Aula 3: Vetores, matrizes e arrays

Profa. Yana Borges

Fevereiro de 2022

1. Vetores

Até agora, nós usamos o operador de dois pontos, `:`, para criar sequências de um número para outro, e a função `c` para concatenar valores e vetores para criar vetores mais longos.

Para recapitular:

```
8.5:4.5 #sequência de números de 8.5 até 4.5
```

```
## [1] 8.5 7.5 6.5 5.5 4.5
```

```
c(1, 1:3, c(5, 8), 13) #valores concatenados em um único vetor
```

```
## [1]  1  1  2  3  5  8 13
```

A função `vector` cria um vetor de um tipo e comprimento especificados. Cada um dos valores no resultado é zero, `FALSE` ou uma string vazia, ou o equivalente a "nada":

```
vector("numeric", 5)
```

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

```
vector("complex", 5)
```

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

```
vector("logical", 5)
```

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

```
vector("character", 5)
```

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

```
vector("list", 5)
```

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

```
## [[5]]
## NULL
```

Nesse último exemplo, NULL é um valor especial "vazio" (não deve ser confundido com NA, que indica um ponto de dados ausente). Por conveniência, existem outras funções equivalentes para evitar que você precise digitar ao criar vetores dessa maneira:

```
numeric(5)
```

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

```
complex(5)
```

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

```
logical(5)
```

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

```
character(5)
```

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

1.1 Sequências

Além do operador de dois pontos, existem várias funções para criar sequências mais gerais. A função `seq` é a mais geral e permite especificar sequências de muitas maneiras diferentes. Na prática, porém, você nunca deve precisar chamá-lo, pois existem três outras funções de sequência especializadas que são mais rápidas e mais fácil de usar, abrangendo casos de uso específicos.

A função `seq.int` nos permite criar uma sequência de um número para outro. Com duas entradas, funciona exatamente como o operador de dois pontos:

```
seq.int(3, 12) #igual a 3:12
```

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

`seq.int` é um pouco mais geral que `:`, pois permite especificar a distância entre os valores intermediários:

```
seq.int(3, 12, 2)
```

```
## [1]  3  5  7  9 11
```

```
seq.int(0.1, 0.01, -0.01)
```

```
##  [1] 0.10 0.09 0.08 0.07 0.06 0.05 0.04 0.03 0.02 0.01
```

A função `seq_along` cria uma sequência de 1 até o comprimento de sua entrada:

```
pp <- c("Peter", "Piper", "picked", "a", "peck", "of", "pickled", "peppers")
for(i in seq_along(pp)) print(pp[i])
```

```
## [1] "Peter"
## [1] "Piper"
## [1] "picked"
## [1] "a"
## [1] "peck"
## [1] "of"
## [1] "pickled"
## [1] "peppers"
```

Para cada um dos exemplos anteriores, você pode substituir `seq.int`, `seq_len` ou `seq_along` por `seq` simples e obter a mesma resposta, embora não seja necessário fazê-lo.

1.2 Length

Todos os vetores têm um comprimento, que nos diz quantos elementos eles contêm. Este é um integer não negativo (sim, vetores de comprimento zero são permitidos), e você pode acessar esse valor com a função `length`. Os valores ausentes ainda contam para o comprimento:

```
length(1:5)
```

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

```
length(c(TRUE, FALSE, NA))
```

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

Uma possível fonte de confusão são os vetores de caracteres. Com estes, o comprimento é o número de strings, não o número de caracteres em cada string. Para isso, devemos usar `nchar`:

```
sn <- c("Sheena", "leads", "Sheila", "needs")
length(sn)
```

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

```
nchar(sn)
```

```
## [1] 6 5 6 5
```

1.3 Indexação de vetores

Muitas vezes podemos querer acessar apenas parte de um vetor, ou talvez um elemento individual. Isso é chamado de indexação e é realizado com colchetes, `[]`.

