Introdução ao Software R

CEDE - UFMG

Dezembro – 2016

Universidade Federal de Minas Gerais

Conteúdo

- Módulo I: Introdução
- Módulo II: Estrutura de dados
- Módulo III: Loops e condições
- Módulo IV: Importando e exportando arquivo texto
- Módulo V: Funções

Módulo I: Conceitos Básicos

Introdução

Baixando e Instalando o R.

- Link para o site oficial do R: `www.r-project.org`.

Download

1. Clique no link CRAN (Comprehensive R Archive Network) na seção *Download*, *Packages*;
2. Escolha um repositório, por exemplo, UFPR;
3. Escolha o link de acordo com o sistema operacional do seu computador, (Ex.: Windows);
4. Escolha a opção *base*, pois as demais são para desenvolvimento de pacotes R;
5. Finalmente clique para baixar o R.

A instalação do R pode ser realizada escolhendo sempre as configurações padrões.

Pacotes, manual e demonstrações

- A versão base do R possui uma coleção enorme de funções:
  - Modelos Estatísticos
  - Algoritmos Computacionais
  - Métodos Matemáticas
  - Visualização de Dados

Pacotes, manual e demonstrações

- A versão base do R possui uma coleção enorme de funções:
  - Modelos Estatísticos
  - Algoritmos Computacionais
  - Métodos Matemáticas
  - Visualização de Dados

Mas as vezes não é suficiente =/!

Pacotes, manual e demonstrações

- A versão base do R possui uma coleção enorme de funções:
  - Modelos Estatísticos
  - Algoritmos Computacionais
  - Métodos Matemáticas
  - Visualização de Dados

Mas as vezes não é suficiente =/!

Pacotes!

Pacotes

- Assim como alguns softwares estatísticos, o R também é extensível através de "módulos". Em R estes módulos são chamados de pacotes, bibliotecas ou packages.

Pacotes

Uma coleção de funções que podem ser escritas em R, C++, Fortran e C e que são chamadas diretamente de dentro do R.

- Um pacote inclui: as funções, dados para exemplificar as funcionalidades do pacote, arquivo com ajuda (help) para cada função, e uma descrição do pacote.
- Qualquer pessoa pode desenvolver seus pacotes e então submeter ao CRAN, disponibilizar através do GitHub ou *standalone*.

Pacotes

- As funcionalidades do R, podem ser ampliadas carregando estes pacotes, tornando um software ainda mais poderoso, capaz de realizar inúmeras tarefas:
  - Análise multivariada;
  - Análise Bayesiana;
  - Manipulação de dados;
  - Gráficos a nível de publicação;
  - Big Data, Deep Learning;
  - Processamento de imagens.

Exemplos de pacotes

Alguns pacotes

- `maptools:` Funções para leitura, exportação e manipulação de estruturas espaciais.
- `cluster:` Funções para análise de clusters.
- `ggplot2:` Criação de gráficos elegantes.
- `rmarkdown:` criação de documentos (dinâmicos) em PDF, Word, HTML.
- `nlme:` Modelos lineares e não-lineares de efeitos mistos.

- O R possui mais de ???? pacotes, e milhares de funções.

Instalando pacotes

- Para instalar um pacote do R que já esteja no CRAN basta usar o comando:

```
> install.packages('ggplot2')
```

- Além da opção de comando, também podemos instalar pacotes utilizando os menus do R (Pacotes -> Instalar pacotes), ou do RStudio (Tools -> Install Packages ...).
- Temos também a opção de instalar pacotes a partir de arquivos *.zip* ou *tar.gz* (Pacotes -> Instalar pacotes a partir de zip locais) ou utilizando o Rstudio (Tools -> Install Packages ... -> Install From)

Carregando pacotes

- Uma vez que o pacote foi instalado não há mais a necessidade de instalar sempre que for utilizar as suas funcionalidades, basta carregar o pacote com os comandos: `library()` ou `require()`.

```
> library(cluster) # ou
> require(cluster)
```

Help?

- Para conhecer quais as funções disponíveis no pacote, faça:

```
> help(package = "survey")
```

- Para pedir ajuda de uma determinada função:

```
> ?glm #forma mais comum de acessar o manual da função
> help("glm")
```

- Obtendo ajuda na internet:

```
> help.search("t.test")
```

Help?

- Procurando por alguma função, mas esqueci o nome:

```
> apropos("lm")
> ??lm #??: procurar em todos os pacotes instalados no R
```

- Para todas as outras coisas existe o **Google**!
- Para algumas demonstrações da capacidade gráfica do R:

```
> demo(graphics)
> demo(persp)
> demo(Hershey)
> demo(plotmath)
```

Linguagem compilada vs interpretada

Compilada ou interpretada?

Linguagem compilada vs interpretada

Compilada ou interpretada?

- Essencialmente o R é uma linguagem de programação interpretada.

Linguagem compilada vs interpretada

Compilada ou interpretada?

- Essencialmente o R é uma linguagem de programação interpretada.
- Porém...

Linguagem compilada vs interpretada

Compilada ou interpretada?

- Essencialmente o R é uma linguagem de programação interpretada.
- Porém...
- É mais correto vê-lo como uma interface para código compilado.

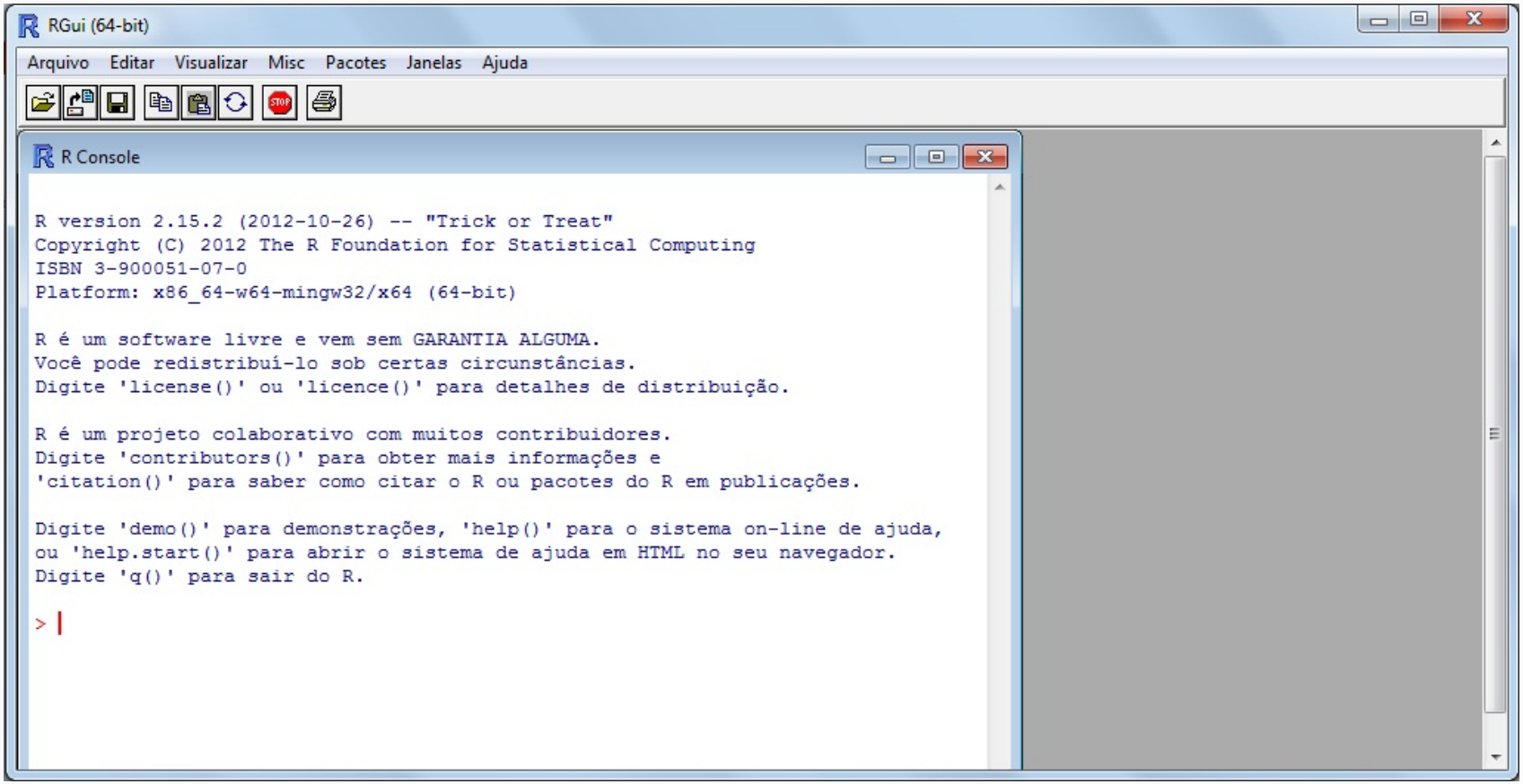
Linguagem compilada vs interpretada

Compilada ou interpretada?

