Introdução ao R

Aplicações em Finanças e Economia

Marcelo S. Perlin (marcelo.perlin@ufrgs.br)

15/03/2024

Introdução ao R - Aplicações em Finanças e Economia

por **Marcelo S. Perlin**

© 2023 Marcelo S. Perlin. Todos direitos reservados.

Publicação Independente. Impresso sob demanda por Amazon.com.
Versão Online disponível em *https://www.msperlin.com/introR/*

Revisão de texto: Diversos
Capa: Rubens Lima - http://capista.com.br/
ISBN (paperback): 9798393912772
ISBN (hardcover): 9798394656910

Histórico de edições:
Primeira edição: 01/XX/202X

Embora o autor tenha boa fé para garantir que as instruções e o código contidos neste trabalho sejam precisos, ele se exime de toda responsabilidade por erros ou omissões, incluindo, sem limitação, a responsabilidade por danos resultantes do uso ou da confiança neste trabalho e em seus resultados. O uso das informações contidas neste trabalho é por sua conta e risco. Se qualquer código deste livro estiver sujeito a licenças de código aberto ou direitos de propriedade intelectual de terceiros, o cumprimento desses direitos e licenças é de sua responsabilidade como usuário.

Todo o código contigo no livro é disponibilizado pela generosa licença do MIT. Portanto, sinta-se livre para utilizá-lo no seu trabalho, desde de que a origem do código seja citada. Uma sugestão de citação é disponibilizada abaixo:

Perlin, M. S. Introdução ao R. Aplicações em Finanças e Economia, Porto Alegre: Marcelo S. Perlin (publicação independente), 2023.

ÍNDICE

PREFÁCIO

Este livro introduz o leitor ao uso do R como ferramenta de computação e análise de dados. O conteúdo foi inspirado em minha outra obra, "Análise de Dados Financeiros e Econômicos com o R", publicada em 2017. Com o tempo, logo percebi que seria desejável ter diversos livros focados dos capítulos, ao invés de uma única obra no tema. Ao final deste livro você irá aprender como utilizar o R para importar e manipular dados e, por fim, reportar tabelas e figuras de uma pesquisa em um relatório técnico.

Sou professor da Universidade Federal do Rio Grande do Sul, onde leciono, na graduação e na pós-graduação, disciplinas relacionadas ao uso do R em análise de dados. A experiência em sala de aula permite enxergar onde os alunos mais erram, e qual o melhor caminho didático para aprender a usar o R. Seja voce um pesquisador acadêmico, ou futuro analista de dados, este livro é um projeto pessoal para disseminar conhecimento sobre a ferramenta para um público maior e mais diversificado.

Outra motivação que tive para escrever o livro foi minha experiência na utilização de códigos disponibilizados por outros pesquisadores. Na maioria das vezes, esses códigos são desorganizados, pouco claros e, possivelmente, funcionam apenas no computador do pesquisador que os escreveu! Assim como se espera que um artigo científico esteja bem escrito, também se deve esperar que o código por trás da respectiva pesquisa seja de qualidade. Porém, esse não é o caso na grande maioria das vezes. Com este livro, irei atacar esse problema, formalizando uma estrutura de código voltada à reprodutibilidade científica, focando em organização e usa-

bilidade. Nesse sentido, espero que as futuras gerações de pesquisadores estejam mais bem preparadas para compartilhar o seu trabalho.

Antes de prosseguir, um aviso. Não iremos trabalhar usos avançados do R. O conteúdo será limitado a exemplos simples e práticos de utilização do *software* para a construção de pesquisa baseada em dados financeiros e econômicos. De fato, um dos desafios na escrita deste livro foi definir o limite entre o material introdutório e o avançado. Procurei, sempre que possível, dosar gradualmente o nível de complexidade. Para leitores interessados em conhecer funções avançadas do programa e o seu funcionamento interno, sugiro a leitura do manual oficial do R (Teetor 2011) e de Wickham (2019).

Com este livro irás aprender os seguinte tópicos:

- Uso do R e RStudio;
- Uso e instalação de pacotes do CRAN e do Github
- Interagir com o seu computador e a internet
- Importação de dados locais e da internet
- Manipulação de objetos básicos e de armazenamento no R
- introdução a programação

Material Suplementar

Todo o material usado no livro, incluindo exemplos de código separados por capítulos, está publicamente disponível na Internet e distribuído com um pacote R denominado `introR`. Este inclui arquivos de dados, e algumas funções que irão facilitar a execução dos exemplos do livro. Se você planeja, como sugerido, escrever código enquanto lê o livro, este pacote ajudará muito em sua jornada.

Para instalar este pacote no seu computador, basta executar algumas linhas de comando no R. Veja o código destacado a seguir e copie e cole o mesmo no prompt do RStudio (canto inferior esquerdo da tela, com um sinal “>”) e pressione Enter para cada comando. Esteja ciente de que você precisará do R e RStudio instalados em seu computador (consulte (**instalacao?**)).

```
# install devtools dependency
install.packages('devtools')

# install book package
```

```
devtools::install_github('msperlin/introR')
```

O que este código fará é instalar o pacote `devtools`, uma dependência necessária para instalar código do Github – um repositório de pacotes onde o livro está hospedado. Depois disso, uma chamada para `install_github('msperlin/introR')` irá instalar o pacote em seu computador.

Depois da instalação, todos os arquivos do livro estarão disponíveis localmente, salvos em uma pasta do seu computador. Iremos usar todos estes arquivos futuramente. Opcionalmente, caso quiser olhar os arquivos, podes copiar todo conteúdo para outra pasta com o código a seguir:

```
introR::copy_book_files(path_to_copy = '~')
```

Veja que o tilda (~) é um atalho para o diretório "Documentos" no Windows (ou "home" no Linux/Mac). Assim, o código anterior descompactará o arquivo do livro na pasta "Documentos/introR-files". O pacote também inclui várias outras funções que serão usadas ao longo do livro. Se você preferir a maneira antiga e consagrada de baixar o arquivo e descompactar manualmente, podes encontrar uma cópia no site do livro.

Conteúdo para Instrutores

Se você for um instrutor de R, aqui encontrarás muito material para usar em suas aulas:

Exercícios estáticos na internet Cada capítulo deste livro inclui exercícios que seus alunos podem praticar. Todas as soluções estão disponíveis na versão online do livro, disponível em .

Exercícios exportáveis para pdf ou plataformas de *e-learning* Todos exercícios do livro estão no formato `exams` (Zeileis et al. 2022) e são exportáveis para arquivos em pdf ou então para plataformas de *e-learning* tal como o *Moodle* ou *Blackboard*. Veja este post no blog para maiores detalhes.

Acesso ao livro na internet Existe uma versão online e gratuita do livro, disponível em <>. Seu alunos terão acesso ao material do próprio celular.

Espero que goste deste livro. O conteúdo tem sido compilado por um longo período de tempo, a base de muito suor e, literalmente, litros de ~~café~~ chá por parte do autor.

Boa leitura!

Marcelo S. Perlin

AGRADECIMENTOS

Este livro não seria possível sem a preciosa autonomia do meu cargo de professor universitário, sendo um dos facilitadores de todos os meus projetos literários, os quais me dedico com muita paixão. Assim, deixo aqui o meu agradecimento a UFRGS (Universidade Federal do Rio Grande do Sul), por possibilitar e incentivar este empreendimento no mercado literário.

Adicionalmente, não posso também deixar de agradecer a toda a comunidade do R. Em especial, agradeço os autores do pacote **{quarto}** (Allaire 2023), sem o qual não seria possível compilar este livro de uma forma tão fácil. Adicionalmente, abaixo destaco os respectivos pacotes disponíveis no CRAN utilizados na produção do livro e suas devidas referências. A lista foi gerada automaticamente e está em ordem alfabética.

{base} (R Core Team 2023b), **{dplyr}** (Wickham, François, et al. 2023), **{forcats}** (Wickham 2023a), **{fs}** (Hester, Wickham, e Csárdi 2023), **{ggplot2}** (Wickham, Chang, et al. 2023), **{glue}** (Hester e Bryan 2022), **{gt}** (Iannone et al. 2023), **{knitr}** (Xie 2023), **{purrr}** (Wickham e Henry 2023), **{quarto}** (Allaire 2023), **{readr}** (Wickham, Hester, e Bryan 2023), **{renv}** (Ushey e Wickham 2023), **{reticulate}** (Ushey, Allaire, e Tang 2023), **{rmarkdown}** (Allaire et al. 2023), **{stats}** (R Core Team 2023c), **{tibble}** (Müller e Wickham 2023), **{tidyr}** (Wickham, Vaughan, e Girlich 2023), **{tidyverse}** (Wickham 2023c)

CAPÍTULO 1

INTRODUÇÃO

É indiscutível que estamos vivendo em um momento histórico onde a produção de informações é totalmente digitalizada e armazenada. Desde compras na internet a geolocalização pelo celular, um volume imenso de dados é produzido e guardado em bancos de dados de diferentes empresas e organizações.

Um ambiente competitivo motiva as empresas a investirem na formação de equipes especializadas em análise de dados para tomarem as melhores decisões corporativas. Sem dúvida, o período atual é bastante prolífico para profissionais com conhecimento e experiência na utilização das ferramentas corretas para a análise computacional de dados.

É nesse ambiente que se destaca o papel do R, uma linguagem de programação voltada para a resolução de problemas computacionais envolvendo análise, processamento, visualização e modelagem de dados. Nas próximas seções, explicarei o que é o R e quais são suas vantagens frente a outras alternativas.

1.1 O que é o R

O R é uma linguagem de programação voltada para a resolução de problemas estatísticos e para a visualização gráfica de dados. O código base

do R foi inspirado na linguagem *S*, inicialmente criada no laboratório da **Bell/AT&T** por **John Chambers** e seus colegas. A ideia de criar uma linguagem de programação voltada a estatística foi redesenhada por dois acadêmicos, **Ross Ihaka** e **Robert Gentleman**, ambos da universidade de *Auckland* - Nova zelândia.

Hoje, R é sinônimo de programação voltada à análise de dados, com uma larga base de usuários acadêmicos e da indústria. É muito provável que analistas de áreas diversas, desde Economia até Biologia, ou mesmo Música, encontrem no R uma quantidade significativa de códigos que facilitem suas análises. No campo empresarial, grandes empresas como *Google* e *Microsoft* já o adotaram como a linguagem interna para a análise de dados. O R é atualmente mantido pelo **R Foundation** e o **R Consortium**, um esforço coletivo para financiar projetos de extensão da linguagem.

E o mais importante: **o R é totalmente livre** e disponível em vários sistemas operacionais. Seja você um usuário do Windows, do Linux/Unix ou do MacOS, existe uma instalação do R para a sua plataforma, e os seus códigos devem rodar entre uma e outra com mínimas alterações.

> Qual a origem do nome R?
>
> O que pode ser dito como uma ato pouco criativo, os desenvolvedores escolheram a letra R pois era a primeira letra de seus primeiros nomes, Ross e Robert.

1.2 Por que Escolher o R

Possivelmente você esteja se perguntando por que deve optar pelo R e investir tempo em sua aprendizagem, ao invés de escolher uma outra linguagem.

Em primeiro lugar, **o R é uma plataforma madura, estável, continuamente suportada e intensamente utilizada na indústria.** Ao escolher o R, você terá a bagagem computacional necessária não somente para uma carreira acadêmica em pesquisa científica, mas também para o trabalho em organizações privadas. Nesse sentido, com a escolha de outra linguagem de programação menos popular ou proprietária/comercial, é provável que tal linguagem não seja utilizada em um ambiente empresarial e isso pode limitar as suas futuras oportunidades profissionais. Sem dúvida, o conhecimento de programação em um plataforma aberta de análise de dados aumenta a sua atratividade como profissional.

Aprender a linguagem do R é fácil. A experiência que tenho ensinando o R em sala de aula me permite afirmar que os alunos, mesmo aqueles sem experiência em programação, apresentam facilidade em aprender a linguagem e em utilizá-la para criar seus próprios códigos de pesquisa. A linguagem é intuitiva e certas normas e funções podem ser estendidas para diferentes casos. Após entender como o programa funciona, fica fácil descobrir novas funcionalidades partindo de uma lógica anterior. Essa notação compartilhada entre procedimentos facilita o aprendizado.

A interface do R e RStudio torna o uso da ferramenta bastante produtivo. A interface gráfica aberta e gratuita disponibilizada pela RStudio/Posit facilita o uso do software, assim como a produtividade do usuário. Utilizando o ambiente de trabalho do R e RStudio, têm-se a disposição diversas ferramentas que facilitam e estendem o uso da plataforma.

Os pacotes do R permitem as mais diversas funcionalidades. Logo veremos que o R permite o uso de código de outros usuários, os quais podem ser localmente instalados através de um simples comando. Esses estendem a linguagem básica do R e possibilitam as mais diversas funcionalidades. Além das funções óbvias de analisar dados, podes utilizar o R para mandar emails, escrever e publicar um livro, criar provas objetivas com conteúdo dinâmico, contar piadas e poemas (é sério!), acessar e coletar dados da internet, entre diversas outras funcionalidades.

O R tem compatibilidade com diferentes linguagens e sistemas operacionais. Se, por algum motivo, o usuário precisar utilizar código de outra linguagem de programação tal como *C++*, *Python*, *Julia*, é fácil integrar a mesma dentro de um programa do R. Diversos pacotes estão disponíveis para facilitar esse processo. Portanto, o usuário nunca fica restrito a uma única linguagem e tem flexibilidade para escolher as suas ferramentas de trabalho.

O R é totalmente gratuito! O programa e todos os seus pacotes são completamente livres, não tendo custo algum de licença e distribuição. Portanto, você pode utilizá-lo e modificá-lo livremente no seu trabalho ou computador pessoal. Essa é uma razão muito forte para a adoção da linguagem em um ambiente empresarial, onde a obtenção de licenças individuais e coletivas de outros softwares comerciais pode incidir um alto custo financeiro.

1.3 Usos do R

O R é uma linguagem de programação completa e qualquer problema computacional relacionado a dados pode ser resolvido com base nela. Dada a adoção do R por diferentes áreas de conhecimento, a lista de possibilidades é extensa. Para o caso de Finanças e Economia, destaco abaixo as possíveis utilizações do programa:

- Substituir e melhorar tarefas intensivas e repetitivas dentro de ambientes corporativos, geralmente realizadas em planilhas eletrônicas;
- Criação de relatórios estruturados periódicos com a tecnologia *RMarkdown* e *quarto*. Podes, por exemplo, criar um relatório automatizado que importe dados brutos e analise as vendas as vendas na empresa no mês.
- Desenvolvimento de rotinas para administrar portfolios de investimentos e executar ordens financeiras;
- Criação de ferramentas para controle, avaliação e divulgação de índices econômicos sobre um país ou região;
- Execução de diversas possibilidades de pesquisa científica através da estimação de modelos econométricos e testes de hipóteses;
- Criação e manutenção de *websites* dinâmicos ou estáticos através do pacotes `shiny`, `blogdown` ou `distill`;

Além dos usos destacados anteriormente, o acesso público a pacotes desenvolvidos por usuários expande ainda mais essas funcionalidades. O site da CRAN (*Comprehensive R Archive Network*)[1] oferece um *Task Views* do software para o tópico de Finanças[2] e Econometria[3]. Nos links é possível encontrar os principais pacotes disponíveis para cada tema. Isso inclui a importação de dados financeiros da internet, a estimação de um modelo econométrico específico, cálculos de diferentes estimativas de risco, entre várias outras possibilidades. A leitura dessa página e o conhecimento desses pacotes são essenciais para aqueles que pretendem trabalhar com Finanças e Economia. Vale destacar, porém, que essa lista é moderada e apresenta apenas os principais itens. A lista completa de pacotes é muito maior do que o apresentado no *Task Views*.

[1] https://cran.r-project.org/web/views
[2] https://cran.r-project.org/web/views/Finance.html
[3] https://cran.r-project.org/web/views/Econometrics.html

1.4 Como Instalar o R

O R é instalado no seu sistema operacional como qualquer outro programa. A maneira mais direta e funcional de instalá-lo é ir ao website do R em e clicar no *link CRAN* do painel *Download*, conforme mostrado na animação a seguir.

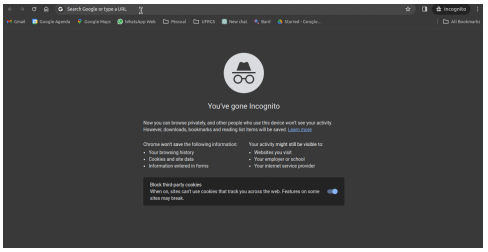


Figura 1.1: Instalando o R no Windows

```
knitr::include_graphics('resources/figs/cran-webshot-01.png')
```

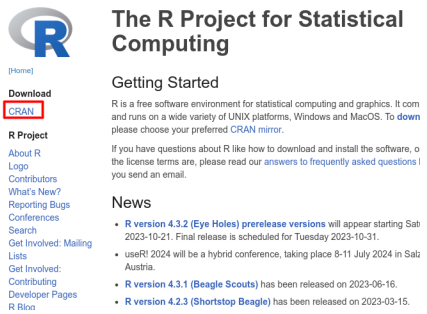


Figura 1.2: Página inicial para o download do R

A próxima tela apresenta a escolha do espelho para baixar os arquivos de instalação. O repositório do CRAN é espelhado em diversas partes do mundo, permitindo acesso rápido para os usuários. Para a grande maioria dos leitores deste livro, essa localidade deve ser o Brasil. Portanto,

você pode escolher um dos links da instituição mais próxima, tal como o da UFPR (Universidade Federal do Paraná). Em caso de dúvida, escolha o repositório do RStudio *0-Cloud* (veja Figura **??**), o qual automaticamente direciona para o local mais próximo.

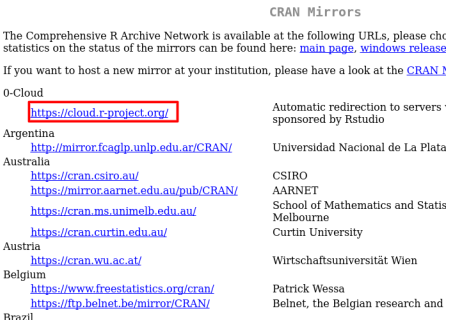


Figura 1.3: Tela com a escolha do espelho para o download

O próximo passo é selecionar o sistema operacional do computador. Devido à maior popularidade da plataforma *Windows*, a partir de agora daremos enfoque à instalação do R nesse sistema. As instruções de instalação nos demais sistemas operacionais podem ser facilmente encontradas na internet. Destaca-se que, independente da plataforma, o modo de uso do R é o mesmo. Existem, porém, algumas exceções, principalmente quando o R interage com o sistema de arquivos. Essas exceções serão destacadas no decorrer do livro. Assim, mesmo que você esteja utilizando Linux ou MacOS, poderá tirar proveito do material aqui apresentado.

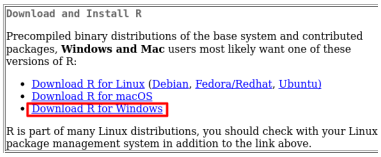


Figura 1.4: Tela com a escolha do sistema operacional

Após clicar no link *Download R for Windows*, a próxima tela irá mostrar as seguintes opções de *download*: *base*, *contrib*, *old.contrib* e *RTools*. Dentre as opções de *download*, a primeira (*base*) deve ser selecionada. O *link* acessa a instalação básica do R para *Windows*. O link *contrib* e *old.contrib* acessa os pacotes/módulos disponíveis para o R. Não precisas acessar estes últimos links, existe uma maneira muito mais fácil de instalar pacotes, como veremos em seguida.

O último link, *RTools*, serve para instalar dependências necessárias no caso do usuário desenvolver e distribuir os seus próprios pacotes de R. Este não

é uma instalação necessária para usuários iniciantes. Porém, saiba que alguns pacotes externos ao CRAN podem exigir a instalação do *RTools* para compilação de código. Minha sugestão é que já **instale o Rtools** e assim evite qualquer problema futuro.

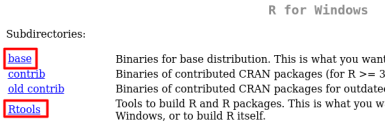


Figura 1.5: Tela com opções de instalação

Após clicar no link *base*, a próxima tela mostrará o link para o *download* do arquivo de instalação do R no *Windows*. Após baixar o arquivo, abra-o e siga os passos da tela de instalação do R. Escolha a língua inglesa em todas etapas do processo. O uso da língua inglessa não é acidental. Este é a melhor forma, mesmo para iniciantes, de se aprender a usar o R. É possível instalar uma versão em português porém isso limita o potencial da ferramenta. Caso não for fluente em inglês, não se preocupe, o vocabulário necessário é básico. Neste momento, nenhuma outra configuração especial é necessária. Sugiro manter todas as escolhas padrão selecionadas e simplesmente ir aceitando as telas de diálogo. Após a instalação do R, partimos para a instalação do RStudio.

```
R_ver <- R.version

major <- R_ver$major
minor <- R_ver$minor

R_version_str <- paste0(major, ".", minor)
```

! Importante

A cada quatro meses uma nova versão do R é lançada, corrigindo *bugs* e implementando novas soluções. Temos dois tipos principais de versões, *major* e *minor*. Por exemplo, na data de compilação do livro, 10/01/2024, a última versão disponível do R é 4.3.2. O primeiro dígito (4) indica a versão *major* e todos os demais são do tipo *minor*. Geralmente, as mudanças *minor* são bem específicas e, possivelmente, terão pouco impacto no seu trabalho.
Porém, mudanças do tipo *major* refletem totalmente no ecossistema de pacotes do R. Toda vez que instalar uma nova versão *major* do R,

terás que reinstalar todos os pacotes utilizados. O problema é que não é incomum problemas de incompatibilidade de pacotes com a nova versão.
Minha dica é: toda vez que uma nova versão *major* do R sair, espere alguns meses antes de instalar na sua máquina. Assim, o autores dos pacotes terão mais tempo para atualizar os seus códigos, minimizando a possibilidade de problemas de compatibilidade.

1.5 Instalando o RStudio

A instalação do R inclui a sua própria interface gráfica, um programa que facilita a edição e execução de nossos *scripts*. Essa, porém, possui várias limitações. O RStudio é um *software* que torna o uso e o visual do R muito mais prático e eficiente. Uma forma de entender essa relação é com uma analogia com carros. Enquanto o R é o motor da linguagem de programação, o RStudio é a carroceria e o painel de instrumentos. Além de apresentar um visual mais atrativo, o RStudio também é acrescido de várias funcionalidades que facilitam a vida do usuário, possibilitando a construção de projetos e pacotes do próprio R, a criação de documentos dinâmicos (*Sweave/knitr*) e a interface com edição de textos em *LaTeX*, entre várias outras. Assim como o R, **o RStudio também é gratuito** e pode ser utilizado no ambiente empresarial.

A instalação do RStudio é mais simples do que a do R. Os arquivos estão disponíveis no endereço disponibilizado no site da empresa Posit Posit. Após acessar a página, clique em *Download RStudio* e depois em *Download Rstudio Desktop*. Logo após, basta selecionar o arquivo relativo ao sistema operacional em que você irá trabalhar. Provavelmente, essa opção será *Windows Vista/7/8/10*. Note que, assim como o R, o RStudio também está disponível para diferentes plataformas.

Destaco que o uso do RStudio não é essencial para desenvolver programas no R. Outros softwares de interface estão disponíveis e podem ser utilizados. Porém, dada minha experiência atual, o RStudio é o programa de interface que oferece a maior variedade de funcionalidades para essa linguagem, além de ser amplamente utilizado, o que justifica a sua escolha. Como uma alternativa ao RStudio, sugiro o vscode, o qual possui uma ótima integração com o R.

1.6 Recursos na Internet

A comunidade R é viva e envolvente. Na internet é possível encontrar uma diversidade de material sobre o uso do R. Diversos usuários, assim como o próprio autor do livro, publicam material sobre o uso R em seus blogs. Isso inclui anúncios de pacotes, publicações sobre análise de dados na vida real, curiosidades, novidades e tutoriais. **R-Bloggers** é um site internacional que agrega esses blogs em um único local, tornando mais fácil para qualquer um acessar e participar. O conteúdo do R-Bloggers, porém, é todo em inglês.

Recentemente, uma lista de blogs locais sobre o R está compilada e organizada por Marcos Vital no Github. Eu recomendo a inscrição no feed do R-Bloggers, além dos blogs nacionais. Não só você será informado sobre o que está acontecendo no universo do R, mas também aprenderá muito lendo artigos e os códigos de outros usuários.

Aprender e usar R pode ser uma experiência social. Várias conferências e grupos de usuários estão disponíveis em muitos países, incluindo o Brasil. O grupo *R Brasil - Programadores* no Facebook é bastante ativo, com um grande número de participantes. Recomendo fortemente a inscrição neste grupo e o acompanhamento das discussões relacionadas ao uso do R. Diversas conferências locais sobre o R são divulgadas nesse grupo.

1.7 Organização e Material do Livro

Este livro tem uma abordagem prática no uso do R e será acompanhado por uma série de códigos que irão exemplificar e mostrar para o leitor as funcionalidades do programa. Para tirar o máximo de proveito do material, sugiro que você primeiro busque entender o código mostrado e, somente então, tente utilizá-lo em seu próprio computador. O índice remissivo disponibilizado no final do livro impresso serve como uma mapa de uso das funções. Toda vez que uma função é chamada no livro, um registro do número da página é criado no índice remissivo. Esse indica, também, o pacote que a função pertence. Podes utilizar este mapa para localizar o uso de qualquer função ou pacote no decorrer do livro.

Sugiro também o uso da versão web do livro[4], a qual permite que os código de exemplo sejam copiados direto para a sua sessão do R. Assim, perderás menos tempo digitando código, sobrando tempo para o estudo.

[4] https://www.msperlin.com/introR/

Aprender a programar em uma nova linguagem é como aprender uma língua estrangeira: o uso no dia-a-dia é de extrema importância para criar fluência. Sempre que possível, teste o código no seu computador e *brinque* com o mesmo, modificando os exemplos dados e verificando o efeito das modificações nas saídas do programa. Procure sempre entender como a rotina estudada pode ajudar na solução de um problema seu. Cada capítulo apresenta no seu final uma lista de exercícios, incluindo questões do tipo desafio. Podes testar as suas habilidades de programação resolvendo as atividades propostas. Vale relembrar que todo o código deste livro está disponibilizado na internet. Não precisas, portanto, escrever o código diretamente do livro. Podes copiar e colar do código fonte disponibilizado no site.

No decorrer da obra, toda demonstração de código terá duas partes: o código em si e sua saída do R. Essa saída nada mais é do que o resultado dos comandos na tela do programa. Todas as entradas e saídas de código serão sinalizadas no texto com um formato especial. Veja o exemplo a seguir:

```
# create a list
L <- list(var1 = 'abc', var2 = 1:5)

# print to prompt
print(L)
```

```
R> $var1
R> [1] "abc"
R>
R> $var2
R> [1] 1 2 3 4 5
```

No caso anterior, os textos `L <- list(var1 = 'abc', var2 = 1:5)` e `print(L)` são os códigos de entrada. A saída do programa é a apresentação na tela dos elementos de `x`, com o símbolo antecessor `R>`. Por enquanto não se preocupe em entender e reproduzir o código utilizado acima. Iremos tratar disso no próximo capítulo.

Note que faço uso da língua inglesa no código, tanto para a nomeação de objetos quanto para os comentários. Isso não é acidental. O uso da língua inglesa facilita o desenvolvimento de código ao evitar caracteres latinos, além de ser uma das línguas mais utilizadas no mundo. Portanto, é importante já ir se acostumando com esse formato. O vocabulário necessário,

porém, é limitado. De forma alguma precisarás ter fluência em inglês para entender o código.

O código também pode ser espacialmente organizado usando novas linhas. Esse é um procedimento comum em torno de argumentos de funções. O próximo pedaço de código é equivalente ao anterior, e executará exatamente da mesma maneira. Observe como usei uma nova linha para alinhar verticalmente os argumentos da função `list`. Você verá em breve que, ao longo do livro, esse tipo de alinhamento vertical é constantemente usado em códigos longos. Afinal, o código tem que necessariamente caber na página do livro impresso.

```
# create a list
L <- list(var1 = 'abc',
          var2 = 1:5)

# print to prompt
print(L)
```

```
R> $var1
R> [1] "abc"
R>
R> $var2
R> [1] 1 2 3 4 5
```

O código também segue uma estrutura bem definida. Uma das decisões a ser feita na escrita de códigos de computação é a forma de nomear os objetos e como lidar com a estrutura do texto do código em geral. É recomendável seguir um padrão limpo de código, de forma que o mesmo seja fácil de ser mantido ao longo do tempo e de ser entendido por outros usuários. Para este livro, foi utilizado uma mistura de escolhas pessoais do autor com o estilo de código sugerido pelo Google. O usuário, porém, é livre para escolher a estrutura que achar mais eficiente. Voltaremos a discutir estrutura de código no capítulo @ref(otimizacao).

1.8 Animações de código

Esta edição do livro inclui animações de código no formato *gif*, as quais devem ajudar a memorizar e visualizar as diferentes operações na plataforma. Para os leitores do livro impresso, obviamente não existe maneira de incluir animações em paper. Como alternativa, cada animação inclui um

QRCode que irá direcionar o leitor para página na web com a animação em questão. A imagem a esquera é a primeira tela de cada animação. Veja o exemplo abaixo na Figura 1.6:

Figura 1.6: Exemplo de animação

Para utilizar, abra seu celular e aponte a câmera para o QRCode da página. Após isso, basta clicar no link que aparecer na tela do celular.

1.9 Exercícios

CAPÍTULO 2

COMO O R FUNCIONA?

A maior dificuldade que um usuário iniciante possui ao começar a desenvolver rotinas com o R é a forma de trabalho. A nossa interação com computadores foi simplificada ao longo dos anos e atualmente estamos confortáveis com o formato de interação do tipo *aponte e clique*. Particularmente para o caso de análise de dados, o uso de planilhas eletrônicas do tipo Excel é provavelmente o primeiro contato de um estudante com plataformas de análise.

O formato *aponte e clique*, muito utilizado em planilhas, permite que o usuário aponte o mouse para um determinado local da tela, clique em um botão e realize uma determinada operação. Uma série de passos nesse sentido permite a execução de tarefas complexas no computador. Mas não se engane, essa forma de interação no formato *aponte e clique* é apenas uma camada por cima do que realmente acontece no computador. Por trás de todo *clique* existe um comando sendo executado, seja na abertura de um arquivo *pdf*, direcionamento do *browser* para uma página na internet ou qualquer outra operação cotidiana.

Enquanto esse formato de interação visual e motora tem seus benefícios ao facilitar e popularizar o uso de computadores, é pouco flexível e eficaz quando se trabalha com procedimentos computacionais. Alternativamente, ao conhecer os possíveis comandos disponíveis ao usuário, é possível criar um arquivo contendo instruções em sequência e, futuramente,

simplesmente pedir que o computador **execute** esse arquivo com os nossos procedimentos. **Uma rotina de computador é nada mais do que um texto que instrui, de forma clara e sequencial, o que o computador deve fazer**. Investe-se certo tempo para a criação do programa, porém, no futuro, esse irá executar sempre da mesma maneira o procedimento gravado. No médio e longo prazo, existe um ganho significativo de tempo entre o uso de uma rotina do computador e uma interface do tipo *aponte e clique*.

Além disso, o **risco de erro humano na execução do procedimento é quase nulo**, pois os comandos e a sua sequência estão registrados no arquivo texto e irão ser executados sempre da mesma maneira. Da mesma forma, esse aglomerado de comandos pode ser compartilhado com outras pessoas, as quais podem replicar os resultados em seus computadores. Essa é uma das grandes razões que justificam a popularização de programação na realização de pesquisa em dados. Todos os procedimentos executados podem ser replicados pelo uso de um *script*.

O R é uma plataforma de programação e o primeiro choque para um novo usuário é o predomínio no uso de código sobre operações com o Mouse. O R e o RStudio possuem algumas funcionalidades através do *mouse*, porém a sua capacidade é otimizada quando os utilizamos via inserção de comandos específicos. Quando um grupo de comandos é realizado de uma maneira inteligente, temos um *script* do R que deve preferencialmente produzir algo importante para nós no final de sua execução.

O R também possibilita a exportação de arquivos, tal como figuras a serem inseridas em um relatório técnico ou informações em um arquivo texto. De fato, o próprio relatório técnico pode ser dinamicamente criado dentro do R através da tecnologia *RMarkdown* e *Quarto*. Por exemplo, este livro que estás lendo foi escrito utilizando a tecnologia Quarto. O conteúdo do livro é compilado com a execução dos códigos e as suas saídas são registradas em texto. Todas as figuras e os dados do livro podem ser atualizados com a execução de um simples comando.

Provavelmente, o produto final de trabalhar com R e RStudio será um script que produz elementos para um relatório de dados. Um bom exemplo de um código simples e polido pode ser encontrado neste link. Abra-o e você verá o conteúdo de um arquivo com extensão *.R* que fará o download dos preços das ações de duas empresas e criará um gráfico e uma tabela. Ao terminar de ler o livro, você irá entender o que está acontecendo no código e como ele realiza o trabalho. Melhor ainda, você poderá melhorá-lo com novas funcionalidades e novas saídas. Caso esteja curioso em ver o script rodar, faça o seguinte: 1) instale R e RStudio no

computador, 2) copie o conteúdo de texto do link para um novo script ("File" -> "New File" -> "R Script"), 3) salve-o com um nome qualquer e, finalizando, 4) pressione `control + shift + enter` para executar o *script* inteiro.

2.1 Objetos e Funções

No R, **tudo é um objeto, e cada tipo de objeto tem suas propriedades**. Por exemplo, o valor de um índice de inflação ao longo do tempo – em vários meses e anos – pode ser representado como um objeto do tipo vetor numérico. As datas em si, no formato YYYY-MM-DD (ano-mês-dia), podem ser representadas como texto (*character*) ou a própria classe `Date`. Por fim, podemos representar conjuntamente os dados de inflação e as datas armazenando-os em um objeto único do tipo *dataframe*, o qual nada mais é do que uma **tabela** com linhas e colunas. Todos esses objetos fazem parte do ecossistema do R e é através da manipulação destes que tiramos o máximo proveito do *software*.

Os principais tipos de objetos do R são:

numéricos (*numeric*) representam números e medidas
caraterer (*character*) representam texto
fatores (*factors*) representam grupos dentros dos dados, tipo "Casado/Solteiro", "Destro/Canhoto"
dataframes ou *tibbles* representam tabelas, as quais podem conter dados numéricos, caracteres e fatores em um único objeto.

Enquanto representamos informações do mundo real com as diferentes classes no R, um tipo especial de objeto é a **função**, a qual representa um procedimento preestabelecido que está disponível para o usuário. O R possui uma grande quantidade de funções, as quais possibilitam que o usuário realize uma vasta gama de procedimentos. Por exemplo, os comandos básicos do R, não incluindo demais pacotes, somam um total de 1268 funções. Com base neles e outros iremos importar dados, calcular médias, testar hipóteses, limpar dados, e muito mais.

Cada função possui um próprio nome. Por exemplo, a função **sort()** é um procedimento que ordena valores utilizados como *input*. Caso quiséssemos ordenear os valores no vetor numérico *[2, 1, 3, 0]*, basta inserir no *prompt* o seguinte comando e apertar *enter*:

```
sort(c(2, 1, 3, 0), decreasing = TRUE)
```

```
R> [1] 3 2 1 0
```

O comando `c(2, 1, 3, 0)` combina os valores em um vetor (maiores detalhes sobre comando `c` serão dados em seção futura). Observe que a função **sort()** é utilizada com parênteses de início e fim. Esses parênteses servem para destacar as entradas (*inputs*), isto é, as informações enviadas para a função produzir alguma coisa. Observe que cada entrada (ou opção) da função é separada por uma vírgula, tal como em `MinhaFuncao(entrada01, entrada02, entrada03, ...)`. No caso do código anterior, note que usamos a opção `decreasing = TRUE`. Essa é uma instrução específica para a função **sort()** ordenar de forma decrescente os elementos do vetor de entrada. Veja a diferença:

```
sort(c(2, 1, 3, 0), decreasing = FALSE)
```

```
R> [1] 0 1 2 3
```

O uso de funções está no coração do R e iremos dedicar grande parte do livro a elas. Por enquanto, essa breve introdução já serve o seu propósito. O principal é entender que uma função usa suas entradas para produzir algo de volta. Nos próximos capítulos iremos utilizar funções já existentes para as mais diferentes finalidades: baixar dados da internet, ler arquivos, realizar testes estatísticos e muito mais. No capítulo @ref(programacao) iremos tratar deste assunto com maior profundidade, incluindo a forma de escrevermos nossas próprias funções.

2.2 Criando Objetos Simples

Um dos comandos mais básicos no R é a definição de objetos. Como foi mostrado nas seções anteriores, pode-se definir um objeto com o uso do comando <-, o qual, para o português, é traduzido para o verbo *defina* (*assign* em inglês). Considere o seguinte código:

```
# set x
my_x <- 123

# set x, y and z in one line
```

```
5 my_x <- 1 ; my_y <- 2; my_z <- 3
```

Lê-se esse código como *x é definido como 123*. A direção da seta define onde o valor será armazenado. Por exemplo, utilizar `123 -> my_x` também funcionaria, apesar de ser uma sintaxe pouco utilizada ou recomendada. Note que também é possível escrever diversos comandos na mesma linha com o uso da semi-vírgula (;).

> **Importante**
>
> O uso do símbolo <- para a definição de objetos é específico do R. Na época da concepção da linguagem *S*, de onde o R foi baseado, existiam teclados com uma tecla específica que definia diretamente o símbolo de seta. Teclados contemporâneos, porém, não possuem mais esta configuração. Uma alternativa é utilizar o atalho para o símbolo, o qual, no Windows, é definido por `alt + -`.

É possível também usar o símbolo = para definir objetos assim como o <-. Saliento que esta é prática comum em outras linguagens de programação. Porém, no ecosistema do R, a utilização do = com esse fim específico não é recomendada. O símbolo de igualdade tem o seu uso especial e resguardado na definição de argumentos de uma função tal como `sort(x = 1:10, decreasing = TRUE)`.

> **Importante**
>
> A nomeação dos objetos criados no R é importante. Tirando alguns casos específicos, o usuário pode nomear os objetos como quiser. Essa liberdade, porém, pode ser um problema. É desejável sempre dar nomes curtos que façam sentido ao conteúdo do objeto e que sejam simples de entender. Isso facilita o entendimento do código por outros usuários e faz parte das normas sugeridas para a estruturação do código do Google.

O R executa o código procurando objetos e funções disponíveis no seu ambiente de trabalho (*enviromnent*). Se tentarmos acessar um objeto que não existe, o R irá retornar uma mensagem de erro:

```
1 print(z)
```

```
R> Error in print(z): object 'z' not found
```

Isso ocorre pois o objeto `z` não existe na sessão atual do R. Se criarmos uma variável `z` como `z <- 123` e repetirmos o comando `print(z)`, não teremos a mesma mensagem de erro.

Um ponto importante aqui é a definição de objetos de classes diferentes com o uso de símbolos específicos. O uso de aspas duplas (`" "`) ou simples (`' '`) define objetos da classe texto enquanto números são definidos pelo próprio valor. Conforme será mostrado, cada objeto no R tem uma classe e cada classe tem um comportamento diferente. Portanto, objetos criados com o uso de aspas pertencem à classe *character*. Podemos confirmar isso via código:

```
# set vars
x <- 1
y <- '1'

# display classes
class(x)
```

```
R> [1] "numeric"
```

```
class(y)
```

```
R> [1] "character"
```

As saídas anteriores mostram que a variável `x` é do tipo numérico, enquanto a variável `y` é do tipo texto (*character*). Ambas fazem parte das classes básicas de objetos no R. Por enquanto, este é o mínimo que deves saber para avançar nos próximos capítulos. Iremos estudar este assunto mais profundamente no capítulo @ref(classes-basicas).

2.3 Criando Vetores

Nos exemplos anteriores criamos objetos simples tal como `x <- 1` e `x <- 'abc'`. Enquanto isso é suficiente para demonstrar os comandos básicos do R, na prática tais comandos são bastante limitados, uma vez que um problema real de análise de dados certamente irá ter um maior volume de informações do mundo real.

Um dos procedimentos mais utilizados no R é a criação de vetores atômicos. Esses são objetos que guardam uma série de elementos. Todos os

elementos de um vetor atômico devem possuir a mesma classe, o que justifica a sua propriedade *atômica*. Um exemplo seria representar no R uma série de preços diários de uma ação. Tal série possui vários valores numéricos que formam um vetor da classe numérica.

Vetores atômicos são criados no R através do uso do comando c()** **, o qual é oriundo do verbo em inglês *combine*. Por exemplo, caso eu quisesse *combinar* os valores 1, 2 e 3 em um vetor, eu poderia fazê-lo através do seguinte comando:

```
# set vector
x <- c(1, 2, 3)

# print it
print(x)
```

```
R> [1] 1 2 3
```

Esse comando funciona da mesma maneira para qualquer número de elementos. Caso necessário, poderíamos criar um vetor com mais elementos simplesmente adicionando valores após o `3`, tal como em `x <- c(1, 2, 3, 4, 5)`.

O uso do comando **c()** não é exclusivo para vetores numéricos. Por exemplo, poderíamos criar um vetor de outra classe de dados, tal como *character*:

```
y <- c('text 1', 'text 2', 'text 3', 'text 4')
print(y)
```

```
R> [1] "text 1" "text 2" "text 3" "text 4"
```

A única restrição no uso do comando **c()** é que todos os itens do vetor **tenham a mesma classe**. Se inserirmos dados de classes diferentes, o R irá tentar transformar os itens para a mesma classe seguindo uma lógica própria, onde a classe mais complexa sempre tem preferência. Caso ele não consiga transformar todos os elementos para uma classe só, uma mensagem de erro será retornada. Observe no próximo exemplo como os valores numéricos no primeiro e segundo elemento de `x` são transformados para a classe de caracteres.

```r
# numeric class
x <- c(1, 2)
class(x)
```

```
R> [1] "numeric"
```

```r
# character class
x <- c(1, 2, '3')
class(x)
```

```
R> [1] "character"
```

Outra utilização do comando **c()** é a combinação de vetores. De fato, isto é exatamente o que fizemos ao executar o código `c(1, 2, 3)`. Neste caso, cada vetor possuía um elemento. Podemos realizar o mesmo com vetores maiores. Veja a seguir:

```r
# set x and y
x <- c(1, 2, 3)
y <- c(4, 5)

# print concatenation between x and y
print(c(x, y))
```

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

Portanto, o comando **c()** possui duas funções principais: criar e combinar vetores.

2.4 Conhecendo os Objetos Criados

Após a execução de diversos comandos no editor ou *prompt*, é desejável saber quais são os objetos criados pelo código. É possível descobrir essa informação simplesmente olhando para o lado direito superior do RStudio, na aba da área de trabalho. Porém, existe um comando que sinaliza a mesma informação no *prompt*. Com o fim de saber quais são as variáveis atualmente disponíveis na memória do R, pode-se utilizar o comando **ls()** . Observe o exemplo a seguir:

```
# set vars
x <- 1
y <- 2
z <- 3

# show current objects
ls()
```

```
R> [1] "x" "y" "z"
```

Os objetos `x`, `y` e `z` foram criados e estavam disponíveis no ambiente de trabalho atual, juntamente com outros objetos. Para descobrir os valores dos mesmos, basta digitar os nomes dos objetos e apertar `enter` no *prompt*:

```
x
```

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

```
y
```

```
R> [1] 2
```

```
z
```

```
R> [1] 3
```

Digitar o nome do objeto na tela tem o mesmo resultado que utilizar a função **print()** . De fato, ao executar o nome de uma variável, internamente o R passa esse objeto para a função **print()** .

No R, conforme já mostrado, todos os objetos pertencem a alguma classe. Para descobrir a classe de um objeto, basta utilizar a função **class()** . Observe no exemplo a seguir que `x` é um objeto da classe numérica e `y` é um objeto da classe de texto (*character*).

```
# set vars
x <- 1
y <- 'a'

# check classes
class(x)
```

```
R> [1] "numeric"
```

```
class(y)
```

```
R> [1] "character"
```

Outra maneira de conhecer melhor um objeto é verificar a sua representação em texto. Todo objeto no R possui uma representação textual e a verificação desta é realizada através da função **str()** :

```
# print textual representation of a vector
x <- 1:10
print(str(x))
```

```
R>  int [1:10] 1 2 3 4 5 6 7 8 9 10
R> NULL
```

Essa função é particularmente útil quando se está tentando entender os detalhes de um objeto mais complexo, tal como uma tabela. A utilidade da representação textual é que nela aparece o tamanho do objeto e suas classes internas. Nesse caso, o objeto `x` é da classe *integer* e possui dez elementos.

2.5 Mostrando e Formatando Informações na Tela

Como já vimos, é possível mostrar o valor de uma variável na tela de duas formas, digitando o nome dela no *prompt* ou então utilizando a função **print()** . Explicando melhor, a função **print()** é voltada para a apresentação de objetos e pode ser customizada. Por exemplo, caso tivéssemos um objeto de classe chamada `MyTable` que representasse um objeto tabular, poderíamos criar uma função chamada `print.MyTable` que irá mostrar uma tabela na tela com um formato especial tal como número de linhas, nomes das colunas, etc. A função **print()** , portanto, pode ser customizada para cada classe de objeto.

Porém, existem outras funções específicas para apresentar texto (e não objetos) no *prompt*. A principal delas é `message`. Essa toma como *input* um texto, processa-o para símbolos específicos e o apresenta na tela. Essa função é muito mais poderosa e personalizável do que **print()** .

