INTRODUÇÃO AO R: STRINGR/LUBRIDATE

C. Lente + Curso-R

2018-01-22

Índice

O que veremos hoje:

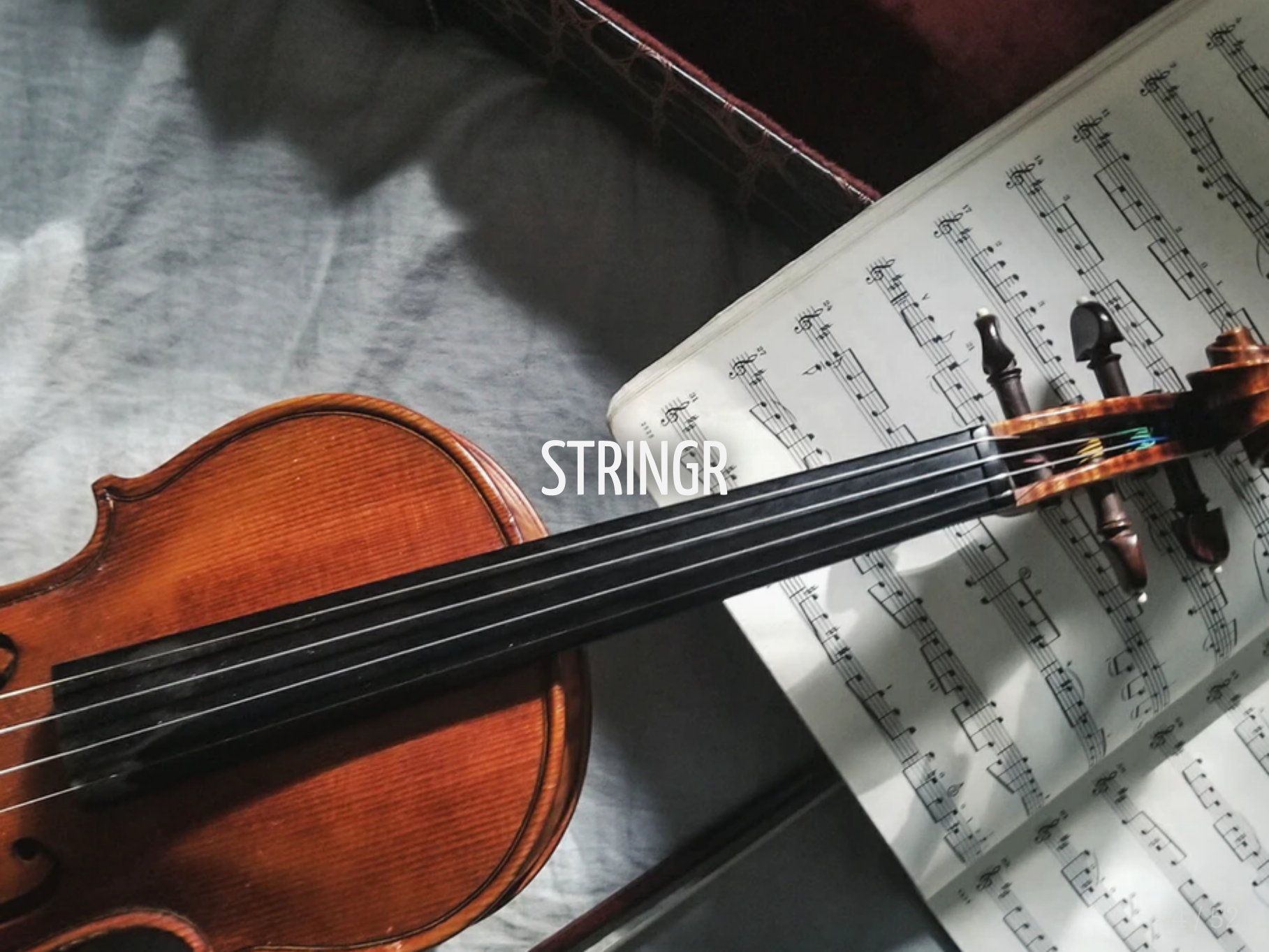
- Stringr
    - O que são strings?
    - Mexer com strings é fácil
    - Regex
    - Mexer com strings é dífícil
- Lubridate
    - O que são datas?
    - Criando datas
    - Mexer com datas é fácil
    - Mexer com datas é difícil

Nossa motivação

"Por que vamos aprender a mexer com strings e datas?" e "Por que tem que ser com esses pacotes dos quais eu nunca ouvi falar?"

