

# Julia Data Science

Jose Storopoli

Rik Huijzer

Lazaro Alonso

Jose Storopoli  
Universidade Nove de Julho - UNINOVE  
Brazil

Rik Huijzer  
University of Groningen  
the Netherlands

Lazaro Alonso  
Max Planck Institute for Biogeochemistry  
Germany

First edition published 2021

<https://juliadatascience.io>

ISBN: 9798489859165

2022-12-11

Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International

# *Contents*

<b>1</b>	<b>Prefácio</b>	<b>3</b>
1.1	O que é Ciéncia de Dados? . . . . .	4
1.2	Engenharia de software . . . . .	5
1.3	Agradecimentos . . . . .	6
<b>2</b>	<b>Por que Julia?</b>	<b>7</b>
2.1	Para os que nunca programaram . . . . .	7
2.2	Para programadores . . . . .	8
2.3	O que Julia pretende alcançar? . . . . .	9
2.4	Julia Por Aí . . . . .	16
<b>3</b>	<b>Básico de Julia</b>	<b>19</b>
3.1	Sintaxe da linguagem . . . . .	19
3.2	Estruturas Nativas de Dados . . . . .	32
3.3	Sistema de Arquivos . . . . .	61
3.4	Biblioteca Padrão de Julia . . . . .	63
<b>4</b>	<b>DataFrames.jl</b>	<b>77</b>
4.1	Carregar e salvar arquivos . . . . .	82
4.2	Indexação e Sumarização . . . . .	87
4.3	Filtro e Subconjunto . . . . .	90
4.4	Select . . . . .	95
4.5	Tipos de Dados e Dados Faltantes . . . . .	98
4.6	Join . . . . .	101
4.7	Transformações de Variáveis . . . . .	106
4.8	Groupby e Combine . . . . .	109
4.9	Desempenho . . . . .	112

<b>5 Visualização de dados com Makie.jl</b>	<b>119</b>
5.1 CairoMakie.jl . . . . .	120
5.2 Atributos . . . . .	121
5.3 Temas . . . . .	127
5.4 Usando LaTeXStrings.jl . . . . .	131
5.5 Cores e mapas de cores . . . . .	134
5.6 Layouts . . . . .	139
5.7 GLMakie.jl . . . . .	149
<b>6 Apêndice</b>	<b>161</b>
6.1 Versões dos Pacotes . . . . .	161
6.2 Formatação . . . . .	161
<b>Referências</b>	<b>165</b>

# 1 Prefácio

Existem várias linguagens de programação e cada uma delas tem seus pontos fortes e fracos. Algumas linguagens são mais rápidas, mas bastante verbosas. Outras linguagens são fáceis de codificar, mas são lentas. Isso é conhecido como o problema das *duas linguagens* e Julia busca eliminar esse problema. Mesmo que nós três somos de áreas distintas, todos acreditamos que Julia é mais efetiva para nossas pesquisas que as outras linguagens que usamos antes. Discorremos sobre esse ponto de vista em Section 2. Contudo, comparada a outras linguagens de programação, Julia é uma das mais recentes. Isso significa que o ecossistema em torno da linguagem é, por vezes, de difícil navegação. É difícil descobrir por onde começar e como todos os diferentes pacotes se encaixam. Por isso decidimos escrever este livro! Desejamos tornar mais acessível para pesquisadores, em especial para nossos colegas, essa linguagem incrível.

Como pontuado anteriormente, cada linguagem de programação tem seus pontos fortes e fracos. Para nós, a ciência de dados é, definitivamente, um ponto forte de Julia. Ao mesmo tempo, nós três usamos ferramentas de ciência de dados no nosso dia a dia. E provavelmente você também quer trabalhar com ciência de dados! Por isso, nosso livro é focado em ciência de dados.

Na próxima etapa dessa seção daremos uma ênfase maior **na importância dos “dados” em ciência de dados** e mostraremos porque a habilidade em manipulação de dados é, e continuará sendo, a **maior demanda** do mercado e da academia. Argumentamos que **incorporar as práticas da engenharia de software à ciência de dados** reduzirá o atrito na atualização e no compartilhamento de códigos entre colaboradores. Grande parte da análise de dados vem de um esforço colaborativo, por isso as práticas de desenvolvimento de software são de grande ajuda.

## 1.0.1 Dados estão em todos os lugares

**Dados são abundantes** e serão ainda mais em um futuro próximo. Um relatório do final do ano de 2012, concluiu que, entre 2005 e 2020, a quantidade de dados armazenados digitalmente cresceria em um fator de 300, partindo de 130 exabytes<sup>1</sup> para impressionantes 40,000 exabytes ([Gantz & Reinsel, 2012](#)). Isso equivale a 40 trilhões de gigabytes: para colocarmos em perspectiva, são mais de 5.2 terabytes para cada ser humano que vive neste planeta! Em 2020,

<sup>1</sup> 1 exabyte (EB) = 1,000,000 terabyte (TB).

em média, cada pessoa criou **1.7 MB de dados por segundo** ([Domo, 2018](#)). Um relatório recente previu que quase **dois terços (65%) dos PIBs nacionais estarão digitalizados até 2022** ([Fitzgerald et al., 2020](#)).

Todas as profissões serão impactadas pelo aumento e disponibilidade cada vez maior de dados ([Chen et al., 2014](#); [Khan et al., 2014](#)) e pela crescente importância que os dados têm tomado. Dados são usados para comunicar e construir conhecimento, assim como na tomada de decisões. É por isso que habilidade com dados é tão importante. Estar confortável ao lidar com dados fará de você um pesquisador e/ou profissional valioso. Em outras palavras, você se tornará **alfabetizado em dados**.

## 1.1 O que é Ciência de Dados?

Ciência de dados não se trata apenas de aprendizado de máquina e estatística, também não é somente sobre predição. Também não é uma disciplina totalmente contida nos campos STEM (Ciências, Tecnologia, Engenharia e Matemática) ([Meng, 2019](#)). Entretanto, podemos afirmar com certeza que ciência de dados é sempre sobre **dados**. Nossos objetivos com este livro são dois:

- Focar na espinha dorsal da ciência de dados: **dados**.
- E o uso da linguagem de programação **Julia** para o processamento de dados.

Explicamos porque Julia é uma linguagem extremamente eficaz para a ciência de dados em [Section 2](#). Por enquanto, vamos focar nos dados.

### 1.1.1 Alfabetização de dados

De acordo com a Wikipedia<sup>2</sup>, a definição formal para **alfabetização de dados** é “**a habilidade de ler, entender, criar e comunicar dados enquanto informação**.”. Também gostamos da concepção informal de que, ao se tornar alfabetizado em dados, você não se sentirá sufocado pelos dados, mas sim, saberá utilizá-los na tomada correta de decisões. Alfabetização de dados é uma habilidade extremamente competitiva. Neste livro iremos abordar dois importantes aspectos da alfabetização de dados:

<sup>2</sup> [https://pt.wikipedia.org/wiki/Alfabetiza%C3%A7%C3%A3o\\_de\\_dados](https://pt.wikipedia.org/wiki/Alfabetiza%C3%A7%C3%A3o_de_dados)

1. **Manipulação de dados** com `DataFrames.jl` ([Section 4](#)). Neste capítulo, você aprenderá a:
  1. Ler dados em CSV e Excel com Julia.
  2. Processar dados em Julia, ou seja, aprender a responder questões com dados.

3. Filtrar dados e criar subconjuntos de dados.
  4. Lidar com dados faltantes.
  5. Unir e combinar dados provenientes de várias fontes.
  6. Agrupar e resumir dados.
  7. Exportar dados do Julia para arquivos CSV e Excel.
2. **Visualização de dados** com `Makie.jl` (Section 5). Neste capítulo você entenderá como:
1. Plotar dados utilizando diversos backends do `Makie.jl`.
  2. Salvar visualizações nos mais diferentes formatos, como PNG ou PDF.
  3. Usar diferentes funções de plotagem para criar diversas formas de visualização dos dados.
  4. Customizar visualizações por meio de atributos.
  5. Usar e criar novos temas para suas plotagens.
  6. Adicionar elementos *LATEX* aos plots.
  7. Manipular cores e paletas.
  8. Criar layouts de figuras complexas.

## 1.2 *Engenharia de software*

Diferentemente de boa parte da literatura sobre ciência de dados, esse livro dá uma ênfase maior para a **estruturação do código**. A razão para isso é que notamos que muitos cientistas de dados simplesmente inserem seu código em um arquivo enorme e rodam as instruções sequencialmente. Uma analogia possível seria forçar as pessoas a lerem um livro sempre do início até o final, sem poder consultar capítulos anteriores ou pular para seções mais interessantes. Isso funciona para projetos menores, mas quanto maior e mais complexo for o projeto, mais problemas aparecerão. Por exemplo, um livro bem escrito é dividido em diferentes capítulos e seções que fazem referência a diversas partes do próprio livro. O equivalente a isso em desenvolvimento de software é **dividir o código em funções**. Cada função tem nome e algum conteúdo. Ao usar corretamente as funções você pode determinar que o computador, em qualquer ponto do código, pule de um lugar para outro e continue a partir daí. Isso permite que você reutilize o código com mais facilidade entre projetos, atualize o código, compartilhe o código, colabore e tenha uma visão mais geral do processo. Portanto, com as funções, você pode **otimizar o tempo**.

Assim, ao ler este livro, você acabará se acostumando a ler e usar funções. Outro benefício em ser hábil na engenharia de software é compreender com mais facilidade o código-fonte dos pacotes que utiliza, algo essencial quando se depura códigos ou quando buscamos entender exatamente como os pacotes que utilizamos funcionam. Por fim, você pode ter certeza de que não inven-

tamos essa ênfase em funções. Na indústria, é comum estimular desenvolvedores a usarem “**funções ao invés de comentários**”. Isso significa que, em vez de escrever um comentário para humanos e algum código para o computador, os desenvolvedores escrevem uma função que é lida por humanos e computadores.

Além disso, nos esforçamos muito para seguir um guia de estilo consistente. Os guias de estilo de programação fornecem diretrizes para a escrita de códigos, por exemplo, sobre onde deve haver espaço em branco e quais nomes devem ser iniciados com letra maiúscula ou não. Seguir um guia de estilo rígido pode parecer pedante e algumas vezes é. No entanto, quanto mais consistente o código for, mais fácil será sua leitura e compreensão. Pra ler nosso código, você não precisa entender nosso guia de estilo. Você perceberá enquanto lê. Se quiser conhecer os detalhes de nosso guia de estilo, acesse Section 6.2.

### 1.3 Agradecimentos

Muitas pessoas colaboraram direta ou indiretamente para a criação deste livro.

Jose Storopoli agradece sua família, em especial sua esposa pelo suporte e compreensão durante a escrita e revisão da obra. Ele também agradece aos seus colegas, em especial a Fernando Serra<sup>3</sup>, Wonder Alexandre Luz Alves<sup>4</sup> e André Librantz<sup>5</sup>, por seu suporte.

Rik Huijzer agradece em primeiro lugar seus supervisores de PhD na Universidade de Groningen, Peter de Jonge<sup>6</sup>, Ruud den Hartigh<sup>7</sup> e Frank Blaauw<sup>8</sup>. Em segundo lugar, agradece aos pais e a sua namorada por toda a compreensão durante feriados, finais de semana e noites dedicadas a este livro.

Lazaro Alonso agradece sua esposa e suas filhas por todo suporte durante o projeto.

Agradecemos também à tradução de Nádia Lebedev e à revisão técnica de Henrique Pougy<sup>9</sup> e Elias Noda<sup>10</sup>.

<sup>3</sup> <https://orcid.org/0000-0002-8178-7313>

<sup>4</sup> <https://orcid.org/0000-0003-0430-950X>

<sup>5</sup> <https://orcid.org/0000-0001-8599-9009>

<sup>6</sup> <https://www.rug.nl/staff/peter.de.jonge/>

<sup>7</sup> <https://www.rug.nl/staff/j.r.den.hartigh/>

<sup>8</sup> <https://frankblaauw.nl/>

<sup>9</sup> <https://github.com/h-pgy>

<sup>10</sup> <https://github.com/Elias-Noda>

## 2 Por que Julia?

O mundo da ciência de dados é repleto de diferentes linguagens open source.

A Indústria tem, em grande parte, adotado as linguagens Python e R. **Por que aprender uma outra linguagem?** Para responder a essa questão, abordaremos duas situações bastante comuns:

1. **Nunca programou antes** – see Section [2.1](#).
2. **Já programou** – see Section [2.2](#).

### 2.1 Para os que nunca programaram

Para a primeira situação, acreditamos que o ponto em comum seja o seguinte.

A ciência de dados te atrai, você tem vontade de aprender sobre e entender como ela pode ajudar sua carreira seja na academia, seja no mercado. Então, você tenta encontrar formas de aprender essa nova habilidade e cai em um mundo de acrônimos complexos: `pandas`, `dplyr`, `data.table`, `numpy`, `matplotlib`, `ggplot2`, `bokeh`, e a lista continua.

E, do nada, você ouve: “Julia.” O que é isso? Como seria diferente de qualquer outra ferramenta usada para ciência de dados?

Por que você deveria gastar seu tempo para aprender uma linguagem de programação que quase nunca é mencionada em processos seletivos, posições em laboratórios, pós-doutorados ou qualquer outro trabalho acadêmico? A resposta para a questão é que **Julia é uma nova abordagem** tanto para programação, quanto para ciência de dados. Tudo que você faz em Python ou R, você pode fazer em Julia com a vantagem de poder escrever um código legível<sup>1</sup>, rápido e poderoso. Assim, Julia tem ganhado força por uma série de motivos.

Então, se você não tem nenhum conhecimento prévio de programação, recomendamos que aprenda Julia como uma primeira linguagem de programação e ferramenta para ciência de dados.

<sup>1</sup> sem quaisquer chamadas de API em C++ ou FORTRAN.

## 2.2 Para programadores

Na segunda situação, a história por trás muda um pouco. Você é uma pessoa que não só sabe programar, como também, provavelmente, vive disso. Você tem familiaridade com uma ou mais linguagens de programação e deve transitar bem entre elas. Você ouviu falar sobre “ciência de dados” e quer surfar essa onda. Você começou a aprender a fazer coisas em `numpy`, a manipular `DataFrames` em `pandas` e como plotar em `matplotlib`. Ou talvez você tenha aprendido tudo isso em R usando `tidyverse` e `tibbles`, `data.frames`, `%>% (pipes)` e `geom_*`...

Então, por alguém ou por algum lugar, você ouviu falar dessa nova linguagem chamada “Julia.” Por que se importar? Você já domina Python ou R e consegue fazer tudo que precisa. Bom, vamos analisar alguns possíveis cenários.

**Alguma vez você já fez em Python ou R:**

1. Algo que não tenha conseguido alcançar a performance necessária? Então, **em Julia, minutos no Python ou R se transformam em segundos**<sup>2</sup>. Nós separamos o Section 2.4 para exemplificar casos de sucesso em Julia tanto na academia quanto no mercado.
2. Tentou algo diferente das convenções `numpy/dplyr` e descobriu que o código estava lento e provavelmente precisaria de magia<sup>3</sup> para torná-lo mais rápido? **Em Julia, você pode fazer seu próprio código customizado sem perder desempenho**.
3. Precisou debugar um código e de repente se viu lendo código fonte em Fortran ou C/C++, sem ter ideia alguma do que fazer? **Em Julia, você lê apenas códigos de Julia, não é preciso programar em outra linguagem para tornar a original mais rápida**. Isso é chamado o “problema das duas linguagens” (see Section 2.3.2). É também o caso quando “você tem uma ideia interessante e tenta contribuir com um pacote open source, mas desiste porque quase tudo não está nem em Python, nem em R, mas em C/C++ ou Fortran”<sup>4</sup>.
4. Quis usar uma estrutura de dados definida em outro pacote e descobriu que não ia funcionar e que você precisaria construir uma interface<sup>5</sup>. **Julia permite que usuários compartilhem e reutilizem códigos de diferentes pacotes com facilidade**. A maior parte dos tipos e funções definidas pelos usuários de Julia, funcionam de imediato<sup>6</sup> e alguns usuários ficam maravilhados ao descobrir como seus pacotes estão sendo usados por outras bibliotecas, das mais diversas formas, algo que nunca poderiam ter imaginado. Temos alguns exemplos em Section 2.3.3.
5. Precisou de uma melhor gestão de projetos, com controle rígido de versões e dependências, de fácil usabilidade e replicável? **Julia tem uma solução de**

<sup>4</sup> dê uma olhada em algumas bibliotecas de aprendizado profundo no GitHub e você descobrirá que Python é apenas 25%-33% do código fonte delas.

<sup>5</sup> esse é um problema do ecossistema Python e, ainda que o R não sofra tanto com isso, também não é tão eficaz.

<sup>6</sup> ou com pouquíssimo esforço.

**gestão de projetos incrível e um ótimo gerenciador de pacotes.** Diferentemente dos gerenciadores de pacotes tradicionais, que instalam e gerenciam um único conjunto global de pacotes, o gerenciador de pacotes de Julia é projetado em torno de “ambientes”: conjuntos independentes de pacotes que podem ser locais para um projeto individual ou compartilhados entre projetos. Cada projeto mantém, independentemente, seu próprio conjunto de versões de pacotes.

Se nós chamamos a sua atenção expondo situações familiares ou mesmo plausíveis, talvez você se interesse em aprender um pouco mais sobre Julia.

Vamos começar!

### 2.3 O que Julia pretende alcançar?

**NOTE:** Nessa seção explicaremos com detalhes o que faz de Julia uma linguagem de programação brilhante. Se essa explicação for muito técnica para você, vá direto para Section 4 para aprender sobre dados tabulares com `DataFrames.jl`.

A linguagem de programação Julia ([Bezanson et al., 2017](#)) é relativamente nova, foi lançada em 2012, e procura ser **fácil e rápida**. Ela “roda como C<sup>7</sup>, mas lê como Python” ([Perkel, 2019](#)). Foi idealizada inicialmente para computação científica, capaz de lidar com **uma grande quantidade de dados e demanda computacional** sendo, ao mesmo tempo, **fácil de manipular, criar e prototipar códigos**.

Os criadores de Julia explicaram porque desenvolveram a linguagem em uma postagem em seu blog em 2012<sup>8</sup>. Eles afirmam:

Somos ambiciosos: queremos mais. Queremos uma linguagem open source, com uma licença permissiva. Queremos a velocidade do C com o dinamismo do Ruby. Queremos uma linguagem que seja homoiconica, com verdadeiros macros como Lisp, mas com uma notação matemática óbvia e familiar como Matlab. Queremos algo que seja útil para programação em geral como Python, fácil para estatística como R, tão natural para processamento de strings quanto Perl, tão poderoso para álgebra linear quanto Matlab, tão bom para integrar programas juntos quanto shell. Algo que seja simples de aprender, mas que deixe os hackers mais sérios felizes. Queremos que seja interativa e que seja compilada.

A maioria dos usuários se sentem atraídos por Julia em função da sua **velocidade superior**. Afinal, Julia é membro de um clube prestigiado e exclusivo. O **petaflop club**<sup>9</sup> é composto por linguagens que excedem a velocidade de **um petaflop no desempenho máximo**. Atualmente, apenas C, C++, Fortran e Julia fazem parte do petaflop club<sup>11</sup>.

<sup>7</sup> às vezes até mais rápido

<sup>8</sup> <https://julialang.org/blog/2012/02/why-we-created-julia/>

<sup>9</sup> <https://www.hpcwire.com/off-the-wire/julia-joins-petaflop-club/>

<sup>10</sup> um petaflop equivale a mil trilhões, ou um quatrilhão de operações com pontos flutuantes por segundo.

Mas velocidade não é tudo que Julia pode oferecer. A **facilidade de uso, o suporte a caracteres Unicode e ser uma linguagem que torna o compartilhamento de códigos algo muito simples** são algumas das características de Julia. Falaremos de todas essas qualidades nessa seção, mas focaremos no compartilhamento de códigos por enquanto.

O ecossistema de pacotes de Julia é algo único. Permite não só o compartilhamento de códigos, como também permite a criação de tipos definidos pelos usuários. Por exemplo, o `pandas` do Python usa seu próprio tipo de `DateTime` para lidar com datas. O mesmo ocorre com o pacote `lubridate` do `tidyverse` do R, que também define o seu tipo próprio de `datetime` para lidar com datas. Julia não precisa disso, ela tem todos os tipos e funcionalidades de datas incluídas na sua biblioteca padrão. Isso significa que outros pacotes não precisam se preocupar com datas. Eles só precisam estender os tipos de `DateTime` de Julia para novas funcionalidades, ao definirem novas funções, sem a necessidade de definirem novos tipos. O módulo `Dates` de Julia faz coisas incríveis, mas estamos nos adiantando. Primeiro, vamos falar de outras características de Julia.

### 2.3.1 Julia Versus outras linguagens de programação

Em Figure 2.1, uma representação altamente opinativa, dividimos as principais linguagens open source e de computação científica em um diagrama 2x2 com dois eixos: **Lento-Rápido** e **Fácil-Difícil**. Deixamos de fora as linguagens de código fechado, porque os benefícios são maiores quando permitimos que outras pessoas usem nossos códigos gratuitamente, assim como quando têm a liberdade para inspecionar elas mesmas o código fonte para sanar dúvidas e resolver problemas.

Consideramos que o C++ e o FORTRAN estão no quadrante Difícil e Rápido. Por serem linguagens estáticas que precisam de compilação, verificação de tipo e outros cuidados e atenção profissional, elas são realmente difíceis de aprender e lentas para prototipar. A vantagem é que elas são linguagens **muito rápidas**.

R e Python estão no quadrante Fácil e Lento. Elas são linguagens dinâmicas, que não são compiladas e executam em tempo de execução. Por causa disso, elas são fáceis de aprender e rápidas para prototipar. Claro que isso tem desvantagens: elas são linguagens **muito lentas**.

Julia é a única linguagem no quadrante Fácil e Rápido. Nós não conhecemos nenhuma linguagem séria que almejaria ser Difícil e Lenta, por isso esse quadrante está vazio.

**Julia é rápida! Muito rápida!** Foi desenvolvida para ser veloz desde o início.

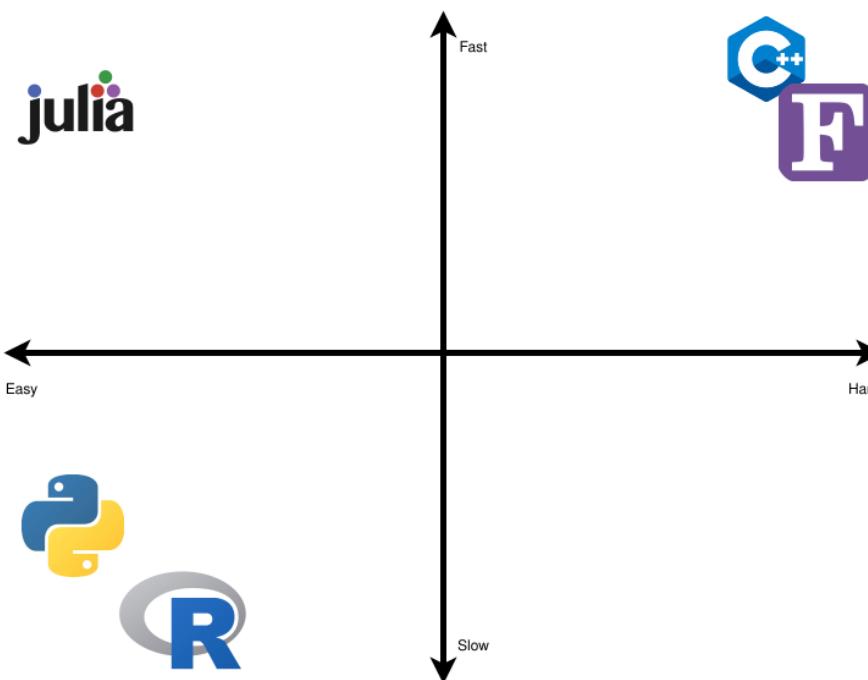


Figure 2.1: Comparações entre linguagens de computação científicas: logos para FORTRAN, C++, Python, R e Julia.

E alcança esse objetivo por meio do despacho múltiplo. Basicamente, a ideia é gerar códigos LLVM<sup>12</sup> muito eficientes. Códigos LLVM, também conhecidos como instruções LLVM, são de baixo-nível, ou seja, muito próximos das operações reais que seu computador está executando. Portanto, em essência, Julia converte o código que você escreveu — que é fácil de se ler — em código de máquina LLVM, que é muito difícil para humanos lerem, mas muito fácil para um computador. Por exemplo, se você definir uma função que recebe um argumento e passar um inteiro para a função, Julia criará um *MethodInstance especializado*. Na próxima vez que você passar um inteiro como argumento para a função, Julia buscará o *MethodInstance* criado anteriormente e redirecionará a execução a ele. Agora, o grande truque é que você também pode fazer isso dentro de uma função que chama uma outra função. Por exemplo, se certo tipo de dado é passado dentro da função *f* e *f* chama a função *g*, e se os tipos de dados passados para *g* são conhecidos e sempre os mesmos, então a função *g* gerada pode ser codificada de forma pré-definida pelo Julia na função *f*! Isso significa que Julia não precisa sequer buscar *MethodInstances* de *f* para *g*, pois o código consegue rodar de forma eficiente. A compensação aqui é que existem casos onde as suposições anteriores sobre a decodificação dos *MethodInstances* são invalidadas. Então, o *MethodInstance* precisa ser recriado, o que leva tempo. Além disso, a desvantagem é que também leva tempo para inferir o que pode ser codificado de forma pré-definida e o que não pode. Isso explica por que Julia demora para executar um código pela primeira vez: ela está otimizando

<sup>12</sup> LLVM significa “Máquina Virtual de Baixo-Nível,” ou, em inglês, Low Level Virtual Machine. Você pode encontrar mais sobre a LLVM no site: (<http://llvm.org>).

seu código em segundo-plano. A segunda e subsequentes execuções serão extremamente rápidas.

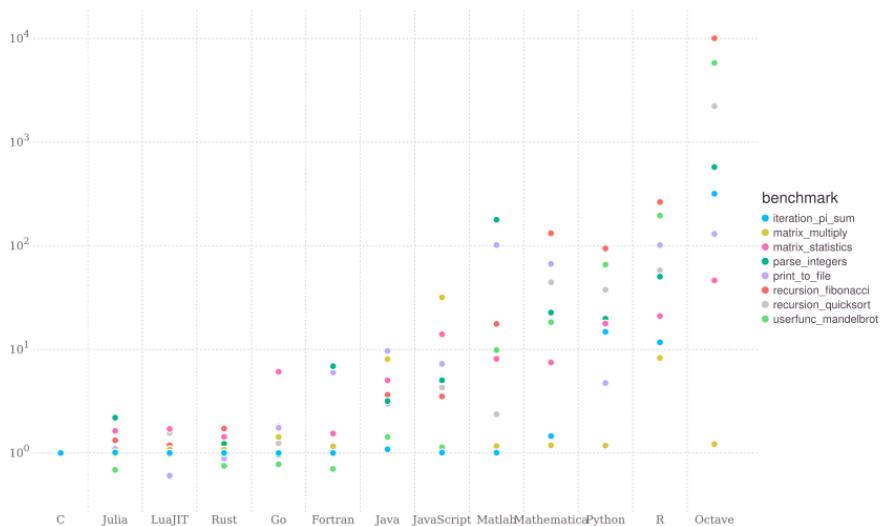
O compilador, por sua vez, faz o que ele faz de melhor: otimiza o código de máquina<sup>13</sup>. Você encontra benchmarks<sup>14</sup> para Julia e para outras linguagens aqui. Figure 2.2 foi retirado da seção de Benchmarks do site de Julia<sup>15,16</sup>. Como você pode perceber, Julia é **de fato** rápida.

<sup>13</sup> se quer saber mais sobre como Julia foi projetada, acesse [Bezanson et al. \(2017\)](#).

<sup>14</sup> <https://julialang.org/benchmarks/>

Figure 2.2: Julia versus outras linguagens de programação. Desertos acima não incluem o tempo de compilação.

<sup>15</sup> <https://julialang.org/benchmarks/>



Nós realmente acreditamos em Julia. Caso contrário, não teríamos escrito este livro. Nós acreditamos que Julia é **o futuro da computação científica e da análise de dados científicos**. Ela permite que o usuário desenvolva códigos rápidos e poderosos com uma sintaxe simples. Normalmente, pesquisadores desenvolvem códigos usando linguagens fáceis, mas muito lentas. Uma vez que o código rode corretamente e cumpra seus objetivos, aí começa o processo de conversão do código para uma linguagem rápida, porém difícil. Esse é o “problema das duas linguagens” e discutiremos ele melhor a seguir.

### 2.3.2 O Problema das Duas Linguagens

O “Problema das Duas Linguagens” é bastante comum na computação científica, quando um pesquisador concebe um algoritmo, ou quando desenvolve uma solução para um problema, ou mesmo quando realiza algum tipo de análise. Em seguida, a solução é prototipada em uma linguagem fácil de codificar (como Python ou R). Se o protótipo funciona, o pesquisador codifica em uma linguagem rápida que, em geral, não é fácil de prototipar (como C++ ou FORTRAN). Assim, temos duas linguagens envolvidas no processo de desenvolvimento de uma nova solução. Uma que é fácil de prototipar, mas não

é adequada para implementação (principalmente por ser lenta). E outra que não é tão simples de codificar e, consequentemente, não é fácil de prototipar, mas adequada para implementação porque é rápida. Julia evita esse tipo de situação por ser a **a mesma linguagem que você prototipa (fácil de usar) e implementa a solução (rápida)**.

Além disso, Julia permite que você use **caracteres Unicode como variáveis ou parâmetros**. Isso significa que não é preciso mais usar `sigma` ou `sigma_i`: ao invés disso use apenas  $\sigma$  ou  $\sigma_i$  como você faria em notação matemática. Quando você vê o código de um algoritmo ou para uma equação matemática, você vê quase a mesma notação e expressões idiomáticas. Chamamos esse recurso poderoso de **“Relação Um para Um entre Código e Matemática”**.

Acreditamos que o “Problema das Duas Linguagens” e a “Relação Um para Um entre Código e Matemática” são melhor descritos por um dos criadores de Julia, Alan Edelman, em um TEDx Talk<sup>17</sup> ([TEDx Talks, 2020](https://youtu.be/qGW0GT1rCvs)).

<sup>17</sup> <https://youtu.be/qGW0GT1rCvs>

### 2.3.3 Despacho Múltiplo

Despacho múltiplo é um recurso poderoso que nos permite estender funções existentes ou definir comportamento personalizado e complexo para novos tipos. Suponha que você queira definir dois novos `structs` para denotar dois animais diferentes:

```
abstract type Animal end
struct Fox <: Animal
    weight::Float64
end
struct Chicken <: Animal
    weight::Float64
end
```

Basicamente, isso diz “defina uma raposa, que é um animal” e “defina uma galinha, que é um animal.” Em seguida, podemos ter uma raposa chamada Fiona e uma galinha chamada Big Bird.

```
fiona = Fox(4.2)
big_bird = Chicken(2.9)
```

A seguir, queremos saber quanto elas pesam juntas, para o qual podemos escrever uma função:

```
combined_weight(A1::Animal, A2::Animal) = A1.weight + A2.weight
```

```
combined_weight (generic function with 1 method)
```

E queremos saber se elas vão se dar bem. Uma maneira de implementar isso é usar condicionais:

```
function naive_trouble(A::Animal, B::Animal)
    if A isa Fox && B isa Chicken
        return true
    elseif A isa Chicken && B isa Fox
        return true
    elseif A isa Chicken && B isa Chicken
        return false
    end
end
```

---

```
naive_trouble (generic function with 1 method)
```

---

Agora, vamos ver se deixar Fiona e Big Bird juntas daria problema:

```
naive_trouble(fiona, big_bird)
```

---

```
true
```

---

OK, isso parece correto. Escrevendo a função `naive_trouble` parece ser o suficiente. No entanto, usar despacho múltiplo para criar uma nova função `trouble` pode ser benéfico. Vamos criar novas funções:

```
trouble(F::Fox, C::Chicken) = true
trouble(C::Chicken, F::Fox) = true
trouble(C1::Chicken, C2::Chicken) = false
```

---

```
trouble (generic function with 3 methods)
```

---

Depois da definição dos métodos, `trouble` fornece o mesmo resultado que `naive_trouble` ↵. Por exemplo:

```
trouble(fiona, big_bird)
```

---

```
true
```

---

E deixar Big Bird sozinha com outra galinha chamada Dora também é bom

```
dora = Chicken(2.2)
trouble(dora, big_bird)
```

---

```
false
```

---

Portanto, neste caso, a vantagem do despacho múltiplo é que você pode apenas declarar tipos e Julia encontrará o método correto para seus tipos. Ainda mais, para muitos casos quando o despacho múltiplo é usado dentro do código, o compilador Julia irá realmente otimizar as chamadas de função. Por exemplo, poderíamos escrever:

```
function trouble(A::Fox, B::Chicken, C::Chicken)
    return trouble(A, B) || trouble(B, C) || trouble(C, A)
end
```

Dependendo do contexto, Julia pode otimizar isso para:

```
function trouble(A::Fox, B::Chicken, C::Chicken)
    return true || false || true
end
```

porque o compilador **sabe** que `A` é a raposa, `B` é a galinha e então isso pode ser substituído pelo conteúdo do método `trouble(F::Fox, C::Chicken)`. O mesmo vale para `trouble(C1::Chicken, C2::Chicken)`. Em seguida, o compilador pode otimizar isso para:

```
function trouble(A::Fox, B::Chicken, C::Chicken)
    return true
end
```

Outro benefício do despacho múltiplo é que quando outra pessoa chega e quer comparar os animais existentes com seu animal, uma zebra por exemplo, é possível. Em seu pacote, eles podem definir um Zebra:

```
struct Zebra <: Animal
    weight::Float64
end
```

e também como as interações com os animais existentes seriam:

```
trouble(F::Fox, Z::Zebra) = false
trouble(Z::Zebra, F::Fox) = false
trouble(C::Chicken, Z::Zebra) = false
trouble(Z::Zebra, F::Fox) = false
```

```
trouble (generic function with 6 methods)
```

Agora, podemos ver se Marty (nossa zebra) está a salvo com Big Bird:

```
marty = Zebra(412)
trouble(big_bird, marty)
```

---

```
false
```

---

Ainda melhor, conseguimos calcular **o peso combinado de zebras e outros animais sem definir qualquer função extra**:

```
combined_weight(big_bird, marty)
```

---

[414.9](#)

---

Então, em resumo, o código que foi escrito pensando apenas para Raposa e Galinha funciona para tipos que **ele nunca tinham visto!** Na prática, isso significa que Julia facilita o reuso do código de outros projetos.

Se você está tão animado quanto nós com o despacho múltiplo, aqui estão mais dois exemplos aprofundados. O primeiro é uma rápida e elegante implementação de um vetor one-hot<sup>18</sup> por [Storopoli \(2021\)](#). O segundo é uma entrevista com Christopher Rackauckas<sup>19</sup> no canal do YouTube de Tanmay Bakshi<sup>20</sup> (assista do minuto 35:07 em diante) ([tanmay bakshi, 2021](#)). Chris explica que, enquanto utilizava o `DifferentialEquations.jl`<sup>21</sup>, um pacote que ele desenvolveu e mantém atualmente, um usuário registrou um problema que seu solucionador de Equações Diferenciais Ordinais (EDO) com quaternions baseado em GPU não funcionava. Chris ficou bastante surpreso com este pedido, já que ele não esperava que alguém combinasse cálculos da GPU com quaternions e resolvendo EDOs. Ele ficou ainda mais surpreso quando descobriu que o usuário cometeu um pequeno erro e que tudo funcionou. A maior parte do mérito é devido ao múltiplo despacho e alto compartilhamento de código/tipos definidos pelo usuário.

