R e RStudio para Iniciantes

Material de Apoio para Cursos Quantitativos do Instituto de Economia da Universidade Federal do Rio de Janeiro (IE/UFRJ)

GPEQ/UFRJ

2024-04-19

Índice

Prefácio

O que você vai aprender

Pretendemos que você domine o *mínimo* necessário de programação em R para executar as tarefas que podem ser requisitadas pelo seu professor, independentemente do curso da área quantitativa em que estiver. Em outras palavras, se te pedirem algo que deva ser elaborado com auxílio de programação em R, você será capaz de fazê-lo após ler este material[1].

Na prática, o quê significa *dominar o mínimo necessário de programação em R?* Inclui entender alguns *conceitos* básicos – para quê serve a programação em nosso contexto, o que é a linguagem de programação R, o que é o RStudio, entre outros – assim como a *sintaxe* da linguagem – ou seja, o ato de escrever um código interpretável propriamente dito.

O que você não vai aprender

Não estamos em um curso de Ciência da Computação: você não irá aprender terminologias difíceis e/ou como a programação, de modo geral, funciona nos *detalhes*. Em outras palavras, vamos nos concentrar apenas em entender o necessário para construir e executar *códigos* em R (não se preocupe, ainda explicaremos o que é um *código em R*) a partir das tarefas que seu professor poderá pedir.

Além disso, o material não te dará proficiência em R. O que queremos dizer com isso? Bom, queremos dizer que você não será uma pessoa que dominará o R de forma *avançada*. Novamente: aqui, te ensinaremos apenas o necessário para que consiga concluir os cursos da área quantitativa. Mas, se você realmente quiser alcançar níveis mais altos, alguns livros podem te ajudar:

- R for Data Science (2ª edição)
- Ciência de Dados em R
- Data Science for Psychologists
- An Introduction to R for Research

[1] Esperamos que os empecilhos que apareçam não sejam por conta de alguma dificuldade no ato de programar em si, mas por dúvidas com relação à matéria propriamente dita. De qualquer forma, fique tranquilo: se você não entendeu alguma parte do material, estaremos **sempre** abertos a te ajudar!

Preciso saber alguma coisa de forma antecipada?

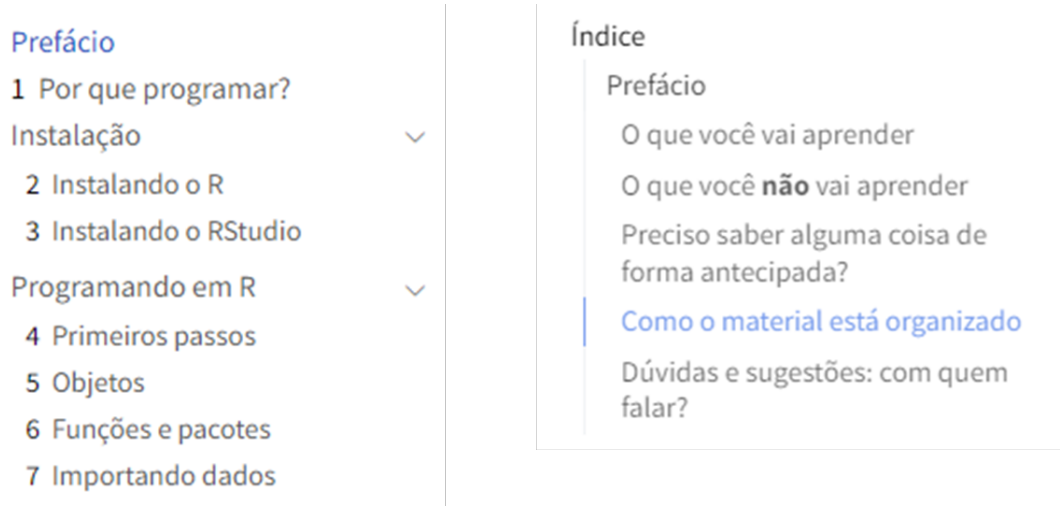
Não. Você não precisa saber absolutamente *nada* de programação em R – não precisa nem mesmo saber o que o termo *programação* significa. O intuito do material é justamente te introduzir aos conceitos mais básicos!

A única coisa que você precisará será de acesso à um computador com internet. Utilizar um computador é necessário pois é nele onde ocorre o ato de programar; ter internet é importante porquê, ao longo dos captíulos, precisaremos que você realize o *download* de certos arquivos – seja para instalar o R e o RStudio ou para *importar* algum arquivo diretamente para este último (não se preocupe, ainda explicaremos o que *importação* de um arquivo significa).

Como o material está organizado

O material está organizado em sete capítulos: o primeiro, que te mostra a motivação para programar, além de outros seis que buscam, em primeiro lugar, te guiar na instalação do R e RStudio e, na sequência, ensinar comandos e conceitos básicos que serão necessários ao longo dos cursos. Com intuito de facilitar o aprendizado, cada capítulo foi repartido em um certo número de seções (e subseções, quando necessário).

A lista de capítulos pode ser observada no menu à *esquerda*. Por sua vez, a lista de seções do capítulo em que você estiver pode ser observada no menu à *direita*. Perceba que, para ser direcionado a um determinado capítulo/seção, basta clicar em seu nome.



"Caramba, queria tanto acessar uma parte específica do material que não lembro muito bem onde está... E agora?" Sem problemas: você pode pesquisar partes do texto ou palavras-chave no campo em branco logo acima do Prefácio!

Dúvidas e sugestões: com quem falar?

"Ué, no meu computador não aparece isso!"
"Caramba, achei aquele trechinho ali meio confuso... podia melhorar..."
"Nossa, que material show!"

Surgiu alguma dúvida ou então quer dar alguma sugestão de melhoria? Estamos totalmente abertos à qualquer tipo de crítica! Envie uma mensagem para pedro.hemsley@ie.ufrj.br.

1 Por que programar?

A programação é o meio que dá vida ao mundo em que vivemos atualmente. Desde os sistemas operacionais que alimentam nossos computadores até os aplicativos de celular que usamos diariamente, a programação tem um papel fundamental. Mas o que exatamente é programação e como ela funciona? Como vamos utilizá-la nas nossas aulas?

De forma simplificada, *programação é a arte de escrever instruções para computadores executarem tarefas específicas.* Essas instruções são escritas através de **linguagens de programação**, que podem variar de simples e diretas, como R e Python, a complexas e poderosas, como C++ ou Java, e armazenadas em **códigos**. *Ao aprender a programar, você essencialmente está ensinando ao computador como resolver problemas, realizar cálculos, manipular dados ou criar interações com algum outro usuário.* Naturalmente, passamos a nos perguntar:

i. *"O que é uma linguagem de programação?"*
ii. *"O que é um código?"*

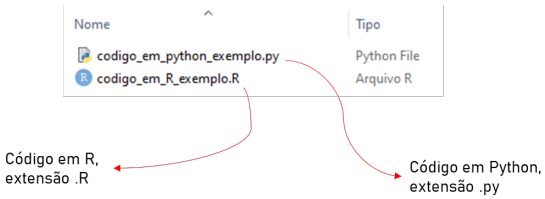
A linguagem de programação é um conjunto de regras e instruções utilizadas para escrever programas de computador. Ela define a sintaxe e a semântica que os programadores devem seguir para comunicar suas intenções ao computador – ou seja, as regras de escrita. Observe que a palavra 'linguagem' não está por acaso. Pense na língua portuguesa, por exemplo: ela possui um conjunto de regras que moldam sua estrutura (tanto em termos de palavras quanto de concordância), isto é, possui uma sintaxe própria, e serve para que duas ou mais pessoas possam se comunicar. De certa forma parecido, não é?

> **i** Linguagem de Programação
>
> É o meio pelo qual os programadores se comunicam com o computador, com intuito de instruí-lo a realizar determinada tarefa. Uma linguagem de programação é caracterizada pela sua *sintaxe* própria.

Por outro lado, o código é o resultado da aplicação dessas regras e instruções em uma linguagem de programação específica. Em outras palavras, o código é o texto ou conjunto de comandos escritos em uma linguagem de programação para realizar uma determinada tarefa ou função. Voltando à analogia da língua portuguesa, pense no código como um texto propriamente escrito em português. É a materialização da comunicação.

Código

É o texto que contém comandos em determinada linguagem de programação. Você pode escrever um código em um arquivo do tipo `.txt`, por exemplo. No entanto, na maioria dos casos, basta observar a extensão do arquivo!



A capacidade de programar tornou-se uma habilidade essencial, especialmente para aqueles que desejam explorar o mundo da estatística e da matemática aplicados à determinada ciência social. Por exemplo, no contexto de interseção entre economia e matemática – principalmente na elaboração e solução de modelos teóricos – e entre economia e estatística – testando hipóteses e realizando previsões – a programação se coloca como uma ferramenta muito útil para economizar tempo de cálculo e garantir que, caso necessário, o mesmo processo seja concluído múltiplas vezes sem erros. Em outras palavras, a programação aplicada à determinada ciência social, como a economia, traz duas principais vantagens, explicadas melhor a seguir.

1.1 Redução no tempo de cálculo

A primeira vantagem é a redução no tempo de cálculo de certos procedimentos que, se feitos de forma manual, levariam vários minutos, horas ou até mesmo dias. Vamos deixar mais claro com um exemplo.

No ensino fundamental, você aprendeu a resolver um sistema de equações simultâneas com 2 equações e 2 incógnitas, muito provavelmente pelo método de substituição. Não levava muito tempo, certo? Acontece que, na cadeira de Álgebra Linear, você aprenderá como identificar a existência e encontrar a solução de sistemas com *muitas* equações e *muitas* variáveis – dado que, ao longo do curso, você terá contato com sistemas determinísticos maiores cujo objetivo é explicar a realidade de forma simplificada. Nos que possuem solução, quanto mais complexo for o sistema, maior será a dificuldade de encontrá-la. Durante o curso, você aprenderá métodos que permitem descobrir a resolução de sistemas desse tipo de forma mais rápida, pelo menos em relação ao método de substituição. No entanto, certo tempo continuará sendo perdido se você os replicar *manualmente*.

Com auxílio da programação, é possível implementar estes mesmos métodos para obter o resultado de forma quase que instantânea! *O tempo que você levaria realizando o procedimento*

de forma manual se reduz próximo a zero – ou fica mínimo, comparado ao tempo que seria gasto se você resolvesse, na mão, todos os cálculos inerentes ao método em questão.

> **Observe que você ainda deve focar em saber como os métodos funcionam, do contrário não será capaz de julgar se o que a máquina fez é realmente aquilo que você desejava!**

1.2 Automação de processos

Na seção anterior, repare que estavamos discorrendo implicitamente sobre cálculos de ocorrência única – ou seja, realizamos o cálculo uma vez e não teríamos mais interesse de fazê-lo novamente em um futuro próximo. Porém, é importante destacar que outro benefício prático do ato de programar é a automação de tarefas repetitivas. Com a programação, é possível escrever e salvar códigos que realizam as mesmas tarefas tediosas de manipulação e análise de dados sempre que necessário, permitindo que os pesquisadores se concentrem em questões analíticas de maior relevância.

Por exemplo, imagine que alguém te peça para calcular a média de alguns valores, digamos dez, que mudam de dia para dia. Você poderia facilmente elaborar um código que, a partir de determinados números (independente de quais sejam), calcule a média correspondente. Uma vez o código escrito e salvo, você pode passar a executá-lo sempre que quiser – no nosso exemplo, você o executaria todos os dias. Repare que calcular a média uma única vez não seria um problema; no entanto, esse procedimento simples se torna custoso quando temos que repetí-lo todo dia. Ao automatizar a tarefa, o pesquisador pode concentrar o tempo economizado no cálculo para, por exemplo, pensar sobre o porquê da média ter apresentado aquele determinado valor.

1.3 Vamos programar... em R!

Em suma, aprender a programar oferece uma série de vantagens tangíveis para quem trabalha com estatística e matemática. Ela tornar o trabalho mais eficiente e produtivo, permitindo que os profissionais explorem dados de maneiras antes inimagináveis e desenvolvam soluções personalizadas para os desafios enfrentados em suas áreas de atuação.

No restante do material, aprenderemos a programar utilizando a *linguagem de programação R*. Em outras palavras, aprenderemos (i) sua *sintaxe*, isto é, a forma correta de escrever comandos para que a máquina seja capaz de interpretar o que queremos como resultado, e (ii) como executar códigos que foram escritos.

Você deve estar se perguntando: *"Por que programar em R?"*. Simplesmente pois é uma linguagem de programação moldada para executar tarefas estatísticas (e matemáticas, em

menor grau). Não à toa, é amplamente utilizada por economistas que desejam constuir e testar hipóteses sobre seus modelos e automatizar coleta de dados econômicos.

⚠ Ao longo da apostila, existem conteúdos que são opcionais, ou seja, você pode pulá-los sem prejuízo para a sequência do material. Pense neles como se fossem aprofundamentos. Sem tempo? Pode passar direto! Eles estarão dispostos em boxes expandíveis, como abaixo.

💡 Outros exemplos de linguagem de programação (Opcional)

O R é a linguagem que escolhemos para utilizar nos nossos cursos. Mas é interessante que você saiba que existem *outras* linguagens de programação por aí – algumas capazes de executar papel semelhante ao próprio R. Em 2023, as linguagens de programação mais utilizadas foram: JavaScript (63% dos programadores que responderam uma pesquisa disseram que a utilizavam[1]), Python (49%) e SQL (48%).
Apenas 4% dos respondentes tiveram contato com R. 'Meu Deus, estamos usando uma linguagem que poucas pessoas usam!' você pode estar pensando agora. Calma, não é bem assim! Realmente, em comparação ao Python e SQL, o R é menos utilizado. Mas, em número absoluto, a comunidade é grande. Além disso, é bom ter em mente que *cada linguagem de programação tem suas vantagens e desvantagens* e, por esse motivo, é preciso *entender o contexto no qual você quer utilizá-la.* O R foi escolhido por sua disponibilidade e eficiência computacional em relação à métodos estatísticos e matemáticos – ideal para cursos de estatística e matemática!