Porque:

- Variáveis que trazem textos ou datas são extremamente **importantes**
- Raramente encontramos tabelas que contém somente números e valores binários
- O R `base` tem funções muito ~~ruins~~ pouco consistentes para lidar com esses tipos de dados
- Os pacotes `lubridate` e `stringr` são excelentes opções que já vêm embutidas no **tidyverse**
- Ambos foram idealizados por ~~um anjo que roubou meu coração~~ Hadley Wickham


STRINGR

O que são strings?

O básico

Strings não passam de sequências de caracteres! Em português o termo é literalmente traduzido como *cadeia de caracteres*.

```
minha_primeira_string <- "Eu adoro o IME!"
minha_primeira_string
```

```
#> [1] "Eu adoro o IME!"
```

Strings aceitam praticamente qualquer coisa que está entre aspas, então podemos ir à loucura:

```
minha_segunda_string <- "こんにちは! Está 40\u00BAC na lá ɐɹoʃ"
minha_segunda_string
```

```
#> [1] "こんにちは! Está 40ºC na lá ɐɹoʃ"
```

EX: Rode a função `class()` em uma string. O que ela realmente é?

Escapando pela tangente

O que acontece se tentarmos colocar uma expressão entre aspas dentro de uma string?

```
"Meu nome é "C. Lente""
```

O R não nos permite fazer isso! Temos duas alternativas:

```
"Meu nome é 'C. Lente'"
"Meu nome é \"C. Lente\""
```

A segunda estratégia usa o que chamamos de *escaping*: um símbolo (\) que indica que o próximo caractere não é o que parece.

```
cat("Meu nome é \"C. Lente\"")
```

```
#> Meu nome é "C. Lente"
```

EX: Verifique as diferenças entre o comportamento entre `print()` e `cat()`.

Caracteres especiais

Usando a nossa nova ferramenta, podemos descrever uma infinitude de caracteres não tradicionais.

```
cat("Uma lista feita à mão:\n\t-Item 1;\n\t-Item dois.")
```

```
#> Uma lista feita à mão:
#>     -Item 1;
#>     -Item dois.
```

Os caracteres especiais `\n` e `\t` indicam respectivamente uma nova linha e uma tabulação.

Se não pudermos usar caracteres não-ASCII (como é com o CRAN), podemos usar a barra oblíqua aliada a códigos Unicode para descrever qualquer caractere imaginável:

```
"\u03b1\u03b2\u03b3"
```

```
#> [1] "αβγ"
```

Vetores de strings

Em outras linguagens (como C), strings são vetores, o que quer dizer que cada posição de uma string corresponde a um caractere! Já no R, `character` é uma **classe atômica** e então isso não acontece.

```
c("Um", "vetor", "de", "strings")[4]
```

```
#> [1] "strings"
```

Isso pode ser um pouco difícil de entender se você já programou em alguma linguagem diferente de R, mas não demora pra pegar o jeito.

A única coisa com que precisamos tomar cuidado é **fechar todas as aspas**! Mesmo que estejamos em um vetor, deixar uma aspa aberta vai fazer o console ficar esperando pelo final da string:

```
> c("Estou declarando", "um vetor", "de strings
+
+
+ SOCORRO!
```

Até agora

- `character` (genericamente conhecido como "string") é um tipo de dado que serve para amazenar textos
- Podemos incluir praticamente qualquer coisa em uma string
- Usando a barra oblíqua para a esquerda, espapamos o caractere que a segue
- Podemos criar vetores de strings como com qualquer outro tipo de dado

Exercício

Crie um vetor que carrega dois pequenos textos. Cada texto deve ter dois parágrafos cada e cada parágrafo deve ter pelo menos 3 linhas. Cada parágrafo deve começar com uma tabulação e nenhuma linha pode passar de 80 caracteres. Pelo menos um dos textos deve ter uma citação.

Dicas: Para saber quantos caracteres tem cada linha, use o contador na esquerda inferior do RStudio. Escreva cada linha separadamente e depois junte-as com caracteres especiais.d

Mexer com strings é fácil

O pacote stringr

Todas as funções do `stringr` começam com o prefixo `str_`. Isso facilita muito quando precisamos buscar buscar alguma coisa no nosso environment!

```
library(stringr)
```

EX: Carregue o `stringr` e procure por suas funções usando a tecla TAB.

A função mais básica do pacote é `str_length()`, que retorna o número de caracteres em uma string. Como todas as funções do `stringr` são vetorizadas, o primeiro argumento delas pode ter tanto uma única string quando um vetor de strings.

```
str_length(c("Um", "Dois", "Três ", "Quatro", "Cinco"))
```

```
#> [1] 2 4 5 6 5
```

EX: Verifique como são contados caracteres escapados.

