Programação funcional no R

Fernando P. Mayer

2022-04-05

Introdução

Utilidade da programação funcional

- **Programação funcional** é quando uma função chama uma outra função para ser aplicada repetidamente percorrendo elementos de um objeto.
- O recurso é útil para fazer tarefas em **série** ou **batelada**.
- Exemplos:
  - Importar vários arquivos em um diretório.
  - Tratar as imagens de um diretório.
  - Realizar a mesma análise para todas as UFs.
  - Fazer o ajuste de regressão polinomial de grau 1 até 5.

Programação funcional no R

- No R básico, a programação funcional é com a família `apply`.
- No `tidyverse` a programação funcional está no `purrr`.
- A principal é a função `map()` e suas variações.
- Além disso, tem
  - Funções para tratamento de excessões.
  - Acumular e reduzir.
  - Aninhar e aplanar objetos.

Estruturas primitivas de repetição e seleção condicional

Estrutura de repetição `for()`

Serve para repetir um ou mais comandos diversas vezes. Para ver como funciona, considere o seguinte exemplo:

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

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

O resultado é a chamada do comando `print()` para cada valor que o índice `i` recebe (nesse caso `i` recebe os valores de 1 a 5).

A sintaxe será sempre nesse formato:

```
for(<índice> in <valores>){
    <comandos>
}
```

Veja outro exemplo em como podemos aplicar o índice:

```
x <- 100:200
for(j in 1:5){
    print(x[j])
}
```

```
# [1] 100
# [1] 101
# [1] 102
# [1] 103
# [1] 104
```

Veja que o índice não precisa ser `i`, na verdade pode ser qualquer letra ou palavra. Nesse caso, veja que utilizamos os valores como índice para selecionar elementos de `x` naquelas posições específicas.

Um outro exemplo seria se quisessemos imprimir o quadrado de alguns números (não necessariamente em sequência):

```
for(i in c(2, 9, 4, 6)){
    print(i^2)
}
```

```
# [1] 4
# [1] 81
# [1] 16
# [1] 36
```

Ou mesmo imprimir caracteres a partir de um vetor de caracteres:

```
for(veiculos in c("carro", "ônibus", "trem", "bicicleta")){
    print(veiculos)
}
```

```
# [1] "carro"
# [1] "ônibus"
# [1] "trem"
# [1] "bicicleta"
```

Exemplo: cálculo de notas de uma disciplina.

```
library(tidyverse)
da <- tibble(
    matricula = c(256, 487, 965,
                  125, 458, 874, 963),
    nome = c("João", "Vanessa", "Tiago",
             "Luana", "Gisele", "Pedro",
             "André"),
    curso = c("Mat", "Mat", "Est", "Est",
              "Est", "Mat", "Est"),
    prova1 = c(80, 75, 95, 70, 45, 55, 30),
    prova2 = c(90, 75, 80, 85, 50, 75, NA),
    prova3 = c(80, 75, 75, 50, NA, 90, 30),
    faltas = c(4, 4, 0, 8, 16, 0, 20))
## Substitui NA por 0
da <- da %>%
    mutate(across(prova1:prova3,
                  ~replace_na(., 0)))
da
```

```
# # A tibble: 7 × 7
#   matricula nome    curso prova1 prova2 prova3 faltas
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl>
# 1       256 João    Mat       80     90     80      4
# 2       487 Vanessa Mat       75     75     75      4
# 3       965 Tiago   Est       95     80     75      0
# 4       125 Luana   Est       70     85     50      8
# 5       458 Gisele  Est       45     50      0     16
# 6       874 Pedro   Mat       55     75     90      0
# 7       963 André   Est       30      0     30     20
```

Para calcular as médias das 3 provas, precisamos inicialmente de um vetor para armazenar os resultados. Esse vetor pode ser um novo objeto ou uma nova coluna no dataframe

```
## Aqui vamos criar uma nova coluna no dataframe, contendo apenas o
## valor 0
da$media <- 0 # note que aqui será usada a regra da reciclagem, ou
                # seja, o valor zero será repetido até completar tod
as
                # as linhas do dataframe
## Estrutura de repetição para calcular a média
for(i in 1:nrow(da)){
    ## Aqui, cada linha i da coluna media sera substituida pelo
    ## respectivo valor da media caculada
    da$media[i] <- sum(da[i, c("prova1", "prova2", "prova3")])/3
}
## Confere os resultados
da
```

```
# # A tibble: 7 × 8
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
```