[1]Fonte: Statista

Parte I

Instalação

Nessa parte, você irá aprender como baixar e instalar o R e o RStudio, além da composição de *layout* de ambos. Construímos cada seção de instalação como um guia do tipo *passo a passo*, de maneira que você precisa apenas seguí-los de forma direta. Neste capítulo, é importante que você já comece a explorar um pouco a interface dos ambientes de programação que te mostraremos.

2 Instalando o R

Nesse capítulo, iremos aprender como baixar e instalar o R para Windows[1]! Optamos por dividir o passo a passo em 7 etapas – mas fique tranquilo, não são passos grandes, apenas fizemos dessa forma para que o conteúdo fique bem *mastigado*, fácil de entender.

Alguns conceitos iniciais (Opcional)

Antes de começar, vamos entender alguns conceitos. A ideia aqui é te ensinar o que significam algumas nomenclaturas e siglas que aparecem ao longo do processo de instalação, em especial *R Foundation* e *CRAN*. Essa parte é totalmente *opcional* e você pode pular direto para o passo a passo caso esteja sem tempo – ou até mesmo interesse.

- **R Foundation**: é uma empresa sem fins lucrativos, criada pelos principais desenvolvedores da linguagem. Quais são seus objetivos? Basicamente três: (i) administrar os direitos autorais da linguagem – e, por consequência, manter seu uso como livre; (ii) apoiar o desenvolvimento do R como um todo, isto é, fornecer informações e criar novos usos básicos, elaborar conferências, guias, entre outros; (iii) servir como ponto focal para todos os usuários da linguagem que desejem interagir com a comunidade de desenvolvedores. De forma resumida, a R Foundation é como se fosse a instituição provedora do básico da linguagem, que busca sempre atualizar e mantê-lo de pé. Se você instala o R e, logo em seguida, percebe que alguma de suas atribuições não está em perfeito funcionamento, provavelmente terá que comunicar à essas pessoas. Grosso modo, exerce um papel próximo ao da Microsoft com o Excel, por exemplo. Uma observação (importante): como o R é um software livre, qualquer pessoa pode desenvolver novas funções ou recursos a partir da linguagem. Por esse motivo, para recursos que estejam além da *base do R*, você deve recorrer à quem os criou! Por exemplo, com relação ao RStudio (que conheceremos mais à frente), devemos nos reportar à empresa Posit, sua desenvolvedora. Na prática, raramente (para não dizer nunca) iremos reportar alguma coisa à R Foundation, mas sim aos desenvolvedores daquele pacote/extensão específico (fique tranquilo, explicaremos mais à frente o conceito de *pacote* para a linguagem).

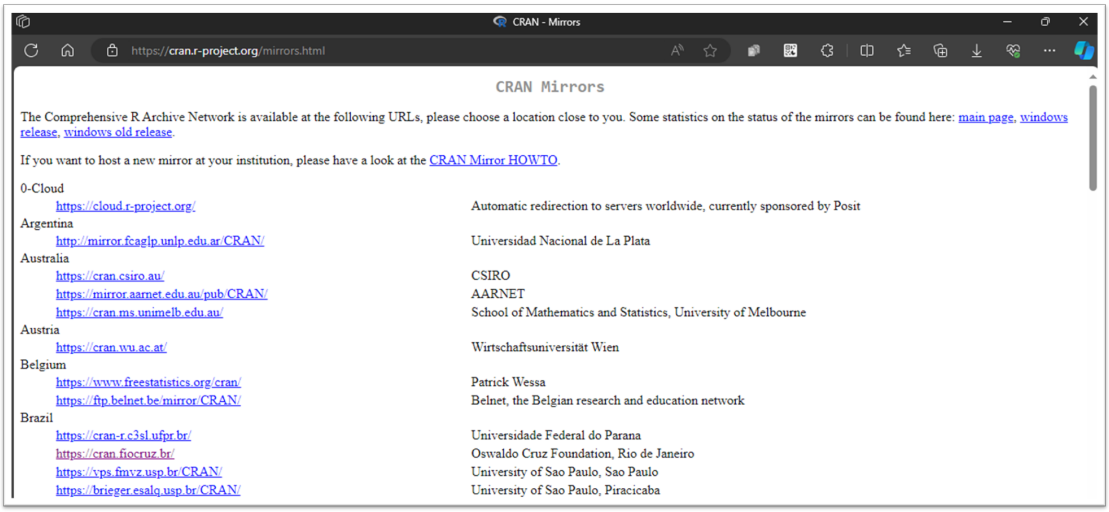
[1] Você pode realizar procedimento equivalente para sistemas operacionais Linux, apenas alterando a opção de *download* quando necessário – isto é, selecionando as opções em que esteja escrito 'Linux', ao invés de 'Windows'.

- **CRAN** (Comprehensive R Archive Network): segundo o própio, é "*uma coleção de sites que carrega material idêntico, consistindo nas distribuições do R, extensões contriubídas, documentação e arquivos binários de R*". *'Meu Deus, o que isso significa?'* Simples: apenas uma coleção de endereços da internet em que podemos baixar a versão mais recente do R, assim como pacotes. Quem mantém o CRAN? Instituições voluntárias; em seus sites, a parte onde é possível baixar arquivos relacionados ao R é chamada de *mirror*. E com quais recursos o CRAN se mantém? Com os da própria instituição participante (principalmente em termos de colaboradores) e, também, da R Foundation!

Essa história toda para dizer: **o arquivo básico que iremos baixar para instalar o R será obtido através de algum *mirror* do CRAN, isto é, a parte do site de alguma instituição voluntária em colaboração com a R Foundation.**

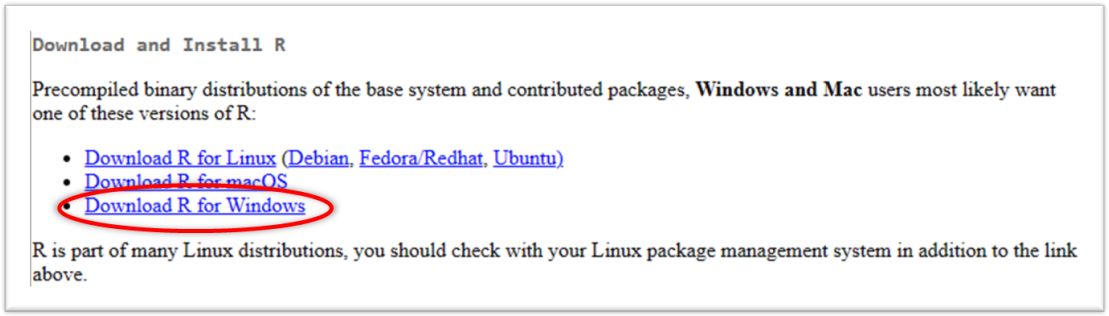
2.1 Sete passos

1. O primeiro passo consiste em escolher um repositório (*mirror)* para baixar o R. No endereço https://cran.r-project.org/mirrors.html encontramos todas as opções disponíveis, por país e em ordem alfabética. No seu computador, deverá aparecer a seguinte tela:

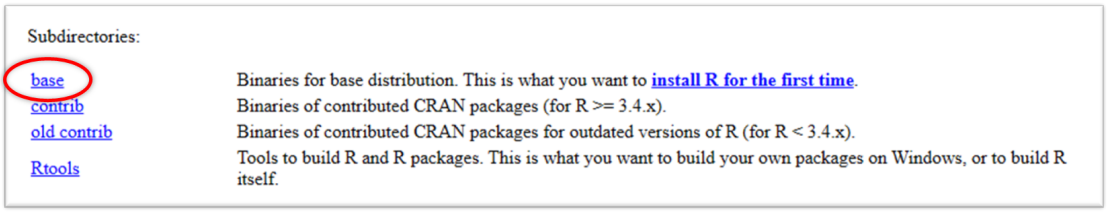


2. Por questões de rapidez/latência, o ideal é escolher o repositório mais próximo de você. Considerando que todos estejam no Rio de Janeiro, vamos então utilizar o *mirror* da Fiocruz.

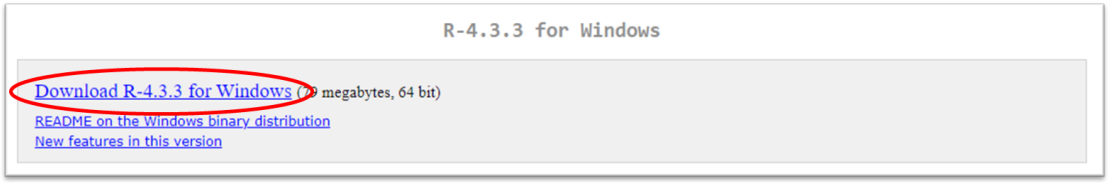
3. Como essa apostila foca na instalação para sistemas operacionais do tipo Windows, vamos clicar então em *Download R for Windows*, na parte superior da página.



4. Na página seguinte, clique em 'base'. Grosso modo, como o nome já indica, iremos baixar os aquivos *base* do R – ou seja, o mínimo necessário que você precisará para poder executar algum código.



5. Na nova página, clique em '*Download R x.x.x for Windows*', sendo 'x.x.x' o número da versão que será baixada. No momento da elaboração deste tutorial, a versão mais recente do R é a 4.3.3.



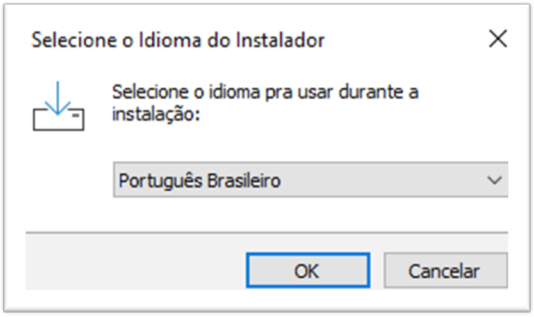
Se você tiver algum problema com o *download*, tente escolher outro servidor no passo 2 – por exemplo, um dos servidores da Universidade de São Paulo.

6. Você receberá um aviso, que varia conforme o navegador em uso, de que o arquivo está sendo baixado. Abaixo, um exemplo no Microsoft Edge:

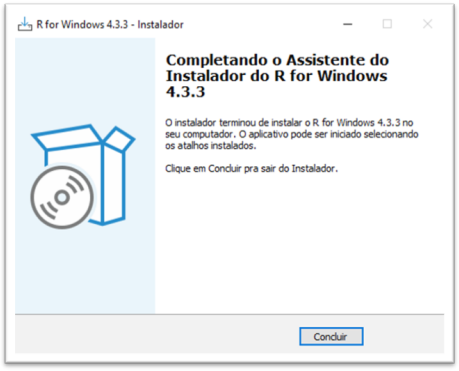
   No Windows, o arquivo será armazenado na pasta 'Downloads' do seu computador (ou na pasta que você previamente configurou como destino para os arquivos baixados).

7. Feito o download, clique duas vezes no arquivo baixado e siga as instruções para instalação. Na prática, basta clicar em 'Avançar' até o fim.

início da instalação

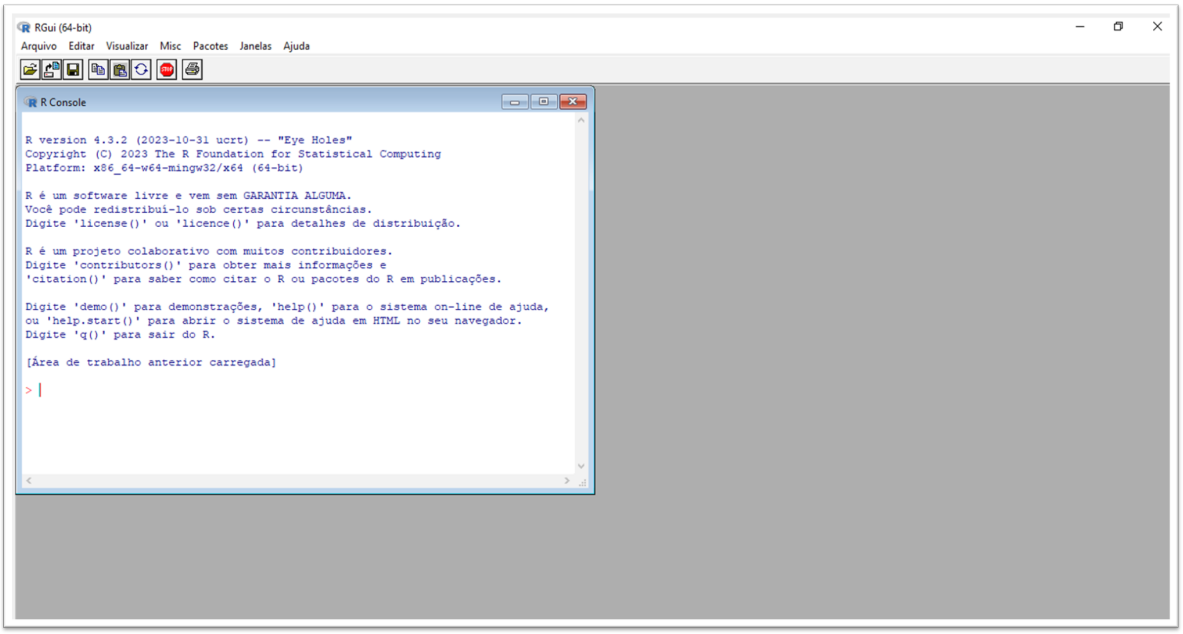


final da instalação



Após o final da instalação, você deverá ser capaz de encontrar e abrir no seu computador o **R Graphical User Interface** ou, como popularmente é conhecido, **RGui**. Ele estará na pasta em que você destinou para instalação; no Windows, algo próximo de:

`C:\ProgramData\Microsoft\Windows\Start Menu\Programs\R`



2.2 Conhecendo o RGui

De forma geral, um GUI permite com que o usuário utilize a linguagem de forma interativa através de botões e dispositivos visuais. Observe que, na parte superior, temos oito botões principais, representados por pequenas imagens.

