



# Informatica

Principi di programmazione ed analisi dati in linguaggio  
Scienze Biologiche



Dr. Bruno Bellisario, PhD

# COSA È E COSA NON È R?

```
dens <- density(data, n = npts)
```

```
dx <- dens$x
```

```
dy <- dens$y
```

```
if(add == TRUE)
```

```
  plot(0, 0,
```

```
  if(!is.null(abl))
```

```
    abline(h = abl, col = "red")
```

```
  if(!is.null(lab))
```

```
    text(0, 0, lab, col = "red")
```

```
  if(!is.null(dx2))
```

```
    dx2 <- c(dx, dx2)
```

```
  if(!is.null(y1))
```

```
    y[1.] <- c(y1, y[1.])
```

```
  seqbelow <- rep(NA, length(dx2))
```

```
  if(Fill == TRUE)
```

```
    confshade(dx2, seqbelow, dy2)
```

→ R è un linguaggio e un ambiente per il calcolo statistico e la grafica disponibile come Software Libero secondo i termini della GNU General Public License della Free Software Foundation sotto forma di codice sorgente.

→ R è un sistema completamente pianificato e coerente, piuttosto che un accrescimento incrementale di strumenti specifici e poco flessibili, come spesso accade con altri software di analisi dei dati.

→ R è un linguaggio multipiattaforma in grado di essere eseguito su un'ampia varietà di piattaforme UNIX e sistemi simili (inclusi FreeBSD e Linux), Windows e MacOS.

→ R non è un software "convenzionale" né un sistema statistico ma un ambiente all'interno del quale vengono implementate tecniche statistiche

# COSA È E COSA NON È R?

COSA È E  
SA NON  
È R?

La Free Software Foundation è un'organizzazione non lucrativa fondata nel 1985 per promuovere il software libero. Il software libero permette di utilizzarlo, studiarlo, modificare e distribuirlo liberamente. La FSF è stata costituita nel 1985, per sostenere il software libero, che promuove la libertà di utilizzare, studiare, modificare e distribuire il software. Distribuire, creare e modificare il software è un diritto fondamentale. La FSF è stata costituita nel 1985, per sostenere il software libero, che promuove la libertà di utilizzare, studiare, modificare e distribuire il software.

La Free Software Foundation

FSF



# FREE SOFTWARE FOUNDATION

La Free Software Foundation (FSF) è un'organizzazione senza scopo di lucro fondata da Richard Stallman il 4 ottobre 1985, per sostenere il movimento del software libero, che promuove la libertà universale di studiare, distribuire, creare e modificare software per computer, distribuito in termini di copyleft ("condividi allo stesso modo"), ad esempio con la propria GNU General Public License. La FSF è stata costituita a Boston, Massachusetts, USA, dove ha sede.

# FONDAMENTI: L'ESSENZA DI R

4

- **1** caratteristica importante di R
  - Basato su vettori: R non è un linguaggio procedurale
- **2** motivi per usare R per Data Science
  - Progettato per i dati: R può manipolare grandi set di dati
  - La grafica è comprensibile: le persone capiscono i dati grafici
- **3** principi fondamentali di R
  - Oggetti: tutto ciò che esiste in R è un oggetto
  - Funzioni: tutto ciò che accade in R è una chiamata di funzione
  - Interfacce: verso altri software sono parte integrante di R
- **4** modi di programmare in R
  - Riga di comando: immissione di comandi R in un terminale
  - File di origine: esecuzione di una serie di comandi da un file salvato
  - Interfaccia R GUI: disponibile per Mac, Windows e Linux
  - Pezzi di codice in RStudio: consente il debug durante la scrittura

# OPERATORI MATEMATICI DI BASE

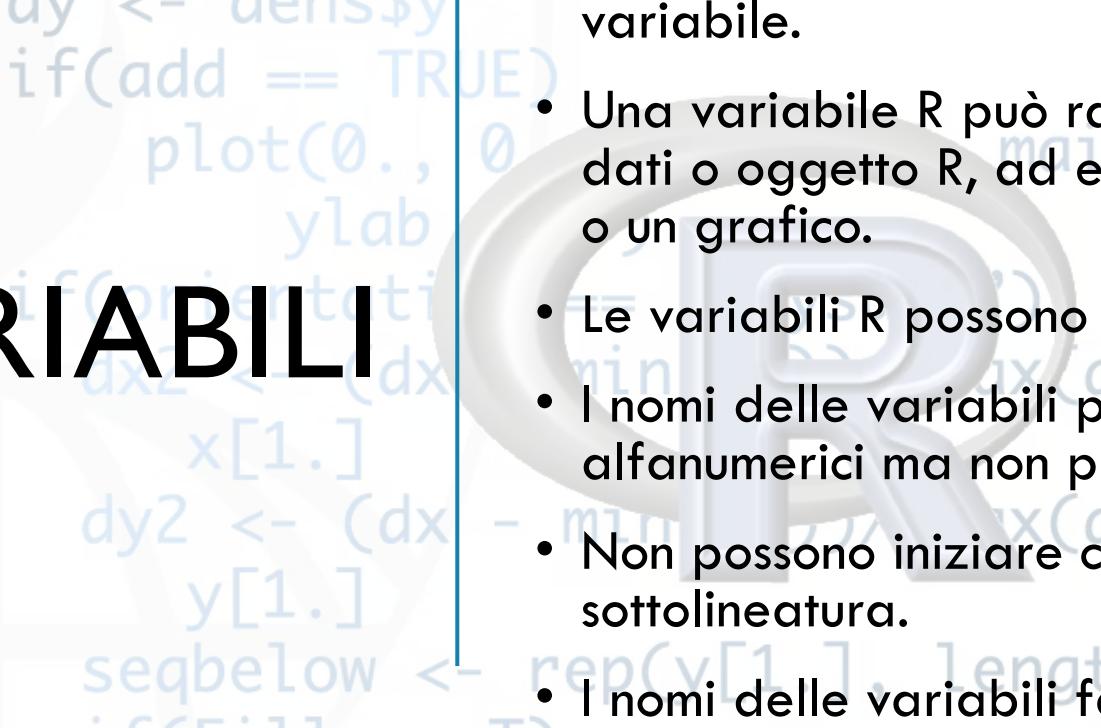
```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0., 0., main = "R")
  lines(dx, dy)
}
dy2 <- (dx - min(dx)) / max(dx)
y[1.] <- rep(y[1.], length(dx))
seqbelow <- if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```

R ha tutte le funzioni matematiche di base e obbedisce all'ordine standard delle operazioni matematiche (PEMDAS):

- Parentesi ()
- Esponenti ^
- Moltiplicazioni x
- Divisioni /
- Addizioni +
- Sottrazioni -



# VARIABILI



# VARIABILI

- A differenza dei linguaggi tipici C++, R non richiede la dichiarazione di una variabile.
- Una variabile R può rappresentare dati o oggetto R, ad esempio un grafico.
- Le variabili R possono essere di qualsiasi tipo.
- I nomi delle variabili possono essere alfanumerici ma non punti.
- Non possono iniziare con un sottolineatura.
- I nomi delle variabili fanno distinzione tra minuscole (case sensitive).

- A differenza dei linguaggi tipizzati staticamente come C++, R non richiede la dichiarazione dei tipi di variabile.
  - Una variabile R può rappresentare qualsiasi tipo di dati o oggetto R, ad esempio una funzione, un risultato o un grafico.
  - Le variabili R possono essere dichiarate nuovamente.
  - I nomi delle variabili possono contenere caratteri alfanumerici ma non punti.
  - Non possono iniziare con un numero o un carattere di sottolineatura.
  - I nomi delle variabili fanno distinzione tra maiuscole e minuscole (case sensitive).

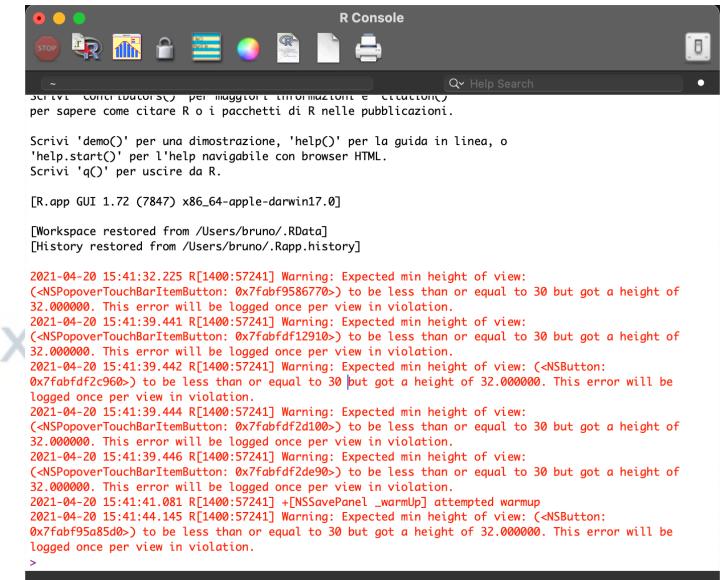
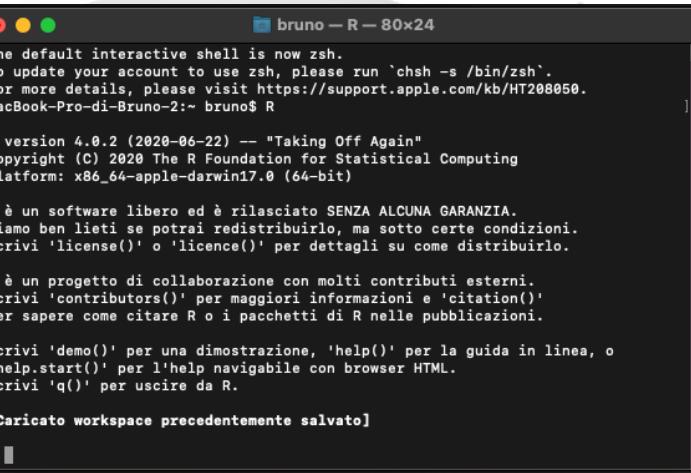


# DOWNLOA D & INSTALLAZI ONE DI R & RSTUDIO

- R è gestito da un team internazionale di sviluppatori che rendono disponibile il linguaggio attraverso la pagina Web **The Comprehensive R Archive Network**. La parte superiore della pagina Web fornisce tre collegamenti per il download di R. Segui il collegamento che descrive il tuo sistema operativo: Windows, Mac o Linux.
- RStudio è un'applicazione come Microsoft Word (tranne che invece di aiutarti a scrivere in inglese, RStudio ti aiuta a scrivere in R). L'interfaccia di RStudio ha lo stesso aspetto per Windows, Mac OS e Linux.
- Puoi scaricare RStudio gratuitamente. Basta fare clic sul pulsante **Scarica RStudio** e seguire le semplici istruzioni che seguono.

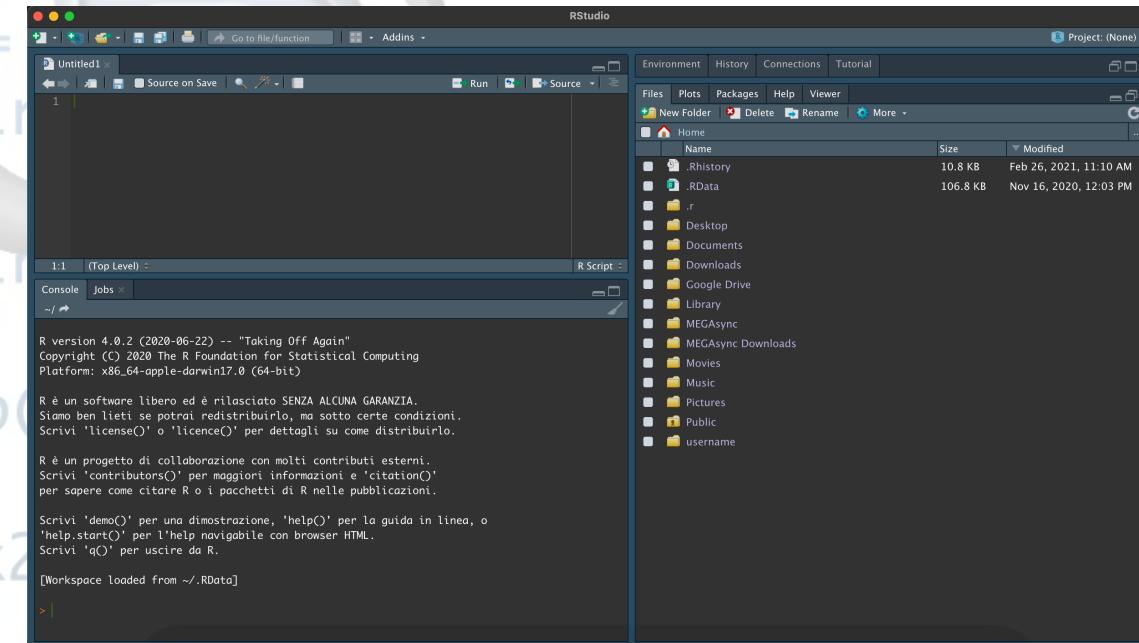
# PANNELLI & BARRA DEGLI STRUMENTI