Agora podemos melhorar o código, tornando-o mais **genérico**. Dessa forma fica mais fácil fazer alterações e procurar erros. Uma forma de melhorar o código acima é generalizando alguns passos.

```
## Armazenamos o número de linhas no dataframe
nlinhas <- nrow(da)
## Identificamos as colunas de interesse no cálculo da média, e
## armazenamos em um objeto separado
provas <- c("prova1", "prova2", "prova3")
## Sabendo o número de provas, fica mais fácil dividir pelo total no
## cálculo da média
nprovas <- length(provas)
## Cria uma nova coluna apenas para comparar o cálculo com o anterior
da$media <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    da$media[i] <- sum(da[i, provas])/nprovas
}
## Confere
da
```

```
# # A tibble: 7 × 8
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
```

Ainda podemos melhorar (leia-se: **otimizar**) o código, se utilizarmos funções prontas do R. No caso da média isso é possível pois a função `mean()` já existe. Em seguida veremos como fazer quando o cálculo que estamos utilizando não está implementado em nenhuma função pronta do R.

```
## Cria uma nova coluna apenas para comparação
da$media <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    da$media[i] <- mean(as.numeric(da[i, provas]))
}
## Confere
da
```

```
# # A tibble: 7 × 8
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
```

```
## A única diferença é que aqui precisamos transformar cada linha em
 um
## vetor de números com as.numeric(), pois
da[1, provas]
```

```
# # A tibble: 1 × 3
#   prova1 prova2 prova3
#    <dbl>  <dbl>  <dbl>
# 1     80     90     80
```

```
## é um data.frame:
class(da[1, provas])
```

```
# [1] "tbl_df"     "tbl"        "data.frame"
```

No caso acima vimos que não era necessário calcular a média através de `soma/total` porque existe uma função pronta no R para fazer esse cálculo. Mas, e se quisessemos, por exemplo, calcular a Coeficiente de Variação (CV) entre as notas das três provas de cada aluno? Uma busca por

```
help.search("coefficient of variation")
```

não retorna nenhuma função (dos pacotes básicos) para fazer esse cálculo. O motivo é simples: como é uma conta simples de fazer não há necessidade de se criar uma função extra dentro dos pacotes. No entanto, nós podemos criar uma função que calcule o CV, e usá-la para o nosso propósito

```
cv <- function(x){
    desv.pad <- sd(x)
    med <- mean(x)
    cv <- desv.pad/med
    return(cv)
}
```

NOTA: na função criada acima o único argumento que usamos foi `x`, que neste caso deve ser um vetor de números para o cálculo do CV. Os argumentos colocados dentro de `function()` devem ser apropriados para o propósito de cada função.

Antes de aplicar a função dentro de um `for()` devemos testá-la para ver se ela está funcioanando de maneira correta. Por exemplo, o CV para as notas do primeiro aluno pode ser calculado "manualmente" por

```
sd(as.numeric(da[1, provas]))/mean(as.numeric(da[1, provas]))
```

```
# [1] 0.06928203
```

E através da função, o resultado é

```
cv(as.numeric(da[1, provas]))
```

```
# [1] 0.06928203
```

o que mostra que a função está funcionando corretamente, e podemos aplicá-la em todas as linhas usando a repetição

```
## Cria uma nova coluna para o CV
da$CV <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    da$CV[i] <- cv(as.numeric(da[i, provas]))
}
## Confere
da
```

```
# # A tibble: 7 × 9
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 1 more variable: CV <dbl>
```

Podemos agora querer calcular as médias ponderadas para as provas. Por exemplo:

- Prova 1: peso 3
- Prova 2: peso 3
- Prova 3: peso 4

Usando a fórmula:

$$\bar{x} = \frac{1}{N}\sum_{i=1}^{n} x_i \cdot w_i$$

onde w_i são os pesos, e $N = \sum_{i=1}^{n} w_i$ é a soma dos pesos. Como já vimos que criar uma função é uma forma mais prática (e elegante) de executar determinada tarefa, vamos criar uma função que calcule as médias ponderadas.

```
med.pond <- function(notas, pesos){
    ## Multiplica o valor de cada prova pelo seu peso
    pond <- notas * pesos
    ## Calcula o valor total dos pesos
    peso.total <- sum(pesos)
    ## Calcula a soma da ponderação
    sum.pond <- sum(pond)
    ## Finalmente calcula a média ponderada
    saida <- sum.pond/peso.total
    return(saida)
}
```

