

Capítulo 3 Importação

Até agora, vimos como criar e usar objetos e bases de dados R – mas isto nem de longe cobre as habilidades necessárias para realizar uma pesquisa. Na verdade, ainda não aprendemos algo essencial: carregar nossas próprias bases de dados para o R. Certamente existem outras coisas úteis para se aprender no R, mas, para os nossos objetivos, esta é quase obrigatória.

Neste capítulo, portanto, veremos como carregar os mais diversos tipos de dados no R – desde aquela planilha velha do *Excel* até formatos mais modernos, como *.json* – com os mais diversos tamanhos – de poucos *kbytes* até de dezenas de *gibabytes*. Com isto, estaremos prontos para por dentro do R os mais diversos tipos de informação para fazer nossas análises: textos, bancos de dados criados em outros *softwares*, mega bancos de dados, arquivos com formatos específicos, entre outros.

Para ilustrar este conteúdo, carregaremos vários arquivos, que estão disponíveis na página de materiais complementares deste livro. O que veremos em seguida, também, pressupõe conhecimento básico sobre `data.frame` no R; caso tenha algumas dúvidas sobre isto, o capítulo 2 é o melhor lugar para começar.

3.1 A importação de dados no R

Só há um segredo para se aprender quando o assunto é carregamento de dados no R: o de que cada tipo de arquivo geralmente requer uma solução específica para carregá-lo (mas veremos exceções). Além disto, também precisamos considerar dois maiores problemas ao fazer este tipo de tarefa. O primeiro deles é o limite de memória do computador, já que, no R, podemos carregar dados até o limite da memória RAM disponível no computador. Já o segundo diz respeito a lidar com erros de acentuação e de reconhecimento de caracteres em cada base que formos trabalhar. Para a nossa sorte, o R nos oferece soluções simples para contornar estes e outros problemas, que veremos na parte final deste capítulo.

Antes de prosseguir, precisaremos instalar alguns pacotes que nos ajudarão a carregar dados¹. Estes pacotes são:

- `dplyr`, para manipular `data.frame`, que já vimos no capítulo 1;
- `readr`, para carregar dados em formato tabular (planilhas em geral);

- `readxl` , para carregar planilhas do *Excel*;
- `readODS` , para carregar planilhas *Open Document*;
- `haven` , para importar dados do *SPSS* e *Stata*; e
- `rio` , para importar diversos tipos de dados.

Para instalar estes pacotes, use `install.packages("nome_do_pacote")` :

```
install.packages("readODS")
install.packages("readxl")
install.packages("haven")
install.packages("dplyr")
install.packages("readr")
install.packages("rio")
```

3.2 Carregando dados

Temos várias formas de salvar informações num computador. Podemos, por exemplo, escrever um texto no *Word* ou *Libre Office* e salvá-lo num arquivo chamado `Meu texto.doc` , assim como podemos criar uma planilha no *Excel* e salvá-la no arquivo `Minha planilha.xls` . O importante aqui é que *da mesma forma que cada um destes programas serve para trabalhar com um tipo específico de arquivo, no R também precisaremos de ferramentas específicas para abrir cada tipo de arquivo*. Às vezes, faremos isto usando funções diferentes. Em outros casos, apenas precisaremos dizer para o R como ele deve proceder – qual *encoding* ele deve usar, onde ficarão os nomes das variáveis, qual é o tipo de delimitar de texto que deverá ser usado, entre outros.

A primeira que precisamos saber, portanto, é: quais soluções deveremos usar para cada tipo de arquivo. Existem formas simples de identificar isto, mas elas pressupõe saber a extensão do arquivo que queremos abrir (as letras depois do ponto ao final do nome do arquivo, e.g., `.doc` , `.xlsx` , etc.), que indicam qual é o seu formato. No *Windows*, podemos descobrir a extensão de um arquivo simplesmente clicando com o botão direito do *mouse* em cima dele e, depois, na opção “Propriedades” no menu que será aberto; feito isto, a extensão do arquivo será exibida logo acima (no campo de texto destacado em azul).

