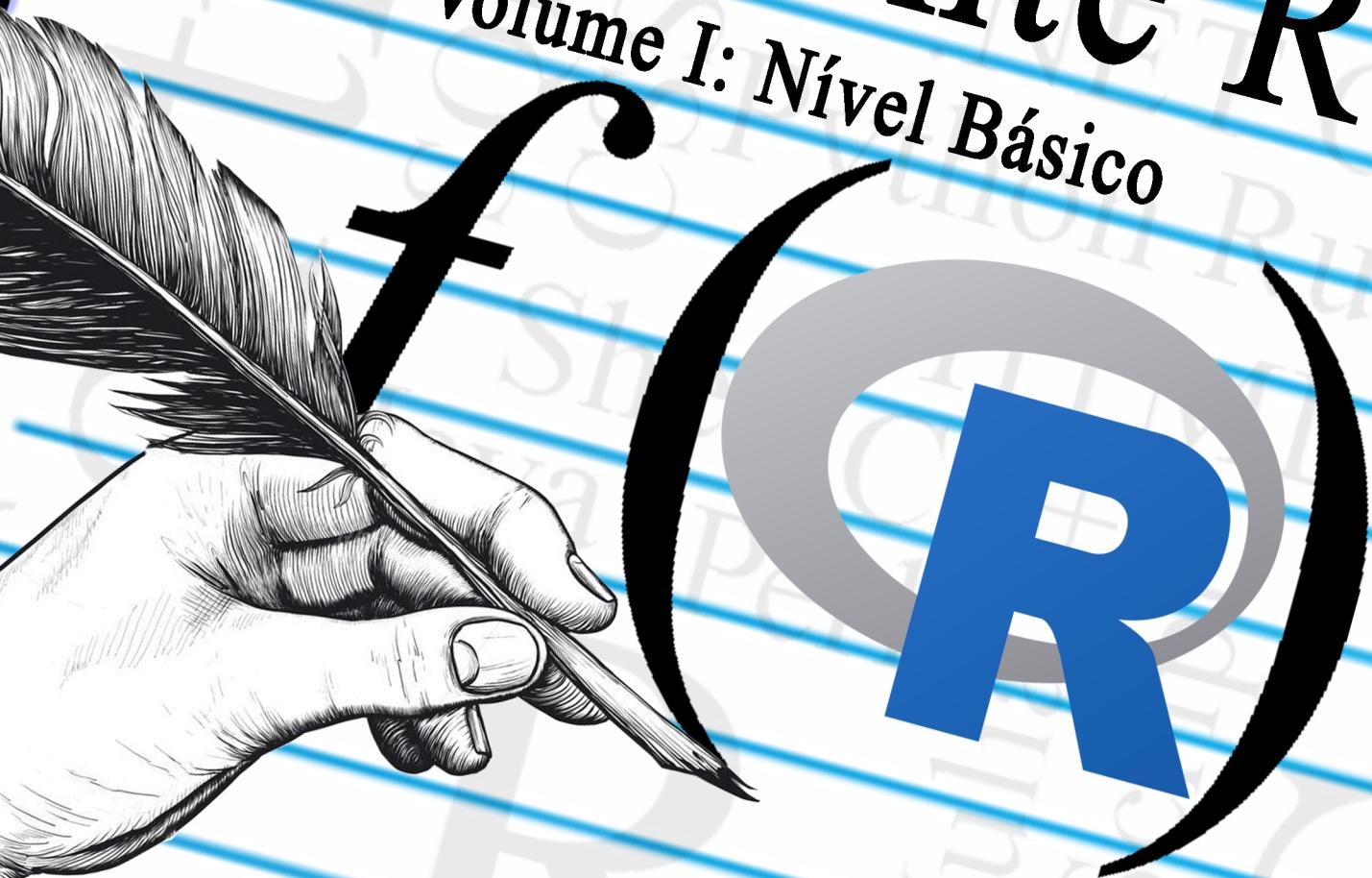


Estudando o Ambiente R

Volume I: Nível Básico



Ben Dêivide
Diego Arthur



Democratizando
Conhecimento

ESTUDANDO O AMBIENTE R

**BEN DÊIVIDE DE OLIVEIRA BATISTA
DIEGO ARTHUR BISPO JUSTINO DE OLIVEIRA**

Estudando o Ambiente R

Volume I: Nível Basico

BEN DÊIVIDE DE OLIVEIRA BATISTA
DIEGO ARTHUR BISPO JUSTINO DE OLIVEIRA



Ouro Branco, MG, 5 de novembro de 2021

© 2021 by Ben Dêivide de Oliveira Batista e Diego Arthur Bispo Justino de Oliveira



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

Direitos de publicação reservados ao seu conhecimento.

Impresso no Brasil - ISBN (Digital):

Impresso no Brasil - ISBN (Impresso):

Projeto Gráfico: Ben Dêivide de Oliveira Batista

Revisão técnica e textual: XX

Revisão de Referências Bibliográficas:

Editoração Eletrônica: Ben Dêivide de Oliveira Batista

Capa: Ben Dêivide de Oliveira Batista

Como citar essa obra:

BATISTA, B. D. O.. **Estudando o Ambiente R.** 1ed. Ouro Branco, MG:[sn]. 2021. 1 v. Disponível em: <<https://bendeivide.github.io/book-eambr02/>>

Autor correspondente e mantenedor da obra:

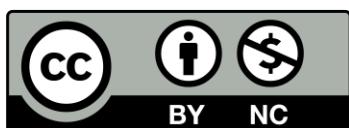
Ben Dêivide de Oliveira Batista

Contato: <ben.deivide@gmail.com>

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

Licença

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



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



Epígrafe

A melhor linguagem é a que você domina!
Ben Dêivide

Sumário

Licença	i
Epígrafe	iii
Prefácio	vi
1 Estratégia	1
1.1 Andamento do projeto	1
1.2 Modelo de caixas	1
1.3 Códigos no texto	2
1.4 Links prontos	2
1.5 Índice remissivo	2
2 Entendendo a coleção Estudando o ambiente R	3
2.1 Volume I: Nível Básico	4
2.2 Volume II: Nível Intermediário	4
2.3 Volume III: Nível Avançado	4
2.4 Demais volumes	5
2.5 Referências complementares da Coleção	5
2.6 Pacotes R utilizados para essa coleção	6
2.7 Materiais didáticos	6
Exercícios propostos	7
3 História e instalação do R	10
3.1 História do R	10
3.2 O que é o R ?	10
3.3 Instalação do R e RStudio	16
4 Como o R trabalha	18
4.1 Introdução	18
4.2 Como utilizar o R e o RStudio	19
4.3 Comandos no R	21
4.3.1 Console e <i>Prompt</i> de comando	21
4.3.2 Comandos elementares	22
4.3.3 Execução de comandos	23
4.3.4 Chamada e correção de comandos anteriores	24
4.4 Ambiente global (área de trabalho ou <i>workspace</i>)	24
4.5 Arquivos .RData e .Rhistory	25
4.6 Criando e salvando um <i>script</i>	25
Exercícios propostos	27

5 Objetos	28
5.1 Introdução	28
5.2 Atributos	30
5.2.1 Atributos intrínsecos	32
5.3 Coerção	39
5.4 Tipo de objetos	41
5.4.1 Vetores	41
5.4.1.1 Vetores escalares ou constantes	41
5.4.1.2 Vetores longos	45
5.4.1.3 Manipulando vetores	49
Exercícios propostos	50
6 Gabarito dos Exercícios	51
Referências Bibliográficas	56
Índice Remissivo	59

Prefácio

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

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

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

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

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

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

Ben Dêivide de Oliveira Batista
Ouro Branco, MG, 5 de novembro de 2021

Estratégia

1.1 Andamento do projeto

1. Última atualização 5 de novembro de 2021;
2. Descrever sempre o que foi realizado!

