UNICAMP

Análise de Dados

Implementação de Funções

Prof. Zanoni Dias

2020

Instituto de Computação – Unicamp

Roteiro

Definindo Funções

Argumentos de uma Função

Escopo

Estruturas de Controle

Funções de Agrupamentos

Definindo Funções

Definindo Funções

- Funções em R podem ser tratadas como outros objetos quaisquer.
- Sendo assim, é possível, por exemplo:
  - Passar funções como argumentos para outras funções.
  - Criar funções dentro de outras funções.
- O retorno de uma função pode ser determinado de duas formas:
  - Implícita: último valor calculado pela função.
  - Explícita: através do comando `return()`.

Definindo Funções

- A diretiva `function()` cria um objeto que representa uma função.
- Sintaxe:

```
f <- function(<argumentos>) {
  # Comandos da funcao
}
```

- A função `is.function()` verifica se um objeto é uma função.

Definindo Funções

- Exemplo:

```
1 mysum <- function(x, y) {
2   x + y
3 }
```

```
1 > is.function(mysum)
2 [1] TRUE
```

```
1 > mysum(2, 5)
2 [1] 7
```

```
1 > mysum(1:3, 8:10)
2 [1] 9 11 13
```

Definindo Funções

- Exemplo:

```
makePower <- function(n) {
  function(x) {
    x^n
  }
}
```

```
> square <- makePower(2)
> cube <- makePower(3)
```

```
> square(7)
[1] 49
```

```
> cube(5)
[1] 125
```

Argumentos de uma Função

Argumentos de uma Função

- Argumentos formais são aqueles listados na definição da função.
- A função `formals()` retorna a lista dos argumentos formais de uma função dada.
- Argumentos tem nomes, que facilitam a passagem de parâmetros.

Definindo Funções

- Exemplo:

```
> formals(matrix)
$data
[1] NA
$nrow
[1] 1
$ncol
[1] 1
$byrow
[1] FALSE
$dimnames
NULL
```

Argumentos de uma Função

- Argumentos podem ter valores padrões, a serem usados na função, caso não sejam especificados outros valores para eles.
- Nem todos os argumentos precisam receber valores na chamada de uma funções, nem mesmos aqueles sem valores padrões.
- É possível verificar se um parâmetro foi informado na chamada da função utilizando a função `missing()`.

Argumentos de uma Função

- Exemplos:

```
1 teste <- function(parametro) {
2   return(missing(parametro))
3 }
```

```
1 > formals(teste)
2 $parametro
```

```
1 > teste()
2 [1] TRUE
```

```
1 > teste("Falta algum parametro?")
2 [1] FALSE
```

Casamento de Valores e Argumentos

- Valores e argumentos podem ser casados posicionalmente ou por nome.
- É possível usar os dois tipos de casamento numa mesma chamada.
- Argumentos com nomes são úteis em funções com muitos argumentos, evitando a memorização da ordem dos argumentos e permitindo que apenas os valores de alguns argumentos sejam fornecidos (deixando os demais com seus valores padrões).
- Valores e argumentos podem ser casados usando apenas parte (inicial) do nome, caso não haja ambiguidade.

Argumentos de uma Função

- Exemplos:

```
subvector <- function(vector, begin = 1,
                      end = length(vector)) {
  return(vector[begin:end])
}
```

```
> subvector(1:10, begin = 5)
[1]  5  6  7  8  9 10
```

```
> subvector(vec = 1:10, end = 5)
[1] 1 2 3 4 5
```

```
> subvector(begin = 3, end = 6, 1:10)
[1] 3 4 5 6
```

Casamento de Valores e Argumentos

- Exemplos (continuação):

```
mydist <- function(p1 = c(0,0), p2 = c(0,0)) {
  sqrt((p1[1] - p2[1])^2 + (p1[2] - p2[2])^2)
}
```

```
> mydist(p1 = c(4,3))
[1] 5
```

```
> mydist(p2 = c(2,5))
[1] 5.385165
```

```
> mydist(p2 = c(2,3), p1 = c(7,15))
[1] 13
```

O Argumento "..."

- O argumento "..." indica um número variável de argumentos que, usualmente, são passados para outras funções.
- O argumento "..." é frequentemente usado para estender uma outra função, sem a necessidade de copiar a lista completa de argumentos.
- O argumento "..." também é usado quando não sabemos a priori o número de argumentos que serão recebidos.
- Argumentos que aparecem após "..." precisam ser nomeados explicitamente e não podem ser casados parcialmente.
- Podemos criar um vetor ou uma lista, para ser manipulados internamente pela função, usando `c(...)` ou `list(...)`, respectivamente.

