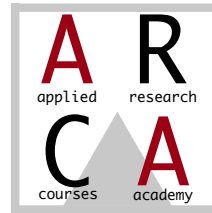


class: title-slide, center, middle

## Giornata 3 - Programmazione in R



Corsi ARCA - @DPSS

Filippo Gambarota

# Programmazione in R

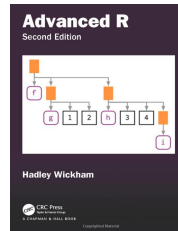
Quello che vedremo in questa sezione sono i principali **costrutti della programmazione** e la loro applicazione in R. Ci sono alcuni punti da considerare:

- Sono concetti trasversali estremamente utili
- Sono alla base di qualunque **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

```
put_image("adv_R.png")
```



# **Costrutti della programmazione in R**

# Costrutti della programmazione in R

- Funzioni
- Programmazione condizionale
- Programmazione iterativa

# Funzioni

Analogamente 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  
plot(x, y, xlim = c(0, 10), ylim = c(0, 30), type = "l")
```

# 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

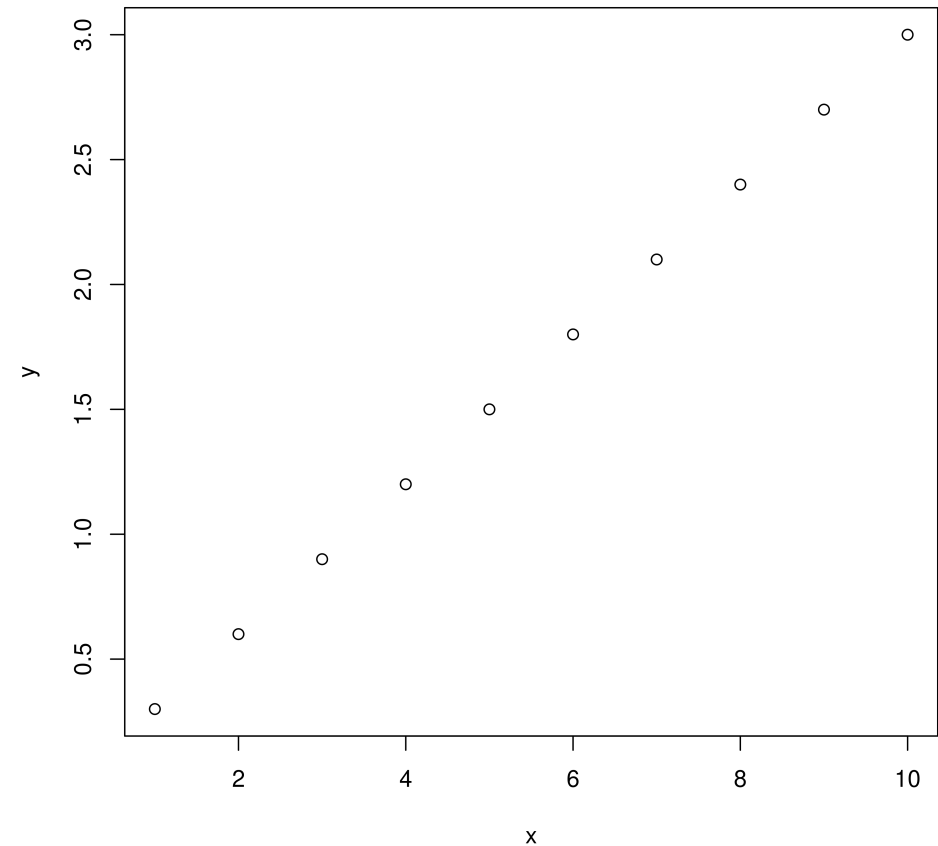
  return(y) # restituisce y
}
```

# 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

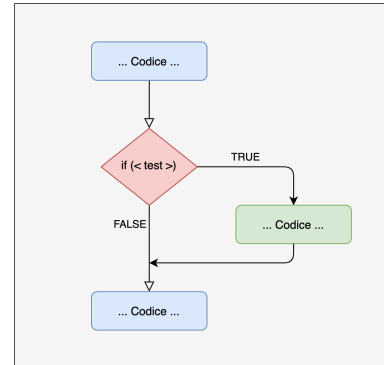
Anche se non sappiamo quali operazioni svolga la funzione `summary()` possiamo immaginare una cosa simile

```
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`:

```
put_image("if_chart.png")
```





# Programmazione condizionale

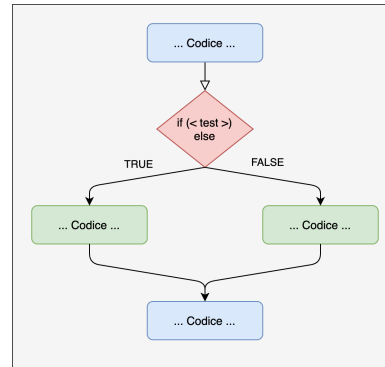
Per lavorare con gli `if statements` dobbiamo avere chiaro:

- 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...

