Rudimentos da Linguagem R

Marcelo Ventura Freire (EACH/USP)

Contents

O que veremos

1. Tipos e Estruturas de Dados em R
2. Programação em R

O que veremos

1. Tipos e Estruturas de Dados em R
    1. Tipos de Dados
    2. Estruturas de Dados

    3. Atributos
    4. “Não-dados”
2. Programação em R

O que veremos

1. Tipos e estruturas de dados em R
2. Programação em R
    1. *Scripts*
    2. Comentários
    3. Funções
    4. Operadores de Atribuição
    5. Operadores de Indexação
    6. Operações e Funções Matemáticas
    7. Operações e Funções de Comparação
    8. Operações e Funções Lógicas
    9. Instruções de Repetições
    10. Instruções de Execução Condicional

Tipos e Estruturas de Dados em R

Tipos de Dados

Tipos de Dados

1. Quantitativo,
2. Textual,
3. Lógico,
4. Categórico
5. Outros

Quantitativo

O R consegue lidar números

- inteiros (*integer*)
    - `1L`, `10L`, `-5L`
- reais (*double*)
    - `1`, `10`, `10.0`, `-10.5`
- complexos (*complex*)
    - `1 + 0i`, `2 + 3i`, `1i`

Valores numéricos são *double* por padrão, a menos que explicitado outro tipo.

Quantitativo

```
2
2L
2 + 0i
```

Quantitativo

```
2
```

```
## [1] 2
```

```
2L
```

```
## [1] 2
```

```
2 + 0i
```

```
## [1] 2+0i
```

Quantitativo

```
2 + 3
2L + 3L
log(3.4)
log(3.4 + 0i)
```

Quantitativo

```
2 + 3
```

```
## [1] 5
```

```
2L + 3L
```

```
## [1] 5
```

```
log(3.4)
```

```
## [1] 1.223775
```

```
log(3.4 + 0i)
```

```
## [1] 1.223775+0i
```

Textual

O R consegue lidar com informações textuais

```
'Isto é um texto'
"Isto também"
paste("Nós somos", "três textos,", "mas vamos virar um só!")
```

Textual

O R consegue lidar com informações textuais

```
'Isto é um texto'
```

```
## [1] "Isto é um texto"
```

```
"Isto também"
```

```
## [1] "Isto também"
```

```
paste("Nós somos", "três textos,", "mas vamos virar um só!")
```

```
## [1] "Nós somos três textos, mas vamos virar um só!"
```

Lógico

O R consegue lidar com valores lógicos:

- verdadeiro (`TRUE` ou `T`)
- falso (`FALSE` ou `F`)

```
2 < 3
x <- (2 < 3)
x
```

Lógico

O R consegue lidar com valores lógicos:

- verdadeiro (`TRUE` ou `T`)
- falso (`FALSE` ou `F`)

```r
2 < 3
```

```
## [1] TRUE
```

```r
x <- (2 < 3)
x
```

```
## [1] TRUE
```

Categórico

O R consegue lidar com dados qualitativos

- nominais (através da função `factor()`)
- ordinais (através da função `ordered()`)

Dados categóricos são mais do que apenas textos em R.

Eles têm metainformação, que os dados textuais não têm.

```
Fatores = Dados + Metadados
```

Categórico

Importante

Rotina como `aov()` (que realiza ANOVA) exigem que variáveis qualitativas tenham sido armazenadas como fatores.

Se você tentar executar `aov()` com uma regressora quantitativa, vai ser retornada a tabela de ANOVA da regressão linear ao invés da tabela de ANOVA da análise de variância

```
Depois não diga que eu não avisei...
```

Categórico

```r
c("criança", "adulto", "idoso")     # só texto
factor(c("criança", "adulto", "idoso"))     # fator
```

Categórico

```r
c("criança", "adulto", "idoso")     # só texto
```

```
## [1] "criança" "adulto"  "idoso"
```

```r
factor(c("criança", "adulto", "idoso"))     # fator
```

```
## [1] criança adulto  idoso
## Levels: adulto criança idoso
```

Categórico

```r
observações <- c(3, 2, 1, 3, 2)
categs <- c("criança", "adulto", "idoso")
factor(observações, levels = 1:3, labels = categs)
ordered(observações, levels = 1:3, labels = categs)
```

Categórico

```r
observações <- c(3, 2, 1, 3, 2)
categs <- c("criança", "adulto", "idoso")
factor(observações, levels = 1:3, labels = categs)
```

```
## [1] idoso   adulto  criança idoso   adulto
## Levels: criança adulto idoso
```

```r
ordered(observações, levels = 1:3, labels = categs)
```

```
## [1] idoso   adulto  criança idoso   adulto
## Levels: criança < adulto < idoso
```

Note as categorias

Categórico

```r
observações <- c(3, 2, 1, 3, 2)
categs <- c("criança", "adulto", "idoso")
factor(observações, levels = 1:3, labels = categs, ordered = T)
ordered(observações, levels = 1:3, labels = categs)
```

Categórico

```r
observações <- c(3, 2, 1, 3, 2)
categs <- c("criança", "adulto", "idoso")
factor(observações, levels = 1:3, labels = categs, ordered = T)
```

```
## [1] idoso   adulto  criança idoso   adulto
## Levels: criança < adulto < idoso
```

```r
ordered(observações, levels = 1:3, labels = categs)
```

```
## [1] idoso   adulto  criança idoso   adulto
## Levels: criança < adulto < idoso
```

A mesma coisa

Categórico

```r
ordered(c("criança", "adulto", "idoso"))
categs <- c("criança", "adulto", "idoso")
ordered(c("criança", "adulto", "idoso"), levels = categs)
```

Categórico

```r
ordered(c("criança", "adulto", "idoso"))
```

```
## [1] criança adulto  idoso
## Levels: adulto < criança < idoso
```

```r
categs <- c("criança", "adulto", "idoso")
ordered(c("criança", "adulto", "idoso"), levels = categs)
```

```
## [1] criança adulto  idoso
## Levels: criança < adulto < idoso
```

O primeiro está errado (salvo situações excepcionais; note as categorias), mas o segundo está certo.

Outros Tipos de "Dados"

Por incrível que pareça, o R consegue tratar código executável como dado.

Outros Tipos de "Dados"

Por exemplo, já vimos

```r
x <- 2
typeof(x)
```

```
## [1] "double"
```

```r
x <- "A"
typeof(x)
```

```
## [1] "character"
```

Outros Tipos de "Dados"

Mas isto é novidade

```
x <- factor("A")
typeof(x)
```

```
## [1] "integer"
```

```
x <- factor
typeof(x)
```

```
## [1] "closure"
```

O primeiro é um dado categórico, mas o tipo do segundo é a própria função que precisamos executar para conseguir um dado categórico (identificado como `closure` pelo R).

Estruturas de Dados

Estruturas de Dados

1. Vetorial (`c()`, `vector()`),
2. Lista/"dicionário" (`list()`),
3. Matricial (`matrix()` e `array()`),
4. Conjunto de dados (`data.frame()`)

Vetores

Por padrão, os dados no R são vetores (*atomic vectors*)

```
2 + 2
```

```
## [1] 4
```

Esse `[1]` antes do resultado é a posição (primeira) do valor `4` dentro do vetor de comprimento unitário que é o resultado da operação acima.

Vetores

A função `c()` gera vetores.

```
c(1, 3, 3, 7)
```

Vetores

A função `c()` gera vetores.

```r
c(1, 3, 3, 7)
```

```
## [1] 1 3 3 7
```

Vetores

A função `c()` também junta e "achata" múltiplos vetores em um único vetor.

```r
c(c(1, 2, 3, 4), c(5, 6, 7, 8))
```

Vetores

A função `c()` também junta e "achata" múltiplos vetores em um único vetor.

```r
c(c(1, 2, 3, 4), c(5, 6, 7, 8))
```

```
## [1] 1 2 3 4 5 6 7 8
```

Vetores

O operador `:` gera sequências.

```r
1:10
1:40
```

Vetores

O operador `:` gera sequências.

```r
1:10
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```r
1:40
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
```

O `[24]` na segunda linha do resultado de `1:40` é um lembrete que o valor `24` ocupa a 24ª posição do resultado da operação.

Vetores

```r
-1:10
-1:-10
-(1:10)
-10:-1
```

Vetores

```
-1:10
```

```
##  [1] -1  0  1  2  3  4  5  6  7  8  9 10
```

```
-1:-10
```

```
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

```
-(1:10)
```

```
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

```
-10:-1
```

```
##  [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1
```

Vetores

Os dados de um vetor **têm** que ser do mesmo tipo.

Se não forem, o R converte todos para um mesmo tipo que os comporte através de *regras de coerção*.

```
c(1, 2, 3, 4)
c(1, 2, 3, 4, "A")
```

Vetores

Os dados de um vetor **têm** que ser do mesmo tipo.

Se não forem, o R converte todos para um mesmo tipo que os comporte através de *regras de coerção*.

```
c(1, 2, 3, 4)
```

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

```
c(1, 2, 3, 4, "A")
```

```
## [1] "1" "2" "3" "4" "A"
```

Vetores

Os dados de um vetor **têm** que ser do mesmo tipo.

Se não forem, o R converte todos para um mesmo tipo que os comporte através de *regras de coerção*.

```
typeof(c(1L, 2L, 3L, 4L)); typeof(c(1, 2L, 3L, 4L))
```

Vetores

Os dados de um vetor **têm** que ser do mesmo tipo.