```
dens <- density(data, n = 1000)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(x, y, type = "l", lwd = 2, col = "black")
  lines(dx, dy, col = "red", lwd = 2)
  if(!is.null(fill))
    confint <- confint(dx, dy, lwd = 2, col = "red", fill = fill)
    if(is.list(confint))
      confshade(dx, confint$lower, confint$upper, dy2 = dy)
    else
      confshade(dx, confint, dy2 = dy)
  else
    confshade(dx, dy, dy2 = dy)
  if(is.list(fill))
    fill <- fill[[1]]
  if(is.character(fill))
    fill <- as.numeric(fill)
  if(is.numeric(fill))
    seqbelow <- rep(y[1.], length(dx))
    if(Fill == T)
      confshade(dx2, seqbelow, dy2 = dy)
    else
      confshade(dx2, seqbelow, dy2 = dy, fill = fill)
  else
    confshade(dx2, seqbelow, dy2 = dy, fill = fill)
  if(is.list(dx))
    dx <- dx[[1]]
  if(is.list(dy))
    dy <- dy[[1]]
```



# PANNELLI & BARRA DEGLI STRUMENTI

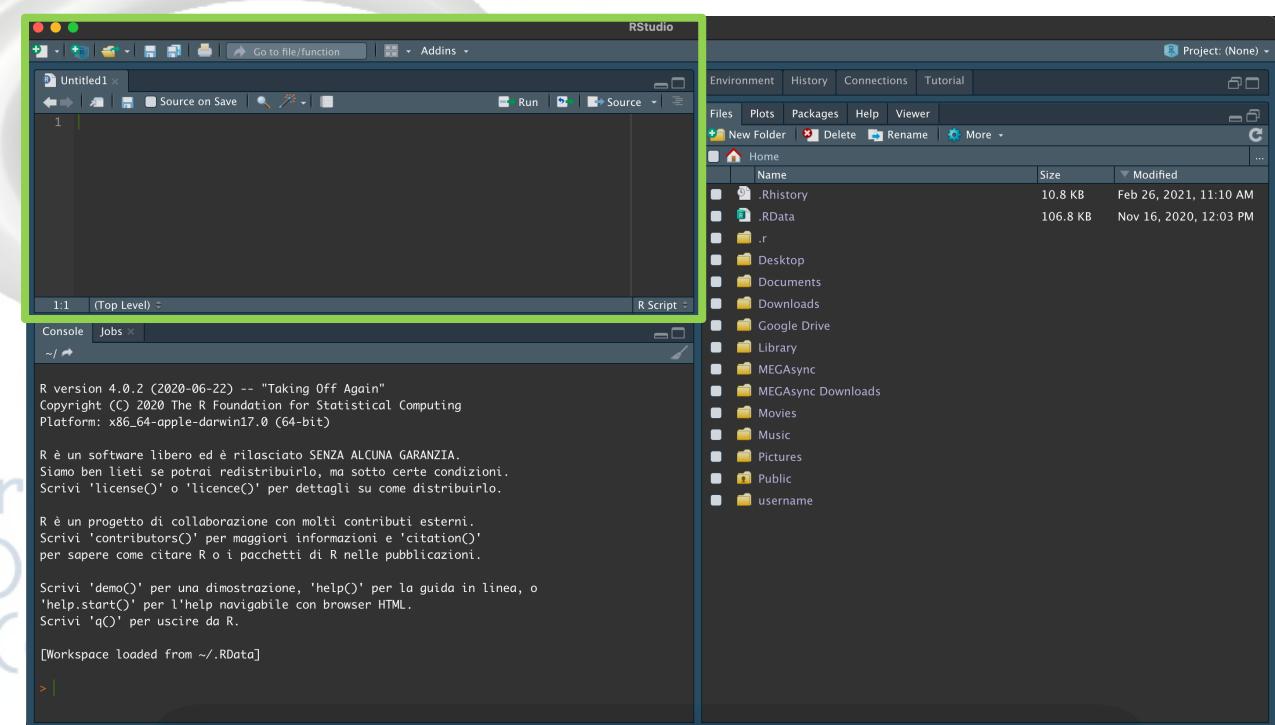
RStudio è un ambiente di sviluppo integrato (IDE) per R, un linguaggio di programmazione per il calcolo statistico e la grafica. È disponibile in due formati: RStudio Desktop è una normale applicazione desktop mentre RStudio Server viene eseguito su un server remoto e consente di accedere a RStudio utilizzando un browser Web.



# PANNELLI & BARRA DEGLI STRUMENTI

