



laziness as a virtue

GENERATOR POWER

True iterators for efficient data processing in Python



LEARNINGWebinar

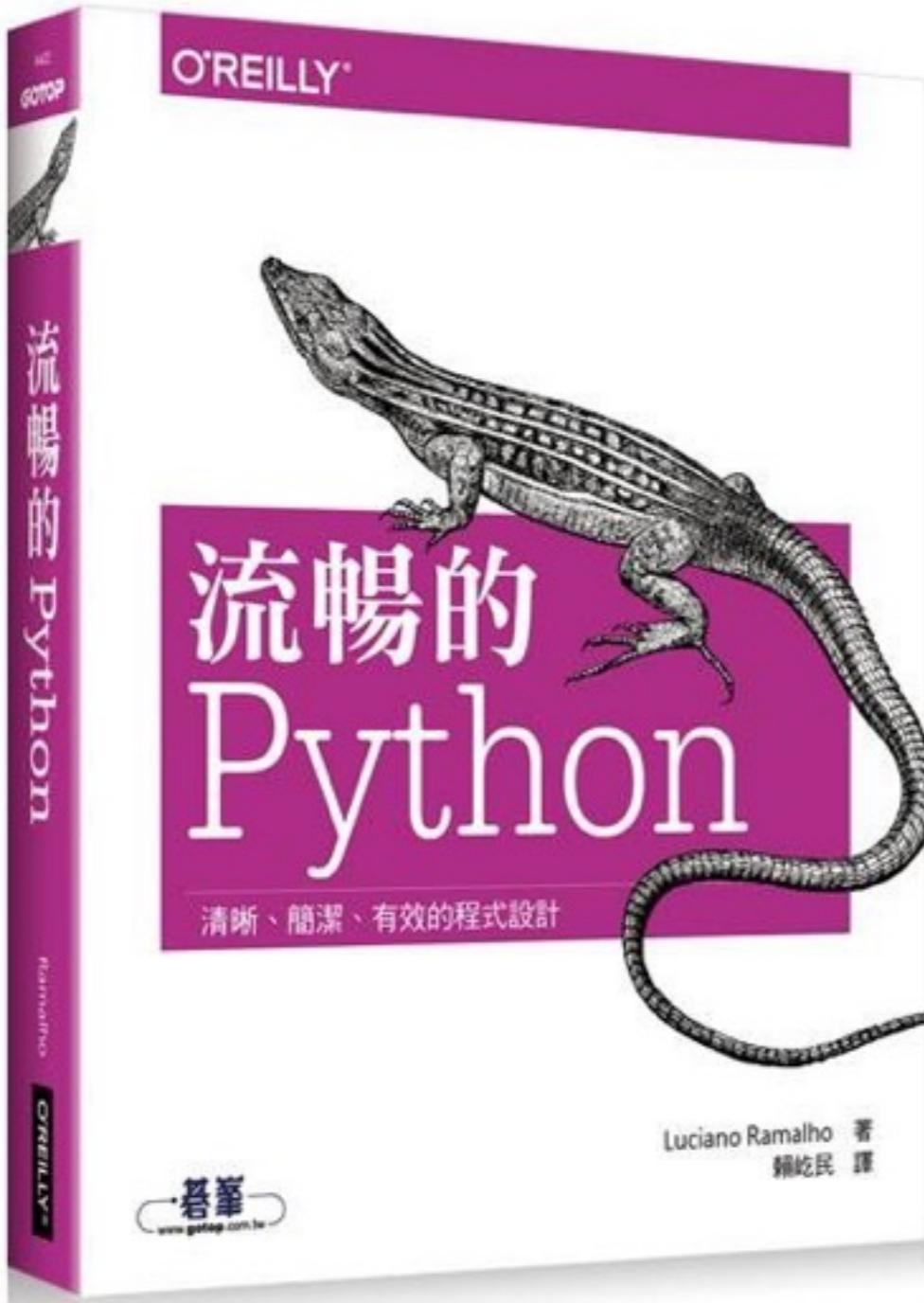
ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com

FLUENT PYTHON, MY FIRST BOOK



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*

ThoughtWorks®

ITERATION

That's what computers are for

ITERATION: C LANGUAGE

```
#include <stdio.h>

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

```
$ ./args alpha bravo charlie
./args
alpha
bravo
charlie
```

ITERATION: C VERSUS PYTHON

```
#include <stdio.h>

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

```
import sys
```

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

ITERATION: X86 INSTRUCTION SET

Loop Instructions [edit]

The `loop` instruction decrements ECX and jumps to the address specified by `arg` unless decrementing ECX caused its value to become zero. For example:

```
mov ecx, 5
start_loop:
; the code here would be executed 5 times
loop start_loop
```

source: x86 Assembly wikibook
https://en.wikibooks.org/wiki/X86_Assembly

ITERATION: BEFORE JAVA 5

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

```
$ java Arguments alpha bravo charlie  
alpha  
bravo  
charlie
```

FOREACH: SINCE JAVA 5

The official name of the *foreach* syntax is "enhanced for"

```
class Arguments2 {  
    public static void main(String[] args) {  
        for (String arg : args)  
            System.out.println(arg);  
    }  
}
```

```
$ java Arguments2 alpha bravo charlie  
alpha  
bravo  
charlie
```

FOREACH: SINCE JAVA 5

The official name of the *foreach* syntax is "enhanced for"

```
class Arguments2 {  
    public static void main(String[] args) {  
        for (String arg : args)  
            System.out.println(arg);  
    }  
}
```

year:
2004

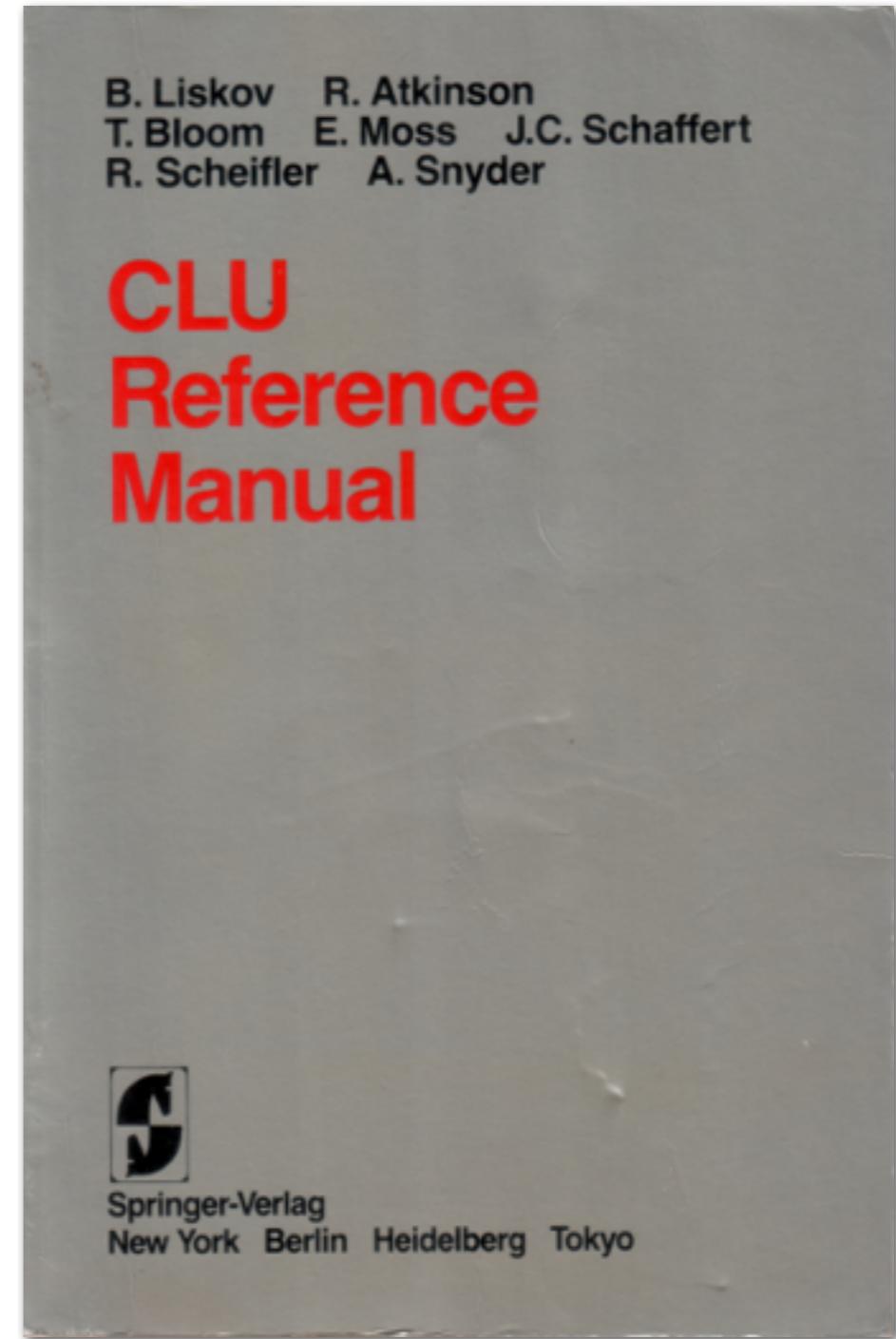
```
import sys
```

year:
1991

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

FOREACH IN BARBARA LISKOV'S CLU

© 2010 Kenneth C. Zirkel — CC-BY-SA



CLU Reference Manual — B. Liskov et. al. — © 1981 Springer-Verlag — also available online from MIT:
<http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-225.pdf>

2

Iterators

§1.2

types of results can be returned in the exceptional conditions. All information about the names of conditions, and the number and types of arguments and results is described in the *iterator heading*. For example,

```
leaves = iter (t: tree) yields (node)
```

is the heading for an iterator that produces all leaf nodes of a tree object. This iterator might be used in a **for** statement as follows:

```
for leaf: node in leaves(x) do
    ... examine(leaf) ...
end
```

1.3 Clusters

A *cluster* implements a data abstraction, which is a *operations* to create and manipulate those objects. The or control abstractions. The *cluster heading* states what *operations* are available, e.g.,

CLU Reference Manual, p. 2
B. Liskov et. al. — © 1981 Springer-Verlag

ITERABLE OBJECTS: THE KEY TO FOREACH

- Python, Java & CLU let programmers define **iterable** objects

```
for item in an_iterable:  
    process(item)
```

- Some languages don't offer this flexibility
 - C has no concept of iterables
 - In Go, only some built-in types are iterable and can be used with foreach (written as the **for ... range** special form)

ITERABLES

For loop data sources

avoidable
belieavable
extensible
fixable
iterable
movable
readable
playable
washable

iterable, adj. – Capable of being iterated.

