Linguagem R de programação Estatística

Professor Eduardo Monteiro de Castro Gomes

Este material tem como objetivo apresentar alguns conceitos fundamentais da linguagem R, suas estruturas de dados e controle e indexação de objetos.

A forma mais simples de utilização do R é como uma calculadora para realizar operações básicas como:

```
2 + 2
```

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

ou

```
1348.75 / 7
```

```
## [1] 192.6786
```

e conjuntos de operações

```
(783 - 139)^2 / 5
```

```
## [1] 82947.2
```

Essa utilização simplória começa a se tornar vantajosa, em relação a uma simples calculadora, a partir do momento em que é possível armazenar o resultado das operações realizadas para reutilização em outras operações.

No exemplo seguinte temos o valor de um determinado produto, 750 reais, ao qual será aplicado um desconto, 12%, e o valor final calculado e salvo em um objeto chamado *preco_final*.

```
preco_final <- 750 * .88
preco_final
```

```
## [1] 660
```

Para fazer a atribuição de um objeto utiliza-se "<-" determinando o valor no lado esquerdo como nome do objeto representado no lado direito. É importante notar que a linguagem faz distinção entre letras minúsculas e maiúsculas. Os nomes podem ser formados por letras, números, "_" e ".", devendo ser iniciados por letras ou por pontos desde que não sejam seguidos por números.

```
nome_valido1 <- 10
.outro_valido <- 10
```

Voltando ao exemplo do produto com desconto, suponha que tem-se o interesse em parcelar o pagamento do produto em 3 parcelas e pode-se utilizar o nome do objeto em que foi armazenado o preço final do produto com o desconto para calcular o valor de cada parcela.

```
preco_final / 3
```

```
## [1] 220
```

Vetores

É natural considerar que as operações que serão realizadas com auxílio da linguagem R consideram grandes números de observações. Uma primeira extensão que vamos considerar aos objetos númericos que vimos no exemplo anterior é pela utilização de vetores que permitem armazenar conjuntos de valores.

A função para criação dos vetores é *c()* em que a letra c é utilizada como abreviação de concatenação que será realizada entre os objetos para serem agrupados em um único objeto.

```
precos <- c(750, 822, 300, 15)
precos
```

```
## [1] 750 822 300  15
```

e temos assim em um único objeto, chamado precos, um vetor que contem os valores 750, 822, 300 e 15. Note que a função *c()* pode fazer tambem a concatenação entre vetores.

```
vetor1 <- c(1,2,3,4)
vetor2 <- c(100,200,300)
vet_concatenado <- c(vetor1,vetor2)
vet_concatenado
```

```
## [1]   1   2   3   4 100 200 300
```

e **vet_concatenado** é o vetor resultante da concatenação entre os vetores *vetor1* e *vetor2*.

Funções úteis para criação de vetores numéricos

Algumas funções comumente utilizadas para criação de vetores com valores numéricos são exemplificadas a seguir:

```
1:10
```

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

```
10:1
```

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

```
-5:3
```

```
## [1] -5 -4 -3 -2 -1  0  1  2  3
```

```
1.5:8
```

```
## [1] 1.5 2.5 3.5 4.5 5.5 6.5 7.5
```

o operador **:** cria sequências de números com início no valor à esquerda incrementando ou decrementando uma unidade até um limite determinado pelo valor à direita do operador.

A função *seq* tem comportamento semelhante mas permite a determinação do tamanho do incremento ou decremento, ou a determinação do números de elementos igualmente espaçados desejados dentro dos limites determinados, conforme os exemplos:

```
seq(from = 1, to = 10, by = .5)
```

```
##  [1]  1.0  1.5  2.0  2.5  3.0  3.5  4.0  4.5  5.0  5.5  6.0  6.5  7.0  7.5
## [15]  8.0  8.5  9.0  9.5 10.0
```

```
seq(from = 1, to = 10, length.out = 25)
```

```
##  [1]  1.000  1.375  1.750  2.125  2.500  2.875  3.250  3.625  4.000  4.375
## [11]  4.750  5.125  5.500  5.875  6.250  6.625  7.000  7.375  7.750  8.125
## [21]  8.500  8.875  9.250  9.625 10.000
```

A função *rep* permite criar um vetor a partir da repetição de um elemento ou vetor, permitindo determinar o número de repetições para cada elemento de forma sequencial ou não.

```
rep(c(1,2,3), times = 3)
```

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

```
rep(c(10,20,30), each = 3)
```

```
## [1] 10 10 10 20 20 20 30 30 30
```

```
rep(c(10,20,30), times = c(1,2,3))
```

```
## [1] 10 20 20 30 30 30
```

É possível gerar números aleatórios a partir de diferentes distribuições de probabilidade. As funções para geração de números aleatórios tem seus nomes formados pela letra *r*, de random, seguido pelo nome ou abreviação do nome da distribuição de probabilidade. A seguir são apresentados exemplos de números gerados pelas distribuições Normal e Poisson.

```
rnorm(n = 10, mean = 100, sd = 25)
```

```
##  [1] 130.32061  73.79323 111.67864  62.44913 104.88670 126.37586  86.91017
##  [8]  89.29094 121.28904 161.35302
```

```r
rpois(n = 15, lambda = 500 )
```

```
##  [1] 504 500 518 496 470 485 501 510 476 504 467 497 481 483 517
```

Tipos de elementos

Até este ponto do texto foram considerados apenas exemplos em que as variáveis utilizadas foram todas numéricas, mas é bastante comum em linguagens de programação que se trabalhe com outros tipos de variáveis. Estaremos interessados aqui, alem das variáveis númericas, nas variáveis do tipo lógico e do tipo caracter.

