

Iterators in Action

Jim Baker

bivio Software

jbaker@zyasoft.com, jbaker@bivio.biz

Benefits of Iterators

- Concise code
- Better performance
- Fun programming
- Other benefits?



A fan of itertools

My Background

- **It's not my work!**
- Extensive usage in some software I developed
- Contributed a few recipes
 - relational joins, observer coroutines

Overview

- Basics
 - protocol, creating, consuming, one liners
- Some examples
 - Six Sigma, YAF, imerge, peek, bisecttrie
- Context managers
- Coroutines

What I Left Out

- No code animations
- No dancers, music, smoke & mirrors
- No time for
 - Recursive generators
 - Coroutines in-depth
 - WSGI, Twisted, databases

Iterator Interface

- Iteration protocol
 - `__iter__()`, `iter()`
 - `next()`
- `for` implicitly uses this protocol
- Or call directly

What's the Big Deal??!!

The Big Deal

- Simplicity, composability
- Deep integration into Python
- Efficiency
 - Minimize resource consumption
 - Avoid function call overhead
 - Can use C goodness - itertools



Creating Iterators

- Iterables - dict, list, tuple, set, frozenset
- Built-ins - enumerate, reversed, sorted, xrange
- Generators
- Generator Expressions
- itertools - islice, tee, chain, izip, **recipes**
- **Implement the iterator protocol!**

Consuming Iterators

- Collections - dict, list, tuple, set, frozenset
- Functions - all, any, min, max, sum, **pipelines**
- for loops - stmt, comprehension, gen exp
- `contextlib.contextmanager` decorator
- Calling `next()` explicitly - **don't forget!**

Strongly Connected Components

```
def strongly_connected_components(G):
    ordering = toposort(G)
    R = reverse(G)
    coloring = dict()
    for v in ordering:
        component = frozenset(DFS_visit(R, coloring, v))
        if component:
            yield component

sorted(strongly_connected_components(G),
       key=len, reverse=True)
```

Or with Generator Exps

```
def strongly_connected_components(G):  
    ordering = toposort(G)  
    R = reverse(G)  
    coloring = dict()  
    components = (component for component in  
                  (frozenset(DFS_visit(R, coloring, v))  
                   for v in ordering)  
                  if component)  
    return sorted(components, key=len, reverse=True)
```

Or reduce away...

```
def strongly_connected_components(G):  
    ordering = toposort(G)  
    R = reverse(G)  
    coloring = dict()  
    return sorted(  
        (component for component in  
         (frozenset(DFS_visit(R, coloring, v))  
          for v in ordering)  
         if component), key=len, reverse=True)
```

6 Sigma Control Charts

- Charts a process
- Determines if process is “in-control”
- Compares against
 - Other processes, views on a given process
 - Specification limits (external to the proc.)

Six Sigma: Xbar & R

- Continuously valued measurements
- Rational subgrouping
 - Central limit theorem
 - Data reduction
- Mean of the process (\bar{X}) vs its range (R)

Xbar & R Computation

- For each subgroup...
 - Compute its mean (\bar{X}) and range (R)
- Compute upper, lower control limits
 - \bar{X}_{dbar} , R_{bar}
- Compute sigma (process standard deviation)

Xbar & R Data Series

```
def XbarR_raw(X, n):  
    """Computes Xbar, R"""  
  
    for grouping in subgroup(X, n):  
        Xbar = mean(grouping)  
        R = max(grouping) - min(grouping)  
        yield Xbar, R
```

subgroup Generator

```
def subgroup(X, n, partial=False):
    grouping = []
    for i, x in enumerate(X):
        if i % n == 0 and i > 0:
            yield grouping
            grouping = []
        grouping.append(x)

    if len(grouping) == n or (partial and grouping):
        yield grouping
```

could use wherever you
need to chunk the iterator
- to page data, perform a
bulk commit, etc.

Similar to itertools recipe
grouper - but longer!

mean Consumer

```
def mean(X):  
    sum = 0.  
    count = 0  
    for x in X:  
        sum += x  
        count += 1  
    return sum/count
```

- OR -

```
def mean(X):  
    X = list(X)  
    return sum(X)/len(X)
```

Xbar & R Constructor

```
def __init__(self, name, series, n=5):
    A2, D3, D4, d2 = self.A2, self.D3, self.D4, self.d2

    self.name = name
    self.normalized = normalized = list(XbarR_raw(series, n))
    self.centerline = centerline = \
        mean(Xbar for Xbar, R in normalized)
    self.Rbar = Rbar = mean(R for Xbar, R in normalized)
    self.Xbar_UCL = centerline + A2[n]*Rbar
    self.Xbar_LCL = centerline - A2[n]*Rbar
    self.R_UCL = D4[n]*Rbar
    if n >= 7:
        self.R_LCL = D3[n]*Rbar
    else:
        self.R_LCL = None

    self.sigma = Rbar/d2[n]
```

Cure Times, in YAF

No.	ct1	ct2	ct3	ct4
1	27.34667	27.50085	29.94412	28.21249
2	27.79695	26.15006	31.21295	31.33272
...				
24	30.04835	27.23709	22.01801	28.69624
25	29.30273	30.83735	30.82735	31.90733

Reading YAF Data

```
from itertools import islice

def read_series(text):
    rows = (row for row in text \
            if row.strip())
    for row in islice(rows, 1, None):
        for col in islice(row.split(), 1, None):
            yield float(col)
```

Putting It Together

Replace with
matplotlib!