As maravilhas da formatação

Uma das tarefas mais comuns que precisamos executar quando se trata de strings é transformar um texto de formatação duvidosa em um texto com a formatação que desejamos:

```
s <- "nAh uNIaUM xOvIeHtIkA, U MiGuXeixxXXxx ixXxKreVi vC"
str_to_lower(s)
```

```
#> [1] "nah uniaum xoviehtika, u miguxeixxxxxx ixxxkrevi vc"
```

```
str_to_upper(s)
```

```
#> [1] "NAH UNIAUM XOVIEHTIKA, U MIGUXEIXXXXXX IXXXKREVI VC"
```

```
str_to_title(s)
```

```
#> [1] "Nah Uniaum Xoviehtika, U Miguxeixxxxxx Ixxxkrevi Vc"
```

Agora o texto está muito mais compreensível!

Tirando pedaços

Outra utilidade importante é retirar fatias indesejadas de strings. Se tivermos um texto com espaços extras no início e no final, podemos recorrer à função `str_trim()`:

```
str_trim(c("M", "F", "F", " M", " F ", "M"))
```

```
#> [1] "M" "F" "F" "M" "F" "M"
```

EX: Tabulações são consideradas espaços?

Já a função `str_sub()` usa as posições dos caracteres nas strings para determinar o que remover:

```
str_sub(c("__SP__", "__MG__", "__RJ__"), start = 3, end = 4)
```

```
#> [1] "SP" "MG" "RJ"
```

EX: Teste `str_sub()` dando valor só para `start` ou só para `end`. O que acontece se passarmos números negativos para ambos os parâmetros?

Concatenação

Assim como temos `length()` e `str_length()`, temos `c()` e `str_c()`:

```
str_c("O valor p é: ", "0.03")
```

```
#> [1] "O valor p é: 0.03"
```

EX: E se passássemos uma variável numérica para essa função?

Vetorização também não é um problema para a `str_c()`:

```
s1 <- c("O R", "O Java")
s2 <- c("bom", "ruim")
str_c(s1, " é muito ", s2)
```

```
#> [1] "O R é muito bom"     "O Java é muito ruim"
```

EX: Use o argumento `sep` para remover a repetição de espaços. Use o argumento `collapse` para juntar as duas frases em uma.

Até agora

- Com `str_length()` podemos contar quantos caracteres tem uma string
- Com `str_to_*()` podemos formatar uma string facilmente
- Com `str_trim()` e `str_sub()` podemos retirar pedaços de strings
- Com `str_c()` podemos concatenar strings

Exercício

Partindo do vetor de strigs `vs`, obtenha o texto `s`:

```
vs <- c("***O número   " , "de caRACTeres", "   nEste TexTO", "é")
s <- "o número de caracteres neste texto é 36"
```

Dicas: Use pipes (você só precisará de uma pipeline) e lembre-se do *placeholder*. Para saber se ambos os textos obtidos são iguais, use o operador de igualdade (=) normalmente.

Regex

Expressões regulares

Regular expressions ou somente "regex" são uma ferramenta que usamos para capturar padrões em strings. Veja um pequeno exemplo com a função `str_detect()` (que detecta se uma determinada string apresenta um certo padrão):

```
str_detect(c("banana", "BANANA", "maca", "nona"), pattern = "na")
```

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

Alguns caracteres tem significados especiais dentro de expressões regulares para que possamos fazer casamentos (*matchings*) mais interessantes.

Os caracteres `.`, `^` e `$` casam respectivamente com qualquer caractere, o início de uma string e o final de uma string.

EX: Mude o valor do argumento `pattern` no código acima para que a expressão dê match com qualquer string que tenha como segunda letra um `a` minúsculo.

Caracteres especiais (o retorno)

Mas o que acontece se quisermos dar match em um ponto? Ou em um cifrão? Assim como fizemos na primeira parte da aula, usando a `\` podemos escapar um caractere também em regex!

Mas tem um problema... Se usarmos `\.` para tentar dar match em um ponto, o R vai escapar a barra e assim obter somente um ponto (concluíndo erroneamente que falta alguma coisa no padrão). Precisamos então escapar **duas vezes**!

```
str_detect(c("ba.ana", "BANA.A", "ma.a", "n.na"), pattern = "a\\.a")
```

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

EX: Como faríamos para dar match em um padrão já escapado (como "\\.")? Dê match em um caractere de nova linha e em um caractere de tabulação.

REGRA GERAL: Quando em dúvida, divida ou multiplique o número de barras por 2... Não é elegante, mas com o tempo você vai pegando o jeito.

