Programação em R

Julho, 2016

Executando código de maneira condicional: `if()`, `if() else`, `ifelse()`

Há ocasiões em queremos ou precisamos executar parte do código **apenas se alguma condição for atendida**. Veremos três formas de fazer isso: `if()`, `if() else` e `ifelse()`.

O `if()`: estrutura

A estrutura básica:

```
if (condicao) {

  # comandos que
  # serao rodados
  # caso condicao = TRUE

}
```

- O início do código se dá com o comando `if` seguido de parênteses e chaves;
- Dentro do parênteses temos uma condição lógica, que deverá ter como resultado ou `TRUE` ou `FALSE`;
- Dentro das chaves temos o bloco de código que será executado se – e somente se – a condição do parênteses for `TRUE`.

O `if()`: exemplo

```
# vetores de condição lógica
cria_x <- TRUE
cria_y <- FALSE
# só executa se cria_x = TRUE
if (cria_x) {
  x <- 1
}
# só executa se cria_y = TRUE
if (cria_y) {
  y <- 1
}

# note que x foi criado mas y não
exists("x"); exists("y")
## [1] TRUE
## [1] FALSE
```

O `if()` com o `else`

Outra forma de executar códigos de maneira condicional é acrescentar ao `if()` o opcional `else`. Estrutura básica:

```
if (condicao) {
  # comandos que
  # serao rodados
  # caso condicao = TRUE
} else {
  # comandos que
  # serao rodados
  # caso condicao = FALSE
}
```

O `if()` com o `else`

Em detalhes:

- O início do código se dá com o comando `if` seguido de parênteses e chaves;
- Dentro do parênteses temos uma condição lógica, que deverá ter como resultado ou `TRUE` ou `FALSE`;
- Dentro das chaves do `if()` temos um bloco de código que será executado se – e somente se – a condição do parênteses for `TRUE`.
- Logo em seguida temos o `else` seguido de chaves;
- Dentro das chaves do `else` temos um bloco de código que será executado se – e somente se – a condição do parênteses for `FALSE`.

O if() com o else: exemplo

Como no caso anterior, vejamos primeiramente um exemplo bastante simples.

```
numero <- 1

if (numero == 1) {
  cat("o numero é igual a 1")
} else {
  cat("o numero não é igual a 1")
}
## o numero é igual a 1
```

O `if()` com o `else`: encadeando

É possível encadear diversos `if() else` em sequência:

```
numero <- 10

if (numero == 1) {
  cat("o numero é igual a 1")
} else if (numero == 2) {
  cat("o numero é igual a 2")
} else {
  cat("o numero não é igual nem a 1 nem a 2")
}
## o numero não é igual nem a 1 nem a 2
```

Exemplo: par ou ímpar?

Vamos criar uma função que nos diga se um número é par ou ímpar. Nela vamos utilizar tanto o `if()` sozinho quanto o `if() else`. Vale relembrar que um número (inteiro) é par se for divisível por 2 e que podemos verificar isso se o resto da divisão (operador %% no R) deste número por 2 for igual a zero.

Exemplo: par ou ímpar?

```
par_ou_impar <- function(x){
  # verifica se o número é um decimal comparando o tamanho da diferença de x e fl
  # se for decimal retorna NA (pois par e ímpar não fazem sentido para decimais)
 if (abs(x - round(x)) > 1e-7) {
    return(NA)
  }

  # se o número for divisível por 2 (resto da divisão zero) retorna "par"
  # caso contrário, retorna "ímpar"
  if (x %% 2 == 0) {
    return("par")
  } else {
    return("impar")
  }
}
```

Exemplo: par ou ímpar?

Testando:

```
par_ou_impar(4)
## [1] "par"

par_ou_impar(5)
## [1] "impar"

par_ou_impar(2.1)
## [1] NA
```

Parece que está funcionando bem, mas...

Lembre-se: `if()` não é vetorizado.