1.2 Modelo de caixas

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



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

Código R 1.1

Script:

```
1 111
```

Console:

```
1111
```

Script:

```
1 1111
```

Console:

```
1111
```

Código R 1.2

Script:

```
1 1111
```

Script:

1 111

1.3 Códigos no texto

midrangeMCP, meu_documento

1.4 Links prontos

- Pacotes: **lobstr**
- Linguagem: R
- R: **R**
- RStudio: **RStudio**
- link estilizado: Advanced R
- link puro: <<https://adv-r.hadley.nz/>>

1.5 Índice remissivo

- Tópico
- subtópico do Tópico
- subsubtópico do Tópico

Entendendo a coleção Estudando o ambiente R

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

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

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

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

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



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

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

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

A seguir, explanaremos sobre cada um dos módulos.

2.1 Volume I: Nível Básico

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

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

2.2 Volume II: Nível Intermediário

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

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

2.3 Volume III: Nível Avançado

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

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

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

2.4 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.

2.5 Referências complementares da Coleção

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

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

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

2.6 Pacotes R utilizados para essa coleção

Apresentamos uma lista de pacotes, Tabela 2.1, utilizados ao longo da coleção para os exemplos abordados, como também para o próprio desenvolvimento dos livros.

Tabela 2.1: Pacotes a serem instalados para o acompanhamento dos exemplos e exercícios da coleção *Estudando o ambiente R*.

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

2.7 Materiais didáticos

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

Exercícios propostos

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

Exercício 2.1: Observe o seguinte script abaixo:

Script:

```

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

```

Console:

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

Script:

```

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

```

Console:

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

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

Solução na página 51

Exercício 2.2: Por que a superatribuição (`<<-`) deve ser usado com cautela, principalmente quando se está desenvolvendo um pacote?

Solução na página 51

Exercício 2.3: Qual a diferença entre anexar e carregar um pacote?

Solução na página 51

Exercício 2.4: Qual a importância da estrutura NAMESPACE em um pacote?

Solução na página 51

Exercício 2.5: Por que devemos usar com cautela as funções internas não exportáveis, de um pacote? Como podemos acessá-las? Como podemos acessar uma função de um pacote sem anexá-lo ao caminho de busca? Por que esta última condição é mais interessante, quando se deseja utilizar poucas funções de um pacote?

Solução na página 51

Exercício 2.6: Para que serve os arquivos .RData e .Rhistory?

Solução na página 51

Exercício 2.7: Quantos objetos temos na linha de comando a seguir?

Script:

```
1 x <- 10
```

Solução na página 51

Exercício 2.8: Precisamos ter o RStudio para utilizar o ambiente R?

Solução na página 51

Exercício 2.9: Quais os três princípios do R? Identifique-os na linha de comando a seguir.

Solução na página 51

Exercício 2.10: Podemos dizer que uma matriz ou array é um vetor? Se sim, o que os diferenciam?

Solução na página 51

Exercício 2.11: Podemos afirmar que um data frame é uma lista?

Solução na página 52

Exercício 2.12: Como podemos salvar um script? Qual a importância de um script?

Solução na página 52

Exercício 2.13: O que são funções anônimas?

Solução na página 52

Exercício 2.14: Podemos afirmar que sempre o ambiente envolvente e ambiente de chamada são ambientes iguais?

Solução na página 52

Exercício 2.15: Considerando os ambientes envolvente, de execução e de chamada, qual dos três é um ambiente temporário? E quando ele é criado?

Solução na página 52

Exercício 2.16: Baseado na linha de comando: `x <- 1`, por que não pode afirmar que “`x` recebe o valor 1”?

Solução na página 52

Exercício 2.17: Como importamos um banco de dados externo do ambiente **R**? E como exportamos um resultado desejado?

Solução na página 52

Exercício 2.18: Por que comentar um código é muito importante para um programador?

Solução na página 52

Exercício 2.19: Qual a importância de uma boa escrita de um código?

Solução na página 52

Exercício 2.20: Por que o **R** tem como um de seus princípios em sua origem, o princípio da interface?

Solução na página 52

Exercício 2.21: Que pacotes no **R** faz com que esse ambiente interaja com as linguagens: Python, Julia, C/C++, FORTRAN, HTML Java? Quais outras interfaces você conhece para o **R**?

Solução na página 52

Exercício 2.22: O que representa os atributos para um objeto? Quais os dois atributos intrínsecos de um objeto? Qual a importância do atributo `class` para um objeto?

Solução na página 52

História e instalação do R

3.1 História do R

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



Figura 3.1: Criadores do R.¹

Durante a época em que estes professores trabalhavam na Universidade de Auckland, Nova Zelândia, desenvolvendo uma implementação alternativa da linguagem S, desenvolvida por John Chambers, que comercialmente era o **S-PLUS**, nasceu em 1991, o projeto da linguagem **R**, em que em 1993 o projeto é divulgado e em 1995, o primeiro lançamento oficial, como software livre com a licença GNU. Devido a demanda de correções da linguagem que estava acima da capacidade de atualização em tempo real, foi criado em 1997, um grupo central voluntário, responsável por essas atualizações, o conhecido R Development Core Team², que hoje está em 20 membros (atualizado em 5 de novembro de 2021): Douglas Bates, John Chambers, Peter Delgaard, Robert Gentleman, Kurt Hornik, Ross Ihaka, Tomas Kalibera, Michael Lawrence, Friedrich Leisch, Uwe Ligges, Thomas Lumley, Martin Maechler, Sebastian Meyer, Paul Murrel, Martyn Plummer, Brian Ripley, Deepayan Sarkarm, Duncan Temple Lang, Luke Tierney e Simon Urbanek.

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

3.2 O que é o R ?

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

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

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

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

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

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

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

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

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

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

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

Escopo léxico



Estude atentamente os exemplos a seguir, porque o escopo léxico é uma das grandes características que existe no ambiente **R**, e que por exemplo, não existe na linguagem **S**. O escopo léxico dá uma grande flexibilidade as funções para a procura dos objetos, ou de um modo menos formal, em busca das variáveis inseridas no corpo da função criada. Se o leitor for um iniciante, sugerimos seguir na leitura até o Capítulo 5, e posteriormente, retornar a esses exemplos.

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

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

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

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

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

Script:

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

- Exemplo 1: As funções têm acesso ao escopo em que foram criadas, Código R 3.1.

Código R 3.1**Script:**

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

Console:

```
[1] 10
```

- Exemplo 2: As variáveis criadas ou alteradas dentro de uma função, permanecem na função, Código R 3.2.

Código R 3.2**Script:**

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

```
[1] 10
```

Script:

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

Console:

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

- Exemplo 3: As variáveis dentro de uma função permanecem nelas, exceto no caso em que a atribuição ao escopo seja explicitamente solicitada, Código R 3.3.

Código R 3.3**Script:**

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

```
[1] 15
```

Script:

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

Console:

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

- Exemplo 4: Por fim, embora a linguagem R tenha um escopo padrão, chamado ambiente global, os escopos de funções podem ser alterados, Código R n3.4.

