

ThoughtWorks®

a preguiça como qualidade

O PODER DOS GERADORES

Como funcionam geradores em Python, e como usá-los para processar grandes volumes de dados com eficiência



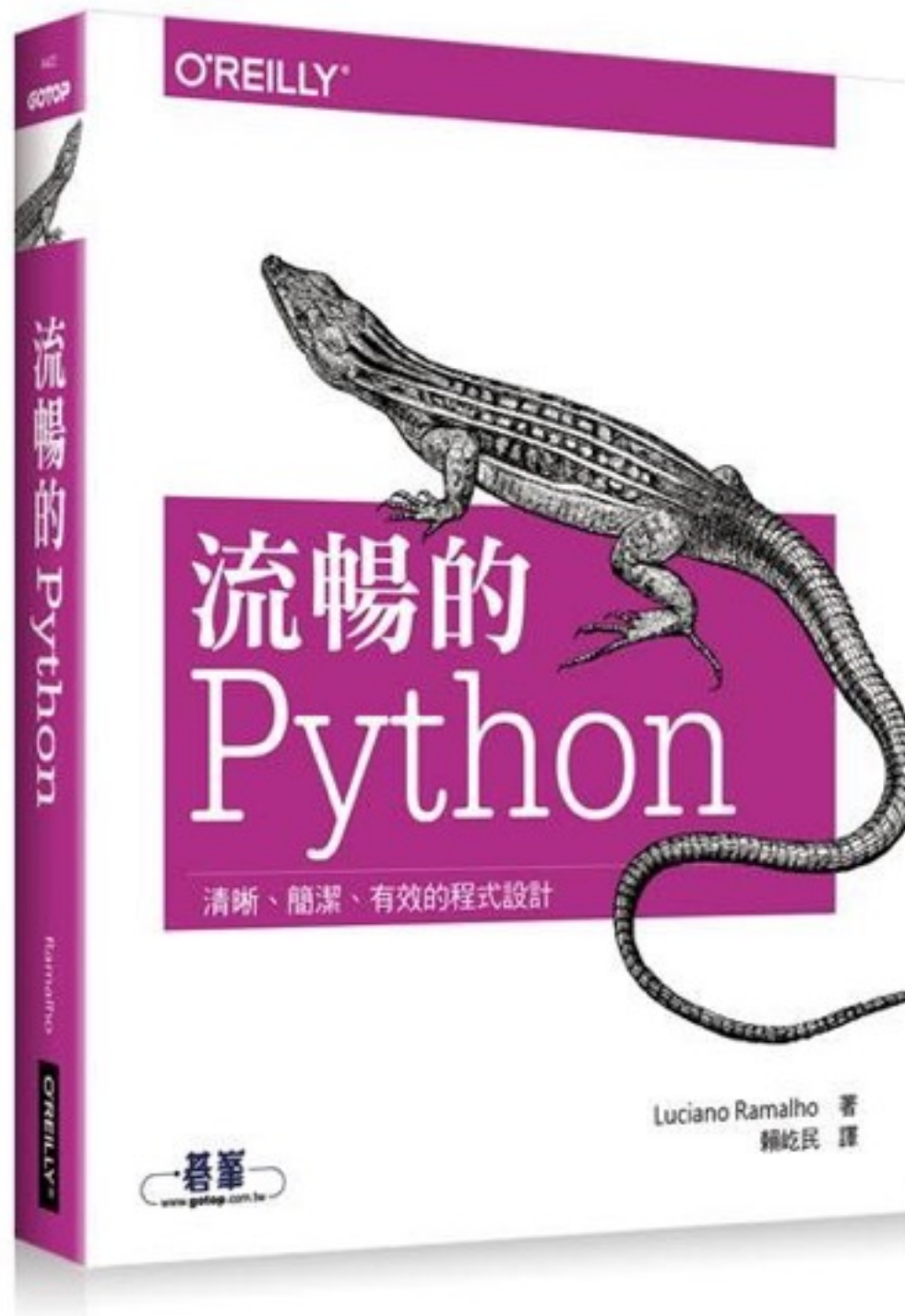
ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com

FLUENT PYTHON, MEU PRIMEIRO LIVRO



Fluent Python (O'Reilly, 2015)

Python Fluente (Novatec, 2015)

**Python к вершинам
мастерства*** (DMK, 2015)

流暢的 **Python**[†] (Gotop, 2016)

also in **Polish, Korean...**

* *Python. To the heights of excellence*

† *Smooth Python*

GERADORES BUILT-IN

Em Python 3, eles estão em toda parte

ZIP, MAP E FILTER EM PYTHON 2

Python 2

```
>>> L = [0, 1, 2]
>>> zip('ABC', L)
[('A', 0), ('B', 1), ('C', 2)]

>>> map(lambda x: x*10, L)
[0, 10, 20]

>>> filter(None, L)
[1, 2]
```

zip:

percorre N iteráveis em paralelo, devolve sequência de N-uplas

map:

aplica função a cada item, devolve sequência de resultados

filter:

aplica predicado a cada item, devolve sequência de itens “verdadeiros”

ZIP, MAP E FILTER: PYTHON 2 × PYTHON 3

Python 2

```
>>> L = [0, 1, 2]
>>> zip('ABC', L)
[('A', 0), ('B', 1), ('C', 2)]

>>> map(lambda x: x*10, L)
[0, 10, 20]

>>> filter(None, L)
[1, 2]
```

Python 3

```
>>> L = [0, 1, 2]
>>> zip('ABC', L)
<zip object at 0x102218408>

>>> map(lambda x: x*10, L)
<map object at 0x102215a90>

>>> filter(None, L)
<filter object at 0x102215b00>
```

Cadê as listas?

O que são estes resultados?

ZIP, MAP E FILTER DEVOLVEM GERADORES

Um gerador é um objeto **iterável**:

```
>>> L = [0, 1, 2]
>>> for par in zip('ABC', L):
...     print(par)
...
('A', 0)
('B', 1)
('C', 2)
```

Para criar a lista, basta passar o gerador para o construtor:

```
>>> list(zip('ABC', L))
[('A', 0), ('B', 1), ('C', 2)]
```

Vários construtores de coleções aceitam iteráveis:

```
>>> dict(zip('ABC', L))
{'C': 2, 'B': 1, 'A': 0}
```

ZIP, MAP E FILTER DEVOLVEM GERADORES

Um gerador implementa a interface **Iterator**:

```
>>> L = [0, 1, 2]
>>> z = zip('ABC', L)
>>> next(z)
('A', 0)
>>> next(z)
('B', 1)
>>> next(z)
('C', 2)
```

Uma vez esgotado, um gerador levanta **StopIteration** e não serve mais para nada, pois não pode ser reiniciado:

```
>>> next(z)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```


ThoughtWorks®

ITERAÇÃO

Não é o mesmo que *interação*!

ITERAÇÃO: LINGUAGEM 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;
}
```

```
$ ./args alfa bravo charlie
./args
alfa
bravo
charlie
```

ITERAÇÃO: LINGUAGENS C E PYTHON

```
#include <stdio.h>

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

```
import sys
```

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

ITERAÇÃO: ANTES DE JAVA 5

```
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: A PARTIR DE JAVA 5

Foreach: oficialmente, *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: JAVA E PYTHON

Foreach: oficialmente, *enhanced for* (“for melhorado”)

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

ano:
2004

```
import sys
```

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

ano:
1991

FOR/FOREACH FUNCIONA PORQUE

- Python (e Java) possuem objetos iteráveis
 - **iterável** = “que pode ser iterado”
- A partir de um objeto **iterável**, é possível se obter um **iterador**
- O **iterador** suporta a função **next** que fornece valores sucessivos para a variável do laço **for**