- Variáveis lógicas

As variáveis do tipo lógico podem assumir apenas dois valores: TRUE e FALSE, representando verdadeiro e falso. Esse tipo de variável lógica será de grande importância e é resultante principalmente de operações de comparação entre elementos do tipo:

```
3 > 5
```

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

```
3 < 5
```

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

```
3 >= 5
```

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

```
3 <= 5
```

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

```
3 == 5
```

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

```
3 == 3
```

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

Um vetor de elementos lógicos pode ser definido pela criação dos elementos conforme o exemplo a seguir:

```
novo_vetor <- c(TRUE, T, FALSE, F)
```

note que a criação de elementos lógicos com valor verdadeiro pode ser feita utilizando a palavra completa *TRUE* ou apenas a primeira letra *T*, sempre com letras maiúsculas. Os elementos com valor falso, de forma análoga podem ser criados com *FALSE* ou apenas *F*.

E vetores com elementos lógicos podem ser criados apartir de operações de comparação entre vetores

```
limite <- 5
notas_alunos <- c(8, 6, 3, 7, 4, 10)
aprovacao <-  notas_alunos >= limite
aprovacao
```

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

e nesse exemplo supondo que temos um vetor com as notas de alunos e a nota limite inferior para aprovação seja 5 então o vetor aprovação traz o resultado lógico indicando para as respectivas notas se o aluno foi aprovado

- Variáveis caracter

As variáveis do tipo caracter são utilizadas para armazenar palavras ou textos e devem ser definidas com a utilização de áspas “”

```
servidores <- c("Ana", "Pedro", "Carolina")
servidores
```

```
## [1] "Ana"      "Pedro"    "Carolina"
```

Um objeto que pode ser útil em algumas aplicações para criação de vetor com caracteres em sequência

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
## [18] "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
## [18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Operações entre vetores

Uma das grandes vantagens da linguagem R de programação estatística é sua implementação de operações entre vetores. Vejamos um exemplo análogo ao visto anteriormente em que tinha-se interesse em calcular o preço final de um produto após aplicar um desconto de 12%. Considere que temos o vetor *precos* que armazena o preço de diferentes produtos e desejamos calcular o preço final de cada um desses produtos após a aplicação do desconto. Pode-se realizar essa operação de forma simples na linguagem R com sua implementação vetorizada das operações

```
precos
```

```
## [1] 750 822 300  15
```

```
precos * .88
```

```
## [1] 660.00 723.36 264.00  13.20
```

e dessa forma foram calculados os preços finais dos quatro produtos com uma única operação. Nesse exemplo foram realizadas as operações entre um vetor e um escalar(vetor de tamanho 1), mas as operações vetorizadas em R permitem também que as operações sejam realizadas entre dois vetores elemento a elemento.

```
vetorA <- c(1,2,3,4)
vetorB <- c(10,20,30,40)
vetorA + vetorB
```

```
## [1] 11 22 33 44
```

No exemplo acima ambos os vetores tinham a mesma dimensão com 4 elementos cada. As operações entre vetores não estão restritas a vetores com mesma dimensão e a linguagem R utiliza-se de um procedimento chamado reciclagem para a realização de operações entre vetores de dimensões diferentes.

Os exemplos abaixo apresentam a forma em que a reciclagem é utilizada

```
v1 <- c(100,200,300,400)
v2 <- c(1,2)
v1 +  v2
```

```
## [1] 101 202 301 402
```

Note que ao realizar a soma entre os elementos dos vetores o primeiro elemento do vetor *v1* é somado ao primeiro elemento do vetor *v2* e o segundo elemento do vetor *v1* é somado ao segundo elemento do vetor *v2*. Ao realizar a soma para o terceiro elemento do vetor *v1* o vetor *v2* não tem mais elementos para a soma e dessa forma o vetor *v2* é reciclado e seu primeiro elemento é utilizado na soma com o terceiro elemento do vetor *v1* e posteriormente o quarto elemento do vetor *v1* é somado ao próximo elemento do vetor reciclado *v2* que é seu segundo elemento.

A linguagem utiliza por padrão esse procedimento de reciclagem, de forma que em nosso exemplo em que o desconto foi aplicado ao preço de diferentes produtos com uma única operação o valor do desconto foi reciclado na multiplicação pelo preço de cada um dos produtos.

Dependendo das dimensões dos vetores considerados na reciclagem, quando as operações de reciclagem não reutilizam completamente o vetor reciclado, uma mensagem de aviso é apresentada ao usuário para alertar sobre uma falta de conformidade entre as dimensões dos vetores utilizados. O exemplo a seguir ilustra esse aviso gerado:

```
V1 <- c(100,200,300)
V2 <- c(1,2)
V1 + V2
```

```
## Warning in V1 + V2: longer object length is not a multiple of shorter
## object length
```

```
## [1] 101 202 301
```

Note que o primeiro elemento de *V1* foi somado ao primeiro elemento de *V2* e o segundo elemento de *V1* foi somado ao segundo elemento de *V2*. Para somar o terceiro elemento de *V1* o vetor *V2* precisou ser reciclado de forma que esse terceiro elemento de **V1** foi somado ao primeiro elemento do vetor reciclado *V2*. O próximo elemento do vetor reciclado *V2* não é utilizado, uma vez que não existem mais elementos do vetor *V1* para ser somado. A operação de soma é realizada com a utilização de reciclagem parcial do vetor *V2* mas uma mensagem de aviso é gerada para alertar o usuário.

Indexação de vetores

Os elementos de um vetor podem ser acessados pela posição em que estão. Quando se tem interesse em acessar um determinado elemento dentro de um objeto utiliza-se [] após o nome do elemento e dentro desse operador deve se indicar os elementos de interesse.

