Generator Tricks
For Systems Programmers

David Beazley
http://www.dabeaz.com

Presented at PyCon'2008

An Introduction

- Generators are cool!

- But what are they?

- And what are they good for?

- That's what this tutorial is about

About Me

- I'm a long-time Pythonista

- First started using Python with version 1.3

- Author : Python Essential Reference

- Responsible for a number of open source Python-related packages (Swig, PLY, etc.)

My Story

My addiction to generators started innocently enough. I was just a happy Python programmer working away in my secret lair when I got "the call." A call to sort through 1.5 Terabytes of C++ source code (~800 weekly snapshots of a million line application). That's when I discovered the os.walk() function. I knew this wasn't going to end well...

Back Story

- I think generators are wicked cool

- An extremely useful language feature

- Yet, they still seem a rather exotic

- I still don't think I've fully wrapped my brain around the best approach to using them

A Complaint

- The coverage of generators in most Python books is lame (mine included)

- Look at all of these cool examples!

  - Fibonacci Numbers

  - Squaring a list of numbers

  - Randomized sequences

- Wow! Blow me over!

This Tutorial

- Some more practical uses of generators

- Focus is "systems programming"

- Which loosely includes files, file systems, parsing, networking, threads, etc.

- My goal : To provide some more compelling examples of using generators

- Planting some seeds

Support Files

- Files used in this tutorial are available here:

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

- Go there to follow along with the examples

Disclaimer

- This isn't meant to be an exhaustive tutorial on generators and related theory

- Will be looking at a series of examples

- I don't know if the code I've written is the "best" way to solve any of these problems.

- So, let's have a discussion

Performance Details

- There are some later performance numbers

- Python 2.5.1 on OS X 10.4.11

- All tests were conducted on the following:

  - Mac Pro 2x2.66 Ghz Dual-Core Xeon

  - 3 Gbytes RAM

  - WDC WD2500JS-41SGB0 Disk (250G)

- Timings are 3-run average of 'time' command

Part I

Introduction to Iterators and Generators

Iteration

- As you know, Python has a "for" statement

- You use it to loop 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)

Iterating over a Dict

- If you loop 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
>>>
```

Iterating over a String

- If you loop over a string, you get characters

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

Iterating over a File

- If you loop 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.
```

Consuming Iterables

- Many functions consume an "iterable" object

- 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

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
>>>
```

Iteration Protocol

- An inside look at the for statement

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

- Underneath the covers

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

- Any object that supports iter() and next() is said to be "iterable."

Supporting Iteration

- User-defined objects can support iteration

- Example: Counting down...

```
>>> 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 __iter__() and next()

Supporting Iteration

- Sample 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
```

Iteration Example

- Example use:

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

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...

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 <u>generate</u> a series of values (using the yield statement)

Generators

- Behavior is quite different than normal func

- Calling a generator function creates an generator object. However, it does <u>not</u> 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

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
>>>
```

Generator Functions

- When the generator returns, iteration stops

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

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

Generators vs. Iterators

- A generator function is slightly different than an object that supports iteration

- A generator is a <u>one-time operation</u>. 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)

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

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>
>>>
```

Generator Expressions

- General syntax

```
(expression for i in s if cond1
             for j in t if cond2
             ...
             if condfinal)
```

- What it means

```
for i in s:
    if cond1:
        for j in t:
            if cond2:
                ...
                if condfinal: yield expression
```

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

Interlude

- We now have two basic building blocks

- Generator functions:

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

- Generator expressions

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

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

Part 2

Processing Data Files

(Show me your Web Server Logs)

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)

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)
```

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...

A Generator Solution

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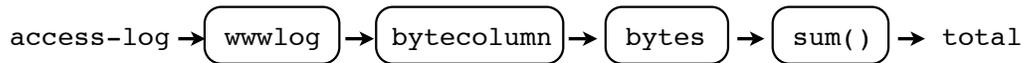
```
wwwlog     = open("access-log")
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
bytes      = (int(x) for x in bytecolumn if x != '-')

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

- Whoa! That's different!

  - Less code

  - A completely different programming style

Generators as a Pipeline

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

```
access-log → [ wwwlog ] → [ bytecolumn ] → [ bytes ] → [ sum() ] → total
```
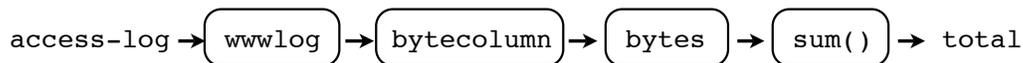
- Each step is defined by iteration/generation

```
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)
```

Being Declarative

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

