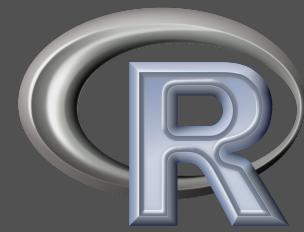




ME115 - Linguagem R

Parte 09

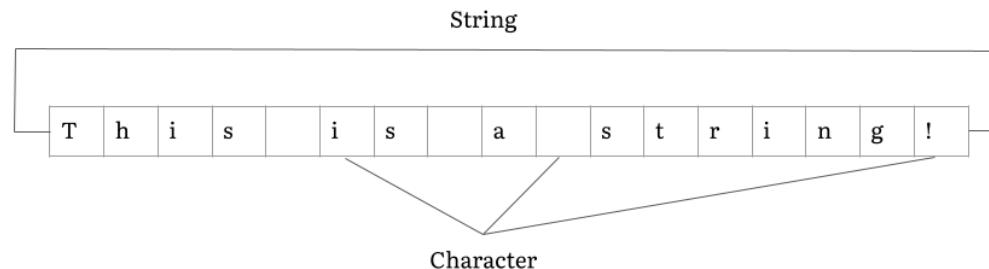
1º semestre de 2023



Strings e Expressões Regulares

Manipulação de Strings

Variáveis de texto, chamadas de *strings*, são muito comuns nos bancos de dados.

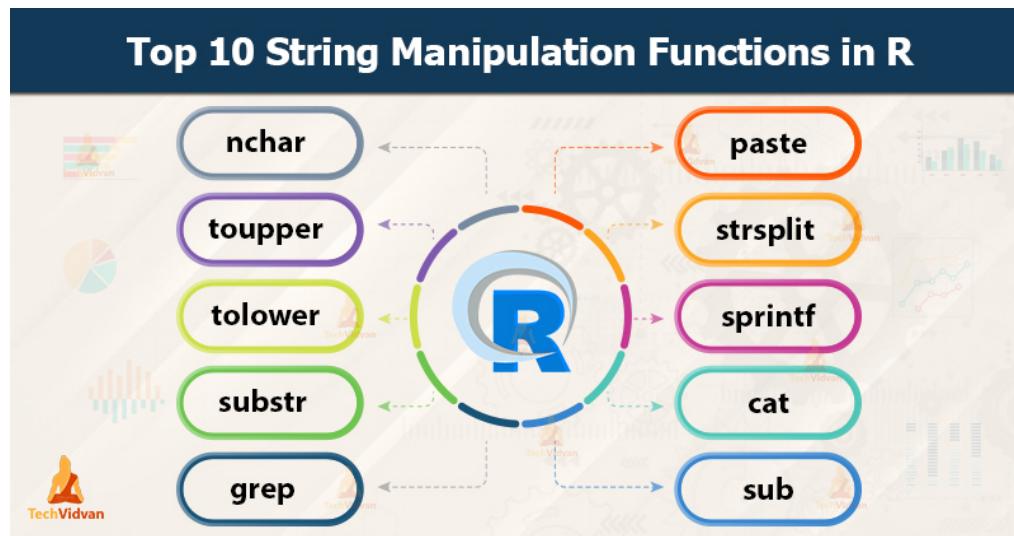


- Geralmente, podem dar bastante trabalho para serem manipuladas.
- Comum encontrarmos categorias não padronizadas. **Exemplo:** uma variável Estado com "SP", "sp", "Sao Paulo", "São Paulo" etc.
- A base do R possui várias funções para manipular textos (*strings*), além de pacotes especializados e mais modernos como o **stringr**.

Fonte da Figura: [https://en.wikipedia.org/wiki/String_\(computer_science\)](https://en.wikipedia.org/wiki/String_(computer_science))

Manipulação de Strings - Base do R

Veja as 10 funções de manipulação de strings mais utilizadas disponíveis na base do R, algumas até já utilizadas nesse curso.



Essas funções são úteis, mas elas não possuem uma interface consistente.

Cada uma tem a sua forma de passar os parâmetros, dificultando a programação durante a análise.

Portanto, focaremos em manipulação de strings usando o pacote **stringr**.

Fonte: <https://techvidvan.com/tutorials/r-string-manipulation/>

O pacote **stringr**

- O pacote **stringr** foi criado com uma linguagem consistente, de forma que o usuário consiga manipular textos com muito mais facilidade.
- Escrito por Hadley Wickham, assim como o **readr**, **tidyR**, **dplyr**, **ggplot2**, entre outros.
- Funções mais rápidas que as da base do R. O pacote foi construídos sobre a biblioteca ICU, implementada em C e C++.
- Fáceis de entender, usar e lembrar.

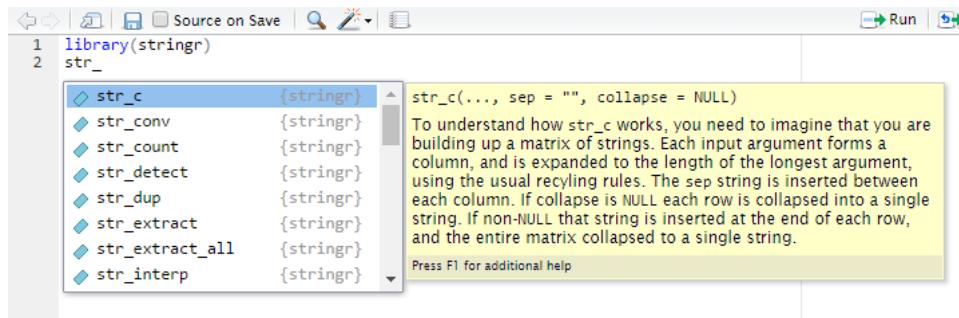
Para carregar o pacote **stringr**, basta instalá-lo e usar a função **library()**:

```
library(stringr)
```



