# 3.3_apply

# `*apply` family

Immaginate di avere una lista di vettori, e di voler applicare la stessa funzione/i ad ogni elemento della lista:

- 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(n = 100, mean = 0, sd = 1),
    vec2=runif(n = 100, min = 0, max = 1),
    vec3=rnorm(n = 100, mean = 0, sd = 1),
    vec4=rnorm(n = 100, mean = 0, sd = 1)
)
```

# *apply family

## Applichiamo media, mediana e std

```r
# inizializzo i vettori
means=vector(mode = "numeric",
             length = length(my_list))
medians=vector(mode = "numeric",
               length = length(my_list))
stds=vector(mode = "numeric",
            length = length(my_list))

# Loop
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]])
}
```

## Risultato

```r
means
```

```
[1]  0.003015646  0.485865676
-0.146824159 -0.023617061
```

```r
medians
```

```
[1] -0.01903783  0.45754464
-0.13383616 -0.06665392
```

```r
stds
```

```
[1] 1.0607975 0.2968258
0.9327184 1.0175538
```

# `*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
```

```
       vec1          vec2          vec3          vec4
0.003015646  0.485865676 -0.146824159 -0.023617061
```

```
medians
```

```
       vec1         vec2         vec3         vec4
-0.01903783  0.45754464 -0.13383616 -0.06665392
```

```
stds
```

```
     vec1       vec2       vec3       vec4
1.0607975 0.2968258 0.9327184 1.0175538
```

# `*apply` family

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

- Cosa può essere la **lista**?
  - lista
  - dataframe
  - vettore
- Cosa può essere la **funzione**?
  - base o importata da un pacchetto
  - custom
  - anonima

# *apply family

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.

## Ciclo for

```
1  vec = 1:5
2  for(i in vec){
3    print(i)}
```

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

## sapply

```
1  vec = 1:5
2  res = sapply(vec, print)
```

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

# *apply family - funzione custom

Possiamo utlizzare anche funzioni create da noi:

```r
center_var = function(x) x - mean(x)

my_list = list(
    vec1 = runif(n = 10, min  = 0, max = 1),
    vec2 = runif(n = 10, min  = 0, max = 1),
    vec3 = runif(n = 10, min  = 0, max = 1)
)

lapply(my_list, center_var)
```

```
$vec1
 [1] -0.4358040239 -0.0006236574  0.3232146126  0.1820956785  0.2252020465
 [6]  0.3545645650  0.1623763425 -0.1299237365 -0.1542468924 -0.5268549349

$vec2
 [1]  0.2671576  0.3189470  0.4580312 -0.1555081 -0.1907819 -0.3242966
 [7] -0.3157417 -0.1315875  0.2056044 -0.1318244

$vec3
 [1] -0.4332595  0.1340095 -0.1395135  0.1190783 -0.1785085  0.1885190
 [7] -0.2708317  0.4011485  0.2832691 -0.1039113
```

# `*apply` family - implicito vs. esplicito

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.

## `sapply` implicito

```
1 vec = 1:5
2 res = sapply(vec, print)
```

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

## `sapply` esplicito

```
1 vec = 1:5
2 res = sapply(vec, function(i) print(i))
```

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

# *apply family - funzione anonima

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:

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

```
$vec1
 [1] -0.4358040239 -0.0006236574  0.3232146126  0.1820956785  0.2252020465
 [6]  0.3545645650  0.1623763425 -0.1299237365 -0.1542468924 -0.5268549349

$vec2
 [1]  0.2671576  0.3189470  0.4580312 -0.1555081 -0.1907819 -0.3242966
 [7] -0.3157417 -0.1315875  0.2056044 -0.1318244

$vec3
 [1] -0.4332595  0.1340095 -0.1395135  0.1190783 -0.1785085  0.1885190
 [7] -0.2708317  0.4011485  0.2832691 -0.1039113
```

**x** è solo un **placeholder** (analogo di i) e può essere qualsiasi lettera o nome!

# Tutte le tipologie di `*apply`

- `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:

```
1  res=lapply(my_list, mean)
2  res
```

```
$vec1
[1] 0.6414094

$vec2
[1] 0.4865552

$vec3
[1] 0.5824988
```

```
1  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:

```
1 res=sapply(my_list, mean)
2 res
```

```
     vec1      vec2      vec3 
0.6414094 0.4865552 0.5824988 
```

```
1 class(res)
```

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

# apply

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

```r
my_df =data.frame(x1 = runif(n = 5,min = 1,max = 10),
                  x2 = runif(n = 5,min = 3,max = 4))
my_df
```

```
        x1       x2
1 8.351905 3.201583
2 3.854663 3.867442
3 7.983905 3.733478
4 5.142101 3.263712
5 6.310265 3.089887
```

# apply

## Applico a tutte le righe (1) la funzione mean:

```
1 apply(my_df, MARGIN = 1, FUN = mean)
```

```
[1] 5.776744 3.861052 5.858691 4.202906 4.700076
```

## Applico a tutte le colonne (2) la funzione mean:

```
1 apply(my_df, MARGIN = 2, FUN = mean)
```

```
      x1       x2 
6.328568 3.431220 
```

# tapply

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

```r
vec=rnorm(75)

index=rep(c("a", "b", "c"), each = 25)

tapply(vec, INDEX = index, mean)
```

```
         a          b          c 
-0.5682334 -0.3593939 -0.1657899 
```

Qui dove *INDEX* è un vettore stringa o un fattore.

# tapply

In questo caso calcoliamo la media per ogni categoria d'età:

```
1 my_df = readr::read_csv("data/mydf_2.csv")
2 head(my_df)
```

```
# A tibble: 6 × 4
     id   age age_cat age_z
  <dbl> <dbl> <chr>   <dbl>
1     1    47 adulto  1.43
2     2    48 adulto  1.52
3     3    33 adulto  0.107
4     4    34 adulto  0.201
5     5    43 adulto  1.05
6     6    49 adulto  1.62
```

```
1 tapply(my_df$age, my_df$age_cat, mean)
```

```
adolescente      adulto     giovane
   17.00000    40.06250    25.55556
```

# 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:

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

```
     vec1      vec2      vec3
0.6414094 0.4865552 0.5824988
```

*FUN.VALUE = numeric(length = 1)*: indica che ogni risultato è un singolo valore numerico.

# mapply

**mapply** permette di gestire **più liste contemporaneamente** per scenari più complessi. Ad esempio vogliamo usare la funzione **rnorm()** e generare 4 con diverse medie e deviazioni stardard in combinazione:

```
1 medie=list(10, 20, 30, 40)
2 stds=list(1, 2, 3, 4)
3 mapply(function(x,y) rnorm(n = 5, mean = x, sd = y), medie, stds, SIMPLIFY
```

```
[[1]]
[1]  9.917191  9.907226  9.083554 10.190695  9.439073

[[2]]
[1] 18.59854 19.28850 20.39383 16.09438 21.53147

[[3]]
[1] 24.74802 29.79667 27.29165 33.23346 28.25332

[[4]]
[1] 36.53141 42.00140 35.98079 38.51464 37.85833
```

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

# mapply

**mapply(function(x,y) rnorm(n = 4, 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) si ottiene con un `for` usando più volte l'iteratore i:

```r
medie=list(10, 20, 30)
stds=list(1,2,3)
res=vector(mode = "list", length = length(medie)) # lista vuota

for(i in 1:length(medie)){
  res[[i]] = rnorm(n = 6, mean = medie[[i]], sd = stds[[i]])
}
res
```

```
[[1]]
[1] 10.380719  9.895178  9.001770  8.943968 10.189790 10.627328

[[2]]
[1] 14.34155 21.46691 20.90914 16.36203 20.91186 20.26559

[[3]]
[1] 31.47355 26.65955 27.31523 30.44201 31.96255 31.16261
```

# `replicate`

Questa funzione permette di ripetere un operazione n volte, senza però utilizzare un iteratore o un placeholder.

## `replicate(n, expr)`

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

# replicate

Campioniamo 1000 volte da una normale e facciamo la media

AKA distribuzione campionaria della media

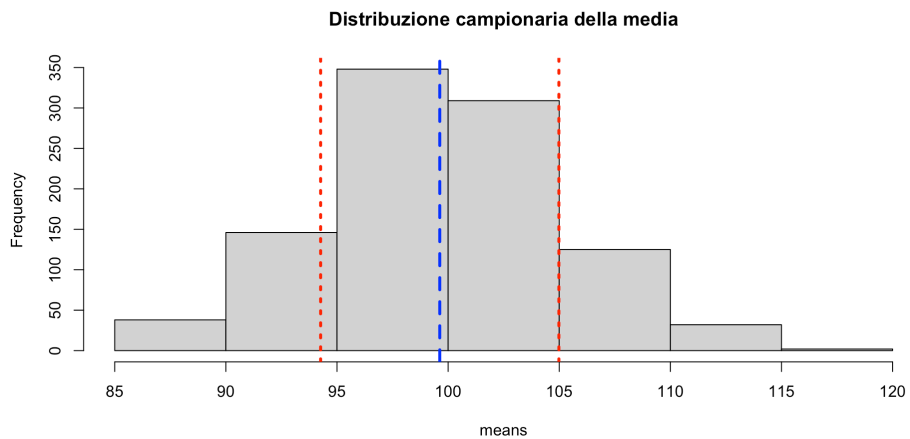
```r
nrep=1000

nsample=30

media=100

sd=30

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



# Ora facciamo un po’ di pratica!

Aprite e tenete aperto questo link:

https://etherpad.wikimedia.org/p/arca-corsoR