SOME ITERABLE OBJECTS & THE ITEMS THEY YIELD

str: Unicode characters

bytes: integers 0...255

tuple: individual fields

dict: keys

set: elements

io.TextIOWrapper:

(text file) Unicode lines

models.query.QuerySet

(Django ORM) DB rows

numpy.ndarray

(NumPy multidimensional array) elements, rows...

```
>>> with open('1.txt') as text:  
...     for line in text:  
...         print(line.rstrip())  
  
alpha  
beta  
gamma  
delta
```

```
>>> d = {'α': 3, 'β': 4, 'γ': 5}  
>>> list(d)  
['γ', 'β', 'α']  
>>> list(d.values())  
[5, 4, 3]  
>>> list(d.items())  
[('γ', 5), ('β', 4), ('α', 3)]
```

OPERATIONS WITH ITERABLES

- Parallel assignment (a.k.a. tuple unpacking)

```
>>> a, b, c = 'XYZ'  
>>> a  
'X'  
>>> b  
'Y'  
>>> c  
'Z'  
>>> g = (n*10 for n in [1, 2, 3])  
>>> a, b, c = g  
>>> a  
10  
>>> b  
20  
>>> c  
30
```

ITERATING OVER ITERABLES OF ITERABLES

- Parallel assignment in **for** loops

```
>>> pairs = [('A', 10), ('B', 20), ('C', 30)]  
  
>>> for label, size in pairs:  
...     print(label, '->', size)  
...  
A -> 10  
B -> 20  
C -> 30
```

ONE ITERABLE PROVIDING MULTIPLE ARGUMENTS

- Function argument unpacking (a.k.a. *splat*)

```
>>> def area(a, b, c):
...     """Heron's formula"""
...     a, b, c = sorted([a, b, c], reverse=True)
...     return ((a+(b+c)) * (c-(a-b)) * (c+(a-b)) * (a+(b-c))) ** .5 / 4
...
>>> area(3, 4, 5)
6.0
>>> t = (3, 4, 5)
>>> area(*t)
6.0
```

expand iterable
into multiple
arguments



REDUCTION FUNCTIONS

Reduction functions: consume a finite iterable and return a scalar value (e.g. the sum, the largest value etc.)

- **all**
- **any**
- **max**
- **min**
- **sum**

```
>>> L = [5, 7, 8, 1, 4, 6, 2, 9, 0, 3]
>>> all(L)
False
>>> any(L)
True
>>> max(L)
9
>>> min(L)
0
>>> sum(L)
45
```

SORT X SORTED

- **.sort():** a **list** method, sorts the list in-place (e.g. `my_list.sort()`)
- **sorted():** a built-in function, consumes an **iterable** and returns a new sorted **list**

```
>>> L = ['grape', 'Cherry', 'strawberry', 'date', 'banana']
>>> sorted(L)
['Cherry', 'banana', 'date', 'grape', 'strawberry']

>>> sorted(L, key=str.lower) # case insensitive
['banana', 'Cherry', 'date', 'grape', 'strawberry']

>>> sorted(L, key=len) # sort by word length
['date', 'grape', 'Cherry', 'banana', 'strawberry']

>>> sorted(L, key=lambda s:list(reversed(s))) # reverse word
['banana', 'grape', 'date', 'strawberry', 'Cherry']
```

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

Django ORM queryset demo

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

this proves
that queryset
is a lazy iterable

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

the database is
hit only when
the for loop
consumes the
results

THE ITERATOR PATTERN

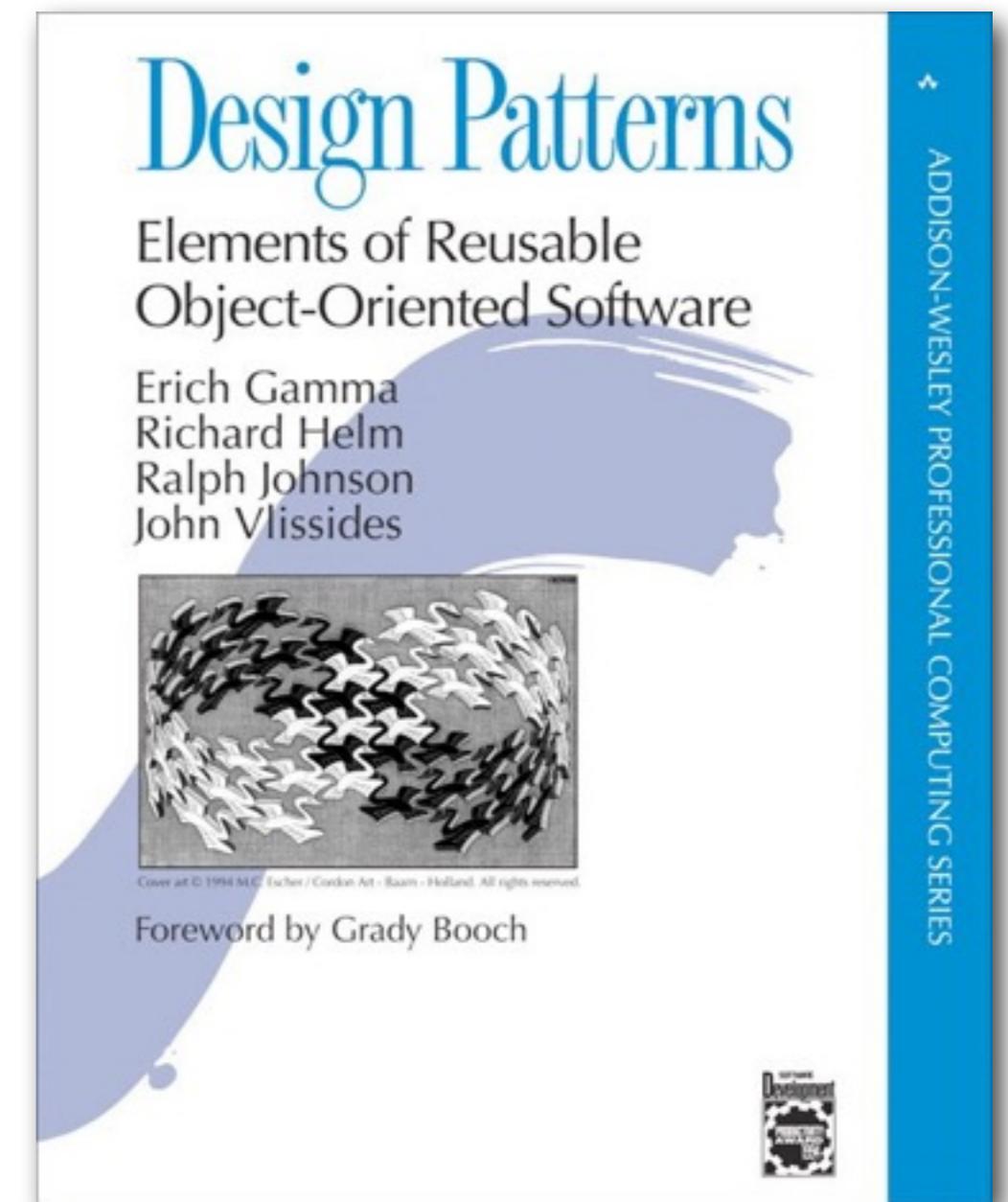
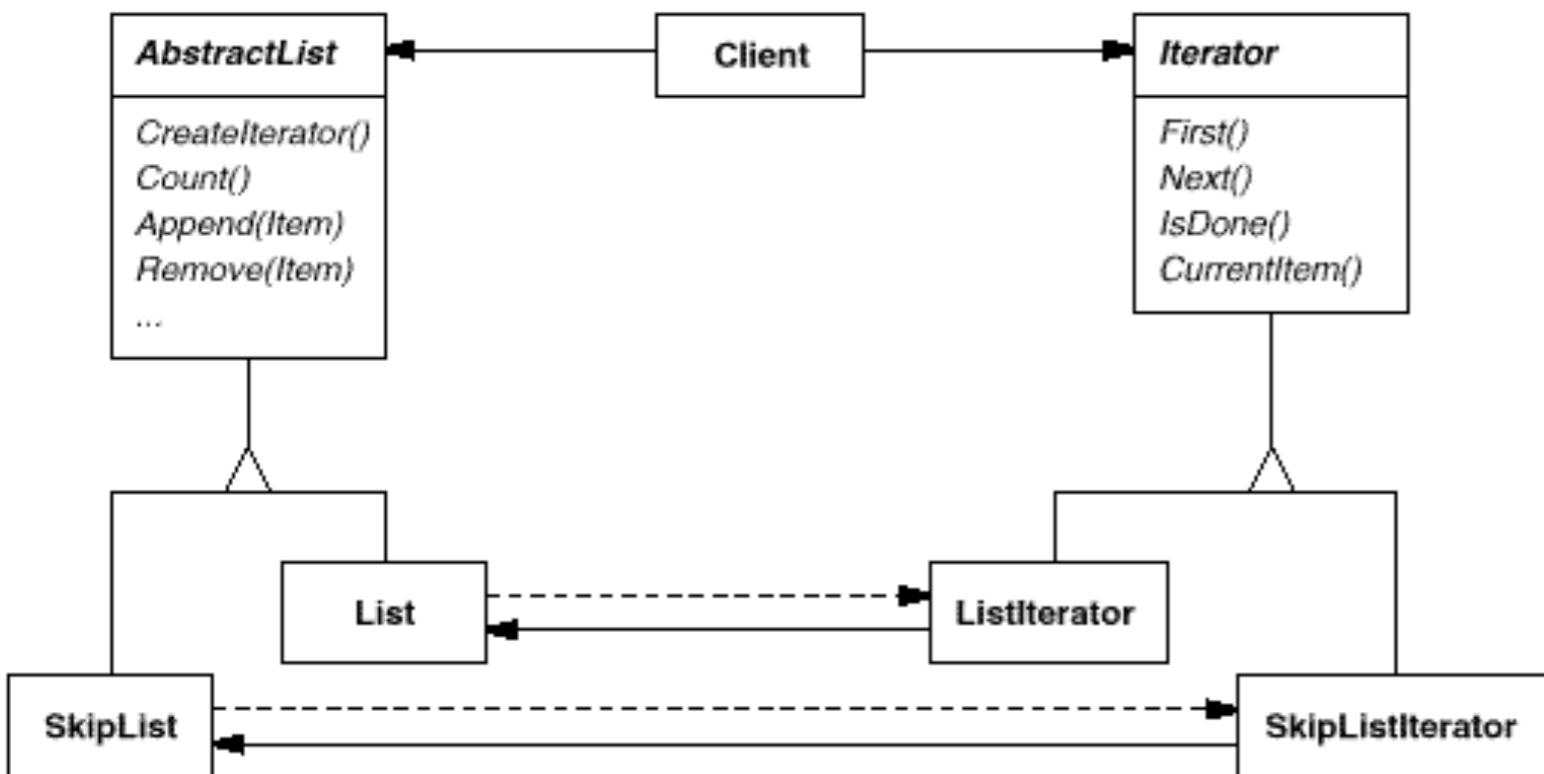
The classic recipe

