Nivelamento de Programação em R - Aula 1-2: Objetos

Umberto Mignozzetti

Fevereiro 2020

Retomando a aula anterior

Continuemos nossos exercícios de onde paramos no script passado. Até agora o que fizemos foi mais explorar o ambiente do R e ambientes complementares que podem nos ajudar em nossos problemas diários. Vimos a diferença entre o R e o RStudio, bem como aprendemos a navegar pelo ambiente de ambos os programas. Sabemos até agora que podemos escrever nossos comandos diretamente no console, mas preferencialmente os escrevemos em algum tipo de documento (scripts, RMarkdown, RNotebook). Passaremos agora a propriamente fazer algumas operações elementares no R, como criar vetores, abrir bancos de dados, bem como fazer algumas manipulações básicas de dados.

Praticando

1.1) Vimos na aula passada como fazer operações básicas no R. Sem olhar para o documento da aula anterior, tente escrever como fazemos as seguintes operações no R:

- Soma
- Subtração
- Multiplicação
- Divisão
- Resto da Divisão
- Exponenciação
- Logaritmo
- Subconjunto
- Operador de verdadeiro
- Operador lógico de falso
- Sinal de igualdade
- Sinal de maior
- Sinal de maior ou igual
- Sinal de menor
- Sinal de menor ou igual
- Sinal de diferença
- Sinal de negação
- Missing Data
- Not a Number

1.2) Também aprendemos a como carregar pacotes dentro do R. Suponha que um pesquisador tenha criado o pacote chamado "meu_pacote". Que comando você utilizaria para instalar tal pacote? Que comando você utilizaria para carregar tal pacote?

1.3) Suponha que dentro deste pacote tenha a função "minha_funcao". Como você acabou de instalar este pacote, ainda não está familiarizado com os termos desta nova função. Como você poderia descobrir como utilizar este comando dentro do R? E fora do R?

Objetos, Funções e Paradigmas de Linguagem - Uma breve análise do R

Uma das formas de se classificar linguagens é se o paradigma delas são *object-oriented programming* ou *functional programming*. De fato, estes não são os únicos paradigmas de linguagem, mas o debate geralmente se centra nesses dois campo. Caso vocês queiram saber mais acerca de tipos de paradiga de linguagem vejam este link. Caso vocês queiram saber mais do debate de linguagem orientada a objetos vs orientadas a funções vejam este link. O exemplo mais típico de uma linguagem *object-oriented* é o Java, em que temos de nos centrar em objetos para fazer nosso trabalho. No outro extremo, temos linguages como o Lisp, que é completamente funcional: centramo-nos em funções, e não objetos, para fazermos nossa tarefa. Podemos nos perguntar então, afinal que tipo de linguagem é o R? Bem, na prática, assim como outras tantas linguagens, o R suporta um pouco de cada uma dessas linguagens. Essa mistura é resultado a própria história de criação do R. O R foi criado a partir da linguagem S, que era inteiramente orientada a objetos. Com a criação e subsequente transformação do R, contudo, esta linguagem acabou suportando vários tipos de paradigmas distintos.

Dois termos deste debate todo são importantes aqui: **objetos** e **funções**. Embora trabalharemos com funções somente em aulas posteriores, é útil entendermos agora o básico da diferença entre funções e objetos. **Objetos**, como o próprio nome diz, são elementos que guardamos em algum lugar. Objetos são os componentes que realmente estamos interessados, como por exemplo, um banco de dados particular. Objetos possuem dois componentes: seu nome e seu valor correspondente. Um nome, como seria de se imaginar, é a forma pela qual chamamos um objeto. Já seu valor é o que este nome representa: um número, um vetor de números, um banco de dados, uma lista, entre outros tipos.

Já **funções** são um conjunto de declarações que desempenham dada tarefa. Funções são, portanto, a forma pela qual operamos nossos objetos. Funções possuem quatro elementos: um nome, seus argumentos, o corpo da função (sua expressão) e o valor a ser retornado. Pensemos em uma função bem básica: a soma. Pensem em uma soma genérica, por exemplo, "2 + 3". Primeiramente, temos seu nome, que em português claro seria exatamente este: soma. Também é claro saber os argumentos, "2" e "3" (ou qualquer outro valor que desejássemos). O "corpo" seria a fórmula que chegaria ao resultado, isto é, "2 + 3". Por fim, o valor retornado seria o resultado da nossa função, que no caso seria "5". Várias funções são integradas ao próprio R. O comando que vimos na aula passada, help(), por exemplo, é uma função. Pacotes do R, similarmente, também podem ser entendidos como um conjunto de funções.

