Programação em R

Julho, 2016

Introdução e Motivação

O que é o R?

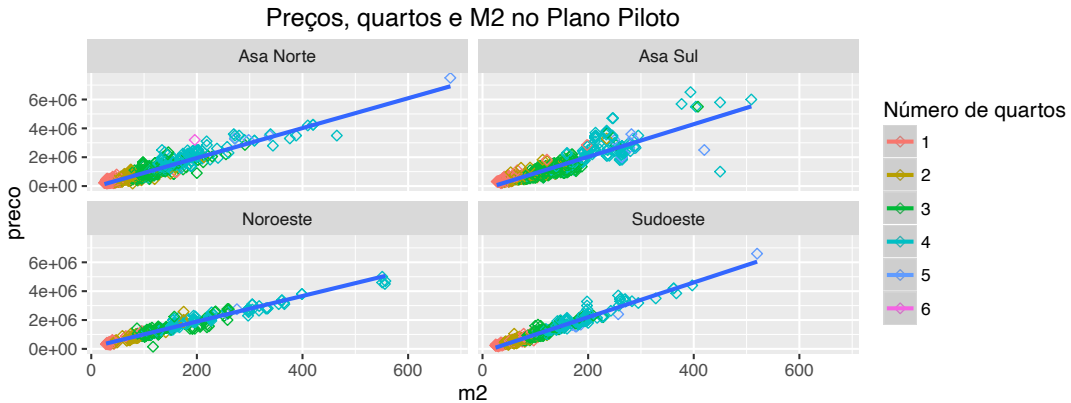
- O R é uma linguagem de programação com foco em análise de dados;
- Criado na Nova Zelândia por dois estatísticos: Ross Ihaka e Robert Gentleman;
  - Baseado na linguagem S, desenvolvida por John Chambers (e colegas) na Bell Laboratories;
- É uma linguagem de programação interpretada, voltada à interação dinâmica com os dados e modelos.

Por que o R?

- O R é gratuito e de código aberto;
- Compatível com todas as plataformas (Windows, Mac, Linux);
- É mais do que um software estatístico:
  - Ambiente que permite explorar dados interativamente; mas, à medida que a análise evolui, é uma linguagem de programação completa para desenvolver e automatizar soluções, desenvolver software (pacotes);
  - Ferramenta poderosa para manipulação, processamento, visualização e análise de dados, bem como simulações e modelagem estatísticas.
- Comunidade grande que contribui ativamente, com pessoas tanto do mercado (Google, AT&T, empresas de investimento e finanças) quanto da academia (professores de diversas universidades):
  - Mais de 8000 pacotes gratuitos e abertos para download.

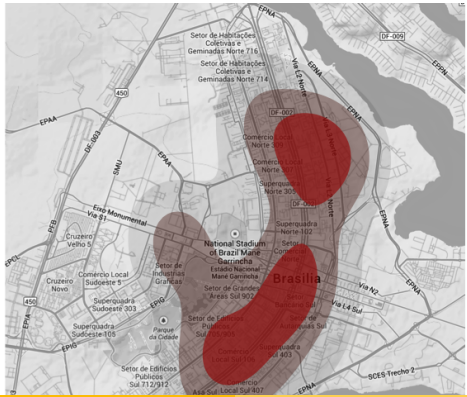
Alguns exemplos

Gráficos e estatística: gráfico de dispersão com preço contra metro quadrado por bairro, cor dos pontos de acordo com número de quartos e linha de regressão.



Alguns exemplos

Mapas e estatística: Webscraping de dados de roubos e download do mapa de Brasília no Google Maps; manipulação dos dados e construção de gráficos de calor geolocalizados.

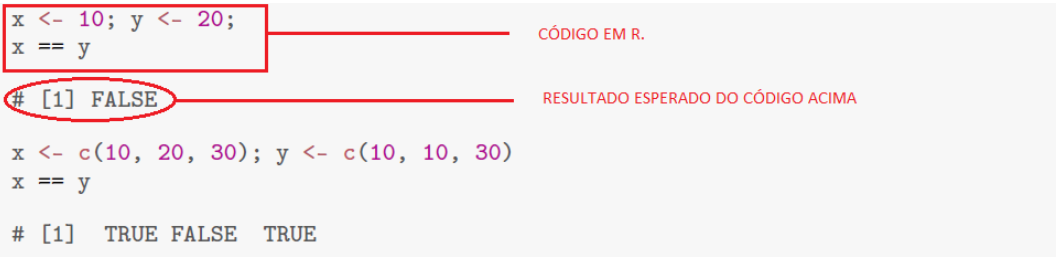


Alguns exemplos

Apresentações e documentos de suas análises: Integração com *markdown*, LaTeX, HTML + JavaScript e Word, o que permite a elaboração de *papers*, relatórios, apresentações de maneira rápida e conjugada à análise.
Esta e as demais apresentações deste curso, bem como as listas de exercícios, foram todas feitas no R.

Sobre os slides

Os slides deste curso apresentam códigos em R com o resultado esperado. Você deve **acompanhar ativamente**, digitando (ou, em último caso, copiando e colando) os códigos. Programação não se aprende somente olhando para a apresentação! Também serão realizados exercícios para consolidação do conteúdo.



Sobre os exercícios

- Durante a apresentação de cada tópico faremos exercícios entre os slides. Os exercícios serão feitos com tempo controlado e todas soluções serão apresentadas. Não se preocupe se não conseguir fazer todos durante o tempo controlado.
- Também teremos listas de exercícios. Como a turma é heterogênea, os exercícios da lista variam em nível de dificuldade e você não precisa conseguir fazer todos - a idéia é justamente ter exercícios tanto para consolidar conhecimentos quanto para desafiar um pouco quem for pegando o conteúdo mais rapidamente.
- Vocês receberão a resolução de todos os exercícios da lista também, entretanto, para resolução "ao vivo" em sala de aula, priorizaremos aqueles mais importantes, a depender do andamento das aulas.

R e Rstudio

Instalação

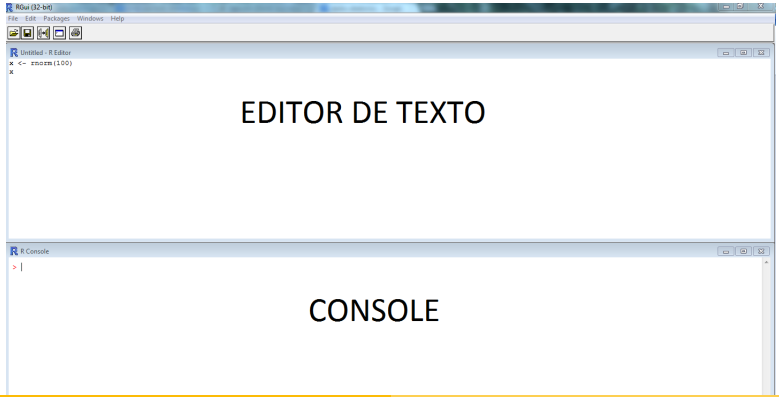
O R e RStudio já vêm com distribuições compiladas para Windows, Mac e Linux. A instalação é bastante fácil e em geral você apenas tem que seguir as instruções da tela. Para ter um ambiente completo de desenvolvimento no R, certifique-se de ter em sua máquina:

- a versão mais recente do R;
- a versão mais recente do RStudio (IDE que vamos usar);
- MikTex (Win) ou MacTex (Mac) para relatórios em LaTeX;
- RTools (Win) ou Xcode com command line tools (Mac), para criar pacotes, usar C++ etc.

Os slides a seguir tomarão como base, em geral, as teclas de atalho do Windows.

R Gui

Ao instalar o R, automaticamente também é instalada uma interface gráfica chamada R Gui. Abra o R no seu computador. A primeira tela que você verá é a do console. Abra também uma tela de editor de texto em "file" -> "new script".



R Gui

Escreva os seguintes comandos no editor de texto e aperte "ctrl+R".

```
1+1
```

O comando será enviado para o console. Agora faça o seguinte gráfico:

```
plot(1:10)
```

Uma nova tela será aberta com o gráfico. Saia do R Gui escrevendo q() no console ou apertando "alt+f4". Ele irá perguntar se você deseja salvar o trabalho e o script. Clique em "não" para os dois casos. Estamos saindo do R Gui pois iremos trabalhar no RStudio.

RStudio

Apesar de o R vir com uma interface gráfica bem interessante, existe um IDE (Integrated Development Environment - Ambiente de Desenvolvimento Integrado) chamado RStudio, com várias funcionalidades e gratuito. No decorrer de nossas aulas iremos utilizar somente o RStudio.
O RStudio tem algumas vantagens em relação ao R Gui:

- Highlight do código;
- Autocomplete;
- Match automático de parenteses e chaves;
- Interface intuitiva para objetos, gráficos, script;
- Projetos (com controle de versão);
- Interação com HTML, entre outras.

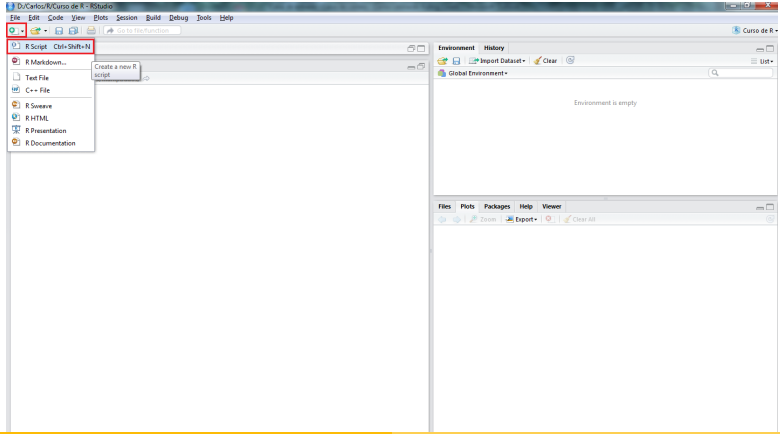
Projetos no RStudio

Vamos criar um projeto do RStudio para o nosso curso de R. Esta é uma maneira simples e fácil de gerenciar os scripts, dados e demais documentos relacionados a um projeto de R em que você esteja trabalhando. Paremos agora para:

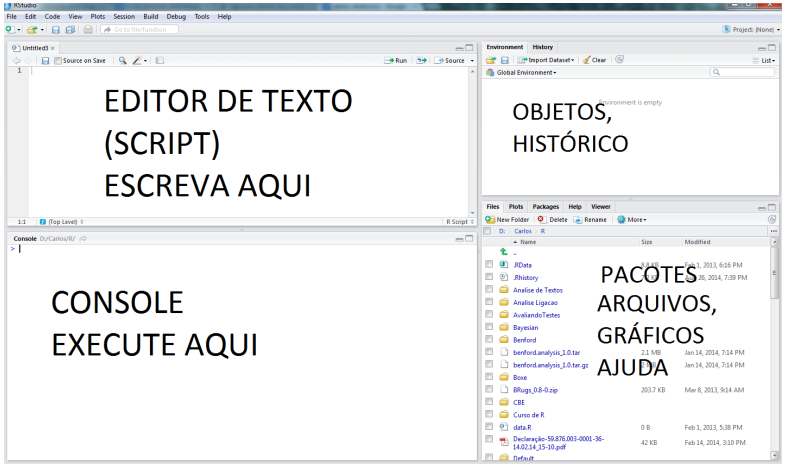
- Criar projeto de RStudio em uma pasta nova;
- Criar uma pasta de "Dados" para guardar algumas bases de dados utilizados em aula;
- Criar uma pasta de slides para guardar os slides (inclusive este) das aulas.

RStudio

Inicie um novo Script em "File" -> "New File" -> "New RScript". Você também pode fazer isso com **"CTRL + SHIFT + N"** ou acessando o botão abaixo.



RStudio


EDITOR DE TEXTO
(SCRIPT)
ESCREVA AQUI
OBJETOS,
HISTÓRICO
CONSOLE
EXECUTE AQUI
PACOTES
ARQUIVOS,
GRÁFICOS
AJUDA

RStudio

- **Script**: A tela superior esquerda do RStudio é o editor de texto onde você vai escrever seus Scripts. Ele possui code highlighting entre outras funcionalidades.
- **Console**: No canto inferior esquerdo fica o console. O console nada mais é do que uma seção aberta de R, em que os comando são executados.
- **Área de trabalho e histórico**: Ficam no canto superior direito. Os objetos criados na seção e o histórico dos comandos podem ser acessados ali.
- **Arquivos, Gráficos, Pacotes, Ajuda**: Ficam no canto inferior direito. Você pode explorar pastas e arquivos diretamente do RStudio na aba "Files"; os gráficos que forem feitos apareceram na aba "Plots". Os pacotes instalados em sua máquina estão listados em "Packages". As ajudas das funções aparecem em "Help". E o "Viewer" serve para visualização de imagens/páginas em HTML e JavaScript.

Comandos pelo Script

Acostume-se a escrever o código no Script ao invés de ficar escrevendo diretamente no console.
Escreva o código abaixo no Script:

```
1+1
```

E aperte "ctrl" + "enter". Isso envia o comando para o console e o resultado é exibido logo abaixo.

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

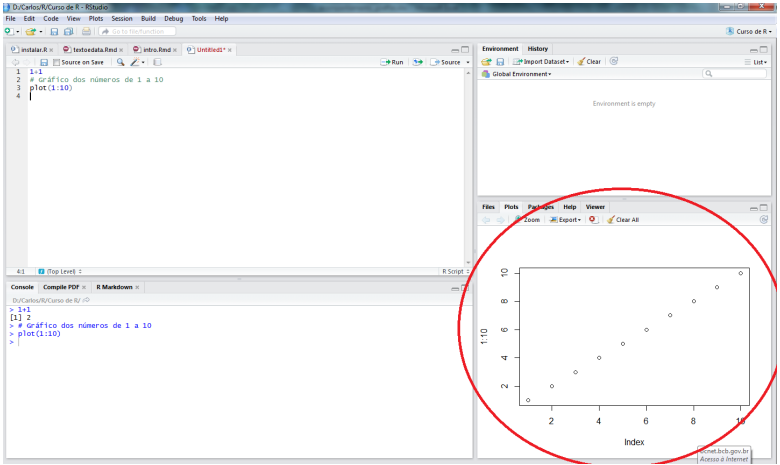
Comandos pelo Script

Agora escreva o seguinte código no Script.

```
# Gráfico dos números de 1 a 10
plot(1:10)
```

O primeiro comando *"# Gráfico dos números de 1 a 10"* é, na verdade, um comentário. Comentários em Scripts do R são seguidos do símbolo #, e tudo que estiver após # não será executado. É uma boa prática comentar seu código, para que ele seja de fácil manutenção, tanto para você mesmo quanto para outros colegas. O segundo comando é de um gráfico. Aperte "ctrl" + "enter" nas duas linhas. O gráfico aparecerá no canto inferior direito do RStudio.

Comandos pelo Script

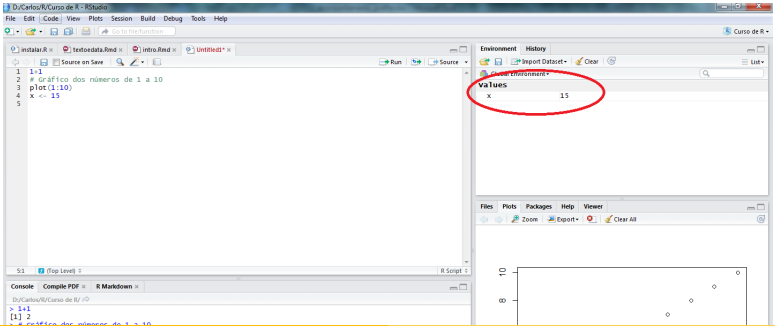


Comandos pelo Script

Agora digite o seguinte comando no editor e aperte "ctrl" + "enter":

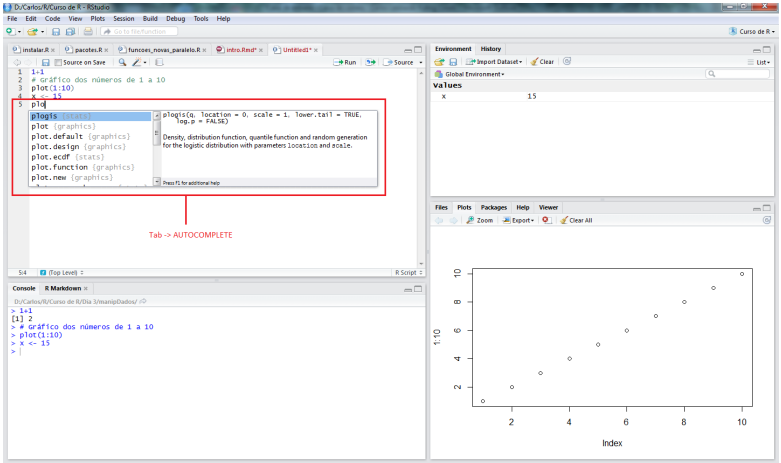
```
x <- 15
```

Atribuímos o valor 15 à variável x. Note que isso aparece no canto superior direito do RStudio.



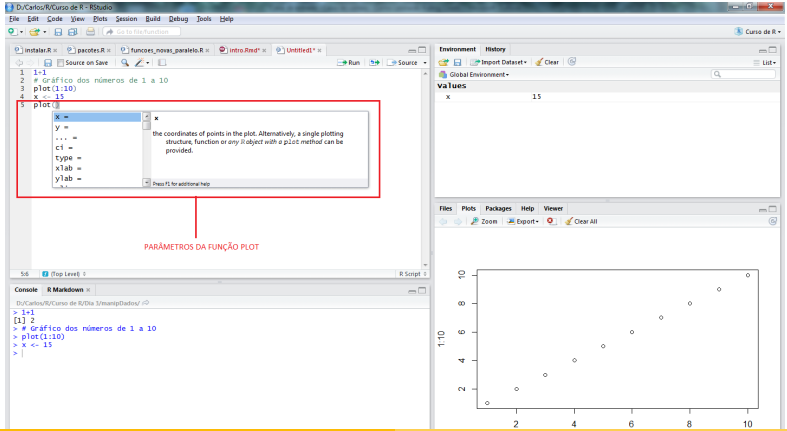
Comandos pelo Script

O RStudio tem autocomplete. Escreve apenas `plo` e aperte **Tab**.



Comandos pelo Script

Isso também funciona dentro da função, para vermos os parâmetros disponíveis. Escreva `plot()` e aperte **Tab**.



Comandos pelo Script

Outras teclas de atalho:

- **CTRL + 1**: Passa o cursor para o Script;
- **CTRL + 2**: Passa o cursor para o console;
- **SETA PARA CIMA (no console)**: acessa o histórico de comandos anteriores.

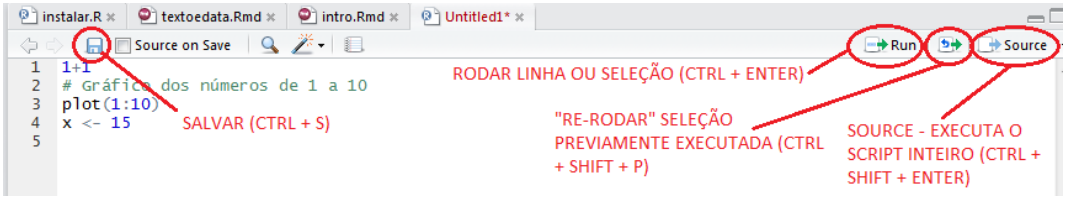
Abra um novo Script:

- **CTRL + ALT + SETA PARA ESQUERDA OU DIREITA**: Navega entre as abas de script abertas.

Comandos pelo Script

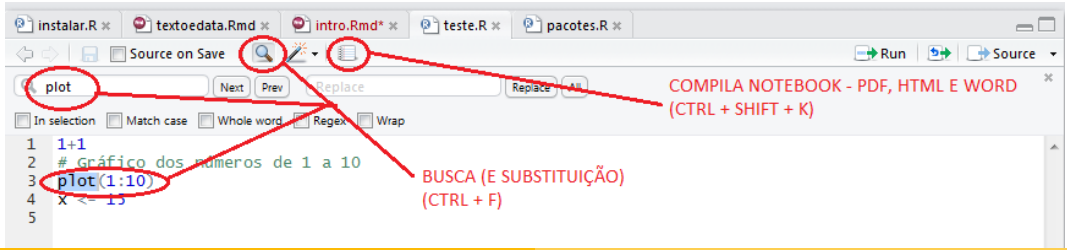
Outras teclas de atalho:

- **CTRL + SHIFT + P**: "Previous command", roda o último comando executado;
- **CTRL + SHIFT + ENTER**: "Source". Executa o Script inteiro;
- **CTRL + S**: Salva o Script;



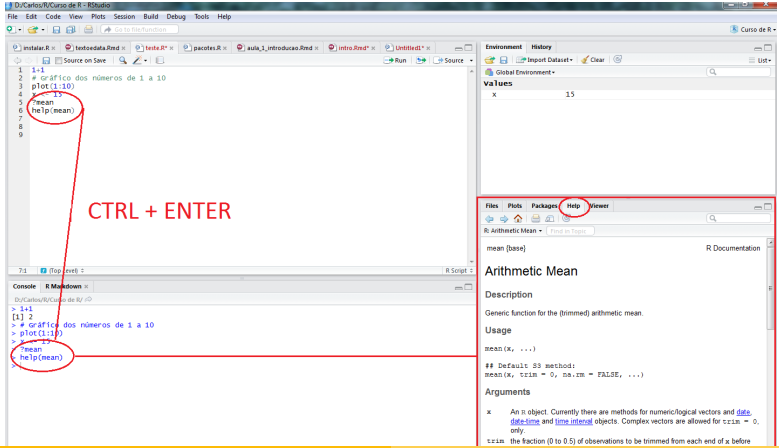
Comandos pelo Script

- **CTRL + L**: Limpa o console;
- **CTRL + F**: Busca (e substituição). Aceita REGEX;
- **CTRL + SHIFT + K**: Compila "Notebook" em PDF, HTML ou Word;
- **ALT + SHIFT + K**: Veja os outros atalhos.
- Veja mais dicas no Cheat Sheet do RStudio (`http://www.ibpad.com.br/wp-content/uploads/2016/04/rstudio-IDE-cheatsheet.pdf`).

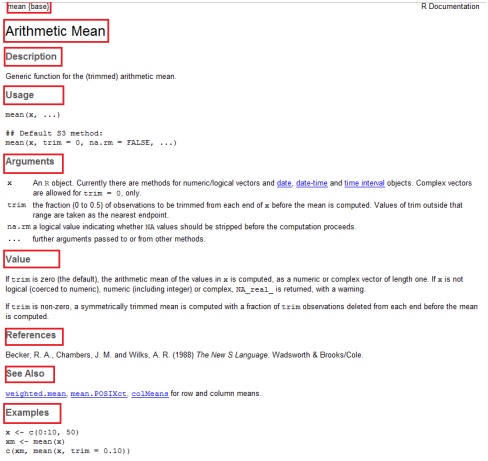


Ajuda - Função Específica

```
?mean; help(mean)
```



Ajuda - Função Específica



mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)

## Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)
```

Arguments

x	An R object. Currently there are methods for numeric/logical vectors and date, date-time and time interval objects. Complex vectors are allowed for trim = 0, only.
trim	the fraction (0 to 0.5) of observations to be trimmed from each end of x before the mean is computed. Values of trim outside that range are taken as the nearest endpoint.
na.rm	a logical value indicating whether NA values should be stripped before the computation proceeds.
...	further arguments passed to or from other methods.

Value

If trim is zero (the default), the arithmetic mean of the values in x is computed, as a numeric or complex vector of length one. If x is not logical (coerced to numeric), numeric (including integer) or complex, NA_real_ is returned, with a warning.

If trim is non-zero, a symmetrically trimmed mean is computed with a fraction of trim observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

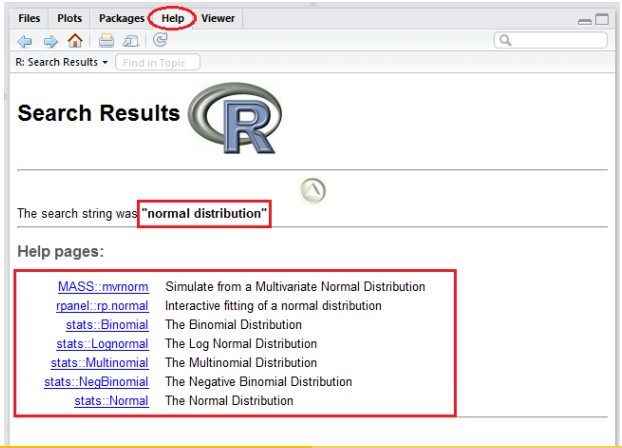
weighted.mean, mean.POSIXct, colMeans for row and column means.

Examples

```
x <- c(0:10, 50)
xm <- mean(x)
c(xm, mean(x, trim = 0.10))
```

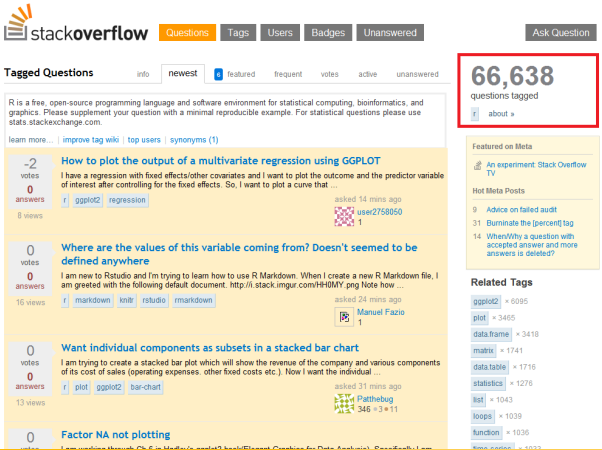
Ajuda - Busca por String

```
??"normal distribution"; help.search("normal distribution")
```



Ajuda - Internet

- StackOverflow (em Inglês e Português)



Ajuda - Internet

- **Grupos de e-mail**: www.r-project.org/mail.html
  - **R Help**: lista de e-mails principal, para dúvidas gerais.
  - **R Announce**: Lista de anúncios sobre desenvolvimento do R.
  - **R Packages**: Lista de anúncios sobre pacotes do R.
  - **R Devel**: Lista de discussão de desenvolvedores no R.
- **Blogs**
  - **R Bloggers**: consolidador de blogs sobre R (em inglês).
  - **Em Português**:
    - Análise Real (`http://analisereal.com`) com livro em elaboração (`http://analisereal.com/introducao-a-analise-de-dados-com-r/`) ;
    - Sociais e Métodos (`https://sociaisemetodos.wordpress.com`);
    - Dados Aleatórios (`http://www.dadosaleatorios.com.br/`) entre outros.

Pacotes

A principal forma de distribuição de códigos no R é por meio de pacotes. Um pacote pode ser entendido como um conjunto de códigos auto-contido que adiciona funcionalidades ao R.
Para carregar um pacote, use a função `library()`.
Ao carregar um pacote, você está adicionando suas funções ao `search` da seção, permitindo que você chame estas funções diretamente. Por exemplo, a função `mvrnorm`, que gera números aleatórios de uma normal multivariada, está no pacote `MASS`.

```
Sigma <- matrix(c(10,3,3,2), nrow = 2, ncol = 2) # Matriz de Var-Covar
mu <- c(1, 10) # Médias
x <- mvrnorm(n = 100, mu, Sigma) # Tenta gerar 100 obs, mas dá erro
## Error in eval(expr, envir, enclos): não foi possível encontrar a função "mvrnorm"

library(MASS) # Carrega pacote
x <- mvrnorm(n = 100, mu, Sigma) # Agora funciona
```

Pacotes

Para ver o que está no search do R, utilize a função search(). Note que o pacote MASS agora esta lá.

```
search()
##  [1] ".GlobalEnv"        "package:MASS"      "package:ggplot2"
##  [4] "package:stats"     "package:graphics"  "package:grDevices"
##  [7] "package:utils"     "package:datasets"  "package:methods"
## [10] "Autoloads"         "package:base"
```

Para descarregar um pacote, utilize a função detach().

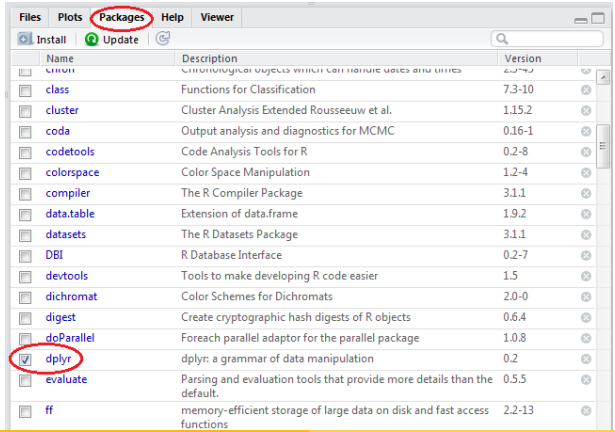
```
detach(package:MASS)
```

Às vezes pacotes tem o mesmo nome de funções. Neste caso, se ambos forem carregados, a função que prevalece é a do pacote que foi carregado por último. Uma outra forma de resolver isso é usar o nome do pacote e o operador :: antes de chamar a função. Neste caso não há ambiguidade.

```
x <- MASS::mvrnorm(n = 100, mu, Sigma)
```

Pacotes

Você também pode carregar ou descarregar pacotes pelo menu do canto inferior direito do RStudio, clicando na caixa ao lado do nome do pacote.



Pacotes

Para instalar um pacote, use a função `install.packages()`. Por exemplo, o pacote `dplyr` é muito utilizado para manipulação de dados. Para instalá-lo e desinstalá-lo, temos os seguintes comandos.

```
install.packages("dplyr")

############## Não rode ##################
remove.packages(dplyr)
```

Pacotes, CRAN e Sites

Grande parte dos pacotes do R estão centralizados em um repositório chamado CRAN (The Comprehensive R Archive Network), com diversos espelhos ao redor do mundo. Agora, vamos explorar um pouco o site oficial do R, **www.r-project.org**, e do CRAN, **cran.r-project.org**.

- Manuais, livros, documentação;
- Listas de e-mais;
- The R Journal;
- Lista de pacotes;
- Task Views.

Exercícios

Sua vez.

- Além do `dplyr`, vamos usar outros pacotes como `ggplot2`, `ggthemes`, `tidyr`, `reshape2` e `stringr`. Instale esses pacotes.
- Faça seu cadastro no StackOverflow e StackOverflow em português.
- Veja os temas do Task View do R. Há algum tema que te interessa?

Soluções

Você pode instalar todos os pacotes de uma só vez com `install.packages()`:

```
install.packages(c("ggplot2", "ggthemes",
                   "tidyr", "reshape2", "stringr"))
```

Um breve exemplo

Imóveis do Plano Piloto

Para ilustrar uma análise no R, veremos um exemplo simples com ofertas online de apartamentos à venda no Plano Piloto,do site *Wimoveis*. Os dados foram obtidos por meio de webscraping utilizando o R e já passaram por um processo de "limpeza" prévio. Não se preocupe em entender os comandos agora, iremos trabalhar isso no decorrer do curso.

Imóveis do Plano Piloto

Baixe os dados em:
`https://dl.dropboxusercontent.com/u/44201187/dados/wi.venda.rds`
Salve na pasta "Dados" do seu projeto.

```
# Carregando os pacotes que vamos usar
library(dplyr)
library(ggplot2)

# Carregando os dados
dados <- readRDS("Dados/wi.venda.rds")
```

Imóveis do Plano Piloto

```
# Vendo a estrutura dos dados
str(dados, vec.len = 1)
## 'data.frame':    3786 obs. of  9 variables:
##  $ bairro   : chr  "Noroeste" ...
##  $ location : chr  "SQNW 310 " ...
##  $ preco    : num  150000 175000 ...
##  $ quartos  : num  3 1 ...
##  $ m2       : num  117 30 ...
##  $ corretora: chr  "12220" ...
##  $ data     : Date, format: "2014-01-23" ...
##  $ link     : chr  "/imovel/venda-apartamento-brasilia-df-3-quartos-sqnw
##  $ pm2      : num  1282 ...
```

Imóveis do Plano Piloto

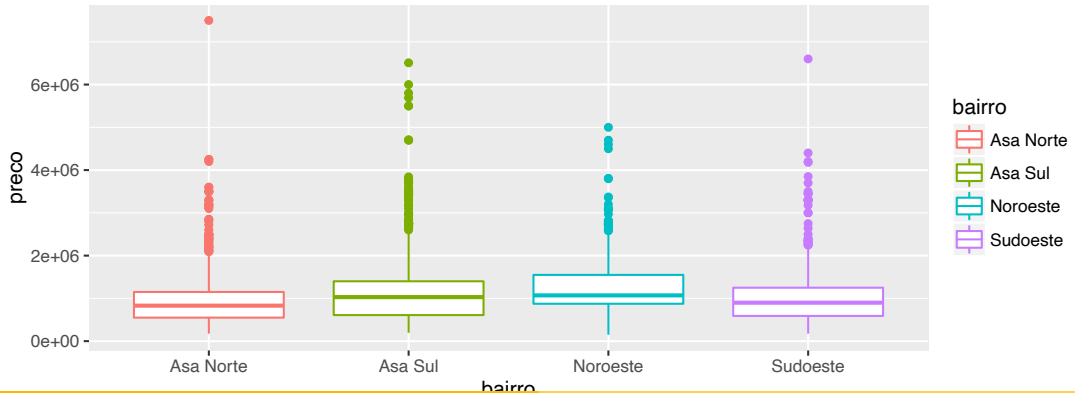
```
# Calculando medianas por bairro
mediana_bairro <- dados %>%
  group_by(bairro) %>%
  summarise(Median_Preco = median(preco),
            Median_Quarto = median(quartos),
            Median_pm2 = median(pm2),
            N = n())

mediana_bairro
## Source: local data frame [4 x 5]
##
##      bairro Median_Preco Median_Quarto Median_pm2     N
##       (chr)        (dbl)         (dbl)      (dbl) (int)
## 1 Asa Norte       830000             3       8929  1331
## 2   Asa Sul      1030000             3       8800  1036
## 3  Noroeste      1070000             3       9934   635
## 4  Sudoeste       899500             3       9706   784
```

Imóveis do Plano Piloto

```
# Rodando um modelo linear
modelo <- lm(preco ~ m2 * bairro + quartos , data = dados)

# resultados do modelo
# não cabe tudo na tela do slide
summary(modelo)
##
## Call:
## lm(formula = preco ~ m2 * bairro + quartos, data = dados)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -3889078  -105771   -10453    90518  2263876
##
## Coefficients:
##                   Estimate Std. Error t value Pr(>|t|)
## (Intercept)         -97799      17814   -5.49  4.3e-08 ***
```

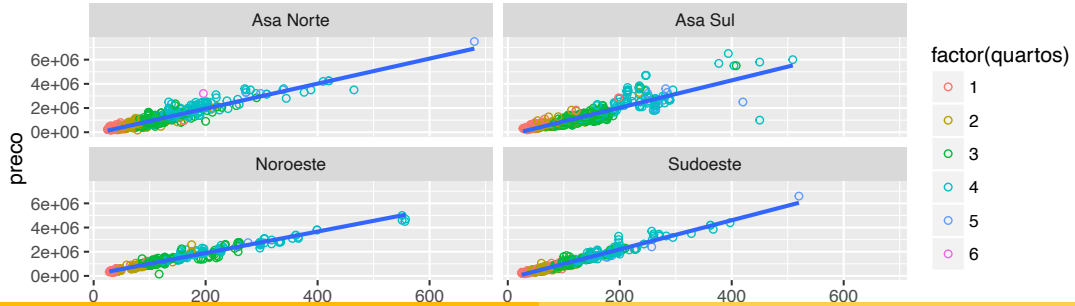
Imóveis do Plano Piloto

```
# Box-Plot dos precos por bairro
ggplot(dados, aes(bairro, preco, color = bairro)) + geom_boxplot()
```



Imóveis do Plano Piloto

```
# Gráfico de dispersão com regressão
ggplot(dados, aes(m2, preco)) +
  geom_point(shape = 1, aes(color = factor(quartos))) +
  geom_smooth(method = "lm") +
  facet_wrap(~bairro)
```



Programação em R

Julho, 2016

Introdução à área de Trabalho

Assignment Operator

No R, praticamente tudo é um objeto. Estes objetos podem ser variáveis, vetores, matrizes, arrays, números, caracteres e, inclusive, funções. Durante o uso do R, você irá criar e guardar estes objetos com um nome, por exemplo:

```
x1 <- 10 # atribui o valor 10 à variável x1
x1
## [1] 10
```

O operador <- é conhecido como "assignment operator" e, no caso acima, atribuiu o valor 10 a um objeto de nome x1. Na maior parte das vezes, o operador = é equivalente a <-. Vejamos:

```
x2 = 20 # atribui o valor 20 à variável x2
x2
## [1] 20
```

Assignment Operator

Entretanto, recomenda-se o uso do <- pois ele funciona sempre como atribuição, enquanto que o operador = é usado em parâmetros dentro de um função e o comportamento é diferente (veremos isso mais a frente).
O assignment operator <- nada mais é do que uma função. Uma função equivalente ao <-, que tem sintaxe mais usual, é a função `assign("nomeObjeto", valorObjeto)`.

```
assign("x3", 4) # cria a variável x3 com o valor 4
x3
## [1] 4

x3 <- 4 # note que é equivalente!
x3
## [1] 4
```

Assignment Operator

Uma vez que você atribuiu uma variável a um nome, você pode utilizá-la em operações e funções utilizando seu nome.

```
x1 + x2 + x3 # soma x1,x2,x3
## [1] 34

x1/x2 # divide x1 por x2
## [1] 0.5

x1 * x2 # multiplica x1 por x2
## [1] 200

y <- x1*x2 + x3 #cria y com o resultado
y
## [1] 204
```

Nomes de variáveis

O R é *case sensitive* (diferentemente do SQL, por exemplo), portanto `x1` e `X1` são objetos diferentes.

```
X1 # Não vai encontrar
## Error in eval(expr, envir, enclos): objeto 'X1' não encontrado

x1 # Encontra
## [1] 10
```

Nomes de variáveis

Nomes de variáveis no R podem conter combinações arbitrárias de números, textos, bem como ponto (.) e *underscore* (_). Entretanto, os nomes não podem **começar** com números ou *underscore*.

```
a1_B2.c15 <- "variável com nome estranho"
a1_B2.c15
## [1] "variável com nome estranho"
```

```
_vai_dar_erro <- 2000
## Error: <text>:1:1: unexpected input
## 1: _
##     ^
```

Encontrando e removendo objetos: `ls()`

Para listar todos os objetos que estão na sua área de trabalho, você pode usar a função `ls()`.

```
ls()
## [1] "a1_B2.c15" "x1"        "x2"        "x3"        "y"
```

Note que apareceram todos os objetos que criamos.

Encontrando e removendo objetos: `rm()`

A função `rm(objeto)` remove um objeto da área de trabalho.