O Argumento "..."

- Exemplos:

```
mylength <- function(...) {
  length(c(...))
}
```

```
> mylength(3, 2, 4)
[1] 3
```

- Qual será a resposta para a seguinte chamada da função?

```
> mylength(3, 2, 4, 1:5)
```

```
[1] 8
```

O Argumento "..."

- Exemplos (continuação):

```
mylength <- function(...) {
  length(list(...))
}
```

```
> mylength(3, 2, 4)
[1] 3
```

- Qual será a resposta para a seguinte chamada da função?

```
> mylength(3, 2, 4, 1:5)
```

```
[1] 4
```

Escopo

Escopo

- Objetos definidos dentro de uma função, assim como os próprios argumentos da função, são considerados locais, ou seja, não existem fora da função. Desta forma, a passagem de parâmetros para funções é feita por valor (cópia).
- `search()`: lista os ambientes onde R deve buscar objetos.
- O ambiente "`.GlobalEnv`" (definições de usuários) é sempre o primeiro e "`package:base`" é sempre o último.
- Exemplo:

```
> search()
[1] ".GlobalEnv"        "tools:rstudio"
[3] "package:stats"     "package:graphics"
[5] "package:grDevices" "package:utils"
[7] "package:datasets"  "package:methods"
[9] "Autoloads"         "package:base"
```

Escopo

- Exemplo:

```
> length <- rev
```

```
> length(c(4:9))
[1] 9 8 7 6 5 4
```

```
> base::length(c(4:9))
[1] 6
```

```
> rm(length)
```

```
> length(c(4:9))
[1] 6
```

Escopo

- `ls()`: lista os nomes conhecidos em um certo ambiente. Caso não especificado, considera o ambiente atual. Dentro de uma função considerará apenas os argumentos e os objetos locais.
- Exemplo:

```
> length(ls(name = "package:base"))
[1] 1218
> length(ls(name = "package:stats"))
[1] 447
```

- `rm()`: remove os objetos especificados.
  - `rm(list = ls())`: remove todos os objetos do ambiente de trabalho atual. Deve ser usado com cuidado.

Estruturas de Controle

Estruturas de Controle

- Principais comandos condicionais:
  - `if()`
  - `if()/else`
  - `ifelse()`
- Principais comandos de repetição:
  - `for()`
  - `while()`

Comandos Condicionais

- odd():

```
odd <- function(x) {
  if (x %% 2 == 1) return(TRUE)
  return(FALSE)
}
```

```
odd <- function(x) {
  if (x %% 2 == 1) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}
```

Comandos Condicionais

- `odd()`:

```r
odd <- function(x) {
  ifelse(x %% 2 == 1, TRUE, FALSE)
}
```

```r
odd <- function(x) {
  x %% 2 == 1
}
```

Comandos Condicionais

- `mymin()`:

```
1 mymin <- function(a, b) {
2   ifelse(a < b, a, b)
3 }
```

- `myabs()`:

```
1 myabs <- function(a) {
2   if (a < 0) {
3     -a
4   } else {
5     a
6   }
7 }
```

Comandos Condicionais

- Equação do 2º grau:

$$ax^2 + bx + c = 0$$

- Fórmula de Bhaskara:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

$$\Delta = b^2 - 4ac$$

Comandos Condicionais

- `bhaskara()`:

```
1  bhaskara <- function(a = 0, b = 0, c = 0) {
2    if (a != 0) {
3      delta <- as.complex(b^2 - 4*a*c)
4      if (delta != 0) {
5        c((-b + sqrt(delta)) / (2 * a),
6          (-b - sqrt(delta)) / (2 * a))
7      } else {
8        -b / (2 * a)
9      }
10   } else {
11     -c / b
12   }
13 }
```

Comandos de Repetição

- Exemplos:

```
printVector <- function(vector) {
  i <- 1
  while (i <= length(vector)) {
    print(vector[i])
    i <- i + 1
  }
}
```

```
printVector <- function(vector) {
  for (i in vector) {
    print(i)
  }
}
```

Comandos de Repetição

- `mysum()`:

```
1 mysum <- function(...) {
2   k <- 0
3   for (i in c(...)) {
4     k <- k + i
5   }
6   return(k)
7 }
```

