Introdução ao R – 2011
Roteiro da aula 3

Alexandre Rademaker

26 de Janeiro de 2011

Resumo

Roteiro baseado no livro: A Beginner's Guide to R (2009), Zuur, Ieno, Meesters. Springer.

1 Vetores

Estruturas de dados unidimensionais. A forma mais comum para criação de vetores é através da função *c*, de concatenação.

```
> Wingcrd <- c(59, 55, 53.5, 55, 52.5,
+     57.5, 53, 55)
> Tarsus <- c(22.3, 19.7, 20.8, 20.3,
+     20.8, 21.5, 20.6, 21.5)
> Head <- c(31.2, 30.4, 30.6, 30.3,
+     30.3, 30.8, 32.5, NA)
> Wt <- c(9.5, 13.8, 14.8, 15.2, 15.5,
+     15.6, 15.6, 15.7)
```

Na aula, para evitar a digitação destes números, também usamos a função *sample*:

```
> Wingcrd <- sample(0:100, 8)
> Tarsus <- sample(0:100, 8)
> Head <- sample(0:100, 8)
> Wt <- sample(0:100, 8)
```

E introduzimos no vetor *Head* um `NA`:

```
> Head[8] <- NA
> Head

[1] 52 15 13 66 55 58 23 NA
```

Vetores também podem armazenar cadeias de caracteres, também chamadas de "strings". Strings são sequencias de caracteres entre aspas.

```
> colors <- c("red", "yellow", "blue")
> more.colors <- c(colors, "green",
+     "magenta", "cyan")
```

Mas o importante é entender que vetores podem conter apenas um tipo de dado. Se um elemento for "string", todos os demais também serão:

```
> z <- c("red", "green", 1)
> z

[1] "red"   "green" "1"
```

Vetores de caracteres são muito usados em R, principalmente para manipulação dos nomes de linhas e colunas dos "data.frames". Alguns operações comuns são:

Seleção de partes da string:

```
> substr(colors, 1, 2)

[1] "re" "ye" "bl"
```

Concatenação de strings:

```
> paste(colors, "flowers")

[1] "red flowers"    "yellow flowers"
[3] "blue flowers"

> paste("several ", colors, "s", sep = "")
```

```
[1] "several reds"    "several yellows"
[3] "several blues"
```

```
> paste("I like", colors, collapse = ", ")
```

```
[1] "I like red, I like yellow, I like blue"
```

Números são transformados em "strings" de forma automática, quando necessário. Observe abaixo a diferença entre o argumento "sep" e "collapse". Experimente.

```
> paste(1:10, collapse = ", ")
```

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

```
> paste(1:10, 1:10, sep = ",")
```

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

Também falamos sobre expressões regulares. As funções *grep* e *gsub* usam expressões regulares para casamento de padrões em cadeias de caracteres:

```
> vars <- c(paste("V", 100:105, sep = ""),
+     "Head", "Squid", "Tarsus", "Wingcrd",
+     "Wt")
> vars
```

```
 [1] "V100"    "V101"    "V102"    "V103"
 [5] "V104"    "V105"    "Head"    "Squid"
 [9] "Tarsus"  "Wingcrd" "Wt"
```

```
> grep("^V", vars)
```

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

```
> vars[grep("^V", vars)]
```

```
[1] "V100" "V101" "V102" "V103" "V104" "V105"
```

A expressão regular `^V` significa "todas as cadeias de caracteres iniciadas com "V". Agora vamos selecionar todas as cadeias terminadas com "d".

```
> vars[grep("d$", vars)]
```

```
[1] "Head"    "Squid"   "Wingcrd"
```

Expressões regulares são muito importantes para manipulação de textos. Infelizmente, apenas introduzimos o assunto. Pode-se fazer muito mais do que apenas selecionar cadeias que começam ou terminam com determinados caracteres. Por exemplo, podemos selecionar as cadeias que contém números:

```
> vars[grep("[0-9]+", vars)]
```

```
[1] "V100" "V101" "V102" "V103" "V104" "V105"
```

Aqui o operador + foi usado para significar "uma ou mais repetições. O operador [foi usado para indicar um conjunto dos caracteres possíveis. O operador − foi usado para definir uma faixa de valores de 0 à 9. Ou seja, definimos um padrão que "casa" com qualquer sequência de algarismos.

