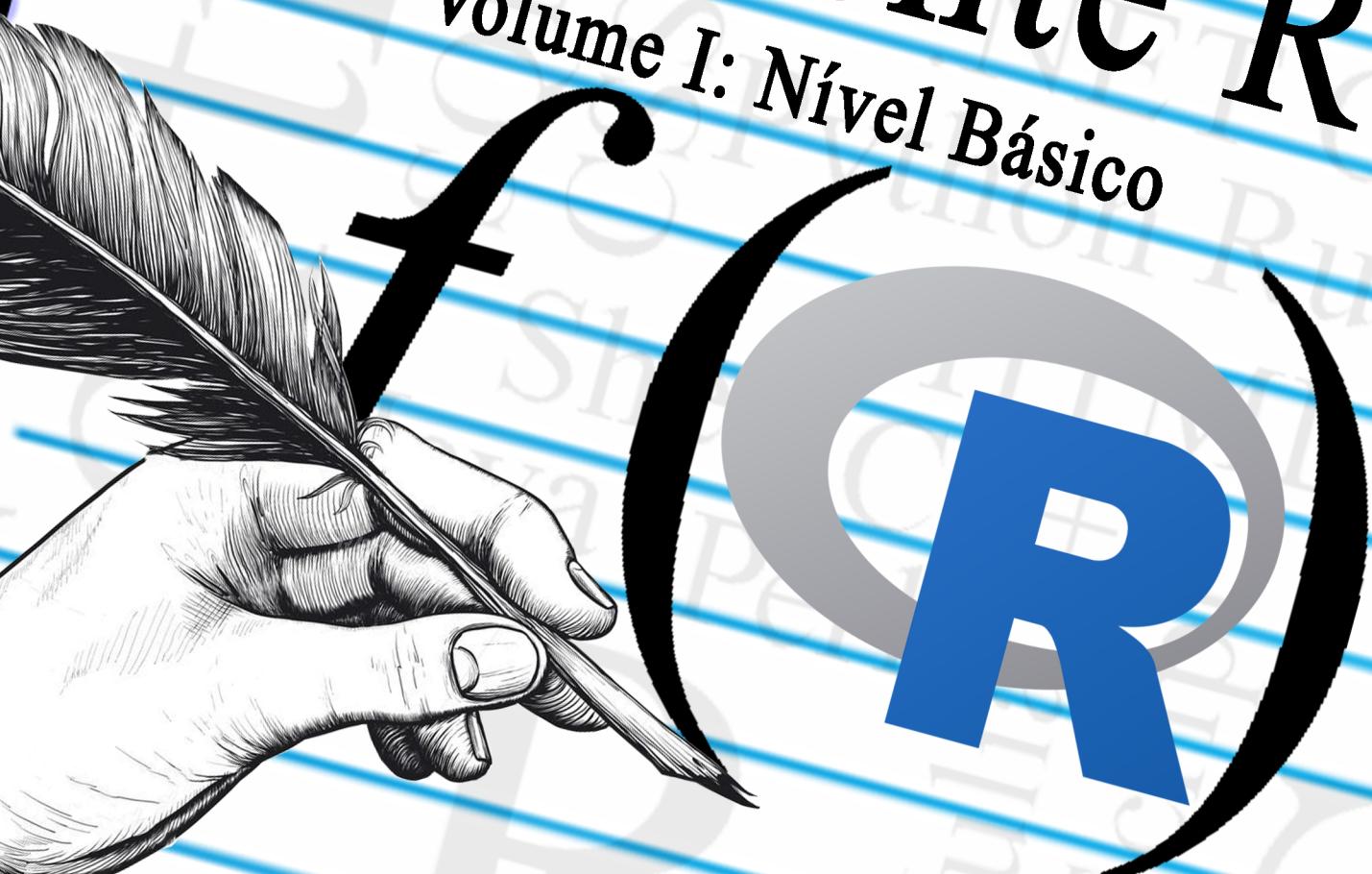


Estudando o Ambiente R

Volume I: Nível Básico



Ben Dêvide
Diego Arthur
Henrique JP Alves



Democratizando
Conhecimento

ESTUDANDO O AMBIENTE R

**BEN DÊIVIDE DE OLIVEIRA BATISTA
DIEGO ARTHUR BISPO JUSTINO DE OLIVEIRA
HENRIQUE JOSÉ PAULA ALVES**

Estudando o Ambiente R

Volume I: Nível Basico

BEN DÊIVIDE DE OLIVEIRA BATISTA
DIEGO ARTHUR BISPO JUSTINO DE OLIVEIRA
HENRIQUE JOSÉ PAULA ALVES



Ouro Branco, MG, 23 de fevereiro de 2022

© 2021 by Ben Dêivide de Oliveira Batista, Diego Arthur Bispo Justino de Oliveira e Henrique José Paula Alves



Esse material está licenciado com uma Licença Creative Commons - Atribuição - Não Comercial 4.0 Internacional. Usamos também a filosofia de trabalho com o Selo Democratizando Conhecimento (DC). O leitor é livre para compartilhar, redistribuir, transformar ou adaptar essa obra, desde que não venha a utilizá-la em nenhuma atividade de propósito comercial. Por fim, a única exigência é a atribuição dos dos créditos aos autores dessa obra.

Direitos de publicação reservados ao seu conhecimento.

Impresso no Brasil - ISBN (Digital):

Impresso no Brasil - ISBN (Impresso):

Projeto Gráfico: Ben Dêivide de Oliveira Batista

Revisão técnica e textual: XX

Revisão de Referências Bibliográficas:

Editoração Eletrônica: Ben Dêivide de Oliveira Batista

Capa: Ben Dêivide de Oliveira Batista

Como citar essa obra:

BATISTA, B. D. O.. **Estudando o Ambiente R.** 1ed. Ouro Branco, MG:[sn]. 2021. 1 v. Disponível em: <<https://bendeivide.github.io/book-eambr01/>>

Autor correspondente e mantenedor da obra:

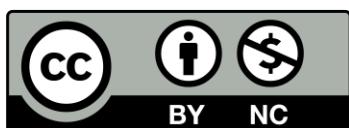
Ben Dêivide de Oliveira Batista

Contato: <ben.deivide@gmail.com>

Site pessoal: <<http://bendeivide.github.io/>>

Licença

Todos os direitos autorais contidos nesse livro são reservados ao seu conhecimento, usufrua-o, pois é totalmente de graça. Use-o com responsabilidade e saiba valorizar.



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição - Não Comercial 4.0 Internacional. Usamos também a filosofia de trabalho com o Selo Democratizando Conhecimento (DC).



Epígrafe

A melhor linguagem é a que você domina!
Ben Dêivide

Sumário

Licença	i
Epígrafe	iii
Prefácio	vii
1 Entendendo a coleção Estudando o ambiente R	1
1.1 Volume I: Nível Básico	2
1.2 Volume II: Nível Intermediário	2
1.3 Volume III: Nível Avançado	2
1.4 Demais volumes	3
1.5 Referências complementares da Coleção	3
1.6 Pacotes R utilizados para essa coleção	4
1.7 Materiais didáticos	4
Exercícios propostos	5
2 História e instalação do R	8
2.1 História do R	8
2.2 O que é o R ?	8
2.3 Instalação do R e RStudio	14
3 Como o R trabalha	16
3.1 Introdução	16
3.2 Como utilizar o R e o RStudio	17
3.3 Comandos no R	19
3.3.1 Console e <i>Prompt</i> de comando	19
3.3.2 Comandos elementares	20
3.3.3 Execução de comandos	21
3.3.4 Chamada e correção de comandos anteriores	22
3.4 Ambiente global (área de trabalho ou <i>workspace</i>)	22
3.5 Arquivos .RData e .Rhistory	23
3.6 Criando e salvando um <i>script</i>	23
Exercícios propostos	25
4 Objetos	26
4.1 Introdução	26
4.2 Atributos	28
4.2.1 Atributos intrínsecos	30
4.3 Coerção	37
4.4 Tipo de objetos	39
4.4.1 Vetores	39

4.4.1.1	Vetores escalares ou constantes	39
4.4.1.2	Vetores longos	44
4.4.1.3	Manipulando vetores	53
4.4.1.4	Aritmética e outras operações	54
4.4.1.5	Operadores lógicos	58
4.4.1.6	Operadores booleanos	60
4.4.2	Matrizes bidimensionais	61
4.4.3	Matrizes multidimensionais	63
4.4.4	Listas	65
4.4.5	Quadro de dados	68
4.4.6	Fatores	70
4.4.7	Datas e horas	70
4.4.8	Objeto para dados temporais	70
4.4.9	Objeto <i>tibble</i>	70
	Exercícios propostos	71
5	Importação e exportação de dados	72
5.1	Introdução	72
5.2	Preparação dos dados	72
5.3	Importando dados	73
5.4	Exportando dados	80
5.5	Exercícios	81
6	Funções no R	82
6.1	Introdução	82
6.2	O que é uma função no R?	82
6.3	Estrutura básica de uma função	82
6.4	Funções em pacotes	84
6.5	Chamadas de funções	85
6.6	Estruturas de controle	86
6.7	Criando funções	99
6.7.1	Função anônima	101
6.7.2	Chamadas de função	101
6.7.3	Ordenação de argumentos	102
6.7.4	Objeto reticências (“...”)	103
6.8	Escopo léxico	105
6.9	Exercícios	108
7	Gráficos no R	109
7.1	Exercícios	110
8	Boas práticas de como escrever um código	111
8.1	Introdução	111
8.2	Exercícios	112
9	Pacotes	113
9.1	Introdução	113
9.2	Estrutura básica de um pacote	114
9.3	Instalação de um pacote	114
9.4	Objetivos de um pacote	115
9.5	Utilizar as funções de um pacote	116
9.6	Carregando e anexando um pacote	119
9.7	NAMESPACE de um pacote	123

9.8 Documentações de um pacote	123
9.9 Operadores :: e ::::	123
9.10 Exercícios	125
10 Ambientes e caminho de busca	126
10.1 Introdução	126
10.2 Exercícios	127
11 Interfaces com outras linguagens	128
11.1 Introdução	128
11.2 Exercícios	129
12 Considerações e preparação para o Volume II	130
12.1 Introdução	130
12.2 Exercícios	131
13 Gabarito dos Exercícios	132
Referências Bibliográficas	145
Índice Remissivo	148

Prefácio

A coleção *Estudando o ambiente R* é fruto de cursos ministrados sobre essa linguagem, bem como consultorias e estudos ao longo dos anos. Em 2005, quando ingressei na academia no curso de Engenharia Agronômica fiquei fascinado com a disciplina de Estatística no segundo semestre do ano corrente. Na sequência, acabo tendo o primeiro contato com o ambiente R, com pouco mais de 9 anos de seu lançamento e redistribuição. Poucos materiais naquela época haviam disponíveis em língua portuguesa. Porém, foi o suficiente para eu entender que estava diante de uma grande ferramenta computacional e estatística, necessária para o entendimento, pois sabia que poderia me gerar além de conhecimento, bons frutos acadêmicos.

Hoje, no ano de 2021, usuário há mais de 15 anos dessa linguagem, percebi que me sentia desconfortável, como apenas usuário dessa ferramenta de trabalho. E assim, quando queremos aprender algo não há ferramenta melhor do que *aprender por ensinar*. E assim, lotado no Departamento de Estatística, Física e Matemática (DEFIM), campus Alto Paraopeba, pela Universidade Federal de São João del-Rei (UFSJ), juntamente com o Centro Acadêmico de Engenharia de Telecomunicações (UFSJ), resolvemos em parceria, ministrar nesse momento de pandemia uma sequência de módulos para o curso R, desde o nível Básico até ao módulo Avançado.

A ideia desse curso foi apresentar algo diferente relacionado a maioria dos cursos em R, que foi sempre apresentar essa ferramenta dentro dos conceitos da área da Estatística. Apesar de uma coisa ser intrínseca a outra, há muitas particularidades no ambiente R que são complexos, e muitas vezes julgados erroneamente. Um dos exemplos clássicos é que *loops* em R são lentos e com alto gasto de memória, quando na realidade, isso ocorre muitas vezes pelo não entendimento do sistema de cópia de objetos nesse ambiente. Ainda mais, o entendimento desses cursos é agravado porque o entendimento sobre a estatística além de um cunho matemático, tem o seu cunho filosófico de como as metodologias foram desenvolvidas, e o entendimento mútuo da Estatística e o ambiente R, podem não ter o conhecimento real que essa potencial ferramenta pode proporcionar, uma vez que muitos assuntos complexos podem estar envolvidos em uma única aula.

Assim, desenvolvemos na coleção *Estudando o ambiente R* os três volumes iniciais, referentes a apenas a linguagem R, sendo *Volume I: Nível Básico*, *Volume II: Nível Intermediário* e *Volume III: Nível Avançado*. Fazendo a alusão dos três livros iniciais sobre a linguagem S de John Chambers, faremos uma explanação sobre assuntos de menor complexidade até noções mais complexas sobre o ambiente R, restringindo apenas a sintaxe e semântica da linguagem. Os volumes subsequentes serão destinados a *Documentações no R*, *Desenvolvimento de pacote R*, *Gráficos*, *Banco de dados*, *Interface Gráfica ao Usuário*, *Interface R com outras linguagens*, **Programação Orientada a Objetos no R**, **Funções do pacote base**, dentre outros.

Tentando engajar nossos alunos, e agora colegas de trabalho, tenho a parceria no Volume I, de Diego Arthur, uma pessoa que tenta se superar a cada desafio e assunto estudado.

Por fim, espero que esse primeiro volume possa servir de referência para os passos iniciais nessa ferramenta tão importante para a área de análise de dados.

Ben Dêivide de Oliveira Batista
Ouro Branco, MG, 23 de fevereiro de 2022

Entendendo a coleção Estudando o ambiente R

A Coleção *Estudando o ambiente R* não tem como objetivo principal de ensinar análise de dados. Mas sim, proporcionar ao leitor um conhecimento sobre a linguagem **R**, de modo que se possa usufruir todos os recursos que esse ambiente possa proporcionar.

Ainda como complemento, não queremos nesse material, convencê-lo a utilizar a linguagem **R**, pois a melhor linguagem é aquela que você domina. Contudo, pretendemos mostrar que os recursos utilizados pelo **R** não estão mais limitados a própria análise de dados. Um exemplo é o material didático dessa coleção que pode ser acessada em: <<https://bendeivide.github.io/cursor/>>, que nesse momento usufruímos da própria linguagem para repassar as nossas experiências sem ao menos ter o domínio sobre linguagens do tipo HTML, CSS, JavaScript, dentre outras, necessárias para uma boa renderização de uma página *web*. Isso mostra a potencial ferramenta de trabalho que o ambiente **R** pode ser para a vida profissional.

Dessa forma, propormos um entendimento sobre a sintaxe e semântica de como a linguagem **R** é desenvolvida. Com isso, o leitor será capaz após a leitura dos dois primeiros volumes, de acompanhar qualquer curso de Estatística com aplicações em **R**, se dedicando apenas ao entendimento na área da Estatística, uma vez que o embasamento sobre o ambiente **R** foi suprido por essa coleção. Essa nova revolução do Dados, se deve ao grande volume de informações obtidos nessa era tecnológica. Juntamente com a Estatística, o **R** se tornará uma poderosa ferramenta para entender os padrões que estão por trás dos dados, que por sinal, é a moeda valiosa do momento, ou melhor, sempre foi!

Aprenderemos também recursos diversos na área da computação, como programação defensiva, desenvolvimento de interfaces gráficas, paralelização, como também recursos na área da estatística sem complexidades teóricas, como o desenvolvimento de gráficos e o uso de banco de dados. Ensinaremos também o desenvolvimento de materiais como artigos, livros, *websites*, *blogs*, *dashboards*. Por fim, chegaremos a maior cobiça de um programador **R**, desenvolver um pacote.

Por que os artigos “o” e “a” para o **R?**



Observem que em muitos momentos utilizamos o artigo “o” para a linguagem **R**. Pois é, isso ocorre porque ela também é considerada um software ou ambiente. Daí, também podemos chamá-la de programa **R**, ou preferivelmente, ambiente **R**.

Os módulos dessa coleção terão os três volumes base para o entendimento do ambiente **R**:

- Volume I: Nível Básico;
- Volume II: Nível Intermediário; e
- Volume III: Nível Avançado.

A seguir, explanaremos sobre cada um dos módulos.

1.1 Volume I: Nível Básico

Esse primeiro volume, que representa o livro corrente, apresenta um breve **histórico** sobre a linguagem, a sua instalação, bem como os recursos que a **IDE¹ RStudio**, o conhecimento da **sintaxe** e **semântica** da linguagem **R**, compreendendo as estruturas bases da linguagem, sobre o que é um **objeto** e como construir uma **função**, o entendimento sobre **fluxos de controle**. O que é um **pacote**, **carregar** e **anexar** um pacote, e quem são as pessoas que fazem parte da manutenção dessa linguagem, também serão assuntos desse primeiro módulo. **Caminho de busca, ambientes** e *namespaces*, teremos noções básicas. Algo muito interessante, que pode mudar a vida de um programador em **R** são as **boas práticas para a escrita de um código**, tema também abordado nesse módulo.

A ideia desse volume é proporcionar um entendimento básico, um primeiro contato com a linguagem, fazendo com que o leitor possa dar os primeiros passos, executando as primeiras linhas de comando. Mas também, dando o enfoque com erros tão recorrentes, como o entendimento sobre um objeto, ou o anexo de um pacote no caminho de busca. Temas como esses, dentre outros, serão a forma inicial que encontramos, para que posteriormente, seja dado um aprofundamento sobre a estrutura de um objeto **R** bem como a sua manipulação, e adicionado a isso, a inserção de como são os paradigmas da programação nesse ambiente. Essa última parte será estudada, no Volume II, apresentado a seguir.

1.2 Volume II: Nível Intermediário

O volume II é introduzido com uma maior caracterização do ambiente **R** quanto ao seu **escopo léxico**, como **linguagem interpretada**, como **programação funcional**, como **programação meta-paradigma**, como **programação dinâmica**; apresentaremos **manipulações de objetos em mais detalhe**, bem como o surgimento de alguns outros objetos como **tibble**, **cópias de objetos**. Uma característica do ambiente **R** é que a linguagem pode ser **orientada a objetos** e isso será estudado nesse módulo. Introduziremos ao **desenvolvimento de pacotes R**, e aprofundaremos sobre os **ambientes**. Por fim, mostraremos como desenvolver Projeto do **RStudio** e integrá-los ao **GitHub**, e dessa forma, introduziremos sobre o sistema **Git**.

Esse talvez seja o maior volume, dentre os três iniciais, porque apesar de não precisarmos entender mais a ideia dos objetos, que foram retratadas no Volume I, a inserção dos paradigmas da programação para este volume, trará uma maior riqueza de características para o **R**, mostrando a sua versatilidade. Também, daremos um maior detalhamento como manipular objetos, e as otimizações existentes da linguagem, como por exemplo, a modificação no local, que se entendida, poderá perceber que o **loop** no ambiente **R** não é lento quanto parece. Ao final desse volume, falaremos sobre como propagar o seu código com o sistema **Git** na plataforma **GitHub**, sincronizado com os projetos do **RStudio**.

1.3 Volume III: Nível Avançado

O Volume III, será a total exploração do manual **R Internals**. Apesar de ser um assunto voltado para membros do **R Core Team**, pretendemos entender como o **R** trabalha nos bastidores. Dessa forma, teremos total controle sobre as nossas rotinas. Contudo, para usuários que pretendem entender o ambiente **R** de forma aplicada, pode avançar esse volume para a leitura dos volumes seguintes.

Nesse volume, faremos uma introdução sobre a linguagem **C**, e entender algumas estruturas, como por exemplo ponteiros, e instruções, como por exemplo, **switch()**, e perceber que a arquitetura dos objetos em **R**, são desenvolvidas dentro dessas ideias. Também será possível usar a linguagem **C** sem necessidade de pacote adicional. Faremos também uma introdução sobre a linguagem **FORTRAN** e **S**, as duas outras linguagens complementares para o entendimento completo dos bastidores do **R**.

¹Do inglês, *Integrated Development Environment*, que significa ambiente de desenvolvimento integrado.

1.4 Demais volumes

Os demais volumes compreendem lacunas necessárias para serem abordadas com profundidade, tais como: **Documentações no R**, **Desenvolvimento de pacote R**, **Gráficos**, **Banco de dados**, **Interface Gráfica ao Usuário**, **Interface R com outras linguagens**, **Programação Orientada a Objetos no R**, **Funções do pacote base**, dentre outros.

1.5 Referências complementares da Coleção

Citaremos alguns livros e materiais utilizados para o desenvolvimento dessa coleção, que alguns podem ser acessados *online*, como também via **bookdown**, tais como:

- Manuais do **R** :
 - An Introduction to R (VENABLES; SMITH; R CORE TEAM, 2021);
 - R Data Import/Export (R CORE TEAM, 2021a);
 - R Installation and Administration (R CORE TEAM, 2021b);
 - Writing R extensions (R CORE TEAM, 2021e);
 - R language definition (R CORE TEAM, 2021d);
 - R Internals (R CORE TEAM, 2021c);
- *Bookdowns*:
 - Advanced R (WICKHAM, 2019);
 - Advanced R Solutions (GROSSER; BUMAN; WICKHAM, 2021);
 - R Packages (WICKHAM, 2015);
 - R for Data Science (WICKHAM; GROLEMUND, 2017);
 - Hands-On Programming with R (GROLEMUND, 2014);
 - R Markdown (XIE; ALLAIRE; GROLEMUND, 2018);
 - bookdown (XIE, 2017);
 - Dynamic documents with R and knitr (XIE, 2015);
 - blogdown (XIE; THOMAS; HILL, 2018);
 - Fundamentals of data visualization (WILKE, 2016);
 - R Graphics Cookbook (CHANG, 2018);
 - ggplot2 (WICKHAM, 2018);
 - Text mining with R (SILGE; ROBINSON, 2017);
 - Mastering shiny (WICKHAM, 2021).
- Livros físicos:
 - Extending R (CHAMBERS, 2016);
 - Software for data analysis: programming with R (CHAMBERS, 2008);
 - R in a Nutshell (ADLER, 2012);
 - R graphics (MURRELL, 2019);
 - The new S language (Livro branco) (BECKER; CHAMBERS; WILKS, 1988);
 - Statistical models in S (Livro azul) (CHAMBERS; HASTIE, 1991);
 - Programming with data (Livro verde) (CHAMBERS; HASTIE, 1998).

Vale salientar que esses três últimos livros, se pudéssemos unir, seria a bíblia do ambiente R.

1.6 Pacotes R utilizados para essa coleção

Apresentamos uma lista de pacotes, Tabela 1.1, utilizados ao longo da coleção para os exemplos abordados, como também para o próprio desenvolvimento dos livros.

Tabela 1.1: Pacotes a serem instalados para o acompanhamento dos exemplos e exercícios da coleção *Estudando o ambiente R*.

Pacote	Finalidade
lobstr	Estudar a sintaxe do ambiente R
codetools	Estudar a sintaxe do ambiente R
XR	Estudar a sintaxe do ambiente R
rlang	Estudar a sintaxe do ambiente R
sloop	Compreender o paradigma da programação orientada a objetos
styler	Auxilia no estilo de código
formatR	Auxilia no estilo de código
distill	Criação da página <i>web</i>
blogdown	Criação da página <i>web</i>
rmarkdown	Criação do material digital
knitr	Criação do material digital
bookdown	Criação dos livros digitais
shiny	Criação dos materiais dinâmicos
learnr	Criação dos materiais dinâmicos
ggplot2	Renderização de gráficos
pryr	Útil para o entendimento mais profundo do ambiente R
abind	Usado para combinar <i>array</i> multidimensionais

1.7 Materiais didáticos

Essa coleção é subsidiada do Curso R sempre ministrado no formato *online*, cujo suporte é auxiliado por vídeo-aulas que podem ser encontrados na própria página do curso, ou pelo canal do Youtube: <<http://youtube.com/bendeivide/>>. Além das vídeo-aulas, *scripts* são disponíveis, e para os iniciantes que ainda não fizeram a instalação do **R** e do **RStudio**, poderão também acompanhar esses exercícios via *shiny*, acessando: <<https://bendeivide.shinyapps.io/Curso-R/>>. Esperamos que esses materiais, possam complementar o aprendizado.

Exercícios propostos

Apresentamos alguns exercícios, dos quais se a resolução for de fácil entendimento, compreendendo a sua complexidade, poderá avançar para a leitura do Volume II.

Exercício 1.1: Observe o seguinte *script* abaixo:

Script:

```

1 # Criando um nome "n" associado a um objeto 10 no escopo da função
2 n <- 10
3 f1 <- function() {
4   # Imprimindo n
5   print(n)
6   # Criando um nome "n" associado a um objeto 15 no corpo da função
7   n <- 15
8   # Imprimindo n
9   print(n)
10 }
11 # Imprimindo 'f1'
12 f1(); n

```

Console:

```
[1] 10
[1] 15
[1] 10
```

Script:

```

13 # Criando um nome "f2" associado a um objeto que eh uma função
14 f2 <- function() {
15   # Imprimindo n
16   print(n)
17   # Criando um nome "n" associado a um objeto 15 no corpo da função
18   n <<- 15
19   # Imprimindo n
20   print(n)
21 }
22 # Imprimindo 'f2'
23 f2(); n

```

Console:

```
[1] 10
[1] 15
[1] 15
```

Por que existe a diferença nos resultados da chamada de f1() e f2()?

Solução na página 132

Exercício 1.2: Por que a superatribuição (<<-) deve ser usado com cautela, principalmente quando se está desenvolvendo funções para um pacote?

Solução na página 132

Exercício 1.3: Qual a diferença entre anexar e carregar um pacote?

Solução na página 132

Exercício 1.4: Qual a importância da estrutura *NAMESPACE* em um pacote?

Solução na página 132

Exercício 1.5: Por que devemos usar com cautela as funções internas não exportáveis, de um pacote? Como podemos acessá-las? Como podemos acessar uma função de um pacote sem anexá-lo ao caminho de busca? Por que esta última condição é mais interessante, quando se deseja utilizar poucas funções de um pacote?

Solução na página 132

Exercício 1.6: Quantos objetos temos na linha de comando a seguir?

Script:

```
1 x <- 10
```

Solução na página 132

Exercício 1.7: Precisamos ter o **RStudio** para utilizar o ambiente **R**?

Solução na página 132

Exercício 1.8: Quais os três princípios do **R**? Identifique-os em termos de linhas de comandos criadas em **R**.

Solução na página 132

Exercício 1.9: Podemos dizer que uma matriz ou *array* é um vetor? Se sim, o que os diferenciam?

Solução na página 132

Exercício 1.10: Podemos afirmar que um *data frame* é uma lista?

Solução na página 132

Exercício 1.11: Como podemos salvar um *script*? Qual a importância de um *script*?

Solução na página 133

Exercício 1.12: O que são funções anônimas?

Solução na página 133

Exercício 1.13: Podemos afirmar que sempre o ambiente envolvente e ambiente de chamada são ambientes iguais?

Solução na página 133

Exercício 1.14: Considerando os ambientes envolvente, de execução e de chamada, qual dos três é um ambiente temporário? E quando ele é criado?

Solução na página 133

Exercício 1.15: Baseado na linha de comando: `x <- 1`, por que não pode afirmar que “`x` recebe o valor 1”?

Solução na página 133

Exercício 1.16: Como importamos um banco de dados externo do ambiente **R**? E como exportamos um resultado desejado?

Solução na página 133

Exercício 1.17: Por que comentar um código é muito importante para um programador?

Solução na página 133

Exercício 1.18: Qual a importância de uma boa escrita de um código?

Solução na página 133

Exercício 1.19: Por que o **R** tem como um de seus princípios em sua origem, o princípio da *interface*?

Solução na página 133

Exercício 1.20: Que pacotes no **R** faz com que esse ambiente interaja com as linguagens: Python, Julia, C/C++, FORTRAN, HTML Java? Quais outras interfaces você conhece para o **R**?

Solução na página 133

Exercício 1.21: O que representa os atributos para um objeto? Quais os dois atributos intrínsecos de um objeto? Qual a importância do atributo `class` para um objeto?

Solução na página 133

Exercício 1.22: Para que serve os arquivos `.RData` e `.Rhistory`?

Solução na página 133

História e instalação do R

2.1 História do R

A linguagem **R** tem a sua primeira aparição científica publicada em 1996, com o artigo intitulado *R: A Language for Data Analysis and Graphics*, cujos os autores são os desenvolvedores da linguagem, George Ross Ihaka e Robert Clifford Gentleman.



Figura 2.1: Criadores do R.¹

Durante a época em que estes professores trabalhavam na Universidade de Auckland, Nova Zelândia, desenvolvendo uma implementação alternativa da linguagem S, desenvolvida por John Chambers, que comercialmente era o **S-PLUS**, nasceu em 1991, o projeto da linguagem **R**, em que em 1993 o projeto é divulgado e em 1995, o primeiro lançamento oficial, como software livre com a licença GNU. Devido a demanda de correções da linguagem que estava acima da capacidade de atualização em tempo real, foi criado em 1997, um grupo central voluntário, responsável por essas atualizações, o conhecido *R Development Core Team*², que hoje está em 20 membros (atualizado em 23 de fevereiro de 2022): Douglas Bates, John Chambers, Peter Delgaard, Robert Gentleman, Kurt Hornik, Ross Ihaka, Tomas Kalibera, Michael Lawrence, Friedrich Leisch, Uwe Ligges, Thomas Lumley, Martin Maechler, Sebastian Meyer, Paul Murrel, Martyn Plummer, Brian Ripley, Deepayan Sarkarm, Duncan Temple Lang, Luke Tierney e Simon Urbanek.

Por fim, o CRAN (Comprehensive R Archive Network) foi oficialmente anunciado em 23 de abril de 1997³. O CRAN é um conjunto de sites (espelhos) que transportam material idêntico, com as contribuições do **R** de uma forma geral.

2.2 O que é o R ?

R é uma linguagem de programação e ambiente de *software* livre e código aberto (*open source*). Entendemos⁴:

¹Fonte das fotos: Robert Gentleman do site: <<https://biocasia2020.bioconductor.org/>> e Ross Ihaka do site: <<https://stat.auckland.ac.nz/>>

²Fontes: <https://www.cran.r-project.org/doc/html/interface98-paper/paper_2.html> e <<https://www.r-project.org/contributors.html>>

³Fonte: <<https://stat.ethz.ch/pipermail/r-announce/1997/000001.html>>

⁴Fonte: <<https://www.gnu.org/philosophy/free-sw.html>>

- **Software livre:** software que respeita a liberdade e sendo de comunidade dos usuários, isto é, os usuários possuem a liberdade de executar, copiar, distribuir, estudar, mudar, melhorar o software. Ainda reforça que um software é livre se os seus usuários possuem quatro liberdades:

1. Liberdade 0 - A liberdade de executar o programa como você desejar, para qualquer propósito;
2. Liberdade 1 - A liberdade de estudar como o programa funciona, e adaptá-la as suas necessidades;
3. Liberdade 2 - A liberdade de redistribuir cópias de modo que você possa ajudar outros;
4. Liberdade 3 - A liberdade de distribuir cópias de suas versões modificadas a outros.

Algo que deve está claro é que um software livre não significa não comercial. Sem esse fim, o software livre não atingiria seus objetivos. Agora perceba que, segundo Richard Stallman⁵, a ideia de software livre faz campanha pela liberdade para os usuários da computação. Por outro lado, o código aberto valoriza principalmente a vantagem prática e não faz campanha por princípios.