Considere o vetor:

```
x <- (1:5)^2
x
```

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

Esses três métodos de indexação retornam os mesmos valores:

```
x[c(1, 3, 5)] #imprime elementos na posição 1, 3 e 5
```

```
## [1]  1  9 25
```

```
x[c(-2, -4)] #imprime todos os elementos, menos 2º e 4º
```

```
## [1]  1  9 25
```

```
x[c(TRUE, FALSE, TRUE, FALSE, TRUE)] #imprime o que for verdadeiro
```

```
## [1]  1  9 25
```

Após nomear cada elemento, este método também retorna os mesmos valores:

```
names(x) <- c("um", "quatro", "nove", "dezesseis", "vinte e cinco")
x[c("um", "nove", "vinte e cinco")]
```

```
##            um          nove vinte e cinco
##             1             9            25
```

A mistura de valores positivos e negativos não é permitida e gerará um erro:

```
x[c(1, -1)]
```

```
## Error in x[c(1, -1)]: somente 0's podem ser usados junto com subscritos negativos
```

Se você usar números positivos ou valores lógicos como índice, os índices ausentes correspondem aos valores ausentes no resultado:

```
x[c(1, NA, 5)]
```

```
##            um          <NA> vinte e cinco
##             1            NA            25
```

```
x[c(TRUE, FALSE, NA, FALSE, TRUE)]
```

```
##            um          <NA> vinte e cinco
##             1            NA            25
```

Valores ausentes não fazem sentido para índices negativos e causam um erro:

```
x[c(-2,NA)] #Isso também não faz sentido!
```

```
## Error in x[c(-2, NA)]: somente 0's podem ser usados junto com subscritos negativos
```

Índices fora do intervalo, além do comprimento do vetor, não causam erro, mas retornam o valor ausente NA. Na prática, geralmente é melhor garantir que seus índices estejam dentro do intervalo do que usar valores fora do intervalo:

```
x[6]
```

```
## <NA>
##   NA
```

Não passar nenhum índice retornará todo o vetor:

```
x[]
```

```
##            um        quatro          nove     dezesseis vinte e cinco
##             1             4             9            16            25
```

A função `which` retorna os **locais** onde um vetor lógico é `TRUE`. Isso pode ser útil para alternar da indexação lógica para a indexação inteira:

```
which(x > 10)
```

```
##     dezesseis vinte e cinco
##             4             5
```

`which.min` e `which.max` são atalhos mais eficientes para `which(min(x))` e `which(max(x))`, respectivamente:

```
which.min(x)
```

```
## um
##  1
```

```
which.max(x)
```

```
## vinte e cinco
##             5
```

1.4 Repetição de vetores

Até agora, todos os vetores que adicionamos têm o mesmo comprimento. Você pode estar se perguntando: "O que acontece se eu tentar fazer aritmética em vetores de diferentes comprimentos?" Se tentarmos adicionar um único número a um vetor, esse número será adicionado a cada elemento do vetor:

```
1:5 + 1
```

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

```r
1 + 1:5
```

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

Ao adicionar dois vetores, R reciclará elementos no vetor mais curto para corresponder ao mais longo:

```r
1:5 + 1:15
```

```
##  [1]  2  4  6  8 10  7  9 11 13 15 12 14 16 18 20
```

Se o comprimento do vetor mais longo não for um múltiplo do comprimento do mais curto, um aviso será dado:

```r
1:5 + 1:7
```

```
## Warning in 1:5 + 1:7: comprimento do objeto maior não é múltiplo do comprimento
## do objeto menor
```

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

Deve-se enfatizar que só porque podemos fazer aritmética em vetores de diferentes comprimentos, isso não significa que *devemos*. Adicionar um valor escalar a um vetor é bom, mas, caso contrário, podemos nos confundir. É muito melhor criar explicitamente vetores de igual comprimento antes de operarmos neles.