```
rm(x3) # remove o objeto x3 da área de trabalho

x3 # não encontrará o objeto
## Error in eval(expr, envir, enclos): objeto 'x3' não encontrado

ls() # note que x3 não aparecerá na lista de objetos
## [1] "a1_B2.c15" "x1"        "x2"        "y"
```

Salvando e carregando objetos na área de trabalho

Para salvar uma cópia de todos os objetos criados você pode utilizar a função `save.image()`:

```
# salva a área de trabalho no arquivo "aula_1.RData"
save.image(file = "aula_1.RData")
```

Salvando e carregando objetos na área de trabalho

Agora que salvamos os objetos, vamos limpar nossa área de trabalho:

```
# remove todos objetos da área de trabalho
rm(list = ls())

# não encontra nada
ls()
## character(0)
```

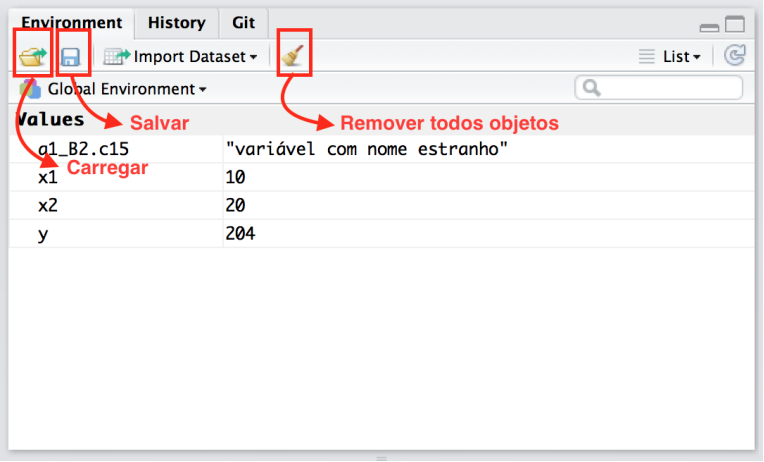
Salvando e carregando objetos na área de trabalho

E vamos recuperar todos os objetos que tínhamos criado com `load()`.

```
# carrega objetos salvos em aula_1.RData
load(file = "aula_1.RData")

 # note que os objetos foram carregados
ls()
## [1] "a1_B2.c15" "x1"        "x2"        "y"
```

Salvando e carregando objetos na área de trabalho

Se você estiver utilizando o RStudio, essas operações são possíveis na janela de objeto:



Salvando e carregando objetos na área de trabalho

Mais para frente veremos outras formas de salvar e ler dados no R, bem como manipulação de pastas e arquivos.

Exercícios

Sua vez.

- Salve sua seção atual do R.
- Remova todos os objetos do ambiente de trabalho.
- Crie três objetos, cada um com uma forma diferente que aprendemos de criar objeto `<-`, `=` e `assign`. Use `ls()` para verificar se os objetos foram criados. Remova um dos objetos. Use `ls()` para ver se o objeto foi removido.
- Remova todos os objetos do ambiente de trabalho.
- Carregue os dados da seção anterior para continuarmos a aula.

Soluções

```r
# Salve sua seção atual do R
save.image(file = "exercicio1.rda")

# Remova todos os objetos do ambiente de trabalho.
rm(list = ls())

# Crie três objetos, cada um com uma forma
# diferente que aprendemos de criar objeto
x1 <- 1
x2 = 2
assign("x3", 3)
```

Soluções

```r
# Use `ls()` para verificar se os objetos foram criados
ls()

# Remova um dos objetos
rm(x1)

# Use `ls()` para ver se o objeto foi removido.
ls()

#  Remova todos os objetos do ambiente de trabalho.
rm(list = ls())

# Carregue os dados da seção anterior para continuarmos a aula.
load(file = "exercicio1.rda")
```

O objeto básico do R: vetores

Vetores: concatenação

No R, a estrutura mais simples de dados é um vetor. Os objetos x1, x2 e x3 são, na verdade, vetores de tamanho 1. Para construir um vetor você pode utilizar a função c(), que **c**oncatena uma quantidade arbitrária de objetos.

```
# concatena x1, 2, 3 e x2.
x3 <- c(x1, 2, 3, x2)
x3
## [1] 10  2  3 20

# concatena "a", "b" e "c"
x4 <- c("a", "b","c")
x4
## [1] "a" "b" "c"
```

Classes de vetores: numeric, integer, complex, character

Os vetores no R podem ser, entre outros, de tipos: *numeric* (número comum), *integer* (inteiro), *complex* (número complexo), *character* (texto), *logical* (lógicos, booleanos). Também há datas e fatores, que discutiremos mais a frente.

```
numero <- c(546.90, 10, 789)

# notem o L
inteiro <- c(1L, 98L)

# notem o i
complexo <- c(20i, 2 + 9i)

# notem as aspas
texto <- c("aspas duplas", 'aspas simples', "aspas 'dentro' do texto")

# sempre maiúsculo
logico <- c(TRUE, FALSE, TRUE)
```

Classes de vetores: `class()`

A função `class()` é útil para identificar a classe de um objeto.

```
class(numero)
## [1] "numeric"

class(inteiro)
## [1] "integer"

class(complexo)
## [1] "complex"

class(texto)
## [1] "character"

class(logico)
## [1] "logical"
```

Classes de vetores: `is.xxx`

Você pode testar se um vetor é de determinada classe com as funções `is.xxx` (sendo "xxx" a classe). Por exemplo:

```
is.numeric(numero)
## [1] TRUE

is.character(numero)
## [1] FALSE

is.character(texto)
## [1] TRUE

is.logical(texto)
## [1] FALSE
```

Coerção: vetores tem somente uma única classe!

Um vetor somente pode ter elementos de uma única classe. Não é possível, por exemplo, misturar textos com números.

Coerção

Quando você mistura elementos de classes diferentes em um vetor, o R faz a coerção do objeto para uma das classes, obedecendo a seguinte ordem de prioridade:

- lógico $<$ inteiro $<$ numérico $<$ complexo $<$ texto

Coerção: vetores tem somente uma única classe!

Vejamos alguns exemplos:

```
x <- c(1, 2, 3L)
class(x)
## [1] "numeric"

x <- c(1, 2, 3L, 4i)
class(x)
## [1] "complex"

x <- c(1, 2, 3L, 4i, "5")
class(x)
## [1] "character"
```

Coerção: as.xxx

Você pode forçar a conversão de um vetor de uma classe para outra com as funções as.xxx (sendo "xxx" a classe). Entretanto, nem sempre essa conversão faz sentido, e pode resultar em erros ou NA's. Por exemplo:

```
as.character(numero) # Vira texto
## [1] "546.9" "10"    "789"

as.numeric(logico) # TRUE -> 1, FALSE -> 0
## [1] 1 0 1

as.numeric(texto) # Não faz sentido
## Warning: NAs introduzidos por coerção
## [1] NA NA NA

as.numeric("1012312") # Faz sentido
## [1] 1e+06
```

Estrutura e tamanho de um objeto

Para ver a e**str**utura de um objeto no R, use a função `str()`. Esta é uma função simples, mas talvez das mais úteis do R.

```
str(x3)
##  num [1:4] 10 2 3 20
str(numero)
##  num [1:3] 547 10 789
str(inteiro)
##  int [1:2] 1 98
str(complexo)
##  cplx [1:2] 0+20i 2+9i
str(texto)
##  chr [1:3] "aspas duplas" "aspas simples" "aspas 'dentro' do texto"
str(logico)
##  logi [1:3] TRUE FALSE TRUE
```

Estrutura e tamanho de um objeto

Para obter o tamanho de um objeto, utilize a função `length()`. Comandos no R podem ser colocados na mesma linha se separados por ponto-e-vírgula (;).

```
length(x1);length(x3);length(x4)
## [1] 1
## [1] 4
## [1] 3

length(numero); length(inteiro); length(complexo)
## [1] 3
## [1] 2
## [1] 2

length(texto); length(logico)
## [1] 3
## [1] 3
```

Nomes dos elementos de um objeto

Objetos podem ter elementos nomeados. Por exemplo, vamos nomear os elementos do vetor `numero`. A função `names()` serve tanto para consultar quanto para alterar os nomes de um objeto.

```
numero
## [1] 547  10 789

names(numero) <- c("numero1", "numero2", "numero3")
numero
## numero1 numero2 numero3
##     547      10     789

names(numero)
## [1] "numero1" "numero2" "numero3"
```

Selecionando elementos de um vetor: índices

Você pode acessar elementos de um vetor por meio de colchetes ([]).

```
numero[1] # apenas primeiro elemento
## numero1
##     547

numero[c(1,2)] # elementos 1 e 2
## numero1 numero2
##     547      10

numero[c(1,3)] # elementos 1 e 3
## numero1 numero3
##     547     789

numero[c(3,1,2)] # troca de posição
## numero3 numero1 numero2
##     789     547      10
```

Selecionando elementos de um vetor: índices negativos

Você pode usar índices negativos para omitir certos elementos:

```
numero[-1] # todos menos o primeiro
## numero2 numero3
##      10     789

numero[-c(1,2)] # todos menos 1 e 2
## numero3
##     789

numero[-c(1,3)] # todos menos 1 e 3
## numero2
##      10
```

Selecionando elementos de um vetor: nomes

Isso também funciona com nomes, caso o vetor seja nomeado.

```
numero["numero1"]
## numero1
##     547

numero["numero2"]
## numero2
##      10

numero[c("numero1", "numero3")]
## numero1 numero3
##     547     789
```

Selecionando elementos de um vetor: TRUE e FALSE

Por fim, vetores lógicos podem ser utilizados para selecionar elementos.

```
numero[c(TRUE, FALSE, FALSE)] # apenas primeiro elemento
## numero1
##     547

numero[c(FALSE, FALSE, TRUE)]  # apenas terceiro elemento
## numero3
##     789

numero[c(TRUE, FALSE, TRUE)] # elementos 1 e 3
## numero1 numero3
##     547     789
```

Alterando elementos de um vetor

Você pode usar o assignment operator (<-) junto com colchetes ([ ]) para alterar elementos específicos do vetor.

```
numero
## numero1 numero2 numero3
##     547      10     789

numero[1] <- 100 # altera elemento 1 para 100
numero
## numero1 numero2 numero3
##     100      10     789

numero[2:3] <- c(12.3, -10) # altera elementos 2 e 3
numero
## numero1 numero2 numero3
##     100      12     -10
```

Ordenando um vetor

A função `order()` retorna um vetor com as posições para que um objeto fique em ordem crescente.

```
order(numero) # indices
## [1] 3 2 1

numero[order(numero)] # ordena numero
## numero3 numero2 numero1
##     -10      12     100
```

A função `sort()` retorna o vetor ordenado.

```
sort(numero)
## numero3 numero2 numero1
##     -10      12     100
```

As duas funções tem o parâmetro `decreasing` que, quando `TRUE`, retornam o vetor em ordem decrescente.

Criando sequências e repetições

Uma forma rápida e fácil de criar uma sequência de inteiros no R é utilizando dois pontos (`:`).

```
# Sequencia de 1 a 10
1:10
##  [1]  1  2  3  4  5  6  7  8  9 10

# Sequencia de -1 a -10
-1:(-10)
##  [1]  -1  -2  -3  -4  -5  -6  -7  -8  -9 -10
```

Criando sequências e repetições

Alternativamente, é possível criar sequências mais flexíveis com a função `seq()`:

```
seq(from = 1, to = 10, by = 3)
## [1]  1  4  7 10
```

E, para repetições, temos a função `rep()`:

```
rep(1, times = 10)
##  [1] 1 1 1 1 1 1 1 1 1 1

rep(c(1,2), times = 5)
##  [1] 1 2 1 2 1 2 1 2 1 2
```

Exercícios

Sua vez.

- Crie dois vetores cada um com uma classe diferente e com tamanhos diferentes. Teste algumas das funções `is.xxx` e `as.xxx` nesses vetores. Utilize as funções `str()` e `length()` para entender melhor a estrutura e tamanho dos vetores que você criou.
- Os vetores abaixo serão de quais classes?
  - `v1 <- c("a", TRUE, 1, 2, 10)`
  - `v2 <- c(TRUE, 1, 1L, 4)`
- Considere o seguinte vetor `y <- 1:3`. Nomeie os elementos desse vetor como `c("e1", "e2", "e3")`. Selecione o primeiro elemento do vetor por nome, por posição e com vetor lógico. Selecione todos menos o terceiro elemento. Altere o segundo elemento do vetor para 10.
- Considere o seguinte vetor `x <- 1:100`. Selecione apenas os elementos pares (2, 4, 6, ...).

Soluções

```r
# Crie um vetor de pelo menos duas classes diferentes e com tamanhos diferentes
x1 <- c("vetor", "de", "texto")
x2 <- 1:10

# Teste algumas das funções `is.xxx` e `as.xxx`
is.integer(x2); as.character(x2); is.character(x1)

# Utilize as funções `str()` e `length()`
str(x1); str(x2)
length(x1); length(x2)

# Os vetores abaixo serão de quais classes?
v1 <- c("a", TRUE, 1, 2, 10)
class(v1)

v2 <- c(TRUE, 1, 1L, 4)
class(v2)
```

Soluções

```r
# Nomeie os elementos
y <- 1:3
names(y) <- c("e1", "e2", "e3")

# Seleciona primeiro
y[1];y["e1"];y[c(TRUE, FALSE, FALSE)]

# Todos menos 3
y[-3]

# altera o 2
y[2] <- 10

# apenas elementos pares
x <- 1:100
x[seq(2, 100, by = 2)]
```

Vetorização, reciclagem, operadores matemáticos, relacionais e lógicos básicos

Aritmética básica e vetorização

O R tem uma série de operadores de aritmética básica, entre eles:

Operador	Descrição
`x + y`	Soma (elemento a elemento)
`x - y`	Subtração (elemento a elemento)
`x * y`	Multiplicação (elemento a elemento)
`x / y`	Divisão (elemento a elemento)
`x ^ y`	Exponenciação (elemento a elemento)
`x %/% y`	Divisão por inteiro (elemento a elemento)
`x %% y`	Resto da divisão (elemento a elemento)

Vetorização!

Muitos operadores e funções do R são vetorizados, isto é, os cálculos são realizados elemento a elemento. Durante o curso esteja sempre atento a isso e tente explorar a vetorização do R para facilitar seus cáculos.

Soma e Subtração

```
# soma
1 + 20
## [1] 21
```

```
# soma vetorizada (elemento a elemento)
c(1,2,3) + c(4,5,6)
## [1] 5 7 9
```

```
# subtração
200 - 2
## [1] 198
```

```
# subtração vetorizada (elemento a elemento)
c(1,2,3) - c(4,5,6)
## [1] -3 -3 -3
```

Multiplicação e Divisão

```
# divisão
200 / 15
## [1] 13
```

```
# divisão vetorizada (elemento a elemento)
c(2,4,6) / c(1,2,3)
## [1] 2 2 2
```

```
# multiplicação
2*10
## [1] 20
```

```
# multiplicação vetorizada (elemento a elemento)
c(10,9,8) * c(1,2,3)
## [1] 10 18 24
```

Exponenciação, Divisão por inteiro, Resto

```
# exponenciação
4^2
## [1] 16
```

```
# exponenciação vetorizada (elemento a elemento)
c(2,2,2) ^ c(1,2,3)
## [1] 2 4 8
```

```
# divisão por inteiro (desconsidera o resto)
7 %/% 3
## [1] 2
```

```
# divisão por inteiro vetorizada
c(7,7) %/% c(3,2)
## [1] 2 3
```

Exponenciação, Divisão por inteiro, Resto

```
# resto da divisão
7 %% 3
## [1] 1
```

```
# resto da divisão vetorizada
c(7,7) %% c(3,2)
## [1] 1 1
```

Reciclagem

O que acontece quando fazemos uma operação com vetores de tamanho diferente? Vejamos:

```
x <- c(1, 2, 3, 4)
x * 2
## [1] 2 4 6 8
```

Note que o R multiplicou todos os elementos de `x` por 2! Este processo é chamado de **reciclagem**, o R vai "expandindo" – ou "reciclando" – os valores do vetor menor até que este fique com a mesma quantidade de elementos que o vetor maior. Ou, mais especificamente, a operação acima é equivalente a:

```
x * c(2, 2, 2, 2)
## [1] 2 4 6 8
```

Reciclagem

E o que ocorreria se o vetor menor tivesse dois elementos? A mesma coisa:

```
x <- c(1, 2, 3, 4)
x * c(2, 3)
## [1]  2  6  6 12

# equivalente
x * c(2, 3, 2, 3)
## [1]  2  6  6 12
```

E o que acontece com o exemplo a seguir? Agora note a mensagem de aviso (warning):

```
x * c(2, 3, 1)
## Warning in x * c(2, 3, 1): comprimento do objeto maior não é múltiplo do
## comprimento do objeto menor
## [1] 2 6 3 8
```

Outras funções matemáticas

Além dos operadores básicos, há uma série de funções matemáticas comumente utilizadas, tais como:

Função	Descrição
`abs(x)`	Valor absoluto.
`log(x)`	Logaritmo natural.
`exp(x)`	Exponencial.
`sqrt(x)`	Raiz quadrada.
`factorial(x)`	Fatorial.

Todas essas funções **são vetorizadas**.

Outras funções matemáticas

```
x <- c(1,2,-3, 4, -20.3) #criando o vetor x
abs(x) # valor absoluto
## [1]  1  2  3  4 20
```

```
# note que NaN's foram gerados, iremos falar disso mais a frente
log(x)
## Warning in log(x): NaNs produzidos
## [1] 0.00 0.69  NaN 1.39  NaN
```

```
exp(x)
## [1] 2.7e+00 7.4e+00 5.0e-02 5.5e+01 1.5e-09
```

```
sqrt(x) # NaN's gerados
## Warning in sqrt(x): NaNs produzidos
## [1] 1.0 1.4 NaN 2.0 NaN
```

Outras funções matemáticas

```
# fatorial
factorial(5) # 5*4*3*2*1
## [1] 120
```

Há também diversas funções trigonométricas, como seno `sin()`, cosseno `cos()`, tangente `tan()` e outras.

```
sin(pi); cos(pi)
## [1] 1.2e-16
## [1] -1
```

Note que o resultado de $sin(\pi)$ não foi exatamente zero (mas é zero em termos práticos). Como qualquer outra linguagem, o R trabalha com números de ponto flutuante, então é preciso tomar algumas preucações. Falaremos mais disso adiante.

Outras funções matemáticas

Funções que calculam estatísticas descritivas dos vetores:

Função	Descrição
`sum(x)` e `cumsum(x)`	Soma e soma acumulada.
`prod(x)` e `cumprod(x)`	Produtório e produtório acumulado.
`min(x)`, `cummin(x)` e `pmin(x, y)`	Mínimo, mínimo acumulado e mínimo par a par.
`max(x)`, `cummax(x)` e `pmax(x, y)`	Máximo, máximo acumulado e máximo par a par.
`mean(x)`	Média.
`var(x)` e `sd(x)`	Variância e desvio-padrão.
`cov(x, y)` e `cor(x, y)`	Covariância e correlação.
`diff(x)`	Primeira diferença.

Outras funções matemáticas

Exemplos:

```
mean(x) # média
## [1] -3.3

sum(x) # somatório
## [1] -16

prod(x) # produtório
## [1] 487

cumsum(x) # somatório acumulado
## [1]   1   3   0   4 -16

cumprod(x) # produtório acumulado
## [1]   1   2  -6 -24 487
```

Outras funções matemáticas

Exemplos:

```
y <- 1:5

var(x) # variância sum((x-mean(x))^2)/(length(x)-1)
## [1] 97

sd(x) # desvio-padrão sqrt(var(x))
## [1] 9.9

median(x) # mediana
## [1] 1

cov(x, y) # covariância sum((x-mean(x))*(y-mean(y)))/(length(x)-1)
## [1] -10

cor(x, y) # correlação de x e y cov(x,y)/(sd(x)*sd(y))
## [1] -0.65
```

Outras funções matemáticas

Exemplos:

```
min(x) # mínimo
## [1] -20

max(x) # máximo
## [1] 4

cummin(x) # mínimo "acumulado"
## [1]   1   1  -3  -3 -20

cummax(x) # máximo "acumulado"
## [1] 1 2 2 4 4

diff(x) # diferença
## [1]   1  -5   7 -24
```

Operadores Relacionais

Vejamos agora alguns operadores relacionais. Estes operadores são importantes para determinar a relação entre dois vetores e, como seu resultado é um vetor lógico, são muito utilizados para fazer `subset`. Da mesma forma que os operadores matemáticos, operadores relacionais também são vetorizados.

Operador	Descrição
`x == y`	x é igual a y? Faz coerção.
`x != y`	x é diferente de y?
`x > y`	x é maior do que y?
`x >= y`	x é maior ou igual a y?
`x < y`	x é menor do que y?
`x <= y`	x é menor ou igual a y?

Operadores Relacionais

Para comparar se dois objetos são iguais, use ==.

```
x <- 10
y <- 20
x == y
## [1] FALSE

x <- c(10, 20, 30)
y <- c(10, 10, 30)
x == y
## [1]  TRUE FALSE  TRUE

x[x == y] # pega elementos de x que são iguais a y
## [1] 10 30
```

Operadores Relacionais

Se você comparar objetos de classes diferentes (um texto com um número, por exemplo), sempre que possível, o operador == irá converter um dos objetos que está sendo comparado para tentar realizar a comparação.

```
x <- c(10, 20)
y <- c("10", "20")
x == y # converte x para texto e compara textualmente.
## [1] TRUE TRUE
```

É preciso tomar cuidado com coerções, pois isso pode gerar resultados inesperados:

```
20 > "100" # -> correto do ponto de vista textual (dicionário)
## [1] TRUE
```

Operadores Relacionais

Caso você queira verificar se dois vetores são exatamente idênticos, você não deve utilizar == e sim `identical()`.

```
identical(x, y)
## [1] FALSE
```

Voltemos, agora, ao caso do valor calculado para $\sin(\pi)$. Lembra que o valor não foi exatamente zero? Isso ocorre porque computadores trabalham com números de ponto flutuante e não têm precisão infinita.

```
identical(sin(pi), 0)
## [1] FALSE

identical(0.1, 0.3 - 0.2)
## [1] FALSE
```

Operadores Relacionais

Assim, sempre que for fazer comparações de números resultantes de cálculos, é preciso considerar uma tolerância de erro. Uma função que faz isso é a `all.equal()` que leva em conta uma tolerância de erro de ponto flutuante.

```
sin(pi) == 0 #  falso, incorreto.
## [1] FALSE

all.equal(sin(pi), 0) #  verdadeiro, correto
## [1] TRUE
```

Operadores Relacionais

Ou você pode testar se a diferença absoluta entre um número e outro é irrelevante (do ponto de vista numérico):

```
abs(sin(pi) - 0) < 1e-12
## [1] TRUE
```

Operadores Relacionais

Além do operador ==, como vimos, também temos os operadores: maior >, maior ou igual >=, menor <, menor ou igual <= e diferente !=. Note, novamente, que os resultados de todas essas operações são objetos do R, mais especificamente, vetores lógicos e podem ser utilizados em operações subsequentes:

```
x <- c(1,2,3,4,5)
y <- c("1","3","2","4","4");

# guarda resultado da comparação em vetor_logico
vetor_logico <- (x >= y)
vetor_logico
## [1]  TRUE FALSE  TRUE  TRUE  TRUE

# usa vetor logico para fazer subset de x
x[vetor_logico]
## [1] 1 3 4 5
```

Operadores Relacionais

Um fato bastante útil de vetores lógicos é o de que eles, quando convertidos para numéricos, são transformados em vetores de 0's e 1's. Assim, por exemplo, no caso anterior, se quisermos saber quantos x são maiores ou iguais a y, basta somar os 1's do `vetor_logico` ou, se quisermos saber a proporção de x que são maiores ou iguais a y, basta tirarmos a média do `vetor_logico`.

```
as.numeric(vetor_logico)
## [1] 1 0 1 1 1

sum(vetor_logico)
## [1] 4

mean(vetor_logico)
## [1] 0.8
```

Operadores lógicos

Podemos operar também com os próprios valores lógicos. O operador `!` nega o valor lógico ao qual é aplicado, isto é, `!FALSE` vira `TRUE` e `!TRUE` vira `FALSE`. No caso anterior, se quisermos saber quantos x são menores do que y, não precisamos comparar novamente, basta negar o resultado do `vetor_logico` e somar.

```
!vetor_logico
## [1] FALSE  TRUE FALSE FALSE FALSE

sum(!vetor_logico)
## [1] 1

identical(!vetor_logico, (x < y)) # são idênticos
## [1] TRUE
```

Operadores lógicos

Operadores lógicos E(AND) & e OU(OR) |.

```
c(TRUE, FALSE) & c(TRUE, TRUE)
## [1]  TRUE FALSE

c(TRUE, FALSE) | c(TRUE, TRUE)
## [1] TRUE TRUE
```

Funções sobre vetores lógicos

Outros comandos que podem ser bastante úteis são o `all()` e o `any()`. O primeiro retorna `TRUE` se todos os elementos forem `TRUE`. Já o segundo retorna `TRUE` se ao menos um elemento for `TRUE`.

```
set.seed(11) # semente da simulação
x <- rnorm(1000, 5, 2) # simula 100 obs normal(5, 2)

any(x > 10) # há algum x maior do do que 10
## [1] TRUE

all(x > -20 & x < 20) # todos os x estão entre -1 e 20?
## [1] TRUE
```

Funções sobre vetores lógicos

A função `which()` retorna a posição dos elementos que são TRUE.

```
which(c(TRUE, FALSE, TRUE))
## [1] 1 3

# quais as posições dos elementos de x
# que são maiores do que 10?
which(x > 10)
## [1] 196 273 523 558 929
```

Funções de conjuntos

Função	Descrição
`setdiff(x, y)`	Conjunto dos elementos de `x` que não estão em `y`
`intersect(x, y)`	Conjunto dos elementos de `x` que estão em `y`
`union(x, y)`	Conjunto união de `x` e `y`
`x %in% y`	Quais elementos de `x` estão em `y`?

Funções de conjuntos

Exemplos:

```
x <- 1:5
y <- c(1:3, 6:10)

setdiff(x, y)
## [1] 4 5

intersect(x, y)
## [1] 1 2 3

union(x, y)
##  [1]  1  2  3  4  5  6  7  8  9 10

x %in% y
## [1]  TRUE  TRUE  TRUE FALSE FALSE
```

NA, NaN, Infinito

Em muitos casos, o resultado de uma operação é infinito ou não determinado. Outras vezes, há valores ausentes em sua base de dados. O R tem objetos especiais para lidar com esses tipos de situação. Por exemplo, quando tiramos o log de um número negativo, obtemos como resultado o valor NaN (Not an Number). Outros valores de interesse são Inf (Infinito) e NA (Not Available).

```
log(-1) #NaN
## Warning in log(-1): NaNs produzidos
## [1] NaN

x <- NaN # atribuindo o valor NaN a x

x + 1 # somar 1 a algo indeterminado é indeterminado
## [1] NaN

NaN == NaN # Não faça isso... comparação lógica, note que deu NA
## [1] NA
```

NA, NaN, Infinito

Infinito:

```
2/0 # infinito
## [1] Inf

x <- 2/0

x - Inf # infinito - infinito não é um número (NaN)
## [1] NaN

1/x # zero
## [1] 0

is.infinite(x)
## [1] TRUE

is.finite(x)
## [1] FALSE
```

NA, NaN, Infinito

O NA representa valores ausentes e é bastante utilizado quando se lida com bases de dados. Muitas funções têm parâmetros que indicam como ela deve lidar com valores ausentes.

```
x <- c(1, 2, NA, 3)
x + 1
## [1]  2  3 NA  4

x + Inf
## [1] Inf Inf  NA Inf

sum(x)
## [1] NA

sum(x, na.rm = TRUE) #na.rm, opção para remover (rm) os NA (na)
## [1] 6
```

NA, NaN, Infinito

Para testar se um elemento é `NA`, use a função `is.na()`:

```
x == NA # Nunca faça isso!!!
## [1] NA NA NA NA

is.na(x) # Maneira correta
## [1] FALSE FALSE  TRUE FALSE
```

Tabelas da verdade

```
x <- c(NA, FALSE, TRUE)
names(x) <- as.character(x)

outer(x, x, "&") # tabela do E(AND) lógico
##         <NA> FALSE  TRUE
## <NA>      NA FALSE    NA
## FALSE FALSE FALSE FALSE
## TRUE      NA FALSE  TRUE

outer(x, x, "|") # tabela do OU(OR) lógico
##        <NA> FALSE TRUE
## <NA>     NA    NA TRUE
## FALSE    NA FALSE TRUE
## TRUE   TRUE  TRUE TRUE
```

Exercícios

Sua vez.
Rode o seguinte código:

```
set.seed(1)
horas_trabalhadas <- rnorm(1000, 8, 0.5)
valor_por_hora <- rnorm(1000, 30, 2)
horas_trabalhadas[sample(1:1000,5)] <- ifelse(rbinom(5,1,0.5),NA,0)
```

Os dados acima representam as horas trabalhadas e os valores recebidos por hora durante 1000 dias de um profissional. Note que algumas observações estão faltando e são NA. Considerando estes dados, responda as seguintes questões (para toda pergunta escreva o código que te dê a resposta, não responda "visualmente").

Exercícios

- Veja a estrutura dos dois vetores. Qual sua classe? Quantas observações têm?
- Há algum NA nos vetores? Se sim, quantos e quais observações?
- Encontre o mínimo e o máximo dos vetores.
- Crie um vetor com o valor recebido por dia. Encontre o mínimo e o máximo.
- Crie um vetor com o valor total recebido no período.
- Substitua os NA dos vetores por 0.
- Por quantos dias o profissional recebeu mais do que R$ 31 por hora? E em termos relativos (percentual do total de dias)? Crie um vetor com os valores recebidos maiores do que R$ 31 e outro vetor com os valores menores do que R$31.
- Crie um vetor com os valores recebidos por dia entre a média mais ou menos 1.5 vezes o DP. Salve sua área de trabalho.

Solução

```
str(horas_trabalhadas);str(valor_por_hora)

class(horas_trabalhadas);class(valor_por_hora)

length(horas_trabalhadas);length(valor_por_hora)

any(is.na(horas_trabalhadas));any(is.na(valor_por_hora))

sum(is.na(horas_trabalhadas));which(is.na(horas_trabalhadas));

min(horas_trabalhadas, na.rm = TRUE); max(horas_trabalhadas, na.rm = TRUE)

min(valor_por_hora); max(valor_por_hora)
```

Solução

```
summary(valor_por_hora)
summary(horas_trabalhadas)
valor_recebido_por_dia <- horas_trabalhadas * valor_por_hora
valor_recebido_total <- sum(valor_recebido_por_dia, na.rm = TRUE)
valor_recebido_total[is.na(valor_recebido_total)] <- 0
valor_recebido_por_dia[is.na(valor_recebido_por_dia)] <- 0
horas_trabalhadas[is.na(horas_trabalhadas)] <- 0
valor_por_hora[is.na(valor_por_hora)] <- 0
sum(valor_por_hora > 31);mean(valor_por_hora > 31)
maior_que_31 <- valor_por_hora[valor_por_hora > 31]
media <- mean(valor_recebido_por_dia)
dp <- sd(valor_recebido_por_dia)
indice <- valor_recebido_por_dia < media + 1.5*dp &
  valor_recebido_por_dia > media - 1.5*dp
entre_m_1.5_dp <- valor_recebido_por_dia[indice]
```

Programação em R

Julho, 2016

Vimos bastante coisa trabalhando apenas com vetores: como selecionar e modificar elementos por nome, posição e vetor lógico, coerção de classes, vetorização, reciclagem... e essas lições servem para praticamente todos os demais objetos do R! O vetor é o objeto básico do qual os demais objetos são constituídos. Nesta seção veremos estes outros objetos, como matrizes (e arrays), data.frames e listas. E tudo o que vimos anteriormente naturalmente se estende para esses objetos, com pequenas modificações.

Matrizes: vetores com dimensões

Criando matrizes: função `matrix()`

É possível criar matrizes no R com a função `matrix()`.

```
matrix(data = dados, ncol = numero_de_colunas, nrow = numero_de_linhas)
```

- No argumento `data` passamos o vetor que desejamos transformar em matriz;
- `ncol` especifica o número de colunas da matriz; e,
- `nrow` especifica o número de linhas da matriz.

Criando matrizes: função `matrix()`

Criemos nossa primeira matriz, com 10 elementos, sendo cinco linhas e duas colunas:

```
minha_matriz <- matrix(data = 1:10, ncol = 2, nrow = 5)
minha_matriz
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

Perceba que os elementos foram preenchidos por coluna. Isto é, os números de 1 a 5 ficaram na primeira coluna e os números de 6 a 10 na segunda.

Criando matrizes: função `matrix()`

Para que a função preencha a matriz por linhas, basta colocar como argumento `byrow = TRUE`:

```
minha_matriz_2 <- matrix(data = 1:10, ncol = 2, nrow = 5, byrow = TRUE)
minha_matriz_2
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
## [3,]    5    6
## [4,]    7    8
## [5,]    9   10
```

A matriz agora foi preenchida por linhas: temos os números 1 e 2 na primeira linha, 3 e 4 na segunda linha e assim por diante.

Omitindo `ncol` ou `nrow`

Você não precisa especificar os dois termos `ncol` e `nrow`, basta especificar um deles que o outro é automaticamente inferido pelo R.

```
matrix(data = 1:10, ncol = 5)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10

matrix(data = 1:10, nrow = 2)
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Estrutura e atributos de uma matriz

A função `str()` fornece algumas informações básicas sobre a estrutura da matriz:

```
str(minha_matriz)
##  int [1:5, 1:2] 1 2 3 4 5 6 7 8 9 10
```

Na descrição `int [1:5, 1:2]` vemos que o objeto `minha_matriz` é composto de inteiros (`int`) e tem duas dimensões, mais especificamente cinco linhas (`[1:5,...]`) e duas colunas (`[...,1:2]`).

Estrutura e atributos de uma matriz

```
# número de linhas da matriz
nrow(minha_matriz)
## [1] 5

# número de colunas da matriz
ncol(minha_matriz)
## [1] 2

# dimensões da matriz (# linhas, # colunas)
dim(minha_matriz)
## [1] 5 2

# tamanho da matriz (quantos elementos no total)
length(minha_matriz)
## [1] 10
```

Nomes das linhas e colunas

As linhas e colunas de uma matriz podem ser nomeadas. Para tanto você pode utilizar `rownames()` e `colnames()`:

```
rownames(minha_matriz) <- letters[1:5]
rownames(minha_matriz)
## [1] "a" "b" "c" "d" "e"

colnames(minha_matriz) <- LETTERS[1:2]
colnames(minha_matriz)
## [1] "A" "B"

str(minha_matriz)
##  int [1:5, 1:2] 1 2 3 4 5 6 7 8 9 10
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:5] "a" "b" "c" "d" ...
##   ..$ : chr [1:2] "A" "B"
```

Matrizes são vetores com um atributo a mais: dimensão

Outra forma bem simples de criar uma matriz é atribuir dimensões a um vetor. Por exemplo, podemos criar um objeto idêntico à `minha_matriz` da seguinte forma:

```
# cria vetor com números de 1 a 10
m <- 1:10

# atribui dimensões ao vetor (5 linhas e 2 colunas)
dim(m) <- c(5, 2)
m
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
```

Operações com matrizes

Operador	Descrição
`== != > >= < <=`	Operadores relacionais elemento a elemento
`+ - / *`	Soma, subtração, divisão e multiplicação elemento a elemento
`%*%`	Multiplicação de matriz
`%x%`	Produto de Kronecker
`t()`	Transposição de matriz
`solve()`	Inversão de matriz (ver também `qr()`, `svd()`, `chol()`)
`det()`	Determinante de uma matriz
`diag()`	Diagonal de uma matriz

Operações elemento a elemento: soma, multiplicação e divisão

Vamos criar uma matriz mais simples para testar essas operações:

```
z <- 1:4
dim(z) <- c(2, 2)
str(z)
##  int [1:2, 1:2] 1 2 3 4
```

Primeiramente a soma (elmento a elemento):

```
# soma elemento a elemento
z + z
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

Operações elemento a elemento: soma, multiplicação e divisão

A soma não pecisa ser de matriz com outra matriz. É possível somar uma matriz com um vetor. Neste caso, a soma é feita seguindo a ordem por coluna:

```
z + c(1, 2, 3, 4)
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8
```

Operações elemento a elemento: soma, multiplicação e divisão

Quando os elementos não são do mesmo tamanho, o R tenta fazer reciclagem (a mesma que vimos na seção de vetores).

```
z + 10
##      [,1] [,2]
## [1,]   11   13
## [2,]   12   14
```

Repare que o R somou cada elemento de z com o número 10. E se somassemos um vetor com dois elementos?

```
z + c(10, 20)
##      [,1] [,2]
## [1,]   11   13
## [2,]   22   24
```

Note que o R não reclamou da operação! Ele reciclou o vetor `c(10, 20)` para as duas colunas de z.

Operações elemento a elemento: soma, multiplicação e divisão

E se o vetor somado fosse de três elementos?

```
z + c(10, 20, 30)
## Warning in z + c(10, 20, 30): comprimento do objeto maior não é múltiplo do
## comprimento do objeto menor
##      [,1] [,2]
## [1,]   11   33
## [2,]   22   14
```

O R faz a operação mas te dá um aviso, pois o tamanho do vetor não é um múltiplo do tamanho da matriz. A reciclagem é uma característica bastante útil do R, mas é preciso tomar cuidado.