- **Código aberto:** Para Richard Stallman⁶ código aberto apoia critérios um pouco mais flexíveis que os do software livre. Todos os códigos abertos de software livre lançados se qualificariam como código aberto. Quase todos os softwares de código aberto são software livre, mas há exceções, como algumas licenças de código aberto que são restritivas demais, de forma que elas não se qualificam como licenças livres. Nesse contexto, o autor cita muitas situações que diferenciam os dois termos. Vale a pena a leitura.

A linguagem **R** é uma combinação da linguagem **S** com a semântica de escopo léxico da linguagem Scheme. Dessa forma, a linguagem **R** se diferencia em dois aspectos principais⁷:

- **Gerenciamento de memória:** usando as próprias palavras de Ross Ihaka⁸, em **R**, alocamos uma quantidade fixa de memória na inicialização e a gerenciamos com um coletor de lixo dinâmico. Isso significa que há muito pouco crescimento de *heap* e, como resultado, há menos problemas de paginação do que os vistos na linguagem **S**.
- **Escopo:** na linguagem **R**, as funções acessam os objetos criadas no corpo da própria função, como também os objetos contidos no ambiente que a função foi criada. No caso da linguagem **S**, isso não ocorre, assim, como por exemplo na linguagem C, em que as funções acessam apenas variáveis definidas globalmente.

Escopo léxico



Estude atentamente os exemplos a seguir, porque o escopo léxico é uma das grandes características que existe no ambiente **R**, e que por exemplo, não existe na linguagem **S**. O escopo léxico dá uma grande flexibilidade as funções para a procura dos objetos, ou de um modo menos formal, em busca das variáveis inseridas no corpo da função criada. Se o leitor for um iniciante, sugerimos seguir na leitura até o Capítulo 4, e posteriormente, retornar a esses exemplos.

Vejamos alguns exemplos para entendimento (Se você ainda não está ambientado ao **R**, estude esse módulo primeiro, e depois reflita sobre esses exemplos). Antes de executar as linhas de comando, instale o pacote **lobstr** como segue:

⁵Fonte: <https://www.gnu.org/philosophy/open-source-misses-the-point.html>

⁶Fonte: <<https://www.gnu.org/philosophy/open-source-misses-the-point.html>>

⁷Fonte: <https://cran.r-project.org/doc/html/interface98-paper/paper_1.html>

⁸Fonte: <https://cran.r-project.org/doc/html/interface98-paper/paper_1.html>

Script:

```
1 # Instale o pacote lobstr
2 install.packages("lobstr")
```

- Exemplo 1: As funções têm acesso ao escopo em que foram criadas, Código R 2.1.

Código R 2.1**Script:**

```
1 # Criando um nome "n" associado a um objeto 10 no escopo da função
2 n <- 10
3 # Criando um nome "funcao" associado a um objeto que eh uma função
4 funcao <- function() {
5   print(n)
6 }
7 # Imprimindo 'funcao'
8 funcao()
```

Console:

```
[1] 10
```

- Exemplo 2: As variáveis criadas ou alteradas dentro de uma função, permanecem na função, Código R 2.2.

Código R 2.2**Script:**

```
1 # Criando um nome "n" associado a um objeto 10 no escopo da função
2 n <- 10
3 lobstr::obj_addr(n) # Identificador do objeto
```

Console:

```
[1] "0x26abd3d8"
```

Script:

```
4 # Criando um nome "funcao" associado a um objeto que eh uma função
5 funcao <- function() {
6   # Imprimindo n
7   print(n)
8   # Criando um nome "n" associado a um objeto 15 no corpo da função
9   n <- 15
10  # Imprimindo n
11  print(n)
12 }
13 # Imprimindo 'funcao'
14 funcao()
```

Console:

```
[1] 10
[1] 15
```

Script:

```
15 # Imprimindo 'n'
16 n
```

Console:

```
[1] 10
```

Script:

```
17 lobstr::obj_addr(n) # Identificador do objeto
```

Console:

```
[1] "0x26abd3d8"
```

- Exemplo 3: As variáveis dentro de uma função permanecem nelas, exceto no caso em que a atribuição ao escopo seja explicitamente solicitada, Código R 2.3.

Código R 2.3**Script:**

```
1 # Criando um nome "n" associado a um objeto 10 no escopo da funcao
2 n <- 10
3 lobstr::obj_addr(n) # Identificador do objeto
```

Console:

```
[1] "0x24ab4308"
```

Script:

```
4 # Criando um nome "funcao" associado a um objeto que eh uma funcao
5 funcao <- function() {
6   # Imprimindo n
7   print(n)
8   # Criando um nome "n" associado a um objeto 15 no corpo da funcao
9   n <- 15
10  # Imprimindo n
11  print(n)
12 }
13 # Imprimindo 'funcao'
14 funcao()
```

Console:

```
[1] 10
[1] 15
```

Script:

```
15 # Observe que depois de usar a superatribuição ("<-") dentro da função,
16 # o nome "n" passou a estar associado ao número 15 e não mais ao número
17   10, observe
17 n
```

Console:

```
[1] 15
```

Script:

```
18 lobjstr::obj_addr(n) # Identificador do objeto
```

Console:

```
[1] "0x24ab4228"
```

- Exemplo 4: Por fim, embora a linguagem R tenha um escopo padrão, chamado ambiente global, os escopos de funções podem ser alterados, Código R n2.4.

Código R 2.4**Script:**

```
1 # Criando um nome 'n' associado a um objeto 10 no escopo da função
  (ambiente global)
2 n <- 10
3 # Criando um nome 'funcao' associado a um objeto que é uma função
  criada no ambiente global
4 funcao <- function() {
5   # Imprimindo n
6   print(n)
7 }
8 # Imprimindo 'funcao' no ambiente global
9 funcao()
```

Console:

```
[1] 10
```

Script:

```

10 # Criando um novo ambiente
11 novo_ambiente <- new.env()
12 # Criando um nome "n" associado ao objeto 20 no ambiente 'novo_ambiente'
13 novo_ambiente$n <- 20
14 # Criando um objeto função no ambiente 'novo_ambiente'
15 environment(funcao) <- novo_ambiente
16 # Imprimindo 'funcao' no ambiente 'novo_ambiente'
17 funcao()

```

Console:

```
[1] 20
```

Como a linguagem é também uma linguagem interpretada cuja base é a linguagem FORTRAN, a linguagem **R** também é uma linguagem interpretada e baseada além da linguagem *S*, tem como base as linguagens de baixo nível C e FORTRAN e a própria linguagem **R**.

Embora o **R** tenha uma interface baseada em linhas de comando, existem muitas interfaces gráficas ao usuário com destaque ao **R**, criado por Joseph J. Allaire, Figura 2.2.



Figura 2.2: J. J. Allaire, o criador do **RStudio**.⁹

Essa *interface* tornou o **R** mais popular, pois além de produzir pacotes de grande utilização hoje como a família de pacotes **tidyverse**, **rmarkdown**, **shiny**, dentre outros, permite uma eficiente capacidade de trabalho de análise de utilização do **R**. Uma vez que o **RStudio** facilita a utilização de muitos recursos por meio de botões, como por exemplo, a criação de um pacote **R**. A quem diga que para um iniciante em **R**, não seja recomendado utilizar o **RStudio** para o entendimento da linguagem. Cremos, que o problema não é a IDE¹⁰ utilizada, e sim, o caminho onde se deseja progredir com a linguagem **R**.

No Brasil, o primeiro espelho do CRAN foi criado na UFPR, pelo grupo do Prof. Paulo Justiniano. Inclusive um dos primeiros materiais mais completos sobre a linguagem **R** produzidos no Brasil, foi dele, iniciado em 2005, intitulado Introdução ao Ambiente Estatístico **R**. Vale a pena assistirmos o evento a palestra: *R Releflões: um pouco de história e experiências com o R*, proferida pelo Prof. Paulo Justiniano, no *R Day - Encontro nacional de usuários do R*, ocorrido em 2018 em Curitiba/UFPR, do qual o vídeo está disponível no Canal (Youtube) LEG UFPR.

⁹Fonte da foto: <<https://rstudio.com/speakers/j.j.-allaire/>>

¹⁰Do inglês, *Integrated Development Environment*, que significa Ambiente de Desenvolvimento Integrado, como por exemplo, o **RStudio**, Emacs, dentre outros, para o **R**.

2.3 Instalação do R e RStudio

Para realizarmos a instalação do ambiente **R**, uma vez que o **RStudio** é apenas uma *IDE*, e sem o **R**, não há sentido instalá-lo, seguimos os seguintes passos:

- Instalação do **R** - <<https://www.r-project.org>>, Figuras 2.3 e 2.4:

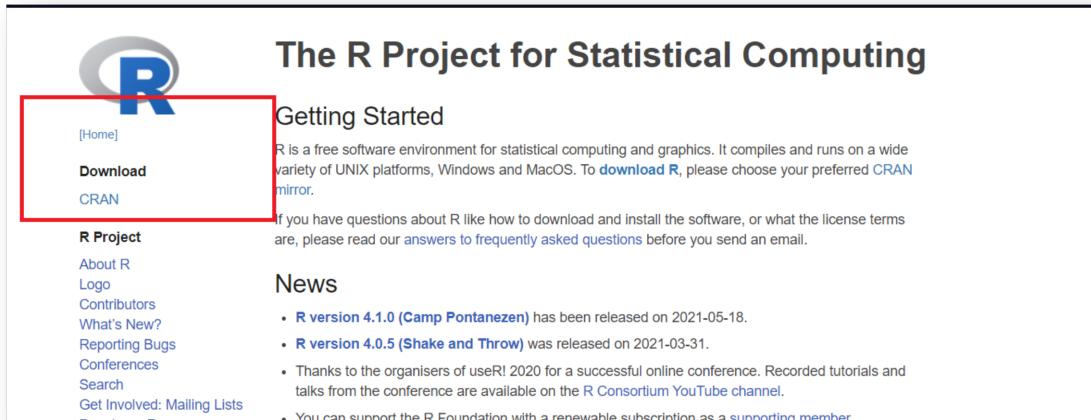


Figura 2.3: Primeiro passo para Instalação do R.

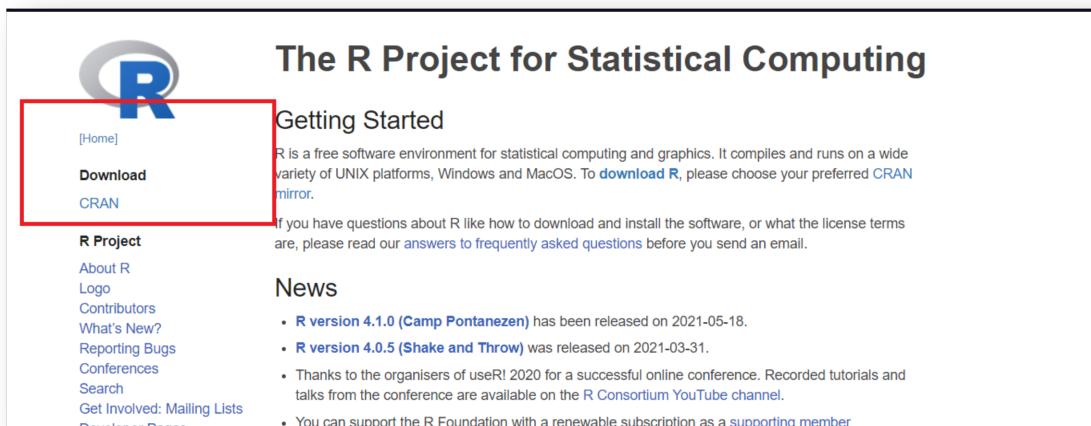


Figura 2.4: Segundo passo para Instalação do R.

- Instalação do **RStudio** - <<https://rstudio.com/products/rstudio/download/#download>>:

Justificamos a utilização do **RStudio** pela quantidade de recursos disponíveis e a diversidade de usuários **R**, que hoje o perfil não é apenas de um programador, mas de um usuário que necessita de uma ferramenta estatística para análise de seus dados. Dessa forma, até por questão de praticidade, e de uso pessoal, não deixaremos de repassar o entendimento sobre a linguagem **R** com o uso do **RStudio**.

Outra coisa importante, é que esses passos para a instalação do **R** e **RStudio** se basearam no sistema operacional *Windows*, mas para detalhes sobre essas instalações em outros sistemas operacionais, acesse: <<https://bendeivide.github.io/cursor>>.

All Installers

Linux users may need to [import RStudio's public code-signing key](#) prior to installation, depending on the operating system's security policy.
RStudio requires a 64-bit operating system. If you are on a 32 bit system, you can use an [older version of RStudio](#).

OS	Download	Size	SHA-256
Windows 10	 RStudio-1.4.1717.exe	156.18 MB	71b36e64
macOS 10.14+	 RStudio-1.4.1717.dmg	203.06 MB	2cf2549d
Ubuntu 18/Debian 10	 rstudio-1.4.1717-amd64.deb	122.51 MB	e27b2645
Fedora 19/Red Hat 7	 rstudio-1.4.1717-x86_64.rpm	138.42 MB	648e2be0
Fedora 28/Red Hat 8	 rstudio-1.4.1717-x86_64.rpm	138.39 MB	c76f620a
Debian 9	 rstudio-1.4.1717-amd64.deb	123.29 MB	e4ea3a60
OpenSUSE 15	 rstudio-1.4.1717-x86_64.rpm	123.15 MB	e69d55db

Figura 2.5: Instalação do RStudio.

Como o **R** trabalha

3.1 Introdução

Para entendermos como o **R** trabalha, iniciamos com uma afirmação de John McKinley Chambers, do qual afirmou que o **R** tem três princípios (CHAMBERS, 2016):



Figura 3.1: John Chambers¹, o criador da linguagem S.

Princípios do R



- **Princípio do Objeto:** Tudo que existe em **R** é um objeto;
- **Princípio da Função:** Tudo que acontece no **R** é uma chamada de função;
- **Princípio da Interface:** Interfaces para outros programas são parte do **R**.

Ao longo de todo o curso, para os três módulos, iremos nos referir a esses princípios. Vamos inicialmente observar uma adaptação da ilustração feita por Paradis (2005), mostrando como o **R** trabalha, Figura 3.2.

Toda ação que acontece no **R** é uma chamada de função (Operadores e funções), que por sua vez é armazenada na forma de um objeto, e este se associa a um nome. A forma de execução de uma função é baseada em argumentos (dados, fórmulas, expressões, etc), que são entradas, ou argumentos padrões que já são pré-estabelecidos na criação da função. Esses tipos de argumentos podem ser modificados na execução da função. Por fim, a saída é o resultado, que é também um objeto, e pode ser usado como argumento de outras funções.

¹Fonte da foto: Retirada de sua página pessoal, <<https://statweb.stanford.edu/~jmc4/>>.

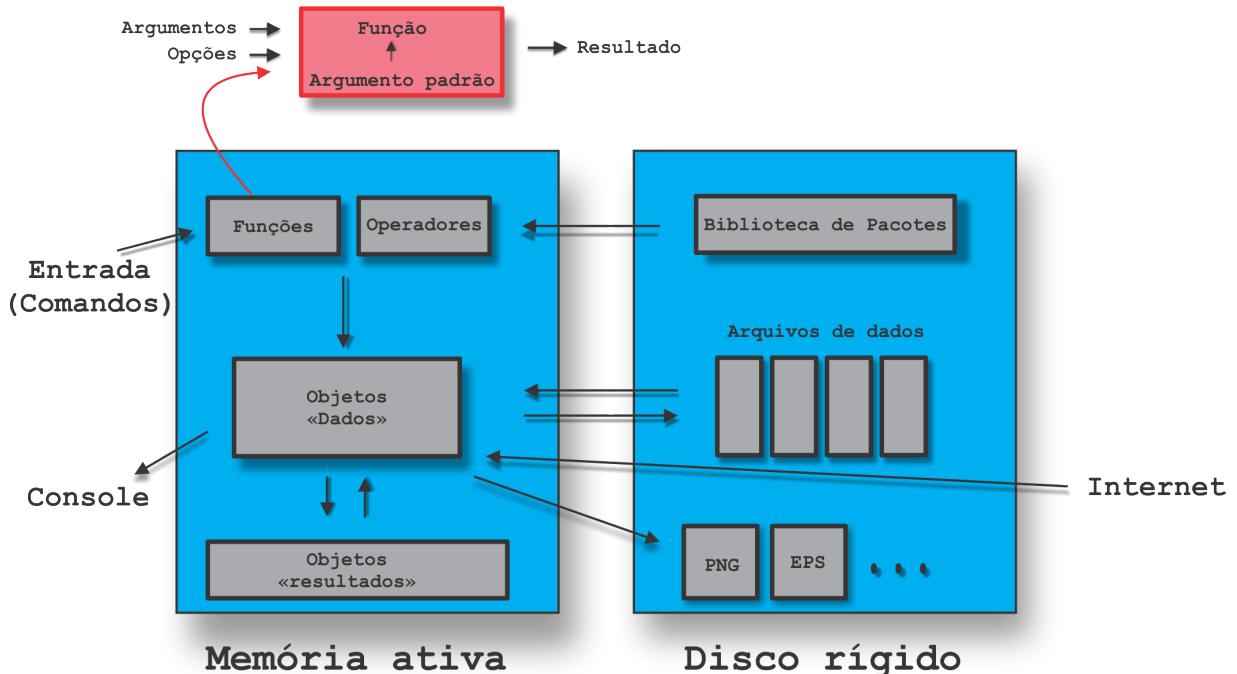


Figura 3.2: Esquema de como o R funciona.

Na Figura 3.2, observamos que todas as ações realizadas sobre os objetos ficam armazenadas na memória ativa do computador. Esses objetos são criados por comandos (teclado ou mouse) através de funções ou operadores (chamada de função), dos quais leem ou escrevem arquivo de dados do disco rígido, ou leem da própria internet. Por fim, o resultado desses objetos podem ser apresentados no console (memória ativa), exportados em formato de imagem, página web, etc. (disco rígido), ou até mesmo ser reaproveitado como argumento de outras funções, porque o resultado também é um objeto.

3.2 Como utilizar o R e o RStudio

A primeira ideia que temos é a linha de comando no R, que é simbolizada pelo *prompt* de comando ">". Este símbolo significa que o R está pronto para receber os comandos do usuário. O *prompt* de comando está localizado no console do R. Vejamos o console do R a seguir, que é o local que recebe as linhas de comando do usuário, Figura 3.3.

O R ao ser iniciado está pronto para ser inserido as linhas de comando desejadas. Uma forma simples de armazenar os seus comandos é por meio de um *Script*, isto é, um arquivo de texto com extensão .R. Para criar basta ir em: Arquivo > Novo script.... Muitas outras informações iremos ver ao longo do curso.

O RStudio se apresenta como uma interface para facilitar a utilização do R, tendo por padrão quatro quadrantes, apresentados na Figura 3.4.

Muitas coisas na interface do R podem se tornar problemas para os usuários, uma vez que janelas gráficas, janelas de *scripts*, dentre outras, se sobrepõe. Uma vantagem no RStudio foi essa divisão de quadrantes, que torna muito mais organizado as atividades realizadas no R. De um modo geral, diremos que o primeiro quadrante é responsável pela entrada de dados, comandos, isto é, o *input*. O segundo quadrante, que é o console do R, representa tanto entrada como saída de informações (*input/output*). Dependendo as atividades as abas podem aumentar. O terceiro quadrante representa informações básicas como objetos no ambiente global, a memória de comandos na aba *History*, dentre outras, e também representa entrada como saída de informações (*input/output*). Por fim, o quarto quadrante é responsável por representação gráficas, instalação de pacotes, renderização de páginas web.

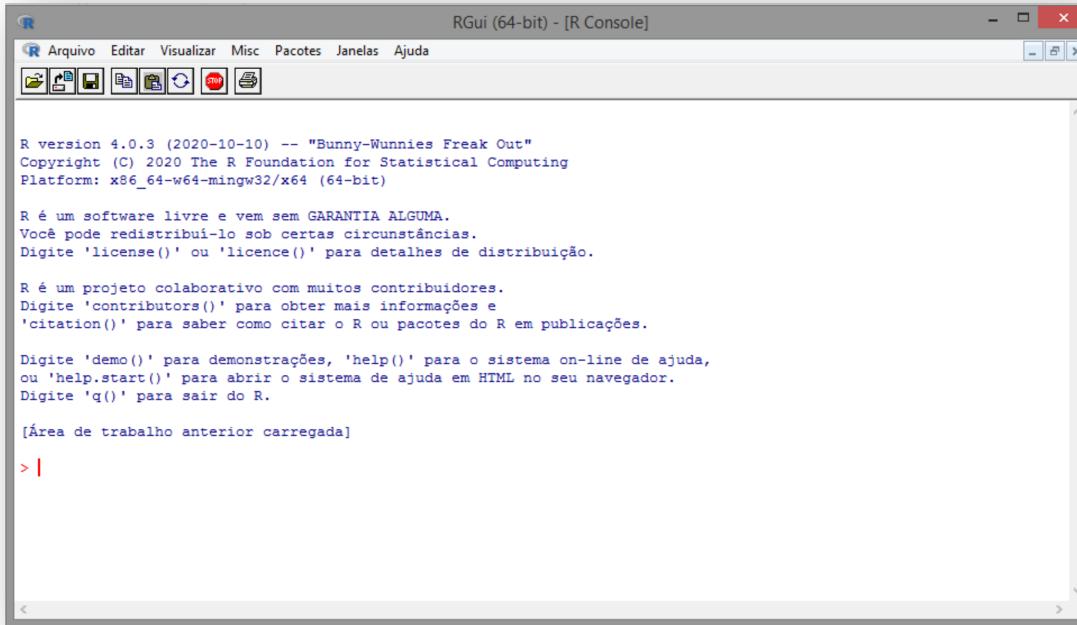


Figura 3.3: Console do R (Versão 4.0.3).

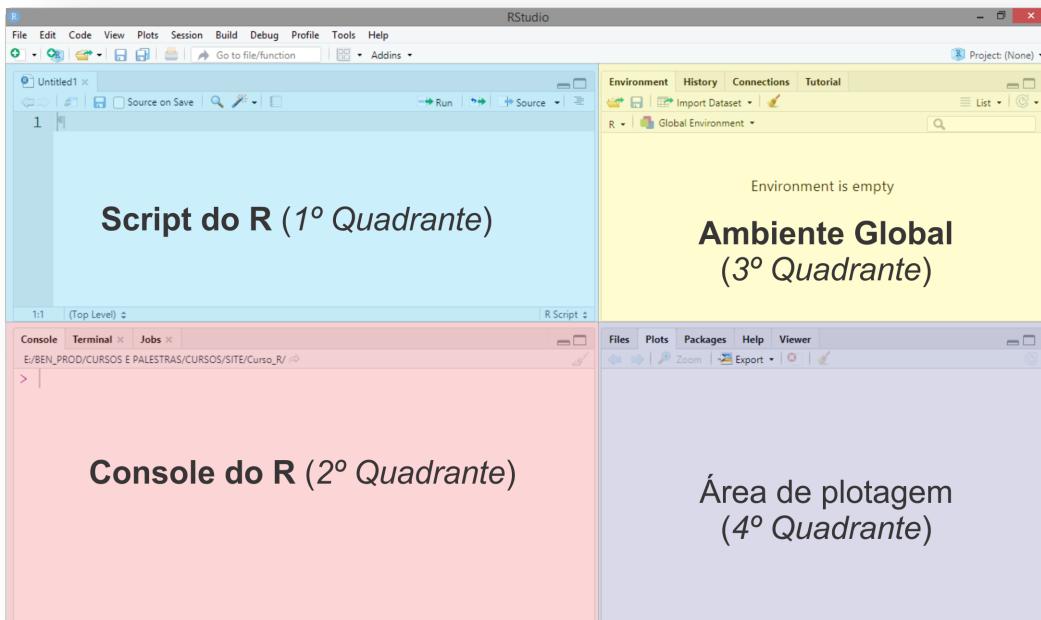


Figura 3.4: Interface do RStudio (Versão 1.4.1103).

3.3 Comandos no R

3.3.1 Console e *Prompt* de comando

Como falado anteriormente, o **R** é uma linguagem baseada em linhas de comando, e as linhas de comando, são executadas uma de cada vez, no *console*. Assim que o *prompt* de comando está visível na tela do *console*, o **R** indica que o usuário está pronto para inserir as linhas de comando. O símbolo padrão do *prompt* de comando é ">", porém ele pode ser alterado. Para isso, vejamos o exemplo do Código R 3.1.

Código R 3.1

Script:

```
1 options(prompt = 'R>')
2 # Toda vez que o console iniciar, começará por 'R>'
3 10
```

Console:

```
R> # Toda vez que o console iniciar, começará por 'R>'
[1] 10
```

O conjunto de símbolos que podem ser utilizados no **R** depende do sistema operacional e do país em que o **R** está sendo executado. Basicamente, todos os símbolos alfanuméricos podem ser utilizados, mas para evitar problemas quanto ao uso das letras aos nomes, opte pelos caracteres *ASCII*.

A escolha do nome associado a um objeto tem algumas regras:

- Deve consistir em letras, dígitos, “.” e “_”;
- Os nomes devem ser iniciado por uma letra ou um ponto não seguido de um número, isto é, Ex.: .123, 1n, dentre outros;
- As letras maiúsculas se distinguem das letras minúsculas;
- Não pode iniciar por “_” ou dígito, é retornado um erro no console caso isso ocorra;
- Não pode usar qualquer uma das palavras reservadas pela linguagem, isto é, TRUE, FALSE, if, for, dentre outras, que pode ser consultado usando o comando ?Reserved().

Um nome que não segue essas regras é chamado de um nome **não sintático**. Um comando que pode ser usado para converter nomes não sintáticos em nomes **sintáticos** é make.names.

Console:

```
> # Nome não sintático
> .123 <- 50
> ## Error in 0.123 <- 50 : lado esquerdo da atribuição inválida (do_set)
> # Qual a sugestão de nome sintático para '.123'?
> make.names(.123)
[1] "X0.123"
```

Apesar dessas justificativas, algumas situações como as apresentadas nos exemplos anteriores são possíveis, ver Wickham (2019) na Seção 2.2.1.

3.3.2 Comandos elementares

Os **comandos elementares** podem ser divididos em **expressões** e **atribuições**. Por exemplo, podemos estar interessados em resolver a seguinte expressão $10 + 15 = 25$. Vejamos o Código R 3.2.

Código R 3.2

Script:

```
1 10 + 15
```

Console:

```
[1] 25
```

No *console* quando passamos pelo comando do Código R 3.2, o **R** avalia essa expressão internamente e imprime o resultado na tela, após apertar o botão *ENTER* do teclado. Esse fato é o que ocorre no segundo princípio mencionado por Chambers (2016), tudo em **R** acontece por uma chamada de função. Na realidade o símbolo `+` é uma função interna do **R**, que chamamos de função primitiva, porque foi implementada em outra linguagem. Assim, esse é o resultado de três objetos (“`10`”, “`+`”, “`15`”) que são avaliados internamente, do qual a função ‘`+(e1, e2)`’ é chamada, e em seguida o resultado é impresso no *console*. Intrinsecamente, podemos também afirmar que a função `print()` também trabalha nessa situação, fazendo o papel de imprimir o resultado no *console*.

Do mesmo modo, se houver algum problema em algum dos objetos o retorno da avaliação pode ser uma mensagem de erro. Um caso muito prático é quando utilizamos o separador de casas decimais para os números sendo a vírgula. Quando na realidade deve ser um ponto “`.`”, respeitando o sistema internacional de medidas que são definidas por padrão no ambiente **R**, e essas configurações podem ser alteradas por meio de `options()`. A vírgula é utilizada para separar elementos, argumentos em uma função, etc. Vejamos um exemplo no Código R 3.3.

Código R 3.3

Script:

```
1 10,5 + 15,5
```

Console:

```
Error: <text>:1:3: ',' inesperado
```

Porém, tem que ficar claro que uma expressão é qualquer comando repassado no *console*. Este comando é avaliado e seu resultado impresso, há menos que explicitamente o usuário queira torná-lo invisível². Caso algum elemento do comando não seja reconhecido pelo **R**, há um retorno de alguma mensagem em forma de “erro” ou “alerta”, tentando indicar o possível problema. Todos esses processos ocorrem na memória ativa do computador, e uma vez o resultado impresso no *console*, o valor é perdido, há menos que você atribua essa expressão a um nome, que erroneamente usamos o termo: “criamos um objeto!”. A atribuição dessa expressão será dada pela junção de dois símbolos `<-`, falado mais a frente. Um comando em forma de atribuição também avalia a sua expressão, um nome se associa ao seu resultado, e o resultado será mostrado, se posteriormente, após a execução você digitar o “nome” atribuído a esse resultado. Vejamos um exemplo o Código R 3.4.

²Basta usar a função `invisible(10 + 15)`, que a expressão é avaliada mas não impressa.

Código R 3.4**Script:**

```

1 Foi criado um objeto do tipo caractere e o nome "meu_nome" foi associado a ele
2 # O 'R' avalia essa expressão, mas não imprime no console!
3 meu_nome <- "Ben"
4 # Para imprimir o resultado da expressão, digitamos o nome "meu_nome" no
   console
5 # e apertamos o botão ENTER do teclado!
6 meu_nome

```

Console:

```
[1] "Ben"
```

3.3.3 Execução de comandos

Quando inserimos um comando no console, executamos uma linha de comando por vez ou separados por ";" em uma mesma linha. Vejamos o Código R 3.5.

Código R 3.5**Script:**

```

1 # Uma linha de comando por vez
2 meu_nome <- "Ben" # Criamos e associamos um nome ao objeto
3 meu_nome # Imprimos o objeto

```

Console:

```
[1] "Ben"
```

Script:

```

1 # Tudo em uma linha de comando
2 meu_nome <- "Ben"; meu_nome

```

Console:

```
[1] "Ben"
```

Se um comando for muito grande e não couber em uma linha, ou caso deseje completar um comando em mais de uma linha, após a primeira linha haverá o símbolo "+" iniciando a linha seguinte ao invés do símbolo de prompt de comando (">"), até que o comando esteja sintaticamente completo. Vejamos o Código R 3.6, a seguir.

Código R 3.6**Script:**

```
1 # Uma linha de comando em mais de uma linha
2 (10 + 10) /
3 2
```

Console:

```
> # Uma linha de comando em mais de uma linha
> (10 + 10) /
+ 2
[1] 10
```

Por fim, todas linhas de comando quando iniciam pelo símbolo jogo da velha, “#” indica um comentário e essa linha de comando não é avaliada pelo console, apenas impressa na tela. E ainda, as linhas de comandos no console são limitadas a aproximadamente 4095 *bytes* (não caracteres).

3.3.4 Chamada e correção de comandos anteriores

Uma vez que um comando foi executado no console, esse comando por ser recuperado usando as teclas de setas para cima e para baixo do teclado, recuperando os comandos anteriormente executados, e que os caracteres podem ser alterados usando as teclas esquerda e direita do teclado, removidas com o botão Delete ou *Backspace* do teclado, ou acrescentadas digitando os caracteres necessários. Uma outra forma de completar determinados comandos já existentes, como por exemplo, uma função que já existe nas bibliotecas de instalação do R , usando o botão *Tab* do teclado. O usuário começa digitando as iniciais, e para completar o nome aperta a tecla *Tab*. Posteriormente, basta completar a linha de comando e apertar *ENTER* para executá-la. Para entender mais detalhes, acesse o *link*: <<https://youtu.be/0MRPmVsPvk4>>, e veja em vídeo-aula mais detalhes.

