Introdução ao R

Iara Passos

Novembro de 2019

Contents

Chapter 1

Introdução

A ideia dessa apostila é que seja utilizada como complemento das aulas, junto com as listas de exercícios disponibilizadas.

1.1 O que é a linguagem R?

R é uma linguagem de programação e um ambiente para implementação de análises estatísticas e gráficos. Um Ambiente de Desenvolvimento Integrado[1] (do inglês, *Integrated Development Environment*, IDE) é um *software* que contêm ferramentas para auxiliar o desenvolvimento de *softwares*, de modo a otimizar e facilitar esse processo. As principais funções de uma IDE são: edição, compilação, depuração e modelagem.

Foi criado em 1993, por Ross Ihaka e por Robert Gentleman do Departamento de Estatística da Universidade de Auckland na Nova Zelândia, baseado na linguagem de programação e ambiente S, desenvolvido por John Chambers na Bell Laboratories (hoje Lucent Technologies). A partir de 1997 passou a ser desenvolvido por um grupo de colaboradores do mundo todo[2].

1.1.1 Comunidade

Uma das vantagens da linguagem R é que, por ser Software Livre, tem uma comunidade de usuários muito grande, que contribui melhorando o código e criando documentação e tutoriais para outros usuários. Além do site do R-Project, o R-bloggers (em inglês) e o bRbloggers reúnem diversos sites/blogs sobre a linguagem R, com tutoriais e informações atualizadas sobre pacotes

[1] https://pt.wikipedia.org/wiki/Ambiente_de_desenvolvimento_integrado
[2] Contributors: https://www.r-project.org/contributors.html

e códigos. O Stack Overflow é uma plataforma que possibilita os usuarios a fazerem perguntas relacionadas a qualquer linguagem de programaçao e obter respostas de outros usuarios. Procurar a sua dúvida lá pode ser bem útil para conseguir avançar no seu código. Para uma melhor aprendizagem, é essencial pesquisar documentação, tutoriais e, principalmente, o que tem sido discutido na comunidade de usuários do R ao redor do mundo. Para começar, é bom acompanhar os três sites citados.

1.2 Instalando o R

A linguagem R é disponibilizada como um Software Livre [3], sem custos e seu código-fonte é aberto e por ser acessado por qualquer pessoa, sobre os termos da Free Software Foundation's GNU General Public License. Para que o seu computador seja capaz de interpretar a linguagem R é necessário que você faça download e instale a linguagem no seu computador, seguindo os passos:

- Acesse o link: https://cran.r-project.org/mirrors.html e escolha um *mirror* para realizar o *download*, dê preferência um no Brasil.
- Na página que abriu escolha o *download* para o seu sistema operacional (Windows, MacOS ou Linux).
- Apos o download abra o arquivo e realize a instalaçao da forma que o seu sistema operacional exige.

1.3 Instalando o RStudio

Só a instalação do R é o suficiente para começar a usá-lo, mas há algumas IDEs que melhoram a interface para o usuário, deixando-a mais prática e fácil de utilizar os recursos e ferramentas disponíveis. Uma das mais utilizadas é o RStudio, que também é um software livre. O download[4]. Para realizar o download para qualquer sistema operacional acesse o link: https://www.rstudio.com/products/rstudio/download/#download. Escolha o sistema operacional correspondente ao seu computador, faça o download do isntalador e siga a instalação.

[3] Licença R Project: https://www.r-project.org/COPYING

[4] Há também uma versão que pode ser utilizada pelo navegador do seu computador e acessada via servidor

Chapter 2

Primeiros Passos

2.1 Janela de comando do R

Abra o R no seu computador. Essa janela é a nossa comunicação com o R. É nesse local que passamos as informações para que sejam interpretadas pela linguagem. A aparência desse ambiente é de um *prompt* de comando, em que comandos são digitados pelo usuário e ao apertar a tecla `enter` esses comandos são enviados para o R e processados. Ao finalizar o processamento do comando, se o comando solicitar algum retorno, o R pode: a) retornar a saída do comando ou b) retornar uma mensagem de erro (se há algum erro na sintaxe do comando).

> Isso é comum em todas as linguagens de programação. O que realmente acontece é que enviamos os comandos em uma linguagem de alto nível (mais próxima do usuário) e o programa traduz esses comandos para uma linguagem de baixo nível na qual a máquina consegue interpretar. O R, por ser uma linguagem interpretativa, faz esse processo diretamente sem que seja necessária a utilização de compiladores mediando essa tradução. Isso facilita a comunicação com o usuário

Digite qualquer número na linha de comando e aperte `enter`. Como a seguir:

```
1456
```

```
## [1] 1456
```

O R processou essa informação e retornou um resultado. Como apenas inserimos um número, sem nenhum comando ou cálculo, ele retornou apenas o próprio número.

> O que é importante entendermos aqui é que essa tela de comando é dinâmica, de modo que cada linha é uma entrada e corresponde

a uma comunicação com o computador. Dessa forma, não podemos voltar e alterar a linha anterior pois aquele comando já foi enviado e processado.

Se você está em uma linha nova você pode utilizar as setas para cima e para baixo do seu teclado para navegar entre os comandos já utilizados. E, caso queira, apertar `enter` para enviar o comando novamente.

2.2 O R como uma calculadora

O R, como outras linguagens de programação, pode funcionar como uma calculadora interativa. Assim, podemos fazer contas no console. Por exemplo, para fazer uma soma:

```
9 + 2
```

```
## [1] 11
```

Agora ele nos retornou o resultado da soma que solicitamos. Teste outras contas básicas utilizando os seguintes símbolos:

- "+" adição
- "-" subtração
- "*" multiplicação
- "/" divisão
- "^" potência

Se você digitar um comando incompleto, como 5 +, e apertar `enter`, o R mostrará um +, o que não tem nada a ver com somar alguma coisa. Isso significa que o R está esperando que você complete o seu comando. Termine o seu comando ou aperte `esc` para recomeçar.

O R ainda tem outros símbolos relacionados a contas matemáticas. A seguinte sequência de caracteres `%%` irá retornar o resto da divisão entre dois números. Por exemplo:

```
5 %% 3
```

```
## [1] 2
```

Como a divisão de 5 por 3 não é exata, ao solicitarmos o resto da divisão com símbolo `%%` o R retorna 2. Teste uma divisão exata. O que ele retorna?

Por fim, a sequência `%%` retorna a parte inteira da divisão de X por Y. Tente novamente na divisão de 5 por 3. O que aconteceu? Qual a diferença?

Exercício 2.1. Faça cálculos mais complexos, com várias operações combinadas.

Importante: Tente entender como o R faz essas operações. **Atenção nos parênteses!!**

> **ERROS** Se você digitar um comando que o R não reconhece, ele retornará uma mensagem de erro. **NÃO ENTRE EM PÂNICO!** Ele só está avisando que não conseguiu interpretar o comando. Você pode digitar outro comando normalmente em seguida ou corrigir o anterior.

Aprender esse tipo de funcionalidade no R, apesar de bem básico, é bastante útil para começarmos a aprender as primeiras interações com a linguagem. Porém, obviamente, não é para isso que queremos aprender alguma linguagem de programação, mas sim para automatizar processos, evitar repetições desnecessárias e realizar análises mais avançadas do que operações de soma e multiplicação.

2.3 Objetos e classes de objetos

Já enviamos informações de números para o R e fizemos operações básicas com ele. Mas e se agora nós enviarmos uma letra ou palavra para ele? Tente fazer isso.

```
a
```

```
## Error in eval(expr, envir, enclos): object 'a' not found
```

Ele retorna o erro. Preste bem atenção no que está escrito no erro. No caso, ele está nos informando que o objeto `a` não foi encontrado. Para entendermos isso, precisamos entender o que é um **objeto**.

Um **objeto**[1], em termos da ciência da computação, é um valor armazenado na memória do computador. Ele pode assumir a forma de variáveis, funções[2] ou estruturas de dados[3]. O objeto que importa para nós, por ora, são aqueles que representam variáveis. Nas próximas etapas do curso, trabalharemos com funções mas não iremos considerá-las como objetos.

> Basicamente, no R, funções são ações realizadas em objetos. Porém, como tudo o que existe no R é um objeto, funções são objetos também.

Então, se um objeto armazena valores, conseguiremos fazer processos mais avançados e evitar repetições nos nossos códigos daqui pra frente. O objeto mais simples que encontramos no R é aquele que armazena apenas uma informação na memória.

[1] Para mais informações sobre objetos: https://pt.stackoverflow.com/questions/205482/em-programa%C3%A7%C3%A3o-o-que-%C3%A9-um-objeto

[2] https://pt.wikipedia.org/wiki/M%C3%A9todo_(programa%C3%A7%C3%A3o)

[3] https://pt.wikipedia.org/wiki/Estrutura_de_dados

A primeira coisa que precisamos definir é um nome para esse objeto (ou variável) e para isso precisamos seguir algumas preceitos - obrigatórios ou aconselhados.

- **Nomes válidos** para variáveis (obrigatório)[4]

  1. **TEM QUE COMEÇAR COM UMA LETRA OU UM PONTO (.)**
     - exemplo `nome`, `genero23`, `.name` são nomes válidos
  2. **NÃO PODE COMEÇAR COM NÚMEROS**
     - exemplo `1x` `0meunome` são nomes inválidos
  3. **SE O PONTO (.) FOR O PRIMEIRO DÍGITO, ELE NÃO PODE SER SEGUIDO DE UM NÚMERO**
     - exemplo: `.23bananas`, `.45ois` são nomes inválidos
  4. **PODE CONTER UNDERSCORE _, DESDE QUE NÃO SEJA O PRIMEIRO DíGITO**
     - exemplo: `meu_nome`, `hello_world` são nomes válidos
  5. **NÃO PODE CONTER ESPAÇO EM BRANCO**
     - exemplo: `meu nome`, `idade jovens` são nomes inválidos
  6. **NÃO PODE SER UMA DAS PALAVRAS RESERVADAS DA LINGUAGEM**
     - Palavras reservadas[5] são: `if else repeat while function for in next break TRUE FALSE NULL Inf NaN NA NA_integer_ NA_real_ NA_complex_ NA_character_` são nomes inválidos

- **Boas práticas** para criação de variáveis

  1. **CAIXA BAIXA** R é `case sensitive`, de modo que ele interpreta como variáveis diferentes `Meu_nome`, `MEU_NOME`, `meu_nome`, `MeU_NoMe` e qualquer outra variação possível. Portanto é recomendado que sejam criadas apenas variáveis em caixa baixa, para evitar confusão.
  2. **VARIÁVEIS COM PONTOS** Ainda que seja permitido pela linguagem, é recomendável evitar variáveis como `meu.nome` pois diversos pacotes utilizam `.` no nome das funções. Para fazer com que seu código seja mais fácil de ser lido por você e por outras pessoas, é bom evitar esse tipo de sintaxe no nome das variáveis.
  3. **VARIÁVEIS COM NOMES DE FUNÇÕES** Também para evitar confusões, evite utilizar nomes de funções do R como nomes de variáveis, como `mean`, `sum`, etc.

  Iremos indicar outras **boas práticas** mais adiante no curso.

Bom, agora que já sabemos como podemos nomear nossos objetos, vamos criar nosso primeiro objeto. Para criar um objeto precisamos armazenar algo dentro

[4] https://cran.r-project.org/doc/FAQ/R-FAQ.html#What-are-valid-names_003f
[5] https://stat.ethz.ch/R-manual/R-devel/library/base/html/Reserved.html

dele. Para que o R entenda que queremos armazenar algo em uma variável, utilizamos o símbolo `<-`.

> O R aceita o `=` como atribuição de variável, mas não é recomendado, por poder ser confundido com operadores lógicos e argumentos de funções.

O R interpreta como o nome da variável o lado para o qual a seta está apontando e como valor atribuído o que está no lado oposto. Assim, atribuímos valores a variável x de qualquer um dos modos: `x <- 5` ou `5 -> x`. Porém, para evitar confusão, usaremos aqui sempre o primeiro formato

Sendo assim, criemos x:

```
x <- 5
```

Como só realizamos a atribuição da variável o R não retornou nada no *prompt* de comando. Para que ele retorne o que está armazenado dentro de uma variável precisamos pedir para que ele faça isso. Por sorte, a sintaxe do R possibilita que façamos isso de forma simples: apenas digitando o nome da variável e apertando `enter`.

```
x
```

```
## [1] 5
```

Pronto! Agora pedimos para o R nos mostrar o que é x ou, equivalentemente, o que está armazenado na variável x. Dessa forma, podemos fazer cálculos com x. Por exemplo:

```
x * 5
```

```
## [1] 25
```

E vejam só, podemos armazenar esse resultado em uma nova variável.

```
y <- x * 5
```

Agora temos duas variáveis: uma `x`, que tem armazenada o número 5, e `y`, que tem armazenada o resultado da operação `x` * 5.

Podemos ir além e criar uma variável `z` que armazena a divisão de `x` por `y`. Qual valor estará armazenado em `z`?

Certo! Mas e se agora digitarmos `x <- 10` quanto vai valer x? O que aconteceu?

> ATENÇÃO! Chegamos em uma parte muito importante do curso. Quando re-atribuímos uma variável já existente nós apagamos o valor que estava dentro dela e escrevemos um valor novo.

Ok! Já sabemos como criar, substituir e fazer contas com variáveis. Só que até agora só trabalhamos com números, mas variáveis podem - e devem - armazenar outros tipos de dados.

Neste caso precisamos de objetos diferentes, ou seja, de classes e formas de armazenamentos diferentes. O R apresenta **quatro** principais classes de objetos: `integer`, `numeric`, `character` e `logical`. Vamos ver cada uma delas:

- `integer` Um objeto de classe `integer` é uma variável numérica que armazena apenas números inteiros (sem pontos flutuantes - os decimais). A diferença para a classe numeric é que a classe integer armazenam números com menos casas e não consegue fazer cálculos com casas decimais. Porem, quando declaramos uma variável numérica o R automaticamente a armazena como tipo `numeric`. É possível forçar a alteração dessa variável para `integer` se a sua variável não for ser utilizada para fazer cálculos e for necessário ocupar menos espaço de armazenamento (por exemplo - ID number). Abreviação: **int**

- `numeric` Classe padrão para objetos numéricos. Armazena núumeros inteiros e do tipo double (ou float), ou seja, com casas decimais. Necessários para realização de cálculos matemáticos. Abreviação: **num**

- `character` Classe do tipo string que armazena dados textuais. Mesmo que haja uma sequência de números os dados serão lidos como textos. Não é possível fazer operações matemáticas e aplicar funções de variáveis numéricas. Abreviação: **char**

- `logical` Classe do tipo lógica (booleana), permite apenas armazenamentos de dados lógicos (TRUE ou FALSE). Utilizada para fazer operações lógicas com os dados. Também pode ser utilizada como variável binária. Abreviação: **logi**

  Ainda há uma classe de variável para números complexos, utilizada para análises matemáticas mais avançadas mas não utilizaremos nesse curso.