Operações elemento a elemento: soma, multiplicação e divisão

O mesmo que falamos para soma vale para as demais operações elemento a elemento:

```
z_mais_z <- z + z
z_mais_z
##      [,1] [,2]
## [1,]    2    6
## [2,]    4    8


dois_z <- 2*z
z_mais_z == dois_z
##      [,1] [,2]
## [1,] TRUE TRUE
## [2,] TRUE TRUE
```

Operações matriciais

Multiplicação matricial:

```
zz <- z %*% z
zz
##      [,1] [,2]
## [1,]    7   15
## [2,]   10   22
```

Transposição:

```
tz <- t(zz)
tz
##      [,1] [,2]
## [1,]    7   10
## [2,]   15   22
```

Operações matriciais

Inversão:

```
inv_zz <- solve(zz)
inv_zz
##       [,1] [,2]
## [1,]  5.5 -3.7
## [2,] -2.5  1.7
```

Determinante:

```
det(zz)
## [1] 4
```

Operações matriciais

Produto de kronecker:

```
z %x% z
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    3    9
## [2,]    2    4    6   12
## [3,]    2    6    4   12
## [4,]    4    8    8   16
```

Diagonal:

```
diag(z)
## [1] 1 4
```

Exemplo: regressão linear

Vamos simular e estimar os parâmetros de uma regressão linear usando operações matriciais:

```
# parametros
n_vars <- 4 # numero de variaveis
n_obs <- 100 # numero de observações

# Simulando y ~ Xb + e
X <- matrix(rnorm(n_obs*n_vars), ncol = n_vars) # vetor X
betas <- c(1, 2, 3, 4) # betas verdadeiros
erro <- rnorm(n_obs) # termo de erro normal(0,1)
y <- X %*% betas + erro # vetor y
```

Exemplo: regressão linear

Estimando por mínimos quadrados:

```
# estimando os betas: (X'X)^-1 X'y
betas_estimados <- solve(t(X) %*% X) %*% t(X) %*% y
betas_estimados[,1]
## [1] 0.95 1.90 2.96 4.17

# Comparando com a estimativa da função nativa do R
modelo <- lm(y ~ X - 1)
coef(modelo)
##   X1   X2   X3   X4
## 0.95 1.90 2.96 4.17
```

Nomes das linhas e colunas

Vamos criar uma matriz de exemplo mais realista:

```
# criando matriz de vendas para exemplo
# quantidade de vendedores
n_vendedores <- 3

# quantidade de dias
n_dias <- 5

# valores de venda (aleatórios)
# média de R$1000,00 por dia
# desvio padrão de R$200,00
set.seed(192)
media <- 1000
desvio <- 200
valores <- rnorm(n_dias*n_vendedores, media, desvio)
```

Nomes das linhas e colunas

```r
# cria matriz
vendas <- matrix(valores, nrow = n_dias, ncol = n_vendedores)

# nomes para linhas
rownames(vendas) <- c("segunda", "terça", "quarta", "quinta", "sexta")

# nomes para colunas
colnames(vendas) <- c("João", "Maria", "Ana")
```

Nomes das linhas e colunas

Vamos ver como ficou nossa matriz:

```
vendas
##          João Maria  Ana
## segunda 1201   927 1043
## terça   1027   920  833
## quarta  1096  1020  944
## quinta   915   952 1018
## sexta    787  1039 1293
```

Selecionando elementos de uma matriz

Tudo o que vimos para selecionar elementos de um vetor vale para matrizes. A principal diferença é que agora você vai estabelecer um índice para linha e outro para a coluna do elemento que você quer:

```
matriz[indice_de_linha, indice_de_coluna]
```

Selecionando elementos de uma matriz

Selecionando linhas (por nome ou posição):

```
vendas["segunda", ] # seleciona linha de nome "segunda"
##  João Maria   Ana
##  1201   927  1043


vendas[1, ] # seleciona primeira linha
##  João Maria   Ana
##  1201   927  1043


vendas[-c(1,2), ] # seleciona tudo exceto linhas 1 e 2
##        João Maria  Ana
## quarta 1096  1020  944
## quinta  915   952 1018
## sexta   787  1039 1293
```

Selecionando elementos de uma matriz

Selecionando colunas (por nome ou posição):

```
vendas[ ,"Ana"] # seleciona colunas de nome "Ana"
## segunda   terça  quarta  quinta   sexta
##    1043     833     944    1018    1293

vendas[ ,3] # seleciona terceira coluna
## segunda   terça  quarta  quinta   sexta
##    1043     833     944    1018    1293

vendas[ ,-3] # seleciona tudo exceto terceira coluna
##         João Maria
## segunda 1201   927
## terça   1027   920
## quarta  1096  1020
## quinta   915   952
## sexta    787  1039
```

Selecionando elementos de uma matriz

Selecionando linhas e colunas (por nome ou posição). Resultados omitidos.

```
# linhas 1, 3 e 5 - colunas João e Ana
vendas[c(1, 3, 5), c("João", "Ana")]

# linhas segunda, quarta e sexta - colunas 1 e 3
vendas[c("segunda", "quarta", "sexta"), c(1, 3)]

# tudo menos linhas 1, 3 e 5 e menos colunas 2 e 3
vendas[-c(1,3,5), -c(2,3)]
```

Cuidado: simplificação de matrizes e drop = FALSE

Uma coisa ao ter cuidado ao fazer subsets com matrizes é que, quando a seleção tem apenas uma linha ou uma coluna, o R simplifica o resultado para um vetor. Na maior parte das vezes isso é bastante conveniente, pois é de fato o que queremos, mas em alguns casos pode ser desejável manter a estrutura de matriz no resultado. Para tanto, basta colocar como argumento `drop = FALSE`.

```
vendas_segunda <- vendas[1, , drop = FALSE]
vendas_segunda
##          João Maria  Ana
## segunda 1201   927 1043

# resultado é uma matriz
dim(vendas_segunda)
## [1] 1 3
```

Alterando valores de uma matriz

É possível combinar a seleção de elementos com o operador <- para realizar alterações em uma matriz:

```
vendas[c("quarta", "quinta"), "Ana"]
## quarta quinta
##    944   1018

vendas[c("quarta", "quinta"), "Ana"] <- c(1500, 2000)

vendas[c("quarta", "quinta"), "Ana"]
## quarta quinta
##   1500   2000
```

Adicionando colunas com cbind()

```
set.seed(145)
José <- rnorm(nrow(vendas), media, desvio)
vendas <- cbind(vendas, José)
vendas
##          João Maria  Ana José
## segunda 1201   927 1043 1137
## terça   1027   920  833 1213
## quarta  1096  1020 1500 1107
## quinta   915   952 2000 1381
## sexta    787  1039 1293 1213
```

Adicionando linhas com rbind()

```
set.seed(90)
sábado <- rnorm(ncol(vendas), media, desvio)
vendas <- rbind(vendas, sábado)
vendas
##          João Maria  Ana José
## segunda 1201   927 1043 1137
## terça   1027   920  833 1213
## quarta  1096  1020 1500 1107
## quinta   915   952 2000 1381
## sexta    787  1039 1293 1213
## sábado  1015   970  823  856
```

Removendo linhas e colunas: basta não selecionar!

Para remover uma linha, basta sobrescrever a matriz original **não selecionando** a linha que você deseja excluir.

```
# Remove a sexta linha
vendas <- vendas[-6, ]
```

A mesma lógica para remover uma coluna:

```
# Remove a quarta coluna
vendas <- vendas[ ,-4]
```

Aplicando funções nas linhas e colunas: Summary

A função `summary()` fornece várias estatísticas descritivas por coluna:

```
summary(vendas)
##       João           Maria           Ana
##  Min.   : 787   Min.   : 920   Min.   : 833
##  1st Qu.: 915   1st Qu.: 927   1st Qu.:1043
##  Median :1027   Median : 952   Median :1293
##  Mean   :1005   Mean   : 972   Mean   :1334
##  3rd Qu.:1096   3rd Qu.:1020   3rd Qu.:1500
##  Max.   :1201   Max.   :1039   Max.   :2000
```

Aplicando funções nas linhas e colunas: Summary

Se você quise usar `summary()` nas linhas, basta transpor a matriz:

```
summary(t(vendas))
##     segunda          terça          quarta         quinta          sexta
##  Min.   : 927   Min.   : 833   Min.   :1020   Min.   : 915   Min.   : 78
##  1st Qu.: 985   1st Qu.: 877   1st Qu.:1058   1st Qu.: 934   1st Qu.: 91
##  Median :1043   Median : 920   Median :1096   Median : 952   Median :103
##  Mean   :1057   Mean   : 927   Mean   :1205   Mean   :1289   Mean   :104
##  3rd Qu.:1122   3rd Qu.: 973   3rd Qu.:1298   3rd Qu.:1476   3rd Qu.:110
##  Max.   :1201   Max.   :1027   Max.   :1500   Max.   :2000   Max.   :129
```

Aplicando funções nas linhas e colunas: rowSums e rowMeans

O R também já fornece duas funções para somas e médias por linhas:

```
# Total de vendas por dia
rowSums(vendas)
## segunda   terça  quarta  quinta   sexta
##    3171    2780    3616    3867    3119

# Média das vendas por dia
rowMeans(vendas)
## segunda   terça  quarta  quinta   sexta
##    1057     927    1205    1289    1040
```

Aplicando funções nas linhas e colunas: colSums e colMeans

E somas e médias por colunas:

```
# Total de vendas por vendedor
colSums(vendas)
##  João Maria   Ana
##  5025  4858  6670

# Média de vendas por vendedor
colMeans(vendas)
##  João Maria   Ana
##  1005   972  1334
```

Mas e se eu quiser aplicar uma função diferente de `sum()` ou `mean()`?

Aplicando funções nas linhas e colunas: `apply()`

Uma função bastante útil no R é a função `apply()`. Ela permite que você aplique uma função nas linhas ou colunas de um objeto. Sua estrutura básica é a seguinte:

```
apply(matriz, dimensão, função)
```

- no primeiro argumento passamos a matriz em que desejamos fazer as operações;
- no segundo dizemos se queremos operar por linha (1) ou por coluna (2); e,
- no terceiro argumento passamos a função que queremos aplicar.

Aplicando funções nas linhas e colunas: apply()

Qual foi o valor máximo vendido em cada dia (linha)?

```
apply(vendas, 1, max)
## segunda   terça  quarta  quinta   sexta
##    1201    1027    1500    2000    1293
```

Qual foi o menor valor de cada vendedor (colunas)?

```
apply(vendas, 2, min)
##  João Maria   Ana
##   787   920   833
```

Aplicando funções nas linhas e colunas: apply()

Você pode aplicar a função que desejar

```
apply(vendas, 1, sd)
## segunda   terça  quarta  quinta   sexta
##     138      97     258     616     253

apply(vendas, 2, sd)
##  João Maria   Ana
##   160    54   449
```

Mais de duas dimensões? Use o array!

Quer trabalhar com mais de duas dimensões? A solução para isso no R é o array. Vejamos um exemplo:

```
# Criando um array com 3 dimensões
# '2 tabelas' com 4 linhas 2 colunas e
meu_array <- array(1:16, dim = c(4,2,2))

# Selecionando nas 3 dimensões
meu_array[1:2, 2, 2]
```

Veremos casos concretos de uso do arrays na parte de manipulação de dados.

Matrizes, como vetores, tem que ter todos os elementos iguais!

Suponha que você queira incluir a variável sexo na nossa matriz. Note que até agora todas nossas variáveis eram numéricas... o que acontecerá se incluirmos uma variável não numérica?

```
vendas2 <- t(vendas)
vendas2
##       segunda terça quarta quinta sexta
## João     1201  1027   1096    915   787
## Maria     927   920   1020    952  1039
## Ana      1043   833   1500   2000  1293

sexo <- c("m", "f", "f")
vendas2 <- cbind(vendas2, sexo)

is.character(vendas2)
## [1] TRUE
```

Matrizes, como vetores, tem que ter todos os elementos iguais!

Todos os valores da matriz foram transformados em texto! O que ocorreu aqui é que a matriz, tal como um vetor, tem que ter todos seus elementos iguais. O que fazer, então, se quisermos montar uma base de dados com várias variáveis de classes diferentes? Neste caso a estrutura ideal é o `data.frame`, que veremos a seguir.

Exercícios

Sua vez.

- Crie o vetor `x <- 1:16`. A partir de `x`, crie uma matriz `m` com dimensões 4 x 4 utilizando a função `matrix()`. Em seguida, transforme `x` em uma matriz atribuindo a `x` as dimensões `c(4,4)`. As matrizes são idênticas?
- Selecione as linhas 1 a 4 e colunas 2 e 4 da matriz `m`. Selecione as linhas em que a coluna 1 seja menor do que 3. Selecione os elementos de `m` menores do que 10.
- Eleve ao quadrado todos os valores pares da matriz (para verificar se um número é par, verifique se o resto da divisão do número por 2 é igual a zero – para calcular o resto da divisão, use %%).

Soluções

```
x <- 1:16

m <- matrix(x, ncol = 4)

dim(x) <- c(4, 4)

identical(x, m)

m[1:4, c(2,4)]

m[m[,1] < 3, ]

m[m < 10]

indice <- m %% 2 == 0
m[indice] <- m[indice]^2
m
```

Data Frames: seu banco de dados no R

Por que um `data.frame`?

Até agora temos utilizado apenas dados de uma mesma classe, armazenados ou em um vetor ou em uma matriz. Mas uma base de dados, em geral, é feita de dados de diversas classes diferentes: no exemplo anterior, por exemplo, podemos querer ter uma coluna com os nomes dos funcionários, outra com o sexo dos funcionários, outra com valores... como guardar essas informações?

Por que um `data.frame`?

A solução para isso é o `data.frame`. O `data.frame` é talvez o formato de dados mais importante do R. No `data.frame` cada coluna representa uma variável e cada linha uma observação. Essa é a estrutura ideal para quando você tem várias variáveis de classes diferentes em um banco de dados.

Criando um data.frame: `data.frame()`

É possível criar um `data.frame` diretamente com a função `data.frame()`:

```
funcionarios <- data.frame(nome = c("João", "Maria", "José"),
                           sexo = c("M", "F", "M"),
                           salario = c(1000, 1200, 1300),
                           stringsAsFactors = FALSE)
funcionarios
##    nome sexo salario
## 1  João    M    1000
## 2 Maria    F    1200
## 3  José    M    1300
```

Discutiremos a opção `stringsAsFactors = FALSE` mais a frente.

Criando um data.frame: `data.frame()`

Vejamos a estrutura do `data.frame`. Note que cada coluna tem sua própria classe.

```
str(funcionarios)
## 'data.frame':    3 obs. of  3 variables:
##  $ nome   : chr  "João" "Maria" "José"
##  $ sexo   : chr  "M" "F" "M"
##  $ salario: num  1000 1200 1300
```

Criando um data.frame: `data.frame()`

O que ocorreria com o `data.frame` `funcionarios` se o transformássemos em uma matriz? Vejamos:

```
as.matrix(funcionarios)
##      nome    sexo salario
## [1,] "João"  "M"  "1000"
## [2,] "Maria" "F"  "1200"
## [3,] "José"  "M"  "1300"
```

Perceba que todas as variáveis viraram `character`! Exatamente por isso precisamos de um `data.frame` neste caso.

Criando um data.frame: as.data.frame()

Também é possível criar um `data.frame` com a função `as.data.frame()`. Voltando ao exemplo das vendas:

```
df.vendas <- as.data.frame(t(vendas))
df.vendas
##       segunda terça quarta quinta sexta
## João     1201  1027   1096    915   787
## Maria     927   920   1020    952  1039
## Ana      1043   833   1500   2000  1293

str(df.vendas)
## 'data.frame':    3 obs. of  5 variables:
##  $ segunda: num  1201 927 1043
##  $ terça  : num  1027 920 833
##  $ quarta : num  1096 1020 1500
##  $ quinta : num  915 952 2000
##  $ sexta  : num  787 1039 1293
```

Manipulando `data.frames` como matrizes

Ok, temos mais um objeto do R, o data.frame ... vou ter que reaprender tudo novamente? Não!

Você pode manipular data.frames como se fossem matrizes!

Praticamente tudo o que vimos para selecionar e modificar elementos em matrizes funciona no `data.frame`.

Manipulando data.frames como matrizes

Selecionando linhas e colunas:

```
# tudo menos linha 1
funcionarios[-1, ]
##    nome sexo salario
## 2 Maria    F    1200
## 3  José    M    1300

# seleciona primeira linha e primeira coluna (vetor)
funcionarios[1, 1]
## [1] "João"

# seleciona primeira linha e primeira coluna (data.frame)
funcionarios[1, 1, drop = FALSE]
##   nome
## 1 João
```

Manipulando data.frames como matrizes

Alterando valores do data.frame como se fosse uma matriz.

```
# aumento de salario para o João
funcionarios[1, "salario"] <- 1100

funcionarios
##    nome sexo salario
## 1  João    M    1100
## 2 Maria    F    1200
## 3  José    M    1300
```

Nomes de linhas e colunas

O `data.frame` sempre terá rownames e colnames.

```
rownames(funcionarios)
## [1] "1" "2" "3"

colnames(funcionarios)
## [1] "nome"    "sexo"    "salario"
```

Detalhe: o `names` trata-se de nomes das colunas do data.frame. **Os elementos do data.frame são seus vetores coluna**.

```
names(funcionarios)
## [1] "nome"    "sexo"    "salario"
```

Extra do data.frame: selecionando e modificando com $ e [[ ]]

Outras formas alternativas de selecionar colunas em um `data.frame` são o `$` e o `[[ ]]`:

```
# Seleciona coluna nome
funcionarios$nome
## [1] "João"  "Maria" "José"

funcionarios[["nome"]]
## [1] "João"  "Maria" "José"

# Seleciona coluna salario
funcionarios$salario
## [1] 1100 1200 1300

funcionarios[["salario"]]
## [1] 1100 1200 1300
```

Tanto o `$` quanto o `[[ ]]` **sempre** retornam um vetor como resultado.

Extra do data.frame: selecionando e modificando com $ e [[ ]]

Também é possível alterar a coluna combinando $ ou [[ ]] com <-:

```
# outro aumento para o João
funcionarios$salario[1] <- 1150

# equivalente
funcionarios[["salario"]][1] <- 1150
funcionarios
##    nome sexo salario
## 1  João    M    1150
## 2 Maria    F    1200
## 3  José    M    1300
```

Extra do data.frame: retornando sempre um data.frame com [ ]

Se você quiser garantir que o resultado da seleção será sempre um data.frame use `drop = FALSE` ou selecione sem a vírgula:

```
# Retorna data.frame
funcionarios[ ,"salario", drop = FALSE]
##   salario
## 1    1150
## 2    1200
## 3    1300

# Retorna data.frame
funcionarios["salario"]
##   salario
## 1    1150
## 2    1200
## 3    1300
```

Tabela resumo: selecionando uma coluna em um data.frame

Resumindo as formas de seleção de uma coluna de um data.frame.

Operador	Descrição
`df[ ,"x"]`	Retorna **vetor** `x` do data.frame `df`.
`df$x`	Retorna **vetor** `x` do data.frame `df`.
`df[["x"]]`	Retorna **vetor** `x` do data.frame `df`.
`df[ ,"x", drop = FALSE]`	Retorna um **data.frame** com a coluna `x`.
`df["x"]`	Retorna um **data.frame** com a coluna `x`.

Criando colunas novas

Com $:

```
funcionarios$escolaridade <- c("Ensino Médio", "Graduação", "Mestrado")
```

Com [,]:

```
funcionarios[, "experiencia"] <- c(10, 12, 15)
```

Com [[ ]]:

```
funcionarios[["avaliacao_anual"]] <- c(7, 9, 10)
```

Com cbind():

```
funcionarios <- cbind(funcionarios,
                      prim_emprego = c("sim", "nao", "nao"),
                      stringsAsFactors = FALSE)
```

Criando colunas novas

Vejamos como ficou nosso data.frame com as novas colunas:

```
funcionarios
##    nome sexo salario escolaridade experiencia avaliacao_anual prim_emprego
## 1  João    M    1150 Ensino Médio          10               7          sim
## 2 Maria    F    1200    Graduação          12               9          nao
## 3  José    M    1300     Mestrado          15              10          nao
```

Removendo colunas

Atribuindo NULL:

```
# deleta coluna prim_emprego
funcionarios$prim_emprego <- NULL
```

Ou selecionando todas colunas menos as que você não quer:

```
# deleta colunas 4 e 6
funcionarios <- funcionarios[, c(-4, -6)]
```

Adicionando linhas

Uma forma simples de adicionar linhas é atribuir a nova linha com <-. Mas cuidado! O que irá acontecer com o `data.frame` com o código abaixo?

```
# CUIDADO!
funcionarios[4, ] <- c("Ana", "F", 2000,  15)
```

Note que nosso data.frame inteiro se transformou em texto! Você sabe explicar por que isso aconteceu?

```
str(funcionarios)
## 'data.frame':    4 obs. of  4 variables:
##  $ nome       : chr  "João" "Maria" "José" "Ana"
##  $ sexo       : chr  "M" "F" "M" "F"
##  $ salario    : chr  "1150" "1200" "1300" "2000"
##  $ experiencia: chr  "10" "12" "15" "15"
```

Adicionando linhas

Antes de prosseguir, transformemos as colunas `salario` e `experiencia` em números novamente:

```
funcionarios$salario <- as.numeric(funcionarios$salario)

funcionarios$experiencia <- as.numeric(funcionarios$experiencia)
```

Adicionando linhas

Se os elementos forem de classe diferente, use a função `data.frame` para evitar coerção:

```
funcionarios[4, ] <- data.frame(nome = "Ana", sexo = "F",
                                salario = 2000, experiencia = 15,
                                stringsAsFactors = FALSE)
```

Também é possível adicionar linhas com `rbind()`:

```
rbind(funcionarios,
      data.frame(nome = "Ana", sexo = "F",
                 salario = 2000,  experiencia = 15,
                 stringsAsFactors = FALSE))
```

Atenção! Não fique aumentando um `data.frame` de tamanho adicionando linhas ou colunas. Sempre que possível pré-aloque espaço!

Removendo linhas

Para remover linhas, basta selecionar apenas aquelas linhas que você deseja manter:

```
# remove linha 4 do data.frame
funcionarios <- funcionarios[-4, ]
```

```
# remove linhas em que salario <= 1150
funcionarios <- funcionarios[funcionarios$salario > 1150, ]
```

Filtrando linhas com vetores logicos

Relembrando: se passarmos um vetor lógico na dimensão das linhas, selecionamos apenas aquelas que são `TRUE`. Assim, por exemplo, se quisermos selecionar aquelas linhas em que a coluna `salario` é maior do que um determinado valor, basta colocar esta condição como filtro das linhas:

```
# Apenas linhas com salario > 1000
funcionarios[funcionarios$salario > 1000, ]
##    nome sexo salario experiencia
## 2 Maria    F    1200          12
## 3  José    M    1300          15

# Apenas linhas com sexo == "F"
funcionarios[funcionarios$sexo == "F", ]
##    nome sexo salario experiencia
## 2 Maria    F    1200          12
```

Alternativas para `rbind()` e `cbind()` para data.frames

As funções `rbind()` e `cbind()` podem não ser muito eficientes. As funções `bind_rows()` e `bind_cols()` do pacote `dplyr` (que veremos mais a frente) são alternativas interessantes. Veremos mais sobre o `dplyr` na aula de manipulações de `data.frames`.

Funções de conveniência: `subset()`

Uma função de conveniência para selecionar linhas e colunas de um `data.frame` é a função `subset()`, que tem a seguinte estrutura:

```
subset(nome_do_data_frame,
       subset = expressao_logica_para_filtrar_linhas,
       select = nomes_das_colunas,
       drop   = simplicar_para_vetor?)
```

Funções de conveniência: subset()

Vejamos alguns exemplos:

```
# funcionarios[funcionarios$sexo == "F",]
subset(funcionarios, sexo == "F")
##    nome sexo salario experiencia
## 2 Maria    F    1200          12

# funcionarios[funcionarios$sexo == "M", c("nome", "salario")]
subset(funcionarios, sexo == "M", select = c("nome", "salario"))
##   nome salario
## 3 José    1300
```

Funções de conveniência: `with`

A função `with()` permite que façamos operações com as colunas do `data.frame` sem ter que ficar repetindo o nome do `data.frame` seguido de `$` , `[ , ]` ou `[[]]` o tempo inteiro. Apenas para ilustrar:

```
# Com o with
with(funcionarios, (salario^3 - salario^2)/log(salario))
## [1] 2.4e+08 3.1e+08

# Sem o with
(funcionarios$salario^3 - funcionarios$salario^2)/log(funcionarios$salario)
## [1] 2.4e+08 3.1e+08
```

Funções de conveniência: `with`

Quatro formas de fazer a mesma coisa (pense em outras formas possíveis):

```
subset(funcionarios, sexo == "M", select = "salario", drop = TRUE)
## [1] 1300

with(funcionarios, salario[sexo == "M"])
## [1] 1300

funcionarios$salario[funcionarios$sexo == "M"]
## [1] 1300

funcionarios[funcionarios$sexo == "M", "salario"]
## [1] 1300
```

Aplicando funções no data.frame: o que funcionava nas matrizes continua valendo

As funções `rowSums()`, `rowMeans()`, `colSums()`, `colMeans()` e `apply()` continuam funcionando normalmente nos `data.frames`. Teste os seguintes códigos no `data.frame` `df.vendas`:

```
rowSums(df.vendas)
##  João Maria   Ana
##  5025  4858  6670

colSums(df.vendas)
## segunda   terça  quarta  quinta   sexta
##    3171    2780    3616    3867    3119
```

Aplicando funções no data.frame: o que funcionava nas matrizes continua valendo

```
apply(df.vendas, 1, max)
##  João Maria   Ana
##  1201  1039  2000

apply(df.vendas, 2, max)
## segunda   terça  quarta  quinta   sexta
##    1201    1027    1500    2000    1293
```

Aplicando funções no data.frame: `sapply` e `lapply`, funções nas colunas (elementos)

Outras duas funções bastante utilizadas no R são as funções `sapply()` e `lapply()`.

- As funções `sapply` e `lapply` aplicam uma função em cada elemento de um objeto.
- Como vimos, os elementos de um `data.frame` são suas colunas. Deste modo, as funções `sapply` e `lapply` aplicam uma função nas colunas de um data.frame.
- A diferença entre uma e outra é que a primeira tenta **s**implificar o resultado enquanto que a segunda sempre retorna uma lista.

Aplicando funções no data.frame: `sapply` e `lapply`, funções nas colunas (elementos)

Testando no nosso `data.frame`:

```
sapply(funcionarios[3:4], mean)
##     salario experiencia
##        1250          14

lapply(funcionarios[3:4], mean)
## $salario
## [1] 1250
##
## $experiencia
## [1] 14
```

Filtrando variáveis antes de aplicar funções: `Filter()`

Como `data.frames` podem ter variáveis de classe diferentes, muitas vezes é conveniente filtrar apenas aquelas colunas de determinada classe (ou que satisfaçam determinada condição). A função `Filter()` (note o F maiúsculo!) é uma maneira rápida de fazer isso:

```
Filter(is.numeric, funcionarios)
##   salario experiencia
## 2    1200          12
## 3    1300          15

Filter(is.character, funcionarios)
##    nome sexo
## 2 Maria    F
## 3  José    M
```

Filtrando variáveis antes de aplicar funções: `filter()`

Exemplo: aplicando a função média e máximo apenas nas colunas numéricas.

```
sapply(Filter(is.numeric, funcionarios), mean)
##     salario experiencia
##        1250          14

sapply(Filter(is.numeric, funcionarios), max)
##     salario experiencia
##        1300          15
```

Manipulando data.frames

Ainda temos muita coisa para falar de manipulação de data.frames e isso merece um espaço especial. Veremos além de outras funções base do R alguns pacotes importantes como `dplyr`, `reshape2` e `tidyr` em uma seção separada do nosso curso.

Exercícios

Sua vez.

- Crie o seguinte `data.frame`: `df <- data.frame(x = letters[1:4], y = 1:4, stringsAsFactors = FALSE)`.
- Adicione a coluna `y2` com o resultado de `y^2`. Remova a coluna `y2`.
- Adicione uma linha em que `x = "e"` e `y = 5`. Remova esta linha.
- Selecione a linha em que `x == "a"`. Selecione apenas as linhas em que `y < 3`. Modifique a coluna `y` fazendo com que os elementos em que `y >= 3` sejam iguais a `y^3`.

Soluções

```r
df <- data.frame(x = letters[1:4], y = 1:4, stringsAsFactors = FALSE)

df$y2 <- df$y^2

df$y2 <- NULL

df[5, ] <- data.frame(x = "e", y = 5, stringsAsFactors = FALSE)
df <- df[-5, ]

df[df$x == "a", ]

df[df$y < 3, ]

df$y[df$y >= 3] <- df$y[df$y >= 3]^3
```

Listas: juntando vários coisas diferentes em um só objeto.

Para que servem listas?

A lista é a estrutura mais flexível do R. Uma lista pode conter objetos arbitrários, como data.frames, matrizes vetores e, inclusive, outras listas. Em geral, as listas são utilizadas para armazenar vários objetos diferentes que tenham algo em comum, como por exemplo, os resultados de cálculos estatísticos.

Para que servem listas?

Vejamos um exemplo prático de uma lista, o resultado da função `lm()` do R. Resultados omitidos pois são muito grandes.

```
# gera variáveis aleatórias
set.seed(1)
x <- rnorm(100)
y <- 10 + 2*x + rnorm(100)

# roda regressão linear
modelo <- lm(y ~ x)
summary(modelo)

# Veja que o objeto modelo é uma lista!
str(modelo, max.level = 1)
```

Criando uma lista: `list()`

Para criar uma lista utiliza-se a função `list()`:

```
minha_lista <- list(x = 1:10,
                    y = letters[1:5],
                    z = list(a = 1,
                             b = list(c = 2)))
str(minha_lista)
## List of 3
##  $ x: int [1:10] 1 2 3 4 5 6 7 8 9 10
##  $ y: chr [1:5] "a" "b" "c" "d" ...
##  $ z:List of 2
##   ..$ a: num 1
##   ..$ b:List of 1
##   .. ..$ c: num 2
```

Selecionando elementos da lista

Selecionando um objeto da lista:

```
minha_lista$x
##  [1]  1  2  3  4  5  6  7  8  9 10

minha_lista[["x"]]
##  [1]  1  2  3  4  5  6  7  8  9 10

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

Selecionando uma lista com certos elementos

Selecionando uma lista com o objeto (note a diferença):

```
minha_lista["x"]
## $x
##  [1]  1  2  3  4  5  6  7  8  9 10

minha_lista[c(1,2)]
## $x
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $y
## [1] "a" "b" "c" "d" "e"
```

Selecionando elementos da lista

As seleções podem ser concatenadas:

```
# seleciona primeiro o elemento z
# depois o elemento b (do elemento z)
# e depois o elemento c (do elemento b)
minha_lista$z$b$c
## [1] 2


# idem anterior
minha_lista[["z"]][["b"]][["c"]]
## [1] 2
```

Tabela resumo: selecionando elementos de uma lista

Resumindo as formas de seleção de um elemento de uma lista:

Operador	Descrição
`lista$x`	Retorna o objeto **x** da lista.
`lista[["x"]]`	Retorna o objeto **x** da lista.
`lista["x"]`	Retorna uma **lista** com o objeto `x`.

Adicionando elementos em uma lista: combine $ ou [[]] com <-

```
dados_da_empresa <- list(vendas = df.vendas,
                         funcionarios = funcionarios)

dados_da_empresa$comentario <- "Um comentario"

str(dados_da_empresa, max.level = 1)
## List of 3
##  $ vendas      :'data.frame':    3 obs. of  5 variables:
##  $ funcionarios:'data.frame':    2 obs. of  4 variables:
##  $ comentario  : chr "Um comentario"
```

Removendo elementos da lista: use NULL ou selecione todos exceto o que quer remover

```
# remove o elemento 'comentario' da lista
dados_da_empresa$comentario <- NULL

dados_da_empresa
## $vendas
##       segunda terça quarta quinta sexta
## João     1201  1027   1096    915   787
## Maria     927   920   1020    952  1039
## Ana      1043   833   1500   2000  1293
##
## $funcionarios
##    nome sexo salario experiencia
## 2 Maria    F    1200          12
## 3  José    M    1300          15
```

Aplicando funções em uma lista: sapply e lapply

As funções `sapply()` e `lapply()` aplicam uma função a cada elemento da lista:

```
set.seed(1)

# cria uma lista com 3 matrizes 2 x 2
lista_de_matrizes <- list(x = matrix(rnorm(4), ncol = 2),
                          y = matrix(rnorm(4), ncol = 2),
                          z = matrix(rnorm(4), ncol = 2))

# calcula o determinante das 3 matrizes ao mesmo tempo
sapply(lista_de_matrizes, det)
##     x     y     z
## -0.85  0.64  0.69
```

Exercícios

Sua vez.

- Considere a lista `minha_lista` criada anteriormente. Adicione uma lista `outra_lista` em `minha_lista` contendo um vetor de inteiros de 1 a 10 e e o data.frame `df.vendas`.
- Selecione ao mesmo tempo os elementos `outra_lista` e `x` de `minha_lista` e salve o resultado em outro objeto.
- Remova a `outra_lista` de `minha_lista`.

Soluções

```
minha_lista$outra_lista <- list(1:10, df.vendas)

outro_objeto <- minha_lista[c("outra_lista", "x")]
str(outro_objeto)

minha_lista[["outra_lista"]] <- NULL
```

Programação em R

Julho, 2016

Funções: definição, argumentos e operadores binários

Por que funções?

Uma das grandes vantagens de usar uma linguagem de programação é automatizar o seu trabalho ou análise. Você será capaz de realizar grande parte do trabalho utilizando as funções internas do R ou de pacotes de terceiros em um script. Entretanto, você ganha ainda mais flexibilidade e agilidade criando suas próprias funções.
Vejamos um exemplo de *script*:

```
# queremos converter para numeric, retirar os dados discrepantes
# dividir por 1000 e arredondar o resultado
precos <- c("0.1", "1250.55", "2346.87", "3467.40", "10000000")
precos <- as.numeric(precos)
precos <- precos[!(precos < 1 | precos > 10000)]
precos <- precos/1000
precos <- round(precos)
precos
## [1] 1 2 3
```

Por que funções?

Nosso *script* faz o trabalho corretamente. Mas imagine que você queira realizar o mesmo procedimento com um vetor de preços diferente, digamos, `precos2`. Da forma como o seu código foi feito, você terá que **copiar e colar** os comandos e substituir os nomes.

```
# novo vetor de precos
precos2 <- c("0.0074", "5547.85", "2669.98", "8789.45", "150000000")
precos2 <- as.numeric(precos2)
precos2 <- precos2[!(precos2 < 1 | precos2 > 10000)]
precos2 <- precos2/1000
precos2 <- round(precos2)
precos2
## [1] 6 3 9
```

Note como isto é ineficiente. Além do trabalho de copiar e colar todo o seu código para cada análise diferente que você desejar fazer, você ainda estará sujeito a diversos erros operacionais, como esquecer de trocar um dos nomes ao copiar e colar em cada análise.

Por que funções?

O ideal, aqui, é criar uma ***função*** que realize este trabalho! Daqui a pouco veremos como fazer isso, primeiramente vejamos como definir uma função no R.

Definição

Uma função, no R, é definida da seguinte forma:

```
nomeDaFuncao <- function(arg1, arg2, arg3 = default3,  ...){
  # corpo da função: uma série de comados válidos.
  return(resultado) # opcional
}
```

- o comando `function()` diz para o R que você está definindo uma função.
- os valores dentro dos parênteses de `function()` são os argumentos (ou parâmetros) da função. Argumentos podem ter valores default, que são definidos com o sinal de igualdade (no caso `arg3` tem como default o valor `default3`). Existe um parâmetro coringa muito útil, o `...`, o qual veremos mais a frente.
- dentro das chaves encontra-se o "corpo" da função, isto é, uma série de comandos válidos que serão realizados.
- o comando `return()` encerra a função e retorna seu argumento. Como veremos, o `return()` é opcional. Caso omitido, a função retorna o último objeto calculado. ***A função só pode retornar um objeto***!

Definição

Criemos nossas primeiras funções:

```
quadrado <- function(x){
  x ^ 2
}
quadrado(3)
## [1] 9
## se for na mesma linha não precisa das chaves
quadrado <- function(x) x ^ 2
quadrado(3)
## [1] 9

## adicionando mais argumentos
elevado_n <- function(x,n) x ^ n
elevado_n(3, 3)
## [1] 27
```

Definição

Funções criam um ambiente local e, em geral, ***não alteram diretamente o objeto ao qual são aplicadas***. Elas tomam um objeto como argumento e criam outro objeto, modificado, como resultado. Na maior parte dos casos, a idéia é não ter efeitos colaterais com objetos fora da função.

```
x <- 10
elevado_n(x, 2) # isso altera o valor de x?
## [1] 100
x # não
## [1] 10

# se você quer salvar o resultado
# tem que atribuir a outro objeto
y <- elevado_n(x, 2)
y
## [1] 100
```

Exercícios

Sua vez.

- Crie uma função `soma_e_subtrai(x, y)` que receba um parâmetro `x` e outro `y` e retorne o resultado tanto da soma `x + y` quanto da subtração `x - y`. Lembre que uma função no `R` retorna apenas um objeto, você terá que combinar os resultados (em um vetor, por exemplo) antes de retorná-los.
- Crie os objetos `z <- 1` e `w <- 2`. Teste sua função usando `z` e `w`.

Exercícios

```
soma_e_subtrai <- function(x, y){
  soma <- x + y
  subtracao <- x - y
  resultado <- c(soma = soma, subtracao = subtracao)
  return(resultado)
}

z <- 1
w <- 2

soma_e_subtrai(z, w)
##      soma subtracao
##         3        -1
```

Voltando ao exemplo

Vamos voltar ao nosso exemplo inicial. Montemos uma função que realiza o tratamento dos dados visto anteriormente:

```
limparDados <- function(dados){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < 1 | dados > 10000)]
  dados <- dados/1000
  dados <- round(dados)
  return(dados)
}
ls() # note que a função foi criada
##  [1] "elevado_n"      "limparDados"    "precos"         "precos2"
##  [5] "quadrado"       "soma_e_subtrai" "w"              "x"
##  [9] "y"              "z"
```

Voltando ao exemplo

Vejamos em detalhes:

- o comando `function()` diz para o R que você está definindo uma função.
- os valores dentro dos parênteses de `function()` são os argumentos da função. No nosso caso definimos um único argumento chamado `dados`.
- dentro das chaves encontra-se o "corpo" da função, isto é, as operações que serão realizadas. Neste caso, transformamos em `numeric`, retiramos dados menores que 1 e maiores do que 10000, dividimos por 1000 e arredondamos.
- a função `return()` encerra a função e retorna o vetor `dados` modificado.

Voltando ao exemplo

Pronta a função, sempre que você quiser realizar essas operações em um vetor diferente, basta utilizar `limparDados`.

```
precos3 <-  c("0.02", "4560.45", "1234.32", "7894.41", "12000000")
precos4 <- c("0.001", "1500000", "1200.9", "2000.2", "4520.5")
precos5 <- c("0.05", "1500000", "1000000", "7123.4", "9871.5")

# limpando os dados
limparDados(precos3)
## [1] 5 1 8
limparDados(precos4)
## [1] 1 2 5
limparDados(precos5)
## [1]  7 10
```

Voltando ao exemplo

Bem melhor do que o *script*.
Note que tínhamos 3 vetores diferentes e bastou chamar a função três vezes, ao invés de ter que copiar e colar três vezes o código. Note, também, que se houver algum erro, **temos que corrigir apenas na função**.
E podemos refinar ainda mais `limparDados`.
Por exemplo, da forma como a função está escrita, os valores de corte de mínimo e de máximo serão sempre 1 e 10000; além disso, os resultados sempre serão divididos por 1000.
E se quisermos modificar esses valores? Basta colocá-los também como argumentos.

Mais argumentos

Colocando mais argumentos:

```
limparDados <- function(dados, min, max, div){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- round(dados)
  return(dados)
}
```

Agora você pode alterar os valores de `min`, `max` e `div` ao aplicar a função.

Mais argumentos

```
precos3 <-  c("0.02", "4560", "1234", "7894", "12000000")

limparDados(precos3, min = 0, max = 5000, div = 2)
## [1]    0 2280  617

limparDados(precos3, min = 0, max = 4000, div = 4)
## [1]   0 308

limparDados(precos3, min = -Inf, max = Inf, div = 1)
## [1] 0.0e+00 4.6e+03 1.2e+03 7.9e+03 1.2e+07
```

Mais argumentos

Note que argumentos são nomeados. Se você colocar os argumentos com seus nomes, a ordem dos argumentos não importa. Entretanto, você pode omitir os nomes dos argumentos, desde que os coloque em ordem correta.

