Python Generator Hacking

David Beazley
http://www.dabeaz.com

Presented at USENIX Technical Conference
San Diego, June 2009

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Introduction

- At PyCon'2008 (Chicago), I gave a popular tutorial on generator functions

  http://www.dabeaz.com/generators

- At PyCon'2009 (Chicago), I followed it up with a tutorial on coroutines (a related topic)

  http://www.dabeaz.com/coroutines

- This tutorial is a kind of "mashup"
- Details from both, but not every last bit

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Goals

- Take a look at Python generator functions
- A feature of Python often overlooked, but which has a large number of practical uses
- Especially for programmers who would be likely to attend a USENIX conference
- So, my main goal is to take this facet of Python, shed some light on it, and show how it's rather "nifty."

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Support Files

- Files used in this tutorial are available here:

http://www.dabeaz.com/usenix2009/generators/

- Go there to follow along with the examples

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Disclaimer

- This isn't meant to be an exhaustive tutorial on every possible use of generators and related theory
- Will mostly go through a series of examples
- You'll have to consult Python documentation for some of the more subtle details

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part I

Introduction to Iterators and Generators

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration

- As you know, Python has a "for" statement
- You use it to iterate over a collection of items

```
>>> for x in [1,4,5,10]:
...     print x,
...
1 4 5 10
>>>
```

- And, as you have probably noticed, you can iterate over many different kinds of objects (not just lists)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iterating over a Dict

- If you iterate over a dictionary you get keys

```
>>> prices = { 'GOOG' : 490.10,
...            'AAPL' : 145.23,
...            'YHOO' : 21.71 }
...
>>> for key in prices:
...     print key
...
YHOO
GOOG
AAPL
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iterating over a String

- If you iterate over a string, you get characters

```
>>> s = "Yow!"
>>> for c in s:
...     print c
...
Y
o
w
!
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iterating over a File

- If you iterate over a file you get lines

```
>>> for line in open("real.txt"):
...     print line,
...
         Real Programmers write in FORTRAN

     Maybe they do now,
     in this decadent era of
     Lite beer, hand calculators, and "user-friendly" softwa
     but back in the Good Old Days,
     when the term "software" sounded funny
     and Real Computers were made out of drums and vacuum tu
     Real Programmers wrote in machine code.
     Not FORTRAN.  Not RATFOR.  Not, even, assembly language
     Machine Code.
     Raw, unadorned, inscrutable hexadecimal numbers.
     Directly.
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Consuming Iterables

- Many functions consume an "iterable"
- Reductions:

```
sum(s), min(s), max(s)
```

- Constructors

```
list(s), tuple(s), set(s), dict(s)
```

- in operator

```
item in s
```

- Many others in the library

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration Protocol

- The reason why you can iterate over different objects is that there is a specific protocol

```
>>> items = [1, 4, 5]
>>> it = iter(items)
>>> it.next()
1
>>> it.next()
4
>>> it.next()
5
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration Protocol

- An inside look at the for statement

```
for x in obj:
    # statements
```

- Underneath the covers

```
_iter = obj.__iter__()        # Get iterator object
while 1:
    try:
         x = _iter.next()     # Get next item
    except StopIteration:     # No more items
         break
    # statements
    ...
```

- Any object that implements this programming convention is said to be "iterable"

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Supporting Iteration

- User-defined objects can support iteration
- Example: a "countdown" object

```
>>> for x in countdown(10):
...     print x,
...
10 9 8 7 6 5 4 3 2 1
>>>
```

- To do this, you just have to make the object implement the iteration protocol

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Supporting Iteration

- One implementation

```
class countdown(object):
    def __init__(self,start):
        self.count = start
    def __iter__(self):
        return self
    def next(self):
        if self.count <= 0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration Example

- Example use:

```
>>> c = countdown(5)
>>> for i in c:
...     print i,
...
5 4 3 2 1
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Supporting Iteration

- Sometimes iteration gets implemented using a pair of objects (an "iterable" and an "iterator")

```
class countdown(object):
    def __init__(self,start):
        self.count = start
    def __iter__(self):
        return countdown_iter(self.count)

def countdown_iter(object):
    def __init__(self,count):
        self.count = count
    def next(self):
        if self.count <= 0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration Example

- Having a separate "iterator" allows for nested iteration on the same object

```
>>> c = countdown(5)
>>> for i in c:
...     for j in c:
...         print i,j
...
5 5
5 4
5 3
5 2
...
1 3
1 2
1 1
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration Commentary

- There are many subtle details involving the design of iterators for various objects
- However, we're not going to cover that
- This isn't a tutorial on "iterators"
- We're talking about generators...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generators

- A generator is a function that produces a sequence of results instead of a single value

```
def countdown(n):
    while n > 0:
        yield n
        n -= 1
>>> for i in countdown(5):
...     print i,
...
5 4 3 2 1
>>>
```

- Instead of returning a value, you generate a series of values (using the yield statement)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generators

- Behavior is quite different than normal func
- Calling a generator function creates an generator object. However, it does not start running the function.

```
def countdown(n):
    print "Counting down from", n
    while n > 0:
        yield n
        n -= 1

>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>>
```

Notice that no output was produced

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generator Functions

- The function only executes on next()

```
>>> x = countdown(10)
>>> x
<generator object at 0x58490>
>>> x.next()
Counting down from 10
10
>>>
```

Function starts executing here

- yield produces a value, but suspends the function
- Function resumes on next call to next()

```
>>> x.next()
9
>>> x.next()
8
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generator Functions

- When the generator returns, iteration stops

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

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generator Functions

- A generator function is mainly a more convenient way of writing an iterator
- You don't have to worry about the iterator protocol (.next, .__iter__, etc.)
- It just works

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generators vs. Iterators

- A generator function is slightly different than an object that supports iteration
- A generator is a one-time operation. You can iterate over the generated data once, but if you want to do it again, you have to call the generator function again.
- This is different than a list (which you can iterate over as many times as you want)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Digression : List Processing

- If you've used Python for awhile, you know that it has a lot of list-processing features
- One feature, in particular, is quite useful
- List comprehensions

```
>>> a = [1,2,3,4]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8]
>>>
```

- Creates a new list by applying an operation to all elements of another sequence

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

List Comprehensions

- A list comprehension can also filter

```
>>> a = [1, -5, 4, 2, -2, 10]
>>> b = [2*x for x in a if x > 0]
>>> b
[2,8,4,20]
>>>
```

- Another example (grep)

```
>>> f = open("stockreport","r")
>>> goog = [line for line in f if 'GOOG' in line]
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

List Comprehensions

- General syntax

```
[expression for x in s if condition]
```

- What it means

```
result = []
for x in s:
    if condition:
       result.append(expression)
```

- Can be used anywhere a sequence is expected

```
>>> a = [1,2,3,4]
>>> sum([x*x for x in a])
30
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

List Comp: Examples

- List comprehensions are hugely useful
- Collecting the values of a specific field