Comandos de Repetição

- `mylength()`:

```
1 mylength <- function(vector) {
2   k <- 0
3   for (i in vector) {
4     k <- k + 1
5   }
6   return(k)
7 }
```

Comandos de Repetição

- `mylength()`:

```
1 mylength <- function(...) {
2   k <- 0
3   for (i in c(...)) {
4     k <- k + 1
5   }
6   return(k)
7 }
```

Comandos de Repetição

- `multlength()`:

```
multlength <- function(...) {
  result <- NULL
  for (i in list(...)) {
    result <- c(result, length(i))
  }
  return(result)
}
```

Comandos de Repetição

```
> length(25:30, matrix(1:12, 3, 4),
+         rnorm(5), sample(10))
Error in length(25:30, matrix(1:12, 3, 4),
  rnorm(5), sample(10)) : 4 arguments passed
  to 'length' which requires 1
```

```
> multlength(25:30, matrix(1:12, 3, 4),
+             rnorm(5), sample(10))
[1] 6 12 5 10
```

Exercícios

- Implemente uma função `myprod()` que calcule o produto de todos os elementos de um vetor dado.
- Implemente uma função `mymin()` que retorne o menor elemento de um vetor dado.
- Implemente uma função `mymean()` que retorne a média dos elementos de um vetor dado.
- Implemente uma função `mymedian()` que retorne a mediana dos elementos de um vetor dado.

Exercícios

- Implemente uma função `subset()` que, dados dois conjuntos, verifique se o primeiro é um subconjunto do segundo.
- Implemente uma função `disjoint()` que, dados dois conjuntos, verifique se os conjuntos são disjuntos.
- Implemente uma função `revstr()` que inverta uma string dada.
- Implemente uma função `repstr()` que, dados uma string, um inteiro não negativo `k` e um separador, gere uma string formada por `k` cópias da string original, cada cópia unida pelo separador fornecido. Caso um separador não seja fornecido, as cópias devem ser simplesmente grudadas entre si. Não use as funções `rep()` ou `do.call()`.

Exercícios

- Implemente uma função `count()` que, dado um vetor e um elemento, retorne o número de ocorrências do elemento no vetor.
- Implemente uma função `index()` que, dado um vetor e um elemento, retorne todas as posições do vetor que sejam iguais ao elemento.
- Implemente uma função `myunique()` que, dado um vetor, retorne um outro vetor, com os elementos do vetor original, sem repetições.
- Implemente uma função `mode()` que, dado um vetor, retorne a moda do vetor, ou seja, um vetor com o(s) elemento(s) mais frequente(s) do vetor original.

Exercícios

- Implemente uma função `gcd2()` que, dados dois números inteiros não negativos, calcule o Máximo Divisor Comum (Greatest Common Divisor) dos números dados.
- Implemente uma função `gcd()` que, dados um ou mais números inteiros não negativos, calcule o Máximo Divisor Comum (Greatest Common Divisor) dos números dados.
- Implemente uma função `lcm()` que, dados um ou mais números inteiros não negativos, calcule o Mínimo Múltiplo Comum (Least Common Multiple) dos números dados.

Algoritmo de Euclides

- O algoritmo de Euclides calcula o Máximo Divisor Comum (MDC) de dois números inteiros, sendo pelo menos um deles diferente de zero.
- Ele é um algoritmo recursivo que se baseia no fato de que $MDC(x, y) = MDC(y, x \bmod y)$, onde *mod* é a operação de resto da divisão inteira.
- Além disso, o algoritmo usa o fato de que $MDC(x, 0) = x$.
- Exemplo:
  $MDC(21, 15) = MDC(15, 21 \bmod 15) =$
  $MDC(15, 6) = MDC(6, 15 \bmod 6) =$
  $MDC(6, 3) = MDC(3, 6 \bmod 3) =$
  $MDC(3, 0) = 3$

Exercícios

- Implemente uma função `mysort()` que ordene os elementos de um vetor dado (em ordem crescente).
- Implemente uma função `dot()` que calcule o produto escalar de dois vetores dados.
- Implemente uma função `mult()` que calcule o produto de duas matrizes dadas.

Exercícios

- Escreva uma função `prime()` que dado um número inteiro ($n > 1$), verique se ele é primo.
- Escreva uma função `nprime()` que dado um número inteiro ($n > 1$), retorne um vetor com todos números primos menores ou iguais a n.
- Escreva uma função `factorize()` que dado um número inteiro ($n > 1$), retorne um vetor com os fatores primos de n.