Quantos e quais

Outra funcionalidade interessante do regex é a possibilidade de passar um número de vezes para um padrão se repetir. `+` indica que um padrão se repete uma ou mais vezes, `*` indica que um padrão se repete zero ou mais vezes, e `{m,n}` indica que um padrão se repete entre `m` e `n` vezes (variações importantes são `{m}`, `{,n}` e `{m,}`).

```
str_detect(c("oi", "oii", "oiii", "oiiii"), pattern = "oi{3,}e*")
```

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

Também importante são os marcadores de conjuntos. Tudo que estiver dentro de um `()` vai ser tratado como uma unidade indivisível; já colocando caracteres dentro de um `[]`, casamos com qualquer um deles.

```
str_detect(c("banana", "baNANA", "BAnana"), ".[Aa](na){2}")
```

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

Miscelânea

Alguns outros padrões de regex que podem se fazer úteis:

- `[a-z]`, `[A-Z]` e `[0-9]` casam com letras minúsculas, letras maiúsculas e números (é possível usá-los juntos)
- `[^abc]` casa com qualquer coisa **menos** `a`, `b` e `c`
- `a?` casa com 0 ou 1 `a`

```
str_detect(c("ba ana", "BANANA", "maca", "nona"), "[^Bn]a ?[ca]")
```

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

Se você estiver se sentindo um pouco perdido, não se preocupe: regex não é um assunto simples. Não fique com medo de consultar tutoriais ou *cheat sheets* na hora que bater uma dúvida!

O melhor recurso ao qual temos acesso quando programando em R é o comando a seguir, que mostra todos os padrões que o `stringr` aceita:

```
?stringi::stringi-search-regex
```

Até agora

- Regex é uma ferramenta que nos permite detectar padrões em strings
- Caracteres como `.`, `^` e `$` casam em situações especiais
- Quantificadores como `?`, `+`, `*` e `{m,n}` determinam quantas vezes um padrão casa
- `[]` e `()` determinam conjuntos de padrões

Exercício

Dado o corpus presente em `stringr::words`, crie expressões regulares que casem com as palavras que:

- Começam com 3 consoantes
- Têm 3 ou mais vogais em sequência
- Têm duas ou mais ocorrências onde uma vogal precede uma consoante

Dica: Usem `str_view()` com `match = TRUE` para ver todas as palavras retornadas pelo matching.

Mexer com strings é dífícil

Não entre em pânico

Essa seção tem "difícil" no nome, mas ela não é tão difícil assim. Não vou ensinar praticamente nenhum outro comando de regex, mas aqui você vai precisar entender como o regex se encaixa com as funções mais importantes do stringr.

Também não perca de vista que, apesar de os exemplos até agora terem sido em vetores de strings, na vida real você proavavelmente estaria aplicando essas funções nas colunas de uma tabela.

E se você precisar de um incentivo...

Substituição

Uma das tarefas mais comuns no tratamento de strings é a substituição de um padrão por outro. Para isso temos as funções `str_replace()` e `str_replace_all()` que substituem, respectivamente, o primeiro ou todos os padrões encontrados.

```
str_replace("banana", pattern = "na", replacement = "XX")
```

```
#> [1] "baXXna"
```

Uma funcionalidade destas (e outras) funções é a possibilidade de usar o padrão procurado na substituição usando referências. Simplesmente use padrões da forma `\\N` no `replacement` onde `N` é o índice de um `()`:

```
str_replace_all("banana", pattern = "(na)", replacement = "XX\\1")
```

```
#> [1] "baXXnaXXna"
```

EX: Dado um número de 11 dígitos, transforme-o em um CPF da forma `544.916.518-84`.

Extração

Com `str_extract()` e `str_extract_all()`, extraímos padrões de strings. Podemos usar isso para tirar de uma string apenas a parte de casa com um padrão:

```
pessoas <- c("João Silva", "Joana Lima", "Madonna")
str_extract(pessoas, pattern = "[:alpha:]+$")
```

```
#> [1] "Silva"   "Lima"    "Madonna"
```

```
str_extract_all(pessoas, pattern = "[A-Z]")
```

```
#> [[1]]
#> [1] "J" "S"
#>
#> [[2]]
#> [1] "J" "L"
#>
#> [[3]]
#> [1] "M"
```

Matching

Com `str_match()` e `str_match_all()` conseguimos quebrar strings em matrizes onde cada coluna é uma parte do match. No caso abaixo, a primeira coluna é o match todo, a segunda é o primeiro `()` e a terceira é o segundo `()`.

```
pessoas <- c("Silva, João", "Lima, Joana", "Madonna")
str_match(pessoas, pattern = "(.*), ([:alpha:]+)")
```

```
#>      [,1]           [,2]    [,3]
#> [1,] "Silva, João" "Silva" "João"
#> [2,] "Lima, Joana" "Lima"  "Joana"
#> [3,] NA            NA      NA
```

No código acima, `:alpha:` representa o conjunto dos caracteres alfabéticos (tanto com quanto sem acento). Isso resolve o problema no qual `[a-zA-Z]` não casaria com `João`.