Há um pequeno problema:

```
x <- 1:5
par_ou_impar(x)
## Warning in if (abs(x - round(x)) > 1e-07) {: a condição tem comprimento > 1 e
## somente o primeiro elemento será usado
## Warning in if (x%%2 == 0) {: a condição tem comprimento > 1 e somente o primei
## elemento será usado
## [1] "impar"
```

Provavelmente não era isso o que esperávamos. O que está ocorrendo aqui?

Os comandos `if()` e `if() else` não são vetorizados.

O `if()` aceita apenas um único valor, seja `TRUE` ou `FALSE`. Se você passar mais de um valor, ele ignora os demais e usa apenas o primeiro.

Lembre-se: `if()` não é vetorizado.

Revendo a questão em um exemplo mais simples, note que o `if()` irá usar apenas o primeiro valor do vetor `c(F, T)`:

```
if (c(F, T)) {

  print("TRUE")

} else {

  "FALSE"
}
## Warning in if (c(F, T)) {: a condição tem comprimento > 1 e somente o pr
## elemento será usado
## [1] "FALSE"
```

A função `ifelse()`

Uma alternativa para casos como esses é utilizar a função `ifelse()`.
Estrutura básica:

```
ifelse(vetor_de_condicoes, valor_se_TRUE, valor_se_FALSE)
```

- o primeiro argumento é um vetor (ou uma expressão que retorna um vetor) com vários `TRUE` e `FALSE`;
- o segundo argumento é o valor que será retornado quando o elemento do `vetor_de_condicoes` for `TRUE`;
- o terceiro argumento é o valor que será retornado quando o elemento do `vetor_de_condicoes` for `FALSE`.

A função `ifelse()`: exemplo

Primeiramente, vejamos um caso trivial:

```
ifelse(c(TRUE, FALSE, FALSE, TRUE), 1, -1)
## [1]  1 -1 -1  1
```

Note que passamos um vetor de condições com `TRUE`, `FALSE`, `FALSE` e `TRUE`.
O valor para o caso `TRUE` é `1` e o valor para o caso `FALSE` é `-1`.
Logo, o resultado é `1`, `-1`, `-1` e `1`.

A função ifelse(): voltando ao par ou ímpar

Vamos criar uma versão com ifelse da nossa função que nos diz se um número é par ou ímpar.

```
par_ou_impar_ifelse <- function(x){

  # se x for decimal, retorna NA, se não for, retorna ele mesmo (x)
  x <- ifelse(abs(x - round(x)) > 1e-7, NA, x)

  # se x for divisivel por 2, retorna 'par', se não for, retorna impar
  ifelse(x %% 2 == 0, "par", "impar")
}
```

A função `ifelse()`: voltando ao par ou ímpar

Perceba que agora a função funciona sem problemas com vetores:

```
par_ou_impar_ifelse(x)
## [1] "impar" "par"   "impar" "par"   "impar"

par_ou_impar_ifelse(c(x, 1.1))
## [1] "impar" "par"   "impar" "par"   "impar" NA
```

Vetorização!

Um tema constante neste curso é fazer com que você pense sempre em explorar a vetorização do R. Este caso não é diferente, poderíamos ter feito a função utilizando apenas comparações vetorizadas!

Vetorização!

```
par_ou_impar_vec <- function(x){
  # transforma decimais em NA
  decimais <- abs(x - round(x)) > 1e-7
  x[decimais] <- NA

  # Cria vetor para aramazenar resultados
  res <- character(length(x))

  # verifica quem é divisível por dois
  ind <- (x %% 2) == 0

  # quem for divisível por dois é par, quem não for é ímpar
  res[ind] <- "par"
  res[!ind] <- "impar"

  return(res)
}
```

Vetorização!

