cap.3 - R Objects

Washington Candeia de Araujo

Maio, 10 2018

Contents

1 Iniciando no Projeto 2: Playing Cards

Neste capítulo o R será usado para construir um baralho de 52 cartas.

1.1 Atomic Vectors

É o tipo mais simples de objeto em R. A maioria das estruturas em R são construídas a partir deste tipo de objeto.
Um **vetor atômico** é um simples vetor contendo dados.

Vetores armazenam dados de forma unidimensional, e cada vetor armazena um tipo de dado.

Tipos de dados básicos de vetores atômicos:
- *doubles*
- *integers*
- *characters*
- *logical*
- *complex*
- *raw*

Para fazer isso é preciso ficar atento às características de cada tipo destes seis tipos de dados.

```
# Nosso dado é um vetor simples
die <- c(1, 2, 3, 4, 5, 6)
is.vector(die)
```

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

```r
# Um único valor também é um vetor
five <- 5
is.vector(five)
```

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

R reconhecerá cada uma das convenções específicas para criação de cada tipo de dados. Por exemplo, se quisermos criar um vetor contendo ***integers*** ou ***characters*** precisamos especificar cada um destes tipos de dados:

```r
# Criando um vetor de inteiros
int <- c(1L, 5L)

# Criando um vetor de caracteres
text <- c("espadas", "copas")

print(int)
```

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

```r
print(text)
```

```
## [1] "espadas" "copas"
```

Agora, iremos observar cada um dos tipos de dados para vetores atômicos.

1.2 Doubles

Vetores com este tipo de dado armazena números comuns. Os números podem ser positivos ou negativos, pequenos ou grandes, podendo ter dígitos decimais. No geral, R salva qualquer número que digitamos como um *double*.

É possível descobrir que tipo estamos usando através da função `typeof`.

```r
typeof(die)
```

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

Algumas funções de R se referem a *double* como ***numerics***.

> *Double* é um termo usado em Ciência da Computação para se referir ao número específico de *bytes* que o computador usa para armazenar um número.

1.3 Integers

Vetores com este tipo de dado armazenam números que podem ser escritos sem um componente decimal. O próprio autor deste livro recomenda não usar este tipo, pois é possível usar *double* para armazenar como um inteiro.

```r
# Especificando inteiros em um vetor
int2 <- c(-1L, -60L, 88L, 5L, 6L)

print(int2)
```

```
## [1]  -1 -60  88   5   6
```

R não salvará um número como um inteiro (*integer*) a não ser que especifiquemos.

A diferença entre um *double* e um *integer* é como R salvará cada um deles na memória do computador. Inteiros são definidos mais precisamente na memória do computador do que *doubles*.

1.4 Characters

Caracteres são bem fáceis de identificar. Eles precisam estar sob aspas e seu tipo é igualmente fácil de identificar.

```
# Characters
text <- c("Hello", "World")

print(text)
```

```
## [1] "Hello" "World"
```

```
typeof(text)
```

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

1.5 Logical

Lógicos estão conforme a lógica matemática, onde os ***booleanos*** são utilizados para estas manipulações.

and - será TRUE se ambos forem TRUE
or - será FALSE se ambos forem FALSE

Logical **AND**	**Resultado**
FALSE AND FALSE	FALSE
FALSE AND TRUE	FALSE
TRUE AND FALSE	FALSE
TRUE AND TRUE	TRUE

Logical **OR**	**Resultado**
FALSE OR FALSE	FALSE
FALSE OR TRUE	TRUE
TRUE OR FALSE	TRUE
TRUE OR TRUE	TRUE

1.6 Atributes

Um **atributo** é um pedaço de informação que você pode anexar a um ***atomic vector*** ou qualquer objeto R. Um atributo seria semelhante aos metadados uma vez que não será visível nem aparecerá no resultado do código. Então, por que utilizar?

Na verdade, alguma funções R podem utilizar os atributos específicos, por isso sua importância. Dependendo do tipo de análise e dados que quisermos armazenar, isso terá sua importância.

