PROGRAMAÇÃO

1. Conceitos Básicos do R

Rita P. Ribeiro

Mestrado em Informática Médica
Curso de Especialização em Informática da Saúde

2016/2017

U.PORTO
FMUP FACULDADE DE MEDICINA UNIVERSIDADE DO PORTO
FC FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

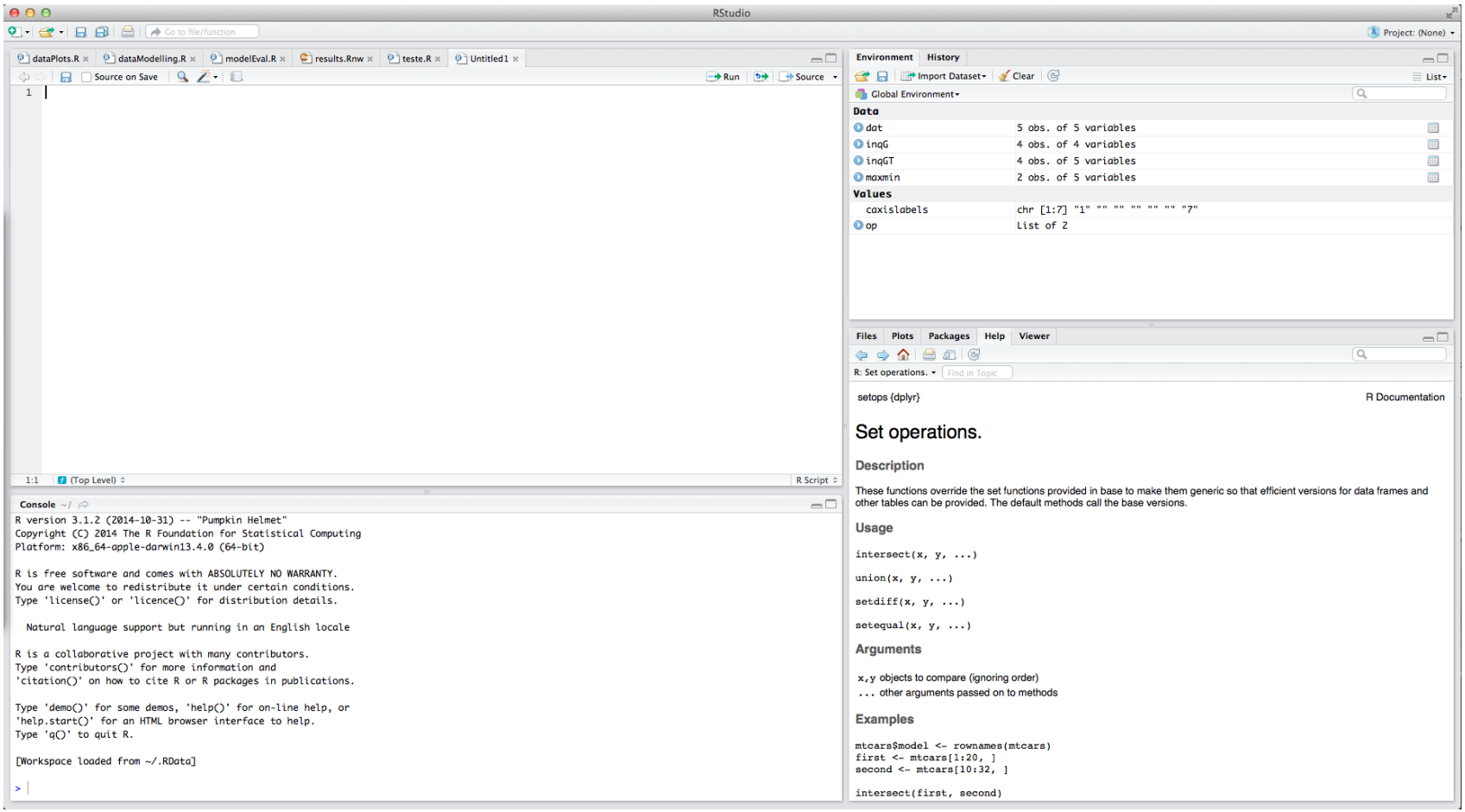
Programa

1. Conceitos Básicos do R
2. Manipulação de Dados
3. Sumarização de Dados
4. Visualização de Dados
5. Geração de Relatórios

Introdução ao R

RStudio

Um IDE para o R. Livre. Funciona em Windows, Mac e Linux



Introdução ao R

Interação básica com a consola

- A forma mais comum de interação com o R é através da sua linha de comando
  - O utilizador digita um comando
  - Pressiona a tecla ENTER
  - O R "devolve" a resposta
- É também possível guardar um conjunto de comandos num ficheiro (tipicamente com a extensão `.R`) e depois pedir ao R para executar todos os comandos no ficheiro de uma vez

Introdução ao R

Interação básica com a consola (2)

- Podemos usar a linha de comando como uma simples calculadora

```
1+3/5*6^2
```

```
## [1] 22.6
```

- Podemos tirar partido das inúmeras funções disponíveis na linguagem R

```
rnorm(5,mean=30,sd=10)
```

```
## [1] 22.30537 22.67923 49.71342 32.47532 47.48740
```

```
mean(sample(1:1000,30)) # composição de funções
```

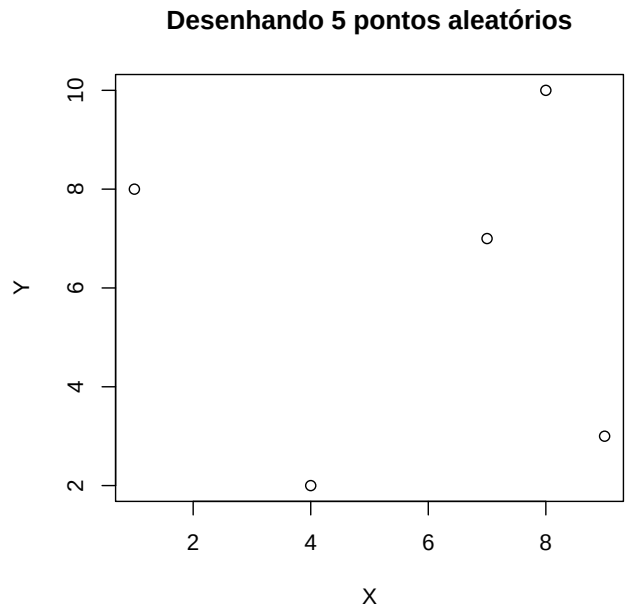
```
## [1] 547.2
```

Introdução ao R

Interação básica com a consola (3)

