

# Introduzione a R

## Giornata 4 - Programmazione in R



Corsi ARCA - @DPSS

Filippo Gambarota

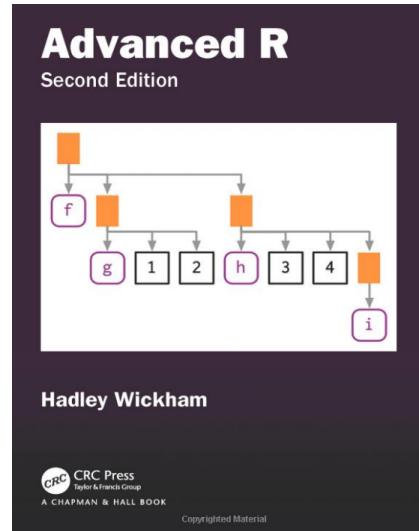
# Programmazione in R

Quello che vedremo in questa sezione sono i principali **costrutti della programmazione** e la loro applicazione in R. Ci sono alcuni punti da considerare:

- Sono concetti trasversali estremamente utili
- Sono alla base di qualcunque **funzionalità già implementata in R**
- Vi permettono di fare qualunque cosa con il linguaggio

# Programmazione in R - Disclaimer

Ci sono delle cose che per tempo e complessità non possiamo affrontare e che sono R specifiche. Per questi aspetti avanzati del linguaggio, il libro **Advanced R** è la cosa migliore



# Costrutti della programmazione in R

# Costrutti della programmazione in R

- Funzioni
- Programmazione condizionale
- Programmazione iterativa

# Funzioni

Analogalmente alle *funzioni matematiche* la funzione in programmazione consiste nell'**astrarre** una serie di operazioni (nel nostro caso una porzione di codice) definendo una serie di operazioni che forniti degli *input* forniscono degli *output* eseguendo una serie di *operazioni*

# Funzioni

Prendiamo l'equazione di una retta:  $y = 2x + 3$  dove 3

```
x <- 1:10  
y <- 2*x + 3
```

# Funzioni

Se vogliamo *astrarre* questa operazione in modo da renderla più generale e utile dobbiamo definire:

- **argomenti funzione**: quelle che in matematica sono le *variabili*
- **corpo funzione**: le **operazioni** che la funzione deve eseguire usando gli argomenti
- **output funzione**: cosa la funzione deve **restituire** come risultato

# Funzioni - Argomenti

Gli **argomenti** sono quelle parti variabili della funzione che vengono definiti e poi sono necessari ad eseguire la funzione stessa. Se vogliamo *astrarre* la retta che abbiamo visto prima dobbiamo definire alcune parti come **variabili**:

La retta  $y = mx + q$  dove  $m$  è la pendenza,  $x$  sono i valori della variabile  $x$  e  $q$  è l'intercetta (valore di  $y$  quando  $x = 0$ ). Quindi possiamo definire in R:

```
retta <- function(m, x, q){ # argomenti
  # body
  # output
}
```

# Funzioni - Body

Il corpo della funzione sono le operazioni da eseguire utilizzando gli argomenti in input. Nel caso della retta semplicemente moltiplicare  $m$  per ogni valore di  $x$  e aggiungere  $q$ . In questo modo otteniamo tutti i valori di  $y$ :

```
retta <- function(m, x, q){ # argomenti  
  y <- m*x + q  
  # output  
}
```

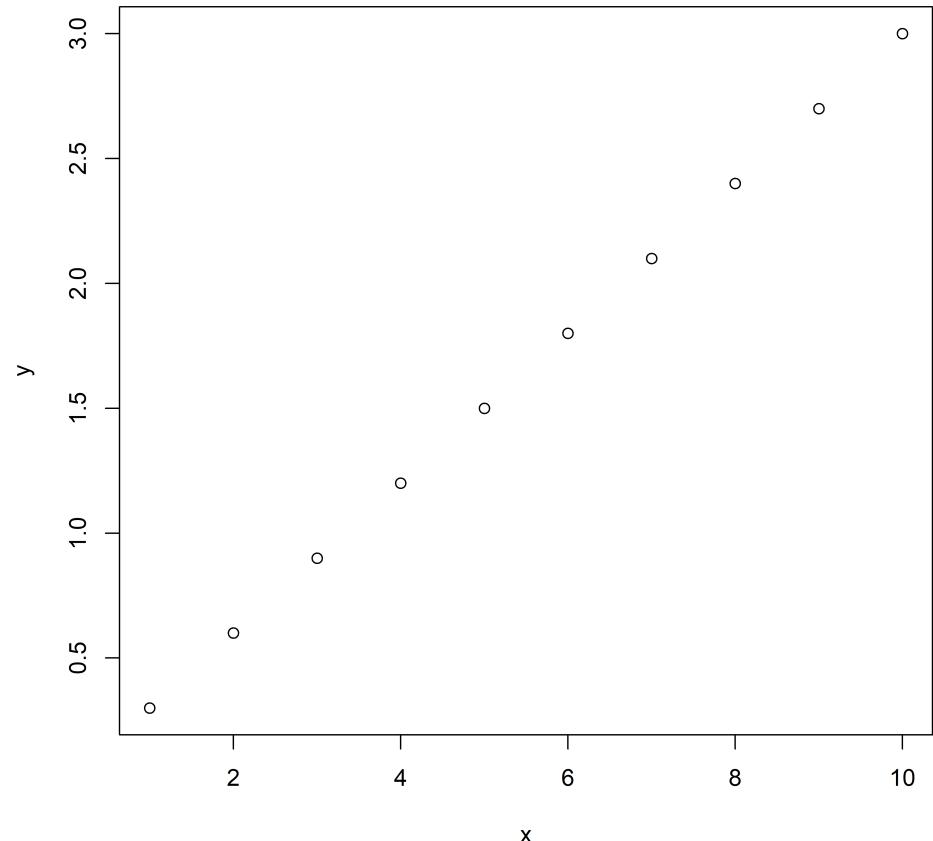
# Funzioni - Output

L'output è il **risultato che la funzione ci restituisce** dopo aver eseguito tutte le operazioni. Nel nostro caso della retta, vogliamo ottenere il rispettivo valore di  $y$  per ogni valore di  $x$  inserito:

```
retta <- function(m, x, q){ # argomenti
  y <- m*x + q
  return(y) # restituisce y
}
```

