Programação em R
Parte I

Autor: Prof. Dr. Pedro Rafael Diniz Marinho

Universidade Federal da Paraíba
Departamento de Estatística da UFPB

Plano de Curso

A ementa está disponível no site do Departamento de Estatística da UFPB.

O plano de curso é de responsabilidade do professor da disciplina e deve estar de acordo com a ementa da disciplina.

O plano de curso é estabelecido pelo professor e ficará a cargo desse a sua construção e alteração, se necessário, no decorrer do curso, desde que o mesmo esteja de acordo com a ementa do curso.

O rigor e profundidade do assunto ficará a cargo do professor e faz parte do seu plano de curso.

Fiquem atentos ao SIGAA UFPB

É aconselhável que o aluno acesse a disciplina no SIGAA - UFPB (SIGAA) com frequência. Todo tipo de notícia, informações e materiais de apoio serão postados no SIGAA.

Fiquem atentos ao SIGAA UFPB

É aconselhável que o aluno acesse a disciplina no SIGAA - UFPB (SIGAA) com frequência. Todo tipo de notícia, informações e materiais de apoio serão postados no SIGAA.

Caso seja necessário entrar em contato com os alunos sobre reposições de aulas e/ou não ocorrência de alguma aula, estas informações estarão, **SEM FALTA**, registradas no SIGAA.

Fiquem atentos ao SIGAA UFPB

É aconselhável que o aluno acesse a disciplina no SIGAA - UFPB (SIGAA) com frequência. Todo tipo de notícia, informações e materiais de apoio serão postados no SIGAA.

Caso seja necessário entrar em contato com os alunos sobre reposições de aulas e/ou não ocorrência de alguma aula, estas informações estarão, **SEM FALTA**, registradas no SIGAA.

Inclusive, todas as aulas (arquivos PDF's utilizados na projeção) serão postados na plataforma.

Recursos utilizados

Boa parte do curso será apoiada pelo uso de **datashow** o que nos ajudará bastante a decorrer sobre os diversos assuntos contidos na ementa desse curso que é bastante ampla.

O quadro será utilizado para resolução de alguns exemplos bem como complementações em que o professor achar conveniente no momento de aula.

Sobre as avaliações

No curso iremos considerar **três avaliações** (provas).

As datas das avaliações estão registradas no SIGAA.

O aluno não poderá utilizar o computador nas avaliações.

Aos alunos interessados (LaTeX)

Figura: Donald Knuth.

TeX é um sistema de tipografia científica desenvolvido por **Donald E. Knuth** que é orientado à produção de textos técnicos e fórmulas matemáticas. A pedido da AMS (*American Mathematical Society*), Donaldh Knuth desenvolveu uma linguagem de computador para editoração de textos com muitas equações.

Aos alunos interessados (LaTeX)

Figura: Donald Knuth.

O trabalho de criação do TeX se estendeu de 1977 a 1998, quando TeX foi disponibilizado gratuitamente. O TeX possui aproximadamente **600 comandos** que controlam a construção de uma página.

Pode-se considerar o TeX como sendo um compilador para textos científicos que produz documentos de alta qualidade tipográfica.

Aos alunos interessados (LaTeX)

O TeX atingiu um estado de desenvolvimento em que Beebe (1990 afirmou):

> "Meu trabalho no desenvolvimento de TeX, METAFONT e as fontes Computer Modern chegou ao final. Eu não irei realizar mudanças futuras, exceto corrigir sérios erros de programação."

Ver em: BEEBE, N. H. Comments on the future of TeX and METAFONT. TUGboat, v. 11, n. 4, p. 490–494, 1990.

Aos alunos interessados (LaTeX)

Figura: Leslie Lamport.

Quase que em paralelo foi desenvolvido por Leslie Lamport o LaTeX. Essas macros definem tipos de documentos, tais como livros, artigos, cartas, entre outros.

Inclusive essa apresentação é um tipo básico de documento que foi produzido em LaTeX.

Aos alunos interessados (LaTeX)

Figura: Leslie Lamport.

Para maiores detalhes leia sobre o pacote `beamer` que está disponível com a maioria das distribuições LaTeX. Há diversos documentos disponíveis nos mais variados idiomas na rede.

`beamer` também está disponível no **The Comprehensive TeX Archive Network (CTAN)**.

Aos alunos interessados (LaTeX)

Figura: CTAN lion drawing by Duane Bibby.

CTAN é o lugar central para todos os tipos de material em torno de TeX. CTAN tem atualmente **5263 pacotes** e **2409 colaboradores** contribuiram para essa quantidade de pacotes.

O símbolo ao lado foi desenhado pelo artista comercial Duane Bibby. Este leão foi utilizado nas ilustrações para o livro TeXBook de Donald Knuth e apareceu com grande frequência em outros materiais.

Aos alunos interessados (LaTeX)

Figura: CTAN lion drawing by Duane Bibby.

Maiores detalhes sobre o CTAN podem ser encontrados em https://www.ctan.org/lion/. Desde dezembro de 1994, a pacote LaTeX está sendo atualizado pela equipe LaTeX 3, dirigida por Frank Mittelbach, para incluir algumas melhorias que já vinham solicitadas a algum tempo. A equipe se preocupa também em reunificar todas as versões modificadas que surgiram desde o aparecimento do LaTeX 2.09.

Aos alunos interessados (LaTeX)

Figura: CTAN lion drawing by Duane Bibby.

O melhor de tudo, o LaTeX é um sistema estável mas com crescimento constante, podendo ser instalado em quase todos os sistemas operacionais.

Usuários de Unix, Linux, Windows ou Mac OS X podem dispor de todo ferramental para produzir ótimos textos com o LaTeX.

Nota: Pronuncia-se "leitec" e não "latéx".

Aos alunos interessados (LaTeX)

Mac™ OS

Como instalo o LaTeXno Linux?

Figura: Tux (Mascote do Linux).

Como instalo o LaTeXno Linux?

Inicialmente é preciso instalar o compilador de LaTeX. Recomento o uso do TeX Live.

A maioria das distribuições linux (Arch, Ubuntu, Fedora, Mint, Sabayon, entre outros "sabores") apresentam esse compilador de LaTeX em seus repositórios.

Por exemplo, no **Arch Linux** e distribuições derivadas que utilizam os mesmos repositórios do Arch como Antergos façam:

```
sudo pacman -S texlive.
```

Como instalo o LaTeXno Linux?

No **Ubuntu** ou qualquer distribuição que faz uso dos repositórios do Ubuntu façam:

```
sudo apt-get install texlive-full.
```

Já os usuários da distribuição **Fedora** e distribuições derivadas que utilizam-se dos mesmos repositórios devem fazer:

```
sudo dnf -y texlive-scheme-full.
```

Observação: Todos os comandos acima devem ser executados no terminal da respectiva distribuição com permissão de super usuário (usuário que pode fazer alterações no sistema operacional).

Como instalo o LaTeX no Windows?

Como instalo o LaTeX no Windows?

Felizmente, há o TeXLive para Windows que poderá ser obtido no site oficial do projeto TeXLive.

O usuário de Windows deverá baixar o arquivo

`install-tl-windows.exe`

que possui aproximadamente **13mb**.

Nota: `install-tl-windows.exe` é apenas o instalador do TeXLive para Windows. Dessa forma, ao final da instalação, o TeXLive terá muito mais que apenas 13mb instalado em seu computador.

Como instalo o LaTeXno Linux?

Mas para escrevermos um texto com qualidade usando o LaTeX precisamos também de um editor de texto.

Na maioria dos casos usamos uma **IDE** (*Integrated Development Environment*) (**Ambiente de Desenvolvimento Integrado**)

Aconselho o uso do TeXstudio que está disponível para Linux, Windows e Mac OS.

O TeXstudio é um software sobre os termos da licença **GPL** (*GNU General Public License*) e pode ser obtido em http://texstudio.sourceforge.net/.

Observação: Aperte F6 para compilar o documento e F7 para visualizar o PDF produzido por meio de código LATEX.

Figura: IDE para escrever códigos em LATEX (TEXstudio).

Vantagens do TEXstudio

Vantagens do TEXstudio

1. É uma IDE leve.

Vantagens do TEXstudio

1. É uma IDE leve.
2. O TEXstudio auto-completa os comandos que vão sendo digitados.

Vantagens do TEXstudio

1. É uma IDE leve.
2. O TEXstudio auto-completa os comandos que vão sendo digitados.
3. É possível ir de um ponto específico do código LATEX para o ponto correspondente no PDF criado.

Vantagens do TEXstudio

1. É uma IDE leve.
2. O TEXstudio auto-completa os comandos que vão sendo digitados.
3. É possível ir de um ponto específico do código LATEX para o ponto correspondente no PDF criado.
4. Tem licença **GPL** (*GNU General Public License*), isto é, não pago por ela e tenho acesso ao seu código fonte.

Vantagens do TEXstudio

1. É uma IDE leve.
2. O TEXstudio auto-completa os comandos que vão sendo digitados.
3. É possível ir de um ponto específico do código LATEX para o ponto correspondente no PDF criado.
4. Tem licença **GPL** (*GNU General Public License*), isto é, não pago por ela e tenho acesso ao seu código fonte.
5. O PDF é visualizado ao lado do código LATEX.

Como aprender LaTeX?

Pergunta do aluno

Okay professor, o senhor me convenceu em utilizar LaTeX. Então, como eu posso aprender os seus comandos? Há algum material interessante para se começar a estudar LaTeX?

Como aprender LaTeX?

Pergunta do aluno

Okay professor, o senhor me convenceu em utilizar LaTeX. Então, como eu posso aprender os seus comandos? Há algum material interessante para se começar a estudar LaTeX?

Resposta do professor legal

Há vários livros e materiais disponibilizados na internet fora a referência considerada nessa disciplina. Há também diversos grupos de discussão sobre LaTeX a exemplo do grupo LaTeX-br que poderá ser encontrados nos Grupos do Google. Um bom material em português sobre LaTeXpoderá ser acessado no link http://www.mat.ufpb.br/lenimar/textos/breve21pdf.zip.

Como eu sei se o LaTeX instalou corretamente em meu computador depois de seguir os passos acima? Reposta: Simples, abra o TeXstudio e digite o comando abaixo e aperte F6 e depois aperte F7 para visualizar o documento.

```
1: \documentclass[a4paper,12pt]{report}
2:
3: \begin{document}
4: Um contato superficial com o \LaTeX. Um amor ao
5: primeiro uso ...
6: \end{document}
```

Observação: Voltaremos a falar sobre LaTeX, porém, conversaremos, antes, um pouco sobre programação e a importância que um estatístico deverá dar à esse conhecimento.

O que é programação?

Programação

O que é programação?

Resposta: Linguagem de programação é um método padronizado para comunicar instruções para um computador por meio de uma sintaxe.

Programação

O que é programação?

Resposta: Linguagem de programação é um método padronizado para comunicar instruções para um computador por meio de uma sintaxe.

Trata-se de um conjunto de regras sintáticas utilizadas para passar instruções para um computador. Por meio dessas regras, é possível que o programador especifique os **tipos de dados** em que o professador irá processar.

Tais dados serão armazenados e/ou transmitidos entre os componentes que forma o computador. Assim, a linguagem também permite especificar quais ações devem ser tomadas e sob quis circunstâncias serão tomadas.

Linguagens

Figura: Diversas linguagens de programação disponíveis para uso.

Por que aprender à programar?

Por que um estatístico deve saber programar?

Por que aprender à programar?

Por que o estatístico deve saber programar?

Resposta: Simplesmente pelo fato de que **não** dá para fazer muita coisa na estatística (principalmente no mercado de trabalho) se o profissional não é capaz de fazer com que o computador resolva os seus problemas.

Por que aprender à programar?

Por que o estatístico deve saber programar?

Resposta: Simplesmente pelo fato de que **não** dá para fazer muita coisa na estatística (principalmente no mercado de trabalho) se o profissional não é capaz de fazer com que o computador resolva os seus problemas.

Um aluno questiona...

Mas professor, temos o SPSS, Excel, SAS, Statistica e outros softwares em que podemos chamar nosso conjunto de dados e apertar centenas de botões e ter **alguns** resultados.

Esclarecedor

"**Ciência da computação** tem tanto a ver com o computador como a Astronomia com o telescópio, a Biologia com o microscópio, ou a Química com os tubos de ensaio. A Ciência não estuda ferramentas, mas o que fazemos e o que descobrimos com elas.- **Edsger Dikstra** (Prêmio Turing em 1972)

Figura: Edsger Dikstra.

Esclarecedor

A frase empregada pelo Edsger Dikstra seria perfeitamente válida se substituirmos a expressão **"Ciência da computação"** por **"Estatística Computacional"**. Porém, não há como se utilizar Estatística Computacional se não dominarmos alguma linguagem de programação e em muitas situações fica difícil desvincular as metologias da estatística computacional da teoria da programação.

Por essa dificuldade de se separar a teoria da estatística computacional da teoria de linguagem de programação é que faz com que a ementa da disciplina contemple o tópico de programação em R, não só aqui, como em diversos outros cursos de estatística computacional.

Máquina de Turing

A máquina de Turing é um dispositivo teórico conhecido como máquina universal, que foi concebido pelo matemático britânico Alan Turing (1912-1954), muitos anos antes de existirem os modernos computadores digitais.

Figura: Exemplo de uma Turing física.

Máquina de Turing

Na Teoria da Computabilidade, um problema é solúvel se há uma Máquina de Turing para aquele problema

Em seu artigo original, Turing demonstra a existência de um **problema insolúvel**.

Existe basicamente dois tipos de máquinas de Turing:

1. Máquina de Turing **Determinística**: (Se 'A', então 'B')
2. Máquina de Turing **Não-Determinística**: (Se 'A", então 'B' **ou** 'C' **ou** 'D' **ou** ...)

Teoria da Computabilidade

A computabilidade é a Teoria da Complexidade Computacional que estudam os **limites** da computação:

1. Quais problemas jamais poderão ser resolvidos por um computador, independente de sua velocidade ou memória?
2. Quais problemas podem ser resolvidos por um computador, mas requerem um período tão extenso de tempo para completar a ponto de tornar a solução impraticável?
3. Em que situações podem ser mais difícil resolver um problema do que verificar cada uma das soluções?

Teoria da Computabilidade

Das três perguntas anteriores, a última é referente às **classes de paradigmas**, são elas:

- Classe **P**: De tempo polinomial determinístico. Os algoritmos pertencentes à esta classe são chamados de **algoritmos eficientes**.
- Classe **NP**: De tempo polinomial não-determinístico.

O conjunto de problemas que não podem ter solução em tempo polinomial mas candidatos a solução podem ser checados em tempo polinomial são problemas pertencentes à classe NP.

Somente uma máquina de Turing não-determinística podem resolver esses problemas. Eles são resolvidos em tempo polinomial por uma máquina de Turing não-determinística que acerta em todos os passos.

Teoria da Computabilidade

Na mátemática, a questão a respeito de $\mathrm{P} = \mathrm{NP}$ ou $\mathrm{P} \neq \mathrm{NP}$ é um problema em aberto.

A grande importância dessa classe (NP) de problemas se baseia no fato de que ela contém muitos problemas de busca e otimização para os quais gostaríamos de saber se há uma solução.

Exemplos: Problema do Caixeiro Viajante, problema da mochila, problemas de otimização descritos na ementa do curso como é o caso do método BFGS.

E se P = NP?

1. Diga adeus a criptografia;
2. Soluções matemáticas não complicadas;
3. Previsão do tempo, terremotos e tsunamis;

Problemas NP (precisa-se de linguagens eficientes)

Problemas NP-Completos fazem parte de nossas vidas...

Na estatística sempre nos deparamos com problemas NP-Completos, isto é, sempre lidamos com problemas que não possuem soluções em tempo polinomial.

Exemplo: Constantemente precisamos estimar parâmetros de um modelo probabilístico por meio do método de máxima verossimilhança, isto é, maximizamos a função de log-verossimilhança de um modelo probabilístico. A otimização global é um problema NP-Completo.

Observação: Atualmente está cada vez mais complicados realizar tais otimizações uma vez que os modelos cada vez mais estão adicionando parâmetros extras o que torna a função extremamente complicada em alguns casos.

Breve História da Linguagem C

Embora possua um nome estranho quando comparada com outras linguagens de programação da terceira geração, como FORTRAN, PASCAL, ou COBOL, a linguagem C é uma das linguagens mais importantes até hoje criada.

Curiosidade: O nome da linguagem (e a própria linguagem) resulta da evolução de uma outra linguagem de programação, desenvolvida pelo programador **Ken Thompson** nos Laboratórios Bell, chamada de B.

Por que aprender à programar?

O professor continua respondendo...

A maioria desses software não possuem as técnicas estatísticas mais recentes e muitas vezes não são capazes de se adequar aos problemas específicos que nos deparamos ao tentar resolver um problema.

Muitas vezes precisamos modificar uma função programada por um outro programador para que ela venha a funcionar como solução ao nosso problema.

Observação: Diversas outras vezes precisamos programar para realizar simulações. É muito comum na estatística estudar propriedades de algumas estatística ou modelo estatístico e querer simular o seu comportamento em diversos cenários diferentes.

Breve História da Linguagem C

Figura: Ken Thompson (sentado) jogando xadrez com um colega.

B trata-se de uma simplificação da linguagem BCPL (*Basic Combined Programming Language*). Assim como BCPL, B só possuía um tipo de dados.

A linguagem B também foi recebeu contribuições do Dennis Ritchie (criador de C).

Breve História da Linguagem C

Figura: Dennis Ritchie (criador da linguagem C).

A linguagem C foi criada em 1972 nos *Bell Telephone Laboratories* por Dennis Ritchie com a finalidade de permitir a escrita do sistema operacional Unix.

Desejava-se uma linguagem de alto nível de modo a evitar o uso do `Assambly`.

Breve História da Linguagem C

Figura: Dennis Ritchie (criador da linguagem C).

Devido às capacidades e através da divulgação do sistema Unix pelas universidades dos Estados Unidos, a linguagem C deixou cedo as portas dos laboratórios Bell.

C disseminou-se e tornou-se conhecida por todos os tipos de programadores, independentemente dos projetos em que estivessem envolvidos.

Breve História da Linguagem C

Figura: Ótimo livro sobre a linguagem C.

Essa dispersão de diferentes projetos utilizando a linguagem C levou a que diferentes organizações desnvolvessem e utilizassem diferentes versões da linguagem C criando assim alguns problemas de compatibilidade, entre diversos outros.

O material ao lado é um livro sobre a linguagem C escrito pelo seu criador.

Breve História da Linguagem C

Figura: Ótimo livro sobre a linguagem C.

Devido ao fenômeno que foi a linguagem C e aos problemas de compatibilidade que existiam na época, o **American National Standards Institute** (ANSI) formou em 1983 um comitê para a definição de um padrão para a linguagem C.

Breve História da Linguagem C

Figura: Ótimo livro sobre a linguagem C.

O padrão tem como objetivo o funcionamento semelhante de todos os compiladores da linguagem, com especificações muito precisas sobre aquilo que a linguagem deve ou não fazer, seus limites, definições, dentre outras coisas.

Mil e Uma Razões para Programar em C

Devido à enorme quantidade de linguagens de programação disponíveis no mercado, seria necessário que uma delas se destacasse muito em relação às outras para conseguir interessar tantos programadores.

A maior parte das linguagens de programação tem um objetivo específico a atingir:

- PASCAL - Ensino de Técnicas de Programação.
- FORTRAN - Cálculo Científico.
- LISP e PROLOG - Vocacionadas para as áreas de Inteligência Artificial.

Pergunta do aluno

Certo, entendi, temos linguagens que se destinam a um objetivo específico como as que foram listadas a cima. Mas quanto à C, a que área de desenvolvimento se destina?

Pergunta do aluno

Certo, entendi, temos linguagens que se destinam a um objetivo específico como as que foram listadas a cima. Mas quanto à C, a que área de desenvolvimento se destina?

Resposta do professor legal

NENHUMA ☺ .

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

1. C é uma linguagem extremamente eficiente.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

1. C é uma linguagem extremamente eficiente.
2. Diversas outras linguagem foram programas utilizando C.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

1. C é uma linguagem extremamente eficiente.
2. Diversas outras linguagem foram programas utilizando C.
3. Na estatística muitas vezes precisamos fazer simulações que são computacionalmente intensivas.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

1. C é uma linguagem extremamente eficiente.
2. Diversas outras linguagem foram programas utilizando C.
3. Na estatística muitas vezes precisamos fazer simulações que são computacionalmente intensivas.
4. C é uma ótima linguagem para se começar a programar.

Por que C?

Qual o porquê de se aprender a linguagem C

C é uma linguagem bastante interessante devido ser uma linguagem de programação de propósito geral, isto é, C pode ser utilizada para os mais variados fins.

Alguns outros motivos para se aprender C são:

1. C é uma linguagem extremamente eficiente.
2. Diversas outras linguagem foram programas utilizando C.
3. Na estatística muitas vezes precisamos fazer simulações que são computacionalmente intensivas.
4. C é uma ótima linguagem para se começar a programar.
5. Códigos em C podem ser importados para R, isto é, R "conversa" com C.

Por que C?

Observação: Pelo fato de C ser uma linguagem de propósito geral (*general purpose*), esta linguagem pode ser utilizada nos mais variados fins, como sistemas operacionais, interfaces gráficas, etc.

Importante

Há uma falácia de que C é uma linguagem extremamente difícil. Na verdade ocorre que muitas pessoas começam a estudar programação por meio de C, momento este em que é somado dificuldades em aprender à programar (lógica de programação) com as dificuldades de se aprender uma sintaxe de uma linguagem de programação.

Observação: Porém, é verdade que muitas coisas temos que fazer nós mesmo em C. Por isso que ela é a linguagem adotada na maioria dos cursos de introdução à programação espalhados pelo mundo.

Por que C?

Outras razões para se utilizar C

C é utilizado quando a velocidade, espaço e portabilidade são importantes. A maioria dos sistemas operacionais das outras linguagens e de grande parte dos softwares e games são escritas em C.

Observação: Há basicamente três padrões de C que podem ser encontrados por aí. São eles:

1. **ANSI** C que é do fim dos anos de 1980 e é utilizado para códigos mais antigos;
2. Muitas coisas foram consertadas no padrão C99 de 1999;
3. Algumas novidades foram acrescentadas no atual padrão C11 lançado em 2011.

Observação: Não existem grandes diferenças entre as versões de C. Iremos destacá-las ao longo do caminho.

Arquitetura de von Neumann

Figura: John von Neumann.

A arquitetura de von Neumann é uma arquitetura de um computador digital que possibilita uma máquina digital armazenar os seus programas no mesmo espaço de memória que os dados, podendo assim manipular tais programas.

"Não há sentido em em ser preciso quando não se sabe sobre o que está a falar- von Neumann.