- Podemos produzir gráficos

```
plot(sample(1:10,5),sample(1:10,5),
     main="Desenhando 5 pontos aleatórios",
     xlab="X",ylab="Y")
```



Variáveis e Objetos

- Tal como acontece em outras linguagens de programação, o R permite a atribuição de **objetos** a **variáveis**:

```
<variavel> <- <objeto>
```

- Estes objetos poderão ser de diferentes tipos (p.ex. um número, texto, uma expressão numérica, uma chamada a uma função, uma tabela de dados, um gráfico, etc.).
- Uma vez feita a atribuição, o interpretador do R irá substituir o nome da variável pelo seu conteúdo.

```
x <- 10
x

## [1] 10

x * 2

## [1] 20
```

Variáveis e Objetos

Atribuição

- Uma variável pode guardar o resultado da avaliação de uma expressão

```
pesoKg <- 60
pesoLb <- pesoKg * 2.20462
pesoLb
```

```
## [1] 132.2772
```

- **Nota**: a atribuição (<-) é destrutiva!
- Isto significa que se atribuirmos a uma variável, que já guarda um objeto, um novo objeto, só este último será guardado sendo o anterior perdido.

```
pesoLb <- 120
pesoLb
```

```
## [1] 120
```

Variáveis e Objetos

Tipos Básicos de Dados

Os objetos em R possuem dois atributos intrínsecos:

- `mode`: indica o tipo básico do conteúdo do objeto
  - `numeric` (p.ex. `5`, `6.3`, `10.344`, `-2.3`, `-7`)
  - `character` (p. ex. `"olá"`, `"está um belo dia"`)
  - `logical` (`TRUE`, `FALSE`)
  - etc.
- `length`: indica o número de elementos contidos no objeto

```
x <- 10
mode(x)

## [1] "numeric"

length(x)

## [1] 1
```

```
mode("ola")

## [1] "character"

mode(ola)

## Error in mode(ola):
object 'ola' not found
```

```
mode(TRUE)

## [1] "logical"

mode(true)

## Error in mode(true):
object 'true' not found
```

Variáveis e Objetos

Tipos de Objetos de Dados

- No R, existem 7 tipos principais de objetos e que se caracterizam pelo tipo de dados que podem conter:

objeto	**tipo de dados (`mode`)**	**vários tipos?**
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

- Mais à frente, iremos ver em mais detalhe cada um deles.

Variáveis e Objetos

Alguns Operadores

- Uma variável pode guardar o resultado de uma expressão composta por um conjunto de operadores.

Aritméticos		Comparação		Lógicos	
`+`	adição	`<`	menor	`!x`	negação
`-`	subtração	`>`	maior	`x&y`	conjunção
`*`	multiplicação	`<=`	menor ou igual	`x&&y`	conjunção (*lazy*)
`/`	divisão	`>=`	maior ou igual	`x|y`	disjunção
`^`	potência	`==`	igual	`x||y`	disjunção (*lazy*)
`%%`	resto	`!=`	diferente	`xor(x,y)`	disjunção excl.
`%/%`	divisão inteira				

```
15 %% 2^2 * 10
```

```
## [1] 30
```

```
(15 %% 2)^2 * 10
```

```
## [1] 10
```

```
4 == 5
```

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

```
4 != 5
```

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

```
TRUE & (4 > 5)
```

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

```
TRUE & !(4 > 5)
```

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

Funções

- Em R a maioria das coisas que podemos fazer são levadas a cabo por funções.
- Uma função é um objeto que recebe um conjunto de zero ou mais objetos de entrada (argumentos ou parâmetros) e devolve um outro objeto.
- O R disponibiliza um conjunto de funções.

```
log(1.5) # calcula o logaritmo na base e
```

```
## [1] 0.4054651
```

```
sin(pi/2) # calcula o seno
```

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

```
sqrt(25) # calcula a raíz quadrada
```

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

Funções

Criar Novas Funções

- Podemos também criar novas funções para automatizar operações frequentes.
  - Podemos, por exemplo, criar uma função, que dado um peso em lbs e um valor de conversão nos fornece o peso em kgs.

```
lb2kg <- function(p,conv) p*conv
lb2kg(100,0.453592)

## [1] 45.3592
```

  - Também podemos indicar que alguns dos parâmetros têm **valores por omissão**

```
lb2kg <- function(p,conv=0.453592) p*conv
lb2kg(100)

## [1] 45.3592
```

Funções

Composição de Funções

- A composição de funções é uma noção importante na matemática
  - $(f \circ g)(x) = f(g(x))$ significa aplicar a função f ao resultado de aplicar a função g a x
- O R é uma linguagem funcional; podemos usar a composição de funções como forma de realizar operações complexas sem termos que armazenar os resultados intermédios
- Exemplo: duas soluções para calcular $e^{\sqrt{10}}$

sem composição de funções

```
x <- 10
temp <- sqrt(x)
y <- exp(temp)
y
```

```
## [1] 23.62434
```

com composição de funções

```
x <- 10
y <- exp(sqrt(x))
y
```

```
## [1] 23.62434
```

Tipos de Objetos de Dados

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Vetores

- Um vetor é uma classe de objetos que contêm informação sobre um conjunto de valores **do mesmo tipo básico de dados**
  - p.ex. os preços de uma série de produtos vendidos numa loja
- Se um conjunto de valores partilha algo de comum e são do mesmo tipo básico de dados, pode fazer sentido guardá-los sob a forma de um vetor
- Um vetor é mais um exemplo de um objecto que podemos guardar numa variável do R

Vetores

Criação

- Vamos criar um vetor com os pesos de 5 indivíduos

```
pesos <- c(62.6,55.4,70.2,75,68.1)
pesos

## [1] 62.6 55.4 70.2 75.0 68.1
```