Por exemplo, caso quiséssemos mostrar na tela o texto `'O valor de x é igual a 2'`, poderíamos fazê-lo da seguinte forma:

```
# set var
x <- 2

# print with message()
message('The value of x is', x)
```

```
R> The value of x is2
```

Função `message` também funciona para vetores:

```
# set vec
x <- 2:5

# print with message()
message('The values in x are: ', x)
```

```
R> The values in x are: 2345
```

A customização da saída da tela é possível através de comandos específicos. Por exemplo, se quiséssemos quebrar a linha da tela, poderíamos fazê-lo através do uso do caractere reservado `\n`:

```
# set char
my_text <- 'First line,\nSecond Line,\nThird Line'

# print with new lines
message(my_text)
```

```
R> First line,
R> Second Line,
R> Third Line
```

Observe que o uso do **print()** não resultaria no mesmo efeito, uma vez que esse comando apresenta o texto como ele é, sem processar para efeitos específicos:

```
print(my_text)
```

```
R> [1] "First line,\nSecond Line,\nThird Line"
```

Outro exemplo no uso de comandos específicos para texto é adicionar um espaçamento *tab* no texto apresentado com o símbolo `\t`. Veja a seguir:

```
# set char with \t
my_text_1 <- 'A and B'
my_text_2 <- '\tA and B'
my_text_3 <- '\t\tA and B'

# print with message()
message(my_text_1)
```

```
R> A and B
```

```
message(my_text_2)
```

```
R>  A and B
```

```
message(my_text_3)
```

```
R>      A and B
```

Vale destacar que, na grande maioria dos casos de pesquisa, será necessário apenas o uso de `\n` para formatar textos de saída. Outras maneiras de manipular a saída de texto no *prompt* com base em símbolos específicos são encontradas no manual oficial do R.

Parte do processo de apresentação de texto na tela é a customização do mesmo. Para isto, existem duas funções muito úteis: **paste()** e **format()** .

A função **paste()** *cola* uma série de caracteres juntos. É uma função muito útil, a qual será utilizada intensamente para o resto dos exemplos deste livro. Observe o código a seguir:

```
# set chars
my_text_1 <- 'I am a text'
my_text_2 <- 'very beautiful'
my_text_3 <- 'and informative.'

# using paste and message
message(paste(my_text_1, my_text_2, my_text_3))
```

```
R> I am a text very beautiful and informative.
```

O resultado anterior não está muito longe do que fizemos no exemplo com a função **print()** . Note, porém, que a função **paste()** adiciona um espaço entre cada texto. Caso não quiséssemos esse espaço, poderíamos usar a função **paste0()** :

```
# using paste0
message(paste0(my_text_1, my_text_2, my_text_3))
```

```
R> I am a textvery beautifuland informative.
```

> **Importante**
>
> Uma alternativa a função `message` é `cat` (*concatenate and print*). Não é incomum encontrarmos códigos onde mensagens para o usuário são transmitidas via `cat` e não `message`. Como regra, dê preferência a `message` pois esta é mais fácil de controlar. Por exemplo, caso o usuário quiser silenciar uma função, omitindo todas saídas da tela, bastaria usar o comando `suppressMessages`.

Outra possibilidade muito útil no uso do **paste()** é modificar o texto entre a junção dos itens a serem colados. Por exemplo, caso quiséssemos adicionar uma vírgula e espaço (,) entre cada item, poderíamos fazer isso através do uso do argumento `sep`, como a seguir:

```
# using custom separator
message(paste(my_text_1, my_text_2, my_text_3, sep = ', '))
```

```
R> I am a text, very beautiful, and informative.
```

Caso tivéssemos um vetor atômico com os elementos da frase em um objeto apenas, poderíamos atingir o mesmo resultado utilizando **paste()** o argumento *collapse*:

```
# using paste with collapse argument
my_text <-c('Eu sou um texto', 'muito bonito', 'e charmoso.')
message(paste(my_text, collapse = ', '))
```

```
R> Eu sou um texto, muito bonito, e charmoso.
```

Prosseguindo, o comando **format()** é utilizado para formatar números e datas. É especialmente útil quando formos montar tabelas e buscarmos apresentar os números de uma maneira visualmente atraente. Por definição, o R apresenta uma série de dígitos após a vírgula:

```
# message without formatting
message(1/3)
```

```
R> 0.333333333333333
```

Caso quiséssemos apenas dois dígitos aparecendo na tela, utilizaríamos o seguinte código:

```
# message with format and two digits
message(format(1/3, digits=2))
```

```
R> 0.33
```

Tal como, também é possível mudar o símbolo de decimal:

```
# message with format and two digits
message(format(1/3, decimal.mark = ','))
```

```
R> 0,3333333
```

Tal flexibilidade é muito útil quando devemos reportar resultados respeitando algum formato local tal como o Brasileiro.

Uma alternativa recente e muito interessante para o comando **paste()** é **stringr**::**str_c()** e **stringr**::**str_glue()** . Enquanto a primeira é quase idêntica a **paste0()** , a segunda tem uma maneira pecular de juntar objetos. Veja um exemplo a seguir:

```
library(stringr)

# define some vars
my_name <- 'Pedro'
my_age <- 23

# using base::paste0
my_str_1 <- paste0('My name is ', my_name, ' and my age is ', my_age)

```

```
# using stringr::str_c
my_str_2 <- str_c('My name is ', my_name, ' and my age is ', my_age)

# using stringr::str_glue
my_str_3 <- str_glue('My name is {my_name} and my age is {my_age}')

identical(my_str_1, my_str_2)
```

```
R> [1] TRUE
```

```
identical(my_str_1, my_str_3)
```

```
R> [1] FALSE
```

```
identical(my_str_2, my_str_3)
```

```
R> [1] FALSE
```

Como vemos, temos três alternativas para o mesmo resultado final. Note que `str_glue` usa de chaves para definir as variáveis dentro do próprio texto. Esse é um formato muito interessante e prático para concatenar textos em um único objeto.

2.6 Conhecendo o Tamanho dos Objetos

Na prática de programação com o R, é muito importante saber o tamanho das variáveis que estão sendo utilizadas. Isso serve não somente para auxiliar o usuário na verificação de possíveis erros do código, mas também para saber o tamanho necessário em certos procedimentos de iteração tal como *loops*, os quais serão tratados em capítulo futuro.

No R, o tamanho do objeto pode ser verificado com o uso de quatro principais funções: **length()** , **nrow()** , **ncol()** e **dim()** .

A função **length()** é destinada a objetos com uma única dimensão, tal como vetores atômicos:

```
# set x
x <- c(2, 3, 3, 4, 2, 1)

```

```
# get length x
n <- length(x)

# display message
message(paste('The length of x is', n))
```

```
R> The length of x is 6
```

Para objetos com mais de uma dimensão, por exemplo matrizes e data-frames, utilizam-se as funções **nrow()** , **ncol()** e **dim()** para descobrir o número de linhas (primeira dimensão) e o número de colunas (segunda dimensão). Veja a diferença a seguir.

```
# set matrix and print it
x <- matrix(1:20, nrow = 4, ncol = 5)
print(x)
```

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

```
# find number of rows, columns and elements
my_nrow <- nrow(x)
my_ncol <- ncol(x)
my_length <- length(x)

# print message
message(paste('\nThe number of lines in x is ', my_nrow))
```

```
R>
R> The number of lines in x is  4
```

```
message(paste('\nThe number of columns in x is ', my_ncol))
```

```
R>
R> The number of columns in x is  5
```

```
message(paste('\nThe number of elements in x is ', my_length))
```

```
R>
R> The number of elements in x is  20
```

Já a função **dim()** mostra a dimensão do objeto, resultando em um vetor numérico como saída. Essa deve ser utilizada quando o objeto tiver mais de duas dimensões. Na prática, esses casos são raros. Um exemplo para a variável `x` é dado a seguir:

```
1  print(dim(x))
```

```
R> [1] 4 5
```

Para o caso de objetos com mais de duas dimensões, podemos utilizar a função `array` para criá-los e **dim()** para descobrir o seu tamanho:

```
1  # set array with dimension
2  my_array <- array(1:9, dim = c(3,3,3))
3
4  # print it
5  print(my_array)
```

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

```
# print its dimension
print(dim(my_array))
```

```
R> [1] 3 3 3
```

Reforçando, cada objeto no R tem suas propriedades e funções específicas para manipulação.

> **Cuidado**
>
> Uma observação importante aqui é que as funções anteriores não servem para descobrir o número de letras em um texto. Esse é um erro bastante comum. Por exemplo, caso tivéssemos um objeto do tipo texto e usássemos a função **length()** , o resultado seria o seguinte:
>
> ```
> # set char object
> my_char <- 'abcde'
>
> # find its length (and NOT number of characters)
> print(length(my_char))
> ```
>
> ```
> R> [1] 1
> ```
>
> Isso ocorre pois a função **length()** retorna o número de elementos. Nesse caso, `my_char` possui apenas um elemento. Para descobrir o número de caracteres no objeto, utilizamos a função **nchar()** , conforme a seguir:
>
> ```
> # using nchar for number of characters
> print(nchar(my_char))
> ```
>
> ```
> R> [1] 5
> ```

CAPÍTULO 3

INTRODUÇÃO AO RSTUDIO

Após instalar os dois programas, R e RStudio, procure o ícone do RStudio na área de trabalho ou via menu *Iniciar*. Note que a instalação do R inclui um programa de interface e isso muitas vezes gera confusão. Verifique que estás utilizado o *software* correto. A janela resultante deve ser igual a figura Figura 3.1, apresentada a seguir.

Observe que o RStudio automaticamente detectou a instalação do R e inicializou a sua tela no lado esquerdo. Caso não visualizar uma tela parecida ou chegar em uma mensagem de erro indicando que o R não foi encontrado, repita os passos de instalação do capítulo anterior (seção @ref(instalacao)).

> **Alternativa ao RStudio**
>
> Este capítulo inteiro é dedicado ao uso do RStudio, o qual entendo ser um dos melhores ambientes de desenvolvimento para o R. Alternativamente, um grande concorrente é o vscode, o qual oferece uma plataforma agnóstica a linguagem de programação, funcionando para R, Python ou qualquer outra lingaguem. Em projeto que envolve múltiplas linguagens, o vscode é ótimo e fácil de usar.

Como um primeiro exercício, clique em *File*, *New File* e *R Script*. Após, um editor de texto deve aparecer no lado esquerdo da tela do RStudio. É

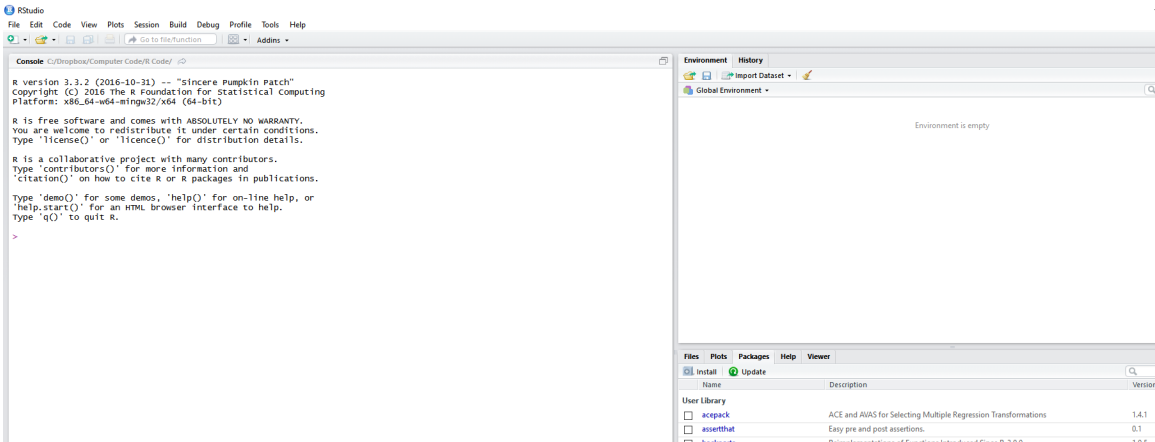


Figura 3.1: A tela do RStudio

nesse editor que iremos inserir os nossos comandos, os quais são executados de cima para baixo, na mesma direção em que normalmente o lemos. Note que essa direção de execução introduz uma dinâmica de recursividade: cada comando depende do comando executado nas linhas anteriores. Após realizar os passos definidos anteriormente, a tela resultante deve ser semelhante à apresentada na figura Figura 3.2.

> **Importante**
>
> Uma sugestão importante aqui é modificar o esquema de cores do RStudio para uma configuração de **tela escura**. Não é somente uma questão estética mas sim de prevenção e melhoria de sua saúde física. Possivelmente irás passar demasiado tempo na frente do computador. Assim, vale a pena modificar as cores da interface para aliviar seus olhos do constante brilho da tela. Dessa forma, conseguirás trabalhar por mais tempo, sem forçar a sua visão. Podes configurar o esquema de cores do RStudio indo na opção *Tools*, *Global Options* e então em *Appearance*. Um esquema de cores escuras que pessoalmente gosto e sugiro é o *Ambience*.

Após os passos anteriores, a tela do RStudio deve estar semelhante a Fi-

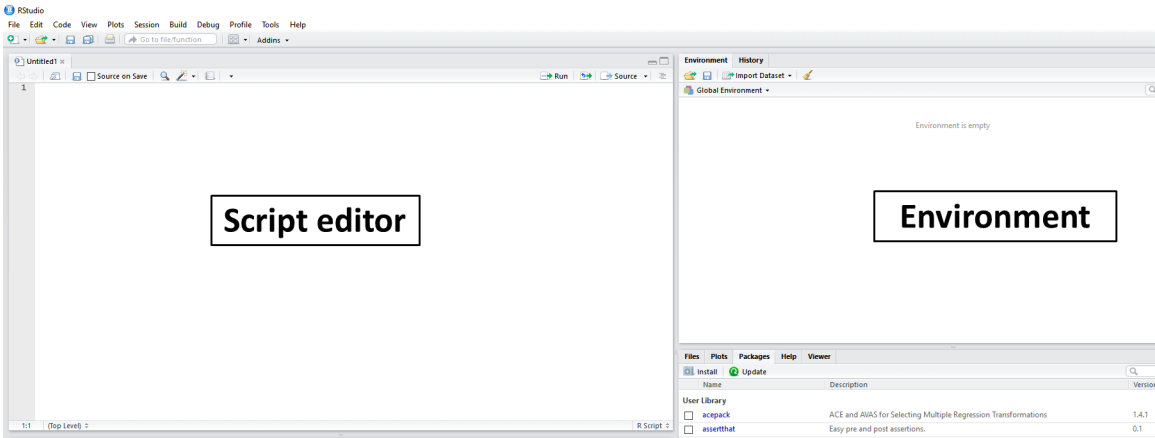


Figura 3.2: Explicando a tela do RStudio

gura Figura 3.2, com os seguintes itens/painéis:

Editor de scripts (*Script editor*): localizado no lado esquerdo e acima da tela. Esse painel é utilizado para escrever código e é onde passaremos a maior parte do tempo.

Console do R (*R prompt*): localizado no lado esquerdo e abaixo do editor de *scripts*. Apresenta o *prompt* do R, o qual também pode ser utilizado para executar comandos. A principal função do *prompt* é testar código e apresentar os resultados dos comandos inseridos no editor de *scripts*.

Área de trabalho (*Environment*): localizado no lado direito e superior da tela. Mostra todos os objetos, incluindo variáveis e funções atualmente disponíveis para o usuário. Observe também a presença do painel *History*, o qual mostra o histórico dos comandos já executados.

Pacotes (*Panel Packages*): mostra os pacotes instalados e carregados pelo R. Um pacote é nada mais que um módulo no R, cada qual com sua finalidade específica. Observe a presença de quatro abas: *Files*, para carregar e visualizar arquivos do sistema; *Plots*, para visualizar figuras; *Help*, para acessar o sistema de ajuda do R e *Viewer*, para mostrar resultados dinâmicos e interativos, tal como uma página da internet.

Como um exercício introdutório, vamos inicializar duas variáveis. Dentro do console do R (lado esquerdo inferior), digite os seguintes comandos e aperte *enter* ao final de cada linha. O símbolo <- é nada mais que a junção de < com -. O símbolo ' representa uma aspa simples e sua localização no teclado Brasileiro é no botão abaixo do *escape* (*esc*), lado esquerdo superior do teclado.

```
# set x and y
x <- 1
y <- 'my text'
```

Após a execução, dois objetos devem aparecer no painel *Environment*, um chamado `x` com o valor 1, e outro chamado `y` com o conjunto de caracteres `'my text'`. O histórico de comandos na aba *History* também foi atualizado com os comandos utilizados anteriormente.

Agora, vamos mostrar na tela os valores de `x`. Para isso, digite o seguinte comando no *prompt* e aperte *enter* novamente:

```
# print x
print(x)
```

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

A função **print()** é uma das principais funções para mostrarmos valores no *prompt* do R. O texto apresentado como `[1]` indica o índice do primeiro número da linha. Para verificar isso, digite o seguinte comando, o qual irá mostrar vários números na tela:

```
# print vector from 50 to 100
print(50:100)
```

```
R>  [1]  50  51  52  53  54  55  56  57  58  59  60  61  62  63
R> [15]  64  65  66  67  68  69  70  71  72  73  74  75  76  77
R> [29]  78  79  80  81  82  83  84  85  86  87  88  89  90  91
R> [43]  92  93  94  95  96  97  98  99 100
```

Nesse caso, utilizamos o símbolo : em `50:100` para criar uma sequência iniciando em 50 e terminando em 100. Observe que temos valores encapsulados por colchetes ([]) no lado esquerda da tela. Esses representam os índices do primeiro elemento apresentado na linha. Por exemplo, o décimo quinto elemento do vetor criado é o valor 64.

3.1 Executando Códigos em um *Script*

Agora, vamos juntar todos os códigos digitados anteriormente e colar na tela do editor (lado esquerdo superior), assim como mostrado a seguir:

```
# set objects
x <- 1
y <- 'my text'

# print it
print(x)
print(1:50)
```

Após colar todos os comandos no editor, salve o arquivo *.R* em alguma pasta pessoal. Esse arquivo, o qual no momento não faz nada de especial, registrou os passos de um algoritmo simples que cria dois objetos e mostra os seus valores. Futuramente esse irá ter mais forma, com a importação de dados, manipulação e modelagem dos mesmos e saída de tabelas e figuras.

No RStudio existem alguns atalhos predefinidos para executar códigos que economizam bastante tempo. Para executar um *script* inteiro, basta apertar `control + shift + s`. Esse é o comando *source*. Com o RStudio aberto, sugiro testar essa combinação de teclas e verificar como o código digitado anteriormente é executado, mostrando os valores no *prompt* do R. Visualmente, o resultado deve ser próximo ao apresentado na figura Figura 3.3.

Outro comando muito útil é a execução por linha. Nesse caso não é executado todo o arquivo, mas somente a linha em que o cursor do *mouse* se encontra. Para isto, basta apertar `control+enter`. Esse atalho é bastante útil no desenvolvimento de rotinas pois permite que cada linha seja testada antes de executar o programa inteiro. Como um exemplo de uso, aponte o cursor para a linha `print(x)` e pressione `control + enter`. Verás que o valor de `x` é mostrado na tela do *prompt*. A seguir destaco esses e outros atalhos do RStudio, os quais também são muito úteis.

- **control+shift+s** executa o arquivo atual do RStudio, sem mostrar comandos no *prompt* (sem eco – somente saída);
- **control+shift+enter**: executa o arquivo atual, mostrando comandos na tela (com eco – código e saída);
- **control+enter**: executa a linha selecionada, mostrando comandos na tela;

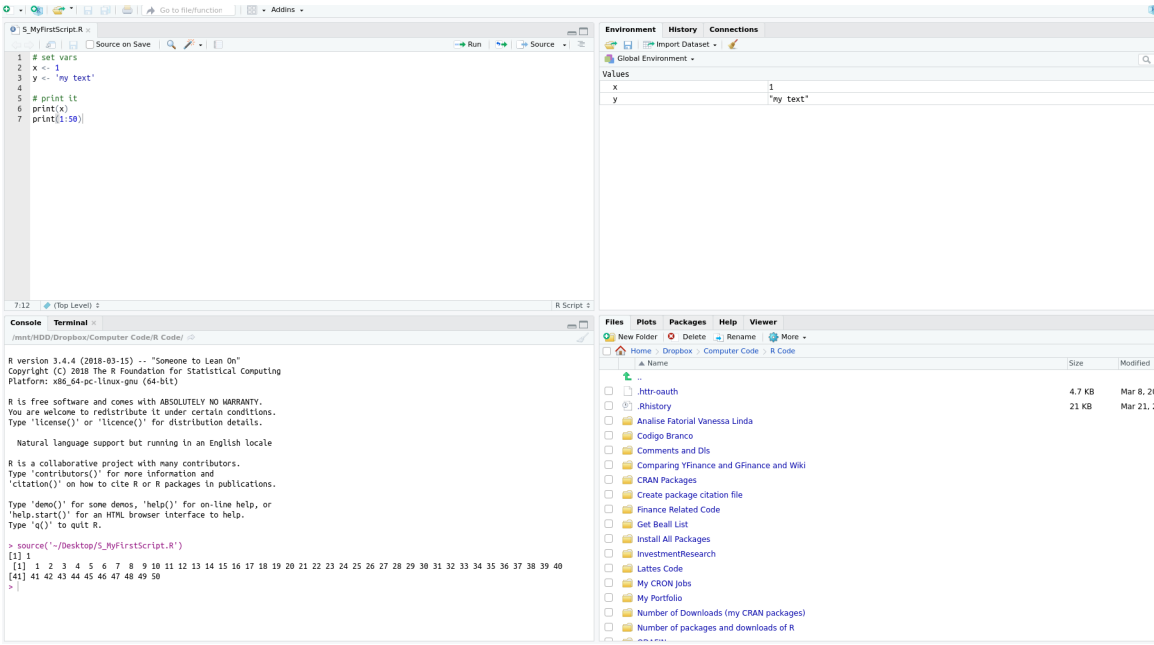


Figura 3.3: Exemplo de Rotina no R

- **control+shift+b**: executa os códigos do início do arquivo até a linha atual onde o cursor se encontra;
- **control+shift+e**: executa os códigos da linha onde o cursor se encontra até o final do arquivo.

Sugere-se que esses atalhos sejam memorizados e utilizados. Isso facilita bastante o uso do programa. Para aqueles que gostam de utilizar o *mouse*, uma maneira alternativa para rodar o código do *script* é apertar o botão *source*, localizado no canto direito superior do editor de rotinas. Isto é equivalente ao atalho `control+shift+s`.

Porém, no mundo real de programação, poucos são os casos em que uma análise de dados é realizada por um *script* apenas. Como uma forma de organizar o código, pode-se dividir o trabalho em N *scripts* diferentes, onde um deles é o "mestre", responsável por rodar os demais.

Neste caso, para executar os *scripts* em sequência, basta chamá-los no *script* mestre com o comando **source()** , como no código a seguir:

```
# Import all data
source('01-import-data.R')

```

```
# Clean up
source('02-clean-data.R')

# Build tables
source('03-build-table.R')
```

Nesse caso, o código anterior é equivalente a abrirmos e executarmos (*control + shift + s*) cada um dos *scripts* sequencialmente.

Como podemos ver, existem diversas maneiras de executar uma rotina de pesquisa. Na prática, porém, iras centralizar o uso em dois comandos apenas: `control+shift+s` para rodar o *script* inteiro e `control+enter` para rodar por linha.

3.2 Tipos de Arquivos

Assim como outros programas, o R e o RStudio possuem um ecossistema de arquivos e cada extensão tem uma finalidade diferente. A seguir apresenta-se uma descrição de diversas extensões de arquivos exclusivos ao R e RStudio. Os itens da lista estão ordenados por ordem de importância e uso.

Arquivos com extensão *.R*: Representam arquivos do tipo texto contendo diversas instruções para o R. Esses são os arquivos que conterão o código da pesquisa e onde passaremos a maior parte do tempo. Também pode ser chamado de um *script* ou rotina de pesquisa. Como sugestão, pode-se dividir toda uma pesquisa em etapas e arquivos numerados. Exemplos: *01-Get-Data.R*, *02-Clean-data.R*, *03-Estimate-Models.R*.

Arquivos com extensão *.RData* e *.rds*: armazenam dados nativos do R. Esses arquivos servem para salvar (ou congelar) objetos do R em um arquivo no disco rígido do computador para, em sessão futura, serem novamente carregados. Por exemplo, podes guardar o resultado de uma pesquisa em uma tabela, a qual é salva em um arquivo com extensão *.RData* ou *.rds*. Exemplos: *Raw-Data.RData*, *Table-Results.rds*.

Arquivos com extensão *.Rmd*, *.Rnw* e *.qmd*: São arquivos relacionados a tecnologia *Rmarkdown* e *Quarto*. O uso desses arquivos permite a criação de documentos onde texto e código são integrados.

Arquivos com extensão *.Rproj*: Contém informações para a edição de projetos no RStudio. O sistema de projetos do RStudio permite a configuração customizada do projeto e também facilita a utilização de ferramentas de controle de código, tal como controle de versões. O seu uso, porém, não é essencial. Para aqueles com interesse em conhecer esta funcionalidade, sugiro a leitura do manual do RStudio. Uma maneira simples de entender os tipos de projetos disponíveis é, no RStudio, clicar em "File", "New project", "New Folder" e assim deve aparecer uma tela com todos os tipos possíveis de projetos no RStudio. Exemplo: *My-Dissertation-Project.Rproj*.

3.3 Testando Código

O desenvolvimento de códigos em R segue um conjunto de etapas. Primeiro você escreverá uma nova linha de comando em uma rotina. Essa linha será testada com o atalho `control + enter`, verificando-se a ocorrência de erros e as saídas na tela. Caso não houver erro e o resultado for igual ao esperado, parte-se para a próxima linha de código.

Um ciclo de trabalho fica claro, a escrita do código da linha atual é seguida pela execução, seguido da verificação de resultados, modificação caso necessário e assim por diante. Esse é um processo normal e esperado. Dado que uma rotina é lida e executada de cima para baixo, você precisa ter certeza de que cada linha de código está corretamente especificada antes de passar para a próxima.

Quando você está tentando encontrar um erro em um *script* preexistente, o R oferece algumas ferramentas para controlar e avaliar sua execução. Isso é especialmente útil quando você possui um código longo e complicado. A ferramenta de teste mais simples e fácil de utilizar que o RStudio oferece é o ponto de interrupção do código. No RStudio, você pode clicar no lado esquerdo do editor e aparecerá um círculo vermelho, como na Figura Figura 3.4.

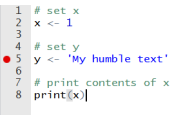


Figura 3.4: Exemplo de debug

O círculo vermelho indica um ponto de interrupção do código que forçará o R a pausar a execução nessa linha. Quando a execução atinge o ponto de

interrupção, o *prompt* mudará para `browser[1]>` e você poderá verificar o conteúdo dos objetos. No console, você tem a opção de continuar a execução para o próximo ponto de interrupção ou interrompê-la. O mesmo resultado pode ser alcançado usando a função **browser()** . Dê uma olhada:

```
# set x
x <- 1

# set y
browser()
y <- 'My humble text'

# print contents of x
print(x)
```

O resultado prático do código anterior é o mesmo que utilizar o círculo vermelho do RStudio, figura Figura 3.4. Porém, o uso do **browser()** permite mais controle sobre onde a execução deve ser pausada. Como um teste, copie e cole o código anterior no RStudio, salve em um novo *script* e execute com *Control + Shift + S*. Para sair do ambiente de depuramento (*debug*), aperte *enter* no *prompt* do RStudio.

3.4 Cancelando a Execução de um Código

Toda vez que o R estiver executando algum código, uma sinalização visual no formato de um pequeno círculo vermelho no canto direito do *prompt* irá aparecer. Caso conseguir ler (o símbolo é pequeno em monitores modernos), o texto indica o termo *stop*. Esse símbolo não somente indica que o programa ainda está rodando mas também pode ser utilizado para cancelar a execução de um código. Para isso, basta clicar no referido botão. Outra maneira de cancelar uma execução é apontar o mouse no *prompt* e pressionar a tecla *Esc* no teclado.

Para testar o cancelamento de código, copie e cole o código a seguir em um *script* do RStudio. Após salvar, rode o mesmo com `control+shift+s`.

```
for (i in 1:100) {
  message('\nRunning code (please make it stop by hitting esc!)')
  Sys.sleep(1)
}
```

O código anterior usa um comando especial do tipo `for` para mostrar a mensagem a cada segundo. Neste caso, o código demorará 100 segundos para rodar. Caso não desejes esperar, aperte `esc` para cancelar a execução. Por enquanto, não se preocupe com as funções utilizadas no exemplo. Iremos discutir o uso do comando `for` no capítulo Capítulo 10.

3.5 Procurando Ajuda

Uma tarefa muito comum no uso do R é procurar ajuda. A quantidade de funções disponíveis para o R é gigantesca e memorizar todas peculariedades é quase impossível. Assim, até mesmo usuários avançados comumente procuram ajuda sobre tarefas específicas no programa, seja para entender detalhes sobre algumas funções ou estudar um novo procedimento.

É possível buscar ajuda utilizando tanto o painel de *help* do RStudio como diretamente do *prompt*. Para isso, basta digitar o ponto de interrogação junto ao objeto sobre o qual se deseja ajuda, tal como em `?mean`. Nesse caso, o objeto **mean()** é uma função e o uso do comando irá abrir o painel de ajuda sobre ela.

No R, toda tela de ajuda de uma função é igual, conforme se vê na Figura 3.5 apresentada a seguir. Esta mostra uma descrição da função **mean()** , seus argumentos de entrada explicados e também o seu objeto de saída. A tela de ajuda segue com referências e sugestões para outras funções relacionadas. Mais importante, os **exemplos de uso da função** aparecem por último e podem ser copiados e colados para acelerar o aprendizado no uso da função.

Caso quiséssemos procurar um termo nos arquivos de ajuda, bastaria utilizar o comando `??"standard deviation"`. Essa operação irá procurar a ocorrência do termo em todos os pacotes do R e é muito útil para aprender como realizar alguma operação, nesse caso o cálculo de desvio padrão.

Como sugestão, o ponto inicial e mais direto para aprender uma nova função é observando o seu exemplo de uso, localizada no final da página de ajuda. Com isto, podes verificar quais tipos de objetos de entrada a mesma aceita e qual o formato e o tipo de objeto na sua saída. Após isso, leia atentamente a tela de ajuda para entender se a mesma faz exatamente o que esperas e quais são as suas opções de uso nas respectivas entradas. Caso a função realizar o procedimento desejado, podes copiar e colar o exemplo de uso para o teu próprio *script*, ajustando onde for necessário.

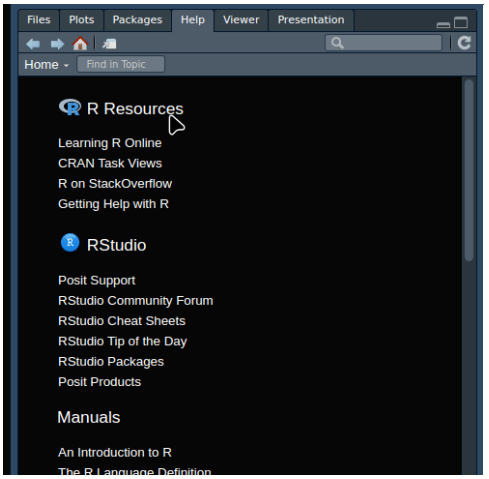


Figura 3.5: Usando o sistema de ajuda do RStudio

Outra fonte muito importante de ajuda é a própria internet. Sites como *stackoverflow.com* e *mailing lists* específicos do R, cujo conteúdo também está na internet, são fontes preciosas de informação. Havendo alguma dúvida que não foi possível solucionar via leitura dos arquivos de ajuda do R, vale o esforço de procurar uma solução via mecanismo de busca na internet. Em muitas situações, o seu problema, por mais específico que seja, já ocorreu e já foi solucionado por outros usuários.

Caso estiver recebendo uma mensagem de erro enigmática, outra dica é copiar e colar a mesma para uma pesquisa no Google. Aqui apresenta-se outro benefício do uso da língua inglesa. É mais provável que encontres a solução se o erro for escrito em inglês, dado o maior número de usuários na comunidade global. Caso não encontrar uma solução desta forma, podes inserir uma pergunta no stackoverflow ou no grupo Brasileiro do R no Facebook.

> **Cuidado**
>
> Toda vez que for pedir ajuda na internet, procure sempre 1) descrever claramente o seu problema e 2) adicionar um código reproduzível do seu problema. Assim, o leitor pode facilmente verificar o que está acontecendo ao rodar o exemplo no seu computador. Não tenho dúvida que, se respeitar ambas regras, logo uma pessoa caridosa lhe ajudará com o seu problema.

3.6 Utilizando *Code Completion* com a Tecla *tab*

Um dos recursos mais úteis do RStudio é o preenchimento automático de código (*code completion*). Essa é uma ferramenta de edição que facilita o encontro de nomes de objetos, nome de pacotes, nome de arquivos e nomes de entradas em funções. O seu uso é muito simples. Após digitar um texto qualquer, basta apertar a tecla *tab* e uma série de opções aparecerá. Veja a Figura @ref(fig:autocomplete) apresentada a seguir, em que, após digitar a letra *f* e apertar *tab*, aparece uma janela com uma lista de objetos que iniciam com a respectiva letra. Figura Figura 3.6

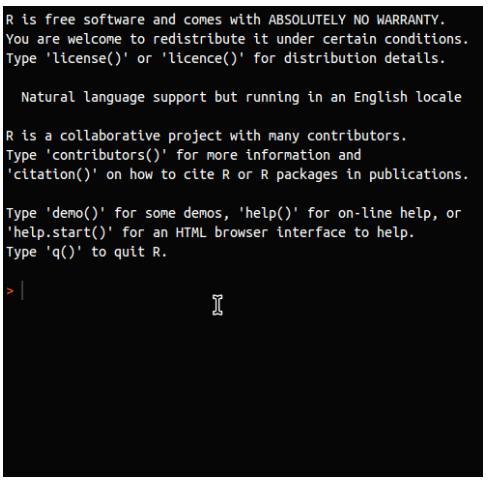


Figura 3.6: Usando o sistema de ajuda do RStudio

Essa ferramenta também funciona para pacotes. Para verificar, digite `library(r)` no *prompt* ou no editor, coloque o cursor entre os parênteses e aperte *tab*. O resultado deve ser algo parecido com a Figura 3.7.

Observe que uma descrição do pacote ou objeto também é oferecida. Isso facilita bastante o dia a dia, pois a memorização das funcionalidades e dos nomes dos pacotes e os objetos do R não é uma tarefa fácil. O uso do *tab* diminui o tempo de investigação dos nomes e evita possíveis erros de digitação na definição destes.

O uso dessa ferramenta torna-se ainda mais benéfico quando os objetos são nomeados com algum tipo de padrão. No restante do livro observarás que os objetos tendem a ser nomeados com o prefixo *my*, como em `my_x`, `my_num`, `my_char`. O uso desse padrão facilita o encontro futuro do nome dos objetos, pois basta digitar *my*, apertar *tab* e uma lista de todos os objetos criados pelo usuário aparecerá.

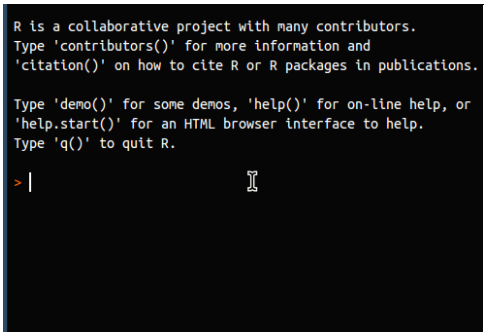


Figura 3.7: Usando o autocomplete para procurar pacotes

Outro uso do *tab* é no encontro de arquivos e pastas no computador. Basta criar uma variável como `my_file <- " "`, apontar o cursor para o meio das aspas e apertar a tecla *tab*. Uma tela com os arquivos e pastas do diretório atual de trabalho aparecerá, conforme mostrado na Figura 3.8. Nesse caso específico, o R estava direcionado para a minha pasta de códigos, em que é possível enxergar diversos trabalhos realizados no passado.

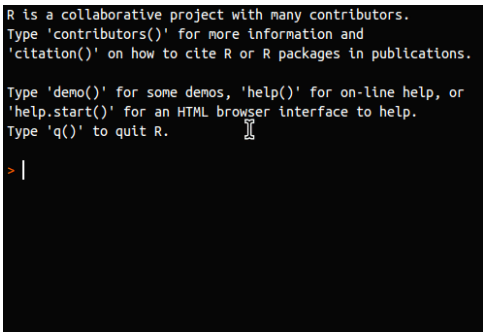


Figura 3.8: Usando o autocomplete para navegar diretórios

Uma dica aqui é utilizar o *tab* com a raiz do computador. Assumindo que o disco do seu computador está alocado para `C:/`, digite `my_file <- "C:/"` e pressione *tab* após o símbolo `/`. Uma tela com os arquivos da raiz do

computador aparecerá no RStudio. Podes facilmente navegar o sistema de arquivos utilizando as setas e *enter*.

O *autocomplete* também funciona para encontrar e definir as entradas de uma função. Veja um exemplo na Figura 3.9.

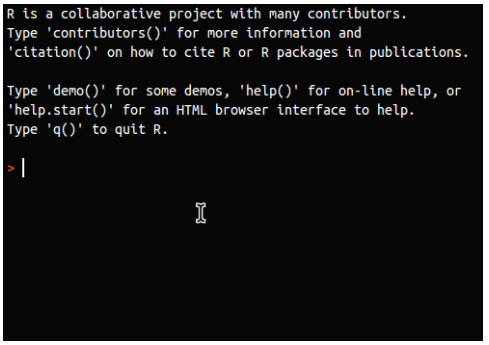


Figura 3.9: Usando o autocomplete para navegar argumentos de funções

> **Importante**
>
> O *autocomplete* é uma das ferramentas mais importantes do RStudio, funcionando para encontro de objetos, locais no disco rígido, pacotes e funções. Acostume-se a utilizar a tecla *tab* o quanto antes e logo verá como fica mais fácil escrever código rapidamente, e sem erros de digitação.

CAPÍTULO 4

PACOTES DO R

Um dos grandes benefícios do uso do R é o seu acervo de pacotes. Esses representam um conjunto de procedimentos agrupados em uma coleção de funções e voltados para a resolução de um problema qualquer. O R tem em sua essência uma filosofia de colaboração. Usuários disponibilizam os seus códigos para outras pessoas utilizarem. E, mais importante, **todos os pacotes são gratuitos**, assim como o R. Por exemplo, considere um caso em que está interessado em baixar dados da internet sobre o desemprego histórico no Brasil. Para isso, basta procurar e instalar o pacote específico que realiza esse procedimento.

Esses pacotes podem ser instalados de diferentes fontes, com as principais sendo **CRAN** (*The Comprehensive R Archive Network*) e **Github**. A cada dia aumenta a quantidade e diversidade de pacotes existentes para o R. O próprio autor deste livro possui diversos pacotes disponíveis no CRAN, cada um para resolver algum problema diferente. Na grande maioria, são pacotes para importar e organizar dados financeiros.

O CRAN é o repositório oficial do R e é livre. Qualquer pessoa pode enviar um pacote e todo código enviado está disponível na internet. Existe, porém, um processo de avaliação que o código passa e certas normas rígidas devem ser respeitadas sobre o formato do código, o manual do usuário e a forma de atualização do pacote. Para quem tiver interesse, um tutorial claro e fácil de seguir é apresentado no site [https://r-pkgs.org/). As regras completas estão disponíveis no site do CRAN - https://cran.r-

project.org/web/packages/policies.html. A adequação do código a essas normas é responsabilidade do desenvolvedor e gera um trabalho significativo, principalmente na primeira submissão.

A lista completa de pacotes disponíveis no CRAN, juntamente com uma breve descrição, pode ser acessada no link *packages* do site do R - https://cran.r-project.org/. Uma maneira prática de verificar a existência de um pacote para um procedimento específico é carregar a página anterior e procurar no seu navegador de internet a palavra-chave que define o seu procedimento. Caso existir o pacote com a palavra-chave, a procura acusará o encontro do termo na descrição do pacote.

Outra fonte importante para o encontro de pacotes é o *Task Views*, em que são destacados os principais pacotes de acordo com a área e o tipo de uso. Veja a tela do *Task Views* na Figura **??**.

CRAN Task Views

Bayesian	Bayesian Inference
ChemPhys	Chemometrics and Computational Physics
ClinicalTrials	Clinical Trial Design, Monitoring, and Analysis
Cluster	Cluster Analysis & Finite Mixture Models
DifferentialEquations	Differential Equations
Distributions	Probability Distributions
Econometrics	Econometrics
Environmetrics	Analysis of Ecological and Environmental Data
ExperimentalDesign	Design of Experiments (DoE) & Analysis of Experimental Data
ExtremeValue	Extreme Value Analysis
Finance	Empirical Finance
Genetics	Statistical Genetics
Graphics	Graphic Displays & Dynamic Graphics & Graphic Devices & Visualization
HighPerformanceComputing	High-Performance and Parallel Computing with R
MachineLearning	Machine Learning & Statistical Learning
MedicalImaging	Medical Image Analysis
MetaAnalysis	Meta-Analysis
Multivariate	Multivariate Statistics
NaturalLanguageProcessing	Natural Language Processing
NumericalMathematics	Numerical Mathematics
OfficialStatistics	Official Statistics & Survey Methodology
Optimization	Optimization and Mathematical Programming
Pharmacokinetics	Analysis of Pharmacokinetic Data
Phylogenetics	Phylogenetics, Especially Comparative Methods
Psychometrics	Psychometric Models and Methods
ReproducibleResearch	Reproducible Research
Robust	Robust Statistical Methods

Figura 4.1: Tela do Task Views

Ao contrário do CRAN, o *Github* não possui restrição quanto ao código enviado e, devido a isso, tende a ser escolhido como ambiente de compartilhamento de código. A responsabilidade de uso, porém, é do próprio usuário. Na prática, é muito comum os desenvolvedores manterem uma versão em desenvolvimento no *Github* e outra oficial no CRAN. Quando a versão em desenvolvimento atinge um estágio de maturidade, a mesma é enviada ao CRAN.

O mais interessante no uso de pacotes é que estes podem ser acessados e instalados diretamente no R via a internet. Para saber qual é a quantidade atual de pacotes no CRAN, digite e execute os seguintes comandos no *prompt*:

```
# find current available packages
df_cran_pkgs <- available.packages()

# get size of matrix
n_cran_pkgs <- nrow(df_cran_pkgs)

# print it
print(n_cran_pkgs)
```

```
R> [1] 20271
```

Atualmente, 2024-01-10 16:17:50.874593, existem 20271 pacotes disponíveis nos servidores do CRAN.

Também se pode verificar a quantidade de pacotes localmente instalados com o comando `installed.packages()`:

```
# get number of local (installed) packages
n_local_pkgs <- nrow(installed.packages())

# print it
print(n_local_pkgs)
```

```
R> [1] 431
```

Nesse caso, o computador em que o livro foi escrito possui 431 pacotes do R instalados. Note que, apesar do autor ser um experiente programador do R, apenas uma pequena fração do pacotes disponíveis no CRAN está sendo usada! A diversidade dos pacotes é gigantesca.

4.1 Instalando Pacotes do CRAN

Para instalar um pacote, basta utilizar o comando **install.packages()** . Como exemplo, vamos instalar um pacote que será utilizado nos capítulos futuros, o **{readr}** (Wickham, Hester, e Bryan 2023):

```
# install pkg readr
install.packages('readr')
```

Copie e cole este comando no *prompt* e pronto! O R irá baixar os arquivos necessários e instalar o pacote **{readr}** (Wickham, Hester, e Bryan 2023) e suas dependências. Após isto, as funções relativas ao pacote estarão prontas para serem usadas após o carregamento do módulo (detalhes a seguir). Observe que definimos o nome do pacote na instalação como se fosse texto, com o uso das aspas ("). Caso o pacote instalado seja dependente de outros pacotes, o R automaticamente instala todos módulos faltantes. Assim, todos os requerimentos para o uso do respectivo pacote já serão satisfeitos e tudo funcionará perfeitamente. É possível, porém, que um pacote tenha uma dependência externa. Como um exemplo, o pacote `RndTexExams` depende da existência de uma instalação do LaTex. Geralmente essa é anunciada na sua descrição e um erro é sinalizado na execução do programa quando o LaTex não é encontrado. Fique atento, portanto, a esses casos.

Aproveitando o tópico, sugiro que o leitor já instale todos os pacotes do **{tidyverse}** (Wickham 2023c) com o seguinte código:

```
# install pkgs from tidyverse
install.packages('tidyverse')
```

O **{tidyverse}** (Wickham 2023c) é um conjunto de pacotes voltados a *data science* e com uma sintaxe própria e consistente, voltada a praticabilidade. Verás que, em uma instalação nova do R, o **{tidyverse}** (Wickham 2023c) depende de uma grande quantidade de pacotes.

4.2 Instalando Pacotes do Github

Para instalar um pacote diretamente do Github, é necessário instalar antes o pacote *devtools*, disponível no CRAN:

```
# install devtools
install.packages('devtools')
```

Após isto, utilize função `devtools::install_github` para instalar um pacote diretamente do Github. Note que o símbolo `::` indica que função `install_github` pertence ao pacote `devtools`. Com esta particular sintaxe, não precisamos carregar todo o pacote para utilizar apenas uma função.

