Aula 4: Listas e Data Frames

Profa. Yana Borges

Fevereiro de 2022

Os vetores, matrizes e arrays que vimos até agora contêm elementos que são todos do mesmo tipo. Listas e data frames são dois tipos que nos permitem combinar diferentes tipos de dados em uma única variável.

1. Listas

Uma lista é, grosso modo, um vetor onde cada elemento pode ser de um tipo diferente. Esta seção trata de como criar, indexar e manipular listas.

1.1 Criando listas

As listas são criadas com a função list, e especificar o conteúdo funciona de maneira muito parecida com a função `c` que já vimos. Você simplesmente lista o conteúdo, com cada argumento separado por uma vírgula. Os elementos da lista podem ser de qualquer tipo de variável — vetores, matrizes e até funções:

```
(a_list <- list(
c(1, 1, 2, 5, 14, 42),
month.abb,
matrix(c(3, -8, 1, -3), nrow = 2),
asin
))
```

```
## [[1]]
## [1]  1  1  2  5 14 42
##
## [[2]]
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
##
## [[3]]
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## [[4]]
## function (x)  .Primitive("asin")
```

Tal como acontece com os vetores, você pode nomear os elementos durante a construção ou depois usando a função `names`:

```
names(a_list) <- c("catalan", "months", "involutary", "arcsin")
a_list
```

```
## $catalan
## [1]  1  1  2  5 14 42
##
```

```
## $months
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
##
## $involutary
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## $arcsin
## function (x)  .Primitive("asin")
```

```
(the_same_list <- list(
catalan = c(1, 1, 2, 5, 14, 42),
months = month.abb,
involutary = matrix(c(3, -8, 1, -3), nrow = 2),
arcsin = asin
))
```

```
## $catalan
## [1]  1  1  2  5 14 42
##
## $months
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
##
## $involutary
##      [,1] [,2]
## [1,]    3    1
## [2,]   -8   -3
##
## $arcsin
## function (x)  .Primitive("asin")
```

Não é obrigatório, mas ajuda se os nomes que você dá aos elementos são nomes de variáveis válidos.

É até possível que elementos de listas sejam eles próprios listas:

```
(main_list <- list( # lista principal
middle_list = list( # lista intermediária
element_in_middle_list = diag(3), #elemento de "middle_list
inner_list = list( #lista interna
element_in_inner_list = pi ^ 1:4, #elemento de inner_list
another_element_in_inner_list = "a"
)
),
element_in_main_list = log10(1:10)
))
```

```
## $middle_list
## $middle_list$element_in_middle_list
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
##
## $middle_list$inner_list
## $middle_list$inner_list$element_in_inner_list
## [1] 3.141593
```

```
##
## $middle_list$inner_list$another_element_in_inner_list
## [1] "a"
##
##
##
## $element_in_main_list
##  [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513 0.8450980
##  [8] 0.9030900 0.9542425 1.0000000
```

Em teoria, você pode manter listas de aninhamento para sempre. Na prática, as versões atuais do R lançarão um erro quando você começar a aninhar suas listas com dezenas de milhares de níveis de profundidade (o número exato é específico da máquina). Felizmente, isso não deve ser um problema para você, já que o código do mundo real onde o aninhamento é mais profundo do que três ou quatro níveis é extremamente raro.

1.2 Comprimento e operações matemáticas com listas

Assim como os vetores, as listas têm um comprimento. O comprimento de uma lista é o número de elementos de nível superior que ela contém:

```
length(a_list)
```

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

```
length(main_list) #não inclui os comprimentos das listas aninhadas
```

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

Como vetores, mas diferentemente de matrizes, as listas não têm dimensões. A função dim correspondentemente retorna `NULL`:

```
dim(a_list)
```

```
## NULL
```

`nrow`, `NROW` e as funções de coluna correspondentes funcionam em listas da mesma forma que em vetores:

```
nrow(a_list)
```

```
## NULL
```

```
ncol(a_list)
```

```
## NULL
```

```
NROW(a_list)
```

