

# **eRando em Ciência de Dado\$**

Luís Gustavo Schuck

2024-10-11

# Índice

<b>Bem Vindo</b>	<b>8</b>
Versões . . . . .	8
<b>Sobre o Autor</b>	<b>9</b>
<b>I Linguagem R</b>	<b>10</b>
R Foundation . . . . .	11
Comunidade . . . . .	12
<b>1 Introdução</b>	<b>13</b>
1.1 Console do R . . . . .	13
1.1.1 Executando Comandos . . . . .	14
1.1.2 Erros . . . . .	14
1.2 Objetos . . . . .	14
1.2.1 Vetores (Atômicos) . . . . .	18
1.3 Criação de Objetos . . . . .	18
1.4 Operações com Objetos . . . . .	19
1.4.1 Coerção . . . . .	21
1.5 Usando Funções . . . . .	22
1.5.1 Argumentos de Funções . . . . .	22
1.5.2 Armazenando Retorno . . . . .	23
1.6 Exibindo Objetos . . . . .	24
1.7 Remoção de Objetos . . . . .	24
1.8 Iniciando o R...Studio . . . . .	25
1.9 Trabalhando com Scripts . . . . .	26
1.9.1 Comentários . . . . .	29
<b>2 Nomeando Objetos</b>	<b>30</b>
2.1 Regras . . . . .	30
2.1.1 Primeiro Caractere . . . . .	31
2.1.2 Case Sensitive . . . . .	32
2.2 Resumo . . . . .	33
2.3 Convenções . . . . .	34

<b>3</b>	<b>Vetores</b>	<b>36</b>
3.1	Introdução . . . . .	36
3.2	Selecionando Elementos . . . . .	37
3.3	Nomeando Componentes . . . . .	37
3.4	Testando . . . . .	38
<b>4</b>	<b>Tipos de Dados</b>	<b>39</b>
4.1	Lógico ( <i>Logical</i> ) . . . . .	39
4.1.1	Valores Faltantes ( <b>NA</b> ) . . . . .	40
4.2	Inteiros ( <i>Integer</i> ) . . . . .	42
4.3	Ponto Flutuante ( <i>Double</i> ) . . . . .	43
4.3.1	<i>Not a Number</i> ( <b>NaN</b> ) . . . . .	43
4.3.2	<b>Inf</b> e <b>-Inf</b> . . . . .	44
4.4	Fatores ( <i>Factor</i> ) . . . . .	45
<b>5</b>	<b>Ambientes</b>	<b>46</b>
5.1	Global Env . . . . .	46
5.2	Ambiente de Pacotes . . . . .	47
5.3	Ambientes “Pai” . . . . .	47
5.4	Criando Ambientes . . . . .	48
<b>6</b>	<b>Operações Lógicas</b>	<b>50</b>
6.1	Operadores Relacionais . . . . .	50
6.2	Operadores Lógicos . . . . .	51
6.2.1	Ou Exclusivo (Xor) . . . . .	51
6.3	Precedência de Operadores Lógicos e Relacionais . . . . .	52
6.4	Funções <code>isTRUE</code> e <code>isFALSE</code> . . . . .	54
6.5	Funções <code>is.</code> . . . . .	54
6.6	<code>All</code> e <code>any</code> . . . . .	55
6.6.1	<code>AnyNA</code> . . . . .	58
6.7	Operador <code>%in%</code> . . . . .	59
<b>7</b>	<b>Listas</b>	<b>61</b>
7.1	Introdução . . . . .	61
7.2	Criando Listas . . . . .	61
7.3	Acessando Componentes . . . . .	61
7.4	Nomeando Componentes . . . . .	62
7.4.1	Nomes Abreviados . . . . .	63
<b>8</b>	<b>Data Frames</b>	<b>64</b>
8.1	O que são data frames ? . . . . .	64
8.1.1	Criando Data Frames . . . . .	65
8.1.2	Aplicar convenções de nomes . . . . .	65

8.2	Atributos . . . . .	66
8.3	Dimensões . . . . .	67
8.4	Acessando Dados . . . . .	67
8.4.1	Índices . . . . .	67
8.4.2	Usando Nomes das Colunas . . . . .	68
8.5	Filtrando Dados . . . . .	68
8.5.1	Classes de retorno . . . . .	70
8.6	Função Subset . . . . .	70
8.7	Junção de Dados . . . . .	72
<b>9</b>	<b>Operador Pipe</b>	<b>75</b>
9.1	Introdução . . . . .	75
9.2	Placeholder . . . . .	77
<b>10</b>	<b>Funções</b>	<b>79</b>
10.1	Criando Funções . . . . .	79
10.1.1	Argumentos - Valores Padrão . . . . .	79
10.2	Função x Ambiente . . . . .	79
10.2.1	Objetos no Ambiente da Função . . . . .	80
10.3	Retorno . . . . .	83
10.4	Recursividade . . . . .	83
10.4.1	Buscar Ambiente Pai (Recursivamente) . . . . .	84
10.5	Funções Genéricas . . . . .	84
10.6	Operadores Unários . . . . .	85
<b>11</b>	<b>Dados Externos</b>	<b>87</b>
11.1	Formato Csv . . . . .	87
11.1.1	Importar Arquivos csv . . . . .	87
11.1.2	Exportar Arquivos csv . . . . .	88
11.2	Formato RDS . . . . .	88
11.2.1	Importar Arquivos RDS . . . . .	88
11.2.2	Exportar Arquivos RDS . . . . .	88
<b>12</b>	<b>Controles de Fluxo</b>	<b>89</b>
12.1	Introdução . . . . .	89
12.2	If . . . . .	89
12.3	Ifelse . . . . .	90
12.4	If Else . . . . .	92
12.5	Laço For . . . . .	92
12.6	While . . . . .	94
12.7	Repeat . . . . .	95
12.8	Break e Next . . . . .	95

<b>13 Gráficos</b>	<b>97</b>
13.1 Introdução . . . . .	97
13.2 Função Plot . . . . .	97
13.3 Função Hist . . . . .	101
<b>14 Sumarização de Dados</b>	<b>105</b>
14.1 Funções Básicas . . . . .	105
14.1.1 Soma . . . . .	105
14.1.2 Média . . . . .	105
14.1.3 Mediana . . . . .	106
14.1.4 Máximo e mínimo . . . . .	106
14.2 Agregação . . . . .	107
14.3 Valores Faltantes - NA . . . . .	109
<b>15 Utilidades</b>	<b>111</b>
15.1 Listar Arquivos . . . . .	111
15.2 Listar diretórios . . . . .	112
15.3 Informações de arquivos . . . . .	112
15.4 Variáveis de Ambiente . . . . .	112
<b>II Pacotes</b>	<b>113</b>
O que são pacotes? . . . . .	114
<b>16 Introdução a Pacotes</b>	<b>115</b>
16.1 Pacotes Instalados . . . . .	115
16.2 Pasta de Instalação . . . . .	117
16.3 Pacotes Disponíveis . . . . .	118
16.4 Dependências de Pacotes . . . . .	118
16.5 Instalação de Pacotes . . . . .	119
<b>17 Pacote data.table</b>	<b>120</b>
17.1 Criando um Data Table . . . . .	120
17.1.1 Função data.table . . . . .	120
17.1.2 Função setDT . . . . .	120
17.2 Filtrando Dados . . . . .	121
17.3 Seleccionando Colunas . . . . .	121
17.4 Alterando Variáveis . . . . .	123
17.4.1 Forma Funcional . . . . .	123
17.4.2 Forma LHS := RHS . . . . .	123
17.5 Agrupando . . . . .	124
17.6 Ordenando Dados . . . . .	124
17.7 Encadeamento . . . . .	125

17.8	Layout	128
17.9	Console	128
17.10	Output	128
17.11	Environment	128
17.12	Source	132
<b>18</b>	<b>Menu Tools</b>	<b>133</b>
18.1	Install Packages	133
18.2	Check for Package Updates	134
18.3	Version Control	137
18.4	Terminal	137
18.5	Background Jobs	137
18.6	Global Options	137
18.6.1	Geral > <i>Basic</i>	138
18.6.2	Appearance	140
<b>III</b>	<b>Estatística</b>	<b>142</b>
	O que é Estatística	143
<b>19</b>	<b>Introdução</b>	<b>144</b>
19.1	Dados Qualitativos	144
19.2	Dados Quantitativos	144
19.3	População	144
19.4	Amostra	144
19.5	Parâmetros	144
19.6	Estatísticas	145
19.7	Tipos de Dados	145
19.7.1	Nominal	145
19.7.2	Ordinal	146
19.7.3	Intervalar	146
19.7.4	Razão	146
<b>20</b>	<b>Teorema de Bayes</b>	<b>147</b>
20.1	Fórmula	147
<b>IV</b>	<b>Ciência de Dados</b>	<b>149</b>
	Definições	150
<b>21</b>	<b>Trade-Off Viés x Variância</b>	<b>151</b>

<b>Bases de Dados</b>	<b>152</b>
Banco Central do Brasil . . . . .	152
Taxa de juros - Selic acumulada no mês . . . . .	152
BNDES . . . . .	152
Por porte de empresa - Aprovações . . . . .	152
<b>Convenções</b>	<b>153</b>
Marcações no Texto . . . . .	153
Nomes de Objetos . . . . .	153
Status do Material . . . . .	154
<b>Referências</b>	<b>155</b>

# Bem Vindo

Este é um livro sobre a utilização da linguagem R em Ciência de Dados.

Este material é um projeto pessoal usado como fonte de consulta e aprendizado, sem compromisso com uma estrutura específica.

Muitas vezes o exposto aqui é a prática (para fixação e exploração) de conceitos apresentados em outros materiais. Assim todas as fontes utilizadas, mesmo que de forma subjetiva, são citadas.

## Versões

A versão deste material é: `0.1.0.r format(Sys.Date(), '%d-%m-%Y')`.

A versão do **R** utilizada é: `r getRversion(), r R.Version()$nickname`.

A versão do **RStudio** é: 2023.06.1+524, Mountain Hydrangea.

A versão do **Quarto** é: `r quarto::quarto_version()`.

---

Última atualização: `r format(file.info('index.qmd')$mtime, '%d/%m/%Y - %H:%M:%S')`



## Sobre o Autor

Status □□□

[Luís Gustavo Schuck](#) é aluno do curso de pós-graduação em **Ciência de Dados e Big Data** da Puc-Minas. É formado em **Análise e Desenvolvimento de Sistemas** (2023) pela Universidade Feevale e em **Gestão Financeira** (2013) pelo Centro Universitário Internacional - Uninter. Possui **Especialização em Business Analytics** (2021) pela Universidade Federal do Rio Grande do Sul - UFRGS, **MBA em Administração e Finanças** (2017) pelo Centro Universitário Internacional - Uninter e **MBA em Gestão Bancária** (2015) pelo Centro Universitário Leonardo da Vinci - Uniasselvi. Possui certificação **ANBIMA CPA-10** (Certificação Profissional ANBIMA Série-10).

Atualmente atua como **Analista na Unidade de Estratégia e Inteligência de Crédito** do Banco do Estado do Rio Grande do Sul ([Banrisul](#)).

Utiliza R desde 2017.

---

Última atualização: 11/10/2024 - 21:48:39

**Parte I**  
**Linguagem R**

Status □□□

Conforme o **R Core Team** (R Core Team 2023c) 'R é uma linguagem e ambiente para computação estatística e gráficos'. Criada na década de 1990, R é uma **linguagem livre** e é distribuída sob a licença [GPLv2](#).

Mantida pela [R Foundation](#), atualmente (outubro/2024) é uma das linguagens mais usadas para Ciência de Dados e está entre as linguagens mais buscadas no Google como pode ser visto pelo [PYPL - Popularity of Programming Language](#).

Worldwide, May 2023 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	27.27 %	-0.5 %
2		Java	16.35 %	-1.6 %
3		JavaScript	9.52 %	+0.2 %
4		C#	6.92 %	-0.3 %
5		C/C++	6.55 %	-0.4 %
6		PHP	5.1 %	-0.5 %
7		R	4.34 %	-0.2 %
8		TypeScript	2.88 %	+0.3 %
9	↑	Swift	2.3 %	+0.1 %
10	↓	Objective-C	2.13 %	-0.1 %

Figura 1: 05/2023 - PYPL Popularity of Programming Language

No [IEEE Spectrum Top Programming Languages 2022](#) a linguagem R também aparece com bastante relevância.

## R Foundation

A R Foundation é uma organização sem fins lucrativos que tem como objetivo promover o desenvolvimento da linguagem R e ser ponto de referência para entidades que desejem

interagir com a comunidade de desenvolvimento do R.

A R Foundation possui uma grande quantidade de apoiadores e doadores. Dentre os principais Patronos do R está a empresa **Posit**, anteriormente **RStudio**, que desenvolve o principal IDE para R, também chamado de **RStudio**.

Você pode ser um [apoiador!](#)

## Comunidade

Uma grande vantagem da linguagem R é a existência de uma grande comunidade de desenvolvimento, assim como uma gama enorme de conteúdos distribuídos através da Internet, muitos de forma livre e de fácil acesso. Abaixo alguns sites com conteúdos muito ricos sobre R:

- [Análise de Dados Financeiros e Econômicos com o R - Versão Online](#)
- [Introdução à Linguagem R: seus fundamentos e sua prática](#)
- [R Manuals](#)
- [Big Book of R](#)
- [R for Data Science](#)
- [Datacamp](#)
- [Statistics Globe](#)
- [R Charts](#)
- [Statistical tools for high-throughput data analysis](#)

---

Última atualização: 11/10/2024 - 21:50:02

# 1 Introdução

Status □□□

Este capítulo tem como objetivo fornecer uma visão inicial mínima para que o usuário possa dar os primeiros passos na linguagem.

## 1.1 Console do R

A tela inicial do R em si é um console, onde são passados comandos e seu interpretador os executa e, se for o caso, exibe saídas. O cursor fica posicionado ao lado do símbolo do *prompt* do R, >. Este símbolo indica que o sistema está pronto para receber novo comando.

```
R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Figura 1.1: Tela Inicial

### 1.1.1 Executando Comandos

A tela inicial fornece algumas sugestões para consulta a dados sobre R, como licença da linguagem, citação, ajudas, etc. Usaremos como exemplo inicial o comando `license()`. Após a digitação do comando devemos confirmar com **ENTER** para que o R execute o comando informado e exiba na tela o resultado, no caso a licença da própria linguagem. Após a execução um novo sinal do *prompt* é exibido em aguardo de um possível próximo comando.

Podemos digitar `q()`, por exemplo, que é a função que efetua o encerramento do R.

Agora considere um cenário diferente, onde executamos o comando `license()` seguido do comando `citation()` (que mostra como deve ser feita a citação da Linguagem R). Conforme os comandos são processados, o console vai sendo preenchido com estes comandos e suas respectivas saídas. A medida que a tela vai ficando “cheia” os dados exibidos no topo vão “sumindo” para dar lugar aos mais recentes, na parte inferior.

#### Buscando Comandos Anteriores

Para buscar comandos executados anteriormente, pode-se usar a seta para cima do teclado. Os comandos vão sendo apresentados do mais recente ao mais antigo.

### 1.1.2 Erros

Sempre que ocorrer algum erro na execução de um comando será exibida no console uma mensagem com o termo **Error**. Muitas vezes a mensagem de erro auxilia na identificação da causa do erro reportado. Abaixo um exemplo com erro retornado pelo R após a tentativa de execução de uma função inexistente (erro na digitação do comando).

```
citatin()
```

```
Error in citatin(): não foi possível encontrar a função "citatin"
```

## 1.2 Objetos

R opera sobre entidades que são conhecidas como **objetos** (R Core Team 2023a, cap 3). Existem diversos tipos de objetos em R como listas, matrizes, bases de dados, séries temporais, gráficos, modelos, etc. Neste capítulo inicial serão utilizados os **vetores**, pois são as estruturas mais básicas.

```
R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"
Copyright (c) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> license()

This software is distributed under the terms of the GNU General
Public License, either Version 2, June 1991 or Version 3, June 2007.
The terms of version 2 of the license are in a file called COPYING
which you should have received with
this software and which can be displayed by RShowDoc("COPYING").
Version 3 of the license can be displayed by RShowDoc("GPL-3").

Copies of both versions 2 and 3 of the license can be found
at https://www.R-project.org/Licenses/.

A small number of files (the API header files listed in
R_DOC_DIR/COPYRIGHTS) are distributed under the
LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later.
This can be displayed by RShowDoc("LGPL-2.1"),
or obtained at the URI given.
Version 3 of the license can be displayed by RShowDoc("LGPL-3").

'Share and Enjoy.'

> |
```

Figura 1.2: Tela Inicial - Licença

```
R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> license()

This software is distributed under the terms of the GNU General
Public License, either Version 2, June 1991 or Version 3, June 2007.
The terms of version 2 of the license are in a file called COPYING
which you should have received with
this software and which can be displayed by RShowDoc("COPYING").
Version 3 of the license can be displayed by RShowDoc("GPL-3").

Copies of both versions 2 and 3 of the license can be found
at https://www.R-project.org/Licenses/.

A small number of files (the API header files listed in
R_DOC_DIR/COPYRIGHTS) are distributed under the
LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later.
This can be displayed by RShowDoc("LGPL-2.1"),
or obtained at the URI given.
Version 3 of the license can be displayed by RShowDoc("LGPL-3").

'Share and Enjoy.'

> q()
```

Figura 1.3: Tela Inicial - Quit (sair)



```

This software is distributed under the terms of the GNU General
Public License, either Version 2, June 1991 or Version 3, June 2007.
The terms of version 2 of the license are in a file called COPYING
which you should have received with
this software and which can be displayed by RShowDoc("COPYING").
Version 3 of the license can be displayed by RShowDoc("GPL-3").

Copies of both versions 2 and 3 of the license can be found
at https://www.R-project.org/Licenses/.

A small number of files (the API header files listed in
R_DOC_DIR/COPYRIGHTS) are distributed under the
LESSER GNU GENERAL PUBLIC LICENSE, version 2.1 or later.
This can be displayed by RShowDoc("LGPL-2.1"),
or obtained at the URI given.
Version 3 of the license can be displayed by RShowDoc("LGPL-3").

'Share and Enjoy.'

> citation()
To cite R in publications use:

  R Core Team (2023). _R: A Language and Environment for Statistical Computing_. R Foundation for
  Statistical Computing, Vienna, Austria. <https://www.R-project.org/>.

A BibTeX entry for LaTeX users is

@Manual{,
  title = {R: A Language and Environment for Statistical Computing},
  author = {{R Core Team}},
  organization = {R Foundation for Statistical Computing},
  address = {Vienna, Austria},
  year = {2023},
  url = {https://www.R-project.org/},
}

We have invested a lot of time and effort in creating R, please cite it when using it for data analysis.
See also 'citation("pkgname")' for citing R packages.
> |

```

Figura 1.4: Tela Inicial - Atualização do Console

## **i** Variáveis

Muitas vezes objetos em R são chamados de **variáveis**, no sentido de que variáveis armazenam dados. Isto ocorre principalmente para objetos que armazenam um único valor, como um único número ou texto.

### 1.2.1 Vetores (Atômicos)

Vetores são entidades que armazenam dados em posições (R Core Team 2023b, cap 2). Os vetores são ditos **atômicos**, pois seus dados são todos do mesmo tipo. Você pode pensar em um vetor como uma “local” onde serão armazenados dados. Os vetores podem ser de um dos 6 tipos abaixo:

Tabela 1.1: Tipos de Vetores

Tipo	Descrição	Exemplo
<i>logical</i>	valor lógico	TRUE, FALSE
<i>integer</i>	número inteiro	1
<i>double</i>	número com ponto flutuante (real)	1.5
<i>complex</i>	número complexo	1i
<i>character</i>	texto ( <i>strings</i> )	'R é software livre.'
<i>raw</i>	bytes	

## 1.3 Criação de Objetos

Para criação de objetos no R são usados os operadores de atribuição, `<-` e `=`. O operador mais usado é o `<-`. Assim para criação de um objeto do tipo vetor, pode ser usado o código abaixo, onde o objeto1 receberá o valor 10.

```
objeto1 <- 10
```

Para criação de variáveis do tipo texto, devem ser usadas aspas, simples ou duplas. Aqui o **objeto2** foi criado com uso de aspas ao início e ao final de **texto** para que o R trate o valor como *character*. Caso não sejam colocadas as aspas, o R entenderá o texto informado como o nome de um objeto.

```
objeto2 = 'texto'  
objeto3 <- texto
```

```
Error in eval(expr, envir, enclos): objeto 'texto' não encontrado
```

Você pode ver o conteúdo de um objeto informando seu nome no console seguido de **ENTER**.

```
objeto1
```

```
[1] 10
```

```
objeto2
```

```
[1] "texto"
```

## 1.4 Operações com Objetos

Objetos podem ser atualizados/alterados, novamente, com o operador `<-`. No exemplo abaixo vamos criar um vetor de nome **objeto3** com o operador `:`, que cria sequências de valores. Em seguida o **objeto3** será atualizado recebendo seu próprio conteúdo acrescido do valor 10.

```
objeto3 <- 1:5  
objeto3
```

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

```
objeto3 <- objeto3 + 10  
objeto3
```

```
[1] 11 12 13 14 15
```

O vetor `objeto3` foi criado com 5 posições, armazenando inicialmente os valores de 1 a 5. Após o vetor recebeu atualização dos valores para 11, 12, 13, 14 e 15. Note que a adição foi efetuada em todo o vetor.

Os valores dos componentes dos vetores podem ser acessados através de seu índice em combinação com o operador de extração `[`. Em R os índices iniciam em 1 (em muitas linguagens de programação o primeiro índice é 0).

Assim para receber como retorno o valor da posição 4 do `objeto3` usamos:

```
objeto3[4]
```

```
[1] 14
```

Podemos também operar sobre este valor, por exemplo, adicionando 10 ao valor da quarta posição do objeto3.

```
objeto3[4] + 10
```

```
[1] 24
```

Note que sem o operador de atribuição o valor da posição 4 do objeto3 não é atualizada dentro do objeto, apenas é exibida no console. Para atualizar seu valor devemos usar a atribuição.

```
objeto3
```

```
[1] 11 12 13 14 15
```

```
objeto3[4] <- objeto3[4] + 10  
objeto3
```

```
[1] 11 12 13 24 15
```

Neste caso apenas o valor da posição 4 foi alterado.

A busca por índices pode ser feita de forma mais elaborada. Por exemplo, para buscar três valores do vetor que estão armazenados de forma contígua podemos usar:

```
objeto3[1:3]
```

```
[1] 11 12 13
```

Já se for necessário buscar dados de índices separados podemos usar a função `c` (combinação), que combina valores para formar um vetor.

```
objeto3[c(1,3,5)]
```

```
[1] 11 13 15
```

O sinal de menos pode ser combinado com o índice e isto causará a exclusão do índice da busca. Por exemplo, usar o índice -5 retorna todos os elementos, exceto o quinto.

```
objeto3[-5]
```

```
[1] 11 12 13 24
```

```
objeto3[c(-2, -4)] # de forma combinada
```

```
[1] 11 13 15
```

### 1.4.1 Coerção

Quando vetores recebem dados de um tipo diferente o R tenta fazer uma operação de **coerção**, transformando os valores a fim de “atender” a todos. Nem sempre esta operação é possível e ela muitas vezes altera o vetor original. No exemplo abaixo o valor da posição 1 do vetor será atualizado para receber a letra **A**. Como o vetor originalmente era do tipo `integer`, o R fará a conversão dos valores para tipo `character`. Desta forma operações matemáticas não serão mais possíveis sobre este vetor.

```
objeto3[1] <- 'A'  
objeto3
```

```
[1] "A" "12" "13" "24" "15"
```

```
objeto3 + 10
```

Error in objeto3 + 10: argumento não-numérico para operador binário

## 1.5 Usando Funções

O coração da linguagem R são suas funções. Através delas são feitas as mais diversas operações sobre os objetos. Basicamente funções devem ser usadas através de seus nomes e com os argumentos dentro de parênteses: `funcao(argumento1, argumento2, ...)`.

Por exemplo, a função `typeof` exige a informação de um argumento (um objeto do R). O R processa esta função e devolve seu retorno, no caso qual o tipo do **objeto1**.

```
typeof(objeto1)
```

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

A função `is.vector`, por sua vez, testa se um objeto é um vetor e retorna um valor lógico como resposta do teste, `TRUE` (verdadeiro) ou `FALSE` (falso).

```
is.vector(5)
```

```
[1] TRUE
```

O valor armazenado em um objeto pode ser visualizado com a função `print`.

```
print(objeto1)
```

```
[1] 10
```

### 1.5.1 Argumentos de Funções

As funções em R podem ter diversos argumentos e muitas vezes estes argumentos possuem valores definidos por padrão. Assim caso o usuário não informe nenhum valor para os argumentos da função esta usará os valores previamente definidos em seu código.

Importante notar que os argumento possuem nomes e estes nomes podem ser omitidos na chamada da função. Voltemos a função `typeof`, ela possui apenas um argumento de nome `x`. Inserindo o argumento na função de forma explícita se obtém mesmo resultado anterior.