- No lado direito da atribuição temos uma chamada à função `c()` usando como argumentos os valores dos pesos que queremos guardar como um vetor
- A função `c()`, de `combine`, pega nos seus argumentos e produz como resultado um vetor contendo esses valores

Vetores

Criação (2)

- A função `c()` permite-nos associar nomes aos membros do vetor.
- No exemplo anterior poderia fazer sentido associar a cada valor o nome do indivíduo,

```
pesos <- c(Marco=62.6,Filipe=55.4,Valter=70.2,Nuno=75,João=68.1)
```

- Isto torna o significado do vetor mais claro, e como iremos ver, vai facilitar o acesso à informação guardada no vetor.

```
pesos
```

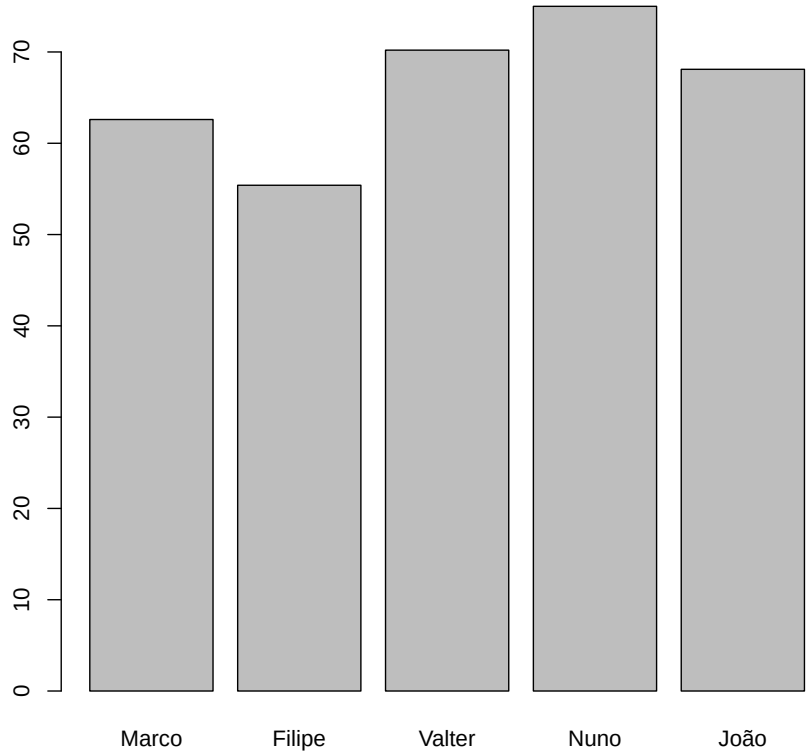
```
##  Marco Filipe Valter   Nuno   João
##   62.6   55.4   70.2   75.0   68.1
```

Vetores

Criação (3)

- Para além de mais claro, o uso de nomes é também recomendado porque o R tira partido destes nomes em várias situações.
- Este é o caso da maioria das funções que produzem gráficos:

```
barplot(pesos)
```



Vetores

Geração

- Podemos gerar vetores a partir de sequências regulares
  - operador `:`

```
# inteiros entre 1 e 10
1:10

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

  - função `rep()`

```
# 10 repetições do valor 1
rep(1,10)

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

# 10 repetições do vetor c(1,2)
rep(c(1,2),10)

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

Vetores

Geração (2)

- Podemos gerar vetores a partir de sequências regulares (cont.)
  - função `seq()`

```
# inteiros ímpares entre 1 e 10
seq(1,10,by=2)

## [1] 1 3 5 7 9

# ...
seq(1,5,length=9)

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

  - etc.

Vetores

Geração (3)

- Podemos gerar vetores a partir de sequências aleatórias
  - função `sample()`

```
# 5 pontos entre os inteiros de 1 a 10
sample(1:10,size=5)

## [1] 10  9  3  6  1
```

  - função `rnorm()`

```
# 5 pontos a partir de uma Distribuição Normal
rnorm(5,mean=0,sd=1)

## [1]  0.5443721  1.0296054 -0.2986404  1.1542553  1.9433560
```

  - função `runif()`

```
# 5 pontos a partir de uma Distribuição Uniforme
runif(5,min=0,max=1)

## [1] 0.3575183 0.7377517 0.5200198 0.6324686 0.5809737
```

  - etc.

Vetores

Indexação

- Sempre que temos objetos com vários valores (p.ex. vetores) podemos estar interessados em aceder a algum desses valores individualmente.
- Esse é o **objetivo da indexação**: aceder a um sub-conjunto dos valores guardados numa variável.
- Em matemática estamos habituados a usar índices. Por exemplo, x_3 normalmente representa o 3º elemento do conjunto de valores x.
- Em R a ideia é semelhante:

```
pesos <- c(Marco=62.6,Filipe=55.4,Valter=70.2,Nuno=75,João=68.1)
pesos[3]

## Valter
##   70.2
```

Vetores

Indexação (2)

- Também podemos usar os nomes das posições nos vetores para a indexação

```
pesos <-  c(Marco=62.6,Filipe=55.4,Valter=70.2,Nuno=75,João=68.1)
pesos["Nuno"]

## Nuno
##   75
```

- `pesos["Nuno"]` acede à posição do vetor `pesos` indexada com o texto `"Nuno"`.
- **Nota**: `pesos[Nuno]` acede à posição do vetor `pesos` indexada pelo conteúdo da variável `Nuno` (que não existe!).

```
pesos[Nuno]

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

Vetores

Indexação (3)

- Também podemos usar um vetor como índice e desta forma aceder a mais do que uma posição ao mesmo tempo

```
pesos <-  c(Marco=62.6,Filipe=55.4,Valter=70.2,Nuno=75,João=68.1)
pesos[c(2,4)]
```

```
## Filipe   Nuno
##   55.4   75.0
```

- Estamos portanto a aceder às posições 2 e 4 do vetor `pesos`
- O mesmo se aplica a vetores de nomes

```
pesos[c("Filipe","Nuno")]
```

```
## Filipe   Nuno
##   55.4   75.0
```

Vetores

Indexação (4)

- Também podemos usar condições lógicas para "questionar" os dados!

```
pesos[pesos > 70]

## Valter   Nuno
##   70.2   75.0
```

- A ideia é que o resultado da indexação serão os valores do vetor `pesos` em que a condição indicada é verdadeira
- O que é obtido com a seguinte instrução?

```
pesos[pesos > mean(pesos)]

## Valter   Nuno   João
##   70.2   75.0   68.1
```

- Este é mais um exemplo de composição de funções!

