#bloco 1

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
3. names(df) <- letters[1:6]
4. df
5. # do not repeat yourself (DRY)
6. df$a[df$a == -99] <- NA
7. df$b[df$b == -99] <- NA
8. df$c[df$c == -98] <- NA
9. df$d[df$d == -99] <- NA
10. df$e[df$e == -99] <- NA
11. df$f[df$g == -99] <- NA
```

##

explicações bloco 1

Código repetido reproduz bugs antigos, gera novos bugs, gera maior trabalho na manutenção, torna a legibilidade ruim, causa relaxamento nas atividades de testes.

Bugs antigos podem ser copiados; novos bugs podem ser gerados em função de pequenas alterações no conteúdo copiado; quando manutenções forem necessárias vários blocos de códigos idênticos podem precisar de alterações; a legibilidade do código fica comprometida e facilmente partes do código podem ser lidas sem devida atenção em função, de aparentemente, ser código já lido e considerado; tende-se a não testar todos os blocos duplicados em função do falso pensamento que são códigos idênticos.

#bloco 2

Encapsular código em uma função pode ser uma solução bem mais apropriada do que duplicar código. O Princípio da Responsabilidade única pode ajudar muito nas questões de definir quais funções devem ser escritas para evitar a replicação de código. Claro, geralmente, repetir código e momentaneamente mais rápido e fácil.

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
3. names(df) <- letters[1:6]
4. df
5. fix_missing <- function(x) {
6. x[x == -99] <- NA
7. x
8. }
9. df$a <- fix_missing(df$a)
10. df$b <- fix_missing(df$b)
11. df$c <- fix_missing(df$c)
12. df$d <- fix_missing(df$d)
```

```
13. df$e <- fix_missing(df$e)
14. df$f <- fix_missing(df$e)
```

Está solução, sem dúvida, reduz a possibilidade de digitar, por engano, um valor diferente de -99, mas ainda mantém a possibilidade de passar como parâmetro para a função uma coluna errada.

Construirmos um “laço for” forneceria uma solução melhor.

 # bloco 3

#bloco 3

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
3. names(df) <- letters[1:6]
4. fix_missing <- function(x) {
5. x[x == -99] <- NA
6. x
7. }
8. for (i in c(1:length(df))){
9. df[i] <- fix_missing(df1[i])
10. }
11. df
```

##

Temos aqui algumas vantagens como: código mais compacto, caso o valor de -99 mudar para qualquer outro valor; a alteração necessária incidirá somente sobre um local; funciona para qualquer número de colunas; a possibilidade de passar parâmetros inválidos para a função diminui muito; não existe a possibilidade de tratar uma coluna de forma diferente em relação a outras colunas.

Outra forma de fazer isso seria iterando sobre o nome das colunas:

#bloco 4

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
3. names(df) <- letters[1:6]
4. fix_missing <- function(x) {
5. x[x == -99] <- NA
6. x
```

```
7. }
8. for (coluna in colnames(df)){
9. df[,coluna] <- fix_missing(df[,coluna])
10. }
11. df
```

##

Porém, ainda poderíamos melhorar a solução. Poderíamos abstrair a variável i ou o nome da coluna, uma vez que está implícito que devemos iterar sobre todo o conjunto de colunas do data frame df.

Na linguagem R, essa funcionalidade é fornecida pela função lapply, conforme mostrado a seguir.

#bloco 5

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
3. names(df) <- letters[1:6]
4. fix_missing <- function(x) {
5. x[x == -99] <- NA
6. x
7. }
8. df[] <- lapply(df, fix_missing)
9. df
```

A função lapply toma cada coluna do data frame df como parâmetro para a função fix_missing, retornando a coluna devidamente transformada.

#####################################

A notação se tornou mais compacta e mais fácil de manter. Por exemplo, supondo que por algum motivo queiramos alterar somente as colunas 2 e 3. Poderíamos fazer da seguinte forma:

#bloco 6

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
3. names(df) <- letters[1:6]
4. df
5. fix_missing <- function(x) {
6. x[x == -99] <- NA
7. x
8. }
9. colunasAlterar <- c(2:3)
10. df[, colunasAlterar] <- lapply(df[,colunasAlterar], fix_missing)
```

11. df

##

A função lapply é um exemplo de função que recebe como parâmetro outra função, isso é comum em outras linguagens de programação, como a linguagem C por exemplo, que embora seja compilada, pode receber o endereço de uma função como parâmetro de outra função.

Vamos ver agora, exemplos de funções que retornam outras funções como parâmetros. Suponhamos que o valor a ser alterado para NA não seja somente -99, mas que também possa assumir os valores -999 e -9999. Lógico, isso poderia ser resolvido simplesmente acrescentado um outro parâmetro à função, mas, a Programação Funcional nos fornece outra maneira de fazer isso. Podemos usar o conceito de Closure. Veremos, primeiro, alguns exemplos de uso.

 # bloco 7

```
1. 1. rm(list=ls())
2. 2. df <- data.frame(replicate(6, sample(c(1:12, -99,-999,-9999), 6, rep =
   TRUE))) 3. names(df) <- letters[1:6]
3. 4.
4. 5. missing_fixer <- function(na_value) {
5. 6. return (function(x) {
6. 7. x[x == na_value] <- NA
7. 8. x
8. 9. })
9. 10. }
10. 11.
11. 12. fix_missing_99 <- missing_fixer(-99)
12. 13. fix_missing_999 <- missing_fixer(-999)
13. 14. fix_missing_9999 <- missing_fixer(-9999)
14. 15.
15. 16. df[] <- lapply(df,fix_missing_99)
16. 17. df[] <- lapply(df,fix_missing_999)
17. 18. df[] <- lapply(df,fix_missing_9999)
```

##

Conforme pode ser visto, a função missing_fixer nos retorna uma função. Nas linhas 12,13, e 14 a função missing_fixer é chamada, retornando uma função que é armazenada em uma variável da classe função.