THE ITERATOR FROM THE GANG OF FOUR

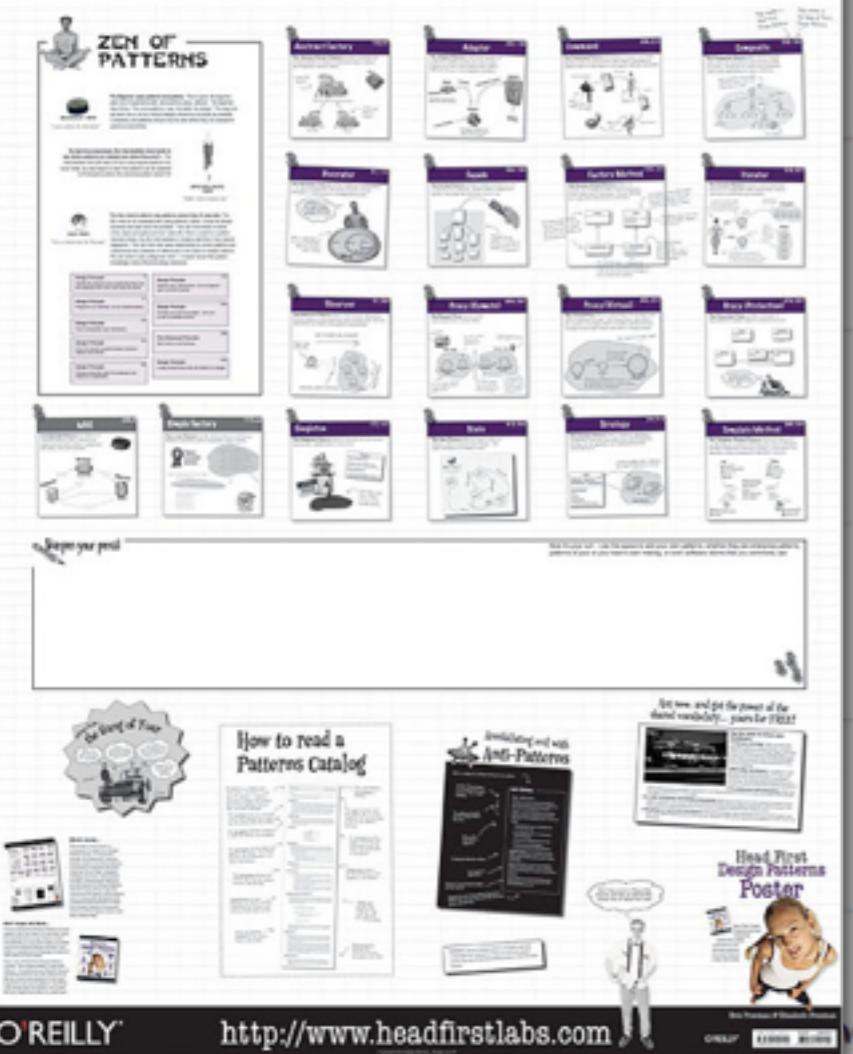
Design Patterns

Gamma, Helm, Johnson & Vlissides

©1994 Addison-Wesley



Your Brain on Design Patterns



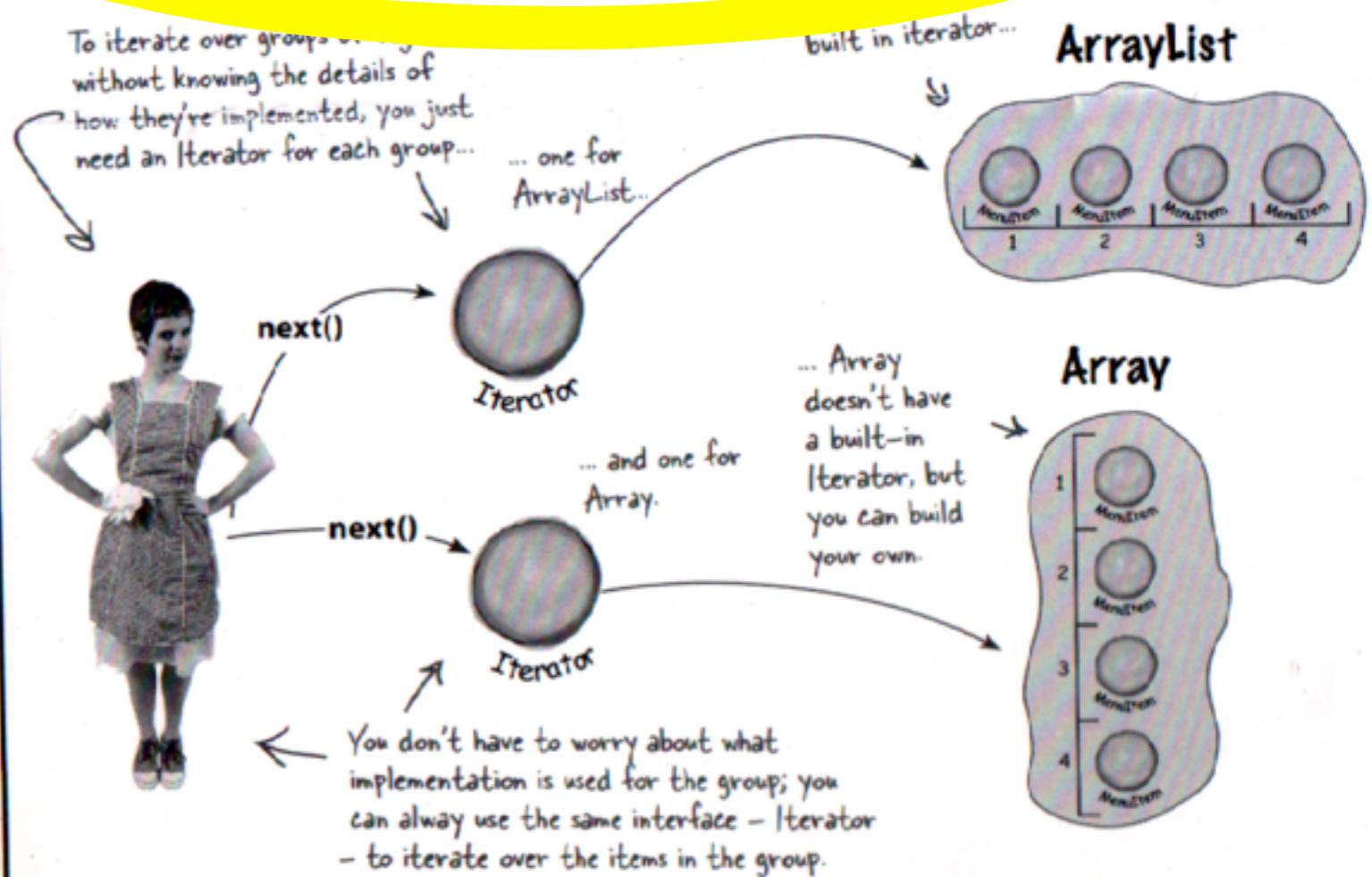
Head First Design Patterns Poster

O'Reilly

ISBN 0-596-10214-3

Iterator

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



THE FOR LOOP MACHINERY

- In Python, the **for** loop, automatically:
 - Obtains an **iterator** from the **iterable**
 - Repeatedly invokes **next()** on the **iterator**, retrieving one item at a time
 - Assigns the item to the loop variable(s)



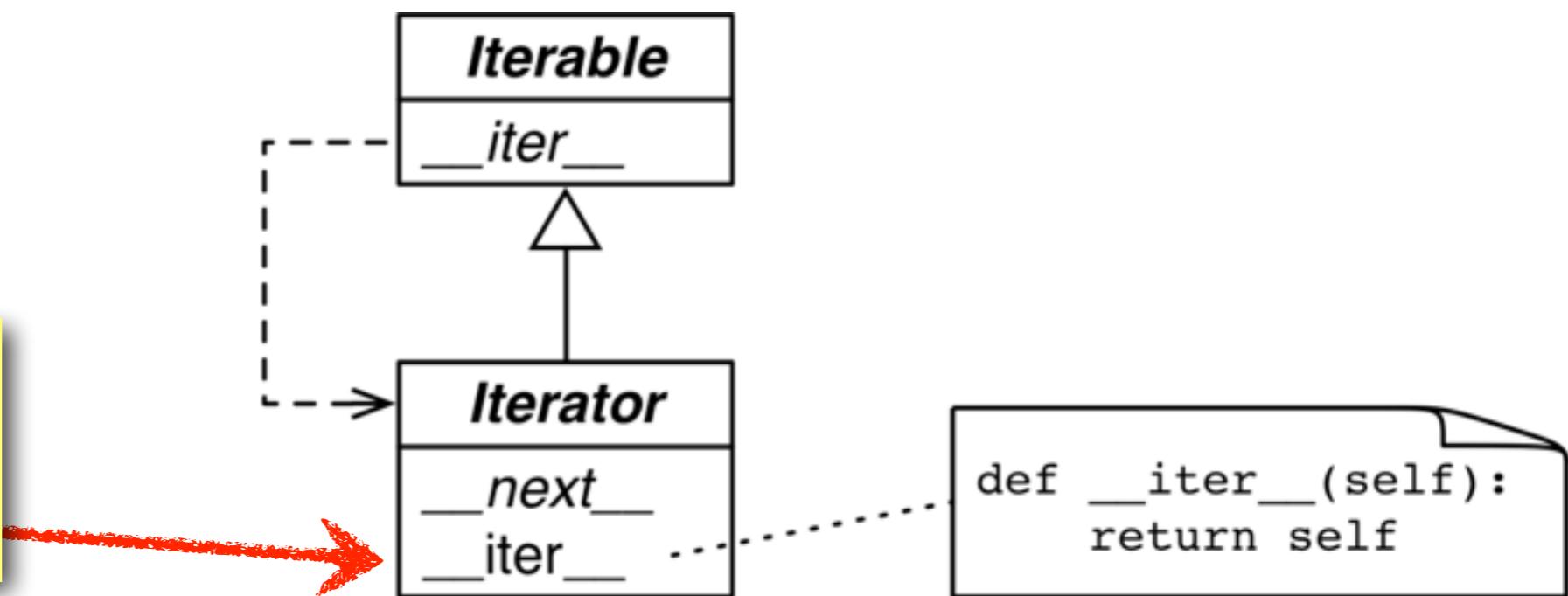
```
for item in an_iterable:  
    process(item)
```

- Terminates when a call to **next()** raises **StopIteration**.

ITERABLE VERSUS ITERATOR

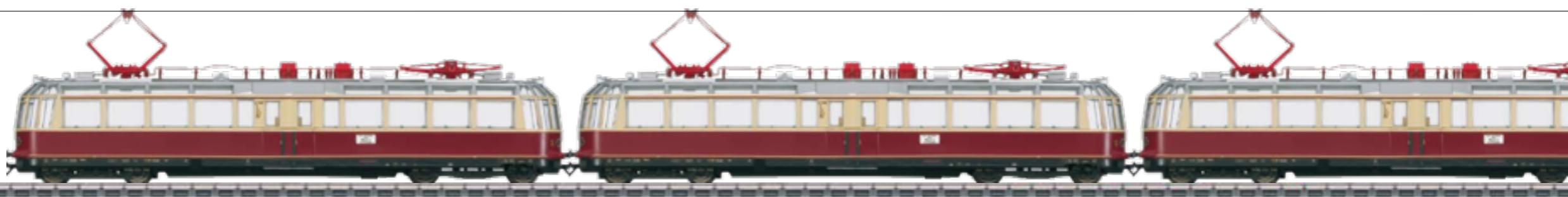
- **iterable**: implements **Iterable** interface (`__iter__` method)
 - `__iter__` method returns an **Iterator**
- **iterator**: implements **Iterator** interface (`__next__` method)
 - `__next__` method returns next item in series and
 - raises **StopIteration** to signal end of the series

Python iterators
are also iterable!



AN ITERABLE TRAIN

An instance of **Train** can be iterated, car by car



```
>>> t = Train(3)
>>> for car in t:
...     print(car)
car #1
car #2
car #3
>>>
```

CLASSIC ITERATOR IMPLEMENTATION

The pattern as described by Gamma et. al.

