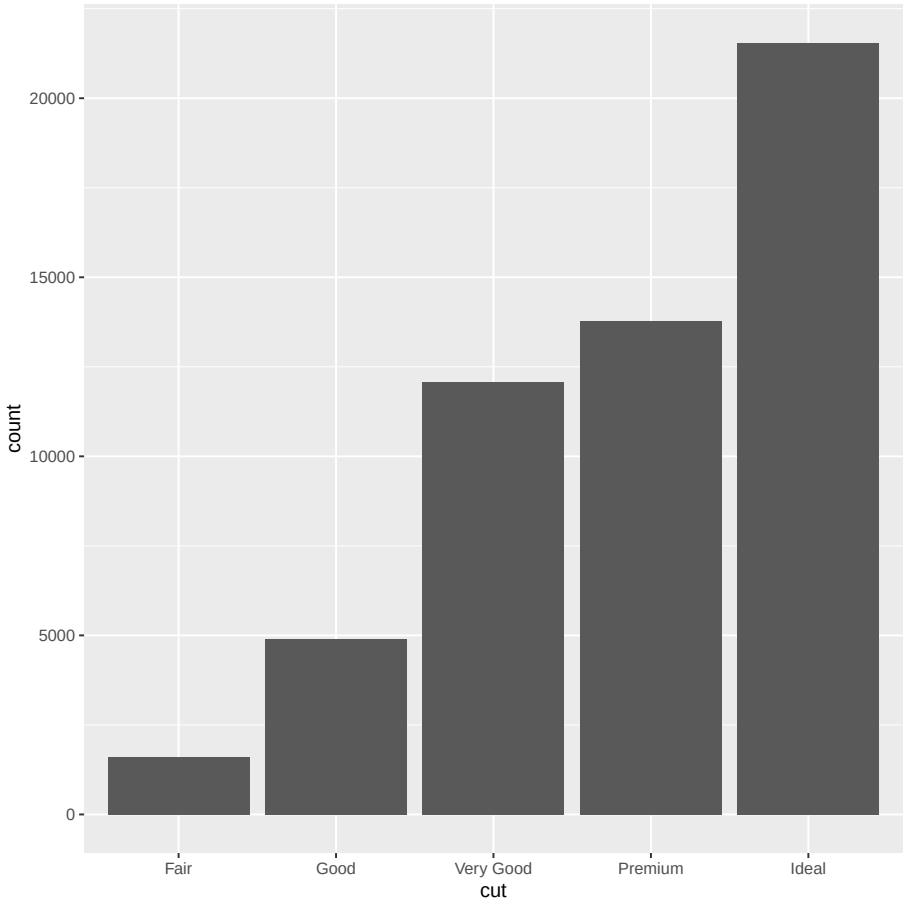

```
## # oppure
## c + facet_grid(. ~ class)
## c + facet_grid(class ~ .)
c + facet_grid(drv ~ cyl) # righe ~ colonne
```



13.4.4 Trasformazioni

Il seguente grafico mostra sulle x la variabile `cut`, mentre sulle y il conteggio senza che sia stato specificato nulla

```
ggplot(data = diamonds) +
  geom_bar(mapping = aes(x = cut))
```