Funções de Agrupamentos

Funções de Agrupamentos

- Comandos de repetição como `for` e `while` são muito úteis em programação em R, mas pouco adequados quando se está trabalhando iterativamente em linha de comando.
- Existem algumas funções que permitem iterar sobre dados de forma mais simples:
  - `lapply()`
  - `sapply()`
  - `apply()`
  - `mapply()`
  - `tapply()`
- Importante: estas funções não são de fato mais rápidas do que os comandos de repetição (`for` e `while`) que vimos anteriormente, mas elas permitem obter resultados de forma simples (em apenas uma linha de código).

`lapply()`

- `lapply()`: dada uma lista e uma função (ou o nome de função), aplica a função a cada um dos elementos da lista, (sempre) gerando uma lista como resultado.
- Caso o objeto fornecido como entrada não seja uma lista, ele será internamente convertido em uma lista (usando a função `as.list()`).

lapply()

- Exemplos:

```
> L <- list(a = 25:30, b = matrix(1:6, 2, 3),
+           c = rnorm(5), d = sample(10))
> lapply(L, mean)
$a
[1] 27.5
$b
[1] 3.5
$c
[1] -0.1760645
$d
[1] 5.5
```

lapply()

- Exemplos (continuação):

```
> lapply(2:4, runif)
```

```
[[1]]
[1] 0.2697742 0.3915765

[[2]]
[1] 0.1848385 0.9926485 0.1681241

[[3]]
[1] 0.8622820 0.7192296 0.3234399 0.6185043
```

lapply()

- Exemplos (continuação):

```
> lapply(2:4, runif, min = 0, max = 10)
```

```
[[1]]
[1] 5.608263 1.409612

[[2]]
[1] 5.028278 4.647778 8.922431

[[3]]
[1] 1.674359 2.597341 6.489205 7.548977
```

lapply()

- Exemplos (continuação):

```
> lapply(datasets::faithful, max)
$eruptions
[1] 5.1
$waiting
[1] 96
```

```
> lapply(faithful, min)
$eruptions
[1] 1.6
$waiting
[1] 43
```

lapply()

- Exemplos (continuação):

```
> lapply(faithful,
+        function(x) {max(x) - min(x)})
$eruptions
[1] 3.5
$waiting
[1] 53
```

`sapply()`

- `sapply()` é similar a `lapply()`, onde o resultado é simplificado, se possível.
- Se o resultado for uma lista, onde cada elemento possui tamanho 1, então um vetor é retornado.
- Se o resultado for uma lista, onde cada elemento possui o mesmo tamanho (> 1), então uma matriz é retornada.
- Se não for possível simplificar o resultado usando uma das duas regras acima, então uma lista é retornada (igual seria por `lapply()`).

sapply()

- Exemplo:

```
> L <- list(a = 25:30, b = matrix(1:6, 2, 3),
+           c = rnorm(5), d = sample(10))
```

```
> sapply(L, mean)
         a          b          c          d
27.5000000  3.5000000 -0.1760645  5.5000000
```

```
> sapply(L, range)
      a b          c  d
[1,] 25 1 -1.1135290  1
[2,] 30 6  0.8266803 10
```

sapply()

- Exemplo:

```
> lapply(faithful, range)
$eruptions
[1] 1.6 5.1
$waiting
[1] 43 96
```

```
> sapply(faithful, range)
     eruptions waiting
[1,]       1.6      43
[2,]       5.1      96
```

sapply()

```
> lapply(faithful, quantile)
$eruptions
     0%      25%      50%      75%     100%
1.60000 2.16275 4.00000 4.45425 5.10000
$waiting
     0%      25%      50%      75%     100%
     43       58       76       82       96
```

```
> sapply(faithful, quantile)
     eruptions waiting
0%     1.60000      43
25%    2.16275      58
50%    4.00000      76
75%    4.45425      82
100%   5.10000      96
```

`apply()`

- `apply()`: dada uma estrutura tabular (matriz ou data frame), uma dimensão (margem) e uma função, calcula o valor da função para todos os elementos daquela dimensão.
- Se a dimensão for igual a 1, então a função é aplicada para todas as linhas da estrutura tabular.
- Se a dimensão for igual a 2, então a função é aplicada para todas as colunas da estrutura tabular.
- No caso de matrizes com 3 ou mais dimensões, é possível indicar múltiplas dimensões de forma a selecionar as submatrizes para as quais a função será aplicada.