No exemplo a seguir instalamos a versão em desenvolvimento do pacote `ggplot2`, cuja versão oficial também está disponível no CRAN:

```
# install ggplot2 from github
devtools::install_github("hadley/ggplot2")
```

Observe que o nome do usuário do repositório também é incluído. No caso anterior, o nome *hadley* pertence ao desenvolvedor do `ggplot2`, Hadley Wickham. No decorrer do livro notará que esse nome aparecerá diversas vezes, dado que Hadley é um prolífico e competente desenvolvedor de diversos pacotes do R e do **{tidyverse}** (Wickham 2023c).

> **Cuidado**
>
> Um aviso aqui é importante. **Os pacotes do github não são moderados. Qualquer pessoa pode enviar código para lá e o conteúdo não é checado de forma independente.** Nunca instale pacotes do github sem conhecer os autores. Apesar de improvável – nunca aconteceu comigo por exemplo – é possível que esses possuam algum código malicioso.

4.3 Carregando Pacotes

Dentro de uma rotina de pesquisa, utilizamos a função `library` para carregar um pacote na nossa sessão do R. Ao fecharmos o RStudio ou então iniciar uma nova sessão do R, os pacotes são descarregados. Vale salientar que alguns pacotes, tal como o **{base}** (R Core Team 2023b) e o `stats`, são inicializados automaticamente a cada nova sessão. A grande maioria, porém, deve ser carregada no início dos *scripts*. Veja o exemplo a seguir:

```
# load dplyr
library(dplyr)
```

A partir disso, todas as funções do pacote estarão disponíveis para o usuário. Note que não é necessário utilizar aspas (") ao carregar o pacote. Caso utilize uma função específica do pacote e não deseje carregar todo ele, pode fazê-lo através do uso do símbolo especial `::`, conforme o exemplo a seguir.

```
# call fct fortune() from pkg fortune
fortunes::fortune(10)
```

```
R>
R> Overall, SAS is about 11 years behind R and S-Plus in
R> statistical capabilities (last year it was about 10 years
R> behind) in my estimation.
R>    -- Frank Harrell (SAS User, 1969-1991)
R>       R-help (September 2003)
```

Nesse caso, utilizamos a função `fortune` do próprio pacote `fortunes`, o qual mostra na tela uma frase possivelmente engraçada escolhida do *mailing list* do R. Nesse caso, selecionamos a mensagem número 10. Se não tiver disponível o pacote, o R mostrará a seguinte mensagem de erro:

```
R> Error in library("fortune") : there is no package called "fortune"
```

Para resolver, utilize o comando `install.packages("fortunes")` para instalar o pacote no seu computador. Execute o código `fortunes::fortune(10)` no *prompt* para confirmar a instalação. Toda vez que se deparar com essa mensagem de erro, deves instalar o pacote que está faltando.

Outra maneira de carregar um pacote é através da função `require`. Essa tem um comportamento diferente da função `library` e deve ser utilizada dentro da definição de funções ou no teste do carregamento do pacote. Caso o usuário crie uma função customizada que necessite de procedimentos de um pacote em particular, o mesmo deve carregar o pacote no escopo da função. Por exemplo, veja o código a seguir, em que criamos uma função dependente do pacote `quantmod`:

```
my_fct <- function(x){
  require(quantmod)

  df <- getSymbols(x, auto.assign = F)
  return(df)
}
```

Nesse caso, a função `getSymbols` faz parte do pacote `quantmod`. Não se preocupe agora com a estrutura utilizada para criar uma função no R. Essa será explicada em capítulo futuro.

Cuidado!

Uma precaução que deve sempre ser tomada quando se carrega um pacote é um possível **conflito de funções**. Por exemplo, existe uma função chamada `filter` no pacote **{dplyr}** (Wickham, François, et al. 2023) e também no pacote `stats`. Caso carregarmos ambos pacotes e chamarmos a função `filter` no escopo do código, qual delas o R irá usar? Pois bem, a **preferência é sempre para o último pacote carregado**. Esse é um tipo de problema que pode gerar muita confusão. Felizmente, note que o próprio R acusa um conflito de nome de funções no carregamento do pacote. Para testar, inicie uma nova sessão do R e carregue o pacote **{dplyr}** (Wickham, François, et al. 2023). Verás que uma mensagem indica haver dois conflitos com o pacote `stats` e quatro com pacote o **{base}** (R Core Team 2023b).

4.4 Atualizando Pacotes

Ao longo do tempo, é natural que os pacotes disponibilizados no CRAN sejam atualizados para acomodar novas funcionalidades ou se adaptar a mudanças em suas dependências. Assim, é recomendável que os usuários atualizem os seus pacotes instalados para uma nova versão através da internet. Esse procedimento é bastante fácil. Uma maneira direta de atualizar pacotes é clicar no botão *update* no painel de pacotes no canto direito inferior do RStudio, conforme mostrado na figura @ref(fig:RStudio-update).

A atualização de pacotes através do *prompt* também é possível. Para isso, basta utilizar o comando `update.packages`, conforme mostrado a seguir.

```
update.packages()
```

O comando `update.packages()` compara a versão dos pacotes instalados em relação a versão disponível no CRAN. Caso tiver alguma diferença, a nova versão é instalada. Após a execução do comando, todos os pacotes estarão atualizados com a versão disponível nos servidores do CRAN.

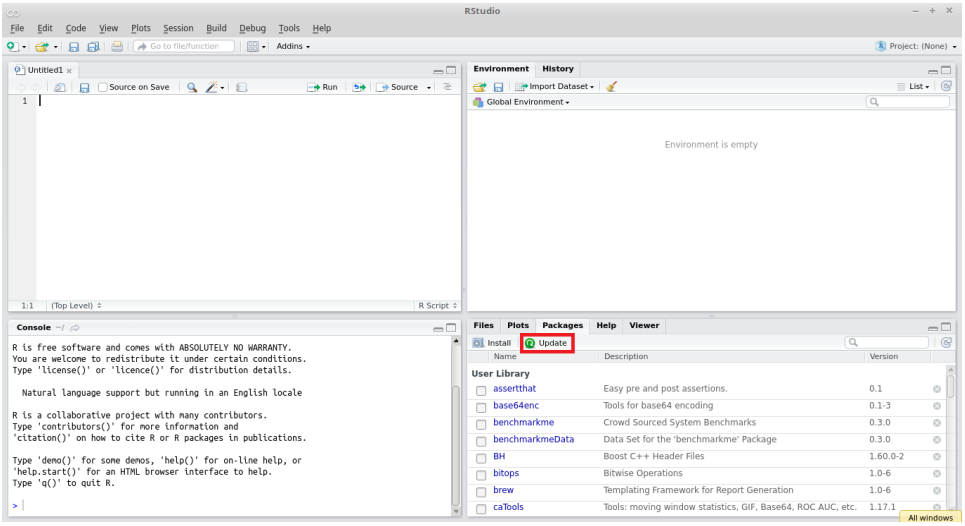


Figura 4.2: Atualizando pacotes no R

Importante

Versionamento de pacotes é extremamente importante para manter a reprodutibilidade do código. Apesar de ser raro de acontecer, é possível que a atualização de um pacote no R modifique, para os mesmos dados, resultados já obtidos anteriormente. Tenho uma experiência particularmente memorável quando um artigo científico retornou da revisão e, devido a atualização de um dos pacotes, não consegui reproduzir os resultados apresentados no artigo. No final deu tudo certo, mas o trauma fica.

Uma solução para este problema é congelar as versões dos pacotes para cada projeto usando a ferramenta `packrat` do RStudio. Em resumo, o `packrat` faz cópias locais dos pacotes utilizados no projeto, os quais têm preferência aos pacotes do sistema. Assim, se um pacote for atualizado no sistema, mas não no projeto, o código R vai continuar usando a versão mais antiga e seu código sempre rodará nas mesmas condições.

CAPÍTULO 5

INTERAGINDO COM O SISTEMA OPERACIONAL E A INTERNET

Um dos grandes benefícios de aprender programação é utilizar computador para interagir com o seu computador ou a internet. Na maioria das situações, iremos criar novas pastas, listar e remover arquivos, além de várias outras operações. Na internet, podemos realizar o *download* de arquivos via código, interagir com serviçoes de nuvem como o Google Drive.

5.1 Mostrando e Mudando o Diretório de Trabalho

Assim como outros softwares, **o R sempre trabalha em algum diretório**. É com base nesse diretório que o R procura arquivos para importar dados. É nesse mesmo diretório que o R salva arquivos, caso não definirmos um endereço no computador explicitamente. Essa saída pode ser um arquivo de uma figura, um arquivo de texto ou uma planilha eletrônica.

> **!** Importante
>
> Como boa prática de criação e organização de *scripts*, deve-se sempre mudar o diretório de trabalho para onde o arquivo do *script* está localizado. Isso facilita a importação e exportação de dados de arqui-

vos.

> **!** Importante
>
> Uma forma simples e direta de mudar o diretório de trabalho é utilizar o sistema de projetos do RStudio. Toda vez que um projeto é aberto, a sessão do R automaticamente será direcionado ao diretório do projeto.

Em sua inicialização, o R possui como diretório *default* a pasta de documentos do usuário cujo atalho é o tilda (`'~'`).

Para mostrar o diretório atual de trabalho, basta utilizar a função **getwd()** :

```
# get current directory
my_dir <- getwd()

# print it
print(my_dir)
```

```
R> [1] "/tmp/RtmpoywnSz/book-compile__470f130a44e0"
```

O resultado do código anterior mostra a pasta onde este livro foi escrito. Esse é o diretório onde os arquivos do livro foram compilados dentro do ambiente Linux.

A mudança de diretório de trabalho é realizada através do comando **setwd()** . Por exemplo, caso quiséssemos mudar o nosso diretório de trabalho para *C:/Minha pesquisa/*, basta digitar no *prompt*:

```
# set dir
my_d <- 'C:/Minha Pesquisa/'
setwd(my_d)
```

Enquanto para casos simples, como o anterior, lembrar o nome do diretório é fácil, em casos práticos o diretório de trabalho pode ser em um lugar mais profundo da raiz de diretórios do sistema de arquivos. Nessa situação, uma estratégia eficiente para descobrir a pasta de trabalho é utilizar um explorador de arquivos, tal como o *Explorer* no Windows. Abra esse aplicativo e vá até o local onde quer trabalhar com o seu *script*. Após isso, coloque o cursor na barra de endereço e selecione todo o endereço.

Aperte *control+c* para copiar o endereço para a área de transferência. Volte para o seu código e cole o mesmo no código. **Atenção nesta etapa, o Windows utiliza a barra invertida para definir endereços no computador, enquanto o R utiliza a barra normal**. Caso tente utilizar a barra invertida, um erro será mostrado na tela. Veja o exemplo a seguir.

```
my_d <- 'C:\Minha pesquisa\'
setwd(my_d)
```

O erro terá a seguinte mensagem:

```
Error: '\M' is an unrecognized escape in character string..."
```

A justificativa para o erro é que a barra invertida \ é um caractere reservado no R e não pode ser utilizado isoladamente. Caso precises, podes definí-lo no objeto de texto com dupla barra, tal como em \\. Veja no exemplo a seguir, onde a dupla barra é substituída por uma barra única:

```
# set char with \
my_char <- 'using \\'

# print it
message(my_char)
```

```
R> using \
```

A solução do problema é simples. Após copiar o endereço, modifique todas as barras para a barra normal, assim como no código a seguir:

```
my_d <- 'C:/Minha pesquisa/'
setwd(my_d)
```

É possível também utilizar barras invertidas duplas \\ na definição de diretórios, porém não se recomenda essa formatação, pois não é compatível com outros sistemas operacionais.

Outro ponto importante aqui é o uso de endereços relativos. Por exemplo, caso esteja trabalhando em um diretório que contém um subdiretório chamado `Data`, podes entrar nele com o seguinte código:

```
# change to subfolder
setwd('Data')
```

Outra possibilidade pouco conhecida no uso de **setwd()** é que é possível entrar em níveis inferiores do sistema de diretórios com `..`, tal como em:

```
# change to previous level
setwd('..')
```

Portanto, caso estejas trabalhando no diretório `C:/My Research/` e executar o comando `setwd('..')`, o diretório atual de trabalho viraria `C:/`, um nível inferior a `C:/My Research/`.

Uma maneira mais moderna e pouco conhecida de definir o diretório de trabalho é usar as funções internas do RStudio. Este é um conjunto de funções que só funcionam dentro do RStudio e fornecem diversas informações sobre o arquivo sendo editado. Para descobrir o caminho do arquivo atual que está sendo editado no RStudio e configurar o diretório de trabalho para lá, você pode escrever:

```
my_path <- dirname(rstudioapi::getActiveDocumentContext()$path)
setwd(my_path)
```

Dessa forma, o *script* mudará o diretório para sua própria localização. Apesar de não ser um código exatamente elegante, ele é bastante funcional. Caso copie o arquivo para outro diretório, o valor de `my_path` muda para o novo diretório. Esteja ciente, no entanto, de que esse truque só funciona no editor de rotinas do RStudio e dentro de um arquivo salvo. O código não funcionará a partir do *prompt*.

> **Importante**
>
> Outro truque bastante útil para definir diretórios de trabalho no R é usar o símbolo `~`. Esse define a pasta `'Documentos'` no *Windows*, a qual é única para cada usuário. Portanto, ao executar `setwd('~')`, irás direcionar o R a uma pasta de fácil acesso e livre modificação pelo usuário atual do computador.

5.1.1 Listando Arquivos e Pastas

Para listar arquivos do computador, basta utilizar o função **list.files()** ou então a alternativa do **fs**::**dir_ls()** . O primeiro argumento define o diretório para listar os arquivos. Na construção deste livro foi criado um diretório

chamado *resources/figs*, onde as figuras utilizadas no livro estão salvas são salvos. Pode-se verificar os arquivos nessa pasta com o seguinte código:

```
my_f <- fs::dir_ls(path = "resources/figs")
print(my_f[1:5])
```

```
R> resources/figs/CAPADigital_DadosFinanceirosR_20180510.jpg
R> resources/figs/Command_view.png
R> resources/figs/ExemploAjuda.png
R> resources/figs/Exemplo_inline_code.png
R> resources/figs/Favicon
```

Observe que nesse diretório encontram-se vários arquivos com extensão *.png*. Destaca-se que também é possível listar os arquivos de forma recursiva, isto é, listar os arquivos de subpastas do endereço original. Para verificar, tente utilizar o seguinte código no seu computador:

```
# list all files recursively
fs::dir_ls(path = getwd(),
           recurse = TRUE)
```

O comando anterior irá listar todos os arquivos existentes na pasta atual e subpastas de trabalho. Dependendo de onde o comando foi executado, pode levar um certo tempo para o término do processo. Caso precisar cancelar a execução, aperte *esc* no teclado.

Para listar pastas (diretórios) do computador, basta utilizar o comando **list.dirs()** ou então a função **fs::dir_ls()** com argumento `type = "directory"`. Veja a seguir.

```
# list directories
my_dirs <- fs::dir_ls(".", type = "directory")
print(my_dirs)
```

```
R> 05-interagindo-computador-internet_files
R> EOCE-Rmd
R> _book
R> backup
R> resources
R> site_libs
```

No caso anterior, o comando lista todos os diretórios do trabalho atual sem recursividade. A saída do comando mostra os diretórios que utilizei para escrever este livro. Isso inclui o diretório de saída do livro (`./_book`), entre diversos outros. Nesse mesmo diretório, encontram-se os capítulos do livro, organizados por arquivos e baseados na linguagem *Quarto*. Para listar somente os arquivos com extensão `.qmd`, utiliza-se o argumento `glob = "*.qmd"` da função **fs**::**dir_ls()** , como a seguir:

```
qmd_files <- fs::dir_ls(".", glob = "*.qmd$")

print(qmd_files)
```

```
R> 00a-prefacio.qmd
R> 00b-agradecimentos.qmd
R> 01-introdução.qmd
R> 02-como-o-r-funciona.qmd
R> 03-rstudio.qmd
R> 04-pacotes.qmd
R> 05-interagindo-computador-internet.qmd
R> 06-importacao-exportacao-dados-locais.qmd
R> 07-importacao-internet.qmd
R> 08-objetos-armazenamento.qmd
R> 09-objetos-basicos.qmd
R> 10-intro-programacao.qmd
R> 99-references.qmd
R> _BemVindo.qmd
R> index.qmd
```

O texto `*.qmd$` orienta o R a procurar todos arquivos que terminam o seu nome com o texto *.Rmd*. Os símbolos `'*''` e `'$'` são operadores específicos para o encontro de padrões em texto em uma linguagem chamada regex (*regular expressions*) e, nesse caso, indicam que o usuário quer encontrar todos arquivos com extensão *.qmd*. O símbolo '*' diz para ignorar qualquer texto anterior a '.qmd' e '$' indica o fim do nome do arquivo. Os arquivos apresentados anteriormente contêm todo o conteúdo deste livro, incluindo este próprio parágrafo!

5.1.2 Apagando Arquivos e Diretórios

A remoção de arquivos é realizada através do comando **file.remove()** ou então **fs**::**file_delete()** :

```
# create temporary file in docs folder
my_file <- '~/MyTemp.csv'
write.csv(x = data.frame(x=1:10),
          file = my_file)

# delete it
fs::file_delete(my_file)
```

Lembre-se que deves ter permissão do seu sistema operacional para apagar um arquivo. Para o nosso caso, o retorno `TRUE` mostra que a operação teve sucesso.

Para deletar diretórios e todos os seus elementos, utilizamos **unlink()** ou então **fs::dir_delete()** :

```
# create temp dir
name_temp <- "TEMP"
fs::dir_create(name_temp)

# fill it with file
my_file <- fs::path(name_temp, "tempfile.csv")
write.csv(x = data.frame(x=1:10),
          file = my_file)

fs::dir_delete(name_temp)
```

A função, neste caso, não retorna nada. Podes checar se o diretório existe com **fs::dir_exists()** :

```
fs::dir_exists(name_temp)
```

```
R>  TEMP
R> FALSE
```

> **Cuidado**
>
> **Tenha muito cuidado** com comandos de remover pastas e arquivos no R, principalmente quando utilizar recursividade, isto é, quanto apagar todas as pastas e arquivos existentes no caminho inputado. Uma execução errada e partes importantes do seu disco rídigo podem ser apagadas, deixando o seu computador inoperável. Saiba que

o R **realmente apaga** os arquivos e não somente manda para a lixeira. Portanto, ao apagar diretórios inteiros, não poderás recuperar os arquivos facilmente.

5.1.3 Utilizando Arquivos e Diretórios Temporários

Toda vez que uma nova sessão do R é inicializada, o programa automaticamente cria uma pasta temporária no seu sistema. É nesse diretório que o R guarda quaisquer arquivos e pastas descartáveis que possam ser necessárias na sua sessão. No momento que a sessão do R é finalizada, tal como quando fechamos o RStudio, as pastas temporárias são removidas.

O endereço do diretório temporário de uma sessão do R é verificado com **tempdir()** ou **fs**::**path_temp()** :

```
my_tempdir <- fs::path_temp()
message(stringr::str_glue('My tempdir is {my_tempdir}'))
```

```
R> My tempdir is /tmp/RtmpFZrnZY
```

O último texto do diretório, neste caso RtmpFZrnZY é aleatóriamente definido e irá trocar a cada nova sessão do R.

A mesma dinâmica é encontrada para nomes de arquivos. Caso queira, por algum motivo, utilizar um nome temporário e aleatório para algum arquivo com extensão *.txt*, utilize **tempfile()** ou **fs**::**file_temp()** e defina o tipo de arquivo com as entradas da função:

```
my_tempfile <- fs::file_temp(ext = '.txt')
message(my_tempfile)
```

```
R> /tmp/RtmpFZrnZY/file4a0458452774.txt
```

Note que o nome do arquivo – file4a0458452774.txt – é totalmente aleatório e mudará a cada chamada da função.

5.1.4 Baixando Arquivos da Internet

O R pode baixar arquivos da Internet diretamente no código. Isso é realizado com a função **download.file()** . Veja o exemplo a seguir, onde baixamos uma planilha de Excel do site da Microsoft para um arquivo temporário:

```
# set link
link_dl <- 'go.microsoft.com/fwlink/?LinkID=521962'
local_file <- fs::file_temp(ext = '.xlsx') # name of local file

download.file(url = link_dl,
              destfile = local_file)
```

O uso de **download.file()** é bastante prático quando se está trabalhando com dados da Internet que são constantemente atualizados. Basta baixar e atualizar o arquivo com dados no início do *script*. Poderíamos continuar a rotina lendo o arquivo baixado e realizando a nossa análise dos dados disponíveis.

Um exemplo nesse caso é a tabela de empresas listadas na bolsa divulgada pela CVM (comissão de valores mobiliários). A tabela está disponível em um arquivo no site. Podemos baixar o arquivo e, logo em seguida, ler os dados.

```
# set destination link and file
my_link <- 'http://dados.cvm.gov.br/dados/CIA_ABERTA/CAD/DADOS/cad_cia_ab
my_destfile <- fs::file_temp(ext = '.csv')

# download file
download.file(my_link,
              destfile = my_destfile,
              mode = "wb")

# read it
df_cvm <- readr::read_csv2(my_destfile,
                      #delim = '\t',
                      locale = locale(encoding = 'Latin1'),
                      col_types = cols())
```

```
R> i Using "','" as decimal and "'.'" as grouping mark. Use `read_delim()` f
```

```
# check available columns
print(names(df_cvm))
```

```
R>  [1] "CNPJ_CIA"           "DENOM_SOCIAL"
R>  [3] "DENOM_COMERC"       "DT_REG"
R>  [5] "DT_CONST"           "DT_CANCEL"
R>  [7] "MOTIVO_CANCEL"      "SIT"
R>  [9] "DT_INI_SIT"         "CD_CVM"
R> [11] "SETOR_ATIV"         "TP_MERC"
R> [13] "CATEG_REG"          "DT_INI_CATEG"
R> [15] "SIT_EMISSOR"        "DT_INI_SIT_EMISSOR"
R> [17] "CONTROLE_ACIONARIO" "TP_ENDER"
R> [19] "LOGRADOURO"         "COMPL"
R> [21] "BAIRRO"             "MUN"
R> [23] "UF"                 "PAIS"
R> [25] "CEP"                "DDD_TEL"
R> [27] "TEL"                "DDD_FAX"
R> [29] "FAX"                "EMAIL"
R> [31] "TP_RESP"            "RESP"
R> [33] "DT_INI_RESP"        "LOGRADOURO_RESP"
R> [35] "COMPL_RESP"         "BAIRRO_RESP"
R> [37] "MUN_RESP"           "UF_RESP"
R> [39] "PAIS_RESP"          "CEP_RESP"
R> [41] "DDD_TEL_RESP"       "TEL_RESP"
R> [43] "DDD_FAX_RESP"       "FAX_RESP"
R> [45] "EMAIL_RESP"         "CNPJ_AUDITOR"
R> [47] "AUDITOR"
```

Existem diversas informações interessantes nestes dados incluindo nome e CNPJ de empresas listadas (ou deslistadas) da bolsa de valores Brasileira. E, mais importante, o arquivo está sempre atualizado. O código anterior estará sempre buscando os dados mais recentes a cada execução.

CAPÍTULO 6

IMPORTAÇÃO E EXPORTAÇÃO DE DADOS LOCAIS

Sem dúvida, a primeira etapa de um *script* de pesquisa é carregar os seus dados em uma sessão do R. Neste capítulo iremos aprender a importar e exportar dados contidos em arquivos locais. Apesar de não ser uma tarefa particularmente difícil, um analista de dados deve entender as diferentes características de cada formato de arquivo e como tirar vantagem deste conhecimento em cada situação. Enquanto algumas facilitam a colaboração e troca de dados, outras podem oferecer um ganho significativo em tempo de execução na leitura e gravação.

Aqui iremos traçar uma lista abrangente com os seguintes formatos e extensões de arquivos:

- Dados delimitados em texto (*csv*);
- Microsoft Excel (*xls*, *xlsx*);
- Arquivos de dados nativos do R (*RData* e *rds*)
- Formato `fst` (*fst*)
- Texto não estruturado (*txt*).

A primeira lição na importação de dados para o R é que o local do arquivo deve ser indicado explicitamente no código. Este endereço é passado para a função que irá ler o arquivo. Veja a definição a seguir:

```
my_file <- 'C:/Data/MyData.csv'
```

Note o uso de barras (/) para designar o diretório do arquivo. Referências relativas também funcionam, tal como em:

```
my_file <- 'Data/MyData.csv'
```

Neste caso, assume-se que na pasta atual de trabalho existe um diretório chamado `Data` e, dentro desse, um arquivo denominado `MyData.csv`. Se o endereço do arquivo é simplesmente o seu nome, assume-se que o mesmo encontra-se na raiz da pasta de trabalho. Para verificar o endereço atual de trabalho, utilize a função **getwd()** .

> **!** Importante
>
> Aqui novamente reforço o uso do *tab* e *autocomplete* do RStudio. É muito mais **fácil e prático** encontrar arquivos do disco rígido do computador usando a navegação via *tab* do que copiar e colar o endereço do seu explorador de arquivos. Para usar, abra aspas no RStudio, coloque o cursor do *mouse* entre as aspas e aperte *tab*.

Um ponto importante aqui é que os **dados serão importados e exportados no R como objetos do tipo** `dataframe`. Isto é, uma tabela contida em um arquivo Excel ou *.csv* se transformará em um objeto do tipo `dataframe` no ambiente de trabalho do R. Quando exportarmos dados, o formato mais comum é esse mesmo tipo de objeto. Convenientemente, `dataframes` são nada mais que tabelas, com linhas e colunas.

Cada coluna do `dataframe` importado terá a sua própria classe, sendo as mais comuns numérica (*numeric*), texto (*character*), fator (*factor*) e data (*Date*). Quando realizando a importação, **é de fundamental importância que os dados sejam representados na classe correta**. Uma vasta quantidade de erros podem ser evitados pela simples checagem das classes das colunas no `dataframe` resultante do processo de importação. Por enquanto somente é necessário entender esta propriedade básica de `dataframes`. Estudaremos esse objeto mais profundamente no capítulo Capítulo 8.

6.1 Pacote introR

Nas seções futuras iremos utilizar o pacote do livro – introR – para carregar diversos exemplos de arquivos. Se você seguiu as instruções da seção *Material Suplementar* localizada no prefácio do livro, já deves ter o pacote instalado. Caso contrário, execute o seguinte código:

```
# install devtools (if not installed)
if (!require(devtools)) install.packages ('devtools')

# install book package
devtools::install_github ('msperlin/introR')
```

Uma vez que você instalou o pacote introR, todos os arquivos de dados usados no livro foram baixados. Podemos verificar os cinco primeiros arquivos disponíveis com o comando **introR**::**data_list()** :

```
# list available data files
print(introR::data_list())
```

```
R>

R> -- Available data files at '/home/msperlin/R/x86_64-pc-linux

R> i CH04_another-funky-csv-file.csv

R> i CH04_example-fst.fst

R> i CH04_example-Rdata.RData

R> i CH04_example-rds.rds

R> i CH04_example-sqlite.SQLite

R> i CH04_example-tsv.tsv

R> i CH04_funky-csv-file.csv

R> i CH04_ibovespa-Excel.xlsx

R> i CH04_ibovespa.csv

R> i CH04_price-and-prejudice.txt

R> i CH04_SP500-Excel.xlsx
```

```
R> i CH04_SP500.csv

R> i CH07_FileWithLatinChar_Latin1.txt

R> i CH07_FileWithLatinChar_UTF-8.txt

R> i CH08_some-stocks-SP500.csv

R> i CH08_wide-example-stocks.csv

R> i CH10_sp500-stocks-long-by-year.csv

R> i CH11_grunfeld.csv

R> i CH11_SP500.csv

R> i CH11_UCI-Credit-Card.csv

R> i EX_B3-stocks.rds

R> i EX_football-br.csv

R> i EX_Ibov_PETR4.csv

R> i EX_ibovespa.rds

R> i EX_SP500-stocks-wide.csv

R> i EX_SP500-stocks-yearly.rds

R> i EX_SP500-stocks.rds

R> i EX_TD-data.rds

R> i EX_TweetsElonMusk.csv

R>

R> v You can get the local path of file using introR::data_path(name_of_file

R> v Example: local_path <- introR::data_path('CH07_FileWithLatinChar_UTF-8.
```

```
R>  [1] "CH04_another-funky-csv-file.csv"
R>  [2] "CH04_example-fst.fst"
R>  [3] "CH04_example-Rdata.RData"
R>  [4] "CH04_example-rds.rds"
R>  [5] "CH04_example-sqlite.SQLite"
R>  [6] "CH04_example-tsv.tsv"
R>  [7] "CH04_funky-csv-file.csv"
R>  [8] "CH04_ibovespa-Excel.xlsx"
R>  [9] "CH04_ibovespa.csv"
R> [10] "CH04_price-and-prejudice.txt"
R> [11] "CH04_SP500-Excel.xlsx"
R> [12] "CH04_SP500.csv"
R> [13] "CH07_FileWithLatinChar_Latin1.txt"
R> [14] "CH07_FileWithLatinChar_UTF-8.txt"
R> [15] "CH08_some-stocks-SP500.csv"
R> [16] "CH08_wide-example-stocks.csv"
R> [17] "CH10_sp500-stocks-long-by-year.csv"
R> [18] "CH11_grunfeld.csv"
R> [19] "CH11_SP500.csv"
R> [20] "CH11_UCI-Credit-Card.csv"
R> [21] "EX_B3-stocks.rds"
R> [22] "EX_football-br.csv"
R> [23] "EX_Ibov_PETR4.csv"
R> [24] "EX_ibovespa.rds"
R> [25] "EX_SP500-stocks-wide.csv"
R> [26] "EX_SP500-stocks-yearly.rds"
R> [27] "EX_SP500-stocks.rds"
R> [28] "EX_TD-data.rds"
R> [29] "EX_TweetsElonMusk.csv"
```

Os arquivos anteriores estão salvos na pasta de instalação dos pacote `introR`. Para ter o caminho completo, basta usar função **introR::data_path()** tendo o nome do arquivo como entrada:

```
# get location of file
my_f <- introR::data_path('CH11_grunfeld.csv')

# print it
print(my_f)
```

A partir de agora iremos usar a função `introR::data_path` para obter o caminho dos arquivos utilizados nos exemplos. Note que, desde que tenha o

pacote `introR` instalado, podes facilmente reproduzir todos os exemplos do livro no seu computador.

6.2 Arquivos *csv*

Considere o arquivo de dados no formato `csv` chamado `'CH04_ibovespa.csv'`, pertencente ao repositório do livro. Vamos copiar o mesmo para a pasta "Meus Documentos" com o uso do tilda (~):

```
# get location of file
my_f <- introR::data_path('CH04_ibovespa.csv')

# copy to ~
file.copy(from = my_f,
          to = '~' )
```

```
R> [1] TRUE
```

```
R> [1] TRUE
```

Caso seja a primeira vez trabalhando com arquivos do tipo *.csv*, sugiro usar o explorador de arquivos do Windows e abrir CH04_ibovespa.csv com qualquer editor de texto instalado, tal como o *Notepad* (veja Figura 6.1. Observe que as primeiras linhas do arquivo definem os nomes das colunas: "ref_date" e "price_close". Conforme notação internacional, as linhas são definidas pela quebra do texto e as colunas pelo uso da vírgula (,).

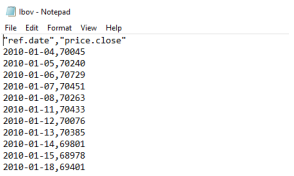


Figura 6.1: Exemplo de arquivo .csv no Notepad

> **O Formato Brasileiro**
>
> Quando trabalhando com dados brasileiros, a notação internacional pode gerar uma confusão desnecessária. Dados locais tendem a usar a vírgula para indicar valores decimais em números, assim como também datas no formato dia/mês/ano. Abaixo apresenta-se algumas

regras de formatação de números e códigos para o caso brasileiro.
Decimal: O decimal no R é definido pelo ponto (.), tal como em `2.5` e não vírgula, como em `2,5`. Esse é o padrão internacional, e a diferença para a notação brasileira gera muita confusão. Em nenhuma situação deve-se utilizar a vírgula como separador de casas decimais. Mesmo quando estiver exportando dados, sempre dê prioridade para o formato internacional.
Caracteres latinos: Devido ao seu padrão internacional, o R apresenta problemas para entender caracteres latinos, tal como cedilha e acentos. Caso possa evitar, não utilize esses tipos de caracteres no código para nomeação de variáveis ou arquivos. Nos objetos de classe texto (`character`), é possível utilizá-los desde que a codificação do objeto esteja correta (*UTF-8* ou *Latin1*). Assim, recomenda-se que o código do R seja escrito na língua inglesa.
Formato das datas: Datas no R são formatadas de acordo com a norma ISO 8601, seguindo o padrão `YYYY-MM-DD`, onde `YYYY` é o ano em quatro números, `MM` é o mês e `DD` é o dia. Por exemplo, uma data em ISO 8601 é 2024-01-10. No Brasil, as datas são formatadas como `DD/MM/YYYY`. Reforçando a regra, sempre dê preferência ao padrão internacional. Vale salientar que a conversão entre um formato e outro é bastante fácil e será apresentada em capítulo futuro.

O conteúdo de CH04_ibovespa.csv é bastante conservador e não será difícil importar o seu conteúdo. Porém, saiba que muitas vezes o arquivo *.csv* vem com informações extras de cabeçalho – o chamado *metadata* – ou diferentes formatações que exigem adaptações. Como sugestão para evitar problemas, antes de prosseguir para a importação de dados em um arquivo *.csv*, abra o arquivo em um editor de texto qualquer e siga os seguintes passos:

1) Verifique a existência de texto antes dos dados e a necessidade de ignorar algumas linhas iniciais. A maioria dos arquivos .csv não contém cabeçalho, porém deves sempre checar. No R, a função de leitura de arquivos *.csv* possui uma opção para ignorar um definido número de linhas antes de começar a leitura do arquivo;

2) Verifique a existência ou não dos nomes das colunas na primeira linha com os dados. Em caso negativo, verifique com o autor qual o nome (e significado) das colunas;

3) Verifique qual o símbolo de separador de colunas. Comumente, seguindo notação internacional, será a vírgula, porém nunca se tem certeza sem checar;

4) Para dados numéricos, verifique o símbolo de decimal, o qual deve ser o ponto (`.`) tal como em `2.5`. Caso necessário, podes ajustar o símbolo na própria função de leitura;

5) Verifique a codificação do arquivo de texto. Normalmente é UTF-8, Latin1 (ISO-8859) ou windows1252. Esses são formatos amplos e devem ser suficientes para a maioria dos idiomas. Sempre que você encontrar símbolos estranhos nas colunas de texto do `dataframe` resultante, o problema é devido a uma diferença na codificação entre o arquivo e o R. Os usuários do Windows podem verificar a codificação de um arquivo de texto abrindo-o no software Notepad++. As informações sobre a codificação estarão disponíveis no canto inferior direito do editor. No entanto, você precisa estar ciente de que o Notepad++ não faz parte da instalação do Windows e pode ser necessário instalá-lo em seu computador. Os usuários de Linux e Mac podem encontrar as mesmas informações em qualquer software editor de texto avançado, como o Kate.

> **!** Importante
>
> Sempre que você encontrar uma estrutura de texto inesperada em um arquivo *.csv*, use os argumentos da função de leitura *csv* para importar as informações corretamente. Repetindo, **nunca modifique dados brutos manualmente**. É muito mais eficiente usar o código R para lidar com diferentes estruturas de arquivos em *.csv*. Pode parecer mais trabalhoso, mas essa política vai economizar muito tempo no futuro, pois, em algumas semanas, você provavelmente esquecerá como limpou manualmente aquele arquivo *.csv* utilizado em pesquisa passada. Com o uso de código para a adaptação da importação de dados, sempre que você precisar atualizar o arquivo de dados, o código irá resolver todos os problemas, automatizando o processo.

6.2.1 Importação de Dados

O R possui uma função nativa chamada **read.csv()** para importar dados de arquivos *.csv*. Porém, esse é um dos muitos casos em que a alternativa do **{tidyverse}** (Wickham 2023c) – **readr**::**read_csv()** – é mais eficiente e mais fácil de trabalhar. Resumindo, **readr**::**read_csv()** lê arquivos mais rapidamente que **read.csv()** , além de usar regras mais inteligentes para definir as classes das colunas importadas.

Este é a primeira vez que usamos um pacote do **{tidyverse}** (Wickham 2023c), neste caso o **{readr}** (Wickham, Hester, e Bryan 2023). Antes de fazer isso, é necessário instalá-lo em sua sessão R. Uma maneira simples de instalar todos os pacotes pertencentes ao **{tidyverse}** (Wickham 2023c) é instalar o módulo de mesmo nome:

```
install.packages('tidyverse')
```

Após executar o código anterior, todos os pacotes **{tidyverse}** (Wickham 2023c) serão instalados em seu computador. Você também deve ter em mente que alguns aspectos dessa instalação podem demorar um pouco. Assim que terminar, carregue o conjunto de pacotes **{tidyverse}** (Wickham 2023c).

```
# load library
library(tidyverse)
```

De volta à importação de dados de arquivos *.csv*, use a função **readr::read_csv()** para carregar o conteúdo do arquivo CH04_ibovespa.csv no R:

```
# set file to read
my_f <- introR::data_path('CH04_ibovespa.csv')

# read data
my_df_ibov <- read_csv(my_f)
```

O conteúdo do arquivo importado é convertido para um objeto do tipo `dataframe` no R. Conforme mencionado no capítulo anterior, cada coluna de um `dataframe` tem uma classe. Podemos verificar as classes de `my_df_ibov` usando a função **dplyr::glimpse()** do pacote **{dplyr}** (Wickham, François, et al. 2023), que também faz parte do **{tidyverse}** (Wickham 2023c):

```
# check content
glimpse(my_df_ibov)
```

```
R> Rows: 3,215
R> Columns: 2
R> $ ref_date    <date> 2010-01-04, 2010-01-05, 2010-01-06, 2~
R> $ price_close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Observe que a coluna de datas (`ref_date`) foi importada como um vetor `Date` e os preços de fechamento como numéricos (`dbl`, precisão dupla). Isso é exatamente o que esperávamos. Internamente, a função **readr**::**read_csv()** identifica as classes das colunas de acordo com seu conteúdo.

Observe também como o código anterior apresentou a mensagem "Parsed with column specification: …". Essa mensagem mostra como a função identifica as classes das colunas lendo as primeiras 1000 linhas do arquivo. Regras inteligentes tentam prever a classe com base no conteúdo importado. Podemos usar essas informações em nosso próprio código copiando o texto e atribuindo-o a uma variável:

```
# set cols from readr import message
my_cols <- cols(
  price_close = col_double(),
  ref_date = col_date(format = "")
)

# read file with readr::read_csv
my_df_ibov <- read_csv(my_f,
                       col_types = my_cols)
```

Como um exercício, vamos importar os mesmos dados, porém usando a classe `character` (texto) para colunas `ref_date`:

```
# set cols from readr import message
my_cols <- cols(
  price_close = col_double(),
  ref_date = col_character()
)

# read file with readr::read_csv
my_df_ibov <- read_csv(my_f,
                       col_types = my_cols)

# check content
glimpse(my_df_ibov)
```

```
R> Rows: 3,215
R> Columns: 2
R> $ ref_date    <chr> "2010-01-04", "2010-01-05", "2010-01-0~
```

```
R> $ price_close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Como esperado, a coluna de datas – `ref_date` – agora foi importada como texto. Assim, o uso de **readr**::**read_csv()** pode ser resumido em duas etapas: 1) leia o arquivo sem argumentos em **readr**::**read_csv()** ; 2) copie o texto das classes de coluna padrão da mensagem de saída e adicione como entrada `col_types`. O conjunto de passos anterior é suficiente para a grande maioria dos casos. O uso da mensagem com as classes das colunas é particularmente útil quando o arquivo importado tem várias colunas e a definição manual de cada classe exige muita digitação.

Uma alternativa mais prática no uso do **readr**::**read_csv()** é confiar na heurística da função e usar a definição padrão das colunas automaticamente. Para isto, basta definir a entrada `col_types` como `cols()`. Veja a seguir:

```
# read file with readr::read_csv
my_df_ibov <- read_csv(my_f,
                       col_types = cols())
```

Agora, vamos estudar um caso mais anormal de arquivo *.csv*. No pacote do livro temos um arquivo chamado `CH04_funky-csv-file.csv` onde:

- o cabeçalho possui texto com informações dos dados;
- o arquivo usará a vírgula como decimal;
- o texto do arquivo conterá caracteres latinos.

As primeiras 10 linhas dos arquivos contém o seguinte conteúdo:

```
R> Example of funky file:
R> - columns separated by ";"
R> - decimal points as ","
R>
R> Data build in 2022-12-28
R> Origin: www.funkysite.com.br
R>
R> ID;Race;Age;Sex;Hour;IQ;Height;Died
R> 001;White;80;Male;00:00:00;92;68;FALSE
R> 002;Hispanic;25;Female;00:00:00;99;68;TRUE
```

Note a existência do cabeçalho até linha de número 7 e as colunas sendo separadas pela semi-vírgula (“;”).

Ao importar os dados com opções padrões (e erradas), teremos o resultado a seguir:

```
df_funky <- readr::read_csv(my_f,
                        col_types = readr::cols())

dplyr::glimpse(df_funky)
```

```
R> Rows: 2
R> Columns: 1
R> $ `Example of funky file:` <chr> "- columns separated by \~
```

Claramente a importação deu errado, com a emissão de diversas mensagens de *warning*. Para resolver, utilizamos o seguinte código, estruturando todas as particularidades do arquivo:

```
df_not_funky <- readr::read_delim(file = my_f,
                              skip = 7, # how many lines do skip
                              delim = ';', # column separator
                              col_types = readr::cols(), # column types
                              locale = readr::locale(decimal_mark = ',')# lo
)

dplyr::glimpse(df_not_funky)
```

```
R> Rows: 100
R> Columns: 8
R> $ ID     <chr> "001", "002", "003", "004", "005", "006", "~
R> $ Race   <chr> "White", "Hispanic", "Asian", "White", "Whi~
R> $ Age    <dbl> 80, 25, 25, 64, 76, 89, 33, 61, 23, 59, 80,~
R> $ Sex    <chr> "Male", "Female", "Male", "Male", "Female",~
R> $ Hour   <time> 00:00:00, 00:00:00, 00:00:00, 00:00:00, 00~
R> $ IQ     <dbl> 92, 99, 98, 105, 109, 84, 109, 109, 99, 126~
R> $ Height <dbl> 68, 68, 69, 69, 67, 73, 65, 72, 70, 66, 63,~
R> $ Died   <lgl> FALSE, TRUE, TRUE, TRUE, FALSE, TRUE, FALSE~
```

Veja que agora os dados foram corretamente importados, com as classes corretas das colunas. Para isso, usamos a função alternativa `readr::read_delim`. O pacote **{readr}** (Wickham, Hester, e Bryan 2023) também possui várias outras funções para situações específicas de importação. Caso a função **readr::read_csv()** não resolva o seu problema na leitura de algum arquivo de dados estruturado em texto, certamente outra função desse pacote resolverá.

6.2.2 Exportação de Dados

Para exportar tabelas em um arquivo *.csv*, basta utilizar a função **readr::write_csv()** . No próximo exemplo iremos criar dados artificiais, salvar em um dataframe e exportar para um arquivo *.csv* temporário. Veja a seguir:

```
library(readr)

# set number of observations
N <- 100

# create dataframe with random data
my_df <- data.frame(y = runif(N),
                    z = rep('a', N))

# write to file
f_out <- tempfile(fileext = '.csv')
write_csv(x = my_df, file = f_out)
```

No exemplo anterior, salvamos o `dataframe` chamado `my_df` para o arquivo file4b7d4563d9e1.csv, localizado na pasta temporária do computador. Podemos verificar o arquivo importando o seu conteúdo:

```
my_df <- read_csv(f_out,
                  col_types = cols(y = col_double(),
                                   z = col_character() ) )
print(head(my_df))
```

```
R> # A tibble: 6 x 2
R>       y z
R>   <dbl> <chr>
R> 1 0.422 a
R> 2 0.987 a
R> 3 0.873 a
R> 4 0.674 a
R> 5 0.536 a
R> 6 0.552 a
```

O resultado está conforme o esperado, um dataframe com duas colunas, a primeira com números e a segunda com texto.

Note que toda exportação com função **readr**::**write_csv()** irá ser formatada, por padrão, com a notação internacional. Caso quiser algo diferentes, verifique as opções disponíveis na função **readr**::**write_delim()** , a qual é muito mais flexível.

6.3 Arquivos *Excel* (*xls* e *xlsx*)

Em Finanças e Economia, é bastante comum encontrarmos dados salvos em arquivos do tipo Microsoft Excel, com extensão *.xls* ou *.xlsx*. Apesar de não ser um formato de armazenamento de dados eficiente, esse é um programa de planilhas bastante popular devido às suas funcionalidades. É muito comum que informações sejam armazenadas e distribuídas dessa forma. Por exemplo, dados históricos do Tesouro Direto são disponibilizados como arquivos *.xls* no site do tesouro nacional. A CVM (Comissão de Valores Mobiliários) e ANBIMA (Associação Brasileira das Entidades dos Mercados Financeiro e de Capitais) também tem preferência por esse tipo de formato em alguns dados publicados em seu *site*.

A desvantagem de usar arquivos do Excel para armazenar dados é sua baixa portabilidade e o maior tempo necessário para leitura e gravação. Isso pode não ser um problema para tabelas pequenas, mas ao lidar com um grande volume de dados, o uso de arquivos Excel é frustrante e não aconselhável. Se possível, evite o uso de arquivos do Excel em seu ciclo de trabalho.