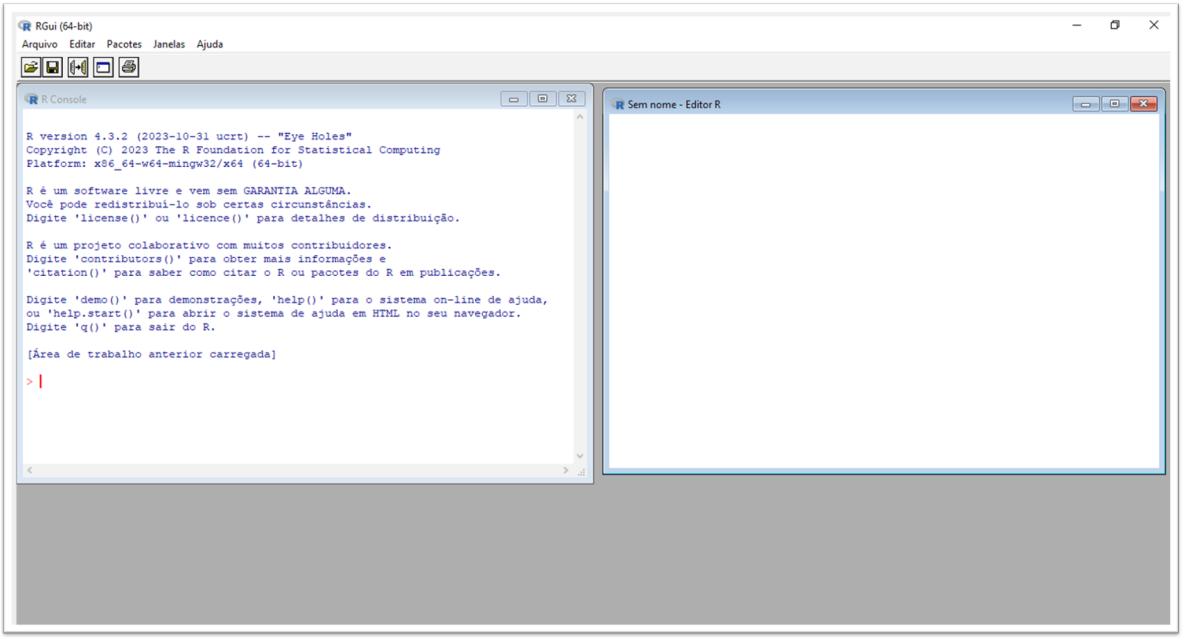
Cada botão executa uma tarefa específica. Os três primeiros, da esquerda para direita, são os mais relevantes:

- 'Abrir script': permite com que você carregue, no Editor de Código, um arquivo que contém linhas de código (script). Arquivos desse tipo, cuja extensão é `.R`, serão os mais importantes da linguagem.
- 'Carregar área de trabalho': *importa* objetos que foram salvos anteriormente em um arquivo do tipo `.RData`.

- 'Salvar área de trabalho': salva objetos criados em um arquivo do tipo `.RData`.

Os botões restantes, em ordem, executam as seguintes tarefas: 'Copiar', 'Colar', 'Copiar e colar', 'Parar computação atual' e 'Imprimir'. Nesse momento, não se preocupe em saber o que significa *importar* ou o que é um arquivo do tipo *.RData*.

Por outro lado, vamos procurar entender melhor o que são o **Console** e o **Editor de Código**. O primeiro corresponde à janela de nome *R Console,* no canto esquerdo da sua tela. Este último, por sua vez, não abre instantâneamente no momento em que você acessa o RGui, mas podemos abrí-lo manualmente através de 'Arquivo' > 'Novo script' – ou, então, carregando um script já existente através do botão 'Abrir script', que vimos anteriormente. Posicionando o Editor de Código ao lado do Console, teremos a seguinte imagem:



Por quê esses espaços terão relevância para nós?

- O Editor de Código é o local em que você escreve os comandos que deseja executar no R, além de comentários que busquem registrar o porquê de você ter escrito determinada parte do seu código. Na prática, um *comentário* é uma linha que não será interpretada – e consequentemente executada – como parte da linguagem. Para registrar um comentário, basta escrever o símbolo '#' antes do que você deseja escrever naquela linha[2]. Um ponto importante: o Editor permite com que salvemos o script que criamos em um arquivo do tipo `.R`. Lembre-se: esse é o principal tipo de arquivo da linguagem.

[2]Note que, se o seu comentário for longo demais, de tal forma que você queira quebrá-lo em duas ou mais linhas, será necessário novamente escrever '#' na próxima linha

- O Console, por sua vez, é o local em que a parte interpretável de código em R (ou seja, tudo exceto comentários) será *efetivamente* executada e os respectivos resultados serão mostrados. É aqui que a mágica efetivamente ocorre! Você também pode executar partes do seu código diretamente no Console, porém os comandos não ficam salvos, são apenas temporários.

Simplificando: **o Editor é o espaço em que você realmente escreverá os códigos em R**. Ele atua como rascunho do seu script, permitindo com que você posteriormente salve o que foi escrito e, consequentemente, volte a executar o mesmo código. Já **o Console é o espaço em que o código é processado, retornando com o resultado dos comandos que você escreveu**.

Entretanto, não iremos utilizá-los através do RGui. No capítulo seguinte, instalaremos e conheceremos um pouco mais sobre outro ambiente, bem mais completo, para se programar em R. *"Meu Deus, aprendi todos esses conceitos à toa?"*, você deve estar se perguntando. Não! Muito do que aprendemos nessa seção voltará a aparecer no capítulo seguinte.

3 Instalando o RStudio

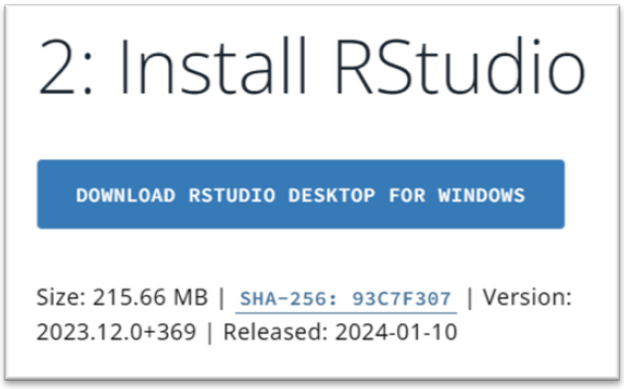
Acontece que o RGui não é tão prático de se usar. Pensando nisso, a empresa Posit criou um Ambiente de Desenvolvimento Integrado (*Integrated Development Environment*, IDE) chamado **RStudio**. Em nosso contexto, tanto GUI quanto IDE são ferramentas que permitem a utilização da linguagem. A diferença é que a IDE tem atributos com a finalidade de facilitar o desenvolvimento dos códigos. Grosso modo, toda IDE é uma GUI mas o inverso não é verdadeiro (nem toda GUI é uma IDE).

Em resumo: **é muito mais fácil utilizar o R através do RStudio e, por este motivo, vamos baixá-lo na sua versão gratuita** (que já é suficiente para os cursos que serão ministrados no Instituto).

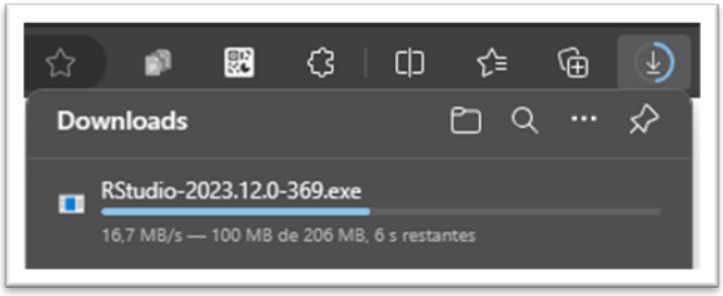
3.1 Três passos

Para instalar o RStudio no Windows, novamente iremos seguir alguns passos – nesse caso, apenas 3:

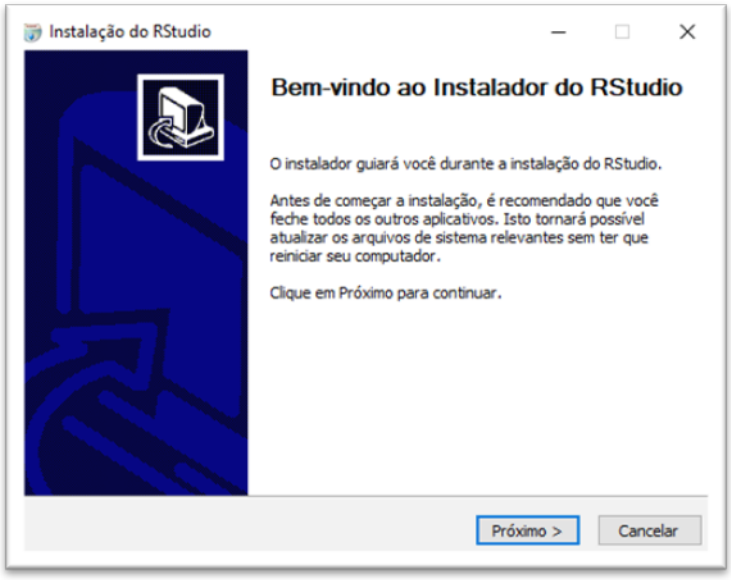
1. Acesse a página de downloads da RStudio: https://posit.co/download/rstudio-desktop/#download. Se você tiver acesso de administrador, basta clicar em *'Download RStudio Desktop for Windows'*.



2. De forma análoga ao *download* do R, você receberá um aviso de que o arquivo está sendo baixado (na sua pasta de 'Downloads' ou similar).

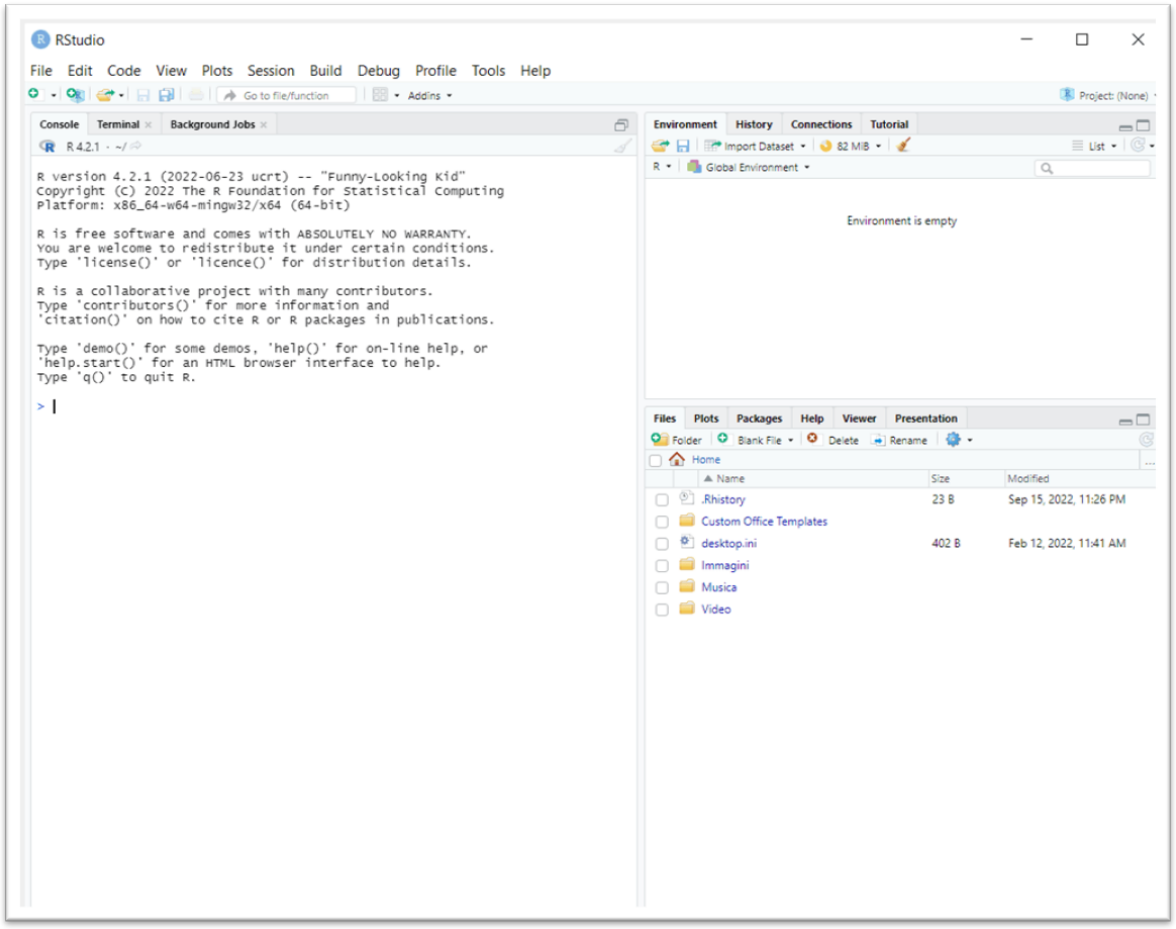


3. Clique duas vezes no arquivo que você baixou e siga as instruções recomendadas de instalação, cuja tela inicial está na imagem abaixo.



Ao final da instalação, você deverá ser capaz de abrir o RStudio no seu computador, resultando em algo similar à imagem abaixo. No Windows, provavelmente você o encontrará no caminho:

`C:\ProgramData\Microsoft\Windows\Start Menu\Programs\RStudio`



Feito? Então estamos prontos para utilizar o R através do RStudio!