Código R 3.4**Script:**

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

Console:

```
[1] 10
```

Script:

```

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

```

Console:

```
[1] 20
```

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

Embora o **R** tenha uma interface baseada em linhas de comando, existem muitas interfaces gráficas ao usuário com destaque ao **R**, criado por Joseph J. Allaire, Figura 3.2.



Figura 3.2: J. J. Allaire, o criador do **RStudio**.⁹

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

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

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

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

3.3 Instalação do R e RStudio

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

- Instalação do **R** - <<https://www.r-project.org>>, Figuras 3.3 e 3.4:

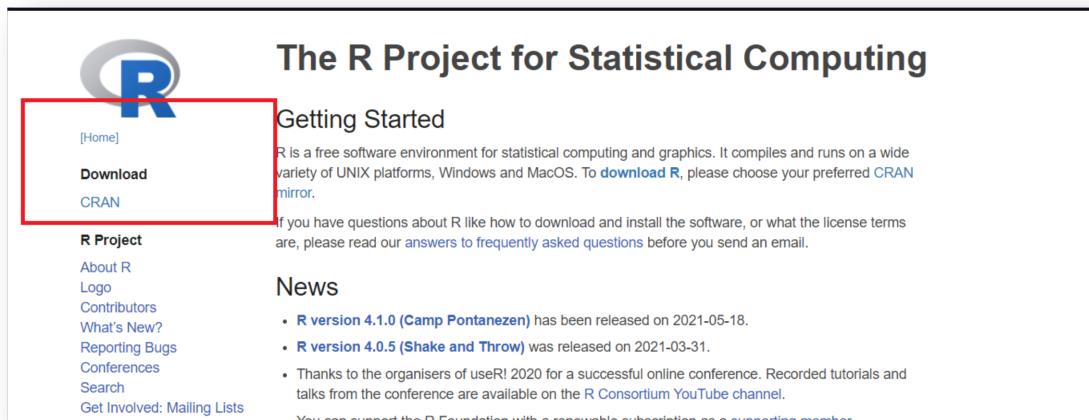


Figura 3.3: Primeiro passo para Instalação do R.

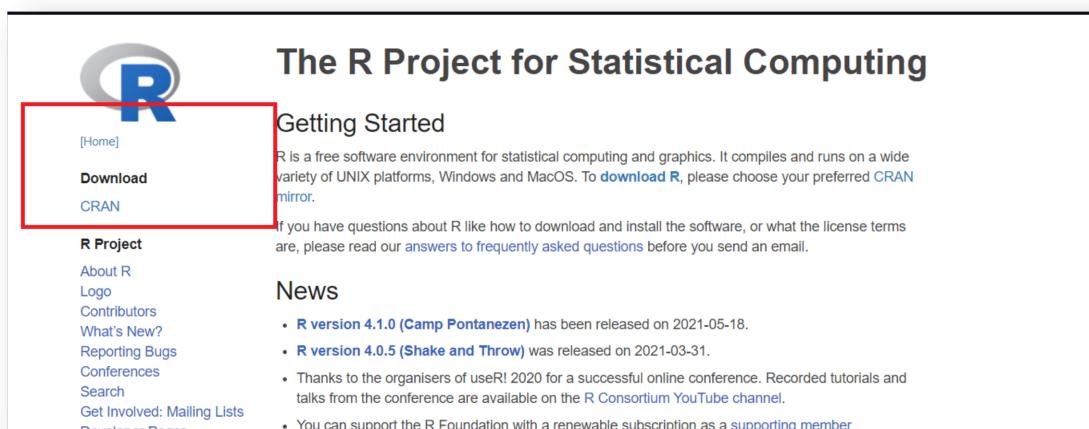


Figura 3.4: Segundo passo para Instalação do R.

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

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

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

All Installers

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

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

Figura 3.5: Instalação do RStudio.

Como o **R** trabalha

4.1 Introdução

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



Figura 4.1: John Chambers¹, o criador da linguagem S.

Princípios do R



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

Ao longo de todo o curso, para os três módulos, iremos nos referir a esses princípios. Vamos inicialmente observar uma adaptação da ilustração feita por Paradis (2005), mostrando como o **R** trabalha, Figura 4.2.

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

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

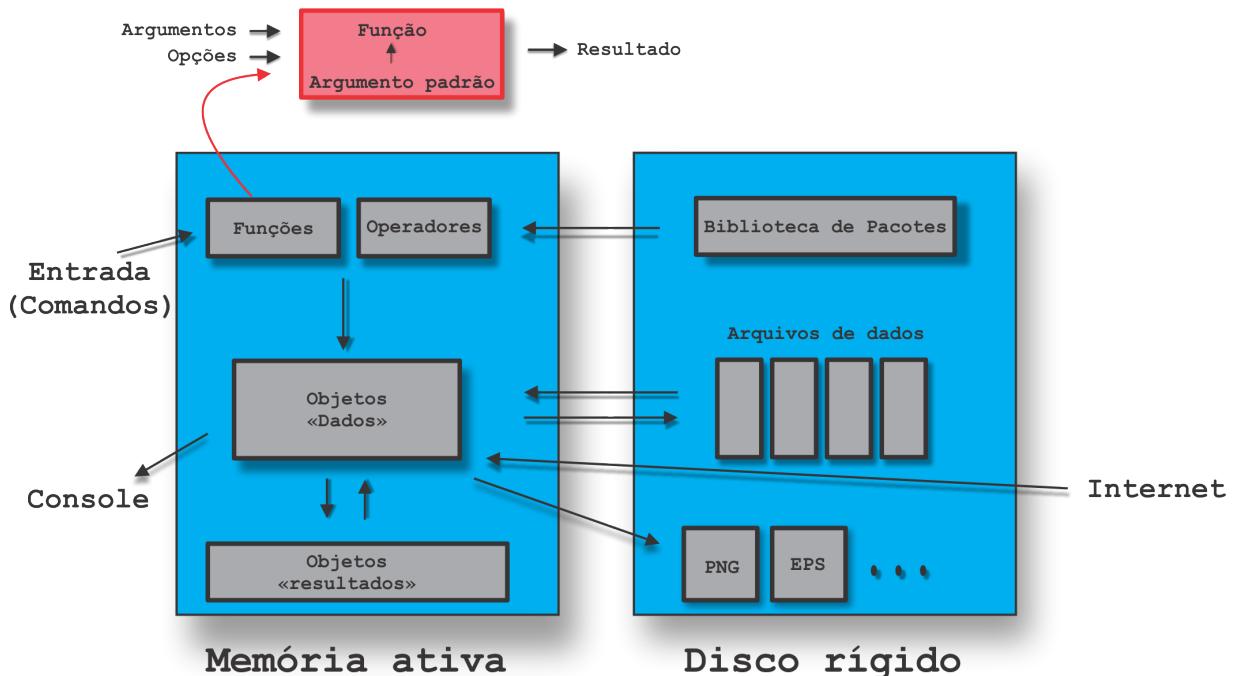


Figura 4.2: Esquema de como o R funciona.

Na Figura 4.2, observamos que todas as ações realizadas sobre os objetos ficam armazenadas na memória ativa do computador. Esses objetos são criados por comandos (teclado ou mouse) através de funções ou operadores (chamada de função), dos quais leem ou escrevem arquivo de dados do disco rígido, ou leem da própria internet. Por fim, o resultado desses objetos podem ser apresentados no console (memória ativa), exportados em formato de imagem, página web, etc. (disco rígido), ou até mesmo ser reaproveitado como argumento de outras funções, porque o resultado também é um objeto.

