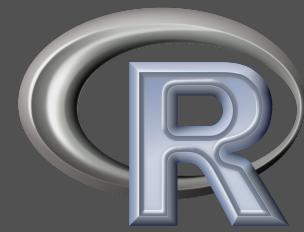




# ME115 - Linguagem R

Parte 04

1º semestre de 2023



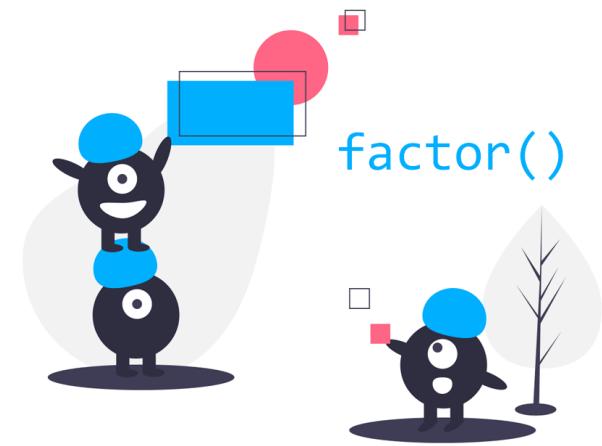
# Fatores

# Fatores

Fatores são usados para representar vetores de dados categóricos.

Pode ser pensado com um tipo especial de vetores de caracteres ou como vetores numérico, no qual cada inteiro tem um rótulo (*label*).

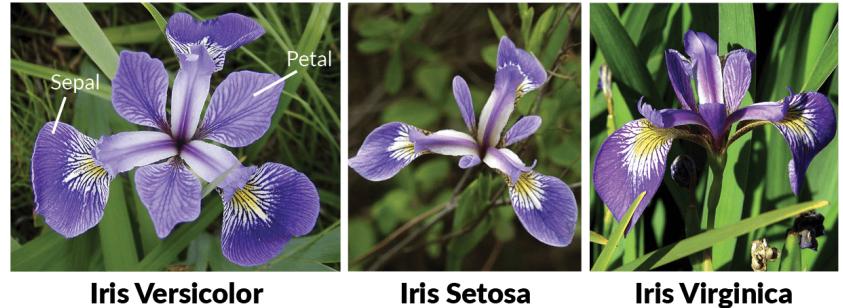
Fatores são fundamentais na modelagem estatística de dados discretos. Eles têm um tratamento especial por funções como `lm()` e `glm()`, que ainda serão muito utilizadas por vocês.



Fonte da imagem: <https://r-coder.com/factor-r/>

# Exemplo *iris*

Nesse famoso conjunto de dados chamado *iris*, temos representadas três espécies da flor *iris*: Setosa, Versicolor e Virginica.



```
data(iris)  
str(iris)
```

```
## 'data.frame': 150 obs. of 5 variables:  
## $ Sepal.Length: num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...  
## $ Sepal.Width : num 3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...  
## $ Petal.Length: num 1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...  
## $ Petal.Width : num 0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...  
## $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

# Exemplo *iris*

A classe da variável `Species` é um fator:



```
class(iris$Species)
```

```
## [1] "factor"
```

```
iris$Species[1:10]
```

```
## [1] setosa setosa setosa setosa setosa setosa setosa setosa setosa  
## Levels: setosa versicolor virginica
```

A função `table()` pode ser usada para contar a frequência de cada categoria:

```
table(iris$Species)
```

```
##  
##    setosa versicolor virginica  
##      50          50          50
```

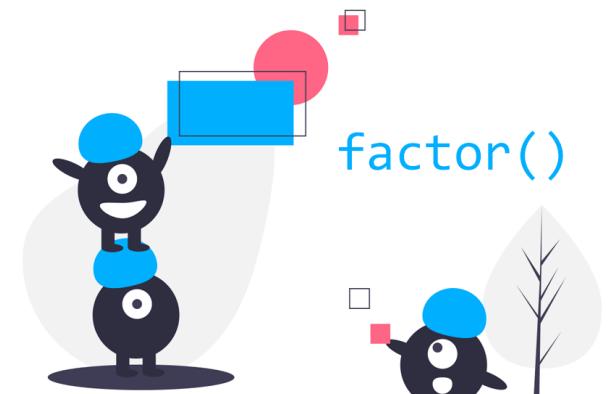
# Como criar um Fator no R

Fatores podem ser:

- **ordenados**: variável categórica ordinal
- **não ordenados**: variável categórica nominal

Para criar um fator usamos a função **factor()**:

```
factor(x = character(), levels, labels = levels,  
       exclude = NA, ordered = is.ordered(x), nmax = NA)
```



**Exemplo:** Suponha que temos um vetor com valores 0 e 1 que representam respostas binárias como “Não” e “Sim”, respectivamente:

```
x <- c(1, 0, 1, 1, 0, 1, 0, 0, 0, 1)  
factor(x)
```

```
## [1] 1 0 1 1 0 1 0 0 0 1  
## Levels: 0 1
```

# Exemplo

Modificando os rótulos (*labels*) de um fator:

```
x <- c(1, 0, 1, 1, 0, 1, 0, 0, 0, 1)
resposta <- factor(x, labels = c("Não", "Sim"))
resposta
```

```
## [1] Sim Não Sim Sim Não Sim Não Não Não Sim
## Levels: Não Sim
```

Você pode trocar a ordem dos níveis:

```
resposta <- factor(x, levels = c("1", "0"), labels = c("Sim", "Não"))
resposta
```

```
## [1] Sim Não Sim Sim Não Sim Não Não Não Sim
## Levels: Sim Não
```

# Fatores ordenados

```
days <- c("Friday", "Tuesday", "Thursday", "Monday", "Wednesday", "Monday",
         "Wednesday", "Monday", "Monday", "Wednesday", "Sunday", "Saturday")
class(days)

## [1] "character"
```

Convertendo o vetor days para fator:

```
my_days <- factor(days)
my_days

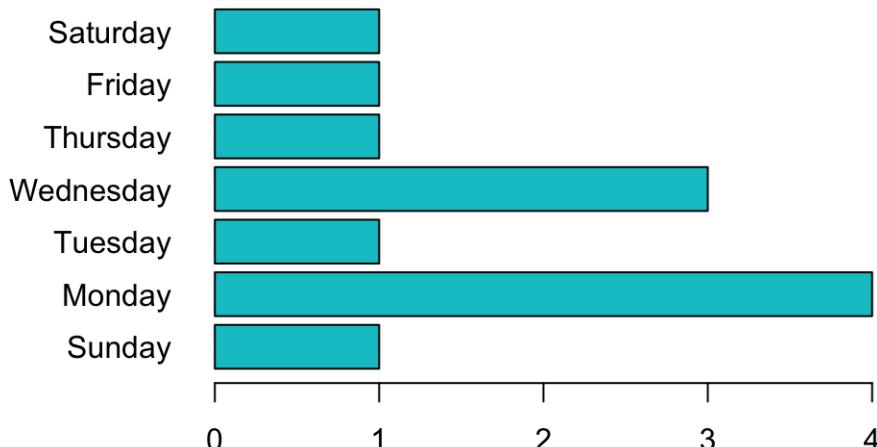
## [1] Friday      Tuesday     Thursday    Monday      Wednesday   Monday      Wednesday   Monday      Monday
## [10] Wednesday  Sunday      Saturday
## Levels: Friday Monday Saturday Sunday Thursday Tuesday Wednesday
```

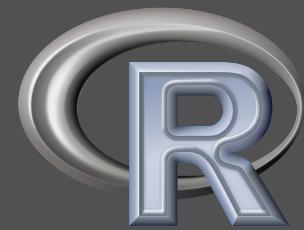
Veja que os níveis (`levels`) do fator é apresentado em ordem alfabética.

# Fatores ordenados

```
my_days <- factor(days, levels = c("Sunday", "Monday", "Tuesday", "Wednesday",
                                     "Thursday", "Friday", "Saturday"),
                     ordered = TRUE)
my_days[1:7]
```