```
put_image("ifelse_chart.png")
```



# 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
```

# **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)
```

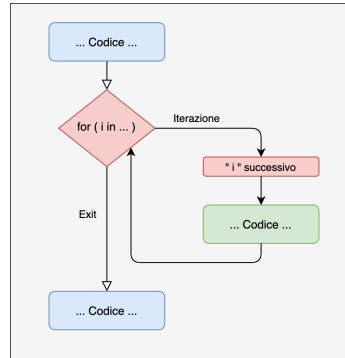
```
## [1] "Hello World"
```

```
print(msg)
```



# For

```
put_image("for_loop.png")
```



# 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"  
## [1] "hello world"  
## [1] "hello world"  
## [1] "hello world"  
## [1] "hello world"  
## [1] "hello world"  
## [1] "hello world"  
## [1] "hello world"  
## [1] "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 continuerà 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] 2.375011
```

```
sum(x)
```

```
## [1] 2.375011
```

**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] 0.01056065 0.54564893 0.04353437 0.05665122
```

medians

```
## [1] 0.1394716 0.5884152 -0.0161907 0.1422362
```

stds

```
## [1] 1.0843821 0.2873811 1.1165488 0.9802230
```

## `*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] 0.01056065 0.54564893 0.04353437 0.05665122
```

```
medians
```

```
## [1] 0.1394716 0.5884152 -0.0161907 0.1422362
```

```
stds
```

```
## [1] 1.0843821 0.2873811 1.1165488 0.9802230
```

## \*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_funs <- list(median = median, mean = mean, sd = sd)

lapply(my_list, function(vec) sapply(my_funs, function(fun) fun(vec)))
```

```
## [[1]]
##      median      mean      sd
## 0.13947164 0.01056065 1.08438211
##
## [[2]]
##      median      mean      sd
## 0.5884152 0.5456489 0.2873811
##
## [[3]]
##      median      mean      sd
## -0.01619070 0.04353437 1.11654877
##
## [[4]]
##      median      mean      sd
## 0.14223623 0.05665122 0.98022302
```

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]  0.052642516  0.017457282  0.009420277 -0.138599201  
## [5]  0.269506462 -0.217838736  0.205877728 -0.108158739  
## [9] -0.011914790 -0.078392799  
##  
## $vec2  
## [1]  0.26770462  0.13941433 -0.35500666  0.03797932  
## [5] -0.44079558 -0.37412150  0.49027779  0.22042182  
## [9] -0.10063045  0.11475631  
##  
## $vec3  
## [1]  0.21331084 -0.47536845 -0.09196819  0.10994311  
## [5] -0.35927185 -0.25928142 -0.19091158  0.39133314  
## [9]  0.29340299  0.36881141
```

## \*apply e funzioni anonime

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]  0.052642516  0.017457282  0.009420277 -0.138599201
## [5]  0.269506462 -0.217838736  0.205877728 -0.108158739
## [9] -0.011914790 -0.078392799
##
## $vec2
## [1]  0.26770462  0.13941433 -0.35500666  0.03797932
## [5] -0.44079558 -0.37412150  0.49027779  0.22042182
## [9] -0.10063045  0.11475631
##
## $vec3
## [1]  0.21331084 -0.47536845 -0.09196819  0.10994311
## [5] -0.35927185 -0.25928142 -0.19091158  0.39133314
## [9]  0.29340299  0.36881141
```

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] 0.5821141
##
## $vec2
## [1] 0.4811838
##
## $vec3
## [1] 0.5720184
```

```
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
## 0.5821141 0.4811838 0.5720184
```

```
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 0.6347566 0.74888845 0.78532924
## 2 0.5995714 0.62059815 0.09664996
## 3 0.5915344 0.12617716 0.48005022
## 4 0.4435149 0.51916315 0.68196151
## 5 0.8516206 0.04038825 0.21274655
## 6 0.3642754 0.10706233 0.31273698
```

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

```
## [1] 0.7229914 0.4389398 0.3992539 0.5482132 0.3682518
## [6] 0.2613582 0.7135201 0.7129709 0.6053914 0.6801638
```

# 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
## -0.58710950 -0.33946686 -0.03869854
```

# 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  
## 0.5821141 0.4811838 0.5720184
```

- `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] 10.720651  9.182059  9.816245  9.809967 10.480913
## [6]  8.701034 10.102578  8.460901  9.983495 10.848291
##
## [[2]]
## [1] 17.16171 20.93514 18.83946 22.50764 19.54066 22.20993
## [7] 20.04863 17.87077 21.43985 20.91768
##
## [[3]]
## [1] 33.89992 28.53802 31.17798 28.25672 33.82293 32.44419
## [7] 32.91359 31.09322 27.48670 26.06519
##
## [[4]]
## [1] 34.98977 32.63910 42.01902 34.53785 37.50011 39.99547
## [7] 35.43058 40.75667 41.47392 41.37619
```

# 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] 11.512546 9.716464 11.040640 11.015412 9.311823
## [6] 10.923788 10.714980 8.944473 8.955070 9.600674
##
## [[2]]
## [1] 14.57028 18.22725 19.56060 18.17379 21.05882 22.23353
## [7] 21.01393 15.92574 21.28903 17.27703
##
## [[3]]
## [1] 25.62603 27.96748 32.08184 26.45317 38.26193 29.27296
## [7] 33.54737 26.82987 26.46748 31.05091
##
## [[4]]
```

`*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*`

```
put_image("purrr.svg")
```

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 perchè 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. Perché?

