Introdução ao R

Prof. Dr. Anderson C. Carniel e Prof. Dr. Francisco C. M. Souza

2020-05-15

1 O que é R?

De acordo com a documentação R (veja como acessar a documentação logo mais), R é um sistema para computação estatística e geração de gráficos para este fim. Dentro desse sistema, existe a linguagem de programação R que tem as seguintes características principais:

- R foi inicialmente escrito por **Robert Gentleman** e **Ross Ihaka** do Departamento de Estatística da *University of Auckland*. Atualmente, R é mantido por diversos colaboradores. (https://www.r-project.org/contributors.html)
- R é principalmente implementado em C e Fortran.
- R é baseado no dialeto do **S** (uma linguagem para análise de dados estatísticos - **S** é derivado da palavra *Statistical*), uma linguagem que foi desenvolvida nos anos 80 na Bell Labs (https://web.archive.org/web/20150723044213/http://www2.research.att.com/areas/stat/doc/94.11.ps). Seu principal desenvolvedor, o **John M. Chambers**, recebeu o *ACM Software Systems Award* por seu trabalho com o **S**.
- R possui uma sintaxe superficialmente similar à da linguagem de programação C e com semântica do **paradigma de programação funcional** (*functional programming language*) com afinidades com as linguagens Lisp e APL. Dessa forma, os dados são essencialmente manipulados por meio de **funções**.
- R é uma linguagem sensível à maiúsculas e minúsculas (*case-sensitive*)

- R armazena e manipula os dados em memória principal (RAM), assim a manipulação de grandes volumes de dados pode exigir bastante memória e processamento
- R manipula os dados em estruturas de dados bem definidas, que são chamados, de uma maneira geral, como **objetos** (**objects**). Exemplos de termos são:
    - *NULL* : significa a falta/ausência de um objeto
    - *symbol* : significa o nome de uma **variável**
    - *list* : significa uma lista
- R utiliza seus tipos de dados principalmente usando o conceito de vetores (*vector*) - conforme discutido aqui
- R é disponibilizado sob a GNU General Public License (https://www.gnu.org/licenses/gpl-3.0.en.html) e pode ser instalado em vários sistemas a partir de seu CRAN Mirrors (https://cran.r-project.org/mirrors.html)
- R está sendo cada vez mais popular, tal como indicado pelo TIOBE (https://www.tiobe.com/tiobe-index/). No momento da escrita desse documento e de acordo com esse ranking, R é a décima linguagem mais popular, tendo um crescimento bastante elevado se comparado à sua posição em 2019.

2 O que é o RStudio?

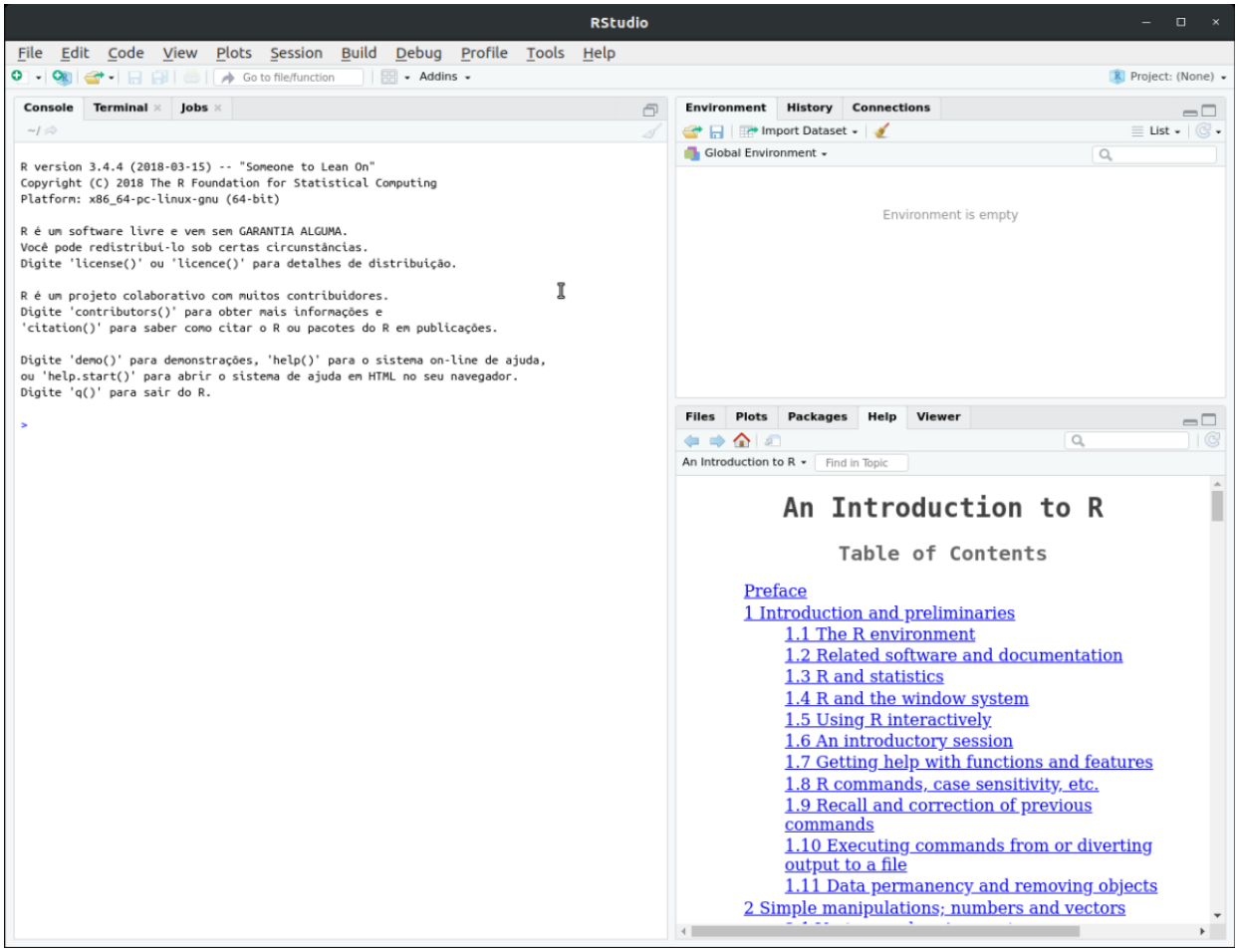
é um **Ambiente de Desenvolvimento Integrado** (do inglês *Integrated Development Environment* - **IDE**) para programação utilizando o R (além de outras linguagens, como Python (https://rstudio.github.io/reticulate/index.html)).