```
## [1] Friday      Tuesday     Thursday    Monday      Wednesday Monday      Wednesday
## Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < Friday < Saturday
```





# Vetorização e Funcionais Família **apply**

# Vetorização

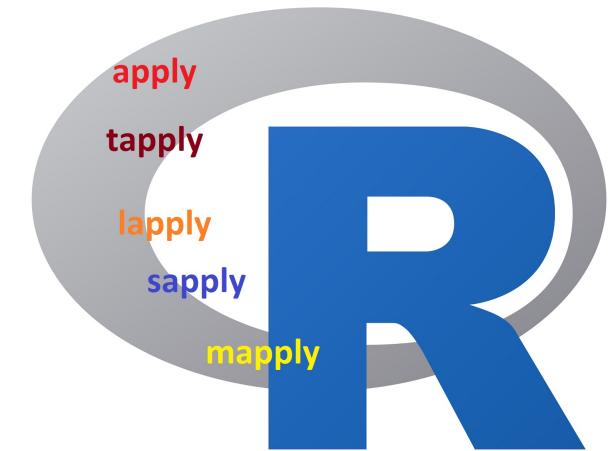
Embora *loops* do tipo `for` sejam um conceito importante para entender a lógica de programação, na prática, eles não são muito usados no R.

Você vai notar que à medida em que você aprende mais sobre a linguagem R, descobre que vetorização é preferível aos laços/loops, pois resulta em um código mais curto e claro.

**Vetorização:** uma função vetorizada é uma função que aplicará a mesma operação a cada um dos elementos do vetor.

```
sqrt(1:10)
```

```
## [1] 1.00 1.41 1.73 2.00 2.24 2.45 2.65 2.83 3.00 3.16
```

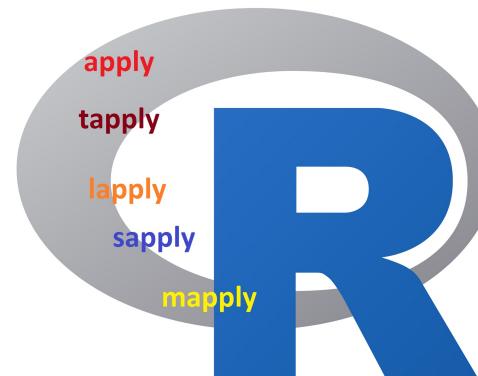


# Funcionais

Funcionais: funções que nos ajudam a aplicar a mesma função a cada entrada em um vetor, matriz, *data frame* ou lista.

Nessa aula, abordaremos os funcionais da família `apply`, disponíveis na base do R:

- `apply()`
- `lapply()`
- `sapply()`
- `tapply()`
- `mapply()`



Essas funções implementam interações (*loops*) de uma forma compacta, basicamente usando uma linha de comando.

# apply

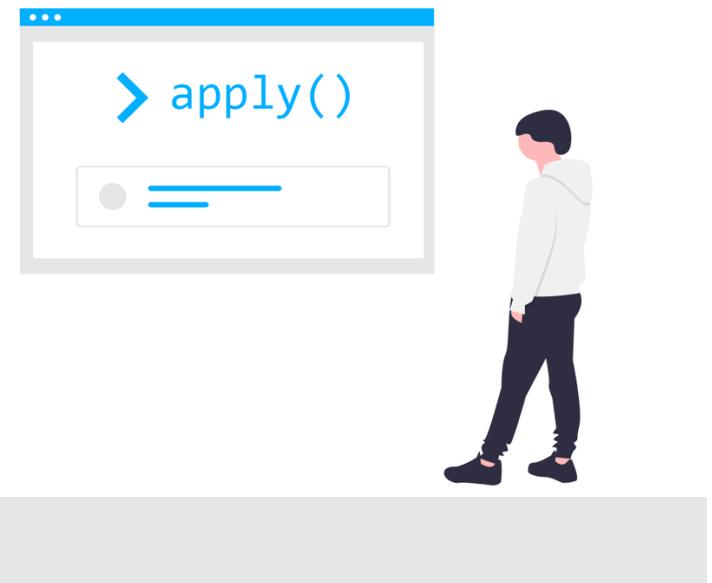
`apply()`: aplica uma função com relação às margens de uma matriz, *data frame* ou array.

A sintaxe da função é a seguinte:

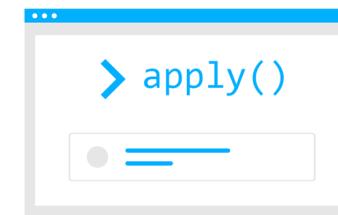
```
apply(X, MARGIN, FUN, ...)
```

- **X**: o vetor ou array com o qual você irá trabalhar
- **MARGIN**: a direção que a função será aplicada (**MARGIN** = 1 para as linhas e **MARGIN** = 2 para colunas).
- **FUN**: a função que será aplicada. Pode ser uma função pré existente, uma função definida pelo usuário ou uma função anônima.
- **...**: argumentos opcionais/extras da função a ser aplicada.

Fonte da imagem: <https://r-coder.com/apply-r/>



# apply



Veja um exemplo simples de uma matriz:

```
m <- matrix(rpois(15, lambda = 10), nrow = 3)
colnames(m) <- paste0("Coluna", 1:ncol(m)) # nomes das colunas
rownames(m) <- paste0("Linha", 1:nrow(m)) # nomes das linhas
m
```

```
##           Coluna1 Coluna2 Coluna3 Coluna4 Coluna5
## Linha1      18      14      17      4      14
## Linha2      16      11      14      8      8
## Linha3       9      15      10     12      9
```

```
apply(m, 1, sum)
```

```
## Linha1 Linha2 Linha3
##      67      57      55
```

# apply

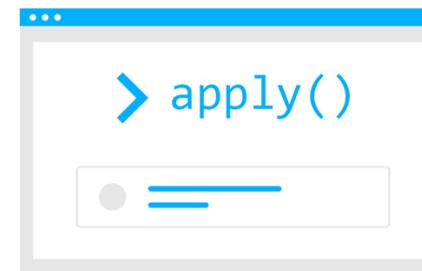
```
##           Coluna1 Coluna2 Coluna3 Coluna4 Coluna5
## Linha1      18      14      17      4      14
## Linha2      16      11      14      8      8
## Linha3      9       15      10     12      9
```

```
apply(m, 2, mean)
```

```
## Coluna1 Coluna2 Coluna3 Coluna4 Coluna5
##      14.3    13.3    13.7     8.0    10.3
```

Se houvessem dados faltantes (`NA`), seria necessário usar o argumento opcional `na.rm = TRUE` para que esses valores não impeçam o cálculo da soma/média:

```
apply(m, 2, mean, na.rm = TRUE)
```



# Somas e Médias de Linhas/Colunas

Apesar de ter usado o cálculo das somas das linhas e média de colunas usando o `apply()`, para esses casos específicos já temos uns atalhos para calcular somas ou médias de linhas/colunas de matrizes ou *data frames*:

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

Recomenda-se o uso dessas funções, pois elas são otimizadas e, portanto, mais rápidas. Apesar que você só notará essa diferença se estiver trabalhando com uma matriz bem grande.

# lapply

`lapply()`: aplica uma função em cada elemento de uma lista ou vetor, retornando uma lista.



```
lapply(X, FUN, ...)
```

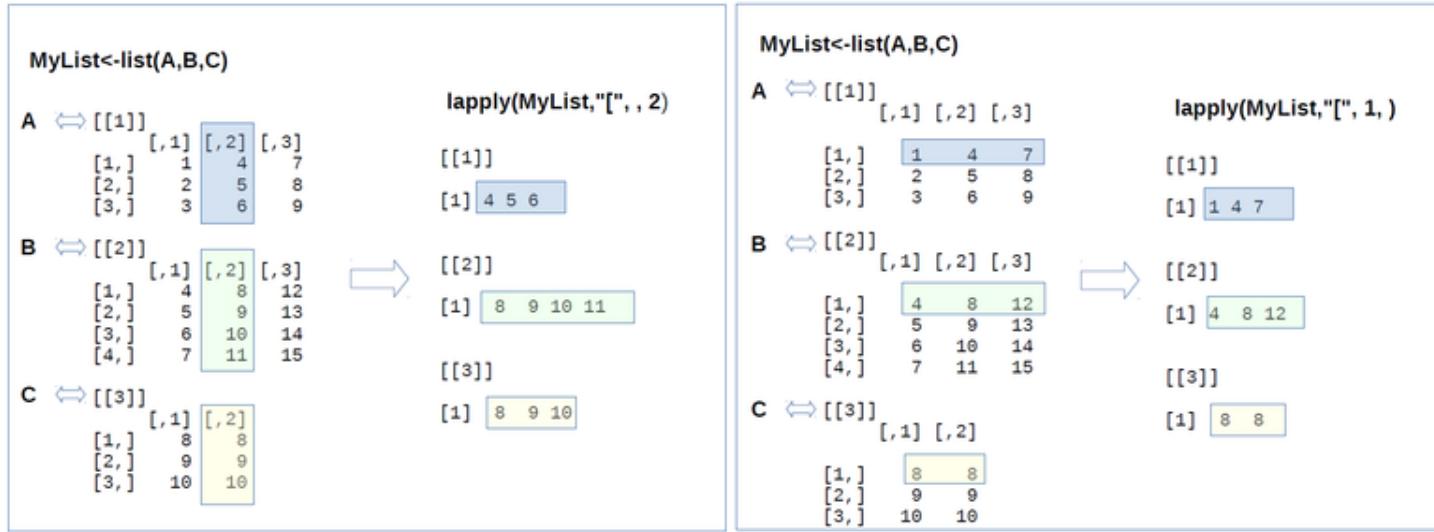
```
list1 <- list(x1 = rnorm(100), x2 = rnorm(1000, mean = 10))
lapply(list1, mean, na.rm = TRUE)
```