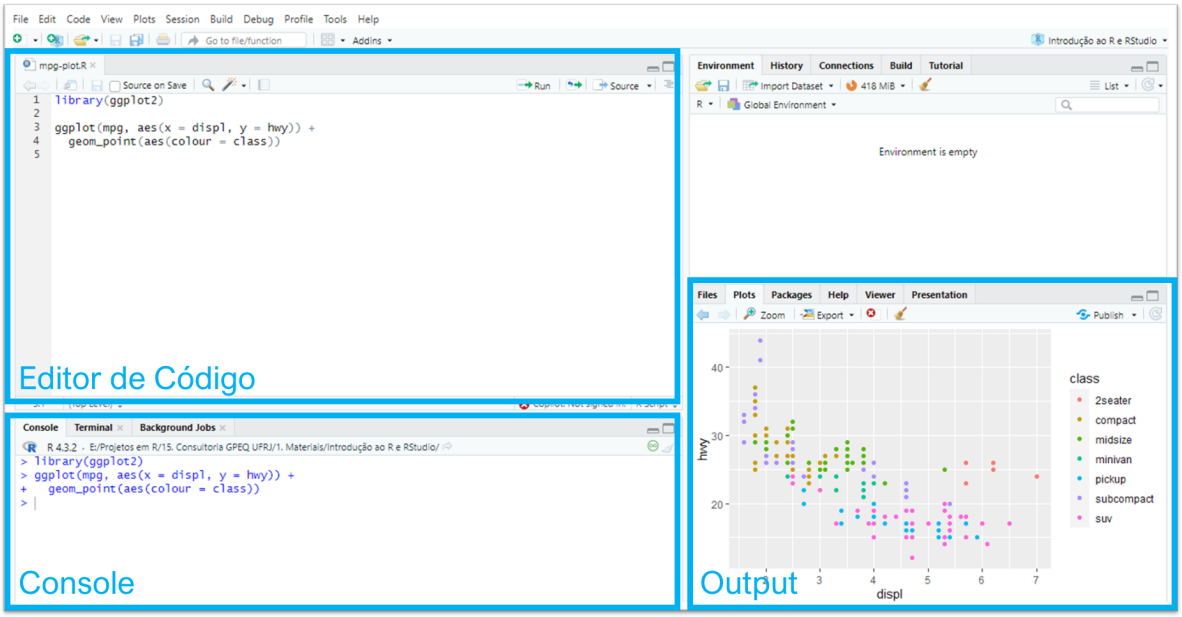
3.2 Conhecendo o RStudio

> **Nota**
>
> A seção 3.2 'Conhecendo o RStudio' é baseada na seção 2.1 'Telas' do livro *Ciência de Dados em R*, feito pelo Curso-R. De qualquer modo, eventuais erros são inteiramente de nossa responsabilidade.

O RStudio será o ambiente no qual iremos trabalhar com a linguagem. Por essa razão, é *muito* importante que você se sinta confortável com o que verá no seu computador após abrí-lo. Nessa seção, iremos compreender melhor o *layout* do RStudio, além das utilidades que ele nos proporciona ao longo do processo de escrita dos códigos.

Ao abrir o RStudio pela primeira vez (como na imagem anterior), você verá inicialmente 3 quadrantes. Um deles, preenchendo a parte esquerda da tela, já conhecemos: é o **Console**, que cumpre o mesmo papel explicado no capítulo anterior. Ao mesmo tempo, o quadrante que mais utilizaremos não aparece inicialmente: é o **Editor de Código**, outro velho conhecido que também possui a mesma atribuição anterior. Tal como no caso do RGui, o Editor não abre automaticamente pois o RStudio não é capaz de saber se o usuário tem o desejo de construir um código do zero – ou seja, criar um novo arquivo com extensão *.R* – ou apenas dar continuidade à algum em que já estava trabalhando.

No fim das contas, teremos 4 quadrantes:



> Por padrão, os quadrantes estarão dispostos na sua tela da forma como mostramos na imagem acima, mas você pode organizá-los da forma que preferir acessando a seção *Pane Layout* da opção `Global options...` no menu `Tools`.

É importante que você entenda que o Editor e o Console são os dois principais quadrantes do RStudio. Passaremos a maior parte do tempo neles. Como não custa nada, vamos relembrar suas respectivas serventias:

- **Editor de Código**: é local em que escreveremos/editaremos nossos códigos, salvando posteriormente em um arquivo do tipo `.R`. Conforme formos avançando, você acabará reparando que temos algumas melhorias em relação ao RGui:

    1. O RStudio colore algumas palavras e símbolos para *facilitar* a leitura do código. Por exemplo, tudo o que for comentário será de uma determinada cor, assim como

tudo que você escrever entre aspas – considerado texto passível de ser executado como parte de um código – será de outra.
2. Outra funcionalidade interessante do Editor no RStudio é a capacidade de você poder buscar e substituir determinadas palavras/expressões que estejam presentes no código, poupando tempo e evitando erros caso o fizessemos de forma maunal; para tal, basta clicar no símbolo da lupa logo acima da primeira linha.
3. Além disso, o RStudio possui o recurso de autocompletar partes de um código! Caso você esteja escrevendo o nome de um *objeto* que ele consiga identificar, receberá automaticamente uma sugestão para completar a escrita, bastando apertar a tecla `Tab` para aceitá-la.

- **Console**: é local em que o código é executado e recebemos as saídas. Nele, temos também o recurso de autocompletar nomes de objetos. Para *limpar* o Console, isto é, excluir o registro do que já foi executado pelo R, basta clicar no símbolo de vassoura, no canto direito superior do quadrante, ou então utilizar o atalho `Ctrl + L`.

Os demais quadrantes do lado *direito* contém painéis auxiliares. O objetivo deles é facilitar pequenas tarefas que fazem parte tanto da programação quanto da análise de dados como, por exemplo, olhar a documentação de funções, analisar os objetos criados em uma sessão do R, procurar e organizar os arquivos que compõem a nossa análise, armazenar e analisar os gráficos criados e muito mais.

No quadrante *superior*, temos

- **Environment**: painel com todos os objetos criados na sessão. Será bastante útil como referência para avaliar os objetos que criamos ou deixamos de criar com determinado comando.
- **History**: painel com um histórico dos comandos rodados.

Já no quadrante *inferior*, temos

- **Files**: mostra os arquivos no diretório de trabalho. Nele, é possível navegar entre as pastas do seu computador! Você pode, por exemplo, abrir um arquivo do tipo `.R` sem necessariamente ter que passar pela janela de busca do seu sistema operacional.
- **Plots**: painel onde os gráficos serão apresentados, caso você crie um código que os produza.
- **Packages**: apresenta todos os pacotes instalados e carregados.
- **Help**: janela onde as documentações de funções serão apresentadas.
- **Viewer**: painel onde relatórios e dashboards serão apresentados.

Além do Console e do Editor, dê atenção especial aos painéis Environment, Help e Plots, nesta ordem.

Parte II

R Básico

Nessa parte do material, você aprenderá a programar na prática. O capítulo inicial terá como objetivo estimular que a escrita de suas primeiras linhas de código; será composto de tarefas *super* simples, mas suficientes para proporcionar uma primeira eperiência à quem nunca programou. Nos dois capítulos seguintes, te guiaremos no entendimento sobre os dois conceitos mais importantes da linguagem: *objetos* e *funções*. Segundo John Chambers, um dos desenvolvedores do R,

> *to understand computations in R, two slogans are helpful:*
>
> - *Everything that exists is an object.*
> - *Everything that happens is a function call.*

Mão na massa!

4 Primeiros passos

> Partes deste capítulo são baseadas na seção 3.2 'R como calculadora' do livro *Ciência de Dados em R*, feito pelo Curso-R. De qualquer modo, eventuais erros são inteiramente de nossa responsabilidade.

Como vimos nos capítulos anteriores, o papel do **Console** no R é interpretar os nossos comandos à luz da linguagem. Ele avalia o código que o passamos e devolve a saída correspondente – se tudo der certo – ou uma mensagem de erro – se o seu código tiver algum problema. Essa operação é chamada de **avaliar**, **executar** ou **rodar** o código. Para que seu código seja executado diretamente no Console, escreva-o e, na sequência, aperte `Enter`. A outra forma de executar uma expressão é escrevê-la em um *script* no **Editor**, selecioná-la (ou apenas clicar na linha em que está) e usar o atalho `Ctrl + Enter`. Assim, o comando é enviado para o Console, onde é diretamente executado.

Nesse capítulo, te mostraremos como o processo funciona! Você *rodará* suas primeiras linhas de código, ordenando ao R que realize operações aritméticas como *adição*, *subtração*, *multiplicação* e *divisão*, além de comparações lógicas simples. O objetivo aqui não é ensinar matemática básica, mas te proporcionar a primeira experiência prática com o R. É o jeito mais fácil de um iniciante se familiarizar com a linguagem, automaticamente se preparando melhor para a execução de linhas de código mais avançadas.

4.1 Operadores Aritméticos

> **i** De agora em diante, cada região sombreada de cinza e centralizada no documento representa um pedaço código. Quando couber, o resultado de sua execução no Console será exposto logo na sequência. Observe que você pode copiar o que está escrito ao passar o cursor sobre a região e clicar no símbolo de prancheta que aparecerá no canto direito superior.

Vamos começar com um exemplo simples! Vamos pedir ao R que some os números `1` e `1`:

```
1 + 1
```

```
[1] 2
```

Nesse caso, o nosso comando foi o código `1 + 1` e a saída foi o valor `2`. Como você pode reproduzir esse comando no RStudio? Inicialmente, copie o que está escrito acima ao clicar no símbolo de prancheta no canto superior direito da região sombreada. Na sequência, cole no Editor de Código e aperte `Ctrl + Enter` (ou então cole no Console, na sequência pressionando apenas `Enter`).

Tente agora jogar no Console a expressão: `2 * 2 - (4 + 4) / 2`. Deu zero? Pronto! Você já é capaz de pedir ao R para fazer *qualquer uma das quatro operações aritméticas básicas*. Repare que as operações e suas precedências são mantidas como na matemática, ou seja, divisão e multiplicação são calculadas antes da adição e subtração, além de os parênteses ditarem a ordem na qual serão realizadas. A seguir, apresentamos a Tabela 4.1 resumindo como fazer as principais operações no R.

Tabela 4.1: Operadores matemáticos no R

Operação	Operador	Exemplo	Resultado
Adição	+	1 + 1	2.00
Subtração	-	4 - 2	2.00
Multiplicação	*	2 * 3	6.00
Divisão	/	5 / 3	1.67
Potenciação	^	4 ^ 2	16.00
Resto da Divisão	%%	5 %% 3	2.00
Parte Inteira da Divisão	%/%	5 %/% 3	1.00

Incluímos os operadores de potênciação, resto e parte inteira de divisão. Na prática, não serão tão utilizados quanto os demais, mas é importante que você os veja pelo menos uma vez durante o processo de aprendizado.

4.2 Operadores Lógicos

O R permite também testar comparações lógicas. Os valores lógicos básicos em R são `TRUE` (ou apenas `T`) e `FALSE` (ou apenas `F`). Por exemplo, podemos pedir ao R que nos diga se é verdadeiro que 5 é menor do que 3. Como a resposta é obviamente negativa, ele retornará `FALSE`, nos dizendo que a proposição que fizemos é falsa.

```
5 < 3
```

```
[1] FALSE
```

Abaixo, introduzimos a Tabela 4.2 com outros operadores lógicos da linguagem.

Tabela 4.2: Operadores lógicos do R

Operação	Operador	Exemplo	Resultado
Maior que	$>$	$2 > 1$	TRUE
Maior ou igual que	$>=$	$2 >= 2$	TRUE
Menor que	$<$	$2 < 3$	TRUE
Menor ou igual que	$<=$	$5 <= 3$	FALSE
Igual à	$==$	$4 == 4$	TRUE
Diferente de	$!=$	$5 != 3$	TRUE
x **e** y	&	x <- c(1, 4, NA, 8) x[!is.na(x) & x > 5]	8
x **ou** y	|	x <- c(1, 4, NA, 8) x[!is.na(x) | x > 5]	1, 4, 8

4.3 Possíveis complicações

Em determinado momento, você pode acabar executando errado algum trecho de código. Como o R se comporta nessas situações?

4.3.1 Comando incompleto

Se você digitar um comando incompleto no Console, como `5 +`, e apertar `Enter`, o R mostrará um `+`, o que não tem nada a ver com a adição da matemática. Isso significa que o R está esperando você enviar **mais** algum código para completar o seu comando. Termine o seu comando ou aperte `Esc` para recomeçar.

```
5 -
+
+ 5
```

```
[1] 0
```

4.3.2 Comando inexistente na linguagem

Se você digitar um comando que o R não reconhece, ele retornará uma mensagem de erro. **Não entre em pânico.** Ele só está te avisando que não conseguiu interpretar o comando.

Imagine que queremos retornar a parte inteira da divisão de `5` por `2`. Nesse caso, devemos digitar `5 %% 2` no Editor e rodar. Mas, se por engano, digitarmos `5 % 2`?

```
5 % 2
```

```
Error: <text>:1:3: unexpected input
1: 5 % 2
      ^
```

Não existe um operador `%` no R, logo esse comando retorna erro! Você pode digitar o comando correto normalmente em seguida.

```
5 %% 2
```

```
[1] 1
```

5 Objetos

Na apresentação dessa parte do material, trouxemos uma citação que, em parte, dizia:

> ***Everything that exists is an object.***

Um objeto é simplesmente um nome que guarda um valor ou código. Não há como ser mais direto: tudo que existe no R é um *objeto, inclusive as funções que veremos no capítulo seguinte*. Nesse capítulo, veremos com detalhe os objetos que são designados à *armazenar dados*. Antes, no entanto, vamos dar um passo para trás e explicar o que são dados.