Para concluir, pensamos que o despacho múltiplo é melhor explicado por um dos criadores de Julia: Stefan Karpinski na JuliaCon 2019<sup>22</sup>.

<sup>18</sup> [https://storopoli.io/Bayesian-Julia/pages/1\\_why\\_Julia/#example\\_one-hot\\_vector](https://storopoli.io/Bayesian-Julia/pages/1_why_Julia/#example_one-hot_vector)

<sup>19</sup> <https://www.chrisrakaukas.com/>

<sup>20</sup> <https://youtu.be/moyPlhvW4Nk?t=2107>

<sup>21</sup> <https://diffeq.sciml.ai/dev/>

<sup>22</sup> <https://youtu.be/kc9HwsxE1OY>

## 2.4 Julia Por Aí

Em Section 2.3, explicamos por que achamos que Julia é uma linguagem de programação única. Mostramos exemplos simples sobre os principais recursos de Julia. Se você quiser se aprofundar em como Julia está sendo usada, temos alguns **casos de uso interessantes**:

1. NASA usa Julia em um supercomputador que analisa “o maior lote de planetas do tamanho da Terra já encontrado”<sup>23</sup> e alcançou uma extraordinária otimização que tornou a execução **1.000x mais rápida** para catalogar 188 milhões de objetos astronômicos em 15 minutos.

<sup>23</sup> <https://exoplanets.nasa.gov/news/1669/ven-rocky-trappist-1-planets-may-be-made-of-similar-stuff/>

2. A Aliança para a Modelagem Climática (Climate Model Alliance - CliMa)<sup>24</sup> usa Julia para **modelar o clima na GPU e CPU**. Lançado em 2018 em colaboração com pesquisadores da Caltech, do NASA Jet Propulsion Laboratory, e da Naval Postgraduate School, a CliMA está utilizando o progresso recente da ciência computacional para desenvolver um modelo do sistema terrestre que pode prever secas, ondas de calor e chuva com precisão e velocidade sem precedentes.
3. O Departamento de Aviação Federal dos Estados Unidos (US Federal Aviation Administration - FAA) está desenvolvendo um **Sistema de Prevenção de Colisões Aéreas (Airborne Collision Avoidance System - ACAS-X)** usando Julia<sup>25</sup>. Esse é um bom exemplo do “Problema das Duas Linguagens” (see Section 2.3). Soluções anteriores usavam Matlab para desenvolver os algoritmos e C++ para uma implementação mais rápida. Agora, FAA usa uma única linguagem para tudo isso: Julia.
4. **Aceleração de 175x** para modelos de farmacologia da Pfizer usando GPUs em Julia<sup>26</sup>. Foi apresentado como um poster<sup>27</sup> na 11ª American Conference of Pharmacometrics (ACoP11) e ganhou um prêmio de qualidade<sup>28</sup>.
5. O Subsistema de Controle de Atitude e Órbita (Attitude and Orbit Control Subsystem - AOCS) do satélite brasileiro Amazonia-1 é **escrito 100% em Julia**<sup>29</sup> por Ronan Arraes Jardim Chagas (<https://ronanarraes.com/>).
6. O Banco Nacional de Desenvolvimento Econômico e Social (BNDES) do Brasil abandonou uma solução paga e optou pela modelagem em Julia (que é código aberto) e teve uma otimização de velocidade de execução em um fator de **10x**.<sup>30</sup>

Se isso não for suficiente, existem mais estudos de caso em Julia Computing website<sup>31</sup>.

<sup>25</sup> <https://youtu.be/19zm1Fn0S9M>

<sup>26</sup> <https://juliacomputing.com/case-studies/pfizer/>

<sup>27</sup> [https://chrisrackauckas.com/assets/Posters/ACoP11\\_Poster\\_Abstracts\\_2020.pdf](https://chrisrackauckas.com/assets/Posters/ACoP11_Poster_Abstracts_2020.pdf)

<sup>28</sup> <https://web.archive.org/web/20210121164011/https://www.go-acop.org/abstract-awards>

<sup>29</sup> <https://discourse.julialang.org/t/julia-and-the-satellite-amazonia-1/57541>

<sup>30</sup> <https://youtu.be/NY0HcGqHj3g>

<sup>31</sup> <https://juliacomputing.com/case-studies/>



# 3 *Básico de Julia*

**OBSERVAÇÃO:** Neste capítulo, descreveremos o básico de Julia como linguagem de programação. Por favor, note que isso não é *estritamente necessário* para você usar Julia como uma ferramenta de manipulação e visualização de dados. Ter um conhecimento básico de Julia definitivamente o tornará mais *eficaz* e *eficiente* no uso de Julia. No entanto, se você preferir começar imediatamente, pode pular para Section 4 e aprender sobre dados tabulares em `DataFrames.jl`.

Aqui, vamos trazer uma visão mais geral sobre a linguagem Julia, *não* algo aprofundado. Se você já está familiarizado e confortável com outras linguagens de programação, nós encorajamos você a ler a documentação de Julia (<https://docs.julialang.org/>). Os documentos são um excelente recurso para você se aprofundar em Julia. Eles cobrem todos os fundamentos e casos extremos, mas podem ser complicados, especialmente se você não estiver familiarizado com a leitura de documentação de software.

Cobriremos o básico de Julia. Imagine que Julia é um carro sofisticado repleto de recursos, como um Tesla novo. Vamos apenas explicar a você como “dirigir o carro, estacioná-lo e como navegar no trânsito.” Se você quer saber o que “todos os botões no volante e painel fazem,” este não é o livro que você está procurando.

## 3.1 *Sintaxe da linguagem*

Julia é uma **linguagem de tipagem dinâmica** com um compilador just-in-time. Isso significa que você não precisa compilar seu programa antes de executá-lo, como precisaria fazer com C++ ou FORTRAN. Em vez disso, Julia pegará seu código, adivinhará os tipos quando necessário e compilará partes do código antes de executá-lo. Além disso, você não precisa especificar explicitamente cada tipo. Julia vai inferir os tipos para você na hora.

As principais diferenças entre Julia e outras linguagens dinâmicas como R e Python são: Primeiro, Julia **permite ao usuário especificar declarações de tipo**. Você já viu algumas declarações de tipo em *Por que Julia?* (Section 2): eles são aqueles dois pontos duplos :: que às vezes vem depois das variáveis. No entanto, se você não quiser especificar o tipo de suas variáveis ou funções, Julia terá o prazer de inferir (adivinhar) para você.

Em segundo lugar, Julia permite que os usuários definam o comportamento da função de acordo com combinações diversas de tipos de argumento por meio do despacho múltiplo. Também falamos sobre despacho múltiplo em Section 2.3. Por meio do despacho múltiplo, nós definimos um comportamento diferente de uma função para um determinado tipo quando escrevemos uma nova função com o mesmo nome da função anterior, mas cuja assinatura contém a especificação deste tipo em seus argumentos.

### 3.1.1 Variáveis

As variáveis são valores que você diz ao computador para armazenar com um nome específico, para que você possa recuperar ou alterar seu valor posteriormente. Julia tem diversos tipos de variáveis, mas, em ciência de dados, usamos principalmente:

- Números inteiros: `Int64`
- Números reais: `Float64`
- Booleanas: `Bool`
- Strings: `String`

Inteiros e números reais são armazenados usando 64 bits por padrão, é por isso que eles têm o sufixo `64` no nome do tipo. Se você precisar de mais ou menos precisão, existem os tipos `Int8` ou `Int128`, por exemplo, nos quais um maior número significa uma maior precisão. Na maioria das vezes, isso não será um problema, então você pode simplesmente seguir os padrões.

Criamos novas variáveis escrevendo o nome da variável à esquerda e seu valor à direita, e no meio usamos o operador de atribuição `=`. Por exemplo:

```
name = "Julia"
age = 9
```

---

9

---

Observe que a saída de retorno da última instrução (`idade`) foi impressa no console. Aqui, estamos definindo duas novas variáveis: `nome` e `idade`. Podemos recuperar seus valores digitando os nomes dados na atribuição:

```
name
```

---

Julia

---

Se quiser definir novos valores para uma variável existente, você pode repetir os passos realizados durante a atribuição. Observe que Julia agora substituirá o valor anterior pelo novo. Suponho que o aniversário de Julia já passou e agora fez 10 anos:

```
age = 10
```

---

```
10
```

---

Podemos fazer o mesmo com `name`. Suponha que Julia tenha ganho alguns títulos devido à sua velocidade incrível. Mudaríamos a variável `name` para o novo valor:

```
name = "Julia Rápidus"
```

---

```
Julia Rápidus
```

---

Também podemos fazer operações em variáveis como adição ou divisão. Vamos ver quantos anos Julia tem, em meses, multiplicando `age` por 12:

```
12 * age
```

---

```
120
```

---

Podemos inspecionar os tipos das variáveis usando a função `typeof`:

```
typeof(age)
```

---

```
Int64
```

---

A próxima pergunta então se torna: “O que mais posso fazer com os inteiros?” Há uma função boa e útil, `methodswith` que expõe todas as funções disponíveis, junto com sua assinatura, para um certo tipo. Aqui, vamos restringir a saída às primeiras 5 linhas:

```
first(methodswith(Int64), 5)
```

---

```
[1] logmvbeta(p::Int64, a::T, b::T) where T<:Real in StatsFuns at /home/runner/.
    ↪ julia/packages/StatsFuns/mQJB7/src/misc.jl:22
[2] logmvbeta(p::Int64, a::Real, b::Real) in StatsFuns at /home/runner/.julia/
    ↪ packages/StatsFuns/mQJB7/src/misc.jl:23
[3] logvgamma(p::Int64, a::Real) in StatsFuns at /home/runner/.julia/packages/
    ↪ StatsFuns/mQJB7/src/misc.jl:8
```

```
[4] read(t::HTTP.ConnectionPool.Transaction, nb::Int64) in HTTP.ConnectionPool
    ↪at /home/runner/.julia/packages/HTTP/aTjcj/src/ConnectionPool.jl:232
[5] write(ctx::MbedTLS.MD, i::Union{Float16, Float32, Float64, Int128, Int16,
    ↪Int32, Int64, UInt128, UInt16, UInt32, UInt64}) in MbedTLS at /home/
    ↪runner/.julia/packages/MbedTLS/lqmet/src/md.jl:140
```

### 3.1.2 Tipos definidos pelo usuário

Ter apenas variáveis à disposição, sem qualquer forma de hierarquia ou relacionamento não é o ideal. Em Julia, podemos definir essa espécie de dado estruturado com um **struct** (também conhecido como tipo composto). Dentro de cada **struct**, você pode especificar um conjunto de campos **fields**. Eles diferem dos tipos primitivos (por exemplo, inteiro e flutuantes) que já são definidos por padrão dentro do núcleo da linguagem Julia. Já que a maioria dos **struct** → são definidos pelo usuário, eles são conhecidos como tipos definidos pelo usuário.

Por exemplo, vamos criar um **struct** para representar linguagens de programação científica em código aberto. Também definiremos um conjunto de campos junto com os tipos correspondentes dentro do **struct**:

```
struct Language
    name :: String
    title :: String
    year_of_birth :: Int64
    fast :: Bool
end
```

Para inspecionar os nomes dos campos, você pode usar o **fieldnames** e passar o **struct** desejado como argumento:

```
fieldnames(Language)
```

```
(:name, :title, :year_of_birth, :fast)
```

Para usar os **struct**, devemos instanciar instâncias individuais (ou “objetos”), cada um com seus próprios valores específicos para os campos definidos dentro do **struct**. Vamos instanciar duas instâncias, uma para Julia e outra para Python:

```
julia = Language("Julia", "Rapidus", 2012, true)
python = Language("Python", "Letargicus", 1991, false)
```

```
Language("Python", "Letargicus", 1991, false)
```

Algo importante de se notar com os `struct` é que não podemos alterar seus valores uma vez que são instanciados. Podemos resolver isso com `mutable struct`. Além disso, observe que objetos mutáveis geralmente serão mais lentos e mais propensos a erros. Sempre que possível, faça com que tudo seja *imutável*. Vamos criar uma `mutable struct`.

```
mutable struct MutableLanguage
    name::String
    title::String
    year_of_birth::Int64
    fast::Bool
end

julia MutableLanguage("Julia", "Rapidus", 2012, true)
```

---

```
MutableLanguage("Julia", "Rapidus", 2012, true)
```

---

Suponha que queremos mudar o campo título do objeto `julia Mutable`. Agora podemos fazer isso já que `julia Mutable` é um `mutable struct` instanciado:

```
julia Mutable.title = "Python Obliteratus"

julia Mutable
```

---

```
MutableLanguage("Julia", "Python Obliteratus", 2012, true)
```

---

### 3.1.3 Operadores booleanos e comparações numéricas

Agora que cobrimos os tipos, podemos passar para os operadores booleanos e a comparação numérica.

Nós temos três operadores booleanos em Julia:

- `!:` NOT
- `&&:` AND
- `||:` OR

Aqui estão exemplos com alguns deles:

```
!true
```

---

```
false
```

---

```
(false && true) || (!false)
```

true

```
(6 isa Int64) && (6 isa Real)
```

true

Com relação à comparação numérica, Julia tem três tipos principais de comparações:

1. **Igualdade**: ou algo é *igual* ou *não igual* em relação a outro

- == “igual”
- != ou ≠ “não igual”

2. **Menor que**: ou algo é *menor que* ou *menor ou igual a*

- < “menor que”
- <= ou ≤ “menor ou igual a”

3. **Maior que**: ou algo é *maior que* ou *maior ou igual a*

- > “maior que”
- >= ou ≥ “maior ou igual a”

Aqui temos alguns exemplos:

```
1 == 1
```

true

```
1 >= 10
```

false

As comparações funcionam até mesmo entre tipos diferentes:

```
1 == 1.0
```

true

Também podemos misturar e combinar operadores booleanos com comparações numéricas:

```
(1 != 10) || (3.14 <= 2.71)
```

```
true
```

### 3.1.4 Funções

Agora que já sabemos como definir variáveis e tipos personalizados como `struct` →, vamos voltar nossa atenção para as **funções**. Em Julia, uma função **mapeia os valores de seus argumentos para um ou mais valores de retorno**. A sintaxe básica é assim:

```
function function_name(arg1, arg2)
    result = stuff with the arg1 and arg2
    return result
end
```

A declaração de funções começa com a palavra-chave `function` seguida do nome da função. Então, entre parênteses (), nós definimos os argumentos separados por uma vírgula ,. Dentro da função, especificamos o que queremos que Julia faça com os parâmetros que fornecemos. Todas as variáveis que definimos dentro de uma função são excluídas após o retorno da função. Isso é bom porque é como se realizasse uma limpeza automática. Depois que todas as operações no corpo da função forem concluídas, instruímos Julia a retornar o resultado com o comando `return`. Por fim, informamos a Julia que a definição da função terminou com a palavra-chave `end`.

Existe também a maneira compacta de definição de funções por meio da **forma de atribuição**:

```
f_name(arg1, arg2) = stuff with the arg1 and arg2
```

É a **mesma função** que antes, mas definida de uma forma diferente, mais compacta. Como regra geral, quando seu código pode caber facilmente em uma linha de até 92 caracteres, a forma compacta é adequada. Caso contrário, basta usar o formato mais longo com a palavra-chave `function`. Vamos mergulhar em alguns exemplos.

### Criando novas funções

Vamos criar uma nova função que soma números:

```
function add_numbers(x, y)
    return x + y
end
```

---

```
add_numbers (generic function with 1 method)
```

---

Agora, podemos usar nossa função `add_numbers`:

```
add_numbers(17, 29)
```

---

```
46
```

---

E ela também funciona com números reais (também chamados em programação de números de ponto-flutuante ou, de forma mais curta, com o jargão “floats”):

```
add_numbers(3.14, 2.72)
```

---

```
5.86
```

---

Além disso, podemos definir comportamentos especializados para nossa função, por meio da especificação de declarações de tipo. Suponha que queremos ter uma função `round_number` que se comporta de maneira diferente se seu argumento for um `Float64` ou `Int64`:

```
function round_number(x::Float64)
    return round(x)
end

function round_number(x::Int64)
    return x
end
```

---

```
round_number (generic function with 2 methods)
```

---

Podemos ver que ela é uma função com múltiplos métodos:

```
methods(round_number)
```

---

```
round_number(x::Float64) in Main at none:1
```

---



---

```
round_number(x::Int64) in Main at none:5
```

---

Mas há um problema: o que acontece se quisermos arredondar um float de 32 bits, `Float32`? Ou um inteiro de 8 bits, `Int8`?

Se você quiser que algo funcione em todos os tipos de float e inteiros, você pode usar um **tipo abstrato** na assinatura de tipo, como `AbstractFloat` ou `Integer`:

```
function round_number(x::AbstractFloat)
    return round(x)
end
```

---

`round_number` (generic function with 3 methods)

---

Agora, funcionará da forma esperada com qualquer tipo de float:

```
x_32 = Float32(1.1)
round_number(x_32)
```

---

1.0

---

**OBSERVAÇÃO:** Podemos inspecionar tipos com as funções `supertypes` e `subtypes` ↵.

Vamos voltar ao nosso `struct Language` que definimos anteriormente. Será um exemplo de despacho múltiplo. Vamos estender a função `Base.show` que imprime a saída de tipos instanciados e de `struct`.

Por padrão, uma `struct` tem um output básico, que você pode observar do caso do `python`. Podemos definir um novo método `Base.show` para nosso tipo `Language` ↵, assim temos uma boa impressão para nossas instâncias de linguagens de programação. Queremos comunicar claramente os nomes, títulos e idades em anos das linguagens de programação. A função `Base.show` aceita como argumentos um tipo `IO` chamado `io` seguido pelo tipo para o qual você deseja definir o comportamento personalizado:

```
Base.show(io::IO, l::Language) = print(
    io, l.name, " ",
    2021 - l.year_of_birth, ", years old, ",
    "has the following titles: ", l.title
)
```

Agora, vamos ver como o output de `python` será:

```
python
```

---

Python 30, years old, has the following titles: Letargicus

---

## Múltiplos Valores de Retorno

Uma função também pode retornar dois ou mais valores. Veja a nova função `add_multiply` abaixo:

```
function add_multiply(x, y)
    addition = x + y
    multiplication = x * y
    return addition, multiplication
end
```

---

```
add_multiply (generic function with 1 method)
```

---

Nesse caso, podemos fazer duas coisas:

1. Podemos, analogamente aos valores de retorno, definir duas variáveis para conter os valores de retorno da função, uma para cada valor de retorno:

```
return_1, return_2 = add_multiply(1, 2)
return_2
```

---

2

---

2. Ou podemos definir apenas uma variável para manter os valores de retorno da função e acessá-los com `first` ou `last`:

```
all_returns = add_multiply(1, 2)
last(all_returns)
```

---

2

---

## Argumentos de Palavra-Chave

Algumas funções podem aceitar argumentos de palavra-chave ao invés de argumentos posicionais. Esses argumentos são como argumentos comuns, exceto pelo fato de serem definidos após os argumentos de função regulares e separados por um ponto e vírgula `;`. Por exemplo, vamos definir uma função `logarithm` que por padrão usa base  $e$  ( $2.718281828459045$ ) como um argumento

de palavra-chave. Perceba que aqui estamos usando o tipo abstrato `Real` para que possamos cobrir todos os tipos derivados de `Integer` e `AbstractFloat`, dado que ambos são subtipos de `Real`:

```
AbstractFloat <: Real && Integer <: Real
```

---

```
true
```

---

```
function logarithm(x::Real; base::Real=2.7182818284590)
    return log(base, x)
end
```

---

```
logarithm (generic function with 1 method)
```

---

Funciona sem especificar o argumento `base` já que fornecemos um **valor de argumento padrão** na declaração da função:

```
logarithm(10)
```

---

```
2.3025850929940845
```

---

E também com o argumento de palavra-chave `base` diferente de seu valor padrão:

```
logarithm(10; base=2)
```

---

```
3.3219280948873626
```

---

## Funções Anônimas

Muitas vezes não nos importamos com o nome da função e queremos criar uma rapidamente. O que precisamos é das **funções anônimas**. Elas são muito usadas no fluxo de trabalho de ciência de dados em Julia. Por exemplo, quando usamos `DataFrames.jl` (Section 4) ou `Makie.jl` (Section 5), às vezes precisamos de uma função temporária para filtrar dados ou formatar os rótulos de um gráfico. É aí que usamos as funções anônimas. Elas são especialmente úteis quando não queremos criar uma função e uma instrução simples seria o suficiente.

A sintaxe é simples. Nós usamos o operador `->`. À esquerda do `->` definimos o nome do parâmetro. E à direita do `->` definimos quais operações queremos realizar no parâmetro que definimos à esquerda de `->`. Segue um exemplo. Suponha que queremos desfazer a transformação de log usando uma expo-  
nenciação:

```
map(x -> 2.7182818284590^x, logarithm(2))
```

---

2.0

---

Aqui, estamos usando a função `map` para mapear convenientemente a função anônima (primeiro argumento) para `logarithm(2)` (segundo argumento). Como resultado, obtemos o mesmo número, porque o logaritmo e a exponenciação são inversos (pelo menos na base que escolhemos – 2.7182818284590)

### 3.1.5 Condicional If-Else-Elseif

Na maioria das linguagens de programação, o usuário tem permissão para controlar o fluxo de execução do computador. Dependendo da situação, queremos que o computador faça uma coisa ou outra. Em Julia, podemos controlar o fluxo de execução com as palavras-chave `if`, `elseif` e `else`. Estas são conhecidas como declarações condicionais.

A palavra-chave `if` comanda Julia a avaliar uma expressão `e`, dependendo se ela é verdadeira (`true`) ou falsa (`false`), a executar certas partes do código. Podemos combinar várias condições `if` com a palavra-chave `elseif` para um fluxo de controle complexo. Assim, podemos definir uma parte alternativa a ser executada se qualquer coisa dentro de `if` ou `elseif` for avaliada como `true`. Esse é o propósito da palavra-chave `else`. Finalmente, como em todos os operadores de palavra-chave que vimos anteriormente, devemos informar a Julia quando a declaração condicional for concluída com a palavra-chave `end`.

Aqui, temos um exemplo com todas as palavras-chave `if-elseif-else`:

```
a = 1
b = 2

if a < b
    "a is less than b"
elseif a > b
    "a is greater than b"
else
    "a is equal to b"
end
```

---

a is less than b

---

Podemos até envelopar isso em uma função chamada `compare`:

```

function compare(a, b)
    if a < b
        "a is less than b"
    elseif a > b
        "a is greater than b"
    else
        "a is equal to b"
    end
end

compare(3.14, 3.14)

```

a is equal to b

### 3.1.6 Laço For

O clássico laço for em Julia segue uma sintaxe semelhante à das declarações condicionais. Você começa com a palavra-chave, nessa caso `for`. Em seguida, você especifica o que Julia deve iterar sobre (ou, no jargão, “loopar”), p. ex., uma sequência. Além disso, como em tudo mais, você deve terminar com a palavra-chave `end`.

Então, para fazer Julia imprimir todos os números de 1 a 10, você pode usar o seguinte laço for:

```

for i in 1:10
    println(i)
end

```

### 3.1.7 Laço While

O laço while é uma mistura das declarações condicionais anteriores com os laços for. Aqui, o laço é executado toda vez que a condição é avaliada como `true`. A sintaxe segue a mesma forma da anterior. Começamos com a palavra-chave `while`, seguido por uma declaração que é avaliada em `true` ou `false`. Como de costume, devemos terminar com a palavra-chave `end`.

Segue um exemplo:

```

n = 0

while n < 3
    global n += 1
end

```

n

3

Como pode ver, devemos usar a palavra-chave `global`. Isso se deve ao **escopo de variável**. Variáveis definidas dentro das declarações condicionais, laços e funções existem apenas dentro delas. Isso é conhecido como o *escopo* da variável. Aqui, precisamos avisar Julia que o `n` dentro do laço `while` está no escopo global por meio do uso da palavra-chave `global`.

Por fim, também usamos o operador `+=` que é uma boa abreviatura para `n = n ↪+ 1`.

### 3.2 Estruturas Nativas de Dados

Julia possui diversas estruturas de dados nativas. Elas são abstrações de dados que representam alguma forma de dado estruturado. Vamos cobrir os mais usados. Eles contém dados homogêneos ou heterogêneos. Uma vez que são coleções, podemos iterar sobre eles com os laços `for`.

Nós cobriremos `String`, `Tuple`, `NamedTuple`, `UnitRange`, `Arrays`, `Pair`, `Dict`, `Symbol`.

Quando você se depara com uma estrutura de dados em Julia, você pode encontrar métodos que a aceitam como um argumento por meio da função `methodswith ↪`. Em Julia, a distinção entre métodos e funções é a seguinte: Cada função pode ter múltiplos métodos, como mostramos anteriormente. A função `methodswith ↪` é boa de se ter por perto. Vejamos o que podemos fazer com uma `String`, por exemplo:

```
first(methodswith(String), 5)
```

---

```
[1] write(fp::FilePathsBase.SystemPath, x::Union{String, Vector{UInt8}}) in
    ↪FilePathsBase at /home/runner/.julia/packages/FilePathsBase/9kSEL/src/
    ↪system.jl:380
[2] write(fp::FilePathsBase.SystemPath, x::Union{String, Vector{UInt8}}, mode)
    ↪in FilePathsBase at /home/runner/.julia/packages/FilePathsBase/9kSEL/src/
    ↪system.jl:380
[3] write(iod::HTTP.DebugRequest.IODebug, x::String) in HTTP.DebugRequest at /
    ↪home/runner/.julia/packages/HTTP/aTjcj/src/IODebug.jl:38
[4] write(buffer::FilePathsBase.FileBuffer, x::String) in FilePathsBase at /home
    ↪/runner/.julia/packages/FilePathsBase/9kSEL/src/buffer.jl:85
[5] write(io::IO, s::Union{SubString{String}, String}) in Base at strings/io.jl:
    ↪244
```

---

### 3.2.1 Fazendo Broadcasting de Operadores e Funções

Antes de mergulharmos nas estruturas de dados, precisamos conversar sobre broadcasting (também conhecido como *vetorização*) e o operador “dot” ..

Podemos vetorizar operações matemáticas como \* (multiplicação) ou + (adição) usando o operador dot. Por exemplo, vetorizar adição implica em mudar + para .+:

```
[1, 2, 3] .+ 1
```

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

Também funciona automaticamente com funções. (Tecnicamente, as operações matemáticas, ou operadores infixos, também são funções, mas isso não é tão importante saber.) Lembra da nossa função `logarithm`?

```
logarithm.([1, 2, 3])
```

```
[0.0, 0.6931471805599569, 1.0986122886681282]
```

### Funções com exclamação !

É uma convenção de Julia acrescentar uma exclamação ! a nomes de funções que modificam um ou mais de seus argumentos. Esta convenção avisa o usuário que a função **não é pura**, ou seja, que tem *efeitos colaterais*. Uma função com efeitos colaterais é útil quando você deseja atualizar uma grande estrutura de dados ou coleção de variáveis sem ter toda a sobrecarga da criação de uma nova instância.

Por exemplo, podemos criar uma função que adiciona 1 a cada elemento de um vetor `v`:

```
function add_one!(v)
    for i in 1:length(v)
        v[i] += 1
    end
    return nothing
end
```

```
my_data = [1, 2, 3]
```

```
add_one!(my_data)
```

```
my_data
```

---

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

---

### 3.2.2 String

**Strings** são representadas delimitadas por aspas duplas:

```
typeof("This is a string")
```

---

```
String
```

---

Também podemos escrever uma string multilinha:

```
text = "
This is a big multiline string.
As you can see.
It is still a String to Julia.
"
```

---

```
This is a big multiline string.
As you can see.
It is still a String to Julia.
```

---

Mas, geralmente, é mais claro usar aspas triplas:

```
s = """
This is a big multiline string with a nested "quotation".
As you can see.
It is still a String to Julia.
"""
```

---

```
This is a big multiline string with a nested "quotation".
As you can see.
It is still a String to Julia.
```

---

Ao usar crases triplas, a tabulação e o marcador de nova linha no início são ignorados por Julia. Isso melhora a legibilidade do código porque você pode indentar o bloco em seu código-fonte sem que esses espaços acabem em sua string.

## Concatenação de Strings

Uma operação comum de string é a **concatenação de string**. Suponha que você queira construir uma nova string que é a concatenação de duas ou mais strings. Isso é realizado em Julia com o operador `*` ou a função `join`. Este símbolo pode soar como uma escolha estranha e realmente é. Por enquanto, muitas bases de código em Julia estão usando este símbolo, então ele permanecerá na linguagem. Se você estiver interessado, pode ler uma discussão de 2015 sobre isso em <https://github.com/JuliaLang/julia/issues/11030>.

```
hello = "Hello"
goodbye = "Goodbye"

hello * goodbye
```

---

```
HelloGoodbye
```

---

Como você pode ver, está faltando um espaço entre `hello` e `goodbye`. Poderíamos concatenar uma string adicional `" "` com `*`, mas isso seria complicado para mais de duas strings. É onde a função `join` vem a calhar. Nós apenas passamos como argumentos as strings dentro dos colchetes `[]` e, em seguida, o separador:

```
join([hello, goodbye], " ")
```

---

```
Hello Goodbye
```

---

## Interpolação de String

Concatenar strings pode ser complicado. Podemos ser muito mais expressivos com **interpolação de string**. Funciona assim: você especifica o que quer que seja incluído em sua string com o cífrão `$`. Aqui está o exemplo anterior, mas agora usando interpolação:

```
"$hello $goodbye"
```

---

```
Hello Goodbye
```

---

Isso funciona mesmo dentro de funções. Vamos revisitar nossa função `test` que foi definida em Section 3.1.5:

```
function test_interpolated(a, b)
    if a < b
```

```
"$a is less than $b"
```

```
elseif a > b
```

```
    "$a is greater than $b"
```

```
else
```

```
    "$a is equal to $b"
```

```
end
```

```
end
```

```
test_interpolated(3.14, 3.14)
```

---

**3.14** is equal to **3.14**

---

## Manipulações de Strings

Existem várias funções para manipular strings em Julia. Vamos demonstrar as mais comuns. Além disso, observe que a maioria dessas funções aceita uma Expressão Regular (RegEx)<sup>1</sup> como argumentos. Não cobriremos RegEx neste livro, mas te encorajamos a aprender sobre elas, especialmente se a maior parte de seu trabalho usa dados textuais.

<sup>1</sup> <https://docs.julialang.org/en/v1/manual/strings/#Regular-Expressions>

Primeiro, vamos definir uma string para brincarmos:

```
julia_string = "Julia is an amazing opensource programming language"
```

---

Julia is an amazing opensource programming language

---

1. `occursin`, `startswith` e `endswith`: São condicionais (retornam `true` ou `false`) se o primeiro argumento é um:

- **substring** do segundo argumento

```
occursin("Julia", julia_string)
```

---

true

---

- **prefixo** do segundo argumento

```
startswith("Julia", julia_string)
```

---

```
false
```

---

- **sufixo** do segundo argumento

```
endswith("Julia", julia_string)
```

---

```
false
```

---

## 2. lowercase, uppercase, titlecase e lowercasefirst:

```
lowercase(julia_string)
```

---

```
julia is an amazing opensource programming language
```

---

```
uppercase(julia_string)
```

---

```
JULIA IS AN AMAZING OPENSOURCE PROGRAMMING LANGUAGE
```

---

```
titlecase(julia_string)
```

---

```
Julia Is An Amazing Opensource Programming Language
```

---

```
lowercasefirst(julia_string)
```

---

```
julia is an amazing opensource programming language
```

---

## 3. replace: introduz uma nova sintaxe, chamada de **Pair**

```
replace(julia_string, "amazing" => "awesome")
```

---

```
Julia is an awesome opensource programming language
```

---

4. `split`: fatia uma string por um delimitador:

```
split(julia_string, " ")
```

---

```
SubString{String}["Julia", "is", "an", "amazing", "opensource", "  
"→programming", "language"]
```

---

## Convertendo em/parseando Strings

Muitas vezes, precisamos **converter** tipos de variáveis em Julia. Para converter um número em uma string, podemos usar a função `string`:

```
my_number = 123  
typeof(string(my_number))
```

---

String

---

Às vezes, queremos o oposto: converter uma string em um número (ou, como se diz no jargão, parsear essa string). Julia tem uma função útil para isso: `parse`.

```
typeof(parse(Int64, "123"))
```

---

Int64

---

Às vezes, queremos jogar pelo seguro com essas conversões. É aí que entra a função `tryparse`. Tem a mesma funcionalidade que `parse` mas retorna um valor do tipo solicitado ou `nothing`. Isso faz com que a `tryparse` seja útil quando buscamos evitar erros. Claro, você precisará lidar com todos aqueles valores `nothing` depois.

```
tryparse(Int64, "A very non-numeric string")
```

---

nothing

---

### 3.2.3 Tupla

Julia tem uma estrutura de dados chamada **tupla**. Ela é muito *especial* em Julia porque ela é frequentemente usada em relação às funções. Uma vez que as funções são um recurso importante em Julia, todo usuário precisa saber o básico das tuplas.

Uma tupla é um **contâiner de tamanho fixo que pode conter vários tipos diferentes**. Uma tupla é um **objeto imutável**, o que significa que não pode ser modificado após a instanciação. Para construir uma tupla, use parênteses () para delimitar o início e o fim, junto com vírgulas , como delimitadores entre valores:

```
my_tuple = (1, 3.14, "Julia")
```

---

```
(1, 3.14, "Julia")
```

---

Aqui, estamos criando uma tupla com três valores. Cada um dos valores é um tipo diferente. Podemos acessá-los por meio de indexação. Assim:

```
my_tuple[2]
```

---

```
3.14
```

---

Também podemos iterar sobre tuplas com a palavra-chave **for**. E até mesmo aplicar funções sobre tuplas. Mas nós nunca podemos **mudar qualquer valor de uma tupla** já que elas são **imutáveis**.

Você se lembra funções que retornam vários valores em Section [3.1.4](#)? Vamos inspecionar o que nossa função `add_multiply` retorna:

```
return_multiple = add_multiply(1, 2)
typeof(return_multiple)
```

---

```
Tuple{Int64, Int64}
```

---

Isso ocorre porque `return a, b` é o mesmo que `return (a, b)`:

```
1, 2
```

---

```
(1, 2)
```

---

Agora você pode ver porque tuplas e funções são frequentemente relacionadas.

Mais uma coisa para pensarmos sobre as tuplas. **Quando você deseja passar mais de uma variável para uma função anônima, adivinhe o que você precisa usar? Tuplas!**

```
map((x, y) -> x^y, 2, 3)
```

---

8

---

Ou ainda, mais do que dois argumentos:

```
map((x, y, z) -> x^y + z, 2, 3, 1)
```

---

9

---

### 3.2.4 Tupla Nomeada

Às vezes, você deseja nomear os valores contidos nas tuplas. É aí que entram as **tuplas nomeadas**. Sua funcionalidade é praticamente a mesma das tuplas: são **imutáveis** e podem conter **todo tipo de valor**.