6.3.1 Importação de Dados

O R não possui uma função nativa para importar dados do Excel e, portanto, deve-se instalar e utilizar certos pacotes para realizar essa operação. Existem diversas opções, porém, os principais pacotes são **{XLConnect}** (Mirai Solutions GmbH 2023), **{xlsx}** (Dragulescu e Arendt 2020), **{readxl}** (Wickham e Bryan 2023) e **{tidyxl}** (Garmonsway 2023).

Apesar de os pacotes anteriores terem objetivos semelhantes, cada um tem suas peculiaridades. Caso a leitura de arquivos do Excel seja algo importante no seu trabalho, aconselho-o fortemente a estudar as diferenças entre esses pacotes. Por exemplo, pacote **{tidyxl}** (Garmonsway 2023) permite a leitura de dados não-estruturados de um arquivo Excel, enquanto **{XLConnect}** (Mirai Solutions GmbH 2023) possibilita a abertura de uma conexão ativa entre o R e o Excel, onde o usuário pode transmitir

dados entre um e o outro, formatar células, criar gráficos no Excel e muito mais.

Nesta seção, daremos prioridade para funções do pacote **{readxl}** (Wickham e Bryan 2023), o qual é um dos mais fáceis e diretos de se utilizar, além de não necessitar de outros softwares instalados (tal como o *Java*). Para instalar o referido pacote, basta utilizar a função **install.packages()** :

```
install.packages('readxl')
```

Imagine agora a existência de um arquivo chamado `CH04_ibovespa-Excel.xlsx` que contenha os mesmos dados do Ibovespa que importamos na seção anterior. A importação das informações contidas nesse arquivo para o R será realizada através da função **readxl::read_excel()** :

```
library(readxl)
library(dplyr)

# set file
my_f <- introR::data_path("CH04_ibovespa-Excel.xlsx")

# read xlsx into dataframe
my_df <- read_excel(my_f, sheet = 'Sheet1')

# glimpse contents
glimpse(my_df)
```

```
R> Rows: 3,215
R> Columns: 2
R> $ ref_date    <dttm> 2010-01-04, 2010-01-05, 2010-01-06, 2~
R> $ price_close <dbl> 70045, 70240, 70729, 70451, 70263, 704~
```

Observe que, nesse caso, as datas já foram importadas com a formatação correta na classe `dttm` (*datetime*). Essa é uma vantagem ao utilizar arquivos do Excel: a classe dos dados do arquivo original é levada em conta no momento da importação. O lado negativo desse formato é a baixa portabilidade dos dados e o maior tempo necessário para a execução da importação. Como regra geral, dados importados do Excel apresentarão um tempo de carregamento mais alto do que dados importados de arquivos *.csv*.

6.3.2 Exportação de Dados

A exportação para arquivo Excel também é fácil. Assim como para a importação, não existe uma função nativa do R que execute esse procedimento. Para tal tarefa, temos pacotes **{xlsx}** (Dragulescu e Arendt 2020) e **{writexl}** (Ooms 2023). Uma diferença aqui é que o pacote **{xlsx}** (Dragulescu e Arendt 2020) oferece mais funcionalidade mas exige a instalação do Java JDK no sistema operacional. No caso do Windows, basta visitar o site do Java e instalar o software na versão 64 bits (opção *Windows Off-line (64 bits)*). Logo após, instale o pacote **{xlsx}** (Dragulescu e Arendt 2020) normalmente no R com o comando `install.packages('xlsx')`.

Para fins de ilustração de uso, vamos utilizar o pacote **{writexl}** (Ooms 2023)

```
# set number of rows
N <- 50

# create random dataframe
my_df <- data.frame(y = seq(1,N),
                    z = rep('a',N))

# write to xlsx
f_out <- fs::file_temp(ext = '.xlsx')

writexl::write_xlsx(x = my_df, path = f_out)
```

6.4 Formato *.RData* e *.rds*

O R possui dois formatos nativos para salvar objetos de sua área de trabalho para um arquivo local com extensão *RData* ou *rds*. O grande benefício, em ambos os casos, é que o arquivo resultante é compacto e o seu acesso é muito rápido. A desvantagem é que os dados perdem portabilidade para outros programas. A diferença entre um formato e outro é que arquivos *RData* podem salvar mais de um objeto, enquanto o formato *.rds* salva apenas um. Na prática, porém, essa **não é uma restrição forte**. No R existe um objeto do tipo *lista* que incorpora outros. Portanto, caso salvarmos uma *lista* em um arquivo *.rds*, podemos gravar no disco quantos objetos forem necessários.

6.4.1 Importação de Dados

Para carregar os dados de um aquivo `RData`, utilizamos a função **load()** :

```
# set a object
my_x <- 1:100

# set temp name of RData file
my_file <- introR::data_path('CH04_example-Rdata.RData')

# load it
load(file = my_file)
```

O arquivo CH04_example-Rdata.RData possui dois objetos, `my_x` e `my_y`, os quais se tornam disponíveis na área de trabalho depois da chamada de **load()** .

O processo de importação para arquivos *.rds* é muito semelhante. A diferença é no uso da função **readr**::**read_rds()** :

```
# set file path
my_file <- introR::data_path('CH04_example-rds.rds')

# load content into workspace
my_x <- readr::read_rds(file = my_file)
```

Comparando o código entre o uso de arquivos `.RData` e `.rds`, note que um benefício no uso de `.rds` é a explícita definição do objeto na área de trabalho. Isto é, o conteúdo de `my_file` em **readr**::**read_rds()** é explicitamente salvo em `my_x`. Quando usamos a função **load()** , no código não fica claro qual o nome do objeto que foi importado. Isso é particularmente inconveniente quando é necessário modificar o nome do objeto importado.

> **!** Importante
>
> Como sugestão, dê preferência ao uso do formato *.rds*, o qual deve resultar em códigos mais transparentes. A diferença de velocidade de acesso e gravação entre um e outro é mínima. O benefício de importar vários objetos em um mesmo arquivo com o formato `RData` torna-se irrelevante quando no uso de objetos do tipo lista, os quais podem incorporar outros objetos no seu conteúdo.

6.4.2 Exportação de Dados

Para criar um novo arquivo *RData*, utilizamos a função **save()** . Veja o exemplo a seguir, onde criamos um arquivo *RData* com dois objetos:

```
# set vars
my_x <- 1:100
my_y <- 1:100

# write to RData
my_file <- fs::file_temp(ext = '.RData')
save(list = c('my_x', 'my_y'),
     file = my_file)
```

Podemos verificar a existência do arquivo::

```
fs::file_exists(my_file)
```

```
R> /tmp/Rtmp4gwvZV/file4b7d1219b97a.RData
R>                                     TRUE
```

Observe que o arquivo file4b7d1219b97a.RData está disponível na pasta temporária.

Já para arquivos *.rds*, salvamos o objeto com função **readr**::**write_rds()** :

```
# set data and file
my_x <- 1:100
my_file <- fs::file_temp(ext = ".rds")

# save as .rds
readr::write_rds(my_x, my_file)

# read it
my_x2 <- readr::read_rds(my_file)

# test equality
print(identical(my_x, my_x2))
```

```
R> [1] TRUE
```

O comando **identical()** testa a igualdade entre os objetos e, como esperado, verificamos que `my_x` e `my_x2` são exatamente iguais.

6.5 Arquivos *fst*

Pacote **{fst}** (Klik 2022) foi especialmente desenhado para possibilitar a gravação e leitura de dados tabulares de forma rápida e com mínimo uso do espaço no disco. O uso deste formato é particularmente benéfico quando se está trabalhando com volumosas bases de dados em computadores potentes. **O grande truque do formato *fst* é usar todos núcleos do computador para importar e exportar dados**, enquanto todos os demais formatos se utilizam de apenas um. Como logo veremos, o ganho em velocidade é bastante significativo.

6.5.1 Importação de Dados

O uso do formato *fst* é bastante simples. Utilizamos a função **fst**::**read_fst()** para ler arquivos:

```
library(fst)

my_file <- introR::data_path('CH04_example-fst.fst')
my_df <- read_fst(my_file)

dplyr::glimpse(my_df)
```

```
R> Rows: 100
R> Columns: 8
R> $ ID     <chr> "001", "002", "003", "004", "005", "006", "~
R> $ Race   <fct> Black, White, Hispanic, Black, White, White~
R> $ Age    <int> 33, 35, 23, 87, 65, 51, 58, 67, 22, 52, 52,~
R> $ Sex    <fct> Male, Female, Male, Female, Male, Male, Fem~
R> $ Hour   <dbl> 0.00000000, 0.00000000, 0.00000000, 0.00000~
R> $ IQ     <dbl> 108, 108, 85, 106, 92, 92, 88, 100, 86, 80,~
R> $ Height <dbl> 72, 63, 77, 72, 71, 74, 64, 69, 63, 72, 70,~
R> $ Died   <lgl> FALSE, TRUE, TRUE, FALSE, FALSE, TRUE, TRUE~
```

Assim como para os demais casos, os dados estão disponíveis na área de trabalho após a importação.

6.5.2 Exportação de Dados

Utilizamos a função **fst**::**write_fst()** para gravar arquivos no formato *fst*, :

```
library(fst)

# create dataframe
N <- 1000
my_file <- fs::file_temp(ext = '.fst')
my_df <- data.frame(x = runif(N))

# write to fst
write_fst(x = my_df, path = my_file)
```

6.5.3 Testando o Tempo de Execução do Formato *fst*

Como um teste do potencial do pacote **{fst}** (Klik 2022), a seguir vamos cronometrar o tempo de leitura e gravação entre *fst* e *rds* para um dataframe com grande quantidade de dados: 5,000,000 linhas e 2 colunas. Iremos reportar também o tamanho do arquivo resultante.

```
library(fst)

# set number of rows
N <- 5000000

# create random dfs
my_df <- data.frame(y = seq(1,N),
                    z = rep('a',N))

# set files
my_file_rds <- fs::file_temp(ext = ".rds")
my_file_fst <- fs::file_temp(ext = ".fst")

# test write
time_write_rds <- system.time(readr::write_rds(my_df, my_file_rds ))
time_write_fst <- system.time(fst::write_fst(my_df, my_file_fst ))

# test read
```

```
time_read_rds <- system.time(readr::read_rds(my_file_rds))
time_read_fst <- system.time(fst::read_fst(my_file_fst))

# test file size (MB)
file_size_rds <- file.size(my_file_rds)/1000000
file_size_fst <- file.size(my_file_fst)/1000000
```

Após a execução, vamos verificar o resultado:

```
# results
my_formats <- c('.rds', '.fst')
results_read <- c(time_read_rds[3], time_read_fst[3])
results_write<- c(time_write_rds[3], time_write_fst[3])
results_file_size <- c(file_size_rds , file_size_fst)

# print text
my_text <- paste0('\nTime to WRITE dataframe with ',
                  my_formats, ': ',
                  results_write, ' seconds', collapse = '')
message(my_text)
```

```
R>
R> Time to WRITE dataframe with .rds: 0.753 seconds
R> Time to WRITE dataframe with .fst: 0.114999999999999 seconds
```

```
my_text <- paste0('\nTime to READ dataframe with ',
                  my_formats, ': ',
                  results_read, ' seconds', collapse = '')
message(my_text)
```

```
R>
R> Time to READ dataframe with .rds: 0.877 seconds
R> Time to READ dataframe with .fst: 0.147 seconds
```

```
my_text <- paste0('\nResulting FILE SIZE for ',
                  my_formats, ': ',
                  results_file_size, ' MBs', collapse = '')
message(my_text)
```

```
R>
R> Resulting FILE SIZE for .rds: 65.000177 MBs
R> Resulting FILE SIZE for .fst: 14.791938 MBs
```

A diferença é gritante! O formato *fst* não somente lê e grava com mais rapidez mas o arquivo resultante também é menor. Porém, saiba que os resultados anteriores foram compilados em um computador com 16 núcleos. É possível que a diferença de tempo para um computador mais modesto não seja tão significativa.

> Dica
>
> Devido ao uso de todos os núcleos do computador, o formato *fst* é altamente recomendado quando estiver trabalhando com dados volumosos em um computador potente. Não somente os arquivos resultantes serão menores, mas o processo de gravação e leitura será consideravelmente mais rápido.

> Dica
>
> Devido ao uso de todos os núcleos do computador, o formato *fst* é altamente recomendado quando estiver trabalhando com dados volumosos em um computador potente. Não somente os arquivos resultantes serão menores, mas o processo de gravação e leitura será consideravelmente mais rápido.

6.6 Dados Não-Estruturados e Outros Formatos

Os pacotes e formatos anteriores são suficientes para resolver o problema de importação de dados na grande maioria das situações. Apesar disso, vale destacar que o R possui outras funções específicas para diferentes formatos. Isso inclui arquivos exportados de outros softwares, tal como SPSS, Matlab, entre vários outros. Se esse for o seu caso, sugiro um estudo aprofundado do pacote **{foreign}** (R Core Team 2023a).

Em alguns casos nos deparamos com dados armazenados de uma forma não estruturada, tal como um texto qualquer. Pode-se importar o conteúdo de um arquivo de texto linha por linha através da função **readr**::**read_lines()** . Veja o exemplo a seguir, onde importamos o conteúdo inteiro do livro *Pride and Prejudice*:

```
# set file to read
my_f <- introR::data_path('CH04_price-and-prejudice.txt')

# read file line by line
my_txt <- readr::read_lines(my_f)

# print 50 characters of first fifteen lines
print(stringr::str_sub(string = my_txt[1:15],
                       start = 1,
                       end = 50))
```

```
R>  [1] "                                [Illustration:"
R>  [2] ""
R>  [3] "                                 GEORGE ALLEN"
R>  [4] "                                   PUBLISHER"
R>  [5] ""
R>  [6] "                             156 CHARING CROSS ROAD"
R>  [7] "                                    LONDON"
R>  [8] ""
R>  [9] "                                 RUSKIN HOUSE"
R> [10] "                                       ]"
R> [11] ""
R> [12] "                                [Illustration:"
R> [13] ""
R> [14] "                  _Reading Jane's Letters._      _Cha"
R> [15] "                                       ]"
```

Neste exemplo, arquivo CH04_price-and-prejudice.txt contém todo o conteúdo do livro *Pride and Prejudice* de Jane Austen, disponível gratuitamente pelo projeto Gutenberg. Importamos todo o conteúdo do arquivo como um vetor de texto denominado `my_txt`. Cada elemento de `my_txt` é uma linha do arquivo do texto original. Com base nisso, podemos calcular o número de linhas do livro e o número de vezes que o nome `'Bennet'`, um dos protagonistas da história, aparece no texto:

```
# count number of lines
n_lines <- length(my_txt)

# set target text
name_to_search <- 'Bennet'

```

```
# set function for counting words
fct_count_bennet <- function(str_in, target_text) {

  require(stringr)


  n_words <- length(str_locate_all(string = str_in,
                                   pattern = target_text)[[1]])

  return(n_words)
}

# use fct for all lines of Pride and Prejudice
n_times <- sum(sapply(X = my_txt,
                      FUN = fct_count_bennet,
                      target_text = name_to_search))

# print results
my_msg <- paste0('The number of lines found in the file is ',
                 n_lines, '.\n',
                 'The word "', name_to_search, '" appears ',
                 n_times, ' times in the book.')
message(my_msg)
```

```
R> The number of lines found in the file is 14529.
R> The word "Bennet" appears 696 times in the book.
```

No exemplo, mais uma vez usamos **sapply()** . Neste caso, a função nos permitiu usar outra função para cada elemento de `my_txt`. Neste caso, procuramos e contamos o número de vezes que a palavra "Bennet" foi encontrada no texto. Observe que poderíamos simplesmente mudar `name_to_search` por qualquer outro nome, caso quiséssemos.

6.6.1 Exportando de Dados Não-Estruturados

Em algumas situações, é necessário exportar algum tipo de texto para um arquivo. Por exemplo: quando se precisa salvar o registro de um procedimento em um arquivo de texto; ou quando se precisa gravar informações em um formato específico não suportado pelo R. Esse procedimento é bastante simples. Junto à função **readr**::**write_lines()** , basta indicar um arquivo de texto para a saída com o argumento `file`. Veja a seguir:

```
# set file
my_f <- tempfile(fileext = '.txt')

# set some string
my_text <- paste0('Today is ', Sys.Date(), '\n',
                  'Tomorrow is ', Sys.Date()+1)

# save string to file
readr::write_lines(x = my_text, file = my_f, append = FALSE)
```

No exemplo, criamos um objeto de texto com uma mensagem sobre a data atual e gravamos a saída no arquivo temporário file4b7d267d3c74.txt. Podemos checar o resultado com a função **readr::read_lines()** :

```
print(readr::read_lines(my_f))
```

```
R> [1] "Today is 2024-01-10"    "Tomorrow is 2024-01-11"
```

6.7 Selecionando o Formato

Após entendermos a forma de salvar e carregar dados de arquivos locais em diferentes formatos, é importante discutirmos sobre a escolha do formato. O usuário deve levar em conta três pontos nessa decisão:

- velocidade de importação e exportação;
- tamanho do arquivo resultante;
- compatibilidade com outros programas e sistemas.

Na grande maioria das situações, o uso de arquivos *csv* satisfaz esses quesitos. Ele nada mais é do que um arquivo de texto que pode ser aberto, visualizado e importado em qualquer programa. Desse modo, fica muito fácil compartilhar dados compatíveis com outros usuários. Além disso, o tamanho de arquivos *csv* geralmente não é exagerado em computadores modernos. Caso necessário, podes compactar o arquivo .csv usando o programa *7zip*, por exemplo, o qual irá diminuir consideravelmente o tamanho do arquivo. Por esses motivos, **o uso de arquivos *csv* para importações e exportações é preferível na grande maioria das situações**.

Existem casos, porém, onde a velocidade de importação e exportação pode fazer diferença. Caso abrir mão de portabilidade não faça diferença ao projeto, o formato *rds* é ótimo e prático. Se este não foi suficiente,

então a melhor alternativa é partir para o *fst*, o qual usa maior parte do *hardware* do computador para importar os dados. Como sugestão, caso puder, apenas **evite o formato do Excel**, o qual é o menos eficiente de todos.

6.8 Exercícios

CAPÍTULO 7

IMPORTAÇÃO DE DADOS VIA PACOTES

Uma das grandes vantagens de se utilizar o R é a quantidade de dados que podem ser importados através da internet. Isso é especialmente prático pois uma base de dados pode ser atualizada através de um simples comando, evitando o tedioso trabalho de coleta manual. Ao usarmos pacotes para importar dados, esta etapa da pesquisa se torna reproduzível e mais rápida, facilitando o compartilhamento e futura execução do nosso código.

Neste capítulo vou descrever e dar exemplos de importação de dados para os mais importantes e estáveis pacotes especializados na importação de para dados financeiros e econômicos no Brasil e exterior. A lista inclui:

{GetQuandlData} (M. S. Perlin 2023) (M. S. Perlin 2023) Importa dados econômicos e financeiros do vasto repositório de dados da plataforma *Quandl*.

{yfR} (M. Perlin 2023b) (R-BatchGetSymbols?) Importa dados de preços diários de ações e índices do *Yahoo Finance*.

{GetTDData} (M. Perlin 2023a) (M. Perlin 2023a) Importa dados de títulos de dívida pública do Brasil diretamente do site do Tesouro Direto.

{**GetBCBData**} **(M. Perlin 2022) (M. Perlin 2022)** Importa dados do grande repositório de séries temporais do Banco Central do Brasil, local obrigatório para qualquer economista que trabalha com dados.

{**GetDFPData2**} **(M. Perlin e Kirch 2023) (M. Perlin e Kirch 2023)** Importa dados do sistema DFP – Demonstrativos Financeiros Padronizados – de empresas negociadas na B3, a bolsa Brasileira. O repositório inclui documentos financeiros tal como o balanço patrimonial, demonstrativos de resultados, entre vários outros.

{**GetFREData**} **(M. Perlin e Kirch 2022) (M. Perlin e Kirch 2022)** Importa dados do sistema FRE – Formulário de Referência – da bolsa Brasileira. Esta inclui diversos eventos e informações corporativas tal como composição do conselho e diretoria, remuneração dos conselheiros, entre outras.

7.1 Pacote {yfR} (M. Perlin 2023b)

Pacote **{yfR}** (M. Perlin 2023b) faz a comunicação do R com os dados financeiros disponíveis no *Yahoo Finance*. Essa gigantesca base de dados inclui valores agregados de preços e volumes negociados de ações na B3 e outras bolsas internacionais na frequência diária. Tudo que se precisa saber para acessar a base de dados são os identificadores das ações (*tickers*) e um período de tempo.

Os diferenciais do **{yfR}** (M. Perlin 2023b) são:

Limpeza e organização: todos os dados financeiros de diferentes *tickers* são mantidos no mesmo `dataframe`, facilitando a análise futura com as ferramentas do **{tidyverse}** (Wickham 2023c).

Controle de erros de importação: todos erros de *download* são registrados na saída do programa. Caso uma ação em particular não exista no *Yahoo Finance*, esta será ignorada e apenas as demais disponíveis serão retornadas na saída do código;

Comparação de datas a um *benchmark*: os dados de ativos individuais são comparados com dados disponíveis para um ativo *benchmark*, geralmente um índice de mercado. Caso o número de datas faltantes seja maior que um determinado limite imposto pelo usuário, a ação é retirada do `dataframe` final.

Uso de sistema de cache: no momento de acesso aos dados, os mesmos são salvos localmente no computador do usuário e são persistentes para cada sessão. Caso o usuário requisitar os mesmos dados na mesma sessão

do R, o sistema de cache será utilizado. Se os dados desejados não estão disponíveis no cache, a função irá comparar e baixar apenas as informações que faltam. Isso aumenta significativamente a velocidade de acesso aos dados, ao mesmo tempo em que minimiza o uso da conexão a Internet;

> **!** Importante
>
> Desde versão 2.6 (2020-11-22) de **{yfR}** (M. Perlin 2023b) a pasta *default* de cache do **{yfR}** (M. Perlin 2023b) se localiza no diretório temporário da sessão do R. Assim, o cache é persistente apenas para a sessão do usuário. Esta mudança foi motivada por quebras estruturais nos dados do *Yahoo Finance*, onde os dados passados registrados em cache não mais estavam corretos devido a eventos coporativos. O usuário, porém, pode trocar a pasta de cache usando a entrada `cache.folder`.

Acesso a *tickers* em índices de mercado: O pacote inclui funções para baixar a composição dos índices Ibovespa, SP500 e FTSE100. Isso facilita a importação de dados para uma grande quantidade de ações. Podes, por exemplo, baixar cotações de todas as ações que fazem parte de certo índice.

Processamento paralelo: Caso o usuário estiver baixando um grande volume de dados do *Yahoo Finance*, uma opção para execução paralela está disponível. Isto é, ao invés de usar apenas um núcleo na requisição dos dados, usamos vários ao mesmo tempo. O efeito prático é, dependendo do número de núcleos do computador, uma diminuição significativa no tempo total de importação.

Flexibilidade de formato: O pacote também oferece funções para modificar o formato dos dados. Caso o usuário deseje uma saída do `dataframe` no formato largo, onde *tickers* são colunas e as linhas os preços/retornos, basta chamar função `BatchGetSymbols::reshape.wide`. Da mesma forma, uma transformação temporal também é possível. Se o usuário desejar dados na frequência semanal, mensal ou anual, basta indicar na entrada `freq.data` da função.

Como exemplo de uso, vamos baixar dados financeiros referentes a quatro ações no último ano (360 dias) usando a função de mesmo nome do pacote. Os *tickers* de cada ação podem ser encontrados nos próprios sites do *Yahoo Finance*. Note que adicionamos texto `.SA` a cada um deles. Essa é uma notação específica do site e vale para qualquer ação Brasileira.

Na chamada da função **{yfR}** (M. Perlin 2023b), utilizamos um valor de `0.95` (95%) para o *input* `thresh.bad.data` e `'^BVSP'` para `bench.ticker`. Isso faz com que a função compare as datas obtidas para cada ativo em relação ao nosso *benchmark*, o índice Ibovespa, cujo *ticker* no *Yahoo Finance* é `^BVSP`. Se, durante o processo de importação, uma ação individual não apresenta mais de 95% de casos válidos em relação ao *benchmark*, esta é retirada da saída.

```
library(BatchGetSymbols)
library(dplyr)

# set tickers
my_tickers <- c('PETR4.SA', 'CIEL3.SA',
                'GGBR4.SA', 'GOAU4.SA')

# set dates and other inputs
first_date <- Sys.Date()-360
last_date <- Sys.Date()
thresh_bad_data <- 0.95   # sets percent threshold for bad data
bench_ticker <- '^BVSP'   # set benchmark as ibovespa

l_out <- BatchGetSymbols(tickers = my_tickers,
                         first.date = first_date,
                         last.date = last_date,
                         bench.ticker = bench_ticker,
                         thresh.bad.data = thresh_bad_data)
```

```
R> Warning: `BatchGetSymbols()` was deprecated in BatchGetSymbols
R> 2.6.4.
R> i Please use `yfR::yf_get()` instead.
R> i 2022-05-01: Package BatchGetSymbols will soon be replaced
R>   by yfR.  More details about the change is available at
R>   github <<www.github.com/msperlin/yfR> You can install yfR
R>   by executing:
R>
R> remotes::install_github('msperlin/yfR')
```

A saída de **{yfR}** (M. Perlin 2023b) é um objeto do tipo lista, ainda não visto no livro. Por enquanto, tudo que precisas saber é que uma lista é um objeto flexível, acomodando outros objetos em sua composição. O acesso a cada elemento de uma lista pode ser feito pelo operador `$`. No capítulo @ref(classe-estrutura) iremos estudar melhor esta classe de objetos.

> **!** Importante
>
> Note que as entradas da função `BatchGetSymbols::BatchGetSymbols` usam o "." em seus nomes, tal como `thresh.bad.data`, e `bench.ticker`, enquanto o livro está escrito usando o traço baixo (_), tal como `thresh_bad_data`, e `bench_ticker`. Esta diferença pode resultar em problemas se, na falta de atenção, o usuário trocar um pelo outro. Como regra, procure dar prioridade para o uso de traço baixo nos nomes de objetos. Infelizmente algumas funções escritas no passado acabaram ficando com a estrutura antiga e, para não prejudicar os usuários, os nomes das entradas foram mantidos.

Voltando ao nosso exemplo, função **{yfR}** (M. Perlin 2023b) retorna uma lista com dois elementos: um `dataframe` com o resultado do processo de importação – `df_control` – e outro `dataframe` com os dados das ações – `df_tickers`. Vamos checar o conteúdo do primeiro dataframe.

```
# print result of download process
print(l_out$df.control)
```

```
R> # A tibble: 4 x 6
R>   ticker   src   download.status total.obs
R>   <chr>    <chr> <chr>                <int>
R> 1 PETR4.SA yahoo OK                   244
R> 2 CIEL3.SA yahoo OK                   244
R> 3 GGBR4.SA yahoo OK                   244
R> 4 GOAU4.SA yahoo OK                   244
R> # i 2 more variables: perc.benchmark.dates <dbl>,
R> #   threshold.decision <chr>
```

Objeto `df.control` mostra que todos *tickers* foram válidos, com um total de 244 observações para cada ativo. Note que as datas batem 100% com o Ibovespa (coluna `perc.benchmark.dates`).

Quanto aos dados financeiros, esses estão contidos em `l_out$df.tickers`:

```
# print df_tickers
glimpse(l_out$df.tickers)
```

```
R> Rows: 976
R> Columns: 10
R> $ price.open          <dbl> 24.41, 24.11, 25.84, 25.00, 25~
```

```
R> $ price.high          <dbl> 24.50, 25.62, 26.03, 25.95, 26~
R> $ price.low           <dbl> 23.83, 24.11, 25.00, 24.79, 25~
R> $ price.close         <dbl> 24.04, 25.52, 25.07, 25.89, 26~
R> $ volume              <dbl> 31973700, 91797500, 69692900, ~
R> $ price.adjusted      <dbl> 18.62798, 19.77480, 19.42610, ~
R> $ ref.date            <date> 2023-01-16, 2023-01-17, 2023-~
R> $ ticker              <chr> "PETR4.SA", "PETR4.SA", "PETR4~
R> $ ret.adjusted.prices <dbl> NA, 0.061564070, -0.017633247,~
R> $ ret.closing.prices  <dbl> NA, 0.061564039, -0.017633258,~
```

Como esperado, a informação sobre preços, retornos e volumes está lá, com as devidas classes de colunas: *dbl* (*double*) para valores numéricos e *date* para as datas. Observe que uma coluna chamada `ticker` também está incluída. Essa indica em que linhas da tabela os dados de uma ação começam e terminam. Mais tarde, no capítulo @ref(limpando-estruturando), usaremos essa coluna para fazer diversos cálculos para cada ação.

7.1.1 Baixando Dados da Composição do Ibovespa

Outra função útil do pacote é `BatchGetSymbols::GetIbovStocks`, a qual importa a composição atual do índice Ibovespa diretamente do site da B3. Esse índice é um termômetro do mercado local e as ações que o compõem são selecionadas devido sua alta negociabilidade. Portanto, sequenciando o uso de `GetIbovStocks` e **{yfR}** (M. Perlin 2023b), podemos facilmente baixar uma volumosa quantidade de dados de ações para o mercado Brasileiro. Considere o seguinte fragmento de código, onde realizamos essa operação:

```
library(BatchGetSymbols)

# set tickers
df_ibov <- GetIbovStocks()
my_tickers <- paste0(df_ibov$tickers,'.SA')


# set dates and other inputs
first_date <- Sys.Date()-30
last_date <- Sys.Date()
thresh_bad_data <- 0.95   # sets percent threshold for bad data
bench_ticker <- '^BVSP'   # set benchmark as ibovespa
cache_folder <- 'data/BGS_Cache' # set folder for cache
```

```
14
15
16 l_out <- BatchGetSymbols(tickers = my_tickers,
17                          first.date = first_date,
18                          last.date = last_date,
19                          bench.ticker = bench_ticker,
20                          thresh.bad.data = thresh_bad_data,
21                          cache.folder = cache_folder)
```

Note que utilizamos a função **paste0()** para adicionar o texto `'.SA'` para cada *ticker* em `df_ibov$tickers`. A saída do código anterior não foi mostrada para não encher páginas e páginas com as mensagens do processamento. Destaco que, caso necessário, poderíamos facilmente exportar os dados em `l_out` para um arquivo `.rds` e futuramente carregá-los localmente para realizar algum tipo de análise.

> Cuidado
>
> Saiba que **os preços do Yahoo Finance não são ajustados a dividendos**. O ajuste realizado pelo sistema é apenas para desdobramentos das ações. Isso significa que, ao olhar séries de preços em um longo período, existe um viés de retorno para baixo. Ao comparar com outro software que faça o ajustamento dos preços por dividendos, verás uma grande diferença na rentabilidade total das ações. Como regra, em uma pesquisa formal, **evite usar dados de ações individuais no Yahoo Finance para períodos longos**. A excessão é para índices financeiros, tal como o Ibovespa, onde os dados do Yahoo Finance são bastante confiáveis uma vez que índices não sofrem os mesmos ajustamentos que ações individuais.

7.2 Pacote {GetTDData} (M. Perlin 2023a)

Arquivos com informações sobre preços e retornos de títulos emitidos pelo governo brasileiro podem ser baixados manualmente no site do Tesouro Nacional. O tesouro direto é um tipo especial de mercado onde pessoa física pode comprar e vender dívida pública. Os contratos de dívida vendidos na plataforma são bastante populares devido a atratividade das taxas de retorno e a alta liquidez oferecida ao investidor comum.

Pacote **{GetTDData}** (M. Perlin 2023a) importa os dados das planilhas em Excel do site do Tesouro Nacional e os organiza. O resultado é um `dataframe` com dados empilhados. Como exemplo, vamos baixar dados de um título prefixado do tipo LTN com vencimento em 2021-01-01. Esse é o tipo de contrato de dívida mais simples que o governo brasileiro emite, não pagando nenhum cupom[1] durante sua validade e, na data de vencimento, retorna 1.000 R$ ao comprador. Para baixar os dados da internet, basta usar o código a seguir:

```
library(GetTDData)

asset_codes <- 'LTN'   # Identifier of assets
maturity <- '010121'  # Maturity date as string (ddmmyy)

# download and read files
df_TD <- td_get(asset_codes)
```

```
R>

R> -- Downloading TD files

R> i Downloading LTN_2005.xls

R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2005.xls' is found, with size 252.

R> i Downloading LTN_2006.xls

R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2006.xls' is found, with size 288.

R> i Downloading LTN_2007.xls

R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2007.xls' is found, with size 260.

R> i Downloading LTN_2008.xls

R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2008.xls' is found, with size 259.

R> i Downloading LTN_2009.xls

R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2009.xls' is found, with size 191.

R> i Downloading LTN_2010.xls
```

[1] O cupom é um pagamento intermediário pago periodicamente durante a validade do contrato financeiro.

```
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2010.xls' is found, with size 148.
R> i Downloading LTN_2011.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2011.xls' is found, with size 161.
R> i Downloading LTN_2012.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2012.xls' is found, with size 144.
R> i Downloading LTN_2013.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2013.xls' is found, with size 147.
R> i Downloading LTN_2014.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2014.xls' is found, with size 147.
R> i Downloading LTN_2015.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2015.xls' is found, with size 142.
R> i Downloading LTN_2016.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2016.xls' is found, with size 176.
R> i Downloading LTN_2017.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2017.xls' is found, with size 174.
R> i Downloading LTN_2018.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2018.xls' is found, with size 174.
R> i Downloading LTN_2019.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2019.xls' is found, with size 177.
R> i Downloading LTN_2020.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2020.xls' is found, with size 176.
R> i Downloading LTN_2021.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2021.xls' is found, with size 175.
R> i Downloading LTN_2022.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2022.xls' is found, with size 175.
```

```
R> i Downloading LTN_2023.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2023.xls' is found, with size 146.
R> i Downloading LTN_2024.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2024.xls' is found, with size 20.5
R>
R> -- Checking files
R> v Found 20 files
R>
R> -- Reading files
R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2005.xls'
R> v    Reading Sheet LTN 040105
R> v    Reading Sheet LTN 010405
R> v    Reading Sheet LTN 010705
R> v    Reading Sheet LTN 011005
R> v    Reading Sheet LTN 010106
R> v    Reading Sheet LTN 010406
R> v    Reading Sheet LTN 010706
R> v    Reading Sheet LTN 011006
R> v    Reading Sheet LTN 010107
R> v    Reading Sheet LTN 010407
R> v    Reading Sheet LTN 010707
R> v    Reading Sheet LTN 010108
R> v    Reading Sheet LTN 010708
R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2006.xls'
R> v    Reading Sheet LTN 010406
```

```
R> v    Reading Sheet LTN 010706

R> v    Reading Sheet LTN 011006

R> v    Reading Sheet LTN 010107

R> v    Reading Sheet LTN 010407

R> v    Reading Sheet LTN 010707

R> v    Reading Sheet LTN 011007

R> v    Reading Sheet LTN 010108

R> v    Reading Sheet LTN 010408

R> v    Reading Sheet LTN 010708

R> v    Reading Sheet LTN 010109

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2007.xls'

R> v    Reading Sheet LTN 010407

R> v    Reading Sheet LTN 010707

R> v    Reading Sheet LTN 011007

R> v    Reading Sheet LTN 010108

R> v    Reading Sheet LTN 010408

R> v    Reading Sheet LTN 010708

R> v    Reading Sheet LTN 011008

R> v    Reading Sheet LTN 010109

R> v    Reading Sheet LTN 010709

R> v    Reading Sheet LTN 011009

R> v    Reading Sheet LTN 010110

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2008.xls'

R> v    Reading Sheet LTN 010408

R> v    Reading Sheet LTN 010708
```

```
R> v    Reading Sheet LTN 011008

R> v    Reading Sheet LTN 010109

R> v    Reading Sheet LTN 010409

R> v    Reading Sheet LTN 010709

R> v    Reading Sheet LTN 011009

R> v    Reading Sheet LTN 010110

R> v    Reading Sheet LTN 010710

R> v    Reading Sheet LTN 010111

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2009.xls'

R> v    Reading Sheet LTN 010409

R> v    Reading Sheet LTN 010709

R> v    Reading Sheet LTN 011009

R> v    Reading Sheet LTN 010110

R> v    Reading Sheet LTN 010710

R> v    Reading Sheet LTN 010111

R> v    Reading Sheet LTN 010112

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2010.xls'

R> v    Reading Sheet LTN 010710

R> v    Reading Sheet LTN 010111

R> v    Reading Sheet LTN 010711

R> v    Reading Sheet LTN 010112

R> v    Reading Sheet LTN 010113

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2011.xls'

R> v    Reading Sheet LTN 010711

R> v    Reading Sheet LTN 010112
```

```
R> v    Reading Sheet LTN 010113

R> v    Reading Sheet LTN 010114

R> v    Reading Sheet LTN 010115

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2012.xls'

R> v    Reading Sheet LTN 010113

R> v    Reading Sheet LTN 010114

R> v    Reading Sheet LTN 010115

R> v    Reading Sheet LTN 010116

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2013.xls'

R> v    Reading Sheet LTN 010114

R> v    Reading Sheet LTN 010115

R> v    Reading Sheet LTN 010116

R> v    Reading Sheet LTN 010117

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2014.xls'

R> v    Reading Sheet LTN 010115

R> v    Reading Sheet LTN 010116

R> v    Reading Sheet LTN 010117

R> v    Reading Sheet LTN 010118

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2015.xls'

R> v    Reading Sheet LTN 010116

R> v    Reading Sheet LTN 010117

R> v    Reading Sheet LTN 010118

R> v    Reading Sheet LTN 010121

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2016.xls'

R> v    Reading Sheet LTN 010117
```

```
R> v    Reading Sheet LTN 010118

R> v    Reading Sheet LTN 010119

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010123

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2017.xls'

R> v    Reading Sheet LTN 010118

R> v    Reading Sheet LTN 010119

R> v    Reading Sheet LTN 010120

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010123

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2018.xls'

R> v    Reading Sheet LTN 010119

R> v    Reading Sheet LTN 010120

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010125

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2019.xls'

R> v    Reading Sheet LTN 010120

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010122

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010125

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2020.xls'

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010122
```

```
R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2021.xls'

R> v    Reading Sheet LTN 010122

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2022.xls'

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> v    Reading Sheet LTN 010129

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2023.xls'

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> v    Reading Sheet LTN 010129

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2024.xls'

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> v    Reading Sheet LTN 010129
```

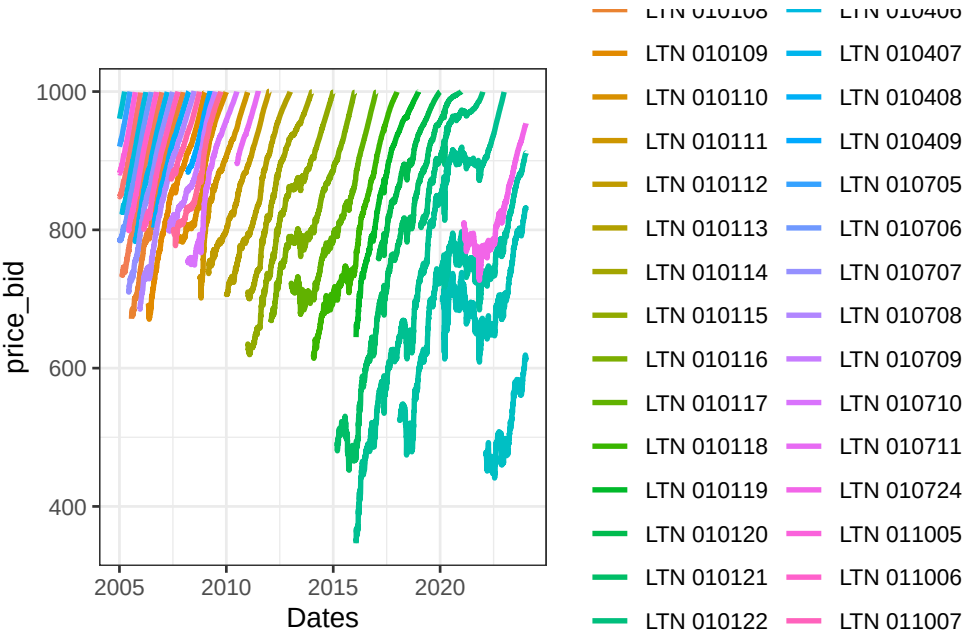
Vamos checar o conteúdo do dataframe:

```
# check content
glimpse(df_TD)
```

```
R> Rows: 24,465
R> Columns: 5
R> $ ref_date   <date> 2005-01-03, 2005-01-03, 2005-01-04, 20~
R> $ yield_bid  <dbl> 0.1774, 0.1836, 0.1839, 0.1841, 0.1842,~
R> $ price_bid  <dbl> 1000.00, 960.66, 961.24, 961.85, 962.48~
R> $ asset_code <chr> "LTN 040105", "LTN 010405", "LTN 010405~
R> $ matur_date <date> 2005-01-04, 2005-04-01, 2005-04-01, 20~
```

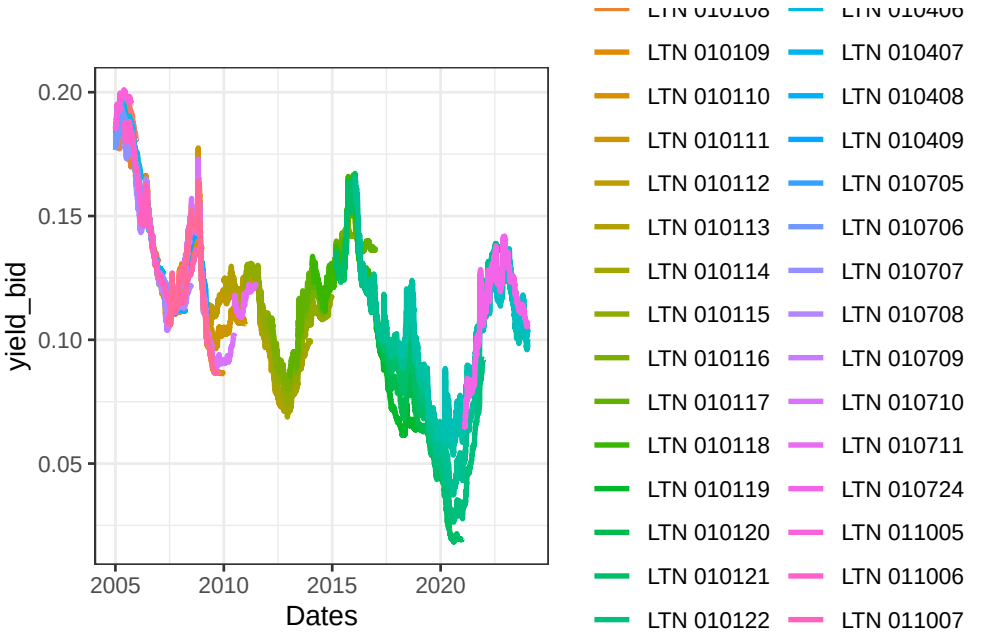
Temos informações sobre data de referência (`ref.date`), retorno contratado (`yield.bid`), preço do contrato na data (`price.bid`), nome do contrato (`asset.code`) e dia de maturidade (`matur.date`). No gráfico a seguir checamos os dados:

```
R> Warning: Using `size` aesthetic for lines was deprecated in ggplot2
R> 3.4.0.
R> i Please use `linewidth` instead.
```



Como esperado de um título de dívida pré-fixado, os preços possuem uma tendência positiva ao longo do tempo, chegando ao valor esperado de

1000 R$ no vencimento em 2021-01-01. Podemos também visualizar as mudanças do *yield* do título:



Os retornos do título tiveram forte queda ao longo dos anos. Este resultado é esperado pois o juros do mercado – taxa SELIC – caiu bastante nos últimos cinco anos.