# Funzioni - Risultato finale

```
retta <- function(m, x, q){ # argomenti  
  y <- m*x + q  
  
  return(y) # restituisce y  
}  
  
x <- 1:10  
m <- 0.3  
q <- 0  
  
y <- retta(m, x, q)
```



# Programmazione condizionale

# Programmazione condizionale

In programmazione solitamente è necessario non solo eseguire una serie di operazione **MA** eseguire delle operazione in funzione di alcune **condizioni**

Facciamo un esempio pratico, la funzione `summary()` in R fornisce un risultato diverso in base al tipo di input. Come è possibile tutto questo? Tramite l'utilizzo di **condizioni**:

```
x <- 1:10 # vettore numerico  
y <- factor(rep(c("a", "b", "c"), each = 10)) # vettore di stringhe  
summary(x)
```

```
##      Min. 1st Qu. Median      Mean 3rd Qu.      Max.  
##     1.00    3.25   5.50    5.50    7.75   10.00
```

```
summary(y)
```

```
## a b c  
## 10 10 10
```

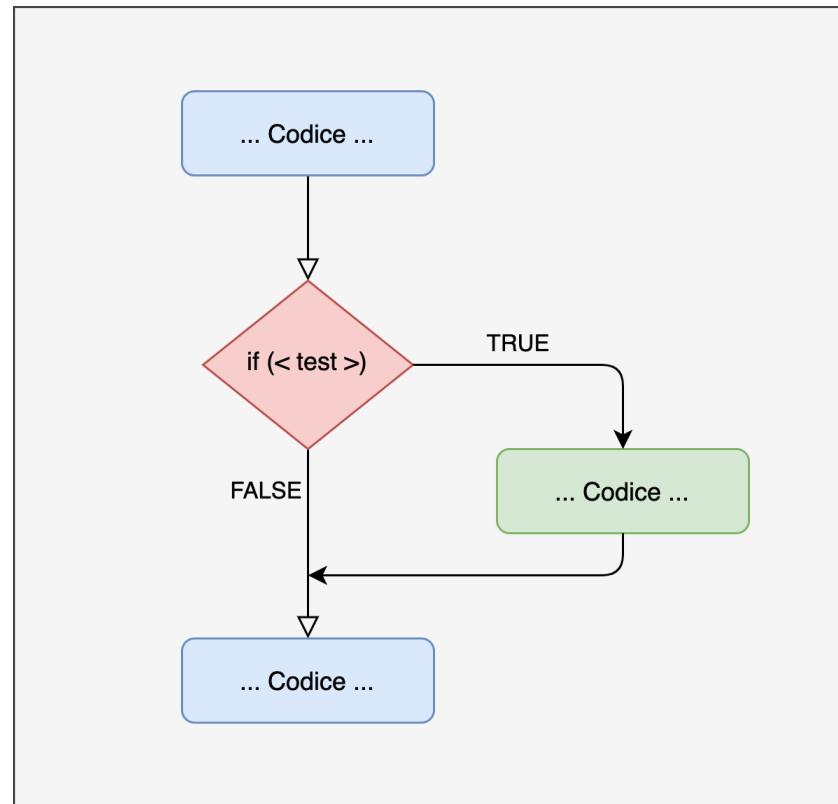
# Programmazione condizionale

Anche se non sappiamo quali operazioni svolga la funzione `summary()` possiamo immaginare una cosa simile

```
summary <- function(argomento){  
  
  # se l'argomento è un vettore numerico  
  # esegui --> operazioni a,b,c  
  
  # se l'argomento è un vettore stringa  
  # esegui --> operazioni d,e,f  
  
  # ...  
}
```

# Programmazione condizionale

Il concetto di `se <condizione> allora fai <operazione>` si traduce in programmazione tramite quelli che si chiamano `if statement`:



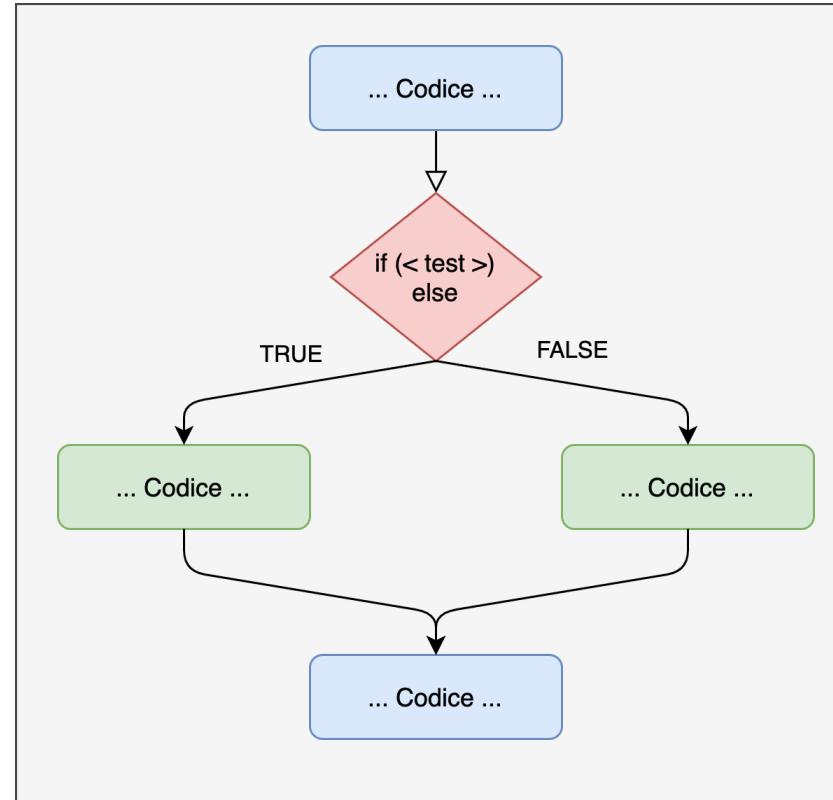
# Programmazione condizionale

Per lavorare con gli `if statements` dobbiamo avere chiaro:

- il concetto di *operatori logici* ovvero `TRUE` e `FALSE`
- il concetto di *operazioni logiche* `TRUE and TRUE = TRUE`

# Programmazione condizionale

Quando una sola condizione non basta...



# Programmazione condizionale

Per poter capire quale struttura condizionale utilizzare è importante capire bene il problema che dobbiamo risolvere.

Ritornando all'esempio della funzione `summary()`, immaginiamo di avere 2 tipi di dati in R; stringhe e numeri.

In questo caso è sufficiente avere un `if statement` che controlla se l'elemento è una stringa/numero e per tutto il resto applicare l'opposto.

# Programmazione condizionale - Tip

Esiste una famiglia di funzioni con prefisso `is.*` che fornisce `TRUE` quando la tipologia di oggetto corrisponde a quella richiesta e `FALSE` in caso contrario.

```
x <- 1:10  
  
is.numeric(x)  
  
## [1] TRUE  
  
is.factor(x)  
  
## [1] FALSE  
  
is.character(x)  
  
## [1] FALSE
```

Possiamo usare queste funzioni per creare un flusso condizionale nella nostra funzione `summary()`

# Programmazione condizionale

Scriviamo una funzione che restituisca la `media` quando il vettore è numerico e la tabella di frequenza (con la funzione `table()`)

```
my_summary <- function(x){  
  
  # testiamo la condizione  
  
  if(is.numeric(x)){  
    return(mean(x))  
  }else{  
    return(table(x))  
  }  
}  
  
x <- 1:10  
my_summary(x)
```

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

```
x <- rep(c("a","b","c"), c(10, 2, 8))  
my_summary(x)
```

```
## x  
## a b c  
## 10 2 8
```

## ifelse()

Un limite di usare gli `if` statements riguarda il fatto che funzionano solo su un singolo valore (i.e. non sono **vettorizzati**):

```
x <- 1:10
if(x < 5){
  print("x è minore di 5")
} else{
  print("x è maggiore di 5")
}
```

```
## [1] "x è minore di 5"
```

La versione vettorizzata è la funzione `ifelse(test, yes, no)`:

```
ifelse(x < 5, "x è minore di 5", "x è maggiore di 5")
```

```
## [1] "x è minore di 5"  "x è maggiore di 5"
## [6] "x è maggiore di 5" "x è maggiore di 5"
```

## ifelse()

Come anche per gli `if` statements normali, posso creare degli `ifelse()` nested quando ho bisogno di testare più alternative. Immaginiamo di avere una colonna/vettore `age` e voler creare un altro vettore dove l'età è divisa in 3 fascie, bambino, adulto, anziano:

```
age <- round(runif(50, 3, 80))
age_ifelse <- ifelse(age < 18,
  yes = "bambino",
  no = ifelse(
    age >= 18 & age < 60,
    "adulto",
    "anziano"
  ))
```

## dplyr::case\_when()

Quando le condizioni da testare sono numerose (indicativamente > 3) può essere tedioso scrivere molti `ifelse()` multipli. Possiamo allora usare la funzione `dplyr::case_when()` del pacchetto `dplyr` che è una generalizzazione di `ifelse()`:

```
age_case_when <- case_when(age < 18 ~ "bambino",
                            age >= 18 & age < 60 ~ "adulto",
                            TRUE ~ "anziano") # con TRUE si identifica "tutto il resto" in modo da non lasciare valori scoperti (ATTENZIONE)
```

I due risultati sono identici:

```
all.equal(age_case_when, age_ifelse)

## [1] TRUE
```

# Programmazione iterativa

# Programmazione iterativa

Il concetto di *iterazione* è alla base di qualsiasi operazione nei linguaggi di programmazione.

In R molte delle operazioni sono vettorizzate. Questo rende il linguaggio più efficiente e pulito MA nasconde il concetto di *iterazione*

# Programmazione iterativa

Esempio: se io vi chiedo di usare la funzione `print()` per scrivere "hello world" nella console 10 volte, come fate?

```
msg <- "Hello World"  
print(msg)
```

```
## [1] "Hello World"
```

```
print(msg)
```

```
## [1] "Hello World"
```

```
print(msg)
```