Arquitetura de von Neumann

A máquina proposta por von Neumann possui as seguintes componentes:

1. Uma **memória**;
2. Uma **unidade aritmética e lógica**;
3. Uma **unidade central de processamento** (CPU), composto por diversos registradores;
4. Uma **unidade de controle**, cuja função é a mesma da tabela de controle de uma Máquina de Turing universal (estabelece as mudanças de estado por meio das entradas).

Compilador

O que é um compilador?

Compilador

O que é um compilador?

Resposta: Um compilador é um programa de computador ou mesmo um grupo de programas que é responsável por traduzir um código fonte escrito em uma linguagem compilada à um programa equivalente do ponto de vista semântico.

Compilador

O que é um compilador?

Resposta: Um compilador é um programa de computador ou mesmo um grupo de programas que é responsável por traduzir um código fonte escrito em uma linguagem compilada à um programa equivalente do ponto de vista semântico.

O compilador traduz o código fonte de uma linguagem de programação de médio/alto nível para uma linguagem de programação de baixo nível (a exemplo da linguagem `Assembly` ou código de máquina).

Bytecode?

Compilador

Bytecode?

Alguns compiladores traduzem o código para um formato intermediário, denominado de **bytecode** que é um código de baixo nível. Sendo assim, o bytecode não é imediatamente um arquivo executável.

Compilador

Bytecode?

Alguns compiladores traduzem o código para um formato intermediário, denominado de **bytecode** que é um código de baixo nível. Sendo assim, o bytecode não é imediatamente um arquivo executável.

Observação: Chamamos de linguagem de baixo nível as linguagem que trabalhando próximo ao hardware. Baixo nível, médio nível ou alto nível em nada em a ver com a qualidade da linguagem de programação.

Compilador

Bytecode?

Alguns compiladores traduzem o código para um formato intermediário, denominado de **bytecode** que é um código de baixo nível. Sendo assim, o bytecode não é imediatamente um arquivo executável.

Observação: Chamamos de linguagem de baixo nível as linguagem que trabalhando próximo ao hardware. Baixo nível, médio nível ou alto nível em nada em a ver com a qualidade da linguagem de programação.

Importante: Jamais confunda bytecode com código de máquina. Bytecode é um formato intermediário que irá ser interpretado em uma máquina virtual que fará a execução.

Compilador

A vantagem do bytecode é que o código torna-se mais **portável**, isto é, podemos com o resultado da compilação executar o código proveniente de um processo de compilação em diversas arquiteturas distintas. Dessa forma, o bytecode irá produzir o mesmo resultado esperado em qualquer arquitetura que possua uma máquina virtual que execute o código intermediário.

Exemplos de linguagem que converte o código fonte para bytecode: `Java` que corre o código sobre a máquina virtual `Java`, `.NET` que corre o código sobre a *Common Language Runtime*.

Compilador

Código Objeto e Código de Máquina

O **código de máquina** é um código binário (0 e 1) que poderá ser executado diretamente pela CPU.

Se abrirmos um arquivo de código de máquina em um editor de texto, veríamos um emaranhado de caracteres sem sentido. É possível ter acesso ao código de máquina em formato hexadecimal por meio de softwares adequados.

O **código objeto** é a saída de um processo de compilação e trata-se de uma parte do código de máquina que ainda não foi vinculado em um programa completo por meio de um **linker**.

Abrindo um Código de Maquina

Ao tentarmos abrir um código de máquina em um editor de texto comum visualizamos algo sem sentido como a sequência de caracteres abaixo:

Abrindo um Código de Maquina

Ao tentarmos abrir um código de máquina em um editor de texto comum visualizamos algo sem sentido como a sequência de caracteres abaixo:

```
MZÀ$Pÿv èŠÿ]Ë3ÀP¸F ë ƒF ¸ < uè2Àëä ÀtBª
Àu C†à2Àùã ¬I,"t "<\u €<"u ¬I öÃ é îY Ê. ‹å‰.
Œ ˜ ã ‰v¸ vüÿv ÿv  èÅ ƒÄ ÿvþÿvü  èüêYY‹V ‹F
ë Rÿvþÿvü  èWíƒÄ ‹å]ËU‹ìƒìHVW‹~ ‹F
```

Porém, é possível ter acesso ao código de máquina utilizando editores próprios que apresentam o código em hexadecimal, como o exemplo que segue no *frame* seguinte.

Compilador

```
C:\Utility>debug v.exe
-d 0 100
0E3D:0000  CD 20 FF 9F 00 9A F0 FE ........O.....
0E3D:0010  F0 07 17 03 F0 07 DF 07 ................
0E3D:0020  FF FF FF FF FF FF FF FF ...............L.
0E3D:0030  D0 0C 14 00 18 00 3D 0E ......=.........
0E3D:0040  05 00 00 00 00 00 00 00 ................
0E3D:0050  CD 21 CB 00 00 00 00 00 .!...........
0E3D:0060  20 20 20 20 20 20 20 20 .....
0E3D:0070  20 20 20 20 20 20 20 20 ........
0E3D:0080  00 0D 76 2E 65 78 65 0D ..v.exe.DOWS\sys
0E3D:0090  74 65 6D 33 32 5C 64 6F tem32\dosx..da r
0E3D:00A0  65 64 65 20 28 63 61 72 ede (carregar an
0E3D:00B0  74 65 73 20 64 6F 20 64 tes do dosx.exe)
```

Compilador

Figura: Diagrama (funcionamento de uma compilador).

Compilador

Alguns autores citam linguagens compiladas em que a tradução do código gera código em `C`.

Maiores detalhes na referência abaixo

Cooper, Torczon. Engineering a Compiler (em inglês). San Francisco: Morgan Kaufmann, 2003. p. 2. ISBN 1-55860-698-X.

Compilador

Importante

É importante não confundir **compilador** com **tradutor** ou **filtro** que também pode ser chamado de **conversor de linguagem**.

O conversor de linguagem é responsável por converter o código de uma linguagem de alto nível para o código de um outra linguagem de médio/alto nível.

Observação: Um programa que traduz uma linguagem de programação de baixo nível para uma linguagem de programação de alto nível é denominado de **descompilador**.

Compilador

Figura: Grace Hopper.

O primeiro compilador foi escrito por Grace Hopper no ano de 1952 para a linguagem de programação `A-0`.

Grace Hopper foi analista de sistemas da Marinha dos Estados Unidos. Ela também criou o primeiro compilador para a linguagem `COBOL`.

Curiosidade: É atribuído à Grace Hopper o termo **bug** utilizado para designar uma falha no código fonte.

Compilador

Figura: Grace Hopper.

Grace Hopper é graduada em matemática e física em 1928 e em 1930 concluiu seu mestrado na Yale University. Em 1934, na mesma Universidade, ela obteve o seu PhD em matemática.

Compilador

Muitos compiladores incluem um **pré-processador** que é um programa separado mas invocado pelo compilador antes do início do processo de tradução.

Normalmente é pre-processador responsável por mudanças no código fonte destinadas de acordo com decisões tomadas em tempo de compilação.

Em programas em `C` há diversas diretivas para inclusão de novos códigos disponíveis em bibliotecas ou código a parte escrito pelo programador que é informado sua existência por meio de diretivas para o pre-processador.

Compilador

Exemplo: É o pré-processador que substitui os comentários do código fonte por espaços em branco. Ou seja, o compilador não "enxerga" nenhum comentário.

Exemplos de linguagens compiladas: `C`, `C++`, `Fortran`, `Objecte-C`, `Ocalm`, `BASIC`, `COBOL`, `Ada`, `D`, entre outras.

Interpretador

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Interpretador

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Seu funcionamento pode ser, em geral, de duas formas:

- O interpretador lê linha-por-linha e converte o código fonte em código objeto (ou bytecode) a medida que vai executando o programa.

Interpretador

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Seu funcionamento pode ser, em geral, de duas formas:

- O interpretador lê linha-por-linha e converte o código fonte em código objeto (ou bytecode) a medida que vai executando o programa.
- O interpretador converte o código fonte por inteiro e depois o executa.

Interpretador

O que é um interpretador?

Resposta: Interpretadores são programas de computador que tem a finalidade de ler o código-fonte de uma linguagem de programação interpretada e o converte em um código executável.

Seu funcionamento pode ser, em geral, de duas formas:

- O interpretador lê linha-por-linha e converte o código fonte em código objeto (ou bytecode) a medida que vai executando o programa.
- O interpretador converte o código fonte por inteiro e depois o executa.

Exemplos de linguagens interpretadas: `R`, `Perl`, `Python`, `Haskell`, `Lua`, `Ruby`, `Lisp`, `JavaScript`, entre outras.

Interpretador

Agora que conhecemos a diferença entre **linguagem de programação compilada** para **linguagem de programação interpretada** e sabemos que `R` é uma linguagem interpretada, jamais fale:

"MEU CÓDIGO R ESTÁ COMPILANDO ..."

Dizer o que está destacado acima é uma prova contundente do pouco conhecimento de computação e/ou da linguagem `R`.

Por que aprender à programar?

Uma linguagem de programação bastante utilizada na estatística é a linguagem R.

Linguagem R

R é uma linguagem de programação para computação estatística e gráficos. R é uma parte oficial do projeto GNU da Free Software Foundation's.

Curiosidade: A linguagem R foi criada originalmente por Ross Ihaka e Robert Gentleman no **Departamento de Estatística** da Universidade de Auckland, Nova Zelândia em agosto de 1993.

Nota: É muito importante que um estatístico saiba programar na linguagem R. Alguns empregos exigem isso. Porém, se não exigirem, o R te ajudará bastante.

Por que aprender à programar?

Figura: Criadores da linguagem R [Robert Gentleman (foto à esquerda) e Ross Ihaka (foto à direita)].

Por que aprender à programar?

Figura: Logo da linguagem R.

Um dos grandes motivos da grande popularidade da linguagem R se deve a grande quantidade de pacotes disponíveis para os usuários da linguagem.

Atualmente há mais de 12750 pacotes para R com o foco nas mais variadas áreas: estatística, matemática, biologia, economia, entre outras.

Por que aprender à programar?

Figura: Logo da linguagem R.
Obtenha a linguagem R em
https://www.r-project.org/.

Observação: Para programar em R **não é suficiente entender alguns pacotes específicos**. É preciso entender a sintaxe base da linguagem que nos permite inclusive criar outros pacotes e melhorar os existentes.

Por que aprender à programar?

Figura: Logo da linguagem R.
Obtenha a linguagem R em
https://www.r-project.org/.

Observação: As novas metodologias estatísticas chegam mais rapidamente em R do que em outros softwares estatísticos pelo fato do R ser uma linguagem livre (código aberto e gratuita).

Por que R?

Algumas respostas:

1. R é uma linguagem de código-aberto;
2. Muitos estatísticos espalhados pelo mundo utilizam R;
3. Grandes empresas utilizam a linguagem R;
4. Há muitos metologias estatísticas empacotadas;
5. R se comunica com a linguagem C/C++ e outras;
6. É possível escrever textos tipográficos de qualidade usando LaTeX e inserir códigos R no texto para produção de gráficos, tabelas e resultados de metodologias estatística.

Observação: Maiores detalhes sobre download e instalação da linguagem R nos diferentes sistemas operacionais e arquiteturas poderão ser obtidos em https://cloud.r-project.org/.

Programação em R

É importante entender que R antes de ser um ambiente para computação estatística e gráficos trata-se de uma linguagem de programação.

Programação em R

É importante entender que R antes de ser um ambiente para computação estatística e gráficos trata-se de uma linguagem de programação.

Muito Importante

Para que seja possível dominar a linguagem R é fundamental esquecermos o paradigma de estudo da linguagem por meio de livros e tutoriais que tentam ensinar R por meio de soluções de problemas estatísticos. Esses materiais são úteis quando se tem um domínio razoável da linguagem. Tentar aprender a linguagem R como um software estatístico e não como uma linguagem de programação irá impor diversas deficiências de programação ao usuário/programador.

RStudio

Para termos uma boa experiência de programação em R, é importante instalar um editor ou IDE (*Integrated Development Environment*) de programação.

Uma boa IDE para programação em R é o **RStudio**. **RStudio** vem evoluindo bastante com o passar dos anos e hoje é a melhor IDE para se trabalhar com R.

Para maiores detalhes e download do **RStudio**, acessar: https://www.rstudio.com/.

Figura: Algumas empresas que utilizam o **RStudio**.

RStudio

Figura: **RStudio** (IDE de programação em R).

Figura: GNU Affero General Public License - GNU AGPL, uma das licenças do **RStudio**.

RStudio

Uma das licenças do **RStudio** é a licença GNU AGPL. Com o **RStudio** sob os termos dessa licença poderemos ter acesso ao código fonte do **RStudio** e este poderá ser executado atrás de um servidor web, do mesmo modo como funciona uma aplicação web.

Observação: Há também uma versão paga do **RStudio** custando $995 dólares por ano. Essa versão trás novas funcionalidades ao **RStudio** como, por exemplo, executar várias análises em paralelo, a possibilidade de executar várias versões do `R` lado a lado, monitora sessões ativas, CPU e utilização de memória, bem como outras funcionalidades administrativas além do suporte técnico.

RStudio

A empresa **RStudio** com sede em Boston, Massachusetts, USA também fornece outros serviços relacionados com a linguagem R e com a ferramenta **RStudio** como é o caso do **Shiny** que é um pacote open-source de R que fornece uma estrutura para construção de aplicativos web usando R.

Shiny ajuda o programador transformar as análises em aplicativos web interativos e não requer conhecimento de HTML, CSS ou JavaScript.

Observe no link alguns exemplos de análises interativas: http://shiny.rstudio.com/gallery/.

Figura: Exemplo de gráfico interativo usando **Shiny**.

RStudio

A equipe do **RStudio** também vem produzindo alguns pacotes importantes que são bastante úteis para programadores de R além do **Shiny**, como é o caso do **rmarkdown**, **knitr**, **ggplot2**, entre outros.

Figura: Alguns pacotes produzidos pela equipe do **RStudio**.

RStudio

No primeiro quadrante da IDE é onde escrevemos nosso código com extensão .R. Para que o interpretador de `R` possa agir sobre um trecho do código, este deverá ser selecionado e em seguida deverá ser teclado **Ctrl** + **Enter**.

No segundo quadrante é onde observamos os objetos criados pelo programador que basicamente são matrizes, listas, dataframes, variáveis, entre outros objetos suportados pela linguagem.

O terceiro quadrante é o prompt de comando de `R`. Toda a saída encontra-se nesse quadrante.

O quarto quadrante é onde podemos consultar o help das funções e onde observamos as saídas gráficas.

RCode

Recentemente surgiu uma nova IDE para programação em R com o nome de **RCode**.

RCode

Recentemente surgiu uma nova IDE para programação em R com o nome de **RCode**.

Algumas características do **RCode** são:

- Tem como objetivo ser otimizada e o mais leve possível, permitindo trabalhar bem como máquinas mais modestas.

RCode

Recentemente surgiu uma nova IDE para programação em R com o nome de **RCode**.

Algumas características do **RCode** são:

- Tem como objetivo ser otimizada e o mais leve possível, permitindo trabalhar bem como máquinas mais modestas.
- Não há necessidade de baixar manualmente o **RCode** cada vez que é atualizado.

RCode

Recentemente surgiu uma nova IDE para programação em R com o nome de **RCode**.

Algumas características do **RCode** são:

- Tem como objetivo ser otimizada e o mais leve possível, permitindo trabalhar bem como máquinas mais modestas.
- Não há necessidade de baixar manualmente o **RCode** cada vez que é atualizado.
- Fornece automaticamente o tempo de execução de cada comando.

RCode

Recentemente surgiu uma nova IDE para programação em R com o nome de **RCode**.

Algumas características do **RCode** são:

- Tem como objetivo ser otimizada e o mais leve possível, permitindo trabalhar bem como máquinas mais modestas.
- Não há necessidade de baixar manualmente o **RCode** cada vez que é atualizado.
- Fornece automaticamente o tempo de execução de cada comando.
- Preenchimento automático de comandos.

O **RCode** possui versões para Linux, macOS e Windows. Maiores detalhes em https://www.pgm-solutions.com/rcode.

RCode

Figura: IDE **RCode** para programação em linguagem R.

Estrutura de dados em R

As estruturas das base de dados em R podem ser organizadas em diversas dimensões (1d, 2d, nd) que podem ser homogêneas (mesmo tipo de dados) ou heterogêneas (diferentes tipos de dados).

Tabela: Estrutura de dados em R.

	Homogênea	**Heterogênea**
1d	Atomic vector	List
2d	Matrix	Data Frame
nd	Array	

Observação: Quase todos os outros objetos da linguagem R são construídos sobre essas fundações.

Estrutura de dados em R

Muito Importante

R não possui strings nem escalares propriamente ditos. Na verdade estes são tratados como um vetor de cardinalidade 1.

Estrutura de dados em R

Muito Importante

`R` não possui strings nem escalares propriamente ditos. Na verdade estes são tratados como um vetor de cardinalidade 1.

Nota: Dado um objeto qualquer em `R`, a melhor forma de intender sua estrutura de dados é utilizando a função `str()`. A função `str()` apresenta um resumo de forma compacta e legível do objeto em questão.

Estrutura de dados em R

Muito Importante

`R` não possui strings nem escalares propriamente ditos. Na verdade estes são tratados como um vetor de cardinalidade 1.

Nota: Dado um objeto qualquer em `R`, a melhor forma de intender sua estrutura de dados é utilizando a função `str()`. A função `str()` apresenta um resumo de forma compacta e legível do objeto em questão.

Porém, espere um pouco antes de utilizar esta função. Nós ainda não sabemos como criar objetos em `R`.

Estrutura de dados em R

A estrutura básica de dados da linguagem `R` é o vetor. Por isso que dizemos que `R` é uma **linguagem vetorial**.

Existem dois tipos de vetores em `R`:

Estrutura de dados em R

A estrutura básica de dados da linguagem `R` é o vetor. Por isso que dizemos que `R` é uma **linguagem vetorial**.

Existem dois tipos de vetores em `R`:

1. Vetores atômicos;

Estrutura de dados em R

A estrutura básica de dados da linguagem `R` é o vetor. Por isso que dizemos que `R` é uma **linguagem vetorial**.

Existem dois tipos de vetores em `R`:

1. Vetores atômicos;
2. Listas.

Estrutura de dados em R

A estrutura básica de dados da linguagem `R` é o vetor. Por isso que dizemos que `R` é uma **linguagem vetorial**.

Existem dois tipos de vetores em `R`:

1. Vetores atômicos;
2. Listas.

Lembre-se: Lista em `R` é um vetor não-atômico. Enquanto um vetor atômico possui elementos do mesmo tipo, uma lista poderá ser heterogênea.

Estrutura de dados em R

Existem três funções úteis para obter as propriedades de um vetor. Estas são:

Estrutura de dados em R

Existem três funções úteis para obter as propriedades de um vetor. Estas são:

1. `typeof()` (retorna o tipo de dados do vetor);

Estrutura de dados em R

Existem três funções úteis para obter as propriedades de um vetor. Estas são:

1. `typeof()` (retorna o tipo de dados do vetor);
2. `length()` (retorna a cardinalidade do vetor);

Estrutura de dados em R

Existem três funções úteis para obter as propriedades de um vetor. Estas são:

1. `typeof()` (retorna o tipo de dados do vetor);
2. `length()` (retorna a cardinalidade do vetor);
3. `attributes()` (retorna metadados adicionais que possam existir).

Faça no prompt do `R`:

```
# Inicializando um vetor em R.
x <- c(1,7,9,7) # x é um vetor atômico.
length(x)
typeof(x)
```

Comentando um código R

Observação: Comentários na linguagem R são realizados utilizando o símbolo #. Um código bem implementado sempre possui uma quantidade razoável de comentários. Há quem diga que um código deverá ter mais comentários do que linhas implementadas. De fato, isto não é ruim!

```
1: # Sempre é bom comentar
2: # um código
3:
4: código
5:
6: # Mais comentários
```

Checando um vetor

Podemos utilizar duas formas para checar se um objeto em `R` é um vetor (atômico ou não-atômico).

- **Forma 1**: `is.vector(x)`, em que `x` é um vetor ou lista;
- **Forma 2**: `is.atomic(x) || is.list(x)`.

Adianto que a melhor forma de fazer isso é a segunda, pois esta forma funciona nas situações em que o vetor possui algum atributo.

Checando um vetor

Podemos utilizar duas formas para checar se um objeto em `R` é um vetor (atômico ou não-atômico).

- **Forma 1**: `is.vector(x)`, em que `x` é um vetor ou lista;
- **Forma 2**: `is.atomic(x) || is.list(x)`.

Adianto que a melhor forma de fazer isso é a segunda, pois esta forma funciona nas situações em que o vetor possui algum atributo.

Importante: Os **tipos atômicos** (*atomic types*) da linguagem `R` são basicamente: `logical`, `integer`, `numeric`, `complex`, `character`, `NULL` e `raw`. Este último tipo não é importante para nós no momento.

Checando um vetor

Exemplo: Note a diferença da saída de `is.atomic(y)` e da obtida por `is.atomic(y) || is.list(y)`.

```
x <- c(a=1,b=2)
y <- c(a=1,b=2)
attr(y,'ano') <- '2017'
# Dois comandos em uma linha.
is.vector(x); is.vector(y)
is.atomic(y) || is.list(y)
```

Se algo ainda parece complicado, na medida que formos avançando na programação em R, as dúvidas serão sanadas.

Vetores atômicos

Vetores atômicos são criados com o comando `c()`. Alguns exemplos:

Vetores atômicos

Vetores atômicos são criados com o comando `c()`. Alguns exemplos:

- **Ex**: `dbl_var <- c(1,2.5,4.5)`. Vetor de variáveis de precisão dupla;

Vetores atômicos

Vetores atômicos são criados com o comando `c()`. Alguns exemplos:

- **Ex**: `dbl_var <- c(1,2.5,4.5)`. Vetor de variáveis de precisão dupla;
- **Ex**: `int_var <- c(1L, 6L, 10L)`. Aqui estamos criando um vetor de inteiros quando colocamos o sufixo `L`;

Vetores atômicos

Vetores atômicos são criados com o comando `c()`. Alguns exemplos:

- **Ex**: `dbl_var <- c(1,2.5,4.5)`. Vetor de variáveis de precisão dupla;
- **Ex**: `int_var <- c(1L, 6L, 10L)`. Aqui estamos criando um vetor de inteiros quando colocamos o sufixo `L`;
- **Ex**: `log_var <- c(TRUE, FALSE, T, F)`. Diferentemente da linguagem `C`, os valores lógicos em `R` são representados por `TRUE` ou `T` e `FALSE` ou `F`;

Vetores atômicos

Vetores atômicos são criados com o comando `c()`. Alguns exemplos:

- **Ex**: `dbl_var <- c(1,2.5,4.5)`. Vetor de variáveis de precisão dupla;
- **Ex**: `int_var <- c(1L, 6L, 10L)`. Aqui estamos criando um vetor de inteiros quando colocamos o sufixo `L`;
- **Ex**: `log_var <- c(TRUE, FALSE, T, F)`. Diferentemente da linguagem `C`, os valores lógicos em `R` são representados por `TRUE` ou `T` e `FALSE` ou `F`;
- **Ex**: `chr_var <- c("string", "outra_string2")`. Vetor de sequências de caracteres.