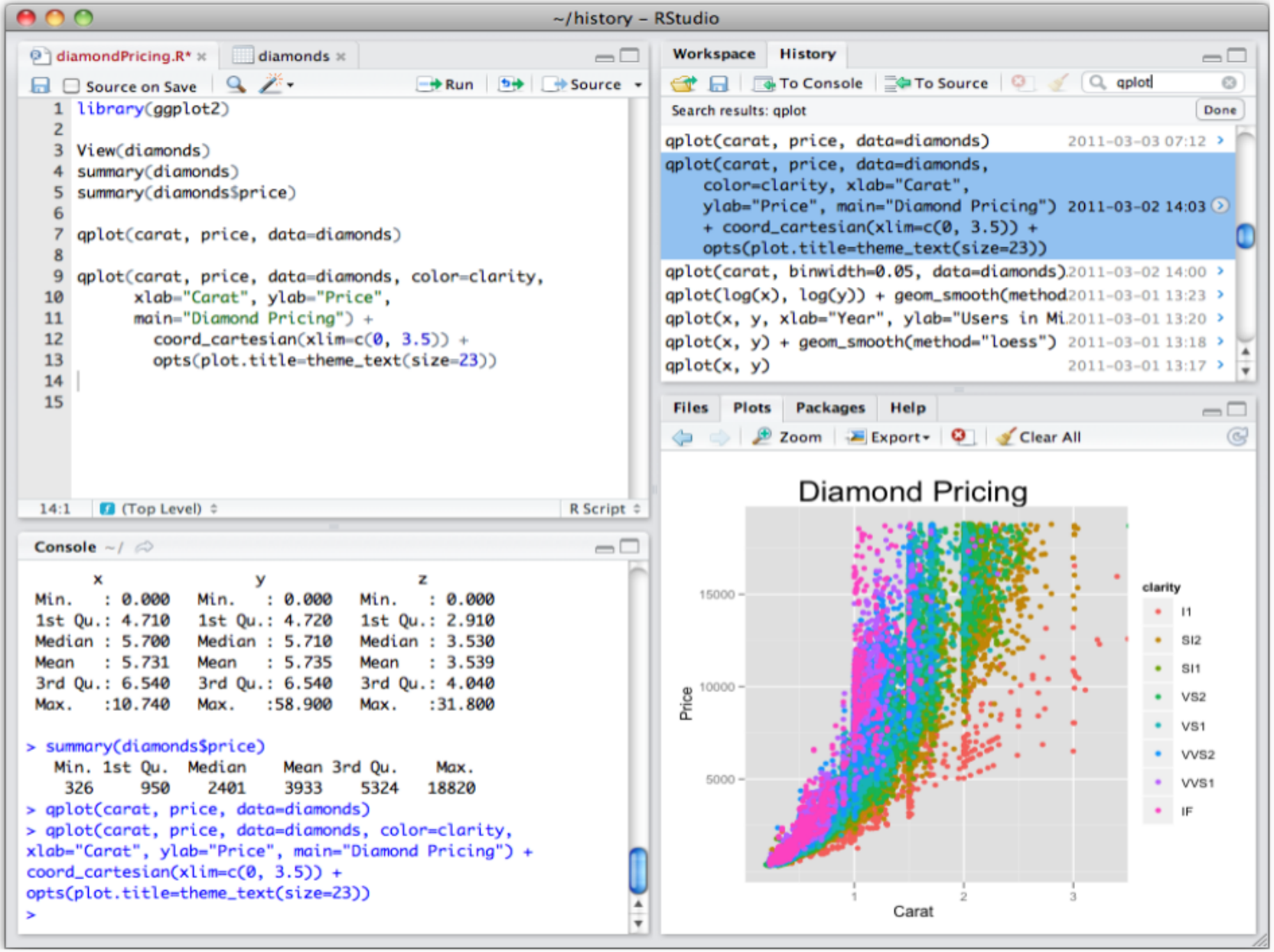
- Essencialmente o R é uma linguagem de programação interpretada.
- Porém...
- É mais correto vê-lo como uma interface para código compilado.
- As principais rotinas são executadas em código compilado (`.C`, `.Call.`, `.Internal`, `.Primitive`)

Ambiente R

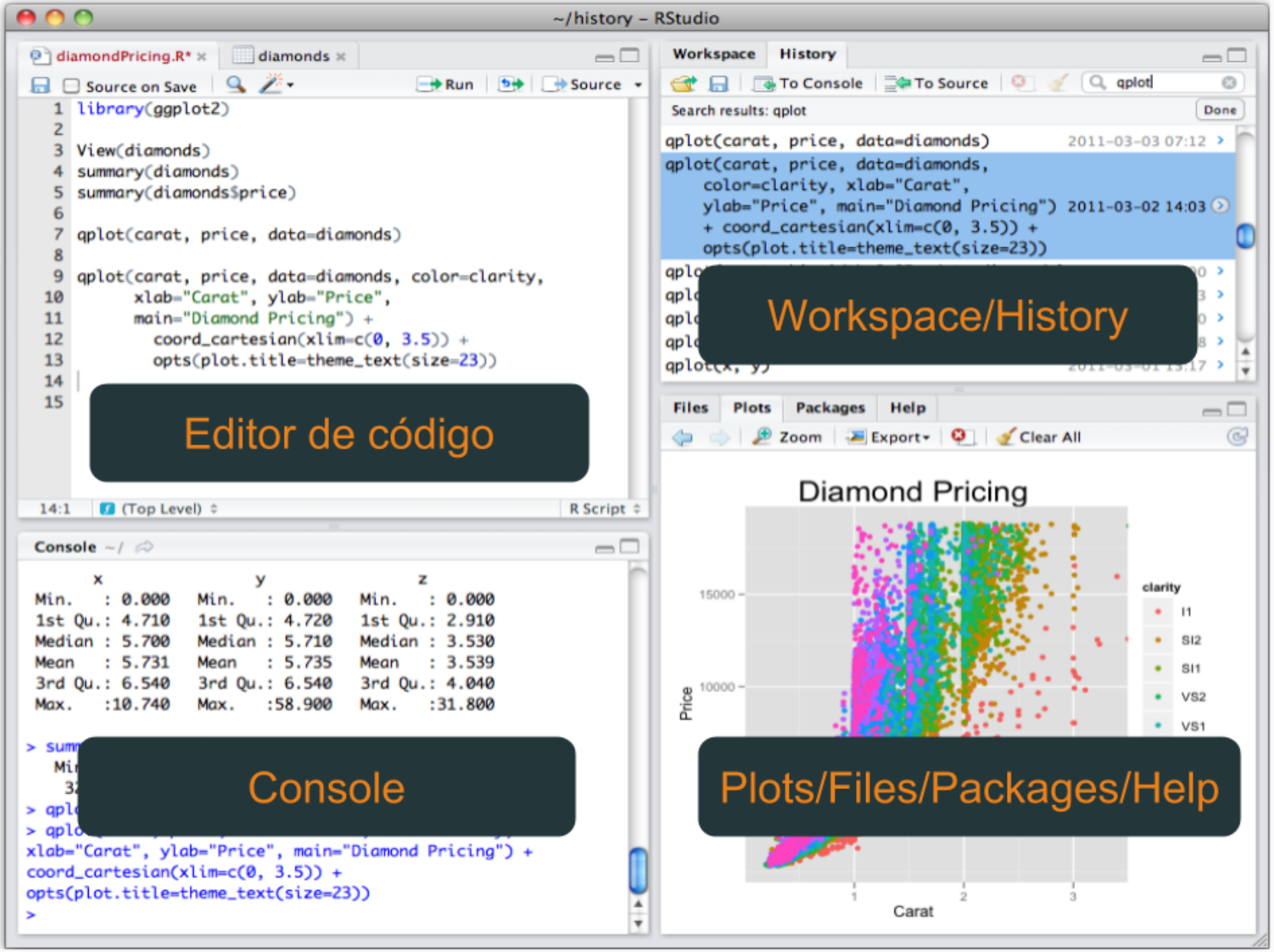
Interface R


RGui (64-bit)
Arquivo Editar Visualizar Misc Pacotes Janelas Ajuda
R Console
R version 2.15.2 (2012-10-26) -- "Trick or Treat"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-w64-mingw32/x64 (64-bit)
R é um software livre e vem sem GARANTIA ALGUMA.
Você pode redistribuí-lo sob certas circunstâncias.
Digite 'license()' ou 'licence()' para detalhes de distribuição.
R é um projeto colaborativo com muitos contribuidores.
Digite 'contributors()' para obter mais informações e
'citation()' para saber como citar o R ou pacotes do R em publicações.
Digite 'demo()' para demonstrações, 'help()' para o sistema on-line de ajuda,
ou 'help.start()' para abrir o sistema de ajuda em HTML no seu navegador.
Digite 'q()' para sair do R.
>

IDE Rstudio


~/history – RStudio
diamondPricing.R*
diamonds
Source on Save
Run
Source
1 library(ggplot2)
2
3 View(diamonds)
4 summary(diamonds)
5 summary(diamonds$price)
6
7 qplot(carat, price, data=diamonds)
8
9 qplot(carat, price, data=diamonds, color=clarity,
10 xlab="Carat", ylab="Price",
11 main="Diamond Pricing") +
12 coord_cartesian(xlim=c(0, 3.5)) +
13 opts(plot.title=theme_text(size=23))
14
15
14:1
(Top Level)
R Script
Console ~/
x y z
Min. : 0.000 Min. : 0.000 Min. : 0.000
1st Qu.: 4.710 1st Qu.: 4.720 1st Qu.: 2.910
Median : 5.700 Median : 5.710 Median : 3.530
Mean : 5.731 Mean : 5.735 Mean : 3.539
3rd Qu.: 6.540 3rd Qu.: 6.540 3rd Qu.: 4.040
Max. :10.740 Max. :58.900 Max. :31.800
> summary(diamonds$price)
Min. 1st Qu. Median Mean 3rd Qu. Max.
326 950 2401 3933 5324 18820
> qplot(carat, price, data=diamonds)
> qplot(carat, price, data=diamonds, color=clarity,
xlab="Carat", ylab="Price", main="Diamond Pricing") +
coord_cartesian(xlim=c(0, 3.5)) +
opts(plot.title=theme_text(size=23))
>
Workspace
History
To Console
To Source
qplot
Search results: qplot
Done
qplot(carat, price, data=diamonds) 2011-03-03 07:12
qplot(carat, price, data=diamonds, color=clarity, xlab="Carat", ylab="Price", main="Diamond Pricing") + coord_cartesian(xlim=c(0, 3.5)) + opts(plot.title=theme_text(size=23)) 2011-03-02 14:03
qplot(carat, binwidth=0.05, data=diamonds) 2011-03-02 14:00
qplot(log(x), log(y)) + geom_smooth(method 2011-03-01 13:23
qplot(x, y, xlab="Year", ylab="Users in Mi 2011-03-01 13:20
qplot(x, y) + geom_smooth(method="loess") 2011-03-01 13:18
qplot(x, y) 2011-03-01 13:17
Files
Plots
Packages
Help
Zoom
Export
Clear All
Diamond Pricing
Price
15000
10000
5000
1
2
3
Carat
clarity
I1
SI2
SI1
VS2
VS1
VVS2
VVS1
IF