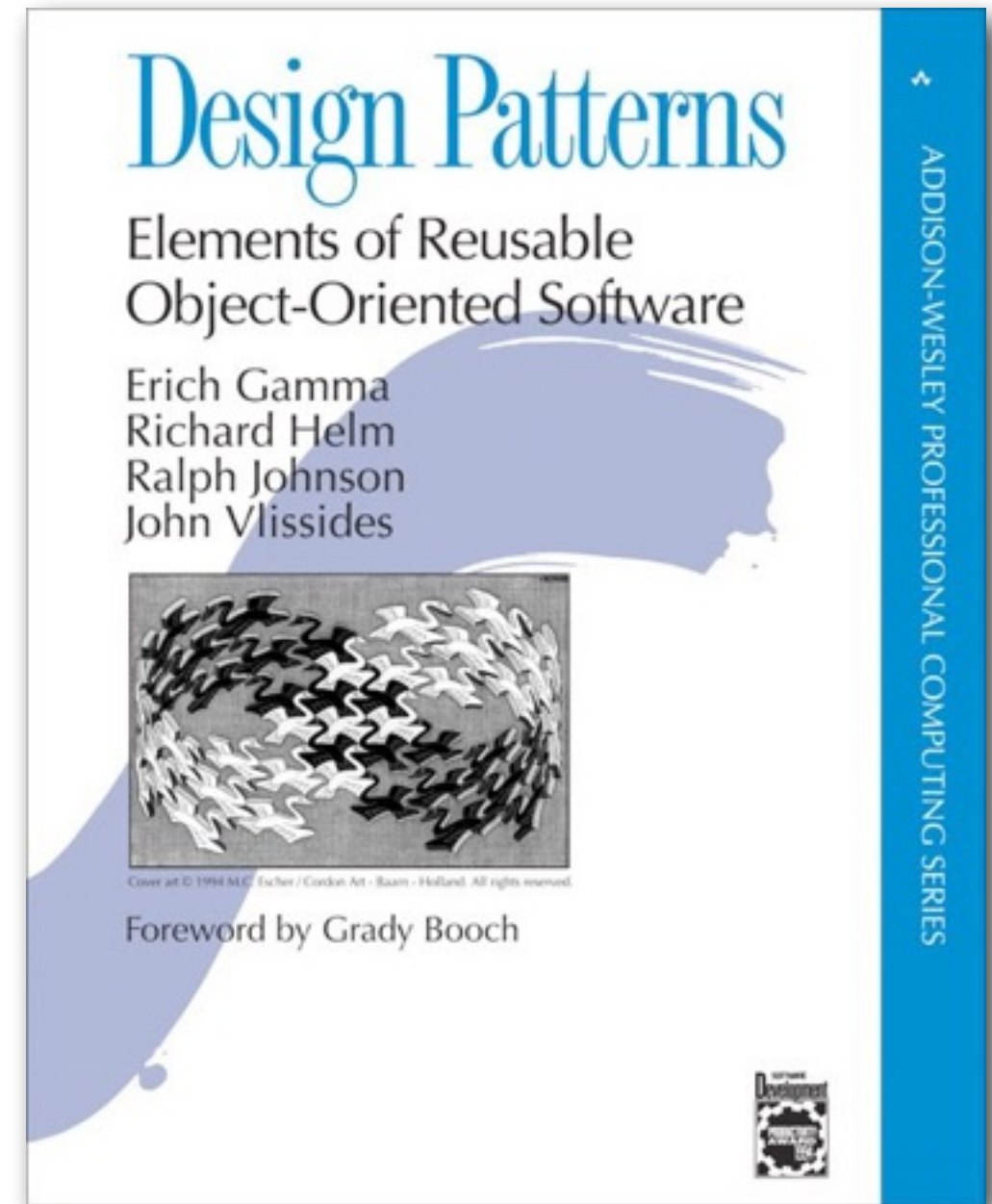
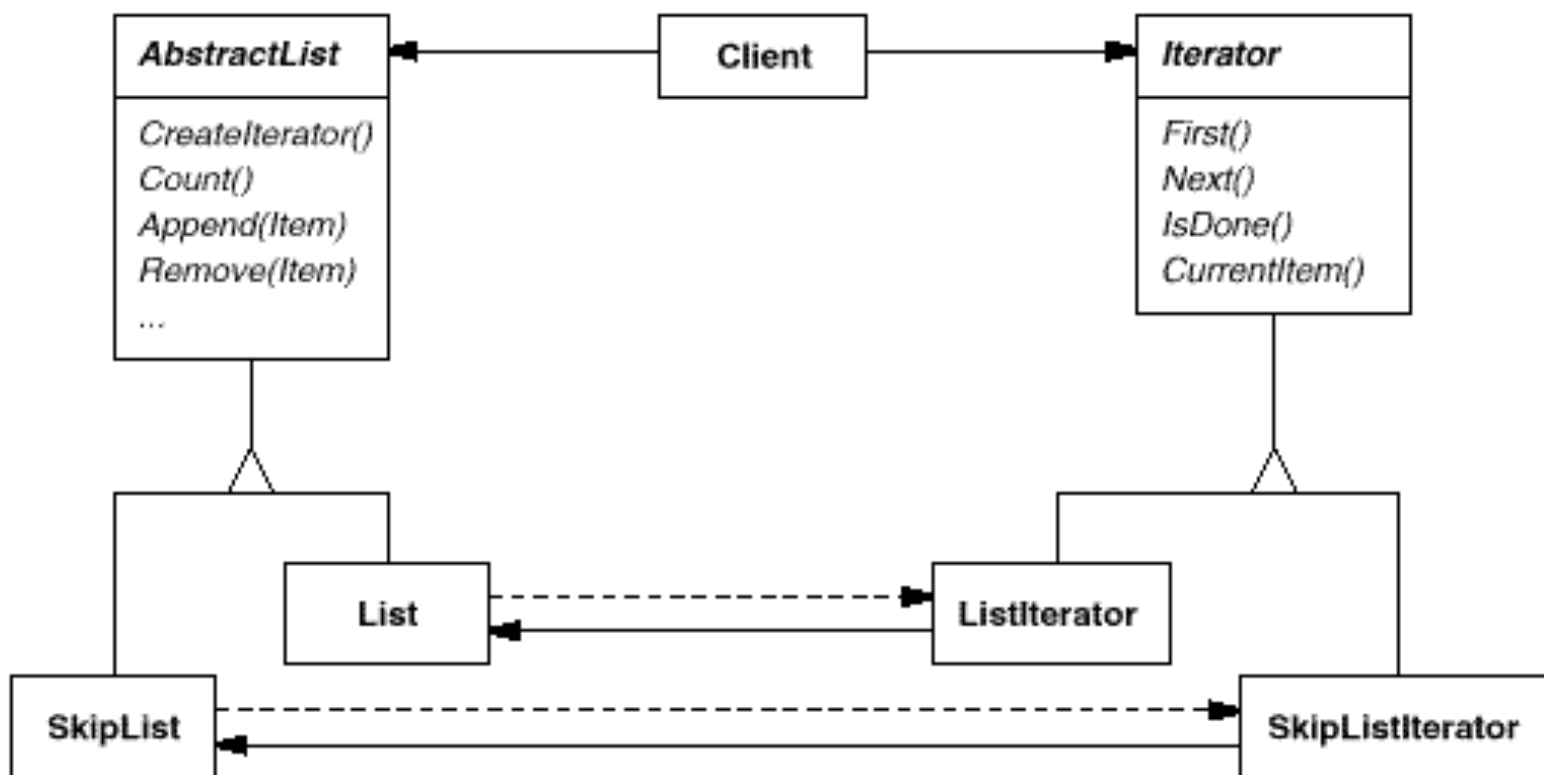
```
import sys
```

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

ITERATOR É UM PADRÃO DE PROJETO

Design Patterns

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





336, 257

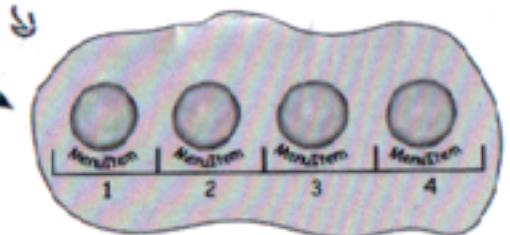
Iterator

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

To iterate over groups of objects without knowing the details of how they're implemented, you just need an Iterator for each group...

ArrayList has a built in iterator...

ArrayList



... one for ArrayList...

Iterator

next()

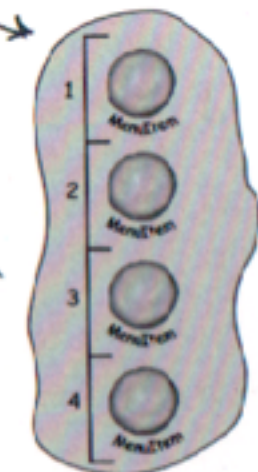
... and one for Array.

next()

Iterator

... Array doesn't have a built-in Iterator, but you can build your own.

Array



You don't have to worry about what implementation is used for the group; you can always use the same interface - Iterator - to iterate over the items in the group.

Head First Design Patterns Poster

O'Reilly

ISBN 0-596-10214-3

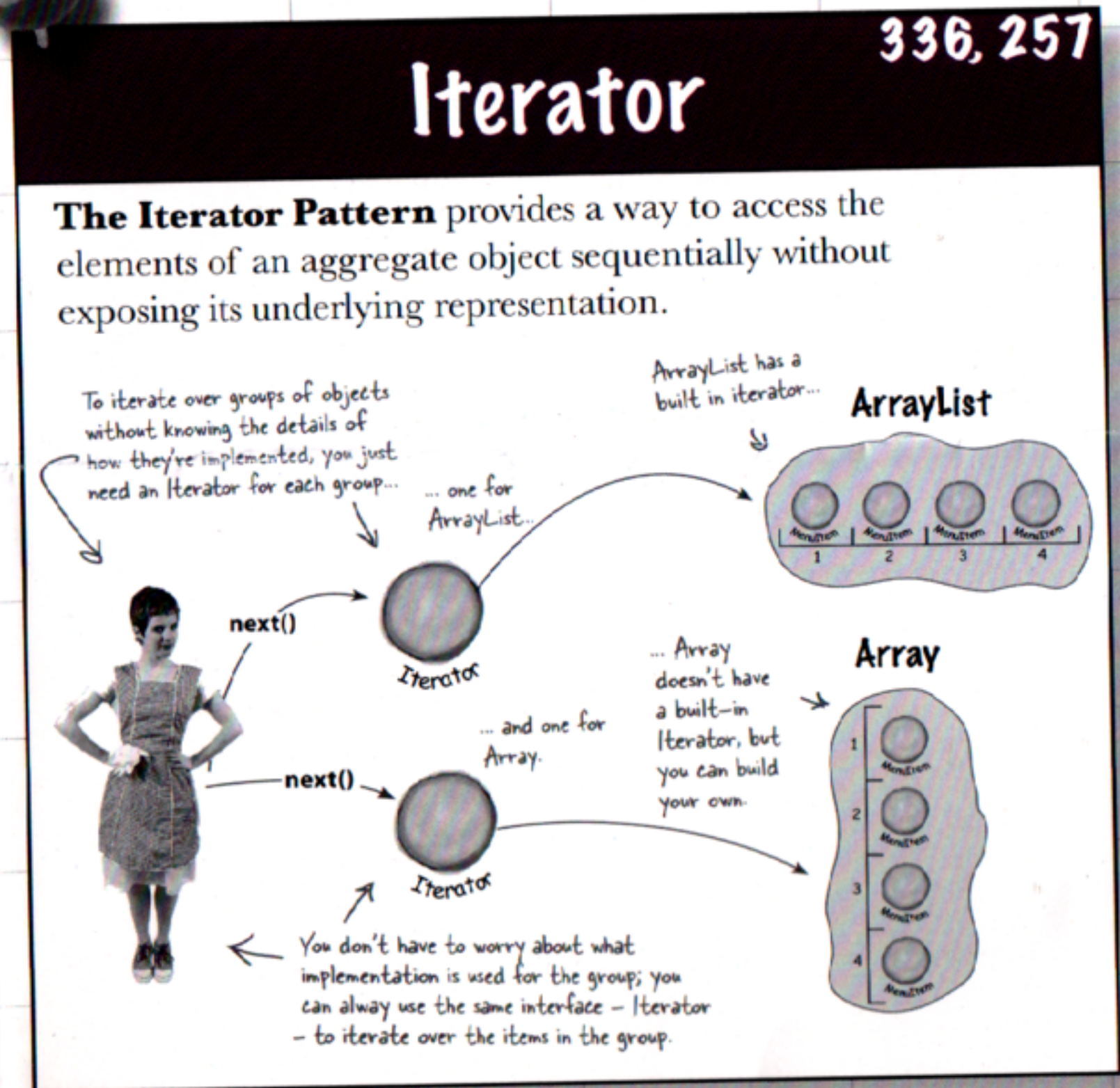
O padrão
Iterator permite
acessar os itens
de uma coleção
sequencialmente,
isolando o cliente
da implementação
da coleção.

