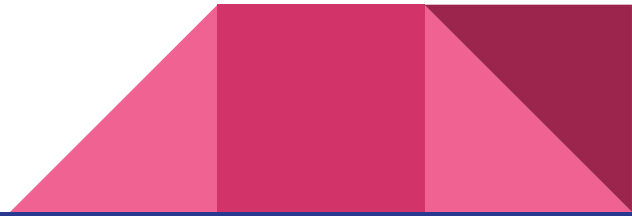
List/dics/set comps e gen expr

Live de python #18

Roteiro

- Functors, funções, sequências e lazy evaluation
- map
- filter
- list comp
- dict comp
- set comp
- gen expr



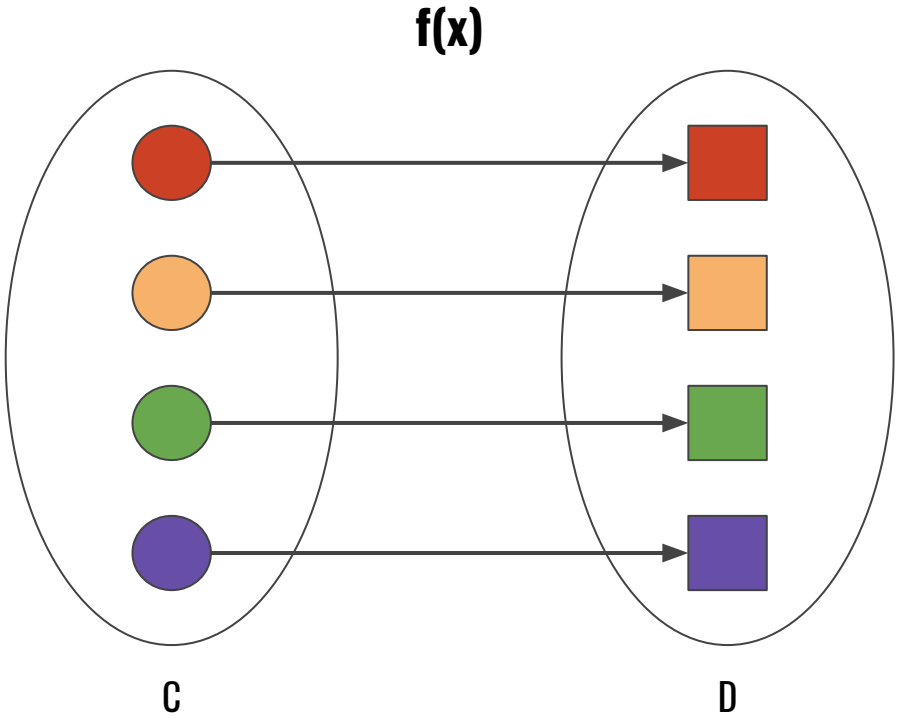
Functor

Na Teoria das categorias, é um **mapeamento** entre categorias que **preserva estruturas**. Os functores podem ser entendidos como homomorfismos na categoria de todas as categorias pequenas (ou seja, a categoria que tem como objetos todas as categorias compostas por objetos que são conjuntos).

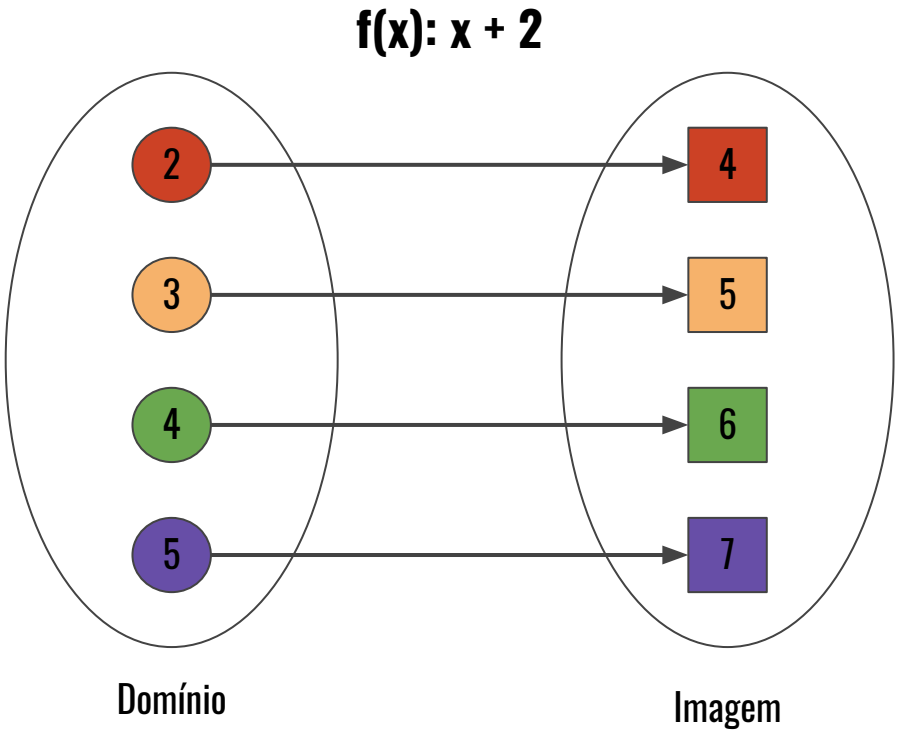
Ou seja:
associa para cada objeto x em um conjunto **C** um objeto F(x) em um conjunto **D**