Armazenando Objetos no R

Tanto para objetos quanto para funções vimos que precisamos nomeá-los para que fiquem registrados no ambiente. Mas como criamos nomes? A forma preferida de se dar nomes no R é usando os símbolos "<-". Há outras maneiras de se dar nomes, mas como veremos adiante, não são práticas muito recomendadas. Na direção dessa nossa flecha (ou seja, à esquerda) damos um nome, sem espaço, ao objeto que criamos. No outro sentido (à direita) escolhemos o objeto que queremos guardar. Suponha, por exemplo, que eu queira guardar o número 2. Eu poderia criar este objeto da seguinte forma:

```
numero_dois <- 2
```

É bem fácil, não?

Caso se guarde uma expressão, o valor que o R retorna é o resultado da mesma. Por exemplo, se criarmos o objeto:

```
minha_soma <- 2 + 3
```

O valor guardado será a soma de 2 + 3, ou seja, 5.

Praticando

1.4) Faça uma soma qualquer e guarde os resultados dentro de um objeto. Reporte o resultado guardado no objeto.

Que nome de objeto devo escolher?

"*There are only two hard things in Computer Science: cache invalidation and naming things.*" – Phil Karlton

Um erro comum de programadores iniciantes é não dar a devida atenção aos nomes que dão a seus objetos. Nomear objetos adequadamente se relaciona à questão da transparência de nossos scripts que discutimos em nossa aula anterior. Nomes bem criados facilitam a leitura do leitor, fazendo com que você se comunique com ele de maneira mais clara.

Dessa forma, o objeto

```
minha_soma <- 2 + 3
```

possui um bom nome, enquanto que o objeto

```
testeaaa <- 2 + 3
```

pouco nos diz acerca de nosso objeto.

Outra dica é que nomes devem ser suficientemente curtos ao mesmo tempo que sejam suficientemente claros. Um nome curto facilita nosso trabalho enquanto programadores, visto que muito provavelmente teremos de chamar um dado objeto inúmeras vezes ao longo de um trabalho. Desta forma, o objeto:

```
minha_soma <- 2 + 3
```

é curto o suficiente, sem que isto prejudique a compreensão do que queremos ao criar este objeto.

Já o objeto:

```
esta_é_a_minha_soma_de_dois_mais_tres <- 2 + 3
```

é extremamente verborrágico, dificultando nosso trabalho ou mesmo a compreensão do leitor.

Por fim, o objeto:

```
s <- 2 + 3
```

Pode ser perigosamente curto. Uma dica, portanto, seria acompanhar a criação deste objeto com algum tipo de comentário (vocês se lembram como inserimos comentários em scripts e no RMarkdown?). O seguinte código, portanto, ficaria mais fácil de ler:

```
s <- 2 + 3 # minha soma
```

Uma última dica de nomeação de objetos seria quanto ao estilo do nome. Como vocês podem ver, pessoalmente eu gosto de usar o sinal o underscore (_) como espaço entre palavras. Outros autores utilizam o sinal de ponto (.), enquanto outros capitalizam as letras de cada palavra. Não há problema de haver diferenças de estilos, é normal que se varie de pessoa a pessoa. Problemas começam surgir, contudo, quando utilizamos diversos estilos em um mesmo projeto. Isto dificulta a vida do próprio programador, que tem de se recordar a cada instante qual o separador que criou para cada nome. Assim sendo, embora não seja necessário pensar em um estilo que você passará a usar para o resto da sua vida, é importante pensar que, para cada projeto, fica mais simples em usar um estilo único.

Outras formas de se dar nomes (e por que não usá-las!)

Vimos até agora que para nomearmos objetos usamos a expressão "<-". Mas esta não é a única forma de nomearmos objetos. Uma forma alternativa de se nomear objetos é utilizando o sinal de igual (=). Por

exemplo, comparem as seguintes expressões:

```
x <- 2
```

```
x = 2
```

Como vocês podem ver, ambas as expressões geram o mesmo resultado: um objeto chamado "x" cujo valor é "2". Como somente usamos uma tecla para escrever "=" e duas teclas para escrever "<-" pode parecer meio tentador usar somente o sinal de igual para nomear objetos. A comunidade do R, no entanto, costuma criticar tal uso. Se você buscar essa discussão em algum outro lugar, verá que usualmente se é recomendado utilizar "<-" para nomear objetos e "=" para nomear parâmetros de funções. Há, contudo, um problema adjacente ao sinal "=" que vai além dessa questão de estilo. O problema principal de se utilizar o sinal de igual é que ele é pouco claro acerca da direção da nomeação. Para deixar mais clara tal afirmação, notem que, embora raramente utilizado, a expressão "->" também pode ser utilizada para criar um objeto. Comparem agora as seguintes expressões:

```
x <- 1
```

```
2 -> y
```

```
x = y
```

Conseguem ver o problema de se usar o sinal de igual agora? A primeira expressão é bem clara, o valor "1" é guardado no objeto cujo nome é "x". O mesmo vale para o segundo caso, o valor "2" é guardado no objeto cujo nome é "y". Mas o que acontece no terceiro caso? É o valor "x" que é guardado com o nome "y", ou é o valor "y" que é guardado com o nome "x"? Por este motivo que preferimos usar a expressão "<-" para criar objetos.