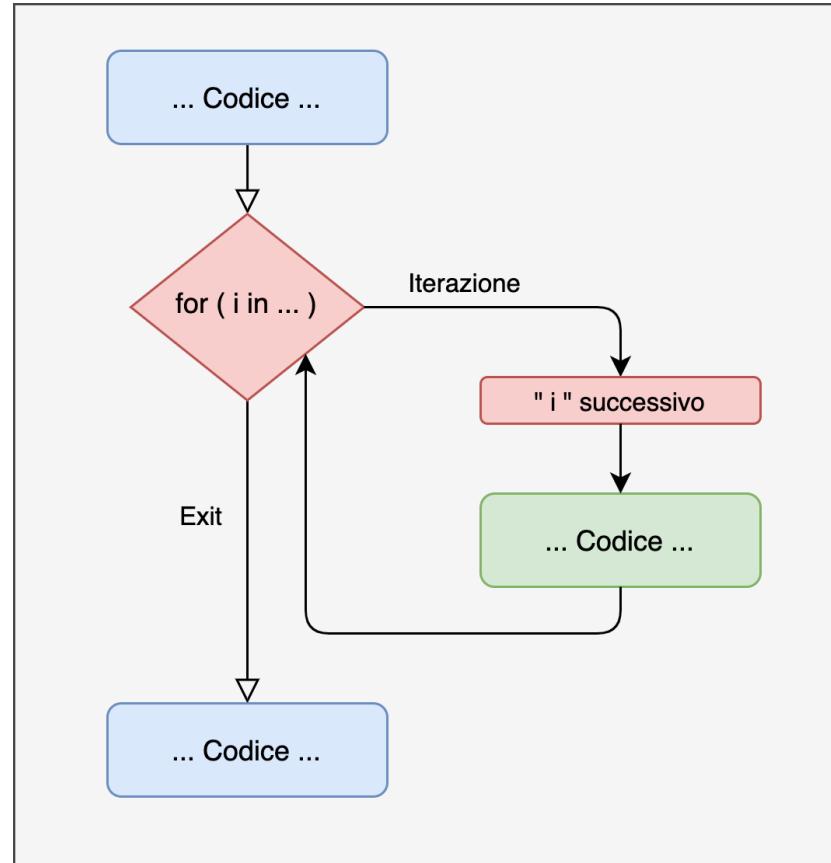
```
## [1] "Hello World"
```

```
print(msg)
```

```
## [1] "Hello World"
```

```
print(msg)
```

# For



# For

La scrittura di un ciclo `for` è:

```
for(i in 1:n){  
    # operazioni  
}
```

# Scomponiamo il ciclo for

Ci sono diversi elementi:

- `for(){}`: è l'implementazione in R (in modo simile all'`if statement`)
- `i`: questo viene chiamato *iteratore* o *indice*. E' un indice generico che può assumere qualsiasi valore e nome. Per convenzione viene chiamato `i`, `j` etc. Questo tiene conto del numero di iterazioni che il nostro ciclo deve fare
- `in <valori>`: questo indica i valori che assumerà l'*iteratore* all'interno del ciclo
- `{ # operazioni }`: sono le operazioni che il ciclo deve eseguire

# Hello world come ciclo for

```
# vogliamo che il ciclo ripeta l'operazione 10 volte

for(i in 1:10){
  print("hello world")
}
```

```
## [1] "hello world"
```

# Ma l'iteratore?

La potenza del ciclo `for` sta nel fatto che l'iteratore `i` assume i valori del vettore specificato dopo `in`, uno alla volta:

```
for(i in 1:10){  
  print(i)  
}
```

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

# For con iteratore vs senza

Questa è una distinzione importante quanto sottile, notate la differenza tra questi due cicli:

```
vec <- 1:5

for(i in 1:length(vec)){
  print(vec[i])
}
```

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

```
vec <- 1:5

for(i in vec){
  print(i)
}
```

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

# While

Il ciclo `while` è una versione più generale del ciclo `for`. Per funzionare utilizza una *condizione logica* e non un iteratore e un range di valori come nel `for`.

```
while(condizione){  
    # operazioni  
}
```

Dove il ciclo continueará fino a che la `condizione` è vera

# While - (Fun)

Provate a scrivere questo ciclo while e vedere cosa succede:

```
x <- 10  
  
while (x < 15) {  
  print(x)  
}
```

Chi mi sa spiegare il risultato?

# While

Questo esercizio è utile per capire che il `while` è un ciclo non pre-determinato e quindi necessita sempre di un modo per essere interrotto, facendo diventare la condizione falsa.

```
x <- 5

while (x < 15) {
  print(x)
  x <- x + 1
}
```

```
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
## [1] 10
## [1] 11
## [1] 12
## [1] 13
## [1] 14
```

# Applicazioni dei cicli

Gli esempi finora sono semplici ma poco utili. Quando il queste strutture iterative sono veramente utili?

Molte delle funzioni che utilizziamo come ad esempio `sum()`, `mean()`, etc. hanno al loro interno una struttura iterativa

Immaginiamo di non avere la funzione `sum()` e di volerla ricreare, come facciamo? Idee?

# Somma come iterazione

Scomponiamo concettualmente la somma, sommiamo i numeri da 1 a 10:

- prendo il primo e lo sommo al secondo (`somma = 1 + 2`)
- prendo la `somma` e la sommo al 3 elemento `somma = somma + 3`
- ...

In pratica abbiamo:

- il nostro vettore da sommare
- un oggetto `somma` che accumula progressivamente le somme precedenti

# Somma come iterazione

```
somma <- 0 # inizializziamo la somma a 0
x <- 1:10

for(i in seq_along(x)){
  somma <- somma + x[i]
}
```

# Somma come iterazione

Mettiamo tutto dentro una funzione

```
my_sum <- function(x){  
  somma <- 0 # inizializziamo la somma a 0  
  
  for(i in seq_along(x)){  
    somma <- somma + x[i]  
  }  
  
  return(somma)  
}  
  
x <- rnorm(100)  
  
my_sum(x)
```

```
## [1] -9.006046
```

```
sum(x)
```

```
## [1] -9.006046
```

**Ma in R c'è qualcosa di meglio...**

# Ma in R c'è qualcosa di meglio...

In R, l'utilizzo **esplicito** dei cicli `for` non è molto diffuso, per 2 motivi:

- R è un linguaggio fortemente **funzionale**
- R è un linguaggio spesso **vettorizzato**
- I cicli `for` sono molto verbosi e non sempre leggibili
- I cicli `for` in R, se non scritti bene, possono essere *estremamente lenti*