Head First Design Patterns Poster

O'Reilly

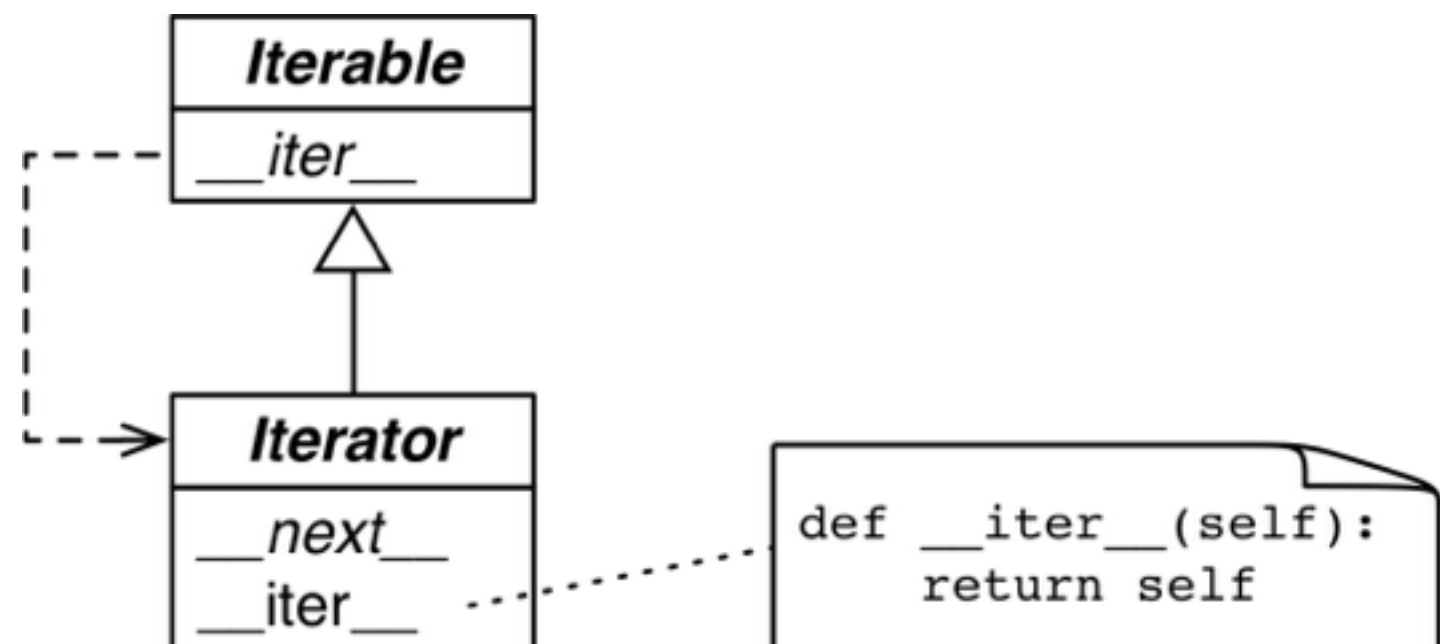
ISBN 0-596-10214-3

sses



ITERÁVEL VERSUS ITERADOR

- Iterável: implementa a interface **Iterable** (método **`__iter__`**)
- Método **`__iter__`** devolve um iterador
- Iterador: implementa a interface **Iterator** (método **`__next__`**)
- Método **`__next__`** devolve o próximo item da série
 - levanta **`StopIteration`** para sinalizar o final da série.
- Em Python, os **iteradores** também são **iteráveis**!

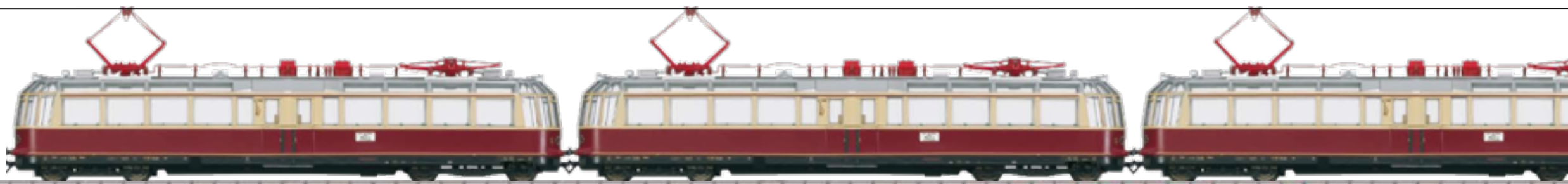


ITERATOR CLÁSSICO

A forma clássica — e não pythônica

UM TREM ITERÁVEL

Uma instância de **Trem** pode ser iterada, vagão por vagão.



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

CÓDIGO DE UM ITERATOR CLÁSSICO

O padrão conforme a receita do livro.

```
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
```

ThoughtWorks®


FUNÇÃO GERADORA

A solução pythônica

UMA FUNÇÃO GERADORA MUITO SIMPLES

Qualquer função que tenha a palavra reservada **yield** em seu corpo é uma *função geradora*.

Quando invocada, a função geradora devolve um objeto gerador



A palavra reservada **gen** foi sugerida no lugar de **def**, mas Guido não topou...

```
>>> 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
```



```

>>> 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
<generator object gen_ab at 0x...>
>>> next(g)
iniciando...
'A'
>>> next(g)
agora vem B:
'B'
>>> next(g)
FIM.
Traceback (most recent call last):
...
StopIteration

```

COMO FUNCIONA

- Invocar a função geradora produz um **objeto gerador**
- O corpo da função só começa a ser executado quando se invoca **next(g)**
- A cada **next(g)**, o corpo da função é executado só até o próximo **yield**.

O FAMOSO GERADOR DE FIBONACCI

Um gerador de série infinita

```
def fibonacci():  
    a, b = 0, 1  
    while True:  
        yield a  
        a, b = b, a + b
```

```
>>> fib = fibonacci()  
>>> for i in range(10):  
...     print(next(fib))  
...  
0  
1  
1  
2  
3  
5  
8  
13  
21  
34
```

GERADOR DE FIBONACCI LIMITADO A "N" ITENS

Mais fácil de usar

```
def fibonacci(n):  
    a, b = 0, 1  
    for _ in range(n):  
        yield a  
        a, b = b, a + b
```

```
>>> for x in fibonacci(10):  
...     print(x)  
...  
0  
1  
1  
2  
3  
5  
8  
13  
21  
34  
>>> list(fibonacci(10))  
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

GERADOR DE PROGRESSÃO ARITMÉTICA

```
def progressão_aritmética(incremento, *, início=0, término=None):  
    infinita = término is None  
    índice = 0  
    resultado = início + incremento * índice  
    while infinita or resultado < término:  
        yield resultado  
        índice += 1  
        resultado = início + incremento * índice
```


```
>>> pa = progressão_aritmética(.1)  
>>> next(pa), next(pa), next(pa), next(pa), next(pa)  
(0.0, 0.1, 0.2, 0.30000000000000004, 0.4)  
>>> from decimal import Decimal  
>>> pa = progressão_aritmética(Decimal('.1'))  
>>> next(pa), next(pa), next(pa), next(pa), next(pa)  
(Decimal('0.0'), Decimal('0.1'), Decimal('0.2'),  
Decimal('0.3'), Decimal('0.4'))  
>>> pa = progressão_aritmética(.5, início=1, término=5)  
>>> list(pa)  
[1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]  
>>> pa = progressão_aritmética(1/3, término=1)  
>>> list(pa)  
[0.0, 0.3333333333333333, 0.6666666666666666]
```

CLASSE TREM COM FUNÇÃO GERADORA

Como um recurso da linguagem torna a receita de um *design pattern* obsoleta:

```
class Trem(object):  
  
    def __init__(self, vagoes):  
        self.vagoes = vagoes  
  
    def __iter__(self):  
        for i in range(self.vagoes):  
            yield 'vagao #%s' % (i+1)
```

Quando invocada, a função geradora devolve um objeto gerador



```
>>> t = Trem(3)  
>>> it = iter(t)  
>>> it  
<generator object __iter__ at 0x...>  
>>> next(it), next(it), next(it)  
( 'vagão #1', 'vagão #2', 'vagão #3' )
```

CONTRASTE: ITERATOR CLÁSSICO × FUNÇÃO GERADORA

Só quem ainda não aprendeu sobre geradores vai querer implementar a receita clássica...

```
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)
```

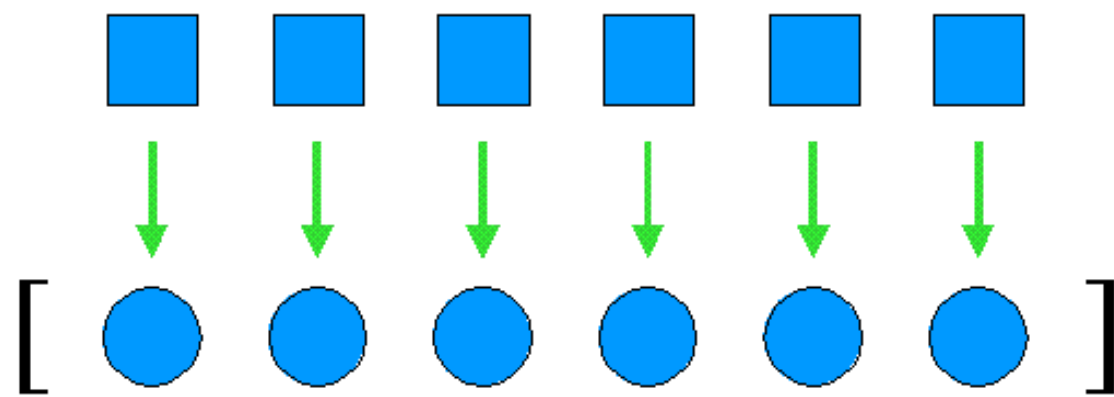
**O gerador
administra
o estado da
iteração
para você**

EXPRESSÕES GERADORAS

Sintaxe ainda mais concisa para criar geradores

LIST COMPREHENSION

Expressão que constrói lista a partir de qualquer iterável finito (desde que haja memória suficiente ;-)



entrada: qualquer iterável

saída: sempre uma lista

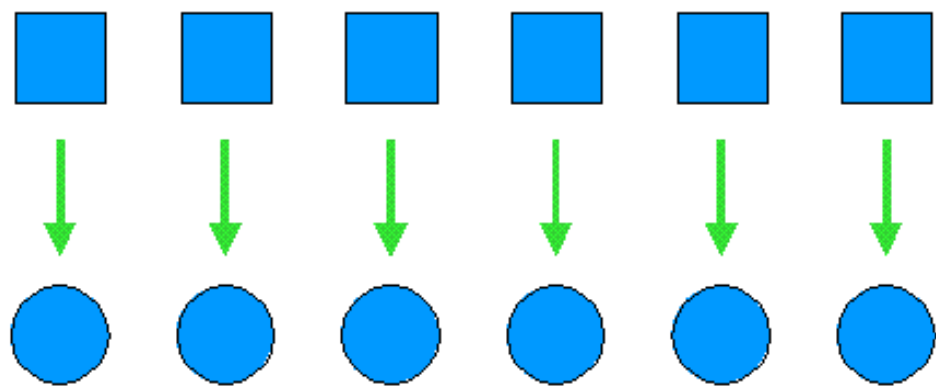
```
>>> s = 'abracadabra'
>>> l = [ord(c) for c in s]
>>> [ord(c) for c in s]
[97, 98, 114, 97, 99, 97, 100, 97, 98, 114, 97]
```

= notação matemática de conjuntos

GENERATOR EXPRESSION

Expressão que constrói gerador a partir de qualquer iterável finito ou não.

O gerador é *lazy*: entrada é consumida um item de cada vez.



entrada: qualquer iterável

saída: sempre um gerador

```
>>> s = 'abracadabra'
>>> g = (ord(c) for c in s)
>>> g
<generator object <genexpr> at 0x102610620>
>>> list(g)
[97, 98, 114, 97, 99, 97, 100, 97, 98, 114, 97]
```

TREM COM EXPRESSÃO GERADORA

Com 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))
```

Com 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)
```

ThoughtWorks®

ITERÁVEIS EM AÇÃO

A solução pythônica

OPERAÇÕES COM ITERÁVEIS

Desempacotamento
de tupla
em atribuições
em chamadas de funções