```
dens <- density(data, n = numps)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(., , 0
  main
  seqbelow <- r
  if(Fill == T)
    confshadeC
```

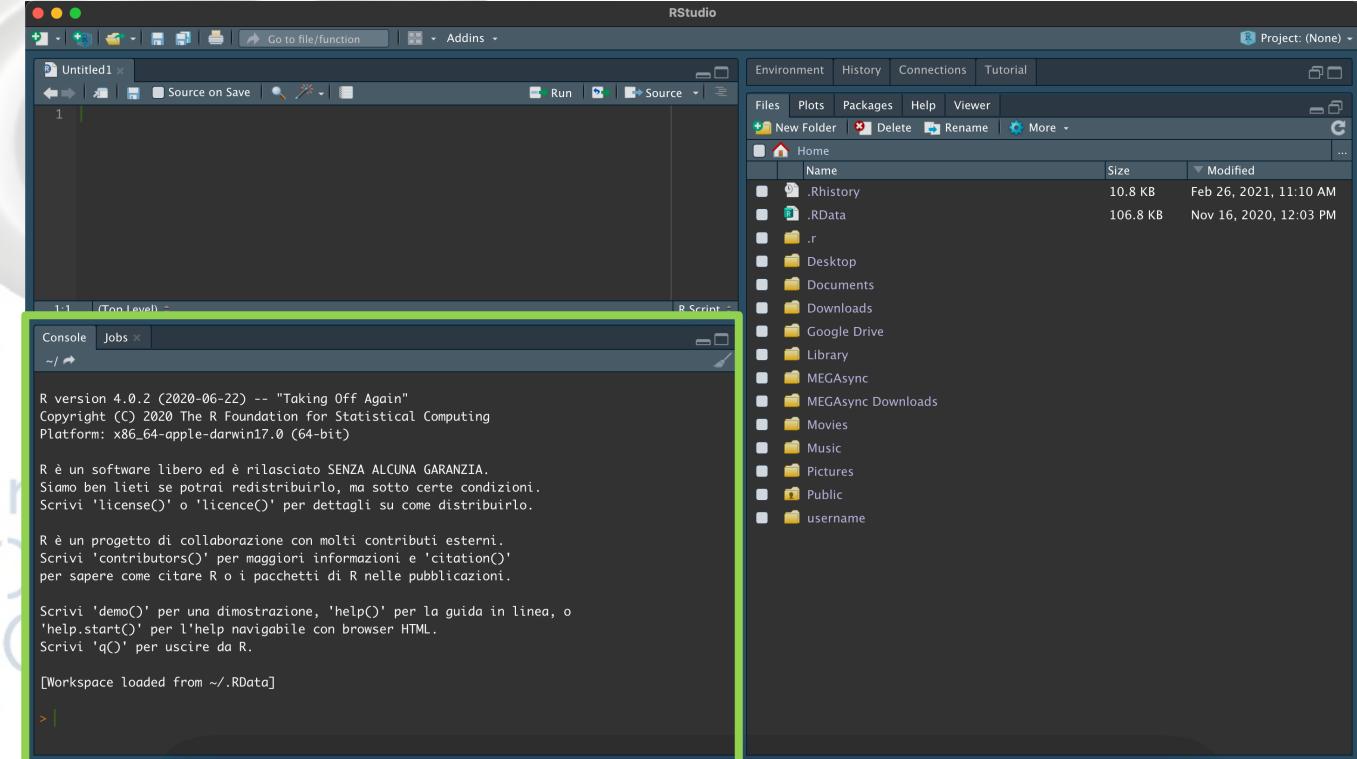
L'editor degli scripts si trova nella parte superiore sinistra dell'ambiente RStudio. Se stai avviando una nuova sessione R o hai chiuso l'ultimo script R aperto, l'editor di script non è visibile. È possibile aprire l'editor di script creando un nuovo script vuoto o aprendo uno script esistente.



# PANNELLI & BARRA DEGLI STRUMENTI

```
dens <- density(data, n = nptcs)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(., , 0, 0, 1, 1,
       xlab = "X", ylab = "Y", main = "main",
       xaxt = "none", yaxt = "none",
       type = "n")
  points(x = dx, y = dy, col = "blue", pch = 19, cex = 1.5)
  if(fill == TRUE) {
    polygon(dx, dy, col = "#0072BD", border = "#0072BD")
  }
}
```

La **console dei comandi** è il luogo in cui R esegue i comandi e di solito risponde a tali comandi. L'unica eccezione è quando R disegna un grafico. La console di comando si trova nella parte inferiore sinistra dell'ambiente.



# PANNELLI & BARRA DEGLI STRUMENTI

```
dens <- density(data, n = npts)
```

```
dx <- dens$x
```

```
dy <- dens$y
```

```
if(add == TRUE)
```

```
plot0., , 0
```

```
TRientati
```

```
dx <- dx[
```

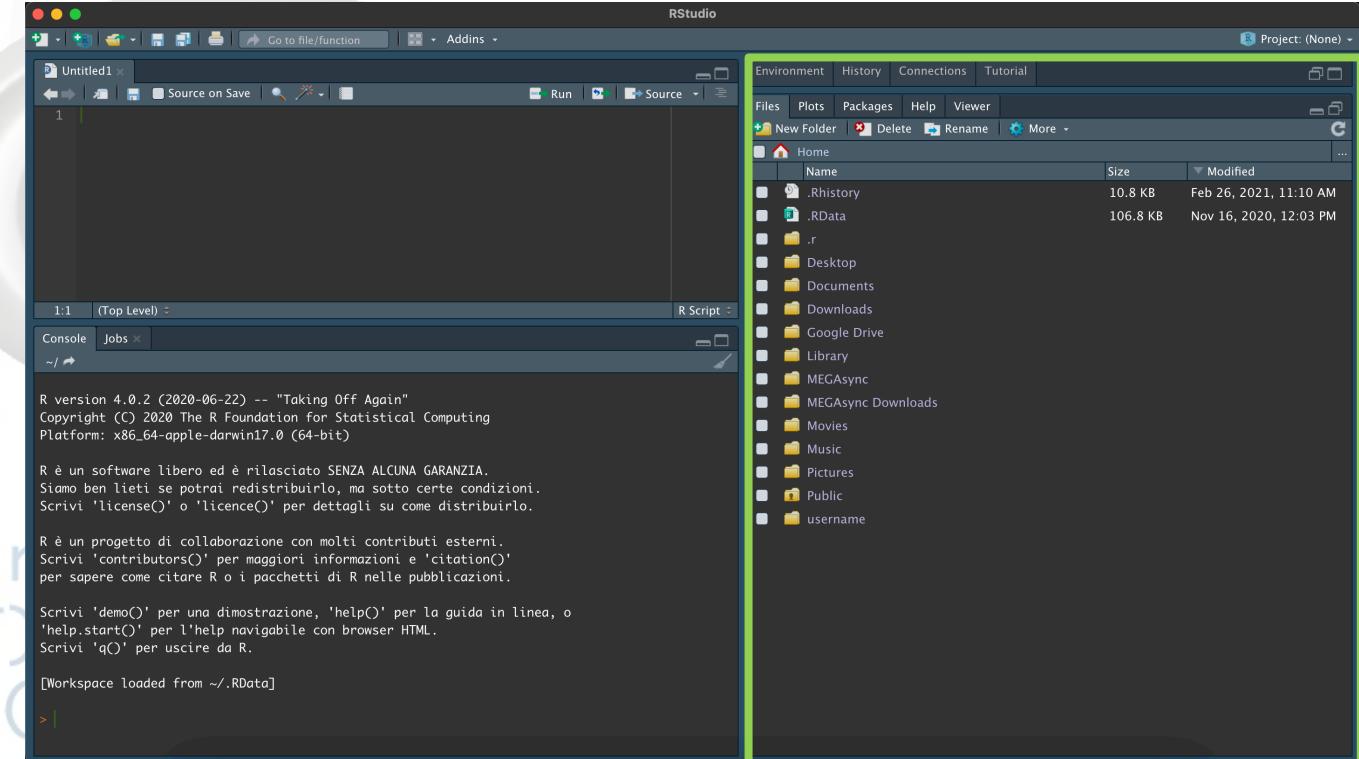
```
y[1.]
```

```
seqbelow <-
```

```
if(Fill == T
```

```
confshade0
```

Nel pannello delle utilità, puoi trovare l'area del tracciato, la directory, la cronologia e il pannello dell'ambiente.



# PACCHETTI IN R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0, type = "n", xlab = "x", ylab = "y", main = "multi")
  if(orientati
    dx2 <- dx
    dy2 <- -m * dx + b
    x[1.] <- seqbelow
    y[1.] <- m * seqbelow + b
    seqbelow <- seqbelow + 1
    if(Fill == T)
      confshade(dx2, seqbelow, dy2)
```

- I **pacchetti** sono raccolte di funzioni e set di dati sviluppati dalla comunità. Aumentano la potenza di R migliorando le funzionalità di base esistenti o aggiungendone di nuove.
- Un pacchetto include codice, documentazione per il pacchetto e le funzioni al suo interno, alcuni test per verificare che tutte le funzioni come dovrebbe e set di dati.
- Le informazioni di base su un pacchetto sono fornite nel file **DESCRIPTION**, dove puoi scoprire cosa fa il pacchetto, chi è l'autore, a quale versione appartiene la documentazione, la data, il tipo di licenza d'uso e le dipendenze del pacchetto.