```
stocknames = [s['name'] for s in stocks]
```

- Performing database-like queries

```
a = [s for s in stocks if s['price'] > 100
                      and s['shares'] > 50 ]
```

- Quick mathematics over sequences

```
cost = sum([s['shares']*s['price'] for s in stocks])
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generator Expressions

- A generated version of a list comprehension

```
>>> a = [1,2,3,4]
>>> b = (2*x for x in a)
>>> b
<generator object at 0x58760>
>>> for i in b: print b,
...
2 4 6 8
>>>
```

- This loops over a sequence of items and applies an operation to each item
- However, results are produced one at a time using a generator

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generator Expressions

- Important differences from a list comp.
  - Does not construct a list.
  - Only useful purpose is iteration
  - Once consumed, can't be reused
- Example:

```
>>> a = [1,2,3,4]
>>> b = [2*x for x in a]
>>> b
[2, 4, 6, 8]
>>> c = (2*x for x in a)
<generator object at 0x58760>
>>>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generator Expressions

- General syntax

```
(expression for x in s if condition)
```

- What it means

```
for x in s:
    if condition:
       yield expression
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Note on Syntax

- The parens on a generator expression can dropped if used as a single function argument
- Example:

```
sum(x*x for x in s)
```

Generator expression

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Interlude

- There are two basic blocks for generators
- Generator functions:

```
def countdown(n):
    while n > 0:
         yield n
         n -= 1
```

- Generator expressions

```
squares = (x*x for x in s)
```

- In both cases, you get an object that generates values (which are typically consumed in a for loop)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 2

Processing Data Files

(Show me your Web Server Logs)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Programming Problem

Find out how many bytes of data were transferred by summing up the last column of data in this Apache web server log

```
81.107.39.38 -  ... "GET /ply/ HTTP/1.1" 200 7587
81.107.39.38 -  ... "GET /favicon.ico HTTP/1.1" 404 133
81.107.39.38 -  ... "GET /ply/bookplug.gif HTTP/1.1" 200 23903
81.107.39.38 -  ... "GET /ply/ply.html HTTP/1.1" 200 97238
81.107.39.38 -  ... "GET /ply/example.html HTTP/1.1" 200 2359
66.249.72.134 - ... "GET /index.html HTTP/1.1" 200 4447
```

Oh yeah, and the log file might be huge (Gbytes)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

The Log File

- Each line of the log looks like this:

```
81.107.39.38 -  ... "GET /ply/ply.html HTTP/1.1" 200 97238
```

- The number of bytes is the last column

```
bytestr = line.rsplit(None,1)[1]
```

- It's either a number or a missing value (-)

```
81.107.39.38 -  ... "GET /ply/ HTTP/1.1" 304 -
```

- Converting the value

```
if bytestr != '-':
   bytes = int(bytestr)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Non-Generator Soln

- Just do a simple for-loop

```
wwwlog = open("access-log")
total = 0
for line in wwwlog:
    bytestr = line.rsplit(None,1)[1]
    if bytestr != '-':
        total += int(bytestr)

print "Total", total
```

- We read line-by-line and just update a sum
- However, that's so 90s...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Generator Solution

- Let's solve it using generator expressions

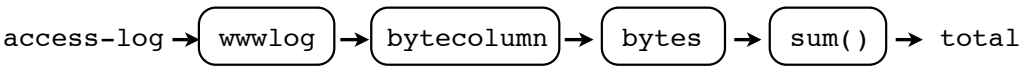
```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

- Whoa! That's different!
  - Less code
  - A completely different programming style

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Generators as a Pipeline

- To understand the solution, think of it as a data processing pipeline



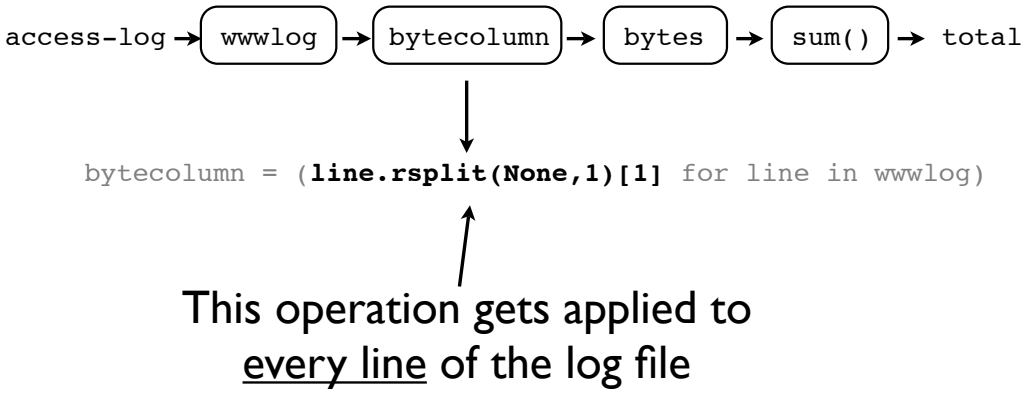
- Each step is defined by iteration/generation

```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Being Declarative

- At each step of the pipeline, we declare an operation that will be applied to the entire input stream



Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Being Declarative

- Instead of focusing on the problem at a line-by-line level, you just break it down into big operations that operate on the whole file
- This is very much a "declarative" style
- The key : Think big...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Iteration is the Glue

- The glue that holds the pipeline together is the iteration that occurs in each step

```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')
print "Total", sum(bytes)
```

- The calculation is being driven by the last step
- The sum() function is consuming values being pulled through the pipeline (via .next() calls)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance

- Surely, this generator approach has all sorts of fancy-dancy magic that is slow.
- Let's check it out on a 1.3Gb log file...

```
% ls -l big-access-log
-rw-r--r-- beazley 1303238000 Feb 29 08:06 big-access-log
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance Contest

```
wwwlog = open("big-access-log")
total = 0
for line in wwwlog:
    bytestr = line.rsplit(None,1)[1]
    if bytestr != '-':
        total += int(bytestr)

print "Total", total
```

Time

27.20

```
wwwlog     = open("big-access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Time

25.96

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Commentary

- Not only was it not slow, it was 5% faster
- And it was less code
- And it was relatively easy to read
- And frankly, I like it a whole better...

"Back in the old days, we used AWK for this and we liked it. Oh, yeah, and get off my lawn!"

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance Contest

```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Time

25.96

```
% awk '{ total += $NF } END { print total }' big-access-log
```

Note:extracting the last column might not be awk's strong point (it's often quite fast)

Time

37.33

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Food for Thought

- At no point in our generator solution did we ever create large temporary lists
- Thus, not only is that solution faster, it can be applied to enormous data files
- It's competitive with traditional tools

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

More Thoughts

- The generator solution was based on the concept of pipelining data between different components
- What if you had more advanced kinds of components to work with?
- Perhaps you could perform different kinds of processing by just plugging various pipeline components together

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

This Sounds Familiar

- The Unix philosophy
- Have a collection of useful system utils
- Can hook these up to files or each other
- Perform complex tasks by piping data