Agora que já sabemos as classes de objetos. Vamos criar objetos para cada classe. Para criar objetos `integer` basta criar uma variável com números, assim como a variável da classe `numeric`. Porém, automaticamente o R cria variáveis numéricas como da classe `numeric`. Entao, para que forcermos o R a armazenar um número como `integer`, é necessario colocar no final do número a letra L. Assim, `inteiro <- 8L` será do tipo `integer` e não `numeric`. O L só entra para avisar o R que queremos que aquela variável seja de outra classe. Se pedirmos para ele nos retornar a variável armazenada ele irá retornar apenas a parte numérica da variável (sem o `L`).

```
inteiro <- 8L
inteiro
```

```
## [1] 8
```

Para criar variáveis de classe `character` também precisamos avisar para o R que aquela variável não é do tipo numérico. Sendo assim, não podemos criar a variável da seguinte forma:

```
string <- texto
```

```
## Error in eval(expr, envir, enclos): object 'texto' not found
```

Para que o R identifique que a variável é da classe `character` precisamos definir o valor entre aspas (" "):

```
string <- "texto"
```

Agora sim, o R interpreta o que está dentro dos parênteses como uma sequência de caracteres. Qualquer sequência de caracteres dentro das aspas será interpretada como um objeto da classe `character`, inclusive números, espaços e caracteres especiais.

Por fim, para criar uma variável do tipo lógica (que so aceitam `TRUE` e `FALSE`) é necessário atribuir valores `TRUE`, `T`, `FALSE` OU `F`. Atenção: não podem estar entre aspas, se não o R interpreta como `character`.

```
log <- FALSE
```

Uma outra forma de utilizar os objetos booleanos é comparando variáveis a partir de `operadores relacionais`. Podemos perguntar para o R se uma variável é **igual** a outra (`==`), **diferente** (`!=`), **maior** ou **menor** (`>` ou `<`) ou **maior/menor ou igual** (`<=` ou `>=`). Como é uma pergunta utilizando sinais de comparação a resposta vai sempre ser do tipo `logical`, ou seja, `TRUE` e `FALSE`.

Assim, podemos perguntar para o R se x é maior que y:

```
x > y
```

```
## [1] FALSE
```

> O `!` pode ser combinado com qualquer outro operador: `>!` (não é maior)

O R nos retorna a resposta `FALSE`, pois x não é maior que y. (`x` já havia sido definido anteriormente como `5` e `y` como `25`).

Exercício 2.2. Utilize os outros operadores relacionais para comparar outras variáveis.

Há ainda os operadores lógicos, que baseiam-se na Teoria dos Conjuntos. São eles: `&&` (**AND/E**), `||` (**OR/OU**). Utilizando esses operadores podemos combinar perguntas para o R. A seguir, pergunto para o R se x é maior que y **E** se y é maior que 10. De modo que o R irá retornar `FALSE` se pelo menos uma das condições é falsa e ira retornar `TRUE` se, e somente se, as duas forem verdadeiras.

```
x > y || y > 10
```

```
## [1] TRUE
```

Exercício 2.3. Utilize o operador lógico "OU". Em quais condições o R retorna `FALSE` e em quais ele retorna `TRUE`?

Por fim, se definirmos uma variável e queremos saber como o R a armazenou podemos utilizar a função `class()` para descobrir a classe do objeto.

> Atenção! Essa é a primeira vez que utilizamos uma função no R. Funções tem uma sintaxe bem definida e argumentos obrigatórios e opcionais.

A função `class()` necessita apenas que seja inserido dentro dos parênteses o objeto que queremos saber a classe. Assim,

```
class(x)
```

```
## [1] "numeric"
```

Perguntamos para o R qual era a classe de x, no qual ele respondeu que x é da classe `numeric`. Faça isso com objetos de outras classes.

Ainda que possamos fazer várias coisas interessantes com o esse tipo mais simples de objeto, trabalhar com objetos mais complexos nos abre mais possibilidades para análises de dados. Porém, antes de avançarmos nos tipos de objetos, vamos ver algumas outras coisinhas que podem nos ajudar no aprendizado aqui para frente.

2.4 Ambiente do RStudio

Até agora trabalhamos apenas com o console do R. A vantagem dele é que ele é mais dinâmico e mais rápido, mas temos dificuldades de, por exemplo, ver quais variáveis já temos atribuídas e armazenadas na memória. De fato, se entrarmos com a função `ls()` o R irá retornar todas as variáveis já atribuídas e armazenadas. Porém, isso dificulta um pouco o trabalho da análise de dados pois temos que ficar solicitando a função toda vez que precisamos saber as variáveis armazenadas.

Para ajudar a comunicação com o usuário foram criadas IDEs (do inglês, *Integrated Development Environment*) para que facilitem algumas visualizações. Após a implementação e disponibilização dessas ferramentas o uso de várias linguagens de programação, inclusive o R, aumentou consideravelmente. Uma delas é o RStudio, software da empresa do mesmo nome que tem desempenhado um papel muito importante dentro da comunidade, criando documentação e ferramentas de aprendizagem para aumentar a utilização do R.

O ambiente do RStudio dispõe de 4 janelas principais. A janela `console` é a mesma que estávamos trabalhando antes. Ao clicarmos em `File > New File > R Script` criamos um arquivo de `script`. Esse arquivo funciona como um bloco de notas, o R só irá lê-lo se selecionarmos as linhas com os comandos e

apertarmos `ctrl+enter`. é importante salvar os arquivos em seu computador com a extensão `.R`.

Na janela `Environment` são mostradas as variáveis que já estão atribuídas na memória. Por fim, a última janela (canto inferior direito) apresenta na aba `Files` os arquivos que estão na pasta raiz do projeto, na aba `Plots` os gráficos quando solicitamos que o R imprima gráficos na tela, na aba `Packages` os pacotes instalados no computador e na aba `Help` podemos ver a documentação das funções e pacotes (esse precisa de conexão na internet).

2.5 Pacotes

Quando instalamos o R no computador, apesar de vir com várias funções básicas, essa instalação inicial não comporta todas as possibilidades de funcionalidade do R. Para tanto, existem os pacotes, que são bibliotecas de funções e até mesmo dados que possibilitam complementar ou otimizar tarefas. Há uma infinidade de pacotes criados pela comunidade de usuários (você pode criar um pacote se quiser) que são disponibilizados no site CRAN. Para instalar um pacote você precisa utilizar o comando `install.packages("nome_pacote")`. Mesmo após a instalação é necessário que toda vez que você vai utilizar o pacote utilize o comando `library(nome_pacote)`. As IDEs possibilitam que a busca e a instalação desses pacotes tambem seja mais prática.

2.5.1 Citando pacotes

Toda vez que você utilizar algum dos pacotes do R em seu trabalho, relatório ou artigo cite o pacote utilizado. Para ver como citar cada pacote utilize a função `citation(pacote)`:

```
citation("data.table")
```

```
##
## To cite package 'data.table' in publications use:
##
##  Matt Dowle and Arun Srinivasan (2019). data.table: Extension of
##   `data.frame`. http://r-datatable.com,
##   https://Rdatatable.gitlab.io/data.table,
##   https://github.com/Rdatatable/data.table.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {data.table: Extension of `data.frame`},
##     author = {Matt Dowle and Arun Srinivasan},
```

```
##     year = {2019},
##   note = {http://r-datatable.com, https://Rdatatable.gitlab.io/data.table,
## https://github.com/Rdatatable/data.table},
##   }
```

2.6 Como conseguir ajuda

Para buscar ajuda você pode ir na aba `Help` do RStudio e procurar o nome da função ou pacote na documentação disponível do R ou ir diretamente no console e digitar `?funçao` (onde função é o nome da função desejada).

Faça buscas na internet e em sites especializados (como o Stackoverflow) para encontrar documentação referente as funções e funcionalidades do R.

Chapter 3

Trabalhando com arrays

Bom, depois dessa pequena pausa vamos continuar conhecendo outros tipos de objetos no R. Até agora já trabalhamos com objetos bem simples que armazenam apenas uma informação por vez mas, na maioria das vezes, precisamos conseguir armazenar em uma mesma variável muitas informações ao mesmo tempo. Esses objetos maiores s~ao conhecidos como `arrays`.

3.1 Vetores

Um tipo de objeto um pouco mais complexo é o do tipo `vetor`. A sintaxe para a criação de um vetor é `c()`, onde os itens internos do vetor são separados por virgula. Assim, se criamos um vetor `c(2, 3, 4, 5)`, criamos um vetor de tamanho 4 que armazena os itens 2, 3, 4 e 5. O vetor seria o equivalente a uma linha de uma tabela.

> A partir de agora passaremos a tratar de tamanhos de objetos. O vetor nada mais é que uma matriz de tamanho 1 x c, ou seja, 1 linha e `c` colunas.

A principal característica do vetor é que ele só armazena objetos do mesmo tipo, ou seja, só podemos ter vetores só de `interger` ou só de `numeric` ou só de `character` ou só de `logical`. Dessa forma, o vetor irá herdar a classe dos objetos que ele contêm. Sendo assim, para termos vetores da classe `numeric` precisamos criar um vetor com objetos da classe `numeric` dentro dele.

Exercício 3.1. Crie um vetor de classe `numeric`, `logical` e `character` e atribua cada um deles a uma variável diferente.

Exercício 3.2. Pergunte ao R a classe de cada um dos vetores.

Bom, como o vetor herda as características da classe então podemos fazer operações matemáticas com os vetores numéricos.

Exercício 3.3. Multiplique o seu vetor `numeric` por um número.

O que aconteceu? Ele mudou o seu vetor original?

Exercício 3.4. Agora crie mais um vetor `numeric` (do mesmo tamanho) e multiple os dois vetores.

O que o R fez?

E se tivéssemos vetores de tamanho diferente?

Exercício 3.5. Multiplique dois vetores `numeric` de tamanhos diferentes.

O que aconteceu?

> O R, diferente de outras linguagens, quando solicitado que faça operações com vetores de tamanhos diferentes ele faz uma reciclagem: alinha os dois vetores e, caso não possuam o mesmo tamanho, vai repetindo o vetor menor até completar o vetor maior. Outras linguagens não permitiriam a operação e retornariam um erro.

A letra `c` na sintaxe da criação de um vetor vem da palavra `combine` pois o vetor nada mais é do que a combinação de vários objetos na sequência.

3.1.1 Nomeação de vetores

Podemos vincular nomes aos vetores com a função `names()`, da seguinte forma:

```
sacola1 <- c(10, 5, 8, 7)
names(sacola1) <- c("Laranja", "Pera", "Uva", "Maça")
```

Agora quando pedimos para ver o vetor nomeado ele aparece da seguinte forma:

```
sacola1
```

```
## Laranja    Pera     Uva    Maça
##      10       5       8       7
```

No caso, como temos apenas uma sacola de feira, o jeito que fizemos faz sentido: atribuímos um vetor e depois usamos a função `names` para nomeá-lo. Mas e se tivermos mais de um vetor de sacolas de feira, com as mesmas características dentro?

```
sacola2 <- c(5, 9, 7, 6)
sacola3 <- c(8, 7, 5, 4)
sacola4 <- c(9, 12, 3, 9)
sacola5 <- c(5, 3, 10, 12)
```

Agora faz sentido otimizarmos a nossa função. Para isso criamos um vetor chamado `nomes` e associamos ele a cada uma dos vetores:

```
nomes <- c("Laranja", "Pera", "Uva", "Maça")
names(sacola2) <- nomes
names(sacola3) <- nomes
names(sacola4) <- nomes
names(sacola5) <- nomes
```

Pronto! Conseguimos nomear todos os vetores de forma mais rápida.

3.1.2 Operações com vetores

Agora queremos saber qual foi o total de cada fruta comprada. Como já sabemos como fazer operações matemáticas com vetores precisamos apenas somar os vetores correspondentes das nossas sacolas de feira e adicionar em um vetor de soma.

```
soma_feira <- sacola1 + sacola2 + sacola3 + sacola4 + sacola5
soma_feira
```

```
## Laranja    Pera     Uva    Maça
##      37      36      33      38
```

Agora o nosso vetor `soma_feira` armazena a soma de cada fruta em todas as sacolas de feira. Assim, podemos responder: quantas laranjas foram compradas no total? Qual foi a fruta mais comprada?

O vetor total não é nomeado. Podemos nomeá-lo. Usando a função `names`.

Exercício 3.6. Nomeie o vetor total de sacolas de feira.

Mas e se quisermos saber a soma de itens de cada sacola? Nesse caso, fazer operaçóes com vetores não nos ajuda. Em outras linguagens de programação teríamos que programar laços de repetição e laços condicionais para realizar essa operação e, se quiséssemos utilizar em outras situações (reaproveitá-la) teríamos

que programar uma função. O R por já vir com varias funções pré-programadas facilita o nosso trabalho. Sendo assim, precisamos apenas utilizar a função `sum()`[1] (olhe a documentação da função).

Exercício 3.7. Faça a soma de cada um dos vetores de sacolas utilizando a função `sum()`. Armazene cada resultado em uma nova variável.

Exercício 3.8. Descubra o total de itens comprados na feira utilizando a função `sum()`. Armazene cada resultado em uma nova variável.

3.1.3 Selecionar elementos dentro de um vetor

Algo útil para fazermos é conseguirmos selecionar elementos dentro dos vetores. Para isso precisamos entender o conceito de `endereçamento`. Por exemplo, o vetor de tamanho quatro tem quatro espaços dentro dele, dentro de cada um desses espaços está armazenado um objeto (um número, um texto ou um objeto de tipo lógico), ou seja, esse vetor tem 4 "endereços" dentro dele. Isso nos facilita pois nem sempre sabemos o que tem dentro do vetor e para descobrirmos não precisamos pedir para o R nos retornar o vetor inteiro (às vezes o vetor é muito grande e esse precedimento se torna inviável).

Para indicarmos que queremos ver algo dentro de um `endereço` utilizamos a sintaxe de `[]`. Para indicarmos um endereço de um vetor utilizamos a sintaxe vetor[x], onde x indica um número de 1 ate n (a maior casa do vetor).

Exercício 3.9. Selecione o elemento 4 do vetor da sacola 3. Atribua esse valor a uma variável.

Podemos também selecionar mais de um valor do vetor. A sintaxe é indicar dentro dos `[]` a seleção a ser selecionada. Veja que isso indica um novo vetor então a sintaxe também deve seguir a sintaxe de um vetor: `vetor[c(x, y, z)]`. Se os números indicam um intervalo podemos utilizar a sintaxe `vetor[x:y]`.

Exercício 3.10. Selecione do vetor `sacola3` os valores 1 e 3 e do vetor sacola2 os valores de 2 a 4.

[1] Sum function: https://www.rdocumentation.org/packages/base/versions/3.6.1/topics/sum

Para vetores nomeados é possível selecionar a partir dos nomes. A lógica é a mesma, mas dentro dos `[]` adicionamos o nome entre `" "` (por ser uma variável de classe textual). Para selecionarmos mais de um nome também fazemos do mesmo jeito.

Exercício 3.11. Selecione a quantidade de laranjas do vetor da sacola 4. Selecione a quantidade de peras e maças do vetor da sacola 5.

3.1.4 Comparações de vetores e entre vetores

Assim como fizemos comparações entre os objetos criados, podemos fazer comparações entre os vetores. Compare o vetor `sacola1` com o vetor `sacola2`. Quando utilizamos os operadores relacionais entre dois vetores o R retorna a comparação de cada um dos itens, ou seja, a comparação do item 1 do primeiro vetor com o item 1 do segundo vetor e assim por diante. Retornando `TRUE` ou `FALSE` em cada um deles. Em vetores nomeados com os mesmos nomes esse procedimento é ainda mais fácil pois o R nos retorna o resultado das comparações com os nomes dos vetores.

Exercício 3.12. Qual dos itens do vetor `sacola2` são maiores que os itens do vetor `sacola4`?

Também podemos fazer essas comparações com a soma de cada sacola.

Exercício 3.13. A soma dos itens da `sacola3` é menor que a soma dos itens da `sacola5`?

Podemos utilizar os mesmos sinais de comparação para saber quais os valores de um vetor são maiores que um número, por exemplo. Assim se perguntarmos ao R `vetor > 5` ele irá nos responder elemento por elemento se é ou não maior que cinco (respondendo com operadores lógicos).