\*apply family

## \*apply family

Immaginate di avere una lista di vettori, e di voler applicare la stessa funzione/i ad ogni elemento della lista. Come fare? ^[1]

- applico manualmente la funzione selezionando gli elementi
- ciclo for che itera sugli elementi della lista e applica la funzione/i
- ...

```
my_list <- list(  
  vec1 <- rnorm(100),  
  vec2 <- runif(100),  
  vec3 <- rnorm(100),  
  vec4 <- rnorm(100)  
)
```

Hadley Wickam - The joy of functional programming - [link](#)

# \*apply family

Applichiamo `media`, `mediana` e `deviazione standard`:

```
means <- vector(mode = "numeric", length = length(my_list))
medians <- vector(mode = "numeric", length = length(my_list))
stds <- vector(mode = "numeric", length = length(my_list))

for(i in 1:length(my_list)){
  means[i] <- mean(my_list[[i]])
  medians[i] <- median(my_list[[i]])
  stds[i] <- sd(my_list[[i]])
}
```

means

```
## [1] -0.05382864  0.50411643  0.09652737 -0.05403042
```

medians

```
## [1] -0.03338468  0.51095730  0.03549972  0.09538814
```

stds

```
## [1] 0.9943513 0.2914769 0.9949019 0.9242835
```

## \*apply family

Funziona tutto! ma:

- il `for` è molto laborioso da scrivere gli indici sia per la lista che per il vettore che stiamo popolando
- dobbiamo *pre-allocare delle variabili* (per il motivo della velocità che dicevo)
- 8 righe di codice (per questo esempio semplice)

## \*apply family

In R è presente una famiglia di funzioni `*apply` come `lapply`, `sapply`, etc. che permettono di ottenere lo stesso risultato in modo più conciso, rapido e semplice:

```
means <- sapply(my_list, mean)
medians <- sapply(my_list, median)
stds <- sapply(my_list, sd)

means
```

```
## [1] -0.05382864  0.50411643  0.09652737 -0.05403042
```

```
medians
```

```
## [1] -0.03338468  0.51095730  0.03549972  0.09538814
```

```
stds
```

```
## [1] 0.9943513 0.2914769 0.9949019 0.9242835
```

## \*apply family - Bonus

Prima di introdurre l'\*apply family un piccolo bonus. Sfruttando il fatto che in R **tutto è un oggetto** possiamo scrivere in modo ancora più conciso:

```
my_funcs <- list(median = median, mean = mean, sd = sd)  
lapply(my_list, function(vec) sapply(my_funcs, function(fun) fun(vec)))
```

```
## [[1]]  
##      median        mean         sd  
## -0.03338468 -0.05382864  0.99435132  
##  
## [[2]]  
##      median        mean         sd  
##  0.5109573  0.5041164  0.2914769  
##  
## [[3]]  
##      median        mean         sd  
##  0.03549972  0.09652737  0.99490190  
##  
## [[4]]  
##      median        mean         sd  
##  0.09538814 -0.05403042  0.92428349
```

Amazing! ora cerchiamo di dare un senso a queste righe di codice!

## \*apply family

```
apply(<lista>, <funzione>)
```

- cosa può essere la `lista`?
  - lista
  - `dataframe`
  - `vettore`
- cosa può essere la `funzione`?
  - funzione *base* o importata *pacchetto*
  - funzione *custom*
  - funzione *anonima*

## \*apply family - intuizione

Prima di analizzare l'\*apply family, credo sia utile un ulteriore parallelismo con il ciclo `for` che abbiamo visto. \*apply non è altro che un ciclo `for`, leggermente semplificato:

```
vec <- 1:5
for(i in vec){
  print(i)
}
```

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

```
vec <- 1:5
res <- sapply(vec, print)
```

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

## \*apply family - spoiler funzione anonima

Quindi come il ciclo `for` scritto come `i in vec` assegna al valore `i` un elemento per volta dell'oggetto `vec`, internamente le funzioni `*apply` prendono il primo elemento dell'oggetto in input (`lista`) e applicano direttamente la funzione che abbiamo scelto.

C'è un modo per rendere esplicito questo, anche nelle funzioni `*apply`:

```
vec <- 1:5  
res <- sapply(vec, print)
```

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

```
vec <- 1:5  
res <- sapply(vec, function(i) print(i))
```

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

# \*apply e funzioni custom

```
center_var <- function(x){  
  x - mean(x)  
}  
  
my_list <- list(  
  vec1 = runif(10),  
  vec2 = runif(10),  
  vec3 = runif(10)  
)  
  
lapply(my_list, center_var)
```

```
## $vec1  
## [1] -0.55944411 -0.64224316  0.11720122  0.28610132 -0.16694580  0.32907466  0.30741131  0.07993166  0.00799079  
## [10]  0.24092212  
##  
## $vec2  
## [1]  0.135083087 -0.333115064 -0.101103943  0.064531171  0.000721796  0.122756606 -0.277722296  0.178063709  0.022237941  
## [10]  0.188546995  
##  
## $vec3  
## [1] -0.01981908  0.17645168  0.51305045 -0.33344090  0.24285221 -0.24193842 -0.41948858  0.41516276 -0.05124459  
## [10] -0.28158554
```

## \*apply e funzioni anonymous

Una funzione anonima è una funzione non salvata in un oggetto ma scritta per essere **eseguita direttamente**, all'interno di altre funzioni che lo permettono:

```
lapply(my_list, function(x) x - mean(x))

## $vec1
## [1] -0.55944411 -0.64224316  0.11720122  0.28610132 -0.16694580  0.32907466  0.30741131  0.07993166  0.00799079
## [10]  0.24092212
##
## $vec2
## [1]  0.135083087 -0.333115064 -0.101103943  0.064531171  0.000721796  0.122756606 -0.277722296  0.178063709  0.022237941
## [10]  0.188546995
##
## $vec3
## [1] -0.01981908  0.17645168  0.51305045 -0.33344090  0.24285221 -0.24193842 -0.41948858  0.41516276 -0.05124459
## [10] -0.28158554
```