```
typeof(x = objeto1)
```

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

Nos casos de omissão do nome dos argumentos, estes receberão os valores informados de acordo com a ordem presente no código. Por exemplo a função `rep.int` retorna os valores indicados no argumento `x` *n* (argumento **times**) vezes.

```
rep.int(5, 4)
```

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

```
rep.int(x = 5, times = 4)
```

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

Perceba que os argumentos podem ser informados em ordem diversa, entretanto devem ser atribuídos de forma explícita. Veja que `rep.int(times = 4, x = 5)` é diferente de `rep.int(4, 5)`.

```
rep.int(times = 4, x = 5)
```

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

```
rep.int(4, 5)
```

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

#### **i** Nota

Algumas funções não possuem argumentos e “apenas” executam seu código, não exigindo interação de entrada por parte do usuário, como por exemplo as funções `Sys.Date()` e `Sys.time()`, que retornam a data e data e hora respectivamente.

### 1.5.2 Armazenando Retorno

Até aqui as funções foram usadas de forma que seus retornos foram apenas exibidos no console. Para que o valor retornado por uma função seja armazenado em um objeto se faz o uso do operador de atribuição.

```
tipo_objeto1 <- typeof(objeto1)
print(tipo_objeto1)
```

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

Agora o objeto **tipo\_objeto1** armazena o valor “*double*” que foi retornado pelo função `typeof`. Veja que a função `is.double` que testa se o objeto é `double` retorna `FALSE`, pois o objeto **tipo\_objeto1** recebeu um valor em formato texto. Dentro deste texto está contida a palavra *double*, mas isto não significa que o tipo do vetor **tipo\_objeto1** passou a ser `double`. Cuidado para não confundir o tipo do objeto com seu conteúdo.

```
is.double(tipo_objeto1)
```

```
[1] FALSE
```

```
typeof(tipo_objeto1)
```

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

```
is.character(tipo_objeto1)
```

```
[1] TRUE
```

## 1.6 Exibindo Objetos

O R possui a função `ls` que exhibe os objetos existentes no ambiente. Veja que a função `ls` possui valores padrão em seus argumentos, assim ela pode ser processada apenas com a digitação do código `ls()`. Mais detalhes em [Funções](#).

```
ls()
```

```
[1] "objeto1"      "objeto2"      "objeto3"      "repo"         "tipo_objeto1"
```

## 1.7 Remoção de Objetos

Objetos podem ser removidos (excluídos) com a função `rm`.



```
rm(objeto2)
ls()
```

```
[1] "objeto1"      "objeto3"      "repo"         "tipo_objeto1"
```

## 1.8 Iniciando o R...Studio

R é uma linguagem de programação e não está focada em oferecer uma interface sofisticada de interação com o usuário. Este papel fica por conta de outras ferramentas, como o **RStudio**, o **IDE** mais usado para a linguagem. Na prática “ninguém” usa o R puro para desenvolver seus projetos.

Desta forma usaremos o RStudio como ferramenta de desenvolvimento, pois ela irá nos fornecer muitas funcionalidades como preenchimento de código (*code completion*), janelas para instalar pacotes, janelas com arquivos de scripts, navegação por pastas, visualização e exportação de gráfico e claro comunicação direta com o R.

Ao longo deste livro serão usadas diversas funcionalidades do RStudio. Porém o foco será sempre no conteúdo, pois o detalhamento das principais funcionalidades do RStudio é tratada em seção [específica](#).

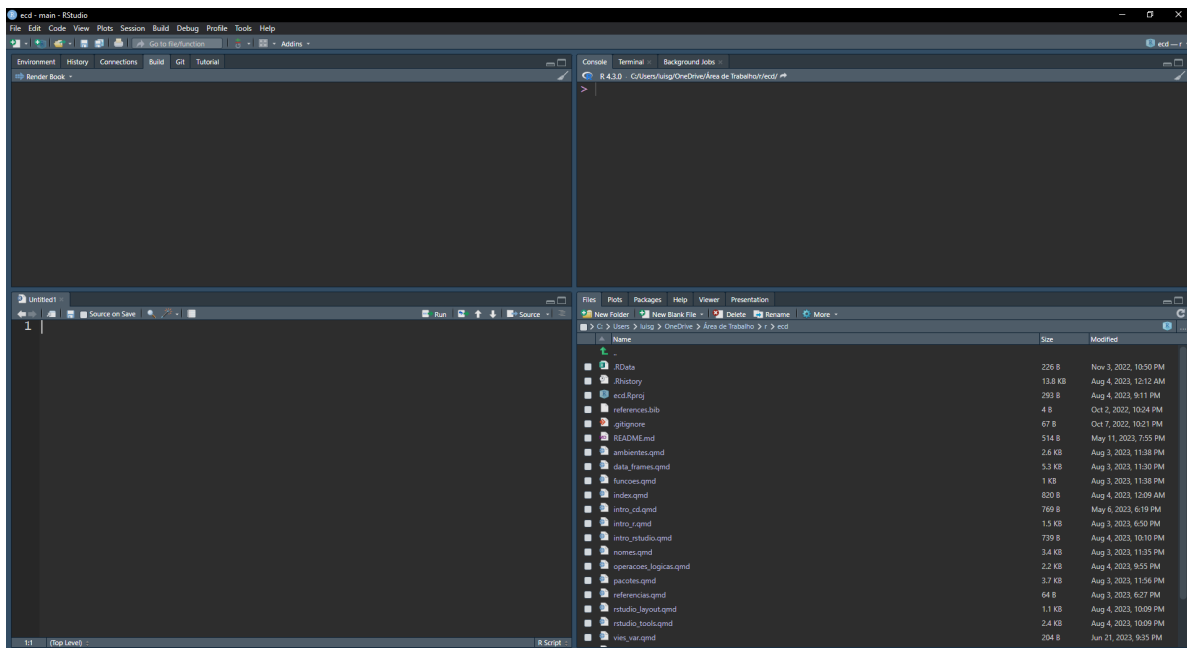
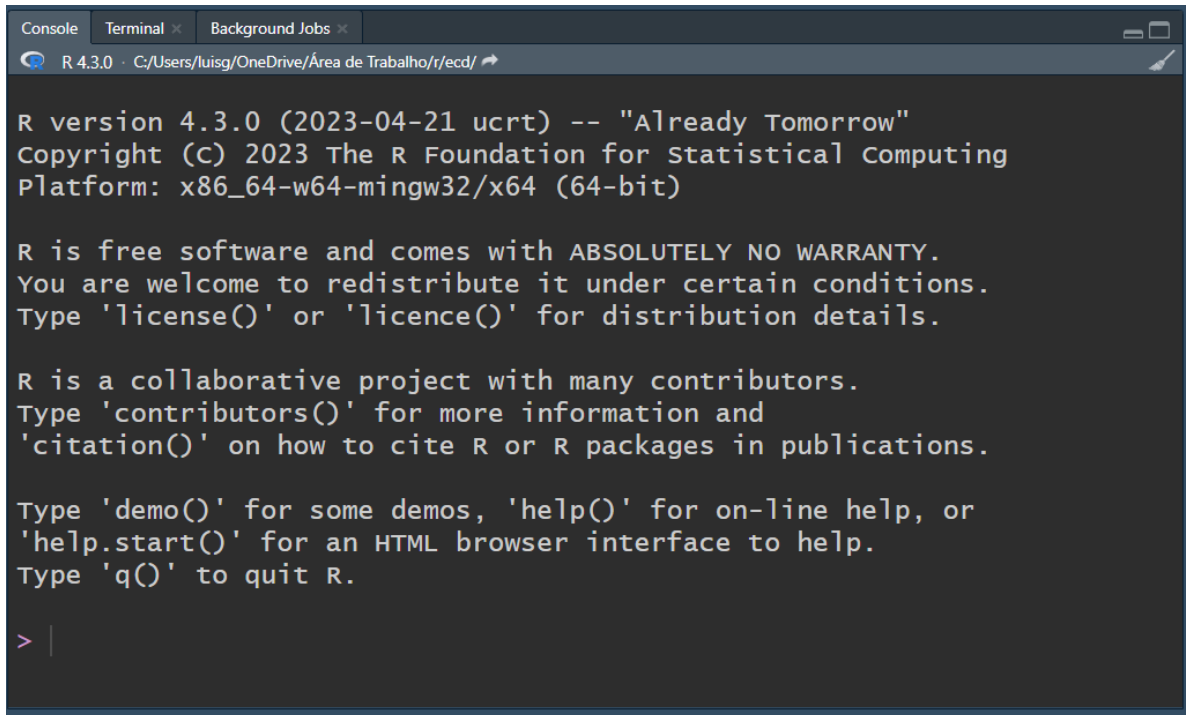


Figura 1.5: Tela Inicial do RStudio

Neste capítulo focaremos no painel **Console**, que “abriga” o R e no **Source**, que permite utilização de scripts.



```
Console Terminal Background Jobs
R 4.3.0 · C:/Users/luisg/OneDrive/Área de Trabalho/r/ecd/

R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"
Copyright (C) 2023 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Figura 1.6: Aba Console

### **i** Outros IDE's

Além do RStudio existem outras ferramentas para utilização em conjunto com R, entre elas: [Jupyter](#), [VS Code](#) e [RKWard](#).

## 1.9 Trabalhando com Scripts

Scripts são arquivos de texto que recebem códigos e conforme desejo do usuário são enviados ao console para execução. Na prática usar o console diretamente é útil para pequenas operações. No Rstudio você pode criar um script em File > New File > R Script. O arquivo de script será aberto no painel **Source**.

Para executar comandos de um arquivo de script você pode usar atalhos de teclado (Ctrl + Enter) ou através do botão Run no topo superior direito da aba Source. Ambas opções executam ou a linha corrente ou a parte do texto selecionada.

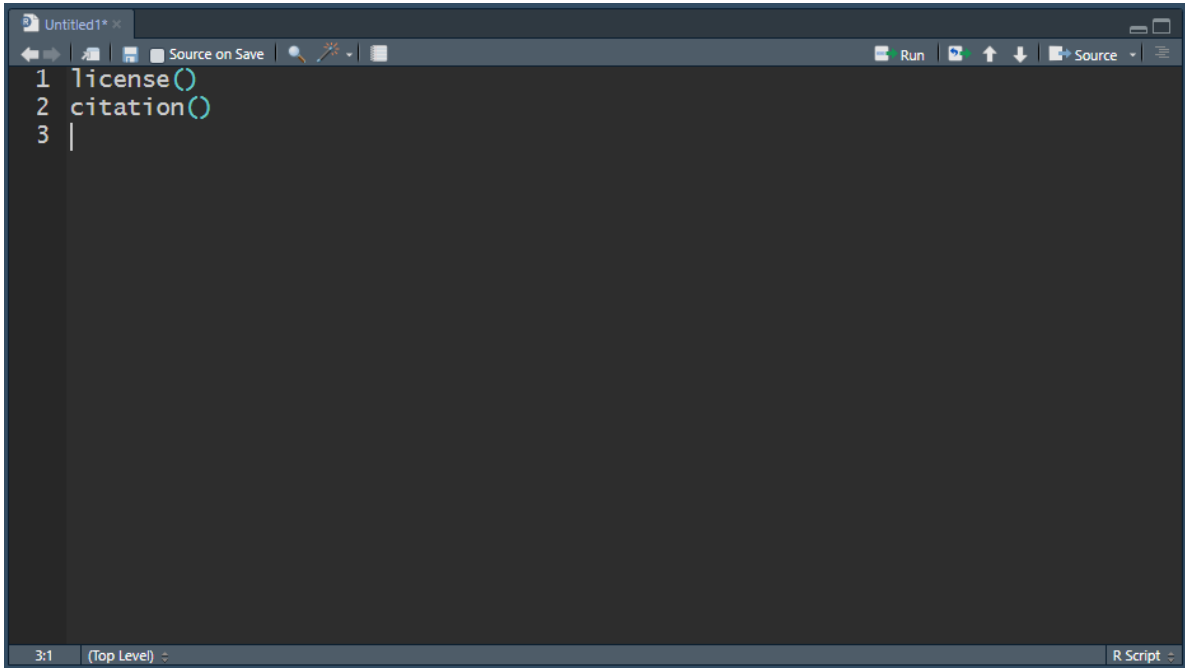


Figura 1.7: Aba Source

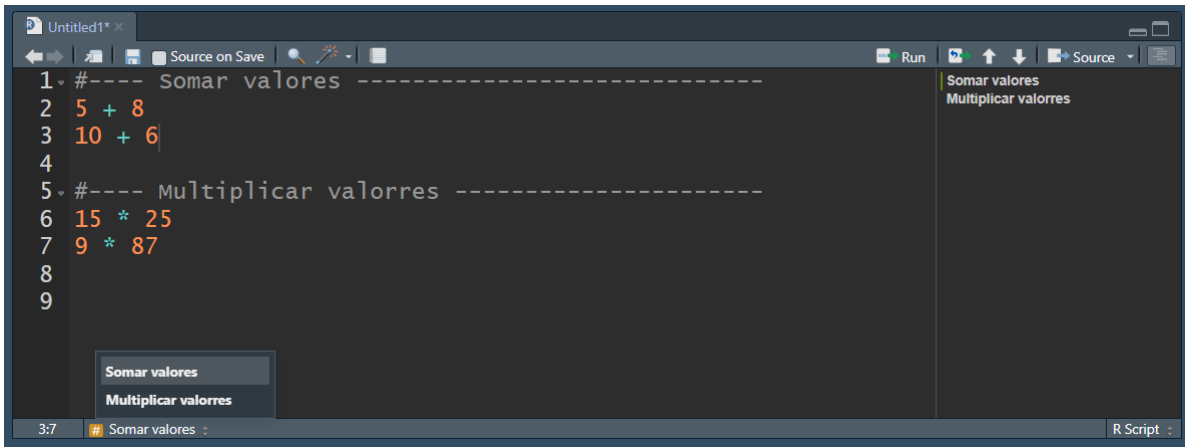


Figura 1.8: Script - Painel Source

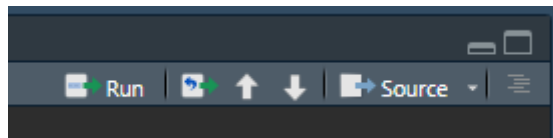
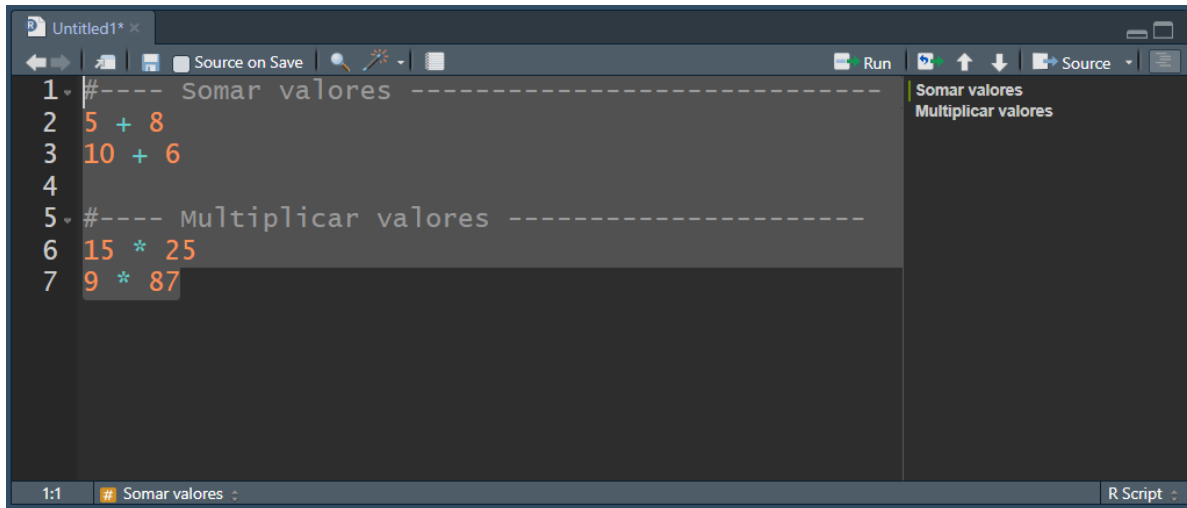


Figura 1.9: Source - Botão Run

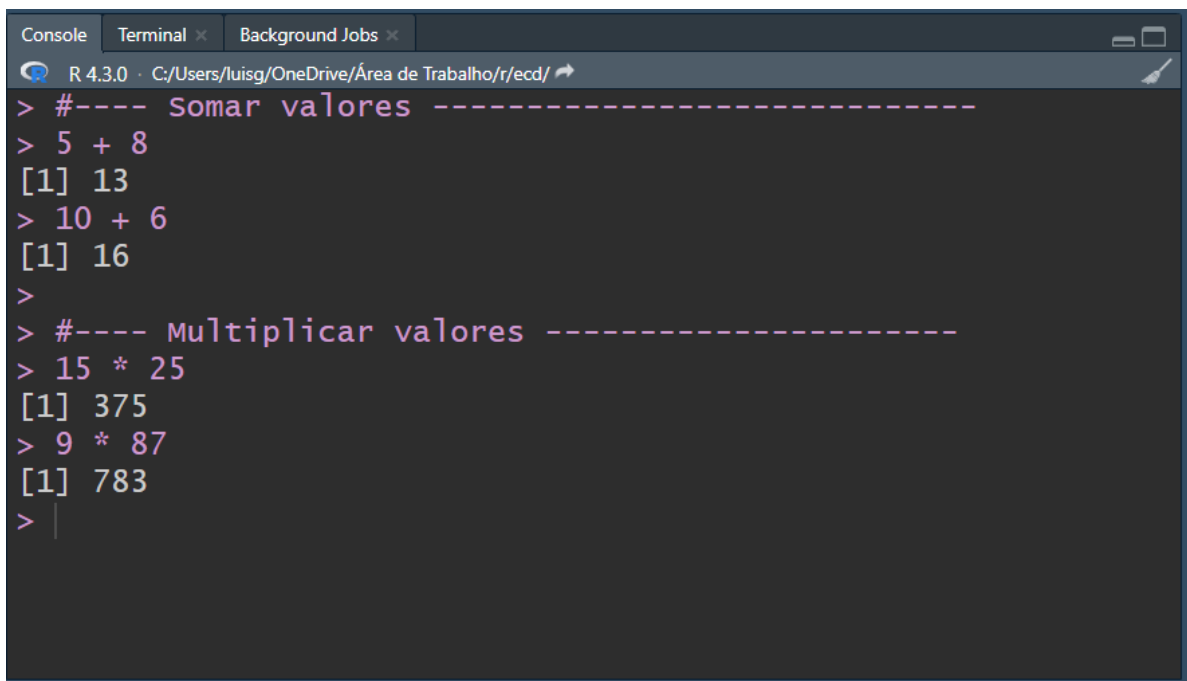


The image shows an R script editor window titled 'Untitled1\*.x'. The script contains the following code:

```
1 #---- Somar valores -----  
2 5 + 8  
3 10 + 6  
4  
5 #---- Multiplicar valores -----  
6 15 * 25  
7 9 * 87
```

The right-hand pane shows the output of the script, which is currently empty. The status bar at the bottom indicates the current line is 1:1 and the code is '# Somar valores'.

Figura 1.10: Script - Código a Processar



The image shows an R console window with the following output:

```
R 4.3.0 · C:/Users/luisg/OneDrive/Área de Trabalho/r/ecd/  
> #---- somar valores -----  
> 5 + 8  
[1] 13  
> 10 + 6  
[1] 16  
>  
> #---- multiplicar valores -----  
> 15 * 25  
[1] 375  
> 9 * 87  
[1] 783  
> |  
>
```

Figura 1.11: Console - Resultado

### 1.9.1 Comentários

R aceita comentários em seu código através do caractere suspenso (*hashtag*), '#'. Comentários são muito importantes para facilitar a leitura do código posteriormente. Uma forma interessante de organizar o seu código é criar uma linha de comentário para separação de etapas de processamento. Perceba que as linhas de comentários foram passadas para o console e este não emitiu nenhuma mensagem, tampouco efetuou qualquer operação.

#### Porquê...

Em operações mais complexas procure colocar comentários que expliquem os motivos de se executar alguma operação e não o que o código está fazendo. Foque no **'porquê'** de cada operação e não no **'o que'**.

---

Última atualização: 11/10/2024 - 21:51:13

## 2 Nomeando Objetos

Status

### 2.1 Regras

A linguagem R aceita muitas possibilidades para nomeação de objetos. Inclusive podem ser criados objetos com espaços em seus nomes e até mesmo com caracteres especiais (desde que entre aspas ou crases).

```
x <- 10  
  
.x <- 10  
  
`nome com espaco` <- 55  
  
'teste 1' <- 2
```

#### Nomes Significativos

Escolha nomes intuitivos e que facilitem a identificação do conteúdo armazenado nos objetos.

Um objeto criado através do uso de aspas ou crases tem seu conteúdo acessado quando “chamado” com crases (*backticks*). Aspas são entendidas como sinalização para strings e assim não retornam o conteúdo do objeto e sim a própria string informada.

```
'teste 1' # retorna como string
```

```
[1] "teste 1"
```

```
`teste 1` # Exibe conteúdo do objeto
```

```
[1] 2
```

### 2.1.1 Primeiro Caractere

Existem algumas regras para iniciar o nome dos objetos. Alguns caracteres “especiais” não podem ser usados, bem como os números.

```
$x <- 10
```

```
Error: <text>:1:1: '$' inesperado
1: $
   ^
```

```
55x <- 10
```

```
Error: <text>:1:3: unexpected symbol
1: 55x
   ^
```

Perceba que números podem ser usados nos nomes, desde que o primeiro caractere seja ‘válido’. Mas o mesmo não ocorre com caracteres “especiais”.

```
x55x <- 888
```

```
x55x
```

```
[1] 888
```

```
x$ <- 10
```

```
Error: <text>:1:4: unexpected assignment
1: x$ <-
   ^
```

Uma alternativa se dá mais uma vez com o uso de aspas ou crases. Com elas é possível ‘burlar’ estas limitações.

```
`teste @!&` <- 123456
```

```
`teste @!&`
```

```
[1] 123456
```

```
'55 teste @!&' <- 10  
`55 teste @!&`
```

```
[1] 10
```

Apesar de possível, objetos com nomes mais complicados como os exemplificados acabam tornando a vida do programador um pouco mais difícil. Em geral, evite caracteres especiais e espaços nos nomes. Caso algum dado (bases de dados) seja carregado de arquivo externo com este tipo de caracteres, faça a uniformização dos nomes o quanto antes.

### 2.1.1.1 Objetos “Ocultos”

Objetos podem ser criados com “.” no início de seus nomes desde que o segundo caractere seja uma letra. Estes são objetos “ocultos” e portanto não aparecem em um comando `ls` “puro”, por exemplo. Tampouco são exibidos na aba Environment do RStudio. Para visualizá-los através da função `ls` deve ser usado o parâmetro `all.names = T`.

```
ls()
```

```
[1] "55 teste @!&"      "nome com espaco" "repo"           "teste @!&"  
[5] "teste 1"           "x"                "x55x"
```

```
ls(all.names = T)
```

```
[1] ".main"              ".QuartoInlineRender" ".x"  
[4] "55 teste @!&"      "nome com espaco"     "repo"  
[7] "teste @!&"         "teste 1"             "x"  
[10] "x55x"
```

### 2.1.2 Case Sensitive

R é uma linguagem *case sensitive*, ou seja, ela diferencia maiúsculas de minúsculas. Assim um objeto com nome de Teste é diferente teste, tesTe, TESTE...



```
teste <- 10
Teste <- 15
tesTe <- 20
TESTE <- 25
```

```
ls()
```

```
[1] "55 teste @!&"      "nome com espaco" "repo"           "teste"
[5] "tesTe"             "Teste"           "TESTE"          "teste @!&"
[9] "teste 1"           "x"               "x55x"
```

### 💡 Campos de Tabelas

Campos (variáveis) de dados tabulados, como planilhas de Excel, seguem as mesmas regras. Este tipo de dado será tratado no capítulo sobre **data frames**.

## 2.2 Resumo

Tabela 2.1: Resumo das Regras para Nomes

Caracteres	Regra	Exceção	Exemplo
Letras	Permitido		objeto variavel
Números	Permitido, após primeiro caractere	Iniciado com '.'	objeto1 1objeto .1objeto
Espaços	Não permitido	Permitido com uso de aspas ou crases	teste 1 'teste 2' '2 teste'
Caracteres especiais	Não permitido	Permitido com uso de aspas ou crases	#teste '# teste' 't #\$\$%'
Ponto '.'	Uso livre inclusive no início		objeto.2 .objeto.2

## 2.3 Convenções

Conforme o seu código em R (e de outra linguagem qualquer) for crescendo você perceberá rapidamente a necessidade de identificar de forma intuitiva os objetos criados. Assim, é muito interessante a utilização de alguma convenção para facilitar sua vida. Existem diversas delas, como **camelCase**, **snake\_case**, **SCREAMING\_SNAKE\_CASE**, **PascalCase**, etc.

```
# camelCase
objetoTeste <- 'Teste camelCase'

# snake_case
objeto_teste <- 'Teste snake_case'
```

Um bom guia é o [The tidyverse style guide](#). Tenha sempre em mente que seu código deve ser lido com facilidade no futuro e muitas vezes por outros usuários.