EX: O que acontece quando quebramos as strings do vetor `pessoas` em todas as letras (usando `str_match_all()`)?

Quebra

Se quisermos quebrar uma string em certos pontos podemos usar `str_split()`. Essa função usa um padrão e divide uma string em um vetor de strings quebrando-a exatamente onde encontrar o padrão.

```
str_split("Você quer um vetor @?", pattern = " ")
```

```
#> [[1]]
#> [1] "Você"  "quer"  "um"    "vetor" "@?"
```

Através de `str_split_fixed()` podemos limitar o número máximo de quebras (mas aí voltamos a obter uma tabela):

```
str_split_fixed("Você quer um vetor @?", pattern = " ", n = 3)
```

```
#>      [,1]   [,2]   [,3]
#> [1,] "Você" "quer" "um vetor @?"
```

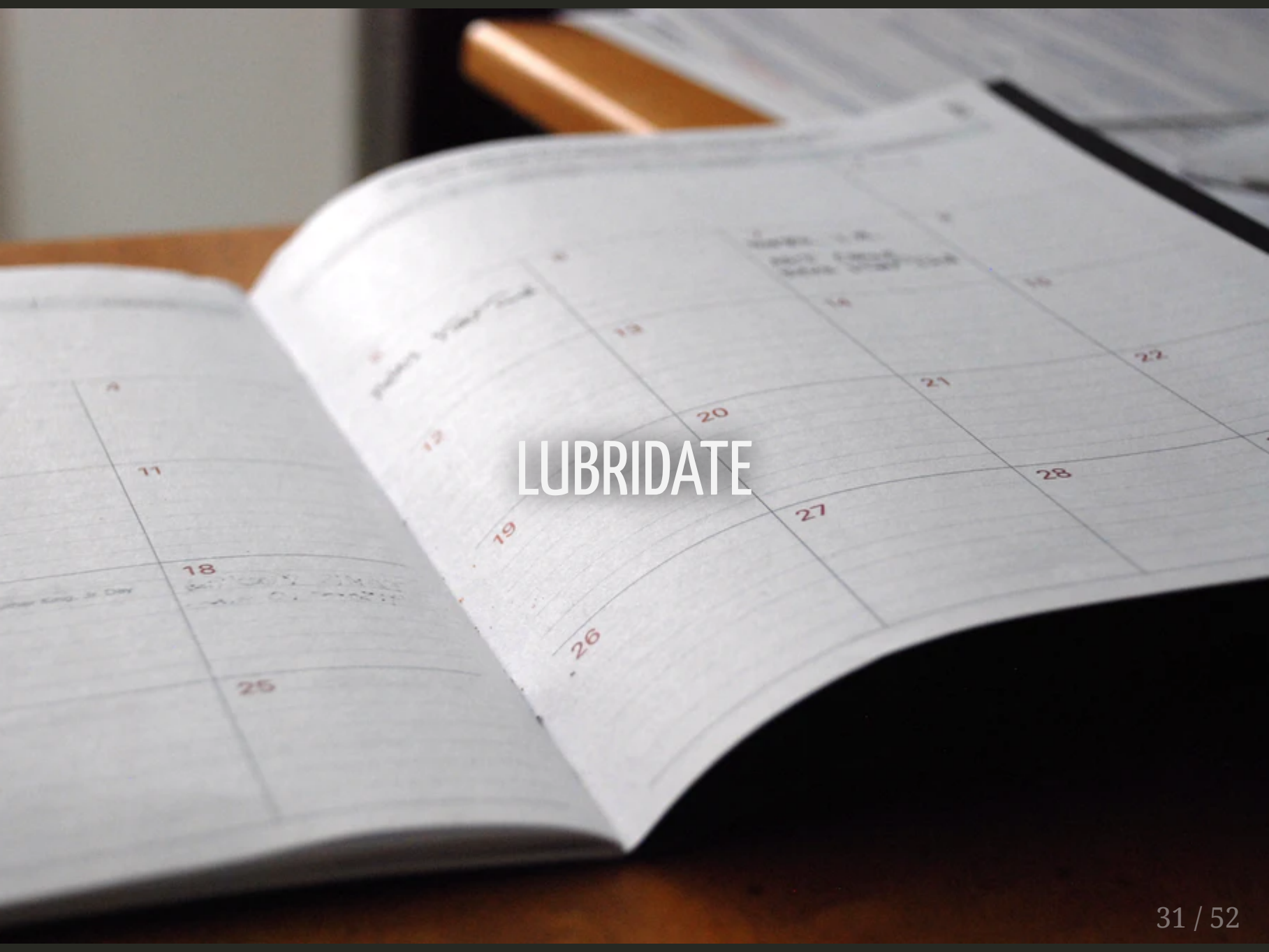
Até agora

- Para substituir um padrão por um texto, podemos usar `str_replace()`
- Para extrar um padrão de uma string, podemos usar `str_extract()`
- Para casar um padrão com um texto, podemos usar `str_match()`
- Para quebrar uma string onde ocorre um padrão, podemos usar `str_split()`