4.2 Como utilizar o R e o RStudio

A primeira ideia que temos é a linha de comando no R, que é simbolizada pelo *prompt* de comando ">". Este símbolo significa que o R está pronto para receber os comandos do usuário. O *prompt* de comando está localizado no console do R. Vejamos o console do R a seguir, que é o local que recebe as linhas de comando do usuário, Figura 4.3.

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

O RStudio se apresenta como uma interface para facilitar a utilização do R, tendo por padrão quatro quadrantes, apresentados na Figura 4.4.

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

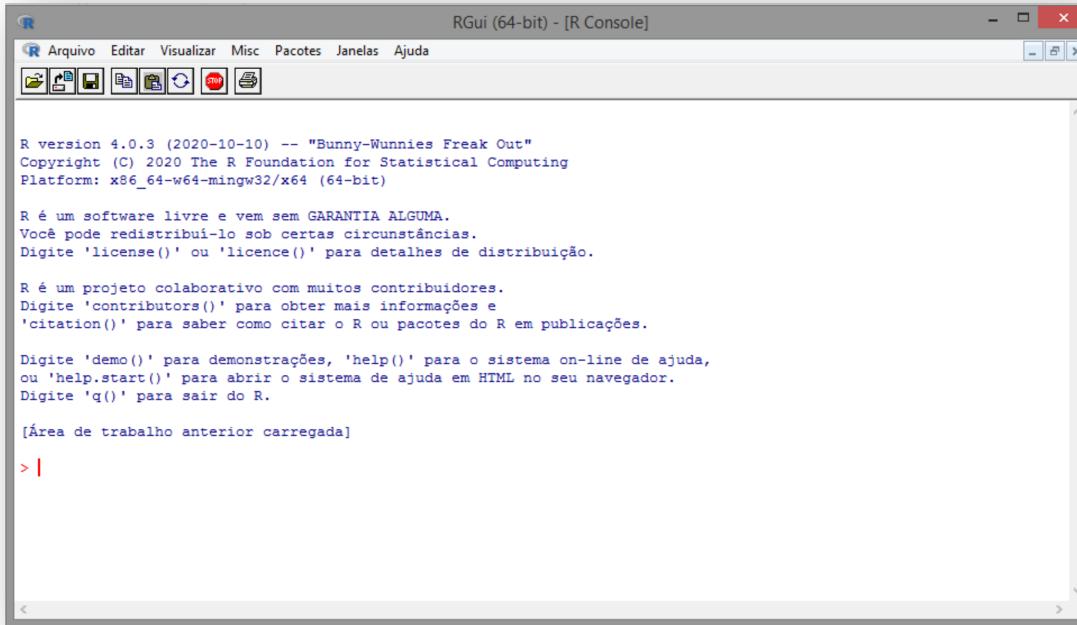


Figura 4.3: Console do R (Versão 4.0.3).

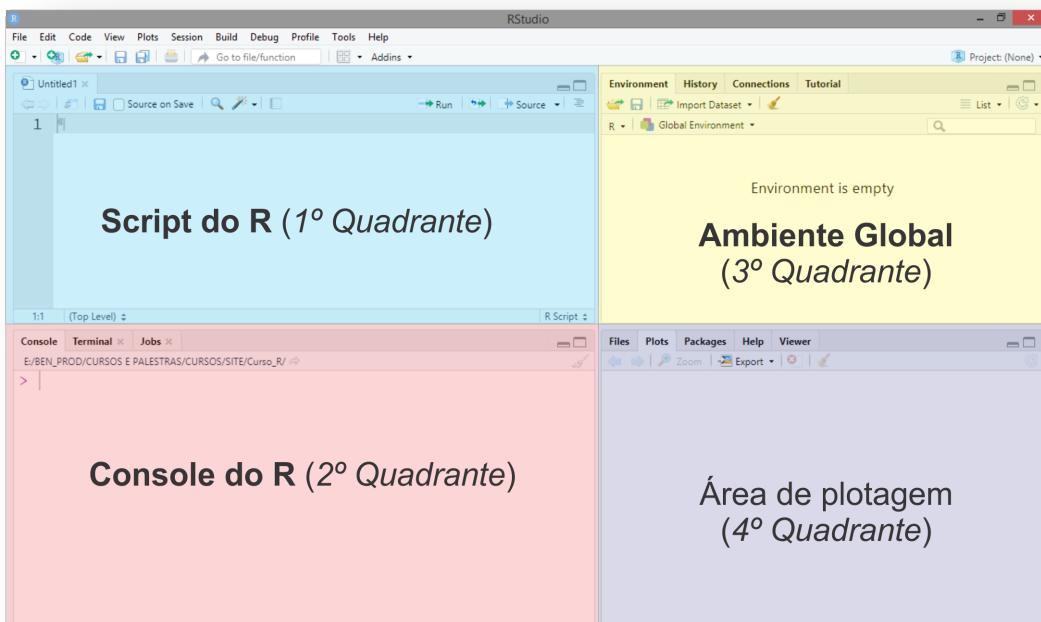


Figura 4.4: Interface do RStudio (Versão 1.4.1103).

4.3 Comandos no R

4.3.1 Console e *Prompt* de comando

Como falado anteriormente, o **R** é uma linguagem baseada em linhas de comando, e as linhas de comando, são executadas uma de cada vez, no *console*. Assim que o *prompt* de comando está visível na tela do *console*, o **R** indica que o usuário está pronto para inserir as linhas de comando. O símbolo padrão do *prompt* de comando é ">", porém ele pode ser alterado. Para isso, vejamos o exemplo do Código R 4.1.

Código R 4.1

Script:

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

Console:

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

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

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

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

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

Console:

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

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

4.3.2 Comandos elementares

Os **comandos elementares** podem ser divididos em **expressões** e **atribuições**. Por exemplo, podemos estar interessados em resolver a seguinte expressão $10 + 15 = 25$. Vejamos o Código R 4.2.

Código R 4.2

Script:

```
1 10 + 15
```

Console:

```
[1] 25
```

No *console* quando passamos pelo comando do Código R 4.2, o **R** avalia essa expressão internamente e imprime o resultado na tela, após apertar o botão *ENTER* do teclado. Esse fato é o que ocorre no segundo princípio mencionado por Chambers (2016), tudo em **R** acontece por uma chamada de função. Na realidade o símbolo `+` é uma função interna do **R**, que chamamos de função primitiva, porque foi implementada em outra linguagem. Assim, esse é o resultado de três objetos (“`10`”, “`+`”, “`15`”) que são avaliados internamente, do qual a função ‘`+(e1, e2)`’ é chamada, e em seguida o resultado é impresso no *console*. Intrinsecamente, podemos também afirmar que a função `print()` também trabalha nessa situação, fazendo o papel de imprimir o resultado no *console*.

Do mesmo modo, se houver algum problema em algum dos objetos o retorno da avaliação pode ser uma mensagem de erro. Um caso muito prático é quando utilizamos o separador de casas decimais para os números sendo a vírgula. Quando na realidade deve ser um ponto “`.`”, respeitando o sistema internacional de medidas que são definidas por padrão no ambiente **R**, e essas configurações podem ser alteradas por meio de `options()`. A vírgula é utilizada para separar elementos, argumentos em uma função, etc. Vejamos um exemplo no Código R 4.3.