Vetores atômicos

Importante

Note que atribuições da da forma `objeto <- valor` faz com que **objeto** seja um vetor de uma única posição.

Vetores atômicos

Importante

Note que atribuições da da forma `objeto <- valor` faz com que **objeto** seja um vetor de uma única posição.

```
1: a <- 7L
2: a[1]
#> 7
# Note que o objeto "a" é um vetor. Pergunte se ele
# é um vetor:
3: is.vector(a) # Já sabia que "a" não iria "list".
#> TRUE
# Poderia ter feito a comparação acima da seguinte forma:
4: is.atomic(a) || is.list(a)
#> TRUE
```

Concatenando um vetor

Importante

Vetores atômicos são sempre "planos" mesmo que tenhamos a ideia de aninhar vários vetores atômicos.

Exemplo: Veja que não existe nenhuma diferença entre executar o código `c(1, c(2, c(3, 4)))` e executar `c(1, 2, 3, 4)`.

Nota: Muito embora para o fim do exemplo acima concatenar vetores não haveria necessidade uma vez que poderíamos criar o vetor de uma única vez utilizando `c(1, 2, 3, 4)`, saber concatenar vetores é bastante necessário.

Acessando elementos

Seja `x` um vetor atômico qualquer. Podemos acessar um elemento, ou conjunto de elementos do vetor `x` utilizando a seguinte sintaxe.

Sintaxe: `x[posições]`, em que `posições` é o vetor de posições ao qual queremos acessar.

Exemplo: Corra o código abaixo:

```
x <- c(2, 7, 8, 9, 7)
y <- x[c(2,4)]
y
#> [1] 7 9
```

Nota: Maiores detalhes serão apresentados mais a frente ao estudarmos operadores de subconjuntos.

Acessando elementos

Nota: Lembre-se que R é uma linguagem vetorial. Sendo assim, em diversas situações não há a necessidade de utilizar loops para acessar cada elemento de um vetor quando precisamos aplicar uma função ou realizar uma operação em seus elementos.

```
a <- c(7.1, 2.3, 3L, TRUE)
b <- a + 1 # Somando 1 ao vetor "a".
c <- a^0.5 # Exponencializando o vetor "a".
a
#> [1] 7.1 2.3 3.0 1.0
b
#> [1] 8.1 3.3 4.0 2.0
c
#> [1] 2.664583 1.516575 1.732051 1.000000
```

Valores faltantes

Missing Values/Not Available

Valores faltantes (*Not Available*) são especificados por `NA`. Em `R`, `NA` é um vetor de comprimento 1. Além disso, `NA` sempre será coagido para o tipo correto se utilizado dentro de `c()`.

Também poderemos especificar o tipo de valor faltante utilizando:

Valores faltantes

Missing Values/Not Available

Valores faltantes (*Not Available*) são especificados por `NA`. Em `R`, `NA` é um vetor de comprimento `1`. Além disso, `NA` sempre será coagido para o tipo correto se utilizado dentro de `c()`.

Também poderemos especificar o tipo de valor faltante utilizando:

- `NA_real_`: um vetor de precisão dupla;

Valores faltantes

Missing Values/Not Available

Valores faltantes (*Not Available*) são especificados por `NA`. Em `R`, `NA` é um vetor de comprimento `1`. Além disso, `NA` sempre será coagido para o tipo correto se utilizado dentro de `c()`.

Também poderemos especificar o tipo de valor faltante utilizando:

- `NA_real_`: um vetor de precisão dupla;
- `NA_integer_`: um vetor de inteiros;

Valores faltantes

Missing Values/Not Available

Valores faltantes (*Not Available*) são especificados por `NA`. Em `R`, `NA` é um vetor de comprimento 1. Além disso, `NA` sempre será coagido para o tipo correto se utilizado dentro de `c()`.

Também poderemos especificar o tipo de valor faltante utilizando:

- `NA_real_`: um vetor de precisão dupla;
- `NA_integer_`: um vetor de inteiros;
- `NA_character_`: um vetor de caracteres.

Valores faltantes

Missing Values/Not Available

Valores faltantes (*Not Available*) são especificados por `NA`. Em `R`, `NA` é um vetor de comprimento `1`. Além disso, `NA` sempre será coagido para o tipo correto se utilizado dentro de `c()`.

Também poderemos especificar o tipo de valor faltante utilizando:

- `NA_real_`: um vetor de precisão dupla;
- `NA_integer_`: um vetor de inteiros;
- `NA_character_`: um vetor de caracteres.

Observação: Lembre-se que na estatística é muito comum lacunas em conjuntos de dados. Na análise de sobrevivência, por exemplo, é comum a presença de censura nos dados.

Tipos e Testes

Em `R` é possível verificar o tipo de um vetor como apresentado anteriormente com a função `typeof()`. Além disso, poderemos chegar o tipo específico dos elementos em um vetor utilizando as funções que seguem:

Tipos e Testes

Em `R` é possível verificar o tipo de um vetor como apresentado anteriormente com a função `typeof()`. Além disso, poderemos chegar o tipo específico dos elementos em um vetor utilizando as funções que seguem:

- **Ex**: `is.character()`. Checa se é um vetor de caracteres;

Tipos e Testes

Em `R` é possível verificar o tipo de um vetor como apresentado anteriormente com a função `typeof()`. Além disso, poderemos chegar o tipo específico dos elementos em um vetor utilizando as funções que seguem:

- **Ex**: `is.character()`. Checa se é um vetor de caracteres;
- **Ex**: `is.double()`. Checa se é um vetor de números reais;

Tipos e Testes

Em `R` é possível verificar o tipo de um vetor como apresentado anteriormente com a função `typeof()`. Além disso, poderemos chegar o tipo específico dos elementos em um vetor utilizando as funções que seguem:

- **Ex**: `is.character()`. Checa se é um vetor de caracteres;
- **Ex**: `is.double()`. Checa se é um vetor de números reais;
- **Ex**: `is.integer()`. Checa se é um vetor de inteiros;

Tipos e Testes

Em R é possível verificar o tipo de um vetor como apresentado anteriormente com a função `typeof()`. Além disso, poderemos chegar o tipo específico dos elementos em um vetor utilizando as funções que seguem:

- **Ex**: `is.character()`. Checa se é um vetor de caracteres;
- **Ex**: `is.double()`. Checa se é um vetor de números reais;
- **Ex**: `is.integer()`. Checa se é um vetor de inteiros;
- **Ex**: `is.logical()`. Checa se é um vetor de valores lógicos;

Tipos e Testes

Em R é possível verificar o tipo de um vetor como apresentado anteriormente com a função `typeof()`. Além disso, poderemos chegar o tipo específico dos elementos em um vetor utilizando as funções que seguem:

- **Ex**: `is.character()`. Checa se é um vetor de caracteres;
- **Ex**: `is.double()`. Checa se é um vetor de números reais;
- **Ex**: `is.integer()`. Checa se é um vetor de inteiros;
- **Ex**: `is.logical()`. Checa se é um vetor de valores lógicos;
- **Ex**: `is.atomic()`. Checa se é um vetor atômico.

Tipos e Testes

Exemplo: Execute o código R que segue logo abaixo:

```
1: int_var <- c(1L, 6L, 10L)
2: typeof(int_var)
#> [1] "integer"
3: is.integer(int_var)
#> [1] "TRUE"
4: is.atomic(int_var)
#> [1] TRUE
5:
6: dbl_var <- c(1, 2.5, 4.5)
7: typeof(dbl_var)
#> [1] "double"
8: is.double(dbl_var)
#> [1] TRUE
9: is.atomic(dbl_var)
#> [1] TRUE
```

Observação

Uma forma geral para checar se um vetor é formado por números inteiros ou de precisão dupla (double) é utilizando a função `is.numeric()`.

Exemplo:

```
is.numeric(int_var)
#> [1] TRUE
is.numeric(dbl_var)
#> [1] TRUE
```

Coerção

Pergunta: Ocorrerá erro se for tentado criar um vetor, em `R`, com objetos de tipo diferente?

Coerção

Pergunta: Ocorrerá erro se for tentado criar um vetor, em R, com objetos de tipo diferente?

Coerção

Pergunta: Ocorrerá erro se for tentado criar um vetor, em R, com objetos de tipo diferente?

Resposta direta: Não.

Coerção

Pergunta: Ocorrerá erro se for tentado criar um vetor, em R, com objetos de tipo diferente?

Uma resposta um pouco maior: Muito embora não ocorrerá erro ao tentarmos criar um vetor com elementos de diferentes tipos, é importante entender que o vetor criado será formado por elementos do mesmo tipo, como dito anteriormente. Neste caso, R irá converter os elementos de um vetor para um tipo específico. Para saber qual é esse tipo, é preciso entender a regra de coerção da linguagem R.

Coerção

Regra de Coerção

Se um vetor for criado com elementos de tipos básicos diferentes, os elementos serão convertidos para o tipo mais flexível. Logo abaixo segue uma sequência de desigualdades em que $A > B$ significa que o tipo de A é mais flexível que o tipo de B.

Coerção

Regra de Coerção

Se um vetor for criado com elementos de tipos básicos diferentes, os elementos serão convertidos para o tipo mais flexível. Logo abaixo segue uma sequência de desigualdades em que $A > B$ significa que o tipo de A é mais flexível que o tipo de B.

$$\text{character} > \text{double} > \text{integer} > \text{logical}.$$

Coerção

Regra de Coerção

Se um vetor for criado com elementos de tipos básicos diferentes, os elementos serão convertidos para o tipo mais flexível. Logo abaixo segue uma sequência de desigualdades em que $A > B$ significa que o tipo de A é mais flexível que o tipo de B.

$$\text{character} > \text{double} > \text{integer} > \text{logical}.$$

Exemplo:

```
1: str(c("a", 1))
#> chr [1:2] "a" "1"
2: str(c(T, T, F, 2))
#> num [1:4] 1 1 0 2
```

Coerção

Exemplo: Nesse exemplo é possível observar que se o tipo mais flexível é **integer** ou **double** e o tipo menos flexível é **logical**, os valores booleanos `TRUE` será convertido para **1** e `FALSE` será convertido para **0**.

Essa característica de converter tipos booleanos para **0** (se `FALSE`) ou **1** (se `TRUE`) permite, por exemplo, com que possamos contar o número de **TRUE's** ou calcular a proporção destes.

Coerção

Exemplo:

```
x <- c(FALSE, FALSE, TRUE)
as.numeric(x)
#> [1] 0 0 1

# Total de TRUEs
sum(x)
#> [1] 1

# Proporção de TRUEs
mean(x)
#> [1] 0.3333
```

Coerção

Nota: Três novas funções foram apresentadas no código logo acima:

Coerção

Nota: Três novas funções foram apresentadas no código logo acima:

1. `as.numeric()`: Esta função aplicada ao vetor `x` forçará o vetor a coagir para um tipo de dados específico;

Coerção

Nota: Três novas funções foram apresentadas no código logo acima:

1. `as.numeric()`: Esta função aplicada ao vetor `x` forçará o vetor a coagir para um tipo de dados específico;
2. `sum()`: Esta função aplicada ao vetor `x` fornecerá a soma do vetor;

Coerção

Nota: Três novas funções foram apresentadas no código logo acima:

1. `as.numeric()`: Esta função aplicada ao vetor `x` forçará o vetor a coagir para um tipo de dados específico;
2. `sum()`: Esta função aplicada ao vetor `x` fornecerá a soma do vetor;
3. `mean()`: Esta função aplicada ao vetor `x` fornecerá a média aritmética do vetor.

Coerção

Nota Importante: `as.numeric()` é um objeto mais geral e converterá os elementos de um vetor para o tipo de dados double. Porém, também é possível utilizar o objeto `as.double()` para produzir o mesmo resultado. No momento, o que pode ser dito para não parecer estranho ao iniciante da linguagem `R` é que `R` possui, basicamente, dois sistemas de orientação a objeto que são mais utilizados, sendo eles o sistema **S3** ao qual o objeto `as.double()` pertence e o sistema **S4** ao qual o objeto `as.numeric()` pertence.

Dica: Na grande maioria dos casos, qualquer uma das opções será uma escolha conveniente.

Coerção

Da mesma forma que podemos converter os elementos de um vetor para o tipo específico double, também podemos fazer isto com os outros tipos de dados.

Coerção

Da mesma forma que podemos converter os elementos de um vetor para o tipo específico double, também podemos fazer isto com os outros tipos de dados.

- `as.character()`: convertendo os elementos do vetor para o tipo básico **character**;

Coerção

Da mesma forma que podemos converter os elementos de um vetor para o tipo específico double, também podemos fazer isto com os outros tipos de dados.

- `as.character()`: convertendo os elementos do vetor para o tipo básico **character**;
- `as.integer()`: convertendo os elementos do vetor para o tipo básico **integer**;

Coerção

Da mesma forma que podemos converter os elementos de um vetor para o tipo específico double, também podemos fazer isto com os outros tipos de dados.

- `as.character()`: convertendo os elementos do vetor para o tipo básico **character**;
- `as.integer()`: convertendo os elementos do vetor para o tipo básico **integer**;
- `as.logical()`: convertendo os elementos do vetor para o tipo básico **logical**.

Coerção

Lembre-se

A coerção ocorre, frequentemente de forma automática. A maioria das funções matemáticas ($+$, log, abs, etc.) coagirá para um valor do tipo double ou inteiro. Já os operadores lógicos (&, $|$, any, etc.) coagirão para um valor lógico. Mas lembre-se que é possível forçar a coação com as funções `as.logical()`, `as.character()`, `as.integer()` e `as.numeric()`.

Nota: Podem existir casos de surgimento de mensagens de avisos (Warnings) em que se a coerção implicar em perca de informação.

Algumas funções matemáticas

Tabela: Algumas funções matemáticas/estatísticas aplicadas ao vetor `x`.

Função em R	Finalidade
`sum(x)`	Retorna a soma do vetor `x`
`mean(x)`	Retorna a média do vetor `x`
`abs(x)`	Retorna o vetor com os valores absolutos
`sqrt(x)`	Retorna o vetor com as raízes quadradas
`sort(x)`	Retorna o vetor `x` ordenado de forma crescente
`sort(x, decreasing=TRUE)`	Retorna o vetor `x` ordenado de forma decrescente
`exp(x)`	Retorna o vetor exponencial
`log(x)`	Retorna o vetor logaritmo natural
`log(x, base=10)`	Retorna o vetor com o logaritmo na base 10
`sin(x)`, `cos(x)`, `tan(x)`	Retorna o vetor com resultados trigonométricos
`length(x)`	Cardinalidade do vetor `x`
`factorial(x)`	Fatorial dos elementos de `x`
`choose(3,2)`	Combinação três, dois a dois
`summary(x)`	Medidas descritivas do vetor `x`

Help

Figura: Procure ajuda!

Até o momento foram apresentadas algumas funções. Muitas outras funções irão surgir no decorrer das aulas. A melhor forma de conseguir ajuda sobre uma determinada função além de estudar pelas aulas ministradas é ler a documentação destas funções. Na documentação apresenta exemplos do uso da função o que poderá ajudar bastante.

Help

Figura: Procure ajuda!

Para algumas funções da tabela acima, por exemplo, vimos o emprego de alguns argumentos. Maiores detalhes sobre estes argumentos bem como outros argumentos suportados por uma função poderão ser encontrados nas documentações das funções.

Help

Existem basicamente duas formas equivalentes de acessar a documentação de uma função disponibilizada na linguagem base ou empacotadas que pode ser `?nome_da_funcao` ou `help("nome_da_funcao")`.

Exercício: Descubra, lendo a documentação, o que fazem as funções `ls` e `rm`.

Uma outra forma de encontrar informações sobre uma determinada função é utilizando a função `find()`.

```
1: find("ls")
#> [1] "package:base"
```

Help

Nota: A função `find("funcao")` irá procurar a existência da função `funcao()` nos pacotes instalados **e** carregados na linguagem.

Help

Nota: A função `find("funcao")` irá procurar a existência da função `funcao()` nos pacotes instalados **e** carregados na linguagem.

A função `find(nome_da_funcao)` fornece o pacote ao qual o nome da função pertence. No caso do exemplo logo acima, a função `ls` pertence ao pacote **base** da linguagem `R`. O pacote **base** está contido em toda instalação básica de `R`.

Help

Nota: A função `find("funcao")` irá procurar a existência da função `funcao()` nos pacotes instalados **e** carregados na linguagem.

A função `find(nome_da_funcao)` fornece o pacote ao qual o nome da função pertence. No caso do exemplo logo acima, a função `ls` pertence ao pacote **base** da linguagem `R`. O pacote **base** está contido em toda instalação básica de `R`.

Há outros pacotes, sob os termos da licença $\mathrm{GPL} \geq 2$ com diversas funções disponíveis para os programadores de `R`, porém, daremos mais detalhes mais a frente.

Help

Um outro comando bastante útil, principalmente nos casos em que lembramos parcialmente o nome de uma função R é o comando `apropos()`.

Uso: `apropos("nome_da_funcao")`.

Exemplo:

```
1: apropos("ps")
#> [1] ".standard_regexps"   "cairo_ps"   "dev.copy2eps"
#> [4] "fileSnapshot"        "Ops"        "Ops.data.frame"
#> [7] "Ops.Date"            "Ops.difftime" "Ops.factor"
#> [10] "Ops.numeric_version" "Ops.ordered" "Ops.POSIXt"
#> [13] "ps.options"          "psigamma"    "psignrank"
#> [16] "pso"                 "setEPS"      "setPS"
#> [19] "supsmu"              "zapsmall"
```

Help

Como mencionado anteriormente, as documentações de uma função R de funções presentes no pacote **base** ou empacotadas em outros pacotes que são distribuídos pela comunidade R possuem exemplos de utilização. Podemos acessar a documentação e copiar e executar um exemplo específico no prompt de comando ou podemos executar todos exemplos de uso de uma função por meio do comando `example()`.

Exemplo:

```
1: example(rm)

rm> tmp <- 1:4
rm> ## work with tmp  and cleanup
rm> rm(tmp)
```

Help

E como faço para listar todos os objetos de um determinado pacote?

Help

E como faço para listar todos os objetos de um determinado pacote?

Resposta: Corra o comando:

```
objects(grep("base",search()))
```

Nesse exemplo, será retornado uma lista com todos os objetos contidos no pacote **base**. Atualmente, o pacote **base** apresenta 1218 objetos.

Help

E como faço para listar todos os objetos de um determinado pacote?

Resposta: Corra o comando:

```
objects(grep("base",search()))
```

Nesse exemplo, será retornado uma lista com todos os objetos contidos no pacote **base**. Atualmente, o pacote **base** apresenta 1218 objetos.

Exercício: O que faz o código abaixo?

```
length(objects(grep("base",search())))
```

Help

Exercício: Para que serve o argumento `ignore.case` da função `grep()`? Lembre-se, agora você já sabe consultar a documentação da linguagem `R`. Apresente um exemplo do uso da função `grep()` modificando o argumento `ignore.case`.

Help

Exercício: Para que serve o argumento `ignore.case` da função `grep()`? Lembre-se, agora você já sabe consultar a documentação da linguagem `R`. Apresente um exemplo do uso da função `grep()` modificando o argumento `ignore.case`.

Entendendo melhor o código acima, note que `search()` é uma função passada como argumento para a função `grep()` que por sua vez é uma função que é passada como argumento para a função `objects` e que esta é uma função que é passada para a função `length()`.

Help

Exercício: Para que serve o argumento `ignore.case` da função `grep()`? Lembre-se, agora você já sabe consultar a documentação da linguagem `R`. Apresente um exemplo do uso da função `grep()` modificando o argumento `ignore.case`.

Entendendo melhor o código acima, note que `search()` é uma função passada como argumento para a função `grep()` que por sua vez é uma função que é passada como argumento para a função `objects` e que esta é uma função que é passada para a função `length()`.

Pergunta: O que cada uma dessas funções faz?

Help

Respostas:

1. `search()`: Lista um vetor de caracteres com os nomes dos pacotes carregados.

Help

Respostas:

1. `search()`: Lista um vetor de caracteres com os nomes dos pacotes carregados.
2. `grep("base", search())`: Busca `"base"` no vetor obtido como retorno da função `search()`. A função `grep()` retorna as posições do vetor retornado por `search()` que contém "strings"que contenham a palavra `"base"`.

Help

Respostas:

1. `search()`: Lista um vetor de caracteres com os nomes dos pacotes carregados.
2. `grep("base", search())`: Busca `"base"` no vetor obtido como retorno da função `search()`. A função `grep()` retorna as posições do vetor retornado por `search()` que contém "strings"que contenham a palavra `"base"`.
3. `objects()`: Lista todos os objetos do pacote correspondente.

Help

Exercício: Explique passo a passo a sintaxe abaixo. Certifique-se que tenha entendido de fato o código abaixo.

```
search()[(grep("gr", search()))]
```

E se eu misturar letras maiúsculas e minúsculas do nome da função?

Help

Sério mesmo? Não há nenhuma forma de tentar encontrar informações sobre uma função caso troque letras maiúsculas por minusculas ou o contrário?

Help

Resposta: Claro que tem! Mas não queira mais do que isso.

Help

Resposta: Claro que tem! Mas não queira mais do que isso.

Help

Nestas situações em que não compreendemos se uma determinada letra é escrita em maiúscula ou minúscula, utilize a função `??`.

Uso: `??nome_da_funcao`.

Exemplo: Corra o código `??Mean` e `??mean` ou `??MeaN`. Diversas funções que fazem dessas expressões regulares irão ser retornadas.

Help

Nestas situações em que não compreendemos se uma determinada letra é escrita em maiúscula ou minúscula, utilize a função `??`.

Uso: `??nome_da_funcao`.

Exemplo: Corra o código `??Mean` e `??mean` ou `??MeaN`. Diversas funções que fazem dessas expressões regulares irão ser retornadas.

Dica: Na internet também é possível obter diversas informações sobre a linguagem `R`. Há diversos materiais (documentação oficial, livros, apostilas, blogs, grupos, etc) focado para o aprendizado da linguagem `R`.

Listas

Litas são também vetores não-atômicos. As listas diferem de vetores atômicos pelo fato de poder conter elementos de tipos diferentes, incluindo listas. Para se construir uma lista, deve ser utilizado o comando `list()` ao invés de `c()`.

Exemplo: Criando uma lista em `R`.

```
1: x <- list(1:3, "a", c(TRUE, FALSE, TRUE), c(2.3, 5.9))
2: str(x)
#> List of 4
#> $ : int [1:3] 1 2 3
#> $ : chr "a"
#> $ : logi [1:3] TRUE FALSE TRUE
#> $ : num [1:2] 2.3 5.9
```

Listas

Figura: Estrutura hierárquica do objeto `x`.