# PACCHETTI IN R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., type = "n", xlab = "x", ylab = "y")
  dx2 <- c(dx, 0)
  x[1.] <- NA
  dy2 <- c(dy, 0)
  y[1.] <- NA
  seqbelow <- rep(y[1.], length(dx))
  if(Fill == T)
    confshade(dx2, seqbelow, dy2)
```

```
Package: stats
Version: 4.0.2
Priority: base
Title: The R Stats Package
Author: R Core Team and contributors worldwide
Maintainer: R Core Team <R-core@r-project.org>
Description: R statistical functions.
License: Part of R 4.0.2
Imports: utils, grDevices, graphics
Suggests: MASS, Matrix, SuppDists, methods, stats4
NeedsCompilation: yes
Built: R 4.0.2; x86_64-apple-darwin17.0; 2020-06-23 23:43:34 UTC; unix
```



# PACCHETTI IN R

```
dens <- density(data)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0,
       main = main,
       tcl = tcl,
       xlab = xlab,
       ylab = ylab)
  if(orientati)
    dx2 <- dx
    y[1.] <- seqbelow
    seqbelow <- rep(c(y[1.], 0), length(dx))
    if(Fill == T)
      confshade(dx2, seqbelow, dy2)
```

I pacchetti possono essere trovati in repository ad-hoc

- **CRAN**: il repository ufficiale, è una rete di server ftp e web gestiti dalla comunità R in tutto il mondo.
- **Bioconductor**: si tratta di un repository specifico per argomento, destinato a software open source per la bioinformatica.
- **Github** : sebbene non sia specifico per R, Github è probabilmente il repository più popolare per i progetti open source.

I pacchetti possono essere installati utilizzando il pannello della console e la sintassi

```
install.packages("package.name")
```

oppure utilizzando la barra degli strumenti **packages** nel pannello delle utilità

# PACCHETTI IN R

```
dens <- density(data[, n])
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., type = "n", xlab = "x", ylab = "y")
  dx2 <- c(dx, x[1])
  dy2 <- c(dy, y[1])
  seqbelow <- 0
  if(Fill == TRUE)
    confshade(dx2, seqbelow, dy2)
```

```
Installing package into '/home/username/R/x86_64-pc-linux-gnu-library/3.3' (as 'lib' is unspecified)

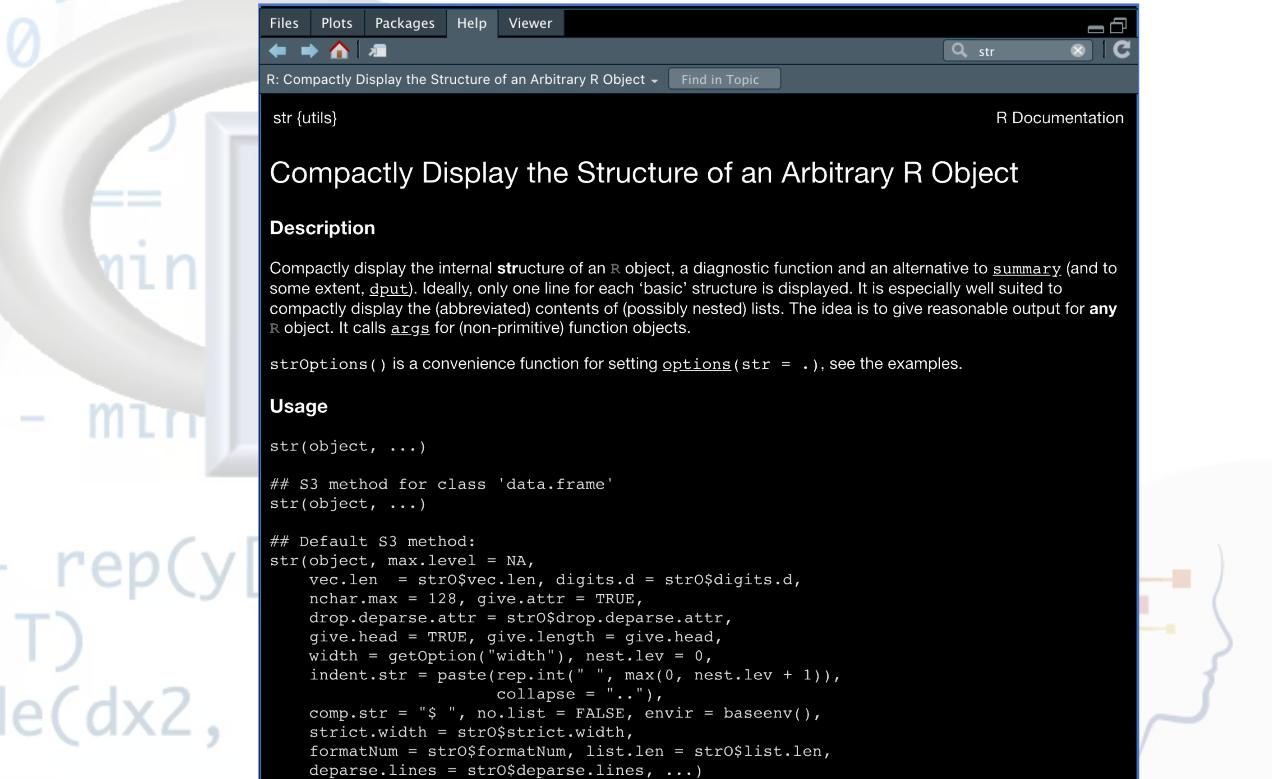
trying URL 'https://cran.rstudio.com/src/contrib/vioplot_0.2.tar.gz' Content type 'application/x-gzip' length 3801 bytes
=====
== downloaded 3801 bytes

* installing *source* package 'vioplot' ... **
R ** preparing package for lazy loading ** help
*** installing help indices ** building package
indices ** testing if installed package can be
loaded * DONE (vioplot)
```

# UTILIZZO DELL'HELP

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., type = "n",
       xlab = xlab,
       ylab = ylab,
       xlim = c(min(dx), max(dx)),
       ylim = c(min(dy), max(dy)))
else
  if(dx[1] <= min(dx))
    dy2 <- (dx - min(dx)) * seqbelow + y[1.]
  else
    dy2 <- rep(y, times = length(dx) - 1)
  if(Fill == T)
    confshade(dx2, dy2, col = col, lty = lty,
              lwd = lwd, border = border)
```

Il pannello della guida ha una propria barra degli strumenti. Questa barra degli strumenti contiene due pulsanti. *Home* [questo ti riporterà alla home page della guida e cancellerà la cronologia delle ricerche della guida] e *find* nella guida.



Files Plots Packages Help Viewer

R: Compactly Display the Structure of an Arbitrary R Object Find in Topic

str {utils}

Compactly Display the Structure of an Arbitrary R Object

**Description**

Compactly display the internal **structure** of an R object, a diagnostic function and an alternative to **summary** (and to some extent, **dput**). Ideally, only one line for each 'basic' structure is displayed. It is especially well suited to compactly display the (abbreviated) contents of (possibly nested) lists. The idea is to give reasonable output for **any** R object. It calls **args** for (non-primitive) function objects.

**strOptions()** is a convenience function for setting **options(str = .)**, see the examples.

**Usage**

```
str(object, ...)

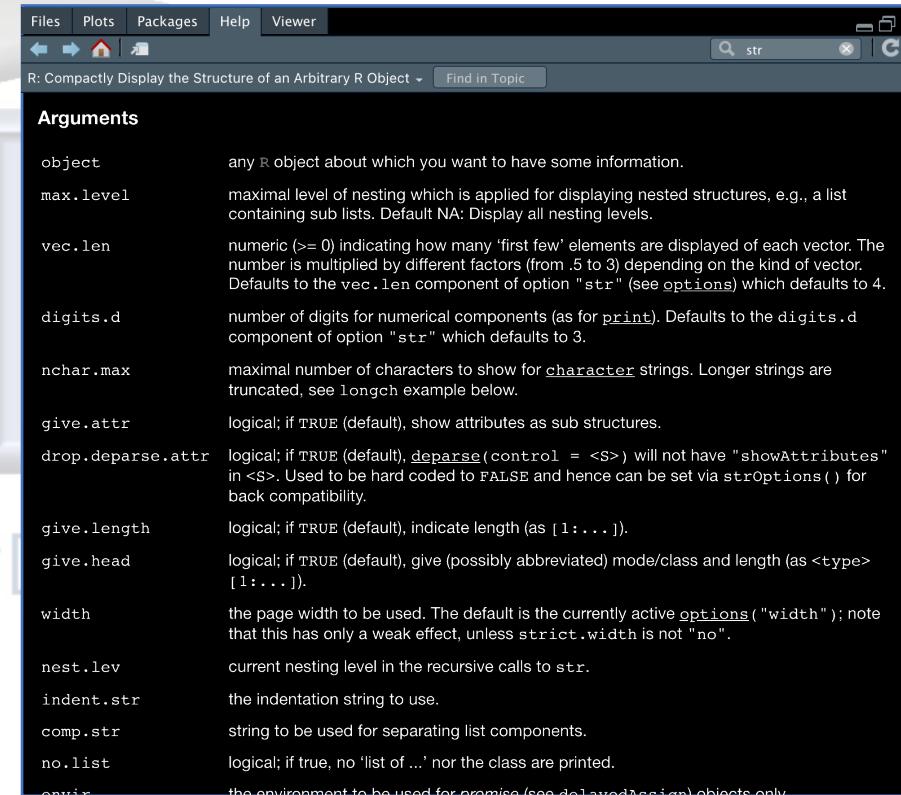
## S3 method for class 'data.frame'
str(object, ...)

## Default S3 method:
str(object, max.level = NA,
  vec.len = str0$vec.len, digits.d = str0$digits.d,
  nchar.max = 128, give.attr = TRUE,
  drop.deparse.attr = str0$drop.deparse.attr,
  give.head = TRUE, give.length = give.head,
  width = getOption("width"), nest.lev = 0,
  indent.str = paste(rep.int(" ", max(0, nest.lev + 1)),
    collapse = "..."),
  comp.str = "$ ", no.list = FALSE, envir = baseenv(),
  strict.width = str0$strict.width,
  formatNum = str0$formatNum, list.len = str0$list.len,
  deparse.lines = str0$deparse.lines, ...)
```