IDE Rstudio


~/history – RStudio
diamondPricing.R*
diamonds
Source on Save
Run
Source
1 library(ggplot2)
2
3 View(diamonds)
4 summary(diamonds)
5 summary(diamonds$price)
6
7 qplot(carat, price, data=diamonds)
8
9 qplot(carat, price, data=diamonds, color=clarity,
10 xlab="Carat", ylab="Price",
11 main="Diamond Pricing") +
12 coord_cartesian(xlim=c(0, 3.5)) +
13 opts(plot.title=theme_text(size=23))
14
15
Editor de código
14:1
(Top Level)
R Script
Console ~/
x y z
Min. : 0.000 Min. : 0.000 Min. : 0.000
1st Qu.: 4.710 1st Qu.: 4.720 1st Qu.: 2.910
Median : 5.700 Median : 5.710 Median : 3.530
Mean : 5.731 Mean : 5.735 Mean : 3.539
3rd Qu.: 6.540 3rd Qu.: 6.540 3rd Qu.: 4.040
Max. :10.740 Max. :58.900 Max. :31.800
> summ
Mi
3
> qplo
> qplo
xlab="Carat", ylab="Price", main="Diamond Pricing") +
coord_cartesian(xlim=c(0, 3.5)) +
opts(plot.title=theme_text(size=23))
>
Console
Workspace
History
To Console
To Source
qplot
Search results: qplot
Done
qplot(carat, price, data=diamonds) 2011-03-03 07:12
qplot(carat, price, data=diamonds,
color=clarity, xlab="Carat",
ylab="Price", main="Diamond Pricing") 2011-03-02 14:03
+ coord_cartesian(xlim=c(0, 3.5)) +
opts(plot.title=theme_text(size=23))
Workspace/History
Files
Plots
Packages
Help
Zoom
Export
Clear All
Diamond Pricing
Price
15000
10000
Carat
1
2
3
clarity
I1
SI2
SI1
VS2
VS1
Plots/Files/Packages/Help

Outras IDE's

IDE's para o R

Outras IDE's

IDE's para o R

- Emacs Speaks Statistics - ESS
- StatET: plugin para o Eclipse.
- TINN-R

Manipulação simples no prompt

A forma mais direta de interagir com o R é através das linhas de comandos.

- Os comandos são digitados no prompt $>$.
- Continuação da linha é indicado por $+$.
- Para submeter os comandos pressione **Enter**.
- Para inserir vários comandos na mesma linha, utilize **;**.

Manipulação simples no prompt

```
> 2 + 3
> 1 - 8
> 4 * 5
> 3 / 5
> 2 ^ 3
```

Use os parênteses para calcular expressões, por exemplo, $\left(\frac{20+7}{3}\right)^2$.

```
> ((20 + 7)/3)^2
[1] 81
```

Manipulação simples no prompt

- O R ignora os espaços em brancos excessivos.
- String/caracteres devem ser inseridos entre aspas simples ou dupla: ' ' ou " "

```
> 18              /       3
[1] 6
```

```
> 25 *
+
+   5
[1] 125
```

```
> "Eu sou uma
+ string quebrada e entre aspas duplas"
[1] "Eu sou uma\nstring quebrada e entre aspas duplas"
```

- Note o símbolo de quebra de linha `\n`.
- Note o símbolo + que indica a continuação do comando.

Criando objetos/variáveis

Princípio 1: Tudo que existe no R é um objeto.

- Para atribuir valores a obejtos, basta usar o operador <-, o qual é a combinação do operador < com −. Como alternativa, podemos utilizar o operador =.

```
> objeto1 <- 3*9
> objeto2 = 8+2
```

- Visualizar o valor armazenado em um objeto, basta digitar o nome do objeto no prompt então apertar **Enter**. Ou usar a função `print()`, ou ainda entre parênteses `(objeto1)`

```
> objeto1
[1] 27
```

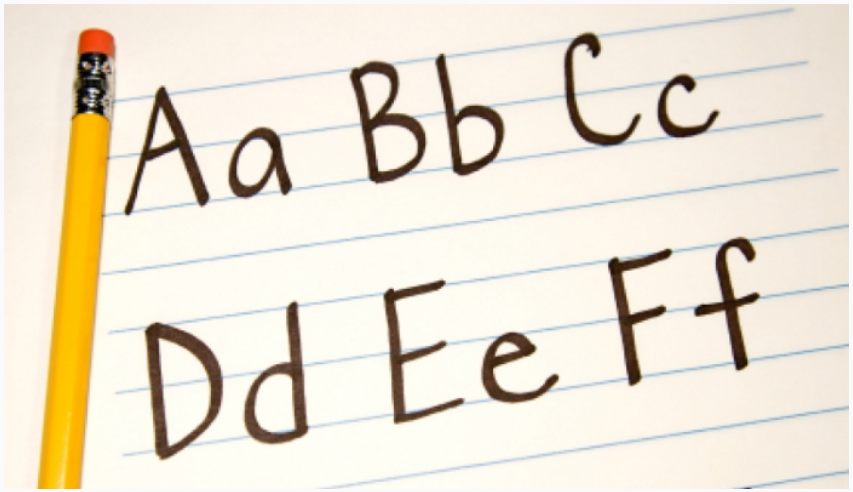
```
> print(objeto1)
[1] 27
```

Case-sensitive

- Assim como a maioria das linguagens de programação o R também é sensível à letras minúsculas e maiúsculas.

```
> (foo <- "todas as letras sao minusculas")
[1] "todas as letras sao minusculas"
```

```
> (FOO <- "todas as letras sao maiusculas")
[1] "todas as letras sao maiusculas"
```

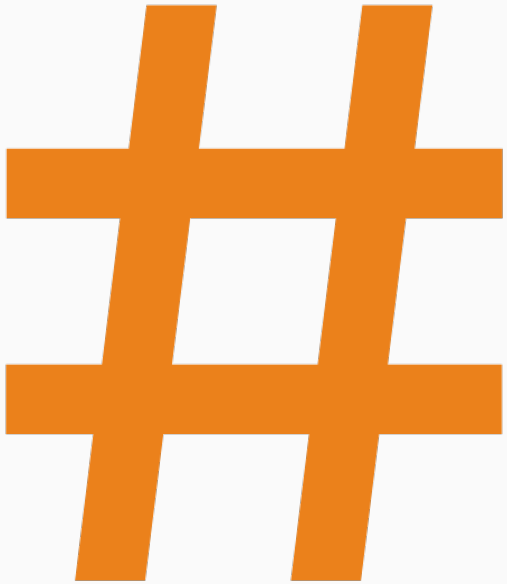


Comentários

- Comentários em R podem ser inseridos depois do caractere #. Desta forma, qualquer comando após o caractere # não será executado.
  - Ex.

```
> 4*2
[1] 8
```

```
> #2*2 Olá eu sou um comentário =)
```



Agora é a sua vez

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 1,5 metros de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 1,5 metros de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

Dica

$$Volume = \pi \times raio^2 \times altura$$

$$\pi = 3.14$$

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

- Notaram alguma coisa diferente no cálculo do volume?

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

- Notaram alguma coisa diferente no cálculo do volume?
- Onde o objeto π foi declarado?

Criando um objeto

Volume de um tubo

Seja um tubo com raio de 10 cm, com 70 cm de comprimento e com uma espessura de 1 cm. Qual o volume deste cubo?

```
> raio <- 10
> espessura <- 1
> comprimento <- 70
> volume <- pi*(raio - espessura)^2*comprimento #calcula o volume do cubo
> volume
[1] 17812.83
```

- Notaram alguma coisa diferente no cálculo do volume?
- Onde o objeto π foi declarado?
- O R armazena algumas quantidades importantes.

Constantes armazenadas no R

```
> pi
[1] 3.141593
```

```
> letters
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
[21] "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" "R" "S" "T"
[21] "U" "V" "W" "X" "Y" "Z"
```

```
> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
```

```
> month.name
 [1] "January"   "February"  "March"     "April"     "May"       "June"
 [7] "July"      "August"    "September" "October"   "November"  "December"
```

```
> Inf
[1] Inf
```

Operadores lógicos

Operadores lógicos

Operadores lógicos: são operados binários para realização de testes entre duas variáveis (objetos). Estas operações retornam o valor `TRUE` (1) ou `FALSE` (0).

Operadores	Descrição
`<`	Menor que
`<=`	Menor ou igual a
`>`	Maior que
`>=`	Maior ou igual a
`==`	Igual a
`!=`	Diferente de
`!x`	Não x
`x | y`	x OU y
`x & y`	x E y

Table 1: Tabela de operadores lógicos.

Usando os operadores lógicos

Exemplos:

```
> x <- 10 #atribuindo o valor 10 ao objeto x
> y <- 20 #atribuindo o valor 20 ao objeto y
> x < y   #x é menor que y?
> x < x   #x é menor que x?
> x <= x  #x é menor igual que x?
> x > y   #x é maior que x?
> x >= y  #x é maior igual que x?
> x == y  #x é igual a y?
> x != y  #x é diferente de y?
```

Dica

Note que há um espaço em branco entre os operadores lógicos. Estes espaços não são obrigatórios, porém tornam o código mais legível.

Agora é a sua vez

Agora é a sua vez

Teste lógico com string

Crie dois objetos em R: um que amarzene a primeira letra do seu primeiro nome e outro com a primeira letra do seu segundo nome. Agore compare estes objetos usando alguns dos operadores lógicos. Por exemplo o operador <=.