(wikipedia - com algumas modificações)

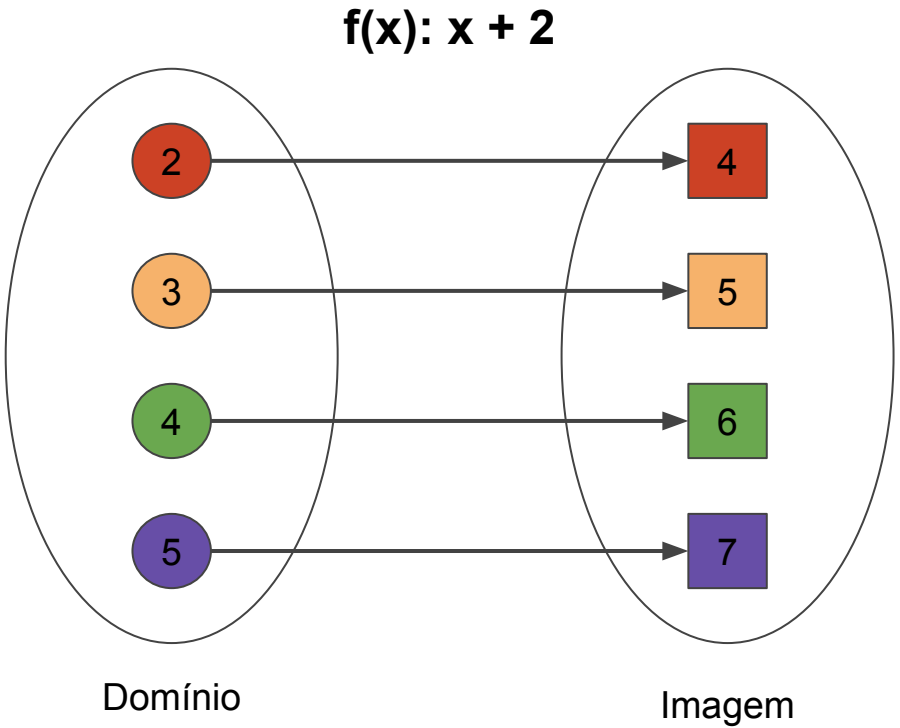
Funções, funções e funções



Funções, funções e funções

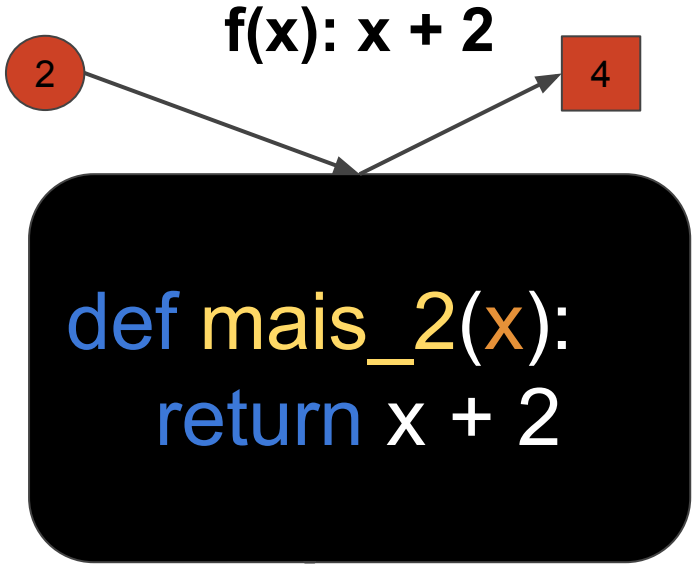


Funções, funções e funções



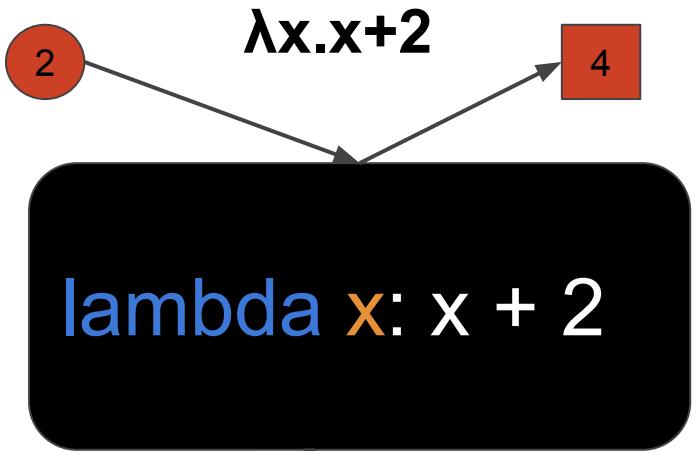

```
def mais_2(x):
    return x + 2
```