# UTILIZZO DELL'HELP

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., xlab = x[1.], ylab = y[1.])
  dy2 <- (dx - min(dx)) * seqbelow
  if(Fill == T)
    confshade(dx2,
```

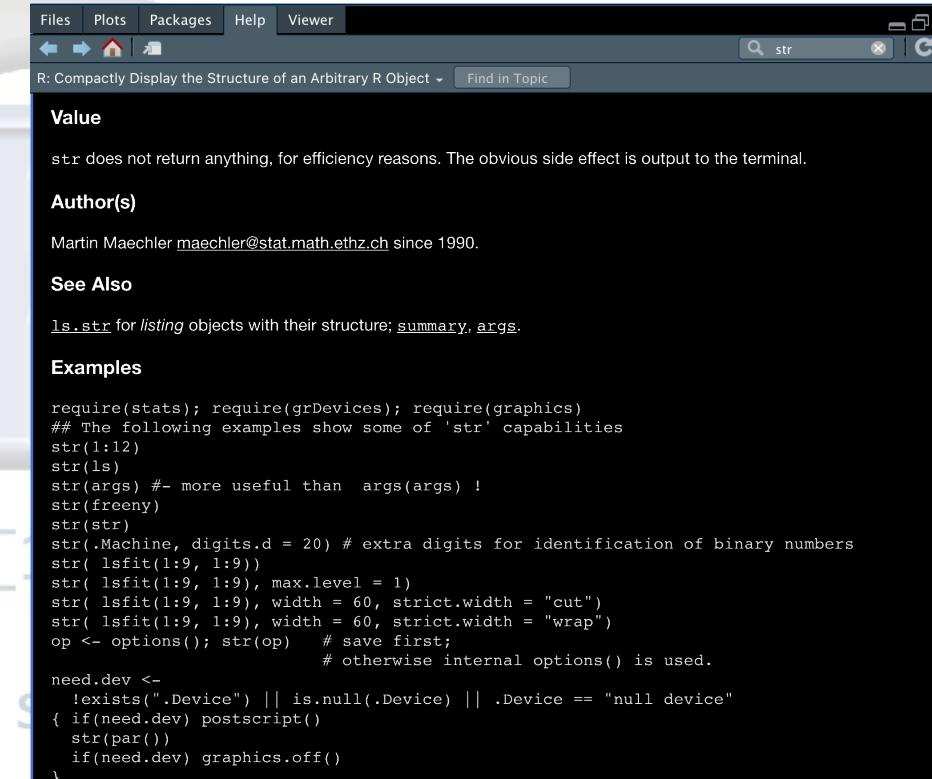
Il pannello della guida ha una propria barra degli strumenti. Questa barra degli strumenti contiene due pulsanti. *Home* [questo ti riporterà alla home page della guida e cancellerà la cronologia delle ricerche della guida] e *find* nella guida.



# UTILIZZO DELL'HELP

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., xlab = x[1.], ylab = y[1.])
  dy2 <- (dx - min(dx)) * seqbelow
  if(Fill == T)
    confshade(dx2, s
```

Il pannello della guida ha una propria barra degli strumenti. Questa barra degli strumenti contiene due pulsanti. *Home* [questo ti riporterà alla home page della guida e cancellerà la cronologia delle ricerche della guida] e *find* nella guida.



# STRUTTURA DEI DATI & ACQUISIZIONE

Tipi di dati in lettura

- Testo normale e file delimitati (.txt, .csv, .tsv)
- File Microsoft Word ed Excel (.doc, .docx, .xls, .xlsx)
- Database (SQLite, MySQL, Microsoft Access)
- Dati software-specifico (SAS, SPSS, NCSS, Octave)

Fonte dei dati in lettura

- in R e nei pacchetti di R
- sulla memoria del tuo computer
- su una rete locale o collegata
- ovunque su Internet



# STRUTTURA DEI DATI & ACQUISIZIONE

- **Omogeneità:** indica se la struttura dei dati è "omogenea", ovvero contiene solo tipi di dati simili, ad esempio tutti numerici, tutte stringhe, ecc. o una combinazione di più tipi di dati, ovvero "eterogenei".
- **Dimensione:** ci dice in che modo verranno archiviati i dati dell'ordine, se lineare (1D), tabulare (2D) e così via.

	1D	2D	Multi-D
Omogenei	Vettore	Matrice	Array
Eterogenei	Lista	Dataframe	

# STRUTTURA DEI DATI & ACQUISIZIONE

- **Vettori:** raccolte di elementi dello stesso tipo
- **Matrici:** contenitori rettangolari di elementi dello stesso tipo
- **Dataframe:** contengono molti tipi di vettori, tutti della stessa lunghezza
- **Array:** Vettori con dimensioni per ogni elemento dello stesso tipo
- **List:** contenitori per elementi di tipi di dati multi-tipo



```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(dx >= TRUE) {
  plot(0, 0, main = "Density Plot", xlab = "x", ylab = "y")
  points(dx, dy)
}
if(orientati
  dx2 <- c(dx, rep(dx[1], length(dx) - 1)))
  - mean(dx2), dx(dy)
  seqbelow <- rep(y[1.], length(dx))
  if(Fill == T)
    confshade(dx2, seqbelow, dy2)
```

# STRUTTURA DEI DATI & ACQUISIZIONE

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(orientat... >= 0
  plot(dx, dy, main = "Dataframe", xlab = "X", ylab = "Y")
else
  plot(dy, dx, main = "Dataframe", xlab = "Y", ylab = "X")
if(Fill == T)
  confshade(dx2, seqbelow, dy2)
```

I dati possono essere importati usando la generica funzione **read()**

È il modo più semplice per importare file di dati delimitati e il risultato è un **Dataframe**.

La funzione **read()** richiede almeno 3 argomenti:

- Il percorso e il nome file o l'URL del file di dati
- Se la prima riga contiene le etichette delle intestazioni di colonna

impostato su TRUE se e solo se la prima riga contiene un campo in meno rispetto al numero di colonne

- Il separatore tra gli elementi di dati



# STRUTTURA DEI DATI & ACQUISIZIONE

# TTUR DATI & JISIZIONE

Importare i dati usando la funzione `read.table`

```
read.table("<FileName>.txt",  
          header = TRUE,  
          sep = ",", dec = ".")
```

Importare i dati usando la funzione `read.csv`

```
read.csv("<FileName>.csv",  
        header = TRUE,  
        sep = ",", dec = ".")
```

Importare i dati da un file Excel usando la funzione `read_excel`

```
read_excel("<FileName>.xlsx",  
          sheet_name)
```

```
<- density(data, n = npts)  
<- dens$x  
<- dens$y  
(add = TRUE)  
plot(0, 0  
main  
read.table("<FileName>.txt",  
          header = TRUE,  
          sep = ",", dec = ".")  
read.csv("<FileName>.csv",  
        header = TRUE,  
        sep = ",", dec = ".")  
- main  
read_excel("<FileName>.xlsx",  
          sheet_name)  
seqbelow <- rep(y[1.], length(d  
if(Fill == T)  
  confshade(dx2, seqbelow, dy2
```

## Importare I dati usando la funzione `read()`

```
main  
read.table("<FileName>.txt", header  
=TRUE, sep=", ", dec=".")  
  
main  
read.csv("<FileName>.csv", header  
=TRUE, sep=", ", dec=".")  
  
main  
read_excel("<FileName>.xlsx", sheet = "  
SheetName")
```

# TIPI DI VARIABILI

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0, xlab = " ", ylab = " ")
  if(orient == "vertical")
    for(i in 1:npts) {
      dx2 <- c(dx[i], seqbelow)
      dy2 <- c(dy[i], rep(y[1.], length(dx2)))
      segments(dx, dy, dx2, dy2)
      if(Fill == T)
        confshade(dx2, seqbelow, dy2)
    }
  else
    for(i in 1:npts) {
      dx2 <- c(seqbelow, dx[i])
      dy2 <- c(rep(y[1.], length(dx)), dy[i])
      segments(dx, dy, dx2, dy2)
      if(Fill == T)
        confshade(dx2, seqbelow, dy2)
    }
}
else
  plot(dx, dy)
```

Le variabili più comuni utilizzate nell'analisi dei dati possono essere classificate come uno dei tre tipi di variabili: nominale, ordinale e intervallo/rapporto.

Comprendere le differenze in questi tipi di variabili è fondamentale, poiché il tipo di variabile determinerà quale analisi statistica sarà valida per quei dati.

Inoltre, il modo in cui riassumiamo i dati con statistiche e grafici sarà determinato dal tipo di variabile.