O RStudio (https://rstudio.com/products/rstudio/download/) possui diversas vantagens:

- Fácil instalação em Windows, Linux e Mac
- Possui versões para uso livre (sob licença AGPL v3 (https://pt.wikipedia.org/wiki/GNU_Affero_General_Public_License))
- Possui versões profissionais pagas com suporte especializado
- Possui múltiplos painéis para programação, execução de códigos e visualização de dados - **Essa característica em especial facilita a execução de projetos de Ciência de Dados!**

2.1 A interface do RStudio

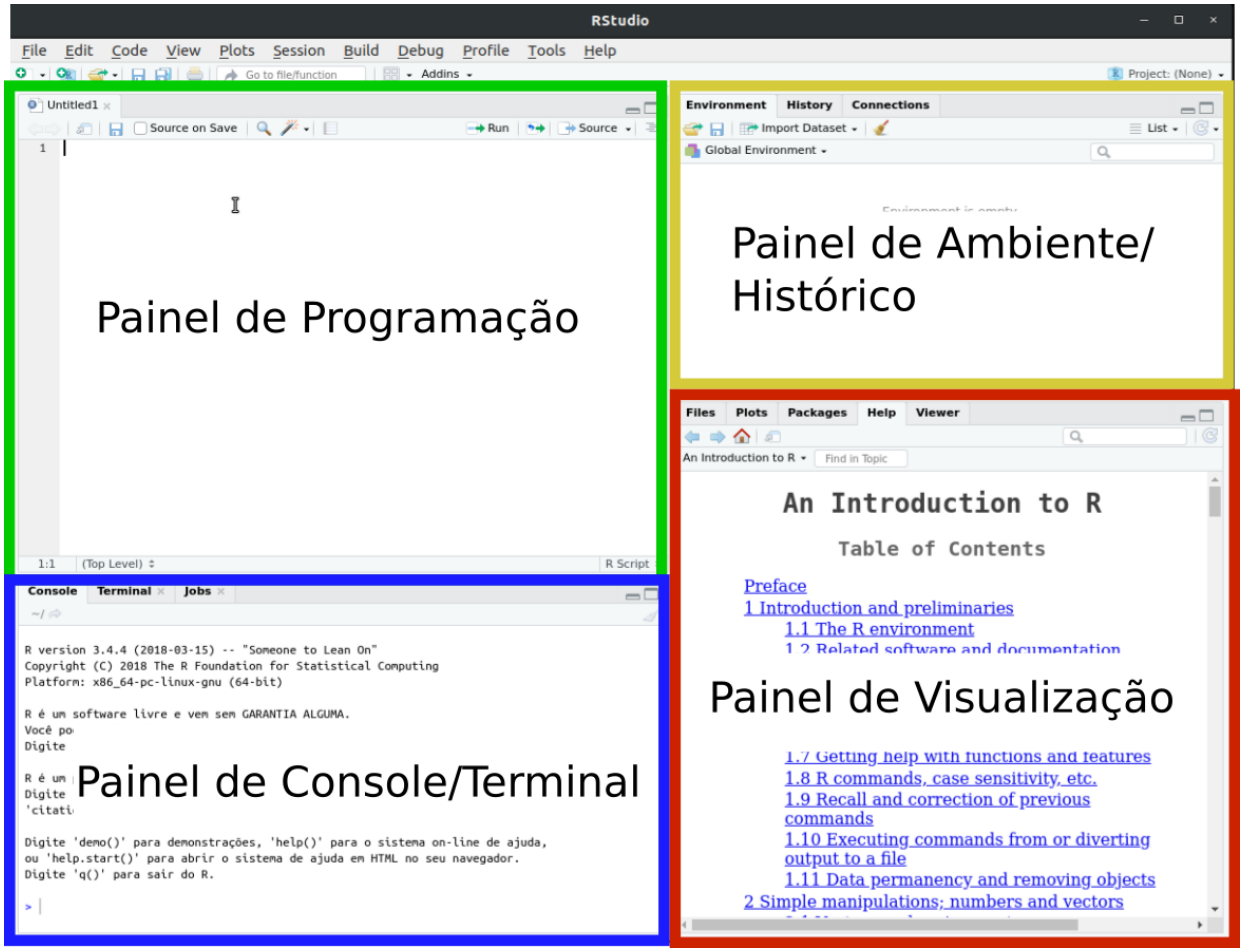
Após instalação do RStudio, sua tela inicial é algo similar à tela a seguir (tal como a da nossa máquina virtual disponibilizada no curso (https://drive.google.com/file/d/1AQ-iYifhNcxFCeln3yfssDGjzxZXyJw2/view)):



Tela Inicial do RStudio

Para começar um programa em R, clique em *File*, então *New File* e por fim, *R Script* (ou utilizando o atalho `Ctrl+Shift+N`).

Assim, mais um painel é adicionado no RStudio, totalizando 4 painéis, conforme descrito a seguir:



Organização dos painéis do RStudio

- **Painel de Programação:** Aqui é onde escrevemos nosso código. Note que o RStudio possui diversas funcionalidades que nos ajudam nessa escrita, como o autocompletar e a descrição de funções sendo usadas. O código, de uma maneira completa, pode ser executado através do Botão “Run”. Opcionalmente, é possível executar trechos de código apenas. Para isso, selecione o trecho de código a ser executado e posteriormente clique no Botão “Run”.
- **Painel de Console/Terminal:** Aqui é onde o código é de fato executado pelo R e acompanhar as saídas no terminal de linha de comando. Você pode também executar códigos em R diretamente no Console, isso é útil para executar códigos bem pontuais que não precisam estar no script de fato.
- **Painel de Ambiente/Histórico:** Aqui é onde podemos ver todos os objetos mantido em memória, ver seus valores e tipos de dados (muito útil para propósitos de debug!). Ademais, na aba History podemos observar o históricos de comandos previamente executados.
- **Painel de Visualização:** Aqui é onde podemos navegar em arquivos (Files), visualizar nossos gráficos (Plots), verificar os pacotes instalados (Packages) e acessar o manual do R (Help). Em especial, o manual provê informações valiosas acerca R (é importante ao menos fazer uma visita).

O recomendado é sempre salvar seus scripts de programação (mesmo o RStudio sendo “espertinho” e deixar salvo em seu histórico). Se organize ;D

2.2 Instalando e carregando pacotes (*packages*)

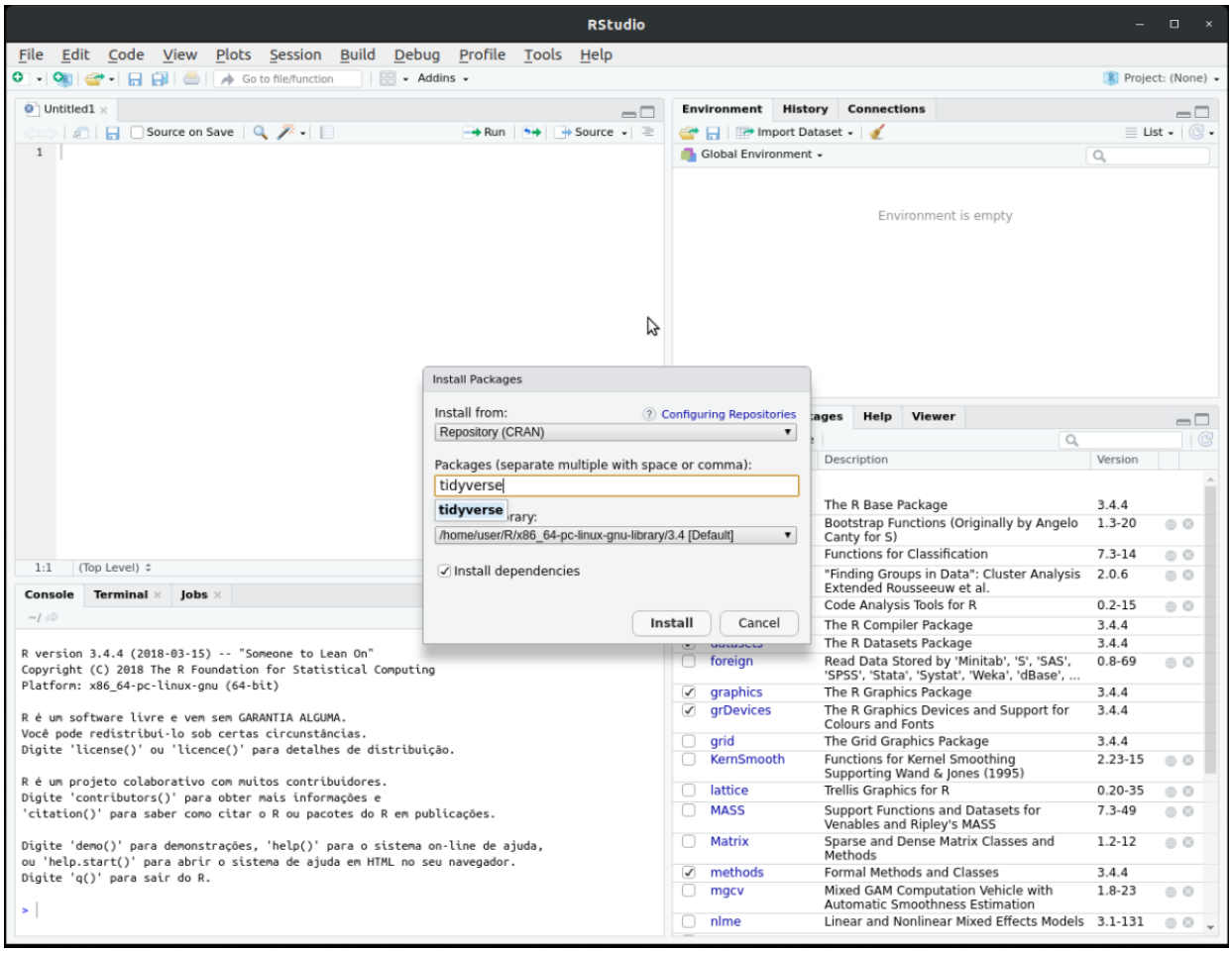
Um pacote (*package*), ou também como conhecido como biblioteca, é uma coleção de funções, dados, e documentação que estende as capacidades nativas do R. A base da linguagem R já conta com diversas funções e tipos de dados que podemos usar sem carregar qualquer pacote externo. Entretanto, para a maioria das manipulações de dados feitas em um projeto de Ciência de Dados, torna-se essencial o uso de pacotes específicos.

