Programação em R

Julho, 2016

Vimos bastante coisa trabalhando apenas com vetores: como selecionar e modificar elementos por nome, posição e vetor lógico, coerção de classes, vetorização, reciclagem... e essas lições servem para praticamente todos os demais objetos do R! O vetor é o objeto básico do qual os demais objetos são constituídos. Nesta seção veremos estes outros objetos, como matrizes (e arrays), data.frames e listas. E tudo o que vimos anteriormente naturalmente se estende para esses objetos, com pequenas modificações.

Matrizes: vetores com dimensões

Criando matrizes: função `matrix()`

É possível criar matrizes no R com a função `matrix()`.

```
matrix(data = dados, ncol = numero_de_colunas, nrow = numero_de_linhas)
```

- No argumento `data` passamos o vetor que desejamos transformar em matriz;
- `ncol` especifica o número de colunas da matriz; e,
- `nrow` especifica o número de linhas da matriz.

Criando matrizes: função `matrix()`

Criemos nossa primeira matriz, com 10 elementos, sendo cinco linhas e duas colunas:

```
minha_matriz <- matrix(data = 1:10, ncol = 2, nrow = 5)
minha_matriz
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

Perceba que os elementos foram preenchidos por coluna. Isto é, os números de 1 a 5 ficaram na primeira coluna e os números de 6 a 10 na segunda.

Criando matrizes: função `matrix()`

Para que a função preencha a matriz por linhas, basta colocar como argumento `byrow = TRUE`:

```
minha_matriz_2 <- matrix(data = 1:10, ncol = 2, nrow = 5, byrow = TRUE)
minha_matriz_2
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
## [4,]    7    8
## [5,]    9   10
```

A matriz agora foi preenchida por linhas: temos os números 1 e 2 na primeira linha, 3 e 4 na segunda linha e assim por diante.

Omitindo ncol ou nrow

Você não precisa especificar os dois termos `ncol` e `nrow`, basta especificar um deles que o outro é automaticamente inferido pelo R.

```
matrix(data = 1:10, ncol = 5)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10

matrix(data = 1:10, nrow = 2)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Estrutura e atributos de uma matriz

A função `str()` fornece algumas informações básicas sobre a estrutura da matriz:

```
str(minha_matriz)
##  int [1:5, 1:2] 1 2 3 4 5 6 7 8 9 10
```

Na descrição `int [1:5, 1:2]` vemos que o objeto `minha_matriz` é composto de inteiros (`int`) e tem duas dimensões, mais especificamente cinco linhas (`[1:5,...]`) e duas colunas (`[...,1:2]`).

Estrutura e atributos de uma matriz

```
# número de linhas da matriz
nrow(minha_matriz)
## [1] 5

# número de colunas da matriz
ncol(minha_matriz)
## [1] 2

# dimensões da matriz (# linhas, # colunas)
dim(minha_matriz)
## [1] 5 2

# tamanho da matriz (quantos elementos no total)
length(minha_matriz)
## [1] 10
```

Nomes das linhas e colunas

As linhas e colunas de uma matriz podem ser nomeadas. Para tanto você pode utilizar `rownames()` e `colnames()`:

```
rownames(minha_matriz) <- letters[1:5]
rownames(minha_matriz)
## [1] "a" "b" "c" "d" "e"

colnames(minha_matriz) <- LETTERS[1:2]
colnames(minha_matriz)
## [1] "A" "B"

str(minha_matriz)
##  int [1:5, 1:2] 1 2 3 4 5 6 7 8 9 10
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:5] "a" "b" "c" "d" ...
##   ..$ : chr [1:2] "A" "B"
```

Matrizes são vetores com um atributo a mais: dimensão

Outra forma bem simples de criar uma matriz é atribuir dimensões a um vetor. Por exemplo, podemos criar um objeto idêntico à `minha_matriz` da seguinte forma:

```
# cria vetor com números de 1 a 10
m <- 1:10

# atribui dimensões ao vetor (5 linhas e 2 colunas)
dim(m) <- c(5, 2)
m
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

Operações com matrizes

Operador	Descrição
`== != > >= < <=`	Operadores relacionais elemento a elemento
`+ - / *`	Soma, subtração, divisão e multiplicação elemento a elemento
`%*%`	Multiplicação de matriz
`%x%`	Produto de Kronecker
`t()`	Transposição de matriz
`solve()`	Inversão de matriz (ver também `qr()`, `svd()`, `chol()`)
`det()`	Determinante de uma matriz
`diag()`	Diagonal de uma matriz

Operações elemento a elemento: soma, multiplicação e divisão

Vamos criar uma matriz mais simples para testar essas operações:

```
z <- 1:4
dim(z) <- c(2, 2)
str(z)
##  int [1:2, 1:2] 1 2 3 4
```

Primeiramente a soma (elmento a elemento):

```
# soma elemento a elemento
z + z
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

Operações elemento a elemento: soma, multiplicação e divisão

A soma não pecisa ser de matriz com outra matriz. É possível somar uma matriz com um vetor. Neste caso, a soma é feita seguindo a ordem por coluna:

```
z + c(1, 2, 3, 4)
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

Operações elemento a elemento: soma, multiplicação e divisão

Quando os elementos não são do mesmo tamanho, o R tenta fazer reciclagem (a mesma que vimos na seção de vetores).

