RUPY 2012

Strongly Dynamic Conference

Connecting Hemispheres

Iteração em Python: do básico ao genial

Luciano Ramalho
luciano@ramalho.org

@ramalhoorg

Comparando: C e Python

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    return 0;
}
```

```python
import sys

for arg in sys.argv:
    print arg
```

Turing.com.br

Iteração em Java

```
class Argumentos {
    public static void main(String[] args) {
        for (int i=0; i < args.length; i++)
            System.out.println(args[i]);
    }
}
```

```
$ java Argumentos alfa bravo charlie
alfa
bravo
charlie
```

Iteração em Java ≥ 1.5

ano: 2004

- *Enhanced* ***for*** (**for** melhorado)

```
class Argumentos2 {
    public static void main(String[] args) {
        for (String arg : args)
            System.out.println(arg);
    }
}
```

```
$ java Argumentos2 alfa bravo charlie
alfa
bravo
charlie
```

Iteração em Java ≥ 1.5

ano: 2004

- *Enhanced* ***for*** (**for** melhorado)

```
class Argumentos2 {
    public static void main(String[] args) {
        for (String arg : args)
            System.out.println(arg);
    }
}
```

ano: 1991

```
import sys

for arg in sys.argv:
    print arg
```

Exemplos de iteração

- Iteração em Python não se limita a tipos primitivos
- Exemplos
  - string
  - arquivo
  - Django QuerySet

```
>>> from django.db import connection
>>> q = connection.queries
>>> q
[]
>>> from municipios.models import *
>>> res = Municipio.objects.all()[:5]
>>> q
[]
>>> for m in res: print m.uf, m.nome
...
GO Abadia de Goiás
MG Abadia dos Dourados
GO Abadiânia
MG Abaeté
PA Abaetetuba
>>> q
[{'time': '0.000', 'sql': u'SELECT
"municipios_municipio"."id", "municipios_municipio"."uf",
"municipios_municipio"."nome",
"municipios_municipio"."nome_ascii",
"municipios_municipio"."meso_regiao_id",
"municipios_municipio"."capital",
"municipios_municipio"."latitude",
"municipios_municipio"."longitude",
"municipios_municipio"."geohash" FROM "municipios_municipio"
ORDER BY "municipios_municipio"."nome_ascii" ASC LIMIT 5'}]
```

demonstração: queryset é um iterável lazy

Python Data Model

dunder

Wednesday, December 12, 12

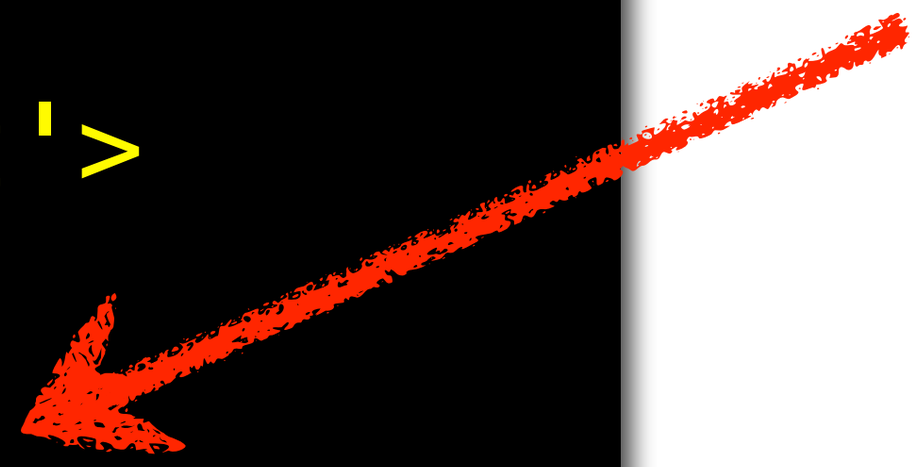

thunder


thunder


down under

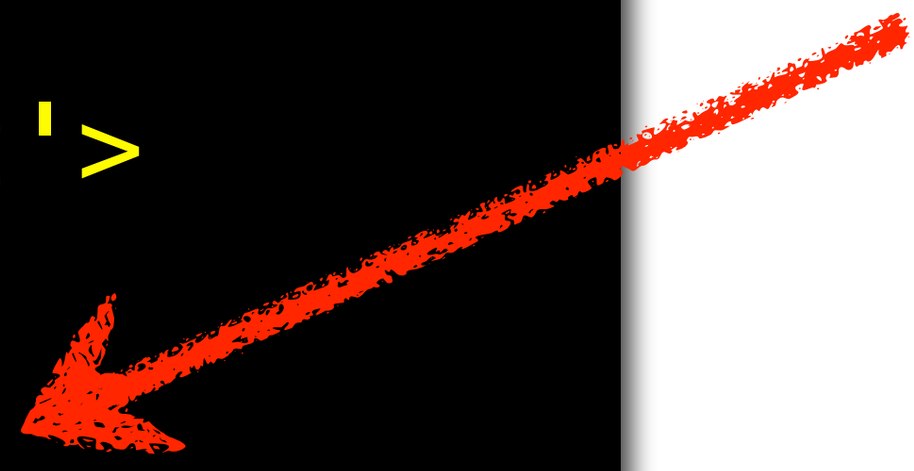

down under

```
>>> x = 7
>>> type(x)
<class 'int'>
>>> x * 6
42
>>> x.__mul__(6)
42
```



double underscore mul double underscore

```
>>> x = 7
>>> type(x)
<class 'int'>
>>> x * 6
42
>>> x.__mul__(6)
42
```



dunder mul!

atributos de um int

```
>>> dir(7)
['__abs__', '__add__', '__and__', '__bool__',
'__ceil__', '__class__', '__delattr__', '__divmod__',
'__doc__', '__eq__', '__float__', '__floor__',
'__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__', '__int__',
'__invert__', '__le__', '__lshift__', '__lt__',
'__mod__', '__mul__', '__ne__', '__neg__', '__new__',
'__or__', '__pos__', '__pow__', '__radd__', '__rand__',
'__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__',
'__rrshift__', '__rshift__', '__rsub__',
'__rtruediv__', '__rxor__', '__setattr__',
'__sizeof__', '__str__', '__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length',
'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

atributos de uma str

```
>>> dir('abc')
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__',
'__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'center',
'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha',
'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

atributos comuns a int e str

```
>>> sorted(set(dir(7)) & set(dir('abc')))
['__add__', '__class__', '__delattr__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__init__',
'__le__', '__lt__', '__mod__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__']
```

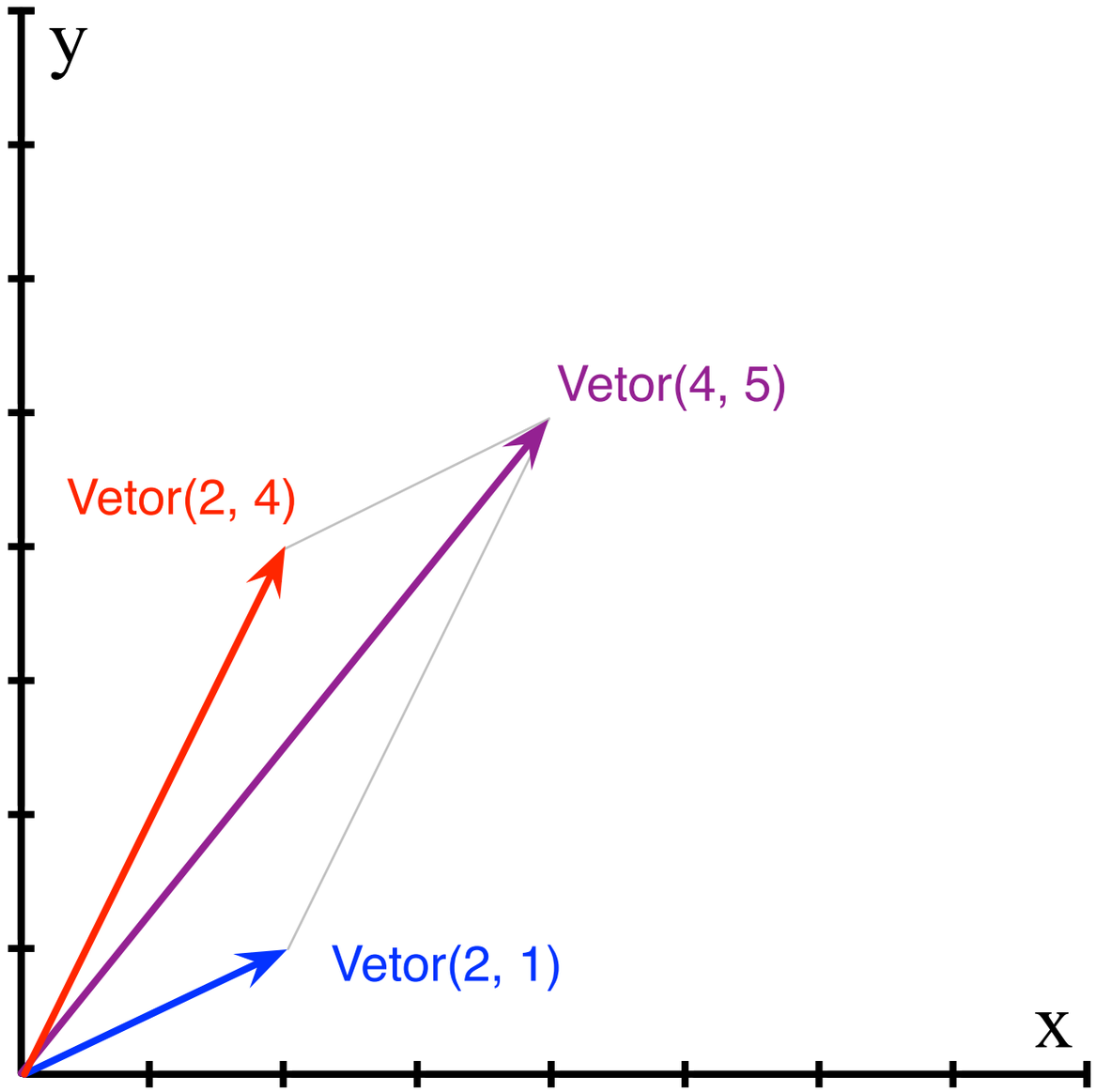
atributos comuns a str e list

```
>>> sorted(set(dir('abc')) & set(dir([])))
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmul__',
'__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'count', 'index']
```

Métodos dunder = “special methods”

- Protocolos genéricos, quase universais
- Úteis para muitas classes em muitos contextos
- **O interpretador Python** invoca estes métodos em determinados contextos
  - conversão, operadores, acesso a atributos e itens, iteração, ciclo de vida do objeto etc.

Exemplo: vetor (2d)



- Campos: x, y
- Métodos:
  - distancia
  - abs (distância até 0,0)
  - + (__add__)
  - * (__mul__) escalar

git oopy/exemplos/vetor.py

oficinas Turing

Vetor

```python
from math import sqrt

class Vetor(object):

    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def __repr__(self):
        return 'Vetor(%s, %s)' % (self.x, self.y)

    def distancia(self, v2):
        dx = self.x - v2.x
        dy = self.y - v2.y
        return sqrt(dx*dx + dy*dy)

    def __abs__(self):
        return self.distancia(Vetor(0,0))

    def __add__(self, v2):
        dx = self.x + v2.x
        dy = self.y + v2.y
        return Vetor(dx, dy)

    def __mul__(self, n):
        return Vetor(self.x*n, self.y*n)
```

```
>>> from vetor import Vetor
>>> v = Vetor(3, 4)
>>> abs(v)
5.0
>>> v1 = Vetor(2, 4)
>>> v2 = Vetor(2, 1)
>>> v1 + v2
Vetor(4, 5)
>>> v1 * 3
Vetor(6, 12)
```

oficinas Turing

Baralho polimórfico

Carta de baralho

```
class Carta(object):

    naipes = 'paus ouros copas espadas'.split()
    valores = '2 3 4 5 6 7 8 9 10 J Q K A'.split()

    def __init__(self, valor, naipe):
        self.valor = valor
        self.naipe = naipe

    def __repr__(self):
        return 'Carta(%r, %r)' % (self.valor, self.naipe)

    def __str__(self):
        return self.valor + ' de ' + self.naipe

    @classmethod
    def todas(cls):
        return [cls(v, n) for n in cls.naipes
                          for v in cls.valores]
```

oficinas Turing

Carta de baralho

```
>>> zape = Carta('4',
'paus')
>>> zape.valor
'4'
>>> zape
Carta('4', 'paus')
>>> monte = Carta.todas()
>>> len(monte)
52
>>> monte[0]
Carta('2', 'espadas')
>>> monte[-3:]
[Carta('Q', 'copas'),
 Carta('K', 'copas'),
 Carta('A', 'copas')]
```

```
class Carta(object):

    naipes = 'paus ouros copas espadas'.spl
    valores = '2 3 4 5 6 7 8 9 10 J Q K A'.

    def __init__(self, valor, naipe):
        self.valor = valor
        self.naipe = naipe

    def __repr__(self):
        return 'Carta(%r, %r)' % (self.valor, self.naipe)

    def __str__(self):
        return self.valor + ' de ' + self.naipe

    @classmethod
    def todas(cls):
        return [cls(v, n) for n in cls.naipes
                          for v in cls.valores]
```

Baralho polimórfico (demo)

```
from carta_ord import Carta

class Baralho(object):

    def __init__(self):
        self.cartas = Carta.todas()

    def __len__(self):
        return len(self.cartas)

    def __getitem__(self, pos):
        return self.cartas[pos]
```

oficinas Turing

Baralho polimórfico (final)