Exercício 3.14. Veja quais casas do vetor da sacola 5 tem valores maiores ou iguais a 10.

Porém, seria mais útil se tivessemos como retorno os números que correspondem a busca que desejamos. Para isso, precisamos inserir `vetor[vetor > x]`. Aqui o R irá retornar os números que são maiores que x. No caso do vetor nomeado a resposta virá acompanhada dos nomes. Lembre-se que dessa forma não temos como saber quais casas tem os valores retornados, o R retorna os elementos que são `TRUE` no seu vetor mas não indica qual era a posição original. Esse é o nosso primeiro contato um tipo de filtragem dos dados.

Exercício 3.15. Armazene em uma variável os valores do vetor `sacola1` que são menores do que 10.

3.1.5 Adição e exclusão de valores em um vetor

Podemos adicionar e excluir valores de um vetor existente. Para adicionarmos valores em um vetor podemos fazer de três formas distintas:

- **Por endereçamento direto** - dessa forma precisamos indicar dentro dos colchetes o endereço da última casa mais um:
  - vetor[x+1] <- 5
- **Por endereçamento indireto** - dessa forma não precisamos saber o tamanho do vetor, indicamos a partir da função `length()` o tamanho do vetor `+ 1`
  - vetor[length(vetor) + 1] <- 9
- **Por recursividade** - substituímos o vetor original por um novo vetor que tem o vetor original seguido de uma casa antes e/ou depois
  - vetor <- c(vetor, 10)
  - vetor <- c(10, vetor)

  Das duas primeiras formas, se indicamos um número diferente de 1 no endereço o R, diferente de outras linguagens, coloca `NA` nas casas entre a última casa do vetor e a casa nova atribuída. Veja o exemplo a seguir.

```
vetor1 <- c(1, 2, 4)
vetor1[5] <- 5
vetor1
```

```
## [1]  1  2  4 NA  5
```

Como o vetor tinha inicialmente 3 casas e indicamos um valor para uma 5ª casa, o R criou a 5ª casa com o valor indicado, mas precisava para tanto criar uma 4ª casa também. Essa ele incluiu um valor `NA` pois o valor dessa casa não foi indicado.

Exercício 3.16. Adicione os valores de soma, calculados anteiormente, nos respectivos vetores de sacolas de feira.

Por fim, para excluir elementos de um vetor, indicamos por recursividade o vetor dentro de [] os valores que queremos excluir com um traço na frente. Se queremos excluir um intervalo indicamos [-c(x:y)] e se queremos excluir números específicos indicamos [-c(x, y, z)].

```
vetor1 <- vetor1[-c(4)]
vetor1
```

```
## [1] 1 2 4 5
```

> No caso da exclusão não é necessario que seja por recursividade. É possivel atribuir um novo vetor para o procedimento.

3.1.6 Funções com vetores

Na última parte dessa aula iremos aprender algumas funções úteis para utilizar em vetores. Ainda que possamos criar vetores de sequência utilizando apenas a sintaxe `x:y`, por exemplo as funções `rep()` e `seq()` nos auxiliam a criar sequências mais personalizadas.

```
#Criando sequências sem as funções rep() ou seq()
seq1 <- 1:8
seq1
```

```
## [1] 1 2 3 4 5 6 7 8
```

```
seq2 <- 2.5:10
seq2
```

```
## [1] 2.5 3.5 4.5 5.5 6.5 7.5 8.5 9.5
```

A função `seq()` possibilita personalizar outros argumentos para a criação da sequência. O principal argumento é o `by` em que podemos definir o incremento da sequência. Por exemplo, se quero uma sequência de 0 a 10 de 2 em 2 utilizo: `seq(0, 10, by = 2)`. Veja outros argumentos na documentação da função `seq()`.

Exercício 3.17. Crie um vetor com uma sequência de numeros iniciando em -20 e indo ate 50 de 5 em 5. Atribua a uma variável.

A função `rep()` possibilita a criação de sequências de numeros repetidos. Por exemplo, se utilizarmos `rep(5,3)` iremos ter um vetor com o número 5 repetido 3 vezes.

```
rep(5,3)
```

```
## [1] 5 5 5
```

Para ter um intervalo podemos utilizar `rep(x:y, n)`, ou seja, um intervalo que vai de x a y repetido n vezes.

```
rep(1:5, 3)
```

```
##  [1] 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5
```

Por fim, podemos criar um intervalo repetindo cada número do intervalo n vezes.

```
rep(1:5, each = 3)
```

```
##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

Outras funções úteis para aprendermos agora são:

- mean(x) - cálculo da média do vetor x
- var(x) - cálculo da variância do vetor x
- max(x) - o valor máximo encontrado no vetor x
- min(x) - o valor mínimo encontrado no vetor x
- sd(x) - o desvio padrão do vetor x
- range(x) - a amplitude do vetor x
- length(x) - o tamanho do vetor x
- rev(x) - inverter o vetor x **Atenção: isso não altera o vetor original**

  Se colocarmos duas ou mais classes diferentes dentro de um mesmo vetor, o R vai forçar que todos os elementos passem a pertencer à mesma classe. Ordem de preferência: character > complex > numeric > integer > logical

3.2 Matrizes

Matrizes são vetores (*arrays*) bidimensionais. Justamente por serem vetores, herdam a mesma característica dos vetores: podem ser apenas de uma mesma classe. Para criar uma matriz precisamos utilizar a função `matrix()`, mas atenção é necessário tomar cuidado com o argumento `byrow`. Por padrão a função `matrix()` define o argumento `byrow` como `FALSE` e portanto, preenche a matriz por colunas. Se quisermos que preencher por linhas precisamos atribuir o argumento `byrow` como `TRUE`. Veja a diferença:

```
matrix(1:30, byrow = FALSE, nrow = 10)
```

```
##       [,1] [,2] [,3]
##  [1,]    1   11   21
##  [2,]    2   12   22
##  [3,]    3   13   23
##  [4,]    4   14   24
##  [5,]    5   15   25
##  [6,]    6   16   26
##  [7,]    7   17   27
##  [8,]    8   18   28
##  [9,]    9   19   29
## [10,]   10   20   30
```

O comando anterior criou uma matriz de 1 a 30 preenchendo por colunas. Agora alterando o argumento `byrow`:

```
matrix(1:30, byrow = TRUE, nrow = 10)
```

```
##       [,1] [,2] [,3]
##  [1,]    1    2    3
##  [2,]    4    5    6
##  [3,]    7    8    9
##  [4,]   10   11   12
##  [5,]   13   14   15
##  [6,]   16   17   18
##  [7,]   19   20   21
##  [8,]   22   23   24
##  [9,]   25   26   27
## [10,]   28   29   30
```

Apesar do primeiro e último item das duas matrizes serem os mesmos todos os outros são bem diferentes. O agumento `nrow` define o número de linhas que desejamos na matriz. No caso, como temos 30 numeros e definimos que queremos 10 linhas teremos 3 colunas ($30/10 = 3$). Se definíssemos 5 linhas teriamos 6 colunas. Como a seguir:

```
matrix(1:30, byrow = TRUE, nrow = 5)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    2    3    4    5    6
## [2,]    7    8    9   10   11   12
## [3,]   13   14   15   16   17   18
## [4,]   19   20   21   22   23   24
## [5,]   25   26   27   28   29   30
```

Utilize sempre múltiplos do número desejado se não o R faz a reciclagem das casas e cria uma matriz maior do que a desejada. Como a seguir:

```
matrix(1:30, byrow = TRUE, nrow = 7)
```

```
## Warning in matrix(1:30, byrow = TRUE, nrow = 7): data length [30] is not a
## sub-multiple or multiple of the number of rows [7]
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
## [4,]   16   17   18   19   20
## [5,]   21   22   23   24   25
## [6,]   26   27   28   29   30
## [7,]    1    2    3    4    5
```

De modo equivalente o argumento `ncol` define o número de colunas da matriz. Os dois argumentos são complemetares, não sendo necessário utilizá-los concomitantemente.

Vamos criar uma matriz com os vetores que tínhamos anteriormente (sacolas de feira). Primeiro criamos um vetor com todas as sacolas concatenadas:

```
sacolas <- c(sacola1, sacola2, sacola3, sacola4, sacola5)
sacolas
```

```
## Laranja   Pera    Uva   Maça Laranja   Pera    Uva   Maça Laranja
##      10      5      8      7      5      9      7      6      8
##   Pera    Uva   Maça Laranja   Pera    Uva   Maça Laranja   Pera
##      7      5      4      9     12      3      9      5      3
##     Uva    Maça
##      10     12
```

Cuidado com a ordem das sacolas!

Depois criamos uma matriz com essas sacolas. Cuidado com o argumento `byrow`! Aqui precisamos que a matriz seja preenchida por linhas, então o argumento `byrow` deve ser verdadeiro! O argumento `nrow` será o número das nossas sacolas.

```
matriz_sacola <- matrix(sacolas, byrow = T, nrow = 5)
matriz_sacola
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   10    5    8    7
## [2,]    5    9    7    6
## [3,]    8    7    5    4
## [4,]    9   12    3    9
## [5,]    5    3   10   12
```

Veja que agora o R criou um objeto diferente: do tipo `Data`. Isso significa que o R interpreta a matriz de forma diferente dos vetores e objetos simples. Uma outra forma de fazer a matriz é fazer de forma direta o vetor dentro do primeiro argumento: `matrix(c(sacola1, sacola2, sacola3, sacola4, sacola5), byrow = T, nrow = 5)`

Porém ao transformar em matriz perdemos os nomes dos nossos vetores (nossas colunas). Vamos criá-los novamente.

Para criar os nomes das colunas precisamos utilizar agora a função `colnames`. Precisamos especificar onde vão os nomes pois agora não estamos mais trabalhando com vetores unidimensionais. Para isso podemos reaproveitar o nosso vetor `nomes`, que utilizamos anteriormente.

```
colnames(matriz_sacola) <- nomes
```

Dissemos para o R que os nomes das colunas de `matriz_sacola` correspondem ao vetor nomes.

Agora precisamos definir os nomes das nossas linhas. Para isso utilizamos a função `rownames`. Como não temos nenhum vetor de nomes para linhas vamos fazer direto dentro da função:

```
rownames(matriz_sacola) <- c("sacola1", "sacola2", "sacola3", "sacola4", "sacola5")
```

Pronto! Agora temos uma matriz com nomes nas linhas e colunas.

```
matriz_sacola
```

```
##         Laranja Pera Uva Maça
## sacola1      10    5   8    7
## sacola2       5    9   7    6
## sacola3       8    7   5    4
## sacola4       9   12   3    9
## sacola5       5    3  10   12
```

Agora queremos fazer as somas das linhas e das colunas. Esse processo pode ser um pouco confuso, por isso precisamos tomar bastante cuidado!

Primeiro fazemos as somas das linhas, ou seja, criamos um vetor que tem a soma das linhas. Como temos 5 linhas, esse vetor terá 5 elementos. Para isso utilizamos a função `rowSums()` na nossa matriz, criando um vetor de soma:

```
somaL <- rowSums(matriz_sacola)
somaL
```

```
## sacola1 sacola2 sacola3 sacola4 sacola5
##      30      27      24      33      30
```

Esse vetor será correspondente a uma nova **coluna** da nossa matriz. Por isso utilizamos a função `cbind()` para colar essa nova coluna na nossa matriz. A função `cbind()` precisa como primeiro argumento o nome da nossa matriz e como segundo o que vamos colar nela (o vetor somaL). Porém a função não sobrescreve a matriz original, então precisamos atribuir toda a função a matriz original:

```
matriz_sacola <- cbind(matriz_sacola, somaL)
matriz_sacola
```

```
##         Laranja Pera Uva Maça somaL
## sacola1      10    5   8    7    30
## sacola2       5    9   7    6    27
## sacola3       8    7   5    4    24
## sacola4       9   12   3    9    33
## sacola5       5    3  10   12    30
```

Agora queremos a soma das colunas. Para isso utilizamos a função `colSums()` e atribuímos ela a um vetor:

```
somaC <- colSums(matriz_sacola)
somaC
```

```
## Laranja    Pera     Uva    Maça   somaL
##      37      36      33      38     144
```

O vetor `somaC` será correspondente a uma nova **linha** da nossa matriz. Assim, precisamos unir essa linha a nossa matriz. Para isso, utilizamos a função `rbind()`. Da mesma forma que a função `cbind()` funciona, a função `rbind()` não sobrescreve a matriz original. Portanto, precisamos atribuir essa função à matriz original:

```
matriz_sacola <- rbind(matriz_sacola, somaC)
matriz_sacola
```

```
##         Laranja Pera Uva Maça somaL
## sacola1      10    5   8    7    30
## sacola2       5    9   7    6    27
## sacola3       8    7   5    4    24
## sacola4       9   12   3    9    33
## sacola5       5    3  10   12    30
## somaC        37   36  33   38   144
```

> De modo a diminuir os passos é possível unir o passo 1 e 2 da seguinte forma: cbind(rowSums(matriz_sacola), matriz_sacola). E os passos 3 e 4 da seguinte forma: rbind(colSums(matriz_sacola), matriz_sacola). Sempre atribuindo a `matriz_sacola`.

3.2.1 Seleção de elementos matriz

Podemos selecionar elementos internos dentro de uma matriz. Para tanto, precisamos indicar o endereço que queremos, como fizemos com a seleção de vetores. Porém dessa vez precisamos indicar o endereço da linha e da coluna. A sintaxe para esse tipo de seleção é: `matrix[x, y]` onde `x` é o número da linha que você quer selecionar e `y` o número da coluna.

Exercício 3.18. Selecione o terceiro elemento da segunda coluna da `matriz_sacola`.

Da mesma forma, podemos selecionar uma coluna ou uma linha inteira da matriz. Para selecionar uma linha inteira deixamos o valor de `y` vazio e se quisermos selecionar uma coluna inteira deixamos o valor de `x` vazio.

Exercício 3.19. Selecione a 5ª linha da `matriz_sacola`.

Exercício 3.20. Selecione a 3ª coluna da `matriz_sacola`.

3.2.2 Operações com matrizes

Da mesma forma que fizemos operações com os vetores podemos fazer operações (+, -, /, *, ^, %%, %/%) de matrizes com escalares, com vetores ou com outras matrizes. Apenas temos que tomar cuidado com o tamanho das matrizes. Se elas forem de tamanho diferentes o R irá reciclar (repetir) a matriz menor.

> O operador `*` não é equivalente a multiplicação matricial das matrizes. Utilizando esse operador o R multiplica o primeiro item da matriz A com o primeiro item da matriz B. Para realizar multiplicação matricial precisamos utilizar o operador `%*%`.

Exercício 3.21. Some duas matrizes do mesmo tamanho. Atribua o resultado a uma terceira matriz.

Exercício 3.22. Some uma matriz com um vetor. Atribua o resultado a uma terceira matriz.

3.3 Factors

Factor[2] é uma estrutura de dados no R utilizada para armazenar variáveis categóricas. São considerados como uma classe especial de vetores e utilizados, principalmente, em análises estatísticas – as variáveis categóricas são interpretadas por modelos estatísticos de forma diferente das variáveis contínuas. Assim, quando armazenamos dados como `factors` garantimos que diversas funções presentes nos pacotes do R tratem esses dados de forma correta.

Ao utilizarmos a função `factor()` o R armazena um vetor de valores inteiros com os correspondentes em rótulos categóricos para serem usados quando o factor é solicitado.

Por exemplo, se tenho um vetor numerico de 0 e 1 e utilizo a função `factor()` informando ao R que o 0 equivale `masculino` e 1 equivale a `feminino`. Ele irá interpretar os 0 e os 1 do meu vetor com o seus respectivos rótulos:

```
genero <- c(0, 1, 0, 1, 1, 0, 0, 1)
genero <- factor(genero, labels = c("feminino", "masculino"))
```

Da mesma forma, posso inserir um vetor textual e utilizar a função `factor()`, o R irá interpretar todos os diferentes caracteres do vetor como um nível diferente. Desse jeito, precisamos tomar cuidado ao inserir os elementos textuais pois "f" é diferente de "F" e o R interpretaria como 2 níveis diferentes.