Se não forem, o R converte todos para um mesmo tipo que os comporte através de *regras de coerção*.

```
typeof(c(1L, 2L, 3L, 4L)); typeof(c(1, 2L, 3L, 4L))
```

```
## [1] "integer"
```

```
## [1] "double"
```

Vetores

Os dados de um vetor **têm** que ser do mesmo tipo.

Se não forem, o R converte todos para um mesmo tipo que os comporte através de *regras de coerção*.

```
c(1L, 2, 3 + 0i)
c(1L, 2, 3 + 0i, "A")
```

Vetores

Os dados de um vetor **têm** que ser do mesmo tipo.

Se não forem, o R converte todos para um mesmo tipo que os comporte através de *regras de coerção*.

```
c(1L, 2, 3 + 0i)
```

```
## [1] 1+0i 2+0i 3+0i
```

```
c(1L, 2, 3 + 0i, "A")
```

```
## [1] "1"    "2"    "3+0i" "A"
```

Listas

Informalmente, as listas podem ser pensadas como vetores que comportam dados de diferentes tipos sem que eles sejam forçados a terem o mesmo tipo em comum.

```
list(1, 2)
list(1, "a")
```

Listas

Informalmente, as listas podem ser pensadas como vetores que comportam dados de diferentes tipos sem que eles sejam forçados a terem o mesmo tipo em comum.

```r
list(1, 2)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

```r
list(1, "a")
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
```

Esse `[[2]]` indica que a 2ª posição da lista é ocupada por um vetor.

O `[1]` seguinte indica que a 1ª posição desse vetor é o valor `2` apresentado.

Listas

É possível aninhar listas dentro de listas.

```r
list(list(1, 2), c(1, 2))
```

Listas

É possível aninhar listas dentro de listas.

```r
list(list(1, 2), c(1, 2))
```

```
## [[1]]
## [[1]][[1]]
## [1] 1
##
## [[1]][[2]]
## [1] 2
##
##
## [[2]]
## [1] 1 2
```

Listas

É possível dar nomes aos elementos de uma lista.

```r
list(x = 1, y = 2, tipo = "novo")
```

Listas

É possível aninhar listas dentro de listas.

```r
list(x = 1, y = 2, tipo = "novo")
```

```
## $x
## [1] 1
##
## $y
## [1] 2
##
## $tipo
## [1] "novo"
```

Listas

```r
list(list(list(1, 2), 3))
```

Listas

```r
list(list(list(1, 2), 3))
```

```
## [[1]]
## [[1]][[1]]
## [[1]][[1]][[1]]
## [1] 1
##
## [[1]][[1]][[2]]
## [1] 2
##
##
## [[1]][[2]]
## [1] 3
```

Listas

Mas dá para "achatar" a lista em um vetor

```r
unlist(list(list(list(1, 2), 3)))
unlist(list(x = 1, y = 2, tipo = "novo"))
```

Listas

Mas dá para "achatar" a lista em um vetor

```
unlist(list(list(list(1, 2), 3)))
```

```
## [1] 1 2 3
```

```
unlist(list(x = 1, y = 2, tipo = "novo"))
```

```
##      x      y   tipo
##    "1"    "2" "novo"
```

Matrizes e *Arrays*

Matrizes podem ser pensadas como vetores com duas "direções" e *arrays* como vetores com várias "direções".

```
matrix(c(1, 2, 3, 4), nrow = 2)
matrix(1:8, nrow = 2)
```

Matrizes e *Arrays*

Matrizes podem ser pensadas como vetores com duas "direções" e *arrays* como vetores com várias "direções".

```
matrix(c(1, 2, 3, 4), nrow = 2)
```

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

```
matrix(1:8, nrow = 2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

Matrizes e *Arrays*

Matrizes podem ser pensadas como vetores com duas "direções" e *arrays* como vetores com várias "direções".

```
array(1:12, dim = c(2, 3, 2))
```

Matrizes e *Arrays*

Matrizes podem ser pensadas como vetores com duas "direções" e *arrays* como vetores com várias "direções".

```r
array(1:12, dim = c(2, 3, 2))
```

```
## , , 1
##
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
```

Matrizes e *Arrays*

Na verdade, vetores são *arrays* unidimensionais e matrizes são *arrays* bidimensionais.

```r
array(1:4, dim = c(4))
array(1:4, dim = c(2,2))
```

Matrizes e *Arrays*

Na verdade, vetores são *arrays* unidimensionais e matrizes são *arrays* bidimensionais.

```r
array(1:4, dim = c(4))
```

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

```r
array(1:4, dim = c(2,2))
```

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

Matrizes e *Arrays*

Por motivos históricos, as matrizes são preenchidas no R por colunas ao invés de ser por linhas, mas isso pode ser mudado.

```r
matrix(1:9, nrow = 3)
matrix(1:9, nrow = 3, byrow = T)
```

Matrizes e *Arrays*

Por motivos históricos, as matrizes são preenchidas no R por colunas ao invés de ser por linhas, mas isso pode ser mudado.

```r
matrix(1:9, nrow = 3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```r
matrix(1:9, nrow = 3, byrow = T)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

Matrizes e *Arrays*

Assim, como os vetores, os valores de uma matriz e *array* têm que ser do mesmo tipo ou então eles são forçados por regras de coreção.

```r
array(c(1, 2, 3, "A"), dim = c(2,2))
```

Matrizes e *Arrays*

Assim, como os vetores, os valores de uma matriz e *array* têm que ser do mesmo tipo ou então eles são forçados por regras de coreção.

```r
array(c(1, 2, 3, "A"), dim = c(2,2))
```

```
##      [,1] [,2]
## [1,] "1"  "3"
## [2,] "2"  "A"
```

Data Frames

No R, o típico conjunto de dados é um *data frame*.

Criamos um *data frame* com a função `data.frame()`

```r
data.frame(
    nome = c("Ana", "Bia", "Ciro"),
    idade = c(20, 22, 27),
    sexo = factor(c("f", "f", "m"))
)
```

Data Frames

No R, o típico conjunto de dados é um *data frame*.

Criamos um *data frame* com a função `data.frame()`

```r
data.frame(
    nome = c("Ana", "Bia", "Ciro"),
    idade = c(20, 22, 27),
    sexo = factor(c("f", "f", "m"))
)
```

```
##   nome idade sexo
## 1  Ana    20    f
## 2  Bia    22    f
## 3 Ciro    27    m
```

Data Frames

Os argumentos de `data.frame()` são as variáveis (colunas) do conjunto de dados e os seus nomes podem ser dados através dos nomes dos argumentos passados.

Todos os argumentos de `data.frame()` devem ser vetores com o mesmo comprimento, correspondendo ao número de observações no conjunto de dados

Data Frames

```r
conjdados <-
    data.frame(
        nome = c("Ana", "Bia", "Ciro"),
        idade = c(20, 22, 27),
        sexo = factor(c("f", "f", "m"))
    )
conjdados
```

Data Frames

```r
conjdados <-
    data.frame(
        nome = c("Ana", "Bia", "Ciro"),
        idade = c(20, 22, 27),
        sexo = factor(c("f", "f", "m"))
    )
conjdados
```

```
##   nome idade sexo
## 1  Ana    20    f
## 2  Bia    22    f
## 3 Ciro    27    m
```

Data Frames

Um *data frame* pode ser tratado como lista e também como matriz.

- aceita `[]` com um vetor de inteiros ou de strings e retorna um *data frame* (que nem uma lista)
- aceita `[[]]` com um inteiro ou string e retorna um vetor (que nem uma lista)
- aceita `[ , ]` com um par de vetores de inteiros ou de strings e retorna um *data frame* (que nem uma matriz)

Data Frames

```
conjdados[1] # data frame
conjdados[1:2] # data frame
```

Data Frames

```
conjdados[1] # data frame
```

```
##   nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```
conjdados[1:2] # data frame
```

```
##   nome idade
## 1  Ana    20
## 2  Bia    22
## 3 Ciro    27
```

Data Frames

```
conjdados[1] # data frame
conjdados[[1]] # vetor
```

Data Frames

```
conjdados[1] # data frame
```

```
##   nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```
conjdados[[1]] # vetor
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

Data Frames

```
conjdados[1] # data frame
conjdados[1, 1] # vetor
```

Data Frames

```
conjdados[1] # data frame
```

```
##   nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```
conjdados[1, 1] # vetor
```

```
## [1] Ana
## Levels: Ana Bia Ciro
```

Data Frames

```
conjdados[1, 1] # vetor
conjdados[1:2, 1:2] # data frame
```

Data Frames

```
conjdados[1, 1] # vetor
```

```
## [1] Ana
## Levels: Ana Bia Ciro
```

```
conjdados[1:2, 1:2] # data frame
```

```
##   nome idade
## 1  Ana    20
## 2  Bia    22
```

Data Frames

```
conjdados[1:2, 1] # vetor
conjdados[1:2, 1:2] # data frame
```

Data Frames

```
conjdados[1:2, 1] # vetor
```

```
## [1] Ana Bia
## Levels: Ana Bia Ciro
```

```
conjdados[1:2, 1:2] # data frame
```

```
##   nome idade
## 1  Ana    20
## 2  Bia    22
```

Data Frames

```
conjdados[1:2, 1] # vetor
conjdados[1:2, "Nome"] # vetor
```

Data Frames

```
conjdados[1:2, 1] # vetor
```

```
## [1] Ana Bia
## Levels: Ana Bia Ciro
```

```
conjdados[1:2, "Nome"] # vetor
```

```
## NULL
```

Atributos

Atributos

As metainformações dos tipos de dados e das estruturas de dados são armazenados na forma de atributos.