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

```
NCOL(a_list)
```

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

Ao contrário dos vetores, a aritmética não funciona em listas. Como cada elemento pode ser de um tipo diferente, não faz sentido poder somar ou multiplicar duas listas. É possível fazer aritmética em elementos de lista, no entanto, assumindo que eles são de um tipo apropriado. Nesse caso, aplicam-se as regras usuais para o conteúdo do elemento. Por exemplo:

```
(l1 <- list(1:5))
```

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

```
(l2 <- list(6:10))
```

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

```
l1[[1]] + l2[[1]]
```

```
## [1]  7  9 11 13 15
```

1.3 Indexando listas

Assim como nos vetores, podemos acessar os elementos da lista usando colchetes, `[]`, índices numéricos positivos ou negativos, nomes de elementos ou um índice lógico. As quatro linhas de código a seguir fornecem o mesmo resultado:

```
l <- list(
first = 1,
second = 2,
third = list(
alpha = 3.1,
beta = 3.2
)
)
l[1:2]
```

```
## $first
## [1] 1
##
## $second
## [1] 2
```

```
l[-1]
```

```
## $second
## [1] 2
##
## $third
## $third$alpha
## [1] 3.1
##
## $third$beta
## [1] 3.2
```

```
l[c("first", "second")]
```

```
## $first
## [1] 1
##
## $second
## [1] 2
```

```
l[c(TRUE, TRUE, FALSE)]
```

```
## $first
## [1] 1
##
## $second
## [1] 2
```

O resultado dessas operações de indexação é outra lista. Às vezes, queremos acessar o conteúdo dos elementos da lista. Existem dois operadores para nos ajudar a fazer isso. Colchetes duplos (`[[]]`) podem receber um único inteiro positivo indicando o índice a ser retornado ou uma única string nomeando esse elemento:

```
l[[1]]
```

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

```
l[["first"]]
```

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

Para elementos nomeados de listas, também podemos usar o operador cifrão, `$`:

```
l$first
```

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

Para acessar elementos aninhados, podemos empilhar os colchetes ou passar um vetor, embora o último método seja menos comum e geralmente mais difícil de ler:

```
l[["third"]]["beta"]
```

```
## $beta
## [1] 3.2
```

```
l[["third"]][["beta"]]
```

```
## [1] 3.2
```

```
l[[c("third", "beta")]]
```

```
## [1] 3.2
```

O comportamento ao tentar acessar um elemento inexistente de uma lista varia dependendo do tipo de indexação que você usou. Para o próximo exemplo, lembre-se de que nossa lista, `l`, tem apenas três elementos.

Se usarmos a indexação de colchetes simples, a lista resultante terá um elemento com o valor `NULL` (e nome `NA`, se a lista original tiver nomes). Compare isso com a indexação ruim de um vetor em que o valor de retorno é `NA`:

```
l[c(4, 2, 5)]
```

```
## $<NA>
## NULL
##
## $second
## [1] 2
##
## $<NA>
## NULL
```

```
l[c("fourth", "second", "fifth")]
```

```
## $<NA>
## NULL
##
## $second
## [1] 2
##
## $<NA>
## NULL
```

Tentar acessar o conteúdo de um elemento com um nome incorreto, seja com colchetes duplos ou um cifrão, retorna `NULL`:

```
l[["fourth"]]
```

```
## NULL
```

```
l$fourth
```

```
## NULL
```

Finalmente, tentar acessar o conteúdo de um elemento com um índice numérico incorreto gera um erro, informando que o subscrito está fora dos limites. Essa inconsistência de comportamento é algo que você só precisa aceitar, embora a melhor defesa seja certificar-se de verificar seus índices antes de usá-los:

```
l[[4]] #gera erro
```

```
## Error in l[[4]]: índice fora de limites
```

1.4 Conversão de vetores em listas

Vetores podem ser convertidos em listas usando a função `as.list`. Isso cria uma lista com cada elemento do mapeamento vetorial para um elemento de lista contendo um valor:

```
busy_beaver <- c(1, 6, 21, 107) #See http://oeis.org/A060843
as.list(busy_beaver)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 6
##
## [[3]]
## [1] 21
##
## [[4]]
## [1] 107
```