A construção das tuplas nomeadas é ligeiramente diferente das tuplas. Você tem os familiares parênteses () e a vírgula , separadora de valor. Mas agora, você **nomeia os valores**:

```
my_namedtuple = (i=1, f=3.14, s="Julia")
```

---

(i = 1, f = 3.14, s = "Julia")

---

Podemos acessar os valores de uma tupla nomeada por meio da indexação como em tuplas regulares ou, alternativamente, **acessá-los por seus nomes** com o . :

```
my_namedtuple.s
```

---

Julia

---

Encerrando nossa discussão sobre tuplas nomeadas, há uma sintaxe *rápida* importante que você verá muito no código de Julia. Frequentemente, os usuários de Julia criam uma tupla nomeada usando o parêntese familiar () e vírgulas , mas sem nomear os valores. Para fazer isso, **comece a construção da tupla**

**nomeada especificando primeiro um ponto e vírgula ; antes dos valores.** Isto é especialmente útil quando os valores que iriam compor a tupla nomeada já estão definidos em variáveis ou quando você deseja evitar linhas longas:

```
i = 1
f = 3.14
s = "Julia"

my_quick_namedtuple = (; i, f, s)
```

---

```
(i = 1, f = 3.14, s = "Julia")
```

---

### 3.2.5 Ranges

Uma **range** em Julia representa um intervalo entre os limites de início e parada. A sintaxe é `start:stop`:

```
1:10
```

---

```
1:10
```

---

Como você pode ver, nosso range instanciado é do tipo `UnitRange{T}` onde `T` é o tipo de dados contido dentro de `UnitRange`:

```
typeof(1:10)
```

---

```
UnitRange{Int64}
```

---

E, se recolhermos todos os valores, temos:

```
[x for x in 1:10]
```

---

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

---

Também podemos construir ranges para outros tipos:

```
typeof(1.0:10.0)
```

---

```
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64
    ↵}, Int64}
```

---

Às vezes, queremos mudar o comportamento padrão do incremento do intervalo. Podemos fazer isso adicionando um incremento específico por meio da sintaxe da range `start:step:stop`. Por exemplo, suponha que queremos um range de `Float64` que vá de 0 a 1 com passos do tamanho de 0.2:

```
0.0:0.2:1.0
```

---

```
0.0:0.2:1.0
```

---

Se você quer “materializar” a range, transformando-a em uma coleção, você pode usar a função `collect`:

```
collect(1:10)
```

---

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

---

Assim, temos uma array do tipo especificado no range entre os limites que definimos. Já que estamos falando de arrays, vamos conversar sobre eles.

### 3.2.6 Array

Na sua forma mais básica, **arrays** contém múltiplos objetos. Por exemplo, elas podem armazenar múltiplos números em uma dimensão:

```
myarray = [1, 2, 3]
```

---

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

---

Na maioria das vezes você quer ter **arrays de tipo único para evitar problemas de performance**, mas observe que elas também podem conter objetos de diferentes tipos:

```
myarray = ["text", 1, :symbol]
```

---

```
Any["text", 1, :symbol]
```

---

Elas são o “pão com manteiga” da ciência de dados, porque as arrays são o que está por trás da maior parte do fluxo de trabalho em **manipulação de dados e visualização de dados**.

Portanto, **arrays são uma estrutura de dados essencial**.

## Tipos de array

Vamos começar com os **tipos de arrays**. Existem vários, mas vamos nos concentrar nos dois mais usados em ciência de dados:

- `Vector{T}`: array **unidimensional**. Escrita alternativa para `Array{T, 1}`.
- `Matrix{T}`: array **bidimensional**. Escrita alternativa para `Array{T, 2}`.

Observe aqui que `T` é o tipo da array subjacente. Então, por exemplo, `Vector{Int→64}` é um `Vector` no qual todos os elementos são `Int64`, e `Matrix{AbstractFloat}` é uma `Matrix` em que todos os elementos são subtipos de `AbstractFloat`.

Na maioria das vezes, especialmente ao lidar com dados tabulares, estamos usando arrays unidimensionais ou bidimensionais. Ambos são tipos `Array` para Julia. Mas, podemos usar os apelidos úteis `Vector` e `Matrix` para uma sintaxe clara e concisa.

## Construção de Array

Como **construímos** uma array? Nesta seção, começamos construindo arrays de uma forma mais baixo-nível. Isso pode ser necessário para escrever código de alto desempenho em algumas situações. No entanto, isso não é necessário na maioria das situações, e podemos, com segurança, usar métodos mais convenientes para criar arrays. Esses métodos mais convenientes serão descritos posteriormente nesta seção.

O construtor de baixo nível para arrays em Julia é o **construtor padrão**. Ele aceita o tipo de elemento como o parâmetro de tipo dentro dos colchetes `{}` e dentro do construtor você passará o tipo de elemento seguido pelas dimensões desejadas. É comum inicializar vetores e matrizes com elementos indefinidos usando o argumento para tipo `undef`. Um vetor de 10 elementos `undef Float64` pode ser construído como:

```
my_vector = Vector{Float64}(undef, 10)
```

---

```
[0.0, 6.9165829189171e-310, 6.9165829102698e-310, 0.0, 6.9165829189171e-310, 6.9
 ↪165829102698e-310, 0.0, 6.9165829189171e-310, 6.9165829102698e-310, 0.0]
```

---

Para matrizes, uma vez que estamos lidando com objetos bidimensionais, precisamos passar dois argumentos de dimensão dentro do construtor: um para **linhas** e outro para **colunas**. Por exemplo, uma matriz com 10 linhas e 2 colunas de elementos indefinidos `undef` pode ser instanciada como:

```
my_matrix = Matrix{Float64}(undef, 10, 2)
```

---

```
10x2 Matrix{Float64}:
 6.91659e-310 6.91659e-310
 6.91659e-310 6.91659e-310
```

---

Nós também temos algumas **apelidos sintáticos** para os elementos mais comuns na construção de arrays:

- `zeros` para todos os elementos inicializados em zero. Observe que o tipo padrão é `Float64` que pode ser alterado se necessário:

```
my_vector_zeros = zeros(10)
```

---

```
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

---

```
my_matrix_zeros = zeros(Int64, 10, 2)
```

---

```
10x2 Matrix{Int64}:
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
 0 0
```

---

- `ones` para todos os elementos inicializados em um:

```
my_vector_ones = ones(Int64, 10)
```

---

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

---

```
my_matrix_ones = ones(10, 2)
```

```
10×2 Matrix{Float64}:
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
1.0 1.0
```

---

Para outros elementos, podemos primeiro instanciar uma array com elementos `undef` e usar a função `fill!` para preencher todos os elementos de uma array com o elemento desejado. Segue um exemplo com [3.14](#) ( $\pi$ ):

```
my_matrix_pi = Matrix{Float64}(undef, 2, 2)
fill!(my_matrix_pi, 3.14)
```

---

```
2×2 Matrix{Float64}:
3.14 3.14
3.14 3.14
```

---

Também podemos criar arrays com **literais de array**. Por exemplo, segue uma matriz 2x2 de inteiros:

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

---

```
2×2 Matrix{Int64}:
1 2
3 4
```

---

Literais de array também aceitam uma especificação de tipo antes dos colchetes []. Então, se quisermos a mesma array 2x2 de antes mas agora como floats, podemos:

```
Float64[[1 2]
          [3 4]]
```

---

```
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

---

Também funciona para vetores:

```
Bool[0, 1, 0, 1]
```

---

```
Bool[0, 1, 0, 1]
```

---

Você pode até **misturar e combinar** literais de array com os construtores:

```
[ones(Int, 2, 2) zeros(Int, 2, 2)]
```

---

```
2×4 Matrix{Int64}:
 1  1  0  0
 1  1  0  0
```

---

```
[zeros(Int, 2, 2)
 ones(Int, 2, 2)]
```

---

```
4×2 Matrix{Int64}:
 0  0
 0  0
 1  1
 1  1
```

---

```
[ones(Int, 2, 2) [1; 2]
 [3 4]           5]
```

---

```
3×3 Matrix{Int64}:
 1  1  1
 1  1  2
 3  4  5
```

---

Outra maneira poderosa de criar uma array é escrever uma **compreensão de array**. Esta maneira de criar arrays é melhor na maioria dos casos: evita loops, indexação e outras operações sujeitas a erros. Você especifica o que deseja fazer dentro dos colchetes []. Por exemplo, digamos que queremos criar um vetor de quadrados de 1 a 10:

```
[x^2 for x in 1:10]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Eles também suportam múltiplas entradas:

```
[x*y for x in 1:10 for y in 1:2]
```

```
[1, 2, 2, 4, 3, 6, 4, 8, 5, 10, 6, 12, 7, 14, 8, 16, 9, 18, 10, 20]
```

E condicionais:

```
[x^2 for x in 1:10 if isodd(x)]
```

```
[1, 9, 25, 49, 81]
```

Tal como acontece com literais de array, você pode especificar o tipo desejado antes dos colchetes []:

```
Float64[x^2 for x in 1:10 if isodd(x)]
```

```
[1.0, 9.0, 25.0, 49.0, 81.0]
```

Finalmente, também podemos criar arrays com **funções de concatenação**. Concatenação é um termo padrão em programação e significa “acorrentar juntos.” Por exemplo, podemos concatenar strings com “aa” e “bb” para conseguir “aabb”:

```
"aa" * "bb"
```

aabb

E podemos concatenar arrays para criar novas arrays:

- `cat`: concatenar arrays de entrada ao longo de uma dimensão específica `dims`

```
cat(ones(2), zeros(2), dims=1)
```

[1.0, 1.0, 0.0, 0.0]

```
cat(ones(2), zeros(2), dims=2)
```

```
2×2 Matrix{Float64}:
1.0  0.0
1.0  0.0
```

- **vcat:** concatenação vertical, uma abreviatura para `cat(...; dims=1)`

```
vcat(ones(2), zeros(2))
```

[1.0, 1.0, 0.0, 0.0]

- **hcat:** concatenação horizontal, uma abreviatura para `cat(...; dims=2)`

```
hcat(ones(2), zeros(2))
```

```
2×2 Matrix{Float64}:
1.0  0.0
1.0  0.0
```

## Inspeção de Arrays

Assim que tivermos arrays, o próximo passo lógico seria **inspeciona-las**. Existem várias funções úteis que permitem ao usuário ter uma visão de qualquer array.

É muito útil saber que **tipo de elementos** existem dentro de uma array. Fazemos isso com `eltype`:

```
eltype(my_matrix_π)
```

---

Float64

---

Depois de conhecer seus tipos, alguém pode se interessar nas **dimensões da array**. Julia tem várias funções para inspecionar as dimensões da array:

- `length`: número total de elementos

```
length(my_matrix_π)
```

---

4

---

- `ndims`: número de dimensões

```
ndims(my_matrix_π)
```

---

2

---

- `size`: esse é um pouco complicado. Por padrão, ele retornará uma tupla contendo as dimensões da array.

```
size(my_matrix_π)
```

---

(2, 2)

---

Você pode obter uma dimensão específica com um segundo argumento para `size`. Aqui, o segundo eixo são as colunas

```
size(my_matrix_π, 2)
```

---

2

---

## Indexação e Fatiamento de Array

Às vezes, queremos inspecionar apenas certas partes de uma array. Chamamos isso de **indexação e fatiamento**. Se você quiser uma observação particular de um vetor, ou uma linha ou coluna de uma matriz, você provavelmente precisará **indexar uma array**.

Primeiro, vou criar um vetor e uma matriz de exemplo para brincar:

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

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

Vamos começar com vetores. Supondo que você queira o segundo elemento de um vetor. Você usa colchetes [] com o **índice** desejado dentro:

```
my_example_vector[2]
```

2

A mesma sintaxe segue com as matrizes. Mas, como as matrizes são arrays bidimensionais, temos que especificar *ambas* linhas e colunas. Vamos recuperar o elemento da segunda linha (primeira dimensão) e primeira coluna (segunda dimensão):

```
my_example_matrix[2, 1]
```

4

Júlia também possui palavras-chave convencionais para o **primeiro** e **último** elementos de uma array: **begin** e **end**. Por exemplo, o penúltimo elemento de um vetor pode ser recuperado como:

```
my_example_vector[end-1]
```

4

Isso também funciona para matrizes. Vamos recuperar o elemento da última linha e segunda coluna:

```
my_example_matrix[end, begin+1]
```

---

8

---

Muitas vezes, não estamos só interessados em apenas um elemento da array, mas em todo um **subconjunto de elementos da array**. Podemos fazer isso **fatiando** uma array. Usamos a mesma sintaxe de índice, mas adicionando dois pontos : para denotar os limites a partir dos quais estamos fatiando a array. Por exemplo, suponha que queremos obter do 2º ao 4º elemento de um vetor:

```
my_example_vector[2:4]
```

---

[2, 3, 4]

---

Poderíamos fazer o mesmo com matrizes. Particularmente com matrizes, se quisermos selecionar **todos os elementos** em uma dimensão seguinte, podemos fazer isso com apenas dois pontos :. Por exemplo, para obter todos os elementos da segunda linha:

```
my_example_matrix[2, :]
```

---

[4, 5, 6]

---

Você pode interpretar isso com algo como “pegue a 2ª linha e todas as colunas.”

Também suporta **begin** e **end**:

```
my_example_matrix[begin+1:end, end]
```

---

[6, 9]

---

## Manipulações de Array

Existem várias formas para **manipular** uma array. O primeiro seria manipular um **único elemento da array**. Nós apenas indexamos a array pelo elemento desejado e procedemos com uma atribuição =:

```
my_example_matrix[2, 2] = 42
my_example_matrix
```

---

3x3 Matrix{Int64}:

1	2	3
4	42	6
7	8	9

---

Ou, você pode manipular um determinado **subconjunto de elementos da array**. Nesse caso, precisamos fatiar a array e, em seguida, atribuir com `=`:

```
my_example_matrix[3, :] = [17, 16, 15]
my_example_matrix
```

**3x3 Matrix{Int64}:**

1	2	3
4	42	6
17	16	15

Observe que tivemos que atribuir um vetor porque nossa array fatiada é do tipo **Vector**:

```
typeof(my_example_matrix[3, :])
```

**Vector{Int64}** (alias for **Array{Int64, 1}**)

A segunda maneira de manipular uma array é **alterando suas dimensões**. Suponha que você tenha um vetor de 6 elementos e deseja torná-lo uma matriz 3x2. Você pode fazer isso com `reshape`, usando a array como o primeiro argumento e uma tupla de dimensões como segundo argumento:

```
six_vector = [1, 2, 3, 4, 5, 6]
tree_two_matrix = reshape(six_vector, (3, 2))
tree_two_matrix
```

**3x2 Matrix{Int64}:**

1	4
2	5
3	6

Você pode convertê-la de volta em um vetor especificando uma tupla com apenas uma dimensão como o segundo argumento:

```
reshape(tree_two_matrix, (6, ))
```

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

A terceira forma pela qual podemos manipular uma array é **aplicando uma função sobre cada elemento da array**. Aqui é onde o operador “dot” `.`, também conhecido como *broadcasting*, entra.

```
logarithm.(my_example_matrix)
```

---

```
3x3 Matrix{Float64}:
0.0      0.693147  1.09861
1.38629  3.73767   1.79176
2.83321  2.77259   2.70805
```

---

O operador dot em Julia é extremamente versátil. Você pode até mesmo usá-lo para vetorizar operadores infixos:

```
my_example_matrix .+ 100
```

---

```
3x3 Matrix{Int64}:
101  102  103
104  142  106
117  116  115
```

---

Uma alternativa para fazer o broadcasting de função sobre um vetor é usar `map`:

```
map(logarithm, my_example_matrix)
```

---

```
3x3 Matrix{Float64}:
0.0      0.693147  1.09861
1.38629  3.73767   1.79176
2.83321  2.77259   2.70805
```

---

Para funções anônimas, `map` geralmente é mais legível. Por exemplo,

```
map(x -> 3x, my_example_matrix)
```

---

```
3x3 Matrix{Int64}:
3      6      9
12    126    18
51    48     45
```

---

é bastante claro. No entanto, a mesma operação utilizando o operador de broadcast fica da seguinte forma:

```
(x -> 3x).(my_example_matrix)
```

---

```
3x3 Matrix{Int64}:
3      6      9
12    126    18
51    48     45
```

---

Além disso, `map` funciona com fatiamento:

```
map(x -> x + 100, my_example_matrix[:, 3])
```

---

```
[103, 106, 115]
```

---

Finalmente, às vezes, e especialmente ao lidar com dados tabulares, queremos aplicar uma função sobre todos os elementos em uma dimensão específica de uma array. Isso pode ser feito com a função `mapslices`. Parecido com `map`, o primeiro argumento é a função e o segundo argumento é a array. A única mudança é que precisamos especificar o argumento `dims` para sinalizar em qual dimensão queremos transformar os elementos.

Por exemplo, vamos usar `mapslice` com a função `sum` em ambas as linhas (`dims=1`) e colunas (`dims=2`):

```
# rows
mapslices(sum, my_example_matrix; dims=1)
```

---

```
1x3 Matrix{Int64}:
 22  60  24
```

---

```
# columns
mapslices(sum, my_example_matrix; dims=2)
```

---

```
3x1 Matrix{Int64}:
 6
52
48
```

---

## Iteração de array

Uma operação comum é iterar sobre uma array com um laço `for`. O laço `for` regular, quando aplicado sobre uma array retorna cada elemento.

O exemplo mais simples é com um vetor.

```
simple_vector = [1, 2, 3]

empty_vector = Int64[]

for i in simple_vector
    push!(empty_vector, i + 1)
end
```

```
empty_vector
```

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

Às vezes, você não quer iterar sobre cada elemento, mas sim sobre cada índice da array. Podemos usar a função `eachindex` combinada com um loop `for` para iterar sobre cada índice de array.

Novamente, vamos mostrar um exemplo com um vetor:

```
forty_twos = [42, 42, 42]

empty_vector = Int64[]

for i in eachindex(forty_twos)
    push!(empty_vector, i)
end

empty_vector
```

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

Nesse exemplo, o `eachindex(forty_twos)` retorna os índices de `forty_twos`, nomeadamente `[1, 2, 3]`.

Da mesma forma, podemos iterar sobre matrizes. O laço `for` padrão itera primeiro sobre as colunas e depois sobre as linhas. Ele irá primeiro percorrer todos os elementos na coluna 1, da primeira à última linha, em seguida, ele se moverá para a coluna 2 de maneira semelhante até cobrir todas as colunas.

Para aqueles familiarizados com outras linguagens de programação: Julia, como a maioria das linguagens de programação científica, é “colunar.” Colunar significa que os elementos da coluna são armazenados lado a lado na memória<sup>2</sup>. Isso também significa que iterar sobre os elementos em uma coluna é muito mais rápido do que sobre os elementos em uma linha.

Ok, vamos mostrar isso em um exemplo:

```
column_major = [[1 3]
                [2 4]]

row_major = [[1 2]
              [3 4]]
```

<sup>2</sup> ou, que os ponteiros de endereço de memória para os elementos na coluna são armazenados um ao lado do outro.

Se fizermos um loop sobre o vetor armazenado de forma ordenada para as colunas, então o resultado também é ordenado:

```
indexes = Int64[]

for i in column_major
    push!(indexes, i)
end

indexes
```

---

[1, 2, 3, 4]

---

No entanto, o resultado não fica ordenado ao interarmos sobre a outra matriz:

```
indexes = Int64[]

for i in row_major
    push!(indexes, i)
end

indexes
```

---

[1, 3, 2, 4]

---

Muitas vezes é melhor usar funções especializadas para esses loops:

- `eachcol`: itera sobre uma array coluna a coluna

```
first(eachcol(column_major))
```

---

[1, 2]

---

- `eachrow`: itera sobre uma array linha a linha

```
first(eachrow(column_major))
```

---

[1, 3]

---

### 3.2.7 Par

Em comparação com a enorme seção sobre arrays, esta seção sobre pares será breve. **Par** é uma estrutura de dados que contém dois objetos (que em geral estão relacionados um ao outro). Construímos um par em Julia usando a seguinte sintaxe:

```
my_pair = "Julia" => 42
```

```
"Julia" => 42
```

Os elementos são armazenados nos campos `first` e `second`.

```
my_pair.first
```

```
Julia
```

```
my_pair.second
```

```
42
```

Mas, na maioria dos casos, é mais fácil usar `first` e `last`<sup>3</sup>:

```
first(my_pair)
```

```
Julia
```

```
last(my_pair)
```

```
42
```

<sup>3</sup> é mais fácil porque `first` e `last` também funcionam em muitas outras coleções, então você não precisa se lembrar de tanta coisa.

Os pares serão muito usados na manipulação e visualização de dados, uma vez que ambos `DataFrames.jl` (Section 4) e `Makie.jl` (Section 5) aceitam objetos do tipo **Pair** em suas funções principais. Por exemplo, com `DataFrames.jl` veremos que `:a => :b` pode ser usado para renomear a coluna `:a` para `:b`.

### 3.2.8 Dict

Se você entendeu o que é um **Pair**, então **Dict** não será um problema. Para todos os propósitos práticos, **Dicts** são mapeamentos de chaves para valores. Por mapeamento, queremos dizer que se você der alguma chave a um **Dict**,

então o `Dict` poderá lhe dizer qual valor pertence àquela chave. Chaves (`key ↪ s`) e valores (`values`) podem ser de qualquer tipo, mas normalmente `keys` são strings.

Existem duas maneiras de construir `Dicts` em Julia. A primeira é passando um vetor de tuplas como `(key, value)` para o construtor `Dict`:

```
name2number_map = Dict([("one", 1), ("two", 2)])
```

```
Dict{String, Int64} with 2 entries:
"two" => 2
"one" => 1
```

Existe uma sintaxe mais legível com base no tipo `Pair` descrito acima. Você também pode passar `Pairs` de `key => values` para o construtor `Dict`:

```
name2number_map = Dict("one" => 1, "two" => 2)
```

```
Dict{String, Int64} with 2 entries:
"two" => 2
"one" => 1
```

Você pode recuperar um `value` de um `Dict` ao indexá-lo pela `key` correspondente:

```
name2number_map["one"]
```

1

Para adicionar uma nova entrada, você indexa o `Dict` pela `key` desejada e atribui um `value` com o operador de atribuição `=`:

```
name2number_map["three"] = 3
```

3

Se você quer checar se um `Dict` tem uma certa `key` você pode usar `keys` e `in`:

```
"two" in keys(name2number_map)
```

true

Para deletar uma `key` você pode usar a função `delete!`:

```
delete!(name2number_map, "three")
```

```
Dict{String, Int64} with 2 entries:
"two" => 2
"one" => 1
```

Ou, para excluir uma chave enquanto retorna seu valor, você pode usar `pop!`:

```
popped_value = pop!(name2number_map, "two")
```

```
2
```

Agora, nosso `name2number_map` tem apenas uma key:

```
name2number_map
```

```
Dict{String, Int64} with 1 entry:
"one" => 1
```

`Dicts` também são usados para manipulação de dados por `DataFrames.jl` (Section 4) e para visualização de dados por `Makie.jl` (Section 5). Logo, é importante conhecer suas funcionalidades básicas.

Existe outra maneira útil de construir `Dicts`. Suponha que você tenha dois vetores e deseja construir um `Dict` com um deles como se fosse `keys` e outro como se fosse `values`. Você pode fazer isso com uma função `zip` que “junta” dois objetos (como um zíper):

```
A = ["one", "two", "three"]
B = [1, 2, 3]

name2number_map = Dict(zip(A, B))
```

```
Dict{String, Int64} with 3 entries:
"two" => 2
"one" => 1
"three" => 3
```

Por exemplo, agora podemos obter o número 3 via:

```
name2number_map["three"]
```

```
3
```

### 3.2.9 Símbolo

`Symbol` na verdade *não* é uma estrutura de dados. É um tipo e se comporta de modo muito parecido com uma string. Em vez de colocar o texto entre aspas, um símbolo começa com dois pontos (:) e pode conter sublinhados:

```
sym = :some_text
```

---

```
:some_text
```

---

Podemos facilmente converter um símbolo em string e vice-versa:

```
s = string(sym)
```

---

```
some_text
```

---

```
sym = Symbol(s)
```

---

```
:some_text
```

---

Um benefício simples dos símbolos é que você digita um caractere a menos, ou seja, `:some_text` versus `"some text"`. Usamos muito `Symbol`s na manipulação de dados com o package `DataFrames.jl` (Section 4) e em visualização de dados com o package `Makie.jl` (Section 5).

### 3.2.10 Operador Splat

Em Julia, temos o operador “splat” ... que é usado em chamadas de função como uma **sequência de argumentos**. Ocasionalmente, usaremos o splat em algumas chamadas de função nos capítulos sobre **manipulação de dados** e **visualização de dados**.

A maneira mais intuitiva de aprender sobre o splat é com um exemplo. A função `add_elements` abaixo leva três argumentos para serem somados:

```
add_elements(a, b, c) = a + b + c
```

---

```
add_elements (generic function with 1 method)
```

---

Agora, suponha que temos uma coleção com três elementos. A maneira ingênuia de fazer isso seria fornecer à função todos os três elementos como argumentos de função:

```
my_collection = [1, 2, 3]

add_elements(my_collection[1], my_collection[2], my_collection[3])
```

6

Aqui é que usamos o operador “splat” ... que pega uma coleção (geralmente uma array, vetor, tupla ou range) e a converte em uma sequência de argumentos:

```
add_elements(my_collection...)
```

6

O ... deve ser incluído após a coleção que queremos espalhar ou “splat” em uma sequência de argumentos. Ambos os exemplos apresentados acima têm o mesmo resultado:

```
add_elements(my_collection...) == add_elements(my_collection[1], my_collection
    ↪[2], my_collection[3])
```

true

Sempre que Julia vê um operador splat dentro de uma chamada de função, ele será convertido em uma sequência de argumentos para todos os elementos da coleção separados por vírgulas.

Também funciona para ranges:

```
add_elements(1:3...)
```

6

### 3.3 Sistema de Arquivos

Em ciéncia de dados, a maioria dos projetos é realizada em um esforço colaborativo. Compartilhamos código, dados, tabelas, figuras e assim por diante. Por trás de tudo, está o **sistema de arquivos do sistema operacional (SO)**. Em um mundo perfeito, o mesmo programa daria a **mesma** saída quando executado em sistemas operacionais **diferentes**. Infelizmente, nem sempre é esse o caso.

Um exemplo disso é a diferença entre os caminhos do Windows, tal como `c:\user\john\`, e do Linux, como `/home/john`. Por isso é importante discutir as **melhores práticas em sistema de arquivos**.

Julia tem recursos de sistema de arquivos nativos que **lidam com as diferenças entre os sistemas operacionais**. Eles estão localizados no módulo `Filesystem`<sup>4</sup> <https://docs.julialang.org/en/v1/base/file/> da biblioteca central `Base` de Julia.

Sempre que você estiver lidando com arquivos como CSV, Excel ou qualquer outro script de Julia, certifique-se de que seu código **funciona em sistemas de arquivos de SOs diferentes**. Isso é facilmente realizado com as funções `joinpath`, `__FILE__` e `pkgdir`.

Se você escrever seu código em um pacote, você pode usar `pkgdir` para obter o diretório raiz do pacote. Por exemplo, para o pacote de Julia Data Science (JDS) que usamos para produzir este livro:

```
/home/runner/work/JuliaDataScience-PT/JuliaDataScience-PT
```

como você pode ver, o código para produzir este livro foi executado em um computador Linux. Se você está usando um script, você pode obter a localização do arquivo de script via

```
root = dirname(__FILE__)
```

O bom desses dois comandos é que eles são independentes de como o usuário iniciou o Julia. Em outras palavras, não importa se o usuário iniciou o programa com `julia scripts/script.jl` ou `julia script.jl`, em ambos os casos os caminhos são os mesmos.

A próxima etapa seria incluir o caminho relativo a partir de `root` para o nosso arquivo desejado. Uma vez que diferentes sistemas operacionais têm maneiras diferentes de construir caminhos relativos com subpastas (alguns usam barras / enquanto outros podem usar barras invertidas \), não podemos simplesmente concatenar o caminho relativo do arquivo com a string `root`. Para isso, temos a função `joinpath`, que unirá diferentes caminhos relativos e nomes de arquivos de acordo com a implementação específica do sistema de arquivos do seu sistema operacional.

Supondo que você tenha um script chamado `my_script.jl` dentro do diretório do seu projeto. Você pode ter uma representação robusta do caminho do arquivo para `my_script.jl` como:

```
joinpath(root, "my_script.jl")
```

---

```
/home/runner/work/JuliaDataScience-PT/JuliaDataScience-PT/my_script.jl
```

---

`joinpath` também lida com **subfolders**. Agora vamos imaginar uma situação comum em que você tem uma pasta chamada `data/` no diretório do seu projeto. Dentro desta pasta, há um arquivo CSV chamado `my_data.csv`. Você pode ter a mesma representação robusta do caminho do arquivo para `my_data.csv` como:

```
joinpath(root, "data", "my_data.csv")
```

```
/home/runner/work/JuliaDataScience-PT/JuliaDataScience-PT/data/my_data.csv
```

É um bom hábito de adquirir, porque é muito provável que evite problemas para você ou outras pessoas mais tarde.

### 3.4 Biblioteca Padrão de Julia

Julia tem uma **biblioteca padrão rica** que está disponível em *toda* instalação de Julia. Ao contrário de tudo o que vimos até agora, por exemplo, tipos, estruturas de dados e sistema de arquivos; você **deve carregar módulos de biblioteca padrão em seu ambiente** para usar um módulo ou função particular.

Isso é feito via `using` ou `import`. Neste livro, carregaremos o código via `using`:

```
using ModuleName
```

Depois de fazer isso, você pode acessar todas as funções e tipos dentro do módulo chamado `ModuleName`.

#### 3.4.1 Datas

Saber como lidar com datas e timestamps é importante na ciência de dados. Como dissemos na seção *Por que Julia?* (Section 2), o `pandas` do Python usa seu próprio tipo de `datetime` para lidar com datas. O mesmo é verdade no tidyverse de R, no pacote `lubridate`, que também define o seu próprio tipo de `datetime` para lidar com datas. Em Julia, os pacotes não precisam escrever sua própria lógica de datas, porque Julia tem um módulo de datas em sua biblioteca padrão chamado `Dates`.

Para começar, vamos carregar o módulo `Dates`:

```
using Dates
```

## Tipos Date e DateTime

O módulo de biblioteca padrão `Dates` tem **dois tipos para trabalhar com datas**:

1. `Date`: representando o tempo em dias e
2. `DateTime`: representando o tempo com precisão de milisegundos.

Nós podemos construir `Date` e `DateTime` com o construtor padrão especificando um número inteiro para representar ano, mês, dia, horas e assim por diante:

```
Date(1987) # year
```

---

1987-01-01

---

```
Date(1987, 9) # year, month
```

---

1987-09-01

---

```
Date(1987, 9, 13) # year, month, day
```

---

1987-09-13

---

```
DateTime(1987, 9, 13, 21) # year, month, day, hour
```

---

1987-09-13T21:00:00

---

```
DateTime(1987, 9, 13, 21, 21) # year, month, day, hour, minute
```

---

1987-09-13T21:21:00

---

Para os curiosos, 13 de setembro de 1987, 21:21 é a hora oficial do nascimento do primeiro autor, José.

Nós também podemos passar tipos “período” ou `Period` para o construtor padrão. **Tipos Period são o equivalente-humano para a representação do tempo** para o computador. `Dates` em Julia têm os seguintes subtipos abstratos de `Period`:

```
subtypes(Period)
```

---

DatePeriod

---

---

TimePeriod

---

que se dividem nos seguintes tipos concretos, e eles são bastante autoexplicativos:

`subtypes(DatePeriod)`

---

Day

---



---

Month

---



---

Quarter

---



---

Week

---



---

Year

---

`subtypes(TimePeriod)`

---

Hour

---



---

Microsecond

---



---

Millisecond

---



---

Minute

---



---

Nanosecond

---



---

Second

---

Assim, poderíamos, alternativamente, construir a hora oficial de nascimento de José como:

`DateTime(Year(1987), Month(9), Day(13), Hour(21), Minute(21))`

---

1987-09-13T21:21:00

---

## Parseando Datas

Na maioria das vezes, não construiremos instâncias `Date` ou `DateTime` do zero. Na verdade, nós provavelmente **parsearemos strings para transformá-las em tipos Date ou DateTime**.

Os construtores `Date` e `DateTime` podem ser alimentados com uma string e uma string de formato de data. Por exemplo, a string "`19870913`" representando 13 de setembro de 1987 pode ser parseada com:

```
Date("19870913", "yyyyymmdd")
```

---

`1987-09-13`

---

Observe que o segundo argumento é uma representação em string do formato. Temos os primeiros quatro dígitos que representam o ano `y`, seguido por dois dígitos para o mês `m` e finalmente dois dígitos para dia `d`.

Também funciona para timestamps com `DateTime`:

```
DateTime("1987-09-13T21:21:00", "yyyy-mm-ddTHH:MM:SS")
```

---

`1987-09-13T21:21:00`

---

Você pode encontrar mais informações sobre como especificar diferentes formatos de data na documentação `Dates'` de Julia<sup>5</sup>. Não se preocupe se você tiver que revisitá-lo o tempo todo, nós mesmos fazemos isso ao trabalhar com datas e timestamps.

De acordo com a documentação `Dates'` de Julia<sup>6</sup>, usar o método `Date(date_string ↗, format_string)` é satisfatório se ele for chamado apenas algumas vezes. Se houver muitas strings de data formatadas de forma semelhante para analisar, no entanto, é muito mais eficiente criar primeiro um tipo `DateFormat`, e, em seguida, o passar em vez de uma string de formato bruto. Então, nosso exemplo anterior se torna:

```
format = DateFormat("yyyyymmdd")
Date("19870913", format)
```

---

`1987-09-13`

---

Como alternativa, sem perda de desempenho, você pode usar o prefixo de string literal `dateformat"..."`:

<sup>5</sup> <https://docs.julialang.org/en/v1/stdlib/Dates/#Dates.DateFormat>

<sup>6</sup> <https://docs.julialang.org/en/v1/stdlib/Dates/#Constructors>

```
Date("19870913", dateformat"yyyymmdd")
```

---

1987-09-13

---

## Extraindo Informações de Data

É fácil **extrair as informações desejadas dos objetos Date e DateTime**. Primeiro, vamos criar uma instância de uma data muito especial:

```
my_birthday = Date("1987-09-13")
```

---

1987-09-13

---

Podemos extrair tudo o que quisermos de my\_birthday:

```
year(my_birthday)
```

---

1987

---

```
month(my_birthday)
```

---

9

---

```
day(my_birthday)
```

---

13

---

O módulo **Dates** de Julia também tem **funções compostas que retornam uma tupla de valores**:

```
yeарmonth(my_birthday)
```

---

(1987, 9)

---

```
monthday(my_birthday)
```

---

(9, 13)

---

```
yearmonthday(my_birthday)
```

---

(1987, 9, 13)

---

Também podemos ver o dia da semana e outras coisas úteis:

```
dayofweek(my_birthday)
```

---

7

---

```
dayname(my_birthday)
```

---

Sunday

---

```
dayofweekofmonth(my_birthday)
```

---

2

---

Sim, José nasceu no segundo domingo de setembro.

**OBSERVAÇÃO:** Aqui está uma dica útil para recuperar apenas os dias de semana de instâncias de **Dates**. Use o `filter` no `dayofweek(your_date) <= 5`. Para o dia útil, você pode verificar o pacote `BusinessDays.jl`<sup>7</sup>.

<sup>7</sup> <https://github.com/JuliaFinance/BusinessDays.jl>

## Operações de Data

Podemos realizar **operações** em instâncias de **Dates**. Por exemplo, podemos adicionar dias a uma instância **Date** ou **DateTime**. Note que as **Dates** em Julia executará automaticamente os ajustes necessários para anos bissextos, e por meses com 30 ou 31 dias (isso é conhecido como aritmética *calendárica*).