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

```
class Train:

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

    def __iter__(self):
        return IteratorTrem(self.cars)

class TrainIterator:

    def __init__(self, cars):
        self.next = 0
        self.last = cars - 1

    def __next__(self):
        if self.next <= self.last:
            self.next += 1
            return 'car #%s' % (self.next)
        else:
            raise StopIteration()
```

GENERATOR FUNCTION

Michael Scott's "true iterators"

A VERY SIMPLE GENERATOR FUNCTION

Any function that has the **yield** keyword in its body is a generator function.

When invoked,
generator function
returns a
generator object

Note:

The **gen** keyword was proposed to replace **def** in generator function headers, but Guido van Rossum rejected it.

```
>>> 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('starting...')
...     yield 'A'
...     print('continuing...')
...     yield 'B'
...     print('The End.')
...
>>> for s in gen_ab(): print(s)
starting...
A
continuing...
B
The End.
>>> g = gen_ab()
>>> g
<generator object gen_ab at 0x...>
>>> next(g)
starting...
'A'
>>> next(g)
continuing...
'B'
>>> next(g)
The End.
Traceback (most recent call last):
...
StopIteration
```

HOW IT WORKS

- Invoking the generator function builds a **generator object**
- The body of the function only starts when **next(g)** is called.
- At each **next(g)** call, the function resumes, runs to the next **yield**, and is suspended again.

THE WORLD FAMOUS FIBONACCI GENERATOR

fibonacci yields an infinite series of integers.

```
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
```

FIBONACCI GENERATOR BOUND TO N ITEMS

Easier to use

```
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]
```

ARITHMETIC PROGRESSION GENERATOR

```
def arithmetic_progression(increment, *, start=0, end=None):
    index = 0
    result = start + increment * index
    while end is None or result < end:
        yield result
        index += 1
        result = start + increment * index
```

```
>>> ap = arithmetic_progression(.1)
>>> next(ap), next(ap), next(ap), next(ap), next(ap)
(0.0, 0.1, 0.2, 0.3000000000000004, 0.4)

>>> from decimal import Decimal
>>> apd = arithmetic_progression(Decimal('.1'))
>>> [next(apd) for i in range(4)]
[Decimal('0.0'), Decimal('0.1'), Decimal('0.2'), Decimal('0.3')]

>>> list(arithmetic_progression(.5, start=1, end=5))
[1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5]

>>> list(arithmetic_progression(1/3, end=1))
[0.0, 0.333333333333333, 0.666666666666666]
```

ITERABLE TRAIN WITH A GENERATOR METHOD

The **Iterator** pattern as a language feature:

```
class Train:

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

    def __iter__(self):
        for i in range(self.cars):
            yield 'car #{}'.format(i+1)
```

Train is now iterable
because `__iter__`
returns a generator!

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

COMPARE: CLASSIC ITERATOR × GENERATOR METHOD

The classic Iterator recipe is obsolete in Python since v.2.2 (2001)

```
class Train:

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

    def __iter__(self):
        return IteratorTrain(self.cars)

class TrainIterator:

    def __init__(self, cars):
        self.next = 0
        self.last = cars - 1

    def __next__(self):
        if self.next <= self.last:
            self.next += 1
            return 'car #{}'.format(self.next)
        else:
            raise StopIteration()
```

```
class Train:

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

    def __iter__(self):
        for i in range(self.cars):
            yield 'car #{}'.format(i+1)
```

Generator
function
handles the
state of the
iteration

BUILT-IN GENERATORS

Common in Python 2, widespread in Python 3

BUILT-IN GENERATOR FUNCTIONS

Consume any iterable object and return generator objects:

enumerate

filter

map

reversed

zip



ZIP, MAP & FILTER IN PYTHON 2

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

not generators!

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

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

zip:

consumes N **iterables** in parallel, yielding **list** of tuples

map:

applies function to each item in **iterable**, returns **list** with results

filter:

returns **list** with items from **iterable** for which predicate results *truthy*

ZIP, MAP & FILTER: PYTHON 2 × PYTHON 3

```
>>> 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 2

In Python 3, **zip**, **map**, **filter** and many other functions in the standard library return **generators**.

```
>>> 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>
```

Python 3

GENERATORS ARE ITERATORS

Generators are iterators, which are also iterable:

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

Build the list explicitly to get what Python 2 used to give:

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

Most collection constructors consume suitable generators:

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

THE ITERTOOLS STANDARD MODULE

- "infinite" generators
 - **count(), cycle(), repeat()**
- generators that consume multiple iterables
 - **chain(), tee(), izip(), imap(), product(), compress()...**
- generators that filter or bundle items
 - **compress(), dropwhile(), groupby(), ifilter(), islice()...**
- generators that rearrange items
 - **product(), permutations(), combinations()...**

Note:

Many of these functions were inspired by the Haskell language

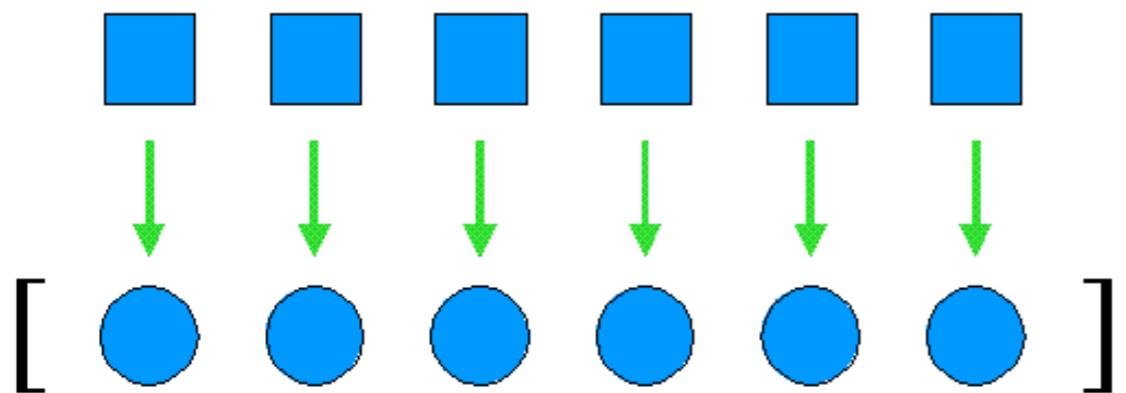
ThoughtWorks®

GENEXPS

Syntax shortcut for building generators

LIST COMPREHENSION

Syntax to build a list from any finite iterable — limited only by available memory.



input: any iterable

output: always a list

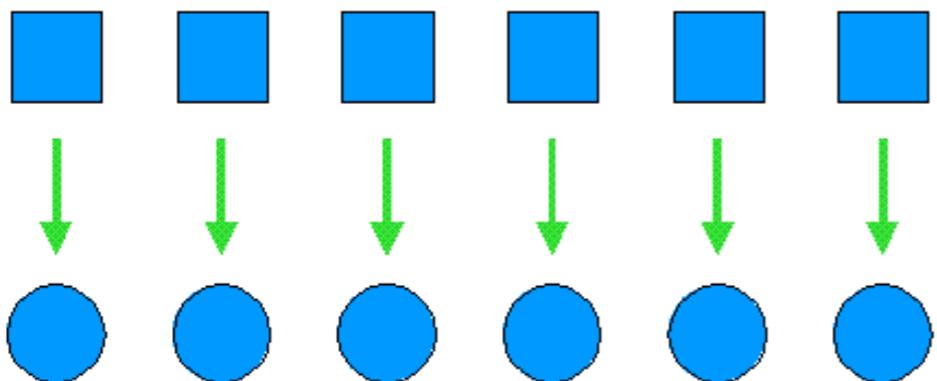
```
>>> s = 'abracadabra'  
>>> l = [ord(c) for c in s]  
>>> [ord(c) for c in s]  
[97, 98, 114, 97, 99, 97, 100, 97, 98, 114, 97]
```

*** syntax borrowed from Haskell and set builder notation**

GENERATOR EXPRESSION

Syntax to build an generator from any iterable.

Evaluated *lazily*: input is consumed one item at a time



input: any iterable

output: a generator

```
>>> 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]
```

TRAIN WITH GENERATOR EXPRESSION

`__iter__` as a plain method returning a generator expression:

```
class Train:

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

    def __iter__(self):
        return ('car #{}'.format(i+1) for i in range(self.cars))
```

`__iter__` as a generator method:

```
class Train:

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

    def __iter__(self):
        for i in range(self.cars):
            yield 'car #{}'.format(i+1)
```

THE PYTHONIC DEFINITION OF ITERABLE

iterable, adj. – (Python) An object from which the `iter()` function can build an **iterator**.

`iter(iterable)`

Returns iterator for iterable, invoking `__iter__` (if available) or building an iterator to fetch items via `__getitem__` with 0-based indices (`seq[0]`, `seq[1]`, etc...)

CASE STUDY

Using generator functions to convert database dumps

CONVERSION OF LARGE DATA SETS

Context

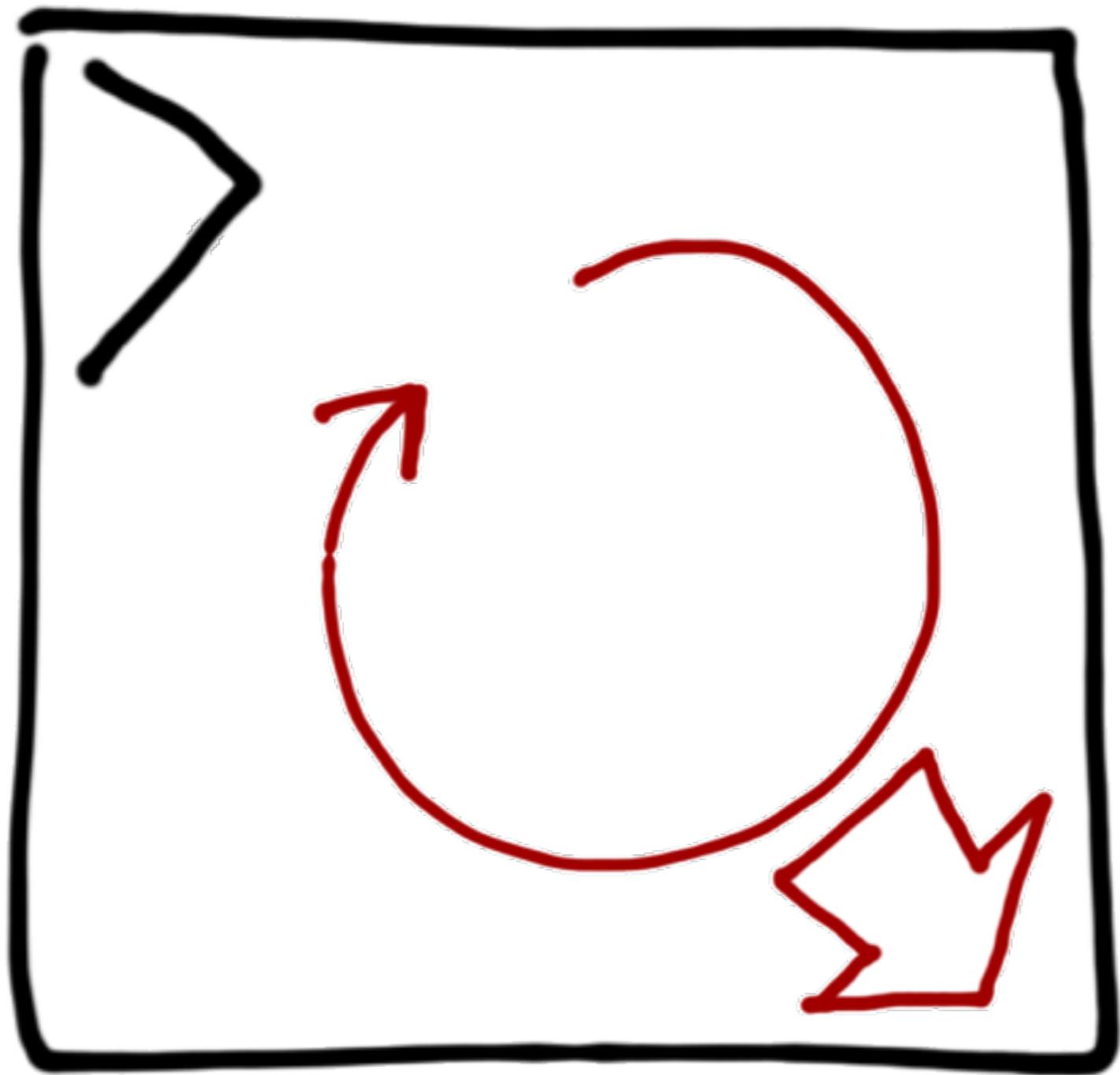
`isis2json`, a command-line tool to convert and refactor semi-structured database dumps; written in **Python 2.7**.

Usage

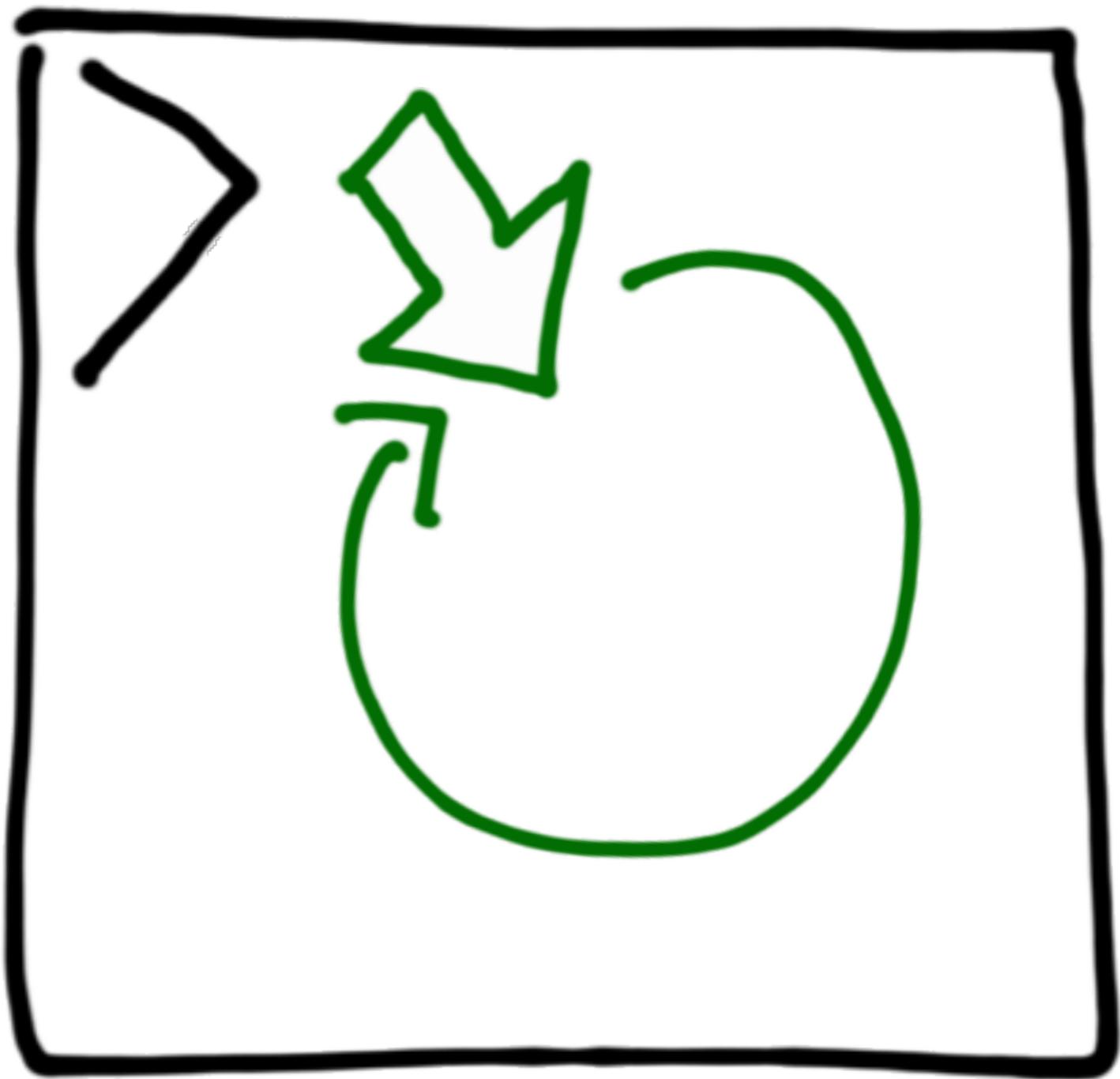
Generator functions to decouple reading from writing logic.

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

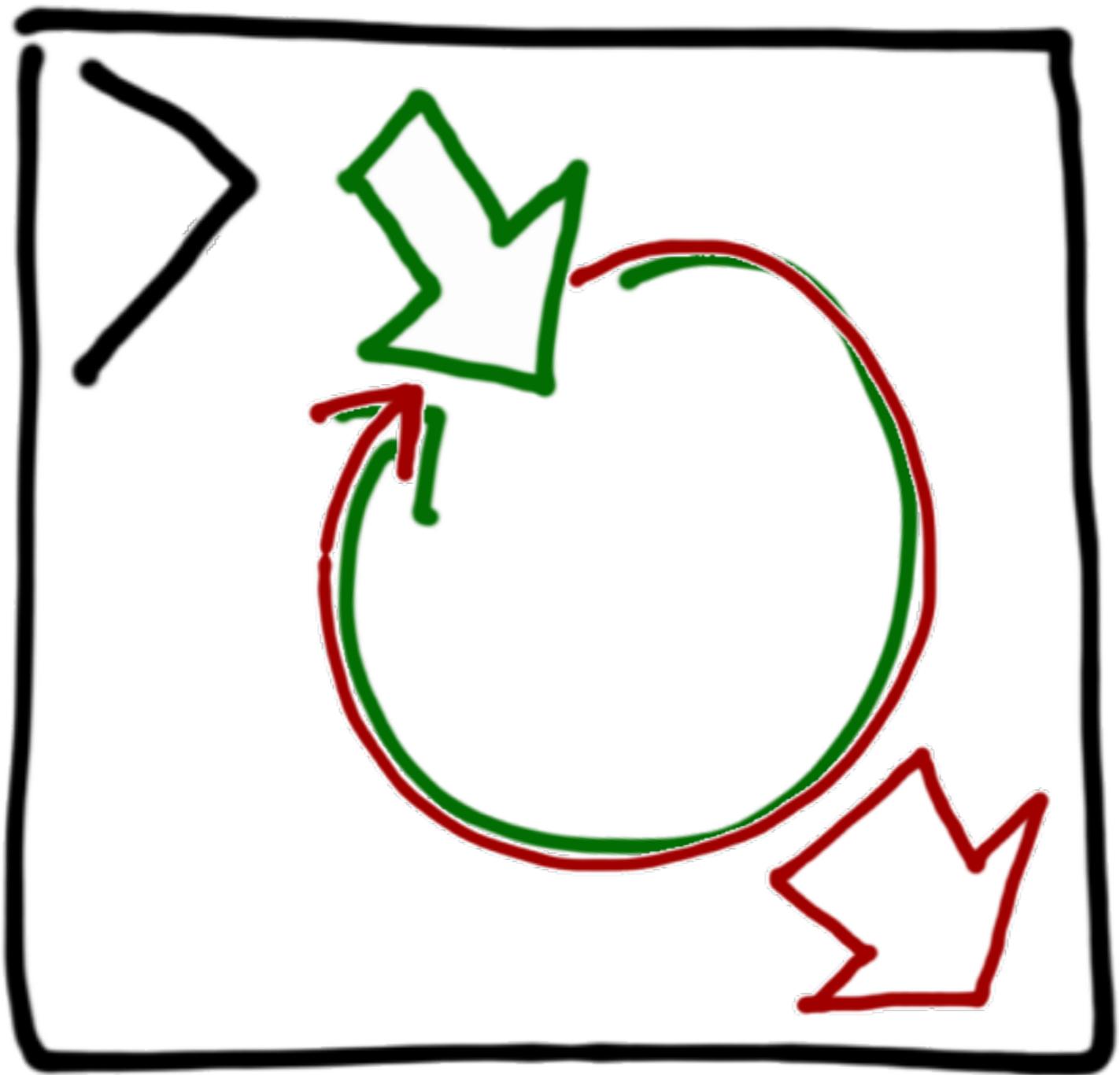
MAIN LOOP: OUTPUTS JSON FILE



ANOTHER LOOP READS RECORDS TO CONVERT

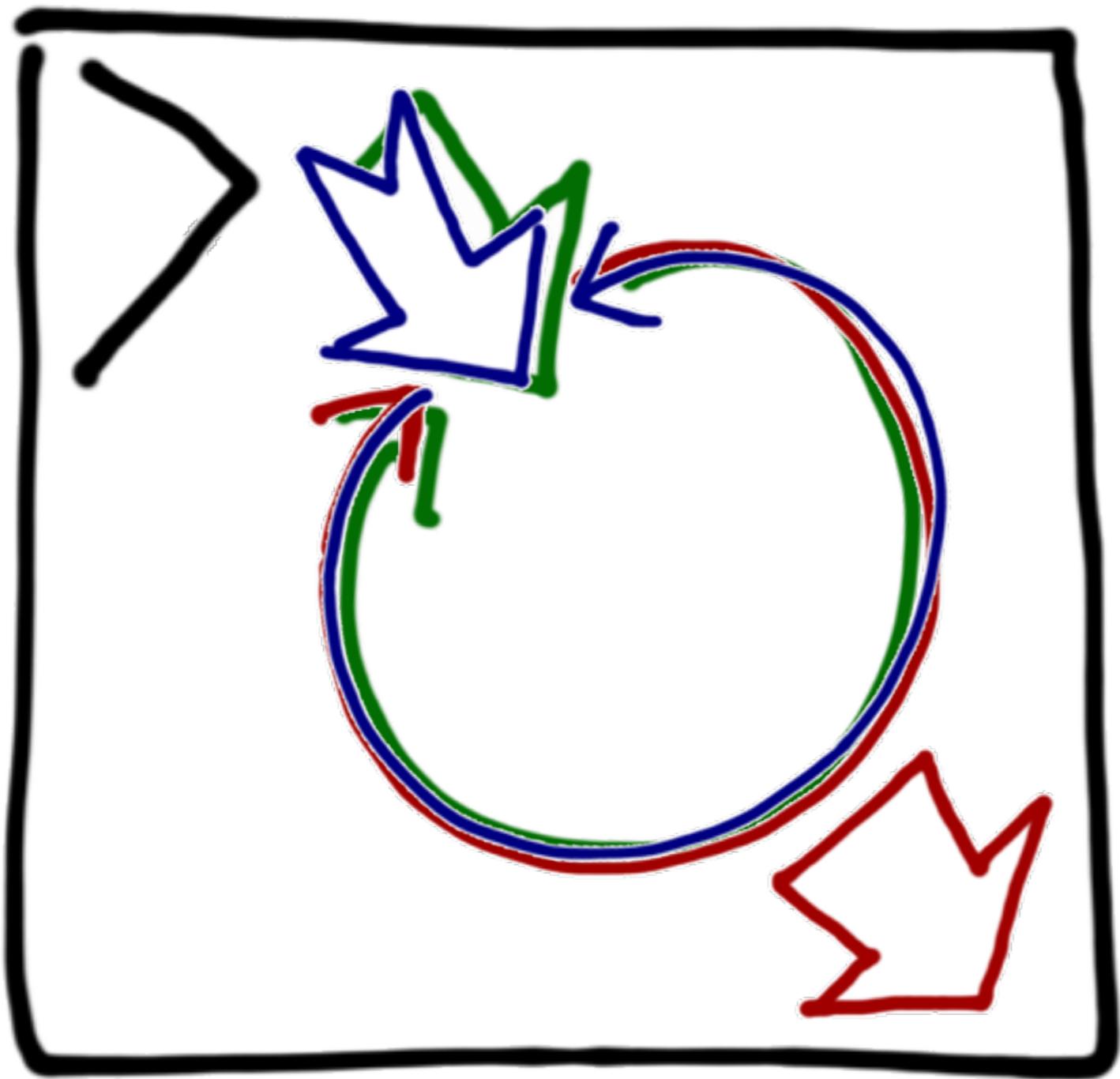


ONE SOLUTION: SAME LOOP READS AND WRITES