É possível observar se um objeto R possui atributos relacionados. Para isso utiliza-se o mesmo tipo de função usada anteriormente:

```
dados <- c(1, 2, 3, 4, 5, 6)
print(dados)
```

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

```
attributes(dados)
```

```
## NULL
```

Como o resultado é `NULL` sabemos que não há nenhum tipo de atributo relacionado ao objeto **dados**.

Agora, iremos observar que tipos de atributos podemos relacionar a um objeto R.

Os principais atributosd de um vetor atômico são:
> Names, Dimensions (dim), Classes

1.6.1 Names

Este é um atributo que pode ser passado a vetores atômicos. Para passar os nomes como atributos, basta passar o objeto R como um argumento à função `names`. Para observar se está tudo O.K. utilize a função `attributes`.

```
# Vamos atribuir a objetos R, nomes
# Esse é o atributo name em R
dados <- c(1, 2, 3, 4, 5, 6)
names(dados) <- c('um', 'dois', 'tres', 'quatro', 'cinco', 'seis')

# Observando os atributos
attributes(dados)
```

```
## $names
## [1] "um"     "dois"   "tres"   "quatro" "cinco"  "seis"
```

```
print(dados)
```

```
##     um   dois   tres quatro  cinco   seis
##      1      2      3      4      5      6
```

```
# Somar
dados + 1
```

```
##     um   dois   tres quatro  cinco   seis
##      2      3      4      5      6      7
```

```
# Para encerrar atributos basta atribuir NULL a este objeto R
names(dados) <- NULL
attributes(dados)
```

```
## NULL
```

```
# Exemplo
eu <- c('Biologo', '34', 'Bioquímica', 'UFRN')
names(eu) <- c('Profissão', 'Idade', 'Área', 'Instituição')
print(eu)
```

```
##    Profissão        Idade         Área  Instituição
##    "Biologo"         "34" "Bioquímica"       "UFRN"
```

1.6.2 Dim

É possível transformar um vetor atômico em um ***array*** n-dimensional atribuindo-lhe (***attribute***!) dimensões com a função `dim`. Essa função cria o atributo específico para dimensões em objetos R.

O interessante é que sepode fornecer mais de duas dimensões (neste caso, **linha**, **coluna** e ***slices***). Vamos observar alguns exemplos.

```
dim(die) <- c(2, 3)  # Duas linhas, Três colunas
print(die)
```

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

```
dim(die) <- c(3, 2)
print(die)
```

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

```
# Criando um dado de três dimensões
# Seria um hypercube, e R mostrará cada um dos slices
dim(die) <- c(1, 2, 3)
print(die)
```

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

1.7 Matrices

Matrizes e R são estruturas de dados que armazenam

Matrizes armazenam valores em um arranjo **bidimensional**.

É possível utilizar um vetor atômico para construção de uma matriz. Para isso, define-se o número de linhas. No mesmo sentido, é possível observar que a distribuição dos dados na matriz se dá por coluna. Assim, se temos um dado de seis faces e definimos nossa matriz com 2 linhas, a primeira coluna receberá na linha 1 o número 1, na linha dois o número dois. Na segundo coluna encontraremos na linha 1 o número 3 e na linha 2 o número 4, e assim sucessivamente.

Para modificar este arranjo, basta utilizar a função `matrix` adicionando como argumento do parâmetro `byrow` o valor 'TRUE'. Isso garantirá que a distribuição dos dados na matriz será iniciada na linha.

```
m <- matrix(die, nrow = 2); print(m)
```

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

```r
m2 <- matrix(die, nrow = 3); print(m2)
```

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

```r
# Matriz com argumento TRUE para parametro byrow
m3 <- matrix(die, nrow = 2, byrow = TRUE); print(m3)
```

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

```r
m4 <- matrix(die, nrow = 3, byrow = TRUE); print(m4)
```

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

1.8 Arrays

Arrays (arranjos) em R são semelhantes às matrizes, porém, eles guardam arranjos ***n-dimensionais*** de dados.
Desta forma, um array pode conter mais de duas dimensões. Para criar um array é preciso indicar um vetor atômico para entrada de dados. A sua funcionalidade é basicamente a mesma do atributo `dim` que foi visto anteriormente.