Esses recursos no **RStudio** são mais dinâmicos e vão mais além. Por exemplo, quando usamos um objeto do tipo função, estes apresentam o que chamamos de argumento(s) dentro do parêntese de uma função, do qual são elementos necessários, para que a função seja executada corretamente. Nesse caso, ao inserir o nome dessas funções no console, usando o **RStudio** , ao iniciá-la com a abertura do parêntese, abre-se uma janela informando todos os argumentos possíveis dessa função. Isso torna muito dinâmico escrever linhas de comando, porque não precisaremos estar lembrando do nome dos argumentos de uma função, mas apenas entender o objetivo dessa função. Para entender mais detalhes, acesse o *link*: <https://youtu.be/KL3WAB_GFNI>, e veja em vídeo-aula mais detalhes.

3.4 Ambiente global (área de trabalho ou *workspace*)

Quando usamos um comando de atribuição no console, o R armazena o nome associado ao objeto criado na área de trabalho (*Workspace*), que nós chamamos de Ambiente Global. Teremos uma seção introdutória na seção Ambientes e caminhos de busca, mas entendamos inicialmente que o objetivo de um ambiente é associar um conjunto de nomes a um conjunto de valores. Vejamos o Código R 3.7.

Código R 3.7**Script:**

```

1 # Nomes criados no ambiente
2 x <- 10 - 6; y <- 10 + 4; w <- "Maria_Isabel"
3 # Verificando os nomes contidos no ambiente global
4 ls()

```

Console:

```

[1] "cran"          "funcao"        "github"        "meu_nome"
[5] "n"             "novo_ambiente" "rlink"         "rstudio"
[9] "w"             "x"              "y"

```

Observemos que todos os objetos criados até o momento estão listados, e o que é mais surpreendente é que ambientes podem conter outros ambientes e até mesmo se conterem. Observe o objeto `meu_nome` é um ambiente e está contido no Ambiente global. Será sempre dessa forma que recuperaremos um objeto criado no console do **R**. Caso contrário, se no console esse comando não for de atribuição esse objeto é perdido.

3.5 Arquivos .RData e .Rhistory

Ao final do que falamos até agora, todo o processo ao inserir linhas de comando do console, e desejarmos finalizar os trabalhos do ambiente **R**, dois arquivos são criados, sob a instrução do usuário em querer aceitar ou não, um `.RData` e outro `.Rhistory`, cujas finalidades são:

- `.RData`: salvar todos os objetos criados que estão atualmente disponíveis;
- `.Rhistory`: salvar todas as linhas de comandos inseridas no console.

Ao iniciar o **R** no mesmo diretório onde esses arquivos foram salvos, é carregado toda a sua área de trabalho anteriormente, bem como o histórico das linhas de comando utilizadas anteriormente.

3.6 Criando e salvando um *script*

A melhor forma de armazenarmos nossas linhas de código inseridas no console é criando um *Script*. Este é um arquivo de texto com a extensão `.R`. Uma vez criada, poderemos ao final salvar o arquivo e guardá-lo para utilizar futuramente.

No **R**, ao ser iniciado poderemos ir no menu em Arquivo > Novo script.... Posteriormente, pode ser inserido as linhas de comando, executadas no console pela tecla de atalho F5. As janelas do *Script* e console possivelmente ficarão sobrepostas. Para uma melhor utilização, estas janelas podem ficar lado a lado, configurando-as no menu em Janelas > Dividir na horizontal (ou Dividir lado a lado).

No **RStudio**, poderemos criar um *Script* no menu em File > New File > R Script, ou diretamente no ícone abaixo da opção File no menu, cujo o símbolo é um arquivo com o símbolo "+" em verde, que é o ícone do New File, e escolher R Script. Esse arquivo abrirá no primeiro quadrante na interface do **RStudio**.

Para salvar, devemos clicar no botão com o símbolo de disquete (R/RStudio), escolher o nome do arquivo e o diretório onde o arquivo será armazenado no seu computador. Algumas ressalvas devem ser feitas:

- Escolha sempre um nome sem caracteres especiais, com acentos, etc.;

- Escolha sempre um nome curto ou abreviado, que identifique a finalidade das linhas de comando escritas;
- Evite espaços se o nome do arquivo for composto. Para isso, use o símbolo *underline* "_";
- Quando escrever um código, evite também escrever caracteres especiais, exceto em casos de necessidade, como imprimir um texto na tela, títulos na criação de gráficos, dentre outras. Nos referimos especificamente, nos comentários do código.

Um ponto bem interessante é o diretório. Quando criamos um *Script* a primeira vez, e trabalhamos nele a pós a criação, muitos erros podem ser encontrados de início. Um problema clássico é a importação de dados. O usuário tem um conjunto de dados e deseja fazer a importação para o **R**, porém, mesmo com todos os comandos corretos, o console retorna um erro, informando que não existe esse arquivo que contém os dados para serem informados. Isso é devido ao diretório de trabalho atual. Para verificar qual o diretório que está trabalhando no momento, use a linha de comando:

Script:

```
1 getwd()
```

Para alterar o diretório de trabalho, o usuário deve usar a seguinte função `setwd("Aqui, deve ser apontado para o local desejado!")`. Supomos que salvamos o nosso *Script* em `C:\meus_scripts_r`. Assim, usamos a função `setwd()` e ao apontarmos o local, as barras devem ser inseridas de modo invertido, isto é, `setwd("C:/meu_scripts_r")`, além de estar entre aspas.

No **RStudio**, isso pode ser feito em Session > Set Working Directory > To Source File Location. Isso levará ao diretório correto do *Script*. Se desejar escolher outro diretório, vá em Session > Set Working Directory > Choose Directory.... Porém, uma vez criado um *Script*, e utilizado novamente, se o usuário estiver abrindo o **RStudio** também naquele primeiro momento, por padrão, o diretório de trabalho correto será o mesmo do diretório do *Script*. Isso acaba otimizando o trabalho.

Devemos nos atentar também, quando trabalhamos utilizando *Scripts* ou arquivos de banco de dados, em locais diferentes do diretório correto. Um outro recurso interessante é a função `source()`, que tem o objetivo de executar todas as linhas de comando de um *Script* sem precisar abri-lo. Isso pode ser útil, quando criamos funções para as nossas atividades, porém elas não se encontram no *Script* de trabalho para o momento. Assim, podemos criar um *Script* auxiliar que armazena todas as funções criadas para as análises desejadas, e no *Script* correto, poderemos chamá-las sem precisar abri o *Script* auxiliar. Todos os objetos passam a estar disponíveis no ambiente global.

Por fim, algo de muita importância para um programador e usuário de linguagem, **comente suas linhas de comando**. Mas faça isso a partir do primeiro dia em que foi desenvolvido o primeiro *Script*. Isso criará um hábito, uma vez que o arquivo não está sendo criado apenas para um momento, mas para futuras consultas. E quando voltamos a *Scripts* com muitas linhas de comando, principalmente depois de algum tempo, e sem comentários, possivelmente você passará alguns instantes para tentar entender o que foi escrito.

Outra coisa importante, é a **boa prática de escrita de um código**, Capítulo 6, e o **RStudio** nos proporciona algumas ferramentas interessantes. Mas isso será visto mais a frente.

Exercícios propostos

Exercício 3.1:

Solução na página 135

Objetos

4.1 Introdução

Definimos um objeto como uma entidade no ambiente R com características internas contendo informações necessárias para interpretar sua estrutura e conteúdo. Essas características são chamadas de **atributos**. Vamos entender o termo estrutura como a disposição de como está o seu conteúdo. Por exemplo, a estrutura de um objeto mais simples no R é um **vetor atômico**, pois os elementos contidos nele, apresenta o mesmo **modo**, um tipo de atributo. Falaremos disso, mais à frente. De forma didática, adaptaremos a representação dos objetos no formato de diagrama. Vejamos a seguinte linha de comando:

Script:

```
1 x <- 10
```

Todo mundo que tem uma certa noção sobre a linguagem **R** afirmaria: “criei um objeto x que recebe o valor 10”. Para Wickham (2019) essa afirmação é imprecisa e pode levar um entendimento equivocado sobre o que acontece de fato. Para o mesmo autor, o correto é afirmar que o objeto 10 está se ligando a um nome. E de fato, o objeto não tem um nome, mas o nome tem um objeto. O símbolo que associa um objeto a um nome é o de atribuição, `<-`, isto é, a junção do símbolo desigualdade menor e o símbolo de menos. Para ver qual objeto associado ao nome, o usuário precisa apenas digitar o nome no console e apertar a tecla *ENTER*. Representaremos em termos de diagrama, um nome se ligando a um objeto, na Figura 4.1.



Figura 4.1: Dizemos que o nome x se liga ao objeto do tipo (estrutura) vetor.

Queremos chamar atenção na Figura 4.1, quando associamos o nome x ao objeto 10. Nesse caso, a flecha aponta no sentido contrário ao símbolo de atribuição para reforçar que o nome se liga ao objeto e não o contrário. O identificador na memória ativa desse objeto pode ser obtida por:

Console:

```
> lobstr::obj_addr(x)
[1] "0xf8a104fc20"
```

O diagrama explica que o nome criado x se associou com um objeto do **tipo** (estrutura) vetor (**vector**) e **modo** numérico (**numeric**)¹, cuja identificação na memória ativa do seu computador foi `<0xf8a104fc20>`. É claro que para cada vez que o usuário abre o ambiente **R** e executar novamente esse comando, ou repetir o comando, esse identificador irá alterar.

¹ou também **double**, usando a função `typeof()`.

Em outra representação, Código R ??, ficará mais claro para a afirmação feita anteriormente, no segundo diagrama, Figura 4.2, que representa a ligação do nome *y* ao mesmo objeto. Os termos nos diagramas, serão usados de acordo com a sintaxe da linguagem usados em inglês para melhor compreensão e fixação, uma vez que os termos na linguagem são baseados nesse idioma.



Observem que não houve a criação de um outro objeto, mas apenas a ligação de mais um nome ao objeto existente, pois o identificador na memória ativa para o objeto não alterou, é o mesmo. Logo, não temos um outro objeto, mas dois nomes que se ligam ao mesmo objeto.

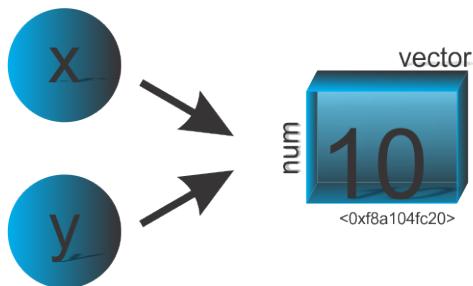


Figura 4.2: Dizemos que o nome *x* e *y* se ligam ao objeto do tipo (estrutura) vetor.

Mais especificamente, acrescentamos um outro diagrama, Figura 4.3, mostrando a representação do ambiente global (*.GlobalEnv*, nome associado ao objeto que representa o ambiente global).

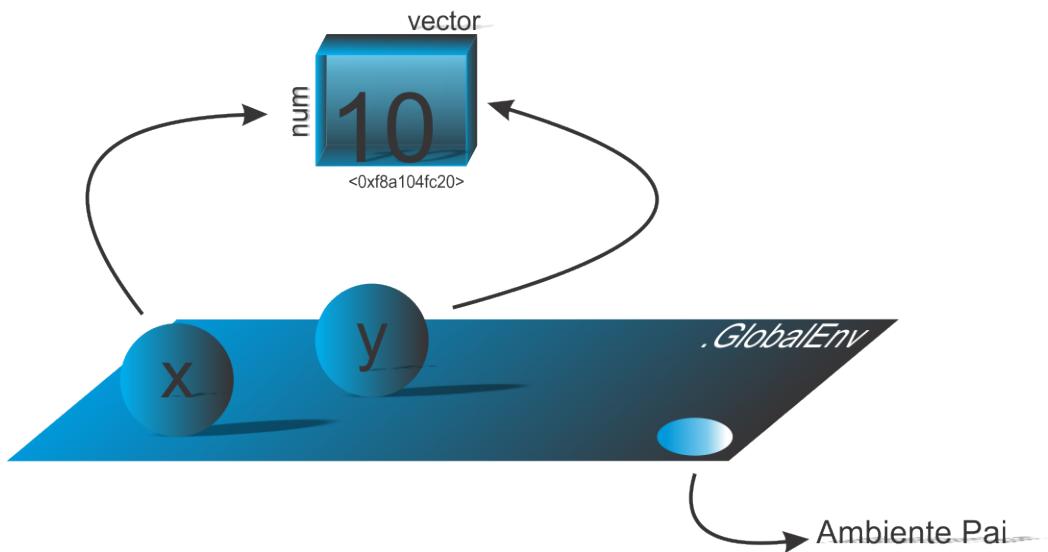


Figura 4.3: Dizemos que o nome *x* e *y* se ligam ao objeto do tipo (estrutura) vetor e essa ligação fica armazenada no ambiente global.

De todo modo, deixaremos para o Volume II, uma abordagem mais profunda sobre o assunto. O

símbolo de atribuição poderá ser representado na direção da esquerda para à direita ou vice-versa, isto é,

Script:

```
1 x <- 10
2 10 -> x
```

Essas duas linhas de comando anteriores podem ter passado despercebidas pelo leitor em uma situação. Se na segunda linha tivéssemos alterado o valor do objeto de 10 para 30, por exemplo, a associação de x seria ao objeto 30. Isso significa que se o nome já existe, ele será apagado da memória ativa do computador e associado ao novo objeto², como observado na sequência,

Console:

```
> lobstr::obj_addr(x)
[1] "0xf8a104fc20"
> x <- 30
> lobstr::obj_addr(x)
[1] "0x42db6dbb50"
```

Uma outra forma menos convencional é usar a função `assign()`, isto é,

Console:

```
> assign("m", 15)
> m
[1] 15
```

Ao invés do símbolo de atribuição, muitos usuários utilizam o símbolo da igualdade “=” para associarmos nomes aos objetos, que o ambiente **R** compreenderá. Contudo, discutiremos mais adiante, Capítulo 6, que o uso da igualdade deverá em **R** ser usado apenas para a utilização em argumentos de uma função.

Quando desejamos executar mais de uma linha de comando por vez, separamos estas pelo símbolo “;”, isto é,

Console:

```
> x <- 10; w <- 15; x; w
[1] 10
[1] 15
```

Neste caso, executamos quatro comandos em uma linha. Associamos dois nomes a dois objetos e imprimimos os seus valores.

Por questão de comodidade, iremos a partir de agora, sempre nos referir a um objeto pelo nome associado a ele, para não estar sempre se expressando como “um nome associado a um objeto”. Mas que fique claro a discussão realizada anteriormente sobre esses conceitos.

Nesse momento, nos limitaremos a falar sobre objetos que armazenam dados, do tipo caracteres, números e operadores lógicos (TRUE/FALSE).

4.2 Atributos

Todos os objetos, terão pelo menos dois tipos de atributos, chamados de atributos intrínsecos. Os demais atributos, quando existem, podem ser verificados pela função `attributes()`. A ideia dos

²Na realidade o coletor de lixo se encarrega de eliminar objetos em desuso.

atributos pode ser pensada como *metadados*, isto é, um conjunto de informações que caracterizam o objeto.

Diremos também que todos os objetos **R** tem uma **classe**, e por meio dessas classes, determinadas funções podem ter comportamento diferente a objetos com classes diferentes. Agora, devemos deixar claro essa informação, apesar do **R** seguir o **princípio do objeto**, nem tudo é orientado a objetos, como por exemplo, observamos na linguagens C++ e Java. Deixemos esse tópico para discorrer no Volume II.

A forma de se verificar a classe de um objeto é pela função `class()`. Contudo, os objetos internos do **R** (base), quando solicitado sua classe pela função `class()`, acabam retornando, algumas vezes, resultados equivocados. Uma alternativa é utilizar a função `sloop::s3_class()` do pacote **sloop**. Isso também será discutido no Volume II.

Devemos nos atentar a uma questão: **existe um atributo também chamado classe (class)**, e nem todos os objetos necessariamente tem esse atributo, apenas aqueles orientados a objetos, como é o caso do objeto com atributo classe. Por exemplo, é devido a classe `factor` no objeto criado pela função `factor()` que apesar do seu resultado ser numérico, este não se comporta como numérico. Isto significa que o atributo classe muda o comportamento de como funções veem esse objeto. Entretanto, mesmo os objetos que não apresentam esse atributo, quando pedimos pela chamada `class()` desse referido objeto, haverá o retorno do que chamamos de **classe implícita**, que nada mais é do que a tipagem do objeto baseado no atributo modo (`mode()` ou `typeof()`). A Classe implícita não é definida pelo atributo `class`, mas pela tipagem do objeto. Isso também será abordado no Volume II.

Para verificarmos se tal objeto tem o atributo `class`, usamos a função `attributes()`. Quando este atributo existe, ele é coincidente com o resultado obtido também pela função `class()`.

O tipo da classe implícita pode ser `numeric`, `logical`, `character`, `list`, `matrix`, `array`. Outros objetos apresentam classes definidas pelo atributo `class`, como `factor`, `data.frame`, dentre outros.

Para remover o efeito do atributo `class`, usamos a função `unclass()` para tal.

Por exemplo, quando criamos um objeto da classe `data.frame`, vejamos o que acontece quando removemos esse atributo no Código R 4.2.

Código R 4.2

Script:

```
1 # Criamos um objeto de classe 'data.frame'
2 dados <- data.frame(a = 1:3, b = LETTERS[1:3])
3 # Imprimindo na tela
4 dados
```

Console:

```
a b
1 1 A
2 2 B
3 3 C
```

Script:

```
5 # Verificando sua classe
6 class(dados)
```

Console:

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

Script:

```
7 # Verificando o efeito do objeto 'dados',
8 # sem o efeito da classe
9 dados2 <- unclass(dados); dados2
```

Console:

```
$a
[1] 1 2 3

$b
[1] "A" "B" "C"

attr(,"row.names")
[1] 1 2 3
```

Script:

```
10 # Qual a classe desse objeto sem o efeito da
11 # classe 'data.frame'
12 class(dados2)
```

Console:

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

Observe que sem o atributo `class= 'data.frame'`, o objeto tem classe `list`. Isto significa que, o objeto tem uma estrutura em forma de `list`, mas se comporta como um `data.frame`, que se apresenta como mostrado anteriormente.

Veremos no Volume II como criar atributos, classes, e mostrar que não conseguiremos mostrar todos os tipos de classes, pois a todo momento se cria classes em objetos **R** no desenvolvimento de pacotes.

4.2.1 Atributos intrínsecos

Todos os objetos tem dois *atributos intrínsecos*: o **modo** e **comprimento**. O **modo** representa a natureza dos elementos objetos. Para o caso dos vetores atômicos, o **modo** dos vetores podem ser cinco, numérico (`numeric`), lógico (`logic`), caractere³ (`character`), complexo (`complex`) ou bruto (`raw`). Este último, não daremos evidência para esse momento, lembrando que essa tipagem está relacionada a linguagem S. O **comprimento** mede a quantidade de elementos no objeto.

Para determinarmos o **modo** de um objeto, usamos a função `mode()`. Vejamos alguns exemplos pelo Código R 4.3.


Código R 4.3
Script:

```
1 # Objeto modo caractere
2 x <- "Ben"; mode(x)
```

³sinônimo: `string`, cadeia de caracteres.

Console:

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

Script:

```
3 # Objeto modo numerico
4 y <- 10L; mode(y)
```

Console:

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

Script:

```
5 # Objeto modo numerico
6 y2 <- 10; mode(y2)
```

Console:

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

Script:

```
7 # Objeto modo logico
8 z <- TRUE; mode(z)
```

Console:

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

Script:

```
9 # Objeto modo complexo
10 w <- 1i; mode(w)
```

Console:

```
[1] "complex"
```

Contudo, essa função `mode()` se baseou nos atributos baseados na linguagem S. Temos uma outra função para verificarmos o **modo** do objeto que é por `typeof()`. O atributo **modo** retornado de um objeto para esta última função, está relacionado a tipagem da linguagem C, Código R 4.4, uma vez que boa parte das rotinas no **R** está nessa linguagem, principalmente as funções do pacote **base**. Existem 24 tipos que serão detalhados no Volume II.

Código R 4.4**Script:**

```
1 # Objeto modo caractere
2 x <- "Ben"; typeof(x)
```

Console:

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

Script:

```
3 # Objeto modo numerico (Inteiro)
4 y <- 10L; typeof(y)
```

Console:

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

Script:

```
5 # Objeto modo numerico (Real)
6 y2 <- 10; typeof(y2)
```

Console:

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

Script:

```
7 # Objeto modo logico
8 z <- TRUE; typeof(z)
```

Console:

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

Script:

```
9 # Objeto modo complexo
10 w <- 1i; typeof(w)
```

Console:

```
[1] "complex"
```

Observamos que apesar de alguns vetores serem vazios, estes ainda tem um modo, observe nas seguintes linhas de comando, no Código R 4.5.

Código R 4.5**Script:**

```
1 # Vetor numérico vazio de comprimento 1
2 numeric(0)
```

Console:

```
numeric(0)
```

Script:

```
3 # Verificando o seu modo
4 mode(numeric(0))
5 typeof(numeric(0))
```

Console:

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

Script:

```
5 # Vetor caractere vazio de comprimento 1
6 character(0)
```

Console:

```
character(0)
```

Script:

```
7 # Verificando o seu modo
8 mode(character(0))
9 typeof(character(0))
```

Console:

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

A diferença existente nos objetos `y` e `y2` para as funções `mode()` e `typeof()` se referem apenas como o **R** armazena essas informações na memória do computador. Podemos perguntar ao **R** se dois números são iguais, assim:

Console:

```
> # 10 eh igual a 10L ?
> 10 == 10L
[1] TRUE
```

Veja que o resultado é `TRUE`, isto é, sim eles são iguais. Agora, veja a próxima linha de comando:

Console:

```
> # 10 eh identico a 10L ?
> identical(10, 10L)
[1] FALSE
```

O retorno agora foi FALSE, que significa que o armazenamento dessas informações não são iguais. Posteriormente, entenderemos no que isso reflete no código do usuário, uma vez que um código escrito pode apresentar uma perda de desempenho simplesmente pela não necessidade de determinados objetos serem copiados.

O termo double retornado pela função `typeof()` significa dupla precisão na linguagem de programação, que acaba tendo uma exigência de mais memória do que o objeto de modo `integer`. Esses termos são utilizados na linguagem C. Já a linguagem S não os diferencia, utiliza tudo como `numeric`.

Aqui vale um destaque para o termo *numérico*, que no R podem ter três significados:

- Pode significar um número real, isto é, para a computação um número de dupla precisão (`numeric` e `double` seriam iguais nesse aspecto), Código R 4.6;

Código R 4.6**Script:**

```
1 # Criacao de dois objetos de modo numeric
2 a <- numeric(1); b <- double(1)
3 # Verificando o modo
4 mode(a); mode(b)
```

Console:

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

Script:

```
5 # Verificando se 'a' e 'b' sao identicos
6 identical(a, b)
```

Console:

```
[1] TRUE
```

- nos sistemas S3 e S4 (orientação a objetos), o termo numérico é usado como atalho para o modo `integer` ou `double`. Esse ponto veremos Volume II. Contudo, vejamos o Código R 4.7;

Código R 4.7**Script:**

```
1 sloop::s3_class(1)
```

Console:

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

Script:

```
5 sloop::s3_class(1L)
```

Console:

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

- Pode ser utilizado (`is.numeric()`) para verificar se determinados objetos tem o modo numérico. Por exemplo, temos um objeto de classe factor que é importante para a área da estatística experimental, representando os níveis de um fator em um experimento. Os elementos desse objeto pode ser número ou caracteres, mas serão representados como sempre por números. Entretanto, não se comportam como numérico, Código R 4.8;

Código R 4.8**Script:**

```
1 # Criando um objeto de atributo classe 'factor':
2 fator <- factor("a"); fator
```

Console:

```
[1] a
Levels: a
```

Script:

```
3 # O atributo classe muda a forma dos elementos. Veja quando retiramos o
    # atributo
4 # classe 'factor', o objeto retorna o valor 1
5 unclass(fator)
```

Console:

```
[1] 1
attr(,"levels")
[1] "a"
```

Script:

```
5 # Para confirmar essa afirmação anterior, vejamos o modo
6 mode(fator); typeof(fator)
```

Console:

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

Script:

```
7 # Apesar do resultado retornar 1, veja que ele nao se comporta como
   numerico
8 is.numeric(fator); is.integer(fator)
```

Console:

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

A Tabela 4.1 a seguir, mostra o retorno dos seis principais modos de um objeto do tipo (estrutura) de vetores atômicos (Os modos apresentados baseiam-se apenas quanto a característica dos dados do objeto. É claro que um objeto não armazena apenas dados. Existem outras naturezas, que serão omitidas nesse momento).

Tabela 4.1: Tipagem dos vetores.

<code>typeof()</code>	<code>mode()</code>
<code>logical</code>	<code>logical</code>
<code>integer</code>	<code>numeric</code>
<code>double</code>	<code>numeric</code>
<code>complex</code>	<code>complex</code>
<code>character</code>	<code>character</code>
<code>raw</code>	<code>raw</code>

O **comprimento** do objeto é informado pela função `length()`, do qual a representação em diagrama informa esse atributo. Vejamos as linhas de comando no Código R 4.9.

Código R 4.9**Script:**

```
1 # Vetor de comprimento 5
2 v1 <- 1:5
3 # Vetor de comprimento 3
4 v2 <- c("Ben", "Maria", "Lana")
5 # Vetor de comprimento quatro
6 v3 <- c(TRUE, FALSE, TRUE, TRUE)
7 # Vejamos o comprimento dos vetores
8 length(v1); length(v2); length(v3)
```

Console:

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

Um diagrama apresentando esses três objetos no ambiente global, pode ser apresentado na Figura 4.4. Observe que acrescentamos agora o **comprimento** dos objetos no diagrama entre colchetes, ao lado do atributo **modo**.

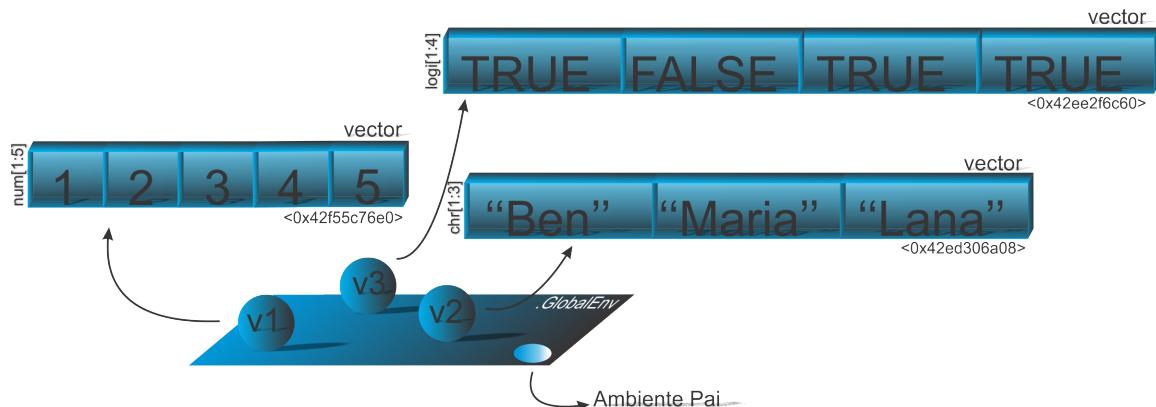


Figura 4.4: Objetos v1, v2 e v3.

Um resumo as funções mencionadas podem ser refletidas com as seguintes indagações:

- `base::class()` e `loop::c3_class()`: Qual o tipo de objeto?
- `base::mode()`: Qual o tipo de dados baseados na linguagem S?
- `base::typeof()`: Qual o tipo de dados baseados na linguagem C?
- `base::attributes()`: O objeto tem atributos?
- `base::length()`: Qual o comprimento do objeto?

Usamos essa sintaxe `pacote::nome_função()` para entendermos qual o pacote da função que utilizamos. Contudo, essa forma tem uma importância no sentido de acesso a funções em um pacote sem necessitar anexá-lo no caminho de busca. Assunto abordado mais a frente.

4.3 Coerção

Como falamos anteriormente, os vetores atômicos armazenam um conjunto de elementos de mesmo **modo**. A coerção é a forma como o **R** coage o **modo** dos objetos. Por exemplo, se um elemento de modo caractere estiver em um vetor, todos os demais elementos serão convertidos para esse modo. Vejamos a linha de comando, a seguir.

Console:

```
> # Criando um objeto x e imprimindo o seu resultado
> x <- c("Nome", 3, 4, 5);x
[1] "Nome" "3"    "4"    "5"
```

Observe que todos os elementos ganharam aspas, isto é, se tornaram um caractere ou uma cadeia de caracteres. A coerção entre vetores de modo numeric, character e logical será sempre como verificado pela Figura 4.5.

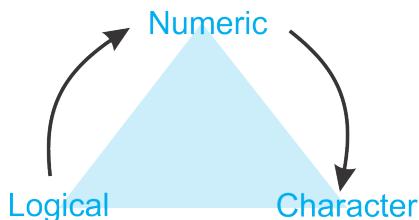


Figura 4.5: Coerção de vetores com tipagem numeric, character e logical.

No caso dos vetores lógicos, todo TRUE se converterá em 1, e FALSE em 0. Porém, os modos dos vetores podem ser coagidos pelo usuário, usando as funções do tipo “as.<modo ou tipo>()” com prefixo “as.”, isto é, se desejarmos que um objeto meu_objeto tenha o modo “character”, basta usar as.character(meu_objeto). Para desejar saber se um objeto é de um determinado modo, usamos as funções do tipo is.<modo ou tipo>(), com o prefixo “is.”. Vejamos o Código R 4.10, para elucidar o que discutimos anteriormente.

Código R 4.10

Script:

```

1 # Objeto de modo numerico
2 minha_idade <- 35
3 mode(minha_idade)

```

Console:

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

Script:

```

1 # Coercão do objeto para modo caractere (`string`)
2 minha_idade <- as.character(minha_idade)
3 mode(minha_idade)

```

Console:

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

Script:

```

1 # Verificando se o objeto tem modo 'character'
2 is.character(minha_idade)

```

Console:

```
[1] TRUE
```

4.4 Tipo de objetos

Por fim, pretendemos falar sobre os principais tipos de objetos. O **tipo** vamos entender como a estrutura de como os dados estão organizados em um objeto, relacionados aos seus atributos. Falamos anteriormente sobre a estrutura mais simples, que é o vetor atômico. Mas entendemos que um vetor em **R** podem ser considerados: atômicos ou listas. Podemos então subdividi-los em:

- Vetores atômicos:
 - Lógicos, Numéricos e Caracteres;
 - Matrizes unidimensionais (*Matrix*) e multidimensionais (*Arrays*);
- Vetores em listas:
 - Listas (*Lists*);
 - Quadro de dados (*Data frames*);

Existem outros, mas para esse módulo, exploraremos estes nas seções seguintes. As funções para as coerções realizadas pelos usuários, são similares as funções de coerção para modo, isto é, usar as funções prefixadas as .<modo>.

Daremos uma visão geral dos objetos apresentados até o momento na Tabela 4.2.

4.4.1 Vetores

Podemos dizer que existem três tipos principais de vetores atômicos:

- Numéricos (`numeric`):
- Inteiro (`integer`);
- Real (termo matemático) ou dupla precisão (termo computacional) (`double`);
- Lógico (`logical`);
- Caractere (`character`)

Existem dois tipos raros que são os complexos (`complex`) e brutos (`raw`), que falaremos no Volume II.