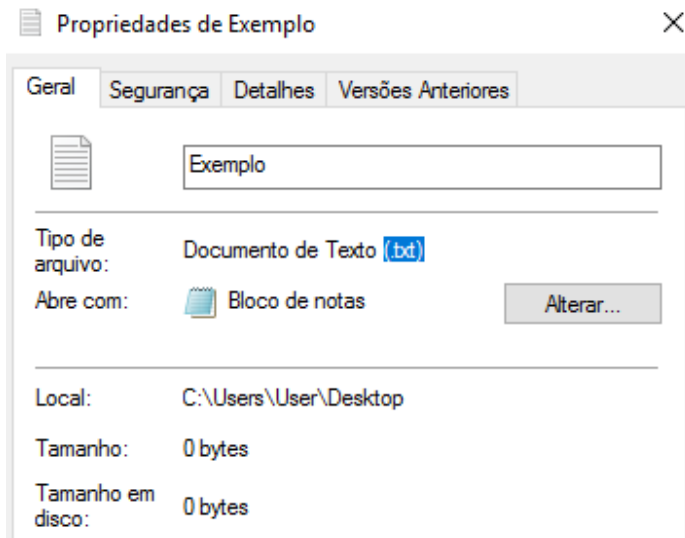


Figura 3.1: Extensão de um arquivo no Windows

Para orientação geral, segue abaixo um resumo dos principais tipos de arquivos de dados, geralmente usados em análises, que aprenderemos a abrir neste capítulo com suas respectivas extensões – e funções e pacotes que usaremos para carregá-los no R.

Tabela 3.1: Tipos de arquivos, suas extensões e funções usadas para carregá-los no R

Arquivo	Extensão	Pacote	Função
Texto delimitado	.txt	readr	read_delim
Texto delimitado	.csv	readr	read_delim , read_csv
Planilha do Excel	xls , xlsx , .ods	readxl , readODS	read_excel , read.ods
Banco de dados do SPSS	.sav , .por	haven	read_sav , read_por
Banco de dados do Stata	.dta	haven	read_dta
Banco de dados do SAS	.sas7bdat	haven	read_sas
JSON	.json	rio	import
R Data	.Rda	-	load

Apesar de parecer muita coisa, a mecânica geral de carregar dados é mais ou menos a mesma para qualquer tipo de arquivo: se aprendermos a usar uma solução, provavelmente saberemos usar as demais. A ideia básica, detalhada em seguida, é mais ou menos esta:

```
objeto <- nome_da_funcao("nome_do_arquivo.extensao", outros_argumentos...)
```

3.2.1 Arquivos de texto delimitado

Começaremos carregando um dos tipos de arquivos mais comuns no R: o `.csv`, de *comma-separated values*, ou valores separados por vírgulas. Além de simples, este formato é flexível (pode ser salvo também em arquivo com extensão `.tab`, `.txt`, etc.) e intuitivo: cada observação no banco (linha) é separada por quebra de parágrafo (nova linha) e cada variável (coluna) é separada por um caractere fixo (como ponto e vírgula ou vírgula). É possível abrir diretamente estes arquivos com algum editor de texto simples para ver isto:

```
"Var1", "Var2"  
1,100  
2,99  
3,98  
4,97  
5,96  
6,95  
7,94  
8,93  
9,92  
10,91  
11,90  
12,89  
13,88  
14,87  
15,86  
16,85  
17,84  
18,83
```

Figura 3.2: Exemplo de arquivo `.csv`

Como é possível notar, temos duas variáveis neste arquivo: “Var1” e “Var2”. Cada linha é uma observação, e os valores de cada variáveis estão separados por uma vírgula. De forma geral, esta é a forma como dados são salvos neste tipo de arquivo – precisamos apenas saber qual é o separador das colunas (no caso, vírgula).

Para carregar este arquivo, usamos a função `read_delim` do pacote `readr`. O procedimento é simples: passamos para a função o nome do arquivo, que deverá estar no diretório corrente de trabalho do R (ou passar o endereço do arquivo no computador), e o delimitador de colunas para o argumento `delim = .`

```
# Carrega o pacote readr
library(readr)

# Carrega os dados do arquivo "exemplo.csv"
meu_banco <- read_delim("exemplo.csv", delim = ",")
```