As funções do **{GetTDData}** (M. Perlin 2023a) também funcionam com vários argumentos como `asset.codes` e `maturity`. Suponhamos que desejamos visualizar todos os preços de todos os prazos disponíveis para títulos do tipo LTN a partir de 2010. Tudo o que precisamos fazer é adicionar o valor `NULL` ao argumento `maturity` e filtrar as datas:

```
library(GetTDData)

asset_codes <- 'LTN'    # Name of asset
maturity <- NULL       # = NULL, downloads all maturities

# download data
df_TD <- td_get(asset_codes)
```

```
R>

R> -- Downloading TD files
```

```
R> i Downloading LTN_2005.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2006.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2007.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2008.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2009.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2010.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2011.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2012.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2013.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2014.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2015.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2016.xls

R> v    Found file in folder, skipping it.

R> i Downloading LTN_2017.xls
```

```
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2018.xls
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2019.xls
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2020.xls
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2021.xls
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2022.xls
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2023.xls
R> v    Found file in folder, skipping it.
R> i Downloading LTN_2024.xls
R> v    '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2024.xls' is found, with size 20.5
R>
R> -- Checking files
R> v Found 20 files
R>
R> -- Reading files
R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2005.xls'
R> v    Reading Sheet LTN 040105
R> v    Reading Sheet LTN 010405
R> v    Reading Sheet LTN 010705
R> v    Reading Sheet LTN 011005
```

```
R> v    Reading Sheet LTN 010106

R> v    Reading Sheet LTN 010406

R> v    Reading Sheet LTN 010706

R> v    Reading Sheet LTN 011006

R> v    Reading Sheet LTN 010107

R> v    Reading Sheet LTN 010407

R> v    Reading Sheet LTN 010707

R> v    Reading Sheet LTN 010108

R> v    Reading Sheet LTN 010708

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2006.xls'

R> v    Reading Sheet LTN 010406

R> v    Reading Sheet LTN 010706

R> v    Reading Sheet LTN 011006

R> v    Reading Sheet LTN 010107

R> v    Reading Sheet LTN 010407

R> v    Reading Sheet LTN 010707

R> v    Reading Sheet LTN 011007

R> v    Reading Sheet LTN 010108

R> v    Reading Sheet LTN 010408

R> v    Reading Sheet LTN 010708

R> v    Reading Sheet LTN 010109

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2007.xls'

R> v    Reading Sheet LTN 010407

R> v    Reading Sheet LTN 010707

R> v    Reading Sheet LTN 011007
```

```
R> v    Reading Sheet LTN 010108

R> v    Reading Sheet LTN 010408

R> v    Reading Sheet LTN 010708

R> v    Reading Sheet LTN 011008

R> v    Reading Sheet LTN 010109

R> v    Reading Sheet LTN 010709

R> v    Reading Sheet LTN 011009

R> v    Reading Sheet LTN 010110

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2008.xls'

R> v    Reading Sheet LTN 010408

R> v    Reading Sheet LTN 010708

R> v    Reading Sheet LTN 011008

R> v    Reading Sheet LTN 010109

R> v    Reading Sheet LTN 010409

R> v    Reading Sheet LTN 010709

R> v    Reading Sheet LTN 011009

R> v    Reading Sheet LTN 010110

R> v    Reading Sheet LTN 010710

R> v    Reading Sheet LTN 010111

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2009.xls'

R> v    Reading Sheet LTN 010409

R> v    Reading Sheet LTN 010709

R> v    Reading Sheet LTN 011009

R> v    Reading Sheet LTN 010110

R> v    Reading Sheet LTN 010710
```

```
R> v    Reading Sheet LTN 010111

R> v    Reading Sheet LTN 010112

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2010.xls'

R> v    Reading Sheet LTN 010710

R> v    Reading Sheet LTN 010111

R> v    Reading Sheet LTN 010711

R> v    Reading Sheet LTN 010112

R> v    Reading Sheet LTN 010113

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2011.xls'

R> v    Reading Sheet LTN 010711

R> v    Reading Sheet LTN 010112

R> v    Reading Sheet LTN 010113

R> v    Reading Sheet LTN 010114

R> v    Reading Sheet LTN 010115

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2012.xls'

R> v    Reading Sheet LTN 010113

R> v    Reading Sheet LTN 010114

R> v    Reading Sheet LTN 010115

R> v    Reading Sheet LTN 010116

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2013.xls'

R> v    Reading Sheet LTN 010114

R> v    Reading Sheet LTN 010115

R> v    Reading Sheet LTN 010116

R> v    Reading Sheet LTN 010117

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2014.xls'
```

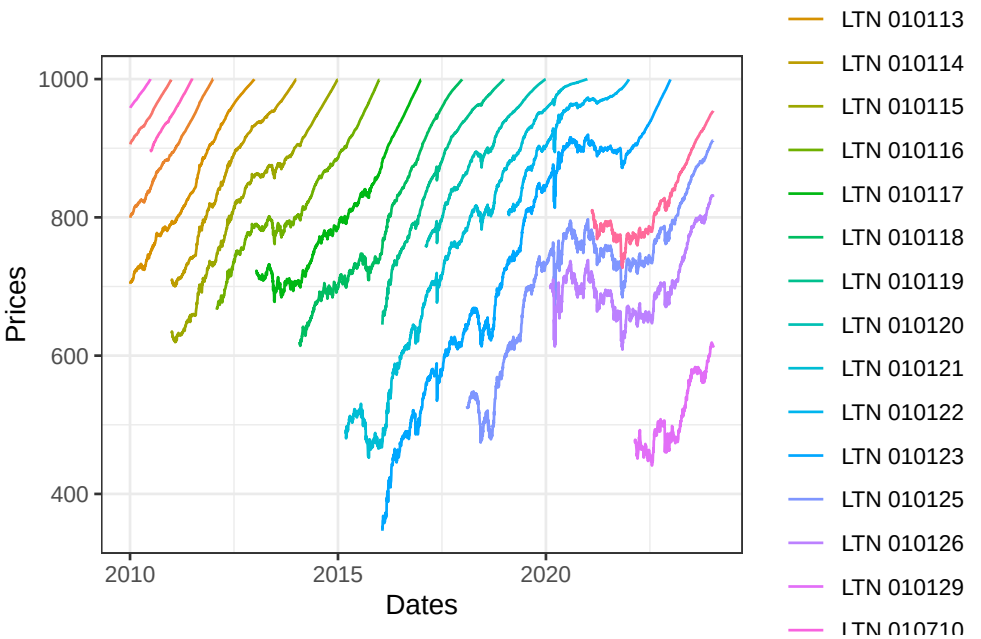
```
R> v    Reading Sheet LTN 010115

R> v    Reading Sheet LTN 010116

R> v    Reading Sheet LTN 010117

R> v    Reading Sheet LTN 010118

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2015.xls'

R> v    Reading Sheet LTN 010116

R> v    Reading Sheet LTN 010117

R> v    Reading Sheet LTN 010118

R> v    Reading Sheet LTN 010121

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2016.xls'

R> v    Reading Sheet LTN 010117

R> v    Reading Sheet LTN 010118

R> v    Reading Sheet LTN 010119

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010123

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2017.xls'

R> v    Reading Sheet LTN 010118

R> v    Reading Sheet LTN 010119

R> v    Reading Sheet LTN 010120

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010123

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2018.xls'

R> v    Reading Sheet LTN 010119

R> v    Reading Sheet LTN 010120

R> v    Reading Sheet LTN 010121
```

```
R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010125

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2019.xls'

R> v    Reading Sheet LTN 010120

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010122

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010125

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2020.xls'

R> v    Reading Sheet LTN 010121

R> v    Reading Sheet LTN 010122

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2021.xls'

R> v    Reading Sheet LTN 010122

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2022.xls'

R> v    Reading Sheet LTN 010123

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126
```

```
R> v    Reading Sheet LTN 010129

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2023.xls'

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> v    Reading Sheet LTN 010129

R> i Reading '/tmp/Rtmp9PmJB1/td-files/LTN/LTN_2024.xls'

R> v    Reading Sheet LTN 010724

R> v    Reading Sheet LTN 010125

R> v    Reading Sheet LTN 010126

R> v    Reading Sheet LTN 010129
```

```r
# remove data prior to 2010
df_TD <- dplyr::filter(df_TD,
                       ref_date >= as.Date('2010-01-01'))
```

Após a importação das informações, plotamos os preços dos diferentes ativos:



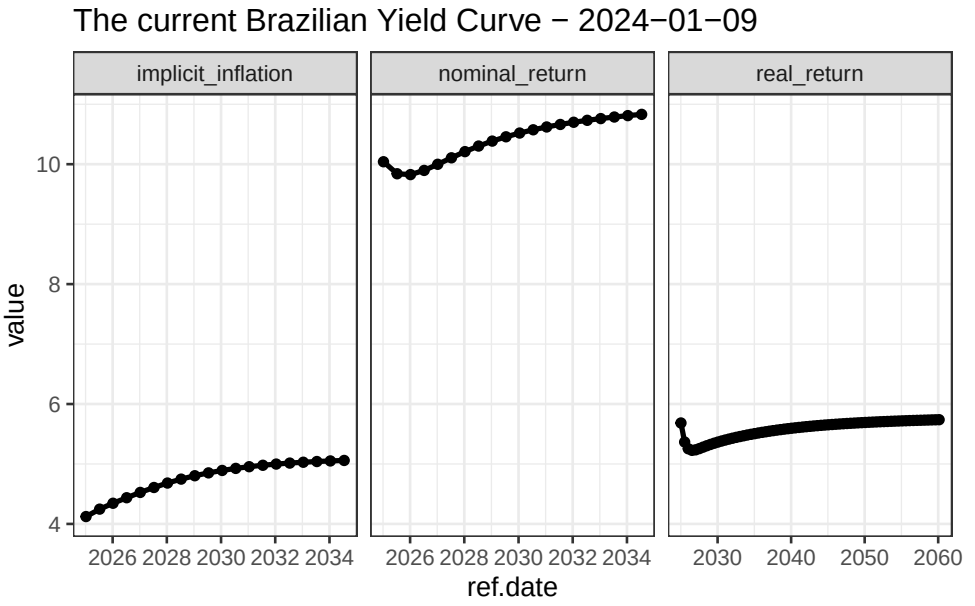
Note como todos contratos do tipo LTN terminam com valor R$ 1.000 em sua data de expiração e possuem uma dinâmica linear de crescimento de preço ao longo do tempo.

Outra funcionalidade do pacote **{GetTDData}** (M. Perlin 2023a) é o acesso a curva de juros atual do sistema financeiro brasileiro diretamente do site da Anbima. Para isso, basta utilizar a função `get.yield.curve`:

```
library(GetTDData)

# get yield curve
df_yield <- get.yield.curve()

# check result
dplyr::glimpse(df_yield)
```

```
R> Rows: 111
R> Columns: 5
R> $ n.biz.days   <dbl> 252, 378, 504, 630, 756, 882, 1008, 1~
R> $ type         <chr> "real_return", "real_return", "real_r~
R> $ value        <dbl> 5.6837, 5.3658, 5.2534, 5.2275, 5.236~
R> $ ref.date     <date> 2025-01-07, 2025-07-10, 2026-01-06, ~
R> $ current.date <date> 2024-01-09, 2024-01-09, 2024-01-09, ~
```

Os dados incluem a curva de juros nominal, juros real e da inflação. Para melhor visualizar as informações, vamos plotá-las em um gráfico:



A curva de juros é uma ferramente utilizada no mercado financeiro com o propósito de representar graficamente a expectatica do mercado sobre juros futuro. Baseada nos preços dos títulos públicos, calcula-se e extrapola-se o juros implícito para cada período futuro. Uma curva ascendente, o formato esperado, indica que é mais caro (maior o juro) tomar dinheiro emprestado no longo prazo.

7.3 Pacote {GetBCBData} (M. Perlin 2022)

O Banco Central Brasileiro (BCB) disponibiliza em seu Sistema de Séries Temporais (SGS) uma vasta quantidade de tabelas relativas a economia do Brasil. Mais importante, estas tabelas são atualizadas constantemente e o acesso é gratuito e sem necessidade de registro.

Como um exemplo, vamos usar o pacote para estudar a inadimplência de crédito no sistema financeiro Brasileiro. O primeiro passo no uso de **{GetBCBData}** (M. Perlin 2022) é procurar o símbolo da série de interesse. Acessando o sistema de séries temporais do BCB, vemos que o código identificador para o percentual total de inandimplência no Brasil é 21082.

No código, basta indicar a série de interesse e o período de tempo desejado:

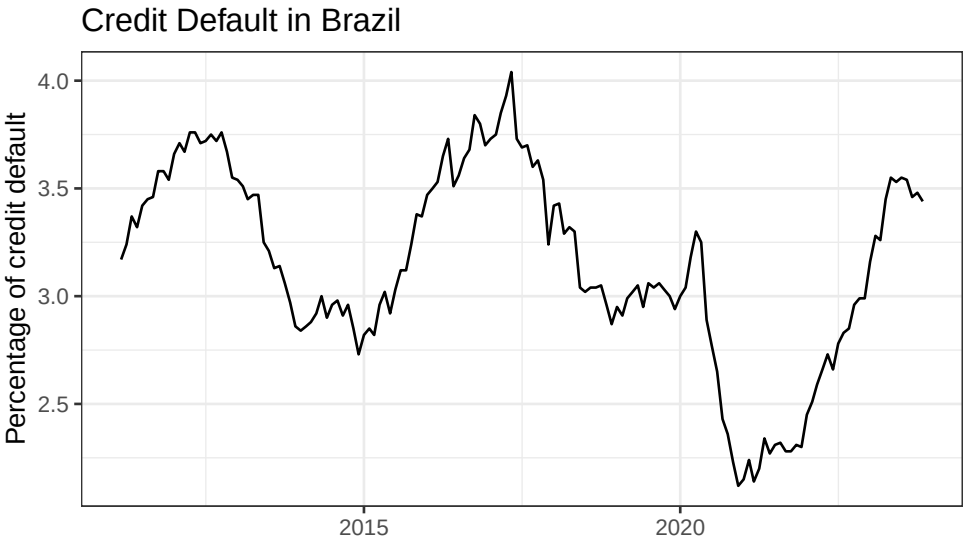
```
library(GetBCBData)
library(dplyr)

# set ids and dates
id_series <- c(perc_default = 21082)
first_date = '2010-01-01'

# get series from bcb
df_cred <- gbcbd_get_series(id = id_series,
                            first.date = first_date,
                            last.date = Sys.Date(),
                            use.memoise = FALSE)
```

```
R>
R> Fetching perc_default [21082] from BCB-SGS from Online API
R>   Found 153 observations
```

```
# check it
glimpse(df_cred)
```

```
R> Rows: 153
R> Columns: 4
R> $ ref.date    <date> 2011-03-01, 2011-04-01, 2011-05-01, 2~
R> $ value       <dbl> 3.17, 3.24, 3.37, 3.32, 3.42, 3.45, 3.~
R> $ id.num      <dbl> 21082, 21082, 21082, 21082, 21082, 210~
R> $ series.name <chr> "perc_default", "perc_default", "perc_~
```

Note que indicamos o nome da coluna na própria definição da entrada `id`. Assim, coluna `series.name` toma o nome de `perc.default`. Esta configuração é importante pois irá diferenciar os dados no caso da importação de diversas séries diferentes. O gráfico apresentado a seguir mostra o valor da série no tempo:



Como podemos ver, a percentagem de inadimplência aumentou a partir de 2015. Para ter uma idéia mais clara do problema, vamos incluir no gráfico a percentagem para pessoa física e pessoa jurídica. Olhando novamente o sistema do BCB, vemos que o símbolos de interesse são 21083 e 21084, respectivamente. O próximo código baixa os dados das duas séries.

```
# set ids
id.series <- c(credit_default_people = 21083,
```

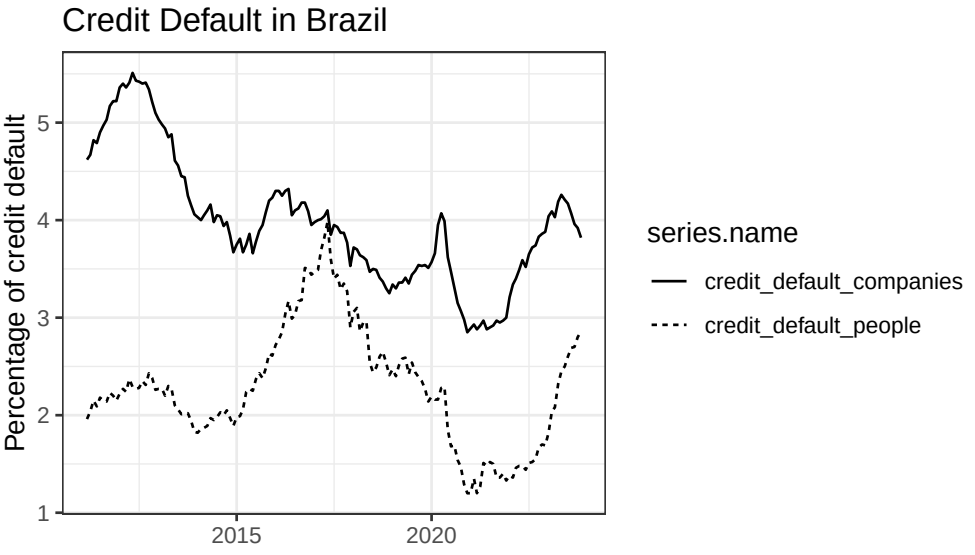
```
                credit_default_companies = 21084)
first.date = '2010-01-01'

# get series from bcb
df_cred <- gbcbd_get_series(id = id.series,
                            first.date = first.date,
                            last.date = Sys.Date(),
                            use.memoise = FALSE)
```

```
R>
R> Fetching credit_default_people [21083] from BCB-SGS from Online API
R>   Found 153 observations
R> Fetching credit_default_companies [21084] from BCB-SGS from Online API
R>   Found 153 observations
```

```
# check output
glimpse(df_cred)
```

```
R> Rows: 306
R> Columns: 4
R> $ ref.date    <date> 2011-03-01, 2011-04-01, 2011-05-01, 2~
R> $ value       <dbl> 1.96, 2.04, 2.15, 2.09, 2.18, 2.15, 2.~
R> $ id.num      <dbl> 21083, 21083, 21083, 21083, 21083, 210~
R> $ series.name <chr> "credit_default_people", "credit_defau~
```

A diferença na saída do código anterior é que agora temos duas séries temporais empilhadas no mesmo `dataframe`. Partimos então para a visualização das séries



Source: SGS − BCB (by GetBCBData)

Como podemos ver, a inadimplência de crédito para pessoa física aumentou muito mais do que a para pessoa jurídica (empresas) nos últimos anos. Poderiámos, facilmente, integrar o código anterior para uma análise mais completa dos dados em algum problema de pesquisa.

> **!** Importante
>
> O sistema BCB-SGS é local obrigatório para qualquer economista sério. A quantidade e variedade de dados é imensa. Podes usar os dados do sistema para automatizar qualquer tipo de relatório econômico.

7.4 Pacote {GetDFPData2} (M. Perlin e Kirch 2023)

Pacote **{GetDFPData2}** (M. Perlin e Kirch 2023) (M. Perlin e Kirch 2023) é uma evolução do pacote `GetDFPData` (**R-GetDFPData?**) e fornece uma interface aberta para todas as demonstrações financeiras distribuídas pela B3 e pela CVM nos sistemas DFP (dados anuais) e ITR (dados trimestrais). Ele não só faz o *download* dos dados, mas também ajusta à inflação e torna as tabelas prontas para pesquisa dentro de um formato tabular. Os diferenciais do pacote em relação a outros distribuidores de dados comerciais são: livre acesso, facilidade para baixar dados em larga escala e a variedade de dados disponíveis.

> **!** Importante
>
> Dados históricos completos e atualizados a partir de 2010 do DFP e ITR estão disponibilizados na seção Data do meu site pessoal. Estes dados são atualizados anualmente.

O ponto de partida no uso de **{GetDFPData2}** (M. Perlin e Kirch 2023) é baixar informações atuais sobre empresas disponíveis. O acesso a tabela é possível com a função `get_info_companies`:

```
library(GetDFPData2)

# get info for companies in B3
df_info <- get_info_companies()
```

```
R> Fetching info on B3 companies

R>  Found cache file. Loading data..

R>  Got 2595 lines for 2461 companies [Actives = 699 Inactives = 1774]
```

```
# check it
names(df_info)
```

```
R>  [1] "CNPJ"                "DENOM_SOCIAL"
R>  [3] "DENOM_COMERC"        "DT_REG"
R>  [5] "DT_CONST"            "DT_CANCEL"
R>  [7] "MOTIVO_CANCEL"       "SIT_REG"
R>  [9] "DT_INI_SIT"          "CD_CVM"
R> [11] "SETOR_ATIV"          "TP_MERC"
R> [13] "CATEG_REG"           "DT_INI_CATEG"
R> [15] "SIT_EMISSOR"         "DT_INI_SIT_EMISSOR"
R> [17] "CONTROLE_ACIONARIO" "TP_ENDER"
R> [19] "LOGRADOURO"          "COMPL"
R> [21] "BAIRRO"              "MUN"
R> [23] "UF"                  "PAIS"
R> [25] "CEP"                 "DDD_TEL"
R> [27] "TEL"                 "DDD_FAX"
R> [29] "FAX"                 "EMAIL"
R> [31] "TP_RESP"             "RESP"
R> [33] "DT_INI_RESP"         "LOGRADOURO_RESP"
```

```
R> [35] "COMPL_RESP"          "BAIRRO_RESP"
R> [37] "MUN_RESP"            "UF_RESP"
R> [39] "PAIS_RESP"           "CEP_RESP"
R> [41] "DDD_TEL_RESP"        "TEL_RESP"
R> [43] "DDD_FAX_RESP"        "FAX_RESP"
R> [45] "EMAIL_RESP"          "CNPJ_AUDITOR"
R> [47] "AUDITOR"
```

Essa tabela disponibiliza os identificadores numéricos das empresas, setores de atividades, atual segmento de governança, *tickers* negociados na bolsa e situação atual (ativa ou não). O número atual de empresas ativas e inativas, a partir de 2024-01-10, está disponível na coluna `SIT_REG`. Observa-se 781 empresas ativas e 1811 canceladas. Essa é uma excelente fonte de informação para um estudo exploratório. Pode-se facilmente filtrar empresas para datas, setores, *tickers* ou segmentos de governança corporativa.

Toda empresa no banco de dados é identificada pelo seu número único da CVM. Função `search_company` permite que o usuário procure o identificador de uma empresa através de seu nome. Dado um texto de entrada – o nome da empresa –, a função procurará uma correspondência parcial com os nomes de todas as empresas disponíveis no banco de dados. Em seu uso, caracteres latinos e maiúsculas e minúsculas são ignorados. Vamos encontrar o nome oficial nome da Grendene, uma das maiores empresas do Brasil. Para isso, basta usar o comando `search_company('grendene')`.

```
df_search <- search_company('grendene')

print(df_search)
```

```
R> # A tibble: 1 x 3
R>   CD_CVM DENOM_SOCIAL SIT_REG
R>    <dbl> <chr>        <chr>
R> 1  19615 GRENDENE SA  ATIVO
```

Vemos que existe um registro para a Grendene: “GRENDENE SA”, com código identificador equivalente a 19615.

Com o identificador da empresa disponível, usamos a função principal do pacote, `get_dfp_data`, para baixar os dados. Definimos o nome oficial da empresa como entrada `companies_cvm_codes` e o período de tempo como entradas `first_year` e `last_year`.

```
library(GetDFPData2)
library(dplyr)

# set options
id_companies <- 19615
first_year <- 2017
last_year  <- 2018

# download data
l_dfp <- get_dfp_data(companies_cvm_codes = id_companies,
                      type_docs = '*', # get all docs
                      type_format = 'con', # consolidated
                      first_year = first_year,
                      last_year = last_year)
```

As mensagens de `GetDFPData2::get_dfp_data` relatam os estágios do processo, desde a aquisição de dados da tabela de referência ao download e leitura dos arquivos da B3. Observe que os arquivos de três sistemas são acessados: DFP (*Demostrativos Financeiros Padronizados*), FRE (*Formulário de Referência*) e FCA (*Formulário Cadastral*). Observe também o uso de um sistema de cache, o qual acelera significativamente o uso do software ao salvar localmente as informações importadas.

Explicando as demais entradas da função `GetDFPData2::get_dfp_data`:

companies_cvm_codes Código numérico das empresas (encontrado via `GetDFPData2::search_company('ambev')`)

type_docs Símbolo do tipo de documento financeiro a ser retornado. Definições: '*' = retorna todos documentos, 'BPA' = Ativo, 'BPP' = passivo, 'DRE' = demonstrativo de resultados do exercício, 'DFC_MD' = fluxo de caixa pelo metodo direto, 'DFC_MI' = fluxo de caixa pelo metodo indireto, 'DMPL' = mutacoes do patrimonio liquido, 'DVA' = demonstrativo de valor agregado.

type_format Tipo de formato dos documentos: consolidado ('con') ou individual ('ind'). Como regra, dê preferência ao tipo consolidado, o qual incluirá dados completos de subsidiárias.

first_year Primeiro ano para os dados

last_year Último ano para os dados

O objeto resultante de `get_dfp_data` é uma `lista` com diversas tabelas. Vamos dar uma olhada no conteúdo de `l_dfp` ao buscar os nomes dos itens da lista, limitando o número de caracteres:

```
stringr::str_sub(names(l_dfp), 1, 40)
```

```
R> [1] "DF Consolidado - Balanço Patrimonial Ati"
R> [2] "DF Consolidado - Balanço Patrimonial Pas"
R> [3] "DF Consolidado - Demonstração das Mutaçõ"
R> [4] "DF Consolidado - Demonstração de Valor A"
R> [5] "DF Consolidado - Demonstração do Fluxo d"
R> [6] "DF Consolidado - Demonstração do Resulta"
```

Como podemos ver, os dados retornados são vastos. Cada item da lista em `l_dfp` é um tabela indexada ao tempo. A explicação de cada coluna não cabe aqui mas, para fins de exemplo, vamos dar uma olhada no balanço patrimonial da empresa, disponível em `l_dfp$"DF Consolidado - Balanço Patrimonial Ativo"`:

```
# save assets in df
fr_assets <- l_dfp$`DF Consolidado - Balanço Patrimonial Ativo`

# check it
dplyr::glimpse(fr_assets)
```

```
R> Rows: 122
R> Columns: 16
R> $ CNPJ_CIA     <chr> "89.850.341/0001-60", "89.850.341/000~
R> $ CD_CVM       <dbl> 19615, 19615, 19615, 19615, 19615, 19~
R> $ DT_REFER     <date> 2017-12-31, 2017-12-31, 2017-12-31, ~
R> $ DT_INI_EXERC <date> NA, NA, NA, NA, NA, NA, NA, NA, NA, ~
R> $ DT_FIM_EXERC <date> 2017-12-31, 2017-12-31, 2017-12-31, ~
R> $ DENOM_CIA    <chr> "GRENDENE S.A.", "GRENDENE S.A.", "GR~
R> $ VERSAO       <dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1~
R> $ GRUPO_DFP    <chr> "DF Consolidado - Balanço Patrimonial~
R> $ MOEDA        <chr> "REAL", "REAL", "REAL", "REAL", "REAL~
R> $ ESCALA_MOEDA <chr> "MIL", "MIL", "MIL", "MIL", "MIL", "M~
R> $ ORDEM_EXERC  <chr> "ÚLTIMO", "ÚLTIMO", "ÚLTIMO", "ÚLTIMO~
R> $ CD_CONTA     <chr> "1", "1.01", "1.01.01", "1.01.02", "1~
R> $ DS_CONTA     <chr> "Ativo Total", "Ativo Circulante", "C~
R> $ VL_CONTA     <dbl> 3576008, 2846997, 30119, 1537477, 836~
R> $ COLUNA_DF    <chr> NA, NA, NA, NA, NA, NA, NA, NA, NA, N~
R> $ source_file  <chr> "dfp_cia_aberta_BPA_con_2017.csv", "d~
```

A exportação dos dados para o Excel também é fácil, basta usar função `GetDFPData2::export_xlsx`:

```
temp_xlsx <- tempfile(fileext = '.xlsx')

export_xlsx(l_dfp = l_dfp, f_xlsx = temp_xlsx)
```

```
R> Warning in writexl::write_xlsx(x = l_dfp, path = f_xlsx):
R> Truncating sheet name(s) to 31 characters
```

```
R> Warning in writexl::write_xlsx(x = l_dfp, path = f_xlsx):
R> Deduplicating sheet names
```

O arquivo Excel resultante conterá cada tabela de `l_dpf` em uma aba diferente da planilha, com uma truncagem nos nomes. Podemos checar o resultado com função `readxl::excel_sheets`:

```
readxl::excel_sheets(temp_xlsx)
```

```
R> [1] "DF Consolidado - Balanço Pat"
R> [2] "DF Consolidado - Balanço Pat_1"
R> [3] "DF Consolidado - Demonstraçã"
R> [4] "DF Consolidado - Demonstraçã_1"
R> [5] "DF Consolidado - Demonstraçã_2"
R> [6] "DF Consolidado - Demonstraçã_3"
```

7.5 Pacote {GetFREData} (M. Perlin e Kirch 2022)

O pacote **{GetFREData}** (M. Perlin e Kirch 2022) importa dados do sistema FRE – Formulário de Referência – da bolsa Brasileira, incluindo eventos e informações corporativas tal como composição do conselho e diretoria, remuneração dos conselhos, entre outras.

A estrutura de uso e a saída das funções de **{GetFREData}** (M. Perlin e Kirch 2022) são muito semelhante as do pacote **{GetDFPData2}** (M. Perlin e Kirch 2023). Veja a seguir um exemplo de uso.

```
library(GetFREData)

# set options
id_companies <- 23264
first_year <- 2021
last_year  <- 2022
```

```
7
8  # download data
9  l_fre <- get_fre_data(companies_cvm_codes = id_companies,
10                        first_year = first_year,
11                        last_year = last_year)
```

Note que o tempo de execução de `get_fre_data` é significativo. Isto deve-se ao *download* e leitura dos arquivos do sistema FRE direto da bolsa. Cada tabela do FRE é importada na lista de saída:

```
1  names(l_fre)
```

```
R>  [1] "df_stockholders"
R>  [2] "df_capital"
R>  [3] "df_stock_values"
R>  [4] "df_mkt_value"
R>  [5] "df_increase_capital"
R>  [6] "df_capital_reduction"
R>  [7] "df_compensation"
R>  [8] "df_compensation_summary"
R>  [9] "df_transactions_related"
R> [10] "df_other_events"
R> [11] "df_stock_repurchases"
R> [12] "df_debt_composition"
R> [13] "df_board_composition"
R> [14] "df_committee_composition"
R> [15] "df_family_relations"
R> [16] "df_family_related_companies"
R> [17] "df_auditing"
R> [18] "df_responsible_docs"
R> [19] "df_stocks_details"
R> [20] "df_dividends_details"
R> [21] "df_intangible_details"
```

Por exemplo, vamos verificar conteúdo da tabela `df_board_composition`, a qual contém informações sobre os componentes dos conselhos das empresas:

```
1  glimpse(l_fre$df_board_composition)
```

```
R> Rows: 65
```

```
R> Columns: 22
R> $ CNPJ_CIA                 <chr> "07.526.557/0001-00", "07~
R> $ DENOM_CIA                <chr> "AMBEV S.A.", "AMBEV S.A.~
R> $ DT_REFER                 <date> 2021-01-01, 2021-01-01, ~
R> $ CD_CVM                   <dbl> 23264, 23264, 23264, 2326~
R> $ ID_DOC                   <dbl> 114809, 114809, 114809, 1~
R> $ VERSAO                   <dbl> 12, 12, 12, 12, 12, 12, 1~
R> $ person.name              <chr> "Leticia Rudge Barbosa Ki~
R> $ person.cpf               <dbl> 25572648817, 7234352783, ~
R> $ person.profession        <chr> "Advogada", "Engenheiro C~
R> $ person.cv                <chr> "Ocupa o cargo de Diretor~
R> $ person.dob               <date> 1976-06-12, 1977-01-15, ~
R> $ code.type.board          <chr> "1", "1", "1", "1", "1", ~
R> $ desc.type.board          <chr> "Director", "Director", "~
R> $ desc.type.board2         <chr> "Diretora Vice-Presidente~
R> $ code.type.job            <chr> "19", "19", "19", "19", "~
R> $ desc.job                 <chr> "Não aplicável, uma vez q~
R> $ date.election            <date> 2021-12-22, 2021-12-22, ~
R> $ date.effective           <date> 2022-01-01, 2022-01-01, ~
R> $ mandate.duration         <chr> "31/12/2024", "31/12/2024~
R> $ ellected.by.controller   <lgl> TRUE, TRUE, TRUE, TRUE, F~
R> $ qtd.consecutive.mandates <dbl> 2, 2, 1, 2, 3, 1, 3, 2, 3~
R> $ percentage.participation <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0~
```

Como podemos ver, para um pesquisador de finanças corporativas, o sistema FRE oferece uma série de informações interessantes. Discutir o conteúdo de cada tabela, porém, vai muito além do propósito dessa seção. Aos interessados, mais detalhes sobre as tabelas do FRE estão disponíveis em (**perlin2018accessing?**).

> **!** Importante
>
> Note que a importação dos dados do FRE inclui uma versão dos arquivos. Toda vez que uma empresa modifica as informações oficiais no sistema da B3, uma nova versão do FRE é criada. Devido a isso, é bastante comum que os dados de um ano para uma empresa possua diferentes versões. Para resolver este problema, o código do **{GetFREData}** (M. Perlin e Kirch 2022), por *default*, importa a versão mais antiga para cada ano. Caso o usuário queira mudar, basta utilizar a entrada `fre_to_read`.

7.6 Outros Pacotes

Nas seções anteriores destacamos os principais pacotes gratuitos para aquisição de dados financeiros e econômicos no Brasil. Muitos desses foram escritos pelo próprio autor do livro e representam uma pequena parcela da totalidade. Não seria justo ignorar o trabalho de outros autores. Assim, reporto abaixo uma seleção de pacotes que vale a pena conhecer:

7.6.1 Pacotes de Acesso Gratuito

`BETS` **(R-BETS?)** Pacote construído e mantido pela equipe da FGV. Permite o acesso aos dados do BCB (Banco Central do Brasil) e IBGE (Instituto Brasileiro de Geografia e Estatística). Também inclui ferramentas para a administração, análise e manipulação dos dados em relatórios técnicos.

`simfinapi` **(R-simfinapi?)** Pacote para acesso ao projeto simfin, incluindo dados financeiros de diversas empresas internacionais. O acesso livre é restrito a um número de chamadas diárias.

7.6.2 Pacotes Comerciais

`Rblpapi` **(R-Rblpapi?)** Permite acesso aos dados da Bloomberg, sendo necessário uma conta comercial.

`IBrokers` **(R-IBrokers?)** API para o acesso aos dados da Interactive Brokers. Também é necessário uma conta comercial.

No CRAN você encontrará muitos outros. A interface para fontes de dados comerciais também é possível. Várias empresas fornecem APIs para facilitar o envio de dados aos seus clientes. Se a empresa de fornecimento de dados que você usa no trabalho não for apresentada aqui, a lista de pacotes CRAN pode ajudá-lo a encontrar uma alternativa viável.

7.7 Acessando Dados de Páginas na Internet (*Webscraping*)

Os pacotes destacados anteriormente são muito úteis pois facilitam a importação de dados específicos diretamente da internet. Em muitos casos,

porém, os dados de interesse não estão disponíveis via API formal, mas sim em uma página na internet - geralmente no formato de uma tabela. O processo de extrair informações de páginas da internet chama-se *webscraping* (raspagem de dados). Dependendo da estrutura e da tecnologia da página da *web* acessada, importar essas informações diretamente para o R pode ser um procedimento trivial – mas também pode se tornar um processo extremamente trabalhoso. Como um exemplo, a seguir vamos raspar dados do Wikipedia sobre a composição do índice SP500.

7.7.1 Raspando Dados do Wikipedia

Em seu site, a Wikipedia oferece uma seção[2] com os componentes do Índice SP500. Essas informações são apresentadas em um formato tabular, Figura @ref(fig:SP500-wikipedia).

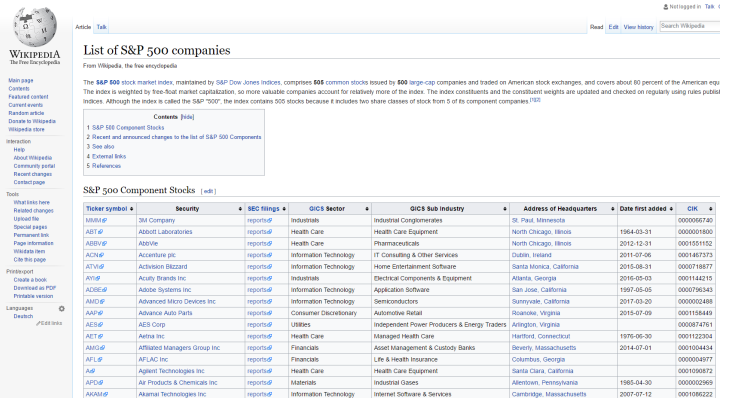


Figura 7.1: Imagem da página do Wikipedia

As informações desta página são constantemente atualizadas, e podemos utilizá-las para importar informações sobre as ações pertencentes ao índice SP500. Antes de nos aprofundarmos no código R, precisamos entender como uma página da web funciona. Resumidamente, uma página da web nada mais é do que uma árvore com nódulos, representada por um código HTML (*Hypertext Markup Language*) extenso interpretado pelo seu navegador. Um valor numérico ou texto apresentado no site geralmente pode ser encontrado dentro do próprio código. Este código tem uma estrutura particular em forma de árvore com ramificações, classes, nomes e identificadores. Além disso, cada elemento de uma página da web possui um endereço, denominado *xpath*. Nos navegadores Chrome e Firefox,

[2] https://en.wikipedia.org/wiki/List_of_S%26P_500_companies

você pode ver o código HTML de uma página da web usando o mouse. Para isto, basta clicar com o botão direito em qualquer parte da página e selecionar *View Page Source* (ou "Ver Código Fonte").

A primeira etapa do processo de raspagem de dados é descobrir a localização das informações de que você precisa. No navegador Chrome, você pode fazer isso clicando com o botão direito no local específico do número/texto no site e selecionando *inspect*. Isso abrirá uma janela extra no navegador a direita. Depois de fazer isso, clique com o botão direito na seleção e escolha *copy* e *copy xpath*. Na Figura @ref(fig:SP500-Wikipedia-webscraping), vemos um espelho do que você deve estar vendo em seu navegador.

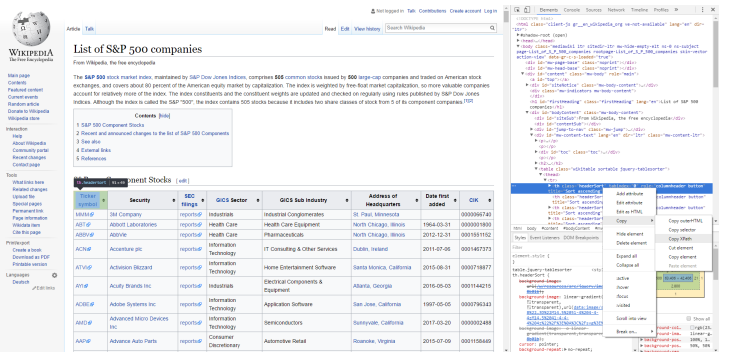


Figura 7.2: Encontrando o xpath da tabela

Aqui, o texto copiado é:

```
'//*[@id="mw-content-text"]/table[1]/thead/tr/th[2]'
```

Este é o endereço do cabeçalho da tabela. Para todo o conteúdo da tabela, incluindo cabeçalho, linhas e colunas, precisamos definir um nível superior da árvore. Isso é equivalente ao endereço `//*[@id ="mw-content-text"]/table[1]`.

Agora que temos a localização do que queremos, vamos carregar o pacote `rvest` (**R-rvest?**) e usar as funções `read_html`, `html_nodes` e`html_table` para importar a tabela desejada para o R:

```
library(rvest)

# set url and xpath
my_url <- 'https://en.wikipedia.org/wiki/List_of_S%26P_500_companies'
my_xpath <- '//*[@id="mw-content-text"]/div/table[1]'
```

```
6
7  # get nodes from html
8  out_nodes <- html_nodes(read_html(my_url),
9                          xpath = my_xpath)
10
11 # get table from nodes (each element in
12 # list is a table)
13 df_SP500_comp <- html_table(out_nodes)
14
15 # isolate it
16 df_SP500_comp <- df_SP500_comp[[1]]
17
18 # change column names (remove space)
19 names(df_SP500_comp) <- make.names(names(df_SP500_comp))
20
21 # print it
22 glimpse(df_SP500_comp)
```

```
R> Rows: 503
R> Columns: 8
R> $ Symbol                <chr> "MMM", "AOS", "ABT", "ABBV",~
R> $ Security              <chr> "3M", "A. O. Smith", "Abbott~
R> $ GICS.Sector           <chr> "Industrials", "Industrials"~
R> $ GICS.Sub.Industry     <chr> "Industrial Conglomerates", ~
R> $ Headquarters.Location <chr> "Saint Paul, Minnesota", "Mi~
R> $ Date.added            <chr> "1957-03-04", "2017-07-26", ~
R> $ CIK                   <int> 66740, 91142, 1800, 1551152,~
R> $ Founded               <chr> "1902", "1916", "1888", "201~
```

O objeto `df_SP500_comp` contém um espelho dos dados do site da Wikipedia. Os nomes das colunas requerem algum trabalho de limpeza, mas o principal está ali. Observe como a saída é semelhante aos dados da função `BatchGetSymbols::GetSP500Stocks`. A razão é simples, ambas buscaram a informação na mesma origem. A diferença é que função `GetSP500Stocks` vai um passo além, limpando os dados importados.

7.8 Exercícios

CAPÍTULO 8

DATAFRAMES E OUTROS OBJETOS

No R, tudo é um objeto e cada classe de objeto tem propriedades diferentes. Por exemplo, um `dataframe` pode ser incrementado com novas colunas ou linhas. Uma coluna numérica de um `dataframe` pode interagir com outros valores numéricos através de operações de multiplicação, divisão e soma. Para colunas com textos, porém, tal propriedade não é válida, uma vez que não faz sentido somar um valor numérico a um texto ou dividir um texto por outro. Entretanto, a classe de texto tem outras propriedades, como a que permite procurar uma determinada sequência textual dentro de um texto maior, a manipulação de partes do texto e a substituição de caracteres específicos, dentre tantas outras possibilidades. **Um dos aspectos mais importantes no trabalho com o R é o aprendizado das classes de objetos e as suas funcionalidades**.

As classes básicas de objetos no R inclui valores numéricos, caracteres (texto), fatores, datas, entre vários outros casos. Na prática, porém, as classes básicas são armazenadas em estruturas de dados mais complexas, tal como `dataframes` e listas. Isso organiza e facilita o trabalho. Imagine realizar um estudo sobre as *63* ações que compõem o índice Ibovespa, onde a base de dados é composta por preços e volumes negociados ao longo de um ano. Caso fôssemos criar um vetor numérico de preços e de volumes para cada ação, teríamos uma quantidade de *126* objetos

para lidar no nosso ambiente do R. Apesar de ser possível trabalhar dessa forma, o código resultante seria desorganizado, difícil de entender e passível de uma série de erros.

Uma maneira mais simples de organizar os nossos dados é criar um objeto com o nome `my_data` e alocar todos os preços e volumes ali. Todas as informações necessárias para executar a pesquisa estariam nesse objeto, facilitando a importação e exportação dos dados. Esses objetos que armazenam outros objetos de classe básica constituem a classe de estrutura de dados. Nessa classificação, estão incluídas tabelas (`dataframes`) e listas (`list`).

8.1 Dataframes

Traduzindo para o português, `dataframe` significa "estrutura ou organização de dados". Grosso modo, um objeto da classe `dataframe` nada mais é do que uma tabela com linhas e colunas. Sem dúvida, **o `dataframe` é o principal objeto utilizado no trabalho com o R e o mais importante de se estudar**. Dados externos são, grande maioria dos casos, importados para o R no formato de tabelas. É na manipulação desses que gastará maior parte do tempo realizando a sua análise. Internamente, um `dataframe` é um tipo especial de lista, onde cada coluna é um vetor atômico com o mesmo número de elementos, porém com sua própria classe. Podemos organizar em um `dataframe` dados de texto juntamente com números, por exemplo.

Note que o formato tabular força a sincronização dos dados no sentido de linhas, isto é, cada caso de cada variável deve ser pareado com casos de outras variáveis. Apesar de simples, esse tipo de estruturação de dados é intuitiva e pode acomodar uma variedade de informações. Cada acréscimo de dados (informações) incrementa as linhas e cada novo tipo de informação incrementa as colunas da tabela.

Um dos pontos positivos na utilização do `dataframe` para a acomodação de dados é que funções de diferentes pacotes irão funcionar a partir dessa classe de objetos. Por exemplo, o pacote de manipulação de dados **{dplyr}** (Wickham, François, et al. 2023), assim como o pacote de criação de figuras `ggplot2`, funcionam a partir de um `dataframe`. Esse objeto, portanto, está no centro de uma série de funcionalidades do R e, sem dúvida, é uma classe de objeto extremamente importante para aprender a utilizar corretamente.

O objeto `dataframe` é uma das classes nativas do R e vem implementado no pacote **{base}** (R Core Team 2023b). Entretanto, o universe **{tidyverse}**

(Wickham 2023c) oferece sua própria versão de um `dataframe`, chamada `tibble`, a qual é utilizada sistematicamente em todos pacotes do **{tidyverse}** (Wickham 2023c). A conversão de um `dataframe` para `tibble` é interna e automática. O `tibble` possui propriedades mais flexíveis que `dataframes` nativos, facilitando de forma significativa o seu uso. Seguindo a nossa preferência para o **{tidyverse}** (Wickham 2023c), a partir de agora iremos utilizar `tibbles` como representantes de `dataframes`.

8.1.1 Criando `dataframes`

A criação de um `dataframe` do tipo `tibble` ocorre a partir da função `tibble`. Note que a criação de um `dataframe` nativo ocorre com a função **data.frame()** , enquanto a criação do `tibble` parte da função `tibble::tibble` ou **dplyr::tibble()** . Para manter o código mais limpo, iremos dar preferência a **dplyr::tibble()** e utilizar o nome *dataframe* para se referir a um `tibble`. Veja o exemplo a seguir, onde criamos uma tabela correspondente a dados financeiros de diferentes ações.

```
# set tickers
ticker <- c(rep('ABEV3',4),
            rep('BBAS3', 4),
            rep('BBDC3', 4))

# set dates
ref_date <- as.Date(rep(c('2010-01-01', '2010-01-04',
                          '2010-01-05', '2010-01-06'),
                        3) )

# set prices
price <- c(736.67, 764.14, 768.63, 776.47,
           59.4  , 59.8  , 59.2  , 59.28,
           29.81 , 30.82 , 30.38 , 30.20)

# create tibble/dataframe
my_df <- tibble::tibble(ticker, ref_date , price)

# print it
print(my_df)
```

```
R> # A tibble: 12 x 3
R>    ticker ref_date   price
```

```
R>    <chr>  <date>     <dbl>
R>  1 ABEV3  2010-01-01 737.
R>  2 ABEV3  2010-01-04 764.
R>  3 ABEV3  2010-01-05 769.
R>  4 ABEV3  2010-01-06 776.
R>  5 BBAS3  2010-01-01  59.4
R>  6 BBAS3  2010-01-04  59.8
R>  7 BBAS3  2010-01-05  59.2
R>  8 BBAS3  2010-01-06  59.3
R>  9 BBDC3  2010-01-01  29.8
R> 10 BBDC3  2010-01-04  30.8
R> 11 BBDC3  2010-01-05  30.4
R> 12 BBDC3  2010-01-06  30.2
```

Observe que utilizamos a função **rep()** para replicar e facilitar a criação dos dados do `dataframe` anterior. Assim, não é necessário repetir os valores múltiplas vezes. Destaca-se que, no uso dos `dataframes`, podemos salvar todos os nossos dados em um único objeto, facilitando o acesso e a organização do código resultante.