Aplicação de uma função simples



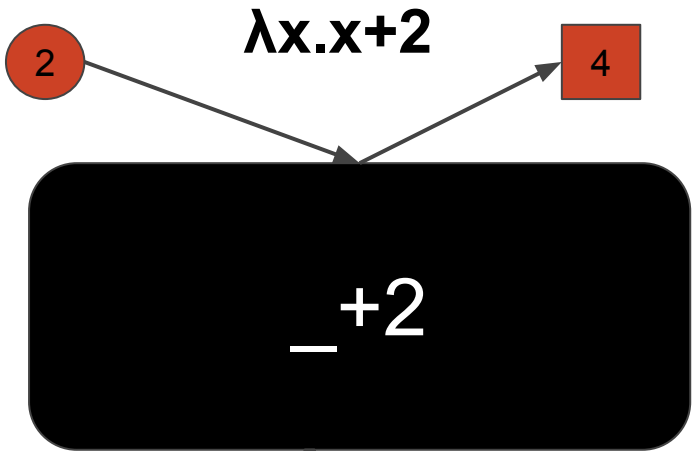

```
In [1]: def mais_2(x):
   ...:     return x + 2
   ...:

In [2]: mais_2(2)
Out[2]: 4
```

Aplicação de uma função anônima [0]



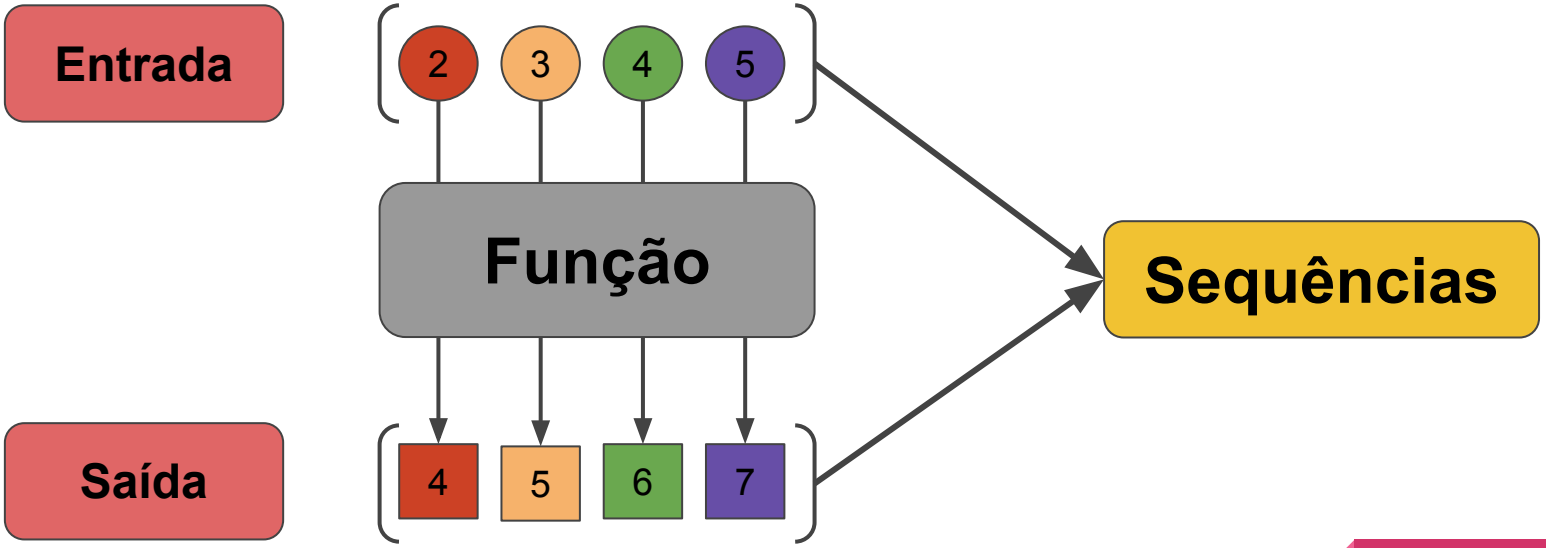

```
In [1]: (lambda x: x + 2)(2)
Out[1]: 4
```