Neste material os nomes de objetos e derivados seguirão a tabela abaixo. Estas definições foram escolhidas a fim de uniformizar o conteúdo apresentado e se baseiam em experiência de uso e no **Tidyverse Style Guide**. Mais detalhes em [Convenções](#).

Tipo Objeto	Convenção	Exemplo
Data.frame, tibble ou data.table	snake_case iniciado por <b>df</b> ( <b>data frame</b> )	df_clientes
Variáveis de datasets	SCREAMING_SNAKE_CASE	NOME_CLIENTE
Funções	camelCase iniciado por <b>fn</b> , sendo a primeira palavra após fn um verbo	fnBuscarClientes
Demais (vetores, listas, etc.)	snake_case	nomes_cidades

### Dica

Evite usar “.” em nome de objetos, pois através do **ponto** o R acessa funções (métodos) de acordo com a classe do objeto. Usar o ponto pode causar certa confusão. Mais detalhes [Funções](#).

---

Grolemund (2014)

R Core Team (2023a)

Wikipedia (2023)

Última atualização: 18/09/2023 - 19:55:08

# 3 Vetores

Status

## 3.1 Introdução

**Vetores** são o tipo de estrutura de dados mais básica no R. Os vetores podem ser criados de diversas formas. Serão criados dois vetores uma com a função `seq`, que cria uma sequência de acordo com os parâmetros informados, e com o operador `:`.

```
vetor_1 <- seq(1, 10)
vetor_2 <- 1:10
```

Podemos testar se dois objetos são idênticos com a função `identical`.

```
identical(vetor_1, vetor_2)
```

```
[1] TRUE
```

Uma outra função muito útil para avaliar um objeto é a função `str`, que exibe a estrutura do objeto.

```
str(vetor_1)
```

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

```
str(vetor_2)
```

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

## 3.2 Selecionando Elementos

Podemos selecionar elementos de **vetores** com os operadores `[]` e `[[[]]`. A diferença principal é que o primeiro pode selecionar diversos elementos, enquanto o segundo apenas um.

```
vetor_1[5]
```

```
[1] 5
```

```
vetor_1[[5]]
```

```
[1] 5
```

```
vetor_2[5:6]
```

```
[1] 5 6
```

```
vetor_2[[5:6]]
```

```
Error in vetor_2[[5:6]]: attempt to select more than one element in vectorIndex
```

## 3.3 Nomeando Componentes

Os componentes de vetores podem ser nomeados com a função `names` e posteriormente acessados pelo seu nome.

```
names(vetor_1) <- c('componente_1',  
                  'componente_2')
```

```
vetor_1['componente_1']
```

```
componente_1  
1
```

Usando a função `str` podemos ver que agora o vetor possui **atributos**, neste caso `nomes`. Como foram inseridos apenas nomes para os primeiros dois elementos os demais possuem valor `NA`.

```
str(vetor_1)
```

```
Named int [1:10] 1 2 3 4 5 6 7 8 9 10  
- attr(*, "names")= chr [1:10] "componente_1" "componente_2" NA NA ...
```

```
attributes(vetor_1)
```

```
$names  
[1] "componente_1" "componente_2" NA NA NA  
[6] NA NA NA NA NA
```

### 3.4 Testando

Podemos testar se um objeto é um vetor com a função `is.vector`.

```
is.vector(vetor_1)
```

```
[1] TRUE
```

```
is.vector(1)
```

```
[1] TRUE
```

```
is.vector('A')
```

```
[1] TRUE
```

---

R Core Team (2023a)

Última atualização: 11/10/2024 - 21:50:40

## 4 Tipos de Dados

Status □□□

### 4.1 Lógico (*Logical*)

Dados do tipo lógico podem assumir basicamente dois valores, verdadeiro (TRUE) e falso (FALSE). Estes valores podem ser abreviados por T e F, respectivamente.

```
is.logical(TRUE)
```

```
[1] TRUE
```

```
is.logical(FALSE)
```

```
[1] TRUE
```

```
typeof(T)
```

```
[1] "logical"
```

```
typeof(F)
```

```
[1] "logical"
```

```
is.logical(1)
```

```
[1] FALSE
```

```
is.logical(0)
```

```
[1] FALSE
```

Testes lógicos retornam valores lógicos.

```
is.logical(5 > 5)
```

```
[1] TRUE
```

```
typeof(10 < 9)
```

```
[1] "logical"
```

TRUE e FALSE são palavras reservadas, portanto não podem ser usadas como objetos.

```
TRUE <- 10
```

```
Error in TRUE <- 10: lado esquerdo da atribuição inválida (do_set)
```

#### 4.1.1 Valores Faltantes (NA)

Em R a constante `NA` (*Not Available*) é usada para expressar valores faltantes. O `NA` é do tipo lógico, mas pode ser atribuído a vetores de outros tipos (exceto *raw*) através de coerção.

```
typeof(NA)
```

```
[1] "logical"
```

```
is.na(NA)
```

```
[1] TRUE
```

```
vetor_1 <- c(1:5, NA, 6:10)  
vetor_1
```

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



```
typeof(vetor_1)
```

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

Os valores faltantes são muito importantes na análise de dados, pois podem influenciar cálculos e transformações. Até mesmo operações aritméticas básicas são influenciadas pelo valores faltantes.

```
1 + 5 + NA
```

```
[1] NA
```

Funções também são influenciadas pela presença dos dados faltantes. A função `max`, por exemplo, que retorna o maior valor dentre os informados, retorna `NA` se este estiver presente.

```
max(vetor_1)
```

```
[1] NA
```

#### 4.1.1.1 Tratando Dados Faltantes

R oferece algumas funções para tratameto de dados faltantes. Abaixo exemplo de uso da função `na.omit`, que devolve os elementos não `NA`. Caso o objeto não contenha valores faltantes, ele será “devolvido” de forma integral.

```
c(1, NA, 3) |> na.omit()
```

```
[1] 1 3  
attr("na.action")  
[1] 2  
attr("class")  
[1] "omit"
```

Muitas funções oferecem o argumento `na.rm` para remoção dos valores faltantes, exemplos: `sum`, `max`, `min`, `prod`.

```
sum(c(1, 2, NA), na.rm = T)
```

```
[1] 3
```

```
prod(c(1, 2, NA), na.rm = T)
```

```
[1] 2
```

## 4.2 Inteiros (*Integer*)

Números inteiros são do tipo `integer` e devem ser criados com a letra **L** ao seu lado. Sem este indicador, por padrão, o R entende o número como do tipo `double`.

```
typeof(1L)
```

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

```
typeof(1)
```

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

Para testar se um número é do tipo inteiro pode-se utilizar a função `is.integer`.

```
is.integer(1)
```

```
[1] FALSE
```

Para transformar um valor para inteiro usa-se a função `as.integer`.

```
is.integer(as.integer(1))
```

```
[1] TRUE
```

```
as.integer(1.99)
```

```
[1] 1
```

## 4.3 Ponto Flutuante (*Double*)

De forma grosseira, *doubles* são valores numéricos com decimais.

```
is.double(1)
```

```
[1] TRUE
```

### 4.3.1 *Not a Number* (NaN)

Valores NaN ('não um número') são valores de tipo **double**. Valores NaN impactam operações lógicas e matemáticas.

```
typeof(NaN)
```

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

```
NaN > 10
```

```
[1] NA
```

```
NaN * 10
```

```
[1] NaN
```

```
10/NaN
```

```
[1] NaN
```

```
5 + NaN
```

```
[1] NaN
```

### 4.3.2 Inf e -Inf

No R os valores **Inf** e **-Inf** representam infinito e infinito negativo, respectivamente.

Estes valores impactam cálculos.

```
is.infinite(Inf)
```

```
[1] TRUE
```

```
5 + Inf
```

```
[1] Inf
```

```
Inf + Inf
```

```
[1] Inf
```

```
-Inf * -1
```

```
[1] Inf
```

```
Inf - Inf
```

```
[1] NaN
```

Mas operações lógicas seguem o 'senso comum'.

```
Inf > 10
```

```
[1] TRUE
```

```
Inf > -Inf
```

```
[1] TRUE
```

```
Inf == Inf
```

```
[1] TRUE
```

Valores infinitos podem ser gerados se muito grandes ou por valores divididos por zeros.

```
10^308
```

```
[1] 1e+308
```

```
10^309
```

```
[1] Inf
```

```
10/0
```

```
[1] Inf
```

```
-10/0
```

```
[1] -Inf
```

Mas veja que zero dividido por zero é NaN.

```
0/0
```

```
[1] NaN
```

```
0/Inf
```

```
[1] 0
```

## 4.4 Fatores (*Factor*)

---

Última atualização: 11/10/2024 - 21:53:49

# 5 Ambientes

Status □□□

## 5.1 Global Env

O **Global Env** é o ambiente “atual” do usuário. É nele que ficam armazenados por padrão os objetos e as funções criadas pelos usuários por exemplo. Ele pode ser “visualizado” com os comandos abaixo:

```
globalenv()
```

```
<environment: R_GlobalEnv>
```

```
.GlobalEnv
```

```
<environment: R_GlobalEnv>
```

Os objetos presentes no ambiente desejado podem ser visualizados com a função `ls`.

```
variavel <- 5  
ls(globalenv())
```

```
[1] "repo"      "variavel"
```

```
ls()
```

```
[1] "repo"      "variavel"
```

## 5.2 Ambiente de Pacotes

Os pacotes também possuem ambientes e podemos listar seu “conteúdo” com a função `ls`. Abaixo usando `ls` para mostrar os 10 primeiros elementos presentes no ambiente do pacote `data.table`.

```
library(data.table)
as.environment("package:data.table")
```

```
<environment: package:data.table>
attr("name")
[1] "package:data.table"
attr("path")
[1] "C:/Users/luisg/OneDrive/Área de Trabalho/r/ecd/.packages/data.table"
```

```
ls(as.environment('package:data.table'))[1:10]
```

```
[1] "%between%" "%chin%" "%flike%" "%ilike%" "%inrange%"
[6] "%like%" "%notin%" "%plike%" "!=" "[.data.table"
```

## 5.3 Ambientes “Pai”

Cada ambiente possui um ambiente de nível superior associado, com exceção do `R_EmptyEnv`.

```
# Ambiente superior ao GlobalEnv
parent.env(.GlobalEnv)
```

```
<environment: package:data.table>
attr("name")
[1] "package:data.table"
attr("path")
[1] "C:/Users/luisg/OneDrive/Área de Trabalho/r/ecd/.packages/data.table"
```

```
# Ambiente superior ao do pacote stats e base
parent.env(as.environment("package:stats"))
```

```
<environment: package:graphics>
attr(,"name")
[1] "package:graphics"
attr(,"path")
[1] "C:/Program Files/R/R-4.4.1/library/graphics"
```

```
parent.env(as.environment("package:base"))
```

```
<environment: R_EmptyEnv>
```

## 5.4 Criando Ambientes

Em R é possível que se faça a criação de novos ambientes.

```
amb1 <- new.env()
```

```
amb1
```

```
<environment: 0x0000028af565bf80>
```

```
parent.env(amb1)
```

```
<environment: R_GlobalEnv>
```

Objetos criados dentro de um ambiente podem ser acessados através do operador \$ após o nome do ambiente. Também é possível utilizar a função ls com o nome do ambiente desejado para que sejam listados seus objetos.

```
# Objeto x do amb1
amb1$x <- 10
amb1$y <- 99

# Objeto x do GlobalEnv
x <- 15

x
```

```
[1] 15
```



```
amb1$x
```

```
[1] 10
```

```
ls(amb1)
```

```
[1] "x" "y"
```

```
amb1$x * amb1$y
```

```
[1] 990
```

---

Grolemund (2014)

Dowle e Srinivasan (2023)

Mastropietro (2019)

Última atualização: 11/10/2024 - 21:49:26

# 6 Operações Lógicas

Status

A linguagem R oferece uma série de operadores para utilização em testes lógicos.

## 6.1 Operadores Relacionais

Operador	Função
>	Maior que
<	Menor que
>=	Maior ou igual a
<=	Menor ou igual a
==	Igual a
!=	Diferente de

```
5 > 6
```

```
[1] FALSE
```

```
5 <= 6
```

```
[1] TRUE
```

```
5 == 6
```

```
[1] FALSE
```

```
5 != 6
```

```
[1] TRUE
```

## 6.2 Operadores Lógicos

Operador	Função
!	Negação
&	E
	Ou
xor	Ou exclusivo
isTRUE	Testa se verdadeiro
isFALSE	Testa se falso

```
!FALSE
```

```
[1] TRUE
```

```
!TRUE
```

```
[1] FALSE
```

```
5 > 6
```

```
[1] FALSE
```

```
!5 > 6
```

```
[1] TRUE
```

```
isTRUE(5 > 6)
```

```
[1] FALSE
```

```
isFALSE(5 > 6)
```

```
[1] TRUE
```

### 6.2.1 Ou Exclusivo (Xor)

O operador `xor` fornece saída verdadeira quando apenas um dos valores for verdadeiro.

```
# Falso XOR Falso = Falso
xor(5 > 6, 6 > 9)
```

[1] FALSE

```
# Verdadeiro XOR Verdadeiro = Falso
xor(5 > 4, 6 > 5)
```

[1] FALSE

```
# Verdadeiro XOR Falso = Verdadeiro
xor(5 > 4, 6 > 9)
```

[1] TRUE

```
# Falso XOR Verdadeiro = Falso
xor(5 > 6, 6 > 5)
```

[1] TRUE

### 6.3 Precedência de Operadores Lógicos e Relacionais

Na utilização de testes lógicos é importante observar a ordem (precedência) de aplicação dos operadores. O uso de parênteses altera a o escopo de aplicação dos operadores.

Tabela 6.3: Precedência de Operadores

Ordem	Operador
1	<, >, <=, >=, ==, !=
2	!
3	&
4	

Abaixo alguns testes.

```
# Falso E Falso = Falso
5 > 6 & 4 > 5
```

[1] FALSE

```
# Verdadeiro E Verdadeiro = Verdadeiro
!5 > 6 & !4 > 5
```

[1] TRUE

```
# Verdadeiro E Falso = Falso
!5 > 6 & 4 > 5
```

[1] FALSE

```
# Negação de(Falso E Falso) = Verdadeiro
!(5 > 6 & 4 > 5)
```

[1] TRUE

```
# Falso OU Falso = Falso
5 > 6 | 4 > 5
```

[1] FALSE

```
# Verdadeiro OU Verdadeiro = Verdadeiro
!5 > 6 | !4 > 5
```

[1] TRUE

```
# Verdadeiro OU Falso = Verdadeiro
!5 > 6 | 4 > 5
```

[1] TRUE

```
# Negação de (Falso OU Falso) = Verdadeiro  
!(5 > 6 | 4 > 5)
```

```
[1] TRUE
```

## 6.4 Funções isTRUE e isFALSE

Estas duas funções oferecem retorno lógico a partir de uma entrada. Exemplo:

```
isTRUE(5 > 6)
```

```
[1] FALSE
```

```
isFALSE(5 > 6)
```

```
[1] TRUE
```

Estas funções oferecem vantagens em testes que possam possuir NAs. Exemplo:

```
isTRUE(NA > 5)
```

```
[1] FALSE
```

```
isFALSE(5 > NA)
```

```
[1] FALSE
```

```
NA > 5
```

```
[1] NA
```

## 6.5 Funções is.

R também oferece uma gama enorme de funções que testam objetos, abaixo alguns exemplos:

```
is.character('A')
```

```
[1] TRUE
```

```
is.numeric(5)
```

```
[1] TRUE
```

```
is.double(5)
```

```
[1] TRUE
```

```
is.na(NA)
```

```
[1] TRUE
```

```
is.na(NaN)
```

```
[1] TRUE
```

```
is.nan(NA)
```

```
[1] FALSE
```

```
is.nan(NaN)
```

```
[1] TRUE
```

Note que NaN é considerado *Not Avalibale*, ao passo que NA não é considerado *Not a Number*.

## 6.6 All e any

As funções `all` e `any` testam se vetores possuem valores TRUE, todos ou pelo menos 1, respectivamente.

```
vetor_logico <- c(T, T, T)
```

```
all(vetor_logico)
```

```
[1] TRUE
```

```
any(vetor_logico)
```

```
[1] TRUE
```

Apenas valores FALSE:

```
vetor_logico <- c(F, F, F)
```

```
all(vetor_logico)
```

```
[1] FALSE
```

```
any(vetor_logico)
```

```
[1] FALSE
```

Vetor com apenas 1 valor TRUE.

```
vetor_logico <- c(T, F, F, F)
```

```
all(vetor_logico)
```

```
[1] FALSE
```

```
any(vetor_logico)
```

```
[1] TRUE
```

Vetor com todos os valores falsos (FALSE).



```
vetor_logico <- c(F, F, F, F)
all(vetor_logico)
```

[1] FALSE

```
any(vetor_logico)
```

[1] FALSE

Note que a presença de valores NA altera completamente o retorno da função `all`, mas não da função `any`.

```
vetor_logico <- c(T, F, T, NA)
all(vetor_logico)
```

[1] FALSE

```
any(vetor_logico)
```

[1] TRUE

Isto ocorre, pois a função `any` só retorna NA se existirem valores NA e FALSE no vetor.

```
vetor_logico <- c(F, NA, NA, NA)
any(vetor_logico)
```

[1] NA

```
vetor_logico <- c(NA, NA, NA, NA)
any(vetor_logico)
```

[1] NA

Ambas as funções aceitam o parâmetro `na.rm`, que remove os valores NA antes de fazer a avaliação.

```
vetor_logico <- c(T, T, T, NA)
all(vetor_logico, na.rm = T)
```

[1] TRUE

```
any(vetor_logico, na.rm = T)
```

[1] TRUE

Com presença de valores falsos;

```
vetor_logico <- c(T, F, F, NA)
all(vetor_logico, na.rm = T)
```

[1] FALSE

```
any(vetor_logico, na.rm = T)
```

[1] TRUE

Equivalente a:

```
vetor_logico <- c(T, F, F)
all(vetor_logico, na.rm = T)
```

[1] FALSE

```
any(vetor_logico, na.rm = T)
```

[1] TRUE

### 6.6.1 AnyNA

Esta função teste se pelo menos um elemento é NA.

```
anyNA(c(10, NA))
```

```
[1] TRUE
```

```
anyNA(c(NA, NA))
```

```
[1] TRUE
```

```
anyNA(c(2, 1))
```

```
[1] FALSE
```

## 6.7 Operador %in%

O operador binário %in% efetua teste de presença do objeto da esquerda (*left hand side*) no da direita (*right hand side*).

```
x <- c(1, 2, 3, 4)
```

```
y <- c(3, 4, 5)
```

```
x %in% y
```

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

```
y %in% x
```

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

Veja que o retorno é dado pelo tamanho do objeto da esquerda. Assim `x %in% y` é uma operação completamente diferente de `y %in% x`.

Em casos de itens repetidos o retorno segue a mesma lógica, avaliando o objeto da esquerda dentro do da direita:

```
x <- c(1, 2, 2)
```

```
y <- 2
```

```
x %in% y
```

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

```
y %in% x
```

```
[1] TRUE
```

---

R Core Team (2023c)

Última atualização: 11/10/2024 - 21:53:21

# 7 Listas

Status □□□

## 7.1 Introdução

**Listas** são objetos que armazenam outros objetos, podendo ser de variados tipos.

## 7.2 Criando Listas

Abaixo um exemplo de criação de **lista** através da função `list`. Os seus componentes serão um data frame, um vetor de números de 1 até 10 e um vetor com as letras do alfabeto.

```
lista_1 <- list(mtcars, 1:10, letters)
```

## 7.3 Acessando Componentes

Para acessar os elementos das listas pode-se usar o operador `[...]`.

```
class(lista_1[[1]])
```

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

Deve-se tomar cuidado ao usar o operador `[]`, pois este operador é genérico e não retorna o componente 1 da lista em sua “forma” original e sim de uma lista contendo o objeto da lista original. Desta forma, não é possível fazer extração de objetos do vetor retornado.

```
class(lista_1[1])
```

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

```
lista_1[2][5]
```

```
[[1]]  
NULL
```

Usando `[[...]]` o objeto retornado mantém sua forma original e a extração ocorre normalmente.

```
class(lista_1[[2]])
```

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

```
lista_1[[2]][5]
```

```
[1] 5
```

## 7.4 Nomeando Componentes

Os componentes das listas podem ser nomeados com a função `names`.

```
names(lista_1) <- c('df_mtcars', 'vetor', 'letras')  
lista_1$letras
```

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

```
identical(lista_1$letras, lista_1[[3]])
```

```
[1] TRUE
```

```
identical(lista_1$letras, lista_1[['letras']])
```

```
[1] TRUE
```

### 7.4.1 Nomes Abreviados

Para acessar componentes de listas nomeadas é possível informar seus nomes de forma abreviada.

```
lista_1$le
```

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

```
#equivalente a  
lista_1$letras
```

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

Veja que a abreviação deve identificar de forma exclusiva os componentes, caso contrário o valor retornado é NULL. A **lista\_3** possui dois componentes que começam com 'le' e assim não é possível fazer a seleção.

```
lista_3 <- list(letras = letters[1:10], letras_maiusculas = LETTERS[1:10])
```

```
lista_3$le
```

```
NULL
```

---

R Core Team (2023a)

Última atualização: 11/10/2024 - 21:51:19

# 8 Data Frames

Status □□□

## 8.1 O que são data frames ?

Conforme o R Core Team (2023b), **data frame** é a estrutura que imita de forma mais próxima um dataset do **SAS** ou **SPSS**. De forma resumida um **data frame** é uma estrutura tabular com colunas (variáveis, atributos, etc) e linhas (registros, casos, observações, instâncias, etc). Diferente de uma matriz um **data frame** pode ter diferentes tipos de dados em suas colunas.

Um **data frame** possui todas as colunas com o mesmo tamanho (quantidade de registros). A classe de um objeto **data frame** possui o nome **data.frame**. Abaixo pode ser visualizada a classe do **data frame iris** (muito usado em exemplos em Ciência de Dados) e também as primeiras linhas com o comando `head`.

```
class(iris)
```

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

```
head(iris)
```

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
1	5.1	3.5	1.4	0.2	setosa
2	4.9	3.0	1.4	0.2	setosa
3	4.7	3.2	1.3	0.2	setosa
4	4.6	3.1	1.5	0.2	setosa
5	5.0	3.6	1.4	0.2	setosa
6	5.4	3.9	1.7	0.4	setosa

Um **data frame** é na verdade uma **lista**, assim as operações efetuadas em **listas** possuem equivalência em **data frames**.



```
typeof(iris)
```

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

### 8.1.1 Criando Data Frames

Objetos da classe **data.frame** podem ser criados com a função `data.frame`.

Aqui serão usadas as convenções de nomes conforme capítulos [Nomeando Objetos](#) e [Convenções](#).

```
df_exemplo <- data.frame(  
  VAR_A = c(1:5),  
  VAR_B = c(101:105)  
)  
df_exemplo
```

	VAR_A	VAR_B
1	1	101
2	2	102
3	3	103
4	4	104
5	5	105

### 8.1.2 Aplicar convenções de nomes

Para continuar os próximos tópicos vamos trabalhar com um **data frame** (**df\_iris**) criado a partir do **data frame iris**. Faremos ajustes nos nomes deste data frame.

```
# criar data frame df_iris  
df_iris <- iris  
# mudar nomes para maiusculas  
names(df_iris) <- toupper(names(df_iris))  
# substituir '.' por '_'  
names(df_iris) <- gsub(names(df_iris), pattern = "\\.", replacement = "_")  
  
class(df_iris)
```

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

```
head(df_iris)
```

```
  SEPAL_LENGTH SEPAL_WIDTH PETAL_LENGTH PETAL_WIDTH SPECIES
1          5.1          3.5          1.4          0.2  setosa
2          4.9          3.0          1.4          0.2  setosa
3          4.7          3.2          1.3          0.2  setosa
4          4.6          3.1          1.5          0.2  setosa
5          5.0          3.6          1.4          0.2  setosa
6          5.4          3.9          1.7          0.4  setosa
```

## 8.2 Atributos

Os atributos “padrão” de um **data frame** são: `names`, `class` e `row.names`. É possível acessá-los com a função `attributes`. O atributo `names` também pode ser obtido com a função `names`.

```
attributes(df_iris)
```

```
$names
```

```
[1] "SEPAL_LENGTH" "SEPAL_WIDTH"  "PETAL_LENGTH" "PETAL_WIDTH"  "SPECIES"
```

```
$class
```

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

```
$row.names
```

```
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
[19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
[37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
[55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
[73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
[91] 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108
[109] 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126
[127] 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144
[145] 145 146 147 148 149 150
```

```
names(df_iris)
```

```
[1] "SEPAL_LENGTH" "SEPAL_WIDTH"  "PETAL_LENGTH" "PETAL_WIDTH"  "SPECIES"
```

## 8.3 Dimensões

A função `dim` retorna as dimensões de um **data frame** (linhas e colunas). Estes dados também podem ser obtidos com as funções `nrow` e `ncol`.

```
dim(df_iris)
```

```
[1] 150  5
```

```
nrow(df_iris)
```

```
[1] 150
```

```
ncol(df_iris)
```

```
[1] 5
```

## 8.4 Acessando Dados

### 8.4.1 Índices

Como a estrutura de um **data frame** é organizada em linhas e colunas, podemos acessar os dados utilizando colchetes (`[ ]`): `base[linha, coluna]`. Podem ser usados intervalos de índices com o operador `:`.

```
# Acessar primeira linha e segunda coluna (Sepal.Width)
df_iris[1, 2]
```

```
[1] 3.5
```

```
# Acessar linhas 1 até 3 e a segunda coluna
df_iris[1:3, 2]
```

```
[1] 3.5 3.0 3.2
```

Apesar de ser possível, utilizar o índice faz com que a referência seja relativa, ou seja, a variável '1' pode mudar caso o **data frame** seja editado. Por exemplo, caso em algum momento anterior a variável **PETAL\_LENGTH** tenha sido excluída, uma nova variável assumirá o índice 1. Além disto, no momento da leitura do código por um usuário não fica claro qual variável está sendo acessada.