```
from carta_ord import Carta

class Baralho(object):

    def __init__(self):
        self.cartas = Carta.todas()

    def __len__(self):
        return len(self.cartas)

    def __getitem__(self, pos):
        return self.cartas[pos]

    def __setitem__(self, pos, valor):
        self.cartas[pos] = valor
```

Python Data Model: special methods

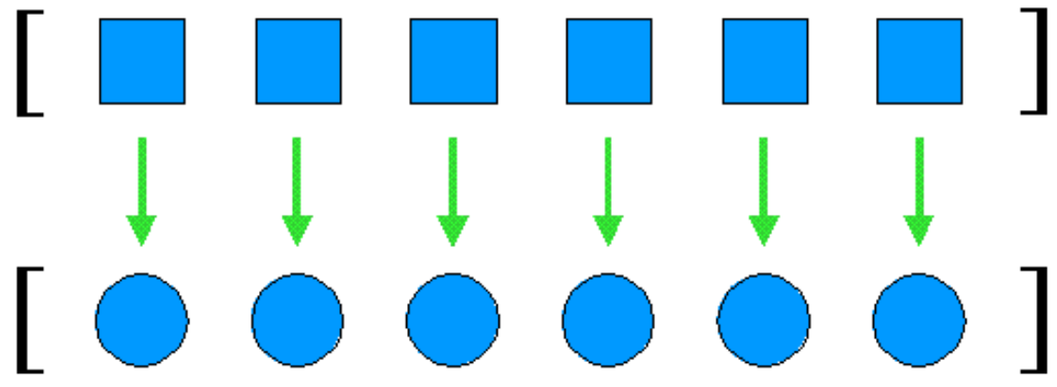
Em Python o comando **for** itera sobre... "iteráveis"

- Definicão preliminar informal:
  - "iterável" = que pode ser iterado
    - assim como:
      "desmontável" = que pode ser desmontado
- Iteráveis podem ser usados em outros contextos além do laço for

List comprehensions

- Expressões que consomem iteráveis e produzem listas

qualquer iterável



resultado: uma lista

```
>>> s = 'abracadabra'
>>> l = [ord(c) for c in s]
>>> [ord(c) for c in s]
[97, 98, 114, 97, 99, 97, 100, 97, 98, 114, 97]
```

Turing.com.br

≈ notação matemática de conjuntos

Set & dict comprehensions

- Expressões que consomem iteráveis e produzem sets ou dicts

```
>>> s = 'abracadabra'
>>> {c for c in s}
set(['a', 'r', 'b', 'c', 'd'])
>>> {c:ord(c) for c in s}
{'a': 97, 'r': 114, 'b': 98, 'c': 99, 'd': 100}
```

Tipos iteráveis embutidos

- basestring
  - str
  - unicode
- dict
- file
- frozenset
- list
- set
- tuple
- xrange

Funções embutidas que consomem iteráveis

- all
- any
- filter
- iter
- len
- map
- max
- min
- reduce
- sorted
- sum
- zip

Turing.com.br

Operações com iteráveis

- Desempacotamento de tupla
  - em atribuições
  - em chamadas de funções

```
>>> a, b, c = 'XYZ'
>>> a
'X'
>>> b
'Y'
>>> c
'Z'
>>> g = (n for n in [1, 2, 3])
>>> a, b, c = g
>>> a
1
>>> b
2
>>> c
3
```

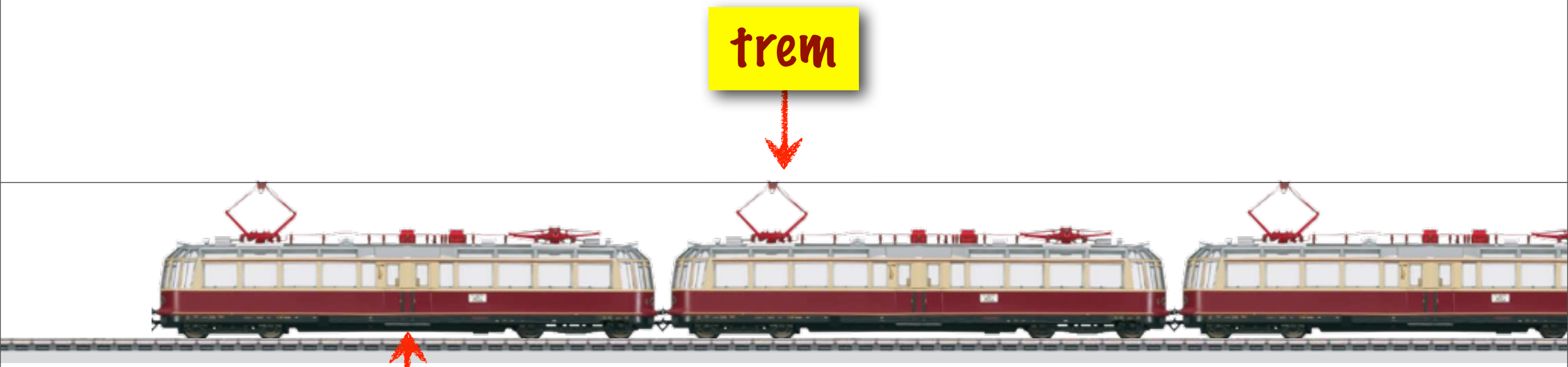
```
>>> def soma(a, b):
...     return a + b
...
>>> soma(1, 2)
3
>>> t = (3, 4)
>>> soma(t)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: soma() takes exactly 2 arguments (1 given)
>>> soma(*t)
7
```

@ramalhoorg

Em Python, um iterável é...

- Um objeto a partir do qual a função **iter** consegue obter um iterador.
- A chamada **iter(x)**:
  - invoca **x.__iter__()** para obter um iterador
  - ou, se **x.__iter__** não existe:
    - fabrica um iterador que acessa os itens de **x** sequenciamente: **x[0], x[1], x[2]** etc.

Trem:
uma sequência de vagões



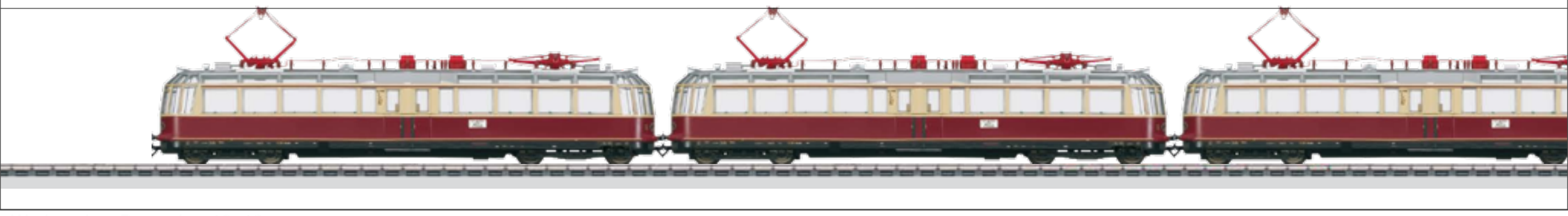
Curiosidade:
sequências eram chamadas "trains" na linguagem ABC, antecessora de Python

Turing.com.br

Trem:
uma sequência de vagões

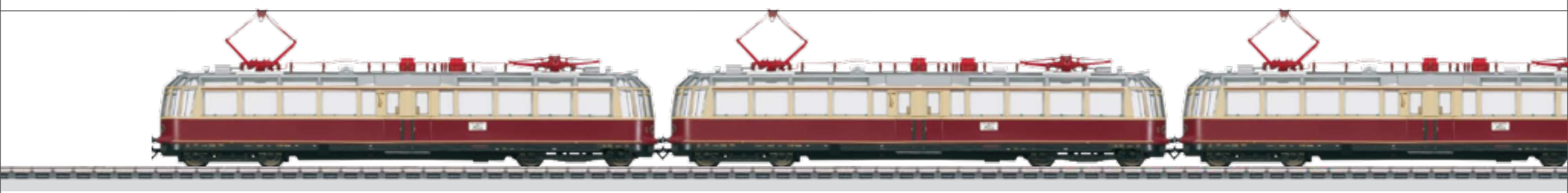
```
>>> t = Trem(4)
>>> len(t)
4
>>> t[0]
'vagao #1'
>>> t[3]
'vagao #4'
>>> t[-1]
'vagao #4'
>>> t[4]
Traceback (most recent call last):
  ...
IndexError: vagao inexistente [4]
```

```
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```



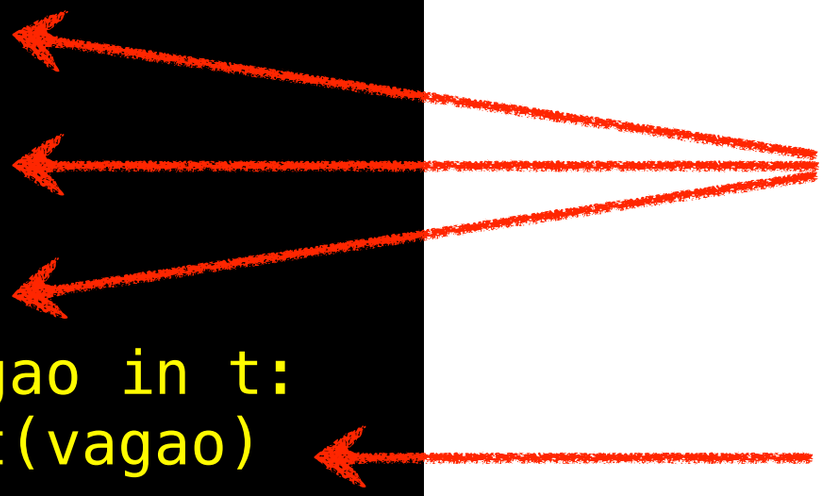
Protocolo de sequência

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __getitem__(self, pos):
        indice = pos if pos >= 0 else self.vagoes + pos
        if 0 <= indice < self.vagoes: # indice 2 -> vagao #3
            return 'vagao #%s' % (indice+1)
        else:
            raise IndexError('vagao inexistente [%s]' % pos)
```



Protocolo de sequência

```
>>> t = Trem(4)
>>> t[0]
'vagao #1'
>>> t[3]
'vagao #4'
>>> t[-1]
'vagao #4'
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```



__getitem__

__getitem__

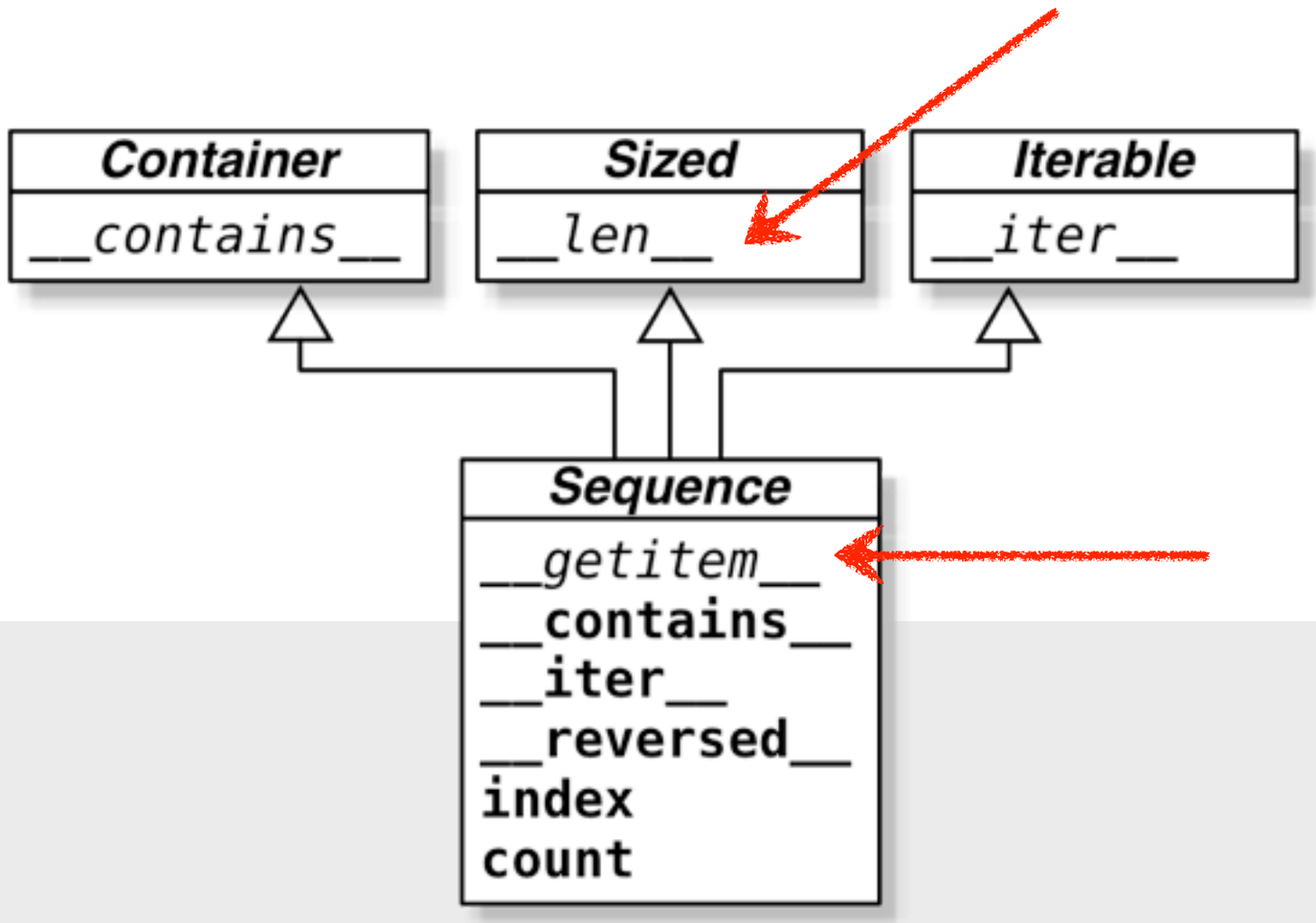
Protocolo de sequência

- protocolo é uma interface informal
  - pode se implementado parcialmente

```
class Trem(object):

    def __init__(self, num_vagoes):
        self.num_vagoes = num_vagoes

    def __getitem__(self, pos):
        indice = pos if pos >= 0 else self.num_vagoes + pos
        if 0 <= indice < self.num_vagoes: # indice 2 -> vagao #3
            return 'vagao #%s' % (indice+1)
        else:
            raise IndexError('vagao inexistente [%s]' % pos)
```