Nota: No exemplo anterior foi passado para a lista `x` a expressão `1:3`. Tal expressão, em `R`, significa que estamos criando um vetor com **sequência** de valores de intervalos unitários de 1 a 3. Perceba que os limites são inclusos nesse caso.

Exemplo: Corra o código que segue:

```
seq1 <- 1.7:7.7
str(seq1)
#> num [1:7] 1.7 2.7 3.7 4.7 5.7 6.7 7.7
seq2 <- 1.7:6.1
str(seq2)
#> num [1:5] 1.7 2.7 3.7 4.7 5.7
```

Nota: Caso haja o interesse de criar uma sequência de valores de intervalos não unitários, isso é possível utilizando a função `seq()` e alterando o parâmetro `by` utilizado para essa finalidade. Para maiores detalhes sobre a função e seus outros argumentos, basta consultar a documentação, isto é, faça `help(seq)` ou `?seq`. **Ex**: Corra o código `seq(from = 1, to = 3)`.

Exercício: Construa um vetor de números pares de 2 a 100 e um outro vetor de números ímpares de números de 1 a 99.

Exercício: Construa uma sequência de números no intervalo fechado $[1, 100]$ por intervalos de comprimento 0.1. Construa uma outra sequência no intervalo fechado $[1, 100]$ com 37 elementos.

Listas

Assim com nos vetores atômicos, as listas também podem apresentar recursividade. Corra o exemplo que segue.

Exemplo:

```
x <- list(list(list(list())))
str(x)
#> List of 1
#>  $ :List of 1
#>   ..$ :List of 1
#>   .. ..$ : list()
is.recursive(x)
#> [1] TRUE
```

Listas

```
1: x <- list(list(list(1)))
2: str(x)
```

Figura: Exemplo de uma lista recursiva (concatenação de listas).

Nota: Ao contrário de listas, note que a concatenação de um vetor não fornece uma estrutura recursiva.

Uso do comando `c()` em listas

O comando `c()` poderá ser utilizado para combinar várias listas em uma. Caso venha ser dada uma combinação atômica de vetores e listas, a função `c()` irá coagir os vetores para uma lista.

Exemplo: Compare as listas armazenadas nas variáveis `x` e `y` no exemplo que segue.

```
x <- list(list(1, 2), c(3, 4))
y <- c(list(1, 2), c(3, 4))
str(x)
#> List of 2
#>  $ :List of 2
#>   ..$ : num 1
#>   ..$ : num 2
#>  $ : num [1:2] 3 4
```

Listas

Exemplo: Compare as listas armazenadas nas variáveis `x` e `y` no exemplo que segue.

```
4: str(y)
#> List of 4
#> $ : num 1
#> $ : num 2
#> $ : num 3
#> $ : num 4
```

Listas

Exemplo: Compare as listas armazenadas nas variáveis `x` e `y` no exemplo que segue.

```
4: str(y)
#> List of 4
#> $ : num 1
#> $ : num 2
#> $ : num 3
#> $ : num 4
```

Nota: Seja `x` uma estrutura de dados qualquer. No `R` podemos utilizar a função `View(x)` para visualizar de forma tabelada tal estrutura de dados. Equivale a clicar na aba **Environment** no **RStudio** e selecionar o objeto.

Listas

Exercício: Da mesma forma que o exemplo anterior, corra o código abaixo e assimile as saídas.

```
1: l <- c(list(c(3,2),"a"),c(1,2))
2: str(l)
#> List of 4
#> $ : num [1:2] 3 2
#> $ : chr "a"
#> $ : num 1
#> $ : num 2
3: m <- list(list(c(3,2),"a"),c(1,2))
4 : str(m)
#> List of 2
#> $ :List of 2
#>  ..$ : num [1:2] 3 2
#>  ..$ : chr "a"
#> $ : num [1:2] 1 2
```

Listas

Nota: É possível checar se um objeto é um lista. Para isso, utiliza-se a função `is.list()`. Nesse caso, será retornado um valor booleano (**TRUE** ou **FALSE**). Além disso, o comando `typeof()` também poderá ser utilizado, porém, neste caso será retornado uma string informando o tipo do objeto. No caso do objeto venha a ser uma lista, o retorno observado será `"list"`.

Exemplo: Corra o código que segue:

```
is.list(mtcars)
#> [1] TRUE

mod <- lm(mpg ~ wt, data = mtcars)
is.list(mod)
#> [1] TRUE
```

Algumas observações para o exemplo anterior:

1. O objeto `mtcars` nada mais é do que um é um conjunto de dados disponível em qualquer instalação básica da linguagem `R` no pacote **datasets**. Para maiores detalhes, faça `help(mtcars)`;
2. A função `lm` é responsável com o ajustamento de modelos lineares. No caso do exemplo, foi ajustado um modelo linear simples de regressão com as variáveis **mpg** e **wt** pertencentes ao conjunto de dados **mtcars**. Para maiores detalhes, faça `help(lm)`.

Listas

Nota: Também é possível converter uma lista para um vetor atômica utilizando o comando `unlist()`. Porém, lembre-se, os elementos da lista serão coagidos para o tipo de dados mais flexível.

Exemplo: Corra o código que segue:

```
m <- list(list(c(3,2),"a"),c(1,2))
unlist(m)
#> [1] "3" "2" "a" "1" "2"
```

Listas

Exercícios: Corra os códigos abaixo e explique o por quê das saídas obtidas.

a) Qual a saída esperada para os códigos `c(1, FALSE)`, `c("a", 1)`, `c(list(1), "a")`, `c(TRUE, 1L)`? Explique.

b) Por que `1 == "1"` é igual à **TRUE**? Além disso, responda o por quê `-1 < FALSE` é igual à **TRUE**.

Nomes

Muitas vezes é útil nomear um vetor. Em R, é possível nomear um vetor de três maneiras:

Nomes

Muitas vezes é útil nomear um vetor. Em R, é possível nomear um vetor de três maneiras:

1. Nomeando no momento em que criamos o vetor x: `x <- c(a = 1, b = 2, c = 3);`

Nomes

Muitas vezes é útil nomear um vetor. Em R, é possível nomear um vetor de três maneiras:

1. Nomeando no momento em que criamos o vetor x: `x <- c(a = 1, b = 2, c = 3);`
2. Nomeando um vetor x previamente criado: `x <- 1:3; names(x) <- c("a", "b", "c");`

Nomes

Muitas vezes é útil nomear um vetor. Em R, é possível nomear um vetor de três maneiras:

1. Nomeando no momento em que criamos o vetor x: `x <- c(a = 1, b = 2, c = 3);`
2. Nomeando um vetor x previamente criado: `x <- 1:3; names(x) <- c("a", "b", "c");`
3. Utilizando a função setNames(): `x <- setNames(1:3, c("a", "b", "c"))`.

Nomes

Exemplo: Corra o código que segue:

```
y <- c(a = 1, 2, 3)
names(y)
#> [1] "a" "" ""

z <- c(1, 2, 3)
names(z)
#> NULL
```

Nota: Nem todos os elementos de um vetor precisa ter um nome. Caso algum deles não sejam será retornado uma string varia. Caso nenhum elemento venha ter um nome, `NULL` será retornado.

Nomes

Caso tenhamos o interesse de eliminar os nomes dos elementos de um vetor poderemos utilizar a função `unname()` ou atribuir `NULL` à `names(x)`.

Exemplo: Corra o código que segue:

```
x <- c(a = 1, b = 2, c = 3)
x
#> a b c
#> 1 2 3
x <- unname(x)
x
#> [1] 1 2 3
x <- c(a = 1, b = 2, c = 3)
names(x) <- NULL
```

Fator

O que é um fator?

Fator é um vetor que pode conter apenas valores predefinidos, e é usado para armazenar dados categóricos.

Exemplo:

```
1: x <- factor(c("a", "b", "b", "a"))
2: x
#> [1] a b b a
#> Levels: a b
3: class(x)
#> [1] "factor"
4: levels(x)
#> [1] "a" "b"
```

Fator

Nota: Caso o programador tente atribuir um elemento que não corresponde aos níveis admitidos pelo fator ocorrerá uma mensagem de erro e na posição do elemento atribuído irá conter `NA`.

Exemplo: Corra o código que segue:

```
1: x <- factor(c("a", "b", "b", "a"))
2: x[2] <- "c"
3:
#> Warning: invalid factor level, NA generated
4: x
#> [1] a <NA> b a
#> Levels: a b
```

Fator

Importante: Não podemos combinar fatores.

Exemplo: Corra o código que segue:

```
# Obs: Não é possível combinar fator. Algo inesperado
# será retornado.
1: c(factor("a"), factor("b"))
#> [1] 1 1
```

Fator

Podemos utilizar a função `table()` para construir uma tabela de frequência com um objeto da classe **factor**.

Exemplo: Corra o código abaixo:

```
sex_char <- c("m", "m", "m","f","f","f,"f")
sex_factor <- factor(sex_char, levels = c("m", "f"))
table(sex_factor)
#> sex_factor
#> m f
#> 3 4
```

Fator

Exemplo: Podemos atribuir os níveis em que um fator poderá assumir após sua criação:

```
1: sex_char <- c("f", "f" ,"f", "f", "f")
2: sex_factor <- factor(sex_char)
3: levels(sex_factor) <- c("m", "f")
4: table(sex_factor)
#> sex_factor
#> m f
#> 5 0
```

Fator

Exercício: O que faz a função `rev()` e o objeto `letters`? Discuta as diferenças de `f1`, `f2` e `f3`.

```
1: f1 <- factor(letters)
2: levels(f1) <- rev(levels(f1))
3: f2 <- rev(factor(letters))
4: f3 <- factor(letters, levels = rev(letters))
5: f1
#> [1] z y x w v u t s r q p o n m l k j i h g f e d c
#> Levels: z y x w v u t s r q p o n m l k j i h g f e d
6: f2
#> [1] z y x w v u t s r q p o n m l k j i h g f e d c
#> Levels: a b c d e f g h i j k l m n o p q r s t u v w
7: f3
#> [1] a b c d e f g h i j k l m n o p q r s t u v w x
#> Levels: z y x w v u t s r q p o n m l k j i h g f e d
```

Removendo objetos

Na estatística trabalhamos com objetos que, em geral, consomem muita memória. Dessa forma, parece bem conveniente entendermos como remover tais objetos da memória. Porém, precisamos antes identificar quais objetos estão carregados na memória. Para isso utilize a função `ls()` ou `objetcs()`.

Exemplo:

```
s1 <- "Eu"
s2 <- "amo"
s3 <- "a"
s4 <- "ESTATÍSTICA."
s5 <- paste(s1, s2, s3, s4, sep = " ")
ls()
objects.size(ls())
```

Removendo objetos

Exercício: Explique resumidamente a utilidade da função `paste()`. Obtenha detalhes na documentação da linguagem. Apresente dois exemplos do uso da função `paste()` fazendo uso dos argumentos `sep` e `collapse`, respectivamente. Perceba a diferença do uso desses argumentos.

É possível remover todos objetos com um único comando? Como?

Resposta: Sim. Utilize o comando `rm(list = ls(all = TRUE))`.

Exercício: Explique o código `rm(list = ls(all = TRUE))`.

Removendo objetos

Importante

Em situações em que desejamos remover objetos "**grandes**" (objetos que ocupam muita memória) é sempre interessante executar a função `gc()`. Ao removermos objetos `R` utilizando a função `rm()` automaticamente o `R` irá devolver ao sistema operacional a memória ocupada por esse objeto. Porém, não necessariamente isso ocorrerá de forma imediata.

Removendo objetos

Importante

Em situações em que desejamos remover objetos "**grandes**" (objetos que ocupam muita memória) é sempre interessante executar a função `gc()`. Ao removermos objetos `R` utilizando a função `rm()` automaticamente o `R` irá devolver ao sistema operacional a memória ocupada por esse objeto. Porém, não necessariamente isso ocorrerá de forma imediata.

Uma chamada da função `gc()` faz com que uma coleta de lixo ocorra

Removendo objetos

Exemplo: Corra o exemplo que segue.

```
x <- rnorm(1e6) # 1e6 = 1 * 10^6
# Busque nas documentações detalhes sobre as funções
# print() e object.size().
print(object.size(x), units = "Mb")
rm(x)
gc()
```

Removendo objetos

Exemplo: Corra o exemplo que segue.

```
x <- rnorm(1e6) # 1e6 = 1 * 10^6
# Busque nas documentações detalhes sobre as funções
# print() e object.size().
print(object.size(x), units = "Mb")
rm(x)
gc()
```

Nota: Observe e entenda o código abaixo.

```
1e6 == 1 * 10^6
#> TRUE
1E6 == 1 * 10^6
#> TRUE
```

Matrizes

Matrizes e Arrays

Adicionar um atributo `dim()` à um vetor atômico permite que ele se comporte como uma matriz multidimensional. O caso especial do **array** é o que chamamos de **matriz** que tem apenas duas dimensões.

Matrizes e Arrays

Adicionar um atributo `dim()` à um vetor atômico permite que ele se comporte como uma matriz multidimensional. O caso especial do **array** é o que chamamos de **matriz** que tem apenas duas dimensões.

Matrizes são usadas comumente como parte da "maquinaria" matemática e estatística. O uso do array é um pouco mais raro, mas é fundamental estar ciente do seu uso.

Matrizes e Arrays

Adicionar um atributo `dim()` à um vetor atômico permite que ele se comporte como uma matriz multidimensional. O caso especial do **array** é o que chamamos de **matriz** que tem apenas duas dimensões.

Matrizes são usadas comumente como parte da "maquinaria" matemática e estatística. O uso do array é um pouco mais raro, mas é fundamental estar ciente do seu uso.

Exercício: Leia a documentação das funções `matrix()` e `array()`. Procure entender os seus argumentos.

Matrizes e Arrays

Adicionar um atributo `dim()` à um vetor atômico permite que ele se comporte como uma matriz multidimensional. O caso especial do **array** é o que chamamos de **matriz** que tem apenas duas dimensões.

Matrizes são usadas comumente como parte da "maquinaria" matemática e estatística. O uso do array é um pouco mais raro, mas é fundamental estar ciente do seu uso.

Exercício: Leia a documentação das funções `matrix()` e `array()`. Procure entender os seus argumentos.

Exercício: Leia a documentação da função `dim()` e construa um vetor atômico de tamanho 10 e em seguida construa uma `M` matriz de dimensão 5 por 2. Utilizando a matriz `M`, construa o vetor `myvector`.

Matrizes e Arrays

Matrizes e arrays são criados com as funções matrix e array, respectivamente, ou utilizando a forma de atribuição com o comando `dim()`.

Exemplo: Corra o exemplo que segue:

```
# Dois argumentos escalares são passados
# representando linha e coluna
a <- matrix(1:6, ncol = 3, nrow = 2)

# Passando o vetor de dimensões do array.
b <- array(1:12, dim = c(2, 3, 2))

# Modificando o objeto c com a função dim().
c <- 1:6
dim(c) <- c(3,2)
```

Matrizes e Arrays

Nota: Podemos utilizar as funções `length()`, `nrow()` e `ncol()` para obter a quantidade de elementos, número de linhas e número de colunas, respectivamente, de uma matriz ou array.

Exemplo: Corra o exemplo abaixo:

```
m <- array(1:12, dim = c(2,2,3))
length(m)
#> [1] 12
ncol(m)
#> [1] 2
nrow(m)
#> [1] 2
```

Nota: A função `dim()` poderá ser utilizada para obter o vetor com as dimensões linha e coluna, respectivamente. Faça `dim(m)` no exemplo anterior.

Matrizes e Arrays

Muitas vezes é útil nomear cada uma das dimensões de uma matriz ou array. Inclusive, em situações em que queremos acessar um subconjunto de um vetor, matriz ou array, podemos levar em conta tais nomes. Veremos isto mais a frente. No momento, focaremos apenas em atribuir nomes às dimensões.

No caso de matrizes (array de dimensão dois) utilizaremos as funções `rownames()` e `colnames()` para atribuir nomes ás dimensões linha e coluna de uma matriz, respectivamente.

Exemplo: Corra o código seguinte:

```
a <- matrix(1:6, ncol = 3, nrow = 2)
rownames(a) <- c("A", "B")
colnames(a) <- c("a", "b", "c")
a
#>   a b c
#> A 1 3 5
#> B 2 4 6
```

Nota: Caso o array possua mais de duas dimensões, utilize a função `dimnames()` aplicado ao array. Deverá ser passado uma lista com os nomes de cada uma das dimensões do objeto.

Exemplo: Corra o código:

```
b <- array(1:12, c(2, 3, 2))
dimnames(b) <- list(c("one", "two"), c("a", "b", "c"),
                    c("A", "B")); b
#> , , A
#>     a b c
#> one 1 3 5
#> two 2 4 6

#> , , B
#>     a  b  c
#> one 7  9 11
#> two 8 10 12
```

Importante observar

Fazer `b <- array(1:12, c(2, 3, 2))` equivale a fazer `b <- array(data = 1:12, dim = c(2, 3, 2))`.

Notas:

Importante observar

Fazer `b <- array(1:12, c(2, 3, 2))` equivale a fazer `b <- array(data = 1:12, dim = c(2, 3, 2))`.

Notas:

1. A diferença é que no primeiro caso não foi especificado os nomes dos argumentos, preocupação esta que foi tomada no segundo caso. O primeiro caso funcionará apenas se a ordem de passagem dos argumentos respeitar a ordem dos argumentos especificada em sua documentação. A ordem poderá ser alterada desde que os nomes dos argumentos venha estar devidamente especificado.

Importante observar

Fazer `b <- array(1:12, c(2, 3, 2))` equivale a fazer `b <- array(data = 1:12, dim = c(2, 3, 2))`.

Notas:

1. A diferença é que no primeiro caso não foi especificado os nomes dos argumentos, preocupação esta que foi tomada no segundo caso. O primeiro caso funcionará apenas se a ordem de passagem dos argumentos respeitar a ordem dos argumentos especificada em sua documentação. A ordem poderá ser alterada desde que os nomes dos argumentos venha estar devidamente especificado.
2. **É sempre uma boa prática de programação especificar os nomes dos argumentos**.

Matrizes e Arrays

Importante

Os arrays são úteis quando necessitamos armazenar uma sequência de matrizes que poderão ser utilizadas posteriormente no código, por exemplo para fazer um produto de duas matrizes.

Matrizes e Arrays

Importante

Os arrays são úteis quando necessitamos armazenar uma sequência de matrizes que poderão ser utilizadas posteriormente no código, por exemplo para fazer um produto de duas matrizes.

Exemplo: Corra o código abaixo.

```
1: myarray <- array(data = NA, dim = c(2, 2, 3))
2: a1 <- matrix(data = c(1,2,3,4), nrow = 2, ncol = 2)
3: a2 <- 2 * a1
4: a3 <- 3 * a2
5: myarray[ , , 1] <- a1
6: myarray[ , , 2] <- a2
7: myarray[ , , 3] <- a3
8: myarray
```

Matrizes e Arrays

Exemplo: Corra o código que segue. O código que segue justifica bem o porquê é uma boa prática de programação considerar os nomes dos argumentos de uma função.

```
# Não tente rodar o código comentado abaixo
# Você poderá não ter memória =).
# b <- array(c(2, 3, 2), 1:12)

c <- array(dim = c(2, 3, 2), data = 1:12)
```

Nota: Como apresentado anteriormente, a função `dim()` fornece um vetor com as dimensões linha e coluna de um array. Se o interesse é obter especificamente a dimensão da linha ou coluna, utilizamos a função `nrow()` ou `ncol()`, respectivamente.

Matrizes e Arrays

É possível concatenar matrizes por linha ou coluna utilizando as funções `rbind()` e `cbind()`, respectivamente.

Exemplo: Corra o código abaixo:

```
v1 <- matrix(1:12, ncol = 4, nrow = 3)
w1 <- matrix(1:8, ncol = 4, nrow = 2 )
rbind(v1,w1)

v2 <- matrix(1:12, ncol = 2, nrow = 4)
w2 <- matrix(1:12, ncol = 3, nrow = 4 )
cbind(v2,w2)
```

Observação Importante

Os elementos de uma matriz poderão ser listas. Trata-se de uma estrutura de dados "exotérica" mas que pode ser útil se desejarmos organizar nossas listas em uma estrutura semelhante à uma grade.

Exemplo: Corra o código que segue:

```
1: l <- list(1:3, "a", TRUE, 1.0)
2: dim(l) <- c(2, 2)
3: l
#>      [,1]        [,2]
#> [1,] Integer,3   TRUE
#> [2,] "a"         1
```

Exercícios: Para complementar alguns detalhes, responda o exercício abaixo.

1. O que a funções `dim()` retorna quando aplicado a um vetor?

Matrizes e Arrays

Exercícios: Para complementar alguns detalhes, responda o exercício abaixo.

1. O que a funções `dim()` retorna quando aplicado a um vetor?
2. Se `is.matrix(x)` retornar `TRUE`, o que irá retornar `is.array(x)`?

Matrizes e Arrays

Exercícios: Para complementar alguns detalhes, responda o exercício abaixo.

1. O que a funções `dim()` retorna quando aplicado a um vetor?
2. Se `is.matrix(x)` retornar `TRUE`, o que irá retornar `is.array(x)`?
3. Como você descreveria os três objetos a seguir? Sem executar o código, o que você espera como resultado?
   - `x1 <- array(1:5, c(1, 1, 5))`
   - `x2 <- array(1:5, c(1, 5, 1))`
   - `x3 <- array(1:5, c(5, 1, 1))`

Exercícios: Responda os exercícios abaixo:

1. Crie três vetores com três elementos **inteiros**, sejam eles os objetos `x`, `y`, `z`. Combine os três vetores para se tornar uma matriz 3×3 em que cada coluna representa um vetor. Altere os nomes das linhas para `a`, `b` e `c`, respectivamente.

Exercícios: Responda os exercícios abaixo:

1. Crie três vetores com três elementos **inteiros**, sejam eles os objetos `x`, `y`, `z`. Combine os três vetores para se tornar uma matriz 3×3 em que cada coluna representa um vetor. Altere os nomes das linhas para `a`, `b` e `c`, respectivamente.

2. Crie um vetor `v` com 15 valores **inteiros** e converta-o em uma matriz `M` (5×3) . Faça isso de duas formas diferentes. Além disso, nomeie as linhas de `l1` à `l5` e as colunas de `c1` à `c3`. Depois, crie a matriz `N` a partir da matriz `M` em que `N` **não** possui nomes nas linhas e colunas.

Data Frames

Um **quadro de dados** (**data frame**) é a forma mais comum de armazenar dados em R.

R Data Editor

Copy Paste Quit

	row.names	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
1	Mazda RX4	21	6	160	110	3.9	2.62	16.46	0	1	4	4
2	Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0	1	4	4
3	Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1	1	4	1
4	Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
5	Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0	0	3	2
6	Valiant	18.1	6	225	105	2.76	3.46	20.22	1	0	3	1
7	Duster 360	14.3	8	360	245	3.21	3.57	15.84	0	0	3	4
8	Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1	0	4	2
9	Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1	0	4	2
10	Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1	0	4	4
11	Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1	0	4	4

Figura: Data frame em R.