Aplicação de uma função anônima [1]

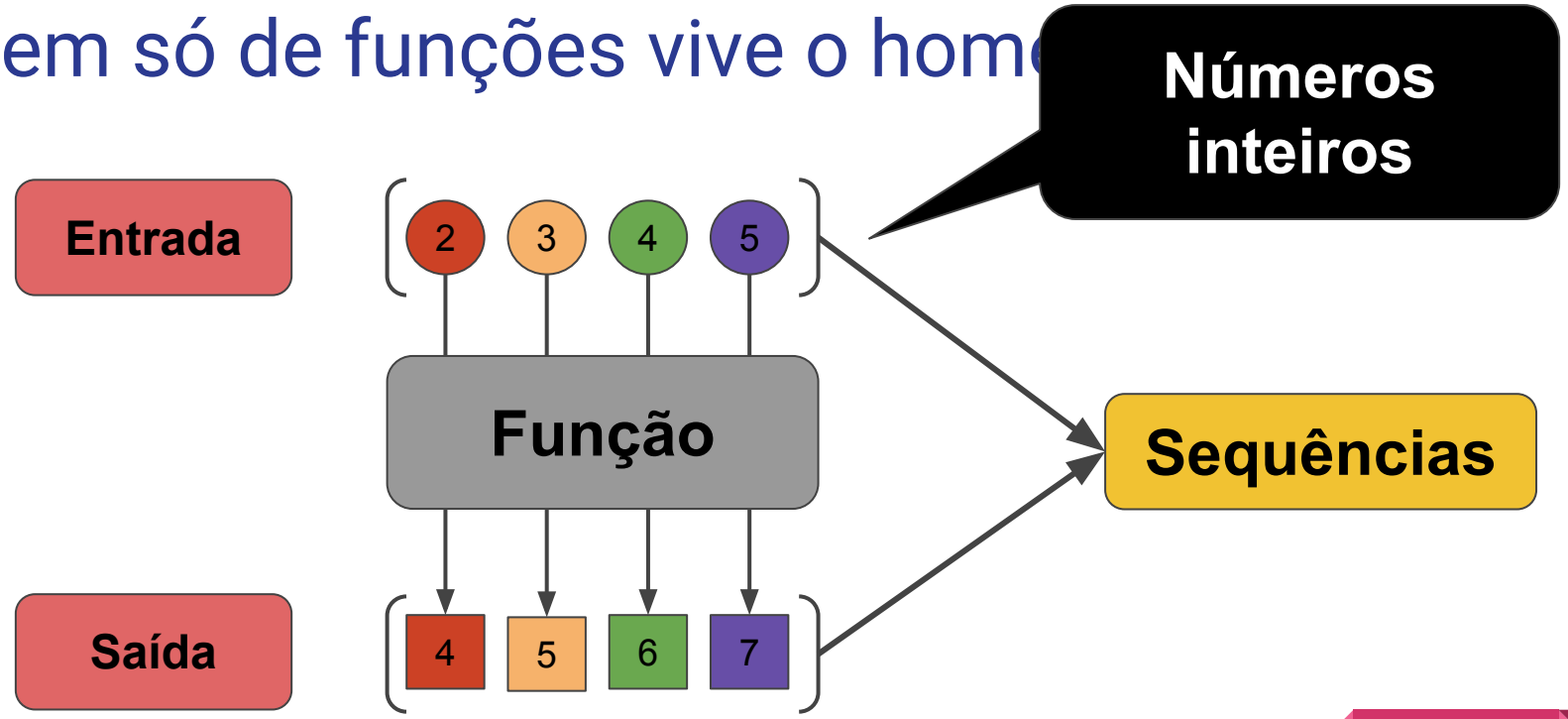



```
In [1]: from fn import _

In [2]: (_+2)(2)
Out[2]: 4
```