apply()

- Exemplos:

```
> m <- matrix(sample(12), nrow = 3, ncol = 4)
> m
     [,1] [,2] [,3] [,4]
[1,]   12    6   10    5
[2,]    8    4    9    7
[3,]    2    3    1   11
```

```
> apply(m, 1, min)      # Minimo de cada linha
[1] 5 4 1
```

```
> apply(m, 2, max)      # Maximo de cada coluna
[1] 12  6 10 11
```

- Exemplos (continuação):

```
> m <- matrix(sample(8))
> dim(m) <- c(2, 2, 2); m
, , 1
     [,1] [,2]
[1,]    4    6
[2,]    2    1
, , 2
     [,1] [,2]
[1,]    7    3
[2,]    5    8
```

apply()

- Exemplos (continuação):

```
# Matriz m (2 x 2 x 2)
, , 1                 |   , , 2
     [,1] [,2]        |        [,1] [,2]
[1,]    4    6        |   [1,]    7    3
[2,]    2    1        |   [2,]    5    8
```

```
# Media das linhas (de todas as faces)
> apply(m, 1, mean)
[1] 5 4
```

```
# Media das linhas (da primeira face)
> apply(m[ , , 1], 1, mean)
[1] 5.0 1.5
```

apply()

- Exemplos (continuação):

```
# Matriz m (2 x 2 x 2)
, , 1                 |   , , 2
     [,1] [,2]        |        [,1] [,2]
[1,]    4    6        |   [1,]    7    3
[2,]    2    1        |   [2,]    5    8
```

```
# Media das colunas (de todas as faces)
> apply(m, 2, mean)
[1] 4.5 4.5
```

```
# Media das colunas (da segunda face)
> apply(m[ , , 2], 2, mean)
[1] 6.0 5.5
```

apply()

- Exemplos (continuação):

```
> total <- sum(datasets::HairEyeColor)
```

```
> apply(HairEyeColor, 1, sum) / total
    Black     Brown       Red     Blond
0.1824324 0.4831081 0.1199324 0.2145270
```

```
> apply(HairEyeColor, 2, sum) / total
    Brown      Blue     Hazel     Green
0.3716216 0.3631757 0.1570946 0.1081081
```

```
> apply(HairEyeColor, 3, sum) / total
     Male    Female
0.4712838 0.5287162
```

Somas e Médias de Linhas e Colunas

- Como vimos anteriormente, existem funções para calcular somas e médias de linhas e colunas de estruturas tabulares:
  - `rowSums(x) = apply(x, 1, sum)`
  - `colSums(x) = apply(x, 2, sum)`
  - `rowMeans(x) = apply(x, 1, mean)`
  - `colMeans(x) = apply(x, 2, mean)`
- As quatro funções acima são otimizadas e são muito mais rápidas (para matrizes grandes) que as versões usando `apply()`.

`mapply()`

- `mapply()` é uma versão multivariada de `sapply()`.
- `mapply()` aplica uma função dada a todos os primeiros elementos dos argumentos (recebidos via `...`), a todos os segundos elementos dos argumentos, e assim por diante.
- Se necessário, `mapply()` recicla os argumentos (se todos não forem do mesmo tamanho).
- Se possível, o resultado será simplificado, como em `sapply()`.

mapply()

- Exemplos:

```
> mapply(rep, 1:3, 5:3)
[[1]]
[1] 1 1 1 1 1
[[2]]
[1] 2 2 2 2
[[3]]
[1] 3 3 3
```

```
> mapply("^", 1:6, 2:3)
[1]  1  8  9 64 25 216
```

mapply()

- Exemplos:

```
> tipo1
[1] 66 75 52 11 38 49 72 97 82 46
> tipo2
[1] 91 34 25 52 46 30 17 36 31 11
> tipo3
[1] 17 18 16 92 40 75 34 63 77 19
> tipo4
[1] 72 29 36 74 88 44 31 59 64 28
```

```
> mapply(min, tipo1, tipo2, tipo3, tipo4)
[1] 17 18 16 11 38 30 17 36 31 11
```

```
> mapply(max, tipo1, tipo2, tipo3, tipo4)
[1] 91 75 52 92 88 75 72 97 82 46
```