Código R 4.3

Script:

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

Console:

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

Porém, tem que ficar claro que uma expressão é qualquer comando repassado no *console*. Este comando é avaliado e seu resultado impresso, há menos que explicitamente o usuário queira torná-lo invisível². Caso algum elemento do comando não seja reconhecido pelo **R**, há um retorno de alguma mensagem em forma de “erro” ou “alerta”, tentando indicar o possível problema. Todos esses processos ocorrem na memória ativa do computador, e uma vez o resultado impresso no *console*, o valor é perdido, há menos que você atribua essa expressão a um nome, que erroneamente usamos o termo: “criamos um objeto!”. A atribuição dessa expressão será dada pela junção de dois símbolos `<-`, falado mais a frente. Um comando em forma de atribuição também avalia a sua expressão, um nome se associa ao seu resultado, e o resultado será mostrado, se posteriormente, após a execução você digitar o “nome” atribuído a esse resultado. Vejamos um exemplo o Código R 4.4.

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

Código R 4.4**Script:**

```

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

```

Console:

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

4.3.3 Execução de comandos

Quando inserimos um comando no console, executamos uma linha de comando por vez ou separados por ";" em uma mesma linha. Vejamos o Código R 4.5.

Código R 4.5**Script:**

```

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

```

Console:

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

Script:

```

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

```

Console:

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

Se um comando for muito grande e não couber em uma linha, ou caso deseje completar um comando em mais de uma linha, após a primeira linha haverá o símbolo "+" iniciando a linha seguinte ao invés do símbolo de prompt de comando (">"), até que o comando esteja sintaticamente completo. Vejamos o Código R 4.6, a seguir.

Código R 4.6**Script:**

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

Console:

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

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

4.3.4 Chamada e correção de comandos anteriores

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

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

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

Quando usamos um comando de atribuição no console, o R armazena o nome associado ao objeto criado na área de trabalho (*Workspace*), que nós chamamos de Ambiente Global. Teremos uma seção introdutória na seção Ambientes e caminhos de busca, mas entendamos inicialmente que o objetivo de um ambiente é associar um conjunto de nomes a um conjunto de valores. Vejamos o Código R 4.7.

Código R 4.7**Script:**

```

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

```

Console:

```

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

```

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

4.5 Arquivos .RData e .Rhistory

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

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

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

4.6 Criando e salvando um *script*

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

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

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

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

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

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

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

Script:

```
1 getwd()
```

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

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

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

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

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

Exercícios propostos

Exercício 4.1:

Solução na página 54

Objetos

5.1 Introdução

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

Script:

```
1 x <- 10
```

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

Representaremos em termos de diagrama, um nome se ligando a um objeto, na Figura 5.1.



Figura 5.1: Dizemos que o nome x se liga ao objeto do tipo (estrutura) vetor.

O identificador na memória ativa desse objeto pode ser obtida por:

Console:

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

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

Essa outra representação, Código R `??`, ficará mais claro para a afirmação feita anteriormente, no segundo diagrama, Figura 5.2, que representa a ligação do nome y ao mesmo objeto. Os termos

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

nos diagramas, serão usados de acordo com a sintaxe da linguagem com os termos em inglês para melhor compreensão e fixação dos termos utilizados em R, uma vez que os termos na linguagem são baseados nesse idioma.

Código R 5.1

Script:

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

Console:

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

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

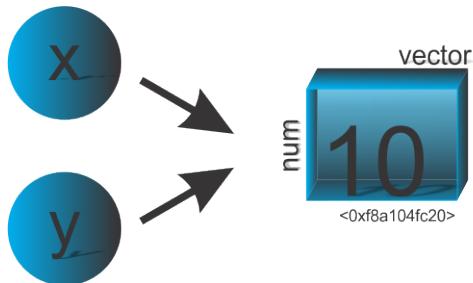


Figura 5.2: Dizemos que o nome x e y se ligam ao objeto do tipo (estrutura) vetor.

Mais especificamente, acrescentamos um outro diagrama, Figura 5.3, mostrando a representação do ambiente global (.GlobalEnv, nome associado ao objeto que representa o ambiente global).

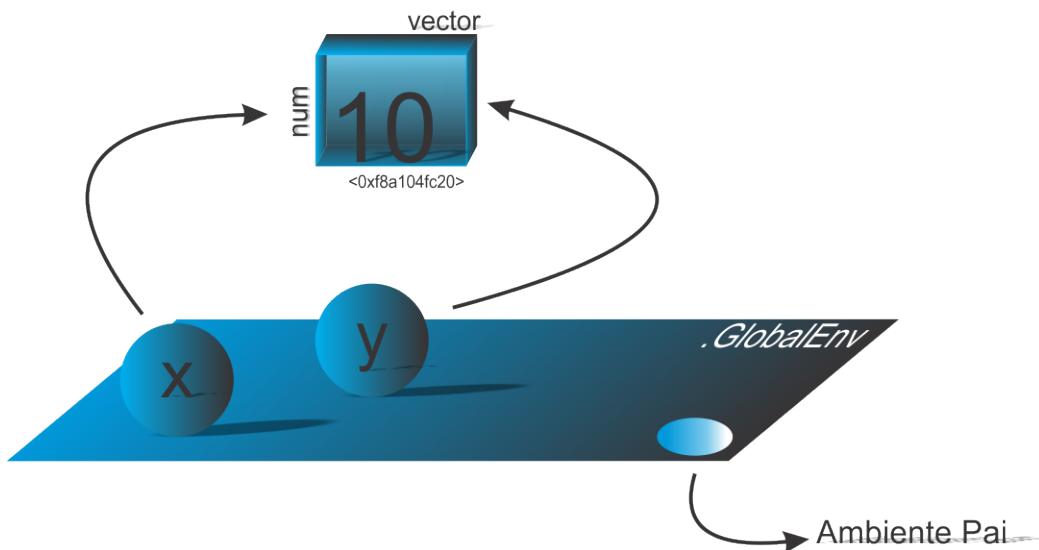


Figura 5.3: Dizemos que o nome x e y se ligam ao objeto do tipo (estrutura) vetor e essa ligação fica armazenada no ambiente global.

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

Script:

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

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

Console:

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

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

Console:

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

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

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

Console:

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

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

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

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

5.2 Atributos

Todos os objetos, terão pelo menos dois tipos de atributos, chamados de atributos intrínsecos. Os demais atributos, quando existem, podem ser verificados pela função `attributes()`. A ideia dos atributos pode ser pensada como *metadados*, isto é, um conjunto de informações que caracterizam o objeto.

Diremos também que todos os objetos R tem uma **classe**, e por meio dessas classes, determinadas funções podem ter comportamento diferente a objetos com classes diferentes. Agora, devemos

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

deixar claro essa informação, apesar do **R** seguir o **princípio do objeto**, nem tudo é orientado a objetos, como por exemplo, observamos na linguagens C++ e Java. Deixemos esse tópico para discorrer no Volume II.

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

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

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

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

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

Por exemplo, quando criamos um objeto da classe `data.frame`, vejamos o que acontece quando removemos esse atributo no Código R 5.2.