# TIPI DI VARIABILI

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0., 0., main = "Variabili Categoriali")
  if(orient == "vertical") {
    dx2 <- (dx - min(dx)) / (length(dx) - 1)
    y[1.] <- seqbelow
    if(Fill == TRUE) {
      confshade(dx2, seqbelow, dy2)
    }
  } else {
    dy2 <- (dy - min(dy)) / (length(dy) - 1)
    x[1.] <- seqbelow
    if(Fill == TRUE) {
      confshade(x, seqbelow, dy2)
    }
  }
}
```

## Dati nominali

Le variabili nominali sono dati i cui livelli sono etichette o descrizioni e che non possono essere ordinati. Esempi di variabili nominali sono sesso, scuola e domande sì/no. Sono anche chiamate variabili "categorical nominali" o "qualitative" e i livelli di una variabile sono talvolta chiamati "classi" o "gruppi".

I livelli delle variabili categoriali non possono essere ordinati. Per la variabile sesso, non ha senso cercare di mettere i livelli "femmina", "maschio" e "altro" in un qualsiasi ordine numerico.

Se i livelli sono numerati per comodità, i numeri sono arbitrari e la variabile non può essere trattata come una variabile numerica.

# TIPI DI VARIABILI

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., main = "Suggerimenti per la scelta delle variabili",
       xlab = "dx", ylab = "dy")
  if(orient == "vertical")
    for(i in 1:npts) {
      dx2 <- (dx[i] - seqbelow[i])
      dy2 <- (dy[i] - seqbelow[i])
      segments(dx[i], dy[i], dx2, dy2)
      points(dx2, dy2, col = "black", pch = 15, cex = 1.5)
    }
    if(Fill == T)
      confshade(dx2, seqbelow, dy2)
  else
    for(i in 1:npts) {
      dx2 <- (dx[i] - seqbelow[i])
      dy2 <- (dy[i] - seqbelow[i])
      segments(dx[i], dy[i], dx2, dy2)
      points(dx[i], dy[i], col = "black", pch = 15, cex = 1.5)
      points(dx2, dy2, col = "black", pch = 15, cex = 1.5)
    }
    if(Fill == T)
      confshade(dx2, seqbelow, dy2)
  }
  if(Fill == T)
    confshade(dx, seqbelow, dy)
}
```

## Dati ordinali

Le variabili ordinali possono essere ordinate o classificate in ordine logico, ma l'intervallo tra i livelli delle variabili non è necessariamente noto.

Le misurazioni soggettive sono spesso variabili ordinali.

Un esempio sarebbe chiedere alle persone di classificare quattro elementi in ordine di preferenza da uno a quattro.



# TIPI DI VARIABILI

# TIPI DI VARIABILI

**Dati di intervallo/raggruppamento**

Le variabili di intervallo/raggruppamento contati: età, altezza, numero di bambini.

I dati di intervallo/raggruppamento sono "quantitativi".

Un'ulteriore divisione dei dati in variabili discrete, i cui valori sono interi o altri valori di conteggio di elementi.

Le variabili continue sono quelle che si estendono all'interno di un intervallo e sono espresse come decimali.

## Dati di intervallo/rapporto

Le variabili di intervallo/rapporto sono valori misurati o contati: età, altezza, peso, numero di studenti.

I dati di intervallo/rapporto sono anche chiamati dati "quantitativi".

Un'ulteriore divisione dei dati di intervallo/rapporto è tra variabili discrete, i cui valori sono necessariamente numeri interi o altri valori discreti, come popolazione o conteggi di elementi.

Le variabili continue possono assumere qualsiasi valore all'interno di un intervallo e quindi possono essere espresse come decimali. Spesso sono quantità misurate.

# TIPI DI VARIABILI

```
dens <- density(data, n = npts)
```

```
dx <- dens$x
```

```
dy <- dens$y
```

```
if(add == TR
```

R **non usa** i termini nominale, ordinale e intervallo/rapporto per i tipi di variabili.

In R, le variabili nominali possono essere codificate come variabili con classi di fattori o caratteri.

I dati di intervallo/rapporto possono essere codificati come variabili con classi numeriche o intere.

Una **L** viene usata con i valori per dire a R di archiviare i dati come una classe intera.

```
dy2 <- (dx - min(dx)) / max(dy)
y[1.] = BugCount.int = c(1L, 2L, 3L, 4L, 5L)
seqbelow <- rep(y[1.], length(dx))
if(Fill == T) [1] "integer"
confshade(dx2, seqbelow, dy2)
```



# PRIMI PASSI CON R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0, 0, xlab = "x", ylab = "y", main = "Density Plot", type = "n")
  if(is.null(dx)) dx <- seq(0, 1, length = npts)
  if(is.null(dy)) dy <- density(data, n = npts)$y
  points(dx, dy)
  if(!is.null(fill)) {
    if(fill == TRUE) {
      polygon(dx, dy, col = fill)
    } else {
      polygon(dx, dy, col = fill, border = border)
    }
  }
  if(is.function(conf)) {
    confshade(dx2, seqbelow, dy2)
  }
}
```

Come abbiamo già visto è possibile usare R per svolgere operazioni matematiche, usando i simboli standard.

Ad esempio, l'uso di `+` più tra due numeri farà comparire la loro somma. In modo analogo è possibile farne la differenza `(-)`, il prodotto `(*)` e la divisione `(/)`.

```
2+2
## [1] 4
2-2
## [1] 0
2*2
## [1] 4
2/2
## [1] 1
```

Le potenze si possono esprimere usando il simbolo `^`. Lo stesso simbolo si può utilizzare per il calcolo delle radici, anche se è possibile richiamare la radice quadrata usando il comando `sqrt()`. Si noti che tale funzione accetta anche valori complessi.

Per la divisione intera tra due numeri è possibile usare il simbolo `%/%`, mentre `%%` restituisce il resto di tale divisione.

Un altro comando di base che può risultare utile è il calcolo del valore assoluto ottenuto con il comando `abs()`.



# PRIMI PASSI CON R

# IMI PASSI CON R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0., 1., xlab = "x", ylab = "y")
  lines(dx, dy)
}
if(!orientatti) {
  dx2 <- c(dx[1], dx[-1])
  dy2 <- c(dy[-1], dy[1])
} else {
  dx2 <- c(dx[-1], dx[1])
  dy2 <- c(dy[1], dy[-1])
}
seqbelow <- seq(0, 1, length.out = length(dx2))
if(Fill == 1) {
  confshade(dx2, seqbelow, dy2, col = "#D9E1F2", border = "#A9B9D7")
  if(!orientatti) {
    lines(dx2, seqbelow)
  } else {
    lines(seqbelow, dy2)
  }
}
if(Fill == 2) {
  confshade(dx2, seqbelow, dy2, col = "#D9E1F2", border = "#A9B9D7)
  if(!orientatti) {
    lines(dx2, seqbelow)
  } else {
    lines(seqbelow, dy2)
  }
}
```

## 2.2 Esponenziale e logaritmo

E' possibile calcolare i valori della funzione esponenziale e logaritmo. E' possibile usare più funzioni: `log()`, `lg()` e `exp()` rispettivamente. Il comando `log()` permette di calcolare il logaritmo naturale. Ad esempio, `log( 5 , base = 3 )` restituisce il logaritmo base 3 del numero 5.

## 2.3 Funzioni Trigonometriche

Le principali funzioni trigonometriche sono `sin()`, `cos()` e `tan()`. Sono disponibili anche le loro inverse: `asin()`, `acos()` e `atan()`. I parametri sono in gradi.

```
sin() # seno
cos() # cosen
tan() # tangente
```

E' possibile utilizzare le stesse funzioni ma con parametri in radiani. Per farlo è sufficiente moltiplicare i valori per  $\pi/180$ . Per esempio, `sin(pi/4)` restituisce il seno di un angolo di 45 gradi.

Anche le funzioni inverse sono già implementate: `asin()`, `acos()` e `atan()`.

## 2.2 Esponenziale e logaritmi

E' possibile calcolare i valori della funzione esponenziale usando il comando `exp()` mentre per il calcolo dei logaritmi è possibile usare più funzioni: `log()`, `log10()` e `log2()` che restituiscono il logaritmo naturale, base 10 e base 2 rispettivamente. Il comando `log()` permette anche di specificare una base differente da `e`, che è quella di default. Ad esempio, `log( 5 , base = 3)` restituisce il logaritmo in base 3 di 5.

## 2.3 Funzioni Trigonometriche

Le principali funzioni trigonometriche sono già implementate in R e si possono richiamare usando i comandi elencati sotto, i cui nomi richiamano le funzioni stesse. Va notato che l'input di queste funzioni è atteso in radianti, **non gradi**.

```
sin() # seno  
cos() # cosen  
tan() # tangente
```

E' possibile utilizzare le stesse funzioni ma con desinenza ***pi*** (***sinpi()***, ***cospipi()*** e ***tanpi()***) se si intende considerare dei multipli di pi-greco.

Anche le funzioni inverse sono già implementate e si possono richiamare con i comandi `acos()`, `asin()`, `atan()`.