Na prática, o que a função `ifelse()` faz é mais ou menos isso o que fizemos – comparações e substituições de forma vetorizada. Note que, neste caso, nossa implementação ficou inclusive um pouco mais rápida do que a solução anterior com `ifelse()`:

```
library(microbenchmark)
microbenchmark(par_ou_impar_vec(1:1e3), par_ou_impar_ifelse(1:1e3))
## Unit: microseconds
##                          expr min  lq mean median  uq  max neval cld
##     par_ou_impar_vec(1:1000)  55  56   82     58  85  995   100  a
##  par_ou_impar_ifelse(1:1000) 320 323  458    338 486 2036   100   b
```

Exercícios

Sua vez.

- Crie, usando `if() else` uma função que verifica se `x` é maior do que 1. Se for, retorna o valor `x^2`. Se não for, verifica se `x` é menor do que -1. Se for, retorna `-x^2`. Se não for nenhum dos casos, retorna o próprio `x`. Sua função é vetorizada?
- Cria a mesma função usando `ifelse()`.
- Crie a função sem usar nem `if() else` nem `ifelse()`.

Soluções

```r
# uma forma diferente..
funcao_if <- function(x){
  if (x > 1) return(x^2)
  if (x < -1) return(-x^2)
  return(x)
}

funcao_ifelse <- function(x) ifelse(x > 1, x^2, ifelse(x < -1, -x^2, x))

funcao <- function(x){
  x[x > 1] <- x[x > 1]^2
  x[x < -1] <- -x[x < -1]^2
  return(x)
}
```

Executando código diversas vezes (loops): `for()` e `while()`

Loops com `for()`

Um loop utilizando `for()` no R tem a seguinte estrutura básica:

```
for(i in conjunto_de_valores){
  # comandos que
  # serão repetidos
}
```

- O início do loop se dá com o comando `for` seguido de parênteses e chaves;
- Dentro do parênteses temos um indicador que será usado durante o loop (no caso escolhemos o nome `i`) e um conjunto de valores que será iterado (`conjunto_de_valores`).
- Dentro das chaves temos o bloco de código que será executado durante o loop.

Em outras palavras, no comando acima estamos dizendo que para cada elemento `i` contido no `conjunto_de_valores` iremos executar os comandos que estão dentro das chaves.

Loops com for(): exemplo

Vamos imprimir na tela os números de 1 a 5.

```
for (i in 1:5) {
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Loops com for(): exemplo

Agora, vamos imprimir na tela as 5 primeiras letras do alfabeto (o R já vem com um vetor com as letras do alfabeto: `letters`).

```
for (i in 1:5) {
  print(letters[i])
}
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Loops com `for()`: exemplo

No mesmo exemplo, ao invés correr o loop no índice de inteiros `1:5`, vamos iterar diretamente sobre os primeiros 5 elementos do vetor `letters`:

```
for (letra in letters[1:5]) {
  print(letra)
}
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Loops com for(): seq_along() e length()

Uma função bastante útil ao fazer loops é a função seg_along(). Ela cria um vetor de inteiros com índices para acompanhar o objeto.

```
# criando um vetor de exemplo
set.seed(119)
x <- rnorm(10)

# inteiros de 1 a 10
seq_along(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Loops com `for()`: `seq_along()` e `length()`

Também é possível criar um vetor de inteiros do tamanho do objeto fazendo uma sequência de 1 até `length(x)`:

```
1:length(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Entretanto, a vantagem de `seq_along()` é que quando o vetor é vazio, ela retorna um vetor vazio e, deste modo, o loop não é executado (o que é o comportamento correto). Já a sequência `1:length(x)` retorna a sequência `1:0`, isto é, uma sequência decrescente de 1 até 0, e loop é executado nestes valores. Vejamos.

Loops com for(): seq_along() e length()

```
# cria vetor vazio
x <- numeric(0)