O pacote **stringr**

- Todas as funções desse pacote começam com o prefixo **str_**.
- Então, caso esqueça o nome de uma função, com o pacote já carregado, basta digitar **str_** e apertar TAB para ver quais são as opções, como mostra a figura abaixo:



A screenshot of the RStudio interface showing a code editor window. The code entered is:

```
1 library(stringr)
2 str_
```

The completion dropdown is open, listing several functions starting with 'str_':

- str_c
- str_conv
- str_count
- str_detect
- str_dup
- str_extract
- str_extract_all
- str_interp

A tooltip for 'str_c' is displayed, containing the following text:

str_c(..., sep = "", collapse = NULL)
To understand how str_c works, you need to imagine that you are building up a matrix of strings. Each input argument forms a column, and is expanded to the length of the longest argument, using the usual recycling rules. The sep string is inserted between each column. If collapse is NULL each row is collapsed into a single string. If non-NULL that string is inserted at the end of each row, and the entire matrix collapsed to a single string.

Press F1 for additional help



- O primeiro argumento da função é sempre uma *string* ou um vetor de *strings*.

stringr - Funções Básicas

Vamos começar pelas funções mais simples do **stringr**, que não utilizam o conceito de expressões regulares.

- **str_length()**: retorna o número de caracteres de cada string do vetor
- **str_to_upper()**: converte a string para caixa alta (todas maiúsculas)
- **str_to_lower()**: converte a da string para caixa baixa (todas minúsculas)
- **str_to_title()**: converte a string para título, ou seja, cada palavra é iniciada em maiúscula
- **str_trim()**: remove os espaços excedentes antes e depois da string
- **str_sub()**: usada para obter uma parte fixa da string
- **str_c()**: concatena strings em uma única string



Comprimento de uma string

`str_length()`: retorna o número de caracteres de cada string do vetor.
É equivalente à função `nchar()` da base.

```
nchar("São Paulo")  
[1] 9
```

```
str_length("São Paulo")  
[1] 9
```

Veja que o espaço (" ") é contado como um caractere.

```
estados <- c("São Paulo", "Rio de Janeiro", NA, "Acre")  
str_length(estados)  
[1] 9 14 NA 4
```

```
length(estados)  
[1] 4
```

Caixas de Texto em strings

Modificam as caixas de texto para todas maiúsculas, todas minúsculas e apenas as primeiras letras maiúsculas, respectivamente.



Equivalente às funções `toupper()` e `tolower()` da base.

```
texto <- "Somos alunos de ME115"
str_to_lower(texto)
[1] "somos alunos de me115"

str_to_upper(texto)
[1] "SOMOS ALUNOS DE ME115"

str_to_title(texto)
[1] "Somos Alunos De Me115"
```

Remover Espaços Excedentes

`str_trim()`: remove os espaços excedentes antes e depois da string.

```
genero <- c("M", "F", "F", " M", " F ", "M")
as.factor(genero)
[1] M   F   F   M   F   M
Levels: F   M F M
```

```
genero_fixed <- str_trim(genero)
factor(genero_fixed)
[1] M F F M F M
Levels: F M
```

`str_squish()`: reduz espaços repetidos dentro de uma string.

```
str_squish(" Strings     com    excessos de      espaços em branco. ")
[1] "Strings com excessos de espaços em branco."
```

Obter partes de uma string

`str_sub()`: usada para obter uma parte fixa do vetor de strings.

```
turmas <- c("ME115B-1S2022", "ME315A-2S2021", "ME322A-1S2021")
str_sub(turmas, start = 8) # extrair do 8º caractere em diante
[1] "1S2022" "2S2021" "1S2021"

str_sub(turmas, end = 5) # extrair até o 5º caractere
[1] "ME115" "ME315" "ME322"

str_sub(turmas, start = 6, end = 6) # extrair apenas o 6º caractere
[1] "B" "A" "A"

str_sub(turmas, start = -4) # extrair os últimos 4 caracteres
[1] "2022" "2021" "2021"

str_sub(turmas, end = -8) # remover os 8 últimos caracteres
[1] "ME115B" "ME315A" "ME322A"
```

Combinar duas ou mais strings

`str_c()`: concatena strings em uma única string.

Por padrão, o separador é `sep = " "`, mas você pode escolher outros.

Dependendo do separador, essa função é equivalente a `paste0()` e `paste()` da base.



```
texto <- "O valor p é: "
pvalor <- "0.03"

str_c(texto, pvalor) # pacote stringr
[1] "O valor p é: 0.03"

paste0(texto, pvalor) # base
[1] "O valor p é: 0.03"
```

str_c() - Função Vetorizada

A função **str_c()** é vetorizada. Veja o exemplo abaixo:

```
string1 <- "ME115"  
string2 <- c("A", "B")  
string3 <- c("pela profa. Larissa", "pelo prof. Rafael")  
  
str_c("Estou cursando ", string1, " na turma ", string2,  
      ", ministrada ", string3, ".")
```