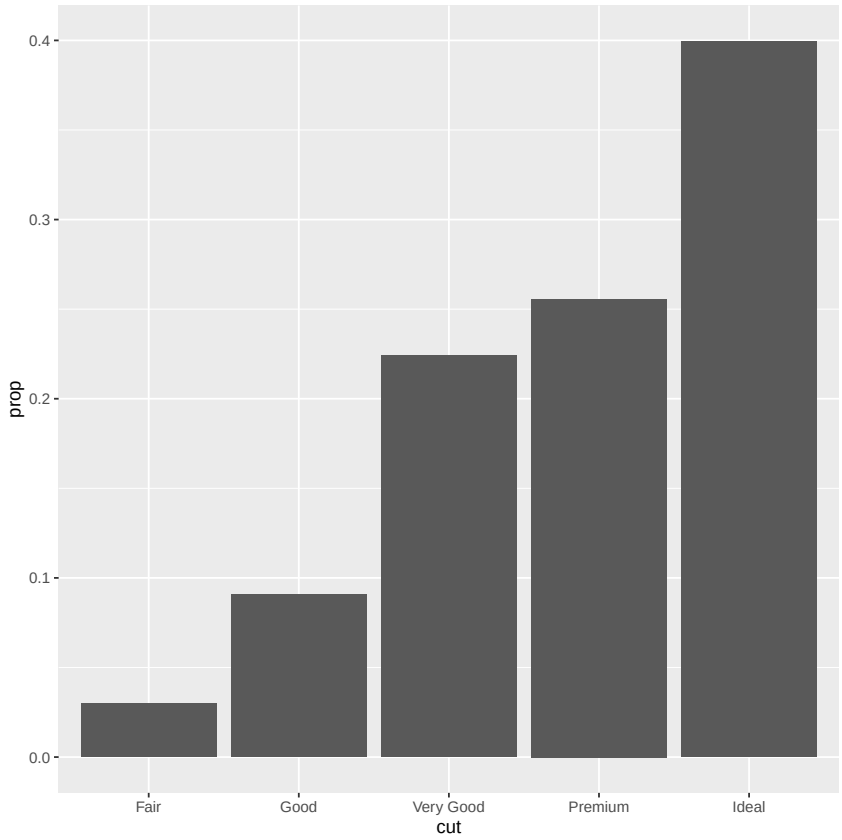
Molti grafici plottano direttamente i dati, altri calcolano statistiche/trasformazioni da plottare: es grafici a barre e istogrammi delle frequenze, gli smoother delle predizioni di un modello, i boxplot calcolano le statistiche da plottare.
L'algoritmo utilizzato per calcolare un nuovo valore è detto `stat` (statistical transformation); di sopra il tutto funziona `geom_bar` ha valore `count` come default stat, ossia utilizza `stat_count`.
In generale:

- si possono utilizzare `geom`s e `stat`s intercambiabilmente. Ad esempio si può ricreare il medesimo plot precedente utilizzando `stat_count` come segue

```
ggplot(data = diamonds) + stat_count(mapping = aes(x = cut))
```

- il tutto funziona perchè ogni geom ha una stat di default; e ogni stat ha una geom di default. Questo fa si che si possano utilizzare le geoms senza preoccuparsi della stat
- se si vuole modificare una stat di default utilizzata si può farlo come segue; per quello che viene calcolato vedere la sezione *computed variables* nell'help di `geom_bar`.

```
ggplot(data = diamonds) +
    geom_bar(mapping = aes(x = cut, y = stat(prop), group = 1))
```