# PRIMI PASSI CON R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0, 0, xlab = "x", ylab = "y", type = "n")
  if(is.null(dx)) dx <- c(0, 1)
  if(is.null(dy)) dy <- c(0, 1)
  for(i in 1:(length(dx)-1)) {
    x1 <- dx[i]
    x2 <- dx[i+1]
    y1 <- dy[i]
    y2 <- dy[i+1]
    seqbelow <- rep(y1, length(dx))
    if(Fill == T)
      confshade(dx2, seqbelow, dy2)
```

## 2.4 Assegnazione e vettori

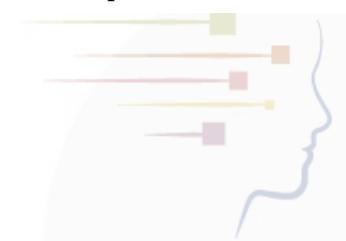
Fin ora abbiamo usato R per calcolare dei singoli valori, in modo analogo a come avremmo usato una calcolatrice. Tuttavia R permette di fare molto di più.

Come primo passo per scoprirne le potenzialità vedremo la possibilità di assegnare valori e di richiamarli successivamente, così come di lavorare con vettori o matrici, invece che con singoli valori.

E' possibile assegnare un valore ad una variabile usando la freccina `->`. Stesso risultato si può ottenere usando `=` oppure `assign()`, tuttavia il primo è decisamente il comando più usato e di più facile lettura.

```
x <- 6
```

Usando il precedente comando viene assegnato il valore 6 alla variabile `x`. In questo modo è possibile richiamarla successivamente, così come è possibile scrivere espressioni più complesse che coinvolgono `x` a prescindere dal suo valore. Questo sarà particolarmente utile se dovremo definire delle funzioni a cui è possibile passare per argomento diversi valori. Si noti che il comando `6 -> x` produce gli stessi risultati del precedente.



# PRIMI PASSI CON R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE)
  plot(0., 0., type = "n")
  abline(v = dx, col = "black", lwd = 1)
  if(is.null(dx2) || is.null(dy2))
    dy2 <- (dx - min(dx)) * 0.05
  else dy2 <- rep(dy, length(dx))
  if(Fill == T)
    confshade(dx2, seqbelow, dy2)
```

Oltre ad assegnare un singolo valore, è possibile considerare dei vettori. Il comando

```
y <- c(1,2,3,4)
y[1]
## [1] 1
```

assegna a y i valori contenuti nel comando `c()`. Tale comando combina i valori in un vettore colonna (ma viene visualizzato come riga). L'accesso ai vettori avviene attraverso indicandone il nome, seguito da parentesi quadre. Ovviamente è possibile svolgere operazioni aritmetiche anche tra vettori, usando la stessa sintassi vista prima. Di default R considera operazioni puntuali tra vettori, cioè l'operazione richiesta viene svolta entrata per entrata. Questo richiede le dimensioni siano le stesse. Qualora non lo fossero, il contenuto del vettore più piccolo viene ripetuto un numero di volte sufficienti da rendere possibile l'operazione. Questo viene anche segnalato da un warning.



# PRIMI PASSI CON R

Un comando utile tra vettori è `t()` che traspone il vettore.

```
z <- t(y)
y

## [1] 1 2 3 4
print(z)

##      [,1] [,2] [,3] [,4]
## [1,]     1     2     3     4
z

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

Chiamare una variabile senza indicare operazioni ne parentesi tonde si ottiene lo stesso risultato. Anche il comando, diversamente dai precedenti, permette di moso uno script.

Dal comando precedente vediamo che `y` e `z` hanno svolgere operazioni matematiche su di essi senza warning.

Un comando utile tra vettori è `t()` che traspone il vettore passato per argomento.

```
z <- t(y)
y

## [1] 1 2 3 4
print(z)

##      [,1] [,2] [,3] [,4]
## [1,]     1     2     3
z

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

Chiamare una variabile senza indicare operazioni ne permette la stampa a video. Inserendo l'assegnazione tra parentesi tonde si ottiene lo stesso risultato. Anche il comando `print()` permette la stampa a video. Inoltre questo comando, diversamente dai precedenti, permette di mostrare l'output anche quando è chiamata dentro una funzione o uno script.

Dal comando precedente vediamo che `y` e `z` hanno stessi valori ma dimensioni diverse. Tuttavia R permette di svolgere operazioni matematiche su di essi senza warning o errori.



# PRIMI PASSI CON R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0, 0, xlab = "x", ylab = "y", type = "n")
  if(is.null(dx)) dx <- seq(0, 1, length = length(dy))
  if(is.null(dy)) dy <- seq(0, 1, length = length(dx))
  if(is.null(dx2)) dx2 <- dx
  if(is.null(dy2)) dy2 <- dy
  if(is.null(seqbelow)) seqbelow <- rep(dy[1], length(dx))
  if(Fill == T) {
    confshade(dx2, seqbelow, dy2)
  } else {
    lines(dx2, seqbelow)
  }
}
```

Per svolgere operazioni tra vettori, come il prodotto righe per colonne si può considerare il comando `%%*`. In questo caso il risultato dipende dall'ordine.

```
y %%*% z
##      [,1] [,2] [,3] [,4]
## [1,]     1     2     3     4
## [2,]     2     4     6     8
## [3,]     3     6     9    12
## [4,]     4     8    12    16
z %%*% y
##      [,1]
## [1,]    30
```



# PRIMI PASSI CON R

```
dens <- density(data, n = npts)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0, 0, type = "n", xlab = "x", ylab = "y")
  if(!is.null(dx)) abline(v = dx, col = "black", lwd = 1)
  if(!is.null(dy)) abline(h = dy, col = "black", lwd = 1)
}
if(Fill == TRUE) {
  dx2 <- c(dx[1], dx[-1])
  y1 <- y[1]
  seqbelow <- rep(y1, length(dx))
  if(Fill == TRUE) {
    confshade(dx2, seqbelow, dy2)
  }
}
```

## 2.5 Sequenze

Nel caso si debba indicare una successione di valori è possibile farlo senza indicarli espressamente tutti, purché essi siano una sequenza regolare.

Ad esempio con `a:b` possiamo usare i due punti per indicare una sequenza di valori con passo 1 da  $a$  fino a  $b$ . Per indicare un passo diverso possiamo usare il comando `seq(a,b,passo)` che produrrà una sequenza di elementi da  $a$  fino a  $b$  distanziati di un valore uguale al *passo* indicato. Se il passo viene omesso è considerato 1 di default.

Per tutte le funzioni è possibile accedere all'help di R usando il comando `help(Nome_funzione)`. Leggendo l'help di `seq()` vediamo ad esempio che possiamo anche indicare in ordine diverso i parametri, a patto di specificarli usando il nome indicato nell'help. Ad esempio `seq(to = b, by = passo, from = a)` restituisce lo stesso output di `seq(a,b,passo)`.

Inoltre è possibile ripetere un valore o un vettore usando il comando `rep()`, ad esempio `rep(1,5)` restituisce cinque volte il valore 1. Nel caso di un vettore, il comando ripeterà 5 volte il vettore.





# PRIMI PASSI CON R

```
dens <- density()
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0, 0, xlab = "x", ylab = "y", main = "Density Plot")
  if(content > 0) {
    dx2 <- c(dx[1], dx)
    dy2 <- c(dy[1], dy)
    seqbelow <- seq(0, content, by = 1)
    if(Fill == TRUE) {
      confshad <- seqbelow * dy2
      for(i in 1:(length(seqbelow) - 1)) {
        if(dx2[i] <= seqbelow[i + 1]) {
          confshad[i] <- confshad[i] + dy2[i + 1]
        }
      }
      confshad <- c(confshad, dy2[length(dy2)])
      lines(dx2, confshad, type = "l", lwd = 2)
    } else {
      lines(dx2, dy2, type = "l", lwd = 2)
    }
  }
}
```

## 2.6 Operazioni su vettori

Oltre alle operazioni che abbiamo già visto esistono altre funzioni di R appositamente pensate per i vettori. Ad esempio le funzioni `min` e `max` restituiscono, rispettivamente, il minimo ed il massimo valore contenuto in un vettore. Per accedere alla posizione di tali valori si combinano i precedenti comandi con `which`. Ad esempio

```
x <- c(3, 5, 7, 1, 3, 3, 9, 8)
min(x)

## [1] 1
which.min(x)

## [1] 4
max(x)

## [1] 9
which.max(x)

## [1] 7
```

Altre funzioni molto utili per lavorare con i vettori sono la funzione `sum()` che calcola la somma di tutti gli elementi di un vettore e la funzione `diff()` che calcola la differenza di un valore da uno dei precedenti (è possibile indicare quanto prima “guardare”).

```
sum(x)

## [1] 39
diff(x)

## [1] 2 2 -6 2 0 6 -1
diff(x, 2)

## [1] 4 -4 -4 2 6 5
```

# PRIMI PASSI CON R

```
dens <- density()
dx <- dens$x
dy <- dens$y
if(add == TR...
plot(0, ...
infcontenti...
dx2 <- c(dx...
x[1]
y[1]
seqbelow <...
if(Fill == ...
confshac...
```

## 2.7 Operatori relazionali e logici

R ci permette anche di valutare espressioni relazionali o logiche. Il loro risultato sarà un valore logico indicato con TRUE o FALSE.

