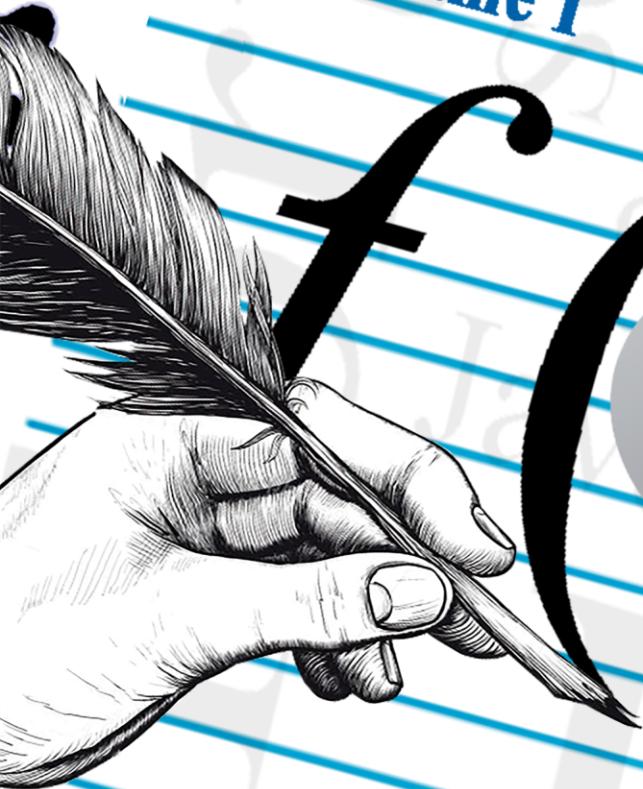


Estudando o ambiente R

R básico

Volume I



Democratizando
Conhecimento

*Ben Dêivide
Diego Arthur*

R BÁSICO

BEN DÊIVIDE DE OLIVEIRA BATISTA
DIEGO ARTHUR BISPO JUSTINO DE OLIVEIRA

R básico

Volume I

BEN DÊIVIDE DE OLIVEIRA BATISTA
DIEGO ARTHUR BISPO JUSTINO DE OLIVEIRA



Democratizando
Conhecimento

Ouro Branco, MG, 28 de agosto de 2022

© 2022 by Ben Dêivide de Oliveira Batista, Diego Arthur Bispo Justino de Oliveira

Direitos de publicação reservados aos autores.

ISBN (Digital): 978-65-00-51600-5

ISBN (Impresso): 978-65-00-51599-2

Projeto Gráfico: Ben Dêivide de O. Batista

Revisão técnica e textual: Colaboração independente

Editoração Digital: Ben Dêivide de O. Batista e Diego Arthur B. J. de Oliveira

Capa: Ben Dêivide de Oliveira Batista

Como citar esta obra (Impressa):

BATISTA, B. D. O.; OLIVEIRA, A. B. J.. **R básico**. Ouro Branco, MG: [s.n.]. 2022. (Estudando o Ambiente R, v.1). ISBN 978-65-00-51599-2.

Como citar esta obra (Digital):

BATISTA, B. D. O.; OLIVEIRA, A. B. J.. **R básico**. Ouro Branco, MG: [s.n.]. 2022. (Estudando o Ambiente R, v.1). ISBN 978-65-00-51600-5. Disponível em: <https://bendeivide.github.io/book-eambr01/>. Acesso em: 28 de fevereiro de 2023.

Autor correspondente e mantenedor da obra:

Ben Dêivide de Oliveira Batista

Contato: ben.deivide@gmail.com

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

Dados Internacionais de Catalogação (CIP)
Divisão de Biblioteca da UFSJ

Batista, Ben Dêivide de Oliveira.

B333r R básico. / Ben Dêivide de Oliveira Batista; Diego Arthur Bispo Justino de Oliveira. - Ouro Branco, MG: [s.n], 2022.
xiv, 321 p. : il. color. (Estudando o ambiente R, v. 1)

Formato digital.

<https://bendeivide.github.io/book-eambr01/>

Inclui bibliografia.

ISBN 978-65-00-51600-5

1. Estatística 2. Probabilidades. I. Oliveira, Diego Arthur Bispo Justino de II. Título III. Série

CDU 519.682

Licença

Este livro foi publicado de forma independente, e sua versão digital está sob a Licença Creative Commons - Atribuição - Não Comercial 4.0 Internacional. Usamos também a filosofia de trabalho com o Selo Democratizando Conhecimento (DC), que pode ser acessada em <https://bendeivide.github.io/dc/>. O leitor é livre para compartilhar, redistribuir, transformar ou adaptar esta obra, desde que não venha a utilizá-la em nenhuma atividade de propósito comercial. Por fim, a única exigência é a atribuição dos créditos aos autores da obra.



Usamos também o selo Selo Democratizando Conhecimento (DC).



<https://bendeivide.github.io/dc/>

A versão digital do livro se encontra em:

<https://bendeivide.github.io/book-eambr01/>

Agradecimentos

Gostaríamos de agradecer inicialmente a Universidade Federal de São João del-Rei. Sem a liberdade de trabalho não teríamos o desempenho necessário para a entrega deste material. Também, devemos fazer menção a algumas pessoas que contribuíram com uma leitura prévia para a melhoria do material, que são:

- Rodrigo Ronchi Rossi Costa (UFSJ)
- Fernando Augusto Teixeira (UFSJ)
- Sérgio de Oliveira (UFSJ)
- Jose Eloy Ottoni (UFSJ)
- Daniel Furtado Ferreira (UFLA)
- Janilson Pinheiro de Assis (UFERSA)
- Wesley de Oliveira Santos (UFERSA)
- Allanna Darlene Vilaça (UNIFACVEST)
- Tiago Olivoto (UFSC)
- Felipe Andrade Velozo (UNIFAL)
- Julio Cesar Gonçalves de Freitas (UFPA)
- Regina Albanese Pose (USCS)
- Dulcídia Carlos Guezimane Ernesto (UFLA)
- José Edimar Vieira Costa Júnior (UFLA)
- André da Silva Pereira (UPF)
- Miguel Angel Acosta Chinchilla (*Alliance Bioversity - CIAT*)
- Daniel Vicente Batista (Univesp)

O nosso muito obrigado pelas sugestões e críticas!

Sobre os autores

Ben Dêivide de Oliveira Batista



Natural de Pau dos Ferros/RN, uma distância de 450 Km da capital Natal. Possui graduação em Engenharia Agronômica pela Universidade Federal Rural do Semi-Árido (2010), mestrado, doutorado e pós-doutorado em Estatística e Experimentação Agropecuária pela Universidade Federal de Lavras (2012, 2016 e 2019). Atualmente é professor de Estatística pela Universidade Federal de São João Del-Rei (UFSJ). Áreas de pesquisa: Estatística computacional, Estatística experimental, Probabilidade e Estatística. Lema para a vida profissional: "*o conhecimento é uma liberdade necessária*". Para mais, acesse: <http://bendeivide.github.io/>. ORCID: <https://orcid.org/0000-0001-7019-8794>.

Diego Arthur Bispo Justino de Oliveira

Natural de Maceió/AL, mas passou grande parte da vida em Brasília/DF. Possui graduação em Bel. Interdisciplinar em Ciência e Tecnologia pela Universidade Federal de São João Del-Rei(2019), dentro da área de Engenharia Mecatrônica. Participa na área de pesquisa: Estatística Computacional e de Software. Atua como Desenvolvedor de Software, especializado em ReactJS e NodeJS pela Rocketseat(2022). Ama a busca pelo conhecimento e o ensino, e se diz grato por todas as oportunidades. Este livro para o autor é uma de suas maiores conquistas até agora, junto ao orientador e professor Ben Dêivide e a UFSJ. Lema para vida profissional e pessoal: "*nossa existência não é apenas em vida ou em matéria, mas também quando deixamos obras, marcas e lembranças na mente de cada um que convivemos. Pois aquela é passageira e, esta, esta é eterna*". Para mais, acesse: <https://digoarthur.github.io/>. ORCID: <https://orcid.org/0000-0001-9627-488X>.



Epígrafe

A melhor linguagem é a que você domina!
Ben Dêivide

Sumário

Licença	i
Agradecimentos	iii
Sobre os autores	v
Epígrafe	vii
Prefácio	xiii
1 Entendendo a coleção Estudando o ambiente R	1
1.1 R Básico (<i>Volume I</i>)	3
1.2 R Intermediário (<i>Volume II</i>)	3
1.3 R Avançado (<i>Volume III</i>)	4
1.4 Demais Volumes	4
1.5 Referências complementares da Coleção	5
1.6 Pacotes R utilizados para a Coleção	6
1.7 Exercícios	8
2 História e instalação do R	13
2.1 História do R	13
2.2 O que é o R?	14
2.3 Instalação do R e RStudio	23
2.4 Exercícios	25
3 Como o R trabalha?	27
3.1 Introdução	27
3.2 Como utilizar o R e o RStudio	29

3.3	Comandos no R	31
3.3.1	Console e Prompt de comando	31
3.3.2	Comandos elementares	32
3.3.3	Execução de comandos	34
3.3.4	Chamada e correção de comandos anteriores	36
3.4	Ambiente global, workspace (área de trabalho ou espaço de trabalho) e diretório de trabalho	36
3.5	Arquivos .RData e .Rhistory	38
3.6	Criando e salvando um script	39
3.7	Exercícios	41
4	Objetos e estruturas de dados	44
4.1	Introdução	44
4.2	Atributos	53
4.2.1	Atributos intrínsecos	58
4.3	Coerção	67
4.4	Estruturas de dados	69
4.4.1	Vetores	70
4.4.1.1	Vetores escalares ou constantes	72
4.4.1.2	Vetores longos	79
4.4.1.3	Manipulando vetores	90
4.4.1.4	Aritmética e outras operações	95
4.4.1.5	Operadores lógicos	100
4.4.1.6	Operadores booleanos	103
4.4.2	Matrizes unidimensionais	105
4.4.3	Matrizes multidimensionais	109
4.4.4	Listas	113
4.4.5	Quadro de dados	117
4.5	Exercícios	122
5	Importação e exportação de dados	129
5.1	Introdução	129
5.2	Preparação dos dados	130
5.3	Importando dados	131
5.4	Exportando dados	139
5.5	Exercícios	153
6	Funções no R	156
6.1	Introdução	156

6.2	O que é uma função no R?	156
6.3	Estrutura básica de uma função	158
6.4	Funções em pacotes	161
6.5	Chamadas de funções	162
6.6	Estruturas de controle ou controle de fluxos	163
6.6.1	<i>Loops</i>	175
6.6.2	Funções da família <i>apply</i>	184
6.7	Criando funções	196
6.7.1	Função anônima	199
6.7.2	Ordenação de argumentos	200
6.7.3	Objeto reticências (“...”)	202
6.8	Escopo léxico	207
6.9	Exercícios	212
7	Boas práticas de como escrever um código	217
7.1	Introdução	217
7.2	Nomes de arquivos	220
7.3	Comentar as linhas de comando	220
7.4	Nomes associados a objetos	221
7.5	Sintaxe	222
7.6	Exercícios	226
8	Pacotes	228
8.1	Introdução	228
8.2	Estrutura básica de um pacote	230
8.3	Instalação de um pacote	231
8.4	Objetivos de um pacote	233
8.5	Utilizar as funções de um pacote	233
8.6	Carregando e anexando um pacote	237
8.7	<i>NAMESPACE</i> de um pacote	240
8.8	Documentações de ajuda de um pacote	241
8.9	Operadores :: e :::	241
8.10	Exercícios	242
9	Ambientes e caminho de busca	247
9.1	Introdução	247
9.2	Ambiente global	247
9.3	Criando ambientes	250
9.4	Ambiente “pai” (ou ambiente superior)	251

9.5 Operador superatribuição (“<<-”)	253
9.6 Caminho de busca	261
9.6.1 Função attach() e o caminho de busca	264
9.7 Exercícios	267
10 Interfaces com outras linguagens	271
10.1 Introdução	271
10.2 Implementação em Python	273
10.3 Implementação em C/C++	274
10.4 Implementação em Tcl/Tk	275
10.5 Exercícios	278
11 Considerações e preparação para o Volume II	280
11.1 Introdução	280
11.2 Exercícios	281
12 Dica dos Exercícios	283
Referências	304
Índice Remissivo	307

Prefácio

A coleção *Estudando o ambiente R* é fruto de cursos ministrados sobre a linguagem **R**, bem como colaborações e estudos ao longo dos anos. Em 2005, quando ingressei no curso de Engenharia Agronômica, fiquei fascinado com a disciplina de Estatística no segundo semestre do ano corrente. Na sequência, obtive o primeiro contato com o ambiente **R** pouco mais de 9 anos de seu lançamento e redistribuição. Naquela época, haviam poucos materiais em língua portuguesa. Porém, foi o suficiente para eu entender que estava diante de uma grande ferramenta computacional e estatística, necessária para a compreensão. À época, inclusive, até sonhei em chegar neste dia, qua seja, o dia do lançamento de uma obra como esta.

Hoje, no ano de 2022, usuário há mais de 16 anos dessa linguagem, percebi um sentimento inquietante somente na posição de usuário do **R**. Eu posso reafirmar que uma estratégia muito poderosa de ensino/aprendizagem é a técnica *aprender por ensinar*, particularmente, porque além de democratizar o conhecimento perpetuando-o, aprendo mais e mais sobre determinado assunto. Dessa forma, lotado no Departamento de Estatística, Física e Matemática (DEFIM) desde 2018, campus Alto Paraopeba, pela Universidade Federal de São João del-Rei (UFSJ), comecei a planejar diversos cursos sobre a linguagem R.

Com o apoio do Centro Acadêmico de Engenharia de Telecomunicações (Catel/UFSJ), iniciei, no ano de 2021, o primeiro curso para o módulo Básico, e na sequência foi realizado o módulo intermediário. Todos no formato *online*, para que o conhecimento ultrapassasse as fronteiras da Universidade. Até o presente momento, o módulo avançado que seria o último curso, está em fase de desenvolvimento. A ideia deu tanto certo, que apesar do momento de pandemia e situação difícil para o mundo, foi realizada mais de 1.000 inscrições, tanto de alunos da UFSJ, como também de alunos da América do Sul, Moçam-

bique, Angola, dentre outros países e outros estados do Brasil. Tudo isso, realizado de forma gratuita e com emissão de certificados emitidos pelo Catel.

Os cursos ministrados sobre o ambiente **R** tentaram de forma simplificada, apresentar as ideias básicas sobre a sintaxe e semântica da linguagem, sendo criado três módulos, o nível básico, intermediário e avançado. No primeiro contato do aluno com o **R** nos cursos, não se aborda situações complexas, mas sim, o fundamental para uso como ferramenta básica. À medida que a aprendizagem nos módulos se avança, os problemas se tornam mais complexos, e a introdução de ferramentas tecnológicas é utilizada para auxiliar os alunos no entendimento sobre a linguagem. O resultado dessa experiência ficará registrada, em forma de conhecimento, na coleção apresentada a seguir.

A coleção *Estudando o ambiente R* apresenta três *Volumes* iniciais, intitulados por: o **R Básico (Volume I)**, o **R Intermediário (Volume II)** e o **R Avançado (Volume III)**. Fazendo a alusão aos três livros iniciais sobre a linguagem S desenvolvidos por John Chambers e colaboradores, é realizado uma explanação desde assuntos mais simples até noções mais complexas sobre o **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 os alunos para que se tornem protagonistas do processo de aprendizagem, e agora colegas de trabalho, tenho a parceria no *Volume I*, de Diego Arthur, um ex-orientando, e agora um profissional que tenta se superar a cada desafio e assunto estudado. Por isso, tenho a imensa satisfação de ter a sua contribuição e experiência neste material.

Por fim, espero que este primeiro *Volume* sirva de referência aos passos iniciais nessa ferramenta tão importante para a análise de dados.

Ben Dêivide de Oliveira Batista
Ouro Branco, MG, 28 de agosto de 2022

Capítulo 1

Entendendo a coleção Estudando o ambiente R

A coleção *Estudando o ambiente R* não tem como objetivo principal ensinar análise de dados, mas proporcionar ao leitor um conhecimento sobre a linguagem **R**, de modo a usufruir de todos os recursos que o ambiente proporciona. Pressupomos também, que na leitura do livro haja um conhecimento básico sobre linguagem de programação.

Linguagem R



O **R** é um ambiente de *software* livre e código aberto, com recursos gráficos, cujo objetivo se inicia pela análise e apresentação de dados até recursos para desenvolvimento de página *web*, por exemplo. Devido a contribuição da linguagem ser realizada por toda a comunidade científica e sociedade como um todo, os recursos desenvolvidos pelo **R** são dos mais diversos possíveis.

Ainda como complemento, não queremos neste material convencê-lo a utilizar o ambiente **R**, pois a melhor linguagem é a que dominamos. Contudo, pretendemos mostrar que os recursos utilizados pelo **R** não estão limitados apenas a análise de dados nos dias atuais. Um exemplo é o desenvolvimento escrito do material didático desta coleção, que pode ser acessado em: <https://bendeivide.github.io/cursor/>. Usufruímos das próprias ferramentas do **R** 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 **R** pode ser para nossa vida profissional.

Dessa forma, propomos um entendimento sobre a sintaxe e semânti-

tica¹ de como a linguagem **R** é desenvolvida. Seremos capazes, após a leitura dos dois primeiros *Volumes*, de acompanhar qualquer curso de Estatística com aplicações em **R**, dedicando-se apenas aos assuntos pertinentes à área da Estatística, uma vez que o embasamento sobre a linguagem será suprida por esta coleção. A nova revolução dos dados, se deve ao grande volume de informações obtidas nessa era tecnológica. Juntamente com a Estatística, o **R** representa uma poderosa ferramenta para entender os padrões que estão por trás dos dados, que por sinal, são a moeda valiosa do momento, ou melhor, sempre foram!

Aprenderemos também recursos diversos na área da computação, como programação defensiva, desenvolvimento de interfaces gráficas ao usuário (IGU), paralelização, recursos na área da estatística, 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**?



Em muitos momentos utilizaremos o artigo “o” para a linguagem **R**. Pois é, isso ocorre porque ela também é considerada um *software* ou ambiente. Daí, chamamos de programa **R**, ou preferivelmente, ambiente **R**.

A coleção apresenta três *Volumes* iniciais para o entendimento do ambiente **R**:

- **R** Básico (*Volume I*);
- **R** Intermediário (*Volume II*); e
- **R** Avançando (*Volume III*).

A seguir, explanaremos sobre cada um dos volumes.

¹Em linguagem de programação, a sintaxe representa a estrutura de como a linguagem é escrita. Ao passo que a semântica representa o comportamento associado a essas estruturas.

1.1 R Básico (Volume I)

Este primeiro *Volume*, apresenta um breve **histórico** sobre a linguagem, a sua instalação, bem como os recursos da *IDE*² **RStudio**, o conhecimento da **sintaxe** e **semântica** da linguagem **R**, as estruturas bases da linguagem, o que é um **objeto** e como construir uma **função**, o entendimento sobre **controle de fluxos**. O que é um **pacote**, **carregar** e **anexar** um pacote, e quem são os mantenedores da linguagem, também serão assuntos deste primeiro *Volume*. Noções básicas sobre o **caminho de busca**, **ambientes** e **namespaces** serão abordados. Algo muito interessante, que pode mudar o *script* de um programador em **R**, são as **boas práticas para a escrita de um código**, tema também abordado neste *Volume*.

A ideia do *Volume I* é proporcionar um entendimento básico, um primeiro contato com a linguagem, fazendo com que nós possamos dar os primeiros passos, executando as primeiras linhas de comando. Apresentaremos erros recorrentes, como o entendimento sobre um objeto, ou o anexo de um pacote ao caminho de busca. Temas como esses, dentre outros, foram a forma inicial que encontramos para aprofundar sobre a estrutura de um objeto **R**, bem como a sua manipulação. Adicionado a isso, faremos a inserção de como são os paradigmas da programação no ambiente. Esta última parte será estudada no *Volume II*, apresentada a seguir.

1.2 R Intermediário (Volume II)

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 algumas outras estruturas de dados como *tibble*, **cópias de objetos**, dentre outras situações. Uma característica do ambiente **R** é que a linguagem pode ser **orientada a objetos** e isso será estudado neste *Volume*. Introduziremos o **desenvolvimento de pacotes R**, e aprofundaremos sobre os **ambientes**. Por fim,

²Do inglês, *Integrated Development Environment*, que significa ambiente de desenvolvimento integrado.

mostraremos como desenvolver um projeto do **RStudio** e integrá-lo ao **GitHub**, e dessa forma, introduziremos sobre o sistema **Git**.

Este talvez seja o maior *Volume* dentre os três iniciais, pois apesar de não ser mais necessário entender a ideia de objeto, retratada no *Volume I*, a inserção dos paradigmas da programação para o *Volume II* trará uma maior riqueza de características para o **R**, mostrando a sua versatilidade. Também, daremos um maior detalhe sobre como manipular objetos, e as otimizações existentes da linguagem como por exemplo, a modificação no local, que se entendida, perceberemos que o *loop* no ambiente **R** não é lento como ouvimos falar. Ao final do *Volume II*, falaremos sobre como propagar um código com o sistema **Git** na plataforma **GitHub**, sincronizado com os projetos do **RStudio**.

1.3 R Avançado (*Volume III*)

No *Volume III*, exploraremos totalmente o 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 este *Volume* para a leitura dos demais.

Faremos também uma introdução sobre a linguagem C, entendendo algumas estruturas, como por exemplo, ponteiros e instruções, para percebermos que a arquitetura dos objetos em **R** são desenvolvidas baseadas nessas estruturas. Usaremos a linguagem C sem necessidade de pacote adicional. Desenvolveremos uma introdução sobre a linguagem FORTRAN e S, duas outras linguagens complementares para o entendimento dos bastidores do **R**.

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 assuntos.

1.5 Referências complementares da Coleção

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

- Manuais do **R**:
 - *An Introduction to R* (VENABLES; SMITH; R CORE TEAM, 2022);
 - *R Data Import/Export* (R CORE TEAM, 2022a);
 - *R Installation and Administration* (R CORE TEAM, 2022b);
 - *Writing R extensions* (R CORE TEAM, 2022e);
 - *R language definition* (R CORE TEAM, 2022d);
 - *R Internals* (R CORE TEAM, 2022c);
- *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 os três últimos livros unidos, são a base de entendimento do ambiente **R**.

1.6 Pacotes R utilizados para a Coleção

Uma lista de alguns pacotes são apresentados na Tabela 1.1, utilizados ao longo deste *Volume* para utilização em: exemplos abordados e consultas, como também para o próprio desenvolvimento do livro. Alguns dos exercícios e exemplos precisarão da instalação desses pacotes. Para isto, podemos consultar o Capítulo 8 e verificar como instalar um pacote no **R**.

Apresentamos também alguns exercícios ao final deste capítulo, dos quais se a resolução for de fácil entendimento, compreendendo a sua complexidade, podemos avançar para a leitura do *Volume II*.

Tabela 1.1: Pacotes a serem consultados para o acompanhamento dos exemplos e exercícios, bem como o desenvolvimento deste material.

Pacote	Finalidade
abind	Usado para combinar <i>array</i> multidimensionais
base	Principal pacote nativo ³

³Chamamos pacotes nativos, os pacotes disponíveis no momento em que o **R** é instalado, são os pacotes padrão e/ou recomendados.

bookdown	Criação dos livros digitais
codetools	Estudar a sintaxe do ambiente R
datasets	Para exemplos de conjuntos de dados (Pacote nativo)
distill	Criação da página <i>web</i>
elipse	Maior controle sobre o objeto retângulos (. . .)
foreign	Importação de arquivos de outros programas (Pacote nativo)
formatR	Auxilia no estilo de código
ggplot2	Renderização de gráficos
graphics	Gráficos (Pacote nativo)
knitr	Criação do material digital
learnr	Criação dos materiais dinâmicos
lobstr	Estudar a sintaxe do ambiente R
magrittr	Mecanismo para encadear comandos
midrangeMCP	Comparações múltiplas baseadas na distribuição da <i>midrange</i>
openxlsx	Ler e exportar arquivos binários, por exemplo, “.xlsx”
pryr	Útil para o entendimento mais profundo do ambiente R
Rcpp	Interface para as linguagens C/C++
RcppEigen	Álgebra linear
readxl	Ler arquivos binários, por exemplo, “.xlsx”
rlang	Estudar a sintaxe do ambiente R
rmarkdown	Criação do material digital
shiny	Criação dos materiais dinâmicos
sloop	Compreender o paradigma da programação orientada a objetos
stats	Ferramenta estatística (Pacote nativo)
styler	Auxilia no estilo de código
tidyverse	Instala e carrega os pacotes da família <i>tidyverse</i>
tinytex	Supporte ao LATEX

utils	Ferramentas úteis ao R (Pacote nativo)
writexl	Exporta arquivos binários, por exemplo, “.xlsx”
xlsx	Ler e exportar arquivos binários, por exemplo, “.xlsx”
XR	Estudar a sintaxe do ambiente R
xtable	Converte quadro de dados para tabelas em LAT_EX

1.7 Exercícios

Exercício 1.1: Qual a diferença entre anexar e carregar um pacote?

Dica na página 283

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

Dica na página 283

Exercício 1.3: Qual a importância da estrutura *NAMESPACE* em um pacote?

Dica na página 283

Exercício 1.4: Por que devemos usar com cautela as funções internas não exportáveis, de um pacote? Como acessá-las? Como usamos 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?

Dica na página 283

Exercício 1.5: Quantos objetos temos na linha de comando a seguir?

Script R:

```
1 x <- 10
```

Dica na página 283

Exercício 1.6: Quais os três princípios do R? Identifique-os em termos de linhas de comandos criadas em R.

Dica na página 283

Exercício 1.7: Como salvamos um script? Qual a importância de um script?

Dica na página 283

Exercício 1.8: O que são funções anônimas?

Dica na página 284

Exercício 1.9: Podemos afirmar que sempre o ambiente envolvente e ambiente de chamada são ambientes iguais?

Dica na página 284

Exercício 1.10: Considerando os ambientes envolvente, de execução e de chamada, qual dos três é um ambiente temporário? E quando este ambiente é criado?

Dica na página 284

Exercício 1.11: Baseado na linha de comando: x <- 1, por que não podemos afirmar que “x recebe o valor 1”?

Dica na página 284

Exercício 1.12: Como importamos um banco de dados externo do ambiente R? E como exportamos um resultado desejado?

Dica na página 284

Exercício 1.13: Por que comentar um código é muito importante para um programador?

Dica na página 284

Exercício 1.14: Precisamos ter o RStudio para utilizar o ambiente R?

Dica na página 284

Exercício 1.15: Podemos dizer que uma matriz ou array é um vetor? Se sim, o que os diferenciam?

Dica na página 284

Exercício 1.16: Considerando o script abaixo:

Script R:

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

Console R:

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

Script R:

```

13 # Criando um nome "f2" associado a uma função
14 f2 <- function() {
15   print(n) # Imprimindo "n"
16   n <- 15 # Criando "n" no ambiente envolvente de "f2"
17   print(n) # Imprimindo n
18 }
19 f2(); n # Imprimindo "f2" e "n"

```

Console R:

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

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

Dica na página 284

Exercício 1.17: Podemos afirmar que um *data frame* é uma lista? Justifique.

Dica na página 285

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

Dica na página 285

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

Dica na página 285

Exercício 1.20: Quais os pacotes **R** dão suporte como uma interface para as linguagens: Python, Julia, C/C++, FORTRAN, HTML

Java? Quais outros pacotes existem no R como uma interface para outras linguagens?

Dica na página 285

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?

Dica na página 285

Exercício 1.22: Para que serve os arquivos .RData e .Rhistory? Apresente exemplos práticos sobre o uso desses arquivos.

Dica na página 285

Exercício 1.23: Qual a relação existente entre o escopo de uma função e os objetos do tipo ambiente (environment)?

Dica na página 285

Exercício 1.24: Como provamos que a função print() está implícita quando executamos uma linha de comando no console?

Dica na página 285

Exercício 1.25: Qual a diferença entre estrutura de dados e tipo de objetos no ambiente R?

Dica na página 285

Capítulo 2

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 autores são os desenvolvedores da linguagem, George Ross Ihaka e Robert Clifford Gentleman.

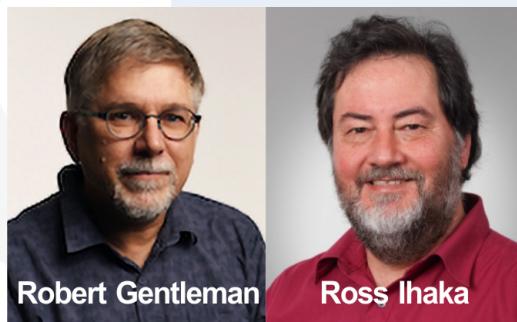


Figura 2.1: Criadores do R.¹

Durante a época em que os professores trabalhavam na Universidade de Auckland, na Nova Zelândia, desenvolvendo uma implementação alternativa da linguagem S², nasce em 1991, o projeto da linguagem **R**, sendo em 1993 a data de divulgação. Em 1995, o primeiro lançamento oficial, como *software* livre com a licença GNU.

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

²A linguagem S foi desenvolvida por John Chambers e colaboradores, comercialmente chamada de **S-PLUS**.

Devido a demanda de correções da linguagem, foi criado, em 1997, um grupo central voluntário responsável por essas atualizações: o *R Development Core Team* ou *R Core Team*³, sendo representado hoje por 20 membros (atualizado em 28 de fevereiro de 2023): 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 1997⁴. O CRAN é um conjunto de sites (espelhos) que armazenam materiais idênticos das 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*). A ideia do R está fundamentada na página da própria linguagem, em <https://www.r-project.org/about.html>, que define o R como um conjunto integrado de recursos de *software* para manipulação de dados, cálculo e exibição gráfica, incluindo:

- instalação eficaz de manipulação e armazenamento de dados;
- conjunto de operadores para cálculos matriciais;
- extensa coleção, coerente e integrada de ferramentas intermediárias para análise de dados;
- facilidades gráficas para análise de dados e exibição em tela ou em cópia impressa, e
- linguagem de programação bem desenvolvida, simples e eficaz que inclui condicionais, *loops*, funções recursivas definidas pelo usuário e recursos de entrada e saída.

Entendemos⁵ por:

³Fontes: https://www.cran.r-project.org/doc/html/interface98-paper/paper_2.html e <https://www.r-project.org/contributors.html>.

⁴Fonte: <https://stat.ethz.ch/pipermail/r-announce/1997/000001.html>.

⁵Fonte: <https://www.gnu.org/philosophy/free-sw.html>.

- **Software livre:** programa onde os usuários, em comunidade própria, possuem a liberdade de executar, copiar, distribuir, estudar, mudar e melhorá-lo. E ainda, um *software* é livre se os seus usuários possuem quatro liberdades:
 1. a liberdade de executar o programa como o usuário desejar, para qualquer propósito;
 2. a liberdade de estudar como o programa funciona, e adaptar as necessidades dos usuários;
 3. a liberdade de redistribuir cópias de modo que os usuários se ajudem;
 4. a liberdade de distribuir cópias de versões modificadas a outros.

Algo que deve estar claro é que um *software* livre não implica em *software* não comercial. Sem este fim, o *software* livre não atingiria seus objetivos. Percebemos por Richard Stallman⁶, que 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 Stallman⁷ o termo surgiu quando alguns usuários não concordavam com o termo *software* livre, devido as suas políticas de liberdade de uso de código. Assim, com apelo aos benefícios práticos do programa publicizado aos empresários, usaram o termo código aberto. Assim, o código aberto não associa valores como “certo” ou “errado”, mas o quanto poderoso e confiável é o programa. Todos os códigos abertos de *softwares* livres lançados se qualificariam como código aberto. Quase todos os *softwares* de código aberto são *softwares* livres, mas há exceções, como algumas licenças restritivas, do qual não podemos, por exemplo, fazer uma versão modificada e usá-la de forma privada. Assim, estes códigos não se qualificam como licenças livres. Nesse contexto, o autor cita muitas situações que diferenciam os dois termos. Vale a pena uma leitura para o entendimento filosófico sobre *software* livre e código aberto.

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

⁷Fonte: <https://www.gnu.org/philosophy/open-source-misses-the-point.html>.

- A ideia do **R** como ambiente, representa a flexibilidade e liberdade com o qual o programador se depara com a linguagem, de modo que temos um sistema totalmente planejado e coerente. O ambiente não é um *software* fechado, sem que implementações adicionais não possam ser realizadas, como parte dos *softwares* estatísticos. O **R** foi projetado como uma linguagem de programação, em que o usuário pode desenvolver as suas próprias funções e disponibilizá-las via pacote. A maior parte do sistema é escrito na própria linguagem **R**, tornando-se mais fácil para os usuários programarem seus algoritmos. Para a computação intensiva, o ambiente vincula códigos programados em C, C++ e FORTRAN, permitindo que possamos programar em C para manipular os objetos no ambiente. De acordo com Chambers (2016), este é o terceiro princípio do **R**: o **R** como uma interface com outras linguagens. Além do mais, os recursos gráficos disponíveis para análises de dados no ambiente são poderosos. Muito embora, citem o **R** como um sistema estatístico, é preferível assumi-lo como um sistema com ferramentas estatísticas implementadas. Percebemos que hoje o **R** apresenta recursos para desenvolvemos páginas *web* estatísticas ou dinâmicas, *dashboards*, apresentações, dentre outros recursos que vão além de análises estatísticas.

A linguagem **R** é um dialeto da linguagem **S** com a semântica de escopo léxico da linguagem Scheme. Dessa forma, o **R** se diferencia em dois aspectos principais⁸:

- **Gerenciamento de memória:** usando as próprias palavras de Ross Ihaka⁹, em **R** alocamos uma quantidade fixa de memória na inicialização e gerenciamos com coletor de lixo dinâmico. Isso significa pouco crescimento de *heap*¹⁰ e, como resultado, menos problemas de paginação comparado com a linguagem **S**.
- **Escopo:** as funções na linguagem **R** acessam os objetos criados no corpo da própria função, assim como os objetos contidos no

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

⁹Fonte: https://cran.r-project.org/doc/html/interface98-paper/paper_1.html.

¹⁰Espaço reservado para variáveis e dados criados durante a execução de um programa.

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



O escopo léxico é uma das grandes características existente no ambiente **R**, e que por exemplo, não existe na linguagem S. O escopo léxico proporciona maior flexibilidade as funções para a busca dos objetos, ou de um modo menos formal, a busca das variáveis inseridas no corpo da função criada.

Vejamos alguns exemplos para entendimento¹¹. Antes de executar as linhas de comando, devemos instalar o pacote **lobstr** como segue:

Script R:

```
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.
- Exemplo 2: As variáveis criadas ou alteradas dentro de uma função permanecem na função, Código R 2.2.
- 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.
- Exemplo 4: Por fim, embora a linguagem **R** tenha um escopo padrão, chamado ambiente global, os escopos de funções podem ser alterados, Código R 2.4.

¹¹Para iniciantes ainda não ambientados ao **R**, sugerimos a leitura até o Capítulo 4, e posteriormente, o retorno aos exemplos.

Código R 2.1**Console R:**

```
> # Criando um nome "n" associado a um objeto 10  
> n <- 10  
> # Nome "funcao" associado a um objeto funcao  
> funcao <- function() print(n)  
> funcao() # Imprimindo "funcao"  
[1] 10
```

Código R 2.2**Script R:**

```
1 # Criando um nome "n" associado a um objeto 10  
2 n <- 10  
3 lobjstr::obj_addr(n) # Endereco na memoria
```

Console R:

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

Script R:

```
4 # Criando um nome "funcao" associado a um objeto  
5 funcao <- function() {  
6   print(n) # Imprimindo n  
7   n <- 15 # Nome "n" associado a um objeto 15  
8   print(n) # Imprimindo n  
9 }  
10 funcao() # Imprimindo "funcao"
```

Console R:

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

Script R:

```
15 # Imprimindo "n"  
16 n
```

Console R:

```
[1] 10
```

Script R:

```
17 lobstr::obj_addr(n) # Endereco na memoria
```

Console R:

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

Código R 2.3**Script R:**

```
1 # Criando um nome "n" associado a um objeto 10  
2 n <- 10  
3 lobstr::obj_addr(n) # Identificador do objeto
```

Console R:

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

Script R:

```
4 # Criando um nome "funcao" associado a um objeto
5 funcao <- function() {
6   print(n) # Imprimindo n
7   n <- 15 # Nome "n" associado a um objeto 15
8   print(n) # Imprimindo n
9 }
10 funcao() # Imprimindo "funcao"
```

Console R:

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

Script R:

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

Console R:

```
[1] 15
```

Script R:

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

Console R:

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

Código R 2.4**Script R:**

```

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

```

Console R:

```
[1] 10
```

Script R:

```

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

```

Console R:

```
[1] 20
```

A linguagem **R** também é interpretada e baseada na linguagem **S**, tendo como base as linguagens de baixo nível **C**, **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 **RStudio**¹², criada por Joseph J. Allaire, Figura 2.2.



Figura 2.2: J. J. Allaire, o criador do **RStudio**.¹³

O **RStudio** tornou o **R** mais popular, pois além de apoiar no desenvolvimento de pacotes de grande utilização 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**. Alguns críticos afirmam para um iniciante em **R**, que não utilizem o **RStudio** para aprender a linguagem. Cremos, que o problema não é a *IDE*¹⁴ utilizada, mas o caminho aonde 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, autor de um dos primeiros materiais mais completos sobre a linguagem **R** produzidos no Brasil, iniciado em 2005, intitulado *Introdução ao Ambiente Estatístico R*. Vale a pena assistir a palestra: *R Refleflões: um pouco de história e experiências com o R*, proferida pelo Prof. Paulo Justiniano, no evento *R Day - En-*

¹²A empresa que mantém o **RStudio** apresentava o mesmo nome da *IDE*. Até que em 2022, o nome da empresa se modificou para **Posit**, uma forma de aumentar o seu alcance comercial na área de ciência de dados, uma vez que o suporte do **RStudio** não se limita mais a linguagem **R**. Atualmente, o **RStudio** dá suporte nativo aos códigos em Python e Julia no **RStudio**, bem como em outras linguagens.

¹³Fonte da foto: <https://rstudio.com/speakers/j.j.-allaire/>.

¹⁴Do inglês, *Integrated Development Environment*, que significa Ambiente de Desenvolvimento Integrado, como por exemplo, o **RStudio**, **Emacs**, dentre outros, para o **R**.

contro nacional de usuários do **R**, ocorrido em 2018 em Curitiba/UFPR, do qual o vídeo está disponível no Canal (Youtube) LEG UFPR.

2.3 Instalação do **R** e **RStudio**

Para realizarmos a instalação do ambiente **R**¹⁵, seguimos os seguintes passos:

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

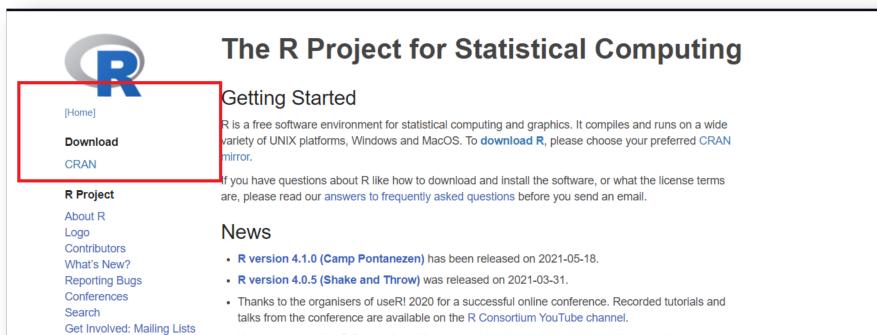


Figura 2.3: Primeiro passo para Instalação do **R**.

- 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**, cujo 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 uso pessoal, não deixaremos de repassar o entendimento sobre a linguagem **R** com o uso do **RStudio**.

O passo a passo para a instalação do **R** e **RStudio** foram baseados no sistema operacional *Windows*, e mais detalhes sobre as instalações

¹⁵Lembramos que o **RStudio** é apenas uma *IDE*, e sem o **R** não há sentido instalá-lo.

O-Cloud	https://cloud.r-project.org/	Automatic redirection to servers worldwide, currently sponsored by RStudio
Algeria	https://cran.usfhb.dz/	University of Science and Technology Houari Boumedienne
Argentina	http://mirror.fcaglp.unlp.edu.ar/CRAN/	Universidad Nacional de La Plata
Australia	https://cran.csiro.au/ https://mirror.aarnet.edu.au/pub/CRAN/ https://cran.ms.unimelb.edu.au/ https://cran.curtin.edu.au/	CSIRO AARNET School of Mathematics and Statistics, University of Melbourne Curtin University
Austria	https://cran.wu.ac.at/	Wirtschaftsuniversität Wien
Belgium	https://www.freestatistics.org/cran/ https://ftp.belnet.be/mirror/CRAN/	Patrick Wessa Belnet, the Belgian research and education network
Brazil	https://nbeqib.uesc.br/mirrors/cran/ https://cran.r.3s1.ufpr.br/ https://cran.fiocruz.br/ https://vps.fmvz.usp.br/CRAN/ https://brieger.esalq.usp.br/CRAN/	Computational Biology Center at Universidade Estadual de Santa Cruz Universidade Federal do Paraná Oswaldo Cruz Foundation, Rio de Janeiro University of São Paulo, São Paulo University of São Paulo, Piracicaba
Bulgaria	https://ftp.uni-sofia.bg/CRAN/	Sofia University

Figura 2.4: Segundo passo para Instalação do R .

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.				
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	
Fedor 19/Red Hat 7	rstudio-1.4.1717-x86_64.rpm	138.42 MB	648e2b00	
Fedor 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 .

em outros sistemas operacionais podem ser acessadas em nossos cursos disponíveis, sobre o **R**, em <https://bendeivide.github.io/>.

2.4 Exercícios

Exercício 2.1: Qual a importância de John Chambers para a linguagem **R**?

Dica na página 286

Exercício 2.2: Por que chamamos a linguagem **R** também de ambiente **R**?

Dica na página 286

Exercício 2.3: Pesquise sobre outras *IDEs* que possam dar suporte ao **R**.

Dica na página 286

Exercício 2.4: Quais as vantagens e cuidados que devemos ter com o ambiente **R**, sabendo que é uma linguagem de escopo léxico?

Dica na página 286

Exercício 2.5: O que a superatribuição tem a ver com o escopo léxico?

Dica na página 286

Exercício 2.6: Poderíamos afirmar que o uso do **RStudio** para iniciantes em **R** não seria uma boa escolha para a aprendizagem dessa linguagem?

Dica na página 286

Exercício 2.7: Qual o objetivo inicial do **R**, e quais avanços alcançados desse ambiente?

Dica na página 286

Exercício 2.8: Quem criou e onde foi desenvolvida a linguagem **R**?

Dica na página 287

Exercício 2.9: O que significa CRAN e qual a sua finalidade?

Dica na página 287

Exercício 2.10: Qual a importância das linguagens C, FORTRAN e S para o ambiente **R**?

Dica na página 287

Capítulo 3

Como o **R** trabalha?

3.1 Introdução

Para entendermos como o **R** trabalha, iniciamos com uma afirmação de John McKinley Chambers, Figura 3.1, dizendo que o **R** tem três princípios (CHAMBERS, 2016):

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.



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

Ao longo de todo os *Volumes* da coleção *Estudando o Ambiente R*, 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 é associado a um nome. A execução de uma função é baseada em argumentos (dados, fórmulas, expressões, etc), que são entradas, ou argumentos padrões, que são entradas pré-estabelecidas na criação da função. Os 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.

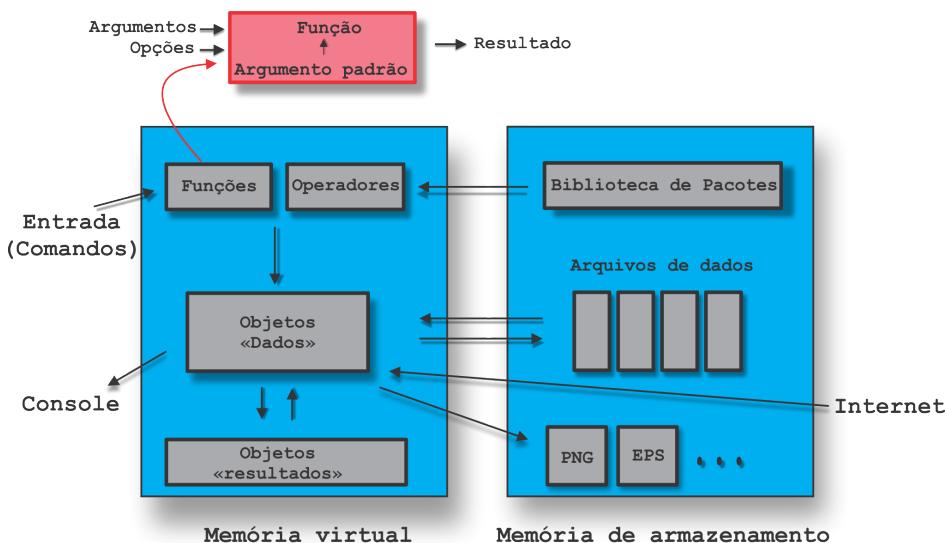


Figura 3.2: Esquema de como o R funciona.

Na Figura 3.2, observamos ainda que todos os objetos criados ficam armazenadas na memória virtual do computador. Os objetos são criados por comandos (teclado ou mouse) através de funções ou operadores (chamada de função), dos quais leem ou escrevem arquivo de dados da memória de armazenamento (*HDD* ou *SSD*²), ou da própria

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

²A sigla *HDD*, do inglês, *Hard Disk Drive*, significa o disco rígido. Já a sigla *SSD*,

web. Por fim, os resultados dos objetos podem ser apresentados no *console* (memória virtual), exportados em formato de imagem, página *web*, para a memória de armazenamento, 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, 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.

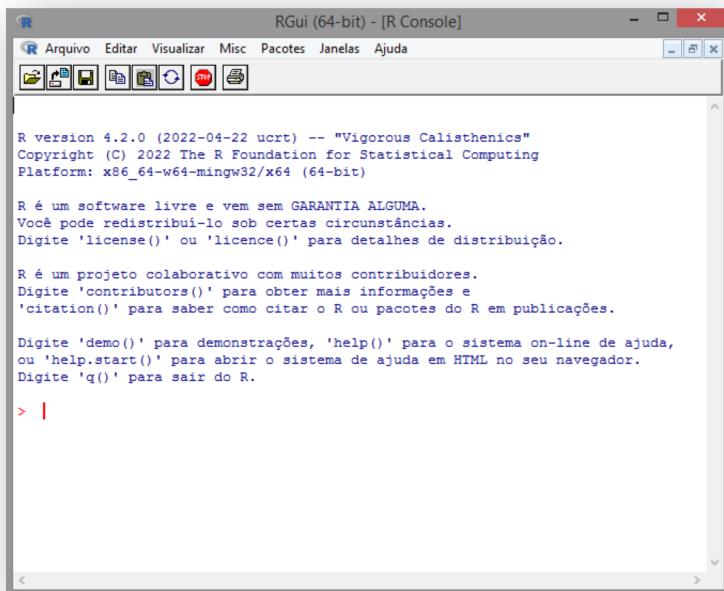


Figura 3.3: *Console* do R.

do inglês, *Solid State Disc*, significa o disco de estado sólido. Os dois são memórias de armazenamentos de um computador, porém o último por apresentar mais rapidez ao sistema, vem substituindo os *HDD* nos computadores mais modernos.

O **R** ao ser iniciado, as linhas de comando do usuário estão prontas para serem digitadas. Uma forma simples de armazenar os 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 deste *Volume*.

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

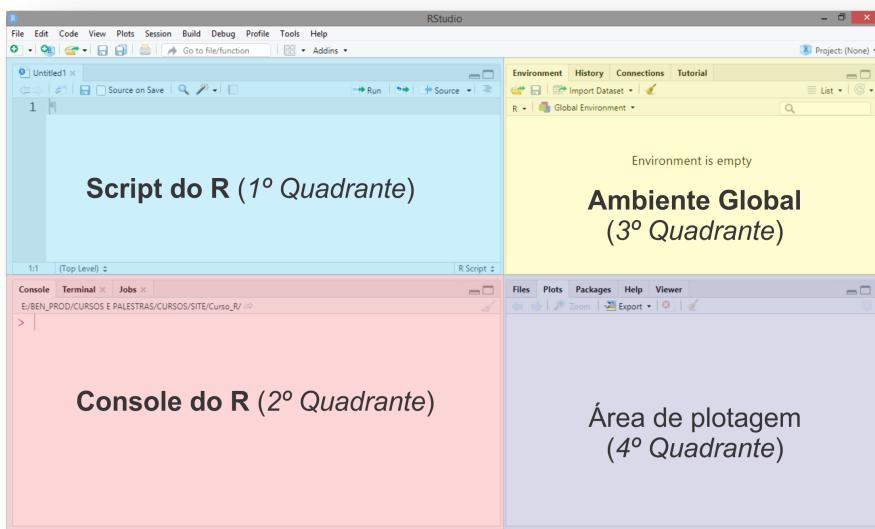


Figura 3.4: Interface do **RStudio**.

Muitas coisas na interface do **R** podem se tornar problemas para os usuários, uma vez que janelas gráficas, janelas de *scripts*, dentre outras, se sobreponem. 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 da demanda de 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 a entrada como a saída de informações (*input/output*). Por fim, o quarto quadrante é responsável por representação gráficas, instalação de pacotes, renderização de páginas *web*, etc..

3.3 Comandos no R

3.3.1 Console e Prompt de comando

As linhas de comando no R são executadas uma de cada vez no *console*. Assim que o *prompt* de comando estiver 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 R:

```
1 # Alterando o simbolo do prompt
2 options(prompt = "R>")
3 10 # Modificacao do simbolo do prompt "R>"
```

Console R:

```
R> 10
[1] 10
```

O conjunto de símbolos que podem ser utilizados no R depende do sistema operacional e do idioma 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 iniciados por uma letra ou um ponto não seguido de um número. Exemplos de nomes problemáticos: .123,

1n, dentre outros;

- As letras maiúsculas se distinguem das letras minúsculas;
- Não iniciamos por “_” ou dígito, é retornado um erro no *console* caso isso ocorra;
- Não usamos qualquer uma das palavras reservadas pela linguagem, isto é, TRUE, FALSE, if, for, dentre outras, que podemos consultar usando o comando ?Reserved.

Um nome que não segue essas regras é chamado de um nome **não sintático**. Um comando que usamos para converter nomes não sintáticos em nomes **sintáticos** é make.names().

Console R:

```
> # Nome nao sintatico
> .123 <- 50
> ## Error in 0.123 <- 50 : lado esquerdo da atribuicao
   invalida (do_set)
> # Qual a sugestao de nome sintatico para ".123"?
> make.names(.123)
[1] "X0.123"
```

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

3.3.2 Comandos elementares

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

Código R 3.2

Script R:

```
1 10 + 15
```

Console R:

```
[1] 25
```

No *console*, quando passamos pelo comando do Código R 3.2, o **R** avalia a expressão internamente e imprime o resultado na tela, após apertar o botão *ENTER* do teclado. Este fato é o 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. Este 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*. Intrinsicamente, também afirmamos que a função `print()` é executada na situação anterior, fazendo o papel de imprimir o resultado.

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**. 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

Console R:

```
> 10,5 + 15,5
Error: ',' inesperado em "10,"
```

Porém, devemos deixar claro que uma expressão³ é qualquer co-

³Não devemos confundir com o objeto tipo “expression”, na tipagem interna em C é denominado “EXPRSXP”. Nos manuais do **R**, preferem chamar os comando repassados ao console de **declaração** ao invés de expressão, para que não haja confusão nos termos.

mando repassado no *console*. O comando é avaliado e o resultado impresso, a menos que explicitamente seja do interesse torná-lo invisível⁴. 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 os processos ocorrem na memória virtual do computador, e uma vez o resultado impresso no *console*, o valor é perdido. Para que isso não ocorra, atribuímos um nome a essa expressão, que erroneamente usamos o termo: “criamos um objeto!”. A atribuição dessa expressão será dada pela junção de dois símbolos “<-”, falado mais à frente. Um comando em forma de atribuição também avalia a sua expressão, um nome se associa ao seu resultado. Após a execução, digitamos o “nome” atribuído ao objeto e o resultado é impresso, Código R 3.4.

Código R 3.4

Script R:

```
1 # Foi criado um objeto do tipo caractere e o nome
2 # "meu_nome" foi associado a ele. O "R" avalia
3 # essa expressao, mas nao imprime no console!
4 meu_nome <- "Ben"
5 # Para imprimir, digitamos o nome "meu_nome" no
6 # console e apertamos o botao ENTER do teclado!
7 meu_nome
```

Console R:

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

3.3.3 Execução de comandos

Quando inserimos um comando no *console*, executamos uma linha de comando por vez ou separados por ponto e vírgula (“;”), em uma mesma linha, Código R 3.5.

⁴Usamos a função `invisible(10 + 15)` para que a expressão seja avaliada mas não impressa.

Código R 3.5**Script R:**

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

Console R:

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

Se um comando for muito grande e não couber em uma linha, ou caso seja necessário completar um comando em mais de uma linha, após a primeira linha haverá o símbolo “+” iniciado na linha seguinte ao invés do símbolo de *prompt* de comando (“>”), até que o comando esteja sintaticamente completo, como pode ser observado no Código R 3.6.

Código R 3.6**Script R:**

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

Console R:

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

Por fim, todas as linhas de comando quando iniciam pelo símbolo jogo da velha (#) resultam em um comentário, e esta linha de comando não é avaliada, apenas impressa. 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*, podemos reexecutá-lo usando as teclas de “setas para cima” e “seta para baixo” do teclado⁵, recuperando os comandos anteriormente executados, e que os caracteres podem ser alterados usando as teclas “seta à esquerda” e “seta à direita” do teclado, ou removidas com o botão Delete ou *Backspace* do teclado, e acrescentadas digitando os caracteres necessários. Uma outra forma de completar determinados comandos já existentes, como por exemplo, funções dos pacotes nativos do **R**, é usar o botão *Tab* do teclado. Para isso, digitamos as iniciais e completamos o nome apertando a tecla *Tab*. Posteriormente, apertamos a tecla *ENTER* para executá-la. Para entender mais detalhes, podemos acessar o *link*: <https://youtu.be/0MRPmVsPvk4>.

Os recursos no **RStudio** são mais dinâmicos e vão além. Por exemplo, quando usamos um objeto do tipo função (“function”, “closure” ou “special”), eles apresentam o que chamamos de argumento(s). Sintaticamente, os argumentos são inseridos dentro do parêntese da função, do qual são elementos necessários para que a função seja executada corretamente. Assim, ao inserir o nome de uma determinada função 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 precisamos lembrar do nome dos argumentos de uma função, mas apenas entender o seu objetivo, Figura 3.5. Para detalhes adicionais no *link*: https://youtu.be/KL3WAB_GFNI.

3.4 Ambiente global, *workspace* (área de trabalho ou espaço de trabalho) e diretório de trabalho

Quando usamos um comando de atribuição no *console*, o **R** armazena o nome associado ao objeto criado no Ambiente Global (.GlobalEnv). O ambiente é um tipo de objeto que armazena objetos nominados, isto é, que associa nomes a objetos que podem ser recuperados. Já a área de trabalho, por definição segundo Venables, Smith e R CORE TEAM

⁵Esse comando está relacionado ao histórico de linhas de comandos, que pode ser verificado com *history()*. O arquivo que armazena as linhas de comando é *.Rhistory*, discutido na Seção 3.4 deste Capítulo.

● 3.4. Ambiente global, workspace (área de trabalho ou espaço de trabalho) e diretório de trabalho

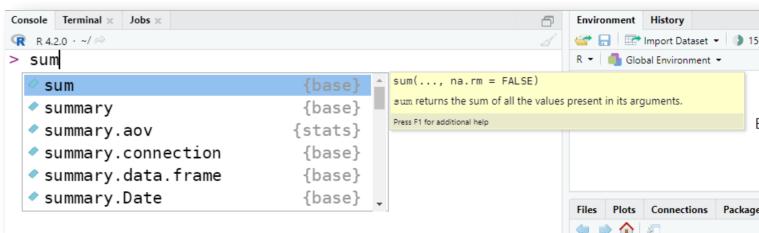


Figura 3.5: Chamada de comandos pelo **RStudio**.

(2022), representa uma coleção de objetos armazenados. Isso implica que o ambiente global representa a área de trabalho e o ambiente que quase sempre o usuário trabalha no **R**.

Ambiente Global e área de trabalho



O ambiente global é a área de trabalho interativa do **R**, mas nem toda área de trabalho é o ambiente global. Um exemplo é o ambiente de pacote, que representa uma área de trabalho e não é o ambiente global

Para listarmos os nomes contidos no ambiente global, usamos `ls()` ou `objects()`, Código R 3.7. Para remover individualmente um dos nomes, usamos `rm("nome")`, e para remover tudo `rm(list = ls())`. Porém, ao finalizarmos uma determinada atividade no **R**, salvamos em um arquivo “`.RData`”, bem como as linhas de comando em “`.Rhistory`”, mencionados na Seção 3.5.

Código R 3.7

Script R:

```
1 # Nomes associados a objetos no ambiente global
2 x <- 10 - 6; y <- 10 + 4; w <- "Maria Isabel"
3 # Verificando os nomes contidos no ambiente global
4 ls()
```

Console R:

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

Por fim, também salvamos as nossas linhas de comando por meio de um *script*, Seção 3.6. Posteriormente, consideramos o diretório onde o nosso *script* está salvo como o diretório de trabalho. O diretório de trabalho é o repositório que acessamos mais facilmente arquivos como banco de dados, imagens, etc., sem necessitar indicar todo o caminho onde o arquivo se encontra, nas linhas de comando de nosso código. O acesso é obtido pela chamada de função `getwd()`, e alterado por `setwd()`, Código R 3.8.

Código R 3.8**Script R:**

```
1 # Diretorio de trabalho
2 getwd()
```

Console R:

```
[1] "C:/eambr01"
```

Script R:

```
3 # Alterando o diretorio de trabalho (barras
4 # invertidas C:/eambr01, ao inves de C:\eambr01)
5 setwd("digite_o_caminho_do_repositorio")
```

3.5 Arquivos .RData e .Rhistory

Ao final de nosso trabalho no **R**, todo o processo ao inserir linhas de comando do *console*, dois arquivos são criados, sob a instrução do usuário

em querer aceitar ou não, Figura 3.6, 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*.

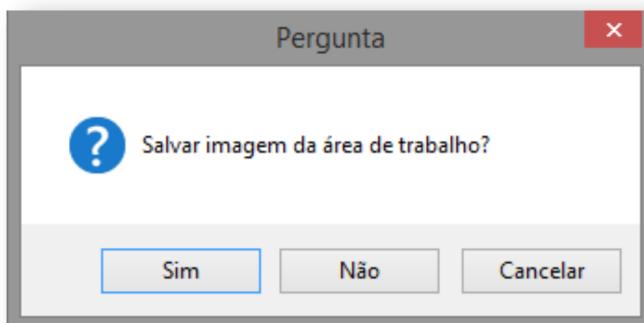


Figura 3.6: Janela de finalização do **R**.

Ao iniciar o **R** no mesmo diretório onde os arquivos foram salvos, é necessário carregar toda a área de trabalho anteriormente, bem como o histórico das linhas de comando. Mais detalhes, sugerimos a leitura no Capítulo 5, Seção 5.4.

3.6 Criando e salvando um *script*

A melhor forma de armazenarmos as linhas de código inseridas no *console* é criar um *script*, um arquivo de texto com a extensão “<>.R”. Uma vez criado, salvamos e guardamos o arquivo para utilizar futuramente.

Criamos um *script* no menu em Arquivo > Novo script. Posteriormente, inserimos as linhas de comando, e as executamos no *console* pela tecla de atalho F5. As janelas do *script* e *console* possivelmente ficarão sobrepostas na *RGui* do **R**. Para uma melhor utilização, essas

janelas podem ficar lado a lado, configurando-as no menu em Janelas > Dividir na horizontal (ou Dividir lado a lado).

No **RStudio**, criamos um *script* no menu em File > New File > R Script, ou diretamente no ícone abaixo da opção File no menu, cujo símbolo é “+” em verde, que é o ícone de New File. Posteriormente, clicamos em R Script. Um arquivo será aberto no primeiro quadrante do **RStudio**. As linhas de comando podem ser executadas pela tecla de atalho CRTL + ENTER, ou por meio do botão Run.

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 computador. Algumas ressalvas:

- Devemos escolher sempre um nome sem caracteres especiais como acentos, etc.;
- Devemos escolher sempre um nome curto ou abreviado, que identifique a finalidade das linhas de comando escritas;
- Devemos evitar espaços se o nome do arquivo for composto. Uma solução é usar entre as palavras o símbolo *underline* “_”;
- Quando escrevemos um código, devemos evitar também escrever caracteres especiais, exceto em casos de necessidade, como imprimir um texto na tela, títulos na criação de gráficos, dentre outros. Nos referimos especificamente aos comentários do código.

Um ponto bem interessante é o diretório. Quando criamos um *script* pela primeira vez e trabalhamos nele, muitos erros podem ser encontrados de início. Um problema clássico é a importação de dados. Por exemplo, temos um conjunto de dados e desejamos fazer a importação para o **R**. Mesmo com todos os comandos corretos, o *console* retorna um erro, informando que não existe o arquivo que contém os dados para serem importados. Isso é devido ao diretório de trabalho atual. Para verificarmos qual diretório estamos trabalhando, usamos a função `getwd()`, como já falado anteriormente.

Para alterarmos o diretório de trabalho, usamos a seguinte função `setwd("Inserir o local desejado!")`. Vamos supor que o *script* foi salvo 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.

Este procedimento no **RStudio**, é realizado da seguinte forma: menu > Session > Set Working Directory ... > To Source File Location. Isso levará ao diretório corrente do *Script*. Se desejarmos escolher outro diretório, seguimos em Session > Set Working Directory > Choose Directory Porém, uma vez criado um *script*, e utilizado posteriormente, o **RStudio**, por padrão, definirá o diretório do *script* como sendo o diretório de trabalho.

Devemos nos atentar também quando trabalhamos utilizando *scripts* ou arquivos de banco de dados em locais diferentes do diretório corrente. Um 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, criamos um *script* auxiliar que armazena todas as funções desenvolvidas para uma determinada análise, e no *script* corrente, essas funções são chamadas sem a necessidade de abrir 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**. Façamos isso a partir do primeiro dia em que foi desenvolvido a primeira rotina. Isso criará um的习惯, uma vez que o arquivo não está sendo criado apenas para um momento, mas para futuras consultas. E quando voltarmos ao referido *script* dessa rotina, principalmente depois de algum tempo, e sem comentários, possivelmente passaremos alguns instantes para tentar entender o que foi escrito. Para mais detalhes, sugerimos em **Boas práticas de escrita de um código**, Capítulo 6, um conjunto de técnicas que dará suporte a essas atividades.

3.7 Exercícios

Exercício 3.1: Como recuperamos os objetos no ambiente **R**? Qual o objeto mais importante dessa recuperação?

Dica na página 288

Exercício 3.2: Como criamos duas execuções de comando em uma única linha de comando?

Dica na página 288

Exercício 3.3: Quais as diferenças em usar a interface *RGui* do R (SO Windows) e a *IDE RStudio*?

Dica na página 288

Exercício 3.4: Quais as diferenças entre nomes sintático, não sintáticos e palavras reservadas no ambiente R?

Dica na página 288

Exercício 3.5: O que representa o ambiente global?

Dica na página 288

Exercício 3.6: Quais as diferenças entre .RData e .Rhistory?

Dica na página 288

Exercício 3.7: Como criar e salvar um *script* no R?

Dica na página 288

Exercício 3.8: Como podemos verificar e mudar o diretório de trabalho? Qual a sua importância?

Dica na página 288

Exercício 3.9: O que entendemos por “boas práticas de escrita de um código”? Por que devemos comentar um código?

Dica na página 289

Exercício 3.10: Ao longo deste capítulo, mencionamos apenas algumas palavras reservadas do ambiente R. Pesquise sobre as demais.



Dica na página 289

Exercício 3.11: Qual a diferença entre diretório de trabalho e diretório corrente?

Dica na página 289

Capítulo 4

Objetos e estruturas de dados

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**, uma espécie de metadado vinculado ao objeto. Vamos entender o termo estrutura como a disposição de como armazenamos as informações na memória virtual do computador e a forma como apresentamos essas informações a nível de usuário. No primeiro caso, dizemos que é a estrutura interna do objeto, isto é, o tipo (ou modo) do objeto. O segundo caso representa a estrutura externa, isto é, a forma de como os usuários observarão e manipularão as informações do objeto. Por exemplo, a estrutura externa pode ser o que chamamos de estrutura de dados, sendo que no **R** a estrutura de dados mais simples é o **vetor atômico**, pois os elementos contidos nele, apresentam o mesmo **modo**¹, cujo modo é um tipo de atributo. Existem outros tipos de estruturas externas não apenas para dados, mas também relacionadas à funções, expressões, etc..

O modo ou o tipo é o atributo que representa a estrutura interna do objeto. Falaremos nisso, mais à frente. De forma didática, adaptaremos a representação dos objetos no formato de diagrama. Toda vez que mencionarmos o termo **modo** ou **tipo**, estaremos nos referindo a mesma característica do objeto. Vejamos a seguinte linha de comando:

¹Ao longo de todo o texto, chamaremos de **modo** como o **tipo** de objeto, que nada mais é do que a **estrutura interna** do objeto.

Script R:

```
1 x <- 10
```

Todos que têm uma certa noção sobre a linguagem **R** afirmariam: “criamos um objeto **x** que recebe o valor 10”. Em grande parte das linguagens de programação, as variáveis são um mecanismo para que possamos acessar os dados armazenados na memória virtual do computador. Em **R**, não temos acesso direto a referência de memória dos dados. Ao invés, temos estruturas específicas, chamadas de objetos, e por meio delas acessamos os dados. Um usuário habituado com outras linguagens de programação chamaria **x** como uma variável, quando na realidade é um objeto do tipo “symbol” que se associa ao objeto 10, tipo “double”. O objeto do tipo “symbol” geralmente chamamos de **nome** e pode ser manipulado como qualquer outro objeto **R**. Assim, o correto é afirmar que o nome está se ligando ao objeto 10. E de fato, o objeto não tem um nome, mas o nome tem um objeto. A função que associa um objeto a um nome é o de “`<-`”, 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** do teclado. Representaremos em termos de diagrama, um nome se ligando a um objeto, na Figura 4.1. Similar ao que foi apresentado no *script* anterior, poderíamos ter associado um nome ao objeto por meio da função `assign()`. Porém, a sintaxe usando “`<-`”, é mais simples.

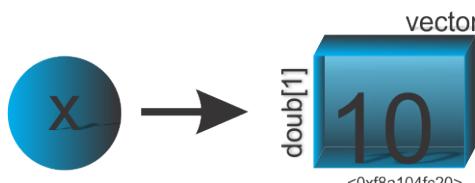


Figura 4.1: Dizemos que o nome **x** se liga ao objeto do tipo “double”(como estrutura de dado é um vetor).²

Queremos chamar atenção para a Figura 4.1, quando associamos o nome **x** ao objeto 10. A flecha no diagrama aponta no sentido contrário

²Quando nos referirmos a um tipo de objeto, no sentido de estrutura de dados, enfatizaremos no texto para que fique claro a distinção.

ao símbolo de atribuição para reforçar que o nome se liga ao objeto e não o contrário. O endereço desse objeto na memória virtual, destinada ao ambiente **R**, pode ser obtido por³:

Console R:

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

A Figura 4.1 mostra que o nome criado **x** se associou com um objeto de **estrutura**⁴ vetor (vector) e **tipo** “double”⁵, em que o endereço de memória virtual na memória virtual do computador foi <0xf8a104fc20>. Cada vez que abrirmos o ambiente **R** e executarmos novamente o comando, ou o repetirmos, o endereço do objeto será alterado.

Em outra representação, Código R 4.1, 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 em inglês para melhor compreensão e fixação, uma vez que os termos são baseados nesse idioma.

Código R 4.1

Script R:

```
1 y <- x
2 lobstr::obj_addr(y)
```

Console R:

```
[1] "0xf8a104fc20"
```

Observamos que não houve a criação de um outro objeto, mas apenas a ligação de mais um nome ao objeto existente, pois o endereço

³Usamos a função `obj_addr()` do pacote **lobstr**. Para instalá-lo, usamos a chamada `install.packages("lobstr")`.

⁴Quando mencionarmos o termo *estrutura de dado*, estamos nos referindo a estrutura externa do objeto, isto é, a forma como vemos os seus elementos.

⁵Na tipagem S, usando a função `mode()`, o objeto é do tipo “numeric”.

na memória virtual para o objeto não se alterou, Código R 4.1. Logo, temos dois nomes que se ligam ao mesmo objeto.

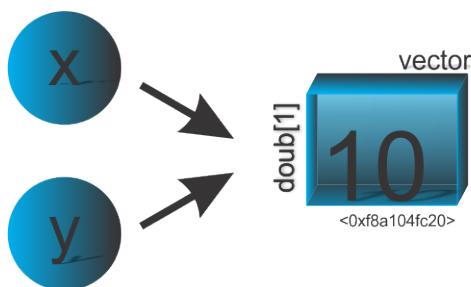


Figura 4.2: Dizemos que o nome x e y se ligam ao vetor.

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).

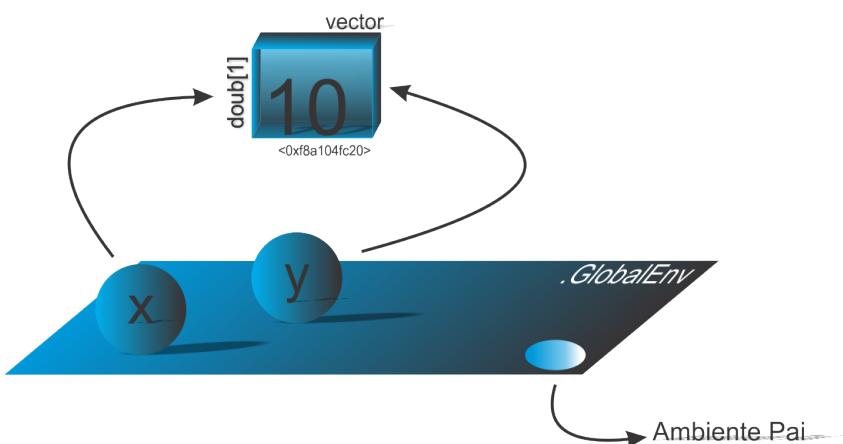


Figura 4.3: Dizemos que o nome x e y se ligam ao vetor e essa ligação fica armazenada no ambiente global.

De todo modo, deixaremos para o *Volume II* uma abordagem mais aprofundada sobre o assunto. O símbolo de atribuição poderá ser representado na direção da esquerda para à direita ou vice-versa, isto é:

Script R:

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

Essas duas linhas de comando anteriores podem passar 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 virtual do computador e associado ao novo objeto⁶, como observado no Código R 4.2.

Código R 4.2**Console R:**

```
> x <- 10
> lobstr::obj_addr(x)
[1] "0xf8a104fc20"
> x <- 30
> lobstr::obj_addr(x)
[1] "0x42db6dbb50"
```

Uma outra forma menos convencional é usar a função `assign()`, Código R 4.3, como já falado anteriormente.

Código R 4.3**Console R:**

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

Mais ainda, a função `assign()` permite que informemos onde a ligação ocorrerá, isto é, em qual ambiente. Porém, isso é assunto para o Capítulo 9.

⁶Na realidade o coletor de lixo se encarrega de eliminar objetos em desuso.

Usamos <- ou = para associação de nomes a objetos?



Muitos usuários utilizam o símbolo da igualdade “=” para associarmos nomes aos objetos, algo que o ambiente **R** compreenderá. Contudo, discutiremos mais adiante, no Capítulo 6, que o uso da igualdade deverá em **R** ser usado apenas para a utilização em argumentos de uma função e não para associação de nomes a objetos, uma vez que há diferenças entre os operadores, como por exemplo, a precedência superior de “<-” sobre “=”. Para mais detalhes, usamos a execução `?assignOps` no *console*.

Quando desejamos executar mais de uma linha de comando por vez, as separamos pelo símbolo “;”, Código R 4.4. Neste caso, executamos quatro comandos em uma linha. Associamos dois nomes a dois objetos e imprimimos os seus valores.

Código R 4.4

Console R:

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

Por questão de comodidade, iremos a partir de agora nos referir a um objeto pelo nome associado a ele, para não precisarmos enfatizar “um nome associado a um objeto”. Mas que deixamos clara a discussão realizada anteriormente sobre essa forma de se expressar.

Quando não associamos nomes aos objetos, a rigor não o recuperamos. Porém, para recuperar o último objeto criado, é possível chamando o objeto `.Last.value`. Outra coisa interessante é que quando desejamos exportar a saída do *console* para um determinado arquivo, basta usar a função `sink("output.txt")`. Por exemplo, neste último comando exportamos os resultados impressos no *console* para o arquivo `output.txt`. Se repetirmos o comando `sink()`, a gravação é finalizada para o arquivo, e os resultados voltam a ser impressos no *console*.

Por fim, deixamos claro duas coisas, o tipo (modo) e a estrutura de dados de um objeto. Existem 24 tipos de objetos, desenvolvidos em linguagem de baixo nível, mais especificamente em C, no ambiente **R**. Como já mencionado, identificamos alguns objetos visíveis a nível de usuário, por meio da função `typeof()`. Como o **R** passou ao longo do tempo por diversas mudanças em sua sintaxe e semântica, temos uma outra função chamada `mode()`, cujo o tipo dos objetos se baseia na sintaxe da linguagem S. Isso porque algumas funções foram desenvolvidas nessa linguagem. Porém, a saída das duas funções diferem em alguns tipos, como pode ser observado na Tabela 4.1.

Tabela 4.1: Tipagem de alguns objetos baseados nas linguagens C e S.

<code>typeof()</code>	<code>mode()</code>
“integer” e “double”	“numeric”
“special”, “builtin” e “closure”	“function”
“symbol”	“name”
“language”	“(” ou “call”

Neste momento, não iremos detalhar todos os tipos de objetos, mas que estes só podem ser criados apenas pela Equipe principal do **R** (*R Core Team*). Então, por exemplo, um tipo de objeto “integer” representa um vetor atômico cujos elementos são números inteiros. Tecnicamente, o vetor em si não é um tipo de objeto, mas uma estrutura de dados. De outro modo, não existe um objeto do tipo base que seja um vetor que armazena qualquer natureza de dado, mas sim, temos tipos de objetos para vetores de caracteres, vetores de números inteiros, vetores lógicos, etc., como pode ser observado no Código R 4.5.

Código R 4.5**Script R:**

```
1 # Criando um vetor
2 x <- vector()
3 # Verificando o tipo base
4 typeof(x)
```

Console R:

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

Script R:

```
3 # Inspecionando o objeto
4 .Internal(inspect(x))
```

Console R:

```
0x000000ba1be78230 10 LGLSXP g0c0 [MARK,REF(4)] (len=0,
tl=0)
```

Ao criar um objeto com a função `vector()`, obtemos um objeto do tipo “logical”, que na tipagem da linguagem C internamente é `LGLSXP`, representa um vetor atômico lógico, e não um objeto do tipo vetor, sem natureza especificada.

Por outro lado, existem muitas estruturas de dados que comumente são chamados erroneamente de tipo de objetos, como por exemplo, uma matriz, que pode ser criada com a função `matrix()`. No Código R 4.6, observamos que o tipo do objeto criado dessa função é “`double`”, que internamente é “`REALSXP`”, isto é, um vetor atômico real com um atributo “`dim`”.

Código R 4.6

Script R:

```
1 # Estrutura de dado - Vetor  
2 x <- c(1, 2, 3, 4); x
```

Console R:

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

Script R:

```
3 typeof(x)
```

Console R:

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

Script R:

```
4 # Estrutura de dado - Matriz  
5 y <- x  
6 dim(y) <- c(2, 2)  
7 y
```

Console R:

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

Script R:

```
8 typeof(y)
```

Console R:

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

Percebemos que apesar do mesmo tipo de objeto, a estrutura modifica a forma de como o observamos, bem como a busca pelos dados. O sistema de indexação dessas estruturas acabam se modificando também. Portanto, o tipo de objeto é uma característica interna, não se muda. Porém, a estrutura de dados é uma característica de semântica, o objeto se modifica de acordo com os seus atributos.

Vamos observar diversos outros atributos e diversas outras estruturas de dados ao longo do livro. E para finalizar, uma última função que queríamos desmistificar é a função `class()`. Alguns atributos, por terem características especiais devido a sua importância, acabam tendo funções próprias, e uma delas é o atributo “`class`”. Este atributo é usado no paradigma da programação orientada a objetos; apesar de tudo em **R** ser um objeto, nem tudo é orientado a objetos. Então, em muitos materiais mencionam que a função `class()` é usada para determinar o tipo de objeto, e isso não é verdade. Mas sim, dentro do paradigma da orientação a objetos, a função `class()` determina a classe em que o objeto pertence. No **R**, o paradigma citado foge aos padrões de orientação a objetos observado em outras linguagens. E uma última curiosidade: apesar do primeiro princípio do **R** afirmar que tudo no ambiente é um objeto, há uma exceção: são os nomes dos genéricos de grupo⁷: `Math`, `Ops`, `Complex` e `Summary`, disponíveis no pacote **base**. Porém, existem funções de mesmo nome no pacote nativo **methods**.

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()`. É devido aos atributos a criação de outras estruturas de dados no **R**, de modo a criarmos novas estruturas a partir de vetores, listas, etc.. Foi assim que surgiram as

⁷Procure os genéricos de grupo executando `?groupGeneric` no *console*.

estruturas como fatores, tabelas de contingências, matrizes, quadro de dados, *tibbles*, dentre outras.

Diremos também que todos os objetos **R** têm **classe(s)** ou **classe(s) implícita(s)**, e por meio dessas classes, determinadas funções podem ter comportamentos diferentes a objetos com classes diferentes. Agora, devemos deixar clara essa informação: apesar do **R** seguir o **princípio do objeto**, nem tudo é orientado a objetos.

A forma de verificar a classe de um objeto é pela função `class()`. Contudo, os objetos internos do **R** (padrão e recomendado⁸), quando solicitada sua classe pela função `class()`, acabam retornando, algumas vezes, resultados equivocados. Uma alternativa é utilizar a função `sloop::s3_class()` do pacote **sloop**.

Devemos nos atentar a uma questão, já falada anteriormente, que: **existe um atributo também chamado “class” (classe)**, e nem todos os objetos necessariamente tem este atributo, apenas aqueles orientados a objetos. Por exemplo, é devido a classe `factor` no objeto criado pela função `factor()` que apesar do seu resultado ser numérico, ele não se comporta como numérico. Percebemos que o atributo “class” muda o comportamento de como as funções veem o objeto. Entretanto, mesmo os objetos que não apresentam esse atributo, quando solicitados pela chamada `class()` do referido objeto, haverá o retorno do que chamamos de **classe implícita**, que nada mais é do que a tipagem do objeto baseada no atributo `modo` (obtido pela função `mode()` ou `typeof()`). A classe implícita não é definida pelo atributo `class`, mas pela tipagem do objeto.

Portanto, determinadas funções apresentam comportamento diferentes para objetos de classes implícitas devido as instruções do tipo `switch()` implementadas em C, e apenas o *R Core Team* tem o privilégio de criar objetos de classes implícitas no **R**.

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()`.

⁸Termos que representam os 30 pacotes, aproximadamente, de pacotes fornecidos com a instalação do **R**. São os pacotes padrão e recomendados para o trabalho mínimo do ambiente **R**.

Nem tudo em R é orientado a objetos!



Devemos refletir: Tudo em **R** é um objeto, mas nem tudo é orientado a objetos! O paradigma de orientação a objetos, em **R** ocorre quando um objeto recebe o atributo `class`. Outros objetos criados internamente no ambiente, que não apresentam este atributo, tem o que chamamos de classes implícitas, e mesmo assim, quando o invocamos por meio de `class()`, é retornado uma classe. Para estes objetos, algumas funções se comportam de maneira diferente, devido a instruções do tipo `switch()` implementadas em C.

O principal paradigma de orientações a objetos (POO) no **R** é o sistema *S3*, um sistema completamente diferente das ideias sobre orientações a objetos de outras linguagens. Portanto, como iniciante sobre POO, recomendamos um estudo teórico sobre esse paradigma, e posteriormente, o estudo sobre o sistema *S3*, para entender a real diferença. De todo modo, não é o objeto estudar POO, para este momento. Apenas, enfatizamos a importância do atributo `class` para os objetos em **R**.

O tipo da classe implícita pode ser `numeric`, `logical`, `character`, `list`, `matrix` ou `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()`.

Por exemplo, quando criamos um objeto da classe `data.frame`, vejamos o que acontece quando removemos este atributo no Código R 4.7.

Observamos que sem o atributo `class = "data.frame"`, o objeto tem classe implícita `list`. Como consequência o objeto tem uma estrutura em forma de lista, mas se comporta como um `data.frame`, que se apresenta como mostrado anteriormente. Um outro atributo interessante apresentado no Código R 4.7 foi o “`row.names`”, o nome das

linhas. Apesar de existir um objeto do tipo “symbol”, que representa o nome associado aos objetos, temos também um atributo “names”, que representaria neste caso os nomes dos elementos em um objeto. A forma de obtermos o atributo “names” é pela chamada de função names(), Código R 4.8.

Código R 4.7

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

```
7 # Objeto sem efeito "class"  
8 dados2 <- unclass(dados); dados2
```

Console R:

```
$a
[1] 1 2 3
$b
[1] "A" "B" "C"
attr(,"row.names")
[1] 1 2 3
```

Script R:

```
10 # Qual a classe implicita?
11 class(dados2)
```

Console R:

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

Código R 4.8**Console R:**

```
> # Criamos um objeto de classe "data.frame"
> dados <- data.frame(a = 1:3, b = LETTERS[1:3])
> # Atributo Nome de "dados"
> names(dados)
[1] "a" "b"
> # Alterando o atributo nome
> names(dados) <- c("num", "letras")
> # Verificando a alteracao
> names(dados)
[1] "num"    "letras"
> dados
  num letras
 1   1      A
 2   2      B
 3   3      C
```

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

4.2.1 Atributos intrínsecos

Todos os objetos têm dois *atributos intrínsecos*: o **modo** e **comprimento**. O **modo** ou **tipo** é o atributo que iniciamos a falar no começo do capítulo. Para o caso dos vetores atômicos, o **modo** dos vetores podem ser cinco: numérico (`numeric`), lógico (`logical`), caractere⁹ (`character`), complexo (`complex`) ou bruto (`raw`). A este último não daremos evidência para o momento, lembrando que o tipo bruto está relacionado à linguagem S. Já o atributo **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 4.9.

Código R 4.9

Script R:

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

Console R:

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

Script R:

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

Console R:

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

⁹sinônimo: *string*, cadeia de caracteres.

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Contudo, a função `mode()` se baseou na tipagem da linguagem S. Temos uma outra função para verificar o **modo** do objeto, que é por `typeof()`. O atributo **modo** retornado de um objeto por `typeof()`, está relacionado com a tipagem da linguagem C, Código R 4.10. Boa parte das rotinas internas no **R** estão nessa linguagem, principalmente as funções primitivas do pacote **base**. A nossa referência principal será baseada em `typeof()`.

Observamos que apesar de alguns vetores serem vazios, eles ainda têm um modo, que pode ser observado no Código R 4.11.

Código R 4.10

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Código R 4.11**Script R:**

```
1 numeric(0) # Vetor de comprimento 1
```

Console R:

```
numeric(0)
```

Script R:

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

Console R:

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

Script R:

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

Console R:

```
character(0)
```

Script R:

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

Console R:

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

A diferença existente nos objetos y e y2 para a função `typeof()` se refere apenas como o **R** armazena essas informações na memória do computador, Código 4.10. Para identificarmos se dois números são iguais no **R**, usamos a seguinte forma sintática:

Console R:

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

Observamos que o resultado é TRUE significa que os objetos são iguais. Agora, vejamos a próxima linha de comando:

Console R:

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

Obtivemos agora o resultado FALSE, significando 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 criados.

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”, que representa um número inteiro. Estes termos são utilizados na linguagem C. Já a linguagem S não os diferencia, tudo é numeric.

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

- Pode significar um número real; para a computação um número de dupla precisão (`numeric` e `double` seriam iguais neste caso), Código R 4.12;

Código R 4.12

Script R:

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

Console R:

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

Script R:

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

Console R:

```
[1] TRUE
```

- nos sistemas S3 e S4 (orientação a objetos), o termo numérico é usado como atalho para o modo “integer” ou “double”. Con tudo, vejamos o Código R 4.13;

Código R 4.13

Script R:

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

Console R:

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

Script R:

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

Console R:

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

- Pode ser utilizado (`is.numeric()`) para verificar se determinados objetos têm 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 do objeto de classe factor podem ser números ou caracteres, mas se comportarão sempre como numérico, como pode ser observado no Código R 4.14.

Código R 4.14

Script R:

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

Console R:

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

Script R:

```
3 # Sem a classe "factor" o objeto retorna 1  
4 unclass(fator)
```

Console R:

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

Script R:

```
5 # Verificando o modo/tipo do objeto  
6 mode(fator); typeof(fator)
```

Console R:

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

Script R:

```
7 # Nao se comporta como numerico  
8 is.numeric(fator); is.integer(fator)
```

Console R:

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

A Tabela 4.2 a seguir, mostra o retorno dos seis principais tipos de objetos para vetores atômicos.

Tabela 4.2: Tipo de objetos para vetores atômicos.

typeof()	mode()
logical	logical
integer	numeric
double	numeric
complex	complex
character	character
raw	raw

O atributo **comprimento** do objeto é o segundo tipo de atributo intrínseco, e pode ser obtido pela função `length()`, do qual a representação na Figura 4.4, informa o atributo. Vejamos as linhas de comando no Código R 4.15.

Código R 4.15

Script R:

```
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 R:

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

Apresentamos um diagrama representando os três objetos criados no Código R 4.15, Figura 4.4. Observamos agora o acréscimo do **com-**

primeiro dos objetos no diagrama entre colchetes, ao lado do atributo **modo**. Usaremos sempre a tipagem baseada em `typeof()` para essas representações.