```
z + 10
##      [,1] [,2]
## [1,]   11   13
## [2,]   12   14
```

Repare que o R somou cada elemento de z com o número 10. E se somassemos um vetor com dois elementos?

```
z + c(10, 20)
##      [,1] [,2]
## [1,]   11   13
## [2,]   22   24
```

Note que o R não reclamou da operação! Ele reciclou o vetor `c(10, 20)` para as duas colunas de z.

Operações elemento a elemento: soma, multiplicação e divisão

E se o vetor somado fosse de três elementos?

```
z + c(10, 20, 30)
## Warning in z + c(10, 20, 30): comprimento do objeto maior não é múltiplo do
## comprimento do objeto menor
##      [,1] [,2]
## [1,]   11   33
## [2,]   22   14
```

O R faz a operação mas te dá um aviso, pois o tamanho do vetor não é um múltiplo do tamanho da matriz. A reciclagem é uma característica bastante útil do R, mas é preciso tomar cuidado.

Operações elemento a elemento: soma, multiplicação e divisão

O mesmo que falamos para soma vale para as demais operações elemento a elemento:

```
z_mais_z <- z + z
z_mais_z
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8


dois_z <- 2*z
z_mais_z == dois_z
##      [,1] [,2]
## [1,] TRUE TRUE
## [2,] TRUE TRUE
```

Operações matriciais

Multiplicação matricial:

```
zz <- z %*% z
zz
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Transposição:

```
tz <- t(zz)
tz
##      [,1] [,2]
## [1,]    7   10
## [2,]   15   22
```

Operações matriciais

Inversão:

```
inv_zz <- solve(zz)
inv_zz
##       [,1] [,2]
## [1,]  5.5 -3.7
## [2,] -2.5  1.7
```

Determinante:

```
det(zz)
## [1] 4
```

Operações matriciais

Produto de kronecker:

```
z %x% z
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    3    9
## [2,]    2    4    6   12
## [3,]    2    6    4   12
## [4,]    4    8    8   16
```

Diagonal:

```
diag(z)
## [1] 1 4
```

Exemplo: regressão linear

Vamos simular e estimar os parâmetros de uma regressão linear usando operações matriciais:

```
# parametros
n_vars <- 4 # numero de variaveis
n_obs <- 100 # numero de observações

# Simulando y ~ Xb + e
X <- matrix(rnorm(n_obs*n_vars), ncol = n_vars) # vetor X
betas <- c(1, 2, 3, 4) # betas verdadeiros
erro <- rnorm(n_obs) # termo de erro normal(0,1)
y <- X %*% betas + erro # vetor y
```

Exemplo: regressão linear

Estimando por mínimos quadrados:

```
# estimando os betas: (X'X)^-1 X'y
betas_estimados <- solve(t(X) %*% X) %*% t(X) %*% y
betas_estimados[,1]
## [1] 0.95 1.90 2.96 4.17

# Comparando com a estimativa da função nativa do R
modelo <- lm(y ~ X - 1)
coef(modelo)
##   X1   X2   X3   X4
## 0.95 1.90 2.96 4.17
```

Nomes das linhas e colunas

Vamos criar uma matriz de exemplo mais realista:

```
# criando matriz de vendas para exemplo
# quantidade de vendedores
n_vendedores <- 3

# quantidade de dias
n_dias <- 5

# valores de venda (aleatórios)
# média de R$1000,00 por dia
# desvio padrão de R$200,00
set.seed(192)
media <- 1000
desvio <- 200
valores <- rnorm(n_dias*n_vendedores, media, desvio)
```

Nomes das linhas e colunas

```r
# cria matriz
vendas <- matrix(valores, nrow = n_dias, ncol = n_vendedores)

# nomes para linhas
rownames(vendas) <- c("segunda", "terça", "quarta", "quinta", "sexta")

# nomes para colunas
colnames(vendas) <- c("João", "Maria", "Ana")
```

Nomes das linhas e colunas

Vamos ver como ficou nossa matriz:

```
vendas
##          João Maria  Ana
## segunda 1201   927 1043
## terça   1027   920  833
## quarta  1096  1020  944
## quinta   915   952 1018
## sexta    787  1039 1293
```

Selecionando elementos de uma matriz

Tudo o que vimos para selecionar elementos de um vetor vale para matrizes. A principal diferença é que agora você vai estabelecer um índice para linha e outro para a coluna do elemento que você quer:

```
matriz[indice_de_linha, indice_de_coluna]
```

Selecionando elementos de uma matriz

Selecionando linhas (por nome ou posição):

```
vendas["segunda", ] # seleciona linha de nome "segunda"
##  João Maria   Ana
##  1201   927  1043


vendas[1, ] # seleciona primeira linha
##  João Maria   Ana
##  1201   927  1043


vendas[-c(1,2), ] # seleciona tudo exceto linhas 1 e 2
##        João Maria  Ana
## quarta 1096  1020  944
## quinta  915   952 1018
## sexta   787  1039 1293
```