Vamos considerar o seguinte vetor para exemplo:

```
set.seed(123)
numeros <- rpois(n = 6, lambda = 300)
numeros
```

```
## [1] 290 320 270 302 329 307
```

se tivermos interesse no número 270 que está na terceira posição pode-se acessa-lo conforme o exemplo:

```
numeros[3]
```

```
## [1] 270
```

pode-se indicar um vetor com os índices de interesse e no exemplo abaixo iremos selecionar os elementos que estão nas posições 2 e 4 do vetor *numeros*

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
numeros[c(2,4)]
```

```
## [1] 320 302
```

Alternativamente à escolha dos elementos que se deseja acessar em um vetor pode-se indicar utilizando o sinal **-** os elementos que não devem ser acessados. No exemplo abaixo vamos acessar todos os elementos do vetor *numeros* menos o segundo elemento

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
numeros[-2]
```

```
## [1] 290 270 302 329 307
```

e de forma análoga pode-se selecionar um vetor de índices que não serão acessados e no exemplo serão acessados todos os elementos do vetor *numeros* exceto os elementos com índices 1 e 4

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
numeros[-c(1,4)]
```

```
## [1] 320 270 329 307
```

A indexação dos elementos de um vetor também pode ser realizada a partir dos elementos lógicos. Suponha que tem-se interesse em acessar os dois primeiros e o último elemento do vetor *numeros*

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
numeros[c(T,T,F,F,F,T)]
```

```
## [1] 290 320 307
```

o vetor de elementos lógicos criado para indicar os elementos que devem ser acessados foi criado manualmente, mas é muito comum que esse vetor seja criado a partir de comparações. Suponha que do vetor *numeros* tenha-se interesse em acessar somente os elementos com valores maiores que 300:

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
numeros > 300
```

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

```
numeros[numeros > 300]
```

```
## [1] 320 302 329 307
```

que é equivalente a fazer manualmente

```
numeros[c(F,T,F,T,T,T)]
```

```
## [1] 320 302 329 307
```

O princípio da reciclagem também será utilizado na indexação utilizando variáveis lógicas. Pode-se, por exemplo utilizar-se dessa propriedade para selecionar apenas os elementos em índices pares fazendo:

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
numeros[c(F,T)]
```

```
## [1] 320 302 307
```

a reciclagem ocorre de forma que o indicador dos índices *c(F,T)* na primeira posição é Falso e portanto o priemeiro elemento não será acessado, o segundo indicador é verdadeiro e portanto e segundo elemento é acessado. Para decidir se o terceiro elemento será acessado o vetor de indicador dos índices é reciclado de forma que para o terceiro elemento o indicador é falso e para o quarto é positivo e a reciclagem prossegue enquanto for necessário, dado o comprimento do vetor que armazena os valores.

Ordenação de vetores

A função *sort()* pode ser utilizada para ordenação de um vetor conforme o exemplo:

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
sort(numeros)
```

```
## [1] 270 290 302 307 320 329
```

tendo como possibilidade a ordenação de forma decrescente por meio da opção vista abaixo:

```
numeros
```

```
## [1] 290 320 270 302 329 307
```

```
sort(numeros, decreasing = TRUE)
```

```
## [1] 329 320 307 302 290 270
```

Em algumas situações é importante saber as posições ordenadas dos dados. Suponha que tenha-se interesse em ordenar o vetor de **numeros** manualmente. Deveriamos primeiro pegar o elemento que está na posição 3 pois esse é o menor número, depois pegar o número que está na posição 1 que é o segundo menor número e as posições em que se deveria pegar os próximos números são 4, 6, 2, 5. E dessa forma ordenariamos o vetor:

```
numeros[c(3,1,4,6,2,5)]
```

```
## [1] 270 290 302 307 320 329
```

Esse processo que foi feito mentalmente, para encontrar as posições em que estão os valores de forma ordenada, é implementado pela função *order()*

```
order(numeros)
```

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

```
numeros[order(numeros)]
```

```
## [1] 270 290 302 307 320 329
```

tendo também a opção de buscar as posições ordenadas de forma decrescente

```
numeros[order(numeros, decreasing = TRUE)]
```

```
## [1] 329 320 307 302 290 270
```

A diferença entre as funções *sort* e *order* será importante quando deseja-se ordenar todo um conjunto de dados pelos valores de uma determinada variável de interesse.

Coerção

Os vetores são implementados em R como objetos ditos homogêneos, em que cada um de seus elementos deve ser do mesmo tipo. Um vetor numérico deve conter somente números, um vetor de caracteres deve conter somente elementos do tipo caractere. Para que a homogeneidade do vetor seja mantida elementos de tipos diferentes são transformados para que tenham o mesmo tipo por coerção. Veja nos exemplos a seguir o comportamento da linguagem R quando elementos de tipos diferentes são concatenados em um objeto homogêneo.

```
mistura <- c(1, "Ana", 3, "Beto")
mistura
```

```
## [1] "1"    "Ana"  "3"    "Beto"
```

note que aos elementos numéricos 1 e 3 foram adicionadas áspas, de forma que esses elementos foram transformados para o tipo caracter para que o vetor mantivesse a propriedade de homegeneidade. Não é possível realizar uma operação matemática com esses elementos transformados, uma vez que não são mais numéricos:

```
mistura[1]
```

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

```
mistura[1] + 1
```

```
## Error in mistura[1] + 1: non-numeric argument to binary operator
```

Ao tentar concatenar em um vetor elementos do tipo lógico com elementos do tipo caractere a transformação também acontece, transformando os elementos do tipo lógico em elementos do tipo caractere conforme pode-se ver no exemplo a seguir com as áspas adicionadas aos elementos lógicos

```
combinado <- c(TRUE, "verdade", FALSE, "mentira", T, F)
combinado
```

```
## [1] "TRUE"    "verdade" "FALSE"   "mentira" "TRUE"    "FALSE"
```

Na concatenação de elementos do tipo lógico com elementos do tipo numérico os elementos do tipo lógico são convertidos em numéricos de forma que valores TRUE são transformados no valor numérico 1 e valores FALSE são transformados em valores numéricos 0.