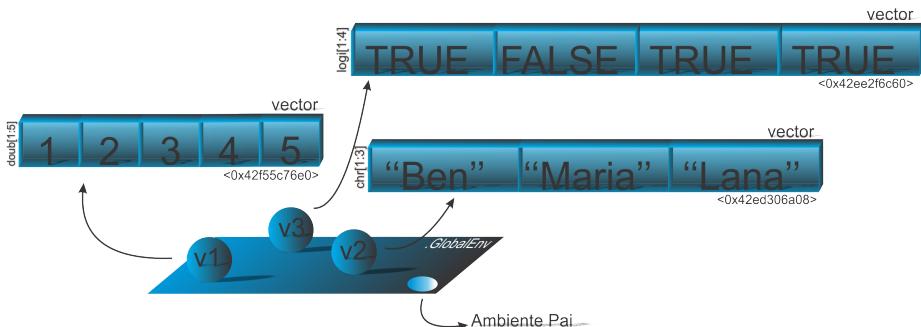


Figura 4.4: Objetos v_1 , v_2 e v_3 .

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

- `base::class()` e `sloop::s3_class()`: Qual a classe do objeto no sistema S3?
- `base::mode()`: Qual a tipo dos objetos baseados na linguagem S?
- `base::typeof()`: Qual a tipo dos objetos 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 ao caminho de busca. Este assunto será abordado mais à 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 tipo “character” estiver em um vetor, todos os demais elementos serão convertidos para o tipo (ou modo), como segue:

Console R:

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

Todos os elementos ganharam aspas (duplas ou simples), 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. Para o caso dos tipos “integer” e “double”, veremos que o primeiro será sempre coagido ao segundo.

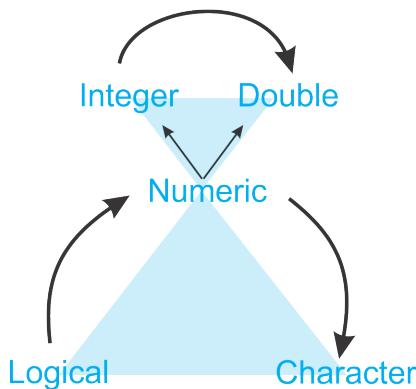


Figura 4.5: Coerção de vetores para os tipos de objetos “numeric” (“integer” e “double”), “character” e “logical”.

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.<nome_função>()” com prefixo “as.”, isto é, se desejarmos que um objeto meu_objeto tenha o modo “character”, usamos as.character(meu_objeto). Para sabermos se um objeto é de um determinado modo, usamos as funções do tipo is.<nome_função>(), com o prefixo “is.”, como já falado anteriormente. Vejamos o Código R 4.16, para elucidar o que discutimos anteriormente.

Código R 4.16**Script R:**

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

Console R:

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

Script R:

```
4 # Coercão do objeto para modo "character"
5 minha_idade <- as.character(minha_idade)
6 mode(minha_idade)
```

Console R:

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

Script R:

```
7 # Verificando se o objeto tem modo "character"
8 is.character(minha_idade)
```

Console R:

```
[1] TRUE
```

4.4 Estruturas de dados

Agora, pretendemos falar sobre as principais estruturas de dados no **R**. Falamos anteriormente sobre a estrutura mais simples, que é o vetor atômico. Mas entendemos que um vetor em **R** podem ser considerado

como: atômico, lista ou expressão, e estruturalmente subdivididos da seguinte forma:

- Vetores atômicos:
 - lógicos, numéricos e caracteres;
 - matrizes unidimensionais (*matrix*) e multidimensionais (*arrays*);
- vetores em listas:
 - listas (*Lists*);
 - quadro de dados (*Data frames*);
- Vetores em expressões:
 - expressões (objetos do tipo “expression”).

Existem outros tipos de estruturas, mas para este *Volume*, exploraremos as estruturas mais utilizadas nos pacotes nativos do **R**. As coerções para as estruturas de dados são similares as funções de coerção para modo, isto é, usar as funções prefixadas as `.<nome_ função>`. Apenas por documentação, apresentamos os vetores em expressões neste momento, porém abordados no *Volume II*. Apesar, das listas serem um vetor mais complexo, na Tabela 4.3, apresentamos uma visão geral dos objetos apresentados até o momento, e separamos a estrutura lista de um vetor, devido as características mais complexas da primeira em relação a segunda.

4.4.1 Vetores

Existem cinco tipos principais de vetores atômicos:

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

Tabela 4.3: Caracterização de objetos estruturado para armazenamento de dados.

Estrutura de dado	Classe ou Classe implícita	Modo	São possíveis vários modos na mesma estrutura de dado?
Vetor	numeric (integer ou double) character, complex, logical, raw, expression, list	numeric (integer ou double), character, complex, logical, raw, expression, list	Não
Matriz	matrix, numeric (integer ou double) character, complex, logical, raw, expression, list	numeric (integer ou double) character, complex, logical, raw, expression, list	Não
<i>Array</i>	array, numeric (integer ou double) character, complex, logical, raw, expression, list	numeric (integer ou double) character, complex, logical, raw, expression, list	Não
Lista	list	list	Sim
Quadro de dados	data.frame	list	Sim

Temos dois tipos raros que são os complexos (`complex`) e brutos (`raw`), que não serão abordados. Os vetores atômicos são a estrutura de dados mais simples que existe no ambiente **R**. As demais estruturas derivadas dos vetores atômicos se modificam à medida que acrescentamos mais atributos. Por exemplo, os fatores são vetores numéricos, cujo atributo que os modifica de um vetor, é o atributo `class = "factor"`, além de um outro atributo chamado `levels` que representa os níveis; as matrizes deixam de ser um vetor atômico convencional por causa do atributo “`dim`”; e assim percebemos que as estruturas vão sendo criadas de acordo com que os atributos são incorporados aos objetos.

4.4.1.1 Vetores escalares ou constantes

O menor comprimento de um vetor é de tamanho `um`, conhecido também como um escalar ou constante. 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. Neste 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 duplas, isto é, `"Ben"`, `"a"`. Pode ser utilizado também aspa simples, `'Ben'`, `'a'`, etc.

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). O termo foi introduzido por Goldberg em 1985, pelo Instituto de Engenheiros Eletricistas 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 exemplo apresentamos o Código R 4.17, e 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 numérico, lógico ou caractere, e neste último caso, não ocorre coerção, apenas as informações de que o primeiro elemento do vetor está ausente, porém, o vetor ainda continua sendo de mesmo tipo de objeto. Desse modo, vamos perceber `NaN` em operações matemáticas que a solução é indeterminada, tais como pode ser visto no Código R 4.17. 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 4.18.

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

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

(continuação...) Qual a diferença entre NaN, NA e NULL?



Agora, observamos 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 a ideia também segue ao ambiente **R**, pois apesar de não sabermos o resultado, o valor elevado a zero será 1, como pode ser verificado no Código R 4.19. Para identificar os 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 à frente.

Por fim, apresentamos a distinção do objeto NULL, isso mesmo um objeto de tipo NULL, mais precisamente NILSXP, e também uma palavra reservada que não é considerado um vetor escalar. Diferentemente de NaN, o objeto NA 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. Este 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 e/ou NA são levados em consideração. No caso, NULL representa valor não existente, e como não há atributo envolvido, este não é contabilizado. Um exemplo pode ser observado no Código R 4.21.

Código R 4.17**Script R:**

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

Console R:

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

Script R:

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

Console R:

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

Código R 4.18**Script R:**

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

Console R:

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

Script R:

```
2 0 / 0
```

Console R:

```
[1] NaN
```

Script R:

```
3 Inf - Inf
```

Console R:

```
[1] NaN
```

Código R 4.19

Script R:

```
1 NA ^ 0
```

Console R:

```
[1] 1
```

Script R:

```
2 NaN ^ 0
```

Console R:

```
[1] 1
```

Código R 4.20**Script R:**1 `NA + 1`**Console R:**

[1] NA

Script R:2 `NaN + 5`**Console R:**

[1] NaN

Script R:3 `NA * 5`**Console R:**

[1] NA

Script R:4 `sqrt(NaN)`**Console R:**

[1] NaN

Script R:5 `NaN + NA`

Console R:

```
[1] NaN
```

Script R:

```
6 NA + NaN
```

Console R:

```
[1] NA
```

Código R 4.21

Script R:

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

Console R:

```
[1] 3
```

Script R:

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

Console R:

```
[1] 3
```

Script R:

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

Console R:

```
[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 4.22.

Código R 4.22

Script R:

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

Console R:

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

Script R:

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

Console R:

```
[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 os elementos sejam inteiros, como pode ser visto no Código R 4.23.

Código R 4.23**Script R:**

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

Console R:

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

Script R:

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

Console R:

```
[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>, como segue:

Console R:

```
> # 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 outras funções para construir sequências regulares. Percebemos que os três objetos são iguais, como pode ser observado na sequência:

Console R:

```
> 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 virtual, como observamos no Código R 4.24¹⁰.

Código R 4.24

Script R:

```
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 # Memoria
6 lobstr::obj_size(vetor.num)
```

Console R:

```
96 B
```

Script R:

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

Console R:

```
80 B
```

Script R:

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

¹⁰Este código foi executado no *RGui x64*, versão 4.2.

Console R:

```
680 B
```

Observamos um objeto do tipo “double” que precisa de mais memória para armazenar os valores do que o objeto do tipo “integer”. O último objeto (`vetor.num3`), gerado pela chamada `` : `()`, aparentemente ocupa mais memória. Porém, essa função apresenta um recurso interessante inserido nas versões posteriores ao **R** (3.5.0), que é chamado de **abreviação alternativa**. Este 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 a sequência gerada em `vetor.num3` é do tipo “integer”.

Dessa forma, percebemos no objeto `vetor.num3` uma economia de memória, dependendo do tamanho do vetor, quando se compara com as outras formas sintáticas, isto é:

Console R:

```
> # 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)) # equiv. a c(1, ..., 10)
96 B
> lobstr::obj_size(c(1:10000)) #equiv. a c(1, ..., 10000)
40,048 B
> lobstr::obj_size(c(1:1000000)) #equiv. a c(1, ..., 1000000)
4,000,048 B
```

Outras funções são usadas para criar sequências de números, tais como: `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 4.25.

Código R 4.25**Script R:**

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

```
11 # Vetor repetido ate uma seq. de tamanho 7  
12 rep(x = 1:3, length.out = 7)
```

Console R:

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

Script R:

```
13 # Versao mais rapida de rep  
14 # sequencia de tamanho 15  
15 rep_len(x = 1:10, length.out = 15)
```

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

```
[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 R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

```
30 # Versao mais rapida para seq(), igual a 1:length(w)  
31 w <- c(4, 3, 6, 9)  
32 seq_along(w)
```

Console R:

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

Script R:

```
34 # Versao mais rapida para seq, igual a 1:4  
35 seq_len(4)
```

Console R:

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

Script R:

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

Console R:

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

Script R:

```
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 R:

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

Script R:

```
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 R:

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

Script R:

```
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 R:

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

Script R:

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

Console R:

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

Script R:

```
46 # Repetir seq(3) 2 vezes e agrupar em lista  
47 replicate(n = 5, seq(3), simplify = FALSE)
```

Console R:

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

Script R:

```
48 # Repetir rnorm(10) cinco vezes e agrupar em matriz  
49 replicate(n = 3, expr = rnorm(3), simplify = TRUE)
```

Console R:

```
[,1] [,2] [,3]  
[1,] 0.04564093 0.31394292 -0.7193373  
[2,] -0.91144307 1.15547056 0.5652449  
[3,] -0.32603749 2.56355701 0.2840667
```

Script R:

```
50 # Repetir rnorm(10) cinco vezes e agrupar em lista
51 replicate(n = 3, expr = rnorm(3), simplify = FALSE)
```

Console R:

```
[[1]]
[1] 2.05622089 -0.95096153  0.37797923
[[2]]
[1] -0.55620379  1.75424840 -0.05765807
[[3]]
[1] -1.23585520 -0.78408176 -1.13901273
```

Script R:

```
52 # Analise de experimento
53 # 2 niveis com 3 repeticoes
54 gl(n = 2, k = 3, labels = c("Control", "Treat"))
```

Console R:

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

Script R:

```
55 # Analise de experimento (20 parcelas e 2 níveis)
56 gl(n = 2, k = 1, length = 20)
```

Console R:

```
[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 R:

```
58 # Analise de experimento (20 parcelas e 2 níveis)
59 gl(n = 2, k = 2, length = 20)
```

Console R:

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

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 tipo de objeto com estrutura de vetor, exceto para raw. Há constantes lógicas NA específicas para tipos de objetos 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 o NA específico. Essa constante contida no vetor não altera o tipo de objeto, isto é,

Console R:

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

Criamos vetores atômicos iniciais sem nenhum elemento (vetor vazio), por meio das funções numeric(0), character(0) e logical(0), isto é,

Console R:

```
> # Vetor numerico de comprimento 0
> v1 <- numeric(0); length(v1)
```

```
[1] 0
> v2 <- character(0); length(v2)
[1] 0
> v3 <- logical(0); length(v3)
[1] 0
```

Para inserirmos valores aos vetores, usamos o sistema de indexação, cujo contador¹¹ se inicia a partir do número 1, no ambiente **R**. Vejamos:

Console R:

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

Assim, como Exercício 4.2, completamos o exemplo do código anterior para os dois outros vetores restantes. Uma vez criado o vetor, se desejarmos acessar os elementos, usamos também o sistema de indexação:

Console R:

```
> # Vetor numerico de comprimento 0
> v1 <- numeric(0)
> v2 <- character(0)
> v3 <- logical(0)
> # Inserimos 3 elementos em v1 e imprimindo o 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)]
```

¹¹Diferente de outras linguagens, como a C, que o contador começa do número 0.

```
[1] 3 10
[1] 3 10
> # Imprimindo todos
> v1
[1] 5 3 10
```

Se estivermos interessados na posição dos valores, usamos funções como: `which()`, `which.max()`, `which.min()`, `match()`, dentre outras. Vejamos o Código R 4.26.

Código R 4.26

Script R:

```
1 # Semente
2 set.seed(10)
3 # vetor
4 amostra <- sample(1:10, 10); amostra
```

Console R:

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

Script R:

```
5 # Olhamos para a posicao do valor 1 em amostra
6 which(amostra == 1)
```

Console R:

```
[1] 10
```

Script R:

```
8 # Olhamos para a posicao do maximo
9 which.max(amostra)
```

Console R:

```
[1] 7
```

Script R:

```
10 # Olhamos para a posicao do minimo  
11 which.min(amostra)
```

Console R:

```
[1] 10
```

Script R:

```
12 # Vamos mudar "amostra"  
13 amostra[5] <- 10; amostra
```

Console R:

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

Script R:

```
14 amostra[9] <- 1; amostra
```

Console R:

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

Script R:

```
15 # Percebemos que existe mais de um minimo e mais de  
16 # um maximo. Essa funcao observa apenas o primeiro  
17 # maximo  
18 which.max(amostra)
```

Console R:

```
[1] 5
```

Script R:

```
17 # Essa função observa apenas o primeiro mínimo  
18 which.min(amostra)
```

Console R:

```
[1] 9
```

Script R:

```
18 # Alternativa  
19 which(amostra == max(amostra))
```

Console R:

```
[1] 5 7
```

Script R:

```
20 which(amostra == min(amostra))
```

Console R:

```
[1] 9 10
```

Script R:

```
21 # Usamos "match" para verificar a posição de  
22 # "1" em "amostra". Se existir mais de um,  
23 # identificaria a posição apenas o primeiro "1"  
24 match(1, amostra)
```

Console R:

```
[1] 9
```

Script R:

```
23 # Usamos a função match() para verificar
24 # as posições do vetor "c(1,2)" em "amostra"
25 # Se existir, identificara a posição dos elementos
26 # do vetor
27 match(c(1, 2), amostra)
```

Console R:

```
[1] 10 6
```

Script R:

```
25 # Saber se apenas existe o valor "1" na amostra
26 1 %in% amostra # Resultado lógico
```

Console R:

```
[1] TRUE
```

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, Código R 4.27.

Código R 4.27

Script R:

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

Console R:

```
[1] 5
```

Script R:

```
3 # O operador "+" eh uma chamada de funcao  
4 `+`(2, 3)
```

Console R:

```
[1] 5
```

Script R:

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

Console R:

```
[1] 1
```

Script R:

```
7 # O operador "-" eh uma chamada de funcao  
8 `-(3, 2)
```

Console R:

```
[1] 1
```

Script R:

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

Console R:

```
[1] 6
```

Script R:

```
11 # O operador "*" eh uma chamada de funcao
12 `*`(3, 2)
```

Console R:

```
[1] 6
```

Script R:

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

Console R:

```
[1] 1.5
```

Script R:

```
15 # O operador "/" eh uma chamada de funcao
16 `/`(3, 2)
```

Console R:

```
[1] 1.5
```

Essas mesmas operações podem ser realizadas elemento a elemento para vetores de comprimento maior que 1, que pode ser observado nas

linhas de comando que segue:

Console R:

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

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

Console R:

```
> # 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 :
  comprimento do objeto maior não é múltiplo do comprimento
  do objeto menor
```

Para que a soma dos vetores fossem realizadas, foi necessário repetir os primeiros elementos do segundo vetor, de modo que o comprimento se tornasse igual ao primeiro. Após isso, a soma é realizada elemento a elemento. Este procedimento ocorre com os demais tipos de operações.

Agora, se desejarmos a operação $\mathbb{X}^t \mathbb{Y}$, isto é:

$$\mathbb{X}^t \mathbb{Y} = [4 \ 5 \ 6] \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = 4 \times 1 + 5 \times 2 + 6 \times 3 = 32, \quad (4.1)$$

onde $\mathbb{X}^t = [4, 5, 6]$ e $\mathbb{Y} = [1, 2, 3]^t$, são vetores ou matrizes 3×1 , temos uma operação com matrizes. Em **R**, usamos a função `%^%` para que a multiplicação de vetores seja calculada como operação com matrizes.

Matematicamente, o símbolo “ t ” representa a transposta de uma matriz, que em R, usamos a função `t()`. Para realizarmos essa operação em R, vejamos o Código 4.28.

Código R 4.28

Script R:

```
1 # Vetores
2 x <- 4:6
3 y <- 1:3
4 # Operacao matricial
5 x %*% y
```

Console R:

```
[,1]
[1,] 32
```

Algumas outras operações podem ser realizadas de acordo com as funções apresentadas na Tabela 4.4. Para mais funções, acessem o manual *An Introduction to R* (VENABLES; SMITH; R CORE TEAM, 2022), ou execute no `console ?Arithmetric`.

Tabela 4.4: Operadores aritméticos e algumas outras funções matemáticas.

Função (Ou operador)	Finalidade
<code>+</code>	Soma unária, por exemplo <code>(+ 4)</code> , ou binária entre dois objetos numéricos
<code>-</code>	Subtração unária, por exemplo <code>(- 3)</code> , ou binária entre dois vetores
<code>*</code>	Multiplicação entre dois vetores
<code>/</code>	Divisão entre dois vetores
<code>^ ou **</code>	Exponenciação binária, isto é <code>2^3</code> ou <code>2 ** 3</code>
<code>%/%</code>	Parte inteira da divisão
<code>%%</code>	Restante da divisão
<code>sum()</code>	Soma de elementos de um vetor
<code>prod()</code>	Produtório dos elementos de um vetor

<code>sqrt()</code>	Raiz quadrada dos elementos de um vetor
<code>log()</code>	Função Logaritmo neperiano
<code>log10()</code>	Função Logaritmo na base 10
<code>exp()</code>	Função exponencial
<code>mean()</code>	Média dos elementos de um vetor
<code>sd()</code>	Desvio padrão dos elementos de um vetor
<code>var()</code>	Variância dos elementos de um vetor
<code>median()</code>	Mediana dos elementos de um vetor
<code>round()</code>	Arredondamento de vetor numérico. Outros tipos são: <code>trunc()</code> , <code>floor()</code> e <code>ceiling()</code>

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 4.5.

Tabela 4.5: Operadores lógicos.

Op. lógico	Sintaxe	Pergunta
<code><</code>	<code>a < b</code>	a é menor que b?
<code>></code>	<code>a > b</code>	a é maior que b?
<code>==</code>	<code>a == b</code>	a é igual b?
<code>!=</code>	<code>a != b</code>	a é diferente b?
<code>>=</code>	<code>a >= b</code>	a é maior ou igual a b?
<code><=</code>	<code>a <= b</code>	a é menor ou igual a b?
<code>%in%</code>	<code>"a" %in% c()</code>	O elemento "a" está no vetor c()?

A operação binária significa que a função exige dois argumentos (ou operandos), isto é, <Arg1> <Operador> <Arg2>. Esta é uma outra forma sintática para funções (operadores binários ou unários). Para mais detalhes, usamos no *console* a chamada `?Syntax`. Vejamos alguns exemplos, no Código R 4.29.

Código R 4.29**Script R:**

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

Console R:

```
[1] FALSE
```

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

```
[1] FALSE
```

Script R:

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

Console R:

```
[1] FALSE FALSE
```

Script R:

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

Console R:

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

Script R:

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

Console R:

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

O interessante no operador `%in%`, que na realidade é uma função

com dois argumentos, é a sua forma sintática de operadores binários especiais do tipo %<nome_sintatico>%, em código de usuário; este tipo de função é um dos mais conhecidos hoje na análise de dados, representado pelo operador operador *pipe* (%>%) do pacote **magrittr** (pertencente à família de pacotes **tidyverse**)¹². A diferença no operador *pipe* para %in%, é que o segundo operando (Argumento 2) é uma função que recebe no primeiro argumento o operando 1 (Argumento 1). Isto cria uma forma sintática de realizar chamadas sequenciais para a obtenção de um resultado, o que acaba simplificando o código de um analista de dados, por exemplo. Este operador acabou ganhando tanta visibilidade na comunidade R, que na versão R (>= 4.1.0), foi implementado o operador *pipe* nativo (|>). Veremos mais detalhes na seção sobre criação de funções, no Capítulo 6.

4.4.1.6 Operadores booleanos

Os operadores booleanos avaliam diversas operações lógicas (condições) para ao final retornar um TRUE ou FALSE, sendo consultado na Tabela 4.6.

Tabela 4.6: Operadores booleanos.

Operador	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 falsa a cond1?
any()	any(cond1, cond2, ...)	Alguma das condições é verdadeira?
all()	all(cond1, cond2, ...)	Todas as condições são verdadeiras?

¹²Para instalar todos os pacotes da família **tidyverse**, usamos a chamada `install.packages("tidyverse")`

Vejamos alguns exemplos de operadores booleanos, no Código R 4.30. Deixamos como sugestão de exercício, o desenvolvimento de rotinas para as demais funções.

Código R 4.30

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Utilizamos os operadores lógicos e booleanos dentro do sistema de indexação dos vetores, facilitando assim a busca pelos elementos que satisfaçam a condição desejada. Vejamos o Código 4.31 para detalhes

desses operadores.

Código R 4.31

Script R:

```
1 x <- 1:10 # Vetor
2 # Operacao logica dentro do sistema de indexacao
3 x[x > 6]
```

Console R:

```
[1] 7 8 9 10
```

Script R:

```
5 # Operacao logica dentro do sistema de indexacao
6 x[x < 3 | x > 7]
```

Console R:

```
[1] 1 2 8 9 10
```

4.4.2 Matrizes unidimensionais

Ao abordar as próximas estruturas de dados daqui para frente, desde matrizes até quadro de dados (*data frame*), não serão apresentadas todas as manipulações possíveis. Mostraremos apenas como criá-las e seus atributos. Assim, não apresentaremos exemplos com as funções específicas para matrizes, por exemplo, porque não é o propósito deste *Volume*. 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, estruturas de dados do tipo matrizes ou *arrays*. Assim como falamos anteriormente, o atributo pode mudar a estrutura do objeto e o comporta-

mento de como as funções observam os dados. Vejamos o Código R 4.32, para o entendimento desse tipo de estrutura.

Código R 4.32

Script R:

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

Console R:

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

Script R:

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

Console R:

```
NULL
```

Script R:

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

Console R:

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

Script R:

```
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 R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

O atributo `dim` recebeu uma informação bidimensional, isto é, o número de linhas e colunas, respectivamente. Uma outra forma de

construir uma matriz é usando a função `matrix()`, que pode ser apresentado no Código R 4.33.

Como verificado no Código R 4.33, *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]`.

Código R 4.33

Script R:

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

Console R:

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

Script R:

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

Console R:

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

4.4.3 Matrizes multidimensionais

A ideia do objeto matriz multidimensional (ou *array*) é similar ao da matriz unidimensional, a diferença é que agora é um vetor atômico de mais de duas dimensões, como pode ser observado no Código R 4.34.

Código R 4.34

Script R:

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

Console R:

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

Script R:

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

Console R:

```
NULL
```

Script R:

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

Console R:

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

Script R:

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

Console R:

```
, , 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 R:

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

Console R:

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

Script R:

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

Console R:

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

Criamos duas matrizes de dimensão (2 x 3). Para acessar os elementos do objeto, usamos 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 uma estrutura *array* é usar a função `array()`, isto é:

Console R:

```
> # 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 4.7. Para auxílio nas funções, devemos usar o símbolo de interrogação (?) antes das funções que desejamos entender, devendo o comando ser executado no *console*. Por exemplo, para pedir ajuda sobre a função `names()` que retorna o atributo `names`, devemos usar a chamada `?names()`.

Apesar de não apresentarmos como manipular matrizes no **R**, para este *Volume*, deixaremos algumas funções realizadas em matrizes, disponíveis nos pacotes nativos do **R**, Tabela 4.8.

A base do algoritmo das funções para matrizes desenvolvidas nos pacotes nativos do **R** veio da linguagem **S** (BECKER; CHAMBERS; WILKS, 1988), como também do *software* de código aberto LAPACK, que pode visto em <https://netlib.org/lapack/>. Devido aos comple-

Tabela 4.7: Funções equivalentes para vetores, matrizes e *arrays*.

Vetor	Matriz	Array
names()	rownames(), colnames()	dimnames()
length()	nrow(), ncol() rbind(), cbind()	dim()
c()		abind::abind()
Sem função!	t()	aperm()
is.vector()	is.matrix()	is.array()

xos cálculos computacionais, a base implementada dos códigos em LAPACK está em C e FORTRAN. Uma outra alternativa para uso de funções nos problemas relacionados à álgebra linear, recomendamos a utilização do pacote **RcppEigen**, uma interface para o **R** cuja a implementação foi toda desenvolvida em C++. Para isso, também é necessário estudar o pacote **Rcpp** uma interface C/C++ para o **R**.

Tabela 4.8: Funções para matrizes.

Funções	Finalidade
+	Soma de duas matrizes elemento a elemento
-	Subtração de duas matrizes elemento a elemento
*	Multiplicação de duas matrizes elemento a elemento
/	Divisão de duas matrizes elemento a elemento
t()	Transposta de uma matriz
%*%	Multiplicação de duas matrizes
crossprod()	Equivalente a $t(A) \%*% B$, sendo A e B duas matrizes
tcrossprod()	Equivalente a $A \%*% t(B)$, sendo A e B duas matrizes
%x%	Produto de Kronecker de duas matrizes
det()	Determinante de uma matriz

<code>solve()</code>	Inversa de uma matriz
<code>rank()</code>	<i>Rank</i> de uma matriz
<code>qr()</code>	Decomposição QR de uma matriz
<code>diag()</code>	Matriz diagonal e matriz identidade
<code>eigen()</code>	Cálculo dos autovalores e autovetores de uma matriz
<code>svd()</code>	Decomposição do valor singular de uma matriz
<code>chol()</code>	Decomposição de Cholesky de uma matriz

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 funções, expressões, ou outras listas. Esta última é o que chamamos de objetos recursivos, o fato dos elementos da lista poder conter também uma lista, por exemplo. Isso acontece também com funções, dentre outros objetos. Para sabermos se um objeto é recursivo, usamos a chamada `is.recursive()`, Código R 4.35, *linha 6*.

A forma de se obter uma lista é pela função `list()`. Também, diferentemente do vetor como estrutura de dado, a lista, além de ser uma estrutura de dado, é um tipo de objeto específico que pode ser verificado por `typeof()` ou `mode()`. Internamente, a tipagem C do objeto é “VECSXP”, e com representação em **R** como “list”. Vejamos os comandos no Código R 4.35.

Acessamos ou alteramos 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 da lista 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 terceiro elemento. Os elementos de um lista são na realidade outros objetos, do qual conseguimos acessar também os seus elementos.

Código R 4.35**Script R:**

```

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

```

Console R:

```

[[1]]
[1] 1 2 3

[[2]]
[1] "e"

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

[[4]]
function (x, ...)
UseMethod("mean")
<bytecode: 0x00000000008e55c60>
<environment: namespace:base>

[[5]]
expression(x ~ y)

```

Script R:

```
6 is.recursive(l0)
```

Console R:

```
TRUE
```

Quando nominamos os objetos contidos nas listas, utilizamos o

operador \$ para acessar os elementos. Vejamos o Código R 4.36.

As listas têm importâncias diversas dentro do ambiente **R**, por exemplo, o atributo em um objeto é armazenado em forma de lista; a coerção entre vetor e lista, sempre força um vetor atômico a uma lista; os argumentos de uma função também são armazenados como uma lista, dentre outras. Vejamos as linhas de comando no Código R 4.37.

Código R 4.36

Script R:

```

1 # Criando uma lista
2 l0 <- list(l01 = 1:3,
3             l02 = letters[5],
4             l03 = list(1, 2, 3),
5             l04 = mean,
6             l05 = expression(x ~ y))
7 # Imprimindo o primeiro elemento da lista "l0"
8 l0$101
9 # Imprimindo o segundo
10 l0$102

```

Console R:

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

Script R:

```

11 # Imprimindo o segundo
12 l0$102

```

Console R:

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

Script R:

```
13 # Imprimindo o segundo, usando a indexacao e o
14 # nome do elemento
15 10[["102"]]
```

Console R:

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

Código R 4.37

Script R:

```
1 # Vejamos as linhas de comando
2 l1 <- list(list(1, 2), c(3, 4))
3 l2 <- c(list(1, 2), c(3, 4))
4 # Vejamos as suas estruturas
5 str(l1)
```

Console R:

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

Script R:

```
6 str(l2)
```

Console R:

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

Observamos no objeto 11 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 a lista dentro da função `c()`, os seus elementos fariam parte de um vetor. O que temos, na realidade, é uma coerção do vetor para uma lista, e portanto, temos uma lista 12 de quatro elementos.

4.4.5 Quadro de dados

O quadro de dados (*Data frame*) é uma estrutura de dados, sendo um objeto do tipo lista com classe `data.frame`. Essa estrutura apresenta 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 o mesmo modo no quadro de dados. Antes, os vetores em modo caractere eram convertidos em objetos de classe “factor”. Para convertê-lo automaticamente, devemos usar o argumento `stringsAsFactors = TRUE`. Como sugestão, preferimos 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 quadro de dados (*data frame*), usamos a função `data.frame()`. Assim como nas listas, inserimos os objetos no quadro de

dados inserindo o nome nas colunas ou não. A forma de acessar os elementos é interessante, usamos a indexação de uma lista ou de uma matriz. Vejamos o Código R 4.38.

Código R 4.38

Script R:

```
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 R:

	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 R:

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

Console R:

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

Script R:

```
9 dados$x
```

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

```
[1] 1
```

Script R:

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

Console R:

```
[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 tipo de estrutura de dados que teremos é

um quadro de dados (*data frame*), assunto discutido mais à frente. A semelhança com a forma retangular de uma matriz faz com que algumas funções para manipular matrizes sejam utilizadas em quadro de dados, como por exemplo:

- 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, utilizamos a função `attach()` para que os elementos do quadro de dados estejam disponíveis (anexados) ao caminho de busca, e assim, possamos acessar os elementos (ou objetos) do quadro de dados sem precisar mencioná-lo, cujo exemplo pode ser visto no Código R 4.39.

Código R 4.39

Script R:

```
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 R:

```

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 R:

```

7 # Usando attach()
8 attach(dados)
9 x; y; z

```

Console R:

```

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

```

A função `attach()` tem implicações, quando por exemplo se deseja inseri-la na construção de um pacote **R**. Para desanexar o quadro de dados, usamos a função `detach()`. A função `attach()` pode ser usada em qualquer objeto lista, sendo anexado ao caminho de busca¹³.

Existem outros tipos de objetos, que ainda serão abordados em capítulos específicos, como por exemplo, funções, ambientes, etc.. Alguns não serão abordados neste momento, mas o importante é ter destacado no capítulo, as diferenças existentes entre objetos e estrutura

¹³Para ver o caminho de busca, usamos a função `search()`.

de dados, para uma escrita de código mais consciente, podendo assim entender o mecanismo básico do ambiente R.

4.5 Exercícios

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

Dica na página 290

Exercício 4.2: De acordo com a saída do *console*:

Console R:

```
> # Vetor numerico de comprimento 0  
> v2 <- character(0)  
> v3 <- logical(0)
```

acrescente três elementos no vetor v2 e v3, pelo sistema de indexação.

Dica na página 290

Exercício 4.3: Crie um quadro de dados (*data frame*) a partir de uma lista, sem usar a função `data.frame()`.

Dica na página 290

Exercício 4.4: Como verificamos a tipagem dos elementos em uma lista? Podemos afirmar que são do tipo “list”?

Dica na página 290

Exercício 4.5: Qual a diferença entre objetos recursivos e objetos atômicos?

Dica na página 290

Exercício 4.6: De acordo com a seguinte linha de comando:

Console R:

> 1:10

Como não associamos nenhum nome a este objeto, podemos recuperá-lo? Em que situações isso é possível?

Dica na página 290

Exercício 4.7: Crie um quadro de dados e posteriormente, acrescente mais uma coluna ao conjunto de valores. Neste mesmo objeto, se desejássemos alterar o nome das colunas, como fariamoss?

Dica na página 290

Exercício 4.8: Como criamos um objeto do tipo lista, cujos elementos não são nomeados? Como acessá-los? Apresente a resposta por meio de exemplos.

Dica na página 290

Exercício 4.9: Vejamos o seguinte *script*:

Script R:

```

1 # Objeto lista
2 dados <- list(
3   mean = mean <- function(...) print("Nada feito!"),
4   y = 1:10,
5   z = rep(c(TRUE, FALSE), 5)
6 )
7 # Quebra do quadro de dados
8 attach(dados)
9 # calcular a media de y
10 mean(y)

```

Por que não foi possível calcular a média para *y*? Será que foi porque não usamos *mean(dado\$y)*? Explique.

Dica na página 291

Exercício 4.10: Vejamos o seguinte Código R 4.40:

Código R 4.40

Script R:

```
1 x <- sample(letters[1:5], 15, replace = TRUE)
2 x <- as.factor(x); x
```

Console R:

```
[1] c d b b e d e a e a b e a a a
Levels: a b c d e
```

Qual a estrutura de dado de x? Qual o tipo de objeto? Apresente os elementos do objeto.

Dica na página 291

Exercício 4.11: De acordo com o seguinte *script*:

Script R:

```
1 x <- c(1, 2, 3, 4, 5); x[3] <- 3
2 y <- c(10, 20, 30, 40, 50); y[6] <- 60
3 x; y
```

Os objetos foram copiados ou não? Como explicar essa condição sintática.

Dica na página 291

Exercício 4.12: Considerando o erro que ocorre no seguinte comando:

Console R:

```
> y <- x = 6
Error in y <- x = 6 : não foi possível encontrar a
função "<-<-"
```

Por que ocorreu esse resultado inesperado?

Dica na página 291

Exercício 4.13: Considerando o seguinte script:

Console R:

```
> x <- c(1, 2, 3, 4, 5)
> lobstr::obj_addr(x)
[1] "0x1a91f540"
> y <- x
> lobstr::obj_addr(y)
[1] "0x1a91f540"
```

Podemos afirmar que x e y são dois objetos diferentes? O que ocorreu nesse processo de execução?

Dica na página 291

Exercício 4.14: Do Exercício 4.13, temos as seguintes alterações realizadas aos seguintes objetos:

Console R:

```
> x[3] <- 10
> y[3] <- 20
> lobstr::obj_addr(x)
[1] "0x1a998570"
> y <- x
> lobstr::obj_addr(y)
[1] "0x1a91f540"
```

Por que o objeto x foi copiado e o objeto y não?

Dica na página 291

Exercício 4.15: De acordo com a Tabela abaixo, temos um conjunto de dados discretos. Estamos interessados em calcular a moda, isto é, o valor de maior frequência. Se existir mais de um valor de maior frequência, e que estas sejam iguais, dizemos que a distribuição de frequência é bimodal, trimodal ou multimodal, para 2, 3 ou mais valores modais, respectivamente. Se todos os valores foram de mesma frequência, a condição é amodal.

X_i	F_i
0	1
1	5
2	2
3	5
4	3
5	5

Sendo assim, desenvolva uma rotina, baseada na Tabela acima, que possa indicar a moda para os dados.

Dica na página 291

Exercício 4.16: Desenvolva uma rotina para verificar se determinado número é par ou ímpar.

Dica na página 291

Exercício 4.17: Como recuperar os objetos criados de um trabalho anterior?

Dica na página 292

Exercício 4.18: Crie uma sequência de valores “1, 2, 3 e 4”, de modo que os valores sejam repetidos dez vezes sequencialmente, e nessa ordem.

Dica na página 292

Exercício 4.19: Identifique o valor máximo e mínimo, em um vetor.

Dica na página 292

Exercício 4.20: Quantos objetos identificamos no seguinte *script*:

Script R:

```
1 x <- 10
```

Dica na página 292

Exercício 4.21: Como identificamos em um quadro de dados:
(i) o número de colunas, (ii) o número de linhas, e (iii) quais os nomes das colunas?

Dica na página 292

Exercício 4.22: Crie dois *scripts*, um auxiliar e outro o que será utilizado para as análises de rotina. No auxiliar, crie uma função (chamada *aux(x)*) que calcula, em linha, a média das observações. Considere que a entrada para essa função seja um quadro de dados, e nele contenha quatro colunas e 10 linhas de valores numéricos. Depois de criado essa função no *script* auxiliar, chame-a no *script* principal. Neste, crie um quadro de dados (*data frame*) com as características mencionadas anteriormente, e execute a função *aux(x)*, carregando-a por meio de *source()* para o *script* principal, em que o argumento *x* recebe o referido quadro de dados criado anteriormente.

Dica na página 292

Exercício 4.23: Para entender a diferença sintática e semântica dos operadores “*<-*” e “*=*”, considere os dois *scripts* abaixo:

Script R:

```
1 matrix(1, nrow = 3)
```

Script R:

```
1 matrix(1, nrow <- 3)
```

Nos dois casos o resultado foi:

Console R:

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

Qual a diferença existente entre os operadores? OBS.: Observamos que os resultados foram iguais, será que realmente existe diferença?

Dica na página 292

Capítulo 5

Importação e exportação de dados

5.1 Introdução

A importação e exportação de dados no ambiente **R** representam situações da análise de dados que seriam facilmente explicadas em poucas linhas. Entretanto, observando o terceiro princípio do **R**, afirmado por Chambers (2016): “Interfaces para outros programas são parte do **R**”, a interação que o ambiente **R** apresenta com outras linguagens o tornou muito versátil, e que se faz necessário a apresentação deste capítulo.

Hoje é uma realidade a interação que o ambiente **R** tem com outros programas. A facilidade em interagir com outras linguagens torna mais complexa a importação e 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 em R Data Import/Export (R CORE TEAM, 2022a). Um outro fator e tema atual é o tamanho dos bancos de dados (*Big Data*), do qual se tem um grande conjunto de variáveis e necessitamos fazer a importação por *APIs*¹, por exemplo, ou outras vias. Temas como estes, serão abordados em outros *Volumes* da coleção *Estudando o Ambiente R*.

Neste momento, limitaremos o 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 dispostos, precisaremos importá-los e armazená-los em um quadro de dados (*data frame*), para que esteja disponível no ambiente global (.GlobalEnv) do **R**, e dessa forma utilizarmos. Ao

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

final do tratamento dos dados, exportamos 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 organizar as variáveis em colunas, com os valores em linhas, Figura 5.1. Sempre a primeira linha das colunas representará o nome das variáveis. Este é outro ponto importante, pois devemos ter a noção que a linguagem irá importar e interpretar esse banco de dados. Quanto mais caracteres diferentes do padrão ASCII, mais difícil poderá ser a leitura dos dados. Assim, sugerimos alguns padrões:

- devemos evitar símbolos fora do padrão alfanumérico;
- devemos evitar o uso de letras minúsculas e maiúsculas. Isso facilitará o acesso a essas variáveis. Contudo, lembramos do padrão de nomes sintáticos observados na Seção 3.3.2;
- como o banco de dados será utilizado para que um programa faça a sua leitura, portanto, deixamos a formatação da apresentação dos dados em arquivo específico, evitando qualquer outro tipo de informação que não seja os dados;
- devemos evitar palavras longas, por exemplo, `segundavariavel` (mau escolha), `segvar` (boa escolha), `seg_var` (boa escolha);
- devemos evitar palavras compostas com espaço entre elas. Como alternativa, usamos o símbolo `_`, por exemplo, `var 2` (mau escolha), `var2` (boa escolha), `var_2` (boa escolha);
- devemos evitar “.” (ponto) em palavras compostas, pois esta sintaxe é responsável pela criação de um método para uma determinada classe no sistema `S3` (um dos paradigmas de orientação a objetos no **R**), e isto pode causar possíveis conflitos na interpretação dos dados.

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 seus códigos internos.

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 R:

```
> # Criando e inserido os elementos do objeto dados
> x <- scan()
1:
```

Após executado a 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 programas externos ao **R** para criação de banco de dados e deixá-lo pronto para ser lido em **R**. O tipo de arquivo de texto que melhor controla a separação de variáveis é com a extensão “`<>.csv`”, uma vez que separamos as colunas por “;”, quase sempre o padrão, ou qualquer outro tipo de símbolo. O arquivo de texto com extensão “`<>.txt`”, geralmente usa espaços o que acaba gerando pro-

blema de leitura no **R**, pois 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 delimitada 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* e *Libre Office*, por exemplo. Eles exportam arquivos binários do tipo “<>.xls”, “<>.xlsx”, dentre outras opções. Uma sugestão para diminuir complicações, é exportar os bancos de dados para arquivos de texto citados acima (<>.csv ou <>.txt). Isso evita a instalação de mais pacotes no ambiente **R**, já que os pacotes nativos apresentam funções para a leitura dos arquivos mencionados.

Para a leitura dos arquivos binários, sugerimos a instalação do pacote **readr**, sabendo que existem diversos outros pacotes de mesmo objetivo.

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

Posteriormente, devemos indicar o arquivo para leitura e algumas opções de tipo de arquivo são apresentadas. Usaremos a opção *From Text (base)*, isto é, realizar a leitura para os tipos de arquivo <>.txt ou <>.csv, e 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 visualizarmos algum problema, devemos informar as opções adicionais, como: separador de variáveis (*Separator*), símbolo para casas decimais (*Decimal*), dentre outras opções. Por fim, digitamos o nome associado ao objeto (*Name*) que será criado do tipo quadro de dados (*data frame*), e clicamos no botão *Import*, Figura 5.4;

Depois de importado o banco de dados, 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.

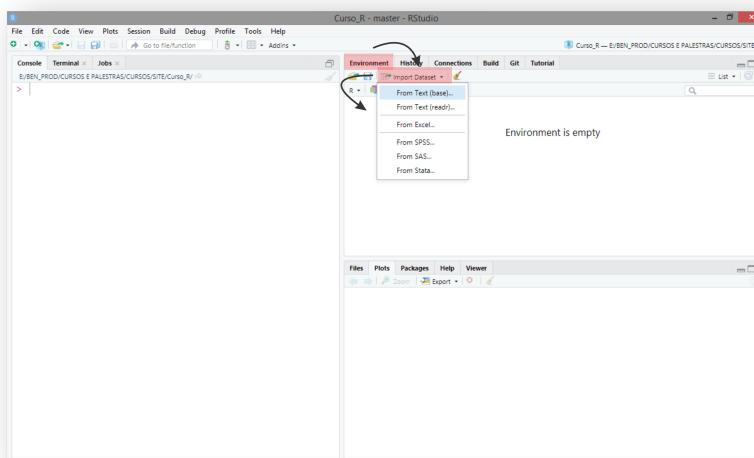


Figura 5.2: Usando o RStudio para importar dados.

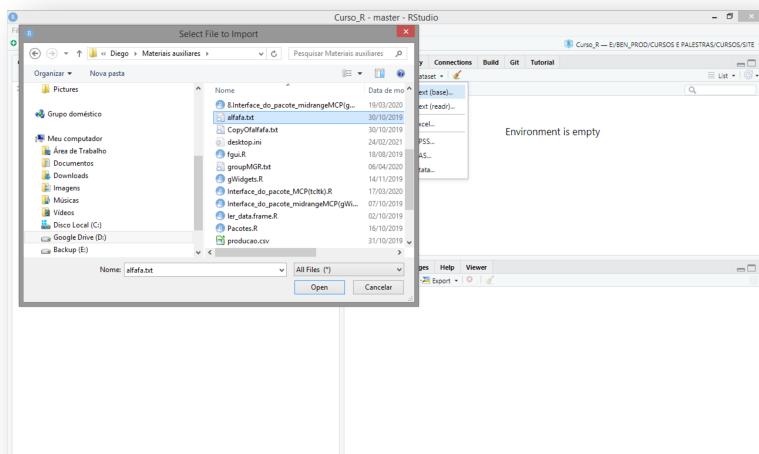


Figura 5.3: Usando o RStudio para importar dados.

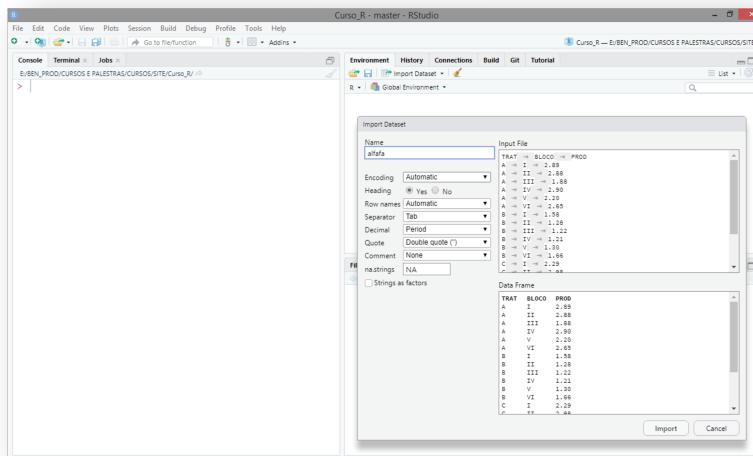


Figura 5.4: Usando o **RStudio** para importar dados.

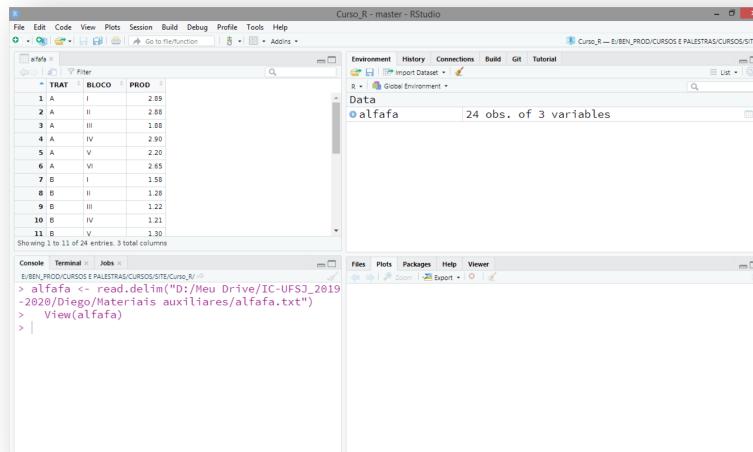


Figura 5.5: Usando o **RStudio** para importar dados.

A outra forma de importar os dados é usar diretamente as linhas de comando com o auxílio da função `read.table()`. Antes de importarmos o banco de dados, é interessante inserir o arquivo de dados no diretório de trabalho no ambiente **R**. Para verificar o diretório de trabalho usamos a função `getwd()`. Para alterar o local do diretório de trabalho usamos `setwd()`. Se este procedimento não for realizado, devemos informar na função `read.table()`, o local exato do arquivo.

Vamos usar como diretório o local `C:\eambr01`, como exemplo. Lembramos que para identificar o caminho do diretório no **R**, o separador de subdiretórios é representado pela barra invertida, **a barra deve ser invertida**. Usaremos três arquivos: `alfafa.txt`, `datast1980.txt` e `producao.csv`, que estão organizados da seguinte forma:

- `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`²

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

²Dados retirados de Steel e Torrie (1980).

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

Verificamos que os três bancos de dados apresentam estruturas de organização de dados diferentes. Os dois primeiros, as variáveis são separadas por espaço, enquanto que o último é separado por “;”. Percebemos também que o separador de casas decimais para alfafa.txt e producao.csv são o ponto, enquanto que o arquivo datast1980.txt,

as decimais são separadas por vírgula. Percebemos que além das extensões dos arquivos, as informações mencionadas anteriormente, são importantes para o sucesso da leitura dos bancos de dados, sendo que as informações são repassadas para as funções responsáveis pela importação no **R**.

Iremos apresentar a forma de como importarmos banco de dados por meio da função `read.table()`. Vejamos as linhas de comando para importar os dados, Código R 5.1. Antecipadamente, vale a pena mencionar alguns argumentos importantes dessa função:

- `file`: banco de dados;
- `header`: argumento lógico, se `header = TRUE`, então implica dizer que as variáveis estão identificadas por um nome, isto é, a primeira linha do banco de dados representa o nome das variáveis; se `header = FALSE` (padrão), caso contrário;
- `sep`: separador de variáveis nas colunas, o padrão é `sep = ""`, isto é, sem espaços; no caso de arquivos de extensão: `<>.csv`, geralmente, usamos para este argumento `sep = ";"`, identificando que a separação das variáveis está representada por `";"`;
- `dec`: separador de casas decimais, sendo o padrão `dec = ".."`.

Apesar de existirem mais argumentos, esses apresentados anteriormente são os quatro argumentos necessários para a importação básica de um banco de dados usando a função `read.table()`. Na última linha de comando do Código 5.1, mostramos que também é possível importar dados de arquivos de texto da *web*, considerando que o usuário está com acesso à *internet* no momento da importação. Um recurso interessante que pode ser usado, principalmente para o último caso, é salvar o banco de dados em um arquivo de dados no arquivo `".RData"`. Assim, todos os dados, inclusive os importados da *internet*, serão armazenados de modo que não precisaremos mais de acesso à *web* para manipulá-los.

Percebemos que as extensões `"<>.txt"` e `"<>.csv"` são idênticas, exceto pela estrutura de como os dados estão dispostos. Para isso, podemos mudar a extensão de um arquivo do tipo `"<>.csv"` para `"<>.txt"` e observarmos em um bloco de notas a diferença entre as duas estruturas de arquivo.

Argumentos em funções



Até agora, usamos as funções no **R** em algumas situações, sem identificar o nome de seus argumentos dentro do parêntese. Quando inserimos os valores dos argumentos na posição correta, não inserimos os seus nomes. 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, `x`, que representa o conjunto de dados. Assim, como sabemos que `x` é o primeiro argumento da função, omitimos o nome e calculamos a média usando uma forma simplificada, pela chamada `mean(valores)`, que é o mesmo que `mean(x = valores)`. Para mais detalhes, usamos a chamada `?mean()`. Reforçamos que as atribuições de valores feitas nos argumentos em uma função é realizada por “=” e não por “<-”. Para verificar essa diferença, temos a sugestão do Exercício 4.23 para mais detalhes.

5.4 Exportando dados

Começamos inicialmente sobre a exportação para arquivos de dados do **R** (`<>.RData`), comentado na seção anterior. Para salvarmos um conjuntos de dados em um arquivo de dados do **R**, usamos a função `save()`. Para carregar e armazená-los no ambiente global, usamos a função `load()`, Código R 5.2. Devemos observar também que ao carregar o arquivo de extensão “.RData” pela função `load()`, os nomes em “.RData” sobrescreverão os nomes no ambiente global se forem idênticos. Vale lembrar que ao finalizarmos um determinado *script*, todos os objetos nominados, bem como as linhas de comando, poderão ser salvas em “.RData” e “.Rhistory”, respectivamente, Figura 3.6. Na realidade, por trás dessa janela o que ocorre são execuções de comandos como: `q()` para finalizar o **R**, `save.image()`³ e `savehistory()` que criam os arquivos .RData e .Rhistory, respectivamente, no diretório de tra-

³`save.image()` é o mesmo que `save(list = ls(all.names = TRUE), file = ".RData", envir = .GlobalEnv)`.

balho. Para carregarmos arquivo de extensão “.Rhistory”, usamos a chamada de função `loadhistory()`.

Código R 5.1

Script R:

```
1 # Diretorio
2 getwd()
3
4 # Mudando para o diretorio de interesse
5 setwd("C:/eambr01")
6
7 # Verificando os arquivos no diretorio de trabalho
8 list.files()
9
10 # Importando os dados para o diretorio do arquivo
11 dados1 <- read.table(file = "C:/eambr01/alfafa.txt",
12     header = TRUE)
13
14 # Considerando que o arquivo esta no diretorio de
15 # trabalho, isto eh, getwd()
16 dados2 <- read.table("alfafa.txt", header = TRUE)
17
18 # Importando os dados com decimais com "," para o
19 # diretorio onde se encontra o arquivo
20 dados3 <- read.table(
21     file = "C:/eambr01/dadost1980.txt",
22     header = TRUE, dec = ","
23 )
24
25 # Considerando que o arquivo esta no diretorio de
26 # trabalho, isto eh, getwd()
27 dados4 <- read.table(
28     file = "dadost1980.txt",
29     header = TRUE, dec = ","
30 )
```

Script R:

```

31 # Importando os dados com decimais ",", e separados
32 # por ";" apontando para o diretório do arquivo
33 dados5 <- read.table(
34   file = "C:/eambr01/producao.csv",
35   header = TRUE,
36   dec = ",",
37   sep = ";"
38 )
39
40 # Considerando que o arquivo está no diretório de
41 # trabalho, isto é, getwd()
42 dados6 <- read.table(
43   file = "producao.csv",
44   header = TRUE,
45   dec = ",",
46   sep = ";"
47 )
48
49 # Importando da internet
50 dados7 <- read.table(
51   file = "https://raw.githubusercontent.com/
52     bendeivide/book-eambr01/main/files/alfafa.txt",
53   header = TRUE
54 )

```

Posteriormente, ao abrirmos um projeto que contenha os arquivos, tanto a área de trabalho armazenada em .RData quanto o histórico das linhas de comando em .Rhistory, serão carregados e disponíveis para a área de trabalho atual. Para salvarmos apenas um objeto, usamos o arquivo de dados R de extensão "<>.rds", Código R 5.3. Já na Tabela 5.4, observamos funções para criar e carregar os arquivos "<>.RData" e "<>.rds".

A exportação de dados para extensões de arquivos do tipo "<>.txt" ou "<>.csv", é realizada pela função `write.table()`, que tem como al-

guns dos argumentos:

- **x**: objeto a ser exportado, preferencialmente estruturas de dados do tipo quadro de dados ou matriz;
- **file**: nome do arquivo a ser exportado, inserindo também a sua extensão; se **file** = "**~**", a saída será impressa no *console*;
- **dec**: separador de decimais;
- **sep**: separador de colunas;
- **row.names**: se um valor lógico (TRUE OU FALSE), indicará se haverá ou não nome nas linhas; se um vetor de caracteres, representará o nomes das linhas a serem escritas;
- **col.names**: se um valor lógico (TRUE OU FALSE), indicará se haverá ou não nome nas colunas; se um vetor de caracteres, representará os nomes das colunas a serem escritas;
- **quote**: um valor lógico (TRUE ou FALSE). Se TRUE, os caracteres serão impressos entre aspas. Se FALSE, nada é feito.

Existem outras funções que podem ser exploradas: `write()`, `write.csv()`, `write.csv2()`, `write.ftable()`, etc.. Deixaremos como exercício a utilização dessas funções. Para verificarmos em detalhes a função `write.table()`, vejamos o Código R 5.4.

Tabela 5.4: Funções para .RData, .rda e .rds

Finalidade	.RData ou .rda	.rds
Criação do arquivo	<code>save()</code>	<code>saveRDS()</code>
Ler o arquivo	<code>load()</code>	<code>readRDS()</code>

Qual a diferença nos arquivos <>.RData, <>.rda e <>.rds?



Os três tipos de extensão para arquivo de dados no **R** são: <>.RData, <>.rda e <>.rds. As extensões <>.RData e <>.rda representam o mesmo tipo de arquivo. Com eles, o usuário consegue armazenar mais de um tipo de objeto. Já o arquivo com a extensão <>.rds, armazena apenas um tipo de objeto. O objeto quando armazenado é serializado^a. Os arquivos <>.rda e <>.rds são utilizados para armazenar metadados de pacotes e bancos de dados no ambiente **R**. Na Tabela 5.4, apresentamos um resumo de funções utilizadas para os três tipos de arquivo.

^aA serialização é o procedimento que transforma a estrutura de dados do objeto em uma cadeia de *bytes*, de modo a ser manipulado, armazenado e transferido, mais facilmente pelas máquinas. Após ter sido armazenado, a volta do estado normal do objeto é a desserialização, e o objeto pode ser manipulado novamente em **R**.

Código R 5.2

Script R:

```

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

Código R 5.3**Script R:**

```

1 # Criando um arquivo temporario trees.rds
2 dados <- tempfile("trees", fileext = ".rds")
3 ## Salvando um unico objeto em dados
4 saveRDS(trees, dados)
5 ## Recarregando dados em um novo objeto
6 trees2 <- readRDS(dados)
7 identical(trees, trees2)

```

Console R:

```
[1] TRUE
```

Script R:

```

8 # ou examinar via conexao, que sera aberto quando
9 # necessario
10 con <- gzfile(dados)
11 # Imprimindo as primeiras linhas
12 head(readRDS(con))

```

Console R:

	Girth	Height	Volume
1	8.3	70	10.3
2	8.6	65	10.3
3	8.8	63	10.2

Script R:

```
11 close(con)
```

Código R 5.4**Script R:**

```

1 # Arquivo temporario
2 x <- tempfile(pattern = "dados", fileext = ".txt")
3 # Objeto
4 d <- data.frame(x = 1, y = 1:10)
5 # Exportando os dados
6 write.table(x = d, file = x, sep = "\t",
7   quote = FALSE, row.names = FALSE)
8 # sep = "\t": representa uma TAB entre as colunas
9 # file = x: arquivo de saída
10 file.show(x) # Abrindo o arquivo

```

Após verificarmos como exportar dados para um arquivo “<>.txt”, de forma análoga, também exportamos para um arquivo de extensão “<>.csv”. Para isso, alteramos os argumentos `file = "C:/dados .csv"` e `sep = ";"`, em que geralmente as colunas são separadas por ponto e vírgula (“;”), mas não sendo uma regra geral. É interessante verificar o argumento `dec`, usado para escolher o símbolo que separa as casas decimais dos valores numéricos; o padrão é separar as decimais por ponto.

Para arquivos binários do tipo “<>.xlsx” e similares, usamos alguns pacotes `readxl`, `openxlsx`, `xlsx`, `writexl`, etc.. Baseado neste último, iremos apresentar um exemplo de exportação no Código R 5.5.

Código R 5.5**Script R:**

```

1 # Instalando e/ou carregando o pacote "readxl"
2 # install.packages("readxl") # Para instalar!
3 library(readxl)
4 # Vendo exemplos de arquivo no pacote
5 readxl_example()

```

Console R:

```
[1] "clippy.xls"     "clippy.xlsx"    "datasets.xls"
[4] "datasets.xlsx" "deaths.xls"     "deaths.xlsx"
[7] "geometry.xls"   "geometry.xlsx"  "type-me.xls"
[10] "type-me.xlsx"
```

Script R:

```
6 # Caminho do arquivo
7 arquivo_xlsx <- readxl_example("datasets.xlsx")
8 # Estrutura de dado "tibble"
9 dado <- read_excel(arquivo_xlsx); dado
```

Console R:

```
# A tibble: 150 x 5
  S.Length S.Width P.Length P.Width Species
    <dbl>    <dbl>     <dbl>    <dbl> <chr>
1      5.1     3.5      1.4     0.2  setosa
2      4.9      3        1.4     0.2  setosa
3      4.7     3.2      1.3     0.2  setosa
4      4.6     3.1      1.5     0.2  setosa
5       5       3.6      1.4     0.2  setosa
# ... with 145 more rows
```

Script R:

```
10 # Estrutura de dado "quadro de dado"
11 head(as.data.frame(dado))
```

Console R:

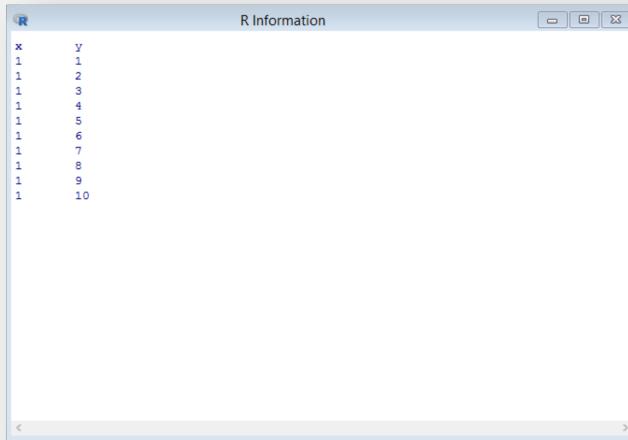
```
S.Length S.Width P.Length P.Width Species
1      5.1     3.5      1.4     .2  setosa
2      4.9     3.0      1.4     0.2  setosa
3      4.7     3.2      1.3     0.2  setosa
```

Script R:

```

12 # Arquivo temporario
13 x <- tempfile(pattern = "dados", fileext = ".txt")
14 d <- data.frame(x = 1, y = 1:10) # Objeto
15 # Exportando os dados
16 write.table(d, file = x, sep = "\t", quote = FALSE,
17   row.names = FALSE)
18 # sep = "\t": representa um TAB entre as colunas
19 # file = x: arquivo saída
20 file.show(x) # Abrindo o arquivo

```



Pelo Código R 5.5, importamos um conjunto de dados para uma estrutura de dados chamada *tibble*, não comentada até agora. Mas, entendemos que é uma estrutura similar ao quadro de dados, com mais informações sobre os seus valores, e que é a estrutura de dados mais utilizada nos pacotes da família de pacotes **tidyverse**. Verificamos que a coerção de *tibble* para um quadro de dados é realizada facilmente usando a função `as.data.frame()`, Código R 5.5. Agora, iremos exportar um conjunto de dados para um arquivo binário, usando o pacote

writexl, Código R 5.6.

Código R 5.6

Script R:

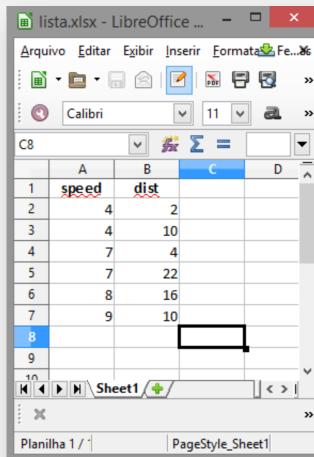
```
1 # Dados do pacote nativo "datasets"
2 library(datasets)
3 dados <- head(cars); dados # Quadro de dados
```

Console R:

	speed	dist
1	4	2
2	4	10
3	7	4
4	7	22
5	8	16
6	9	10

Script R:

```
3 # Instalando e/ou carregando writexl
4 ## install.packages("writexl")
5 library(writexl)
6 # Diretório de trabalho
7 dirtrab <- getwd()
8 # Nome do arquivo
9 nome <- "dados.xlsx"
10 # Caminho
11 caminho <- paste(dirtrab, nome, sep = "/")
12 # Exportando os dados para um ".xlsx"
13 file <- writexl::write_xlsx(dados, caminho)
14 # Mortrando o arquivo
15 file.show(file)
```



	A	B	C	D
1	speed	dist		
2	4	2		
3	4	10		
4	7	4		
5	7	22		
6	8	16		
7	9	10		
8				
9				

Script R:

```
16 # Vamos criar agora listas de quadro de dados
17 d1 <- head(cars); d1 # valores iniciais de "cars"
18 d2 <- tail(cars); d2 # valores finais de "cars"
```

Console R:

```
> d1 <- head(cars); d1 # valores iniciais de "cars"
   speed dist
 1      4    2
 2      4   10
 3      7    4
> d2 <- tail(cars); d2 # valores finais de "cars"
   speed dist
48     24   93
49     24  120
50     25   85
```

Script R:

```
19 lista <- list(elem01 = d1, elem02 = d2); lista
```

Console R:

```
$elem01
  speed dist
1      4     2
2      4    10
3      7     4

$elem02
speed dist
48     24    93
49     24   120
50     25    85
```

Script R:

```
20 # Diretorio de trabalho
21 dirtrab <- getwd()
22 # Nome do arquivo
23 nome <- "lista.xlsx"
24 # Caminho
25 caminho <- paste(dirtrab, nome, sep = "/")
26 # Exportando os dados para um ".xlsx"
27 file <- writexl::write_xlsx(lista, caminho)
28 # Mortrando o arquivo
29 file.show(file)
```

	A	B
1	speed	dist
2	4	2
3	4	10
4	7	4
5	7	22
6	8	16
7	9	10
8		
9		
10		

	A	B
1	speed	dist
23	54	
24	70	
24	92	
24	93	
24	120	
25	85	

Para os usuários de L^AT_EX, um pacote interessante é o **xtable**. Conseguimos exportar no *console*, o código em L^AT_EX, como por exemplo um quadro de dados. Vejamos Código 5.7.

Código R 5.7

Script R:

```

1 # Instalando e/ou carregando o pacote "xtable"
2 ## install.packages("xtable")
3 library(xtable)
4 library(datasets)
5 # Dados
6 dados <- head(mtcars[,1:3]); dados

```

Console R:

	mpg	cyl	disp
Mazda RX4	21.0	6	160
Mazda RX4 Wag	21.0	6	160
Datsun 710	22.8	4	108
Hornet 4 Drive	21.4	6	258
Hornet Sportabout	18.7	8	360
Valiant	18.1	6	225

Script R:

```
5 # Exportando para LaTeX
6 xtable(dados)
```

Console R:

```
% latex table generated in R 4.1.3 by xtable 1.8-4
  package
% Wed Jun 22 14:32:31 2022
\begin{table}[ht]
  \centering
  \begin{tabular}{rrrr}
    \hline
    & mpg & cyl & disp \\
    \hline
    Mazda RX4 & 21.00 & 6.00 & 160.00 \\
    Mazda RX4 Wag & 21.00 & 6.00 & 160.00 \\
    Datsun 710 & 22.80 & 4.00 & 108.00 \\
    Hornet 4 Drive & 21.40 & 6.00 & 258.00 \\
    Hornet Sportabout & 18.70 & 8.00 & 360.00 \\
    Valiant & 18.10 & 6.00 & 225.00 \\
    \hline
  \end{tabular}
\end{table}
\index{Latex\LaTeX}
```

A renderização em \LaTeX ficou assim:

	mpg	cyl	disp
Mazda RX4	21.00	6.00	160.00
Mazda RX4 Wag	21.00	6.00	160.00
Datsun 710	22.80	4.00	108.00
Hornet 4 Drive	21.40	6.00	258.00
Hornet Sportabout	18.70	8.00	360.00
Valiant	18.10	6.00	225.00

Portanto, devemos lembrar que o **R** hoje está muito dinâmico, e os recursos de importação e exportação são diversos. Por exemplo, um pacote nativo **foreign** permite a importação de dados do Minitab, S, SAS, SPSS, Stat, Systat, dBase, etc.. Em relação a exportação, tivemos inicialmente uma ferramenta chamada *Sweave*, hoje uma função do pacote nativo **utils**, do qual conseguimos gerar um documento em *PDF*⁴, que resulta da execução de comandos em **R** e **LATEX** em um mesmo arquivo, isto é, desenvolvemos relatórios de nossas análises cuja saída exporta códigos, gráficos, resultado de funções, etc., tudo compilado em *PDF*. Para mais detalhes em: <https://stat.ethz.ch/R-manual/R-devel/library/utils/doc/Sweave.pdf>, para obter mais informações.

Com o surgimento do pacote **rmarkdown**, **knitr**, dentre outros, podemos agora exportar análises, relatórios, apresentações de dados e muitos outros formatos, para arquivos de diversas extensões: <>.pdf, <>.html, <>.epub, <>.tex, etc.. Usamos o pacote **shiny** como uma interface para as linguagens HTML, JavaScript, CSS, dentre outras, e com isso, desenvolvemos arquivos dinâmicos em HTML para a criação de elementos gráficos como: botões, menus, caixas de texto, e muitas outras opções. Devido a complexidade de utilização dos pacotes, abordaremos em *Volumes* específicos da coleção *Estudando o Ambiente R*.

5.5 Exercícios

Exercício 5.1:

De acordo com o Código R 5.1, verifique cada argumento da função `read.table()`, e posteriormente, imprima os objetos para observar a estrutura de dados como um quadro de dados.

⁴Do inglês, *Portable Document Format*

Faça estudos, alterando o valor dos argumentos para verificar o que ocorre com a estrutura de dados.

Dica na página 293

Exercício 5.2:

Use o Código R 5.4, e faça as adaptações para exportar um arquivo de extensão “<>.csv”. Ao exportar, faça a leitura em **R**, para verificar se a exportação é passível do **R** ler o conjunto de dados novamente.

Dica na página 293

Exercício 5.3:

Baseado na tabela abaixo, crie um quadro de dados, exporte-o para um arquivo “<>.xlsx”. Posteriormente, tente ler o conjunto de dados novamente para importá-lo para o **R**.

#	speed	dist
1	4.00	2.00
2	4.00	10.00
3	7.00	4.00
4	7.00	22.00
5	8.00	16.00
6	9.00	10.00
7	10.00	18.00
8	10.00	26.00
9	10.00	34.00
10	11.00	17.00
11	11.00	28.00
12	12.00	14.00
13	12.00	20.00
14	12.00	24.00
15	12.00	28.00

Dica na página 293

Exercício 5.4: Baseado no Exercício 5.3, use o pacote **openxlsx**, para a mesma finalidade. Faça o mesmo usando o pacote **xlsx**

Dica na página 293

Exercício 5.5: Pesquise mais sobre os pacotes **rmarkdown** e **shiny**, bem como a ferramenta *Sweave*, para verificar os recursos de exportação de arquivos em forma de documentos, podem ser obtidos. Isso poderá proporcionar uma maior autonomia na atividades desenvolvidas usando o **R**, no sentido de obter documentações elegantes das análises desenvolvidas. **OBS.:** Se a exportação desejada for na extensão <>.pdf será necessário instalar o **LATEX**. Uma alternativa para os usuários que não tiverem ou não desejarem instalar o **LATEX**, exceto para ferramenta *Sweave*, é instalar o pacote **tinytex** pelo próprio **R**.

Dica na página 293

Exercício 5.6: Apesar de desafiante, não é impossível a criação de interfaces gráficas ao usuário no **R**. Assim, recomendamos uma pesquisa e estudo sobre os pacotes **shiny** e **tcltk** para o desenvolvimento de uma interface que seja possível importar e exportar um conjunto de dados. O desenho da interface é de livre escolha.

Dica na página 294

Exercício 5.7: Procure por arquivos de dados de alguns programas como: Minitab, S, SAS, SPSS, Stat e Systat, e realize a importação dos dados por meio do pacote **foreign**.

Dica na página 294

Exercício 5.8: Verifique a diferença entre as funções: `write()`, `write.csv()`, `write.csv2()` e `write.ftable()`, e apresente exemplos de aplicações.

Dica na página 294

Capítulo 6

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 (`<-`) temos uma chamada de função dada por: ``<-`(x, 10)`. Quando digitamos `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. Percebemos que situações básicas até mais complexas, temos por trás uma chamada de função.

De outro modo, entendamos que uma função no **R** é um objeto do tipo “closure”, “special” ou “builtin”. Assim como qualquer outro objeto, as funções também contém seus atributos e características específicas. As funções criadas a nível de usuário, serão do tipo “closure”. Para entendermos um pouco mais sobre esse tipo de objeto e sua importância, convidamos a leitura das próximas seções do capítulo.

6.2 O que é uma função no R?

Os princípios afirmados por Chambers (2016) são interligados, principalmente os dois primeiros, porque como falado anteriormente, apesar de tudo que acontece no **R** ser uma chamada de função, a função é um objeto, com estrutura definida como qualquer outro. Por exemplo, será que a função `apply()` apresenta atributos intrínsecos? Vejamos nas linhas de comando que segue:

Console R:

```
> mode(apply) # Modo
[1] "function"
> length(apply) # Comprimento
[1] 1
```

Ainda mais, dizemos que a linguagem **R** tem um estilo funcional, e devido a este fato, dentre outros, um aprofundamento será abordado no *Volume II*. As funções podem ser divididas como três tipos de objetos que podem ser apresentados na Tabela 6.1.

Tabela 6.1: Tipo de funções.

Tipo de função	typeof()	mode()
Criada por usuário	closure	function
Função interna que não avalia seus argumentos	special	function
Função interna que avalia seus argumentos	builtin	function

As funções internas usam as chamadas `.Internal()` e `.Primitive()`, que apresentam algumas diferenças, mas não serão detalhadas para este momento. O interessante é saber que essas funções representam uma interface para as linguagens de baixo nível, do qual as funções foram implementadas. Para mais detalhes, podemos consultar os manuais de ajuda com as chamadas `?Internal()` e `?Primitive()`.

Sem descrever tantos detalhes sobre a diferença desses tipos de objetos, vejamos o Código 6.1 para uma abordagem de como identificá-los. A diferença que existente entre os tipos de funções estão relacionados na forma como seus argumentos são avaliados. Lembrando que funções internas só podem ser desenvolvidas pelo *R Core Team*. Para saber se um determinado objeto é uma função, usamos a função `is.function()`.

Código R 6.1**Console R:**

```
> typeof(list); typeof(mean); typeof(`[`)
[1] "builtin"
[1] "closure"
[1] "special"
> mode(list); mode(mean); mode(`[`)
[1] "function"
[1] "function"
[1] "function"
```

A ideia de função no ambiente **R** não é pensada como uma relação matemática, mas como um sistema que tem uma entrada e saída de informações. Existem funções no ambiente **R** que organizam dados, dentre outras ações, que não necessariamente realizam operações matemáticas, por exemplo. Vejamos a função `sort()` do pacote **base**, que ordena de forma crescente ou decrescente um conjunto de valores, como pode ser observado no Código R 6.2.

As funções do tipo “closure” (funções criadas por meio da função `function()`, apresentam três estruturas básicas: argumento, corpo e ambiente, que serão abordadas em detalhes 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 internas, escritas na linguagem C, essa regra foge à exceção. Como já mencionado, as funções internas são funções primitivas de tipo “builtin” ou “special”, e para verificarmos a tipagem desses objetos, devemos sempre usar `typeof()` ao invés de `mode()`.

Código R 6.2

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

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

Script R:

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

Console R:

```
{
  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 R:

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

Console R:

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

Observando o Código R 6.2, os argumentos `x`, `decreascing` e `...`, são nomes que aguardam receber objetos para a execução da função `sort()`. Nem todos os argumentos necessitam receber objetos, os chamamos de argumentos padrão, como o caso do argumento `decreascing` com padrão igual a `FALSE`, que significa que o ordenamento dos dados serão de forma não-decrescente. Observamos ainda na função `sort()`, que entramos apenas com o argumento `x = y`, não precisando inserir `decreascing = FALSE`, porque este é um argumento padrão. Agora, para modificarmos o argumento padrão, acrescentamos a alteração da seguinte forma:

Console R:

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

O objeto reticências ou três pontos (“`...`”) é um argumento especial, geralmente utilizado em uma função quando não se sabe o número exato de argumentos. Sua representação na tipagem interna em C é “`DOTSXP`”, sendo que mais detalhes serão observados ainda neste

capítulo.

O próximo item é o corpo da função, um componente onde inserimos 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 digitando no *console* apenas o nome sem o parêntese, por exemplo, `sort` para o caso anterior.

Por fim, último componente é o ambiente, o objeto que armazena a ligação entre o nome e a respectiva função. Como exemplo, temos a função `sort()` que apresenta o ambiente do pacote **base** como o componente ambiente.

6.4 Funções em pacotes

Observamos que as funções utilizadas até agora, não foram criadas pelo usuário, mas originadas de pacotes.

O que é um pacote?



Entenda por pacote um diretório contendo subdiretórios e arquivos específicos. Em um dos subdiretórios, temos o local onde armazenamos as funções criadas. Ao instalar e anexar o pacote no **R**, todas as funções tornam-se disponíveis no ambiente global. Os pacotes nativos padrão sempre estão disponíveis para execução. Falaremos sobre os demais subdiretórios e arquivos específicos no Capítulo 8.

Alguns pacotes estão disponíveis quando instalamos o **R**, dizemos que estes são os pacotes nativos do **R** (pacotes padrão e recomendados). O principal pacote deles é o **base**. Os demais pacotes desenvolvidos podem ser obtidos via *CRAN*, sendo assunto abordado mais à frente.

O ambiente **R** apresenta uma versatilidade de manuais para a linguagem. Por exemplo, para verificar informações sobre um determinado pacote como o **base**, usamos `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 determinada parte de nome é `apropos()`, por exemplo, para o exemplo anterior temos `apropos("sort")`. O pacote **base** sempre estará anexado, isto é, disponível ao caminho de busca para a utilização, porque é um pacote nativo padrão. 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 a função `read.dbf()` do pacote **foreign**, um dos pacotes nativos do **R**, que não está anexado¹ ao inicializar o **R**, porque é um pacote nativo recomendado e não padrão. Assim, para anexá-lo usamos a chamada `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*, usamos a dupla interrogação "???" 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, recomendamos a leitura do Capítulo 8.

6.5 Chamadas de funções

As chamadas de funções podem ocorrer de três formas: aninhada, intermediária ou a do tipo *pipe*. Todas essas formas estão implementadas no **R** de forma nativa, porém a do tipo *pipe*, foi implementada 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:

- Aninhada:

Console R:

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

- Intermediária:

¹Anexar um pacote ao caminho de busca pressupõe que este esteja instalado no seu sistema operacional.

Console R:

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

• *Pipe:***Console R:**

```
> # 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 uma 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, que é uma função.

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 e visualizações gráficas, sem necessariamente precisarmos associar nomes aos objetos, de modo a armazenar seus resultados.

6.6 Estruturas de controle ou controle de fluxos

Quando desejamos realizar processos repetitivos ou condicionamento a determinadas ações, usamos as estruturas de controle. Assim, como

em outras linguagens, as funções mais comuns são: `if()`, `switch()`, `ifelse()`, `while()`, `repeat` e `for()`. Todos os nomes dessas funções são palavras reservadas no ambiente **R**, isto é, não podemos associar estes nomes a objetos. As estruturas de controle são do tipo “special”. Por exemplo, para a função `for()`, obtemos o seu tipo por `typeof(`for`)`. Apenas a função `ifelse()` é do tipo “closure”, e o nome não é uma palavra reservada, sendo uma criada por `function()`, o mesmo tipo de objeto quando nós criarmos as nossas funções.

A sintaxe das estruturas de controle de tipo “special” é:

Script R:

```
1 # Primeira sintaxe
2 função (condição) {
3   expressão
4 }
5 # Segunda sintaxe
6 função (condição) expressão
```

Para o caso das funções `repeat` e `switch()`, a sintaxe foge um pouco do padrão mencionado anteriormente, sendo explicadas mais à frente. Apesar da linguagem **R** ser interpretada, há discussões quanto ao seu desempenho em relação as funções *loops*, pois quando comparada com outras linguagens, se afirma que o **R** tem baixo desempenho nesse aspecto. Em alguns casos, a construção do algoritmo proporciona isso, e não a implementação das funções em si. Um exemplo que citamos são as cópias de objetos que podem ter um gasto de memória virtual imenso no processo e proporcionar um gasto computacional. Isso foi devido a forma de como o algoritmo copiou os objetos, e não ao desempenho das funções *loops* na linguagem.

A primeira estrutura é a função `if()`, apresentada no Código R 6.3, linhas 1-4, com sintaxe simplificada apresentada na linha 5. O fluxograma dessa função pode ser observado na Figura 6.1.

Código R 6.3

Script R:

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

Script R:

```
5 if (condição) instruções sob condição = TRUE
```

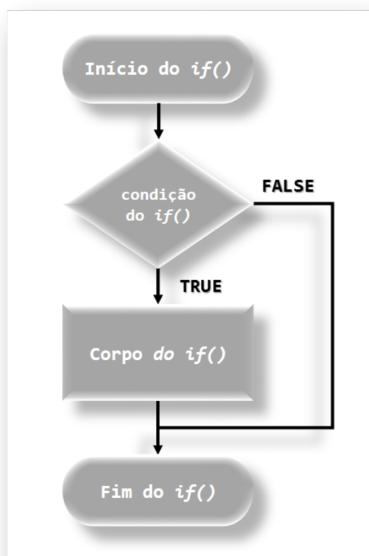


Figura 6.1: Fluxograma da função if().

Para elucidarmos o primeiro caso, observamos o Código R 6.4. Após associado o nome *x* com o objeto 5, isto é *i* <- 5, a função if() foi chamada e a condição verificada se *i* > 3. Como essa condição foi

verdadeira, o resultado é impresso, encerrando assim a execução.

Código R 6.4

Script R:

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

Console R:

```
[1] "Maior que 3!"
```

Juntamente com o `if()`, utilizamos a função `else`, cuja forma sintática é apresentada no Código R 6.5. O fluxograma da combinação dessas duas funções podem ser representados pela Figura 6.2.

Código R 6.5

Script R:

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

Script R:

```
8   if (condição) instr1 else instr2
```

Uma forma mais simplificada unindo as funções `if()` e `else` pode ser encontrada ainda no Código R 6.5, *linha 8*. Porém, uma forma

mais eficiente dessa junção resultou na criação da função `ifelse()`, que pode ser representada sintaticamente no Código R 6.8. Vejamos mais um exemplo no Código R 6.6.

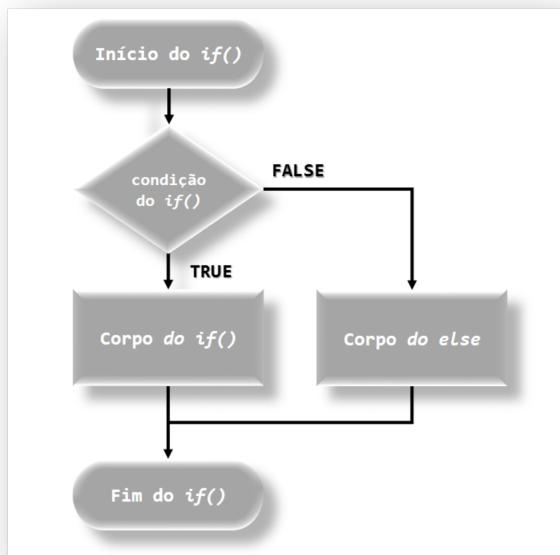


Figura 6.2: Fluxograma da função `if()` e `else`.

No Código R 6.6, observamos na primeira forma que a condição para `if()` avaliar, não é necessário perguntar se `is.numeric(x) == TRUE`, porque é implícito e esta condição leva a primeira expressão `print("Isto 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 o primeiro item seria avaliado na versão **R** (4.1 ou inferior) com uma alerta de mensagem. Já na versão do **R** (4.2), essa mesma situação retorna um erro. Para entender, vejamos o Código R 6.7.

Código R 6.6**Script R:**

```

1 x <- 10 # Objeto numerico
2 # Estrutura "if"
3 if (is.numeric(x)) {
4   print("Isto é um número")
5 } else {
6   print("Isto não é um número")
7 }
```

Console R:

```
[1] "Isto é um número"
```

Script R:

```

9 # eh o mesmo que
10 if (is.numeric(x) == TRUE) {
11   print("Isto é um número")
12 } else {
13   print("Isto não é um número")
14 }
```

Console R:

```
[1] "Isto é um número"
```

O fato da condição `if()` não ser vetorizada para `w < x`, Código R 6.7, observamos um erro ao final do resultado. Isso implica que a função `if()` deve ser executada entre vetores escalares, como observamos no Código 6.7, linha 13. Neste último caso, observamos que o resultado é impresso no *console* sem mensagem de erro. Uma saída para avaliar cada elemento de um vetor a uma determinada condição, seria utilizar a função `for()` conjugada com `if()` e `else`, por exemplo.

Código R 6.7

Script R:

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

Console R:

```
Error in if (w < x) { : a condição tem comprimento > 1
```

Script R:

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

Console R:

```
Error in if (w < x) { : a condição tem comprimento > 1
```

Script R:

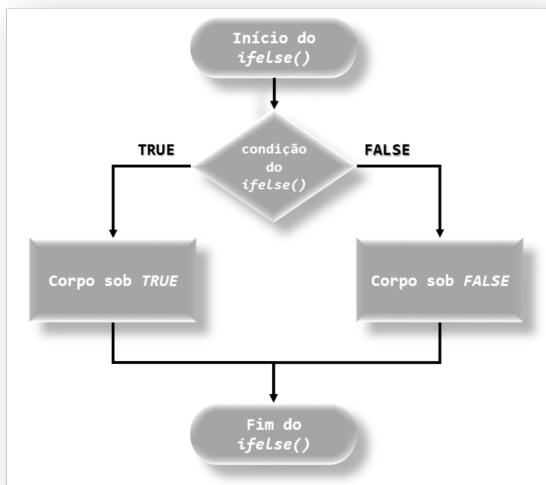
```
12 # Forma correta de aplicar if()
13 x <- 5 ; w <- 3 # escalar
14 if (w < x) x else w
```

Console R:

```
[1] 5
```

Como forma de vetorização da função `if()`, foi criada a função `ifelse()`, em que o fluxograma dessa função pode ser observado pela

Figura 6.3.

Figura 6.3: Fluxograma da função `ifelse()`.**Código R 6.8****Script R:**

```

1 ifelse(test = condição,
2       yes = expressão sob TRUE,
3       no = expressão sob FALSE)
  
```

Funções vetorizadas

Funções vetorizadas são funções que operam em cada um dos elementos de um vetor, sem necessitar de um *loop* para este fim. Por exemplo, a multiplicação em **R** é vetorizada, como pode ser observado no Código R 6.9.

Código R 6.9**Script R:**

```
1 # Função vetorizada "*"
2 x <- 1:4; x * 2
```

Console R:

```
[1] 2 4 6 8
```

Script R:

```
4 # Se não fosse vetorizada, fariamos
5 x <- numeric()
6 for (i in 1:4) {
7   x[i] <- i * 2
8 }
9 x
```

Console R:

```
[1] 2 4 6 8
```

Por meio do Código R 6.10, usando a função `ifelse()`, resolvemos o problema apresentado no Código R 6.7. Verificamos que a função `ifelse()` por ser vetorizada, o resultado retornado ocorre como esperado.