Selecionando elementos de uma matriz

Selecionando colunas (por nome ou posição):

```
vendas[ ,"Ana"] # seleciona colunas de nome "Ana"
## segunda   terça  quarta  quinta   sexta
##    1043     833     944    1018    1293

vendas[ ,3] # seleciona terceira coluna
## segunda   terça  quarta  quinta   sexta
##    1043     833     944    1018    1293

vendas[ ,-3] # seleciona tudo exceto terceira coluna
##         João Maria
## segunda 1201   927
## terça   1027   920
## quarta  1096  1020
## quinta   915   952
## sexta    787  1039
```

Selecionando elementos de uma matriz

Selecionando linhas e colunas (por nome ou posição). Resultados omitidos.

```
# linhas 1, 3 e 5 - colunas João e Ana
vendas[c(1, 3, 5), c("João", "Ana")]

# linhas segunda, quarta e sexta - colunas 1 e 3
vendas[c("segunda", "quarta", "sexta"), c(1, 3)]

# tudo menos linhas 1, 3 e 5 e menos colunas 2 e 3
vendas[-c(1,3,5), -c(2,3)]
```

Cuidado: simplificação de matrizes e `drop = FALSE`

Uma coisa ao ter cuidado ao fazer subsets com matrizes é que, quando a seleção tem apenas uma linha ou uma coluna, o R simplifica o resultado para um vetor. Na maior parte das vezes isso é bastante conveniente, pois é de fato o que queremos, mas em alguns casos pode ser desejável manter a estrutura de matriz no resultado. Para tanto, basta colocar como argumento `drop = FALSE`.

```
vendas_segunda <- vendas[1, , drop = FALSE]
vendas_segunda
##          João Maria  Ana
## segunda 1201   927 1043

# resultado é uma matriz
dim(vendas_segunda)
## [1] 1 3
```

Alterando valores de uma matriz

É possível combinar a seleção de elementos com o operador <- para realizar alterações em uma matriz:

```
vendas[c("quarta", "quinta"), "Ana"]
## quarta quinta
##    944   1018

vendas[c("quarta", "quinta"), "Ana"] <- c(1500, 2000)

vendas[c("quarta", "quinta"), "Ana"]
## quarta quinta
##   1500   2000
```

Adicionando colunas com cbind()

```
set.seed(145)
José <- rnorm(nrow(vendas), media, desvio)
vendas <- cbind(vendas, José)
vendas
##          João Maria  Ana José
## segunda 1201   927 1043 1137
## terça   1027   920  833 1213
## quarta  1096  1020 1500 1107
## quinta   915   952 2000 1381
## sexta    787  1039 1293 1213
```

Adicionando linhas com rbind()

```
set.seed(90)
sábado <- rnorm(ncol(vendas), media, desvio)
vendas <- rbind(vendas, sábado)
vendas
##          João Maria  Ana José
## segunda 1201   927 1043 1137
## terça   1027   920  833 1213
## quarta  1096  1020 1500 1107
## quinta   915   952 2000 1381
## sexta    787  1039 1293 1213
## sábado  1015   970  823  856
```

Removendo linhas e colunas: basta não selecionar!

Para remover uma linha, basta sobrescrever a matriz original **não selecionando** a linha que você deseja excluir.

```
# Remove a sexta linha
vendas <- vendas[-6, ]
```

A mesma lógica para remover uma coluna:

```
# Remove a quarta coluna
vendas <- vendas[ ,-4]
```

Aplicando funções nas linhas e colunas: Summary

A função `summary()` fornece várias estatísticas descritivas por coluna:

```
summary(vendas)
##       João           Maria           Ana
##  Min.   : 787   Min.   : 920   Min.   : 833
##  1st Qu.: 915   1st Qu.: 927   1st Qu.:1043
##  Median :1027   Median : 952   Median :1293
##  Mean   :1005   Mean   : 972   Mean   :1334
##  3rd Qu.:1096   3rd Qu.:1020   3rd Qu.:1500
##  Max.   :1201   Max.   :1039   Max.   :2000
```

Aplicando funções nas linhas e colunas: Summary

Se você quise usar `summary()` nas linhas, basta transpor a matriz:

```
summary(t(vendas))
##     segunda          terça          quarta         quinta          sexta
##  Min.   : 927   Min.   : 833   Min.   :1020   Min.   : 915   Min.   : 78
##  1st Qu.: 985   1st Qu.: 877   1st Qu.:1058   1st Qu.: 934   1st Qu.: 9
##  Median :1043   Median : 920   Median :1096   Median : 952   Median :103
##  Mean   :1057   Mean   : 927   Mean   :1205   Mean   :1289   Mean   :104
##  3rd Qu.:1122   3rd Qu.: 973   3rd Qu.:1298   3rd Qu.:1476   3rd Qu.:110
##  Max.   :1201   Max.   :1027   Max.   :1500   Max.   :2000   Max.   :129
```