```
reunido <- c(10, TRUE, 1000, FALSE, 200, T, 500, F)
reunido
```

```
## [1]   10    1 1000    0  200    1  500    0
```

Essas transformações entre tipos de elementos também podem ser feitas por meio de funções. As funções deste tipo tem seus nomes definidos pelo tipo de variável que se deseja obeter com a transformação.

Para transformar elementos em numérico utiliza-se **as.numeric()**

```
logicos <- c(T, F, TRUE, FALSE)
as.numeric(logicos)
```

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

Para transformar elementos em lógicos utiliza-se **as.logical()**

```
numericos <- c(0, 5, 0, 5000)
as.logical(numericos)
```

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

e nesse tipo de transformação o valor zero é transformado em **FALSE** e os valores diferentes de zero em **TRUE**.

Para transformar elementos em caracteres utiliza-se **as.character()**

```
numericos
```

```
## [1]    0    5    0 5000
```

```
as.character(numericos)
```

```
## [1] "0"    "5"    "0"    "5000"
```

A partir do conhecimento dos objetos do tipo vetor uma extensão simples é para os objetos do tipo matriz que tem suas propriedades ilustradas na seção seguinte.

Matrizes

As matrizes são objetos com comportamento semelhante aos vetores mas apresentam duas dimensões organizando seus elementos em linhas e colunas

```
mat1 <- matrix(data = 1:12, nrow = 3)
mat1
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   11
## [3,]    3    6    9   12
```

na definição da matriz acima foram definidos os valores que compõe a matriz e foi definida uma das dimensões da matriz. A outra dimensão não precisou ser definida pois as matrizes em R sempre devem ser completas em sua forma retangular e assim se existem 12 elementos dispostos em 3 linhas consequentemente serão utilizadas 4 colunas. Nos casos em que as dimensões e a quantidade valores não for equivalente o princípio de reciclagem será utilizado para que a matriz tenha a forma retangular e seja completa com elementos. Nos casos em que a reciclagem não é completa uma mensagem de aviso é passada.

```
mat2 <- matrix(1:5, nrow = 2)
```

```
## Warning in matrix(1:5, nrow = 2): data length [5] is not a sub-multiple or
## multiple of the number of rows [2]
```

```
mat2
```

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

Note que os números das matrizes foram preenchidos por colunas, mas pode-se optar por fazer o preenchimento por linhas utilizando o argumento $byrow = TRUE$ na definição da matriz

```
mat3 <- matrix(data = 1:12, nrow = 3, byrow = TRUE)
mat3
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

Os princípios de reciclagem se aplicam tambêm para as operações com matrizes conforme ilustrado a seguir

```
mat3 * 10
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10   20   30   40
## [2,]   50   60   70   80
## [3,]   90  100  110  120
```

```
mat3 * c(-10,10,1000)
```

```
##      [,1]  [,2]  [,3]  [,4]
## [1,]  -10   -20   -30   -40
## [2,]   50    60    70    80
## [3,] 9000 10000 11000 12000
```

A indexação dos elementos na matrix segue o mesmo princípio de endereçamento considerado para os vetores, mas agora com um endereço de linha e outro de coluna na forma [linha , coluna]. Por exemplo, o elemento que está na segunda linha e terceira coluna da matriz *mat3*, que é o número 7, pode ser acessado por:

```
mat3[ 2 , 3 ]
```

```
## [1] 7
```

Podendo-se omitir o endereço de uma das dimensões para indicar o acesso a toda uma linha ou coluna. No exemplo a seguir é selecionada toda a quarta coluna da matriz *mat3* omitindo-se o endereço de linha e selecionando a quarta coluna.

```
mat3[,4]
```

```
## [1]  4  8 12
```

É possível utilizar vetores para indicar endereços de linhas e colunas e assim para selecionar a primeira e terceira linha da matriz *mat3* pode-se utilizar um vetor conforme o exemplo

```
mat3[ c(1,3) , ]
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    9   10   11   12
```

As matrizes também são objetos homgêneos, de forma que todos os elementos contidos em uma matriz devem ser do mesmo tipo. Assim como ocorre para os vetores a linguagem faz a coerção dos elementos inseridos para que todos sejam conforme no sentido de serem do mesmo tipo.

É muito comum que as informações armazenadas contenham tipos diferentes de informações envolvendo números, nomes, categorias. Para armazenar informações de diferentes tipos em um mesmo objeto pode se utilizar um *data.frame*.

Dataframe

O dataframe é um objeto semelhante a matriz por ter duas dimensões sendo elas linahs e colunas mas o dataframe tem a propriedade que cada coluna pode ser de um tipo diferente. Pode-se considerar que um dataframe é um agrupamento de vetores em colunas, onde cada coluna deve ser homogênea e possuir elementos de um único tipo.

Nesse contexto, de uma forma geral, as linhas representam as observações ou indivíduos e as colunas representam as diferentes variáveis coletadas para cada observação ou indivíduo.