```
my_birthday + Day(90)
```

---

1987-12-12

---

Podemos adicionar quantas quisermos:

```
my_birthday + Day(90) + Month(2) + Year(1)
```

---

1989-02-11

---

Caso você esteja se perguntando: “O que posso fazer com datas mesmo? O que está disponível?” então você pode usar `methodswith` para verificar. Mostramos apenas os primeiros 20 resultados aqui:

```
first(methodswith(Date), 20)
```

---

```
[1] show(io::IO, dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/io.jl:736
[2] show(io::IO, ::MIME{Symbol("text/plain")}, dt::Date) in Dates at /opt/
    ↪hostedtoolcache/julia/1.7.3/x64/share/julia/stdlib/v1.7/Dates/src/io.jl:7
    ↪34
[3] DateTime(dt::Date, t::Time) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64
    ↪/share/julia/stdlib/v1.7/Dates/src/types.jl:403
[4] Day(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share/julia/
    ↪stdlib/v1.7/Dates/src/periods.jl:36
[5] Month(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share/julia
    ↪/stdlib/v1.7/Dates/src/periods.jl:36
[6] Quarter(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share/
    ↪julia/stdlib/v1.7/Dates/src/periods.jl:36
[7] Week(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share/julia/
    ↪stdlib/v1.7/Dates/src/periods.jl:36
[8] Year(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share/julia/
    ↪stdlib/v1.7/Dates/src/periods.jl:36
[9] firstdayofmonth(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:84
[10] firstdayofquarter(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x6
    ↪4/share/julia/stdlib/v1.7/Dates/src/adjusters.jl:157
[11] firstdayofweek(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:52
[12] firstdayofyear(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:119
[13] lastdayofmonth(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:100
[14] lastdayofquarter(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:180
[15] lastdayofweek(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:68
[16] lastdayofyear(dt::Date) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/adjusters.jl:135
[17] +(dt::Date, t::Time) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share
    ↪/julia/stdlib/v1.7/Dates/src/arithmetic.jl:19
[18] +(dt::Date, y::Year) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/share
    ↪/julia/stdlib/v1.7/Dates/src/arithmetic.jl:27
[19] +(dt::Date, z::Month) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/arithmetic.jl:54
```

```
[20] +(x::Date, y::Quarter) in Dates at /opt/hostedtoolcache/julia/1.7.3/x64/
    ↪share/julia/stdlib/v1.7/Dates/src/arithmetic.jl:73
```

A partir disso, podemos concluir que também podemos usar o operador de sinal de mais + e menos -. Vamos ver quantos anos o José tem, em dias:

```
today() - my_birthday
```

12873 days

A **duração padrão** de tipos de `Date` é uma instância de `Day`. Para o `DateTime`, a duração padrão é uma instância de `Millisecond`:

```
DateTime(today()) - DateTime(my_birthday)
```

1112227200000 milliseconds

## Intervalos de Data

Uma coisa boa sobre o módulo `Dates` é que também podemos construir facilmente **intervalos de data e hora**. Julia é inteligente o suficiente para não ter que definir todos os tipos de intervalo e operações que abordamos em Section 3.2.5. Ela apenas estende as funções e operações definidas para `range` para os tipos `Date`. Isso é conhecido como despacho múltiplo e já abordamos isso em *Por que Julia?* (Section 2).

Por exemplo, suponha que você deseja criar um intervalo de tipo `Day`. Isso é fácil de fazer com o operador dois-pontos ::

```
Date("2021-01-01"):Day(1):Date("2021-01-07")
```

2021-01-01

2021-01-02

2021-01-03

2021-01-04

2021-01-05

---

2021-01-06

---



---

2021-01-07

---

Não há nada de especial em usar `Day(1)` como o intervalo, podemos **usar qualquer tipo** `Period` como intervalo. Por exemplo, usando 3 dias como intervalo:

```
Date("2021-01-01"):Day(3):Date("2021-01-07")
```

---



---

2021-01-01

---



---

2021-01-04

---



---

2021-01-07

---

Ou mesmo meses:

```
Date("2021-01-01"):Month(1):Date("2021-03-01")
```

---



---

2021-01-01

---



---

2021-02-01

---



---

2021-03-01

---

Perceba que o tipo deste intervalo é um `StepRange` com o `Date` e um tipo concreto `Period` que usamos como intervalo dentro do operador ::

```
date_interval = Date("2021-01-01"):Month(1):Date("2021-03-01")
typeof(date_interval)
```

---



---

StepRange{Date, Month}

---

Podemos converter isso para um **vetor** com a função `collect`:

```
collected_date_interval = collect(date_interval)
```

---



---

2021-01-01

---



---

2021-02-01

---

---

2021-03-01

---

E teremos todas as **funcionalidades de array disponíveis**, como, por exemplo, indexação:

```
collected_date_interval[end]
```

---

2021-03-01

---

Também podemos fazer **broadcast de operações de data** em um vetor de **Dates**:

```
collected_date_interval .+ Day(10)
```

---

2021-01-11

---



---

2021-02-11

---



---

2021-03-11

---

Da mesma forma, esses exemplos funcionam para tipos `DateTime` também.

### 3.4.2 Números Aleatórios

Outro módulo importante na biblioteca padrão de Julia é o módulo `Random`. Este módulo lida com **geração de números aleatórios**. `Random` é uma biblioteca rica e, se você está interessado, deve consultar a documentação `Random` de Julia<sup>8</sup>. Vamos cobrir *somente* três funções: `rand`, `randn` e `seed!`.

Para começar, primeiro carregamos o módulo `Random`. Como sabemos exatamente o que queremos carregar, podemos também carregar explicitamente os métodos que queremos usar:

```
using Random: rand, randn, seed!
```

<sup>8</sup> <https://docs.julialang.org/en/v1/stdlib/Random/>

Nós temos **duas funções principais que geram números aleatórios**:

- `rand`: faz a amostragem de um **elemento aleatório** de uma estrutura ou tipo de dados.
- `randn`: gera um número aleatório que segue uma **distribuição normal padrão** (média 0 e desvio padrão 1) de um tipo específico.

**OBSERVAÇÃO:** Observe que essas duas funções já estão no módulo `Base` de Julia. Então, você não precisa importar `Random` se estiver planejando usá-los.

## rand

Por padrão, se você chamar `rand` sem argumentos, ele retornará um `Float64` no intervalo  $[0, 1)$ , o que significa no intervalo compreendido entre os limites 0 inclusivo e 1 exclusivo:

```
rand()
```

---

0.8830643295421374

---

Você pode modificar argumentos `rand` de várias maneiras. Por exemplo, suponha que você queira mais de 1 número aleatório:

```
rand(3)
```

---

[ 0.7253840756214206, 0.17373996677252268, 0.05524107416544999 ]

---

Ou você quer um intervalo diferente:

```
rand(1.0:10.0)
```

---

9.0

---

Você também pode especificar um tamanho de incremento diferente dentro do intervalo e um tipo diferente. Aqui estamos usando números sem ponto . então Julia irá interpretá-los como `Int64`:

```
rand(2:2:20)
```

---

8

---

Você também pode misturar e combinar argumentos:

```
rand(2:2:20, 3)
```

---

[ 16, 20, 16 ]

---

`rand` também aceita uma coleção de elementos como, por exemplo, uma tupla:

```
rand((42, "Julia", 3.14))
```

---

42

---

E também arrays:

```
rand([1, 2, 3])
```

---

3

---

**DictS:**

```
rand(Dict(:one => 1, :two => 2))
```

---

:one => 1

---

Para terminar todas as opções de argumentos `rand`, você pode especificar as dimensões de número aleatório desejadas em uma tupla. Se você fizer isso, o tipo retornado será uma array. Por exemplo, aqui uma matriz 2x2 de números `Float64` entre 1.0 e 3.0:

```
rand(1.0:3.0, (2, 2))
```

---

2×2 Matrix{Float64}:

2.0 3.0  
2.0 1.0

---

## randn

`randn` segue o mesmo princípio geral de `rand`, mas agora ele só retorna números gerados a partir da **distribuição normal padrão**. A distribuição normal padrão é a distribuição normal com média 0 e desvio padrão 1. O tipo padrão é `Float64` e só permite subtipos de `AbstractFloat` ou `Complex`:

```
randn()
```

---

0.32732815366588475

---

Só podemos especificar o tamanho:

```
randn((2, 2))
```

---

2×2 Matrix{Float64}:

1.04702	0.517787
-0.572624	-0.760324

---

**seed!**

Para terminar a visão geral de `Random`, vamos falar sobre **reprodutibilidade**. Muitas vezes, queremos fazer algo **replicável**. Ou seja, queremos que o gerador de números aleatórios gere a **mesma sequência aleatória de números**. Podemos fazer isso com a função `seed!`:

```
seed!(123)
rand(3)
```

---

```
[0.521213795535383, 0.5868067574533484, 0.8908786980927811]
```

---

```
seed!(123)
rand(3)
```

---

```
[0.521213795535383, 0.5868067574533484, 0.8908786980927811]
```

---

A fim de evitar a repetição tediosa e ineficiente de `seed!` em todo lugar, podemos definir uma instância de `seed!` e passá-la como o primeiro argumento de `rand` **ou** `randn`.

```
my_seed = seed!(123)
```

---

```
Random.TaskLocalRNG()
```

---

```
rand(my_seed, 3)
```

---

```
[0.19090669902576285, 0.5256623915420473, 0.3905882754313441]
```

---

```
rand(my_seed, 3)
```

---

```
[0.19090669902576285, 0.5256623915420473, 0.3905882754313441]
```

---

**OBSERVAÇÃO:** Se você quiser que seu código seja reproduzível, basta chamar `seed!` no começo do seu script. Isso cuidará da reprodutibilidade sequencial das operações `Random`. Não há necessidade de usá-la dentro de todo `rand` e `randn`.

### 3.4.3 Downloads

Uma última coisa da biblioteca padrão de Julia para cobrirmos é o **módulo Download**. Será muito breve porque iremos cobrir apenas uma única função chamada `download`.

Suponha que você queira **fazer o download de um arquivo da internet para o seu armazenamento local**. Você pode fazer isso com a função `download`. O primeiro e único argumento obrigatório é o url do arquivo. Você também pode especificar como um segundo argumento o caminho de saída desejado para o arquivo baixado (não se esqueça das práticas recomendadas do sistema de arquivos!). Se você não especificar um segundo argumento, Julia irá, por padrão, criar um arquivo temporário com a função `tempfile`.

Vamos carregar o método `download`:

```
using Download: download
```

Por exemplo, vamos baixar nosso arquivo `Project.toml` do repositório GitHub `JuliaDataScience`<sup>9</sup>. Observe que a função `download` não é exportada pelo módulo `Downloads`, então temos que usar a sintaxe `Module.function`. Por padrão, ele retorna uma string que contém o caminho do arquivo para o arquivo baixado:

```
url = "https://raw.githubusercontent.com/JuliaDataScience/JuliaDataScience/main/
      ↪Project.toml"

my_file = Downloads.download(url) # tempfile() being created
```

---

/tmp/jL\_gSRXHJ

---

Com `readlines`, nós podemos observar as primeiras 4 linhas do arquivo que baixamos:

```
readlines(my_file)[1:4]
```

---

```
4-element Vector{String}:
"name = \"JDS\""
"uuid = \"6c596d62-2771-44f8-8373-3ec4b616ee9d\""
"authors = [\"Jose Storopoli\", \"Rik Huijzer\", \"Lazaro Alonso\"]"
""
```

---

**OBSERVAÇÃO:** Para interações HTTP mais complexas, como interação com APIs da web, consulte o pacote `HTTP.jl`<sup>10</sup>.

<sup>9</sup> <https://github.com/JuliaDataScience/JuliaDataScience>

<sup>10</sup> <https://github.com/JuliaWeb/HTTP.jl>

## 4 *DataFrames.jl*

Em geral, os dados vêm em formato tabular. Por tabular, queremos dizer que os dados consistem em uma tabela que contém linhas e colunas. As colunas normalmente contêm o mesmo tipo de dados, enquanto as linhas são de tipos diferentes. As linhas, na prática, denotam observações enquanto as colunas indicam variáveis. Por exemplo, podemos ter uma tabela de programas de TV contendo o país em que cada um foi produzido e nossa classificação pessoal, acesse [Table 4.1](#).

name	country	rating
Game of Thrones	United States	8.2
The Crown	England	7.3
Friends	United States	7.8
...	...	...

Table 4.1: TV shows.

Aqui, as reticências significam que esta pode ser uma tabela muito longa e mostramos apenas algumas linhas. Ao analisar dados, muitas vezes levantamos questões interessantes sobre eles, também chamadas de *queries* (ou consultas) de dados. Para tabelas grandes, os computadores são capazes de responder a perguntas desse tipo muito mais rápido do que você faria manualmente. Alguns exemplos de questões para os dados seriam:

- Qual programa de TV recebeu a nota mais alta?
- Quais programas de TV foram produzidos nos Estados Unidos?
- Quais programas de TV foram produzidos no mesmo país?

Mas, como pesquisador, você verá que a ciência real muitas vezes começa com várias tabelas ou fontes de dados. Por exemplo, se também tivéssemos dados das classificações de outra pessoa para os programas de TV ([Table 4.2](#)):

name	rating
Game of Thrones	7
Friends	6.4
...	...

Table 4.2: Ratings.

Agora, poderíamos fazer as seguintes perguntas:

- Qual é a avaliação média de Game of Thrones?
- Quem deu a classificação mais alta para Friends?
- Quais programas de TV foram avaliados por você, mas não pela outra pessoa?

Ao longo deste capítulo, mostraremos como você pode responder facilmente a essas perguntas em Julia. Para fazer isso, primeiro mostramos porque precisamos de um pacote Julia chamado `DataFrames.jl`. Nas próximas seções, mostraremos como você pode usar este pacote e também como escrever transformações de dados (Section 4.9).

Vejamos uma tabela de notas escolares como a Table 4.3:

name	age	grade_2020
Bob	17	5.0
Sally	18	1.0
Alice	20	8.5
Hank	19	4.0

Table 4.3: Grades for 2020.

Aqui, o nome da coluna tem um tipo `string`, idade tem um tipo `integer` e nota um tipo `float`.

Até agora, este livro tratou apenas do básico de Julia. Esse básico é bom para muitas coisas, mas não para tabelas. Para mostrar que precisamos de mais, vamos tentar armazenar os dados tabulares em arrays:

```
function grades_array()
    name = ["Bob", "Sally", "Alice", "Hank"]
    age = [17, 18, 20, 19]
    grade_2020 = [5.0, 1.0, 8.5, 4.0]
    (; name, age, grade_2020)
end
```

Agora, os dados são armazenados na chamada forma colunar, o que é complicado quando queremos obter dados de uma linha:

```
function second_row()
    name, age, grade_2020 = grades_array()
    i = 2
    row = (name[i], age[i], grade_2020[i])
end
second_row()
```

---

```
("Sally", 18, 1.0)
```

---

Ou ainda, se você quiser ver a nota de Alice, primeiro você precisa descobrir em que linha Alice está:

```
function row_alice()
    names = grades_array().name
    i = findfirst(names .== "Alice")
end
row_alice()
```

---

```
3
```

---

e então podemos obter o valor:

```
function value_alice()
    grades = grades_array().grade_2020
    i = row_alice()
    grades[i]
end
value_alice()
```

---

```
8.5
```

---

DataFrames.jl é capaz de resolver esse tipo de problemas com facilidade. Você pode começar carregando `DataFrames.jl` com `using`:

```
using DataFrames
```

Com `DataFrames.jl`, podemos definir uma `DataFrame` para armazenar nossos dados tabulares:

```
names = ["Sally", "Bob", "Alice", "Hank"]
grades = [1, 5, 8.5, 4]
df = DataFrame(; name=names, grade_2020=grades)
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

o que nos dá uma variável `df` que contém nossos dados em formato de tabela.

**OBSERVAÇÃO:** Isso funciona, mas há uma coisa que precisamos mudar imediatamente. Neste exemplo, definimos as variáveis `name`, `grade_2020` e `df` em escopo global. Isso significa que essas variáveis podem ser acessadas e editadas de qualquer lugar. Se continuássemos escrevendo o livro assim, teríamos algumas centenas de variáveis no final do livro, embora os dados que colocamos na variável `name` só deveriam ser acessados via `DataFrame!` As variáveis `name` e `grade_2020` não foram feitas para serem mantidas por muito tempo! Agora, imagine se mudássemos o conteúdo de `grade_2020` algumas vezes nesse livro. Dado apenas o livro como PDF, seria quase impossível descobrir o conteúdo da variável ao final.

Podemos resolver isso facilmente usando funções.

Vamos fazer a mesma coisa de antes, mas agora em uma função:

```
function grades_2020()
    name = ["Sally", "Bob", "Alice", "Hank"]
    grade_2020 = [1, 5, 8.5, 4]
    DataFrame(; name, grade_2020)
end
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Table 4.5: Grades 2020.

Note que `name` e `grade_2020` são destruídas depois que a função retorna, ou seja, só estão disponíveis na função. Existem dois outros benefícios ao fazer isso. Primeiro, agora está claro para o leitor onde `name` e `grade_2020` pertencem: eles pertencem às notas de 2020. Em segundo lugar, é fácil determinar qual será a saída de `grades_2020()` em qualquer ponto do livro. Por exemplo, agora podemos atribuir os dados a uma variável `df`:

```
df = grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Podemos também mudar os conteúdos de `df`:

```
df = DataFrame(name = ["Malice"], grade_2020 = [10])
```

name	grade_2020
Malice	10

E ainda assim recuperar os dados originais de volta sem nenhum problema:

```
df = grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Claro, pressupondo que a função não seja redefinida. Prometemos não fazer isso neste livro, porque é uma má ideia exatamente por este motivo. Ao invés de “mudarmos” a função, vamos fazer uma nova e lhe daremos um nome claro.

Portanto, retornemos ao construtor `DataFrames`. Como você deve ter visto, a maneira de criar um é simplesmente passar vetores como argumentos para o construtor `DataFrame`. Você pode aparecer com qualquer vetor de Julia válido e ele funcionará **contanto que os vetores tenham o mesmo comprimento**. Valores duplicados, símbolos Unicode e qualquer tipo de número são aceitos:

```
DataFrame(σ = ["a", "a", "a"], δ = [π, π/2, π/3])
```

σ	δ
a	3.141592653589793
a	1.5707963267948966
a	1.0471975511965976

Normalmente, em seu código, você criaria uma função que envelopa uma ou mais funções `DataFrames`. Por exemplo, podemos fazer uma função para obter as notas de um ou mais `names`:

```
function grades_2020(names :: Vector{Int})
    df = grades_2020()
    df[names, :]
end
```

```
grades_2020([3, 4])
```

name	grade_2020
Alice	8.5
Hank	4.0

Esta forma de usar funções para envelopar funcionalidades básicas em linguagens de programação e pacotes é bastante comum. Basicamente, você pode pensar em Julia e `DataFrames.jl` como peças de Lego. Eles fornecem peças muito **genéricas** que permitem que você crie coisas para seu caso de uso **específico** como neste exemplo de notas. Usando essas peças, você pode fazer um script de análise de dados, controlar um robô ou o que você quiser construir.

Até agora, os exemplos eram bastante complicados, porque tínhamos que usar índices. Nas próximas seções, mostraremos como carregar e salvar dados, e muitas outras peças de Lego poderosas fornecidas por `DataFrames.jl`.

## 4.1 Carregar e salvar arquivos

Ter os dados apenas dentro dos programas Julia e não ser capaz de carregá-los ou salvá-los seria muito limitante. Portanto, começamos mencionando como armazenar e carregar arquivos do HD. Focamos em formatos de arquivo CSV, veja Section 4.1.1, e Excel, veja Section 4.1.2, uma vez que esses são os formatos de armazenamento de dados mais comuns para dados tabulares.

### 4.1.1 CSV

Arquivos Comma-separated values (CSV, valores separados por vírgula) são eficazes no armazenamento de tabelas. Os arquivos CSV têm duas vantagens sobre outros arquivos de armazenamento de dados. Primeiro, eles fazem exatamente o que o nome indica que fazem, ou seja, armazenam valores, separando-os por vírgulas , (ou, em sistemas operacionais ambientados no Brasil, separados por ponto-e-vírgula, pois a vírgula é utilizada como separador decimal). Este acrônimo também é usado como extensão de arquivo. Portanto, certifique-se de salvar seus arquivos usando a extensão “.csv” tal como “my-file.csv.” Para demonstrar a aparência de um arquivo CSV, podemos instalar o pacote `CSV.jl`<sup>1</sup>:

```
julia> ]
```

```
pkg> add CSV
```

<sup>1</sup> <http://csv.juliadata.org/latest/>

e carregá-lo via:

```
using CSV
```

Agora podemos usar nossos dados anteriores:

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

e escrevê-los em um arquivo, para em seguida lê-los novamente:

```
function write_grades_csv()
    path = "grades.csv"
    CSV.write(path, grades_2020())
end
```

```
path = write_grades_csv()
read(path, String)
```

---

```
name,grade_2020
Sally,1.0
Bob,5.0
Alice,8.5
Hank,4.0
```

---

Aqui, também vemos o segundo benefício do formato de dados CSV: os dados podem ser lidos usando um editor de texto simples. Isso difere de muitos formatos de dados alternativos que requerem software proprietário, por exemplo, Excel.

Isso funciona muito bem, mas e se nossos dados **contiverem vírgulas**, como valores? Se tivéssemos de escrever ingenuamente os dados com vírgulas, isso tornaria os arquivos muito difíceis de se converter de volta para uma tabela. Por sorte, CSV.jl resolve esse problema para nós de forma automática. Considere os seguintes dados com vírgulas ,:

```
function grades_with_commas()
    df = grades_2020()
    df[3, :name] = "Alice,"
    df
```

```
end
grades_with_commas()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice,	8.5
Hank	4.0

Table 4.12: Grades with commas.

Se escrevermos isso, teremos:

```
function write_comma_csv()
    path = "grades-commas.csv"
    CSV.write(path, grades_with_commas())
end
path = write_comma_csv()
read(path, String)
```

---

```
name,grade_2020
Sally,1.0
Bob,5.0
"Alice,",8.5
Hank,4.0
```

---

Logo, `CSV.jl` adiciona aspas " em torno dos valores contendo vírgulas. Outra maneira comum de resolver esse problema é gravar os dados em um formato de arquivo **tab-separated values** (TSV, valores separados por tabulações). Isso pressupõe que os dados não contêm tabulações, o que é válido na maioria dos casos.

Além disso, observe que os arquivos TSV também podem ser lidos usando um editor de texto simples e esses arquivos usam a extensão ".tsv."

```
function write_comma_tsv()
    path = "grades-comma.tsv"
    CSV.write(path, grades_with_commas(); delim='\t')
end
read(write_comma_tsv(), String)
```

---

```
name      grade_2020
Sally    1.0
Bob     5.0
Alice,   8.5
Hank    4.0
```

---

Também é possível encontrar formatos de arquivo de texto, como arquivos CSV e TSV, que usam outros delimitadores, como ponto-e-vírgula ";" espaços "," ou mesmo algo tão incomum como "π."

```
function write_space_separated()
    path = "grades-space-separated.csv"
    CSV.write(path, grades_2020(); delim=' ')
end
read(write_space_separated(), String)
```

```
name grade_2020
Sally 1.0
Bob 5.0
Alice 8.5
Hank 4.0
```

Por convenção, ainda assim é melhor dar a extensão ".csv" a arquivos com delimitadores especiais, como ";"

O carregamento de arquivos CSV usando `CSV.jl` é feito de maneira semelhante. Você pode usar `CSV.read` e especificar qual o formato do retorno da função. Nós especificamos um `DataFrame`.

```
path = write_grades_csv()
CSV.read(path, DataFrame)
```

	name	grade_2020
	Sally	1.0
	Bob	5.0
	Alice	8.5
	Hank	4.0

Convenientemente, `CSV.jl` irá inferir automaticamente os tipos de dados das colunas para nós:

```
path = write_grades_csv()
df = CSV.read(path, DataFrame)
```

4x2 DataFrame		
Row	name	grade_2020
1	Sally	1.0
2	Bob	5.0
3	Alice	8.5
4	Hank	4.0

Funciona mesmo com dados muito mais complexos:

```
my_data = """
    a,b,c,d,e
    Kim,2018-02-03,3,4.0,2018-02-03T10:00
"""
path = "my_data.csv"
write(path, my_data)
df = CSV.read(path, DataFrame)
```

1×5 DataFrame					
Row	a	b	c	d	e
	String <sup>3</sup>	Date	Int64	Float64	Datetime
1	Kim	2018-02-03	3	4.0	2018-02-03T10:00:00

Essas noções básicas de CSV devem abranger a maioria dos casos de uso. Para obter mais informações, consulte a documentação do `csv.jl`<sup>2</sup> e, especialmente, o docstring do construtor `CSV.File`<sup>3</sup>.

<sup>2</sup> <https://csv.juliadata.org/stable>

<sup>3</sup> <https://csv.juliadata.org/stable/#CSV.File>

#### 4.1.2 Excel

Existem vários pacotes Julia para ler arquivos Excel. Neste livro, nós vamos cobrir apenas o `XLSX.jl`<sup>4</sup>, porque é o pacote mais ativamente mantido no ecossistema Julia que lida com dados do Excel. Como um segundo benefício, `XLSX.jl` é escrito em Julia puro, o que torna mais fácil para nós inspecionarmos e entendermos o que está acontecendo nos bastidores.

<sup>4</sup> <https://github.com/felipenoris/XLSX.jl>

Carregue `XLSX.jl` via

```
using XLSX:
    eachablerow,
    readxlsx,
    writetable
```

Para escrever arquivos, definimos uma pequena função auxiliar para dados e nomes de colunas:

```
function write_xlsx(name, df::DataFrame)
    path = "$name.xlsx"
    data = collect(eachcol(df))
    cols = names(df)
    writetable(path, data, cols)
end
```

Agora, podemos escrever facilmente os dados das notas escolares em um arquivo Excel:

```
function write_grades_xlsx()
    path = "grades"
    write_xlsx(path, grades_2020())
    "$path.xlsx"
end
```

Ao ler de volta, veremos que `XLSX.jl` coloca os dados em um tipo `XLSXFile` e podemos acessar a aba desejada como um `Dict`:

```
path = write_grades_xlsx()
xf = readxlsx(path)
```

```
XLSXFile("grades.xlsx") containing 1 Worksheet
  sheetname size      range
-----
  Sheet1  5x2          A1:B5
```

```
xf = readxlsx(write_grades_xlsx())
sheet = xf["Sheet1"]
eachtblrow(sheet) |> DataFrame
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Observe que cobrimos apenas o básico de `XLSX.jl` mas usos mais poderosos e customizações estão disponíveis. Para obter mais informações e opções, consulte a documentação do módulo `XLSX.jl`<sup>5</sup>.

<sup>5</sup> <https://felipenoris.github.io/XLSX.jl/stable/>

## 4.2 Indexação e Sumarização

Vamos voltar para o exemplo dos dados `grades_2020()` definidos antes:

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0

name	grade_2020
Alice	8.5
Hank	4.0

Para recuperar um **vetor** para `name`, podemos acessar o `DataFrame` com o `.`, como fizemos anteriormente com `structs` em Section 3:

```
function names_grades1()
    df = grades_2020()
    df.name
end
names_grades1()
```

---

["Sally", "Bob", "Alice", "Hank"]

---

ou podemos indexar um `DataFrame` de modo muito parecido com uma `Array` utilizando símbolos e caracteres especiais. O **segundo índice é a indexação da coluna**:

```
function names_grades2()
    df = grades_2020()
    df[!, :name]
end
names_grades2()
```

---

["Sally", "Bob", "Alice", "Hank"]

---

Perceba que `df.name` é exatamente o mesmo que o comando `df[!, :name]`, o que você pode verificar fazendo:

```
julia> df = DataFrame(id=[1]);
julia> @edit df.name
```

Em ambos os casos, ele dará a coluna `:name`. Também existe o comando `df[:, :name]` que copia a coluna `:name`. Na maioria dos casos, `df[!, :name]` é a melhor aposta, pois é mais versátil e faz uma modificação no local.

Para qualquer **linha**, digamos a segunda linha, podemos usar o **primeiro índice como indexação de linha**:

```
df = grades_2020()
df[2, :]
```

name	grade_2020
Bob	5.0

ou criar uma função para nos dar qualquer linha  $i$  que quisermos:

```
function grade_2020(i::Int)
    df = grades_2020()
    df[i, :]
end
grade_2020(2)
```

name	grade_2020
Bob	5.0

Podemos também obter apenas a coluna `names` para as 2 primeiras linhas usando **fatiamento** (novamente, de modo similar a um `Array`):

```
grades_indexing(df) = df[1:2, :name]
grades_indexing(grades_2020())
```

---

["Sally", "Bob"]

---

Se assumirmos que todos os nomes na tabela são únicos, também podemos escrever uma função para obter a nota de uma pessoa por meio de seu `name ↛`. Para fazer isso, convertemos a tabela de volta para uma das estruturas de dados básicas de Julia (veja Section 3.2) que é capaz de criar mapeamentos, a saber `Dicts`:

```
function grade_2020(name::String)
    df = grades_2020()
    dic = Dict(zip(df.name, df.grade_2020))
    dic[name]
end
grade_2020("Bob")
```

---

5.0

---

que funciona porque `zip` itera pelo `df.name` e `df.grade_2020` ao mesmo tempo como um “zipper”:

```
df = grades_2020()
collect(zip(df.name, df.grade_2020))
```

---

```
("Sally", 1.0)
```

---



---

```
("Bob", 5.0)
```

---



---

```
("Alice", 8.5)
```

---



---

```
("Hank", 4.0)
```

---

Entretanto, converter um `DataFrame` para `Dict` só é útil quando os elementos são únicos. Geralmente esse não é o caso e é por isso que precisamos aprender como `filter` (filtrar) um `DataFrame`.

## 4.3 Filtro e Subconjunto

Existem duas maneiras de remover linhas de um `DataFrame`, uma é `filter` (Section 4.3.1) e outra é `subset` (Section 4.3.2). `filter` foi adicionado à biblioteca `DataFrames.jl` anteriormente, é mais poderoso e também tem uma sintaxe mais coerente em relação às bibliotecas básicas de Julia. É por isso que vamos iniciar essa seção discutindo `filter` primeiro. `subset` é mais recente e, comumente, é mais conveniente de usar.

### 4.3.1 Filtro

A partir de agora, nós começaremos a adentrar funcionalidades mais robustas da biblioteca `DataFrames.jl`. Para fazer isso, precisaremos aprender sobre algumas funções, como `select` e `filter`. Mas não se preocupe! Pode ser um alívio saber que o **objetivo geral do design de `DataFrames.jl` é manter o número de funções que um usuário deve aprender em um mínimo**<sup>6</sup>.

Como antes, retomamos a partir de `grades_2020`:

```
grades_2020()
```

<sup>6</sup> De acordo com Bogumił Kamiński (desenvolvedor e mantenedor líder do `DataFrames ↗.jl`) no Discourse (<https://discourse.julialang.org/t/pull-dataframes-columns-to-the-front/60327/5>).

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Podemos filtrar linhas usando `filter(source => f::Function, df)`. Perceba como

essa função é similar à função `filter(f::Function, V::Vector)` do módulo `Base` ↪ de Julia. Isso ocorre porque `DataFrames.jl` usa **despacho múltiplo** (see Section 2.3.3) para definir um novo método de `filter` que aceita `DataFrame` como argumento.

À primeira vista, definir e trabalhar com uma função `f` para filtrar pode ser um pouco difícil de se usar na prática. Aguente firme, esse esforço é bem pago, uma vez que é uma forma muito poderosa de filtrar dados. Como um exemplo simples, podemos criar uma função `equals_alice` que verifica se sua entrada é igual “Alice”:

```
equals_alice(name::String) = name == "Alice"
equals_alice("Bob")
```

---

false

---

```
equals_alice("Alice")
```

---

true

---

Equipados com essa função, podemos usá-la como nossa função `f` para filtrar todas as linhas para as quais `name` equivale a “Alice”:

```
filter(:name => equals_alice, grades_2020())
```

name	grade_2020
Alice	8.5

Observe que isso não funciona apenas para `DataFrame`, mas também para vetores:

```
filter(equals_alice, ["Alice", "Bob", "Dave"])
```

---

["Alice"]

---

Podemos torná-lo um pouco menos prolixo usando uma **função anônima** (veja Section 3.1.4):

```
filter(n -> n == "Alice", ["Alice", "Bob", "Dave"])
```

---

["Alice"]

---

que também podemos usar em `grades_2020`:

```
filter(:name => n -> n == "Alice", grades_2020())
```

name	grade_2020
Alice	8.5

Recapitulando, esta chamada de função pode ser lida como “para cada elemento na linha `:name`, vamos chamar o elemento `n`, e checar se `n` se iguala a Alice.” Para algumas pessoas, isso ainda é muito prolixo. Por sorte, Julia adicionou uma *aplicação de função parcial* de `==`. Os detalhes não são importantes – apenas saiba que você pode usá-la como qualquer outra função:

```
filter(:name => ==( "Alice"), grades_2020())
```

name	grade_2020
Alice	8.5

Para obter todas as linhas que *não* são Alice, `==` (igualdade) pode ser substituído por `!=` (desigualdade) em todos os exemplos anteriores:

```
filter(:name => !=("Alice"), grades_2020())
```

name	grade_2020
Sally	1.0
Bob	5.0
Hank	4.0

Agora, para mostrar **porque funções anônimas são tão poderosas**, podemos criar um filtro um pouco mais complexo. Neste filtro, queremos as pessoas cujos nomes começem com A ou B e tenham uma nota acima de 6:

```
function complex_filter(name, grade)::Bool
    interesting_name = startswith(name, 'A') || startswith(name, 'B')
    interesting_grade = 6 < grade
    interesting_name && interesting_grade
end
```

```
filter([:name, :grade_2020] => complex_filter, grades_2020())
```

name	grade_2020
Alice	8.5

### 4.3.2 Subconjunto

A função `subset` foi adicionada para tornar mais fácil trabalhar com valores ausentes (Section 4.5). Em contraste com `filter`, `subset` funciona em colunas completas ao invés de linhas ou valores únicos. Se quisermos usar nossas funções definidas anteriormente, devemos envolvê-las dentro de `ByRow`:

```
subset(grades_2020(), :name => ByRow(equals_alice))
```

name	grade_2020
Alice	8.5

Também perceba que `DataFrame` é agora o primeiro argumento `subset(df, args ↪ ...)`, enquanto que em `filter` foi o segundo `filter(f, df)`. A razão para isso é que Julia define filtro como `filter(f, V::Vector)` e `DataFrames.jl` optou por manter a consistência com as funções Julia existentes que foram estendidas para tipos de `DataFrames` de despacho múltiplo.

**OBSERVAÇÃO:** A maioria das funções nativas de `DataFrames.jl`, as quais `subset` → pertence, tem uma **assinatura de função consistente que sempre recebe um DataFrame como primeiro argumento**.

Assim como com `filter`, também podemos usar funções anônimas dentro de `subset`:

```
subset(grades_2020(), :name => ByRow(name -> name == "Alice"))
```

name	grade_2020
Alice	8.5

Ou, a aplicação de função parcial para `==`:

```
subset(grades_2020(), :name => ByRow(==( "Alice" )))
```

name	grade_2020
Alice	8.5

Em última análise, vamos mostrar o verdadeiro poder de `subset`. Primeiro, criamos um dataset com alguns valores ausentes:

```
function salaries()
    names = ["John", "Hank", "Karen", "Zed"]
    salary = [1_900, 2_800, 2_800, missing]
    DataFrame(; names, salary)
end
salaries()
```

names	salary
John	1900
Hank	2800
Karen	2800
Zed	missing

Table 4.27: Salaries.

Esses dados são sobre uma situação plausível em que você deseja descobrir os salários de seus colegas e ainda não descobriu o do Zed. Embora não queiramos incentivar essas práticas, suspeitamos que seja um exemplo interessante. Suponha que queremos saber quem ganha mais de 2.000. Se usarmos `filter`, sem levar em consideração os valores ‘faltantes,’ ele falhará:

```
filter(:salary => >(2_000), salaries())
```

---

```
TypeError: non-boolean (Missing) used in boolean context
Stacktrace:
[1] (::DataFrames.var"#99#100"{Base.Fix2{typeof(>), Int64}})(x::Missing)
@ DataFrames ~/julia/packages/DataFrames/dgZn3/src/abstractdataframe/
    ↪ abstractdataframe.jl:1178
...
...
```

`subset` também falhará, mas felizmente nos apontará para uma solução fácil:

```
subset(salaries(), :salary => ByRow(>(2_000)))
```

---

```
ArgumentError: missing was returned in condition number 1 but only true or false
    ↪ are allowed; pass skipmissing=true to skip missing values
Stacktrace:
[1] _and(x::Missing)
@ DataFrames ~/julia/packages/DataFrames/dgZn3/src/abstractdataframe/subset
    ↪ .jl:11
```

...

Então, só precisamos passar o argumento de palavra-chave `skipmissing=true`:

```
subset(salaries(), :salary => ByRow(>(2_000)); skipmissing=true)
```

names	salary
Hank	2800
Karen	2800

## 4.4 Select

Enquanto `filter` remove linhas, `select` remove colunas. Entretanto, `select` é muito mais versátil do que apenas remover colunas, como discutiremos nesta seção. Primeiro, vamos criar um dataset com múltiplas colunas:

```
function responses()
    id = [1, 2]
    q1 = [28, 61]
    q2 = [:us, :fr]
    q3 = ["F", "B"]
    q4 = ["B", "C"]
    q5 = ["A", "E"]
    DataFrame(; id, q1, q2, q3, q4, q5)
end
responses()
```

id	q1	q2	q3	q4	q5
1	28	us	F	B	A
2	61	fr	B	C	E

Table 4.29: Responses.

Aqui, os dados representam respostas para cinco perguntas (`q1`, `q2`, ..., `q5`) em um determinado questionário. Começaremos “selecionando” algumas colunas deste dataset. Como de costume, usamos símbolos para especificar colunas:

```
select(responses(), :id, :q1)
```

id	q1
1	28
2	61

Também podemos usar strings se quisermos:

```
select(responses(), "id", "q1", "q2")
```

id	q1	q2
1	28	us
2	61	fr

Para selecionar tudo *menos* uma ou mais colunas, use `Not` com a coluna que não se deseja selecionar:

```
select(responses(), Not(:q5))
```

id	q1	q2	q3	q4
1	28	us	F	B
2	61	fr	B	C

Ou, com múltiplas colunas:

```
select(responses(), Not([:q4, :q5]))
```

id	q1	q2	q3
1	28	us	F
2	61	fr	B

É possível também misturar e combinar colunas que queremos preservar com colunas que não (ou `Not`) queremos selecionar:

```
select(responses(), :q5, Not(:id))
```

q5	q1	q2	q3	q4
A	28	us	F	B
E	61	fr	B	C

Perceba como `q5` agora é a primeira coluna no `DataFrame` retornado por `select`. Existe uma maneira mais inteligente de conseguir o mesmo usando `:`. O caractere de dois pontos `:` pode ser pensado como “todas as colunas que ainda não incluímos.” Por exemplo:

```
select(responses(), :q5, :)
```

q5	id	q1	q2	q3	q4
A	1	28	us	F	B
E	2	61	fr	B	C

Ou, para colocar `q5` na segunda posição<sup>7</sup>:

```
select(responses(), 1, :q5, :)
```

id	q5	q1	q2	q3	q4
1	A	28	us	F	B
2	E	61	fr	B	C

<sup>7</sup> obrigado ao Sudete pela sugestão no Discourse (<https://discourse.julialang.org/t/pull-dataframes-columns-to-the-front/60327/4>).

**OBSERVAÇÃO:** Como você deve ter observado, existem várias maneiras de selecionar uma coluna. Elas são conhecidas como *seletores de coluna*<sup>8</sup>.

Podemos usar:

- **Symbol:** `select(df, :col)`
- **String:** `select(df, "col")`
- **Integer:** `select(df, 1)`

<sup>8</sup> <https://bkamins.github.io/julialang/2021/02/06/colsel.html>

Até mesmo renomear colunas é possível via `select` usando a sintaxe de par origem => destino:

```
select(responses(), 1 => "participant", :q1 => "age", :q2 => "nationality")
```

participant	age	nationality
1	28	us
2	61	fr

Além disso, graças ao operador “splat” ... (see Section 3.2.10), também podemos escrever:

```
renames = (1 => "participant", :q1 => "age", :q2 => "nationality")
select(responses(), renames...)
```

participant	age	nationality
1	28	us
2	61	fr

## 4.5 Tipos de Dados e Dados Faltantes

Como discutido em Section 4.1, `CSV.jl` fará o seu melhor para adivinhar os tipo de dados das colunas de sua tabela. No entanto, isso nem sempre funcionará perfeitamente. Nesta seção, mostramos porque os tipos adequados são importantes e corrigimos os tipos de dados incorretos. Para ser mais claro sobre os tipos, mostramos a saída de texto para `DataFrames` ao invés de uma tabela bem formatada. Nesta seção, trabalharemos com o seguinte dataset:

```
function wrong_types()
    id = 1:4
    date = ["28-01-2018", "03-04-2019", "01-08-2018", "22-11-2020"]
    age = ["adolescent", "adult", "infant", "adult"]
    DataFrame(; id, date, age)
end
wrong_types()
```

4x3 DataFrame			
Row	id	date	age
	Int64	String	String
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	3	01-08-2018	infant
4	4	22-11-2020	adult

Como a coluna de data tem o tipo incorreto, a ordenação dessa coluna não funcionará corretamente:

```
sort(wrong_types(), :date)
```

4x3 DataFrame			
Row	id	date	age
	Int64	String	String
1	3	01-08-2018	infant
2	2	03-04-2019	adult
3	4	22-11-2020	adult
4	1	28-01-2018	adolescent

Para corrigir a ordenação, podemos usar o módulo `Date` da biblioteca padrão de Julia, conforme descrito em Section 3.4.1:

```
function fix_date_column(df::DataFrame)
    strings2dates(dates::Vector) = Date.(dates, DateFormat("dd-mm-yyyy"))
    dates = strings2dates(df[!, :date])
```

```

df[!, :date] = dates
df
end
fix_date_column(wrong_types())

```

**4x3 DataFrame**

Row	id	date	age
	Int64	Date	String
1	1	2018-01-28	adolescent
2	2	2019-04-03	adult
3	3	2018-08-01	infant
4	4	2020-11-22	adult

Agora, a ordenação funcionará conforme o planejado:

```

df = fix_date_column(wrong_types())
sort(df, :date)

```

**4x3 DataFrame**

Row	id	date	age
	Int64	Date	String
1	1	2018-01-28	adolescent
2	3	2018-08-01	infant
3	2	2019-04-03	adult
4	4	2020-11-22	adult

Para a coluna `age` (idade), temos um problema semelhante:

```
sort(wrong_types(), :age)
```

**4x3 DataFrame**

Row	id	date	age
	Int64	String	String
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	4	22-11-2020	adult
4	3	01-08-2018	infant

Isso não está certo, porque uma criança é mais jovem do que adultos e adolescentes. A solução para este problema e para qualquer tipo de dado categórico é usar `CategoricalArrays.jl`:

```
using CategoricalArrays
```

Com o pacote `CategoricalArrays.jl`, podemos adicionar níveis que representam a ordem da variável categórica em questão para nossos dados:

```
function fix_age_column(df)
    levels = ["infant", "adolescent", "adult"]
    ages = categorical(df[!, :age]; levels, ordered=true)
    df[!, :age] = ages
    df
end
fix_age_column(wrong_types())
```

**4x3 DataFrame**

Row	id	date	age
	Int64	String	Cat...
1	1	28-01-2018	adolescent
2	2	03-04-2019	adult
3	3	01-08-2018	infant
4	4	22-11-2020	adult

**OBSERVAÇÃO:** Observe também que estamos passando o argumento `ordered →= true` que diz para a função `categorical` do módulo `CategoricalArrays.jl` que nossos dados categóricos são “ordenados.” Sem isso, qualquer tipo de ordenação ou de comparações do tipo maior/menor não seria possível.

Agora, podemos classificar os dados corretamente na coluna `age` (idade):

```
df = fix_age_column(wrong_types())
sort(df, :age)
```

**4x3 DataFrame**

Row	id	date	age
	Int64	String	Cat...
1	3	01-08-2018	infant
2	1	28-01-2018	adolescent
3	2	03-04-2019	adult
4	4	22-11-2020	adult

Como definimos funções convenientes, agora podemos definir nossos dados fixos apenas executando as chamadas de função:

```
function correct_types()
    df = wrong_types()
    df = fix_date_column(df)
    df = fix_age_column(df)
```

```
end
correct_types()
```

---

**4x3 DataFrame**

Row	id	date	age
	Int64	Date	Cat...
1	1	2018-01-28	adolescent
2	2	2019-04-03	adult
3	3	2018-08-01	infant
4	4	2020-11-22	adult

---

Já que a variável `age` (idade) em nossos dados é ordinal (`ordered=true`), podemos comparar adequadamente as categorias de idade:

```
df = correct_types()
a = df[1, :age]
b = df[2, :age]
a < b
```

---

true

---

o que daria comparações erradas se o tipo de elemento fosse strings:

```
"infant" < "adult"
```

---

false

---

## 4.6 Join

No início deste capítulo, mostramos várias tabelas e levantamos questões também relacionadas às várias tabelas. No entanto, não falamos sobre como combinar de tabelas ainda, o que faremos nesta seção. Em `DataFrames.jl`, a combinação de várias tabelas é feita via *joins*. Os joins são extremamente poderosos, mas pode demorar um pouco para você entendê-los. Não é necessário saber os joins abaixo de cor, porque a documentação `DataFrames.jl`<sup>9</sup>, juntamente com este livro, listará-los para você. Mas, é essencial saber que os joins existem. Se você alguma vez se pegar dando voltas sobre as linhas de um `DataFrame` e comparando-o com outros dados, então você provavelmente precisará de um dos joins abaixo.

No Section 4, introduzimos as notas escolares para o ano de 2020 com `grades_→2020`:

<sup>9</sup> <https://DataFrames.juliadata.org/stable/man/joins/>

```
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Agora, vamos combinar `grades_2020` com as notas de 2021:

```
grades_2021()
```

name	grade_2021
Bob	9.5
Sally	9.5
Hank	6.0

Para fazer isso, vamos utilizar os joins. `DataFrames.jl` lista não menos que sete tipos de join. Isso pode parecer assustador no início, mas espere porque todos eles são úteis e vamos mostrá-los.

#### 4.6.1 *innerjoin*

O primeiro é `innerjoin`. Suponha que temos dois datasets A e B com as respectivas colunas `A_1`, `A_2`, ..., `A_n` e `B_1`, `B_2`, ..., `B_m` e uma das colunas tem o mesmo nome, digamos `A_1` e `B_1` são ambas chamadas `:id`. Então, o inner join em `:id` irá percorrer todos os elementos em `A_1` e compará-lo aos elementos em `B_1`. Se os elementos são **os mesmos**, então ele irá adicionar todas as informações de `A_2`, ..., `A_n` e `B_2`, ..., `B_m` depois da coluna `:id`.

Ok, não se preocupe se você ainda não compreendeu esta descrição. O resultado do join nos datasets de notas será assim:

```
innerjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0

Observe que apenas “Sally” e “Hank” estão em ambos datasets. O nome *inner*

`join` faz sentido, uma vez que, em matemática, o *set intersection* (ou a intersecção de conjuntos) é definido por “todos os elementos em  $A$ , que também estão em  $B$ , ou todos os elementos em  $B$  que também estão em  $A$ .”

#### 4.6.2 *outerjoin*

Talvez você esteja pensando agora “se temos um *inner* (interno), provavelmente também temos um *outer* (externo).” Sim, você adivinhou certo!

O `outerjoin` é muito menos rigoroso do que o `innerjoin` e pega qualquer linha que encontrar que contenha um nome **em pelo menos um dos datasets**:

```
outerjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob	5.0	missing
Alice	8.5	missing
Bob 2	missing	9.5

Portanto, este método pode criar dados `faltantes` mesmo que nenhum dos datasets originais tivesse valores ausentes.

#### 4.6.3 *crossjoin*

Podemos obter ainda mais dados `faltantes` se usarmos o `crossjoin`. Esse tipo de join retorna o **produto cartesiano das linhas**, que é basicamente a multiplicação das linhas, ou seja, para cada linha crie uma combinação com qualquer outra linha:

```
crossjoin(grades_2020(), grades_2021(); on=:id)
```

```
MethodError: no method matching crossjoin(::DataFrame, ::DataFrame; on=:id)
Closest candidates are:
  crossjoin(::DataFrames.AbstractDataFrame, ::DataFrames.AbstractDataFrame;
    ↪makeunique) at ~/.julia/packages/DataFrames/dgZn3/src/join/composer.jl:14
    ↪12 got unsupported keyword argument "on"
  crossjoin(::DataFrames.AbstractDataFrame, ::DataFrames.AbstractDataFrame, !
    ↪Matched::DataFrames.AbstractDataFrame...; makeunique) at ~/.julia/
    ↪packages/DataFrames/dgZn3/src/join/composer.jl:1434 got unsupported
    ↪keyword argument "on"
...
...
```

Ops! Já que `crossjoin` não leva os elementos na linha em consideração, não precisamos especificar o argumento `on` para o que queremos juntar:

```
crossjoin(grades_2020(), grades_2021())
```

---

ArgumentError: Duplicate variable names: :name. Pass `makeunique=true` to make  
them unique using a suffix automatically.

Stacktrace:

```
[1] add_names(ind::DataFrames.Index, add_ind::DataFrames.Index; makeunique::  
    Bool)  
    @ DataFrames ~/julia/packages/DataFrames/dgZn3/src/other/index.jl:434  
[2] merge!(x::DataFrames.Index, y::DataFrames.Index; makeunique::Bool)  
...
```

---

Ops de novo! Esse é um erro bastante comum com `DataFrames` e `joins`. As tabelas para as notas de 2020 e 2021 têm um nome de coluna duplicado, a saber `:name`. Como antes, o erro do output de `DataFrames.jl` mostra uma sugestão simples que pode corrigir o problema. Podemos apenas passar `makeunique=true` para resolver isso:

```
crossjoin(grades_2020(), grades_2021(); makeunique=true)
```

name	grade_2020	name_1	grade_2021
Sally	1.0	Bob 2	9.5
Sally	1.0	Sally	9.5
Sally	1.0	Hank	6.0
Bob	5.0	Bob 2	9.5
Bob	5.0	Sally	9.5
Bob	5.0	Hank	6.0
Alice	8.5	Bob 2	9.5
Alice	8.5	Sally	9.5
Alice	8.5	Hank	6.0
Hank	4.0	Bob 2	9.5
Hank	4.0	Sally	9.5
Hank	4.0	Hank	6.0

Então, agora, temos uma linha para cada nota de todos nos datasets de notas de 2020 e 2021. Para consultas diretas, como “quem tem a nota mais alta?” o produto cartesiano geralmente não é tão útil, mas para consultas “estatísticas,” pode ser.

#### 4.6.4 `leftjoin` e `rightjoin`

**Mais úteis para projetos de dados científicos são os `leftjoin` e `rightjoin`.** O `leftjoin` (ou `join` à esquerda) fornece todos os elementos do `DataFrame` à esquerda:

```
leftjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob	5.0	missing
Alice	8.5	missing

Aqui, as notas para “Bob” e “Alice” estavam faltando na tabela de notas de 2021, então é por isso que também existem elementos faltantes. A join à direita faz quase que o oposto:

```
rightjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob 2	missing	9.5

Agora, as notas de 2020 estão faltando.

Perceba que `leftjoin(A, B) != rightjoin(B, A)`, porque a ordem das colunas será diferente. Por exemplo, compare o output abaixo com o output anterior:

```
leftjoin(grades_2021(), grades_2020(); on=:name)
```

name	grade_2021	grade_2020
Sally	9.5	1.0
Hank	6.0	4.0
Bob 2	9.5	missing

#### 4.6.5 semijoin e antijoin

Por último, temos `semijoin` e `antijoin`.

O semi join é ainda mais restritivo que o inner join. Retorna **apenas elementos do DataFrame da esquerda que estão em ambos** DataFrames. Ele é como se fosse uma combinação do join da esquerda com o inner join.

```
semijoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020
Sally	1.0
Hank	4.0

O oposto do semi join é o anti join. Ele retorna **apenas os elementos do DataFrame da esquerda que não estão no DataFrame da direita**:

```
antijoin(grades_2020(), grades_2021(); on:=name)
```

name	grade_2020
Bob	5.0
Alice	8.5

## 4.7 Transformações de Variáveis

Em Section 4.3.1, vimos que `filter` funciona pegando uma ou mais colunas de origem e filtrando-as por meio da aplicação de uma função de “filtragem.” Para recapitular, aqui está um exemplo de filtro usando a sintaxe `source => f :: Function: filter(:name => name -> name == "Alice", df)`.

Em Section 4.4, vimos que `select` pode pegar uma ou mais colunas de origem e colocá-las em uma ou mais colunas de destino `source => target`. Também para recapitular aqui está um exemplo: `select(df, :name => :people_names)`.

Nesta seção, discutimos como **transformar** variáveis, ou seja, como **modificar dados**. Em `DataFrames.jl`, a sintaxe é `source => transformation => target`.

Como antes, usamos o dataset `grades_2020`:

```
function grades_2020()
    name = ["Sally", "Bob", "Alice", "Hank"]
    grade_2020 = [1, 5, 8.5, 4]
    DataFrame(; name, grade_2020)
end
grades_2020()
```

name	grade_2020
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0

Suponha que queremos aumentar todas as notas em `grades_2020` por 1. Primeiro, definimos uma função que leva como argumento um vetor de dados e retorna todos os seus elementos acrescidos de 1. Depois usamos a função `transform` do `DataFrames.jl` que, como todas as funções nativas `DataFrames.jl`, tem como primeiro argumento um `DataFrame`, seguido pela sintaxe de transformação:

```
plus_one(grades) = grades .+ 1
transform(grades_2020(), :grade_2020 => plus_one)
```

name	grade_2020	grade_2020_plus_one
Sally	1.0	2.0
Bob	5.0	6.0
Alice	8.5	9.5
Hank	4.0	5.0

Aqui, a função `plus_one` recebe toda a coluna `:grade_2020`. Essa é a razão pela qual adicionamos o caractere de “ponto” do broadcasting `.` antes do operador `+`. Para uma recapitulação sobre broadcasting por gentileza veja Section 3.2.1.

Como dissemos acima, a minilinguagem do módulo `DataFrames.jl` é sempre `source => transformation => target`. Então, se quisermos manter a nomenclatura da coluna alvo (`target`) na coluna de output podemos fazer:

```
transform(grades_2020(), :grade_2020 => plus_one => :grade_2020)
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

Também podemos usar o argumento de palavra-chave `renamecols=false`:

```
transform(grades_2020(), :grade_2020 => plus_one; renamecols=false)
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

A mesma transformação também pode ser escrita com `select`:

```
select(grades_2020(), :, :grade_2020 => plus_one => :grade_2020)
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

onde o `:` significa “selecione todas as colunas” como descrito em Section 4.4. Como alternativa, você também pode utilizar o broadcasting de Julia e modificar a coluna `grade_2020` acessando-a com `df.grade_2020`:

```
df = grades_2020()
df.grade_2020 = plus_one.(df.grade_2020)
df
```

name	grade_2020
Sally	2.0
Bob	6.0
Alice	9.5
Hank	5.0

Mas, embora o último exemplo seja mais fácil, pois se baseia em operações nativas de Julia, é **altamente recomendável usar as funções fornecidas por `DataFrames.jl` na maioria dos casos porque são mais robustas e fáceis de trabalhar.**

#### 4.7.1 Transformações Múltiplas

Para mostrar como transformar duas colunas ao mesmo tempo, usamos os dados sobre os quais realizamos o left join em Section 4.6:

```
leftjoined = leftjoin(grades_2020(), grades_2021(); on=:name)
```

name	grade_2020	grade_2021
Sally	1.0	9.5
Hank	4.0	6.0
Bob	5.0	missing
Alice	8.5	missing

Com isso, podemos adicionar uma coluna dizendo se alguém foi aprovado pelo critério de que todas as suas notas estivessem acima 5.5:

```
pass(A, B) = [5.5 < a || 5.5 < b for (a, b) in zip(A, B)]
transform(leftjoined, [:grade_2020, :grade_2021] => pass; renamecols=false)
```

name	grade_2020	grade_2021	grade_2020_grade_2021
Sally	1.0	9.5	true
Hank	4.0	6.0	true
Bob	5.0	missing	missing
Alice	8.5	missing	true

Podemos limpar o resultado e colocar a lógica em uma função para obter uma lista de todos os alunos aprovados:

```
function only_pass()
    leftjoined = leftjoin(grades_2020(), grades_2021(); on=:name)
    pass(A, B) = [5.5 < a || 5.5 < b for (a, b) in zip(A, B)]
    leftjoined = transform(leftjoined, [:grade_2020, :grade_2021] => pass => :
        ↪pass)
    passed = subset(leftjoined, :pass; skipmissing=true)
    return passed.name
end
only_pass()
```

---

```
["Sally", "Hank", "Alice"]
```

---

## 4.8 Groupby e Combine

Para a linguagem de programação R, Wickham (2011) popularizou a chamada estratégia `_split-apply-combine` (ou dividir-aplicar-combinar) para transformações de dados. Em essência, essa estratégia **divide** o dataset em grupos distintos, **aplica** uma ou mais funções para cada grupo e, depois, **combina** o resultado. `DataFrames.jl` suporta totalmente o método dividir-aplicar-combinar. Usaremos o exemplo das notas do aluno como antes. Suponha que queremos saber a nota média de cada aluno:

```
function all_grades()
    df1 = grades_2020()
    df1 = select(df1, :name, :grade_2020 => :grade)
    df2 = grades_2021()
    df2 = select(df2, :name, :grade_2021 => :grade)
    rename_bob2(data_col) = replace.(data_col, "Bob 2" => "Bob")
    df2 = transform(df2, :name => rename_bob2 => :name)
    return vcat(df1, df2)
end
all_grades()
```

name	grade
Sally	1.0
Bob	5.0
Alice	8.5
Hank	4.0
Bob	9.5
Sally	9.5
Hank	6.0

A estratégia é **dividir** o dataset em alunos distintos, **aplicar** a função média para cada aluno e **combinar** o resultado.

A divisão é chamada `groupby` e passamos como segundo argumento o ID da coluna em que queremos dividir o dataset:

```
groupby(all_grades(), :name)
```

```
GroupedDataFrame with 4 groups based on key: name
Group 1 (2 rows): name = "Sally"
Row | name      grade
| String    Float64
-----
1 | Sally     1.0
2 | Sally     9.5
Group 2 (2 rows): name = "Bob"
Row | name      grade
| String    Float64
-----
1 | Bob       5.0
2 | Bob       9.5
Group 3 (1 row): name = "Alice"
Row | name      grade
| String    Float64
-----
1 | Alice     8.5
Group 4 (2 rows): name = "Hank"
Row | name      grade
| String    Float64
-----
1 | Hank     4.0
2 | Hank     6.0
```

Nós aplicamos a função `mean` da biblioteca padrão de Julia no módulo `Statistics`

```
using Statistics
```

Para aplicá-la, utilizamos a função `combine`:

```
gdf = groupby(all_grades(), :name)
combine(gdf, :grade => mean)
```

name	grade_mean
Sally	5.25
Bob	7.25
Alice	8.5
Hank	5.0

Imagine ter que fazer isso sem as funções `groupby` e `combine`. Precisaríamos iterar sobre nossos dados para dividi-los em grupos, em seguida, iterar sobre os registros em cada divisão para aplicar a função `e`, finalmente, iterar sobre cada grupo para obter o resultado final. Vê-se assim que é muito bom conhecer a técnica dividir-aplicar-combinar.

#### 4.8.1 Múltiplas Colunas de Origem

Mas, e se quisermos aplicar uma função à várias colunas de nosso dataset?

```
group = [:A, :A, :B, :B]
X = 1:4
Y = 5:8
df = DataFrame(; group, X, Y)
```

group	X	Y
A	1	5
A	2	6
B	3	7
B	4	8

Isso é feito de maneira semelhante:

```
gdf = groupby(df, :group)
combine(gdf, [:X, :Y] .=> mean; renamecols=false)
```

group	X	Y
A	1.5	5.5
B	3.5	7.5

Perceba que usamos o operador dot `.` antes da seta à direita `=>` para indicar que o `mean` tem que ser aplicado a múltiplas colunas de origem `[:X, :Y]`.

Para usar funções compostas, uma maneira simples é criar uma função que faça as transformações compostas pretendidas. Por exemplo, para uma série de valores, vamos primeiro pegar o `mean` seguido de `round` para um número inteiro (também conhecido como um inteiro `Int`):

```
gdf = groupby(df, :group)
rounded_mean(data_col) = round(Int, mean(data_col))
combine(gdf, [:X, :Y] .=> rounded_mean; renamecols=false)
```

group	X	Y
A	2	6
B	4	8

## 4.9 Desempenho

Até agora, não pensamos em fazer nosso código `DataFrames.jl` **rápido**. Como tudo em Julia, `DataFrames.jl` pode ser bem veloz. Nesta seção, daremos algumas dicas e truques de desempenho.

### 4.9.1 Operações *in-loco*

Como explicamos em Section 3.2.1, funções que terminam com uma exclamação ! são um padrão comum para denotar funções que modificam um ou mais de seus argumentos. O contexto do código de alta performance em Julia, *significa* que \*\*funções com ! apenas mudarão no local os objetos que fornecemos como argumentos.

Quase todas as funções `DataFrames.jl` que vimos tem uma “! gêmea”. Por exemplo, `filter` tem um *in-loco* `filter!`, `select` tem `select!`, `subset` tem `subset!`, e assim por diante. Observe que essas funções **não** retornam um novo `DataFrame ↩`, mas, ao invés vez disso, elas **atualizam** o `DataFrame` sobre o qual atuam. Além disso, `DataFrames.jl` (versão 1.3 em diante) suporta *in-loco* `leftjoin` com a função `leftjoin!`. Essa função atualiza o `DataFrame` esquerdo com as colunas unidas do `DataFrame` direito. Há uma ressalva de que cada linha da tabela esquerda deve corresponder a *no máximo* uma linha da tabela direita.

Se você deseja a mais alta velocidade e desempenho em seu código, definitivamente deve usar as funções ! ao invés das funções regulares de `DataFrames.jl`.

Vamos voltar para o exemplo da função `select` no começo de Section 4.4. Aqui está o `DataFrame` `responses`:

```
responses()
```

id	q1	q2	q3	q4	q5
1	28	us	F	B	A
2	61	fr	B	C	E

Agora vamos desempenhar a seleção com a função `select`, como fizemos antes:

```
select(responses(), :id, :q1)
```

id	q1
1	28
2	61

Aqui está a função *in-loco*:

```
select!(responses(), :id, :q1)
```

id	q1
1	28
2	61

O macro `@allocated` nos diz quanta memória foi alocada. Em outras palavras, **quanta informação nova o computador teve que armazenar em sua memória enquanto executava o código**. Vamos ver qual será o desempenho:

```
df = responses()
@allocated select(df, :id, :q1)
```

6976

```
df = responses()
@allocated select!(df, :id, :q1)
```

6752

Como pudemos ver, `select!` aloca menos que `select`. Portanto, é mais rápido e consome menos memória.

#### 4.9.2 Copiar vs Não Copiar Colunas

Existem **duas formas de acessar a coluna DataFrame**. Elas diferem na forma como são acessadas: uma cria uma “visualização” para a coluna sem copiar e

a outra cria uma coluna totalmente nova copiando a coluna original.

A primeira usa o operador dot regular `.` seguido pelo nome da coluna, como em `df.col`. Essa forma de acesso **não copia** a coluna `col`. Ao invés disso `df.col` → cria uma “visualização” que é um link para a coluna original sem realizar nenhuma alocação. Além do mais, a sintaxe `df.col` é a mesma que `df[!, :col]` com a exclamação `!` como a seletora de linha.

A segunda forma de acessar uma coluna `DataFrame` é a `df[:, :col]` com os dois pontos `:` como o seletor de linha. Esse tipo de acesso **copia** a coluna `col`, portanto, tenha cuidado, pois isso pode produzir alocações indesejadas.

Como antes, vamos experimentar essas duas maneiras de acessar uma coluna no `DataFrame responses`:

```
df = responses()
@allocated col = df[:, :id]
```

555774

```
df = responses()
@allocated col = df[!, :id]
```

0

Quando acessamos uma coluna sem copiá-la estamos fazendo alocações zero e nosso código deve ser mais rápido. Então, se você não precisa de uma cópia, sempre accesse suas colunas `DataFrames` com `df.col` ou `df[!, :col]` ao invés de `df[:, :col]`.

#### 4.9.3 `CSV.read` versus `CSV.File`

Se você der uma olhada no output ajuda para `CSV.read`, você verá que existe uma função de conveniência idêntica à função chamada `CSV.File` com os mesmos argumentos de palavras-chave. Ambos `CSV.read` e `CSV.File` vão ler o conteúdo de um arquivo CSV, mas eles se diferem no comportamento padrão. `CSV.read`, por padrão, **não fará cópias** dos dados de entrada. Ao invés disso, `CSV.read` irá passar todos os dados para o segundo argumento (conhecido como “sink”).

Então, algo assim:

```
df = CSV.read("file.csv", DataFrame)
```

passará todos os dados recebidos de `file.csv` para o sink `DataFrame`, retornando assim um tipo `DataFrame` que vamos armazenar na variável `df`.

Para o caso do `CSV.File`, o comportamento padrão é o oposto: ele fará cópias de todas as colunas contidas no arquivo CSV. Além disso, a sintaxe é um pouco diferente. Precisamos embrulhar tudo que o `CSV.File` retorna em uma função construtora `DataFrame`:

```
df = DataFrame(CSV.File("file.csv"))
```

Ou, com o operador pipe |>:

```
df = CSV.File("file.csv") |> DataFrame
```

Como dissemos, `CSV.File` fará cópias de cada coluna no arquivo CSV subjacente. Em última análise, se você quiser o máximo de desempenho, você definitivamente usaria `CSV.read` em vez de `CSV.File`. É por isso que cobrimos apenas `CSV.read` em Section 4.1.1.

#### 4.9.4 Múltiplos Arquivos CSV.jl

Agora vamos voltar nossa atenção para o `CSV.jl`. Especificamente, o caso em que temos vários arquivos CSV para ler em um único `DataFrame`. Desde a versão 0.9 do `CSV.jl` podemos fornecer um vetor de strings representando nomes de arquivos. Antes, precisávamos realizar algum tipo de leitura de vários arquivos e, em seguida, concatenar verticalmente os resultados em um único `DataFrame`. Para exemplificar, o código abaixo lê vários arquivos CSV e os concatena verticalmente usando `vcat` em um único `DataFrame` com a função `reduce`:

```
files = filter(endswith(".csv"), readdir())
df = reduce(vcat, CSV.read(file, DataFrame) for file in files)
```

Uma característica adicional é que `reduce` não será paralelizado porque precisa manter a ordem de `vcat` que segue a mesma ordem do vetor `files`.

Com esta funcionalidade em `CSV.jl` nós simplesmente passamos o vetor `files` para a função `CSV.read`:

```
files = filter(endswith(".csv"), readdir())
df = CSV.read(files, DataFrame)
```

`CSV.jl` designará um arquivo para cada thread disponível no computador enquanto ele concatena lentamente cada saída analisada por thread em um `DataFrame` ↵. Portanto, temos o **benefício adicional do multithreading** que não temos com a opção `reduce`.

#### 4.9.5 Compressão CategoricalArrays.jl

Se você estiver lidando com dados com muitos valores categóricos, ou seja, muitas colunas com dados textuais que representam dados qualitativos de alguma forma diferentes, você provavelmente se beneficiaria usando a compressão `CategoricalArrays.jl`.

Por padrão, `CategoricalArrays.jl` usará um inteiro sem sinal no tamanho de 32 bits `UInt32` para representar as categorias subjacentes:

```
typeof(categorical(["A", "B", "C"]))
```

```
CategoricalVector{String, UInt32, String, CategoricalValue{String, UInt32}, Union{}}
```

Isso significa que `CategoricalArrays.jl` pode representar até  $2^{32}$  categorias diferentes em um determinado vetor ou coluna, o que é um valor enorme (perto de 4,3 bilhões). Você provavelmente nunca precisaria ter esse tipo de capacidade para lidar com dados regulares<sup>10</sup>. É por isso que `categorical` tem um argumento `compress` que aceita `true` ou `false` para determinar se os dados categóricos subjacentes são compactados ou não. Se você passar `compress=true`, `CategoricalArra- ↵.jl` tentará compactar os dados categóricos subjacentes para a menor rep- resentação possível em `UInt`. Por exemplo, o vetor `categorical` anterior seria representado como um inteiro sem sinal de tamanho 8 bits `UInt8` (principa- mente porque este é o menor inteiro sem sinal disponível em Julia):

```
typeof(categorical(["A", "B", "C"]; compress=true))
```

```
CategoricalVector{String, UInt8, String, CategoricalValue{String, UInt8}, Union{}}
```

O que tudo isso significa? Suponha que você tenha um grande vetor. Por exemplo, considere um vetor com um milhão de entradas, mas apenas 4 categorias subjacentes: A, B, C ou D. Se você não compactar o vetor categórico resultante, você terá um milhão de entradas armazenadas como `UInt32`. Por outro lado, se você compactar, você terá um milhão de entradas armazenadas como `UInt8`. Usando a função `Base.summarysize` podemos obter o tamanho subjacente, em bytes, de um determinado objeto. Então, vamos quantificar quanta memória mais precisaríamos ter se não comprimissemos nosso um milhão de vetores categóricos:

```
using Random
```

<sup>10</sup> observe também que dados regulares (até 10.000 linhas) não são big data (mais de 100.000 linhas). Portanto, se você estiver lidando principalmente com big data, tenha cuidado ao limitar seus valores categóricos.

```
one_mi_vec = rand(["A", "B", "C", "D"], 1_000_000)
Base.summarysize(categorical(one_mi_vec))
```

```
4000612
```

4 milhões de bytes, que é aproximadamente 3,8 MB. Não nos entenda mal, esta é uma boa melhoria em relação ao tamanho da string bruta:

```
Base.summarysize(one_mi_vec)
```

```
8000076
```

Reduzimos 50% do tamanho dos dados brutos usando uma representação subjacente padrão `CategoricalArrays.jl` como `UInt32`.

Agora vamos ver como nos sairíamos com a compressão:

```
Base.summarysize(categorical(one_mi_vec; compress=true))
```

```
1000564
```

Reduzimos o tamanho para 25% (um quarto) do tamanho original do vetor não compactado sem perder informações. Nosso vetor categórico compactado agora tem 1 milhão de bytes, que é aproximadamente 1,0 MB.