Antes de aplicar a função para o caso geral, sempre é importante testar e conferir o resultado em um caso menor. Podemos verificar o resultado da média ponderada para o primeiro aluno

```
sum(da[1, provas] * c(3, 3, 4))/10
```

```
# [1] 83
```

e testar a função para o mesmo caso

```
med.pond(notas = da[1, provas], pesos = c(3, 3, 4))
```

```
# [1] 83
```

Como o resultado é o mesmo podemos aplicar a função para todas as linhas através do `for()`

```
## Cria uma nova coluna para a média ponderada
da$MP <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    da$MP[i] <- med.pond(da[i, provas], pesos = c(3, 3, 4))
}
## Confere
da
```

```
# # A tibble: 7 × 10
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 2 more variables: CV <dbl>, MP <dbl>
```

NOTA: uma função para calcular a média ponderada já existe implementada no R. Veja `?weighted.mean()` e confira os resultados obtidos aqui

Repare na construção da função acima: agora usamos dois argumentos, `notas` e `pesos`, pois precisamos dos dois vetores para calcular a média ponderada. Repare também que ambos argumentos não possuem um valor padrão. Poderíamos, por exemplo, assumir valores padrão para os pesos, e deixar para que o usuário mude apenas se achar necessário.

```
## Atribuindo pesos iguais para as provas como padrão
med.pond <- function(notas, pesos = rep(1, length(notas))){
    ## Multiplica o valor de cada prova pelo seu peso
    pond <- notas * pesos
    ## Calcula o valor total dos pesos
    peso.total <- sum(pesos)
    ## Calcula a soma da ponderação
    sum.pond <- sum(pond)
    ## Finalmente calcula a média ponderada
    saida <- sum.pond/peso.total
    return(saida)
}
```

Repare que neste caso, como os pesos são iguais, a chamada da função sem alterar o argumento `pesos` gera o mesmo resultado do cálculo da média comum.

```
## Cria uma nova coluna para a média ponderada para comparação
da$MP <- 0
## A estrutura de repetição fica
for(i in 1:nlinhas){
    da$MP[i] <- med.pond(da[i, provas])
}
## Confere
da
```

```
# # A tibble: 7 × 10
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 2 more variables: CV <dbl>, MP <dbl>
```

Estrutura de seleção `if()`

Uma estrutura de seleção serve para executar algum comando apenas se alguma condição (em forma de **expressão condicional**) seja satisfeita. Geralmente é utilizada dentro de um `for()`.

No exemplo inicial poderíamos querer imprimir um resultado caso satisfaça determinada condição. Por exemplo, se o valor de `x` for menor ou igual a 105, então imprima um texto informando isso.

```
x <- 100:200
for(j in 1:5) {
    if(x[j] <= 103) {
        print(cbind(x = x[j], msg = "Menor ou igual a 103"))
    }
}
```

```
#      x     msg
# [1,] "100" "Menor ou igual a 103"
#      x     msg
# [1,] "101" "Menor ou igual a 103"
#      x     msg
# [1,] "102" "Menor ou igual a 103"
#      x     msg
# [1,] "103" "Menor ou igual a 103"
```

Mas também podemos considerar o que aconteceria caso contrário. Por exemplo, se o valor de `x` for maior do que 105, então imprima outro texto.

```
x <- 100:200
for(j in 1:5){
    if(x[j] <= 103){
        print(cbind(x = x[j], msg = "Menor ou igual a 103"))
    } else{
        print(cbind(x = x[j], msg = "Maior do que 103"))
    }
}
```

```
#      x     msg
# [1,] "100" "Menor ou igual a 103"
#      x     msg
# [1,] "101" "Menor ou igual a 103"
#      x     msg
# [1,] "102" "Menor ou igual a 103"
#      x     msg
# [1,] "103" "Menor ou igual a 103"
#      x     msg
# [1,] "104" "Maior do que 103"
```

A sintaxe será sempre no formato:

```
if(<condição>){
    <comandos que satisfazem a condição>
} else{
   <comandos que não satisfazem a condição>
}
```

Como vimos acima, a especificação do `else{}` não é obrigatória.