## 8.4.2 Usando Nomes das Colunas

Existem diversas outras formas para acessar dados de um **data frame**, inclusive utilizando o nome da coluna de forma explícita.

```
# Acessar primeira linha e segunda coluna (pelo nome)
df_iris[1:3, 'SEPAL_WIDTH']
```

```
[1] 3.5 3.0 3.2
```

Uma forma bastante comum é através da utilização do operador \$ para acessar a coluna pelo seu nome.

```
# Acessar primeira linha e segunda coluna
df_iris[1, ]$SEPAL_WIDTH
```

```
[1] 3.5
```

```
# Acessar linhas 1 até 3 e a segunda coluna
df_iris[1:3, ]$SEPAL_WIDTH
```

```
[1] 3.5 3.0 3.2
```

### Nome Abreviado

Assim como nas **listas**, variáveis de um **data frame** podem ser acessadas com o mínimo de caracteres que as identifiquem dentro do **data frame**. Por exemplo, `df_iris$SP` retornará a variável **SPECIES**.

## 8.5 Filtrando Dados

Digamos que se deseje acessar apenas dados que cumpram determinada condição. Para isto, na seleção das linhas do **data frame**, deve ser informada condição lógica na forma abaixo:

```
# Retorna valores de Sepal.Width onde Petal.Length for maior do que 6
x <- df_iris[df_iris$PETAL_LENGTH > 6, 'SEPAL_WIDTH']
y <- df_iris[df_iris$PETAL_LENGTH > 6.5, ]$SEPAL_WIDTH

x
```

```
[1] 3.0 2.9 3.6 3.8 2.6 2.8 2.8 3.8 3.0
```

```
y
```

```
[1] 3.0 3.8 2.6 2.8
```

```
# Função que compara os objetos  
identical(x, y)
```

```
[1] FALSE
```

O retorno é dado pelas linhas em que a variável **PETAL\_LENGTH** atende as condições declaradas. Este teste retorna um vetor de valores lógicos, e os valores TRUE são os que “permanecem”. Abaixo outro exemplo:

```
head(df_iris$PETAL_LENGTH) > 1.4
```

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

Aplicando este vetor de valores lógicos, o R entende que as posições correspondentes a TRUE devem ser mantidas. No exemplo abaixo, as posições (linhas) 4 e 6 atendem a condição especificada, portanto apenas estas serão selecionadas.

```
df_iris2 <- head(df_iris)  
filtro <- head(df_iris2$PETAL_LENGTH) > 1.4  
filtro
```

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

```
df_iris2[filtro, 'SEPAL_WIDTH']
```

```
[1] 3.1 3.9
```

Equivalente ao comando abaixo:

```
df_iris2[c(4, 6), 'SEPAL_WIDTH']
```

```
[1] 3.1 3.9
```

### 8.5.1 Classes de retorno

Os filtros em **data frames** usados com \$ ou [ ] (com apenas 1 variável) retornam vetores e não **data frames**. Desta forma se perde a classe e a estrutura tabular característica do **data frame** original.

```
class(df_iris[1:3, 1])
```

```
[1] "numeric"
```

```
class(df_iris[1:3, 'SEPAL_WIDTH'])
```

```
[1] "numeric"
```

Entretanto, sendo selecionadas mais de uma coluna, a classe retornada segue sendo **data.frame**.

```
class(df_iris[1:3, c("SEPAL_LENGTH", "SEPAL_WIDTH")])
```

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

```
class(df_iris[1:3, 1:2])
```

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

## 8.6 Função Subset

A função `subset` permite efetuar filtro em um **data frame** e muitas vezes oferece uma forma mais organizada visualmente, principalmente quando em filtros com muitas condições. Uma outra vantagem é que a função `subset` retorna faz a seleção em um `data.frame` e retorna um **data frame**, mesmo com a seleção de apenas 1 variável.

Esta função também permite seleção de colunas a serem mantidas. Note que a função `subset` não demanda que o **data frame** seja referenciado antes das variáveis e também aceita os nomes das variáveis sem aspas. Isto torna o código mais legível.

```
class(subset(df_iris, select = SEPAL_WIDTH))
```

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

```
df_mtcars <- mtcars
# mudar nomes para maiusculas
names(df_mtcars) <- toupper(names(df_mtcars))

subset(x = df_mtcars, # dados
       subset = MPG > 25, # filtro
       select = c(MPG, CYL, HP)) # colunas
```

	MPG	CYL	HP
Fiat 128	32.4	4	66
Honda Civic	30.4	4	52
Toyota Corolla	33.9	4	65
Fiat X1-9	27.3	4	66
Porsche 914-2	26.0	4	91
Lotus Europa	30.4	4	113

Usando um filtro um pouco mais complexo e sem inserir o nome dos argumentos da função (x, subset e select):

```
df_mtcars_filtrado <- subset(df_mtcars, # dados
                             MPG > 25 & CYL == 4 & HP > 70, # filtro
                             c(MPG, CYL, HP)) # colunas
```

```
df_mtcars_filtrado
```

	MPG	CYL	HP
Porsche 914-2	26.0	4	91
Lotus Europa	30.4	4	113

Nos exemplos anteriores foram declaradas de forma explícita as variáveis a serem mantidas. Para declarar as variáveis a serem excluídas basta utiliza o sinal de subtração -, de forma análoga a seleção por índices em componentes de vetores.

```
df_mtcars |>
  subset(select = -c(DISP, DRAT, VS, AM)) |>
  head()
```

	MPG	CYL	HP	WT	QSEC	GEAR	CARB
Mazda RX4	21.0	6	110	2.620	16.46	4	4
Mazda RX4 Wag	21.0	6	110	2.875	17.02	4	4
Datsun 710	22.8	4	93	2.320	18.61	4	1
Hornet 4 Drive	21.4	6	110	3.215	19.44	3	1
Hornet Sportabout	18.7	8	175	3.440	17.02	3	2
Valiant	18.1	6	105	3.460	20.22	3	1

## 8.7 Junção de Dados

Uma grande necessidade ao se trabalhar com dados tabulados é a junção de dados. A junção nada mais é do que usar bases de dados diferentes e carregar dados entre elas a partir de uma chave de identificação. Vamos usar duas bases de dados, uma com código e nome do município e outra com o código do município e sua população. Estes dados foram buscados em IBGE (s.d.).

```
df_cidades <-
  data.frame(
    COD_MUNICIPIO = c('4314902', '3550308', '3304557'),
    NOME = c('Porto Alegre', 'São Paulo', 'Rio de Janeiro')
  )

df_populacao <-
  data.frame(
    COD_MUNICIPIO = c('4314902', '3550308', '3304557'),
    POPULACAO = c(1332570, 11451245, 6211423)
  )

head(df_cidades)
```

```
  COD_MUNICIPIO      NOME
1      4314902 Porto Alegre
2      3550308   São Paulo
3      3304557 Rio de Janeiro
```

```
head(df_populacao)
```

```
  COD_MUNICIPIO POPULACAO
1      4314902   1332570
2      3550308  11451245
3      3304557   6211423
```



Para juntar estes dados, usaremos como chave de identificação presente nas duas tabelas o campo **COD\_MUNICIPIO**. A função usada, `merge` exige dois argumentos **x** e **y**, que são as bases de dados que usaremos para a junção.

```
df_completo <- merge(x = df_cidades, y = df_populacao,
                    by = "COD_MUNICIPIO")

head(df_completo)
```

	COD_MUNICIPIO	NOME	POPULACAO
1	3304557	Rio de Janeiro	6211423
2	3550308	São Paulo	11451245
3	4314902	Porto Alegre	1332570

Este exemplo é o mais básico, onde os dados presentes em ambas tabelas são das mesmas cidades e também são ligadas por apenas uma chave de identificação. Vejamos um exemplo um pouco mais realista, onde alguns dados não estão presentes em ambas tabelas.

```
# rbind faz a inclusao de linha nas as bases criadas
df_cidades <- rbind(df_cidades, c('3106200', 'Belo Horizonte'))
df_populacao <- rbind(df_populacao, c('4106902', 1773733))

df_completo <- merge(x = df_cidades, y = df_populacao,
                    by = "COD_MUNICIPIO")

head(df_completo)
```

	COD_MUNICIPIO	NOME	POPULACAO
1	3304557	Rio de Janeiro	6211423
2	3550308	São Paulo	11451245
3	4314902	Porto Alegre	1332570

Veja que os dados de Belo Horizonte e do Município de código 4106902 (Curitiba) não foram inseridos no data frame resultante. Por padrão a função `merge` faz a junção pelos dados presentes nos dois data frames. Caso desejemos especificar, usamos os parâmetros `all.x` e `all.y`.

Usando `all.x` informamos ao R que desejamos que todas as linhas presentes na base passada como argumento `x` sejam mantidas. Onde não existirem dados para estas linhas na tabela `y` serão preenchidos com `NA`.

```
df_completo_x <- merge(x = df_cidades, y = df_populacao,  
                      by = "COD_MUNICIPIO", all.x = T)  
  
head(df_completo_x)
```

	COD_MUNICIPIO	NOME	POPULACAO
1	3106200	Belo Horizonte	<NA>
2	3304557	Rio de Janeiro	6211423
3	3550308	São Paulo	11451245
4	4314902	Porto Alegre	1332570

De forma análoga, usar `all.y` informa para que as linhas da base y sejam mantidas.

```
df_completo_y <- merge(x = df_cidades, y = df_populacao,  
                      by = "COD_MUNICIPIO", all.y = T)  
  
head(df_completo_y)
```

	COD_MUNICIPIO	NOME	POPULACAO
1	3304557	Rio de Janeiro	6211423
2	3550308	São Paulo	11451245
3	4106902	<NA>	1773733
4	4314902	Porto Alegre	1332570

Para cruzamento de todas as linhas das duas tabelas usamos o argumento `all`.

```
df_completo <- merge(x = df_cidades, y = df_populacao,  
                    by = "COD_MUNICIPIO", all = T)  
  
head(df_completo)
```

	COD_MUNICIPIO	NOME	POPULACAO
1	3106200	Belo Horizonte	<NA>
2	3304557	Rio de Janeiro	6211423
3	3550308	São Paulo	11451245
4	4106902	<NA>	1773733
5	4314902	Porto Alegre	1332570

---

Última atualização: 11/10/2024 - 21:53:29

# 9 Operador Pipe

Status □□□

## 9.1 Introdução

Muitas vezes seu código demanda muitas transformações e acaba ficando muito verboso e de difícil entendimento. Uma forma de facilitar a compreensão em torno das operações em sequência é criar um fluxo em que as operações vão sendo efetuadas em sequência, onde as entradas são as saídas do passo anterior.

O operador `|>` (*pipe*) existe com este intuito, organizar as operações em um fluxo contínuo. O *pipe* foi implementado a partir da versão 4.1.0 do R e passa um valor para uma função. Os dados são passados do lado esquerdo (**lhs** - *left hand side*) para o lado direito (**rhs** - *right hand side*). O valor do lado esquerdo (lhs) é passado como o primeiro argumento da função do lado direito (rhs).

Vejamos um exemplo simplificado onde o vetor que possui números de 1 até 20 é passado para a função `head`. Com o uso do `|>` o vetor é passado como **primeiro argumento** da função `head` e esta por sua vez exibe os seis primeiros elementos.

```
c(1:20) |> head()
```

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

O código acima é equivalente a:

```
head(c(1:20))
```

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

Caso se deseje alterar o número de elementos, basta usar o argumento **n**.

```
c(1:20) |> head(n = 10)
```

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

Equivalente a:

```
head(c(1:20), n = 10)
```

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

Vejamos um outro exemplo, um pouco mais realista e complexo: usar a base **mtcars** e a partir desta selecionar casos em que o campo **mpg** seja maior do que 10 e após criar uma variável chamada **media\_hp**, que será a média a partir do campo **hp**. Poderia ser feito algo do tipo:

```
df_mtcars <- subset(mtcars, mpg > 10)
media_hp <- mean(df_mtcars$hp)
media_hp
```

```
[1] 146.6875
```

Mesmo sendo um processo pequeno com apenas 2 operações bastante corriqueiras, ler o código já se torna enfadonho, para dizer o mínimo. Também não fica claro, em uma passada de olhos, se as operações possuem relação entre si.

Imagine agora criar as mesmas operações de forma “concatenada” em que uma transformação é passada para a seguinte até que se chegue ao final do fluxo. Em linguagem “humana” algo do tipo:

data frame  filtrar casos  selecionar variável  calcular média

Em R:

```
mtcars |>
  subset(mpg > 10) |>
  subset(select = hp, drop = T) |>
  mean()
```

```
[1] 146.6875
```

```
# ou de forma mais sucinta
mtcars |>
  subset(mpg > 10, select = hp, drop = T) |>
  mean()
```

```
[1] 146.6875
```

Este código é equivalente ao anterior, porém aqui fica mais claro que todas as transformações foram feitas a fim de obter o valor da média de **hp** dos casos desejados (**mpg > 10**). Para fazer a atribuição do resultado em uma variável basta, como de costume, ao início ou ao final usar o operador de atribuição `<-`.

```
media_hp <- mtcars |>
  subset(mpg > 10) |>
  subset(select = hp, drop = T) |>
  mean()
```

```
media_hp
```

```
[1] 146.6875
```

```
# ou de forma menos usual
mtcars |>
  subset(mpg > 10) |>
  subset(select = hp, drop = T) |>
  mean() -> media_hp
```

```
media_hp
```

```
[1] 146.6875
```

## 9.2 Placeholder

A partir da versão **4.2.0** o *pipe* passou a ter um **placeholder** (símbolo `_`) que serve para que o valor **lhs** seja passado para outro argumento que não o primeiro da função **rhs**.

```
8 |> head(c(1:20), n = _)
```

```
[1] 1 2 3 4 5 6 7 8
```

Equivalente a:

```
head(c(1:20), n = 8)
```

```
[1] 1 2 3 4 5 6 7 8
```

A partir da versão **4.3.0** o *placeholder* também pode ser utilizado para operações de extrações com [. Replicando o exemplo do cálculo de **media\_hp**, porém agora fazendo a extração da variável **hp** que é retornada como um vetor e passada para a função `mean`.

```
media_hp <- mtcars |>  
  subset(mpg > 10) |>  
  _$hp |>  
  mean()
```

```
media_hp
```

```
[1] 146.6875
```

---

R Core Team (2023d)

Wickham (2023/04/21)

Última atualização: 11/10/2024 - 21:53:36

# 10 Funções

Status

## 10.1 Criando Funções

Funções podem ser criadas através do comando `function`.

```
fnSomar <- function(param1, param2) {  
  param1 + param2  
}
```

```
fnSomar(5, 8)
```

```
[1] 13
```

Para visualizar o código de uma função podemos usar seu nome sem os parênteses.

```
fnSomar
```

```
function(param1, param2) {  
  param1 + param2  
}
```

### 10.1.1 Argumentos - Valores Padrão

## 10.2 Função x Ambiente

As funções possuem seus próprios ambientes. Abaixo uma função criada para exibir seu ambiente e seu ambiente 'pai'.

```
fnExibirEnvs <- function() {
  print('Ambiente atual:')
  print(environment())

  print(paste(
    'Ambiente Pai:',
    environmentName(parent.env(environment()
      )))
})

fnExibirEnvs()
```

```
[1] "Ambiente atual:"
<environment: 0x0000017e18c2f628>
[1] "Ambiente Pai: R_GlobalEnv"
```

### 10.2.1 Objetos no Ambiente da Função

Objetos que são criados dentro de uma função existem apenas dentro do ambiente desta função. Abaixo um exemplo de variável criada dentro do ambiente da função e que não é acessível no **GlobalEnv**.

```
fnTeste <- function(){
  y <- 15
  x <- 80
  ls()
}

fnTeste()
```

```
[1] "x" "y"
```

```
y
```

```
Error in eval(expr, envir, enclos): objeto 'y' não encontrado
```

Objetos que existam no ambiente corrente não são alterados caso por estarem dentro do ambiente de uma função. A variável **x** é inicializada com valor 10 no ambiente corrente. Ela pode ser acessada pela função mesmo não sendo informada em algum argumento.



```
x <- 10

fnTeste2 <- function(){
  y <- 15
  x + y
}

fnTeste2()
```

[1] 25

```
y
```

Error in eval(expr, envir, enclos): objeto 'y' não encontrado

```
x
```

[1] 10

Entretanto, caso a variável **x** seja alterada no ambiente da função ela não é alterada no ambiente corrente.

```
x <- 10

fnTeste3 <- function(){
  y <- 15
  x <- 80
  x + y
}

fnTeste3()
```

[1] 95

```
x
```

[1] 10

### 10.2.1.1 Operador de Super Atribuição (<<-)

Usando o operador de super atribuição <<- é possível alterar objetos que estejam fora do ambiente de uma função. Neste caso a variável **x** é atualizada no ambiente que está acima do ambiente da função. A variável **y** continua não existindo fora da função, porém agora a variável **x** é atualizada em ambos ambientes.

```
ls(envir = globalenv())
```

```
[1] "fnExibirEnvs" "fnSomar"      "fnTeste"      "fnTeste2"     "fnTeste3"
[6] "repo"         "x"
```

```
x
```

```
[1] 10
```

```
fnTeste4 <- function(){
  y <- 15
  x <<- 80
  x + y
}
```

```
fnTeste4()
```

```
[1] 95
```

```
y
```

```
Error in eval(expr, envir, enclos): objeto 'y' não encontrado
```

```
x
```

```
[1] 80
```

Apesar de, neste caso, produzirem o mesmo retorno, as funções `fnTeste3` e `fnTeste4` impactam de formas distintas o ambiente do R.

## 10.3 Retorno

Na criação de funções, é possível utilizar o comando `return` a fim de definir o que será retornado pela função.

```
fnRetorno <- function(){  
  return('Este é o retorno da função!')  
}  
fnRetorno()
```

```
[1] "Este é o retorno da função!"
```

## 10.4 Recursividade

Como outras linguagens de programação, R permite o uso recursivo de funções.

```
fnRecursividade <- function(x){  
  if(x > 100) return('X ultrapassou 100. Fim!')  
  x <- x + 1  
  print(paste('Valor atual de x:', x))  
  fnRecursividade(x)  
}  
fnRecursividade(95)
```

```
[1] "Valor atual de x: 96"  
[1] "Valor atual de x: 97"  
[1] "Valor atual de x: 98"  
[1] "Valor atual de x: 99"  
[1] "Valor atual de x: 100"  
[1] "Valor atual de x: 101"
```

```
[1] "X ultrapassou 100. Fim!"
```

### 10.4.1 Buscar Ambiente Pai (Recursivamente)

Abaixo função que busca recursivamente os ambientes e seus 'pais' até que se chegue no 'último' ambiente, o `R_EmptyEnv`.

```
fnBuscarEnvsPai <- function(ambiente, nivel = 1){  
  
  if(environmentName(ambiente)=="R_EmptyEnv"){  
    return ('Ambiente informado é R_EmptyEnv. Fim da busca.')  }  
  
  marcacao <- ''  
  for (i in 1:nivel){  
  
    marcacao <- paste0(' ', marcacao)  
  }  
  
  writeLines(paste0(marcacao, '|-- ', environmentName(parent.env(ambiente))))  
  
  nivel <- nivel + 1  
  fnBuscarEnvsPai(parent.env(ambiente), nivel = nivel)  
  
}  
fnBuscarEnvsPai(globalenv())
```

```
|-- package:stats  
|-- package:graphics  
|-- package:grDevices  
|-- package:utils  
|-- package:datasets  
|-- package:methods  
|-- Autoloads  
|-- base  
|-- R_EmptyEnv
```

```
[1] "Ambiente informado é R_EmptyEnv. Fim da busca."
```

## 10.5 Funções Genéricas

Funções genéricas são funções que contém métodos associados. Os métodos são chamados de acordo com a classe do objeto informado.

Por exemplo a função `print` possui diversos métodos associados. Quando um **data.frame** é informado como argumento o R chama a função `print.data.frame`, ao passo que se o argumento da função for uma data o método invocado é `print.Date`.

```
mtcars[1:2, 1:5] |> print.data.frame()
```

```
      mpg cyl disp  hp drat
Mazda RX4      21   6  160 110  3.9
Mazda RX4 Wag  21   6  160 110  3.9
```

```
Sys.Date() |> print.Date()
```

```
[1] "2024-10-11"
```

```
# o mesmo que
mtcars[1:2, 1:5] |> print()
```

```
      mpg cyl disp  hp drat
Mazda RX4      21   6  160 110  3.9
Mazda RX4 Wag  21   6  160 110  3.9
```

```
Sys.Date() |> print()
```

```
[1] "2024-10-11"
```

## 10.6 Operadores Unários

A operador de soma `+` é na verdade uma função e é em geral utilizado na forma de operador unário, ou seja, recebe os valores da direita e da esquerda como seus argumentos. Veja o código que é reotrnado quando a função é chamada sem informação de argumentos e parênteses:

```
`+`
```

```
function (e1, e2) .Primitive("+")
```

Você poderia usar a função de somar de outra forma com a função `do.call` ou mesmo chamando esta com uso de crases. Obviamente não é a forma mais sucinta, mas serve para demonstrar que `+` não passa de uma função como as demais.

```
do.call(`+`, list(1,3))
```

```
[1] 4
```

```
`+`(8,7)
```

```
[1] 15
```

O mais interessante é que você também pode criar funções para serem usadas na forma de operador unário. Para isto basta criá-la com `%` ao início e ao fim do nome da função. Abaixo as duas formas de uso desta função:

```
`%fnSomar2%` <- function(x, y){  
  x + y  
}
```

```
10 %fnSomar2% 15
```

```
[1] 25
```

```
`%fnSomar2%`(2,5)
```

```
[1] 7
```

---

Grolemund (2014)

Última atualização: 11/10/2024 - 21:50:57

# 11 Dados Externos

Status

## 11.1 Formato Csv

### 11.1.1 Importar Arquivos csv

Muitas vezes os dados que o usuário possui acesso em sua Instituição estão armazenados em bancos de dados. Porém dados distribuídos por entidades públicas muitas vezes estão em formato **csv**.

Neste exemplo vamos importar a base **Estatísticas de Aprovações - Por Porte de Empresa** do BNDES.

```
df_aprovacoes_porte <-  
  read.csv(  
    './data/aprovacoes_por_porte_de_empresa.csv',  
    header = T,  
    sep = ';',  
    dec = ',',  
    quote = "\"" )
```

```
df_aprovacoes_porte |> head()
```

	ano	mes	micro	pequena	media	grande
1	1995	1	128.4699	0	10.18922	380.2330
2	1995	2	106.3283	0	16.21161	495.5282
3	1995	3	234.5488	0	13.69085	715.9591
4	1995	4	125.2196	0	16.44511	403.9919
5	1995	5	209.4168	0	20.88794	477.2529
6	1995	6	122.5179	0	23.86818	473.2194

### 11.1.2 Exportar Arquivos csv

Para salvar um arquivo em csv deve ser informado como parâmetro da função `write.csv` o nome do objeto e o arquivo no qual ele será salvo.

```
df_aprovacoes_porte |>
  write.csv(paste0(tempdir(), '/df_aprovacoes_porte.csv'))
```

## 11.2 Formato RDS

O formato RDS é específico do R e possui diversas vantagens em relação ao uso do formato csv, entre elas:

- Permite compactação
- Permite salvar objetos de diversos tipo (bases de dados, vetores, listas, funções, etc)
- Mantém a formatação dos dados

### 11.2.1 Importar Arquivos RDS

```
df_aprovacoes_porte <- readRDS('data/aprovacoes_por_porte_de_empresa.RDS')
```

### 11.2.2 Exportar Arquivos RDS

```
df_aprovacoes_porte |>
  saveRDS(paste0(tempdir(), '/df_aprovacoes_porte.RDS'))
```

---

Última atualização: NA



# 12 Controles de Fluxo

Status

## 12.1 Introdução

Assim como outras linguagens de programação R oferece uma série de operadores para controle de fluxo de código.