# 1:length(x)
# note que o loop é executado (o que é errado)
for (i in 1:length(x)) print(i)
## [1] 1
## [1] 0

# seq_along
# note que o loop não é executado (o que é correto)
for (i in seq_along(x)) print(i)
```

Vetorização, funções nativas e loops

- Como vimos, o R é vetorizado. Muitas vezes, quando você pensar que precisa usar um loop, ao pensar melhor, descobrirá que não precisa. Em geral é possível resolver o problema de maneira vetorizada e usando funções nativas do R.
- Para quem está aprendendo a programar diretamente com o R, isso é algo que virá naturalmente. Todavia, para quem já sabia programar em outras linguagens de programação – como C – pode ser difícil se acostumar a pensar desta maneira.

Vetorização, funções nativas e loops

Vejamos um exemplo trivial. Suponha que você queira dividir os valores de um vetor `x` por 10. Se o R não fosse vetorizado, você teria que fazer algo como:

```
# criando vetor de exemplo
x <- 10:20

# divide cada elemento por 10
for (i in seq_along(x))
  x[i] <- x[i]/10

# resultado
x
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Vetorização, funções nativas e loops

Mas o R **é** vetorizado e, portanto, este é o tipo de loop **que não faz sentido** na linguagem. É muito mais rápido e fácil de enteder escrever simplesmente `x/10`, como já tínhamos aprendido nas primeiras aulas!

```
# recriando vetor de exemplo
x <- 10:20

# divide cada elemento por 10
x <- x/10
x
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Vetorização, funções nativas e loops

Vejamos um caso um pouco mais complicado. Suponha que você queira, gerar um passeio aleatório com um algoritmo simples: a cada período você pode andar para frente (+1) ou para trás (-1) com probabilidades iguais.

```
set.seed(1)

# número de passos
n <- 1000
# vetor para armazenar o passeio aleatório
passeio <- numeric(n)
# primeiro passo
passeio[1] <- sample(c(-1, 1), 1)
# demais passos
for (i in 2:n) {
  # passo i é o onde você estava (passeio[i-1])   mais o passo seguinte
  passeio[i] <- passeio[i - 1] + sample(c(-1, 1), 1)
}
```

Vetorização, funções nativas e loops

É possível fazer tudo isso com apenas uma linha de maneira "vetorizada" e bem mais eficiente: crie todos os `n` passos de uma vez e faça a soma acumulada.

```
set.seed(1)
passeio2 <- cumsum(sample(c(-1, 1), n, TRUE))

# verifica se são iguais
all.equal(passeio, passeio2)
## [1] TRUE
```

Vetorização, funções nativas e loops

Então, você deve estar se perguntando: "não é para usar loops nunca"?

Não é isso. Em algumas situações loops são inevitáveis e podem inclusive ser mais fáceis de ler e de entender. O ponto aqui é apenas lembrá-lo de explorar a vetorização do R.

DICA: pré-alocar espaço antes do loop

Um erro bastante comum de quem está começando a programar em R é "crescer" objetos durante o loop. Isto tem um impacto substancial na performance do seu programa!

- Sempre que possível, crie um objeto, antes de iniciar o loop, para armazenar os resultados de cada iteração.

Vamos calcular os `n` primeiros números da sequência de Fibonacci:
$F_1 = 0, F_2 = 1, F_3 = 1, F_4 = 2, F_5 = 3, F_6 = 5, F_7 = 8, F_8 = 13, F_9 = 21$... Note que a sequência de Fibonacci pode ser definida da seguinte forma:

- os primeiros dois números são 0 e 1, isto é, $F_1 = 0, F_2 = 1$;
- A partir daí, os números subsequentes são a soma dos dois números anteriores, isto é, $F_i = F_{i-1} + F_{i-2}$ para todo $i > 2$.

Vejamos uma forma de implementar isto no R usando `for()` e criando um vetor para armazenar os resultados.

DICA: pré-alocar espaço antes do loop