```
import java  
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class Solution {  
    public int[] findRedundantConnection(int[][] edges) {  
        int n = edges.length; // number of nodes  
        int m = edges[0].length; // number of edges  
        int[] parent = new int[n];  
        int[] rank = new int[n];  
        for (int i = 0; i < n; i++) {  
            parent[i] = i;  
            rank[i] = 1;  
        }  
  
        for (int i = 0; i < m; i++) {  
            int u = edges[i][0];  
            int v = edges[i][1];  
            if (findParent(u) == findParent(v)) {  
                return edges[i];  
            } else {  
                union(u, v);  
            }  
        }  
        return null;  
    }  
  
    private int findParent(int u) {  
        if (parent[u] == u) {  
            return u;  
        } else {  
            return findParent(parent[u]);  
        }  
    }  
  
    private void union(int u, int v) {  
        int pu = findParent(u);  
        int pv = findParent(v);  
        if (pu == pv) {  
            return;  
        } else if (rank[pu] > rank[pv]) {  
            parent[pv] = pu;  
        } else if (rank[pu] < rank[pv]) {  
            parent[pu] = pv;  
        } else {  
            parent[pu] = pv;  
            rank[pv]++;  
        }  
    }  
}
```

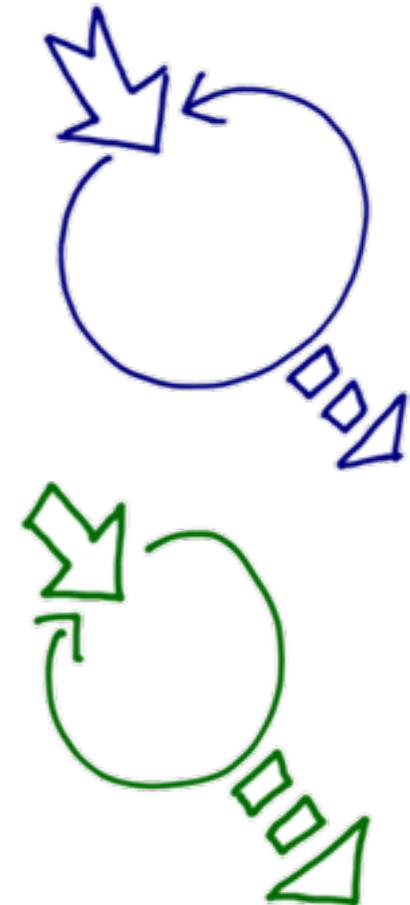
HOW TO SUPPORT A NEW INPUT FORMAT?



SOLUTION: GENERATOR FUNCTIONS

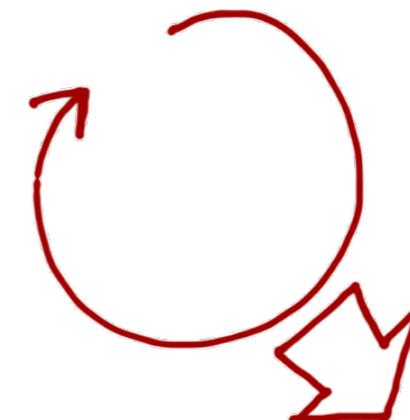
iterMstRecords

generator function: yields MST records



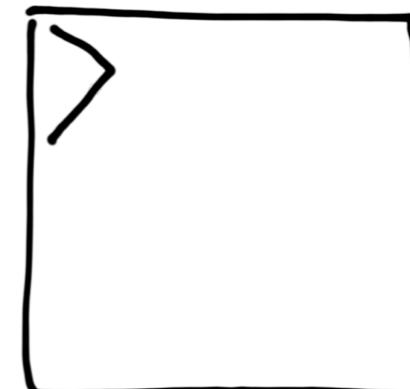
iterIsoRecords

generator function: yields ISO-2709 records



writeJsonArray

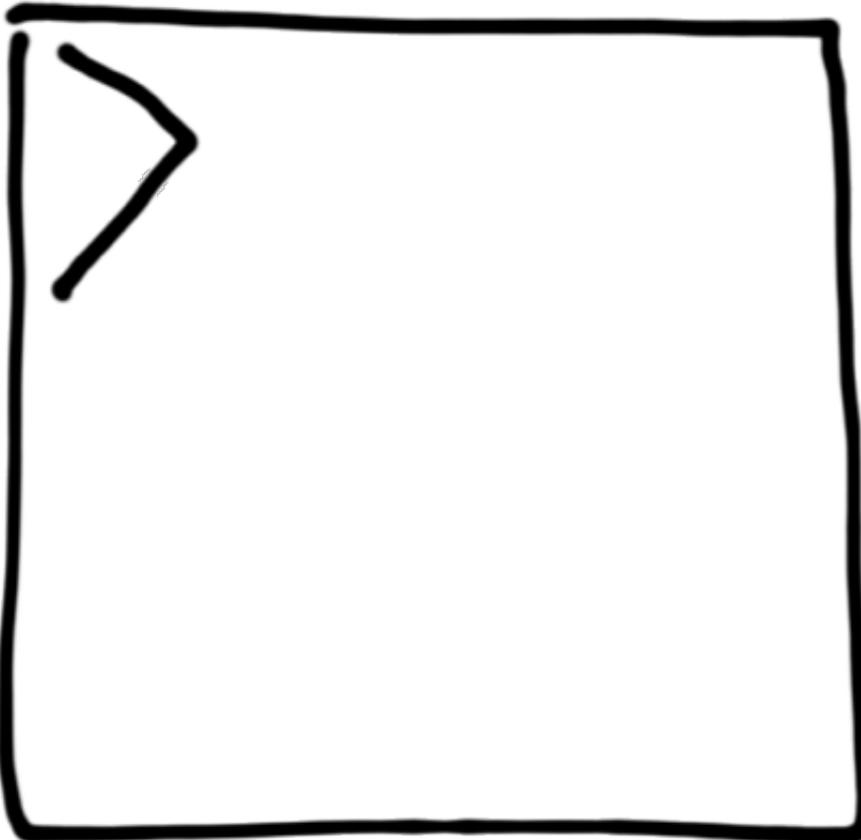
consumes and outputs records



main

parses command-line arguments

MAIN: PARSE COMMAND-LINE



```
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', '--mfns', 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.mfns:
            print('UNSUPPORTED: -n/--mfns 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.mfns, args.type, args.prefix,
                      args.constant)
    if args.couch:
        args.out.write('}\n')
    args.out.close()

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

MAIN: SELECTING INPUT GENERATOR FUNCTION

pick generator function
depending on input file
extension

```
args = parser.parse_args()
if args.file_name.lower().endswith('.mst'):
    iterRecords = iterMstRecords
else:
    if args.mfn:
        print('UNSUPPORTED: -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()
```

chosen generator function
passed as argument

WRITING JSON RECORDS

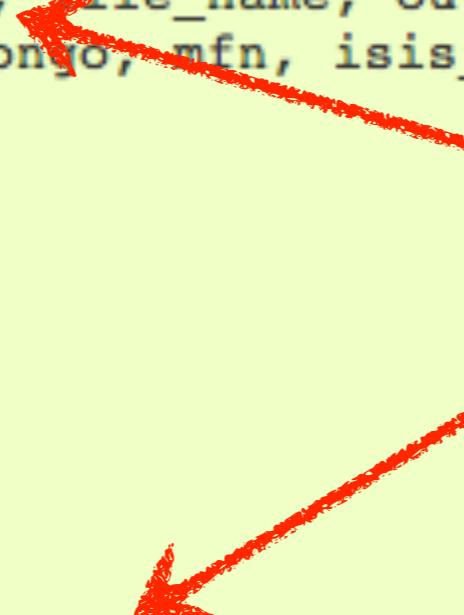


```
def writeJsonArray(iterRecords, file_name, output, qty, skip, id_tag,
                   gen_uuid, mongo, mfn, isis_json_type, prefix, constant):
    start = skip
    end = start + qty
    if not mongo:
        output.write('[')
    if id_tag:
        id_tag = str(id_tag)
        ids = set()
    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')
```

WRITING JSON RECORDS

writeJsonArray gets generator function as first argument, then uses a **for** loop to consume that generator.

```
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)
```



READING ISO-2709 RECORDS

Input **for** loop reads each ISO-2709 record, populates a dict with its fields, and yields the dict.

```
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()
```



generator
function!