```
genero <- c("M", "F", "F", "M", "M", "F", "F", "M")
genero <- factor(genero)
```

Também podemos fazer factors que sejam ordenados, ou seja, informar que existe uma ordem pré-estabelecida das categorias. Para isso adicionamos o argumento `order = T` e no argumento `levels` inserimos as categorias na ordem desejada:

```
vetor_satisf <- c("Bom", "Ruim", "Excelente", "Razoável", "Razoável", "Ótimo", "Bom",
vetor_satisf <- factor(vetor_satisf, order = TRUE, levels = c("Péssimo", "Ruim", "Razoa
```

> Se informamos que há uma ordem das categorias (`order= TRUE`), mas não especificamos a ordem no argumento `levels` o R irá definir a ordem pela ordem alfabética dos fatores.

3.4 Dataframes

Como os elementos dentro de uma matriz não podem ser de classes/tipos diferentes precisamos de um outro tipo de objeto para tratar banco de dados quando

[2] https://www.stat.berkeley.edu/~s133/factors.html

queremos trabalhar com vários tipos de dados. Um *dataframe* é capaz de armazenar objetos de classes diferentes e trata as colunas como variáveis e as linhas como observações (casos). Porém, não podemos armazenar objetos de classes diferentes dentro de uma mesma coluna (ou variável).

Para criar um *dataframe* utilizamos a função `data.frame()`. Podemos indicar vetores ou uma matriz para serem transformados em data frames. Porém, a função `data.frame()` vê os vetores inseridos como colunas. Então, para trnasformar os nossos vetores `sacolas` em data frame precisamos ou primeiro transformar eles em uma matriz e preenchê-lo da maneira correta ou utilizamos a função `transpose()` do pacote `data.table` (dessa forma não teremos os nomes das linhas, das colunas, a soma das linhas ou das colunas).

```
data_sacola <- data.frame(matriz_sacola)
data_sacola
```

```
##         Laranja Pera Uva Maça somaL
## sacola1      10    5   8    7    30
## sacola2       5    9   7    6    27
## sacola3       8    7   5    4    24
## sacola4       9   12   3    9    33
## sacola5       5    3  10   12    30
## somaC        37   36  33   38   144
```

Criamos um data frame com os dados das sacolas que tínhamos mas o nosso maior objetivo é utilizar data frames para banco de dados com classes de objetos diferentes. Vamos, então, criar um data frame com variáveis de diversos tipos. Dentro de cada linha da função `data.frame()` há a criação de uma variavel. As que tem a função `sample()` fazem sorteios dentro dos limites especificados (com repetição), as com a função `factor()` criam variáveis categóricas com a repetição definida no vetor do segundo argumento e a função `paste()` cria uma sequência de `0000` seguida de números de 1 a 30 (o equivalente de um ID de indivíduos). Por fim, a função `colnames()` define os nomes das colunas do nosso banco:

```
banco <- data.frame(paste("0000", 1:30, sep = ""),
                    factor(rep(c("b", "n", "i", "o"), c(10, 10, 6, 4))),
                    factor(rep(c("f","m"), c(16,14))),
                    sample(c(16:60), 30, replace = T),
                    factor(rep(c("superior","tecnico", "medio", "fundamental"), c(5, 8, 12, 5))),
                    sample(seq(1000, 30000, by = 1000), 30, replace = T),
                    factor(rep(c("solteiro","casado", "viuvo", "separado"), c(10, 10, 2, 8))))
colnames(banco) <- c("individuo", "raca", "sexo", "idade", "escol", "renda", "civil")
banco
```

```
##    individuo raca sexo idade       escol renda    civil
## 1      00001    b    f    23    superior  8000 solteiro
## 2      00002    b    f    35    superior  7000 solteiro
```

```
## 3      00003    b    f    43    superior  8000 solteiro
## 4      00004    b    f    58    superior  1000 solteiro
## 5      00005    b    f    32    superior 13000 solteiro
## 6      00006    b    f    31     tecnico 20000 solteiro
## 7      00007    b    f    39     tecnico  2000 solteiro
## 8      00008    b    f    48     tecnico  6000 solteiro
## 9      00009    b    f    19     tecnico 25000 solteiro
## 10    000010    b    f    18     tecnico 25000 solteiro
## 11    000011    n    f    28     tecnico 25000   casado
## 12    000012    n    f    52     tecnico 29000   casado
## 13    000013    n    f    39     tecnico 24000   casado
## 14    000014    n    f    50       medio 16000   casado
## 15    000015    n    f    42       medio 30000   casado
## 16    000016    n    f    59       medio  3000   casado
## 17    000017    n    m    53       medio 23000   casado
## 18    000018    n    m    21       medio 30000   casado
## 19    000019    n    m    50       medio 10000   casado
## 20    000020    n    m    40       medio  4000   casado
## 21    000021    i    m    50       medio 13000    viuvo
## 22    000022    i    m    21       medio 15000    viuvo
## 23    000023    i    m    46       medio  6000 separado
## 24    000024    i    m    50       medio 23000 separado
## 25    000025    i    m    18       medio 23000 separado
## 26    000026    i    m    39 fundamental 16000 separado
## 27    000027    o    m    22 fundamental 26000 separado
## 28    000028    o    m    19 fundamental 23000 separado
## 29    000029    o    m    55 fundamental 30000 separado
## 30    000030    o    m    39 fundamental  4000 separado
```

Como as variáveis idade e renda são construídas a partir de um sorteio aleatório cada vez que o código é rodado são criados resultados diferentes.

3.4.1 Funções básicas para data frames

Toda a vez que abrimos um novo data frame é importante utilizarmos 4 funções básicas: `head()`, `tail()`, `str()` e `summary()`.

As funções `head()` e `tail()` mostram, respectivamente, os primeiros e os últimos 6 casos (linhas) do seu banco de dados. Se você quiser aumentar o número de linhas mostrado precisa adicionar o argumento `n =` e definir a quantidade desejada.

Exercício 3.23. Mostre os 10 primeiros e 10 últimos itens do data frame `banco`.

A função `str()` retorna a estrutura de cada uma das variáveis do banco:

```
str(banco)
```

```
## 'data.frame':    30 obs. of  7 variables:
## $ individuo: Factor w/ 30 levels "00001","000010",..: 1 12 23 25 26 27 28 29 30 2 ...
## $ raca    : Factor w/ 4 levels "b","i","n","o": 1 1 1 1 1 1 1 1 1 1 ...
##  $ sexo     : Factor w/ 2 levels "f","m": 1 1 1 1 1 1 1 1 1 1 ...
##  $ idade    : int  23 35 43 58 32 31 39 48 19 18 ...
## $ escol   : Factor w/ 4 levels "fundamental",..: 3 3 3 3 3 4 4 4 4 4 ...
## $ renda   : num  8000 7000 8000 1000 13000 20000 2000 6000 25000 25000 ...
## $ civil   : Factor w/ 4 levels "casado","separado",..: 3 3 3 3 3 3 3 3 3 3 ...
```

E a função `summary()` retorna um resumo de todas as variáveis:

```
summary(banco)
```

```
##    individuo  raca   sexo       idade              escol
##  00001  : 1   b:10   f:16   Min.   :18.00   fundamental: 5
##  000010 : 1   i: 6   m:14   1st Qu.:24.25   medio      :12
##  000011 : 1   n:10          Median :39.00   superior   : 5
##  000012 : 1   o: 4          Mean   :37.97   tecnico    : 8
##  000013 : 1                 3rd Qu.:50.00
##  000014 : 1                 Max.   :59.00
##  (Other):24
##      renda            civil
##  Min.   : 1000   casado  :10
##  1st Qu.: 7250   separado: 8
##  Median :16000   solteiro:10
##  Mean   :16267   viuvo   : 2
##  3rd Qu.:24750
##  Max.   :30000
##
```

Com essas funções conseguimos ver como o R está interpretando cada uma das variáveis. Já vimos, por exemplo, que ele armazenou a variável indivíduo como um `factor`, o que não é o que gostaríamos. Depois veremos como transformar essa variável em texto.

3.4.2 Selecionando elementos dentro de um dataframe

Para selecionar elementos dentro de um dataframe podemos trabalhar com a mesma lógica das matrizes. Utilizando os símbolos `[ , ]` para delimitar a linha (ou intervalo de linhas) e a coluna (ou intervalo de colunas).

Exercício 3.24. Selecione o elemento que está armazenado na linha 20 e na coluna 4.

Exercício 3.25. Selecione a linha 10 inteira. Selecione a coluna 2 inteira.

Podemos também selecionar um intervalo dentro de uma coluna. Para isso colocamos o intervalo de linhas que queremos (p.ex. `4:15`) e colocamos o número da coluna desejada. Porém, como nomeamos as nossas colunas, podemos colocar o nome da coluna no lugar do número:

```
banco[9:20, "renda"]
```

```
##  [1] 25000 25000 25000 29000 24000 16000 30000  3000 23000 30000 10000
## [12]  4000
```

Da mesma forma, podemos selecionar variáveis de um caso ou de uma seleção de casos:

```
banco[c(3, 4, 8), 3:5]
```

```
##   sexo idade    escol
## 3    f    43 superior
## 4    f    58 superior
## 8    f    48  tecnico
```

No exemplo anterior selecionamos dos indivíduos 3, 4 e 8 as variáveis de 3 a 5 (`sexo`, `idade` e `escol`).

Porém, os objetos do tipo `data frames` tem uma caracteristica diferente: o R automaticamente lê as colunas como variáveis. Então, podemos indicar as colunas a partir da sintaxe `$`. A função `subset()` nos ajuda a fazer filtragens mais avançadas:

```
subset(banco, banco$sexo == "m")
```

```
##    individuo raca sexo idade       escol renda    civil
## 17    000017    n    m    53       medio 23000   casado
## 18    000018    n    m    21       medio 30000   casado
## 19    000019    n    m    50       medio 10000   casado
## 20    000020    n    m    40       medio  4000   casado
## 21    000021    i    m    50       medio 13000    viuvo
## 22    000022    i    m    21       medio 15000    viuvo
## 23    000023    i    m    46       medio  6000 separado
## 24    000024    i    m    50       medio 23000 separado
## 25    000025    i    m    18       medio 23000 separado
```

```
## 26    000026    i    m    39 fundamental 16000 separado
## 27    000027    o    m    22 fundamental 26000 separado
## 28    000028    o    m    19 fundamental 23000 separado
## 29    000029    o    m    55 fundamental 30000 separado
## 30    000030    o    m    39 fundamental  4000 separado
```

Selecionamos no exemplo anterior apenas os casos que correspondem a seleção `sexo == "m"`, ou seja, serem do sexo `masculino`. Podemos combinar filtragens do banco utilizando os operadores lógicos (& e |) e relacionais (> < >= <= != ==) já aprendidos anteriormente.

Exercício 3.26. Atribua a uma nova variável os indivíduos que tem idade menor que 35 anos e não tem nível superior.

A função `subset()` nos retorna os casos inteiros, mas as vezes apenas queremos saber quais casos correspondem a nossa filtragem. Podemos, então, utilizar a função `which()`.

```
which(banco$escol != "superior" & banco$renda > 5000)
```

```
##  [1]  6  8  9 10 11 12 13 14 15 17 18 19 21 22 23 24 25 26 27 28 29
```

Podemos, por fim, ordenar nosso banco a partir de uma variável desejada. Para isso, utilizamos a função `order()` e definimos no argumento se terá a ordem crescente ou decrescente. Essa função nos retorna um vetor com as posições das variáveis na ordem desejada. A partir desse vetor podemos criar um data frame novo.

```
ordem_nova <- order(banco$idade, decreasing = "F")
```

No comando anterior criamos um vetor com as posições que correspondem a ordem do nosso banco a partir da variável idade de forma crescente.

```
banco_ord <- banco[ordem_nova, ]
```

Agora temos um novo banco com a ordem desejada.

3.5 Listas

O último tipo de objeto que iremos ver são as listas, mas antes vamos recapitular os objetos que já vimos:

- Vetores - são *arrays* de uma dimensão, podem ter valores `numeric`, `character`, `logical`, `integer` ou `complex`. Porém, os elementos dentro

de um vetor sempre devem ter a mesma classe o vetor, portanto, herda essa classe.

- Matrizes - são *arrays* de duas dimensões, podem ter valores `numeric`, `character`, `logical`, `integer` ou `complex`. São criadas a partir dos vetores portanto, herdam a mesma característica: elementos dentro de uma matriz tem sempre a mesma classe/tipo.
- Data frames - objetos bidimensionais, podem ter valores `numeric`, `character`, `logical`, `integer` ou `complex` dentro de um mesmo objeto. Porém, os elementos de uma coluna devem ser do mesmo tipo de dado, mas colunas diferentes podem ter tipos diferentes de dados.

As listas, por sua vez, aceitam diferentes tipos de dado, de diferentes tamanhos, características. Podem armazenar objetos de forma ordenada, que podem ser matrizes, vetores, dataframes ou outras listas. Não é necessário que estejam ligados de alguma forma. Listas são um **super data**!

Para criar uma lista utilize a função `list()` e concatene dentro dos parênteses os objetos que quer colocar dentro da lista.

Chapter 4

Manipulando e analisando data frames

Agora que já conhecemos todos os tipos de objetos no R, vamos começar a trabalhar com bancos de dados. Aqui iremos trabalhar com os dados dos passageiros do Titanic que estão disponíveis no pacote `Stat2Data`. Então, a primeira coisa que precisamos fazer é instalar o pacote via comando `install.packages("Stat2Data")`. Após a instalação precisamos chamar o pacote e depois precisamos baixar o banco que iremos utilizar:

```
library(Stat2Data)
data("Titanic")
```

A primeira coisa que fazemos quando abrimos um banco de dados novo é ver como ele está organizado, quais são suas variavei e se ele tem muitos erros. Para isso é útil utilizarmos as funções `head()` e `tail()`. Como padrão elas mostram as seis primeiras e seis últimas linhas do banco. Caso, desejem que o R mostre mais itens é possível utilizar o argumento `n =` dentro da função e inserir o número desejado.

Exercício 4.1. Veja os primeiros 10 casos e últimos 10 casos do banco Titanic.

Depois de vermos como está o começo (head) e o final (tail) do nosso banco, utilizamos duas outras funções que nos mostrarão como está estruturado o banco e um resumo de como estão as variáveis.

```
str(Titanic)
```

```
## 'data.frame':    1313 obs. of  6 variables:
```

```
## $ Name   : Factor w/ 1310 levels "Abbing, Mr Anthony",..: 22 25 26 27 24 31 45 46 50 54 ..
## $ PClass  : Factor w/ 4 levels "*","1st","2nd",..: 2 2 2 2 2 2 2 2 2 2 ...
##  $ Age     : num  29 2 30 25 0.92 47 63 39 58 71 ...
## $ Sex    : Factor w/ 2 levels "female","male": 1 1 2 1 2 2 1 2 1 2 ...
##  $ Survived: int  1 0 0 0 1 1 1 0 1 0 ...
##  $ SexCode : int  1 1 0 1 0 0 1 0 1 0 ...
```

A função `str()`, de *structure* nos mostra a estrutura do nosso banco. Aqui já conseguimos ver que temos seis variáveis e as classes de cada uma delas. Já conseguimos identificar alguns problemas no nosso banco. Você consegue identificá-los?

