R PARA CIÊNCIA DE DADOS: STRINGR/LUBRIDATE/PURRR

C. Lente + Curso-R

2018-02-02

Índice

O que veremos hoje:

- Stringr
    - Mexer com strings é fácil
    - Regex
    - Mexer com strings é dífícil
- Lubridate
    - Criando datas
    - Mexer com datas é fácil
    - Mexer com datas é difícil
- Purrr
    - O que são listas?
    - Iterações simples
    - Iterações avançadas


STRINGR

Mexer com strings é fácil

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?

Caracteres especiais

Usando uma barra oblíqua para a esquerda, podemos criar caracteres especiaias. Esta técnica se chama *escaping*.

```
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"
```

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.

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
```

Mexer com strings é dífícil

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"
```

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 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

Criando 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 17:32:18 -03"
```

O formato padrão de `date` e `datetime` é `AAAA-MM-DD HH:MM:SS`. Desta forma, não conseguimos converter uma data brasileira para o tipo `date` por padrão.

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

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

EX: Qual é o formato esperado por `as_date()`? Rode `today()`.

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()`.

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

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).


PURRR

O que são listas?

Listas

Listas são objetos muito semelhantes a vetores, com a diferença de que elas não precisam ser homogêneas. Isso quer dizer que os elementos de uma lista podem ter qualquer tipo (mesmo que eles não sejam iguais entre si).

```
l <- list(
  um_numero = 123, um_vetor = c(TRUE, FALSE, TRUE),
  uma_string = "abc", uma_lista = list(1, 2, 3))
str(l)
```

```
#> List of 4
#>  $ um_numero : num 123
#>  $ um_vetor  : logi [1:3] TRUE FALSE TRUE
#>  $ uma_string: chr "abc"
#>  $ uma_lista :List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

EX: Rode o comando acima sem a função `str()`.

Indexação

Para acessar os elementos de uma lista precisamos tomar cuidado com a diferença entre `[]` e `[[]]`. O primeiro acesa uma posição, enquanto o segundo acessa um elemento.

```
l[3]
```

```
#> $uma_string
#> [1] "abc"
```

```
l[[3]]
```

```
#> [1] "abc"
```

Uma funcionalidade interessante é a composição de `[[]]` caso precisemos de indexações mais profundas, ou seja, acessar os elementos de uma lista que é por sua vez um elemento de uma lista.

EX: Acesse o segundo elemento do quarto elemento de `l` com `[[]]`.

O pacote purrr

As funções mais básicas do `purrr` facilitam nossas vidas quando estamos trabalhando com listas. `pluck()` é uma abstração de `[[]]`, enquanto `set_names()` nos ajuda a atribuir nomes para os elementos de uma lista.

```
library(purrr)
pluck(l, "uma_lista", 2)
```

```
#> [1] 2
```

```
l <- set_names(l, c("um", "dois", "três", "quatro"))
str(l)
```

```
#> List of 4
#>  $ um    : num 123
#>  $ dois  : logi [1:3] TRUE FALSE TRUE
#>  $ três  : chr "abc"
#>  $ quatro:List of 3
#>   ..$ : num 1
#>   ..$ : num 2
#>   ..$ : num 3
```

Até agora

- `list` (mais conhecido como "lista") é uma estrutura de dados heterogênea
- Podemos indexar posições e elementos de uma lista com `[]` e `[[]]` respectivamente
- Podemos compor os operadores acima para encontrar exatamente o que desejamos
- Usando `pluck()` e `set_names()` conseguimos indexar e nomear listas com facilidade

Exercício

Crie uma lista com 3 níveis de profundidade e, depois de criada, atribua nomes para os elementos do terceiro nível (usando apenas `set_names()` e `pluck()`).

Dica: Uma operação da forma `pluck()<-` é plenamente possível.

Iterações simples

Laços

Laços (comumente conhecidos como *loops*) são a ferramenta que utilizamos para iterar ao longo de listas ou vetores. Suponha que temos a função e o vetor abaixo e que queremos aplicar a função a cada elemento do vetor.

```
soma_um <- function(x) { x + 1 }
obj <- 10:12
```

Utilizando um loop, isso é uma tarefa simples.

```
for (i in 1:3) {
  obj[i] <- soma_um(obj[i])
}
obj
```

```
#> [1] 11 12 13
```

EX: Reescreva o loop acima, mas agora considerado que `obj` é uma lista de 3 elementos.