A função `rep` é muito útil para esta tarefa, permitindo criar um vetor com elementos repetidos:

```r
rep(1:5, 3)
```

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

```r
rep(1:5, each = 3)
```

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

```r
rep(1:5, times = 1:5)
```

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

```r
rep(1:5, length.out = 7)
```

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

Assim como a função seq, rep tem uma variante

mais simples e rápida, rep.int, para o caso mais comum:

```r
rep.int(1:5, 3) #o mesmo que rep(1:5, 3)
```

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

2 Matrizes e Arrays

As variáveis vetoriais que vimos até agora são objetos unidimensionais, pois têm comprimento, mas não têm outras dimensões. As matrizes contêm dados retangulares multidimensionais. "Retangular" significa que cada linha tem o mesmo comprimento, e da mesma forma para cada coluna e outras dimensões. Matrizes são um caso especial de matrizes bidimensionais.

2.1 Criando Arrays e Matrizes

Para criar um array, você chama a função `array`, passando um vetor de valores e um vetor de dimensões.

```
(tres_d_array <- array(
1:24,
dim = c(4, 3, 2),
))
```

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

```
class(tres_d_array)
```

```
## [1] "array"
```

Opcionalmente, você também pode fornecer nomes para cada dimensão:

```
(three_d_array <- array(
1:24,
dim = c(4, 3, 2),
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei"),
c("un", "deux")
)
))
```

```
## , , un
##
##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
##
## , , deux
##
##       ein zwei drei
## one    13   17   21
## two    14   18   22
## three  15   19   23
## four   16   20   24
```

```
class(three_d_array)
```

```
## [1] "array"
```

A sintaxe para criar matrizes é semelhante, mas em vez de passar um argumento `dim`, você especifica o número de linhas ou o número de colunas:

```
(a_matrix <- matrix(
1:12,
nrow = 4, #ncol = 3 funciona igual
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei")
)
))
```

```
##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
```

```
class(a_matrix)
```

```
## [1] "matrix" "array"
```

Essa matriz também pode ser criada usando a função `array`. A seguinte matriz bidimensional é idêntica à matriz que acabamos de criar:

```
(two_d_array <- array(
1:12,
dim = c(4, 3),
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei")
)
))
```

```
##       ein zwei drei
## one     1    5    9
## two     2    6   10
## three   3    7   11
## four    4    8   12
```

```
identical(two_d_array, a_matrix)
```

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

```
class(two_d_array)
```

```
## [1] "matrix" "array"
```

Quando você cria uma matriz, os valores que você passou preenchem a matriz em colunas. Também é possível preencher a matriz linha a linha especificando o argumento `byrow = TRUE`:

```
matrix(
1:12,
nrow = 4,
byrow = TRUE,
dimnames = list(
c("one", "two", "three", "four"),
c("ein", "zwei", "drei")
)
```

```
)
```

```
##       ein zwei drei
## one     1    2    3
## two     4    5    6
## three   7    8    9
## four   10   11   12
```

2.2 Linhas, colunas e dimensões

Tanto para matrizes quanto para arrays, a função `dim` retorna um vetor de inteiros das dimensões da variável:

```
dim(three_d_array)
```

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

```
dim(a_matrix)
```

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

Para matrizes, as funções `nrow` e `ncol` retornam o número de linhas e colunas, respectivamente:

```
nrow(a_matrix)
```

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

```
ncol(a_matrix)
```

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

`nrow` e `ncol` também funcionam em arrays, retornando a primeira e a segunda dimensões, respectivamente, mas geralmente é melhor usar `dim` para objetos de dimensões mais altas:

```
nrow(three_d_array)
```

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

```
ncol(three_d_array)
```

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

A função `length` que usamos anteriormente com vetores também funciona em matrizes e arrays. Neste caso retorna o produto de cada uma das dimensões:

```
length(three_d_array)
```

```
## [1] 24
```

```
length(a_matrix)
```

```
## [1] 12
```

Também podemos remodelar uma matriz ou array atribuindo uma nova dimensão com `dim`. Isso deve ser usado com cuidado, pois remove os nomes das dimensões:

```
dim(a_matrix) <- c(6, 2)
a_matrix
```