Data Frames

Nota: Por se tratar de uma estrutura bidimensional assim como matrizes, todos os comandos (`dim()`,`names()`, `colnames()`, `ncol()`, `nrow()`, etc) apresentados anteriormente ao tratarmos de matrizes poderão ser considerado à um data frame.

Data Frames

Nota: Por se tratar de uma estrutura bidimensional assim como matrizes, todos os comandos (`dim()`,`names()`, `colnames()`, `ncol()`, `nrow()`, etc) apresentados anteriormente ao tratarmos de matrizes poderão ser considerado à um data frame.

Observação Importante

A forma mais conveniente de organizar um base de dados de modo a facilitar as análises estatísticas é considerar um data frame. No artigo **Tidy Data** publicado em 2014 por **Hadley Wickham** no **Journal of Statistical Software**. No artigo o autor aborta a dificuldade de se organizar uma base de dados e como tornar a limpeza (tratamento dos dados) tão fácil e eficaz quanto possível.

Data Frames

Nota: Por se tratar de uma estrutura bidimensional assim como matrizes, todos os comandos (`dim()`,`names()`, `colnames()`, `ncol()`, `nrow()`, etc) apresentados anteriormente ao tratarmos de matrizes poderão ser considerado à um data frame.

Observação Importante

A forma mais conveniente de organizar um base de dados de modo a facilitar as análises estatísticas é considerar um data frame. No artigo **Tidy Data** publicado em 2014 por **Hadley Wickham** no **Journal of Statistical Software**. No artigo o autor aborta a dificuldade de se organizar uma base de dados e como tornar a limpeza (tratamento dos dados) tão fácil e eficaz quanto possível.

Nota: Hadley Wickham é estatístico e e pesquisador chefe da equipe do **RStudio**.

Data Frames

Figura: Hadley Wickham.

Hadley Wickham também é criador da biblioteca **ggplot2** para construção de gráficos em R. A imagem ao lado é produzida pela biblioteca **ggplot2**.

Há o livro **ggplot2: Elegant Graphics for Data Analysis** escrito por ele e complementa a documentação da biblioteca. O livro é do projeto **UseR!** e publicado pela editora Springer.

Data Frames

Para criar um data frame utilizamos a função `data.frame()` que possui como entrada vetores nomeados.

Exemplo: Corra o código que segue:

```
1: df <- data.frame(x = 1:3, y = c("a", "b", "c"))
2: str(df)
#> 'data.frame':    3 obs. of 2 variables:
#> $ x: int 1 2 3
#> $ y: Factor w/ 3 levels "a","b","c": 1 2 3
```

Nota: Observe que por padrão um vetor de strings em um fator. Esse comportamento poderá ser alterado se usamos o argumento `stringAsFactors = FALSE` na função `data.frame()`.

Data Frames

Exemplo: Execute o código abaixo:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"),
                 stringsAsFactors = FALSE)
str(df)
#> 'data.frame': 3 obs. of 2 variables:
#> $ x: int 1 2 3
#> $ y: chr "a" "b" "c"
```

Data Frames

Como um `data.frame` é uma classe **S3**, seu tipo reflete o vetor subjacente usado para construí-lo: a **lista**.

Exemplo: Utilizando a função `class()` ou explicitamente a função `is.data.frame()`.

```
typeof(df)
#> [1] "list"
class(df)
#> [1] "data.frame"
is.data.frame(df)
#> [1] TRUE
```

Nota: Veja que o tipo de um data frame é uma lista.

Data Frames

Assim como no caso de matrizes, podemos combinar data frames usando as funções `rbind()` e `cbind()`.

Exemplo: Corra o código seguinte:

```
df <- data.frame(x = 1:3, y = c("a", "b", "c"))
cbind(df, data.frame(z = 3:1))
#>   x y z
#> 1 1 a 3
#> 2 2 b 2
#> 3 3 c 1
rbind(df, data.frame(x = 10, y = "z"))
#>   x y
#> 1 1 a
#> 2 2 b
#> 3 3 c
#> 4 10 z
```

Data Frames

Importante

É um erro comum achar que concatenar dois vetores usando a função `cbind()` irá criar um data frame. Na verdade, fazer isto acarretará na criação de uma matriz. Para que um data frame seja criado, considere envolver a função `cbind()` pela função `data.frame()`.

Exemplo: Corra o código que segue:

```
good <- data.frame(a = 1:2, b = c("a", "b"),
                   stringsAsFactors = FALSE)
str(good)
#> 'data.frame': 2 obs. of 2 variables:
#> $ a: int 1 2
#> $ b: chr "a" "b"
```

Data Frames

Nota: É possível que cada coluna de um data frame seja uma lista. Essa característica é em um data frame é bastante importante.

Exemplo: Corra o código:

```
df <- data.frame(x = 1:3)
df$y <- list(1:2, 1:3, 1:4)
df
#>   x        y
#> 1 1        1, 2
#> 2 2     1, 2, 3
#> 3 3  1, 2, 3, 4
```

Data Frames

Exercício: Um professor irá criar um data frame para armazenar as notas de 5 alunos de sua turma em que cada aluno poderá ter realizado no máximo 3 avaliações. Crie um data frame com duas colunas sendo a primeira com os nomes dos alunos e a segunda com as notas das avaliações.

Solução:

```
avaliacoes <- list(c(8.12,9.81,8.20), c(8.58,9.28,
                     8.22), c(7.13,9.35,9.25),
                     c(7.13,9.35,9.25),c(9.61,7.99,
                     8.04))
alunos <- c("Wanusa", "Juliana", "Leticia", "Bianca",
            "Claryce")
notas_alunos <- data.frame(nomes = alunos,
                 avaliacoes = NA)
notas_alunos$avaliacoes <- avaliacoes
# Trataremos melhor sobre as funções da família apply
# mais adiante.
lapply(X = notas_alunos[, 2], FUN = mean)
```

Data Frames

Nota: É possível acessar cada vetor (coluna) de um data frame usando o nome do objeto seguido do símbolo $ e do nome da coluna. No exemplo logo acima, para acessar a primeira coluna faríamos `df$x` equivalente a fazer `df[,1]`. Analogamente para a segunda coluna deveríamos fazer `df$y` que equivale a fazer `df[,2]`.

Importante

Tentar atribuir diretamente uma lista como argumento da função `data.frame()` ocorrerá um erro.

Exemplo: Corra o código que segue:

```
1: data.frame(x = 1:3, y = list(1:2, 1:3, 1:4))
#> Error: arguments imply differing number of rows:
#> 2, 3, 4
```

Data Frames

O exemplo logo abaixo contorna mostra como contornar tal problema. Para isto, bastará envolver a função `list()`, passada como argumento da função `data.frame()`, com a função `I()`.

Data Frames

O exemplo logo abaixo contorna mostra como contornar tal problema. Para isto, bastará envolver a função `list()`, passada como argumento da função `data.frame()`, com a função `I()`.

Exemplo:

```
dfl <- data.frame(x = 1:3, y = I(list(1:2, 1:3, 1:4)))
# Acessando a linha dois e coluna de nome "y".
# O comando abaixo poderia ser substituido por
# dfl[2,2].
dfl[2, "y"]
#> [[1]]
#> [1] 1 2 3
```

Data Frames

Exercício: Com base no exercício anterior referente às notas dos 5 alunos, construa o data frame com os nomes e notas dos alunos atribuindo a lista de notas como argumento da função `data.frame()`.

Solução:

```
avaliacoes <- list(c(8.12,9.81,8.20), c(8.58,9.28,
                     8.22), c(7.13,9.35,9.25),
                     c(7.13,9.35,9.25),c(9.61,7.99,
                     8.04))
alunos <- c("Wanusa", "Juliana", "Leticia", "Bianca",
            "Claryce")
notas_alunos <- data.frame(nomes = alunos,
                 avaliacoes = I(avaliacoes))
notas_alunos$avaliacoes <- avaliacoes
# Trataremos melhor sobre as funções da família apply
# mais adiante.
lapply(X = notas_alunos[, 2], FUN = mean)
```

Data Frames

De forma análoga, é possível ter uma coluna de um data frame sendo uma matriz, desde que o número de linhas da matriz seja o mesmo da dimensão linha do data frame.

Exemplo: Corra o código abaixo:

```
dfm <- data.frame(x = 1:3,
                  y = I(matrix(1:9, nrow = 3)))
dfm[2, "y"]
#>      [,1] [,2] [,3]
#> [1,]    2    5    8
```

Data Frames

Exercício: Refaça o exercício das notas dos 5 alunos considerando que a segunda coluna do data frame criado para armazenar os nomes e avaliações é uma matriz de dimensão 5×3.

Solução:

```
avaliacoes <- list(c(8.12,9.81,8.20), c(8.58,9.28,
                8.22), c(7.13,9.35,9.25), c(7.13,
                9.35,9.25), c(9.61,7.99,8.04))

avaliacoes <- matrix(data = unlist(avaliacoes),
                nrow = 5, ncol = 3, byrow = TRUE)


alunos <- c("Wanusa", "Juliana", "Leticia", "Bianca",
             "Claryce")

notas_alunos <- data.frame(nomes = alunos,
                  avaliacoes = I(avaliacoes))
```

Data Frames

Exercício: Realize uma estatística descritiva das notas dos alunos utilizando a função `summary()`.

Data Frames

Exercício: Realize uma estatística descritiva das notas dos alunos utilizando a função `summary()`.

Solução:

```
1: apply(X = notas_alunos$avaliacoes, MARGIN = 1,
         FUN = summary)
```

Exercício: Realize uma estatística descritiva das notas dos alunos utilizando a função `summary()`.

Solução:

```
1: apply(X = notas_alunos$avaliacoes, MARGIN = 1,
         FUN = summary)

         [,1]     [,2]     [,3]     [,4]     [,5]
Min.    8.120 8.220000 7.130000 7.130000 7.990000
1st Qu. 8.160 8.400000 8.190000 8.190000 8.015000
Median  8.200 8.580000 9.250000 9.250000 8.040000
Mean    8.710 8.693333 8.576667 8.576667 8.546667
3rd Qu. 9.005 8.930000 9.300000 9.300000 8.825000
Max.    9.810 9.280000 9.350000 9.350000 9.610000
```

Acessando subconjunto - Vetor

Os operadores de subconjunto da linguagem `R` são poderosos e rápidos. O domínio de tais operadores permite que o programador expresse sucintamente operações complexas de uma maneira que outras linguagem podem não fazer.

Considere o vetor atômico `x <- c(2.1, 4.2, 3.3, 5.4)`. Os exemplos seguintes deverão ser replicados e considerarão esse vetor.

Acessando subconjunto - Vetor

Exemplo: Inteiros positivos retornam os elementos nas respectivas posições e números reais serão truncados.

```
1: x[c(3, 1)]
#> [1] 3.3 2.1
2: x[order(x)] # Faça ?order para detalhes.
#> [1] 2.1 3.3 4.2 5.4
3:
# Duplicated indices yield duplicated values
4: x[c(1, 1)]
#> [1] 2.1 2.1
# Real numbers are silently truncated to integers
5: x[c(2.1, 2.9)]
#> [1] 4.2 4.2
```

Acessando subconjunto - Vetor

Exemplo: Números inteiros negativos omite os elementos omite a respectiva posição e valores reais negativos serão truncados.

```
1: x[-c(3, 1)]
#> [1] 4.2 5.4
2:
3: x[c(-1, 2)]
#> Error: only 0's may be mixed with negative subscripts
```

Acessando subconjunto - Vetor

Exemplo: Vetores lógicos selecionam elementos onde o valor lógico corresponde à verdade. Assim, uma expressão lógica também poderá ser utilizada para selecionar os elementos de um vetor atômico.

```
1: x[c(TRUE, TRUE, FALSE, FALSE)]
#> [1] 2.1 4.2
2: x[x > 3]
#> [1] 4.2 3.3 5.4
```

Acessando subconjunto - Vetor

Importante: Se o vetor lógico tiver um comprimento menor que o vetor ao qual queremos extrair um subconjunto, o vetor lógico será reciclado para ter o mesmo comprimento.

```
x[c(TRUE, FALSE)]
#> [1] 2.1 3.3
# Equivalent to
x[c(TRUE, FALSE, TRUE, FALSE)]
#> [1] 2.1 3.3
```

Acessando subconjunto - Vetor

Exemplo: Um valor faltante no vetor de índices irá produzir uma informação faltante na respectiva posição do vetor saída.

```
x[c(TRUE, TRUE, NA, FALSE)]
#> [1] 2.1 4.2 NA
```

Acessando subconjunto - Vetor

Exemplo: **Nada** retorna o vetor original. Isto não é útil para vetores mas é bastante útil para arrays e data frames.

```
1: x[]
#> [1] 2.1 4.2 3.3 5.4
```

Exemplo: **Zero** retornará um vetor de comprimento zero.

```
1: a <- x[0]
2: a[1] = 2; a[2] = 7
3: a
#> [1] 2 7
```

Acessando subconjunto - Vetor

Exemplo: Vetor de caracteres retornará os elementos de nomes correspondentes.

```
1: y <- setNames(x, letters[1:4])
#>   a   b   c   d
#> 2.1 4.2 3.3 5.4
2: y[c("d", "c", "a")]
#>   d   c   a
#> 5.4 3.3 2.1
```

Nota: Faça `?letters` e leia o manual.

Exercício: De que outra forma poderíamos criar e nomear o objeto `y` sem utilizar a função `setNames()`?

Acessando subconjunto - Vetor

Exemplo: A não correspondência exata de um nome não provocará erro. Nesse caso, a linguagem `R` retornará `NA`.

```
z <- c(abc = 1, def = 2)
z[c("a", "d")]
#> <NA> <NA>
#>   NA   NA
```

Acessando subconjunto - Matrizes e Array

Exemplo: Corra o código logo abaixo:

```
1: a <- matrix(1:9, nrow = 3)
2: colnames(a) <- c("A", "B", "C"); a[1:2, ]
#>      A B C
#> [1,] 1 4 7
#> [2,] 2 5 8
3: a[c(T, F, T), c("B", "A")]
#>      B A
#> [1,] 4 1
#> [2,] 6 3
4: a[0, -2]
#> A C
```

Acessando subconjunto - Matrizes e Array

Exemplo: Também é possível passar uma matriz como vetor de índices. Procure nas documentações da linguagem entender o funcionamento das funções `outer()` e `paste()`.

```
1: vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
#>       [,1]  [,2]  [,3]  [,4]  [,5]
#> [1,] "1,1" "1,2" "1,3" "1,4" "1,5"
#> [2,] "2,1" "2,2" "2,3" "2,4" "2,5"
#> [3,] "3,1" "3,2" "3,3" "3,4" "3,5"
#> [4,] "4,1" "4,2" "4,3" "4,4" "4,5"
#> [5,] "5,1" "5,2" "5,3" "5,4" "5,5"
2:
3: vals[c(4, 15)]
#> [1] "4,1" "5,3"
```

Acessando subconjunto - Matrizes e Array

```
vals <- outer(1:5, 1:5, FUN = "paste", sep = ",")
select <- matrix(ncol = 2, byrow = TRUE,
                 c(1, 1, 3, 1, 2, 4))
vals[select]
#> [1] "1,1" "3,1" "2,4"
```

Notas:

1. O argumento `byrow = TRUE` faz com que o preenchimento da matriz seja realizado por linha. Por padrão `byrow = FALSE`.
2. Obviamente a matriz com `select` deverá possuir duas colunas, uma vez que a matriz possui uma estrutura bidimensional. Os elementos na primeira coluna refere-se ao elemento linha e o elemento da coluna refere-se ao elemento coluna da matriz ao qual será obtido o subconjunto.

Acessando subconjunto - Data Frame

Exemplo: Data frames possuem as mesmas características de matrizes para acessar seus subconjuntos. Corra o código que segue:

```
df <- data.frame(x = 1:3, y = 3:1, z = letters[1:3])
df[df$x == 2, ]
#>   x y z
#> 2 2 2 b
df[c(1, 3), ]
#>   x y z
#> 1 1 3 a
#> 3 3 1 c
# There are two ways to select columns from a data
# frame
```

Acessando subconjunto - Data Frame

```
5: # Like a list:
6: df[c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
7: # Like a matrix
8: df[, c("x", "z")]
#>   x z
#> 1 1 a
#> 2 2 b
#> 3 3 c
```

Nota Importante: Passar um vetor como índices de um objeto da classe data frame fará com que a linguagem R selecione as colunas correspondentes ao índices do vetor.

Acessando subconjunto - Data Frame

Exercício: Inicialmente, crie a lista abaixo:

Acessando subconjunto - Data Frame

Exercício: Inicialmente, crie a lista abaixo:

```
notas <- list(aluno_1 = c(7.1, 3.2, NA),
              aluno_2 = c(2.7, 8.8, 10.0),
              aluno_3 = c(0.0, NA, NA),
              aluno_4 = c(7.7, 8.4, 6.3),
              aluno_5 = c(3.6, 6.6, 8.1),
              aluno_6 = c(NA, NA, NA),
              aluno_7 = c(7.4, 7.1, 7.3),
              aluno_8 = c(10.0, NA, 7.0),
              aluno_9 = c(1.6, 3.2, 5.3),
              aluno_10 = c(8.8, 9.2, 8.0))
```

Acessando subconjunto - Data Frame

a) Crie o vetor `status` contendo o status dos dez alunos. Considere: **A** (aprovado), **REP** (reprovado), **F** (final). **Dica**: Considere as regras de nossa instituição.

Acessando subconjunto - Data Frame

a) Crie o vetor `status` contendo o status dos dez alunos. Considere: **A** (aprovado), **REP** (reprovado), **F** (final). **Dica**: Considere as regras de nossa instituição.

b) Crie o vetor `alunos` com os nomes dos alunos. Considere os nome fictício **Aluno_1** para o primeiro aluno e de forma análoga para os demais alunos. Crie esse vetor utilizando a função `paste()`.

Acessando subconjunto - Data Frame

a) Crie o vetor `status` contendo o status dos dez alunos. Considere: **A** (aprovado), **REP** (reprovado), **F** (final). **Dica**: Considere as regras de nossa instituição.

b) Crie o vetor `alunos` com os nomes dos alunos. Considere os nome fictício **Aluno_1** para o primeiro aluno e de forma análoga para os demais alunos. Crie esse vetor utilizando a função `paste()`.

c) Construa um data frame de nome **histórico** com as variáveis **nomes**, **notas** e **status**.

Acessando subconjunto - Data Frame

d) Com base no data frame de nome **historico**, construa um outro data frame com o nome **aprovados** com os alunos aprovados. De forma análoga para os demais status.

Acessando subconjunto - Data Frame

d) Com base no data frame de nome **historico**, construa um outro data frame com o nome **aprovados** com os alunos aprovados. De forma análoga para os demais status.

e) Suponha que o professor está interessado em saber quais alunos foram ou tem alguma chance de assumir o status de aprovado. Construa o data frame de nome **bons_alunos** com estes alunos.

Acessando subconjunto - Data Frame

d) Com base no data frame de nome **historico**, construa um outro data frame com o nome **aprovados** com os alunos aprovados. De forma análoga para os demais status.

e) Suponha que o professor está interessado em saber quais alunos foram ou tem alguma chance de assumir o status de aprovado. Construa o data frame de nome **bons_alunos** com estes alunos.

f) Modifique os nomes das linhas do data frame **historico** colocando **id_1** na primeira linha e respectivamente no mesmo padrão para as demais linhas.

Acessando subconjunto - Data Frame

g) Obtenha por meio do data frame de nome **historico** um novo data frame (**historico_na**) com os alunos que deixaram ao menos uma prova para repor.

Acessando subconjunto - Data Frame

g) Obtenha por meio do data frame de nome **historico** um novo data frame (**historico_na**) com os alunos que deixaram ao menos uma prova para repor.

h) Apenas para os alunos que fizeram as três avaliações, obtenha uma média aritmética das avaliações. Acrescente a variável de nome **media** no data frame **historico**.

Acessando subconjunto - Data Frame

Exercício: Utilizando o conjunto de dados de nome **state.x77** responda os itens abaixo:

a) Construa o data frame **dados** a partir de **state.x77**.

Acessando subconjunto - Data Frame

Exercício: Utilizando o conjunto de dados de nome **state.x77** responda os itens abaixo:

a) Construa o data frame **dados** a partir de **state.x77**.

b) Obtenha um data frame de nome **dados_1** com as observações de **dados** que possua população maior que 4246, isto é, com os estados estadunidenses que possua uma população maior que 4246 (quatro milhões duzentos e quarenta e seis mil).

Acessando subconjunto - Data Frame

Exercício: Utilizando o conjunto de dados de nome **state.x77** responda os itens abaixo:

a) Construa o data frame **dados** a partir de **state.x77**.

b) Obtenha um data frame de nome **dados_1** com as observações de **dados** que possua população maior que 4246, isto é, com os estados estadunidenses que possua uma população maior que 4246 (quatro milhões duzentos e quarenta e seis mil).

c) Obtenha o data frame **dados_2** com as observações população maior que 4246 e menores que 8 milhões, isto é, menor que 8000.

Acessando subconjunto - Data Frame

Exercício: Utilizando o conjunto de dados de nome **state.x77** responda os itens abaixo:

a) Construa o data frame **dados** a partir de **state.x77**.

b) Obtenha um data frame de nome **dados_1** com as observações de **dados** que possua população maior que 4246, isto é, com os estados estadunidenses que possua uma população maior que 4246 (quatro milhões duzentos e quarenta e seis mil).

c) Obtenha o data frame **dados_2** com as observações população maior que 4246 e menores que 8 milhões, isto é, menor que 8000.

d) Obtenha o vetor **estados_c** com os nomes dos estados que obedecem os critérios do item c.

Acessando subconjunto - Data Frame

e) Construa o data frame **dados_3** com os estados estadunidenses com população maior que 1.5 vezes a média dos 50 estados considerados. Obtenha o vetor com o nome dos estados que obedecem essa regra.

Acessando subconjunto - Data Frame

e) Construa o data frame **dados_3** com os estados estadunidenses com população maior que 1.5 vezes a média dos 50 estados considerados. Obtenha o vetor com o nome dos estados que obedecem essa regra.

f) Construa o data frame **dados_4** com os estados estadunidenses com população maior que duas vezes a mediana dos 50 estados e que tenha uma população com expectativa de vida maior que 71.84 anos.

Acessando subconjunto - Data Frame

e) Construa o data frame **dados_3** com os estados estadunidenses com população maior que 1.5 vezes a média dos 50 estados considerados. Obtenha o vetor com o nome dos estados que obedecem essa regra.

f) Construa o data frame **dados_4** com os estados estadunidenses com população maior que duas vezes a mediana dos 50 estados e que tenha uma população com expectativa de vida maior que 71.84 anos.

g) Obtenha o data frame **dados_5** com os estados estadunidenses com renda maior que a média nacional, expectativa de vida maior que 72 anos.

Acessando subconjunto - Data Frame

e) Construa o data frame **dados_3** com os estados estadunidenses com população maior que 1.5 vezes a média dos 50 estados considerados. Obtenha o vetor com o nome dos estados que obedecem essa regra.

f) Construa o data frame **dados_4** com os estados estadunidenses com população maior que duas vezes a mediana dos 50 estados e que tenha uma população com expectativa de vida maior que 71.84 anos.

g) Obtenha o data frame **dados_5** com os estados estadunidenses com renda maior que a média nacional, expectativa de vida maior que 72 anos.

h) Adicione ao data frame **dados** duas linhas com a média de todas as variáveis e variâncias, respectivamente.

Acessando subconjunto - Lista

Vamos fazer uso de uma metáfora que pode nos ajudar. Em listas é muito comum o uso dos operadores `$` e `[[`. Entender da seguinte forma poderá ajudar:

Acessando subconjunto - Lista

Vamos fazer uso de uma metáfora que pode nos ajudar. Em listas é muito comum o uso dos operadores `$` e `[[`. Entender da seguinte forma poderá ajudar:

1. `[[`: é usado para extrair itens únicos.

Acessando subconjunto - Lista

Vamos fazer uso de uma metáfora que pode nos ajudar. Em listas é muito comum o uso dos operadores `$` e `[[`. Entender da seguinte forma poderá ajudar:

1. `[[`: é usado para extrair itens únicos.
2. `$`: é uma sintaxe útil à sintaxe acima quando os elementos de uma lista estão nomeados.

Acessando subconjunto - Lista

Metáfora

Se a lista `x` é um trem que transporta objetos, então `x[[5]]` é o objeto no carro 5; e `x[4:6]` é um trem de carros 4 à 6.

Acessando subconjunto - Lista

Metáfora

Se a lista `x` é um trem que transporta objetos, então `x[[5]]` é o objeto no carro 5; e `x[4:6]` é um trem de carros 4 à 6.

