#bloco 1

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

```
rm(list=ls())
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df

fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}
df$a <- fix_missing(df$a)
df$b <- fix_missing(df$b)
df$c <- fix_missing(df$c)
```

```
df$d <- fix_missing(df$d)
df$e <- fix_missing(df$e)
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

rm(list=ls())
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]

fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

for (i in c(1:length(df))){
    df[i] <- fix_missing(df1[i])
}
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

```
rm(list=ls())
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
```

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

for (coluna in colnames(df)){
  df[,coluna] <- fix_missing(df[,coluna])
}
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

```
rm(list=ls())
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]

fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

df[] <- lapply(df, fix_missing)
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

```
rm(list=ls())
df <- data.frame(replicate(6, sample(c(1:10, -99), 6, rep = TRUE)))
names(df) <- letters[1:6]
df
```

```
fix_missing <- function(x) {
  x[x == -99] <- NA
  x
}

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

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

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)
}
##############################################################################
```

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.

Exercícios.

1 - Conforme “#bloco 1”, crie um data frame com 7 linhas e 7 colunas chamado df, escolha os números que farão parte dos dados do data frame usando a função sample. A função sample

deve escolher os números que farão parte do data frame entre os números de 1 a 100 mais os números -9, -99,-999 e -9999. Nomeie as colunas do data frame com letras do alfabeto.

2 - Nas colunas 3 e 4 do data frame df, caso haja valores negativos, os mesmos devem ser substituídos por NA (not available)

3 - Crie uma função que use o conceito de Closure conforme visto até o presente momento.