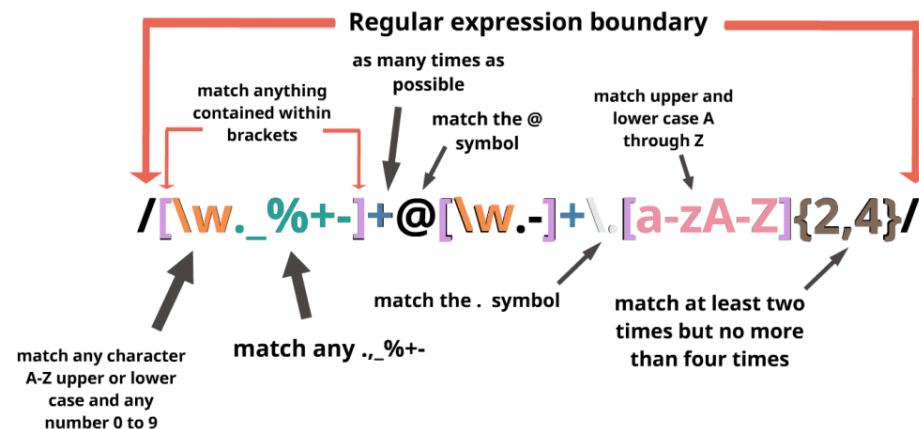
```
## [1] "Estou cursando ME115 na turma A, ministrada pela profa. Larissa."  
## [2] "Estou cursando ME115 na turma B, ministrada pelo prof. Rafael."
```

Note que o comprimento da **string1** é menor que o comprimento das demais strings. Então, esse elemento é reciclado.

Expressões Regulares

Ao trabalhar com textos/strings, é comum surgir a necessidade do emprego de expressões regulares (*regex* ou *regexp*).

Expressões Regulares: permitem identificar em buscas de texto, conjuntos de caracteres, palavras e outros padrões por meio de uma sintaxe concisa.



Não se assuste! À primeira vista, uma expressão regular pode parecer bem confusa.

Expressões Regulares

Expressões Regulares: combinação de caracteres literais e metacaracteres.

São usadas em várias linguagens de programação, e não apenas em R, como por exemplo: Python, Perl, etc.

Para visualizar e aprender como as expressões regulares trabalham, iremos usar as funções `str_view()`, `str_view_all()`, e também explorar outras funções do `stringr`.

Padrões simples: procurar por uma sequência literal de caracteres em uma string ou vetor de strings:

```
frase <- c("Eu não programava em R. Agora estou cursando ME115 e adorando!")  
str_view_all(frase, "an")
```

```
## [1] | Eu não programava em R. Agora estou curs<an>do ME115 e ador<an>do!
```

Metacaracteres

Metacaracteres: são símbolos não alfa-numéricos, como . \ | () [{ \$ * + ?.

Os metacaracteres têm significados especiais nas expressões regulares.

| Character | Brief Description | Simple Example |
|-----------|------------------------------|----------------|
| . | any character (wildcard) | w.kly |
| ^ | Begins with | ^where |
| [] | character set | [0-9] |
| | either or | hi bye |
| + | Once or more | me+ |
| * | zero occurrences or more | me* |
| {} | exact number of times | I{4} |
| \ | special character operations | \w |
| \$ | Ends with | \$bar |

Para buscar por um metacaracter literalmente no R, é preciso usar barras duplas \\ antes. Exemplo:

- **pattern = "\\."** busca exatamente pelo ponto “.”
- **pattern = ". "** busca por qualquer coisa

Fonte da Figura: [Towards Data Science - An Introduction to Regular Expressions](#)

Exemplo - Expressões Regulares

A tabela mostra a aplicação de cinco regex em seis strings distintas.

- `^ban` reconhece apenas o que começa exatamente com “ban”.
- `b ?an` reconhece tudo que tenha “ban”, com ou sem espaço entre o “b” e o “a”.
- `ban` reconhece tudo que tenha “ban”, apenas minúsculo.
- `BAN` reconhece tudo que tenha “BAN”, apenas maiúsculo.
- `ban$` reconhece apenas o que termina exatamente em “ban”

| strings | <code>^ban</code> | <code>b ?an</code> | <code>ban</code> | <code>BAN</code> | <code>ban\$</code> |
|-----------------|-------------------|--------------------|------------------|------------------|--------------------|
| abandonado | FALSE | TRUE | TRUE | FALSE | FALSE |
| ban | TRUE | TRUE | TRUE | FALSE | TRUE |
| banana | TRUE | TRUE | TRUE | FALSE | FALSE |
| BANANA | FALSE | FALSE | FALSE | TRUE | FALSE |
| ele levou ban | FALSE | TRUE | TRUE | FALSE | TRUE |
| pranab anderson | FALSE | TRUE | FALSE | FALSE | FALSE |

Fonte: [Curso-R - Seção 7.4](#)

Indicadores de Posição

Por padrão, expressões regulares irão corresponder a qualquer parte de uma string, mas às vezes, é interessante indicar onde queremos fazer a busca, se no início ou final da string. Para isso, temos duas âncoras específicas:

- ^ para corresponder ao início da string
- \$ para corresponder ao final da string.

```
x <- c("apple pie", "apple", "apple cake")
str_detect(x, "^apple.")
```

```
## [1] TRUE FALSE TRUE
```

```
str_detect(x, "apple$")
```

```
## [1] FALSE TRUE FALSE
```

Conjuntos ou Classes de Caracteres

Colocando caracteres dentro de colchetes [], reconhecemos quaisquer caracteres desse conjunto. Alguns exemplos práticos:

- [Cc]asa para reconhecer “casa” em maiúsculo ou minúsculo.
- [0-9] para reconhecer somente números. O mesmo vale para letras [a-z], [A-Z], [a-zA-Z] etc.
- O símbolo ^ dentro do colchete significa negação. Por exemplo, [^0-9] significa pegar tudo o que não é número.
- O símbolo . fora do colchete indica “qualquer caractere”, mas dentro do colchete é apenas ponto.
- Use [:space:] + para reconhecer espaços e [:punct:] + para reconhecer pontuações.