```
x <- list(1:3, "a", 4:6)
```

Acessando subconjunto - Lista

Continuando com a metáfora, ao extrair um subconjunto de uma lista temos duas opções. Você poderá criar um trem menor ou extrair o conteúdo de um dos vagões. Essa é de forma metafórica a diferença entre os operadores `$` e `[[`, respectivamente.

Acessando subconjunto - Lista

Continuando com a metáfora, ao extrair um subconjunto de uma lista temos duas opções. Você poderá criar um trem menor ou extrair o conteúdo de um dos vagões. Essa é de forma metafórica a diferença entre os operadores `$` e `[[`, respectivamente. A imagem abaixo mostra isso de uma forma mais lúdica.

Acessando subconjunto - Lista

Entendida a metáfora, as figuras abaixo são autoexplicativas:

Acessando subconjunto - Lista

Entendida a metáfora, as figuras abaixo são autoexplicativas:

Acessando subconjunto - Lista

Exemplo: O operador `[[` é similar ao operador `[`, exceto pelo fato que `[[` retornará apenas um único valor. O operador `$` é uma abreviação do operador `[[`.

```
a <- list(a = 1, b = 2)
a[[1]]
#> [1] 1
a[["a"]]
#> [1] 1
```

Acessando subconjunto - Lista

```
# If you do supply a vector it indexes recursively
b <- list(a = list(b = list(c = list(d = 1))))
b[[c("a", "b", "c", "d")]]
#> [1] 1
# Same as
b[["a"]][["b"]][["c"]][["d"]]
#> [1] 1
# Same as
a$b$c$d
#> [1] 1
```

Programação em R
Parte II

Autor: Prof. Dr. Pedro Rafael Diniz Marinho

Universidade Federal da Paraíba
Departamento de Estatística da UFPB

Simplificando vs Preservando

Ao acessarmos subconjuntos de um objeto de tipo específico o comportamento a estrutura do objeto gerado poderá ser preservada ou simplificada.

Tabela: Simplificando ou preservando a estrutura de dados de um objeto.

Estrutura	Simplificando	Preservando
Vetor	`x[[1]]`	`x[1]`
Lista	`x[[1]]`	`x[1]`
Fator	`x[1:4, drop = T]`	`x[1:4]`
Array	`x[1, ]` ou `x[ ,1]`	`x[1, , drop = F]` ou `x[, 1, drop = F]`
Data Frame	`x[, 1]` ou `x[[1]]`	`x[, 1, drop = F]` ou `x[1]`

Simplificando vs Preservando

Exemplo (vetor atômico): A simplificação remove os nomes.

```
1: x <- c(a = 1, b = 2)
2: x[1]
#> a
#> 1
3: x[[1]]
#> [1] 1
```

Simplificando vs Preservando

Exemplo (lista): A simplificação retornará o objeto dentro da lista e não uma lista de um elemento.

```
y <- list(a = 1, b = 2)
str(y[1])
#> List of 1
#> $ a: num 1
str(y[[1]])
#> num 1
```

Simplificando vs Preservando

Exemplo (fator): A simplificação descarta os níveis não utilizados.

```
1: z <- factor(c("a", "b"))
2: z[1]
#> [1] a
#> Levels: a b
3: z[1, drop = TRUE]
#> [1] a
#> Levels: a
```

Simplificando vs Preservando

Exemplo (matriz ou array): Se os índices de ao menos uma das dimensões tem comprimento 1, cairá a dimensão.

```
1: a <- matrix(1:4, nrow = 2)
2: a[1, , drop = FALSE]
#>      [,1] [,2]
#> [1,]    1    3
3: a[1, ]
#> [1] 1 3
```

Simplificando vs Preservando

Exemplo (data frame): Se uma das dimensões é informada e a outra é deixada em branco, a estrutura será preservada.

```
1: df <- data.frame(a = 1:2, b = 1:2)
2: str(df[1])
#> 'data.frame':    2 obs. of 1 variable:
#> $ a: int 1 2
3: str(df[[1]])
#> int [1:2] 1 2
4: str(df[, "a", drop = FALSE])
#> 'data.frame': 2 obs. of 1 variable:
#> $ a: int 1 2
5: str(df[, "a"])
#> int [1:2] 1 2
```

Simplificando vs Preservando

Operadores `$` e `[[`

Fazer `x$y` é equivalente a fazer `x[["y", exact = FALSE]]`, porém há situações em que possamos preferir o uso do operador `[[` ao invés do operador `$`. O exemplo abaixo tenta mostrar isto.

Exemplo: Corra o código que segue:

```
var <- "cyl"
mtcars$var
#> NULL
# Instead use [[
mtcars[[var]]
#> [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4
#> 8 8 8 8 4 4 4 8 6 8 4
```

Simplificando vs Preservando

```
1: x <- list(abc = 1)
2: x$a
#> [1] 1
3: x[["a"]]
#> NULL
```

Nota: Perceba que o operador `$` permite que possamos completar parcialmente o nome de um elemento. Isto só poderá ser permitido com o operador `[[` caso viermos à passarmos como argumento `exact = FALSE` que por padrão é igual à `TRUE`.

Atribuição

Todos os operadores de subconjuntos estudados anteriormente podem ser combinados com atribuição para modificar valores selecionados do vetor de entrada. O exemplo que segue tenta mostrar um bom resumo sobre atribuição. Tente correr o exemplo e compreender os detalhes.

Exemplo: Corra o código que segue:

```
x <- 1:5
x[c(1, 2)] <- 2:3
x
#> [1] 2 3 3 4 5
```

Atribuição

```
# The length of the LHS needs to match the RHS
x[-1] <- 4:1
x
#> [1] 2 4 3 2 1

# Note that there's no checking for duplicate indices
x[c(1, 1)] <- 2:3
x
#> [1] 3 4 3 2 1
```

Atribuição

```
# You can't combine integer indices with NA
x[c(1, NA)] <- c(1, 2)

#> Error: NAs are not allowed in subscripted assignments
# But you can combine logical indices with NA
# (where they're treated as false).
x[c(T, F, NA)] <- 1
x
#> [1] 1 4 3 1 1
```

Atribuição

```
# This is mostly useful when conditionally modifying
# vectors
df <- data.frame(a = c(1, 10, NA))
df$a[df$a < 5] <- 0
df$a
#> [1] 0 10 NA
```

Nota: A função `lapply()` poderá ser bastante útil para retornar uma lista de mesmo tamanho do seu primeiro argumento (vetor, matriz, data frame, etc), em que cada elemento da lista retornada é o resultado da aplicação de uma função passada como segundo argumento à função `lapply()` aos elementos do objeto passado como primeiro argumento.

Atribuição

Exercício: Corra o código que segue. Discutam entre si o funcionamento da função `lapply()` e observe outros detalhes.

```
mtcars[] <- lapply(X = mtcars, FUN = as.integer)
mtcars
mtcars <- lapply(X = mtcars, FUN = as.integer)
mtcars
```

Correspondência

Suponha que tenhamos um vetor de notas inteiras e um uma descrição da propriedade da nota, como mostra o exemplo que segue:

Exemplo: Corra o código abaixo e preste atenção no que está sendo feito. Isto poderá ser útil.

Correspondência

Suponha que tenhamos um vetor de notas inteiras e um uma descrição da propriedade da nota, como mostra o exemplo que segue:

Exemplo: Corra o código abaixo e preste atenção no que está sendo feito. Isto poderá ser útil.

```
grades <- c(1, 2, 2, 3, 1)
info <- data.frame(grade = 3:1, desc = c("Excellent",
                   "Good", "Poor"), fail = c(F, F, T))
# Using match
id <- match(grades, info$grade)
info[id, ]
```

Correspondência

```
# Using rownames
rownames(info) <- info$grade
info[as.character(grades), ]
```

Exercício: Estude a documentação das funções `match()` e `%in%` (faça `?"%in%"`).

Interação com o usurário

A linguagem `R` possui diversas funções que têm que permitem o programa interagir com o utilizado como é o caso da função `print()` que poderá ser utilizada para escrever o conteúdo de qualquer objeto.

Exemplo: Corra o código que segue:

```
1: # O código na linha 3 não irá imprimir nada
2: # muito embora as sequências serão executadas.
3: for (i in 1:i) 1:i
4: # Imprimindo no prompt de comando.
5: for (i in 1:i) print(i)
```

Interação com o usurário

Uma outra função muito eficiente para escrever objetos e que possibilita receber um número qualquer de argumentos transformando seus argumentos em strings concatenadas é a função `cat()`.

Exemplo: Execute o código logo abaixo:

```
nota <- 7
cat("Joãozinho: Professor,\nqual foi minha nota?
Professor: Sua nota foi", nota, "Joãozinho\t...",
     sep = " ")
```

Exercício: Leia a documentação as funções `print()` e `cat()`.

Interação com o usurário

Também é possível que o usuário de um código R insira os dados durante a execução do código. Para isto é normalmente utilizado a função `scan()`.

```
1: x <- scan(n = 3)
#> 1: 1 2 3
#> Read 3 items
2: y <- scan(what = character())
3: "Como estatístico" "devo"
4: "gostar de " "programar ..."
5: cat(y)
#> Como estatístico devo gostar de  programar ...
```

Instruções Condicionais

É muito comum em qualquer código fazermos uso de instruções condicionais que permitem o programador explicitar diferentes alternativas que podem vir a serem executadas dependendo de alguma condição testada na altura da execução do programa pelo interpretador da linguagem.

A instrução `if()` permite que uma condição seja avaliada e caso sera verdadeira, o bloco correspondente à instrução é executado. Em caso que a instrução seja falsa, é possível fazer com que outro bloco de instruções seja executado.

Exemplo: Corra o código abaixo:

Instruções Condicionais

```
x <- 7
if(x > 0)
{
    cat(‘x é positivo.\n’)
    y <- z / x
} else {
            cat(‘x não é positivo! \n’)
            y <- z
}
```

Nota: É importante prestar atenção à indentação do código. Dessa forma, conseguiremos ressaltar e entender melhor a estrutura do código de forma a aumentar a legibilidade do código.

Instruções Condicionais

O código acima não seria tão legível se fosse apresentado na forma:

```
1: x <- 7
2: if(x > 0)
3: {
4: cat(‘x é positivo.\n’)
5: y <- z / x
6: } else {
7: cat(‘x não é positivo! \n’)
8: y <- z
9: }
```

Observação: Note que a cláusula `else` é opcional.

Instruções Condicionais

Também é possível aninhar diversas instruções `if()`. Observe o exemplo abaixo.

Exemplo:

```
1: idade <- 30
2: if (idade < 18) {
3:    grupo <- 1
4: } else if (idade < 35) {
5:    grupo <- 2
6: } else if (idade < 65) {
7:    grupo <- 3
8: } else {
9:    grupo <- 4
10:}
11:grupo
```

Instruções Condicionais

Notas:

1. Chamamos bloco de instruções o conjunto de código entre $\{\}$;
2. Muito embora os blocos de instruções no exemplo anterior apresente apenas uma instrução no interior do bloco, estes poderiam conter diversas outras instruções.

Instruções Condicionais

Observação: Caso a instrução `if` possua ser escrita em apenas uma única linha, as chaves de bloco poderão ser omitidas.

Exercício:

```
1: nota <- 7
2: if(nota < 7) cat ("Aluno: =(") else cat("Aluno: =)")
```

Nota: Como foi omitido o bloco na instrução `if`, note que a instrução `else` também teve que ser iniciada na linha 2 do código acima.

Instruções Condicionais

Uma função `ifelse()` também poderá ser utilizada para se tra balhar com o controle de fluxo da linguagem `R`. Esta função permite apenas três argumentos em que o primeiro é uma condição que será testada. Em caso verdadeiro, o resultado da expressão do segundo argumento será retornado. Caso contrário, será retornado a expressão passada como terceiro argumento à função.

Exemplo: Corra o código abaixo:

```
1: nota <- 7
2: ifelse(nota >= 7, "=)", "=(")
#> [1] "=)"
```

Instruções Condicionais

Exemplo: Observe um outro exemplo

```
1: set.seed(2)
2: x <- rnorm(n = 5,  mean = 0, sd = 1)
3: sig <- ifelse(x < 0, "-", "+")
4: sig
#> "-" "+" "+" "-" "-"
```

Nota: A linha 1 do código acima fixa um valor de semente para o gerador de números pseudo-aleatório que é utilizado na geração de números aleatórios de uma determinada distribuição de probabilidade. A linha 2 gera 5 números pseudo-aleatórios com distribuição normal centrada no zero e variância 1. Falaremos mais a frente com os devidos detalhes sobre geração de números pseudo-aleatórios.

Instruções Condicionais

Uma outra instrução condicional em que poderá ser utilizada para escolher uma de várias alternativas possíveis é a instrução `switch()`. Tal função consiste em uma série de argumentos que a depender do primeiro argumento algum dos outros argumentos será retornado.

Nota: O primeiro argumento poderá ser um número ou uma string ou qualquer expressão que resulte um número ou caractere.

```
set.seed(0) # Fixando semente do gerador.
expressao <- 1
vetor_normal <- rnorm(10) # Faça ?rnorm para detalhes.
switch(EXPR = expressao, round(mean(vetor_normal),1),
round(median(vetor_normal)))
#> 0.4
```

Instruções Condicionais

Exemplo: Replique o exemplo anterior fazendo `expressao = 3`. Nesse caso, nada será retornado, isto é, `NULL` será o tipo de retorno.

Exemplo: Considere o exemplo que segue em que o argumento `EXPR` da instrução `switch()` é uma string.

```
semaforo <- "verde"
switch(EXPR = semaforo, verde = "siga",
       amarelo = "atenção",
       vermelho = "pare")
```

Instruções Condicionais

Importante: As instruções poderão ser maiores, isto é, conter mais de uma linha. Dessa forma, o programador deverá utilizar um bloco de instruções ({}).

Exercício: Escreva um programa que calcule o imposto pago por mulheres e por homens, sabendo que as mulheres pagam 10% e que os homens pagam 5% a mais do que as mulheres.

Solução:

```
# Lendo entrada do teclado.
meu_salario <- function(){
salario <- as.numeric(readline(prompt =
                                "Entre com um salário: "))
sexo <- tolower(readline(prompt =
                            "Informe o sexo (m ou f): "))
switch(sexo,
       "m" = {
                imposto = 0.15;
                cat("O salário a ser pago é ",
                     salario - salario*imposto)
             },
```

```
13:         "f" = {
14:                     imposto = 0.1;
15:                     cat("O salário a ser pago é ",
16:                         salario - salario*imposto)
17:                 }
18:         )
19:}
```

Exercício: Descreva para que servem as funções `tolower()`, `toupper()` e `readline()`. Apresente exemplos do uso de cada uma delas.

Loop

Loop

A linguagem R possui diversas instruções de laço que nos permite repetir blocos de instruções um número predefinido de vezes ou até que uma condição não seja mais verdadeira.

Uma das instruções bastante utilizada para repetição iterativa de um bloco de instruções é a instrução `while()`. A forma geral da sintaxe da instrução `while()` na linguagem R é:

Sintaxe:

```
while (condição boolena)
{
  bloco de insturções a se repetir.
}
```

Loop

Exemplo: Corra o código abaixo:

```
1: i <- 1
2: while(i<=7)
3: {
4:    cat("i",i, " = ", i, "\n", sep="")
5:    i <- i + 1
6: }
#> i1 =  1
#> i2 =  2
#> i3 =  3
#> i4 =  4
#> i5 =  5
#> i6 =  6
#> i7 =  7
```

Loop

Exercício: Escreva um programa que coloque na tela a tabuada de 7 utilizando a instrução `while()`.

Loop

Exercício: Escreva um programa que coloque na tela a tabuada de 7 utilizando a instrução `while()`.

```
i <- 1
while(i<=10)
{
    cat("7 x ", i, " = ", 7 * i, "\n")
    i <- i + 1 # incrementando i.
}
#> 7 x 1 = 7
#> 7 x 2 = 14
#> 7 x 3 = 21
#> 7 x 4 = 28
#> ...
```

Loop

Importante

Cuidado com loops infinitos. Uma instrução de loop deverá sempre alcançar um fim. Dessa forma, garanta que a condição do seu laço em algum momento se torne falsa. Esse erro é muito comum quando esquecemos de incrementar ou decrementar a variável de controle do laço.

Exemplo: Exemplo de loop infinito.

```
1: i <- 1
2: while(i<=10)
3: {
4:    cat("7 x ", i, " = ", 7 * i, "\n")
5: }
```

Loop

Exemplo: Imprimindo o conteúdo de `u` enquanto a expressão `u < 0.5` permanecer verdadeira.

```
u <- runif(n = 1, min = 0, max = 1)
while (u < 0.5)
{
  cat("u = ", u, "\n", sep="")
  u <- runif(n = 1, min = 0, max = 1)
}
```

Loop

Uma outra instrução de repetição bastante útil é a `repeat`. Tal instrução é executada indefinidamente até que alguma condição force sua interrupção.

Exemplo: Corra o código que segue:

```
1: texto <- c()
2: repeat {
3:   fr <- readline(prompt = "Introduza uma frase?
4:                    (frase vazia termina) ")
5:   if (fr == ‘’) break else texto <- c(texto,fr)
6:}
```

Importante

A instrução `break` faz com que uma instrução de repetição seja finalizada. Dessa forma, no exemplo anterior, temos que ao ser verdade a condição da instrução condicional `if()` a instrução `break` será avaliada, forçando assim o término da instrução `repeat`.

Nota: A instrução `break` poderá ser utilizada em qualquer instrução apresentadas (`while()` e `repeat`) bem como na instrução `for()` que será apresentada mais a frente.

Loop

Existe também uma outra instrução bastante útil quando utilizada em instruções de repetição que é a instrução `next`. Quando a instrução `next` é avaliada, todas as instruções que seguem são ignoradas e a instrução de repetição irá para a próxima iteração.

Exemplo: Corra o código:

```
vetor <- c()
repeat {
   nro <- as.numeric(readline(prompt =
           "Introduza um nro positivo ?
            (zero termina) "))
   if (nro < 0) next
   if (nro == 0) break
   vetor <- c(vetor,nro)
}
```

Loop

Assim como em quase todas as linguagens de programação, não poderia faltar a instrução `for()` em que uma variável de controle irá percorrer um conjunto.

Sintaxe:

```
for (var in conjunto)
{
   meu bloco de instruções
}
```

Loop

Exercício: Utilizando a instrução de repetição `for` construa um pequeno programa que com base em um vetor de valores no intervalo $[0, 1]$, some apenas os valores maiores que 0.7. **Dica**: (Para economizar tempo, faça `vetor <- runif(n = 10, min = 0, max = 1)` para gerar o vetor é observada de uma v.a. X_i, tal que $X_i \sim \mathcal{U}(0, 1)$, $i = 1, \ldots, 10$.)

Uma possível solução:

```
set.seed(0) # Fixando a semente do gerador.
vetor <- runif(n = 10, min = 0, max = 1)
soma <- 0

for (indice in vetor)
{
    if (indice > 0.7)
    valor <- indice
    else valor <- 0
    soma <- soma + valor
}

soma
#>  [1] 3.64797
```

Loop

Muito Importante

Há diversas situações em R em que podemos evitar o uso de instruções de repetições. Por exemplo, a situação do exercício acima é uma delas. Evitar estas situações ajudará muito a aumentar a eficiência do código.

Exercício: Refaça o exercício anterior sem utilizar nenhuma instrução de loop.

Loop

Muito Importante

Há diversas situações em R em que podemos evitar o uso de instruções de repetições. Por exemplo, a situação do exercício acima é uma delas. Evitar estas situações ajudará muito a aumentar a eficiência do código.

Exercício: Refaça o exercício anterior sem utilizar nenhuma instrução de loop.

Solução:

```
set.seed(0)
vetor <- runif(n = 10, min = 0, max = 1)
soma <- sum(vetor[vetor > 0.7])
soma
#> [1] 3.64797
```

Loop

Exercício: Dê uma consultada na documentação da função `Sys.time()`. Utilize esta função para obter o tempo de execução, em segundos, das funções implementadas nos dois exercícios anteriores.

Nota: Muitas vezes é suficiente utilizar a função `Sys.time()` para se ter uma ideia do tempo de execução de um programa ou de um trecho. Porém, existe funções específicas para mensurar o tempo de forma mais precisa. Estas funções propões realizar **benchmark** mais precisos.

O que é benchmark?

Resposta: Benchmark é o processo de executar um programa ou conjunto de operações a fim de avaliar o desempenho relativo de uma função ou objeto. O desempenho é testado por meio de um conjunto de testes padrões e ensaios sobre a função ou objeto.

Uma das formas eficientes de fazer benchmark na linguagem `R` é utilizar a função `microbenchmark()` do pacote **microbenchmark** que poderá ser instalado fazendo

```
install.packages("microbenchmark")
```

no prompt de comando da linguagem.

Loop

A função `microbenchmark()` fornece o desempenho com nanosegundos, em que $1\,\mathrm{s} = 1 \times 10^9\,\mathrm{ns}$ (nanosegundos).

Nota: É possível alterar a unidade de mensuração de tempo modificando o argumento `unit` da função `microbenchmark()`. Alguns argumentos possíveis são:

- ns: nanosegundos;
- ms: milissegundo
- s: segundos;
- Outros argumentos como unidade de frequência são também suportados, como por exemplo: hz, khz, mhz.

Loop

Exercício: Leia a documentação da função `microbenchmark()`. Reproduza os dois exemplos anteriores e realize o benchmarks nos dois casos.

Exercício: Leia a documentação da função `microbenchmark()`. Reproduza os dois exemplos anteriores e realize o benchmarks nos dois casos.

Solução:

```
library(microbenchmark)

# Exemplo 1:

set.seed(0)
vetor <- runif(n = 10, min = 0, max = 1)
soma <- 0
```

Loop

```
microbenchmark(
for (indice in vetor)
{
    if (indice > 0.7)
        valor <- indice
    else valor <- 0
    soma <- soma + valor
}, unit = "s")

# Exemplo 2:

set.seed(0)
vetor <- runif(n = 10, min = 0, max = 1)
microbenchmark(soma <- sum(vetor[vetor > 0.7]),
               unit = "s")
```

Loop

Nota: O valor médio do benchmark é sempre uma boa estimativa para o desempenho de uma função ou trecho de código, uma vez que é a média de diversos benchmarks (por padrão 100). Para alterar o número de testes, é possível modificar o parâmetro `times` da função `microbenchmark()` para um outro inteiro.

Nota: O valor médio do benchmark é sempre uma boa estimativa para o desempenho de uma função ou trecho de código, uma vez que é a média de diversos benchmarks (por padrão 100). Para alterar o número de testes, é possível modificar o parâmetro `times` da função `microbenchmark()` para um outro inteiro.

Lição:

Loop

Nota: O valor médio do benchmark é sempre uma boa estimativa para o desempenho de uma função ou trecho de código, uma vez que é a média de diversos benchmarks (por padrão 100). Para alterar o número de testes, é possível modificar o parâmetro `times` da função `microbenchmark()` para um outro inteiro.

Lição: Loops são estruturas computacionalmente intensivas na linguagem `R` e que muitas vezes podemos evitar utilizando a vetorização das operações que a linguagem disponibiliza.

Loop

Tentei de tudo, e não achei formas de evitar o loop ...

Loop

Tentei de tudo, e não achei formas de evitar o loop ...

Loop

Sem problemas, use sem peso na consciência. De fator eles são verdadeiramente úteis e muitas vezes é realmente inevitável não utilizar loops ...

Loop

Sem problemas, use sem peso na consciência. De fator eles são verdadeiramente úteis e muitas vezes é realmente inevitável não utilizar loops ...

Loop

Exercício: Escreva o trecho de código abaixo utilizando a instrução `while()`.