> **Dica**
>
> O conteúdo de `dataframes` também pode ser visualizado no próprio RStudio. Para isso, basta clicar no nome do objeto na aba *environment*, canto superior direito da tela. Após isso, um visualizador aparecerá na tela principal do programa. Essa operação é nada mais que uma chamada a função **View()** . Portanto, poderíamos visualizar o `dataframe` anterior executando o comando `View(my_df)`.

8.1.2 Inspecionando um `dataframe`

Após a criação do `dataframe`, o segundo passo é conhecer o seu conteúdo. Particularmente, é importante tomar conhecimento dos seguintes itens em ordem de importância:

Número de linhas e colunas O número de linhas e colunas da tabela resultante indicam se a operação de importação foi executada corretamente. Caso os valores forem diferentes do esperado, deve-se checar o arquivo de importação dos dados e se as opções de importação fazem sentido para o arquivo.

Nomes das colunas É importante que a tabela importada tenha nomes que façam sentido e que sejam fáceis de acessar. Portanto, o segundo passo na inspeção de um `dataframe` é analisar os nomes das colunas e seus respectivos conteúdos. Confirme que cada coluna realmente apresenta um nome intuitivo e relacionado ao problema.

Classes das colunas Cada coluna de um `dataframe` tem sua própria classe. É de suma importância que as classes dos dados estejam corretamente especificadas. Caso contrário, operações futuras podem resultar em um erro. Por exemplo, caso um vetor de valores numéricos seja importado com a classe de texto (`character`), qualquer operação matemática nesse vetor irá resultar em um erro no R.

Existência de dados omissos (`NA`) Devemos também verificar o número de valores `NA` (*not available*) nas diferentes colunas. Sempre que você encontrar uma grande proporção de valores `NA` na tabela importada, você deve descobrir o que está acontecendo e se a informação está sendo importada corretamente. Conforme mencionado no capítulo anterior, os valores `NA` são contagiosos e transformarão qualquer objeto que interagir com um `NA`, também se tornará um `NA`.

Uma das funções mais recomendadas para se familiarizar com um `dataframe` é **dplyr::glimpse()** . Essa mostra na tela o nome e a classe das colunas, além do número de linhas/colunas. Abusamos dessa função nos capítulos anteriores. Veja um exemplo simples a seguir:

```
1  library(dplyr)
2
3  # check content of my_df
4  glimpse(my_df)
```

```
R> Rows: 12
R> Columns: 3
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
```

Em muitas situações, o uso de **dplyr::glimpse()** é suficiente para entender se o processo de importação de dados ocorreu de forma satisfatória. Porém, uma análise mais profunda é entender qual a variação de cada coluna nos dados importados. Aqui entra o papel da função **summary()** :

```
1  # check variation my_df
2  summary(my_df)
```

```
R>     ticker             ref_date              price
R>  Length:12          Min.   :2010-01-01   Min.   : 29.81
R>  Class :character   1st Qu.:2010-01-03   1st Qu.: 30.71
R>  Mode  :character   Median :2010-01-04   Median : 59.34
R>                     Mean   :2010-01-04   Mean   :283.73
R>                     3rd Qu.:2010-01-05   3rd Qu.:743.54
R>                     Max.   :2010-01-06   Max.   :776.47
```

Note que **summary()** interpreta cada coluna de forma diferente. Para o primeiro caso, coluna `ticker`, mostra apenas o tamanho do vetor. No caso de datas e valores numéricos, essa apresenta o máximo, mínimo, mediana e quartis. Por exemplo, uma observação extrema (*outlier*) poderia ser facilmente identificada na análise da saída textual de **summary()** .

Uma alternativa moderna para **summary()** é **skimr**::**skim()** , que fornece mais detalhes sobre os dados:

```
# Check content of my_df
skimr::skim(my_df)
```

Você não só obtém as classes das colunas, mas também mais informações sobre as diferentes classes de dados:

- Para a coluna `tickers`, você obtém o número de casos ausentes, valores únicos e mais;
- Para a coluna `dates`, você obtém o mínimo e o máximo de dados, bem como o número de datas disponíveis;
- Para colunas de valores numéricos, você obtém média, desvio padrão e quantis e mais;

Embora não seja suficiente, uma simples chamada a **skimr**::**skim()** pode fornecer uma grande quantidade de informações sobre os dados sendo importados.

> **Dica**
>
> Toda vez que se deparar com um novo `dataframe` no R, pegue o hábito de verificar o seu conteúdo com funções **dplyr**::**glimpse()** e **skimr**::**skim()** . Assim, poderá perceber problemas de importação e/ou conteúdo dos arquivos lidos. Com experiência irás perceber que muitos erros futuros em código podem ser sanados por uma simples inspeção das tabelas importadas.

8.1.3 Operador de *pipeline*

O operador de *pipeline* (ou sequenciamento, em tradução livre) é uma ferramenta fundamental na análise de dados com R. Resumidamente, ele permite que operações de dados sejam realizadas sequencialmente e de forma modular, aumentando a legibilidade e a facilidade de manutenção do código resultante. O operador é amplamente utilizado no pacote **{tidyverse}** (Wickham 2023c) e foi proposto pela primeira vez no pacote r cite_pkg("magrittr") com o símbolo %>%. Recentemente, na versão 4.1 do R, lançada em 18 de maio de 2021, um novo operador pipe (nativo) foi introduzido (|>), facilitando seu uso por todos.

Para explicar melhor, imagine uma situação onde temos três funções para aplicar nos dados salvos em um `dataframe`. Cada função depende da saída de outra função. Isso requer o encadeamento de suas chamadas. Usando o operador de *pipeline*, podemos escrever o procedimento de manipulação `dataframe` com o seguinte código:

```
my_tab <- my_df |>
  fct1(arg1) |>
  fct2(arg2) |>
  fct3(arg3)
```

Usamos símbolo |> no final de cada linha para vincular as operações. As funções `fct*` são operações realizadas em cada etapa. O resultado de cada linha é passado para a próxima função de forma sequencial. Assim, não há necessidade de criar objetos intermediários. Veja a seguir duas formas alternativas de realizar a mesma operação sem o operador de *pipeline*:

```
# version 1
my_tab <- fct3(fct2(fct1(my_df,
                         arg1),
                    arg2),
               arg1)

# version 2
temp1 <- fct1(my_df, arg1)
temp2 <- fct2(temp1, arg2)
my_tab <- fct3(temp1, arg3)
```

Observe como as alternativas formam um código com estrutura estranha e passível a erros. Provavelmente não deves ter notado, mas ambos os códigos possuem erros de digitação. Para o primeiro, o último `arg1` deveria ser `arg3` e, no segundo, a função `fct3` está usando o `dataframe temp1` e não `temp2`. Este exemplo deixa claro como o uso de *pipelines* torna o código mais elegante e legível. A partir de agora iremos utilizar o operador `|>` de forma extensiva.

8.1.4 Acessando Colunas

Um objeto do tipo `dataframe` utiliza-se de diversos comandos e símbolos que também são usados em matrizes e listas. Para descobrir os nomes das colunas de um `dataframe`, temos duas funções: **names()** ou **colnames()** :

```
# check names of df
names(my_df)
```

```
R> [1] "ticker"   "ref_date" "price"
```

```
colnames(my_df)
```

```
R> [1] "ticker"   "ref_date" "price"
```

Ambas também podem ser usadas para modificar os nomes das colunas:

```
# set temp df
temp_df <- my_df

# check names
names(temp_df)
```

```
R> [1] "ticker"   "ref_date" "price"
```

```
# change names
names(temp_df) <- paste0('Col', 1:ncol(temp_df))

# check names
names(temp_df)
```

```
R> [1] "Col1" "Col2" "Col3"
```

Destaca-se que a forma de usar **names()** é bastante distinta das demais funções do R. Nesse caso, utilizamos a função ao lado esquerdo do símbolo de `assign` (<-). Internamente, o que estamos fazendo é definindo um atributo do objeto `temp_df`, o nome de suas colunas.

Para acessar uma determinada coluna, podemos utilizar o nome da mesma de diversas formas:

```
# isolate columns of df
my_ticker <- my_df$ticker
my_prices <- my_df[['price']]

# print contents
print(my_ticker)
```

```
R>  [1] "ABEV3" "ABEV3" "ABEV3" "ABEV3" "BBAS3" "BBAS3" "BBAS3"
R>  [8] "BBAS3" "BBDC3" "BBDC3" "BBDC3" "BBDC3"
```

```
print(my_prices)
```

```
R>  [1] 736.67 764.14 768.63 776.47  59.40  59.80  59.20  59.28
R>  [9]  29.81  30.82  30.38  30.20
```

> ⚠ Cuidado!
>
> Toda vez que estiver acessando colunas de um `dataframe`, o resultado é um vetor atômico com os dados da coluna. Isso é importante saber pois as propriedades do objeto se modificam.

Note o uso do duplo colchetes (`[[]]`) para selecionar colunas. Vale apontar que, no R, **um objeto da classe `dataframe` é representado internamente como uma lista**, onde cada elemento é uma coluna. Isso é importante saber, pois alguns comandos de listas também funcionam para `dataframes`. Um exemplo é o uso de duplo colchetes (`[[]]`) para selecionar colunas por posição:

```
print(my_df[[2]])
```

```
R>  [1] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
R>  [5] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
R>  [9] "2010-01-01" "2010-01-04" "2010-01-05" "2010-01-06"
```

Para acessar linhas e colunas específicas de um `dataframe`, basta utilizar colchetes simples:

```
print(my_df[1:5,2])
```

```
R> # A tibble: 5 x 1
R>   ref_date
R>   <date>
R> 1 2010-01-01
R> 2 2010-01-04
R> 3 2010-01-05
R> 4 2010-01-06
R> 5 2010-01-01
```

```
print(my_df[1:5,c(1,2)])
```

```
R> # A tibble: 5 x 2
R>   ticker ref_date
R>   <chr>  <date>
R> 1 ABEV3  2010-01-01
R> 2 ABEV3  2010-01-04
R> 3 ABEV3  2010-01-05
R> 4 ABEV3  2010-01-06
R> 5 BBAS3  2010-01-01
```

```
print(my_df[1:5, ])
```

```
R> # A tibble: 5 x 3
R>   ticker ref_date   price
R>   <chr>  <date>     <dbl>
R> 1 ABEV3  2010-01-01 737.
R> 2 ABEV3  2010-01-04 764.
R> 3 ABEV3  2010-01-05 769.
R> 4 ABEV3  2010-01-06 776.
R> 5 BBAS3  2010-01-01  59.4
```

Essa seleção de colunas também pode ser realizada utilizando o nome das mesmas da seguinte forma:

```
print(my_df[1:3, c('ticker','price')])
```

```
R> # A tibble: 3 x 2
R>   ticker price
R>   <chr>  <dbl>
R> 1 ABEV3   737.
R> 2 ABEV3   764.
R> 3 ABEV3   769.
```

ou, pelo operador de *pipeline* e a função **dplyr::select()** :

```
library(dplyr)

my.temp <- my_df |>
  select(ticker, price) |>
  glimpse()
```

```
R> Rows: 12
R> Columns: 2
R> $ ticker <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS3"~
R> $ price  <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59.8~
```

8.1.5 Modificando um `dataframe`

Para criar novas colunas em um `dataframe`, basta utilizar a função **dplyr::mutate()** . Aqui iremos abusar do operador de *pipeline* (|>) para sequenciar as operações:

```
library(dplyr)

# add columns with mutate
my_df <- my_df |>
  mutate(ret = price/lag(price) -1,
         my_seq1 = 1:nrow(my_df),
         my_seq2 =  my_seq1 +9) |>
  glimpse()
```

```
R> Rows: 12
R> Columns: 6
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966~
R> $ my_seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
R> $ my_seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2~
```

Note que precisamos indicar o `dataframe` de origem dos dados, nesse caso o objeto `my_df`, e as colunas são definidas como argumentos em **dplyr**::**mutate()** . Observe também que usamos a coluna `price` na construção de `ret`, o retorno aritmético dos preços. Um caso especial é a construção de `my_seq2` com base em `my_seq1`, isto é, antes mesmo dela ser explicitamente calculada já é possível utilizar a nova coluna para criar outra. Vale salientar que a nova coluna deve ter exatamente o mesmo número de elementos que as demais. Caso contrário, o R retorna uma mensagem de erro.

A maneira mais tradicional, e comumente encontrada em código, para criar novas colunas é utilizar o símbolo `$`:

```
# add new column with base R
my_df$my_seq3 <- 1:nrow(my_df)

# check it
glimpse(my_df)
```

```
R> Rows: 12
R> Columns: 7
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966~
R> $ my_seq1  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
R> $ my_seq2  <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 2~
R> $ my_seq3  <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

Portanto, o operador `$` vale tanto para acessar quanto para criar novas colunas.

Para remover colunas de um `dataframe`, basta usar **dplyr**::**select()** com operador negativo para o nome das colunas indesejadas:

```
# removing columns
my_df_temp <- my_df |>
  select(-my_seq1, -my_seq2, -my_seq3) |>
  glimpse()
```

```
R> Rows: 12
R> Columns: 4
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS~
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47, 59.40, 59~
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966~
```

No uso de funções nativas do R, a maneira tradicional de remover colunas é alocar o valor nulo (`NULL`):

```
# set temp df
temp_df <- my_df

# remove cols
temp_df$price <- NULL
temp_df$ref_date  <- NULL

# check it
glimpse(temp_df)
```

```
R> Rows: 12
R> Columns: 5
R> $ ticker  <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3", "BBAS3~
R> $ ret     <dbl> NA, 0.037289424, 0.005875887, 0.010199966,~
R> $ my_seq1 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
R> $ my_seq2 <dbl> 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20~
R> $ my_seq3 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

8.1.6 Filtrando um `dataframe`

Uma operação bastante comum no R é filtrar linhas de uma tabela de acordo com uma ou mais condições. Por exemplo, caso quiséssemos apenas os dados da ação `ABEV3`, poderíamos utilizar a função **dplyr**::**filter()** para filtrar a tabela:

```
library(dplyr)

# filter df for single stock
my_df_temp <- my_df |>
  filter(ticker == 'ABEV3') |>
  glimpse()
```

```
R> Rows: 4
R> Columns: 7
R> $ ticker   <chr> "ABEV3", "ABEV3", "ABEV3", "ABEV3"
R> $ ref_date <date> 2010-01-01, 2010-01-04, 2010-01-05, 2010-~
R> $ price    <dbl> 736.67, 764.14, 768.63, 776.47
R> $ ret      <dbl> NA, 0.037289424, 0.005875887, 0.010199966
R> $ my_seq1  <int> 1, 2, 3, 4
R> $ my_seq2  <dbl> 10, 11, 12, 13
R> $ my_seq3  <int> 1, 2, 3, 4
```

A função também aceita mais de uma condição. Veja a seguir onde filtramos os dados para `'ABEV3'` em datas após ou igual a `'2010-01-05'`:

```
library(dplyr)
# filter df for single stock and date
my_df_temp <- my_df |>
  filter(ticker == 'ABEV3',
         ref_date >= as.Date('2010-01-05')) |>
  glimpse()
```

```
R> Rows: 2
R> Columns: 7
R> $ ticker   <chr> "ABEV3", "ABEV3"
R> $ ref_date <date> 2010-01-05, 2010-01-06
R> $ price    <dbl> 768.63, 776.47
R> $ ret      <dbl> 0.005875887, 0.010199966
R> $ my_seq1  <int> 3, 4
R> $ my_seq2  <dbl> 12, 13
R> $ my_seq3  <int> 3, 4
```

Aqui utilizamos o símbolo == para testar uma igualdade. Iremos estudar mais profundamente a classe de testes lógicos no capítulo Capítulo 9.

8.1.7 Ordenando um `dataframe`

Após a criação ou importação de um `dataframe`, pode-se ordenar seus componentes de acordo com os valores de alguma coluna. Um caso bastante comum em que é necessário realizar uma ordenação explícita é quando importamos dados financeiros em que as datas não estão em ordem crescente. Na grande maioria das situações, dados temporais devem estar ordenados de acordo com a antiguidade, isto é, dados mais recentes são alocados na última linha da tabela. Essa operação é realizada através do uso da função **order()** ou **dplyr::arrange()** .

Como exemplo, considere a criação de um `dataframe` com os valores a seguir:

```
library(tidyverse)

# set df
my_df <- tibble(col1 = c(4,1,2),
                col2 = c(1,1,3),
                col3 = c('a','b','c'))

# print it
print(my_df)
```

```
R> # A tibble: 3 x 3
R>    col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     4     1 a
R> 2     1     1 b
R> 3     2     3 c
```

A função **order()** retorna os índices relativos à ordenação dos valores dados como entrada. Para o caso da primeira coluna de `my_df`, os índices dos elementos formadores do novo vetor, com seus valores ordenados em forma crescente, são:

```
idx <- order(my_df$col1)
print(idx)
```

```
R> [1] 2 3 1
```

Portanto, ao utilizar a saída da função **order()** como indexador do `dataframe`, acaba-se ordenando o mesmo de acordo com os valores da coluna `col1`. Veja a seguir:

```
my_df_2 <- my_df[order(my_df$col1), ]
print(my_df_2)
```

```
R> # A tibble: 3 x 3
R>    col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     2     3 c
R> 3     4     1 a
```

Essa operação de ordenamento também pode ser realizada levando em conta mais de uma coluna. Veja o exemplo a seguir, onde se ordena o `dataframe` pelas colunas `col2` e `col1`.

```
idx <- order(my_df$col2, my_df$col1)
my_df_3 <- my_df[idx, ]
print(my_df_3)
```

```
R> # A tibble: 3 x 3
R>    col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
R> 2     4     1 a
R> 3     2     3 c
```

No **{tidyverse}** (Wickham 2023c), a forma de ordenar `dataframes` é pelo uso da função **dplyr**::**arrange()** . No caso de ordenamento decrescente, encapsulamos o nome das colunas com `desc`:

```
# sort ascending, by col1 and col2
my_df <- my_df |>
  arrange(col1, col2) |>
  print()
```

```
R> # A tibble: 3 x 3
R>    col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     1     1 b
```

```
R> 2     2     3 c
R> 3     4     1 a
```

```
# sort descending, col1 and col2
my_df <- my_df |>
  arrange(desc(col1), desc(col2)) |>
  print()
```

```
R> # A tibble: 3 x 3
R>    col1  col2 col3
R>   <dbl> <dbl> <chr>
R> 1     4     1 a
R> 2     2     3 c
R> 3     1     1 b
```

O resultado prático no uso de **dplyr::arrange()** é o mesmo de **order()** . Um dos seus benefícios é a possibilidade de encadeamento de operações através do uso do *pipeline*.

8.1.8 Combinando e Agregando `dataframes`

Em muitas situações de análise de dados será necessário juntar `dataframes` distintos em um único objeto. Tabelas diferentes são importadas no R e, antes de analisar os dados, precisamos combinar as informações em um único objeto. Nos casos mais simples, onde as tabelas a serem agregadas possuem o mesmo formato, nós as juntamos de acordo com as linhas, verticalmente, ou colunas, horizontalmente. Para esse fim, temos as funções **dplyr::bind_rows()** e **dplyr::bind_cols()** no tidyverse e **rbind()** e **cbind()** nas funções nativas do R. Observe o exemplo a seguir.

```
library(dplyr)

# set dfs
my_df_1 <- tibble(col1 = 1:5,
                  col2 = rep('a', 5))

my_df_2 <- tibble(col1 = 6:10,
                  col2 = rep('b', 5),
                  col3 = rep('c', 5))

# bind by row
```

```
my_df <- bind_rows(my_df_1, my_df_2) |>
  glimpse()
```

```
R> Rows: 10
R> Columns: 3
R> $ col1 <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
R> $ col2 <chr> "a", "a", "a", "a", "a", "b", "b", "b", "b", ~
R> $ col3 <chr> NA, NA, NA, NA, NA, "c", "c", "c", "c", "c"
```

Note que, no exemplo anterior, os nomes das colunas são os mesmos. De fato, a função **dplyr::bind_rows()** procura os nomes iguais em ambos os objetos para fazer a junção dos `dataframes` corretamente. As colunas que não ocorrem em ambos objetos, tal como `col3` no exemplo, saem como `NA` no objeto final. Já para o caso de **dplyr::bind_cols()** , os nomes das colunas devem ser diferentes, porém o número de linhas deve ser o mesmo.

```
# set dfs
my_df_1 <- tibble(col1 = 1:5, col2 = rep('a', 5))
my_df_2 <- tibble(col3 = 6:10, col4 = rep('b', 5))

# bind by column
my_df <- bind_cols(my_df_1, my_df_2) |>
  glimpse()
```

```
R> Rows: 5
R> Columns: 4
R> $ col1 <int> 1, 2, 3, 4, 5
R> $ col2 <chr> "a", "a", "a", "a", "a"
R> $ col3 <int> 6, 7, 8, 9, 10
R> $ col4 <chr> "b", "b", "b", "b", "b"
```

Para casos mais complexos, onde a junção deve ser realizada de acordo com algum índice tal como uma data, é possível juntar `dataframes` diferentes com o uso das funções da família `dplyr::join*` tal como **dplyr::inner_join()** , **dplyr::left_join()** , **dplyr::right_join()** , entre outras. A descrição de todas elas não cabe aqui. Iremos descrever apenas o caso mais provável, **dplyr::inner_join()** . Essa combina os dados, mantendo apenas os casos onde existe o índice em ambos.

```
# set df
my_df_1 <- tibble(date = as.Date('2016-01-01')+0:10,
```

```
                    x = 1:11)

my_df_2 <- tibble(date = as.Date('2016-01-05')+0:10,
                    y = seq(20,30, length.out = 11))
```

Note que os `dataframes` criados possuem uma coluna em comum, `date`. A partir desta coluna que agregamos as tabelas com **dplyr::inner_join()** :

```
# aggregate tables
my_df <- inner_join(my_df_1, my_df_2)
```

```
R> Joining with `by = join_by(date)`
```

```
glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-01-~
R> $ x    <int> 5, 6, 7, 8, 9, 10, 11
R> $ y    <dbl> 20, 21, 22, 23, 24, 25, 26
```

O R automaticamente verifica a existência de colunas com mesmo nome nos `dataframes` e realiza a junção por essas. Caso quiséssemos juntar `dataframes` onde os nomes das colunas para utilizar o índice não são iguais, temos duas soluções: modificar os nomes das colunas ou então utilizar argumento `by` em **dplyr::inner_join()** . Veja a seguir:

```
# set df
my_df_3 <- tibble(ref_date = as.Date('2016-01-01')+0:10,
                    x = 1:11)

my_df_4 <- tibble(my_date = as.Date('2016-01-05')+0:10,
                    y = seq(20,30, length.out = 11))

# join by my_df_3$ref_date and my_df_4$my_date
my_df <- inner_join(my_df_3, my_df_4,
                      by = c('ref_date' = 'my_date'))

glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ ref_date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016~
R> $ x        <int> 5, 6, 7, 8, 9, 10, 11
R> $ y        <dbl> 20, 21, 22, 23, 24, 25, 26
```

Para o caso de uso da função nativa de agregação de `dataframes`, **merge()** , temos que indicar explicitamente o nome da coluna com argumento `by`:

```
# aggregation with base R
my_df <- merge(my_df_1, my_df_2, by = 'date')

glimpse(my_df)
```

```
R> Rows: 7
R> Columns: 3
R> $ date <date> 2016-01-05, 2016-01-06, 2016-01-07, 2016-01-~
R> $ x    <int> 5, 6, 7, 8, 9, 10, 11
R> $ y    <dbl> 20, 21, 22, 23, 24, 25, 26
```

Note que, nesse caso, o `dataframe` resultante manteve apenas as informações compartilhadas entre ambos os objetos, isto é, aquelas linhas onde as datas em `date` eram iguais. Esse é o mesmo resultado quando no uso do **dplyr**::**inner_join()** .

As demais funções de agregação de tabelas – **dplyr**::**left_join()** , **dplyr**::**right_join()** e **dplyr**::**full_join()** – funcionam de forma muito semelhante a **dplyr**::**inner_join()** , exceto na escolha da saída. Por exemplo, **dplyr**::**full_join()** retorna todos os casos/linhas entre tabela 1 e 2, incluindo aqueles onde não tem o índice compartilhado. Para estes casos, a coluna do índice sairá como `NA`. Veja o exemplo a seguir:

```
# set df
my_df_5 <- tibble(ref_date = as.Date('2016-01-01')+0:10,
                  x = 1:11)

my_df_6 <- tibble(ref_date = as.Date('2016-01-05')+0:10,
                  y = seq(20,30, length.out = 11))

# combine with full_join
my_df <- full_join(my_df_5, my_df_6)
```

```
R> Joining with `by = join_by(ref_date)`
```

```
# print it
print(my_df)
```

```
R> # A tibble: 15 x 3
R>    ref_date       x     y
R>    <date>     <int> <dbl>
R>  1 2016-01-01     1    NA
R>  2 2016-01-02     2    NA
R>  3 2016-01-03     3    NA
R>  4 2016-01-04     4    NA
R>  5 2016-01-05     5    20
R>  6 2016-01-06     6    21
R>  7 2016-01-07     7    22
R>  8 2016-01-08     8    23
R>  9 2016-01-09     9    24
R> 10 2016-01-10    10    25
R> 11 2016-01-11    11    26
R> 12 2016-01-12    NA    27
R> 13 2016-01-13    NA    28
R> 14 2016-01-14    NA    29
R> 15 2016-01-15    NA    30
```

8.1.9 Extensões ao `dataframe`

Como já foi relatado nos capítulos anteriores, um dos grandes benefícios no uso do R é a existência de pacotes para lidar com os problemas específicos dos usuários. Enquanto um objeto tabular do tipo `tibble` é suficiente para a maioria dos casos, existem benefícios no uso de uma classe alternativa. Ao longo do tempo, diversas soluções foram disponibilizadas por desenvolvedores.

Por exemplo, é muito comum trabalharmos com dados exclusivamente numéricos que são indexados ao tempo. Isto é, situações onde cada informação pertence a um índice temporal - um objeto da classe data/tempo. As linhas dessa tabela representam um ponto no tempo, enquanto as colunas indicam variáveis numéricas de interesse. Nesse caso, faria sentido representarmos os nossos dados como objetos do tipo **{xts}** (**R-xts?**). O grande benefício dessa opção é que a agregação e a manipulação de variáveis em função do tempo é muito fácil. Por exemplo, podemos transformar dados de frequência diária para a frequência semanal com apenas uma linha

de comando. Além disso, diversas outras funções reconhecem automaticamente que os dados são indexados ao tempo. Um exemplo é a criação de uma figura com esses dados. Nesse caso, o eixo horizontal da figura é automaticamente organizado com as datas.

Veja um caso a seguir, onde carregamos os dados anteriores como um objeto **{xts}** (**R-xts?**):

```
1 library(xts)
```

```
R> Loading required package: zoo

R>
R> Attaching package: 'zoo'

R> The following objects are masked from 'package:base':
R>
R>     as.Date, as.Date.numeric

R>
R> ########################## Warning from 'xts' package ##################
R> #
R> # The dplyr lag() function breaks how base R's lag() function is supposed
R> # work, which breaks lag(my_xts). Calls to lag(my_xts) that you type or
R> # source() into this session won't work correctly.
R> #
R> # Use stats::lag() to make sure you're not using dplyr::lag(), or you can
R> # conflictRules('dplyr', exclude = 'lag') to your .Rprofile to stop
R> # dplyr from breaking base R's lag() function.
R> #
R> # Code in packages is not affected. It's protected by R's namespace mecha
R> # Set `options(xts.warn_dplyr_breaks_lag = FALSE)` to suppress this warni
R> #
R> ##############################################################################

R>
R> Attaching package: 'xts'

R> The following objects are masked from 'package:dplyr':
R>
R>     first, last
```

```
# set data
ticker <- c('ABEV3', 'BBAS3','BBDC3')

date <- as.Date(c('2010-01-01', '2010-01-04',
                  '2010-01-05', '2010-01-06'))

price_ABEV3 <- c(736.67, 764.14, 768.63, 776.47)
price_BBAS3 <- c(59.4, 59.8, 59.2, 59.28)
price_BBDC3 <- c(29.81, 30.82, 30.38, 30.20)

# build matrix
my_mat <- matrix(c(price_BBDC3, price_BBAS3, price_ABEV3),
                 nrow = length(date) )

# set xts object
my_xts <- xts(my_mat,
              order.by = date)

# set correct colnames
colnames(my_xts) <- ticker

# check it!
print(my_xts)
```

```
R>            ABEV3 BBAS3  BBDC3
R> 2010-01-01 29.81 59.40 736.67
R> 2010-01-04 30.82 59.80 764.14
R> 2010-01-05 30.38 59.20 768.63
R> 2010-01-06 30.20 59.28 776.47
```

O código anterior pode dar a impressão de que o objeto `my_xts` é semelhante a um `dataframe`, porém, não se engane. Por estar indexado a um vetor de tempo, objeto `my_xts` pode ser utilizado para uma série de procedimentos temporais, tal como uma agregação por período temporal. Veja o exemplo a seguir, onde agregamos duas variáveis de tempo através do cálculo de uma média a cada semana.

```
N <- 500

my_mat <- matrix(c(seq(1, N), seq(N, 1)), nrow=N)

```

```
5  my_xts <- xts(my_mat, order.by = as.Date('2016-01-01')+1:N)
6
7  my_xts.weekly.mean <- apply.weekly(my_xts, mean)
8
9  print(head(my_xts.weekly.mean))
```

```
R>              X.1   X.2
R> 2016-01-03  1.5 499.5
R> 2016-01-10  6.0 495.0
R> 2016-01-17 13.0 488.0
R> 2016-01-24 20.0 481.0
R> 2016-01-31 27.0 474.0
R> 2016-02-07 34.0 467.0
```

Em Finanças e Economia, as agregações com objetos **{xts}** (**R-xts?**) são extremamente úteis quando se trabalha com dados em frequências de tempo diferentes. Por exemplo, é muito comum que se agregue dados de transação no mercado financeiro em alta frequência para intervalos maiores. Assim, dados que ocorrem a cada segundo são agregados para serem representados de 15 em 15 minutos. Esse tipo de procedimento é facilmente realizado no R através da correta representação dos dados como objetos **{xts}** (**R-xts?**). Existem diversas outras funcionalidades desse pacote. Encorajo os usuários a ler o manual e aprender o que pode ser feito.

Indo além, existem diversos outros tipos de `dataframes` customizados. Por exemplo, o `dataframe` proposto pelo pacote **{data.table}** (**R-data.table?**) prioriza o tempo de operação nos dados e o uso de uma notação compacta para acesso e processamento. O **{tibbletime}** (**R-tibbletime?**) é uma versão orientada pelo tempo para `tibbles`. Caso o usuário esteja necessitando realizar operações de agregação de tempo, o uso deste pacote é fortemente recomendado.

8.1.10 Outras Funções Úteis

head - Retorna os primeiros `n` elementos de um `dataframe`.

```
1  my_df <- tibble(col1 = 1:5000, col2 = rep('a', 5000))
2  head(my_df, 5)
```

```
R> # A tibble: 5 x 2
R>    col1 col2
R>   <int> <chr>
R> 1     1 a
R> 2     2 a
R> 3     3 a
R> 4     4 a
R> 5     5 a
```

tail - Retorna os últimos `n` elementos de um `dataframe`.

```
tail(my_df, 5)
```

```
R> # A tibble: 5 x 2
R>    col1 col2
R>   <int> <chr>
R> 1  4996 a
R> 2  4997 a
R> 3  4998 a
R> 4  4999 a
R> 5  5000 a
```

complete.cases - Retorna um vetor lógico que testa se as linhas contêm apenas valores existentes e nenhum `NA`.

```
my_x <- c(1:5, NA, 10)
my_y <- c(5:10, NA)
my_df <- tibble(my_x, my_y)

print(my_df)
```

```
R> # A tibble: 7 x 2
R>    my_x  my_y
R>   <dbl> <int>
R> 1     1     5
R> 2     2     6
R> 3     3     7
R> 4     4     8
R> 5     5     9
R> 6    NA    10
R> 7    10    NA
```

```
print(complete.cases(my_df))
```

```
R> [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
```

```
print(which(!complete.cases(my_df)))
```

```
R> [1] 6 7
```

na.omit - Retorna um `dataframe` sem as linhas onde valores `NA` são encontrados.

```
print(na.omit(my_df))
```

```
R> # A tibble: 5 x 2
R>    my_x  my_y
R>   <dbl> <int>
R> 1     1     5
R> 2     2     6
R> 3     3     7
R> 4     4     8
R> 5     5     9
```

unique - Retorna um `dataframe` onde todas as linhas duplicadas são eliminadas e somente os casos únicos são mantidos.

```
my_df <- tibble(col1 = c(1,1,2,3,3,4,5),
                col2 = c('A','A','A','C','C','B','D'))

print(my_df)
```

```
R> # A tibble: 7 x 2
R>    col1 col2
R>   <dbl> <chr>
R> 1     1 A
R> 2     1 A
R> 3     2 A
R> 4     3 C
R> 5     3 C
R> 6     4 B
R> 7     5 D
```

```
print(unique(my_df))
```

```
R> # A tibble: 5 x 2
R>    col1 col2
R>   <dbl> <chr>
R> 1     1 A
R> 2     2 A
R> 3     3 C
R> 4     4 B
R> 5     5 D
```

8.2 Listas

Uma lista (`list`) é uma classe de objeto extremamente flexível e já tivemos contato com ela nos capítulos anteriores. Ao contrário de vetores atômicos, a lista não apresenta restrição alguma em relação aos tipos de elementos nela contidos. Podemos agrupar valores numéricos com caracteres, fatores com datas e até mesmo listas dentro de listas. Quando agrupamos vetores, também não é necessário que os mesmos tenham um número igual de elementos. Além disso, podemos dar um nome a cada elemento. Essas propriedades fazem da lista o objeto mais flexível para o armazenamento e estruturação de dados no R. Não é acidental o fato de que listas são muito utilizadas como retorno de funções.

8.2.1 Criando Listas

Uma lista pode ser criada através do comando **list()** , seguido por seus elementos separados por vírgula:

```
library(dplyr)

# create list
my_l <- list(c(1, 2, 3),
             c('a', 'b'),
             factor('A', 'B', 'C'),
             tibble(col1 = 1:5))

# use base::print
```

```
10 print(my_l)
```

```
R> [[1]]
R> [1] 1 2 3
R>
R> [[2]]
R> [1] "a" "b"
R>
R> [[3]]
R> [1] <NA>
R> Levels: C
R>
R> [[4]]
R> # A tibble: 5 x 1
R>    col1
R>   <int>
R> 1     1
R> 2     2
R> 3     3
R> 4     4
R> 5     5
```

```
1 # use dplyr::glimpse
2 glimpse(my_l)
```

```
R> List of 4
R>  $ : num [1:3] 1 2 3
R>  $ : chr [1:2] "a" "b"
R>  $ : Factor w/ 1 level "C": NA
R>  $ : tibble [5 x 1] (S3: tbl_df/tbl/data.frame)
R>   ..$ col1: int [1:5] 1 2 3 4 5
```

Note que juntamos no mesmo objeto um vetor atômico numérico, outro de texto, um fator e um `tibble`. A apresentação de listas com o comando **print()** é diferente dos casos anteriores. Os elementos são separados verticalmente e os seus índices aparecem com duplo colchete (`[[ ]]`). Conforme será explicado logo a seguir, é dessa forma que os elementos de uma lista são armazenados e acessados.

Assim como para os demais tipos de objeto, os elementos de uma lista também podem ter nomes, o que facilita o entendimento e a interpretação das informações do problema em análise. Por exemplo, considere o

caso de uma base de dados com informações sobre determinada ação negociada na bolsa. Nesse caso, podemos definir uma lista como:

```
# set named list
my_named_l <- list(ticker = 'TICK4',
                   market = 'Bovespa',
                   df_prices = tibble(P = c(1,1.5,2,2.3),
                                      ref_date = Sys.Date()+0:3))

# check content
glimpse(my_named_l)
```

```
R> List of 3
R>  $ ticker   : chr "TICK4"
R>  $ market   : chr "Bovespa"
R>  $ df_prices: tibble [4 x 2] (S3: tbl_df/tbl/data.frame)
R>   ..$ P       : num [1:4] 1 1.5 2 2.3
R>   ..$ ref_date: Date[1:4], format: "2024-01-10" ...
```

> **Dica**
>
> Toda vez que for trabalhar com listas, facilite a sua vida ao nomear todos os elementos de forma intuitiva. Isso facilita o acesso aos elementos e evita possíveis erros no código.

8.2.2 Acessando os Elementos de uma Lista

Os elementos de uma lista podem ser acessados através do uso de duplo colchete (`[[ ]]`), tal como em:

```
# accessing elements from list
print(my_named_l[[2]])
```

```
R> [1] "Bovespa"
```

```
print(my_named_l[[3]])
```

```
R> # A tibble: 4 x 2
R>       P ref_date
R>   <dbl> <date>
R> 1   1   2024-01-10
R> 2   1.5 2024-01-11
R> 3   2   2024-01-12
R> 4   2.3 2024-01-13
```

Também é possível acessar os elementos com um colchete simples (`[ ]`), porém, tome cuidado com essa operação, pois o resultado não vai ser o objeto em si, mas uma outra lista. Esse é um equívoco muito fácil de passar despercebido, resultando em erros no código. Veja a seguir:

```
# set list
my_l <- list('a',
             c(1,2,3),
             factor('a','b'))

# check classes
class(my_l[[2]])
```

```
R> [1] "numeric"
```

```
class(my_l[2])
```

```
R> [1] "list"
```

Caso tentarmos somar um elemento a `my_l[2]`, teremos uma mensagem de erro:

```
my_l[2] + 1
```

```
R> Error in my_l[2] + 1: non-numeric argument to binary operator
```

Esse erro ocorre devido ao fato de que uma lista não tem operador de soma. Para corrigir, basta utilizar o duplo colchete, tal como em `my_l[[2]]+1`. O acesso a elementos de uma lista com colchete simples somente é útil quando estamos procurando uma sublista dentro de uma lista maior. No exemplo anterior, caso quiséssemos obter o primeiro e o segundo elemento da lista `my_l`, usaríamos:

```
# set new list
my_new_l <- my_l[c(1,2)]

# check contents
print(my_new_l)
```

```
R> [[1]]
R> [1] "a"
R>
R> [[2]]
R> [1] 1 2 3
```

```
class(my_new_l)
```

```
R> [1] "list"
```

No caso de listas com elementos nomeados, os mesmos podem ser acessados por seu nome através do uso do símbolo `$` tal como em `my_named_l$df_prices` ou `[['nome']]`, tal como em `my_named_l[['df_prices']]`. Em geral, essa é uma forma mais eficiente e recomendada de interagir com os elementos de uma lista. Como regra geral no uso do R, sempre dê preferência ao acesso de elementos através de seus nomes, seja em listas, vetores ou `dataframes`. Isso evita erros, pois, ao modificar os dados e adicionar algum outro objeto na lista, é possível que o ordenamento interno mude e, portanto, a posição de determinado objeto pode acabar sendo modificada.

> **Dica**
>
> Saiba que a ferramenta de *autocomplete* do RStudio também funciona para listas. Para usar, digite o nome da lista seguido de `$` e aperte *tab*. Uma caixa de diálogo com todos os elementos disponíveis na lista irá aparecer. A partir disso, basta selecionar apertando *enter*.

Veja os exemplos a seguir, onde são apresentadas as diferentes formas de se acessar uma lista.

```
# different ways to access a list
my_named_l$ticker
my_named_l$price
my_named_l[['ticker']]
```

```
my_named_l[['price']]
```

Vale salientar que também é possível acessar diretamente os elementos de um vetor que esteja dentro de uma lista através de colchetes encadeados. Veja a seguir:

```
# accessing elements of a vector in a list
my_l <- list(c(1,2,3),
             c('a', 'b'))

print(my_l[[1]][2])
```

```
R> [1] 2
```

```
print(my_l[[2]][1])
```

```
R> [1] "a"
```

Tal operação é bastante útil quando interessa apenas um elemento dentro de um objeto maior criado por alguma função.

8.2.3 Adicionando e Removendo Elementos de uma Lista

A remoção, adição e substituição de elementos de uma lista também são procedimentos fáceis. Para adicionar ou substituir, basta definir um novo objeto na posição desejada da lista:

```
# set list
my_l <- list('a', 1, 3)
dplyr::glimpse(my_l)
```

```
R> List of 3
R>  $ : chr "a"
R>  $ : num 1
R>  $ : num 3
```

```
# add new elements to list
my_l[[4]] <- c(1:5)
my_l[[2]] <- c('b')
```

```
4
5  # print result
6  dplyr::glimpse(my_l)
```

```
R> List of 4
R>  $ : chr "a"
R>  $ : chr "b"
R>  $ : num 3
R>  $ : int [1:5] 1 2 3 4 5
```

A operação também é possível com o uso de nomes e operador `$`:

```
1  # set list
2  my_l <- list(elem1 = 'a', name1=5)
3
4  # set new element
5  my_l$name2 <- 10
6  dplyr::glimpse(my_l)
```

```
R> List of 3
R>  $ elem1: chr "a"
R>  $ name1: num 5
R>  $ name2: num 10
```

Para remover elementos de uma lista, basta definir o elemento para o símbolo reservado `NULL` (nulo):

```
1  # set list
2  my_l <- list(text = 'b', num1 = 2, num2 = 4)
3  dplyr::glimpse(my_l)
```

```
R> List of 3
R>  $ text: chr "b"
R>  $ num1: num 2
R>  $ num2: num 4
```

```
1  # remove elements
2  my_l[[3]] <- NULL
3  dplyr::glimpse(my_l)
```

```
R> List of 2
R>  $ text: chr "b"
R>  $ num1: num 2
```

```
my_l$num1 <- NULL
dplyr::glimpse(my_l)
```

```
R> List of 1
R>  $ text: chr "b"
```

Outra maneira de retirar elementos de uma lista é utilizando um índice negativo para os elementos indesejados. Observe a seguir, onde eliminamos o segundo elemento de uma lista:

```
# set list
my_l <- list(a = 1, b = 'texto')

# remove second element
dplyr::glimpse(my_l[[-2]])
```

```
R>  num 1
```

Assim como no caso de vetores atômicos, essa remoção também pode ser realizada por condições lógicas. Veja a seguir:

```
# set list
my_l <- list(1, 2, 3, 4)

# remove elements by condition
my_l[my_l > 2] <- NULL
dplyr::glimpse(my_l)
```

```
R> List of 2
R>  $ : num 1
R>  $ : num 2
```

Porém, note que esse atalho só funciona porque todos os elementos de `my_l` são numéricos.

8.2.4 Processando os Elementos de uma Lista

Um ponto importante a ser destacado a respeito de listas é que os seus elementos podem ser processados e manipulados individualmente através de funções específicas. Este é um tópico particular de programação com o R, mas que vale a apresentação aqui.

Por exemplo, imagine uma lista com vetores numéricos de diferentes tamanhos, tal como a seguir:

```
# set list
my_l_num <- list(c(1, 2, 3),
                 seq(1:50),
                 seq(-5, 5, by = 0.5))
```

Caso quiséssemos calcular a média de cada elemento de `my_l_num` e apresentar o resultado na tela como um vetor, poderíamos fazer isso através de um procedimento simples, processando cada elemento individualmente:

```
# calculate mean of vectors
mean_1 <- mean(my_l_num[[1]])
mean_2 <- mean(my_l_num[[2]])
mean_3 <- mean(my_l_num[[3]])

# print it
print(c(mean_1, mean_2, mean_3))
```

```
R> [1]  2.0 25.5  0.0
```

O código anterior funciona, porém não é recomendado devido sua falta de escabilidade. Isto é, caso aumentássemos o volume de dados ou objetos, o código não funcionaria corretamente. Se, por exemplo, tivéssemos um quarto elemento em `my_l_num` e quiséssemos manter essa estrutura do código, teríamos que adicionar uma nova linha `mean_4 <- mean(my_l_num[[4]])` e modificar o comando de saída na tela para `print(c(mean_1, mean_2, mean_3, mean_4))`.

Uma maneira mais fácil, elegante e inteligente seria utilizar a função **sapply()** . Nela, basta indicar o nome do objeto de tipo lista e a função que queremos utilizar para processar cada elemento. Internamente, os cálculos são realizados automaticamente. Veja a seguir:

```
# using sapply
my_mean <- sapply(my_l_num, mean)

# print result
print(my_mean)
```

```
R> [1]  2.0 25.5  0.0
```

O uso da função **sapply()** é preferível por ser mais compacto e eficiente do que a alternativa – a criação de `mean_1`, `mean_2` e `mean_3`. Note que o primeiro código, com médias individuais, só funciona para uma lista com três elementos. A função **sapply()** , ao contrário, funcionaria da mesma forma em listas de qualquer tamanho. Caso tivéssemos mais elementos, nenhuma modificação seria necessária no código anterior, o que o torna extensível a chegada de novos dados.