Percebemos no Código R 6.10, que o resultado agora é um vetor de comprimento 6. Isso significa que todos os valores de `w` foram avaliados na condição `w < x`. Porém, como forma de melhorar e proteger possíveis erros ao código, dado uma condição de entrada do usuário, estamos interessados em resultados específicos para determinadas condições. Daí, usamos o que chamamos de programação defensiva, sendo que inserimos essa estratégia, geralmente, no corpo das funções. No Código R 6.11, apresentamos os passos iniciais para o desenvolvimento.

Código R 6.10**Script R:**

```
1 x <- 5 ; w <- 3:8 # Objetos
2 ifelse(w < x, x, w) # Usando "ifelse"
```

Console R:

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

Código R 6.11**Script R:**

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

Console R:

```
numero_par: 2
```

A ideia no Código R 6.11 é escolher um número inteiro e o algoritmo verificar se é par ou ímpar. O que pode ocorrer como supostos erros que foram protegidos pelo código. Um primeiro possível erro é avaliar um valor que não seja numérico. Neste caso, a primeira condição `if(!is.numeric(x))` retorna uma mensagem cujo valor não

é um número. O segundo erro que poderia ocorrer é do valor inserido não ser um número inteiro. Dessa forma, truncamos o número apenas usando a sua parte inteira e desprezando a decimal sem o arredondamento. Tentarmos prever os erros dos comandos de entrada, em face do código desenvolvido pelo programador, é o que chamamos de programação defensiva, dando o suporte de a própria função identificar o possível erro para quem a utiliza.

Um próximo exemplo, apresentado no Código R 6.12, mostrará inicialmente por meio da condição `if()` e `else` um código mais complexo, que pode ser simplificado na sequência com o uso da função `switch()`, Código R 6.13. O fluxograma da função `switch()` pode ser observado pela Figura 6.4.

Percebemos pelo Código R 6.12, que muitas condições `if()` são necessárias para escolher a medida a ser calculada, definida pelo objeto `opcao`. Mas quando usamos a função `switch()`, Código R 6.13, o algoritmo acaba sendo simplificado. Essa função é originada da instrução em C de mesmo nome.

Código R 6.12

Script R:

```

1 set.seed(15) # Fixando a semente
2 x <- rnorm(1000) # Gerando 1000 numeros aleatorios
3 # Medidas descritivas
4 opcao <- "media"
5 if (opcao == "media") { # Media
6   cat("A média aritmética é:", round(mean(x), 4))
7 } else {
8   if (opcao == "mediana") { # Mediana
9     cat("A mediana é:", round(mean(x), 4))
10 } else {
11   if (opcao == "medapar") { # Media aparada
12     cat("A média aparada é:",
13         round(mean(x, trim = 0.1), 4))
14   }
15 }
16 }
```

Console R:

```
A média aritmética é: 0.037
```

O primeiro argumento da função `switch()`, representa justamente a opção ao qual desejamos que a função `switch()` avalie. Por exemplo, no Código R 6.13, escolhemos `opcao <- "media"`. Então, a função busca na lista de opções pelo nome "media", e sua respectiva expressão é avaliada. Na lista de opções, criamos e associamos novas funções à opção desejada, como pode ser observada pela Figura 6.4. Uma outra curiosidade sobre a função `switch()`, é que em baixo nível essa função é largamente utilizada nos códigos internos do **R**.

Código R 6.13**Script R:**

```
1 set.seed(15) # Fixando a semente
2 x <- rnorm(1000) # Gerando 1000 numeros aleatorios
3 # medida descritiva
4 # opcoes:
5 # "media", "mediana", "medapar"(media aparada)
6 switch(opcao,
7     media = cat("A média aritmética é:",
8             round(mean(x), 4)),
9     mediana = cat("A mediana é:",
10             round(mean(x), 4)),
11     medapar = cat("A média aparada é:",
12             round(mean(x, trim = 0.1), 4))
13 )
```

Console R:

```
A média aritmética é: 0.037
```

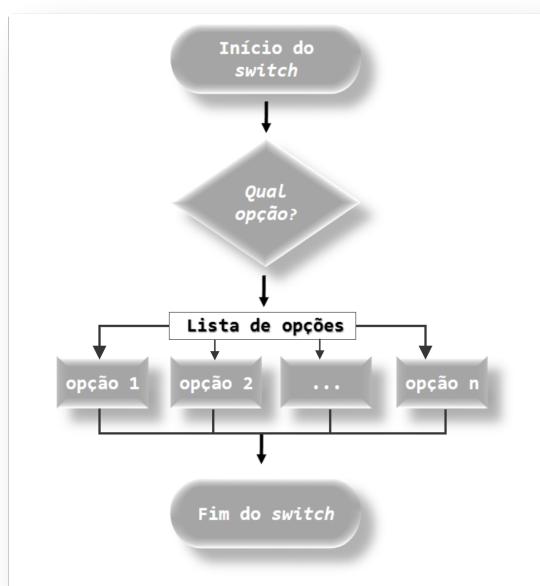
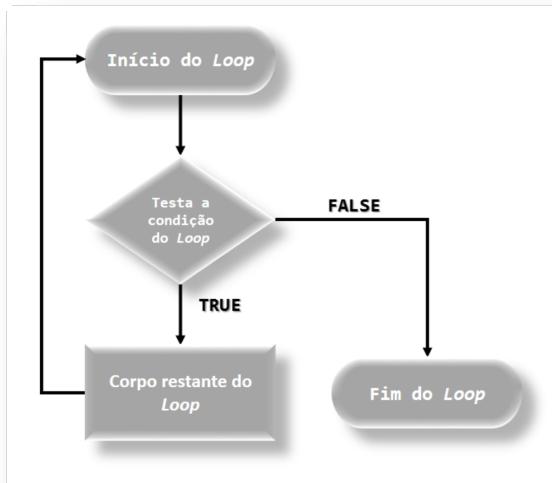


Figura 6.4: Fluxograma da função `switch()`.

6.6.1 Loops

As declarações que se repetem, dada uma determinada condição, em um código, as definimos como *loops*. As três estruturas básicas de *loop* no R são: `repeat`, `while()` e `for()`. Dizemos que um *loop* é um conjunto de declarações (expressões) em um algoritmo que se repete um número de vezes até que seja alcançado o objetivo desejado. 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.

A primeira função *loop* é `repeat`, cuja sintaxe é apresentada no Código R 6.14.

Figura 6.5: Fluxograma do *loop*.**Código R 6.14****Script R:**

```

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 6.15.

Observamos que a condição em `repeat` se repete até $i \leq 5$, Código R 6.15. Para que isso ocorra, nós quebramos o ciclo de repetição com a função `break`, em que o objeto `i <- 1`, linha 1 do Código R 6.15, é o contador do ciclo. Percebemos que a execução `print(i)`, linha 8 do

Código R 6.15, ocorre a cada ciclo como resultado, e que na sequência, o contador se atualiza, $i \leftarrow i + 1$, *linha* 9 do Código R 6.15. No momento em que o contador i é igual a 5, este valor é impresso no *console* e em seguida é atualizado para $i \leftarrow 5 + 1$, o que resulta no valor igual a 6. Neste 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 a função *break* entra em ação. Isto resulta na quebra do ciclo, e então a função *repeat* é encerrada. O intuito do Código R 6.15] não é apresentar a forma mais eficiente de explorar estas funções, mas mostrar além de suas sintaxes, o comportamento semântico.

Código R 6.15

Script R:

```

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

Console R:

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

No próximo exemplo, apresentamos a função *next*, Código 6.16. Mais uma vez, uma forma didática de entender essa função. O que diferencia o Código R 6.15 para o Código R 6.16 é o acréscimo da função *next* ao código. Quando a condição *if(i == 3)* é verdadeira, *linha* 9 do Código R 6.16, as *linhas* 10 e 11 são executadas. Primeiro atualizamos o contador $i \leftarrow 3 + 1$, *linha* 10 do Código R 6.16, e posteriormente, a

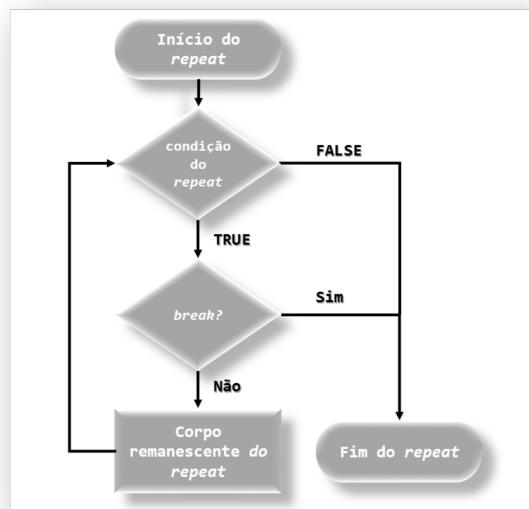


Figura 6.6: Fluxograma das funções `repeat()` e `break`.

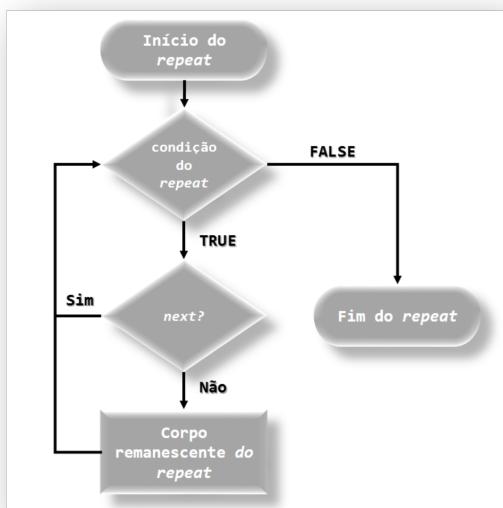


Figura 6.7: Fluxograma das funções `repeat()` e `next`.

função `next`, *linha* 10 do Código R 6.16, é executada. Inserimos a *linha* 12 no Código R 6.16] para termos a ideia da função `next`, que é avançar o ciclo quando esta função é executada, ou seja, quando o contador 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, o que não ocorre. O que aconteceu foi o avanço de ciclo em `repeat`. Devemos ter atenção com essa função, porque como não há critério de parada explícito, podemos incorrer em um *loop* infinito.

Código R 6.16

Script R:

```

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

Console R:

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

Como alternativa a função `repeat`, temos a função `while()`, cuja vantagem é a condição de parada explícita da função. O fluxograma é

dado pela Figura 6.8 tem sintaxe apresentada no Código R 6.17.

Código R 6.17

Script R:

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

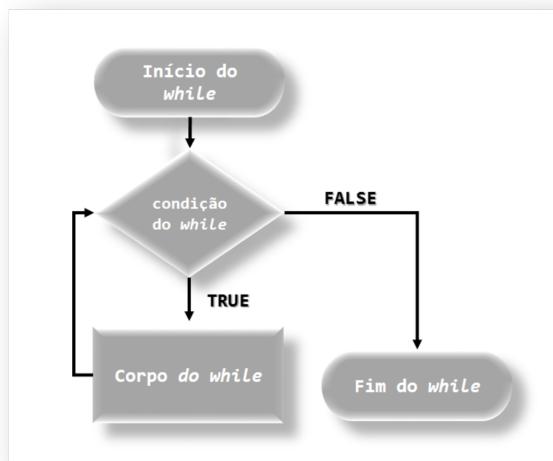


Figura 6.8: Fluxograma da função while().

Na função `while()` não precisamos usar a função `break` como critério de parada, como usado na função `repeat()`. Isso ocorre, porque a sintaxe da função `while()`, define explicitamente o fim do ciclo de repetição. Vejamos o exemplo no Código R 6.18, para mais detalhes.

Código R 6.18**Script R:**

```

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

Console R:

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

Percebemos no Código R 6.18, que diferentemente do Código R 6.15, a função `while()` declara seu critério de parada na expressão dentro do parêntese, *linha 4*. Logo, os valores impressos no *console* são os números de 1 a 5.

Devemos deixar claro que isso não significa restrição ao uso da função `break`. Dependendo do objetivo, impomos uma determinada quebra de ciclo por essa função. Tudo irá depender da implementação do código realizada pelo desenvolvedor. Por exemplo, vamos apresentar no Código R 6.19, uma implementação da função `next` dentro do ciclo de repetição da função `while()`, a seguir.

Observamos o Código R 6.19, similar ao que foi realizado com a função `repeat()`, Código R 6.16, do qual, impomos a restrição de não imprimir o valor 3, utilizando `next` dentro da função `while()`. Por fim, apresentamos mais uma opção de *loop*, é a função `for()`, com cuja sintaxe é apresentada no Código R 6.20.

Código R 6.19

Script R:

```

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

Console R:

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

Código R 6.20

Script R:

```

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

Com a função `for()`, também utilizamos `break` e `next`, de modo que o critério de parada também está declarada no parêntese, assim como a função `while()`, observando uma pequena mudança sintática no critério, como visto no Código R 6.20. Porém, o controle do ciclo nessa situação é maior, e menos necessário utilizar as funções `break` e `next`, como eventualmente pode ocorrer com a função `while()`. Isso porque a sintaxe da função `for()` define dentro do parêntese o critério de parada bem como o contador, tornando assim, mais simplificado a

implementação. Vejamos o Código R 6.9 para mais detalhes.

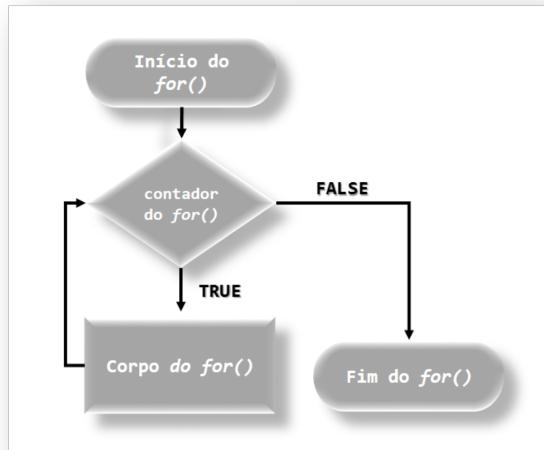


Figura 6.9: Fluxograma da função `for()`.

Código R 6.21

Script R:

```

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

Observamos que o mesmo algoritmo usado anteriormente com as outras estruturas de controle, para a função `for()`, em uma linha de comando o mesmo problema é resolvido, Código R 6.9, isto é, com essa estrutura de controle o código se torna mais simples para ser implementado. Vejamos um próximo exemplo, no Código R 6.22.

Código R 6.22

Script R:

```

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

Console R:

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

Mais uma vez utilizando a função `next`, sem necessidade de atualizar o contador `i`, uma vez que está implícito na função `for()`. O que fizemos no Código R 6.22 foi imprimir todos os valores de 1 a 5, exceto o valor 3. Isso foi possível por meio de `next`. Muitos outros recursos e possibilidades com essas estruturas de controle são possíveis, e a ideia foi mostrar de forma objetiva a sintaxe e semântica dessas funções.

Na próxima subseção, apresentaremos uma família de funções em **R**, chamada família *apply*. O objetivo dessas funções é evitar os *loops*, uma condição dentre outras características que representa o estilo funcional do ambiente **R**. Não estamos interessados em discutir se isto tornará o código mais eficiente ou não, mas uma simplificação e alternativa que pode ser implementada na linguagem.

6.6.2 Funções da família *apply*

Apesar de uma das construções mais interessantes na programação ser o *loop*, é bom observar que essas construções implementadas em diversas linguagens apresentam grandes diferenças, em termos de desempenho. Em **R**, os *loops* podem demandar um gasto computacional dependendo de sua implementação devido a diversos fatores, como:

cópias na modificação e estilo da linguagem interpretada, dentre outras características. Alternativamente, usamos uma série de funções vetorizadas, e dentre elas, temos as da família *apply*.

As funções da família *apply* foram implementadas no pacote **base**, e portanto, disponíveis ao usuário a qualquer momento. O objetivo dessas funções é manipular estruturas de dados como vetores, matrizes, *arrays*, listas e quadro de dados (*data frames*) de maneira repetitiva sem a utilização de *loop*. As funções são: *apply()*, *lapply()*, *sapply()*, *tapply()*, *mapply()*, *rapply()* e *eapply()*. A primeira função a ser discutida é *apply()*, que retorna um *array* ou uma lista obtida pela aplicação de uma função nas linhas ou colunas da entrada de um objeto seja matriz ou *array*. Vejamos a sintaxe dessa função no Código R 6.23.

Código R 6.23

Script R:

```
1 apply(X, MARGIN, FUN, ..., simplify = TRUE)
```

Os argumentos da função são:

- *X*, argumento que recebe o objeto matriz ou *array*;
- *MARGIN*, argumento que recebe 1, se a função em *FUN* deve ser aplicado na linha, ou recebe 2, se *FUN* deve ser aplicado nas colunas;
- *FUN*, argumento que recebe a função desejada;
- *...*, argumento que recebe argumentos adicionais para *FUN*, e
- *simplify*, argumento lógico para retorno de resultados simplificados (TRUE) ou não (FALSE).

Por exemplo, supomos que temos 5 amostras aleatórias de tamanho 10, com reposição, em um conjunto de valores de 1 a 1000, e desejamos computar a média aritmética dessas amostras que serão inseridas em colunas numa matriz. Para isso, vejamos o Código R 6.24.

Código R 6.24**Script R:**

```

1 # 5 amostras
2 (am1 <- sample(x = 1:1000, size = 10, replace = T))
3 (am2 <- sample(x = 1:1000, size = 10, replace = T))
4 (am3 <- sample(x = 1:1000, size = 10, replace = T))
5 (am4 <- sample(x = 1:1000, size = 10, replace = T))
6 (am5 <- sample(x = 1:1000, size = 10, replace = T))

```

Console R:

```

[1] 698 685 714 563 666 688 348 474 338 197
[1] 21 576 755 959 303 158 20 971 181 96
[1] 589 844 206 259 676 330 227 125 791 452
[1] 925 550 807 60 593 950 967 121 752 606
[1] 543 369 948 629 259 344 872 350 242 403

```

Script R:

```

7 # Amostras em colunas
8 amost_col <- matrix(c(am1, am2, am3, am4, am5),
9 10, 5); amost_col

```

Console R:

```

[,1] [,2] [,3] [,4] [,5]
[1,] 698 21 589 925 543
[2,] 685 576 844 550 369
[3,] 714 755 206 807 948
[4,] 563 959 259 60 629
[5,] 666 303 676 593 259
[6,] 688 158 330 950 344
[7,] 348 20 227 967 872
[8,] 474 971 125 121 350
[9,] 338 181 791 752 242
[10,] 197 96 452 606 403

```

Script R:

```
10 # Calculando a media por coluna  
11 apply(X = amost_col, MARGIN = 2, FUN = mean)
```

Console R:

```
[1] 537.1 404.0 449.9 633.1 495.9
```

Script R:

```
12 # Amostras em linhas  
13 amost_lin <- matrix(c(am1, am2, am3, am4, am5),  
14      5, 10, byrow = TRUE); amost_lin
```

Console R:

```
[,1] [,2] [,3] [,4] [,5]  
[1,] 698 685 714 563 666  
[2,] 21 576 755 959 303  
[3,] 589 844 206 259 676  
[4,] 925 550 807 60 593  
[5,] 543 369 948 629 259  
[,6] [,7] [,8] [,9] [,10]  
[1,] 688 348 474 338 197  
[2,] 158 20 971 181 96  
[3,] 330 227 125 791 452  
[4,] 950 967 121 752 606  
[5,] 344 872 350 242 403
```

Script R:

```
15 # Calculando a media por linhas  
16 apply(X = amost_lin, MARGIN = 1, FUN = mean)
```

Console R:

```
[1] 537.1 404.0 449.9 633.1 495.9
```

Script R:

```
17 # Adicionamos argumentos adicionais  
18 # em FUN (Media truncada em 10%)  
19 apply(X = amost_lin, MARGIN = 1, FUN = mean,  
20       trim = 0.1)
```

Console R:

```
[1] 557.500 381.125 441.250 663.000 471.125
```

Script R:

```
21 # Usando o argumento simplify  
22 # O padrao eh simplify = TRUE  
23 apply(X = amost_lin, MARGIN = 1, FUN = mean,  
24       simplify = FALSE)
```

Console R:

```
[[1]]  
[1] 537.1  
[[2]]  
[1] 404  
[[3]]  
[1] 449.9  
[[4]]  
[1] 633.1  
[[5]]  
[1] 495.9
```

A próxima função é lapply(), com sintaxe apresentada no Código R 6.25.

Código R 6.25

Script R:

```
1 lapply(X, FUN, ...)
```

Os argumentos da função são:

- X, argumento que recebe uma lista;
- FUN, argumento que recebe a função desejada, e
- ..., argumento que recebe argumentos adicionais para FUN.

Observamos que a sintaxe da função é muito parecida com a da função apply(), e sua implementação pode ser verificada no Código R 6.26.

Código R 6.26

Script R:

```
1 # Lista
2 lapply(list(x = 1:10, y = 11:20), mean)
```

Console R:

```
$x
[1] 5.5
$y
[1] 15.5
```

Script R:

```
3 lapply(list(x = 1:10, y = 11:20), "[[", 2)
```

Console R:

```
$x
[1] 2
$y
[1] 12
```

Script R:

```
4 lapply(list(mat1 = matrix(1:12, 4, 3)), "[", , 2)
```

Console R:

```
$mat1
[1] 5 6 7 8
```

A função seguinte é `sapply()`, que é um encapsulamento (*wrapper*) da função `lapply()`, e o acréscimo sintático do argumento padrão `simplify = TRUE`. Assim, a forma sintática da função pode ser observada no Código R 6.27.

Código R 6.27**Script R:**

```
1 sapply(X, FUN, ... , simplify = TRUE,
2     USE.NAMES = TRUE),
```

Os argumentos da função são::

- X, argumento que recebe uma lista;
- FUN, argumento que recebe a função desejada;
- ..., argumento que recebe argumentos adicionais para FUN;
- `simplify`, argumento lógico, se TRUE retorna o resultado de forma simplificada, sendo um vetor atômico, matriz ou *array*; se FALSE o retorno é uma lista;

- USE .NAMES, argumento lógico; se TRUE é retornado o nome inserido nos objetos da lista; se FALSE, caso contrário.

Na realidade, o que a função faz é melhorar a saída de `lapply()`, retornando um vetor, matriz ou array, sendo observado pelo Código R 6.28.

Código R 6.28

Script R:

```
1 lapply(list(1:10, 11:20), mean)
```

Console R:

```
[[1]]
[1] 5.5

[[2]]
[1] 15.5
```

Script R:

```
2 sapply(list(1:10, 11:20), mean)
```

Console R:

```
[1] 5.5 15.5
```

Script R:

```
3 # "simplify = FALSE" em "sapply()"
4 # eh equivalente a "lapply()"
5 sapply(list(1:10, 11:20), mean, simplify = FALSE)
```

Console R:

```
[[1]]  
[1] 5.5  
  
[[2]]  
[1] 15.5
```

Uma forma multivariada da função `apply`, é a função `mapply()`, com sintaxe no Código R 6.29.

Código R 6.29**Script R:**

```
1 mapply(FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,  
2     USE.NAMES = TRUE),
```

Os argumentos da função são:

- FUN, argumentos que recebe a função desejada;
- ..., argumentos para vetorização;
- MoreArgs, uma lista com argumentos adicionais a FUN;
- SIMPLIFY, argumento lógico, se TRUE retorna o resultado de forma simplificada, sendo um vetor atômico, matriz ou *array*; se FALSE o retorno é uma lista;
- USE.NAMES, argumento lógico; se TRUE é retornado o nome inserido nos objetos da lista; se FALSE, caso contrário.

Vejamos alguns exemplos de aplicação para a função `mapply()`, no Código R 6.30.

Código R 6.30

Script R:

```
1 mapply(FUN = mean, list(x = 1:10, y = 11:20),  
2 USE.NAMES = TRUE)
```

Console R:

```
x     y  
5.5 15.5
```

Script R:

```
2 mapply(FUN = mean, list(x = 1:10, y = 11:20),  
3 USE.NAMES = FALSE)
```

Console R:

```
[1] 5.5 15.5
```

Script R:

```
3 mapply(FUN = rep, x = 1:3, times = 1:3)
```

Console R:

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

Script R:

```
4 mapply(FUN = rep, x = 1:4, times = 9)
```

Console R:

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

Script R:

```
5 mapply(rep, times = 1:2, MoreArgs = list(x = 4))
```

Console R:

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

Um outro exemplo interessante usando `mapply()` pode ser usado supondo um conjunto de dados `1:38` e o objetivo é reamostrar, com reposição, 10 amostras de comprimento 10. Assim, temos:

Console R:

```
> mapply(sample, size = rep(10, 10), MoreArgs = list(x =
  1:38))
 [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,] 21   5   30  16  32  33  31   4   31  28
 [2,] 25   18   8   4   35  24  19   16  37  17
 [3,] 3    7   31  11  23  23  21   37  26  27
 [4,] 12   9   19   2   21  15  33   11  35   5
 [5,] 32   10   3   24   3   30  27   8   4   26
 [6,] 29   12   12  36  18  34  34   21  27  18
 [7,] 16   26   37   1   6   38  20   1   28  35
 [8,] 34   13   25  22  12  16  23   6   32   7
 [9,] 8    11   2   35   1   4   15  36  14  25
 [10,] 26  36   21  37   2   32   3   29  17  38
```

Podemos gerar 10 amostras de tamanho $n = 30$, de uma distribuição normal padrão da seguinte forma:

Console R:

```
> round(mapply(rnorm, n = rep(15, 5)), 5)
 [,1]      [,2]      [,3]      [,4]      [,5]
 [1,] -0.30972  0.04602 -0.53221 -0.48907 -1.11718
 [2,] -1.50149 -0.45802  1.32203  1.52889  0.56108
 [3,] -0.53646  0.38820  0.40452 -1.68816 -0.19447
 [4,]  0.32744  0.14499 -0.11372 -0.82293 -0.85429
 [5,] -0.50727 -0.76791  1.00992 -0.03697 -1.77689
 [6,] -0.20016 -0.22963  0.33083  0.41958 -0.79271
 [7,]  0.66449 -0.44782 -1.60611  0.69807  0.26933
 [8,]  0.28903 -0.70761 -0.90939  0.37719  0.21062
 [9,] -2.06655  0.86232  0.15750  0.36099 -0.67254
 [10,] -1.31116 -0.39083  0.08715 -0.22557  1.13463
 [11,] -0.25557  1.67938 -1.52589 -0.34123 -1.70374
 [12,] -0.21823  0.00122 -0.23712  0.15944  0.21615
 [13,] -0.30213 -0.49104 -1.16288 -1.22180  0.41063
 [14,] -1.96097  0.02953  0.35955  0.00803 -1.10046
 [15,]  0.43501  0.35404 -0.36389  0.37067  0.25605
```

Há mais recursos que podem ser realizados com a família de funções *apply* e sugerimos uma consulta nos manuais de ajuda dessas funções.

6.7 Criando funções

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

Como falado anteriormente, a estrutura da função criada se mantém: argumento, corpo e ambiente. Para isso, usaremos a função `function()`. O tipo do objeto é `closure`². Vejamos a sintaxe pelo Código R 6.31.

Código R 6.31

Script R:

```

1 # Forma usual
2 nome_funcao <- function(arg1, arg2, ...) {
3   corpo - comandos...
4 }
5 # Forma simplificada
6 nome_funcao <- function(arg1, arg2, ...) corpo
7
8 # Forma alternativa (>= R 4.1)
9 nome_funcao <- \(arg1, arg2, ...) {
10   corpo - comandos...
11 }
12
13 # Sintaxe simplificada (>= R 4.1)
14 nome_funcao <- \(arg1, arg2, ...) corpo

```

Desse modo, apresentamos o primeiro exemplo no Código R 6.32, do qual criamos uma função chamada `fun1()`, cujo argumento de entrada é `x`.

²Usamos `typeof()` para verificar o tipo do objeto.

Código R 6.32**Script R:**

```

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

Observamos que uma função é um objeto como qualquer outro. 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 o objeto associado com res, por meio da função return(). Para executar fun1(), fazemos:

Script R:

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

Console R:

```
[1] 6
```

Ao assumir $x = 5$ no argumento, essa informação é repassada para o corpo da função fun1(), onde existe o nome x que se associa ao objeto 5. A função soma “+” é chamada, e a expressão $x + 1$ é avaliada, cujo nome res se associa ao resultado da expressão. Por fim, quando a função fun1() é chamada, o resultado associado com res é impressa no *console*, por meio de return(res), são as chamadas funções com saídas explícitas. Vamos verificar os três componentes da função fun1():

Console R:

```

> # 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()`, sendo observado no Código R 6.33.

Código R 6.33

Console R:

```
> # Funcao
> fun2 <- function(x) x + 1
> # Executando
> fun2(5)
[1] 6
```

Caso a função `return()` não estivesse no corpo da função em desenvolvimento com o uso das chaves, seria retornado a execução da última linha de comando, assim como ocorreu pela chamada `fun2()`, que só apresentava uma única expressão no corpo função. Este comportamento semântico ocorre devido a chave ("{"), que é uma função do tipo `special`, avaliar todas as expressões entre as chaves, mas apenas o último resultado é impresso. Para mais detalhes, podemos consultar o manual de ajuda com a chamada `?Paren`. Porém, devemos entender melhor a semântica das chaves, como pode ser observado pelo Código R 6.34.

Código R 6.34**Script R:**

```

1 # Funcao fun3
2 fun3 <- function(x) {
3   y <- x + 1
4 }
5 # Funcao fun4
6 fun4 <- function(x) {
7   y <- x + 1
8   y
9 }
10 # Chamando as funcoes
11 fun4(1)

```

Console R:

```
[1] 2
```

Script R:

```
12 fun3(1)
```

A função `fun4()` imprime o valor 2, porque a última linha de comando no corpo da função é o nome `y`. Sabemos que neste caso, implicitamente a função `print()` é chamada, e por isso, o resultado associado com o nome é impresso no *console*. Já para a função `fun3()`, não houve a mesma ocorrência, pois a última linha de comando, `y <- x + 1`, associa o nome `y` com o resultado de `x + 1`. Agora, implicitamente a função `invisible()` é chamada e nada impresso.

No caso de termos uma função que não se associa a nome algum, a denominamos de função anônima, abordada a seguir.

6.7.1 Função anônima

Existe também o que chamamos de **função anônima**, da qual não associamos nome as funções. Estas funções não podem ser recuperadas. O

uso das funções anônimas 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 R:

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

Console R:

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

A função `integrate()` é utilizada para o cálculo da integral, do qual passamos os argumentos: função (`f`), limite inferior (`lower`) e limite superior (`upper`) da integração, respectivamente. Observamos que não houve necessidade de nomear a função repassada para o argumento `f`, 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 Ordenação de argumentos

Os argumentos nas funções podem ser nomeados ou não. Quando nomeados, não importa a ordem como os argumentos são inseridos na função. Já os argumentos não nomeados, os seus valores precisam estar na ordem como a função foi desenvolvida. Vejamos o Código R 6.35, para entender melhor.

Código R 6.35**Script R:**

```

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   )
7   return(res)
8 }
9 set.seed(15) # Semente
10 x <- rnorm(1000) # Objeto
11 # Argumentos nomeados na função
12 estdesc(x = x, opcao = "media")

```

Console R:

[1] 0.037

Script R:

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

Console R:

[1] 0.037

Script R:

```

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

```

Console R:

[1] 0.037

Script R:

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

Console R:

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

Criamos uma função `estdesc()` para calcular algumas medidas descritivas de posição. Os seus argumentos são, nessa ordem, `x` e `opcao`. Observamos pelo Código R 6.35, quando descrevemos o nome dos argumentos na função `estdesc()`, a ordem como inserimos seus argumentos não importa. Porém, quando não o nominamos, a declaração se torna mais simples, sendo o primeiro valor inserido em `estdesc()` correspondente ao argumento `x`, e o último valor, ao argumento `opcao`, nessa ordem.

6.7.3 Objeto reticências (“...”)

O objeto reticências é um tipo especial de objeto “`pairlist`”, mais especificamente do tipo “`DOTSXP`” na tipagem interna em C. O tipo de objeto “`pairlist`” é usado bastante internamente no **R**, dificilmente utilizado no código interpretado. Desse modo, não temos acesso fácil a estrutura do objeto “`...`”.

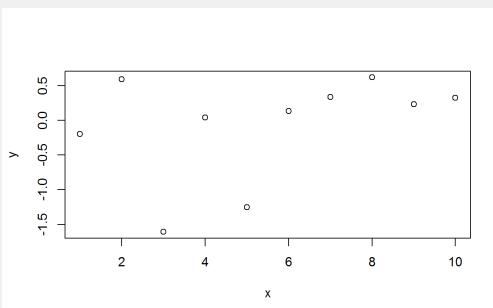
Contudo, a reticências tem um papel fundamental nas funções, porque temos a liberdade de inserir mais argumentos além dos definidos na criação da função. Na função `plot()`, por exemplo, função do pacote nativo **graphics**, o argumento “`...`” é usado na função, por apresentar muitas opções para o estilo gráfico desejado. Vejamos como a usamos em uma função no Código R 6.36.

Código R 6.36**Script R:**

```

1 # Funcao que plota um grafico
2 grafico <- 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 grafico(x = x, y = y)

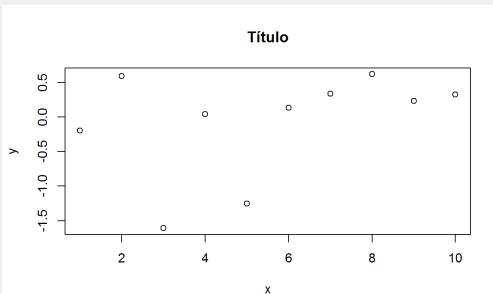
```

**Script R:**

```

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

```



Se pesquisarmos sobre a função `plot()`, usando a chamada `?plot.default`, verificaremos que ela é uma função de nível superior na criação de gráficos, apresentando muitos argumentos devido a complexidade com que se exige na criação de um gráfico. Dessa forma, criamos a função `grafico()`, com apenas dois argumentos nominados, do qual representam as coordenadas e também inserimos o objeto `reticências` (...). Observamos que a função `plot()` por também apresentar o objeto `reticências`, toda a lista de argumentos repassados para `grafico()` também é repassada para `plot()`. Para que isso ocorra, observamos no Código 6.36, linhas 3, que o objeto `reticências` é explicitado na função `plot()`. Caso isso não fosse realizado, os argumentos adicionais não seriam repassados, como pode ser observado no Código 6.37.

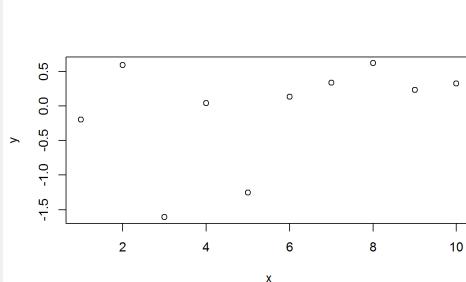
Código R 6.37

Script R:

```

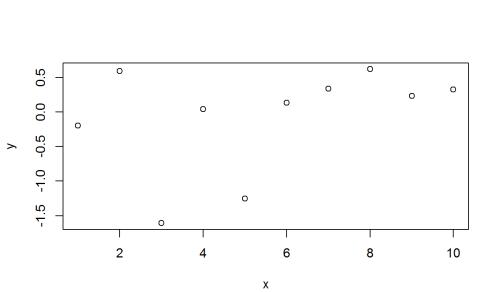
1 # Funcao que plota um grafico
2 grafico <- function(x, y, ...) {
3     plot(x = x, y = y)
4 }
5 # Vetores
6 x <- 1:10; y <- rnorm(10)
7 # Chamada 1, com os argumentos definidos
8 grafico(x = x, y = y)

```



Script R:

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



Os argumentos adicionados por meio de “...” podem ser recuperados em forma lista, no corpo das funções. Também apresentamos outras características que podem ser obtidas nos argumentos definidos no objeto reticências, como segue:

Console R:

```
> # Recuperando o valor do primeiro argumento
> aux <- \(...) ..1
> aux(x = 1, y = 2, z = NULL)
[1] 1
> # Recuperando o valor do segundo argumento, e assim por diante
> aux <- \(...) ..2
> aux(x = 1, y = 2, z = NULL)
[1] 2
> # Recuperando os nomes
> aux <- \(...) ...names()
> aux(x = 1, y = 2, z = NULL)
[1] "x" "y" "z"
> # Recuperando o numero de argumentos em ...
> aux <- \(x, ...) ...elt(x) # x eh um argumento da função
> aux(x = 1, y = 2, z = NULL)
[1] 2
```

```
> # Recuperando o numero de argumentos em "..."
> aux <- \(...) ...length()
> aux(x = 1, y = 2, z = NULL)
[1] 3
> # Recuperando os argumentos de "..."
> aux <- \(...) list(...)
> aux(x = 1, y = 2, z = NULL)
$x
[1] 1

$y
[1] 2

$z
NULL
```

Por fim, uma coisa que vale ressaltar é que argumentos adicionais nominados erroneamente em funções com reticências (...), os erros são omitidos na saída do resultado das funções. Um exemplo é a função genérica `mean(x, ...)`, cuja função apresenta apenas um argumento `x`. Vejamos o que acontece se colocarmos elementos adicionais erroneamente:

Console R:

```
> mean(1, 2, 3, 4)
[1] 1
```

Observamos que `x = 1`, e os demais foram argumentos adicionais, inseridos erroneamente. Logo, a função retornou apenas o valor 1. Como os argumentos adicionais foram inseridos errados, nenhuma mensagem de alerta ou erro foi indicado pela função após a sua execução. Assim, apesar de ser um objeto muito interessante do ambiente **R**, devemos utilizar com cautela. Para um maior controle dos erros, o pacote chamado **elipse** foi desenvolvido pela equipe do **RStudio** para tal fim.

De todo modo, devido à complexidade do objeto reticências, aprofundaremos o assunto no *Volume II*, apresentando algumas estratégias de como utilizá-lo.

6.8 Escopo léxico

Vamos retornar aos componentes da função `fun1()`, desenvolvida no Código R 6.32, reapresentando as três estruturas de uma função:

Console R:

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

O último componente é o ambiente onde o nome `fun1` foi associado à função, sendo chamado de **ambiente envolvente**; neste caso, é o ambiente global. Contudo, quando a função é executada, momentaneamente é criado o **ambiente de execução**; é nele que os nomes dentro do corpo da função são associados aos objetos. Vejamos um primeiro exemplo no Código R 6.38.

Código R 6.38

Script R:

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

Console R:

[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 daqueles presentes no ambiente envolvente. Esta é uma primeira característica do **escopo léxico** nas funções em **R**.

Atribuição e Escopo



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 envolvente. 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 definido no ambiente de execução, a função inicialmente verifica se não existe algum objeto associado com o nome no corpo da função, e em caso negativo, ele será procurado no **ambiente de chamada**³ da função, como pode ser observado no Código R 6.39.

³Ambiente onde a função foi chamada.

Código R 6.39**Script R:**

```

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

```

Console R:

```
[1] 10
```

O resultado de `fun()` foi 10, Código R 6.39, porque a função procurou no ambiente de execução e não encontrou o nome. A função foi até o ambiente superior, no caso, o ambiente de chamada. Como falado anteriormente, todo ambiente tem um pai⁴ (ou ambiente superior). Essa hierarquização é observada ao caminho de busca, que pode ser acessada por `search()`, ou seja:

Console R:

```

> # Simplificamos o termo "package" por "pack"
> search()
[1] ".GlobalEnv"  "pack:magrittr"  "pack:leaflet"
[4] "pack:stats"   "pack:graphics"  "pack:grDevices"
[7] "pack:utils"   "pack:datasets"  "pack:methods"
[10] "Autoloads"    "pack:base"

```

O ambiente corrente do **R** sempre será o 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. O ambiente de execução só existe quando a função é chamada e não quando é definida, como pode ser verificado no seguinte código:

⁴O ambiente vazio não tem ambiente superior.

Console R:

```
> # Função
> fun <- function() x + 10
> # Objeto 1
> x <- 10
> # Chamada1
> fun()
[1] 20
> # Objeto 2
> x <- 20
> # Chamada 2
> fun()
[1] 30
```

Observamos 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`. A função procura os valores quando ela é executada, e não quando é criada. Isso é a característica de pesquisa dinâmica do escopo léxico. Uma última situação pode ser observada no Código R 6.40.

Código R 6.40**Script R:**

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

Console R:

```
[1] 2
```

Script R:

```
10 # Chamada 2
11 fun()
```

Console R:

```
[1] 2
```

Nessas linhas de comando do Código R 6.40, 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. O resultado, independente do número de chamadas, será sempre o mesmo, porque uma outra característica do escopo léxico no **R** é o **novo começo**, uma 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 chamada da função `function()` não necessariamente necessita de definição de argumentos, devido a flexibilidade do escopo léxico das funções em **R**, exemplos nos Códigos R 6.38 e 6.39. É essa característica que faz com que os comandos no corpo das funções encontrem os objetos que não estão definidos na própria função.

Queríamos ressaltar que o estilo funcional do ambiente **R** exige um conhecimento mais aprofundado sobre o paradigma da programação, que foge ao escopo deste *Volume*. Mas ao final, foi possível observar diversas características existentes numa função, bem como suas peculiaridades com a sintaxe e semântica quando observamos outras linguagens. E por fim, a flexibilidade de uma função no **R**, obriga ao usuário um maior conhecimento sobre esse tipo de objeto.

6.9 Exercícios

Exercício 6.1: Verifique se as funções, a seguir, são primitivas ou não: `sd`, `UseMethod`, `switch`, ‘`<-`’, ‘`%%`’, ‘`%in%`’, ‘`>`’, ‘`=`’, `is.primitive` e `is.function`.

Dica na página 295

Exercício 6.2: Após verificado pelo Exercício 6.1, os tipos de funções, use as chamadas `formals()`, `body()`, `environment()`, para verificar as três estruturas de uma função. O que pode ser observado entre funções primitivas e funções do tipo “closure”?

Dica na página 295

Exercício 6.3: Considere um conjunto de valores gerado por `rnorm(30)`. Calcule a média dos dados e armazene o resultado associado com um nome. Posteriormente, imprima o resultado. Reformule todo o processo, usando o operador `pipe (|>)`.

Dica na página 295

Exercício 6.4: Crie uma função para verificar se um número é par ou ímpar.

Dica na página 295

Exercício 6.5: Crie uma função para calcular um fatorial, isto é, $n!$.

Dica na página 295

Exercício 6.6: Crie um conjunto de valores positivos e negativos, por exemplo, com `runif(30, -5, 6)`. Use as estruturas de controle para encontrar o primeiro valor negativo, considerando os valores da esquerda para à direita. Ainda neste problema, tente resolvê-la sem utilizar as estruturas de controle.

Dica na página 296

Exercício 6.7: Use a função `repeat` para verificar e imprimir os números de 1 a 100, quais são valores pares.

Dica na página 296

Exercício 6.8:

Crie uma função para calcular a moda, uma medida estatística de tendência central. A moda é dada pelo valor de maior frequência. Se todos os valores se repetirem igualmente, a condição é **amodal**. Se dois valores se repetem em mais do que os demais, em mesma frequência, a condição é **bimodal**, e assim por diante, para a condição **trimodal**, e por fim, a condição é **multimodal** quando mais de três se repetem mais vezes e em frequências iguais. OBS.: Considere a condição para dados não agrupados.