Praticando

1.5) Veja por você mesmo o que acontece quando criamos os objetos:

```
x <- 1

2 -> y

x = y
```

O que acontece no último caso? Qual elemento (x ou y) virou o nome do objeto? E qual elemento (x ou y) virou o valor do objeto?

Removendo objetos

Para remover objetos utilizamos a versão rm(). O nome dessa função vem exatamente de "remove" (lembra do que falamos de facilidade de escrita versus a facilidade de se reconhecer algo que criamos pelo nome?). Caso queiramos remover o objeto "x", por exemplo, escrevemos:

```
x <- 1

rm(x)
```

Podemos também remover mais de um objeto por vez:

```
x <- 1

y <- 2

rm(x,y)
```

Mas tomem cuidado - uma vez que um objeto foi removido, não há como restaurá-lo! É importante que se salve os objetos, portanto, se vocês acham que continuarão a utilizá-lo. Por outro lado, isto mostra outra vantagem de se anotar os códigos que utilizamos (via script ou RMarkdown): caso anotemos passo-a-passo do que fizemos, fica muito mais fácil de reproduzir algo que eventualmente removemos por acidente.

Classes de Objeto

Objetos podem ser subdivididos ainda em suas "classes". Cada classe de objeto possui uma propriedade específica. Apresentamos aqui as classes principais de objetos que o R suporta:

- **Vector**. Vectors são vetores, isto é, uma sequência de dados de quaisquer tipos. Podem ser subcategorizados ainda em outros tipos de vetores (numeric, character).
- **Numeric**. Vetores numéricos (vocês devem se lembrar destes vetores em aulas de matemática...).
- **Logical**. Vetores booleanos, ou seja, de valores verdadeiros ou falsos.
- **Character**. Vetor de caracteres. Por excelência, caracteres são as letras de nosso alfabeto, espaço, sinais de pontuação e demais sinais gráficos. Números também podem ser representados enquanto caracteres: 2 é um número, enquanto "2" é um caracter.
- **Factor**. Vetor de fatores. Classe conceptualmente parecida com caracteres, fatores são vetores que assumem uma quantidade limitada de valores (p.ex., meses do ano, gênero etnia). Usualmente utilizado para variáveis qualitativas, embora também possamos usar vetores de caracteres para isso.
- **Ordered Factor**. Vetor de fatores ordenados. Similar ao vetor de fatores, porém os fatores contém uma ordem hierárquica. P.ex: nível educacional, faixa salarial, escalas de Likert.
- **Date**. Vetor de datas do calendário. Utiliza o formato de datas internacional ISO 8601, p.ex.: "2019-12-31".
- **Matrix**. Matriz simples bidimensional. Todas as colunas de uma matriz devem conter a mesma mesma de variável (numérica, lógica, caracter).
- **Array**. Matriz multidimensional.
- **Data Frame**. Banco de dados. Uma forma de se pensar nos data frames é que são um conjunto de vetores, com cada coluna representando um vetor distinto. Diferentemente das matrizes, aceita que cada vetor seja de uma classe distinta.
- **List**. Lista. Listas no R são objetos extremamente versáteis, podendo-se incluir objetos de classes inteiramente distintas: vetores numéricos, bancos de dados, vetores lógicos... A principal vantagem da lista sobre o data frame, portanto, é que o dado não precisa ser retangular. Como veremos em nossa aula de *webscraping* algo que é muito comum termos são listas dentro de listas. Pode parecer um pouco confuso agora, mas veremos futuramente como trabalhar com elas.

Para saber qual é a classe de um objeto, utilize a função class(). Por exemplo:

```
x <- 10000

class(x)
```

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

```
y <- TRUE

class(y)
```

```
## [1] "logical"
```

Vetores

Até agora trabalhamos com objetos de apenas um único elemento. Isto pode parecer um pouco entediante, principalmente quando o que queremos aprender é como analisar dados complexos. Passaremos agora a trabalhar com outros tipos de objetos.