1. `attributes()` e `str()`
2. `names()`,
3. `levels()` e `nlevels()`
4. `dim()` e `dimnames()`
5. `rownames()` e `colnames()`
6. `ncols()` e `nrows()`

`attributes()` e `str()`

```r
attributes(conjdados)
```

attributes() e str()

```r
attributes(conjdados)
```

```
## $names
## [1] "nome"  "idade" "sexo"
##
## $row.names
## [1] 1 2 3
##
## $class
## [1] "data.frame"
```

attributes() e str()

```r
str(conjdados)
```

attributes() e str()

```r
str(conjdados)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ nome : Factor w/ 3 levels "Ana","Bia","Ciro": 1 2 3
##  $ idade: num  20 22 27
##  $ sexo : Factor w/ 2 levels "f","m": 1 1 2
```

names()

```r
names(conjdados)
names(conjdados) <- c("Nome", "Idade", "Sexo")
conjdados
```

names()

```r
names(conjdados)
```

```
## [1] "nome"  "idade" "sexo"
```

```r
names(conjdados) <- c("Nome", "Idade", "Sexo")
conjdados
```

```
##   Nome Idade Sexo
## 1  Ana    20    f
## 2  Bia    22    f
## 3 Ciro    27    m
```

levels() e nlevels()

```r
grupo.etário <-
    ordered(c(3, 2, 1, 3, 2),
            levels = 1:3,
            labels = c("criança", "adulto", "idoso"))
grupo.etário
```

levels() e nlevels()

```r
grupo.etário <-
    ordered(c(3, 2, 1, 3, 2),
            levels = 1:3,
            labels = c("criança", "adulto", "idoso"))
grupo.etário
```

```
## [1] idoso   adulto  criança idoso   adulto
## Levels: criança < adulto < idoso
```

levels() e nlevels()

```r
levels(grupo.etário)
nlevels(grupo.etário)
```

levels() e nlevels()

```r
levels(grupo.etário)
```

```
## [1] "criança" "adulto"  "idoso"
```

```r
nlevels(grupo.etário)
```

```
## [1] 3
```

levels() e nlevels()

```
levels(grupo.etário) <- c("CRI", "ADU", "IDO")
grupo.etário
```

levels() e nlevels()

```
levels(grupo.etário) <- c("CRI", "ADU", "IDO")
grupo.etário
```

```
## [1] IDO ADU CRI IDO ADU
## Levels: CRI < ADU < IDO
```

levels() e nlevels()

```
levels(grupo.etário) <- c("criança", "adulto", "idoso", "imortal")
grupo.etário
nlevels(grupo.etário)
```

levels() e nlevels()

```
levels(grupo.etário) <- c("criança", "adulto", "idoso", "imortal")
grupo.etário
```

```
## [1] idoso   adulto  criança idoso   adulto
## Levels: criança < adulto < idoso < imortal
```

```
nlevels(grupo.etário)
```

```
## [1] 4
```

dim() e dimnames()

```
dim(conjdados)
dimnames(conjdados)
```

dim() e dimnames()

```
dim(conjdados)
```

```
## [1] 3 3
```

```r
dimnames(conjdados)
```

```
## [[1]]
## [1] "1" "2" "3"
##
## [[2]]
## [1] "Nome"  "Idade" "Sexo"
```

rownames() e colnames()

```r
rownames(conjdados)
colnames(conjdados)
```

rownames() e colnames()

```r
rownames(conjdados)
```

```
## [1] "1" "2" "3"
```

```r
colnames(conjdados)
```

```
## [1] "Nome"  "Idade" "Sexo"
```

ncols() e nrows()

```r
ncol(conjdados)
nrow(conjdados)
```

ncols() e nrows()

```r
ncol(conjdados)
```

```
## [1] 3
```

```r
nrow(conjdados)
```

```
## [1] 3
```

"Não-Dados" do R

Os "Não-Dados" do R

1. Dado faltante (*missing*): `NA` – *Not Available*
2. Dado não numérico: `NaN` – *Not a Number*
3. Valor Infinito: `Inf`
4. Ausência de valor: `NULL`

Dado Faltante

```
idades <- c(18, 28, NA, 32)
idades
log(idades)
```

Dado Faltante

```
idades <- c(18, 28, NA, 32)
idades
```

```
## [1] 18 28 NA 32
```

```
log(idades)
```

```
## [1] 2.890372 3.332205       NA 3.465736
```

Dado Não Numérico

```
idades <- c(18, 28, -2, 32)
idades
log(idades)
```

Dado Não Numérico

```
idades <- c(18, 28, -2, 32)
idades
```

```
## [1] 18 28 -2 32
```

```
log(idades)
```

```
## Warning in log(idades): NaNs produzidos
```

```
## [1] 2.890372 3.332205      NaN 3.465736
```

Valor Infinito

```
idades <- c(18, 28, 0, 32)
idades
log(idades)
```

Valor Infinito

```
idades <- c(18, 28, 0, 32)
idades
```

```
## [1] 18 28  0 32
```

```
log(idades)
```

```
## [1] 2.890372 3.332205     -Inf 3.465736
```

Ausência de Valor

```
c()
```

Ausência de Valor

```
c()
```

```
## NULL
```

Programação em R

Programação em R

1. *Scripts*
2. Comentários
3. Funções
4. Operadores de atribuição
5. Operadores de indexação
6. Operações e funções matemáticas
7. Operações e funções de comparação
8. Operações e funções lógicas
9. Instruções de repetição
10. Instruções de execução condicional

Scripts

Scripts

Em R, tudo é uma expressão e, nesse sentido, o próprio R é apenas uma grande calculadora, que lê expressões e as avalia, retornando um valor ou objeto.

Chamamos de ***script*** ou **programa** o arquivo de texto que contém o conjunto de operações e expressões em R a serem realizadas no seu conjunto de dados e que resultarão em tabelas, gráficos ou resultados.

Esse arquivo é geralmente identificado pelas extensões `.R` ou `.r`

Scripts

O comando `source("calcula.R")` executa o *script* que está armazenado no diretório ou pasta atual com o nome `analise.R`.

```
writeLines(readLines("calcula.R"))
```

```
## x <- 2 + 2
## print(x)
```

```
source("calcula.R")
```

```
## [1] 4
```

O R lê e avalia sequencialmente as operações que estão no arquivo `calcula.R`.

Comentários

Comentários

É possível armazenar informações importantes sobre o seu *script* dentro do seu próprio *script*, de modo que você consiga lembrar quando você precisar no futuro quando você já tiver esquecido.

Para isso, basta colocar o símbolo `#` em qualquer ponto do seu *script*, que o R ignorará tudo o que estiver escrito do símbolo `#` até o final da linha.

```
print("Isto será impresso")  # mas isto aqui sequer será avaliado pelo R
```

```
## [1] "Isto será impresso"
```

Comentários

DICA

DOCUMENTE O SEU *SCRIPT* COM COMENTÁRIOS

SEMPRE

É sério.

Seis meses após você ter escrito o seu programa (tipo, quando você estiver escrevendo a sua monografia, dissertação ou tese...), você **NÃO VAI LEMBRAR** por que catzo você fez as escolhas que você fez.

Funções

Funções em R

Em R, é possível criar um objeto que realiza uma mesma sequência de operações fixas com um conjunto de objetos do R quaisquer que você indique.

Chamamos esse tipo de objeto uma **função**.

Uma função pode recebe como **argumentos** ou **parâmetros** um certo número de objetos, realiza operações e pode retornar um objeto como resultado das operações realizadas.

Funções em R

Por exemplo, a função `log()` recebeu um argumento numérico 10 e retornou o valor do logaritmo neperiano (i.e., de base e=2.718281828) do seu argumento

```
log(10)
```

Funções em R

Por exemplo, a função `log()` recebeu um argumento numérico 10 e retornou o valor do logaritmo neperiano (i.e., de base e=2.718281828) do seu argumento

```
log(10)
```

```
## [1] 2.302585
```

Funções em R

A função `log()` também poderia ter recebido dois argumentos numéricos: o valor cujo logaritmo é desejado e a *base* desejada do logaritmo.

```
log(10, 10)
log(10, 2)
```

Funções em R

A função `log()` também poderia ter recebido dois argumentos numéricos: o valor cujo logaritmo é desejado e a *base* desejada do logaritmo.

```
log(10, 10)
```

```
## [1] 1
```

```
log(10, 2)
```

```
## [1] 3.321928
```

Usando Funções Existentes

Como já vimos, para executar uma função existente, fazemos uma referência ao seu nome seguido de parênteses cercando os argumentos da função (caso haja).

Caso avaliemos apenas o nome da função *sem* os parênteses, o objeto é retornado e veremos o código da função.

```
log       #  isto não é uma chamada à função logarítmica
log(10)   # já isto sim, pois há os parênteses
```

Usando Funções Existentes

Como já vimos, para executar uma função existente, fazemos uma referência ao seu nome seguido de parênteses cercando os argumentos da função (caso haja).

Caso avaliemos apenas o nome da função *sem* os parênteses, o objeto é retornado e veremos o código da função.

```
log       #  isto não é uma chamada à função logarítmica
```

```
## function (x, base = exp(1))  .Primitive("log")
```

```
log(10)   # já isto sim, pois há os parênteses
```

```
## [1] 2.302585
```

Usando Funções Existentes

Note que os argumentos de `log()` têm nomes e que `log()` tem um valor padrão para o argumento `base`, caso esse argumento seja omitido, mas não tem um valor padrão para o argumento `x`.