```
>>> 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
```

```
>>> 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
```

OBJETOS ITERÁVEIS

Alguns exemplos:

str

bytes

list

set

tuple

io.TextIOWrapper

(arquivo texto)

models.query.QuerySet

(Django)

```
>>> with open('1.txt') as arq:
...     for lin in arq:
...         print(lin.rstrip())
...
alfa
beta
gama
delta
```

```
>>> octetos = b'Python'
>>> for octeto in octetos:
...     print(octeto)
...
80
121
116
104
111
110
>>> list(octetos)
[80, 121, 116, 104, 111, 110]
```

```
>>> from django.db import connection
>>> q = connection.queries
>>> q
[]
```

```
>>> from django.db import connection
>>> q = connection.queries
>>> q
[]
>>> from municipios.models import *
>>> res = Municipio.objects.all()[:5]
>>> q
[]
```

```
>>> 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'}]
```



```
>>> 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'}]
```

conclusão:
queryset é
um iterável
preguiçoso
(lazy iterable)

FUNÇÕES EMBUTIDAS QUE CONSOMEM ITERÁVEIS

Funções de redução: aceitam iteráveis finitos e devolvem um valor escalar (ex. um número, ou o maior item etc.)

all

any

max

min

sum

Consome qualquer iterável finito e devolve uma lista ordenada:

sorted

FUNÇÕES GERADORAS EMBUTIDAS

Podem lidar com iteráveis potencialmente ilimitados, e devolvem geradores:

enumerate

filter

map

reversed

zip



O 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()...**

A FUNÇÃO ITER

iter(iterável)

Devolve um iterador sobre o iterável.

Invoca **__iter__** ou constrói um iterador usando **__getitem__** com índices a partir de 0.

iter(função, sentinela)

Constrói um iterador que invoca repetidamente a função até que o um valor igual à sentinela seja gerado.

```
>>> from random import  
randint  
>>> def d6():  
...     return randint(1, 6)  
...  
>>> for dado in iter(d6, 6):  
...     print(dado)  
...  
4  
1  
4  
2  
>>> for dado in iter(d6, 6):  
...     print(dado)  
...  
>>> for dado in iter(d6, 6):  
...     print(dado)  
...  
3  
2  
3
```

EXEMPLO REAL

Caso de uso de funções geradoras para conversão de dados

CONVERSÃO DE GRANDES MASSAS DE DADOS

Contexto

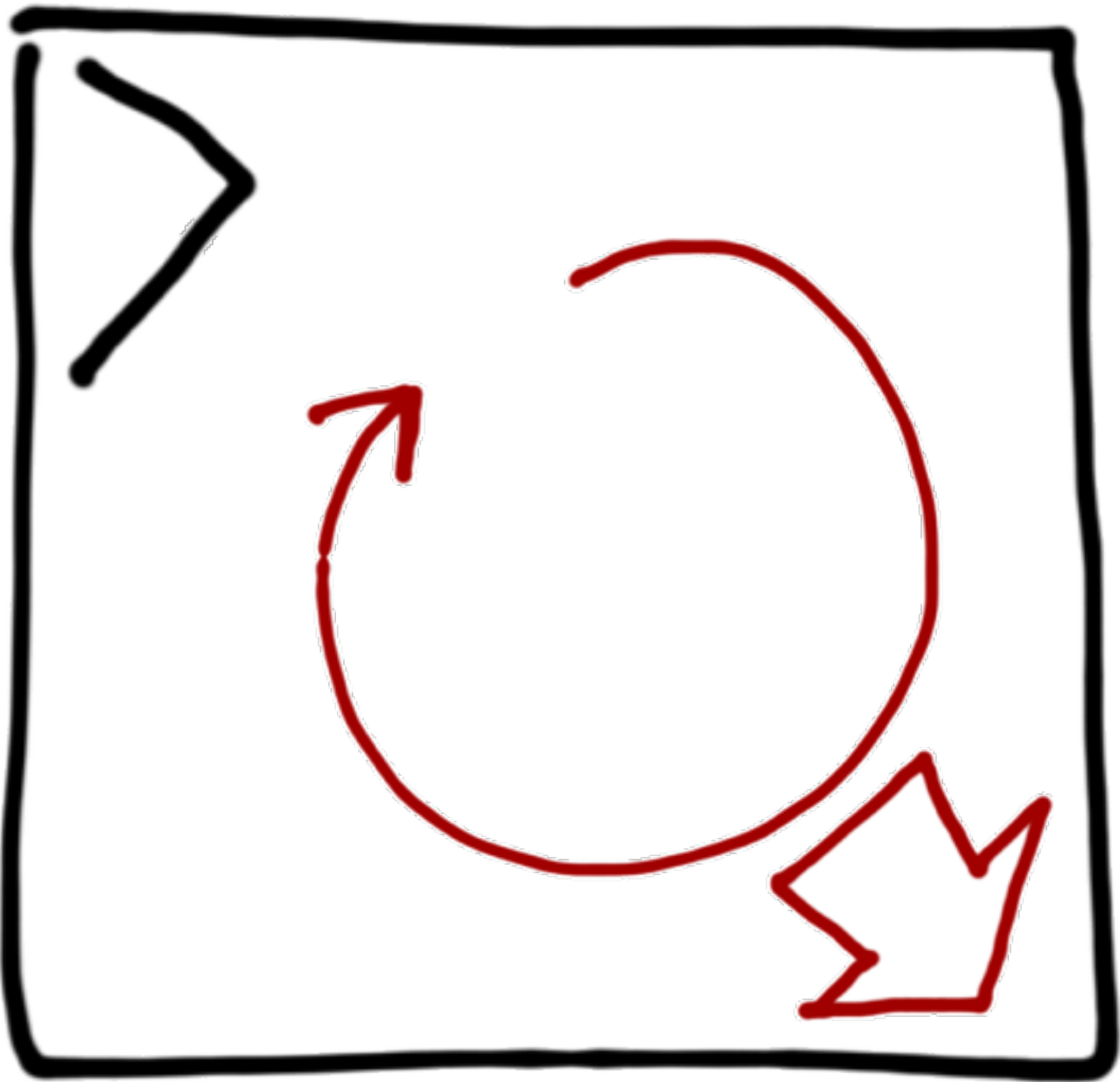
ferramenta para conversão de bases de dados semi-estruturadas.

Uso

funções geradoras para desacoplar laços de leitura e escrita

<https://github.com/fluentpython/isis2json>

LAÇO PRINCIPAL ESCREVE ARQUIVO JSON



```
import sys
import argparse
from json import dumps
import os

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input', type=str, required=True, help='Input file path')
    parser.add_argument('-o', '--output', type=str, required=True, help='Output file path')
    parser.add_argument('-f', '--format', type=str, required=True, help='Output format (json, jsonl, csv, tsv, yaml, xml, etc.)')
    parser.add_argument('-s', '--separator', type=str, required=True, help='Separator character (comma, tab, space, etc.)')
    parser.add_argument('-d', '--delimiter', type=str, required=True, help='Delimiter character (comma, tab, space, etc.)')
    parser.add_argument('-e', '--encoding', type=str, required=True, help='Encoding (utf-8, utf-16, etc.)')
    parser.add_argument('-m', '--mode', type=str, required=True, help='Mode (append, overwrite, etc.)')
    parser.add_argument('-v', '--verbose', type=bool, required=True, help='Verbose mode')
    parser.add_argument('-q', '--quiet', type=bool, required=True, help='Quiet mode')
    parser.add_argument('-h', '--help', type=bool, required=True, help='Help')
    parser.add_argument('--version', type=bool, required=True, help='Version')
    args = parser.parse_args()

    # Check if input file exists
    if not os.path.exists(args.input):
        print(f"Input file {args.input} does not exist.")
        sys.exit(1)

    # Check if output file exists
    if os.path.exists(args.output):
        if args.mode == 'append':
            print(f"Output file {args.output} already exists. Appending.")
        else:
            print(f"Output file {args.output} already exists. Overwriting.")
    else:
        print(f"Output file {args.output} does not exist. Creating.")

    # Open input file
    with open(args.input, 'r', encoding=args.encoding) as f:
        # Read all lines
        lines = f.readlines()

    # Process each line
    for line in lines:
        # Strip newline
        line = line.strip()

        # Skip empty lines
        if not line:
            continue

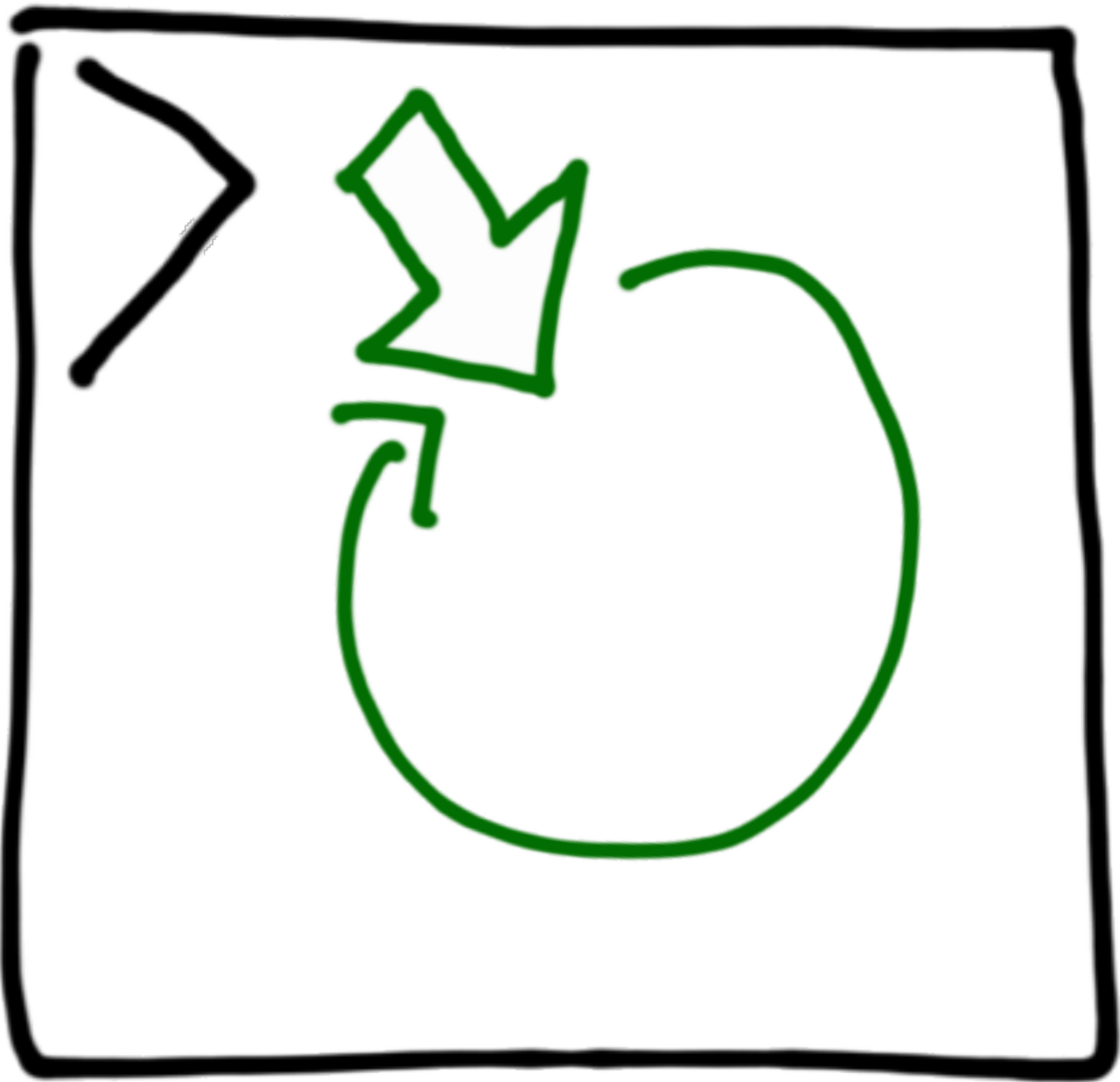
        # Parse JSON
        try:
            data = json.loads(line)
        except json.JSONDecodeError as e:
            print(f"Error parsing JSON: {e}")
            continue