### **i** Nota

Controles de fluxo são declarações usadas na linguagem, mas **não são funções**.

## 12.2 If

O controle `if` é a estrutura de controle mais básica que tomada de decisão e “direcionamento” de código. Em caso negativo do teste lógico nenhuma operação é executada.

```
x <- 5

# Códigos equivalentes
if(x > 4) print('x é maior do que quatro')
```

```
[1] "x é maior do que quatro"
```

```
if(x > 4) { print('x é maior do que quatro')}
```

```
[1] "x é maior do que quatro"
```

```
if(x > 4) {
  print('x é maior do que quatro') }
```

```
[1] "x é maior do que quatro"
```

```
if(x > 4) { print('x é maior do que quatro')
}
```

```
[1] "x é maior do que quatro"
```

```
if(x > 4) {
  print('x é maior do que quatro')
} # o mais organizado
```

```
[1] "x é maior do que quatro"
```

Note que se o teste não retornar TRUE ou FALSE o R reportará erro.

```
x <- NA
if (x > 4) print('x é maior do que quatro')
```

Error in if (x > 4) print("x é maior do que quatro"): valor ausente onde TRUE/FALSE necessário

## 12.3 Ifelse

R possui a **função** `ifelse`, que apesar de não ser para controle de fluxo, possui lógica de uso muito semelhante ao `if` e por este motivo será tratada neste capítulo. Esta função efetua teste em valor de entrada e define um valor a ser retornado caso verdadeiro e outro caso falso.

O retorno de `ifelse` possui o mesmo formato da estrutura informada no argumento **test**. Esta função pode ser usada para atribuição em data frames de forma mais sucinta.

Vejam os um exemplo:

```
df_mtcars6 <-
  mtcars |>
  subset(select = c('hp', 'mpg', 'cyl')) |>
  head()

df_mtcars6
```

	hp	mpg	cyl
Mazda RX4	110	21.0	6
Mazda RX4 Wag	110	21.0	6
Datsun 710	93	22.8	4
Hornet 4 Drive	110	21.4	6
Hornet Sportabout	175	18.7	8
Valiant	105	18.1	6

```
df_mtcars6[df_mtcars6$hp > 100, 'RESULTADO'] <-
  df_mtcars6[df_mtcars6$hp > 100, ]$mpg

df_mtcars6[df_mtcars6$hp <= 100, 'RESULTADO'] <-
  df_mtcars6[df_mtcars6$hp <= 100,]$cyl

df_mtcars6
```

	hp	mpg	cyl	RESULTADO
Mazda RX4	110	21.0	6	21.0
Mazda RX4 Wag	110	21.0	6	21.0
Datsun 710	93	22.8	4	4.0
Hornet 4 Drive	110	21.4	6	21.4
Hornet Sportabout	175	18.7	8	18.7
Valiant	105	18.1	6	18.1

```
# com ifelse
df_mtcars6$RESULTADO2 <-
  ifelse(df_mtcars6$hp > 100,
        df_mtcars6$mpg,
        df_mtcars6$cyl)

df_mtcars6
```

	hp	mpg	cyl	RESULTADO	RESULTADO2
Mazda RX4	110	21.0	6	21.0	21.0
Mazda RX4 Wag	110	21.0	6	21.0	21.0
Datsun 710	93	22.8	4	4.0	4.0
Hornet 4 Drive	110	21.4	6	21.4	21.4
Hornet Sportabout	175	18.7	8	18.7	18.7
Valiant	105	18.1	6	18.1	18.1

## 12.4 If Else

O `if else` pode ser usado para inserir uma ação após o retorno negativo do teste feito pelo `if`.

```
x <- 3
if(x > 4) {
  print('x é maior do que quatro')
} else {
  print('x não é maior do que quatro')
}
```

```
[1] "x não é maior do que quatro"
```

Veja que podem ser usadas muitas declaração `else` em sequência.

```
x <- 3
if(x > 3) {
  print('x é maior do que três')
} else if (x < 3){
  print('x é menor do que três')
} else if (x == 3){
  print('x é igual a três')
}
```

```
[1] "x é igual a três"
```

## 12.5 Laço For

Um laço `for` é uma estrutura que efetua uma determinada quantidade de passos de acordo com a sequência informada. a declaração deve ser feita no formato: `for(x in seq)`, sendo `x` a variável que será atualizada a cada iteração iniciando no primeiro valor informado em `seq` e encerrando no último. Um exemplo:

```
for(x in 1:5){
  print(paste('Iteração:', x))
}
```

```
[1] "Iteração: 1"  
[1] "Iteração: 2"  
[1] "Iteração: 3"  
[1] "Iteração: 4"  
[1] "Iteração: 5"
```

Caso se deseje mudar o incremento a cada passo pode ser usada a função `seq`. Também é possível usar um passo decrescente.

```
for(x in seq(2, 1, -0.25)) {  
  print(paste('Valor de x:', x))  
}
```

```
[1] "Valor de x: 2"  
[1] "Valor de x: 1.75"  
[1] "Valor de x: 1.5"  
[1] "Valor de x: 1.25"  
[1] "Valor de x: 1"
```

No exemplo acima, `x` é inicializada com valor 1 e vai sendo incrementada em 1 unidade ao início do próximo passo. Um laço `for` também pode fazer iterações sobre vetores com texto, por exemplo.

```
for(i in c('São Paulo', 'Rio de Janeiro', 'Porto Alegre')){  
  print(paste('Cidade atual:', i))  
}
```

```
[1] "Cidade atual: São Paulo"  
[1] "Cidade atual: Rio de Janeiro"  
[1] "Cidade atual: Porto Alegre"
```

No laço `for` a sequência no qual será feita a iteração é considerada antes de se iniciar o laço, assim mesmo se houver alguma alteração nesta sequência em um dos passos esta alteração não impactará na execução.

```
x <- 3  
for(i in 1:x) {  
  x <- x + 2  
  print(x)  
}
```

```
[1] 5
[1] 7
[1] 9
```

## 12.6 While

### For x While

Um laço `for` é utilizado quando se tem uma sequência definida de passos. Caso se deseje executar alguma operação até o atendimento de uma condição, use `while`.

A estrutura `while` possui a seguinte configuração: `while(condição)`. Assim, a repetição do código dentro de um bloco `while` ocorre até que a condição não seja mais satisfeita. Exemplo:

```
condicao <- 5
while(condicao < 7){
  print(condicao)
  condicao <- condicao + 1
}
```

```
[1] 5
[1] 6
```

A variável **condicao** é iniciada com valor 5 e atende a condição **se < 7**. O código então é executado, imprimindo no console o valor da variável e após esta recebe seu próprio valor mais 1. Na iteração seguinte seu valor é 6 e ainda atende a condição de **se < 7**. Na terceira iteração seu valor, novamente atualizado, será 7 e não cumprindo a condição o `while` é encerrado.

A estrutura `while` pode ficar operando indefinidamente se por algum motivo a condição seja sempre atendida. Teste o código abaixo e veja que ele rodará indefinidamente. Você pode pará-lo teclando **ESC** no teclado.

```
while(TRUE){
  print(condicao)
  condicao <- condicao + 1
}
```

## 12.7 Repeat

E estrutura `repeat` funciona de forma análoga ao `while`, entretanto esta não testa condição de parada. Para efetuar a parada o usuário deve fazer de forma explícita.

```
x <- 1
y <- 2

repeat{
  x <- x + y
  print(x)
  if (x > 11) break
}
```

```
[1] 3
[1] 5
[1] 7
[1] 9
[1] 11
[1] 13
```

## 12.8 Break e Next

O `break` encerra as estruturas `for`, `while` e `repeat`. `Next`, por sua vez, interrompe a execução da iteração atual e inicia a próxima.

Abaixo um exemplo com uso de `break` em um laço `for`. Note que apenas a primeira iteração é executada.

```
for (i in 1:3){
  print(i)
  break
}
```

```
[1] 1
```

Abaixo um exemplo de uso do `next`. Repare que o comando `print('Teste')` nunca é executado, pois o `next` interrompe a execução da iteração atual, assim tudo que for colocado após esta linha (dentro do laço) não é executado.

```
for (i in 1:3){  
  print(i)  
  next  
  print('Teste')  
}
```

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

Em caso de laços embutidos, `break` e `next` impactam apenas o mais “interno”.

```
for (i in 1:3){  
  for (j in 1:3){  
    break  
    print(paste('Valor de j é:', j))  
  }  
  print(paste('Valor de i é:', i))  
}
```

```
[1] "Valor de i é: 1"  
[1] "Valor de i é: 2"  
[1] "Valor de i é: 3"
```

---

DataMentor (s.d.)

R Core Team (2023c)

Última atualização: 11/10/2024 - 21:49:34



# 13 Gráficos

Status □□□

## 13.1 Introdução

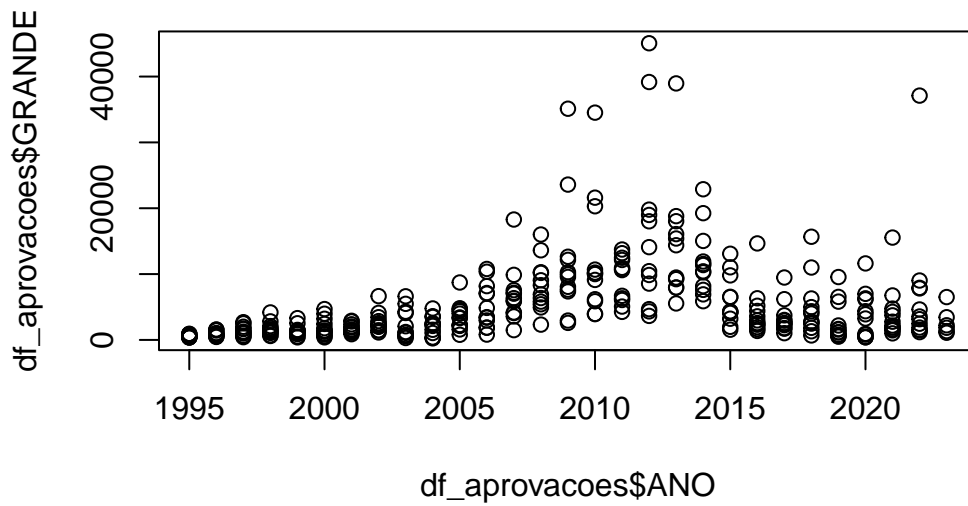
R oferece uma série de funções nativas para criação de gráficos. Estas funções possuem muitos parâmetros que permitem melhorar visualmente as apresentações dos gráficos.

## 13.2 Função Plot

A função `plot`, do pacote **graphics**, é uma função que gera gráficos de dispersão e oferece uma gama de opções para customização. Mas vejamos primeiro um exemplo mínimo.

Os argumentos `x` e `y` são usados nas coordenadas e já são suficientes para a geração do. Por padrão o gráfico gerado é de pontos (argumento `type = "p"`).

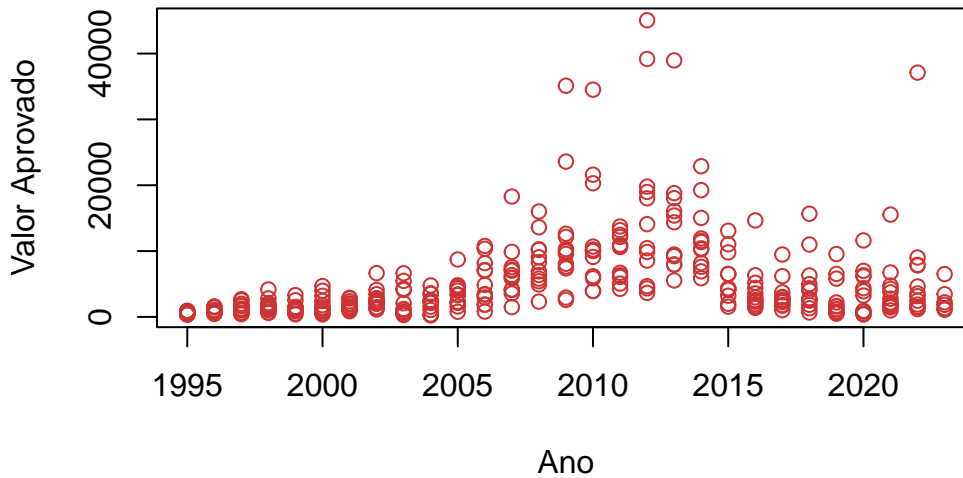
```
df_aprovacoes <- readRDS('data/aprovacoes_por_porte_de_empresa.RDS')  
  
plot(x = df_aprovacoes$ANO, y = df_aprovacoes$GRANDE)
```



Podemos incluir diversos elementos no gráfico, por exemplo, `main` é o argumento que define o título, enquanto que `xlab` e `ylab` são os *labels* dos eixos e `col` define a cor. Note que o R possui muitas cores que podem ser identificadas como *strings*. Você pode ver as disponíveis com a função `colors`.

```
plot(df_aprovacoes$ANO, df_aprovacoes$GRANDE,
     col = 'brown3', main = 'Gráfico - Aprovações - Porte: Grande',
     xlab = 'Ano', ylab = 'Valor Aprovado')
```

## Gráfico – Aprovações – Porte: Grande



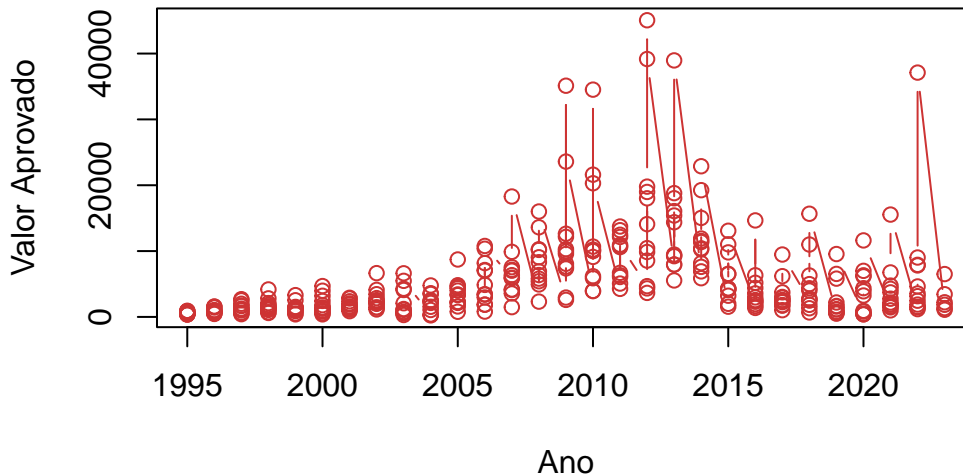
Veja que o gráfico foi exibido com pontos para os valores. O parâmetro `type` permite outras opções:

- `p` (padrão): pontos
- `l`: linhas
- `b`: pontos e linhas
- `c`: pontos 'vazios' ligados por linhas

Obviamente, nem todos os tipos de gráficos se enquadram para todos os tipos de dados. Se usarmos um gráfico de pontos e linhas nos dados do exemplo acima, as linhas não farão muito sentido, pois elas farão a ligação entre os pontos.

```
plot(df_aprovacoes$ANO, df_aprovacoes$GRANDE,  
     type = 'b',  
     col = 'brown3',  
     main = 'Gráfico - Aprovações - Porte: Grande',  
     xlab = 'Ano', ylab = 'Valor Aprovado')
```

## Gráfico – Aprovações – Porte: Grande



Vamos criar gráfico de linhas com a base de dados da Taxa Selic Acumulada no Mês divulgada pelo Banco Central do Brasil. Vamos utilizar os parâmetros `lty` (de *line type*) para determinar o tipo de linha e `lwd` (de *line width*) para determinar a largura da linha.

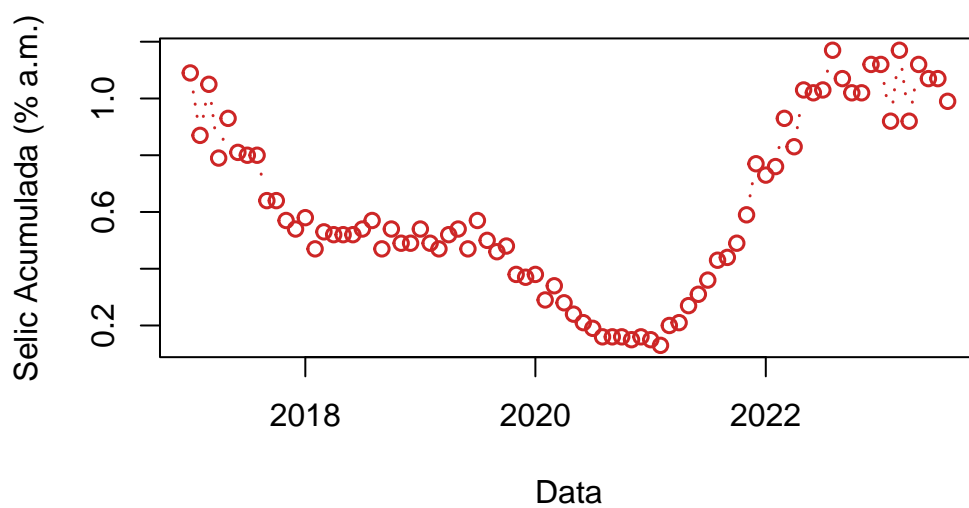
```
df_taxa_selic <- read.csv2('./data/csv_serie_sgs_4390.csv',
                           dec = ',', quote = '\"',
                           col.names = c('DATA', 'SELIC'))

# formatar data
df_taxa_selic$DATA <- as.Date(df_taxa_selic$DATA, format = '%d/%m/%Y')

# filtrar a partir de 2017
df_taxa_selic <- df_taxa_selic |>
  subset(DATA >= '2017-01-01')

plot(df_taxa_selic$DATA, df_taxa_selic$SELIC,
      type = 'b', col = 'firebrick3', lty = 3, lwd = 1.5,
      main = 'Taxa Selic Acumulada Mensal',
      xlab = 'Data', ylab = 'Selic Acumulada (% a.m.)')
```

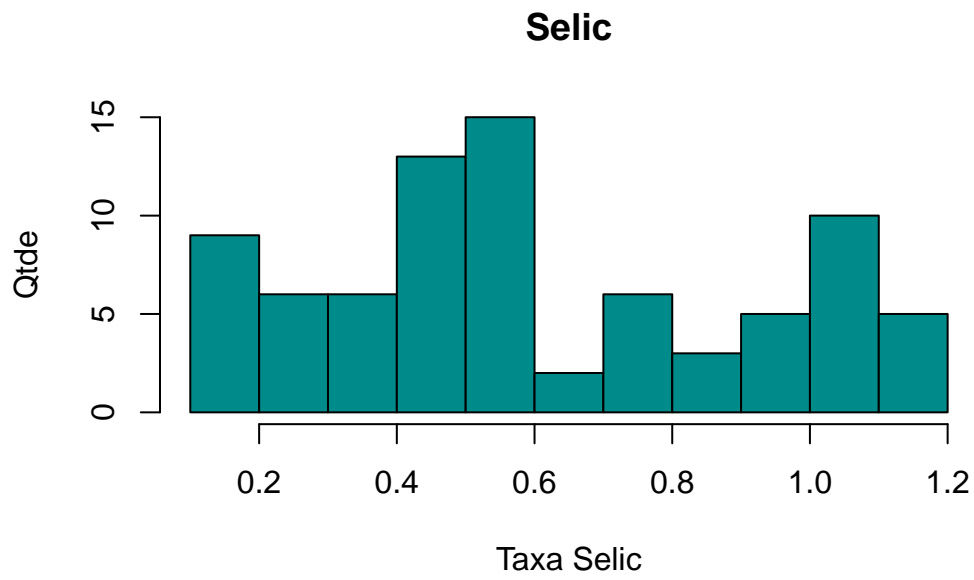
## Taxa Selic Acumulada Mensal



### 13.3 Função Hist

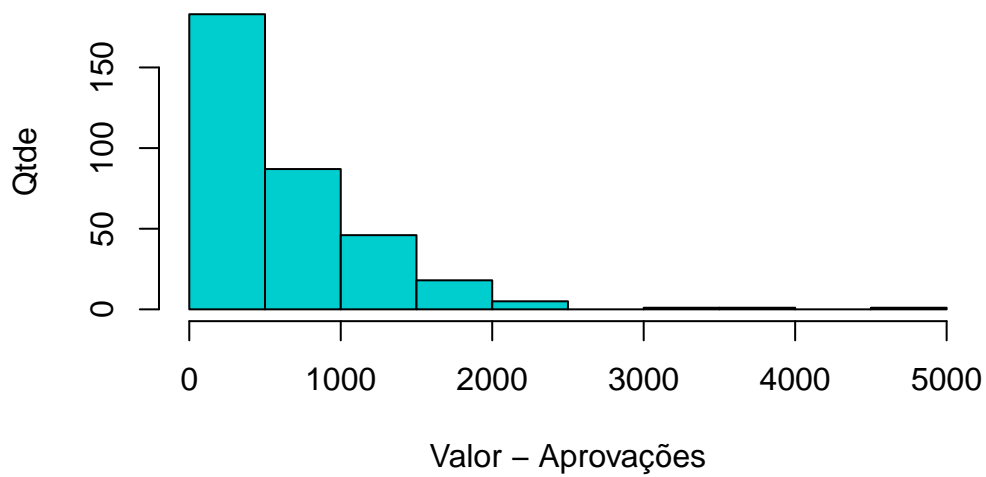
Abaixo um exemplo de um histograma:

```
hist(df_taxa_selic$SELIC, col = 'cyan4',  
     main = 'Selic',  
     xlab = 'Taxa Selic',  
     ylab = 'Qtde')
```



```
hist(df_aprovacoes$PEQUENA, col = 'cyan3',  
     main = 'Aprovações - Porte: Pequena',  
     xlab = 'Valor - Aprovações',  
     ylab = 'Qtde')
```

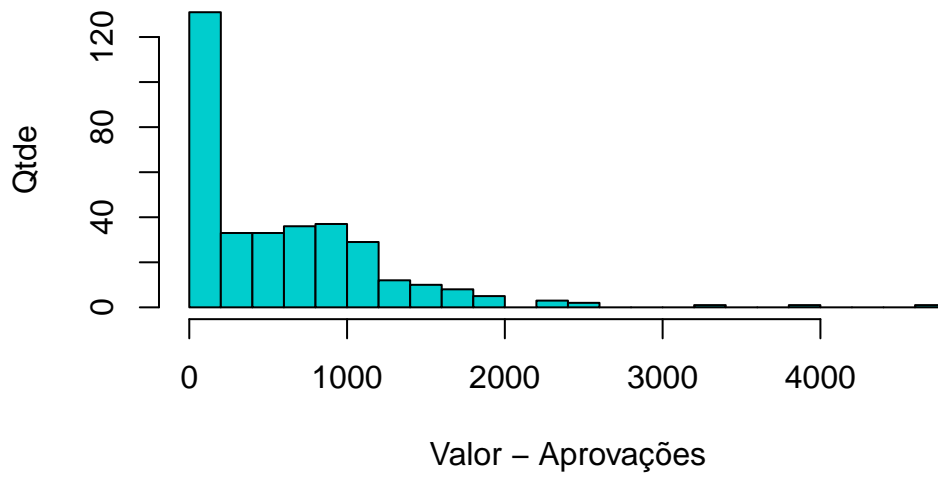
## Aprovações – Porte: Pequena



Com o argumento *breaks* podemos controlar quantas divisões serão exibidas no histograma.

```
hist(df_aprovacoes$PEQUENA, col = 'cyan3',  
     main = 'Aprovações - Porte: Pequena',  
     xlab = 'Valor - Aprovações',  
     ylab = 'Qtde',  
     breaks = 20)
```

## Aprovações – Porte: Pequena



Kassambara (s.d.)

Schmuller (2019)

Última atualização: 11/10/2024 - 21:51:03



# 14 Sumarização de Dados

Status □□□

## 14.1 Funções Básicas

Nesta seção são detalhadas algumas das funções mais básicas para sumarização de dados.

### 14.1.1 Soma

A função `sum` retorna a soma dos elementos informados como argumentos.

```
sum(1:10)
```

```
[1] 55
```

```
sum(NA)
```

```
[1] NA
```

```
sum(NA, na.rm = T)
```

```
[1] 0
```

### 14.1.2 Média

```
mean(1:20)
```

```
[1] 10.5
```

```
mean(c(NA, 1, 2, 3))
```

```
[1] NA
```

```
mean(NA, na.rm = T)
```

```
[1] NaN
```

### 14.1.3 Mediana

```
median(1:11)
```

```
[1] 6
```

```
median(c(NA, 1, 2, 3))
```

```
[1] NA
```

```
median(NA, na.rm = T)
```

```
[1] NA
```

### 14.1.4 Máximo e mínimo

```
max(99:15)
```

```
[1] 99
```

```
min(15:99)
```

```
[1] 15
```

Estas funções também oferecem o argumento `na.rm`. Veja que vetores vazios recebem retornos `Inf` e `-Inf`.

```
max(c(NA, NA), na.rm = T)
```

Warning in max(c(NA, NA), na.rm = T): nenhum argumento não faltante para max;  
retornando -Inf

```
[1] -Inf
```

```
min(c(NA, NA), na.rm = T)
```

Warning in min(c(NA, NA), na.rm = T): nenhum argumento não faltante para min;  
retornando Inf

```
[1] Inf
```

## 14.2 Agregação

Uma tarefa muito rotineira em análise de dados é a sumarização de valores por grupos de interesse.