Um tipo de objeto extremamente comum de ser trabalhado no R são vetores. Vetores, como o nome indica, são uma coleção de elementos. Assim como nossos objetos de um único elemento, vetores podem ser de diversas classes (numérica, lógica, caracter, etc). Para criarmos um vetor, usamos a função c(). Para quem tiver curiosidade, o "c" dessa função vem de "combine", isto é, estamos combinando diversos elementos. Um exemplo de vetor seria:

```
x <- c(1, 2, 3.98, 4.89, 5, 6.676127, 7, 8, 9, 10.2)
```

Ou ainda:

```
y <- c(TRUE, TRUE, FALSE, FALSE, TRUE, TRUE)
```

Por fim:

```
frutas <- c("maçã", "pera", "banana")
```

Para o caso de vetores numéricos, por vezes não queremos escrever cada componente de um vetor individualmente. Nestes casos, podemos usar o operador ":". Este operador gera um vetor de números inteiros de um número até outro (inclusive). Por exemplo:

```
x <- c(1:99804)
```

Gera os números de 1 a 99804.

Note que não precisamos escrever c() para criar um vetor quando utilizamos o operador ":":

```
y <- 1:99804
```

Praticando

1.6) O que acontece quando criamos um vetor usando somente c( )? Teste você mesmo.

1.7) O que acontece quando criamos um vetor usando c(" ")?

1.8) O que acontece quando combinamos elementos de classes distintas em um único vetor? A forma que o R funciona este problema é hierarquizando os tipos de classes. Descubra a classe de cada um dos objetos descritos abaixo:

```
x <- c(2, 3, TRUE)

y <- c(TRUE, "banana")

z <- c(2, "maçã")

w <- c(2, TRUE, "uva")
```

Operações com Vetores

Assim como objetos de um único elemento, é possível fazer operações aritméticas com vetores. Podemos fazer esta operação com um único elemento, por exemplo:

```
meu_vetor <- c(1:20)
```

```
meu_vetor + 5
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
```

Como também podemos fazer operações entre vetores:

```
meu_vetor <- c(1:20)

meu_outro_vetor <- c(2:21)

meu_outro_vetor - meu_vetor
```

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

Caso os vetores tenham tamanhos distintos, o R retornará um aviso de erro e fará a operação até o último elemento do menor vetor:

```
vetor_pequeno <- c(1:5)

vetor_grande <- c(5:20)

vetor_pequeno + vetor_grande
```

```
## Warning in vetor_pequeno + vetor_grande: longer object length is not a multiple
## of shorter object length
```

```
##  [1]  6  8 10 12 14 11 13 15 17 19 16 18 20 22 24 21
```

Manipulando Vetores

Usamos os colchetes ("[]") para identificar elementos de um vetor. A forma mais básica de se usar os colchetes seria identificar a posição de um elemento desejado dentro de um vetor. Por exemplo, se quisermos saber o quinto elemento de um vetor qualquer, podemos fazer:

```
x <- c(4:70)

x[5]
```

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

E o R retornará o valor desejado (no caso, 8).

Mas não é só isso que podemos fazer com os colchetes. Podemos, por exemplo, inserir operações. Caso queiramos que saber os elementos de um vetor que sejam maiores do que 5, podemos escrever:

```
x <- c(1:10)

x[x>5]
```

```
## [1]  6  7  8  9 10
```

E o R retorna somente os números que cumprem esta condição.

é possível também salvar esta operação em um novo objeto, fazendo:

```
x <- c(1:10)

y <- x[x>5]
```

Bancos de Dados

Outro tipo extremamente comum de objeto que trabalhamos no R são bancos de dados. Assim como vetores são um conjunto de elementos únicos, bancos de dados podem ser pensados como um conjunto de vetores. Assim como as matrizes, chamamos os data frames de "dados retangulares". Isto se dá porque cada uma das colunas do banco de dados possui um mesmo comprimento.

A forma mais básica de se criar um banco de dados é através do comando data.frame(). Para criar devidamente nosso data frame, precisamos dar nome para cada uma das variáveis (colunas) que criamos. Por exemplo:

```
meus_dados <- data.frame(
  frutas = c("banana", "maçã", "uva", NA),
  numeros = c(1, 2, 3, 4),
  checagem = c(TRUE, FALSE, TRUE, TRUE)
)
```

Note que aqui devemos usar o sinal de igual ("=") para nomear as colunas dentro de nosso data frame. Note também que na variável "frutas" colocamos um sinal de NA no fim. Como vimos, em dados retangulares não pode haver colunas de tamanhos distintos, adicionamos NA para garantir o tamanho correto de nossos dados.