        # Format output
        if args.format == 'json':
            output_line = json.dumps(data, separators=(args.separator, args.delimiter), encoding=args.encoding)
        elif args.format == 'jsonl':
            output_line = json.dumps(data, separators=(args.separator, args.delimiter), encoding=args.encoding) + '\n'
        elif args.format == 'csv':
            output_line = csv.dumps(data, separators=(args.separator, args.delimiter), encoding=args.encoding)
        elif args.format == 'tsv':
            output_line = tsv.dumps(data, separators=(args.separator, args.delimiter), encoding=args.encoding)
        elif args.format == 'yaml':
            output_line = yaml.dumps(data, separators=(args.separator, args.delimiter), encoding=args.encoding)
        elif args.format == 'xml':
            output_line = xml.dumps(data, separators=(args.separator, args.delimiter), encoding=args.encoding)
        else:
            print(f"Unsupported format: {args.format}")
            continue

        # Write to output file
        with open(args.output, 'a', encoding=args.encoding) as f:
            f.write(output_line)

    # Print summary
    print(f"Processed {len(lines)} lines. Output file: {args.output}")
```


UM OUTRO LAÇO LÊ REGISTROS A CONVERTER



```
import sys
import argparse
from math import sqrt
import os

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', type=str, required=True)
    parser.add_argument('--output', type=str, required=True)
    parser.add_argument('--format', type=str, required=True)
    parser.add_argument('--verbose', type=bool, default=False)
    args = parser.parse_args()

    input_file = args.input
    output_file = args.output
    format = args.format
    verbose = args.verbose

    if not os.path.exists(input_file):
        print(f"Input file {input_file} does not exist.")
        sys.exit(1)

    if not os.path.exists(output_file):
        print(f"Output file {output_file} does not exist.")
        sys.exit(1)

    if format not in ['json', 'xml', 'yaml']:
        print(f"Format {format} is not supported.")
        sys.exit(1)

    if verbose:
        print(f"Input file: {input_file}")
        print(f"Output file: {output_file}")
        print(f"Format: {format}")
        print(f"Verbose: {verbose}")

    # Read input data
    with open(input_file, 'r') as f:
        data = f.read()

    # Parse input data
    if format == 'json':
        data = json.loads(data)
    elif format == 'xml':
        data = xml.etree.ElementTree.fromstring(data)
    elif format == 'yaml':
        data = yaml.safe_load(data)

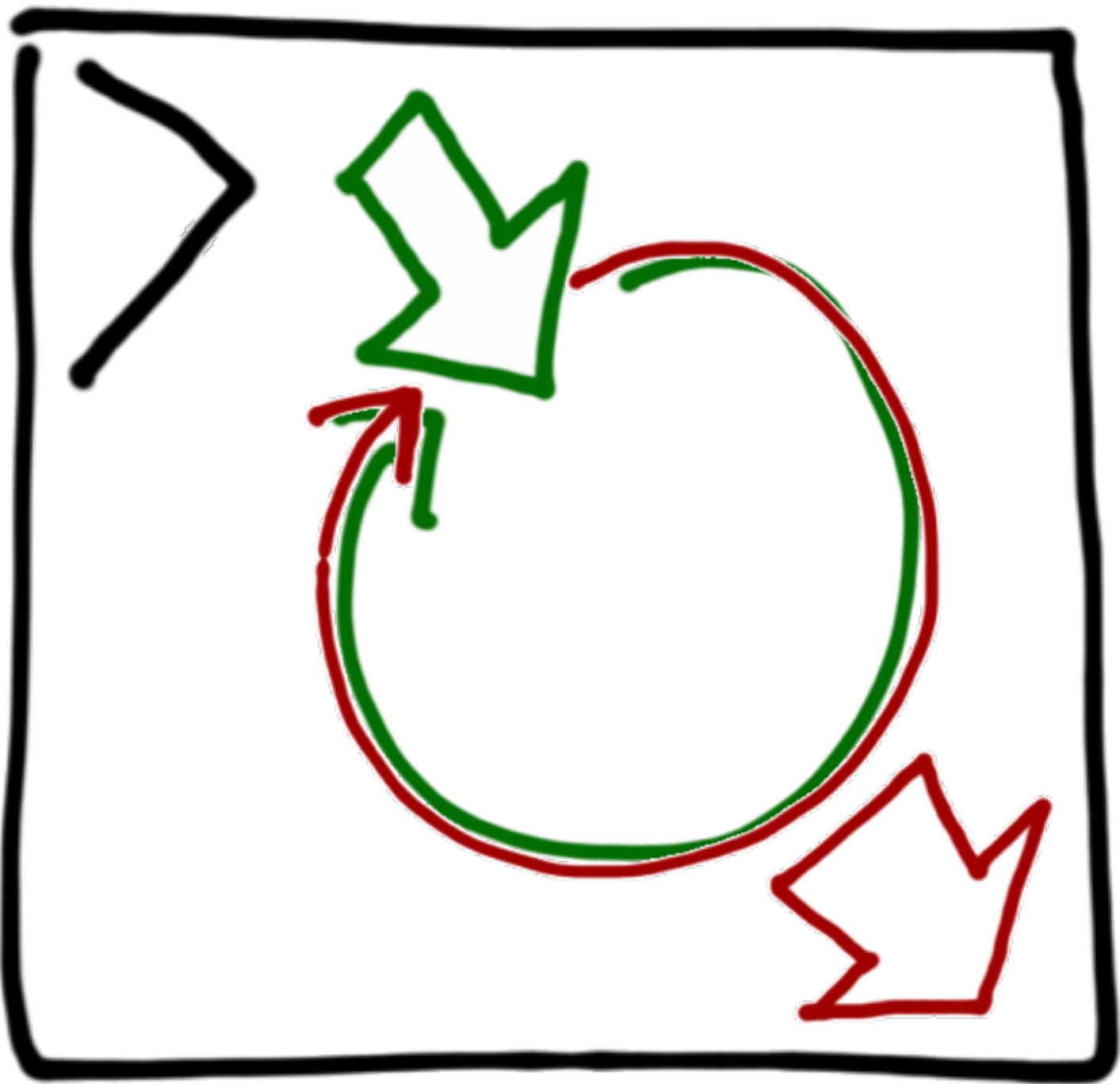
    # Convert data
    if format == 'json':
        data = json.dumps(data, indent=2)
    elif format == 'xml':
        data = xml.etree.ElementTree.tostring(data, encoding='utf-8')
    elif format == 'yaml':
        data = yaml.dump(data, default_flow_style=False)

    # Write output data
    with open(output_file, 'w') as f:
        f.write(data)

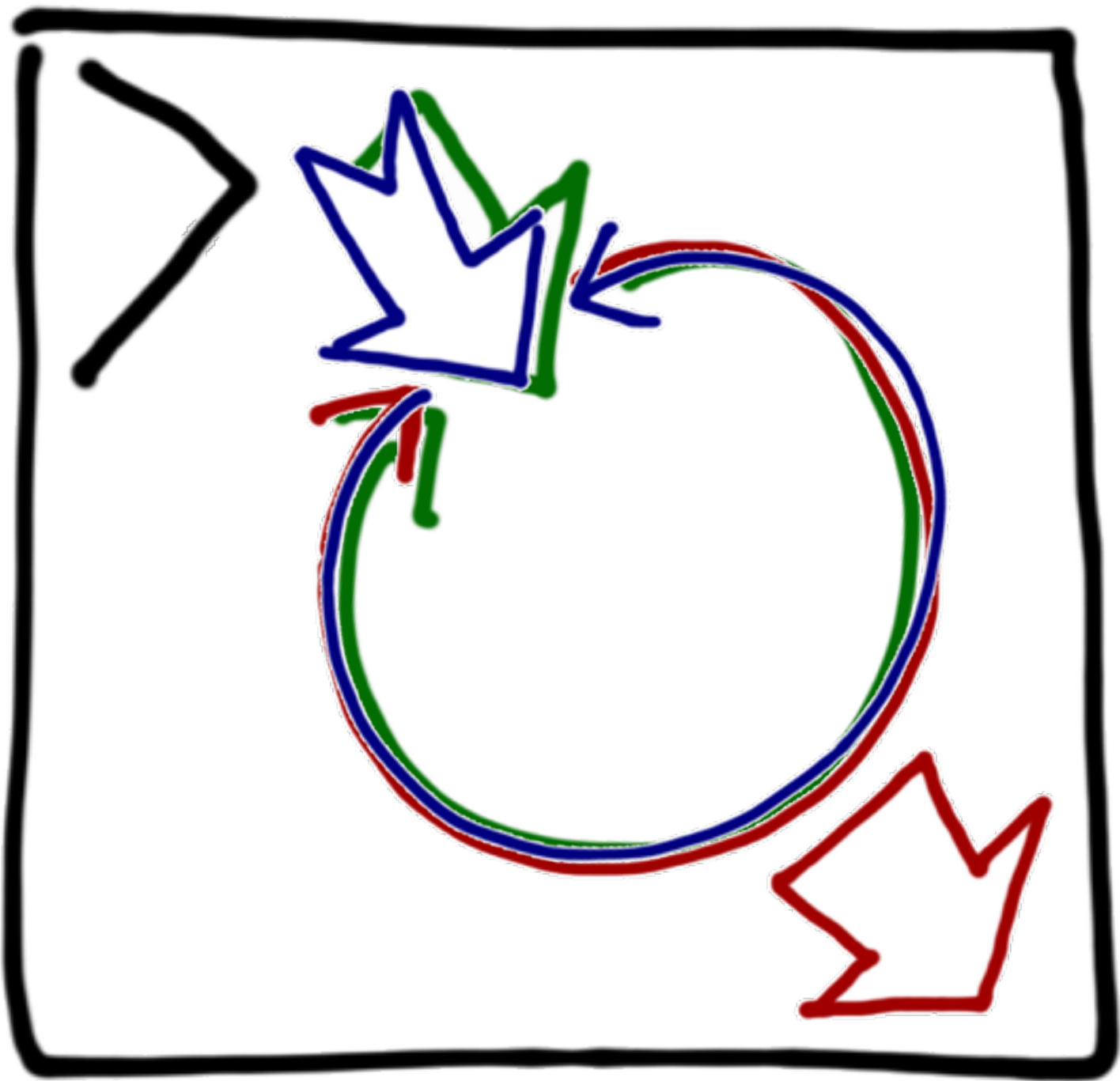
    if verbose:
        print(f"Output file: {output_file}")
        print(f"Format: {format}")
        print(f"Verbose: {verbose}")