```
summary(Titanic)
```

```
##                              Name      PClass         Age
##  Carlsson, Mr Frans Olof      :   2   *  :  1   Min.   : 0.17
##  Connolly, Miss Kate          :   2   1st:322   1st Qu.:21.00
##  Kelly, Mr James              :   2   2nd:279   Median :28.00
##  Abbing, Mr Anthony           :   1   3rd:711   Mean   :30.40
##  Abbott, Master Eugene Joseph:   1             3rd Qu.:39.00
##  Abbott, Mr Rossmore Edward   :   1             Max.   :71.00
##  (Other)                      :1304             NA's   :557
##      Sex         Survived         SexCode
##  female:462   Min.   :0.0000   Min.   :0.0000
##  male  :851   1st Qu.:0.0000   1st Qu.:0.0000
##               Median :0.0000   Median :0.0000
##               Mean   :0.3427   Mean   :0.3519
##               3rd Qu.:1.0000   3rd Qu.:1.0000
##               Max.   :1.0000   Max.   :1.0000
##
```

A função `summary()`, mostra um resumo das nossas variáveis. Conseguimos identificar outros problemas no nosso banco. Quais?

> Cada banco de dados terá erros e problemas diferentes. Aqui, iremos trabalhar alguns desses possíveis erros como exemplo.

4.1 Arrumando o banco de dados

Podemos identificar os seguintes problemas no banco Titanic:

1 - A variável `Name` está sendo lida pelo R como um factor enquanto deveria estar sendo lida como um `char` 2 - A variável `PClass` tem um elemento atribuído como `*` sendo lido como um dos fatores, precisamos transformá-lo em `NA` 3 - A variável `Age` tem valores menores do que 1 com decimais. é melhor transformar esses valores em `0` 4 - As variáveis `Survived` e `SexCode` estão sendo lidas como `numeric`, precisamos lê-las como `factor`

Vamos resolver cada um dos problemas a seguir.

Primeiro, transformamos as variáveis que precisam ser transformadas em outras classes. Para mudar a classe de uma variável utilizamos as funções da família `as.classe()` (onde "classe" é a classe para qual queremos transformar). A seguir transformamos a variável `Name` em `char` substituindo a coluna original:

```
Titanic$Name <- as.character(Titanic$Name)
```

> Quando fazemos a alteração da variável do modo acima estamos substituindo a coluna original. Caso isso não seja o desejado é necessário mudar o nome da coluna na atribuição. Se a coluna não existir o R irá criar uma coluna nova com esse nome.

E transformamos as variáveis `Survived` e `SexCode` em factor:

```
Titanic$Survived <- as.factor(Titanic$Survived)
Titanic$SexCode <- as.factor(Titanic$SexCode)
```

Agora vamos resolver o problema com a variável `PClass`. Para descobrirmos qual elemento tem o valor `*` precisamos utilizar a função `which()` da seguinte forma:

```
which(Titanic$PClass == "*")
```

```
## [1] 457
```

Agora sabemos que o elemento `457` armazena na variável `PClass` o elemento `*`.

Vamos dar uma olhada no elemento `457` inteiro?

```
Titanic[457, ]
```

```
##                     Name PClass Age  Sex Survived SexCode
## 457 Jacobsohn Mr Samuel      *  NA male        0       0
```

Para substituir o valor dentro da `PClass` utilizamos o seguinte comando:

```
Titanic$PClass[Titanic$PClass == "*"] <- NA
```

Vamos verificar novamente o elemento `457`:

```
Titanic[457, ]
```

```
##                     Name PClass Age  Sex Survived SexCode
## 457 Jacobsohn Mr Samuel   <NA>  NA male        0       0
```

Fizemos a substituição, mas se pedirmos novamente o resumo das variáveis veremos que o factor `*` ainda estará lá, ainda que nos informe que nenhum caso corresponde a esse `factor`:

```
summary(Titanic)
```

```
##      Name           PClass        Age           Sex      Survived
```

```
##  Length:1313        *   :  0   Min.   : 0.17   female:462   0:863
##  Class :character   1st :322   1st Qu.:21.00   male  :851   1:450
##  Mode  :character   2nd :279   Median :28.00
##                     3rd :711   Mean   :30.40
##                     NA's:  1   3rd Qu.:39.00
##                                Max.   :71.00
##                                NA's   :557
##  SexCode
##  0:851
##  1:462
##
##
##
##
##
```

Precisamos atualizar os fatores da `PClass`:

```
Titanic$PClass <- factor(Titanic$PClass)
```

Agora sim:

```
summary(Titanic)
```

```
##      Name           PClass        Age           Sex      Survived
##  Length:1313        1st :322   Min.   : 0.17   female:462   0:863
##  Class :character   2nd :279   1st Qu.:21.00   male  :851   1:450
##  Mode  :character   3rd :711   Median :28.00
##                     NA's:  1   Mean   :30.40
##                                3rd Qu.:39.00
##                                Max.   :71.00
##                                NA's   :557
##  SexCode
##  0:851
##  1:462
##
##
##
##
##
```

Bom, agora só falta arrumarmos a variável idade.

Primeiro vamos descobrir quais elementos tem o valor menor do que 1:

```
which(Titanic$Age < 1)
```

```
## [1]   5 359 545 617 752 764
```

Seria bom se também pudéssemos dar uma olhadas nesses elementos, já que não

são tantos. Lembram da função `subset()`?

```
subset(Titanic, Titanic$Age < 1)
```

```
##                                     Name PClass  Age    Sex Survived
## 5          Allison, Master Hudson Trevor   1st 0.92   male        1
## 359         Caldwell, Master Alden Gates   2nd 0.83   male        1
## 545       Richards, Master George Sidney   2nd 0.80   male        1
## 617                  Aks, Master Philip   3rd 0.83   male        1
## 752 Danbom, Master Gilbert Sigvard Emanuel   3rd 0.33   male      0
## 764 Dean, Miss Elizabeth Gladys (Millvena)   3rd 0.17 female      1
##     SexCode
## 5         0
## 359       0
## 545       0
## 617       0
## 752       0
## 764       1
```

Vamos utilizar a mesma lógica da substituição de `PClass` para substituir as idades menores que 1 por 0:

```
Titanic$Age[Titanic$Age < 1] <- 0
```

Agora que arrumamos tudo que precisavamos arrumar no banco, vamos dar uma olhada no nosso banco:

```
summary(Titanic)
```

```
##      Name           PClass         Age           Sex      Survived
##  Length:1313        1st :322   Min.   : 0.00   female:462   0:863
##  Class :character   2nd :279   1st Qu.:21.00   male  :851   1:450
##  Mode  :character   3rd :711   Median :28.00
##                     NA's:  1   Mean   :30.39
##                                3rd Qu.:39.00
##                                Max.   :71.00
##                                NA's   :557
##  SexCode
##  0:851
##  1:462
##
##
##
##
##
```

4.2 Criando variáveis novas

Podemos criar uma variável nova transformando a variável `Age` em categorias. Para isso precisamos utilizar a função `cut()`. Iremos cortar a nossa variável em 5 categorias: `até 15 anos`, `de 15 a 30 anos`, `de 30 a 40 anos`, `de 40 a 60 anos` e `mais de 60 anos`. Precisamos identificar no argumento `breaks` os limites do intervalos desejados a partir de um vetor ou inserir o número de intervalos (nesse caso a função irá definir os cortes, depois será necessário mudar os nomes dos cortes). Para criar uma variável nova precisamos atribuir a função em uma coluna do banco `Titanic` que ainda não existe (o R irá criá-la com a atribuição desejada):

```
Titanic$AgeCat <- cut(Titanic$Age,
                      breaks = c(-Inf,15,30,40,60,75),
                      labels = c("Até 15", "De 15 a 30",
                                 "De 30 a 40", "De 40 a 60", "Mais de 60"))
```

Vamos dar uma olhada no nosso banco:

```
summary(Titanic)
```

```
##     Name           PClass        Age           Sex      Survived
##  Length:1313       1st :322   Min.   : 0.00   female:462   0:863
##  Class :character   2nd :279   1st Qu.:21.00   male  :851   1:450
##  Mode  :character   3rd :711   Median :28.00
##                     NA's:  1   Mean   :30.39
##                                3rd Qu.:39.00
##                                Max.   :71.00
##                                NA's   :557
##  SexCode        AgeCat
##  0:851   Até 15    : 73
##  1:462   De 15 a 30:359
##          De 30 a 40:150
##          De 40 a 60:152
##          Mais de 60: 22
##          NA's      :557
##
```

> Caso precise remover alguma coluna do banco utilize a seguinte lógica: `banco <- banco[,-x]` onde o x é o número da coluna que queremos excluir. Se quiser excluir mais de uma coluna utilize `-c(x, y, z)` (para colunas separadas) ou `-[x:y]` (para intervalos de coluna)

##Tabelas de frequência e de contigência

Algo muito comum nas primeiras análises com bancos de dados são as tabelas de frequências. Com elas podemos ver como estão distribuídos nossos dados. A função `table()` retorna a frequência absoluta da variável desejada:

```
freq_class <- table(Titanic$PClass)
freq_class
```

```
##
## 1st 2nd 3rd
## 322 279 711
```

Se quisermos saber a frequência relativa utilizamos a função `prop.table()` junto com a função anterior (ou atribuímos a função `table()` a uma variável e depois utilizamos `prop.table()` nessa variável).

```
prop.table(table(Titanic$PClass))
```

```
##
##       1st       2nd       3rd
## 0.2454268 0.2126524 0.5419207
```

Outra tabela importante é a tabela de contigência (ou tabela cruzada), em que queremos ver a frequência das ocorrências dos cruzamentos das varia'veis, considerando suas categorias. As funções `table()` e `ftable()` retornam as tabelas de contingência em uma matriz. Para o cruzamento de duas variáveis o resultado da utilização das duas funções são bem parecidas:

```
class_sex <- table(Titanic$PClass, Titanic$Sex)
class_sex
```

```
##
##       female male
##   1st    143  179
##   2nd    107  172
##   3rd    212  499
```

```
ftable(Titanic$PClass, Titanic$Sex)
```

```
##      female male
##
## 1st     143  179
## 2nd     107  172
## 3rd     212  499
```

Porém, quando queremos cruzar mais de duas variáveis vemos diferenças relevantes entre os resultados das funções:

```
table(Titanic$PClass, Titanic$Sex, Titanic$Survived)
```

```
## , ,  = 0
##
##
##       female male
##   1st      9  120
```

```
##   2nd     13  147
##   3rd    132  441
##
## , ,  = 1
##
##
##       female male
##   1st    134   59
##   2nd     94   25
##   3rd     80   58
```

A função `table()` separa a última variável inserida e mostra as tabelas cruzadas com as outras variáveis para cada uma das categorias da variável três. De forma diferente, a função `ftable()` retorna uma tabela organizada com as três variáveis:

```
ftable(Titanic$PClass, Titanic$Sex, Titanic$Survived)
```

```
##               0   1
##
## 1st female    9 134
##     male    120  59
## 2nd female   13  94
##     male    147  25
## 3rd female  132  80
##     male    441  58
```

Ainda, podemos utilizar a função `ftable()` com a função `xtabs()` para termos uma tabela de contingência com os nomes corretos das variáveis.

```
ftable(xtabs(~PClass+Sex+Survived, data = Titanic))
```

```
##               Survived   0   1
## PClass Sex
## 1st    female            9 134
##        male            120  59
## 2nd    female           13  94
##        male            147  25
## 3rd    female          132  80
##        male            441  58
```

Agora, se queremos ver a as tabelas de contingências com os valores relativos das frequências precisamos utilizar as funções `ftable()` e `prop.table()` conjuntamente, mas adicionamos um `1` para o percentual na linha e um `2` para o percentual na coluna:

```
prop.table(ftable(Titanic$Sex,Titanic$Survived),1)*100
```

```
##                0        1
```

```
##
## female  33.33333 66.66667
## male    83.31375 16.68625
```

```
prop.table(ftable(Titanic$Sex,Titanic$Survived),2)*100
```

```
##                0        1
##
## female  17.84473 68.44444
## male    82.15527 31.55556
```

Novamente, `table()` e `ftable()` funcionam da mesma forma quando tratamos de duas variáveis, mas quando estamos trabalhando com mais de duas variáveis ftable() retorna uma tabela mais organizada:

```
prop.table(ftable(Titanic$Sex,Titanic$Survived,Titanic$PClass),1)*100
```

```
##                 1st       2nd       3rd
##
## female 0   5.844156  8.441558 85.714286
##        1  43.506494 30.519481 25.974026
## male   0  16.949153 20.762712 62.288136
##        1  41.549296 17.605634 40.845070
```

Agora, com a tabela `class_sex` que criamos anteriormente podemos fazer testes de independência utilizando as funções `chisq.test()` e `fisher.test()`, por exemplo. A função `summary()` também pode ser utilizada para o teste do chi-quadrado:

```
summary(class_sex)
```

```
## Number of cases in table: 1312
## Number of factors: 2
## Test for independence of all factors:
##  Chisq = 22.217, df = 2, p-value = 1.499e-05
```

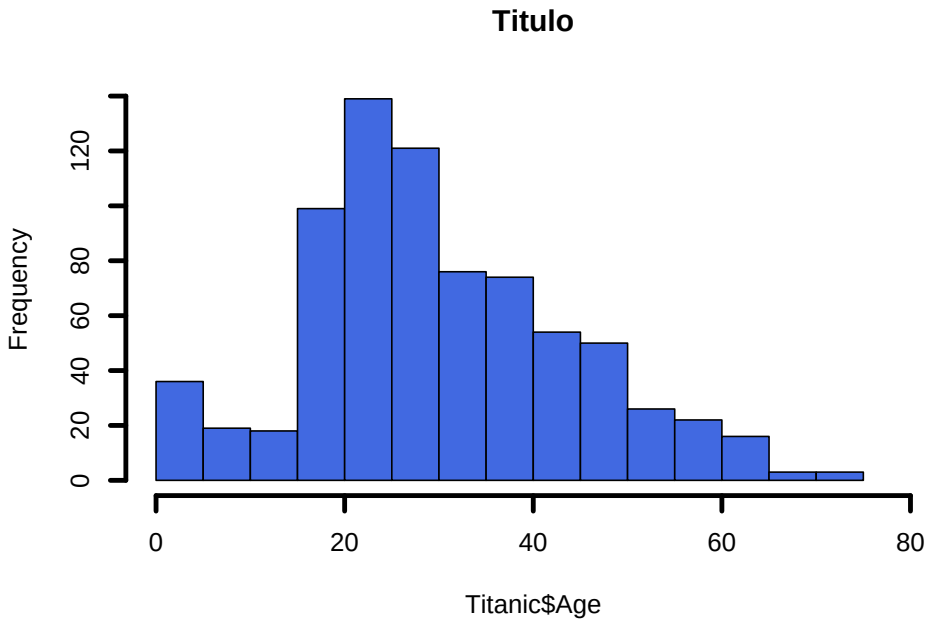
4.3 Gráficos base do R

Agora veremos rapidamente a construção de alguns gráficos do pacote base do R e algumas das personalizações possíveis. Iremos ver no capítulo {#ggplot} como fazer gráficos com o pacote `ggplot2`, mas por ora os gráficos base serão suficientes.

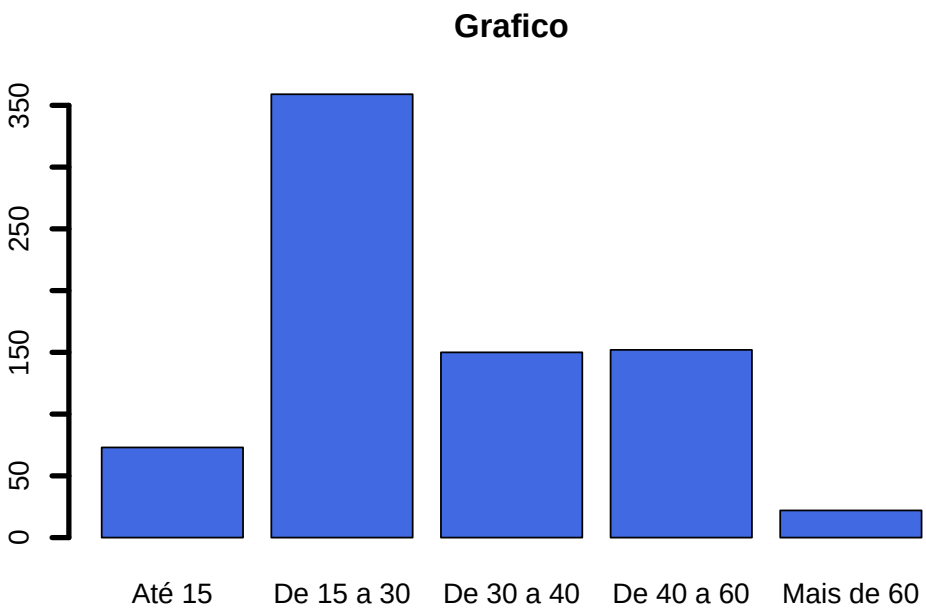
> No geral, todas as personalizações apresentadas nos gráficos são possíveis de serem feitas nos outros gráficos, respeitando os limites de cada tipo de gráficos. Há uma infinidade de adaptações possíveis para cada gráfico, para saber como realizá-las leia a documentação

referente a cada função de gráfico no `help` ou procure tutoriais e documentações na internet.

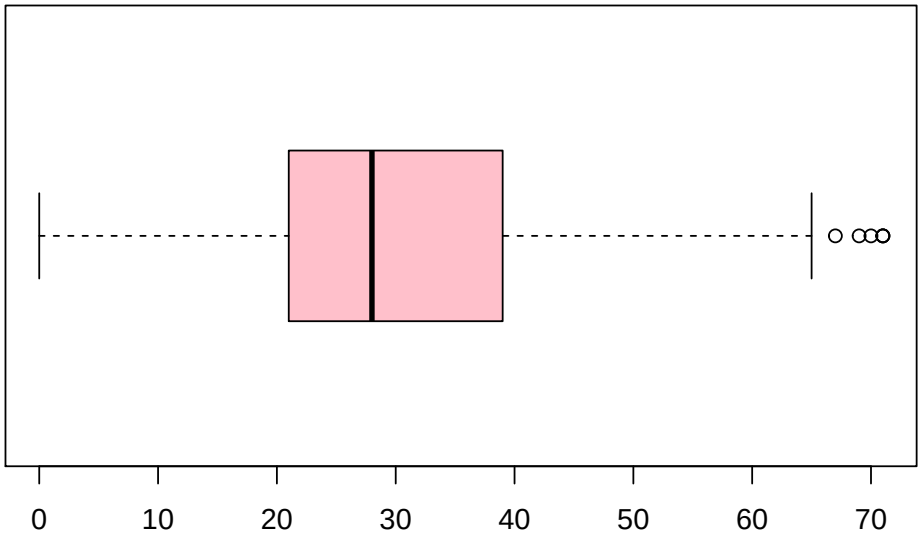
Para gerar histogramas utilizamos a função `hist()`, podemos delimitar a extensão dos eixos `x` e `y` com os argumentos `xlim` e `ylim`, dentro precisamos colocar um vetor com os limites desejados. Para adicionar um título para os gráficos definimos o argumento `main`. Ainda podemos definir a cor e espessura das linhas dos eixos com `col` e `lwd`:



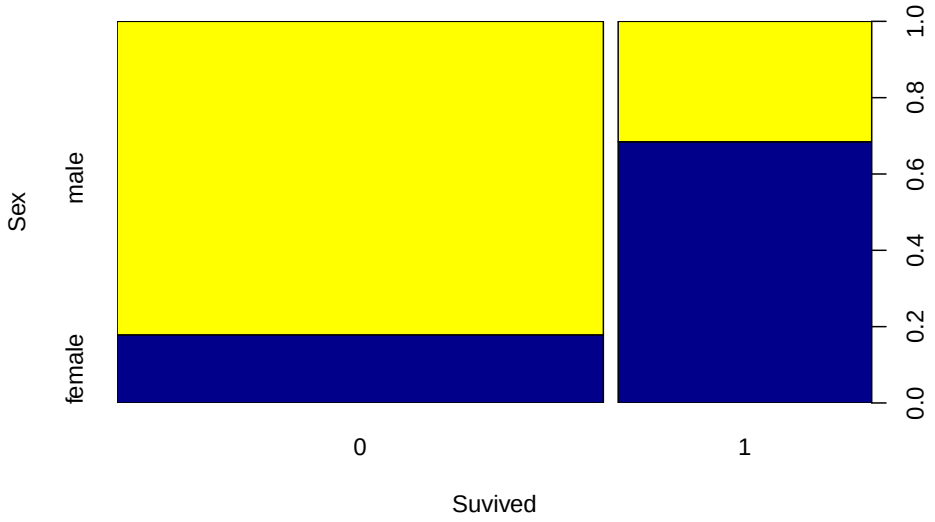
Para as variáveis categóricas geramos gráficos de barras com a função `barplot()` a partir da tabela de frequências da variável:



A função `boxplot()` possibilita a criação de boxplots. Podemos rotacionar os gráficos com o argumento `horizontal = TRUE`:

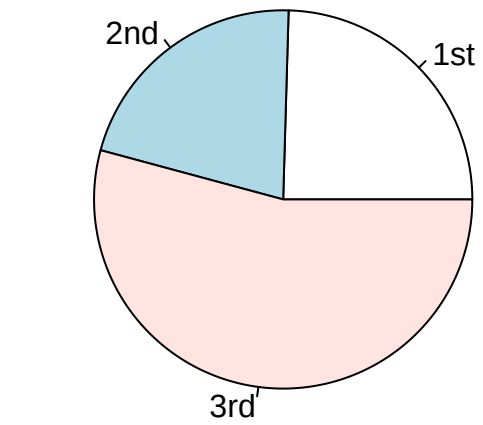


Podemos também realizar gráficos com duas variáveis categóricas combinadas. Com os argumentos `xlab` e `ylab` adicionamos os nomes dos eixos `x` e `y`:



A partir da frequência da variável `PClass` que atribuímos anteriormente podemos realizar um gráfico de pizza, com a função `pie()`:

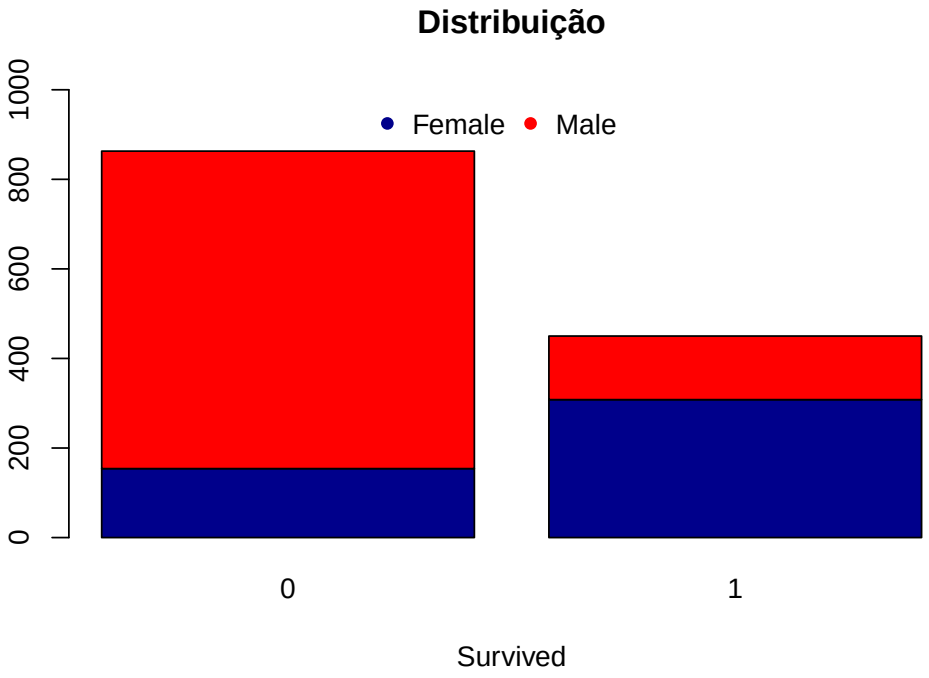
```
pie(freq_class, main = "Frequência das variável PClass")
```



A partir da tabela de contigência das variáveis `Sex` e `Survived` podemos criar o gráfico de barras empilhadas. Adicionamos ainda a função `legend()` para adicionar a legenda das cores da variável `Sex`. Com o argumento da legenda `horiz = TRUE` dizemos que queremos que a legenda esteja na horizontal (uma categoria ao lado da outra), o argumento `pch` informa o código do símbolo que

desejamos utilizar na legenda[1], `inset` informa onde queremos colocar a legenda em relação a margem e `bty` como queremos a caixa da legenda (no caso deixamos a caixa sem borda e sem preenchimento). Ainda, a localização da legenda pode ser: `bottomright`, `bottom`, `bottomleft`, `left`, `topleft`, `top`, `topright`, `right` ou `center`.

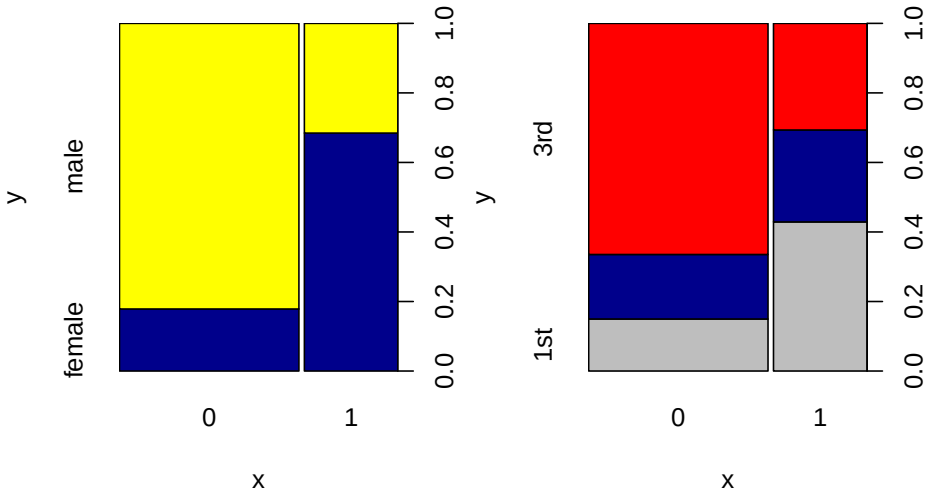
```
barplot(table(Titanic$Sex, Titanic$Survived), ylim = c(0,1000), xlab = "Survived", main = "Distri
legend("top", c("Female", "Male"), horiz = TRUE, col = c("darkblue", "red"), pch = 16, inset = c(
```



Por fim, podemos combinar diversos gráfico em uma mesma imagem. Para isso precisamos primeiro utilizar a função `par`, com o argumento `mfrow` informando a quantidade de linhas e colunas que queremos adicionar na nossa imagem. Aqui utilizaremos 1 linha com 2 colunas pois queremos 2 gráficos um do lado do outro. E o argumento `oma` informa o tamanho das margens externas da figura na ordem inferior, esquerda, superior e direita. Depois, nas linhas abaixo inserimos os gráficos desejados:

```
par(mfrow = c(1,2), oma = c(1,1,1,1))
plot(Titanic$Survived, Titanic$Sex, col = c("Dark Blue", "Yellow"))
plot(Titanic$Survived, Titanic$PClass, col = c("Gray", "Dark Blue", "Red"))
```

[1] Add Points to a Plot: https://www.rdocumentation.org/packages/graphics/versions/3.6.1/topics/points



4.4 Importação e exportação de bancos de dados

Os bancos de dados disponibilizados no R são muito úteis mas também precisamos importar nossos bancos de dados para análise. O R possui diversas funcionalidades para importação dos mais diversos tipos de bancos de dados (.csv, .xlsx, .sav, etc). Aqui iremos ver como importar e exportar bancos `.csv`.

Para tanto precisamos utilizar as funções `read.csv()` para importar e `write.csv()` para exportar. O formato padrão de importação é: read.csv("nomebanco.csv", na.strings = "", stringsAsFactors = FALSE, header = TRUE, encoding ="UTF-8")

Onde,

- `nomebanco` é o nome do arquivo que você deseja importar. Se o arquivo não está no diretório raiz do seu projeto, é necessário colocar todo o endereço (Atenção que no Windows é necessário inverter as barras do endereço). é necessário a extensão do arquivo.
- `na.strings` é a indicação para o R de qual string ele deve ler como NA. Se você não sabe como está configurado o seu banco utilize "" neste argumento. Assim o R irá ler apenas os locais em branco como NA
- `header` é a indicação de que o seu banco tem um cabeçalho. Assim, o R irá ler a primeira linha como nome das colunas. Se o seu banco não tem cabeçalho atribua `FALSE` a esse argumento
- `encoding` é a indicação da codificação do seu banco. Geralmente no Brasil utilizamos o `UTF-8`

  Atenção! Se o seu banco está comfigurado com `,` como padrão de separação para decimais dos números é necessário utilizar a função `read.csv2()`

Para exportar um banco já trabalhado no R utilize o seguinte comando:

write.csv(banco, file = "nomearquivo.csv", na = "NA")

Onde,

- `banco` é o nome do banco no R que você deseja exportar
- `file` é o nome do arquivo que você deseja criar. Será salvo automaticamente na pasta raiz do seu projeto a não ser que você indique um outro endereço. é necessário a extensão do arquivo.
- `na` é a indicação de como estão os NAs no seu banco.

  Atenção! Se você deseja que seu banco seja salvo no padrão de seperação de decimais com `,` é necessário utilizar a função `write.csv2()`

Chapter 5

Introdução ao Tidyverse

Os pacotes `tidyverse` são uma coleção de pacotes criados por Hadley Wickham e sua equipe (na RStudio)[1] para facilitar o tratamento e visualização dos dados. Aqui, iremos ver alguns desses pacotes nos próximos capítulos. Para darmos início aos pacotes `tidyverse` precisamos instalá-los. Para instalar todos os pacotes do `tidyverse` utilize o comando `install.packages("tidyverse")` ou apenas os pacotes desejados. Aqui iremos utilizar os pacotes `tidyr`, `dplyr`, `stringr` e `lubridate`. Depois de instalados chame os pacotes com a função `library()`.

Antes de começarmos vamos só entender o que os criadores do `tidyverse` consideram como `tidydata`. A ideia de um banco de dados "tidy" é um banco de dados que tem:

1 - Cada variável forma uma coluna; 2 - Cada observação forma uma linha; 3 - Cada tipo de observação forma uma tabela (ou banco de dados)

Parece um pouco óbvio isso, mas certamente você já viu bancos de dados assim:

Nome	Preto	Amarelo	Castanho	Outro
Joao	0	1	0	0
Luiz	1	0	0	0
Fabiana	0	0	1	0
Marcela	1	0	0	0

Esse mesmo banco em um formato `tidy` ficaria:

Nome	Cor de cabelo
Joao	Amarelo

[1] Tidy Data: https://vita.had.co.nz/papers/tidy-data.html

Nome	Cor de cabelo
Luiz	Preto
Fabiana	Castanho
Marcela	Preto

Aqui iremos apresentar algumas das ferramentas dos pacotes do `tidyverse` para transformar os bancos em `tidydata`.

Primeiramente, iremos utilizar o banco Países. Salve o arquivo `.csv` em seu computador e importe-o para o R sem considerar strings como factors e lendo as cédulas em branco como NA (no subcapítulo 4.4 há algumas instruções para importação de banco de dados).

Após importar o banco `Países` veja a estrutura e o resumo do banco.