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 3

Fun with Files and Directories

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Programming Problem

You have hundreds of web server logs scattered across various directories. In additional, some of the logs are compressed. Modify the last program so that you can easily read all of these logs

```
foo/
    access-log-012007.gz
    access-log-022007.gz
    access-log-032007.gz
    ...
    access-log-012008
bar/
    access-log-092007.bz2
    ...
    access-log-022008
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

os.walk()

- A very useful function for searching the file system

```
import os

for path, dirlist, filelist in os.walk(topdir):
    # path     :  Current directory
    # dirlist  :  List of subdirectories
    # filelist :  List of files
    ...
```

- This utilizes generators to recursively walk through the file system

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

find

- Generate all filenames in a directory tree that match a given filename pattern

```
import os
import fnmatch

def gen_find(filepat,top):
    for path, dirlist, filelist in os.walk(top):
        for name in fnmatch.filter(filelist,filepat):
            yield os.path.join(path,name)
```

- Examples

```
pyfiles = gen_find("*.py","/")
logs    = gen_find("access-log*","/usr/www/")
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance Contest

```
pyfiles = gen_find("*.py","/")
for name in pyfiles:
    print name
```

Wall Clock Time

559s

```
% find / -name '*.py'
```

Wall Clock Time

468s

Performed on a 750GB file system containing about 140000 .py files

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A File Opener

- Open a sequence of filenames

```
import gzip, bz2
def gen_open(filenames):
    for name in filenames:
        if name.endswith(".gz"):
             yield gzip.open(name)
        elif name.endswith(".bz2"):
             yield bz2.BZ2File(name)
        else:
             yield open(name)
```

- This is interesting.... it takes a sequence of filenames as input and yields a sequence of open file objects (with decompression if needed)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

cat

- Concatenate items from one or more source into a single sequence of items

```
def gen_cat(sources):
    for s in sources:
        for item in s:
            yield item
```

- Example:

```
lognames = gen_find("access-log*", "/usr/www")
logfiles = gen_open(lognames)
loglines = gen_cat(logfiles)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

grep

- Generate a sequence of lines that contain a given regular expression

```
import re

def gen_grep(pat, lines):
    patc = re.compile(pat)
    for line in lines:
        if patc.search(line): yield line
```

- Example:

```
lognames = gen_find("access-log*", "/usr/www")
logfiles = gen_open(lognames)
loglines = gen_cat(logfiles)
patlines = gen_grep(pat, loglines)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example

- Find out how many bytes transferred for a specific pattern in a whole directory of logs

```
pat        = r"somepattern"
logdir     = "/some/dir/"

filenames  = gen_find("access-log*",logdir)
logfiles   = gen_open(filenames)
loglines   = gen_cat(logfiles)
patlines   = gen_grep(pat,loglines)
bytecolumn = (line.rsplit(None,1)[1] for line in patlines)
bytes      = (int(x) for x in bytecolumn if x != '-')

print "Total", sum(bytes)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Important Concept

- Generators decouple iteration from the code that uses the results of the iteration
- In the last example, we're performing a calculation on a sequence of lines
- It doesn't matter where or how those lines are generated
- Thus, we can plug any number of components together up front as long as they eventually produce a line sequence

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 4

Parsing and Processing Data

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Programming Problem

Web server logs consist of different columns of data. Parse each line into a useful data structure that allows us to easily inspect the different fields.

```
81.107.39.38 - - [24/Feb/2008:00:08:59 -0600] "GET ..." 200 7587
```

```
host referrer user [datetime] "request" status bytes
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Parsing with Regex

- Let's route the lines through a regex parser

```
logpats = r'(\S+) (\S+) (\S+) \[(.*?)\] '\
          r'"(\S+) (\S+) (\S+)" (\S+) (\S+)'

logpat = re.compile(logpats)

groups  = (logpat.match(line) for line in loglines)
tuples  = (g.groups() for g in groups if g)
```

- This generates a sequence of tuples

```
('71.201.176.194', '-', '-', '26/Feb/2008:10:30:08 -0600',
'GET', '/ply/ply.html', 'HTTP/1.1', '200', '97238')
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Tuple Commentary

- I generally don't like data processing on tuples

```
('71.201.176.194', '-', '-', '26/Feb/2008:10:30:08 -0600',
'GET', '/ply/ply.html', 'HTTP/1.1', '200', '97238')
```

- First, they are immutable--so you can't modify
- Second, to extract specific fields, you have to remember the column number--which is annoying if there are a lot of columns
- Third, existing code breaks if you change the number of fields

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Tuples to Dictionaries

- Let's turn tuples into dictionaries

```
colnames   = ('host','referrer','user','datetime',
              'method','request','proto','status','bytes')

log        = (dict(zip(colnames,t)) for t in tuples)
```

- This generates a sequence of named fields

```
{ 'status'  : '200',
  'proto'   : 'HTTP/1.1',
  'referrer': '-',
  'request' : '/ply/ply.html',
  'bytes'   : '97238',
  'datetime': '24/Feb/2008:00:08:59 -0600',
  'host'    : '140.180.132.213',
  'user'    : '-',
  'method'  : 'GET'}
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Field Conversion

- You might want to map specific dictionary fields through a conversion function (e.g., int(), float())

```
def field_map(dictseq,name,func):
    for d in dictseq:
        d[name] = func(d[name])
        yield d
```

- Example: Convert a few field values

```
log = field_map(log,"status", int)
log = field_map(log,"bytes",
                lambda s: int(s) if s !='-' else 0)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Field Conversion

- Creates dictionaries of converted values

```
{ 'status': 200,
  'proto': 'HTTP/1.1',
  'referrer': '-',
  'request': '/ply/ply.html',
  'datetime': '24/Feb/2008:00:08:59 -0600',
  'bytes': 97238,
  'host': '140.180.132.213',
  'user': '-',
  'method': 'GET'}
```

Note conversion

- Again, this is just one big processing pipeline

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

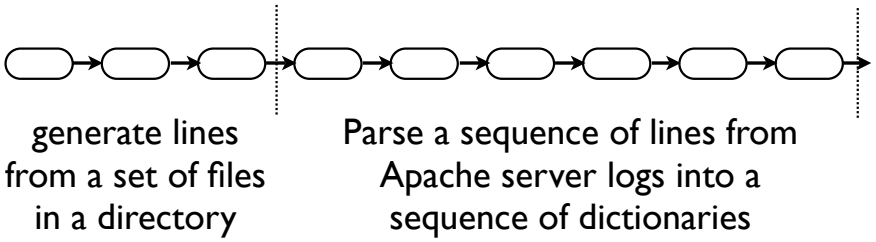
The Code So Far

```
lognames = gen_find("access-log*","www")
logfiles = gen_open(lognames)
loglines = gen_cat(logfiles)
groups   = (logpat.match(line) for line in loglines)
tuples   = (g.groups() for g in groups if g)

colnames = ('host','referrer','user','datetime','method',
            'request','proto','status','bytes')

log      = (dict(zip(colnames,t)) for t in tuples)
log      = field_map(log,"bytes",
                     lambda s: int(s) if s != '-' else 0)
log      = field_map(log,"status",int)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Getting Organized