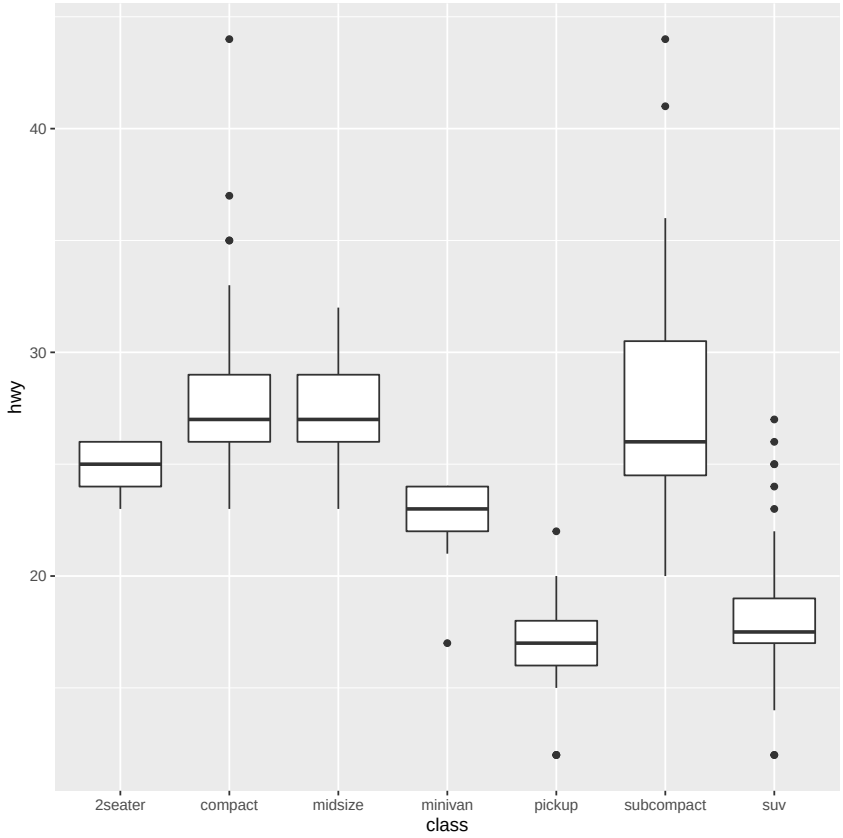


13.4.5 Coordinate

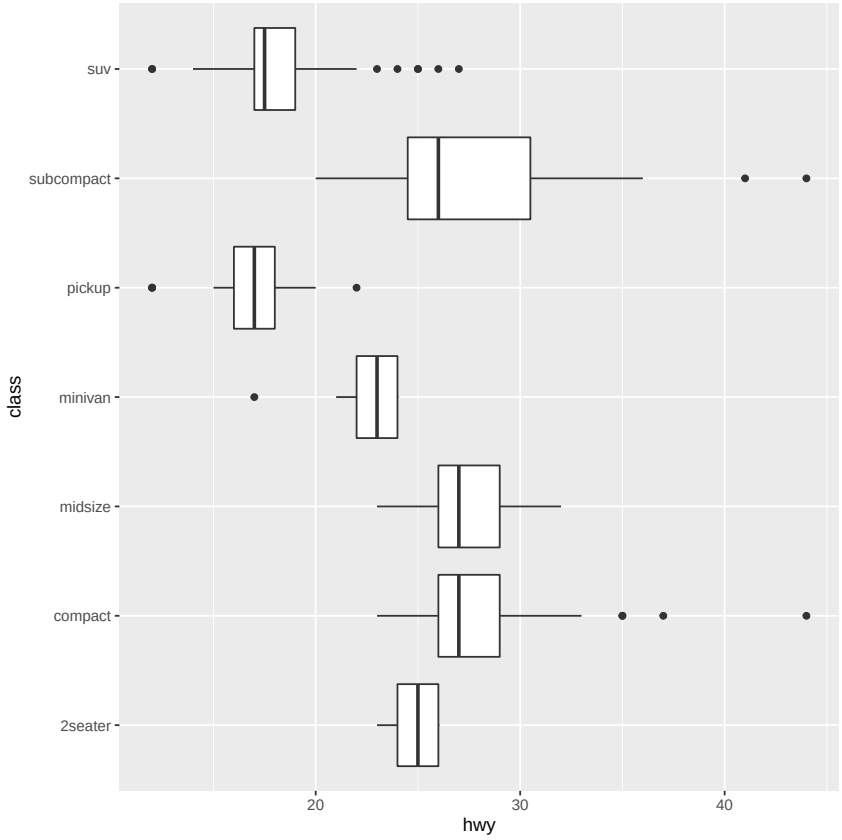
Possibile lavorare sulle coordinate:

- `coord_flip` switch `x` e `y` ed è utile ad esempio per orientare i grafici verticali in orizzontale (o per etichette lunghe)

```
a <-  ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom_boxplot()
a
```


40
30
20
hwy
2seater
compact
midsize
minivan
pickup
subcompact
suv
class

```
a + coord_flip()
```



- `coord_quickmap` setta l'aspetto corretto per le mappe sets the aspect ratio correctly for maps

13.5 Colori

Spesso si desidera specificare i colori da assegnare alle parti di un plot o modificare il set di colori fornito di default; vi sono diverse funzioni del sistema base e alcuni pacchetti esterni che possono venire in aiuto.
Per specificare colorisi può alternativamente:

- specificare un numero (`col=1` è il nero, `2` il rosso, `3` il verde ecc)
- specificare una stringa identificativa del colore: la lista completa si ha mediante `color()`
- una stringa esadecimale rgb del tipo `#RRGGBB`, generabile mediante la funzione `rgb`, o importabile da altri software specifici (`gcolor2`)