Para criar um *array* é preciso fornecer um vetor atômico como primeiro argumento, e um vetor de dimensões como segundo argumento. Este último é agora chamado de `dim`.

```r
# Criando um array
# ar será um array 2 x 2 x 3 (duas linhas, duas colunas, em três slices)
ar <- array(c(11:14, 21:24, 31:34), dim = c(2, 2, 3)); print(ar)
```

```
## , , 1
##
##      [,1] [,2]
## [1,]   11   13
## [2,]   12   14
##
## , , 2
##
##      [,1] [,2]
## [1,]   21   23
## [2,]   22   24
##
## , , 3
##
##      [,1] [,2]
## [1,]   31   33
## [2,]   32   34
```

1.9 Factors

```
  # Factors
gender <- factor(c('feminino', 'masculino', 'feminino', 'feminino', 'feminino', 'feminino'))
typeof(gender)
```

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

```
class(gender)
```

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

```
attributes(gender)
```

```
## $levels
## [1] "feminino"  "masculino"
##
## $class
## [1] "factor"
```

```
# Como R armazena o factor
unclass(gender)
```

```
## [1] 1 2 1 1 1 1
## attr(,"levels")
## [1] "feminino"  "masculino"
```

R mostrará o primeiro nível como sendo 1 (feminino) e o segundo nível será 2 (masculino).

Como é visto em `dico`, abaixo, há mais de dois fatores, por isso os níveis irão aumentar também.

```
# Quando se tem factors envolvidos na tabela, matriz, data.frame
# observar como estes estão armazenados (1 ou 2 ?) usando unclass
dico <- factor(c('pétalas', 'sépalas', 'tronco', 'frutos', 'inflorescencia'))
attributes(dico)
```

```
## $levels
## [1] "frutos"         "inflorescencia" "pétalas"        "sépalas"
## [5] "tronco"
##
## $class
## [1] "factor"
```

```
class(dico)
```

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

```
unclass(dico)
```

```
## [1] 3 4 5 1 2
## attr(,"levels")
## [1] "frutos"         "inflorescencia" "pétalas"        "sépalas"
## [5] "tronco"
```

Fatores tornam uma tarefa fácil colocar variáveis categóricas em modelos estatísticos devido às variáveis serem codificadas como números. Porém, a complicação de fatores é que estes confundem por se assemelharem a caracteres (*strings*), mas se comportarem como inteiros.

R tentará converter *strings* de caracteres a fatores quando os dados forem abertos ou criados no ambiente do interpretador. No geral, a experiência será melhor se não se permitir ao R fazer fatores até que se peça ao mesmo que seja feito.

Assim como outros tipos em R, pode-se transformar fatores em *strings* de caracteres com a função `as.character`.

```
char <- as.character(dico)
typeof(char)
```

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

```
class(char)
```

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

```
attributes(char)
```

```
## NULL
```

```
unclass(char)
```

```
## [1] "pétalas"        "sépalas"        "tronco"         "frutos"
## [5] "inflorescencia"
```

Exercício, p.51

```
## Exercício, p.51
# Faça uma jogada de cartas virtuais por combinar
# 'as', 'kopas' e 1 em um vetor
# Que tipo de vetor atômico surgirá?
cards <- c('as', 'kopas', 1)
typeof(cards); attributes(cards); class(cards); unclass(cards)
```

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

```
## NULL
```

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

```
## [1] "as"    "kopas" "1"
```

```
# Uma vez que o interpretador R não suporta vetores atômicos com mais de um tipo de dado
# R, por coerção, transformará todos os tipos de dados dentro do vetor cards
# em tipo character, classe character.
unclass(cards)
```

```
## [1] "as"    "kopas" "1"
```

Comparar:

```
  # Factors
gender <- factor(c('feminino', 'masculino', 'feminino', 'feminino', 'feminino', 'feminino'))
typeof(gender)
```

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

```
class(gender)
```

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

```
attributes(gender)
```

```
## $levels
## [1] "feminino"  "masculino"
##
## $class
## [1] "factor"
```

```r
# Como R armazena o factor
unclass(gender)
```

```
## [1] 1 2 1 1 1 1
## attr(,"levels")
## [1] "feminino"  "masculino"
```

> Se você quiser colocar múltiplos tipos de dados em um vetor, R converterár os elementos a simples tipos de dados.