Vetores

Vetorização de Operações

- A grande maioria das funções e operadores do R podem ser aplicadas diretamente a conjuntos de valores (p.ex. vetores)
- Suponhamos que pretendemos saber os pesos anteriores em libras (lbs).

```
umKg <- 2.20462 # valor de 1kg em lb
pesos * umKg
```

```
##    Marco   Filipe   Valter     Nuno     João
## 138.0092 122.1359 154.7643 165.3465 150.1346
```

- Multiplicámos **um** número por um **conjunto** de números!
- O resultado é um outro conjunto de números que são obtidos pela multiplicação de cada número no vetor `pesos` por 2.20462

Vetores

Vetorização de Operações (2)

- Exemplo de vetorização de funções:

```
round(pesos)

##  Marco Filipe Valter   Nuno   João
##     63     55     70     75     68
```

- Ao aplicar a função `round()` a um vetor em vez de a um só número, ela vai produzir como resultado um vetor com o mesmo tamanho, resultante de aplicar a função a cada número no vetor original.

Vetores

Vetorização de Operações (3)

- Podemos fazer coisas semelhantes com dois conjuntos de números.
- Suponhamos que também temos a altura de cada um dos indivíduos:

```
alturas <- c(Marco=1.65,Filipe=1.60,Valter=1.69,Nuno=1.72,João=1.73)
alturas
```

```
##  Marco Filipe Valter   Nuno   João
##   1.65   1.60   1.69   1.72   1.73
```

- Podemos calcular o IMC de cada um deles:

```
IMC <- round(pesos/alturas^2,1)
IMC
```

```
##  Marco Filipe Valter   Nuno   João
##   23.0   21.6   24.6   25.4   22.8
```

Vetores

Vetorização de Operações (4)

- As condições lógicas com vetores que vimos anteriormente são outro exemplo de vetorização

```
pesos > 70

##  Marco Filipe Valter   Nuno   João
##  FALSE  FALSE   TRUE   TRUE  FALSE
```

- Estamos a comparar os 5 pesos com um só número (`70`). O resultado é um vetor com os valores obtidos por cada uma das comparações. Por vezes é verdade, noutros casos é falso.
- No caso de `pesos[pesos > 70]`, a expressão de indexação é um vetor de valores booleanos cujo significado é aceder às posições onde estão os valores verdadeiros.

Vetores

Algumas Funções

- Exemplos de funções aplicáveis a vetores:

`max(x)`	máximo dos elementos de `x`
`mean(x)`	média dos elementos de `x`
`median(x)`	mediana dos elementos de `x`
`min(x)`	mínimo dos elementos de `x`
`prod(x)`	produto dos elementos de `x`
`range(x)`	intervalo de valores dos elementos de `x`
`round(x,n)`	arredonda os elementos de `x` a `n` casas decimais
`sort(x)`	ordena os elementos de `x`
`sum(x)`	soma dos elementos de `x`
`unique(x)`	os elementos de `x` sem duplicados
`which(x == a)`	o índice do elemento de `x` que é igual a `a`
`which.max(x)`	o índice do máximo dos elementos de `x`
`which.min(x)`	o índice do mínimo dos elementos de `x`
`...`	

- Algumas destas funções são apenas aplicáveis a vetores numéricos ...

Tipos de Objetos de Dados

Matrizes

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Matrizes

- Como os vetores, as matrizes também podem ser usadas para guardar **conjuntos** de valores do mesmo tipo básico de dados que estejam de alguma forma relacionados
- Todavia as matrizes "espalham" os valores por duas dimensões: as linhas e as colunas
- Voltemos ao cenário dos pesos e alturas. Poderia fazer mais sentido guardar esta informação numa matriz em vez de em dois vetores
- As colunas poderiam corresponder ao peso e à altura e as linhas aos indivíduos.

Matrizes

Criação

- Vejamos como criar essa matriz:

```
pesoalt <- matrix(c(62.6,55.4,70.2,75,68.1,
                    1.65,1.60,1.69,1.72,1.73),
                  nrow=5,ncol=2)
```

- A função `matrix()` pode ser usada para criar matrizes
- Temos que lhe dar pelo menos os valores a colocar na matriz bem como o nº de linhas e/ou colunas

```
pesoalt

##      [,1] [,2]
## [1,] 62.6 1.65
## [2,] 55.4 1.60
## [3,] 70.2 1.69
## [4,] 75.0 1.72
## [5,] 68.1 1.73
```

Matrizes

Criação (2)

```
pesoalt <- matrix(c(62.6,55.4,70.2,75,68.1,
                    1.65,1.60,1.69,1.72,1.73),
                  nrow=5,ncol=2)
pesoalt

##      [,1] [,2]
## [1,] 62.6 1.65
## [2,] 55.4 1.60
## [3,] 70.2 1.69
## [4,] 75.0 1.72
## [5,] 68.1 1.73
```

- O parâmetro `nrow` indica qual o número de linhas
- O parâmetro `ncol` indica o número de colunas
- Por omissão, os valores fornecidos são "espalhados"por coluna.