Vejam um exemplo na base Estatísticas de Aprovações - Por Porte de Empresa (BNDES). Uma informação de interesse, por exemplo, pode ser o valor total de Aprovações por ano para cada porte das empresas. Como a base de dados originalmente traz os valores abertos para cada mês, será necessária operação de agregação.

```
df_aprovacoes_porte <-  
  readRDS(  
    './data/aprovacoes_por_porte_de_empresa.RDS')  
  
df_aprovacoes_porte |> head()
```

	ANO	MES	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	1	128.4699	0	10.18922	380.2330
2	1995	2	106.3283	0	16.21161	495.5282
3	1995	3	234.5488	0	13.69085	715.9591
4	1995	4	125.2196	0	16.44511	403.9919
5	1995	5	209.4168	0	20.88794	477.2529
6	1995	6	122.5179	0	23.86818	473.2194

Podemos então fazer a agregação usando a variável **ANO** como variável chave na função `aggregate`. Esta função exige (dentre outros argumentos opcionais) um objeto sobre os quais a função informada será aplicada, uma lista de elementos para definir o agrupamento e a função a ser aplicada. No exemplo, usaremos a função `sum`, que retorna a soma dos valores.

```
somatorio <-  
  aggregate(subset(df_aprovacoes_porte, select = c(-ANO, -MES)),  
            by = list(df_aprovacoes_porte$ANO),  
            sum)  
  
head(somatorio)
```

	Group.1	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	1711.645	0.000000	216.8279	7048.91
2	1996	1431.916	0.000000	271.6768	11362.08
3	1997	2179.985	1.053206	137.0893	16672.74
4	1998	1375.066	34.909957	1454.6207	20161.79
5	1999	1394.739	426.779631	1083.7491	16634.18
6	2000	2366.893	864.823930	1534.4957	22858.39

Veja que a função `subset` foi usada pois as somas de **ANO** e **MES** não são de interesse aqui. Sem removê-las a função `aggregate` faria a soma de seus valores.

A mesma operação pode ser feita usando o *pipe*:

```
df_aprovacoes_porte |>  
  subset(select = c(-ANO, -MES)) |>  
  aggregate(by = list(df_aprovacoes_porte$ANO),  
            sum) |>  
  head()
```

	Group.1	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	1711.645	0.000000	216.8279	7048.91
2	1996	1431.916	0.000000	271.6768	11362.08
3	1997	2179.985	1.053206	137.0893	16672.74
4	1998	1375.066	34.909957	1454.6207	20161.79
5	1999	1394.739	426.779631	1083.7491	16634.18
6	2000	2366.893	864.823930	1534.4957	22858.39

Uma forma equivalente e ainda mais sucinta é possível com a utilização do `.` (indicando todas as variáveis) e do `~` (indicando que as variáveis “dependem” ou são “explicadas” pela

variável **ANO**). Como aqui a variável **ANO** será indicada como “explicativa” das demais ela não deve ser descartada no comando `subset`, como feito no exemplo anterior.

```
df_aprovacoes_porte |>
  subset(select = -MES) |>
  aggregate(by = . ~ ANO, sum) |>
  head()
```

	ANO	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	1711.645	0.000000	216.8279	7048.91
2	1996	1431.916	0.000000	271.6768	11362.08
3	1997	2179.985	1.053206	137.0893	16672.74
4	1998	1375.066	34.909957	1454.6207	20161.79
5	1999	1394.739	426.779631	1083.7491	16634.18
6	2000	2366.893	864.823930	1534.4957	22858.39

### 14.3 Valores Faltantes - NA

A função `aggregate` possui como padrão o argumento `na.action = na.omit`, assim os valores NA são omitidos. Caso seja necessário considerar os valores faltantes deve ser informada uma função “alternativa”, que trata estes registros. No exemplo abaixo foi informado `NULL`, ou seja, “nenhuma” função a ser aplicada sobre os valores faltantes. Assim eles serão considerados no cálculo.

```
df_aprovacoes_porte_na <- df_aprovacoes_porte

df_aprovacoes_porte_na[1,]$MICRO <- NA

head(df_aprovacoes_porte_na)
```

	ANO	MES	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	1	NA	0	10.18922	380.2330
2	1995	2	106.3283	0	16.21161	495.5282
3	1995	3	234.5488	0	13.69085	715.9591
4	1995	4	125.2196	0	16.44511	403.9919
5	1995	5	209.4168	0	20.88794	477.2529
6	1995	6	122.5179	0	23.86818	473.2194

```
df_aprovacoes_porte_na |>
  subset(select = -MES) |>
  aggregate(by = . ~ ANO, sum,
            na.action = NULL) |>
  head()
```

	ANO	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	NA	0.000000	216.8279	7048.91
2	1996	1431.916	0.000000	271.6768	11362.08
3	1997	2179.985	1.053206	137.0893	16672.74
4	1998	1375.066	34.909957	1454.6207	20161.79
5	1999	1394.739	426.779631	1083.7491	16634.18
6	2000	2366.893	864.823930	1534.4957	22858.39

Veja que agora a soma para o ano de 1995 para o porte MICRO é NA.

Caso seja inserido `na.rm = T` como argumento da função `sum`, os valores faltantes são desconsiderados novamente, mesmo `na.action` sendo nulo.

```
df_aprovacoes_porte_na |>
  subset(select = -MES) |>
  aggregate(by = . ~ ANO, sum, na.rm = T,
            na.action = NULL) |>
  head()
```

	ANO	MICRO	PEQUENA	MEDIA	GRANDE
1	1995	1583.175	0.000000	216.8279	7048.91
2	1996	1431.916	0.000000	271.6768	11362.08
3	1997	2179.985	1.053206	137.0893	16672.74
4	1998	1375.066	34.909957	1454.6207	20161.79
5	1999	1394.739	426.779631	1083.7491	16634.18
6	2000	2366.893	864.823930	1534.4957	22858.39

---

Última atualização: 11/10/2024 - 21:53:42

# 15 Utilidades

Status

R oferece uma série de funções para interação com o ambiente externo.

## 15.1 Listar Arquivos

A função `list.files` exibe arquivos em um diretório informado no argumento `path`.

```
list.files('./data/')
```

```
[1] "aprovacoes_por_porte_de_empresa.csv" "aprovacoes_por_porte_de_empresa.RDS"  
[3] "csv_serie_sgs_4390.csv"
```

Podem ser retornados os caminhos completos dos arquivos com o parâmetro `full.names`.

```
list.files('./data/', full.names = T)
```

```
[1] "./data/aprovacoes_por_porte_de_empresa.csv"  
[2] "./data/aprovacoes_por_porte_de_empresa.RDS"  
[3] "./data/csv_serie_sgs_4390.csv"
```

Esta função também oferece a opção de buscar arquivos com algum padrão em seu nome através do parâmetro `pattern`.

```
list.files('.', pattern = '.yaml')
```

```
[1] "_quarto.yaml"
```

## 15.2 Listar diretórios

De forma análoga à função `list.files`, a função `list.dir` exibe os diretórios de um caminho informado. Porém esta função possui `TRUE` como valores padrão dos argumentos `full.names` e `recursive`.

```
list.dirs("C:/Arquivos de Programas/R/", recursive = F)
```

```
[1] "C:/Arquivos de Programas/R/R-4.3.0" "C:/Arquivos de Programas/R/R-4.4.1"
```

## 15.3 Informações de arquivos

A função `file.info` retorna uma série de informações sobre o arquivo como tamanho, modo, horário de modificação, etc.

```
t(file.info('./data/aprovacoes_por_porte_de_empresa.csv'))
```

```
./data/aprovacoes_por_porte_de_empresa.csv
size "24212"
isdir "FALSE"
mode "444"
mtime "2023-08-22 20:20:27"
ctime "2023-08-22 21:22:18"
atime "2024-10-11 22:29:16"
exe "no"
```

## 15.4 Variáveis de Ambiente

Existem muitas variáveis de ambiente em R e você também pode fazer uso destas para armazenar valores fora de objetos. Por exemplo, para buscar o nome do usuário que está utilizando a máquina no momento você pode usar o comando abaixo.

```
Sys.getenv('USERNAME')
```

```
[1] "luisg"
```

---

Última atualização: 11/10/2024 - 21:50:46



**Parte II**  
**Pacotes**

## O que são pacotes?

Um pacote em R é basicamente um conjunto de funções e/ou funcionalidades criadas por terceiros que “expandem” o poder da linguagem. A principal opção para instalação de pacotes é através do [CRAN](#). O CRAN é um repositório que contém milhares de pacotes (21469 em 11/10/2024). Nele também podem ser encontrados pacotes em suas versões “antigas”. Caso algum pacote não esteja hospedado no CRAN, ele também pode ser instalado, diretamente do arquivo fornecido pelo desenvolvedor do pacote por exemplo (muitos distribuem através do [Github](#)).

Existem alguns pacotes “especiais” em R que compõem a própria linguagem. Estes pacotes possuem suas versões idênticas à da linguagem e são “classificados” com prioridade “base”. Assim quando se faz a instalação da linguagem R, muitos pacotes também são instalados.

---

R Core Team (2023c)

Última atualização: 11/10/2024 - 21:50:10

# 16 Introdução a Pacotes

Status □□□

## 16.1 Pacotes Instalados

Podemos ver os pacotes instalados com o comando `installed.packages`:

```
# Exibindo 5 primeiros
as.data.frame(installed.packages())$Package[1:5]
```

```
[1] "askpass" "backports" "base64enc" "bit" "bit64"
```

A função `installed.packages` retorna uma série de informações a respeito dos pacotes. Abaixo alguns exemplos de pacotes bastante utilizados. Para simplificar a visualização foi usada função `t`, que transpõe o `data.frame` de colunas para linhas.

```
pacotes <- as.data.frame(installed.packages())
# pacote base
t(pacotes[pacotes$Package == 'base',])
```

	base
Package	"base"
LibPath	"C:/Program Files/R/R-4.4.1/library"
Version	"4.4.1"
Priority	"base"
Depends	NA
Imports	NA
LinkingTo	NA
Suggests	"methods"
Enhances	"chron, date, round"
License	"Part of R 4.4.1"
License_is_FOSS	NA

```
License_restricts_use NA
OS_type                NA
MD5sum                 NA
NeedsCompilation       NA
Built                  "4.4.1"
```

```
# pacote MASS
t(pacotes[pacotes$Package == 'MASS',])
```

```
Package                MASS
LibPath                "C:/Program Files/R/R-4.4.1/library"
Version                "7.3-60.2"
Priority                "recommended"
Depends                "R (>= 4.4.0), grDevices, graphics, stats, utils"
Imports                "methods"
LinkingTo              NA
Suggests               "lattice, nlme, nnet, survival"
Enhances               NA
License                "GPL-2 | GPL-3"
License_is_FOSS        NA
License_restricts_use NA
OS_type                NA
MD5sum                 NA
NeedsCompilation       "yes"
Built                  "4.4.1"
```

```
# pacote bit64
t(pacotes[pacotes$Package == 'bit64',])
```

```
Package                bit64
LibPath                "C:/Users/luisg/OneDrive/Área de Trabalho/r/ecd/.packages"
Version                "4.0.5"
Priority                NA
Depends                "R (>= 3.0.1), bit (>= 4.0.0), utils, methods, stats"
Imports                NA
LinkingTo              NA
Suggests               NA
Enhances               NA
License                "GPL-2 | GPL-3"
```

```
License_is_FOSS      NA
License_restricts_use NA
OS_type              NA
MD5sum              NA
NeedsCompilation     "yes"
Built                "4.4.0"
```

Pode ser visto no campo *Priority* que para o pacote base o conteúdo é “base”, isto significa que este faz parte da instalação do R. Já o pacote [MASS](#), por exemplo, é um pacote recomendado. O pacote bit64, que é um pacote “normal”, não possui informação no campo *Priority*.

Também podemos visualizar dados do pacote (arquivo *DESCRIPTION* do próprio pacote) com o comando `packageDescription`:

```
packageDescription('base')
```

```
Package: base
Version: 4.4.1
Priority: base
Title: The R Base Package
Author: R Core Team and contributors worldwide
Maintainer: R Core Team <do-use-Contact-address@r-project.org>
Contact: R-help mailing list <r-help@r-project.org>
Description: Base R functions.
License: Part of R 4.4.1
Suggests: methods
Enhances: chron, date, round
Built: R 4.4.1; ; 2024-06-14 08:24:09 UTC; windows

-- File: C:/PROGRA~1/R/R-44~1.1/library/base/Meta/package.rds
```

## 16.2 Pasta de Instalação

O R possui pastas de instalação dos pacotes. Para visualizá-las basta usar o comando `.libPaths`. A pasta padrão de instalação traz os diversos pacotes que foram instalados junto com o R (os “básicos” e os recomendados).

```
.libPaths()
```

```
[1] "C:/Users/luisg/OneDrive/Área de Trabalho/r/ecd/.packages"
[2] "C:/Program Files/R/R-4.4.1/library"
```

```
# Exibir 10 primeiros da primeira pasta
list.files(.libPaths()[1])[1:10]
```

```
[1] "askpass"    "backports" "base64enc" "bit"        "bit64"     "blob"
[7] "broom"     "bslib"     "cachem"    "callr"
```

```
# Exibir 10 primeiros da segunda pasta
list.files(.libPaths()[2])[1:10]
```

```
[1] "base"      "boot"      "class"     "cluster"   "codetools" "compiler"
[7] "datasets" "foreign"   "graphics"  "grDevices"
```

## 16.3 Pacotes Disponíveis

A função `available.packages` procura pacotes disponíveis no valor informado no argumento `repos`. Por padrão é buscado de `getOption("repos")`.

```
# Definir repositório
options(repos = 'https://cran.rstudio.com/')

# Exibindo 5 primeiros
available.packages()[1:5]
```

```
[1] "A3"          "AalenJohansen" "AATtools"      "ABACUS"
[5] "abasequence"
```

## 16.4 Dependências de Pacotes

Os pacotes podem e em sua maioria utilizam funções de outros pacotes. Estes “outros pacotes” são denominadas de dependências. As informações de dependências também constam no *DESCRIPTION* do pacote.

O pacote `tools`, que faz parte da base do R, oferece uma função para busca de dependências de pacotes. Inclusive existe a opção de recursividade, ou seja, busca também as dependências das dependências do pacote desejado.

```
tools::package_dependencies('dplyr')
```

```
$dplyr  
 [1] "cli"          "generics"    "glue"        "lifecycle"   "magrittr"  
 [6] "methods"     "pillar"     "R6"          "rlang"       "tibble"  
[11] "tidyselect"  "utils"      "vctrs"
```

```
tools::package_dependencies('dplyr', recursive = T)
```

```
$dplyr  
 [1] "cli"          "generics"    "glue"        "lifecycle"   "magrittr"  
 [6] "methods"     "pillar"     "R6"          "rlang"       "tibble"  
[11] "tidyselect"  "utils"      "vctrs"       "fansi"       "utf8"  
[16] "pkgconfig"  "withr"      "grDevices"  "graphics"
```

## 16.5 Instalação de Pacotes

Para efetuar a instalação de pacotes usa-se a função `install.packages`. Os pacotes podem ser instalados diretamente de repositórios na Internet (como o CRAN) ou de arquivos locais.

---

Última atualização: 11/10/2024 - 21:49:19

# 17 Pacote data.table

Status

O pacote **data.table** oferece estrutura para manipulação de bases de dados de forma mais rápida que a base do R. As estruturas de dados são chamadas de **data tables** e funcionam de forma similar a um **data frame**.

```
library(data.table)

df_aprovacoes <-
  readRDS('./data/aprovacoes_por_porte_de_empresa.RDS')
```

## 17.1 Criando um Data Table

### 17.1.1 Função data.table

A criação de um **data table** se dá de forma análoga a um **data frame** através da função `data.table`.

```
df_exemplo <- data.table(x = 1:10)
df_exemplo |> class()
```

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

### 17.1.2 Função setDT

Também podemos criar um **data table** a partir de um **data frame** com o comando `setDT` ou com a função `as.data.table`.

```
df_aprovacoes_as <- as.data.table(df_aprovacoes)
df_aprovacoes_as |> class()
```



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

```
df_aprovacoes <- setDT(df_aprovacoes)
df_aprovacoes |> class()
```

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

```
identical(df_aprovacoes_as, df_aprovacoes)
```

```
[1] TRUE
```

As operações realizadas com **data tables** possuem sintaxe específica e bastante sucinta. As operações são feitas sobre a base de dados na seguinte forma:

*base[ linhas, colunas, agregação ]*

## 17.2 Filtrando Dados

De forma análogo a um **data frame** padrão, os filtros nas linhas ocorrem antes da vírgula e podem usar operadores lógicos e relacionais. Uma grande vantagem é que as colunas podem ser referenciadas apenas pelo seu nome, sem incluir o nome do objeto.

```
df_aprovacoes[ANO == 2022 & MES > 6, ]
```

	ANO	MES	MICRO	PEQUENA	MEDIA	GRANDE
	<int>	<int>	<num>	<num>	<num>	<num>
1:	2022	7	1763.7773	3843.3850	3286.456	1873.511
2:	2022	8	2378.3861	4640.6140	3965.469	3147.633
3:	2022	9	874.7404	1198.8596	2163.194	4625.476
4:	2022	10	419.9439	932.4995	1970.361	9020.808
5:	2022	11	323.8321	828.8072	2259.412	7908.244
6:	2022	12	413.4742	1087.6930	3795.945	37107.079

## 17.3 Selecionando Colunas

Para seleção de colunas, podemos colocar o nome da coluna diretamente após a vírgula, porém o retorno é dado em forma de vetor e não mantém a estrutura do **data table**.

```
df_aprovacoes[, GRANDE] |>
  is.vector()
```

```
[1] TRUE
```

```
df_aprovacoes[, GRANDE] |>
  is.data.table()
```

```
[1] FALSE
```

Para seleção mais robusta devem ser informadas as colunas dentro de uma lista, que pode ser abreviada por um ponto.

```
df_aprovacoes[, list(GRANDE)] |>
  is.data.table()
```

```
[1] TRUE
```

```
# selecionando mais de uma coluna
df_aprovacoes[, list(GRANDE, PEQUENA)] |>
  head(3)
```

	GRANDE	PEQUENA
	<num>	<num>
1:	380.2330	0
2:	495.5282	0
3:	715.9591	0

```
# usando ponto
df_aprovacoes[, .(GRANDE, PEQUENA)] |>
  head(3)
```

	GRANDE	PEQUENA
	<num>	<num>
1:	380.2330	0
2:	495.5282	0
3:	715.9591	0

## 17.4 Alterando Variáveis

A criação/alteração de variáveis pode ser feita basicamente de duas formas: forma **funcional** ou **LHS := RHS**. Em ambas as formas as operações são feitas diretamente no próprio objeto, não exigindo o operador de atribuição <-.

### 17.4.1 Forma Funcional

Na forma funcional se faz o uso do operador := (*assignment by reference*), que deixa o código bastante limpo e reduzido, facilitando a leitura.

```
df_aprovacoes[, RAZAO_GRANDE_MICRO := GRANDE / MICRO]
df_aprovacoes[, RAZAO_GRANDE_MICRO] |> head()
```

```
[1] 2.959705 4.660361 3.052496 3.226267 2.278962 3.862450
```

É possível efetuar a atualização/criação de várias variáveis simultaneamente.

```
df_aprovacoes[, `:=`(RAZAO_GRANDE_PEQUENA = GRANDE / PEQUENA,
                      RAZAO_GRANDE_MEDIA = GRANDE/MEDIA)]
```

### 17.4.2 Forma LHS := RHS

Uma forma alternativa pode ser usada com dois “vetores”, um com as variáveis a serem criadas e outro com os valores a serem atribuídos.

```
df_aprovacoes2 <- copy(df_aprovacoes)
df_aprovacoes[, c('RAZAO_GRANDE_PEQUENA', 'RAZAO_GRANDE_MEDIA') :=
                 list(GRANDE / PEQUENA, GRANDE/MEDIA)]

identical(df_aprovacoes, df_aprovacoes2)
```

```
[1] TRUE
```

## 17.5 Agrupando

O agrupamento/agregação ocorre com o operador `by`. Exemplo com criação da média de valores aprovados para o porte **MICRO** por ano, sendo que o ano deve ser a partir de 2020.

```
df_aprovacoes[ANO > 2019, list(MEAN_MICRO = mean(MICRO)), by = ANO]
```

	ANO	MEAN_MICRO
	<int>	<num>
1:	2020	536.0065
2:	2021	536.2352
3:	2022	662.1226
4:	2023	312.2800

## 17.6 Ordenando Dados

A função `setorder` permite ordenação da base de dados sem exigir atribuição como usado com R base (`df <- df[order(df$var),]`).

```
df_aprovacoes |> head()
```

	ANO	MES	MICRO	PEQUENA	MEDIA	GRANDE	RAZAO_GRANDE_MICRO
	<int>	<int>	<num>	<num>	<num>	<num>	<num>
1:	1995	1	128.4699	0	10.18922	380.2330	2.959705
2:	1995	2	106.3283	0	16.21161	495.5282	4.660361
3:	1995	3	234.5488	0	13.69085	715.9591	3.052496
4:	1995	4	125.2196	0	16.44511	403.9919	3.226267
5:	1995	5	209.4168	0	20.88794	477.2529	2.278962
6:	1995	6	122.5179	0	23.86818	473.2194	3.862450
	RAZAO_GRANDE_PEQUENA		RAZAO_GRANDE_MEDIA				
			<num>	<num>			
1:			Inf	37.31717			
2:			Inf	30.56625			
3:			Inf	52.29471			
4:			Inf	24.56609			
5:			Inf	22.84825			
6:			Inf	19.82637			

```
setorder(df_aprovacoes, -ANO, MES)
df_aprovacoes |> head(n = 8)
```

	ANO	MES	MICRO	PEQUENA	MEDIA	GRANDE	RAZAO_GRANDE_MICRO	
	<int>	<int>	<num>	<num>	<num>	<num>	<num>	
1:	2023	1	235.2636	654.6506	1048.4110	1337.163	5.683680	
2:	2023	2	528.9812	1639.7768	2111.9314	1088.138	2.057045	
3:	2023	3	260.0790	647.7431	1546.9274	1843.952	7.089969	
4:	2023	4	157.7072	501.0023	1190.8649	3445.090	21.844841	
5:	2023	5	119.9866	519.2932	1797.9574	2222.737	18.524876	
6:	2023	6	571.6626	1985.3709	3554.8101	6506.082	11.380983	
7:	2022	1	293.7928	412.8598	826.2161	1182.581	4.025222	
8:	2022	2	195.4029	571.3932	910.6876	1460.593	7.474776	
			RAZAO_GRANDE_PEQUENA			RAZAO_GRANDE_MEDIA		
			<num>			<num>		
1:			2.0425595			1.2754186		
2:			0.6635891			0.5152336		
3:			2.8467333			1.1920093		
4:			6.8763948			2.8929308		
5:			4.2803128			1.2362571		
6:			3.2770111			1.8302194		
7:			2.8643651			1.4313220		
8:			2.5561957			1.6038352		

## 17.7 Encadeamento

Existe a possibilidade de operações encadeadas. Estas operações são úteis quando se deseja transformar uma variável recém criada, por exemplo.

```
df_aprovacoes3 <- copy(df_aprovacoes)
df_aprovacoes3[, `:=`(DATA_ATUAL = Sys.Date(),
                       ANO_ATUAL = format(DATA_ATUAL, '%y'))]
```

Error in eval(jsub, SDeval, parent.frame()): objeto 'DATA\_ATUAL' não encontrado

Sem o encadeamento não é possível extrair o ano da variável **DATA\_ATUAL** pois ela ainda não foi criada.

```
df_aprovacoes4 <- copy(df_aprovacoes)
df_aprovacoes4[, DATA_ATUAL := Sys.Date()][,
  ANO_ATUAL := format(DATA_ATUAL, '%y')]
```

O encadeamento é equivalente ao uso do *pipe* e o *placeholder* indicando o data table.

```
df_aprovacoes5 <- copy(df_aprovacoes)
df_aprovacoes5[, DATA_ATUAL := Sys.Date()] |>
  _[, ANO_ATUAL := format(DATA_ATUAL, '%y')]

identical(df_aprovacoes4, df_aprovacoes5)
```

[1] TRUE

—————  
Dowle  
e Sri-  
niva-  
san  
(2023)

engine:  
knitr  
—————

[Status] (convencoes.html#status-do-material)

## O que é o RStudio?

O **RStudio** é um IDE (Integrated Development Environment) criado pela **Posit** (<https://posit.co/>)

```
::: {.cell}
::: {.cell-output-display}
![RStudio - Tela inicial](images/rstudio/rstudio_layout.png){#fig-rstudio}
:::
:::
```

```
::: {.cell}
::: {.cell-output-display}
![About RStudio](images/rstudio/about_rstudio.png){#fig-rstudio-about}
:::
:::
```

---

[RStudio - User Guide](<https://docs.posit.co/ide/user/>)

[IDE]([https://en.wikipedia.org/wiki/Integrated\\_development\\_environment](https://en.wikipedia.org/wiki/Integrated_development_environment))

Última atualização: 11/10/2024 - 21:49:56

:::

```
`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4ifQ== -->`{=html}
```

```
````{=html}
```

```
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJjaGFwdGVyIiwiaWYm9va...
...`
```

# Introdução ao RStudio

```
````````{.quarto-title-block template='C:\Users\luisg\AppData\Local\Programs\Quarto\share\pro...
---
```

```
engine: knitr
```

```
---
```

## 17.8 Layout

Status

O RStudio possui basicamente 4 painéis dimensionáveis e cada um deles painéis pode trazer uma série de abas. Você pode configurar a localização de cada painel conforme sua preferência nos menus: View > Panes > Pane Layout ou em Tools > Global Options > Pane Layout.

