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

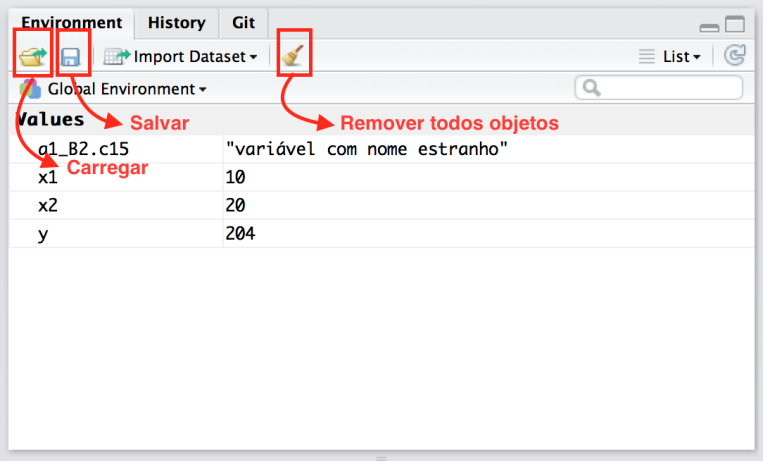
E vamos recuperar todos os objetos que tínhamos criado com `load()`.

```
# 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]
```