Se cada elemento da lista contiver um valor escalar, também é possível converter essa lista em um vetor usando as funções que já vimos (`as.numeric`, `as.character` e assim por diante):

```
as.numeric(list(1, 6, 21, 107))
```

```
## [1]   1   6  21 107
```

Essa técnica não funcionará nos casos em que a lista contém elementos não escalares. Este é um problema real, pois além de armazenar diferentes tipos de dados, as listas são muito úteis para armazenar dados do mesmo tipo, mas com formato não retangular:

```
(prime_factors <- list(
two = 2,
three = 3,
four = c(2, 2),
five = 5,
six = c(2, 3),
seven = 7,
eight = c(2, 2, 2),
nine = c(3, 3),
```

```
ten = c(2, 5)
))
```

```
## $two
## [1] 2
##
## $three
## [1] 3
##
## $four
## [1] 2 2
##
## $five
## [1] 5
##
## $six
## [1] 2 3
##
## $seven
## [1] 7
##
## $eight
## [1] 2 2 2
##
## $nine
## [1] 3 3
##
## $ten
## [1] 2 5
```

Este tipo de lista pode ser convertido em um vetor usando a função unlist (às vezes é tecnicamente possível fazer isso com listas de tipo misto, mas raramente útil):

```
unlist(prime_factors)
```

```
##    two  three  four1  four2   five   six1   six2  seven eight1 eight2 eight3
##      2      3      2      2      5      2      3      7      2      2      2
##  nine1  nine2   ten1   ten2
##      3      3      2      5
```

1.5 Combinando listas

A função `c` que usamos para concatenar vetores também funciona para concatenar listas:

```
c(list(a = 1, b = 2), list(3))
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## [[3]]
## [1] 3
```

Se o usarmos para concatenar listas e vetores, os vetores serão convertidos em listas (como se `as.list` tivesse

sido chamado) antes que a concatenação ocorra:

```
c(list(a = 1, b = 2), 3)
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## [[3]]
## [1] 3
```

Também é possível usar as funções `cbind` e `rbind` em listas, mas os objetos resultantes são realmente muito estranhos. São matrizes com elementos possivelmente não escalares ou listas com dimensões, dependendo de como você deseja vê-las:

```
(matrix_list_hybrid <- cbind(
list(a = 1, b = 2),
list(c = 3, list(d = 4))
))
```

```
##   [,1] [,2]
## a 1    3
## b 2    List,1
```

```
str(matrix_list_hybrid)
```

```
## List of 4
##  $ : num 1
##  $ : num 2
##  $ : num 3
##  $ :List of 1
##   ..$ d: num 4
##  - attr(*, "dim")= int [1:2] 2 2
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:2] "a" "b"
##   ..$ : NULL
```

1.6 NULL

`NULL` é um valor especial que representa uma variável vazia. Seu uso mais comum é em listas, mas também surge em data frames e argumentos de função.

Ao criar uma lista, você pode especificar que um elemento deve existir, mas não deve ter conteúdo. Por exemplo, a lista a seguir contém feriados bancários do Reino Unido[1] para 2013 por mês. Alguns meses não têm feriados, então usamos `NULL` para representar essa ausência:

```
(uk_bank_holidays_2013 <- list(
Jan = "New Year's Day",
Feb = NULL,
Mar = "Good Friday",
Apr = "Easter Monday",
May = c("Early May Bank Holiday", "Spring Bank Holiday"),
Jun = NULL,
Jul = NULL,
```

[1] Feriados bancários são feriados.

```
Aug = "Summer Bank Holiday",
Sep = NULL,
Oct = NULL,
Nov = NULL,
Dec = c("Christmas Day", "Boxing Day")
))
```

```
## $Jan
## [1] "New Year's Day"
##
## $Feb
## NULL
##
## $Mar
## [1] "Good Friday"
##
## $Apr
## [1] "Easter Monday"
##
## $May
## [1] "Early May Bank Holiday" "Spring Bank Holiday"
##
## $Jun
## NULL
##
## $Jul
## NULL
##
## $Aug
## [1] "Summer Bank Holiday"
##
## $Sep
## NULL
##
## $Oct
## NULL
##
## $Nov
## NULL
##
## $Dec
## [1] "Christmas Day" "Boxing Day"
```