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

```
## [5,]    5   11
## [6,]    6   12
```

`nrow`, `ncol` e `dim` retornam `NULL` quando aplicados a vetores. As funções `NROW` e `NCOL` são contrapartes de `nrow` e `ncol` que fingem que os vetores são matrizes com uma única coluna (ou seja, vetores de coluna no sentido matemático):

```
identical(nrow(a_matrix), NROW(a_matrix))
```

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

```
identical(ncol(a_matrix), NCOL(a_matrix))
```

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

```
recaman <- c(0, 1, 3, 6, 2, 7, 13, 20)
nrow(recaman)
```

```
## NULL
```

```
NROW(recaman)
```

```
## [1] 8
```

```
ncol(recaman)
```

```
## NULL
```

```
NCOL(recaman)
```

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

```
dim(recaman)
```

```
## NULL
```

2.3 Nomes de linha, coluna e dimensão

Da mesma forma que os vetores têm nomes para os elementos, as matrizes têm nomes de linhas e nomes de colunas para as linhas e colunas. Por razões históricas, há também uma função `row.names`, que faz a mesma coisa que `rownames`, mas não há `col.names` correspondente, então é melhor ignorá-la e usar `rownames`. Assim como no caso de `nrow`, `ncol` e `dim`, a função equivalente para arrays é `dimnames`. O último retorna uma lista (veremos mais tarde) de vetores de caracteres. No trecho de código a seguir, `a_matrix` foi restaurada ao seu estado anterior, antes que suas dimensões fossem alteradas:

```
rownames(a_matrix)
```

```
## [1] "one"   "two"   "three" "four"
```

```
colnames(a_matrix)
```

```
## [1] "ein"  "zwei" "drei"
```

```
dimnames(a_matrix)
```

```
## [[1]]
## [1] "one"   "two"   "three" "four"
##
## [[2]]
## [1] "ein"  "zwei" "drei"
```

```
rownames(three_d_array)
```

```
## [1] "one"   "two"   "three" "four"
```

```
colnames(three_d_array)
```

```
## [1] "ein"  "zwei" "drei"
```

```
dimnames(three_d_array)
```

```
## [[1]]
## [1] "one"   "two"   "three" "four"
##
## [[2]]
## [1] "ein"  "zwei" "drei"
##
## [[3]]
## [1] "un"   "deux"
```

2.4 Indexando arrays

A indexação funciona da mesma forma que com vetores, exceto que agora temos que especificar um índice para mais de uma dimensão. Como antes, usamos colchetes para denotar um índice e ainda temos quatro opções para especificar o índice (inteiros positivos, inteiros negativos, valores lógicos e nomes de elementos). É perfeitamente permitido especificar os índices para diferentes dimensões de diferentes maneiras. Os índices para cada dimensão são separados por vírgulas:

```
a_matrix[1, c("zwei", "drei")] #elementos na 1ª linha, 2ª e 3ª colunas
```

Para incluir tudo de uma dimensão, deixe o índice correspondente em branco:

```
a_matrix[1, ]
```

```
##  ein zwei drei
##    1    5    9
```

```
a_matrix[, c("zwei", "drei")] #todos da primeira linha
```

```
##       zwei drei
## one      5    9
## two      6   10
## three    7   11
## four     8   12
```

```
a_matrix[, c("zwei", "drei")] #todas da segunda e terceira colunas
```

```
##       zwei drei
## one      5    9
## two      6   10
## three    7   11
## four     8   12
```

Exercícios

1. Crie uma sequência com os primeiros vinte números pares. Nomeie os elementos do vetor que você acabou de criar com as primeiras 20 letras do alfabeto. Selecione os números onde o nome é uma vogal.
2. A função `diag` tem vários usos, um dos quais é pegar um vetor como entrada e criar uma matriz quadrada com esse vetor na diagonal. Crie uma matriz de 11 por 11 com a sequência de 5 a 0 a 5 (ou seja, 5, 4, ... , 1, 0, 1, ..., 5).