Dica na página 296

Exercício 6.9: Crie uma matriz (5×5) e calcule o valor máximo dos valores, por linha. Posteriormente, faça o mesmo, com base nas colunas. Para o desenvolvimento, use a estrutura de controle `for()`.

Dica na página 296

Exercício 6.10: Baseado no Exercício 6.7, implemente-o usando funções da família `apply`.

Dica na página 296

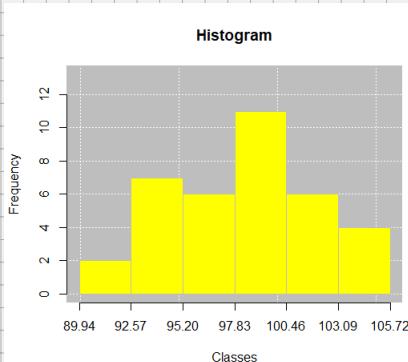
Exercício 6.11: Vejamos a seguinte implementação:

Script R:

```

1 # Instalando o pacote leem
2 install.packages("leem")
3 # Anexando o pacote leem
4 library(leem)
5 # Criando um histograma
6 x <- rnorm(36, 100, 4)
7 x <- new_leem(x, variable = 2)
8 x <- tabfreq(x)
9 hist(x)

```



Como poderíamos executar o código usando o operador *pipe* nativo (`|>`)?

Dica na página 297

Exercício 6.12: Apresente um código cujo ambiente de chamada de uma função não seja o ambiente global.

Dica na página 297

Exercício 6.13: Como reescrevemos a linha de comando:
``/`(`^+`(5, 15), `*`(2, 10))`?

Dica na página 297

Exercício 6.14: O que representa a chamada `?`?`()`?

Dica na página 297

Exercício 6.15: Qual a diferença sintática e semântica das funções `fun01()` e `fun02()` apresentadas no *script* a seguir?

Script R:

```
1 fun01 <- function(){  
2   x + 1  
3 }  
4  
5 fun02 <- function(...) {  
6   x + 1  
7 }
```

Dica na página 297

Exercício 6.16: Já verificamos por meio da função `mean()`, que o objeto reticências (...) apesar de interessante, omite erros nos argumentos adicionais de funções. Retomando o problema encontrado em `mean()`,

Console R:

```
> mean(1, 2, 3, 4)  
[1] 1
```

como poderíamos criar uma função com a chamada `media(1, 2, 3, 4)`, por exemplo, e esta função nos retornasse o valor correto da média, para o conjunto de dados, que é 2,5? Percebemos que os valores estão na função cada um como argumento diferente, e não como um conjunto de dados atribuídos a um único argumento. **Observação:** Podemos usar o primeiro argumento como um vetor ou não, como pode ser observado com a função `sum()`:

Console R:

```
> sum(1:4)
[1] 10
> sum(1, 2, 3, 4)
[1] 10
```

Dica na página 297

Capítulo 7

Boas práticas de como escrever um código

7.1 Introdução

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

- I) guardá-lo para futuras consultas, e/ou
- II) compartilhamento.

Nos dois casos, percebemos que uma possível consulta sobre o código escrito, ou continuação do código inacabado depois de um tempo, nos trará de volta àquelas linhas de código, de modo a indagar: quais as ideias por trás da escrita? A resposta dependerá da forma como foi desenvolvido o *script*, isto é, as boas práticas de escrita de código.

A primeira ferramenta a ser configurada pelo **RStudio** é acionar todas as opções de diagnóstico do código. Para isso, devemos seguir os passos via 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.

Após as configurações, colorações nas linhas de comando ocorrem, distinguindo diversas formas sintáticas, como: linhas de comentário, funções, espaçamentos, dentre outras estruturas da linguagem.

Uma vez realizado o procedimento, vamos para o passo seguinte das boas práticas de como escrever um *script*. Temos algumas ferramentas disponíveis, como o pacote **styler** e como alternativa o pacote **formatR**, que automatiza todo o código, seja em *script* contido em um pacote, ou diretório. Para mais detalhes em <https://yihui.org/formatr>. Para instalar e anexar o pacote **styler**, usamos as linhas de comando como segue:

Console R:

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

Vejamos a Figura 7.1, para entender a funcionalidade desse pacote, na ordem dos passos 1, 2, 3 e 4, respectivamente. Percebemos a modificação que ocorre no código da Figura 7.1a (Passo 1), e após a organização do código que pode ser observado na Figura 7.1d (Passo 4). As duas rotinas podem ser visualizadas no Código R 7.1.

Código R 7.1

- Código sem estilo de boas práticas

Script R:

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

- Código com estilo de boas práticas

Script R:

```

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

A screenshot of the RStudio interface showing a script editor window titled "Untitled1". The code in the editor matches the one shown in the "Script R:" box. The status bar at the bottom indicates "10:1 (Top Level)".

(a) Passo 1

A screenshot of the RStudio interface showing the "Addins" menu open. The "styler" package is listed under the "STYLER" section. An arrow points from the "styler" entry to the "Value to clipboard" option in the dropdown menu.

(b) Passo 2

A screenshot of the RStudio interface showing the "Addins" menu open. The "styler" package is highlighted in the "STYLER" section. An arrow points from the "styler" entry to the "Style active file" option in the dropdown menu.

(c) Passo 3

A screenshot of the RStudio interface showing the R code with syntax highlighting applied. The code is now color-coded: "# Contador" is blue, "i <- 1" is green, "repeat {" is red, "if (i > 5) {" is red, "break" is red, "else {" is red, "print(i)" is blue, "i <- i + 1" is green, and "}" is red. An arrow points from the "Pretty-print active file" option in the dropdown menu to the highlighted code.

(d) Passo 4

Figura 7.1: Passos para configurar o pacote **styler**.

Nas próximas seções, falaremos sobre algumas sugestões para uma

boa escrita de código.

7.2 Nomes de arquivos

Apesar de toda a facilidade de se utilizar o pacote **styler**, vamos apresentar algumas ideias. A primeira noção é o nome do arquivo. Devemos padronizar a extensão “<>.R” e evitar espaços em nomes compostos, por exemplo:

Script R:

```
1 # Boa escolha:  
2 # -----  
3 # script.R  
4 nome_composto.R  
5 nome-composto.R  
6  
7 # Mau escolha:  
8 # -----  
9 script.r  
10 nome composto.r
```

7.3 Comentar as linhas de comando

Sempre que escrevemos uma linha de comando ou uma sequência de linhas de comando para um mesmo fim, devemos comentar sobre a ideia que o fragmento de código representa para o algoritmo, usando “#”. Devemos evitar acentos e símbolos especiais, porque os novos acessos ao *script*, dependendo da codificação de caracteres que o **RStudio** (ou no próprio **R**) esteja utilizando ou até mesmo o sistema operacional, podem desconfigurar todo o arquivo. Sugestões:

Script R:

```
1 # Boa escolha:  
2 # -----  
3 # Objeto x
```

```

4 x <- 1:10
5 # Calculo da media
6 mean(x)
7 # Mau escolha:
8 # -----
9 x <- 1:10
10 mean(x)

```

7.4 Nomes associados a objetos

Os nomes associados aos objetos são o mecanismo de recuperar os objetos após a criação, e não os nomes das variáveis que são utilizados em relatórios para escrever sobre um determinado problema. Portanto, devemos escolher nomes mais simples, curtos, sem acentos, evitando símbolos de codificação ASCII, e de preferência apenas letras minúsculas. Sugestões:

Script R:

```

1 # Boa escolha:
2 # -----
3 # nome_curto
4 aux1
5 # Mau escolha:
6 # -----
7 # nome_muito_grande
8 Nome_Grande
9 Aux1

```

Devemos evitar, sempre que possível, nomes já utilizados no **R**, como também as palavras reservadas, para outros fins, isto é:

Script R:

```

1 # Mau escolha:
2 # -----
3 T <- "Nada"

```

```

4 c <- 5
5 # Nome ja usado para a funcao que calcula o desvio padrao
6 sd <- 5 + 1
7 # Nome ja usado para a funcao que calcula a media
8 mean <- 3 * 4

```

Para verificar se determinado nome já existe, usamos a função `exists()`; por exemplo, para saber se existe o nome “`mean`”, usamos `exists("mean")`. Se a função retornar `TRUE`, significa que existe o nome procurado, caso contrário, `FALSE`.

7.5 Sintaxe

Um dos erros mais comuns na sintaxe de um código é o **espaçamento** entre os operadores básicos na linguagem **R**, exceto para os operadores `:`, `::` e `:::`. Devemos sempre usar um espaço após a vírgula, como segue no *script*:

Script R:

```

1 # Boa escolha:
2 # -----
3 x <- 1:10
4 media <- mean(x + 1 / length(x), na.rm = TRUE)
5 base::mean(x)
6
7 # Mau escolha:
8 # -----
9 x <- 1 : 10
10 media<-mean(x+1/length(x),na.rm=TRUE)
11 base :: mean(x)

```

Para facilitar, utilizamos a função `styler::style_text()`, isto é:

Console R:

```

> comando <- "media<-mean(x+1/length(x),na.rm=TRUE)"
> styler::style_text(comando)

```

```
media <- mean(x + 1 / length(x), na.rm = TRUE)
```

Se necessário, podemos usar mais de um espaço em uma linha para o alinhamento da atribuição `<-` ou `=`, bem como um espaço antes do parêntese, a menos que seja uma função, ou seja:

Script R:

```
1 # Boa escolha:
2 # -----
3 data.frame(a = 1,
4             b = "Ben")
5 function() {
6     x <- 10
7     vari <- x + 1
8     return(vari)
9 }
10 for (i in 1:10) i + 1
11 # Mau escolha:
12 # -----
13 function () 1
14 for(i in 1:10)i+1
```

Não devemos inserir espaços na condição entre parêntese ou nos itens de indexação, a menos que este último contenha uma vírgula, como segue:

Script R:

```
1 # Boa escolha:
2 # -----
3 if (verbose)
4 x11 <- mat[1, 1]
5 x1 <- mat[1, ]
6 # Mau escolha:
7 # -----
8 if ( verbose )
9 x11 <- mat[1,1]
```

```
10 x1 <- mat[1, ]
```

Quando usamos chaves em um comando, devemos evitar abri-la e fechá-la na mesma linha. O mesmo é recomendado para funções, devemos adicionar recuo com dois espaços às linhas de comando inseridas dentro das chaves, para facilitar o entendimento hierárquico das funções, isto é:

Script R:

```
1 # Boa escolha:  
2 # -----  
3 fx <- function(x) {  
4   if (x > 2) {  
5     print("Maior que 2!")  
6   } else {  
7     print("Menor que 2!")  
8   }  
9 }  
10 for (i in 1:10) {  
11   x <- i + 1  
12 }  
13 # Mau escolha:  
14 # -----  
15 fx <- function(x) {  
16   if (x > 2) { print("Maior que 2!")}  
17   else { print("Menor que 2!")}  
18 }  
19 for (i in 1:10) {x <- i + 1}
```

Ao usar funções, devemos inicialmente, abrir a chave respeitando o espaço para fechar o parêntese. Por fim, usamos a atribuição com `<-`, e para definir os argumentos de uma função, não só pela sintaxe, mas por outros problemas já comentados anteriormente. Assim, para exemplificarmos, segue mais um *script*:

Script R:

```
1 # Boa escolha:  
2 # -----  
3 fx <- function(x = NULL) 10  
4 a <- "Nome"  
5 # Mau escolha:  
6 # -----  
7 fx = function(x = NULL) 10  
8 a = "Nome"
```

Para consultas complementares, deixamos a sugestão de leitura do capítulo sobre código no livro *R packages* (WICKHAM, 2015), em <https://r-pkgs.org/r.html#code-style>, o guia de estilo *tidyverse* em <https://style.tidyverse.org/> e o guia de estilo R do *Google* em <https://google.github.io/styleguide/Rguide.html>.

7.6 Exercícios

Exercício 7.1:

De uma forma geral, a conduta das boas práticas de escrita de um código, além da elegância e organização da escrita, quais outras vantagens podem ser verificadas?

Dica na página 298

Exercício 7.2:

Usando o operador *pipe*, apresente uma boa escrita de código para o seguinte *script*:

Script R:

```
1 airquality |> subset(Temp > 80,  
2 select = c(Ozone, Temp)) |>  
3 apply(2, mean, na.rm = TRUE)
```

Dica na página 298

Exercício 7.3:

Quais outras sugestões de uma boa escrita de código poderiam ser propostas?

Dica na página 298

Exercício 7.4:

Baseado nos elementos de uma lista, organize-os para uma boa escrita, no seguinte *script*:

Script R:

```
1 lista <- list(total = a + b + c, media =  
2 (a + b + c) / n, maximo = max(c(a, b, c))  
3 , amplitude = max(c(a, b, c)) - min(c(a, b, c)))
```

Dica na página 298

Exercício 7.5:

Reescreva o código abaixo, dentro do estilo das boas práticas de um código.

Script R:

```

1 dSMR <- function (x, size, df, np = 32, log = FALSE)
2 {
3   nn <- max(length(x), length(size), length(df),
4   length(np))
5   if (nn == length(x) | length(x) > nn)
6     xx <- cbind(x)
7   else if (length(x) == 1 | length(x) < nn)
8     xx <- cbind(rep(x, length = nn))
9   if (nn == length(size) | length(size) > nn)
10    xx <- cbind(xx, size)
11  else if (length(size) == 1 | length(size) < nn)
12    xx <- cbind(xx, rep(size, length = nn))
13  if (nn == length(df) | length(df) > nn)
14    xx <- cbind(xx, df)
15  else if (length(df) == 1 | length(df) < nn)
16    xx <- cbind(xx, rep(df, length = nn))
17  if (nn == length(np) | length(np) > nn)
18    xx <- cbind(xx, np)
19  else if (length(np) == 1 | length(np) < nn)
20    xx <- cbind(xx, rep(np, length = nn))
21  dtched <- function(xx) return(dMR(xx[1],
22  xx[2], xx[3], xx[4]))
23  d <- apply(xx, 1, dtched)
24  if (log == TRUE)
25    d <- log(d)
26  return(d)
27 }
```

Dica na página 298

Capítulo 8

Pacotes

8.1 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 cerca de 30 pacotes nativos (padrão e recomendados), 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` executado pelo *prompt* de comando do sistema operacional, que será visto no *Volume V*, coleção *Estudando o Ambiente R*. Isso acontece no SO *Linux*, e para as plataformas *Windows* e *Macintosh*, existem também os pacotes binários ou compactados com a extensão “.zip” ou “.tgz”, respectivamente.

Pacote R



Um pacote é a unidade básica para o compartilhamento de um código em **R**.

Atualmente, até 25/08/2022, o número de pacotes disponíveis no CRAN é 18.525. Encontramos 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 após a submissão,

uma checagem mais aprofundada 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 onde podem ser disponibilizados os pacotes, como por exemplo, Biocondutor, R-Forge, **GitHub**. O **GitHub** está sendo muito utilizado nos 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 garantir 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 pode ser submetido ao journal do **R**, o *The R Journal* ou ao *Journal of Statistical Software*, por exemplo. Aí sim, o pacote é analisado em toda sua composição computacional quanto metodológica.

Portanto, entendam 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 revista 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 este adendo pelo seguinte motivo: quando pesquisamos sobre determinado assunto, podem existir diversos pacotes para um determinado assunto. Dessa forma, qual pacote devemos escolher? Aquele que é mais fácil de utilizar? Uma boa pesquisa para a escolha não deve levar em consideração apenas a facilidade de utilização do pacote, mas saber se suas funções realmente retornam os resultados confiáveis para aquilo que se destina.

Para uma instalação mais rápida dos pacotes, podemos optar pelos espelhos disponíveis nos países em que estamos utilizando o **R**, 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 (Universidade Federal do Paraná). Mas temos mais quatro espelhos: dois na USP (Universidade de São Paulo) e uma na Fiocruz/RJ (Fundação Osvaldo Cruz). Para mais espelhos, podemos

acessar a página <https://cran.r-project.org/mirrors.html>.

Devemos deixar claro que, erroneamente, alguns usuários usam o termo “biblioteca” como um sinônimo de “pacote”.

Biblioteca não é um pacote!



Uma biblioteca em **R** não é um pacote, mas um diretório que armazena diversos pacotes.

Nas documentações do **R**, a 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 SO *Windows* são as *DLLs*.

8.2 Estrutura básica de um pacote

A estrutura básica de um pacote é apresentada na Figura 8.1.

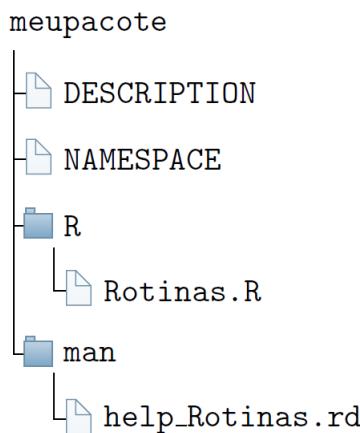


Figura 8.1: Esqueleto básico de um pacote.

Vejamos as ideias básicas dos subdiretórios e arquivos:

- DESCRIPTION:** arquivo de texto contendo informações básicas como o título do pacote, versão, licença, descrição, nome dos auto-

res, e o mantenedor do pacote; quando alterações no pacote forem necessários ou dúvidas de como utilizá-lo, será para o mantenedor a realização do contato para tal fim. As informações citadas são obrigatórias, porém existem mais informações que podem ser adicionadas, dependendo da proposta e funcionalidades do pacote;

- b) **NAMESPACE**: arquivo que se destina a importação e exportação de funções no pacote. Será neste arquivo que informaremos quais os pacotes dependentes, isto é, quais as funções ou outros objetos que utilizaremos de outros pacotes;
- c) **R/**: subdiretório que apresenta os *scripts* com as funções em **R**, o subdiretório principal do pacote;
- d) **man/**: subdiretório que apresenta os arquivos de ajuda, com extensão “.Rd”. Uma vez instalado o pacote no **R**, o acesso aos manuais de ajuda do pacote estarão disponíveis, devido a este subdiretório.

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

8.3 Instalação de um pacote

A instalação de um pacote via **R** pode ser obtida por meio da chamada da função `install.packages("nome_pacote")`. Por exemplo, vamos tentar instalar o pacote **midrangeMCP**, da seguinte forma:

Console R:

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

Dependendo de onde estivermos executado essa linha de comando, como: **R**, **RStudio** ou outra *IDE*, será solicitado o espelho por onde desejamos realizar a instalação. Quando isso ocorrer, e será apenas uma vez, na primeira instalação de um pacote no **R**. Para as próximas instalações de pacotes não será necessário informar o espelho do *CRAN*, a menos que seja do nosso interesse.

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 a *url*: <http://cran.r-project.org/package=midrangeMCP>

midrangeMCP. Para qualquer outro pacote, mudamos apenas o nome do pacote na *url*, e assim, estaremos na página do repositório do pacote desejado. O pacote fonte, como falado anteriormente, é compactado com a extensão “<>.tar.gz”, como para o exemplo do pacote citado, temos: midrangeMCP_3.1.1.tar.gz. O pacote binário tem a compactação zipada, como: midrangeMCP_3.1.1.zip para o *SO Windows* e midrangeMCP_3.1.1.tgz para o *SO 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 computador e instalá-lo via comando:

Console R:

```
> 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, devemos informar o local onde o pacote se encontra no computador. Para o *SO Window* ou *SO Macintosh*, é possível instalar também, a partir dos pacotes binários.

Muitos dos desenvolvedores estão disponibilizando seus projetos de pacotes, principalmente no **GitHub**, inclusive com manuais de ajuda com mais detalhes. Instalamos os pacotes diretamente do **GitHub**, da seguinte forma: inicialmente, instalamos o pacote **devtools**; posteriormente, instalamos o pacote desejado. Por exemplo, vamos instalar o pacote **midrangeMCP** diretamente de seu repositório do **GitHub**, e identificamos inicialmente o usuário do repositório do pacote, que neste caso, é “*bendeivide*”. Com essas informações para a instalação do pacote, usamos as linhas de comando:

Console R:

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

Contudo, devemos dar a preferência pela instalação via *CRAN*. Por lá, teremos a garantia de que os pacotes serão estáveis para a utilização nos sistemas operacionais mencionados acima.

Por fim, 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 bá-

sicos (*SO Windows, Unix e SO Mac*) exigidos pelo **R**. Dessa forma, se as correções não forem realizadas no tempo determinado pelos mantenedores do **R** (*R Core Team*), esses pacotes se tornarão órfãos, ou seja, desativados sob o *CRAN*.

8.4 Objetivos de um pacote

A ideia de um pacote **R** deve representar como uma ferramenta para otimizar as atividades do dia a dia na utilização da linguagem. Suponha que todos os dias carregamos uma sequência de *scripts* via comando `source()`, para disponibilizar nossas funções no ambiente global. Isso acaba gerando processos repetitivos de trabalhos desnecessários.

Ao invés, podemos desenvolver um pacote, contendo todas as funções necessárias para nossas análises. De outro modo, uma vez o pacote instalado e anexado ao caminho de busca, todas as nossas 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.

8.5 Utilizar as funções de um pacote

Uma vez instalado o pacote, precisamos carregar e anexá-lo, para utilizarmos todos recursos disponíveis, como funções, dentre outros objetos. Isso significa, disponibilizar na memória virtual de seu sistema operacional e inseri-lo ao caminho de busca, respectivamente. Para fazer essas duas ações ao mesmo tempo, usamos 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 8.1.

Código R 8.1

Script R:

```
1 # Carregando e anexando o pacote midrangeMCP
2 library(midrangeMCP)
3 #-----
4 # Dados simulados de um experimento em DIC
5 # (Delineamento Inteiramente Casualizado)
6 # Variavel resposta
7 rv <- c(
8   100.08, 105.66, 97.64, 100.11, 102.60,
9   121.29, 100.80, 99.11, 104.43, 122.18, 119.49,
10  124.37, 123.19, 134.16, 125.67, 128.88, 148.07,
11  134.27, 151.53, 127.31
12 )
13 # Tratamento
14 treat <- factor(rep(LETTERS[1:5], each = 4))
15 # Anava
16 res <- anova(aov(rv ~ treat))
17 DFerror <- res$Df[2]
18 MSerror <- res$`Mean Sq`[2]
19 # Aplicando testes
20 results <- midrangeMCP::MRtest(
21   y = rv,
22   trt = treat,
23   dferror = DFerror,
24   mserror = MSerror,
25   alpha = 0.05,
26   main = "PCMs",
27   MCP = c("all")
28 )
29 results
```

Console R:

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

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

Mean Grouping Midrange Test

Statistics:

Exp.Mean	CV	MSerror	Df	n	Stud.Mid	Ext.DMS	Int.DMS
118.542	7.08	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

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 R:

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 R:

Tukey Midrange Test

Statistics:

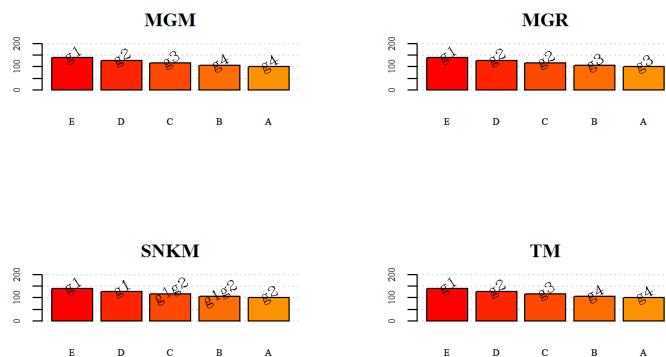
	Exp.Mean	CV	MSerror	Df	n	Stud.Mid	Ext.DMS	Int.DMS
	118.542	7.08	70.47488	15	5	1.089968	5.90246	4.575105

Groups:

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

Script R:

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



8.6 Carregando e anexando um pacote

Anteriormente, falamos que para utilizar as funções de um pacote, usamos as funções `library()` ou `require()`, e que nestas funções o nome dos pacotes podem ser inseridos entre aspas ou não. Carregar um pacote significa disponibilizá-lo na memória virtual. Para acessar uma função de um pacote após ter sido carregado, usamos o operador `::`, isto é, `nome_pacote::nome_função`. Assim, chamaremos a função necessária sem anexar o pacote ao caminho de busca. Estudaremos no Capítulo 9, um pouco mais sobre o caminho de busca. Para este momento, entendemos que é um caminho hierarquizado de ambientes que armazena os nomes dos objetos em forma de lista. A função para ver o caminho de busca é `search()`, como pode ser observado no Código R 8.2.

Código R 8.2

Script R:

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

Console R:

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

Simplificamos a representação package:<nome_pacote> por pack:<nome_pacote>.

Com as linhas de comando apresentadas no Código 8.3, percebemos que, ao executar a função `MRwrite()` do pacote **midrangeMCP** usando a função `::`, o caminho de busca não será alterado.

Código R 8.3**Script R:**

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

Console R:

```
\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}
```

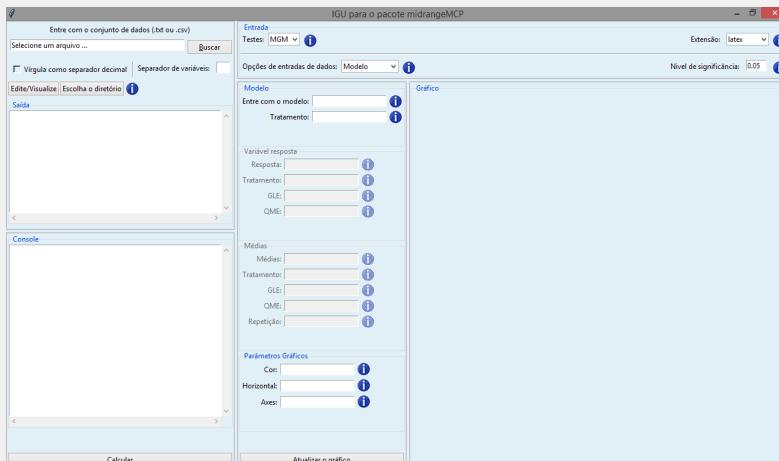
Observamos que o pacote não foi anexado, Código 8.3, apenas carregado. Se desejarmos utilizar alguma função do pacote digitando apenas o nome no *console*, não será possível, porque o pacote não está anexado. Vejamos outra situação pelo Código R 8.4.

Código R 8.4

Console R:

```
> search() # Caminho de busca
[1] ".GlobalEnv" "pack:magrittr" "pack:leaflet"
[4] "pack:stats" "pack:graphics" "pack:grDevices"
[7] "pack:utils" "pack:datasets" "pack:methods"
[10] "Autoloads" "pack:base"
> library(midrangeMCP) # Carregando e anexando um pacote
> search() # caminho de busca
[1] ".GlobalEnv" "pack:midrangeMCP" "pack:magrittr"
[4] "pack:leaflet" "pack:stats" "pack:graphics"
[7] "pack:grDevices" "pack:utils" "pack:datasets"
[10] "pack:methods" "Autoloads" "pack:base"
> guimidrangeMCP() # funcao do pacote midrangeMCP
```

Por questão de estética do código, simplificamos a representação dos ambientes de pacote de `pack:<nome_pacote>` para `pack:<nome_pacote>`.



Com o uso da função `library()`, Código R 8.4, percebemos que

o caminho de busca foi alterado, porque agora temos o ambiente de pacote `pack::midrangeMCP`. Isto significa que agora podemos acessar os objetos do pacote apenas digitando os nomes associados 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*).

Quando anexar ou carregar um pacote?



Anexaremos um pacote quando muitas funções são necessárias para o desenvolvimento de nossas rotinas. Caso contrário, optamos por carregar usando o operador `:::`. Essa decisão evita possíveis conflitos que possam existir entre funções devido a quantidade de pacotes anexados ao caminho de busca.

8.7 NAMESPACE de um pacote

No início do capítulo, falamos sobre o esqueleto de um pacote, isto é, os componentes básicos de um pacote. Um dos arquivos foi o arquivo *NAMESPACE*. Este 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 ou por meio do operador `::`. As funções importadas são aquelas utilizadas de outros pacotes, sendo necessárias apenas 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. Geralmente, funções internas passam por modificações, melhorias, etc., que por isso justifica a sua não exportação a nível de usuário. De outra 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, o padrão no *NAMESPACE* é o comando: `exportPattern("^[^\\\\.]")`, que significa que todas as funções no pacote serão exportadas que não iniciam por um ponto ('.') .

Funções iniciadas por “.” (ponto)



Todas as funções desenvolvidas em um pacote, cujos nomes iniciam por um ponto, são uma função interna não exportável em um pacote. Isso é um padrão, não uma regra sintática, pois se criarmos uma função nominada dessa forma, e desejarmos exportá-la, esta será, o que contraria apenas a convenção utilizada no ambiente R.

8.8 Documentações de ajuda de um pacote

Toda função exportada de um pacote precisa de um arquivo de ajuda (<>.Rd). As funções exportadas deverão ter esses tipos de arquivos inseridos no subdiretório *man/*. Mais detalhes sobre o desenvolvimento de pacotes serão abordados no *Volume V*, da coleção *Estudando o Ambiente R*.

8.9 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 a nível de usuário, usamos o operador “:::”. Vejamos um exemplo no Código R 8.5.

As funções internas dos pacotes devem ser utilizadas com muita cautela, uma vez que essas funções podem passar por atualizações, mudanças. Como já falamos anteriormente, não são funções exportadas. Alguns pacotes podem passar por atualizações, fazendo com que essas funções também podem ser atualizadas ou até mesmo alteradas. Com as mesmas justificativas anteriores, não recomendamos a utilização de importação de funções internas de outros pacotes no desenvolvimento de pacotes. 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 pacote, e por isso que elas são exportadas.

Código R 8.5**Console R:**

```
> # Carregando e chamando funções exportadas do pacote
   SMR
> SMR::pSMR(q = 2, size = 10, df = 3)
[1] 0.9905216
> # Carregando e chamando funções de SMR
> SMR::GaussLegendre(size = 4)
$nodes
[1] -0.8611363 -0.3399810  0.3399810  0.8611363
$weights
[1] 0.3478548  0.6521452  0.6521452  0.3478548
```

8.10 Exercícios

Exercício 8.1: Pesquise na página do pacote **SMR**: <https://cran.r-project.org/package=SMR> e verifique as suas informações básicas, como: autores, mantenedor, versão, descrição, etc.

Dica na página 299

Exercício 8.2:

Supondo que criamos a seguinte função:

Script R:

```
1 mean <- function(x, ...) "Não imprima nada"
```

Sabemos que o ambiente envolvente é o ambiente global. Agora, precisamos calcular a média do vetor 1:10. Para isso, iremos usar a função `mean()` do pacote **base** que computa a média de um conjunto de valores, isto é:

Console R:

```
> mean(1:10)
[1] "Não imprima nada"
```

O que ocorreu de errado? Como calculamos a média do vetor?

Dica na página 299

Exercício 8.3: Um dos testes muito utilizados na estatística é o teste *t de Student*. Utilizamos no **R** a função `t.test()`, do pacote nativo **stats**. Quando desejamos verificar como são realizados os cálculos, digitamos o nome da função para verificar o código interno, isto é:

Console R:

```
> t.test
function (x, ...)
UseMethod("t.test")
<bytecode: 0x000000223a79bdd8>
<environment: namespace:stats>
```

Percebemos que a saída não apresenta o código para os cálculos que realizamos do teste *t*. Isso ocorre porque a função é o que chamamos de genérico ou função genérica, uma estrutura no ambiente **R** específica para orientação a objetos com classes definidas. Estes genéricos despacham para métodos específicos, do qual, quando um objeto não tem uma classe definida, o despacho ocorre para um método padrão. No caso desse genérico, se `x` não tiver uma classe explícita, o despacho ocorrerá para o método (que é uma função) `t.test.default()`. Porém, esta função não é exportável, isto é, uma função interna do pacote **stats**. Como podemos acessá-la e verificar o código interno?

Dica na página 299

Exercício 8.4: Os primeiros pacotes submetidos ao *CRAN* nas primeiras versões do **R**, não tinham o arquivo **NAMESPACE**,

que hoje é exigido. Qualquer tentativa de instalação desses pacotes nessas situações, não seriam bem sucedidas nas versões atuais. Dessa forma, os pacotes que não se atualizam com as versões do R, por esse motivo ou por outros, podem se tornar órfãos, isto é, pacotes desativados. Neste problema específico, propomos como exercício a reativação do pacote **mlCopulaSelection**, baixando o arquivo <>.tar.gz na página <https://cran.r-project.org/src/contrib/Archive/mlCopulaSelection/>. O objetivo, é recriar um arquivo <>.tar.gz deste pacote, inserindo o arquivo *NAMESPACE* faltante, e necessário para a sua instalação. Posteriormente, instale-o. Alguns problemas adicionais nos arquivos de ajuda podem ocorrer, porém, a sua instalação será realizada.

Dica na página 299

Exercício 8.5: Instale o pacote **leem** a partir do **GitHub**.

Dica na página 299

Exercício 8.6: Baseado no Exercício 8.5, instale o pacote **leem** a partir do **CRAN**, e verifique quais diferenças nessas duas formas de instalação. Verifique também se houve alguma diferença na estrutura do pacote.

Dica na página 299

Exercício 8.7: O argumento `dependencies` da função `install.packages()`, permite com que além de instalar o pacote, instale as suas dependências. Pesquise no manual de ajuda da função para saber mais detalhes sobre esse argumento, e instale um pacote do seu interesse, instalando também as suas dependências.

Dica na página 300

Exercício 8.8:

Ainda sobre o Exercício 8.3, se verificarmos as últimas linhas da função interna `t.test.default()`, que segue:

Script R:

```
1 function(x, y = NULL,
2         alternative = c(
3             "two.sided",
4             "less",
5             "greater"
6         ),
7         mu = 0, paired = FALSE,
8         var.equal = FALSE,
9         conf.level = 0.95, ...){
10 # Códigos omitidos ...
11 rval <- list(
12     statistic = tstat,
13     parameter = df,
14     p.value = pval,
15     conf.int = cint,
16     estimate = estimate,
17     null.value = mu,
18     stderr = stderr,
19     alternative = alternative,
20     method = method,
21     data.name = dname
22 )
23 class(rval) <- "htest"
24 rval
25 }
```

o objeto `rval` é uma lista, porém com um atributo `class` igual a “`htest`”. Quando chamamos essa função `t.test()`, cujo argumento `x` não tem classe definida, o resultado pode ser observado no código na sequência:

Console R:

```
> set.seed(10) # Semente
> x <- rnorm(30, 100, 2); mu <- 90
> t.test(x = x, mu = mu)
One Sample t-test
t = 29.415, df = 29, p-value < 2.2e-16
alternative hypothesis: true mean is not equal to 90
95 percent confidence interval:
98.66327 99.95803
sample estimates:
mean of x
99.31065
```

Observamos que a saída não está nada parecido com uma saída de lista, que deveria ser:

Console R:

```
$statistic
t
29.41453

$parameter
df
29

Alguns resultados foram omitidos!

$method
[1] "One Sample t-test"

$data.name
[1] "x"
```

Por que isso ocorreu? Como justificar esse comportamento para a função `t.test()`?

Dica na página 300

Capítulo 9

Ambientes e caminho de busca

9.1 Introdução

No Capítulo 6, discutimos dois pontos interessantes sobre objetos: a atribuição e o escopo. Estes dois pontos estão intimamente relacionados ao objeto ambiente, de tipo “environment”. O ambiente é um objeto que armazena, em forma de lista, as ligações dos nomes associados aos objetos. Porém, existem diferenças entre o objeto lista e o objeto ambiente, com quatro exceções (WICKHAM, 2019):

- Cada nome deve ser único;
- Os nomes em um ambiente não são ordenados;
- Um ambiente tem um pai (ou também chamado de ambiente superior);
- Ambientes não são copiados quando modificados.

Muitas dessas definições são complexas para este momento. Aprofundaremos o assunto no *Volume II*. Contudo, introduziremos algumas características importantes para esse tipo de objeto.

9.2 Ambiente global

Como já mencionado em capítulos anteriores, o espaço de trabalho do **R** é conhecido como ambiente global, pois é onde todo o processo de interação da linguagem ocorre. Existe um nome específico associado a esse objeto: `.GlobalEnv`, mas que em outras situações recebe o nome `R_GlobalEnv`. O ambiente global também pode ser acessado

pela chamada de função `globalenv()`. Para sabermos quais os nomes existentes, usamos a função `ls()`, Código R 9.1.

Código R 9.1

Script R:

```
1 # Nomes no ambiente global
2 ls()
```

Console R:

```
character(0)
```

Quando o resultado da função é `character(0)`, significa que não existem nomes criados no ambiente global. O ambiente corrente é informado pela função `environment()`, Código R 9.2.

Código R 9.2

Script R:

```
1 # Comparando os ambientes
2 identical(environment(), .GlobalEnv)
```

Console R:

```
[1] TRUE
```

Script R:

```
1 # Forma errada de comparar ambientes (Erro...)
2 environment() == .GlobalEnv
```

Console R:

```
Error in environment() == .GlobalEnv: comparação (1) é
possível apenas para tipos lista ou atômicos
```

A segunda forma é equivocada no Código R 9.2, porque o operador *booleano*, “==”, é aplicado apenas a vetores atômicos ou listas, e o objeto ambiente não tem uma estrutura de vetor e também não é uma lista. Acrescentamos ainda que não utilizamos o sistema de indexação numerado (baseado na ordenação dos elementos), como pode ser visto no Código 9.3, *linha 9*.

Código R 9.3

Script R:

```
1 # Criando objetos no ambiente global
2 b <- 2; a <- "Ben"; x <- TRUE
3 # Verificando os nomes no ambiente global
4 ls()
```

Console R:

```
[1] "a" "b" "x"
```

Script R:

```
5 # Acessando o objeto "a"
6 .GlobalEnv$a
```

Console R:

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

Script R:

```
7 .GlobalEnv[["a"]]
```

Console R:

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

Script R:

```
8 # Acessando o primeiro nome (Erro...)
9 .GlobalEnv[[1]]
```

Console R:

Error in .GlobalEnv[[1]]: argumentos errados para
obtenção de subconjuntos de um ambiente

A última linha de comando, Código R 9.3 retorna um erro, porque os nomes em ambientes não são ordenados. Ao invés, devemos chamar os resultados por meio de função \$ ou [[<nome_obj>]].

9.3 Criando ambientes

A criação de um ambiente é dada pela função new.env(), e no Código R 9.4, verificamos como inserir ligações entre nomes e objetos.

Código R 9.4**Script R:**

```
1 # Criando objetos no ambiente global
2 b <- 2; a <- "Ben"; x <- TRUE
3 # Verificando os nomes no ambiente global
4 ls()
```

Console R:

```
[1] "a" "b" "x"
```

Script R:

```
5 # Criando um objeto ambiente no ambiente global
6 amb1 <- new.env()
7 # Inserindo nomes no ambiente "amb1"
8 amb1$d <- 3; amb1$e <- FALSE
9 # Verificando nomes no ambiente global
10 ls()
```

Console R:

```
[1] "a"     "amb1"  "b"     "x"
```

Script R:

```
11 # Verificando nomes no ambiente "amb1"
12 ls(envir = amb1)
```

Console R:

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

9.4 Ambiente “pai” (ou ambiente superior)

Todo ambiente tem um ambiente pai (ou também chamado de ambiente superior). Quando um nome não é encontrado no ambiente corrente, o R procurará no ambiente pai. Para sabermos como determinar o ambiente superior de um determinado ambiente, usamos a função `parent.env()`, Código R 9.5.

Código R 9.5**Script R:**

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

Console R:

```
<environment: R_GlobalEnv>
```

Para criarmos um determinado ambiente, cujo pai seja um ambiente específico, usamos a função `new.env()`, em que o argumento `parent` define o ambiente superior. O padrão é o ambiente corrente, representado pela função `parent.frame()`. Podemos também usar a função ``parent.env<-`()` para redefinir o ambiente superior de outro ambiente após a sua criação. Para elucidar, iremos apresentar um exemplo no Código R 9.6.

Código R 9.6**Script R:**

```
1 # Criando dois ambientes
2 amb01 <- new.env()
3 amb02 <- new.env(parent = amb01)
4 # Estes dois ambientes tem pai o ambiente pai
5 parent.env(amb01); parent.env(amb02)
```

Console R:

```
<environment: R_GlobalEnv>
<environment: 0x0000009fad75fbe8>
```

Script R:

```
6 # Endereço na memória de amb01
7 amb01
```

Console R:

```
<environment: 0x0000009fad75fbe8>
```

Script R:

```
8 # Mudando o ambiente superior
9 parent.env(amb01) <- emptyenv()
10 # Verificando novamente o ambiente superior
11 parent.env(amb01)
```

Console R:

```
<environment: R_EmptyEnv>
```

O único ambiente que não tem pai é o ambiente vazio, `R_EmptyEnv`, acessado pela função `emptyenv()`, que pode ser observado no Código R 9.7.

Código R 9.7

Script R:

```
1 parent.env(emptyenv())
```

Console R:

```
Error in parent.env(emptyenv()): o ambiente vazio não
tem pai
```

9.5 Operador superatribuição (“<<-”)

A atribuição (`<-`) é uma função que associa um nome a um objeto no ambiente corrente. Quando usamos o **R**, quase sempre este ambiente é o ambiente global. A superatribuição (`<<-`) cria um nome e o associa

a um objeto no ambiente pai do ambiente de onde essa associação foi criada. Vejamos o Código R 9.8.

Código R 9.8

Script R:

```
1 # Criando o objeto x e o imprimindo
2 x <- 0; x
```

Console R:

```
[1] 0
```

Script R:

```
3 # Criando uma função com a superatribuição
4 f1 <- function() {
5   # Obj2
6   x <- 1
7   # Modificando x do ambiente global
8   x <<- 2
9   # Imprimindo o ambiente de execução
10  env <- environment()
11  # Imprimindo o Obj2
12  res <- list(
13    x = x,
14    "Ambiente de execução" = env,
15    "Ambiente Pai" = parent.env(env)
16  )
17  # Retornando a lista
18  return(res)
19 }
20 # Imprimindo f1
21 f1()
```

Console R:

```
$x  
[1] 1  
  
$`Ambiente de execução`  
<environment: 0x0000000009b9a698>  
  
$`Ambiente Pai`  
<environment: R_GlobalEnv>
```

Script R:

```
18 # Imprimindo x  
19 x
```

Console R:

```
[1] 2
```

Script R:

```
20 # Imprimindo o ambiente envolvente de f1  
21 environment(f1)
```

Console R:

```
<environment: R_GlobalEnv>
```

Script R:

```
22 # Imprimindo os nomes do ambiente global  
23 ls()
```

Console R:

```
[1] "x"  "f1"
```

No Código 9.7, temos um caso interessante, porque vemos o mesmo nome (`x`), associado a objetos diferentes, em ambientes diferentes. Alguns ambientes são criados após a chamada da função `function()`, os chamados **ambientes funcionais**. Um deles é o ambiente envolvente, já falado no Capítulo 6. O ambiente envolvente da função `f1()` é o ambiente global. Já no corpo da função `f1()`, um outro ambiente surge quando a função é chamada, é o ambiente de execução. Vamos observar os endereços na memória, no ambiente de execução, quando executamos a função mais de uma vez, Código R 9.9.