READING ISO-2709 RECORDS

```
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()
```

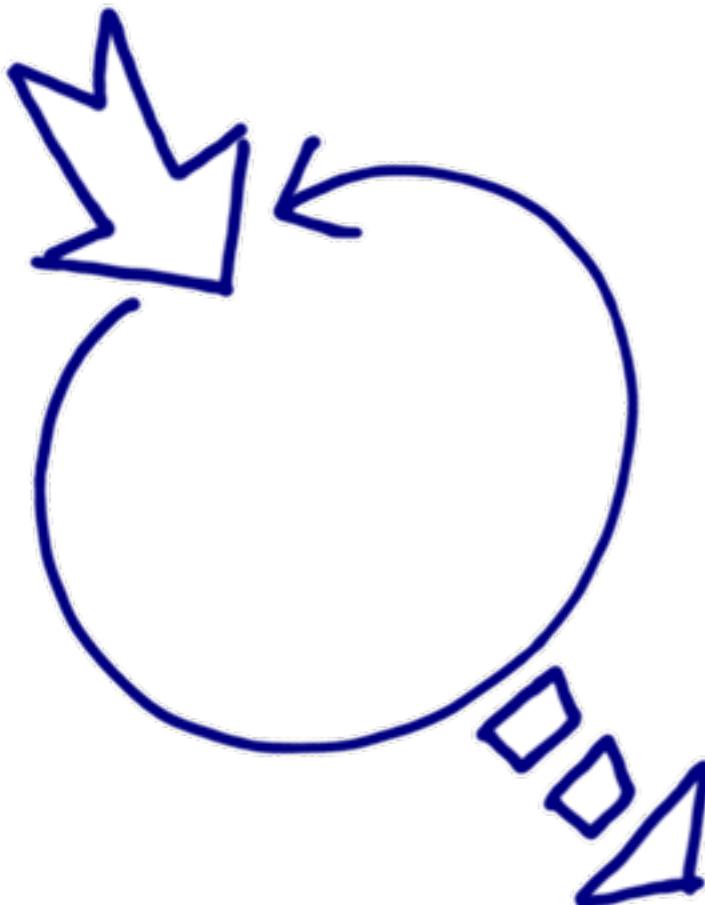
creates new dict
at each iteration

yields dict populated
with record fields

READING .MST RECORDS

Input **for** loop reads each .MST record, populates a dict with its fields, and yields the dict.

another generator function!



```
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()
```

```

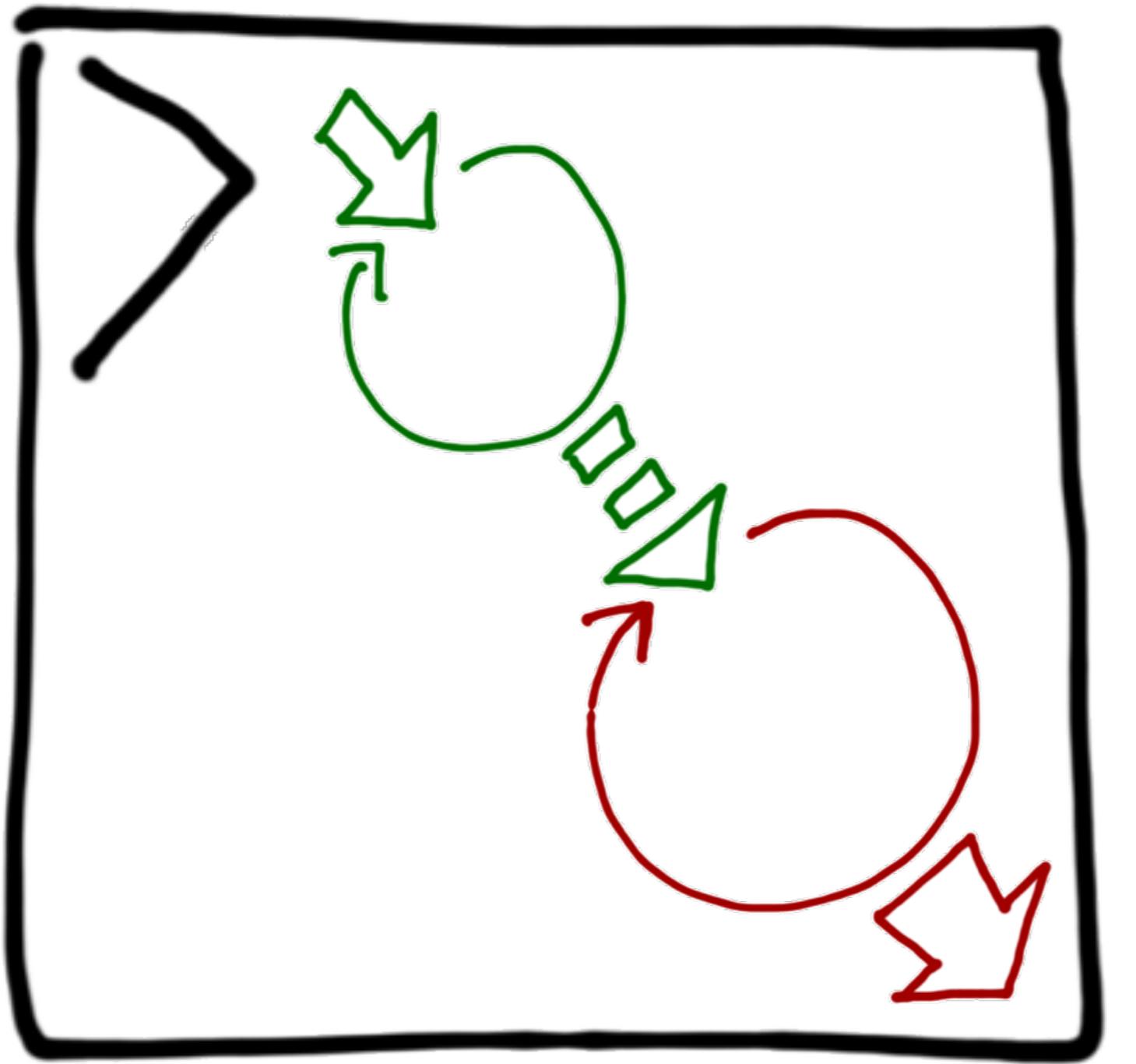
def iterMstRecords(master_file_name, isis_json_type):
    try:
        from bruma.master import MasterFactory, Record
    except ImportError:
        print('IMPORT ERROR: Jython 2.5 and Bruma.jar are required '
              'to read .mst files')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()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