É importante entender a diferença entre `NULL` e o valor omisso especial `NA`. A maior diferença é que `NA` é um valor escalar, enquanto `NULL` não ocupa espaço - tem comprimento zero:

```
length(NULL)
```

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

```
length(NA)
```

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

Você pode testar NULL usando a função is.null. Os valores ausentes não são nulos:

```r
is.null(NULL)
```

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

```r
is.null(NA)
```

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

O teste inverso não faz muito sentido. Como NULL tem comprimento zero, não temos nada para testar para ver se está faltando:

```r
is.na(NULL)
```

```
## logical(0)
```

`NULL` também pode ser usado para remover elementos de uma lista. Definir um elemento como `NULL` (mesmo que já contenha `NULL`) o removerá. Suponha que, por algum motivo, queiramos mudar para um calendário romano antigo de 10 meses, removendo janeiro e fevereiro:

```r
uk_bank_holidays_2013$Jan <- NULL
uk_bank_holidays_2013$Feb <- NULL
uk_bank_holidays_2013
```

```
## $Mar
## [1] "Good Friday"
##
## $Apr
## [1] "Easter Monday"
##
## $May
## [1] "Early May Bank Holiday" "Spring Bank Holiday"
##
## $Jun
## NULL
##
## $Jul
## NULL
##
## $Aug
## [1] "Summer Bank Holiday"
##
## $Sep
## NULL
##
## $Oct
## NULL
##
## $Nov
## NULL
##
## $Dec
## [1] "Christmas Day" "Boxing Day"
```

Para definir um elemento existente como `NULL`, não podemos simplesmente atribuir o valor de `NULL`, pois isso removerá o elemento. Em vez disso, deve ser definido como list(`NULL`). Agora suponha que o governo do Reino Unido se torne malvado e cancele o feriado de verão:

```r
uk_bank_holidays_2013["Aug"] <- list(NULL)
uk_bank_holidays_2013
```

```
## $Mar
## [1] "Good Friday"
##
## $Apr
## [1] "Easter Monday"
##
## $May
## [1] "Early May Bank Holiday" "Spring Bank Holiday"
##
## $Jun
## NULL
##
## $Jul
## NULL
##
## $Aug
## NULL
##
## $Sep
## NULL
##
## $Oct
## NULL
##
## $Nov
## NULL
##
## $Dec
## [1] "Christmas Day" "Boxing Day"
```

2. Data Frames

Os data fames são usados para armazenar dados semelhantes a planilhas. Eles podem ser pensados como matrizes onde cada coluna pode armazenar um tipo diferente de dados, ou listas não aninhadas onde cada elemento tem o mesmo tamanho.

2.1 Criando Data Frames

Criamos data frames com a função `data.frame`:

```r
(a_data_frame <- data.frame(
x = letters[1:5],
y = rnorm(5),
z = runif(5) > 0.5
))
```

```
##   x          y     z
## 1 a  1.9494502 FALSE
## 2 b  0.5013712  TRUE
## 3 c  1.2508850 FALSE
## 4 d -0.1858202 FALSE
```

```
## 5 e  0.9193375 FALSE
```

```
class(a_data_frame)
```

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

Observe que cada coluna pode ter um tipo diferente das outras colunas, mas que todos os elementos dentro de uma coluna são do mesmo tipo. Observe também que a classe do objeto é `data.frame`, com um ponto em vez de um espaço.