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

SOLUÇÃO POSSÍVEL: MESMO LAÇO LÊ E GRAVA

[illegible]

MAS E A LÓGICA PARA LER OUTRO FORMATO?



```

import sys
import argparse
import re
import math

def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', type=str, required=True)
    parser.add_argument('--output', type=str, required=True)
    parser.add_argument('--mode', type=str, required=True)
    parser.add_argument('--seed', type=int, default=42)
    parser.add_argument('--batch_size', type=int, default=100)
    parser.add_argument('--num_epochs', type=int, default=100)
    parser.add_argument('--num_workers', type=int, default=4)
    parser.add_argument('--device', type=str, default='cuda')
    parser.add_argument('--verbose', type=bool, default=True)
    parser.add_argument('--log_dir', type=str, default='./logs')
    parser.add_argument('--save_dir', type=str, default='./saves')
    parser.add_argument('--load_dir', type=str, default='./loads')
    parser.add_argument('--resume', type=bool, default=False)
    parser.add_argument('--test_only', type=bool, default=False)
    parser.add_argument('--eval_only', type=bool, default=False)
    parser.add_argument('--train_only', type=bool, default=False)
    parser.add_argument('--all', type=bool, default=False)
    parser.add_argument('--help', type=bool, default=False)
    args = parser.parse_args()
    return args

def main():
    args = parse_args()
    if args.help:
        parser.print_help()
        return
    if args.verbose:
        print('Mode: %s' % args.mode)
        print('Input: %s' % args.input)
        print('Output: %s' % args.output)
        print('Seed: %d' % args.seed)
        print('Batch size: %d' % args.batch_size)
        print('Num epochs: %d' % args.num_epochs)
        print('Num workers: %d' % args.num_workers)
        print('Device: %s' % args.device)
        print('Log dir: %s' % args.log_dir)
        print('Save dir: %s' % args.save_dir)
        print('Load dir: %s' % args.load_dir)
        print('Resume: %s' % args.resume)
        print('Test only: %s' % args.test_only)
        print('Eval only: %s' % args.eval_only)
        print('Train only: %s' % args.train_only)
        print('All: %s' % args.all)
    # Create directories
    if not os.path.exists(args.log_dir):
        os.makedirs(args.log_dir)
    if not os.path.exists(args.save_dir):
        os.makedirs(args.save_dir)
    if not os.path.exists(args.load_dir):
        os.makedirs(args.load_dir)
    # Set random seed
    random.seed(args.seed)
    torch.manual_seed(args.seed)
    np.random.seed(args.seed)
    # Create data loaders
    train_loader, val_loader, test_loader = create_data_loaders(
        args.input, args.batch_size, args.num_workers, args.device)
    # Create model
    model = create_model(args)
    # Create optimizer
    optimizer = create_optimizer(model)
    # Create scheduler
    scheduler = create_scheduler(optimizer)
    # Create trainer
    trainer = create_trainer(model, optimizer, scheduler, train_loader, val_loader, test_loader, args)
    # Train
    trainer.train()
    # Evaluate
    trainer.evaluate()
    # Save
    trainer.save()
    # Load
    trainer.load()
    # Test
    trainer.test()
    # All
    trainer.all()

if __name__ == '__main__':
    main()

```

SOLUÇÃO: FUNÇÕES GERADORAS

iterMstRecords

função geradora: lê registros MST

iterIsoRecords

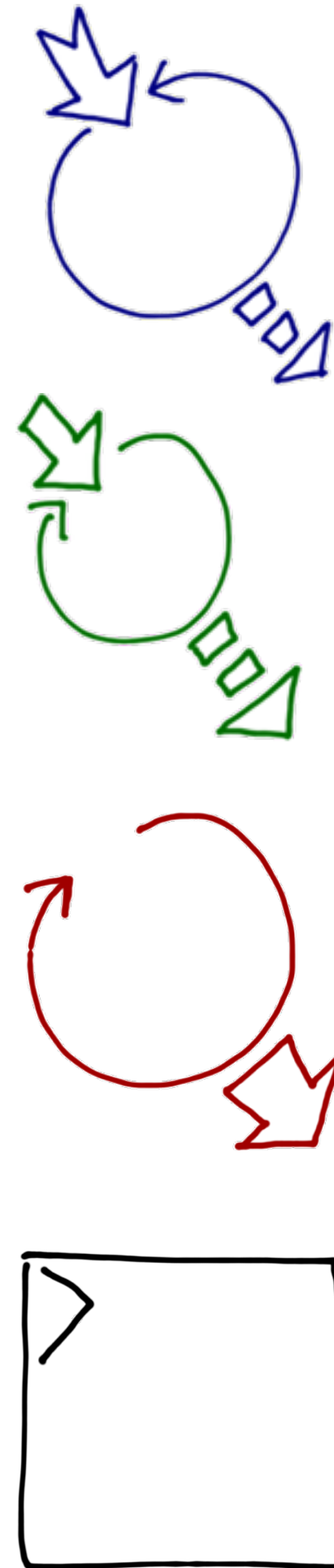
função geradora: lê registros ISO-2709

writeJsonArray

itera por registros; salva em novo formato

main

função principal



```

import sys
import argparse
from math import sqrt, ceil

parser = argparse.ArgumentParser()
parser.add_argument(
    "-i", "--input", type=str, required=True,
    help="Input file path (must be a text file)"
)
parser.add_argument(
    "-o", "--output", type=str, required=True,
    help="Output file path (must be a text file)"
)
args = parser.parse_args()

input_file = args.input
output_file = args.output

# Read input data
with open(input_file, "r") as f:
    lines = f.readlines()

# Parse input data
def parse_line(line):
    """Parse a line of input data into a dictionary of values"""
    line = line.strip()
    if not line:
        return {}
    parts = line.split(",")
    values = {}
    for part in parts:
        key, value = part.split(":")
        values[key] = value
    return values

# Parse all lines
data = []
for line in lines:
    data.append(parse_line(line))

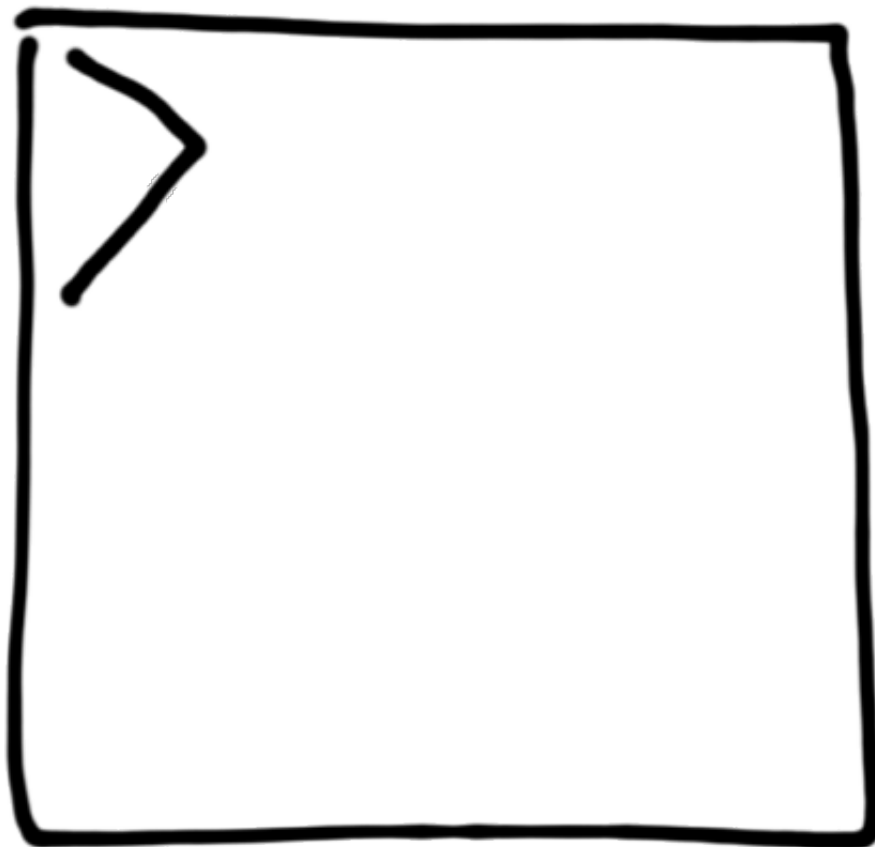
# Calculate statistics
def calculate_statistics(data):
    """Calculate statistics for the input data"""
    n = len(data)
    if n == 0:
        return {}
    # Calculate mean
    mean = sum([v["value"] for v in data]) / n
    # Calculate variance
    variance = sum([(v["value"] - mean) ** 2] for v in data) / n
    # Calculate standard deviation
    std_dev = sqrt(variance)
    return {"mean": mean, "variance": variance, "std_dev": std_dev}

# Write output
def write_output(output_file, statistics):
    """Write the calculated statistics to the output file"""
    with open(output_file, "w") as f:
        f.write(f"Mean: {statistics['mean']}\n")
        f.write(f"Variance: {statistics['variance']}\n")
        f.write(f"Standard Deviation: {statistics['std_dev']}\n")

# Main execution
statistics = calculate_statistics(data)
write_output(output_file, statistics)

```


MAIN: LER ARGUMENTOS



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

    # add the arguments
    parser.add_argument(
        'file_name', metavar='INPUT.{mat|iso}',
        help='.mat 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 .mat (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 .mat 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('.mat'):
        iterRecords = iterMatRecords
    else:
        if args.mfn:
            print('UNSUPPORTED: -n/--mfn option only available for .mat 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()
```

MAIN: SELEÇÃO DO GERADOR 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('UNSUPPORTED: -m/--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

ESCREVER REGISTROS JSON

Laço de saída em
writeJsonArray



```
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')
```

ESCREVER REGISTROS JSON

writeJsonArray itera pelo gerador construído pela função que recebe como primeiro argumento.

```
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)
```


LER REGISTROS ISO-2709

Laço de entrada em **iterJsonRecords** lê cada registro ISO-2709 e produz um dicionário com seus campos

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()
```

LER REGISTROS ISO-2709

```
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

LER REGISTROS .MST

Laço de entrada em **iterMstRecords** lê cada registro .MST e produz um dicionário com seus campos

função geradora!