4.4.1.1 Vetores escalares ou constantes

O menor comprimento de um vetor é de tamanho um, conhecido também como um escalar. Porém, para o **R** tudo é observado como um vetor. As sintaxes para os tipos especiais são:

- os vetores lógicos assumem valores: TRUE ou FALSE, ou abreviados, T ou F, respectivamente. Existem valores especiais devido a precisão de operações na programação, que são os chamados pontos flutuantes. Nesse caso temos: Inf, -Inf e NaN, quando o resultado tende a ∞ , $-\infty$, sem número, respectivamente;
- os vetores numéricos do tipo ‘double’ podem ser representados de forma decimal (0.123), científica (1.23e5), ou hexadecimal (3E0A);
- os vetores numéricos do tipo `integer` são representados pela letra L ao final do **número inteiro**, isto é, 1L, 1.23e5L, etc.;
- os caracteres são representados pelas palavras, letras, números ou caracteres especiais entre aspas, isto é, 'Ben', 'a'. Pode ser utilizado também aspa simples, 'Ben', 'a', etc.

Tabela 4.2: Caracterização de objetos estruturado para armazenamento de dados.

Objeto	Classe	Modo	São possíveis vários modos no mesmo objeto?
Vetor		numeric (integer ou double) character, complex, logical, raw	numeric (integer ou double), character, complex, logical, raw
Matriz	matrix	numeric (integer ou double) character, complex, logical, raw	Não
<i>Array</i>	array	numeric (integer ou double) character, complex, logical, raw, expression, function	Não
Lista	list	numeric (integer ou double) character, complex, logical, raw	Sim
Quadro de dados	data.frame	numeric (integer ou double) character, complex, logical, raw	Sim

Qual a diferença entre NaN, NA e NULL?

As palavras reservadas `NaN` (do inglês, *Not a Number*) e `NA` (do inglês, *Not Available*) representam valores ausentes, com uma grande diferença, `NaN` apesar do nome é um tipo de valor numérico não representável na aritmética de pontos flutuantes (GOLDBERG, 1991). Esse termo foi introduzido por em 1985 pelo Instituto de Engenheiros Elétricos e Eletrônicos (IEEE), para definir as representações de outras quantidades não finitas, como por exemplo os infinitos. No **R**, para verificarmos se um número é finito ou infinito, usamos respectivamente, `is.finite()` e `is.infinite()`^a. O primeiro retorna todos os não infinitos e não ausentes. Contudo, `NA` se refere a qualquer valor não ausente, que não necessariamente seja numérico. Como podemos comprovar isso, pela pirâmide de coerção na Figura 4.5. Vejamos o Código R 4.11. Percebemos no primeiro caso que `NaN` tem um comportamento de número, com modo `double`^b, então é coagido a caractere, e perde a sua natureza de número. Já `NA` é do tipo lógico, portanto, pode ser um valor ausente para qualquer natureza de vetor, seja `numeric`, `logical` ou `character`, e nesse último caso, não ocorre coerção, apenas a informações de que o primeiro elemento do vetor está ausente, porém, o vetor ainda continua sendo de modo `character`.

Desse modo, nós vamos perceber `NaN` em operações matemáticas que a solução é indeterminada, tais como pode ser visto no Código R `??`. Porém, quando um conjunto de dados é apresentado e desejamos representar um valor ausente, é preferível `NA`. Operações realizadas com `NaN` ou `NA`, retornam `NaN` ou `NA` na maioria das vezes, como pode ser visto no Código R 4.12. Agora observemos como é interessante a ideia criada para essas palavras reservadas. Sabemos que qualquer valor de potência zero é sempre igual a 1, seja um número positivo ou negativo. Então essa ideia também segue ao ambiente **R**, pois apesar de não sabermos o valor ausente, esse valor elevado a zero será 1, como pode ser verificado no Código R 4.13.

Para identificar esses valores usamos `is.nan()` e `is.na()`. Apesar de muito parecidos `is.nan(NA)` retorna `FALSE`, de modo que, se usarmos o operador booleano `==` ou `identical()`, o resultado também será `FALSE`. Isso ocorre porque `NA` tem muitas variações, que serão vistas mais a frente.

Por fim, apresentamos a distinção do objeto `NULL`, isso mesmo um objeto de tipo `NULL`, mais precisamente `NILSXPc`, e também uma palavra reservada. Diferentemente de `NaN` e `NA` que é um vetor de comprimento 1. Nos manuais do **R**, é dito que o objeto `NULL` aparece sempre em funções ou expressões cujos resultados são não definidos. Esse objeto é o único no **R** que não tem atributo, sendo muito usado em argumentos padrão em funções quando inicialmente não se define nada para o referido argumento. Para verificar se um objeto é `NULL`, usamos `is.null()` e coagimos por `as.null()`. Agora, apresentamos uma diferença básica entre `NULL`, `NaN` e `NA`, em que estes últimos quando definidos em um objeto, apesar do valor ausente ou perdido, é sabido que existe o valor. Assim, na contabilização do número de elementos do vetor, por exemplo, `NaN` ou `NA` é contabilizado. No caso, `NULL` representa valor não existente, e como não há atributo envolvido, este não é contabilizado. Observe no Código R 4.15.

^aNo **R** há uma palavra reservada para o infinito, `Inf` e `-Inf`.

^bIsso pode ser verificado usando `.Internal(inspect(NaN))`, detalhes no Volume II.

^cO objeto `NULL` será mais explorado no Volume II.



Código R 4.11

Script:

```
1 c(NaN, "a")
```

Console:

```
[1] "NaN" "a"
```

Script:

```
2 c(NA, "a")
```

Console:

```
[1] NA "a"
```

Código R 4.12

Script:

```
1 sqrt(-1)
```

Console:

```
[1] NaN  
Warning message:  
In sqrt(-1) : NaNs produced
```

Script:

```
2 0 / 0
```

Console:

```
[1] NaN
```

Script:

```
3 Inf - Inf
```

Console:

```
[1] NaN
```

Código R 4.13**Script:**1 NA⁰**Console:**

[1] 1

Script:2 NaN⁰**Console:**

[1] 1

Código R 4.14**Script:**

1 NA + 1

Console:

[1] NA

Script:

2 NaN + 5

Console:

[1] NaN

Script:

3 NA * 5

Console:

[1] NA

Script:

4 sqrt(NaN)

Console:

[1] NaN

Script:

```
5  NaN + NA
```

Console:

```
[1] NaN
```

Script:

```
6  NA + NaN
```

Console:

```
[1] NA
```

Código R 4.15**Script:**

```
1 length(c(NA, 1, 2))
```

Console:

```
[1] 3
```

Script:

```
2 length(c(NaN, 1, 2))
```

Console:

```
[1] 3
```

Script:

```
3 length(c(NULL, 1, 2))
```

Console:

```
[1] 2
```

4.4.1.2 Vetores longos

Os vetores longos podem ser criados pela função `c()`, a inicial da palavra concatenar (do inglês, *concatenate*), que significa agrupar. Vejamos um primeiro exemplo no Código R 4.16.

Código R 4.16**Script:**

```
1 # Criando um vetor 'double'
2 vetor.num <- c(1, 2, 3, 4, 5); vetor.num
```

Console:

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

Script:

```
1 typeof(vetor.num)
```

Console:

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

Uma coisa interessante é que por padrão, a função `c()` sempre cria um vetor de modo `double`, a menos que o usuário determine que estes elementos sejam inteiros, como pode ser visto no Código R 4.17.

Código R 4.17**Script:**

```
1 # Criando um vetor 'integer'
2 vetor.num2 <- c(1L, 2L, 3L, 4L, 5L); vetor.num
```

Console:

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

Script:

```
1 typeof(vetor.num2)
```

Console:

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

Uma forma mais eficiente para criarmos um vetor com elementos de sequências regulares, é por meio da função primitiva `(:)`, isto é, `<menor valor da sequência>:<maior valor da sequência>`, isto é,

Console:

```
> # Criando uma sequência de 1 a 5
> vetor.num3 <- 1:5; vetor.num3; typeof(vetor.num3)
[1] 1 2 3 4 5
[1] "integer"
```

Veremos mais a frente outras funções para construir sequências regulares. Se verificarmos os três objetos, veremos que todos eles são iguais:

Console:

```
> vetor.num == vetor.num2
[1] TRUE TRUE TRUE TRUE TRUE
> vetor.num == vetor.num3
[1] TRUE TRUE TRUE TRUE TRUE
> vetor.num2 == vetor.num3
[1] TRUE TRUE TRUE TRUE TRUE
```

O que vai diferenciá-los é a forma de armazená-lo (double ou integer), e por consequência, o espaço na memória ativa, como podemos observar no Código R 4.18.

Código R 4.18

Script:

```
1 # Objetos:
2 vetor.num <- c(1, 2, 3, 4, 5)
3 vetor.num2 <- c(1L, 2L, 3L, 4L, 5L)
4 vetor.num3 <- 1:5
5 # Memória:
6 lobstr::obj_size(vetor.num)
```

Console:

```
96 B
```

Script:

```
6 lobstr::obj_size(vetor.num2)
```

Console:

```
96 B
```

Script:

```
7 lobstr::obj_size(vetor.num3)
```

Console:

```
96 B
```

O que podemos observar é que o vetor de modo double precisa de mais memória para armazenar os valores do que o objeto de modo integer. O último objeto, gerado pela chamada ‘:’(), aparentemente ocupa mais memória. Porém, essa função apresenta um recurso interessante apresentado nas versões posteriores **R** (3.5.0), que é chamado de **abreviação alternativa**. Esse recurso faz com que a sequência de números não seja armazenada completamente, apenas os extremos. Isso significa que para qualquer tamanho de sequência, a ocupação de memória do objeto será sempre a mesma. Lembrando que essa sequência sempre terá o modo double na tipagem C. Outras formas de criar sequências de números é usando as funções `rep()`, `rep_len()` (mais rápido), `seq()`, `seq_along()`

(mais rápido) e `seq_len()` (mais rápido), `sequence()`, `replicate()`, `gl()`, e que pode ser observado no Código R 4.19.

Código R 4.19

Script:

```
1 # Repete o numero 2 tres vezes
2 rep(x = 2, times = 3)
```

Console:

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

Script:

```
3 # Repete o vetor 1:3 tres vezes
4 rep(x = 1:3, times = 3)
```

Console:

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

Script:

```
5 # Repete cada numero do vetor, tres vezes
6 rep(x = 1:3, each = 3)
```

Console:

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

Script:

```
7 # Repete cada numero do vetor duas vezes,
8 # porem, o comprimento dessa sequencia esta
9 # limitado a 4
10 rep(1:3, each = 2, length.out = 4)
```

Console:

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

Script:

```
11 # O vetor eh repetido ate obter uma sequencia de tamanho 7
12 rep(x = 1:3, length.out = 7)
```

Console:

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

Script:

```
13 # (Versao mais rapida de rep) O vetor eh repetido ate obter uma
14 # sequencia de tamanho 15
15 rep_len(x = 1:10, length.out = 15)
```

Console:

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

Script:

```
16 # Sequencia criada de 1 a 2, espacada em 0.1
17 seq(from = 1, to = 2, by = 0.1)
```

Console:

```
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Script:

```
18 # Sequencia criada de 1 a 10, espacada em 1
19 seq(from = 1, to = 10, by = 1)
```

Console:

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

Script:

```
20 # Sequencia criada de 1 a 10 de forma equiespaciada de comprimento 20
21 seq(from = 1, to = 10, length.out = 20)
```

Console:

```
[1] 1.000000 1.473684 1.947368 2.421053 2.894737
[6] 3.368421 3.842105 4.315789 4.789474 5.263158
[11] 5.736842 6.210526 6.684211 7.157895 7.631579
[16] 8.105263 8.578947 9.052632 9.526316 10.000000
```

Script:

```
22 # Eh o mesmo que 1:length(y)
23 y <- rnorm(10)
24 seq(along.with = y)
```

Console:

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

Script:

```
25 # Sequencia de 1 a 20
26 seq(20)
```

Console:

```
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[17] 17 18 19 20
```

Script:

```
27 # Sequencia criada de 10 a 100 de mesmo comprimento de x
28 x <- 1:10
29 seq(from = 10, to = 100, along.with = x)
```

Console:

```
[1] 10 20 30 40 50 60 70 80 90 100
```

Script:

```
30 # (Versao mais rapida para seq) Eh o mesmo que
31 # 1:length(w)
32 w <- c(4, 3, 6, 9)
33 seq_along(w)
```

Console:

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

Script:

```
34 # (Versao mais rapida para seq) Eh o mesmo que 1:4
35 seq_len(4)
```

Console:

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

Script:

```
36 # Eh o mesmo que seq(3) e seq(2) concatenados
37 sequence(nvec = c(3, 2))
```

Console:

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

Script:

```
38 # Eh o mesmo que c(seq(from = 2, length.out = 3), seq(from = 2, length.out =
2))
39 sequence(nvec = c(3, 2), from = 2L)
```

Console:

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

Script:

```
40 # Eh o mesmo que c(seq(from = 2, by = 2, length.out = 3), seq(from = 2, by =
2, length.out = 2))
41 sequence(nvec = c(3, 2), from = 2L, by = 2L)
```

Console:

```
[1] 2 4 6 2 4
```

Script:

```
42 # Eh o mesmo que c(seq(by = -1, length.out = 3), seq(by = 1, length.out = 2)
43 sequence(nvec = c(3, 2), by = c(-1L, 1L))
```

Console:

```
[1] 1 0 -1 1 2
```

Script:

```
44 # Repetir seq(3) cinco vezes e agrupar em matriz
45 replicate(n = 5, seq(3), simplify = TRUE)
```

Console:

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

Script:

```
46 # Repetir seq(3) cinco vezes e agrupar em lista
47 replicate(n = 5, seq(3), simplify = FALSE)
```

Console:

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

[[2]]
[1] 1 2 3

[[3]]
[1] 1 2 3

[[4]]
[1] 1 2 3

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

Script:

```
48 # Repetir rnorm(10) cinco vezes e agrupar em matriz
49 replicate(n = 5, expr = rnorm(10), simplify = TRUE)
```

Console:

	[,1]	[,2]	[,3]	[,4]	[,5]
[1,]	0.04564093	0.31394292	-0.7193373	0.1619644	1.44376461
[2,]	-0.91144307	1.15547056	0.5652449	-0.1769829	-0.73825047
[3,]	-0.32603749	2.56355701	0.2840667	0.2972674	0.17699210
[4,]	1.07288768	0.54190221	0.5033347	-0.6726417	-0.39392794
[5,]	-2.86546926	0.14235754	0.6171612	-0.1262995	-0.54758751
[6,]	-1.17814087	0.03795303	-0.4327536	-0.4075832	-0.10747510
[7,]	-0.45082684	0.56284770	-2.0979494	-0.9825204	-0.07095238
[8,]	-0.64146359	0.68855332	1.3879608	0.7976189	0.62176274
[9,]	-2.11079953	-0.20735310	-0.4822425	1.5073785	0.24446975
[10,]	-1.28789305	-1.11694324	0.4970817	-0.2412251	-0.53750894

Script:

```
50 # Repetir rnorm(10) cinco vezes e agrupar em lista
51 replicate(n = 5, expr = rnorm(10), simplify = FALSE)
```

Console:

```
[[1]]
[1] 2.05622089 -0.95096153 0.37797923 1.97553481 -0.28580088
[6] 0.03638828 -0.55331311 -0.55742017 0.03625810 0.67758385

[[2]]
[1] -0.55620379 1.75424840 -0.05765807 0.87522543 -0.73049306
[6] -1.07638261 1.16772202 -0.34813540 -0.11869188 0.21335863

[[3]]
[1] -1.23585520 -0.78408176 -1.13901273 -1.72720387 -1.39621880
[6] -0.02914018 0.40058218 1.59691114 -1.39752817 0.04067871

[[4]]
[1] 0.4106562 0.3112637 2.3177098 0.8866842 1.0332637
[6] 0.2612088 -0.7237314 1.3323660 -0.7692072 0.3259163

[[5]]
[1] 0.20261552 0.57348341 -0.90615813 -0.25560386 -2.32047150
[6] 0.40605430 -0.94521682 -0.20904290 0.49023806 -0.01623409
```

Script:

```
52 # Analise de experimento:
53 # 2 niveis com 8 repeticoes
54 gl(n = 2, k = 8, labels = c("Control", "Treat"))
```

Console:

```
[1] Control Control Control Control Control Control Control Control
[9] Treat   Treat   Treat   Treat   Treat   Treat   Treat   Treat
Levels: Control Treat
```

Script:

```
55 # Analise de experimento:
56 ## 20 parcelas com dois niveis (1 e 2)
57 gl(n = 2, k = 1, length = 20)
```

Console:

```
[1] 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2 1 2
Levels: 1 2
```

Script:

```
58 # Analise de experimento:
59 ## 20 parcelas com dois niveis (1 e 2)
60 gl(n = 2, k = 2, length = 20)
```

Console:

```
[1] 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2
Levels: 1 2
```

Dessa forma, poderemos ter com o último objeto (`vetor.num3`) uma economia de memória, dependendo do tamanho do seu vetor, quando se compara com as outras opções, isto é,

Console:

```
> # Tamanho de memoria dos objetos
> lobstr::obj_size(1:10)
680 B
> lobstr::obj_size(1:10000)
680 B
> lobstr::obj_size(1:1000000)
680 B
> lobstr::obj_size(c(1:10))
96 B
> lobstr::obj_size(c(1:1000))
40,048 B
> lobstr::obj_size(c(1:1000000))
4,000,048 B
```

4.4.1.3 Manipulando vetores

Quando algum elemento de um vetor não está disponível, representamos pela constante lógica `NA`, que pode ser coagida para qualquer outro modo de vetor, exceto para `raw`. Podemos ter constantes lógicas `NA` específicas para modos específicos: `NA_integer_`, `NA_real_` (o equivalente para o modo `double`), `NA_complex_` e `NA_character_`. Entretanto, dependendo de onde o `NA` é inserido, o atributo `modo` no objeto já converte para `NA` específico de acordo com o seu atributo `modo`. Essa constante contida no vetor não altera o modo do vetor, isto é,

Console:

```
> typeof(c(1, 2, 3, NA))
[1] "double"
> typeof(c(1, 2, 3, NA))
[1] "double"
> typeof(c("c", "b", "a", NA))
[1] "character"
```

Podemos criar vetores atômicos iniciais sem nenhuma elemento, por meio das funções `numeric(0)`, `character(0)` e `logical(0)`, isto é,

Console:

```
> # Vetor numerico de comprimento 0
> v1 <- numeric(0); length(v1)
[1] 0
> v2 <- character(0); length(v2)
[1] 0
> v3 <- logical(0); length(v2)
[1] 0
```

Para inserirmos valores a esses vetores usamos o sistema de indexação, que no caso da linguagem R, o contador começa a partir do número 1⁴. Vejamos,

Console:

```
> # Vetor numerico de comprimento 0
> v1 <- numeric(0)
> v2 <- character(0)
> v3 <- logical(0)
> # Inserimos 3 elementos em v1 e depois imprimimos o seu resultado
> v1[1] <- 5; v1[2] <- 3; v1[3] <- 10; v1
[1] 5 3 10
> length(v1)
[1] 3
```

Assim, como exercício vocês podem completar para os dois outros vetores. Uma vez criado o vetor, se desejarmos acessar os seus elementos, usamos também o sistema de indexação:

Console:

```
> # Vetor numerico de comprimento 0
> v1 <- numeric(0)
> v2 <- character(0)
> v3 <- logical(0)
>
> # Inserimos 3 elementos em v1 e depois imprimimos o seu resultado
> v1[1] <- 5; v1[2] <- 3; v1[3] <- 10
>
> # Imprimindo apenas o primeiro valor
> v1[1]
[1] 5
>
> # Imprimindo os dois ultimos
> v1[2:3]; v1[c(2, 3)]
[1] 3 10
[1] 3 10
>
> # Imprimindo todos
> v1
[1] 5 3 10
```

4.4.1.4 Aritmética e outras operações

As operações com vetores não necessariamente são as operações realizadas baseadas na álgebra de matrizes. O que a linguagem 'R' faz é realizar as operações elemento a elemento, mantendo o comprimento de tamanho igual ao tamanho do maior vetor na operação. Vejamos as operações aritméticas entre vetores de tamanho 1, a seguir.

Código R 4.20

Script:

```
1 # Soma de dois vetores
2 2 + 3
```

⁴Diferente de outras linguagens, como a C, que o contador começa do número 0.

Console:

```
[1] 5
```

Script:

```
3 # Exceto pela sintaxe, '+' eh uma chamada de funcao
4 `+`(2, 3)
```

Console:

```
[1] 5
```

Script:

```
5 # Subtracao de dois vetores
6 3 - 2
```

Console:

```
[1] 1
```

Script:

```
7 # Exceto pela sintaxe, '-' eh uma chamada de funcao
8 `-`(3, 2)
```

Console:

```
[1] 1
```

Script:

```
9 # Multiplicacao de dois vetores
10 3 * 2
```

Console:

```
[1] 6
```

Script:

```
11 # Exceto pela sintaxe, '*' eh uma chamada de funcao
12 `*`(3, 2)
```

Console:

```
[1] 6
```

Script:

```
13 # Divisao de dois vetores
14 3 / 2
```

Console:

```
[1] 1.5
```

Script:

```
15 # Exceto pela sintaxe, '/' eh uma chamada de funcao
16 `/^(3, 2)
```

Console:

```
[1] 1.5
```

Essas mesmas operações podem ser realizadas elemento a elemento para vetores de comprimento maior que 1, observemos a próxima execução de comandos.

Console:

```
> # Soma de vetores
> c(4, 5, 6) + c(1, 2, 3)
[1] 5 7 9
> # Subtracao de vetores
> c(4, 5, 6) - c(1, 2, 3)
[1] 5 7 9
> # Multiplicacao de dois vetores
> c(4, 5, 6) * c(1, 2, 3)
[1] 4 10 18
> # Divisao de dois vetores
> c(4, 5, 6) / c(1, 2, 3)
[1] 4.0 2.5 2.0
```

Quando os vetores não têm mesmo comprimento, o **R** completará de forma sequencial o menor vetor até que ele atinja o tamanho do maior vetor, isto é,

Console:

```
> # Soma de vetores de comprimento diferente
> 1:10 + 3:10
[1] 4 6 8 10 12 14 16 18 12 14
Warning message:
In 1:10 + 3:10 :
  longer object length is not a multiple of shorter object length
```

O segundo vetor repetiu os elementos 3, 4, 5, isto é, os três primeiros elementos do vetor, para que o seu comprimento se tornasse igual ao comprimento do primeiro vetor. Após isso, foi realizado a soma elemento a elemento. Esse procedimento ocorre com os demais tipos de operações. Demais operações podem ser realizadas de acordo com as funções apresentadas na Tabela 4.3.

Demais funções podem ser procuradas no manual An Introduction to R (VENABLES; SMITH; R CORE TEAM, 2021), ou execute no *console* `?Arithmetic`.

Tabela 4.3: Funções ou operadores matemáticos.

Função (Ou operador)	Finalidade
+	Soma unária, por exemplo (+ 4), ou binária entre dois vetores
-	Subtração unária, por exemplo (- 3), ou binária entre dois vetores
*	Multiplicação entre dois vetores
/	Divisão entre dois vetores
$^$ ou $**$	Exponenciação binária, isto é 2^3 ou $2 ** 3$
$\% / \%$	Divisão inteira
$\% \%$	Restante da divisão
sum()	Soma de elementos de um vetor
prod()	Produtório dos elementos de um vetor
sqrt()	Raiz quadrada dos elementos de um vetor
log()	Função Logaritmo neperiano
log10()	Função Logaritmo na base 10
exp()	Função exponencial
mean()	Média dos elementos de um vetor
sd()	Desvio padrão dos elementos de um vetor
var()	Variância dos elementos de um vetor
median()	Mediana dos elementos de um vetor
round()	Arredondamento de vetor numérico. Outros tipos são: trunc(), floor() e ceiling()

4.4.1.5 Operadores lógicos

Os operadores lógicos têm a função de avaliar determinada condição e retornar TRUE ou FALSE, sendo apresentados na Tabela 4.4.

Tabela 4.4: Operadores lógicos.

Operador lógico	Sintaxe	Pergunta
<	a < b	a é menor que b?
>	a > b	a é maior que b?
==	a == b	a é igual b?
!=	a != b	a é diferente b?
>=	a >= b	a é maior ou igual a b?
<=	a <= b	a é menor ou igual a b?
%in%	'a' %in% c('a', 'b', 'c')	O elemento 'a' está no vetor c('a', 'b', 'c')?

A operação binária significa que a função exige dois argumentos (ou operandos), isto é, <Arg1> <Operador> <Arg2>. Para mais detalhes, use no console ?Syntax. Vejamos alguns exemplos, no Código R 4.21.

Código R 4.21

Script:

```
1 # Operador '>' entre vetores de comprimento 1
2 1 > 3
```

Console:

```
[1] FALSE
```

Script:

```
3 # Operador '<' com vetor de comprimento maior que 1
4 1 < c(0, 1, 3)
```

Console:

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

Script:

```
5 # Operador '==' entre vetores
6 c(1, 2, 3) == c(3, 2, 1)
```

Console:

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

Script:

```
7 # Operador '%in%' verificando se os elementos do primeiro vetor
8 # estao no segundo vetor
9 1 %in% c(3, 4, 5)
```

Console:

```
[1] FALSE
```

Script:

```
10 # Operador '%in%' verificando se os elementos do primeiro vetor
11 # estao no segundo vetor
12 c(1, 2) %in% c(3, 4, 5)
```

Console:

```
[1] FALSE FALSE
```

Script:

```
13 # Operador '%in%' verificando se os elementos do primeiro vetor
14 # estao no segundo vetor
15 c(1, 2, 3) %in% c(3, 4, 5)
```

Console:

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

Script:

```
16 # Operador '%in%' verificando se os elementos do primeiro vetor
17 # estao no segundo vetor
18 c(1, 2, 3, 4) %in% c(3, 4, 5)
```

Console:

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

O que é interessante nesse operador `%in%`, que na realidade é uma função com dois argumentos, constitui uma forma de criar operadores binários especiais do tipo `%<nome_sintatico>%`, que esse tipo de função é uma das mais conhecidas hoje na análise de dados usando o operador pipe (`%>%`) do pacote **magrittr** da família de pacotes **tidyverse**. A diferença no operador pipe é que o segundo operando (Argumento 2) é uma função que recebe no primeiro argumento o operando 1 (Argumento 1). Por fim, o operador `%>%` acaba sendo um operado unário. Esse operador acabou ganhando tanta visibilidade que na versão **R** ($\geq 4.1.0$), foi implementado a versão nativa do operador `pipe` (`|>`).

Veremos mais detalhes na seção sobre criação de funções.

4.4.1.6 Operadores booleanos

O operadores booleanos avaliam diversas operações lógicas (condições) para ao final retornar um TRUE ou FALSE. Na Tabela 4.5 a seguir, com esses operadores e suas indagações.

Tabela 4.5: Operadores booleanos.

Operador booleano (ou função)	Sintaxe	Pergunta
& ou &&	cond1 & cond2	As cond1 e cond2 são verdadeiras?
ou	cond1 cond2	A cond1 ou cond2 é(são) verdadeira(s)?
xor()	xor(cond1, cond2)	Apenas a cond1 ou a cond2 é verdadeira?
!	!cond1	É falso a cond1?
any()	any(cond1, cond2, ...)	Alguma das condições são verdadeiras?
all()	all(cond1, cond2, ...)	Todas as condições são verdadeiras?

Vejamos alguns exemplos de operadores booleanos, no Código R 4.22. Deixamos como sugestão de exercício, o desenvolvimento de rotinas apresentando condições para os demais operadores booleanos.

Código R 4.22

Script:

```
1 # Criando objetos
2 x <- 1:3
3 y <- 1:3
4 z <- c(1, 2, 4)
5 # Primeira condicao
6 x == y
```

Console:

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

Script:

```
7 # Segunda condicao
8 y == z
```

Console:

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

Script:

```
9 # Terceira condicao
10 x == y & y == z
```

Console:

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

4.4.2 Matrizes bidimensionais

A apresentação dos próximos objetos daqui pra frente, desde matrizes até quadro de dados (*data frame*) não é apresentar todas as manipulações possíveis sobre esses objetos. Mas mostrar a sua estrutura e condições básicas impõe sobre eles. Assim, não apresentaremos funções para manipulações com matrizes, por exemplo, porque isso não é o propósito do curso. Daremos a ideia de que uma matriz é na realidade um vetor bidimensional, assim como um quadro de dados que na realidade é uma lista.

Quando usamos um atributo chamado `dim` em um vetor atômico, criamos na realidade vetores bi ou multidimensionais, isto é, objetos do tipo matrizes ou *arrays*. Assim como falamos anteriormente, o atributo pode mudar a estrutura do objeto. Vejamos o Código R 4.23, para o entendimento desse tipo de objeto.

Código R 4.23**Script:**

```
1 # Criando um vetor atomico
2 x <- 1:6; x
```

Console:

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

Script:

```
3 # Verificando se o objeto 'x' tem atributo adicionado
4 attributes(x)
```

Console:

```
NULL
```

Script:

```
5 # Vamos verificar a classe do objeto x
6 sloop::s3_class(x)
```

Console:

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

Script:

```
7 # Adicionando o atributo dim
8 dim(x) <- c(2, 3) # 2 x 3 = 6 (Comp do vetor)
9 # attr(x, "dim") <- c(2, 3)
10 # Observando agora o comportamento do objeto 'x'
11 x
```

Console:

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

Script:

```
12 # Verificando se o objeto 'x' tem atributo adicionado
13 attributes(x)
```

Console:

```
$dim
[1] 2 3
```

Script:

```
14 # Verificando a classe do objeto
15 sloop::s3_class(x)
```

Console:

```
[1] "matrix" "integer" "numeric"
```

O atributo `dim` recebeu uma informação bidimensional, isto é, o número de linhas e colunas, respectivamente. Uma outra forma para construir uma matriz é usando a função `matrix()`, que pode ser apresentado no Código R 4.24.

Código R 4.24**Script:**

```
1 # Criando uma matriz (Numeros inseridos em linhas)
2 matrix(1:6, 2, 3)
```

Console:

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

Script:

```
3 # Criando uma matriz (Numeros inseridos em colunas)
4 matrix(1:6, 2, 3, byrow = TRUE)
```

Console:

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

Como verificado no Código R 4.24 [linha 4], quando desejamos criar uma matriz inserindo os valores em colunas, usamos o argumento `byrow = TRUE`. Para acessarmos ou alterarmos os elementos de uma matriz, usamos o sistema de indexação similar ao vetor, porém, devemos indexar as linhas e colunas. Por exemplo, o elemento da primeira linha e primeira coluna pode ser obtido por `x[1, 1]`, e assim por diante. Todos os elementos da linha 1, `x[1,]`, ou todos os elementos da coluna 1, `x[, 1]`.

4.4.3 Matrizes multidimensionais

A ideia do objeto matriz multidimensional (ou *array*) é similar ao da matriz, a diferença é que agora é um vetor atômico de mais de duas dimensões, como pode ser observado no Código R 4.25.

Código R 4.25**Script:**

```
1 # Criando um vetor atomico
2 x <- 1:12; x
```

Console:

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

Script:

```
3 # Verificando se o objeto 'x' tem atributo adicionado
4 attributes(x)
```

Console:

```
NULL
```

Script:

```
5 # Vamos verificar a classe do objeto x
6 sloop::s3_class(x)
```

Console:

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

Script:

```

7 # Adicionando o atributo dim
8 dim(x) <- c(2, 3, 2) # 2 x 3 x 2 = 12 (Comp do vetor x)
9 # attr(x, "dim") <- c(2, 3, 2)
10 # Observando agora o comportamento do objeto 'x'
11 x

```

Console:

```

, , 1

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

, , 2

[,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

```

Script:

```

12 # Verificando novamente se 'x' tem atributo
13 attributes(x)

```

Console:

```
$dim
[1] 2 3 2
```

Script:

```

14 # Verificando a classe do objeto
15 sloop::s3_class(x)

```

Console:

```
[1] "array"   "integer" "numeric"
```

Criamos duas matrizes de dimensão (2 x 3). Para acessar os elementos desse objeto, usaremos também o sistema de indexação, agora acrescentando a terceira dimensão. Por exemplo, para acessar o elemento da linha 1, coluna 1, matriz 1, temos `x[1, 1, 1]`, ou todos os elementos da linha 1, matriz 1, temos `x[1, , 1]`. Uma outra forma de criar um objeto *array* é usar a função `array()`, isto é,

Console:

```
> # Criando um array
> array(1:12, c(2, 3, 2))
, , 1

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

, , 2

[,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Mostramos um quadro resumo de funções que podem ser utilizadas (WICKHAM, 2019), correlatas para vetores, matrizes e *arrays*, Tabela 4.6. Para auxílio nas funções, use sempre o símbolo de interrogação antes das funções e execute no console. Por exemplo, ajuda da função `names()`, use `?names()`.