```
6 > 10
## [1] FALSE
6 <= 10
## [1] TRUE
is_bigger <- 6 > 10
is_bigger
## [1] FALSE
as.integer(is_bigger)
## [1] 0
```

R permette di valutare diverse espressioni relazionali, oltre a maggiore (uguale) o minore (uguale). Ad esempio è possibile valutare se due valori o variabili siano diversi != o uguali ==. Questo può essere particolarmente utile quando si definiscono delle funzioni proprie e si valuta se una condizione è soddisfatta.

R permette anche di valutare gli operatori logici & (and), | (or), xor e ! (not).

Questi operatori possono essere valutati sia su vettori logici, che di numeri qualsiasi. Nel secondo caso, tuttavia, tutto ciò che è diverso da zero conterà come TRUE, e solo lo 0 conterà come FALSE. Nel caso in cui si definiscano dei vettori misti (dove sono presenti sia valori logici che interi/reali/complessi), verranno tutti convertiti nel formato numerico presente.

# PRIMI PASSI CON R

# IMI PASSI CON R

## 2.8 Matrici

Come abbiamo visto i vettori sono considerati in generale vettori riga, e non è possibile generare una matrice usando il comando `c()`.

Tuttavia le matrici esistono e si possono definire usando la funzione `cbind()` che unisce i vettori passati come argomento in una matrice dove i vettori argomento sono le colonne. La funzione `rbind()` ha la stessa funzione, ma i vettori passati come argomento saranno le righe della matrice. In tutti e due i casi, i vettori passati come argomento devono avere stessa dimensione (no uno riga e uno colonna). La funzione `dim()` restituisce le dimensioni dell'oggetto passato come argomento. La funzione `length()` (che avevamo visto con i vettori) ci restituisce il *prodotto* delle dimensioni.

```
a <- cbind(c(1, 2, 3), c(4, 5, 6))
dim(a)
## [1] 3 2

b <- rbind(c(1, 2, 3), c(4, 5, 6))
dim(b)
## [1] 2 3

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

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

# PRIMI PASSI CON R

```
dens <- density(c)
dx <- dens$x
dy <- dens$y
if(add == TRUE) {
  plot(0, 0, xlab = "x", ylab = "y", type = "n")
  if(content == "line") {
    lines(x, y)
  } else if(content == "point") {
    points(x, y)
  }
}
if(add == FALSE) {
  plot(x, y, xlab = "x", ylab = "y", main = "Density Plot")
}
if(is.null(dx2)) {
  dx2 <- (x[1] - x[0]) / 10
}
y[1] <- dy[1]
seqbelow <- seq(0, y[1], by = dx2)
if(Fill == TRUE) {
  confshade(dx2, seqbelow, dy2)
}
```

## 2.9 Stringhe (vettori di caratteri)

R permette di manipolare anche vettori di caratteri, o stringhe, che possono essere salvati anche come vettori. Le stringhe vengono delimitate da doppie virgolette " " (o anche semplici virgolette ' ').

```
nomi <- c("Francesco", "Sofia", "Alessandro")
nomi[1]

## [1] "Francesco"
nomi_e_numeri <- c("Francesco", "Sofia", "Alessandro", 45)
```

In R non possono convivere nello stesso array caratteri e numeri. Se ad esempio nell'assegnazione indichiamo nomi e numeri, questi ultimi verranno convertiti in caratteri.

Una funzione molto utile per maneggiare stringhe, ma anche altri risultati, è la funzione `paste()` che concatena dei vettori dopo averli trasformati in stringhe.

```
paste(1:12)

## [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"   "10"  "11"  "12"
(nth <- paste0(1:12, c("st", "nd", "rd", rep("th", 9))))
```

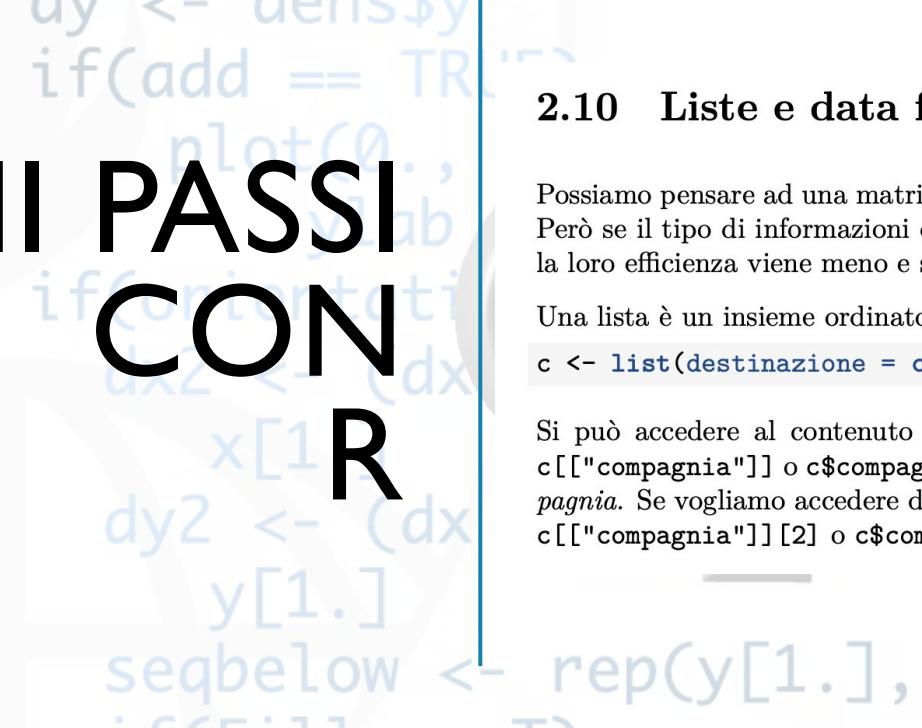
```
## [1] "1st"  "2nd"  "3rd"  "4th"  "5th"  "6th"  "7th"  "8th"  "9th"  "10th"
## [11] "11th" "12th"
```

Questa funzione può essere molto utile, ad esempio, se si devono creare vettori di nomi per delle variabili.

Per capire il tipo di dati contenuto in un vettore possiamo usare il comando `typeof()` o `class()`.

# PRIMI PASSI CON R

# IMI PASSI CON R



## 2.10 Liste e data frame

Possiamo pensare ad una matrice come a un insieme ordinato di oggetti. Però se il tipo di informazioni che dobbiamo memorizzare è diverso, la loro efficienza viene meno e si possono utilizzare liste.

Una lista è un insieme ordinato di oggetti di diverso tipo:

```
c <- list(destinazione = c("London", "Paris", "Berlin"),  
          compagnia = c("Alitalia", "Air France", "Lufthansa"),  
          numero_volo = c(1234, 5678, 9012),  
          orario_partenza = c("10:00", "12:30", "14:45"))
```

Si può accedere al contenuto di una lista con `c[["compagnia"]]` o `c$compagnia`. Tuttavia, le liste non sono particolarmente efficienti per memorizzare dati strutturati. Se vogliamo accedere direttamente all'elemento 2 della lista `c[["compagnia"]]`, dobbiamo scrivere `c[["compagnia"]][2]` o `c$compagnia[2]`.

## 2.10 Liste e data frame

Possiamo pensare ad una matrice come ad un metodo efficace per immagazzinare informazioni numeriche, ed è così! Però se il tipo di informazioni che dobbiamo immagazzinare è misto, ad esempio contiene sia numeri che caratteri, la loro efficienza viene meno e si possono preferire altri metodi.

Una lista è un insieme ordinato di oggetti. Si possono definire liste usando il comando `list()`.

```
c <- list(destinazione = c("London", "Madrid"), compagnia = c("Ryanair", "EasyJet"), costo = c(60, 80))
```

Si può accedere al contenuto di una lista sia per posizione con le doppie parentesi `c[[2]]` oppure per nome `c["compagnia"]` o `c$compagnia`. Tutti i precedenti comandi restituiscono il contenuto della lista definito da *compagnia*. Se vogliamo accedere direttamente ad un elemento possiamo usare indifferentemente i comandi `c[[2]][2]`, `c[["compagnia"]][2]` o `c$compagnia[2]`.



# PRIMI PASSI CON R

```
dens <- density(  
dx <- dens$x  
dy <- dens$y  
if(add == TR  
plot(0.,  
lab  
ifContenti  
dx2 <- dx  
x[1.]  
dy2 <- (dx  
y[1.]  
seqbelow <  
if(Fill ==  
confshac
```

## 2.10.1 Data frame

Un'altra struttura dati che permette di immagazzinare dati di tipo misto sono i *data frame*. Questa struttura è di gran lunga la più usata per leggere e manipolare dati in R.

I *data frame* sono liste di tipo “*data.frame*” contenenti variabili con lo stesso numero di righe, il cui identificativo è univoco.

```
L3 <- LETTERS[1:3]  
fac <- sample(L3, 10, replace = TRUE)  
(d <- data.frame(x = 1, y = 1:10, fac = fac))
```

```
##   x  y fac  
## 1 1  1  B  
## 2 1  2  B  
## 3 1  3  C  
## 4 1  4  C  
## 5 1  5  C  
## 6 1  6  C  
## 7 1  7  B  
## 8 1  8  A  
## 9 1  9  B  
## 10 1 10  A  
data.frame(1, 1:10, fac)
```

```
##   X1 X1.10 fac  
## 1  1     1  B  
## 2  1     2  B  
## 3  1     3  C  
## 4  1     4  C  
## 5  1     5  C  
## 6  1     6  C  
## 7  1     7  B  
## 8  1     8  A  
## 9  1     9  B  
## 10 1    10  A
```