Conjuntos ou Classes de Caracteres

Vejas as duas tabelas abaixo com algumas classes/conjuntos de caracteres. A tabela da direita é chamada de classes POSIX.

| Anchor | Description |
|--------------|--|
| [aeiou] | match any specified lower case vowel |
| [AEIOU] | match any specified upper case vowel |
| [0123456789] | match any specified numeric value |
| [0-9] | match any range of specified numeric values |
| [a-z] | match any range of lower case letter |
| [A-Z] | match any range of upper case letter |
| [a-zA-Z0-9] | match any of the above |
| [^aeiou] | match anything other than a lowercase vowel |
| [^0-9] | match anything other than the specified numeric values |

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

| Anchor | Description |
|------------|--|
| [:lower:] | lower-case letters |
| [:upper:] | upper-case letters |
| [:alpha:] | alphabetic characters |
| [:digit:] | numeric values |
| [:alnum:] | alphanumeric characters |
| [:blank:] | blank characters (space & tab) |
| [:cntrl:] | control characters |
| [:punct:] | punctuation characters: ! " # % & ' () * + , - . / ; |
| [:space:] | space characters: tab, newline, vertical tab, space, etc |
| [:xdigit:] | hexadecimal digits: 0-9 A B C D E F a b c d e f |
| [:print:] | printable characters |
| [:graph:] | graphical characters |

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Importante: o valor passado para o argumento `pattern` de qualquer função do pacote `stringr` será entendido como uma `regex`.

Sequências

Na tabela ao lado, indicamos como identificar uma sequência de caracteres de um certo tipo.

Veja no exemplo abaixo, no qual queremos identificar onde existem dígitos e, possivelmente, substituí-los por algum outro caractere como “_”.

| Anchor | Description |
|--------|------------------------------|
| \d | match a digit character |
| \D | match a non-digit character |
| \s | match a space character |
| \S | match a non-space character |
| \w | match a word |
| \W | match a non-word |
| \b | match a word boundary |
| \B | match a non-word boundary |
| \h | match a horizontal space |
| \H | match a non-horizontal space |
| \v | match a vertical space |
| \V | match a non-vertical space |

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

```
texto <- "Meu CPF é 111.222.333-44"
str_view_all(texto, pattern = "\\\d")
## [1] | Meu CPF é <1><1><1>.<2><2><2>.<3><3><3>-<4><4>
```

```
# substituir qualquer dígito com underscore
str_replace_all(texto, pattern = "\\\d", replacement = "_")
## [1] "Meu CPF é ___.___._-_"
```

Quantificadores

Com os quantificadores você pode controlar quantas vezes um padrão aparece.

- ?: 0 ou 1
- +: 1 ou mais
- *: 0 ou mais
- {n}: exatamente n
- {n,}: n ou mais
- {,m}: no máximo m
- {n,m}: entre n e m

| Quantifier | Description |
|------------|---|
| ? | the preceding item is optional and will be matched at most once |
| * | the preceding item will be matched zero or more times |
| + | the preceding item will be matched one or more times |
| {n} | the preceding item is matched exactly n times |
| {n,} | the preceding item is matched n or more times |
| {n,m} | the preceding item is matched at least n times, but not more than m times |

*adapted from *Handling and Processing Strings in R* (Sanchez, 2013)

Exemplo:

```
str_view_all(texto, pattern = "[0-9]{3}")
```

```
## [1] | Meu CPF é <111>.<222>.<333>-44
```

Funções do **stringr** com expressões regulares

Já vimos as funções básicas do **stringr** e aprendemos um pouco de regex. Vamos então às funções mais avançadas.

Basicamente, essas funções buscarão padrões em um vetor de strings e farão alguma coisa quando encontrá-lo. O que será feito é indicado pelo verbo que está como sufixo das funções **str_***() abaixo.

As principais funções são:

str_detect(), **str_replace()**, **str_extract()**, **str_split()** e **str_subset()**.

Todo valor passado para o argumento **pattern** dessas funções será entendido como uma **regex**.



Detectar um Padrão



`str_detect()`: retorna TRUE se a regex é compatível com a string e FALSE caso contrário.

```
cidados <- c("S. José do Rio Preto", "São Paulo", "S. S. da Gramma")  
  
str_detect(cidados, pattern = "[Pp]aulo$")  
[1] FALSE TRUE FALSE
```

Na base do R, a função equivalente seria `grepl()`:

```
grepl(pattern = "[Pp]aulo$", cidados)  
[1] FALSE TRUE FALSE
```

Compare a sintaxe de `str_detect()` versus `grepl()`.

Substituir um Padrão



`str_replace()` e `str_replace_all()`: substituem a primeira ou todas as ocorrências de um padrão, respectivamente, por um outro padrão.

```
str_replace(cidades, pattern = "S[.]", replacement = "São")
[1] "São José do Rio Preto" "São Paulo" "São S. da Gramma"
```

```
str_replace_all(cidades, pattern = "S[.]", replacement = "São")
[1] "São José do Rio Preto" "São Paulo" "São São da Gramma"
```

Na base do R, as funções equivalentes a essas seriam `sub()` e `gsub()`:

```
sub(pattern = "S[.]", replacement = "São", cidades) # equivalente a str_replace()
gsub(pattern = "S[.]", replacement = "São", cidades) # equivalente a str_replace_all()
```

Remover um Padrão

`str_remove()` e `str_remove_all()`: removem a primeira ou todas as ocorrências de um padrão, respectivamente.