A maioria dos pacotes a serem usados ao longo da nossa disciplina estão dispostos no pacote chamado `tidyverse`. Os pacotes dentro do `tidyverse` já são projetados para trabalharem de maneira integrada. A seguir vamos conferir duas maneiras de instalar esse (super-)pacote.

A primeira forma é instalá-lo por meio da execução de uma simples linha de código:

```
install.packages("tidyverse")
```

A segunda forma é acessar a aba Packages no Painel de Visualização e clicar no botão "Install". Uma nova janela se abrirá para você informar o nome do pacote a ser instalado e onde o mesmo será instalado. Após digitar `tidyverse` (que será autocompletado), o pacote pode então ser instalado clicando em Install, como na tela a seguir:



Tela Inicial do RStudio

Os dois procedimentos anteriores podem ser utilizados para instalarem outros pacotes do R.

A instalação do pacote não significa que a partir de agora todos os programas escritos em R consigam usar as funções do `tidyverse`. Para que um programa possa utilizar alguma função desse pacote, devemos carregá-lo.

O comando para carregar um pacote em R é através da função `library()`. Como por exemplo:

```
library(tidyverse)
```

Que terá a seguinte saída ao ser executado:

```
## ── Attaching packages ─────────────────────────────────────── tidyverse
1.3.0 ──
```

```
## ✓ ggplot2 3.3.0     ✓ purrr   0.3.3
## ✓ tibble  3.0.1     ✓ dplyr   0.8.5
## ✓ tidyr   1.0.2     ✓ stringr 1.4.0
## ✓ readr   1.3.1     ✓ forcats 0.5.0
```

```
## ── Conflicts ─────────────────────────────────────── tidyverse_conflic
ts() ──
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Isso mostra que todo o conjunto de pacotes dentro do `tidyverse` foi carregado com sucesso. Uma atenção especial ao `ggplot2` , que será introduzido na próxima semana de estudos.

3 Programando e executando códigos em R

O objetivo desse material (e da disciplina) não é se aprofundar na programação do R. O objetivo é aprender os conceitos básicos e vários atalhos que nos ajudarão a entender e manipular dados em R. O intuito é que este material (e disciplina) ajude-o a entender os fundamentos da linguagem e que permita evoluir nela posteriormente, executando tarefas mais complexas.

3.1 Introdução

Existem alguns fundamentos (https://r.introbook.stuzog.com/the-fundamentals-of-r.html#four-fundamentals) na utilização do R

- **[Uma]** importante característica a se lembrar:
  - R não é procedural e suas estruturas de dados são principalmente baseadas em vetores (vamos ver adiante).
- **[Duas]** razões básicas para usar R em projetos de Ciência de Dados:
  - R é desenvolvido para manipular Dados! Tudo gira em torno deles.
  - R provê funcionalidades para visualização rápida de dados! Isso ajuda a entendê-los.
- **[Três]** princípios básicos do R (fundamentado pelo John Chambers):
  - Objeto! Podemos assumir que tudo em R é um objeto.
  - Funções! Dados são manipulados por funções.
  - Interfaces! Os pacotes devem se comunicar de forma integrada.
- **[Quatro]** formas de programar em R:
  - Linha de comando (via Console)
  - Arquivo fonte (a ser executado conjuntamente)
  - Interface gráfica (via painel do RStudio, por exemplo)
  - Execução de partes de código (via painel do RStudio, por exemplo)

Agora vamos praticar um pouco! Abra seu RStudio, crie um novo arquivo R, e vamos nos aventurar no mundo do R.

3.2 Variáveis e tipos de dados

Uma variável é, um contexto mais semântico, alguma quantidade, qualidade ou propriedade que você possa armazenar (ou medir). Um valor é um estado de uma variável a partir de um momento que você armazena algo (ou faça alguma medição).

Suponha que você queira fazer um programa em R que calcula o resultado de uma equação de segundo grau da forma $ax^2 + bx + c = 0$. Podemos usar a fórmula de Bhaskara para resolver tais tipos de equações, com as seguintes soluções:

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$
$$x_2 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

Essas equações dependem dos valores dos coeficientes a, b, c para serem resolvidas. Ou seja, esses coeficientes são as nossas variáveis, cujos valores são usados para resolver a expressão matemática.

Se tivermos a seguinte equação: $2x^2 + 8x + 3 = 0$. Podemos iniciar a escrita de um programa R da seguinte maneira:

```
a <- 2
b <- 8
c <- 3
```

No código acima declaramos as 3 variáveis necessárias para o nosso problema. Fizemos muito mais que isso! Já atribuimos valores à elas, por meio do `<-` . A atribuição de valores também pode ser feita por meio do `=` , mas é preferível que `<-` seja usado para evitar confusões.

O nome de uma variável deve seguir algumas regras, tal como começar com uma letra, conter somente caracteres alfanuméricos (incluindo `.` ou `_`). Além disso, não é recomendado dar nomes de variáveis padrões do R.

Ao executar o código anterior no R, não iremos obter nenhuma saída no Console. Para verificarmos o valor de uma variável, basta executar uma linha de comando com o seu nome:

```
a
```

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

Ou ainda, usar funções explícitas, como o `print()` :

```
print(b)
```

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

Usando variáveis fica quase que direto a transformação de fórmulas para a linguagem de programação R. Ademais, conseguimos resolver problemas de forma genérica, tal como o nosso caso das equações:

```
x1 <- (-b - sqrt(b ^ 2 - 4 * a * c)) / (2 * a)
x2 <- (-b + sqrt(b ^ 2 - 4 * a * c)) / (2 * a)

print(paste0("O valor de x1 é ", x1, " e o valor de x2 é ", x2))
```

```
## [1] "O valor de x1 é -3.58113883008419 e o valor de x2 é -0.4188611699158
1"
```

Nesse caso, mostramos o valor de maneira formatada usando uma função do pacote básico do R, `paste0()`, a qual realizada a concatenação de valores. Não se preocupe em entender todas as funções e operadores da fórmula acima, isso será explicado logo em seguida.

Podemos trocar os valores dos coeficientes e nossa solução permanecerá a mesma. Claro, esse código precisa de ajustes! Por exemplo, caso o resultado dentro da raiz quadrada seja negativo (o que acontece?)

Praticamente tudo em R são objetos e isso inclui as variáveis. Note que não definimos explícitamente um tipo de dado para uma variável em nossas definições anteriores. O motivo é que R não é uma linguagem fortemente tipada, ou seja, ela não requer a definição de um tipo de dado pois essa definição é feita pelo próprio R a priori.

Abaixo estão os tipos de dados que talvez teremos mais contato ao projetar aplicações de Ciência de Dados (o conjunto de funções iniciadas por `is.[nome do tipo do dado]()` ajuda-nos a checar o tipo de dado; outra possibilidade é a função `class()`):

- **Numeric**: este tipo de dado pode assumir ainda dois sub tipos - **integer** e **double**. Alguns exemplos:

```
y <- 10
is.numeric(y)
## [1] TRUE
is.double(y)
## [1] TRUE
is.integer(y)
## [1] FALSE
class(y)
## [1] "numeric"
```

A resposta negativa de `y` ser um **integer** é que um tipo numérico é tratado como **double** por padrão. Para "forçar" o tipo inteiro, deve-se colocar a letra `L` após o valor indicado, tal como:

```
y <- 10L
is.numeric(y)
## [1] TRUE
is.double(y)
## [1] FALSE
is.integer(y)
## [1] TRUE
class(y)
## [1] "integer"
```

- **Character** (também conhecido como **Nominal**): este tipo de dado manipula cadeias de caracteres (conhecidas como *strings*). Por exemplo:

```
nome <- "antonio"
is.character(nome)
## [1] TRUE
class(nome)
## [1] "character"
```

- **Date**: este tipo de dado armazena datas (possivelmente com hora). Para criar um objeto deste tipo de dado, usamos as funções `as.Date()` e `as.POSIXct`, informando a data no formato ISO 8601. Alguns exemplos:

```
data <- "2020-05-09"
class(data)
## [1] "character"
data <- as.Date(data)
class(data)
## [1] "Date"

data_com_hora <-as.POSIXct("2020-05-09 09:00")
data_com_hora
## [1] "2020-05-09 09:00:00 -03"
```

- **Logical**: este tipo de dado armazena ou `TRUE` (`T` ou `1`) ou `FALSO` (`F` ou `0`).

```
logi <- TRUE
class(logi)
## [1] "logical"
is.logical(logi)
## [1] TRUE
```

Além disso, temos o tipo de dado `NULL`. Ele é usado para indicar a ausência de uma variável (ou vetor).

Por fim, temos o `NA` que significa *not available*. Isto é, não está disponível de fato um valor definido/disponível (a ausência de um valor para uma variável).

Dica

Passe o mouse sobre uma variável no painel superior a direita no RStudio e você verá a quantidade de bytes ocupada por seu valor e o seu tipo de dado entre parênteses.

3.3 Estrutura de dados básica: o vetor e a lista

A estrutura de dados fundamental do R é o vetor (*vector*). Um vetor nada mais é que a coleção de elementos. Essencialmente, todos os tipos de dados apresentados anteriormente são vetores de um único elemento.

Podemos ter os chamados **Vetores atômicos**, os quais armazenam valores do mesmo tipo. Portanto, esse tipo de vetor é **homogêneo**.

Além disso, temos as **Listas** (*list*), as vezes chamadas de vetores recursivos, pois uma lista pode conter uma outra lista. Esse tipo de vetor pode armazenar dados **heterogêneos** (ou seja, misturar tipos de dados em uma mesma estrutura).

Podemos criar um vetor ao utilizar a função `c()` (que significa *combine*) e listando os seus elementos dentro da função. Exemplos são dados a seguir:

Um vetor contendo inteiros:

```
x <- c(1, 2, 4, 5, 6, 10)
x
## [1]  1  2  4  5  6 10
```

Um vetor contendo caracteres (ou seja, *strings*):

```
coisas <- c("casa", "carro", "televisão")
coisas
## [1] "casa"      "carro"     "televisão"
```

Podemos ainda criar uma lista usando a função `list`:

```
l <- list("casa", "carro", 1:5, x)
l
## [[1]]
## [1] "casa"
##
## [[2]]
## [1] "carro"
##
## [[3]]
## [1] 1 2 3 4 5
##
## [[4]]
## [1]  1  2  4  5  6 10
```

Ao executar os comandos acima, você irá notar que o RStudio deixa o `coisas e x` como *Values* e o `l` como *Data* (no painel superior a direita), e ao clicar no `l` podemos visualizar seus valores.

Note o uso do operador `:` entre os números, para produzir números em um determinado intervalo. Note ainda que `[[i]]`, com $1 <= i >= 4$, para mostrar os itens da lista `l`, denotando um elemento `i` da lista (veremos mais sobre essas notações posteriormente). Observe também que o primeiro elemento de um vetor/lista é o 1 (ao contrário de muitas linguagens que se inicia pelo 0).

Todo vetor possui 2 características fundamentais:

1. Um **tipo**, determinado pela função `typeof()`.
2. Um **tamanho**, determinado pela função `length()`.

Algums exemplos:

```
typeof(l)
## [1] "list"
length(l)
## [1] 4

typeof(x)
## [1] "double"
length(x)
## [1] 6
```

Vetores e listas podem ser estendidos para ter um comportamento diferenciado. Por exemplo:

- **Factors**, os quais são valores que expressam categorias. Isto é, variáveis que tem um tamanho fixo e seus valores são bem conhecidos (chamados de `levels`). Isso se assemelha ao tipo `enum` de outras linguagens, sendo que internamente um **Factor** é tratado como vetor de inteiros.
- **Data frames** e **tibbles**. Um **data frame** (ou **data.frame**) é composto de linhas e colunas, organizando os dados de maneira tabular. Cada nome de coluna é uma variável (ou seja, **vetor** ou **lista**) e cada linha é uma observação (*observation* - termo usado em várias referências; implementado como um **vetor**/**lista**). Ou seja, uma observação consiste de valores associados à diversas variáveis.

Iremos voltar a falar sobre esses tipos de dados ao discutir o processo de carregamento de dados no R e ao longo da disciplina.

Iremos observar também que vários operadores podem ser aplicados diretamente sobre vetores. Dessa forma, o operador é aplicado para cada valor do vetor. O resultado da operação é um novo vetor. Em seus estudos, você pode encontrar os seguintes termos que descrevem esse tipo de comportamento: *element-wise operator* e *vectorised operation/operator*.

3.4 O básico

3.4.1 Operadores aritméticos

R possui diversos operadores que podem ser usados para executar expressões matemáticas diretamente como se fosse uma calculadora.

```
1 + 1 #adição
## [1] 2
1 - 1 #subtração
## [1] 0
1 / 10 #divisão
## [1] 0.1
1 * 1 #multiplicação
## [1] 1
3 ^ 3 #potência
## [1] 27
(1 + 5) * 5 / (75 - (2 ^ 3)) #uma expressão matemática
## [1] 0.4477612
```

Assim como outras linguagens, R segue a ordem padrão de operadores do **PEMDAS**:

1. **P**arentheses ( )
2. **E**xponents ^
3. **M**ultiplication x
4. **D**ivision /
5. **A**ddition +
6. **S**ubtraction -

Além disso, temos o *mod* e a *divisão inteira* (com mesma prioridade da divisão):

```
4 %% 2 # esse é o módulo em R
## [1] 0
4 %/% 3 # essa é a divisão inteira no R
## [1] 1
#note que o caracter # simboliza comentários em R
#comentários não são processados e servem para documentar internamente o cód
igo
```

Conforme comentado anteriormente, vários operadores podem ser usados sobre vetores e isso não é diferente para os operadores aritméticos.