Neste exemplo, as linhas foram numeradas automaticamente de um a cinco. Se qualquer um dos vetores de entrada tivesse nomes, os nomes das linhas teriam sido retirados do primeiro desses vetores. Por exemplo, se y tivesse nomes, eles seriam dados ao data frame:

```
y <- rnorm(5)
names(y) <- month.name[1:5]
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5
)
```

```
##          x          y     z
## January  a  0.2993542 FALSE
## February b  1.8063054 FALSE
## March    c -0.7044750 FALSE
## April    d -0.4665162  TRUE
## May      e -0.1408546 FALSE
```

Esse comportamento pode ser substituído passando o argumento `row.names = NULL` para a função `data.frame`:

```
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5,
row.names = NULL
)
```

```
##   x          y     z
## 1 a  0.2993542  TRUE
## 2 b  1.8063054 FALSE
## 3 c -0.7044750  TRUE
## 4 d -0.4665162 FALSE
## 5 e -0.1408546 FALSE
```

Também é possível fornecer seus próprios nomes de linha usando `row.names`. Este vetor será convertido em caractere, caso ainda não seja desse tipo:

```
data.frame(
x = letters[1:5],
y = y,
z = runif(5) > 0.5,
row.names = c("Jackie", "Tito", "Jermaine", "Marlon", "Michael")
)
```

```
##          x          y     z
## Jackie   a  0.2993542  TRUE
## Tito     b  1.8063054 FALSE
```

```
## Jermaine c -0.7044750 FALSE
## Marlon   d -0.4665162 FALSE
## Michael  e -0.1408546 FALSE
```

Os nomes de linha podem ser recuperados ou alterados posteriormente, da mesma maneira que com matrizes, usando `rownames` (ou `row.names`). Da mesma forma, `colnames` e `dimnames` podem ser usados para obter ou definir os nomes de coluna e dimensão, respectivamente. Na verdade, mais ou menos todas as funções que podem ser usadas para inspecionar matrizes também podem ser usadas com data frames. `nrow`, `ncol` e `dim` também funcionam exatamente da mesma maneira que em matrizes:

```
rownames(a_data_frame)
```

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

```
colnames(a_data_frame)
```

```
## [1] "x" "y" "z"
```

```
dimnames(a_data_frame)
```

```
## [[1]]
## [1] "1" "2" "3" "4" "5"
##
## [[2]]
## [1] "x" "y" "z"
```

```
nrow(a_data_frame)
```

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

```
ncol(a_data_frame)
```

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

```
dim(a_data_frame)
```

```
## [1] 5 3
```

Existem duas peculiaridades que você precisa estar ciente. Primeiro, `length` retorna o mesmo valor que `ncol`, não o número total de elementos no data frame. Da mesma forma, os nomes retornam o mesmo valor que os nomes das colunas. Para maior clareza do código, recomendo que você evite essas duas funções e use `ncol` e `colnames`:

```
length(a_data_frame)
```

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

```
names(a_data_frame)
```

```
## [1] "x" "y" "z"
```

É possível criar um data frame passando diferentes comprimentos de vetores, desde que os comprimentos permitam que os mais curtos sejam reciclados um número exato de vezes. Mais tecnicamente, o menor múltiplo comum de todos os comprimentos deve ser igual ao vetor mais longo:

```
data.frame( #comprimentos 1, 2 e 4 estão OK
x = 1, #reciclado 4 vezes
y = 2:3, #reciclado duas vezes
z = 4:7 #a entrada mais longa; sem reciclagem
)
```

```
##   x y z
```

```
## 1 1 2 4
## 2 1 3 5
## 3 1 2 6
## 4 1 3 7
```

Se os comprimentos não forem compatíveis, um erro será lançado:

```
data.frame( #lengths 1, 2 e 3 causam um erro
x = 1, #múltiplo comum mais baixo é 6, que é maior que 3
y = 2:3,
z = 4:6
)
```

```
## Error in data.frame(x = 1, y = 2:3, z = 4:6): arguments imply differing number of rows: 1, 2, 3
```

Uma outra consideração ao criar data frames é que, por padrão, os nomes das colunas são verificados como nomes de variáveis exclusivos e válidos. Este recurso pode ser desativado passando `check.names = FALSE` para `data.frame`:

```
data.frame(
"A column" = letters[1:5],
"!@#$%^&*()" = rnorm(5),
"..." = runif(5) > 0.5,
check.names = FALSE
)
```

```
##   A column !@#$%^&*()   ...
## 1        a -0.7521886 FALSE
## 2        b  0.5695999 FALSE
## 3        c -1.0407632  TRUE
## 4        d -0.6836566  TRUE
## 5        e  0.4342610 FALSE
```

Em geral, ter nomes de colunas fora do padrão é uma má ideia. A duplicação de nomes de colunas é ainda pior, pois pode levar a bugs difíceis de encontrar quando você começa a usar subconjuntos. Desative a verificação do nome da coluna por sua conta e risco.