Aplicando funções nas linhas e colunas: rowSums e rowMeans

O R também já fornece duas funções para somas e médias por linhas:

```
# Total de vendas por dia
rowSums(vendas)
## segunda   terça  quarta  quinta   sexta
##    3171    2780    3616    3867    3119

# Média das vendas por dia
rowMeans(vendas)
## segunda   terça  quarta  quinta   sexta
##    1057     927    1205    1289    1040
```

Aplicando funções nas linhas e colunas: colSums e colMeans

E somas e médias por colunas:

```
# Total de vendas por vendedor
colSums(vendas)
##  João Maria   Ana
##  5025  4858  6670

# Média de vendas por vendedor
colMeans(vendas)
##  João Maria   Ana
##  1005   972  1334
```

Mas e se eu quiser aplicar uma função diferente de `sum()` ou `mean()`?

Aplicando funções nas linhas e colunas: `apply()`

Uma função bastante útil no R é a função `apply()`. Ela permite que você aplique uma função nas linhas ou colunas de um objeto. Sua estrutura básica é a seguinte:

```
apply(matriz, dimensão, função)
```

- no primeiro argumento passamos a matriz em que desejamos fazer as operações;
- no segundo dizemos se queremos operar por linha (1) ou por coluna (2); e,
- no terceiro argumento passamos a função que queremos aplicar.

Aplicando funções nas linhas e colunas: `apply()`

Qual foi o valor máximo vendido em cada dia (linha)?

```
apply(vendas, 1, max)
## segunda   terça  quarta  quinta   sexta
##    1201    1027    1500    2000    1293
```

Qual foi o menor valor de cada vendedor (colunas)?

```
apply(vendas, 2, min)
##  João Maria   Ana
##   787   920   833
```

Aplicando funções nas linhas e colunas: apply()

Você pode aplicar a função que desejar

```
apply(vendas, 1, sd)
## segunda   terça  quarta  quinta   sexta
##     138      97     258     616     253

apply(vendas, 2, sd)
##  João Maria   Ana
##   160    54   449
```

Mais de duas dimensões? Use o array!

Quer trabalhar com mais de duas dimensões? A solução para isso no R é o array. Vejamos um exemplo:

```
# Criando um array com 3 dimensões
# '2 tabelas' com 4 linhas 2 colunas e
meu_array <- array(1:16, dim = c(4,2,2))

# Selecionando nas 3 dimensões
meu_array[1:2, 2, 2]
```

Veremos casos concretos de uso do arrays na parte de manipulação de dados.

Matrizes, como vetores, tem que ter todos os elementos iguais!

Suponha que você queira incluir a variável sexo na nossa matriz. Note que até agora todas nossas variáveis eram numéricas... o que acontecerá se incluirmos uma variável não numérica?

```
vendas2 <- t(vendas)
vendas2
##       segunda terça quarta quinta sexta
## João     1201  1027   1096    915   787
## Maria     927   920   1020    952  1039
## Ana      1043   833   1500   2000  1293

sexo <- c("m", "f", "f")
vendas2 <- cbind(vendas2, sexo)

is.character(vendas2)
## [1] TRUE
```

Matrizes, como vetores, tem que ter todos os elementos iguais!

Todos os valores da matriz foram transformados em texto! O que ocorreu aqui é que a matriz, tal como um vetor, tem que ter todos seus elementos iguais. O que fazer, então, se quisermos montar uma base de dados com várias variáveis de classes diferentes? Neste caso a estrutura ideal é o `data.frame`, que veremos a seguir.

Exercícios

Sua vez.

- Crie o vetor `x <- 1:16`. A partir de `x`, crie uma matriz `m` com dimensões 4 x 4 utilizando a função `matrix()`. Em seguida, transforme `x` em uma matriz atribuindo a `x` as dimensões `c(4,4)`. As matrizes são idênticas?
- Selecione as linhas 1 a 4 e colunas 2 e 4 da matriz `m`. Selecione as linhas em que a coluna 1 seja menor do que 3. Selecione os elementos de `m` menores do que 10.
- Eleve ao quadrado todos os valores pares da matriz (para verificar se um número é par, verifique se o resto da divisão do número por 2 é igual a zero – para calcular o resto da divisão, use %%).

Soluções

```r
x <- 1:16

m <- matrix(x, ncol = 4)

dim(x) <- c(4, 4)

identical(x, m)

m[1:4, c(2,4)]

m[m[,1] < 3, ]

m[m < 10]

indice <- m %% 2 == 0
m[indice] <- m[indice]^2
m
```

Data Frames: seu banco de dados no R

Por que um `data.frame`?

Até agora temos utilizado apenas dados de uma mesma classe, armazenados ou em um vetor ou em uma matriz. Mas uma base de dados, em geral, é feita de dados de diversas classes diferentes: no exemplo anterior, por exemplo, podemos querer ter uma coluna com os nomes dos funcionários, outra com o sexo dos funcionários, outra com valores... como guardar essas informações?

Por que um `data.frame`?

A solução para isso é o `data.frame`. O `data.frame` é talvez o formato de dados mais importante do R. No `data.frame` cada coluna representa uma variável e cada linha uma observação. Essa é a estrutura ideal para quando você tem várias variáveis de classes diferentes em um banco de dados.