Dentro dos painéis *Environment* e *Files* podem ser adicionadas ou removidas diversas abas (basta marcar/desmarcar *checkbox*). Muitas delas ficam ocultas e são “chamadas” pelo RStudio apenas quando necessárias.

## 17.9 Console

Neste painel está embutido o R propriamente dito.

## 17.10 *Output*

Painel com diversas saídas fornecidas. Gráficos (*Plots*), Estrutura de Pastas(*Files*), Ajuda (*Help*), Pacotes(*Packages*), etc aparecem neste painel. Este é um painel muito útil para navegação nos arquivos do projeto e visualização/exportação de gráficos.

## 17.11 *Environment*

Apresenta os objetos criados no ambiente do R.



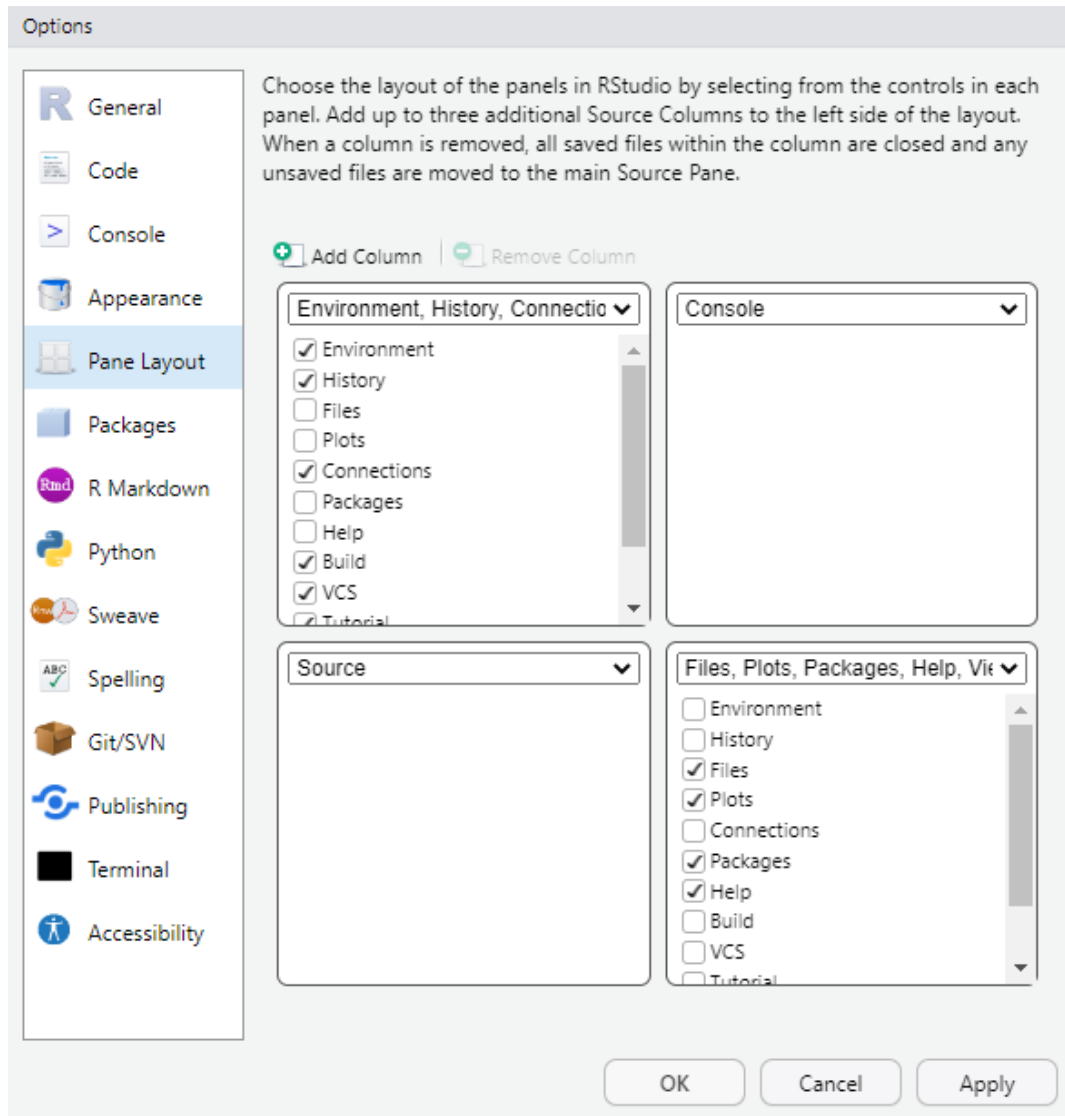
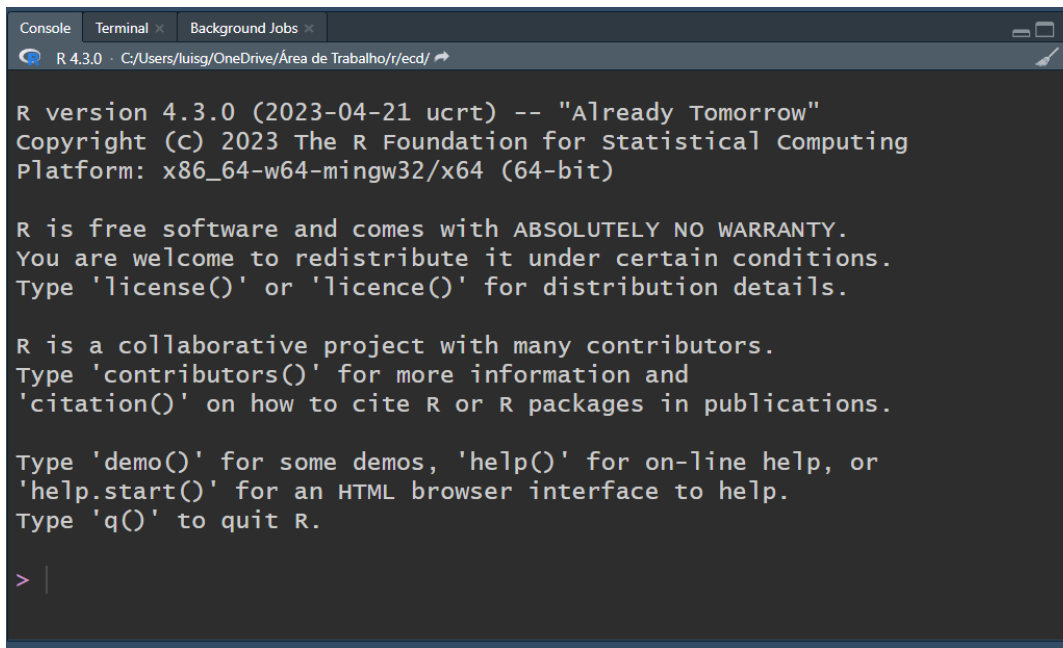


Figura 17.1: Pane Layout



R version 4.3.0 (2023-04-21 ucrt) -- "Already Tomorrow"  
Copyright (C) 2023 The R Foundation for Statistical Computing  
Platform: x86\_64-w64-mingw32/x64 (64-bit)

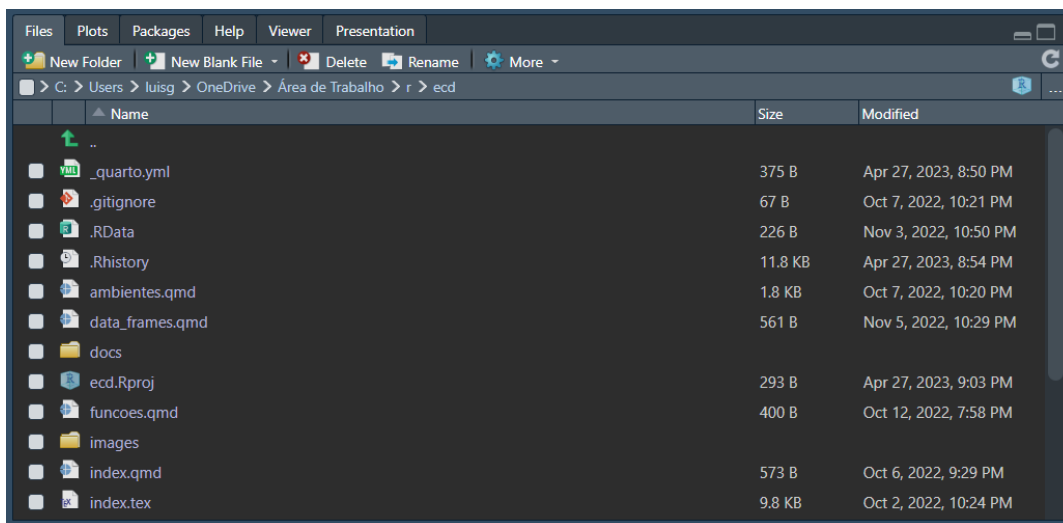
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.

>

Figura 17.2: Console



Name	Size	Modified
..		
.._quarto.yml	375 B	Apr 27, 2023, 8:50 PM
.gitignore	67 B	Oct 7, 2022, 10:21 PM
.RData	226 B	Nov 3, 2022, 10:50 PM
.Rhistory	11.8 KB	Apr 27, 2023, 8:54 PM
ambientes.qmd	1.8 KB	Oct 7, 2022, 10:20 PM
data_frames.qmd	561 B	Nov 5, 2022, 10:29 PM
docs		
ecd.Rproj	293 B	Apr 27, 2023, 9:03 PM
funcoes.qmd	400 B	Oct 12, 2022, 7:58 PM
images		
index.qmd	573 B	Oct 6, 2022, 9:29 PM
index.tex	9.8 KB	Oct 2, 2022, 10:24 PM

Figura 17.3: Files

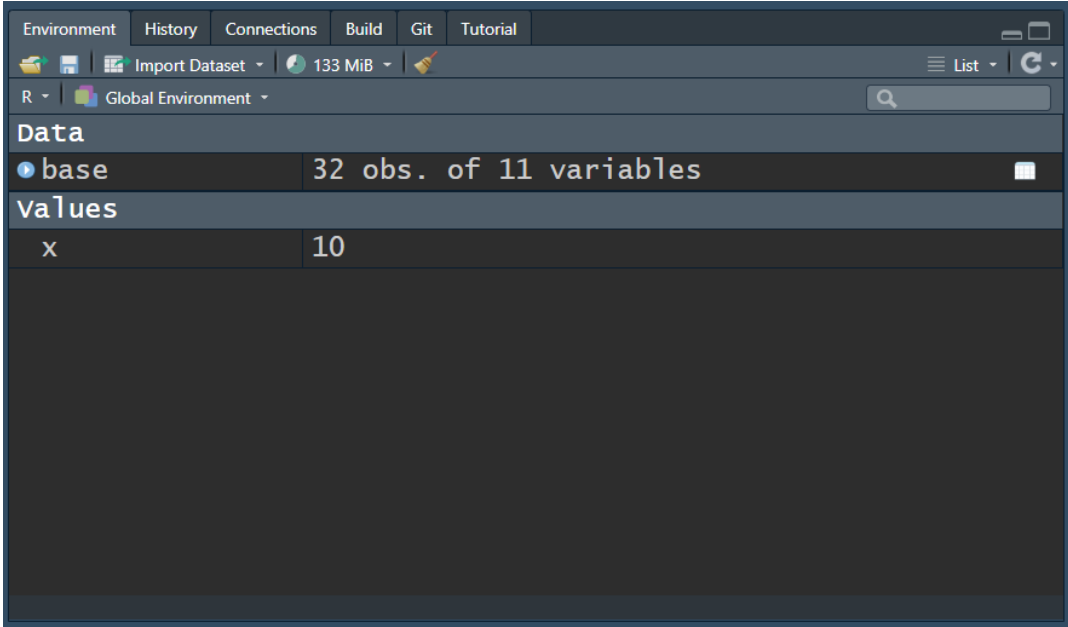


Figura 17.4: Environment

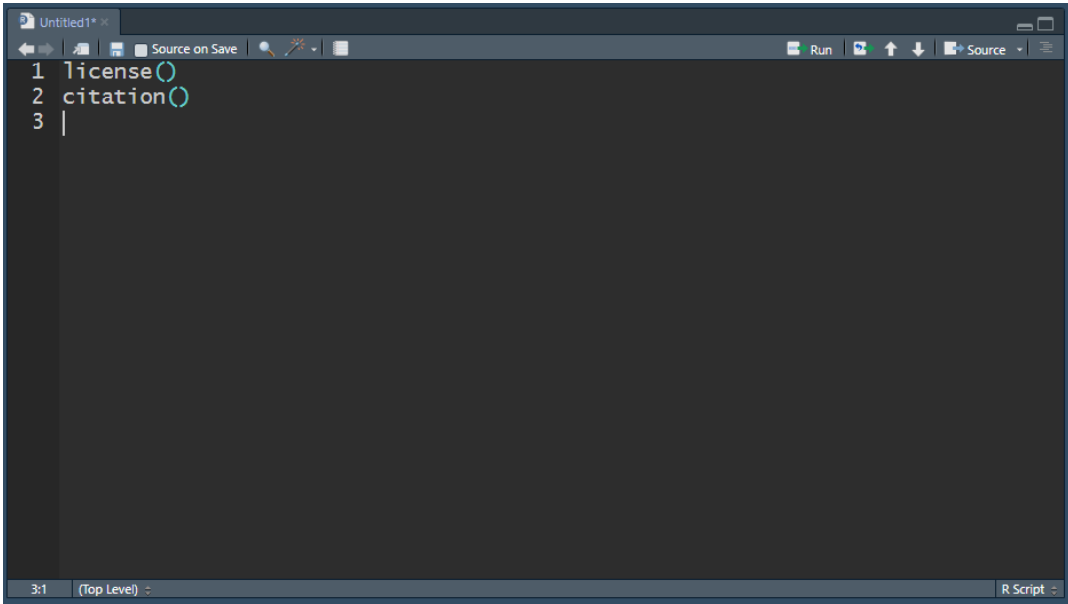


Figura 17.5: Source

## 17.12 *Source*

Aqui são abertos os arquivos de códigos (*scripts*, Rmarkdown, Quarto, SQL, etc).

---

[RStudio - User Guide](#)

[IDE](#)

Última atualização: 11/10/2024 - 21:50:26

# 18 Menu Tools

Status □□□

O menu Tools oferece uma série de funcionalidades para configuração do ambiente de trabalho.

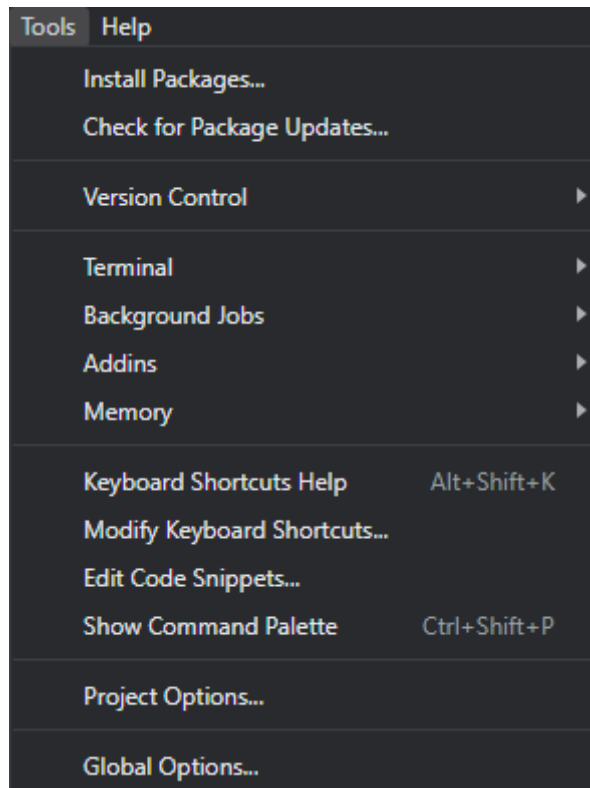


Figura 18.1: Menu Tools

## 18.1 Install Packages

Nesta opção é aberta a janela para instalação de pacotes.

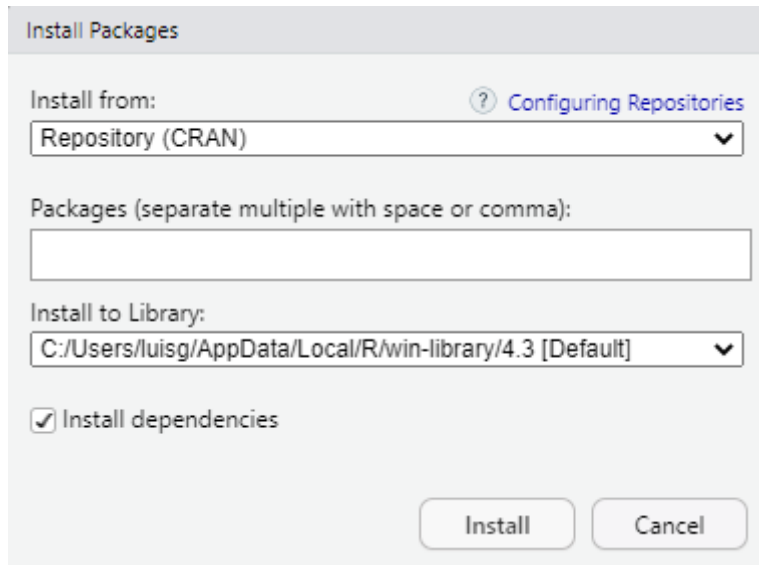


Figura 18.2: Install Packages

- **Install From:** local de busca dos pacotes a serem instalados
  - **Repository:** repositório configurado
    - \* Packages: nome dos pacotes a serem instalados. Podem ser escolhidos múltiplos pacotes, devendo ser separados por espaço ou vírgula
  - **Package Archive File:** opção para busca de arquivo a partir da máquina do usuário. Esta opção habilita botão para busca do pacote
    - \* Package archive: arquivo do pacote a ser instalado
- **Install to Library:** pasta de instalação dos pacotes
- **Install Dependencies:** marcação para que seja feita instalação de dependências dos pacotes selecionados.

## 18.2 Check for Package Updates

Esta opção abre a janela Update Packages, permitindo visualizar quais pacotes possuem versões mais recentes. A coluna NEWS possibilita visualizar o arquivo com dados de atualizações feitas no pacote.

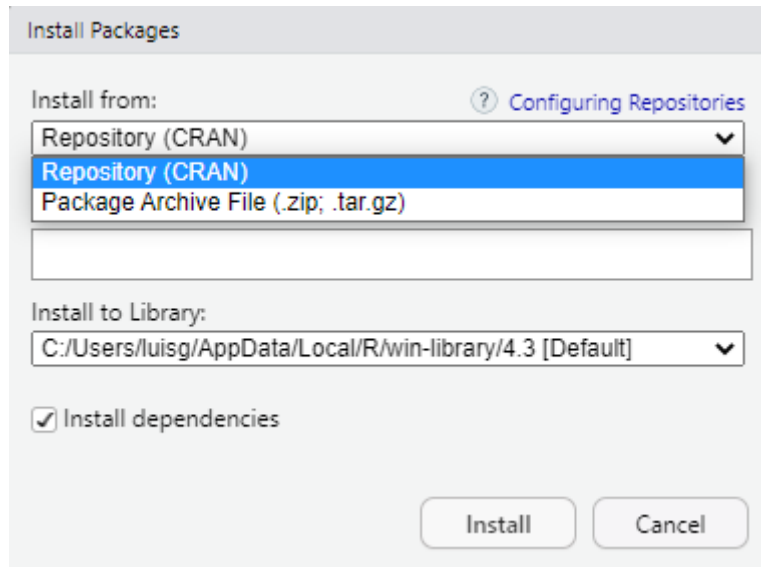


Figura 18.3: Install From

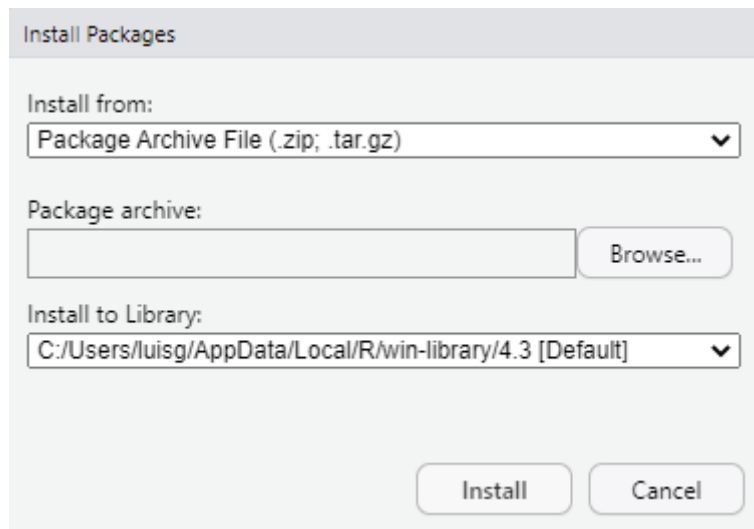


Figura 18.4: Package Archive File

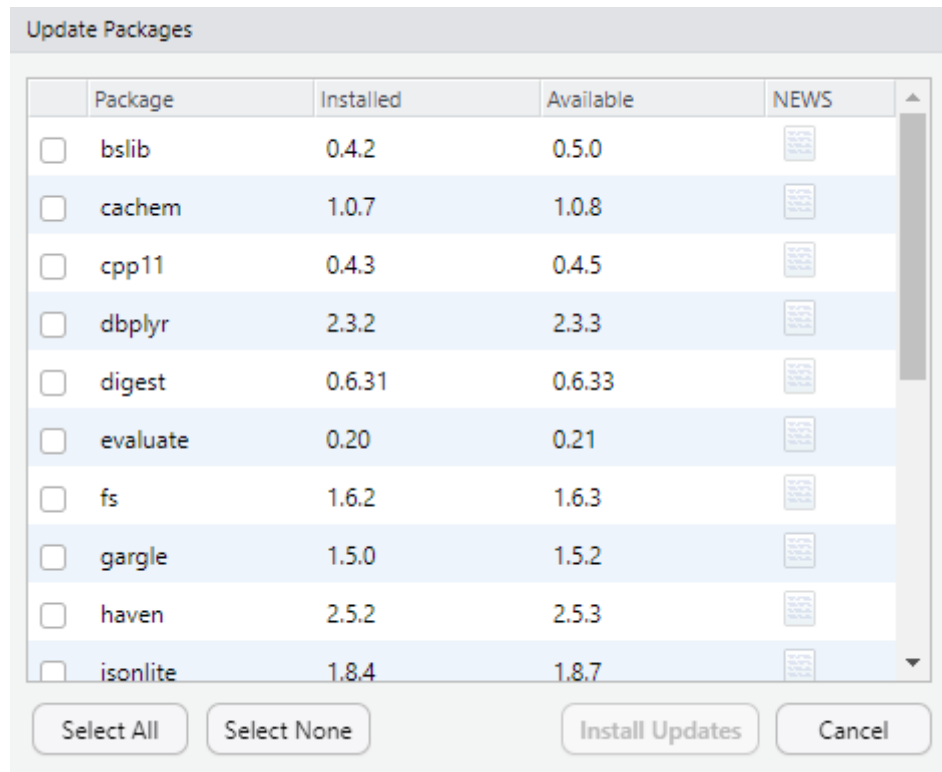


Figura 18.5: Check for Package Updates



## 18.3 Version Control

Oferece opção de controle de versões de código através do Git ou SVN.

## 18.4 Terminal

Permite acesso ao terminal do sistema operacional a partir do RStudio.