2.2 Indexando Data Frames

Há muitas maneiras diferentes de indexar um data frame. Para começar, pares de quatro índices vetoriais diferentes (inteiros positivos, inteiros negativos, valores lógicos e caracteres) podem ser usados exatamente da mesma maneira que com matrizes. Esses comandos selecionam o segundo e o terceiro elementos das duas primeiras colunas:

```
a_data_frame[2:3, -3]
```

```
##   x         y
## 2 b 0.5013712
## 3 c 1.2508850
```

```
a_data_frame[c(FALSE, TRUE, TRUE, FALSE, FALSE), c("x", "y")]
```

```
##   x         y
## 2 b 0.5013712
## 3 c 1.2508850
```

Como mais de uma coluna foi selecionada, o subconjunto resultante também é um data frame. Se apenas uma coluna tivesse sido selecionada, o resultado teria sido simplificado para ser um vetor:

```r
class(a_data_frame[2:3, -3])
```

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

```r
class(a_data_frame[2:3, 1])
```

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

Se quisermos selecionar apenas uma coluna, a indexação em estilo de lista (colchetes duplos com um inteiro ou nome positivo, ou o operador cifrão com um nome) também pode ser usada. Todos esses comandos selecionam o segundo e o terceiro elementos da primeira coluna:

```r
a_data_frame$x[2:3]
```

```
## [1] "b" "c"
```

```r
a_data_frame[[1]][2:3]
```

```
## [1] "b" "c"
```

```r
a_data_frame[["x"]][2:3]
```

```
## [1] "b" "c"
```

2.3 Manipulação básica de Data Frames

Assim como as matrizes, os data frames podem ser transpostos usando a função `t`, mas no processo todas as colunas (que se tornam linhas) são convertidas para o mesmo tipo, e tudo se torna uma matriz:

```r
t(a_data_frame)
```

```
##   [,1]         [,2]         [,3]         [,4]         [,5]
## x "a"          "b"          "c"          "d"          "e"
## y " 1.9494502" " 0.5013712" " 1.2508850" "-0.1858202" " 0.9193375"
## z "FALSE"      "TRUE"       "FALSE"      "FALSE"      "FALSE"
```

Os data frames também podem ser unidos usando `cbind` e `rbind`, supondo que eles tenham os tamanhos apropriados. `rbind` é inteligente o suficiente para reordenar as colunas para corresponder. O `cbind` não verifica os nomes das colunas para duplicatas, portanto, tenha cuidado com isso:

```r
another_data_frame <- data.frame( #mesmas colunas que a_data_frame, ordem diferente
z = rlnorm(5), #log números distribuídos normalmente
y = sample(5), #os números de 1 a 5, em alguma ordem
x = letters[3:7]
)
rbind(a_data_frame, another_data_frame)
```

```
##    x          y         z
## 1  a  1.9494502 0.0000000
## 2  b  0.5013712 1.0000000
## 3  c  1.2508850 0.0000000
## 4  d -0.1858202 0.0000000
## 5  e  0.9193375 0.0000000
## 6  c  5.0000000 0.4194638
## 7  d  3.0000000 1.2156717
## 8  e  4.0000000 3.4988565
## 9  f  1.0000000 0.2502886
## 10 g  2.0000000 4.0348502
```

```r
cbind(a_data_frame, another_data_frame)
```

```
##   x          y     z         z y x
## 1 a  1.9494502 FALSE 0.4194638 5 c
## 2 b  0.5013712  TRUE 1.2156717 3 d
## 3 c  1.2508850 FALSE 3.4988565 4 e
## 4 d -0.1858202 FALSE 0.2502886 1 f
## 5 e  0.9193375 FALSE 4.0348502 2 g
```