Come per i cicli `for` (ricordo che `*apply` e `for` sono identici), `x` è solo un placeholder (analogo di `i`) e può essere qualsiasi lettera o nome

# Tutte le tipologie di `*apply`

Vediamo tutti i tipi di `*apply` che ci sono. Alcuni sono più *utili* altri più *robusti* e altri ancora poco utilizzati:

- `lapply()`: la funzione di base
- `sapply()`: *simplified-apply*
- `tapply()`: poco utilizzata, utile con i *fattori*
- `apply()`: utile per i *dataframe/matrici*
- `mapply()`: versione multivariata, utilizza *più liste contemporaneamente*
- `vapply()`: utilizzata dentro le funzioni e pacchetti

# lapply

`lapply` sta per list-apply e restituisce sempre una lista, applicando la funzione ad ogni elemento della lista in input:

```
res <- lapply(my_list, mean)  
res
```

```
## $vec1  
## [1] 0.6488907  
##  
## $vec2  
## [1] 0.7606104  
##  
## $vec3  
## [1] 0.4698074
```

```
class(res)
```

```
## [1] "list"
```

# sapply

`sapply` sta per simplified-apply e (cerca) di restituire una versione più semplice di una lista, applicando la funzione ad ogni elemento della lista in input:

```
res <- sapply(my_list, mean)  
res
```

```
##      vec1      vec2      vec3  
## 0.6488907 0.7606104 0.4698074
```

```
class(res)
```

```
## [1] "numeric"
```

# apply

`apply` funziona in modo specifico per dataframe o matrici, applicando una funzione alle righe o alle colonne:

- `apply(dataframe, index, fun)`

```
# index 1 = riga, 2 = colonna
my_dataframe <- data.frame(my_list)
head(my_dataframe)
```

```
##           vec1      vec2      vec3
## 1 0.089446633 0.8956935 0.4499883
## 2 0.006647577 0.4274953 0.6462590
## 3 0.766091957 0.6595064 0.9828578
## 4 0.934992055 0.8251416 0.1363665
## 5 0.481944935 0.7613322 0.7126596
## 6 0.977965396 0.8833670 0.2278689
```

```
apply(my_dataframe, 1, mean)
```

```
## [1] 0.4783761 0.3601340 0.8028187 0.6321667 0.6519789 0.6964004 0.4965030 0.8508222 0.6194309 0.6757307
```

```
apply(my_dataframe, 2, mean)
```

# tapply

`tapply` permette di applicare una funzione ad un *vettore*, dividendo questo vettore in base ad una variabile categoriale:

- `tapply(dataframe, index, fun)`: dove `index` è un vettore di stringa o un fattore

```
vec <- rnorm(75)
index <- rep(c("a", "b", "c"), each = 25)

tapply(vec, index, mean)
```

```
##          a            b            c
##  0.04371342 -0.09070020  0.17959196
```

## vapply

`vapply` è una versione più *solida* delle precedenti dal punto di vista di programmazione. In pratica permette (e richiede) di specificare in anticipo la tipologia di dato che ci aspettiamo come risultato

```
vapply(X = , FUN = , FUN.VALUE = ,... )
```

```
vapply(my_list, FUN = mean, FUN.VALUE = numeric(length = 1))
```

```
##      vec1      vec2      vec3
## 0.6488907 0.7606104 0.4698074
```

- `my_list, FUN = mean`: è esattamente uguale a `sapply/lapply`
- `FUN.VALUE = numeric(length = 1)`: indica che ogni risultato è un singolo valore numerico

## mapply

Questa è quella più complicata ma anche molto utile. Praticamente permette di gestire più liste contemporaneamente per scenari più complessi. Ad esempio vogliamo usare la funzione `rnorm()` e generare vettori con diverse **medie** e **deviazioni standard** in combinazione.

```
medie <- list(10, 20, 30, 40)
stds <- list(1,2,3,4)
mapply(function(x, y) rnorm(n = 10, mean = x, sd = y), medie, stds, SIMPLIFY = FALSE)

## [[1]]
## [1] 11.346738 10.855191 10.905297 9.365732 11.199642 9.848514 10.274577 9.607727 9.744790 9.899508
##
## [[2]]
## [1] 21.52083 21.21616 21.88517 23.99179 20.14811 19.70998 17.90297 24.52241 18.28226 17.32871
##
## [[3]]
## [1] 33.91628 25.53216 30.41152 30.93187 30.13395 36.36537 27.18235 28.48627 30.22389 21.24562
##
## [[4]]
## [1] 39.43072 37.99866 31.54536 36.96803 36.26470 46.48045 42.92063 41.34825 39.38414 38.17279
```

**IMPORTANTE**, tutte le liste incluse devono avere la stessa dimensione!

# mapply

```
mapply(function(x, y) rnorm(n = 10, mean = x, sd = y), medie, stds, SIMPLIFY = FALSE)
```

- `function(...)`: è una funzione anonima come abbiamo visto prima che può avere *n* elementi
- `rnorm(n = 10, mean = x, sd = y)`: è l'effettiva funzione anonima dove abbiamo i placeholders `x` and `y`
- `medie, stds`: sono **in ordine** le liste corrispondenti ai placeholders indicati, quindi `x = medie` e `y = stds`.
- `SIMPLIFY = FALSE`: semplicemente dice di restituire una lista e non cercare (come `sapply`) di semplificare il risultato