`tapply()`

- `tapply()`: dado um vetor, um vetor de fatores e uma função, aplica a função no vetor divido em subconjuntos, de acordo com o vetor de fatores.
- Se o segundo argumento não for um vetor de fatores, ele será convertido (internamente) usando `as.factor()`.
- Se possível, o resultado será simplificado, de forma similar à função `sapply()`.

tapply()

- Exemplo:

```
> x <- c(rnorm(100), runif(100), sample(100))
> f <- gl(n = 3, k = 100,
+         labels = c("norm", "unif", "sample"))
> tapply(x, f, range)
$norm
[1] -2.697509 2.365277
$unif
[1] 0.0004929237 0.9999519107
$sample
[1] 1 100
```

tapply()

- Exemplo (continuação):

```
> tapply(datasets::mtcars$mpg,
+        datasets::mtcars$cyl, mean)
       4        6        8
26.66364 19.74286 15.10000
```

```
> tapply(mtcars$qsec, mtcars$cyl, mean)
       4        6        8
19.13727 17.97714 16.77214
```

```
> tapply(mtcars$hp, mtcars$vs, mean)
       0        1
189.72222 91.35714
```

tapply()

- Podemos criar uma função para "discretizar" um vetor:

```
qfactor <- function(vector) {
  q <- quantile(vector)
  result <- NULL
  for (i in vector)
    if (i <= q["25%"])
      result <- c(result, "q1")
    else if (i <= q["50%"])
      result <- c(result, "q2")
    else if (i <= q["75%"])
      result <- c(result, "q3")
    else result <- c(result, "q4")
  return(as.factor(result))
}
```

tapply()

- Exemplo:

```
> tapply(mtcars$mpg, qfactor(mtcars$hp), mean)
      q1        q2        q3        q4
27.50000 21.31111 17.15000 13.41429
```

```
> tapply(mtcars$mpg, qfactor(mtcars$qsec), max)
  q1   q2   q3   q4
26.0 30.4 30.4 33.9
```

```
> tapply(mtcars$hp, qfactor(mtcars$mpg), mean)
       q1        q2        q3       q4
225.62500 163.88889 112.37500 73.5714
```

tapply()

- Exemplo (continuação):

```
> tapply(datasets::Loblolly$height,
+        datasets::Loblolly$age, min)
    3       5      10      15      20      25
 3.46    9.03   25.45   37.79   48.31   56.43
```

```
> tapply(Loblolly$height, Loblolly$age, mean)
    3       5      10      15      20      25
 4.24   10.21   27.44   40.54   51.47   60.29
```

```
> tapply(Loblolly$height, Loblolly$age, max)
    3       5      10      15      20      25
 4.81   11.37   30.21   44.40   55.82   64.10
```

tapply()

- Exemplo (continuação):

```
> tapply(datasets::airquality$Temp,
+        datasets::airquality$Month, mean)
       5        6        7        8        9
65.54839 79.10000 83.90323 83.96774 76.90000
```

```
> tapply(airquality$Solar, airquality$Month,
+        mean, na.rm = TRUE)
       5        6        7        8        9
181.2963 190.1667 216.4839 171.8571 167.4333
```

```
> tapply(airquality$Temp, airquality$Wind > 10,
+        mean)
   FALSE     TRUE
80.80247 74.59722
```

tapply()

- Exemplo (continuação):

```
> tapply(datasets::iris$Petal.Length,
+        datasets::iris$Species, mean)
    setosa versicolor  virginica
     1.462      4.260      5.552
```

```
> tapply(iris$Petal.Width, iris$Species, mean)
    setosa versicolor  virginica
     0.246      1.326      2.026
```

```
> tapply(iris$Petal.Length / iris$Petal.Width,
+        iris$Species, mean)
    setosa versicolor  virginica
  6.908000   3.242837   2.780662
```

- Exemplo (continuação):

```
> tapply(iris$Petal.Length, iris$Species,
+        summary)
$setosa
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   1.400   1.500   1.462   1.575   1.900
$versicolor
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   3.00    4.00    4.35    4.26    4.60    5.10
$virginica
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  4.500   5.100   5.550   5.552   5.875   6.900
```

tapply()

- Exemplo (continuação):

```
> simplify2array(tapply(iris$Petal.Length,
+                       iris$Species, summary))
        setosa versicolor virginica
Min.     1.000      3.00     4.500
1st Qu.  1.400      4.00     5.100
Median   1.500      4.35     5.550
Mean     1.462      4.26     5.552
3rd Qu.  1.575      4.60     5.875
Max.     1.900      5.10     6.900
```