Casos os nomes sejam omitidos, os argumentos são avaliados na ordem em que aparecem, mas se os nomes estiverem presentes, a ordem com a qual aparecem não importa.

Usando Funções Existentes

```
log(10, 2)
log(x = 10, base = 2)
log(base = 2, x = 10)
```

Usando Funções Existentes

```
log(10, 2)
```

```
## [1] 3.321928
```

```r
log(x = 10, base = 2)
```

```
## [1] 3.321928
```

```r
log(base = 2, x = 10)
```

```
## [1] 3.321928
```

Usando Funções Existentes

Se omitíssemos o argumento `x`, que não tem um valor padrão para o caso de omissão, receberíamos uma mensagem do R

```r
log()
log(base = 10)
```

Usando Funções Existentes

Se omitíssemos o argumento `x`, que não tem um valor padrão para o caso de omissão, receberíamos uma mensagem do R

```r
log()
```

```
## Error in eval(expr, envir, enclos): argumento "x" ausente, sem padrão
```

```r
log(base = 10)
```

```
## Error in eval(expr, envir, enclos): argumento "x" ausente, sem padrão
```

Usando Funções Existentes

Para descobrir quais parâmetros têm valor padrão e quais não, podemos usar o *help* (no caso de funções dos pacotes) ou ver a definição da função

```r
log
```

Usando Funções Existentes

Para descobrir quais parâmetros têm valor padrão e quais não, podemos usar o *help* (no caso de funções dos pacotes) ou ver a definição da função

```r
log
```

```
## function (x, base = exp(1))  .Primitive("log")
```

Note que `base` é seguido de `=exp(1)` (que é o seu valor padrão), ao passo que `x` não.

Na prática, um parâmetro com valor padrão é um parâmetro opcional da função.

Criando as suas Próprias Funções

Você cria uma função usando `function` seguido de parênteses e de uma expressão que será avaliada toda vez que a função for chamada e será retornada como o valor da função.

```
f <- function() return(expressão)
```

ou simplesmente

```
f <- function() expressão
```

Essa expressão é chamada de **corpo** da função.

Criando as suas Próprias Funções

Se você quiser que sua função receba um ou mais argumentos para serem utilizados no corpo da função, você deve incluí-los entre os parênteses após `function`.

```
f <- function(x) x + 2
g <- function(x, y) x * y / 2
f(5)
g(3, 7)
```

Criando as suas Próprias Funções

Se você quiser que sua função receba um ou mais argumentos para serem utilizados no corpo da função, você deve incluí-los entre os parênteses após `function`.

```
f <- function(x) x + 2
g <- function(x, y) x * y / 2
f(5)
```

```
## [1] 7
```

```
g(3, 7)
```

```
## [1] 10.5
```

Criando as suas Próprias Funções

Se chamar a função com mais ou menos argumentos do que a função espera receber, você receberá uma mensagem de erro.

```
f(5, 3)
g(2)
```

Criando as suas Próprias Funções

Se chamar a função com mais ou menos argumentos do que a função espera receber, você receberá uma mensagem de erro.

```
f(5, 3)
```

```
## Error in f(5, 3): unused argument (3)
```

```
g(2)
```

```
## Error in g(2): argumento "y" ausente, sem padrão
```

Criando as suas Próprias Funções

Se você escolher definir um valor padrão para um argumento na definição da função, então ele pode ser omitido sem erro quando a função for executada.

```
g <- function(x, y = 7) x * y / 2
g(3, 7)
g(3)
```

Criando as suas Próprias Funções

Se você escolher definir um valor padrão para um argumento na definição da função, então ele pode ser omitido sem erro quando a função for executada.

```
g <- function(x, y = 7) x * y / 2
g(3, 7)
```

```
## [1] 10.5
```

```
g(3)
```

```
## [1] 10.5
```

Criando as suas Próprias Funções

Você pode precisar realizar várias operações intermediárias antes de chegar ao resultado que você pretende retornar na sua função.

Nesse caso, é possível executar essas várias operações dentro da função (inclusive criando variáveis temporárias, que serão descartadas após o término da execução da função) e retornar apenas o resultado da última expressão executada ou o resultado da expressão passada para o `return()`.

Criando as suas Próprias Funções

Basta envolver entre chaves `{}` o conjunto de expressões a serem consideradas como o corpo da função.

Para separar as expressões do corpo da função (dentro do `{}`), basta deixar cada expressão em uma linha separada.

```
g <- function(x) {
    print(summary(x))
    hist(x)
    sum((x ^ 2) * log(x + 3))
}
```

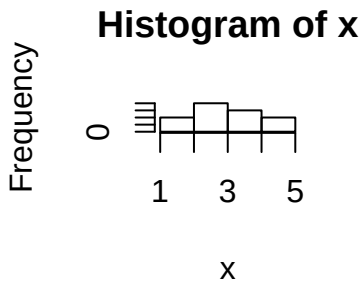
Criando as suas Próprias Funções

```
g(c(2,3,1,4,3,5,3,4,5,3,4))
```

Criando as suas Próprias Funções

```
g(c(2,3,1,4,3,5,3,4,5,3,4))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   1.000   3.000   3.000   3.364   4.000   5.000
```




```
## [1] 269.7032
```

Criando as suas Próprias Funções

Se for o caso, é possível deixar mais de uma expressão na mesma linha, desde que separadas por ponto e vírgula `;`.

```
g <- function(x) {print(summary(x)); hist(x); sum((x ^ 2) * log(x + 3))}
```

Criando as suas Próprias Funções

Caso uma expressão fique muito grande para ficar em uma única linha, basta pular para a linha seguintes ***desde que*** o R perceba que a expressão ainda não acabou, como, por exemplo,

```r
h <- function(x) {              # o R continua lendo na próxima linha,
                                # pois falta o corpo da função
    x ^ 3 + 2 * x ^ 2 +         # idem, pois falta um valor após o +
    30 * x - 50 * x ^ (1 / 2) # acabou
}
h(2)
```

Criando as suas Próprias Funções

Caso uma expressão fique muito grande para ficar em uma única linha, basta pular para a linha seguintes ***desde que*** o R perceba que a expressão ainda não acabou, como, por exemplo,

```r
h <- function(x) {              # o R continua lendo na próxima linha,
                                # pois falta o corpo da função
    x ^ 3 + 2 * x ^ 2 +         # idem, pois falta um valor após o +
    30 * x - 50 * x ^ (1 / 2) # acabou
}
h(2)
```

```
## [1] 5.289322
```

Criando as suas Próprias Funções

Além de definir o valor de retorno, `return()` também encerra a execução da função e volta para a chamada da função ao invés de executar o restante do corpo da função.

```r
h <- function(x) {
    if (x >= 0)
        return(x)
    else
        return(-x)
}
```

Criando as suas Próprias Funções

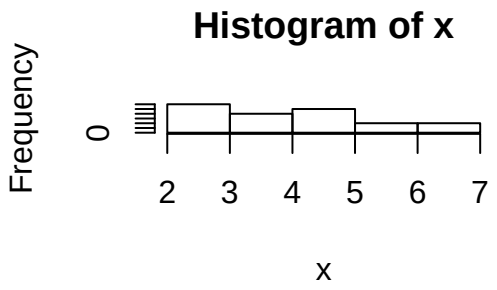
Se você não quiser retornar valor nenhum, simplesmente use `return()` sem nada dentro dos parênteses.

```r
h <- function(x) {
    hist(x)
    return()
}
h(c(2,5,3,5,7,4,6,7,3,6,4,5,3,5,3,5,4,2,4))
```

Criando as suas Próprias Funções

Se você não quiser retornar valor nenhum, simplesmente use `return()` sem nada dentro dos parênteses.

```r
h <- function(x) {
    hist(x)
    return()
}
h(c(2,5,3,5,7,4,6,7,3,6,4,5,3,5,3,5,4,2,4))
```




```
## NULL
```

Operadores de Atribuição

Operadores de Atribuição

Como já vimos, o operador `<-` associa um objeto do R a um nome.

Mas há também o operador `->`

```r
x <- 2
2 -> y
x == y
```

Operadores de Atribuição

Como já vimos, o operador `<-` associa um objeto do R a um nome.

Mas há também o operador `->`

```r
x <- 2
2 -> y
x == y
```

```
## [1] TRUE
```

Operadores de Indexação

Operadores de Indexação

1. Os operadores `[]`, `[[]]` e `$`
2. Indexando nas linhas e nas colunas
3. Indexando com vetores numéricos positivos e negativos
4. Indexando com vetores lógicos
5. Indexando com vetores textuais
6. A opção `drop = F` do operador `[]`

Operador []

O operador `[]` pode ser usado de várias formas

- com vetor
- com matriz
- com *array*
- com lista
- com *data frame*

Operador [] com vetor

```
dado <- 1:10
dado
dado[2]
dado[c(2, 5)]
```

Operador [] com vetor

```
dado <- 1:10
dado
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
dado[2]
```

```
## [1] 2
```

```
dado[c(2, 5)]
```

```
## [1] 2 5
```

Operador [] com matriz

```
dado <- matrix(1:9, ncol = 3)
dado
dado[1] # retorna elemento
dado[4]
```

Operador [] com matriz

```
dado <- matrix(1:9, ncol = 3)
dado
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
dado[1] # retorna elemento
```

```
## [1] 1
```

```
dado[4]
```

```
## [1] 4
```

Operador [] com matriz

```
dado[1:4]  # retorna elementos
dado[1, 1] # retorna elemento
dado[1, 2]
dado[2, 1]
```

Operador [] com matriz

```
dado[1:4]  # retorna elementos
```

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

```
dado[1, 1] # retorna elemento
```

```
## [1] 1
```

```
dado[1, 2]
```

```
## [1] 4
```

```
dado[2, 1]
```

```
## [1] 2
```

Operador [] com matriz

```
dado[2:3, 1:2] # retorna matriz
dado[2, ] # retorna linha como vetor
dado[, 1] # retorna coluna como vetor
```

Operador [] com matriz

```r
dado[2:3, 1:2] # retorna matriz
```

```
##      [,1] [,2]
## [1,]    2    5
## [2,]    3    6
```

```r
dado[2, ] # retorna linha como vetor
```

```
## [1] 2 5 8
```

```r
dado[, 1] # retorna coluna como vetor
```

```
## [1] 1 2 3
```

Operador [] com matriz

Opção drop = FALSE

As expressões `dado[2, ]` e `dado[, 1]` retornaram dois vetores, ao invés de terem retornado matrizes $1x3$ e $3x1$ respectivamente.