```
#  argumentos em ordem diferente
limparDados(max = 5000, div = 2, min = 0, dados = precos3)
## [1]    0 2280  617

# argumentos sem nomes (na ordem)
limparDados(precos3, 0, 4000, 4)
## [1]   0 308
```

Argumentos Default

Com mais argumentos, se você esquecer de especificar algum, ocorrerá um erro:

```
limparDados(precos3, max = 5000, div = 1)
## Error in limparDados(precos3, max = 5000, div = 1): argumento "min" ausente, se
```

Para sanar isto, basta definir valores padrão (default).

Argumentos Default

Colocando argumentos padrão (default):

```
limparDados <- function(dados, min = 1, max = 10000, div = 1000){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- round(dados)
  return(dados)
}
```

Podemos usar a função omitindo os argumentos que possuem default.

Argumentos Default

```
# usa o default para min
limparDados(precos3, max = 5000, div = 1)
## [1] 4560 1234

# usa o default para min e div
limparDados(precos3, max = Inf)
## [1]     5     1     8 12000

# usa o default para tudo
limparDados(precos3)
## [1] 5 1 8
```

Exercícios

Sua vez.

- Crie uma função `divide(x, divisor)` que divide `x` pelo `divisor` e retorne o resultado. Faça com que a função tenha um default de `divisor = 1` caso o usuário não passe nada. Teste sua função.
- Crie uma função `media(x, remove.nas)` que calcule a media de `x` usando a função `mean()` e que possa remover ou não os `NA`s dependendo do parâmetro `remove.nas` (lembre que a função `mean` já tem o argumento `na.rm` para remover `NA`s, você vai simplesmente repassar o argumento). Faça com que a função tenha como default `remove.nas = TRUE`. Teste sua função.

Exercícios

```
divide <- function(x, divisor = 1){
  resultado <- x/divisor
  return(resultado)
}

divide(10)
## [1] 10
divide(10, 5)
## [1] 2

media <- function(x, remove.nas = TRUE) mean(x, na.rm = remove.nas)
media(c(NA, 1:10))
## [1] 5.5
```

Funções podem ser argumentos

Funções também podem ser passadas como argumentos de funções. Por exemplo, suponha que você não queira sempre usar o `round()` para arredondamento. Você pode colocar a função final que é aplicada a `dados` como um argumento.

```
limparDados <- function(dados, min = 1,
                        max = 10000, div = 1000, funcao = round){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- funcao(dados)
  return(dados)
}
```

Funções podem ser argumentos

Se quisermos usar a função `floor()` ao invés de `round()`, basta trocar o argumento `funcao`.

```
# usou os defaults
limparDados(precos3)
## [1] 5 1 8

# usa floor ao invés de round
limparDados(precos3, funcao = floor)
## [1] 4 1 7

# funcao anonima que pega x e retorna x (não faz nada com x)
limparDados(precos3, funcao = function(x) x)
## [1] 4.6 1.2 7.9
```

Funções anônimas

Como vimos no exemplo anterior, você pode definir uma função no próprio argumento. Estas funções são chamadas de **anônimas**.

```
limparDados(precos3, funcao = function(x) x)
## [1] 4.6 1.2 7.9
limparDados(precos3, funcao = function(x) x ^ 2)
## [1] 20.8  1.5 62.3
limparDados(precos3, funcao = function(x) log(x + 1))
## [1] 1.7 0.8 2.2
limparDados(precos3, funcao = function(x) {
  x <- round(x)
  x <- as.complex(x)
  x <- (-x) ^ (x/10)
} )
## [1]  0.00-2.24i  0.95-0.31i -4.27-3.10i
```

O ...

O R tem ainda um argumento *coringa* os "três pontos" `...`.
O `...` permite repassar argumentos para outras funções dentro da sua função, sem necessariamente ter que elencar todas as possibilidades de antemão.

```
limparDados <- function(dados, min = 1,
                        max = 10000, div = 1000, funcao = round, ...){
  dados <- as.numeric(dados)
  dados <- dados[!(dados < min | dados > max)]
  dados <- dados/div
  dados <- funcao(dados, ...)
  return(dados)
}
```

Agora podemos passar argumentos para `funcao(dados, ...)`

O ...

A função `round()`, por exemplo, tem o argumento `digits`. Ou a nossa função `elevado_n()` tem o argumento `n`. Podemos repassar esses argumentos para essas funções por meio do `...`.

```
limparDados(precos3)
## [1] 5 1 8
limparDados(precos3, digits = 1)
## [1] 4.6 1.2 7.9
limparDados(precos3, funcao = elevado_n, n = 2)
## [1] 20.8  1.5 62.3
```

Operadores binários

Lembra que falamos que +, no R, na verdade é a função `'+'(x,y)`? Funções deste tipo são chamadas de operadores binários. E, no R, você também pode definir seus operadores binários, com auxílio do %.
Vamos fazer um operador binário que cole textos:

```
"%+%" <- function(x, y) paste(x, y)
```

Agora podemos colar textos usando %+%:

```
"colando" %+% "textos" %+% "com nosso" %+% "operador"
## [1] "colando textos com nosso operador"
```

Operadores binários

Vejamos outro exemplo:

```
"%depois%" <- function(x, fun) fun(x)
```

Olhe que interessante:

```
set.seed(10)
x <- rnorm(100)
sqrt(exp(mean(x)))
## [1] 0.93
x %depois% mean %depois% exp %depois% sqrt
## [1] 0.93
```

A imaginação é o limite.

Exercícios

Sua vez.

- Defina uma função que retorne o mínimo, a mediana e o máximo de um vetor. Faça com que a função lide com NA's e que isso seja um argumento com default;
- Defina uma versão "operador binário" da função `rep`. Faça com que tenha seguinte sintaxe: `x %rep% n` retorna o objeto `x` repetido `n` vezes.
- Defina uma função que normalize/padronize um vetor (isto é, subtraia a média e divida pelo desvio-padrão). Faça com que a função tenha a opção de ignorar NA's. Permita ao usuário escolher outros parâmetros para a média (Dica: ...);
- Dados um vetor y e uma matriz de variáveis explicativas X, defina uma função que retorne os parâmetros de uma regressão linear de x contra y, juntamente com os dados originais usados na regressão. (Dicas: use álgebra matricial. Use uma lista para retornar o resultado)

Soluções

```
mmm <- function(x, na.rm = TRUE){

  # calcula min, median, e max, guarda em resultado
  resultado <- c(min(x, na.rm = na.rm),
                 median(x, na.rm = na.rm),
                 max(x, na.rm = na.rm))

  # nomeia o vetor para facilitar consulta
  names(resultado) <- c("min", "mediana", "max")

  #retorna vetor
  return(resultado)
}

mmm(c(1,2,3, NA))
##     min mediana     max
##       1       2       3
```

Soluções

```
"%rep%" <- function(x, n) rep(x, n)


7 %rep% 5
## [1] 7 7 7 7 7
```

Soluções

```r
padronize <- function(x, na.rm = TRUE, ...){

  m <- mean(x, na.rm = na.rm, ...) # calcule a média
  dp <- sd(x, na.rm = na.rm)       # calcule o dp
  pad <- (x - m)/dp                # padronize os dados

  attr(pad, "media") <- m        # guarda a média original p/ consulta
  attr(pad, "dp") <- dp          # guarda o dp original p/ consulta

  return(pad)                    # retorna o vetor pad já com atributos
}

padronize(1:5)
```

Soluções

```r
ols <- function(X, y){

  b <- solve(t(X) %*% X) %*% t(X) %*% y  # ols ((X'X)^-1)X'Y

  # guarda resultados em lista nomeada
  resultado <- list(coef = b, X = X, y = y)

  # retorna resultado
  return(resultado)
}
```

Soluções

Testando nosa função `ols()`:

```
# cria dados simulados
set.seed(10)                          # para reproducibilidade
X <- matrix(rnorm(300), ncol = 3)       # vetor X
y <- X %*% c(3,6,9) + rnorm(100)        # y = Xb + e, b=c(3, 6, 9), e~N(0,1)

# vamos testar a formula
resultado <- ols(X, y)
str(resultado)
resultado$coef
```

Um pouco sobre escopo de funções

Escopo

O R tem o que se chama, em programação, de **escopo léxico**. Grosso modo, quando uma função não encontra um objeto dentro de seu ambiente local, ela procura nos seus ambientes "pais" e, somente se não encontrar nada, retorna um erro.

```
x <- 2 # definimos um x na área de trabalho
func <- function(){
  x <- 3
  # definimos um x dentro da função
  # isso altera o valor de x fora da função?
  print(x)
  # print vai ser 2 ou 3?
  rm(x)
  # removemos o x. Qual x?
  print(x)
  # vai dar erro?
}
```

Escopo

Testando:

```
func()
## [1] 3
## [1] 2
```

Vamos remover x da área de trabalho.

```
rm(x)
```

E agora?

```
func()
## [1] 3
## Error in print(x): objeto 'x' não encontrado
```

Escopo

Vamos complicar o exemplo. Vamos definir uma função `func2()` que define um x localmente e chama a função `func()` definida anteriormente:

```
x <- 500
func2 <- function(){
 x <- 1000
 func() # vai achar quais x?
}

func2()
## [1] 3
## [1] 500
rm(x) # e agora?
func2()
## [1] 3
## Error in print(x): objeto 'x' não encontrado
```

Escopo

A função `func()` não encontra o `x` definido dentro do ambiente local de `func2()` porque o ambiente "pai" de `func()` é o ambiente global.
Não entraremos em detalhes de escopo léxico vs escopo dinâmico, mas é importante você entender um pouco disto pois alguns "bugs" que você irá encontrar na verdade são comportamentos esperados.

Um pouco sobre métodos de funções

Métodos

Um último assunto que veremos brevemente é o uso de métodos em funções no R. Para ilustrar, vejamos a definição da função `plot()`.

```
plot
## function (x, y, ...)
## UseMethod("plot")
## <bytecode: 0x7fa5623b7a70>
## <environment: namespace:graphics>
```

O único comando da função é `UseMethod("plot")`!
O que isso quer dizer?

Métodos

Na verdade, a função `plot()` é uma **função genérica**.
Ela verifica a classe do objeto em que está sendo aplicada e repassa os seus argumentos `(x, y, ...)` para funções específicas de `plot` a depender da classe do objeto. Essas funções específicas, ou métodos, são denominadas `plot.classe.do.objeto`. Se não tiver um método específico para a classe, os argumentos são repassados para a função default (`plot.default`).
Digite os seguintes comandos para ver as funções `plot.default()` e `plot.data.frame()`:

```
plot.default
graphics:::plot.data.frame
```

O mesmo ocorre com a fução `print()`.

Métodos

Para exemplificar, vamos complementar a função `ols()` que criamos anteriormente, e definir que o objeto que ela retorna é de classe "nossa_classe".

```
ols <- function(X, y){
  b <- solve(t(X) %*% X) %*% t(X) %*% y
  resultado <- list(coef = b, X = X, y = y)
  class(resultado) <- "nossa_classe"
  return(resultado)
}
```

Métodos

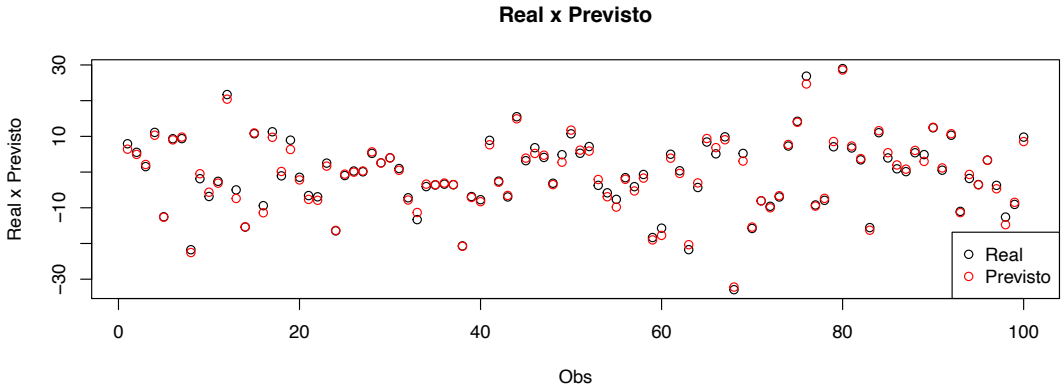
Vamos definir um método de `plot` para a classe "nossa_classe".

```
plot.nossa_classe <- function(x) {
  previsto <- X %*% (x$coef)
  plot(y,
       xlab = "Obs",
       main = "Real x Previsto",
       ylab = "Real x Previsto")
  points(previsto, col = "red")
  legend("bottomright", col = c("black", "red"),
         pch=1, legend = c("Real", "Previsto")
         )
}
```

Quando você chamar a função `plot()` para o resultado da função `ols`, ela automaticamente repassará os argumentos para a função `plot.nossa_classe()`.

Métodos

```
resultado <- ols(X, y)
plot(resultado)
```



Métodos

Outro exemplo: vamos criar métodos para nossa função `limparDados`. Note que a função dá erro se o argumento `dados` for um `data.frame`.

```
df_precos <- data.frame(precos3,
                        precos4,
                        precos5,
                        stringsAsFactors = FALSE)
# vai dar erro
limparDados(df_precos)
## Error in limparDados(df_precos): objeto (list) não é coercível para tipo 'doubl
```

Métodos

Vamos, então, definir a função que criamos como a função default e definir `limparDados` como uma função genérica.

```
# copiando limparDados para limparDados.default
limparDados.default <- limparDados

# definindo limparDados como genérica
limparDados <- function(dados, ...){
  UseMethod("limparDados")
}
```

Métodos

Note que `limparDados` continua funcionando normalmente com os vetores simples.

```
limparDados(precos3)
## [1] 5 1 8
```

Vamos criar um método de `data.frame` para `limparDados` (neste caso vamos usar `lapply()` para aplicar a função em todas as colunas do data.frame):

```
limparDados.data.frame <- function(dados, ...){
  lapply(dados, limparDados, ...)
}
```

Métodos

Agora a função automaticamente procura o método adequado para a classe de objeto ao qual está sendo aplicada.

```
limparDados(precos3)
## [1] 5 1 8
limparDados(df_precos)
## $precos3
## [1] 5 1 8
##
## $precos4
## [1] 1 2 5
##
## $precos5
## [1]  7 10
```

Métodos

Para ver os métodos de uma função, use `methods(funcao)`.

```
# Irá mostrar todos os métodos de print
# dos pacotes carregados
methods(print)

# Irá mostrar todos os métodos de plot
# dos pacotes carregados
methods(plot)
```

Para ver os métodos de uma classe, use `methods(class="classe")`.

```
# Irá mostrar todos os métodos para data.frame
methods(class="data.frame")
```

Programação em R

Julho, 2016

Executando código de maneira condicional: `if()`, `if() else`, `ifelse()`

Há ocasiões em queremos ou precisamos executar parte do código **apenas se alguma condição for atendida**. Veremos três formas de fazer isso: `if()`, `if() else` e `ifelse()`.

O `if()`: estrutura

A estrutura básica:

```
if (condicao) {

  # comandos que
  # serao rodados
  # caso condicao = TRUE

}
```

- O início do código se dá com o comando `if` seguido de parênteses e chaves;
- Dentro do parênteses temos uma condição lógica, que deverá ter como resultado ou `TRUE` ou `FALSE`;
- Dentro das chaves temos o bloco de código que será executado se – e somente se – a condição do parênteses for `TRUE`.

O `if()`: exemplo

```
# vetores de condição lógica
cria_x <- TRUE
cria_y <- FALSE
# só executa se cria_x = TRUE
if (cria_x) {
  x <- 1
}
# só executa se cria_y = TRUE
if (cria_y) {
  y <- 1
}

# note que x foi criado mas y não
exists("x"); exists("y")
## [1] TRUE
## [1] FALSE
```

O `if()` com o `else`

Outra forma de executar códigos de maneira condicional é acrescentar ao `if()` o opcional `else`. Estrutura básica:

```
if (condicao) {
  # comandos que
  # serao rodados
  # caso condicao = TRUE
} else {
  # comandos que
  # serao rodados
  # caso condicao = FALSE
}
```

O `if()` com o `else`

Em detalhes:

- O início do código se dá com o comando `if` seguido de parênteses e chaves;
- Dentro do parênteses temos uma condição lógica, que deverá ter como resultado ou `TRUE` ou `FALSE`;
- Dentro das chaves do `if()` temos um bloco de código que será executado se – e somente se – a condição do parênteses for `TRUE`.
- Logo em seguida temos o `else` seguido de chaves;
- Dentro das chaves do `else` temos um bloco de código que será executado se – e somente se – a condição do parênteses for `FALSE`.

O if() com o else: exemplo

Como no caso anterior, vejamos primeiramente um exemplo bastante simples.

```
numero <- 1

if (numero == 1) {
  cat("o numero é igual a 1")
} else {
  cat("o numero não é igual a 1")
}
## o numero é igual a 1
```

O `if()` com o `else`: encadeando

É possível encadear diversos `if() else` em sequência:

```
numero <- 10

if (numero == 1) {
  cat("o numero é igual a 1")
} else if (numero == 2) {
  cat("o numero é igual a 2")
} else {
  cat("o numero não é igual nem a 1 nem a 2")
}
## o numero não é igual nem a 1 nem a 2
```

Exemplo: par ou ímpar?

Vamos criar uma função que nos diga se um número é par ou ímpar. Nela vamos utilizar tanto o `if()` sozinho quanto o `if() else`. Vale relembrar que um número (inteiro) é par se for divisível por 2 e que podemos verificar isso se o resto da divisão (operador %% no R) deste número por 2 for igual a zero.

Exemplo: par ou ímpar?

```
par_ou_impar <- function(x){
  # verifica se o número é um decimal comparando o tamanho da diferença de x e fl
  # se for decimal retorna NA (pois par e ímpar não fazem sentido para decimais)
 if (abs(x - round(x)) > 1e-7) {
    return(NA)
  }

  # se o número for divisível por 2 (resto da divisão zero) retorna "par"
  # caso contrário, retorna "ímpar"
  if (x %% 2 == 0) {
    return("par")
  } else {
    return("impar")
  }
}
```

Exemplo: par ou ímpar?

Testando:

```
par_ou_impar(4)
## [1] "par"

par_ou_impar(5)
## [1] "impar"

par_ou_impar(2.1)
## [1] NA
```

Parece que está funcionando bem, mas...

Lembre-se: `if()` não é vetorizado.

Há um pequeno problema:

```
x <- 1:5
par_ou_impar(x)
## Warning in if (abs(x - round(x)) > 1e-07) {: a condição tem comprimento > 1 e
## somente o primeiro elemento será usado
## Warning in if (x%%2 == 0) {: a condição tem comprimento > 1 e somente o primei
## elemento será usado
## [1] "impar"
```

Provavelmente não era isso o que esperávamos. O que está ocorrendo aqui?

Os comandos `if()` e `if() else` não são vetorizados.

O `if()` aceita apenas um único valor, seja `TRUE` ou `FALSE`. Se você passar mais de um valor, ele ignora os demais e usa apenas o primeiro.

Lembre-se: `if()` não é vetorizado.

Revendo a questão em um exemplo mais simples, note que o `if()` irá usar apenas o primeiro valor do vetor `c(F, T)`:

```
if (c(F, T)) {

  print("TRUE")

} else {

  "FALSE"
}
## Warning in if (c(F, T)) {: a condição tem comprimento > 1 e somente o pr
## elemento será usado
## [1] "FALSE"
```

A função `ifelse()`

Uma alternativa para casos como esses é utilizar a função `ifelse()`.
Estrutura básica:

```
ifelse(vetor_de_condicoes, valor_se_TRUE, valor_se_FALSE)
```

- o primeiro argumento é um vetor (ou uma expressão que retorna um vetor) com vários `TRUE` e `FALSE`;
- o segundo argumento é o valor que será retornado quando o elemento do `vetor_de_condicoes` for `TRUE`;
- o terceiro argumento é o valor que será retornado quando o elemento do `vetor_de_condicoes` for `FALSE`.

A função `ifelse()`: exemplo

Primeiramente, vejamos um caso trivial:

```
ifelse(c(TRUE, FALSE, FALSE, TRUE), 1, -1)
## [1]  1 -1 -1  1
```

Note que passamos um vetor de condições com `TRUE`, `FALSE`, `FALSE` e `TRUE`.
O valor para o caso `TRUE` é `1` e o valor para o caso `FALSE` é `-1`.
Logo, o resultado é `1`, `-1`, `-1` e `1`.

A função `ifelse()`: voltando ao par ou ímpar

Vamos criar uma versão com `ifelse` da nossa função que nos diz se um número é par ou ímpar.

```
par_ou_impar_ifelse <- function(x){

  # se x for decimal, retorna NA, se não for, retorna ele mesmo (x)
  x <- ifelse(abs(x - round(x)) > 1e-7, NA, x)

  # se x for divisivel por 2, retorna 'par', se não for, retorna impar
  ifelse(x %% 2 == 0, "par", "impar")
}
```

A função `ifelse()`: voltando ao par ou ímpar

Perceba que agora a função funciona sem problemas com vetores:

```
par_ou_impar_ifelse(x)
## [1] "impar" "par"   "impar" "par"   "impar"

par_ou_impar_ifelse(c(x, 1.1))
## [1] "impar" "par"   "impar" "par"   "impar" NA
```

Vetorização!

Um tema constante neste curso é fazer com que você pense sempre em explorar a vetorização do R. Este caso não é diferente, poderíamos ter feito a função utilizando apenas comparações vetorizadas!

Vetorização!

```
par_ou_impar_vec <- function(x){
  # transforma decimais em NA
  decimais <- abs(x - round(x)) > 1e-7
  x[decimais] <- NA

  # Cria vetor para aramazenar resultados
  res <- character(length(x))

  # verifica quem é divisível por dois
  ind <- (x %% 2) == 0

  # quem for divisível por dois é par, quem não for é ímpar
  res[ind] <- "par"
  res[!ind] <- "impar"

  return(res)
}
```

Vetorização!

Na prática, o que a função `ifelse()` faz é mais ou menos isso o que fizemos – comparações e substituições de forma vetorizada. Note que, neste caso, nossa implementação ficou inclusive um pouco mais rápida do que a solução anterior com `ifelse()`:

```
library(microbenchmark)
microbenchmark(par_ou_impar_vec(1:1e3), par_ou_impar_ifelse(1:1e3))
## Unit: microseconds
##                          expr min  lq mean median  uq  max neval cld
##     par_ou_impar_vec(1:1000)  55  56   82     58  85  995   100  a
##  par_ou_impar_ifelse(1:1000) 320 323  458    338 486 2036   100   b
```

Exercícios

Sua vez.

- Crie, usando `if() else` uma função que verifica se `x` é maior do que 1. Se for, retorna o valor `x^2`. Se não for, verifica se `x` é menor do que -1. Se for, retorna `-x^2`. Se não for nenhum dos casos, retorna o próprio `x`. Sua função é vetorizada?
- Cria a mesma função usando `ifelse()`.
- Crie a função sem usar nem `if() else` nem `ifelse()`.

Soluções

```r
# uma forma diferente..
funcao_if <- function(x){
  if (x > 1) return(x^2)
  if (x < -1) return(-x^2)
  return(x)
}

funcao_ifelse <- function(x) ifelse(x > 1, x^2, ifelse(x < -1, -x^2, x))

funcao <- function(x){
  x[x > 1] <- x[x > 1]^2
  x[x < -1] <- -x[x < -1]^2
  return(x)
}
```

Executando código diversas vezes (loops): `for()` e `while()`

Loops com `for()`

Um loop utilizando `for()` no R tem a seguinte estrutura básica:

```
for(i in conjunto_de_valores){
  # comandos que
  # serão repetidos
}
```

- O início do loop se dá com o comando `for` seguido de parênteses e chaves;
- Dentro do parênteses temos um indicador que será usado durante o loop (no caso escolhemos o nome `i`) e um conjunto de valores que será iterado (`conjunto_de_valores`).
- Dentro das chaves temos o bloco de código que será executado durante o loop.

Em outras palavras, no comando acima estamos dizendo que para cada elemento `i` contido no `conjunto_de_valores` iremos executar os comandos que estão dentro das chaves.

Loops com for(): exemplo

Vamos imprimir na tela os números de 1 a 5.

```
for (i in 1:5) {
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Loops com for(): exemplo

Agora, vamos imprimir na tela as 5 primeiras letras do alfabeto (o R já vem com um vetor com as letras do alfabeto: `letters`).

```
for (i in 1:5) {
  print(letters[i])
}
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Loops com `for()`: exemplo

No mesmo exemplo, ao invés correr o loop no índice de inteiros `1:5`, vamos iterar diretamente sobre os primeiros 5 elementos do vetor `letters`:

```
for (letra in letters[1:5]) {
  print(letra)
}
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

Loops com for(): seq_along() e length()

Uma função bastante útil ao fazer loops é a função seg_along(). Ela cria um vetor de inteiros com índices para acompanhar o objeto.

```
# criando um vetor de exemplo
set.seed(119)
x <- rnorm(10)

# inteiros de 1 a 10
seq_along(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Loops com `for()`: `seq_along()` e `length()`

Também é possível criar um vetor de inteiros do tamanho do objeto fazendo uma sequência de 1 até `length(x)`:

```
1:length(x)
##  [1]  1  2  3  4  5  6  7  8  9 10
```

Entretanto, a vantagem de `seq_along()` é que quando o vetor é vazio, ela retorna um vetor vazio e, deste modo, o loop não é executado (o que é o comportamento correto). Já a sequência `1:length(x)` retorna a sequência `1:0`, isto é, uma sequência decrescente de 1 até 0, e loop é executado nestes valores. Vejamos.

Loops com for(): seq_along() e length()

```
# cria vetor vazio
x <- numeric(0)

# 1:length(x)
# note que o loop é executado (o que é errado)
for (i in 1:length(x)) print(i)
## [1] 1
## [1] 0

# seq_along
# note que o loop não é executado (o que é correto)
for (i in seq_along(x)) print(i)
```

Vetorização, funções nativas e loops

- Como vimos, o R é vetorizado. Muitas vezes, quando você pensar que precisa usar um loop, ao pensar melhor, descobrirá que não precisa. Em geral é possível resolver o problema de maneira vetorizada e usando funções nativas do R.
- Para quem está aprendendo a programar diretamente com o R, isso é algo que virá naturalmente. Todavia, para quem já sabia programar em outras linguagens de programação – como C – pode ser difícil se acostumar a pensar desta maneira.

Vetorização, funções nativas e loops

Vejamos um exemplo trivial. Suponha que você queira dividir os valores de um vetor `x` por 10. Se o R não fosse vetorizado, você teria que fazer algo como:

```
# criando vetor de exemplo
x <- 10:20

# divide cada elemento por 10
for (i in seq_along(x))
  x[i] <- x[i]/10

# resultado
x
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Vetorização, funções nativas e loops

Mas o R **é** vetorizado e, portanto, este é o tipo de loop **que não faz sentido** na linguagem. É muito mais rápido e fácil de enteder escrever simplesmente `x/10`, como já tínhamos aprendido nas primeiras aulas!

```
# recriando vetor de exemplo
x <- 10:20

# divide cada elemento por 10
x <- x/10
x
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Vetorização, funções nativas e loops

Vejamos um caso um pouco mais complicado. Suponha que você queira, gerar um passeio aleatório com um algoritmo simples: a cada período você pode andar para frente (+1) ou para trás (-1) com probabilidades iguais.

```
set.seed(1)

# número de passos
n <- 1000
# vetor para armazenar o passeio aleatório
passeio <- numeric(n)
# primeiro passo
passeio[1] <- sample(c(-1, 1), 1)
# demais passos
for (i in 2:n) {
  # passo i é o onde você estava (passeio[i-1])   mais o passo seguinte
  passeio[i] <- passeio[i - 1] + sample(c(-1, 1), 1)
}
```

Vetorização, funções nativas e loops

É possível fazer tudo isso com apenas uma linha de maneira "vetorizada" e bem mais eficiente: crie todos os `n` passos de uma vez e faça a soma acumulada.

```
set.seed(1)
passeio2 <- cumsum(sample(c(-1, 1), n, TRUE))

# verifica se são iguais
all.equal(passeio, passeio2)
## [1] TRUE
```

Vetorização, funções nativas e loops

Então, você deve estar se perguntando: "não é para usar loops nunca"?

Não é isso. Em algumas situações loops são inevitáveis e podem inclusive ser mais fáceis de ler e de entender. O ponto aqui é apenas lembrá-lo de explorar a vetorização do R.

DICA: pré-alocar espaço antes do loop

Um erro bastante comum de quem está começando a programar em R é "crescer" objetos durante o loop. Isto tem um impacto substancial na performance do seu programa!

- Sempre que possível, crie um objeto, antes de iniciar o loop, para armazenar os resultados de cada iteração.

Vamos calcular os `n` primeiros números da sequência de Fibonacci:
$F_1 = 0, F_2 = 1, F_3 = 1, F_4 = 2, F_5 = 3, F_6 = 5, F_7 = 8, F_8 = 13, F_9 = 21$... Note que a sequência de Fibonacci pode ser definida da seguinte forma:

- os primeiros dois números são 0 e 1, isto é, $F_1 = 0, F_2 = 1$;
- A partir daí, os números subsequentes são a soma dos dois números anteriores, isto é, $F_i = F_{i-1} + F_{i-2}$ para todo $i > 2$.

Vejamos uma forma de implementar isto no R usando `for()` e criando um vetor para armazenar os resultados.

DICA: pré-alocar espaço antes do loop

```
n <- 9
# crie um vetor de tamanho n
# para armazenar os n resultados
fib <- numeric(n)
# comece definindo as condições iniciais
# F1 = 0 e F2 = 1
fib[1] <- 0
fib[2] <- 1
# Agora para todo i > 2
# calculamos Fi = F(i-1) + F(i - 2)
for (i in 3:n) {
  fib[i] <- fib[i - 1] + fib[i - 2]
}
# conferindo resultado
fib
## [1]  0  1  1  2  3  5  8 13 21
```

Loops com while()

Estrutura básica:

```
while (condicao) {
  # codigo a ser executado
  # até que condição seja TRUE
  # em geral a condição será atualizada
  # dentro do código
}
```

- O início do loop se dá com o comando `while` seguido de parênteses e chaves;
- Dentro do parênteses temos uma condição lógica que será testada;
- Enquanto a condição lógica for verdadeira, bloco de código que está entre chaves será executado repetidamente.

Loops com `while()`: exemplo

Lembra que com o for contamos de 1 até 5? Como fazer a mesma coisa com o while?

```
# i inicial
i <- 1
# condição:
# enquanto i for menor ou igual a 5
while (i <= 5) {
  print(i)
  # atualiza i (cuidado com loop infinito!)
  i <- i + 1
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

loop infinito

Se você entrar em um loop infinito, aperte `esc` para sair. Vamos testar: o código abaixo irá iniciar um loop infinito incrementando a variável `i`. Deixe o código rodando por um tempo e, depois, aperte `esc` para interromper.

```
i <- 1
while (i>=1) {
  print(i)
  i <- i  + 1
}
```

- Por que o código acima está em loop infinito?

Qual a diferença? `while()` vs `for()`

Usar o `while()` para contar de 1 até 5 é bem mais complicado do que com o `for()`, então qual a vantagem do `while()`? O `while()` é fundamental quando precisamos rodar uma função repetidas vezes **mas não sabemos quantas** vezes!

```
set.seed(1)
x <- rnorm(1)
i <- 1

while (x < 2) {
  i <- i + 1
  x <- rnorm(1)
}

# quantas vezes rodou?
i
## [1] 61
```

Mais controle sobre loops: `break()`

A função `break()` interrompe a execução de um loop no momento em que é chamada.

- exemplo: quero que o loop rode de 1 até 10, mas se por acaso a variável `u` for maior do que `0.8`, o loop é interrompido.

```
set.seed(25)
for (i in 1:10) {
  u <- runif(1)
  if (u > 0.8) break()
  print(i)
}
## [1] 1
## [1] 2
## [1] 3
```

Mais controle sobre loops: next() e break()

A função next() faz com que o loop passe para a próxima iteração no momento em que é chamada.

- exemplo: quero que o loop rode de 1 até 10, mas se por acaso a variável u for maior do que 0.5, eu pulo aquela parte do loop.

```
set.seed(25)
for (i in 1:10) {
  u <- runif(1)
  if (u > 0.5) next()
  print(i)
}
## [1] 1
## [1] 3
## [1] 5
## [1] 8
## [1] 9
## [1] 10
```

Outra forma de fazer while(): repeat + break

A função repeat repete o código entre chaves indefinidamente. Deve ser utilizada **sempre** em conjunto com a função break().

```
# contando de 1 até 5
i <- 1
repeat {
  print(i)
  i <- i + 1
  if (i > 5) break()
}
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

Exercícios

Sua vez.
As funções que você irá implementar aqui serão até mais de 100 vezes mais lentas do que as funções nativas do R. Estes exercícios são para você treinar a construção de loops, um pouco de lógica de programação, e entender o que as funções do R estão fazendo – de maneira geral – por debaixo dos panos.

Exercícios

1. Crie uma função que encontre o máximo de um vetor (faça uma vez usando `for()` e outra usando `while()`). Compare os resultados de sua implementação com a função `max()` do R. **Dicas**: você terá que percorrer o vetor e comparar elementos.
2. Crie uma função que calcule o fatorial de `n` (faça uma vez usando `for()` e outra usando `while()`). Compare os resultados de sua implementação com a função `factorial()` do R. **Dica**: para quem não lembra, o fatorial de um número `n` é a multiplicação de todos os número de 1 até `n`. Por exemplo, o fatorial de 6 é $6 * 5 * 4 * 3 * 2 * 1$.

Soluções

```
# cria vetor para comparar resultados
set.seed(123)
x <- rnorm(100)

# 1) loop para encontrar máximo (com for)
max_loop <- function(x){
  max <- x[1]
  for (i in 2:length(x)) {
    if (x[i] > max) {
      max <- x[i]
    }
  }
  return(max)
}
all.equal(max(x), max_loop(x))
## [1] TRUE
```

Respostas

```
# 2) loop para fatorial (com for)
fatorial <- function(n){
  if (n == 0) return(1)
  fat <- 1
  for (i in 1:n) {
    fat <- fat*i
  }
  return(fat)
}

all.equal(factorial(10), fatorial(10))
## [1] TRUE
```

A forma do R de fazer loops: voltando à família apply!

Já vimos várias funções da família apply, como `apply()`, `lapply()` e `sapply()`, quando estudamos matrizes, data.frames e listas. Agora vamos ver novamente essas funções, mas sob outra ótica: as funções da família apply nada mais são do que funções que facilitam sua vida, fazendo loops para você!

Por que a família apply?

Vamos calcular a média de cada uma das colunas do `data.frame` `mtcars` usando loops. Para isso precisamos:

- saber quantas colunas existem no `data.frame`;
- criar um vetor para armazenar os resultados;
- nomear o vetor de resultados com os nomes das colunas; e
- fazer um loop para cada coluna.

Por que a família apply?

```
# (i) quantas colunas no data.frame
n <- ncol(mtcars)

# (ii) vetor para armazenar resultados
medias <- numeric(n)

# (iii) nomeando vetor com nomes das colunas
names(medias) <- colnames(mtcars)

# (iv) loop para cada coluna
for (i in seq_along(mtcars)) medias[i] <- mean(mtcars[,i])

# resultado final
medias
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81
```

Por que a família apply?

Gastamos várias linhas para fazer essa simples operação. Como já vimos, é bastante fácil fazer isso no R com apenas uma linha:

```
sapply(mtcars, mean)
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81
```

- Mas, imagine que não existisse a função `sapply()` no R. Se quiséssemos aplicar outra função para cada coluna, teríamos que copiar e colar todo o código novamente, certo?
- Sim, você poderia fazer isso, mas não seria uma boa prática. Neste caso, como já vimos, o ideal seria criar uma função!

Por que a família apply?

Façamos, portanto, uma função que nos permita aplicar uma fução arbitrária nas colunas de um data.frame.

```
meu_sapply <- function(x, funcao){

  n <- length(x)

  resultado <- numeric(n)

  names(resultado) <- names(x)

  for(i in seq_along(x)){
    resultado[i] <- funcao(x[[i]])
  }

  return(resultado)
}
```

Por que a família apply?

Perceba que ficou bastante simples percorrer todas as colunas de um `data.frame` para aplicar a função que você quiser:

```
meu_sapply(mtcars, mean)
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##  20.09   6.19 230.72 146.69   3.60   3.22  17.85   0.44   0.41   3.69   2.81

meu_sapply(mtcars, sd)
##    mpg    cyl   disp     hp   drat     wt   qsec     vs     am   gear   carb
##   6.03   1.79 123.94  68.56   0.53   0.98   1.79   0.50   0.50   0.74   1.62

meu_sapply(mtcars, max)
##   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
##  33.9   8.0 472.0 335.0   4.9   5.4  22.9   1.0   1.0   5.0   8.0
```

Por que a família apply?

É isso o que as funções da família `apply` são: são funções que fazem loops para você. Elas automaticamente cuidam de toda a parte chata do loop como, por exemplo, criar um objeto de tamanho correto para pré-alocar os resultados. Além disso, em grande parte das vezes essas funções serão mais eficientes do que se você mesmo fizer a implementação.

Principais funções apply

- **Aplicar função nas dimensões:** como vimos, a função `apply()` aplica funções nas linhas, colunas ou outras dimensões de uma matriz, data.frame ou array.
- **Aplicar função nos elementos:** como vimos, para aplicar uma função a cada elemento de um objeto, podemos usar `lapply()`, `sapply()` ou `vapply()`. A diferença entre elas é o formato do resultado.
  1. A função **l**apply retorna uma **l**ista;
  2. A função **s**apply tenta **s**implificar o resultado para um objeto mais simples (como um vetor ou matriz); e,
  3. A função **v**apply espera como resultado um formato de **v**alor específico (caso contrário, retorna erro).
- **Aplicar funções em múltiplos elementos**: a função `mapply()` pode ser consideradas uma versão multivariada do `sapply()`. O `mapply()` aplica uma função em todos elementos de **m**últiplos objetos ao mesmo tempo.
- **Repetir código em simulações de Mote Carlo**: para isso temos a função `replicate()`, que replica uma expressão diversas vezes.

Aplicando funções em dimensões: `apply()`

```
set.seed(1)
x <- matrix(rnorm(9), ncol = 3)

apply(x, 1, mean)
## [1]  0.49  0.42 -0.36

apply(x, 2, mean)
## [1] -0.43  0.37  0.60
```

Aplicando funções em elementos: `lapply()` - retorna lista

```
lista_de_matrizes <- list(x = x, tx = t(x))

# invertendo todas ao mesmo tempo
lapply(lista_de_matrizes, solve)
## $x
##        [,1]   [,2]  [,3]
## [1,] -0.500  0.829 -0.64
## [2,]  0.454 -0.029 -0.35
## [3,] -0.078  1.161  0.31
##
## $tx
##       [,1]   [,2]   [,3]
## [1,] -0.50  0.454 -0.078
## [2,]  0.83 -0.029  1.161
## [3,] -0.64 -0.347  0.314
```

Aplicando funções em elementos: sapply() - simplifica resultado

Calculando determinantes de todas as matrizes:

```
lapply(lista_de_matrizes, det)
## $x
## [1] -1.6
##
## $tx
## [1] -1.6
```

Com sapply() resultado já vem como vetor:

```
sapply(lista_de_matrizes, det)
##    x   tx
## -1.6 -1.6
```

Aplicando funções em elementos: vapply() - checa resultado

```
ok  <- list(x = matrix(1:10, ncol = 2),
            y = matrix(11:20, ncol = 2))

nao_ok <- list(x = matrix(1:10, ncol = 2),
               y = matrix(11:20, ncol = 5))

vapply(ok, colMeans, numeric(2))
##      x  y
## [1,] 3 13
## [2,] 8 18

vapply(nao_ok, colMeans, numeric(2))
## Error in vapply(nao_ok, colMeans, numeric(2)): valores devem ser de comprimento
##  mas o resultado de FUN(X[[2]]) tem comprimento 5
```

Quando usar cada um: `lapply()`, `sapply()` vs `vapply()`

- O `sapply()` é para uso interativo. Quando você está explorando uma base de dados, o `sapply()` facilita seu trabalho tentando simplificar o resultado da operação. Entretanto, o `sapply()` não te dá sempre o mesmo resultado (às vezes pode ser um vetor, às vezes uma lista), e isso pode ser perigoso para usar em funções.
- A função `lapply()` é mais previsível que o `sapply()`: ela sempre vai te retornar uma lista. Neste caso, você vai ter o trabalho de simplificar o resultado manualmente, mas não terá a surpresa de vir um resultado em um formato diferente do que você esperava.
- Por fim o `vapply()` é para quando você quer ser bastante restrito no tipo de resultado que você deseja obter para evitar bugs. Por exemplo: o resultado esperado sempre tem que ser um vetor numérico de tamanho 2 – e se vier qualquer valor diferente disso, você quer que a função pare e forneça uma mensagem de erro.

Passando argumentos extras

Se você olhar a definição das funções, verá que o `...` aparece em todas elas. Por quê?

- `apply(X, MARGIN, FUN, ...)`
- `lapply(X, FUN, ...)`
- `sapply(X, FUN, ..., simplify = TRUE, USE.NAMES = TRUE)`
- `vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)`

Olhe na ajuda:

- ***... optional arguments to FUN***.

Isto é, como vimos na aula de funções, os `...` servem justamente para passar argumentos arbitrários para função `FUN` que está sendo aplicada!

Simulações de Monte Carlo: replicate()

O replicate() é uma função de conveniência para repetir a execução de uma expressão diversas vezes no R. Sua estrutura básica é a seguinte:

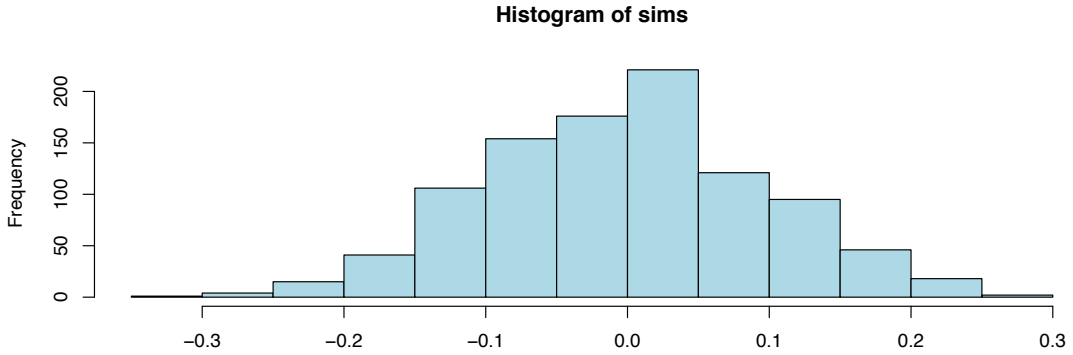
```
replicate(numero_de_repeticoes, expressao)
```

A função replicate() é bastante utilizada para simulações de Monte Carlo.

Simulações de Monte Carlo: `replicate()`

Exemplo: distribuição de média amostral.