O código acima já salva os dados do arquivo no objeto chamado `meu_banco`, que é `data.frame`. Com isto, podemos usar a função `glimpse` do pacote `dplyr` para visualizar a estrutura do banco.

```
# Carrega o pacote dplyr
library(dplyr)

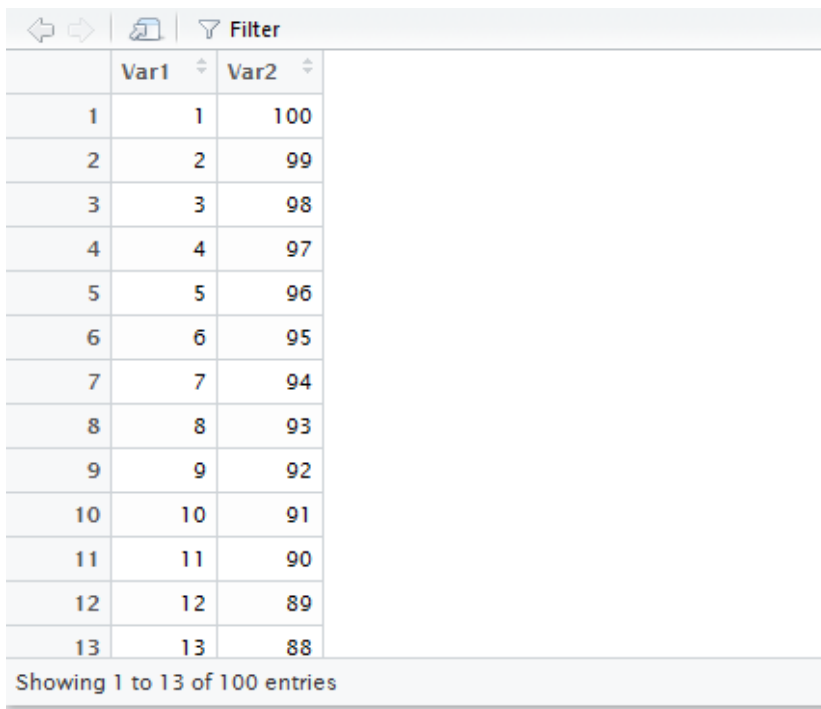
# Visualiza a estrutura do banco
glimpse(meu_banco)
```

```
## Observations: 100
## Variables: 2
## $ Var1 <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17...
## $ Var2 <dbl> 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, ...
```

Ou podemos usar a função `view` para visualizar os dados do banco.

```
View(meu_banco)
```

O que deverá abrir uma nova aba no *RStudio* semelhante a essa:



	Var1	Var2
1	1	100
2	2	99
3	3	98
4	4	97
5	5	96
6	6	95
7	7	94
8	8	93
9	9	92
10	10	91
11	11	90
12	12	89
13	13	88

Showing 1 to 13 of 100 entries

Figura 3.3: Visualizando o conteúdo do arquivo .csv

A função `read_delim` ainda pode ser adaptada para outros tipos de arquivos de texto delimitado, como `.txt` ou `.tab`; ou para abrir arquivos com outros delimitadores de colunas, como ponto e vírgula (`delim = ";"`) ou TAB (`delim = "\t"` , o que indica à função que as colunas são separadas por dois espaços simples). Os exemplos abaixo fazem exatamente isto.

```
banco1 <- read_delim("exemplo_ponto_virgula.csv", delim = ";")
banco2 <- read_delim("exemplo_texto.txt", delim = ",")
banco3 <- read_delim("exemplo_tabular.tab", delim = ",")
banco4 <- read_delim("exemplo_espacos.csv", delim = "\tab")
```

Além de arquivos armazenados no computador, também podemos carregar arquivos armazenados na internet: no lugar do nome do arquivo, é só passar para a função o *link* de onde o arquivo está hospedado.

```
banco5 <- read_delim("https://github.com/tidyverse/readr/raw/master/inst/extdata/mtcars.
```

Além destas extensões e do argumento `delim`, a função `read_delim` também nos permite passar outras instruções para o R carregar um arquivo. Dentre estas, a mais útil é `skip`, que serve para indicar a partir de qual linha queremos iniciar o carregamento dos dados (que pode ser usada para pular linhas que não estão formatadas corretamente).

Na pasta de materiais complementares deste capítulo, disponível *online*, temos um arquivo chamado `peessoas.csv`, que contém os nomes e as idades, salvas em duas variáveis, de algumas pessoas fictícias. Abrindo este arquivo com um editor de texto simples, veremos que o conteúdo dele está organizado de uma forma um pouco diferente do que já vimos anteriormente:

```
"Exemplo"

"NOME";"IDADE"
"Maria";45
"João";23
"Antônio";14
"Ana";9
"José";60
"Rosa";36
```

Figura 3.4: Visualizando o conteúdo do arquivo `peessoas.csv`

Dá para perceber logo de cara que existe a palavra “Exemplo” numa linha acima do restante do conteúdo do arquivo e que, além disso, esta linha possui apenas um campo – o próprio texto “Exemplo”. Vamos tentar carregar este arquivo com a função `read_delim`, que já vimos, para ver como o R lerá estes dados.

```
# Carrega o arquivo "peessoas.csv"
peessoas <- read_delim("peessoas.csv", delim = ";")
```

Podemos, agora, visualizar ele com `view`. O resultado, como dá pra perceber abaixo, não contém as duas colunas que vimos no arquivo bruto de texto – está faltando a variável `IDADE`. Mais do que isso, o arquivo também foi carregado com aquele texto “Exemplo” como título da variável, e o valor da primeira observação ficou sendo “NOME” (que é, de fato, o nome da variável).

```
View(peessoas)
```