```
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()
```



```

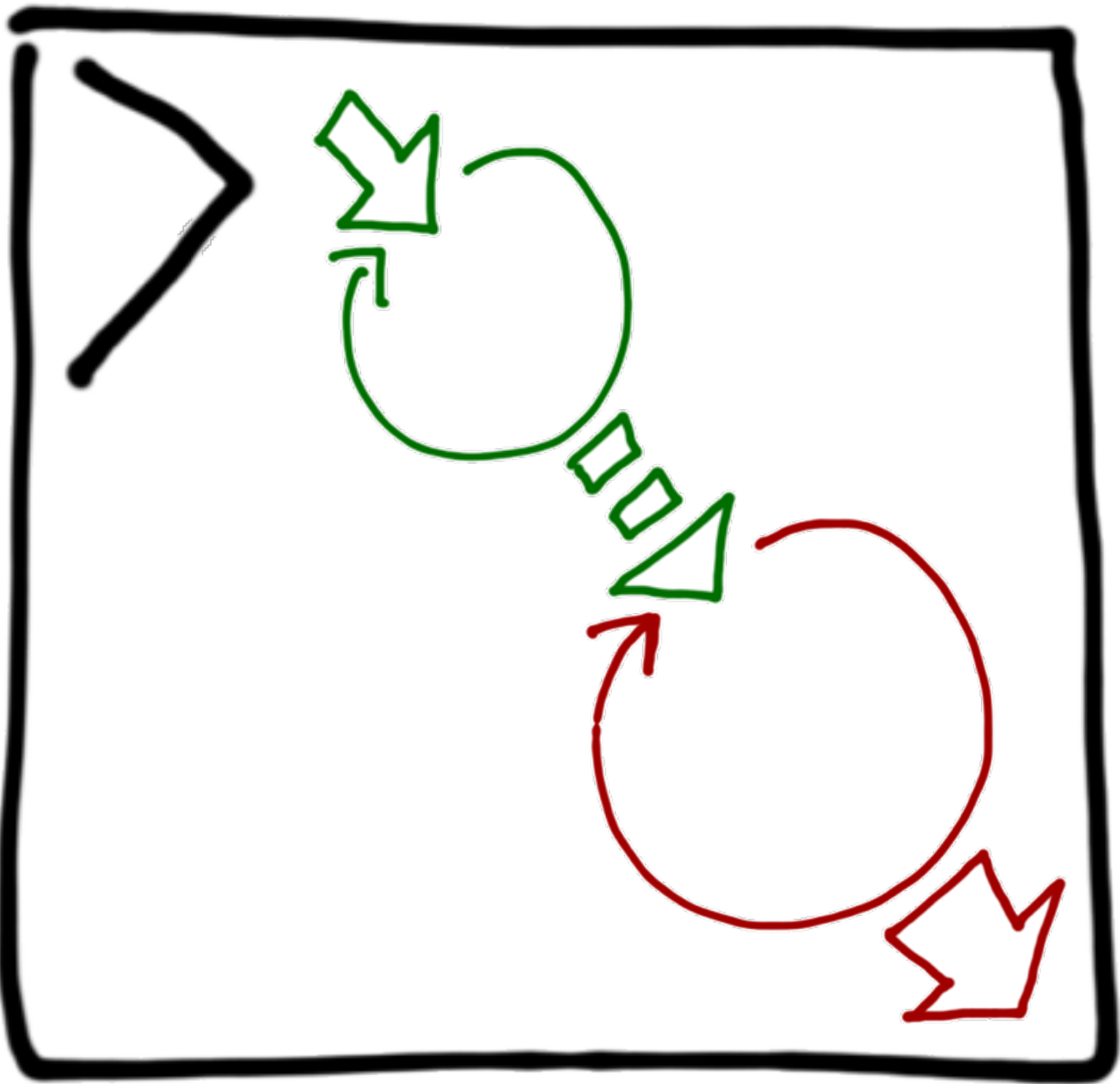
def iterMstRecords(master_file_name, isis_json_type):
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('IMPORT ERROR: Jython 2.5 and Bruma.jar are required '
              'to read .mst files')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()
    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()

```

cria um novo dict
a cada iteração

produz (yield) registro
na forma de um dict

GERADORES NA PRÁTICA



```
import sys
import argparse
import uuid
import os

try:
    import json
except ImportError:
    if os.name == 'java': # running Jython
        from com.mongodb import JythonCode as json
    else:
        import simplejson as json

SKIP_INACTIVE = True
DEFAULT_QTY = 2**31
ISIS_MFM_KEY = 'mfm'
ISIS_ACTIVE_KEY = 'active'
SUBFIELD_DELIMITERS = '-'
INPUT_ENCODING = 'utf8'

def iterMatRecords(master_file_name, isia_json_type):
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('UNUSUT: bruma: Jython 2.5 and Bruma.jar are required '
              'to read .mat files')
        raise SystemExit
    mat = MasterFactory.getInstance(master_file_name).open()
    for record in mat:
        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_MFM_KEY] = record.getMFM()
        for field in record.getFields():
            field_key = str(field.getKey())
            field_occurrences = fields.setdefault(field_key, [])
            if isia_json_type == 1:
                content = {}
                for subfield in field.getSubfields():
                    subfield_key = subfield.getKey()
                    content[subfield_key] = subfield.getContent()
            else:
                subfield_occurrences = content.setdefault(subfield_key, [])
                subfield_occurrences.append(subfield.getContent())
            field_occurrences.append(content)
        elif isia_json_type == 1:
            content = {}
            for subfield in field.getSubfields():
                subfield_key = subfield.getKey()
                if subfield_key == '-':
                    content.insert(0, subfield.getContent())
                else:
                    content.append(SUBFIELD_DELIMITERS+subfield_key+
                                  subfield.getContent())
            field_occurrences.append(''.join(content))
        else:
            raise NotImplementedError(
                'ISIS-JSON type is conversion not yet '
                'implemented for .mat input' % isia_json_type)
        yield field
    mat.close()

def iterIsoRecords(iso_file_name, isia_json_type):
    from iso709 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 zeros
            field_occurrences = fields.setdefault(field_key, [])
            content = field.value.decode(INPUT_ENCODING, 'replace')
            if isia_json_type == 1:
                field_occurrences.append(content)
            elif isia_json_type == 2:
                field_occurrences.append(expand(content))
            elif isia_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError(
                    'ISIS-JSON type is conversion not yet '
                    'implemented for .iso input' % isia_json_type)
        yield fields
    iso.close()

def writeJsonArray(iterRecords, file_name, output, qty, skip, id_tag,
                  gen_uuid, mongo, mfm, isia_json_type, prefix, constant):
    start = skip
    end = start + qty
    if not mongo:
        output.write('')
    if id_tag:
        id_tag = str(id_tag)
        id = set()
    else:
        id_tag = ''
    for i, record in enumerate(iterRecords(file_name, isia_json_type)):
        if i >= end:
            break
        if i > start and not mongo:
            output.write('\n')
        if start == 1 < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
                if occurrences is None:
                    msg = 'id tag %s not found in record %s' %
                        (id_tag, record[ISIS_MFM_KEY])
                    raise ValueError(msg % (id_tag, i))
                if len(occurrences) > 1:
                    msg = 'multiple id tags %s found in record %s' %
                        (id_tag, record[ISIS_MFM_KEY])
                    raise ValueError(msg % (id_tag, i))
                else: # ok, we have one and only one id field
                    if isia_json_type == 1:
                        id = occurrences[0]
                    elif isia_json_type == 2:
                        id = occurrences[0][0]
                    elif isia_json_type == 3:
                        id = occurrences[0]['']
                    if id in id:
                        msg = 'duplicate id %s in tag %s, record %s' %
                            (id, id_tag, record[ISIS_MFM_KEY])
                        raise ValueError(msg % (id, id_tag, i))
                    record[id] = id
                    id.add(id)
            elif gen_uuid:
                record[id] = unicode(uuid())
            elif mfm:
                record[id] = record[ISIS_MFM_KEY]
            if prefix:
                # iterate over a fixed sequence of tags
                for tag in tuple(record):
                    if str(tag).isdigit():
                        record[prefix+tag] = record[tag]
                    # 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')

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

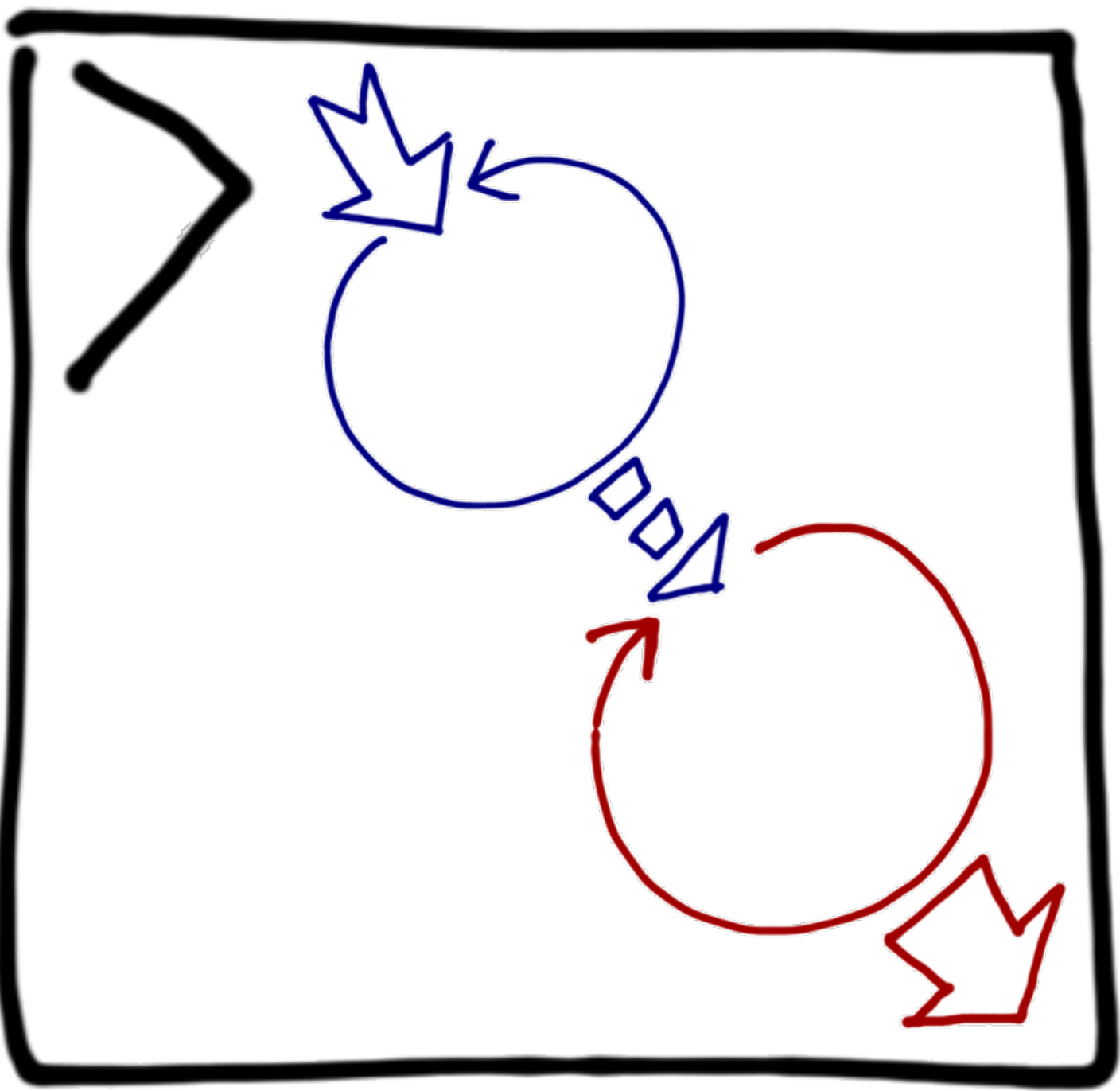
    # add the arguments
    parser.add_argument(
        'file_name', metavar='INPUT', (mat|iso),
        help='mat or .iso file to read')
    parser.add_argument(
        '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
        help='the file where the JSON output should be written'
        ' (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 mongolaport utility')
    parser.add_argument(
        '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
        help='ISIS-JSON type, sets field structure: 1=string, 2=list, 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 .mat (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', '--mfm', action='store_true',
        help='generate an "_id" from the MFM of each record'
        '(available only for .mat input)')
    parser.add_argument(
        '-v', '--constant', type=str, metavar='TAG:VALUE', default='',
        help='Include a constant tag/value in every record (ex. -k type:Id)')
    ...