Código R 9.9

Script R:

```
1 f1()$`Ambiente de execução`
```

Console R:

```
<environment: 0x0000000028ea9b38>
```

Script R:

```
2 f1()$`Ambiente de execução`
```

Console R:

```
<environment: 0x000000002b2f74d8>
```

Script R:

```
3 f1()$`Ambiente de execução`
```

Console R:

```
<environment: 0x000000002b04bd60>
```

Toda vez que a função `f1()` é chamada, Código 9.8, um novo ambiente de execução é criado, pois os endereços na memória são diferen-

tes. Isso significa que estes objetos foram armazenados em espaços de memória diferentes, e que portanto, são objetos diferentes.

Retornando ao Código R 9.7 e ao operador superatribuição, percebemos que o ambiente pai do ambiente de execução, é o ambiente envolvente de `f1()`, que é o ambiente global. Assim, ao chamarmos `f1()`, observamos a seguinte situação semântica para o operador superatribuição: o nome `x` no ambiente global passou a estar associado ao valor 2, porque foi alterado por `<<-`, mas o nome `x`, no ambiente de execução de `f1()`, continuou associado ao valor 1, porque a função retornou o mesmo resultado. Isso mostra que a superatribuição não associa um nome a um objeto no ambiente atual, mas em um ambiente pai, se não existir, ou altera o objeto com o nome já existente. Vejamos o complemento do que acabamos de afirmar no próximo exemplo no Código 9.10.

Código R 9.10

Script R:

```
1 # Verificando os nomes no ambiente global
2 ls()
```

Console R:

```
character(0)
```

Script R:

```
3 # Criando uma funcao
4 f2 <- function() {
5   x <<- 2
6 }
7 # Executando f2
8 f2()
```

Script R:

```
9 # Verificando novamente os nomes no ambiente global  
10 ls()
```

Console R:

```
[1] "f2" "x"
```

Script R:

```
11 # Verificando o valor de x  
12 x
```

Console R:

```
[1] 2
```

Percebemos que a superatribuição executada dentro de f2() criou o nome x no ambiente pai e associou ao valor 2, como pode ser observado na execução da *linha 10* do Código 9.10. Em um próximo exemplo, consideraremos um ambiente envolvente que não seja o ambiente global, Código R 9.11.

Pelo Código 9.11, quando uma função é criada dentro de outra, o ambiente de execução da função superior, contador(), em nosso exemplo é o ambiente envolvente da função interna, aux(). Dessa forma, o ambiente de execução de contador() não será mais efêmero, isto é, não será apagado após a execução, como pode ser visto na chamada de contador1(), *linhas 14-16*. Observamos que executamos contador1() três vezes. O objeto associado ao nome i foi atualizado, devido a superatribuição, a cada chamada da mesma função. Ao passo que, quando realizamos uma nova chamada de contador(), por meio de contador2(), o resultado de i retorna o valor 1, porque um novo ambiente de execução para contador() foi criado, como observado.

Código R 9.11**Script R:**

```

1 # Funcao contador
2 contador <- function() {
3   i <- 0
4   env1 <- environment()
5   aux <- function() {
6     i <- i + 1
7     env2 <- environment()
8     res2 <- list(
9       i = i,
10      `^AmbExec_aux` = env2,
11      `^AmbExec_contador` = env1
12    )
13   return(res2)
14 }
15 }
16 # Chamada de funcao
17 contador1 <- contador()
18 contador1()

```

Console R:

```

$i
[1] 1

$AmbExec_aux
<environment: 0x000000002ae7d6e0>

$AmbExec_contador
<environment: 0x000000002af0ea08>

```

Script R:

```
15 contador1()
```

Console R:

```
$i  
[1] 2  
  
$AmbExec_aux  
<environment: 0x00000002adca7d0>  
  
$AmbExec_contador  
<environment: 0x00000002af0ea08>
```

Script R:

```
16 contador1()
```

Console R:

```
$i  
[1] 3  
  
$AmbExec_aux  
<environment: 0x00000002ad1b2d8>  
  
$AmbExec_contador  
<environment: 0x00000002af0ea08>
```

Script R:

```
17 # Chamada de funcao  
18 contador2 <- contador()  
19 contador2()
```

Console R:

```
$i
[1] 1

$AmbExec_aux
<environment: 0x000000000647ce58>

$AmbExec_contador
<environment: 0x000000002ab3add8>
```

9.6 Caminho de busca

Por fim, o **R** poderá encontrar os nomes associados a objetos pelo caminho de busca. Além dos ambientes criados e o ambiente global, existem os ambientes de pacotes. Toda vez que um pacote for anexado ao caminho de busca, o ambiente de pacote anexado será sempre o pai do ambiente global, Código R 9.12.

Código R 9.12**Script R:**

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

Console R:

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

Script R:

```
3 # Anexando o pacote SMR
4 library(SMR)
5 # Verificando o caminho de busca
6 search()
```

Console R:

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

Script R:

```
7 # Carregando o pacote midrangeMCP
8 library(midrangeMCP)
9 # Verificando o caminho de busca
10 search()
```

Console R:

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

Por questão de estética do código, simplificamos a representação dos ambientes de pacote de `package:<nome_pacote>` para `pack:<nome_pacote>`.

A lista dos ambientes no caminho de busca segue a ordem hierárquica dos ambientes, de modo que o ambiente global será sempre o espaço de trabalho, isto é, o ambiente corrente. Não foi apresentado na lista o ambiente vazio, mas que pode ser observado com o pacote `rlang` no Código R 9.13.

Código R 9.13**Script R:**

```
1 # Criando um ambiente
2 amb2 <- new.env()
3 # Verificando seus parentais
4 rlang::env_parents(env = amb2, last = emptyenv())
```

Console R:

```
[[1]] $ <env: global>
[[2]] $ <env: package:midrangeMCP>
[[3]] $ <env: package:SMR>
[[4]] $ <env: package:magrittr>
[[5]] $ <env: package:leaflet>
[[6]] $ <env: package:stats>
[[7]] $ <env: package:graphics>
[[8]] $ <env: package:grDevices>
[[9]] $ <env: package:utils>
[[10]] $ <env: package:datasets>
[[11]] $ <env: package:methods>
[[12]] $ <env: Autoloads>
[[13]] $ <env: package:base>
[[14]] $ <env: empty>
```

Dessa forma, é pelo caminho de busca que o **R** procurará os nomes. Se o ambiente envolvente de uma função, por exemplo, for o ambiente vazio, o **R** procurará pelas funções básicas no pacote nativo **base** e não será encontrado, pois no ambiente vazio não há nomes e nem ambientes parentais. Por isso que o **R** depende do escopo léxico para tudo, Código R 9.14.

No Código R 9.14, usamos o pacote **codetools** como recurso de verificar as dependências de funções usadas internamente, usadas de outros pacotes, para o desenvolvimento da função `f3()`. Quando redefinimos o ambiente envolvente da função como sendo o ambiente vazio, executamos a função de `f3()` e percebemos que funções básicas, como o operador soma, que são executadas facilmente no espaço de trabalho, e retornam um erro, informando que não existe este ope-

rador. Acontece que pelo caminho de busca, não há ambiente pai do ambiente vazio, e muito menos nomes associados a objetos, portanto, não existe função “+”, necessária internamente em `f3()`, e assim, há o retorno do erro.

Código R 9.14

Script R:

```
1 # Criando uma função
2 f3 <- function() x + 1
3 # Modificando o ambiente envolvente de f3
4 environment(f3) <- emptyenv()
5 # Dependências externas da função f3
6 codetools::findGlobals(f3)
```

Console R:

```
[1] "+" "x"
```

Script R:

```
7 # Chamando a função f3
8 f3()
```

Console R:

```
Error in x + 1: não foi possível encontrar a função "+"
```

9.6.1 Função `attach()` e o caminho de busca

Em capítulos anteriores, discutimos sobre os recursos que a função `attach()` pode nos fornecer para acessar mais facilmente os elementos de listas. Porém, devido ao caminho de busca, essa condição semântica pode ser tornar um problema e deve ser utilizada com atenção. Vejamos o Código R 9.15.

Código R 9.15**Script R:**

```

1 # objeto quadro de dados
2 dados <- data.frame(sd = 1:3, var = (1:3) ^ 2)
3 # Caminho de busca
4 search()

```

Console R:

```

[1] ".GlobalEnv" "pack:midrangeMCP" "pack:SMR"
[4] "pack:magrittr" "pack:leaflet" "pack:stats"
[7] "pack:graphics" "pack:grDevices" "pack:utils"
[10] "pack:datasets" "pack:methods" "Autoloads"
[13] "pack:base"

```

Script R:

```

5 # anexando "dados" ao caminho de busca
6 attach(dados)
7 # Verificando novamente o caminho de busca
8 search()

```

Console R:

```

[1] ".GlobalEnv" "dados" "pack:midrangeMCP"
[4] "pack:SMR" "pack:magrittr" "pack:leaflet"
[7] "pack:stats" "pack:graphics" "pack:grDevices"
[10] "pack:utils" "pack:datasets" "pack:methods"
[13] "Autoloads" "pack:base"

```

Script R:

```

9 # Imprimindo sd
10 sd

```

Console R:

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

Script R:

```
11 # Desanexando "dados"
12 detach(dados)
13 # Imprimindo sd
14 sd
```

Console R:

```
function (x, na.rm = FALSE)
sqrt(var(if (is.vector(x) || is.factor(x)) x else
as.double(x),
na.rm = na.rm))
<bytecode: 0x0000000029010c70>
<environment: namespace:stats>
```

Quando criamos o objeto dados, uma de suas colunas estava nomeada por sd, que também é o nome de uma função do pacote nativo **stats**, que representa o desvio padrão. Porém, quando anexamos o objeto dados ao caminho de busca, um novo ambiente é criado, e será o pai do ambiente global. O ambiente criado receberá o mesmo nome do objeto, e os nomes dos elementos dados são copiados para este ambiente. Assim, o nome sd foi procurado e não encontrado no ambiente corrente (ambiente global), seguindo a busca do nome para o ambiente pai (dados), o nome é encontrado. Percebemos que em termos de ordenamento, o ambiente dados está na frente do ambiente de pacote package:stats, que também existe contém o nome sd. Assim, ao imprimir o objeto associado com sd no *console*, o resultado está relacionado com o objeto no ambiente dados, e não a função que computa o desvio padrão, vinculada no ambiente de pacote package:stats. Nesses casos, se usarmos a superatribuição, a alteração ocorrerá apenas na cópia dos elementos no ambiente anexado, e não nos elementos do objeto original. Caso haja a atribuição, o nome será criado no ambiente

global.

Ainda no Código R 9.15, quando desanexamos o objeto dados, e chamamos `sd` novamente, verificamos que é impresso o código da função que computa o desvio padrão, pois já não mais existe o ambiente dados.

Se muitos pacotes ou objetos forem anexados, nos deparamos, frequentemente, com esses problemas. Por isso, é preferível o uso da indexação ou `$` para acessar os elementos de uma lista, ou o operador `::` no caso de acessar objetos de pacotes, evitando assim, conflitos na procura de nomes.

Contudo, internamente em um pacote, é devido ao ambiente *nAMESPACE* do pacote que esses problemas não ocorrem. Devido a complexidade de entendimento para este momento, deixaremos para os demais *Volumes* a sua abordagem.

9.7 Exercícios

Exercício 9.1:

Observamos as seguintes linhas de comando:

Console R:

```
> # objeto quadro de dados
> dados <- list(sd = function(x) "Nada", var = (1:3)^2)
> # quebrando dados (attach)
> attach(dados)
> # Calculando o desvio padrao
> sd(1:10) # Ops...
[1] "Nada"
```

Mesmo com o `attach()` em `dados`, como poderíamos calcular o desvio padrão, função do pacote **stats**?

Dica na página 301

Exercício 9.2:

Parece incompreensível, mas um ambiente pode se conter no ambiente **R**. Pensando nisso, crie um ambiente, e faça com que ele esteja contido nele mesmo.

Dica na página 301

Exercício 9.3: No Capítulo 8, quando falamos sobre funções, introduzimos diversos ambientes que foram complementados neste momento. Portanto, elabore uma rotina que seja possível identificar os ambientes: de chamada, envolvente e de execução.

Dica na página 301

Exercício 9.4:

Baseado nas seguintes linhas de comando:

Console R:

```
> # objeto quadro de dados
> obj <- data.frame(x = 1, y = 2)
> # quebrando dados (attach)
> attach(obj)
> # Funcao aux
> aux <- function(...) {
+   x <- "Nada"
+ }
> # Chamando aux()
> aux()
> # O que ocorreu com os dois resultados?
> x
[1] "Nada"
> obj$x
[1] 1
```

O que podemos explicar sobre o comportamento semântico da superatribuição (<-)? E por que os resultados x e obj\$x foram diferentes?

Dica na página 301

Exercício 9.5:

Vejamos o script:

Script R:

```

1 # Ambiente a
2 a <- new.env()
3 # Função aux
4 aux <- function() x
5 # Criando um objeto
6 x <- 2
7 # Chamando aux
8 aux()
9 # Criando um obj em a
10 a$x <- 20
11 # Chamando aux
12 aux()
13 # Modificando o ambiente envolvente de aux
14 environment(aux) <- a
15 # Chamando aux
16 aux()
17 # Criando aux2
18 aux2 <- function() x <-- 10
19 # Mudando o ambiente envolvente
20 environment(aux2) <- a
21 # Chamando aux2
22 aux2()
23 # Vejamos os objetos
24 x
25 a$x

```

Por que ao chamarmos `aux()`, *linha 12*, o resultado não sera o valor `20`? Por que na *linha 16*, o resultado da chamada `aux()` se modificará para `20`? O que ocorre com os resultados dos objetos nas *linhas 24-25*?

Dica na página 301

Exercício 9.6: Um dos problemas no desenvolvimento de pacotes, é a interferência nas “variáveis globais” do ambiente

R, e isso é uma das políticas da linguagem, acessada em <https://cran.r-project.org/web/packages/policies.html>. Isto é, pacotes não podem alterar o ambiente global com a sua instalação. Porém, em algumas situações se faz necessário a criação de objetos no ambiente global, quando desenvolvemos pacotes. Um exemplo, são os elementos gráficos quando criamos nossas *GUI* por meio da linguagem **Tcl/Tk**, usando as funções do pacote **tcltk**. Pensando nisso, apresente soluções de como poderíamos resolver o problema, isto é, criar objetos acessados no ambiente global, sem alterá-lo.

Dica na página 301

Exercício 9.7: Observamos o seguinte erro na chamada `f()`, no *console* a seguir:

Console R:

```
> f <- function() 1 / 1
> environment(f) <- emptyenv()
> f()
Error in 1 / 1 : não foi possível encontrar a função "/"
```

Uma das funções mais básicas no **R** é o operador soma, e ele não foi encontrado. Por que isso ocorreu? Explique.

Dica na página 301

Capítulo 10

Interfaces com outras linguagens

10.1 Introdução

Como afirmado no terceiro princípio do **R** (CHAMBERS, 2016):

- **Princípio da Interface:** Interfaces para outros programas são parte do **R**.

Interface em linguagem de programação



Para o nosso contexto, vamos entender por interface como um intermediador entre duas ou mais linguagens, estabelecendo uma forma de interação entre elas. Contudo, sabemos que a ideia sobre interface em programação é mais amplo do que acabamos de mencionar.

Nos pacotes nativos do **R**, temos integrações prontas para implementar códigos em C e FORTRAN, como também outras linguagens. Por exemplo, no **R** existe um pacote chamado **tcltk** que integra a linguagem Tcl/Tk para o **R**. Este pacote nos permite desenvolver interfaces gráficas para os nossos códigos, de modo que ao utilizar não seja necessário ter contato direto com a implementação em linguagem **R**, mas com elementos gráficos como: botões, menus, etc.. Isto isola e protege possíveis erros que o usuário pode cometer com a aplicação.

Sabemos que o **R** é uma linguagem interpretada, e nenhum código nessa linguagem será compilado em código de máquina ou compilado diretamente nesse nível, e portanto, mesmo em um *script* básico para um código **R**, o usuário ainda estará usando interfaces, como observado no Código R 10.1.

Código R 10.1**Script R:**

```
1 # Operacao basica
2 pi / 2
```

Ao executarmos essa linha de comando, chamamos:

Console R:

```
> `/`
function (e1, e2) .Primitive("/")
```

Observamos que a chamada da função ` `/ `() para realizar a operações entre os dois valores, é uma interface primitiva. Como já falado no Capítulo 6, são funções implementadas em C, em sua grande maioria. Dessa forma, para não precisarmos chamar a função diretamente, usamos a interface ` `/ `().

Diversos outros pacotes, disponibilizados sob o CRAN realizaram diversas outras linguagens, que elencamos alguns na Tabela 10.1.

Tabela 10.1: Algumas interfaces **R**.

Pacote	Linguagem integrada
tcltk	Linguagem Tcl/Tk
RGtk2 (Desativado)	Linguagem Gtk+
rJava	Linguagem Java
rmarkdown	Linguagens HTML, JavaScript, CSS, Markdown, L ^A T _E X
reticulate	Linguagem Python
JuliaCall, XRJulia	Linguagem Julia
Rcpp	Linguagem C/C++
gecoder	Linguagem Ruby

Existe uma interface *web* bem interessante que integra as linguagens Julia, Python e **R**, chamado **Jupyter**, bem como o **RStudio**. Muitas outras interfaces são integradas ao **R**, que podem ser pesquisadas pela

internet.

Como motivação para este momento, iremos realizar três aplicações com as linguagens Python, C++ e Tcl/Tk para verificarmos que essas integrações são fáceis de serem realizadas, dentro de um conhecimento básico sobre tais linguagens. Descreveremos um estudo aprofundado sobre interfaces no **R**, no *Volume III*, da coleção *Estudando o Ambiente R*.

10.2 Implementação em Python

Inicialmente, faremos a instalação e anexaremos o pacote **reticulate**:

Console R:

```
> install.packages(reticulate)
> library(reticulate)
```

Existem diversas formas de integração. Vamos pensar na forma mais básica, da qual temos um *script* em Python, com a extensão <>.py, e na sequência, vamos carregá-lo no **R**. Chamaremos o arquivo de add.py, que segue o código interno:

Script Python:

```
1 def add(x, y):
2     return x + y
```

Vamos agora chamar a função add() em um *script R*, Código R 10.2.

Código R 10.2

Script R:

```
1 # Carregando o script add.py
2 reticulate::source_python("add.py")
3 add(5, 10) # Chamando a função add
```

Console R:

```
[1] 15
```

10.3 Implementação em C/C++

Inicialmente, faremos a instalação e anexaremos o pacote **Rcpp**:

Console R:

```
> install.packages(Rcpp)
```

Vamos realizar a mesma aplicação feita em Python, que segue o *script C/C++*, com extensão <>.cpp. Chamaremos o arquivo de add.cpp, que segue o código interno:

Script C/C++:

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3 // [[Rcpp::export]]
4 int add(int x, int y) {
5     return x + y;
6 }
```

Vamos agora chamar a função add() em um *script R*, Código R 10.3.

Código R 10.3

Script R:

```
1 # Carregando o script add.cpp
2 Rcpp::sourceCpp("add.cpp")
3 # Chamando a função add
4 add(5, 10)
```

Console R:

```
[1] 15
```

10.4 Implementação em Tcl/Tk

No caso da implementação em Tcl/Tk, não precisaremos instalar pacotes, pois o pacote nativo **tcltk** está disponível com a instalação do **R**. Ao invés, anexamos ao caminho de busca, usamos `library(tcltk)`. Iremos implementar uma interface gráfica contendo uma janela com entradas de dois números e um botão para calcular a soma, que segue:

Script R:

```

1 # Anexando o pacote
2 library(tcltk)
3 # Janela principal
4 main <- tkoplevel(width = 400, height = 300)
5 tkpack.propagate(main, FALSE)
6 # Texto inicial
7 tkpack(tklabel(main,
8   text = "Soma de dois números inteiros"
9  ))
10 # Quadro 1
11 tkpack(q1 <- tkframe(main), side = "top")
12 tkpack(
13   tklabel(
14     q1,
15     text = "Insira o primeiro número: ",
16     padx = 3
17   ),
18   side = "left", anchor = "e"
19 )
20 # Entrada 1

```

```
21 var1 <- tclVar("Insira um número inteiro")
22 tkpack(
23   entry1 <- tkentry(q1, textvariable = var1, width = 25),
24   side = "left", anchor = "ne"
25 )
26 # Quadro 2
27 tkpack(q2 <- tkframe(main), side = "top")
28 tkpack(
29   tklabel(q2,
30     text = "Insira o segundo número: ", padx = 3
31   ),
32   side = "left", anchor = "e"
33 )
34 # Entrada 2
35 var2 <- tclVar("Insira um número inteiro")
36 tkpack(
37   entry2 <- tkentry(q2,
38     textvariable = var2,
39     width = 25
40   ),
41   side = "left",
42   anchor = "ne"
43 )
44 # Botao
45 tkpack(
46   botao <- tkbutton(main, text = "Somar dois números"),
47   side = "top"
48 )
49 # Funcao auxiliar
50 f1 <- function(...) {
51   if ((is.na(as.numeric(tclvalue(var1))) |
52     is.na(as.numeric(tclvalue(var2))))) {
53     tkpack(
54       tklabel(main,
55         text = "Insira Valores numéricos!"
56       ),
57       side = "top"
```

```
58     )
59 } else {
60   res <- as.numeric(tclvalue(var1)) +
61   as.numeric(tclvalue(var2))
62   tkpack(
63     tklabel(
64       main,
65       text = paste("A soma é igual a ", res)
66     ),
67     side = "top"
68   )
69 }
70 }
71 # Acao ao Botao
72 tkbind(botao, "<ButtonRelease>", f1)
```

O resultado é a interface apresentada na Figura 10.1. Condições mais complexas poderão existir, e é imprescindível o entendimento de como as funções desses pacotes trabalham. Em algumas situações, é necessário entender a linguagem da qual se deseja integrar com o **R**. Falamos isso porque, por exemplo, o manual de ajuda do pacote **tcltk** é muito pobre em exemplos, e se faz necessário estudar a própria linguagem para entender como interagir com as funções que integram as duas linguagens (interfaces). Por fim, esse princípio demonstra o real alcance que o ambiente **R** pode ter com outras linguagens, e desse modo, podemos trabalhar conjuntamente de acordo com a demanda necessária para as nossas implementações e problemas a serem estudados, para a devida solução.

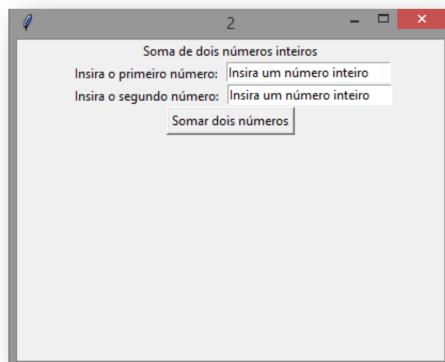


Figura 10.1: Interface **R** com a linguagem Tcl/Tk.

10.5 Exercícios

Exercício 10.1: Para desejarmos saber sobre funções escritas em baixo nível nos pacotes nativos do **R**, podemos usar a função `pryr::show_c_source()`. Por exemplo, a função ``[[-`()` é uma função primitiva, e buscamos detalhes sobre seu código da seguinte forma: `pryr::show_c_source(.Primitive("[["))`. Pesquise sobre mais funções primitivas e verifique seu código interno.

Dica na página 302

Exercício 10.2: Quais outros pacotes não documentados neste capítulo são interfaces para outras linguagens no ambiente **R**?

Dica na página 302

Exercício 10.3: A ideia sobre interface em programação tem um sentido mais amplo do que foi apresentado neste capítulo. Apresente-a com exemplos no ambiente **R**.

Dica na página 302

Exercício 10.4: De acordo com o Código C, apresentado a seguir:

Script C/C++:

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3 // [[Rcpp::export]]
4 double div(int x, int y) {
5     return x / y;
6 }
```

compile o código, fazendo com que a função div() seja reconhecida em linguagem R.

Dica na página 302

Capítulo 11

Considerações e preparação para o Volume II

11.1 Introdução

Ao final de tudo o que discutimos, tentamos tratar as ideias básicas por trás dos três princípios do **R**, sem enfatizar profundamente o terceiro princípio, mas principalmente os fundamentos de um objeto e a ideia sobre função. Tentamos repassar algo mais técnico sobre esta linguagem. Contudo, sabemos que algo introdutório deve ter uma flexibilidade quanto a profundidade das discussões e exemplos abordados. Mesmo assim, este *Volume I* deu suporte a problemas mais complexos, que serão mais bem explorados nos dois *Volumes* seguintes (*Volume II* e *III*).

Faremos constantemente atualizações, quando necessárias, para melhorar este material, e assim, auxiliar nos estudos de quem se interessar em estudar a linguagem **R**. Para isso, pedimos também a contribuição como leitor. Caso encontrem alguma discordância neste material ou complementos sobre a discussão, entre em contato pelas nossas redes sociais, como também por email: ben.deivide@gmail.com.

Na Figura 11.1, apresentamos um resumo do que vem para o *Volume II*. Neste *Volume*, teremos o objetivo de detalhar o ambiente **R**, caracterizando melhor a sua linguagem, aprofundando **manipulações de objetos, cópias de objetos**, bem como **ambientes**. Complementaremos também sobre **programação funcional, programação orientada a objetos, metaprogramação** e uma introdução sobre **desenvolvimento de pacotes**.

Esperamos que a leitura tenha despertado as potencialidades que essa linguagem poderá proporcionar ao ambiente profissional, e que

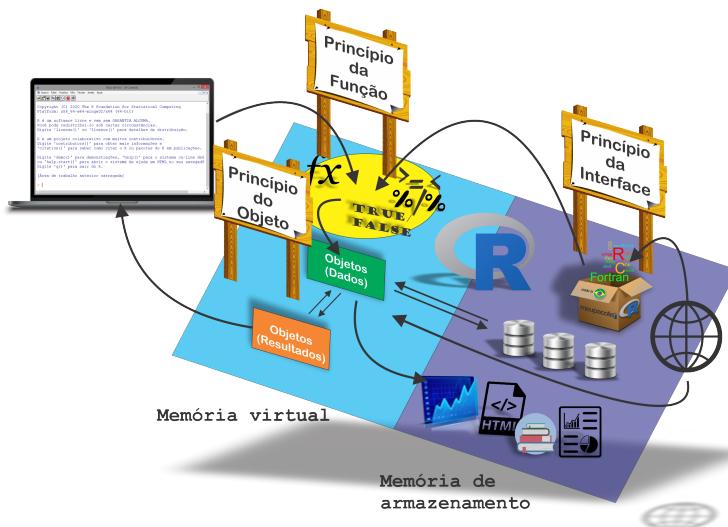


Figura 11.1: Relembrando como o **R** funciona.

nossas experiências documentadas nessa coleção sirvam de inspiração e consulta para o estudo. Saiba que é muito bom compartilhar algo que aprendemos, pois solidificamos a aprendizagem e reforçamos que **o conhecimento é uma liberdade necessária**.

Deixaremos alguns exercícios na sequência, para motivação sobre os assuntos que virão no *Volume II*. Bons estudos!

11.2 Exercícios

Exercício 11.1: O que representa uma função genérica no sistema S3 do paradigma da programação orientada a objetos no ambiente **R**?

Dica na página 303

Exercício 11.2: O que significa despacho de método no paradigma de orientação a objetos?

Dica na página 303

Exercício 11.3: Existem muitos ambientes na linguagem **R**. Qual a diferença entre o ambiente de ligação e ambiente envolvente? Como os ambientes interferem em nosso código?

Dica na página 303

Exercício 11.4: Como criamos métodos para os genéricos de grupos?

Dica na página 303

Exercício 11.5: Quais outras estruturas de dados existem no ambiente **R**?

Dica na página 303

Exercício 11.6: Por que o ambiente **R** não é uma linguagem estritamente funcional? Apresente exemplos.

Dica na página 303

Exercício 11.7: De acordo com o código:

Console R:

```
> faux <- function(x = 1, y = faux2()) x  
> faux()  
[1] 1
```

por que a função `faux()` ao ser executada não apresentou um erro, já que em um de seus argumentos apresentava a função `faux2()` que não existe no ambiente global?

Dica na página 303

Dica dos Exercícios

Dica dos Exercícios do Capítulo 1

Dica do Exercício 1.1 na página 8:

Como sugestão, recomendamos a leitura do Capítulo 8.

Dica do Exercício 1.2 na página 8:

Como sugestão, recomendamos a leitura do Capítulo 9 .

Dica do Exercício 1.3 na página 8:

Como sugestão, recomendamos a leitura do Capítulo 8.

Dica do Exercício 1.4 na página 8:

Como sugestão, recomendamos a leitura do Capítulo 8.

Dica do Exercício 1.5 na página 8:

Se a resposta foi 1, recomendamos ler todo o livro.

Dica do Exercício 1.6 na página 9:

Recomendamos ler todo livro.

Dica do Exercício 1.7 na página 9:

Como sugestão, recomendamos a leitura do Capítulo 3.

Dica do Exercício 1.8 na página 9:

Como sugestão, recomendamos a leitura do Capítulo 6.

Dica do Exercício 1.9 na página 9:

Como sugestão, recomendamos a leitura dos Capítulos 6 e 9.

Dica do Exercício 1.10 na página 9:

Como sugestão, recomendamos a leitura dos Capítulos 6 e 9.

Dica do Exercício 1.11 na página 9:

Como sugestão, recomendamos a leitura do Capítulo 4.

Dica do Exercício 1.12 na página 9:

Como sugestão, recomendamos a leitura do Capítulo 5.

Dica do Exercício 1.13 na página 10:

Como sugestão, recomendamos a leitura do Capítulo 7.

Dica do Exercício 1.14 na página 10:

Como sugestão, recomendamos a leitura do Capítulo 2.

Dica do Exercício 1.15 na página 10:

Como sugestão, recomendamos a leitura do Capítulo 4.

Dica do Exercício 1.16 na página 10:

A resposta está relacionada ao escopo léxico das funções e a superatribuição! Ainda não entendeu? Então leia os Capítulos 2 e 6. Para uma maior profundidade, recomendamos o Capítulo 9 sobre ambientes.

Dica do Exercício 1.17 na página 11:

Como sugestão, recomendamos a leitura do Capítulo 4.

Dica do Exercício 1.18 na página 11:

Como sugestão, recomendamos a leitura do Capítulo 7.

Dica do Exercício 1.19 na página 11:

Como sugestão, recomendamos a leitura dos Capítulos 3 e 10.

Dica do Exercício 1.20 na página 11:

Como sugestão, recomendamos a leitura dos Capítulos 3 e 10.

Dica do Exercício 1.21 na página 12:

Como sugestão, recomendamos a leitura do Capítulo 4.

Dica do Exercício 1.22 na página 12:

Como sugestão, recomendamos a leitura dos Capítulos 3 e 5.

Dica do Exercício 1.23 na página 12:

O escopo é a forma como as funções procuram pelos nomes, e o ambiente é o objeto no **R** que armazena as ligações dos nomes associados aos objetos.

Dica do Exercício 1.24 na página 12:

Como sugestão, devemos criar um objeto orientado ao sistema S3 para uma determinada classe, por exemplo, e desenvolver um método do genérico `print()` para a referida classe. Posteriormente, ao imprimir o objeto, poderemos visualizar a condição implícita da função `print()`.

Dica do Exercício 1.25 na página 12:

Como sugestão, recomendamos a leitura do Capítulo 4.

Dica dos Exercícios do Capítulo 2

Dica do Exercício 2.1 na página 25:

John Chambers foi o principal criador da linguagem S, e toda a base programada para a área da estatística no R está fundamentada nessa linguagem, bem como outras ferramentas computacionais. Observamos nos manuais de ajuda dos objetos em R , principalmente os do pacote **base**, que as referências para a implementação usam em grande parte são os livros do John Chambers, vulgarmente chamados de livros branco, azul e verde.

Dica do Exercício 2.2 na página 25:

Como sugestão, podemos pesquisar em <https://www.r-project.org/about.html>, para complementar as informações contidas neste livro.

Dica do Exercício 2.3 na página 25:

Podemos realizar uma pesquisa nas páginas de buscas do tipo: “R programming ide”, “IDE for R”, que será encontrado muitas outras opções.

Dica do Exercício 2.4 na página 25:

Como sugestão, recomendamos a leitura do Capítulo 2, Seção 2.2, e Capítulo 9.

Dica do Exercício 2.5 na página 25:

Como sugestão, recomendamos a leitura do Código 2.4.

Dica do Exercício 2.6 na página 25:

Como sugestão, recomendamos a leitura do Capítulo 2, Seções 2.2 e 2.3.

Dica do Exercício 2.7 na página 25:

Informações iniciais se encontram no início do referente ca-

pítulo, porém podemos pesquisar também em <https://www.r-project.org/about.html>.

Dica do Exercício 2.8 na página 26:

Como sugestão, recomendamos a leitura do Capítulo 2, Seção 2.1.

Dica do Exercício 2.9 na página 26:

Além da leitura inicial deste capítulo, podemos pesquisar em <https://cran.r-project.org/>.

Dica do Exercício 2.10 na página 26:

Terceiro princípio do R: Princípio da interface.

Dica dos Exercícios do Capítulo 3

Dica do Exercício 3.1 na página 41:

Como sugestão, recomendamos a leitura sobre ambientes.

Dica do Exercício 3.2 na página 42:

Como sugestão, recomendamos a leitura sobre execução de comandos.

Dica do Exercício 3.3 na página 42:

Como sugestão, recomendamos a leitura sobre como usar o **R** e **RStudio**.

Dica do Exercício 3.4 na página 42:

Como sugestão, recomendamos a leitura na subseção sobre *console* e *prompt* de comando, deste capítulo.

Dica do Exercício 3.5 na página 42:

Como sugestão, recomendamos a leitura da seção sobre ambiente global e diretório de trabalho, do referido capítulo. Mais detalhes pode ser encontrado no Capítulo 9.

Dica do Exercício 3.6 na página 42:

Como sugestão, recomendamos a leitura sobre a seção sobre comandos elementares.

Dica do Exercício 3.7 na página 42:

Como sugestão, recomendamos a leitura sobre comandos elementares do referido capítulo.

Dica do Exercício 3.8 na página 42:

Como sugestão, recomendamos o uso das funções `setwd()` e `getwd()`.

Dica do Exercício 3.9 na página 42:

Neste Capítulo, apresentamos algumas ideias sobre algumas práticas de uma boa escrita de código. Mais detalhes podem ser vistos no Capítulo 7.

Dica do Exercício 3.10 na página 42:

Podemos pesquisar no próprio ambiente R com a chamada ?Reserved.

Dica do Exercício 3.11 na página 43:

Como sugestão, recomendamos a leitura da Seção sobre ambiente global e diretório de trabalho.

Dica dos Exercícios do Capítulo 4

Dica do Exercício 4.1 na página 122:

Como sugestão, recomendamos a leitura sobre os *arrays*.

Dica do Exercício 4.2 na página 122:

Como sugestão, recomendamos a leitura sobre os sistemas de indexação.

Dica do Exercício 4.3 na página 122:

Como sugestão, recomendamos a leitura sobre o atributo `class` e quadro de dados.

Dica do Exercício 4.4 na página 122:

Como sugestão, podemos usar a função `typeof()`!

Dica do Exercício 4.5 na página 122:

Podemos usar as funções `is.atomic()` e `is.recursive()`, respectivamente, para identificar se um objeto é atômico ou recursivo.

Dica do Exercício 4.6 na página 122:

Podemos usar o objeto `.Last.value`!

Dica do Exercício 4.7 na página 123:

Como sugestão, recomendamos a leitura sobre sistema de indexação, funções `cbind()`, `names()`.

Dica do Exercício 4.8 na página 123:

Como sugestão, podemos usar o sistema de indexação para lista!

Dica do Exercício 4.9 na página 123:

O erro ocorre devido ao uso da função `attach()`! Como sugestão, recomendamos a leitura do Capítulo 9.

Dica do Exercício 4.10 na página 124:

Podemos usar as funções `typeof()` e `unclass()` para responder o referido exercício.

Dica do Exercício 4.11 na página 124:

Como sugestão, recomendamos estudar sobre a função `lobstr::obj_addr()` para inspecionar os objetos! Pesquise sobre **cópia e modificação no local** de objetos **R**.

Dica do Exercício 4.12 na página 124:

Isso identifica uma das diferenças entre as duas funções “`<-`” e “`=`”, que é a precedência superior de uma sobre a outra. Para mais, sugerimos executar a chamada `?assign0ps` no `console` para acessar os manuais de ajuda do **R** sobre atribuição.

Dica do Exercício 4.13 na página 125:

Como sugestão, recomendamos a leitura sobre cópia de objetos.

Dica do Exercício 4.14 na página 125:

Como sugestão, recomendamos a pesquisa sobre **contagem de referência**.

Dica do Exercício 4.15 na página 126:

Como sugestão, podemos usar o sistema de indexação auxiliado pelas funções `which()` e `max()`, para uma das soluções do problema.

Dica do Exercício 4.16 na página 126:

Podemos usar o operador `%%`.

Dica do Exercício 4.17 na página 126:

Como sugestão, recomendamos a leitura sobre *scripts*, .RData e .Rhistory.

Dica do Exercício 4.18 na página 126:

Como sugestão, recomendamos a utilização das funções como rep(), seq(), etc..

Dica do Exercício 4.19 na página 127:

Como sugestão, recomendamos as funções como: max() e min(), dentre outras.

Dica do Exercício 4.20 na página 127:

Como sugestão, podemos refletir sobre as ideias do primeiro princípio: “Tudo que existe no R é um objeto”!

Dica do Exercício 4.21 na página 127:

Como sugestão, recomendamos as funções como: length(), dim(), names(), etc.

Dica do Exercício 4.22 na página 127:

Essa questão é avançada para este capítulo. Portanto, sugerimos a leitura antecipada dos Capítulos 4 e 5 para um melhor entendimento do problema. Posteriormente, recomendamos utilizar a função source() para carregar linhas de comando de outros *scripts*, e para uma das possibilidades de solução do problema, podemos usar as funções da família *apply*.

Dica do Exercício 4.23 na página 127:

Sugerimos a pesquisa com a chamada ?assign0ps no *console* para mais detalhes.

Dica dos Exercícios do Capítulo 5

Dica do Exercício 5.1 na página 153:

Para verificar o tipo do objeto, podemos usar a função `typeof()`!

Dica do Exercício 5.2 na página 154:

Para exportar, podemos usar a função `write.table()`. Para a leitura dos dados, podemos usar a função `read.table()`. Lembrando que, para exportarmos os dados com separador de casas decimais sendo a vírgula, na função `read.table()`, devemos passar essa informação no argumento `sep = ", "`, para que haja a conversão para ponto, e o **R** consiga fazer a leitura correta dessas informações.

Dica do Exercício 5.3 na página 154:

Podemos usar os pacotes **readxl** e **writexl**, para a exportação e leitura dos dados, respectivamente.

Dica do Exercício 5.4 na página 155:

Para o pacote **openxlsx**, use as funções `openxlsx::read.xlsx()` e `openxlsx::write.xlsx()`. Estude os seus argumentos pelos documentos de ajuda. Para o pacote **xlsx**, use as funções `xlsx::read.xlsx()` e `xlsx::write.xlsx()`. Dúvidas sobre algumas dessas funções, podemos usar `?write.xlsx()`, para saber mais sobre `write.xlsx()`, por exemplo.

Dica do Exercício 5.5 na página 155:

Para pesquisar sobre **rmarkdown**, podemos acessar: <https://pkgs.rstudio.com/rmarkdown/>. Para pesquisar sobre o **shiny**, podemos acessar <https://shiny.rstudio.com/>. Por fim, para pesquisar sobre o `Sweave()`, podemos acessar os manuais de ajuda, isto é, `?Sweave()`, ou acessar <https://stat.ethz.ch/R-manual/R-devel/library/utils/doc/Sweave.pdf>

Dica do Exercício 5.6 na página 155:

Para o desenvolvimento de uma interface, sugerimos uma pesquisa nos *sites* de busca para a verificação de modelos e como ocorrem as suas implementações. É importante também a consulta nos manuais de ajuda dos pacotes indicados.

Dica do Exercício 5.7 na página 155:

Como sugestão, podemos acessar os manuais de ajuda do pacote **foreign**.

Dica do Exercício 5.8 na página 155:

Como sugestão, podemos acessar os manuais de ajuda das funções no pacote **base**.

Dica dos Exercícios do Capítulo 6

Dica do Exercício 6.1 na página 212:

Como sugestão, podemos usar a função `is.primitive()`.

Dica do Exercício 6.2 na página 212:

A forma como as funções primitivas foram desenvolvidas, em linguagem C, em sua maioria, segue uma estrutura diferente, que para este *Volume*, foge do escopo a sua explanação. Sugerimos também uma pesquisa com a chamada `? .Primitive`.

Dica do Exercício 6.3 na página 212:

O objetivo do exercício é identificar e entender chamada de funções intermediárias, e quando devemos utilizar o operador *pipe*.

Dica do Exercício 6.4 na página 212:

Sugerimos a função `%` para auxiliar na identificação se um número é par ou ímpar.

Dica do Exercício 6.5 na página 212:

Use as estruturas de controle, e as chamadas recursivas de código. Estude a função `Recall()` para obter uma implementação mais eficiente. De todo modo, como não falamos sobre as funções recursivas, uma sugestão é buscar nas páginas de pesquisa sobre o assunto. Será encontrado diversos materiais. Segue uma possível solução para o problema:

Script R:

```
1 factorial <- function(x) {  
2   if(x > 1) {  
3     x * Recall(x - 1)  
4   } else {  
5     1  
6   }  
7 }
```

Dica do Exercício 6.6 na página 212:

Usando as estruturas de controles, dependendo da implementação pode ser utilizado muitas funções, não necessariamente uma função, resolvemos o problema. Para o caso do não uso, pelo sistema de indexação juntamente com os operadores lógicos, podemos ter uma solução viável.

Dica do Exercício 6.7 na página 213:

Como sugestão, recomendamos estudar a implementação da função repeat nos Códigos R 6.15 e 6.16.

Dica do Exercício 6.8 na página 213:

Sugerimos a utilização das estruturas de controle, juntamente com as funções if() e similares para a solução.

Dica do Exercício 6.9 na página 213:

Como sugestão, recomendamos estudar as implementações nos Códigos R 6.21 e 6.22. Para computar o valor máximo, usamos a função max().

Dica do Exercício 6.10 na página 213:

Podemos usar a função apply(). Como sugestão, podemos consultar o Código R 6.24.

Dica do Exercício 6.11 na página 213:

Como sugestão, recomendamos a leitura da Secção 6.5 deste capítulo.

Dica do Exercício 6.12 na página 214:

Como sugestão, podemos criar uma função dentro de outra. A chamada da função interna sempre será o ambiente de execução da principal.

Dica do Exercício 6.13 na página 214:

Como sugestão, devemos lembrar das formas sintáticas dos operadores binários.

Dica do Exercício 6.14 na página 215:

Podemos saber mais detalhes sobre determinadas funções nos manuais de ajuda do R.

Dica do Exercício 6.15 na página 215:

Estudar sobre o escopo léxico e o objeto reticências (...).

Dica do Exercício 6.16 na página 215:

Como sugestão, recomendamos estudar o Código R ?? para a implementação.

Dica dos Exercícios do Capítulo 7

Dica do Exercício 7.1 na página 226:

Algumas opções de pesquisa no capítulo sobre código no livro *R packages* (WICKHAM, 2015), em <https://r-pkgs.org/r.html#code-style>, o guia de estilo *tidyverse* em <https://style.tidyverse.org/> e o guia de estilo R do Google em <https://google.github.io/styleguide/Rguide.html>.