	Exemplo
1	NOME
2	Maria
3	João
4	Antônio
5	Ana
6	José
7	Rosa

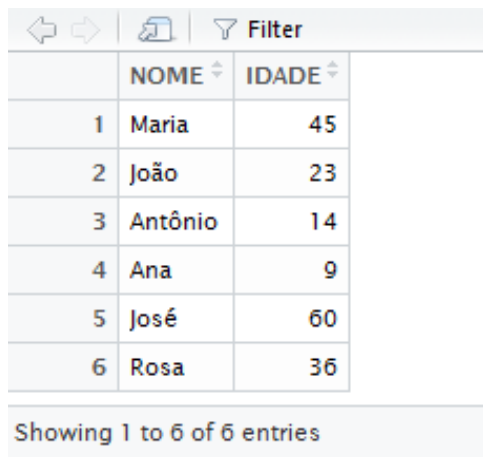
Showing 1 to 7 of 7 entries

Figura 3.5: Visualizando o banco

Para corrigir isso, precisamos usar o argumento `skip` da função `read_delim` para pedir que ela carregue os dados pulando algumas linhas (1, 2, 3, etc., linhas) – exatamente para pular aquele “Exemplo” e aquela linha em branco depois disso. Fazemos isto assim:

```
# Carrega o arquivo "pessoas.csv" pulando tres linhas
pessoas <- read_delim("pessoas.csv", delim = ";", skip = 3)
```

Se usarmos `View` na sequência, veremos que o `data.frame`, agora, está correto.



	NOME	IDADE
1	Maria	45
2	João	23
3	Antônio	14
4	Ana	9
5	José	60
6	Rosa	36

Showing 1 to 6 of 6 entries

Figura 3.6: Visualizando o banco carregado corretamente

Mas este argumento não esgota as possibilidades da função `read_delim`. Ao contrário, ela possui diversos outros argumentos que podem ser úteis para contornar erros. Abaixo, segue a descrição de alguns deles (para ver outros, digite no console `help(read_delim)`).

Tabela 3.2: Outros argumentos da função `read_delim`

Argumento	Descrição	Uso
<code>quote</code>	Delimitador de campos textuais.	<code>quote = "\""</code>
<code>col_names</code>	Passa novos nomes para as variáveis carregadas.	<code>col_names = c("Nome1", "Nome2", ...)</code>
<code>locale</code>	Muda as configurações de horário e acentuação.	Veremos adiante.

3.3 Outros formatos

Uma vez que aprendemos como carregar arquivos com extensão `.csv`, fica fácil carregar qualquer outro arquivo. O que veremos a seguir, portanto, são as funções e os pacotes que podemos usar para carregar outros tipos de arquivo. De forma complementar, nas duas últimas seções aprenderemos a lidar com os erros mais frequentes quando tentamos carregar algum arquivo e a exportar dados para arquivos dos mais diversos formatos.

3.3.1 Planilhas

Para abrir planilhas do Excel, com extensões `.xls` ou `.xlsx`, usamos a função `read_excel` do pacote `readxl`, que é semelhante à função `read_delim`. Exemplo:

```
# Carrega o pacote readxl
library(readxl)

# Carrega a planilha 'populacao_brasil.xls' na pasta do livro
dados <- read_excel("populacao_brasil.xls")
```

Novamente, a primeira coisa que passamos para a função é o nome do arquivo (ou, aqui também, o *link* de onde o arquivo está hospedado na internet) – na maioria dos casos, apenas isto é suficiente. Mas também podemos passar argumentos opcionais para a função, tais como: `sheet`, que indica o número da planilha dentro do arquivo (1 para a primeira, 2 para a segunda, e assim por diante); e `skip`, que diz quantas linhas a função deve pular para começar a ler o conteúdo do arquivo, exatamente como na função `read_delim`. No exemplo abaixo, carregamos a segunda planilha do mesmo arquivo, pedindo também para a função começar a ler os dados a partir da primeira linha.

```
# Carrega a primeira planilha do arquivo pulando a primeira linha
dados <- read_excel("populacao_brasil.xls", sheet = 1, skip = 1)
```

O único grande problema do pacote `readxl` é que ele não serve para abrir planilhas feitas pelo *Open Office* (*OpenDocument Spreadsheet*, extensão `.ods`). No entanto, existe um pacote específico para isto que nos deixa contornar esta limitação: o pacote `readODS`. Basicamente, precisamos apenas carregá-lo e usar a função `read_ods` para carregar este tipo de arquivo, como abaixo.

```
# Carrega o pacote readxl
library(readODS)

# Carrega a planilha 'populacao_brasil.ods' na pasta do livro
dados <- read.ods("populacao_brasil.ods", sheet = 1)
```