## mapply come for

Lo stesso risultato (in modo più verboso e credo meno intuitivo) si ottiene con un `for` usando più volte l'iteratore `i`:

```
medie <- list(10, 20, 30, 40)
stds <- list(1,2,3,4)

res <- vector(mode = "list", length = length(medie))

for(i in 1:length(medie)){
  res[[i]] <- rnorm(10, mean = medie[[i]], sd = stds[[i]])
}

res

## [[1]]
## [1] 11.169725 10.765628  9.488761 10.859841 11.758599 10.432526 10.207910 11.392780 11.567589 11.367156
##
## [[2]]
## [1] 21.36599 18.49624 21.12178 18.77147 18.74120 23.95120 18.73967 22.59055 20.92746 21.34965
##
## [[3]]
## [1] 25.11392 26.67587 30.95449 24.11077 29.68023 27.93769 28.65553 30.30804 27.19500 23.66675
##
## [[4]]
## [1] 38.97816 42.83830 41.65930 40.05615 44.43277 39.35387 37.27550 44.31865 34.98216 36.82147
```

\*apply alcune precisazioni

## \*apply vettore vs lista

Abbiamo sempre usato esplicitamente `liste` fino ad ora, ma le funzioni `*apply` sono direttamente applicabili anche a **vettori**

- se usiamo un vettore di  $n$  elementi, allora itereremo da `1:n`
- se usiamo una lista di  $n$  elementi, allora iteriamo da `1:n` dove il singolo elemento può essere qualsiasi cosa

```
my_vec <- 1:5
my_list <- list(a = 1:2, b = 3:4, c = 5:6)
res <- sapply(my_vec, print)
```

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

```
res <- sapply(my_list, print)
```

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

## \*apply come un for

Nulla ci vieta (ma perdiamo l'aspetto intuitivo e conciso) di usare le funzioni \*apply esattamente come un ciclo `for`, usando un **iteratore**:

```
medie <- c(10, 20, 30, 40)
stds <- c(1,2,3,4)

res <- lapply(1:length(medie), function(i){
  rnorm(n = 10, mean = medie[i], sd = stds[i])
})
```

Trovo tuttavia più chiara l'alternativa usando `mapply`:

```
mapply(function(x, y) rnorm(n = 10, mean = x, sd = y), medie, stds, SIMPLIFY = FALSE)
```

**Extra:** `purrr::map*`

## Extra: purrr::map\*



Senza addentrarci troppo in questo modo, c'è una famiglia di funzioni che una volta imparato `*apply` vi consiglio di usare perchè più consistenti e intuitive, la `map*` family.

## Extra: purrr::map\*

Per usare `purrr::map*` è sufficiente installare il pacchetto `purrr` con `install.packages("purrr")` ed iniziare ad usare le nuove funzioni. La sintassi è esattamente la stessa di `*apply` (qualche modifica ma potete usare la stessa) ma invece che usare una funzione per tutto, abbiamo molte funzioni per ogni casistica:

- `map(lista, funzione)` è l'analogo di `lapply()` e fornisce sempre una lista
- `map_dbl(lista, funzione)` applica la funzione ad ogni elemento e **si aspetta che** il risultato sia un vettore di *double*
- `map_lgl(lista, funzione)` applica la funzione ad ogni elemento e **si aspetta che** il risultato sia un vettore *logico*

**Extra:** `replicate()` and `repeat()`

## Extra: `replicate()` and `repeat()`

Ci sono altre due funzioni in R che permettono di *iterare*. Sono meno utilizzate perché si ottengono gli stessi risultati usando un semplice `for` o `*apply`.

- `replicate()` permette di ripetere un'operazione  $n$  volte, senza però utilizzare un iteratore o un placeholder.
- `repeat()` anche `repeat` permette di ripetere ma fino a che non si verifica un'certa condizione (**logica**). Ha una struttura simile al ciclo `while`

# Extra: Formula syntax

# Formula syntax

In R molte operazioni vengono eseguite usando la **formula syntax** `something ~ something else` ad esempio:

- modelli statistici: `lm(y ~ x, data = data)`, `t.test(y ~ factor, data = data)`
- plot: `boxplot(y ~ x, data = data)`
- ...

In cosa consiste?

# Formula syntax

Senza andare nei dettagli tecnici, R usa una cosa che si chiama *lazy evaluation*. In altri termini "salva" delle operazioni per essere eseguite in un secondo momento. Tutti sappiamo che se scriviamo un nome (senza virgolette) e questo non è associato ad un oggetto otteniamo un errore. Tuttavia alcune funzioni come `library()` non forniscono errore. Perchè?

```
stats # errore
```

```
## Error in eval(expr, envir, enclos): object 'stats' not found
```

```
library(stats) # no errore
```

# Formula syntax

La ragione è che R è in grado di salvare un'espressione per usarla poi in uno specifico contesto (ad esempio dentro una funzione). La `formula syntax` è un esempio. Usando la tilde `~` possiamo creare delle `formule` che R può utilizzare in specifici contesti:

```
y
```

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

```
x
```

```
## [1] 0.562538269 -1.352965903 0.872602161 -0.695933408 1.735947629 -0.716001291 -0.829652421 0.219130257  
## [9] 0.424901602 1.374546109 -0.728303723 0.434067967 0.942549020 -0.005015997 -0.770029095 -3.297029895  
## [17] -1.883726757 1.022372574 -1.676234190 0.011002606 1.529719877 2.674790880 0.665106431 0.404846505  
## [25] 0.884786708 -1.379918304 -1.575924535 -2.400439001 -0.263610885 -1.575275335 -1.330935165 -0.323312826  
## [33] 0.445873378 -1.113913026 0.189676018 -0.333559816 0.195009341 0.726838827 -2.066275521 0.207941087  
## [41] 0.631513211 -0.454026634 -0.114577044 -1.225683197 -0.971214125 0.389659484 -0.671198790 0.786358434  
## [49] -0.260492278 -0.498850239 0.456229564 0.525313280 0.471903488 -1.586137058 0.867324222 0.529302379  
## [57] 0.819638901 -0.981444537 -1.356334531 0.485097965 0.281995995 -0.570626554 -0.232196367 1.267142273  
## [65] -1.187706293 -0.960436956 -0.312625221 -0.209932321 0.261132426 -0.034569366 0.996672646 -0.081841545  
## [73] -1.977968150 0.196962693 1.998762994 1.482147024 0.463375588 -0.006174532 0.265410397 1.428395699  
## [81] -0.023357759 0.899885863 -1.083683236 1.183152522 2.346811233 -1.835408464 -0.915650400 0.974859983  
## [89] 0.314767243 -1.516159512 0.369554107 0.661199678 0.702893063 -0.984711651 -1.523063854 0.512504697  
## [97] -1.880941524 -1.394718821 1.758474427 0.311052931
```