```
## $x1
## [1] -0.108
##
## $x2
## [1] 9.99
```

Fonte da imagem: <https://r-coder.com/lapply-r/>

# lapply

Veja mais um exemplo do uso do `lapply()`:



Note que a função (FUN) usada em `lapply()` é o operador "`[`", pois o objetivo é extraír a segunda coluna (imagem à esquerda) ou a primeira linha (imagem à direita) de cada elemento da lista `MyList`.

Fonte: <https://www.datacamp.com/community/tutorials/r-tutorial-apply-family>

# lapply

Exemplo: retornar o comprimento de cada elemento da lista.

```
lista2 <- list(A = 1:100, B = 2021, C = c("Yes", "No", "No"))
lapply(lista2, length)
```

```
## $A
## [1] 100
##
## $B
## [1] 1
##
## $C
## [1] 3
```



Veja que, às vezes, a saída como lista não parece ser a melhor opção.

# sapply

**sapply()**: a mesma função que **lapply()**, mas tenta simplificar a saída em um vetor ou array, em vez de uma lista.



```
sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)
```

**Exemplo:**

```
lista2 <- list(A = 1:100, B = 2021, C = c("Yes", "No", "No"))
sapply(lista2, length)
```

```
##   A   B   C
## 100   1   3
```

Dizemos que o **sapply()** é um *wrapper* do **lapply()**. Veja também a função **vapply()** e sua diferença em relação ao **sapply()**.

Fonte da imagem: <https://r-coder.com/sapply-r/>

# replicate

replicate: usado para avaliar repetidamente uma expressão.

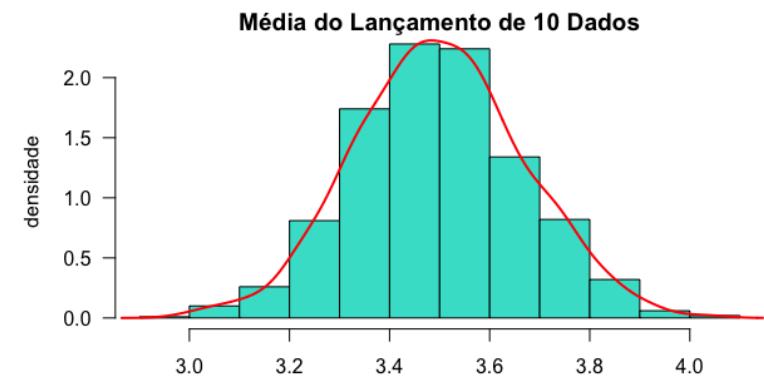
```
replicate(n, expr, simplify = "array")
```

**Exemplo:** Queremos gerar o lançamento de 100 dados e calcular a média. Repita isso 1000 e armazene essas médias em um vetor.

```
medias_dados <- replicate(1000, mean(sample(1:6, 100, replace = TRUE)))  
medias_dados[1:6]
```

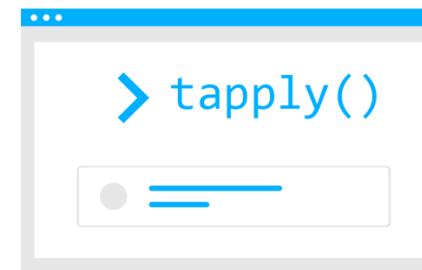
```
## [1] 3.75 3.31 3.57 3.38 3.50 3.14
```

**Nota:** Essa simulação representa um resultado muito importante: o **Teorema do Limite Central**, ou seja, a distribuição da média amostral.



# tapply

`tapply()`: aplica uma função em subconjuntos de um vetor. Tais subconjuntos (*subsets*) são definidos pela combinação das categorias de um ou mais fatores.