Isso cria problemas, pois muitos conjuntos de dados utilizados para análises (*datasets*) contém múltiplos tipos de dados

1.10 Coercion

Aqui entrará o conceito de coerção em R. Muitos dados diferentes serão convertidos a um só tipo de dado por coerção, no interpretador R. Esse comportamento de R parece ser inconveniente, porém não o é. Coisas úteis podem surgir a partir da coerção.

Então, como R faz a coerção de tipos de dados?
- Se uma *string* de caractere está presente no vetor atômico, R converterá tudo o mais em um vetor para *strings* de caracteres.
- Se um vetor contém lógicos e números, R converterá os lógicos a números.
- Todo `TRUE` será convertido a 1; todo `FALSE` tornará-se um 0.

Resumindo, tudo será convertido a um caractere se caractere estiver presente. Lógicos serão convertidos a números, se aqueles forem colocados junto a números no vetor.

Para transformar um tipo em outro, basta usar a velha função `as`:

```r
as.character(1)
```

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

```r
as.character(0)
```

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

```r
as.logical(1)
```

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

```r
as.logical(0)
```

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

```r
as.numeric(TRUE)
```

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

```r
as.numeric(FALSE)
```

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

Existe um tipo de objeto em R que aceita diferentes tipos de dados, sem a necessidade de construirmos vetores separados para cada tipo, ou colocar tudo em um mesmo vetor e este acaber sendo modificado por coerção?

> Existe um tipo de objeto R que aceita diferentes tipos de dados e evita coerção, a lista.

1.11 Listas

A primeira informação sobre listas é que ela **unidiemensional**, assim como os vetores atômicos.
Então por que não usar vetores? Na verdade o comportamento de listas é diferente. Este é um tipo de objeto que não agrupa valores individuais, mas sim **objetos**. Então, uma lista pode conter vetores atomicos e até outras listas.

Exemplo do livro, ista contendo três objetos:

1. vetor numérico com 31 elementos (primeiro elemento da lista);

2. vetor caracteres contendo um único elemento (segundo elemento da lista);

3. lista contendo dois elementos (terceiro elemento da lista);

```
list1 <- list(100:130, "R", list(TRUE, FALSE))
print(list1)
```

```
## [[1]]
##  [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
## [18] 117 118 119 120 121 122 123 124 125 126 127 128 129 130
##
## [[2]]
## [1] "R"
##
## [[3]]
## [[3]][[1]]
## [1] TRUE
##
## [[3]][[2]]
## [1] FALSE
```

Colchetes duplos indicam que elemento da lista está sendo mostrado. E cada elemento terá seu sub-elemento dentro de sua demonstração de elemento. Eles estão abaixo do elmento. Por exemplo: [[3]] é o terceiro elemento da lista, e abaixo dele serão demonstrados os sub-elemtnos [1] e [2]. Note que cada sub-elemento que está sendo mostrado possui - acima dele - numeração entre colchetes duplos (no nosso caso, [[1]] para elemento 1 do primeiro sub-elmento de [[3]]).

```
lista <- list(c("Washington", "Candeia"), list(1:10), factor(c("H", "H", "F")))

typeof(lista); class(lista); attributes(lista); unclass(lista)
```

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

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

```
## NULL
```

```
## [[1]]
## [1] "Washington" "Candeia"
##
## [[2]]
## [[2]][[1]]
##  [1]  1  2  3  4  5  6  7  8  9 10
##
##
## [[3]]
## [1] H H F
```

```
## Levels: F H
```

Uso de listas em R pode ser um complicador se não houver critérios para sua criação. Porém, elas são flexibilidade faz deste objeto uma ferramenta de armazenamento útil para armazenamento em R, pois é possível armazenar qualquer coisa com uma lista.