- As a processing pipeline grows, certain parts of it may be useful components on their own



- A series of pipeline stages can be easily encapsulated by a normal Python function

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Packaging

- Example : multiple pipeline stages inside a function

```
def lines_from_dir(filepat, dirname):
    names   = gen_find(filepat,dirname)
    files   = gen_open(names)
    lines   = gen_cat(files)
    return lines
```

- This is now a general purpose component that can be used as a single element in other pipelines

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Packaging

- Example : Parse an Apache log into dicts

```
def apache_log(lines):
    groups     = (logpat.match(line) for line in lines)
    tuples     = (g.groups() for g in groups if g)

    colnames   = ('host','referrer','user','datetime','method',
                  'request','proto','status','bytes')

    log        = (dict(zip(colnames,t)) for t in tuples)
    log        = field_map(log,"bytes",
                           lambda s: int(s) if s != '-' else 0)
    log        = field_map(log,"status",int)

    return log
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example Use

- It's easy

```
lines = lines_from_dir("access-log*","www")
log   = apache_log(lines)

for r in log:
    print r
```

- Different components have been subdivided according to the data that they process

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Food for Thought

- When creating pipeline components, it's critical to focus on the inputs and outputs
- You will get the most flexibility when you use a standard set of datatypes
- For example, using standard Python dictionaries as opposed to custom objects

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Query Language

- Now that we have our log, let's do some queries
- Find the set of all documents that 404

```
stat404 = set(r['request'] for r in log
                  if r['status'] == 404)
```

- Print all requests that transfer over a megabyte

```
large = (r for r in log
           if r['bytes'] > 1000000)

for r in large:
    print r['request'], r['bytes']
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Query Language

- Find the largest data transfer

```
print "%d %s" % max((r['bytes'],r['request'])
                    for r in log)
```

- Collect all unique host IP addresses

```
hosts = set(r['host'] for r in log)
```

- Find the number of downloads of a file

```
sum(1 for r in log
        if r['request'] == '/ply/ply-2.3.tar.gz')
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Query Language

- Find out who has been hitting robots.txt

```
addrs = set(r['host'] for r in log
             if 'robots.txt' in r['request'])

import socket
for addr in addrs:
    try:
        print socket.gethostbyaddr(addr)[0]
    except socket.herror:
        print addr
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance Study

- Sadly, the last example doesn't run so fast on a huge input file (53 minutes on the 1.3GB log)
- But, the beauty of generators is that you can plug filters in at almost any stage

```
lines = lines_from_dir("big-access-log",".")
lines = (line for line in lines if 'robots.txt' in line)
log   = apache_log(lines)
addrs = set(r['host'] for r in log)
...
```

- That version takes 93 seconds

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Some Thoughts

- I like the idea of using generator expressions as a pipeline query language
- You can write simple filters, extract data, etc.
- You you pass dictionaries/objects through the pipeline, it becomes quite powerful
- Feels similar to writing SQL queries

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 5

Processing Infinite Data

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Question

- Have you ever used 'tail -f' in Unix?

```
% tail -f logfile
...
... lines of output ...
...
```

- This prints the lines written to the end of a file
- The "standard" way to watch a log file
- I used this all of the time when working on scientific simulations ten years ago...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Infinite Sequences

- Tailing a log file results in an "infinite" stream
- It constantly watches the file and yields lines as soon as new data is written
- But you don't know how much data will actually be written (in advance)
- And log files can often be enormous

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Tailing a File

- A Python version of 'tail -f'

```
import time
def follow(thefile):
    thefile.seek(0,2)      # Go to the end of the file
    while True:
         line = thefile.readline()
         if not line:
             time.sleep(0.1)    # Sleep briefly
             continue
         yield line
```

- Idea : Seek to the end of the file and repeatedly try to read new lines. If new data is written to the file, we'll pick it up.

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example

- Using our follow function

```
logfile  = open("access-log")
loglines = follow(logfile)

for line in loglines:
    print line,
```

- This produces the same output as 'tail -f'

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example

- Turn the real-time log file into records

```
logfile  = open("access-log")
loglines = follow(logfile)
log      = apache_log(loglines)
```

- Print out all 404 requests as they happen

```
r404  = (r for r in log if r['status'] == 404)
for r in r404:
    print r['host'],r['datetime'],r['request']
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Commentary

- We just plugged this new input scheme onto the front of our processing pipeline
- Everything else still works, with one caveat-functions that consume an entire iterable won't terminate (min, max, sum, set, etc.)
- Nevertheless, we can easily write processing steps that operate on an infinite data stream

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 6

Decoding Binary Records

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Incremental Parsing

- Generators are a useful way to incrementally parse almost any kind of data
- One example : Small binary encoded records
- Python has a struct module that's used for this
- Let's look at a quick example

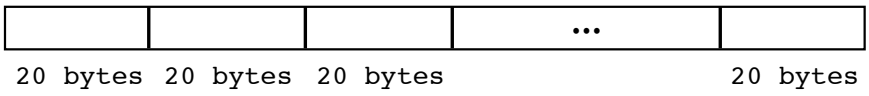
Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Struct Example

- Suppose you had a file of binary records encoded as follows

```
Byte offsets   Description     Encoding
-------------- --------------- ----------------------
0-8            Stock name      (8 byte string)
9-11           Price           (32-bit float)
12-15          Change          (32-bit float)
16-19          Volume          (32-bit unsigned int)
```

- Each record is 20 bytes
- Here's the underlying file



Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Incremental Parsing

- Here's a generator that rips through a file of binary encoded records and decodes them

```
# genrecord.py
import struct

def gen_records(record_format, thefile):
    record_size = struct.calcsize(record_format)
    while True:
          raw_record = thefile.read(record_size)
          if not raw_record:
              break
          yield struct.unpack(record_format, raw_record)
```

- This reads record-by-record and decodes each one using the struct library module

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Incremental Parsing

- Example:

```
from genrecord import *

f = open("stockdata.bin","rb")
records = gen_records("8sffi",f)
for name, price, change, volume in records:
     # Process data
     ...
```

- Notice : Logic concerning the file parsing and record decoding is hidden from view

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Problem

- Reading files in small chunks (e.g., 20 bytes) is grossly inefficient
- It would be better to read in larger chunks with some underlying buffering

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Buffered Reading

- A generator that reads large chunks of data

```
def chunk_reader(thefile, chunksize):
    while True:
        chunk = thefile.read(chunksize)
        if not chunk: break
        yield chunk
```

- A generator that splits chunks into records

```
def split_chunks(chunk,recordsize):
    for n in xrange(0,len(chunk),recordsize):
        yield chunk[n:n+recordsize]
```

- Notice how these are general purpose

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Buffered Reading

- A new version of the record generator

```
# genrecord2.py
import struct

def gen_records(record_format, thefile):
    record_size = struct.calcsize(record_format)
    chunks      = chunk_reader(thefile,1000*record_size)
    records     = split_chunks(chunks,record_size)
    for r in records:
        yield struct.unpack(record_format, r)
```

- This version is reading data 1000 records at a time, but still producing a stream of individual records

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 7

Flipping Everything Around
(from generators to coroutines)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Coroutines and Generators

- Generator functions have been supported by Python for some time (Python 2.3)
- In Python 2.5, generators picked up some new features to allow "coroutines" (PEP-342).
- Most notably: a new send() method
- However, this feature is not nearly as well understood as what we have covered so far

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Yield as an Expression

- In Python 2.5, can use yield as an *expression*
- For example, on the right side of an assignment

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line,
```

- Question : What is its value?