```
sims <- replicate(1000, mean(rnorm(100)))
hist(sims, col = "lightblue")
```



Aplicando funções em múltiplos argumentos: `mapply()`

A função `mapply()` pode ser vista como um `sapply()` "multivariado". O comando:

```
mapply(funcao, x, y, z)
```

É equivalente a:

```
funcao(x[1], y[1], z[1])
funcao(x[2], y[2], z[2])
funcao(x[3], y[3], z[3])
funcao(x[4], y[4], z[4])
...
funcao(x[n], y[n], z[n])
funcao(x[n], y[n], z[n])
```

Aplicando funções em múltiplos argumentos: `mapply()`

Exemplo: quero a média aparada de `x1` e de `x2`, mas quero que em `x1` `trim = 0.1` e em `x2` `trim = 0.2`. Fazendo com `mapply()`:

```
set.seed(112)
x1 <- rnorm(100)
x2 <- rnorm(100)
mapply(mean, x = list(x1, x2), trim = list(0.1, 0.2))
## [1]  0.13 -0.03
```

O que seria equivalente a:

```
mean(x1, trim = 0.1)
## [1] 0.13

mean(x2, trim = 0.2)
## [1] -0.03
```

Tabela resumo

Função	Descrição
`apply()`	Aplica função nas dimensões (linha, coluna, etc...) do objeto.
`lapply()`	Aplica função em todos elementos do objeto. Retorna uma lista.
`sapply()`	Similar a `lapply` mas tenta simplificar resultado.
`vapply()`	Similar a `sapply` mas checa formato do resultado.
`mapply()`	Versão multivariada do `sapply()`. Aplica função a todos elementos de vários objetos.
`replicate()`	Replica expressão um número pré-estabelecido de vezes.

Tem mais...

Veremos outras opções quando nos aprofundarmos em manipulações de `data.frames`!

Programação em R

Julho, 2016

O que veremos

Vamos dar um foco adicional na manipulação de data frames, pois esta é uma atividade que será bastante recorrente. Em geral, você gasta a maior parte do seu tempo arrumando, explorando e colocando os dados em uma forma adequada para análise. Veremos:

- Uma breve revisão e algumas novas funções (adicionar, remover colunas, subset etc);
- A estratégia Dividir, Aplicar e Combinar com funções nativas do R e `dplyr`;
- Como colocar seus dados no formato ideal com o pacotes `reshape2` e `tidyr`

Manipulação de data.frames: revisão e algumas funções

Carregando a base de dados

Iremos trabalhar com dados de oferta online de imóveis no plano piloto no mês de agosto de 2014.

```
arquivo <- url("https://dl.dropboxusercontent.com/u/44201187/dados.rds")
con <- gzcon(arquivo)
imoveis <- readRDS(con)
close(con)
```

Carregando a base de dados

Vendo a estrutura da base:

```
str(imoveis, vec.len = 1)
## 'data.frame':    366666 obs. of  13 variables:
##  $ bairro   : chr  "Park Sul" ...
##  $ location : chr  "SMAS APARTAMENTO SOMENTE PARA DIÁRIA/TEMPORADA !!!" ...
##  $ preco    : num  400 600 ...
##  $ quartos  : num  3 1 ...
##  $ m2       : num  95 30 ...
##  $ corretora: chr  "21900" ...
##  $ data     : Date, format: "2014-08-01" ...
##  $ link     : chr  "/imovel/aluguel-apartamento-brasilia-df-3-quartos-smas-1223980" ...
##  $ estado   : chr  "df" ...
##  $ cidade   : chr  "brasilia" ...
##  $ imovel   : chr  "apartamento" ...
##  $ tipo     : chr  "aluguel" ...
##  $ coleta   : Date, format: "2014-08-01" ...
```

Renomeando Colunas

Note que a segunda coluna, que contém o endereço do apartamento, está com o nome `location`. Vamos renomeá-la para `endereco`.

```
names(imoveis)[2]
## [1] "location"
names(imoveis)[2] <- "endereco"
names(imoveis)[2]
## [1] "endereco"
```

Adicionando colunas

Note que não há uma coluna com preço por metro quadrado. Vamos adicionar esta coluna:

```
imoveis$pm2 <- imoveis$preco/imoveis$m2
```

Uma forma conveniente de realizar isto é com a função `with`:

```
imoveis$pm2 <- with(imoveis, preco/m2)
str(imoveis$pm2)
##  num [1:366666] 4.21 20 20 21.67 28.38 ...
```

Head e Tail

A base de dados é relativamente grande (mais de 300 mil obs) então é preciso tomar cuidado para não ficar imprimindo os valores na tela. Para "espionar" os dados, duas funções convenientes são `head` e `tail`. Os resultados são omitidos, pois são muito grandes para caber no slide.

```
# head - primeiras observações
head(imoveis) # mostra 6 primeiras linhas
imoveis[1:6,] # equivalente
head(imoveis, 10) # mostra 10 primeiras linhas
imoveis[1:10,] # equivalente

# tail - últimas observações
tail(imoveis) # mostra as 6 últimas linhas
imoveis[(nrow(imoveis)-5):nrow(imoveis),] # equivalente
tail(imoveis, 10) # mostra as 10 últimas linhas
imoveis[(nrow(imoveis)-10):nrow(imoveis),] # equivalente
```

Subset

Vamos revisar brevemente algumas formas de selecionar linhas e colunas do data.frame:

```
# com números
imoveis[1,] #primeira linha, todas as colunas
imoveis[,1] #primeira coluna, todas as linhas
imoveis[1,1] #primeira linha e primeira coluna
imoveis[c(2,4,6), c(1,3:5)] #linhas 2, 4 e 6, e colunas 1,3,4,5.
imoveis[1:10,-(1:5)] #linhas 1:10, exclui colunas 1 a 5
head(imoveis[-c(1,2,5),]) # exclui linhas 1, 2 e 5

# com nomes
imoveis[imoveis$bairro == "Asa Sul", c("data", "bairro", "preco")]
subset(imoveis, bairro == "Asa Sul", select=c(data, bairro, preco))
```

Subset

E comparar as diferentes formas de selecionar uma coluna:

```
precos1 <- imoveis$preco # vetor
precos2 <- imoveis[ ,"preco"]  # vetor
precos3 <- imoveis[["preco"]]  # vetor
precos4 <- imoveis["preco"]  # data.frame
precos5 <- subset(imoveis, select=preco)  # data.frame
precos6 <- imoveis[,"preco", drop=FALSE] # data.frame

str(precos1)
##  num [1:366666] 400 600 600 650 650 680 700 700 700 700 ...
str(precos4)
## 'data.frame':    366666 obs. of  1 variable:
##  $ preco: num  400 600 600 650 650 680 700 700 700 700 ...
rm(list=ls(pattern="precos"))
```

Filtrando com índices lógicos

Vamos passar alguns filtros no data.frame. Como selecionar os imóveis que estavam à venda no dia 31 de Agosto que foram anunciados a mais de 1 milhão de reais?

```
indice <- with(imoveis, coleta=="2014-08-31" &
                        tipo=="venda" &
                        preco > 1e6)
sum(indice) # quantos imóveis?
## [1] 3826
dia31_venda_maior1m <- imoveis[indice,] # filtrando
```

Função subset

A mesma operação com `subset`:

```
dia31_venda_maior1m <- subset(imoveis,
                              coleta == "2014-08-31" &
                              tipo == "venda" &
                              preco > 1e6)
```

Criando subgrupos com cut

Note que a variável m2 é uma variável numérica contínua. Mas podemos querer analisar esta variável em categorias discretas, como até 50m2, de 50m2 a 100m2. A função `cut` serve para isso.

```
imoveis$m2_cat <-
  cut(imoveis$m2,
      breaks= c(0, 50, 150, 200, 250, Inf),
      labels=c("de 0 a 50 m2", "de 50 a 150 m2",
               "de 150 a 200 m2","de 200 a 250 m2",
               "mais de 250 m2") )

str(imoveis$m2_cat)
##  Factor w/ 5 levels "de 0 a 50 m2",..: 2 1 1 1 1 1 1 1 1 1 ...
```

Unique

A função unique retorna as observações únicas de um vetor. Vejamos com mais detalhes quais são os valores de algumas colunas de nossa base de dados.

```
unique(imoveis$tipo)
## [1] "aluguel" "venda"
unique(imoveis$imovel)
## [1] "apartamento"    "casa"           "kitinete"       "loja"
## [5] "sala-comercial"
unique(imoveis$bairro)
##  [1] "Park Sul"                          "Asa Norte"
##  [3] "Lago Norte"                        "Lago Sul"
##  [5] "Vila Planalto"                     "Sudoeste"
##  [7] "Asa Sul"                           "Granja do Torto"
##  [9] "Noroeste"                          "Octogonal"
## [11] "Setor Habitacional Jardim Botânico" "Vila Telebrasília"
## [13] "Brasília"                          "Park Way"
## [15] "Taquari"
```

Unique

Quantas obserções únicas de imóveis temos neste 1 mês de coleta? Considere o **link** do anúncio como seu identificador único.

```
unique(imoveis$coleta) # quais os dias de coleta únicos?
##  [1] "2014-08-01" "2014-08-02" "2014-08-03" "2014-08-04" "2014-08-05"
##  [6] "2014-08-06" "2014-08-07" "2014-08-08" "2014-08-09" "2014-08-12"
## [11] "2014-08-13" "2014-08-14" "2014-08-15" "2014-08-16" "2014-08-17"
## [16] "2014-08-18" "2014-08-19" "2014-08-20" "2014-08-21" "2014-08-22"
## [21] "2014-08-23" "2014-08-24" "2014-08-25" "2014-08-26" "2014-08-27"
## [26] "2014-08-28" "2014-08-29" "2014-08-30" "2014-08-31"
length(unique(imoveis$link)) # quantas observações únicas?
## [1] 16635
```

duplicated

Vamos eliminar os duplicados para trabalhar com as observações únicas do mês. A função `duplicated` retorna um vetor lógico com TRUE para uma observação que apareceu anteriormente.

```
duplicados <- duplicated(imoveis$link)
unicos <- imoveis[!duplicados, ]
```

Eliminando NA's

Uma função de conveniência para eliminar NA de um data.frame é a `na.omit`. Mas, **cuidado**, ela irá eliminar todas as observações que cotenham ao menos um NA em alguma coluna.

```
unicos <- na.omit(unicos)
```

Exercícios

Sua vez.
Para cada uma das bases (`imoveis` e `unicos`), quantos registros únicos temos para cada coluna? Para cada uma das bases (`imoveis` e `unicos`), quantos NA's temos para cada coluna?

Soluções

Lembra que o `sapply` aplica uma função a todos os elementos do objeto?

```
sapply(imoveis, function(x) length(unique(x)))
sapply(unicos, function(x) length(unique(x)))
sapply(imoveis, function(x) sum(is.na(x)))
sapply(unicos, function(x) sum(is.na(x)))
```

Dividir, Aplicar e Combinar

Um padrão recorrente

Nossa base de dados contém preços tanto de aluguel quanto de venda de apartamentos. Suponha que queiramos tirar a médias dos preços. Não faz muito sentido tirar a média dos dois tipos (aluguel e venda) juntos, certo? Como poderíamos fazer isso então?
Uma possível solução seria fazer o seguinte:
Primeiramente, dividimos nossa base em duas outras para cada grupo: aluguel e venda.

```
# 1) separar a base em duas bases diferentes:
#    - aluguel; e,
#    - venda
aluguel <- unicos[unicos$tipo == "aluguel", ]
venda <- unicos[unicos$tipo == "venda", ]
```

Um padrão recorrente

Em seguida nós calculamos a média para cada uma das bases que criamos.

```
# 2) calcular a média para cada uma das bases
media_aluguel <- mean(aluguel$preco)
media_venda   <- mean(venda$preco)
```

Por fim, nós combinamos os resultados em um único vetor:

```
# 3) combinar os resultados em um único vetor
medias = c(aluguel = media_aluguel, venda = media_venda)
medias
## aluguel   venda
##   29672 1440468
```

Um padrão recorrente

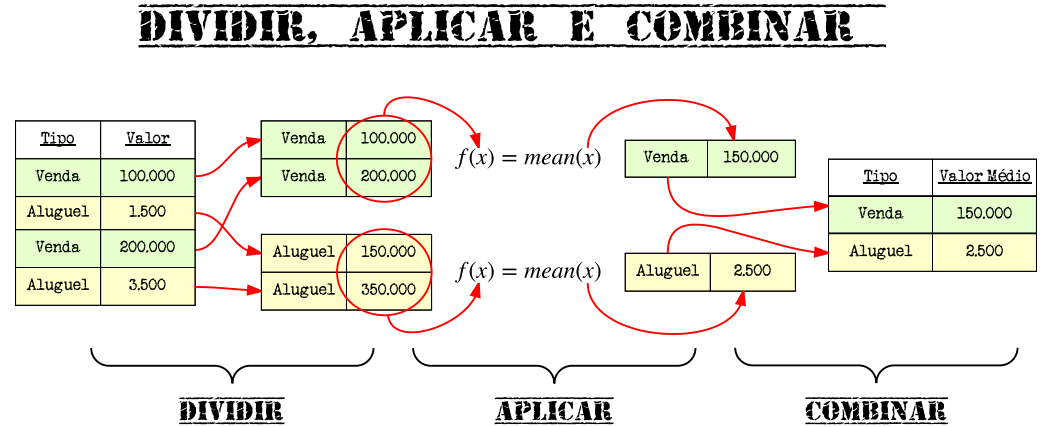
Nós gastamos cerca de 5 linhas para chegar ao resultado. E se eu te dissesse que podemos fazer isso (ou coisas mais complexas) em apenas uma ou duas linhas? Para você saber onde vamos chegar, seguem alguns exemplos:

```
# com tapply
tapply(X = unicos$preco, INDEX = unicos$tipo, FUN = mean)

# com aggregate
aggregate(x = list(media = unicos$preco), by = list(tipo = unicos$tipo),
          FUN = mean)

# com dplyr
library(dplyr)
unicos %>% group_by(tipo) %>% summarise(media = mean(preco))
```

Dividir, Aplicar e Combinar (Split, Apply and Combine)

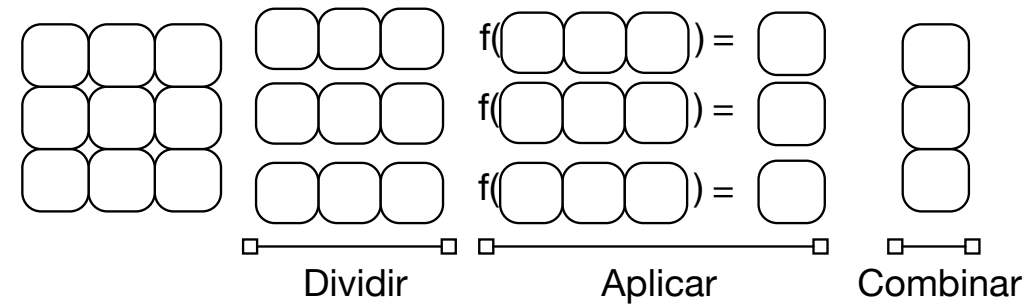
O padrão de análise que fizemos anteriormente é bastante recorrente em análise de dados. Dentro da comunidade do R é conhecido como "Dividir, Aplicar e Combinar" (DAC) ou, em inglês, "Split, Apply and Combine" (SAC). Neste caso específico, nós pegamos um vetor (o vetor de preços), dividimos segundo algum critério (por tipo), aplicamos um função em cada um dos grupos separadamente (no nosso caso, a média) e depois combinamos os resultados novamente. Para quem conhece SQL, muitas dessas operações são similares ao `group by`, ou, para quem usa Excel, similar a uma tabela dinâmica (mas não equivalentes, o conceito aqui é mais flexível).

Dividir, Aplicar e Combinar (Split, Apply and Combine)



Dividir, Aplicar e Combinar (Split, Apply and Combine)

Nós já vimos este padrão diversas vezes com as funções do tipo apply. Por exemplo, ao aplicar uma função por linhas, você está *dividindo* a matriz por uma das dimensões, *aplicando* funções a cada uma das partes e *combinando* os resultados em um vetor:



Vejamos agora algumas peças para construir essa estratégia de análise de dados usando as funções base.

Dividir: a função split

```
str(split)
## function (x, f, drop = FALSE, ...)
```

Dividir: a função split

```
str(split)
## function (x, f, drop = FALSE, ...)
```

- **x**: é o vetor ou data.frame que será divido;
- **f**: são os fatores que irão definir os grupos de divisão.

Dividir: a função split

```
str(split)
## function (x, f, drop = FALSE, ...)
```

- **x**: é o vetor ou data.frame que será divido;
- **f**: são os fatores que irão definir os grupos de divisão.

O resultado da função é uma lista com os vetores ou data.frames dos grupos.

Dividir: a função split

No nosso exemplo:

```
# 1) Dividir o data.frame segundo uma lista de fatores
alug_venda <- split(unicos$preco, unicos$tipo)

# Note o resultado
# temos na lista alug_venda dois vetores
# um para aluguel
# e outro para venda
str(alug_venda, max.level = 1)
## List of 2
##  $ aluguel: num [1:5246] 400 600 600 650 650 680 700 700 700 700 ...
##  $ venda  : num [1:11383] 750 845 159000 170000 175000 180000 180000 18
```

Aplicar e combinar - voltando à família apply

Como visto, o resultado de split é uma lista para cada categoria. Agora queremos aplicar uma função a cada um dos elementos dessa lista. Já vimos uma função que faz isso: `lapply()`:

```
medias <- lapply(alug_venda, mean)
medias
## $aluguel
## [1] 29672
##
## $venda
## [1] 1440468
```

Aplicar e combinar - voltando à família apply

Mas o lapply nos dá uma lista e queremos um vetor. Então agora queremos simplificar o resultado. Uma das formas seria utilizar uma função que vocês já aprenderam: `unlist()`

```
unlist(medias)
## aluguel   venda
##   29672 1440468
```

Aplicar e combinar - voltando à família apply

Existe uma função de conveniência que, em conjunto com `rbind()` e `cbind()` pode ser bastante útil para simplificar resultados: a função `do.call()`. Como ela funciona?

```
do.call("alguma_funcao", lista_de_argumentos) =
  alguma_funcao(lista_de_argumentos[1],
                lista_de_argumentos[2],
                  ...,
                lista_de_argumentos[n])
```

Aplicar e combinar - voltando à família apply

No nosso caso temos apenas duas médias, então não seria complicado elencar um por um os elementos no `rbind()` ou no `cbind()`.
Todavia, imagine que tívessemos centenas de médias. Neste casso a função `do.call()` é bastante conveniente.

```
do.call("rbind", medias)
##             [,1]
## aluguel   29672
## venda   1440468

do.call("cbind", medias)
##      aluguel   venda
## [1,]   29672 1440468
```

Aplicando e simplificando ao mesmo tempo

Agora podemos encaixar conceitualmente outra função que já tínhamos aprendido, o `sapply()`. Essa função tenta fazer os dois passos ao mesmo tempo: aplicar e combinar (que neste caso é *simplificar*):

```
sapply(alug_venda, mean)
## aluguel   venda
##   29672 1440468
```

Aplicando e simplificando ao mesmo tempo

Como vimos, existe, ainda, uma versão mais restrita do `sapply()`: o `vapply()`. Lembrando que a principal diferença entre eles é que o `vapply()` exige que você especifique o formato do resultado esperado da operação. Enquanto o o primeiro é mais prático para uso interativo, o segundo é mais seguro para programar suas próprias funções, pois se o resultado não vier conforme esperado ele irá acusar o erro.

```
vapply(alug_venda, mean, numeric(1))
## aluguel   venda
##   29672 1440468

vapply(alug_venda, mean, character(1))
## Error in vapply(alug_venda, mean, character(1)): valores devem ser do tipo 'character',
##  mas o resultado de FUN(X[[1]]) é de tipo 'double'
```

Antes de prosseguir... outliers

Aplicando a função `summary()` para cada elemento de alug_venda:

```
# Aplicando a função summary para cada elemento de alug_venda
lapply(alug_venda, summary)
## $aluguel
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##         0      1100      2000     29700      3700 120000000
##
## $venda
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
##         0    450000    850000   1440000   1450000 995000000
```

Antes de prosseguir... outliers

Note a clara presença de **outliers**, possivelmente dados errados. Então, antes de prosseguir, façamos o seguinte: vamos "limpar" o data.frame `unicos` retirando os valores muito discrepantes de preço por metro quadrado (Para venda, valores abaixo de 3000 ou acima de 20000. Para aluguel, valores abaixo de 25 ou acima de 60). Depois vamos salvar o resultado em um data.frame chamado `limpos`.

Antes de prosseguir... outliers

```
# filtro para venda
ok.venda <- with(unicos, tipo == "venda" &
                         pm2 > 3000 &
                         pm2 < 20000)

# filtro para aluguel
ok.aluguel <- with(unicos, tipo == "aluguel" &
                           pm2 > 25 &
                           pm2 < 100)
# juntando os dois
ok <- (ok.venda | ok.aluguel)

# separando outliers e limpos
outliers <- unicos[!ok,]
limpos <- unicos[ok,]
```

Dividir, Aplicar e Combinar: tapply

A função `tapply()` tem a seguinte estrutura:

```
str(tapply)
## function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- **X**: o objeto que será agregado. *Ex: preços*;
- **INDEX**: uma lista de vetores que servirão de índice para argegar o objeto. *Ex: bairros*;
- **FUN**: a função que será aplicada a X para cada INDEX. *Ex: mediana*.
- **simplify**: tentará simplificar o resultado para uma estrutura mais simples?

Dividir, Aplicar e Combinar: tapply

Vejamos alguns exemplos: calcular a mediana do metro quadrado para aluguel e para venda:

```
tapply(limpos$pm2, limpos$tipo, median)
## aluguel   venda
##      36    8983
```

Agora para aluguel e venda, separado por bairro:

```
tapply(limpos$pm2, list(limpos$bairro, limpos$tipo), median)
##                                   aluguel venda
## Asa Norte                              35  9005
## Asa Sul                                39  9167
## Brasília                               NA 11844
## Granja do Torto                        NA  4138
## Lago Norte                             34  6826
## Lago Sul                               33  5500
## Noroeste                               38  9654
## Octogonal                              29  8652
## Park Sul                               42  9544
```

Dividir, Aplicar e Combinar: tapply

Outro exemplo: qual a mediana do preço por metro quadrado dos apartamentos, separados aluguel, venda e número de quartos?

```
aps <- limpos[limpos$imovel == "apartamento", ]

with(aps, tapply(pm2, list(quartos, tipo), median))
##    aluguel venda
## 1       37  9790
## 2       33  9048
## 3       30  9127
## 4       32 10194
## 5       32 10204
## 6       NA  5481
## 11      NA  3580
```

Dividir, Aplicar e Combinar: aggregate

O `aggregate` é similar ao `tapply` mas, ao invés de retornar um array, retorna um data.frame com uma coluna para cada índice e apenas uma coluna de valor.
A função `aggregate` tem duas sintaxes principais.
A primeira, similar ao `tapply` é:

```
aggregate(dados$valor, by = list(dados$indice1, dados$indice2), funcao)
```

Já a segunda sintaxe utiliza a *formula interface* do R e é do tipo:

```
aggregate(valor ~ indice1 + indice2, dados, funcao)
```

Dividir, Aplicar e Combinar: aggregate

Exemplo: calculando a mediana do preço por metro quadrado, separada por bairro, venda ou aluguel, e tipo de imóvel. Note a diferença do formato deste resultado para o formato do `tapply`.

```
pm2_bairro_tipo_imovel <- aggregate(formula = pm2 ~ bairro + tipo + imovel,
                                    data = limpos,
                                    FUN = median)
head(pm2_bairro_tipo_imovel)
##       bairro    tipo      imovel pm2
## 1  Asa Norte aluguel apartamento  31
## 2    Asa Sul aluguel apartamento  31
## 3 Lago Norte aluguel apartamento  34
## 4   Lago Sul aluguel apartamento  29
## 5   Noroeste aluguel apartamento  37
## 6  Octogonal aluguel apartamento  29
```

Dividir, Aplicar e Combinar: aggregate

Você também pode passar mais de uma variável a ser agregada. Vamos calcular a mediana do preço, metro quadrado e preço por metro quadrado dos valores de aluguel de apartamento. Note que tudo isso pode ser passado diretamente ao aggregate.

```
mediana_aluguel <- aggregate(cbind(preco, m2, pm2) ~ bairro,
                             data = limpos,
                             subset = (tipo == "aluguel" &
                                       imovel == "apartamento"),
                             FUN = median)

mediana_aluguel[order(mediana_aluguel$pm2, decreasing = TRUE), ]
##          bairro preco m2 pm2
## 7      Park Sul  2500 44  41
## 5      Noroeste  2675 75  37
## 3    Lago Norte  1800 55  34
## 8      Sudoeste  2700 80  33
## 1     Asa Norte  2300 70  31
## 2       Asa Sul  2700 76  31
```

dplyr: Eficiente e intuitivo

Com as funções da família `apply` e similares, você consegue fazer praticamente tudo o que você precisa para explorar os dados e deixá-los no(s) formato(s) necessário(s) para análise. E é importante você ser exposto a essas funções para se familiarizar com o ambiente R.
Entretanto, muitas vezes essas funções podem **deixar a desejar em performance** e existe um pacote **bastante rápido** para manipulação de data.frame e com **sintaxe muito intuitiva** chamado `dplyr`. É provável que para o grosso de suas necessidades o `dplyr` seja a solução mais rápida e eficiente.
Vejamos.

dplyr: Funções principais

- **filter**: filtra um data.frame com vetores lógicos. Por exemplo, filtrar valores de pm2 menores ou maiores do que determinado nível.
- **select**: seleciona uma ou mais colunas de um data.frame. Por exemplo, selecionar a coluna de preços.
- **mutate**: cria uma nova coluna. Por exemplo, criar a coluna pm2 como preco/m2.
- **arrange**: orderna o data.frame com base em uma coluna. Por exemplo, ordernar do maior ao menor pm2.
- **group_by**: agrupa um data.frame por índices. Por exemplo, agrupar os dados de imóveis por bairro e número de quartos.
- **summarise**: geralmente utilizado após o group_by. Calcula valores por grupo. Por exemplo, tirar a média ou mediana do preço por bairro.

dplyr: Conectando tudo com %>%

O `dplyr` vem também com o *pipe operator* `%>%` do pacote `magritrr`. Basicamente, o operador faz com que você possa escrever `x %>% f()` ao invés de `f(x)`. Na prática, isso tem uma grande utilidade: você vai poder escrever o código de manipulação dos dados da mesma forma que você pensa nas atividades.

Ex: pegue a base de dados `limpos`, filtre apenas os dados coletados de apartamento, selecione as colunas bairro e preco, crie uma coluna `pm2 = preco/m2`, ordene os dados de forma decrescente em pm2 e mostre apenas as 6 primeiras linhas (head).

```
library(dplyr)
limpos %>% filter(imovel == "apartamento") %>%
  select(bairro, preco, m2) %>% mutate(pm2 = preco/m2) %>%
  arrange(desc(pm2)) %>% head()
```

dplyr: Agrupando e sumarizando

Pegue a base de dados `limpos`, filtre apenas os dados de venda de apartamento. Agrupe os dados por bairro. Calcule as medianas do preço, m2 e pm2 e o número de observações. Filtre apenas os grupos com mais de 30 observações. Ordene de forma decrescente com base na mediana de pm2.

```
limpos %>%
  filter(imovel == "apartamento", tipo == "venda") %>%
  group_by(bairro) %>%
  summarise(Mediana_Preco = median(preco),
            Mediana_M2 = median(m2),
            Mediana_pm2 = median(pm2),
            Obs = length(pm2)) %>%
  filter(Obs > 30) %>%
  arrange(desc(Mediana_pm2))
```

dplyr: Voltando um pouco aos textos

Vamos usar a nossa base de dados para fazer uma busca de apartamentos.
Apartamento na Asa Sul ou Asa Norte, entre 101-404, para aluguel, com preço menor do que R$ 2.000,00.

```
consulta <-
  imoveis %>% filter(grepl("SQ(N|S) (4|3|2|1)0(1|2|3|4)", endereco),
                     tipo == "aluguel",
                     bairro %in% c("Asa Sul", "Asa Norte"),
                     preco < 2200,
                     coleta == "2014-08-31")
```

Melhor do que a busca do *wimoveis*.

Exercícios

Sua vez.
Considerando a base de dados `limpos`, Responda:

- Qual o bairro com o maior preço mediano de venda?
- Qual o bairro com o maior preço por m2 mediano de venda?
- Qual o bairro com o maior preço mediano de venda para apartamentos?
- Qual o bairro com o maior preço mediano de venda para lojas?

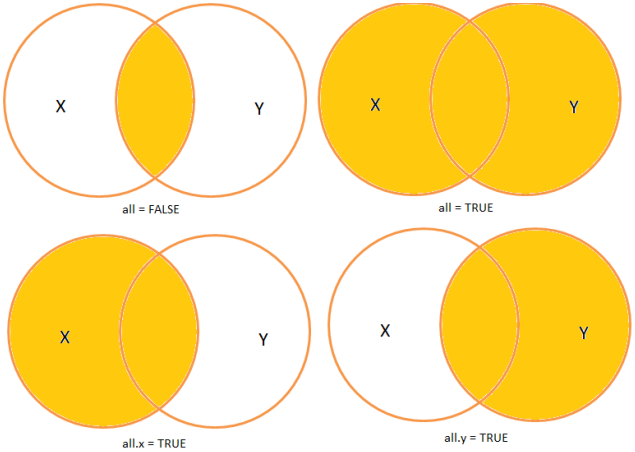
Colocando seus dados em forma

Merge

A função `merge()` serve para combinar data.frames. A função tenta identificar quais são as colunas identificadores em comum entre dois data frames para realizar a combinação. Para quem conhece SQL, a função `merge()` é equivalente ao `join`. Alguns argumentos da função são:

- **x**: um data.frame
- **y**: um data.frame
- **by**: a coluna em comum nos data.frames pela qual será feito o merge.
- **all**, **all.x**, **all.y**: especifica o tipo do merge. O default é FALSE e é equivalente ao "natural join" do SQL; "all" é equivalente ao "outer join"; "all.x" é equivalente ao "left outer join" e "all.y" é equivalente ao "right outer join".

Merge



Merge

Vejamos um exemplo. Vamos usar o aggregate para calcular a mediana do pm2 separadamente para aluguel e para venda. Depois vamos usar o merge para juntar os dados:

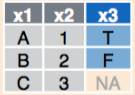
```
dados1 <- aggregate(formula = pm2 ~ bairro,
                    data = limpos,
                    subset = (limpos$tipo == "aluguel"),
                    FUN = median)

dados2 <- aggregate(formula = pm2 ~ bairro,
                    data = limpos,
                    subset = (limpos$tipo == "venda"),
                    FUN = median)

names(dados1)[2] <- "aluguel"
names(dados2)[2] <- "venda"
```

Merge

```
merge_all_false <- merge(dados1, dados2, all = FALSE)

merge_all_true <- merge(dados1, dados2, all = TRUE)

str(merge_all_false) # 11 linhas
## 'data.frame':    11 obs. of  3 variables:
##  $ bairro : chr  "Asa Norte" "Asa Sul" "Lago Norte" "Lago Sul" ...
##  $ aluguel: num  34.9 38.7 34.5 33.3 37.8 ...
##  $ venda  : num  9005 9167 6826 5500 9654 ...

str(merge_all_true) # 15 linhas
## 'data.frame':    15 obs. of  3 variables:
##  $ bairro : chr  "Asa Norte" "Asa Sul" "Brasília" "Granja do Torto" ...
##  $ aluguel: num  34.9 38.7 NA NA 34.5 ...
##  $ venda  : num  9005 9167 11844 4138 6826 ...
```

Joins do dplyr

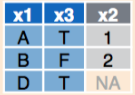
O dplyr também vem com funções de merge (join).



x1	x2	x3
A	1	T
B	2	F
C	3	NA

dplyr::**left_join(a, b, by = "x1")**

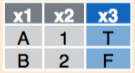
Join matching rows from b to a.



x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::**right_join(a, b, by = "x1")**

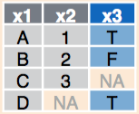
Join matching rows from a to b.



x1	x2	x3
A	1	T
B	2	F

dplyr::**inner_join(a, b, by = "x1")**

Join data. Retain only rows in both sets.



x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::**full_join(a, b, by = "x1")**

Join data. Retain all values, all rows.

Joins do dplyr

Alguns exemplos:

```
# igual merge com all = FALSE
innerjoin <- inner_join(dados1, dados2)
## Joining by: "bairro"

# igual merge com all = TRUE
fulljoin <- full_join(dados1, dados2)
## Joining by: "bairro"
```

Wide x Long

A tabela do `tapply` seria um exemplo do que chamamos de tabela no formato *wide*: as colunas representam grupos.
Já o formato do `aggregate` ou do `dplyr` é o que chamamos de formato **long**: há várias colunas identificadoras e apenas uma coluna de valores.
É bastante comum usar dados no formato **long** em pacotes de visualização, como o `ggplot2` ou `lattice`. O formato **wide**, por sua vez, é bastante utilizado em display de tabelas.
Assim, muitas vezes queremos passar nossos dados de um formato para o outro.

reshape2: Melt

Transforma dados no formato **wide** para o formato **long**:

```
# dados no formato wide
ap_wide <- tapply(aps$preco, list(aps$bairro, aps$m2_cat), median)
ap_wide[1:2, 1:3]
##           de 0 a 50 m2 de 50 a 150 m2 de 150 a 200 m2
## Asa Norte       260000         669999          1730000
## Asa Sul         350000         680000          1450000

#carrega o pacote e transforma em long
library(reshape2)
ap_long <- melt(ap_wide)
head(ap_long, 2)
##        Var1         Var2  value
## 1 Asa Norte de 0 a 50 m2 260000
## 2   Asa Sul de 0 a 50 m2 350000
```

reshape2: Cast - dcast

Transforma dados no formato **long** para o formato **wide**. Existe duas funções de **cast**: **dcast** e **acast** sendo que a primeira retorna um data.frame e a segunda um array. Vejamos o **dcast**:

```
long <- aggregate(pm2 ~ bairro + tipo + imovel + quartos,
                  data = limpos,
                  median)
head(long, 2)
##      bairro    tipo   imovel quartos pm2
## 1 Asa Norte aluguel kitinete       0  32
## 2   Asa Sul aluguel kitinete       0  35

wide <- dcast(data = long,
              formula = imovel + quartos ~ tipo + bairro,
              value.var = "pm2", sum)
wide[1:2, 1:4]
##        imovel quartos aluguel_Asa Norte aluguel_Asa Sul
## 1 apartamento       1                33               44
## 2 apartamento       2                33               30
```

reshape2: Cast - acast

Para mais dimensões do que duas, é preciso usar um *array*:

```
# note que cada dimensão é separada por ~
# para um data.frame só podemos ter um ~
# para arrays podemos ter vários
cast2 <- acast(long,
               imovel~quartos~tipo~bairro,
               value.var="pm2", sum)

str(cast2, vec.len=2)
##  num [1:5, 1:13, 1:2, 1:15] 0 0 ...
##  - attr(*, "dimnames")=List of 4
##   ..$ : chr [1:5] "apartamento" "casa" ...
##   ..$ : chr [1:13] "0" "1" ...
##   ..$ : chr [1:2] "aluguel" "venda"
##   ..$ : chr [1:15] "Asa Norte" "Asa Sul" ...
```

tidyr: reshape novo para data.frames

O `tidyr` é um pacote novo para fazer diversas tarefas para arrumar seus dados, entre elas transformar do formato **wide** para o formato **long** e vice-versa. Entretanto, só serve para data.frames. Vejamos alguns exemplos:

```
library(tidyr)
library(dplyr)
tidy_wide <- long %>%
  spread(imovel, pm2)

head(tidy_wide)
##      bairro    tipo quartos apartamento casa kitinete loja sala-comercial
## 1 Asa Norte aluguel       0          NA   NA       32   48             47
## 2 Asa Norte aluguel       1          33   NA       33   35             34
## 3 Asa Norte aluguel       2          33   NA       37   36             NA
## 4 Asa Norte aluguel       3          29   29       NA   NA             NA
## 5 Asa Norte aluguel       4          32   NA       NA   NA             NA
## 6 Asa Norte aluguel       5          NA   NA       NA   NA             50
```

tidyr: reshape novo para data.frames

Agora fazendo a operação inversa:

```
tidy_long <- tidy_wide %>%
  gather(imovel, pm2, -bairro, - tipo, - quartos)

head(tidy_long)
##      bairro    tipo quartos      imovel pm2
## 1 Asa Norte aluguel       0 apartamento  NA
## 2 Asa Norte aluguel       1 apartamento  33
## 3 Asa Norte aluguel       2 apartamento  33
## 4 Asa Norte aluguel       3 apartamento  29
## 5 Asa Norte aluguel       4 apartamento  32
## 6 Asa Norte aluguel       5 apartamento  NA
```

Há outras funções interessantes no **tidyr** como `unite()`, `separate()` e `extract()`.