Exercício

Partindo de `stringr::sentences`, crie o vetor `no_the`, onde todas as ocorrências da palavra "the" (ou "The") são removidas (mas tendo em mente que as frases devem continuar começando com letra maiúscula)

Dica: Tente criar uma tabela com `stringr::sentences` para poder operar em colunas usando `dplyr::mutate()`. É possível resolver esse problema com apenas uma pipeline.


LUBRIDATE

O que são datas?

O básico

Quando tratamos de datas e horários em liguagens de programação, geralmente estamos falando sobre um **número** que será expresso como uma **string**.

```
library(lubridate)
now()
```

```
#> [1] "2019-05-05 17:25:24 -03"
```

Acima vemos que `now()` retorna o *datetime* (data-hora) atual*, incluindo o fuso horário no qual estamos. Mas se convertermos essa pseudo-string para `integer`, vemos que o número retornado não faz muito sentido...

```
as.integer(now())
```

```
#> [1] 1557087924
```

[*] Ele era atual quando eu fiz os slides...

Unix time

Unix time é a forma como todos os aparelhos eletrônicos modernos sabem que horas são. O número enorme que vimos no slide anterior represente quantos **segundos** se passaram desde o início do dia 01/01/1970.

```
as.integer(as_datetime("1970-01-01 00:00:00"))
```

```
#> [1] 0
```

EX: O que acontece quando convertemos uma data-hora anterior a 1970?

Agora que você sabe como datas e horas são armazenadas, já é possível deduzir por que esse é um tipo de dado problemático: nós pensamos em datas como um texto mas na verdade estamos trabalhando com um número!

```
now() + 2592000
```

```
#> [1] "2019-06-04 17:25:24 -03"
```

EX: O que de interessante aconteceu quando adicionamos 1 mês a `now()`?

ISO 8601

Como você já deve ter notado, todas as funções até agora retornavam as datas e horários na forma `AAAA-MM-DD HH:MM:SS`. Isso não é à toa! Este formato é...

> ... definido pela *Organização Internacional para Padronização* no padrão "ISO 8601" que define "elementos de dados e formatos de intercâmbio para representação e manipulação de datas e horas".

Desta forma, todos os outros formatos (inclusive o que usamos no dia-a-dia) precisa ser convertido antes de poder ser interpretado pelos aparelhos eletrônicos.

```
as_date("20/01/2018")
```

```
#> [1] NA
```

Agora já temos todos os ingredientes para um problemão... Datas e horas são escritos como strings, mas representados internamente como números e existe um padrão internacional para escrever datas, mas nenhum país o usa.

Até agora

- Existem dois tipos principais para trabalhar com datas e horários: `date` e `datetime`
- Com `today()` e `now()` podemos obter informações sobre a data-hora atual (incluindo fuso horário)
- Com `as_date()` e `as_datetime()` conseguimos converter uma data ou data-hora no padrão ISO 8601 para o formato compreendido pelo R

Exercício

Analise 3 conjuntos de funções: `as_date()` e `as_datetime()`, `make_date()` e `make_datetime()`, `date()`. O que diferencia esses grupos de funções? Por que não existe uma função `datime()`?

Dica: Para ler a documentação de uma função, execute um comando da forma `?nome_da_funcao`. Não se preocupe com os conceitos que você ainda não viu, você não precisa deles para entender essas funções.

Criando datas

Com datas bem formatadas

O jeito mais simples de criar uma data ou uma data-hora é com as primeiras funções que vimos: `as_date()` e `as_datetime()`. Para que elas funcionem, a entrada precisa estar praticamente 100% formatada no ISO 8601.

```
as_datetime("2018-01-20 00:02:05")
```

```
#> [1] "2018-01-20 00:02:05 UTC"
```

EX: Passe uma data para `as_datetime()` e uma data-hora para `as_date()`.