```
summary(paises)
```

```
##     pais              codigo           continente           X1970
##  Length:249       Length:249       Length:249       Min.   :201.0
##  Class :character  Class :character  Class :character  1st Qu.:354.0
##  Mode  :character  Mode  :character  Mode  :character  Median :501.0
##                                                       Mean   :508.1
##                                                       3rd Qu.:668.0
##                                                       Max.   :794.0
##
##      X1971            X1972            X1973          X1974
##  Min.   :200.0   Min.   :200.0   Min.   :202   Min.   :201.0
##  1st Qu.:325.0   1st Qu.:350.0   1st Qu.:396   1st Qu.:338.5
##  Median :505.0   Median :495.0   Median :544   Median :481.0
##  Mean   :495.2   Mean   :496.7   Mean   :529   Mean   :495.9
##  3rd Qu.:642.0   3rd Qu.:644.0   3rd Qu.:658   3rd Qu.:656.5
##  Max.   :798.0   Max.   :800.0   Max.   :799   Max.   :800.0
##  NA's   :4                       NA's   :4     NA's   :1
##      X1975            X1976            X1977           X1978
##  Min.   :202.0   Min.   :200.0   Min.   :202.0   Min.   :200.0
##  1st Qu.:358.0   1st Qu.:339.0   1st Qu.:360.0   1st Qu.:338.0
##  Median :501.0   Median :512.0   Median :503.0   Median :492.0
##  Mean   :495.4   Mean   :497.5   Mean   :505.5   Mean   :491.5
##  3rd Qu.:631.5   3rd Qu.:641.0   3rd Qu.:667.0   3rd Qu.:637.0
##  Max.   :798.0   Max.   :796.0   Max.   :791.0   Max.   :800.0
##  NA's   :1                       NA's   :4       NA's   :2
##      X1979            X1980            X1981            X1982
##  Min.   :201.0   Min.   :207.0   Min.   : 200.0   Min.   :204.0
##  1st Qu.:335.8   1st Qu.:384.5   1st Qu.: 365.0   1st Qu.:369.0
##  Median :492.5   Median :530.0   Median : 509.0   Median :508.0
##  Mean   :487.6   Mean   :514.3   Mean   : 531.3   Mean   :511.5
```

```
##  3rd Qu.:626.8   3rd Qu.:645.0   3rd Qu.: 629.0   3rd Qu.:671.0
##  Max.   :789.0   Max.   :799.0   Max.   :7520.0   Max.   :800.0
##  NA's   :1       NA's   :1       NA's   :2
##       X1983           X1984           X1985           X1986
##  Min.   :201.0   Min.   : 200.0   Min.   :202.0   Min.   : 200.0
##  1st Qu.:354.0   1st Qu.: 345.0   1st Qu.:383.0   1st Qu.: 374.5
##  Median :487.0   Median : 490.0   Median :515.0   Median : 514.5
##  Mean   :494.8   Mean   : 519.5   Mean   :518.7   Mean   : 531.6
##  3rd Qu.:648.0   3rd Qu.: 651.0   3rd Qu.:674.0   3rd Qu.: 659.5
##  Max.   :800.0   Max.   :7000.0   Max.   :800.0   Max.   :5150.0
##                                                    NA's   :3
##       X1987           X1988           X1989           X1990
##  Min.   :200.0   Min.   : 200.0   Min.   :200.0   Min.   :201.0
##  1st Qu.:357.5   1st Qu.: 346.0   1st Qu.:351.2   1st Qu.:351.0
##  Median :485.0   Median : 498.0   Median :510.0   Median :501.5
##  Mean   :499.7   Mean   : 528.9   Mean   :508.4   Mean   :498.1
##  3rd Qu.:641.5   3rd Qu.: 656.2   3rd Qu.:680.5   3rd Qu.:635.8
##  Max.   :798.0   Max.   :7550.0   Max.   :800.0   Max.   :800.0
##  NA's   :2       NA's   :1        NA's   :1       NA's   :1
##       X1991           X1992           X1993           X1994
##  Min.   :202.0   Min.   :203.0   Min.   :201.0   Min.   :200.0
##  1st Qu.:356.5   1st Qu.:351.0   1st Qu.:364.5   1st Qu.:353.0
##  Median :520.5   Median :481.0   Median :518.0   Median :494.0
##  Mean   :512.2   Mean   :494.6   Mean   :507.3   Mean   :506.6
##  3rd Qu.:666.0   3rd Qu.:646.0   3rd Qu.:648.2   3rd Qu.:666.0
##  Max.   :794.0   Max.   :799.0   Max.   :800.0   Max.   :800.0
##  NA's   :3                       NA's   :1
##       X1995           X1996           X1997          X1998
##  Min.   :203.0   Min.   :201.0   Min.   :200   Min.   :203.0
##  1st Qu.:372.0   1st Qu.:338.0   1st Qu.:345   1st Qu.:335.5
##  Median :530.0   Median :470.0   Median :492   Median :508.0
##  Mean   :518.3   Mean   :484.1   Mean   :502   Mean   :503.3
##  3rd Qu.:668.0   3rd Qu.:627.0   3rd Qu.:669   3rd Qu.:673.0
##  Max.   :800.0   Max.   :799.0   Max.   :800   Max.   :799.0
##  NA's   :4       NA's   :1                     NA's   :2
##       X1999           X2000           date
##  Min.   :200.0   Min.   :201.0   Length:249
##  1st Qu.:356.5   1st Qu.:358.0   Class :character
##  Median :504.5   Median :476.0   Mode  :character
##  Mean   :505.0   Mean   :495.7
##  3rd Qu.:664.2   3rd Qu.:634.0
##  Max.   :797.0   Max.   :799.0
##  NA's   :3
```

5.1 Verificação de NAs

Primeiro iremos verificar se o R identificou se há `missings` (NAs) no nosso banco (lembre-se que importamos as cédulas em branco como NAs). Para isso iremos utilizar 3 funções. A função `is.na()` retorna `TRUE` (quando não é NA) e `FALSE` (quando é NA) para todos elementos do banco. Não é muito útil, mas poderemos utilizá-la combinada com outras funções. Combinando com a função `any()` descobrimos se há algum NA no nosso banco:

```
any(is.na(paises))
```

```
## [1] TRUE
```

Mas seria melhor se tivéssemos mais informações. Combinando `is.na()` com a função `sum()` descobrimos quantos NAs temos no nosso banco:

```
sum(is.na(paises))
```

```
## [1] 42
```

5.2 Criando tidydata com o pacote tidyr

O nosso banco países tem um grande problema: temos diversas variáveis com nomes de anos fazendo com que o nosso banco tenha muitas variáveis. Lembrando do exemplo do começo do capítulo o certo seria termos uma variável `ano` com o número do ano correspondente e uma varíavel `valor` com o valor correspondente do ano. Isso faria com que a quantidade de linhas do nosso banco seja multiplicada por 30 pois iremos repetir cada país 30 vezes (uma para cada ano).

Poderíamos fazer essa alteração na mão utilizando editores de planilhas ou, com linguagens de programação, usando laços lógicos e de repetição. Por sorte, o pacote `tidyr` nos ajuda nesse sentido. A função `gather()` faz exatamente o que queremos. O primeiro argumento da função `gather` é o banco que iremos transformar. Depois, o nome da coluna em que juntaremos todas as outras (as 30 colunas com informações dos anos, no nosso caso) e o nome da coluna que tem os valores correspondentes da coluna anterior (no caso criamos a coluna `valor`). Por fim, informamos quais são as colunas que não irão sofrer alteração com a seguinte sintaxe: `-c(coluna1, coluna2)`. A seguir fazemos a transformação atribuindo ao banco `pais_tidy`:

```
pais_tidy <- gather(paises, ano, valor, -c(pais, codigo, continente, date))
```

Veja como o nosso banco `pais_tidy` ficou:

```
summary(pais_tidy)
```

```
##      pais              codigo           continente
```

```
##  Length:7719        Length:7719        Length:7719
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##      date               ano                valor
##  Length:7719        Length:7719        Min.   : 200.0
##  Class :character   Class :character   1st Qu.: 354.0
##  Mode  :character   Mode  :character   Median : 503.0
##                                        Mean   : 506.1
##                                        3rd Qu.: 654.0
##                                        Max.   :7550.0
##                                        NA's   :42
```

Agora temos um banco `tidy`. Veja a diferença entre as dimensões do banco `paises` e do banco `pais_tidy`:

	paises	pais_tidy
n_rows	249	7719
n_cols	35	6

Bom, também podemos fazer o processo inverso de transformar o banco em um `tidydata`. No caso, revertiríamos ao que era originalmente o nosso banco `paises`. Para isso utilizamos a função `spread()`. Nela, informamos no primeiro argumento o banco que iremos retirar as nossas informações (no caso é o banco `pais_tidy`). Depois informamos qual a colunas que iremos separar e várias colunas (`ano` no nosso caso) e a coluna que contem os valores dessa coluna (`valor` no nosso caso):

```
pais_wide <- spread(pais_tidy, ano, valor)
```

Veja a primeira linha do nosso banco `pais_wide`:

```
##                                          pais codigo continente
## 1                                    Dominica    DM         NA
##     date X1970 X1971 X1972 X1973 X1974 X1975 X1976 X1977 X1978 X1979
## 1 23/05/17   769   501   597   212   307   258   529   550   206   280
##  X1980 X1981 X1982 X1983 X1984 X1985 X1986 X1987 X1988 X1989 X1990 X1991
## 1   221   578   503   686   374   416    NA   572   561   638   559   294
##   X1992 X1993 X1994 X1995 X1996 X1997 X1998 X1999 X2000
## 1   777   579   634   560   583   783   674   667   521
```

Outras funcionalidades do `tidyr` é criar bancos unindo e separando colunas de um banco já existente. Por exemplo, se eu quero um banco que tenha as colunas `pais` e `codigo` do meu banco `paises` unidas em uma coluna só posso unir essas colunas utilizando a função `unite()`. Informo no primeiro argumento o nome do meu banco original, depois o nome da coluna nova (a união das colunas) e as

colunas que quero unir na sequência desejada. Depois informo com qual símbolo devo separar as informações na nova coluna.

```
pais_unido <- unite(paises, pais_cod, pais, codigo, sep = "/")
```

A primeira linha do meu banco:

```
##                     pais_cod continente X1970 X1971 X1972 X1973
## 1               Afganistán/ AF         AS   335   263   366   757
##  X1974 X1975 X1976 X1977 X1978 X1979 X1980 X1981 X1982 X1983 X1984 X1985
## 1  603   266   571   336   480   402   552   526   228   463   211   593
##  X1986 X1987 X1988 X1989 X1990 X1991 X1992 X1993 X1994 X1995 X1996 X1997
## 1  568   346   495   341   212   399   364   333   340   460   374   674
##   X1998 X1999 X2000     date
## 1   490   665   443 07/09/14
```

Para fazer o processo contrário da função `unite()` utilizo a função `separate()`. Primeiro informo qual banco será separado, depois o nome da coluna a ser separada (argumento `col =`), depois o nome das colunas a serem criadas no argumento `into = c()` e o caracter na qual a função irá separar a coluna no argumento `sep =` (no caso é o símbolo `/`):

```
pais_sep <- separate(pais_unido, col = pais_cod, into = c("pais", "cod"), sep = "/")
```

A primeira linha do nosso banco:

```
##                     pais  cod continente X1970 X1971 X1972 X1973
## 1               Afganistán  AF         AS   335   263   366   757
##  X1974 X1975 X1976 X1977 X1978 X1979 X1980 X1981 X1982 X1983 X1984 X1985
## 1  603   266   571   336   480   402   552   526   228   463   211   593
##  X1986 X1987 X1988 X1989 X1990 X1991 X1992 X1993 X1994 X1995 X1996 X1997
## 1  568   346   495   341   212   399   364   333   340   460   374   674
##   X1998 X1999 X2000     date
## 1   490   665   443 07/09/14
```

5.3 Manipulação de strings

O pacote `stringr` tem várias ferramentas para nos ajudar na manipulação de strings. O nosso banco `paises` veio com vários erros na coluna `pais`. Alguns nomes de países tem espaços antes e/ou depois do nome do país. A função `str_trim()` ajuda a retirar esses espaços de forma automatizada. Atribuímos o resultado da função ao próprio banco.

> Preste muita atenção ao realizar funções sobrescrevendo o banco original

```
pais_tidy$pais <- str_trim(pais_tidy$pais)
```

Veja que a função limpou os nomes dos países no banco `pais_tidy`.

Exercício 5.1. Realize a limpeza dos nomes nos outros bancos criados a partir do banco `paises`.

Uma outra função útil é a `str_pad()`, que adiciona strings a uma coluna de um banco de dados. Por exemplo, se quisermos adicionar o caracter `X` depois das siglas do continente na coluna `continente` do banco `pais_unido`. Atribuímos a coluna do banco desejado a função `str_pad()` com o primeiro argumento a referência da coluna desejada, depois no argumento `width =` informamos qual vai ser o tamanho dessa nova `string` (no caso as siglas já tinham o tamanho 2, com a adição de um novo caracter era irá passar a ter tamanho 3), o argumento `side =` informa onde a string irá (left ou right) e o argumento `pad =` informa qual a sequência de caracteres que iremos adicionar:

```
pais_unido$continente <- str_pad(pais_unido$continente, width = 3, side = "right", pad = "X")
```

Resultando em:

```
##                      pais_cod continente X1970 X1971 X1972 X1973
## 1               Afganistán/ AF        ASX   335   263   366   757
## 2  Albania                / AL        EUX   516   270   403   450
##  X1974 X1975 X1976 X1977 X1978 X1979 X1980 X1981 X1982 X1983 X1984 X1985
## 1   603   266   571   336   480   402   552   526   228   463   211   593
## 2   236   314   668   412   549   468   729   297   706   729   401   364
##  X1986 X1987 X1988 X1989 X1990 X1991 X1992 X1993 X1994 X1995 X1996 X1997
## 1   568   346   495   341   212   399   364   333   340   460   374   674
## 2   263   628   709   729   352   506   253   705   278   515   645   689
##   X1998 X1999 X2000     date
## 1   490   665   443 07/09/14
## 2   676   439   795 08/09/14
```

Outra funcionalidade é detectar onde em um banco uma série de caracteres se encontra com a função `str_detect()`:

```
str_detect(pais_sep$continente, "NA")
```

```
##   [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
##  [12] FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE  TRUE
##  [23] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
##  [34] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [45] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE
##  [56]  TRUE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
##  [67] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##  [78]  TRUE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
##  [89] FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [100] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
## [111] FALSE FALSE FALSE FALSE  TRUE FALSE  TRUE  TRUE FALSE FALSE  TRUE
## [122] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
## [133] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [144] FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
## [155] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
## [166] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE
## [177] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE
## [188] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE
## [199]  TRUE  TRUE  TRUE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
## [210] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [221] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [232] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [243] FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Podemos também substituir uma série de caracteres com a função `str_replace()`. Por exemplo, substituíremos na coluna `continente` do banco `pais_sep` todas os lugares que forem `NA` por `North America`:

```
pais_sep$continente <- str_replace(pais_sep$continente, "NA", "North America")
```

Por fim, precisamos retirar da coluna `ano` em `pais_tidy` a string `X` que foi herdada do nome das colunas de `pais`. Para isso utilizamos a função `str_remove()`:

```
pais_tidy$ano <- str_remove(pais_tidy$ano, "X")
```

> A função `str_remove` remove apenas o primeiro correspondente que ela encontra. Se a coluna tivesse mais de um "X" e quiséssemos remover todos os "X" da coluna deveríamos utilizar a função `str_remove_all()`.