```
access-log → [ wwwlog ] → [ bytecolumn ] → [ bytes ] → [ sum() ] → total
```

```
bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
```

This operation gets applied to
every line of the log file

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...

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)
```
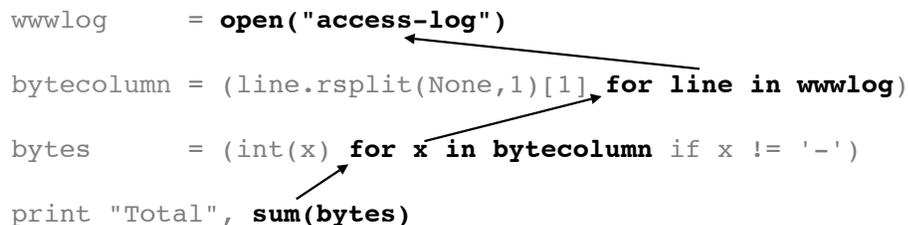
- The calculation is being driven by the last step

- The sum() function is consuming values being pushed through the pipeline (via .next() calls)

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
```

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

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!"

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
```

Time

37.33

Note:extracting the last column may not be awk's strong point

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

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

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

Part 3

Fun with Files and Directories

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
```

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

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/")
```

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

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

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)
```

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)
```

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)
```

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

Part 4

Parsing and Processing Data

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
```

$\downarrow$

host referrer user [datetime] "request" status bytes

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')
```

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'}
```

Field Conversion

- Map specific dictionary fields through a function

```
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)
```

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

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)
```

Packaging

- To make it more sane, you may want to package parts of the code into functions

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

- This is a generate purpose function that reads all lines from a series of files in a directory

Packaging

- Parse an Apache log

```
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
```

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

---

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']
```

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')
```

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
```

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

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

Part 5

Processing Infinite Data

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...

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

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.

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'

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']
```

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

---

Thoughts

- This data pipeline idea is really quite powerful

- Captures a lot of common systems problems

- Especially consumer-producer problems

Part 6

Feeding the Pipeline

Feeding Generators

- In order to feed a generator processing pipeline, you need to have an input source

- So far, we have looked at two file-based inputs

- Reading a file

```
lines = open(filename)
```

- Tailing a file

```
lines = follow(open(filename))
```

Generating Connections

- Generate a sequence of TCP connections

```
import socket
def receive_connections(addr):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    s.bind(addr)
    s.listen(5)
    while True:
        client = s.accept()
        yield client
```

- Example:

```
for c,a in receive_connections(("",9000)):
    c.send("Hello World\n")
    c.close()
```

Generating Messages

- Receive a sequence of UDP messages

```
import socket
def receive_messages(addr,maxsize):
    s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
    s.bind(addr)
    while True:
        msg = s.recvfrom(maxsize)
        yield msg
```

- Example:

```
for msg, addr in receive_messages(("",10000),1024):
    print msg, "from", addr
```

I/O Multiplexing

- Generating I/O events on a set of sockets

```python
import select
def gen_events(socks):
    while True:
        rdr,wrt,err = select.select(socks,socks,socks,0.1)
        for r in rdr:
            yield "read",r
        for w in wrt:
            yield "write",w
        for e in err:
            yield "error",e
```

- Note: Using this one is little tricky

- Example : Reading from multiple client sockets

I/O Multiplexing

```python
clientset = []

def acceptor(sockset,addr):
    for c,a in receive_connections(addr):
        sockset.append(c)

acc_thr = threading.Thread(target=acceptor,
                           args=(clientset,("",12000))
acc_thr.setDaemon(True)
acc_thr.start()

for evt,s in gen_events(clientset):
    if evt == 'read':
        data = s.recv(1024)
        if not data:
            print "Closing", s
            s.close()
            clientset.remove(s)
        else:
            print s,data
```

Consuming a Queue

- Generate a sequence of items from a queue

```
def consume_queue(thequeue):
    while True:
        item = thequeue.get()
        if item is StopIteration: break
        yield item
```

- Note: Using StopIteration as a sentinel

- Might be used to feed a generator pipeline as a consumer thread

# Consuming a Queue

- Example:

```
import Queue, threading

def consumer(q):
    for item in consume_queue(q):
        print "Consumed", item
    print "Done"

in_q = Queue.Queue()
con_thr = threading.Thread(target=consumer,args=(in_q,))
con_thr.start()