Exemplo

Vamos calcular sua média de buscas no Google por hora, dia da semana, mês...

Programação em R

Julho, 2016

Trabalhando com textos no R

Criando textos

No R, textos são representados por vetores do tipo `character`. Você pode criar manualmente um elemento do tipo `character` colocando o texto entre aspas, podendo ser tanto aspas simples (‘texto’) quanto aspas duplas (“texto”).

```
# criando um vetor de textos
# aspas simples
x1 <- 'texto 1'

# aspas duplas
x2 <- "texto 2"
```

Criando textos

Como já vimos, para saber se um objeto é do tipo texto você pode utilizar a função `is.character()` e também é possível converter objetos de outros tipos para textos utilizando a função `as.character()`.

```
# criando um vetor de inteiros
x3 <- 1:10

# É texto? Não.
is.character(x3)
## [1] FALSE

# Convertendo para texto
x3 <- as.character(x3)

# Agora é texto? Sim.
is.character(x3)
## [1] TRUE
```

Operações com textos

Operações como ordenação e comparações são definidas para textos. A ordenação de um texto é feita de maneira lexicográfica, tal como em um dicionário.

```
# ordenação de letras
sort(c("c", "d", "a", "f"))
## [1] "a" "c" "d" "f"

# ordenação de palavras
# tal como um dicionário
sort(c("cor", "casa", "árvore", "zebra", "branco", "banco"))
## [1] "árvore" "banco"  "branco" "casa"   "cor"    "zebra"
```

Operações com textos

Como a comparação é lexicográfica, é preciso tomar alguns cuidados. Por exemplo, o texto "2" é maior do que o texto "100". Se por acaso seus números forem transformados em texto, você não vai receber uma mensagem de erro na comparação `"2" > "100"` mas sim um resultado errado: `TRUE`.

```
# CUIDADO!
2   >  100
## [1] FALSE
"2" > "100"
## [1] TRUE

# b > a
"b" > "a"
## [1] TRUE
```

Imprimindo textos

Se você estiver usando o R de modo interativo, chamar o objeto fará com que ele seja exibido na tela usando `print()`.

```
# Imprime texto na tela
print(x1)
## [1] "texto 1"

# Quando em modo interativo
# Equivalente a print(x1)
x1
## [1] "texto 1"
```

Imprimindo textos

Se você não estiver usando o R de modo interativo — como ao dar `source()` em um script ou dentro de um loop — é preciso chamar explicitamente uma função que exiba o texto na tela.

```
# sem print não acontece nada
for(i in 1:3) i

# com print o valor de i é exibido
for(i in 1:3) print(i)
## [1] 1
## [1] 2
## [1] 3
```

Imprimindo textos

Existem outras opções para "imprimir" e formatar textos além do `print()`. Uma função bastante utilizada para exibir textos na tela é a função `cat()` (concatenate and print).

```
cat(x1)
## texto 1

cat("A função cat exibe o texto sem aspas:", x1)
## A função cat exibe o texto sem aspas: texto 1
```

Imprimindo textos

Por padrão, `cat()` separa os textos com um espaço em branco, mas é possível alterar este comportamento com o argumento `sep`.

```
cat(x1, x2)
## texto 1 texto 2

cat(x1, x2, sep = " - ")
## texto 1 - texto 2
```

Imprimindo textos

Outra funções úteis são `sprintf()` e `format()`, principalmente para formatar e exibir números. Para mais detalhes sobre as funções, olhar a ajuda `?sprintf` e `?format`.

```
# %.2f (float, 2 casas decimais)
sprintf("R$ %.2f", 312.12312312)
## [1] "R$ 312.12"

# duas casas decimais, separador de milhar e decimal
format(10500.5102, nsmall=2, big.mark=".", decimal.mark=",")
## [1] "10.500,51"
```

Caracteres especiais

Como fazemos para gerar um texto com separação entre linhas no R? Criemos a separação de linhas manualmente para ver o que acontece:

```
texto_nova_linha <- "texto
com nova linha"

texto_nova_linha
## [1] "texto\ncom nova linha"
```

Note que aparece um `\n` no meio do texto.

Caracteres especiais

O \n é um caractere especial que simboliza justamente uma nova linha. Quando você exibe um texto na tela com `print()`, caracteres especiais não são processados e aparecem de maneira literal. Já se você exibir o texto na tela usando `cat()`, os caracteres especiais serão processados. No nosso exemplo, o \n será exibido como uma nova linha.

```
# print: \n aparece literalmente
print(texto_nova_linha)
## [1] "texto\ncom nova linha"

# cat: \n aparece como nova linha
cat(texto_nova_linha)
## texto
## com nova linha
```

Caracteres especiais

Caracteres especiais são sempre "escapados" com a barra invertida \ . Além da nova linha (\n), outros caracteres especiais recorrentes são o tab (\t) e a própria barra invertida, que precisa ela mesma ser escapada (\\). Vejamos alguns exemplos:

```
cat("colocando uma \nnova linha")
## colocando uma
## nova linha
cat("colocando um \ttab")
## colocando um     tab
cat("colocando uma \\ barra")
## colocando uma \ barra
cat("texto com novas linhas e\numa barra no final\n\\")
## texto com novas linhas e
## uma barra no final
## \
```

Caracteres especiais

Para colocar aspas simples ou duplas **dentro** do texto há duas opções. A primeira é alternar entre as aspas simples e duplas, uma para definir o objeto do tipo `character` e a outra servido literalmente como aspas.

```
# Aspas simples dentro do texto
aspas1 <- "Texto com 'aspas' simples dentro"
aspas1
## [1] "Texto com 'aspas' simples dentro"

# Aspas duplas dentro do texto
aspas2  <- 'Texto com "aspas" duplas dentro'
cat(aspas2)
## Texto com "aspas" duplas dentro
```

Caracteres especiais

Outra opção é colocar as aspas como caracter especial. Neste caso, não é preciso alternar entre aspas simples e duplas.

```
aspas3 <- "Texto com \"aspas\" duplas"
cat(aspas3)
## Texto com "aspas" duplas

aspas4 <- 'Texto com \'aspas\' simples'
cat(aspas4)
## Texto com 'aspas' simples
```

Utilidade das funções de exibição

Qual a utilidade de funções que exibam coisas na tela? Um caso bastante comum é exibir mensagens durante a execução de alguma rotina ou função. Por exemplo, você pode exibir o percentual de conclusão de um loop a cada 25 rodadas:

```
for (i in 1:100) {
  # imprime quando o resto da divisão
  # de i por 25 é igual a 0
  if (i %% 25 == 0) {
    cat("Executando: ", i, "%\n", sep = "")
  }
  # alguma rotina
  Sys.sleep(0.01)
}
## Executando: 25%
## Executando: 50%
## Executando: 75%
## Executando: 100%
```

Utilidade das funções de exibição

Outro uso frequente é criar métodos de exibição para suas próprias classes. Vejamos um exemplo simples de uma fução base do R, a função `rle()`, que computa tamanhos de sequências repetidas de valores em um vetor. O resultado da função é uma lista, mas ao exibirmos o objeto na tela, o `print` não é igual ao de uma lista comum:

```
x <- rle(c(1,1,1,0))

# resultado é uma lista
str(x)
## List of 2
##  $ lengths: int [1:2] 3 1
##  $ values : num [1:2] 1 0
##  - attr(*, "class")= chr "rle"
```

Utilidade das funções de exibição

```
# print do objeto na tela não é como uma lista comum
x
## Run Length Encoding
##   lengths: int [1:2] 3 1
##   values : num [1:2] 1 0

# tirando a classe do objeto veja que o print agora é como uma lista comum
unclass(x)
## $lengths
## [1] 3 1
##
## $values
## [1] 1 0
```

Utilidade das funções de exibição

Isso ocorre porque a classe `rle` tem um método de `print` próprio, `print.rle()`:

```
print.rle <- function (x, digits = getOption("digits"), prefix = "", ...)
{
    if (is.null(digits))
        digits <- getOption("digits")
    cat("", "Run Length Encoding\n", "  lengths:", sep = prefix)
    utils::str(x$lengths)
    cat("", "  values :", sep = prefix)
    utils::str(x$values, digits.d = digits)
    invisible(x)
}
```

Tamanho do texto

A função `nchar()` retorna o número de caracteres de um elemento do tipo texto. Note que isso é diferente da função `length()` que retorna o tamanho do **vetor**.

```
# O vetor x1 tem apenas um elemento
length(x1)
## [1] 1

# O elemento do vetor x1 tem 7 caracteres
# note que espaços em brancos contam
nchar(x1)
## [1] 7
```

Tamanho do texto

A função `nchar()` é vetorizada.

```
# vetor do tipo character
y <- c("texto 1", "texto 11")

# vetor tem dois elementos
length(y)
## [1] 2

# O primeiro elemento tem 7 caracteres
# O segundo 8.
nchar(y) # vetorizada
## [1] 7 8
```

Manipulando textos

Manipulação de textos é uma atividade bastante comum na análise de dados. O R possui uma série de funções para isso e suporta o uso de expressões regulares. Nesta seção veremos as principais funções de manipulação de textos.

Colando (ou concatenando) textos

A função `paste()` é uma das funções mais úteis para manipulação de textos. Como o próprio nome diz, ela serve para "colar" textos.

```
# Colando textos
tipo <- "Apartamento"
bairro <- "Asa Sul"
texto <- paste(tipo,"na", bairro )
texto
## [1] "Apartamento na Asa Sul"
```

Colando (ou concatenando) textos

Por default, `paste()` separa os textos com um espaço em branco. Você pode alterar isso modificando o argumento `sep`. Caso não queira nenhum espaço entre as strings, basta definir `sep = ""` ou utilizar a função `paste0()`. Como usual, todas essas funções são vetorizadas.

```
# separação padrão
paste("x", 1:5)
## [1] "x 1" "x 2" "x 3" "x 4" "x 5"

# separando por ponto
paste("x", 1:5, sep=".")
## [1] "x.1" "x.2" "x.3" "x.4" "x.5"

# sem separação, usando paste0.
paste0("x", 1:5)
## [1] "x1" "x2" "x3" "x4" "x5"
```

Colando (ou concatenando) textos

Note que foram gerados 5 elementos diferentes nos exemplos acima. É possível "colar" todos os elementos em um único texto com a opção `collapse()`.

```
paste("x", 1:5, sep="", collapse = " ")
## [1] "x1 x2 x3 x4 x5"

paste("x", 1:5, sep="", collapse=" --> ")
## [1] "x1 --> x2 --> x3 --> x4 --> x5"
```

Separando textos

Outra atividade frequente em análise de dados é separar um texto em elementos diferentes. Por exemplo, suponha que você tenha que trabalhar com um conjunto de números, mas que eles estejam em um formato de texto separados por ponto e vírgula:

```
dados <- "1;2;3;4;5;6;7;8;9;10"
dados
## [1] "1;2;3;4;5;6;7;8;9;10"
```

Com a função `strsplit()` é fácil realizar essa tarefa:

```
dados_separados <- strsplit(dados, split=";")
dados_separados
## [[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Separando textos

Note que o resultado da função é uma lista. Agora é possível converter os dados em números e trabalhar normalmente.

```
# convertendo o resultado em número
dados_separados <- as.numeric(dados_separados[[1]])

# agora é possível trabalhar com os números
# média
mean(dados_separados)
## [1] 5.5
# soma
sum(dados_separados)
## [1] 55
```

Maiúsculas e minúsculas

Passando textos para CAIXA ALTA ou caixa baixa.

```
toupper(texto)
## [1] "APARTAMENTO NA ASA SUL"

tolower(texto)
## [1] "apartamento na asa sul"
```

Encontrando partes de um texto

Quando você estiver trabalhando com suas bases de dados, muitas vezes será preciso encontrar certas palavras ou padrões dentro do texto. Por exemplo, imagine que você tenha uma base de dados de aluguéis de apartamentos e você gostaria de encontrar imóveis em um certo endereço. Vejamos este exemplo com dados online de aluguel em Brasília.

```
# Carrega arquivo de anúncios de aluguel (2014)
arquivo <- url("https://dl.dropboxusercontent.com/u/44201187/aluguel.rds")
con <- gzcon(arquivo)
aluguel <- readRDS(con)
close(con)
```

Encontrando partes de um texto

Vejamos a estrutura da nossa base de dados:

```
str(aluguel, vec.len = 1)
## 'data.frame':    2612 obs. of  5 variables:
##  $ bairro  : chr  "Asa Norte" ...
##  $ endereco: chr  "CLN 310 BLOCO A " ...
##  $ quartos : num  1 1 ...
##  $ m2      : num  22.9 26 ...
##  $ preco   : num  650 750 ...
##  - attr(*, "na.action")=Class 'omit'  Named int [1:120] 15943 16001 ...
##   .. ..- attr(*, "names")= chr [1:120] "15943" ...
```

Encontrando partes de um texto

Temos mais de 2 mil anúnciso, como encontrar aqueles apartamentos que queremos, como, por exemplos, os que contenham "CLN 310" no endereço? Neste caso você pode utilizar a função `grep()` para encontrar padrões dentro do texto. A função retornará o índice das observações que contém o texto:

```
busca_indice <- grep(pattern = "CLN 310", aluguel$endereco)
busca_indice
## [1]    1 1812
aluguel[busca_indice, ]
##         bairro                              endereco quartos m2 preco
## 1    Asa Norte                       CLN 310 BLOCO A        1 23   650
## 1812 Asa Norte CLN 310 BLOCO E ENTRADA 52 SALA 216        0 30   900
```

Encontrando partes de um texto

Uma variante da função `grep()` é a função `grepl()`, que realiza a mesma coisa, mas ao invés de retornar um índice numérico, retorna um vetor lógico:

```
busca_logico <- grepl(pattern = "CLN 310", aluguel$endereco)
str(busca_logico)
##  logi [1:2612] TRUE FALSE FALSE FALSE FALSE FALSE ...
aluguel[busca_indice, ]
##         bairro                              endereco quartos m2 preco
## 1    Asa Norte                       CLN 310 BLOCO A        1 23   650
## 1812 Asa Norte CLN 310 BLOCO E ENTRADA 52 SALA 216        0 30   900
```

Substituindo partes de um texto

A função `sub()` substitui o primeiro padrão (`pattern`) que encontra:

```
texto2 <- paste(texto, ", Apartamento na Asa Norte")
texto2
## [1] "Apartamento na Asa Sul , Apartamento na Asa Norte"

# Vamos substituir "apartamento" por "Casa"
# Mas apenas o primeiro caso
sub(pattern = "Apartamento",
    replacement = "Casa",
    texto2)
## [1] "Casa na Asa Sul , Apartamento na Asa Norte"
```

Substituindo partes de um texto

Já a função `gsub()` substitui todos os padrões que encontra:

```
# Vamos substituir "apartamento" por "Casa"
# Agora em todos os casos
gsub(pattern = "Apartamento",
    replacement = "Casa",
    texto2)
## [1] "Casa na Asa Sul , Casa na Asa Norte"
```

Substituindo partes de um texto

Você pode usar as funções `sub()` e `gsub()` para "deletar" partes indesejadas do texto, basta colocar como `replacement` um caractere vazio `""`. Um exemplo bem corriqueiro, quando se trabalha com com nomes de arquivos, é a remoção das extensões:

```
# nomes dos arquivos
arquivos <- c("simulacao_1.csv","simulacao_2.csv")

# queremos eliminar a extensão .csv
# note que o ponto precisa ser escapado
nomes_sem_csv <- gsub("\\.csv", "", arquivos)
nomes_sem_csv
## [1] "simulacao_1" "simulacao_2"
```

Extraindo partes específicas de um texto

Às vezes você precisa extrair apenas algumas partes específicas de um texto, em determinadas posições. Para isso você pode usar as funções `substr()` e `substring()`. Para essas funções, você basicamente passa as posições dos caracteres inicial e final que deseja extrair.

```
# extraindo caracteres da posição 4 à posição 8
x <- "Um texto de exemplo"
substr(x, start = 4, stop = 8)
## [1] "texto"
```

Extraindo partes específicas de um texto

É possível utilizar essas funções para alterar partes específicas do texto.

```
# substituindo caracteres da posição 4 à posição 8
substr(x, start = 4, stop = 8) <- "TEXTO"
x
## [1] "Um TEXTO de exemplo"
```

Extraindo partes específicas de um texto

A principal diferença entre `substr()` e `substring()` é que a segunda permite você passar vários valores iniciais e finais:

```
# pega caracteres de (4 a 8) e de (10 a 11)
substring(x, first = c(4, 10), last = c(8, 11))
## [1] "TEXTO" "de"

# pega caracteres de (1 ao último), (2 ao último) ...
substring("abcdef", first = 1:6)
## [1] "abcdef" "bcdef"  "cdef"   "def"    "ef"     "f"
```

Facilitando tudo: o pacote `stringr`

Você deve ter notado que vimos várias funções com nome bem diferentes, `paste()`, `sub()`, `gsub()`, `substr()`, `substring()`... para facilitar sua vida, o Hadley Wickham criou um pacote chamado `stringr` que tem várias funções de manipulação de textos com interface mais consistente e mais simples de usar. Todas as funções começam com o nome `str_` e o primeiro argumento é sempre a string que desejamos trabalhar.

```
# Instale o pacote se não tiver instalado
# install.packages(stringr)
library(stringr)
```

Facilitando tudo: o pacote stringr

```
# tolower e toupper e Title
str_to_lower(texto)
## [1] "apartamento na asa sul"
str_to_upper(texto)
## [1] "APARTAMENTO NA ASA SUL"
str_to_title(texto)
## [1] "Apartamento Na Asa Sul"

# removendo espaços em branco
str_trim(c(" 12    ", "   12321   "))
## [1] "12"    "12321"

# preenchendo até completar um número de caracteres
str_pad(c("12", "12321"), 10, pad = 0)
## [1] "0000000012" "0000012321"
```

Facilitando tudo: o pacote stringr

```
#  split
str_split(dados, ";")
## [[1]]
##  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

# detectando textos
cln_310 <- str_detect(aluguel$endereco, pattern = "CLN 310")
which(cln_310)
## [1]    1 1812

# substituindo partes do texto (primeira ocorrência)
str_replace(texto2, "Apartamento", "Casa")
## [1] "Casa na Asa Sul , Apartamento na Asa Norte"

# substituindo partes do texto (todas ocorrência)
str_replace_all(texto2, "Aparamento", "Casa")
## [1] "Apartamento na Asa Sul , Apartamento na Asa Norte"
```

Facilitando tudo: o pacote stringr

```
# extraindo textos
x <- "dsj1302932kl2014-09-01da5465464-546jsl139022015-12-12çsa"

# extraindo primeira ocorrência
str_extract(x, pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}")
## [1] "2014-09-01"

# estraindo todas ocorrências
str_extract_all(x, pattern = "[0-9]{4}-[0-9]{2}-[0-9]{2}")
## [[1]]
## [1] "2014-09-01" "2015-12-12"
```

Expressões regulares

Tudo o que vimos aqui aceita expressões regulares! Com expressões regulares é possível fazer buscas simples mas poderosas, como vimos no último exemplo. Entretanto, infelizmente, este tema foge ao escopo do nosso curso.

Representando dados categóricos: os fatores

Criando fatores: `factor()`

Fatores são uma forma de representar dados categóricos no R. Você pode criar um fator com a função `factor()`.

```
bairros <- c("Asa Sul","Asa Norte", "Sudoeste", "Asa Sul","Asa Norte",
             "Noroeste","Asa Norte", "Sudoeste", "Asa Norte")
fac_bairros <- factor(bairros)
str(fac_bairros)
##  Factor w/ 4 levels "Asa Norte","Asa Sul",..: 2 1 4 2 1 3 1 4 1
```

Note que os fatores são representados por números, mas têm um atributo `levels` com os nomes de cada categoria.

Fatores ordenados e não ordenados

O fator `fac_bairros` que criamos anteriormente é o que chamamos de fator não ordenado, pois uma categoria não é "superior" à outra. Entretanto, é possível criar fatores ordenados no R. Com ordenação, é possivel realizar comparações de variáveis categóricas.

```
temps <- c("Alta", "Baixa", "Média", "Média", "Média",
           "Alta", "Alta", "Média", "Baixa","Baixa","Baixa")

fac_temps <- factor(temps, order=TRUE,
                    levels=c("Baixa", "Média", "Alta"))

fac_temps[1] > fac_temps[2]
## [1] TRUE
```

Mudandos os levels

Podemos facilmente renomear todos os factors mudando apenas os levels. Vamos abreviar os levels das temperaturas para “B”, “M”, “A” e adicionar um outro “MA” que significaria “Muito Alta”.

```
fac_temps
##  [1] Alta  Baixa Média Média Média Alta  Alta  Média Baixa Baixa Baixa
## Levels: Baixa < Média < Alta
levels(fac_temps) <- c("B", "M", "A", "MA")
fac_temps
##  [1] A B M M M A A M B B B
## Levels: B < M < A < MA
```

Summary e table

Note que os summary's de um vetor de caracteres e um factor são diferentes.

```
summary(temps)
##    Length     Class      Mode
##        11 character character
summary(fac_temps)
##  B  M  A MA
##  4  4  3  0
```

Summary e table

O `summary()` do fator usa a função `table()`. Mas veja as diferenças entre um `table()` aplicado a um texto e aplicado a um `factor`.

```
table(temps)
## temps
##  Alta Baixa Média
##     3     4     4

table(fac_temps)
## fac_temps
##  B  M  A MA
##  4  4  3  0
```

Summary e table

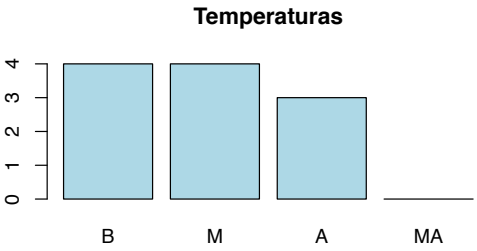
Você pode usar o `table()` para montar tabelas com mais de um fator.

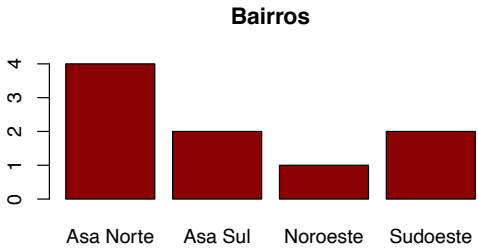
```
grupo <- rep(c("G1", "G2"),length.out=11)
table(grupo, temps)
##       temps
## grupo Alta Baixa Média
##    G1    2     2     2
##    G2    1     2     2
```

plot

Fatores tem um método para `plot`.

```
# gráficos dispostos em 1 linha e 2 colunas
par(mfrow=c(1,2))
#plot
plot(fac_temps, col="lightblue", main="Temperaturas")
plot(fac_bairros, col="darkred", main="Bairros")
```





Cuidado!

O default em muitas funções do R é transformar strings em factors (textos para fatores). Isto pode ser fonte de erros caso não se tome cuidado. Por exemplo:

```
numeros <- c(5,6,7,8)
numeros <- as.factor(numeros)
as.numeric(numeros)
## [1] 1 2 3 4
```

Cuidado!

O que aconteceu?

```
str(numeros)
##  Factor w/ 4 levels "5","6","7","8": 1 2 3 4

# como extrair os números de volta
as.numeric(as.character(numeros))
## [1] 5 6 7 8
```

Cuidado!

Fatores também são estruturas rígidas, você não pode incluir uma observação em um fator para a qual não exista um level.

```
# com texto, sem problemas
bairros[5] <- "Octogonal"
bairros
## [1] "Asa Sul"   "Asa Norte" "Sudoeste"  "Asa Sul"   "Octogonal" "Noroeste"
## [7] "Asa Norte" "Sudoeste"  "Asa Norte"
# com factor
fac_bairros[5] <- "Octogonal"
## Warning in `[<-.factor`(`*tmp*`, 5, value = "Octogonal"): invalid factor level,
## NA generated
fac_bairros
## [1] Asa Sul   Asa Norte Sudoeste  Asa Sul   <NA>      Noroeste  Asa Norte
## [8] Sudoeste  Asa Norte
## Levels: Asa Norte Asa Sul Noroeste Sudoeste
```

Recomendação

Recomenda-se, portanto, trabalhar com strings e somente transformar para factor quando realmente necessário. Na maior parte dos casos, use o parâmetro `stringsAsFactors = FALSE` para evitar que strings sejam transformadas em fatores sem que você perceba.

Representando e manipulando datas

Date

O R também tem uma série de funções para lidar com datas.Tendo uma data em formato texto, como "01 de janeiro de 2014", é possível transformá-la em um objeto do tipo "Date", em que são possíveis operações como comparação, adição etc.

```
Data <- "01 de janeiro de 2014"

as.Date(Data, format="%d de %B de %Y")
## [1] "2014-01-01"
```

Note que tivemos que explicar para o R, via `format`, como a data está codificada no texto. Em palavras, estamos dizendo que: primeiramente, temos o dia (%d) seguido da palavra "de"; depois temos o mês por extenso (%b) seguido da palavra "de"; e, por fim o ano com 4 dígitos (%Y).

Date

Note que as.Date transforma o objeto em tipo Date.

```
Data <- as.Date(Data, format="%d de %B de %Y"); str(Data)
##  Date[1:1], format: "2014-01-01"
```

Agora podemos fazer operações, tais como:

```
Data + 1
## [1] "2014-01-02"
Data - 1
## [1] "2013-12-31"
weekdays(Data)
## [1] "Quarta Feira"
Data > "2013-12-01"
## [1] TRUE
```

Date

```
months(Data + 31)
## [1] "Fevereiro"

quarters(Data)
## [1] "Q1"

seq.Date(from = Data, by = 1, length.out = 10L)
##  [1] "2014-01-01" "2014-01-02" "2014-01-03" "2014-01-04" "2014-01-05"
##  [6] "2014-01-06" "2014-01-07" "2014-01-08" "2014-01-09" "2014-01-10"
```

Date

As opções do format são:

Format	Descrição
%Y	Ano com 4 dígitos
%y	Ano com 1/2 dígitos
%m	Mês com 4 dígitos
%B	Mês por Extenso completo
%b	Mês por Extenso abreviado
%d	dia com 1/2 dígitos
%A	Dia da semana por extenso
%a	Dia da semana por abreviado
%w	Dia da semana número

Date

Você também pode usar estas opções para imprimir a data no formato desejado. Por exemplo:

```
Data
## [1] "2014-01-01"
cat(
  format(Data,
         format="isto ocorreu numa %A, \ndia %d de %B de %Y.")
  )
## isto ocorreu numa Quarta Feira,
## dia 01 de Janeiro de 2014.
```

POSIXct e POSIXlt

É possível, ainda, adicionar informações sobre hora, minuto e segundo para as datas. Existem dois formatos principais que tratam disso, o `POSIXct` (número de segundos após 1970) e `POSIXlt` (lista nomeada com objetos de data e hora).

Format	Descrição
%H	Hora (00-23)
%I	Hora (1-12)
%M	Minutos (00-59)
%S	Segundos (00-61)
%p	AM/PM (não para Brasil)

POSIXct e POSIXlt

Exemplo:

```
Data <- "01 de janeiro de 2014 às 14h e 40m"
ct <- as.POSIXct(Data,
                 format="%d de %B de %Y às %Hh e %Mm")
ct
## [1] "2014-01-01 14:40:00 BRST"
lt <- as.POSIXlt(Data,
                 format="%d de %B de %Y às %Hh e %Mm")
lt
## [1] "2014-01-01 14:40:00 BRST"
```

POSIXct e POSIXlt

As operações são feitas em segundos:

```
ct + 3600 # soma uma hora
## [1] "2014-01-01 15:40:00 BRST"

lt - 60 # subtrai um minuto
## [1] "2014-01-01 14:39:00 BRST"
```

As funções anteriores continuam funcionando:

```
months(ct)
## [1] "Janeiro"

weekdays(lt)
## [1] "Quarta Feira"
```

Para aprofundar

Existem alguns pacotes que estendem as funcionalidades para lidar com datas e séries temporais no R.

- `lubridate`: funções de conveniência para lidar com formatação de datas.
- `ts`, `zoo`, `xts`: pacotes que estendem os objetos de séries temporais do R, interessantes para quem modela séries temporais. O `xts` é construído em cima tanto do `zoo` quanto do `ts`.

Programação em R

Julho, 2016

O processo de análise de dados

Todo processo de análise de dados envolve, em geral, três grandes etapas: a entrada de dados no programa; o processamento e análise desses dados; a saída de dados do programa. Nesta aula apresentaremos a primeira e a última etapa: alguns modos de entrada e saída de dados no R.

Arquivos, pastas e os formatos próprios do R: RData e rds

Diretório de trabalho

Toda vez que você abre uma sessão do R, ele realizará operações de leitura e gravação de dados no diretório de trabalho.

```
# qual é o diretório de trabalho atual?
# no seu computador o resultado vai ser diferente
getwd() # get working directory
[1] "C:/"
```

A qualquer momento você pode alterar o diretório de trabalho com `setwd()`.

```
setwd("D:/Curso de R")
getwd()
[1] "D:/Curso de R"
```

Diretório de trabalho

Note que o separador é o contrário do que se usa no windows, pois a barra \ é um comando especial. Uma opção, caso você queira usar o padrão do windows, é usar duas barras.

```
setwd("C:\\diretorio1\\diretorio2")
```

Existe, ainda, uma função de conveniência que cria o caminho do arquivo para você a `file.path()`

```
file.path("C:",
          "diretorio1",
          "diretorio2",
          "arquivo.csv")
## [1] "C:/diretorio1/diretorio2/arquivo.csv"
```

Manipulando arquivos e pastas

O R possui uma série de funções para manipulação de arquivos e pastas. Vejamos algumas:

```
# listar os arquivos e diretórios que estão em uma pasta
list.files()
## [1] "Dados"     "io.pdf"    "io.Rmd"    "Rprof.out"

# listar os diretórios e subdiretórios que estão
# em uma pasta
list.dirs()
## [1] "."                   "./Dados"             "./Dados/Agosto"
## [4] "./Dados/outra pasta" "./Dados/Setembro"
```

Manipulando arquivos e pastas

```
# verificar se um arquivo existe
file.exists("io.pdf")
## [1] TRUE

file.exists("arquivoquenaoexiste")
## [1] FALSE
```

Manipulando arquivos e pastas

```
# criar arquivo
file.create("arquivoquenaoexiste")
## [1] TRUE

file.exists("arquivoquenaoexiste")
## [1] TRUE
```

Manipulando arquivos e pastas

```
# remover arquivo
file.remove("arquivoquenaoexiste")
## [1] TRUE

file.exists("arquivoquenaoexiste")
## [1] FALSE
```

Manipulando arquivos e pastas

```
# criar pastas
dir.create("Nova Pasta"); list.dirs()
## [1] "."                    "./Dados"              "./Dados/Agosto"
## [4] "./Dados/outra pasta" "./Dados/Setembro"     "./Nova Pasta"

file.remove("Nova Pasta"); list.dirs()
## [1] TRUE
## [1] "."                    "./Dados"              "./Dados/Agosto"
## [4] "./Dados/outra pasta" "./Dados/Setembro"
```

Manipulando arquivos e pastas

É possível também renomear e mover arquivos com `file.rename()`, ou copiar arquivos `file.copy()` entre outras funções. No limite, você pode usar diretamente os comandos do `shell` (DOS) do Windows.

```
# Cria diretório via DOS
shell("md teste")

# Remove diretório via DOS
shell("rmdir teste")
```

RData, rda e rds

Como vimos em exemplos durante as aulas, o R tem dois formatos próprios:

- `RData` ou `rda`: salva um ou vários objetos da área de trabalho. Carrega com o mesmo nome que foi salvo.
- `rds`: salva apenas um objeto. É possível carregar com nome diferente.

RData, rda e rds

Você pode salvar objetos no formato `RData` ou `rda` com a função `save()`. Para carregar, use a função `load()`. É possível salvar mais de um objeto ao mesmo tempo.

```
mtcars <- mtcars
# salva em rdata
save(mtcars, file = "mtcars.RData")
rm(mtcars)
ls()
## character(0)

# carrega novamente
load(file = "mtcars.RData")
ls()
## [1] "mtcars"
```

RData, rda e rds

A segunda opção para salvar e ler objetos do R são os objetos do tipo `rds`. Para tanto você irá utilizar as funções `saveRDS()` e `readRDS()`. Uma das principais diferenças com relação a `save()` e `load()` é que, enquanto estas salvam e carregam os objetos com o seu nome original, as funções RDS permitem a você carregar o objeto com um nome diferente.

```
saveRDS(mtcars, file = "mtcars.rds")
rm(mtcars)

# carrega em um objeto com nome diferente
dados <- readRDS("mtcars.rds")
ls()
## [1] "dados"
```

Planilhas e web: csv, xlsx, XML e JSON

CSV

Uma das formas mais convenientes de importar e exportar dados pelo R e por meio de arquivos `.csv`. O csv é um formato entendido por virtualmente quase todo software. Existe uma série de funções (derivadas da `read.table`) que fazem este trabalho, veja mais em `?read.table`.
Argumentos que necessitam atenção especial:

- `dec`: determina o símbolo utilizado para decimal. No Brasil, utilizamos "," mas em muitos outros lugares o padrão é ".".
- `sep`: como os dados estão separados? Por tab (\t), por vírgula (,), por ponto e vírgula (;), por espaço (" ")?
- `fileEncoding`: o padrão do R, em geral, é trabalhar com caracteres ASCII. No Brasil, são comuns os padrões latin1 ou UTF-8. Ler o arquivo com o padrão errado pode resultar em caracteres "estranhos".

CSV

Para ilustrar como salvar um csv, vamos utilizar a função `write.csv2()`. Ela e igual à `write.csv()` mas já tem como padrão `sep = ";"` e `dec = ","`, que são comuns no Brasil.