```
n <- 9
# crie um vetor de tamanho n
# para armazenar os n resultados
fib <- numeric(n)
# comece definindo as condições iniciais
# F1 = 0 e F2 = 1
fib[1] <- 0
fib[2] <- 1
# Agora para todo i > 2
# calculamos Fi = F(i-1) + F(i - 2)
for (i in 3:n) {
  fib[i] <- fib[i - 1] + fib[i - 2]
}
# conferindo resultado
fib
## [1]  0  1  1  2  3  5  8 13 21
```

Loops com `while()`

Estrutura básica:

```
while (condicao) {
  # codigo a ser executado
  # até que condição seja TRUE
  # em geral a condição será atualizada
  # dentro do código
}
```

- O início do loop se dá com o comando `while` seguido de parênteses e chaves;
- Dentro do parênteses temos uma condição lógica que será testada;
- Enquanto a condição lógica for verdadeira, bloco de código que está entre chaves será executado repetidamente.

Loops com `while()`: exemplo

Lembra que com o for contamos de 1 até 5? Como fazer a mesma coisa com o while?

```
# i inicial
i <- 1
# condição:
# enquanto i for menor ou igual a 5
while (i <= 5) {
  print(i)
  # atualiza i (cuidado com loop infinito!)
  i <- i + 1
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

loop infinito

Se você entrar em um loop infinito, aperte `esc` para sair. Vamos testar: o código abaixo irá iniciar um loop infinito incrementando a variável `i`. Deixe o código rodando por um tempo e, depois, aperte `esc` para interromper.

```
i <- 1
while (i>=1) {
  print(i)
  i <- i  + 1
}
```

- Por que o código acima está em loop infinito?

Qual a diferença? `while()` vs `for()`

Usar o `while()` para contar de 1 até 5 é bem mais complicado do que com o `for()`, então qual a vantagem do `while()`? O `while()` é fundamental quando precisamos rodar uma função repetidas vezes **mas não sabemos quantas** vezes!

```
set.seed(1)
x <- rnorm(1)
i <- 1

while (x < 2) {
  i <- i + 1
  x <- rnorm(1)
}

# quantas vezes rodou?
i
## [1] 61
```

Mais controle sobre loops: `break()`

A função `break()` interrompe a execução de um loop no momento em que é chamada.

- exemplo: quero que o loop rode de 1 até 10, mas se por acaso a variável `u` for maior do que `0.8`, o loop é interrompido.

```
set.seed(25)
for (i in 1:10) {
  u <- runif(1)
  if (u > 0.8) break()
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
```

Mais controle sobre loops: next() e break()

A função next() faz com que o loop passe para a próxima iteração no momento em que é chamada.

- exemplo: quero que o loop rode de 1 até 10, mas se por acaso a variável u for maior do que 0.5, eu pulo aquela parte do loop.

```
set.seed(25)
for (i in 1:10) {
  u <- runif(1)
  if (u > 0.5) next()
  print(i)
}
## [1] 1
## [1] 3
## [1] 5
## [1] 8
## [1] 9
## [1] 10
```

Outra forma de fazer while(): repeat + break

A função repeat repete o código entre chaves indefinidamente. Deve ser utilizada **sempre** em conjunto com a função break().

```
# contando de 1 até 5
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) break()
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Exercícios

Sua vez.
As funções que você irá implementar aqui serão até mais de 100 vezes mais lentas do que as funções nativas do R. Estes exercícios são para você treinar a construção de loops, um pouco de lógica de programação, e entender o que as funções do R estão fazendo – de maneira geral – por debaixo dos panos.

Exercícios

1. Crie uma função que encontre o máximo de um vetor (faça uma vez usando `for()` e outra usando `while()`). Compare os resultados de sua implementação com a função `max()` do R. **Dicas**: você terá que percorrer o vetor e comparar elementos.
2. Crie uma função que calcule o fatorial de `n` (faça uma vez usando `for()` e outra usando `while()`). Compare os resultados de sua implementação com a função `factorial()` do R. **Dica**: para quem não lembra, o fatorial de um número `n` é a multiplicação de todos os número de 1 até `n`. Por exemplo, o fatorial de 6 é $6 * 5 * 4 * 3 * 2 * 1$.

Soluções