5.1 Dados

Segundo a Oxford Languages, dados são

> fatos e estatísticas coletadas de forma conjunta para referência ou análise.

Na prática, dados nos mostram informações sobre determinado indivíduo ou situação que procuramos descrever, seja uma pessoa, instituição, comportamento, condição geográfica, etc. O número de horas que você dormiu essa noite é um dado. A lista que relata quem é ou não calvo na sua família é uma *coleção* de dados. A expectativa, hoje, de quanto será a inflação acumulada nos próximos 12 meses é um dado. A variação percentual do Produto Interno Bruto (PIB) real no último trimestre é um dado. A lista que mostra a sequência de variações do PIB real nos últimos dez trimestres é uma *série temporal*, isto é, dados em sequência ao longo do tempo.

5.1.1 Tipo & Forma

Vamos nos aprofundar um pouco mais. Ao lidar formalmente com dados, **devemos ter mente que eles são compostos por uma ou mais variáveis e seus valores**. *Uma variável é uma dimensão ou propriedade que descreve uma unidade de observação* (por exemplo, uma pessoa) e normalmente pode assumir valores diferentes. Por outro lado, os *valores são as instâncias concretas que uma variável atribui a cada unidade de observação e são ainda caracterizados por seu intervalo* (por exemplo, valores categóricos versus valores contínuos) *e seu tipo* (por

exemplo, valores lógicos, numéricos ou de caracteres). Estaremos interessados no *tipo* dos dados. A Tabela 5.1 apresenta os que podem aparecer com maior frequência.

Tabela 5.1: Tipos mais comuns de dados

Tipo	Serve para representar...	Exemplo
Númerico	números do tipo *integer* (inteiro) ou *double* (reais)	1, 3.2, 0.89
Texto *(string)*	caracteres (letras, palavras ou setenças)	"Ana jogou bola"
Lógico	valores verdade do tipo lógico (valores booleanos)	TRUE, FALSE, NA
Tempo	datas e horas	14/04/1999

Voltando ao primeiro exemplo, uma pessoa pode ser descrita pelas variáveis *nome*, *número de horas dormidas* e *se dormiu ou não mais de oito horas*. Os valores correspondentes a essas variáveis seriam do tipo texto (por exemplo, "Pedro"), numéricos (número de horas) e lógicos (`TRUE` ou `FALSE`, definido em função do tempo descansado[1]). **Note a diferença entre *dado* e *valor*.** O número 10 é um valor, sem significado. Por outro lado, *"10 horas dormidas"* é um dado, caracterizado pelo valor 10 e pela variável *"horas dormidas"*.

Outro aspecto importante sobre os dados está em sua forma, ou seja, como os dados podem ser organizados. A Tabela 5.2 apresenta as formas mais comuns de organização.

Tabela 5.2: Formas pelas quais os dados podem ser organizados

Formato	Os dados se apresentam como...	Exemplo
Escalar	elementos individuais	"AB", 4, TRUE
Retangular	dados organizados em i linhas e j colunas	Vetores e Tabelas de Dados
Não-retangular	junção de uma ou mais estruturas de dados	Listas

Um escalar é um elemento único, que pode ser de qualquer tipo. Ou seja, a representação elementar de um dado se dá através de um escalar! Por exemplo, o tipo sanguíneo de determinada pessoa, representado pelos caracteres "AB", é um escalar do tipo texto. Você pode pensar no escalar como um dado organizado em 1 linha e 1 coluna.

Por sua vez, dados retangulares são àqueles cuja organização ocorre em i linhas e j colunas, tal que $i, j \in \mathbb{N}$ e $i > 1$ ou $j > 1$. As formas retangulares mais comuns são *vetores*, *matrizes* e *tabelas de dados*. Uma matriz é uma forma de organização de dados *númericos* em em i linhas

[1] Se o número de horas que a pessoa descansou for maior do que 8, então a variável deverá apresentar valor igual a `TRUE` – ou seja, é verdade que a pessoa dormiu mais de 8 horas. Caso contrário, `FALSE`.

e j colunas. Quando uma matriz possui i linhas e 1 coluna *ou* 1 linha e j colunas, chamamos de vetor-coluna e vetor-linha, respectivamente; em muitos casos, chamamos apenas de *vetor*. Assim, o vetor é um caso especial de matriz unidimensional. As tabelas de dados, por outro lado, possuem i linhas e j colunas, tal que $i > 1$ e $j > 1$. Além disso, aceitam todos os tipos de dado – por exemplo, númericos, de textos ou lógicos – em qualquer que seja a combinação de linha e coluna.

Por sua vez, dados não-retangulares se referem a toda organização de dados que não seja feita em linhas e colunas relacionadas entre si. A forma mais comum é a lista. Observe um exemplo abaixo: cada característica pode ser entendida como um elemento de uma lista. Apesar de pertencerem a mesma estrutura, os elementos não se comunicam entre si.

	Jorge	Laís	Matheus	Laura	Nathália
Gênero	Masculino	Feminino	Masculino	Feminino	Feminino
	Jorge	Laís	Matheus	Laura	Nathália
Idade	18	23	22	21	21
	Jorge	Laís	Matheus	Laura	Nathália
Altura (cm)	180	170	170	175	168
	Jorge	Laís	Matheus	Laura	Nathália
Peso (kg)	76	65	70	68	66

Nesse caso em específico, conseguimos fazer a transição para uma tabela (forma retangular) pois todos os elementos são são características das mesmas pessoas. Em uma tabela de dados, automaticamente temos uma relação entre os dados: cada linha contém características de uma unidade específica.

Tabela 5.4

Nome	Gênero	Idade	Altura (cm)	Peso (kg)
Jorge	Masculino	18	180	76
Laís	Feminino	23	170	65
Matheus	Masculino	22	175	70
Laura	Feminino	21	181	68
Nathália	Feminino	21	168	66

5.2 Estruturas de Dados no R

Na seção anterior, vimos os conceitos de *tipo* e *forma*. Tenha em mente que são duas definições que existem independentemente de qualquer linguagem de programação – elas versam sobre *dados* de forma geral.

Por outro lado, agora veremos o conceito e alguns exemplos de *estrutura de dados* para o R. **A estrutura de dados é a forma pela qual o R classificará um objeto em relação ao *tipo* e a *forma* dos dados que contém.** Existe uma estrutura de dados para cada combinação de tipo e forma? **Não.** Compreender as principais estruturas disponíveis no R requer vê-las como uma combinação de

(a) Determinado formato de dados
(b) Número de tipos de dados (se é único ou possui vários)

5.2.1 Criando e armazenando objetos na memória

Antes de conhecê-las, no entanto, vamos entender melhor os comandos para criar e armazenar *qualquer* objeto (seja ele para armazenar dados, como nesse capítulo, ou para criar funções, que serão vistas no próximo) na memória do R.

Para *criar* e *armazenar* um objeto, sempre escreveremos inicialmente seu nome (escolhido por você), seguido de um dos *operadores de atribuição* (ou *assingment operators*, como são conhecidos) e, por fim, o objeto propriamente dito com as informações de nosso interesse. O principal operador de atribuição para se criar objetos é `<-`. Outro operador que é comumente utilizado para cumprir a mesma tarefa é `=`. Ainda que exista uma leve diferença entre ambos, ao longo dos cursos será possível utilizar o operador de sua preferência. Por ser o ideal, utilizaremos `<-` no restante do material.

```
nome_do_objeto <- >objeto com informações<

nome do objeto =  >objeto com informações<
```

No parágrafo anterior, observe que está escrito *'criar e armazenar'*. Nós poderíamos simplesmente criar um objeto, sem armazená-lo na memória do R. Nesse caso, não teríamos o nome do objeto disponível na aba **Environment** (ou seja, ele não seria armazenado no nosso ambiente de trabalho) e seria bem mais complicado registrar todas as mudanças que viermos a fazer nele. Aconteceria apenas a ocorrência de uma única saída no Console com a estrutura do objeto criado (de forma semelhante ao que fizemos no capítulo anterior) – o quê não tem grande utilidade para nós, exceto caso você queira verificar a estrutura do objeto antes de realmente armazená-lo.

Ao mesmo tempo, você irá perceber que a *criação e armazenamento de um objeto não implica sua visualização imediata*. Isso signfinica que, ao dar o comando para criar algum objeto, não acontecerá nada no Console. Você pode acabar achando que o processo falhou ou algo do tipo, mas não é nada disso! Como dissemos, a mudança ocorrerá na aba Environment, onde deverá aparecer o nome do novo objeto. Para visualizar o objeto criado, escreva seu nome e rode a linha de código. Agora sim o objeto aparecerá no Console.

```
nome_do_objeto
```

Se você criou e armazenou um objeto na memória, ele ficará por lá até que você encerre sua sessão atual (feche o RStudio) ou, então, que o remova. Para remover qualquer objeto basta escrever `rm(nome_do_objeto)`. Caso tenha o desejo de remover vários objetos, basta separar seus nomes com vírgula. Para remover *todos* os objetos que aparecem na aba Environment, use `rm(list = ls())`.

```
rm(nome_do_objeto1)

rm(nome_do_objeto1, nome_do_objeto2, ...)

rm(list = ls())
```

5.2.2 Valor único

Ao criar um objeto com valor único, estamos armazenando um escalar que pode variar quanto ao tipo (por exemplo, númerico, *string* ou valor lógico). Nesse caso, a *estrutura* do objeto *será idêntica* ao *tipo* – o que faz sentido, afinal estamos falando de um objeto de uma única linha e coluna.

5.2.2.1 Numérico

Um objeto **númerico** contém apenas um número (por exemplo, 1, 2, 4.13, π, entre outros). Se quisessemos atribuir o valor numérico 5 à um objeto chamado `x`, como poderiamos fazer? Observe abaixo e replique no seu RStudio!

```
x <- 5
```

Note que o Console não retorna nenhuma mensagem ou valor. Como dissemos na seção anterior, a única diferença que você deve ser capaz de observar é no painel Environment, no quadrante superior direito do RStudio. O nome do novo objeto aparecerá lá. Para que você possa visualizar o conteúdo do objeto criado, terá que escrever apenas seu nome e rodar a linha de código!

```
x
```

```
[1] 5
```

5.2.2.2 Textual

Uma **sequência de caracteres** (*character string*, ou apenas *string*) é um conjunto de caracteres dentro de um par de aspas e pode ou não incluir espaços. Por exemplo, "elevada" e "pressão arterial elevada" são objetos de caracteres com um único valor de *string*.

```
y <- "pressão arterial elevada"
y
```

```
[1] "pressão arterial elevada"
```

5.2.3 Vetor

Um **vetor** contém uma coleção ordenada de dados indexados pelos inteiros $1, 2, ..., n$, onde n é o comprimento do vetor. *O vetor como estrutura de dados é a combinação da forma vetor com dados de um único tipo, não necessariamente numérico.* No exemplo abaixo, um vetor númerico, isto é, que contém apenas números.

```
z <- c(5, 8, 12)
z
```

```
[1]  5  8 12
```

Se você tentar criar um vetor contendo dois tipos de dados diferentes, ele converterá todos os dados para o tipo texto. A única exceção é com relação a entradas do tipo lógico `NA` *(Not Available)*, que representam a ausência de determinado dado.

```
z <- c(5, "texto", TRUE, NA)
z
```

```
[1] "5"     "texto" "TRUE"  NA
```

É inegável que, a partir deste ponto, as estruturas começam a ficar mais interessantes. Lembre-se que o vetor tem uma dimensão e pode ter muitas informações armazenadas. É natural que, em determinadas situações, desejemos acessar apenas valores específicos dentre os que constam nele.

Como podemos fazer isso? Note que podemos um número associar a cada elemento de um vetor, representando a linha ou coluna em que consta. A esse número chamamos de *índice*. Dessa forma, fica fácil acessar qualquer um de seus valores: basta escrever, ao lado de seu nome e entre colchetes, o índice que está associado à este valor. Por exemplo, vamos acessar o segundo elemento do vetor `z` que acabamos de criar.

```
z[2] # Acessando o segundo elemento do vetor z
```

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

Você também acessar mais de um valor específico.

```
z[2:4] # Acessando do segundo ao quarto elemento do vetor z
```

```
[1] "texto" "TRUE"  NA
```

Algumas formas de criar vetores padronizados

Em determinadas situações, você pode ter o desejo de criar um vetor que tenha certo padrão. Por exemplo, se quiséssemos criar um vetor contendo todos os números inteiros de 1 a 7, como faríamos? Uma opção, de fato, é escrever `c(1,2,3,4,5,6,7)`. No entanto, essa abordagem não é prática quando a sequência for grande: imagine ter que escrever todos os inteiros de 1 a 100, por exemplo! Nessas ocasiões, podemos gerar o vetor desejado escrevendo os limites inferior e superior separados pelo operador `:`!

```
2:10                # Criando uma sequência de números de 2 a 10
```

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

Mas... e se quiséssemos selecionar uma sequência com mesmo intervalo, de tal maneira que o termo seguinte fosse igual ao termo anterior mais dois? Podemos usar a função `seq()`! Note que, tanto neste caso quanto no anterior, estamos falando da criação de vetores numéricos.

```
seq(2, 10, by = 2) # Criando um sequência de 2 a 10 pulando um número.
```

```
[1]  2  4  6  8 10
```

Por fim, e não menos importante, você ter o desejo de criar um vetor com valores repetidos – seja número, texto ou lógico. Para essa tarefa, utilizamos a função `rep()`!

```
rep(3, 5)          # Repetindo o número 3, 5 vezes
```

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

```
rep(c(1,"ab"), 3)  # Repetindo o vetor (1,"ab"), 3 vezes
```

```
[1] "1"  "ab" "1"  "ab" "1"  "ab"
```