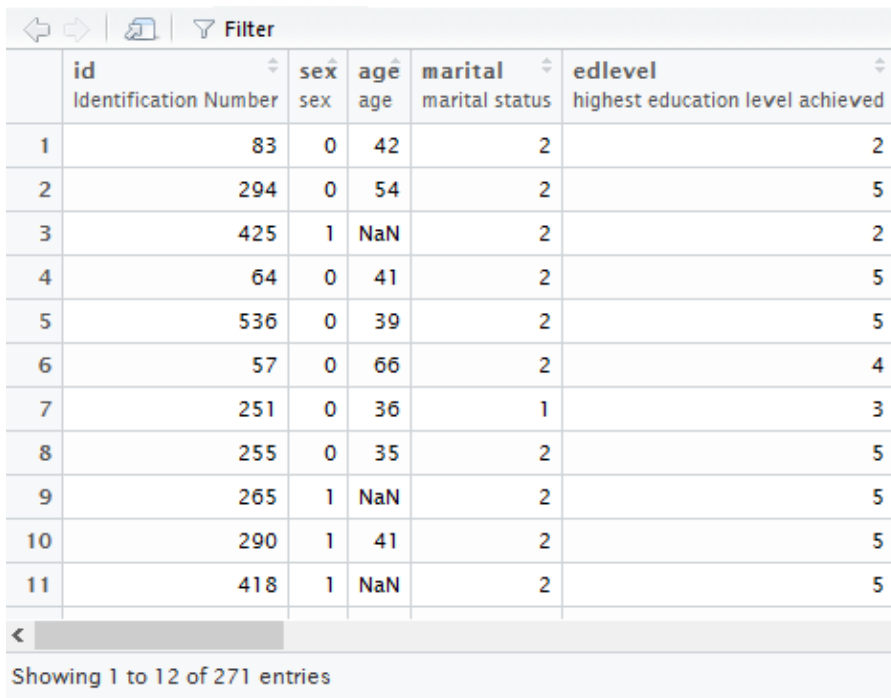
3.3.2 SPSS, Stata e SAS

Alguns *softwares* de análise de dados possuem arquivos próprios para armazenar dados. Estes são os casos do *SPSS* (arquivos `.sav` e `.por`), *Stata* (arquivos `.dta`) e *SAS* (arquivos `.sas7bdat`), todos os três muito populares na academia e no mercado. Para abri-los, recorreremos ao pacote `haven`, que usa o código de outro pacote desenvolvido em `C`, o `ReadStat`, para fazer o trabalho. Para mostrar com isto funciona, carregaremos um banco de dados de um *survey* realizado na Austrália para avaliar o impacto de privações de sono.² Novamente, o uso do pacote é auto-explicativo.

```
# Carrega o pacote haven
library(haven)

# Carrega o arquivo
dados <- read_sav("sleep.sav")
```

Nos três casos, o pacote faz o trabalho de manter as informações originais dos arquivos ao máximo possível. Em arquivos do *SPSS*, isso inclui manter os *labels* originais das variáveis e os seus tipos (a função `read_sav` converte variáveis numéricas e categóricas para seus tipos respectivos no `R`). Podemos verificar isto abrindo o objeto onde salvamos o arquivo do `nome_do_arquivo`, como na imagem abaixo – os *labels* aparecem logo abaixo do nome das variáveis. Quando possível, para arquivos do *Stata* e do *SAS* o mesmo também ocorre.³



	id Identification Number	sex sex	age age	marital marital status	edlevel highest education level achieved
1	83	0	42	2	2
2	294	0	54	2	5
3	425	1	NaN	2	2
4	64	0	41	2	5
5	536	0	39	2	5
6	57	0	66	2	4
7	251	0	36	1	3
8	255	0	35	2	5
9	265	1	NaN	2	5
10	290	1	41	2	5
11	418	1	NaN	2	5

Showing 1 to 12 of 271 entries

Figura 3.7: Labels de um banco de dados do SPSS

3.3.3 JSON

Outro formato útil, ainda que pouco utilizado na academia, é o *JSON (JavaScript Object Notation)* – que pode ser encontrado cada vez mais em sites e API's, como a de Dados Abertos do Governo Federal. Em parte, o formato é útil porque armazena o conteúdo num formato legível, de fácil compreensão; de outro, porque ele pode ser aberto facilmente por várias linguagens de programação.