Isso ocorreu porque, por padrão, o operador `[]` descarta a estrutura se o resultado da expressão for uma matriz linha ou matriz coluna.

Para evitar esse comportamento padrão, podemos usar a opção `drop = FALSE` do operador `[]`

Operador [] com matriz

Opção drop = FALSE

```r
dado[2, ] # retorna linha como vetor
dado[2, , drop = F] # retorna matriz
```

Operador [] com matriz

Opção drop = FALSE

```r
dado[2, ] # retorna linha como vetor
```

```
## [1] 2 5 8
```

```r
dado[2, , drop = F] # retorna matriz
```

```
##      [,1] [,2] [,3]
## [1,]    2    5    8
```

Operador [] com matriz

Opção drop = FALSE

```
dado[, 1] # retorna coluna como vetor
dado[, 1, drop = F] # retorna matriz
```

Operador [] com matriz

Opção drop = FALSE

```
dado[, 1] # retorna coluna como vetor
```

```
## [1] 1 2 3
```

```
dado[, 1, drop = F] # retorna matriz
```

```
##      [,1]
## [1,]    1
## [2,]    2
## [3,]    3
```

Operador [] com *array*

```
tensor <-
    array(1:8, dim = c(2,2,2))
tensor
```

Operador [] com *array*

```
tensor <-
    array(1:8, dim = c(2,2,2))
tensor
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
```

Operador [] com *array*

```
tensor[1, 1, 1] # retorna elemento
tensor[1, 1, 2]
tensor[1, 2, 1]
tensor[2, 1, 1]
```

Operador [] com *array*

```
tensor[1, 1, 1] # retorna elemento
```

```
## [1] 1
```

```
tensor[1, 1, 2]
```

```
## [1] 5
```

```
tensor[1, 2, 1]
```

```
## [1] 3
```

```
tensor[2, 1, 1]
```

```
## [1] 2
```

Operador [] com *array*

```
tensor[1, 1, ] # retorna vetor
tensor[1, , 1]
tensor[, 1, 1]
```

Operador [] com *array*

```
tensor[1, 1, ] # retorna vetor
```

```
## [1] 1 5
```

```
tensor[1, , 1]
```

```
## [1] 1 3
```

```
tensor[, 1, 1]
```

```
## [1] 1 2
```

Operador [] com *array*

Opção `drop = FALSE`

Da mesma forma que para matrizes, o operador `[]` também descarta a estrutura por padrão se o resultado for um *array* unidimensional.

Também aqui, é possível usar a opção `drop = FALSE`

Operador [] com *array*

Opção `drop = FALSE`

```
tensor[1, 1, , drop = F] # retorna array
```

Operador [] com *array*

Opção `drop = FALSE`

```
tensor[1, 1, , drop = F] # retorna array
```

```
## , , 1
##
##      [,1]
## [1,]    1
##
## , , 2
##
##      [,1]
## [1,]    5
```

Operador [] com *array*

Opção `drop = FALSE`

```
tensor[1, , 1, drop = F] # retorna array
```

Operador [] com *array*

Opção `drop = FALSE`

```
tensor[1, , 1, drop = F] # retorna array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
```

Operador [] com *array*

Opção `drop = FALSE`

```
tensor[, 1, 1, drop = F] # retorna array
```

Operador [] com *array*

Opção `drop = FALSE`

```
tensor[, 1, 1, drop = F] # retorna array
```

```
## , , 1
##
##      [,1]
## [1,]    1
## [2,]    2
```

Operador [] com *array*

```
tensor[1, , ] # retorna matriz
tensor[, 1, ]
tensor[, , 1]
```

Operador [] com *array*

```
tensor[1, , ] # retorna matriz
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    3    7
```

```
tensor[, 1, ]
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
```

```
tensor[, , 1]
```

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

Operador [] com *array*

```r
tensor[1, , ] # retorna matriz
tensor[1] # retorna elemento
tensor[1, ] # dá erro
```

Operador [] com *array*

```r
tensor[1, , ] # retorna matriz
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    3    7
```

```r
tensor[1] # retorna elemento
```

```
## [1] 1
```

```r
tensor[1, ] # dá erro
```

```
## Error in tensor[1, ]: número incorreto de dimensões
```

Operador [] com lista

```r
lista <- list(1, 2, 3)
lista
```

Operador [] com lista

```r
lista <- list(1, 2, 3)
lista
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
```

Operador [] com lista

```r
lista[1] # retorna lista
lista[1:2] # retorna lista
lista[c(1, 2)] # o mesmo resultado
```

Operador [] com lista

```r
lista[1] # retorna lista
```

```
## [[1]]
## [1] 1
```

```r
lista[1:2] # retorna lista
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

```r
lista[c(1, 2)] # o mesmo resultado
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

Operador [] com lista

Aqui, `drop = FALSE` não surte efeito algum.

```r
lista[c(1, 2)]
lista[c(1, 2), drop = F]
```

Operador [] com lista

Aqui, `drop = FALSE` não surte efeito algum.

```r
lista[c(1, 2)]
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

```r
lista[c(1, 2), drop = F]
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

Operador [] com lista

```r
lista[c(1, 2)] # retorna lista
lista[1, 2] # dá erro
```

Operador [] com lista

```r
lista[c(1, 2)] # retorna lista
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

```r
lista[1, 2] # dá erro
```

```
## Error in lista[1, 2]: número incorreto de dimensões
```

Operador [] com lista

Nem mesmo com

```r
lista <- list(list(1, 2), list(1, 2))
lista
lista[1, 2]
```

Operador [] com lista

Nem mesmo com

```r
lista <- list(list(1, 2), list(1, 2))
lista
```

```
## [[1]]
## [[1]][[1]]
## [1] 1
```

```
##
## [[1]][[2]]
## [1] 2
##
##
## [[2]]
## [[2]][[1]]
## [1] 1
##
## [[2]][[2]]
## [1] 2
```

```
lista[1, 2]
```

```
## Error in lista[1, 2]: número incorreto de dimensões
```

Operador [] com lista

Nem mesmo com

```
lista[1, 2]
```

Operador [] com lista

Nem mesmo com

```
lista[1, 2]
```

```
## Error in lista[1, 2]: número incorreto de dimensões
```

Operador [] com lista

```
lista[1, 2] # dá erro
lista[1] # retorna lista
```

Operador [] com lista

```
lista[1, 2] # dá erro
```

```
## Error in lista[1, 2]: número incorreto de dimensões
```

```
lista[1] # retorna lista
```

```
## [[1]]
## [[1]][[1]]
## [1] 1
##
## [[1]][[2]]
## [1] 2
```

Operador [] com *data frame*

```
conjdados # retorna data frame
conjdados[1, 1] # retorna vetor de fator
conjdados[3, 2] # retorna vetor numérico
```

Operador [] com *data frame*

```
conjdados # retorna data frame
```

```
##   Nome Idade Sexo
## 1  Ana    20    f
## 2  Bia    22    f
## 3 Ciro    27    m
```

```
conjdados[1, 1] # retorna vetor de fator
```

```
## [1] Ana
## Levels: Ana Bia Ciro
```

```
conjdados[3, 2] # retorna vetor numérico
```

```
## [1] 27
```

Operador [] com *data frame*

```
conjdados[1] # retorna data frame
conjdados[1, ] # retorna data frame
conjdados[, 1] # retorna vetor
conjdados[1, 1] # retorna vetor
```

Operador [] com *data frame*

```
conjdados[1] # retorna data frame
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```
conjdados[1, ] # retorna data frame
```

```
##   Nome Idade Sexo
## 1  Ana    20    f
```

```r
conjdados[, 1] # retorna vetor
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

```r
conjdados[1, 1] # retorna vetor
```

```
## [1] Ana
## Levels: Ana Bia Ciro
```

Operador [] com *data frame*

```r
conjdados[1] # retorna data frame
conjdados[, 1] # retorna vetor
```

Operador [] com *data frame*

```r
conjdados[1] # retorna data frame
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```r
conjdados[, 1] # retorna vetor
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

Operador [] com *data frame*

```r
conjdados[1] # retorna um data frame
conjdados["Nome"] # a mesma coisa
```

Operador [] com *data frame*

```r
conjdados[1] # retorna um data frame
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```r
conjdados["Nome"] # a mesma coisa
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

Operador [] com *data frame*

Opção drop

A opção `drop = F` funciona com *data frames* também.

Apesar de existe uma opção `drop = T`, ela não exerce o efeito esperado e apenas faz um mensagem de aviso ser emitida.