Criando um data.frame: `data.frame()`

É possível criar um `data.frame` diretamente com a função `data.frame()`:

```
funcionarios <- data.frame(nome = c("João", "Maria", "José"),
                           sexo = c("M", "F", "M"),
                           salario = c(1000, 1200, 1300),
                           stringsAsFactors = FALSE)
funcionarios
##    nome sexo salario
## 1  João    M    1000
## 2 Maria    F    1200
## 3  José    M    1300
```

Discutiremos a opção `stringsAsFactors = FALSE` mais a frente.

Criando um data.frame: `data.frame()`

Vejamos a estrutura do `data.frame`. Note que cada coluna tem sua própria classe.

```
str(funcionarios)
## 'data.frame':    3 obs. of  3 variables:
##  $ nome   : chr  "João" "Maria" "José"
##  $ sexo   : chr  "M" "F" "M"
##  $ salario: num  1000 1200 1300
```

Criando um data.frame: `data.frame()`

O que ocorreria com o `data.frame` `funcionarios` se o transformássemos em uma matriz? Vejamos:

```
as.matrix(funcionarios)
##      nome    sexo salario
## [1,] "João"  "M"  "1000"
## [2,] "Maria" "F"  "1200"
## [3,] "José"  "M"  "1300"
```

Perceba que todas as variáveis viraram `character`! Exatamente por isso precisamos de um `data.frame` neste caso.

Criando um data.frame: as.data.frame()

Também é possível criar um `data.frame` com a função `as.data.frame()`. Voltando ao exemplo das vendas:

```
df.vendas <- as.data.frame(t(vendas))
df.vendas
##       segunda terça quarta quinta sexta
## João     1201  1027   1096    915   787
## Maria     927   920   1020    952  1039
## Ana      1043   833   1500   2000  1293

str(df.vendas)
## 'data.frame':    3 obs. of  5 variables:
##  $ segunda: num  1201 927 1043
##  $ terça  : num  1027 920 833
##  $ quarta : num  1096 1020 1500
##  $ quinta : num  915 952 2000
##  $ sexta  : num  787 1039 1293
```

Manipulando `data.frames` como matrizes

Ok, temos mais um objeto do R, o data.frame ... vou ter que reaprender tudo novamente? Não!

Você pode manipular data.frames como se fossem matrizes!

Praticamente tudo o que vimos para selecionar e modificar elementos em matrizes funciona no `data.frame`.

Manipulando data.frames como matrizes

Selecionando linhas e colunas:

```
# tudo menos linha 1
funcionarios[-1, ]
##    nome sexo salario
## 2 Maria    F    1200
## 3  José    M    1300

# seleciona primeira linha e primeira coluna (vetor)
funcionarios[1, 1]
## [1] "João"

# seleciona primeira linha e primeira coluna (data.frame)
funcionarios[1, 1, drop = FALSE]
##   nome
## 1 João
```

Manipulando data.frames como matrizes

Alterando valores do data.frame como se fosse uma matriz.

```
# aumento de salario para o João
funcionarios[1, "salario"] <- 1100

funcionarios
##    nome sexo salario
## 1  João    M    1100
## 2 Maria    F    1200
## 3  José    M    1300
```

Nomes de linhas e colunas

O `data.frame` sempre terá rownames e colnames.

```
rownames(funcionarios)
## [1] "1" "2" "3"

colnames(funcionarios)
## [1] "nome"    "sexo"    "salario"
```

Detalhe: o `names` trata-se de nomes das colunas do data.frame. **Os elementos do data.frame são seus vetores coluna**.

```
names(funcionarios)
## [1] "nome"    "sexo"    "salario"
```

Extra do data.frame: selecionando e modificando com $ e [[ ]]

Outras formas alternativas de selecionar colunas em um `data.frame` são o `$` e o `[[ ]]`:

```
# Seleciona coluna nome
funcionarios$nome
## [1] "João"  "Maria" "José"

funcionarios[["nome"]]
## [1] "João"  "Maria" "José"

# Seleciona coluna salario
funcionarios$salario
## [1] 1100 1200 1300

funcionarios[["salario"]]
## [1] 1100 1200 1300
```

Tanto o `$` quanto o `[[ ]]` **sempre** retornam um vetor como resultado.

Extra do data.frame: selecionando e modificando com $ e [[ ]]

Também é possível alterar a coluna combinando $ ou [[ ]] com <-:

```
# outro aumento para o João
funcionarios$salario[1] <- 1150

# equivalente
funcionarios[["salario"]][1] <- 1150
funcionarios
##    nome sexo salario
## 1  João    M    1150
## 2 Maria    F    1200
## 3  José    M    1300
```

Extra do data.frame: retornando sempre um data.frame com [ ]

Se você quiser garantir que o resultado da seleção será sempre um data.frame use `drop = FALSE` ou selecione sem a vírgula:

```
# Retorna data.frame
funcionarios[ ,"salario", drop = FALSE]
##   salario
## 1    1150
## 2    1200
## 3    1300

# Retorna data.frame
funcionarios["salario"]
##   salario
## 1    1150
## 2    1200
## 3    1300
```

Tabela resumo: selecionando uma coluna em um data.frame

Resumindo as formas de seleção de uma coluna de um data.frame.