Tutti i possibili colori stringa che si possono utilizzare in qualsiasi funzione di plot sono listati da `colors`

```
colors()[1:10]

##  [1] "white"         "aliceblue"     "antiquewhite"  "antiquewhite1" "antiquewhite
##  [7] "antiquewhite4" "aquamarine"    "aquamarine1"   "aquamarine2"
```

Se occorre scegliere un **insieme di colori** che stiano bene tra loro si possono utilizzare funzioni come `rainbow`, `hcl` e simili; si veda i relativi `example`.
Scale di grigi possono essere generate con `grey`.
Vi è un basso supporto per i *fill-pattern* (ovvero riempimenti in bianco e nero con disegnini).
Il pacchetto `grDevices` ha due funzioni che prendono in input colori e aiutano a **interpolare tra due colori**:

- `colorRamp`: prende una palette di colori e ritorna una funzione che prende in input un valore tra 0 e 1, che indica gli estremi della palette (es vedi funzione `gray` che interpola tra bianco e nero, attraverso una scala di grigi)
- `colorRampPalette`: prende una palette di colori e ritorna una funzione che prende un intero e restituisce un vettore di colori che interpolano la paletta (come `heat.colors` o `topo.colors`)

`colorRamp` crea una funzione che fornisce in ouput una matrice con i valori r (prima colonna) g (seconda colonna) e b (terza) del colore impostato

```
## Creiamo una progressione tra rosso e blu
pal <- colorRamp(c("red", "blue"))
## Rosso
pal(0)

##      [,1] [,2] [,3]
## [1,]  255    0    0

## Di sotto il blu
pal(1)

##      [,1] [,2] [,3]
## [1,]    0    0  255

## Un colore interpolato
pal(0.5)

##       [,1] [,2]  [,3]
## [1,] 127.5    0 127.5

## Usiamo una sequenza per creare piu colori
pal(seq(0,1,length=10))

##            [,1] [,2]      [,3]
##  [1,] 255.00000    0   0.00000
##  [2,] 226.66667    0  28.33333
##  [3,] 198.33333    0  56.66667
##  [4,] 170.00000    0  85.00000
```

```
##  [5,] 141.66667    0 113.33333
##  [6,] 113.33333    0 141.66667
##  [7,]  85.00000    0 170.00000
##  [8,]  56.66667    0 198.33333
##  [9,]  28.33333    0 226.66667
## [10,]   0.00000    0 255.00000

## Come creare il colore da dare in input in formato rgb
rgb(pal(0.5)/255)

## [1] "#800080"

rgb(pal(seq(0,1,length=10))/255 )

##  [1] "#FF0000" "#E3001C" "#C60039" "#AA0055" "#8E0071" "#71008E" "#5500AA" "#3900C6" "#1C00E
## [10] "#0000FF"
```

`colorRampPalette` è simile ma crea una funzione che restituisce un vettore con un numero di colori specificato, interpolanti gli estremi

```
pal2 <- colorRampPalette(c("red","yellow"))
## il primo è rosso il secondo giallo (per fare giallo servono
## rosso e verde)
pal2(2)

## [1] "#FF0000" "#FFFF00"

## Dieci sfumature tra rosso e giallo
pal2(10)

##  [1] "#FF0000" "#FF1C00" "#FF3800" "#FF5500" "#FF7100" "#FF8D00" "#FFAA00" "#FFC600" "#FFE20
## [10] "#FFFF00"
```