Portanto, sempre que possível, no interesse do desempenho, considere usar `compress=true` em seus dados categóricos.



## 5 Visualização de dados com Makie.jl

Da palavra japonesa Maki-e, que é uma técnica para polvilhar verniz com pó de ouro e prata. Os dados são o ouro e a prata da nossa era, então vamos espalhá-los lindamente pela tela!

*Simon Danisch, criador do Makie.jl*

Makie.jl<sup>1</sup> é um ecossistema de plotagem de alto desempenho, extensível e multiplataforma para a linguagem de programação Julia. Na nossa opinião, é o pacote de plotagem mais bonito e versátil.

<sup>1</sup> <http://makie.juliaplots.org/stable/index.html>

Assim como muitos pacotes de plotagem, o código é dividido em vários pacotes. Makie.jl é o pacote de *frontend* que define todas as funções de plotagem necessárias para criar objetos de plotagem. Esses objetos armazenam todas as informações sobre as plotagens, mas ainda precisam ser convertidos em uma imagem. Para converter esses objetos de plotagem em uma imagem, você precisa de um dos *backends* Makie. Por padrão, o Makie.jl é reexportado por cada *backend*, então você só precisa instalar e carregar o *backend* que deseja usar.

Existem três *backends* principais que implementam concretamente todos os recursos de renderização abstratos definidos no Makie. Um para gráficos vetoriais 2D não interativos com qualidade de publicação: CairoMakie.jl. Outro para plotagem 2D e 3D interativa em janelas GLFW.jl independentes (que também rodam na GPU), GLMakie.jl. E o terceiro, uma plotagem 2D e 3D interativa baseada em WebGL que roda dentro de navegadores, WGLMakie.jl. Veja a documentação de Makie<sup>2</sup>.

<sup>2</sup> [http://makie.juliaplots.org/stable/documentation/backends\\_and\\_output/](http://makie.juliaplots.org/stable/documentation/backends_and_output/)

Neste livro, mostraremos apenas exemplos para CairoMakie.jl e GLMakie.jl.

Você pode ativar qualquer *backend* usando o pacote apropriado e chamando sua função `activate!`. Por exemplo:

```
using GLMakie  
GLMakie.activate!()
```

Agora, vamos começar com *plots* com qualidade de publicação. Mas, antes de sairmos plotando, é importante saber como salvar nossos gráficos. A opção mais fácil para salvar uma figura `fig` é digitar `save("filename.png", fig)`. Outros formatos também estão disponíveis para CairoMakie.jl, como `svg` e `pdf`. A resolução da imagem de saída pode ser facilmente ajustada passando argumentos extras. Por exemplo, para formatos vetoriais, você especifica `pt_per_unit`:

```
save("filename.pdf", fig; pt_per_unit=2)
```

ou

```
save("filename.pdf", fig; pt_per_unit=0.5)
```

Para `png`, você especifica `px_per_unit`. Veja Backends & Output<sup>3</sup> para mais detalhes.

<sup>3</sup> [https://makie.juliaplots.org/stable/documentation/backends\\_and\\_output/](https://makie.juliaplots.org/stable/documentation/backends_and_output/)

## 5.1 CairoMakie.jl

Vamos começar com nosso primeiro *plot*, um gráfico de dispersão com algumas observações conectadas por linhas:

```
using CairoMakie
CairoMakie.activate!()
```

```
fig = scatterlines(1:10, 1:10)
```

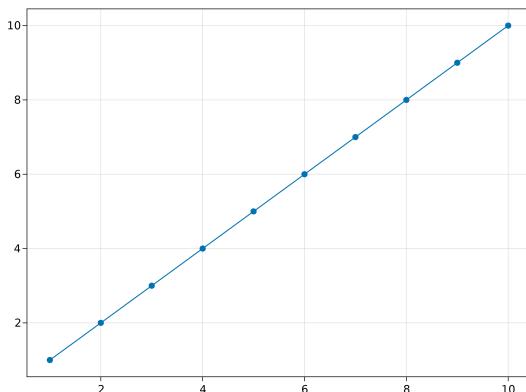


Figure 5.1: First plot.

Observe que o gráfico anterior é a saída padrão, que provavelmente precisaremos ajustar usando nomes de eixo e rótulos.

Observe também que toda função de plotagem como `scatterlines` cria e retorna novos objetos do tipo `Figure`, `Axis` e `plot` dentro de uma coleção chamada `FigureAxisPlot`. Estes são conhecidos como os métodos *non-mutating* (imutáveis). Por outro lado, os métodos *mutating* (mutáveis, por exemplo, `scatterlines!`, observe o `!`) apenas retornam um objeto do tipo `plot` que pode ser anexado a um determinado `axis` (eixo) ou à `current_figure()` (figura atual).

A próxima pergunta que se pode ter é: como mudo a cor ou o tipo de marcador? Isso pode ser feito por meio de `attributes` (atributos), o que faremos na próxima seção.

## 5.2 Atributos

Um gráfico personalizado pode ser criado usando `attributes` (atributos). Os atributos podem ser definidos através de argumentos de palavras-chave. Uma lista de atributos para cada objeto de plotagem pode ser visualizada via:

```
fig, ax, pltobj = scatterlines(1:10)
pltobj.attributes
```

---

```
Attributes with 15 entries:
color => RGBA{Float32}(0.0,0.447059,0.698039,1.0)
colormap => viridis
colorrange => Automatic()
cycle => [:color]
inspectable => true
linestyle => nothing
linewidth => 1.5
marker => Circle
markercolor => Automatic()
markercolormap => viridis
markercolorrange => Automatic()
markersize => 9
model => Float32[1.0 0.0 0.0 0.0; 0.0 1.0 0.0 0.0; 0.0 0.0 1.0 0.0; 0.0 0.0 0.
    ↪ 0 1.0]
strokecolor => black
strokewidth => 0
```

---

Ou como uma chamada de dicionário ([Dict](#)), `pltobject.attributes.attributes`.

Pedir ajuda no REPL como `?lines` ou `help(lines)` para qualquer função de plotagem mostrará seus atributos correspondentes acrescidos de uma breve descrição de como usar essa função específica. Por exemplo, para `lines`:

```
help(lines)
```

---

```
lines(positions)
lines(x, y)
lines(x, y, z)
```

```
Creates a connected line plot for each element in (x, y, z), (x, y) or
positions.
```

| Tip  
|  
| You can separate segments by inserting NaNs.

lines has the following function signatures:

```
(Vector, Vector)
(Vector, Vector, Vector)
(Matrix)
```

Available attributes for Lines are:

```
color
colormap
colorrange
cycle
depth_shift
diffuse
inspectable
linestyle
linewidth
nan_color
overdraw
shininess
specular
ssao
transparency
visible
```

---

Não apenas os objetos de tipo *plot* têm atributos, como também os objetos *Axis* (eixo) e *Figure* (figura) os possuem. Por exemplo, para figura, temos *backgroundcolor* (cor de fundo), *resolution* (resolução), *font* (fonte) e *fontsize* (tamanho da fonte) e o *figure\_padding* (preenchimento ou passe-partout) que altera a quantidade de espaço ao redor do conteúdo da figura, veja a área cinza no *plot*, *Figure* (@ fig:custom\_plot). Ele aceita como argumentos um número único para todos os lados, ou uma tupla de quatro números para esquerda, direita, inferior e superior, representando cada um dos lados.

*Axis* tem muito mais atributos, alguns deles são *backgroundcolor* (cor de fundo), *xgridcolor* (cor da grade do eixo x) e *title* (título). Para uma lista completa basta digitar `help(Axis)`.

Assim, para nosso próximo plot, designaremos vários atributos de uma só vez, como segue:

```
lines(1:10, (1:10).^2; color=:black, linewidth=2, linestyle=:dash,
      figure(; figure_padding=5, resolution=(600, 400), font="sans",
              backgroundcolor=:grey90, fontsize=16),
```

```
axis(; xlabel="x", ylabel="x^2", title="title",
      xgridstyle=:dash, ygridstyle=:dash))
current_figure()
```

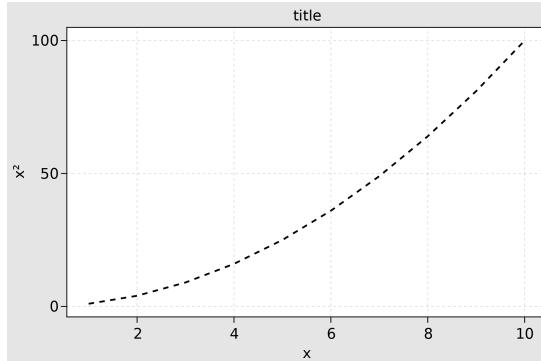


Figure 5.2: Custom plot.

Este exemplo já possui a maioria dos atributos que grande parte dos usuários normalmente executará. Provavelmente, também seria bom ter uma `legend` (legenda). O que fará mais sentido quando utilizarmos mais de uma função de visualização. Então, vamos `append` (acrescentar) outra mutação em nosso `plot` object e adicionar as legendas correspondentes chamando `axislegend`. A legenda criada irá coletar todos os `label`s que você pode ter passado para suas funções de plotagem e por padrão estará localizada na posição superior direita. Para uma posição diferente, o argumento `position=:ct` é chamado, onde `:ct` significa que vamos colocar nosso rótulo no ‘centro’ e no ‘topo,’ veja Figura Figure 5.3:

```
lines(1:10, (1:10).^2; label="x^2", linewidth=2, linestyle=nothing,
      figure(; figure_padding=5, resolution=(600, 400), font="sans",
              backgroundcolor=:grey90, fontsize=16),
      axis(; xlabel="x", title="title", xgridstyle=:dash,
            ygridstyle=:dash))
scatterlines!(1:10, (10:-1:1).^2; label="Reverse(x)^2")
axislegend("legend"; position=:ct)
current_figure()
```

Outras posições também estão disponíveis ao combinarmos `left(l)`, `center(c)`, ↵ `right(r)` com `bottom(b)`, `center(c)`, `top(t)`. Por exemplo, para o topo superior esquerdo, use `:lt`.

No entanto, escrever essa quantidade de código apenas para duas linhas é complicado. Portanto, se você planeja fazer muitos *plots* com a mesma estética geral, definir um tema é sempre melhor. Podemos fazer isso com `set_theme!()` como ilustrado pelo exemplo abaixo.

A plotagem da figura anterior deve ter as novas configurações padrão definidas

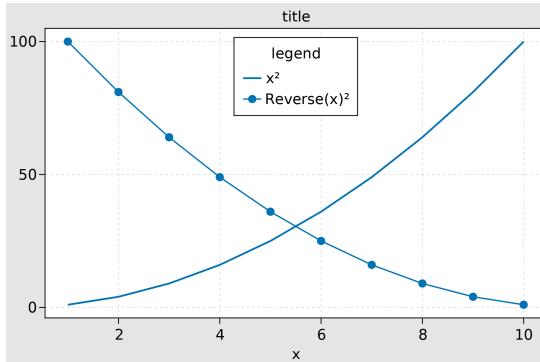


Figure 5.3: Custom plot legend.

```
por set_theme!(kwargs):
```

```
set_theme!(; resolution=(600, 400),
    backgroundcolor=(:orange, 0.5), fontsize=16, font="sans",
    Axis=(backgroundcolor=:grey90, xgridstyle=:dash, ygridstyle=:dash),
    Legend=(bgcolor=(:red, 0.2), framecolor=:dodgerblue))
lines(1:10, (1:10).^2; label="x2", linewidth=2, linestyle=nothing,
    axis=(; xlabel="x", title="title"))
scatterlines!(1:10, (10:-1:1).^2; label="Reverse(x)2")
axislegend("legend"; position=:ct)
current_figure()
set_theme!()
caption = "Set theme example."
```

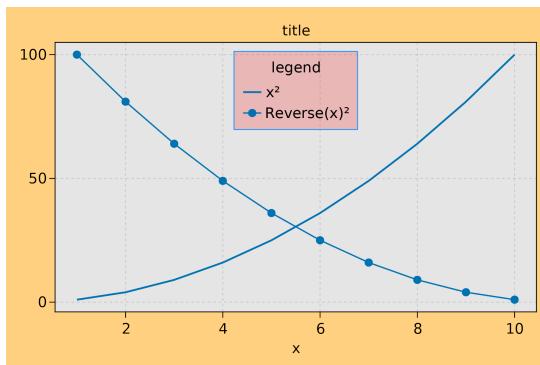


Figure 5.4: Set theme example.

Perceba que a última linha é `set_theme!()`, que irá redefinir as configurações padrão do Makie. Para mais themes por favor vá a Section 5.3.

Antes de passarmos para a próxima seção, vale a pena ver um exemplo onde um array de atributos é passado de uma só vez para uma função de plotagem. Para esse exemplo, usaremos a função de plotagem `scatter` para fazer um gráfico de dispersão.

Os dados para isso podem ser um array com 100 linhas e 3 colunas, aqui ger-

ados aleatoriamente a partir de uma distribuição normal. Aqui, a primeira coluna pode ser as posições no eixo  $x$ , a segunda as posições em  $y$  e a terceira um valor associado intrínseco para cada ponto. O último pode ser representado em um gráfico por uma ‘cor’ diferente ou com um tamanho de marcador diferente. Em um gráfico de dispersão podemos fazer os dois.

```
using Random; seed!
seed!(28)
xyvals = randn(100, 3)
xyvals[1:5, :]
```

```
5×3 Matrix{Float64}:
 0.550992  1.27614  -0.659886
 -1.06587  -0.0287242  0.175126
 -0.721591  -1.84423  0.121052
 0.801169  0.862781  -0.221599
 -0.340826  0.0589894  -1.76359
```

A seguir, o `plot` correspondente pode ser visto em Figure 5.5:

```
fig, ax, pltobj = scatter(xyvals[:, 1], xyvals[:, 2]; color=xyvals[:, 3],
    label="Bubbles", colormap=:plasma, markersize=15 * abs.(xyvals[:, 3]),
    figure=(; resolution=(600, 400)), axis=(; aspect=DataAspect()))
limits!(-3, 3, -3, 3)
Legend(fig[1, 2], ax, valign=:top)
Colorbar(fig[1, 2], pltobj, height=Relative(3 / 4))
fig
caption = "Bubble plot."
```

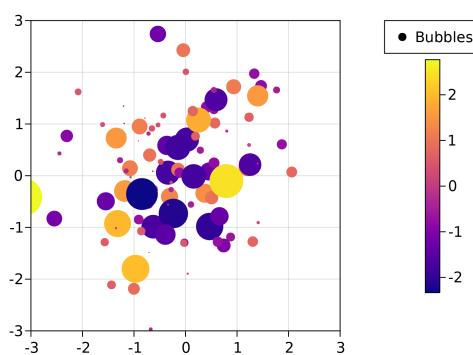


Figure 5.5: Bubble plot.

onde decomponemos a tupla `FigureAxisPlot` em `fig`, `ax`, `pltobj`, para podermos adicionar um `Legend` e `Colorbar` fora do objeto plotado. Vamos discutir opções de `layout` mais detalhadamente em in Section 5.6.

Fizemos alguns exemplos básicos, mas ainda interessantes, para mostrar como usar o `Makie.jl` e agora você deve estar se perguntando: o que mais podemos

fazer? Quais são todas as possíveis funções de plotagem disponíveis em `Makie ↵.jl`? Para responder essa pergunta, contamos com uma *cheat sheet* em Figure 5.6. Isso funciona especialmente bem com o *backend* `CairoMakie.jl`.

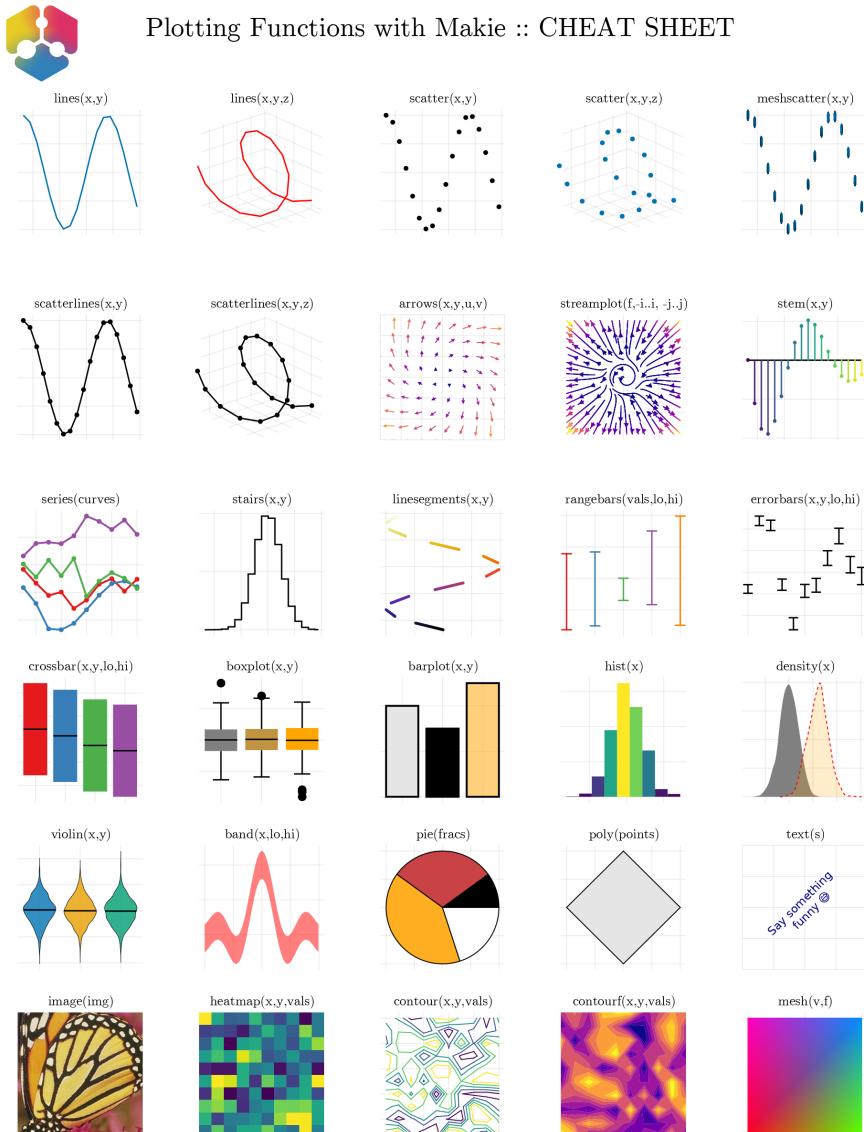


Figure 5.6: Funções de plotagem: Cheat Sheet. Saída dada por `Cairomakie`.

Para completar, em Figure 5.7, mostramos as funções correspondentes *cheat sheet* para `GLMakie.jl`, que dão suporte principalmente para plotagens 3D. Elas serão explicadas em detalhes em Section 5.7.

Agora que temos uma ideia de todas as coisas que podemos fazer, vamos voltar e continuar com o básico. É hora de aprendermos a mudar a aparência geral

Learn more in Julia Data Science, <https://juliadatascience.io> · <http://makie.juliaplots.org> · Makie v0.15.0 · CairoMakie v0.6.3 · Updated: 2021-08-03

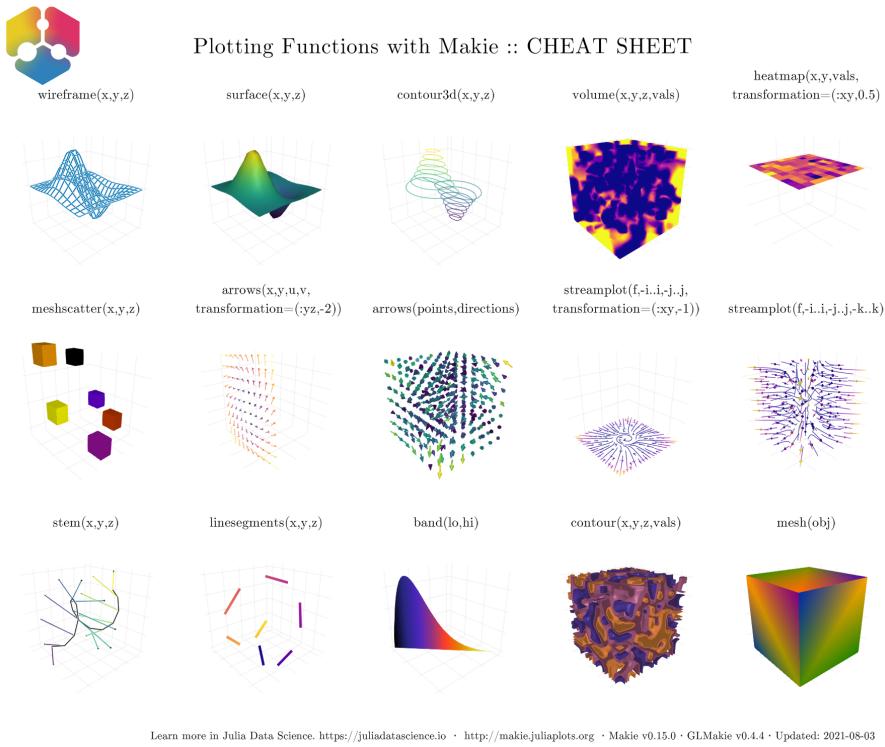


Figure 5.7: Funções de plotagem: Cheat Sheet. Saída dada por GLMakie.

dos nossos *plots*.

### 5.3 Temas

Existem várias maneiras de modificar a aparência geral de seus *plots*. Você pode usar um tema predefinido<sup>4</sup> ou seu próprio tema personalizado. Por exemplo, use um tema escuro predefinido via `with_theme(your_plot_function, ↪theme_dark())`. Ou construa o seu próprio com `Theme(kwargs)` ou até mesmo atualize o que está ativo com `update_theme!(kwargs)`.

<sup>4</sup> [http://makie.juliaplots.org/stable/documentation/theming/predefined\\_themes/index.html](http://makie.juliaplots.org/stable/documentation/theming/predefined_themes/index.html)

Você também pode fazer `set_theme!(theme; kwargs...)` para alterar o tema do padrão atual para `theme` e substituir ou adicionar atributos fornecidos por `kwargs ↪`. Se você fizer isso e quiser redefinir todas as configurações anteriores, faça `set_theme!()` sem argumentos. Veja os exemplos a seguir, onde preparamos uma função de plotagem de teste com características diferentes, de forma que a maioria dos atributos para cada tema possa ser apreciada.

```
using Random: seed!
seed!(123)
y = cumsum(randn(6, 6), dims=2)
```

---

```
6x6 Matrix{Float64}:
 0.808288  0.386519  0.355371  0.0365011 -0.0911358  1.8115
 -1.12207 -2.47766 -2.16183 -2.49928 -2.02981 -1.37017
 -1.10464 -1.03518 -3.19756 -1.18944 -2.71633 -3.80455
 -0.416993 -0.534315 -1.42439 -0.659362 -0.0592298  0.644529
  0.287588  1.50687  2.36111  2.54137  0.48751  0.630836
  0.229819  0.522733  0.864515  2.89343  2.06537  2.21375
```

---

Uma matrix de tamanho (20, 20) com entradas aleatórias, para que possamos plotar um mapa de calor. O intervalo em  $x$  e  $y$  também é especificado.

```
using Random: seed!
seed!(13)
xv = yv = LinRange(-3, 0.5, 20)
matrix = randn(20, 20)
matrix[1:6, 1:6] # first 6 rows and columns
```

---

```
6x6 Matrix{Float64}:
 -0.271257  0.894952  0.728865  -0.293849 -0.449277 -0.0948871
 -0.193033 -0.421286 -0.455905 -0.0576092 -0.756621 -1.47419
 -0.123177  0.762254  0.773921 -0.38526 -0.0659695 -0.599284
 -1.47327  0.770122  1.20725  0.257913  0.111979  0.875439
 -1.82913 -0.603888  0.164083 -0.118504  1.46723  0.0948876
  1.09769  0.178207  0.110243 -0.543203  0.592245  0.328993
```

---

Portanto, nossa função de plotagem se parece com o seguinte:

```
function demo_themes(y, xv, yv, matrix)
    fig, _ = series(y; labels=["$i" for i = 1:6], markersize=10,
        color=:Set1, figure=(; resolution=(600, 300)),
        axis=(; xlabel="time (s)", ylabel="Amplitude",
            title="Measurements"))
    hmap = heatmap!(xv, yv, matrix; colormap=:plasma)
    limits!(-3.1, 8.5, -6, 5.1)
    axislegend("legend"; merge=true)
    Colorbar(fig[1, 2], hmap)
    fig
end
```

Observe que a função `series` foi usada para plotar várias linhas e dispersões de uma só vez com seus rótulos correspondentes. Além disso, um mapa de calor com sua barra de cores foi incluído. Atualmente, existem dois temas escuros, um chamado `theme_dark()` e outro `theme_black()`:

```
with_theme(theme_dark()) do
    demo_themes(y, xv, yv, matrix)
```

```

end
with_theme(theme_black()) do
    demo_themes(y, xv, yv, matrix)
end

```

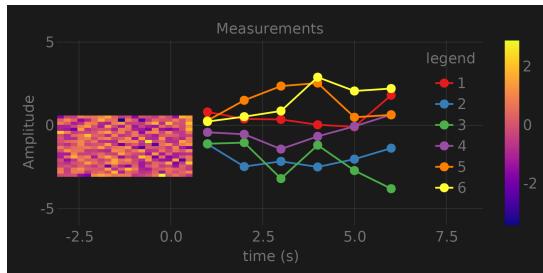


Figure 5.8: Theme dark.

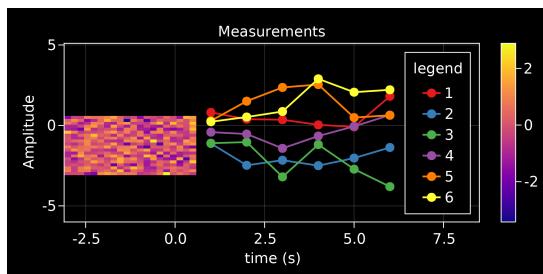


Figure 5.9: Theme black.

E mais três temas claros chamados, `theme_ggplot2()`, `theme_minimal()` e `theme_light()`. Útil para *plots* de tipo de publicação mais padrão.

```

with_theme(theme_ggplot2()) do
    demo_themes(y, xv, yv, matrix)
end
with_theme(theme_minimal()) do
    demo_themes(y, xv, yv, matrix)
end
with_theme(theme_light()) do
    demo_themes(y, xv, yv, matrix)
end

```

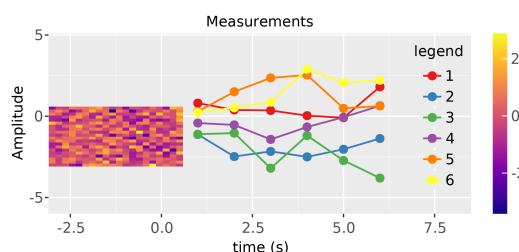


Figure 5.10: Theme ggplot2.

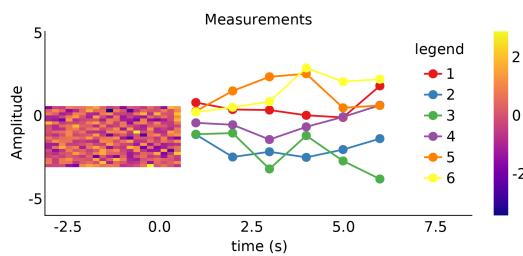


Figure 5.11: Theme minimal.

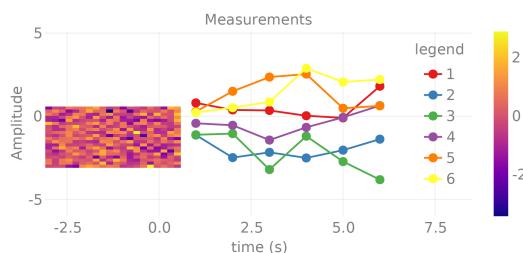


Figure 5.12: Theme light.

Outra alternativa é definir um `Theme` fazendo `with_theme(your_plot, your_theme())`. Por exemplo, o tema a seguir pode ser uma versão simples para um modelo de qualidade de publicação:

```
publication_theme() = Theme(
    fontsize=16, font="CMU Serif",
    Axis=(xlabelsize=20, xgridstyle=:dash, ygridstyle=:dash,
          xtickalign=1, ytickalign=1, yticksize=10, xticksize=10,
          xlabelpadding=-5, xlabel="x", ylabel="y"),
    Legend=(framecolor=(:black, 0.5), bgcolor=(:white, 0.5)),
    Colorbar=(ticksize=16, tickalign=1, spinewidth=0.5),
)
```

Que, por simplicidade, usamos para plotar `scatterlines` e um `heatmap`.

```
function plot_with_legend_and_colorbar()
    fig, ax, _ = scatterlines(1:10; label="line")
    hm = heatmap!(ax, LinRange(6, 9, 15), LinRange(2, 5, 15), randn(15, 15);
                 colormap=:Spectral_11)
    axislegend("legend"; position=lt)
    Colorbar(fig[1, 2], hm, label="values")
    ax.title = "my custom theme"
    fig
end
```

Então, usando o `Theme` definido anteriormente, a saída é mostrada na Figura (Figure 5.13).

```
with_theme(plot_with_legend_and_colorbar, publication_theme())
```

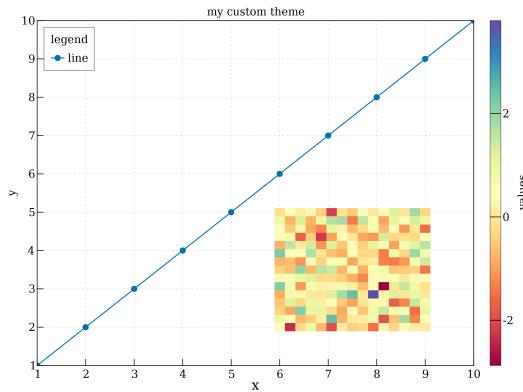


Figure 5.13: Themed plot with Legend and Colorbar.

Agora, se algo precisar ser alterado após `set_theme!(your_theme)`, podemos fazer isso com `update_theme!(resolution=(500, 400), fontsize=18)`, por exemplo. Outra abordagem será passar argumentos adicionais para a função `with_theme`:

```
fig = (resolution=(600, 400), figure_padding=1, backgroundcolor=:grey90)
ax = (; aspect=DataAspect(), xlabel=L"x", ylabel=L"y")
cbar = (; height=Relative(4 / 5))
with_theme(publication_theme(); fig..., Axis=ax, Colorbar=cbar) do
    plot_with_legend_and_colorbar()
end
```

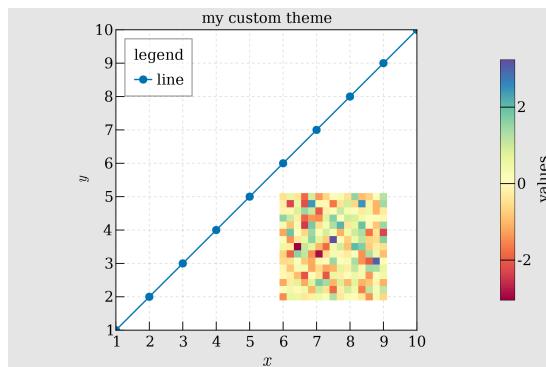


Figure 5.14: Theme with extra args.

Agora, vamos seguir em frente e fazer um *plot* com strings em LaTeX e um tema personalizado.

## 5.4 Usando LaTeXStrings.jl

Suporte LaTeX em Makie.jl também está disponível via `LaTeXStrings.jl`:

**using** LaTeXStrings

Casos de uso simples mostraremos abaixo (Figure 5.15). Um exemplo básico inclui strings LaTeX para rótulos e legendas x-y:

```
function LaTeX_Strings()
    x = 0:0.05:4π
    lines!(x, x -> sin(3x) / (cos(x) + 2) / x; label=L"\frac{\sin(3x)}{x(\cos(x)+2)}",
    figure=(; resolution=(600, 400)), axis=(; xlabel=L"x")
    lines!(x, x -> cos(x) / x; label=L"\cos(x)/x")
    lines!(x, x -> exp(-x); label=L"e^{-x}")
    limits!(-0.5, 13, -0.6, 1.05)
    axislegend(L"f(x)")
    current_figure()
end
```

```
with_theme(LaTeX_Strings, publication_theme())
```

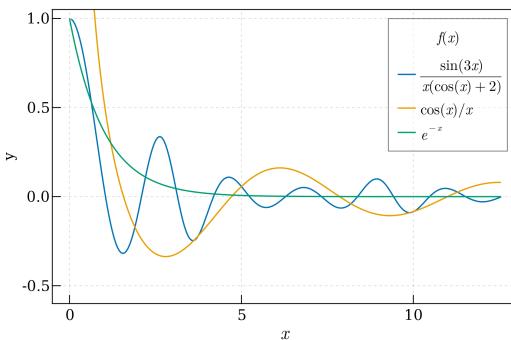


Figure 5.15: Plot with LaTeX strings.

Um exemplo mais complicado será com alguma equação como texto e aumentando a numeração de legenda para curvas em um *plot*:

```
function multiple_lines()
    x = collect(0:10)
    fig = Figure(resolution=(600, 400), font="CMU Serif")
    ax = Axis(fig[1, 1], xlabel=L"x", ylabel=L"f(x,a)")
    for i = 0:10
        lines!(ax, x, i .* x; label=latexstring("$(i) x"))
    end
    axislegend(L"f(x)"; position=:lt, nbanks=2, labelsize=14)
    text!(L"f(x,a) = ax", position=(4, 80))
    fig
end
multiple_lines()
```

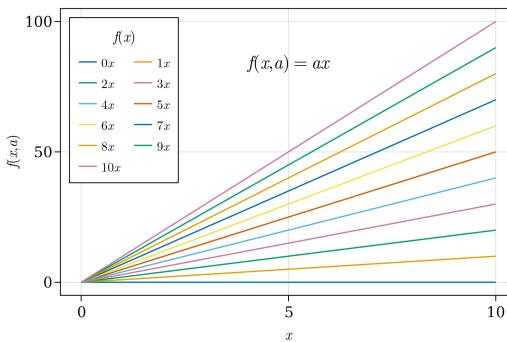


Figure 5.16: Multiple lines.

Mas, algumas linhas têm cores repetidas, então isso não é bom. Adicionar alguns marcadores e estilos de linha geralmente ajuda. Então, vamos fazer isso usando `Cycles`<sup>5</sup> para esses tipos. Definir `covary=true` permite alternar todos os elementos juntos:

```
function multiple_scatters_and_lines()
    x = collect(0:10)
    cycle = Cycle([:color, :linestyle, :marker], covary=true)
    set_theme!(Lines=(cycle=cycle,), Scatter=(cycle=cycle,))
    fig = Figure(resolution=(600, 400), font="CMU Serif")
    ax = Axis(fig[1, 1], xlabel=L"x", ylabel=L"f(x,a)")
    for i in x
        lines!(ax, x, i .* x; label=latexstring("$(i) x"))
        scatter!(ax, x, i .* x; markersize=13, strokewidth=0.25,
            label=latexstring("$(i) x"))
    end
    axislegend(L"f(x)"; merge=true, position=:lt, nbanks=2, labelsize=14)
    text!(L"f(x,a) = ax", position=(4, 80))
    set_theme!() # reset to default theme
    fig
end
multiple_scatters_and_lines()
```

<sup>5</sup> <http://makie.juliaplot.org/stable/documentation/theming/index.html#cycles>

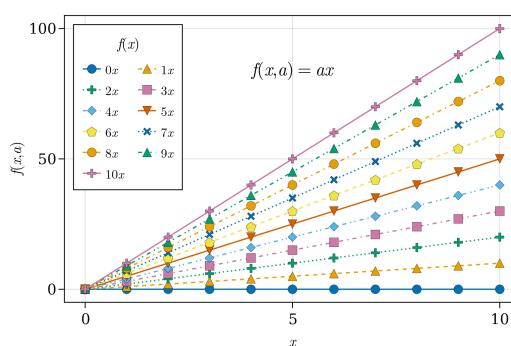


Figure 5.17: Multiple Scatters and Lines.

