Introdução ao R

Iara Passos

2019-11-11

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

Aula 1

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 usar um pacote você precisa primeiro instalá-lo e quando for utilizá-lo precisa "chamar" ele no código, mas isso veremos mais para frente. As IDEs possibilitam que a busca e a instalação desses pacotes tambem seja mais prática.

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.

2.7 Vetores

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

Exercício 2.5. 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 2.6. Multiplique o seu vetor `numeric` por um número.

O que aconteceu? Ele mudou o seu vetor original?

Exercício 2.7. 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 2.8. 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.

2.7.1 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.

2.7.2 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 2.9. 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()`[6] (olhe a documentação da função).

Exercício 2.10. Faça a soma de cada um dos vetores de sacolas utilizando a função `sum()`. Armazene cada resultado em uma nova variável.

Exercício 2.11. Descubra o total de itens comprados na feira utilizando a função `sum()`. Armazene cada resultado em uma nova variável.

2.7.3 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).

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

Exercício 2.12. 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 2.13. Selecione do vetor `sacola3` os valores 1 e 3 e do vetor sacola2 os valores de 2 a 4.

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

2.7.4 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 2.15. 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 2.16. 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 2.17. 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 2.18. Armazene em uma variável os valores do vetor `sacola1` que são menores do que 10.

2.7.5 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 2.19. 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.

2.7.6 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 2.20. 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

2.8 Listas de exercícios

Faça as listas de exercícios referente a essa aula no repositório do curso.

Chapter 3

Aula 2

3.1 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`.

3.1.1 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 3.1. 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 3.2. Selecione a 5ª linha da `matriz_sacola`.

Exercício 3.3. Selecione a 3ª coluna da `matriz_sacola`.

3.1.2 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 3.4. Some duas matrizes do mesmo tamanho. Atribua o resultado a uma terceira matriz.

Exercício 3.5. Some uma matriz com um vetor. Atribua o resultado a uma terceira matriz.

3.2 Factors

Factor[1] 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:

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

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

> 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.

3.3 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 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
                    sample(seq(1000, 30000, by = 1000), 30, replace = T),
                    factor(rep(c("solteiro","casado", "viuvo", "separado"), c(10, 10, 1
colnames(banco) <- c("individuo", "raca", "sexo", "idade", "escol", "renda", "civil")
banco
```

```
##    individuo raca sexo idade       escol renda    civil
## 1      00001    b    f    31    superior 14000 solteiro
## 2      00002    b    f    32    superior 27000 solteiro
## 3      00003    b    f    33    superior  5000 solteiro
## 4      00004    b    f    37    superior 22000 solteiro
## 5      00005    b    f    39    superior  1000 solteiro
## 6      00006    b    f    34     tecnico 29000 solteiro
## 7      00007    b    f    33     tecnico  5000 solteiro
## 8      00008    b    f    16     tecnico 22000 solteiro
## 9      00009    b    f    50     tecnico 13000 solteiro
## 10    000010    b    f    57     tecnico 28000 solteiro
## 11    000011    n    f    49     tecnico 14000   casado
## 12    000012    n    f    18     tecnico 12000   casado
## 13    000013    n    f    17     tecnico 27000   casado
## 14    000014    n    f    46       medio  4000   casado
## 15    000015    n    f    44       medio 20000   casado
## 16    000016    n    f    60       medio 14000   casado
## 17    000017    n    m    28       medio 23000   casado
## 18    000018    n    m    53       medio 28000   casado
## 19    000019    n    m    34       medio  5000   casado
## 20    000020    n    m    24       medio 10000   casado
```

```
## 21    000021    i    m    20       medio 22000    viuvo
## 22    000022    i    m    36       medio 15000    viuvo
## 23    000023    i    m    57       medio 23000 separado
## 24    000024    i    m    16       medio 23000 separado
## 25    000025    i    m    51       medio  6000 separado
## 26    000026    i    m    16 fundamental  4000 separado
## 27    000027    o    m    51 fundamental 11000 separado
## 28    000028    o    m    30 fundamental 25000 separado
## 29    000029    o    m    26 fundamental 21000 separado
## 30    000030    o    m    59 fundamental  9000 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.

3.3.1 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 3.6. 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  31 32 33 37 39 34 33 16 50 57 ...
## $ escol   : Factor w/ 4 levels "fundamental",..: 3 3 3 3 3 4 4 4 4 4 ...
## $ renda   : num  14000 27000 5000 22000 1000 29000 5000 22000 13000 28000 ...
## $ 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.   :16.00   fundamental: 5
##  000010 : 1   i: 6   m:14   1st Qu.:26.50   medio      :12
##  000011 : 1   n:10          Median :34.00   superior   : 5
##  000012 : 1   o: 4          Mean   :36.57   tecnico    : 8
##  000013 : 1                 3rd Qu.:49.75
##  000014 : 1                 Max.   :60.00
##  (Other):24
##      renda            civil
##  Min.   : 1000   casado  :10
##  1st Qu.: 9250   separado: 8
##  Median :14500   solteiro:10
##  Mean   :16067   viuvo   : 2
##  3rd Qu.:23000
##  Max.   :29000
##
```

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.

3.3.2 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 3.7. Selecione o elemento que está armazenado na linha 20 e na coluna 4.

Exercício 3.8. 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] 13000 28000 14000 12000 27000  4000 20000 14000 23000 28000  5000
## [12] 10000
```

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    33 superior
## 4    f    37 superior
## 8    f    16  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    28       medio 23000   casado
## 18    000018    n    m    53       medio 28000   casado
## 19    000019    n    m    34       medio  5000   casado
## 20    000020    n    m    24       medio 10000   casado
## 21    000021    i    m    20       medio 22000    viuvo
## 22    000022    i    m    36       medio 15000    viuvo
## 23    000023    i    m    57       medio 23000 separado
## 24    000024    i    m    16       medio 23000 separado
## 25    000025    i    m    51       medio  6000 separado
## 26    000026    i    m    16 fundamental  4000 separado
## 27    000027    o    m    51 fundamental 11000 separado
## 28    000028    o    m    30 fundamental 25000 separado
## 29    000029    o    m    26 fundamental 21000 separado
## 30    000030    o    m    59 fundamental  9000 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 3.9. 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 15 16 17 18 20 21 22 23 24 25 27 28 29 30
```

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.

3.4 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

Aula 3

Chapter 5

Aula 4

Chapter 6

Aula 5