O exemplo a seguir ilustra um dataframe que armazena informações médicas de personagens fictícios.

```
tabela_medica <- data.frame(nome = c("Ana","Bia","Carol","Daniela","Fernanda"),
                            idade = c(30,40,50,60,70),
                            numeroconsultas = c(3,7,1,6,2),
                            tem_plano=c(TRUE,FALSE,FALSE,TRUE,TRUE))
tabela_medica
```

```
##       nome idade numeroconsultas tem_plano
## 1      Ana    30               3      TRUE
## 2      Bia    40               7     FALSE
## 3    Carol    50               1     FALSE
## 4  Daniela    60               6      TRUE
## 5 Fernanda    70               2      TRUE
```

O acesso aos elementos do dataframe podem ser feitos de forma semelhante ao acesso na matriz, pelo endereço de linha e coluna [linha,coluna] e as diferentes colunas podem também ser acessadas por seus nomes de duas formas conforme os seguintes exemplos:

```
tabela_medica$idade
```

```
## [1] 30 40 50 60 70
```

```
tabela_medica[,c("nome","tem_plano")]
```

```
##       nome tem_plano
## 1      Ana      TRUE
## 2      Bia     FALSE
## 3    Carol     FALSE
## 4  Daniela      TRUE
## 5 Fernanda      TRUE
```

- Ordenação do data.frame

É comum o interesse em ordenar os dados por uma variável de interesse. Deve-se observar que é fundamental preservar as informações de cada um dos indivíduos. Deve-se determinar a ordem em que as linhas do data.frame será apresentado e a função *order* será utilizada. No exemplo seguinte a tabela médica é ordenada pelo número de consultas, de forma que as pessoas com mais consultas são apresentadas no topo da tabela.

```
tabela_medica
```

```
##       nome idade numeroconsultas tem_plano
## 1      Ana    30               3      TRUE
## 2      Bia    40               7     FALSE
## 3    Carol    50               1     FALSE
## 4  Daniela    60               6      TRUE
## 5 Fernanda    70               2      TRUE
```

```
tabela_medica[order(tabela_medica$numeroconsultas, decreasing = TRUE),]
```

```
##       nome idade numeroconsultas tem_plano
## 2      Bia    40               7     FALSE
## 4  Daniela    60               6      TRUE
## 1      Ana    30               3      TRUE
## 5 Fernanda    70               2      TRUE
## 3    Carol    50               1     FALSE
```

Todas as colunas de um dataframe devem ter a mesma dimensão ou número de elementos. Um objeto que permite armazenar variáveis com tipos diferentes e dimensões diferentes é a lista.

Lista

A lista é um objeto separado em *containers* de forma que cada um desses *containers* pode conter um objeto(vetor, matriz, dataframe, lista). Veja o seguinte exemplo:

```
lista1 <- list(
  numeros = numeros,
  matriz = mat3,
  tabela = tabela_medica
)
lista1
```

```
## $numeros
## [1] 290 320 270 302 329 307
##
## $matriz
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
```

```
## [3,]    9   10   11   12
##
## $tabela
##       nome idade numeroconsultas tem_plano
## 1      Ana    30               3      TRUE
## 2      Bia    40               7     FALSE
## 3    Carol    50               1     FALSE
## 4  Daniela    60               6      TRUE
## 5 Fernanda    70               2      TRUE
```

Pode-se acessar os elementos da lista usando seus nomes ou posições conforme os exemplos a seguir

```
lista1["numeros"]
```

```
## $numeros
## [1] 290 320 270 302 329 307
```

```
lista1[1]
```

```
## $numeros
## [1] 290 320 270 302 329 307
```

Ao acessar o vetor de números contido no primeiro container da lista o resultado é uma lista com um único container preenchido com o vetor. Para ter acesso ao vetor diretamente sem que esse esteja em uma lista pode-se utilizar as seguintes opções:

```
lista1$numeros
```

```
## [1] 290 320 270 302 329 307
```

```
lista1[[1]]
```

```
## [1] 290 320 270 302 329 307
```

O resultado da diferença entre acessar um objeto que está em container da lista diretamente ou acessar uma cópia de lista em que o objeto de interesse está no primeiro *container* pode ser visto no exemplo seguinte:

```
lista1[1]
```

```
## $numeros
## [1] 290 320 270 302 329 307
```

```
lista1[1]*10
```

```
## Error in lista1[1] * 10: non-numeric argument to binary operator
```

```
lista1[[1]]
```

```
## [1] 290 320 270 302 329 307
```

```
lista1[[1]]*10
```

```
## [1] 2900 3200 2700 3020 3290 3070
```

Note que a operação não foi possível quando o objeto acessado era uma lista com um vetor em seu primeiro *container* mas foi possível quando o objeto acessado era um vetor.

Até esta parte foram consideradas as principais estruturas de dados utilizadas na linguagem R: escalar, vetor, matriz, data.frame e lista. Para melhor utilização das ferramentas da linguagem é importante conhecer também a forma para a criação de funções.

Funções

As funções permitem o reuso de código e simplificam a realização de tarefas repetidas que devem ser realizadas em conjuntos de dados. As funções tem como propriedades um conjunto de entradas ou parâmetros e um objeto ou comportamento como saída.

Em um primeiro exemplo será considerada uma função bem simples que recebe uma entrada *X1* e soma 2 unidades a essa entrada como saída.

```
soma2 <- function(X1){
  X1 + 2
}
```

Para utilizar a função basta chamá-la passando o argumento de interesse

```
soma2(10)
```

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

Note que na linguagem R não é preciso definir o tipo dos parâmetros das funções, de forma que elas podem ter comportamento flexível

```
soma2(c(10,100,1000))
```

```
## [1]   12  102 1002
```

e quando um vetor é passado como argumento da função o retorno dessa função também é um vetor.

É importante perceber que como não é necessária a definição do tipo dos parâmetros a função pode ter comportamentos indesejados quando tipos de objetos não previstos são passados como argumentos.

```
soma2("vinte")
```

```
## Error in X1 + 2: non-numeric argument to binary operator
```

É recomendável portanto fazer verificação dos tipos de argumentos das funções conforme se vê em funções mais complexas.