That's super ugly

```
print XbarR("Cure Time", series=read_series(data), n=4)
>>> Xbar&R(name=Cure Time,
Xbar(avg=30.40, UCL=34.73, LCL=26.08, sigma=2.88),
Range(avg=5.93, UCL=13.54, LCL=None))
```

X-bar Control Chart for Curetime



R Control Chart for Curetime



imerge (R. Hettinger)

```
import heapq
def imerge(*iterables):
    h = []
    for it in map(iter, iterables):
        try: h.append([it.next(), it])
        except StopIteration: pass
    heapq.heapify(h)

    while 1:
        try:
            while 1:
                value, it = top = h[0]
                yield value
                top[0] = it.next()
                heapq._siftup(h, 0) # maintain heapq invariant
            except StopIteration:
                heapq.heappop(h)
        except IndexError:
            return
```

solution:
Raymond
Hettinger,
modified to remove
non-pertinent
optimizations

imerge Usage

```
from imerge import imerge
from glob import iglob # 2.5, or use glob
from logparser import logparser

def consolidated_log(pattern):
    return imerge(logparser(open(path)) \
                  for path in iglob(pattern))

for ts, record in consolidated_log(pattern):
    analyze_record(record)
```

itertools.tee

- Manages the buffering of iterators
- Can take advantage of `__copy__`
- Foundation for `pairwise`, `peek`, etc

peek (Peter Otten)

```
import itertools

def peek(iterable, n=None):
    a, b = itertools.tee(iterable)
    if n is None:
        return a.next(), b
    else:
        return list(itertools.islice(a, n)), b
```

Using peek

- State machines
- Complex log file parsing
- Sweep algorithms
- Etc.

Hand-waving time!

No time to show complex algorithms. So just trust me!


Complex log parsing. Have you ever seen a situation where literal text blocks with carriage returns, whatever is placed into a log file?

Not something you would do yourself.

But let's say you had a scenario where you can't parse a timestamp... might have to conclude it's continued from the line before

Context Managers

- Enables “Resource Allocation Is Initialization”
- Use `with`-statement to control scope
- Old alternative: `try-except-finally`
- Examples
 - Database sessions, transactions, cursors
 - Files, locks, other resources
 - Observers!



RAII is a big deal
SQLAlchemy on
Jython

Using Coroutines

```
from __future__ import with_statement
from observer import consumer, observation

@consumer
def do_something():
    while True:
        stuff = (yield)
        # now do something with stuff

container = {}
with observation(observe=container,
                 notify=[do_something()]) as observed:
    # now modify `observed`, changes sent to coroutine
```

contextlib



This is nice &
short!

from my recipe,
observer
coroutines

```
from contextlib import contextmanager
```

```
@contextmanager
```

```
def observation(observe, notify, on_delete=False):  
    yield Observation(observe, notify, on_delete)
```

bisecttrie

- Store records in a trie, as simply as possible
- Lookup by prefix for a subset of record attrs
- Return ordered by some relevancy metric
 - Important if only looking up a short prefix
 - Currency, urgency, close associate, etc.
- **Fast! Simple! Enterprise class!**

Using bisecttrie

```
# load it up
cursor = session.cursor()
trie = BisectTrie(mangle=True)
cursor.execute(stmt)
trie.load((row[0:9], row[0]) \
         for row in cursor)

# then sometime later
most_relevant_keys = heapq.nsmallest(20, \
    ((relevancy(item), k) for k in trie.find(prefix)))
```

Loading bisecttrie

```
def load(self, it):
    for Ks,V in it:
        for K in Ks:
            try:
                mangled = self.mangler(K)
                if mangled:
                    self.lookup.append((mangled,V))
            except:
                pass
    self.lookup.sort()
```

Finding with bisecttrie

```
def find(self, prefix):
    lookup = self.lookup
    mangled = self.mangler(prefix)
    len_mangled = len(mangled)

    i = bisect_left(lookup, (mangled, None))
    seen = set()
    while True:
        K, V = lookup[i]
        if K[:len_mangled] > mangled:
            break
        if V not in seen:
            yield V
            seen.add(V)
        i += 1
```

Hamming Numbers

- All numbers $2^i 3^j 5^k$ where $i, j, k \geq 0$
- Sorted!
- Classic functional programming problem
 - Mark Jason Dominus, *Higher Order Perl*
 - `test_generators.py` in Python test suite
 - Even better: `tee`

Hamming (Jeff Epler)

```
def hamming():
    def _hamming(j, k):
        yield 1
        hamming = generators[j]
        for i in hamming:
            yield i * k
    generators = []
    generator = imerge(_hamming(0, 2), \
        imerge(_hamming(1, 3), _hamming(2, 5)))
    generators[:] = tee(generator, 4)
    return generators[3]

for i, num in enumerate(islice(\
    hamming(), 2000000, 2000100)):
    print i + 2000000, num
```

snake eating its
tail!

feedback/
recurrence

Questions?