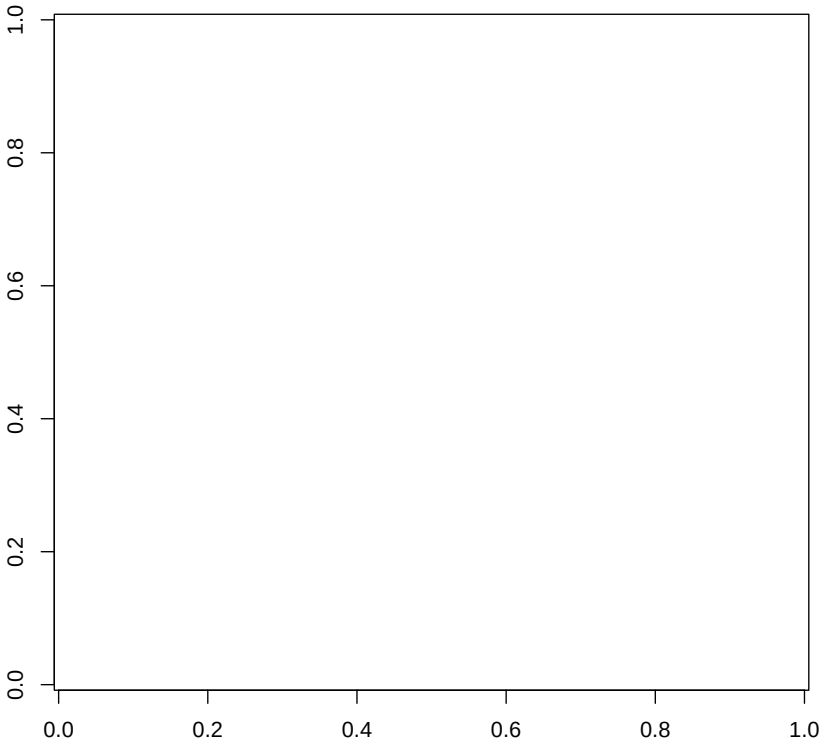
Un pacchetto del CRAN che contiene interessanti palette di colori è `RColorBrewer`. Vi sono tre tipologie di palette:

- palette *sequenziali*: utili per dati ordinati (qualitativi o continui)
- palette *qualitative*: per dati qualitativi non ordinati
- palette *diverging*: per colori che divergono/differenziano da qualcosa (es deviazioni dalla media via via più evidenziate mediante colore)

Si veda anche http://colorbrewer2.org/ in proposito.
Le palette disponibili possono esser ottenute mediante `display.brewer.all`. Il primo gruppo è quello delle palette sequenziali, il secondo (che inizia con `Set3`) quello delle palette qualitative il terzo quello delle palette diverging[6].
L'altra (unica) funzione utile del pacchetto è `brewer.pal`, che prende in argomento il numero di colori che si desiderano nella propria palette (solitamente piccolo, 2, 3 o al massimo 4) e il name del palette come secondo argomento). Le informazioni che si ottengono da questi, possono esser passati a `colorRamp` e `colorRampPalette`. Vediamo un esempio:

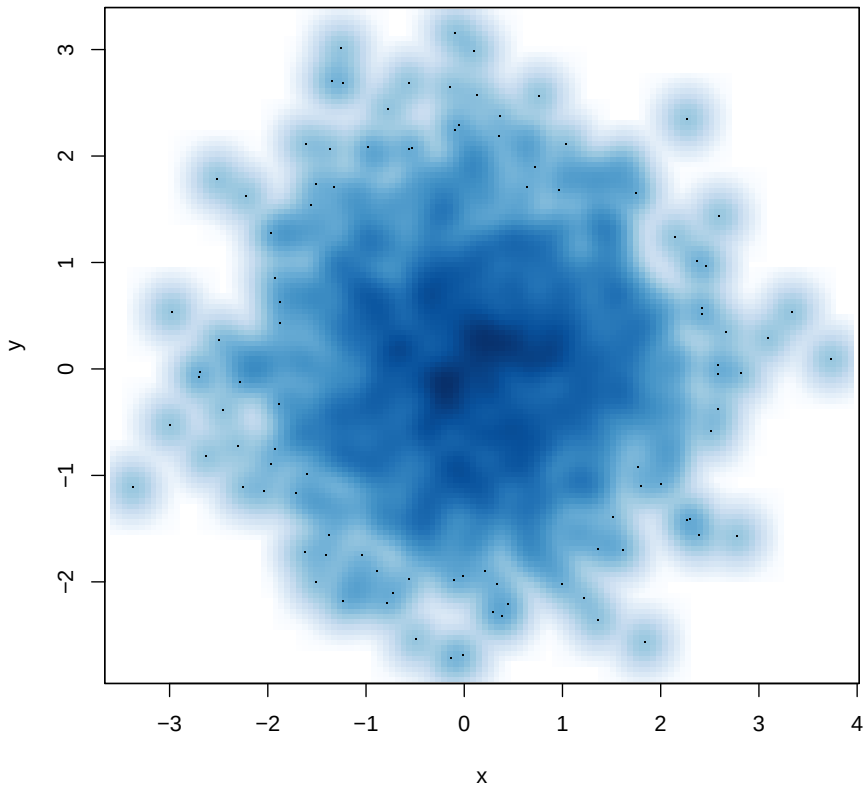
[6] Si può pensare quelli a sinistra del centro come i colori positivi, a destra i colori negativi

```
library(RColorBrewer)
## 3 è il numero di colori che vogliamo per il nostro pallette
cols <- brewer.pal(3, "BuGn")
cols

## [1] "#E5F5F9" "#99D8C9" "#2CA25F"

pal3 <- colorRampPalette(cols)
image(volcano, col=pal(20))
```