Operador [] com *data frame*

Opção drop

```r
conjdados[, "Nome"]
conjdados[, "Nome", drop = F] # funciona como esperado
```

Operador [] com *data frame*

Opção drop

```r
conjdados[, "Nome"]
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

```r
conjdados[, "Nome", drop = F] # funciona como esperado
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

Operador [] com *data frame*

Opção drop

```r
conjdados["Nome"]
conjdados["Nome", drop = T]  # NÃO funciona como esperado
```

Operador [] com *data frame*

Opção drop

```
conjdados["Nome"]
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

```
conjdados["Nome", drop = T]  # NÃO funciona como esperado
```

```
## Warning in `[.data.frame`(conjdados, "Nome", drop = T): 'drop' argument
## will be ignored
```

```
##   Nome
## 1  Ana
## 2  Bia
## 3 Ciro
```

Operador [] com *data frame*

```
conjdados[, 1] # retorna um vetor de fatores
conjdados[, "Nome"] # a mesma coisa
```

Operador [] com *data frame*

```
conjdados[, 1] # retorna um vetor de fatores
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

```
conjdados[, "Nome"] # a mesma coisa
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

Operador [[]]

O operador `[[]]` pode ser usado com

- listas
- *data frames*

Operador [[]] com lista

```
lista <- list(x = 1:5, y = 2:10, z = 3:9)
lista
lista[[1]] # retorna um elemento
```

Operador [[]] com lista

```
lista <- list(x = 1:5, y = 2:10, z = 3:9)
lista
```

```
## $x
## [1] 1 2 3 4 5
##
## $y
## [1]  2  3  4  5  6  7  8  9 10
##
## $z
## [1] 3 4 5 6 7 8 9
```

```
lista[[1]] # retorna um elemento
```

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

Operador [[]] com lista

```
lista <- list(x = list(1,2), y = 2:10, z = 3:9)
lista
```

Operador [[]] com lista

```
lista <- list(x = list(1,2), y = 2:10, z = 3:9)
lista
```

```
## $x
## $x[[1]]
## [1] 1
##
## $x[[2]]
## [1] 2
##
##
## $y
## [1]  2  3  4  5  6  7  8  9 10
##
## $z
## [1] 3 4 5 6 7 8 9
```

Operador [[]] com lista

```
lista[[1]] # retorna um elemento (que é uma lista)
lista[[c(1,2)]] # retorna o elemento da lista retornada acima
```

Operador [[]] com lista

```
lista[[1]] # retorna um elemento (que é uma lista)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
```

```
lista[[c(1,2)]] # retorna o elemento da lista retornada acima
```

```
## [1] 2
```

Operador [[]] com lista

Compare

```
lista[c(1,2)]
lista[[c(1,2)]]
```

Operador [[]] com lista

```
lista[c(1,2)]
```

```
## $x
## $x[[1]]
## [1] 1
##
## $x[[2]]
## [1] 2
##
##
## $y
## [1]  2  3  4  5  6  7  8  9 10
```

```
lista[[c(1,2)]]
```

```
## [1] 2
```

Operador [[]] com *data frame*

```
conjdados[[1]]
conjdados[["Nome"]]
```

Operador [[]] com *data frame*

```
conjdados[[1]]
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

```
conjdados[["Nome"]]
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

Operador $

Funciona tanto com lista como com *data frame*.

```
lista$y
conjdados$Nome
```

Operador $

Funciona tanto com lista como com *data frame*.

```
lista$y
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

```
conjdados$Nome
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

Operador $

Compare

```
lista[["y"]]
lista$y
```

Operador $

Compare

```
lista[["y"]]
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

```
lista$y
```

```
## [1]  2  3  4  5  6  7  8  9 10
```

Operador $

Compare

```
conjdados[["Nome"]]
conjdados$Nome
```

Operador $

Compare

```
conjdados[["Nome"]]
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

```
conjdados$Nome
```

```
## [1] Ana  Bia  Ciro
## Levels: Ana Bia Ciro
```

Indexando com vetores numéricos negativos

Ao utilizar o operador `[]` com vetores de inteiros negativos, obtemos como resultado a estrutura de dados **sem** as posições informadas.

Indexando com vetores numéricos negativos

```
idades <- c(20, 23, 18, 39, 32, 27, 19, 35)
idades
idades[-c(3, 2, 5, 7)] # sem os 2o, 3o, 5o e 7o elementos
idades[-1:-5] # sem os elementos 1o a 5o
```

Indexando com vetores numéricos negativos

```r
idades <- c(20, 23, 18, 39, 32, 27, 19, 35)
idades
```

```
## [1] 20 23 18 39 32 27 19 35
```

```r
idades[-c(3, 2, 5, 7)] # sem os 2o, 3o, 5o e 7o elementos
```

```
## [1] 20 39 27 35
```

```r
idades[-1:-5] # sem os elementos 1o a 5o
```

```
## [1] 27 19 35
```

Indexando com vetores lógicos

Ao utilizar o operador `[]` com vetores de lógicos, obtemos como resultado a estrutura de dados **com** as posições correspondentes a `TRUE` e **sem** as posições correspondentes a `FALSE` no vetor lógico.

O vetor usado como índice **precisa** necessariamente ter as mesmas dimensões que a estrutura de dados.

Indexando com vetores lógicos

```r
idades[c(T, T, F, F, T, F, F, T)] # 8 elementos
conjdados[c(T, F, T), "Nome"] # 3 linhas
conjdados[c(T, F, T), c(F, T)] # 3 linhas e 2 colunas
```

Indexando com vetores lógicos

```r
idades[c(T, T, F, F, T, F, F, T)] # 8 elementos
```

```
## [1] 20 23 32 35
```

```r
conjdados[c(T, F, T), "Nome"] # 3 linhas
```

```
## [1] Ana  Ciro
## Levels: Ana Bia Ciro
```

```r
conjdados[c(T, F, T), c(F, T)] # 3 linhas e 2 colunas
```

```
## [1] 20 27
```

Indexando com vetores lógicos

Se o vetor índice não tiver as mesmas dimensões da estrutura de dados, poderão ou não ocorrer erros.

```r
idades[c(T, T, F, F)] # idade tem 8 elementos
idades[c(T, T, F, F, T, T, F, F, T, T, F, F)] # idade tem 8 elementos
```

Indexando com vetores lógicos

```r
idades[c(T, T, F, F)] # idade tem 8 elementos
```

```
## [1] 20 23 32 27
```

```r
idades[c(T, T, F, F, T, T, F, F, T, T, F, F)] # idade tem 8 elementos
```

```
## [1] 20 23 32 27 NA NA
```

No primeiro caso, o vetor índice foi "reciclado": se `indice` for igual ao vetor `c(T, T, F, F)`, então `idades[indice]` virará `idades[c(indice,indice)]`.

No segundo caso, o vetor índice avançou para além do fim da estrutura de dados, resultando nos `NA`s observados.

Operadores e Funções Matemáticas

Operadores Matemáticos

1. `+` – adição
2. `-` – subtração
3. `*` – produto escalar
4. `/` – divisão real
5. `^` ou `**` – potência
6. `%%` – resto de divisão
7. `%/%` – divisão inteira
8. `%*%` – produto interno
9. `%o%` – produto externo

Operações Matemáticas

No R, todas as operações foram pensadas para serem realizadas com vetores

```r
2 + 2
```

```
## [1] 4
```

```r
c(1,2,3,4) * c(10, 20, 30, 40)
```

```
## [1]  10  40  90 160
```

```
(1:5) ^ 2
```

```
## [1]  1  4  9 16 25
```

Um parênteses: Operadores Binários em Geral

O R pode ser bastante complacente com as operações entre vetores.

Quando as dimensões dos dois vetores não coincidem, o R "recicla" o menor vetor quantas vezes for necessário para dar conta do vetor maior.

```
c(1, 2, 3, 4, 5, 6) + c(2, 3, 4)
```

```
## [1]  3  5  7  6  8 10
```

```
c(1, 2, 3, 4, 5, 6, 7, 8, 9) + c(2, 3, 4)
```

```
## [1]  3  5  7  6  8 10  9 11 13
```

Um parênteses: Operadores Binários em Geral

Contudo, se o comprimento do maior vetor não for um múltiplo do comprimento do menor vetor, o R realiza a operação do mesmo jeito, mas sinaliza a discrepância através de uma mensagem de aviso.