2 Variáveis categóricas

O tipo "factor" do R armazena variáveis categóricas.

```
> grp1 <- c("control", "treatment",
+     "control", "treatment")
> fgrp1 <- factor(grp1)
```

Vejam a diferença:

```
> str(grp1)
```

```
 chr [1:4] "control" "treatment" ...
```

```
> str(fgrp1)
```

```
 Factor w/ 2 levels "control","treatment": 1 2 1 2
```

Ou ainda, usando a função *summary*:

```
> summary(grp1)
```

```
   Length     Class      Mode
        4 character character
```

```
> summary(fgrp1)
```

```
  control treatment
        2         2
```

Factors podem ser transformados em números ou cadeias de caracteres, bastando a seleção dos seus níveis ou rótulos:

```
> as.integer(fgrp1)

[1] 1 2 1 2

> levels(fgrp1)

[1] "control"   "treatment"

> levels(fgrp1)[as.integer(fgrp1)]

[1] "control"   "treatment" "control"
[4] "treatment"
```

3 Valores Booleanos

Também chamados, "valores de verdade". Vetores booleanos podem ser usados para selecionar elementos de vetores:

```
> a <- c(TRUE, FALSE, FALSE, TRUE)
> b <- c(13, 7, 8, 2)
> b[a]

[1] 13  2
```

Quando necessários, valores booleanos são convertidos para numéricos onde `FALSE` tem valor 0 e `TRUE` valor 1:

```
> sum(a)

[1] 2
```

O inverso também pode ocorrer, conversão de valores numéricos para valores booleanos. O zero é o `FALSE` e qualquer outro valor é `TRUE`:

```
> a & (b - 2)

[1]  TRUE FALSE FALSE FALSE
```

R contém todos os operadores básicos para trabalhamos com valores booleanos:

```
> !a

[1] FALSE  TRUE  TRUE FALSE

> c <- c(TRUE, TRUE, FALSE, TRUE)
> a & c

[1]  TRUE FALSE FALSE  TRUE

> a && c

[1] TRUE

> a | c

[1]  TRUE  TRUE FALSE  TRUE

> a || c

[1] TRUE
```

Observem a diferença entre os operadores `&` (`&&`) e `|` (`||`). As versões curtas operam elemento à elemento dos vetores. As versões longas operam apenas nas primeiras posições dos vetores e sempre retornam valores escalares.

Em geral, valores booleanos são criados quando executamos comparações ou, de forma mais genérica, operações relacionais, com outros tipos de valores. Seja:

```
> a <- c(3, 6, 9)
> b <- c(4, 6, 8)
```

Alguns exemplos de operações relacionais com estes vetores são:

```
> a < b

[1]  TRUE FALSE FALSE

> a <= b

[1]  TRUE  TRUE FALSE

> a == 4

[1] FALSE FALSE FALSE

> a != 4

[1] TRUE TRUE TRUE

> a <= 3

[1]  TRUE FALSE FALSE

> a[a < b]

[1] 3
```

4 O valor NA

Como remover o valor NA de vetores? Vejamos um vetor que já construimos na aula anterior:

```
> alguns.pares <- NULL
> alguns.pares[seq(2, 20, 2)] <- seq(2,
+     20, 2)
> alguns.pares

 [1] NA  2 NA  4 NA  6 NA  8 NA 10 NA 12 NA
[14] 14 NA 16 NA 18 NA 20
```

Qualquer comparação com NA resulta em NA:

```
> NA == 1

[1] NA

> NA == "teste"

[1] NA

> NA == NA

[1] NA
```

Logo, não podemos simplesmente tentar:

```
> alguns.pares != NA

 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[14] NA NA NA NA NA NA NA
```