# Formula syntax e aggregate()

Un esempio utile è la funzione `aggregate()` molto interessante per applicare funzioni a dataframes. Immaginate di avere il dataset `iris` e calcolare la media per ogni livello del fattore `Species`:

```
tapply(iris$Sepal.Length, iris$Species)
```

```
aggregate(Sepal.Length ~ Species, FUN = mean, data = iris)
```

```
##           Species Sepal.Length  
## 1      setosa      5.006  
## 2 versicolor      5.936  
## 3 virginica      6.588
```

*# Anche creando un oggetto, ma solo come formula*

```
my_formula <- Sepal.Length ~ Species  
my_char <- "Sepal.Length ~ Species"  
aggregate(my_char, FUN = mean, data = iris)
```

# Formula syntax e aggregate()

Ma anche operazioni più complesse:

```
my_iris <- iris
my_iris$fac <- rep(c("a", "b", "c"), 50)
aggregate(Sepal.Length ~ Species + fac, mean, data = my_iris)
```

```
##      Species fac Sepal.Length
## 1      setosa   a    5.052941
## 2 versicolor   a    5.770588
## 3  virginica   a    6.756250
## 4      setosa   b    5.011765
## 5 versicolor   b    6.018750
## 6  virginica   b    6.447059
## 7      setosa   c    4.950000
## 8 versicolor   c    6.023529
## 9  virginica   c    6.570588
```

# Replicate

```
replicate(n, expr)
```

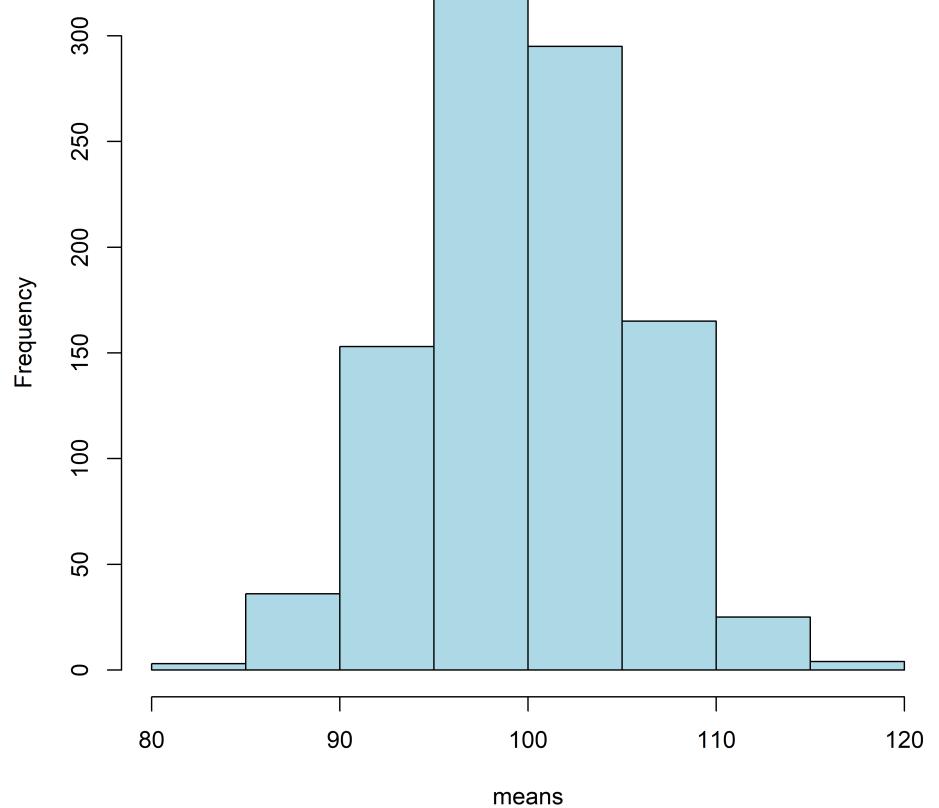
- `n` è il numero di ripetizioni
- `expr` è la porzione di codice da ripetere

```
# Campioniamo 1000 volte da una normale e facciamo la media AKA dist

nrep <- 1000
nsample <- 30
media <- 100
ds <- 30

means <- replicate(n = nrep, expr = {
  mean(rnorm(nsample, media, ds))
})
```

Histogram of means



# repeat()

```
repeat {  
  # cose da ripetere  
  
  if(...){ # condizione da valutare  
  
    break # ferma il loop  
  }  
}
```

```
i <- 1  
  
repeat {  
  print(i)  
  i = i + 1  
  if(i > 3){  
    break  
  }  
}
```

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

## repeat() **vs** while

```
i <- 1

repeat {
  print(i)
  i = i + 1
  if(i > 3){
    break
  }
}
```

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

```
i <- 1

while(i < 4){
  print(i)
  i <- i + 1
}
```

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

- `repeat` valuta la condizione una volta finita l'iterazione, mentre `while` all'inizio. Se la condizione non è `TRUE` all'inizio, il `while` non parte mentre `repeat` sì.