Uma das vantagens do R é que ele já vem com alguns bancos de dados enbutidos. Com isso, podemos fazer alguns testes em nossas aulas sem ter que nos preocuparmos com a disponibilidade de bancos. O banco "mtcars", por exemplo, é um desses data frames. Para carregá-lo apenas precisamos fazer:

```
data("mtcars")
```

Note o uso de aspas. Ou então podemos carregá-lo em um objeto de nome de nossa preferência, fazendo:

```
dados <- mtcars
```

Note que desta vez não utilizamos aspas.

Praticando

1.9) O que aconteceria se carregássemos o mtcars usando aspas? Teste você mesmo. Descreva como o R interpretou tal comando.

```
dados <- "mtcars"
```

Note que também é possível carregar bancos de dados através de certos pacotes. O pacote Zelig é um deles. Vamos instalá-lo:

```
install.packages("Zelig", repos = "http://cran.us.r-project.org")
```

Há várias fontes de bancos de dados públicos. Para um conjunto de dados abertos brasileiros, é possível acessar o dados.gov.br. Já o Harvard Databse é um renomado hospedador de bancos de dados da Universidade de Harvard. Nos últimos anos, a Google lançou uma ferramenta própria de busca de bancos de dados, o Google Dataset Search. Por fim, recomendamos que chequem a conta de GitHub do awesomedata para um compilado de dados de vários países.

Visualizando data frames

Olhando para a aba Environment, vemos que o objeto "dados" que criamos possui 32 observações e 11 variáveis. Até agora não vimos exatamente, contudo, como aprender mais acerca dos dados que carregamos.

Há várias formas de se visualizar os dados de um data frame. A função head() serve para vermos o começo (ou seja, a "cabeça") dos dados. Vejamos como é o banco mtcars.

```r
dados <- mtcars

head(dados)
```

```
##                    mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

Por default o comando head() imprime as 5 primeiras linhas do dataframe. Podemos mudar isso diretamente incluindo o número de linhas que desejamos imprimir. Por exemplo:

```r
head(dados, 10)
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

Já o comando tail() mostra o fim do data frame (ou seja, seu "rabo"). Usando o comando, temos:

```r
tail(dados)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

E também podemos fazer:

```r
tail(dados, 12)
```

```
##                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Toyota Corona    21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger 15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin      15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28       13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird 19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9        27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2    26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa     30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L   15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino     19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora    15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E       21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

Já o comando str() mostra automaticamente a classe de cada uma das variáveis (colunas) de nosso data frame, além de também servir como o comando head() de mostrar os primeiros valores do banco.

```
str(dados)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Podemos também manualmente olhar o tamanho de nossos dados. A função nrow() retorna o número de linhas (rows) do banco:

```
nrow(dados)
```

```
## [1] 32
```

Similarmente, o comando ncol() retorna o número de colunas:

```
ncol(dados)
```

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

O comando dim() mostra as dimensões do banco, i.e., número de linhas e colunas:

```
dim(dados)
```

```
## [1] 32 11
```

Se o comando dim() mostra as dimensões de um objeto, o que aconteceria se usássemos dim() em um vetor? Podemos testar isso:

```
vetor <- 1:10

dim(vetor)
```

```
## NULL
```

Como vocês podem ver, o R não entende vetores como sendo objetos de dimensão única, mandando o valor nulo (NULL) como resposta. Para vermos o comprimento de um vetor usamos a função length():

```
length(vetor)
```

```
## [1] 10
```

Por fim, a forma mais completa de se olhar para um data frame é através da função View()

```
View(dados)
```

Como vocês podem ver, o banco mtcars possui linhas nomeadas, mas isso nem sempre é assim. Na prática, nada muda se a linha é nomeada ou enumerada.

Mudando de paradigma: evite ficar dependente da função View()!

Normalmente, seria de se pensar que devemos olhar a todo instante para nosso banco de dados caso queiramos trabalhar com ele. Isso parece ainda mais natural para uma pessoa que está acostumada a fazer análises usando programas como o Excel ou o Access. Conforme vocês forem se acostumando a mexer com dados, vocês aprenderão que essa prática é na verdade muito contra-producente! Claro, quando abrimos um banco de dados pela primeira vez, desejamos usar a função View() e outras similares para checar nossos dados. Mas não precisamos ficar perdendo tempo olhando para os dados a cada instante para trabalhar com eles. Devemos usar essas ferramentas de visualização de dados, portanto, somente ocasionalmente.

Manipulando data frames

Veremos mais propriamente como manipular data frames em nossas aulas seguintes. Isto não impede, contudo, de aprendemos o mínimo possível a partir de agora.

Vimos agora pouco que podemos manipular elementos específicos de vetores usando colchetes. O que aconteceria caso fizéssemos o mesma para data frames? Vamos testar:

```
dados[5]
```

```
##                     drat
## Mazda RX4           3.90
## Mazda RX4 Wag       3.90
## Datsun 710          3.85
## Hornet 4 Drive      3.08
## Hornet Sportabout   3.15
## Valiant             2.76
## Duster 360          3.21
## Merc 240D           3.69
## Merc 230            3.92
## Merc 280            3.92
## Merc 280C           3.92
## Merc 450SE          3.07
## Merc 450SL          3.07
## Merc 450SLC         3.07
## Cadillac Fleetwood  2.93
## Lincoln Continental 3.00
## Chrysler Imperial   3.23
## Fiat 128            4.08
## Honda Civic         4.93
## Toyota Corolla      4.22
## Toyota Corona       3.70
## Dodge Challenger    2.76
## AMC Javelin         3.15
## Camaro Z28          3.73
## Pontiac Firebird    3.08
## Fiat X1-9           4.08
## Porsche 914-2       4.43
## Lotus Europa        3.77
## Ford Pantera L      4.22
## Ferrari Dino        3.62
## Maserati Bora       3.54
## Volvo 142E          4.11
```

Como pode se ver, ao usarmos o comando dados[5] o R imprimiu todos os elementos da 5ª coluna (cujo nome

é drat).

Como data frames são objetos de 2 dimensões, precisamos especificar ambass se quisermos que o R retorne apenas um único elemento. A ordem que fazemos, assim é dados[linha, coluna]. Por exemplo:

```
dados[5, 1]
```

```
## [1] 18.7
```

Retorna o elemento da 5ª linha e da 1ª coluna.

Outra forma de se referir a colunas em data frames é utilizando o cifrão ($). Escrevemos o nome de nosso objeto seguido do cifrão e o nome da coluna desejada. Por exemplo:

```
dados$hp
```

```
##  [1] 110 110  93 110 175 105 245  62  95 123 123 180 180 180 205 215 230  66  52
## [20]  65  97 150 150 245 175  66  91 113 264 175 335 109
```

Como vocês podem ver, ao escrevermos o símbolo de $, o próprio R cria uma caixa com sugestões dos nomes das variáveis. Caso vocês queiram olhar todos os nomes das variáveis de nosso banco, podemos usar a função names:

Podemos começar vendo o nome de cada uma de nossas variáveis (isto é, das colunas). Usando a função names() temos:

```
names(dados)
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

Caso não queiramos nos referis às colunas pela sua posição, podemos misturar o sinal de $ e os colchetes. Por exemplo, podemos escrever:

```
dados$hp[20]
```

```
## [1] 65
```

Que nos retorna o 20º elemento da coluna "hp".

Assim como vetores, podemos também inserir argumentos lógicos dentro dos colchetes quando manipulamos data frames. Por exemplo:

```
dados_2 <- dados[dados$drat > 3,]
```

Temos de prestar atenção em algumas coisas aqui. Em primeiro lugar, observe que criamos um segundo banco de dados (dados_2) antes de manipular nossos dados. Isso é uma prática muito útil quando manipulamos dados. Por vezes, queremos reverter nossas manipulações a um estado anterior. Mudar o objeto que estamos trabalhando evita, assim, maiores dores de cabeça no futuro. Em segundo lugar, note que escrevemos duas vezes o nome de nosso objeto. Fora dos colchetes, escrevemos o data frame que quereremos manipular; dentro dele, escrevemos a operação que queremos fazer. No caso, queremos que a variável "drat" do objeto "dados" seja maior que 3. Para realizar este comando, escrevemos, portanto, "dados$drat $>$ 3". Por fim, o sinal da vírgula é utilizado para manter a dimensionalidade do objeto. Observe que o valor lógico que inserimos (dados$drat $>$ 3) se refere aos valores possíveis da linha. Logo, temos de descrever as operações que fazemos na coluna. Ao adicionar o "," e terminar nossos colchetes, portanto, estamos dizendo que queremos um valor específico para as linhas, e que queremos que as colunas se mantenham como estão.

Se isto tudo parece confuso, não se preocupe. Esta forma de manipular dados pode realmente ser um pouco trabalhosa. Antes da introdução do tidyverse, era assim que os programadores em R manipulavam dados em geral. É comum se referir à esta manipulação por colchetes de "R base". Nas próximas aulas trabalharemos com manipulação de dados com o tidyverse. Vocês poderão verificar quão mais simples a manipulação pode ser quando usamos este conjunto de pacotes.