Map

A função `map()` não passa de uma abstração de laços como o que acabamos de ver. Ela recebe um objeto (`.x`) e uma função (`.f`), aplicando esta a cada elemento to objeto (este pode ser tanto uma lista quanto um vetor).

```
obj <- map(obj, soma_um)
str(obj)
```

```
#> List of 3
#>  $ : num 11
#>  $ : num 12
#>  $ : num 13
```

Usando o argumento `...` de `map()` podemos passar argumentos para nossa função que permanecerão constantes ao longo de todas as iterações

```
soma_n <- function(x, n = 1) { x + n }
obj <- map(obj, soma_n, n = 3)
```

EX: O que acontece se passarmos uma lista nomeada para `map()`?

Achatamento

`map()` sempre retorna uma lista independente objeto recebido porque ela não pode assumir nada sobre o resultado. Se quisermos achatar os resultados só precisamos chamar uma função da família `map_***()` (onde `***` é a abreviação do tipo do objeto que deve ser retornado).

```
map_dbl(obj, soma_n, n = 3)
```

```
#> [1] 13 14 15
```

Os tipos possíveis são: `dbl` (números), `chr` (strings), `dfc` (*data frame columns*), `dfr` (*data frame rows*), `int` (inteiros), `lgl` (lógicos).

Alternativamente, se quisermos explicitar o achatamento ou fazê-lo sem a necessidade de um `map()`, podemos usar `flatten_***()`.

```
map(obj, soma_n, n = 3) %>% flatten_dbl()
```

```
#> [1] 13 14 15
```

Fórmulas

Por padrão, `map()` sempre substitui o primeiro arugmento da função pelos elementos do objeto iterado. Se quisermos mudar isso, podemos usar fórmulas.

```
map_dbl(obj, ~soma_n(x = 10, n = .x))
```

```
#> [1] 20 21 22
```

Como podemos ver, o construtor ~ declara que estamos usando uma fórmula, enquanto o *placeholder* `.x` serve para marcar onde devem ser colocados os elementos de `obj` a cada iteração.

```
obj2 <- list(list(1:3, 6:8), list(11:13, 16:18))
map(obj2, ~map_dbl(.x, mean)) %>% str()
```

```
#> List of 2
#>  $ : num [1:2] 2 7
#>  $ : num [1:2] 12 17
```

Map em paralelo

Se quisermos fazer uma iteração ao longo de dois vetores/listas, isso é perfeitamente possível usando `map2()` e a família `map2_***`. Desta vez temos dois *placeholders* (`.x` e `.y`), mas fora isso a função se comporta exatamente da mesma forma.

```
map2_dbl(1:3, 11:13, soma_n)
```

```
#> [1] 12 14 16
```

Se precisarmos iterar sobre mais que dois objetos, podemos usar `pmap()` e a família `pmap_***()`. Neste caso, a entrada deve ser uma lista onde cada elemento é um dos objetos sobre o qual devemos iterar.

```
soma_mn <- function(x, m, n) { x + m + n }
pmap_dbl(list(1:3, 11:13, 21:23), soma_mn)
```

```
#> [1] 33 36 39
```

Até agora

- `map()` é uma função que abstrai laços de forma a simplificar (e "pipeabilizar" iterações)
- `map2()` e `pmap()` generalizam `map()` para iterações paralelas
- Com os sufixos `_dbl()`, `_chr()`, `_lgl()` e assim por diante podemos achatar resultados
- Usando fórmulas e placeholders conseguimos definir funções lambda diretamente de dentro da chamada de `map()`

Exercício

Usando apenas duas chamadas de função (uma para `dplyr::tibble()` e outra para uma função do `purrr`), transforme a lista `l` na tabela `tl`:

```
t <- list(1:3, 11:13)
tl <- dplyr::tibble(col = 1:11, col1 = 2:12, col2 = 3:13)
```

Dica: Não se preocupe com os nomes das colunas, isso não é o importante.