Operador	Descrição
`df[ ,"x"]`	Retorna **vetor** `x` do data.frame `df`.
`df$x`	Retorna **vetor** `x` do data.frame `df`.
`df[["x"]]`	Retorna **vetor** `x` do data.frame `df`.
`df[ ,"x", drop = FALSE]`	Retorna um **data.frame** com a coluna `x`.
`df["x"]`	Retorna um **data.frame** com a coluna `x`.

Criando colunas novas

Com $:

```
funcionarios$escolaridade <- c("Ensino Médio", "Graduação", "Mestrado")
```

Com [,]:

```
funcionarios[, "experiencia"] <- c(10, 12, 15)
```

Com [[ ]]:

```
funcionarios[["avaliacao_anual"]] <- c(7, 9, 10)
```

Com cbind():

```
funcionarios <- cbind(funcionarios,
                      prim_emprego = c("sim", "nao", "nao"),
                      stringsAsFactors = FALSE)
```

Criando colunas novas

Vejamos como ficou nosso data.frame com as novas colunas:

```
funcionarios
##    nome sexo salario escolaridade experiencia avaliacao_anual prim_emprego
## 1  João    M    1150 Ensino Médio          10               7          sim
## 2 Maria    F    1200    Graduação          12               9          nao
## 3  José    M    1300     Mestrado          15              10          nao
```

Removendo colunas

Atribuindo NULL:

```
# deleta coluna prim_emprego
funcionarios$prim_emprego <- NULL
```

Ou selecionando todas colunas menos as que você não quer:

```
# deleta colunas 4 e 6
funcionarios <- funcionarios[, c(-4, -6)]
```

Adicionando linhas

Uma forma simples de adicionar linhas é atribuir a nova linha com <-. Mas cuidado! O que irá acontecer com o `data.frame` com o código abaixo?

```
# CUIDADO!
funcionarios[4, ] <- c("Ana", "F", 2000,  15)
```

Note que nosso data.frame inteiro se transformou em texto! Você sabe explicar por que isso aconteceu?

```
str(funcionarios)
## 'data.frame':    4 obs. of  4 variables:
##  $ nome       : chr  "João" "Maria" "José" "Ana"
##  $ sexo       : chr  "M" "F" "M" "F"
##  $ salario    : chr  "1150" "1200" "1300" "2000"
##  $ experiencia: chr  "10" "12" "15" "15"
```

Adicionando linhas

Antes de prosseguir, transformemos as colunas `salario` e `experiencia` em números novamente:

```
funcionarios$salario <- as.numeric(funcionarios$salario)

funcionarios$experiencia <- as.numeric(funcionarios$experiencia)
```

Adicionando linhas

Se os elementos forem de classe diferente, use a função `data.frame` para evitar coerção:

```
funcionarios[4, ] <- data.frame(nome = "Ana", sexo = "F",
                                salario = 2000, experiencia = 15,
                                stringsAsFactors = FALSE)
```

Também é possível adicionar linhas com `rbind()`:

```
rbind(funcionarios,
      data.frame(nome = "Ana", sexo = "F",
                 salario = 2000,  experiencia = 15,
                 stringsAsFactors = FALSE))
```

Atenção! Não fique aumentando um `data.frame` de tamanho adicionando linhas ou colunas. Sempre que possível pré-aloque espaço!

Removendo linhas

Para remover linhas, basta selecionar apenas aquelas linhas que você deseja manter:

```
# remove linha 4 do data.frame
funcionarios <- funcionarios[-4, ]
```

```
# remove linhas em que salario <= 1150
funcionarios <- funcionarios[funcionarios$salario > 1150, ]
```

Filtrando linhas com vetores logicos

Relembrando: se passarmos um vetor lógico na dimensão das linhas, selecionamos apenas aquelas que são `TRUE`. Assim, por exemplo, se quisermos selecionar aquelas linhas em que a coluna `salario` é maior do que um determinado valor, basta colocar esta condição como filtro das linhas:

```
# Apenas linhas com salario > 1000
funcionarios[funcionarios$salario > 1000, ]
##    nome sexo salario experiencia
## 2 Maria    F    1200          12
## 3  José    M    1300          15

# Apenas linhas com sexo == "F"
funcionarios[funcionarios$sexo == "F", ]
##    nome sexo salario experiencia
## 2 Maria    F    1200          12
```

Alternativas para `rbind()` e `cbind()` para data.frames

As funções `rbind()` e `cbind()` podem não ser muito eficientes. As funções `bind_rows()` e `bind_cols()` do pacote `dplyr` (que veremos mais a frente) são alternativas interessantes. Veremos mais sobre o `dplyr` na aula de manipulações de `data.frames`.