Equivalente a fazer `str_replace(string, pattern, "")`.

```
cidados <- c("S. José do Rio Preto", "São Paulo", "S. S. da Gramma")
```

```
str_remove(cidados, pattern = "S.{1,2} ")
[1] "José do Rio Preto" "Paulo" "S. da Gramma"
```

Se queremos remover todas as ocorrências de “São” ou “S.”:

```
str_remove_all(cidados, pattern = "S.{1,2} ")
[1] "José do Rio Preto" "Paulo" "da Gramma"
```



Extrair um Padrão



`str_extract()` e `str_extract_all()`: extraem a primeira ou todas as ocorrências de um padrão, respectivamente, de um vetor de strings.

Nesse exemplo, iremos extrair todos os sobrenomes do vetor de nomes:

```
r_core_group <- c('Douglas Bates', 'John Chambers', 'Peter Dalgaard',
                  'Robert Gentleman', 'Kurt Hornik', 'Ross Ihaka', '...')

str_extract(r_core_group, pattern = '[[:alpha:]]+$')
str_extract(r_core_group, pattern = '[A-Za-z]+$')

[1] "Bates"      "Chambers"    "Dalgaard"    "Gentleman"   "Hornik"      "Ihaka"       NA
```

Separar uma string



`str_split()`: separa uma string em várias de acordo com um separador.

```
fruits <- c("apples and oranges and pears and bananas", "pineapples and mangos")  
  
str_split(fruits, pattern = " and ") # retorna uma lista  
str_split(fruits, pattern = " and ", simplify = TRUE) # retorna uma matriz  
[,1]      [,2]      [,3]      [,4]  
[1,] "apples"   "oranges"  "pears"   "bananas"  
[2,] "pineapples" "mangos"  ""        ""
```

Equivalente à função `strsplit()` da base.

`str_split_fixed()`: retorna um número fixo de colunas determinado por `n`.

```
str_split_fixed(fruits, pattern = " and ", n = 2)
```

Subconjuntos



`str_subset()`: retorna somente as strings compatíveis com a regex.

```
frases <- c('o rato roeu', 'a roupa do rei', 'de roma')
str_subset(frases, pattern = 'd[aeo]')

[1] "a roupa do rei" "de roma"
```

É o mesmo que indexar o vetor `frases` usando a função o vetor lógico resultante de `str_detect()`.

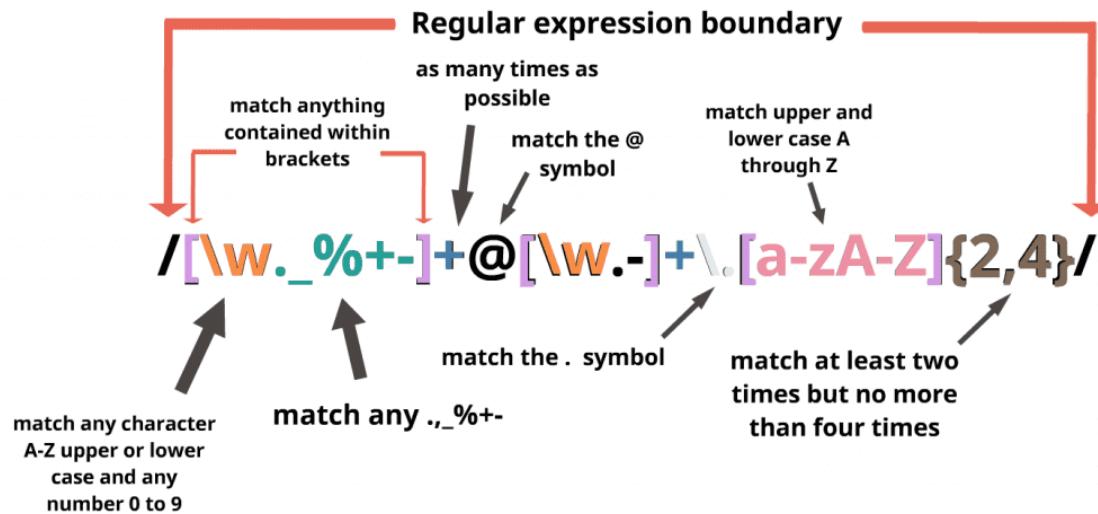
```
frases[str_detect(frases, 'd[eo]')]

[1] "a roupa do rei" "de roma"
```

Expressões Regulares

Como aprender expressões regulares? PRÁTICA

Veja se agora fica mais claro o que a expressão regular abaixo faz.



Identificar um endereço de e-mail do tipo username@domain.com.