Para abrir este tipo de arquivo, usaremos a função `import` do pacote `rio` (abreviação de *R Input/Output*; veremos outras utilidades dele adiante). O arquivo de exemplo, que usaremos nos próximos capítulos, vem da página da Transparência Internacional, da pesquisas *Corruption Perceptions Index 2015*.⁴

```
# Carrega o pacote rio
library(rio)

# Carrega o banco de dados do CPI 2015
cpi <- import("cpi-data.json")
```

3.3.4 R Data

Por fim, temos o formato nativo do R, o *R Data*. Obviamente, ele serve principalmente para salvar e carregar dados exportados pelo R para uso no próprio R, mas seu uso tem algumas vantagens atrativas. Em primeiro lugar, e diferentemente de formatos de texto como *.csv* e *.tab*, arquivos *.Rda* são binários – o que, traduzindo, permite que se guarde muito mais informação em menos espaço, inclusive forçando a compressão dos dados. Em segundo lugar, ler e salvar estes arquivos pelo R é geralmente mais rápido, e isto apesar da compressão. Por fim, ele salva e carrega objetos de um jeito mais intuitivo, como mostra o exemplo a seguir.

```
# Carrega os mesmos dados da CPI 2015, agora em formato .Rda
load("cpi2015.Rda")
```

Não é preciso carregar nenhum pacote, nem realizar nenhuma configuração: o objeto carregado vai direto para a memória do R, onde pode ser visto na aba *Environment* do *RStudio*.

3.3.5 Outros formatos

Embora tenhamos visto como abrir os tipos de arquivos mais comuns – delimitados por texto, planilhas e de outros *softwares* – existe uma infinidade de formas de armazenar dados em arquivos e, muitas vezes, precisamos recorrer a alguma ferramenta que nos permita ler estes dados no R. Neste ponto, recomendamos o pacote *rio*, visto acima, para este tipo de tarefa.

Resumidamente, o *rio* funciona como uma espécie de canivete suíço para a importação e exportação de dados: basta usar a função `import` e passar para ela o nome do arquivo que queremos abrir; a partir disto, a função identifica o formato do arquivo que estamos tentando abrir e chama internamente a função e especificações mais adequadas para tanto – sem precisar passar nada além disso para a função. O lado ruim do pacote, por outro lado, é que ele naturalmente não dá conta de abrir qualquer arquivo, apenas alguns entre vários tipos. Entre outros, os arquivos suportados incluem: *.csv*, *.tsv*, *.fst*, *.psv*, *.fwf*, *.Rda*, *.Rds*, *.json*, *.dta*, *.sav*, *.xls*, *.mpt*, *.dif*, entre outros⁵. Exemplo de funcionamento da função `import`:

```
# Carrega o pacote rio  
library(rio)  
  
# Importa alguns dados  
dados <- import("exemplo.csv")  
dados2 <- import("sleep.sav")
```

3.4 Lidando com erros

Aprender a manusear as funções mais adequadas para importar cada tipo de arquivo cobre boa parte do que precisamos para trabalhar com nossos dados no R, mas não tudo. Em particular, com frequência usamos a ferramenta adequada e, mesmo assim, obtemos algum erro: o arquivo não abre, o R trava, ou ainda os dados vêm desconfigurados. Este tipo de coisa raramente é coberto em materiais didáticos, apesar de ser importante termos algumas noções básicas de como identificar – e de como contornar – estes erros.

3.4.1 Especificação do delimitador

Em arquivos delimitados de texto, talvez o erro que mais cometemos é especificar de forma errada o delimitador: passar uma vírgula quando ele é, na verdade, ponto e vírgula; ou passar ponto e vírgula quando ele é TAB. Aqui, o truque é quase banal: tentar abrir o arquivo com um editor de texto simples para olhar os dados: na maioria das vezes, isto já nos permite localizar o identificador. O problema desta solução é que isto pode não dar certo se o arquivo for muito grande (e o editor de texto não conseguir abri-lo).

Outra solução é ir na tentativa e erro. Por exemplo:

```
# Se isto nao der certo...  
banco <- read_delim("exemplo_ponto_virgula.csv", delim = ",")  
  
# Tentamos isto...  
banco <- read_delim("exemplo_ponto_virgula.csv", delim = "\tab")  
  
# E se tambem nao der, tentamos isto  
banco <- read_delim("exemplo_ponto_virgula.csv", delim = ";")
```

3.4.2 Células vazias

Alguns arquivos às vezes vêm com células vazias, isto é, com informações não preenchidas (como *missings*), e isto pode ocasionar erros ou confusões. Em geral, isto ocorre mais em arquivos de texto delimitados, mas as funções que mostramos aqui para abri-los (`read_delim` , principalmente) nos dão notificações sobre estes erros. Os dados são carregados normalmente, mas ficamos sabendo onde procurar lacunas na base.