5.2.3.1 Fator

Um **fator** é um tipo especial de vetor que contém valores numéricos subjacentes $1, 2, ..., n$, mas cada um desses n valores possui um rótulo de texto associado (que pode ou não ser o valor numérico). Esses valores rotulados são os **níveis** *(levels)* do fator. Um uso comum de um fator é armazenar uma variável categórica. **Depois de criar um vetor de fator com níveis específicos, nenhum elemento desse vetor poderá assumir um valor que não seja um de seus níveis pré-atribuídos.**

Você pode criar um fator a partir de um vetor de caracteres e o R assumirá que os valores únicos são os rótulos dos níveis. Por exemplo, no exemplo abaixo os níveis serão "lento", "normal" e "rápido".

```
y <- factor(c("super rápido", "super lento", "normal", "super rápido", "normal"))
y         # Rodar o vetor de fator também irá retornar os níveis
```

```
[1] super rápido super lento  normal       super rápido normal
Levels: normal super lento super rápido
```

Se quiser alterar os rótulos, você pode fazê-lo atribuindo um novo valor aos seus níveis. Por exemplo, suponha que queiramos que os rótulos sejam maiúsculos.

```
levels(y) <- c("Super Rápido", "Normal", "Super Lento")
levels(y)
```

```
[1] "Super Rápido" "Normal"       "Super Lento"
```

Como alternativa, você pode atribuir novos rótulos de nível ao criar o fator. Isso tem a vantagem adicional de permitir que você decida em que ordem os níveis devem aparecer. Quando criamos o fator, R atribuiu automaticamente os níveis pegando os valores exclusivos de y e colocando-os em ordem alfabética. Por vários motivos (como criar um gráfico de barras posteriormente), você pode querer que os níveis estejam em uma ordem diferente. **Você pode especificar a ordem dos níveis ao criar a variável, mas tome cuidado porque se você deixar de fora um valor que aparece nos dados esse valor acabará definido como ausente (`NA`).**

No exemplo anterior, gostaríamos que a ordem fosse da velocidade menor para o maior.

```
# Insira os valores originais nos níveis
# Insira os novos valores nos rótulos
# Tenha certeza que a ordem dos níveis e dos rótulos correspondem

y <- factor(c("super rápido", "super lento", "normal", "super rápido", "normal"),
            levels = c("super lento", "normal", "super rápido"),
            labels = c("Super Lento", "Normal", "Super Rápido"))
levels(y)
```

```
[1] "Super Lento"  "Normal"       "Super Rápido"
```

```
y
```

```
[1] Super Rápido Super Lento  Normal       Super Rápido Normal
Levels: Super Lento Normal Super Rápido
```

5.2.4 Matriz

Uma **matriz** contém uma coleção bidimensional de dados indexados por pares de inteiros (i, j). *A matriz como estrutura de dados é a combinação da forma matriz com dados de um único tipo, não necessariamente numérico.* Abaixo, uma matriz numérica.

```
x <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3)
x
```

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

Assim como os vetores, as matrizes não podem conter valores de diferentes tipos. Se você tentar criar uma matriz contendo valores numéricos e de caracteres, por exemplo, ela converterá os valores numéricos em caracteres. Note que você pode definir o número de linhas e colunas que uma matriz venha a possuir.

```
z <- matrix(c(1,2,"c","d","e","f"), nrow = 3, ncol = 2)
z
```

```
     [,1] [,2]
[1,] "1"  "d"
[2,] "2"  "e"
[3,] "c"  "f"
```

```
z <- matrix(c(1,2,"c","d","e","f"), nrow = 2, ncol = 3)
z
```

```
     [,1] [,2] [,3]
[1,] "1"  "c"  "e"
[2,] "2"  "d"  "f"
```

É importante ressaltar que, no R, uma matriz criada com i linhas e 1 coluna (ou 1 linha e j colunas) continua sendo interpretada como uma matriz, ao invés de ser interpretada como vetor-coluna (ou vetor-linha).

Como a matriz é um objeto de natureza bidimensional, podemos acessar seus elementos individuais através da inserção dos seus índices de linha *e* coluna. Por exemplo, para acessar o elemento presente na segunda linha e terceira coluna da matriz `z` que aramazenamos por último, rode:

```
z[2,3]
```

```
[1] "f"
```

Outra forma de acessar algum dado específico da matriz é pensá-la como sendo um único vetor, o qual vai sendo repartido conforme termina o tamanho que você pré-selecionou paras colunas. Dessa forma, podemos retornar determinado elemento pensando em seu índice de vetor. Por exemplo, poderíamos acessar o dado `f` pensando que é o sexto elemento do equivalente ao vetor `c("a","b","c","d","e","f")`.

```
z[6]
```

```
[1] "f"
```

Ao mesmo tempo, podemos acessar apenas uma coluna ou linha específica. Para tal, selecione a coluna ou linha que deseja retornar e deixe a coordenada restante como espaço vazio. No código abaixo, vamos selecionar inicialmente a primeira linha da matriz `z` e, na sequência, sua terceira coluna.

```
z[1,]
```

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

```
z[,3]
```

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

5.2.5 Data frame

Como matrizes (e vetores) contêm dados de apenas um tipo (por exemplo, todas as células são dados numéricos, de caracteres ou lógicos), precisamos de outra estrutura de dados para dados heterogêneos.

A necessidade de armazenar dados heterogêneos não é nada exótico ou incomum. Na verdade, mesmo os conjuntos de dados mais simples exigem a mistura de vários tipos de dados. Por exemplo, imagine que queremos armazenar um conjunto de dados que contém informações básicas sobre um grupo de pessoas, assim como na Tabela 5.4. Cada uma dessas cinco variáveis pode ser armazenada como um vetor (as duas primeiras do tipo caractere, as outras do tipo numérico). Para armazenar todas as cinco variáveis em uma única estrutura de dados, podemos combinar os cinco vetores em uma tabela retangular. As tabelas são a forma mais frequente de armazenar dados!

E qual o nome da estrutura de dados que armazena tabelas de dados no R? São os *data frames! O data frame como estrutura de dados é a combinação da forma tabela e da presença de qualquer tipo de dado.* No *chunk* abaixo, vamos recriar a Tabela 5.4 como exemplo.

```r
info_pessoas <- data.frame(Nome = c("Jorge", "Laís", "Matheus", "Laura", "Nathália"),
                           Gênero = c("Masculino", "Feminino", "Masculino", "Feminino", "Fem
                           Idade = c(18, 23, 22, 21, 21),
                           Altura = c(180, 170, 175, 181, 168),
                           Peso = c(76, 65, 70, 68, 66))

info_pessoas
```

```
      Nome    Gênero Idade Altura Peso
1    Jorge Masculino    18    180   76
2     Laís  Feminino    23    170   65
3  Matheus Masculino    22    175   70
4    Laura  Feminino    21    181   68
5 Nathália  Feminino    21    168   66
```

Você pode acessar qualquer dado específico de um *data frame* a partir do mesmo procedimento utilizado com matrizes. Por exemplo, para acessar o dado contido na segunda linha da primeira coluna, basta rodar `info_pessoas[2,1]`.

```r
info_pessoas[2,1]
```

```
[1] "Laís"
```

De forma semelhante, podemos acessar uma coluna ou linha específica.

```r
info_pessoas[,2] # Retornando dados da segundo coluna
```

```
[1] "Masculino" "Feminino"  "Masculino" "Feminino"  "Feminino"
```

```r
info_pessoas[1,] # Retornando dados da primeira linha
```

```
   Nome    Gênero Idade Altura Peso
1 Jorge Masculino    18    180   76
```

É interessante notar que um *data frame* pode ser pensado como a junção de múltiplos vetores-coluna, cada um representando determinada variável! Isso nos dá outra forma de selecionar colunas específicas: basta colocar entre colchetes o índice do vetor-coluna que você deseja selecionar. Note, no entanto, uma diferença: nesta *sintaxe*, o objeto que retornará ainda terá estrutura de um *data frame* (agora com cinco linhas e uma coluna) ao invés de vetor, como no *chunk* anterior.

```
info_pessoas[2]
```

```
     Gênero
1 Masculino
2  Feminino
3 Masculino
4  Feminino
5  Feminino
```

5.2.6 Lista

Uma lista contém uma coleção ordenada de objetos, sendo que estes podem ser de tipos diferentes. *A lista como estrutura de dados é a combinação da forma lista (representando dados não-retangulares) e da presença de qualquer tipo de dado.* Na prática, uma lista pode aceitar **qualquer** objeto de dados como elemento – inclusive uma outra lista!

Para que fique mais claro, abaixo está uma lista que contém um objeto com cada tipo de estrutura vista até agora! Colocamos nesta lista um valor único, um vetor, uma matriz e um data frame. Eles serão armazenados em um novo objeto, que nomeamos de `lista_exemplo`.

```
lista_exemplo <- list("5", c(1,2,3), matrix(c(2,2,3,4), 2, 2), info_pessoas)
```

Observe que a lista, apesar de poder contar com objetos de várias estruturas (e, consequentemente, dimensões), acaba por ter uma única dimensão. Você pode acessar seus elementos de forma *parecida* com o caso de um vetor. A diferença é que, no caso de uma lista, teremos que utilizar duplo colchetes. Abaixo, um exemplo de como acessar o quarto elemento da `lista_exemplo` – no caso, o *data frame* `info_pessoas` que criamos anteriormente.

```
lista_exemplo[[4]]
```

```
      Nome    Gênero Idade Altura Peso
1    Jorge Masculino    18    180   76
2     Laís  Feminino    23    170   65
3  Matheus Masculino    22    175   70
4    Laura  Feminino    21    181   68
5 Nathália  Feminino    21    168   66
```

Listas são mais úteis do que você pode estar pensando nesse momento. Elas permitem que você agrupe objetos de um mesmo assunto, mas com diferentes estruturas, em um único objeto 'central'. Em muitos casos, facilita a organização.

6 Funções e pacotes

A segunda parte da citação dizia:

> ***Everything that happens is a function call.***

Se algum objeto foi criado, armazenado ou transformado, tivemos a participação de uma *função*. Dessa forma, podemos dizer que o R é uma linguagem de programação *funcional*, ou seja, ocorre através da execução de funções. Na prática, quase todos os comandos que iremos realizar tem como base uma função. Tenha em mente que uma função também é um objeto, assim como tudo que *existe* no R. Nesse capítulo, iremos aprender o que são funções no contexto da linguagem R, além de suas utilidades, como escrevê-las e, por fim, como utilizá-las.

6.1 O que é uma função?

Antes, vamos pensar em funções no contexto matemático. Segundo Stewart (2015),

> Uma função é uma regra que atribui, para cada elemento x em um conjunto A, exatamente um elemento, chamado $f(x)$, em um conjunto B.

Em que o conjunto A chamamos de *domínio*, compreendendo todos os valores que a função pode *aceitar*, ao passo que o conjunto B é conhecido como *imagem*, compreendendo todos valores que a função consegue *retornar*. Podemos representar uma função de quatro formas diferentes:

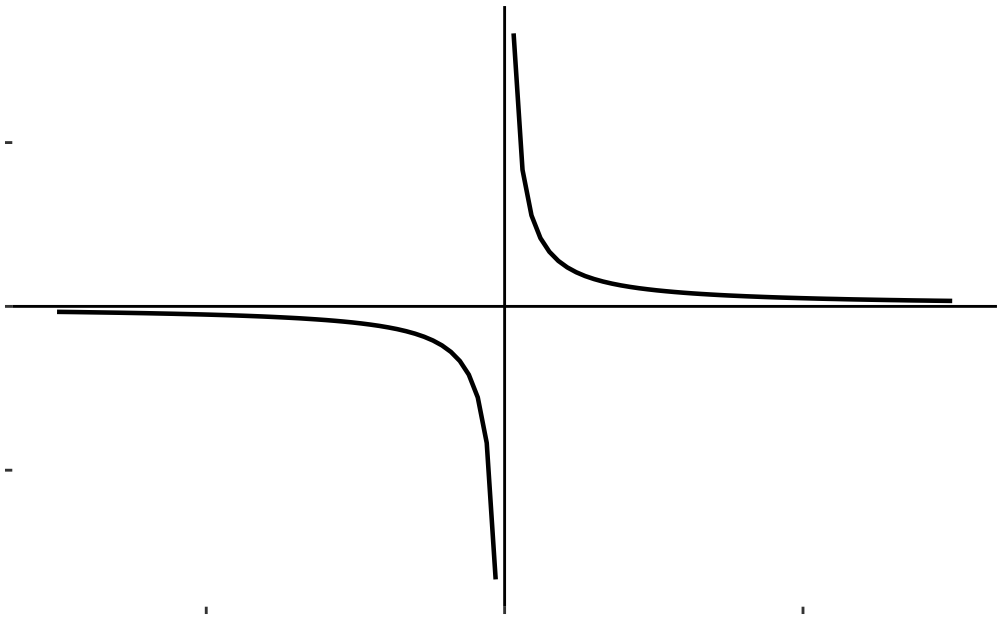
- Verbalmente (através de palavras);
- Numericamente (através uma tabela de dados);
- Visualmente (através de um gráfico);
- Algebricamente (através de uma fórmula)

Vamos deixar os conceitos mais claros. Imagine que te propronho a seguinte regra: dividirei o número 2 por todos os números *possíveis*. Observe que, nesse exemplo, os números possíveis são todos *exceto* zero – afinal de contas, qualquer número dividido por zero resulta em uma indefinição matemática. Podemos dizer, portanto, que o domínio da nossa função é dado por *todos os números reais exceto zero*. Ao mesmo tempo, quais valores podem surgir como

resultado dessa nossa regrinha? Novamente, *todos os reais exceto zero* – afinal, nenhum número que utilizemos será tão grande em valor absoluto tal que a divisão resulte em zero, ao passo que qualquer valor extremo (tanto negativo, quanto positivo) pode ser alcançado *plugando* na função valores menores do que um em módulo[1]! Portanto, nesse caso, a imagem da nossa função é idêntica ao domínio.