Essa visão e implementação de código voltado a procedimentos genéricos é um dos lemas para tornar o uso do R mais eficiente. **A regra é simples: sempre escreva códigos que sejam adaptáveis a chegada de novos dados**. Em inglês, isso é chamado de regra *DRY* (*don't repeat yourself*). Caso você esteja repetindo códigos e abusando do *control + c/control + v*, como no exemplo anterior, certamente existe uma solução mais elegante e flexível que poderia ser utilizada. No R, existem diversas outras funções da família `apply` para esse objetivo. Essas funções serão explicadas com maiores detalhes no capítulo Capítulo 10.

8.2.5 Outras Funções Úteis

unlist - Retorna os elementos de uma lista em um único vetor atômico.

```
my_named_l <- list(ticker = 'XXXX4',
                   price = c(1,1.5,2,3),
                   market = 'Bovespa')
my_unlisted <- unlist(my_named_l)
print(my_unlisted)
```

```
R>    ticker    price1    price2    price3    price4    market
R>   "XXXX4"       "1"     "1.5"       "2"       "3" "Bovespa"
```

```
class(my_unlisted)
```

```
R> [1] "character"
```

as.list - Converte um objeto para uma lista, tornando cada elemento um elemento da lista.

```
my_x <- 10:13
my_x_as_list <- as.list(my_x)
print(my_x_as_list)
```

```
R> [[1]]
R> [1] 10
R>
R> [[2]]
R> [1] 11
R>
R> [[3]]
R> [1] 12
R>
R> [[4]]
R> [1] 13
```

names - Retorna ou define os nomes dos elementos de uma lista. Assim como para o caso de nomear elementos de um vetor atômico, usa-se a função **names()** alocada ao lado esquerdo do símbolo <-.

```
my_l <- list(value1 = 1, value2 = 2, value3 = 3)
print(names(my_l))
```

```
R> [1] "value1" "value2" "value3"
```

```
my_l <- list(1,2,3)
names(my_l) <- c('num1', 'num2', 'num3')
print(my_l)
```

```
R> $num1
R> [1] 1
R>
R> $num2
R> [1] 2
R>
R> $num3
R> [1] 3
```

8.3 Exercícios

CAPÍTULO 9

AS CLASSES BÁSICAS DE OBJETOS

As classes básicas são os elementos mais primários na representação de dados no R. Nos capítulos anteriores utilizamos as classes básicas como colunas em uma tabela. Dados numéricos viraram uma coluna do tipo **numeric()** , enquanto dados de texto viraram um objeto do tipo `character`.

Neste capítulo iremos estudar mais a fundo as classes básicas de objetos do R, incluindo a sua criação até a manipulação do seu conteúdo. Este capítulo é de suma importância pois mostrará quais operações são possíveis com cada classe de objeto e como podes manipular as informações de forma eficiente. Os tipos de objetos tratados aqui serão:

- Numéricos (**numeric()**)
- Texto (`character`)
- Fatores (**factor()**)
- Valores lógicos (`logical`)
- Datas e tempo (`Date` e **{timeDate}** (**R-timeDate?**))
- Dados Omissos (`NA`)

9.1 Objetos Numéricos

Uma das classes mais utilizadas em pesquisa empírica. Os valores numéricos são representações de uma quantidade. Por exemplo: o preço de uma ação em determinada data, o volume negociado de um contrato financeiro em determinado dia, a inflação anual de um país, entre várias outras possibilidades.

9.1.1 Criando e Manipulando Vetores Numéricos

A criação e manipulação de valores numéricos é fácil e direta. Os símbolos de operações matemáticas seguem o esperado, tal como soma (+), diminuição (−), divisão (/) e multiplicação (∗). Todas as operações matemáticas são efetuadas com a orientação de elemento para elemento e possuem notação vetorial. Isso significa, por exemplo, que podemos manipular vetores inteiros em uma única linha de comando. Veja a seguir, onde se cria dois vetores e realiza-se diversas operações entre eles.

```
# create numeric vectors
x <- 1:5
y <- 2:6

# print sum
print(x+y)
```

```
R> [1]  3  5  7  9 11
```

```
# print multiplication
print(x*y)
```

```
R> [1]  2  6 12 20 30
```

```
# print division
print(x/y)
```

```
R> [1] 0.5000000 0.6666667 0.7500000 0.8000000 0.8333333
```

```
# print exponentiation
print(x^y)
```

```
R> [1]     1     8    81  1024 15625
```

Um diferencial do R em relação a outras linguagens é que, nele, são aceitas operações entre vetores diferentes. Por exemplo, podemos somar um vetor numérico de quatro elementos com outro de apenas dois. Nesse caso, aplica-se a chamada **regra de reciclagem** (*recycling rule*). Ela define que, se dois vetores de tamanho diferente estão interagindo, o vetor menor é repetido tantas vezes quantas forem necessárias para obter-se o mesmo número de elementos do vetor maior. Veja o exemplo a seguir:

```
# set x with 4 elements and y with 2
x <- 1:4
y <- 2:1

# print sum
print(x + y)
```

```
R> [1] 3 3 5 5
```

O resultado de `x + y` é equivalente a `1:4 + c(2, 1, 2, 1)`. Caso interagirmos vetores em que o tamanho do maior não é múltiplo do menor, o R realiza o mesmo procedimento de reciclagem, porém emite uma mensagem de `warning`:

```
# set x = 4 elements and y with 3
x <- c(1, 2, 3, 4)
y <- c(1, 2, 3)

# print sum (recycling rule)
print(x +y)
```

```
R> Warning in x + y: longer object length is not a multiple of
R> shorter object length

R> [1] 2 4 6 5
```

Os três primeiros elementos de `x` foram somados aos três primeiros elementos de `y`. O quarto elemento de `x` foi somado ao primeiro elemento de `y`. Uma vez que não havia um quarto elemento em `y`, o ciclo reinicia, resgatando o primeiro elemento de `y` e resultando em uma soma igual a 5.

Os elementos de um vetor numérico também podem ser nomeados quando na criação do vetor:

```
# create named vector
x <- c(item1 = 10,
       item2 = 14,
       item3 = 9,
       item4 = 2)

# print it
print(x)
```

```
R> item1 item2 item3 item4
R>    10    14     9     2
```

Para nomear os elementos após a criação, podemos utilizar a função **names()** . Veja a seguir:

```
# create unnamed vector
x <- c(10, 14, 9, 2)

# set names of elements
names(x) <- c('item1', 'item2', 'item3', 'item4')

# print it
print(x)
```

```
R> item1 item2 item3 item4
R>    10    14     9     2
```

Vetores numéricos vazios também podem ser criados. Em algumas situações de desenvolvimento de código faz sentido pré-alocar o vetor antes de preenchê-lo com valores. Nesse caso, utilize a função **numeric()** :

```
# create empty numeric vector of length 10
my_x <- numeric(length = 10)

# print it
print(my_x)
```

```
R>  [1] 0 0 0 0 0 0 0 0 0 0
```

Observe que, nesse caso, os valores de `my_x` são definidos como zero.

9.1.1.1 Criando Sequências de Valores

Existem duas maneiras de criar uma sequência de valores no R. A primeira, que já foi utilizada nos exemplos anteriores, é o uso do operador `:`. Por exemplo, `my_seq <- 1:10` ou `my_seq <- -5:5`. Esse método é bastante prático, pois a notação é clara e direta.

Porém, o uso do operador `:` limita as possibilidades. A diferença entre os valores adjacentes é sempre *1* para sequências ascendentes e *-1* para sequências descendentes. Uma versão mais poderosa para a criação de sequências é o uso da função **seq()** . Com ela, é possível definir os intervalos entre cada valor com o argumento `by`. Veja a seguir:

```
# set sequence
my_seq <- seq(from = -10, to = 10, by = 2)

# print it
print(my_seq)
```

```
R>  [1] -10  -8  -6  -4  -2   0   2   4   6   8  10
```

Outro atributo interessante da função **seq()** é a possibilidade de criar vetores com um valor inicial, um valor final e o número de elementos desejado. Isso é realizado com o uso da opção `length.out`. Observe o código a seguir, onde cria-se um vetor de *0* até *10* com *20* elementos:

```
# set sequence with fixed size
my_seq <- seq(from = 0, to = 10, length.out = 20)

# print it
print(my_seq)
```

```
R>  [1]  0.0000000  0.5263158  1.0526316  1.5789474  2.1052632
R>  [6]  2.6315789  3.1578947  3.6842105  4.2105263  4.7368421
R> [11]  5.2631579  5.7894737  6.3157895  6.8421053  7.3684211
R> [16]  7.8947368  8.4210526  8.9473684  9.4736842 10.0000000
```

No caso anterior, o tamanho final do vetor foi definido e a própria função se encarregou de descobrir qual a variação necessária entre cada valor de `my_seq`.

9.1.1.2 Criando Vetores com Elementos Repetidos

Outra função interessante é a que cria vetores com o uso de repetição. Por exemplo: imagine que estamos interessado em um vetor preenchido com o valor 1 dez vezes. Para isso, basta utilizar a função **rep()** :

```
# repeat vector three times
my_x <- rep(x = 1, times = 10)

# print it
print(my_x)
```

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

A função também funciona com vetores. Considere uma situação onde temos um vetor com os valores `c(1,2)` e gostaríamos de criar um vetor maior com os elementos `c(1, 2, 1, 2, 1, 2)` - isto é, repetindo o vetor menor três vezes. Veja o resultado a seguir:

```
# repeat vector three times
my_x <- rep(x = c(1, 2), times = 3)

# print it
print(my_x)
```

```
R> [1] 1 2 1 2 1 2
```

9.1.1.3 Criando Vetores com Números Aleatórios

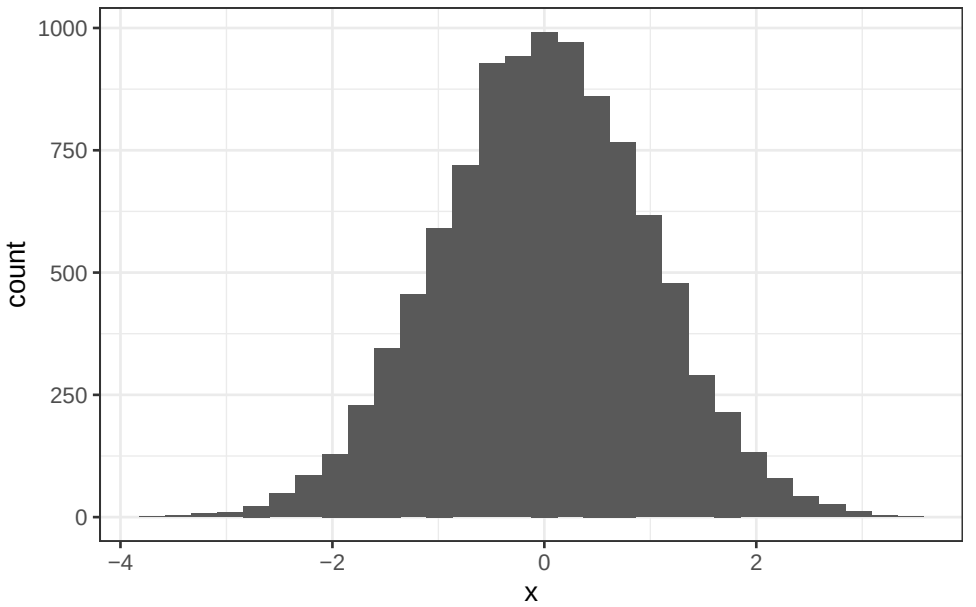
Em muitas situações será necessário a criação de números aleatórios. Esse procedimento numérico é bastante utilizado para simular modelos matemáticos em Finanças. Por exemplo, o método de simulação de preços de ativos de Monte Carlo parte da simulação de números aleatórios (**mcleish2011monte?**). No R, existem diversas funções que criam números aleatórios para diferentes distribuições estatísticas. As mais utilizadas, porém, são as funções **rnorm()** e **runif()** .

A função **rnorm()** gera números aleatórios da distribuição Normal, com opções para a média (tendência) e o desvio padrão (variabilidade). Veja o seu uso a seguir:

```
# generate 10000 random numbers from a Normal distribution
my_rnd_vec <- rnorm(n = 10000,
                    mean = 0,
                    sd = 1)

# print first 20 elements
print(my_rnd_vec[1:20])
```

```
R>  [1]  1.9458156 -1.8150075  1.6399213 -0.6760233 -1.3520480
R>  [6]  0.2390089 -0.5602420 -2.5951910  0.4588666  0.9098799
R> [11]  0.9036514 -0.1834885 -0.3067361  1.2085282 -2.2443810
R> [16]  1.0787225  0.2373673  1.9225758  0.5382652  0.7071240
```

O código anterior gera uma grande quantidade de números aleatórios de uma distribuição Normal com média zero e desvio padrão igual a um. Podemos verificar a distribuição dos números gerados com um histograma:



Como esperado, temos o formato de sino que caracteriza uma distribuição do tipo Normal. Como exercício, podes trocar as entradas **mean()** e **sd()** e confirmar como isso afeta a figura gerada.

Já a função **runif()** gera valores aleatórios da distribuição uniforme entre um valor máximo e um valor mínimo. Ela é geralmente utilizada para simular probabilidades. A função **runif()** tem três parâmetros de entrada: o

número de valores aleatórios desejado, o valor mínimo e o valor máximo. Veja exemplo a seguir:

```
# create a random vector with minimum and maximum
my_rnd_vec <- runif(n = 10,
                    min = -5,
                    max = 5)

# print it
print(my_rnd_vec)
```

```
R>  [1]  1.894668 -4.650248  4.524448  3.187659 -2.274724
R>  [6] -3.030924 -1.481241 -1.132249  3.764846 -2.859107
```

Observe que ambas as funções anteriores são limitadas à sua respectiva distribuição. Uma maneira alternativa e flexível de gerar valores aleatórios é utilizar a função **sample()** . Essa tem como entrada um vetor qualquer e retorna uma versão embaralhada de seus elementos. A sua flexibilidade reside no fato de que o vetor de entrada pode ser qualquer coisa. Por exemplo, caso quiséssemos criar um vetor aleatório com os números `c(0, 5, 15, 20, 25)` apenas, poderíamos fazê-lo da seguinte forma:

```
# create sequence
my_vec <- seq(from = 0, to = 25, by=5)

# sample sequence
my_rnd_vec <- sample(my_vec)

# print it
print(my_rnd_vec)
```

```
R> [1] 25  5 15 20  0 10
```

A função **sample()** também permite a seleção aleatória de um certo número de termos. Por exemplo, caso quiséssemos selecionar aleatoriamente apenas um elemento de `my_vec`, escreveríamos o código da seguinte maneira:

```
# sample one element of my_vec
my_rnd_vec <- sample(my_vec, size = 1)

# print it
```

```
print(my_rnd_vec)
```

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

Caso quiséssemos dois elementos, escreveríamos:

```
# sample two elements of my_vec
my_rnd_vec <- sample(my_vec, size = 2)

# print it
print(my_rnd_vec)
```

```
R> [1]  5 20
```

Também é possível selecionar valores de uma amostra menor para a criação de um vetor maior. Por exemplo, considere o caso em que se tem um vetor com os números `c(10, 15, 20)` e deseja-se criar um vetor aleatório com dez elementos retirados desse vetor menor, com repetição. Para isso, podemos utilizar a opção `replace = TRUE`.

```
# create vector
my_vec <- c(5, 10, 15)

# sample
my_rnd_vec <- sample(x = my_vec, size = 10, replace = TRUE)
print(my_rnd_vec)
```

```
R>  [1] 10  5 10 15 15  5 15 10  5 15
```

Vale destacar que a função **sample()** funciona para qualquer tipo ou objeto, não sendo, portanto, exclusiva para vetores numéricos. Poderíamos, também, escolher elementos aleatórios de um vetor de texto ou então uma lista:

```
# example of sample with characters
print(sample(c('elem 1', 'elem 2', 'elem 3'),
             size = 1))
```

```
R> [1] "elem 2"
```

```
# example of sample with list
print(sample(list(x = c(1,1,1),
                  y = c('a', 'b')),
             size = 1))
```

```
R> $x
R> [1] 1 1 1
```

É importante ressaltar que a geração de valores aleatórios no R (ou qualquer outro programa) **não é totalmente aleatória!** De fato, o próprio computador escolhe os valores dentre uma fila de valores possíveis. Cada vez que funções tal como **rnorm()** , **runif()** e **sample()** são utilizadas, o computador escolhe um lugar diferente dessa fila de acordo com vários parâmetros, incluindo a data e horário atual. Portanto, do ponto de vista do usuário, os valores são gerados de forma imprevisível. Para o computador, porém, essa seleção é completamente determinística e guiada por regras claras.

Uma propriedade interessante no R é a possibilidade de selecionar uma posição específica na fila de valores aleatórios utilizando função **set.seed()** . É ela que **fixa a semente** para gerar os valores. Na prática, o resultado é que todos os números e seleções aleatórias realizadas pelo código serão iguais em cada execução. O uso de **set.seed()** é bastante recomendado para manter a reprodutibilidade dos códigos envolvendo aleatoriedade. Veja o exemplo a seguir, onde utiliza-se essa função.

```
# fix seed
set.seed(seed = 10)

# set vec and print
my_rnd_vec_1 <- runif(5)
print(my_rnd_vec_1)
```

```
R> [1] 0.50747820 0.30676851 0.42690767 0.69310208 0.08513597
```

```
# set vec and print
my_rnd_vec_2 <- runif(5)
print(my_rnd_vec_2)
```

```
R> [1] 0.2254366 0.2745305 0.2723051 0.6158293 0.4296715
```

No código anterior, o valor de **set.seed()** é um inteiro escolhido pelo usuário. Após a chamada de **set.seed()** , todas as seleções e números aleatórios irão iniciar do mesmo ponto e, portanto, serão iguais. Motivo o leitor a executar o código anterior no R. Verás que os valores de `my_rnd_vec_1` e `my_rnd_vec_2` serão exatamente iguais aos valores colocados aqui.

O uso de **set.seed()** também funciona para o caso de **sample()** . Veja a seguir:

```
# fix seed
set.seed(seed = 15)

# print vectors
print(sample(1:10))
```

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

```
print(sample(10:20))
```

```
R>  [1] 13 15 10 17 20 14 19 12 11 18 16
```

Novamente, execute os comandos anteriores no R e verás que o resultado na tela bate com o apresentado aqui.

9.1.2 Acessando Elementos de um Vetor Numérico

Todos os elementos de um vetor numérico podem ser acessados através do uso de colchetes (`[ ]`). Por exemplo, caso quiséssemos apenas o primeiro elemento de `x`, teríamos:

```
# set vector
x <- c(-1, 4, -9, 2)

# get first element
first_elem_x <- x[1]

# print it
print(first_elem_x)
```

```
R> [1] -1
```

A mesma notação é válida para extrair porções de um vetor. Caso quisés-semos um subvetor de `x` com o primeiro e o segundo elemento, faríamos essa operação da seguinte forma:

```
# sub-vector of x
sub_x <- x[1:2]

# print it
print(sub_x)
```

```
R> [1] -1  4
```

Para acessar elementos nomeados de um vetor numérico, basta utilizar seu nome junto aos colchetes.

```
# set named vector
x <- c(item1 = 10, item2 = 14, item3 = -9, item4 = -2)

# access elements by name
print(x['item2'])
```

```
R> item2
R>    14
```

```
print(x[c('item2','item4')])
```

```
R> item2 item4
R>    14    -2
```

O acesso aos elementos de um vetor numérico também é possível através de testes lógicos. Por exemplo, caso tivéssemos interesse em saber quais os valores de `x` que são maiores do que *0*, o código resultante seria da seguinte forma:

```
# find all values of x higher than zero
print(x[x > 0])
```

```
R> item1 item2
R>    10    14
```

Os usos de regras de segmentação dos dados de acordo com algum critério é chamado de indexação lógica. Os objetos do tipo `logical` serão tratados mais profundamente em seção futura deste capítulo.

9.1.3 Modificando e Removendo Elementos de um Vetor Numérico

A modificação de um vetor numérico é muito simples. Basta indicar a posição dos elementos e os novos valores com o símbolo de *assign* (<-):

```
# set vector
my_x <- 1:4

# modify first element to 5
my_x[1] <- 5

# print result
print(my_x)
```

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

Essa modificação também pode ser realizada em bloco:

```
# set vector
my_x <- 0:5

# set the first three elements to 5
my_x[1:3] <- 5

# print result
print(my_x)
```

```
R> [1] 5 5 5 3 4 5
```

O uso de condições para definir elementos é realizada pela indexação:

```
# set vector
my_x <- -5:5

# set any value lower than 2 to 0
my_x[my_x<2] <- 0
```

```
6
7  # print result
8  print(my_x)
```

```
R>  [1] 0 0 0 0 0 0 0 2 3 4 5
```

A remoção de elementos é realizada com o uso de índices negativos:

```
1  # create vector
2  my_x <- -5:5
3
4  # remove first and second element of my_x
5  my_x <- my_x[-(1:2)]
6
7  # show result
8  print(my_x)
```

```
R> [1] -3 -2 -1  0  1  2  3  4  5
```

Note como o uso do índice negativo em `my_x[-(1:2)]` retorna o vetor original sem o primeiro e segundo elemento.

9.1.4 Criando Grupos

Em algumas situações será necessário entender quantos casos da amostra estão localizados entre um determinado intervalo. Por exemplo, imagine o vetor dos retornos diários de uma ação, isto é, a variação percentual dos preços de fechamento entre um dia e outro. Uma possível análise de risco que pode ser realizada é dividir o intervalo de retornos em cinco partes e verificar o percentual de ocorrência dos valores em cada um dos intervalos. Essa análise numérica é bastante semelhante à construção e visualização de histogramas.

A função **cut()** serve para criar grupos de intervalos a partir de um vetor numérico. Veja o exemplo a seguir, onde cria-se um vetor aleatório oriundo da distribuição Normal e cinco grupos a partir de intervalos definidos pelos dados.

```
1  # set rnd vec
2  my_x <- rnorm(10)
3
```

```
# "cut" it into 5 pieces
my_cut <- cut(x = my_x, breaks = 5)
print(my_cut)
```

```
R>  [1] (-1.57,-1.12]  (0.252,0.71]   (-1.12,-0.66]
R>  [4] (-0.204,0.252] (-0.66,-0.204] (-1.57,-1.12]
R>  [7] (0.252,0.71]   (-0.204,0.252] (0.252,0.71]
R> [10] (-0.204,0.252]
R> 5 Levels: (-1.57,-1.12] (-1.12,-0.66] ... (0.252,0.71]
```

Observe que os nomes dos elementos da variável `my_cut` são definidos pelos intervalos e o resultado é um objeto do tipo fator. Em seções futuras, iremos explicar melhor esse tipo de objeto e as suas propriedades.

No exemplo anterior, os intervalos para cada grupo foram definidos automaticamente. No uso da função **cut()** , também é possível definir quebras customizadas nos dados e nos nomes dos grupos. Veja a seguir:

```
# set random vector
my_x <- rnorm(10)

# create groups with 5 breaks
my_cut <- cut(x = my_x, breaks = 5)

# print it!
print(my_cut)
```

```
R>  [1] (-1.3,-0.3]  (-0.3,0.697] (-0.3,0.697] (-2.3,-1.3]
R>  [5] (-0.3,0.697] (0.697,1.69] (0.697,1.69] (0.697,1.69]
R>  [9] (-1.3,-0.3]  (1.69,2.7]
R> 5 Levels: (-2.3,-1.3] (-1.3,-0.3] ... (1.69,2.7]
```

Note que os nomes dos elementos em `my_cut` foram definidos como intervalos e o resultado é um objeto do tipo fator. É possível também definir intervalos e nomes customizados para cada grupo com o uso dos argumentos `labels` e `breaks`:

```
# create random vector
my_x <- rnorm(10)

# define breaks manually
my_breaks <- c(min(my_x)-1, -1, 1, max(my_x)+1)
```

```
6
7  # define labels manually
8  my_labels <- c('Low','Normal', 'High')
9
10 # create group from numerical vector
11 my_cut <- cut(x = my_x, breaks = my_breaks, labels = my_labels)
12
13 # print both!
14 print(my_x)
```

```
R>  [1]  0.5981759  1.6113647 -0.4373813  1.3526206  0.4705685
R>  [6]  0.4702481  0.3963088 -0.7304926  0.6531176  1.2279598
```

```
1  print(my_cut)
```

```
R>  [1] Normal High   Normal High   Normal Normal Normal Normal
R>  [9] Normal High
R> Levels: Low Normal High
```

Como podemos ver, os nomes dos grupos estão mais amigáveis para uma futura análise.

9.1.5 Outras Funções Úteis

as.numeric - Converte determinado objeto para numérico.

```
1  my_text <- c('1', '2', '3')
2  class(my_text)
```

```
R> [1] "character"
```

```
1  my_x <- as.numeric(my_text)
2  print(my_x)
```

```
R> [1] 1 2 3
```

```
1  class(my_x)
```

```
R> [1] "numeric"
```

sum - Soma os elementos de um vetor.

```
my_x <- 1:50
my_sum <- sum(my_x)
print(my_sum)
```

```
R> [1] 1275
```

max - Retorna o máximo valor numérico do vetor.

```
x <- c(10, 14, 9, 2)
max_x <- max(x)
print(max_x)
```

```
R> [1] 14
```

min - Retorna o mínimo valor numérico do vetor.

```
x <- c(12, 15, 9, 2)
min_x <- min(x)
print(min_x)
```

```
R> [1] 2
```

which.max - Retorna a posição do máximo valor numérico do vetor.

```
x <- c(100, 141, 9, 2)
which.max_x <- which.max(x)
cat(paste('The position of the maximum value of x is ', which.max_x))
```

```
R> The position of the maximum value of x is  2
```

```
cat(' and its value is ', x[which.max_x])
```

```
R>  and its value is  141
```

which.min - Retorna a posição do mínimo valor numérico do vetor.

```
x <- c(10, 14, 9, 2)
which.min_x <- which.min(x)
cat(paste('The position of the minimum value of x is ',
```

```
4             which.min_x, ' and its value is ', x[which.min_x]))
```

```
R> The position of the minimum value of x is  4  and its value is  2
```

sort - Retorna uma versão ordenada de um vetor.

```
1  x <- runif(5)
2  print(sort(x, decreasing = FALSE))
```

```
R> [1] 0.1623069 0.8347800 0.8553657 0.9099027 0.9935257
```

```
1  print(sort(x, decreasing = TRUE))
```

```
R> [1] 0.9935257 0.9099027 0.8553657 0.8347800 0.1623069
```

cumsum - Soma os elementos de um vetor de forma cumulativa.

```
1  my_x <- 1:25
2  my_cumsum <- cumsum(my_x)
3  print(my_cumsum)
```

```
R>  [1]   1   3   6  10  15  21  28  36  45  55  66  78  91 105
R> [15] 120 136 153 171 190 210 231 253 276 300 325
```

prod - Realiza o produto de todos os elementos de um vetor.

```
1  my_x <- 1:10
2  my_prod <- prod(my_x)
3  print(my_prod)
```

```
R> [1] 3628800
```

cumprod - Calcula o produto cumulativo de todos os elementos de um vetor.

```
1  my_x <- 1:10
2  my_prod <- cumprod(my_x)
3  print(my_prod)
```

```
R>  [1]       1       2       6      24     120     720    5040
R>  [8]   40320  362880 3628800
```

9.2 Classe de Caracteres (texto)

A classe de caracteres, ou texto, serve para armazenar informações textuais. Um exemplo prático em Finanças seria o reconhecimento de uma ação através dos seus símbolos de identificação (*tickers*) ou então por sua classe de ação: ordinária ou preferencial. Este tipo de dado tem sido utilizado cada vez mais em pesquisa empírica (**gentzkow2017text?**), resultando em uma diversidade de pacotes.

O R possui vários recursos que facilitam a criação e manipulação de objetos de tipo texto. As funções básicas fornecidas com a instalação de R são abrangentes e adequadas para a maioria dos casos. No entanto, pacote **{stringr}** (Wickham 2023b) do **{tidyverse}** (Wickham 2023c) fornece muitas funções que expandem a funcionalidade básica do R na manipulação de texto.

Um aspecto positivo de **{stringr}** (Wickham 2023b) é que as funções começam com o nome `str_` e possuem nomes informativos. Combinando isso com o recurso de preenchimento automático (*autocomplete*) pela tecla *tab*, fica fácil de localizar os nomes das funções do pacote. Seguindo a prioridade ao universo do **{tidyverse}** (Wickham 2023c), esta seção irá dar preferência ao uso das funções do pacote **{stringr}** (Wickham 2023b). As rotinas nativas de manipulação de texto serão apresentadas, porém de forma limitada.

9.2.1 Criando um Objeto Simples de Caracteres

Todo objeto de caracteres é criado através da encapsulação de um texto por aspas duplas (`" "`) ou simples (`' '`). Para criar um vetor de caracteres com *tickers* de ações, podemos fazê-lo com o seguinte código:

```
1 my_assets <- c('PETR3', 'VALE4', 'GGBR4')
2 print(my_assets)
```

```
R> [1] "PETR3" "VALE4" "GGBR4"
```

Confirma-se a classe do objeto com a função **class()** :

```
1 class(my_assets)
```

```
R> [1] "character"
```

9.2.2 Criando Objetos Estruturados de Texto

Em muitos casos no uso do R, estaremos interessados em criar vetores de texto com algum tipo de estrutura própria. Por exemplo, o vetor `c("text 1", "text 2", ..., "text 20")` possui um lógica de criação clara. Computacionalmente, podemos definir a sua estrutura como sendo a junção do texto `text` e um vetor de sequência, de 1 até 20.

Para criar um vetor textual capaz de unir texto com número, utilizamos a função **stringr::str_c()** ou **paste()** . Veja o exemplo a seguir, onde replica-se o caso anterior com e sem espaço entre número e texto:

```
library(stringr)

# create sequence
my_seq <- 1:20

# create character
my_text <- 'text'

# paste objects together (without space)
my_char <- str_c(my_text, my_seq)
print(my_char)
```

```
R>  [1] "text1"  "text2"  "text3"  "text4"  "text5"  "text6"
R>  [7] "text7"  "text8"  "text9"  "text10" "text11" "text12"
R> [13] "text13" "text14" "text15" "text16" "text17" "text18"
R> [19] "text19" "text20"
```

```
# paste objects together (with space)
my_char <- str_c(my_text, my_seq, sep = ' ')
print(my_char)
```

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

```
# paste objects together (with space)
my_char <- paste(my_text, my_seq)
print(my_char)
```

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

O mesmo procedimento também pode ser realizado com vetores de texto. Veja a seguir:

```
# set character value
my_x <- 'My name is'

# set character vector
my_names <- c('Marcelo', 'Ricardo', 'Tarcizio')

# paste and print
print(str_c(my_x, my_names, sep = ' '))
```

```
R> [1] "My name is Marcelo"  "My name is Ricardo"
R> [3] "My name is Tarcizio"
```

Outra possibilidade de construção de textos estruturados é a repetição do conteúdo de um objeto do tipo caractere. No caso de texto, utiliza-se a função **stringr**::**str_dup() /strrep()** para esse fim. Observe o exemplo a seguir:

```
my_char <- str_dup(string = 'abc', times = 5)
print(my_char)
```

```
R> [1] "abcabcabcabcabc"
```

9.2.3 Objetos Constantes de Texto

O R também possibilita o acesso direto a todas as letras do alfabeto. Esses estão guardadas nos objetos reservados chamados `letters` e `LETTERS`:

```
# print all letters in alphabet (no cap)
print(letters)
```

```
R>  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n"
R> [15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
```

```
# print all letters in alphabet (WITH CAP)
print(LETTERS)
```

```
R>  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
R> [15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Observe que em ambos os casos não é necessário criar os objetos. Por serem constantes embutidas automaticamente na área de trabalho do R, elas já estão disponíveis para uso. Podemos sobrescrever o nome do objeto com outro conteúdo, porém isso não é aconselhável. Nunca se sabe onde esse objeto constante está sendo usado. Outros objetos de texto constantes no R incluem `month.abb` e `month.name`. Veja a seguir o seu conteúdo:

```
# print abreviation and full names of months
print(month.abb)
```

```
R>  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep"
R> [10] "Oct" "Nov" "Dec"
```

```
print(month.name)
```

```
R>  [1] "January"   "February"  "March"     "April"
R>  [5] "May"       "June"      "July"      "August"
R>  [9] "September" "October"   "November"  "December"
```

9.2.4 Selecionando Pedaços de um Texto

Um erro comum praticado por iniciantes é tentar selecionar pedaços de um texto através do uso de colchetes. Observe o código abaixo:

```
# set char object
my_char <- 'ABCDE'

# print its second character: 'B' (WRONG - RESULT is NA)
print(my_char[2])
```

```
R> [1] NA
```

O resultado `NA` indica que o segundo elemento de `my_char` não existe. Isso acontece porque o uso de colchetes refere-se ao acesso de **elementos** de um vetor atômico, e não de caracteres dentro de um texto maior. Observe o que acontece quando utilizamos `my_char[1]`:

```
print(my_char[1])
```

```
R> [1] "ABCDE"
```

O resultado é simplesmente o texto *ABCDE*, que está localizado no primeiro item de `my_char`. Para selecionar pedaços de um texto, devemos utilizar a função específica **stringr**::**str_sub()** /**substr()** :

```
# print third and fourth characters
my_substr <- str_sub(string = my_char,
                     start = 4,
                     end = 4)
print(my_substr)
```

```
R> [1] "D"
```

Essa função também funciona para vetores atômicos. Vamos assumir que você importou dados de texto e o conjunto de dados bruto contém um identificador de 3 dígitos de uma empresa, sempre na mesma posição do texto. Vamos simular a situação no R:

```
# build char vec
my_char_vec <- paste0(c('123','231','321'),
                      ' - other ignorable text')
print(my_char_vec)
```

```
R> [1] "123 - other ignorable text"
R> [2] "231 - other ignorable text"
R> [3] "321 - other ignorable text"
```

Só estamos interessados na informação das três primeiras letras de cada elemento em `my_char_vec`. Para selecioná-los, podemos usar as mesmas funções que antes.

```
# get ids with stringr::str_sub
ids.vec <- str_sub(my_char_vec, 1, 3)
print(ids.vec)
```

```
R> [1] "123" "231" "321"
```

> **!** Importante
>
> **Operações vetorizadas são comuns e esperadas no R**. Quase tudo o que você pode fazer para um único elemento pode ser expandido para vetores. Isso facilita o desenvolvimento de rotinas pois pode-se facilmente realizar tarefas complicadas em uma série de elementos, em uma única linha de código.

9.2.5 Localizando e Substituindo Pedaços de um Texto

Uma operação útil na manipulação de textos é a localização de letras e padrões específicos com funções **stringr::str_locate() /regexpr()** e **stringr::str_locate_all() /gregexpr()** . É importante destacar que, por *default*, essas funções utilizam de expressões do tipo *regex* - expressões regulares (**thompson1968programming?**). Essa é uma linguagem específica para processar textos. Diversos símbolos são utilizados para estruturar, procurar e isolar padrões textuais. Quando utilizada corretamente, o *regex* é bastante útil e de extrema valia.

Usualmente, o caso mais comum em pesquisa é verificar a posição ou a existência de um texto menor dentro de um texto maior. Isto é, um padrão explícito e fácil de entender. Por isso, a localização e substituição de caracteres no próximo exemplo será do tipo fixo, sem o uso de *regex*. Tal informação pode ser passada às funções do pacote **{stringr}** (Wickham 2023b) através de outra função chamada **stringr::fixed()** .

O exemplo a seguir mostra como encontrar o caractere *D* dentre uma série de caracteres.

```
1  library(stringr)
2
3  my_char <- 'ABCDEF-ABCDEF-ABC'
4  pos = str_locate(string = my_char, pattern = fixed('D') )
5  print(pos)
```

```
R>      start end
R> [1,]     4   4
```

Observe que a função **stringr::str_locate()** retorna apenas a primeira ocorrência de *D*. Para resgatar todas as ocorrências, devemos utilizar a função **stringr::str_locate_all()** :

```
# set object
my_char <- 'ABCDEF-ABCDEF-ABC'

# find position of ALL 'D' using str_locate_all
pos = str_locate_all(string = my_char, pattern = fixed('D'))
print(pos)
```

```
R> [[1]]
R>      start end
R> [1,]     4   4
R> [2,]    11  11
```

Para substituir caracteres em um texto, basta utilizar a função **stringr::str_replace()** ou **sub()** e **stringr::str_locate_all()** ou **gsub()** . Vale salientar que `str_replace` substitui a primeira ocorrência do caractere, enquanto **stringr::str_locate_all()** executa uma substituição global - isto é, aplica-se a todas as ocorrências. Veja a diferença a seguir:

```
# set char object
my_char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with sub
my_char <- sub(x = my_char,
               pattern = 'ABC',
               replacement = 'XXX')
print(my_char)
```

```
R> [1] "XXXDEF-ABCDEF-ABC"
```

```
# substitute the FIRST 'ABC' for 'XXX' with str_replace
my_char <- 'ABCDEF-ABCDEF-ABC'
my_char <- str_replace(string = my_char,
                       pattern = fixed('ABC'),
                       replacement = 'XXX')
print(my_char)
```

```
R> [1] "XXXDEF-ABCDEF-ABC"
```

E agora fazemos uma substituição global dos caracteres.

```
# set char object
my_char <- 'ABCDEF-ABCDEF-ABC'

# substitute the FIRST 'ABC' for 'XXX' with str_replace
my_char <- str_replace_all(string = my_char,
                           pattern = 'ABC',
                           replacement = 'XXX')
print(my_char)
```

```
R> [1] "XXXDEF-XXXDEF-XXX"
```

Mais uma vez, vale ressaltar que as operações de substituição também funcionam em vetores. Dê uma olhada no próximo exemplo.

```
# set char object
my_char <- c('ABCDEF','DBCFE','ABC')

# create an example of vector
my_char_vec <- str_c(sample(my_char, 5, replace = T),
                     sample(my_char, 5, replace = T),
                     sep = ' - ')

# show it
print(my_char_vec)
```

```
R> [1] "ABCDEF - ABC"    "ABCDEF - ABCDEF" "ABCDEF - ABC"
R> [4] "ABCDEF - DBCFE"  "DBCFE - ABCDEF"
```

```
# substitute all occurrences of 'ABC'
my_char_vec <- str_replace_all(string = my_char_vec,
                               pattern = 'ABC',
                               replacement = 'XXX')

# print result
print(my_char_vec)
```

```
R> [1] "XXXDEF - XXX"    "XXXDEF - XXXDEF" "XXXDEF - XXX"
R> [4] "XXXDEF - DBCFE"  "DBCFE - XXXDEF"
```

9.2.6 Separando Textos

Em algumas situações, principalmente no processamento de textos, é possível que se esteja interessado em quebrar um texto de acordo com algum separador. Por exemplo, o texto `abc ; bcd ; adf` apresenta informações demarcadas pelo símbolo `;`. Para separar um texto em várias partes, utilizamos a função **stringr::str_split() /strsplit()** . Essas quebram o texto em diversas partes de acordo com algum caractere escolhido. Observe os exemplos a seguir:

```
# set char
my_char <- 'ABCXABCXBCD'

# split it based on 'X' and using stringr::str_split
split_char <- str_split(my_char, 'X')

# print result
print(split_char)
```

```
R> [[1]]
R> [1] "ABC" "ABC" "BCD"
```

A saída dessa função é um objeto do tipo lista. Para acessar os elementos de uma lista, deve-se utilizar o operador `[[ ]]`. Por exemplo, para acessar o texto `bcd` da lista `split_char`, executa-se o seguinte código:

```
print(split_char[[1]][2])
```

```
R> [1] "ABC"
```

Para visualizar um exemplo de dividir textos em vetores, veja o próximo código.

```
# set char
my_char_vec <- c('ABCDEF','DBCFE','ABFC','ACD')

# split it based on 'B' and using stringr::strsplit
split_char <- str_split(my_char_vec, 'B')

# print result
print(split_char)
```

```
R> [[1]]
R> [1] "A"    "CDEF"
R>
R> [[2]]
R> [1] "D"   "CFE"
R>
R> [[3]]
R> [1] "A"  "FC"
R>
R> [[4]]
R> [1] "ACD"
```

Observe como, novamente, um objeto do tipo `list` é retornado. Cada elemento é correspondente ao processo de quebra de texto em `my_char`.

9.2.7 Descobrindo o Número de Caracteres de um Texto

Para descobrir o número de caracteres de um texto, utilizamos a função **stringr::str_length() /nchar()** . Ela também funciona para vetores atômicos de texto. Veja os exemplos mostrados a seguir:

```
# set char
my_char <- 'abcdef'

# print number of characters using stringr::str_length
print(str_length(my_char))
```

```
R> [1] 6
```

E agora um exemplo com vetores.

```
#set char
my_char <- c('a', 'ab', 'abc')

# print number of characters using stringr::str_length
print(str_length(my_char))
```

```
R> [1] 1 2 3
```

9.2.8 Gerando Combinações de Texto

Um truque útil no R é usar as funções **expand.grid()** e **tidyr::expand_grid()** para criar todas as combinações possíveis de elementos em diferentes objetos. Isso é útil quando você quer criar um vetor de texto combinando todos os elementos possíveis de diferentes vetores. Por exemplo, se quisermos criar um vetor com todas as combinações entre dois vetores de texto, podemos escrever:

```
library(tidyverse)

# set vectors
my_vec_1 <- c('John ', 'Claire ', 'Adam ')
my_vec_2 <- c('is fishing.', 'is working.')

# create df with all combinations
my_df <- tidyr::expand_grid(name = my_vec_1,
                            verb = my_vec_2)

# print df
print(my_df)
```

```
R> # A tibble: 6 x 2
R>   name      verb
R>   <chr>     <chr>
R> 1 "John "   is fishing.
R> 2 "John "   is working.
R> 3 "Claire " is fishing.
R> 4 "Claire " is working.
R> 5 "Adam "   is fishing.
R> 6 "Adam "   is working.
```

```
# paste columns together in tibble
my_df <- my_df |>
  mutate(phrase = paste0(name, verb) )

# print result
print(my_df)
```

```
R> # A tibble: 6 x 3
R>   name      verb        phrase
```

```
R>   <chr>     <chr>       <chr>
R> 1 "John "   is fishing. John is fishing.
R> 2 "John "   is working. John is working.
R> 3 "Claire " is fishing. Claire is fishing.
R> 4 "Claire " is working. Claire is working.
R> 5 "Adam "   is fishing. Adam is fishing.
R> 6 "Adam "   is working. Adam is working.
```

Aqui, usamos a função **tidyr**::**expand_grid()** para criar um `dataframe` contendo todas as combinações possíveis de `my_vec_1` e `my_vec_2`. Posteriormente, colamos o conteúdo das colunas do `dataframe` usando **stringr**::**str_c()** .

9.2.9 Codificação de Objetos `character`

Para o R, um *string* de texto é apenas uma sequência de *bytes*. A tradução de *bytes* para caracteres é realizada de acordo com uma estrutura de codificação. Para a maioria dos casos de uso do R, especialmente em países de língua inglesa, a codificação de caracteres não é um problema pois os textos importados no R já possuem a codificação correta. Ao lidar com dados de texto em diferentes idiomas, tal como Português do Brasil, a codificação de caracteres é algo que você deve entender pois eventualmente precisará lidar com isso.

Vamos explorar um exemplo. Aqui, vamos importar dados de um arquivo de texto com a codificação `'ISO-8859-9'` e verificar o resultado.

```
1 # read text file
2 my_f <- introR::data_path('CH07_FileWithLatinChar_Latin1.txt')
3
4 my_char <- readr::read_lines(my_f)
5
6 # print it
7 print(my_char)
```

```
R> [1] "A casa \xe9 bonita e tem muito espa\xe7o"
```

O conteúdo original do arquivo é um texto em português. Como você pode ver, a saída de **readr**::**read_lines()** mostra todos os caracteres latinos com símbolos estranhos. Isso ocorre pois a codificação foi manualmente trocada no arquivo para `'ISO-8859-9'`, enquanto a função **readr**::**read_lines()** utiliza `'UTF-8'` como *default*. A solução mais fácil e di-

reta é modificar a codificação esperada do arquivo nas entradas de **readr**::**read_lines()** . Veja a seguir, onde importamos um arquivo com a codificação correta (`'Latin1'`):

```
my_char <- readr::read_lines(my_f,
                             locale = readr::locale(encoding='Latin1'))

# print it
print(my_char)
```

```
R> [1] "A casa é bonita e tem muito espaço"
```

Os caracteres latinos agora estão corretos pois a codificação em **readr**::**read_lines()** é a mesma do arquivo, `'Latin1'`. Uma boa política neste tópico é sempre verificar a codificação de arquivos de texto importados e combiná-lo em R. A maioria das funções de importação tem uma opção para fazê-lo. Quando possível, sempre dê preferência para `'UTF-8'`. Caso necessário, programas de edição de texto, tal como o notepad++, possuem ferramentas para verificar e trocar a codificação de um arquivo.

9.2.10 Outras Funções Úteis

`stringr::str_to_lower`/`base::tolower` - Converte um objeto de texto para letras minúsculas.

```
print(stringr::str_to_lower('ABC'))
```

```
R> [1] "abc"
```

`stringr::str_to_upper`/`base::toupper` - Convertem um texto em letras maiúsculas.