Matrizes

Indexação

- Semelhante aos vetores mas agora com duas dimensões

```
pesoalt

##      [,1] [,2]
## [1,] 62.6 1.65
## [2,] 55.4 1.60
## [3,] 70.2 1.69
## [4,] 75.0 1.72
## [5,] 68.1 1.73

pesoalt[4,2]

## [1] 1.72
```

- Também podemos obter toda uma linha ou coluna,

```
pesoalt[1,] # 1a linha

## [1] 62.60  1.65
```

```
pesoalt[,2] # 2a coluna

## [1] 1.65 1.60 1.69 1.72 1.73
```

Matrizes

Indexação (2)

- Como nos vetores também aqui podemos dar nomes às duas dimensões das matrizes

```
colnames(pesoalt) <- c("peso","altura")
rownames(pesoalt) <- c("Marco","Filipe","Valter","Nuno","João")
pesoalt
```

```
##        peso altura
## Marco  62.6   1.65
## Filipe 55.4   1.60
## Valter 70.2   1.69
## Nuno   75.0   1.72
## João   68.1   1.73
```

- As funções `colnames()` e `rownames()` podem ser usadas para obter os nomes das respetivas dimensões ou para indicar quais são atribuindo-lhes vetores com os nomes.

Matrizes

Indexação (3)

- Os nomes também podem ser usados para indexação

```
pesoalt
```

```
##        peso altura
## Marco  62.6   1.65
## Filipe 55.4   1.60
## Valter 70.2   1.69
## Nuno   75.0   1.72
## João   68.1   1.73
```

```
pesoalt["Marco",]
```

```
##   peso altura
##  62.60   1.65
```

```
pesoalt["Valter","altura"]
```

```
## [1] 1.69
```

Tipos de Objetos de Dados

Arrays

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Arrays

Criação

- *Arrays* são extensões das matrizes para mais do que 2 dimensões
- Podemos criar um *array* com a função `array()`

```
a <- array(1:12,dim=c(2,3,2))
a

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

Arrays

Indexação

- O mesmo tipo de esquemas vistos até agora podem ser usados nos *arrays*

```
a[1,2,1]
```

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

```
a[1,,2]
```

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

```
a[,,1]
```

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

Tipos de Objetos de Dados

Listas

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Listas

- As listas são coleções ordenadas de objetos.
- A estes objetos damos o nome de *componentes* da lista.
- As componentes de uma lista não têm que ser do mesmo tipo de dados ou tamanho, o que as torna muito flexíveis

Listas

Criação

- As listas podem ser criadas da seguinte forma:

```
lst <- list(id=43534543,nome="Marco",dados=c(peso=62.6,altura=1.65))
lst
```

```
## $id
## [1] 43534543
##
## $nome
## [1] "Marco"
##
## $dados
##   peso altura
##  62.60   1.65
```

Listas

Indexação

- Para aceder ao conteúdo de uma componente da lista podemos usar o seu nome,

```
lst$dados
```

```
##   peso altura
##  62.60   1.65
```

- Podemos também aceder a várias componentes ao mesmo tempo, ou seja a uma sub-lista

```
lst[c('nome','dados')]
```

```
## $nome
## [1] "Marco"
##
## $dados
##   peso altura
##  62.60   1.65
```

Listas

Indexação (2)

- Podemos ainda aceder às componentes individuais das listas pela sua posição na lista, ao estilo dos vetores,

```
lst[[2]]

## [1] "Marco"
```

- Usamos duplos parentesis retos! Parentesis únicos teriam um outro significado no contexto das listas,

```
lst[2]

## $nome
## [1] "Marco"
```

- O resultado é uma lista (efetivamente uma sub-lista de `lst`), enquanto que com os parentesis duplos obtivemos o valor que está nessa componente (que neste caso é uma *string*)

Tipos de Objetos de Dados

Data Frames

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Data Frames

- Os data frames são estruturas semelhantes às matrizes, mas onde as colunas podem ser de diferentes tipos.
- Os data frames podem ser vistos como tabelas de dados com uma linha por observação, composta por vários atributos.
- Muitos cenários são melhor representados por data frames.
- Por exemplo, o estudo de um tratamento sobre um conjunto de indivíduos.

Data Frames

Criação

- Podemos criar um data frame da seguinte forma:

```
estud <- data.frame(
                    id=c("43534543","32456534"),
                    nome=c("Marco","Filipe"),
                    peso=c(62.6,55.4),
                    altura=c(1.65,1.60)
                    )
estud
```

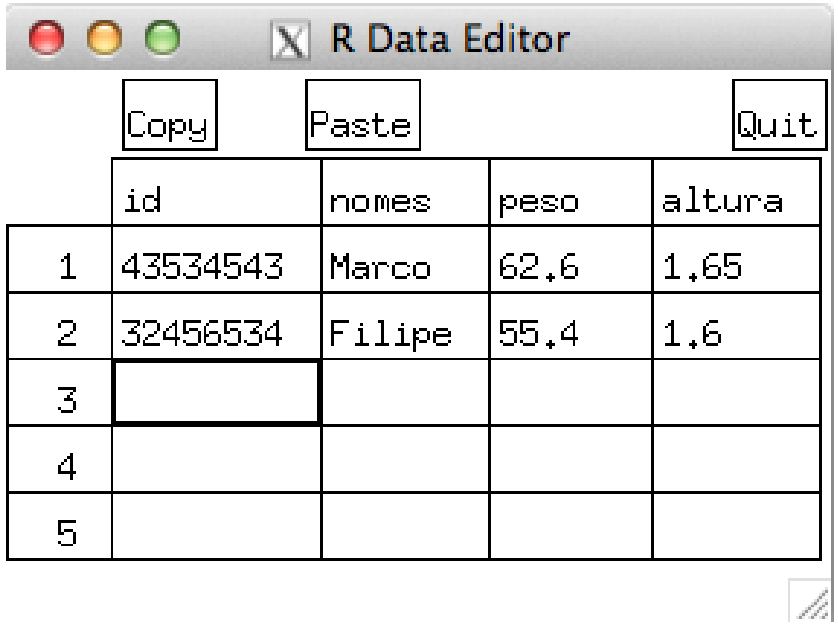
```
##         id   nome peso altura
## 1 43534543  Marco 62.6   1.65
## 2 32456534 Filipe 55.4   1.60
```

Data Frames

Criação (2)

- Se temos demasiados dados para introduzir é mais prático fazê-lo usando um interface tipo folha de cálculo fornecido com o R,

```
> estud <- edit(estud)
```



Data Frames

Indexação

- Os *data frames* são visualizados como uma tabela de dados