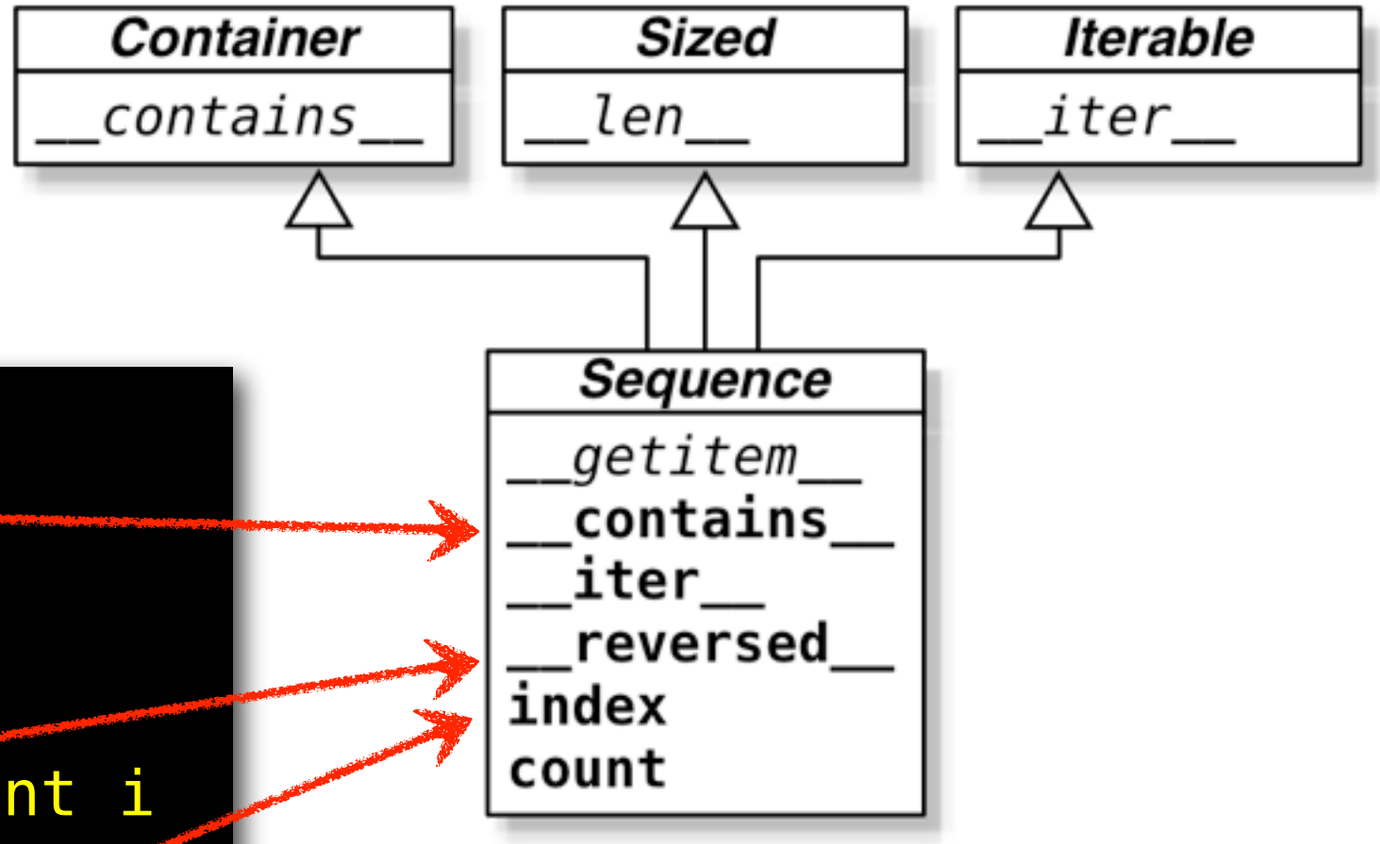
Sequence ABC

- Abstract Base Class




```
from collections import Sequence

class Trem(Sequence):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __len__(self):
        return self.vagoes

    def __getitem__(self, pos):
        indice = pos if pos >= 0 else self.vagoes + pos
        if 0 <= indice < self.vagoes: # indice 2 -> vagao #3
            return 'vagao #%s' % (indice+1)
        else:
            raise IndexError('vagao inexistente [%s]' % pos)
```

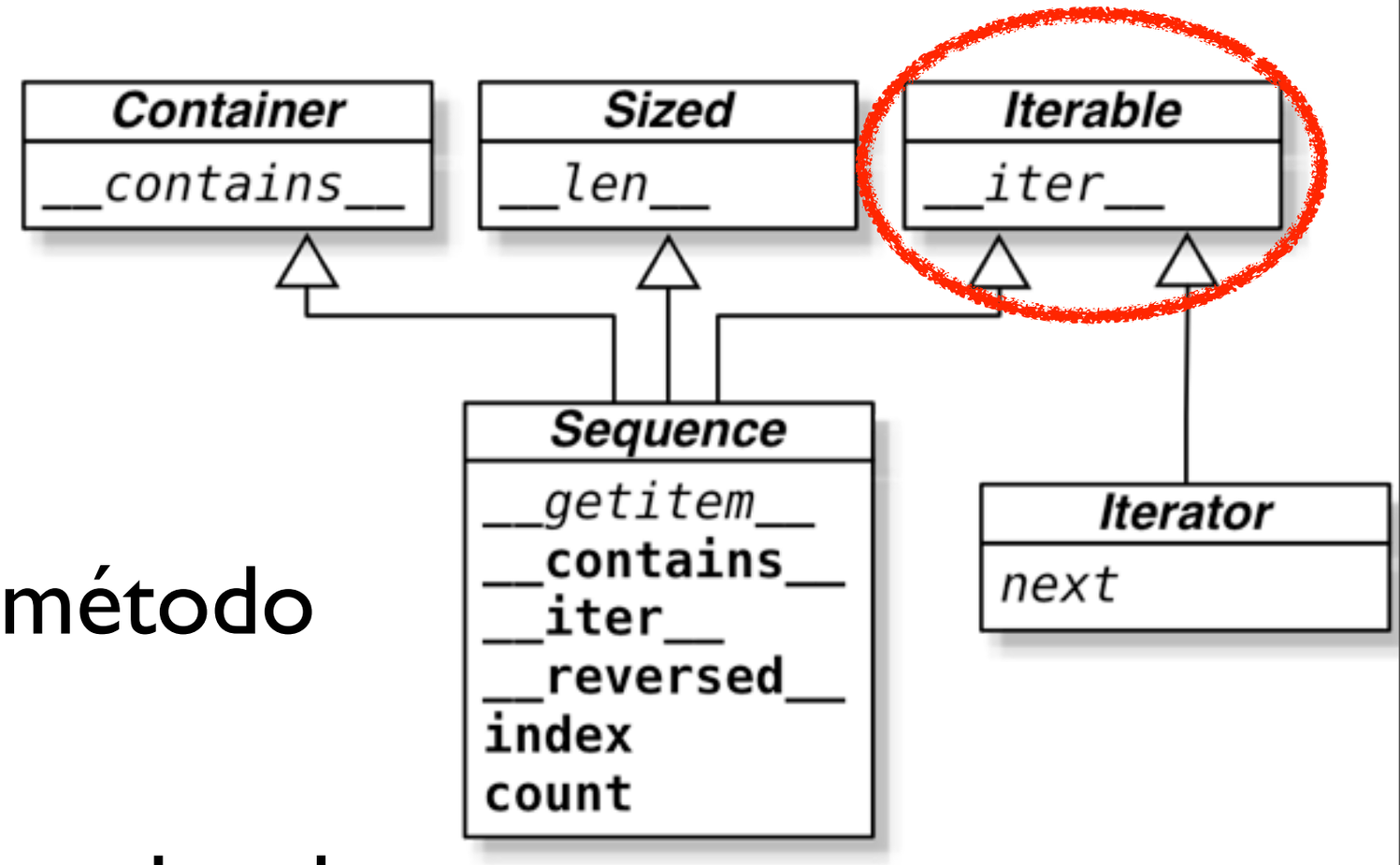
Herança de Sequence




```
>>> t = Trem(4)
>>> 'vagao #2' in t
True
>>> 'vagao #5' in t
False
>>> for i in reversed(t): print i
...
vagao #4
vagao #3
vagao #2
vagao #1
>>> t.index('vagao #2')
1
>>> t.index('vagao #7')
Traceback (most recent call last):
  ...
ValueError
```

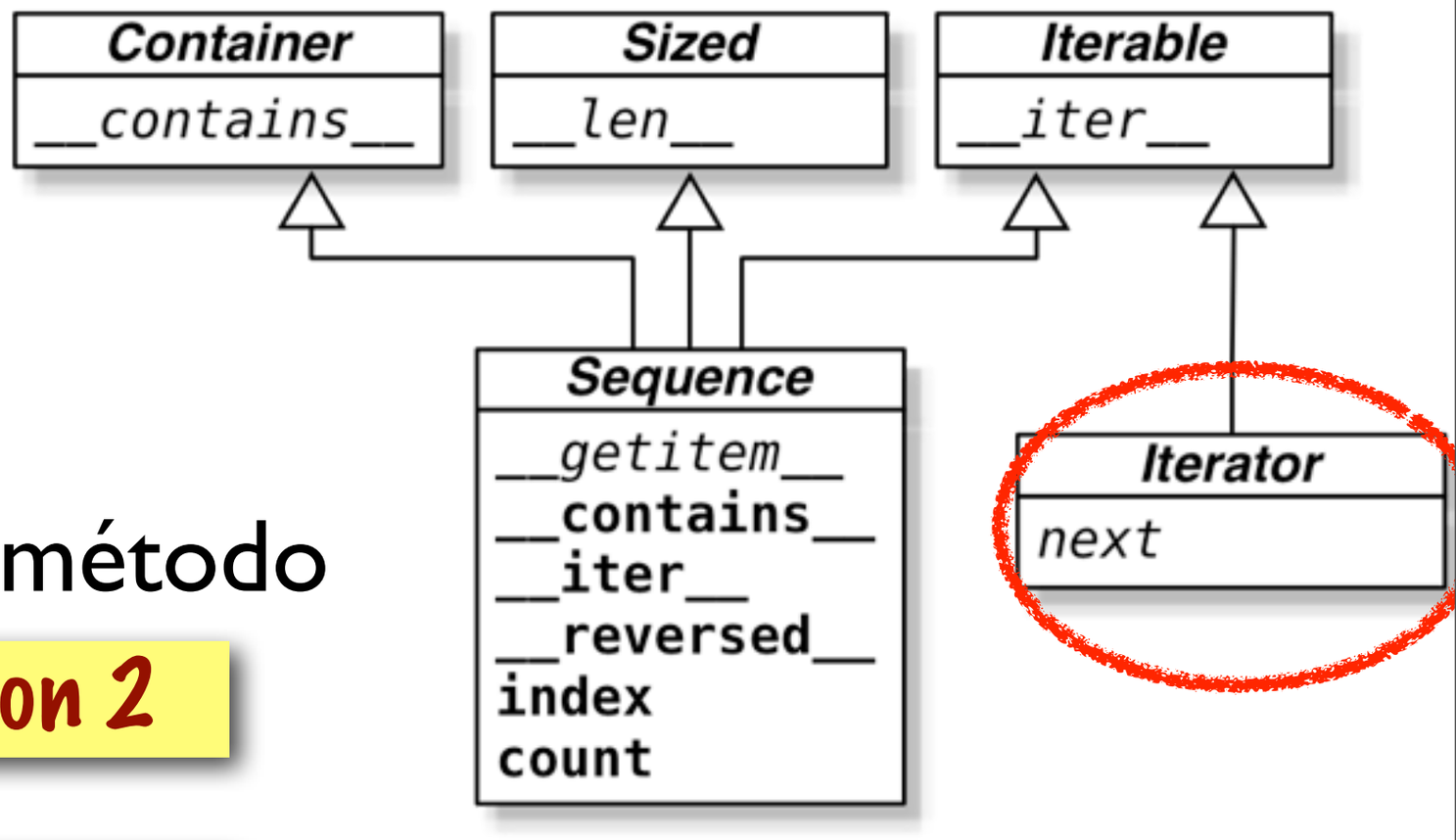
```
from collections import Sequence

class Trem(Sequence):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __len__(self):
        return self.vagoes

    def __getitem__(self, pos):
```

Interface Iterable



- Iterable provê um método **__iter__**
- O método **__iter__** devolve uma instância de **Iterator**
- Você normalmente não chama **__iter__**, quem chama é o Python
  - mas se precisar, use **iter(x)**