Iterações avançadas

Condicionais

Uma coisa que é extremamente fácil de integrar a laços são condicionais. Não é raro precisarmos transformar apenas alguns dos elementos de uma lista ou vetor, tarefa que não pode ser cumprida com nenhuma das funções que vimos até agora.

```
for (i in seq_along(obj)) {
  if (obj[i]%%2 == 1) { obj[i] <- 2*obj[i] }
  else                { obj[i] <- obj[i] }
}
```

Para lidar com situações como a do laço acima é que o pacote `purrr` nos fornece a função `map_if()`. Esta função recebe, além de um objeto e uma função, um predicado (`.p`) que ela utilizará para verificar se a função deve ser aplicada naquela iteração.

```
eh_impar <- function(x) { x%%2 == 1 }
map_if(obj, eh_impar, ~2*.x) %>% flatten_dbl()
```

```
#> [1] 10 22 12
```

Condicionais (cont.)

Como você já deve estar imaginando, existe uma forma de simplificar ainda mais a expressão que acabamos de ver. `.p` também aceita fórmulas, permitindo que passemos funções lambda como predicados.

```
map_if(obj, ~.x%%2==1, ~2*.x) %>% flatten_dbl()
```

```
#> [1] 10 22 12
```

Por fim, se não quisermos passar uma condição propriamente dita para determinar em quais elementos deve ser aplicada a função, podemos também usar `map_at()` (que recebe em seu argumento `.at` um vetor de índices para aplicar `.f`).

```
map_at(obj, c(1, 3), ~2*.x) %>% flatten_dbl()
```

```
#> [1] 20 11 24
```

EX: Por que fui obrigado a utilizar `flatten_dbl()` até agora?

Acúmulo e redução

O último tipo de iteração que veremos se chama *accumulate* (ou *reduce*). Neste tipo de laço, aplicamos uma função em um elemento e no resultado acumulado da aplicação até aquele momento.

```
for (i in 2:length(obj)) {
  obj[i] <- obj[i-1] + obj[i]
}
obj
```

```
#> [1] 10 21 33
```

A função `accumulate()` nos permite reproduzir perfeitamente o laço acima. Aqui, o *placeholder* `.x` indica o resultado acumulado enquanto `.y` indica o elemento da iteração correspondente.

```
accumulate(obj, ~.x+.y)
```

```
#> [1] 10 21 33
```

Acúmulo e redução (cont.)

A função irmã de `accumulate()` se chama `reduce()`. Ela se comporta de forma muito semelhante, mas retorna apenas o último resultado obtido (jogando fora todos os passos intermediários).

```
reduce(obj, ~.x+.y)
```

```
#> [1] 33
```

`accumulate()` e `reduce()` não têm muitas variações, apenas as suas versões a direção "backward". Estas variações começam a iteração pela direita e não pela esquerda.

```
accumulate(obj, ~.x+.y, .dir = "backward")
```

```
#> [1] 33 23 12
```

EX: Por que, dentre todas as funções vistas até agora, estas retornam vetores naturalmente?

Até agora

- Com `map_if()` e `map_at()` podemos determinar apenas alguns elementos do vetor/lista de entrada onde aplicar a função
- Com `accumulate()` e `reduce()` conseguimos aplicar funções acumuladas nas entradas

Exercício

Partindo de `ggplot2::diamonds`, obtenha o máximo cumulativo das colunas numéricas sem usar `cummax()`. No final você deve obter uma tabela com as mesmas colunas que `ggplot2::diamonds` (inclusive com os mesmos nomes), mas onde as colunas numéricas agora representam os máximos cumulativos das suas versões originais.

Dicas: Você não precisará utilizar nenhum achatamento, só `dplyr::bind_cols()`. É perfeitamente possível escrever uma fórmula dentro da outra.

Miscelânea

Aqui deixo uma lista de outras funções interessantíssimas do `purrr` que foram cortadas desta aula por causa do tempo limitado:

- `keep()`/`discard()`: para filtrar elementos de listas
- `possibly()`: o melhor jeito de fazer um try-catch em R
- `transpose()`: transpõe listas
- `partial()`: pré-preenche os argumentos de uma função
- `walk()`: uma versão do `map()` para quando só precisamos dos efeitos colaterais
- `modify()`: um atalho para `x[] <- map(x, .f)`
- A família `is()`: para verificações extremamente estritas em objetos dos mais variados tipos
- `imap()`: um atalho para `map2(x, seq_along(x), ...)`
- `invoke()`: consegue invocar uma função com uma lista de argumentos
- `lift()`: um auxiliador para composição de funções
- `when()`: uma abstração de condicionais "pipeável"
- ...

OBRIGADO!