```
estud
```

```
##         id   nome peso altura
## 1 43534543  Marco 62.6   1.65
## 2 32456534 Filipe 55.4   1.60
```

- Os dados individuais podem ser acedidos de forma semelhante às matrizes

```
estud[2,3]
```

```
## [1] 55.4
```

```
estud[1,"nome"]
```

```
## [1] Marco
## Levels: Filipe Marco
```

Data Frames

Indexação (2)

- A função `subset()` pode ser usada para facilitar a realização de consultas aos dados

```
estud
```

```
##         id   nome peso altura
## 1 43534543  Marco 62.6   1.65
## 2 32456534 Filipe 55.4   1.60
```

```
subset(estud,peso > 60,nome)
```

```
##    nome
## 1 Marco
```

```
subset(estud,altura < 1.60,c(id,nome))
```

```
## [1] id   nome
## <0 rows> (or 0-length row.names)
```

Tipos de Objetos de Dados

Factors

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Factors

- Um factor é um objeto frequentemente usado para representar variáveis categóricas.
- Para além de guardar o conjunto de valores (como num vetor), guarda também as "categorias"possíveis para essa variável, mesmo que nem todas estejam presentes nos dados.
- Por omissão, as categorias são o conjunto de valores diferentes presente nos dados.
- Em R, estas categorias são designadas por `levels`.

Factors

Criação

- Exemplos de factors:

```
factor(rep(1:3,2))

## [1] 1 2 3 1 2 3
## Levels: 1 2 3

corOlhos <- factor(c("castanha","verde","castanha","azul","castanha"),
             levels=c("azul","castanha","verde"))
corOlhos

## [1] castanha verde    castanha azul     castanha
## Levels: azul castanha verde

levels(corOlhos)

## [1] "azul"     "castanha" "verde"
```

Tipos de Objetos de Dados

Séries Temporais

objeto	tipo de dados (`mode`)	vários tipos?
`vector`	`numeric, character, complex, logical`	não
`matrix`	`numeric, character, complex, logical`	não
`array`	`numeric, character, complex, logical`	não
`list`	`numeric, character, complex, logical, function, expression, ...`	sim
`data.frame`	`numeric, character, complex, logical`	sim
`factor`	`numeric, character`	não
`ts`	`numeric, character, complex, logical`	não

Séries Temporais

- Uma série temporal consiste numa coleção de observações ao longo do tempo.
- Geralmente, a sequência de observações é feita em intervalos de tempo uniformes.
- Podemos dizer que uma série temporal é uma sequência de números coletados em intervalos regulares durante um período de tempo.
- O R possui várias classes de objectos que podem ser usadas para guardar séries temporais
- Vamos usar a infra-estrutura fornecida pela *package* `xts`
  Nota: esta é uma *package* extra que terá que ser instalada antes de poder ser usada.

Séries Temporais

Criação

- Em Inglaterra, foi feito um estudo sobre a evolução das listas de espera em hospitais públicos, entre 2009 e 2010. Tomemos esse exemplo.
- A função `xts()` pode ser usada para criar uma série temporal,

```
library(xts)
listaEsp <- xts(c(621506,615877,625942,633645,624284,614754),
    as.Date(c("2009-10-31","2009-11-30","2009-12-31",
              "2010-01-31","2010-02-28","2010-03-31")))
listaEsp
```

```
##                [,1]
## 2009-10-31 621506
## 2009-11-30 615877
## 2009-12-31 625942
## 2010-01-31 633645
## 2010-02-28 624284
## 2010-03-31 614754
```

Séries Temporais

Criação (2)

- A função `xts` tem dois argumentos: a série temporal e as etiquetas temporais da mesma
- O segundo argumento terá que conter datas
- A função `as.Date()` pode ser usada para converter *strings* em datas
- Se fornecermos uma matriz como primeiro argumento da função `xts()` iremos gerar uma série multivariada em que cada coluna representa uma das variáveis a ser medida em cada instante temporal. Temos pois que fornecer tantas datas quantas as linhas da matriz.

Séries Temporais

Indexação

- Podemos indexar os objectos criados pela função `xts()` da seguinte forma,

```
listaEsp[3]

##                [,1]
## 2009-12-31 625942
```

- Todavia, é bem mais interessante fazer consultas temporais,

```
listaEsp["2010-02-28"]

##                [,1]
## 2010-02-28 624284

listaEsp["2010-02"]

##                [,1]
## 2010-02-28 624284
```

Séries Temporais

Indexação (2)

```
listaEsp["2010-01-31/"]

##                [,1]
## 2010-01-31 633645
## 2010-02-28 624284
## 2010-03-31 614754

listaEsp["2010-01-31/2010-03-31"]

##                [,1]
## 2010-01-31 633645
## 2010-02-28 624284
## 2010-03-31 614754
```

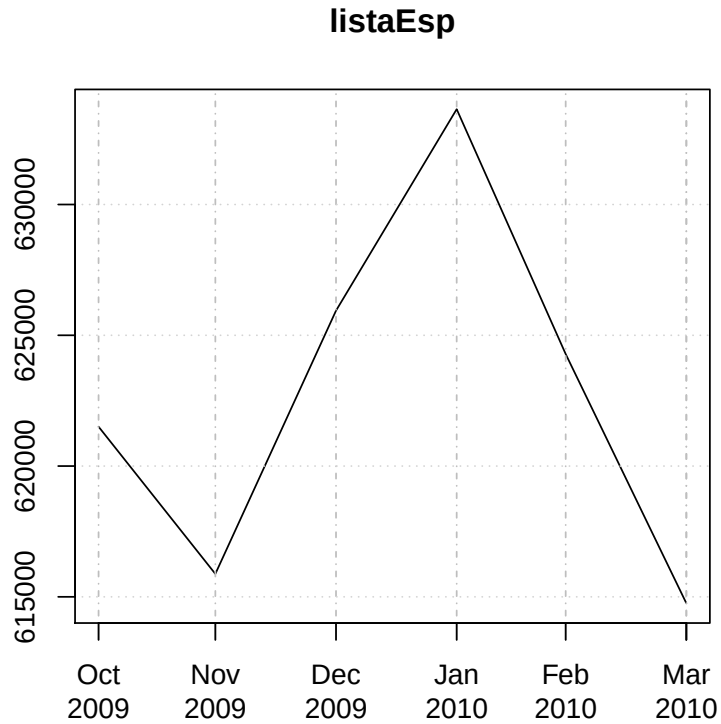
- O índice é uma *string* que pode representar intervalos de datas usando o símbolo / ou omitindo parte da data. Em vez do símbolo / também é possível usar : :

Séries Temporais

Gráficos

- A função `plot()` pode ser usada para visualizar a evolução da série ao longo do tempo
- O R trata de escolher as etiquetas dos eixos mais adequadas,