```
# cria vetor para comparar resultados
set.seed(123)
x <- rnorm(100)

# 1) loop para encontrar máximo (com for)
max_loop <- function(x){
  max <- x[1]
  for (i in 2:length(x)) {
    if (x[i] > max) {
      max <- x[i]
    }
  }
  return(max)
}
all.equal(max(x), max_loop(x))
## [1] TRUE
```

Respostas

```
# 2) loop para fatorial (com for)
fatorial <- function(n){
  if (n == 0) return(1)
  fat <- 1
  for (i in 1:n) {
    fat <- fat*i
  }
  return(fat)
}

all.equal(factorial(10), fatorial(10))
## [1] TRUE
```

A forma do R de fazer loops: voltando à família apply!

Já vimos várias funções da família apply, como `apply()`, `lapply()` e `sapply()`, quando estudamos matrizes, data.frames e listas. Agora vamos ver novamente essas funções, mas sob outra ótica: as funções da família apply nada mais são do que funções que facilitam sua vida, fazendo loops para você!

Por que a família apply?

Vamos calcular a média de cada uma das colunas do `data.frame` `mtcars` usando loops. Para isso precisamos:

- saber quantas colunas existem no `data.frame`;
- criar um vetor para armazenar os resultados;
- nomear o vetor de resultados com os nomes das colunas; e
- fazer um loop para cada coluna.

Por que a família apply?

```
# (i) quantas colunas no data.frame
n <- ncol(mtcars)

# (ii) vetor para armazenar resultados
medias <- numeric(n)

# (iii) nomeando vetor com nomes das colunas
names(medias) <- colnames(mtcars)

# (iv) loop para cada coluna
for (i in seq_along(mtcars)) medias[i] <- mean(mtcars[,i])

# resultado final
medias
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81
```

Por que a família apply?

Gastamos várias linhas para fazer essa simples operação. Como já vimos, é bastante fácil fazer isso no R com apenas uma linha:

```
sapply(mtcars, mean)
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81
```

- Mas, imagine que não existisse a função `sapply()` no R. Se quiséssemos aplicar outra função para cada coluna, teríamos que copiar e colar todo o código novamente, certo?
- Sim, você poderia fazer isso, mas não seria uma boa prática. Neste caso, como já vimos, o ideal seria criar uma função!

Por que a família apply?

Façamos, portanto, uma função que nos permita aplicar uma fução arbitrária nas colunas de um `data.frame`.

```
meu_sapply <- function(x, funcao){

  n <- length(x)

  resultado <- numeric(n)

  names(resultado) <- names(x)

  for(i in seq_along(x)){
    resultado[i] <- funcao(x[[i]])
  }

  return(resultado)
}
```

Por que a família apply?

Perceba que ficou bastante simples percorrer todas as colunas de um `data.frame` para aplicar a função que você quiser:

```
meu_sapply(mtcars, mean)
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81

meu_sapply(mtcars, sd)
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##   6.03   1.79 123.94  68.56   0.53   0.98   1.79   0.50   0.50   0.74   1.62