Un'altra funzione che fa uso di colorRampPalette è `smoothScatter`, una funzione per la stampa di scatterplot utile per casi in cui vi sono numerosi punti e quindi si preferisce associare la presenza di tanti punti ad una maggiore intensità di colore (piuttosto che a una zona nera): è tipo un istogramma 3d visto dall'alto.

```
x <- rnorm(1000)
y <- rnorm(1000)
smoothScatter(x,y)
```



La **trasparenza** del colore può essere specificata mediante il parametro `alpha` della funzione `rgb` (0 consiste nella massima trasparenza e 1 è non trasparente).

```
rgb( pal(seq(0,1,length=5))/255 , alpha=.5)

## [1] "#FF000080" "#BF004080" "#80008080" "#4000BF80" "#0000FF80"
```

13.6 Note di Teoria dei Grafici

Il grafico è una rappresentazione sintetica della realtà che permette con un solo colpo d'occhio di sintetizzare una gran quantità di informazione altrimenti non gestibili dalla nostra mente.
Il beneficio del grafico non è tuttavia solo di tipologia *economia di pensiero e gestione celebrale dell'informazione*: osservare che un set di osservazioni segue un particolare pattern ci permetterà spesso di *muoverci dallo specifico al generale*; è l'abilità dei grafici di suggerire teorie e far sorgere domande che li rende così importanti.

Regole per buoni grafici In questa sezione effettueremo alcune riflessioni sulle *regole* base *per disegnare grafici di buona qualità*:

- non ha senso utilizzare i grafici per visualizzare limitate quantità di dati: il cervello umano riesce ad afferrare uno , due o tre valori: ad esempio non ha senso fare un diagramma di frequenze percentuali per i maschi e le femmine di una classe, sono semplicemente due percentuali.
- i grafici sono buoni se si basano su dati buoni: la creatività non può produrre buoni grafici a partire da dati di scarsa qualità
- rendere le cose semplici e facilmente comprensibili: complessità immotivata puó esser introdotta mediante abbellimenti non necessari, colori ed effetti 3d. Non ha senso utilizzare sei colori ed effetti tridimensionali per rappresentare cinque valori che possono esser resi bene con un plot storico.
- i grafici non dovrebbero fornire una visione distorta dei dati che ritraggono: l'utilizzo di scale appropriate e non distorte (es spazio grafico da 1 a 2 e da 5 a 10 uguale, occhio all'uso logaritmico che appiattisce)
- il "lie factor" = (dimensione effetto mostrato dal grafico)/(dimensione effetto mostrato dai dati) non deve superare uno

Per disegnare buoni grafici:

- se il fenomeno da rappresentare è semplice, mantienilo semplice
- se è complesso, cerca di semplificarlo
- dire la verità, non distorcere i dati

La percezione celebrale della realtà Il cervello percepisce l'informazione visiva tramite gli occhi. Un'illusione si ha quando il cervello deriva un informazione non corretta da ciò che viene visualizzato. Illusioni possono provenire da diverse fonti; in particolare quelle geometriche includono:

- distorsioni da lunghezza
- distorsioni da angolo
- distorsioni da area
- distorsioni da forma