Interface Iterator



- Iterator provê um método **next**

  ou

  __next__

Python 2

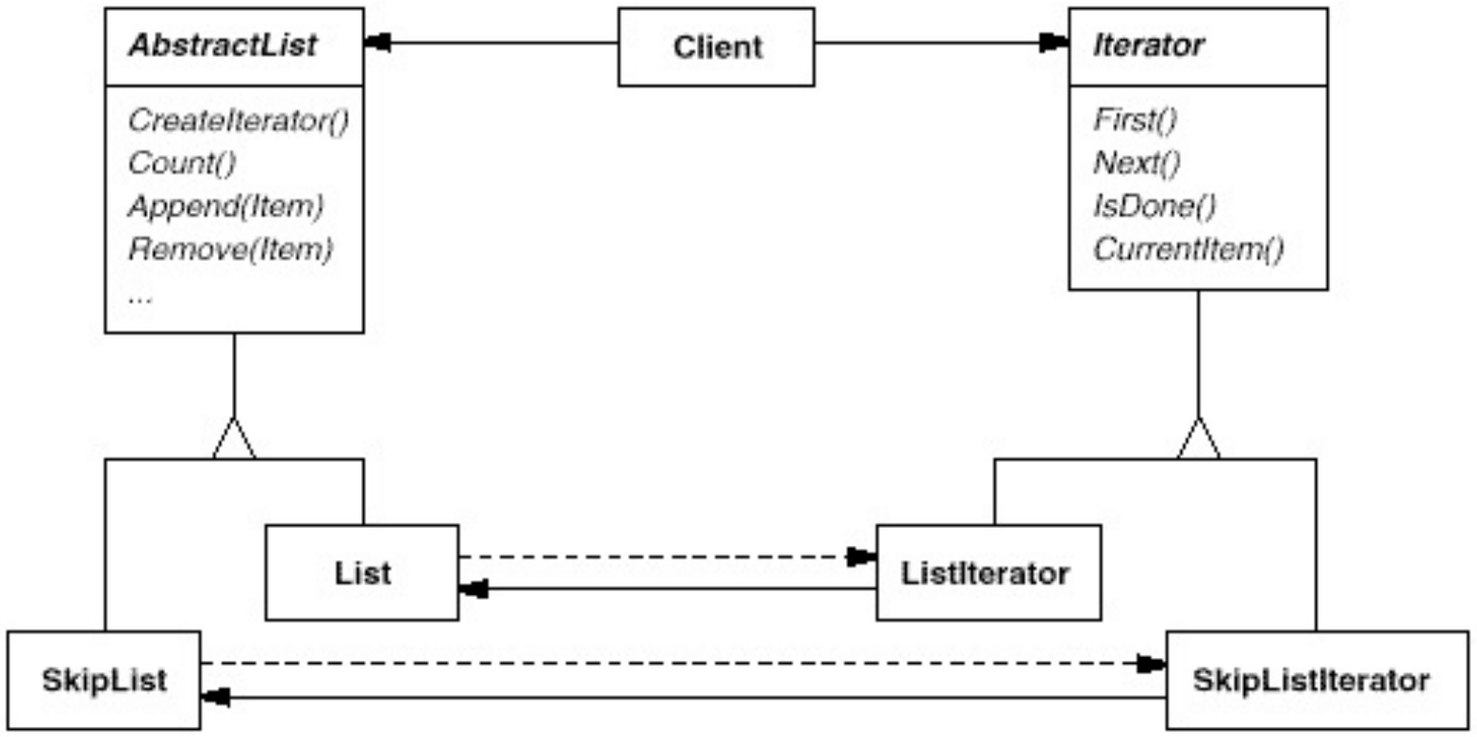
Python 3

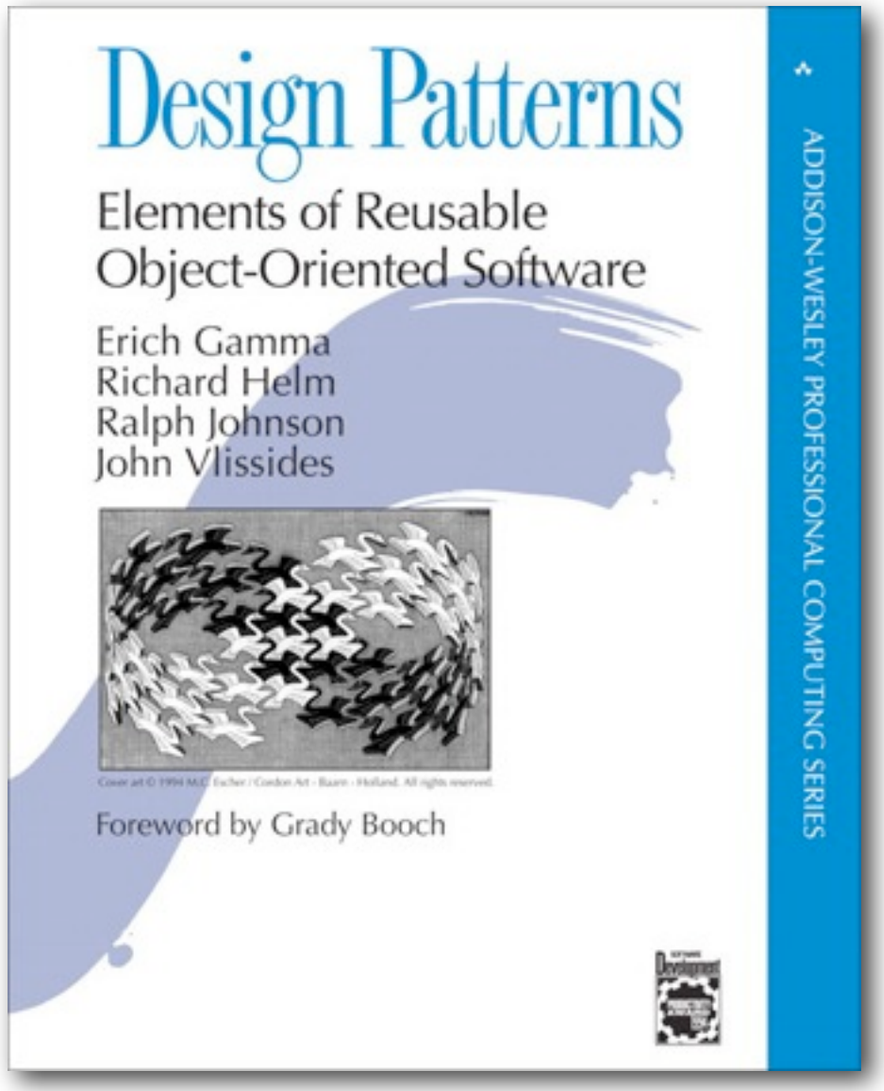
- **next/__next__** devolve o próximo item
- Você normalmente não chama **__next__**
  - mas se precisar, use **next(x)**

Python ≥ 2.6

Iterator é...