```
write.csv2(mtcars, "mtcars.csv")
```

csv

Para ilustrar como ler um csv, vamos utilizar a função `read.csv()`, colocando os parâmetros corretos para leitura:

```
carros <- read.csv("mtcars.csv", dec = ",", sep = ";")
```

Neste caso poderíamos ter lido também com `read.csv2()` que já teria os parâmetros desejados.

xlsx

Outra forma de interação bastante comum é com dados em planilha Excel. Há vários pacotes que permitem ler e salvar arquivos em Excel. Esse pacotes, em geral, usam bibliotecas externas escritas em outras linguagens, por exemplo:

Pacote	Acessa o Excel por
readxl	**C e C++ (somente leitura)**
xlsx	**Java**
XLconnect	Java
openxlsx	C++
RODBC	ODBC
gdata	perl

É importante saber esse detalhe porque você pode encontrar alguma incompatibilidade dependendo do computador que estiver usando. Para leitura, recomendo o pacote `readxl` e para gravação o `xlsx`.

xlsx

Vejamos um exemplo de como salvar os dados em excel com o pacote `xlsx`. Para salvar o objeto `mtcars` como uma planilha excel, você digitaria o seguinte comando:

```
library(xlsx)
write.xlsx(mtcars, "mtcars.xlsx")
```

xlsx

Vamos agora ler os dados da planilha excel que acabamos de criar tanto com o pacote `xlsx` quanto com o pacote `readxl`:

```
ex1 <- read.xlsx("mtcars.xlsx",sheetIndex = 1)

# com read_excel
library(readxl)
ex2 <- read_excel("mtcars.xlsx", sheet = 1)
```

JSON e XML

Dados provenientes da web em geral vêm em formatos JSON ou XML. Os pacotes que recomendo para leitura desses dados são o `jsonlite` e `xml2`, respectivamente. Vimos um exemplo de leitura de dados JSON quando carregamos nossos dados de buscas do Google.

Formato	Pacote recomendado
JSON	jsonlite
XML	xml2

Pacotes estatísticos e bancos de dados

STATA, SPP e SAS

Para ler dados de pacotes estatísticos, existem dois pacotes principais: o `foreign` e o `haven`. Aqui recomendo utilizar o `haven`. As funções são bem intuitivas e simples de utilizar. Principais funções:

Função do Haven	Pacote estatístico
`read_dta()`	Lê arquivos do STATA
`read_stata()`	Lê arquivos do STATA
`write_stata()`	Salva arquivos STATA
`read_por()`	Lê arquivos do SPSS
`read_sav()`	Lê arquivos do SPSS
`read_spss()`	Lê arquivos do SPSS
`write_sav()`	Salva arquivos do SPSS
`read_sas()`	Lê arquivos do SAS

Bancos de dados

Por fim, há pacotes do R para conversar com bancos de dados externos, entre eles cabe destacar:

Pacote	Conversa com...
RMySQL	MySQL
RDOBC	SQL Server via ODBC
RPostgres	Postgres
RSQLite	SQLite
rmongodb, mongolite	MongoDB
RCassandra	Cassandra

... e vários outros pacotes!

Programação em R

Julho, 2016

Base R: desenhando seus gráficos

Dados

Vamos resgatar nossa base de dados de imóveis e passar alguns filtros.

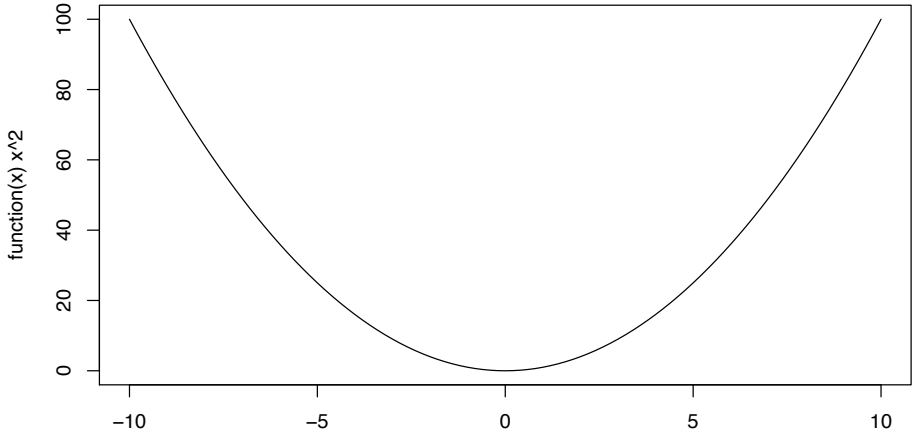
```
library(ggplot2)
library(dplyr)
limpos <- readRDS("Dados/limpos.rds")
venda <- limpos %>%
         filter(tipo == "venda",
                quartos < 10,
                imovel == "apartamento",
                bairro %in% c("Asa Sul","Asa Norte",
                              "Noroeste","Sudoeste"))
```

Função plot

A função plot é uma função genérica, e possui vários métodos diferentes dependendo do tipo de objeto.

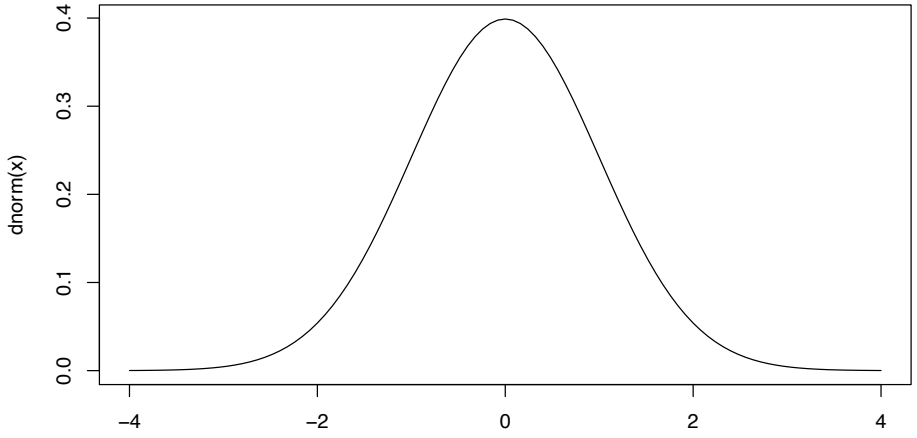
Plot de função

```
plot(function(x) x^2, from = -10, to = 10)
```



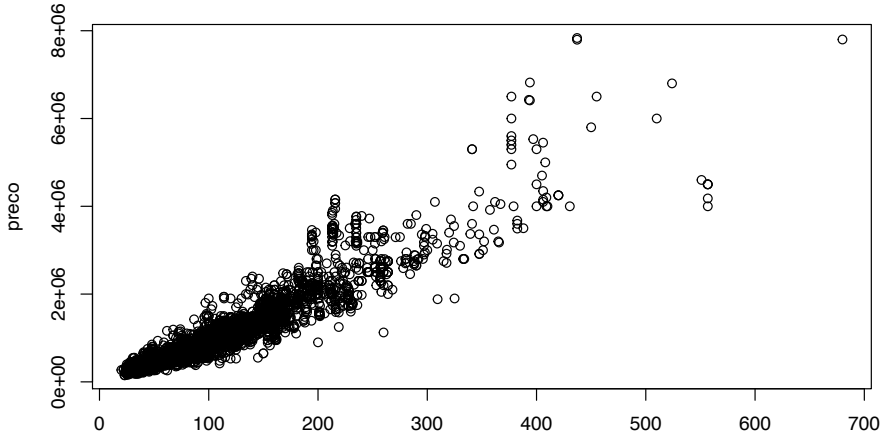
Plot de expressão

```
curve(dnorm(x), -4, 4)
```



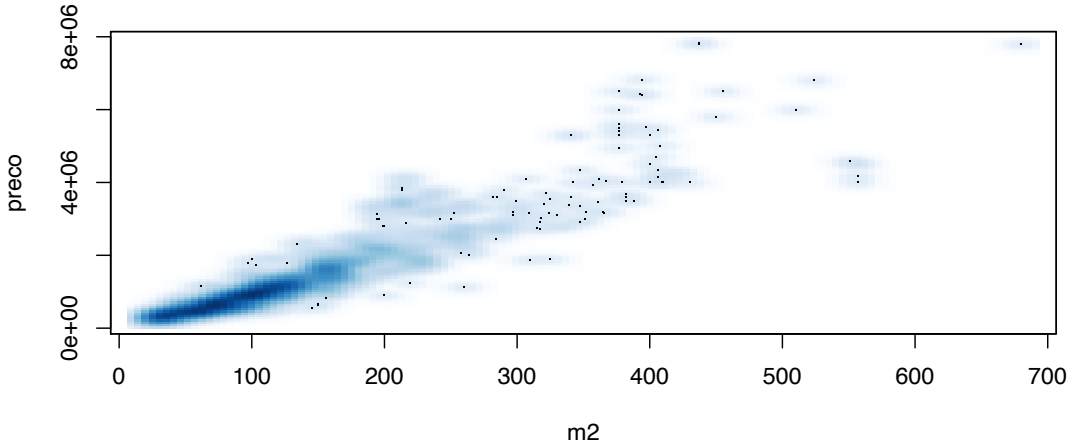
Scatter Plot

```
with(venda, plot(m2, preco))
```



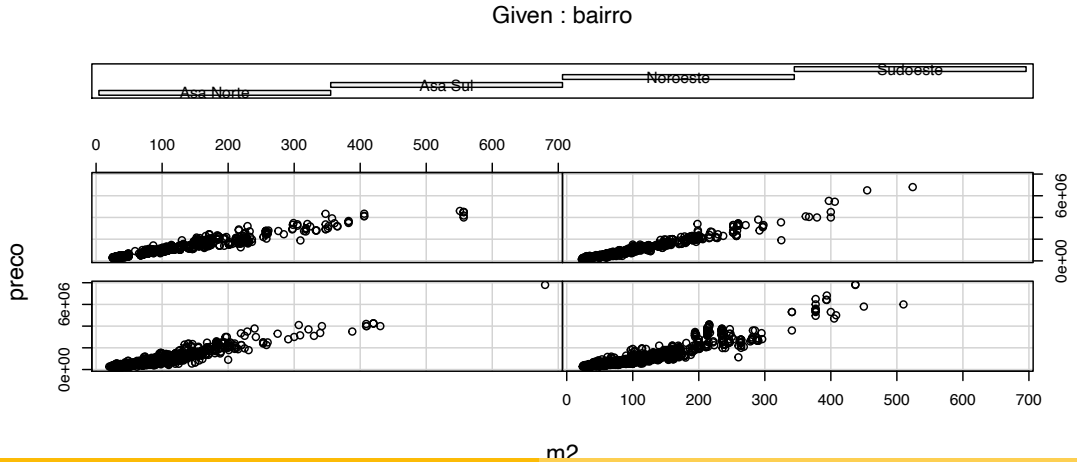
Smooth Scatter Plot

```
with(venda, smoothScatter(m2, preco))
```



Coplot

```
with(venda, coplot(preco ~ m2 | bairro))
```

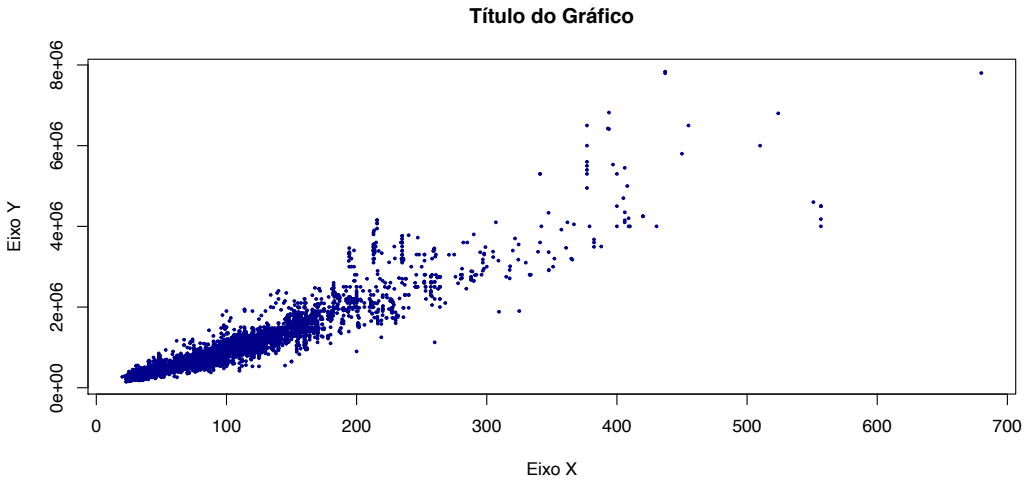


Personalizando parâmetros

Para ver os parâmetros disponíveis, digite `?par`:

```
with(venda, plot(m2, preco,
                 col = "darkblue",
                 main = "Título do Gráfico",
                 xlab = "Eixo X",
                 ylab = "Eixo Y",
                 pch = 20,
                 cex = 0.5))
```

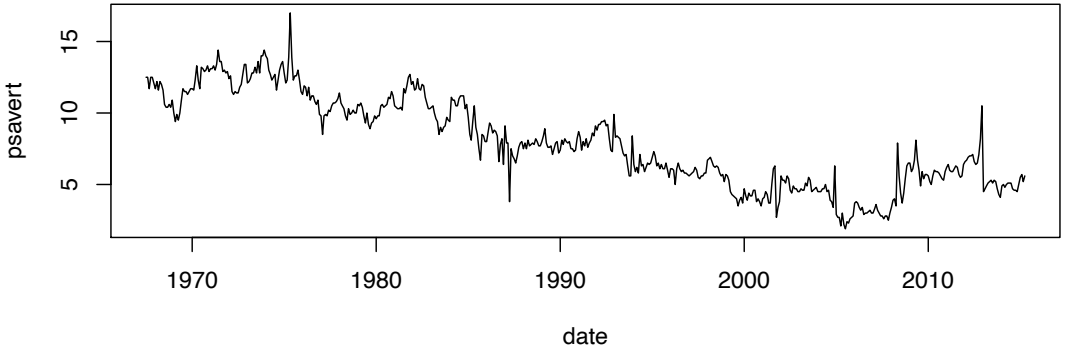
Personalizando parâmetros



Line plot

Basta mudar `type="l"`. Note que o gráfico pode ter notação em formula também:

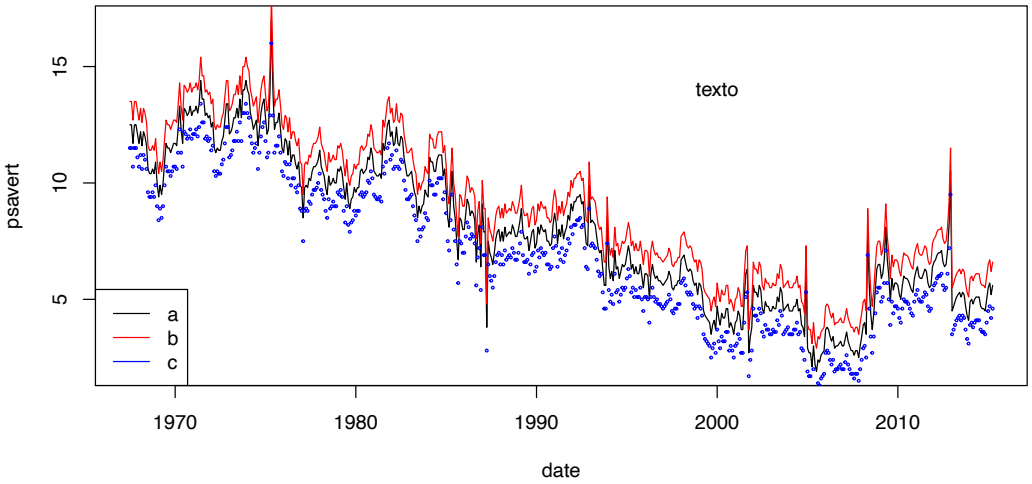
```
library(ggplot2) # para a base de dados economics
plot(psavert ~ date, type = "l", data = economics)
```



Adicionando linhas, pontos, textos e legenda

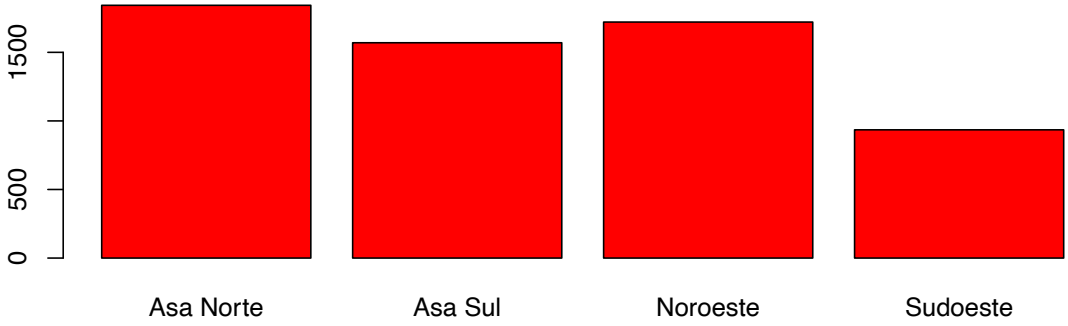
```
lines(economics$date, economics$psavert + 1, col = "red")

points(economics$date, economics$psavert - 1, col = "blue", cex = 0.3)

text(x = as.Date("2000-01-01"), y = 14, "texto")

legend("bottomleft", col=c("black", "red", "blue"),
       legend = c("a", "b", "c"), lty = 1)
```

Adicionando linhas, pontos, textos e legenda



Barplot

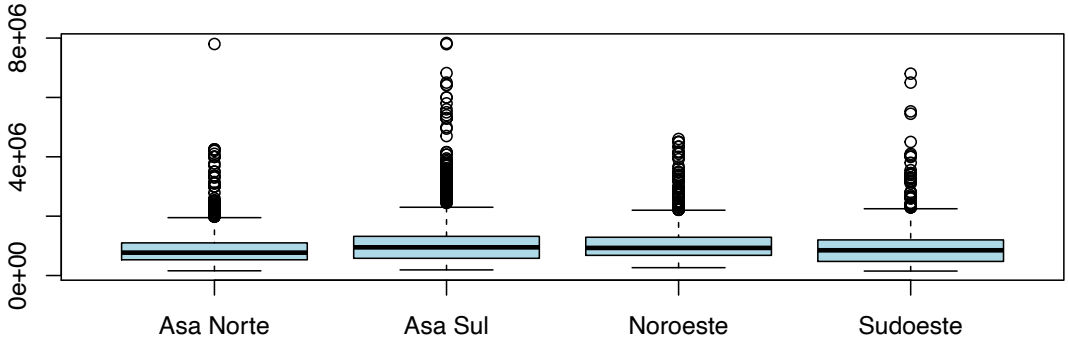
```
barplot(table(venda$bairro), col = "red")
```



Boxplot

Função `boxplot()`:

```
boxplot(preco ~ bairro, data = venda, col = "lightblue")
```



Histograma

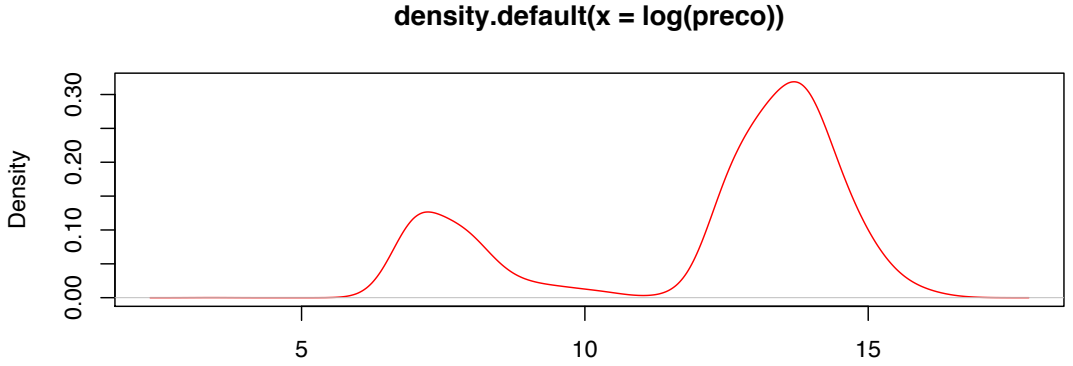
Função `hist()`:

```
histograma <- with(limpos, hist(log(preco), col = "lightblue"))
```

Densidade

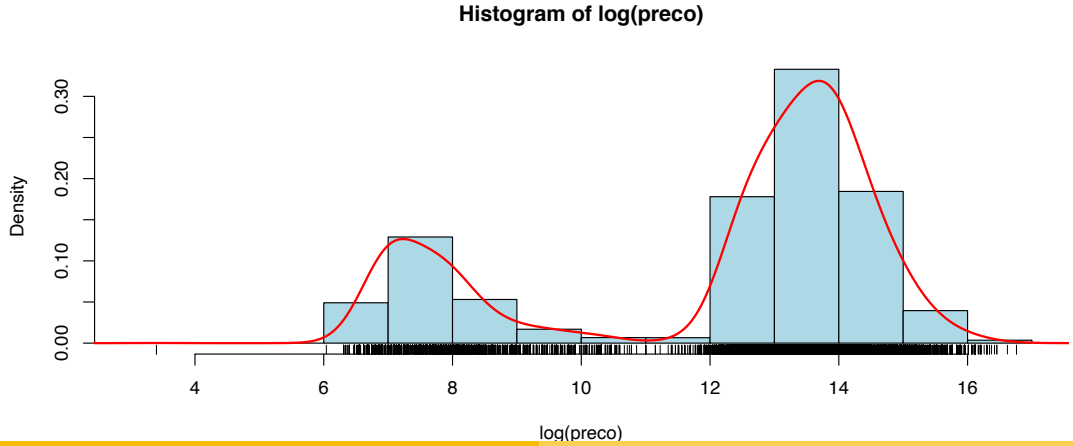
Calculando e plotando a densidade:

```
densidade <- with(limpos, density(log(preco)))
plot(densidade, col = "red")
```



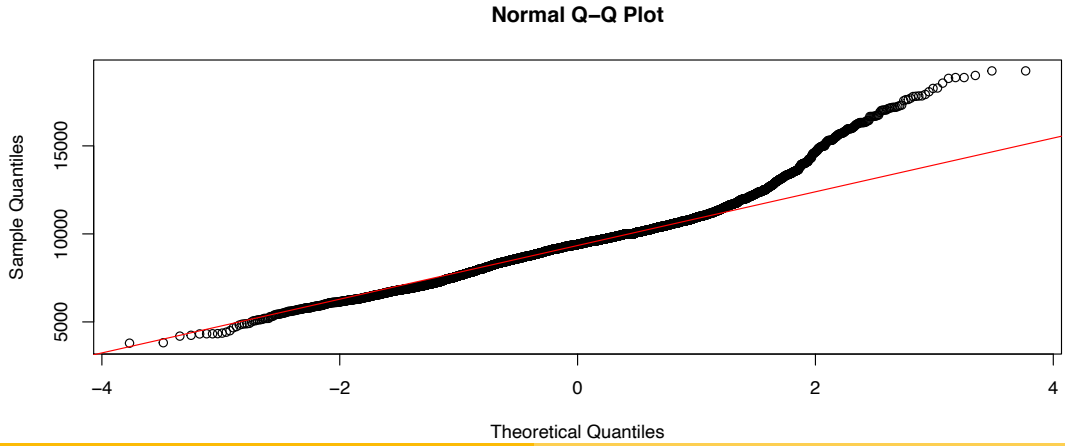
Histograma mais densidade e rug

```
plot(histograma, col = "lightblue", freq = FALSE)
lines(densidade, col = "red", lwd = 2); rug(log(limpos$preco))
```



Quantile plots

```
qqnorm(venda$pm2)
qqline(venda$pm2, col="red")
```



Vários gráficos ao mesmo tempo

Para plotar vários gráficos ao mesmo tempo, utilizar `par(mfrow=c(n,m))` ou `par(mfcol=c(n,m))`

ggplot2: Uma gramática de gráficos

Utilizando gráficos para explorar sua base de dados

Os gráficos base do R são bastante poderosos e com eles é possível fazer muita coisa. Entretanto, eles podem ser um pouco demorados para explorar dinamicamente sua base de dados. O pacote `ggplot2` é uma alternativa atraente para resolver este problema. O `ggplot2` é um pouco diferente de outros pacotes gráficos pois não segue a lógica de desenhar elementos na tela; ao invés disso, a sintaxe do `ggplot2` segue uma "gramática de gráficos estatísticos" baseada no Grammar of Graphics de Wilkinson (2005).

Utilizando gráficos para explorar sua base de dados

No começo, pode parecer um pouco diferente essa forma de construir gráficos. Todavia, uma aprendidos os conceitos básicos da gramática, você vai pensar em gráficos da mesma forma que pensa numa análise de dados, construindo seu gráfico iterativamente, com visualizações que ajudem a revelar padrões e informações interessantes gastando poucas linhas de código. É um investimento que vale a pena.

Utilizando gráficos para explorar sua base de dados

Antes de continuar, você precisa instalar e carregar os pacotes que vamos utilizar nesta seção. Além do próprio `ggplot2`, vamos utilizar também os pacotes `ggthemes` e `gridExtra`.

```
# Instalando os pacotes (caso não os tenha instalados)
install.packages(c("ggplot2", "ggthemes", "gridExtra"))

# Carregando os pacotes
library(ggplot2)
library(ggthemes)
library(gridExtra)
```

A "gramática dos gráficos"

Mas o que seria essa gramática de gráficos estatísticos? Podemos dizer que um gráfico estatístico é um **mapeamento** dos dados para propriedades **estéticas** (cor, forma, tamanho) e **geométricas** (pontos, linhas, barras) da tela. O gráfico também pode conter **transformações estatísticas** e múltiplas **facetas** para diferentes subconjuntos dos dados. É a combinação de todas essas **camadas** que forma seu gráfico estatístico. Deste modo, os gráficos no `ggplot2` são construídos por meio da **adição de camadas**. Cada gráfico, *grosso modo*, é composto de:

- Uma base de dados (um data.frame, preferencialmente no formato **long**);
- Atributos estéticos (**aes**thetics);
- Camadas, contendo
  - Objetos **geom**étricos;
  - Transformações e**stat**ísticas;
- **Facet**as; e,
- Demais ajustes.

aes: x e y - geom_point()

Vejamos um exemplo simples de **diagrama de dispersão** com os dados de preço e metro quadrado dos imóveis da nossa base de dados.

```
ggplot(data = venda, aes(x = m2, y = preco)) + geom_point()
```

Traduzindo o comando acima do ggplot2, nós começamos chamando a função `ggplot()` que inicializa o gráfico com os seguintes parâmetros:

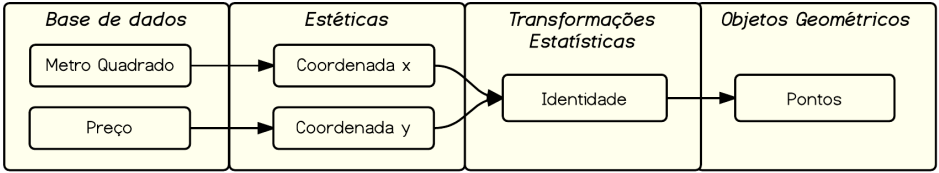
- **data**: aqui indicamos que estamos usando a base de dados `venda`;
- **aes**: aqui indicamos as **estéticas** que estamos mapeando. Mais especificamente, estamos dizendo que vamos mapear o eixo x na variável `m2` e o eixo y na variável `preco`.

Em seguida, adicionamos um objeto geométrico:

- **geom_point()**: estamos falando ao `ggplot` que queremos adicionar o ponto como objeto geométrico.

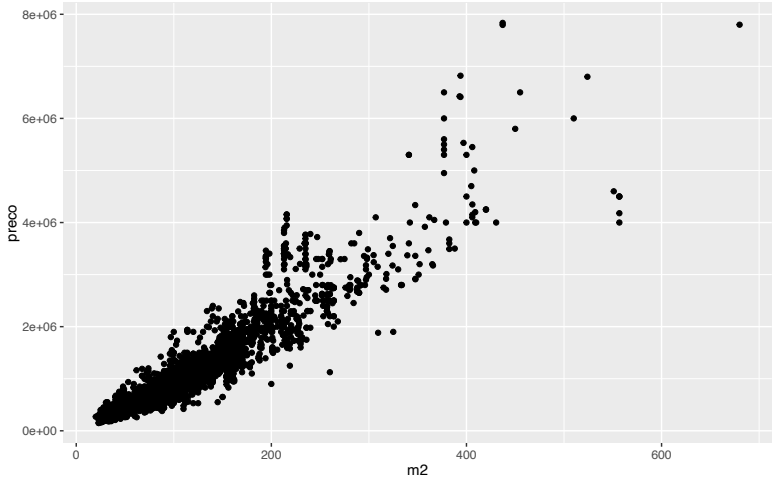
aes: x e y - geom_point()

Com relação às transformações estatísticas, neste caso não estamos realizando nenhuma. Isto é, estamos plotando os dado sem quaisquer modificações. Em termos esquemáticos, nós estamos fazendo o seguinte mapeamento:



aes: x e y - geom_point()

Como resultado, temos:



Outros geoms: pontos, retas, boxplots, regressões

Vimos como exemplo o `geom_poin()`, mas o `ggplot2` vem com vários **geoms** diferentes e abaixo listamos os mais utilizados:

Tipo de Gráfico	**geom**
scatterplot (gráfico de dispersão)	`geom_point()`
barchart (gráfico de barras)	`geom_bar()`
boxplot	`geom_boxplot()`
line chart (gráfico de linhas)	`geom_line()`
histogram (histograma)	`geom_histogram()`
density (densidade)	`geom_density()`
smooth (aplica modelo estatístico)	`geom_smooth()`

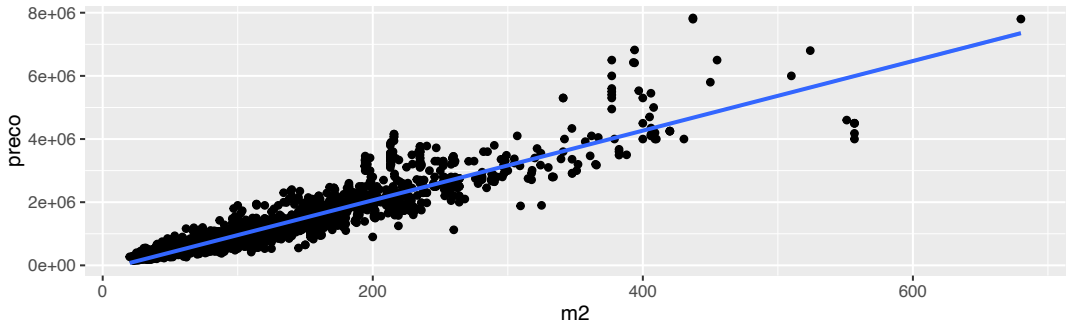
Outros geoms - experimente

```
# Scatter plot
ggplot(data = venda, aes(x = m2, y = preco)) + geom_point()

# Line plot
ggplot(data = venda, aes(x = m2, y = preco)) + geom_line()

# Histogram
ggplot(data = venda, aes(x = preco)) + geom_histogram()

# Density
ggplot(data = venda, aes(x = preco)) + geom_density()

# Boxplot
ggplot(data = venda, aes(x = bairro, y = preco)) + geom_boxplot()

# Smoother (lm, loess, gam)
ggplot(data = venda, aes(x = m2, y = preco)) + geom_smooth(method = "lm")
```

Combinando geoms

Os geoms podem sem combinados:

```
# pontos mais reta de regressão
ggplot(data = venda, aes(x = m2, y = preco)) + geom_point() +
  geom_smooth(method = "lm")
```



aes: mapeando cor, tamanho, forma etc

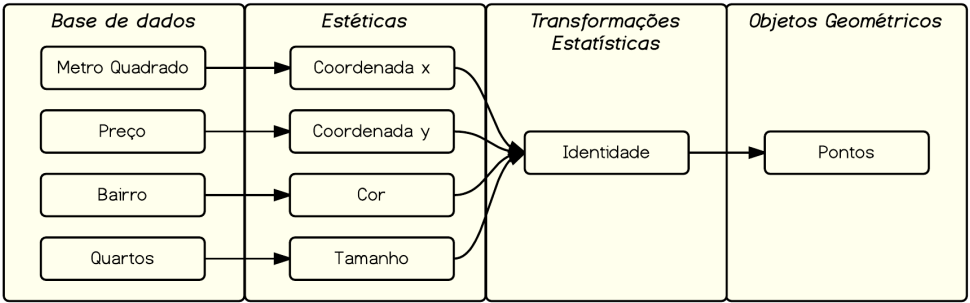
Um gráfico no plano tem apenas duas coordenadas, `x` e `y`, mas nossa base de dados tem, em geral, vários colunas... como podemos representá-las? Uma forma de fazer isso é mapear variáveis em outras propriedades estéticas do gráfico, tais como **cor**, **tamanho** e **forma**. Isto é, vamos expandir as variáveis que estamos meapeando nos **aes**thetics.

aes: mapeando cor, tamanho, forma etc

Para exemplificar, vamos mapear cada bairro em uma cor diferente e o número de quartos no tamanho dos pontos.

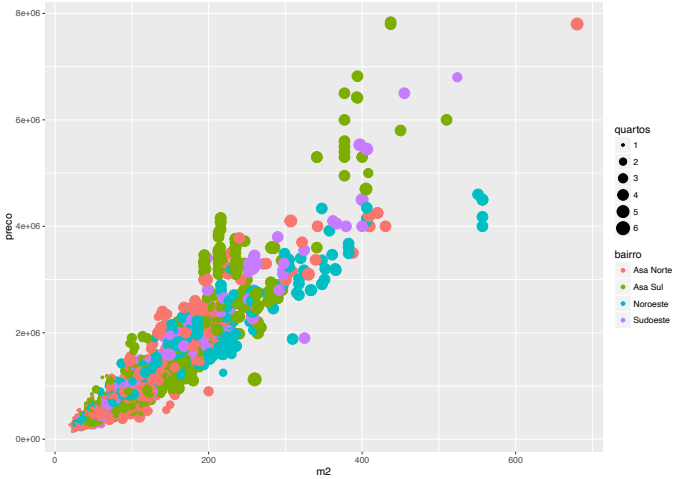
```
ggplot(data = venda, aes(x = m2, y = preco, color = bairro, size = quartos)) +
  geom_point()
```

Nosso esquema ficaria da seguinte forma.



aes: mapeando cor, tamanho, forma etc

E o gráfico resultante:



aes: mapear é diferente de determinar!

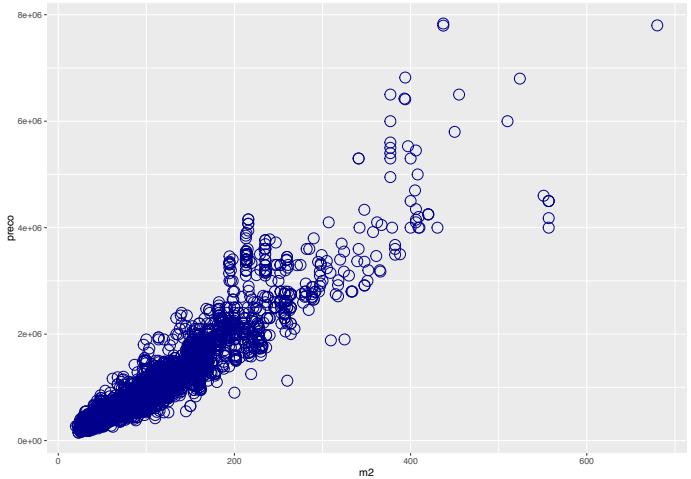
Uma dúvida bastante comum quando as pessoas começam a aprender o `ggplot2` é a diferença entre mapear variáveis em certo atributo estético e determinar certo atributo estético. Quando estamos mapeando variáveis, fazemos isso **dentro** do comando `aes()`. Quando estamos apenas mudando a estética do gráfico, sem vincular isso a alguma variável, fazemos isso **fora** do comando `aes()`.

aes: mapear é diferente de determinar!

Por exemplo, no comando abaixo mudamos a cor, o tamanho e a forma dos pontos do scatter plot. Entretanto, essas mudanças foram apenas cosméticas e não representam informações de variáveis da base de dados e, portanto, não possuem legenda.

```
# muda o tamanho, a cor e a forma dos pontos
# note que não há legenda, pois não estamos
# mapeando os dados a atributos estéticos
ggplot(data = venda, aes(x = m2, y = preco)) +
  geom_point(color = "darkblue", shape = 21, size = 5)
```

aes: mapear é diferente de determinar!



Combinando aes e geom

Os gráficos do `ggplot2` são construídos em etapas e podemos combinar uma série de camadas compostas de **aes** e **geoms** diferentes, adicionando informações ao gráfico iterativamente. Toda informação que você passa dentro do comando inicial `ggplot()` é repassada para os `geoms()` seguintes. Assim, as estéticas que você mapeia dentro do comando `ggplot()` valem para todas as comadas subsequentes; por outro lado, as estéticas que você mapeia dentro dos **geoms** valem apenas para aquele **geom** especificamente. Vejamos um exemplo.

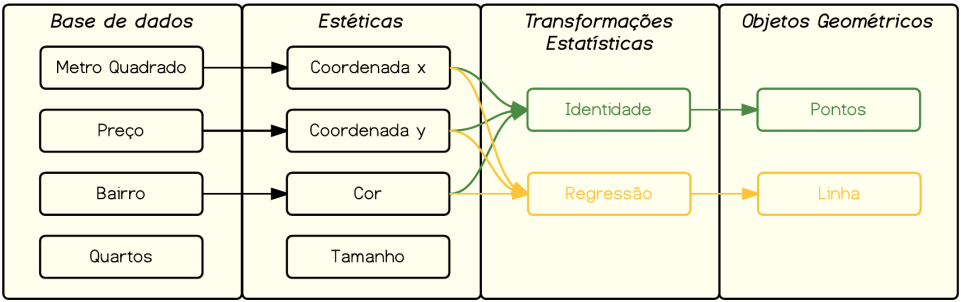
Combinando aes e geom

O comando abaixo mapeia o bairro como cor dentro do comando `ggplot()`. Dessa forma, tanto nos pontos `geom_point()`, quanto nas regressões `geom_smooth()` temos cores mapeando bairros, resultando em várias regressões diferentes.

```
# aes(color) compartilhado
ggplot(venda, aes(m2, preco, color = bairro)) +
  geom_point() +
  geom_smooth(method = "lm")
```

Combinando aes e geom

Vejamos no esquema:



Combinando aes e geom

Resultado:

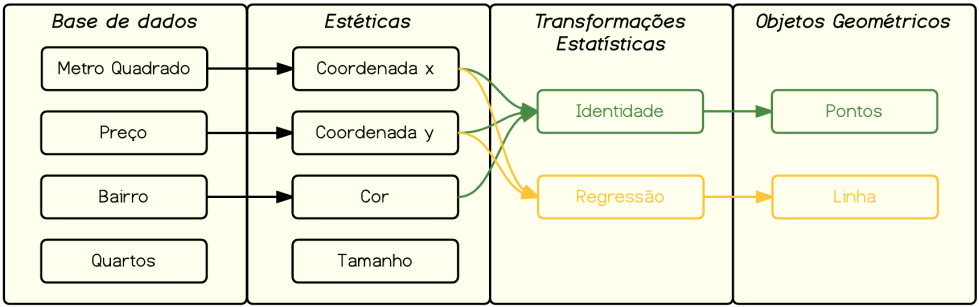
Combinando aes e geom

Mas e se você quisesse manter os pontos com cores diferentes com apenas uma regressão para todas observações? Neste caso, temos que mapear os bairros nas cores **apenas** para os pontos. Note que no comando a seguir passamos a estética `color = bairro` apenas para `geom_poin()`.