Como outro exemplo de função considere que seja corriqueiro o interesse em extrair os 3 maiores elementos de um vetor numérico. A função seguinte faz essa extração

```
maiores <- function(vetor){
  ordenado <- sort(vetor , decreasing = TRUE)
  ordenado[3:1]
}
```

```
vetor_aleatorio <- rpois(10,1000)
vetor_aleatorio
```

```
##  [1]  959  945 1038 1011 1012 1003  982 1040 1026  998
```

```
maiores(vetor_aleatorio)
```

```
## [1] 1026 1038 1040
```

Nos exemplos anteriores as funções tinham apenas um elemento em seu conjunto de entradas, mas é comum que as funções tenham múltiplos parâmetros.

Veja o exemplo da função que extrai os k maiores valores de um vetor, em que o valor de k é flexível e pode ser alterado conforme a necessidade.

```
maiores <- function(vetor, k){
  ordenado <- sort(vetor, decreasing = TRUE)
  ordenado[k:1]
```

```
}
maiores(vetor_aleatorio, 2)
```

```
## [1] 1038 1040
```

```
maiores(vetor_aleatorio, 4)
```

```
## [1] 1012 1026 1038 1040
```

Execução condicional

A estrutura de execução condicional permite determinar o fluxo de execução do programa.

Num primeiro exemplo de execução condicional utilizando a **tabela_medica** vamos considerar que os indivíduos que não tem plano de saúde precisam ter registrados o valor devido em consultas supondo que o preço por consulta seja de R$150,00.

A função que calcula o valor devido recebe como entrada todas as informações do paciente e calcula como saída o valor da consulta multiplicado pelo número de consultas apenas para os que não tem plano, e para os outros retorna o valor zero.

```
valor_devido <- function(pessoa){
  if(pessoa[,4] == TRUE){
    return(0)
  }else{
    return(pessoa[,3] * 150.00)
  }
}
```

e assim para a Ana, a função deve calcular o valor igual a zero pois Ana tem plano de saúde

```
tabela_medica
```

```
##       nome idade numeroconsultas tem_plano
## 1      Ana    30               3      TRUE
## 2      Bia    40               7     FALSE
## 3    Carol    50               1     FALSE
## 4  Daniela    60               6      TRUE
## 5 Fernanda    70               2      TRUE
```

```
valor_devido(tabela_medica[1,])
```

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

para Bia a função deve calcular o valor para as 7 consultas registradas

```
valor_devido(tabela_medica[2,])
```

```
## [1] 1050
```

A execução condicional ocorreu a partir da determinação se a variável que referente ao plano de saúde tem valor verdadeiro ou falso, e para cada uma das possíveis condições, a execução da função tem seu comportamento diferenciado.

Para a execução dessa função que calcula os valores devidos para todos os indivíduos no data.frame podem ser utilizadas laços de repetição.

Laços de repetição

Para a utilização de laço de reptição no cálculo dos valores devidos para todos os indivíduos da *tabelamedica* será criado um novo vetor vazio *valores* para armazenar os valores devidos calculados e então um laço do tipo *for* será utilizado com um contador i que varia do número 1 até o número 5 que representa o número total de linhas no dataframe desse exemplo

```
valores <- c()
for(i in 1:5){
  valores[i] <- valor_devido(tabela_medica[i,])
}
valores
```

```
## [1]    0 1050  150    0    0
```

```
tabela_medica$valores <- valores
tabela_medica
```

```
##       nome idade numeroconsultas tem_plano valores
## 1      Ana    30               3      TRUE       0
## 2      Bia    40               7     FALSE    1050
## 3    Carol    50               1     FALSE     150
## 4  Daniela    60               6      TRUE       0
## 5 Fernanda    70               2      TRUE       0
```

Alternativamente pode-se utilizar um laço do tipo *while* que repetirá o código enquanto o contador i utilizado, que começa com valor 1, for menor que o valor 5. Note que o contador i é incrementado a cada passagem do código:

```
valores1 <- c()
i <- 1
while(i<5){
  valores1[i] <- valor_devido(tabela_medica[i,])
  i <- i + 1
}
valores1
```

```
## [1]    0 1050  150    0
```

na linguagem R também é possível mapear a aplicação das funções repetidamente a conjuntos de elementos por meio da família apply de funções.

Nos exemplos vistos até este ponto os dados foram digitados maunalmente para ilustração das principais funcionalidades básicas da linguagem, mas é importante que seja possível importar dados de outras fontes.

Importação de dados

A função básica para importação de dados para R é a função *read.table*. A utilização dessa função permite importar diferentes tipos de dados por meio da definição dos parâmetros que controlam as diferentes opções entre os tipos de arquivos, sendo elas: separador de campos, presença de cabeçalho, separador de números decimais e etc.

No exemplo seguinte serão importados dados referentes a pesca de camarões no estado do Rio Grande do Norte

```
camaroes <- read.table("https://raw.githubusercontent.com/cursoRunb/CursoRJ2018/master/dados/dadoscamara
                       head = TRUE,
                       dec=",")
```

```
## Warning in file(file, "rt"): cannot open URL 'https://
## raw.githubusercontent.com/cursoRunb/CursoRJ2018/master/dados/
## dadoscamarao.txt': HTTP status was '404 Not Found'
```

```
## Error in file(file, "rt"): cannot open the connection to 'https://raw.githubusercontent.com/cursoRunb
```

```
head(camaroes)
```

```
## Error in head(camaroes): object 'camaroes' not found
```

para a importação desse conjunto de dados foram passados como argumentos o endereço do arquivo de interesse, a informação que os dados tem um cabeçalho, de forma que a primeira linha dos dados é referente ao nome das variáveis e finalmente a indicação que os números decimais estão separados por vírgulas.