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Coroutines

- If you use yield more generally, you get a coroutine
- These do more than generate values
- Instead, functions can consume values sent to it.

```
>>> g = grep("python")
>>> g.next()              # Prime it (explained shortly)
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>>
```

- Sent values are returned by (yield)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Coroutine Execution

- Execution is the same as for a generator
- When you call a coroutine, nothing happens
- They only run in response to next() and send() methods

```
>>> g = grep("python")
>>> g.next()
Looking for python
>>>
```

Notice that no output was produced

On first operation, coroutine starts running

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Coroutine Priming

- All coroutines must be "primed" by first calling .next() (or send(None))
- This advances execution to the location of the first yield expression.

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print line,
```

.next() advances the coroutine to the first yield expression

- At this point, it's ready to receive a value

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Using a Decorator

- Remembering to call .next() is easy to forget
- Solved by wrapping coroutines with a decorator

```
def coroutine(func):
    def start(*args,**kwargs):
        cr = func(*args,**kwargs)
        cr.next()
        return cr
    return start

@coroutine
def grep(pattern):
    ...
```

- I will use this in most of the future examples

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Closing a Coroutine

- A coroutine might run indefinitely
- Use .close() to shut it down

```
>>> g = grep("python")
>>> g.next()              # Prime it
Looking for python
>>> g.send("Yeah, but no, but yeah, but no")
>>> g.send("A series of tubes")
>>> g.send("python generators rock!")
python generators rock!
>>> g.close()
```

- Note: Garbage collection also calls close()

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Catching close()

- close() can be caught (GeneratorExit)

```
@coroutine
def grep(pattern):
    print "Looking for %s" % pattern
    try:
        while True:
            line = (yield)
            if pattern in line:
                print line,
    except GeneratorExit:
        print "Going away.  Goodbye"
```

- You cannot ignore this exception
- Only legal action is to clean up and return

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 8

Coroutines, Pipelines, and Dataflow

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Processing Pipelines

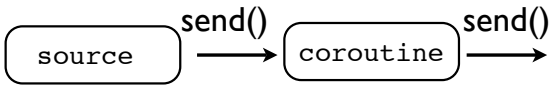
- Coroutines can also be used to set up pipes

- You just chain coroutines together and push data through the pipe with send() operations

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Pipeline Sources

- The pipeline needs an initial source (a producer)



- The source drives the entire pipeline

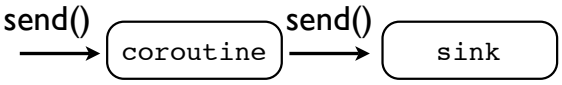
```
def source(target):
    while not done:
        item = produce_an_item()
        ...
        target.send(item)
        ...
    target.close()
```

- It is typically not a coroutine

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Pipeline Sinks

- The pipeline must have an end-point (sink)



- Collects all data sent to it and processes it

```
@coroutine
def sink():
    try:
        while True:
            item = (yield)    # Receive an item
            ...
    except GeneratorExit:     # Handle .close()
         # Done
         ...
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

An Example

- A source that mimics Unix 'tail -f'

```
import time
def follow(thefile, target):
    thefile.seek(0,2)      # Go to the end of the file
    while True:
         line = thefile.readline()
         if not line:
             time.sleep(0.1)    # Sleep briefly
             continue
         target.send(line)
```

- A sink that just prints the lines

```
@coroutine
def printer():
    while True:
         line = (yield)
         print line,
```

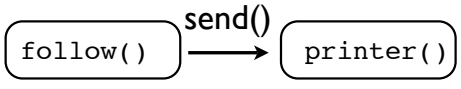
Copyright (C) 2009, David Beazley, http://www.dabeaz.com

An Example

- Hooking it together

```
f = open("access-log")
follow(f, printer())
```

- A picture

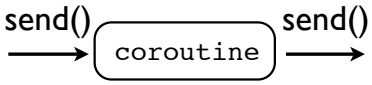


- Critical point : follow() is driving the entire computation by reading lines and pushing them into the printer() coroutine

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Pipeline Filters

- Intermediate stages both receive and send



- Typically perform some kind of data transformation, filtering, routing, etc.

```
@coroutine
def filter(target):
    while True:
        item = (yield)           # Receive an item
        # Transform/filter item
        ...
        # Send it along to the next stage
        target.send(item)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Filter Example

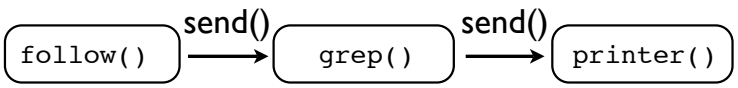
- A grep filter coroutine

```
@coroutine
def grep(pattern,target):
    while True:
        line = (yield)           # Receive a line
        if pattern in line:
            target.send(line)    # Send to next stage
```

- Hooking it up

```
f = open("access-log")
follow(f,
       grep('python',
       printer()))
```

- A picture

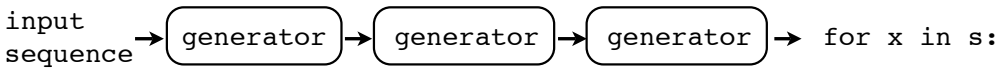


Copyright (C) 2009, David Beazley, http://www.dabeaz.com

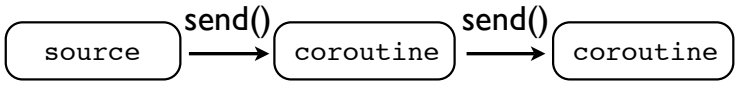
Interlude

- Coroutines flip generators around

generators/iteration



coroutines

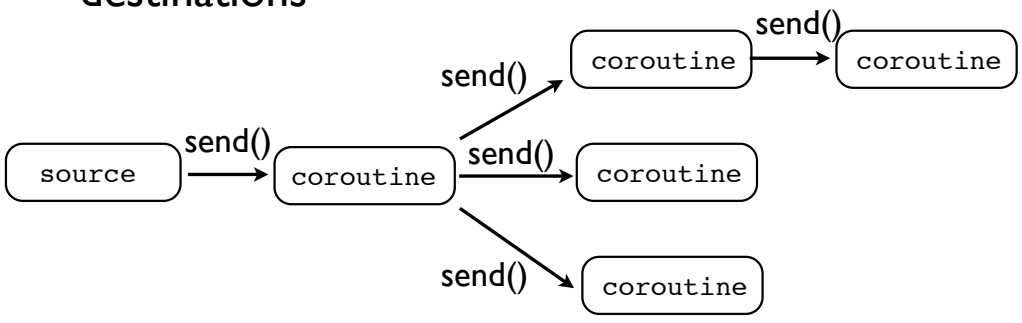


- Key difference. Generators pull data through the pipe with iteration. Coroutines push data into the pipeline with send().

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Being Branchy

- With coroutines, you can send data to multiple destinations



- The source simply "sends" data. Further routing of that data can be arbitrarily complex

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example : Broadcasting

- Broadcast to multiple targets