E voilà. Um *plot* de qualidade de publicação está aqui. O que mais podemos pedir? Bem, e quanto a diferentes cores ou paletas padrão? Em nossa próxima seção, veremos como usar novamente `Cycles`<sup>6</sup> e conheça um pouco mais sobre eles, além de algumas palavras-chave adicionais para conseguir isso.

<sup>6</sup> <http://makie.juliaplot.org/stable/documentation/theming/index.html#cycles>

## 5.5 Cores e mapas de cores

Escolher um conjunto apropriado de cores ou barra de cores para sua plotagem é uma parte essencial para apresentação de resultados. `Colors.jl`<sup>7</sup> é suportado em `Makie.jl` para que você possa usar cores nomeadas<sup>8</sup> ou passar valores RGB → ou RGBA. Além disso, os mapas de cores `ColorSchemes.jl`<sup>9</sup> e `PerceptualColourMaps.jl`<sup>10</sup> também podem ser usados. Vale a pena saber que você pode reverter um mapa de cores fazendo `Reverse(:colormap_name)` para obter uma cor transparente ou mapa de cores com `color=(:red, 0.5)` e `colormap=(:viridis, 0.5)`.

Diferentes casos de uso serão mostrados a seguir. Então vamos definir um tema personalizado com novas cores e uma paleta de cores.

<sup>7</sup> <https://github.com/JuliaGraphics/Colors.jl>

<sup>8</sup> <https://juliographics.github.io/Colors.jl/latest/namedcolors/>

<sup>9</sup> <https://github.com/JuliaGraphics/ColorSchemes.jl>

<sup>10</sup> [https://github.com/peterekovesi/PerceptualColourMaps.jl](https://github.com/peterkovesi/PerceptualColourMaps.jl)

Por padrão `Makie.jl` tem um conjunto predefinido de cores para percorrê-las automaticamente. Conforme mostrado nas figuras anteriores, onde nenhuma cor específica foi definida. A substituição desses padrões é feita chamando a palavra-chave `color` na função de plotagem e especificando uma nova cor por meio de um `Symbol` ou `String`. Veja isso em ação no exemplo a seguir:

```
function set_colors_and_cycle()
    # Epicycloid lines
    x(r, k, θ) = r * (k .+ 1.0) .* cos.(θ) .- r * cos.((k .+ 1.0) .* θ)
    y(r, k, θ) = r * (k .+ 1.0) .* sin.(θ) .- r * sin.((k .+ 1.0) .* θ)
    θ = LinRange(0, 6.2π, 1000)
    axis = (; xlabel=L"x(\theta)", ylabel=L"y(\theta)",
              title="Epicycloid", aspect=DataAspect())
    figure = (; resolution=(600, 400), font="CMU Serif")
    fig, ax, _ = lines(x(1, 1, 0), y(1, 1, 0); color="firebrick1", # string
                       label=L"1.0", axis=axis, figure=figure)
    lines!(ax, x(4, 2, 0), y(4, 2, 0); color=:royalblue1, #symbol
          label=L"2.0")
    for k = 2.5:0.5:5.5
        lines!(ax, x(2k, k, 0), y(2k, k, 0); label=latexstring("$(k)")) #cycle
    end
    Legend(fig[1, 2], ax, latexstring("k, r = 2k"), merge=true)
    fig
end
set_colors_and_cycle()
```

Onde, nas duas primeiras linhas, usamos a palavra-chave `color` para especificar nossa cor. O resto está usando o padrão do conjunto de cores do ciclo. Mais

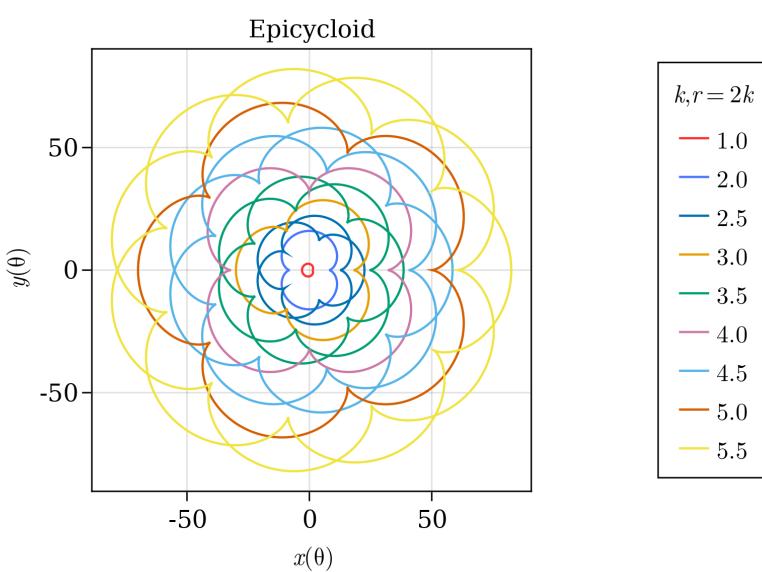


Figure 5.18: Set colors and cycle.

tarde, aprenderemos como fazer um ciclo personalizado.

Em relação aos mapas de cores, já estamos familiarizados com a palavra-chave `colormap` para `heatmaps` e `scatters`. Aqui, mostramos que um mapa de cores também pode ser especificado por meio de um `Symbol` ou uma `String`, semelhante a cores. Ou até mesmo um vetor de cores `RGB`. Vamos fazer nosso primeiro exemplo chamando mapas de cores como `Symbol`, `String` e `cgrad` para valores categóricos. Cheque `?cgrad` para mais informações.

```
figure = (; resolution=(600, 400), font="CMU Serif")
axis = (; xlabel=L"x", ylabel=L"y", aspect=DataAspect())
fig, ax, pltobj = heatmap(rand(20, 20); colorrange=(0, 1),
    colormap=Reverse(:viridis), axis=axis, figure=figure)
Colorbar(fig[1, 2], pltobj, label = "Reverse colormap Sequential")
fig
```

Ao definir um `colorrange` geralmente os valores fora deste intervalo são coloridos com a primeira e a última cor do mapa de cores. No entanto, às vezes é melhor especificar a cor desejada em ambas as extremidades. Fazemos isso com `highclip` e `lowclip`:

```
using ColorSchemes
```

```
figure = (; resolution=(600, 400), font="CMU Serif")
axis = (; xlabel=L"x", ylabel=L"y", aspect=DataAspect())
fig, ax, pltobj=heatmap(randn(20, 20); colorrange=(-2, 2),
```

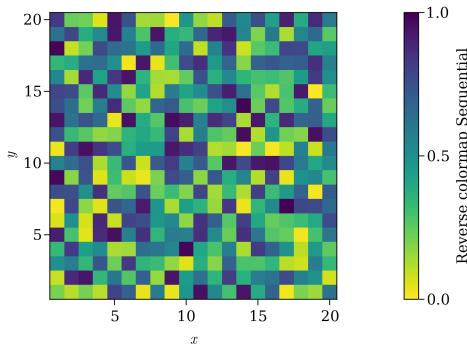


Figure 5.19: Reverse colormap sequential and colorange.

```
colormap="diverging_rainbow_bgymr_45_85_c67_n256",
highclip=:black, lowclip=:white, axis=axis, figure=figure)
Colorbar(fig[1, 2], pltobj, label = "Diverging colormap")
fig
```

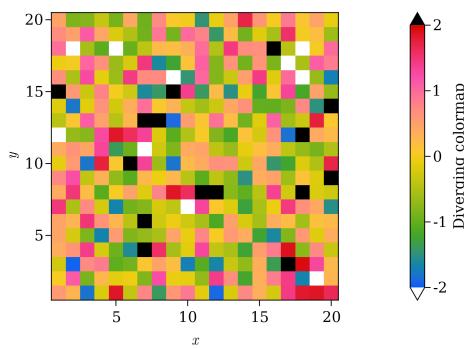


Figure 5.20: Diverging Colormap with low and high clip.

Mas mencionamos que também vetores RGB são opções válidas. Para o nosso próximo exemplo, você pode passar o mapa de cores personalizado *perse* ou usar `cgrad` para forçar um Colorbar categórico.

```
using Colors, ColorSchemes
```

```
figure = (; resolution=(600, 400), font="CMU Serif")
axis = (; xlabel=L"x", ylabel=L"y", aspect=DataAspect())
cmap = ColorScheme(range(colorant"red", colorant"green", length=3))
mygrays = ColorScheme([RGB{Float64}(i, i, i) for i in [0.0, 0.5, 1.0]])
fig, ax, pltobj = heatmap(rand(-1:1, 20, 20);
    colormap=cgrad(mygrays, 3, categorical=true, rev=true), # cgrad and Symbol,
    ↪mygrays,
    axis=axis, figure=figure)
cbar = Colorbar(fig[1, 2], pltobj, label="Categories")
cbar.ticks = ([-0.66, 0, 0.66], ["-1", "0", "1"])
fig
```

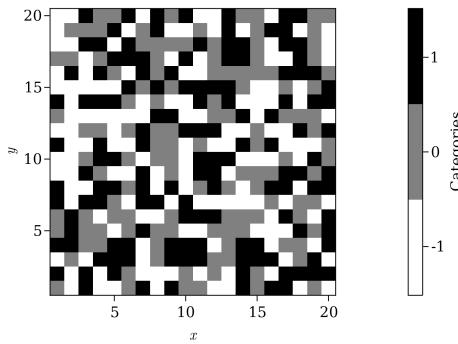


Figure 5.21: Categorical Colormap.

Por fim, os tiques na barra de cores para o caso categorial não são centralizados por padrão em cada cor. Isso é corrigido passando ticks personalizados, como em `cbar.ticks = (positions, ticks)`. A última situação é passar uma tupla de duas cores para o `colormap` como símbolos, strings ou uma mistura. Você obterá um mapa de cores interpolado entre essas duas cores.

Além disso, cores codificadas em hexadecimal também são aceitas. Então, no topo do nosso mapa de calor, vamos colocar um ponto semitransparente usando:

```
figure = (; resolution=(600, 400), font="CMU Serif")
axis = (; xlabel=L"x", ylabel=L"y", aspect=DataAspect())
fig, ax, pltobj = heatmap(rand(20, 20); colorrange=(0, 1),
    colormap(:red, "black"), axis=axis, figure=figure)
scatter!(ax, [11], [11], color="#C0C0C0", 0.5), markersize=150)
Colorbar(fig[1, 2], pltobj, label="2 colors")
fig
```

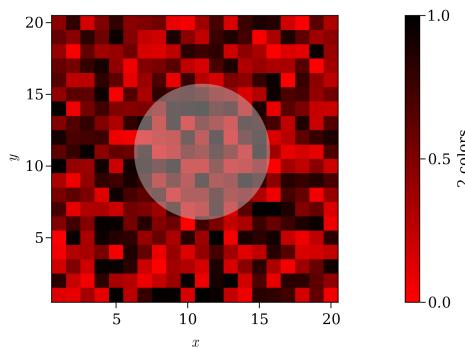


Figure 5.22: Colormap from two colors.

### 5.5.1 Ciclo personalizado

Aqui, poderíamos definir um `Tema` global com um novo ciclo de cores, mas essa **não é a maneira recomendada** de fazer isso. É melhor definir um novo tema e

usar como mostramos antes. Vamos definir um novo com um `cycle` para `:color ↪, :linestyle, :marker` e um novo padrão `colormap`. Vamos adicionar esses novos atributos ao nosso `publication_theme` anterior.

```
function new_cycle_theme()
    # https://nanx.me/ggsci/reference/pal_locuszoom.html
    my_colors = ["#D43F3AFF", "#EEA236FF", "#5CB85cff", "#46B8DAFF",
                 "#357EBdff", "#9632B8FF", "#B8B8B8FF"]
    cycle = Cycle([:color, :linestyle, :marker], covary=true) # alltogether
    my_markers = [:circle, :rect, :utriangle, :dtriangle, :diamond,
                  :pentagon, :cross, :xcross]
    my_linestyle = [nothing, :dash, :dot, :dashdot, :dashdotdot]
    Theme(
        fontsize=16, font="CMU Serif",
        colormap=:linear_bmy_10_95_c78_n256,
        palette=(color=my_colors, marker=my_markers, linestyle=my_linestyle),
        Lines=(cycle=cycle,), Scatter=(cycle=cycle,),
        Axis=( xlabelsize=20, xgridstyle=:dash, ygridstyle=:dash,
               xtickalign=1, ytickalign=1, yticksize=10, xticksize=10,
               xlabelpadding=-5, xlabel="x", ylabel="y"),
        Legend=(framecolor(:black, 0.5), bgcolor(:white, 0.5)),
        Colorbar=( ticksize=16, tickalign=1, spinewidth=0.5),
    )
end
```

E aplique-o a uma função de plotagem como a seguinte:

```
function scatters_and_lines()
    x = collect(0:10)
    xh = LinRange(4, 6, 25)
    yh = LinRange(70, 95, 25)
    h = randn(25, 25)
    fig = Figure(resolution=(600, 400), font="CMU Serif")
    ax = Axis(fig[1, 1], xlabel=L"x", ylabel=L"f(x,a)")
    for i in x
        lines!(ax, x, i .* x; label=latexstring("$(i) x"))
        scatter!(ax, x, i .* x; markersize=13, strokewidth=0.25,
                 label=latexstring("$(i) x"))
    end
    hm = heatmap!(xh, yh, h)
    axislegend(L"f(x)"; merge=true, position=:lt, nbanks=2, labelszie=14)
    Colorbar(fig[1, 2], hm, label="new default colormap")
    limits!(ax, -0.5, 10.5, -5, 105)
    colgap!(fig.layout, 5)
    fig
end
```

```
with_theme(scatters_and_lines, new_cycle_theme())
```

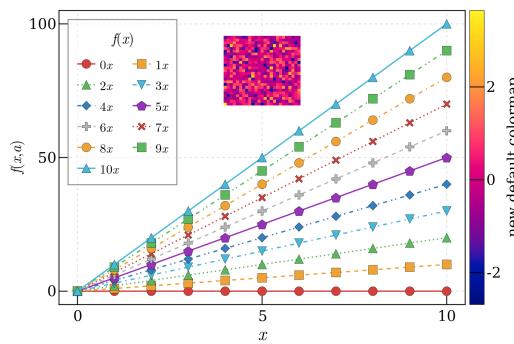


Figure 5.23: Custom theme with new cycle and colormap.

Neste ponto, você deve ter **controle completo** sobre suas cores, estilos de linha, marcadores e mapas de cores para seus *plots*. A seguir, veremos como gerenciar e controlar *layouts*.

## 5.6 Layouts

Um *canvas/layout* completo é definido por `Figure`, que pode ser preenchido com conteúdo após ser criado. Começaremos com um arranjo simples de um `Axis`, uma `Legend` e uma `Colorbar`. Para esta tarefa, podemos pensar no canvas como um arranjo de `rows` e `columns` na indexação de uma `Figure` bem como um `Array ↪/Matrix` regular. O conteúdo do `Axis` estará na *linha 1, coluna 1*, por exemplo `fig[1, 1]`, a `Colorbar` na *linha 1, coluna 2*, ou seja, `fig[1, 2]`. E a `Legend` na *linha 2* e nas *colunas 1 e 2*, ou seja, `fig[2, 1:2]`.

```
function first_layout()
    seed!(123)
    x, y, z = randn(6), randn(6), randn(6)
    fig = Figure(resolution=(600, 400), backgroundcolor=:grey90)
    ax = Axis(fig[1, 1], backgroundcolor=:white)
    pltobj = scatter!(ax, x, y; color=z, label="scatters")
    lines!(ax, x, 1.1y; label="line")
    Legend(fig[2, 1:2], ax, "labels", orientation=:horizontal)
    Colorbar(fig[1, 2], pltobj, label="colorbar")
    fig
end
first_layout()
```

Isso já parece bom, mas poderia ser melhor. Podemos corrigir problemas de espaçamento usando as seguintes palavras-chave e métodos:

- `figure_padding=(left, right, bottom, top)`
- `padding=(left, right, bottom, top)`

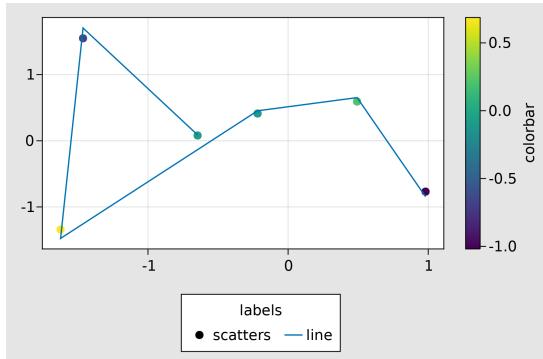


Figure 5.24: First Layout.

Levar em consideração o tamanho real de uma Legend ou Colorbar é feito por:

- tellheight=true ou false
- tellwidth=true ou false

Definir como true levará em consideração o tamanho real (altura ou largura) para uma Legend ou Colorbar. Consequentemente, as coisas serão redimensionadas de acordo.

O espaço entre colunas e linhas é especificado como:

- colgap!(fig.layout, col, separation)
- rowgap!(fig.layout, row, separation)

*Column gap* (colgap!), se col for fornecido, a lacuna será aplicada a essa coluna específica. *Row gap* (rowgap!) , se a linha for fornecido, a lacuna será aplicada a essa linha específica.

Além disso, veremos como colocar conteúdo nas **protrusões**, i.e. o espaço reservado para título: x e y; ou ticks ou label. Fazemos isso plotando em fig →[i, j, protrusion] onde protrusion pode ser Left(), Right(), Bottom() e Top(), ou para cada canto TopLeft(), TopRight(), BottomRight(), BottomLeft(). Veja abaixo como essas opções estão sendo utilizadas:

```
function first_layout_fixed()
    seed!(123)
    x, y, z = randn(6), randn(6), randn(6)
    fig = Figure(figure_padding=(0, 3, 5, 2), resolution=(600, 400),
        backgroundcolor=:grey90, font="CMU Serif")
    ax = Axis(fig[1, 1], xlabel=L"x", ylabel=L"y",
        title="Layout example", backgroundcolor=:white)
    pltobj = scatter!(ax, x, y; color=z, label="scatters")
    lines!(ax, x, 1.1y, label="line")
    Legend(fig[2, 1:2], ax, "Labels", orientation=:horizontal,
        tellheight=true, titleposition=:left)
    Colorbar(fig[1, 2], pltobj, label="colorbar")
```

```

# additional aesthetics
Box(fig[1, 1, Right()], color=(:slateblue1, 0.35))
Label(fig[1, 1, Right()], "protrusion", textsize=18,
      rotation=pi / 2, padding=(3, 3, 3, 3))
Label(fig[1, 1, TopLeft()], "(a)", textsize=18, padding=(0, 3, 8, 0))
colgap!(fig.layout, 5)
rowgap!(fig.layout, 5)
fig
end
first_layout_fixed()

```

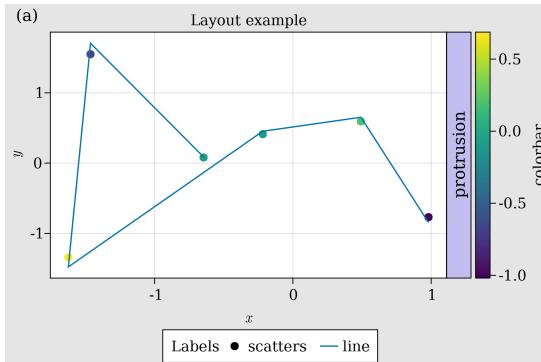


Figure 5.25: First Layout Fixed.

Aqui, ter o rótulo (a) no `TopLeft()` provavelmente não é necessário, isso só fará sentido para mais de dois `plots`. Para o nosso próximo exemplo vamos continuar usando as ferramentas anteriores e mais algumas para criar uma figura mais rica e complexa.

Você pode ocultar decorações e espinhas de eixos com:

- `hidedecorations!(ax; kwargs...)`
- `hidexdecorations!(ax; kwargs...)`
- `hideydecorations!(ax; kwargs...)`
- `hidespines!(ax; kwargs...)`

Lembre-se, sempre podemos pedir ajuda para ver que tipo de argumentos podemos usar, por exemplo,

```
help(hidespines!)
```

---

```
hidespines!(la::Axis, spines::Symbol... = (:l, :r, :b, :t)...)
```

```
Hide all specified axis spines. Hides all spines by default, otherwise
choose with the symbols :l, :r, :b and :t.
```

```
hidespines! has the following function signatures:
```

```
(Vector, Vector)
(Vector, Vector, Vector)
(Matrix)
```

Available attributes for Combined{Makie.MakieLayout.hidespines!} are:

Alternativamente, para decorações:

```
help(hidedecorations!)
```

```
hidedecorations!(la::Axis)
```

Hide decorations of both x and y-axis: label, ticklabels, ticks and grid.

hidedecorations! has the following function signatures:

```
(Vector, Vector)
(Vector, Vector, Vector)
(Matrix)
```

Available attributes for Combined{Makie.MakieLayout.hidedecorations!} are:

Para elementos que você **não deseja ocultar**, apenas passe-os com `false`, ou seja, `hidedecorations!(ax; ticks=false, grid=false)`.

A sincronização do seu Axis é feita via:

- `linkaxes!`, `linkyaxes!` e `linkxaxes!`

Isso pode ser útil quando eixos compartilhados são desejados. Outra maneira de obter eixos compartilhados será definindo `limits!`.

Definir limites de uma vez ou independentemente para cada eixo é feito chamando

- `limits!(ax; l, r, b, t)`, onde `l` é esquerda, `r` direita, `b` inferior e `t` superior.

Você também pode fazer `ylims!(low, high)` ou `xlims!(low, high)`, e até mesmo abrir fazendo `ylims!(low=0)` ou `xlims!(high=1)`.

Agora, o exemplo:

```
function complex_layout_double_axis()
    seed!(123)
    x = LinRange(0, 1, 10)
    y = LinRange(0, 1, 10)
    z = rand(10, 10)
```

```

fig = Figure(resolution=(600, 400), font="CMU Serif", backgroundcolor=:
    ↪grey90)
ax1 = Axis(fig, xlabel=L"x", ylabel=L"y")
ax2 = Axis(fig, xlabel=L"x")
heatmap!(ax1, x, y, z; colormrange=(0, 1))
series!(ax2, abs.(z[1:4, :]); labels=["lab $i" for i = 1:4], color=:Set1_4)
hm = scatter!(10x, y; color=z[1, :], label="dots", colormrange=(0, 1))
hideydecorations!(ax2, ticks=false, grid=false)
linkyaxes!(ax1, ax2)
#layout
fig[1, 1] = ax1
fig[1, 2] = ax2
Label(fig[1, 1, TopLeft()], "(a)", textsize=18, padding=(0, 6, 8, 0))
Label(fig[1, 2, TopLeft()], "(b)", textsize=18, padding=(0, 6, 8, 0))
Colorbar(fig[2, 1:2], hm, label="colorbar", vertical=false, flipaxis=false)
Legend(fig[1, 3], ax2, "Legend")
colgap!(fig.layout, 5)
rowgap!(fig.layout, 5)
fig
end
complex_layout_double_axis()

```

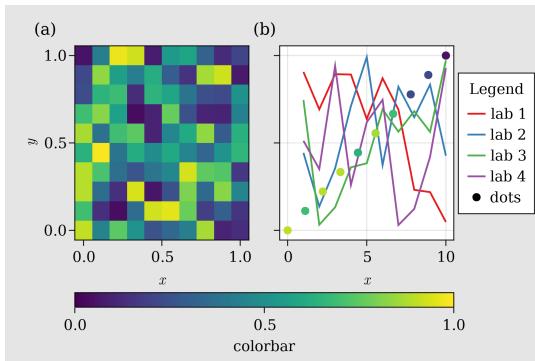


Figure 5.26: Complex layout double axis.

Então, agora nosso Colorbar precisa ser horizontal e as marcações da barra precisam estar na parte inferior. Isso é feito configurando `vertical=false` e `flipaxis ↪=false`. Além disso, observe que podemos chamar muitos `Axis` em `fig`, ou mesmo `Colorbar` e `Legend`, e depois construir o layout.

Outro layout comum é uma grade de quadrados para mapas de calor:

```

function squares_layout()
    seed!(123)
    letters = reshape(collect('a':'d'), (2, 2))
    fig = Figure(resolution=(600, 400), fontsize=14, font="CMU Serif",
        backgroundcolor=:grey90)
    axs = [Axis(fig[i, j], aspect=DataAspect()) for i = 1:2, j = 1:2]
    hms = [heatmap!(axs[i, j], randn(10, 10), colormrange=(-2, 2))
        for i = 1:2, j = 1:2]

```

```

Colorbar(fig[1:2, 3], hms[1], label="colorbar")
[Label(fig[i, j, TopLeft()], "($(letters[i, j]))", textsize=16,
      padding=(-2, 0, -20, 0)) for i = 1:2, j = 1:2]
colgap!(fig.layout, 5)
rowgap!(fig.layout, 5)
fig
end
squares_layout()

```

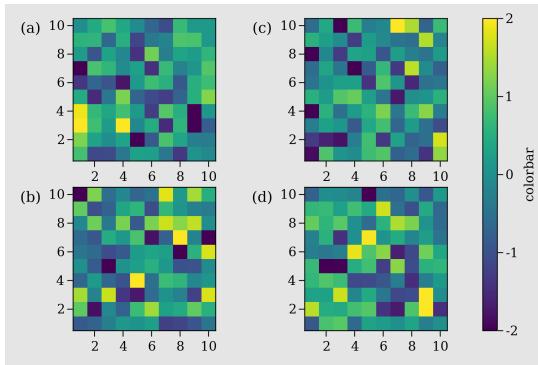


Figure 5.27: Squares layout.

onde todos os rótulos estão em **protrusões** e cada `Axis` tem uma razão `AspectData ↪()`. A `Colorbar` está localizada na terceira coluna e se expande da linha 1 até a linha 2.

O próximo caso usa o chamado **modo de alinhamento** `Mixed()`, o que é especialmente útil ao lidar com grandes espaços vazios entre `Axis` devido a tiques longos. Ainda, o módulo `Dates` da biblioteca padrão de Julia será necessário para esse exemplo.

```
using Dates
```

```

function mixed_mode_layout()
    seed!(123)
    longlabels = ["$(today() - Day(1))", "$(today())", "$(today() + Day(1))"]
    fig = Figure(resolution=(600, 400), fontsize=12,
                 backgroundcolor=:grey90, font="CMU Serif")
    ax1 = Axis(fig[1, 1])
    ax2 = Axis(fig[1, 2], xticklabelrotation=pi / 2, alignmode=Mixed(bottom=0),
               xticks=[1, 5, 10], longlabels)
    ax3 = Axis(fig[2, 1:2])
    ax4 = Axis(fig[3, 1:2])
    axs = [ax1, ax2, ax3, ax4]
    [lines!(ax, 1:10) for ax in axs]
    hidexdecorations!(ax3; ticks=false, grid=false)
    Box(fig[2:3, 1:2, Right()], color=(:slateblue1, 0.35))
    Label(fig[2:3, 1:2, Right()], "protrusion", rotation=pi / 2, textsize=14,

```

```

padding=(3, 3, 3, 3))
Label(fig[1, 1:2, Top()], "Mixed alignmode", textsize=16,
      padding=(0, 0, 15, 0))
colsizer!(fig.layout, 1, Auto(2))
rowsizer!(fig.layout, 2, Auto(0.5))
rowsizer!(fig.layout, 3, Auto(0.5))
rowgap!(fig.layout, 1, 15)
rowgap!(fig.layout, 2, 0)
colgap!(fig.layout, 5)
fig
end
mixed_mode_layout()

```

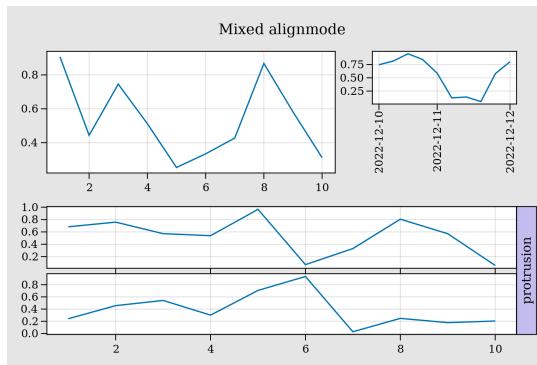


Figure 5.28: Mixed mode layout.

Aqui, o argumento `alignmode=Mixed(bottom=0)` desloca a caixa delimitadora para a parte inferior, de forma a alinhar com o painel à esquerda preenchendo o espaço.

Também, veja como `colsizer!` e `rowsizer!` estão sendo usados para diferentes colunas e linhas. Você também pode colocar um número ao invés de `Auto()` mas então tudo vai ser corrigido. E, além disso, pode-se também dar um `height ↪` ou `width` ao definir o `Axis`, como em `Axis(fig, height=50)` que será corrigido também.

### 5.6.1 *Axis* aninhado (subplots)

Também é possível definir um conjunto de `Axis` (*subplots*) explicitamente e usá-lo para construir uma figura principal com várias linhas e colunas. Por exemplo, o seguinte é um arranjo “complicado” de `Axis`:

```

function nested_sub_plot!(fig)
    color = rand(RGBf)
    ax1 = Axis(fig[1, 1], backgroundcolor=(color, 0.25))
    ax2 = Axis(fig[1, 2], backgroundcolor=(color, 0.25))
    ax3 = Axis(fig[2, 1:2], backgroundcolor=(color, 0.25))

```

```

ax4 = Axis(fig[1:2, 3], backgroundcolor=(color, 0.25))
return (ax1, ax2, ax3, ax4)
end

```

que, quando usado para construir uma figura mais complexa fazendo várias chamadas, obtemos:

```

function main_figure()
    fig = Figure()
    Axis(fig[1, 1])
    nested_sub_plot!(fig[1, 2])
    nested_sub_plot!(fig[1, 3])
    nested_sub_plot!(fig[2, 1:3])
    fig
end
main_figure()

```

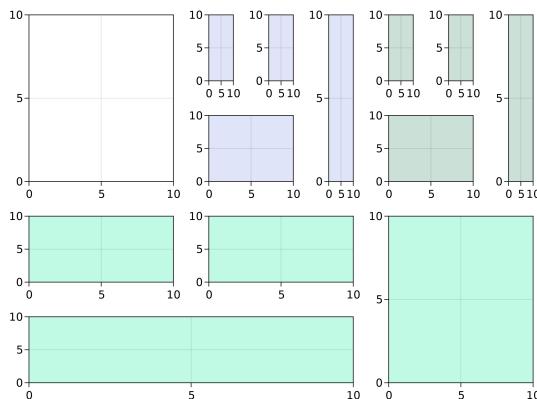


Figure 5.29: Main figure.

Observe que diferentes funções de `subplot` podem ser chamadas aqui. Também, cada `Axis` aqui é uma parte independente de `Figure`. Então, se você precisar fazer alguma operação `rowgap!` ou `colsizes!`, você precisará fazê-lo em cada um deles de forma independente ou em todos eles juntos.

Para `Axis` (`subplots`) agrupados podemos usar `GridLayout()` que, então, poderia ser usado para compor um `Figure`.

### 5.6.2 *GridLayout* aninhado

Ao usar o `GridLayout()` podemos agrupar `subplots`, permitindo mais liberdade na construção de figuras complexas. Aqui, usando nosso `nested_sub_plot!` anterior, definimos três subgrupos e um `Axis` normal:

```

function nested_Grid_Layouts()
    fig = Figure(backgroundcolor=RGBf(0.96, 0.96, 0.96))

```

```

ga = fig[1, 1] = GridLayout()
gb = fig[1, 2] = GridLayout()
gc = fig[1, 3] = GridLayout()
gd = fig[2, 1:3] = GridLayout()
gA = Axis(ga[1, 1])
nested_sub_plot!(gb)
axsc = nested_sub_plot!(gc)
nested_sub_plot!(gd)
[hidedecorations!(axsc[i], grid=false, ticks=false) for i = 1:length(axsc)]
colgap!(gc, 5)
rowgap!(gc, 5)
rowsize!(fig.layout, 2, Auto(0.5))
colsizes!(fig.layout, 1, Auto(0.5))
fig
end
nested_Grid_Layouts()

```

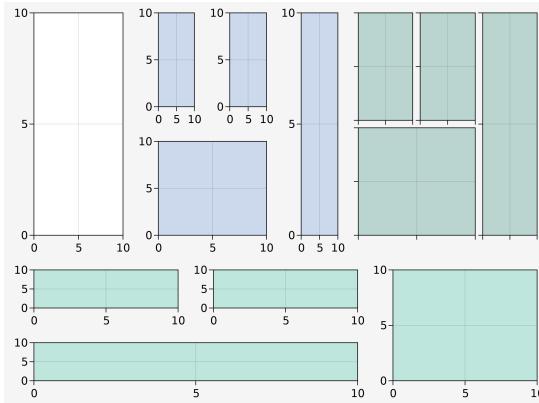


Figure 5.30: Nested Grid Layouts.

Agora, usando `rowgap!` ou `colsizes!` sobre cada grupo é possível `rowsize!`, `colsizes!`  
 $\hookrightarrow$  também pode ser aplicado ao conjunto de `GridLayout()`.