```
x <- c(25, 2, 5, 8, 9)
x * 2
```

```
## [1] 50  4 10 16 18
```

Podemos inclusive fazer operações sobre vários vetores.

```
#note que iremos reaproveitar a variável x
y <- c(3, 5, 6, 1, 0)
z <- c(45, -9, 2, 4, 5)
x + y
## [1] 28  7 11  9  9
x - y
## [1] 22 -3 -1  7  9
x ^ y + 1
## [1] 15626    33 15626     9     2
x - y + z * 2
## [1] 112 -21   3  15  19
```

3.4.2 Operadores relacionais

Um operador relacional retorna um valor lógico (TRUE ou FALSE) indicando se o operador é satisfeito ou não. Além disso, pode-se retornar um NA . Abaixo é mostrado a tabela de operadores relacionais do R:

Operador	Descrição
<	Menor que
>	Maior que
<=	Menor ou igual
>=	Maior ou igual
==	Igual
!=	Diferente

Alguns exemplos de uso:

```
1 < 10
## [1] TRUE
(12 * 5) > 50
## [1] TRUE

#podemos usar também sobre vetores e o resultado? um outro vetor!
m <- c(1.3, 4.5, 6)
m > 2
## [1] FALSE  TRUE  TRUE
#podemos observar que apenas os 2 últimos valores são maiores que 2, portant
o, suas respectivas posições são "marcadas"

2 == 2
## [1] TRUE
2 == NA
## [1] NA
NA != 2
## [1] NA

NA > 5
## [1] NA
```

O mal do `NA` pode ser catastrófico ao analisar os dados pois ele pode contaminar uma base de dados inteira! Uma maneira de evitar isso, é checando se uma variável tem o valor `NA` (isso será útil futuramente):

```
v <-NA
is.na(v) == TRUE
## [1] TRUE
```

Ao digitar o `is.` e segurar Ctrl+Espaço o RStudio te fornece diversas opções de funções para checar seu tipo de dado, dentre elas está a `is.na()`.

3.4.3 Operadores lógicos

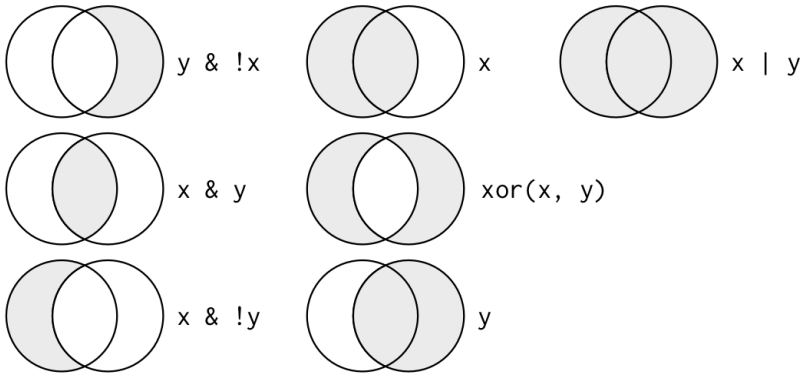
Os operadores lógicos do R são o AND (&), OR (|) e NOT (!) e eles são cruciais para combinar expressões lógicas retornadas pelos operadores relacionais. A seguir a tabela verdade do R é mostrada.

```
a <- TRUE
b <- FALSE
c <- NA
# AND
a & a
## [1] TRUE
a & b
## [1] FALSE
a & c
## [1] NA
b & b
## [1] FALSE
b & c
## [1] FALSE
c & c
## [1] NA

#OR
a | a
## [1] TRUE
a | b
## [1] TRUE
a | c
## [1] TRUE
b & b
## [1] FALSE
b & c
## [1] FALSE
c & c
## [1] NA

#NOT
!a
## [1] FALSE
!b
## [1] TRUE
!c
## [1] NA
```

Note que além dos 2 valores verdade `TRUE` e `FALSE` temos o `NA`. Já que ele significa *not available*, operadores lógicos sobre ele também resultam em `NA`.

Operadores lógicos são úteis para selecionarmos um subconjunto de dados que satisfaz um determinado conjunto de restrições. Tais restrições são combinadas por meio de operadores AND (&), OR (|) e NOT (!).

A imagem abaixo mostra o uso dos operadores lógicos (fonte: R for Data Science (https://r4ds.had.co.nz/transform.html)). O círculo da esquerda denota uma condição a ser atendida (*x*) e o da direita denota outra condição (*y*). A parte em cinza mostra que partes cada operador lógico seleciona:




```
v <- c(10, 3, 2.5, 9, 11, 245, 3, 14, 56, 1, 2, 0)
#vamos "marcar" as posições dos números pares (círculo da esquerda) e que sã
o maiores que 6 (círculo da direita)
v %% 2 == 0 & v > 6
```

```
##  [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FA
LSE
```

Atenção que o vetor resultante apenas **marca** quais elementos do vetor `v` satisfaz as restrições definidas. Na próxima semana iremos ver como de fato filtrar os dados. Mas podemos adiantar que isso pode ser feito com a função `filter()`.

Atenção ao uso dos operadores lógicos

Os operadores lógicos `&` e `|` são aplicados sobre vetores, ou seja, eles fazem a checagem lógica a cada elemento de um vetor. Isso significa que esses operadores não tem o mesmo significado clássico de outras linguagens de programação, como C.

Uma vez que os operadores lógicos `&` e `|` são para serem usados em vetores (pois ele são processados em um conjunto de dados), precisamos usar outros operadores lógicos para o AND e OR quando queremos apenas processar um valor único. Ou seja, criar *short-circuits* nas condições de estruturas de decisão e repetição (como `if` e `while`). Para isso, devemos utilizar os seguintes operadores lógicos: `&&` para o AND e `||` para o OR. Isso ficará mais claro ao avançarmos na disciplina.

4 Importação de dados em R

Um dos processos iniciais de um projeto de Ciência de Dados é o carregamento dos dados. Estes podem ser oriundos de diversas fontes, variando seu formato e tipos de dados. Ao carregar um arquivo de dados no R, ele mesmo vai definir seus tipos de dados por padrão (claro, se nenhuma interferência for feita ao executar o comando).

Nesta seção iremos entender como carregar 2 tipos de arquivos:

- .csv
- .xml

O objetivo é entender como funciona as funções usadas para fazer essa carga e os tipos de dados que o R vincula à esses dados. Sua tarefa também inclui entender os parâmetros adicionais de cada função usada.

4.1 Importando um arquivo com valores separados por delimitadores

Um arquivo .csv é basicamente um arquivo texto em formato tabular cujo valores são separadas por algum caracter **delimitador**. O **csv** significa *comma-separated values*, ou seja, os valores são separados por `,`. Entretanto, podemos ter um arquivo .tsv onde os valores são separados pelo `tab`.

Esses arquivos podem ser carregados ao R usando as seguintes funções do pacote `readr` dentro do `tidyverse`:

- `read_csv()` - lê um arquivo .csv tradicional (separado por vírgulas)
- `read_csv2()` - lê um arquivo .csv separado por ponto e vírgulas e considera a vírgula como separador de decimal
- `read_tsv()` - lê um arquivo .tsv tradicional (separado por `tab`)
- `read_delim()` - lê um arquivo delimitado por algum caracter qualquer

Essas funções são documentadas no manual do `tidyverse`. (https://readr.tidyverse.org/reference/read_delim.html) - se acostume a sempre ler a documentação de função pois essa é uma tarefa bastante habitual ao programar projetos de Ciência de Dados. Por exemplo, a documentação define a função `read_csv()` da seguinte forma:

```
read_csv(file, col_names = TRUE, col_types = NULL,
  locale = default_locale(), na = c("", "NA"), quoted_na = TRUE,
  quote = "\"", comment = "", trim_ws = TRUE, skip = 0,
  n_max = Inf, guess_max = min(1000, n_max),
  progress = show_progress(), skip_empty_rows = TRUE)
```

3 observações:

- **[Uma]** boa documentação sempre nos dará todos os parâmetros e seus possíveis valores, ensinando-nos a usá-la.
- **[Duas]** coisas a serem notadas:
  - um **parâmetro** é algo a ser passado para a função (quando nós a executamos) entre parênteses.
  - o parâmetro deve ou (i) seguir a ordem disposta na documentação, ou (ii) seu nome e valor devem ser informados (por exemplo, `read_csv(file = 'caminho para o nosso arquivo', col_names = FALSE)` para indicar onde nosso arquivo está armazenado (`file`) e que ele não possui nomes de colunas em seu cabeçalho (`col_names`))
- **[Três]** pontos importantes sobre funções:
  - o seu nome (lembre-se! R é sensível às maiúsculas e minúsculas)
  - quais são seus parâmetros com valores exigidos (iremos notar isso ao ver que somente o nome do parâmetro é dado na documentação, sem qualquer valor padrão - ou seja, sem o sinal de `=` e um valor)
  - quais são seus parâmetros com valores padrões (ou seja, parâmetros com acompanhados com a atribuição `=` de um valor)

Agora que entendemos, de maneira prática (já usamos funções ao longo desse material) e teórica como executamos funções, vamos utilizar a função que lê o nosso arquivo disponibilizado.

Inicialmente, temos que importar o nosso pacote, lembra-se?

```
library(tidyverse)
```

Agora podemos carregar nosso arquivo .csv (oriundo do Portal Brasileiro de Dados Abertos (http://www.dados.gov.br/)) e aplicar algumas funções e operações vistas anteriormente, aplicando os conceitos vistos. Nossa base de dados contém informações sobre o dinheiro em circulação no Brasil (http://www.dados.gov.br/dataset/dinheiro-em-circulacao):

> Diariamente, são feitas movimentações que alteram a quantidade e composição do numerário em circulação. O presente conjunto de dados tem por objetivo apresentar as quantidades diárias por Valor da Denominação e Família (espécie e categoria).

- **Data:** formato ano-mês-dia - indica a data a que se refere o registro de quantidade daquela denominação e família;
- **Família:** Agrupamento da espécie (cédulas, moedas e peças comuns e comemorativas) e característica da espécie;
- **Denominação:** valor do numerário;
- **Quantidade:** montante em circulação.

Como pode ser visto anteriormente, as colunas desse .csv são Data | Família | Denominação | Quantidade. Entretanto, nosso .csv possui 2 características principais:

- ele não tem linha de cabeçalho e portanto, precisamos específicar quais são os nomes das colunas
- ele é na verdade separado por ponto e vírgula, usa a vírgula para separador de decimal e o ponto para separador de milhar.

Dessa forma, podemos utilizar a função `read_csv2()` que ao ser executada, mostrará os seguintes dados:

Data	Família	Denominação	Quantidade
<date>	<chr>	<chr>	<dbl>
1995-01-02	Moedas - 1a. Família (inox)	0.01	834342314
1995-01-02	Moedas - 1a. Família (inox)	0.05	636711876
1995-01-02	Moedas - 1a. Família (inox)	0.10	583076666
1995-01-02	Moedas - 1a. Família (inox)	0.25	146883388
1995-01-02	Moedas - 1a. Família (inox)	0.50	329791540
1995-01-02	Cédulas - 1a. família	1.00	276894464
1995-01-02	Moedas - 1a. Família (inox)	1.00	176004272
1995-01-02	Moedas comemorativas 300 anos CMB	2.00	4980
1995-01-02	Moedas comemorativas Tetra campeonato	4.00	618
1995-01-02	Cédulas - 1a. família	5.00	233378314

1-10 of 10,000 rows	Previous	**1**	2	3	4	5	6	…	1000	Next

Um exemplo completo é mostrado abaixo. Note que é necessário informar o caminho completo onde o arquivo está armazenado. No exemplo, este arquivo está na pasta `/data/R-files/`. Sua tarefa é entender (praticando) como funciona os operadores `[[]]`, `$` e `[,]`.

```
#opcionalmente, pode-se colocar a url direto do arquivo .csv
df <- read_csv2("/data/R-files/MeioCirculante_DadosAbertos.csv", col_names =
c("Data", "Família", "Denominação", "Quantidade"))
## Using ',' as decimal and '.' as grouping mark. Use read_delim() for more
 control.
## Parsed with column specification:
## cols(
##   Data = col_date(format = ""),
##   Família = col_character(),
##   Denominação = col_character(),
##   Quantidade = col_number()
## )
#ao clicar na variável df no painel superior a direita do RStudio, poderemos
ver seu conteúdo tabular graficamente

#mostrando alguns valores específicos
df[[1]]
##     [1] "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02"
##     [6] "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02"
##    [11] "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-03" "1995-01-03"
##    [16] "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03"
##    [21] "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03"
##    [26] "1995-01-03" "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04"
##    [31] "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04"
##    [36] "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-05"
##    [41] "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05"
##    [46] "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05"
...

df[, 3]
## # A tibble: 346,556 x 1
##    Denominação
##    <chr>
##  1 0.01
##  2 0.05
##  3 0.10
##  4 0.25
##  5 0.50
##  6 1.00
##  7 1.00
...

df[4, ]
## # A tibble: 1 x 4
##   Data       Família                    Denominação Quantidade
##   <date>     <chr>                      <chr>            <dbl>
## 1 1995-01-02 Moedas - 1a. Família (inox) 0.25         146883388

df[[2]][[2]]
## [1] "Moedas - 1a. Família (inox)"

df$Data
##     [1] "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02"
##     [6] "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-02"
##    [11] "1995-01-02" "1995-01-02" "1995-01-02" "1995-01-03" "1995-01-03"
##    [16] "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03"
##    [21] "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03" "1995-01-03"
```

```
##    [26] "1995-01-03" "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04"
##    [31] "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04"
##    [36] "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-04" "1995-01-05"
##    [41] "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05"
##    [46] "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05" "1995-01-05"
...
unique(df$Data) #essa é uma função importante, o que ela faz?
##    [1] "1995-01-02" "1995-01-03" "1995-01-04" "1995-01-05" "1995-01-06"
##    [6] "1995-01-09" "1995-01-10" "1995-01-11" "1995-01-12" "1995-01-13"
##   [11] "1995-01-16" "1995-01-17" "1995-01-18" "1995-01-19" "1995-01-20"
##   [16] "1995-01-23" "1995-01-24" "1995-01-25" "1995-01-26" "1995-01-27"
##   [21] "1995-01-30" "1995-01-31" "1995-02-01" "1995-02-02" "1995-02-03"
##   [26] "1995-02-06" "1995-02-07" "1995-02-08" "1995-02-09" "1995-02-10"
##   [31] "1995-02-13" "1995-02-14" "1995-02-15" "1995-02-16" "1995-02-17"
##   [36] "1995-02-20" "1995-02-21" "1995-02-22" "1995-02-23" "1995-02-24"
##   [41] "1995-03-01" "1995-03-02" "1995-03-03" "1995-03-06" "1995-03-07"
##   [46] "1995-03-08" "1995-03-09" "1995-03-10" "1995-03-13" "1995-03-14"
...
df$Denominação[[5]]
## [1] "0.50"