for i in xrange(100):
    in_q.put(i)
in_q.put(StopIteration)
```

Part 7

Extending the Pipeline

Multiple Processes

- Can you extend a processing pipeline across processes and machines?

process 1

socket
pipe

process 2
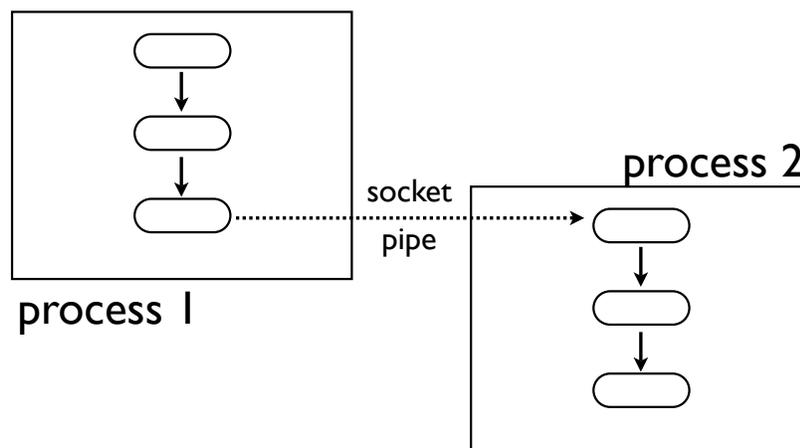
Pickler/Unpickler

- Turn a generated sequence into pickled objects

```
def gen_pickle(source):
    for item in source:
        yield pickle.dumps(item)

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

- Now, attach these to a pipe or socket

Sender/Receiver

- Example: Sender

```
def sendto(source,addr):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.connect(addr)
    for pitem in gen_pickle(source):
        s.sendall(pitem)
    s.close()
```

- Example: Receiver

```
def receivefrom(addr):
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    s.bind(addr)
    s.listen(5)
    c,a = s.accept()
    for item in gen_unpickle(c.makefile()):
        yield item
    c.close()
```

Example Use

- Example: Read log lines and parse into records

```
# netprod.py

lines = follow(open("access-log"))
log   = apache_log(lines)
sendto(log,("",15000))
```
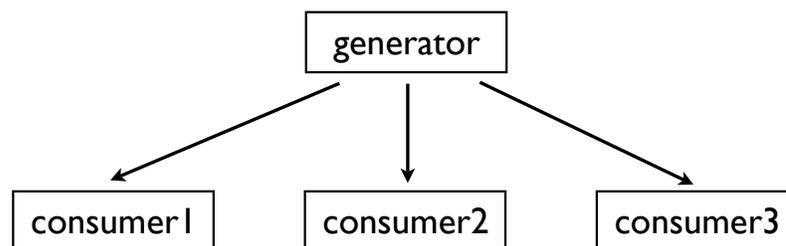
- Example: Pick up the log on another machine

```
# netcons.py
for r in receivefrom(("",15000)):
    print r
```

Fanning Out

- In all of our examples, the processing pipeline is driven by a single consumer

```
for item in gen:
    # Consume item
```

- Can you expand the pipeline to multiple consumers?

Broadcasting

- Consume a generator and send items to a set of consumers

```
def broadcast(source, consumers):
    for item in source:
        for c in consumers:
            c.send(item)
```

- This changes the control-flow

- The broadcaster is what consumes items

- Those items have to be <u>sent to consumers</u> for processing

1-97

Consumers

- To create a consumer, define an object with a send method on it

```
class Consumer(object):
    def send(self,item):
        print self, "got", item
```

- Example:

```
c1 = Consumer()
c2 = Consumer()
c3 = Consumer()

lines = follow(open("access-log"))
broadcast(lines,[c1,c2,c3])
```

1-98

Consumers

- Sadly, inside consumers, it is not possible to continue the same processing pipeline idea

- In order for it to work, there has to be a single iteration that is driving the pipeline

- With multiple consumers, you would have to be iterating in more than one location at once

- You can do this with threads or distributed processes however

Network Consumer

- Example:

```
import socket,pickle
class NetConsumer(object):
    def __init__(self,addr):
        self.s = socket.socket(socket.AF_INET,
                               socket.SOCK_STREAM)
        self.s.connect(addr)
    def send(self,item):
        pitem = pickle.dumps(item)
        self.s.sendall(pitem)
    def close(self):
        self.s.close()
```

- This will route items to a network receiver

Network Consumer

- Example Usage:

```python
class Stat404(NetConsumer):
    def send(self,item):
        if item['status'] == 404:
            NetConsumer.send(self,item)

lines = follow(open("access-log"))
log   = apache_log(lines)

stat404 = Stat404(("somehost",15000))

broadcast(log, [stat404])
```

- The 404 entries will go elsewhere...