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

```
x
```

```
## [1] 0.35207161 -1.91067805 -1.07334914 0.08112519
## [5] -0.43977057 -0.57662580 0.10790639 0.27626738
## [9] 1.01144188 0.33245125 1.36166104 0.09741107
## [13] 1.18328595 -0.05652952 -0.22144944 -0.33448789
## [17] -0.95383169 0.36793870 1.50590889 1.31642244
## [21] -0.32425033 0.33136430 -1.87996830 0.55974024
## [25] 0.20721675 -0.10526405 -1.06624739 0.88143566
## [29] -1.21067582 0.11048332 -0.03358909 -0.48546116
## [33] -0.39102601 2.05061622 -0.75343083 0.12023341
## [37] -0.35770272 0.32912772 0.72030149 1.21448075
## [41] -0.66934149 -0.65804905 -0.19367262 0.61817829
## [45] 0.04741731 0.53017148 0.17766372 -0.48207642
```

## Formula syntax e `aggregate()`

Un esempio utile è la funzione `aggregate()` molto interessante per applicare funzioni a dataframe. 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"
```



# 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)
```

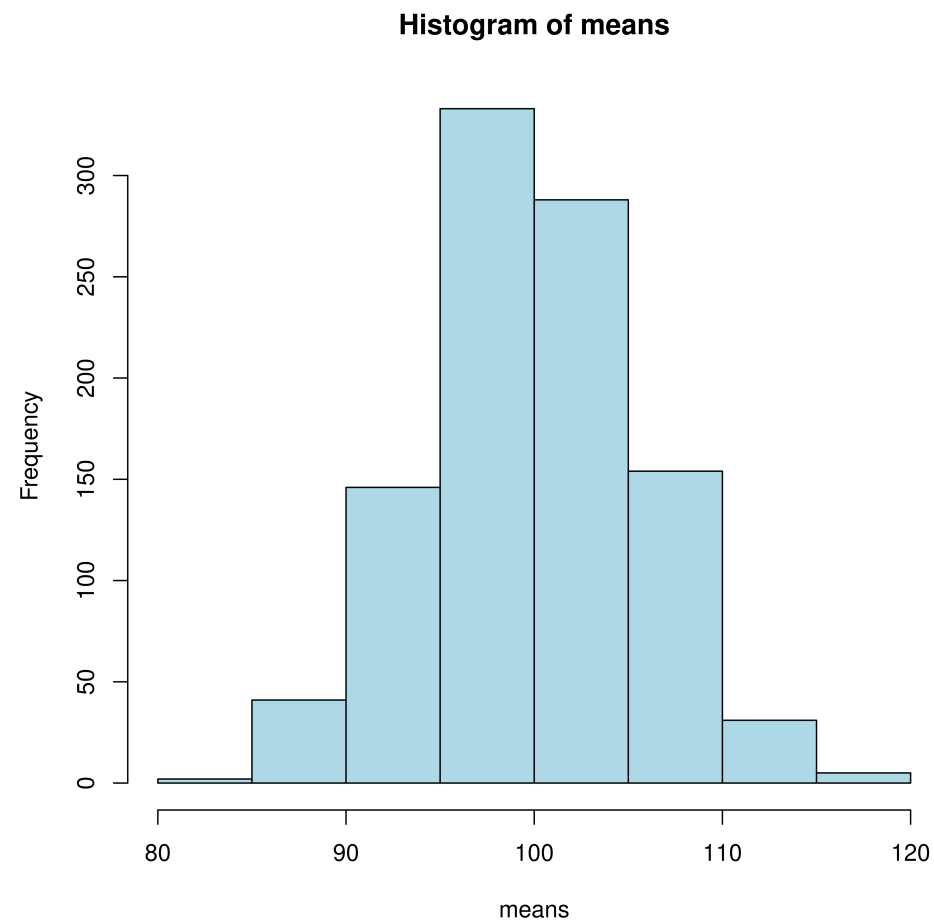
```
##      Species fac Sepal.Length
## 1    setosa  a    5.052941
## 2 versicolor  a    5.770588
## 3 virginica  a    6.756250
## 4    setosa  b    5.011765
## 5 versicolor  b    6.018750
## 6 virginica  b    6.447059
## 7    setosa  c    4.950000
## 8 versicolor  c    6.023529
## 9 virginica  c    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 distr  
nrep <- 1000  
nsample <- 30  
media <- 100  
ds <- 30  
  
means <- replicate(n = nrep, expr = {  
  mean(rnorm(nsample, media, ds))  
})
```



# 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.