Nem só de funções vive o homem



Nem só de funções vive o hom



Lazy evaluation - Avaliação preguiçosa

São retornos (as vezes de funções) que só são computados quando chamados

Comumente são chamados pelo método __next__() pela função next()

Lazy evaluation - Geradores - Call-by-need

```
 2016-03-07 17:39:26 ★  Babbage in ~
○ → python -m memory_profiler  test.py
Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
     3   26.938 MiB    0.000 MiB   @profile
     4                             def vetor():
     5   70.750 MiB   43.812 MiB       return [x for x in range(1058**2)]


Filename: test.py

Line #    Mem usage    Increment   Line Contents
================================================
     7   70.750 MiB    0.000 MiB   @profile
     8                             def g_vetor():
     9   70.750 MiB    0.000 MiB       return (x for x in range(1058**2))
```

Lazy evaluation - Geradores - Call-by-need

```
2016-03-07 17:39:26 ★  Babbage in ~
○ → python -m memory_profiler test.py
Filename: test.py

Line #    Mem usage    Increment
================================
     3   26.938 MiB    0.000 MiB
     4
     5   70.750 MiB   43.812 MiB


Filename: test.py

Line #    Mem usage    Increment
================================
     7   70.750 MiB    0.000 MiB
     8
     9   70.750 MiB    0.000 MiB
```

```python
@profile
def vetor():
    return [x for x in range(1058**2)]

@profile
def g_vetor():
    return (x for x in range(1058**2))

#1119364
x = vetor()
y = g_vetor()
```

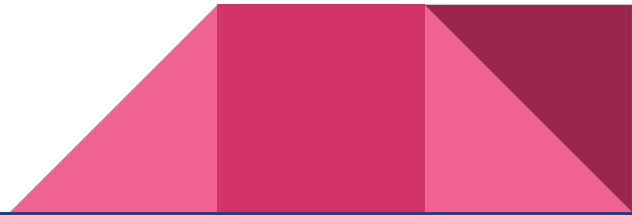
Map

Função com retorno preguiçoso, que dada uma função **f(x)**, aplicada a uma sequência **S**, retorna uma nova sequência lazy da aplicação de f(x) para todos os elementos de **S** formando assim uma nova sequência que vamos chamar de **Z**.

Ou seja:

map(f, S) -> Z => map(lambda x: **x*2**, [1,2,3]) -> [2, 4, 6]

OBS: a função (f(x)) só pode receber um único argumento



Map

Função com retorno preg **(x)**, aplicada a uma sequência **S**, retorna uma ão de f(x) para todos os elementos de **S** formand e vamos chamar de **Z**.

Aqui o operador aritmético é usado

Ou seja:

map(f, S) -> Z => map(lambda x: **x*2**, [1,2,3]) -> [2, 4, 6]

OBS: a função (f(x)) só pode receber um único argumento

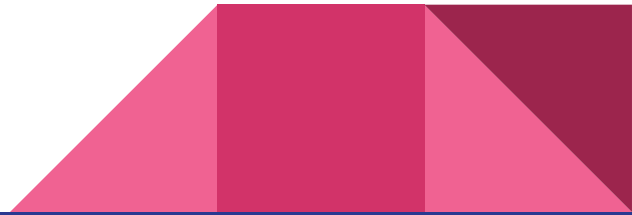
Filter

Função com retorno preguiçoso, que dada uma função **f(x)**, aplicada a uma sequência **S**, retorna uma nova sequência lazy da aplicação de f(x) para todos os elementos de **S** formando assim uma nova sequência que vamos chamar de **Z**.

Ou seja:

filter(f, S) -> Z => filter(lambda x: **x>2**, [1,2,3]) -> [3]

OBS: a função (f(x)) só pode receber um único argumento



Filter

Função com retorno preg**(x)**, aplicada a uma sequência **S**, retorna uma**ão de f(x) para todos os elementos de **S** formand**e vamos chamar de **Z**.