Tabela 4.6: Operadores booleanos.

Vetor	Matriz	Array
<code>names()</code>	<code>rownames()</code> , <code>colnames()</code>	<code>dimnames()</code>
<code>length()</code>	<code>nrow()</code> , <code>ncol()</code>	<code>dim()</code>
<code>c()</code>	<code>rbind()</code> , <code>cbind()</code>	<code>abind::abind()</code>
<code>-</code>	<code>t()</code>	<code>aperm()</code>
<code>is.null(dim(x))</code>	<code>is.matrix()</code>	<code>is.array</code>

4.4.4 Listas

As listas são como vetores atômicos, porém mais complexos, isto é, os elementos de uma lista são vetores atômicos, como também outras listas, funções, expressões. Esta última é o que chamamos de objetos recursivos. A forma de se obter uma lista é pela função `list()`. Vejamos os comandos no Código R 4.26.

Código R 4.26**Script:**

```
1 # Criando uma lista
2 l0 <- list(1:3, letters[5], list(1, 2, 3),
3             mean, expression(x ~ y))
4 # Imprimindo a lista
5 l0
```

Console:

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

[[2]]
[1] "e"

[[3]]
[[3]][[1]]
[1] 1

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

[[3]][[3]]
[1] 3

[[4]]
function (x, ...)
UseMethod("mean")
<bytecode: 0x0000000008e55c60>
<environment: namespace:base>

[[5]]
expression(x ~ y)
```

Podemos acessar ou alterar os elementos de uma lista por meio do operador \$, ou pelo sistema de indexação, que diferencia um pouco da indexação dos vetores. Por exemplo, o primeiro elemento desse vetor pode ser acessado por `10[[1]]`, o terceiro `10[[3]]`, e assim por diante. Para acessar informações específicas dentro dos elementos, usamos `10[[3]][[2]]`, isto é, imprimimos o segundo valor do segundo elemento. Os elementos de um lista são na realidade outros objetos, do qual conseguimos acessar também os elementos desses objetos.

Quando nominamos os objetos contidos nas listas, podemos utilizar o operador \$, para acessar esses objetos. Vejamos Código R 4.27.

Código R 4.27**Script:**

```
1 # Criando uma lista
2 10 <- list(101 = 1:3,
3             102 = letters[5],
4             103 = list(1, 2, 3),
5             104 = mean,
6             105 = expression(x ~ y))
7 # Imprimindo o primeiro elemento (objeto) da lista '10'
8 10$101
9 # Imprimindo o segundo
10 10$102
```

Console:

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

Script:

```
11 # Imprimindo o segundo
12 10$102
```

Console:

```
[1] "e"
```

As listas têm importâncias diversas dentro do ambiente R, por exemplo, o atributo em um objeto é armazenado em forma de lista. A coerção sempre força um vetor atômico a uma lista. Vejamos as linhas de comando a seguir, Código R 4.28.

Código R 4.28**Script:**

```
1 # Vejamos as linhas de comando
2 11 <- list(list(1, 2), c(3, 4))
3 12 <- c(list(1, 2), c(3, 4))
4 # Vejamos as suas estruturas
5 str(11)
```

Console:

```
List of 2
$ :List of 2
..$ : num 1
..$ : num 2
$ : num [1:2] 3 4
```

Script:

```
6 str(12)
```

Console:

```
List of 4
$ : num 1
$ : num 2
$ : num 3
$ : num 4
```

Observamos no objeto 11, temos uma lista cujos elementos são outra lista, o elemento 3 e o elemento 4. O vetor `c(3, 4)` se transformou em dois elementos de 11. No objeto 12, poderíamos pensar que como a lista está dentro da função `c()`, os elementos da lista fariam parte dos elementos de um vetor. Porém isso não ocorre. O que temos é uma coerção em que a lista força ao vetor a se tornar lista. Por fim, temos em 12 quatro elementos em uma lista.

4.4.5 Quadro de dados

O objeto quadro de dados (*Data frame*) é uma lista com classe `data.frame`, em que contém dois atributos. Porém, com algumas restrições:

- Os componentes devem ser vetores uni ou multidimensionais, listas ou até mesmo quadro de dados;
- As colunas das matrizes, listas ou quadro de dados são inseridas como colunas do quadro de dados;
- A partir da versão **R** (4.0.0), os vetores terão mesmo modo no quadro de dados. Antes os vetores em modo caractere eram convertidos em objeto do tipo fator. Para convertê-lo automaticamente use o argumento `stringsAsFactors = TRUE`. Por sugestão, prefira a mudança usando a função `factor()`, para ter um maior controle dos níveis;
- Os objetos inseridos no quadro de dados devem ter o mesmo comprimento.

Para criarmos um objeto do tipo quadro de dados (*data frame*), usamos a função `data.frame()`. Assim, como nas listas podemos inserir os objetos no quadro de dados inserindo o nome nas colunas ou não. A forma de acessar os elementos é interessante, podemos usar a sintaxe de indexação de uma lista ou de uma matriz. Vejamos o Código R 4.29.

Código R 4.29

Script:

```
1 # Criando um quadro de dados
2 dados <- data.frame(x = 1:10,
3                      y = letters[1:10],
4                      z = rep(c(TRUE, FALSE), 5))
5 # Imprimindo dados
6 dados
```

Console:

	x	y	z
1	1	a	TRUE
2	2	b	FALSE
3	3	c	TRUE
4	4	d	FALSE
5	5	e	TRUE
6	6	f	FALSE
7	7	g	TRUE
8	8	h	FALSE
9	9	i	TRUE
10	10	j	FALSE

Script:

```
7 # Acessando os elementos de forma de lista
8 dados[[1]]
```

Console:

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

Script:

```
9 dados$x
```

Console:

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

Script:

```
10 # Acessando os elementos em forma de matriz
11 dados[1, ] # Coluna 1
```

Console:

```
x y z
1 1 a TRUE
```

Script:

```
12 dados[1, 1] # Elemento da linha 1 coluna 1
```

Console:

```
[1] 1
```

Script:

```
13 dados[, 1] # Linha 1
```

Console:

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

Quando importamos um conjunto de dados, por exemplo usando a função `read.table()`, o objeto que armazena esses dados é um quadro de dados. Assunto discutido mais a frente.

A semelhança com a forma retangular de uma matriz, faz com que algumas funções utilizadas em matrizes sejam utilizadas em quadro de dados:

- As funções `rownames()` e `colnames()`, retornam ou inserem os nomes das linhas e colunas, respectivamente. A função `names()` retorna o nome das colunas.
- A dimensão das linhas e colunas podem ser obtidas pelas funções `nrow()` e `ncol()`, respectivamente. A função `length()` retorna o número de colunas.

Em algumas situações, estamos interessados em otimizar o nosso tempo de programação, e achamos muito demorado ou não conveniente a utilização da sintaxe `objeto$elemento` para acessar os elementos de uma lista. Dessa forma, poderemos utilizar a função `attach()` para que os elementos do quadro de dados estejam disponíveis (anexados) no caminho de busca, e assim, possamos acessar os elementos (ou objetos) do quadro de dados sem precisar mencioná-lo. Vejamos,

Essa função `attach()` tem implicações, quando por exemplo se deseja inseri-la na construção de um pacote **R**. Iremos discutir esse ponto mais a frente. Para desanexar o quadro de dados, use

`detach()`. A função `attach()` é genérica e pode ser usada em qualquer objeto de modo list ser anexado no caminho de busca.

4.4.6 Fatores

4.4.7 Datas e horas

4.4.8 Objeto para dados temporais

4.4.9 Objeto *tibble*

Exercícios propostos

Exercício 4.1: Apresente um Código R para criarmos um *array* de dimensão 4.

Solução na página 136

Importação e exportação de dados

5.1 Introdução

A importação/exportação de dados era algo que em poucas linhas conseguíamos explicar sobre o ambiente **R**, no sentido de análise de dados. Entretanto, observando o terceiro princípio do R, afirmado por Chambers (2016): “*Interfaces para outros programas são parte do R*”.

Hoje é uma realidade a interação que o ambiente **R** tem com outras interfaces (programas, linguagens, etc.). A facilidade em utilizar outras linguagens dentro do ambiente **R** torna assim mais complexo a importação/exportação de dados, uma vez que o objetivo do **R**, apesar do *R Core Team* ainda limitar a sua definição como o ambiente para a computação estatística, a ferramenta se tornou tão versátil, que hoje torna humilde essa definição. Para mais detalhes acesse o manual **??R Data Import/Export** (R CORE TEAM, 2021a). Um outro fator e tema atual é a era dos grandes bancos de dados (*Big Data*), do qual se tem um grande conjunto de variáveis e necessitamos fazer a importação por APIs¹, por exemplo, ou outras vias. Temas como esses, serão abordados em outros volumes da coleção *Estudando o Ambiente R*.

Nesse momento, limitaremos esse assunto ao objetivo de termos um conjunto de dados em arquivos de texto (extensões do tipo **.txt**, **.csv**, **.xls**), formato binário (**.xls** ou **.xlsx**) ou digitados manualmente pelo teclado do computador. Assim, a primeira forma de como os dados estão disponíveis, precisaremos importá-los e armazená-los em um quadro de dados (*data frame*), para que esteja disponível na área de trabalho (ambiente global) do **R**, e dessa forma, possamos utilizá-lo. Ao final do tratamento dos dados, podemos exportar essas informações para arquivos externos, e daí também, usaremos os arquivos de textos e o formato binário (**.xls**), mencionados anteriormente.

5.2 Preparação dos dados

A primeira coisa que devemos entender quando desejamos construir o arquivo de dados, é entender que sempre organizaremos as variáveis em colunas, com os seus valores em linhas, Figura **??fig:bdados**). Sempre a primeira linha das colunas representará o nome das variáveis. Esse é outro ponto importante, pois devemos ter a noção que alguma linguagem irá ler esse banco de dados. Assim, quanto mais caracteres diferentes do padrão ASCII, mais difícil será a leitura desses dados. Assim, sugerimos alguns padrões:

- Evitem símbolos fora do padrão alfanumérico;
- Evitem mistura de letras minúsculas com letras maiúsculas. Isso facilitará o acesso a essas variáveis. Contudo, lembremos do padrão de nomes sintéticos permissíveis observados na seção 3.3.2;
- Lembremos que o banco de dados será utilizado para que um programa faça a sua leitura, portanto, deixemos a formatação da apresentação dos dados para arquivos específico. Sendo assim, evitem comentários nesses arquivos, ou qualquer outro tipo de informação que não seja o banco de dados;

¹Do inglês, *Application Programming Interface*, que significa Interface de programação de aplicativos.

- Evitem palavras longas, por exemplo, segundavariavel (má escolha), segvar (boa escolha), seg_var (boa escolha);
- Evitem palavras compostas com espaço entre elas. Para isso use o símbolo _, por exemplo, var_2 (má escolha), var2 (boa escolha), var_2 (boa escolha);

var1	var2	var3	var4	var5
a	1	2.5	TRUE	1
b	3	2.7	FALSE	2
c	2	4	TRUE	5
d	4	5	TRUE	4
e	5	6	FALSE	3

Figura 5.1: Modelo estrutural de um banco de dados.

5.3 Importando dados

A função primária responsável pela importação de dados é a função `scan()`. Por exemplo, funções como `read.table()`, `read.csv()` e `read.delim()`, usam a função `scan()` em seu algoritmo.

A primeira ideia sobre importação de dados pode ser inserindo-os pelo teclado no próprio ambiente **R**. Para isso, usaremos a função `scan()`, isto é,

Console:

```
# Criando e inserido os elementos do objeto dados
x <- scan()
```

Após executado essa linha de comando, aparecerá no console 1: que significa, digitar o primeiro valor do objeto `x`, e depois clicar em *ENTER*. Depois 2:, que significa digitar o segundo valor, e clicar em *ENTER*. Depois de inserido todos os valores necessários, aperte a tecla *ENTER* duas vezes no console, para sair da função `scan()`.

O mais tradicional é usar programa para criação de banco de dados e deixá-lo pronto para o **R** lê-lo. O tipo de arquivo de texto que melhor controla a separação de variáveis é com a extensão `.csv`, uma vez que separamos as variáveis por “;”, é o padrão. O arquivo de texto com extensão `.txt`, geralmente usa espaços. Isso acaba gerando problema de leitura no **R**, porque muitos usuários usam nomes de variáveis muito grandes, palavras compostas, de forma a desalinhlar as colunas das variáveis. Daí, como a separação das variáveis é por meio de espaços, acaba gerando problema de leitura. Uma outra forma, é fazer importação de dados gerados pelo próprio **R**, extensão `.RData`.

Temos a opção de usar um editor de banco de dados para essas extensões por meio de programas como *MS Excel*, *Libre Office*, dentre outros. Estes exportam arquivos binários do tipo `.xls`, `.xlsx`, dentre outros. Uma sugestão para diminuir complicações, é exportar os bancos de dados para arquivos de texto citados acima, que também é possível ser exportado por esses programas. Isso evita a necessidade de ser instalado mais pacotes e dor de cabeça. Porém, para quem ainda

deseja enfrentar, sugerimos a leitura do pacote **readr**, como exemplo, porém existem diversos outros pacotes para este mesmo fim.

Uma vez que o banco de dados está pronto, a leitura destes pode ser feita por alguns caminhos. Mostraremos o mais trivial que é o botão *Import Dataset*, terceiro quadrante, aba *Environment*, na IDE do **RStudio**, como observado na Figura 5.2.

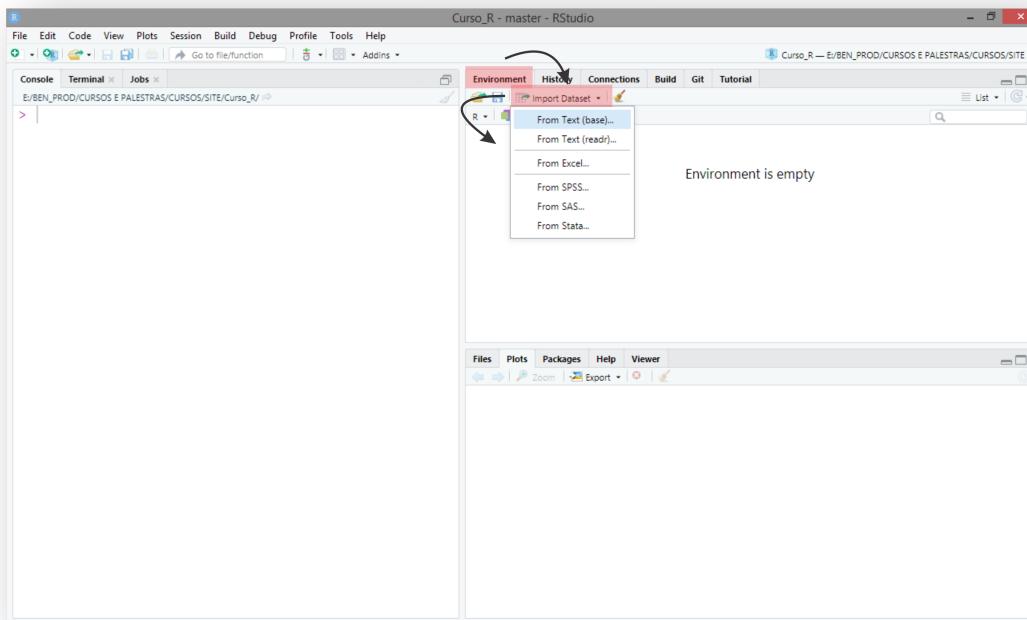


Figura 5.2: Usando o RStudio para importar dados.

Posteriormente, indique o arquivo para leitura. Aparece algumas opções de tipo de arquivo. Em nosso caso, usaremos a opção *From Text (base)*, que significa realizar a leitura para os tipos de arquivo **.txt** ou **.csv**. Daí os passos seguintes são:

- 1) Escolher o arquivo para leitura dos dados (Figura 5.3);
- 2) Configurar a leitura do banco de dados. Uma prévia pode ser vista no quadro *Data Frame*. Se for visualizado, algum problema, isso significa que deve ser informado opções adicionais como separador de variáveis (*Separator*), símbolo para casas decimais (*Decimal*), dentre outras opções. Por fim, digitar o nome associado ao objeto (*Name*) que será criado do tipo quadro de dados (*data frame*), e clicar no botão *Import* (Figura 5.4);
- 3) Uma vez inserido, o **RStudio** apresenta a linha de comando utilizada para importar os dados no console (2º quadrante), o conjunto de dados (1º quadrante), e a ligação entre o nome e o objeto no ambiente global (3º quadrante), Figura 5.5.

A outra forma é utilizar linhas de comando. Para isso utilizaremos a função `read.table()`. Antes de importarmos o banco de dados, algo interessante é inserir o arquivo de dados no diretório de trabalho no ambiente **R**. Para verificar o ambiente de trabalho use a função `getwd()`. Para alterar o local do ambiente de trabalho use `setwd()`. Se esse procedimento não for realizado, o usuário deve informar na função `read.table()`, o local exato do arquivo de texto.

Vamos usar como diretório o local `C:\cursor`. Lembre-se que no **R**, a barra deve ser invertida. Vamos inserir nesse diretório três arquivos `alfafa.txt`, `datast1980.txt` e `producao.csv`.

Os três conjuntos de dados são:

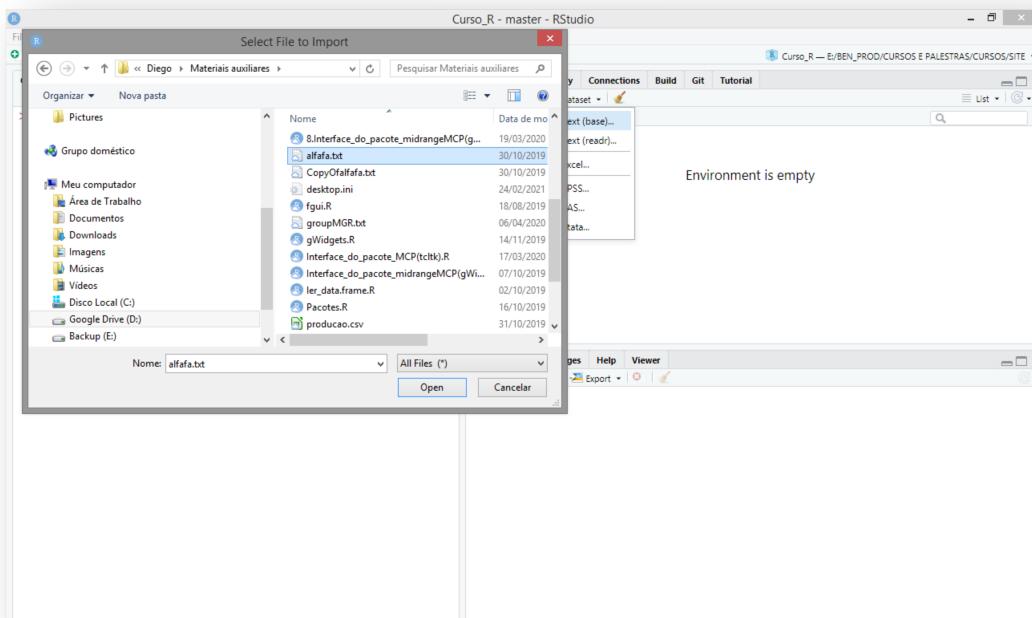


Figura 5.3: Usando o RStudio para importar dados.

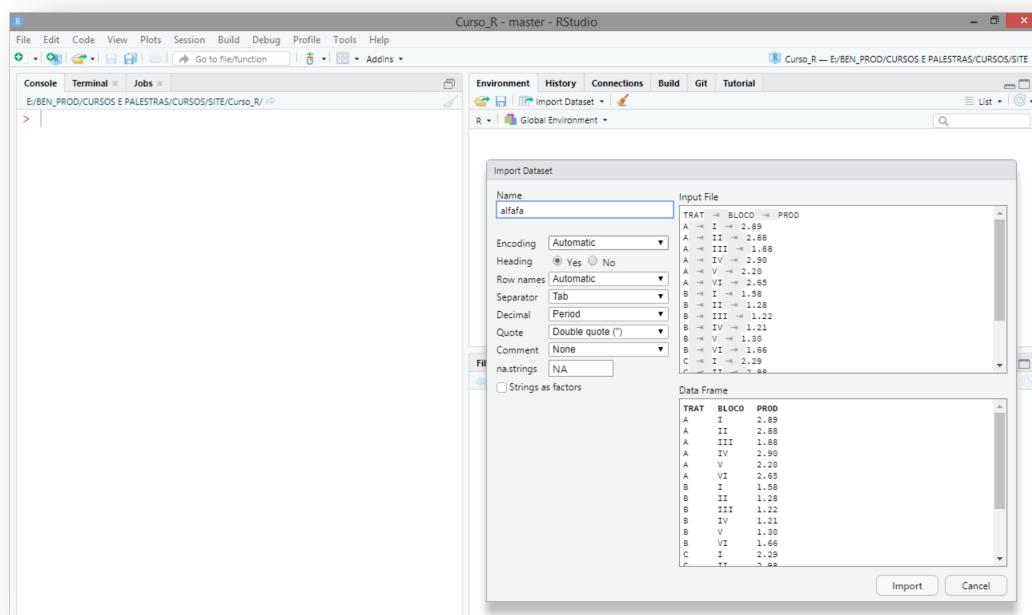


Figura 5.4: Usando o RStudio para importar dados.

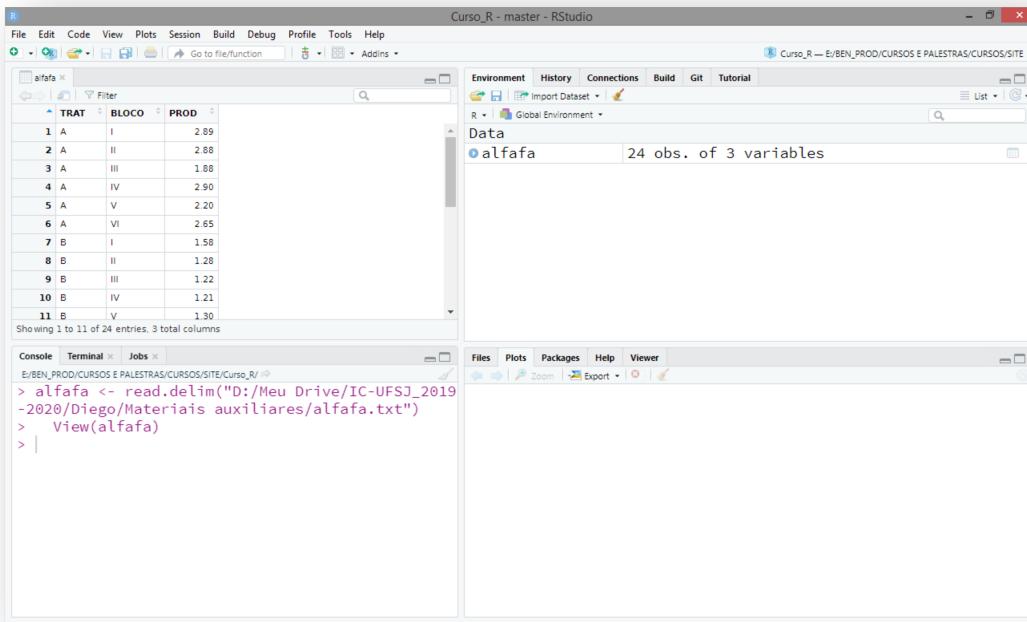


Figura 5.5: Usando o RStudio para importar dados.

- `alfafa.txt`:

TRAT	BLOCO	PROD
A	I	2.89
A	II	2.88
A	III	1.88
A	IV	2.90
A	V	2.20
A	VI	2.65
B	I	1.58
B	II	1.28
B	III	1.22
B	IV	1.21
B	V	1.30
B	VI	1.66
C	I	2.29
C	II	2.98
C	III	1.55
C	IV	1.95
C	V	1.15
C	VI	1.12
D	I	2.56
D	II	2.00
D	III	1.82
D	IV	2.20
D	V	1.33
D	VI	1.00

- datast1980.txt

trt	y
1	19,4
1	32,6
1	27,0
1	32,1
1	33,0
2	17,7
2	24,8
2	27,9
2	25,2
2	24,3
3	17,0
3	19,4
3	9,1
3	11,9
3	15,8
4	20,7
4	21,0
4	20,5
4	18,8
4	18,6
5	14,3
5	14,4
5	11,8
5	11,6
5	14,2
6	17,3
6	19,4
6	19,1
6	16,9
6	20,8

- producao.csv

x;y
1;6.7
2;7.9
3;9.1
4;6.6
5;7.5
6;8.8
7;7.7
8;7.6
9;6.5
10;7.9
11;8.7
12;6.2
13;7.9
14;7.4
15;9.7
16;6.2
17;4.9
18;5.6
19;7
20;6

Vejamos as linhas de comando para importar os dados, Código R 5.1.

Código R 5.1**Script:**

```

1 # Diretorio
2 getwd()
3 # Mudadando para o diretorio de interesse
4 setwd("C:/cursor")
5 # Verificando os arquivos no diretorio de trabalho
6 list.files()
7 # Importando os dados apontando para o diretorio do arquivo
8 dados1 <- read.table(file = "C:/cursor/alfafa.txt", header = TRUE)
9 # Considerando que o arquivo esta no diretorio de
10 # trabalho, isto eh, getwd()
11 dados2 <- read.table("alfafa.txt", header = TRUE)
12 # Importando os dados com decimais com ',' apontando para o diretorio do
#     arquivo
13 dados3 <- read.table(file = "C:/cursor/dadost1980.txt", header = TRUE, dec =
#     ",")
14 # Considerando que o arquivo esta no diretorio de
15 # trabalho, isto eh, getwd()
16 dados4 <- read.table(file = "dadost1980.txt", header = TRUE, dec = ",")
17 # Importando os dados com decimais ',', e separados por ';' apontando para o
#     diretorio do arquivo
18 dados5 <- read.table(file = "C:/cursor/producao.csv", header = TRUE, dec =
#     ",", sep = ";")
19 # Considerando que o arquivo esta no diretorio de
20 # trabalho, isto eh, getwd()
21 dados6 <- read.table(file = "producao.csv", header = TRUE, dec = ",", sep =
#     ";")
22 # Importando da internet
23 dados7 <- read.table(file =
#     "https://raw.githubusercontent.com/bendeivide/book-eambr01/main/files/_"
#     alfafa.txt", header = TRUE)

```

Na última linha de comando, mostramos que também é possível importar dados de arquivos de texto da internet, e claro considerando que o usuário está com acesso a internet no momento da importação. É um recurso interessante que pode ser feito, principalmente para este caso, é salvar o banco de dados em um arquivo de dados no '.RData'. Dessa forma, todos os dados, inclusive os importados da internet serão agora armazenados nesse tipo de arquivo, e não precisaremos, nesse caso, de acesso a internet. Para salvar, usamos a função `save()`. Para carregar os dados e armazená-lo no ambiente global, usamos a função `load()`, Código R 5.2.

Código R 5.2**Script:**

```

1 # Diretorio
2 getwd()
3
4 # Verificando os arquivos do diretorio de trabalho
5 list.files()
6
7 # Importando os dados da internet
8 dados7 <- read.table(file =
  "https://raw.githubusercontent.com/bendeivide/book-eambr01/main/files/_"
  "alfafa.txt", header = TRUE)
9
10 # Salvando em '.RData'
11 save(dados7, file = "alfafa.RData")
12
13 # Carregando '.RData' para o ambiente global
14 load("alfafa.RData")

```

Percebemos que as extensões .txt e .csv são idênticos, exceto pela estrutura de como os dados estão dispostos. Para comprovar isso, o usuário manualmente poderá mudar a extensão de um arquivo do tipo .csv para um arquivo .txt e observar em um bloco de notas.

Até agora, usamos as funções no **R**, em algumas situações, sem apresentar os argumentos dessas funções dentro dos parênteses. Isso porque quando inserimos os valores dos argumentos na posição correta destes, não precisaremos inserir o nome dos argumentos. Por exemplo, já usamos anteriormente a função **mean()** que calcula a média de um conjunto de valores, por exemplo, **valores <- 1:10**. Temos como primeiro argumento para essa função o **x** que representa um objeto **R** que recebe os valores para o cálculo. Assim, como sabemos que **x** é o primeiro argumento dessa função, podemos omitir o seu nome e calcular a média por **mean(valores)**, que é o mesmo que **mean(x = valores)**. Para mais detalhes, ?**mean()**, como também, para mais detalhes sobre a função **read.table()**, use ?**read.table()**.

5.4 Exportando dados

5.5 Exercícios

Exercício 5.1:

Solução na página 137

Funções no R

6.1 Introdução

Mais uma vez, nos reportamos aos princípios do **R** (CHAMBERS, 2016), mais especificamente ao segundo princípio, “*Tudo que acontece no R é uma chamada de função*”. Quando associamos um nome a um objeto (`x <- 10`) pelo símbolo de atribuição (`<-`), o que temos é uma chamada de função realizando esse processo, isto é, ‘`<-‘(x, 10)`. Ao digitar `x` no console e posteriormente apertando o botão *ENTER* do teclado, nos bastidores, estamos na realidade chamando a função `print(x)` para imprimir o valor que o nome se associa. Desde coisas básicas como essas, até coisas mais complexas, temos sempre por trás uma chamada de função.

6.2 O que é uma função no R?

Os princípios falados por Chambers (2016) são interligados, principalmente os dois primeiros, porque apesar de tudo que acontece no **R** ser uma chamada de função, a função é um objeto, com estrutura definida como qualquer outro objeto, de modo `function()`¹, assim como os vetores. Ainda mais, dizemos que a linguagem **R** tem um estilo funcional, devido a esse fato. Mais isso é assunto para o Volume II.

A ideia de função aqui não é pensada como uma relação matemática, mas como um sistema que tem uma entrada e saída. Podemos ter funções no ambiente **R** que organizam dados, e não operações matemáticas por exemplo. Vejamos a função `sort()` do pacote **base** que ordena de forma crescente ou descrecente um conjunto de valores, como pode ser observado no Código R 6.1.

As funções de modo `closure` (funções criadas por meio de `function()`), do qual o seu modo pode ser verificado por `typeof()`, apresenta três estruturas básicas: argumento, corpo e ambiente, que será abordada a seguir.

6.3 Estrutura básica de uma função

As funções podem ser divididas em três componentes:

- Argumentos, função `formals()`,
- Corpo, função `body()` e
- Ambiente, função `environment()`.

Para o caso das funções primitivas, escritas na linguagem C, essa regra foge a excessão, e será detalhado no Volume II. Dizemos que funções são primitivas de modo `builtin` ou `special`. Para verificarmos a tipagem desses objetos, devemos sempre usar `typeof()` ou invés de `mode()`.

¹Para esse caso, use `mode()` para verificar.

Código R 6.1**Script:**