Como também vimos na seção anterior, outro jeito de criar datas é passando seus componentes individuais para `make_date()` e `make_datetime()`. Este método é particularmente útil quando tratando tabelas!

```
make_date(2018, 01, 20)
```

```
#> [1] "2018-01-20"
```

Fugindo do ISO 8601

Se quisermos criar uma data (sem horário), podemos usar a família `dmy()`, `ymd()`, `mdy()` e assim por diante... Elas procuram os campos (*day*, *month* e *year*) na ordem em que a respectiva letra aparece no nome da função.

```
dmy("20/01/2018")
```

```
#> [1] "2018-01-20"
```

EX: O que acontece se passarmos o número `20012018` para a função acima?

Para data-horas, a lógica é a mesma: `dmy_hms()`, `ymd_hms()`, etc. Agora as letras depois do sublinhado representam *hour*, *minute* e *second*.

```
dmy_hms("20/01/2018 12:02:50")
```

```
#> [1] "2018-01-20 12:02:50 UTC"
```

EX: E se não quisermos especificar os minutos ou segundos de um datetime?

Fusos horários

Um componente importante de datetimes que ainda não abordamos diretamente são os fusos horários. Praticamente todas as funções de criação de datas têm um argumento `tz` que nos permite especificar o fuso.

```
t1 <- dmy_hms("01/06/2015 12:00:00", tz = "America/New_York")
t2 <- dmy_hms("01/06/2015 13:00:00", tz = "America/Sao_Paulo")
t1 == t2
```

```
#> [1] TRUE
```

EX: Crie um vetor `c(t1, t2)`. O que acontece quando você o imprime?

Para trocar o fuso de um datetime, basta usar `with_tz()`:

```
with_tz(t1, tzone = "Australia/Lord_Howe")
```

```
#> [1] "2015-06-02 02:30:00 +1030"
```

EX: Dê uma olhada na lista de fusos presentes em `OlsonNames()`.

Até agora

- Com `as_date()` e `as_datetime()` podemos criar datas a partir do ISO 8601
- Com `make_*()` podemos criar datas a partir de seus componentes
- Com as famílias `ymd()` e `ymd_hms()` podemos criar datas a partir de qualquer formato
- Podemos atribuir fusos a data-horas com o argumento `tz`

Exercício

Partindo do vetor de strigs `vt`, obtenha a data-hora `t`. Você deve fazer isso de duas formas diferentes: uma deve usar somente o pacote `lubridate` e a outra deve usar o pacote `stringr` também.

```
vt <- c("2015", "31", "03", "02", "59")
t <- ymd_hm("2015-03-31 02:59")
```

Mexer com datas é fácil

Componentes

Depois que aprendemos a construir datas, entender seus componentes é uma tareafa razoavelmente simples. Usando os nomes em inglês das diferentes unidades de medida de uma data-hora, podemos extrair cada unidade separadamente.

```
dt <- ymd_hms("2016-07-08 12:34:56")
c(year(dt), month(dt), day(dt), hour(dt), minute(dt), second(dt))
```

```
#> [1] 2016    7    8   12   34   56
```

Além destas funções básicas, também temos acesso a algumas variações. `yday()` nos dá o dia do ano, enquanto `wday()` nos dá o dia da semana (1 = domingo e 7 = sábado).

```
c(yday(dt), wday(dt))
```

```
#> [1] 190   6
```

EX: Para que servem os argumentos de `wday()`?

Componentes (cont.)

Uma propriedade interessante dos componentes é que podemos atribuir valores diretamente a eles. Basta usar o operador de atribuição (`<-`) na seleção de um componente:

```
year(dt) <- 2020
dt
```

```
#> [1] "2020-07-08 12:34:56 UTC"
```

EX: Tente atribuir um valor inválido (maior que `31`) para `day(dt)`

Também é possível arredondar uma data-hora para o componente mais próximo: use `roud_date()` e passe o nome de um componente para o argumento `unit`. Se você precisar dos operadores de teto e chão, eles também estão disponíveis (`floor_date()` e `ceiling_date()`).

```
round_date(dt, "day")
```

```
#> [1] "2020-07-09 UTC"
```