- um padrão de projeto



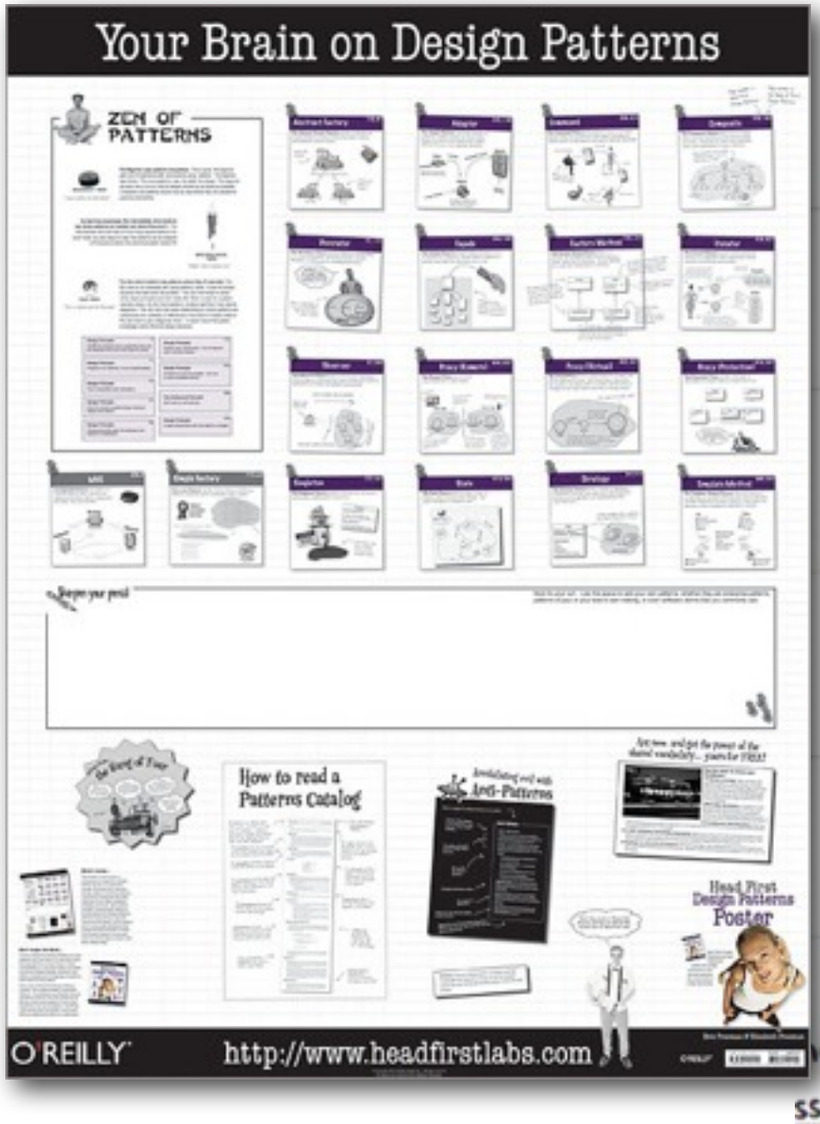


Design Patterns

Gamma, Helm, Johnson & Vlissides
Addison-Wesley,
ISBN 0-201-63361-2

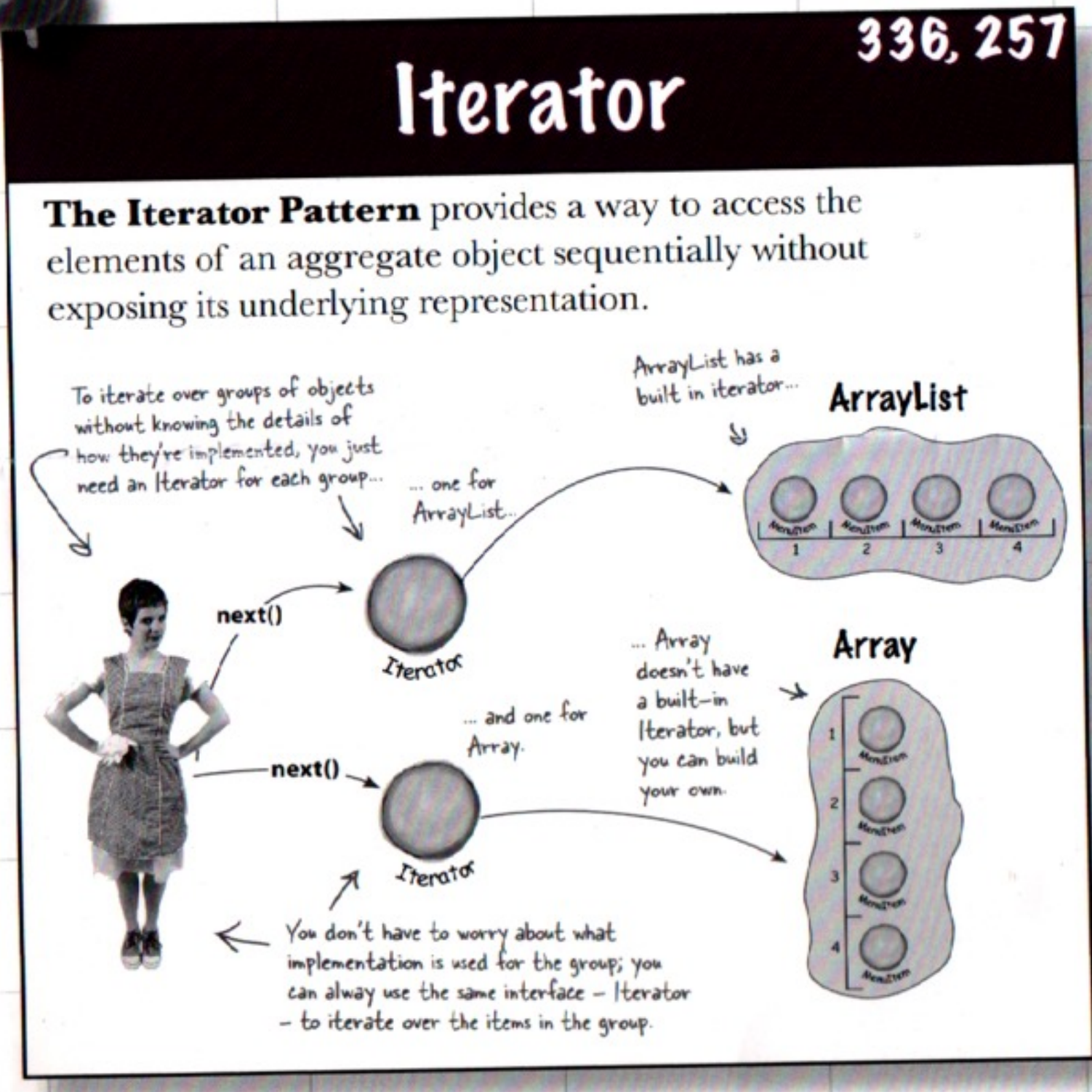
Turing.com.br

@ramalhoorg



Head First Design Patterns Poster

O'Reilly,
ISBN 0-596-10214-3



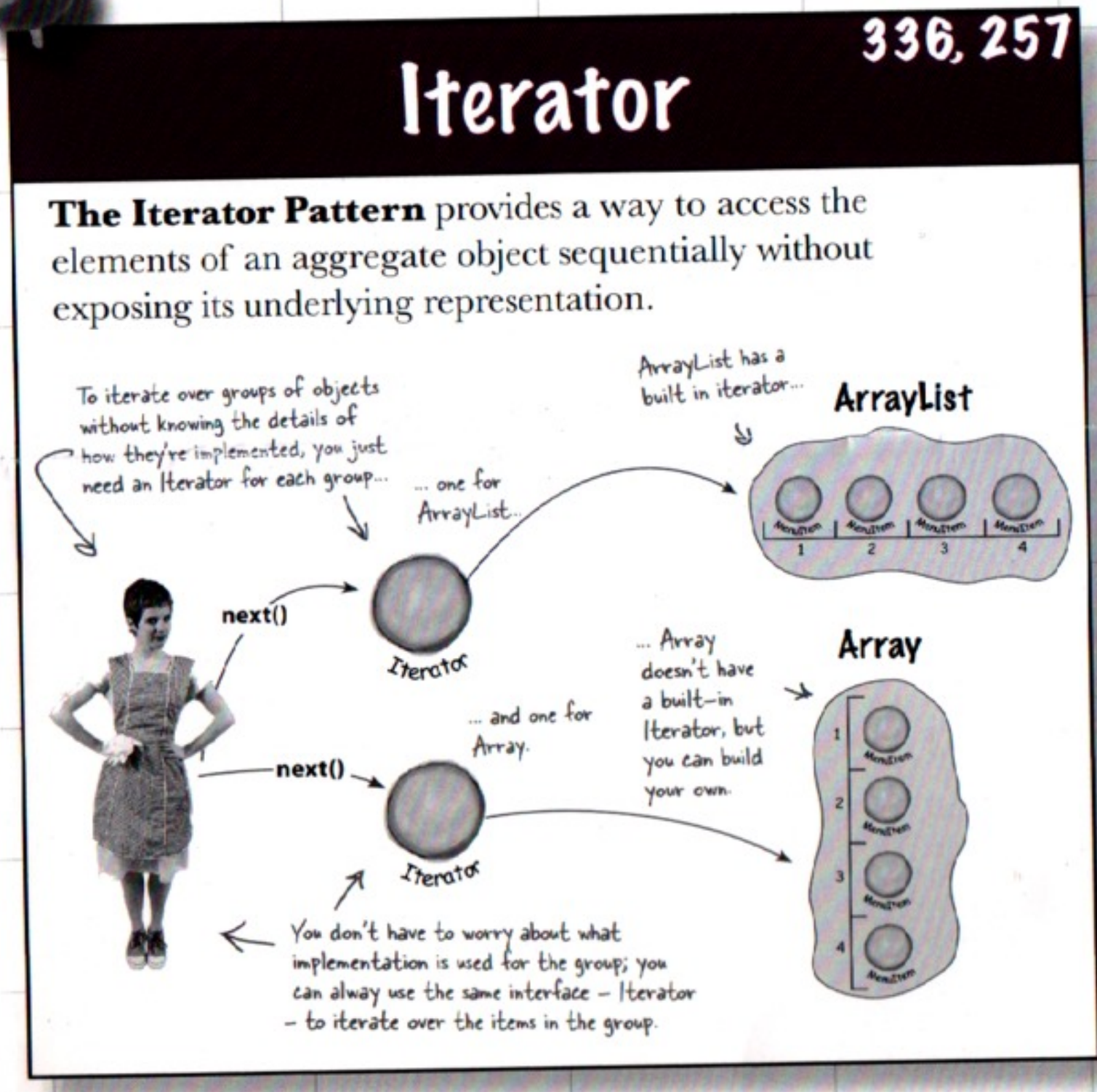
Turing.com.br

O padrão **Iterator** permite acessar os itens de uma coleção sequencialmente, isolando o cliente da implementação da coleção.

Head First Design Patterns Poster

O'Reilly,
ISBN 0-596-10214-3

Turing.com.br



Trem com iterator

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        return IteradorTrem(self.vagoes)

class IteradorTrem(object):

    def __init__(self, vagoes):
        self.atual = 0
        self.ultimo_vagao = vagoes - 1

    def next(self):
        if self.atual <= self.ultimo_vagao:
            self.atual += 1
            return 'vagao #%s' % (self.atual)
        else:
            raise StopIteration()
```

```
>>> t = Trem(4)
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

@ramalhoorg

Trem com iterator

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        return IteradorTrem(self.vagoes)

class IteradorTrem(object):

    def __init__(self, vagoes):
        self.atual = 0
        self.ultimo_vagao = vagoes - 1

    def next(self):
        if self.atual <= self.ultimo_vagao:
            self.atual += 1
            return 'vagao #%s' % (self.atual)
        else:
            raise StopIteration()
```

iter(t)

```
>>> t = Trem(4)
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

- **for vagao in t:**
  - invoca **iter(t)**
    - devolve **IteradorTrem**

Trem com iterator

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        return IteradorTrem(self.vagoes)

class IteradorTrem(object):

    def __init__(self, vagoes):
        self.atual = 0
        self.ultimo_vagao = vagoes - 1

    def next(self):
        if self.atual <= self.ultimo_vagao:
            self.atual += 1
            return 'vagao #%s' % (self.atual)
        else:
            raise StopIteration()
```

next(it_trem)

```
>>> t = Trem(4)
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

- **for vagao in t:**
  - invoca **iter(t)**
    - devolve **IteradorTrem**
  - invoca **next(it_trem)** até que ele levante **StopIteration**

Em Python, um iterável é...

- Um objeto a partir do qual a função **iter** consegue obter um iterador.
- A chamada **iter(x)**:
  - invoca **x.__iter__()** para obter um iterador
  - **ou**, se **x.__iter__** não existe:
    - fabrica um iterador que acessa os itens de x sequenciamente: **x[0], x[1], x[2]** etc.

interface Iterable

protocolo de sequência

Iteração em C (exemplo 2)

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    int i;
    for(i = 0; i < argc; i++)
        printf("%d : %s\n", i, argv[i]);
    return 0;
}
```

```
$ ./args2 alfa bravo charlie
0 : ./args2
1 : alfa
2 : bravo
3 : charlie
```

Iteração em Python (ex. 2)

não Pythonico!

```
import sys

for i in range(len(sys.argv)):
    print i, ':', sys.argv[i]
```

```
$ python args2.py alfa bravo charlie
0 : args2.py
1 : alfa
2 : bravo
3 : charlie
```

Iteração em Python (ex. 2)

Pythonico!

```
import sys

for i, arg in enumerate(sys.argv):
    print i, ':', arg
```

```
$ python args2.py alfa bravo charlie
0 : args2.py
1 : alfa
2 : bravo
3 : charlie
```

Turing.com.br

Iteração em Python (ex. 2)

```
import sys

for i, arg in enumerate(sys.argv):
    print i, ':', arg
```

isso constroi um gerador

o gerador produz uma tupla (indice, item) sob demanda a cada iteração

o gerador é um iterável preguiçoso!

```
$ python args2.py alfa bravo charlie
0 : args2.py
1 : alfa
2 : bravo
3 : charlie
```

Como funciona enumerate

enumerate constroi um gerador

o gerador produz uma tupla (indice, item) a cada next(e)

```
>>> e = enumerate('Turing')
>>> e
<enumerate object at 0x...>
>>> next(e)
(0, 'T')
>>> next(e)
(1, 'u')
>>> next(e)
(2, 'r')
>>> next(e)
(3, 'i')
>>> next(e)
(4, 'n')
>>> next(e)
(5, 'g')
>>> next(e)
Traceback (most recent...):
  ...
StopIteration
```

Turing.com.br

Iterator x generator

- Gerador é uma generalização do iterador
- Por definição, um objeto iterador produz itens iterando sobre outro objeto (alguma coleção)
- Um gerador é um iterável que produz itens sem necessariamente acessar uma coleção
  - ele pode iterar sobre outro objeto mas também pode gerar itens por contra própria, sem qualquer dependência externa (ex. Fibonacci)

Função geradora

- Quaquer função que tenha a palavra reservada **yield** em seu corpo é uma **função geradora**

```
>>> def gen_123():
...     yield 1
...     yield 2
...     yield 3
...
>>> for i in gen_123(): print(i)
1
2
3
>>> g = gen_123()
>>> g
<generator object gen_123 at ...>
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Turing.com.br

Objeto gerador

- Quando invocada, a **função geradora** devolve um **objeto gerador**

```
>>> def gen_123():
...     yield 1
...     yield 2
...     yield 3
...
>>> for i in gen_123(): print(i)
1
2
3
>>> g = gen_123()
>>> g
<generator object gen_123 at ...>
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Objeto gerador

- O objeto gerador é um iterável, implementa **.next()** ou **.__next__()**

Python 2

Python 3

- Use **next(gerador)**

Python ≥ 2.6

```
>>> def gen_123():
...     yield 1
...     yield 2
...     yield 3
...
>>> for i in gen_123(): print(i)
1
2
3
>>> g = gen_123()
>>> g
<generator object gen_123 at ...>
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Turing.com.br

```
>>> def gen_ab():
...     print('iniciando...')
...     yield 'A'
...     print('agora vem B:')
...     yield 'B'
...     print('FIM.')
...
>>> for s in gen_ab(): print(s)
iniciando...
A
agora vem B:
B
FIM.
>>> g = gen_ab()
>>> g  # doctest: +ELLIPSIS
<generator object gen_ab at 0x...>
>>> next(g)
iniciando...
'A'
>>> next(g)
agora vem B:
'B'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Como funciona

- Invocar uma **função geradora** produz um **objeto gerador**
- O corpo da função só começa a ser executado quando se invoca **next**

@ramalhoorg

```
>>> def gen_ab():
...     print('iniciando...')
...     yield 'A'
...     print('agora vem B:')
...     yield 'B'
...     print('FIM.')
...
>>> for s in gen_ab(): print(s)
iniciando...
A
agora vem B:
B
FIM.
>>> g = gen_ab()
>>> g  # doctest: +ELLIPSIS
<generator object gen_ab at 0x...>
>>> next(g)
iniciando...
'A'
>>> next(g)
agora vem B:
'B'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Como funciona

- Quando **next(g)** é invocado, o corpo da função é executado só até o primeiro **yield**

@ramalhoorg

```
>>> def gen_ab():
...     print('iniciando...')
...     yield 'A'
...     print('agora vem B:')
...     yield 'B'
...     print('FIM.')
...
>>> for s in gen_ab(): print(s)
iniciando...
A
agora vem B:
B
FIM.
>>> g = gen_ab()
>>> g  # doctest: +ELLIPSIS
<generator object gen_ab at 0x...>
>>> next(g)
iniciando...
'A'
>>> next(g)
agora vem B:
'B'
>>> next(g)
Traceback (most recent call last):
...
StopIteration
```

Como funciona

- Invocando **next(g)** novamente, a execução avança até o próximo **yield**

@ramalhoorg

Trem c/ função geradora

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        for i in range(self.vagoes):
            yield 'vagao #%s' % (i+1)
```

iter(t)

```
>>> t = Trem(4)
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

- **for vagao in t:**
  - invoca **iter(t)**
    - devolve **gerador**
  - invoca **next(gerador)** até que ele levante **StopIteration**

Iterador clássico x gerador

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        return IteradorTrem(self.vagoes)

class IteradorTrem(object):

    def __init__(self, vagoes):
        self.atual = 0
        self.ultimo_vagao = vagoes - 1

    def next(self):
        if self.atual <= self.ultimo_vagao:
            self.atual += 1
            return 'vagao #%s' % (self.atual)
        else:
            raise StopIteration()
```

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        for i in range(self.vagoes):
            yield 'vagao #%s' % (i+1)
```

1 classe,
3 linhas de código

2 classes,
12 linhas de código

Iterador clássico x gerador

```python
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        return IteradorTrem(self.vagoes)

class IteradorTrem(object):

    def __init__(self, vagoes):
        self.atual = 0
        self.ultimo_vagao = vagoes - 1

    def next(self):
        if self.atual <= self.ultimo_vagao:
            self.atual += 1
            return 'vagao #%s' % (self.atual)
        else:
            raise StopIteration()
```

```python
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        for i in range(self.vagoes):
            yield 'vagao #%s' % (i+1)
```

O gerador administra o contexto para você

Expressão geradora (genexp)

```
>>> g = (c for c in 'ABC')
>>> for l in g:
...     print l
...
A
B
C
>>> g = (c for c in 'ABC')
>>> g
<generator object <genexpr> at 0x10045a410>
```

Expressão geradora

- Quando avaliada, devolve um **objeto gerador**

```
>>> g = (c for c in 'ABC')
>>> for l in g:
...     print l
...
A
B
C
>>> g = (c for c in 'ABC')
>>> g
<generator object <genexpr> at
0x10045a410>
>>> next(g)
'A'
>>> next(g)
'B'
>>> next(g)
'C'
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Trem c/ expressão geradora

```
class Trem(object):

    def __init__(self, num_vagoes):
        self.num_vagoes = num_vagoes

    def __iter__(self):
        return ('vagao #%s' % (i+1)
                    for i in range(self.num_vagoes))
```

iter(t)

```
>>> t = Trem(4)
>>> for vagao in t:
...     print(vagao)
vagao #1
vagao #2
vagao #3
vagao #4
```

- for vagao in t:
  - invoca iter(t)
    - devolve gerador
  - invoca gerador.next() até que ele levante StopIteration

Função geradora x genexp

```
class Trem(object):

    def __init__(self, vagoes):
        self.vagoes = vagoes

    def __iter__(self):
        for i in range(self.vagoes):
            yield 'vagao #%s' % (i+1)
```