Agora é a sua vez

Teste lógico com string

Crie dois objetos em R: um que amarzene a primeira letra do seu primeiro nome e outro com a primeira letra do seu segundo nome. Agore compare estes objetos usando alguns dos operadores lógicos. Por exemplo o operador <=.

```
> #Luís Gustavo
> primeira_letra_do_meu_primeiro_nome <- 'L'
> primeira.letra.do.meu.segundo.nome  <- 'G'
> primeira_letra_do_meu_primeiro_nome <= primeira.letra.do.meu.segundo.nome
```

Agora é a sua vez

Teste lógico com string

Crie dois objetos em R: um que amarzene a primeira letra do seu primeiro nome e outro com a primeira letra do seu segundo nome. Agore compare estes objetos usando alguns dos operadores lógicos. Por exemplo o operador <=.

```
> #Luís Gustavo
> primeira_letra_do_meu_primeiro_nome <- 'L'
> primeira.letra.do.meu.segundo.nome  <- 'G'
> primeira_letra_do_meu_primeiro_nome <= primeira.letra.do.meu.segundo.nome
```

Assimilando

- Strings são tratadas com aspas.
- O nome do objeto pode ser tão longo quanto você queira.
- Podemos usar os caracteres _ e . nos nomes dos objetos.
- Não podem iniciar o números: `1luis <- 2`.

Classes de objetos

Classes de objetos

- Em uma análise estatística existem diferentes tipos de dados: numéricos, categóricos, ordinais, univariados, bivariados, multivariados, etc.
- R possui diferentes classes para acomodar estas diferentes natureza dos dados.

Classes

- `numeric()`: números com casas decimais (double). Ex.: 2.1.
- `integer()`: números inteiros. Ex.: 5L.
- `logical()`: `TRUE` ou `FALSE`.
- `character()`: caracteres/strings. Ex.: "Hello!"

Classe: `numeric()`

- Para saber a classe de um objeto devemos utilizar a função `class()`. É recomendado ler o help das funções que iremos aprender neste treinamento. `?class()`.

```
> x <- 12.5
> class(x)
[1] "numeric"
```

```
> y <- 10
> class(y)
[1] "numeric"
```

```
> ?class()
> ?numeric()
```

- Podemos declarar um vetor da classe númerico usando a função `numeric()`. Mais adiante iremos entender o conceito de vetor.

```
> vetor_numerico <- numeric(length = 10)
```

Classe: integer()

- Para criar um objeto da classe integer devemos utilizar o operador **L**.

```
> inteiro <- 50L
> class(inteiro)
[1] "integer"
```

```
> ?integer()
```

Classe: `logical()`

- Objetos da classe `logical` podem ser obtidos através da comparação entre variáveis (objetos) e assumem apenas os valores `TRUE` (T) ou `FALSE` (F).

```
> logico <- 2 < 3 #dois é menor que três?
> class(logico)
[1] "logical"
```

```
> (logico <- F) #posso sobrescrever o objeto
[1] FALSE
```

- Operadores lógicos também podem ser aplicados a objetos da classe `logical`.

```
> u <- TRUE; v <- FALSE     #criando objetos
> u & v                     #u E v
> u | v                     #u OU v
> !v                        #negação de v
```

Classe: character()

- `character()`: objetos do tipo `character` são utilizados para representar *{strings} no R. Ou seja, variáveis de natureza textual.*

```
> nome <- "Gov. Valadares" #criando objeto
> class(nome)              #classe do objeto nome
[1] "character"
```

```
> toupper(nome)            #todas maiusculas
[1] "GOV. VALADARES"
```

```
> tolower(nome)            #todas minusculas
[1] "gov. valadares"
```

Classe: character()

- No R existe uma infinidade de funções para manipular strings, além de conseguir interpretar as famosas expressões regulares.

Algumas funções para string

- paste(): concatena strings.
- grep(): números inteiros. Ex.: 5L.
- gsub(): TRUE ou FALSE.
- substr(): caracteres/strings. Ex.: "Hello!"

```
> gv_uf <- paste('Gov.          Valadares', 'MG', sep = ' - ')
> gv_uf <- sub(" +", " ", gv_uf)
> gv_longo  <- gsub(pattern = 'Gov.', replacement = 'Governador', x = gv_uf)
> posicao   <- regexpr(pattern = ' - ', text = gv_longo)
> gv_sem_uf <- substr(x = gv_longo, start = 1, stop = posicao - 1)
```

Classes de objetos

- As funções do tipo `as.CLASSE` são utilizadas para atribuir uma classe ao objeto. Ou seja, elas tentam forçar um obejto ser da classe `CLASSE`.
- Já as funções `is.CLASSE` testam se o objeto é da classe `CLASSE`. Neste caso, o retorno desta função um objeto da classe `logical()`.

```
> as.integer(pi)
[1] 3
```

```
> is.integer(3.14)
[1] FALSE
```

```
> as.integer("5.45")
[1] 5
```

```
> as.integer("Minas Gerais")
[1] NA
```

```
> is.integer("Brasil")
[1] FALSE
```

```
> as.integer(TRUE)
[1] 1
```

Agora é a sua vez

Agora é a sua vez

Teste lógico com string

- Faça `time <- 'Democr56ata'`
- Qual a classe do objeto `time`?
- Utilize a função `substr()` para obter o número que está no objeto `time` e atribua este valor em um outro objeto. Ex.: `numero <- substr(???)`.
- Qual a classe do objeto `numero`? Transforme-o para a classe adequada.

Agora é a sua vez

Teste lógico com string

- Faça `time <- 'Democr56ata'`
- Qual a classe do objeto `time`?
- Utilize a função `substr()` para obter o número que está no objeto `time` e atribua este valor em um outro objeto. Ex.: `numero <- substr(???)`.
- Qual a classe do objeto `numero`? Transforme-o para a classe adequada.

```
> time <- 'Democr56ata'
> class(time)
> numero <- as.numeric(substr(time, 7, 8))
> class(numero)
```

Módulo II: Estrutura de dados

Estrutura de dados

- Até este momento estamos apenas trabalhando com objetos escalares, ou seja, com um único valor. Agora em diante iremos conhecer estruturas de armazenamento de dados em R.
- Dados: são as informações obtidas de uma unidade experimental ou *observacional*.
- Exemplo: 'Estes são tubos de aços produzidos na empresa XYZ com espessura de 1 cm e comprimento 80 cm.'

```
> dados
    id fabricante espessura comprimento
1 0001         XYZ     1.000      80.001
2 0002         ABC     0.988      79.999
3 0003         KML     1.001      81.001
```

Estrutura de dados

No `R` existem várias estruturas para armazenar dados. Desde de dados que podem ser modelados em uma tabela, quando dados de de natureza textual ou espacial. Abaixo é listado as estruturas mais comuns para iniciar na linguagem `R`.

- **Vetores**: `c()`, estrutura unidimensional;
- **Matrizes**: `matrix()`, estrutura bidimensional;
- **Arranjos** (*arrays*): `array()`, é uma generalização de matriz, por exemplo, um cubo;
- **Listas**: `list()`, a estrutura de dados mais genérica do `R`;
- **Data frames**: `data.frame()`, caso especial de uma lista;

Vetores

- **vetores**: um vetor é um sequência de dados do mesmo tipo ou classe. Ou seja, um vetor só pode conter valores númericos, ou lógicos, ou de caracteres.
- **Atenção**: nunca um vetor será composto por um valor numérico e lógico ao mesmo tempo.
- **Exemplo**: usando a função `c()` para criar um vetor. Esta função tem como argumento os elementos que irão compor o nosso vetor, e sua tarefa é concatenar todos os elementos em um único objeto.

```
> (numerico.vet <- c(1, 3, 1, 9))
[1] 1 3 1 9
```

```
> (logico.vet <- c(T, TRUE, FALSE, T))
[1]  TRUE  TRUE FALSE  TRUE
```

```
> (caractere.vet <- c('a', 'cc', 'dd'))
[1] "a"  "cc" "dd"
```

```
> ?c()
```

- **Tarefa:** encontra a classe de cada um dos objetos criados acima.

Vetores: atributos

- Estas novas estruturas possuem alguns atributos que são de nosso interesse. Por exemplo, sua classe, comprimento do vetor, nomes de cada posição do vetor, etc.
- Para sabermos a classe de um objeto, já sabemos que basta aplicar a função `class()` no objeto.
- O comprimento de um vetor, ou quantos elementos este vetor possui, pode ser obtido através da função `length()`.

```
> length(numerico.vet)
[1] 4
```

```
> ?length #consulte o help desta função
```

Vetores: atributos

- O atributo de nomes de cada elemento pode ser acessado com a função `names()`.

```
> xx <- c(1, 2, 3)
> names(xx) #atributo vazio
NULL
```

```
> names(xx) <- c('Posicão 1', 'Posição 2', 'Posição 3')
> xx
Posicão 1 Posição 2 Posição 3
        1         2         3
```

Combinando vetores

- Podemos combinar/concatenar vetores usando a mesma função `c()`.
- **Exemplo**: vamos combinar os vetores `vet1` e `vet2` e armazenar em um terceiro objeto `vet3`.

```
> vet1 <- c( 10, 20, 30, 40)
> vet2 <- c( 60, 70, 80, 90, 100, 110)
> (vet3 <- c(vet1, vet2))
 [1]  10  20  30  40  60  70  80  90 100 110
```

- **Tarefa**: tente `c(vet2, vet1)`.

Criando vetores longos

- A entrada de dados diretamente no R não é recomendada, porém em alguns momentos é necessário criar alguns vetores grandes. Abaixo listamos algumas funções:

Funções rep() e seq()

- `rep()`: replica os valores passados para a função.
- `seq()`: cria uma sequência, sendo possível controlar a que passo a sequência cresce.
- `:` atalho para função `seq()`, quando queremos criar uma sequência que é incrementada por uma unidade.

```
> rep(x = c(1, 2, 3), each = 3)
[1] 1 1 1 2 2 2 3 3 3
```

```
> rep(x = c(1, 2, 3), times = 3)
[1] 1 2 3 1 2 3 1 2 3
```

- `each = XX`, indica que cada elemento será repetido XX vezes.
- `times = XX`, indica que o vetor será repetido XX vezes.