Voltando ao exemplo das notas, podemos adicionar uma coluna com a condição do aluno: `aprovado` ou `reprovado` de acordo com a sua nota. Para isso precisamos criar uma condição (nesse caso se a nota é maior do que 7), e verificar se ela é verdadeira.

```
## Nova coluna para armazenar a situacao
da$res <- NA # aqui usamos NA porque o resultado será um
             # caracter
## Estrutura de repetição
for(i in 1:nlinhas){
    ## Estrutura de seleção (usando a média ponderada)
    if(da$MP[i] >= 70){
        da$res[i] <- "aprovado"
    } else{
        da$res[i] <- "reprovado"
    }
}
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

O modo do R: vetorização

As funções vetorizadas do R, além de facilitar e resumir a execução de tarefas repetitivas, também são computacionalmente mais eficientes, *i.e.* o tempo de execução das rotinas é muito mais rápido.

Já vimos que a **regra da reciclagem** é uma forma de vetorizar cálculos no R. Os cálculos feitos com funções vetorizadas (ou usando a regra de reciclagem) são muito mais eficientes (e preferíveis) no R. Por exemplo, podemos criar um vetor muito grande de números e querer calcular o quadrado de cada número. Se pensássemos em usar uma estrutura de repetição, o cálculo seria o seguinte:

```
## Vetor com uma sequência de 1 a 1.000.000
x <- 1:1e6
## Calcula o quadrado de cada número da sequência em x usando for()
y1 <- numeric(length(x)) # vetor de mesmo comprimento de x que vai
                         # receber os resultados
for(i in 1:length(x)){
    y1[i] <- x[i]^2
}
```

Mas, da forma vetorial e usando a regra da reciclagem, a mesma operação pode ser feita apenas com

```
## Calcula o quadrado de cada número da sequência em x usando a regra
da
## reciclagem
y2 <- x^2
## Confere os resultados
identical(y1, y2)
```

```
# [1] TRUE
```

Note que os resultados são exatamente iguais, mas então porque se prefere o formato vetorial? Primeiro porque é muito mais simples de escrever, e segundo (e principalmente) porque a forma vetorizada é muito mais **eficiente computacionalmente**. A eficiência computacional pode ser medida de várias formas (alocação de memória, tempo de execução, etc), mas apenas para comparação, vamos medir o tempo de execução destas mesmas operações usando o `for()` e usando a regra da reciclagem.

```
## Tempo de execução usando for()
y1 <- numeric(length(x))
st1 <- system.time(
    for(i in 1:length(x)){
        y1[i] <- x[i]^2
    }
)

## Tempo de execução usando a regra da reciclagem
st2 <- system.time(
    y2 <- x^2
)

rbind(st1, st2)
```

```
#     user.self sys.self elapsed user.child sys.child
# st1     0.061    0.000   0.062          0         0
# st2     0.254    0.007   0.261          0         0
```

Olhando o resultado de `elapsed`, que é o tempo total de execução de uma função medido por `system.time()`, notamos que usando a regra da reciclagem, o cálculo é aproximadamente 0.062/0.261 = 0.24 vezes mais rápido. Claramente esse é só um exemplo de um cálculo muito simples. Mas em situações mais complexas, a diferença entro o tempo de execução das duas formas pode ser muito maior.

Uma nota de precaução

Existem duas formas básicas de tornar um loop `for` no R mais rápido:

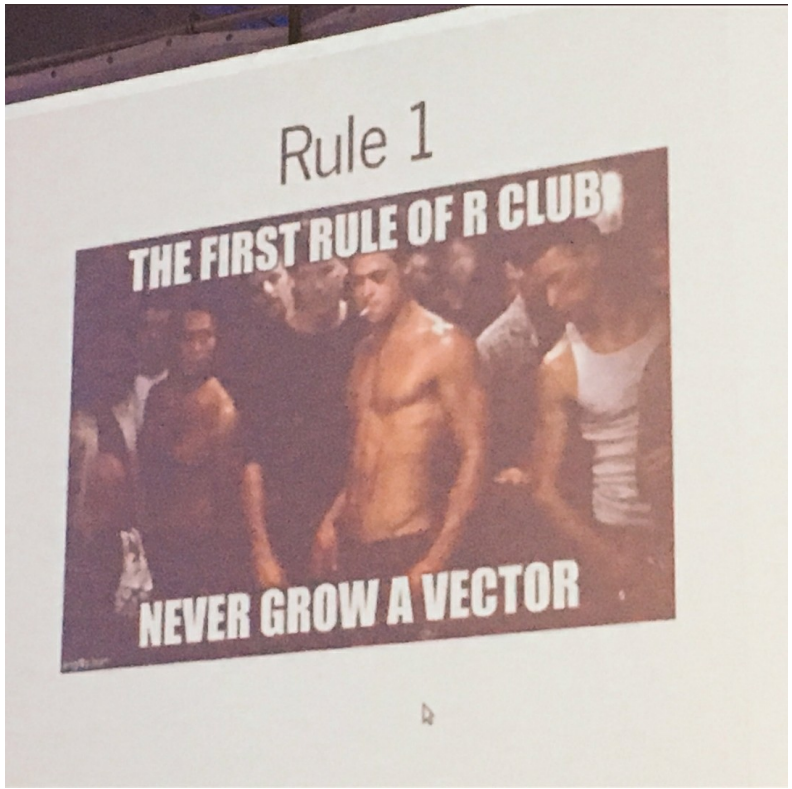
1. Faça o máximo possível fora do loop
2. Crie um objeto com tamanho suficiente para armazenar *todos* os resultados do loop **antes** de executá-lo

Veja este exemplo:

```
## Vetor com uma sequência de 1 a 1.000.000
x <- 1:1e6

## Cria um objeto de armazenamento com o mesmo tamanho do resultad
o
st1 <- system.time({
    out1 <- numeric(length(x))
    for(i in 1:length(x)){
        out1[i] <- x[i]^2
    }
})

## Cria um objeto de tamanho "zero" e vai "crescendo" esse vetor
st2 <- system.time({
    out2 <- numeric(0)
    for(i in 1:length(x)){
        out2[i] <- x[i]^2
    }
})

## Cria um objeto de tamanho "zero" e cresce o vetor usando a funç
ão c()
## NUNCA faça isso!!
st3 <- system.time({
    out3 <- numeric(0)
    for(i in 1:length(x)){
        out3 <- c(out3, x[i]^2)
    }
})

identical(out1, out2, out3)


rbind(st1, st2, st3)

#     user.self sys.self  elapsed user.child sys.child
# st1     0.061    0.000    0.062          0         0
# st2     0.254    0.007    0.261          0         0
# st3  1473.201   18.146 1497.057          0         0
```

Essa simples diferença gera um aumento de tempo de execução da segunda forma, em relação à primeira, de aproximadamente 0.261/0.062 = 4.21 vezes. Já utilizando a terceira forma, "crescendo" o vetor com a função `c()`, o aumento de tempo (em relação ao primeiro) é de aproximadamente 1497.057/0.062 = 24146 vezes! Isso acontece porque o vetor `out` precisa ter seu tamanho aumentado com um elemento a cada iteração. Para fazer isso, o R precisa encontrar um espaço na memória que possa armazenar o objeto maior. É necessário então copiar o vetor de saída e apagar sua versão anterior antes de seguir para o próximo loop. Ao final, foi necessário escrever um milhão de vezes na memória do computador.

Já no primeiro caso, o tamanho do vetor de armazenamento nunca muda, e a memória para esse vetor já foi alocada previamente, de uma única vez.



Voltando ao exemplo das notas, por exemplo, o cálculo da média simples poderia ser feita diretamente com a função `apply()`

```
da$media <- apply(X = da[, provas], MARGIN = 1, FUN = mean)
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

As médias ponderadas poderiam ser calculadas da mesma forma, e usando a função que criamos anteriormente

```
da$MP <- apply(X = da[, provas], MARGIN = 1, FUN = med.pond)
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

Mas note que como temos o argumento `pesos` especificado com um padrão, devemos alterar na própria função `apply()`

```
da$MP <- apply(X = da[, provas], MARGIN = 1,
               FUN = med.pond, pesos = c(3, 3, 4))
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

> NOTA: veja que isso é possível devido à presença do argumento `...` na função `apply()`, que permite passar argumentos de outras funções dentro dela.