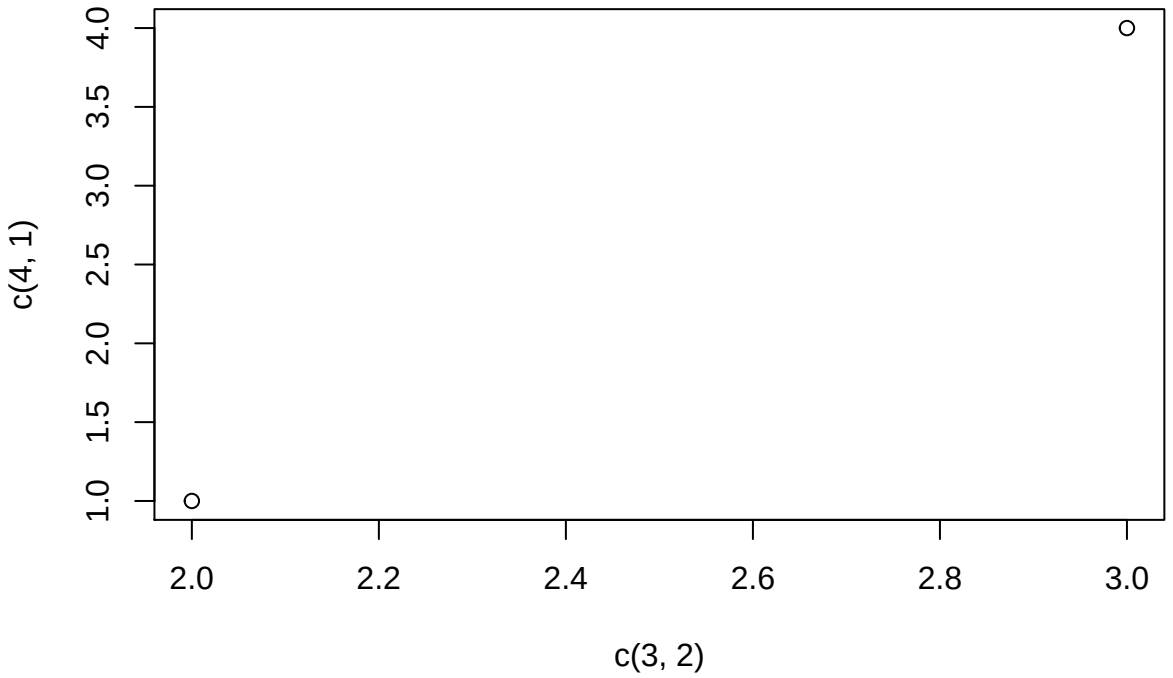
A função *head* utilizada após a importação dos dados tem como objetivo exibir as primeiras linhas do banco de dados e é comumente utilizada para verificar se a importação dos dados foi bem sucedida.

Gráficos

Para ilustrar algumas possibilidades gráficas serão explorados os dados de camarões importadas no exemplo anterior.

Inicialmente será considerado o mapeamento de pontos no sistema de coordenadas cartesianas em que cada ponto é um par ordenado de coordenadas. Por exemplo um conjunto de pontos: $P1$ com coordenadas $(X_1 = 3, Y_1 = 4)$ e $P2$ com coordenadas $(X_1 = 2, Y_1 = 1)$ será representado gráficamente utilizando a seguinte função:

```
plot(x = c(3,2), y = c(4,1))
```



e pode-se ver os pontos representados em suas respectivas coordenadas.

A partir desse conceito básico da representação dos pontos por suas coordenadas será criado um diagrama de dispersão representando a associação entre os pesos e os comprimentos dos camarões do conjunto de dados de exemplo. Cada ponto representa um dos 120 animais capturados de forma que sua localização no gráfico faz referência ao seu comprimento total e seu peso

```
plot(camaroes$Ct, camaroes$Peso)
```

```
## Error in plot(camaroes$Ct, camaroes$Peso): object 'camaroes' not found
```

Para ilustrar as diferentes formas de controlar as opções gráficas serão feitas modificações nesse gráfico para que ele esteja mais informativo e adequado para a publicação em um relatório. O primeiro passo é controlar os rótulos dos eixos X e Y

```
plot(camaroes$Ct, camaroes$Peso,
     xlab = "Comprimento total em mm", ylab = "Peso em gramas")
```

```
## Error in plot(camaroes$Ct, camaroes$Peso, xlab = "Comprimento total em mm", : object 'camaroes' not
```

Pode-se atribuir um título ao gráfico utilizando o parâmetro *main* da função

```
plot(camaroes$Ct, camaroes$Peso,
     xlab = "Comprimento total em mm", ylab = "Peso em gramas",
     main = "Camarões do Rio Grande do Norte")
```

```
## Error in plot(camaroes$Ct, camaroes$Peso, xlab = "Comprimento total em mm", : object 'camaroes' not
```

Uma vez que temos informações sobre o sexo desses camarões pode-se utilizar alguma estratégia para diferenciar os pontos referentes aos camarões machos e fêmeas. Para utilizar cores na diferenciação dos pontos será preciso indicar para cada camarão qual cor deve ser utilizada em seu ponto. Como as cores serão definidas por seus nomes em inglês será utilizada uma função para criar um vetor com as cores "blue" para camarões macho e "pink" para camarões fêmea

```
cor_camarao <- ifelse(camaroes$Sexo == "M","blue","pink")
```

```
## Error in ifelse(camaroes$Sexo == "M", "blue", "pink"): object 'camaroes' not found
```

```
plot(camaroes$Ct, camaroes$Peso,
     xlab = "Comprimento total em mm", ylab = "Peso em gramas",
     main = "Camarões do Rio Grande do Norte",
     col = cor_camarao)
```

```
## Error in plot(camaroes$Ct, camaroes$Peso, xlab = "Comprimento total em mm", : object 'camaroes' not
```