1.12 Data Frames

Cartas de um baralho (*deck of cards*).

Data.frame é a versão bidimensional de uma lista. Proveem uma maneira excelente de fornecer uma forma de armazenar um baralho (cartas em um baralho). Na verdade, os data.frames em R são o que seriam as planilhas do Excel.

Os dados são armazenados em colunas, e cada coluna pode conter um tipo diferente de dado (numérico, character, lógico). Porém, é preciso ficar atento na construção de um data.frame, pois todas as colunas devem ter o mesmo comprimento.

> Cada coluna em um data.frame de R deve conter o mesmo comprimento.

Para criar um data.frame há a função `data.frame`.

```
# Usando argumentos nomeados para criar dados em data.frame
# O nome do argumento será o nome da coluna.
# O que está contido no argumento nomeado fará parte das colunas
df <- data.frame(face = c("ace", "two", "six"),
                 suit = c("clubs" ,"clubs", "clubs" ),
                 value = c(1, 2, 3))

str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ face : Factor w/ 3 levels "ace","six","two": 1 3 2
##  $ suit : Factor w/ 1 level "clubs": 1 1 1
##  $ value: num  1 2 3
```

É possível observar no **painel de ambiente, história, connection, e git** do Rstudio que o ojbeto `df` criado (um data.frame) possui três tipos de objetos agrupados: **face, suit** e **value**. Para observar no **console** pode-se utilizar a função `str`.

```
# Tipo de df:
typeof(df)
```

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

```
# "Usando função str para ver o conteúdo de df:
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ face : Factor w/ 3 levels "ace","six","two": 1 3 2
##  $ suit : Factor w/ 1 level "clubs": 1 1 1
##  $ value: num  1 2 3
```

```
print(df)
```

```
##   face  suit value
## 1  ace clubs     1
## 2  two clubs     2
## 3  six clubs     3
```

Data.frames são objetos do tipo lista, mas pertencentes à classe "data.frame".

Observe que todos os caracteres no data.frame (argumentos `face` e `suit`, respectivamente) foram, por coerção, passados ao tipo character.

É possível evitar que ocorra coerção e os tipos sejam modificados em um data.frame R. Para isso, basta utilizar um argumento `stringsAsfactors` como `FALSE`.

```
# Exemplo de data frame sem coerção de tipos
df <- data.frame(face = c("ace", "two", "six"),
                 suit = c("clubs", "clubs", "clubs"),
                 value = c(1, 2, 3),
                 stringsAsFactors = FALSE)

# Observando modificações
print(df)
```

```
##   face  suit value
## 1  ace clubs     1
## 2  two clubs     2
## 3  six clubs     3
```

```
# str
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ face : chr  "ace" "two" "six"
##  $ suit : chr  "clubs" "clubs" "clubs"
##  $ value: num  1 2 3
```

Note que agora, os objetos face e suit estão como *chr* de characters. Não são mais factors.

1.13 Carregando Dados

CSV - Arquivos de texto simples onde os dados estão separados por vírgulas.

```
#head(deck)
```

1.14 Salvando Dados

Salvar uma cópia do arquivo CSV. Usar a função `write.csv`.

```
# Salvar uma cópia
#write.csv(deck, file = "cards.csv", row.names = FALSE)
```

Por que usar o argumento `row.names = FALSE`?
Para prevenir R de adicionar mais uma coluna no início do data.frame.
Essa coluna conteria números, para numerar as linhas conforme o seu número. Por isso row.names. Isso acarretaria problemas ao abrir este arquivo, pois nem mesmo o R poderia entender o que seria essa primeira coluna. Ele entenderia que essa coluna seria a primeira coluna do data.frame, o que acarretaria em problemas nas futuras análises.