```

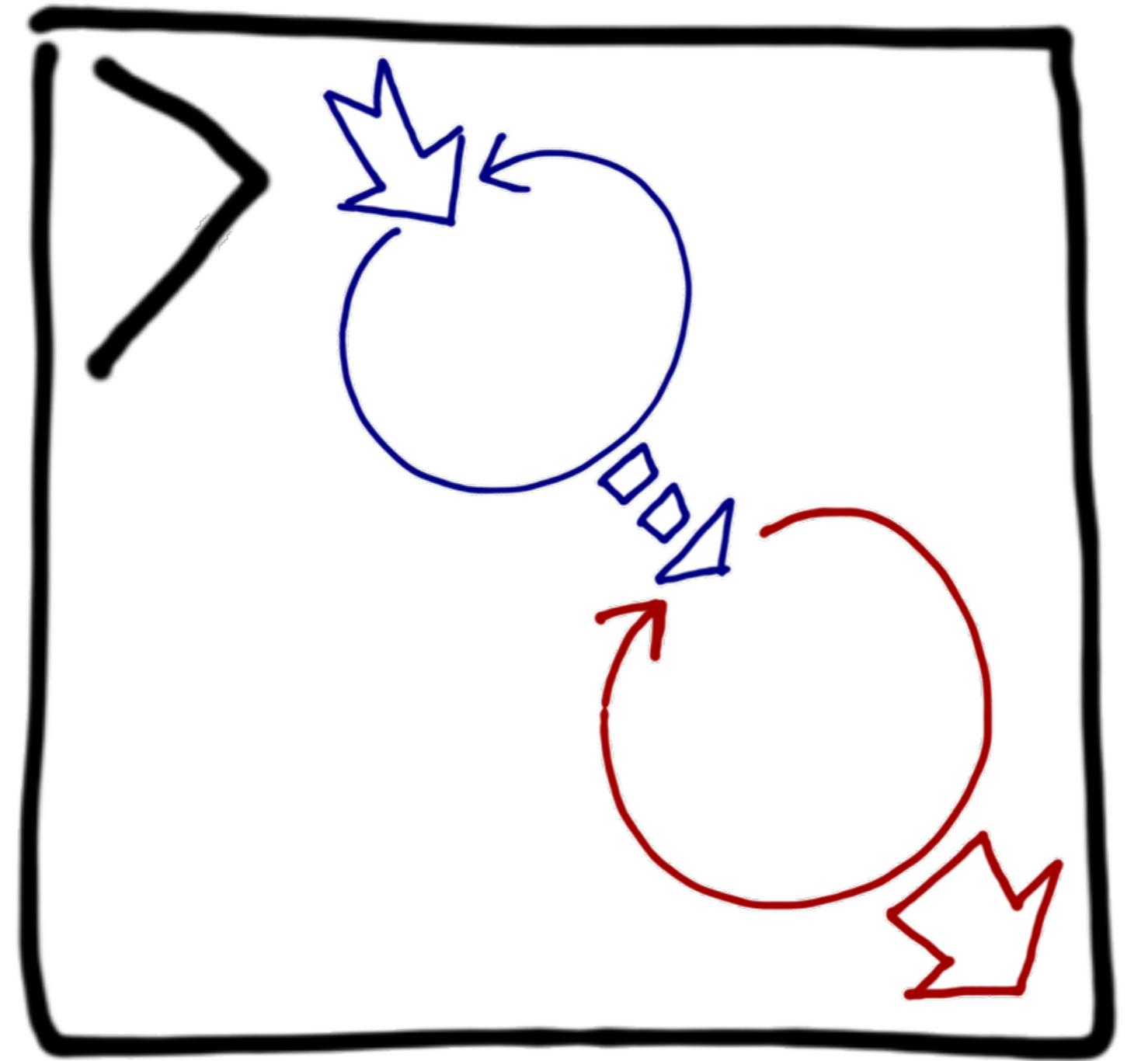
creates new dict
at each iteration

yields dict populated
with record fields

PROBLEM SOLVED



PROBLEM SOLVED



```
import sys
import argparse
from uuid import uuid
import os

try:
    import json
except ImportError:
    if os.name == 'java': # assuming Python
        from com.thoughtworks.json import JsonCodec as json
    else:
        import simplejson as json

#SET DEBUG=1 To
DEBUG_QTY = 2
ISIB_MFN_KEY = 'mfn'
ISIB_TYPE = 'active'
SUBFIELD_DELIMITER = ''
INPUT_ENCODING = 'cp1252'

def iterIsibRecords(master_file_name, isim_json_type):
    try:
        from brume.master import MasterFactory, Record
    except ImportError:
        print('MasterFactory, Record, Python 2.5 and Brume.jar are required')
        raise
    rnum = SystemExit
    master_factory = getInstance(master_file_name).open()
    for record in master_factory:
        if SKIP_INACTIVE:
            if record.getStatus() != Record.Status.ACTIVE:
                continue
        else:
            if record.getStatus() != Record.Status.ACTIVE:
                yield record
        field_id = record.getFieldId()
        field_occurrences = field_id.getOccurrences()
        if isim_json_type == 1:
            content = record.getContent()
            for subfield in field_id.getSubfields():
                if subfield.getKey() == 31:
                    content.insert(0, subfield.getContent())
                else:
                    subfield_occurrences = content.setdefault(subfield.getKey(),[])
                    subfield_occurrences.append(subfield.getContent())
                    field_occurrences.append(content)
        elif isim_json_type == 11:
            content = record.getContent()
            for subfield in field_id.getSubfields():
                if subfield.getKey() == 31:
                    content.insert(0, subfield.getContent())
                else:
                    subfield_occurrences = content.setdefault(subfield.getKey(),[])
                    subfield_occurrences.append(subfield.getContent())
                    content.append(SUBFIELD_DELIMITER+subfield.getKey+content)
                    field_occurrences.append(content)
        else:
            raise NotImplementedError('not yet')
    master_factory.close()

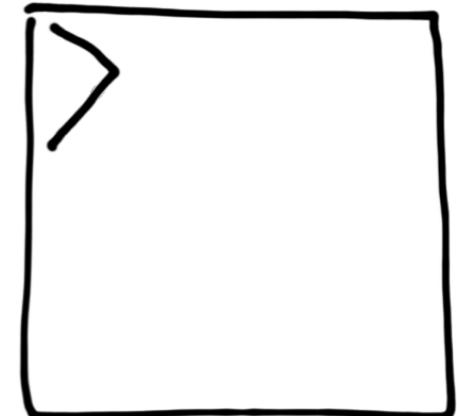
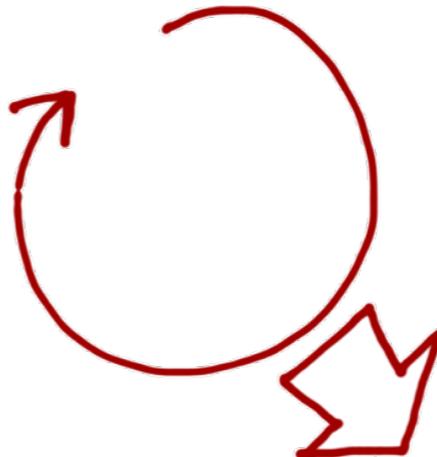
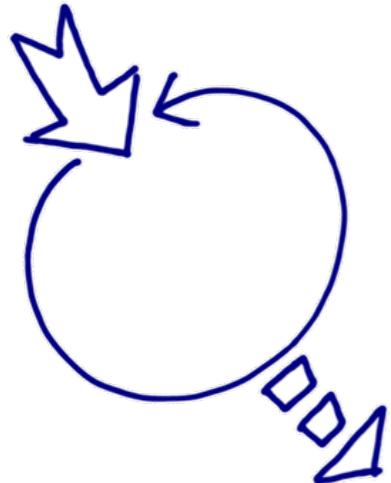
def iterIsoRecords(iso_file_name, isim_json_type):
    try:
        from iso2709 import IsoFile
        from iso2709 import expand
    except ImportError:
        print('IsoFile, expand not implemented')
        raise
    iso = IsoFile(iso_file_name)
    for record in iso:
        fields = []
        for field in record.directory:
            field_key = str(int(field.tag)) # remove leading zeros
            field_occurrences = fields.setdefault(field.key,[])
            content = field.getValue()
            if isim_json_type == 1:
                field_occurrences.append(content)
            elif isim_json_type == 21:
                field_occurrences.append(expand(content))
            elif isim_json_type == 22:
                field_occurrences.append(dict(expand(content)))
            else:
                raise NotImplementedError('not yet')
        yield fields
    iso.close()

def writeIsoArray(iterRecords, file_name, output, qty, skip, id_tag,
                 gen_uuid, mongo, mfn, isim_json_type, prefix, constant):
    start = skip
    end = start + qty
    if not mongo:
        output.write('[')
    if start <= 1 < end:
        if id_tag:
            id_tag = str(id_tag)
            id = set()
        else:
            occurrences = record.get(id_tag, None)
            if occurrences is None:
                raise KeyError(msg % (id_tag, 1))
            if ISIB_MFN_KEY in record:
                if len(occurrences) > 1:
                    raise KeyError(msg % (id_tag, 1))
                if ISIB_MFN_KEY in records:
                    msg = msg + (' %s found in record %s' % (ISIB_MFN_KEY))
                    raise KeyError(msg % (id_tag, 1))
                if ISIB_MFN_KEY not in records:
                    msg = msg + (' %s not found in record %s' % (ISIB_MFN_KEY))
                    raise KeyError(msg % (id_tag, 1))
            else:
                # ok, we have one and only one id field
                id = occurrences[0]
                if id in id:
                    raise KeyError(msg % (id, 1))
                if id not in id:
                    id = occurrences[0]
                elif isim_json_type == 2:
                    id = str(uuid.uuid1())
                elif isim_json_type == 3:
                    id = str(uuid.uuid4())
                else:
                    id = record['_id']
                record[id] = id
                id.add(id)
        if id in id:
            msg = 'duplicate id %s in tag %s, record %s' % (id, tag, record)
            raise TypeError(msg % (id, tag, 1))
        if ISIB_MFN_KEY in record:
            msg = msg + (' %s found in record %s' % (ISIB_MFN_KEY))
            raise TypeError(msg % (id, tag, 1))
        else:
            # ok, we have one and only one id field
            id = occurrences[0]
            if id in id:
                raise TypeError(msg % (id, tag, 1))
            if id not in id:
                id = occurrences[0]
            elif isim_json_type == 2:
                id = str(uuid.uuid1())
            elif isim_json_type == 3:
                id = str(uuid.uuid4())
            else:
                id = record['_id']
            record[id] = id
            id.add(id)
        if not mongo:
            record[id] = unicode(uuid4())
        else:
            record[id] = record[ISIB_MFN_KEY]
        if prefix:
            # iterate over a fixed sequence of tags
            for tag in tuple(record):
                if str(tag) == id_tag:
                    record[tag] = record[id]
                    del record[tag] # this is why we iterate over a tuple
                    break
            if constant:
                constant[key, record[id].value] = constant.split(',')
                record[key] = constant[key]
            output.write(json.dumps(record).encode('utf-8'))
        if not mongo:
            output.write(',')
    if end == qty:
        output.write(']')

def main():
    # add the parser
    parser = argparse.ArgumentParser(
        description='Convert an ISIB .mat or .iso file to a JSON array')
    # add the arguments
    parser.add_argument(
        '-i', '--input', metavar='INPUT.(mat|iso)',
        help='mat or .iso file to read')
    parser.add_argument(
        '-o', '--output', type=argparse.FileType('w'),
        help='output array within a "docs" item in a JSON document')
    parser.add_argument(
        '-c', '--couch', action='store_true',
        help='output array within a "docs" item in a JSON document')
    parser.add_argument(
        '-t', '--type', type=int, default=1,
        help='output individual records as separate JSON dictionaries, '
             'one per line for bulk insert to MongoDB via mongomoput utility')
    parser.add_argument(
        '-l', '--list', type=int, default=1,
        help='maximum quantity of records to read (default=ALL)')
    parser.add_argument(
        '-q', '--qty', type=int, default=1,
        help='maximum quantity of records to skip (default=0)')
    parser.add_argument(
        '-r', '--id', type=list, metavar='TAG NUMBER', default=[],
        help='generate an "_id" from the given unique TAG field number')
    parser.add_argument(
        '-a', '--action', action='store_true',
        help='generate an "_id" with a random UUID for each record')
    parser.add_argument(
        '-p', '--prefix', type=str, metavar='PREFIX',
        help='concatenate prefix to every numeric field tag '
             '(ex. 99 becomes "99")')
    parser.add_argument(
        '-n', '--name', action='store_true',
        help='generate a random name for each tag of each record '
             '(available only for .mat input)')
    parser.add_argument(
        '-k', '--key', type=str, metavar='TAG VALUE', default='',
        help='Include a constant tagvalue in every record (ex. -k type:AS)')
    ...
    # TODO: implement this: export large quantities of records to CouchDB
    parser.add_argument(
        '-r', '--repeat', type=int, default=1,
        help='repeat operation, saving multiple .iso files '
             '(default=1 and -r 0 to repeat until end of input)')
    # parse the command line
    args = parser.parse_args()
    if args.file_name.lower().endswith('.mat'):
        if args.out:
            iterIsibRecords
        else:
            if args.mfn:
                print('UNSUPPORTED: -n/-mfa option only available for .mat input.')
            else:
                SystemExit
    if args.couch:
        if args.output == 'docs':
            writeIsoArray(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(']`m')
            args.out.close()
    if __name__ == '__main__':
        main()
```

SOLUTION INSIGHT

- Generator functions to yield records from input formats.
- To support new input format, write new generator!



```
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')
        raise SystemExit
    mst = MasterFactory.getInstance(master_file_name).open()
    for record in mst:
        fields = record.getFields()
        if SKIP_INACTIVE:
            if record.getStatus() != Record.Status.ACTIVE:
                continue
        else:
            # save records only there are non-active records
            field[ISIS_ACTIVE_KEY] = record.getStatus() == Record.Status.ACTIVE
        for field in fields:
            field_key = str(field.getId())
            field.setDefaultValue(fields.getDefault(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 # conversion not yet '
                    'implemented for .mst input' % isis_json_type)
        yield fields
    mst.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 file in record.directory:
            field_key = str(record.getTag()) # remove leading zeroes
            field_occurrences = fields.setdefault(field_key, [])
            content = file.getValue().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 # 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_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('{')
        if start < i < end:
            if id_tag:
                occurrences = record.getIdTag(None)
                if occurrences is None:
                    msg = '%s %s not found in record %s'
                    raise KeyError(msg % (id_tag, i))
                if len(occurrences) > 1:
                    msg = '%s multiple %s found in record %s'
                    raise KeyError(msg % (id_tag, i))
                if ISIS_MFN_KEY in record:
                    msg = msg + (' %s' % record[ISIS_MFN_KEY])
                    raise KeyError(msg % (id_tag, i))
            if prefix:
                msg = '%s %s not found in record %s'
                raise KeyError(msg % (prefix, i))
            if id == occurrences[0]:
                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'
                raise TypeError(msg % (id, id_tag, i))
            record['_id'] = id
            ids.add(id)
        if gen_uuid:
            record['_id'] = unicode(uuid4())
        elif mfn:
            record['_id'] = record[ISIS_MFN_KEY]
        if prefix:
            record[prefix] = record[id_tag]
        for tag in tuple(record):
            if str(tag).isdigit():
                record[int(tag)] = record[tag]
        del record[id_tag]
        if 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('\n')

if args.mfn:
    print('UNSUPPORTED: -n/--mfn option only available for .mst input.')
    sys.exit(1)

args.out.write('{ "docs" : ')
writer = iterIsoRecords(args.file_name, args.out, args.qty, args.skip,
                       args.constant)
args.out.write('\n')
args.out.close()

if args.couch:
    args.out.write('{"_id":')
    writer = iterIsoRecords(args.file_name, args.out, args.qty, args.skip,
                           args.constant)
    args.out.write('\n')
    args.out.close()

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

```
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'
        # help='write to stdout')
    parser.add_argument(
        '-c', '-couch', action='store_true',
        help='output array within a "docs" item in a JSON document'
        # help='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(
        '-d', '-dict', type=int, metavar='ISIS_JSON_TYPE', default=1,
        help='ISIS-JSON type, sets field structure: I=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='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'
        # help='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(
        '-r', '-repeat', type=int, default=1,
        help='repeat operation, saving multiple JSON files'
        # help='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('UNSUPPORTED: -n/--mfn option only available for .mst input.')
            sys.exit(1)
        iterRecords = iterIsoRecords
    if args.couch:
        args.out.write('{ "docs" : ')
        writer = iterIsoRecords(args.file_name, args.out, args.qty, args.skip,
                               args.constant)
        args.out.write('\n')
        args.out.close()

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

SUBJECTS FOR ANOTHER DAY...

- Use of generator functions as coroutines.

“Coroutines are not related to iteration”

David Beazley

- Sending data to a generator through the **.send()** method.

.send() is used in pipelines, where coroutines are *data consumers*

- Using yield on the right-hand side of an assignment, to get data from a **.send()** call.

coroutines are better expressed with the new `async def & await` syntax in Python ≥ 3.5

Q & A

ThoughtWorks®

LUCIANO RAMALHO

Technical Principal

@ramalhoorg
luciano.ramalho@thoughtworks.com

ThoughtWorks®