```
print(toupper('abc'))
```

```
R> [1] "ABC"
```

```
print(stringr::str_to_upper('abc'))
```

```
R> [1] "ABC"
```

9.3 Fatores

A classe de fatores (**factor()**) é utilizada para representar grupos ou categorias dentro de uma base de dados no formato tabular. Por exemplo, imagine um banco de informações com os gastos de diferentes pessoas ao longo de um ano. Nessa base de dados existe um item que define o gênero do indivíduo: masculino ou feminino (M ou F). Essa respectiva coluna pode ser importada e representada como texto, porém, no R, a melhor maneira de representá-la é através do objeto fator, uma vez que a mesma representa uma categoria.

A classe de fatores oferece um significado especial para denotar grupos dentro dos dados. Essa organização é integrada aos pacotes e facilita muito a vida do usuário. Por exemplo, caso quiséssemos criar um gráfico para cada grupo dentro da nossa base de dados, poderíamos fazer o mesmo simplesmente indicando a existência de uma variável de fator para a função de criação da figura. Outra possibilidade é determinar se as diferentes médias de uma variável numérica são estatisticamente diferentes para os grupos dos nossos dados. Podemos também estimar um determinado modelo estatístico para cada grupo. Quando os dados de categorias são representados apropriadamente, o uso das funções do R torna-se mais fácil e eficiente.

9.3.1 Criando Fatores

A criação de fatores dá-se através da função **factor()** :

```
1 my_factor <- factor(c('M', 'F', 'M', 'M', 'F'))
2 print(my_factor)
```

```
R> [1] M F M M F
R> Levels: F M
```

Observe, no exemplo anterior, que a apresentação de fatores com a função **print()** mostra os seus elementos e também o item chamado `Levels`. Esse último identifica os possíveis grupos que abrangem o vetor - nesse caso apenas `M` e `F`. Se tivéssemos um número maior de grupos, o item `Levels` aumentaria. Veja a seguir:

```
1 my_factor <- factor(c('M','F','M','M','F','ND'))
2 print(my_factor)
```

```
R> [1] M  F  M  M  F  ND
R> Levels: F M ND
```

Um ponto importante na criação de fatores é que os `Levels` são inferidos através dos dados criados, e isso pode não corresponder à realidade. Por exemplo, observe o seguinte exemplo:

```
my_status <- factor(c('Solteiro', 'Solteiro', 'Solteiro'))
print(my_status)
```

```
R> [1] Solteiro Solteiro Solteiro
R> Levels: Solteiro
```

Nota-se que, por ocasião, os dados mostram apenas uma categoria: `Solteiro`. Entretanto, sabe-se que outra categoria do tipo `Casado` é esperada. No caso de utilizarmos o objeto `my_status` da maneira que foi definida anteriormente, omitiremos a informação de outros gêneros, e isso pode ocasionar problemas no futuro tal como a criação de gráficos incompletos. Nessa situação, o correto é definir os `Levels` manualmente da seguinte maneira:

```
my_status <- factor(c('Solteiro', 'Solteiro', 'Solteiro'),
                    levels = c('Solteiro', 'Casado'))
print(my_status)
```

```
R> [1] Solteiro Solteiro Solteiro
R> Levels: Solteiro Casado
```

9.3.2 Modificando Fatores

Um ponto importante sobre os objetos do tipo fator é que seus `Levels` são **imutáveis e não atualizam-se com a entrada de novos dados**. Em outras palavras, não é possível modificar os valores dos `Levels` após a criação do objeto. Toda nova informação que não for compatível com os `Levels` do objeto será transformada em `NA` (*Not available*) e uma mensagem de `warning` irá aparecer na tela. Essa limitação pode parecer estranha a primeira vista porém, na prática, ela evita possíveis erros no código. Veja o exemplo a seguir:

```
# set factor
my_factor <- factor(c('a', 'b', 'a', 'b'))

# change first element of a factor to 'c'
my_factor[1] <- 'c'
```

```
R> Warning in `[<-.factor`(`*tmp*`, 1, value = "c"): invalid
R> factor level, NA generated
```

```
# print result
print(my_factor)
```

```
R> [1] <NA> b    a    b
R> Levels: a b
```

Nesse caso, a maneira correta de proceder é primeiro transformar o objeto da classe fator para a classe caractere e depois realizar a conversão:

```
# set factor
my_factor <- factor(c('a', 'b', 'a', 'b'))

# change factor to character
my_char <- as.character(my_factor)

# change first element
my_char[1] <- 'c'

# mutate it back to class factor
my_factor <- factor(my_char)

# show result
print(my_factor)
```

```
R> [1] c b a b
R> Levels: a b c
```

Utilizando essas etapas temos o resultado desejado no vetor `my_factor`, com a definição de três `Levels`: `a`, `b` e `c`.

O universo **{tidyverse}** (Wickham 2023c) também possui um pacote próprio para manipular fatores, o **{forcats}** (Wickham 2023a). Para o problema atual de modificação de fatores, podemos utilizar função

forcats::**fct_recode()** . Veja um exemplo a seguir, onde trocamos as siglas dos fatores:

```
library(forcats)

# set factor
my.fct <- factor(c('A', 'B', 'C', 'A', 'C', 'M', 'N'))

# modify factors
my.fct <- fct_recode(my.fct,
                     'D' = 'A',
                     'E' = 'B',
                     'F' = 'C')

# print result
print(my.fct)
```

```
R> [1] D E F D F M N
R> Levels: D E F M N
```

Observe como o uso da função **forcats**::**fct_recode()** é intuitivo. Basta indicar o novo nome dos grupos com o operador de igualdade.

9.3.3 Convertendo Fatores para Outras Classes

Outro ponto importante no uso de fatores é a sua conversão para outras classes, especialmente a numérica. Quando convertemos um objeto de tipo fator para a classe caractere, o resultado é o esperado:

```
# create factor
my_char <-factor(c('a', 'b', 'c'))

# convert and print
print(as.character(my_char))
```

```
R> [1] "a" "b" "c"
```

Porém, quando fazemos o mesmo procedimento para a classe numérica, o que o R retorna é **longe do esperado**:

```
# set factor
my_values <- factor(5:10)

# convert to numeric (WRONG)
print(as.numeric(my_values))
```

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

Esse resultado pode ser explicado pelo fato de que, internamente, fatores são armazenados como índices, indo de 1 até o número total de `Levels`. Essa simplificação minimiza o uso da memória do computador. Quando pedimos ao R para transformar esses fatores em números, ele entende que buscamos o número do índice e não do valor. Para contornar, é fácil: basta transformar o objeto fator em caractere e, depois, em numérico, conforme mostrado a seguir:

```
# converting factors to character and then to numeric
print(as.numeric(as.character(my_values)))
```

```
R> [1]  5  6  7  8  9 10
```

> **Cuidado**
>
> Tenha muito cuidado ao transformar fatores em números. Lembre-se sempre de que o retorno da conversão direta serão os índices dos `levels` e não os valores em si. Esse é um *bug* bem particular que pode ser difícil de identificar em um código complexo.

9.3.4 Criando Tabelas de Contingência

Após a criação de um fator, podemos calcular a ocorrência de cada fator com a função **table()** . Essa também é chamada de tabela de contingência. Em um caso simples, com apenas um fator, a função **table()** conta o número de ocorrências de cada categoria, como a seguir:

```
# create factor
my_factor <- factor(sample(c('Pref', 'Ord'),
                           size = 20,
                           replace = TRUE))

```

```
# print contingency table
print(table(my_factor))
```

```
R> my_factor
R>  Ord Pref
R>    9   11
```

Um caso mais avançado do uso de **table()** é utilizar mais de um fator para a criação da tabela. Veja o exemplo a seguir:

```
# set factors
my_factor_1 <- factor(sample(c('Pref', 'Ord'),
                             size = 20,
                             replace = TRUE))

my_factor_2 <- factor(sample(paste('Grupo', 1:3),
                             size = 20,
                             replace = TRUE))

# print contingency table with two factors
print(table(my_factor_1, my_factor_2))
```

```
R>            my_factor_2
R> my_factor_1 Grupo 1 Grupo 2 Grupo 3
R>        Ord        2       4       3
R>        Pref       3       4       4
```

A tabela criada anteriormente mostra o número de ocorrências para cada combinação de fator. Essa é uma ferramenta descritiva simples, mas bastante informativa para a análise de grupos de dados.

9.3.5 Outras Funções

levels - Retorna os `Levels` de um objeto da classe fator.

```
my_factor <- factor(c('A', 'A', 'B', 'C', 'B'))
print(levels(my_factor))
```

```
R> [1] "A" "B" "C"
```

as.factor - Transforma um objeto para a classe fator.

```
my_y <- c('a','b', 'c', 'c', 'a')
my_factor <- as.factor(my_y)
print(my_factor)
```

```
R> [1] a b c c a
R> Levels: a b c
```

split - Com base em um objeto de fator, cria uma lista com valores de outro objeto. Esse comando é útil para separar dados de grupos diferentes e aplicar alguma função com **sapply()** ou `lapply`.

```
my_factor <- factor(c('A','B','C','C','C','B'))
my_x <- 1:length(my_factor)

my_l <- split(x = my_x, f = my_factor)

print(my_l)
```

```
R> $A
R> [1] 1
R>
R> $B
R> [1] 2 6
R>
R> $C
R> [1] 3 4 5
```

9.4 Valores Lógicos

Testes lógicos em dados são centrais no uso do R. Em uma única linha de código podemos testar condições para uma grande quantidade de casos. Esse cálculo é muito utilizado para encontrar casos extremos nos dados (*outliers*) e também para separar diferentes amostras de acordo com algum critério.

9.4.1 Criando Valores Lógicos

Em uma sequência de 1 até 10, podemos verificar quais são os elementos maiores que 5 com o seguinte código:

```
# set numerical
my_x <- 1:10

# print a logical test
print(my_x > 5)
```

```
R>  [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
R> [10]  TRUE
```

```
# print position of elements from logical test
print(which(my_x > 5))
```

```
R> [1]  6  7  8  9 10
```

A função **which()** do exemplo anterior retorna os índices onde a condição é verdadeira (TRUE). O uso do **which()** é recomendado quando se quer saber a posição de elementos que satisfazem alguma condição.

Para realizar testes de igualdade, basta utilizar o símbolo de igualdade duas vezes (==).

```
# create char
my_char <- rep(c('abc','bcd'), 5)

# print its contents
print(my_char)
```

```
R>  [1] "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc" "bcd" "abc"
R> [10] "bcd"
```

```
# print logical test
print(my_char == 'abc')
```

```
R>  [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
R> [10] FALSE
```

Para o teste de inigualdades, utilizamos o símbolo `!=`:

```
# print inequality test
print(my_char != 'abc')
```

```
R>  [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
R> [10]  TRUE
```

Destaca-se que também é possível testar condições múltiplas, isto é, a ocorrência simultânea de eventos. Utilizamos o operador `&` para esse propósito. Por exemplo: se quiséssemos verificar quais são os valores de uma sequência de 1 a 10 que são maiores que 4 **e** menores que 7, escreveríamos:

```
my_x <- 1:10

# print logical for values higher than 4 and lower than 7
print((my_x > 4)&(my_x < 7) )
```

```
R>  [1] FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
R> [10] FALSE
```

```
# print the actual values
idx <- which( (my_x > 4)&(my_x < 7) )
print(my_x[idx])
```

```
R> [1] 5 6
```

Para testar condições não simultâneas, isto é, ocorrências de um ou outro evento, utilizamos o operador `|`. Por exemplo: considerando a sequência anterior, acharíamos os valores maiores que 7 **ou** menores que 4 escrevendo:

```
# location of elements higher than 7 or lower than 4
idx <- which( (my_x > 7)|(my_x < 4) )

# print elements from previous condition
print(my_x[idx])
```

```
R> [1]  1  2  3  8  9 10
```

Observe que, em ambos os casos de uso de testes lógicos, utilizamos parênteses para encapsular as condições lógicas. Poderíamos ter escrito `idx <- which( my_x > 7|my_x < 4 )`, porém o uso do parênteses deixa o código mais claro ao isolar os testes de condições e sinalizar que o resultado da operação será um vetor lógico. Em alguns casos, porém, o uso do parênteses indica hierarquia na ordem das operações e portanto não pode ser ignorado.

Outro uso interessante de objetos lógicos é o teste para saber se um item ou mais pertence a um vetor ou não. Para isso utilizamos o operador `%in%`. Por exemplo, imagine que tens os *tickers* de duas ações, `c('ABC', 'DEF')` e queres saber se é possível encontrar esses tickers na coluna de outra base de dados. Essa é uma operação semelhante ao uso do teste de igualdade, porém em notação vetorial. Veja um exemplo a seguir:

```
library(dplyr)
# location of elements higher than 7 or lower than 4
my_tickers <- c('ABC', 'DEF')

# set df
n_obs <- 100
df_temp <- tibble(tickers = sample(c('ABC', 'DEF', 'GHI', 'JKL'),
                                   size = n_obs,
                                   replace = TRUE),
                  ret = rnorm(n_obs, sd = 0.05) )

# find rows with selected tickers
idx <- df_temp$tickers %in% my_tickers

# print elements from previous condition
glimpse(df_temp[idx, ])
```

```
R> Rows: 43
R> Columns: 2
R> $ tickers <chr> "ABC", "ABC", "ABC", "DEF", "DEF", "ABC", ~
R> $ ret     <dbl> 0.042864781, 0.017056405, 0.011198439, 0.0~
```

O `dataframe` mostrado na tela possui dados apenas para ações em `my_tickers`.

9.5 Datas e Tempo

A representação e manipulação de datas é um importante aspecto das pesquisas em Finanças e Economia. Manipular datas e horários de forma correta, levando em conta mudanças decorridas de horário de verão, feriados locais, em diferentes zonas de tempo, não é uma tarefa fácil! Felizmente, o R fornece um grande suporte para qualquer tipo de operação com datas e tempo.

Nesta seção estudaremos as funções e classes nativas que representam e manipulam o tempo em R. Aqui, daremos prioridade as funções do pacote **{lubridate}** (Spinu, Grolemund, e Wickham 2023). Existem, no entanto, muitos pacotes que podem ajudar o usuário a processar objetos do tipo data e tempo de forma mais avançada. Caso alguma operação com data e tempo não for encontrada aqui, sugiro o estudo dos pacotes **{chron}** (**R-chron?**), **{timeDate}** (**R-timeDate?**) e **{bizdays}** (**R-bizdays?**).

Antes de começarmos, vale relembrar que toda data no R segue o formato ISO 8601 (`YYYY-MM-DD`), onde `YYYY` é o ano em quatro números, `MM` é o mês e `DD` é o dia. Por exemplo, uma data em ISO 8601 é 2024-01-10. Deves familiarizar-se com esse formato pois toda importação de dados com formato de datas diferente desta notação exigirá conversão. Felizmente, essa operação é bastante simples de executar com o **{lubridate}** (Spinu, Grolemund, e Wickham 2023).

9.5.1 Criando Datas Simples

No R, existem diversas classes que podem representar datas. A escolha entre uma classe de datas e outra baseia-se na necessidade da pesquisa. Em muitas situações não é necessário saber o horário, enquanto que em outras isso é extremamente pertinente pois os dados são coletados ao longo de um dia.

A classe mais básica de datas é `Date`. Essa indica dia, mês e ano, apenas. No **{lubridate}** (Spinu, Grolemund, e Wickham 2023), criamos datas verificando o formato da data de entrada e as funções **lubridate**::**ymd()** (year-month-date), **lubridate**::**dmy()** (day-month-year) e **lubridate**::**mdy()** (month-day-year). Veja a seguir:

```
1 library(lubridate)
2
3 # set Date object
```

```
print(ymd('2021-06-24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(dmy('24-06-2021'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(mdy('06-24-2021'))
```

```
R> [1] "2021-06-24"
```

Note que as funções retornam exatamente o mesmo objeto. A diferença no uso é somente pela forma que a data de entrada está estruturada com a posição do dia, mês e ano.

Um benefício no uso das funções do pacote **{lubridate}** (Spinu, Grolemund, e Wickham 2023) é que as mesmas são inteligentes ao lidar com formatos diferentes. Observe no caso anterior que definimos os elementos das datas com o uso do traço (-) como separador e valores numéricos. Outros formatos também são automaticamente reconhecidos:

```
# set Date object
print(ymd('2021/06/24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(ymd('2021&06&24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(ymd('2021 june 24'))
```

```
R> [1] "2021-06-24"
```

```
# set Date object
print(dmy('24 of june 2021'))
```

```
R> [1] "2021-06-24"
```

Isso é bastante útil pois o formato de datas no Brasil é `dia/mês/ano` (DD/MM/YYYY). Ao usar **lubridate::dmy()** para uma data brasileira, a conversão é correta:

```
# set Date from dd/mm/yyyy
my_date <- dmy('24/06/2021')

# print result
print(my_date)
```

```
R> [1] "2021-06-24"
```

Já no pacote **{base}** (R Core Team 2023b), a função correspondente é **as.Date()** . O formato da data, porém, deve ser explicitamente definido com argumento **format()** , conforme mostrado a seguir:

```
# set Date from dd/mm/yyyy with the definition of format
my_date <- as.Date('24/06/2021', format = '%d/%m/%Y')

# print result
print(my_date)
```

```
R> [1] "2021-06-24"
```

Os símbolos utilizados na entrada **format()** , tal como `%d` e `%Y`, são indicadores de formato, os quais definem a forma em que a data a ser convertida está estruturada. Nesse caso, os símbolos `%Y`, `%m` e `%d` definem ano, mês e dia, respectivamente. Existem diversos outros símbolos que podem ser utilizados para processar datas em formatos específicos. Um panorama das principais codificações é apresentado a seguir:

Código	Valor	Exemplo
%d	dia do mês (decimal)	0
%m	mês (decimal)	12
%b	mês (abreviado)	Abr
%B	mês (nome completo)	Abril

Código	Valor	Exemplo
%y	ano (2 dígitos)	16
%Y	ano (4 dígitos)	2021

Os símbolos anteriores permitem a criação de datas a partir de variados formatos. Observe como a utilização das funções do **{lubridate}** (Spinu, Grolemund, e Wickham 2023), em relação a **{base}** (R Core Team 2023b), são mais simples e fáceis de utilizar, justificando a nossa escolha.

9.5.2 Criando Sequências de Datas

Um aspecto interessante no uso de objetos do tipo `Date` é que eles interagem com operações de adição de valores numéricos e com testes lógicos de comparação de datas. Por exemplo: caso quiséssemos adicionar um dia à data `my_date` criada anteriormente, bastaria somar o valor 1 ao objeto:

```
# create date
my_date <- ymd('2021-06-24')

# find next day
my_date_2 <- my_date + 1

# print result
print(my_date_2)
```

```
R> [1] "2021-06-25"
```

A propriedade também funciona com vetores, o que deixa a criação de sequências de datas muito fácil. Nesse caso, o próprio R encarrega-se de verificar o número de dias em cada mês.

```
# create a sequence of Dates
my_date_vec <- my_date + 0:15

# print it
print(my_date_vec)
```

```
R>  [1] "2021-06-24" "2021-06-25" "2021-06-26" "2021-06-27"
R>  [5] "2021-06-28" "2021-06-29" "2021-06-30" "2021-07-01"
R>  [9] "2021-07-02" "2021-07-03" "2021-07-04" "2021-07-05"
R> [13] "2021-07-06" "2021-07-07" "2021-07-08" "2021-07-09"
```

Uma maneira mais customizável de criar sequências de datas é utilizar a função **seq()** . Com ela, é possível definir intervalos diferentes de tempo e até mesmo o tamanho do vetor de saída. Caso quiséssemos uma sequência de datas de dois em dois dias, poderíamos utilizar o seguinte código:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-03-20')

# set sequence
my_date_date <- seq(from = my_date_1,
                    to = my_date_2,
                    by = '2 days')

# print result
print(my_date_date)
```

```
R> [1] "2021-03-07" "2021-03-09" "2021-03-11" "2021-03-13"
R> [5] "2021-03-15" "2021-03-17" "2021-03-19"
```

Caso quiséssemos de duas em duas semanas, escreveríamos:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-04-20')

# set sequence
my_date_date <- seq(from = my_date_1,
                    to = my_date_2,
                    by = '2 weeks')

# print result
print(my_date_date)
```

```
R> [1] "2021-03-07" "2021-03-21" "2021-04-04" "2021-04-18"
```

Outra forma de utilizar **seq()** é definir o tamanho desejado do objeto de saída. Por exemplo, caso quiséssemos um vetor de datas com 10 elementos, usaríamos:

```
# set first and last Date
my_date_1 <- ymd('2021-03-07')
my_date_2 <- ymd('2021-10-20')

# set sequence
my_date_vec <- seq(from = my_date_1,
                   to = my_date_2,
                   length.out = 10)

# print result
print(my_date_vec)
```

```
R>  [1] "2021-03-07" "2021-04-01" "2021-04-26" "2021-05-21"
R>  [5] "2021-06-15" "2021-07-11" "2021-08-05" "2021-08-30"
R>  [9] "2021-09-24" "2021-10-20"
```

O intervalo entre as datas em `my_date_vec` é definido automaticamente pelo R.

9.5.3 Operações com Datas

É possível descobrir a diferença de dias entre datas simplesmente diminuindo uma data da outra:

```
# set dates
my_date_1 <- ymd('2015-06-24')
my_date_2 <- ymd('2016-06-24')

# calculate difference
diff_date <- my_date_2 - my_date_1

# print result
print(diff_date)
```

```
R> Time difference of 366 days
```

A saída da operação de subtração é um objeto da classe `diffdate`, o qual possui a classe de lista como sua estrutura básica. Destaca-se que a notação de acesso aos elementos da classe `diffdate` é a mesma utilizada para listas. O valor numérico do número de dias está contido no primeiro elemento de `diff_date`:

```
# print difference of days as numerical value
print(diff_date[[1]])
```

```
R> [1] 366
```

Podemos testar se uma data é maior do que outra com o uso das operações de comparação:

```
# set date and vector
my_date_1 <- ymd('2016-06-20')
my_date_vec <- ymd('2016-06-20') + seq(-5,5)

# test which elements of my_date_vec are older than my_date_1
my_test <- (my_date_vec > my_date_1)

# print result
print(my_test)
```

```
R>  [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE
R> [10]  TRUE  TRUE
```

A operação anterior é bastante útil quando se está buscando filtrar um determinado período de tempo nos dados. Nesse caso, basta buscar nas datas o período específico em que estamos interessados e utilizar o objeto lógico da comparação para selecionar os elementos. Veja o exemplo a seguir:

```
library(dplyr)
library(lubridate)

# set first and last dates
first_date <- ymd('2016-06-01')
last_date <- ymd('2016-06-15')

# create `dataframe` and glimpse it
my_temp_df <- tibble(date_vec = ymd('2016-05-25') + seq(0,30),
```

```
                        prices=seq(1,10,
                                   length.out = length(date_vec)))

glimpse(my_temp_df)
```

```
R> Rows: 31
R> Columns: 2
R> $ date_vec <date> 2016-05-25, 2016-05-26, 2016-05-27, 2016~
R> $ prices   <dbl> 1.0, 1.3, 1.6, 1.9, 2.2, 2.5, 2.8, 3.1, 3~
```

```
# find dates that are between the first and last date
my_idx <- (my_temp_df$date_vec >= first_date) &
  (my_temp_df$date_vec <= last_date)

# use index to filter `dataframe`
my_temp_df_filtered <- my_temp_df |>
  filter(my_idx) |>
  glimpse()
```

```
R> Rows: 15
R> Columns: 2
R> $ date_vec <date> 2016-06-01, 2016-06-02, 2016-06-03, 2016~
R> $ prices   <dbl> 3.1, 3.4, 3.7, 4.0, 4.3, 4.6, 4.9, 5.2, 5~
```

Nesse caso, o vetor final de preços da coluna `prices` contém apenas informações para o período entre `first_date` e `last_date`.

9.5.4 Lidando com Data e Tempo

O uso da classe `Date` é suficiente quando se está lidando apenas com datas. Em casos em que é necessário levar em consideração o horário, temos que utilizar um objeto do tipo `datetime`.

No pacote **{base}** (R Core Team 2023b), uma das classes utilizadas para esse fim é a `POSIXlt`, a qual armazena o conteúdo de uma data na forma de uma lista. Outra classe que também é possível utilizar é a `POSIXct`, que armazena as datas como segundos contados a partir de `1970-01-01`. Devido ao seu formato de armazenamento, a classe `POSIXct` ocupa menos memória do computador. Junto ao **{lubridate}** (Spinu, Grolemund, e Wickham 2023), a classe utilizada para representar data-tempo é `POSIXct`

e portanto daremos prioridade a essa. Vale destacar que todos os exemplos apresentados aqui também podem ser replicados para objetos do tipo `POSIXlt`.

O formato tempo/data também segue a norma ISO 8601, sendo representado como `ano-mês-dia horas:minutos:segundos zonadetempo` (YYYY-MM-DD HH:mm:SS TMZ). Veja o exemplo a seguir:

```
# creating a POSIXct object
my_timedate <- as.POSIXct('2021-01-01 16:00:00')

# print result
print(my_timedate)
```

```
R> [1] "2021-01-01 16:00:00 -03"
```

O pacote **{lubridate}** (Spinu, Grolemund, e Wickham 2023) também oferece funções inteligentes para a criação de objetos do tipo data-tempo. Essas seguem a mesma linha de raciocínio que as funções de criar datas. Veja a seguir:

```
library(lubridate)

# creating a POSIXlt object
my_timedate <- ymd_hms('2021-01-01 16:00:00')

# print it
print(my_timedate)
```

```
R> [1] "2021-01-01 16:00:00 UTC"
```

Destaca-se que essa classe adiciona automaticamente o fuso horário. Caso seja necessário representar um fuso diferente, é possível fazê-lo com o argumento `tz`:

```
# creating a POSIXlt object with custom timezone
my_timedate_tz <- ymd_hms('2021-01-01 16:00:00',
                          tz = 'GMT')

# print it
print(my_timedate_tz)
```

```
R> [1] "2021-01-01 16:00:00 GMT"
```

É importante ressaltar que, para o caso de objetos do tipo `POSIXlt` e `POSIXct`, **as operações de soma e diminuição referem-se a segundos** e não dias, como no caso do objeto da classe `Date`.

```
1 # Adding values (seconds) to a POSIXlt object and printing it
2 print(my_timedate_tz + 30)
```

```
R> [1] "2021-01-01 16:00:30 GMT"
```

Assim como para a classe `Date`, existem símbolos específicos para lidar com componentes de um objeto do tipo data/tempo. Isso permite a formatação customizada de datas. A seguir, apresentamos um quadro com os principais símbolos e os seus respectivos significados.

Código	Valor	Exemplo
%H	Hora (decimal, 24 horas)	23
%I	Hora (decimal, 12 horas)	11
%M	Minuto (decimal, 0-59)	12
%p	Indicador AM/PM	AM
%S	Segundos (decimal, 0-59)	50

A seguir veremos como utilizar essa tabela para customizar datas.

9.5.5 Personalizando o Formato de Datas

A notação básica para representar datas e data/tempo no R pode não ser a ideal em algumas situações. No Brasil, por exemplo, indicar datas no formato `YYYY-MM-DD` pode gerar bastante confusão em um relatório formal. É recomendado, portanto, modificar a representação das datas para o formato esperado, isto é, `DD/MM/YYYY`.

Para formatar uma data, utilizamos a função **format()** . Seu uso baseia-se nos símbolos de data e de horário apresentados anteriormente. A partir desses, pode-se criar qualquer customização. Veja o exemplo a seguir, onde apresenta-se a modificação de um vetor de datas para o formato brasileiro:

```
# create vector of dates
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-01-15'),
                by = '1 day')

# change format
my_dates_br <- format(my_dates, '%d/%m/%Y')

# print result
print(my_dates_br)
```

```
R>  [1] "01/01/2021" "02/01/2021" "03/01/2021" "04/01/2021"
R>  [5] "05/01/2021" "06/01/2021" "07/01/2021" "08/01/2021"
R>  [9] "09/01/2021" "10/01/2021" "11/01/2021" "12/01/2021"
R> [13] "13/01/2021" "14/01/2021" "15/01/2021"
```

O mesmo procedimento pode ser realizado para objetos do tipo data/tempo (`POSIXct`):

```
# create vector of date-time
my_datetime <- ymd_hms('2021-01-01 12:00:00') + seq(0,560,60)

# change to Brazilian format
my_dates_br <- format(my_datetime, '%d/%m/%Y %H:%M:%S')

# print result
print(my_dates_br)
```

```
R>  [1] "01/01/2021 12:00:00" "01/01/2021 12:01:00"
R>  [3] "01/01/2021 12:02:00" "01/01/2021 12:03:00"
R>  [5] "01/01/2021 12:04:00" "01/01/2021 12:05:00"
R>  [7] "01/01/2021 12:06:00" "01/01/2021 12:07:00"
R>  [9] "01/01/2021 12:08:00" "01/01/2021 12:09:00"
```

Pode-se também customizar para formatos bem específicos. Veja a seguir:

```
# set custom format
my_dates_custom <- format(my_dates,
                          'Year=%Y | Month=%m | Day=%d')

```

```
# print result
print(my_dates_custom)
```

```
R>  [1] "Year=2021 | Month=01 | Day=01"
R>  [2] "Year=2021 | Month=01 | Day=02"
R>  [3] "Year=2021 | Month=01 | Day=03"
R>  [4] "Year=2021 | Month=01 | Day=04"
R>  [5] "Year=2021 | Month=01 | Day=05"
R>  [6] "Year=2021 | Month=01 | Day=06"
R>  [7] "Year=2021 | Month=01 | Day=07"
R>  [8] "Year=2021 | Month=01 | Day=08"
R>  [9] "Year=2021 | Month=01 | Day=09"
R> [10] "Year=2021 | Month=01 | Day=10"
R> [11] "Year=2021 | Month=01 | Day=11"
R> [12] "Year=2021 | Month=01 | Day=12"
R> [13] "Year=2021 | Month=01 | Day=13"
R> [14] "Year=2021 | Month=01 | Day=14"
R> [15] "Year=2021 | Month=01 | Day=15"
```

9.5.6 Extraindo Elementos de uma Data

Para extrair elementos de datas tal como o ano, mês, dia, hora, minuto e segundo, uma alternativa é utilizar função **format()** . Observe o próximo exemplo, onde recuperamos apenas as horas de um objeto `POSIXct`:

```
library(lubridate)

# create vector of date-time
my_datetime <- seq(from = ymd_hms('2021-01-01 12:00:00'),
                   to = ymd_hms('2021-01-01 18:00:00'),
                   by = '1 hour')

# get hours from POSIXlt
my_hours <- as.numeric(format(my_datetime, '%H'))

# print result
print(my_hours)
```

```
R> [1] 12 13 14 15 16 17 18
```

Da mesma forma, poderíamos utilizar os símbolos `%M` e `%S` para recuperar facilmente minutos e segundos de um vetor de objetos `POSIXct`.

```
# create vector of date-time
my_datetime <- seq(from = ymd_hms('2021-01-01 12:00:00'),
                   to = ymd_hms('2021-01-01 18:00:00'),
                   by = '15 min')

# get minutes from POSIXlt
my_minutes <- as.numeric(format(my_datetime, '%M'))

# print result
print(my_minutes)
```

```
R>  [1]  0 15 30 45  0 15 30 45  0 15 30 45  0 15 30 45  0 15
R> [19] 30 45  0 15 30 45  0
```

Outra forma é utilizar as funções do **{lubridate}** (Spinu, Grolemund, e Wickham 2023), tal como **lubridate**::**hour()** e **lubridate**::**minute()** :

```
# get hours with lubridate
print(hour(my_datetime))
```

```
R>  [1] 12 12 12 12 13 13 13 13 14 14 14 14 15 15 15 15 16 16
R> [19] 16 16 17 17 17 17 18
```

```
# get minutes with lubridate
print(minute(my_datetime))
```

```
R>  [1]  0 15 30 45  0 15 30 45  0 15 30 45  0 15 30 45  0 15
R> [19] 30 45  0 15 30 45  0
```

Outras funções também estão disponíveis para os demais elementos de um objeto data-hora.

9.5.7 Conhecendo o Horário e a Data Atual

O R inclui várias funções que permitem o usuário utilizar no seu código o horário e data atual do sistema. Isso é bastante útil quando se está criando registros e é importante que a data e horário de execução do código seja conhecida futuramente.

Para conhecer o dia atual, basta utilizarmos a função **Sys.Date()** ou **lubridate::today()** :

```
library(lubridate)

# get today
print(Sys.Date())
```

```
R> [1] "2024-01-10"
```

```
# print it
print(today())
```

```
R> [1] "2024-01-10"
```

Para descobrir a data e horário, utilizamos a função **Sys.time()** ou **lubridate::now()** :

```
# get time!
print(Sys.time())
```

```
R> [1] "2024-01-10 16:21:44 -03"
```

```
# get time!
print(now())
```

```
R> [1] "2024-01-10 16:21:44 -03"
```

Com base nessas, podemos escrever:

```
library(stringr)

# example of log message
my_str <- str_c('This code was executed in ', now())

# print it
print(my_str)
```

```
R> [1] "This code was executed in 2024-01-10 16:21:44.189303"
```

9.5.8 Outras Funções Úteis

weekdays - Retorna o dia da semana de uma ou várias datas.

```
# set date vector
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-01-5'),
                by = '1 day')

# find corresponding weekdays
my_weekdays <- weekdays(my_dates)

# print it
print(my_weekdays)
```

```
R> [1] "Friday"   "Saturday" "Sunday"   "Monday"   "Tuesday"
```

months - Retorna o mês de uma ou várias datas.

```
# create date vector
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-12-31'),
                by = '1 month')

# find months
my_months <- months(my_dates)

# print result
print(my_months)
```

```
R>  [1] "January"   "February"  "March"     "April"
R>  [5] "May"       "June"      "July"      "August"
R>  [9] "September" "October"   "November"  "December"
```

quarters - Retorna a localização de uma ou mais datas dentro dos quartis do ano.

```
# get quartiles of the year
my_quarters <- quarters(my_dates)
print(my_quarters)
```

```
R>  [1] "Q1" "Q1" "Q1" "Q2" "Q2" "Q2" "Q3" "Q3" "Q3" "Q4" "Q4"
R> [12] "Q4"
```

OlsonNames - Retorna um vetor com as zonas de tempo disponíveis no R. No total, são mais de 500 itens. Aqui, apresentamos apenas os primeiros cinco elementos.

```
# get possible timezones
possible_tz <- OlsonNames()

# print it
print(possible_tz[1:5])
```

```
R> [1] "Africa/Abidjan"     "Africa/Accra"
R> [3] "Africa/Addis_Ababa" "Africa/Algiers"
R> [5] "Africa/Asmara"
```

Sys.timezone - Retorna a zona de tempo do sistema.

```
# get current timezone
print(Sys.timezone())
```

```
R> [1] "America/Sao_Paulo"
```

cut - Retorna um fator a partir da categorização de uma classe de data e tempo.

```
# set example date vector
my_dates <- seq(from = ymd('2021-01-01'),
                to = ymd('2021-03-01'),
                by = '5 days')

# group vector based on monthly breaks
my_month_cut <- cut(x = my_dates,
                    breaks = 'month')

# print result
print(my_month_cut)
```

```
R>  [1] 2021-01-01 2021-01-01 2021-01-01 2021-01-01 2021-01-01
R>  [6] 2021-01-01 2021-01-01 2021-02-01 2021-02-01 2021-02-01
R> [11] 2021-02-01 2021-02-01
```

```
R> Levels: 2021-01-01 2021-02-01
```

```
# set example datetime vector
my_datetime <- as.POSIXlt('2021-01-01 12:00:00') + seq(0,250,15)

# set groups for each 30 seconds
my_cut <- cut(x = my_datetime, breaks = '30 secs')

# print result
print(my_cut)
```

```
R>  [1] 2021-01-01 12:00:00 2021-01-01 12:00:00
R>  [3] 2021-01-01 12:00:30 2021-01-01 12:00:30
R>  [5] 2021-01-01 12:01:00 2021-01-01 12:01:00
R>  [7] 2021-01-01 12:01:30 2021-01-01 12:01:30
R>  [9] 2021-01-01 12:02:00 2021-01-01 12:02:00
R> [11] 2021-01-01 12:02:30 2021-01-01 12:02:30
R> [13] 2021-01-01 12:03:00 2021-01-01 12:03:00
R> [15] 2021-01-01 12:03:30 2021-01-01 12:03:30
R> [17] 2021-01-01 12:04:00
R> 9 Levels: 2021-01-01 12:00:00 ... 2021-01-01 12:04:00
```

9.6 Dados Omissos - `NA` (*Not available*)

Uma das principais inovações do R em relação a outras linguagens de programação é a representação de dados omissos através de objetos da classe `NA` (*Not Available*). A falta de dados pode ter inúmeros motivos, tal como a falha na coleta de informações ou simplesmente a não existência dos mesmos. Esses casos são tratados por meio da remoção ou da substituição dos dados omissos antes realizar uma análise mais profunda. A identificação desses casos, portanto, é de extrema importância.

9.6.1 Definindo Valores `NA`

Para definirmos os casos omissos nos dados, basta utilizar o símbolo `NA`:

```
# a vector with NA
my_x <- c(1, 2, NA, 4, 5)

```

```
# print it
print(my_x)
```

```
R> [1]  1  2 NA  4  5
```

Vale destacar que a operação de qualquer valor NA com outro sempre resultará em NA.

```
# example of NA interacting with other objects
print(my_x + 1)
```

```
R> [1]  2  3 NA  5  6
```

Isso exige cuidado quando se está utilizando alguma função com cálculo recursivo, tal como **cumsum()** e **cumprod()** . Nesses casos, todo valor consecutivo ao NA será transformado em NA. Veja os exemplos a seguir com as duas funções:

```
# set vector with NA
my_x <- c(1:5, NA, 5:10)

# print cumsum (NA after sixth element)
print(cumsum(my_x))
```

```
R>  [1]  1  3  6 10 15 NA NA NA NA NA NA NA
```

```
# print cumprod (NA after sixth element)
print(cumprod(my_x))
```

```
R>  [1]   1   2   6  24 120  NA  NA  NA  NA  NA  NA  NA
```

> **Cuidado**
>
> Toda vez que utilizar as funções **cumsum()** e **cumprod()** , certifique-se de que não existe algum valor NA no vetor de entrada. Lembre-se de que todo NA é contagiante e o cálculo recursivo irá resultar em um vetor repleto de dados faltantes.

9.6.2 Encontrando e Substituindo Valores NA

Para encontrar os valores NA em um vetor, basta utilizar a função **is.na()** :

```
# set vector with NA
my_x <- c(1:2, NA, 4:10)

# find location of NA
idx_na <- is.na(my_x)
print(idx_na)
```

```
R>  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
R> [10] FALSE
```

Para substituí-los, use indexação com a saída de **is.na()** :

```
# set vector
my_x <- c(1, NA, 3:4, NA)

# replace NA for 2
my_x[is.na(my_x)] <- 2

# print result
print(my_x)
```

```
R> [1] 1 2 3 4 2
```

Outra maneira de limpar o objeto é utilizar a função **na.omit()** , que retorna o mesmo objeto mas sem os valores NA. Note, porém, que o tamanho do vetor irá mudar e o objeto será da classe omit, o que indica que o vetor resultante não inclui os NA e apresenta, também, a posição dos elementos NA encontrados.

```
# set vector
my_char <- c(letters[1:3], NA, letters[5:8])

# print it
print(my_char)
```

```
R> [1] "a" "b" "c" NA  "e" "f" "g" "h"
```

```
# use na.omit to remove NA
my_char <- na.omit(my_char)

# print result
print(my_char)
```

```
R> [1] "a" "b" "c" "e" "f" "g" "h"
R> attr(,"na.action")
R> [1] 4
R> attr(,"class")
R> [1] "omit"
```

Apesar de o tipo de objeto ter sido trocado, devido ao uso de **na.omit()** , as propriedades básicas do vetor inicial se mantêm. Por exemplo: o uso de **nchar()** no objeto resultante é possível.

```
# trying nchar on a na.omit object
print(nchar(my_char))
```

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

Para outros objetos, porém, recomenda-se cautela quando no uso da função **na.omit()** .

9.6.3 Outras Funções Úteis

complete.cases - Retorna um vetor lógico que indica se as linhas do objeto possuem apenas valores não omissos. Essa função é usada exclusivamente para `dataframes` e matrizes.

```
# create matrix
my_mat <- matrix(1:15, nrow = 5)

# set an NA value
my_mat[2,2] <- NA

# print index with rows without NA
print(complete.cases(my_mat))
```

```
R> [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

9.7 Exercícios

CAPÍTULO 10

INTRODUÇÃO A PROGRAMAÇÃO COM O R

TODO: write more

REFERÊNCIAS BIBLIOGRÁFICAS

Allaire, JJ. 2023. *quarto: R Interface to Quarto Markdown Publishing System*. https://github.com/quarto-dev/quarto-r.

Allaire, JJ, Yihui Xie, Christophe Dervieux, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, et al. 2023. *rmarkdown: Dynamic Documents for R*. https://github.com/rstudio/rmarkdown.

Dragulescu, Adrian, e Cole Arendt. 2020. *xlsx: Read, Write, Format Excel 2007 and Excel 97/2000/XP/2003 Files*. https://github.com/colearendt/xlsx.

Garmonsway, Duncan. 2023. *tidyxl: Read Untidy Excel Files*. https://github.com/nacnudus/tidyxl.

Hester, Jim, e Jennifer Bryan. 2022. *glue: Interpreted String Literals*. https://github.com/tidyverse/glue.

Hester, Jim, Hadley Wickham, e Gábor Csárdi. 2023. *fs: Cross-Platform File System Operations Based on libuv*. https://fs.r-lib.org.

Iannone, Richard, Joe Cheng, Barret Schloerke, Ellis Hughes, Alexandra Lauer, e JooYoung Seo. 2023. *gt: Easily Create Presentation-Ready Display Tables*. https://gt.rstudio.com.

Klik, Mark. 2022. *fst: Lightning Fast Serialization of Data Frames*. http://www.fstpackage.org.

Mirai Solutions GmbH. 2023. *XLConnect: Excel Connector for R*. https://mirai-solutions.ch https://github.com/miraisolutions/xlconnect.

Müller, Kirill, e Hadley Wickham. 2023. *tibble: Simple Data Frames*. https://tibble.tidyverse.org/.

Ooms, Jeroen. 2023. *writexl: Export Data Frames to Excel xlsx Format*. https://docs.ropensci.org/writexl/.

Perlin, Marcelo. 2022. *GetBCBData: Imports Datasets from BCB (Central*

Bank of Brazil) using Its Official API. https://github.com/msperlin/GetBCBData/.

———. 2023a. *GetTDData: Get Data for Brazilian Bonds (Tesouro Direto)*. https://github.com/msperlin/GetTDData/.

———. 2023b. *yfR: Downloads and Organizes Financial Data from Yahoo Finance*. https://github.com/ropensci/yfR.

Perlin, Marcelo S. 2023. *GetQuandlData: Fast and Cached Import of Data from Quandl Using the json API*. https://github.com/msperlin/GetQuandlData/.

Perlin, Marcelo, e Guilherme Kirch. 2022. *GetFREData: Reading FRE Corporate Data of Public Traded Companies from B3*. https://github.com/msperlin/GetFREData/.

———. 2023. *GetDFPData2: Reading Annual and Quarterly Financial Reports from B3*. https://github.com/msperlin/GetDFPData2/.

R Core Team. 2023a. *foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ...* https://svn.r-project.org/R-packages/trunk/foreign/.

———. 2023c. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

———. 2023b. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. https://www.R-project.org/.

Spinu, Vitalie, Garrett Grolemund, e Hadley Wickham. 2023. *lubridate: Make Dealing with Dates a Little Easier*. https://lubridate.tidyverse.org.

Teetor, Paul. 2011. *R cookbook: Proven recipes for data analysis, statistics, and graphics*. " O'Reilly Media, Inc.".

Ushey, Kevin, JJ Allaire, e Yuan Tang. 2023. *reticulate: Interface to Python*. https://rstudio.github.io/reticulate/.

Ushey, Kevin, e Hadley Wickham. 2023. *renv: Project Environments*. https://rstudio.github.io/renv/.

Wickham, Hadley. 2019. *Advanced r*. CRC press.

———. 2023a. *forcats: Tools for Working with Categorical Variables (Factors)*. https://forcats.tidyverse.org/.

———. 2023b. *stringr: Simple, Consistent Wrappers for Common String Operations*. https://stringr.tidyverse.org.

———. 2023c. *tidyverse: Easily Install and Load the Tidyverse*. https://tidyverse.tidyverse.org.

Wickham, Hadley, e Jennifer Bryan. 2023. *readxl: Read Excel Files*. https://readxl.tidyverse.org.

Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen,

Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, e Dewey Dunnington. 2023. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. https://ggplot2.tidyverse.org.

Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, e Davis Vaughan. 2023. *dplyr: A Grammar of Data Manipulation*. https://dplyr.tidyverse.org.

Wickham, Hadley, e Lionel Henry. 2023. *purrr: Functional Programming Tools*. https://purrr.tidyverse.org/.

Wickham, Hadley, Jim Hester, e Jennifer Bryan. 2023. *readr: Read Rectangular Text Data*. https://readr.tidyverse.org.

Wickham, Hadley, Davis Vaughan, e Maximilian Girlich. 2023. *tidyr: Tidy Messy Data*. https://tidyr.tidyverse.org.

Xie, Yihui. 2023. *knitr: A General-Purpose Package for Dynamic Report Generation in R*. https://yihui.org/knitr/.

Zeileis, Achim, Bettina Gruen, Friedrich Leisch, e Nikolaus Umlauf. 2022. *exams: Automatic Generation of Exams in R*. https://www.R-exams.org/.

ÍNDICE