Criando vetores longos

- Criando um vetor usando a função `seq()`.

```
> seq(from = 10, to = 110, by = 10) #lembra do vet3 <- c(vet1, vet2) ?
 [1]  10  20  30  40  50  60  70  80  90 100 110
```

```
> seq(from = 0, to = 1, length.out = 10)
 [1] 0.0000000 0.1111111 0.2222222 0.3333333 0.4444444 0.5555556 0.6666667 0.7777778
 [9] 0.8888889 1.0000000
```

- A função `seq()`, cria um vetor que inicia `from = INICIO` e termina `to = FIM`.
- `by = XX`, indica que a sequência será construída de XX em XX.
- `length.out = XX`, indica que o vetor terá exatamente o comprimento igual a XX.

Agora é a sua vez

Agora é a sua vez

Operações com vetores

- Experimente digitar no R:
  - `x <- 1:5; y <- c(2:4, 1, 2)`
  - `x == seq(1, 5, by = 1)`
  - `x < y`

Agora é a sua vez

Operações com vetores

- Experimente digitar no R:
  - `x <- 1:5; y <- c(2:4, 1, 2)`
  - `x == seq(1, 5, by = 1)`
  - `x < y`

O que aconteceu?

Operações com vetores

- O R também é conhecido por ser uma linguagem vetorizada. Por exemplo, no exercício anterior, quando comparamos `x < y`, o teste foi realizado para todo o vetor. Com isso, cada elemento do vetor `x` foi comparado com o seu respectivo par do vetor `y`.
- Outras operações também podem ser executadas, por exemplo:

```
> a <- 1:5; b <- 3:7
> a + b #somando elemento a elemento
> a - b #subtraindo elemento a elemento
> a * b #multiplicando elemento a elemento
> a / b #dividindo elemento a elemento
> a ^ b #exponeciando elemento a elemento
```

- Note que o resultado é sempre um vetor de mesmo comprimento que os vetores `a` e `b`.
- E se os vetores tivessem comprimentos diferentes?

Operações com vetores

- **Atenção**: quando realizamos as mesmas operações anteriores, porém com vetores de comprimentos diferentes.

```
> u <- c(10, 20, 30)
> v <- 1:9
> u + v
[1] 11 22 33 14 25 36 17 28 39
```

```
> w <- 1:10
> u + w
Warning in u + w: longer object length is not a multiple of shorter object length
 [1] 11 22 33 14 25 36 17 28 39 20
```

Acessando elementos do vetor

- O acesso aos elementos de um vetor é realizado através do operador colchetes: `[ ]`.

vetor[INDICE]

- Por exemplo, para acessar o índice (posição) 4 do vetor `x`, basta fazer `x[4]`

```
> x <- 10:1
> x[4]
[1] 7
```

- Índice negativo: quando o sinal negativo é usado na frente do índice, o resultado é um vetor com o membro referente a este índice removido do vetor:

vetor[-INDICE]

```
> x[-4]
[1] 10  9  8  6  5  4  3  2  1
```

Acessando elementos do vetor

Nota: o vetor resultado da consulta pode ser armazenado e utilizado em outras análises, ou seja, a saída (*output*) do R também pode ser utilizada como dados de entrada!

```
> novo_x <- x[-5]
> length(x)
[1] 10
```

```
> length(novo_x)
[1] 9
```

Acessando elementos do vetor

- Uma forma ainda mais interessante de acessar elementos de um vetor é utilizando os operadores lógicos.
- Esta forma também é chamada de filtro. Lembra que ao realizar uma operação lógica sobre um vetor, nós temos como output um outro vetor de `TRUE` e `FALSE`.
- Suponha que temos um vetor de idades e queremos selecionar apenas as idades acima de 35 anos.

```
> idade <- c(34, 27, 20, 28, 32, 43, 31, 18, 45, 36)
> idade > 35
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

- Alguma sugestão?

Acessando elementos do vetor

- Uma forma ainda mais interessante de acessar elementos de um vetor é utilizando os operadores lógicos.
- Esta forma também é chamada de filtro. Lembra que ao realizar uma operação lógica sobre um vetor, nós temos como output um outro vetor de `TRUE` e `FALSE`.
- Suponha que temos um vetor de idades e queremos selecionar apenas as idades acima de 35 anos.

```
> idade <- c(34, 27, 20, 28, 32, 43, 31, 18, 45, 36)
> idade > 35
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE  TRUE
```

- Alguma sugestão?

- Basta fazermos `vetor[operacao_logica]`.

```
> idade[idade > 35]
[1] 43 45 36
```

Agora é a sua vez

Agora é a sua vez

Acessando vetores.

- Temos o vetor `altura` com a altura de 15 mulheres. Queremos criar mais três vetores:
  - Um com as mulheres com altura menor ou igual a 160.
  - Um com as mulheres com altura maior que 160 e menor ou igual a 170.
  - Um com as mulheres com altura maior que 170.
- Remover a altura 180 do vetor `altura`.

```
> altura <- c(150, 152, 145, 157, 167, 172, 175, 170, 165, 177, 162, 180, 160, 155, 147)
```

Agora é a sua vez

Acessando vetores.

- Temos o vetor `altura` com a altura de 15 mulheres. Queremos criar mais três vetores:
  - Um com as mulheres com altura menor ou igual a 160.
  - Um com as mulheres com altura maior que 160 e menor ou igual a 170.
  - Um com as mulheres com altura maior que 170.
- Remover a altura 180 do vetor `altura`.

```
> altura <- c(150, 152, 145, 157, 167, 172, 175, 170, 165, 177, 162, 180, 160, 155, 147)
```

```
> altura <- c(150, 152, 145, 157, 167, 172, 175, 170, 165, 177, 162, 180, 160, 155, 147)
> menor160     <- altura[altura <= 160]
> entre160e170 <- altura[altura > 160 & altura <= 170]
> maior170 <- altura[altura > 170]
>
> #removendo 180
> altura[altura != 180]
 [1] 150 152 145 157 167 172 175 170 165 177 162 160 155 147
```

Vetores: funções

Algumas funções que são aplicadas sobre vetores

- `mean(x)`: média
- `sd(x)`: desvio padrão
- `min(x)`: mínimo
- `max(x)`: máximo
- `range(x)`: vetor com mínimo e máximo
- `sum(x)`: soma todos os elementos
- `exp(x)`: exponencia todos os elementos
- `sqrt(x)`: raiz quadrada
- `log(x)`: logarítmo natural

Matrizes

- **matriz**: é uma coleção de vetores lado a lado, em que cada vetor tem exatamente o mesmo comprimento e são da mesma classe. Cada linha ou coluna de uma matriz individualmente será um vetor.
- Exemplo de uma matriz com 2 linhas e 3 colunas:

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

Matrizes

- **matriz**: é uma coleção de vetores lado a lado, em que cada vetor tem exatamente o mesmo comprimento e são da mesma classe. Cada linha ou coluna de uma matriz individualmente será um vetor.
- Exemplo de uma matriz com 2 linhas e 3 colunas:

$$A = \begin{bmatrix} 2 & 4 & 3 \\ 1 & 5 & 7 \end{bmatrix}$$

- Representando esta matriz no R:

```
> A <- matrix(
+ c(2,4,3,1,5,7), # elementos de dados
+ nrow = 2,       # numero de linhas
+ ncol = 3,       # numero de colunas
+ byrow = TRUE)   # preencher matriz pelas linhas
> A
     [,1] [,2] [,3]
[1,]    2    4    3
[2,]    1    5    7
```

Acessando elementos de uma matriz

- O elemento na linha `m` da coluna `n` da matriz `A` pode ser acessado pela expressão `A[m,n]`:

```
> A[2, 3] # elemento na linha 2, coluna 3
[1] 7
```

- A linha `m` inteira de `A` pode ser extraída por `A[m, ]`

```
> A[2, ] # acessando a linha 2 completa
[1] 1 5 7
```

- A coluna `n` inteira de `A` pode ser extraída por `A[, n]`

```
> A[, 3] # acessando a coluna 3 completa
[1] 3 7
```

- Podemos também extrair mais de uma linha ou coluna por vez:

```
> A[, c(1, 3)] # as colunas 1 e 3
     [,1] [,2]
[1,]    2    3
[2,]    1    7
```

Matriz: atributos

- As matrizes também possuem seus atributos: nomes das linhas, nomes das colunas, número de linha e colunas.
- `rownames()`: atribui e acessa os nomes das linhas
- `colnames()`: atribui e acessa os nomes das colunas
- `nrow()`: retorna o número de linhas da matriz
- `ncol()`: retorna o número de colunas da matriz
- `dim()`: retorna o número de linhas e colunas da matriz

```
> dim(A)
[1] 2 3
```

```
> nrow(A)
[1] 2
```

```
> ncol(A)
[1] 3
```

```
> rownames(A) <- letters[1:nrow(A)]
> colnames(A) <- LETTERS[1:ncol(A)]
> A
  A B C
a 2 4 3
b 1 5 7
```

Combinando matrizes

- Assim como os vetores, as matrizes também pode ser concatenadas/combinadas usando as funções:
- `cbind()`: concatena matrizes lado a lado, ou por colunas
- `rbind()`: concatena matrizes empilhando-as, ou por linhas
- Ambas funções podem concatenar vetores e então resultar em uma matriz, como no exemplo abaixo.