```
@coroutine
def broadcast(targets):
    while True:
        item = (yield)
        for target in targets:
            target.send(item)
```

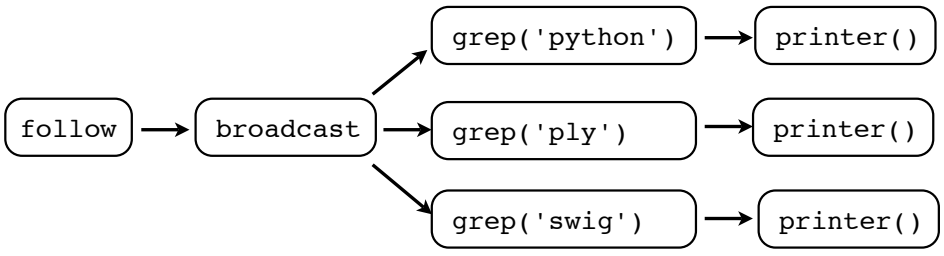
- This takes a sequence of coroutines (targets) and sends received items to all of them.

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example : Broadcasting

- Example use:

```
f = open("access-log")
follow(f,
       broadcast([grep('python',printer()),
                  grep('ply',printer()),
                  grep('swig',printer())])
  )
```

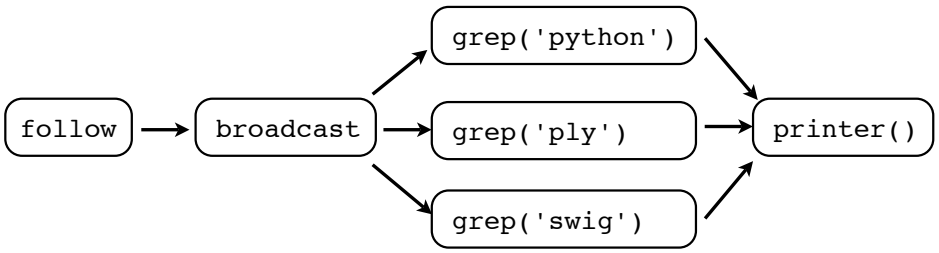


Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Example : Broadcasting

- A more disturbing variation...

```
f = open("access-log")
p = printer()
follow(f,
       broadcast([grep('python',p),
                  grep('ply',p),
                  grep('swig',p)])
  )
```



Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Interlude

- Coroutines provide more powerful data routing possibilities than simple iterators
- If you built a collection of simple data processing components, you can glue them together into complex arrangements of pipes, branches, merging, etc.
- Although you might not want to make it excessively complicated (although that might increase/decrease one's job security)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 9

Coroutines and Event Dispatching

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Event Handling

- Coroutines can be used to write various components that process event streams
- Pushing event streams into coroutines
- Let's look at an example...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Problem

- Where is my ^&#&@* bus?
- Chicago Transit Authority (CTA) equips most of its buses with real-time GPS tracking
- You can get current data on *every* bus on the street as a big XML document
- Use "The Google" to search for details...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

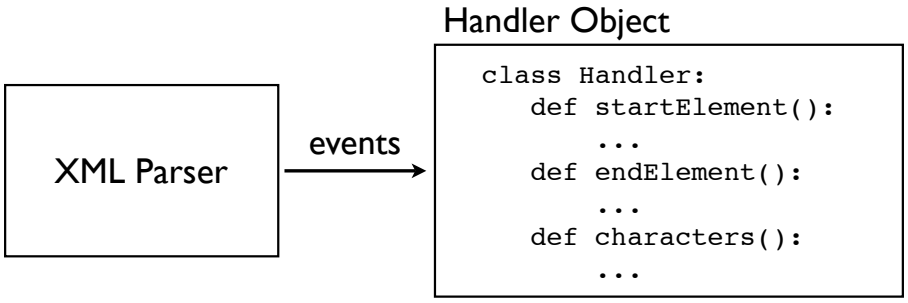
Some XML

```
<?xml version="1.0"?>
  <buses>
    <bus>
      <id>7574</id>
      <route>147</route>
      <color>#3300ff</color>
      <revenue>true</revenue>
      <direction>North Bound</direction>
      <latitude>41.925682067871094</latitude>
      <longitude>-87.63092803955078</longitude>
      <pattern>2499</pattern>
      <patternDirection>North Bound</patternDirection>
      <run>P675</run>
      <finalStop><![CDATA[Paulina & Howard Terminal]]></finalStop>
      <operator>42493</operator>
   </bus>
   <bus>
    ...
   </bus>
  </buses>
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

XML Parsing

- There are many possible ways to parse XML
- An old-school approach: SAX
- SAX is an event driven interface



Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Minimal SAX Example

```
import xml.sax

class MyHandler(xml.sax.ContentHandler):
    def startElement(self,name,attrs):
        print "startElement", name
    def endElement(self,name):
        print "endElement", name
    def characters(self,text):
        print "characters", repr(text)[:40]

xml.sax.parse("somefile.xml",MyHandler())
```

- You see this same programming pattern in other settings (e.g., HTMLParser module)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Some Issues

- SAX is often used because it can be used to incrementally process huge XML files without a large memory footprint
- However, the event-driven nature of SAX parsing makes it rather awkward and low-level to deal with

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

From SAX to Coroutines

- You can dispatch SAX events into coroutines
- Consider this SAX handler

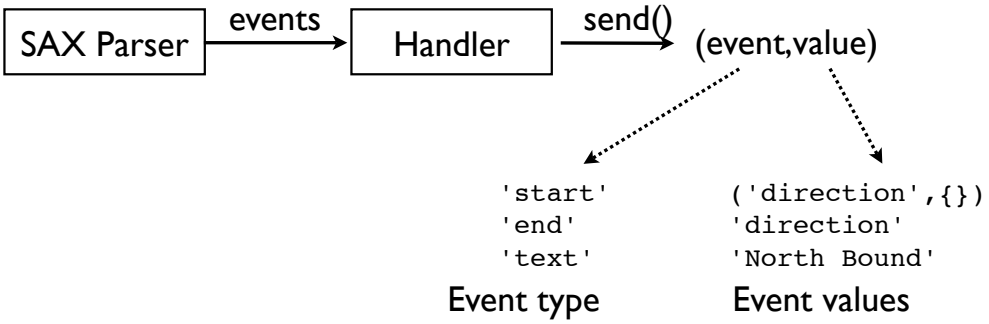
```
import xml.sax

class EventHandler(xml.sax.ContentHandler):
    def __init__(self,target):
        self.target = target
    def startElement(self,name,attrs):
        self.target.send(('start',(name,attrs._attrs)))
    def characters(self,text):
        self.target.send(('text',text))
    def endElement(self,name):
        self.target.send(('end',name))
```