```
1 # Vetor
2 y <- c(5, 3, 4); y
```

Console:

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

Script:

```
3 # Funcao
4 sort(x = y)
```

Console:

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

Script:

```
5 # Argumentos da funcao sort
6 formals(sort)
```

Console:

```
$x
$decreasing
[1] FALSE
$...
```

Script:

```
7 # Corpo da funcao
8 body(sort)
```

Console:

```
{
  if (!is.logical(decreasing) || length(decreasing) != 1L)
    stop("'decreasing' must be a length-1 logical vector.\nDid you intend to
set 'partial'?")
  UseMethod("sort")
}
```

Script:

```
1 # Ambiente
2 environment(sort)
```

Console:

```
<environment: namespace:base>
```

Nesse caso, os argumentos `x`, `decreasing` e ..., são nomes que aguardam receber objetos para a execução da função `sort()`. Nem todos os argumentos necessitam receber objetos, a estes chamamos de argumentos padrão, como o caso do argumento `decreasing` com padrão igual a `FALSE`, que significa que o ordenamento dos dados será de forma não-decrescente. Observe que na função `sort()` entramos apenas com o argumento `x = y`, não precisando inserir `decreasing = FALSE`. Agora, para modificar o argumento padrão, basta acrescentar a alteração na função, isto é,

Console:

```
> # Funcao
> sort(x = y, decreasing = TRUE)
[1] 5 4 3
```

O '...' é um argumento especial e significa que pode conter qualquer número de argumentos. Geralmente é utilizado em uma função quando não se sabe o número exato de argumentos. Veremos ainda nesse capítulo mais sobre esse argumento.

O próximo item é o corpo da função. É nele que inserirmos as instruções, isto é, as linhas de comandos necessárias a que se destina a sua criação. Uma outra forma de acessarmos o corpo das funções é digitar no console apenas o seu nome sem o parêntese, isto é, `sort`.

Por fim, o ambiente que no caso da função `sort()` representa o ambiente do pacote **base**, isto é, o *namespace* **base**.

6.4 Funções em pacotes

Podemos observar que essas funções utilizadas até agora, não foram criadas pelo usuário. Estas funções vieram do que chamamos de pacotes². Alguns pacotes estão disponíveis quando instalamos o **R**, dizemos que estes são os pacotes nativos do **R**, para a linguagem. O principal pacote deles é o **base**. Os demais pacotes desenvolvidos podem ser obtidos via CRAN, e falaremos mais adiante.

O ambiente **R** apresenta uma versatilidade de manuais para a linguagem. Por exemplo, para verificar informações sobre um determinado pacote como o **base**, use `help(package = 'base')`. A função `help()` pode ser utilizada para funções de pacotes anexados. Por exemplo, `help('sort')`. Uma outra função que pode ser usada para procurar por funções com determinado parte de nome é `apropos()`, isto é, para o exemplo anterior, temos `apropos('sort')`. O pacote **base** sempre estará anexado, isto é, disponível no caminho de busca para a utilização. Para os que não estão anexados, a função `help` deve informar o nome da função que necessita de ajuda, bem como o seu pacote. Por exemplo, temos uma função `read.dbf()` do pacote **foreign**, Base do **R**, porém, esse pacote não está anexado³ ao inicializar o **R**. Assim, caso não seja anexado o pacote, basta executar `library('foreign')` no console. A ajuda sobre a função pode ser realizada com `help('read.dbf', package = 'foreign')`. Outras sintaxes para a função `help()` é usar `? antes do nome de uma função de um pacote anexado`, isto é, `?sort()` para o caso da função estudada.

Para os manuais de ajuda na internet, use `??` antes do nome da função, por exemplo `??sort()`, `read.dbf()`, etc. Essa sintaxe não precisa dos pacotes estarem anexados para ajuda de determinada função, porém o pacote necessita estar instalado. Para mais detalhes sobre pacotes, leia o Capítulo 9.

²Entenda por pacote como uma estrutura contendo diretórios e arquivos específicos. Em um desses diretórios, temos o local que armazenamos as nossas funções criadas. Ao instalar e anexar esse pacote, todas essas funções tornam-se disponíveis para o usuário. No caso dos pacotes nativos, basta apenas anexá-lo, exceto para o pacote **base**.

³A anexação de um pacote no caminho de busca pressupõe que este esteja instalado no seu computador.

6.5 Chamadas de funções

As chamadas de funções podem ocorrer de três formas: aninhada, intermediária ou de forma *pipe*. Todas essas formas estão implementadas no **R** de forma nativa, porém este último, foi implementado inicialmente pelo pacote **magrittr**, e na versão do **R 4.1**, foi implementado o operador *pipe* nativo (`|>`), sendo aprofundado no Volume II.

Suponha que desejamos calcular o desvio padrão de um conjunto de valores. Vamos utilizar as três formas de chamadas de função, que segue:

- Aninhado:

Console:

```
> # Gerando 100 numeros aleatorios de uma distribuicao normal
> set.seed(10) # Semente
> x <- rnorm(100)
> # Calculando o desvio padrao
> sqrt(var(x))
[1] 0.9412359
```

- Intermediária:

Console:

```
> # Calculando o desvio padrao
> vari <- var(x)
> desvpad <- sqrt(vari); desvpad
[1] 0.9412359
```

- *Pipe*:

Console:

```
# Para usar o pipe (magrittr), substitua |> por %>%, e instale e anexe o pacote:
# install.packages(magrittr)
# library(magrittr)
# Calculando o desvio padrao
> x |>
>   var() |>
>   sqrt()
[1] 0.9412359
```

A ideia da chamada de função aninhada é inserir função como argumento de funções sem necessidade de associar nomes aos objetos. A ordem de execução começa sempre da direita para a esquerda. No caso da chamada de função intermediária, associamos nomes a cada função, e os passos seguem. Por fim, o operador especial pipe (`%>%`) tem como primeiro operando o primeiro argumento da função no segundo operando.

Quando desenvolvemos pacotes, preferimos os dois primeiros, pois é a forma tradicional de chamadas de função no **R**. A chamada de função *pipe* é muito utilizada para ciência de dados, uma vez que trabalhamos com manipulações de dados, visualizações gráficas, sem necessariamente precisarmos associar nomes aos objetos, de modo a armazenar seus resultados. Dessa forma é preferível o operador *pipe*.

6.6 Estruturas de controle

As funções que utilizaremos, a seguir, são utilizadas quando desejamos realizar processos repetitivos para um determinado fim ou condicionado, as famosas estruturas de controle. Assim, como em outras linguagens, as funções utilizadas são: `if()`, `switch()`, `ifelse()`, `while()`, `repeat`, `for()`. Todos esses nomes são reservados no ambiente **R**, isto é, não podemos associar esses nomes a objetos. Esses objetos tem modo *special*⁴, porque as expressão não são necessariamente avaliadas. Já a função `ifelse()` é de modo *closure*. Este último é um tipo de objeto que é uma função criada por `function()`, e será o mesmo tipo de objeto que os usuários terão quando desenvolverem as suas próprias funções.

A sintaxe das estruturas de controle de tipo *special* é:

Script:

```
1 função (condição) {
2   expressão
3 }
```

A sintaxe das funções `repeat` e `switch()` foge um pouco desse padrão, e explicaremos em sua aplicação, mais a frente. Apesar da linguagem **R** ser interpretada, acaba ganhando a fama de que as funções *loops* são mais lentas. Em alguns casos, a construção do algoritmo proporciona isso e não a implementação dessas funções em si. Um exemplo que podemos citar, são as cópias de objetos que podem ter um gasto de memória ativa imenso no processo e proporcionar um gasto computacional. E isso foi devido a forma de como o algoritmo copiou os objetos, e não o desempenho das funções *loops*. Veremos isso em detalhes no Volume II.

A primeira estrutura é o `if()`, com sintaxe:

Script:

```
1 if (condição) {
2   # Corpo do if()
3   instruções sob condição = TRUE
4 }
```

O fluxograma para o entendimento da função `if()` pode ser observado na Figura 6.1. Podemos também representar a função `if()` sintaticamente com a função `else` da seguinte forma:

Agora, acrescentando a função `else` trabalhando conjuntamente com a função `if()`, temos a sua forma sintática dada por:

Script:

```
1 if (condição) {
2   # Corpo do if()
3   instruções sob condição = TRUE
4 } else {
5   # Corpo do else
6   instruções sob condição = FALSE
7 }
```

em que o fluxograma da combinação dessas duas funções podem ser representados pela Figura 6.2.

Uma forma mais simplificada unindo as funções `if()` e `else`, resultaram na função `ifelse()`, que pode ser representada sintaticamente da seguinte forma:

⁴Por exemplo, para a função `for()` use `typeof('for')` para verificar.

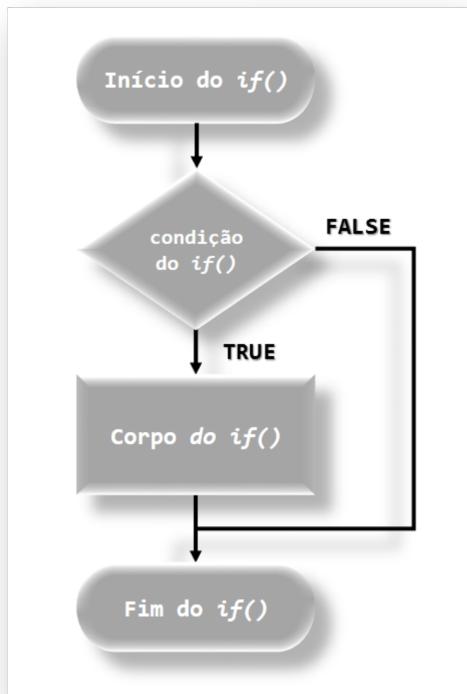


Figura 6.1: Fluxograma da função `if()`.

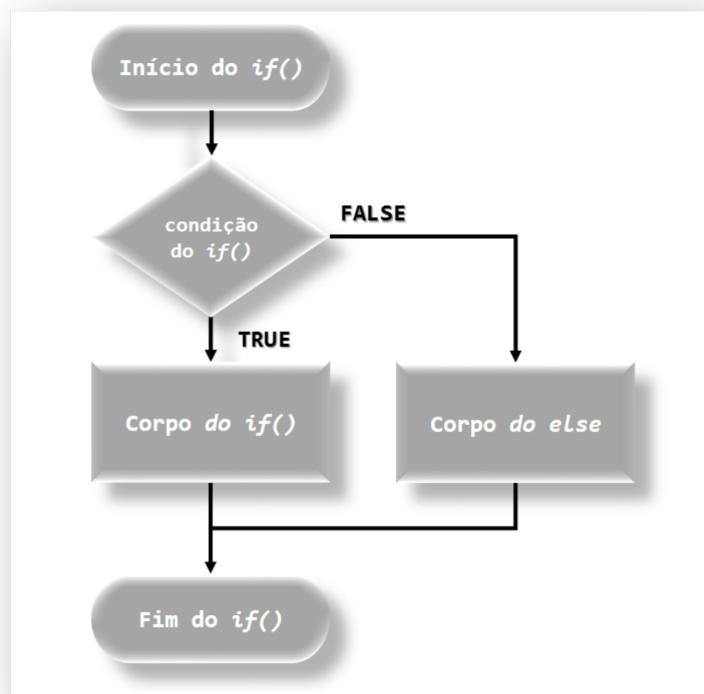


Figura 6.2: Fluxograma da função `if() e else`.

Script:

```
1 if (condição) instr1
2 if (condição) instr1 else instr2
```

Para elucidarmos essas formas sintáticas, observemos o Código R 6.2.

Código R 6.2**Script:**

```
1 # Objeto
2 i <- 5
3 # Estrutura if()
4 if (i > 3) {
5   print("Maior_que_3!")
6 }
```

Console:

```
[1] "Maior_que_3!"
```

Como observado no Código R 6.2, após associado o nome x associado ao objeto 5, isto é `i <- 5`, a função `if()` foi chamada e a condição foi verificada se `i < 3`. Como essa condição era verdadeira, a expressão sob a condição verdadeira é impressa, e a função `if` é encerrada. Vejamos mais um exemplo no Código R 6.3, a seguir.

Código R 6.3**Script:**

```
1 # Objeto numerico
2 x <- 10
3 # Estrutura 'if'
4 if (is.numeric(x)) {
5   print("Isso_é_um_número")
6 } else {
7   print("Isso_não_é_um_número")
8 }
```

Console:

```
[1] "Isso_é_um_número"
```

Script:

```
9 # eh o mesmo que
10 if (is.numeric(x) == TRUE) {
11   print("Isso_é_um_número")
12 } else {
13   print("Isso_não_é_um_número")
14 }
```

Console:

```
[1] "Isso é um número"
```

Observamos na primeira forma que a condição para `if()` avaliar, nós não precisamos perguntar se `is.numeric(x) == TRUE`, porque isso já é implícito na função, e acaba levando a primeira expressão `print("Isso é um número")`. De todo modo, em um segundo momento explicitamos a condição, e verificamos que o resultado para este caso será o mesmo. No R, essa estrutura de controle não é vetorizado, isto é, se a condição houver um vetor lógico maior que 1, apenas os primeiros itens serão usados. Para entender, vejamos o Código R 6.4.

Código R 6.4**Script:**

```
1 # Objetos
2 x <- 5
3 w <- 3:8
4 # Primeira sintaxe (Preferível)
5 if (x < w) {
6   x
7 } else {
8   w
9 }
```

Console:

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

Script:

```
10 # Segunda forma
11 if (x < w) x else w
```

Console:

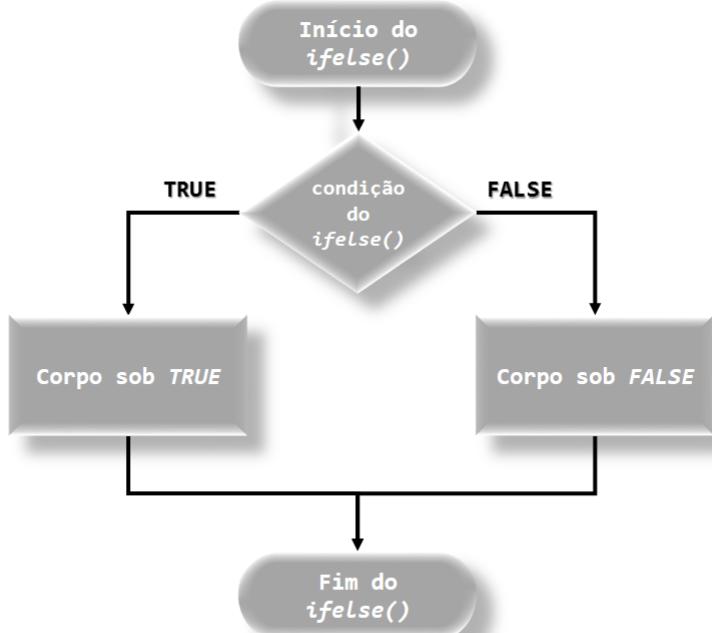
```
[1] 3 4 5 6 7 8
```

Observamos que o fato de `x < y` ser vetorizado, tivemos problema com a condição `if()`. A partir de `w > 5`, o resultado era para ter retornado sempre o valor 5, isto é, o valor verdadeiro (`x`) para a condição. Isso implica que a condição deve ser sempre entre vetores escalares. Uma saída seria utilizar a função `for()`, conjugada com `if()` e `else`. Porém, como forma de vetorizar essa condição, foi criado a função `ifelse()`, em que o fluxograma dessa função pode ser observado pela Figura, cuja forma sintática é dada a seguir.

Script:

```
1 ifelse (condição, expressão sob TRUE, expressão sob FALSE)
```

Podemos ver que a aplicação do Código R 6.4 retornou um resultado que não se esperaria como o desejado, pelo fato da função `if()` não ser vetorizada. Assim, por meio do Código R 6.5 usando

Figura 6.3: Fluxograma da função `ifelse()`.

a função `ifelse()`, verificaremos que esta por ser vetorizada, o resultado retornado ocorre como esperado.

Código R 6.5

Script:

```

1 # Objetos
2 x <- 5
3 w <- 3:8
4 # Primeira sintaxe (Preferível)
5 ifelse(x < w, x, w)
  
```

Console:

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

Podemos estar interessados em resultados específicos para determinadas condições, e daí, usar o que chamamos de programação defensiva, podendo ser apresentado no Código R 6.6.

Código R 6.6**Script:**

```

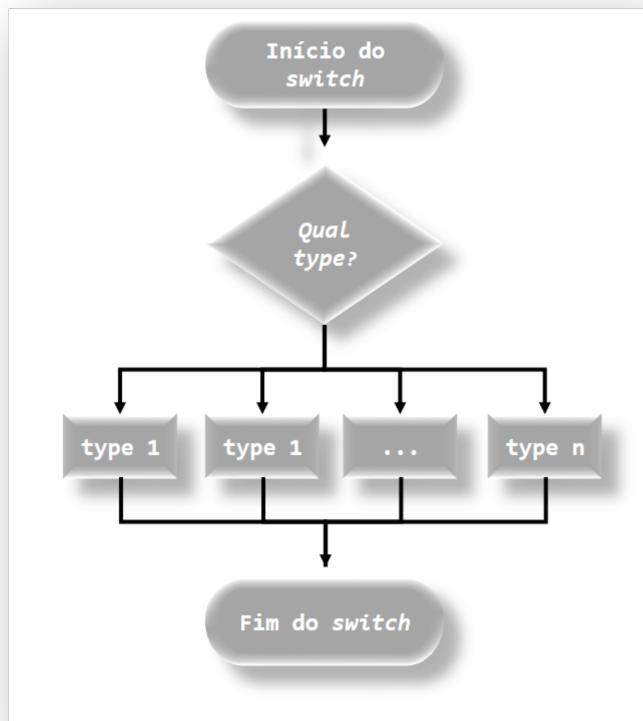
1 x <- 2 # numero
2
3 # Estrutura 'if'
4 if (!is.numeric(x)) {
5   "Nao_eh_numero"
6 } else {
7   if ((trunc(x) %% 2) == 0) {
8     cat("numero_par:", trunc(x))
9   } else {
10     if ((trunc(x) %% 2) == 1) {
11       cat("numero_impar:", trunc(x))
12     }
13   }
14 }
```

Console:

```
[1] "numero_par"
```

A ideia no Código R 6.6 é escolher um numero inteiro o algoritmo verificar se este é par ou ímpar. O que pode ocorrer como supostos erros que foram protegidos pelo código, o primeiro é de se avaliar um valor que não seja número. Nesse caso, a primeira condição `if(!is.numeric(x))` retornar uma mensagem que o valor não é um número. O segundo erro que poderia ocorrer é do valor inserido não ser um número inteiro. Nesse caso, truncamos o número apenas usando a sua parte inteira e desprezando a decimal, sem o arredondamento. Tentar evitar erros previsíveis, é o que chamamos de programação defensiva, uma forma do próprio usuário conseguir identificar o erro mais facilmente.

Um próximo exemplo, apresentado no Código R 6.6, mostrará inicialmente por meio da condição `if()` e `else` pode se apresentar um código mais complexo, sendo simplificado na sequência com o uso da função `switch()`, Código R 6.8. O fluxograma da função `switch()` pode ser observado pela Figura 6.4.

Figura 6.4: Fluxograma da função `switch()`.

Código R 6.7

Script:

```

1 # Objeto
2 set.seed(15) # Fixando a semente
3 x <- rnorm(1000) # Gerando 1000 numeros aleatorios
4 # medida descritiva
5 opcao <- "media" # opcoes: "media", "mediana", "medapar" (media aparada)
6 if (opcao == "media") {
7   cat("A_média_aritmética_é:", round(mean(x), 4))
8 } else {
9   if (opcao == "mediana") {
10     cat("A_mediana_é:", round(mean(x), 4))
11   } else {
12     if (opcao == "medapar") {
13       cat("A_média_aparada_é:", round(mean(x, trim = 0.1), 4))
14     }
15   }
16 }
  
```

Console:

A média aritmética é: 0.037

Código R 6.8**Script:**

```

1 # Objeto
2 set.seed(15) # Fixando a semente
3 x <- rnorm(1000) # Gerando 1000 numeros aleatorios
4 # medida descritiva
5 opcao <- "media" # opcoes: "media", "mediana", "medapar" (media aparada)
6 switch(opcao,
7   media = cat("A_média_aritmética_é:", round(mean(x), 4)),
8   mediana = cat("A_mediana_é:", round(mean(x), 4)),
9   medapar = cat("A_média_aparada_é:", round(mean(x, trim = 0.1), 4))
10 )

```

Console:

A média aritmética é: 0.037

As três estruturas básicas de *loop* no **R** são: `repeat`, `while()` e `for()`. Dizemos que um *loop* é um conjunto de instruções (expressões) em um algoritmo que se repete um número de vezes até que sejam alcançados os objetivos desejados. A repetição é controlada pelo contador, um objeto **R**. Cada repetição representa um ciclo do *loop*. Enquanto a condição do *loop* for verdadeira, os ciclos se repetem atualizando o contador, Figura 6.5. Algumas funções não têm condição explícita (`repeat` e `while`), e precisam de funções adicionais em suas expressões.

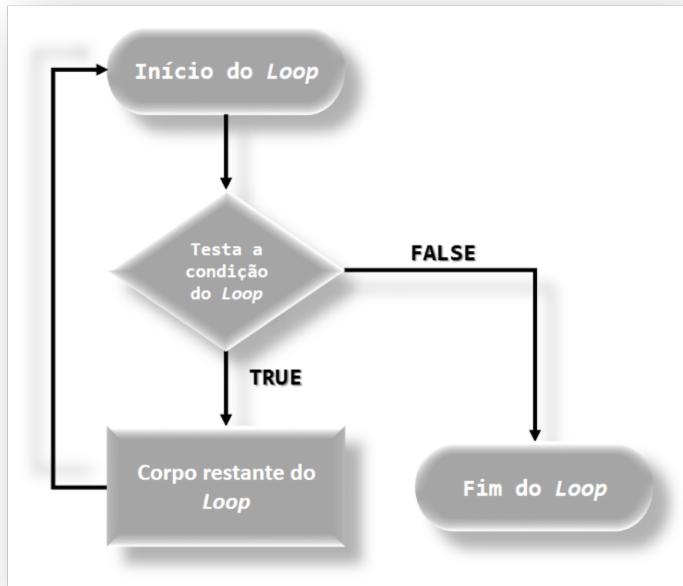


Figura 6.5: Fluxograma do *loop*.

A primeira função *loop* é `repeat`, apresentamos sua sintaxe:

Script:

```
1 repeat {
2   expressão ...
3 }
```

Juntamente com o `repeat`, usamos as funções `break` e `next`, pois a função `repeat` não tem uma condição explícita. Para o entendimento, criamos dois fluxogramas, sendo o primeiro entendendo a função `repeat` junto com a função `break` e o outro a função `repeat` juntamente com a função `next`, dos quais podem ser verificadas pelas Figuras 6.6 e 6.7, respectivamente. Vejamos, alguns exemplos para essas funções, iniciando pelo Código R 6.9.

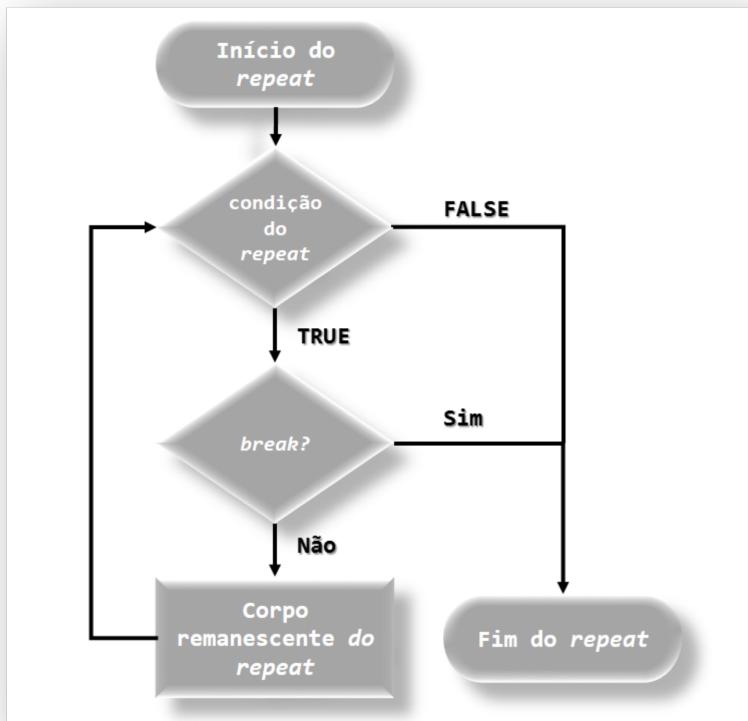
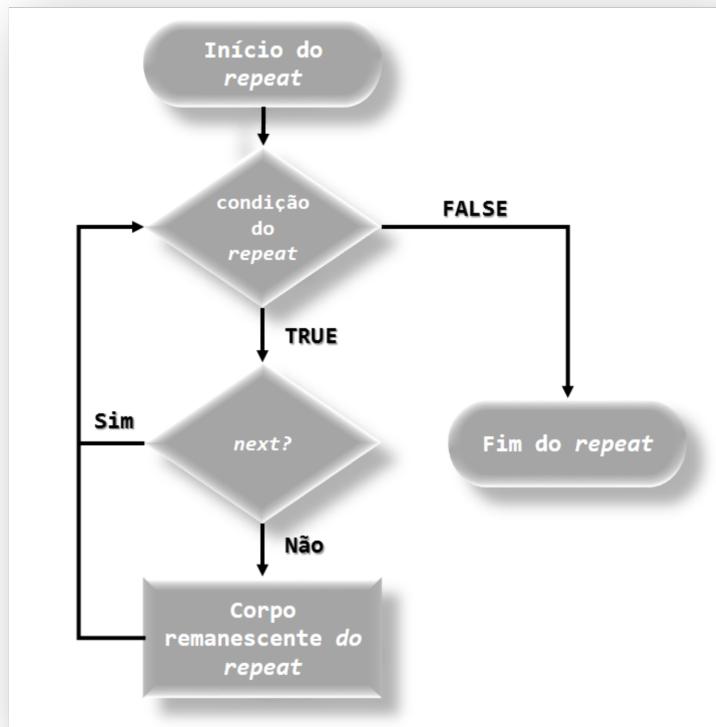


Figura 6.6: Fluxograma das funções `repeat` e `break`.

Figura 6.7: Fluxograma das funções `repeat` e `next`.**Código R 6.9****Script:**

```

1 # Contador
2 i <- 1
3 # Loop repeat
4 repeat {
5   if (i > 5) {
6     break
7   } else {
8     print(i)
9     i <- i + 1
10 }
11 }
  
```

Console:

```

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

Observamos que a condição em `repeat` se repete até $i > 5$, ou seja, será impresso apenas os valores de 1 a 5. Para que isso ocorra, nós quebramos o ciclo de repetição com a função `break`, em que o objeto `i <- 1` [linha 1, Código R 6.9] é o contador do ciclo. Percebemos que cada ciclo apresenta no console como resultado a execução `print(i)` [linha 8, Código R 6.9], e que na sequência, o contador se atualiza, `i <- i + 1` [linha 9, Código R 6.9]. Essa situação se repete enquanto $i < 6$, porque nessa situação a condição em `else` é sempre a executada. Assim, no momento em que o contador `i` é igual a 5, esse valor é impresso no console, e na sequência este é atualizado para `i <- 5 + 1`, isto é, o contador assume valor igual a 6. Nesse momento, o ciclo recomeça e quando a linha 5 é avaliada, a condição $i > 5$ passa a ser verdadeira, então a linha 6 é avaliada e função `break` entra em ação. Isso resulta na quebra do ciclo, e então a função `repeat` é encerrada. O próximo exemplo no Código 6.10, apresentamos a função `next`.

Código R 6.10

Script:

```

1 # Contador
2 i <- 1
3 # Loop repeat
4 repeat {
5   if (i > 5) {
6     break
7   }
8   else {
9     if (i == 3) {
10      i <- i + 1
11      next
12      print(i + 1)
13    }
14    print(i)
15    i <- i + 1
16  }
17 }
```

Console:

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

O que diferencia o Código R 6.9 para o Código R 6.10 é o acréscimo da função `next` ao código. Quando a condição `if(i == 3)` [linha 9, Código R 6.10] é verdadeira, as linhas 10 e 11 são executadas. Primeiro atualizamos o contador `i <- 3 + 1` [linha 10, Código R 6.10] e posteriormente, a função `next` [linha 10, Código R 6.10] é executada. Inserimos a linha 12 no Código R 6.10] para termos a ideia da função `next`, que é avançar o ciclo quando esta função é executada, ou seja, quando o contador quando assume valor 3, as linhas 10 e 11 são executadas, e após a função `next` ser chamada, as linhas 12, 14 e 15 não são executadas, porque se fossem, o valor 5 seria impresso duas vezes, e isso não ocorreu. O que aconteceu foi o avanço de ciclo. Devemos ter uma atenção a função `repeat`, porque como ela não tem um critério de parada explícito, podemos incorrer em um loop infinito.

Similar a função `repeat()`, temos a função `while()`, que agora, temos uma condição de parada, cujo fluxograma é dado pela Figura 6.8, com sintaxe dada por:

Script:

```
1 while (condição) {
2     expressão ...
3 }
```

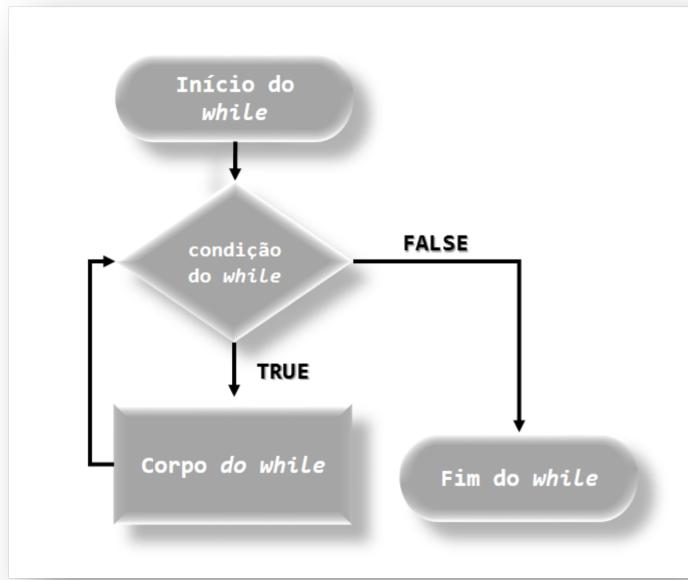


Figura 6.8: Fluxograma da função while.

Nessa função, também podemos usar as funções `break` e `next`, como usados na função `repeat`. Vejamos o exemplo no Código R 6.11.