```
> Segunda <- c(23, 29, 27, 28, 25)
> Terça   <- c(21, 24, 29, 31, 21)
> Quarta  <- c(24, 26, 25, 27, 31)
> Quinta  <- c(21, 27, 32, 21, 21)
> Sexta   <- c(22, 33, 27, 24, 33)
> Sábado  <- c(30, 28, 25, 24, 20)
> Domingo <- c(21, 21, 20, 30, 26)
>
> dias_uteis <- cbind(Segunda, Terça, Quarta, Quinta, Sexta)
> final_de_semana <- cbind(Sábado, Domingo)
```

Combinando matrizes

- Agora vamos concatenar as duas matrizes com a função `cbind()`:

```
> dias_uteis
     Segunda Terça Quarta Quinta Sexta
[1,]      23    21     24     21    22
[2,]      29    24     26     27    33
[3,]      27    29     25     32    27
[4,]      28    31     27     21    24
[5,]      25    21     31     21    33
```

```
> final_de_semana
     Sábado Domingo
[1,]     30      21
[2,]     28      21
[3,]     25      20
[4,]     24      30
[5,]     20      26
```

```
> (mes <- cbind(dias_uteis, final_de_semana))
     Segunda Terça Quarta Quinta Sexta Sábado Domingo
[1,]      23    21     24     21    22     30      21
[2,]      29    24     26     27    33     28      21
[3,]      27    29     25     32    27     25      20
[4,]      28    31     27     21    24     24      30
[5,]      25    21     31     21    33     20      26
```

Combinando matrizes

- Podemos concaternar (empilhar) usando a função `rbind()`:

```
> matriz1 <- matrix(1:8, nrow = 2, ncol = 4)
> matriz2 <- matrix(1:16, nrow = 4, ncol = 4)
> (matriz_rbind <- rbind(matriz1, matriz2))
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8
[3,]    1    5    9   13
[4,]    2    6   10   14
[5,]    3    7   11   15
[6,]    4    8   12   16
```

Acessando matrizes: `rownames()` e `colnames()`

- A matrizes também podem ser acessadas usando os nomes das linhas e colunas:

```
> rownames(mes) <- paste('Semana', 1:nrow(mes), sep = '')
> mes['Semana1', ] #o output desta consulta é um vetor
Segunda   Terça  Quarta  Quinta   Sexta  Sábado Domingo
     23      21      24      21      22      30      21
```

```
> mes[1:2, 'Quinta']
Semana1 Semana2
     21      27
```

```
> mes[c('Semana1', 'Semana3'), c('Terça', 'Quarta')]
        Terça Quarta
Semana1    21     24
Semana3    29     25
```

Agora é a sua vez

Agora é a sua vez

Acessando matrizes

- Considere a matriz `mes` e calcule:
  - A média da coluna `Quarta`.
  - A desvio-padrão da `Semana3`.
  - A média de cada uma dos dias da semana.
- Dica: `colMeans()`

```
> mean(mes[, 'Quarta'])
[1] 26.6
```

```
> sd(mes['Semana3', ])
[1] 3.735289
```

```
> colMeans(mes)
Segunda   Terça  Quarta  Quinta   Sexta  Sábado Domingo
   26.4    25.2    26.6    24.4    27.8    25.4    23.6
```

Matrizes: funções

Algumas funções que são aplicadas sobre matrizes

- Todas as funções que são aplicadas em vetores.
- `colMeans(x)`: média de cada coluna
- `rowMeans(x)`: média de cada linha
- `colSums(x)`: soma de cada coluna
- `rowSums(x)`: soma de cada linha

Listas: `list()`

- Lista é a estrutura de dados mais genérica do R, pois ela comporta vetores e matrizes de diferentes classes.

```
> num  <- c(2, 3, 5)
> car  <- letters[1:5]
> logi <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
> x <- list(num, car, logi, 10)
> x
[[1]]
[1] 2 3 5

[[2]]
[1] "a" "b" "c" "d" "e"

[[3]]
[1]  TRUE FALSE  TRUE FALSE FALSE

[[4]]
[1] 10
```

- Note que os vetores são de classes e tamanhos diferentes.

Acessando elementos de uma lista

- Para acessar uma parte da lista usa-se o operador colchetes simples [ ].

```
> x[2]
[[1]]
[1] "a" "b" "c" "d" "e"
```

```
> class(x[2])
[1] "list"
```

- Note que usar o colchetes simples, o objeto retornado é da classe `list()`.
- Para acessar os valores armazenados em uma posição da lista temos que usar os colchetes duplos [[ ]]

```
> x[[2]]
[1] "a" "b" "c" "d" "e"
```

```
> class(x[[2]])
[1] "character"
```

Acessando elementos de uma lista

- Podemos modificar seu conteúdo diretamente:

```
> x[[2]][1] <- "WW"
> x[[2]]
[1] "WW" "b"  "c"  "d"  "e"
```

```
> car
[1] "a" "b" "c" "d" "e"
```

- Note que o vetor `car` foi apenas utilizado para construir a lista `x`, com isso ao alterar `x`, não estamos alterando os valores de `car`.

Nomes de membros de listas

- Podemos atribuir nomes aos membros de uma lista.

```
> compras <- list(pc = c("notebook", "desktop"), ano = c(1998, 2005, 20
> compras
$pc
[1] "notebook" "desktop"

$ano
[1] 1998 2005 2008 2012
```

```
> names(compras) #nome de cada elemento da lista
[1] "pc"  "ano"
```

- Note o símbolo $ no output acima. Este mesmo símbolo pode ser utilizado para acessar uma posicão da lista.

```
> compras$pc
[1] "notebook" "desktop"
```

Outras formas de acessar uma lista

- Usando o nome para acessar:

```
> compras[["pc"]]
[1] "notebook" "desktop"
```

- Usando um vetor para acessar multiplas posições:

```
> x[c(1, 3)]
[[1]]
[1] 2 3 5

[[2]]
[1]  TRUE FALSE  TRUE FALSE FALSE
```

Outras formas de acessar uma lista

- Atribuindo nomes ao membros da lista:

```
> names(x)
NULL
```

```
> names(x) <- c('numeros', 'caractere', 'logico', 'escalar')
> x[c('numeros', 'caractere')]
$numeros
[1] 2 3 5

$caractere
[1] "WW" "b"  "c"  "d"  "e"
```

Data frames: `data.frame()`

- Um `data frame` é uma lista de vetores de `igual comprimento`.

```
> nomes <- c('Joao', 'Lara', 'Manoel', 'Pedro', 'Denise')
> idade <- c(30, 43, 21, 34, 25)
> sexo  <- c('M', 'F', 'M', 'M', 'M')
> (df <- data.frame(nomes, idade, sexo, stringsAsFactors = F))
   nomes idade sexo
1   Joao    30    M
2   Lara    43    F
3 Manoel    21    M
4  Pedro    34    M
5 Denise    25    M
```

Acessando data.frame

- Como `data.frame` é um caso especial de uma lista, logo todas as formas de acessar uma lista podem ser usadas com um `data.frame`.

```
> df$nomes      #acessando direto os elementos
[1] "Joao"   "Lara"   "Manoel" "Pedro"  "Denise"
```

```
> df[['nomes']] #acessando direto os elementos
[1] "Joao"   "Lara"   "Manoel" "Pedro"  "Denise"
```

```
> df['nomes']   #acessando uma coluna, tente class(df['nomes'])
   nomes
1   Joao
2   Lara
3 Manoel
4  Pedro
5 Denise
```

Acessando data.frame

- Podemos usar também a mesma forma de acessar matrizes.

```
> df[, 1]         #acessando a coluna 1
[1] "Joao"   "Lara"   "Manoel" "Pedro"  "Denise"
```

```
> df[, 'nomes']  #acessando a coluna nome
[1] "Joao"   "Lara"   "Manoel" "Pedro"  "Denise"
```

```
> df[2, ]         #acessando a linha 2
  nomes idade sexo
2  Lara    43    F
```

```
> df[2:4, 2:3]   #linhas 2,3,4 e colunas 2 e 3
  idade sexo
2    43    F
3    21    M
4    34    M
```

data.frame: funções

Algumas funções que são aplicadas sobre data.frame

- `head(x)`: aprensenta as primeiras 6 linhas
- `str(x)`: estrutura do `data.frame`
- `summary(x)`: resumo do `data.frame`

data.frame: funções

```
> head(df)
   nomes idade sexo
1   Joao    30    M
2   Lara    43    F
3 Manoel    21    M
4  Pedro    34    M
5 Denise    25    M
```

```
> str(df)
'data.frame':   5 obs. of  3 variables:
 $ nomes: chr  "Joao" "Lara" "Manoel" "Pedro" ...
 $ idade: num  30 43 21 34 25
 $ sexo : chr  "M" "F" "M" "M" ...
```

```
> summary(df)
    nomes               idade          sexo
 Length:5           Min.   :21.0   Length:5
 Class :character   1st Qu.:25.0   Class :character
 Mode  :character   Median :30.0   Mode  :character
                    Mean   :30.6
                    3rd Qu.:34.0
                    Max.   :43.0
```

Agora é a sua vez

Agora é a sua vez

Trabalhando com `data.frame`

- Carregue o `data.frame` `mtcars` usando a função `data(mtcars)`.
- `?mtcars` conheça a base de dados.
- Utilize as funções `str()`, `summary()` e `head()`.
- Cacule a média para cada uma das colunas.

Módulo III: Loops e condições

Loops: `for()`

- Loops são replicações de uma mesma tarefa para diferentes valores.
- O `R` possui três opções de loops: `for`, `while`, `repeat`
- **Sintaxe:**

Estrutura do `for`

```
for(var in seq) {

tarefa que depende de var

...

}
```

- A ideia é que o valor do objeto `var` vai variar de acordo com vetor `seq`. Por exemplo:

```
> for(i in 1:3) {
+   print(i)
+ }
[1] 1
[1] 2
[1] 3
```

Loops: `for()`

- Exemplo calculando o desvio-padrão de cada variável do `mtcars`.

```
> desvio <- numeric(ncol(mtcars)) #definindo um vetor numerico
> for(cc in 1:ncol(mtcars)) {
+   desvio[cc] <- sd(mtcars[, cc])
+ }
```

- Ou, podemos passar um vetor de caracteres.