    # 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 to repeat until end of input)')
    ...

    # parse the command line
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mat'):
        iterRecords = iterMatRecords
    else:
        if args.mfm:
            print('UNUSUT: -n/-mfn option only available for .mat input.')
            raise SystemExit
        iterRecords = iterIsoRecords
    if args.couch:
        args.out.write(['docs' + '\n'])
        writeJsonArray(iterRecords, args.file_name, args.out, args.qty, args.skip,
            args.id, args.uuid, args.mongo, args.mfm, args.type, args.prefix,
            args.constant)
    if args.couch:
        args.out.write('\n')
        args.out.close()

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

GERADORES NA PRÁTICA



```
import sys
import argparse
import uuid
import os

try:
    import json
except ImportError:
    if os.name == 'java': # running Jython
        from com.mongodb import JythonCode as json
    else:
        import simplejson as json

SKIP_INACTIVE = True
DEFAULT_QTY = 2**31
ISIS_MFM_KEY = 'mfm'
ISIS_ACTIVE_KEY = 'active'
SUBFIELD_DELIMITERS = ' '
INPUT_ENCODING = 'utf8'

def iterMatRecords(master_file_name, isia_json_type):
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('UNUSUTED: Jython 2.5 and Bruma.jar are required '
              'to read .mat files')
        raise SystemExit
    mat = MasterFactory.getInstance(master_file_name).open()
    for record in mat:
        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_MFM_KEY] = record.getMFM()
        for field in record.getFields():
            field_key = str(field.getKey())
            field_occurrences = fields.setdefault(field_key, [])
            if isia_json_type == 1:
                content = {}
                for subfield in field.getSubfields():
                    subfield_key = subfield.getKey()
                    if subfield_key == '':
                        content[subfield_key] = subfield.getContent()
                    else:
                        subfield_occurrences = content.setdefault(subfield_key, [])
                        subfield_occurrences.append(subfield.getContent())
            field_occurrences.append(content)
        elif isia_json_type == 1:
            content = {}
            for subfield in field.getSubfields():
                subfield_key = subfield.getKey()
                if subfield_key == '':
                    content[subfield_key] = subfield.getContent()
                else:
                    content[subfield_key] = subfield.getContent()
            field_occurrences.append(''.join(content))
        else:
            raise NotImplementedError(
                'ISIS-JSON type is conversion not yet '
                'implemented for .mat input' % isia_json_type)
    yield fields
    mat.close()

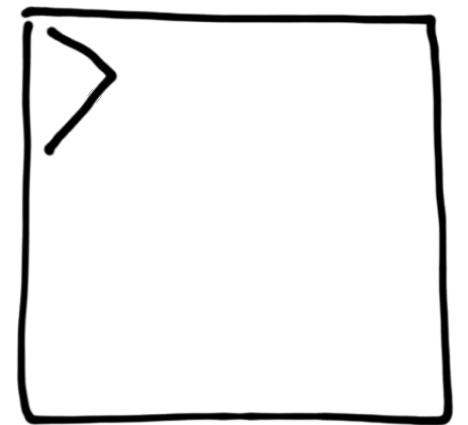
def iterIsoRecords(iso_file_name, isia_json_type):
    from iso709 import IsoFile
    from subfield import expand
    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = {}
        for field in record.directory:
            field_key = str(int(field.tag)) # remove leading zeros
            field_occurrences = fields.setdefault(field_key, [])
            content = field.value.decode(INPUT_ENCODING, 'replace')
            if isia_json_type == 1:
                field_occurrences.append(content)
            elif isia_json_type == 2:
                field_occurrences.append(expand(content))
            elif isia_json_type == 3:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError(
                    'ISIS-JSON type is conversion not yet '
                    'implemented for .iso input' % isia_json_type)
        yield fields
    iso.close()

def writeJsonArray(iterRecords, file_name, output, qty, skip, id_tag,
                  gen_uuid, mongo, mfm, isia_json_type, prefix, constant):
    start = skip
    end = start + qty
    if not mongo:
        output.write('')
    if id_tag:
        id_tag = str(id_tag)
        id = set()
    else:
        id_tag = ''
    for i, record in enumerate(iterRecords(file_name, isia_json_type)):
        if i >= end:
            break
        if i > start and not mongo:
            output.write('')
        if start == 1 < end:
            if id_tag:
                occurrences = record.get(id_tag, None)
                if occurrences is None:
                    msg = 'id tag %s not found in record %s' % (id_tag, i)
                    if ISIS_MFM_KEY in record:
                        msg = msg + ' (mfm=%s)' % record[ISIS_MFM_KEY]
                    raise ValueError(msg % (id_tag, i))
                if len(occurrences) > 1:
                    msg = 'multiple id tags %s found in record %s' % (id_tag, i)
                    if ISIS_MFM_KEY in record:
                        msg = msg + ' (mfm=%s)' % record[ISIS_MFM_KEY]
                    raise ValueError(msg % (id_tag, i))
                else: # ok, we have one and only one id field
                    if isia_json_type == 1:
                        id = occurrences[0]
                    elif isia_json_type == 2:
                        id = occurrences[0][0]
                    elif isia_json_type == 3:
                        id = occurrences[0]['']
                    if id in id:
                        msg = 'duplicate id %s in tag %s, record %s' % (id, id_tag, i)
                        if ISIS_MFM_KEY in record:
                            msg = msg + ' (mfm=%s)' % record[ISIS_MFM_KEY]
                            raise ValueError(msg % (id, id_tag, i))
                        record['id'] = id
                    id.add(id)
            elif gen_uuid:
                record['_id'] = unicode(uuid())
            elif mfm:
                record['_mfm'] = record[ISIS_MFM_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('')
            output.write('')

def main():
    # create the parser
    parser = argparse.ArgumentParser(
        description='Convert an ISIS .mat or .iso file to a JSON array')
    # add the arguments
    parser.add_argument(
        'file_name', metavar='INPUT(.mat|.iso)',
        help='mat or .iso file to read')
    parser.add_argument(
        '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
        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 mongolaport utility')
    parser.add_argument(
        '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
        help='ISIS-JSON type, sets field structure: 1=string, 2=list, 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 .mat (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', '--uid', 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', '--mfm', action='store_true',
        help='generate an "_mfm" from the MFM of each record'
        '(available only for .mat input)')
    parser.add_argument(
        '-v', '--constant', type=str, metavar='TAG:VALUE', default='',
        help='Include a constant tag/value in every record (ex. -k type:Id)')
    ...
    # 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 -s to repeat until end of input)')
    ...
    # parse the command line
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mat'):
        iterRecords = iterMatRecords
    else:
        if args.mfm:
            print('UNUSUTED: -n/-mfm option only available for .mat 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.mfm, args.type, args.prefix,
            args.constant)
    if args.couch:
        args.out.write(']')
        args.out.close()
    if __name__ == '__main__':
        main()
```

VISÃO GERAL DA SOLUÇÃO

Duas funções geradoras de entrada para alimentar um laço de saída. Fácil estender para mais formatos de entrada!



```
def iterMatRecords(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 .mat files')
        raise SystemExit
    mat = MasterFactory.getInstance(master_file_name).open()
    for record in mat:
        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_key] = 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 .mat input' % isis_json_type)
    yield fields
    mat.close()
```

```
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()
```

```
def writeJsonArray(iterRecords, file_name, output, qty, skip, id_tag,
                  gen_uid, 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(',')
        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' % (id_tag, i)
                    if ISIS_MFN_KEY in record:
                        mfn = record[ISIS_MFN_KEY]
                        raise KeyError(msg % (id_tag, i))
                    if len(occurrences) > 1:
                        msg = 'multiple id tags %s found in record %s' % (id_tag, i)
                        if ISIS_MFN_KEY in record:
                            mfn = 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][1]
                    if id in ids:
                        msg = 'duplicate id %s in tag %s, record %s' % (id, id_tag, i)
                        if ISIS_MFN_KEY in record:
                            mfn = record[ISIS_MFN_KEY]
                            raise TypeError(msg % (id, id_tag, i))
                        record['id'] = id
                        ids.add(id)
            elif gen_uid:
                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):
                    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('{}')
```

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

    # add the arguments
    parser.add_argument(
        'file_name', metavar='INPUT.mat[iso]',
        help='.mat or .iso file to read')
    parser.add_argument(
        '-o', '--out', type=argparse.FileType('w'), default=sys.stdout,
        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 mongolimport utility)')
    parser.add_argument(
        '-t', '--type', type=int, metavar='ISIS_JSON_TYPE', default=1,
        help='ISIS-JSON type, sets field structure: 1=string, 2=list, 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 .mat (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', '--uid', 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 '99p')')
    parser.add_argument(
        '-n', '--mfn', action='store_true',
        help='generate an "id" from the MFN of each record'
        (available only for .mat 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('.mat'):
        iterRecords = iterMatRecords
    else:
        if args.mfn:
            print('UNSUPPORTED: -n/--mfn option only available for .mat 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.uid, args.mongo, args.mfn, args.type, args.prefix,
                        args.constant)
    if args.couch:
        args.out.write(']\n')
    args.out.close()

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


ASSUNTOS QUE EVITAMOS...

Envio de dados para um gerador através do método **.send()** (em vez de **next()**).

Uso de yield como uma expressão para obter o dado enviado por **.send()**.

Uso de funções geradoras como co-rotinas.

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

**“Co-rotinas não têm relação com iteração”
David Beazley**

¿PREGUNTAS?

ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com

ThoughtWorks®