Uma vez que essa diferenciação por cores foi feita é interessante adicionar uma legenda para indicar a relação entre as cores e os sexos dos camarões. A função legend permite que se defina as coordenadas que a caixa de legenda será posicionada.

```
plot(camaroes$Ct, camaroes$Peso,
     xlab = "Comprimento total em mm", ylab = "Peso em gramas",
     main = "Camarões do Rio Grande do Norte",
     col = cor_camarao)
```

```
## Error in plot(camaroes$Ct, camaroes$Peso, xlab = "Comprimento total em mm", : object 'camaroes' not
```

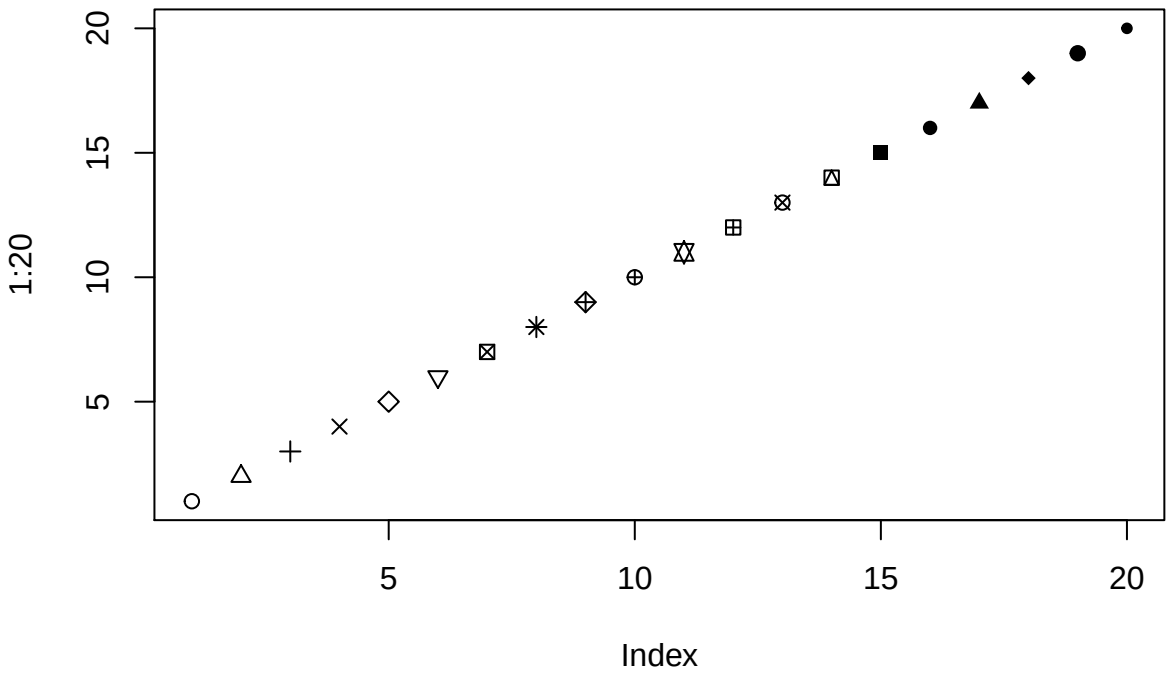
```
legend(x = 60,y = 30,legend = c("Macho","Femea"), fill = c("blue","pink"))
```

```
## Error in strwidth(legend, units = "user", cex = cex, font = text.font): plot.new has not been called
```

Uma vez que a localidade de pesca de cada animal essa informação também pode ser adicionada ao gráfico pela utilização de tipos diferentes de pontos.

Antes de adicionar essa informação ao gráfico de camarões a o gráfico seguinte ilustra diferente tipos de pontos disponíveis.

```
plot(1:20,pch=1:20)
```



e assim como os tipos de pontos são definidos por números, será criada uma variável auxiliar para definir o tipo de ponto que irá representar cada diferente localidade de pesca

```
local_numero <- as.numeric(camaroes$Local)
```

```
## Error in eval(expr, envir, enclos): object 'camaroes' not found
```

```
plot(camaroes$Ct, camaroes$Peso,
     xlab = "Comprimento total em mm", ylab = "Peso em gramas",
     main = "Camarões do Rio Grande do Norte",
     col = cor_camarao,
     pch = local_numero)
```

```
## Error in plot(camaroes$Ct, camaroes$Peso, xlab = "Comprimento total em mm", : object 'camaroes' not
```

```
legend(x = 60,y = 30,
       legend = c("Macho","Femea","BaiaFormosa","DiogoLopes","Touros"),
       col = c("blue","pink","black","black","black"),
       pch = c(15,15,1,2,3))
```

```
## Error in strwidth(legend, units = "user", cex = cex, font = text.font): plot.new has not been called
```

Para representar variáveis quantitativas em outras formas gráficas pode-se considerar os seguintes exemplos:

```
hist(camaroes$Peso, main = "Histograma", xlab="Peso em gramas")
```

```
## Error in hist(camaroes$Peso, main = "Histograma", xlab = "Peso em gramas"): object 'camaroes' not fo
```

```
boxplot(camaroes$Peso ~ camaroes$Sexo)
```

```
## Error in eval(predvars, data, env): object 'camaroes' not found
```

Para variáveis qualitativas deve-se fazer a contagem nas diferentes categorias para criar gráficos.

```
table(camaroes$Sexo)
```

```
## Error in table(camaroes$Sexo): object 'camaroes' not found
```

```
barplot(table(camaroes$Sexo))
```

```
## Error in table(camaroes$Sexo): object 'camaroes' not found
```

```
pie(table(camaroes$Local))
```

```
## Error in table(camaroes$Local): object 'camaroes' not found
```

Relação com algumas das funções vistas nos exemplos desta seção

- c()
- matrix()
- list()
- operador :
- seq()
- rep()
- rnorm()
- rpois()
- sort()
- order()
- function()
- if()
- ifelse()
- for()
- while()
- table()
- plot()
- legend()
- hist()
- boxplot()
- barplot()
- pie()