Também poderíamos usar a função `weighted.mean()` implementada no R

```
da$MP <- apply(X = da[, provas], MARGIN = 1,
               FUN = weighted.mean, w = c(3, 3, 4))
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

O Coeficiente de Variação poderia ser calculado usando nossa função `cv()`

```
da$CV <- apply(X = da[, provas], MARGIN = 1, FUN = cv)
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

Finalmente, a estrutura de repetição `if()` também possui uma forma vetorizada através da função `ifelse()`. Essa função funciona da seguinte forma:

```
ifelse(<condição>, <valor se verdadeiro>, <valor se falso>)
```

Dessa forma, a atribuição da situação dos alunos poderia ser feita da seguinte forma:

```
da$res <- ifelse(da$MP >= 70, "aprovado", "reprovado")
da
```

```
# # A tibble: 7 × 11
#   matricula nome    curso prova1 prova2 prova3 faltas media
#       <dbl> <chr>   <chr>  <dbl>  <dbl>  <dbl>  <dbl> <dbl>
# 1       256 João    Mat       80     90     80      4  83.3
# 2       487 Vanessa Mat       75     75     75      4  75
# 3       965 Tiago   Est       95     80     75      0  83.3
# 4       125 Luana   Est       70     85     50      8  68.3
# 5       458 Gisele  Est       45     50      0     16  31.7
# 6       874 Pedro   Mat       55     75     90      0  73.3
# 7       963 André   Est       30      0     30     20  20
# # … with 3 more variables: CV <dbl>, MP <dbl>, res <chr>
```

A família de funções `*apply()`

As funções da chamada família `*apply()` são as implementações básicas de operações vetorizadas no R. Sempre que possível é desejável utilizar estas funções no lugar das estruturas de repetição. Em qualquer situação, a performance destas funções (em tempo computacional) será sempre superior

A função `apply()`, como já vista acima, é capaz de fazer operações nas linhas (`MARGIN = 1`) e também nas colunas (`MARGIN = 2`).

```
## Médias por LINHA: média das 3 provas para cada aluno
apply(X = da[, provas], MARGIN = 1, FUN = mean)
```

```
# [1] 83.33333 75.00000 83.33333 68.33333 31.66667 73.33333
# [7] 20.00000
```

```
## Médias por COLUNA: média de cada uma das 3 provas para todos os
## alunos
apply(X = da[, provas], MARGIN = 2, FUN = mean)
```

```
#   prova1   prova2   prova3
# 64.28571 65.00000 57.14286
```

As funções `sapply()` e `lapply()` são semelhantes à `apply()`, mas operam somente nas colunas.

```
## sapply simpilifica o resultado para um vetor
sapply(da[, provas],  mean)
```

```
#   prova1   prova2   prova3
# 64.28571 65.00000 57.14286
```

```
## lapply retorna o resultado em formato de lista
lapply(da[, provas],  mean)
```

```
# $prova1
# [1] 64.28571
#
# $prova2
# [1] 65
#
# $prova3
# [1] 57.14286
```

A função `tapply()` é similar às anteriores (opera somente nas colunas), mas permite separar o resultado por alguma outr variável (`INDEX`).

```
## Média da prova 1 por situação
tapply(da$prova1,  da$res, mean)
```

```
#  aprovado reprovado
#  76.25000  48.33333
```

```
## Média da prova 2 por situação
tapply(da$prova2,  da$res, mean)
```

```
#  aprovado reprovado
#        80        45
```

```
## Média da prova 3 por situação
tapply(da$prova3,  da$res, mean)
```

```
#  aprovado reprovado
#  80.00000  26.66667
```

No entanto, a função `tapply()` aceita somente uma variável por vez. Se quisermos, por exemplo, obter a média por situação das 3 provas de uma só vez, podemos usar a função `aggregate()`.

```
## Mesmo resultado da tapply, mas agora em formato de data frame
aggregate(prova1 ~ res, data = da, FUN = mean)
```

```
#         res   prova1
# 1  aprovado 76.25000
# 2 reprovado 48.33333
```

```
aggregate(prova2 ~ res, data = da, FUN = mean)
```

```
#         res prova2
# 1  aprovado     80
# 2 reprovado     45
```

```
aggregate(prova3 ~ res, data = da, FUN = mean)
```

```
#         res   prova3
# 1  aprovado 80.00000
# 2 reprovado 26.66667
```

```
## Mas aqui podemos passar as 3 colunas de uma vez
aggregate(cbind(prova1, prova2, prova3) ~ res,
          data = da, FUN = mean)
```

```
#         res   prova1 prova2   prova3
# 1  aprovado 76.25000     80 80.00000
# 2 reprovado 48.33333     45 26.66667
```

Exercícios