- It does nothing, but send events to a target

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

An Event Stream

- The big picture



- Observe : Coding this was straightforward

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Event Processing

- To do anything interesting, you have to process the event stream
- Example: Convert bus elements into dictionaries (XML sucks, dictionaries rock)

```
<bus>
  <id>7574</id>
  <route>147</route>
  <revenue>true</revenue>
  <direction>North Bound</direction>
  ...
 </bus>
```

```
{
  'id' : '7574',
  'route' : '147',
  'revenue' : 'true',
  'direction' : 'North Bou
  ...
}
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

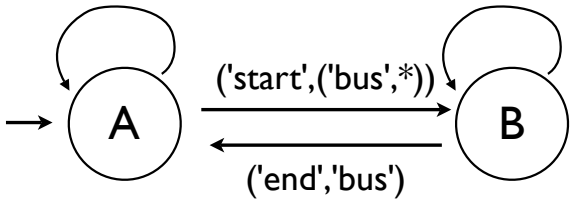
Buses to Dictionaries

```
@coroutine
def buses_to_dicts(target):
    while True:
        event, value = (yield)
        # Look for the start of a <bus> element
        if event == 'start' and value[0] == 'bus':
            busdict = { }
            fragments = []
            # Capture text of inner elements in a dict
            while True:
                event, value = (yield)
                if event == 'start':   fragments = []
                elif event == 'text':  fragments.append(value)
                elif event == 'end':
                    if value != 'bus':
                        busdict[value] = "".join(fragments)
                    else:
                        target.send(busdict)
                        break
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

State Machines

- The previous code works by implementing a simple state machine



- State A: Looking for a bus
- State B: Collecting bus attributes
- Comment : Coroutines are perfect for this

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Buses to Dictionaries

```
@coroutine
def buses_to_dicts(target):
    while True:                                          # A
        event, value = (yield)
        # Look for the start of a <bus> element
        if event == 'start' and value[0] == 'bus':
            busdict = { }
            fragments = []
            # Capture text of inner elements in a dict
            while True:                                  # B
                event, value = (yield)
                if event == 'start':   fragments = []
                elif event == 'text':  fragments.append(value)
                elif event == 'end':
                    if value != 'bus':
                        busdict[value] = "".join(fragments)
                    else:
                        target.send(busdict)
                        break
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Filtering Elements

- Let's filter on dictionary fields

```
@coroutine
def filter_on_field(fieldname,value,target):
    while True:
        d = (yield)
        if d.get(fieldname) == value:
            target.send(d)
```

- Examples:

```
filter_on_field("route","22",target)
filter_on_field("direction","North Bound",target)
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Processing Elements

- Where's my bus?

```
@coroutine
def bus_locations():
    while True:
        bus = (yield)
        print "%(route)s,%(id)s,\"%(direction)s\",\
              "%(latitude)s,%(longitude)s" % bus
```

- This receives dictionaries and prints a table

```
22,1485,"North Bound",41.880481123924255,-87.62948191165924
22,1629,"North Bound",42.01851969751819,-87.6730209876751
...
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Hooking it Together

- Find all locations of the North Bound #22 bus (the slowest moving object in the universe)

```
xml.sax.parse("allroutes.xml",
              EventHandler(
                   buses_to_dicts(
                   filter_on_field("route","22",
                   filter_on_field("direction","North Bound",
                   bus_locations())))
              ))
```

- This final step involves a bit of plumbing, but each of the parts is relatively simple

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

How Low Can You Go?

- I've picked this XML example for reason
- One interesting thing about coroutines is that you can push the initial data source as low-level as you want to make it without rewriting all of the processing stages
- Let's say SAX just isn't quite fast enough...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

XML Parsing with Expat

- Let's strip it down....

```
import xml.parsers.expat

def expat_parse(f,target):
    parser = xml.parsers.expat.ParserCreate()
    parser.buffer_size = 65536
    parser.buffer_text = True
    parser.returns_unicode = False
    parser.StartElementHandler = \
       lambda name,attrs: target.send(('start',(name,attrs)))
    parser.EndElementHandler = \
       lambda name: target.send(('end',name))
    parser.CharacterDataHandler = \
       lambda data: target.send(('text',data))
    parser.ParseFile(f)
```

- expat is low-level (a C extension module)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance Contest

- SAX version (on a 30MB XML input)

```
xml.sax.parse("allroutes.xml",EventHandler(
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

8.37s

- Expat version

```
expat_parse(open("allroutes.xml"),
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

4.51s

(83% speedup)

- No changes to the processing stages

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Going Lower

- You can even drop send() operations into C
- A skeleton of how this works...

```
PyObject *
py_parse(PyObject *self, PyObject *args) {
    PyObject *filename;
    PyObject *target;
    PyObject *send_method;
    if (!PyArg_ParseArgs(args,"sO",&filename,&target)) {
        return NULL;
    }
    send_method = PyObject_GetAttrString(target,"send");
    ...

    /* Invoke target.send(item) */
    args   = Py_BuildValue("(O)",item);
    result = PyEval_CallObject(send_meth,args);
    ...
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Performance Contest

- Expat version

```
expat_parse(open("allroutes.xml"),
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

4.51s

- A custom C extension written directly on top of the expat C library (code not shown)