```
for (i in 1:20)
{
  if (i == 10) next
  else cat("i = ", i, "\n", sep = "")
}
```

Loop

Exercício: Escreva um programa em R utilizando as instruções de loop vistas anteriormente de modo a fornecer a seguinte estrutura a depender do valor de *n*.

```
Para n = 1        Para n = 2        Para n = 3  ...
*                 *                 *
                  **                **
                                    ***
```

Loop

Uma solução possível:

```
n <- 5
for (i in 1:n)
{
  j <- 1
  while (j <= i)
  {
     cat("*")
     j <- j + 1
  }
  cat("\n")
}
```

Exercício: Refaça o exemplo anterior de tal forma que a estrutura obtida seja:

```
Para n = 1        Para n = 2        Para n = 3  ...
A                 A                 A
                  BB                BB
                                    CCC
```

Loop

Um solução possível:

```
n <- 5
for (i in 1:n)
{
  j <- 1
  while (j <= i)
  {
     cat(LETTERS[i])
     j <- j + 1
  }
  cat("\n")
}
```

Loop

Exercício: Suponha que um professor possa fazer um número qualquer de avaliações e deseja construir um programa em `R` que receba as notas do aluno. O programa deverá informar se o aluno foi aprovado, reprovado ou irá para prova final. Em caso do aluno ir para prova final, o programa deverá informar qual a nota a partir da qual o aluno irá assumir o status de aprovado na disciplina.

Loop

Regra para avaliação:

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;
2. O aluno que tiver nota interior à 4.0 assumirá o status de reprovado sem direito à avaliação final;

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;
2. O aluno que tiver nota interior à 4.0 assumirá o status de reprovado sem direito à avaliação final;
3. Alunos com notas no intervalo $[4.0, 7.0)$ terá direito à fazer a prova final.

Regra para avaliação:

1. O aluno que tiver a média das primeiras provas maior ou igual à 7.0 assumirá status de aprovado sem precisar ir para avaliação final;
2. O aluno que tiver nota interior à 4.0 assumirá o status de reprovado sem direito à avaliação final;
3. Alunos com notas no intervalo $[4.0, 7.0)$ terá direito à fazer a prova final.
4. A média final será uma média ponderada que terá peso 4 para a nota final e peso 6 para a média das primeiras notas. O aluno com média final maior ou igual à 5.0 será considerado aprovado na disciplina.

Uma solução:

```
aluno <- function()
{
  n_avaliacoes <- as.numeric(readline(prompt =
                    "Quantas avaliações?: "))
  i <- 1
  v_provas <- NULL
  for (i in 1:n_avaliacoes)
     v_provas[i] <- as.numeric(readline(prompt =
                      paste("Nota ", i, " = ",
                      sep = "")))
   media <- mean(v_provas)

   if (media >= 7) situacao <- "Aprovado(a)"
   else if (media < 4) situacao <- "Reprovado(a)."
```

Loop

```
15:   else{
16:
17:      cat("O aluno(a) irá para final e precisará ter
18:      tirado nota maior ou igual à ", (50-6*media)/4,
19:      " para passar.\n")
20:
21:      final <- as.numeric(readline(prompt =
22:               "\nQual a nota da final?: "))
23:
24:      if (final >= (50-6*media)/4)
25:         situacao <- "Aprovado(a)"
26:   }
27:
28:   list(media = round(media,2), situacao = situacao)
29: }
```

Loop

Nota: Soluções interativa, em geral, devem ser evitadas. Por exemplo, no problema acima poderíamos ter informado antecipadamente um vetor com as notas. Em geral, nas simulações que precisamos realizar, não desejaremos essa interação. No caso das simulações, desejamos colocar o código para ser interpretado mesmo quando não estamos com contato físico com o computador.

Loop

Exercício: Utilizando as instruções de loop, some todos elementos de uma matriz. **Dica**: Gere aleatoriamente uma matriz de dimensões 5 por 2.

Solução:

```
set.seed(0)
matriz <- matrix(runif(10), 5, 2)
soma <- 0
for (coluna in 1:ncol(matriz))
  for (linha in 1:nrow(matriz))
      soma <- soma + matriz[linha, coluna]
soma
#> [1] 6.35005
```

Loop

Tornando ainda a falar sobre evitar loops, lembre-se de muitas funções em R são vetorizáveis. No caso do exercício anterior temos, poderíamos ter solucionado o problema fazendo:

Exemplo: Corra o código abaixo:

```
set.seed(0) # Fixando a semente do gerador.
matriz <- matrix(runif(10), 5, 2)
sum(matriz) # Dois loops evitados.
#> [1] 6.35005
```

Muito mais simples e eficiente resolver este problema da forma acima.

Loop

Exercício: Utilizando as instruções de loop, retorne um vetor com as somas das colunas de uma matriz. **Dica**: Gere aleatoriamente uma matriz de dimensões 10 por 10.

Loop

Exercício: Utilizando as instruções de loop, retorne um vetor com as somas das colunas de uma matriz. **Dica**: Gere aleatoriamente uma matriz de dimensões 10 por 10.

```
matriz <- matrix(runif(100, min = 0, max = 1),
            ncol = 10, nrow = 10)
soma <- 0
vetor <- NULL
for (coluna in 1:ncol(m))
{
  for (linha in 1:nrow(m))
  {
     soma <- soma + matriz[linha, coluna]
  } vetor[coluna] <- soma
}
```

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

```
apply(X, MARGIN, FUN, ...)
```

Informações sobre os parâmetros:

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

```
apply(X, MARGIN, FUN, ...)
```

Informações sobre os parâmetros:

- `X`: Matriz ou array;

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

```
apply(X, MARGIN, FUN, ...)
```

Informações sobre os parâmetros:

- `X`: Matriz ou array;
- `MARGIN`: 1 e 2 indicam linha e coluna, respectivamente;

Loop

Em situações como a do exercício anterior é possível utilizar funções que permitem aplicar operações a uma das dimensões de uma matriz ou array. A função `apply()` é uma delas.

Forma Geral:

```
apply(X, MARGIN, FUN, ...)
```

Informações sobre os parâmetros:

- `X`: Matriz ou array;
- `MARGIN`: 1 e 2 indicam linha e coluna, respectivamente;
- `FUN`: Função que queremos aplicar.

Loop

Exemplo: O problema anterior poderia ser resolvido de forma mais eficiente como segue:

```
matriz <- matrix(runif(100, min = 0, max = 1),
                 ncol = 10, nrow = 10)
apply(X = matriz, NARGIN = 2, FUN = min)
```

Loop

Uma outra função bastante útil na linguagem R é a função `tapply()`. Tal função é bastante semelhante à função `apply()`, porém com a vantagem de ser possível aplicar uma função à um subconjunto dos dados.

Exemplo:

```
data(warpbreaks)
head(warpbreaks) # Faça ?head para mais detalhes.

tapply(X = warpbreaks$breaks, INDEX =
       warpbreaks[, -1], FUN = sum)
#>    tension
#> wool  L   M   H
#> A   401 216 221
#> B   254 259 169
```

Loop

Uma outra função bastante útil para programadores de R é a função `sapply()` que permite aplicar uma função a todos os elementos de um vetor ou lista.

Exemplo:

```
lista <- list(t1 = sample(15), t2 =
                c(7.7,3.4,4.7,8.02), f3 = runif(n = 100))
sapply(lista, quantile)
#>        t1    t2         f3
#> 0%    1.0 3.400 0.01339033
#> 25%   4.5 4.375 0.33342987
#> 50%   8.0 6.200 0.48781071
#> 75%  11.5 7.780 0.76719336
#> 100% 15.0 8.020 0.99268406
```

Loop

Exercício: Estude a documentação da função `quantile()`.

Exercício: A linguagem `R` também apresenta a função `lapply()`. Estude a documentação desta função. Cite a principal diferença entre as funções. Apresente exemplos.

Existe ainda uma versão multivariada das funções `lapply()` e `sapply()` que é a função `mapply()`. As funções `lapply()` e `sapply()` atuam somente sobre os elementos de uma única lista. Porém, a função `mapply()` a função agirá sobre o primeiro elemento de cada um dos argumentos, em seguida ao segundo elemento e assim sucessivamente.

Loop

Exercício: Corra o código abaixo. Depois, explique o funcionamento da função `mapply()`.

```
1: mapply(rep, 1:4, 4:1)
#> [[1]]
#> [1] 1 1 1 1
#>
#> [[2]]
#> [1] 2 2 2
#>
#> [[3]]
#> [1] 3 3
#>
#> [[4]]
#> [1] 4
```

Loop

Exemplo: Corra o código:

```
l1 <- list(a = LETTERS[c(4,6,12,6)],
           b = LETTERS[c(1,5,21,1)])
l2 <- list(c = LETTERS[c(4,14,22,20)],
           d = LETTERS[c(15,15,1,15)])
mapply(paste, l1$a, l1$b, l2$c, l2$d)
#> "D A D O" "F E N O" "L U V A" "F A T O"
```

Exercício: Walter está jogando um jogo com dois dados de 6 lados equiprováveis. O jogo consiste em lançar ambos os dados e caso o resultado seja divisível por 3, ele ganhará $\$6,00$ dólares, caso contrário, ele perderá $\$3,00$ dólares. Simule 100 mil repetições desse experimento e obtenha o valor esperado de um jogador.

Loop

Exercício: Walter está jogando um jogo com dois dados de 6 lados equiprováveis. O jogo consiste em lançar ambos os dados e caso o resultado seja divisível por 3, ele ganhará $\$6,00$ dólares, caso contrário, ele perderá $\$3,00$ dólares. Simule 100 mil repetições desse experimento e obtenha o valor esperado de um jogador.

Dica: Utilize a função `sample()` para realizar o sorteio das faces de um dado, isto é, sortear números de 1 à 6 para cada um dos dados. Consulte a documentação para maiores detalhes.

Exercício: Suponha que tenhamos uma urna com bolas de mesmo tamanho enumeradas de 1 à 100. Considere o experimento aleatório de retirar uma bola da urna e observar o seu número até obtermos a bola com número desejado. Nesse experimento, será considerado reposição, isto é, caso não tenha sido observado a numeração desejada, a bola será devolvida à urna. Simule no computador 10 mil repetições desse experimento e obtenha uma média das retiradas necessárias para se obter o número desejado.

Loop

Exercício: Um dono de cassino estuda disponibilizar um novo jogo e solicita uma consultoria estatística para saber se o jogo será viável para o cassino, isto é, se o valor esperado em dólares do lucro obtido será positivo. **Pergunta-se**: Qual o valor esperado do lucro do cassino sabendo que o jogador pagará $\$10,00$ dólares para ter direito a jogar e em cada lançamento o jogador irá lucrar $\$1,50$ dólares.

Loop

Exercício: Um dono de cassino estuda disponibilizar um novo jogo e solicita uma consultoria estatística para saber se o jogo será viável para o cassino, isto é, se o valor esperado em dólares do lucro obtido será positivo. **Pergunta-se**: Qual o valor esperado do lucro do cassino sabendo que o jogador pagará $\$10,00$ dólares para ter direito a jogar e em cada lançamento o jogador irá lucrar $\$1,50$ dólares.

1. Dois dados são lançados e caso a soma for 5, 6, 7, 8 ou 9 o jogo termina imediatamente.

Loop

Exercício: Um dono de cassino estuda disponibilizar um novo jogo e solicita uma consultoria estatística para saber se o jogo será viável para o cassino, isto é, se o valor esperado em dólares do lucro obtido será positivo. **Pergunta-se**: Qual o valor esperado do lucro do cassino sabendo que o jogador pagará $\$10,00$ dólares para ter direito a jogar e em cada lançamento o jogador irá lucrar $\$1,50$ dólares.

1. Dois dados são lançados e caso a soma for 5, 6, 7, 8 ou 9 o jogo termina imediatamente.
2. Se nenhum dos resultamos acima não forem obtido, o jogador continua lançando ambos os dados até obter uma soma igual à 11 ou 12.

Dica: Realize uma simulação considerando 100 mil jogos e obtenha o valor médio de lançamentos e a probabilidade (aproximada) de o jogador jogar apenas uma partida.

Exercício: Seja $X_1, \ldots, X_n$ uma amostra aleatória da variável aleatória $X \sim \mathcal{N}(\mu, \sigma^2)$. Além disso, considere os estimadores $\hat{\sigma}^2$ e S^2 apresentados abaixo:

$$\begin{aligned} \hat{\sigma}^2 &= \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})^2 \qquad (1) \\ &\text{e} \\ S^2 &= \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2. \end{aligned}$$

Nota: Uma forma de compararmos dois estimadores é através do **Erro Quadrático Médio** (EQM) dos estimadores. Dessa forma, sejam $\hat{\theta}_1$ e $\hat{\theta}_2$ dois estimadores para um parâmetro θ **desconhecido**. Diremos que $\hat{\theta}_1$ é **melhor** que $\hat{\theta}_2$ se

$$EQM(\hat{\theta}_1) \leq EQM(\hat{\theta}_2),$$

em que $EQM(\hat{\theta}) = E[(\hat{\theta} - \theta)^2]$.

Lembre-se também que o **viés** de um estimador $\hat{\theta}$ do parâmetro θ é dado por

$$B(\hat{\theta}) = E(\hat{\theta}) - \theta.$$

Se $B(\hat{\theta}) = 0$ diremos que $\hat{\theta}$ é um **estimador não viesado** para θ, isto é, $E(\hat{\theta}) = \theta$.

Loop

Vamos obter os resultados por meio de **simulação**. Entenda **simulação** como uma simplificação da realidade em um ambiente virtual.

Dessa forma, serão gerados diversos cenários, onde em cada um deles serão gerados uma sequência de observações de uma variável aleatória $X \sim \mathcal{N}(0, 1)$. A referida sequência de observações formam um conjunto de dados distinto em cada iteração. Esse conjunto de dados simula um conjunto de dados real que poderíamos obter em uma situação prática no mundo real. Muito provavelmente, tais conjuntos de dados são distintos no que diz respeito a sequência de valores observados como ocorre na realidade.

Observação

Como os estimadores $\hat{\sigma}^2$ e S^2 se propõe estimar σ^2 (parâmetro de uma distribuição gaussiana), devemos gerar observações de variáveis aleatórias com distribuição gaussiana.

Para termos resultados razoáveis nas nossas simulações, é importante considerarmos um número elevado de repetições do experimento.

Observação

Como os estimadores $\hat{\sigma}^2$ e S^2 se propõe estimar σ^2 (parâmetro de uma distribuição gaussiana), devemos gerar observações de variáveis aleatórias com distribuição gaussiana.

Para termos resultados razoáveis nas nossas simulações, é importante considerarmos um número elevado de repetições do experimento.

Dica

Não há um número exato de repetições que precisaremos considerar. De toda forma, é razoável considerar um número de iterações no intervalo $[10000, +\infty)$.

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

1. Gere uma amostra aleatória de tamanho n, tal que $X_i \sim \mathcal{N}(0,1)\,\forall i$;

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

1. Gere uma amostra aleatória de tamanho n, tal que $X_i \sim \mathcal{N}(0,1)\,\forall i$;
2. Obtenha as estimativas $\hat{\sigma}^2$ e S^2 e guarde as estimativas;

Loop

Abaixo está apresentado um algoritmo para implementação da simulação, em que levará em consideração **20 mil** iterações.

Algoritmo: Para cada uma das 20 mil iterações, faça:

1. Gere uma amostra aleatória de tamanho n, tal que $X_i \sim \mathcal{N}(0,1)\,\forall i$;
2. Obtenha as estimativas $\hat{\sigma}^2$ e S^2 e guarde as estimativas;
3. Calcule o $EQM(\hat{\sigma}^2)$, $EQM(S^2)$, $B(\hat{\sigma}^2)$ e $B(S^2)$.

Ao final das 20 mil iterações, tire uma média de cada uma das estimativas no passo **2** e das estimativas do passo **3**.

Loop

Construa uma tabela com a média das estimativas obtidas por cada um dos estimadores ($\hat{\sigma}^2$ e S^2 obtidos no passo **2**) e com as estimativas obtidas no passo **3** para os tamanhos de amostra $n = 20, 60, 100$ e 1000. Discuta qual o melhor estimador com base nos resultados assintóticos obtidos pela simulação.

Loop

Construa uma tabela com a média das estimativas obtidas por cada um dos estimadores ($\hat{\sigma}^2$ e S^2 obtidos no passo **2**) e com as estimativas obtidas no passo **3** para os tamanhos de amostra $n = 20, 60, 100$ e 1000. Discuta qual o melhor estimador com base nos resultados assintóticos obtidos pela simulação.

Nota: Você talvez acaba de fazer sua primeira **simulação de Monte Carlo**. Mais adiante no curso apresentaremos mais detalhes a respeito de simulações de Monte Carlo.

Loop

Construa uma tabela com a média das estimativas obtidas por cada um dos estimadores ($\hat{\sigma}^2$ e S^2 obtidos no passo **2**) e com as estimativas obtidas no passo **3** para os tamanhos de amostra $n = 20, 60, 100$ e 1000. Discuta qual o melhor estimador com base nos resultados assintóticos obtidos pela simulação.

Nota: Você talvez acaba de fazer sua primeira **simulação de Monte Carlo**. Mais adiante no curso apresentaremos mais detalhes a respeito de simulações de Monte Carlo.

Observação: Avaliar assintoticamente um estimador quer dizer que iremos avaliar o estimador quando $n \to +\infty$.

Funções

Apesar da linguagem R possuir diversas funções disponíveis em uma infinidade de pacotes oficiais e não oficiais para os mais diversos fins, sempre precisaremos construir nossas funções. Sempre há algo que não tenha sido implementado ou as vezes precisaremos modificar um pouco a estrutura de uma função já existente implementada por você ou outro programador.

Funções

Apesar da linguagem R possuir diversas funções disponíveis em uma infinidade de pacotes oficiais e não oficiais para os mais diversos fins, sempre precisaremos construir nossas funções. Sempre há algo que não tenha sido implementado ou as vezes precisaremos modificar um pouco a estrutura de uma função já existente implementada por você ou outro programador.

Nota: Na maioria das vezes, sempre é útil dividir nosso programa em funções que realizam tarefas específicas. Estas funções devem ser implementadas da forma mais geral possível para que possam ser reaproveitadas em outros programas sem a necessidade de modificações.

Funções

Na linguagem R as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

Funções

Na linguagem R as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

1. Função possui uma lista de argumentos;

Funções

Na linguagem R as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

1. Função possui uma lista de argumentos;
2. Corpo;

Funções

Na linguagem R as funções também são objetos que podem ser manipulados de forma semelhante à outros objetos de estudamos anteriormente.

Três características principais de uma função:

1. Função possui uma lista de argumentos;
2. Corpo;
3. Ambiente em que a função está definida.

Funções

Funções

1. A lista de argumentos refere-se à uma lista de símbolos (os argumentos da função) que são separados por vírgula, em que tais argumentos podem possuir valores por padrão. Além disso, é possível que uma função possua um número variado de argumentos. Neste caso, é utilizado o argumento ... . Daremos detalhes mais a frente.

Funções

1. A lista de argumentos refere-se à uma lista de símbolos (os argumentos da função) que são separados por vírgula, em que tais argumentos podem possuir valores por padrão. Além disso, é possível que uma função possua um número variado de argumentos. Neste caso, é utilizado o argumento ... . Daremos detalhes mais a frente.

2. O corpo da função trata-se de um conjunto de instruções da linguagem `R` e normalmente forma um bloco de instruções;

Funções

1. A lista de argumentos refere-se à uma lista de símbolos (os argumentos da função) que são separados por vírgula, em que tais argumentos podem possuir valores por padrão. Além disso, é possível que uma função possua um número variado de argumentos. Neste caso, é utilizado o argumento ... . Daremos detalhes mais a frente.

2. O corpo da função trata-se de um conjunto de instruções da linguagem `R` e normalmente forma um bloco de instruções;

3. O ambiente de definição da função refere-se ao ambiente ativo quando a função foi criada. Isso determina quais são os objetos visíveis pela função.

Funções

Sintaxe da declaração de uma função:

Funções

Sintaxe da declaração de uma função:

```
nome_da_funcao <- function(argumentos){
    bloco de instruções correspondente
    ao corpo da função.
}
```

Funções

Apesar de ser desejável que uma função contenha argumentos que possam receber dados e informações de como processar as informações passadas, uma função poderá possuir nenhum argumento. Normalmente essas funções são bastante específicas para se tornar útil mesmo sem possuir argumentos. Não é o caso da função que segue no exemplo que segue.

Exemplo: Corra o código seguinte:

```
funcao <- function(){
  cat(paste(c("<3", LETTERS[c(1,13,15)], " ",
  LETTERS[c(13,5,21)], " ", LETTERS[c(16,18,15,6,5,
  19,19,15,18)], "<3"), collapse = " "))
}
```

Funções

Note que a criação de uma função consiste na atribuição do conteúdo no bloco de código que compõe a função e seus argumentos a um nome, como qualquer outro objeto da linguagem `R`.

Exemplo: Corra o código abaixo que refere-se a implementação da função `celsiustofar()` que converter uma temperatura informada em graus Celsius (passada ao argumento `cel` da função) para Fahrenheit.

Funções

Note que a criação de uma função consiste na atribuição do conteúdo no bloco de código que compõe a função e seus argumentos a um nome, como qualquer outro objeto da linguagem `R`.

Exemplo: Corra o código abaixo que refere-se a implementação da função `celsiustofar()` que converter uma temperatura informada em graus Celsius (passada ao argumento `cel` da função) para Fahrenheit.