meu_sapply(mtcars, max)
##   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##  33.9   8.0 472.0 335.0   4.9   5.4  22.9   1.0   1.0   5.0   8.0
```

Por que a família apply?

É isso o que as funções da família `apply` são: são funções que fazem loops para você. Elas automaticamente cuidam de toda a parte chata do loop como, por exemplo, criar um objeto de tamanho correto para pré-alocar os resultados. Além disso, em grande parte das vezes essas funções serão mais eficientes do que se você mesmo fizer a implementação.

Principais funções apply

- **Aplicar função nas dimensões:** como vimos, a função `apply()` aplica funções nas linhas, colunas ou outras dimensões de uma matriz, data.frame ou array.
- **Aplicar função nos elementos:** como vimos, para aplicar uma função a cada elemento de um objeto, podemos usar `lapply()`, `sapply()` ou `vapply()`. A diferença entre elas é o formato do resultado.
  1. A função **l**apply retorna uma **l**ista;
  2. A função **s**apply tenta **s**implificar o resultado para um objeto mais simples (como um vetor ou matriz); e,
  3. A função **v**apply espera como resultado um formato de **v**alor específico (caso contrário, retorna erro).
- **Aplicar funções em múltiplos elementos**: a função `mapply()` pode ser consideradas uma versão multivariada do `sapply()`. O `mapply()` aplica uma função em todos elementos de **m**últiplos objetos ao mesmo tempo.
- **Repetir código em simulações de Mote Carlo**: para isso temos a função `replicate()`, que replica uma expressão diversas vezes.

Aplicando funções em dimensões: `apply()`

```
set.seed(1)
x <- matrix(rnorm(9), ncol = 3)

apply(x, 1, mean)
## [1]  0.49  0.42 -0.36

apply(x, 2, mean)
## [1] -0.43  0.37  0.60
```

Aplicando funções em elementos: `lapply()` - retorna lista

```
lista_de_matrizes <- list(x = x, tx = t(x))

# invertendo todas ao mesmo tempo
lapply(lista_de_matrizes, solve)
## $x
##        [,1]   [,2]  [,3]
## [1,] -0.500  0.829 -0.64
## [2,]  0.454 -0.029 -0.35
## [3,] -0.078  1.161  0.31
##
## $tx
##       [,1]   [,2]   [,3]
## [1,] -0.50  0.454 -0.078
## [2,]  0.83 -0.029  1.161
## [3,] -0.64 -0.347  0.314
```

Aplicando funções em elementos: sapply() - simplifica resultado

Calculando determinantes de todas as matrizes:

```
lapply(lista_de_matrizes, det)
## $x
## [1] -1.6
##
## $tx
## [1] -1.6
```

Com sapply() resultado já vem como vetor:

```
sapply(lista_de_matrizes, det)
##    x   tx
## -1.6 -1.6
```

Aplicando funções em elementos: vapply() - checa resultado

```
ok  <- list(x = matrix(1:10, ncol = 2),
            y = matrix(11:20, ncol = 2))

nao_ok <- list(x = matrix(1:10, ncol = 2),
               y = matrix(11:20, ncol = 5))