```
tapply(x, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

- `x`: um objeto para o qual pode-se obter subconjuntos.
- `INDEX`: uma lista de um ou mais fatores de mesmo comprimento que `x`, para definir os grupos.

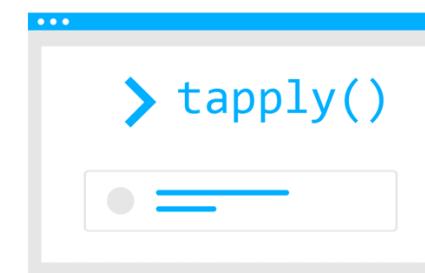
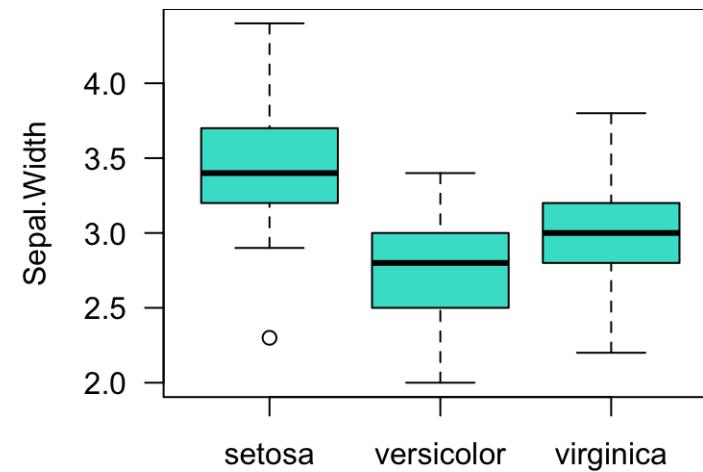
Fonte da imagem: <https://r-coder.com/tapply-r/>

# tapply

Vamos voltar no exemplo dos dados `iris` e calcular a mediana de `Sepal.Width` para cada espécie:

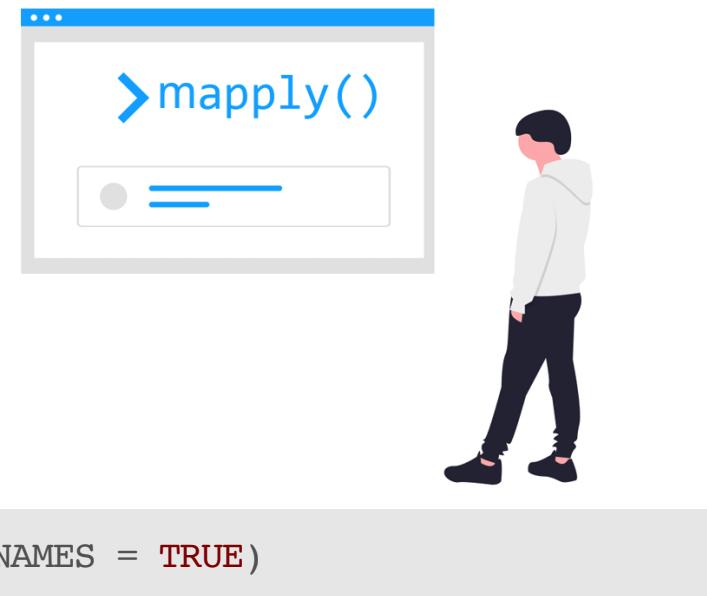
```
tapply(iris$Sepal.Width, iris$Species, median)
```

```
##      setosa versicolor  virginica
##      3.4       2.8       3.0
```



# mapply

`mapply()`: versão multivariada do `sapply()`. Aplica uma função com os primeiros elementos de cada argumento em ..., depois os segundos elementos, e assim por diante.



```
mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
```

- `FUN` é a função a ser aplicada
- `...` são os argumentos que serão iterados na função
- `MoreArgs` é uma lista de outros argumentos para a função
- `SIMPLIFY` indica se o resultado deve ser simplificado

Argumentos com comprimentos diferentes são reciclados.