Código R 5.2

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

```
$a
[1] 1 2 3

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

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

Script:

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

Console:

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

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

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

5.2.1 Atributos intrínsecos

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

Para determinarmos o **modo** de um objeto, usamos a função `mode()`. Vejamos alguns exemplos pelo Código R 5.3.


Código R 5.3
Script:

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

³sinônimo: `string`, cadeia de caracteres.

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Contudo, essa função `mode()` se baseou nos atributos baseados na linguagem S. Temos uma outra função para verificarmos o **modo** do objeto que é por `typeof()`. O atributo **modo** retornado de um objeto para esta última função, está relacionado a tipagem da linguagem C, Código R 5.4, uma vez que boa parte das rotinas no **R** está nessa linguagem, principalmente as funções do pacote **base**. Existem 24 tipos que serão detalhados no Volume II.

Código R 5.4

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Observamos que apesar de alguns vetores serem vazios, estes ainda tem um modo, observe nas seguintes linhas de comando, no Código R 5.5.

Código R 5.5**Script:**

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

Console:

```
numeric(0)
```

Script:

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

Console:

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

Script:

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

Console:

```
character(0)
```

Script:

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

Console:

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

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

Console:

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

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

Console:

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

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

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

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

- Pode significar um número real, isto é, para a computação um número de dupla precisão (`numeric` e `double` seriam iguais nesse aspecto), Código R 5.6;

Código R 5.6**Script:**

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

Console:

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

Script:

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

Console:

```
[1] TRUE
```

- nos sistemas S3 e S4 (orientação a objetos), o termo numérico é usado como atalho para o modo `integer` ou `double`. Esse ponto veremos Volume II. Contudo, vejamos o Código R 5.7;

Código R 5.7**Script:**

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

Console:

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

Script:

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

Console:

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

- Pode ser utilizado (`is.numeric()`) para verificar se determinados objetos tem o modo numérico. Por exemplo, temos um objeto de classe factor que é importante para a área da estatística experimental, representando os níveis de um fator em um experimento. Os elementos desse objeto pode ser número ou caracteres, mas serão representados como sempre por números. Entretanto, não se comportam como numérico, Código R 5.8;

Código R 5.8**Script:**

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

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

Script:

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

Console:

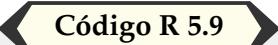
```
[1] FALSE
[1] FALSE
```

A Tabela 5.1 a seguir, mostra o retorno dos seis principais modos de um objeto do tipo (estrutura) de vetores atômicos (Os modos apresentados baseiam-se apenas quanto a característica dos dados do objeto. É claro que um objeto não armazena apenas dados. Existem outras naturezas, que serão omitidas nesse momento).

Tabela 5.1: Tipagem dos vetores.

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

O **comprimento** do objeto é informado pela função `length()`, do qual a representação em diagrama informa esse atributo. Vejamos as linhas de comando no Código R 5.9.


Código R 5.9
Script:

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

Console:

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

Um diagrama apresentando esses três objetos no ambiente global, pode ser apresentado na Figura 5.4. Observe que acrescentamos agora o **comprimento** dos objetos no diagrama entre colchetes, ao lado do atributo **modo**.

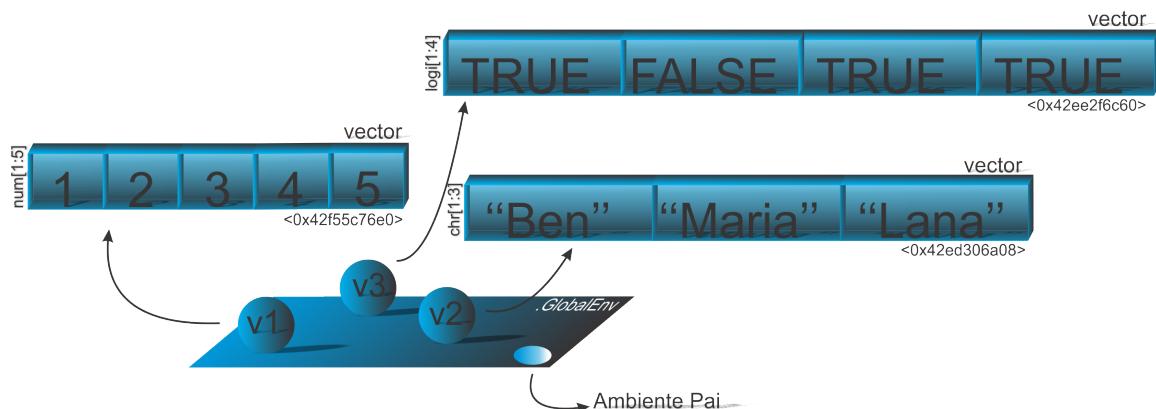


Figura 5.4: Objetos v1, v2 e v3.

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

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

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

5.3 Coerção

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

Console:

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

Observe que todos os elementos ganharam aspas, isto é, se tornaram um caractere ou uma cadeia de caracteres. A coerção entre vetores de modo numeric, character e logical será sempre como verificado pela Figura 5.5.

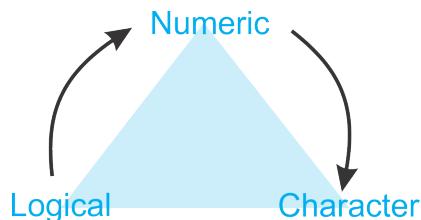


Figura 5.5: Coerção de vetores com tipagem numeric, character e logical.

No caso dos vetores lógicos, todo TRUE se converterá em 1, e FALSE em 0. Porém, os modos dos vetores podem ser coagidos pelo usuário, usando as funções do tipo “as.<modo ou tipo>()” com prefixo “as.”, isto é, se desejarmos que um objeto meu_objeto tenha o modo “character”, basta usar as.character(meu_objeto). Para desejar saber se um objeto é de um determinado modo, usamos as funções do tipo is.<modo ou tipo>(), com o prefixo “is.”. Vejamos o Código R 5.10, para elucidar o que discutimos anteriormente.

Código R 5.10

Script:

```

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

```

Console:

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

Script:

```

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

```

Console:

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

Script:

```

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

```

Console:

```
[1] TRUE
```

5.4 Tipo de objetos

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

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

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

Daremos uma visão geral dos objetos apresentados até o momento na Tabela 5.2.

5.4.1 Vetores

Podemos dizer que existem três tipos principais de vetores atômicos:

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

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

5.4.1.1 Vetores escalares ou constantes

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

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

Tabela 5.2: Caracterização de objetos estruturado para armazenamento de dados.

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

Qual a diferença entre NaN, NA e NULL?

As palavras reservadas `NaN` (do inglês, *Not a Number*) e `NA` (do inglês, *Not Available*) representam valores ausentes, com uma grande diferença, `NaN` apesar do nome é um tipo de valor numérico não representável na aritmética de pontos flutuantes (GOLDBERG, 1991). Esse termo foi introduzido por em 1985 pelo Instituto de Engenheiros Elétricos e Eletrônicos (IEEE), para definir as representações de outras quantidades não finitas, como por exemplo os infinitos. No **R**, para verificarmos se um número é finito ou infinito, usamos respectivamente, `is.finite()` e `is.infinite()`^a. O primeiro retorna todos os não infinitos e não ausentes. Contudo, `NA` se refere a qualquer valor não ausente, que não necessariamente seja numérico. Como podemos comprovar isso, pela pirâmide de coerção na Figura 5.5. Vejamos o Código R 5.11. Percebemos no primeiro caso que `NaN` tem um comportamento de número, com modo `double`^b, então é coagido a caractere, e perde a sua natureza de número. Já `NA` é do tipo lógico, portanto, pode ser um valor ausente para qualquer natureza de vetor, seja `numeric`, `logical` ou `character`, e nesse último caso, não ocorre coerção, apenas a informações de que o primeiro elemento do vetor está ausente, porém, o vetor ainda continua sendo de modo `character`.

Desse modo, nós vamos perceber `NaN` em operações matemáticas que a solução é indeterminada, tais como pode ser visto no Código R `??`. Porém, quando um conjunto de dados é apresentado e desejamos representar um valor ausente, é preferível `NA`. Operações realizadas com `NaN` ou `NA`, retornam `NaN` ou `NA` na maioria das vezes, como pode ser visto no Código R 5.12. Agora observemos como é interessante a ideia criada para essas palavras reservadas. Sabemos que qualquer valor de potência zero é sempre igual a 1, seja um número positivo ou negativo. Então essa ideia também segue ao ambiente **R**, pois apesar de não sabermos o valor ausente, esse valor elevado a zero será 1, como pode ser verificado no Código R 5.13.

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

Por fim, apresentamos a distinção do objeto `NULL`, isso mesmo um objeto de tipo `NULL`, mais precisamente `NILSXPc`, e também uma palavra reservada. Diferentemente de `NaN` e `NA` que é um vetor de comprimento 1. Nos manuais do **R**, é dito que o objeto `NULL` aparece sempre em funções ou expressões cujos resultados são não definidos. Esse objeto é o único no **R** que não tem atributo, sendo muito usado em argumentos padrão em funções quando inicialmente não se define nada para o referido argumento. Para verificar se um objeto é `NULL`, usamos `is.null()` e coagimos por `as.null()`. Agora, apresentamos uma diferença básica entre `NULL`, `NaN` e `NA`, em que estes últimos quando definidos em um objeto, apesar do valor ausente ou perdido, é sabido que existe o valor. Assim, na contabilização do número de elementos do vetor, por exemplo, `NaN` ou `NA` é contabilizado. No caso, `NULL` representa valor não existente, e como não há atributo envolvido, este não é contabilizado. Observe no Código R 5.15.

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

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

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



Código R 5.11**Console:**