Quando disegnamo un grafico effettuiamo un encoding di valori numerici in attributi grafici (linee, aree, colori, angoli); quando poi andiamo a guardare il grafico, il nostro obiettivo è decodificare attributi grafici ed estrarre informazioni sui valori che in essi sono stati codificati.
Per disegnare grafici in una maniera efficiente occorre scegliere quali attributi grafici sono più facilmente decodificati, dati i fenomeni che vogliamo rappresentare.
Sarebbe pertanto interessante selezionare i possibili attributi grafici e ordinarli sulla base della semplicita' di decodifica.
Vi sono una serie di regole empiriche riguardanti la percezione del visivo:

- la probabilita' di notare una differenza tra diversi oggetti cresce al crescere della differenza stessa
- l'informazione percepita dalla mente dipende dall'informazione vera, dal metodo di codifica e dal singolo che decodifica.

Cleveland e McGill hanno condotto un esperimento per costruire un rank dei migliori encoding grafici, considerando 127 soggetti cui sono stati sottoposti 7 encoding e a cui sono stati richiesti 3 giudizi su "quanto fosse quella rappresentazione".
Il criterio di valutazione era stato:

```
errore = | valore_pensato\% - valore_vero\% |
```

Il rank degli encoding (dal migliore al peggiore è il seguente):

- posizione su scala comune (es plot serie storica)
- posizione su medesima scala, anche se non allineati (es boxplot)
- lunghezza
- angoli/pendenze (es confrontare due rette dalla pendenza)
- area (istogrammi con basi di differente ampiezza) +
- volume
- proprietà colori

Il consiglio è di utilizzare gli encoding più alti possibili, preferibilmente i primi tre; attenzione all'utilizzo del quarto, evitare quinto, sesto e settimo. Ad esempio è dimostrato che la percezione delle aree e dei volumi è conservativa: tendiamo a sottostimare i valori grandi (aree grandi) rispetto alle piccole e sovrastimare le piccole rispetto alle grandi.

Capitolo 14

Misc e sandbox

14.1 Interfaccia con database

14.1.1 Access

14.1.1.1 Interagire con Access ed altri DMBS

Le strade per farlo sono diverse; una cross-platform è il pacchetto `RODBC`. Si vedrà una **sessione tipica** di relazione con un DBMS.
Una sessione tipica inizia con la creazione di una *connessione*; questo in `RODBC` è fatto mediante la funzione `odbcConnect`, alla quale specificare il la *data source name*[1], login e pass. Per ciò che riguarda Access, in ambiente Windows (con access installato) si può comandare

```
con <- odbcConnectAccess("file.mdb", uid= "admin", pwd = "" )
```

`con` è la *connessione* con cui si deve interagire d'ora in poi.

Ottenere informazioni Per **ottenere informazioni sulla connessione** basta chiamare/stampare l'oggetto `con`.
Per osservare le **tabelle disponibili** in un dato DBMS, comandare

```
sqlTables(con, tableType="TABLE")$TABLE_NAME
```

Per **conoscere le colonne di una determinata tabella** si utilizza `sqlColumns`

```
sqlColumns(con, "miatabella")
```

Salvare dati Per ottenere un **data.frame da una tabella del DBMS**:

```
ima <- sqlFetch(con, "tblValidazioneIMA", stringsAsFactors=F)
```

Cancellare dati `sqlClear` elimina tutti i record di una tabella; `sqlDrop` elimina una tabella.

[1] Ad esempio `mysql://john:pass@localhost:port/my_db`

Eseguire un comando SQL Per **eseguire un generico comando SQL**, si utilizza `sqlQuery`: ad esempio

```
sqlQuery(con, "delete from TabellaPrincipale")
```

Chiudere una connessione Per **chiudere una connessione** si utilizza `odbcClose`

```
odbcClose(con)
rm(con)
```

14.2 Effettuare confronti

Per effettuare confronti sono utili le funzioni `identical` e `all.equal`

`identical` Serve per testare la perfetta uguaglianza di due oggetti arbitrari

```
identical(Indometh, Indometh)

## [1] TRUE
```