Onde dois data frames compartilham colunas, eles podem ser mesclados usando a função `merge`. `merge` fornece uma variedade de opções para fazer junções no estilo de banco de dados. Para unir dois data frames, você precisa especificar quais colunas contêm os valores chave para corresponder. Por padrão, a função `merge` usa todas as colunas comuns dos dois data frames, mas mais comumente você desejará usar apenas uma única coluna de ID compartilhada. Nos exemplos a seguir, especificamos que a coluna `x` contém nossos IDs usando o argumento `by`:

```r
merge(a_data_frame, another_data_frame, by = "x")
```

```
##   x        y.x   z.x       z.y y.y
## 1 c  1.2508850 FALSE 0.4194638   5
## 2 d -0.1858202 FALSE 1.2156717   3
## 3 e  0.9193375 FALSE 3.4988565   4
```

```r
merge(a_data_frame, another_data_frame, by = "x", all = TRUE)
```

```
##   x        y.x   z.x       z.y y.y
## 1 a  1.9494502 FALSE        NA  NA
## 2 b  0.5013712  TRUE        NA  NA
## 3 c  1.2508850 FALSE 0.4194638   5
## 4 d -0.1858202 FALSE 1.2156717   3
## 5 e  0.9193375 FALSE 3.4988565   4
## 6 f         NA    NA 0.2502886   1
## 7 g         NA    NA 4.0348502   2
```

Onde um data frame tem todos os valores numéricos, as funções `colSums` e `colMeans` podem ser usadas para calcular as somas e médias de cada coluna, respectivamente. Da mesma forma, `rowSums` e `rowMeans` calculam as somas e médias de cada linha:

```r
colSums(a_data_frame[, 2:3])
```

```
##        y        z
## 4.435224 1.000000
```

```r
colMeans(a_data_frame[, 2:3])
```

```
##         y         z
## 0.8870447 0.2000000
```

3. Resumo

- As listas podem conter diferentes tamanhos e tipos de variáveis em cada elemento.
- As listas são variáveis recursivas, pois podem conter outras listas.
- Você pode indexar listas usando `[]`, `[[]]` ou `$`.
- `NULL` é um valor especial que pode ser usado para criar elementos de lista "vazios".
- Data frames armazenam dados semelhantes a planilhas.
- Data frames têm algumas propriedades de matrizes (são retangulares) e algumas listas (diferentes colunas podem conter diferentes tipos de variáveis).

- Os data frames podem ser indexados como matrizes ou listas semelhantes.
- A função `merge` permite que você faça junções no estilo de banco de dados em data frames.

Exercícios

Exercício 1

Crie uma variável do tipo lista com três elementos contendo: os sete primeiros números primos; os quatro primeiros pares de números primos gêmeos; números primos pares diferentes de 2. Dê nome aos três elementos.

Exercício 2

O R vem com vários conjuntos de dados integrados, incluindo os famosos dados de íris (flores, não olhos), coletados por Anderson e analisados por Fisher na década de 1930. Digite `iris` para ver o conjunto de dados. Crie um novo data frame chamado "petalas", extraído do conjunto de dados `iris`, com a terceira e quarta colunas; com as seis primeiras e seis últimas linhas. Renomeie as colunas do objeto "petalas" por "comprimento" e "largura", respectivamente. Calcule a média das variáves "comprimento" e "largura".

Exercício 3

Os conjuntos de dados `beaver1` e `beaver2` contêm temperaturas corporais de dois castores.

a. Crie dois objetos contendo as três primeiras linhas de `beaver1` e `beaver2`, respectivamente. Faça a mesclagem dos dois objetos usando a chave `activ` e guarde em `castor`. Renomeie as colunas de `castor` com letras do alfabeto, a partir da segunda coluna.

b. Crie uma lista com os elementos `petalas` e `castor`, obtidos nos exercícios 1 e 2. Assegure-se de que os elementos da lista recebam os nomes de `petalas` e `castor`, respectivamente.

c. A partir da lista criada no exercício 3 b, crie um vetor contendo o quinto elemento da coluna `comprimento` do elemento `petalas` e o oitavo elemento da penúltima coluna do elemento `castor`.