Manipulação de Data/Hora

Lubridate

- Manipular data e hora pode ser algo complexo e trabalhoso.
- Os métodos que usamos com datas e horas devem ser robustos para fusos horários, dias bissextos, horários de verão e outras peculiaridades e o R não têm esses recursos no seu pacote base.
- Para sanar esses problemas, temos o pacote **lubridate**.
- Esse pacote torna mais fácil fazer as coisas que o R já faz com data/hora e possível fazer as coisas que R não faz.
- O pacote **lubridate** não faz parte do **tidyverse**, mas basta carregá-lo (desde que esteja instalado) da maneira usual e trabalhar com datas/horas:

```
library(lubridate)
```



Converter Strings para Datas

Funções práticas e fáceis para converter strings para objetos da classe “Date”:

`ymd()`, `ymd_hms`, `dmy()`, `dmy_hms`, `mdy()`.

```
ymd(20101215)
#> [1] "2010-12-15"
```

```
mdy("4/1/17")
#> [1] "2017-04-01"
```

```
mdy(02022002)
#[1] "2002-02-02"
```

```
dmy("04/06/2011")
#> [1] "2011-06-04"
```

```
dmy(12022002)
#> [1] "2002-02-12"
```

Converter Strings para Datas com Hora

Se a data incluir informações de hora, adicione o sufixo `_h`, `_hm`, ou `_hms` ao nome das funções `ymd()`, `mdy()`, `dmy()`, de acordo com o formato de hora.

Para ler as datas em um determinado fuso horário, forneça o nome oficial desse fuso horário no argumento `tz`.

```
arrive <- ymd_hms("2011-06-04 12:00:00", tz = "Pacific/Auckland")
arrive
#> [1] "2011-06-04 12:00:00 NZST"
```

```
leave <- ymd_hms("2011-08-10 14:00:00", tz = "Pacific/Auckland")
leave
#> [1] "2011-08-10 14:00:00 NZST"
```



lubridate: Funções Vetorizadas

As funções do `lubridate` são vetorizadas e prontas para serem usadas em configurações interativas e dentro de funções.

Exemplo: considere a função para avançar uma data para o último dia do mês.



```
last_day <- function(date) {  
  ceiling_date(date, "month") - days(1)  
}  
  
last_day(ymd_hms("2000-01-01 00:00:00"))  
  
## [1] "1999-12-31 UTC"
```

Extraindo informações de Datas/Horas

Para extrair informações de horários de data use as funções `second`, `minute`, `hour`, `day`, `wday`, `yday`, `week`, `month`, `year` e `tz`.

As funções `wday` e `month` possuem o argumento opcional `label`, que substitui sua saída numérica pelo nome do dia da semana ou mês.

```
arrive <- ymd_hms("2011-06-04 12:00:00", tz = "Pacific/Auckland")
second(arrive) <- 25
arrive
#> [1] "2011-06-04 12:00:25 NZST"

second(arrive) <- 0
wday(arrive)
#> [1] 7

wday(arrive, label = TRUE)
#> [1] Sat
#> Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat
```

Trabalhando com `with_tz` e `force_tz`

Existem duas coisas muito úteis para fazer com datas e fusos horários:

1. Exiba o mesmo momento em um fuso horário diferente.
2. Crie um novo momento combinando uma hora existente com um novo fuso horário.

Isso é feito pelas funções `with_tz()` e `force_tz()`, respectivamente.

Exemplo: o autor do `lubridate`, Tal Galili, estava em Auckland, Nova Zelândia. Ele marcou uma reunião virtual com o seu coautor, Hadley, às 9AM (horário de Auckland). Que horas seriam para Hadley, que estava em Houston/TX?

```
meeting <- ymd_hms("2011-07-01 09:00:00", tz = "Pacific/Auckland")
with_tz(meeting, "America/Chicago")
#> [1] "2011-06-30 16:00:00 CDT"
```

Trabalhando com `with_tz` e `force_tz`

No exemplo anterior, vimos que as reuniões ocorreram às 4:00PM no horário de Hadley e às 9:00AM na Nova Zelândia. Parece ser um dia diferente devido à curvatura da Terra.

E se Hadley cometeu um erro e marcou a reunião às 9:00AM no horário dele? Que horas seria então a reunião para Tal?

```
meeting <- ymd_hms("2011-07-01 09:00:00", tz = "Pacific/Auckland")
mistake <- force_tz(meeting, "America/Chicago")
with_tz(mistake, "Pacific/Auckland")

#> [1] "2011-07-02 02:00:00 NZST"
```

A reunião seria às 2:00AM no horário de Tal!

Intervalos de Tempo

Exemplo: Tal ficou em Auckland no período de 4 de junho de 2011 a 10 de agosto de 2011.

Podemos criar esse intervalo de tempo de duas maneiras:

```
arrive <- ymd_hms("2011-06-04 12:00:00", tz = "Pacific/Auckland")
leave <- ymd_hms("2011-08-10 14:00:00", tz = "Pacific/Auckland")
```

```
auckland <- interval(arrive, leave)
auckland
#> [1] 2011-06-04 12:00:00 NZST--2011-08-10 14:00:00 NZST
```

```
auckland <- arrive %--% leave
auckland
#> [1] 2011-06-04 12:00:00 NZST--2011-08-10 14:00:00 NZST
```



Intervalos de Tempo

O amigo de Tal, Chris, viajou para várias conferências naquele ano, incluindo a do *Joint Statistical Meetings* (JSM). Isso o tirou do país de 20 de julho até o final de agosto.



```
jsm <- interval(ymd(20110720, tz = "Pacific/Auckland"),  
                 ymd(20110831, tz = "Pacific/Auckland"))  
  
jsm  
#> [1] 2011-07-20 NZST--2011-08-31 NZST
```

A visita de Tal coincidirá com as viagens de Chris?

```
int_overlaps(jsm, auckland)  
#> [1] TRUE
```

Intervalos de Tempo

Em que parte da visita Tal e Chris se encontraram?

```
setdiff(auckland, jsm)
#> [1] 2011-06-04 12:00:00 NZST--2011-07-20 NZST
```

Outras funções que funcionam com intervalos incluem:

```
int_start(), int_end(),
int_flip(), int_shift(), int_aligns(),
union(), intersect(), setdiff() e %within%.
```



Aritmética com Data e Hora

O `lubridate` também fornece duas classes de intervalo de tempo gerais: durações e períodos.