vapply(ok, colMeans, numeric(2))
##      x  y
## [1,] 3 13
## [2,] 8 18

vapply(nao_ok, colMeans, numeric(2))
## Error in vapply(nao_ok, colMeans, numeric(2)): valores devem ser de comprimento
##  mas o resultado de FUN(X[[2]]) tem comprimento 5
```

Quando usar cada um: `lapply()`, `sapply()` vs `vapply()`

- O `sapply()` é para uso interativo. Quando você está explorando uma base de dados, o `sapply()` facilita seu trabalho tentando simplificar o resultado da operação. Entretanto, o `sapply()` não te dá sempre o mesmo resultado (às vezes pode ser um vetor, às vezes uma lista), e isso pode ser perigoso para usar em funções.
- A função `lapply()` é mais previsível que o `sapply()`: ela sempre vai te retornar uma lista. Neste caso, você vai ter o trabalho de simplificar o resultado manualmente, mas não terá a surpresa de vir um resultado em um formato diferente do que você esperava.
- Por fim o `vapply()` é para quando você quer ser bastante restrito no tipo de resultado que você deseja obter para evitar bugs. Por exemplo: o resultado esperado sempre tem que ser um vetor numérico de tamanho 2 – e se vier qualquer valor diferente disso, você quer que a função pare e forneça uma mensagem de erro.

Passando argumentos extras

Se você olhar a definição das funções, verá que o `...` aparece em todas elas. Por quê?

- `apply(X, MARGIN, FUN, ...)`
- `lapply(X, FUN, ...)`
- `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`
- `vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)`

Olhe na ajuda:

- ***... optional arguments to FUN***.

Isto é, como vimos na aula de funções, os `...` servem justamente para passar argumentos arbitrários para função `FUN` que está sendo aplicada!

Simulações de Monte Carlo: `replicate()`

O `replicate()` é uma função de conveniência para repetir a execução de uma expressão diversas vezes no R. Sua estrutura básica é a seguinte:

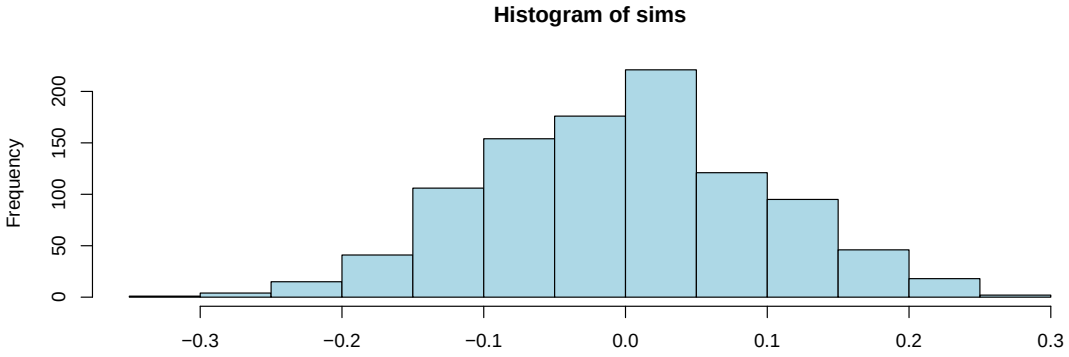
```
replicate(numero_de_repeticoes, expressao)
```

A função `replicate()` é bastante utilizada para simulações de Monte Carlo.

Simulações de Monte Carlo: `replicate()`

Exemplo: distribuição de média amostral.

```
sims <- replicate(1000, mean(rnorm(100)))
hist(sims, col = "lightblue")
```



Aplicando funções em múltiplos argumentos: `mapply()`

A função `mapply()` pode ser vista como um `sapply()` "multivariado". O comando:

```
mapply(funcao, x, y, z)
```

É equivalente a:

```
funcao(x[1], y[1], z[1])
funcao(x[2], y[2], z[2])
funcao(x[3], y[3], z[3])
funcao(x[4], y[4], z[4])
...
funcao(x[n], y[n], z[n])
funcao(x[n], y[n], z[n])
```

Aplicando funções em múltiplos argumentos: `mapply()`

Exemplo: quero a média aparada de `x1` e de `x2`, mas quero que em `x1` `trim = 0.1` e em `x2` `trim = 0.2`. Fazendo com `mapply()`:

```
set.seed(112)
x1 <- rnorm(100)
x2 <- rnorm(100)
mapply(mean, x = list(x1, x2), trim = list(0.1, 0.2))
## [1]  0.13 -0.03
```

O que seria equivalente a:

```
mean(x1, trim = 0.1)
## [1] 0.13

mean(x2, trim = 0.2)
## [1] -0.03
```

Tabela resumo

Função	Descrição
`apply()`	Aplica função nas dimensões (linha, coluna, etc...) do objeto.
`lapply()`	Aplica função em todos elementos do objeto. Retorna uma lista.
`sapply()`	Similar a `lapply` mas tenta simplificar resultado.
`vapply()`	Similar a `sapply` mas checa formato do resultado.
`mapply()`	Versão multivariada do `sapply()`. Aplica função a todos elementos de vários objetos.
`replicate()`	Replica expressão um número pré-estabelecido de vezes.

Tem mais...

Veremos outras opções quando nos aprofundarmos em manipulações de `data.frames`!