```
c(1, 2, 3, 4, 5) + c(2, 3, 4)
```

```
## Warning in c(1, 2, 3, 4, 5) + c(2, 3, 4): comprimento do objeto maior não é
## múltiplo do comprimento do objeto menor
```

```
## [1] 3 5 7 6 8
```

Operações Matemáticas

Note a importância dos parênteses

```
(1:5) ^ 2
```

```
## [1]  1  4  9 16 25
```

```
1:5 ^ 2
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
## [24] 24 25
```

Na segunda expressão, a operação `^` é realizada antes da operação `:`.

Operações Matemáticas

Note a diferença entre `/`, `%/%` e `%%`

```r
2:10 / 2
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```r
2:10 %/% 2
```

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

```r
2:10 %% 2
```

```
## [1] 0 1 0 1 0 1 0 1 0
```

Operações Matemáticas

Note a diferença entre os três operadores de multiplicação `*`, `%*%` e `%o%`

```r
1:3 * 1:3
```

```
## [1] 1 4 9
```

```r
1:3 %*% 1:3
```

```
##      [,1]
## [1,]   14
```

```r
1:3 %o% 1:3
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    2    4    6
## [3,]    3    6    9
```

Funções Matemáticas

Boa parte das funções matemáticas básicas estão disponíveis no R e são vetorizadas.

1. Função exponencial `exp()`
2. Funções logaritmicas `log()`, `log10()` e `log2()`
3. Funções gerais `abs()`, `sign()` e `sqrt()`
4. Funções de arredondamento/truncamento: `floor()`, `ceiling()`, `trunc()`, `round()`, `signif()`
5. Outras Funções
    1. Funções trigonométricas: `cos`, `sin`, `tan`, `acos`, `asin`, `atan`
    2. Funções hiperbólicas: `cosh`, `sinh`, `tanh`, `acosh`, `asinh`, `atanh`

Funções Matemáticas

```r
exp(1:3)
```

```
## [1]  2.718282  7.389056 20.085537
```

```r
log(1:3)
```

```
## [1] 0.0000000 0.6931472 1.0986123
```

```r
abs(-2:2)
```

```
## [1] 2 1 0 1 2
```

```r
sign(-2:2)
```

```
## [1] -1 -1  0  1  1
```

Funções Matemáticas

```r
floor((-10:10)/10)
```

```
##  [1] -1 -1 -1 -1 -1 -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  1
```

```r
ceiling((-10:10)/10)
```

```
##  [1] -1  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1  1  1  1  1  1
```

```r
trunc((-10:10)/10)
```

```
##  [1] -1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1
```

```r
round((-10:10)/10)
```

```
##  [1] -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  0  0  0  1  1  1  1  1
```

Funções Matemáticas

```r
(1:4)/3
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333
```

```r
signif((1:4)/3, 1)
```

```
## [1] 0.3 0.7 1.0 1.0
```

```r
signif((1:4)/3, 2)
```

```
## [1] 0.33 0.67 1.00 1.30
```

```r
signif((1:4)/3, 3) # digitos significativos != casas decimais
```

```
## [1] 0.333 0.667 1.000 1.330
```

Funções Matemáticas

```r
(1:4)/3
```

```
## [1] 0.3333333 0.6666667 1.0000000 1.3333333
```

```r
round((1:4)/3, 1)
```

```
## [1] 0.3 0.7 1.0 1.3
```

```r
round((1:4)/3, 2)
```

```
## [1] 0.33 0.67 1.00 1.33
```

```r
round((1:4)/3, 3)
```

```
## [1] 0.333 0.667 1.000 1.333
```

Operações e Funções Matemáticas

Na verdade, cada operador é um atalho conveniente para uma função

```r
2 + 2
```

```
## [1] 4
```

```r
sum(2, 2)
```

```
## [1] 4
```

Operações e Funções Matemáticas

Na verdade, cada operador é um atalho conveniente para uma função

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

Operações e Funções Matemáticas

Note que o ‘ (backtick, aspa simples reversa, ou tique) serve para fazer referência ao símbolo `+`. Sem ele, o R daria erro por falta dos valores a serem somados.

```
`+`
```

```
## function (e1, e2)  .Primitive("+")
```

```
+
```

```
## Error: <text>:2:0: unexpected end of input
## 1: +
##    ^
```

Operações e Funções de Comparação

Operações e Funções de Comparação

1. `<`
2. `<=`
3. `>`
4. `>=`
5. `==`
6. `!=`
7. `%in%`

Operações e Funções de Comparação

```
1:10 < 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
1:10 <= 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```r
1:10 > 5
```

```
##  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
1:10 >= 5
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

Operações e Funções de Comparação

```r
1:10 == 5
```

```
##  [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
```

```r
1:10 != 5
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
1:10 %in% c(2, 3, 5, 7, 9, 11, 13)
```

```
##  [1] FALSE  TRUE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

Operações e Funções Lógicas

Operações e Funções Lógicas

1. `!`
2. `&`
3. `&&`
4. `|`
5. `||`
6. `xor()`
7. `any()`
8. `all()`

Operações e Funções Lógicas

```r
!c(T, F)
```

```
## [1] FALSE  TRUE
```

```r
c(T, T, F, F) & c(T, F, T, F)
```

```
## [1]  TRUE FALSE FALSE FALSE
```

```r
c(T, T, F, F) | c(T, F, T, F)
```

```
## [1]  TRUE  TRUE  TRUE FALSE
```

```r
xor(c(T, T, F, F), c(T, F, T, F))
```

```
## [1] FALSE  TRUE  TRUE FALSE
```

Operações e Funções Lógicas

```r
any(c(T, F))
```

```
## [1] TRUE
```

```r
all(c(T, F))
```

```
## [1] FALSE
```

Ordem de Precedência

Ordem decrescente de precedência entre operadores

1) `$`	8) `+` e `-` binários
2) `[]` e `[[]]`	9) `<`, `>`, `<=`, `>=`, `==` e `!=`
3) `^`	10) `!`
4) `-` e `+` unários	11) `&` e `&&`
5) `:`	12) `|` e `||`
6) `%%` e `%/%`	13) `->`
7) `*` e `/`	14) `<-`

Exemplos de aplicação

```r
idades
```

```
## [1] 20 23 18 39 32 27 19 35
```

```r
idades < 30
```

```
## [1]  TRUE  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

```r
idades[idades < 30]
```

```
## [1] 20 23 18 27 19
```

Exemplos de aplicação

```
conjdados[conjdados$Idade < 25, c("Nome", "Sexo")]
```

```
##   Nome Sexo
## 1  Ana    f
## 2  Bia    f
```

Instruções de Repetição

Instruções de Repetição (*Loop*)

1. A instrução `for`
    1. `1:N`,
    2. `names()`
    3. `seq()`
2. *As instruções* ***`while`****,* ***`repeat`****,* ***`break`*** *e* ***`next`***

Instrução `for`

A instrução `for` serve para repetir um trecho de código quantas vezes for necessário.

A sintaxe da instrução `for` é

```
for (variável in sequência) expressão
```

Se houver mais de a expressão, use `{}`

```
for (variável in sequência) {expressão; ...; expressão}
```

Instrução `for`

Quebrar a instrução em mais de uma linha geralmente aumenta a legibilidade do seu código.

```
for (variável in sequência)
    expressão

for (variável in sequência) {
    expressão
    ...
    expressão
}
```

Instrução `for`

É possível executar este código ...

```
for (i in 1:3) print(idades[i])
```

```
## [1] 20
## [1] 23
## [1] 18
```

Instrução `for`

... ao invés deste

```
print(idades[1])
```

```
## [1] 20
```

```
print(idades[2])
```

```
## [1] 23
```

```
print(idades[3])
```

```
## [1] 18
```

Instrução `for`

Claro, esse foi um exemplo bobinho, pois poderíamos ter executado

```
print(idades[1:3])
```

```
## [1] 20 23 18
```

Exemplo de Aplicação

Mas este exemplo não é tão bobinho

```
for (nome in names(conjdados))
    print(class(conjdados[[nome]]))
```

```
## [1] "factor"
## [1] "numeric"
## [1] "factor"
```

Exemplo de Aplicação

Isso é curioso, pois `conjdados` foi definido como

```
conjdados <-
    data.frame(
        nome = c("Ana", "Bia", "Ciro"),
        idade = c(20, 22, 27),
        sexo = factor(c("f", "f", "m"))
    )
```

e, depois, mudamos os nomes da variáveis para

```
names(conjdados) <- c("Nome", "Idade", "Sexo")
```

mas não mudamos o tipo de `Nome` de textual (`character`) para categórico (`factor`).

Exemplo de Aplicação

Por que a primeira variável é categórica ao invés de texto?