- **Períodos:** as funções para criação de períodos são nomeadas de acordo com as unidades de tempo adicionadas de um sufixo `s`, como no plural. Exemplo: `months()`, `days()`, `minutes()`, etc.
- **Durações:** as funções para criação de durações são nomeadas de acordo com as unidades de tempo adicionadas de um prefixo `d` e um sufixo `s`. Exemplo: `dmonths()`, `ddays()`, `dminutes()`, etc.

```
minutes(2) # período  
#> [1] "2M 0S"  
  
dminutes(2) # duração  
#> [1] "120s (~2 minutes)"
```

Aritmética com Data e Hora

A classe `duration` sempre fornecerá resultados matematicamente precisos. Um ano de duração será sempre igual a 365 dias. As funções relacionadas aos períodos, por outro lado, flutuam da mesma forma que a linha do tempo.

Exemplo: as durações serão mais precisas que o período se for um ano bissexto.

```
leap_year(2011) # ano regular
#> [1] FALSE
ymd(20110101) + dyears(1)
#> [1] "2012-01-01 06:00:00 UTC"
ymd(20110101) + years(1)
#> [1] "2012-01-01"

leap_year(2012) # ano bissexto
#> [1] TRUE
ymd(20120101) + dyyears(1)
#> [1] "2012-12-31 06:00:00 UTC"
ymd(20120101) + years(1)
#> [1] "2013-01-01"
```

Aritmética com Data e Hora

Você pode usar períodos e durações para fazer aritmética básica com horários de datas.

Exemplo: se Tal quisesse marcar com Hadley uma reunião virtual semanal e recorrente, poderia ser feito:

```
meeting <- ymd_hms("2011-07-01 09:00:00", tz = "Pacific/Auckland")
meetings <- meeting + weeks(0:5)
meetings

#> [1] "2011-07-01 09:00:00 NZST" "2011-07-08 09:00:00 NZST"
#> [3] "2011-07-15 09:00:00 NZST" "2011-07-22 09:00:00 NZST"
#> [5] "2011-07-29 09:00:00 NZST" "2011-08-05 09:00:00 NZST"
```



Aritmética com Data e Hora

Hadley viajou para conferências ao mesmo tempo que Chris.

Nesse caso, qual dessas reuniões seria afetada?



```
jsm
#> [1] 2011-07-20 NZST--2011-08-31 NZST

meeting <- ymd_hms("2011-07-01 09:00:00", tz = "Pacific/Auckland")
meetings <- meeting + weeks(0:5)
meetings %within% jsm
#> [1] FALSE FALSE FALSE TRUE TRUE TRUE

meetings[meetings %within% jsm]
#> [1] "2011-07-22 09:00:00 NZST" "2011-07-29 09:00:00 NZST"
#> [3] "2011-08-05 09:00:00 NZST"
```

Aritmética com Data e Hora



Quanto tempo durou a estada de Tal em Auckland?

```
auckland / dmonths(1) # em meses  
#> [1] 2.20397
```

```
auckland / ddays(1) # em dias  
#> [1] 67.08333
```

Alternativamente, podemos fazer divisão inteira (`%/%`). Às vezes, isso é mais sensato, já que não é óbvio como expressar o resto como uma fração de um mês, dado que a duração de um mês muda constantemente.

```
auckland %/% dmonths(1) # divisão inteira em meses  
#> [1] 2
```

```
auckland %/% ddays(1) # divisão inteira em dias  
#[1] 67
```

Aritmética com Data e Hora

Você pode transformar qualquer intervalo de tempo em um período de tempo generalizado, podendo especificar as unidades, com a função `as.period()`.



```
auckland  
#> [1] 2011-06-04 12:00:00 NZST--2011-08-10 14:00:00 NZST  
  
as.period(auckland)  
#> [1] "2m 6d 2H 0M 0S"
```

O operador módulo (`%%`) aplicado em um intervalo de tempo retorna o resto da divisão como um novo intervalo (menor), que pode ser então transformado em um período menor.

```
as.period(auckland %% months(1)) ## divisão por módulo  
#> [1] "6d 2H 0M 0S"
```

String manipulation with stringr - Cheat Sheet

String manipulation with stringr :: CHEAT SHEET

The `stringr` package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.



Detect Matches

| | |
|--|---|
| | <code>str_detect(string, pattern, negate = FALSE)</code> Detect the presence of a pattern match in a string. Also <code>str_like()</code> , <code>str_detect(fruit, "a")</code> |
| | <code>str_starts(string, pattern, negate = FALSE)</code> Detect the presence of a pattern match at the beginning of a string. Also <code>str_ends()</code> , <code>str_starts(fruit, "a")</code> |
| | <code>str_which(string, pattern, negate = FALSE)</code> Find the indexes of strings that contain a pattern match. <code>str_which(fruit, "a")</code> |
| | <code>str_locate(string, pattern)</code> Locate the positions of pattern matches in a string. Also <code>str_locate_all()</code> , <code>str_locate(fruit, "a")</code> |
| | <code>str_count(string, pattern)</code> Count the number of matches in a string. <code>str_count(fruit, "a")</code> |