```
> c(NaN, "a")
[1] "NaN" "a"
> c(NA, "a")
[1] NA "a"
```

Código R 5.12**Console:**

```
> sqrt(-1)
[1] NaN
Warning message:
In sqrt(-1) : NaNs produced
> 0 / 0
[1] NaN
> Inf - Inf
[1] NaN
```

Código R 5.13**Console:**

```
> NA^0
[1] 1
> NaN^0
[1] 1
```

Código R 5.14**Console:**

```
> NA + 1
[1] NA
> NaN + 5
[1] NaN
> NA * 5
[1] NA
> sqrt(NaN)
[1] NaN
> NaN + NA
[1] NaN
> NA + NaN
[1] NA
```

Código R 5.15**Console:**

```
> length(c(NA, 1, 2))
[1] 3
> length(c(NaN, 1, 2))
[1] 3
> length(c(NULL, 1, 2))
[1] 2
```

5.4.1.2 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 5.16.

Código R 5.16**Script:**

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

Console:

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

Script:

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

Console:

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

Uma coisa interessante é que por padrão, a função `c()` sempre cria um vetor de modo `double`, a menos que o usuário determine que estes elementos sejam inteiros, como pode ser visto no Código R 5.17.

Código R 5.17**Script:**

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

Console:

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

Script:

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

Console:

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

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

Console:

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

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

Console:

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

O que vai diferenciá-los é a forma de armazená-lo (double ou integer), e por consequência, o espaço na memória ativa, como podemos observar no Código R 5.18.

Código R 5.18**Script:**

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

Console:

```
96 B
```

Script:

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

Console:

```
96 B
```

Script:

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

Console:

```
96 B
```

O que podemos observar é que o vetor de modo `double` precisa de mais memória para armazenar os valores do que o objeto de modo `integer`. O último objeto, gerado pela chamada ‘`:`()`, aparentemente ocupa mais memória. Porém, essa função apresenta um recurso interessante apresentado nas versões posteriores **R** (3.5.0), que é chamado de **abreviação alternativa**. Esse recurso faz com que a sequência de números não seja armazenada completamente, apenas os extremos. Isso significa que para qualquer tamanho de sequência, a ocupação de memória do objeto será sempre a mesma. Lembrando que essa sequência sempre terá o modo `double` na tipagem C. Outras formas de criar sequências de números é usando as funções `rep()`, `rep_len()` (mais rápido), `seq()`, `seq_along()` (mais rápido) e `seq_len()` (mais rápido), `sequence()`, `replicate()`, `gl()`, e que pode ser observado no Código R 5.19.

Código R 5.19**Console:**

```
> # Repete o numero 2 tres vezes
> rep(x = 2, times = 3)
[1] 2 2 2
> # Repete o vetor 1:3 tres vezes
> rep(x = 1:3, times = 3)
[1] 1 2 3 1 2 3 1 2 3
> # Repete cada numero do vetor, tres vezes
> rep(x = 1:3, each = 3)
[1] 1 1 1 2 2 2 3 3 3
> # Repete cada numero do vetor duas vezes,
> # porem, o comprimento dessa sequencia esta
> # limitado a 4
> rep(1:3, each = 2, length.out = 4)
[1] 1 1 2 2
> # O vetor eh repetido ate obter uma
> # sequencia de tamanho 7
> rep(x = 1:3, length.out = 7)
[1] 1 2 3 1 2 3 1
> # (Versao mais rapida de rep) O vetor eh repetido ate obter uma
> # sequencia de tamanho 15
> rep_len(x = 1:10, length.out = 15)
[1] 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5
```

Console:

```

> # Sequencia criada de 1 a 2, espacada em 0.1
> seq(from = 1, to = 2, by = 0.1)
[1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
> # Sequencia criada de 1 a 10, espacada em 1
> seq(from = 1, to = 10, by = 1)
[1] 1 2 3 4 5 6 7 8 9 10
> # Sequencia criada de 1 a 10 de forma equisepacada
> # de comprimento 20
> seq(from = 1, to = 10, length.out = 20)
[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
> # Eh o mesmo que 1:length(y)
> y <- rnorm(10)
> seq(along.with = y)
[1] 1 2 3 4 5 6 7 8 9 10
> # Sequencia de 1 a 20
> seq(20)
[1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
[17] 17 18 19 20
> # Sequencia criada de 10 a 100 de mesmo comprimento
> # de x
> x <- 1:10
> seq(from = 10, to = 100, along.with = x)
[1] 10 20 30 40 50 60 70 80 90 100
> # (Versao mais rapida para seq) Eh o mesmo que
> # 1:length(w)
> w <- c(4, 3, 6, 9)
> seq_along(w)
[1] 1 2 3 4
> # (Versao mais rapida para seq) Eh o mesmo que 1:4
> seq_len(4)
[1] 1 2 3 4

```

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

Console:

```

> # Tamanho de memoria dos objetos
> lobstr::obj_size(1:10)
680 B
> lobstr::obj_size(1:10000)
680 B
> lobstr::obj_size(1:100000)
680 B
> lobstr::obj_size(c(1:10))
96 B
> lobstr::obj_size(c(1:10000))
40,048 B
> lobstr::obj_size(c(1:100000))
4,000,048 B

```

5.4.1.3 Manipulando vetores

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

Console:

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

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

Console:

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

Exercícios propostos

Exercício 5.1:

Solução na página 55

Gabarito dos Exercícios

Solução dos Exercícios do Capítulo 1

Solução do Exercício 2.1 na página 7:

A resposta está relacionada ao escopo léxico das funções e a superatribuição! Ainda não entendeu? Então leia os Capítulos 3 e 5. Para uma maior profundidade, veja o Capítulo sobre ambientes no Volume II.

Solução do Exercício 2.2 na página 7:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.3 na página 8:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.4 na página 8:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.6 na página 8:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.6 na página 8:

Para entender, leia o Capítulo 4.

Solução do Exercício 2.7 na página 8:

Se respondeu 1, leia todo o livro.

Solução do Exercício 2.8 na página 8:

Para entender, leia o Capítulo 3.

Solução do Exercício 2.9 na página 8:

Leia todo o livro.

Solução do Exercício 2.10 na página 8:

Para entender, leia o Capítulo 5.

Solução do Exercício 2.11 na página 8:

Para entender, leia o Capítulo 5.

Solução do Exercício 2.12 na página 8:

Para entender, leia o Capítulo 4.

Solução do Exercício 2.13 na página 8:

Para entender, leia o Capítulo 5.

Solução do Exercício 2.14 na página 8:

Para entender, leia o Capítulo 5.

Solução do Exercício 2.15 na página 9:

Para entender, leia o Capítulo 5.

Solução do Exercício 2.17 na página 9:

Para entender, leia o Capítulo 5.

Solução do Exercício 2.17 na página 9:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.18 na página 9:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.19 na página 9:

Para entender, leia o Capítulo ??.

Solução do Exercício 2.20 na página 9:

Para entender, leia os Capítulos 4 e ??.

Solução do Exercício 2.21 na página 9:

Para entender, leia os Capítulos 4 e ??.

Solução do Exercício 2.22 na página 9:

Para entender, leia os Capítulos 5.

Solução dos Exercícios do Capítulo 3

Solução dos Exercícios do Capítulo 4

Solução do Exercício 4.1 na página 27:

Solução dos Exercícios do Capítulo 5

Solução do Exercício 5.1 na página 50:

Referências Bibliográficas

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

- R CORE TEAM. *Writing R extensions*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- SILGE, J.; ROBINSON, D. *Text mining with R*. O'Reilly Media, 2017. ISBN 978-1491981658. Disponível em: <<https://www.tidytextmining.com/>>.
- VENABLES, W. N.; SMITH, D. M.; R CORE TEAM. *An Introduction to R: Notes on R a programming environment for data analysis and graphics*. Vienna, Austria, 2021. Version 4.1.1 (10-08-2021). Disponível em: <<https://cran.r-project.org/manuals.html>>.
- WICKHAM, H. *R Packages*. 2nd. ed. Sebastopol: O'Reilly Media, 2015. ISBN 9781491910597. Disponível em: <<https://r-pkgs.org/index.html>>.
- WICKHAM, H. *ggplot2: Elegant graphics for data analysis*. 2. ed. New York: Springer, 2018. 276 p. (Use R!). ISBN 978-3319242750. Disponível em: <<https://ggplot2-book.org/>>.
- WICKHAM, H. *Advanced R*. 2nd. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2019. ISBN 978-0815384571. Disponível em: <<https://adv-r.hadley.nz/>>.
- WICKHAM, H. *Mastering shiny*. O'Reilly Media, 2021. ISBN 978-1492047384. Disponível em: <<https://mastering-shiny.org/>>.
- WICKHAM, H.; GROLEMUND, G. *R for Data Science*. Sebastopol: O'Reilly Media, 2017. ISBN 978-1-491-91039-9. Disponível em: <<https://r4ds.had.co.nz/>>.
- WILKE, C. O. *Fundamentals of data visualization: A primer on making informative and compelling figures*. Sebastopol: O'Reilly Media, 2016. 390 p. ISBN 978-3319242750. Disponível em: <<https://ggplot2-book.org/>>.
- XIE, Y. *Dynamic documents with R and knitr*. 2. ed. Boca Raton, Florida: Chapman and Hall/CRC, 2015. ISBN 978-1498716963. Disponível em: <<https://yihui.org/knitr/>>.
- XIE, Y. *bookdown: Authoring books and technical documents with r markdown*. Boca Raton, Florida: Chapman and Hall/CRC, 2017. ISBN 978-1138700109. Disponível em: <<https://bookdown.org/yihui/bookdown/>>.
- XIE, Y.; ALLAIRE, J. J.; GROLEMUND, G. *R Markdown: The definitive guide*. Chapman and Hall/CRC, 2018. ISBN 978-1449359119. Disponível em: <<https://bookdown.org/yihui/rmarkdown/>>.
- XIE, Y.; THOMAS, A.; HILL, A. P. *blogdown: Creating websites with r markdown*. Boca Raton, Florida: Chapman and Hall/CRC, 2018. 172 p. (The R Series). ISBN 978-0815363729. Disponível em: <<https://bookdown.org/yihui/blogdown/>>.

Índice Remissivo



Índice Remissivo

-
- .RData, 8
 - .Rhistry, 8
 - abreviação alternativa, 47
 - Ambiente
 - chamada, 8, 9
 - envolvente, 8, 9
 - execução, 9
 - Anexar, 8
 - Atribuição, 22, 28
 - Atributos, 9, 28, 30
 - class, 9
 - intrínsecos, 9, 32
 - comprimento, 32
 - modo, 28, 32
 - nome, 22
 - Caminho de busca, 8
 - Carregar, 8
 - Chamada de função, 18, 22
 - Classe, 31
 - Classe implícita, 31
 - Comandos elementares, 22
 - Console, 19, 21, 22, 24
 - Escopo
 - léxico, 49
 - Estudando o ambiente R, 3
 - demais volumes, 5
 - volume I, 4
 - volume II, 4
 - volume III, 4
 - Execução de comandos, 23
 - Expressão, 22
 - Funções
 - anônimas, 8
 - internas de um pacote, 8
 - primitiva, 22
 - Linguagem
 - C, 4, 9
 - C++, 9
 - FORTRAN, 4, 9
 - HTML, 9
 - Java, 9
 - Julia, 9
 - Python, 9
 - S, 4, 11
 - John McKinley Chambers, 18
 - Linhas de comando, 19, 21
 - Memória ativa, 19
 - Namespace
 - arquivo, 8
 - Nomes, 21
 - não sintáticos, 21
 - sintáticos, 21
 - Objeto, 8
 - lista, 8
 - data frame, 8
 - vetor, 8
 - array, 8
 - atômico, 28, 38
 - matriz, 8
 - Pacotes, 6
 - abind, 6
 - base, 39
 - blogdown, 6
 - bookdown, 6
 - codetools, 6
 - distill, 6
 - formatR, 6
 - ggplot2, 6
 - knitr, 6
 - learnr, 6
 - lobstr, 6, 11, 46
 - pryr, 6
 - rlang, 6
 - rmarkdown, 6, 15
 - shiny, 6, 15

- sloop, 6, 36
- styler, 6
- XR, 6
- Princípio
 - função, 18
 - Interface, 9
 - interface, 9, 18
 - objeto, 18, 31
- Prompt de comando, 19, 21
- R, 8
 - software* livre, 11
 - ambiente, 1, 3
 - artigos, 3
 - banco de dados, 3
 - CRAN, 10
 - primeiro espelho no Brasil, 15
 - código aberto, 11
 - dashboards, 3
 - escopo, 11
 - gerenciamento de memória, 11
 - gráficos, 3
 - história, 10
 - IGU, 3
 - Instalação, 16
 - interface, 15
 - linguagem, 1, 3
 - livros, 3
 - materiais didáticos, 6
 - paralelização, 3
 - programa, 1, 3
 - programação defensiva, 3
 - R Core Team, 10
 - semântica, 3
 - sintaxe, 3
 - três princípios, 8
 - websites, 3
- RStudio, 8, 19
 - Instalação, 16
 - J. J. Allaire, 15
 - quadrantes, 19
 - primeiro, 19
 - quarto, 19
 - segundo, 19
 - terceiro, 19
- Script, 8
- Superatribuição, 7, 49
- Tipagem
 - Linguagem C, 33
 - linguagem S, 32
- Tópico, 2
 - subtopico, 2
 - subsubtopico, 2