Por causa da sintaxe de `data.frame`

```
data.frame # para ver o corpo da função
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = default.stringsAsFactors())
## {
##     data.row.names <- if (check.rows && is.null(row.names))
##         function(current, new, i) {
##             if (is.character(current))
##                 new <- as.character(new)
##             if (is.character(new))
##                 current <- as.character(current)
##             if (anyDuplicated(new))
##                 return(current)
##             if (is.null(current))
##                 return(new)
##             if (all(current == new) || all(current == ""))
##                 return(new)
##             stop(gettextf("mismatch of row names in arguments of 'data.frame', item %d",
##                 i), domain = NA)
##         }
##     else function(current, new, i) {
##         if (is.null(current)) {
##             if (anyDuplicated(new)) {
##                 warning(gettextf("some row.names duplicated: %s --> row.names NOT used",
##                   paste(which(duplicated(new)), collapse = ",")),
##                   domain = NA)
##                 current
##             }
##             else new
##         }
##         else current
##     }
##     object <- as.list(substitute(list(...)))[-1L]
```

```
##     mirn <- missing(row.names)
##     mrn <- is.null(row.names)
##     x <- list(...)
##     n <- length(x)
##     if (n < 1L) {
##         if (!mrn) {
##             if (is.object(row.names) || !is.integer(row.names))
##                 row.names <- as.character(row.names)
##             if (anyNA(row.names))
##                 stop("row names contain missing values")
##             if (anyDuplicated(row.names))
##                 stop(gettextf("duplicate row.names: %s", paste(unique(row.names[duplicated(row.names)
##                   collapse = ", ")), domain = NA)
##         }
##         else row.names <- integer()
##         return(structure(list(), names = character(), row.names = row.names,
##             class = "data.frame"))
##     }
##     vnames <- names(x)
##     if (length(vnames) != n)
##         vnames <- character(n)
##     no.vn <- !nzchar(vnames)
##     vlist <- vnames <- as.list(vnames)
##     nrows <- ncols <- integer(n)
##     for (i in seq_len(n)) {
##         xi <- if (is.character(x[[i]]) || is.list(x[[i]]))
##             as.data.frame(x[[i]], optional = TRUE, stringsAsFactors = stringsAsFactors)
##         else as.data.frame(x[[i]], optional = TRUE)
##         nrows[i] <- .row_names_info(xi)
##         ncols[i] <- length(xi)
##         namesi <- names(xi)
##         if (ncols[i] > 1L) {
##             if (length(namesi) == 0L)
##                 namesi <- seq_len(ncols[i])
##             vnames[[i]] <- if (no.vn[i])
##                 namesi
##             else paste(vnames[[i]], namesi, sep = ".")
##         }
##         else if (length(namesi)) {
##             vnames[[i]] <- namesi
##         }
##         else if (fix.empty.names && no.vn[[i]]) {
##             tmpname <- deparse(object[[i]], nlines = 1L)[1L]
##             if (substr(tmpname, 1L, 2L) == "I(") {
##                 ntmpn <- nchar(tmpname, "c")
##                 if (substr(tmpname, ntmpn, ntmpn) == ")")
##                   tmpname <- substr(tmpname, 3L, ntmpn - 1L)
##             }
##             vnames[[i]] <- tmpname
##         }
##         if (mirn && nrows[i] > 0L) {
##             rowsi <- attr(xi, "row.names")
##             if (any(nzchar(rowsi)))
##                 row.names <- data.row.names(row.names, rowsi,
```

```
##                     i)
##         }
##         nrows[i] <- abs(nrows[i])
##         vlist[[i]] <- xi
##     }
##     nr <- max(nrows)
##     for (i in seq_len(n)[nrows < nr]) {
##         xi <- vlist[[i]]
##         if (nrows[i] > 0L && (nr%%nrows[i] == 0L)) {
##             xi <- unclass(xi)
##             fixed <- TRUE
##             for (j in seq_along(xi)) {
##                 xi1 <- xi[[j]]
##                 if (is.vector(xi1) || is.factor(xi1))
##                   xi[[j]] <- rep(xi1, length.out = nr)
##                 else if (is.character(xi1) && inherits(xi1, "AsIs"))
##                   xi[[j]] <- structure(rep(xi1, length.out = nr),
##                     class = class(xi1))
##                 else if (inherits(xi1, "Date") || inherits(xi1,
##                   "POSIXct"))
##                   xi[[j]] <- rep(xi1, length.out = nr)
##                 else {
##                   fixed <- FALSE
##                   break
##                 }
##             }
##             if (fixed) {
##                 vlist[[i]] <- xi
##                 next
##             }
##         }
##         stop(gettextf("arguments imply differing number of rows: %s",
##             paste(unique(nrows), collapse = ", ")), domain = NA)
##     }
##     value <- unlist(vlist, recursive = FALSE, use.names = FALSE)
##     vnames <- unlist(vnames[ncols > 0L])
##     if (fix.empty.names && any(noname <- !nzchar(vnames)))
##         vnames[noname] <- paste("Var", seq_along(vnames), sep = ".")[noname]
##     if (check.names) {
##         if (fix.empty.names)
##             vnames <- make.names(vnames, unique = TRUE)
##         else {
##             nz <- nzchar(vnames)
##             vnames[nz] <- make.names(vnames[nz], unique = TRUE)
##         }
##     }
##     names(value) <- vnames
##     if (!mrn) {
##         if (length(row.names) == 1L && nr != 1L) {
##             if (is.character(row.names))
##                 row.names <- match(row.names, vnames, 0L)
##             if (length(row.names) != 1L || row.names < 1L ||
##                 row.names > length(vnames))
##                 stop("'row.names' should specify one of the variables")
```

```
##             i <- row.names
##             row.names <- value[[i]]
##             value <- value[-i]
##         }
##         else if (!is.null(row.names) && length(row.names) !=
##             nr)
##             stop("row names supplied are of the wrong length")
##     }
##     else if (!is.null(row.names) && length(row.names) != nr) {
##         warning("row names were found from a short variable and have been discarded")
##         row.names <- NULL
##     }
##     if (is.null(row.names))
##         row.names <- .set_row_names(nr)
##     else {
##         if (is.object(row.names) || !is.integer(row.names))
##             row.names <- as.character(row.names)
##         if (anyNA(row.names))
##             stop("row names contain missing values")
##         if (anyDuplicated(row.names))
##             stop(gettextf("duplicate row.names: %s", paste(unique(row.names[duplicated(row.names)]),
##                 collapse = ", ")), domain = NA)
##     }
##     attr(value, "row.names") <- row.names
##     attr(value, "class") <- "data.frame"
##     value
## }
## <bytecode: 0x25675c8>
## <environment: namespace:base>
```

Exemplo de Aplicação

Tem muito...

```
args(data.frame) # para ver apenas os argumentos da função
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = default.stringsAsFactors())
## NULL
```

Agora sim, podemos ver que `data.frame` tem vários parâmetros e que `stringsAsFactors` tem por valor padrão o resultado da função `default.stringsAsFactors()`, que é

```
default.stringsAsFactors()
```

```
## [1] TRUE
```

Exemplo de Aplicação

Ou seja, `data.frame()` automaticamente converte vetores de texto em fatores.

Para evitar esse comportamento padrão, `conjdados` deveria ter sido definido com

```r
conjdados <-
    data.frame(
        nome = c("Ana", "Bia", "Ciro"),
        idade = c(20, 22, 27),
        sexo = factor(c("f", "f", "m")),
        stringsAsFactors = FALSE
    )
```

Exemplo de Aplicação

Agora sim

```r
for (nome in names(conjdados))
    print(class(conjdados[[nome]]))
```

```
## [1] "character"
## [1] "numeric"
## [1] "factor"
```

Exemplo de Aplicação

Por fim, note utilizamos `conjdados[[nome]]` dentro de `class()` ao invés de `conjdados[nome]`.

`conjdados[nome]` não teria retornado vetores, mas sim a mesma estrutura de `conjdados`.

```r
for (nome in names(conjdados))
    print(class(conjdados[nome]))
```

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

Veremos mais sobre isso adiante no curso.

Instrução `for`

Na verdade, podemos usar todo tipo de estrutura (vetor, lista, matriz, *array* ou *data frame*) dentro de um `for` ou qualquer função que retorne uma estrutura.

Por exemplo, aqui

```r
for (variavel in conjdados)
    print(class(variavel))
```

```
## [1] "character"
## [1] "numeric"
## [1] "factor"
```

literalmente os vetores que compõem as colunas de `conjdados` são os valores que `variavel` assume em cada iteração do *loop*.

Instrução `for`

Interessante para usar com `for`:

1. `seq()`
2. `names()`

Instrução `for`

Função `seq()`

1. `seq(num1, num2)`
   - o mesmo que `num1:num2`
2. `seq(num1, num2, num3)`
   - é tipo `num1:num2`, mas pulando de `num3` em `num3`
3. `seq(num1, num2, length.out = num3)`
   - é tipo `num1:num2`, mas com `num3` termos
4. `seq(estrutura)`
   - o mesmo que `1:length(estrutura)`

Instrução `for`

Função `seq()`

```
seq(10, 20)
```

```
##  [1] 10 11 12 13 14 15 16 17 18 19 20
```

```
seq(10, 20, 2)
```

```
## [1] 10 12 14 16 18 20
```

```
seq(10, 20, length.out = 3)
```

```
## [1] 10 15 20
```

Instrução `for`

Função `seq()`

```
idades
```

```
## [1] 20 23 18 39 32 27 19 35
```

```r
seq(idades)
```

```
## [1] 1 2 3 4 5 6 7 8
```

Instrução `for`

Função `seq()`

Note a diferença

```r
seq(1, 20, 3) # os termos não ultrapassam 20
```

```
## [1]  1  4  7 10 13 16 19
```

```r
seq(1, 20, length.out = 3) # ajusta o salto para dar 3 termos
```

```
## [1]  1.0 10.5 20.0
```

Outras Instruções de *Loop*

Existem outras instruções referentes a *loop* em R, cumpre falar sobre elas, mas **eu** desaconselho vocês as usarem até terem mais experiência com R.

1. `while`
2. `repeat`
3. `break`
4. `next`

Outras Instruções de *Loop*

1. `while`
   - `while`(*condição*) *expressão*
     - repete *expressão* enquanto *condição* for verdadeira
2. `repeat`
   - `repeat` *expressão*
     - repete *expressão* para todo o sempre
       - a menos que um `break` seja executado
3. `break`
   - interrompe um *loop* e vai para a próxima expressão
4. `next`
   - vai para a próxima iteração

Outras Instruções de *Loop*

Um exemplo de *loop* com `while`

```
resposta <- readline(prompt = "Qual é a sua resposta?")
while (resposta!=42)
{
  print("A resposta tem que ser 42");
  response <- readline(prompt = "Qual é a sua resposta?")
}
```

Outras Instruções de *Loop*

O mesmo *loop* com `repeat` e `break`

```
repeat
{
  response <- readline(prompt = "Qual é a sua resposta?")
  if (resposta == 42)
      break
  print("A resposta tem que ser 42");
}
```

Outras Instruções de *Loop*

Um exemplo com `next`

```
for (k in 1:10)
{
    if (k %% 2)
        next
    print(k)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

Instruções de Execução Condicional

Instruções de Execução Condicional

1. As instruções `if` e `else`
2. A função `ifelse()`
3. A função `switch()`

AQUI!!! Um slide para cada um desses