Pois o vetor resultante será todo composto por NA. Mas podemos usar a função *is.na*:

```
> is.na(alguns.pares)

 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE
 [7]  TRUE FALSE  TRUE FALSE  TRUE FALSE
[13]  TRUE FALSE  TRUE FALSE  TRUE FALSE
[19]  TRUE FALSE

> alguns.pares[!is.na(alguns.pares)]

 [1]  2  4  6  8 10 12 14 16 18 20
```

5 Combinando dados

Obviamente, é comum precisamos de mais dimensões que apenas 1 para nossas estruturas de dados. Podemos combinar vetores formando estruturas bidimensionais:

```
> Z <- cbind(Wingcrd, Tarsus, Head,
+     Wt)
> Z

     Wingcrd Tarsus Head Wt
[1,]      83     50   52 96
[2,]      18     85   15 84
[3,]      42     64   13 19
[4,]      60     74   66 77
[5,]      63     41   55 39
[6,]      88      8   58 38
[7,]      33     47   23 91
[8,]      26     98   NA 59

> dim(Z)

[1] 8 4
```

Então a resposta da função *dim* é um vetor, logo posso recuperar o número de linhas com:

```
> dim(Z)[1]

[1] 8
```

Além da função *dim*, podemos obter o número de linhas e colunas com:

```
> nrow(Z)

[1] 8

> ncol(Z)

[1] 4
```

E podemos naturalmente usar o operador "[" para selecionar partes da nova estrutura bidimensional:

```
> Z[, 1]

[1] 83 18 42 60 63 88 33 26
```

```
> Z[1:8, 1]

[1] 83 18 42 60 63 88 33 26

> Z[2, ]

Wingcrd  Tarsus    Head      Wt
     18      85      15      84

> Z[2, 1:4]

Wingcrd  Tarsus    Head      Wt
     18      85      15      84

> Z[, c(-1, -3)]

     Tarsus Wt
[1,]     50 96
[2,]     85 84
[3,]     64 19
[4,]     74 77
[5,]     41 39
[6,]      8 38
[7,]     47 91
[8,]     98 59

> Z[, -c(1, 3)]

     Tarsus Wt
[1,]     50 96
[2,]     85 84
[3,]     64 19
[4,]     74 77
[5,]     41 39
[6,]      8 38
[7,]     47 91
[8,]     98 59
```

Notem a diferença entre selecionar o valor da primeira linha e segunda coluna, primeiro comando, de selecionar o valor da posição 1 e 2, segundo comando. No primeiro caso, o operador "[" recebeu dois argumentos. No segundo caso, recebeu apenas um vetor. Ainda no segundo caso, a matriz foi tratada como vetor, de forma contínua.

```
> Z[1, 2]

Tarsus
    50

> Z[c(1, 2)]

[1] 83 18
```

Podemos também combinar os vetores por linhas:

```
> Z2 <- rbind(Wingcrd, Tarsus, Head,
+     Wt)
> Z2

        [,1] [,2] [,3] [,4] [,5] [,6] [,7]
Wingcrd   83   18   42   60   63   88   33
Tarsus    50   85   64   74   41    8   47
Head      52   15   13   66   55   58   23
Wt        96   84   19   77   39   38   91
        [,8]
Wingcrd   26
Tarsus    98
Head      NA
Wt        59
```

6 Outra forma de criar vetores e matrizes

Quando estivemos preocupados com performance, é sempre aconselhável inicializar os vetores com seus tamanhos pré-alocados:

```
> W <- vector(length = 8)
> W[1] <- 59
> W[2] <- 55
> W[3] <- 53.5
> W[4] <- 55
> W[5] <- 52.5
> W[6] <- 57.5
> W[7] <- 53
> W[8] <- 55
```

Na construção de matrizes, vale a mesma regra de reciclagem de valores. Abaixo, o valor 10 foi reciclado, transformado em um vetor de 32 posições com o valor 10 para ser então "organizado" na forma de uma matriz.

```
> matrix(10, nrow = 8, ncol = 4)

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

Esta mesma matriz poderia ser construída apenas com a informação do número desejado de linhas, o número de colunas e tamanho do vetor passado permitem que o R calcule as colunas necessárias:

```
> matrix(sample(1:100, 32), nrow = 8)

     [,1] [,2] [,3] [,4]
[1,]   10   15   14   71
[2,]   48   40    8   92
[3,]   98   27   29   76
[4,]   22  100   18   23
[5,]   99   62   91   38
[6,]   58   25   53   72
[7,]   89   97    5   68
[8,]    6   84   69   66
```

Também podemos atribuir vetores a linhas ou colunas inteiras de matrizes:

```
> Dmat <- matrix(nrow = 8, ncol = 4)
> Dmat[, 1] <- c(59, 55, 53.5, 55, 52.5,
+     57.5, 53, 55)
> Dmat[, 2] <- c(22.3, 19.7, 20.8, 20.3,
+     20.8, 21.5, 20.6, 21.5)
> Dmat[, 3] <- c(31.2, 30.4, 30.6, 30.3,
+     30.3, 30.8, 32.5, NA)
> Dmat[, 4] <- c(9.5, 13.8, 14.8, 15.2,
+     15.5, 15.6, 15.6, 15.7)
> Dmat

     [,1] [,2] [,3] [,4]
[1,] 59.0 22.3 31.2  9.5
[2,] 55.0 19.7 30.4 13.8
[3,] 53.5 20.8 30.6 14.8
[4,] 55.0 20.3 30.3 15.2
[5,] 52.5 20.8 30.3 15.5
[6,] 57.5 21.5 30.8 15.6
[7,] 53.0 20.6 32.5 15.6
[8,] 55.0 21.5   NA 15.7
```

E podemos dar nomes as dimensões:

```
> colnames(Dmat) <- c("Wingcrd", "Tarsus",
+     "Head", "Wt")
> Dmat

     Wingcrd Tarsus Head   Wt
[1,]    59.0   22.3 31.2  9.5
[2,]    55.0   19.7 30.4 13.8
[3,]    53.5   20.8 30.6 14.8
[4,]    55.0   20.3 30.3 15.2
[5,]    52.5   20.8 30.3 15.5
[6,]    57.5   21.5 30.8 15.6
[7,]    53.0   20.6 32.5 15.6
[8,]    55.0   21.5   NA 15.7
```

O mesmo vale para os nomes de linhas. Mas, em geral, as linhas não tem nomes:

```
> rownames(Dmat)

NULL

> rownames(Dmat) <- paste("O", 1:8,
+     sep = "")
> Dmat

   Wingcrd Tarsus Head   Wt
O1    59.0   22.3 31.2  9.5
O2    55.0   19.7 30.4 13.8
O3    53.5   20.8 30.6 14.8
O4    55.0   20.3 30.3 15.2
O5    52.5   20.8 30.3 15.5
O6    57.5   21.5 30.8 15.6
O7    53.0   20.6 32.5 15.6
O8    55.0   21.5   NA 15.7
```

Também podemos passar o resultado da função *cbind* para a função *as.matrix* que transforma o objeto dado em matriz. Obviamente, neste caso isto não é necessário, a saída do cbind já é uma matriz.

```
> Dmat2 <- as.matrix(cbind(Wingcrd,
+     Tarsus, Head, Wt))
```

7 Data frames

Até agora, todas as estruturas, vetores ou matrizes, são forçadas a armazenas em todos os seus componentes, o mesmo tipo de dado (números, cadeias de caracteres, factors, booleanos). Data frames são estruturas de dados bidimensionais onde cada coluna pode conter um tipo de dado. A idéia é que dados armazenados em data frames devem ser lidos como as linhas representando observações e as colunas os valores observados para cada variável.

```
> Dfrm <- data.frame(WC = Wingcrd, TS = Tarsus,
+     HD = Head, W = Wt)
> Dfrm
```

```
  WC TS HD  W
1 83 50 52 96
2 18 85 15 84
3 42 64 13 19
4 60 74 66 77
5 63 41 55 39
6 88  8 58 38
7 33 47 23 91
8 26 98 NA 59
```

Observem a diferença entre a variável Wt do data.frame $Dfrm$ e a variável Wt do ambiente. Observem também o uso do operador `$` para selecionar uma coluna do data.frame.