```
class Trem(object):

    def __init__(self, num_vagoes):
        self.num_vagoes = num_vagoes

    def __iter__(self):
        return ('vagao #%s' % (i+1) for i in range(self.num_vagoes))
```

Construtores embutidos que consomem e produzem iteráveis

- dict
- enumerate
- frozenset
- list
- reversed
- set
- tuple

Módulo itertools

- geradores (potencialmente) infinitos
  - count(), cycle(), repeat()
- geradores que combinam vários iteráveis
  - chain(), tee(), izip(), imap(), product(), compress()...
- geradores que selecionam ou agrupam itens:
  - compress(), dropwhile(), groupby(), ifilter(), islice()...
- Iteradores que produzem combinações
  - product(), permutations(), combinations()...

Geradores em Python 3

- Várias funções e métodos da biblioteca padrão que devolviam listas agora devolvem geradores:
  - **`dict.keys(), dict.items(), dict.values()`**...
    - **`range(...)`**
      - como **`xrange`** no Py 2 (mais que um gerador)
- Quando precisar de uma lista, basta passar o gerador para o construtor de **`list`**:

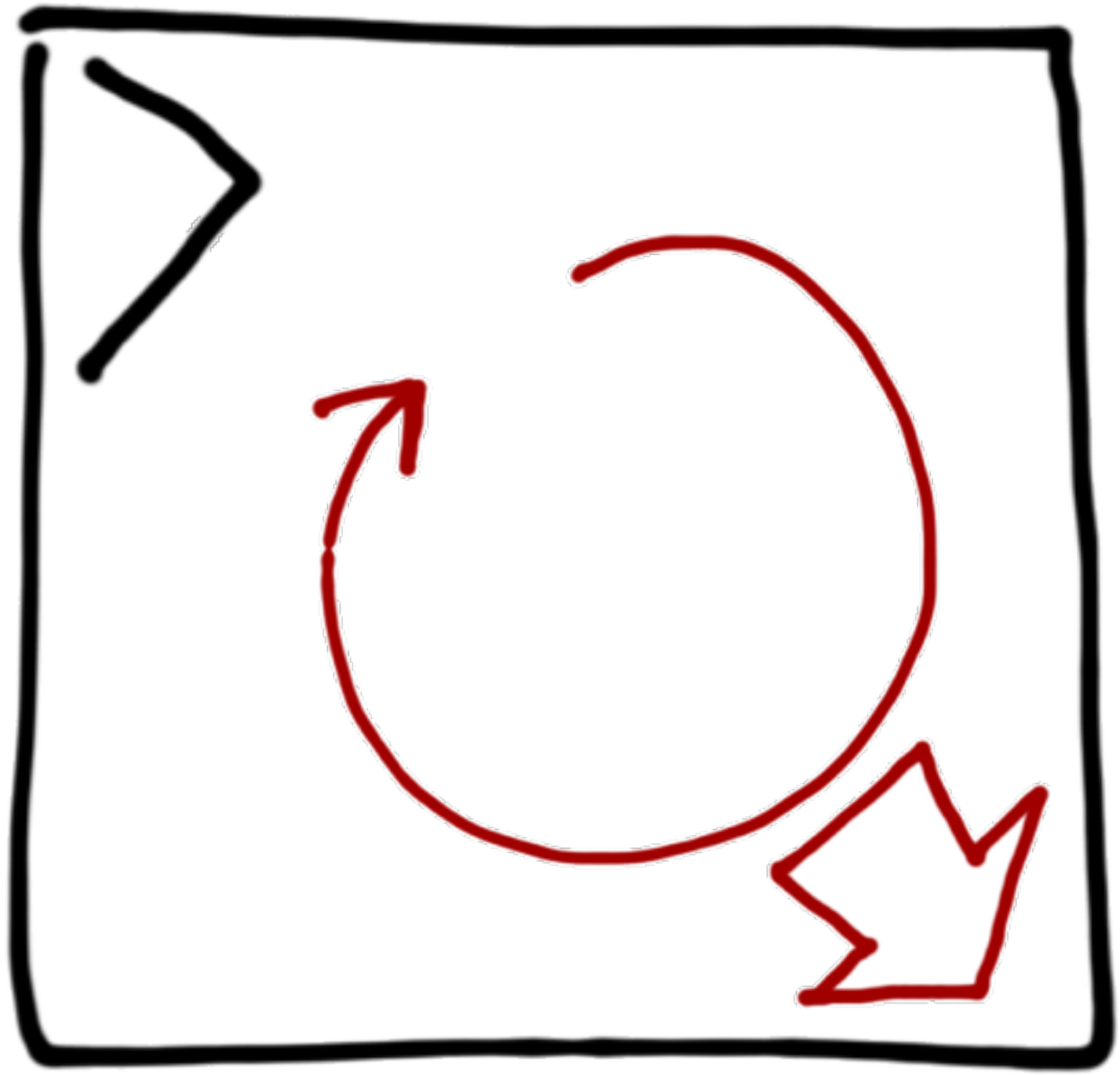
```
list(range(10))
```

Exemplo prático com funções geradoras

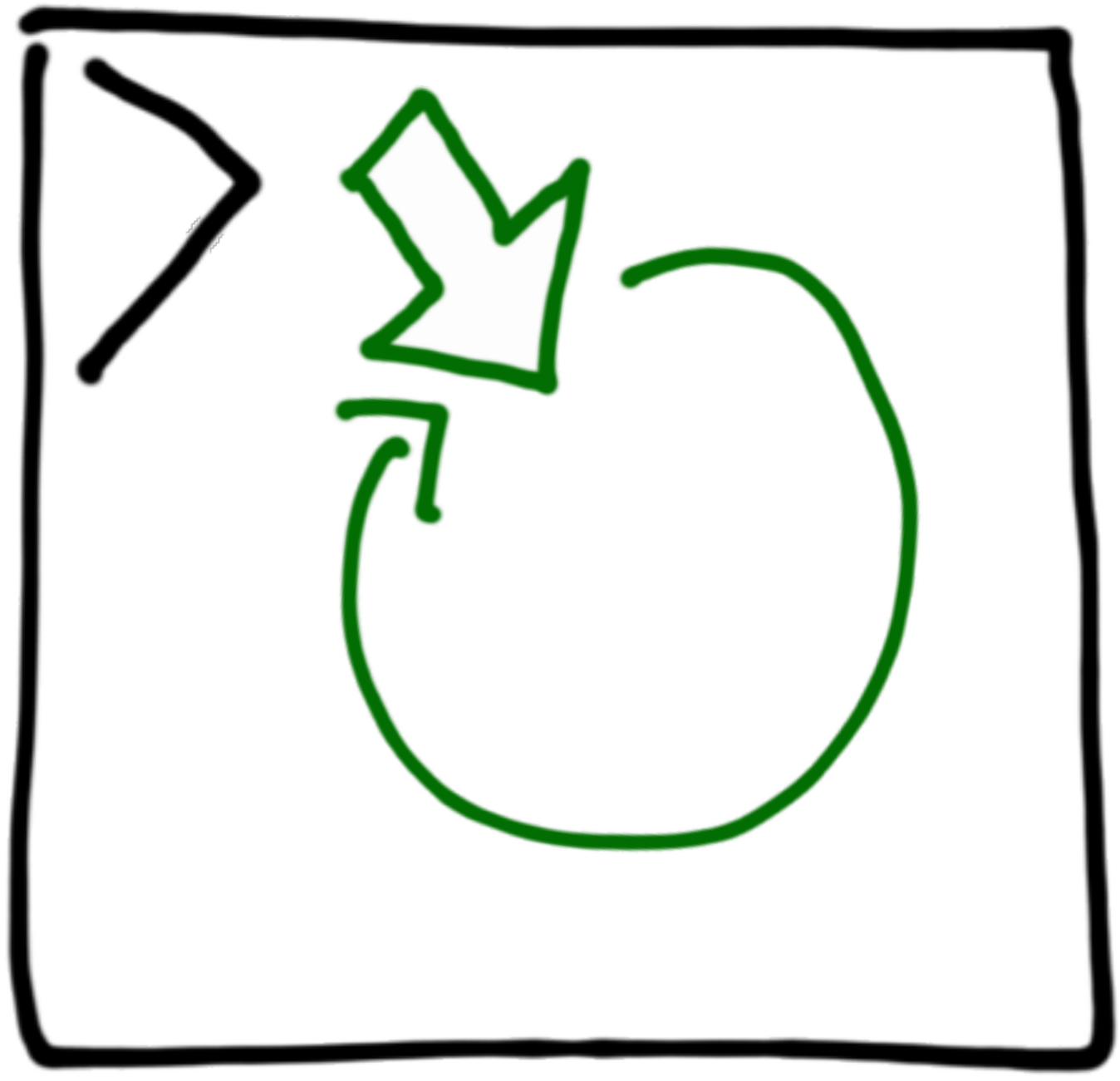
- Funções geradoras para desacoplar laços de leitura e escrita em uma ferramenta para conversão de bases de dados semi-estruturadas

`https://github.com/ramalho/isis2json`

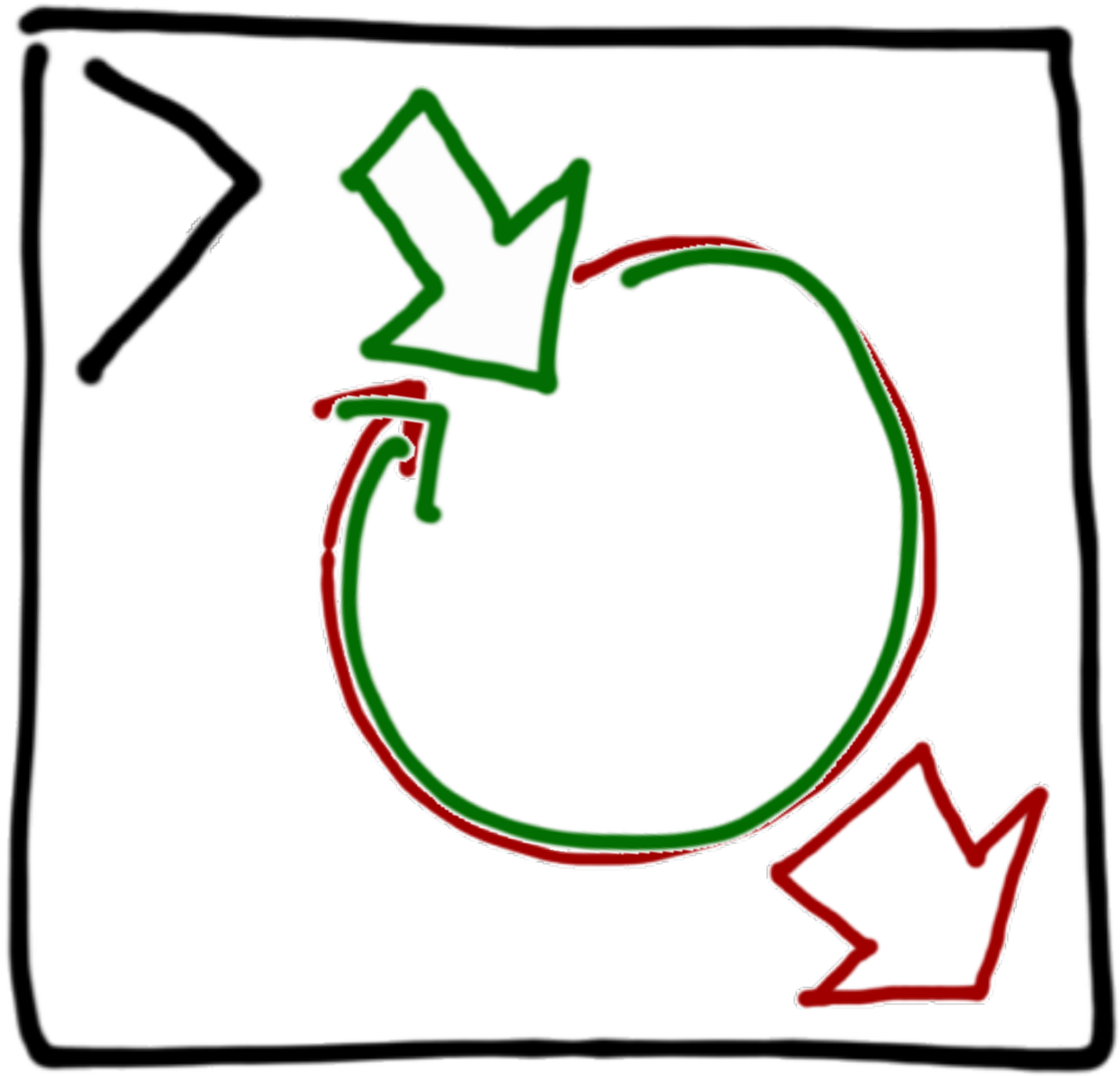
Laço principal escreve arquivo JSON



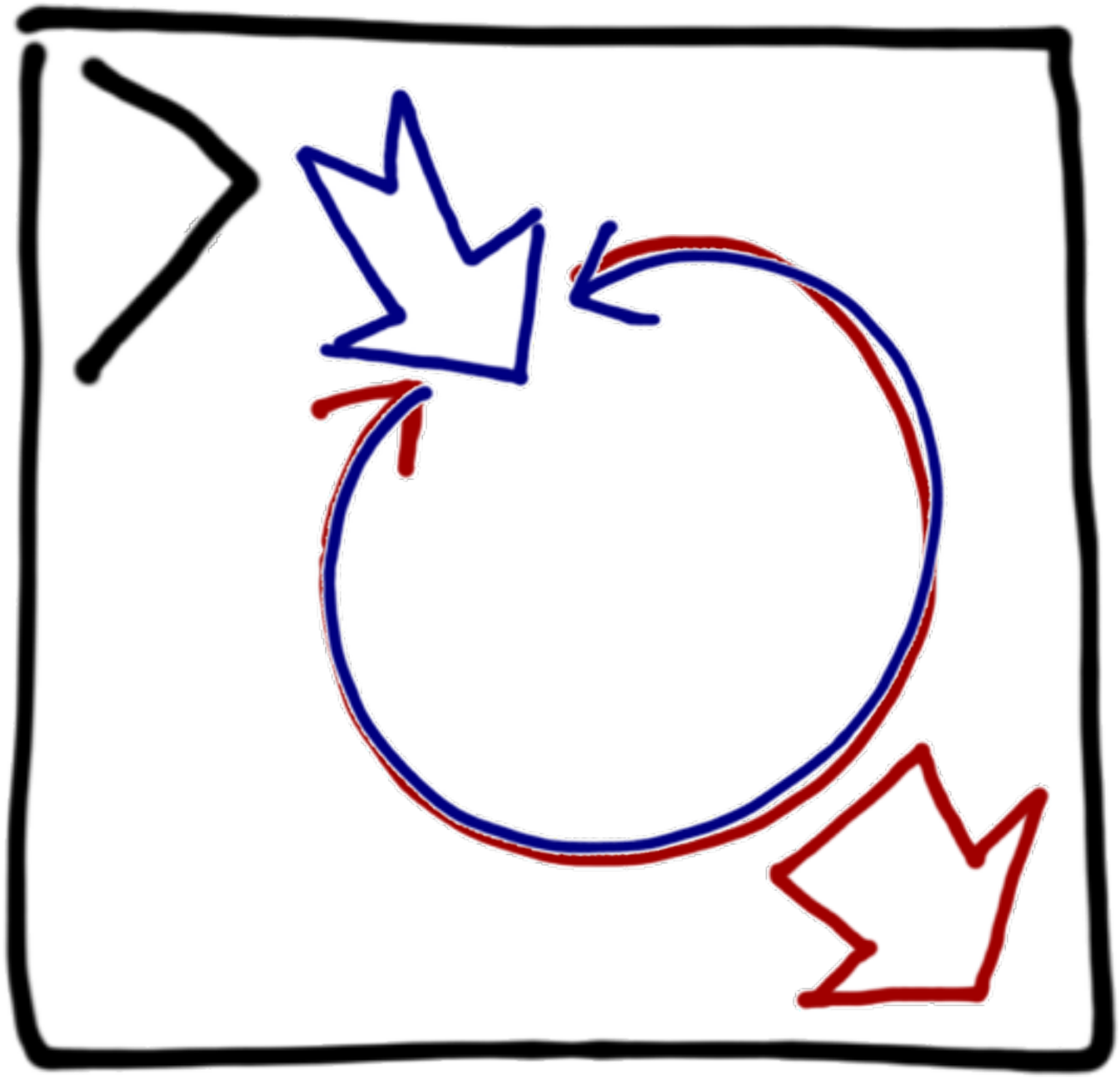
Um outro laço lê os registros a converter



Implementação possível: o mesmo laço lê e grava



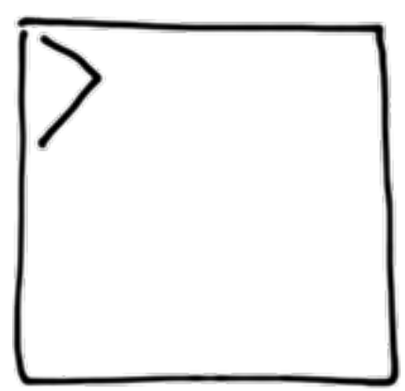
Mas e a lógica para ler outro formato?



Funções do script

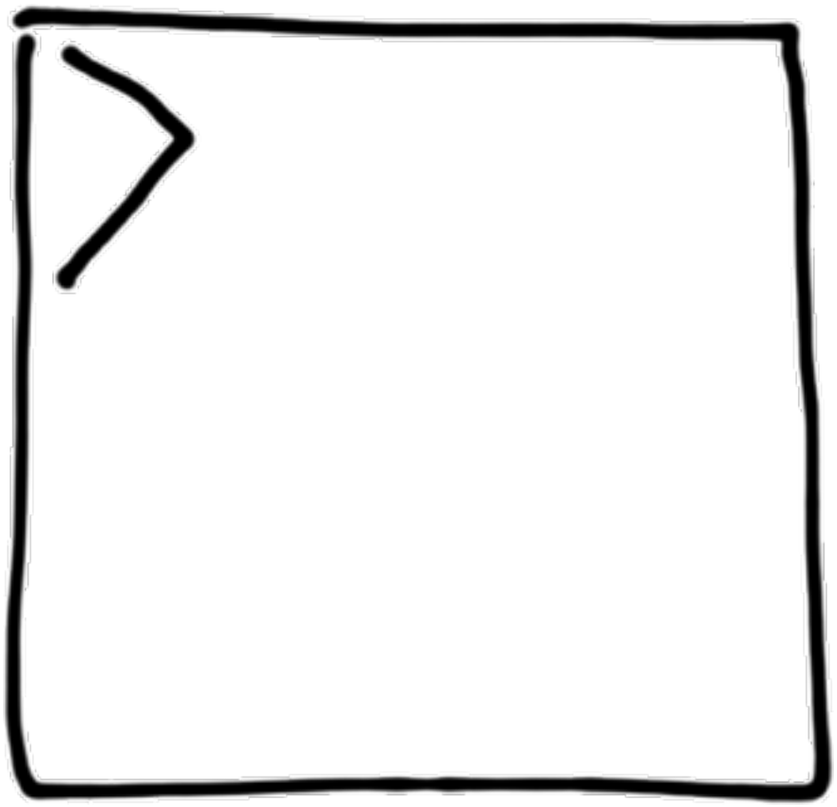
- iterMstRecords*
- iterIsoRecords*
- writeJsonArray
- main

* funções geradoras



Turing.com.br

Função main: leitura dos argumentos



