Programação em R

Copyright: Carlos Cinelli

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

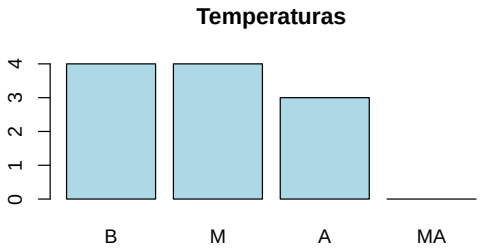
Você pode usar o `table()` para montar tabelas com mais de um fator.

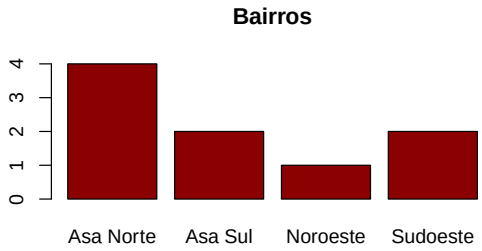
```
grupo <- rep(c("G1", "G2"),length.out=11)
table(grupo, temps)
##       temps
## grupo Alta Baixa Média
##    G1    2     2     2
##    G2    1     2     2
```

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`.