3.4.3 Problemas de acentuação

Outro problema comum para quem trabalha com bancos de dados que contêm informações textuais (nomes, endereços, etc.) é a acentuação. Volta e meia importamos um arquivo com ã ou â que são exibidos como ¢ e ˆ no lugar.

Explicar por que isto acontece foge muito do escopo deste livro, mas é útil entender que cada sistema possui um conjunto de caracteres válidos para se escrever texto: em português, temos acentos; em inglês, não. Assim, quando informações escritas usando um conjunto de caracteres particular, que chamamos de *encoding*, é trasposto para outro conjunto, coisas como estas ocorrem. E trocar de sistema operacional, abrir arquivos criados por um *software* em outro, entre outros, são situações onde isto pode acontecer.

Em português, usamos principalmente os *encodings* UTF-8 e latin1 (mas existem outros, alguns mais específicos) e, portanto, nossa primeira tentativa de corrigir estes erros é passando estes *encodings* para as funções que usamos para carregar dados que possam conter acentos usados em português. No caso da função `read_delim` , isto seria feito da seguinte forma:

```
# Caso o arquivo 'exemplo.csv' tivesse erro de encoding, tentariamos...
dados <- read_delim("exemplo.csv", delim = ",", locale = locale(encoding = "UTF-8"))

# Ou tentariamos...
dados <- read_delim("exemplo.csv", delim = ",", locale = locale(encoding = "latin1"))
```

Às vezes, isto não resolve: o *encoding* do arquivo não é nenhum dos dois. Para a nossa sorte, o pacote `readr` possui uma função chamada `guess_encoding` que tenta descobrir o *encoding* de um arquivo. Caso UTF-8 e latin1 não sirvam, portanto, tente o seguinte:

```
library(readr)
guess_encoding("exemplo.csv")
```

```
## # A tibble: 1 x 2
##   encoding confidence
##   <chr>          <dbl>
## 1 ASCII          1
```

E aqui vemos que o *encoding* do arquivo `exemplo.csv`, que já carregamos antes, é provavelmente `ASCII` (um tipo de *encoding* com suporte para inglês, sem acentos).

3.4.4 Bancos muito grandes

O `R` possui uma maior limitação que, como já dito, é a de não suportar carregar dados muito pesados. Na verdade, a razão disto é que o `R` armazena as informações na memória RAM do computador, e não no disco rígido. Deste modo, a placa de memória (se com 8gb, ou 16gb, etc.) é quem dita o tamanho dos arquivos que podemos carregar. Mas, normalmente, isto é suficiente: 1gb de dados é muita, muita informações.

Caso não seja, precisamos apelar para outras soluções, entre inúmeras⁶ disponíveis. Uma das mais simples é por meio do pacote `sparklyr`, que fornece uma interface para o [Apache Spark](#), um *framework* de processamento de dados massivos. Novamente, foge um pouco do escopo do livro explicar em detalhes o que é este recurso e como ele resolve nosso problema. Em resumo, o que ele faz é o seguinte: em vez de carregar dados para o `R`, ele passa as informações uma aplicação em Java e, feito isso, opera nos dados quebrando tarefas em pedaços; como resultado, conseguimos trabalhar com dados que não cabem na memória do computador (ele trabalha com cada parte de cada vez) e de forma bastante rápida.

O [website](#)⁷ do pacote oferece um guia de como usar o `sparklyr`, o que, basicamente, consiste no seguinte.

```

# Instala o pacote sparklyr
install.packages("sparklyr")

# Baixa e instala uma versao do Spark (o pacote lida com os detalhes da instalacao)
spark_install(version = "1.6.2")

# Cria uma conexao ao Spark
sc <- spark_connect(master = "local")

# Copia dados da memoria do R para o Spark
capitais_spark <- copy_to(sc, capitais)

```

Com isto, o banco já pode ser manipulado com o pacote `dplyr`, que veremos no capítulo 4. Feitas as operações de manipulação, coletamos os resultados com a função `collect` do `sparklyr`. Para os nossos propósitos, carregar bases grandes diretamente de um arquivo pode ser feito com a função `spark_read_csv`.

```

# Carrega um arquivo ficticio (apenas um exemplo)
arquivo_spark <- spark_read_csv(sc, "nome_da_base", "base_muito_grande.csv")