```
plot(listaEsp)
```



Conversão entre Tipos de Dados

- O R oferece a possibilidade de conversão entre os tipos principais de dados.
- Existem funções que permitem verificar o tipo de dados do objeto e converter para outro tipo de dados.

`is.vector()`	`as.vector()`
`is.numeric()`	`as.numeric()`
`is.character()`	`as.character()`
`is.matrix()`	`as.matrix()`
`is.list()`	`as.list()`
`is.data.frame()`	`as.data.frame()`
`is.factor()`	`as.factor()`
`is.xts()`	`as.xts()`

Conversão entre Tipos de Dados

Alguns Exemplos

- Conversão de um vetor de valores numéricos para valores de texto

```
v <- c(1,2,2,3,4,4,5)
v

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

```
as.character(v)

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

- Conversão de um vetor de valores numéricos para um factor

```
v <- c(1,2,2,3,4,4,5)
v

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

```
as.factor(v)

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

Conversão entre Tipos de Dados

Alguns Exemplos (2)

- Conversão de um vetor para uma lista

```
v <- letters[1:2]
v

## [1] "a" "b"
```

```
as.list(v)

## [[1]]
## [1] "a"
##
## [[2]]
## [1] "b"
```

- Conversão de uma lista para um vetor.

```
l <- list("a"=1,"b"=2)
l

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

```
as.numeric(l)

## [1] 1 2
```

Valores Especiais

- `NA`
- `NULL`
- `Inf` e `-Inf`
- `NaN`

Valores Especiais

NA

- Quando colecionamos um conjunto de valores, nem sempre temos a garantia de que todos eles serão conhecidos.
- Sempre que um valor não é conhecido, o R atribui-lhe um valor especial: `NA` (`Not Available`).
- O resultado de qualquer operação envolvendo um `NA` é também `NA`.

```
v <- c(23,18,NA,21,19)
v*2

## [1] 46 36 NA 42 38

mean(v)

## [1] NA
```

Valores Especiais

NA (2)

- A função `is.na()` retorna o valor `TRUE` para os valores `NA`.

```
v <- c(23,18,NA,21,19)
is.na(v)

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

- Podemos, limitar a aplicação de uma função, por exemplo, aos casos que não são `NA`.

```
v <- c(23,18,NA,21,19)
mean(v[!is.na(v)])

## [1] 20.25

mean(v,na.rm=TRUE)

## [1] 20.25
```

Valores Especiais

NA (3)

- O valor `NA` também pode decorrer de uma conversão de tipos que não seja possível...

```
as.numeric("A")

## Warning:  NAs introduced by coercion

## [1] NA
```

- ... ou da indexação de um valor inexistente.

```
v <- 1:4
v[10]

## [1] NA
```

Valores Especiais

NULL

- Existe também um outro tipo de valor especial que o R usa para representar o nulo: `NULL`
- `NULL` é frequentemente usado em argumentos em funções para representar o facto de nenhum valor ter sido passado a esse argumento.
- A função `is.null()` retorna o valor `TRUE` para os valores `NULL`.

```
f <- function(x=NULL) is.null(x)
f(2)

## [1] FALSE

f()

## [1] TRUE
```

- Nota: `NULL` $\neq$ `NA`

Valores Especiais

Inf e -Inf

- O computador tem uma capacidade para a representação de valores numéricos que é finita.
- Se o resultado de um cálculo for demasiado grande, O R retorna `Inf` para representar $+\infty$ e `-Inf` para representar $-\infty$.

```
2^1024

## [1] Inf

5/0

## [1] Inf
```

- A função `is.infinite()` retorna o valor `TRUE` para os valores `Inf` ou `-Inf`.

```
is.infinite(5/0)

## [1] TRUE
```

Valores Especiais

NaN

- Existe um outro tipo de valor especial que é produzido por cálculos numéricos: `NaN` (`Not a Number`)

```
0/0

## [1] NaN

Inf - Inf

## [1] NaN
```

- A função `is.nan()` retorna o valor `TRUE` para os valores `NaN`.

```
is.nan(0/0)

## [1] TRUE

is.nan(Inf - Inf)

## [1] TRUE
```

Instruções de Controlo de Fluxo

- O interpretador do R executa as instruções sequencialmente.
- As instruções são agrupadas em bloco usando chavetas `{}`; o uso de chavetas é opcional no caso de existir apenas uma instrução.
- O valor de um grupo de instruções é o resultado da última expressão avaliada.
- Quando pretendemos alterar o fluxo de execução dos blocos de instruções, podemos recorrer a:
  - instruções de execução condicional (`if-else`,`switch`);
  - instruções de execução em ciclo (`for`, `while`) .

Instruções de Controlo de Fluxo

Execução Condicional: `if-else`

```
if(cond) expr
```

```
if(cond) expr1 else expr2
```

```
calcIMC <- function(peso,altura) {
  imc <- peso/altura^2
  if(imc < 19.1)
     print("abaixo do peso")
   else
     if(imc >= 19.1 & imc <= 25.8)
        print("peso normal")
     else
        print("acima do peso")
  }
```

```
calcIMC(55,1.70)

## [1] "abaixo do peso"

calcIMC(70,1.70)

## [1] "peso normal"

calcIMC(85,1.70)

## [1] "acima do peso"
```

Instruções de Controlo de Fluxo

Execução Condicional: `switch`

```
switch(expr, ...)
```

```
calcula <- function(funcao,x) {
  fx <- switch(funcao,
  "media" = mean(x),
  "mediana" = median(x),
  "desviopad" = sd(x)
  )
  print(paste("resultado = ",fx))
}
```