Código R 6.11**Script:**

```
1 # Contador
2 i <- 1
3 # Loop while
4 while (i <= 5) {
5   print(i)
6   i <- i + 1
7 }
```

Nesse caso, não foi necessário utilizar a função `break`, devido a condição que ela permite impor ao ciclo. Vejamos um outro exemplo, no Código R 6.12.

Código R 6.12

Script:

```

1 # Contador
2 i <- 1
3 # Loop while
4 while (i <= 5) {
5   if (i == 3) {
6     i <- i + 1
7     next
8   }
9   print(i)
10  i <- i + 1
11 }
```

Observamos a utilização da função `next`, similar ao que foi realizado com a função `repeat`. Por fim, a função `for()`, com cuja sintaxe é dada por:

Script:

```

1 for (contador in lista) {
2   expressão ...
3 }
```

Com essa função, também podemos utilizar `break` e `next`. Porém, o controle do ciclo nessa situação é maior, e menos necessário essas funções. Vejamos o Código R 6.9.

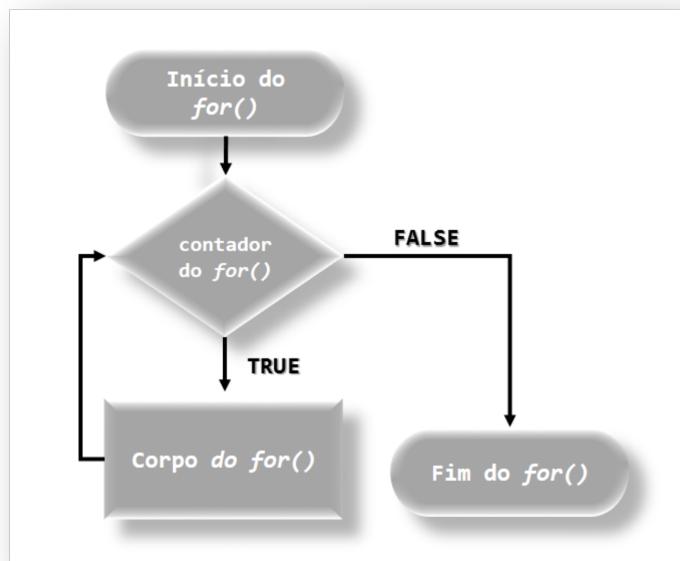


Figura 6.9: Fluxograma da função `for()`.

Código R 6.13**Script:**

```

1 # Loop for
2 for (i in 1:5) {
3   print(i)
4 }
```

Observamos que a implementação, dos algoritmos anteriores, é bem mais simples usando a função `for()`. Vejamos um próximo exemplo, no Código R 6.14.

Código R 6.14**Script:**

```

1 # Loop for
2 for (i in 1:5) {
3   if (i == 3) {
4     next
5   }
6   print(i)
7 }
```

Mais uma vez, foi utilizado a função `next`, sem necessidade agora, de atualizar o contador `i`, uma vez que já está definido na função `for()`. Veremos na próxima subseção, que temos funções de alto nível que podem substituir as funções *loops*. São as chamadas funções da família *apply*.

6.7 Criando funções

Até esse momento, usamos funções já desenvolvidas no **R**, seja dos pacotes nativos, seja via instalação dos pacotes via CRAN. Agora, iremos desenvolver as nossas próprias funções.

Como falado anteriormente, no início, a estrutura da função criada se mantém, argumento, corpo e ambiente. Para isso, usaremos a função `function()`. O modo desse objeto é *closure*⁵. Vejamos a sua sintaxe,

Script:

```

1 # Forma usual
2 nome_funcao <- function(arg1, arg2, ...) {
3   corpo: comandos..
4 }
5 # Forma simplificada
6 nome_funcao <- function(arg1, arg2, ...) corpo
```

Desse modo, apresentamos o primeiro exemplo no Código R 6.15, a seguir.

⁵Use `typeof()` para verificar o modo.

Código R 6.15

Script:

```

1 # Criando a função 'fun1'
2 fun1 <- function(x) {
3   res <- x + 1
4   return(res)
5 }
```

Nesse caso, temos uma função chamada `fun1()`, cujo argumento de entrada é `x`. Observemos que uma função é como um objeto do tipo `vector`, associamos um nome ao objeto da mesma forma. O corpo apresenta uma delimitação por chaves ..., em que apresenta um comando de atribuição, cujo nome `res` se associa ao resultado da soma `x + 1`. Por fim, o resultado dessa função, imprime `res`, por meio da função `return()`. Para executar `fun1()`, fazemos:

Script:

```
1 fun1(x = 5)
```

Console:

```
[1] 6
```

O que aconteceu foi que ao assumir `x = 5` no argumento, essa informação foi repassada para o corpo da função `fun1()`, aonde existia o nome `x`, que se associou ao objeto 5, para esse caso. A função `+` é chamada, e a expressão `x + 1` é avaliada, cujo nome `res` se associa ao resultado dessa expressão. Por fim, quando a função `fun1()` é chamada, o resultado de `res` é impressa no console, por meio de `return(res)`. Vamos verificar os três componentes da função `fun1()` no console, isto é,

Console:

```

> # Argumentos
> formals(fun1)
$x
> # Corpo
> body(fun1)
{
  res <- x + 1
  return(res)
}
> # Ambiente
> environment(fun1)
<environment: R_GlobalEnv>
```

O corpo da função é executado de forma sequencial, a partir da primeira linha de comando até a última. Apesar de recomendado, também não é obrigatório o uso da função `return()`, são as chamadas saídas explícitas⁶, sendo observado como segue,

Script:

```

1 > # Função
2 > fun2 <- function(x) x + 1
3 > # Executando
```

⁶Esse assunto será abordado no Volume II.

```
4 > fun2(5)
5 [1] 6
```

6.7.1 Função anônima

Podemos também ter o que chamamos de **função anônima**, da qual não associamos nome as funções. Contudo, sendo criada esta não pode ser recuperada como qualquer outro objeto. Essa forma é interessante quando não precisamos dela após o seu uso. Por exemplo, queremos calcular a integral,

$$\int_0^1 x^2 dx = \frac{1}{3},$$

e criamos uma função x^2 . Então,

Script:

```
1 integrate(f = function(x) x^2,
2           lower = 0,
3           upper = 1)
```

Console:

```
0.3333333 with absolute error < 3.7e-15
```

A função `integrate()` é utilizada para o cálculo de integral, do qual, passamos os argumentos, função (`f`), limite inferior (`lower`) e limite superior (`upper`) da integração, respectivamente. Observemos que não houve necessidade de nomear a função no argumento, pois não há objetivo de ser reutilizado. Isso também ocorre muito na área de ciência de dados, quando muitas vezes fazemos manipulações com os dados ao mesmo tempo, sem necessidade de associar nomes aos resultados intermediários.

6.7.2 Chamadas de função

As chamadas de funções ocorrem de três formas: aninhado, intermediário e via *pipe*. Vejamos o cálculo do desvio padrão novamente, Código R 6.16.

Código R 6.16

Script:

```
1 # Funcao auxiliar 1
2 aux1 <- function(x) x - mean(x)
3 # Funcao auxiliar 2
4 aux2 <- function(x) x^2
5 # Funcao auxiliar 3
6 aux3 <- function(x) {
7   sum(x) / (length(x) - 1)
8 }
9 # Gerando 100 numeros aleatorios de uma distribuicao normal
10 set.seed(10)
11 x <- rnorm(100)
12 # Calculo do desvio padrao (aninhado)
13 sqrt(aux3(aux2(aux1(x))))
```

Console:

```
[1] 0.9412359
```

Script:

```
13 # Calculo do desvio padrao (intemediario)
14 dp <- aux1(x)
15 dp <- aux2(dp)
16 dp <- aux3(dp)
17 dp <- sqrt(dp)
18 dp
```

Console:

```
[1] 0.9412359
```

Script:

```
19 # Calculo do desvio padrao (pipe)
20 x |>
21   aux1() |>
22   aux2() |>
23   aux3() |>
24   sqrt()
```

Console:

```
[1] 0.9412359
```

Mais detalhes sobre esses três aspectos serão abordados no Volume II.

6.7.3 Ordenação de argumentos

Os argumentos nas funções podem ser nomeados ou não. Quando nomeados, a ordem como são inseridos na função não importa. Já os argumentos não nomeados, seus valores precisam estar na ordem como a função foi desenvolvida. Vejamos o Código R 6.17, para entender melhor.

Código R 6.17**Script:**

```

1 estdesc <- function(x, opcao) {
2   res <- switch(opcao,
3     media = round(mean(x), 4),
4     mediana = round(mean(x), 4),
5     medapar = round(mean(x, trim = 0.1), 4))
6   return(res)
7 }
8 # Objeto
9 set.seed(15)
10 x <- rnorm(1000)
11 # Argumentos nomeados na função
12 estdesc(x = x, opcao = "media")

```

Console:

```
[1] 0.037
```

Script:

```
13 estdesc(opcao = "media", x = x)
```

Console:

```
[1] 0.037
```

Script:

```

14 # Argumentos não nomeados ordenados
15 estdesc(x, "media")

```

Console:

```
[1] 0.037
```

Script:

```

16 # Argumentos não ordenados (Gera erro)
17 estdesc("media", x)

```

Console:

```
Error in switch(opcao, media = round(mean(x), 4), mediana = round(mean(x), : EXPR
deve ser um vetor de comprimento 1
```

6.7.4 Objeto reticências (“...”)

O objeto reticências é do tipo `pairlist`, um tipo de objeto usado bastante internamente no **R**, e dificilmente utilizado no código interpretado. Desse modo, não temos acesso direto a estrutura do

objeto Contudo, esse objeto tem um papel fundamental nas funções, quando damos a liberdade de inserir mais argumentos além dos definidos na criação da função. Esse argumento é usado na função `plot()`, por exemplo, do pacote nativo **base**. Vejamos como o argumento ... pode ser usado em uma função, no Código R 6.18.

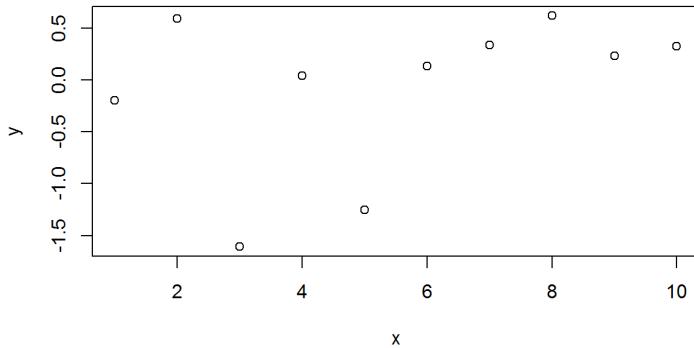
Código R 6.18

Script:

```

1 # Função que plota um gráfico
2 gráfico <- function(x, y, ...) {
3   plot(x = x, y = y, ...)
4 }
5 # Vetores
6 x <- 1:10; y <- rnorm(10)
7 # Chamada 1, com os argumentos definidos
8 gráfico(x = x, y = y)

```

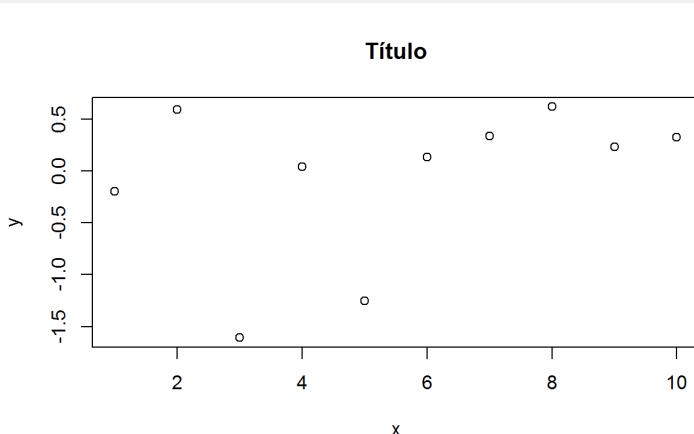


Script:

```

9 # Chamada 2, inserindo argumentos não definidos
10 gráfico(x = x, y = y, main = "Título")

```



Isso ocorre, porque temos a função `plot()` no corpo da função `gráfico()` que apresenta muitos argumentos. Dessa forma, criamos a função `gráfico()`, com os argumentos que representam as

coordenadas, mas a reticências ('...') garantem que os demais argumentos da função plot, omitidos, possam ser utilizados.

6.8 Escopo léxico

Vamos retornar aos componentes da função fun1(), desenvolvido no Código R 6.15, reapresentando novamente as três estruturas de uma função,

Console:

```
> # Argumentos
> formals(fun1)
$x
> # Corpo
> body(fun1)
{
  res <- x + 1
  return(res)
}
> # Ambiente
> environment(fun1)
<environment: R_GlobalEnv>
```

Observemos que o último componente é o ambiente onde o nome fun1 foi associado a função. Este ambiente é chamado de **ambiente envolvente**. Nesse caso é o ambiente global. Contudo, quando a função é executada, momentaneamente é criado o **ambiente de execução**. É neste ambiente que os nomes que estão no corpo da função são associados aos objetos. Vejamos um primeiro exemplo, Código R 6.19.

Código R 6.19

Script:

```
1 x <- 10
2 fun <- function() {
3   x <- 2
4   x
5 }
6 # Chamando a função fun
7 fun()
```

Console:

```
[1] 2
```

Por causa do ambiente de execução que o objeto x dentro da função é retornado, ao invés do que foi definido fora da função. Isso porque o ambiente de execução mascara os nomes definidos dentro da função dos nomes definidos fora da função. Esse é uma primeira característica do **escopo léxico** nas funções em **R**.

Anteriormente, falamos sobre a **atribuição**, que representa a forma como os nomes se associam aos objetos. Agora, o **escopo** vem a ser a forma como os nomes encontram seus valores associados. O termo **léxico** significa que as funções podem encontrar nomes e seus respectivos valores associados, definidos no ambiente onde a função foi definida, isto é, no ambiente de função. Claro que isso segue regras, e a primeira foi a **máscara de nome** falada anteriormente.

Porém quando não existe um nome vinculado a um objeto, e este foi definido no ambiente de função, o valor é repassado para o corpo da função, como pode ser observado no Código R 6.20.

Código R 6.20

Script:

```
1 x <- 10
2 fun <- function() {
3   x
4 }
5 # Chamando a função fun
6 fun()
```

Console:

```
[1] 10
```

O resultado de `fun()` foi 10, porque como a função procurou no ambiente de execuções e não encontrou esse nome, a função foi até o ambiente superior, no caso, o ambiente de execuções. Como falado anteriormente, todo ambiente tem um pai (ou ambiente superior). Essa hierarquização é observada no caminho de busca, que pode ser acessado por `search()`, ou seja,

Console:

```
> search()
[1] ".GlobalEnv"      "package:magrittr"  "package:leaflet"
[4] "package:stats"    "package:graphics" "package:grDevices"
[7] "package:utils"     "package:datasets" "package:methods"
[10] "Autoloads"        "package:base"
```

O ambiente corrente do **R** sempre será o ambiente ambiente global (`.GlobalEnv`). O ambiente de execução não aparece, porque ele é momentâneo. Então, após buscar no ambiente de execução e não encontrar, é pelo caminho de busca que a função irá procurar pelos objetos inseridos no corpo da função. Ele só existe quando a função é chamada e não quando ela é definida, isto é,

Console:

```
> # Função
> fun <- function() x + 10
> # Objeto 1
> x <- 10
> # Chamada 1
> fun()
[1] 20
> # Objeto 2
> x <- 20
> # Chamada 2
> fun()
[1] 30
```

Observe que após ser criada a função `fun()`, o nome `x` se associou ao objeto 10. Posteriormente, `fun()` foi chamada e o resultado foi 20. Contudo, o nome `x` se associou a outro objeto 20, e após a segunda chamada da função `fun()`, o resultado foi 30, porque a função procura os valores quando ela é executada, e não quando é criada, é a característica de pesquisa dinâmica do escopo léxico. Uma outra situação pode ser observada no Código R 6.21

Código R 6.21**Script:**

```

1 # Objeto
2 n <- 1
3 # Funcao
4 fun <- function() {
5   n <- n + 1
6   n
7 }
8 # Chamada 1
9 fun()

```

Console:

```
[1] 2
```

Script:

```

10 # Chamada 2
11 fun()

```

Console:

```
[1] 2
```

Nessas linhas de comando, poderíamos pensar que após ter executado a primeira chamada, o valor retornado seria 2, e a segunda chamada retornaria o valor 3, como ocorre com as variáveis estáticas na linguagem C, por exemplo. Aqui nesse caso, o resultado independente do número de chamadas, será sempre o mesmo porque uma outra característica do escopo léxico no **R** é o **novo começo**, porque a cada vez que a função é executada um novo ambiente de execução é criado, e portanto, cada execução dos comandos de atribuição e expressão são executados de forma independentes nas chamadas de funções.

Algo que não havíamos falado anteriormente, é que a função `function()` não necessariamente necessita de definição de argumentos, devido a flexibilidade do escopo léxico das funções em **R**. É essa característica que faz com que os comandos no corpo das funções encontrem os objetos que não estão definidos na própria função. Mais detalhes, serão encontrado no *Volume II: Nível Intermediário* da coleção *Estudando o Ambiente R*.

6.9 Exercícios

Exercício 6.1:

Solução na página 138

Gráficos no R



7.1 Exercícios

Exercício 7.1:

Solução na página 139

Boas práticas de como escrever um código

8.1 Introdução

Nesse momento, entendemos os principais objetos para escrevermos os nossos *scripts*. Quando escrevemos um código, duas consequências ocorrem:

- guardá-lo para futuras consultas, ou
- compartilhamento.

Nesses dois casos, percebemos que alguém irá ler esse código, ou até mesmo o próprio usuário, irá retornar àquelas linhas de código e tentar raciocinar quais as ideias por trás disso tudo. Para um melhor entendimento de seu *script*, nada mais importante do que uma boa escrita, separação das estruturas por hierarquização, comentários, etc.

Uma primeira ferramenta que pode ser configurada para quem usa o **RStudio** é acionar todas as opções de diagnóstico do seu código. Para isso no menu:

- Tools > Global options > Code > Editing. Marque todas as opções em General;
- Tools > Global options > Code > Display, Marque todas as opções;
- Tools > Global options > Code > Diagnostics. Marque todas as opções em R Diagnostics.

Com isso, colorações nas linhas de comando ocorrerão, distinguindo diversas estruturas, como linhas de comentário, funções, espaçamentos, dentre outras coisas.

Uma vez feito isso, vamos para o passo seguinte que são as boas práticas de como se escrever um *script*. Temos algumas ferramentas prontas, como o pacote **styler** e como alternativa o pacote **formatR**, que automatiza todo o nosso código seja em *script*, contido em um pacote, ou diretório. Acesse <https://yihui.org/formatr>, para mais detalhes. Para instalar e anexar o pacote **styler**, use as linhas de comando:

Console:

```
> # Instalando pacote  
> install.packages(styler)  
> # Carregando e anexando  
> library(styler)
```

Vejamos a Figura 8.1, para entendermos a funcionalidade desse pacote.

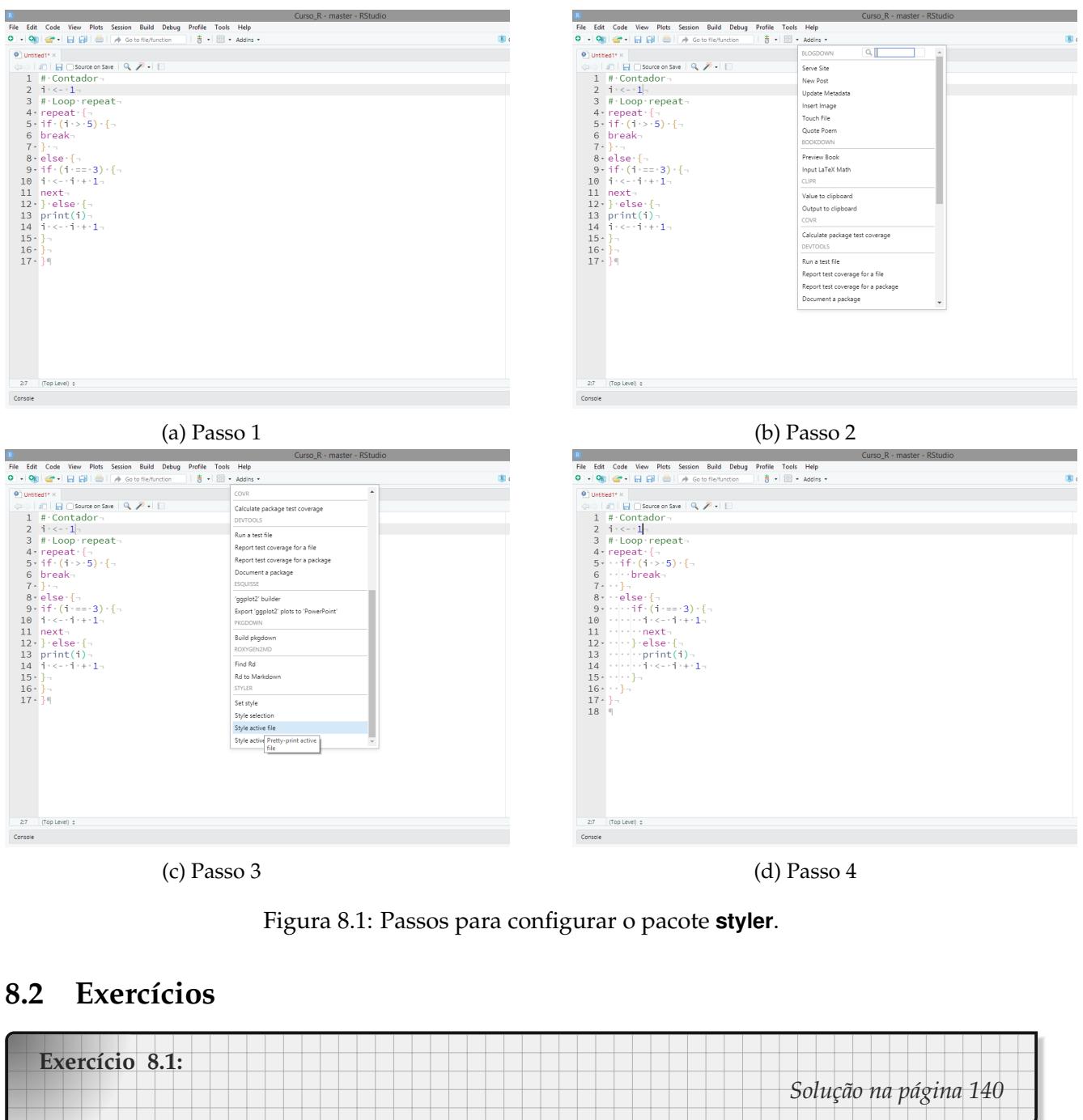


Figura 8.1: Passos para configurar o pacote **styler**.

8.2 Exercícios

Exercício 8.1:

Solução na página 140

Pacotes

9.1 Introdução

]

Um pacote em **R** é um diretório de arquivos necessários para carregar um código de funções, dados, documentações de ajuda, testes, etc. O próprio **R** em sua instalação, contém 30 pacotes nativos, que contém as funções mínimas para a utilização do ambiente. No pacote, não há apenas códigos em **R**, mas um pacote fonte (do inglês, *source package*), contendo os arquivos mencionados acima, ou um arquivo compactado, de extensão *.tar.gz* do pacote fonte, ou um pacote instalado, resultado do comando **R CMD INSTALL**, que será visto no *Volume V: Desenvolvimento de Pacotes R*, coleção *Estudando o Ambiente R*. Isso acontece no SO *Linux*. Para as plataformas *Windows* e *Macintosh*, existem também os pacotes binários ou compactados com a extensão *.zip* ou *.tgz*, respectivamente.

Um pacote, portanto é a unidade básica para o compartilhamento de um código. Atualmente, até 01/12/2021, o número de pacotes disponíveis no *CRAN* é 18.524. Podemos encontrar a lista de pacotes por data de publicação ou por ordem alfabética. Qualquer usuário pode publicar um pacote e disponibilizá-lo sob o *CRAN*. Para isso, uma série de testes iniciais são realizados no próprio ambiente **R**, para verificar se o pacote em desenvolvimento não contém problemas previsíveis, e posteriormente, uma checagem mais aprofundada, após a submissão, é realizada por algum dos mantenedores do **R** (*R Development Core Team*). Isso significa dizer que se um pacote está disponível no *CRAN*, além de sua estabilidade, o pacote será executável nas três plataformas mais usadas em sistema operacional, *SO Linux* ou sistemas *Unix*, *SO Windows* e *SO Macintosh*. Isso é um padrão hoje no **R**.

Há outros repositórios que podem ser disponibilizados os pacotes, como por exemplo, Bioconductor, R-forge, GitHub. Este último está sendo muito utilizado, nesses últimos anos. Porém, quando os pacotes estão em repositórios diferentes do *CRAN*, não haverá garantias dos padrões mencionados anteriormente. Além do mais, devemos entender que a principal preocupação dos mantenedores do **R**, é que os pacotes funcionem corretamente, naquilo em que os mesmos objetivam, mas em nada é discutido sobre a metodologia científica, ou de análises, a que o pacote se destina. Isso é verificado, quando o pacote após disponível no *CRAN*, também é submetido para o journal do **R**, o *The R Journal* ou ao *Journal of Statistical Software*, por exemplo. Aí sim, o pacote é esmiuçado tanto no aspecto computacional, quanto no metodológico.

Portanto, entenda que nem todo pacote sob o *CRAN* é confiável naquilo que se propõe, o que não significa dizer, que um pacote é confiável se apenas tiver sido publicado nas revistas anteriores ou em qualquer outra específica para a área de interesse, mas a cautela é sempre necessária, tentando entender quem são os desenvolvedores, ou fazendo uma pesquisa mais ampla sobre o referido pacote, para que assim, a decisão da escolha seja confiável. Fazemos esse adendo, porque quando se vai pesquisar sobre um determinado assunto poderá existir diversos pacotes para o mesmo, e a pergunta será, qual pacote devemos escolher? Aquele que é mais fácil utilizar? E daí, uma boa pesquisa para a escolha não deve ser levado apenas em consideração a facilidade de utilização do pacote, mas saber se as funções do pacote realmente retornam os resultados confiáveis para aquilo que se destina. Fica a nossa dica.

Para uma instalação mais rápida dos pacotes, optem pelos espelhos disponíveis nos países em que vivem, uma vez que a transferência de dados ocorrem mais rapidamente. Aqui no Brasil, por

exemplo, o primeiro espelho desenvolvido e ativo até hoje é o da UFPR. Mas, temos mais quatro espelhos: dois na USP, uma na Fiocruz/RJ, e outra na UESC.

Um termo que deve ficar claro, que erroneamente, alguns usuários chamam o termo biblioteca como um sinônimo de pacote. Nas documentações do **R**, biblioteca é o diretório onde os pacotes são instalados, também chamados de diretório de biblioteca ou diretório de árvores. O outro sentido de biblioteca é o de biblioteca compartilhada (dinâmica ou estática), que armazenam código compilado que se vinculam aos pacotes, por exemplo, no Windows são as *DLLs*.

9.2 Estrutura básica de um pacote

A estrutura básica de um pacote é apresentada na Figura 9.1.

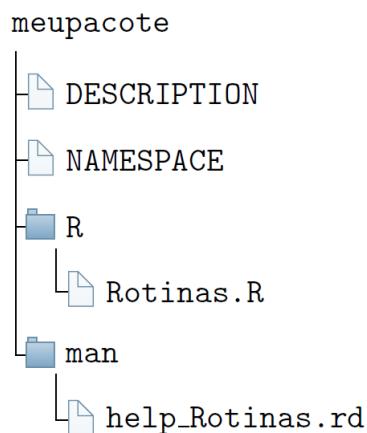


Figura 9.1: Esqueleto básico de um pacote.

Vejamos as ideias básicas desses subdiretórios e arquivos:

- DESCRIPTION:** Esse é um arquivo de texto, contendo informações básicas como o título do pacote, versão, licença, descrição, nome dos autores, e o mantenedor do pacote, isto é, para quando um pacote estiver com problema ou o *CRAN* entre em contato, será para este último. Essas as informações obrigatórias que devem ter nesse arquivo;
- NAMESPACE:** Esse arquivo embora tenha muita semelhança com a linguagem **R**, o seu conteúdo, se destina a importação e exportação de funções no pacote. Será nesse arquivo, que diremos quais os pacotes que ele depende, isto é, as funções, e quais as funções exportadas, visíveis, que devem ser apresentadas aos usuários;
- R/:** Esse subdiretório apresenta os *scripts* com as funções em **R**. é o cérebro do pacote;
- man/:** Esse subdiretório apresenta os arquivos de ajuda, com extensão *.Rd*. Isso significa, que uma vez instalado o pacote no **R**, o acesso aos manuais de ajuda do referido pacote, estarão disponíveis, graças a esses arquivos.

Claro, que quando os pacotes se tornam mais complexos, outros subdiretórios e arquivos são necessário. Mas isso é assunto para o *Volume V: Desenvolvimento de Pacotes R*, coleção *Estudando o Ambiente R*.

9.3 Instalação de um pacote

A instalação de um pacote via **R** pode ser feito pela função `install.packages(pkgs = "nome_pacote")`. Por exemplo, vamos tentar instalar o pacote **midrangeMCP**, da seguinte forma:

Console:

```
> install.packages("midrangeMCP")
```

Pode ser que nesse processo, dependendo de onde o usuário esteja executando essa linha de comando, interface do **R** ou **RStudio**, que seja solicitado o espelho por onde deseja fazer a instalação. Isso é apenas um atalho para ter um acesso mais rápido na instalação do pacote. A sugestão é escolher um espelho de seu país de origem.

Uma forma simples de se ter detalhes do pacote na internet, tais como, baixar o pacote fonte ou o pacote binário do **midrangeMCP**, por exemplo, é sempre usar essa url: <<http://cran.r-project.org/package=midrangeMCP>>. Para qualquer outro pacote, basta mudar o nome do pacote na url, e assim, estaremos na página do repositório do pacote. O pacote fonte, como falado anteriormente, é compactado com extensão *.tar.gz*, no caso, *midrangeMCP_3.1.1.tar.gz*. O pacote binário tem a compactação zipada, *midrangeMCP_3.1.1.tar.zip* para o Windows e *midrangeMCP_3.1.1.tgz* para o Macintosh. O acesso aos arquivos do pacote mencionados no esqueleto são disponíveis no pacote fonte.

Uma outra forma possível de instalação é baixar o arquivo do pacote fonte para o seu computador e instalá-lo, via comando:

Console:

```
> install.packages(pkgs = "./midrangeMCP.tar.gz", repos = NULL, type = "source")
```

Consideramos que o arquivo do pacote esteja no diretório de trabalho do usuário. Caso contrário, deve ser informado o local onde pacote se encontra no computador. Para o Window ou Macintosh, é possível instalar também, a partir dos pacotes binários.

Muitos dos desenvolvedores, estão disponibilizando seus projetos de pacotes, principalmente no **GitHub**, inclusive com manuals de ajuda com maiores detalhes. Pode ser possível instalar esses pacotes por esse repositório. Precisamos inicialmente do pacote **devtools**, e posteriormente a instalação do pacote. Segue as linhas de comando:

Console:

```
> install.packages("devtools")
> install_github("bendeivide/midrangeMCP")
```

Contudo, devemos dar a preferência pela instalação via *CRAN*. Por lá, teremos a garantia que os pacotes estão estáveis para a utilização nas referidas plataformas mencionadas acima.