```
> Dfrm$W
```

```
[1] 96 84 19 77 39 38 91 59
```

```
> tmp <- Wt
> rm(Wt)
> Dfrm$W
```

```
[1] 96 84 19 77 39 38 91 59
```

```
> Wt <- tmp
> rm(tmp)
```

Uma coluna do data.frame é um vetor, e podemos selecionar valores usando tudo que já vimos sobre vetores. Em um data.frame, todas as colunas devem ter a mesma quantidade de valores, são vetores de igual dimensão. Obviamente, algumas posições podem conter NA.

8 Listas

Finalmente, o tipo de dado mais genérico de R, as listas. Listas podem conter qualquer "mix" de elementos. De diferentes tamanhos e tipos. Listas são estruturas recursivas, podem conter outras listas.

```
> x1 <- c(1, 2, 3)
> x2 <- sample(letters, 5)
> x3 <- 3
> x4 <- matrix(sample(1:100, 6), nrow = 3,
+     ncol = 2)
```

Na criação das listas, podemos ou não dar nomes aos componentes. Vejam os dois últimos argumentos abaixo:

```
> Y <- list(x1 = x1, x2 = x2, x3 = x3,
+     x4, Dfrm)
> Y
```

```
$x1
[1] 1 2 3

$x2
[1] "z" "o" "l" "j" "y"

$x3
[1] 3

[[4]]
     [,1] [,2]
[1,]   10   77
[2,]   64   93
[3,]   81   62

[[5]]
  WC TS HD  W
1 83 50 52 96
2 18 85 15 84
3 42 64 13 19
4 60 74 66 77
5 63 41 55 39
6 88  8 58 38
7 33 47 23 91
8 26 98 NA 59
```

Para acessar partes de uma lista, podemos usar o operador `[` ou `[[`. O primeiro, o mesmo usado para vetores, faz um filtro na lista, retornando uma lista filtrada para os elementos solicitados. O segundo, recupera apenas um componente da lista, e retorna o componente, não uma lista de um componente.

```
> Y[c(1, 5)]

$x1
[1] 1 2 3

[[2]]
  WC TS HD  W
1 83 50 52 96
2 18 85 15 84
3 42 64 13 19
4 60 74 66 77
5 63 41 55 39
6 88  8 58 38
7 33 47 23 91
8 26 98 NA 59

> Y[[5]]

  WC TS HD  W
1 83 50 52 96
2 18 85 15 84
3 42 64 13 19
4 60 74 66 77
5 63 41 55 39
6 88  8 58 38
7 33 47 23 91
8 26 98 NA 59
```

Também podemos usar os nomes dos componentes ao invés de seus índices:

```
> Y["x1"]

$x1
[1] 1 2 3

> Y[["x1"]]

[1] 1 2 3
```

A diferença fica evidênte se observarmos a estrutura do retorno dos operadores `[` e `[[`:

```
> str(Y["x1"])

List of 1
 $ x1: num [1:3] 1 2 3

> str(Y[["x1"]])

 num [1:3] 1 2 3
```

Por que precisamos de listas? Listas são estruturas bastante genéricas, geralmente usadas por funções complexas que precisam retornar diferentes informações.

```
> MyModel <- lm(WC ~ Wt, data = Dfrm)
> MyModel

Call:
lm(formula = WC ~ Wt, data = Dfrm)

Coefficients:
(Intercept)           Wt
    59.7455      -0.1292
```

Experimente no console do R:

```
> str(MyModel)
```

A saída é muito grande para colocar aqui neste documento!

Vejam como a função *lm* pode receber no seu argumento "data" pode receber um data.frame, permitindo que nos demais argumentos, ao invés de passarmos `Dfrm$WC` possamos simplesmente passar o nome da variável (coluna) do data.frame nomeado no argumento "data". No entanto, nem todas as funções podem receber uma argumento "data". O comando abaixo seria um erro:

```
> mean(WC, data = Dfrm)
```