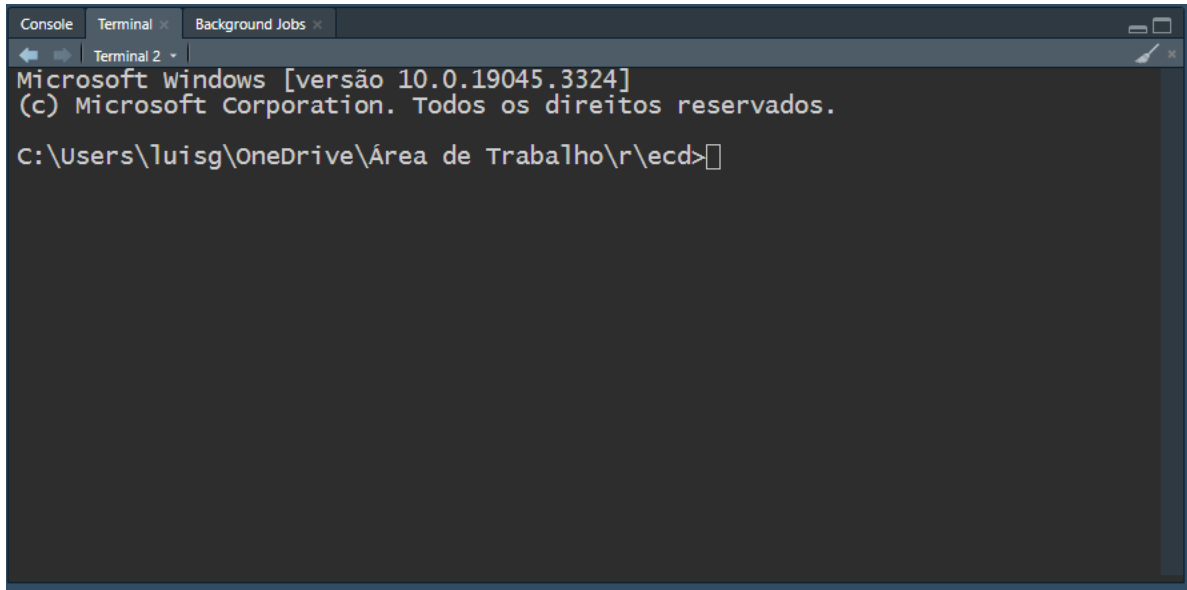


Figura 18.6: Terminal

## 18.5 Background Jobs

Fornecer opções para execução de 'Jobs', basicamente scripts em R, em outra instância do R. Desta forma a sessão aberta no RStudio não fica ocupada e permite que o usuário continue seu trabalho. Esta opção é muito útil para processamentos mais demorados.

## 18.6 Global Options

Esta opção abre a janela Options do RStudio onde podem ser feitas as principais configurações de comportamento da ferramenta.

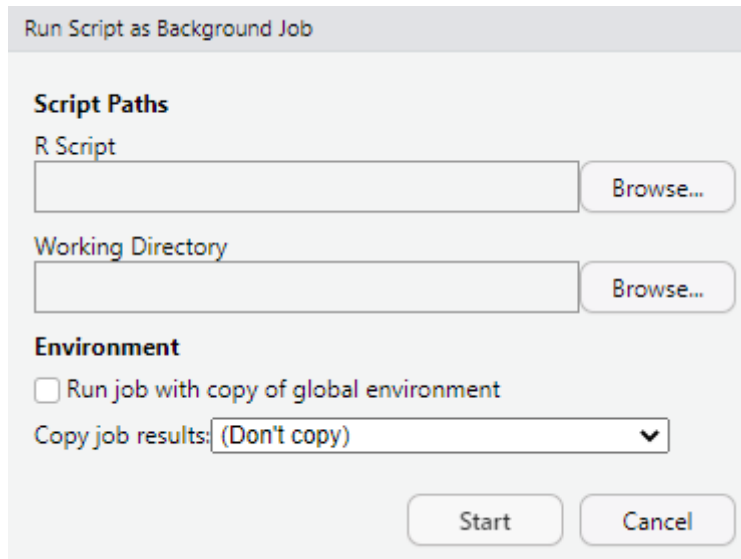


Figura 18.7: Background Jobs

### 18.6.1 Geral > Basic

Nesta tela inicial Geral > Basic podemos definir muitas características do RStudio, algumas das principais:

- **R Sessions**

- *R Version*: versão a ser usada do R dentro do RStudio. Esta versão pode ser alterada caso exista uma outra instalação no computador.
- *Restore most recently opened project at startup*: define se o projeto mais recente será carregado ao inicializar.
- *Restore previously open source documents at startup*: define se arquivos de código (*sources*) recentemente usados serão carregados ao inicializar.

- **Workspace**

- *Restore .RData into workspace at startup*: define se ao ser inicializado o RStudio carregará o arquivo *.RData* do projeto. Esta opção pode ser muito útil, pois resgata a sessão anterior onde ela foi fechada. Entretanto caso sejam usados arquivos muito grandes a inicialização pode demorar.
- *Save workspace to .RData on exit*: o “inverso” do anterior, define se os dados da sessão serão salvos ao fechar o RStudio. As opções são: *Always*, *Never* e *Ask*.

- **History**

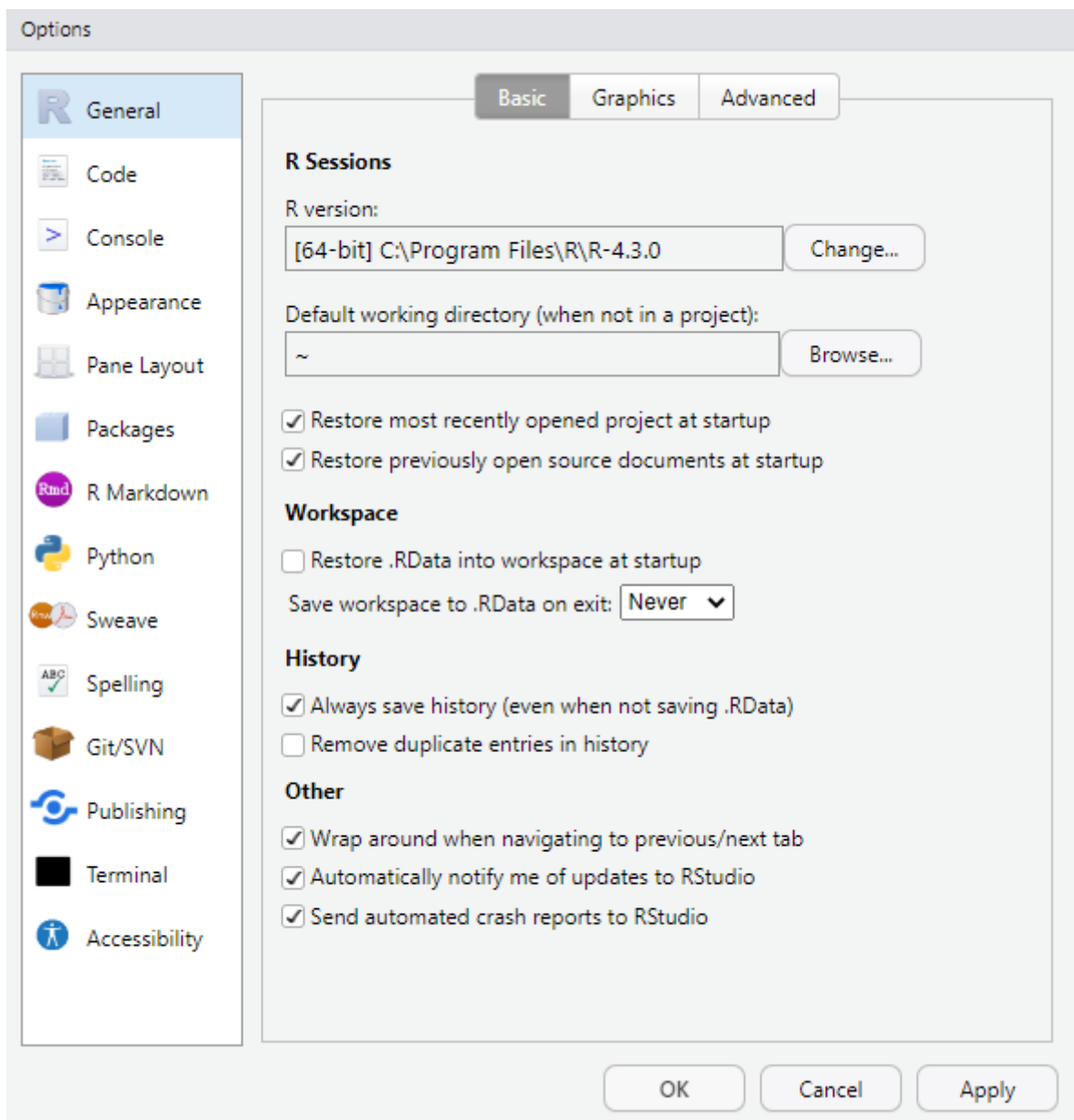


Figura 18.8: Global Options

- *Always saves History*: os comandos passados para o R serão ou não armazenados para consulta posterior?
- *Remove duplicate entries*: elimina as repetições, muitas vezes quando se efetuam testes os mesmos comandos são executados diversas vezes.

- **Other**

- *Automatically notify me of RStudio updates*: verificar e avisar o usuário se existirem atualizações do RStudio.

## 18.6.2 Appearance

Aqui são disponibilizadas diversas configurações visuais para o RStudio, inclusive permitindo a inclusão de um tema externo.

---

[RStudio - User Guide](#)

[IDE](#)

Última atualização: 11/10/2024 - 21:50:21

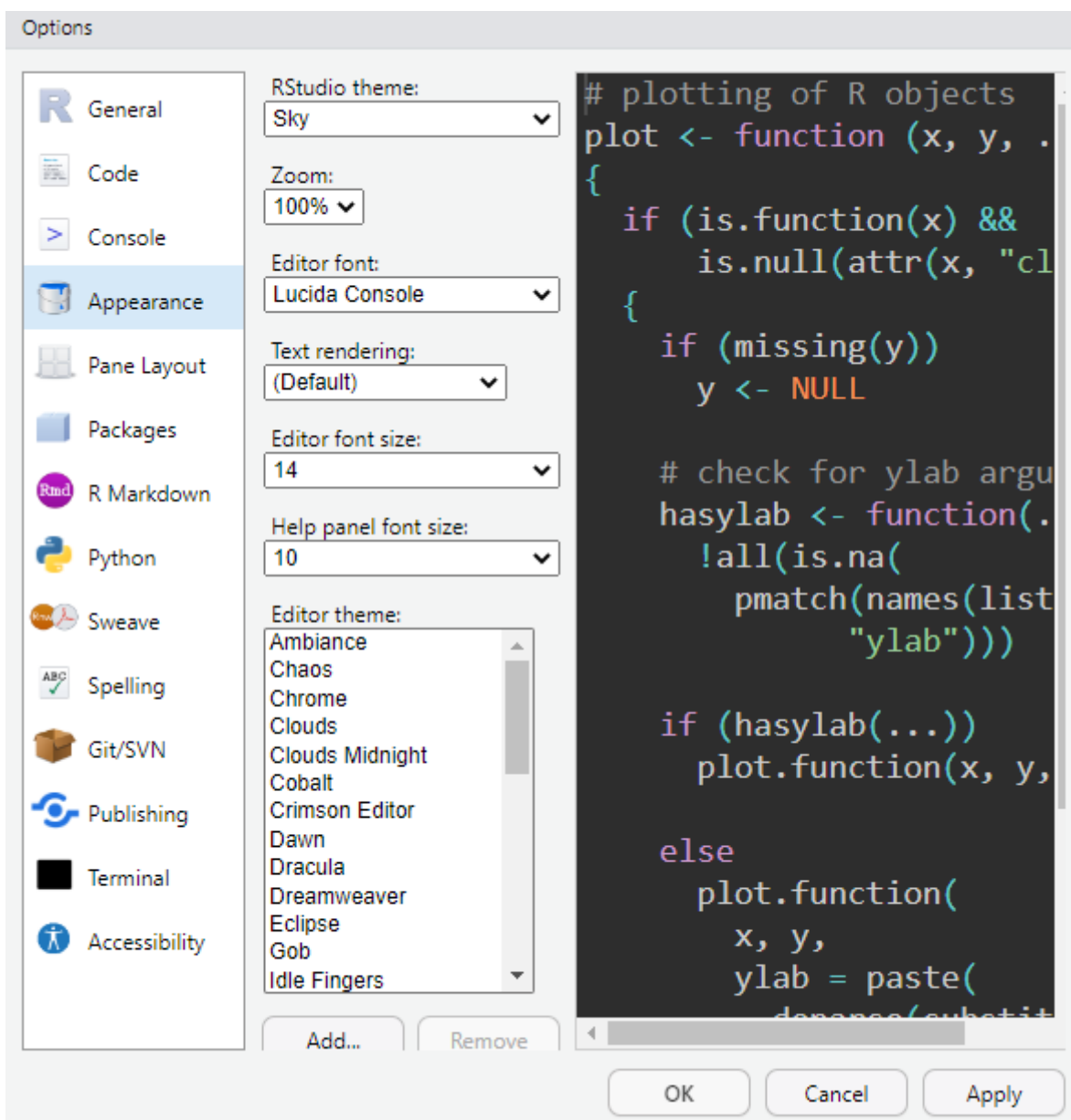


Figura 18.9: Global Options - Appearance

**Parte III**  
**Estatística**

Status □□□

## O que é Estatística

Para Larson e Farber (2007), 'estatística é a ciência que se ocupa de coletar, organizar, analisar e interpretar dados a fim de tomar decisões'.

---

Última atualização: 11/10/2024 - 21:45:38

# 19 Introdução

Status □□□

Neste capítulo serão discutidos alguns dos principais conceitos de Estatística e que serão base para a continuação.

## 19.1 Dados Qualitativos

São dados não numéricos, podem ser nominais ou ordinais. Operações matemáticas, como somas, diferenças, etc não fazem sentido para este tipo de dado.

## 19.2 Dados Quantitativos

São dados numéricos como contagens, medidas, etc. Podem ser contínuos (assumem valores do conjunto dos números Reais) ou discretos (assumem um conjunto finito de valores).

## 19.3 População

O termo população é usado em estatística no sentido de todo um conjunto de dados de interesse.

## 19.4 Amostra

Amostra é a definição usada para frações de uma população.

## 19.5 Parâmetros

Parâmetros são valores numéricos de uma população.



## 19.6 Estatísticas

Semelhante ao parâmetro, uma estatística trata de valores referentes a um grupo. Mas neste caso o grupo é uma parte da população, ou seja, uma amostra.

## 19.7 Tipos de Dados

### 19.7.1 Nominal

Dados do tipo **nominal** são qualitativos, usados para nomeação (rótulos) de grupos, classes, categorias, etc. Nenhum cálculo matemático pode ser feito sobre estes dados.

Um exemplo bastante comum de dado nominal é o **sexo**, geralmente descritos pelos valores F e M. Em R os dados nominais são criados com a função `factor`.

```
sexos <- c('F', 'M')
typeof(sexos)
```

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

```
is.factor(sexos)
```

```
[1] FALSE
```

```
sexos <- factor(c('F', 'M'))
is.factor(sexos)
```

```
[1] TRUE
```

```
typeof(sexos)
```

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

```
class(sexos)
```

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

Note que números podem ser usados como dados nominais e ainda assim não podem ser feitas operações matemáticas sobre eles.

```
sexos <- factor(c('1', '2'))  
sexos + 2
```

Warning in Ops.factor(sexos, 2): '+' não faz sentido para fatores

[1] NA NA

### 19.7.2 Ordinal

### 19.7.3 Intervalar

### 19.7.4 Razão

---

Larson e Farber (2007)

Gary A. simon (2000)

Luiz Paulo Fávero (2017) R Core Team (2023c)

Última atualização: 11/10/2024 - 21:47:23

## 20 Teorema de Bayes

Status □□□

### 20.1 Fórmula

$$P(B_i|A) = \frac{P(B_i \cap A)}{P(A)}$$

Exemplo:

Sendo tirada 1 bola vermelha, qual a probabilidade de esta bola ter sido retirada da caixa 1?

Caixa	Vermelhas	Azuis
1	4	4
2	2	6

As probabilidades “básicas” são:

$$P(B_1) = P(Caixa1) = 1/2$$

$$P(B_2) = P(Caixa2) = 1/2$$

$$P(A|B_1) = P(Vermelha|Caixa1) = 4/8$$

$$P(A|B_2) = P(Vermelha|Caixa2) = 2/8$$

A probabilidade de que uma bola vermelha tenha sido retirada da caixa 1 é dada por:

$$P(Caixa1|Vermelha) = \frac{P(B_1) \cdot P(A|B_1)}{P(A)}$$

O numerador da equação é:

$$P(B_1).P(A|B_1) = P(Caixa_1).P(Vermelha|Caixa_1)$$

$$P(Caixa_1).P(Vermelha|Caixa_1) = \frac{1}{2} \cdot \frac{4}{8}$$

O denominador da equação é:

$$P(A) = P(Vermelha) = P(Caixa_1).P(Vermelha|Caixa_1) + P(Caixa_2).P(Vermelha|Caixa_2)$$

$$P(Vermelha) = \frac{1}{2} \cdot \frac{4}{8} + \frac{1}{2} \cdot \frac{2}{8} = \frac{2}{8} + \frac{1}{8} = \frac{3}{8} = 0.375$$

$$P(Caixa_1|Vermelha) = \frac{\frac{1}{2} \cdot \frac{4}{8}}{\frac{1}{2} \cdot \frac{4}{8} + \frac{1}{2} \cdot \frac{2}{8}}$$

Portanto, a probabilidade é de:

$$P(Caixa_1|Vermelha) = \frac{\frac{1}{4}}{\frac{1}{4} + \frac{1}{8}} = \frac{\frac{1}{4}}{\frac{3}{8}} = \frac{1}{4} \cdot \frac{8}{3} = \frac{8}{12} = 0.66666...$$

Abaixo exemplo do mesmo cálculo executado em R.

```
p_cx1 <- 1/2
p_cx2 <- 1/2
p_vermelha_cx1 <- 4/8
p_vermelha_cx2 <- 2/8

p_vermelha <- p_cx1 * p_vermelha_cx1 + p_cx2 * p_vermelha_cx2

(p_cx1 * p_vermelha_cx1)/p_vermelha
```

```
[1] 0.6666667
```

---

Luiz Paulo Fávero (2017)

Última atualização: 11/10/2024 - 21:47:30

**Parte IV**

**Ciência de Dados**

Status □□□

## Definições

Segundo Provost e Fawcett (Provost e Fawcett 2016, p 3) 'data science é um conjunto de princípios fundamentais que norteiam a extração de conhecimentos a partir de dados'.

Uma definição da IBM (2023) para Ciência de Dados:

'A ciência de dados combina matemática e estatística, programação especializada, análise avançada, inteligência artificial (IA) e aprendizado de máquina com conhecimento em domínio específico para revelar insights acionáveis ocultos nos dados de uma organização'.

Uma definição mais sucinta da AWS (2023):

'A ciência de dados é o estudo dos dados para extrair insights significativos para os negócios'.

---

Última atualização: 11/10/2024 - 21:47:06

# 21 Trade-Off Viés x Variância

Status:

---

Última atualização: 11/10/2024 - 21:48:44

# Bases de Dados

Neste material são usadas diversas bases de dados abertas. Abaixo as fontes e as descrições.

## Banco Central do Brasil

### Taxa de juros - Selic acumulada no mês

**Acesso em:** 28/08/2023 - 20:15

Taxa de juros que representa a taxa média ajustada das operações compromissadas com prazo de um dia útil lastreadas com títulos públicos federais custodiados no Sistema Especial de Liquidação e de Custódia (Selic). Divulgação em % a.m.

## BNDES

### Por porte de empresa - Aprovações

**Acesso em:** 22/08/2023 - 20:20

[Dicionário de dados.](#)

**Descrição:** Estatísticas de aprovações de financiamentos considerando porte do cliente, setor, distribuição regional, grupo de produtos e outros.

---

Última atualização: 11/10/2024 - 21:48:41



# Convenções

Status □□□

## Marcações no Texto

A fim de facilitar a leitura e evidenciar itens importantes no texto, serão adotadas as seguintes marcações:

Tipo do Texto	Marcação	Exemplo
Funções, argumentos, operadores do R	Código	<code>print</code>
Objetos criados no R (vetores, data frames, variáveis de data frames) e termos relevantes	Negrito	<b>df_mtcars</b>
Palavras de língua estrangeira	Itálico	<i>pipe</i>

## Nomes de Objetos

Abaixo convenções a serem usadas neste material.

Tabela 21.2: Convenções de código

Tipo Objeto	Convenção	Exemplo
Data.frame, tibble ou data.table	snake_case iniciado por df ( <b>data frame</b> )	df_clientes
Variáveis de datasets	SCREAMING_SNAKE_CASE	NOME_CLIENTE
Funções	camelCase iniciado por <b>fn</b> , sendo a primeira palavra após fn um verbo	fnBuscarClientes
Demais (vetores, listas, etc.)	snake_case	nomes_cidades

## Status do Material

Para indicação de status do material apresentado serão usados os símbolos abaixo no topo de cada capítulo:

Indicador	Estrutura	Conteúdo	Status Geral
□□□	não iniciado	não iniciado	não iniciado
□□□	incipiente	não criado	incipiente
□□□	incipiente	incipiente	incipiente
□□□	incipiente	em revisão	incipiente
□□□	em revisão	em revisão	em revisão
□□□	em revisão	amadurecido	em revisão
□□□	em revisão	incipiente	incipiente
□□□	incipiente	amadurecido	incipiente
□□□	incipiente	desatualizado	desatualizado
□□□	amadurecido	incipiente	incipiente
□□□	amadurecido	em revisão	em revisão
□□□	amadurecido	desatualizado	desatualizado
□□□	amadurecido	amadurecido	amadurecido

- □ **incipiente:** recém iniciado, é o status mais volátil. Após melhores definições passa a ser marcado como □.
- □ **em revisão:** em alterações (grandes) para melhorias. Após este status será marcado como □.
- □ **amadurecido:** já passado por revisão. Pode sofrer pequenas alterações e atualizações. Caso se identifique que necessite de grandes alterações será marcado como:
  - □ para revisão por decisão
  - □ para revisão por desatualização
- □ **desatualizado:** necessita ser reescrito por força maior, como desatualização de conceitos ou códigos.
  - **não iniciado:** serve como marcação de 'todo'. usado para seções que se entendem necessárias mas que ainda não foram iniciadas.

---

Última atualização: 11/10/2024 - 21:46:49

## Referências

- AWS. 2023. “O que é ciência de dados?” 2023. <https://aws.amazon.com/pt/what-is/data-science/>.
- DataMentor. s.d. “R repeat loop”. <https://www.datamentor.io/r-programming/repeat-loop>.
- Dowle, Matt, e Arun Srinivasan. 2023. *data.table: Extension of 'data.frame'*.
- Gary A. Simon, John E. Freund e. 2000. *Estatística Aplicada: economia, administração e contabilidade*. Bookman.
- Grolemund, Garrett. 2014. *Hands-On Programming with R*. O'Reilly. <https://rstudio-education.github.io/hopr/>.
- IBGE. s.d. “IBGE - Cidades”. <https://cidades.ibge.gov.br/>.
- IBM. 2023. “What is data science?” 2023. <https://www.ibm.com/topics/data-science>.
- Kassambara, Al boukadel. s.d. “Statistical tools for high-throughput data analysis”. <http://www.sthda.com/english/>.
- Larson, Ron, e Betsy Farber. 2007. *Estatística Aplicada*. Prentice Hall.
- Luiz Paulo Fávero, Patrícia Belfiore. 2017. *Manual de Análise de Dados*. Elsevier.
- Mastropietro, Daniel. 2019. “Getting an environment’s name in R: the envnames package”. 2019. <https://www.r-bloggers.com/2019/05/getting-an-environments-name-in-r-the-envnames-package>.
- Provost, Foster, e Tom Fawcett. 2016. *DataScience para Negócios*. Alta Books.
- R Core Team. 2023a. *An Introduction to R*. R Foundation for Statistical Computing. <https://cran.r-project.org/doc/manuals/r-release/R-intro.html>.
- . 2023b. *R Language Definition*. R Foundation for Statistical Computing. <https://cran.r-project.org/doc/manuals/r-release/R-lang.html>.
- . 2023c. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. <https://www.R-project.org/>.
- . 2023d. “R: A Language and Environment for Statistical Computing”. Vienna, Austria: R Foundation for Statistical Computing. 2023. <https://cran.r-project.org/doc/manuals/r-devel/NEWS.html>.
- Schmuller, Joseph. 2019. *Análise Estatística com R*. Alta Books.
- Wickham, Hadley. 2023/04/21. “Differences between the base R and magrittr pipes”. 2023/04/21. <https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/>.
- Wikipedia, the free encyclopedia. 2023. “Naming convention (programming)”. 2023. [https://en.wikipedia.org/wiki/Naming\\_convention\\_\(programming\)](https://en.wikipedia.org/wiki/Naming_convention_(programming)).