#vamos tentar fazer um filtro para buscar somente as moedas em circulação em
2017-05-16.
circulacao <- df$Data == "2017-05-16"
#será que a variável é o que realmente queremos? veja o tipo de dados que el
a possui

#para fazer esse tipo de filtragem, precisamos avançar um pouco nos conceito
s, usando o pacote dplyr (já dentro do tidyverse)

circulacao_real <- filter(df, Data == "2017-05-16")

#explore a funcionalidade da função filter() seguindo o exemplo acima e usan
do os operadores lógicos e relacionais
#veremos ele de forma mais detalhada na próxima semana

#qual é a soma do montante em circulação em 2017-05-16?
sum(circulacao_real$Quantidade)
## [1] 30798552774
```

Nota importante

Em R existe a função base chamada `read.csv` que a príncipio faz a mesma funcionalidade que a função `read_csv`. Você irá notar que R possui diversas funções de diferentes pacotes que possuem as mesmas funcionalidades (ou funcionalidades muito parecidas). Entretanto, uma diferença pode ser crucial, o **desempenho**. Nesse sentido, `read_csv` é melhor em desempenho (http://yetanothermathprogrammingconsultant.blogspot.com/2016/12/reading-csv-files-in-r-readcsv-vs.html).

4.2 Importando um arquivo .xml

Um arquivo .xml (do inglês *Extensible Markup Language*) é bastante utilizado em arquivos de configuração ou para transferência de dados entre aplicações. Seu formato é baseado em `tags` dispostas de maneira **hierárquica**.

Esse tipo de arquivo pode ser carregado ao R usando a seguinte função `read_xml()` do pacote `xml2` (https://cran.r-project.org/web/packages/xml2/xml2.pdf). Entretanto, devido à flexibilidade de um arquivo .xml, essa função não nos dá um `data.frame` bem definido. Ao invés disso, precisamos navegar nos seus filhos e elementos. E após essa navegação, podemos montar um `data.frame` como quisermos.

Assim, precisamos instalar o pacote `xml2` :

```
install.packages("xml2")
```

Um exemplo de uso é ao carregar o arquivo .xml disponibilizado. Sua tarefa é compreender como os conceitos vistos antes (por exemplo, sobre **listas** e **vetores**) podem ser usados para construir um `data.frame` por meio de sua "função construtora".

Um pouco mais sobre o .xml disponibilizado (também oriundo do Portal Brasileiro de Dados Abertos (http://www.dados.gov.br/)) é discutido a seguir. O .xml contém informações sobre a Pesquisa Industrial Anual - Produto (http://www.dados.gov.br/dataset/pj-pesquisa-industrial-anual-produto):

> Tabelas SIDRA Tabela 5806 - Produção e vendas dos produtos e/ou serviços industriais, segundo as classes de atividades e os produtos - Prodlist Indústria 2013

Um exemplo completo é mostrado abaixo. Note que é necessário informar o caminho completo onde o arquivo está armazenado. No exemplo, este arquivo está na pasta `/data/R-files/` .

```
#primeiro temos que carregar o pacote xml2
library(xml2)

dados_xml <- read_xml("/data/R-files/PesquisaIndustrialAnual-Produto.xml")

xml_name(dados_xml) #o pacote xml2 possui diversas funções para entendermos
 o nosso arquivo xml
## [1] "ArrayOfValorDescritoPorSuasDimensoes"

xml_children(dados_xml)
## {xml_nodeset (13)}
##  [1] <ValorDescritoPorSuasDimensoes>\n  <V>Valor</V>\n  <MC>Unidade de Me
dida ...
##  [2] <ValorDescritoPorSuasDimensoes>\n  <V>2590773318</V>\n  <MC>40</MC>
\n  < ...
##  [3] <ValorDescritoPorSuasDimensoes>\n  <V>2548936587</V>\n  <MC>40</MC>
\n  < ...
##  [4] <ValorDescritoPorSuasDimensoes>\n  <V>2633128551</V>\n  <MC>40</MC>
\n  < ...
##  [5] <ValorDescritoPorSuasDimensoes>\n  <V>2888458617</V>\n  <MC>40</MC>
\n  < ...
##  [6] <ValorDescritoPorSuasDimensoes>\n  <V>94639</V>\n  <MC>1020</MC>\n
 <MN> ...
##  [7] <ValorDescritoPorSuasDimensoes>\n  <V>93119</V>\n  <MC>1020</MC>\n
 <MN> ...
##  [8] <ValorDescritoPorSuasDimensoes>\n  <V>89738</V>\n  <MC>1020</MC>\n
 <MN> ...
##  [9] <ValorDescritoPorSuasDimensoes>\n  <V>91249</V>\n  <MC>1020</MC>\n
 <MN> ...
## [10] <ValorDescritoPorSuasDimensoes>\n  <V>2165993301</V>\n  <MC>40</MC>
\n  < ...
## [11] <ValorDescritoPorSuasDimensoes>\n  <V>2158084466</V>\n  <MC>40</MC>
\n  < ...
## [12] <ValorDescritoPorSuasDimensoes>\n  <V>2164230347</V>\n  <MC>40</MC>
\n  < ...
## [13] <ValorDescritoPorSuasDimensoes>\n  <V>2304912151</V>\n  <MC>40</MC>
\n  < ...

#em suma, nosso XML possui 11 tipos de tags que denotam atributos (repare qu
e essa não é bem a organização única de todos os arquivos XML e isso pode va
riar bastante)

#essa função vai conseguir identificar os nomes das tags de maneira adequada
xml_ns_strip(dados_xml)

#essa função extrai todos os nós que segue um padrão do xpath (linguagem de
 XML)
nodes_v <- xml_find_all(dados_xml, "//V")
#aqui conseguimos extrair o valor dentro da tag (para extrair de atributos,
 usamos xml_attr)
val_v <- xml_text(nodes_v)
val_v
##  [1] "Valor"      "2590773318" "2548936587" "2633128551" "2888458617"
##  [6] "94639"      "93119"      "89738"      "91249"      "2165993301"
## [11] "2158084466" "2164230347" "2304912151"

#note que o primeiro valor de val_v é uma palavra (na verdade, nome da colun
```

```
a), e por isso não conseguimos converter para o tipo numeric. Podemos fazer
 isso da seguinte forma:

val_v <- as.numeric(val_v[2:length(val_v)]) # note que extraimos somente os
 valores 2 até o tamanho do vetor (ignorando o primeiro valor) --> isso é fe
ito pelo operador : Posteriormente, o resultado disso é convertido em numeri
c, pois meio da função as.numeric
val_v
##  [1] 2590773318 2548936587 2633128551 2888458617      94639      93119
##  [7]      89738      91249 2165993301 2158084466 2164230347 2304912151