Dica do Exercício 7.2 na página 226:

Algumas opções de pesquisa no capítulo sobre código no livro *R packages* (WICKHAM, 2015), em <https://r-pkgs.org/r.html#code-style>, o guia de estilo *tidyverse* em <https://style.tidyverse.org/> e o guia de estilo R do Google em <https://google.github.io/styleguide/Rguide.html>.

Dica do Exercício 7.3 na página 226:

Algumas opções de pesquisa no capítulo sobre código no livro *R packages* (WICKHAM, 2015), em <https://r-pkgs.org/r.html#code-style>, o guia de estilo *tidyverse* em <https://style.tidyverse.org/> e o guia de estilo R do Google em <https://google.github.io/styleguide/Rguide.html>.

Dica do Exercício 7.4 na página 226:

Algumas opções de pesquisa no capítulo sobre código no livro *R packages* (WICKHAM, 2015), em <https://r-pkgs.org/r.html#code-style>, o guia de estilo *tidyverse* em <https://style.tidyverse.org/> e o guia de estilo **R** do Google em <https://google.github.io/styleguide/Rguide.html>.

Dica do Exercício 7.5 na página 227:

Como sugestão, podemos executar a chamada `SMR::dSMR` para verificar o código interno da função `dSMR()`. Para instalar o pacote **SMR**, via CRAN, usamos o comando `install.packages("SMR")`.

Dica dos Exercícios do Capítulo 8

Dica do Exercício 8.1 na página 242:

Como sugestão, recomendamos baixar o arquivo `.tar.gz` e procurar pelo arquivo `DESCRIPTION`.

Dica do Exercício 8.2 na página 242:

O problema está relacionado a anexação e caminho de busca. Para um estudo mais profundo, sugerimos a leitura do Capítulo 9.

Dica do Exercício 8.3 na página 243:

Como sugestão, recomendamos a leitura da Seção 8.9 deste capítulo.

Dica do Exercício 8.4 na página 243:

Recomendamos assistir ao vídeo <https://youtu.be/FKItT65gphM>.

Dica do Exercício 8.5 na página 244:

Como sugestão, recomendamos a leitura na opção instalação do `leem` na página do pacote em: <https://bendeivide.github.io/leem/#instalação>.

Dica do Exercício 8.6 na página 244:

Como sugestão, recomendamos a leitura na opção instalação do `leem` na página do pacote em: <https://bendeivide.github.io/leem/#instalação>. Geralmente, as versões dos pacotes no `GitHub` se atualizam a todo momento, e nem sempre as versões via `CRAN` acompanham essa velocidade, até mesmo pelas limitações de tempo que o mantenedor tem para atualizar o pacote de uma versão anterior, para uma versão seguinte.

Dica do Exercício 8.7 na página 244:

Podemos usar a chamada `?install.packages()`.

Dica do Exercício 8.8 na página 244:

Lembramos que tudo que é impresso no *console* é por meio da chamada `print()` nos bastidores, sendo que essa função também é um genérico.

Dica dos Exercícios do Capítulo 9

Dica do Exercício 9.1 na página 267:

Como sugestão, podemos usar o operador `::`.

Dica do Exercício 9.2 na página 267:

Podemos usar a função `new.env()` para criar um ambiente.

Dica do Exercício 9.3 na página 268:

Como sugestão, recomendamos a leitura dos Capítulos 8 e 9.

Dica do Exercício 9.4 na página 268:

Como sugestão, recomendamos a leitura sobre superatribuição, Seção 9.5.

Dica do Exercício 9.5 na página 268:

Como sugestão, recomendamos a leitura sobre ambiente envolvente e superatribuição.

Dica do Exercício 9.6 na página 269:

Como sugestão, recomendamos a leitura sobre ambiente superior. Em outras situações, podemos usar a função `on.exit()`, como solução do problema.

Dica do Exercício 9.7 na página 270:

Devemos estudar sobre a hierarquização de ambientes.

Dica dos Exercícios do Capítulo 10

Dica do Exercício 10.1 na página 278:

Para explorarmos o código interno de funções primitivas, podemos usar a função `pryr::show_c_source()`.

Dica do Exercício 10.2 na página 278:

Nos fóruns de perguntas, como por exemplo <https://stackoverflow.com>, é um bom começo para pesquisar.

Dica do Exercício 10.3 na página 278:

Podemos pensar, por exemplo, que a chamada de uma função é uma interface para executar todo o código interno (corpo da função). Muitos outros contextos o termo interface por ser abordado.

Dica do Exercício 10.4 na página 279:

Uma das soluções apresentadas neste capítulo é o uso do pacote `Rcpp`.

Dica dos Exercícios do Capítulo 11

Dica do Exercício 11.1 na página 281:

Como sugestão, recomendamos a leitura sobre o sistema S3!

Dica do Exercício 11.2 na página 281:

Como sugestão, recomendamos estudar sobre o paradigma da orientação a objetos.

Dica do Exercício 11.3 na página 282:

Como sugestão, recomendamos a leitura sobre os ambientes.

Dica do Exercício 11.4 na página 282:

Como sugestão, recomendamos a leitura sobre métodos no paradigma da programação orientada a objetos, bem como os genéricos de grupo no ambiente R. Um exemplo é o método “verb | +.gg |” para o genérico

Dica do Exercício 11.5 na página 282:

Por exemplo, datas, horas e fusos horários.

Dica do Exercício 11.6 na página 282:

Como sugestão, recomendamos a leitura sobre o paradigma da programação funcional.

Dica do Exercício 11.7 na página 282:

Como sugestão, recomendamos a leitura sobre avaliação preguiçosa dos argumentos nas funções.

Referências

- ADLER, J. *R in a Nutshell*. Sebastopol: O'Reilly Media, 2012. 699 p.
- 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. 550 p.
- CHAMBERS, J. M. *Software for Data Analysis: Programming with R*. New York: Springer, 2008. 498 p. (Statistics and Computing).
- CHAMBERS, J. M. *Extending R*. Boca Raton, Florida: Chapman and Hall/CRC, 2016. 364 p. (The R Series).
- CHAMBERS, J. M.; HASTIE, T. J. *Statistical Methods in S*. London: Chapman & Hall, 1991. 624 p.
- CHAMBERS, J. M.; HASTIE, T. J. *Programming with Data: A guide to the s language*. Ney York: Springer, 1998. 484 p.
- CHANG, W. *R Ggraphics Cookbook*. Sebastopol: O'Reilly Media, 2018. 444 p. 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. 230 p. 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. 302 p. Disponível em: <https://adv-r.hadley.nz/>.

MURRELL, P. *R Graphics*. 3. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2019. 441 p. (The R Series).

PARADIS, E. *R for Beginners*. France: Institut des Sciences de l'Évolution, 2005. 72 p. Disponível em: https://cran.r-project.org/doc/contrib/Paradis-rdebut_en.pdf.

R CORE TEAM. *R data Import/Export*. Vienna, Austria, 2022. 31 p. Version 4.2. Disponível em: <https://cran.r-project.org/manuals.html>.

R CORE TEAM. *R instalation and administration*. Vienna, Austria, 2022. 73 p. Version 4.2. Disponível em: <https://cran.r-project.org/manuals.html>.

R CORE TEAM. *R internals*. Vienna, Austria, 2022. 68 p. Version 4.2. Disponível em: <https://cran.r-project.org/manuals.html>.

R CORE TEAM. *R language definition*. Vienna, Austria, 2022. 55 p. Version 4.2. Disponível em: <https://cran.r-project.org/manuals.html>.

R CORE TEAM. *Writing R extensions*. Vienna, Austria, 2022. 203 p. Version 4.2. Disponível em: <https://cran.r-project.org/manuals.html>.

SILGE, J.; ROBINSON, D. *Text mining with R*. O'Reilly Media, 2017. 194 p. Disponível em: <https://www.tidytextmining.com/>.

STEEL, R. G. D.; TORRIE, J. H. *Principles and procedures of statistics: A biometrical approach*. 2. ed. New York: McGraw-Hill Book Company, 1980. 512 p.

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, 2022. 99 p. Version 4.2. Disponível em: <https://cran.r-project.org/manuals.html>.

WICKHAM, H. *R Packages*. Sebastopol: O'Reilly Media, 2015. 202 p. 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!). Disponível em: <https://ggplot2-book.org/>.

- WICKHAM, H. *Advanced R*. 2nd. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2019. 588 p. Disponível em: <https://adv-r.hadley.nz/>.
- WICKHAM, H. *Mastering shiny*. O'Reilly Media, 2021. 372 p. Disponível em: <https://mastering-shiny.org/>.
- WICKHAM, H.; GROLEMUND, G. *R for Data Science*. Sebastopol: O'Reilly Media, 2017. 550 p. 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. Disponível em: <https://ggplot2-book.org/>.
- XIE, Y. *Dynamic documents with R and knitr*. 2. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2015. 190 p. 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. 138 p. Disponível em: <https://bookdown.org/yihui/bookdown/>.
- XIE, Y.; ALLAIRE, J. J.; GROLEMUND, G. *R Markdown: The definitive guide*. Chapman and Hall/CRC, 2018. 304 p. 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). Disponível em: <https://bookdown.org/yihui/blogdown/>.

Índice Remissivo

- .RData, 12, 37–39, 42, 132, 138, 139, 141, 143, 292
- .Rhistory, 12, 36–39, 42, 139, 141, 292
- Abreviação alternativa, 82
- Álgebra linear, 112
- Ambiente, 3, 36, 37, 41, 207, 237, 247, 284
 - corrente, 209, 248, 251, 252, 262
 - de chamada, 9, 208, 209
 - de execução, 9, 207–209, 211, 256, 257
 - de pacote, 37, 239, 261
 - envolvente, 9, 207, 208, 242, 256, 257, 263
 - global, 17, 30, 36, 37, 41, 42, 47, 129, 132, 139, 207, 209, 233, 242, 247, 248, 253, 256, 258, 261, 262, 266, 269, 289
 - .GlobalEnv, 36, 47, 129, 140, 209, 247–249
 - R_GlobalEnv, 197, 207, 247, 251, 252, 254, 255
 - pai ou superior, 47, 67, 209, 251, 257, 258, 266, 301
 - vazio, 209, 262, 263
 - R_EmptyEnv, 253
- Ambiente de trabalho, 240
- Ambientes funcionais, 256
- Anexar pacotes, 3, 8, 161, 162, 233, 237, 238, 240, 261, 299
- API*, 129
- Área de trabalho, 36, 37, 39, 141
- Argumento padrão, 160, 190
- Arquivo de dados do R
 - <>.RData, 142
 - <>.rda, 142
 - <>.rds, 142
- Arquivos
 - binário, 129, 130, 132, 145, 147
 - texto, 129
- Aspas
 - duplas, 72
 - simples, 72
- Atribuição, 34, 36, 46, 47, 208, 211, 247, 266, 291
 - >, 47
 - <-, 8–10, 17–21, 32, 34, 35, 37, 44–48, 52, 56, 58, 59, 63, 64, 66, 68, 79–82, 85, 86, 90–93, 99, 101, 104–106, 109, 113, 115, 122, 124, 125, 127, 131, 138, 140, 143–146, 148–150,

- 156, 159, 162, 165–168,
170–174, 176, 177, 179–
181, 185, 186, 196, 198,
202, 204, 205, 207–209,
212, 218–224, 226, 227,
233, 242, 244, 245, 249,
250, 252–254, 257, 258,
260, 263, 264, 267, 275,
291
=, 48, 127, 138, 144, 146, 148–
150, 224, 226, 244, 245,
252, 254, 258, 265, 275,
291
?assign0ps, 48, 291, 292
assign, 45, 48
- Atributos, 12, 44, 53, 55, 58, 105,
115
class, 12, 53–55, 290
dim, 51, 72, 105–107, 109, 110
intrínseco, 12, 53, 58, 66
comprimento, 58, 66, 72,
117
modo, 44, 50, 58, 59, 67,
70, 157
levels, 64, 72, 89, 90, 124
names, 56, 111, 290
row.names, 55, 56
tipo, *veja* Atributos, intrínseco, modo, 50, 58, 157
- Banco de dados, 4, 41, 130–132,
138
arquivo de dados, 130
exportar, 9, 129, 139, 145, 153,
154
importar, 9, 129, 131, 132,
153, 154
preparação dos dados, 130
- Biblioteca de pacotes, 230
- Biblioteca de pacotes, 233
Big Data, 129
Biocondutor, 229
Boas práticas de um código, 3,
11, 41, 42, 217
- Cadeia de caracteres, 58, 68
Caminho de busca, 3, 8, 120, 121,
161, 209, 233, 237–239,
247, 261–264, 275, 299
- Carregar pacotes, 3, 8, 233, 237,
240
- Chamada de comandos anteriores, 36
- Chamada de função, 28, 33, 156,
162, 211, 231, 256
aninhada, 162, 163
intermediária, 162, 295
pipe, 163, 295
- Ciência de dados, 22, 163
- Classe de objetos, 54, 58, 70, 243,
245
data.frame, 55, 56, 117
factor, 54, 55, 64, 72, 117
implícita, 54, 55, 70
array, 55, 110
character, 55
double, 64
integer, 64, 106, 107, 109,
110
list, 55, 56
logical, 55
matrix, 55, 107
numeric, 55, 64, 106, 107,
109, 110
- Coercão, 67, 68, 70, 115
- Coletor de lixo, 48
- Comandos elementares, 32
- atribuição, 32

- declaração, *veja* Comandos elementares, expressão expressão, 32
- Comentário, 35, 217
#, 18–21, 34, 35, 37, 58, 92–99, 101, 102, 104–110, 113, 115, 116, 118, 120, 122, 123, 131, 140, 143–146, 148, 150–152, 156, 159, 160, 162, 163, 166–174, 177, 179–181, 198, 207–209, 218–223, 237, 248–250, 252, 254, 255, 257, 258, 260, 263–266, 273–275
- Concatenar, 79
- Console*, 12, 29, 31, 33–36, 38–40, 45, 49, 53, 99, 100, 111, 131, 288
- Contagem de referência, 291
- Controle de fluxos, 3, 163
- Correção de comandos, 36
- Criando ambientes, 250
- Criando funções, 196
- Cópia de objeto, 3, 124, 164, 266, 291
- Desenvolvimento de pacotes, 3, 4, 58, 269
- Diretório, 40, 41
- Diretório corrente, 41, 43
- Diretório de trabalho, 36, 38, 40, 41, 43, 135, 139, 232, 289
- Documentações de um pacote, 241
- Documentações no R, 4
- Encapsulamento, 190
- Endereço na memória virtual de objetos, 46, 256
- Escopo, 12, 208, 247, 285
da função, 10
léxico, 3, 17, 25, 207, 208, 210, 263, 284, 297
padrão, 17
- Escopo léxico, 208
- Espaço de trabalho, 36, 263
- Espelho, 229, 231
- Estrutura básica de um pacote, 230
DESCRIPTION, 230, 299
man/, 230, 241
NAMESPACE, 230, 240, 243
R/, 230
- Estrutura de controle, 163, 213, 296
- Estrutura de dados, 3, 12, 44–46, 50, 51, 53, 69, 70, 72, 124
array, 6, 10, 70, 105, 109, 111, 122, 185, 190, 192, 290
expressão, 70
fator, 54, 72
lista, 11, 55, 70, 105, 113–115, 117, 122, 185, 189, 190, 192, 245, 249, 264, 267
data frame, *veja* Estrutura de dados, lista, quadro de dados, 70, 105, 117, 120, 122, 129, 132, 185, 265
quadro de dados, 8, 54, 70, 105, 117, 120, 122, 123, 127, 129, 132, 142, 147, 185, 290
tibble, 3, 54, 147

- matriz, 10, 54, 70, 72, 105, 111, 117, 120, 185, 190, 213
- tabelas de contingência, 54
- vetor, 10, 45–47, 70, 104, 105, 111, 117, 127, 170, 190
- bidimensional, *veja* Estrutura de dados, matriz escalar ou constante, 72
- longo, 79
- multim dimensional, *veja* Estrutura de dados, array
- unidimensional, *veja* Estrutura de dados, matriz
- vazio, 59, 90
- vetor atômico, 44, 50, 51, 58, 66, 67, 69, 70, 72, 90, 105, 111, 113, 192, 249
- bruto, 72
- caractere, 70, 142
- complexo, 72
- inteiro, *veja* Estrutura de dados, vetor atômico, numérico
- lógico, 70
- numérico, 70
- real, *veja* Estrutura de dados, vetor atômico, numérico
- Estrutura externa do objeto, 46
- Estrutura interna do objeto, 44
- Estruturas de controle, 164, 212, 295, 296
- Estudando o ambiente R, 1, 153
- demais volumes, 4
- volume IV, 228
- volume V, 231, 241
- volume I, 3, 4, 6, 111, 280
- volume II, 3, 6, 47, 58, 70, 72, 157, 162, 206, 247, 280
- volume III, 4, 273, 280
- Execução de comandos, 34, 42
- + , 35
- R
- tecla de atalho F5, 39
- RStudio
- botão Run, 40
- tecla de atalho CRTL+ENTER, 40
- Função, 156, 210
- anônima, 9, 199
- componentes
- ambiente, 158, 161, 196, 212
- argumento, 138, 158, 160, 196, 212
- corpo, 158, 161, 196, 211, 212
- família *apply*, 184, 213, 292
- genérica, 206, 243
- interna, 158
- interna de um pacote, 8, 240
- loop, 164, 170, 175, 177, 184
- ciclo de repetição, 175
- contador, 175
- O que é uma função no R?, 156
- para matrizes, 112
- primitiva, 33, 59, 158, 160, 212, 278, 295, 302
- recursiva, 295
- saída explícita, 197
- vetorizada, 170, 185
- Genéricos de grupo, 53
- ?groupGeneric, 53

- Complex, 53
- Math, 53
- Ops, 53
- Summary, 53
- Git*, 4
- GitHub*, 4, 229, 232, 244
- Gráficos, 4, 203
- GUI*, 239
- Gui de estilo R do *Google*, 225
- IDE*, 23
- IDE*, 3, 22, 25
- Importação de dados, 40
- Instalação de um pacote, 161, 231
- Interface Gráfica ao Usuário, 2, 4, 155
- John McKinley Chambers, 13, 16, 25, 27
- Journal of Statistical Software*, 229
- Jupyter, 272
- LAPACK, 111
- LAT_EX, 7, 8, 151, 153, 272
- Linguagem
 - C, 4, 7, 11, 16, 17, 21, 33, 50, 54, 63, 67, 91, 112, 158, 173, 211, 271, 272, 274, 279
 - C++, 7, 11, 16, 112, 272–274
 - CSS, 1, 153, 272
 - FORTRAN, 4, 11, 16, 21, 112, 271
 - Gtk+, 272
 - HTML, 1, 11, 153, 272
 - Java, 12, 272
 - JavaScript, 1, 153, 272
 - Julia, 11, 22, 272
 - Markdown, 272
 - Python, 11, 22, 272, 273
- Python
 - Python, 273, 274
 - Ruby, 272
 - S, 4, 13, 16, 17, 21, 27, 50, 63, 67, 111
 - S-PLUS, 13
 - S, 17
 - Scheme, 16
 - Tcl/Tk, 269, 271–273, 275
 - Linguagem de baixo nível, 50, 157
 - Linguagem interpretada, 3, 271
 - Linhas de comando, 39
 - Linhas de comando, 30, 31, 35–39, 41, 42, 135
 - Manipulando vetores, 90
 - Manipular matrizes, 111
 - Manipulação de objetos, 3
 - Matriz
 - multidimensional, 109, *veja* Estrutura de dados, array
 - unidimensional, *veja* Estrutura de dados, matriz, 109
 - Memória virtual, 28, 34, 45, 46, 48, 62, 81, 164, 233, 237
 - Modificação no local, 291
 - Máscara de nome, 208
 - Método, 243
 - Namespace, 3
 - ambiente, 161, 267
 - arquivo, 8, 230, 240, 243
 - Nome de arquivos, 220
 - Nomes, 31
 - não sintáticos, 32, 42
 - sintáticos, 32, 42, 130
 - Novo começo, 211

- Numérico, 63, 64
 - dupla precisão, 63
 - inteiro, 63
 - número real, 63
- O que é um pacote?, 3, 161, 228
- Objetivos de um pacote, 233
- Objeto, 8, 34, 41, 44–46, 49, 54, 125, 132, 156
 - atômico, 122
 - interno, 54
 - recursivo, 113, 122
- Operador
 - aritmético, 95
 - binário, 99, 100, 297
 - booleano, 103, 104, 111
 - lógico, 100, 104, 296
 - pipe*, 103, 162, 163, 212, 214, 226, 295
 - únario, 99, 100
- Operações aritméticas, 95
 - ?Aritmetic, 99
- Ordenação de argumentos, 200
- Pacotes, 6, 161, 228
 - abind, 6
 - abind(), 112
 - base, 158, 161
 - bookdown, 7
 - codetools, 7, 263
 - findGlobals, 264
 - datasets
 - airquality, 226
 - devtools, 232
 - install_github(), 232
 - distill, 7
 - ellipse, 7
 - ellipse, 206
 - foreign, 162
- formatR, 7, 218, 225
- gecoder, 272
- ggplot2, 7
- JuliaCall, 272
- knitr, 7, 153
- learnr, 7
- leem, 213, 244
 - new_leem(), 213
 - tabfreq(), 213
- lobstr, 7, 17, 45
 - obj_addr(), 18–20, 45, 46, 48, 125, 291
 - obj_size(), 81, 82
- magrittr, 103, 162
 - %>% , 103
- magrittr, 7
- midrangeMCP, 7, 231–233, 238, 239, 262
 - guimidrangeMCP(), 239
 - MRbarplot(), 236
 - MRtest(), 233
 - MRwrite(), 238
- openxlsx, 7, 145, 155, 293
 - read.xlsx(), 293
- pryr, 7
 - show_c_source(), 278, 302
- Rcpp, 7, 112, 272, 274, 302
 - sourceCpp(), 274
- RcppEigen, 7, 112
- readr, 132
- readxl, 7, 145, 293
 - readxl_example(), 145
 - readxl_excel(), 145
- reticulate, 272, 273
 - source_python(), 273
- RGtk2, 272
- rJava, 272
- rlang, 7, 262
 - env_parents, 263

- rmarkdown, 7, 22, 153, 155, 272, 293
 shiny, 7, 22, 153, 155, 293
 sloop, 7, 54
 s3_class(), 54, 64, 67, 106, 109, 110
 SMR, 227, 241, 242, 261, 298
 dSMR, 227, 298
 pSMR(), 241
 styler, 7, 218, 220, 225
 tidyverse, 7, 103, 147
 writexl, 8, 145, 148, 293
 write_xlsx(), 148, 150
 xlsx, 8, 145, 155, 293
 read.xlsx(), 293
 write.xlsx(), 293
 XR, 7, 8
 XRJulia, 272
 xtable, 8, 151
 xtable(), 152
 Pacotes nativos, 70, 161
 base, 4, 6, 53, 59, 185, 242, 263
 !, 103, 159, 172
 !=, 100, 159
 |, 103, 105, 227, 275
 |>, 103, 162, 163, 212, 214, 226, 295
 ||, 103, 159, 266
 &, 103
 &&, 103
 `?`(), 138
 (), 35, 185, 265
 *, 76, 78, 96, 98, 99, 112, 170, 171, 214, 221
 `*`(), 97
 **, 99
 +, 32, 33, 35, 37, 76, 78, 96, 98, 99, 112, 176, 177, 179–181, 196–199, 209, 214, 215, 218, 219, 221–224, 226, 258, 263, 264, 275
 `+`(), 96
 -, 72, 75, 76, 78, 82, 96, 99, 112, 212, 226
 `--`(), 96
 ->, 47
 ., 123, 160, 202–205, 215, 242, 275, 297
 ...elt(), 205
 ...length(), 205
 ...names(), 205
 .GlobalEnv, 36, 47, 129, 140, 209, 247–249
 .Internal(), 51, 72, 157
 .Last.value, 49, 290
 .Primitive(), 157, 272
 /, 35, 75, 97–99, 112, 214, 226, 272
 `/`(), 97, 272
 :, 56, 57, 66, 80–86, 91, 92, 98, 99, 106, 108, 109, 111, 113, 115, 118, 120, 122–124, 138, 144, 146, 151, 168, 170, 171, 183, 185, 189–194, 202, 204, 215, 220, 222, 224, 233, 242, 265
 ::, 18, 19, 46, 48, 67, 82, 106, 109–111, 125, 148, 150, 222, 233, 236–241, 263, 267, 273, 274, 291
 :::, 222, 241
 <, 100, 101, 105, 165, 168, 171
 <-, 8–10, 17–21, 32, 34, 35, 37, 44–48, 52, 56, 58, 59,

63, 64, 66, 68, 79–82, 85,
 86, 90–92, 99, 101, 104–
 106, 109, 113, 115, 122,
 124, 125, 127, 131, 138,
 140, 143–146, 148–150,
 156, 159, 162, 165–168,
 170–174, 176, 177, 179–
 181, 185, 186, 196, 198,
 202, 204, 205, 207–209,
 212, 218–224, 226, 227,
 233, 242, 244, 245, 249,
 250, 252–254, 257, 258,
 260, 263, 264, 267, 275,
 291
 `<`(), 156
 <<-, 8, 10, 19, 20, 253, 254,
 257, 258, 263, 264, 266
 <=, 100, 180, 181
 =, 48, 56, 57, 82–85, 87–89,
 115, 118, 120, 123, 124,
 127, 138, 143, 144, 146,
 148–150, 159, 160, 173,
 174, 185–190, 197, 202,
 204, 205, 212, 223, 224,
 226, 232, 244, 245, 252,
 254, 258, 275, 291
 ==, 62, 73, 80, 92, 94, 100,
 101, 104, 167, 172, 173,
 177, 179, 181, 227, 248,
 249
 >, 100, 105, 166, 168, 212,
 224
 >=, 100
 `?`(), 53, 99, 100, 111, 157,
 161, 215, 293
 `??`(), 162, 177, 293
 [, 57, 91, 105, 108, 111, 113,
 118, 120, 124, 125, 151,
 171, 190, 223, 233
 `[]`(), 157
 [<-, 91, 93, 124
 [[, 113, 115, 118, 189, 249,
 250, 278
 \$, 115, 118, 120, 123, 233,
 249, 250, 256, 267
 `\$<-`(), 21
 %*, 98, 99, 112
 %/, 99
 %%*, 99, 172, 212, 291, 295
 %in%, 95, 100–102, 212
 %x%, 112
 &, 104
 \(), 196, 205
 ^, 76, 78, 99, 265
 print(), 12, 285
 {, 18–20, 159, 166–169, 171–
 173, 175, 177, 179–183,
 196–198, 200, 202, 204,
 207–210, 215, 219, 223,
 224, 244, 254, 257, 258
 , 233
 ~, 113, 115
 all(), 103
 any(), 103
 apply(), 156, 185–188, 192,
 226, 227, 296
 apropos(), 161
 array(), 111
 as.character(), 68
 as.data.frame(), 146, 147
 as.double(), 266
 as.factor(), 124
 as.null(), 73
 as.numeric(), 275
 assign(), 45, 48
 attach(), 120, 121, 123, 264,
 265, 291
 `attr<-`(), 106, 109

- `attributes()`, 53, 54, 67, 106, 107, 109, 110
- `body()`, 158, 159, 197, 207, 212
- `break`, 176, 177, 179–182, 218, 219
- `c()`, 52, 66, 68, 74, 78, 79, 82, 86, 87, 89, 91, 95, 98, 100–102, 104, 106, 109, 111, 112, 116, 117, 123–125, 159, 233
- `car()`, 172
- `cat()`, 173, 174
- `cbind()`, 111, 227, 290
- `ceiling()`, 100
- `character()`, 61, 90, 91, 122
- `chol()`, 112
- `class()`, 53, 54, 56, 67, 244
- `close()`, 144
- `colnames()`, 111, 120
- `crossprod()`, 112
- `data.frame()`, 56, 57, 117, 118, 120, 122, 144, 146, 223, 265
- `det()`, 112
- `detach()`, 121, 239, 266
- `diag()`, 112
- `dim()`, 111, 292
- ``dim<-`()`, 52, 106, 109
- `dimnames()`, 111
- `double()`, 63
- `eapply()`, 185
- `eigen()`, 112
- `else`, 166–169, 172–174, 218, 219, 224, 227, 266
- `emptyenv()`, 253, 263, 264
- `environment()`, 20, 158, 160, 197, 207, 212, 248, 254, 255, 258
- ``environment<-`()`, 21
- `exists()`, 222
- `exp()`, 100
- `expression()`, 113, 115
- `F`, 72
- `factor()`, 54, 64, 117, 233
- `FALSE`, 62, 65, 66, 68, 72, 73, 88, 100–104, 118, 120, 121, 123, 138, 142, 144, 146, 159, 160, 166, 170, 185, 188, 190, 191, 193, 222, 227, 244, 250, 266, 275
- `file.show()`, 146, 148, 150
- `floor()`, 100
- `for()`, 163, 168, 171, 175, 181–184, 213, 223, 224, 296
- `formals()`, 158, 159, 197, 207, 212
- `function()`, 10, 17, 18, 20, 123, 158, 163, 196, 198, 207, 215, 223, 224, 227, 242, 244, 254, 257, 258, 266, 275
- `getwd()`, 38, 40, 135, 140, 143, 148, 150
- `gl()`, 82, 89
- `globalenv()`, 247
- `gzfile()`, 144
- `head()`, 144, 146, 148, 149, 151
- `help()`, 161
- `identical()`, 62, 63, 73, 144, 248
- `if()`, 32, 163–169, 172–174, 177, 183, 218, 219, 223, 227, 266, 296
- `return()`, 227

ifelse(), 163, 166, 168, 170,
 171
Inf, 72
invisible(), 34, 199
is.array(), 111
is.atomic(), 290
is.factor(), 266
is.finite(), 72
is.function(), 157, 212
is.infinite(), 72
is.integer(), 64
is.logical(), 159
is.matrix(), 111
is.na(), 73, 275
is.nan(), 73
is.null(), 73
is.numeric(), 64, 167, 172
is.primitive(), 212, 295
is.recursive(), 113, 290
is.vector(), 111, 266
lapply(), 185, 189–191
length(), 66, 67, 78, 82,
 85, 86, 111, 120, 156, 159,
 222, 227, 292
LETTERS, 56, 57, 233
letters, 113, 115, 118, 120,
 124
library(), 239
library(), 145, 148, 151,
 161–163, 218, 233, 237,
 261, 262, 273, 275
list(), 113, 115, 116, 149,
 157, 189–193, 226, 244,
 254, 267
list.files(), 140, 143
load(), 139, 142, 143
loadhistory(), 140
log(), 100
log10(), 100
logical(), 90, 91, 122
ls(), 37, 140, 247–250, 255,
 257
make.names(), 32
mapply(), 185, 192, 195
match(), 92, 94, 95
matrix(), 51, 108, 186, 187,
 190
max(), 94, 226, 227, 291,
 292, 296
mean(), 100, 113, 115, 138,
 157, 174, 186–189, 191–
 193, 215, 220, 222, 242
median(), 100
min(), 94, 226, 292
mode(), 46, 50, 54, 58, 59,
 61, 63, 66–68, 113, 156–
 158
NA, 72–74, 76, 78, 90
NA_character_, 90
NA_complex_, 90
NA_integer_, 90
NA_real_, 90
names(), 57, 111, 120, 290,
 292
`**names<-`()**, 57
NaN, 72–76, 78
ncol(), 111, 120
new.env(), 20, 21, 250, 252,
 263
next, 176, 177, 179, 181–
 184
nrow(), 111, 120
NULL, 72, 73, 78, 106, 109,
 192, 205, 224, 232, 244
numeric(), 61, 63, 90, 91,
 171
objects(), 37
on.exit(), 301

- options(), 31, 33
- new.env(), 252
- parent.env(), 251, 252, 254
- `parent.env<-`(), 252
- paste(), 148, 150, 275
- pi, 272
- print(), 10, 17, 18, 20, 33, 123, 156, 166, 167, 177, 180, 181, 183, 199, 218, 219, 224, 285, 300
- prod(), 99
- q(), 139
- qr(), 112
- rank(), 112
- rapply(), 185
- rbind(), 111
- read.csv(), 131
- read.dbf(), 161, 162
- read.delim(), 131
- read.table(), 131, 135, 138, 140, 143, 153, 293
- readRDS(), 144
- Recall(), 295
- rep(), 82–84, 118, 120, 123, 193, 194, 227, 233, 292
- rep_len(), 82–84
- repeat(), 163, 164, 175–177, 179–181, 213, 218, 219, 296
- replicate(), 82, 87, 88
- require(), 233, 237
- return(), 197, 207, 254, 258
- return(), 196, 198
- rm(), 37
- round(), 100, 174, 195
- rownames(), 111, 120
- sample(), 92, 124, 185
- mapply(), 192–194
- sapply(), 185, 190, 191
- save(), 139, 142, 143
- save.image(), 139
- savehistory(), 140
- saveRDS(), 142, 144
- scan(), 131
- sd(), 100, 212
- search(), 121, 209, 237, 239, 261, 262, 265
- seq(), 82, 84, 85, 87, 88, 292
- seq_along(), 82, 86
- seq_len(), 82, 86
- sequence(), 82, 86, 87
- set.seed(), 92, 162, 173, 174, 245
- setwd(), 38, 135, 140
- sink(), 49
- solve(), 112
- sort(), 158–162
- source(), 41, 127, 233, 292
- sqrt(), 75, 76, 100, 162, 163, 266
- stop(), 159
- str(), 116
- subset(), 226
- sum(), 99, 215
- svd(), 112
- switch(), 54, 163, 173, 174, 212
- T, 72
- t(), 99, 111, 112
- tail(), 149
- tapply(), 185
- tcrossprod(), 112
- tempfile(), 144, 146
- typeof(), 50
- TRUE, 32, 59, 60, 62, 63, 66, 68, 69, 72, 80, 87, 88, 95, 100–104, 108, 113, 114,

- 117–121, 123, 124, 138–140, 142–144, 160, 164, 166, 167, 170, 185, 187, 188, 190–192, 222, 226, 227, 239, 248–250
- `trunc()`, 100, 172
- `typeof()`, 50, 52, 54, 59, 61–63, 66, 67, 79, 80, 113, 157, 158, 163, 196, 290, 291, 293
- `unclass()`, 55, 56, 64, 291
- `UseMethod()`, 159, 212
- `var()`, 100, 162, 163
- `vector()`, 51
- `which()`, 92, 94, 291
- `which.max()`, 92, 93
- `which.min()`, 92, 93
- `while()`, 163, 164, 175, 179–181
- `write()`, 142, 155
- `write.csv()`, 142, 155
- `write.csv2()`, 142, 155
- `write.ftable()`, 142, 155
- `file.show()`, 144
- `write.table()`, 141, 144, 146, 293
- `xor()`, 103
- `datasets`, 7, 148
 - `cars`, 148
 - `mtcars`, 151
- `foreign`, 7, 153, 155, 161, 294
- `graphics`, 7, 202
 - `hist()`, 213
 - `plot()`, 202–204
- `methods`, 53
- não documentado
 - `inspect()`, 51, 72
- padrão, 54
- recomendado, 54
- `stats`, 7, 243, 266
- `anova()`, 233
- `aov()`, 233
- `integrate()`, 200
- `rnorm()`, 82, 162, 173, 174, 195, 202, 204, 212, 245
- `rnorm()`, 85, 88
- `runif()`, 212
- `sd()`, 266
- `t.test()`, 243, 245
- `var()`, 266
- `tcltk`, 269, 271, 272, 275, 277
 - `tclvalue()`, 275
 - `tclVar()`, 275
 - `tkbutton()`, 275
 - `tkentry()`, 275
 - `tkframe()`, 275
 - `tklabel()`, 275
 - `tkpack()`, 275
 - `tkpack.propagate()`, 275
 - `toplevel()`, 275
- `utils`, 8, 153
 - `history()`, 36
 - `install.packages()`, 17, 45, 145, 148, 151, 163, 218, 231, 232, 241, 244, 273, 274, 298
 - `Sweave()`, 153, 155, 293
- Pacotes órfãos, 232
- Palavras reservadas, 32, 42, 73, 74, 164, 221
 - `..1`, 205
 - `..2`, 205
- ?Reserved, 32
- `break`, 176, 177, 179–182, 218, 219
- `else`, 166–169, 172–174, 218, 219, 224, 227, 266

- FALSE, 62, 65, 66, 68, 73, 88, 101–104, 118, 120, 121, 123, 138, 144, 159, 166, 170, 188, 190, 191, 193, 222, 227, 244, 250, 266, 275
for, 32, 163, 168, 171, 175, 181–184, 213, 223, 224, 296
function, 158, 163, 196, 198, 207, 215, 223, 224, 227, 242, 244, 254, 257, 258, 266, 275
if, 32, 163–169, 172–174, 177, 183, 218, 219, 223, 227, 266, 296
in, 181, 183
Inf, 72, 73, 75
NA, 72–74, 76, 78, 90
NA_character_, 90
NA_complex_, 90
NA_integer_, 90
NA_real_, 90
NaN, 72–76, 78
next, 176, 177, 179, 181–184
NULL, 72, 73, 78, 106, 109, 192, 205, 224, 232, 244
repeat, 175–177, 179–181, 213, 218, 219, 296
TRUE, 32, 62, 63, 66, 68, 69, 72, 87, 88, 95, 100–104, 114, 118–121, 123, 124, 138–140, 160, 164, 166, 167, 170, 185, 187, 188, 190–192, 222, 226, 227, 239, 249, 250
while, 163, 164, 175, 179–181
Pesquisa dinâmica, 210
Posit, 22
- Prefixos e/ou sufixos**
as.<nome_função>, 68–70, 266, 275
i, 59
is.<nome_função>, 64, 68, 69, 72, 73, 111, 113, 157, 167, 172, 266, 275, 290
L, 58, 59, 62, 64, 72, 79, 159
r<nome_dist>, 82, 85, 88
- Princípios do R**, 9, 27
função, 27, 156
interface, 11, 16, 27, 129, 271, 287
objeto, 27, 53, 54, 292
- Programação**
defensiva, 2, 171, 173
dinâmica, 3
funcional, 3
meta-paradigma, 3
orientada a objetos, 4, 53, 54
sistema S3, 64, 67, 130
sistema S4, 64
- Prompt de comando*, 29, 31, 288
>, 29, 35
- R**, 10, 14, 27, 29
dashboards, 2
websites, 2
ambiente, 2, 16, 25
artigos, 2
banco de dados, 2
como o R trabalha?, 27
CRAN, 14, 26, 161, 196, 228, 229, 232, 233, 272, 298
primeiro espelho no Brasil, 22
criadores do R
George Ross Ihaka, 13, 16

- Robert Clifford Gentleman,
13
código aberto, 14, 15
escopo, 16
estilo funcional, 157
gerenciamento de memória,
16
gráficos, 2
história, 13
instalação, 23
linguagem, 2, 25
livros, 2
manuais, 33, 74, 161
R Internals, 4
An Introduction to R, 5, 99
R Data Import/Export, 5, 129
R Installation and Administration, 5
R Internals, 5
R language definition, 5
Writing R extensions, 5
O que é o R?, 14
paralelização, 2
programa, 2
R Core Team, 14, 50, 54, 129,
157, 233
R Core Team, 4
semântica, 1, 3, 50, 53
sintaxe, 1, 3, 46, 50, 72, 130,
164, 222
?Syntax, 100
software livre, 14, 15
R-Forge, 229
Reciclagem
vetor, 98
Referência de memória, 45
Repositório de pacotes, 229
Representação
científica, 72
 decimal, 72
 hexadecimal, 72
RStudio, 4, 10, 22, 23, 25, 29, 30,
36, 40–42, 132, 206, 217,
220, 231, 272
entrada de dados, 30
instalação, 23
 J. J. Allaire, 22
quadrantes, 30
 primeiro, 30, 40
 quarto, 31
 segundo, 30
 terceiro, 30
Script, 9, 10, 30, 38–42, 45, 123,
127, 139, 217, 292
<>.R, 220
<>.R, 39
salvar, 40
Separador de declarações ou expressões
;, 34, 35, 37, 49, 58, 59, 64,
68, 79–81, 90–93, 121, 124,
145, 148, 149, 151, 159,
162, 186, 187, 249, 250,
252
Serialização, 142
Sistema de indexação, 53, 91, 104,
111, 113, 122, 290, 291,
296
String, *veja Cadeia de caracteres*, 68
Superatribuição, 8, 25, 257, 258,
266, 284
<<-, 8, 10, 19, 20, 253, 254,
257, 258
Símbolos, 31, 130
The R Journal, 229

- Tipagem, 50, 67
 linguagem C, 50, 51, 59
 linguagem S, 46, 50, 58, 59
- Tipagem interna de objetos
 expressão
 EXPRSXP, 33
 lista
 VECSXP, 113
 lógico
 LGXSXP, 51
 nulo
 NILSXP, 73
 numérico
 REALSXP, 51
 reticências
 DOTSXP, 160, 202
- Tipos de Objeto, 12, 45, 50, 53, 90, 124
 ambiente, 12, 47, 233, 247, 285
 environment, 247
 bruto, 58
 raw, 58, 66, 72, 90
 caractere, 58
 character, 58, 59, 61, 66, 68, 70, 90
 complexo, 58
 complex, 58, 59, 66, 72
 expressão, 113
 expression, 33, 70
 função, 36, 113, 156, 157, 210
 builtin, 50, 156–158
 closure, 36, 50, 156–158, 163, 196, 212
 function (tipagem S), 36, 50
 special, 36, 50, 156–158, 163, 164, 198
 linguagem
 language, 50
 lista, 123, 237, 247, 264, 290
 list, 113, 122
 lógico, 58, 68
 logical, 51, 58, 59, 66, 68, 70
 nome, 34, 36, 45, 46, 49, 123, 132, 156, 165, 221, 237, 247, 251
 name (tipagem S), 50
 symbol, 45, 50, 56
 nulo
 NULL, 78, 106, 109, 192, 205, 224, 232, 244
 numérico, 58
 double, 45, 46, 50–52, 59, 61, 63, 66, 68, 70, 72, 79, 81, 82, 90
 integer, 50, 59, 61, 63, 66, 68, 72, 79–82
 numeric (tipagem S), 46, 50, 58, 59, 61, 63, 66, 68
 reticências, 160, 203, 215, 297
 . . . , 7
- Variável, 45, 130
 Vetorização, 169
- Workspace*, *veja* Área de trabalho

O livro **R básico**, *Volume I* da Coleção *Estudando o Ambiente R*, representa o passo inicial para estudar a linguagem **R**. Este *Volume* apresenta um breve histórico sobre a linguagem, a sua instalação, bem como os recursos da **IDE RStudio**, o conhecimento da sintaxe e semântica, as estruturas bases, o que é um objeto e como construir uma função, o entendimento sobre controle de fluxos. O que é um pacote, carregar e anexar um pacote, e quem são os mantenedores da linguagem, também serão assuntos abordados. Noções básicas sobre o caminho de busca, ambientes e *namespaces* complementam a abordagem sobre o **R**. Algo muito interessante, que pode mudar o *script* de um programador em **R**, são as boas práticas para a escrita de um código, tema também abordado. Vale a pena salientar que o objetivo do livro não é convencer o leitor a adjetivar a linguagem **R** como a melhor dentre as outras. Mas, apresentá-la como uma ferramenta de trabalho com recursos dos mais diversos possíveis. Tenham uma boa leitura!

ISBN 978-65-00-51600-5



9 786500 516005