```
> desvio2 <- numeric(ncol(mtcars)) #definindo um vetor numerico
> names(desvio2) <- names(mtcars)
>
> for(cc in names(mtcars)) {
+   desvio2[cc] <- sd(mtcars[, cc])
+ }
> desvio == desvio2
 mpg  cyl disp   hp drat   wt qsec   vs   am gear carb
TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Loops: while

- **Sintaxe:**

Estrutura do while

```
while(VERDADEIRO) {

executa a tarefa

...

}
```

- **Exemplo**:

```
> resposta <- ""
> while(resposta != "R") {
+   resposta <- readline(prompt = "Qual a é a linguagem franca da ciência dos daods?")
+ }
```

Loops: `repeat`

- **Sintaxe:**

Estrutura do `repeat`

```
repeat {

executa a tarefa

...

}
```

- **Exemplo**:

```
> x <- 1
> repeat {
+    print(x)
+    x = x+1
+    if (x == 6){
+        break
+    }
+ }
```

- Note o comando `if()`

Condições

- No R existe duas principais formas de usar condições: `if(){}else{}` e `ifelse()`.
- **Sintaxe:**

Estrutura do `if()`

```
if(condicao) {

executa ESTA tarefa se a condicao for verdadeira

} else {

caso contrário execute este OUTRA tarefa

}
```

- **Exemplo**:

```
> for(i in 1:10) {
+   if(i > 7) {
+     print(i)
+   }
+ }
[1] 8
[1] 9
[1] 10
```

Condições

- A função `ifelse(teste, sim, nao)` avalia o teste para cada entrada do vetor e executa uma tarefa se for verdadeira ou outra caso contrário.
- Versão vetorizada do tradicional `if(){}else{}`.
- **Exemplo**:

```
> nomes <- c('Joao', 'Lara', 'Manoel', 'Pedro', 'Denise')
> idade <- c(30, 43, 21, 34, 25)
> sexo  <- c('M', 'F', 'M', 'M', 'M')
> df <- data.frame(nomes, idade, sexo, stringsAsFactors = F)
> (df$sexo <- ifelse(df$sexo == 'M', 'Masculino', 'Feminino'))
[1] "Masculino" "Feminino"  "Masculino" "Masculino" "Masculino"
```

Módulo IV: Lendo dados de arquivos no formato texto

Lendo arquivo texto

- Arquivo de texto: arquivo de texto plano, sem qualquer formatação especial, e pode ser visualizado em qualquer editor de texto simples.
- Para leitura de arquivo texto iremos usar a função `read.table()`.

```
> args(read.table)
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
    numerals = c("allow.loss", "warn.loss", "no.loss"), row.names,
    col.names, as.is = !stringsAsFactors, na.strings = "NA",
    colClasses = NA, nrows = -1, skip = 0, check.names = TRUE,
    fill = !blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
    comment.char = "#", allowEscapes = FALSE, flush = FALSE,
    stringsAsFactors = default.stringsAsFactors(), fileEncoding = "",
    encoding = "unknown", text, skipNul = FALSE)
NULL
```

```
> ?read.table()
```

Lendo arquivo texto

- Ler um banco de dados com colunas separadas por vírgulas e ponto como separador decimal.

```
> dados <- read.table(file = "../data/reg3.csv",
+                     sep = ",", #separador
+                     dec = ".", #decimal
+                     header = TRUE) #cabeçalho com o nome das variaveis
```

- `file`: caminho com o nome do arquivo a ser importado.
- `sep`: caractere separador das variáveis (colunas) (Ex.: vírgula)
- `dec`: caractere para casas decimais (Ex.: ponto)
- `header`: se o arquivo contem o nome das variáveis (TRUE)

Agora é a sua vez

Agora é a sua vez

Importando dados de arquivo texto

- Importar o banco de dados `"data/Dados_VSB/Dados_Fic_Enf.csv"`, onde os campos estão separados por `;` e `.` como separador decimal.
- Quantas colunas e linhas tem este `data.frame`?

Diferentes caracteres representando missing

- Suponha que temos um banco de dados, onde os missing são representados por 99 e 9999 e queremos que o R entenda estes são NA. "Missing" é tratado como uma constante própria no R, esta constante é NA.
- Ler o banco de dados sem avisar o R quem so os missings.

```
> idh99 <- read.table("../data/idh.csv", sep = ";", dec = ".",header =
```

- Ler o banco de dados utilizando o parâmetro na.strings.

```
> idh99 <- read.table("../data/idh.csv", sep = ";", dec = ".",
+                     header = TRUE, na.strings = c(99, 9999))
```

Exportando arquivos de texto

- Para salvar um arquivo texto no R tem a mesma lógica da leitura.
- Usaremos a função `write.table()`.

```
> write.table(idh99, "idh99.csv",
+             sep = ";", dec = ".",
+             row.names = FALSE,
+             quote=FALSE)
```

- `row.names`: não inserir o nome das linhas no arquivo de saída.
- `quote`: não colocar aspas nas variáveis do tipo character.

Casos particulares da função `read.table()`

- A função `read.csv()` são para bases de dados em que os campos são separados por vírgula e as casas decimais separadas por ponto.

```
> reg3 <- read.csv("../data/reg3.csv")
```

- A função `read.csv2()` são para bancos de dados em que os campos são separados por ponto e vírgula e as casas decimais separadas por vírgula.
- Para exportar os arquivos também temos casos particulares para a função `write.table()`. Sendo elas, `write.csv()` e `write.csv2()`.

Importando planilha do Excel

- Para importar planilhas do Excel para o R devemos utilizar o pacote readxl.

```
> #install.package(readxl)
> library(readxl)
> dados_excel <- read_excel(path = 'data/datasets.xlsx', sheet = 1)
> ?read_excel
```

- path: caminho para a planilha de dados
- sheet: nome ou número da aba na planilha

Módulo V: Funções

Definições básicas

Para que serve uma função?

- Existem procedimentos que são repetidos diversas vezes em um *script*
  - Encontrar o maior/menor valor em uma lista
  - Encontrar a média de cada coluna de uma matriz
  - Fazer o resumo estatístico de uma variável
  - ...

- Nestes casos é possível automatizar o processo através de uma função

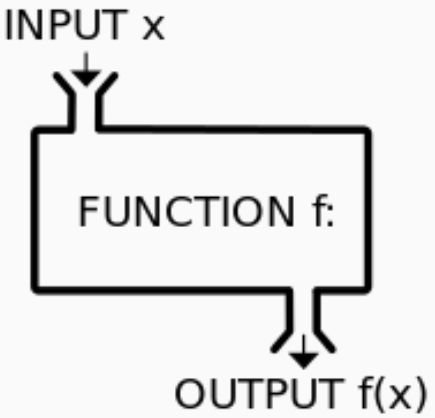
- Com isso o código fica mais enxuto, simples e legível

- Outro ponto positivo é que dada uma alteração na função não é necessário alterar o código em diversos pontos

Estrutura de uma função

Estrutura de uma função

```
NomeDaFuncao <- function(arg1, arg2, ...){
procedimento1
...
procedimenton
return(resultado)
}
```



Componentes de uma função

Uma função é composta basicamente de três partes:

Partes de uma função

- `body`: Representa os procedimentos que a função executa
- `formals`: Representa os argumentos de input da função
- `environment`: Representa o ambiente em que a função está definida

Considere a seguinte função de exemplo:

```
> produto <- function(x, n){
+   resultado <- n*x
+   return(resultado)
+   }
```

Componentes de uma função

```
> formals(produto)
$x


$n
```

```
> body(produto)
{
    resultado <- n * x
    return(resultado)
}
```

```
> environment(produto)
<environment: R_GlobalEnv>
```

Ambiente global, pacotes e funções primitivas

Funções do ambiente global (criadas pela usuário)

```
> environment(produto)              # Ambiente global
<environment: R_GlobalEnv>
```

Funções relacionadas à algum pacote.

```
> environment(ggplot2::ggplot)      # Pacote ggplot2
<environment: namespace:ggplot2>
```

Funções primitivas não possuem um ambiente dentro do R.

```
> environment(sum)                  # Função primitiva
NULL
```

Exemplo

```
> divisores <- function(x = 10){
+
+   resposta <- rep(0, x-1)
+   for(i in 2:(x-1)){
+     resposta[i-1] <- ifelse(x%%i == 0, 1, 0)}
+
+   divisores <- which(resposta == 1)+1
+
+   if(resposta[1] == 1){
+     par_impar <- "Par"
+   }else{
+     par_impar <- "Ímpar"
+   }
+
+   return(list(div = divisores, par_impar = par_impar))
+ }
```

Exemplo

```
> divisores(x = 4)
$div
[1] 2

$par_impar
[1] "Par"
```

```
> divisores(x = 10)
$div
[1] 2 5

$par_impar
[1] "Par"
```

```
> environment(produto)
<environment: R_GlobalEnv>
```

Funções do R base

Lista de todas as funções do R (base):

▸ Funções do R-base

- A base do R conta com muitas funções matemáticas
- Além disso existem funções para:
  - Manipulação de dados numéricos
  - Manipulação de dados textuais
  - Manipulação de datas
  - ...
- Grande parte das funções são vetorizadas
- Exemplos:
  - `log()`, `sqrt()`, `cos()`, `sin()`, `factorial()`, ...

Variáveis locais e globais

Variáveis criadas dentro de uma função não são acessadas no ambiente global

```
> f1 <- function(a){
+   b <- 10
+   resultado <- b-a
+   return(resultado)
+ }
>
> f1(a = 8)
[1] 2
```

```
> b
Error in eval(expr, envir, enclos): object 'b' not found
```

Ou seja, as variáveis estão definidas apenas dentro do escopo da função

Variáveis locais e globais

Qual o resultado da seguinte função:

```
> b <- 10
>
> f1 <- function(a){
+   resultado <- b-a
+   return(b-a)
+ }
>
> f1(a = 8)
```

Variáveis locais e globais

Qual o resultado da seguinte função:

```
> b <- 10
>
> f1 <- function(a){
+   resultado <- b-a
+   return(b-a)
+ }
>
> f1(a = 8)
```

```
[1] 2
```

Variáveis locais e globais

Qual o resultado da seguinte função:

```
> b <- 10
>
> f1 <- function(a){
+   resultado <- b-a
+   return(b-a)
+ }
>
> f1(a = 8)
```

```
[1] 2
```

Uma variável não definida mas utilizada dentro de uma função é procurada no ambiente global!!!