Funções de conveniência: `subset()`

Uma função de conveniência para selecionar linhas e colunas de um `data.frame` é a função `subset()`, que tem a seguinte estrutura:

```
subset(nome_do_data_frame,
       subset = expressao_logica_para_filtrar_linhas,
       select = nomes_das_colunas,
       drop   = simplicar_para_vetor?)
```

Funções de conveniência: subset()

Vejamos alguns exemplos:

```
# funcionarios[funcionarios$sexo == "F",]
subset(funcionarios, sexo == "F")
##    nome sexo salario experiencia
## 2 Maria    F    1200          12

# funcionarios[funcionarios$sexo == "M", c("nome", "salario")]
subset(funcionarios, sexo == "M", select = c("nome", "salario"))
##   nome salario
## 3 José    1300
```

Funções de conveniência: `with`

A função `with()` permite que façamos operações com as colunas do `data.frame` sem ter que ficar repetindo o nome do `data.frame` seguido de `$` , `[ , ]` ou `[[]]` o tempo inteiro. Apenas para ilustrar:

```
# Com o with
with(funcionarios, (salario^3 - salario^2)/log(salario))
## [1] 2.4e+08 3.1e+08

# Sem o with
(funcionarios$salario^3 - funcionarios$salario^2)/log(funcionarios$salario)
## [1] 2.4e+08 3.1e+08
```

Funções de conveniência: `with`

Quatro formas de fazer a mesma coisa (pense em outras formas possíveis):

```
subset(funcionarios, sexo == "M", select = "salario", drop = TRUE)
## [1] 1300

with(funcionarios, salario[sexo == "M"])
## [1] 1300

funcionarios$salario[funcionarios$sexo == "M"]
## [1] 1300

funcionarios[funcionarios$sexo == "M", "salario"]
## [1] 1300
```

Aplicando funções no data.frame: o que funcionava nas matrizes continua valendo

As funções `rowSums()`, `rowMeans()`, `colSums()`, `colMeans()` e `apply()` continuam funcionando normalmente nos `data.frames`. Teste os seguintes códigos no `data.frame` `df.vendas`:

```
rowSums(df.vendas)
##  João Maria   Ana
##  5025  4858  6670

colSums(df.vendas)
## segunda   terça  quarta  quinta   sexta
##    3171    2780    3616    3867    3119
```

Aplicando funções no data.frame: o que funcionava nas matrizes continua valendo

```
apply(df.vendas, 1, max)
##  João Maria   Ana
##  1201  1039  2000

apply(df.vendas, 2, max)
## segunda   terça  quarta  quinta   sexta
##    1201    1027    1500    2000    1293
```

Aplicando funções no data.frame: `sapply` e `lapply`, funções nas colunas (elementos)

Outras duas funções bastante utilizadas no R são as funções `sapply()` e `lapply()`.

- As funções `sapply` e `lapply` aplicam uma função em cada elemento de um objeto.
- Como vimos, os elementos de um `data.frame` são suas colunas. Deste modo, as funções `sapply` e `lapply` aplicam uma função nas colunas de um data.frame.
- A diferença entre uma e outra é que a primeira tenta **s**implificar o resultado enquanto que a segunda sempre retorna uma lista.

Aplicando funções no data.frame: `sapply` e `lapply`, funções nas colunas (elementos)

Testando no nosso `data.frame`:

```
sapply(funcionarios[3:4], mean)
##     salario experiencia
##        1250          14

lapply(funcionarios[3:4], mean)
## $salario
## [1] 1250
##
## $experiencia
## [1] 14
```

Filtrando variáveis antes de aplicar funções: `Filter()`

Como `data.frames` podem ter variáveis de classe diferentes, muitas vezes é conveniente filtrar apenas aquelas colunas de determinada classe (ou que satisfaçam determinada condição). A função `Filter()` (note o F maiúsculo!) é uma maneira rápida de fazer isso:

```
Filter(is.numeric, funcionarios)
##   salario experiencia
## 2    1200          12
## 3    1300          15

Filter(is.character, funcionarios)
##    nome sexo
## 2 Maria    F
## 3  José    M
```

Filtrando variáveis antes de aplicar funções: `filter()`

Exemplo: aplicando a função média e máximo apenas nas colunas numéricas.

```
sapply(Filter(is.numeric, funcionarios), mean)
##     salario experiencia
##        1250          14

sapply(Filter(is.numeric, funcionarios), max)
##     salario experiencia
##        1300          15
```

Manipulando data.frames

Ainda temos muita coisa para falar de manipulação de data.frames e isso merece um espaço especial. Veremos além de outras funções base do R alguns pacotes importantes como `dplyr`, `reshape2` e `tidyr` em uma seção separada do nosso curso.

Exercícios

Sua vez.

- Crie o seguinte `data.frame`: `df <- data.frame(x = letters[1:4], y = 1:4, stringsAsFactors = FALSE)`.
- Adicione a coluna `y2` com o resultado de `y^2`. Remova a coluna `y2`.
- Adicione uma linha em que `x = "e"` e `y = 5`. Remova esta linha.
- Selecione a linha em que `x == "a"`. Selecione apenas as linhas em que `y < 3`. Modifique a coluna `y` fazendo com que os elementos em que `y >= 3` sejam iguais a `y^3`.