```
def main():
    # create the parser
    parser = argparse.ArgumentParser(
        description='Convert an ISIS .mst or .iso file to a JSON array')

    # add the arguments
    parser.add_argument(
        'file_name', metavar='INPUT.(mst|iso)',
        help='.mst or .iso file to read')
    parser.add_argument(
        '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
        metavar='OUTPUT.json',
        help='the file where the JSON output should be written'
             ' (default: write to stdout)')
    parser.add_argument(
        '-c', '--couch', action='store_true',
        help='output array within a "docs" item in a JSON document'
             ' for bulk insert to CouchDB via POST to db/_bulk_docs')
    parser.add_argument(
        '-m', '--mongo', action='store_true',
        help='output individual records as separate JSON dictionaries,'
             ' one per line for bulk insert to MongoDB via mongoimport utility')
    parser.add_argument(
        '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
        help='ISIS-JSON type, sets field structure: 1=string, 2=alist, 3=dict '
            '(default=1)')
    parser.add_argument(
        '-q', '--qty', type=int, default=DEFAULT_QTY,
        help='maximum quantity of records to read (default=ALL)')
    parser.add_argument(
        '-s', '--skip', type=int, default=0,
        help='records to skip from start of .mst (default=0)')
    parser.add_argument(
        '-i', '--id', type=int, metavar='TAG_NUMBER', default=0,
        help='generate an "_id" from the given unique TAG field number'
             ' for each record')
    parser.add_argument(
        '-u', '--uuid', action='store_true',
        help='generate an "_id" with a random UUID for each record')
    parser.add_argument(
        '-p', '--prefix', type=str, metavar='PREFIX', default='',
        help='concatenate prefix to every numeric field tag '
            '(ex. 99 becomes "v99")')
    parser.add_argument(
        '-n', '--mfn', action='store_true',
        help='generate an "_id" from the MFN of each record'
             ' (available only for .mst input)')
    parser.add_argument(
        '-k', '--constant', type=str, metavar='TAG:VALUE', default='',
        help='Include a constant tag:value in every record (ex. -k type:AS)')

    '''
    # TODO: implement this to export large quantities of records to CouchDB
    parser.add_argument(
        '-r', '--repeat', type=int, default=1,
        help='repeat operation, saving multiple JSON files'
             ' (default=1, use -r 0 to repeat until end of input)')
    '''
    # parse the command line
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mst'):
        iterRecords = iterMstRecords
    else:
        if args.mfn:
            print('UNSUPORTED: -n/--mfn option only available for .mst input.')
            raise SystemExit
        iterRecords = iterIsoRecords
    if args.couch:
        args.out.write('{ "docs" : ')
    writeJsonArray(iterRecords, args.file_name, args.out, args.qty, args.skip,
        args.id, args.uuid, args.mongo, args.mfn, args.type, args.prefix,
        args.constant)
    if args.couch:
        args.out.write('}\n')
    args.out.close()

if __name__ == '__main__':
    main()
```

Função main: seleção do formato de entrada

escolha da função geradora de leitura depende do formato de entrada

```
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mst'):
        iterRecords = iterMstRecords
    else:
        if args.mfn:
            print('UNSUPORTED: -n/--mfn option only available for .mst input.')
            raise SystemExit
        iterRecords = iterIsoRecords
    if args.couch:
        args.out.write('{ "docs" : ')
    writeJsonArray(iterRecords, args.file_name, args.out, args.qty, args.skip,
        args.id, args.uuid, args.mongo, args.mfn, args.type, args.prefix,
        args.constant)
    if args.couch:
        args.out.write('}\n')
    args.out.close()

if __name__ == '__main__':
    main()
```

função geradora escolhida é passada como argumento

writeJsonArray:

escrever registros em JSON

```
def writeJsonArray(iterRecords, file_name, output, qty, skip, id_tag,
                   gen_uuid, mongo, mfn, isis_json_type, prefix, constant):
    start = skip
    end = start + qty
    if not mongo:
        output.write('[')
    if id_tag:
        id_tag = str(id_tag)
        ids = set()
    else:
        id_tag = ''
    for i, record in enumerate(iterRecords(file_name, isis_json_type)):
        if i >= end:
            break
        if i > start and not mongo:
            output.write(',')
        output.write('\n')
        if start <= i < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
                if occurrences is None:
                    msg = 'id tag #%s not found in record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise KeyError(msg % (id_tag, i))
                if len(occurrences) > 1:
                    msg = 'multiple id tags #%s found in record %s'
                    if ISIS_MFN_KEY in record:
                        msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                    raise TypeError(msg % (id_tag, i))
                else: # ok, we have one and only one id field
                    if isis_json_type == 1:
                        id = occurrences[0]
                    elif isis_json_type == 2:
                        id = occurrences[0][0][1]
                    elif isis_json_type == 3:
                        id = occurrences[0]['_']
                    if id in ids:
                        msg = 'duplicate id %s in tag #%s, record %s'
                        if ISIS_MFN_KEY in record:
                            msg = msg + (' (mfn=%s)' % record[ISIS_MFN_KEY])
                        raise TypeError(msg % (id, id_tag, i))
                    record['_id'] = id
                    ids.add(id)
            elif gen_uuid:
                record['_id'] = unicode(uuid4())
            elif mfn:
                record['_id'] = record[ISIS_MFN_KEY]
            if prefix:
                # iterate over a fixed sequence of tags
                for tag in tuple(record):
                    if str(tag).isdigit():
                        record[prefix+tag] = record[tag]
                        del record[tag] # this is why we iterate over a tuple
                        # with the tags, and not directly on the record dict
            if constant:
                constant_key, constant_value = constant.split(':')
                record[constant_key] = constant_value
            output.write(json.dumps(record).encode('utf-8'))
    if not mongo:
        output.write('\n]')
    output.write('\n')
```

Turing.com.br

@ramalhoorg

writeJsonArray:

itera sobre umas das funções geradoras

```
def writeJsonArray(iterRecords, file_name, output, qty, skip, id_tag,
                   gen_uuid, mongo, mfn, isis_json_type, prefix, constant):
    start = skip
    end = start + qty
    if not mongo:
        output.write('[')
    if id_tag:
        id_tag = str(id_tag)
        ids = set()
    else:
        id_tag = ''
    for i, record in enumerate(iterRecords(file_name, isis_json_type)):
        if i >= end:
            break
        if i > start and not mongo:
            output.write(',')
        output.write('\n')
        if start <= i < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
```

iterIsoRecords:

ler registros de arquivo ISO-2709

função geradora!

```
def iterIsoRecords(iso_file_name, isis_json_type):
    from iso2709 import IsoFile
    from subfield import expand

    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = {}
        for field in record.directory:
            field_key = str(int(field.tag)) # remove leading zeroes
            field_occurrences = fields.setdefault(field_key,[])
            content = field.value.decode(INPUT_ENCODING,'replace')
            if isis_json_type == 1:
                field_occurrences.append(content)
            elif isis_json_type == 2:
                field_occurrences.append(expand(content))
            elif isis_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError(
                    'ISIS-JSON type %s conversion not yet '
                    'implemented for .iso input' % isis_json_type)

        yield fields
    iso.close()
```

Turing.com.br

@ramalhoorg

iterIsoRecords

```
def iterIsoRecords(iso_file_name, isis_json_type):
    from iso2709 import IsoFile
    from subfield import expand

    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = {}
        for field in record.directory:
            field_key = str(int(field.tag)) # remove leading zeroes
            field_occurrences = fields.setdefault(field_key,[])
            content = field.value.decode(INPUT_ENCODING,'replace')
            if isis_json_type == 1:
                field_occurrences.append(content)
            elif isis_json_type == 2:
                field_occurrences.append(expand(content))
            elif isis_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError(
                    'ISIS-JSON type %s conversion not yet '
                    'implemented for .iso input' % isis_json_type)

        yield fields
    iso.close()
```

cria um novo dict a cada iteração

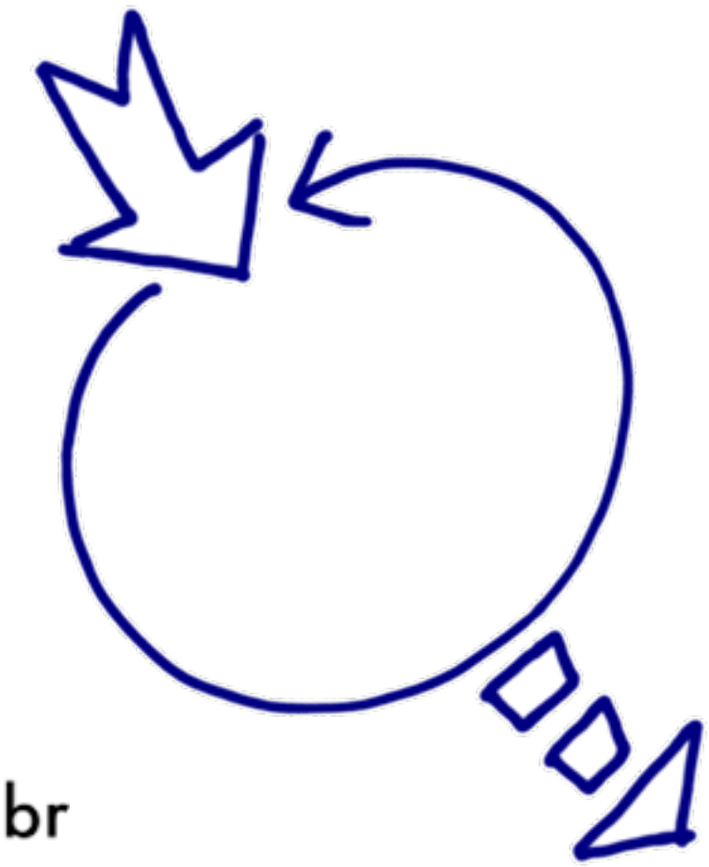
produz (yield) registro na forma de um dict

iterMstRecords:

função geradora!

ler registros de arquivo ISIS .MST