```

3.4.5 Erros humanos

Neste final, vamos falar de erros humanos: digitar errado o nome de um arquivo, passar o local errado de onde o arquivo está, usar uma função que abre um tipo de arquivo para tentar abrir arquivos de outro formato. Mesmo parecendo algo trivial, tanto pessoas aprendendo R quanto usuários(as) avançados(as) cometem este tipo de erro toda hora. Nosso alerta final, portanto, é: certifique-se de ter usado a função correta, de não ter digitado nada errado e de garantir de que o endereço do arquivo (ou o diretório corrente do R) está certo. Caso contrário, erros como estes ocorrerão:

```

# Nome do arquivo errado
banco <- read_delim("exemplo_texto.txt", delim = ",")

## Error: 'exemplo_texto.txt' does not exist in current working directory ('/Users/denis

```



```
# Funcao errada
```

```
banco <- read_csv("exemplo_tabular.tab", delim = ",")
```

```
## Error in read_csv("exemplo_tabular.tab", delim = ","): unused argument (delim = ",")
```

3.5 Exportando dados

Importar dados para o R é relativamente fácil, como vimos, mas exportá-los é ainda mais. Tendo já alguns dados armazenados na memória do R, usamos funções semelhantes as de carregamento para exportá-los. Dentre estas, as principais são:

Tabela 3.3: Funções de exportação de dados

Arquivo	Extensão	Pacote	Função
Texto delimitado	.txt	readr	write_delim
Texto delimitado	.csv	readr	write_delim
SPSS	.sav	haven	write_sav
Stata	.dta	haven	write_dta
SAS	.sas7bdat	haven	write_sas
Outros	-	rio	export
Outros	-	rio	convert

Para exportar um `data.frame` qualquer, o procedimento básico é mais ou menos o seguinte: o primeiro argumento que passamos para a função é o nome do objeto seguido do nome do arquivo que queremos criar entre aspas (não podemos esquecer de incluir a extensão do arquivo, que, no exemplo a seguir, é `.txt`).

```
# Carrega o pacote readr
library(readr)

# Cria um data.frame com duas variaveis
banco <- data.frame(x = 1:10, y = 1:10)

# Exporta ele para um arquivo .txt
write_delim(banco, "banco.txt")
```

As demais funções são semelhantes.

```
# Outros pacotes
library(haven)
library(rio)

# Exporta para .sav
write_sav(banco, "banco.sav")

# Exporta para .dta
write_dta(banco, "banco.dta")

# Exporta para .json (e' preciso declarar 'file =')
export(banco, "banco.json")
```

Ainda usando o pacote `rio`, também podemos converter diretamente um arquivo de um formato para outro, o que nos poupa o trabalho de, primeiro, ler o arquivo para, então, exportá-lo. Com uma única função fazemos tudo isto simultaneamente. Como exemplo, podemos converter o arquivo `exemplo.csv`, que está na pasta de materiais complementares deste livro, para `.sav`, formato do SPSS.

```
# Converte o arquivo 'exemplo.csv' para .sav
convert("exemplo.csv", "exemplo.sav")
```

Para esta função, tudo o que precisamos fazer é passar o nome, ou o endereço com o nome, do arquivo que queremos converter e, como segundo argumento, o nome do arquivo que queremos criar – com a nova extensão. A depender do tamanho do arquivo, em poucos

segundos a conversão é concluída. Mais tipos de conversão que a função `convert` executa podem ser vistos digitando `?convert` no console.

1. As ferramentas que usaremos aqui não são nem de longe as únicas, nem necessariamente as melhores, para carregar dados – na verdade, o próprio `R` já vem com algumas funções nativas para importação de dados. Nossa escolha aqui reflete mais nossa experiência trabalhando com o `R` : as funções que usamos são simples, nos permitem contornar diversos erros e, em geral, são rápidas. Para outras opções para carregar dados, ver [LINK](#).↵
2. Detalhes do *survey* podem ser vistos em: <http://spss.allenandunwin.com.s3-website-ap-southeast-2.amazonaws.com/data-files.html>.↵
3. As funções `read_dta` e `read_sas` possuem alguns argumentos adicionais, que podem ser úteis para corrigir acentuação e especificar outros detalhes.↵
4. A página da pesquisa, bem como os dados e outros recursos, estão disponíveis em: <https://www.transparency.org/cpi2015/>.↵
5. Para ver a lista completa de arquivos suportados ver <https://github.com/leeper/rio>↵
6. Para alguns exemplos, ver https://rstudio-pubs-static.s3.amazonaws.com/72295_692737b667614d369bd87cb0f51c9a4b.html.↵
7. Disponível em: <http://spark.rstudio.com/>.↵