`all.equal` Serve per testare l'eguaglianza approssimata nei numerici, con la possibilità di specificare la tolleranza al di sotto della quale considerare due numeri uguali (di default settata ad un valore dipendente dalla macchina).
La funzione restituisce una stima della distanza se questa è sopra la tolleranza o `TRUE` se. Nel primo caso è necessario usarla accompagnata con `isTRUE`

```
## Utilizzo la tolleranza di default
all.equal(c(0.5, 0.6), c(0.6, 0.7))

## [1] "Mean relative difference: 0.1818182"

isTRUE(all.equal(c(0.5, 0.6), c(0.6, 0.7)))

## [1] FALSE

## Posso specificare la tolleranza (qui ci accontentiamo di
## differenze grossolane)
all.equal(c(0.5, 0.6), c(0.6, 0.7), tolerance = 0.2)

## [1] TRUE
```

14.3 Inserire un elemento (vettore) in una certa posizione di una lista (`data.frame`)

Si utilizzi `append`, specificando l'opzione `after`; essendo vettori, liste e data.frame delle liste si possono usare con tutti questi dati

```
## esempio con vettore
append(letters[1:3], 'x', 2)

## [1] "a" "b" "x" "c"

## esempio con lista
x <- list('a' = 1, 'b' = 2, 'c' = 3)
append(x, list('z' = 0), 2)

## $a
## [1] 1
##
## $b
## [1] 2
##
## $z
## [1] 0
##
## $c
## [1] 3

## esempio con data.frame
anscombe <- append(anscombe, list('missing' = rep(NA, nrow(anscombe))), 2)
as.data.frame(anscombe)

##    x1 x2 missing x3 x4    y1   y2    y3    y4
## 1  10 10      NA 10  8  8.04 9.14  7.46  6.58
## 2   8  8      NA  8  8  6.95 8.14  6.77  5.76
## 3  13 13      NA 13  8  7.58 8.74 12.74  7.71
## 4   9  9      NA  9  8  8.81 8.77  7.11  8.84
## 5  11 11      NA 11  8  8.33 9.26  7.81  8.47
## 6  14 14      NA 14  8  9.96 8.10  8.84  7.04
## 7   6  6      NA  6  8  7.24 6.13  6.08  5.25
## 8   4  4      NA  4 19  4.26 3.10  5.39 12.50
## 9  12 12      NA 12  8 10.84 9.13  8.15  5.56
## 10  7  7      NA  7  8  4.82 7.26  6.42  7.91
## 11  5  5      NA  5  8  5.68 4.74  5.73  6.89
```

14.4 Eliminare i livelli non utili dei factor

Si utilizza `droplevels`, che ha metodi sia per il singolo `factor` che per un `data.frame` (in questo secondo caso si possono specificare anche i fattori che si desidera escludere dalla procedura). Un esempio con singolo vettore:

```
## Esempio singolo fattore
a <- factor(rep(1:4, 2), levels = 1:5, labels = letters[1:5])
table(a)

## a
```

```
## a b c d e
## 2 2 2 2 0

a <- droplevels(a)
table(a)

## a
## a b c d
## 2 2 2 2
```

14.5 Modificare le singole variabili di un dataset serialmente

Torna utile allo scopo la funzione `within`. La funzione valuta delle espressioni in un determinato environment (che può essere una lista), dopodichè ritorna l'environment stesso (eventualmente modificato dalle espressioni valutate).

```
head(iris)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa

iris2 <- within(iris, {
    levels(Species) <-  letters[1:3]
    Petal.Width <- 2
})

head(iris2)

##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4           2       a
## 2          4.9         3.0          1.4           2       a
## 3          4.7         3.2          1.3           2       a
## 4          4.6         3.1          1.5           2       a
## 5          5.0         3.6          1.4           2       a
## 6          5.4         3.9          1.7           2       a
```

14.6 Creare barre di avanzamento

Si usi `utils::txtProgressBar`. Un esempio minimale all'interno di una funzione è il seguente

```
f <- function(...){
    ## inizia progress bar (min e max settati come
    ## ipotetica percentuale di avanzamento)
    pb <- txtProgressBar(min = 0, max = 100)
    for (avanzamento in 1:100){
        ## do stuff per ogni step
        Sys.sleep(0.1)
        ## aggiorna la barra di avanzamento a fine lavoro
        setTxtProgressBar(pb, avanzamento)
    }
    close(pb)
}

f()
```