Perceba que definimos a função acima de modo *verbal*, ou seja, com palavras. Poderíamos também descrevê-la numericamente através de uma tabela ou, então, de forma visual com um gráfico (ambas abaixo). De todo modo, a forma mais comum de se descrever uma função é através de sua *fórmula* que, no nosso exemplo, seria $f(x) = 2/x$.

x	$f(x)$
-3	-0.67
-2	-1.00
-1	-2.00
1	2.00
2	1.00
3	0.67



[1] Pense em números como $0,01$, por exemplo. Inserindo na nossa função, teríamos então $f(x) = 2/0,01 = 200$. Se inseríssemos $0,001$, obteríamos $f(x) = 2/0,001 = 2000$. O mesmo vale para valores negativos, apenas alterando o sinal do resultado. Perceba que, sendo em módulo menor do que um, quanto mais próximo de zero for o número, mais extremo será o resultado da função.

Não precisamos nos aprofundar nos conceitos. **O que você precisa guardar dessa seção é o fato de podermos pensar na função como uma *caixa preta* que, ao receber elementos de A, os transforma em um determinado valor *(output)* presente em B.** Como vimos, a forma pela qual essa 'caixa preta' irá transformar x em $f(x)$ é a nossa regra propriamente dita.

6.2 O que é uma função para o R?

E por quê essa historinha toda sobre funções matemáticas nos interessa? Simplesmente pois podemos traçar um paralelo com o conceito de função para a linguagem R! **No R, uma função é um objeto que recebe argumentos *(inputs)* e executa uma ação sobre ou a partir deles, de acordo com o bloco de código nela embutido, te devolvendo um determinado resultado *(output)*. A lógica é a mesma de uma função matemática!** O nome não é por acaso.

De certa forma, continuamos a ter um domínio, pois cada função atende um número específico de estruturas de objeto e/ou tipos de variável – algumas funções podem aceitar apenas vetores como argumento, por exemplo, enquanto outras podem ser específicas para variavéis númericas. Ao mesmo tempo, continuamos a ter uma imagem, associada aos resultados possíveis. E qual é o equivalente à regra? É o *bloco de código embutido na função!*

6.2.1 Vantagens

Nas seções anteriores, compreendemos um pouco melhor como funciona o mecanismo de uma função. Mas ainda pode haver dúvida do tipo: *"Beleza, mas em qual contexto prático que ela será útil?"*

O grande benefício de uma função se constitui no fato de seu bloco de código interior, condicionado ao valor dos argumentos, realizar sempre a mesma tarefa quando a rodamos! Isso significa que as funções permitem automatizar tarefas comuns de uma forma mais legível, evitando a prática de 'copiar e colar' repetidamente as *mesmas* linhas de código, que serão substituídas pelo nome da função e seus argumentos. Na prática, além da melhor compreensão do código, eliminamos a chance de cometer erros bobos ao copiar e colar (por exemplo, acabar atualizando o nome de uma variável em um lugar, mas não em outro) e tornamos mais fácil reutilizar o trabalho que foi escrito em outros projetos, aumentando a produtividade.

6.2.2 Criando

A *sintaxe* para criar uma função é a seguinte:

```
nome_da_funcao <- function(arg1 = default1, ..., argn = defaultn) {
  >bloco de código<
}
```

Perceba que o uso o operador `<-` nos mostra que, ao criar uma função, estamos criando um *objeto* – que, nesse caso, não é designado especificamente a armazenar dados. Entre parentêses, definimos o nome dos argumentos e, caso necessário, seus respectivos valores de *default*. Na sequência, entre chaves, escrevemos o bloco de código que rodará sobre os *inputs*. Como exemplo, vamos criar a função `soma2`.

```
soma2 <- function(somando1, somando2) {
  (somando1 + somando2) ^ 2
}
```

O que ela faz? Soma dois números e eleva esse resultado intermediário ao quadrado. A função criada poderá ser vista no quadrante superior direito, no painel **Environment**.

6.2.3 Utilizando

Por sua vez, a *sintaxe* para usar uma função é:

```
nome_da_funcao(arg1, ..., argn)
```

Em primeiro lugar, é necessário escrever o nome da função no Editor de Código. Ao lado, entre parênteses, escreveremos seus argumentos – no exemplo acima, `arg1`, `arg2` até `argn`; uma função pode ser construída de modo a ter qualquer número `n` de argumentos e eles serão sempre separados por vírgula. Esses argumentos são os nossos *inputs*. Por fim, rodamos a linha em que a escrevemos, fazendo com que seja executada pelo R e seu resultado apareça no Console.

> **i** Exemplo 1
>
> Vamos tomar como exemplo a função `sum()`. O que ela faz? Segundo sua *documentação*:
>
> > `sum` retorna a soma de todos os valores presentes em seus argumentos
>
> Na prática, como o nome já nos indica, ela tem como serventia somar todos os números que lhe forem passados. Se quisessemos utilizá-la para obter o resultado da soma dos números 4, 7 e 9, como faríamos? Se você pensou em `sum(4, 7, 9)`, acertou!
>
> ```
> sum(4, 7, 9)
> ```

```
[1] 20
```

Perceba que utilizamos três argumentos, um para cada número que somamos: 4, 7 e 9 estão associados a `arg1`, `arg2` e `arg3`, respectivamente. No entanto, dado que a função `sum()` aceita objetos como *vetores*, *matrizes* e *dataframes*, poderíamos ter utilizado apenas um único argumento!

```
sum(c(4, 7, 9))
```

```
[1] 20
```

```
sum(matrix(c(4, 7, 9)))
```

```
[1] 20
```

```
sum(data.frame(c(4, 7, 9)))
```

```
[1] 20
```

No *chunk* acima, `c(4, 7, 9)`, `matrix(c(4, 7, 9))` e `data.frame(c(4, 7, 9))` estão associados apenas ao `arg1`! Em muitos casos, entender os tipos de objetos aceitos pela função será importante para a *eficiência* do código. Imagine que estivessemos com interesse de somar todos os valores de uma certa coluna em determinado dataframe. Como poderíamos realizar essa tarefa? Dado que cada coluna de um dataframe é simplesmente um vetor, poderíamos inserí-la diretamente na função, como um único argumento!

```
df1 = data.frame(x = c(4, 7, 9))
sum(df1$x)
```

```
[1] 20
```

Lembre-se de ficar atento com relação à estrutura e/ou tipo de variável que determinada função pode aceitar. Será que se trocassemos o número 4 por "4", a função ainda rodaria? A respota é **não**, afinal de contas "4" é interpretado como texto, e não como número (e você não consegue somar textos)!

```
sum(c("4", 7, 9))
Error in sum(df1$x) : 'type' inválido (character) do argumento
```

Em muitas situações, teremos argumentos *nomeados*. Isto ocorre pois nem todo argumento

será processado da mesma forma pelo código embutido na função. Também é muito comum que certos argumentos tenham um valor pré-determinado como *default*, isto é, caso você não especifique algum valor para aquele argumento, o valor de *default* será utilizado.

i Exemplo 2

Nesse caso, vamos tomar como exemplo a função `paste()`. Qual seu papel?

> Concatenar vetores após converter em caractere

Portanto, a função `paste` irá transformar os vetores que introduzirmos em vetores com dados do tipo *character* e, na sequência, irá juntá-los. Na prática, serve para juntar palavras e/ou caracteres que estão inicialmente separados em uma única variável do tipo texto. Por exemplo, podemos estar dispostos a juntar as palavras "Estou", "apredendendo" e "a usar o R" em um único vetor.

```
paste("Estou", "aprendendo", "a usar o R")
```

```
[1] "Estou aprendendo a usar o R"
```

Note que o caractere *default* (padrão) utilizado para separar as palavras é o espaço em branco! E se quisessemos separá-las por vírgula? Nesse caso, teríamos que especificar o argumento `sep` com esse delimitador!

```
paste("Estou", "aprendendo", "a usar o R", sep = ",")
```

```
[1] "Estou,aprendendo,a usar o R"
```

Por fim note que, mesmo sem você saber, já utilizamos funções nos últimos capítulos!

- `c()`: função utilizada para criar um vetor;
- `matrix()`: para criar uma matriz;
- `data.frame()`: para criar um *data frame*;
- `list()`: para criar uma lista.

Todas essas funções utilizavam como argumento principal os dados que tínhamos interesse e, através de seu código embutido, criavam e armazenavam o objeto na memória do R!

Operadores são funções! (Opcional)

Sim, é isso mesmo que você leu! Lembre-se das nossas máximas: tudo que *existe* no R é um *objeto* e tudo que *acontece* é uma execução de *função*. É algo simples e direto ver que operadores como `+` , `:` e `<-` existem na linguagem – não à toa estão sendo mencionados. Mas perceba que eles também criam, armazenam ou transformam objetos! Por exemplo, o operador `+` transforma `1+1` em `2`! Pense nos operadores como um tipo especial de função de dois argumentos, o primeiro posicionado à esquerda e o segundo à direita.
"Mas uma função não deveria ser escrita como `nome_da_funcao(arg1, ..., argn)`*?"*
Você pode escrevê-los dessa forma também! Só não será tão útil.

```
`+`(1, 3)     # O mesmo que 1 + 3
```

```
[1] 4
```

```
`:`(1, 10)    # 1:10
```

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

```
`<-`(x, 10)   # x <- 10
```

Funções no Excel (Opcional)

Se você já usou o Microsoft Excel em algum momento da sua vida, com certeza já teve contato com o alguma função! Por exemplo, lá temos a função `=SOMA()`, que realiza a mesma tarefa da função `sum()` no R! Inclusive, também é possível criar funções no Excel através de sua linguagem de programação própria, o *Visual Basic for Applications* (VBA).

6.3 Pacotes

Chamamos de *pacote* um conjunto de dados e/ou funções, acompanhadas de suas respectivas *documentações*, que foram criadas e disponibilizadas por alguma pessoa. Existem pacotes padrão (ou básicos) que são considerados parte do código-fonte do R e estão automaticamente disponíveis como parte da instalação do R (ou seja, foram criados pelos desenvolvedores da linguagem). No entanto, definitivamente não são a maioria: o grosso dos pacotes disponíveis é de autoria dos membros da comunidade. Normalmente, as funções que integram um pacote estão relacionadas à determinado tema, ainda que isso não seja obrigatório.

Por exemplo, suponha que criemos quatro funções que antes não existiam na linguagem:

`soma()`, `subtração()`, `multiplicacao()` e `divisao()`, representando as quatro operações aritméticas básicas. Como estão relacionadas à um mesmo tema, poderiamos agrupá-las em um pacote chamado `aritmetica` e, na sequência, disponibilizá-lo em algum *repositório* para que outros usuários da linguagem pudessem baixá-lo. **As pessoas que instalassem e carregassem nosso pacote teriam acesso às quatro funções acima, sem necessidade de ter que criá-las do zero!** Ao mesmo tempo, o autor poderia adicionar um dataframe ao pacote, com intuito de possibilitar que você teste as funções que ele criou[2].

Perceba que, quando alguém cria uma função ou disponibiliza dados através de um pacote, fica complicado compreender inicialmente todas as características que esses objetos possuem. Por exemplo: ainda que os nomes sejam sugestivos, você saberia dizer com precisão o comportamento das quatro funções do pacote que criamos? Quais os tipos de objeto e variável que elas aceitam? Quais e quantos argumentos cada uma aceita? Provavelmente não. Para responder à todas essas perguntas, os autores disponibilizam uma *documentação* para cada objeto do pacote! Para acessá-la, basta rodar o nome da função/conjunto de dados acrescido de `?` no início. O texto irá aparecer no quadrante inferior direito do RStudio, no painel **Help**[3].

6.3.1 Instalando

A maneira mais comum de se baixar e instalar um pacote é através do CRAN! O mesmo local em que baixamos o R também atua como um repositório centralizado de pacotes. Mas fique tranquilo: você não precisará acessar o site novamente! Para baixar e instalar um pacote que está no CRAN, utilizaremos a função `install.packages()`, pertencente ao pacote `utils` (que é um pacote básico). Basta rodar `install.packages("nome_do_pacote")`. No exemplo abaixo, a instalação do pacote `readr`.

```
install.packages("readr")
```

Realizada a instalação com sucesso, já passa a ser possível utilizar suas funções/dados. *Nesse momento*, você deverá escrever o nome do pacote acrescido de `::` e, na sequência, o nome da função/dado.

```
readr::read_csv(...)
```

[2]Outra possibilidade seria adicionar um dataframe simplesmente pelo interesse em utilizar as informações que podem estar contidas nele. Por exemplo, existem pacotes que contém apenas dataframes com informações de tabelas de livros-texto específicos; com isso, os usuários ganham o poder de replicar os resultados encontrados pelo autor, facilitando o aprendizado. É importante ressaltar, contudo, que pacotes normalmente são compostos apenas por funções.

[3]As documentações sempre aparecerão escritas em inglês.

6.3.2 Carregando

"Mas será sempre necessário escrever o nome do pacote antes da função?" **Não.** Perceba que os objetos oriundos de pacotes básicos podem ser executados diretamente. Isso ocorre pois eles são automaticamente *carregados* na sua sessão atual.

Resumindo: mesmo que você tenha instalado um pacote externo com sucesso, para usar seus objetos diretamente (sem precisar escrever seu nome antes) é necessário *carregá-lo* na sessão atual. Para carregar um pacote, rode a função `library()` acrescida do nome do pacote, *sem* aspas.

```
library(readr)
```

Agora, é possível executar o objeto sem precisar escrever o nome do pacote antes.

```
read_csv(...)
```

A prática de carregar pacotes é a mais utilizada. Em outras palavras: sempre que instalarmos algum pacote, na sequência iremos carregá-lo na sessão atual para que possamos utilizar seus objetos de forma direta.