Aqui o operador lógico é usado

Ou seja:

filter(f, S) -> Z => filter(lambda x: **x>2**, [1,2,3]) -> [3]

OBS: a função (f(x)) só pode receber um único argumento

Map, pra que? Tenho comps!

	map	Comp
List	list(map(func, iter))	[func(x) for x in iter]
Gen	map(func, iter)	(func(x) for x in iter)
Set	set(map(func, iter))	{func(x) for x in iter}
Dict	dict(map(func, iter)	{func(x):func(y) for x,y in iter}

Filter, pra que? Tenho comps!

	map	Comp
List	list(filter(func, iter))	[x for x in iter if func(x)]
Gen	filter(func, iter)	(x for x in iter if func(x))
Set	set(filter(func, iter))	{x for x in iter if func(x)}
Dict	dict(filter(func, iter))	{x:y for x,y in iter if func(x) and func(y)}

Sintaxe das comps/expr

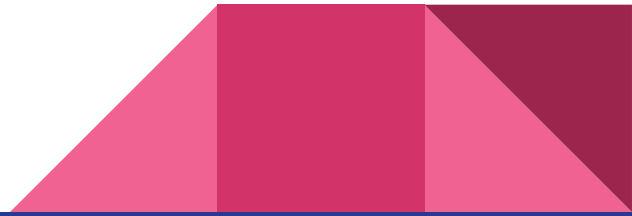
<Caráter de abertura>

<val> | <val + op_atitimético> | <func(val)>

<for val in sequência>

<if val op_logico>

<Caráter de fechamento>



Sintaxe das comps/expr

<Caráter de abertura>

<val> | <val + op_atitimético> | <fun

<for val in sequência>

<if val op_logico>

<Caráter de fechamento>

[-> listas;
(-> geradores;
{ -> conjuntos e dicionarios

Sintaxe das comps/expr

<Caráter de abertura>

<val> | <val + op_atitimético> | <func(val)>

<for val in sequênc

<if val op_logico>

<Caráter de fechamento>

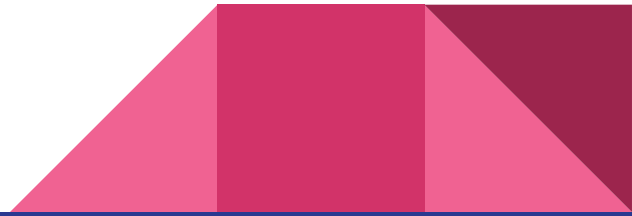
val -> usado na iteração
op -> (-, +, *, **, /, // ...)

Exemplos reais

[x for x in [1, 2, 3, 4]] -> [1, 2, 3, 4] (lista)

{x for x in [1, 2, 3, 4]} -> {1, 2, 3, 4} (conjunto)

(x for x in [1, 2, 3, 4]) -> (lazy obj)



CODE !!!

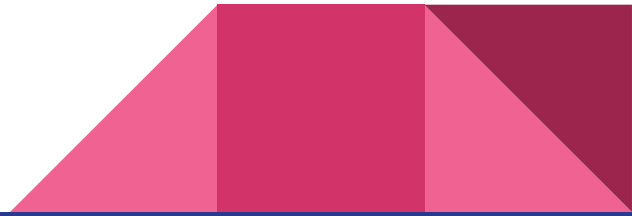
Exemplo 1 - Montando um baralho

Crie um baralho usando a combinação de duas sequências:

A: list(range(2, 11)) + 'Q J K A'.split()

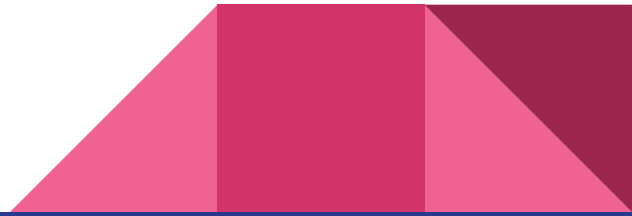
B: ['E', 'O', 'P', 'C']

C: ???



Exemplo 2 - Filtrando cartas de copas

Dada a sequência **C**, filtre somente as cartas em que o naipe seja de Copas, ou C



Exemplo 3 - Filtrando as cartas de rei (K)

Dada a sequência **C**, filtre somente as cartas em que o valor seja Rei, ou K