```
celsiustofar <- function(cel)
{
  temperatura <- 1.8 * cel + 32
  temperatura
}
celsiustofar(cel = 21.3)
#> [1] 70.34
```

Funções

Nota: Caso a função possa ser escrita em uma única linha, o uso das chaves que delimitam o bloco da função ({}) poderá ser omitido. Considere o exemplo abaixo:

```
1: celsiustofar <- function(cel) 1.8 * cel + 32
```

Exercício: Suponhamos que temos o vetor `temp` de temperaturas em graus Celsius que encontra-se apresentado mais a baixo. Construa um programa que utilize a função `celsiustofar()` e retorne um vetor de temperaturas convertidas para Fahrenheit.

```
1: # Considere no exemplo o vetor abaixo:
2: temp <- c(27.8,19.3,20.7,18.29,25.0,25.1,32.3,
3:           37.6,32.2,19.02,22.75)
```

Funções

Exercício: Altere a função `celsiustofar()` apresentada no exemplo anterior de modo que a função possa converter a temperatura de graus Celsius para Fahrenheit ou de Fahrenheit para graus Celsius.

Funções

Exercício: Altere a função `celsiustofar()` apresentada no exemplo anterior de modo que a função possa converter a temperatura de graus Celsius para Fahrenheit ou de Fahrenheit para graus Celsius.

Solução:

```
celsiustofar <- function(temp, ctof = TRUE)
{
    ctof <- as.logical(ctof)
    if(!is.logical(ctof) | is.na(ctof))
        stop("Um valor lógico não foi informado.")

    temperatura <- ifelse(ctof == TRUE, 1.8 *
                                temp + 32, (temp - 32)/1.8)
    temperatura
}
```

Nota: O operador `!` é o operador **NOT** lógico na linguagem `R`.

Observe que a solução do exercício logo acima não ocasionaria problemas caso o usuário viesse a fornecer as strings (`"TRUE"`, `"FALSE"`, `"T"`, `"F"`, `"True"` e `"False"`) além dos valores lógicos `T`, `TRUE`, `F` e `FALSE`, propriamente ditos.

Exercício: Faça `help(is.na)` e leia a documentação. Faça o mesmo para a função `stop()`.

Funções

Importante

O resultado da última computação de uma função será o retorno da função. Nos exemplos acima, `temperatura` é o retorno da função `celsiustofar()`. Porém, em `R`, a função `return()` poderá ser utilizada para encerrar a execução da função no ponto da função em que `return()` se encontra. Sendo assim, o argumento de `return()` será retornado como resultado da função que estamos implementando.

Funções

Exercício: Construa uma função que calcula o IMC (Índice de Massa Corporal) de uma pessoa e retorne o IMC e a situação da pessoa com respeito ao seu peso.

Funções

Exercício: Construa uma função que calcula o IMC (Índice de Massa Corporal) de uma pessoa e retorne o IMC e a situação da pessoa com respeito ao seu peso.

Cálculo do índice: $\mathrm{IMC} = \mathrm{peso}/\mathrm{altura}^2$.

Considere a tabela que segue a baixo as correspondências entre o IMC e a situação do paciente de acordo com o índice. Utilize essa tabela para construir o programa do exercício acima.

Tabela: Situação do peso segundo o IMC.

IMC	**Situação**
<17	Muito abaixo do peso
[17; 18.49]	Abaixo do peso
[18.5; 24.99]	Peso normal
[25; 29.99]	Acima do peso
[30; 34.99]	Obesidade I
[35; 39.99]	Obsesidade II (severa)
>40	Obesidade III (mórbida)

Uma possível solução:

```
imc <- function(peso,altura)
{
    # A altura deverá ser informada em metros.
    # O peso deverá ser informado em kg.

    # Uma pessoa que colocar valores negativos
    # não merece nem comentário. Não retorne
    # nada.
    if (peso <= 0 | altura <= 0) return(NULL)

    if (altura >= 3) warning("Você é desse planeta?")

    imc <- peso/altura^2
```

Funções

```
14:   if (imc < 17) situacao <- "Muito abaixo do peso"
15:   else if (imc >= 17 & imc <= 18.49)
16:       situacao <- "Abaixo do peso"
17:   else if (imc >= 18.5 & imc <= 24.99)
18:       situacao <- "Peso normal"
19:   else if (imc >= 25 & imc <= 29.99)
20:       situacao <- "Acima do peso"
21:   else if (imc >= 30 & imc <= 34.99)
22:       situacao <- "Obesidade I"
23:   else if (imc >= 35 & imc <= 39.99)
24:       situacao <- "Obesidade II (severa)"
25:   else situacao <- "Obesidade III (mórbida)"
26:
27:   list(imc = imc, situacao = situacao)
28: }
```

Funções

Nota: Observe no programa acima as instruções na instrução condicional `if()` seja avaliada, a função retornará `NULL` e nada mais será avaliado.

Algumas funções matemáticas e estatísticas que podem ser úteis para a realização de computação científica serão listadas abaixo. Ficará como exercício o aluno ler a documentação de cada uma dessas funções e executarem os exemplos contidos na documentação.

Tais funções são úteis na implementações de novas funções que frequentemente utilizamos na computação científica.

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;
- `range(x)`: Vetor com o valor máximo e mínimo de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;
- `range(x)`: Vetor com o valor máximo e mínimo de `x`;
- `length(x)`: Quantidade de elementos de `x`;

Funções

Algumas funções matemáticas e estatísticas:

Sejam `x` e `y` um vetor de dados, então:

- `sum(x)`: somas dos elementos do vetor `x`;
- `max(x)`: Máximo dos elementos do vetor `x`;
- `min(x)`: Mínimo dos elementos de `x`;
- `which.max(x)`: Índice do maior valor de `x`;
- `which.min(x)`: Índice do menor valor de `x`;
- `range(x)`: Vetor com o valor máximo e mínimo de `x`;
- `length(x)`: Quantidade de elementos de `x`;
- `mean(x)`: Média dos valores de `x`;

Funções

- `median(x)`: Mediana dos valores de `x`;

Funções

- `median(x)`: Mediana dos valores de `x`;
- `sd(x)`: Desvio padrão dos elementos do vetor `x`;
- `var(x)`: Variância dos elementos do vetor `x`;
- `quantile(x)`: Os quartis de `x`;
- `scale(x)`: Normaliza os elementos do vetor `x`, isto é, subtrai os elementos da média e divide pelo desvio-padrão;
- `sort(x)`: Ordena os elementos do vetor `x`;
- `rank(x)`: Faz o *ranking* dos elementos de `x`;
- `log(x,base)`: Calcula o logaritmo dos elementos de `x` na `base` escolhida. Se nenhuma base for escolhida, será calculado o logaritmo natural;
- `exp(x)`: Calcula o exponencial dos elementos de `x`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;
- `match(x,y)`: Retorna as posições dos elementos de `x` em `y`;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;
- `match(x,y)`: Retorna as posições dos elementos de `x` em `y`;
- `union(x,y)`: Retorna o vetor união;

Funções

- `sqrt(x)`: Calcula a raiz quadrada dos elementos de `x`;
- `abs(x)`: Valor absoluto dos elementos de `x`;
- `round(x,n)`: Arredonda os elementos de `x` para `n` casas decimais;
- `cumsum(x)`: Obtém um vetor em que o elemento i é a soma dos elementos x_1 à x_i;
- `cumprod(x)`: O mesmo para o produto;
- `match(x,y)`: Retorna as posições dos elementos de `x` em `y`;
- `union(x,y)`: Retorna o vetor união;
- `intersect(x,y)`: Obtém um vetor com a interseção dos vetores `x` e `y`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;
- `choose(n,k)`: Calcula o número de combinações n, k a k;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;
- `choose(n,k)`: Calcula o número de combinações n, k a k;
- `factorial(x)`: Calcula o fatorial dos elementos de `x`;

Funções

- `setdiff(x,y)`: Retorna os elementos do vetor `x` que não pertence ao vetor `y`;
- `is.element(x,y)`: Retorna `TRUE` se o elemento de `x` está contido no vetor `y` e `FALSE`, caso contrário. A função `is.element()` é semelhante ao operador `%in%`;
- `choose(n,k)`: Calcula o número de combinações n, k a k;
- `factorial(x)`: Calcula o fatorial dos elementos de `x`;
- `pi`: Retorna o vetor com um único elemento com o valor de π;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;
- `det(A)`: Determinante da matriz `A`, se `A` é uma matriz quadrada;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;
- `det(A)`: Determinante da matriz `A`, se `A` é uma matriz quadrada;
- `solve(A)`: A inversa da matriz `A`, se `A` é uma matriz quadrada;

Funções

Algumas funções para álgebra matricial:

Sejam A e B matrizes e b um vetor atômico, então:

- `diag(a,nrow,ncol)`: Constrói uma matriz diagonal com `nrow` linhas e `ncol` coluna, usando o número `a`;
- `t(A)`: A matriz transposta de `A`;
- `A%*%B`: O produto matricial entre as matrizes `A` e `B`;
- `det(A)`: Determinante da matriz `A`, se `A` é uma matriz quadrada;
- `solve(A)`: A inversa da matriz `A`, se `A` é uma matriz quadrada;
- `solve(A,b)`: Resolve o sistema de equações lineares $Ax = b$;

Funções

- `eigen(A)`: Auto-valores e auto-vetores da matriz `A`, se `A` é uma matriz quadrada.

Exercício: Apresente exemplos do uso das funções estatísticas e algébricas apresentadas acima.

Exercício: Refaça o exercício referente as simulações de Monte Carlo para avaliação dos estimadores $\hat{\sigma}^2$ e S^2 em frames anteriores utilizando o conceito de funções.

Nota: Sabemos o limite da frequência relativa da ocorrência de um evento A qualquer obtido da realização de um experimento aleatório converge para a probabilidade da ocorrência de A, isto é

$$P(A) = \lim_{n \to +\infty} \frac{n_A}{n},$$

em que n_A é o número de ocorrência do evento de interesse A em n repetições do experimento aleatório (**teoria frequentista da probabilidade**).

Nota: Sabemos o limite da frequência relativa da ocorrência de um evento A qualquer obtido da realização de um experimento aleatório converge para a probabilidade da ocorrência de A, isto é

$$P(A) = \lim_{n \to +\infty} \frac{n_A}{n},$$

em que n_A é o número de ocorrência do evento de interesse A em n repetições do experimento aleatório (**teoria frequentista da probabilidade**).

Por exemplo, no experimento aleatório que consiste em lança uma moeda e verificar a ocorrência de "cara" poderíamos estar interessados no evento $A = \{$"ocorrência de cara e um lançamento"$\}$. Então n_A é o número de ocorrência de caras em n lançamentos da moeda.

Como a moeda é honesta, isto é, $P(\text{“}cara\text{”}) = P(\text{“}coroa\text{”})$, esperamos que

$$P(A) = P(\text{“}cara\text{”}) = \lim_{n \to +\infty} \frac{n_A}{n} = \frac{1}{2}.$$

Como a moeda é honesta, isto é, $P(\text{“}cara\text{”}) = P(\text{“}coroa\text{”})$, esperamos que

$$P(A) = P(\text{“}cara\text{”}) = \lim_{n \to +\infty} \frac{n_A}{n} = \frac{1}{2}.$$

Exercício: Construa uma função que simule o lançamento de uma moeda. Posteriormente simule para para $n = 10$, 20, 50, 100, 1000 e 100000 lançamentos da moeda. Obtenha o limite da frequência relativa para cada tamanho de amostra com uma aproximação à $P(A)$. Comente o código. **Dica**: utilize a função `sample()` para amostrar no conjunto $\{0, 1\}$, em que 0 refere-se a **coroa** e 1 refere-se à **cara**.

Nota: Observe que o exercício propõe simular um experimento real e bastante simples que consiste no lançamento de uma moeda honesta. Mais um exemplo de uma simulação de Monte Carlo.

Funções

Em R, caracteres especiais não são permitidos para nomes de funções. Por exemplo, não é permitido criar uma função de nome +, −, * ou /.

Muito embora tratamos os operadores acima como operadores binários, pois assim o são, poderemos fazer uso desses operadores literalmente especificando o nome da função (do operador) e os seus argumentos suportados.

Exemplo: Corra o código que segue:

```
1: "+"(2,5)
2: "*"(3,4)
3: "-"(5,1)
4: "/"(2,8)
5: "+"(1, "*"(2,3))
```

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

1. `body()`: mostra o código inserido no corpo da função;

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

1. `body()`: mostra o código inserido no corpo da função;
2. `formals()`: retorna a lista de argumentos que controla o funcionamento da função;

Funções

Observação: Quando imprimimos uma função em `R`, normalmente é mostrado os três principais componentes (argumentos, corpo e ambiente). Caso venhamos imprimir uma função e seu ambiente não é especificado, isso significa que a função foi criada no ambiente global (**workspace**).

É possível extrair as três principais componentes de uma função utilizando as funções:

1. `body()`: mostra o código inserido no corpo da função;
2. `formals()`: retorna a lista de argumentos que controla o funcionamento da função;
3. `environment()`: retorna o "map" da localização das variáveis utilizadas na função.

Funções

Exemplo: Corra o código abaixo:

```
1: f <- function(x) x^2
2: f
#> function(x) x^2 # Não foi retornado o ambiente.
3:
4: formals(f)
#> $x
5: body(f)
#> x^2
6: environment(f)
#> <environment: R_GlobalEnv>
```

Funções

Nota: Tais funções também podem ser utilizadas para modificar uma função.

Exemplo: Corra o exemplo abaixo:

```
f <- function(x) x^2
body(f) <- expression(x^3 + c)
formals(f) <- list(x = 1, c = 2)
f
#> function (x = 1, c = 2)
#> x^3 + c
```

Funções

As funções `formals()`, `body()` e `environment()` poderão retornar `NULL` caso elas não estejam escritas em `R`.

Nota: Algumas funções em `R` podem ser escritas em `C`, `C++` ou mesmo `Fortran`.

Exemplo: A função `sum()` é escrita em `C`, em que o código é chamado pela função usando `.Primitive()`. Sendo assim, as funções `formals()`, `body()` e `environment()` retornarão `NULL`.

Funções

```
1: sum
#> function (..., na.rm = FALSE) .Primitive("sum")
2: formals(sum)
#> NULL
3: body(sum)
#> NULL
4: environment(sum)
#> NULL
```

Exercício: Construa a função `myderiv()` para o cálculo da derivada numérica de uma função em um ponto. Lembre-se, se f é uma função diferenciável em x_0, a derivada de $f(x)$ no ponto x_0 é dada por

$$f^{'}(x_0) = \lim_{h \to 0} \frac{f(x_0 + h) - f(x_0)}{h}.$$

Por meio dessa definição, tomando $h \neq 0$, temos que uma aproximação à $f^{'}(x_0)$ é dada por:

$$f^{'}(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

Dica: h não poderá ser "muito" pequeno uma vez que isso poderia implicar em cancelamentos catastróficos.

Solução:

```
f <- function(x){
  x^4 * sin(x)
}

numeric_derivation <- function(func, x0 = 1, h = 0.0001){
  if (h == 0 | h >= 1) stop("h devara assumir valores no
      intervalo (0,1).")

  a <- func(x0 + h)
  b <- f(x0)

  # Buscando um novo valor de h.
  if((a - b) == 0) { # Cancelamento catastrofico.
    repeat{
      h <- h * 10
      a <- func(x0 + h)
      b <- f(x0)
```

Funções

```
17        # Criterio de parada do repeat.
18        if((a-b) > 1e-5) break
19      }
20    }
21    return(list(derivada = round((a - b)/h, 4), h_usado = h))
22  }
23  numeric_derivation(func = f, x0 = 1)
```

Funções

Nota: Funções primitivas só são encontradas no pacote base e, como elas operam em um nível baixo, podem ser mais eficientes. Como são escritas em outras linguagens, estas normalmente possuem regras diferentes para correspondência dos argumentos. Normalmente a equipe de desenvolvimento da linguagem `R` tenta evitar a criação de funções primitivas, a menos que não haja outra opção.

Funções

Exercício: Construa a função `myloglikelihood()`, em `R`, que receba como argumentos uma função densidade de probabilidade implementada pelo usuário e uma amostra aleatória. A função deverá retornar o valor de log-verossimilhança com base nos argumentos.

Funções

Exercício: Construa a função `myloglikelihood()`, em `R`, que receba como argumentos uma função densidade de probabilidade implementada pelo usuário e uma amostra aleatória. A função deverá retornar o valor de log-verossimilhança com base nos argumentos.

Exercício: Melhore a função `myloglikelihood()` para que esta também identifique se a função digitada é ou não uma função densidade de probabilidade.

Depuração

É bastante comum implementarmos um programa em alguma linguagem de programação e observarmos ocorrências de erros. Também é comum não sabermos exatamente a fonte. Para facilitar a busca de tais problemas ou comportamentos imprevistos, algumas linguagens possuem comandos ou ferramentas que permitem a realização de **debugging**. A linguagem `R` possui diversas funções para este fim. No momento, abordaremos duas:

Depuração

Depuração

1. `traceback()`: Permite o programador obter a sequência de funções que foram chamadas até ocorrer o mau comportamento;

Depuração

1. `traceback()`: Permite o programador obter a sequência de funções que foram chamadas até ocorrer o mau comportamento;

2. `browser()`: Caso a função acima não ajude a resolver o problema, ou então se conseguimos encontrar sua origem mas não entendemos o seu porquê, poderemos utilizar a função `browser()` que fará que todo o código após sua posição seja executado passo a passo e sob a mesma ordem.

Nota: Ao encontrar a função `browser()` a linguagem `R` vai parar e só executará a instrução seguinte quando o programador solicita o avanço por meio de um **ENTER**, parando logo em seguida aguardando um novo comando.

Depuração

Nota: Em cada uma da paradas a linguagem `R` apresentará um prompt de comando especial (`Browse[1]>`) em que o programador poderá fazer quase tudo o que é permitido fazer no prompt de comando "normal"(>), como por exemplo checar o conteúdo de objetos.

Exemplo: Corra o código abaixo:

```
funcao <- function(n,x) {
  s <- 0
  for(i in 1:n) {
     s <- s + sqrt(x)
     x <- x-1
  }; s
}; funcao(2,4)
#> [1] 3.732051
```

Depuração

```
8: funcao(3,1)
#> [1] NaN
#> Warning message:
#> In sqrt(x) : NaNs produzidos
```

Depuração

```
8: funcao(3,1)
#> [1] NaN
#> Warning message:
#> In sqrt(x) : NaNs produzidos
```

Nota: Quando implementamos uma função, na grande maioria dos casos não queremos que o retorno seja `NaN`. O retorno `NaN` (**Not a Number**) normalmente refere-se ao retorno de uma expressão matemática que não pode ser avaliada. Por exemplo, `Inf - Inf`, `0 * Inf`, `Inf / Inf`, `0 / 0`, entre outros casos possíveis.

Depuração

Vamos incluir uma chamada à função `browser()` no corpo da função antes do local ao qual achamos que o erro encontra-se presente.

Exemplo:

```
funcao <- function(n,x) {
  s <- 0
  browser()
  for(i in 1:n) {
     s <- s + sqrt(x)
     x <- x-1
  }; s
}; funcao(3,1)
```

Escopo de Variáveis

Ao criarmo novos objetos (vetores, matrizes, funções, entre outros) na linguagem `R`, estamos criando-os em um ambiente denominado de **workspace** que também poderá ser chamado de **ambiente de topo**.

Por exemplo, ao invocarmos à função `ls()`, o que obtemos é um vetor de objetos nesse ambiente de topo. Porém, há diversas situações em que levam a criação de novos ambientes, levando assim a uma hierarquia de ambientes aninhando uns aos outros até o ambiente de topo.

Escopo de Variáveis

Qual a importância disso?

Escopo de Variáveis

Qual a importância disso?

Resposta: A importância de entender o escopo de variáveis reside nas regras de **scope** ou de **visibilidade** dos objetos em R que dependem do ambiente em que eles são definidos.

Escopo de Variáveis

Qual a importância disso?

Resposta: A importância de entender o escopo de variáveis reside nas regras de **scope** ou de **visibilidade** dos objetos em R que dependem do ambiente em que eles são definidos.

Importante: Quando criamos uma função, ela ficará associada ao ambiente ao qual foi criada, em que este ambiente normalmente é o **workspace**. Porém, ao chamarmos uma função, a linguagem R irá executá-la em um novo ambiente "**por baixo**"do ambiente em que ela foi chamada. Dessa forma, o ambiente onde serão executadas as instruções que formam o corpo da função não será o mesmo que executará a função.

Escopo de Variáveis

Essa diferença entre ambiente onde um objeto (nesse caso uma função) é chamado e ambiente onde as instruções são avaliadas tem impacto nos objetos que são visíveis em cada um destes dois ambientes.

Exemplo: Corra o código que segue:

```
x <- 2; z <- 3
f <- function(x) {
  a <- 34
  y <- x / 4 * a * z
  y
}
f(2)
#  [1] 51
a
# Error: Object "a" not found
```

Escopo de Variáveis

Observe duas coisas:

1. Note que o objeto a foi definido no bloco de instruções da função `f()` (linhas 3 à 5). Essas instruções são executadas em um ambiente mais a baixo do **workspace**. Sendo assim, ao chamar o objeto a na linha 8 (**workspace**), este não será reconhecido.

2. Por outro lado, note que na linha 4 fazemos referência ao objeto z. Muito embora z não esteja definido no ambiente que executará o bloco de instruções da função `f()`, o objeto será reconhecido no ambiente mais abaixo.