Alguns pacotes, por falta de manutenção, seja por atualizações do **R** ou por qualquer outro motivo, podem se tornar incompatíveis para utilização sobre alguns dos três sistemas operacionais básicos (*SO Windows*, *Unix* e *SO Mac*) exigidos pelo **R**. Dessa forma, se as correções não forem feitas, estes pacotes se tornam órfãos, ou seja, desativados sob o *CRAN*. O primeiros pacotes sob o *CRAN*, por exemplo, não tinham o arquivo **NAMESPACE**, que hoje é exigido. Qualquer tentativa de instalação desses pacotes nessas situações, não serão bem sucedidas. Dessa forma, fizemos uma vídeo-aula, como tentativa de recuperar os pacotes desativados. Porém, deixemos claro que nem sempre é possível a instalação de pacotes desativados.

9.4 Objetivos de um pacote

A ideia de um pacote para um usuário **R** deve representar como uma ferramenta para otimizar suas atividades do dia-a-dia na utilização da linguagem. Suponha que o usuário seja um cientista de dados, e todos os dias ele carrega uma sequência de *scripts*, via *source*, para disponibilizar suas funções no ambiente global. Isso acaba gerando processos repetitivos de trabalho desnecessários.

Ao invés, o cientista de dados pode desenvolver um pacote, e esse pacote conter todas as funções necessárias para as suas análises. De uma vez, o pacote instalado e anexado no caminho de busca,

todas as suas funções estarão disponíveis para utilização. Portanto, o entendimento disso, permite uma maior eficiência de trabalho.

Outro ponto é que a experiência contida em um pacote pode ser propagada mais facilmente para outros usuários, mostrando que o conhecimento é uma liberdade necessária. Tanto pelo CRAN, quanto por outras plataformas, o pacote pode ser disponibilizado.

9.5 Utilizar as funções de um pacote

Uma vez instalado o pacote, precisamos carregar e anexá-lo, para que possamos utilizar os recursos disponíveis no pacote, como funções, dados, etc. Isso significa, disponibilizar na memória e inseri-lo no caminho de busca, respectivamente. Para fazer essas duas ações ao mesmo tempo, use a função `library()` ou `require()`. A primeira função se for utilizada sem argumento algum, retorna todos os pacotes instalados na biblioteca de pacotes do R . Vejamos o exemplo do pacote **midrangeMCP**, no Código R 9.1.

Código R 9.1

Script:

```

1 > # Carregando e anexando o pacote midrangeMCP
2 > library(midrangeMCP)
3 > # Usando a função MRtest() desse pacote
4 > #-----
5 > # Dados simulados de um experimento em DIC (Delineamento Inteiramente
   # Casualizado)
6 > # Variavel resposta
7 > rv <- c(100.08, 105.66, 97.64, 100.11, 102.60, 121.29, 100.80,
8 +         99.11, 104.43, 122.18, 119.49, 124.37, 123.19, 134.16,
9 +         125.67, 128.88, 148.07, 134.27, 151.53, 127.31)
10 > # Tratamento
11 > treat <- factor(rep(LETTERS[1:5], each = 4))
12 > # Anava
13 > res    <- anova(aov(rv~treat))
14 > DFerror <- res$Df[2]
15 > MSerror <- res$`Mean Sq`[2]
16 > # Aplicando testes
17 > results <- midrangeMCP::MRtest(y = rv,
18 +             trt = treat,
19 +             dferror = DFerror,
20 +             mserror = MSerror,
21 +             alpha = 0.05,
22 +             main = "PCMs",
23 +             MCP = c("all"))

```

Console:

```
MCP's based on distributions of the studentized midrange and range
```

```
Study: PCMs
```

```
Summary:
```

	Means	std.r	Min	Max
A	100.87	3.40	4	97.64 105.66
B	105.95	10.33	4	99.11 121.29
C	117.62	9.02	4	104.43 124.37
D	127.97	4.74	4	123.19 134.16
E	140.30	11.42	4	127.31 151.53

Console:

```
Mean Grouping Midrange Test
```

```
Statistics:
```

Exp.Mean	CV	MSerror	Df	n	Stud.Midrange	Ext.DMS	Int.DMS
118.542	7.08182	70.47488	15	5	1.089968	5.90246	4.575105

```
Groups:
```

Means	Groups
E 140.30	g1
D 127.97	g2
C 117.62	g3
B 105.95	g4
A 100.87	g4

Console:

```
Mean Grouping Range Test
```

```
Statistics:
```

Exp.Mean	CV	MSerror	Df	n	Stud.Range	DMS
118.542	7.08182	70.47488	15	5	4.366985	18.33027

```
Groups:
```

Means	Groups
E 140.30	g1
D 127.97	g2
C 117.62	g2
B 105.95	g3
A 100.87	g3

Console:

```
SNK Midrange Test
```

Statistics:

	Exp.Mean	CV	MSerror	Df	n	Stud.Midrange	DMS
comp1	118.542	7.0818	70.4749	15	5	1.0900	5.9025
comp2	118.542	7.0818	70.4749	15	4	1.1646	6.2159
comp3	118.542	7.0818	70.4749	15	3	1.2828	6.7121
comp4	118.542	7.0818	70.4749	15	2	1.5072	7.6536

Groups:

	Means	Groups
E	140.30	g1
D	127.97	g1
C	117.62	g1g2
B	105.95	g1g2
A	100.87	g2

Console:

```
Tukey Midrange Test
```

Statistics:

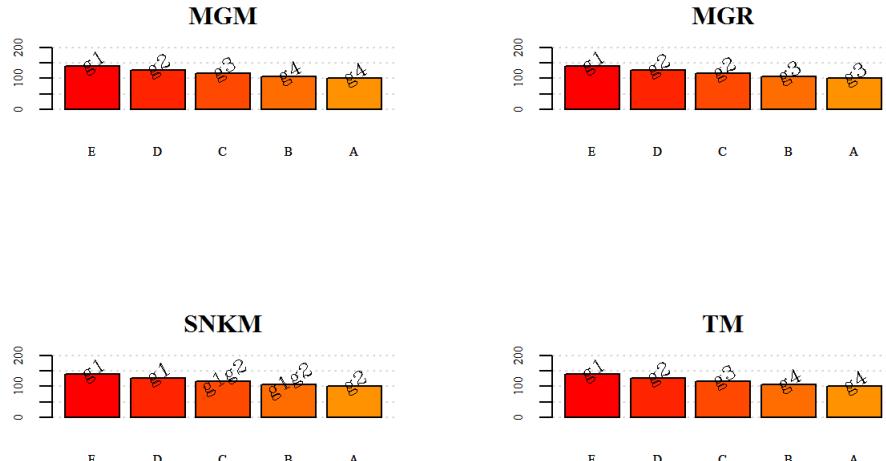
	Exp.Mean	CV	MSerror	Df	n	Stud.Midrange	Ext.DMS	Int.DMS
	118.542	7.08182	70.47488	15	5	1.089968	5.90246	4.575105

Groups:

	Means	Groups
E	140.30	g1
D	127.97	g2
C	117.62	g3
B	105.95	g4
A	100.87	g4

Script:

```
10 midrangeMCP::MRbarplot(results)
```



9.6 Carregando e anexando um pacote

Anteriormente, falamos que usamos a função `library()` ou `require()` para carregar e anexar um pacote para utilizar suas funções, após a instalação. Carregar um pacote significa disponibilizar na memória ativa. Para acessar uma função de um pacote após ter sido carregado, usamos o operador `::`, isto é, `nome_pacote::nome_função`. Isto significa, que será chamado a função necessária sem anexar o pacote no caminho de busca. Estudaremos no Capítulo 10, um pouco mais sobre caminho de busca. Para esse momento, entendamos que é um caminho hierarquizado de ambientes, isto é, objetos que armazenam, em forma de lista, nomes associados a objetos. A função para ver o caminho de busca é `search()`, como pode ser observado no Código R 9.2.

Código R 9.2

Script:

```
1 # Caminho de busca
2 search()
```

Console:

```
[1] ".GlobalEnv"      "package:magrittr"   "package:leaflet"
[4] "package:stats"    "package:graphics"  "package:grDevices"
[7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"        "package:base"
```

Script:

```
3 # Carregando e chamando uma função de um pacote
4 midrangeMCP::MRwrite(results, extension = "latex")
```

Console:

```
Table in latex of results of the MGM test

% latex table generated in R 4.1.0 by xtable 1.8-4 package
% Tue Nov 09 13:21:55 2021
\begin{table}[ht]
\centering
\begin{tabular}{lrl}
\hline
trt & Means & Groups \\
\hline
E & 140.30 & g1 \\
D & 127.97 & g2 \\
C & 117.62 & g3 \\
B & 105.95 & g4 \\
A & 100.87 & g4 \\
\hline
\end{tabular}
\end{table}
```

Console:

```
Table in latex of results of the MGR test

% latex table generated in R 4.1.0 by xtable 1.8-4 package
% Tue Nov 09 13:21:55 2021
\begin{table}[ht]
\centering
\begin{tabular}{lrl}
\hline
trt & Means & Groups \\
\hline
E & 140.30 & g1 \\
D & 127.97 & g2 \\
C & 117.62 & g2 \\
B & 105.95 & g3 \\
A & 100.87 & g3 \\
\hline
\end{tabular}
\end{table}
```

Console:

```
Table in latex of results of the SNKM test

% latex table generated in R 4.1.0 by xtable 1.8-4 package
% Tue Nov 09 13:21:55 2021
\begin{table}[ht]
\centering
\begin{tabular}{lrl}
\hline
trt & Means & Groups \\
\hline
E & 140.30 & g1 \\
D & 127.97 & g1 \\
C & 117.62 & g1g2 \\
B & 105.95 & g1g2 \\
A & 100.87 & g2 \\
\hline
\end{tabular}
\end{table}
```

Console:

```
Table in latex of results of the TM test

% latex table generated in R 4.1.0 by xtable 1.8-4 package
% Tue Nov 09 13:21:55 2021
\begin{table}[ht]
\centering
\begin{tabular}{lrl}
\hline
trt & Means & Groups \\
\hline
E & 140.30 & g1 \\
D & 127.97 & g2 \\
C & 117.62 & g3 \\
B & 105.95 & g4 \\
A & 100.87 & g4 \\
\hline
\end{tabular}
\end{table}
```

Console:

```
Table in latex of results of descriptive statistics

% latex table generated in R 4.1.0 by xtable 1.8-4 package
% Tue Nov 09 13:21:55 2021
\begin{table}[ht]
\centering
\begin{tabular}{lrrrrr}
\hline
trt & Means & std & r & Min & Max \\
\hline
A & 100.87 & 3.40 & 4.00 & 97.64 & 105.66 \\
B & 105.95 & 10.33 & 4.00 & 99.11 & 121.29 \\
C & 117.62 & 9.02 & 4.00 & 104.43 & 124.37 \\
D & 127.97 & 4.74 & 4.00 & 123.19 & 134.16 \\
E & 140.30 & 11.42 & 4.00 & 127.31 & 151.53 \\
\hline
\end{tabular}
\end{table}
```

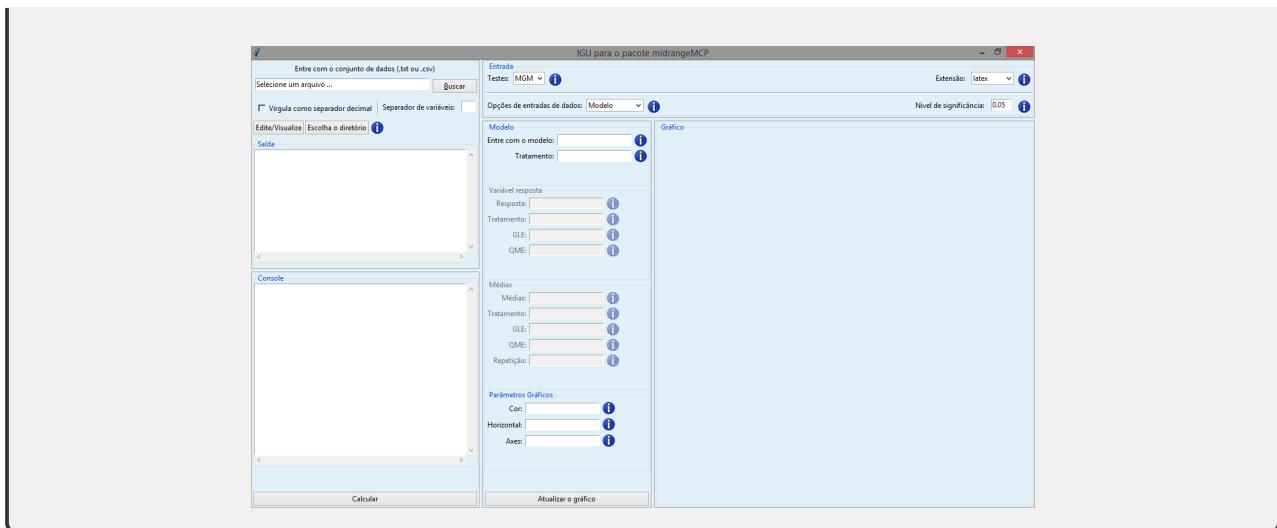
See yours tables in Console

Format: latex

Com as linhas de comando apresentadas anteriormente, percebemos ao executar a função `MRwrite()` do pacote **midrangeMCP**, usando `::` que o caminho de busca não foi alterado. Isso significa que o pacote não foi anexado, apenas carregado, ou seja, se o usuário desejar usar alguma função do pacote digitando apenas o nome no console, não será possível, porque o pacote não está anexado ao caminho de busca. Vejamos outra situação pelo Código R 9.3.

Código R 9.3**Console:**

```
> # Caso o pacote midrangeMCP esteja anexado, use:
> # detach("package:midrangeMCP", unload = TRUE)
> # Caminho de busca
> search()
[1] ".GlobalEnv"      "package:magrittr"  "package:leaflet"
[4] "package:stats"    "package:graphics" "package:grDevices"
[7] "package:utils"    "package:datasets" "package:methods"
[10] "Autoloads"        "package:base"
> # Carregando e anexando um pacote
> library(midrangeMCP)
> # Verificando novamente o caminho de busca
> search()
[1] ".GlobalEnv"      "package:midrangeMCP" "package:magrittr"
[4] "package:leaflet"  "package:stats"     "package:graphics"
[7] "package:grDevices" "package:utils"     "package:datasets"
[10] "package:methods"  "Autoloads"       "package:base"
> # Chamando uma função do pacote
> guimidrangeMCP()
```



Com o uso da função `library()`, percebemos que o caminho de busca foi alterado, porque agora temos o ambiente de pacote `package:midrangeMCP`. Isso significa que agora poderemos acessar os objetos desse pacote apenas digitando o nome associado a eles. Por fim, a última linha de comando, representa a interface gráfica ao usuário para o pacote, o que chamamos de *GUI* (do inglês, *Graphical User Interface*).

9.7 NAMESPACE de um pacote

No início da seção sobre pacotes, falamos sobre o esqueleto de um pacote, isto é, os componentes básicos de um pacote. Um dos arquivos foi o **NAMESPACE**. Esse arquivo é responsável pela exportação e importação de funções. As funções exportadas de um pacote, por meio desse arquivo, são aquelas visíveis após a anexação do pacote ao caminho de busca, ou por meio do operador `::`. As funções importadas são aquelas utilizadas de outros pacotes, utilizadas internamente ao referido pacote.

As funções ditas internas são aquelas não mencionadas no **NAMESPACE**. Em muitas situações, precisamos de funções internas necessárias para a finalidade do pacote, que muitas vezes não é objetivo final para disponibilidade dos usuários, mas códigos intermediários para a boa funcionalidade do pacote. Dessa forma, uma boa escolha para que não haja conflitos em nomes associados a objetos no ambiente de trabalho, é a decisão de não exportá-los.

Porém, quando se cria um pacote, por exemplo, pelo **RStudio**, o padrão no **NAMESPACE** é o comando: `exportPattern(''^[^\.\.]''')`, que significa que todas as funções no pacote serão exportadas que não iniciam por um ponto ('`.`').

9.8 Documentações de um pacote

Toda função exportada de um pacote precisa de um arquivo de ajuda (`.Rd`). Rodas as funções deverão ter esses tipos arquivos inseridos no subdiretório `man/`. Mais detalhes sobre o desenvolvimento de pacotes será abordado no *Volume V: Desenvolvimento de Pacotes R*, coleção *Estudando o Ambiente R*.

9.9 Operadores `::` e `:::`

Como falamos anteriormente, para chamarmos uma função sem a necessidade de anexar o pacote, usamos o operador `::`. Comentamos também, que algumas funções não eram exportadas pelo **NAMESPACE** de um pacote. Contudo, se desejarmos visualizar ou executá-las, poderemos utilizar o operador '`:::`'. Vejamos um exemplo, nas linhas de comando a seguir.

Console:

```
> # Instale o pacote SMR
> # install.packages(SMR) # Descomente a linha de comando para instalar
> # Carregando e chamando funções exportadas do pacote SMR
> SMR::pSMR(q = 2, size = 10, df = 3)
[1] 0.9905216
> # Carregando e chamando funções não exportadas ao pacote
> SMR:::GaussLegendre(size = 4)
$nodes
[1] -0.8611363 -0.3399810  0.3399810  0.8611363

$weights
[1] 0.3478548 0.6521452 0.6521452 0.3478548
```

As funções internas dos pacotes devem ser utilizadas com muita cautela, uma vez que são funções que podem passar por atualizações, mudanças. Isso porque, como não são funções exportadas, alguns pacotes podem passar por atualizações, e desse modo, estas funções também podem ser atualizadas ou até mesmo alteradas.

Outro ponto interessante é que não se recomenda a utilização de importação de funções internas de outros pacotes no desenvolvimento de pacotes, uma vez que são funções que podem passar por mudanças drásticas, e portanto, gerar problemas nas rotinas. Se uma função em um pacote não foi exportada, é porque o desenvolvedor tem um bom motivo para tal situação. As funções exportadas são de fato a essência do objetivo de um pacote, e por isso que elas são exportadas.

9.10 Exercícios

Exercício 9.1:

Solução na página 141

Ambientes e caminho de busca

10.1 Introdução



10.2 Exercícios

Exercício 10.1:

Solução na página 142

Interfaces com outras linguagens

11.1 Introdução



11.2 Exercícios

Exercício 11.1:

Solução na página 143

Considerações e preparação para o Volume II

12.1 Introdução



12.2 Exercícios

Exercício 12.1:

Solução na página 144

Gabarito dos Exercícios

Solução dos Exercícios do Capítulo 1

Solução do Exercício 1.1 na página 5:

A resposta está relacionada ao escopo léxico das funções e a superatribuição! Ainda não entendeu? Então leia os Capítulos 2 e 4. Para uma maior profundidade, veja o Capítulo sobre ambientes no Volume II.

Solução do Exercício 1.2 na página 5:

Para entender, leia o Capítulo 8.

Solução do Exercício 1.3 na página 6:

Para entender, leia o Capítulo 7.

Solução do Exercício 1.4 na página 6:

Para entender, leia o Capítulo 7.

Solução do Exercício 1.5 na página 6:

Para entender, leia o Capítulo 7.

Solução do Exercício 1.6 na página 6:

Se respondeu 1, leia todo o livro.

Solução do Exercício 1.7 na página 6:

Para entender, leia o Capítulo 2.

Solução do Exercício 1.8 na página 6:

Leia todo o livro.

Solução do Exercício 1.9 na página 6:

Para entender, leia o Capítulo 4.

Solução do Exercício 1.10 na página 6:

Para entender, leia o Capítulo 4.

Solução do Exercício 1.11 na página 6:

Para entender, leia o Capítulo 3.

Solução do Exercício 1.12 na página 6:

Para entender, leia o Capítulo 4.

Solução do Exercício 1.13 na página 6:

Para entender, leia o Capítulo 4.

Solução do Exercício 1.14 na página 7:

Para entender, leia o Capítulo 4.

Solução do Exercício 1.16 na página 7:

Para entender, leia o Capítulo 4.

Solução do Exercício 1.16 na página 7:

Para entender, leia o Capítulo 5.

Solução do Exercício 1.17 na página 7:

Para entender, leia o Capítulo 6.

Solução do Exercício 1.18 na página 7:

Para entender, leia o Capítulo 6.

Solução do Exercício 1.19 na página 7:

Para entender, leia os Capítulos 3 e 10.

Solução do Exercício 1.20 na página 7:

Para entender, leia os Capítulos 3 e 10.

Solução do Exercício 1.21 na página 7:

Para entender, leia os Capítulos 4.

Solução do Exercício 1.22 na página 7:

Para entender, leia o Capítulo 3.

Solução dos Exercícios do Capítulo 2

Solução dos Exercícios do Capítulo 3

Solução do Exercício 3.1 na página 25:

Solução dos Exercícios do Capítulo 4

Solução do Exercício 4.1 na página 71:

Solução dos Exercícios do Capítulo 5

Solução do Exercício 5.1 na página 81:

Solução dos Exercícios do Capítulo 6

Solução do Exercício 6.1 na página 108:

Solução dos Exercícios do Capítulo 7

Solução do Exercício 7.1 na página 110:

Solução dos Exercícios do Capítulo 8

Solução do Exercício 8.1 na página 112:

Solução dos Exercícios do Capítulo 9

Solução do Exercício 9.1 na página 125:

Solução dos Exercícios do Capítulo 10

Solução do Exercício 10.1 na página 127:

Solução dos Exercícios do Capítulo 11

Solução do Exercício 11.1 na página 129:

Solução dos Exercícios do Capítulo 12

Solução do Exercício 12.1 na página 131:

Referências Bibliográficas

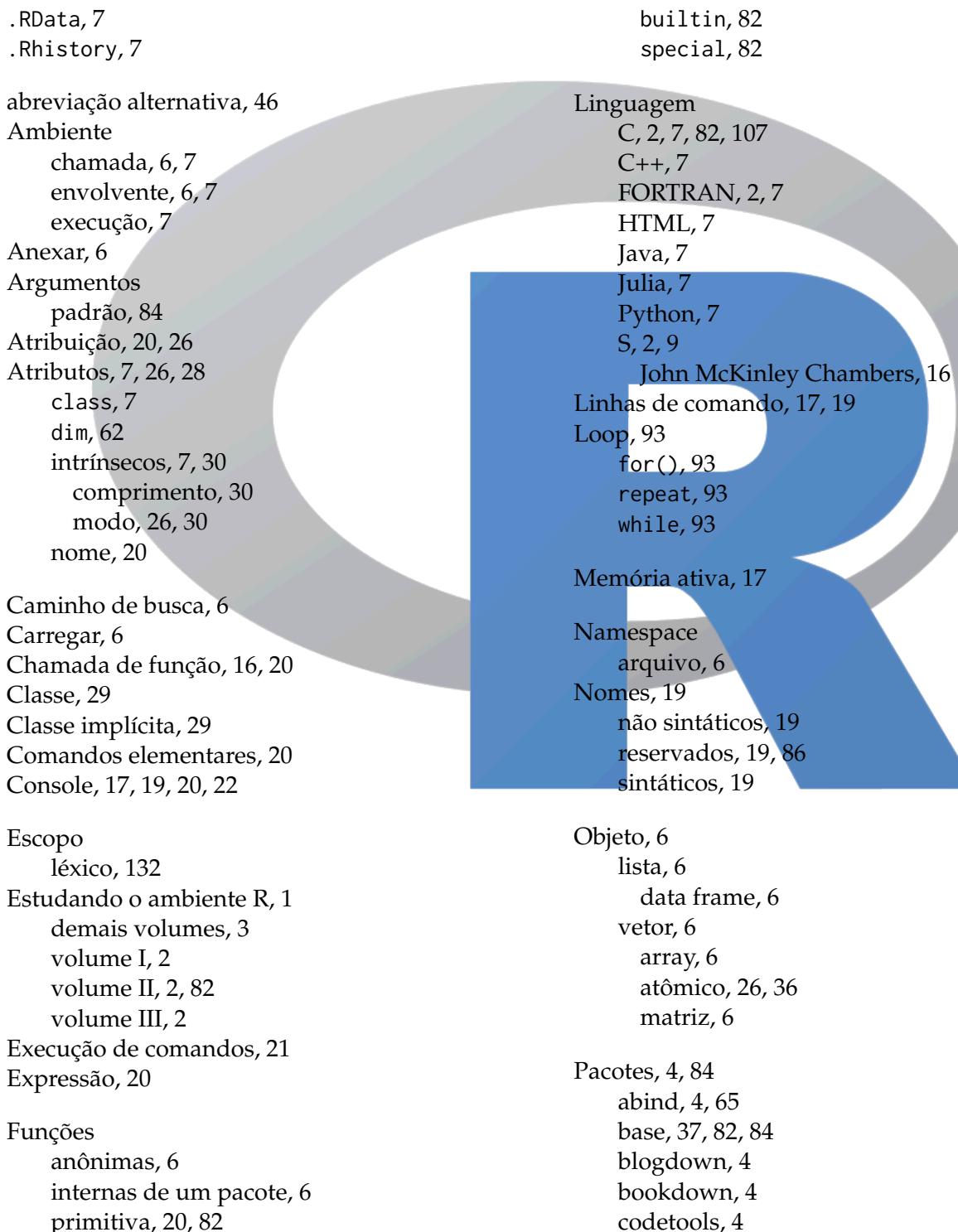
- ADLER, J. *R in a Nutshell*. Sebastopol: O'Reilly Media, 2012. ISBN 978-1-449-31208-4.
- BECKER, R. A.; CHAMBERS, J. M.; WILKS, A. R. *The New S Language*: A programming environment for data analysis and graphics. Boca Raton, Flórida: CRC Press, 1988.
- CHAMBERS, J. M. *Software for Data Analysis*: Programming with R. New York: Springer, 2008. (Statistics and Computing). ISBN 978-0-387-75935-7.
- CHAMBERS, J. M. *Extending R*. Boca Raton, Florida: Chapman and Hall/CRC, 2016. (The R Series). ISBN 978-1-4987-7572-4.
- CHAMBERS, J. M.; HASTIE, T. J. *Statistical Methods in S*. London: Chapman & Hall, 1991.
- CHAMBERS, J. M.; HASTIE, T. J. *Programming with Data*: A guide to the s language. Ney York: Springer, 1998.
- CHANG, W. *R Graphics Cookbook*. Sebastopol: O'Reilly Media, 2018. 444 p. ISBN 978-1491978603. Disponível em: <<https://r-graphics.org/>>.
- GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, v. 23, n. 1, p. 5–48, 1991.
- GROLEMUND, G. *Hands-On Programming with R: Write Your Own Functions and Simulations*. Sebastopol: O'Reilly Media, 2014. ISBN 9781449359119. Disponível em: <<https://rstudio-education.github.io/hopr/>>.
- GROSSER, M.; BUMAN, H.; WICKHAM, H. *Advanced R Solutions*. 2nd. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2021. ISBN 9781000409079. Disponível em: <<https://adv-r.hadley.nz/>>.
- MURRELL, P. *R graphics*. 3. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2019. 423 p. (The R Series). ISBN 978-1498789059.
- PARADIS, E. *R for Beginners*. 2005. Disponível em: <https://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf>.
- R CORE TEAM. *R data Import/Export*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- R CORE TEAM. *R instalation and administration*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- R CORE TEAM. *R internals*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- R CORE TEAM. *R language definition*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.

- R CORE TEAM. *Writing R extensions*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- SILGE, J.; ROBINSON, D. *Text mining with R*. O'Reilly Media, 2017. ISBN 978-1491981658. Disponível em: <<https://www.tidytextmining.com/>>.
- VENABLES, W. N.; SMITH, D. M.; R CORE TEAM. *An Introduction to R: Notes on R a programming environment for data analysis and graphics*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- WICKHAM, H. *R Packages*. 2nd. ed. Sebastopol: O'Reilly Media, 2015. ISBN 9781491910597. Disponível em: <<https://r-pkgs.org/index.html>>.
- WICKHAM, H. *ggplot2: Elegant graphics for data analysis*. 2. ed. New York: Springer, 2018. 276 p. (Use R!). ISBN 978-3319242750. Disponível em: <<https://ggplot2-book.org/>>.
- WICKHAM, H. *Advanced R*. 2nd. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2019. ISBN 978-0815384571. Disponível em: <<https://adv-r.hadley.nz/>>.
- WICKHAM, H. *Mastering shiny*. O'Reilly Media, 2021. ISBN 978-1492047384. Disponível em: <<https://mastering-shiny.org/>>.
- WICKHAM, H.; GROLEMUND, G. *R for Data Science*. Sebastopol: O'Reilly Media, 2017. ISBN 978-1491-91039-9. Disponível em: <<https://r4ds.had.co.nz/>>.
- WILKE, C. O. *Fundamentals of data visualization: A primer on making informative and compelling figures*. Sebastopol: O'Reilly Media, 2016. 390 p. ISBN 978-3319242750. Disponível em: <<https://ggplot2-book.org/>>.
- XIE, Y. *Dynamic documents with R and knitr*. 2. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2015. ISBN 978-1498716963. Disponível em: <<https://yihui.org/knitr/>>.
- XIE, Y. *bookdown: Authoring books and technical documents with r markdown*. Boca Raton, Florida: Chapman and Hall/CRC, 2017. ISBN 978-1138700109. Disponível em: <<https://bookdown.org/yihui/bookdown/>>.
- XIE, Y.; ALLAIRE, J. J.; GROLEMUND, G. *R Markdown: The definitive guide*. Chapman and Hall/CRC, 2018. ISBN 978-1449359119. Disponível em: <<https://bookdown.org/yihui/rmarkdown/>>.
- XIE, Y.; THOMAS, A.; HILL, A. P. *blogdown: Creating websites with r markdown*. Boca Raton, Florida: Chapman and Hall/CRC, 2018. 172 p. (The R Series). ISBN 978-0815363729. Disponível em: <<https://bookdown.org/yihui/blogdown/>>.

Índice Remissivo



Índice Remissivo



- distill, 4
- foreign, 84
- formatR, 4
- ggplot2, 4
- knitr, 4
- learnr, 4
- lobstr, 4, 9, 27, 46
- magrittr, 59, 85
- pryr, 4
- readr, 74
- rlang, 4
- rmarkdown, 4, 13
- shiny, 4, 13
- sloop, 4, 29, 34, 61, 64
- styler, 4
- tidyverse, 59
- XR, 4
- Princípio
 - função, 16, 82
 - interface, 7, 16, 72
 - objeto, 16, 29
- Programação defensiva, 90
- Prompt de comando, 17, 19
- R, 6
 - software* livre, 9
 - ambiente, 1
 - artigos, 1
 - banco de dados, 1
 - CRAN, 8
 - primeiro espelho no Brasil, 13
 - código aberto, 9
 - dashboards, 1
 - escopo, 9
 - estilo funcional, 82
 - gerenciamento de memória, 9
 - gráficos, 1
 - história, 8
 - IGU, 1
 - Instalação, 14
 - interface, 13
 - linguagem, 1
 - livros, 1
 - materiais didáticos, 4
 - paralelização, 1
 - programa, 1
 - programação defensiva, 1
 - R Core Team, 8, 72
 - semântica, 1
 - sintaxe, 1
 - três princípios, 6, 16
 - websites, 1
- RStudio, 6, 17, 74
- Instalação, 14
- J. J. Allaire, 13
- quadrantes, 17
 - primeiro, 17
 - quarto, 17
 - segundo, 17
 - terceiro, 17
- Script, 6
- Superatribuição, 5, 132
- Tipagem
 - Linguagem C, 31
 - linguagem S, 30