Analisando Data Frames

Um comando bem útil é o summary(). Usamos tal comando para visualizar um sumário de nossas variáveis. Por exemplo:

```
summary(dados)
```

```
##       mpg             cyl             disp             hp       
##  Min.   :10.40   Min.   :4.000   Min.   : 71.1   Min.   : 52.0  
##  1st Qu.:15.43   1st Qu.:4.000   1st Qu.:120.8   1st Qu.: 96.5  
##  Median :19.20   Median :6.000   Median :196.3   Median :123.0  
##  Mean   :20.09   Mean   :6.188   Mean   :230.7   Mean   :146.7  
##  3rd Qu.:22.80   3rd Qu.:8.000   3rd Qu.:326.0   3rd Qu.:180.0  
##  Max.   :33.90   Max.   :8.000   Max.   :472.0   Max.   :335.0  
##       drat             wt             qsec             vs        
##  Min.   :2.760   Min.   :1.513   Min.   :14.50   Min.   :0.0000  
##  1st Qu.:3.080   1st Qu.:2.581   1st Qu.:16.89   1st Qu.:0.0000  
##  Median :3.695   Median :3.325   Median :17.71   Median :0.0000  
##  Mean   :3.597   Mean   :3.217   Mean   :17.85   Mean   :0.4375  
##  3rd Qu.:3.920   3rd Qu.:3.610   3rd Qu.:18.90   3rd Qu.:1.0000  
##  Max.   :4.930   Max.   :5.424   Max.   :22.90   Max.   :1.0000  
##        am              gear            carb      
##  Min.   :0.0000   Min.   :3.000   Min.   :1.000  
##  1st Qu.:0.0000   1st Qu.:3.000   1st Qu.:2.000  
##  Median :0.0000   Median :4.000   Median :2.000  
##  Mean   :0.4062   Mean   :3.688   Mean   :2.812  
##  3rd Qu.:1.0000   3rd Qu.:4.000   3rd Qu.:4.000  
##  Max.   :1.0000   Max.   :5.000   Max.   :8.000
```

Também podemos analisar apenas uma de nossas variáveis, se assim preferirmos.

```
summary(dados$cyl)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##   4.000   4.000   6.000   6.188   8.000   8.000
```

O que implica que o comando também funciona com vetores:

```
vetor <- 1:10

summary(vetor)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
##    1.00    3.25    5.50    5.50    7.75   10.00
```

Caso seja desejado apenas uma estatística pontual, e não o sumário completo, podemos usar a função apropriada:

```
sum(dados$cyl)
```

```
## [1] 198
```

```
mean(dados$cyl)
```

```
## [1] 6.1875
```

```
sd(dados$cyl)
```

```
## [1] 1.785922
```

```r
var(dados$cyl)
```

```
## [1] 3.189516
```

```r
median(dados$cyl)
```

```
## [1] 6
```

```r
max(dados$cyl)
```

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

```r
min(dados$cyl)
```

```
## [1] 4
```

```r
quantile(dados$cyl, probs = c(0, 0.25, 0.5, 0.75, 1))
```

```
##   0%  25%  50%  75% 100%
##    4    4    6    8    8
```

vejamos o que acontece quando usamos o summary() para descrever vetores de caracteres:

```r
animais <- c("cachorro", "gato", "gato", "gato", "cachorro", "cachorro",
             "papagaio")


summary(animais)
```

```
##    Length     Class      Mode
##         7 character character
```

Não é muito útil, não é mesmo?

Para analisar variáveis de caracter usamos a função table(). Esta função conta quantas vezes cada elemento único aparece em dado vetor.

```r
table(animais)
```

```
## animais
## cachorro     gato papagaio
##        3        3        1
```

Podemos também facilmente fazer tabelas de dupla entrada. Instalemos o pacote "mosaicData" para testar tabelas de dupla entrada no banco de dados de fumantes (Whickham) encontrado neste pacote.

```r
install.packages("mosaicData", repos = "http://cran.us.r-project.org")
```

```
##
## The downloaded binary packages are in
##  /var/folders/5q/q64vd8n119q70lj5fhs8x3d40000gn/T//Rtmp9qKhAQ/downloaded_packages
```

```r
library(mosaicData)
```

```r
data(Whickham)

table(Whickham$smoker, Whickham$outcome)
```

```
##
##       Alive Dead
##   No    502  230
##   Yes   443  139
```

Importando e Exportando Objetos

O R possui várias maneiras diferentes de abrir e salvar objetos. Há dois tipos de arquivos nativos do R: arquivos .rmd e arquivos .RData. Podemos salvar nosso banco usando:

```
save(dados, file = "meus_dados.RData")


saveRDS(dados, file = "meus_dados.rds")
```

Note que para utilizar ambas as funções escrevemos o nome do objeto que queremos salvar (dados) e o nome que queremos que o arquivo tenha ("meus_dados").