```
# aes(color) apenas nos pontos
ggplot(venda, aes(m2, preco)) +
  geom_point(aes(color = bairro)) +
  geom_smooth(method = "lm")
```

Combinando aes e geom

Vejamos no esquema:



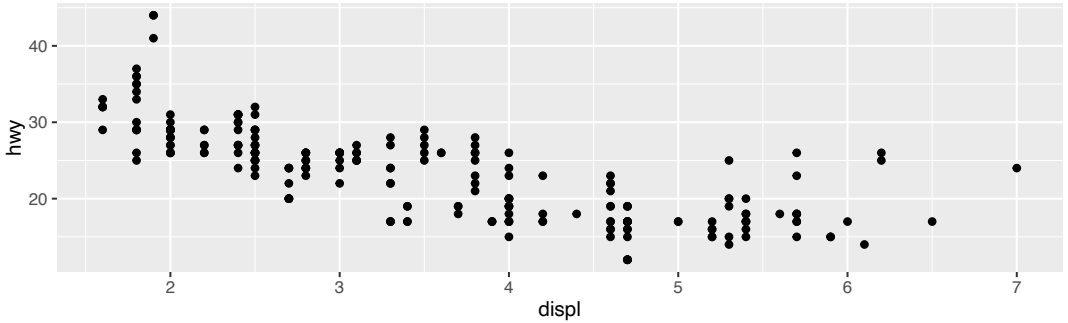
Combinando aes e geom

Resultado:

Cilindradas, cilindros e Milhas por Galão

Um exemplo legal de como um gráfico pode revelar informações. Gráfico simples:

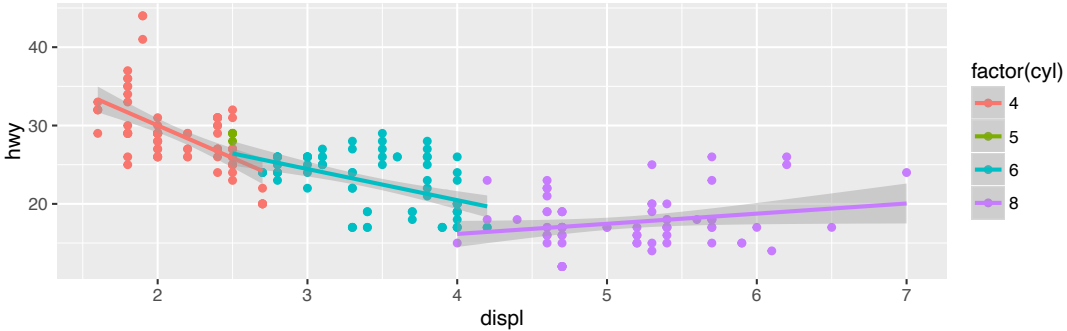
```
ggplot(mpg, aes(displ, hwy)) + geom_point()
```



Cilindradas, cilindros e Milhas por Galão

Gráfico com cor e regressão por cilindro:

```
ggplot(mpg, aes(displ, hwy, col = factor(cyl))) + geom_point() +
  geom_smooth(method = "lm")
```



Adicionando facetas

Você pode dividir o gráfico por variáveis categóricas, usando o `facet_wrap()`.

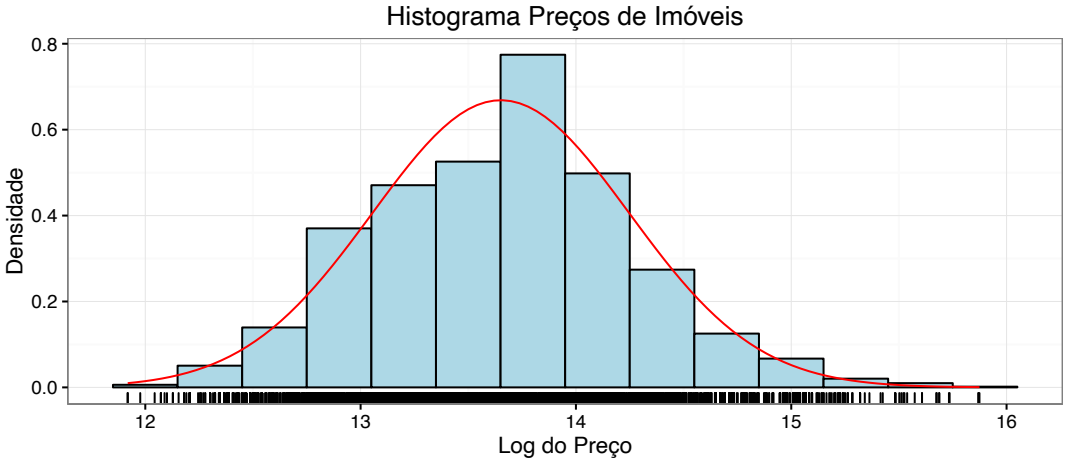
```
ggplot(venda, aes(m2, preco)) +
  geom_point(aes(col = factor(quartos))) +
  geom_smooth(method = "lm") +
  facet_wrap(~bairro)
```

Personalizando mais o gráfico

Colocando *labels*, *títulos*, e mudando o *fundo* para branco:

```
media <- mean(log(venda$preco))
dp <- sd(log(venda$preco))
ggplot(data = venda, aes(x = log(preco))) +
  geom_histogram(aes(y = ..density..), binwidth = 0.3,
                 fill = "lightblue", col = "black") +
  geom_rug() +
  stat_function(fun = dnorm, args = list(mean = media, sd = dp),
                color = "red") +
  xlab("Log do Preço") +
  ylab("Densidade") +
  ggtitle("Histograma Preços de Imóveis") +
  theme_bw()
```

Personalizando mais o gráfico

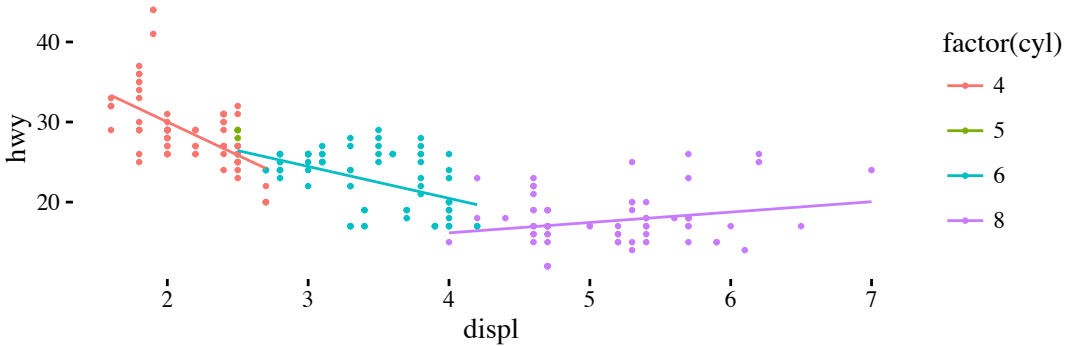


Temas pré-prontos

O pacote `ggthemes` já vem com vários temas pré-programados, replicando formatações de sites como The Economist, The Wall Street Journal, FiveThirtyEight, ou de outros aplicativos como o Stata, Excel entre outros. Esta é uma forma rápida e fácil de adicionar um estilo diferente ao seu gráfico. Experimente com os temas!

Temas pré-prontos

```
library(ggthemes)
ggplot(mpg, aes(displ, hwy, col = factor(cyl))) +
  geom_point() +
  geom_smooth(method = "lm", se = F) +
  theme_tufte()
```



Exercícios

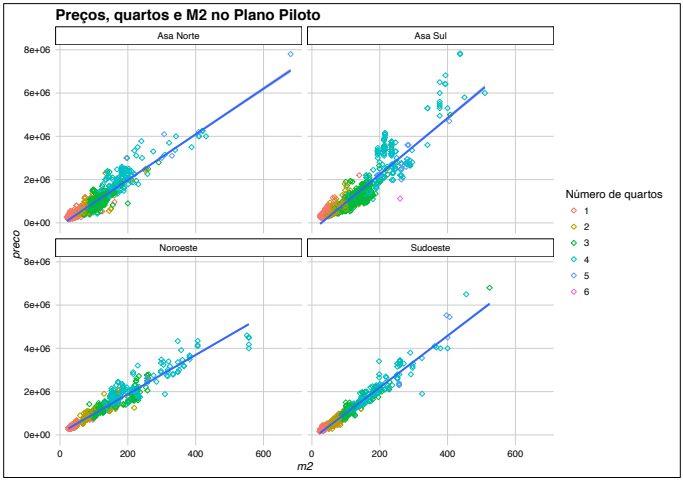
Sua vez.

- Faça um gráfico similar ao apresentado no primeiro dia de aula: "gráfico de dispersão de preço contra metro quadrado por bairro, cor dos pontos de acordo com número de quartos e linha de regressão".
- Adicione um tema do `ggthemes` a seu gráfico.

Soluções

```
ggplot(venda, aes(m2, preco)) +
  geom_point(shape = 5, aes(color = factor(quartos))) +
  geom_smooth(method = "lm") +
  facet_wrap(~bairro) +
  ggtitle("Preços, quartos e M2 no Plano Piloto") +
  scale_color_discrete("Número de quartos") +
  theme_gdocs()
```

Soluções


Preços, quartos e M2 no Plano Piloto
Asa Norte
Asa Sul
Noroeste
Sudoeste
preco
m2
0e+00
2e+06
4e+06
6e+06
8e+06
0
200
400
600
Número de quartos
1
2
3
4
5
6

Programação em R

Julho, 2016

Debugando seu código

Bugs sempre vão existir

Um *bug* é um comportamento não esperado no seu programa.

- Errar é humano, sempre haverá algum tipo de erro (ou erro potencial) em seu código.

Temos dois tipos erro:

- Erro de sitaxe:
  - Ex: esquecer uma vírgula ou aspas.
  - Você recebe uma mensagem de erro.
- Erro semântico:
  - Executar um comando **válido** mas que não faz o que você acha que faz.
  - Você **não** recebe uma mensagem de erro.

Antes de tudo!

Sempre leia as mensagens de warning e de erro!

- Sim, no começo você não vai entender nada. ***Mas isso é normal!*** Nunca deixe de ler a mensagem de erro e de tentar entender o que o R está te dizendo. Leia com calma, uma, duas vezes. Busque no Google. Com o passar do tempo **você vai** entender as mensagens de erro.

Lendo uma mensagem de erro: alguns erros comuns

Erro simples:

```
"1" + "2"
## Error in "1" + "2": argumento não-numérico para operador binário
```

Note que o R te fala onde aconteceu o erro. `Error in "1" + "2"`!

Lendo uma mensagem de erro: alguns erros comuns

Selecionando dimensões que não existem:

```
x <- 1:10
x[,2]
## Error in x[, 2]: número incorreto de dimensões
```

Lendo uma mensagem de erro: alguns erros comuns

Selecionando uma coluna que não existe em uma matriz.

```
m <- matrix(x, ncol = 2)
m
##      [,1] [,2]
## [1,]    1    6
## [2,]    2    7
## [3,]    3    8
## [4,]    4    9
## [5,]    5   10
m[,3]
## Error in m[, 3]: índice fora de limites
```

Lendo uma mensagem de erro: alguns erros comuns

Selecionando uma coluna que não existe no `data.frame`. Se selecionarmos como matriz dá erro:

```
df <- data.frame(x = 1, y = 2)
df[,"z"]
## Error in `[.data.frame`(df, , "z"): undefined columns selected
```

Todavia, note que com `$` ou `[[ ]]` o R retorna `NULL`, é um erro "semântico". Cuidado:

```
df$z
## NULL
df[["z"]]
## NULL
```

Lendo uma mensagem de erro: alguns erros comuns

Argumentos não existentes:

```
f <- function(x) x^2
f(y = 1:10)
## Error in f(y = 1:10): unused argument (y = 1:10)
```

Foram vários erros?

Ocorreram vários erros? Sempre vá no primeiro erro!
Rode o código abaixo. Aparecerão três erros, mas os dois últimos são apenas consequência do primeiro.

```
x <- 1:10
y <- sum(x
z <- x + y
w <- z^2
h <- w + y
```

Erros em funções: recuperando a sequência de chamadas

Vamos definir quatro funções diferentes, em que uma chama a outra. Defina essas funções em um arquivo separado e dê *source*.

```
f1 <- function(x) x + "1"
f2 <- function(x) f1(x) + 10
f3 <- function(x) f2(x)*2
f4 <- function(x) f3(x)^2
```

Erros em funções: recuperando a sequência de chamadas

Agora vamos rodar o seguinte código:

```
f4(1)
```

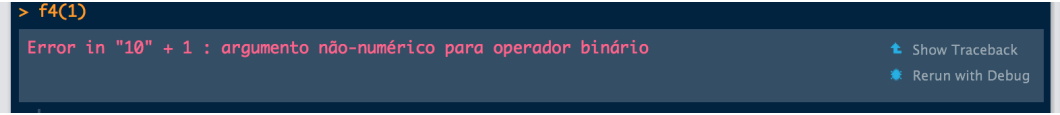


Figure 1: debug

Note que o RStudio te dá a opção de mostrar o `traceback()` e de rodar o código novamente com `debug()` (que veremos a seguir).

Erros em funções: recuperando a sequência de chamadas

O `traceback()` te fornece a sequência de chamadas de funções até o momento em que o erro ocorreu.

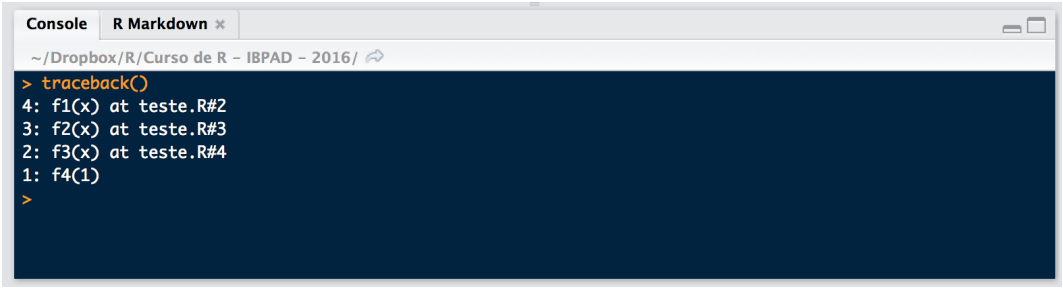


Figure 2: debug

Neste exemplo, agora sabemos que o erro foi na função `f1()` e, portanto, podemos começar nosso trabalho de debug investigando por que a função `f1()` está dando erro.

Executando passo a passo: `browser()` e `debug()`

Temos duas funções importantes para debugar seu código linha a linha: `browser()` e `debug()`.

- A função `browser()`, quando inserida dentro de uma função (ou loop), faz com que a execução pare naquele ponto e permite que você acompanhe o que está ocorrendo passo a passo a partir dali. Você pode continuar a execução da função passo usando `n`, continuar toda a seção do código utilizando `c` ou sair do modo debug utilizando `Q`.
- A função `debug()` tem a mesma lógica do `browser()`, mas ao invés de parar em um ponto específico ela para a execução logo no início do código. Com o comando `debug(mean)`, por exemplo, toda hora que você executar a função `mean()` você executará passo a passo a função. Para parar de debugar, use `undebug()`. Para debugar apenas uma vez, `debugonce()`.

Executando passo a passo: `browser()` e `debug()`

O código abaixo vai dar erro:

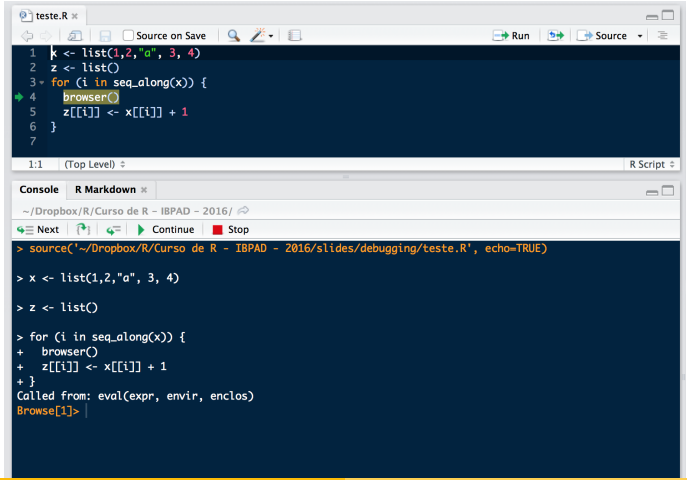
```
x <- list(1,2,"a", 3, 4)
z <- list()
for (i in seq_along(x)) {
  z[[i]] <- x[[i]] + 1
}
## Error in x[[i]] + 1: argumento não-numérico para operador binário
```

Agora, em um arquivo seperado, escreva o código abaixo com `browser()` no meio do loop e faça o *source* do arquivo.

```
x <- list(1,2,"a", 3, 4)
z <- list()
for (i in seq_along(x)) {
  browser()
  z[[i]] <- x[[i]] + 1
}
```

Executando passo a passo: `browser()` e `debug()`

No RStudio, você terá a seguinte tela. Paremos agora para ver no RStudio como funciona cada um desses comandos:



Executando passo a passo: `browser()` e `debug()`

Vamos executar a função `lm()` passo a passo usando `debug()`. Primeiro, fale ao `R` que você quer debugar a função `lm()`.

```
debug(lm)
```

Agora chame o código que queremos executar:

```
modelo <- lm(mpg ~ cyl, data = mtcars)
```

Após debugar a função, lembre de chamar `undebug()`.

```
undebug(lm)
```

Continuando a execução mesmo com erros: `try()`.

Em algumas situações queremos continuar executando um código mesmo que ocorra um erro.

- Por exemplo, em um loop de webscraping, muitas vezes algumas páginas podem conter problemas, mas não queremos que o loop seja interrompido. Queremos que o loop continue e eventualmente trataremos os problemas em outro momento.

Para estes casos, você pode utilizar a função `try()`.

Continuando a execução mesmo com erros: `try()`.

```
x <- list(1,"a", 2)
z <- list()
for (i in seq_along(x)) {
  z[[i]] <- try(x[[i]] + 1)
}
z
## [[1]]
## [1] 2
##
## [[2]]
## [1] "Error in x[[i]] + 1 : argumento não-numérico para operador binário\n"
## attr(,"class")
## [1] "try-error"
## attr(,"condition")
## <simpleError in x[[i]] + 1: argumento não-numérico para operador binário>
##
## [[3]]
## [1] 3
```

Programação em R - Exercícios - Lista 1

Carlos Cinelli

Julho, 2016

1. Crie três variáveis no R com as três formas diferentes de criar objetos que você aprendeu:

- Verifique se as variáveis foram criadas. Crie uma quarta variável concatenando as três variáveis.
- Remova apenas as três primeiras variáveis do ambiente de trabalho com apenas um comando.

2. Crie variáveis de classes `numeric`, `integer`, `complex`, `character` e `logical` com tamanhos diferentes (maiores do que 1):

- Verifique se as variáveis estão no ambiente de trabalho. Veja a estrutura, classe e tamanho dessas variáveis.
- Aplique as funções `is.xxxx` e as funções `as.xxxx` aos objetos e verifique seu comportamento.
- Crie um vetor numérico e e dê nome a cada um dos elementos como "obs1", "obs2", ... utilizando a função `names()`. Utilize a função `paste()` para criar os nomes (olhe a ajuda em `?paste`). Faça alguns subsets do vetor usando os nomes.
- Salve a variável numérica como `x.rds` usando a função `saveRDS()` e como `x.rda` usando a função `save()`. Tente ler a ajuda das funções para entender como funcionam (`?saveRDS` e `?save`). A seção de exemplo pode ser útil. Não se preocupe, aprenderemos essas funções com mais detalhes em outra aula, o objetivo é exercitar o uso da ajuda do R.
- Remova todas variáveis do ambiente de trabalho com apenas um comando.
- Carregue novamente a variável numérica usando a função `load()` no arquivo `x.rda`. Agora carregue usando a função `readRDS()` no arquivo `x.rds`. Você notou alguma diferença entre essas duas formas de carregar a variável?

3. Verifique as classes, estruturas, tamanho e o resultado da função `summary()` nos seguintes vetores:

- `x1 <- 1:10`
- `x2 <- rnorm(10)`
- `x3 <- c("a", "b", "c")`
- `x4 <- c(T, F, T)`
- `x5 <- c(1, "a", 2)`
- `x6 <- c(1, TRUE, 0)`
- `x7 <- c(1, 2, 3, 4L)`
- `x8 <- c(1L, 2, TRUE, "a")`

4. Construa os seguintes vetores com 100 observações de uma normal(0,1):

```
set.seed(1)
x <- rnorm(100)
y <- rnorm(100)
```

- Qual é o primeiro elemento de x? Qual o terceiro elemento de y?
- Delete os 10 primeiros elementos de x e de y.
- Qual é 56th elemento de x? Ele é maior do que 56th elemento de y?
- Qual é a média de x? Qual a diferença da média de x com a média de y?
- Qual é o desvio-padrão de x? Qual a variância de y?
- Qual é a correlação de x e y?
- Qual a soma, produtório, soma acumulada e produtório acumulado de x?
- Como você utilizaria a função `cumsum()` para calcular a média acumulada de x?
- Crie os vetores `x_cres` e `x_decres` com x ordenado de formas crescente e decrescente.
- Crie um vetor z concatenando x e y.
- Selecione apenas os valores de y maiores do que sua média.

- Selecione os valores de x maiores ou iguais a 0.5, ou menores do que -1.5.
- Selecione os valores de x maiores do que 0.5 e menores ou iguais a 1.
- Crie novos vetores com os resultados da soma, multiplicação, divisão, parte inteira da divisão, resto da divisão e exponenciação de x com y.
- Verifique se existem NA's ou NaN's nos vetores criados. Caso existam, substitua os NA's ou NaN's por 0.
- Crie um vetor de diferenças entre x sua primeira defasagem. Crie outro com a diferença da terceira defasagem. Crie um com as segundas diferenças do primeiro lag de x (veja o help da função `diff()`). Verifique a estrutura e tamanho dos vetores.
- Selecione apenas cada segundo elemento do vetor x.

5. Considere os vetores `x <- 1:6` e `y <- 2:1`. Qual o tamanho de cada vetor? Qual o resultado de `x + y`? Explique o que aconteceu.

6. Crie os seguintes vetores (**Dica**: use `:`, `seq()`, `rep()` etc):

- 1, 2, 3, 4, . . . , 10;
- 10, 9, 8, 7, ..., 1;
- 2, 4, 6, 8, ..., 200;
- 10.5, 9.5, 8.5, ..., 0;
- de 1.06 até 2.98 com 67 elementos;
- 1, 7, 1, 7, ... de tamanho 140.
- "a", "b", "a", "b", ... de tamanho 10
- 1, 1, 1, ... 100 vezes junto com 5, 5, 5, ... 25 vezes.
- 1, 1, 1, 2, 2, 2, . . . , 10, 10, 10;

7. Crie um vetor de 1 até 100 e o transforme em matrizes quadráticas conforme ilustrado abaixo.

(**Dica**: use as opções `ncol`, `nrow` e `byrow` da função `matrix()`)

Ordenada por colunas:

```
#       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#  [1,]    1   11   21   31   41   51   61   71   81    91
#  [2,]    2   12   22   32   42   52   62   72   82    92
#  [3,]    3   13   23   33   43   53   63   73   83    93
#  [4,]    4   14   24   34   44   54   64   74   84    94
#  [5,]    5   15   25   35   45   55   65   75   85    95
#  [6,]    6   16   26   36   46   56   66   76   86    96
#  [7,]    7   17   27   37   47   57   67   77   87    97
#  [8,]    8   18   28   38   48   58   68   78   88    98
#  [9,]    9   19   29   39   49   59   69   79   89    99
# [10,]   10   20   30   40   50   60   70   80   90   100
```

Ordenada por linhas:

```
#       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
#  [1,]    1    2    3    4    5    6    7    8    9    10
#  [2,]   11   12   13   14   15   16   17   18   19    20
#  [3,]   21   22   23   24   25   26   27   28   29    30
#  [4,]   31   32   33   34   35   36   37   38   39    40
#  [5,]   41   42   43   44   45   46   47   48   49    50
#  [6,]   51   52   53   54   55   56   57   58   59    60
#  [7,]   61   62   63   64   65   66   67   68   69    70
#  [8,]   71   72   73   74   75   76   77   78   79    80
#  [9,]   81   82   83   84   85   86   87   88   89    90
# [10,]   91   92   93   94   95   96   97   98   99   100
```

8. Considere os seguintes vetores gerados de uma distribuição t com 1 e 2 graus de liberdade:

```r
set.seed(10)
x <- rt(10, df = 1)
y <- rt(10, df = 2)
```

- Crie uma matriz xy combinando x e y como linhas. Qual é a estrutura desta matriz?
- Crie um matriz XY combinando x e y como colunas. Qual é a estrutura desta matriz?
- Crie um vetor z concatenando x e y. Crie com z uma matriz com duas linhas. Crie com z uma matriz com duas colunas.
- Transforme x e y em matrizes de 10 linhas e uma coluna. Crie uma matriz `m` com o resultado de x vezes a transposta de y. Notar que a multiplicação é matricial.
- Selecione: (i) a linha 10 de `m`; (ii) a coluna 5 de `m`; (iii) o elemento na terceira linha e segunda coluna; os elementos de `m` maiores do que zero; (iv) os elementos de `m` menores do que zero e calcular a média; (vi) a diagonal de `m`.

9. Crie 2 matrizes, A e B, com 5 linhas e 5 colunas e elementos de uma normal(0,1). Antes de gerar os valores aleatórios, defina a semente `set.seed(1)`. (**Dica:** uma matriz 5 por 5 vai precisar de 25 elementos; a função para gerar dados de uma normal é `rnorm` e a função para criar uma matriz é `matrix`).

- Quais os resultados das funções `length()`, `ncol()`, `nrow()`, `dim()`, `str()`, `min()`, `max()`, `summary()` e `is.matrix` nas matrizes?
- Crie matrizes A2 e B2 compostas dos elementos de A e B em valores absolutos;
- Crie uma matriz C com a soma (elemento a elemento) de A e B;
- Crie uma matriz D com a multiplicação (elemento a elemento) de A e B;
- Inverta as matrizes. Calcule o determinante das matrizes;
- Crie a uma matriz A3 excluindo a primeira linha e a primeira coluna de A;
- Calcule a média dos elementos de A tais que $A[i] \leq B[i]$;
- Crie uma matriz C tal que $C[i] = 1$ se $A[i] \geq B[i]$ e 0 caso contrário;
- Utilize a função `rownames()` para nomear as linhas de A como "linha1", "linha2" ..., e a função `colnames()` para nomear as colunas de A como "coluna1", "coluna2" ... . Utilize a função `paste()` para criar os nomes. Faça alguns subsets utilizando os nomes.

10. Suponha uma lista do tipo

```r
lista <- list( elemento1 = rnorm(10),
               elemento2 = 1L,
               elemento3 = "a",
               elemento4 = c(1,"b"),
               elemento5 = c(TRUE, TRUE, FALSE),
               elemento6 = c(10i, 20i))
```

- Quais os resultados de `names()`, `length()`, `str()`, `dim()`, `summary()` e `is.list()` na `lista`?
- Qual o primeiro elemento da lista? E o terceiro? Quais as formas de selecionar estes elementos?
- Os elementos da lista são de classes diferentes? Quais as classes de cada elemento?
- Adicione um novo elemento na lista, chamado `elemento7`, com o resultado de `elemento1` mais `elemento2`.
- Delete o `elemento1` da lista (**Dica**: atribuir o valor `NULL` a um elemento da lista deleta este elemento. Isso também vale para `data.frames`!).
- Existe uma função no R chamada `unlist` que, como diz o nome, "desfaz" a lista, a transformando em um vetor simples. Qual o objeto resultante de `unlist(lista)`?

11. Listas podem ter estruturas arbitrariamente complexas, com listas dentro de listas. Considere a seguinte `lista2`.

```r
lista2 <- list( lista_dentro_da_lista = lista,
                outra_lista = list(numero=c(1,2,4),
                                   mais_lista=list(c(1,3,4), TRUE)))
```

- Quais os resultados de `names()`, `length()`, `str()` e `summary()` na `lista2`?
- Como acessar o `elemento1` contido em `lista_dentro_da_lista` que está em `lista2`?
- Como acessar o segundo elemento do objeto `mais_lista` dentro de `outra_lista` da `lista2`?

12. Rode o seguinte comando para criar um data.frame:

```
set.seed(14)
df <- data.frame(x = rnorm(10), y = rnorm(10))
```

- Verifique os resutados de `length()`, `nrow()`, `ncol()`, `dim()`, `class()`, `is.data.frame()`,`str()`, `summary()` em `df`. Quantas observações temos na base de dados? Quantas variáveis? Qual a classe do objeto? Quais as classes das variáveis?
- Crie um vetor `z <- rlnorm(10)`. Adicione este vetor como mais uma variável, chamada `z`, em `df`? Quais as diferentes formas de fazer isso?
- Como adicionar mais uma linha com `x=1, y =2, e z=3` ao data.frame?
- Crie uma variável `w` em `df` com o resultado de `x+y+z`.
- Delete a variável `x` de `df` (**Dica**: veja a dica do exercício 10!).
- Delete as últimas cinco linhas de `df`.

13. Carrege a base de dados `wi.venda.rds` em um objeto chamado `dados`. Estes dados são de oferta online de apartamentos no Plano Piloto (Asa Sul, Asa Norte, Sudoeste e Noroeste).

- Verifique os resutados de `length()`, `nrow()`, `ncol()`, `dim()`, `class()`, `is.data.frame()` em `dados`. Quantas observações temos na base de dados? Quantas variáveis? Qual a classe do objeto?
- Verifique os resultados de `names()` e `colnames()`. São os mesmos?
- Aplique `str()`, `summary()` em `dados` para entender melhor quais são os dados do data.frame. Quais as classes de cada coluna?
- Quais foram a média, mediana, desvio-padrão e variância dos precos dos anúncios de apartamento no Plano Piloto neste dia? (faça com `with()`, `$` e `[ ]`)
- Quais foram a média, mediana, desvio-padrão e variância do número de quartos dos anúncios? (faça com `with()`, `$` e `[ ]`)
- Quais foram a média, mediana, desvio-padrão e variância do tamanho dos apartamentos? (faça com `with()`, `$` e `[ ]`)
- Qual a correlação entre preços e metro quadrado? (faça com `with()`, `$` e `[ ]`)
- A coluna `bairro` contém a informação dos bairros dos anúncios. Quantos anúncios temos na Asa Sul (em número e percentual)? Quantos anúncios temos na Asa Norte (em número e percentual)? (faça com `with()`, `$` e `[ ]`)
- Crie um vetor chamado `preco_asa_sul` com os preços da Asa Sul. E outro chamado `preco_asa_norte` com os precos da Asa Norte. Qual dos bairros tem a maior mediana de preços de apartamento? (faça com `with()`, `$` e `[ ]`)
- Crie uma coluna `pm2` em `dados` com o resultado de `preco` dividido por `m2` e repita o exercício anterior (criando os vetores `pm2_asa_sul` e `pm2_asa_norte`). Qual dos dois bairros tem a maior mediana de preços por metro quadrado? (faça com `with()`, `$` e `[ ]`)
- Delete a coluna `link` de `dados`.

14. **Exercício extra**: data.frame é uma lista.

- Rode o comando `is.list()` em dados. Qual o resultado?
- A função `unclass()` retira a classe "especial" dos objetos revertendo-o à sua classe mais básica. Rode o seguinte código abaixo. Note que um data.frame nada mais é do que uma lista com certas características (por exemplo: atributo row.names; cada coluna com o mesmo número de linhas).

Programação em R - Exercícios - Lista 2

Carlos Cinelli

Julho, 2016

1. Considere os seguintes objetos:

```
set.seed(321)

mat <- matrix(rchisq(100, df = 2), ncol = 10)

df <- data.frame(grupo = rep(c("a","b"), 5),
                 col1 = rbinom(10, 1, 0.5),
                 col2 = rcauchy(10))

lista <- lapply(1:10, function(x) matrix(rnorm(100), ncol = 10))
```

- Em `mat`, calcule as somas e as médias, por linhas e por columas, utilizando `apply` e compare o resultado com as funções `rowSums`, `colSums`, `rowMeans` e `colMeans`. **Extra**: em `mat`, calcule a média móvel com janela de 5 observações, por linha e por coluna, utilizando a função `rollapply` do pacote `zoo` (não vimos essa função em aula, olhe a ajuda em `?rollapply`).
- Em `df`, utilize `lapply` somente nas colunas numéricas para calcular a função `sum(x^2)/length(x)`. Faça isso tanto definindo a função fora do apply, quanto definindo a função dentro do próprio `lapply` como uma fução anônima.
- Em `lista` utilize `lapply` para calcular os determinantes, máximo e mínimo (ao mesmo tempo) de cada matriz dentro da lista.

2. Filtrando e aplicando funções em data.frames:

```
set.seed(10)

df <- data.frame(x = rnorm(100),
                 y = sample(letters, 100, replace =  TRUE),
                 z = sample(LETTERS, 100, replace = TRUE),
                 a = rnorm(100))
```

- Crie um vetor chamado `numericas` usando o `sapply()` para descobrir quais colunas do `data.frame` são numéricas. Utilize o vetor `numericas` para selecionar apenas as colunas numéricas do `data.frame` e use novamente o `sapply()` para calcular os desvios-padrão dessas colunas.
- Agora use a mesma lógica da questão anterior para filtrar apenas os vetores do tipo factor e aplicar a função `table()` nessas colunas.
- Repita os exercícios anteriores usando `Filter()` no primeiro passo (isto é, no passo de selecionar apenas certas colunas).

3. Vamos criar nossas próprias funções para aplicar em `data.frame`s. Leia a base de dados `wi.venda.rds` com o comando `dados <- readRDS("Dados/wi.venda.rds")`. Considerando esta base, faça:

- Crie uma função que receba um vetor e calcule diversas estatísticas descritivas, como a média, média aparada, mediana, variância, desvio-padrão, mínimo, máximo, alguns quantis (olhe a função `quantile()`) e outras que você quiser. **Extra**: faça com que a função retorne NA se o vetor não for numérico e emita um warning para avisar o usuário (veja a ajuda da função `warning()`).
- Após criar sua função, use `sapply()` ou `lapply()` em conjunto com `Filter()` para aplicar sua função às colunas numéricas da nossa base de dados.

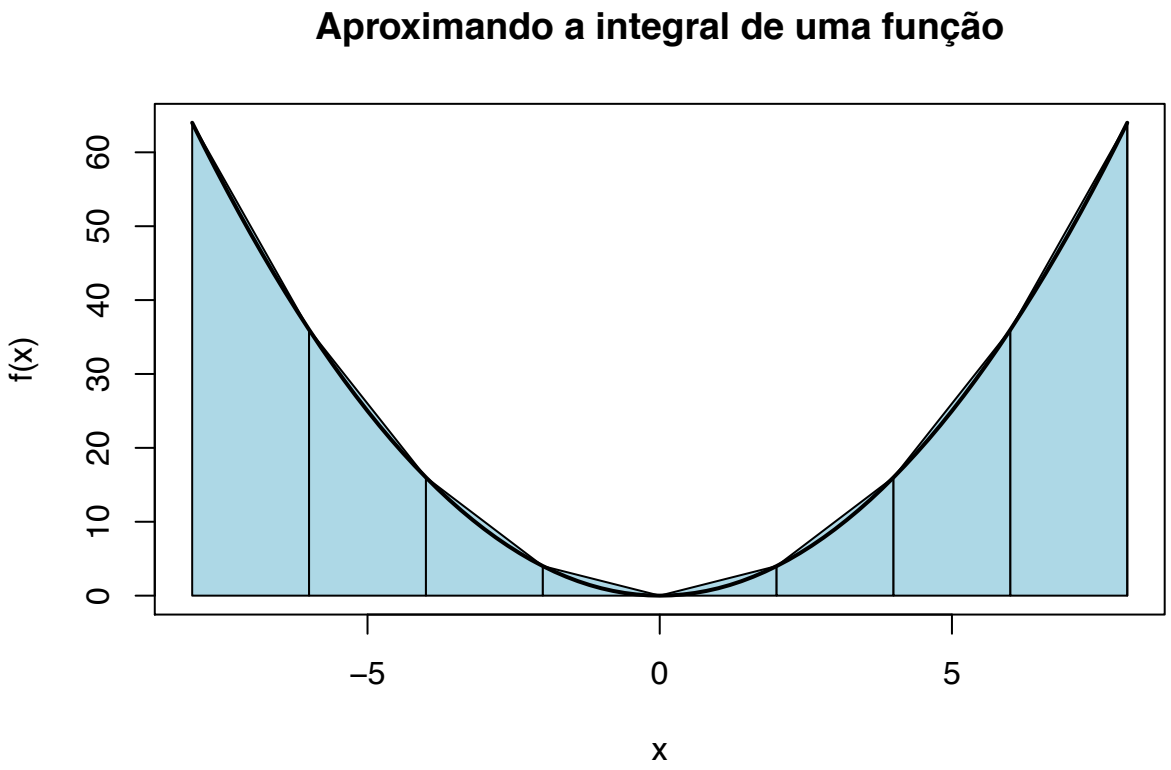
- Agora crie dois `data.frame`s separados `dados_asa_sul` e `dados_asa_norte` com os dados somente da Asa Sul e da Asa Norte, respectivamente. Aplique sua função nesses `data.frame`s e compare os resultados dos dois bairros.

EXTRA - Exercitando a implementação de algoritmos

4. Criando algumas funções:

- Seja n um número inteiro. É possível linearizar o quadrado de n por meio do sequinte algoritmo: $n^2 = 1 + 3 + ... + 2n - 1$. Crie uma função de argumento `n` que gere o quadrado de `n` por meio do algoritmo mencionado. Compare seu resultado com `n^2`. Agora faça com que a função verifique se o argumento `n` é um número redondo e, se não for, faça com que retorne um erro (Dica: veja a ajuda da função `stop()`).
- Crie uma função, usando um loop, que encontre o produtório de um vetor. Compare o resultado de sua função com a função `prod()` do R. **Extra**: agora dê uma olhada na ajuda da função `Reduce()`. Reescreva sua função sem usar loop, usando apenas o `Reduce()`.
- Crie uma função, usando um loop, que encontre o somatório acumulado de um vetor. Compare o resultado de sua função com a função `cumsum()` do R.
- A sequência de fibonacci é definida por: $fib_1 = 0$, $fib_2 = 1$, $fib_3 = 1$, $fib_3 = 2$, $fib_i = fib_{i-1} + fib_{i-2}, \forall i \geq 3$. Crie uma função `fib(n)`, usando um loop, que calcule o enésimo termo da sequência de fibonacci. Faça com que função retorne um erro se o argumento passado não for um inteiro. (Dica: para retornar uma mensagem de erro use a função `stop()`).

5. Aproximando o valor de uma integral: considere uma função qualquer `f(x)`. Suponha que você queira calcular a área embaixo desta função (a integral). É possível aproximar esta área somando pequenos trapézios, como ilustrado na figura a seguir:



Com base nisso, crie uma função no R chamada `integral(a, b, k, f)` que:

- crie uma sequência `x` que vai de a até b em k intervalos equidistantes de tamanho $\frac{b-a}{k}$ (Dica: use a função `seq(a, b, by = (b-a)/k)`).

- crie um vetor `fx` com o resultado da função `f()` em cada ponto da sequencia x criada anteriormente (lembre que as funções do R são vetorizadas!).
- Calcule as áreas dos trapézios embaixo da curva `f()` e que depois some todas as áreas. A área de um trapézio é dada por: $A_i = \frac{f(x_i)+f(x_{i+1})}{2}\frac{(b-a)}{k}$ e a soma das áreas por $A = \sum_i A_i$.
- O resultado da função que acabamos de criar é uma aproximação numérica da integral de $f(x)$ definida no intervalo $x \in (a, b)$, $\int_a^b f(x)dx$. Compare o resultado da sua função coma função `integrate(f, a, b)` do R.

EXTRA - Recursão

Não ensinamos em aula, mas o R também aceita funções recursivas, isto é, funções que podem chamar a si próprias. Um fato importante em definições recursivas é não permitir uma recursão infinita, colocando uma condição de finalização. Por exemplo, a função abaixo seria um implementação recursiva de um fatorial:

```
fatorial <- function(n){
  # condição para evitar recursão infinita
  if(n==1) return(1)
  # recursão
  return(n*fatorial(n-1))
}
fatorial(5) # testando
factorial(5)
```

Com base nisso, tente definir uma função que calcule o enésimo termo de fibonacci usando recursão. Compare o resultado da função recursiva com a função usando *loop*. Qual é mais lenta? O que está acontecendo? Há como resolver isso?

Programação em R - Exercícios - Lista 3

Carlos Cinelli

Julho, 2016

1. Considere os dados em `roubo2.rds`. Esses dados contêm registros online de crimes por todo o Brasil, e foram coletados do site *Onde Fui Roubado?* (http://www.ondefuiroubado.com.br).

- Utilize `sapply` para calcular o número de NA's por colunas.
- Calcule o número de crimes por cidade e ordene de forma decrescente. Onde houve mais crimes registrados? Quantos crimes foram registrados em Brasília?
- Calcule o número de crimes por tipo de crime. Qual o tipo de crime mais frequente? Calcule, por cidade, a frequencia relativa de tipo de crime. Das cidades que tem mais de 100 registros, qual cidade tem a maior proporção de assalto à mão armada?
- Crie uma coluna que indique o dia da semana em que o crime foi cometido. Em que dia da semana foram registrados mais crimes? E considerando somente brasília? Quais os tipos de crimes mais cometidos para cada dia da semana?

2. Considere os dados `dados.rds`.

- O índice de concentração de Herfindahl–Hirschman (HHI) de um mercado é dado pelo somatório do quadrado do market share das empresas. Considere como empresa a corretora. Calcule o HHI para a oferta de imóveis de Brasília, separado por aluguel e venda (elimine os imóveis duplicados). Qual mercado parece ser mais concentrado? Agora calcule o HHI separado por bairro. Qual bairro parece ter um mercado mais concentrado?
- Crie uma série temporal com a mediana dos preços de imóveis, de aluguel e venda, no plano piloto (a coluna `coleta` contém a data em que o preço do imóvel foi coletado, utilize esta coluna). Crie a mesma série separada por tipo de imóvel. Calcule a média e desvio-padrão móvel das séries (veja a ajuda da função `rollapply()` do pacote `zoo`), com uma janela de 15 dias.
- Retire os outliers de `dados` e elimine os duplicados: quais os bairros com o maior aluguel mediano para cada tipo de imóvel? Quais os bairros com a maior mediana de preço por m2 de venda, por tipo de imóvel?
- Usando a base completa, tente encontrar imóveis com as características que você deseja. Utilize os dados da coleta em 2014-08-31 e use a função `grepl()` em algum momento do filtro.

3. Considere os dados de salários de servidores públicos federais `20140131_Remuneracao.csv` e os dados de cadastro `20140131_Cadastro.csv` dentro do arquivo zip `201401_servidores.zip` (descompacte o zip se necessário):

- Leia os arquivos no R usando os comandos ao final da questão (pode ser necessário modificar alguns argumentos do comando).

- Selecione da base de salários apenas as colunas ID do servidor e a remuneração básica bruta.
- Selecione da base de cadastro apenas as colunas de ID do servidor, nome do servidor e órgão de exercício.
- Renomeie as colunas para nomes mais fáceis de usar (e deixe as duas colunas de ID do servidor com o mesmo nome).
- Crie uma base de dados chamada `salcad` com o resultado do merge entre as duas bases acima.
- Com a base criada, responda: qual(is) o(s) órgão(s) com maiores remunerações média, mediana, máxima e mínima? Qual o órgão com maior variabilidade de salários (escolha uma medida de dispersão)?

```
salarios <- read.csv("Dados/20140131_Remuneracao.csv",
                     dec = ",",
                     header = TRUE,
                     fileEncoding = "latin1",
                     sep = "\t",
                     stringsAsFactors = FALSE)

cadastro <- read.csv("Dados/20140131_Cadastro.csv",
                     dec = ",",
                     header = TRUE,
                     fileEncoding = "ASCII",
                     sep = "\t",
                     stringsAsFactors = FALSE)
```