Variáveis locais e globais

Para atualizar um objeto do escopo global de dentro da uma função utiliza-se o operador '<<-'

```
> b <- 10
> val <- 10
>
> f1 <- function(a){
+   val <<- 20
+   return(b-a)
+ }
>
> f1(a = 10)
[1] 0
```

```
> val
[1] 20
```

Agora é a sua vez

Criando uma função

Crie uma função que recebe um vetor numérico de qualquer tamanho e retorna uma lista contendo:

- O tamanho do vetor (função `length()`)
- A soma do vetor (função `sum()`)
- Se o tamanho do vetor for par:
  - Retornar o primeiro elemento
- Se o tamanho do vetor for ímpar:
  - Retornar o último elemento

Solução

```
> RetornaLista <- function(x){
+   tam <- length(x)
+   soma <- sum(x)
+   par <- (tam%%2 == 0)
+
+   if(par){
+     elemento <- x[1]
+   } else{
+     elemento <- x[tam]
+   }
+
+   resultado <- list(tamanho = tam,
+                     soma = soma,
+                     elemento = elemento)
+   return(resultado)
+ }
```

Solução

```
> RetornaLista(x = c(1, 3, 5, 7, 9))
$tamanho
[1] 5

$soma
[1] 25

$elemento
[1] 9
```

```
> RetornaLista(x = c(11, 3, 54, 7, 99, 22))
$tamanho
[1] 6

$soma
[1] 196

$elemento
[1] 11
```

Operadores

Operadores

Operadores também são funções, porém podem ser utilizadas de duas formas diferentes:

```
> 2 + 2*(8/4)
[1] 6
```

```
> '+'(2, '*'(2, '/'(8, 4)))
[1] 6
```

```
> 3 > 4
[1] FALSE
```

```
> '>'(3, 4)
[1] FALSE
```

Operadores

O mesmo vale para outras funções básicas

```
> i <- 1
> x <- letters[1:3]
>
> 'if'(x[i] == "a", print("Sim!"), print("Não"))
[1] "Sim!"
```

```
> 'for'(i, x, print(i))
[1] "a"
[1] "b"
[1] "c"
```

```
> '['(x, 3)
[1] "c"
```

Criando operadores

Para criar um operador basta criar uma função entre percentuais '%funcao%'

```
> '%soma_mult%' <- function(x, y){
+   z <- x + y
+   result <- x*z
+   return(result)
+ }
>
> 4 %soma_mult% 6
[1] 40
```

```
> 10 %soma_mult% 2
[1] 120
```

Tipos de *inputs*

As funções em R podem receber qualquer tipo de estrutura (argumentos, caracteres, listas e até mesmo outras funções)

```
> ApplyFunc2List <- function(lista, funcao){
+   dimensao <- length(lista)
+   resultado <- rep(0, dimensao)
+
+   for(i in 1:dimensao){
+     resultado[i] <- funcao(lista[[i]])
+   }
+
+   return(resultado)
+ }
>
> lista <- list(a = c(1, 2, 3), b = c(4, 5, 6))
>
> ApplyFunc2List(lista = lista, funcao = function(x) x[1])
[1] 1 4
```

Argumentos *default*

Quando criamos uma função com diversos parâmetros é necessário definir o valor de cada um deles...

```
> raiz <- function(pNum, pRaiz)
+ {
+   pNum^(1/pRaiz)
+ }
>
> raiz(pNum = 4)
Error in raiz(pNum = 4): argument "pRaiz" is missing, with no default
```

Argumentos *default*

Porém, nem sempre necessitamos alterar todos os parâmetros de uma função. Nestes casos, podemos criar parâmetros padrões que só são alterados quando o usuário solicita.

```
> raiz <- function(pNum, pRaiz = 2)
+ {
+   pNum^(1/pRaiz)
+ }
>
> raiz(pNum = 4)
[1] 2
```

Argumentos *default*

```
> raiz(pNum = 4)
[1] 2
```

```
> raiz(pNum = 4, pRaiz = 2)
[1] 2
```

```
> raiz(pNum = 4, pRaiz = 15)
[1] 1.096825
```

```
> raiz(pNum = 4, pRaiz = 100)
[1] 1.013959
```

O argumento '...'

Muitas vezes utilizamos funções secundárias para criação de uma função. Quando o interesse não são os parâmetros da função secundária podemos utilizar o argumento '...'

```
> SalvaBase <- function(file, ...){
+   data(cars)
+   write.table(x = cars, file = file, ...)
+   return("A base foi salva com sucesso")
+ }
>
> SalvaBase(file = "../data/cars.csv",
+           sep = ";", dec = ".")
[1] "A base foi salva com sucesso"
```

Desta forma o código fica mais enxuto e mais flexível para o usuário.

Tipos de *outputs*

Tipos de *outputs*

Os *outputs* podem ser os mais variados possíveis, podendo retornar um único resultado ou diversos (através de uma lista). Além disso, os resultados podem ser de diferentes tipos (listas, matrizes, funções, ...).

```
> RetornaObjetos <- function(ncol, nrow){
+   matriz <- matrix(0, ncol = ncol, nrow = nrow)
+   lista <- list(ncol = ncol, nrow = nrow)
+   funcao <- function(ncol, nrow){ncol*nrow}
+
+   return(list(matriz = matriz,
+               lista = lista,
+               funcao = funcao))
+ }
```

Tipos de *outputs*

```
> RetornaObjetos(ncol = 4, nrow = 1)
$matriz
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0

$lista
$lista$ncol
[1] 4

$lista$nrow
[1] 1


$funcao
function (ncol, nrow)
{
    ncol * nrow
}
<environment: 0x444fe80>
```

Mensagens de erro

A função *warning()*

A função `warning()` serve para auxiliar o programadr nos casos em que ocorrem pequenos imprevistos. Esses imprevistos não param o programa, porém é importante que o usuário tenha atenção e saiba que os resultados subsequentes podem conter erros.

```
> log(-1)
Warning in log(-1): NaNs produced
[1] NaN
```

Uso:

```
> calcula_razao <- function(x, y){
+   if(y == 0){
+     warning("y = 0 -> Divisão por zero!")
+   }
+
+   return(x/y)
+ }
```

A função *warning()*

```
> calcula_razao(x = 10, y = 10)
[1] 1
```

```
> calcula_razao(x = 10, y = 5)
[1] 2
```

```
> calcula_razao(x = 10, y = 10e-10)
[1] 1e+10
```

```
> calcula_razao(x = 10, y = 0)
Warning in calcula_razao(x = 10, y = 0): y = 0 -> Divisão por zero!
[1] Inf
```

A função *stop()*

A função `stop` pode ser utilizada para tornar suas funções mais robustas e trazer mais informações aos usuários. O objetivo desta função é parar o processo dado algum evento e retornar uma mensagem de informação.

Exemplo:

```
> nchar_function <- function(char){
+   if(!is.character(char)){
+     stop("O input deve ser um caracter")
+   }
+   return(nchar(char))
+ }
>
> nchar_function(char = "VSB")
[1] 3
```

```
> nchar_function(char = 10)
Error in nchar_function(char = 10): O input deve ser um caracter
```

Criando um log de erros

Em alguns problemas práticos erros são esperados, pois, nem sempre é possível prever todos as possibilidades de erros. Neste caso salvar um arquivo contendo os erros e *warnings* obtidos é muito útil e interessante. Podemos configurar o R para salvar um log de erros:

```
> funcao_erro <- function() {
+    cat(geterrmessage(), file = "../data/error.txt",
+        append = T)
+ }
>
> options("error" = funcao_erro)
>
> 1 + "2"
Error in 1 + "2": non-numeric argument to binary operator
```

Agora é a sua vez

Criando uma função um pouco mais complexa

Criando uma função complexa

Crie uma função com um argumento padrão e que recebe o argumento "...", essa função deve:

- Receber três vetores e montar uma matriz (cbind ou rbind)
- Caso a dimensão dos vetores seja diferente o programa deve parar
- Retornar a matriz construída
- Retornar um vetor com a soma de cada coluna

Solução

```
> RetornaSoma <- function(vet1 = rep(1, 5), vet2 = vet1,
+                         vet3 = vet1,
+                         ...){
+   if(!(length(vet1) == length(vet2) &
+        length(vet2) == length(vet3))){
+     stop("\n Os vetores não tem a mesma dimensão")
+   }
+   matriz <- cbind(vet1, vet2, vet3)
+   nCols <- ncol(matriz)
+   somaCols <- rep(0, nCols)
+
+   for(i in 1:nCols){
+     somaCols[i] <- sum(matriz[,i])
+   }
+
+   return(list(matriz = matriz,
+               soma = somaCols))
+ }
```

Solução

```
> RetornaSoma(vet1 = c(1,2,3), vet2 = c(4, 5, 6),
+             vet3 = c(7, 8, 9), sep = ";", dec = ".")
$matriz
     vet1 vet2 vet3
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

$soma
[1]  6 15 24
```

```
> RetornaSoma(vet1 = c(1,2,3), vet2 = c(4, 5, 6),
+             vet3 = c(7, 8), sep = ";", dec = ".")
Error in RetornaSoma(vet1 = c(1, 2, 3), vet2 = c(4, 5, 6), vet3 = c(7, :
 Os vetores não tem a mesma dimensão
```

Documentação de funções

Documentação de funções

Para entender uma função, seus argumentos e seus *outputs* pode-se consultar a documentação das funções.

Normalmente estas documentações possuem exemplos que auxiliam no entendimento da função.

```
> help(merge)
>
> ?merge
>
> ??merge
```

Veja por exemplo a função '`??expand.grid`'