```
calcula("media",1:10)

## [1] "resultado =  5.5"

calcula("desviopad",1:10)

## [1] "resultado =  3.02765035409749"

calcula("mediana",1:10)

## [1] "resultado =  5.5"
```

Instruções de Controlo de Fluxo

Execução em Ciclo: `for`

```
for (var in seq) expr
```

```
parimparF <- function(vals) {
  n <- length(vals)
  for(i in 1:n) {
    if(vals[i] %% 2 == 0)
       print(paste(vals[i],"é par"))
    else
       print(paste(vals[i],"é ímpar"))
  }
}
```

```
parimparF(1:4)

## [1] "1 é ímpar"
## [1] "2 é par"
## [1] "3 é ímpar"
## [1] "4 é par"

parimparF(sample(10,5))

## [1] "1 é ímpar"
## [1] "2 é par"
## [1] "7 é ímpar"
## [1] "4 é par"
## [1] "8 é par"
```

Instruções de Controlo de Fluxo

Execução em Ciclo: `while`

```
while (cond) expr
```

```
parimparW <- function(vals) {
  n <- length(vals)
  i <- 1
  while(i <= n) {
     if(vals[i] %% 2 == 0)
       print(paste(vals[i]," é par"))
     else
       print(paste(vals[i]," é ímpar"))
     i <- i + 1
  }
}
```

```
parimparW(1:4)

## [1] "1  é ímpar"
## [1] "2  é par"
## [1] "3  é ímpar"
## [1] "4  é par"

parimparW(sample(10,5))

## [1] "9  é ímpar"
## [1] "5  é ímpar"
## [1] "10  é par"
## [1] "7  é ímpar"
## [1] "1  é ímpar"
```

Ainda sobre Funções

Âmbito das Variáveis

- As variáveis definidas dentro da função são variáveis locais, isto é, não são visíveis fora da função. Os parâmetros duma função também são variáveis locais.
- As variáveis definidas fora das funções são variáveis globais e são sempre visíveis.

```
calcIMC <- function(peso,altura) {
  imc <- peso/altura^2
  if(imc < 19.1)
     print("abaixo do peso")
   else
     if(imc >= 19.1 & imc <= 25.8)
        print("peso normal")
     else
        print("acima do peso")
  }
```

```
p <- 60; a <- 1.65
calcIMC(p,a)

## [1] "peso normal"

p # variável global

## [1] 60

imc # variável local de calcIMC

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

Ainda sobre Funções

Valor de Retorno

- O valor de retorno de uma função pode ser um qualquer objeto em R.
- Esse valor é explicitado pela função `return()`. Esta função faz com que a execução da função **termine, de imediato**, e retorne o valor indicado para o ponto onde foi chamada a função.
- No caso de não existir a chamada à função `return()`, o R retorna o valor da última expressão avaliada na execução da função.

```
f <- function(x,a=1,b=2) {
  return(a*x + b)
}
f(4)

## [1] 6
```

```
f <- function(x,a=1,b=2) {
  a*x + b
}
f(4)

## [1] 6
```

Ainda sobre Funções

Valor de Retorno (2)

- Numa função só pode existir um valor de retorno.
- Se desejarmos retornar mais do que um valor, podemos colecionar os valores de retorno num objeto (vetor, lista, etc.) e, depois, retornar esse objeto.

```
f <- function(x,a=1,b=2) {
  y <- a*x + b
  c(x=x,y=y)
}
f(4)

## x y
## 4 6
```

Ainda sobre Funções

Vetorização de Funções

- Vimos que o R, tal como qualquer linguagem de programação, possui instruções de ciclo.
- Contudo, comparativamente a outras linguagens de programação, estas instruções de ciclo são muito menos utilizadas em R.
- O R oferece uma alternativa mais eficiente:a vetorização de operações e também de funções.
- A vetorização de funções permite a aplicação de uma função aos elementos de um dado tipo de objeto.
- A diferença a nível de eficiência é, especialmente, notória para grandes quantidades de dados.

Ainda sobre Funções

Vetorização de Funções: `apply`

- A função `apply(X,MARGIN,FUN,...)` permite a aplicação de uma função (`FUN`) a um array ou matriz (`X`) por linha (`MARGIN=1`), coluna (`MARGIN=2`) ou ambas (`MARGIN=c(1,2)`).

```
m <- matrix(1:8,nrow=2,ncol=4,
            byrow=TRUE)
m

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8

apply(m,1,sum) # por linha

## [1] 10 26

apply(m,2,sum) # por coluna

## [1]  6  8 10 12
```

```
parimpar <- function(x) {
  if(x %% 2 == 0)
     return("par")
  else
     return("ímpar")
}
apply(m,c(1,2),parimpar)

##      [,1]    [,2]  [,3]    [,4]
## [1,] "ímpar" "par" "ímpar" "par"
## [2,] "ímpar" "par" "ímpar" "par"
```

Ainda sobre Funções

Vetorização de Funções: `lapply`

- A função `lapply(X,FUN,...)` aplica uma dada função (`FUN`) sobre cada um dos elementos de uma lista, data frame ou vetor `X` e retorna os resultados numa lista.

```
l <- list(a="Ola",b=3:7,
     c=data.frame(X=10:11,Y=LETTERS[10:11]))
l
```

```
## $a
## [1] "Ola"
##
## $b
## [1] 3 4 5 6 7
##
## $c
##    X Y
## 1 10 J
## 2 11 K
```

```
lapply(l,mode)
```

```
## $a
## [1] "character"
##
## $b
## [1] "numeric"
##
## $c
## [1] "list"
```

Ainda sobre Funções

Vetorização de Funções: `sapply`

- A função `sapply(X,FUN,...)` é semelhante à `lapply()`, mas tenta simplificar o resultado a uma estrutura de dados. mais elementar.

```
l <- list(a="Ola",b=3:7,c=data.frame(X=10:11,Y=LETTERS[10:11]))
sapply(l,mode)

##           a           b           c
## "character"   "numeric"      "list"

sapply(1:8,parimpar)

## [1] "ímpar" "par"   "ímpar" "par"   "ímpar" "par"   "ímpar" "par"
```