A solução não é elegante, pois estamos contrariando o princípio Não se Repita ou Do not repeat yourself. Acertaremos isso mais à frente. O Conceito de Closure não está ligado somente a funções que retornam uma função como resultado do processamento, isso também será visto mais mais à frente.

#bloco 8

```
1. rm(list=ls())
2. df <- data.frame(replicate(6, sample(c(1:12, -99,-999,-9999), 6, rep =
     TRUE))) names(df) <- letters[1:6]
3. xx <- df
4. missing_fixer <- function(na_value) {
5. return (function(x) {
6. x[x == na_value] <- NA
7. x
8. })
9. }
10. for (f in c(missing_fixer(-99),missing_fixer(-999),missing_fixer(-9999))){
11. df[] <- lapply(df,f)
12. }
```

##

Na Programação Funcional, funções são objetos (ainda não no sentido da orientação a objetos) que podem ser usados como parâmetros (fizemos isso quando estudamos funções compostas); podem também ser usadas como retorno de funções (fizemos isso quando iniciamos o estudo dos conceito de Closure), podem ser armazenadas em estruturas de dados como vetores (fizemos isso no bloco 8), elas podem ser anônimas, não necessariamente ter um nome. Cada ambiente de Programação Funcional implementa essas características de forma particulares.

Exercício 1.

1) Conforme vídeo: Programação funcional 3, crie uma data frame com 6 linhas e 7 colunas, o valor de cada célula deve estar compreendido entre 0 e 30. Permita que números sejam usados mais de uma vez. Cada uma das colunas deve ter um nome começando com a letra “a” e terminando com a sétima letra do alfabeto. Para nomear as colunas use a função:
names(df) <- letters[1:7]

2) Construa uma função que calcule a média aritmética de uma coluna do data frame. Para isso use as funções disponibilizadas pela linguagem: sum e length. Sum para somar todos os valores de uma coluna e length para contar quantos elementos há em cada coluna. A média para a coluna “a” será calculada da seguinte forma: sum(df$a/lenght(df$a).

3) Faço o mesmo o cálculo usando em vez das funções sum e lenght a função mean disponibilizada pela linguagem R.
A média para a coluna df$a será calculada da seguinte forma mean(df$a)

O desvio padrão é uma medida que informa quanto os valores de um conjunto de dados qualquer estão longe da média. Na linguagem R existe uma função, sd, que calcula o desvio padrão. Por exemplo, para calcular o desvio padrão da coluna df$a faremos da seguinte forma: sd(df$a).

4) Construa, conforme bloco 4 do material entregue em aula, um laço “for” que calcule o desvio padrão de cada uma das colunas do data frame.
Template do laço for:

```
desvioPadrao <- c()
i <- 1
for (coluna in colnames(df)){
  desvioPadrao[i] <- # chamada da função
  i <- i + 1
}
print(desvioPadrao)
```

Através da funcão class(desvioPadrao) perceba que a variável desvioPadrão é do tipo númeric.

5) Refaça o cálculo do desvio padrão para cada coluna do data frame sem utilizar o laço “for” e o contador “i”. Use a função lapply cujo exemplo foi mostrado no bloco 5. Armazene o valor do desvio padrão de cada coluna na variável chamada desvioPadrao.

Através da função class(desvioPadrao) perceba que a variável desvioPadrão é do tipo list.

6) Conforme bloco 6, calcule o desvio padrão somente das colunas dois e três do data frame df.

7) Repita o exercício anterior, porém agora construa uma função desvio padrão meu_sd e altere o comando lapply para não suar sd e sim meu_sd.

Explicações sobre a função desvio padrão.

Conforme dito, o desvio padrão calcula a média das distâncias que os dados se encontram da média. Seja x um vetor de números.
Calculamos a média aritmética de x. mediaDeX <- sum(x)/length(x)
Calculamos o quadrado da distância de cada número até a média (x - mediaDeX)^2
Dividimos (x - mediaDeX)^2 pelo número de elementos de x menos 1:
((x - mediaDeX)^2)/(length(x)-1)
Extraímos a raiz quadrada da soma (sum((x - mediaDeX)^2)/(length(x)-1))^0.5

8) Substitua os números de cada coluna que estão abaixo da média pelo valor da média. Use o código do bloco 5 como exemplo.

No bloco 8 do material apresentado em sala de aula há um exemplo de como funções anônimas, sem nome, podem ser armazenadas em uma estrutura de dados. Depois, cada uma das funções pode ser aplicada a um conjunto de dados.

```
rm(list=ls())
df <- data.frame(replicate(6, sample(c(1:12, -99,-999,-9999), 6, rep = TRUE)))
names(df) <- letters[1:6]

missing_fixer <- function(na_value) {
 return (function(x) {
  x[x == na_value] <- NA
  x
})
}
for (f in c(missing_fixer(-99),missing_fixer(-999),missing_fixer(-9999))){
 df[] <- lapply(df,f)
}
```

Vocês devem entender o que esse bloco de código faz e criar um bloco semelhante, isto é, que armazene funções em um vetor e depois aplique essas funções em um conjunto de dados, conforme mostrado acima.