```
cxmlparse.parse("allroutes.xml",
    buses_to_dicts(
    filter_on_field("route","22",
    filter_on_field("direction","North Bound",
    bus_locations())))))
```

2.95s

(55% speedup)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Interlude

- Processing events is a situation that is well-suited for coroutine functions
- With event driven systems, some kind of event handling loop is usually in charge
- Because of that, it's really hard to twist it around into a programming model based on iteration
- However, if you just push events into coroutines with send(), it works fine.

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Part 10

From Data Processing to Concurrent Programming

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

The Story So Far

- Generators and coroutines can be used to define small processing components that can be connected together in various ways
- You can process data by setting up pipelines, dataflow graphs, etc.
- It's all been done in simple Python programs

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

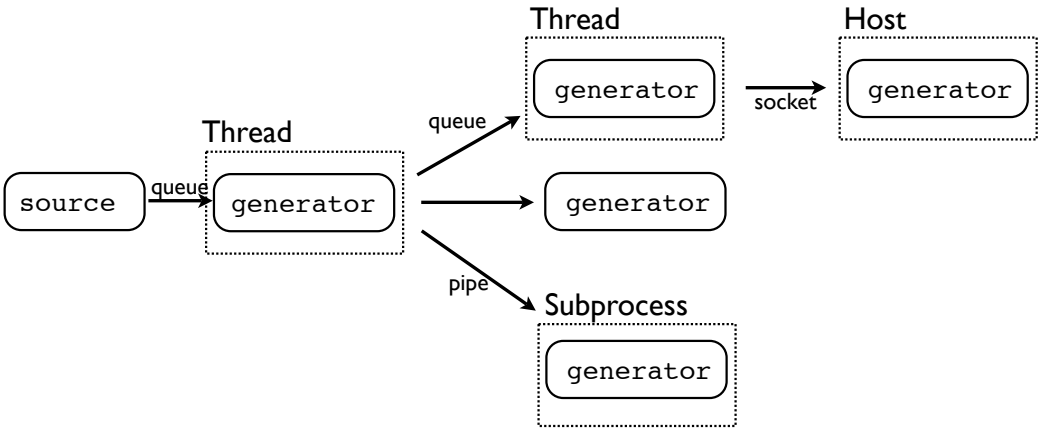
An Interesting Twist

- Pushing data around nicely ties into problems related to threads, processes, networking, distributed systems, etc.
- Could two generators communicate over a pipe or socket?
- Answer : Yes, of course

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Basic Concurrency

- You can package generators inside threads or subprocesses by adding extra layers



- Will sketch out some basic ideas...

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Pipelining over a Pipe

- Suppose you wanted to bridge a generator/ coroutine data flow across a pipe or socket
- Doing that is actually pretty easy
- Just use the pickle module to serialize objects and add some functions to deal with the communication

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Pickler/Unpickler

- Turn a generated sequence into pickled objects

```
def gen_sendto(source,outfile):
    for item in source:
        pickle.dump(item,outfile)

def gen_recvfrom(infile):
    while True:
         try:
             item = pickle.load(infile)
             yield item
         except EOFError:
             return
```

- Now, attach these to a pipe or socket

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Sender/Receiver

- Example: Sender

```
# netprod.py
import subprocess
p = subprocess.Popen(['python','netcons.py'],
                     stdin=subprocess.PIPE)

lines = follow(open("access-log"))
log   = apache_log(lines)
gen_sendto(log,p.stdin)
```

- Example: Receiver

```
# netcons.py
import sys
for r in get_recvfrom(sys.stdin):
    print r
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Subprocess Target

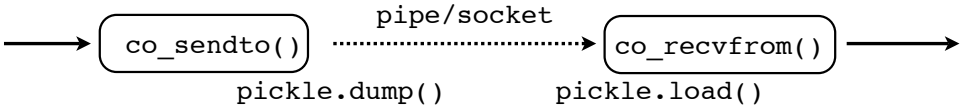
- Bridging coroutines over a pipe/socket

```
@coroutine
def co_sendto(f):
    try:
        while True:
            item = (yield)
            pickle.dump(item,f)
            f.flush()
    except StopIteration:
        f.close()

def co_recvfrom(f,target):
    try:
        while True:
            item = pickle.load(f)
            target.send(item)
    except EOFError:
        target.close()
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Subprocess Target

- High Level Picture



- Of course, the devil is in the details...
- You would not do this unless you can recover the cost of the underlying communication (e.g., you have multiple CPUs and there's enough processing to make it worthwhile)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Process Example

- A parent process

```
# Launch a child process
import subprocess
p = subprocess.Popen(['python','child.py'],
                     stdin=subprocess.PIPE)

# Feed data into the child
xml.sax.parse("allroutes.xml", EventHandler(
     buses_to_dicts(
     co_sendto(p.stdin))))
```

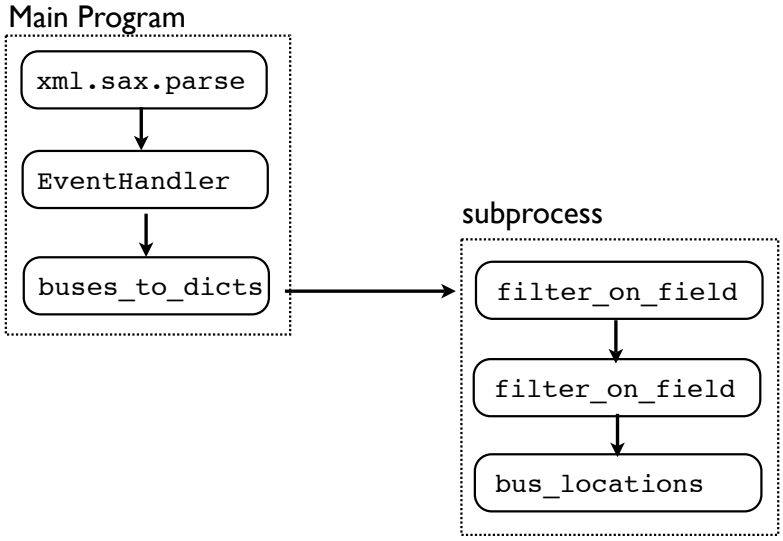
- A child process

```
# child.py
import sys
...
co_recvfrom(sys.stdin,
         filter_on_field("route","22",
         filter_on_field("direction","North Bound",
         bus_locations())))
```

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

A Picture

- Here is an overview of the last example



Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Wrap Up

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

The Big Idea

- Generators are an incredibly useful tool for a variety of "systems" related problem
- Power comes from the ability to set up processing pipelines and route data around
- Can create components that plugged into the pipeline as reusable pieces
- Can extend processing pipelines in many directions (networking, threads, etc.)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Code Reuse

- There is an interesting reuse element
- You create a lot of small processing parts and glue them together to build larger apps
- Personally, I like it a lot better than what I see people doing with various OO patterns involving callbacks (e.g., the strategy design pattern and variants)

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

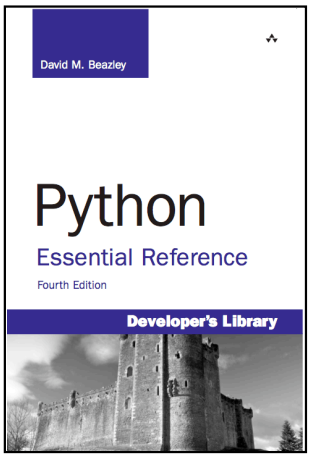
Pitfalls

- Programming with generators involves techniques that a lot of programmers have never seen before in other languages
- Springing this on the uninitiated might cause their head to explode
- Error handling is really tricky because you have lots of components chained together and the control-flow is unconventional
- Need to pay careful attention to debugging, reliability, and other issues.

Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Shameless Plug

- Further details on useful applications of generators and coroutines will be featured in the "Python Essential Reference, 4th Edition"
- Look for it (Summer 2009)
- I also teach Python classes



Copyright (C) 2009, David Beazley, http://www.dabeaz.com

Thanks!

- I hope you got some new ideas from this class
- Please feel free to contact me

http://www.dabeaz.com

Copyright (C) 2009, David Beazley, http://www.dabeaz.com