Similarmente, podemos importar esses arquivos para nosso ambiente da seguinte maneira:

```
load("dados") #abre o arquivo RData

dados <- readRDS("meus_dados.rds") #abre arquivo .rds
```

É bem possível que você esteja trabalhando com pessoas que não tenham R instalado na própria máquina. Na tabela abaixo fazemos um resumo das funções de importação e exportação de dados:

Nome da função	Pacote	Tipo	Explicação
load()	R base	Importação	Importa arquivos .RData
save()	R base	Exporatção	Exporta arquivos .RData
readRDS()	R base	Importação	Importa arquivos .rds
saveRDS()	R base	Exportação	Exporta arquivos .rds
read.table()	R base	Importação	Importa arquivos .txt
write.table()	R base	Exportação	Exporta arquivos .txt
read.csv()	R base	Importação	Importa arquivos .csv separados por ","
write.csv()	R base	Exportação	Exporta arquivos .csv separado por ","
read.csv2()	R base	Importação	Importa arquivos .csv separado por ";"
write.csv2()	R base	Exportação	Exportação arquivos .csv separado por ";"
read_dta	haven	Importação	Exporta arquivos .dta (Stata)
write_dta	haven	Exportação	Exporta arquivos .dta (Stata)
read_sav	haven	Importação	Exporta arquivos .sav (SPSS)
write_sav	haven	Exportação	Exporta arquivos .sav (SPSS)
read.xlsx	openxslx	Importação	Importa arquivos .xlsx (Excel)
write.xlsx	openxslx	Exportação	Exporta .xlsx (Excel)

Caso vocês estejam se perguntando, a diferença entre o write.csv() e write.csv2() se dá pelo encoding do csv. O write.csv() usa o formato americano de arquivos de csv, ou seja, separa as colunas por vírgula. Já o write.csv2() usa o formato que estamos acostumados, separado por ponto-e-vírgula.

Para cada função de importação/exportação de dados do R base, o tidyverse possui um equivalente. Por exemplo, a versão tidyverse do read.csv() é o read_csv(). Essas funções são análogas, então pouco importa qual utilzamos.

Cobrimos aqui os tipos de dados mais comuns de serem trabalhados no dia-a-dia. Caso vocês precisem importar/exportar arquivos em outros formatos, lembrem-se que sempre podem usar a função help.search() para aprender novas funções.

Praticando

1.10) Crie um data frame igual à tabela de importação/exportação que fizemos. Salve os arquivos em cada um dos formatos que acabamos de aprender.

Aonde meus dados são salvos?

Acabamos de aprender como importar e exportar dados no R. Mas qual o diretório que o R salva os dados? Por padrão, o R trabalha no diretório "Documents"/"Documentos". Para checar qual o diretório atual que o R está trabalhando utlizamos a função getwd():

```
getwd()
```

Já para alterar o diretório, utilizamos a função setwd(). Por exemplo:

```
setwd("C:/Users/Fulano/Documents/Projects/meus_dados")
```

Note que usamos aqui a barra simples (/), e não a barra invertida (\), como é de costume em sistemas Windows. Observe também que devemos escrever o nome do diretório em aspas.

Fix the Code

2.1)

```
numeros <- c(1 2 3 4)
```

2.2)

```
animais <- c("cachorro" "gato" "papagaio" "coelho")
```

2.3)

```
animais <- C("cachorro", "gato", "papagaio", "coelho")
```

2.4)

```
Head(mtcars)
```

2.5)

```
str(Mtcars)
```

2.6)

```
dim[mtcars]
```

2.7)

```
nomes(mtcars)
```

2.8)

```
head(mtcars, x = 10)
```

2.9)

```
animais <- c("cachorro", "gato", "papagaio", "coelho")

numeros <- c(1,2,3,4,5,6,7,8,9,10)

dados <- data.frame(animais,numeros)
```

2.10)

```
dados <- c(1:100)

save("meus_dados.RData", file = dados)
```

Desafio

3.1) Crie um vetor de todos números inteiros (integers) maiores do que 1 e menores do que 1000000. Em seguida, crie um vetor com as mesmas características, contudo apenas números pares. Dica: obviamente não queremos que vocês digitem cada elemento um por um... Descubra como pular elementos em um vetor.