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

  ...

```r
my_list=list(
    vec1=rnorm(100),
    vec2=runif(100),
    vec3=rnorm(100),
    vec4=rnorm(100)
)
```

*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] -0.128529699  0.495316069
0.122400405  0.007952732
```

```r
medians
```

```
[1] -0.0529248  0.4621598
0.1522610 -0.1695774
```

```r
stds
```

```
[1] 1.0059772 0.2720942
0.9990584 0.9774053
```

`*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
-0.128529699  0.495316069  0.122400405  0.007952732
```

```
medians
```

```
      vec1        vec2        vec3        vec4
-0.0529248   0.4621598   0.1522610 -0.1695774
```

```
stds
```

```
     vec1       vec2       vec3       vec4
1.0059772 0.2720942 0.9990584 0.9774053
```

*apply family

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:

```r
# lista di funzioni
my_funs=list(median = median, mean = mean, sd = sd)

# applico ai vettori della mia lista le funzioni nella lista my_fun
lapply(my_list, function(vec) sapply(my_funs, function(fun) fun(vec)))
```

```
$vec1
    median       mean         sd
-0.0529248 -0.1285297  1.0059772

$vec2
   median      mean        sd
0.4621598 0.4953161 0.2720942

$vec3
   median      mean        sd
0.1522610 0.1224004 0.9990584
```

```
$vec4
      median          mean            sd
 0.16057373  0.00705873 0.07740527 6
```

`*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(10),
    vec2 = runif(10),
    vec3 = runif(10)
)

lapply(my_list, center_var)
```

```
$vec1
 [1] -0.25316054 -0.12448609 -0.28867871 -0.23084011 -0.42329234  0.28199159
 [7]  0.24561858  0.38549950  0.37065756  0.03669055

$vec2
 [1]  0.205941637  0.001943035 -0.051154863 -0.218376739  0.314510873
 [6] -0.152319605  0.497838579  0.063465268 -0.300237316 -0.361610870

$vec3
 [1] -0.38274728 -0.04131644  0.14867418 -0.20965653  0.25502829 -0.10610675
 [7] -0.18126103 -0.07377529  0.20025981  0.39090104
```

`*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] -0.25316054 -0.12448609 -0.28867871 -0.23084011 -0.42329234  0.28199159
 [7]  0.24561858  0.38549950  0.37065756  0.03669055

$vec2
 [1]  0.205941637  0.001943035 -0.051154863 -0.218376739  0.314510873
 [6] -0.152319605  0.497838579  0.063465268 -0.300237316 -0.361610870

$vec3
 [1] -0.38274728 -0.04131644  0.14867418 -0.20965653  0.25502829 -0.10610675
 [7] -0.18126103 -0.07377529  0.20025981  0.39090104
```

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] 0.5478854

$vec2
[1] 0.4964002

$vec3
[1] 0.5837265
```

```
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 
0.5478854 0.4964002 0.5837265 
```

```
1 class(res)
```

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

apply

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

```
1 my_df =data.frame(x1 = runif(5,1,10), x2 = runif(5,3,4))
2 my_df
```

```
        x1       x2
1 2.226692 3.910184
2 9.760877 3.911140
3 1.037286 3.562642
4 2.227523 3.508360
5 2.923148 3.699508
```

apply

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

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

```
[1] 3.068438 6.836008 2.299964 2.867942 3.311328
```

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

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

```
      x1       x2 
3.635105 3.718367 
```

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 
-0.1307892 -0.2045587 -0.2853245 
```

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    40 adulto        0.725
2     2    15 adolescente -1.70
3     3    14 adolescente -1.80
4     4    35 adulto        0.239
5     5    32 adulto       -0.0518
6     6    30 giovane      -0.246
```

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

```
adolescente      adulto     giovane
   16.00000    39.94118    24.90000
```

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
0.5478854 0.4964002 0.5837265
```

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] 10.826449  8.959037 10.875223 11.987235  9.982475

[[2]]
[1] 21.21998 18.61132 21.62333 21.50720 19.17973

[[3]]
[1] 31.89766 31.10035 27.41191 26.83203 37.80864

[[4]]
[1] 43.55759 37.59010 39.26600 40.98921 34.26420
```

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:

```
1 medie=list(10, 20, 30)
2 stds=list(1,2,3)
3 res=vector(mode = "list", length = length(medie)) # lista vuota
4
5 for(i in 1:length(medie)){
6   res[[i]] = rnorm(6, mean = medie[[i]], sd = stds[[i]])
7 }
8 res
```

```
[[1]]
[1]  8.539646  9.463136  9.064800 10.400413  9.480565  8.240795

[[2]]
[1] 24.43222 20.14991 20.63186 18.94143 20.73819 18.53196

[[3]]
[1] 36.08281 29.16968 27.09859 27.87979 27.73650 30.68962
```

`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

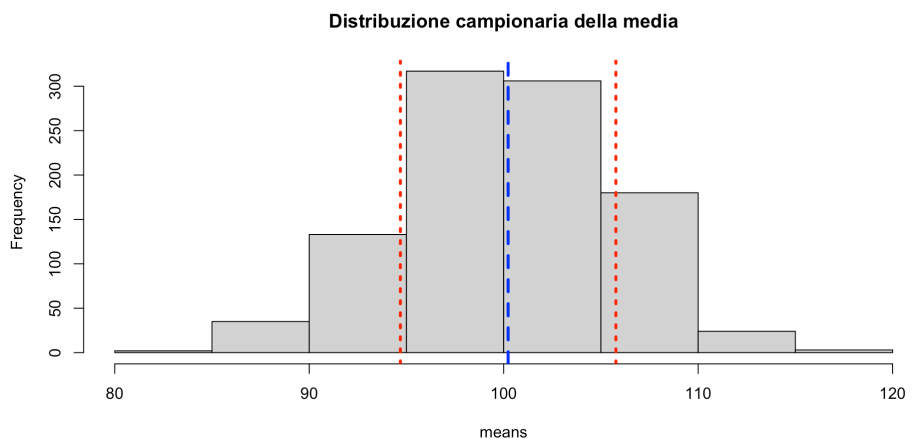
```r
nrep=1000

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