#agora que entendemos como capturar os elementos do nosso XML, podemos fazer
um data.frame. Uma das formas possíveis é a seguinte:

nodes_mc <- xml_find_all(dados_xml, "//MC")
val_mc <- xml_text(nodes_mc)
val_mc <- as.numeric(val_mc[2:length(val_mc)])

nodes_mn <- xml_find_all(dados_xml, "//MN")
val_mn <- xml_text(nodes_mc)
val_mn <- val_mc[2:length(val_mn)]

nodes_nc <- xml_find_all(dados_xml, "//NC")
val_nc <- xml_text(nodes_mc)
val_nc <- as.numeric(val_mc[2:length(val_nc)])

nodes_nn <- xml_find_all(dados_xml, "//NN")
val_nn <- xml_text(nodes_mc)
val_nn <- val_mc[2:length(val_nn)]

nodes_d1c <- xml_find_all(dados_xml, "//D1C")
val_d1c <- xml_text(nodes_mc)
val_d1c <- as.numeric(val_mc[2:length(val_d1c)])

nodes_d1n <- xml_find_all(dados_xml, "//D1N")
val_d1n <- xml_text(nodes_mc)
val_d1n <- val_mc[2:length(val_d1n)]

nodes_d2c <- xml_find_all(dados_xml, "//D2C")
val_d2c <- xml_text(nodes_mc)
val_d2c <- as.numeric(val_mc[2:length(val_d2c)])

nodes_d2n <- xml_find_all(dados_xml, "//D2N")
val_d2n <- xml_text(nodes_mc)
val_d2n <- val_mc[2:length(val_d2n)]

nodes_d3c <- xml_find_all(dados_xml, "//D3C")
val_d3c <- xml_text(nodes_mc)
val_d3c <- as.numeric(val_mc[2:length(val_d3c)])

nodes_d3n <- xml_find_all(dados_xml, "//D3N")
val_d3n <- xml_text(nodes_mc)
val_d3n <- as.numeric(val_mc[2:length(val_d3n)])

nodes_d4c <- xml_find_all(dados_xml, "//D4C")
val_d4c <- xml_text(nodes_mc)
val_d4c <- as.numeric(val_mc[2:length(val_d4c)])
```

```
nodes_d4n <- xml_find_all(dados_xml, "//D4N")
val_d4n <- xml_text(nodes_mc)
val_d4n <- as.numeric(val_mc[2:length(val_d4n)])

df_do_xml <- data.frame(
  "Valor" = val_v,
  "Unidade de Medida (Código)" = val_mc,
  "Unidade de Medida" = val_mn,
  "Nível Territorial (Código)" = val_nc,
  "Nível Territorial" = val_nn,
  "Brasil (Código)" = val_d1c,
  "Brasil" = val_d1n,
  "Variável (Código)" = val_d2c,
  "Variável" = val_d2n,
  "Ano (Código)" = val_d3c,
  "Ano" = val_d3c,
  "Classes das atividades industriais e produtos - Prodlist 2016 (Código)" =
val_d4c,
  "Classes das atividades industriais e produtos - Prodlist 2016" = val_d4n
)
```

Após carregar o arquivo .xml no `df_do_xml` , temos os seguintes dados:

Valor	Unidade.de.Medida..Código.	Unidade.de.Medida	Nível.Territorial..Códi
<dbl>	<dbl>	<dbl>	<d
2590773318	40	40	
2548936587	40	40	
2633128551	40	40	
2888458617	40	1020	1
94639	1020	1020	1
93119	1020	1020	1
89738	1020	1020	1
91249	1020	40	
2165993301	40	40	
2158084466	40	40	

1-10 of 12 rows | 1-4 of 13 columns Previous **1** 2 Next

Uau! Você deve estar pensando: quantas variáveis e repetição de código para construir esse `data.frame` !

Por isso que podem existir outras maneiras de se fazer a mesma coisa (as vezes de maneira muito mais simples) e o nosso objetivo não é mostrar todas, mas apenas apresentar caminhos possíveis de solução. **Essa é a arte da programação ;D**

Uma opção de melhoramento, principalmente para evitar a repetição de código, é definirmos nossa própria **função** (a ser abordado futuramente). De maneira objetiva, a solução ficaria (assumindo que você esteja com os pacotes carregados):

```
dados_xml <- read_xml("/data/R-files/PesquisaIndustrialAnual-Produto.xml")
xml_ns_strip(dados_xml)

pegar_dados_xml <- function (xml, nome_tag, formato_numerico = FALSE) {
  nodes <- xml_find_all(xml, paste0("//", nome_tag))
  valores <- xml_text(nodes)
  valores <- valores[2:length(valores)]

  #aqui retornamos o que queremos já convertendo os valores para numérico, s
e for o caso
  if(formato_numerico)
    as.numeric(valores)
  else
    valores
}

df_do_xml2 <- data.frame(
  "Valor" = pegar_dados_xml(dados_xml, "V", TRUE),
  "Unidade de Medida (Código)" = pegar_dados_xml(dados_xml, "MC", TRUE),
  "Unidade de Medida" = pegar_dados_xml(dados_xml, "MN"),
  "Nível Territorial (Código)" = pegar_dados_xml(dados_xml, "NC", TRUE),
  "Nível Territorial" = pegar_dados_xml(dados_xml, "NN"),
  "Brasil (Código)" = pegar_dados_xml(dados_xml, "D1C", TRUE),
  "Brasil" = pegar_dados_xml(dados_xml, "D1N"),
  "Variável (Código)" = pegar_dados_xml(dados_xml, "D2C", TRUE),
  "Variável" = pegar_dados_xml(dados_xml, "D2N"),
  "Ano (Código)" = pegar_dados_xml(dados_xml, "D3C", TRUE),
  "Ano" = pegar_dados_xml(dados_xml, "D3N", TRUE),
  "Classes das atividades industriais e produtos - Prodlist 2016 (Código)" =
pegar_dados_xml(dados_xml, "D4C", TRUE),
  "Classes das atividades industriais e produtos - Prodlist 2016" = pegar_da
dos_xml(dados_xml, "D4N", TRUE)
)
```

Retomaremos sobre funções quando formos estudar um pouco mais sobre programação em R =)

5 Exercícios

Resolva a seguinte lista de exercícios e submeta suas respostas no Moodle, conforme orientado na página da disciplina (em suma, existe uma questão no questionário aberto para você mandar as respostas desses exercícios como um arquivo .pdf).

1. Qual é a diferença entre os tipos de dados **integer** e **double**? Dica: utilize o código abaixo para justificar sua resposta:

```
(a <- sqrt(2) ^ 2)
## [1] 2
a - 2
## [1] 4.440892e-16

-1 / 0
## [1] -Inf
0 / 0
## [1] NaN
1 / 0
## [1] Inf
```

2. Escreva um programa em R que processe a seguinte fórmula matemática abaixo (declare as variáveis necessárias e valores para ela):

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

3. Complete o código R abaixo, trocando os `??` , para calcular: (i) a soma dos valores do vetor `x` , (ii) a média de valores do vetor `x` , (iii) o desvio padrão dos valores do vetor `x` , e (iv) valores máximos e mínimos do vetor `x` .

```
x <- seq(1, 1000) #pesquise o que essa nova função faz
soma <- #??
media <- #??
desvio_padrao <- #??
maximo <- #??
minimo <- #??
```

4. Qual é a ordem de execução dos operadores da expressão abaixo? E o seu resultado?

```
4 + 2 - 1 / 10 ^ 3
```

5. Qual é o problema com o seguinte código?

```
a <- c(1, 2, 4)
b <- c(10, 20)
a + b
```

6. Por que o resultado da expressão lógica abaixo é `NA` ? Explique e justifique com suas próprias palavras.

```
NA == NA
```

7. Considerando o vetor abaixo, faça as condições exigidas.

```
vec <- c(2.4, 5, 2, 4, 34, 13, 56, 84)
```

a. Os números ímpares e menores que 10.

b. Os números entre 10 e 55 (incluindo as suas extremidades).