### 5.6.3 Plots `inset`

Atualmente, fazer gráficos `inset` é um pouco complicado. Aqui, mostramos duas maneiras possíveis de fazer isso definindo inicialmente as funções auxiliares. A primeira é fazendo um `BBox`, que fica em todo o espaço Figure:

```

function add_box_inset(fig; left=100, right=250, bottom=200, top=300,
    bgcolor=:grey90)
    inset_box = Axis(fig, bbox=BBox(left, right, bottom, top),
        ticklabelsize=12, yticklabelsize=12, backgroundcolor=bgcolor)
    # bring content upfront
    translate!(inset_box.scene, 0, 0, 10)
    elements = keys(inset_box.elements)
    filtered = filter(ele -> ele != :xaxis && ele != :yaxis, elements)

```

```

foreach(ele -> translate!(inset_box.elements[ele], 0, 0, 9), filtered)
    return inset_box
end

```

Então, o inset é feito facilmente, como em:

```

function figure_box_inset()
    fig = Figure(resolution=(600, 400))
    ax = Axis(fig[1, 1], backgroundcolor=:white)
    inset_ax1 = add_box_inset(fig; left=100, right=250, bottom=200, top=300,
        bgcolor=:grey90)
    inset_ax2 = add_box_inset(fig; left=500, right=600, bottom=100, top=200,
        bgcolor=(:white, 0.65))
    lines!(ax, 1:10)
    lines!(inset_ax1, 1:10)
    scatter!(inset_ax2, 1:10, color=:black)
    fig
end
figure_box_inset()

```

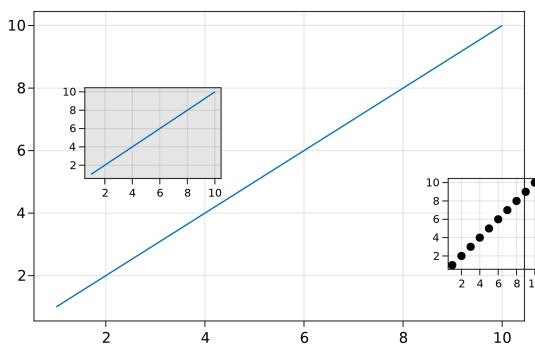


Figure 5.31: Figure box inset.

onde as dimensões Box estão vinculadas ao resolution Figure. Observe que um inset também pode estar fora do Axis. A outra abordagem é definir um novo Axis em uma posição `fig[i, j]` especificando seu `width`, `height`, `halign` and `valign`. Fazemos isso na seguinte função:

```

function add_axis_inset(; pos=fig[1, 1], halign=0.1, valign=0.5,
    width=Relative(0.5), height=Relative(0.35), bgcolor=:lightgray)
    inset_box = Axis(pos, width=width, height=height,
        halign=halign, valign=valign, xticklabelsize=12, yticklabelsize=12,
        backgroundcolor=bgcolor)
    # bring content upfront
    translate!(inset_box.scene, 0, 0, 10)
    elements = keys(inset_box.elements)
    filtered = filter(ele -> ele != :xaxis && ele != :yaxis, elements)
    foreach(ele -> translate!(inset_box.elements[ele], 0, 0, 9), filtered)
    return inset_box
end

```

```
end
```

Veja que no exemplo a seguir o `Axis` com fundo cinza será redimensionado se o tamanho total da figura for alterado. Os *insets* são limitados pelo posicionamento do `Axis`.

```
function figure_axis_inset()
    fig = Figure(resolution=(600, 400))
    ax = Axis(fig[1, 1], backgroundcolor=:white)
    inset_ax1 = add_axis_inset(; pos=fig[1, 1], halign=0.1, valign=0.65,
        width=Relative(0.3), height=Relative(0.3), bgcolor=:grey90)
    inset_ax2 = add_axis_inset(; pos=fig[1, 1], halign=1, valign=0.25,
        width=Relative(0.25), height=Relative(0.3), bgcolor=(:white, 0.65))
    lines!(ax, 1:10)
    lines!(inset_ax1, 1:10)
    scatter!(inset_ax2, 1:10, color=:black)
    fig
end
figure_axis_inset()
```

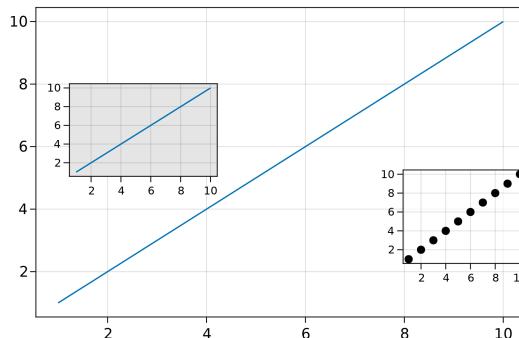


Figure 5.32: Figure axis inset.

E isso deve cobrir os casos mais usados para layout com Makie. Agora, vamos fazer alguns bons exemplos 3D com `GLMakie.jl`.

## 5.7 `GLMakie.jl`

`CairoMakie.jl` fornece todas as nossas necessidades de imagens 2D estáticas. Mas às vezes queremos interatividade, principalmente quando estamos lidando com imagens 3D. A visualização de dados em 3D também é uma prática comum para obter insights de seus dados. É aqui que `GLMakie.jl` pode ser útil, já que usa OpenGL<sup>11</sup> como um *backend* que adiciona interatividade e capacidade de resposta a *plots*. Como antes, um *plot* simples inclui, é claro, linhas e pontos. Então, vamos começar com eles e como já sabemos como os layouts funcionam, vamos colocar isso em prática.

<sup>11</sup> <http://www.opengl.org/>

### 5.7.1 Dispersão e Linhas

Para os gráficos de dispersão temos duas opções, a primeira é `scatter(x, y, ↪z)` e a segunda é `meshscatter(x, y, z)`. Na primeira, os marcadores não são escalonados nas direções dos eixos, mas na segunda, porque são geometrias reais no espaço 3D. Veja o próximo exemplo:

```
using GLMakie
GLMakie.activate!()
```

```
function scatters_in_3D()
    seed!(123)
    xyz = randn(10, 3)
    x, y, z = xyz[:, 1], xyz[:, 2], xyz[:, 3]
    fig = Figure(resolution=(1600, 400))
    ax1 = Axis3(fig[1, 1]; aspect=(1, 1, 1), perspectiveness=0.5)
    ax2 = Axis3(fig[1, 2]; aspect=(1, 1, 1), perspectiveness=0.5)
    ax3 = Axis3(fig[1, 3]; aspect=:data, perspectiveness=0.5)
    scatter!(ax1, x, y, z; markersize=50)
    meshscatter!(ax2, x, y, z; markersize=0.25)
    hm = meshscatter!(ax3, x, y, z; markersize=0.25,
        marker=FRect3D(Vec3f(0), Vec3f(1)), color=1:size(xyz)[2],
        colormap=:plasma, transparency=false)
    Colorbar(fig[1, 4], hm, label="values", height=Relative(0.5))
    fig
end
scatters_in_3D()
```

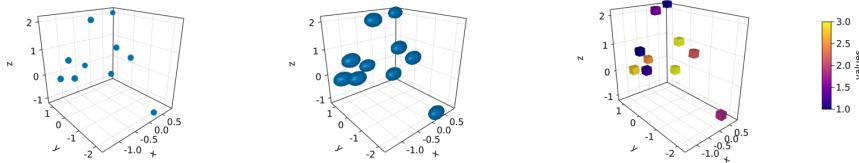


Figure 5.33: Scatters in 3D.

Observe também que uma geometria diferente pode ser passada como marcadores, ou seja, um quadrado/ângulo e podemos atribuir-lhes uma `colormap` também. No painel central, pode-se obter esferas perfeitas fazendo `aspect = :data` como no painel direito.

E fazendo `lines` ou `scatterlines` é também bem simples:

```
function lines_in_3D()
    seed!(123)
    xyz = randn(10, 3)
```

```

x, y, z = xyz[:, 1], xyz[:, 2], xyz[:, 3]
fig = Figure(resolution=(1600, 400))
ax1 = Axis3(fig[1, 1]; aspect=(1, 1, 1), perspectiveness=0.5)
ax2 = Axis3(fig[1, 2]; aspect=(1, 1, 1), perspectiveness=0.5)
ax3 = Axis3(fig[1, 3]; aspect=:data, perspectiveness=0.5)
lines!(ax1, x, y, z; color=1:size(xyz)[2], linewidth=3)
scatterlines!(ax2, x, y, z; markersize=50)
hm = meshscatter!(ax3, x, y, z; markersize=0.2, color=1:size(xyz)[2])
lines!(ax3, x, y, z; color=1:size(xyz)[2])
Colorbar(fig[2, 1], hm; label="values", height=15, vertical=false,
    flipaxis=false, ticksizer=15, tickalign=1, width=Relative(3.55 / 4))
fig
end
lines_in_3D()

```

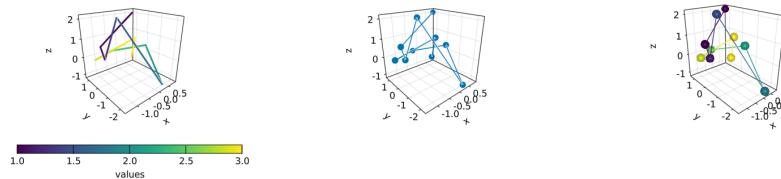


Figure 5.34: Lines in 3D.

Plotando uma `surface` também é fácil de ser fazer assim como um `wireframe` e linhas `contour` em 3D.

### 5.7.2 `surfaces, wireframe, contour, contourf e contour3d`

Para mostrar estes casos, utilizaremos a seguinte função `peaks`:

```

function peaks(; n=49)
    x = LinRange(-3, 3, n)
    y = LinRange(-3, 3, n)
    a = 3 * (1 .- x') .^ 2 .* exp.(-(x' .^ 2) .- (y .+ 1) .^ 2)
    b = 10 * (x' / 5 .- x' .^ 3 .- y .^ 5) .* exp.(-x' .^ 2 .- y .^ 2)
    c = 1 / 3 * exp.(-(x' .+ 1) .^ 2 .- y .^ 2)
    return (x, y, a .- b .- c)
end

```

A saída para as diferentes funções de plotagem é:

```

function plot_peaks_function()
    x, y, z = peaks()
    x2, y2, z2 = peaks(; n=15)
    fig = Figure(resolution=(1600, 400), fontsize=26)
    axs = [Axis3(fig[1, i]; aspect=(1, 1, 1)) for i = 1:3]

```

```

hm = surface!(axs[1], x, y, z)
wireframe!(axs[2], x2, y2, z2)
contour3d!(axs[3], x, y, z; levels=20)
Colorbar(fig[1, 4], hm, height=Relative(0.5))
fig
end
plot_peaks_function()

```

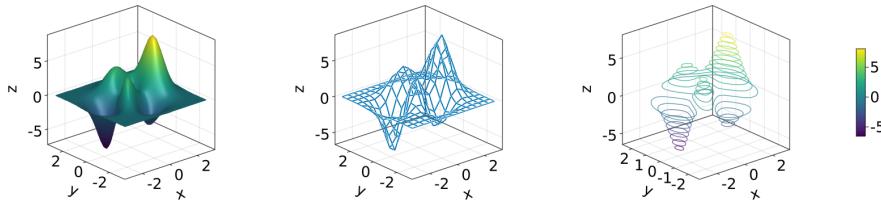


Figure 5.35: Plot peaks function.

Mas, também pode ser plotado com um `heatmap(x, y, z)`, `contour(x, y, z)` ou `contourf(x, y, z)`:

```

function heatmap_contour_and_contourf()
    x, y, z = peaks()
    fig = Figure(resolution=(1600, 400), fontsize=26)
    axs = [Axis(fig[i, i]; aspect=DataAspect()) for i = 1:3]
    hm = heatmap!(axs[1], x, y, z)
    contour!(axs[2], x, y, z; levels=20)
    contourf!(axs[3], x, y, z)
    Colorbar(fig[1, 4], hm, height=Relative(0.5))
    fig
end
heatmap_contour_and_contourf()

```

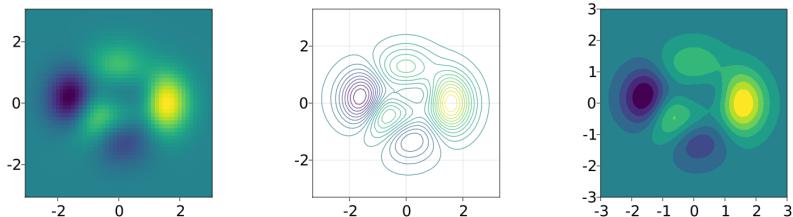


Figure 5.36: Heatmap contour and contourf.

Adicionalmente, ao mudarmos `Axis` para um `Axis3`, estes *plots* estarão automaticamente no plano x-y:

```

function heatmap_contour_and_contourf_in_a_3d_plane()
    x, y, z = peaks()
    fig = Figure(resolution=(1600, 400), fontsize=26)

```

```

axs = [Axis3(fig[1, i]) for i = 1:3]
hm = heatmap!(axs[1], x, y, z)
contour!(axs[2], x, y, z; levels=20)
contourf!(axs[3], x, y, z)
Colorbar(fig[1, 4], hm, height=Relative(0.5))
fig
end
heatmap_contour_and_contourf_in_a_3d_plane()

```

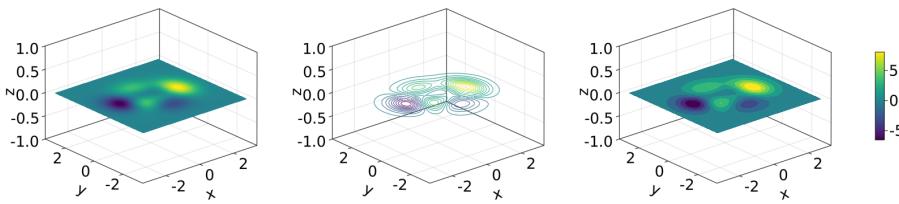


Figure 5.37: Heatmap, contour and contourf in a 3d plane.

Algo que também é bem fácil de fazer é misturar todas essas funções de plotagens em um único *plot*:

```
using TestImages
```

```

function mixing_surface_contour3d_contour_and_contourf()
    img = testimage("coffee.png")
    x, y, z = peaks()
    cmap = :Spectral_11
    fig = Figure(resolution=(1200, 800), fontsize=26)
    ax1 = Axis3(fig[1, 1]; aspect=(1, 1, 1), elevation=pi / 6, xzpanelcolor=(:black, 0.75),
                perspectiveness=0.5, yzpanelcolor=:black, zgridcolor=:grey70,
                ygridcolor=:grey70, xgridcolor=:grey70)
    ax2 = Axis3(fig[1, 3]; aspect=(1, 1, 1), elevation=pi / 6, perspectiveness
                ↪=0.5)
    hm = surface!(ax1, x, y, z; colormap=(cmap, 0.95), shading=true)
    contour3d!(ax1, x, y, z .+ 0.02; colormap=cmap, levels=20, linewidth=2)
    xmin, ymin, zmin = minimum(ax1.finallimits[])
    xmax, ymax, zmax = maximum(ax1.finallimits[])
    contour!(ax1, x, y, z; colormap=cmap, levels=20, transformation(:xy, zmax))
    contourf!(ax1, x, y, z; colormap=cmap, transformation(:xy, zmin))
    Colorbar(fig[1, 2], hm, width=15, ticksize=15, tickalign=1, height=Relative
             ↪(0.35))
    # transformations into planes
    heatmap!(ax2, x, y, z; colormap=:viridis, transformation(:yz, 3.5))
    contourf!(ax2, x, y, z; colormap=:CMRmap, transformation(:xy, -3.5))
    contourf!(ax2, x, y, z; colormap=:bone_1, transformation(:xz, 3.5))
    image!(ax2, -3 .. 3, -3 .. 2, rot90(img); transformation(:xy, 3.8))
    xlims!(ax2, -3.8, 3.8)

```

```

ylims!(ax2, -3.8, 3.8)
zlims!(ax2, -3.8, 3.8)
fig
end
mixing_surface_contour3d_contour_and_contourf()

```

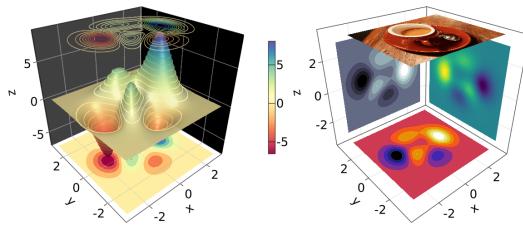


Figure 5.38: Mixing surface, contour3d, contour and contourf.

Não é ruim, certo? É claro que qualquer `heatmaps`, `contours`, `contourfs` ou `image` pode ser plotado em qualquer *plot*.

### 5.7.3 arrows e streamplots

`arrows` e `streamplot` são *plots* que podem ser úteis quando queremos saber as direções que uma determinada variável seguirá. Veja uma demonstração abaixo<sup>12</sup>:

```
using LinearAlgebra
```

Estamos usando o módulo `LinearAlgebra` da biblioteca padrão de Julia.

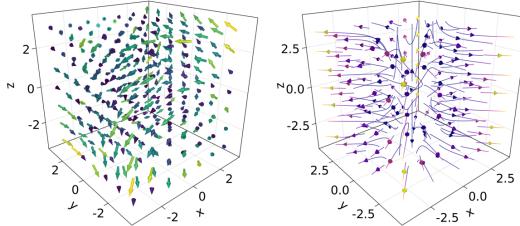
```

function arrows_and_streamplot_in_3d()
    ps = [Point3f(x, y, z) for x = -3:1:3 for y = -3:1:3 for z = -3:1:3]
    ns = map(p -> 0.1 * rand() * Vec3f(p[2], p[3], p[1]), ps)
    lengths = norm.(ns)
    flowField(x, y, z) = Point(-y + x * (-1 + x^2 + y^2)^2, x + y * (-1 + x^2 +
        ↪y^2)^2,
        z + x * (y - z^2))
    fig = Figure(resolution=(1200, 800), fontsize=26)
    axs = [Axis3(fig[1, i]; aspect=(1, 1, 1), perspective=0.5) for i = 1:2]
    arrows!(axs[1], ps, ns, color=lengths, arrowsize=Vec3f0(0.2, 0.2, 0.3),
        linewidth=0.1)
    streamplot!(axs[2], flowField, -4 .. 4, -4 .. 4, -4 .. 4, colormap=:plasma,
        gridsize=(7, 7), arrow_size=0.25, linewidth=1)
    fig
end
arrows_and_streamplot_in_3d()

```

Outros exemplos interessantes são `mesh(obj)`, `volume(x, y, z, vals)`, e `contour(x, ↪y, z, vals)`.

Figure 5.39: Arrows and streamplot in 3d.



#### 5.7.4 Mesh e Volumes

Visualizações de *mesh* são úteis quando você quer plotar geometrias, como uma `Sphere` ou um Retângulo, ex: `FRect3D`. Outra abordagem para visualizar pontos no espaço 3D é chamando as funções `volume` e `contour`, que implementam *ray tracing*<sup>13</sup> para simular uma grande variedade de efeitos ópticos. Veja os próximos exemplos:

<sup>13</sup> [https://en.wikipedia.org/wiki/Ray\\_tracing\\_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics))

```
using GeometryBasics
```

```
function mesh_volume_contour()
    # mesh objects
    rectMesh = FRect3D(Vec3f(-0.5), Vec3f(1))
    recmesh = GeometryBasics.mesh(rectMesh)
    sphere = Sphere(Point3f(0), 1)
    # https://juliageometry.github.io/GeometryBasics.jl/stable/primitives/
    spheremesh = GeometryBasics.mesh(Tesselation(sphere, 64))
    # uses 64 for tesselation, a smoother sphere
    colors = [rand() for v in recmesh.position]
    # cloud points for volume
    x = y = z = 1:10
    vals = randn(10, 10, 10)
    fig = Figure(resolution=(1600, 400))
    axs = [Axis3(fig[1, i]; aspect=(1, 1, 1), perspectiveness=0.5) for i = 1:3]
    mesh!(axs[1], recmesh; color=colors, colormap=:rainbow, shading=false)
    mesh!(axs[1], spheremesh; color=(:white, 0.25), transparency=true)
    volume!(axs[2], x, y, z, vals; colormap=Reverse(:plasma))
    contour!(axs[3], x, y, z, vals; colormap=Reverse(:plasma))
    fig
end
mesh_volume_contour()
```

Note que aqui estamos traçando duas *mesh* no mesmo eixo, uma esfera transparente e um cubo. Até agora, cobrimos a maioria dos casos de uso 3D. Outro exemplo é `?linesegments`.

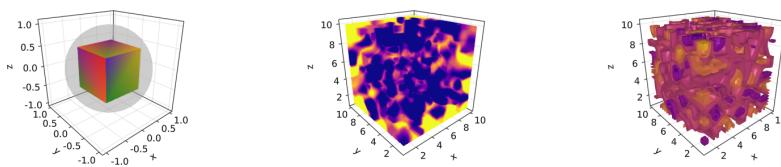


Figure 5.40: Mesh volume contour.

Tomando como referência o exemplo anterior, pode-se fazer o seguinte *plot* personalizado com esferas e retângulos:

```
using GeometryBasics, Colors
```

Para as esferas, vamos fazer um *grid* retangular. Além disso, usaremos uma cor diferente para cada uma delas. Adicionalmente, podemos misturar esferas e um plano retangular. Em seguida, definimos todos os dados necessários.

```
seed!(123)
spheresGrid = [Point3f(i,j,k) for i in 1:2:10 for j in 1:2:10 for k in 1:2:10]
colorSphere = [RGBA(i * 0.1, j * 0.1, k * 0.1, 0.75) for i in 1:2:10 for j in
    ↪1:2:10 for k in 1:2:10]
spheresPlane = [Point3f(i,j,k) for i in 1:2.5:20 for j in 1:2.5:10 for k in
    ↪1:2.5:4]
cmap = get(colorschemes[:plasma], LinRange(0, 1, 50))
colorsPlane = cmap[rand(1:50,50)]
rectMesh = FRect3D(Vec3f(-1, -1, 2.1), Vec3f(22, 11, 0.5))
recmesh = GeometryBasics.mesh(rectMesh)
colors = [RGBA(rand(4)...)] for v in recmesh.position]
```

Então, o *plot* é feito simplesmente com:

```
function grid_spheres_and_rectangle_as_plate()
    fig = with_theme(theme_dark()) do
        fig = Figure(resolution=(1200, 800))
        ax1 = Axis3(fig[1, 1]; aspect=:data, perspectiveness=0.5, azimuth=0.72)
        ax2 = Axis3(fig[1, 2]; aspect=:data, perspectiveness=0.5)
        meshscatter!(ax1, spheresGrid; color = colorSphere, markersize = 1,
            shading=false)
        meshscatter!(ax2, spheresPlane; color=colorsPlane, markersize = 0.75,
            lightposition=Vec3f(10, 5, 2), ambient=Vec3f(0.95, 0.95, 0.95),
            backlight=1.0f0)
        mesh!(recmesh; color=colors, colormap=:rainbow, shading=false)
        limits!(ax1, 0, 10, 0, 10, 0, 10)
        fig
    end
    fig
end
```

```
grid_spheres_and_rectangle_as_plate()
```

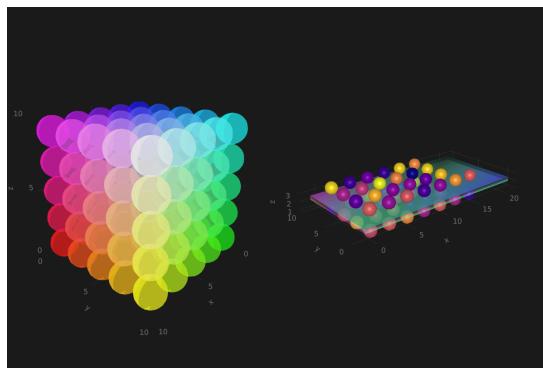


Figure 5.41: Grid spheres and rectangle as plate.

Aqui, o retângulo é semi-transparente devido ao canal alfa adicionado à cor RGB. A função de retângulo é bastante versátil, por exemplo, `box 3D` é fácil implementar que por sua vez pode ser usada para traçar um histograma 3D. Veja nosso próximo exemplo, onde estamos usando novamente nossa função `peaks` e algumas definições adicionais:

```
x, y, z = peaks(; n=15)
δx = (x[2] - x[1]) / 2
δy = (y[2] - y[1]) / 2
cbarPal = :Spectral_11
ztmp = (z .- minimum(z)) ./ (maximum(z) .- minimum(z)))
cmap = get(colorschemes[cbarPal], ztmp)
cmap2 = reshape(cmap, size(z))
ztmp2 = abs.(z) ./ maximum(abs.(z)) .+ 0.15
```

aqui  $\delta x, \delta y$  são usados para especificar o tamanho das `box`. `cmap2` será a cor de cada `box` e `ztmp2` será usado como o parâmetro de transparência. Veja o resultado na próxima figura.

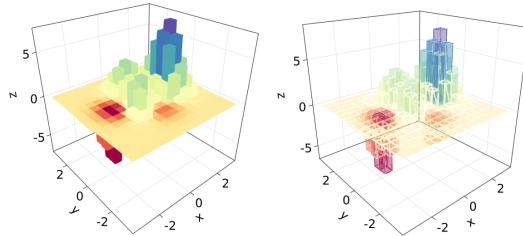
```
function histogram_or_bars_in_3d()
    fig = Figure(resolution=(1200, 800), fontsize=26)
    ax1 = Axis3(fig[1, 1]; aspect=(1, 1, 1), elevation=π/6,
                perspectiveness=0.5)
    ax2 = Axis3(fig[1, 2]; aspect=(1, 1, 1), perspectiveness=0.5)
    rectMesh = FRect3D(Vec3f0(-0.5, -0.5, 0), Vec3f0(1, 1, 1))
    meshscatter!(ax1, x, y, 0*z, marker = rectMesh, color = z[:],
                markersize = Vec3f.(2δx, 2δy, z[:]), colormap = :Spectral_11,
                shading=false)
    limits!(ax1, -3.5, 3.5, -3.5, 3.5, -7.45, 7.45)
    meshscatter!(ax2, x, y, 0*z, marker = rectMesh, color = z[:],
                markersize = Vec3f.(2δx, 2δy, z[:]), colormap = (:Spectral_11, 0.25),
                shading=false, transparency=true)
    for (idx, i) in enumerate(x), (idy, j) in enumerate(y)
```

```

rectMesh = FRect3D(Vec3f(i - δx, j - δy, 0), Vec3f(2δx, 2δy, z[idx, idy]
    ↪]))
recmesh = GeometryBasics.mesh(rectMesh)
lines!(ax2, recmesh; color=(cmap2[idx, idy], ztmp2[idx, idy]))
end
fig
end
histogram_or_bars_in_3d()

```

Figure 5.42: Histogram or bars in 3d.



Note que você pode também usar `lines` ou `wireframe` sobre um objeto `mesh`.

### 5.7.5 Linhas Preenchidas e `band`

Para o nosso último exemplo vamos mostrar como fazer uma curva preenchida em 3d com `band` e alguns `linesegments`:

```

function filled_line_and_linesegments_in_3D()
    xs = LinRange(-3, 3, 10)
    lower = [Point3f(i, -i, 0) for i in LinRange(0, 3, 100)]
    upper = [Point3f(i, -i, sin(i) * exp(-(i + i))) for i in range(0, 3, length
        ↪=100)]
    fig = Figure(resolution=(1200, 800))
    axs = [Axis3(fig[1, i]; elevation=pi/6, perspectiveness=0.5) for i = 1:2]
    band!(axs[1], lower, upper, color=repeat(norm.(upper), outer=2), colormap=:
        ↪CMRmap)
    lines!(axs[1], upper, color=:black)
    linesegments!(axs[2], cos.(xs), xs, sin.(xs), linewidth=5, color=1:length(xs
        ↪))
    fig
end
filled_line_and_linesegments_in_3D()

```

Finalmente, nossa jornada fazendo *plots* 3D chegou ao fim. Você pode combinar tudo o que expostos aqui para criar imagens 3D incríveis!

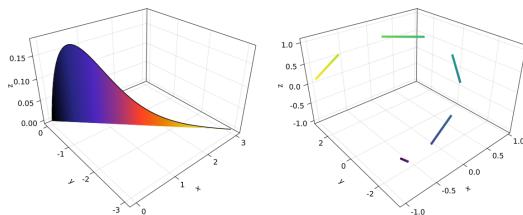


Figure 5.43: Filled line and linesegments in 3D.



# 6 Apêndice

## 6.1 Versões dos Pacotes

Esse livro foi feito com Julia 1.7.3 e os seguintes pacotes:

```
Books 1.2.8
CSV 0.10.8
CairoMakie 0.7.5
CategoricalArrays 0.10.7
ColorSchemes 3.20.0
Colors 0.12.8
DataFrames 1.4.4
Distributions 0.25.79
FileIO 1.16.0
GLMakie 0.5.5
GeometryBasics 0.4.5
ImageMagick 1.2.2
LaTeXStrings 1.3.0
Makie 0.16.6
QuartzImageIO 0.7.4
Reexport 1.2.2
StatsBase 0.33.21
TestImages 1.7.1
XLSX 0.7.10
```

Build: 2022-12-11 21:5 UTC

## 6.2 Formatação

Neste livro, nós tentamos manter a formatação de código-fonte o mais consistente possível. Isto permite uma melhor leitura e escrita de código-fonte. Definimos o padrão de formatação em três partes.

### 6.2.1 Julia Style Guide

Primeiramente, nós tentamos aderir as convenções do Julia Style Guide<sup>1</sup>. Mais importante, escrevemos funções e não scripts (veja também Section 1.2). Além disso, usamos convenções de nomenclatura consistente com o módulo `base` de Julia:

<sup>1</sup> <https://docs.julialang.org/en/v1/manual/style-guide/>

- Uso de *camelcase* para módulos: `module` JuliaDataScience, `struct` MyPoint. (Note que a denominação de *camelcase* é porque a capitalização das palavras, tais como “iPad” ou “CamelCase,” faz com que a palavra se assemelhe que nem as costas de um camelo.)
- Denominação de funções usando caixa baixa (*lowercase*) e separando palavras por *underline* (\_). Também é permitido omitir o separador quando nomeando funções. Por exemplo, estes nomes de funções são consistentes com as convenções: `my_function`, `myfunction` e `string2int`.

Adicionalmente, evitamos usar parênteses em condicionais, ou seja, escrevemos `if` `a == b` ao invés de `if` (`a == b`) e usamos 4 espaços para cada nível de indentação.

### 6.2.2 BlueStyle

O Blue Style Guide<sup>2</sup> adiciona diversas convenções aos padrões do Guia de Estilo de Julia. Algumas dessas regras podem soar pedantes, mas descobrimos que elas fazem o código ficar mais legível.

<sup>2</sup> <https://github.com/Invenia/BlueStyle>

Do Blue Style Guide, nós aderimos especificamente à:

- No máximo 92 caracteres por linha em código-fonte (para arquivos Markdown, linhas mais longas são permitidas).
- Quando carregando código com `using`, usar apenas uma declaração por módulo por linha.
- Não há whitespace no fim das linhas. Whitespace no fim das linhas faz com que a inspeção do código seja mais difícil pois eles não mudam o comportamento do código mas podem ser interpretados como mudanças.
- Evitar espacos adicionais dentro dos parênteses. Escreva `string(1, 2)` ao invés de `string( 1 , 2 )`.
- Variáveis globais devem ser evitadas.
- Tente limitar nomes de funções para apenas uma ou duas palavras.
- Use o ponto-e-virgula para distinção se um argumento é posicional ou de palavra-chave. Por exemplo, `func(x; y=3)` ao invés de `func(x, y=3)`.
- Evite usar espaços múltiplos para alinhar coisas. Escreva

```
a = 1
lorem = 2
```

ao invés de

```
a      = 1
lorem = 2
```

- Sempre que apropriado, coloque espaços ao redor de operadores binários, por exemplo, `1 == 2` ou `y = x + 1`.
- Indente quotações triplas:

```
s = """
    my long text:
    [...]
    the end.
"""
```

- Não omite zeros in floats (mesmo que Julia permita). Portanto, escreva `1.0` ao invés de `1.` e escreva `0.1` ao invés de `.1`.
- Use `in` dentro de loops `for` e não `=` ou `∈` (mesmo que Julia permita).

### 6.2.3 Nossos incrementos ao Blue Style Guide

- No texto, referenciamos uma chamada de função `M.foo(3, 4)` como `M.foo` e não `M.foo(...)` ou `M.foo()`.
- Quando falando sobre pacotes, tais como o pacote `DataFrames`, nós explicitamente escrevemos sempre `DataFrames.jl`. Isto faz com que seja fácil reconhecer que estamos nos referindo à um pacote.
- Para nome de arquivos, mantemos a notação como “`file.txt`” e não `file.txt` ou `file.txt`, pois é consistente com o código-fonte.
- Para nomes e colunas em tabelas, tais como a coluna `x`, usamos a coluna `:x`, porque é consistente com o código-fonte.
- Não usamos símbolos Unicode fora dos blocos de código. Isto foi necessário por conta de um bug na geração do PDF.
- A linha antes de cada código de bloco termina com dois pontos (`:`) para indicar que aquela linha pertence aquele bloco de código.

## Carregamento de símbolos

Prefira sempre carregar símbolos de maneira explícita, ou seja, prefira `using A ↩: foo` ao invés de `using A` quando não estiver usando o REPL (veja também *JuMP Style Guide, 2021*). Neste contexto, um símbolo significa um identificado de um objeto. Por exemplo, mesmo que não pareça natural, internamente

DataFrame, π e CSV são todos símbolos. Podemos averiguar isto se usarmos uma função introspectiva de Julia tal como `isdefined`:

```
isdefined(Main, :π)
```

```
true
```

Adjacente de ser explícito no uso de `using`, prefira também `using A: foo` ao invés de `import A: foo` porque este último faz com que seja fácil estender acidentalmente `foo`. Note que isto não é somente aplicável para Julia: carregamento implícito de símbolos por `from <module> import *` também é desencorajado em Python (van Rossum et al., 2001).

A razão da importância de ser explícito está relacionada com versionamento semântico. Com versionamento semântico (<http://semver.org>), o número da versão está relacionado se um pacote possui ou não mudanças *breaking*. Por exemplo, uma mudança *non-breaking* que atualiza o pacote `A` se dá quando o pacote migra da versão `0.2.2` para `0.2.3`. Com tais mudanças *non-breaking*, você não precisa se preocupar se o seu pacote vai quebrar (*break*), em outras palavras, dar um erro ou mudar o comportamento de execução. Se um pacote `A` migra de versão `0.2` para `1.0`, então esta atualização é *breaking* e você provavelmente terá que fazer mudanças no seu código para fazer com que o pacote `A` funcione novamente. **Porém**, exportando símbolos extras é considerado uma mudança *non-breaking*. Então, com carreamento implícito de símbolos **mudanças non-breaking podem quebrar seu pacote**. Por isso que é uma boa prática apeans carregar símbolos de maneira explícita.

# Referências

- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1), 65–98.
- Chen, M., Mao, S., & Liu, Y. (2014). Big data: A survey. *Mobile Networks and Applications*, 19(2), 171–209.
- Domo. (2018). *Data never sleeps 6.0*. [https://www.domo.com/assets/downloads/18\\_domo\\_data-never-sleeps-6+verticals.pdf](https://www.domo.com/assets/downloads/18_domo_data-never-sleeps-6+verticals.pdf)
- Fitzgerald, S., Jimenez, D. Z., S., F., Yorifuji, Y., Kumar, M., Wu, L., Carosella, G., Ng, S., Parker, P., R. Carter, & Whalen, M. (2020). IDC FutureScape: Worldwide digital transformation 2021 predictions. *IDC FutureScape*.
- Gantz, J., & Reinsel, D. (2012). The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2007(2012), 1–16.
- JuMP style guide. (2021). <https://jump.dev/JuMP.jl/v0.21/developers/style/#using-vs.-import>
- Khan, N., Yaqoob, I., Hashem, I. A. T., Inayat, Z., Mahmoud Ali, W. K., Alam, M., Shiraz, M., & Gani, A. (2014). Big data: Survey, technologies, opportunities, and challenges. *The Scientific World Journal*, 2014.
- Meng, X.-L. (2019). Data science: An artificial ecosystem. *Harvard Data Science Review*, 1(1). <https://doi.org/10.1162/99608f92.ba20f892>
- Perkel, J. M. (2019). Julia: Come for the syntax, stay for the speed. *Nature*, 572(7767), 141–142. <https://doi.org/10.1038/d41586-019-02310-3>
- Storopoli, J. (2021). *Bayesian statistics with julia and turing*. <https://storopoli.io/Bayesian-Julia>
- tanmay bakshi. (2021). *Baking Knowledge into Machine Learning ModelsChris Rackauckas on TechLifeSkills w/ Tanmay Ep.55*. <https://youtu.be/moyPlhv w4Nk>
- TEDx Talks. (2020). *A programming language to heal the planet together: Julia | Alan Edelman | TEDxMIT*. <https://youtu.be/qGW0GT1rCvs>
- van Rossum, G., Warsaw, B., & Coghlan, N. (2001). *Style guide for Python code* (PEP No. 8). <https://www.python.org/dev/peps/pep-0008/>

Wickham, H. (2011). The split-apply-combine strategy for data analysis. *Journal of Statistical Software*, 40(1), 1–29.