Soluções

```r
df <- data.frame(x = letters[1:4], y = 1:4, stringsAsFactors = FALSE)

df$y2 <- df$y^2

df$y2 <- NULL

df[5, ] <- data.frame(x = "e", y = 5, stringsAsFactors = FALSE)
df <- df[-5, ]

df[df$x == "a", ]

df[df$y < 3, ]

df$y[df$y >= 3] <- df$y[df$y >= 3]^3
```

Listas: juntando vários coisas diferentes em um só objeto.

Para que servem listas?

A lista é a estrutura mais flexível do R. Uma lista pode conter objetos arbitrários, como data.frames, matrizes vetores e, inclusive, outras listas. Em geral, as listas são utilizadas para armazenar vários objetos diferentes que tenham algo em comum, como por exemplo, os resultados de cálculos estatísticos.

Para que servem listas?

Vejamos um exemplo prático de uma lista, o resultado da função `lm()` do R. Resultados omitidos pois são muito grandes.

```
# gera variáveis aleatórias
set.seed(1)
x <- rnorm(100)
y <- 10 + 2*x + rnorm(100)

# roda regressão linear
modelo <- lm(y ~ x)
summary(modelo)

# Veja que o objeto modelo é uma lista!
str(modelo, max.level = 1)
```

Criando uma lista: `list()`

Para criar uma lista utiliza-se a função `list()`:

```
minha_lista <- list(x = 1:10,
                    y = letters[1:5],
                    z = list(a = 1,
                             b = list(c = 2)))
str(minha_lista)
## List of 3
##  $ x: int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ y: chr [1:5] "a" "b" "c" "d" ...
##  $ z:List of 2
##   ..$ a: num 1
##   ..$ b:List of 1
##   .. ..$ c: num 2
```

Selecionando elementos da lista

Selecionando um objeto da lista:

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

minha_lista[["x"]]
##  [1]  1  2  3  4  5  6  7  8  9 10

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

Selecionando uma lista com certos elementos

Selecionando uma lista com o objeto (note a diferença):

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

minha_lista[c(1,2)]
## $x
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $y
## [1] "a" "b" "c" "d" "e"
```

Selecionando elementos da lista

As seleções podem ser concatenadas:

```
# seleciona primeiro o elemento z
# depois o elemento b (do elemento z)
# e depois o elemento c (do elemento b)
minha_lista$z$b$c
## [1] 2


# idem anterior
minha_lista[["z"]][["b"]][["c"]]
## [1] 2
```

Tabela resumo: selecionando elementos de uma lista

Resumindo as formas de seleção de um elemento de uma lista:

Operador	Descrição
`lista$x`	Retorna o objeto **x** da lista.
`lista[["x"]]`	Retorna o objeto **x** da lista.
`lista["x"]`	Retorna uma **lista** com o objeto `x`.

Adicionando elementos em uma lista: combine $ ou [[]] com <-

```
dados_da_empresa <- list(vendas = df.vendas,
                         funcionarios = funcionarios)

dados_da_empresa$comentario <- "Um comentario"

str(dados_da_empresa, max.level = 1)
## List of 3
##  $ vendas      :'data.frame':    3 obs. of  5 variables:
##  $ funcionarios:'data.frame':    2 obs. of  4 variables:
##  $ comentario  : chr "Um comentario"
```

Removendo elementos da lista: use NULL ou selecione todos exceto o que quer remover

```
# remove o elemento 'comentario' da lista
dados_da_empresa$comentario <- NULL

dados_da_empresa
## $vendas
##       segunda terça quarta quinta sexta
## João     1201  1027   1096    915   787
## Maria     927   920   1020    952  1039
## Ana      1043   833   1500   2000  1293
##
## $funcionarios
##    nome sexo salario experiencia
## 2 Maria    F    1200          12
## 3  José    M    1300          15
```

Aplicando funções em uma lista: sapply e lapply

As funções `sapply()` e `lapply()` aplicam uma função a cada elemento da lista:

```
set.seed(1)

# cria uma lista com 3 matrizes 2 x 2
lista_de_matrizes <- list(x = matrix(rnorm(4), ncol = 2),
                          y = matrix(rnorm(4), ncol = 2),
                          z = matrix(rnorm(4), ncol = 2))

# calcula o determinante das 3 matrizes ao mesmo tempo
sapply(lista_de_matrizes, det)
##     x     y     z
## -0.85  0.64  0.69
```

Exercícios

Sua vez.

- Considere a lista `minha_lista` criada anteriormente. Adicione uma lista `outra_lista` em `minha_lista` contendo um vetor de inteiros de 1 a 10 e e o data.frame `df.vendas`.
- Selecione ao mesmo tempo os elementos `outra_lista` e `x` de `minha_lista` e salve o resultado em outro objeto.
- Remova a `outra_lista` de `minha_lista`.

Soluções

```
minha_lista$outra_lista <- list(1:10, df.vendas)

outro_objeto <- minha_lista[c("outra_lista", "x")]
str(outro_objeto)

minha_lista[["outra_lista"]] <- NULL
```