6.4 Operador Pipe %>%

Nem sempre conseguiremos atingir o resultado que queremos utilizando apenas uma função. Por esse motivo, em muitas situações utilizaremos o *resultado* de uma função como *argumento* de *outra* função.

Por exemplo, suponha que você queira somar dois números e, na sequência, comparar este resultado intermediário com 5 e 9, de modo a retornar o valor máximo entre os três números. Observe que a função `sum()` não é suficiente para realizar tal tarefa de modo completo: você até conseguirá somar dois números quaisquer, mas não será capaz de posteriormente comparar o resultado com o restante para saber qual é o valor máximo entre eles. Nesse caso, poderíamos então utilizar o resultado da função `sum()` como *argumento* da função `max()`!

```
max(sum(4, 3), 5, 9)
```

```
[1] 9
```

O R sempre executará as funções *interiores* primeiro. Ou seja, primeiro executa `sum(4,3)`, retornando 7, e na sequência executa `max(7, 5, 9)`, cujo resultado será 9, dado que este é o maior dentre os três números utilizados como argumento.

O problema com esse tipo de *sintaxe* é que, conforme aproveitamos os resultados anteriores de outra função como argumento para as seguintes, mais confuso o código fica. Imagine se o número 4 também fosse resultado de alguma outra função: o código estaria bem mais difícil de entender!

```
max(sum(outra_funcao(...), 3), 5, 9)
```

Com a finalidade de simplificar situações desse tipo, criou-se o operador Pipe, representado por `%>%`. Este operador permite com que o resultado da função anterior se torne, *implicitamente,* o primeiro argumento da função seguinte! Poderíamos então reescrever nosso exemplo para:

```
sum(4,3) %>% max(5, 9)
```

```
[1] 9
```

O que o código acima nos diz é que `sum(4,3)` será interpretado como o primeiro argumento da função `max()`; automaticamente, 5 e 9 se tornam o segundo e o terceiro argumentos, respectivamente.

Para utilizar o operador Pipe, antes é necessário instalar e carregar o pacote `magrittr`:

```
install.packages("magrittr")
library(magrittr)
```

Na sequência, aperte `Ctrl + Shift + M` no Editor ou Console.

6.4.1 Pipe nativo

O uso do operador Pipe se tornou tão poopular que os desenvolvedores do R resolveram incorporar uma versão própria que já vem pré-instalada, conhecida como Pipe nativo *(native Pipe)*. Ele exerce o mesmo papel principal (organização de código, como demonstrado na seção anterior) mas deixa a desejar em outras partes. A única vantagem é não precisar instalar e carregar um pacote. Portanto, recomendamos ainda o uso do Pipe 'original'.

De todo modo, caso você queira utilizá-lo, basta substituir `%>%` por `|>`. Para continuar usando o atalho `Ctrl + Shift + M`, vá em *Tools* > Global Options > *Code* e marque *Use native pipe operator*.

Parte III

Dados

O sétimo capítulo te ensinará a como armazenar informações externas no R. É importante que você saiba esse tópico pois, em algumas matérias, seu professor lhe entregará arquivos com informações nos quais algumas tarefas deverão ser executadas com auxílio da linguagem. O oitavo capítulo, por sua vez, te ensinará como modificar as informações de determinada estrutura de dados.

7 Importando

Imagine que você compre um computador produzido *fora* do Brasil. Nesse caso, dizemos que você *importou* o computador de determinado país que o produziu, não é? Inclusive, segundo a Oxford Languages, o verbo *importar* pode ser definido como

> trazer de outro país, estado ou município.

No campo da programação, *importar* mantém signifcado semelhante: trazer dados externos para nosso ambiente, de forma que possamos manipulá-los com a linguagem. Aplicando à nossa realidade, queremos trazer tabelas de dados para a memória do R, na aba Environment, de forma que possamos manipulá-las posteriormente!

7.1 Definindo o diretório de trabalho

Antes de importar, é interessante definirmos nosso *diretório de trabalho*, que corresponde ao caminho para a pasta fixa que iremos utilizar para criar ou armazenar arquivos. Pense no diretório como a pasta do seu computador que servirá como local de armazenamento de todos os arquivos relacionados ao trabalho/projeto que você estiver executando naquele momento – sejam scripts, planilhas, etc.

Para configurar determinada pasta como diretório de trabalho, aperte `Ctrl + Shift + H` e, na sequência, selecione a pasta que desejar. Perceba que esse comando rodará a função `setwd()` no Console, cujo argumento é o caminho para a pasta. Você também pode selecionar o diretório de trabalho desta maneira direta.

```
setwd("caminho_para_pasta")
```

Note que, no RStudio, o caminho para o diretório atual aparece na parte superior do painel de Console, ao lado do número da versão do R que você estiver utilizando no momento. Ao mesmo tempo, você pode retornar o diretório atual de trabalho apenas rodando a função `getwd()`, sem nenhum argumento.

7.2 Funções mais utilizadas para importação

Com o propósito de importar dados para o RStudio, iremos aprender a utilizar algumas funções específicas. Repare que estamos falando de *funções*, ou seja, existe mais de uma função que busca permitir com que o RStudio seja capaz de ler e armazenar dados internamente em sua memória, para posterior manipulação. Isso acontece pois não existe um único tipo de arquivo capaz de armazenar dados.

Dois pacotes serão utilizados: `readr`, que faz parte do *tidyverse*, e `openxlsx`. Portanto, é necessário que você os instale, caso ainda não tenha feito e, na sequência, carrege os pacotes.

```
install.packages("readr")
install.packages("openxlsx")

library(readr)
library(openxlsx)
```

7.2.1 O pacote readr – lendo arquivos delimitados

O pacote `readr` é utilizado principalmente para ler arquivos delimitados por algum caractere específico. *'Como assim, arquivos delimitados por caractere?'* Em muitos casos, as tabelas de dados são tão grandes – ou seja, várias observações (linhas) e variáveis (colunas) – que será necessário reduzir seu tamanho com intuito de facilitar o compartilhamento com terceiros. Uma saída é comprimir todas as variáveis em uma única coluna, em que os dados são *separados por algum caractere especial*, mantendo o mesmo número de observações anterior. Cada linha continua representando uma observação de determinada unidade. Passamos de n observações e m colunas para n observações e 1 coluna.

7.2.1.1 `read_csv()`

Para começar, vamos nos concentrar no tipo de arquivo de dados retangular mais comum: CSV, que é a abreviação de *Comma Separeted Values* (valores separados por vírgula, em português). Abaixo, temos a aparência de um arquivo CSV simples, contendo informações de estudantes. A primeira linha, comumente chamada de *cabeçalho*, fornece os nomes das colunas, e as cinco linhas seguintes fornecem os dados.

```
matrícula,Nome,comida.favorita,PlanoDeRefeição,IDADE,peso
1,Jorge,Acarajé,Regime,18,76
2,Laís,Macarrão,Livre,23,65
3,Matheus,Carne,Regime,22,70
4,Laura,Frango,Livre,21,68
```

```
5,Nathália,Peixe,Regime,21,66
```

A Tabela 7.1 mostra uma representação dos mesmos dados em uma tabela.

Tabela 7.1Dados do arquivo estudantes.csv como tabela.

matrícula	Nome	comida.favorita	PlanoDeRefeição	IDADE	peso
1	Jorge	Acarajé	Regime	18	76
2	Laís	Macarrão	Livre	23	65
3	Matheus	Carne	Regime	22	70
4	Laura	Frango	Livre	21	68
5	Nathália	Peixe	Regime	21	66

Quando temos um arquivo do tipo `csv`, podemos importá-lo para o R usando a função `read_csv()`. O primeiro argumento é o mais importante: o *caminho* para o arquivo. Você pode pensar no caminho como o *endereço* do arquivo, ou seja, é o local em que ele está armazenado.

Se o arquivo estiver no seu computador, é necessário escrever o caminho em termos de diretório e pastas, além do nome do arquivo e sua extensão. Lembre-se que já fixamos nosso diretório de trabalho. Basta então escrever o restante do caminho, considerando que o arquivo está na pasta `dados`[1] e se chama `estudantes.csv`!

```
estudantes <- read_csv("dados/estudantes.csv")
## Rows: 5 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (3): Nome, comida.favorita, PlanoDeRefeição
## dbl (3): matrícula, IDADE, peso
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Por outro lado, você pode baixar o arquivo diretamente de algum *link* hospedado na internet! A vantagem dessa alternativa consiste no fato de quê você não precisará realizar novamente o *download* do arquivo no seu computador em caso de atualização do arquivo original – além, é claro, de tornar seu diretório de trabalho mais limpo.

[1] A pasta `dados` foi criada com intuito de aprimorar a organização do diretório de trabalho. Você **não** precisa tê-la em seu computador. Se você já fixou o diretório base, pode importar arquivos apenas com `read_csv("nome_do_arquivo.csv")`.

```
estudantes <- read_csv("https://raw.githubusercontent.com/ieufrjquant/intror/master/dados/est
```

Independente da forma com que você especifique o caminho para o arquivo ao executar a função `read_csv()`, note que ela exibe uma mensagem informando o número de linhas e colunas de dados, o delimitador que foi usado e as especificações das colunas (nomes das colunas organizadas pelo tipo de dados que a coluna contém).

Observe também que, em ambos os casos, atribuimos a tabela de dados ao objeto `estudantes`, ficando disponível para visualização no painel Environment. Quando for importar alguma tabela para o R, é importante que você atribua os dados a um objeto – cujo nome, como sabemos, é de livre escolha. Caso contrário, o arquivo será apenas lido no Console, ao invés de ficar efetivamente armazenado para manipulação[2]. Por fim, importante observar que o objeto criado terá estrutura de *data frame*.

> Funções para outros tipos de arquivo delimitados (Opcional)
>
> Depois de dominar `read_csv()`, usar as outras funções do readr é simples; é apenas uma questão de saber qual função buscar:
>
> - `read_csv2()` lê arquivos separados por ponto e vírgula. Eles usam `;` em vez de `,` para separar campos e são comuns em países que usam `,` como marcador decimal.
> - `read_tsv()` lê arquivos delimitados por tabulações.
> - `read_delim()` lê arquivos com qualquer delimitador, tentando adivinhar automaticamente o delimitador se você não especificá-lo.
> - `read_fwf()` lê arquivos de largura fixa. Você pode especificar campos por suas larguras com `fwf_widths()` ou por suas posições com `fwf_positions()`.
> - `read_table()` lê uma variação comum de arquivos de largura fixa onde as colunas são separadas por espaços em branco.

7.2.2 O pacote readxl – lendo planilhas

Nesta seção, iremos nos concentrar em importar dados de *planilhas*, especificamente os que foram agrupados em planilhas de *Excel*. Veremos como importar todos os dados de uma planilha com apenas uma única aba, assim como de uma aba específica, caso exista mais de uma.

[2] É exatamente a mesma lógica de armazenamento vista no capítulo sobre objetos.

7.2.2.1 `read.xlsx()`

A maioria dos arquivos escritos em Excel hoje possui uma extensão do tipo `.xlsx`. Portanto, vamos focar na função do pacote `openxlsx` que melhor lide com planilhas desse tipo: `read.xlsx()`. Como no caso anterior, note que o objeto criado terá estrutura de um *data frame*.

7.2.2.1.1 Planilha

Esse caso se aplica quando você desejar importar *todos* os dados que estão na *primeira e única* aba de uma planilha. O procedimento é muito parecido com o que fizemos no caso de arquivos delimitados: precisaremos do endereço para o arquivo que desejamos importar, que pode ser um caminho de pastas no seu computador ou um *link* externo.

```
estudantes <- read.xlsx("dados/estudantes.xlsx")
estudantes
```

```
  matrícula     Nome comida.favorita PlanoDeRefeição IDADE peso
1         1    Jorge         Acarajé          Regime    18   76
2         2     Laís        Macarrão           Livre    23   65
3         3  Matheus           Carne          Regime    22   70
4         4    Laura          Frango           Livre    21   68
5         5 Nathália           Peixe          Regime    21   66
```

7.2.2.1.2 Aba específica

Para importar dados de uma aba específica de determinada planilha é necessário especificar, além do endereço do arquivo, o nome ou o índice da aba que você deseja. No exemplo abaixo, importamos os dados completos do *Maddison Project Database 2020*, que estão contidos na terceira aba da planilha disponibilizada pelos organizadores.

```
dados <- read.xlsx("https://www.rug.nl/ggdc/historicaldevelopment/maddison/data/mpd2020.xlsx
                   sheet = "Full data")

dados <- read.xlsx("https://www.rug.nl/ggdc/historicaldevelopment/maddison/data/mpd2020.xlsx
                   sheet = 3)

dados
```

8 Manipulando

Perceba que estamos construindo uma sequência lógica de aprendizado. No Capítulo 4, nos familiarizamos um pouco melhor com comandos no R e o ambiente do RStudio. Nos Capítulo 5 e Capítulo 6, aprendemos os conceitos de objeto e função, respectivamente, utilizando a experiência adquirida no capítulo anterior para criá-los e armazená-los sem a desconfiança inicial de quem nunca teve contato com a linguagem. Por fim, no Capítulo 7 desenvolvemos a capacidade de importar dados de fontes externas ao R – sejam eles do mundo real ou apenas construídos pelo seu professor. Em resumo, você pode pensar que chegamos neste momento com um certo conhecimento e prática na linguagem, aptos então a aprender como *modificar* um objeto contendo dados.

"Como assim, modificar um objeto?" Modificar significa alterar ou transformar seus dados. Podemos, por exemplo, adicionar/remover linhas ou colunas contendo novas informações. Ou, então, simplesmente alterar determinado dado sem alterar o número de linhas ou colunas da estrutura.

8.1 Sem funções externas

8.2 dplyr