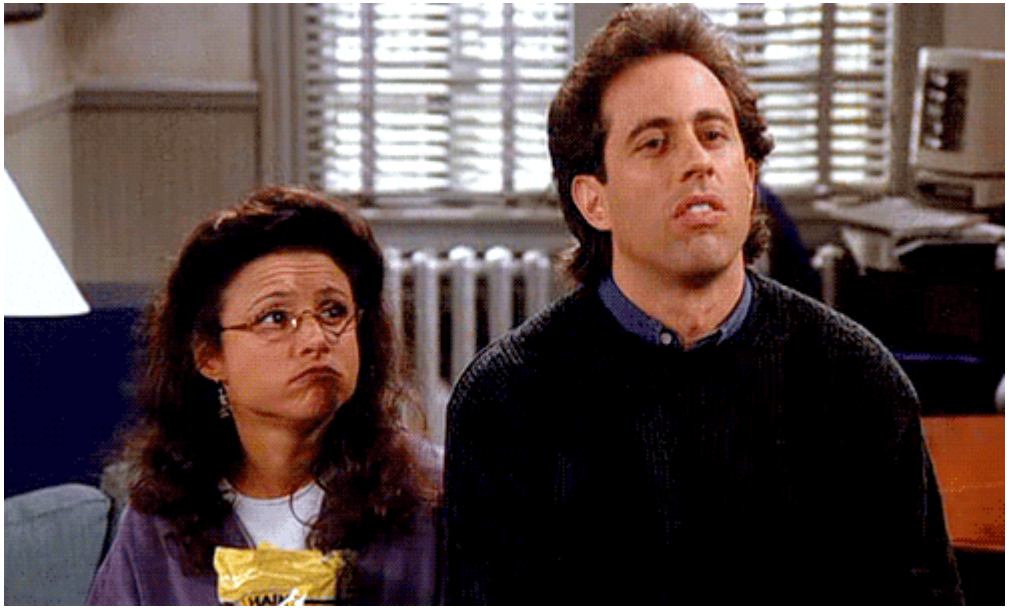
Mexer com datas é difícil

Talvez entre em pânico

Como você deve ter notado, a seção passada foi bastante curta e nem teve uma recapitulação... Foi esse o caso porque a utilidade real do pacote lubridate não está em extrair os componentes das data-horas (apesar de as vezes isso vir a calhar).

O que veremos nesta seção é a real vantagem do pacote, mas infelizmente não é um assunto tão simples, então pode ser que você fique meio perdido. Isso é normal e demora um bom tempo até que as coisas comecem a fazer sentido.

E se você precisar de um incentivo...



Durações

No R tradicional, mexer com a diferença entre dois objetos data-hora pode ser algo muito complicado e de comportamento imprevisível, mas o `lubridate` nos fornece uma interface consistente com `duration` que sempre retorna a duração em segundos.

```
as.duration(today() - dmy("28/12/1995"))
```

```
#> [1] "736905600s (~23.35 years)"
```

Se quisermos construir uma duração, basta utilizar as funções de componente que já aprendemos no plural e com um `d` na frente:

```
dyears(1) + dweeks(12) + dhours(15)
```

```
#> [1] "38847600s (~1.23 years)"
```

EX: Encontre a data de amanhã usando `today()` e um construtor de duração.

Períodos

Como nem sempre queremos diferenças de tempo e aritmética com datas resumidas a uma duração em segundos, o `lubridate` nos fornece o conceito de `periods` (durações que são legíveis para um humano). Seus contrutures são os componentes no plural:

```
years(1) + weeks(12) + hours(15)
```

```
#> [1] "1y 0m 84d 15H 0M 0S"
```

A maior diferença entre durações e períodos aparece quando lidamos com as variações naturais no comprimento das unidades temporais. Veja por exemplo o que acontece quando adicionamos `dyears(1)` e `years(1)` a um ano bissexto:

```
c(ymd("2016-01-01") + dyears(1), ymd("2016-01-01") + years(1))
```

```
#> [1] "2016-12-31" "2017-01-01"
```

EX: Onde mais esse tipo de diferença poderia aparecer?

Intervalos

Para complicar ainda mais o que acabamos de ver, imagine que precisamos determinar quantos dias cabem em um mês. Isso naturalmente depende porque tem todo mês tem o mesmo número de dias...

```
months(1) / days(1)
```

```
#> estimate only: convert to intervals for accuracy
```

```
#> [1] 30.4375
```

Para isso temos o conceito de `interval`, uma duração com um ponto de início. Usando o operador infixo `%--%`, determinamos um intervalo e assim fica fácil de obter um resultado preciso:

```
(today() %--% (today() + months(1))) / days(1)
```

```
#> [1] 31
```

EX: `(today() %--% (today() + years(1))) / months(1)` funciona?

Até agora

- Usando componentes (`day()`, `month()`, `year()`, etc.) podemos trabalhar com as partes de uma data-hora
- Com durações podemos realizar operações com diferenças de tempo
- Com períodos temos acesso a durações em uma forma mais legível e interpretável
- Com intervalos podemos dar um ponto inicial a uma operação temporal

Exercício

Partindo de `lubridate::lakers`, determine, em média, quanto tempo o Lakers (`team == "LAL"`) demora para arremessar a primeira bola (`etype == "shot"`) no primeiro período (`period == 1`).

Dicas: Lembre-se da aula de `dplyr`! É possível resolver esse exercício com apenas uma pipeline (na qual precisa haver apenas um `mutate()`). Entenda a função `ms()` (ela não é o que parece).

OBRIGADO!