Por fim, nas funções `base` do R temos duas funções que nos são úteis para tratar com strings: `tolower()` e `toupper()`. A primeira faz com que todos os caracteres sejam transformados em caixa baixa e a segunda em caixa alta.

Veja a função `tolower()` na coluna `continente` do banco `pais_wide`:

```
pais_wide$continente <- tolower(pais_wide$continente)
head(pais_wide)
```

```
##                                       pais codigo continente
## 1                                 Dominica     DM         na
## 2                             Burkina Faso     BF         af
## 3                                   Brasil     BR         sa
## 4                                Guadalupe     GP         na
## 5                  Emiratos Árabes Unidos     AE         as
## 6                              Afganistán     AF         as
##     date X1970 X1971 X1972 X1973 X1974 X1975 X1976 X1977 X1978 X1979
## 1 23/05/17   769   501   597   212   307   258   529   550   206   280
## 2 19/09/14   382   527   350   694   771   671   260   489   316   227
## 3 12/06/17   426   239   616   532   467   439   361   242   650   493
## 4 15/10/15   726   467   399   534   589   542   482   722   533   414
```

```
## 5 19/05/17   700   330   607   593   670   780   417   718   628   498
## 6 07/09/14   335   263   366   757   603   266   571   336   480   402
##   X1980 X1981 X1982 X1983 X1984 X1985 X1986 X1987 X1988 X1989 X1990 X1991
## 1   221   578   503   686   374   416    NA   572   561   638   559   294
## 2   234   612   754   735   599   661   771   319   291   208   471   385
## 3   615   286   273   661   781   654   425   776   317   582   464   288
## 4   435   224   572   745   239   242   657   684   429   463   520   367
## 5   662   462   752   212   374   677   651   720   676   333   591   566
## 6   552   526   228   463   211   593   568   346   495   341   212   399
##    X1992 X1993 X1994 X1995 X1996 X1997 X1998 X1999 X2000
## 1    777    579    634    560    583    783    674    667    521
## 2    623    513    730    764    413    480    247    507    488
## 3    583    538    537    456    784    694    774    343    356
## 4    471    672    718    651    475    788    534    522    391
## 5    612    315    529    699    296    471    215    745    365
## 6    364    333    340    460    374    674    490    665    443
```

E a função `toupper()` na coluna `pais` do banco `pais_sep`:

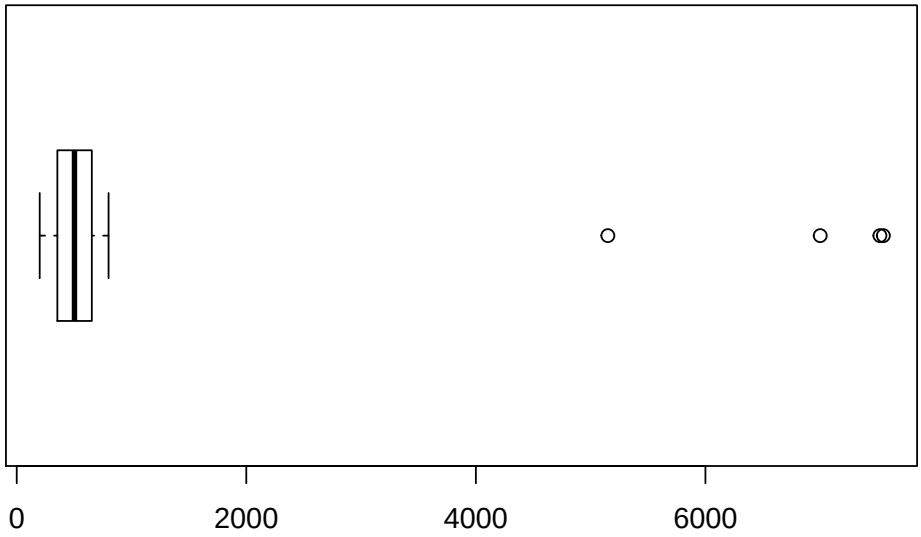
```
pais_sep$pais <- toupper(pais_sep$pais)
head(pais_sep)
```

```
##                    pais  cod   continente X1970 X1971 X1972 X1973
## 1               AFGANISTÁN  AF           AS   335   263   366   757
## 2  ALBANIA                  AL           EU   516   270   403   450
## 3                 ALEMANIA  DE           EU   505   309   787   635
## 4                  ANDORRA  AD           EU   749   400   636   414
## 5                   ANGOLA  AO           AF   310    NA   399   594
## 6 ANGUILA                   AI  North America  348   254   613   621
##   X1974 X1975 X1976 X1977 X1978 X1979 X1980 X1981 X1982 X1983 X1984 X1985
## 1   603   266   571   336   480   402   552   526   228   463   211   593
## 2   236   314   668   412   549   468   729   297   706   729   401   364
## 3   566   298   748   657   263   612   464   605   736   387   764   278
## 4   264   556   766   362   645   427   600   258   516   355   410   375
## 5   273   441   519   597   794   436   553   592   791   255   763   551
## 6   481   254   645   388   279   589   579   800   502   389   359   434
##   X1986 X1987 X1988 X1989 X1990 X1991 X1992 X1993 X1994 X1995 X1996 X1997
## 1   568   346   495   341   212   399   364   333   340   460   374   674
## 2   263   628   709   729   352   506   253   705   278   515   645   689
## 3   253   359   389   714   339   707   631   648   780   637   286   311
## 4   681   218   203   222   676   251   671   784   796   225   439   460
## 5   770   399   422   235   223   622   502   491   458   292   741   249
## 6   428   389   768   580   782   792   770   511   670   672   364   216
##    X1998 X1999 X2000     date
## 1    490    665    443 07/09/14
## 2    676    439    795 08/09/14
## 3    769    533    787 09/09/14
```

```
## 4   795   446   256 10/09/14
## 5   269   399   526 11/09/14
## 6   260   461   477 12/09/14
```

5.4 Identificando erros no banco

Alguns dos valores do nosso banco estão com erros (provavelmente por erro de digitação). Vamos dar uma olhada no `boxplot` da coluna valor:



Os valores das colunas do banco `paises` foram gerados entre 200 e 800. Portanto, a partir do boxplot podemos ver que há algo errado com o nosso banco. Vamos dar uma olhada no resumo das variáveis:

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##   200.0   354.0   503.0   506.1   654.0  7550.0      42
```

> O banco `paises` foi criado para ilustração dessa apostila. Ainda que pareça muito simples o exemplo podemos ter casos na realidade de situações em que idade de indivíduos, por exemplo, aparece como 200 no lugar de 20.

Vamos verificar quais são esses elementos:

```
subset(pais_tidy, pais_tidy$valor > 800)
```

```
##                              pais codigo continente    date  ano
## 2751                      Armenia    AM         AS 14/06/17 1981
## 3512 Bolivia, Estado Plurinacional de    BO         SA 03/06/17 1984
## 4004                       Baréin    BH         AS 02/09/14 1986
```

```
## 4495                          Aruba    AW         NA 15/06/17 1988
##      valor
## 2751  7520
## 3512  7000
## 4004  5150
## 4495  7550
```

Bom, pelo que podemos ver aparentemente foi um erro de digitação. Todos os casos tem um `0` a mais. Como estamos tratando de números e é simplesmente um `0` no final, vamos dividir todos os casos por `10`. Primeiro criamos um vetor com um endereço dos elementos que correspondem a busca (`pais_tidy$valor > 800`). Depois, substituímos esses valores por eles mesmos divididos por 10:

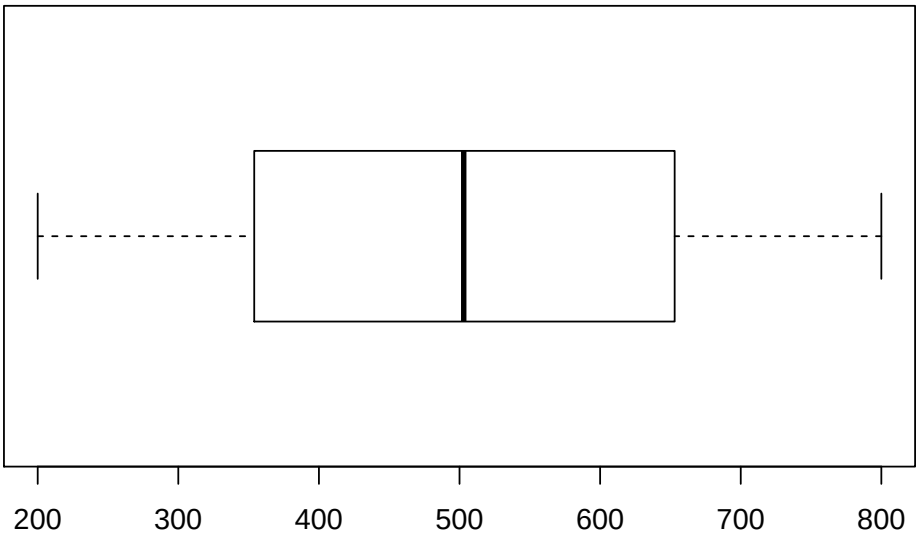
```
replace <- which(pais_tidy$valor > 800)
pais_tidy$valor[c(replace)] <- pais_tidy$valor[c(replace)]/10
```

Vamos ver como eles ficaram?

```
pais_tidy[c(replace), ]
```

```
##                               pais codigo continente    date  ano
## 2751                       Armenia    AM         AS 14/06/17 1981
## 3512 Bolivia, Estado Plurinacional de   BO         SA 03/06/17 1984
## 4004                        Baréin    BH         AS 02/09/14 1986
## 4495                         Aruba    AW         NA 15/06/17 1988
##      valor
## 2751   752
## 3512   700
## 4004   515
## 4495   755
```

E como será que o boxplot estará agora?



Pronto! Eliminamos todos os erros da variável `ano`.

5.5 Trabalhando com data e hora

O `tidyverse` também possui ótimas ferramentas para trabalharmos com datas e hora. O pacote `lubridate` agrega essas ferramentas. A coluna `date` do nosso banco `pais_tidy` não está sendo lida pelo R como uma data, mas sim como um conjunto de caracteres:

```
class(pais_tidy$date)
```

```
## [1] "character"
```

Bom, isso é um problema para análises futuras. Vamos então indicar para o R qual formato está a data do nosso banco. Se você der uma olhada no banco `pais_tidy` verá que as datas estão da seguinte maneira: dd/mm/aa (em português). Para o R entender que isso é um formato de data precisamos indicar para ele. Assim, utilizamos as funções de formato de data do pacote `lubridate`. No caso, a função será `dmy` indicando que o formato é `day`, `month` e `year`:

```
pais_tidy$date <- dmy(pais_tidy$date)
```

Veja agora como está a estrutura da nossa variável `date`:

```
str(pais_tidy)
```

```
## 'data.frame':    7719 obs. of  6 variables:
##  $ pais     : chr  "Afganistán" "Albania" "Alemania" "Andorra" ...
##  $ codigo    : chr  " AF " " AL " " DE " " AD " ...
```

```
##  $ continente: chr  "AS" "EU" "EU" "EU" ...
##  $ date      : Date, format: "2014-09-07" "2014-09-08" ...
##  $ ano       : chr  "1970" "1970" "1970" "1970" ...
##  $ valor     : num  335 516 505 749 310 348 249 378 325 329 ...
```

Agora as datas foram convertidas para o formato que o R entende como data:

```
class(pais_tidy$date)
```

```
## [1] "Date"
```

Veja o resumo das nossas variáveis:

```
summary(pais_tidy)
```

```
##      pais              codigo           continente
##  Length:7719        Length:7719        Length:7719
##  Class :character   Class :character   Class :character
##  Mode  :character   Mode  :character   Mode  :character
##
##
##
##
##       date                ano                valor
##  Min.   :2014-08-05   Length:7719        Min.   :200.0
##  1st Qu.:2014-10-06   Class :character   1st Qu.:354.0
##  Median :2017-04-28   Mode  :character   Median :503.0
##  Mean   :2016-05-21                      Mean   :502.9
##  3rd Qu.:2017-06-29                      3rd Qu.:653.0
##  Max.   :2018-05-08                      Max.   :800.0
##                                          NA's   :42
```

Podemos também informar outros formatos de datas. Utilizando a mesma função mas com a abreviação do mês:

```
dmy("17 Nov 2015")
```

```
## [1] "2015-11-17"
```

Utilizando o formato com as horas, minutos e segundos:

```
mdy_hms("July 15, 2012 12:56:09")
```

```
## [1] "2012-07-15 12:56:09 UTC"
```

> Trabalhar com datas é bem mais complicado do que foi demonstrado aqui. Para mais informações veja o capítulo sobre datas e hora do livro "R for Data Science"

5.6 Ajustes finais

Depois de todas essas alterações, para o nosso banco ficar pronto para análise vamos converter as variáveis `pais`, `codigo`, `continente` e `ano`. As três primeiras iremos transformar em `factor`:

```
pais_tidy$pais <- factor(pais_tidy$pais)
pais_tidy$codigo <- factor(pais_tidy$codigo)
pais_tidy$continente <- factor(pais_tidy$continente)
```

Por fim, vamos criar uma variável `ano2` (cópia de `ano`) que será convertida em `factor` e vamos converter a variável `ano` em `numeric`:

```
pais_tidy$ano2 <- pais_tidy$ano
pais_tidy$ano2 <- factor(pais_tidy$ano2)
pais_tidy$ano <- as.numeric(pais_tidy$ano)
```

Agora o nosso banco estará assim:

```
##          pais          codigo     continente      date
##  Afganistán:  31    AD     :  31   AF:1798    Min.   :2014-08-05
##  Albania   :  31    AE     :  31   AN: 155    1st Qu.:2014-10-06
##  Alemania  :  31    AF     :  31   AS:1643    Median :2017-04-28
##  Andorra   :  31    AG     :  31   EU:1612    Mean   :2016-05-21
##  Angola    :  31    AI     :  31   NA:1271    3rd Qu.:2017-06-29
##  Anguila   :  31    AL     :  31   OC: 806    Max.   :2018-05-08
##  (Other)   :7533   (Other):7533   SA: 434
##       ano            valor            ano2
##  Min.   :1970   Min.   :200.0   1970   : 249
##  1st Qu.:1977   1st Qu.:354.0   1971   : 249
##  Median :1985   Median :503.0   1972   : 249
##  Mean   :1985   Mean   :502.9   1973   : 249
##  3rd Qu.:1993   3rd Qu.:653.0   1974   : 249
##  Max.   :2000   Max.   :800.0   1975   : 249
##                 NA's   :42      (Other):6225
```

Chapter 6

Gráficos com ggplot2

Chapter 7

Relatórios com Markdown