Fonte: Imagem modificada de <https://r-coder.com/sapply-r/>

# mapply

```
list(rep(1, 5), rep(2, 3), rep(3, 7))
```

Nesse caso, podemos usar o `mapply()`:

```
mapply(rep, x = 1:3, times = c(5, 3, 7))
```

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

# mapply

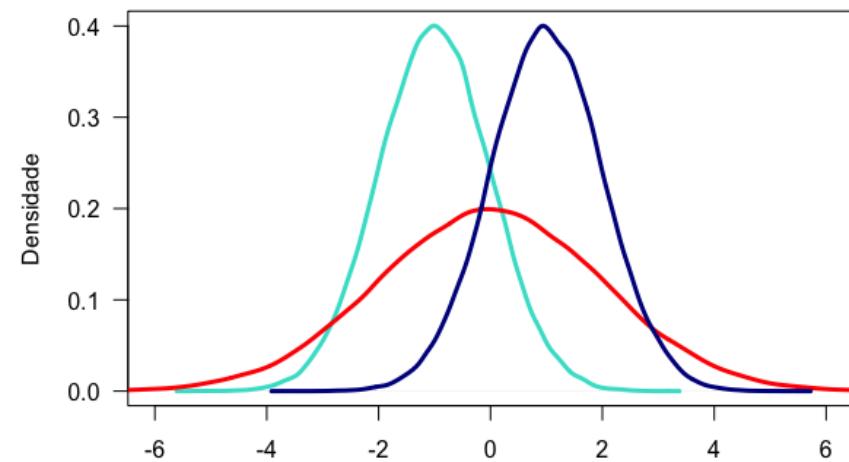
Suponha que nós queremos gerar 100 observações de variáveis aleatórias normais com as seguintes médias e variâncias:  $N(-1, 1)$ ,  $N(0, 4)$  e  $N(1, 1)$ .

```
normais <- mapply(rnorm, n = 100, mean = -1:1, sd = c(1, 2, 1))

apply(normais, 2, function(x) c(Media = mean(x), Variancia = var(x)))
```

```
##          [,1]  [,2]  [,3]
## Media     -1.072 -0.197 1.02
## Variancia  0.905  4.196 1.04
```

Note que a função usada no `apply()` foi criada diretamente dentro do comando.



# Vetorizar uma função

**Vetorizar:** transformar uma função que normalmente aceita somente escalares como argumentos em uma nova função que aceita vetores de argumentos.

Como vimos, a função `mapply()` pode ser usada para vetorizar uma função.

```
normais <- mapply(rnorm, n = 100, mean = -1:1, sd = c(1, 2, 1))
```

Outra alternativa é a função `vectorize()`, que automaticamente pode criar uma versão vetorizada da sua função.

```
vrnorm <- Vectorize(rnorm, c("n", "mean", "sd"))
vnormais <- vrnorm(n = 100, mean = -1:1, sd = c(1, 2, 1))
apply(vnormais, 2, function(x) c(Media = mean(x), Variancia = var(x)))
```

```
##          [,1]   [,2]   [,3]
## Media     -1.072 -0.197  1.02
## Variancia  0.905  4.196  1.04
```

# Pacote **purrr** e a família **map**

O pacote **purrr** também contém funcionais que podem substituir *loops* e aplicam uma função em cada elemento de um vetor.

Existe uma função para cada tipo de vetor de saída, que é determinado pelo sufixo da função `map_*`( ).

- `map()` retorna uma lista.
- `map_lgl()` retorna um vetor lógico.
- `map_int()` retorna um vetor de inteiros.
- `map_dbl()` retorna um vetor de valores numéricos.
- `map_chr()` retorna um vetor de caracteres.

Não cobriremos essas funções nessa aula, mas estudando a família `apply`, vocês conseguirão entender e usar a família `map` também.



# Referências

Algumas referências utilizadas para a construção desse material:

[R Programming – Roger Peng](#)

[Introdução à Ciência de Dados - Rafael A. Irizarry](#)

[R for Data Science - Hadley Wickham e Garrett Grolemund](#)

[R Coder - Introduction to R](#)



Slides produzidos pela profa. Tatiana Benaglia.