```
def iterMstRecords(master_file_name, isis_json_type):
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('IMPORT ERROR: Jython 2.5 and Bruma.jar are required '
              'to read .mst files')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()
    for record in mst:
        fields = {}
        if SKIP_INACTIVE:
            if record.getStatus() != Record.Status.ACTIVE:
                continue
        else: # save status only there are non-active records
            fields[ISIS_ACTIVE_KEY] = record.getStatus() == Record.Status.ACTIVE
        fields[ISIS_MFN_KEY] = record.getMfn()
        for field in record.getFields():
            field_key = str(field.getId())
            field_occurrences = fields.setdefault(field_key,[])
            if isis_json_type == 3:
                content = {}
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content['_'] = subfield.getContent()
                    else:
                        subfield_occurrences = content.setdefault(subfield_key,[])
                        subfield_occurrences.append(subfield.getContent())
                field_occurrences.append(content)
            elif isis_json_type == 1:
                content = []
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content.insert(0, subfield.getContent())
                    else:
                        content.append(SUBFIELD_DELIMITER+subfield_key+
                                       subfield.getContent())
                field_occurrences.append(''.join(content))
            else:
                raise NotImplementedError(
                    'ISIS-JSON type %s conversion not yet '
                    'implemented for .mst input' % isis_json_type)
        yield fields
    mst.close()
```



Turing.com.br

@ramalhoorg

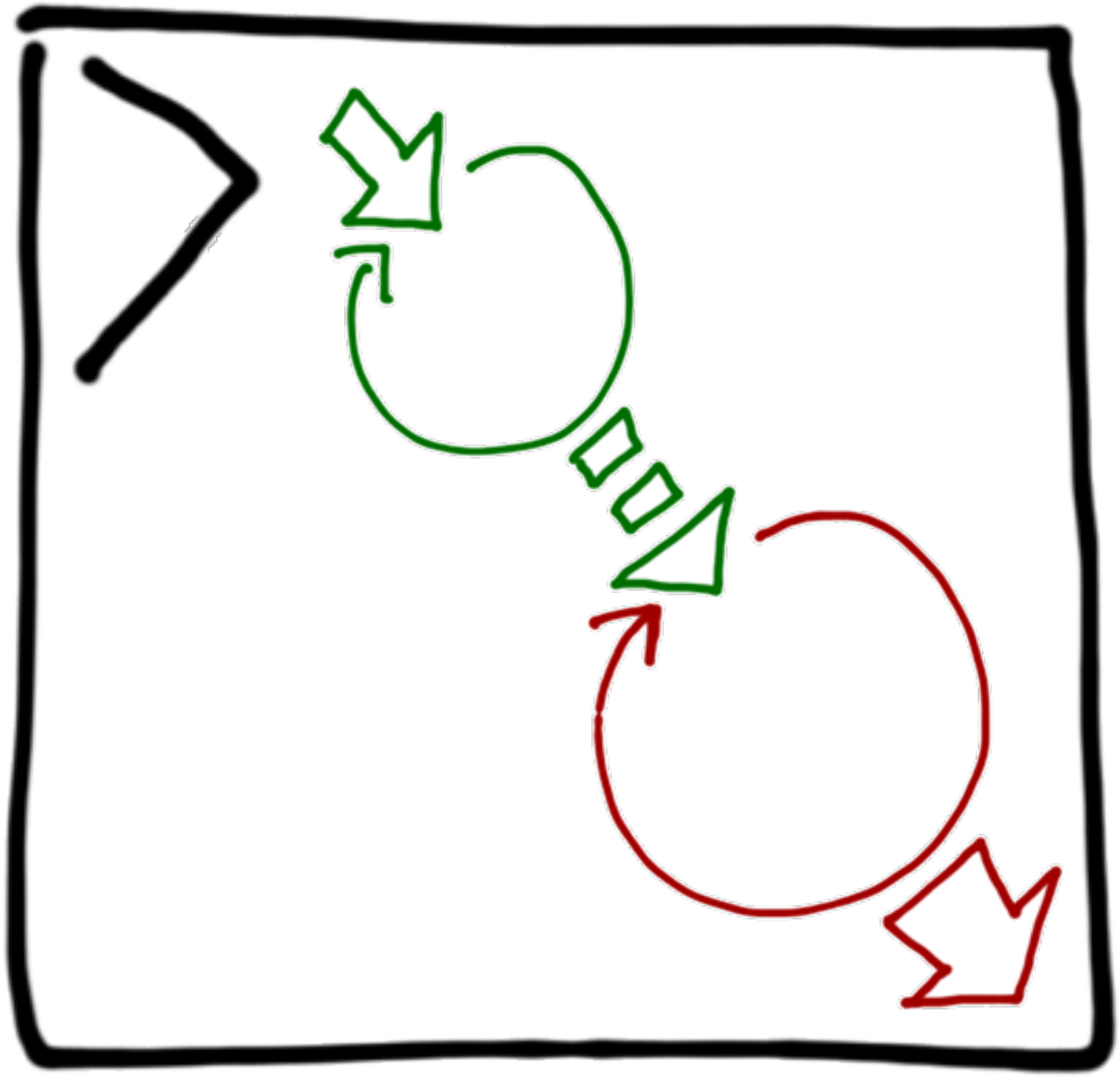
iterMstRecords

```
def iterMstRecords(master_file_name, isis_json_type):
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('IMPORT ERROR: Python 2.7 + Bruma.jar are required '
              'to read ISIS files')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()
    for record in mst:
        fields = {}
        if SKIP_INACTIVE:
            if record.getStatus() != Record.Status.ACTIVE:
                continue
        else: # save status only there are non-active records
            fields[ISIS_ACTIVE_KEY] = record.getStatus() == Record.Status.ACTIVE
        fields[ISIS_MFN_KEY] = record.getMfn()
        for field in record.getFields():
            field_key = str(field.getId())
            field_occurrences = fields.setdefault(field_key,[])
            if isis_json_type == 3:
                content = {}
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content['_'] = subfield.getContent()
                    else:
                        subfield_occurrences = content.setdefault(subfield_key,[])
                        subfield_occurrences.append(subfield.getContent())
                field_occurrences.append(content)
            elif isis_json_type == 1:
                content = []
                for subfield in field.getSubfields():
                    subfield_key = subfield.getId()
                    if subfield_key == '*':
                        content.insert(0, subfield.getContent())
                    else:
                        content.append(SUBFIELD_DELIMITER+subfield_key+
                                       subfield.getContent())
                field_occurrences.append(''.join(content))
            else:
                raise NotImplementedError(
                    'ISIS-JSON type %s conversion not yet '
                    'implemented for .mst input' % isis_json_
        yield fields
    mst.close()
```

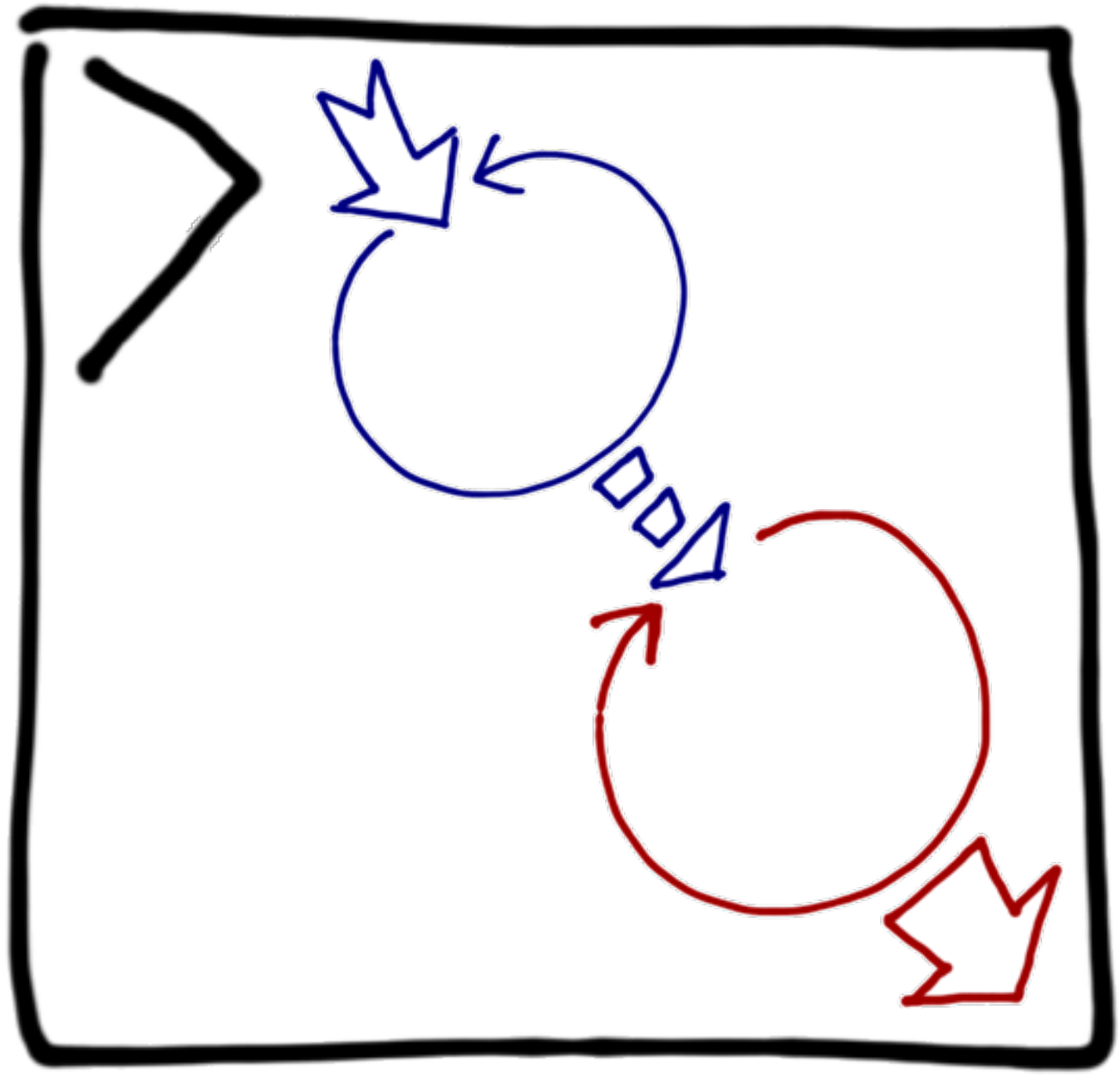
cria um novo dict a cada iteração

produz (yield) registro na forma de um dict

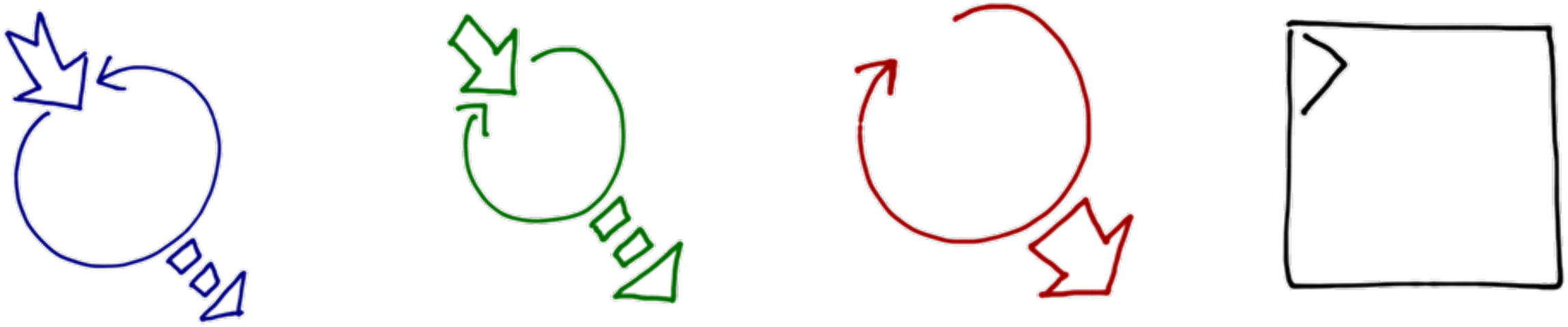
Geradores na prática



Geradores na prática



Geradores na prática



Faltou apresentar...

- Envio de dados para um gerador através do método **`.send()`** (em vez de **`.next()`**), e uso de **`yield`** como uma expressão para obter o dado enviado

.send() não costuma ser usado no contexto de iteração mas em pipelines

- Uso de funções geradoras como co-rotinas

"Coroutines are not related to iteration"
David Beazley

Turing.com.br

Faltou apresentar...

- Envio de dados para um gerador através do método **`.send()`** (em vez de **`.next()`**), e uso de **`yield`** como uma expressão para obter o dado enviado

.send() não costuma ser usado no contexto de iteração mas em pipelines

- Uso de funções geradoras como co-rotinas

"Co-rotinas não têm relação com iteração"
David Beazley

Oficinas Turing:
computação para programadores

- Próximos lançamentos:
  - 1ª turma de **Python para quem usa Django**
  - 3ª turma de **Objetos Pythonicos**
  - 4ª turma de **Python para quem sabe Python**

Para saber mais sobre estes cursos **ONLINE**
escreva para:
ramalho@turing.com.br

Turing.com.br