Subset Strings

| | |
|--|---|
| | <code>str_sub(string, start = 1L, end = -1L)</code> Extract substrings from a character vector. <code>str_sub(fruit, 1, 3); str_sub(fruit, -2)</code> |
| | <code>str_subset(string, pattern, negate = FALSE)</code> Return only the strings that contain a pattern match. <code>str_subset(fruit, "p")</code> |
| | <code>str_extract(string, pattern)</code> Return the first pattern match found in each string, as a vector. Also <code>str_extract_all()</code> to return every pattern match. <code>str_extract(fruit, "[aeiou]")</code> |
| | <code>str_match(string, pattern)</code> Return the first pattern match found in each string, as a matrix with a column for each () group in pattern. Also <code>str_match_all()</code> . <code>str_match(sentences, "(a)(the) ([^])")</code> |
| | |

Manage Lengths

| | |
|--|---|
| | <code>str_length(string)</code> The width of strings (i.e. number of code points, which generally equals the number of characters). <code>str_length("fruit")</code> |
| | <code>str_pad(string, width, side = c("left", "right", "both"), pad = " ")</code> Pad strings to constant width. <code>str_pad(fruit, 17)</code> |
| | <code>str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...")</code> Truncate the width of strings, replacing content with ellipsis. <code>str_trunc(sentences, 6)</code> |
| | <code>str_trim(string, side = c("both", "left", "right"))</code> Trim whitespace from the start and/or end of a string. <code>str_trim(str_pad(fruit, 17))</code> |
| | <code>str_squish(string)</code> Trim whitespace from each end and collapse multiple spaces into single spaces. <code>str_squish(str_pad(fruit, 17, "both"))</code> |

Mutate Strings

| | |
|--|--|
| | <code>str_sub()</code> <- value. Replace substrings by identifying the substrings with <code>str_sub()</code> and assigning into the results. <code>str_sub(fruit, 1, 3) <- "st"</code> |
| | <code>str_replace(string, pattern, replacement)</code> Replace the first matched pattern in each string. Also <code>str_remove()</code> . <code>str_replace(fruit, "p", "a")</code> |
| | <code>str_replace_all(string, pattern, replacement)</code> Replace all matched patterns in each string. Also <code>str_remove_all()</code> . <code>str_replace_all(fruit, "p", "a")</code> |
| | <code>str_to_lower(string, locale = "en")</code> Convert strings to lower case. <code>str_to_lower(sentences)</code> |
| | <code>str_to_upper(string, locale = "en")</code> Convert strings to upper case. <code>str_to_upper(sentences)</code> |
| | <code>str_to_title(string, locale = "en")</code> Convert strings to title case. Also <code>str_to_sentence()</code> . <code>str_to_title(sentences)</code> |



Join and Split

| | |
|--|---|
| | <code>str_c(..., sep = "")</code> Join multiple strings into a single string. <code>str_c(letters, LETTERS)</code> |
| | <code>str_flatten(string, collapse = "")</code> Combines into a single string, separated by collapse. <code>str_flatten(fruit, ",")</code> |
| | <code>str_dup(string, times)</code> Repeat strings times times. Also <code>str_unique()</code> to remove duplicates. <code>str_dup(fruit, times = 2)</code> |
| | <code>str_split_fixed(string, pattern, n)</code> Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also <code>str_split()</code> to return a list of substrings and <code>str_split_n()</code> to return the nth substring. <code>str_split_fixed(sentences, "", n=3)</code> |
| | <code>str_glue(..., sep = "", .envir = parent.frame())</code> Create a string from expressions to evaluate. <code>str_glue("Pi is {pi}")</code> |
| | <code>str_glue_data(x, ..., sep = "", .envir = parent.frame(), na = "NA")</code> Use a data frame, list, or environment to create a string from strings and expressions to evaluate. <code>str_glue_data(mtcars, "rownames[mtcars] has {hp} hp")</code> |

Order Strings

| | |
|--|--|
| | <code>str_order(x, decreasing = FALSE, na.last = TRUE, locale = "en", numeric = FALSE, ...)</code> Return the vector of indexes that sorts a character vector. <code>str_order(fruit)</code> |
| | <code>str_sort(x, decreasing = FALSE, na.last = TRUE, locale = "en", numeric = FALSE, ...)</code> Sort a character vector. <code>str_sort(fruit)</code> |

Helpers

| | |
|--|--|
| | <code>str_view_all(string, pattern, match = NA)</code> View HTML rendering of all regex matches. Also <code>str_view()</code> to see only the first match. <code>str_view_all(sentences, "[aeiou]")</code> |
| | <code>str_equal(x, y, locale = "en", ignore_case = FALSE, ...)</code> Determine if two strings are equivalent. <code>str_equal(c("a", "b"), c("a", "c"))</code> |
| | <code>str_wrap(string, width = 80, indent = 0, exdent = 0)</code> Wrap strings into nicely formatted paragraphs. <code>str_wrap(sentences, 20)</code> |
| | This is a long sentence. This is a long sentence. |

¹ See bit.ly/1SO639-1 for a complete list of locales.

CC BY SA Posit Software, PBC • info@posit.co • posit.co • Learn more at stringr.tidyverse.org • Diagrams from @GWWdor on Twitter • stringr 1.5.0 • Updated: 2023-05

Fonte: [String manipulation with stringr - Cheat Sheet](#)

Referências

Algumas referências utilizadas para a construção desse material:

- [R for Data Science - Chapters 14 e 16](#)
- [Curso R - Capítulo 7](#)
- [Stringr Vignette](#)
- [Stringr - Cheat Sheet](#)
- [Lubridate - Cheat Sheet](#)
- [Regular Expressions 101](#)



Slides produzidos pelas profas. Tatiana Benaglia e Mariana R. Motta.