Consumer Thread

- Example:

```python
import Queue, threading

class ConsumerThread(threading.Thread):
    def __init__(self,target):
        threading.Thread.__init__(self)
        self.setDaemon(True)
        self.in_queue = Queue.Queue()
        self.target = target
    def send(self,item):
        self.in_queue.put(item)
    def generate(self):
        while True:
            item = self.in_queue.get()
            yield item
    def run(self):
        self.target(self.generate())
```

Consumer Thread

- Sample usage (building on earlier code)

```
def find_404(log):
    for r in (r for r in log if r['status'] == 404):
        print r['status'],r['datetime'],r['request']

def bytes_transferred(log):
    total = 0
    for r in log:
        total += r['bytes']
        print "Total bytes", total

c1 = ConsumerThread(find_404)
c1.start()
c2 = ConsumerThread(bytes_transferred)
c2.start()

lines = follow(open("access-log"))  # Follow a log
log   = apache_log(lines)           # Turn into records
broadcast(log,[c1,c2])              # Broadcast to consumers
```
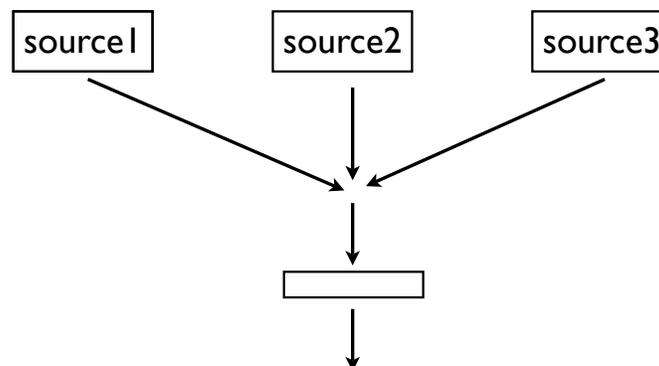

Multiple Sources

- In all of our examples, the processing pipeline is being fed by a single source

- But, what if you had multiple sources?

Concatenation

- Concatenate one source after another

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

- This generates one big sequence

- Consumes each generator one at a time

- Only works with generators that terminate

Parallel Iteration

- Zipping multiple generators together

```
import itertools

z = itertools.izip(s1,s2,s3)
```

- This one is only marginally useful

- Requires generators to go lock-step

- Terminates when the first exits

Multiplexing

- Consumer from multiple generators in real-time--producing values as they are generated

- Example use

```
log1 = follow(open("foo/access-log"))
log2 = follow(open("bar/access-log"))

lines = gen_multiplex([log1,log2])
```

- There is no way to poll a generator. So, how do you do this?

Multiplexing Generators

```
def gen_multiplex(genlist):
    item_q = Queue.Queue()
    def run_one(source):
        for item in source: item_q.put(item)

    def run_all():
        thrlist = []
        for source in genlist:
            t = threading.Thread(target=run_one,args=(source,))
            t.start()
            thrlist.append(t)
        for t in thrlist: t.join()
        item_q.put(StopIteration)

    threading.Thread(target=run_all).start()
    while True:
        item = item_q.get()
        if item is StopIteration: return
        yield item
```

Multiplexing Generators

```
def gen_multiplex(genlist):
    item_q = Queue.Queue()
    def run_one(source):
        for item in source: item_q.put(item)

    def run_all():
        thrlist = []
        for source in genlist:
            t = threading.Thread(ta
            t.start()
            thrlist.append(t)
        for t in thrlist: t.join()
        item_q.put(StopIteration)

    threading.Thread(target=run_all).start()
    while True:
        item = item_q.get()
        if item is StopIteration: return
        yield item
```

Each generator runs in a thread and drops items onto a queue

Multiplexing Generators

```
def gen_multiplex(genlist):
    item_q = Queue.Queue()
    def run_one(source):
        for item in source: item_q.put(item)

    def run_all():
        thrlist = []
        for source in genlist:
            t = threading.Thread(target=run_one,args=(source,))
            t.start()
            thrlist.append(t)
        for t in thrlist: t.join()
        item_q.put(StopIteration)

    threading.Thread(target=run_all
    while True:
        item = item_q.get()
        if item is StopIteration: return
        yield item
```

Pull items off the queue and yield them

Multiplexing Generators

```python
def gen_multiplex(genlist):
    item_q = Queue.Queue()
    def run_one(source):
        for item in source: item_
```

Run all of the generators, wait for them to terminate, then put a sentinel on the queue (StopIteration)

```python
    def run_all():
        thrlist = []
        for source in genlist:
            t = threading.Thread(target=run_one,args=(source,))
            t.start()
            thrlist.append(t)
        for t in thrlist: t.join()
        item_q.put(StopIteration)

    threading.Thread(target=run_all).start()
    while True:
        item = item_q.get()
        if item is StopIteration: return
        yield item
```

Part 8

Various Programming Tricks (And Debugging)

Putting it all Together

- This data processing pipeline idea is powerful

- But, it's also potentially mind-boggling

- Especially when you have dozens of pipeline stages, broadcasting, multiplexing, etc.

- Let's look at a few useful tricks

Creating Generators

- Any single-argument function is easy to turn into a generator function

```
def generate(func):
    def gen_func(s):
        for item in s:
            yield func(item)
    return gen_func
```

- Example:

```
gen_sqrt = generate(math.sqrt)
for x in gen_sqrt(xrange(100)):
    print x
```

Debug Tracing

- A debugging function that will print items going through a generator

```python
def trace(source):
    for item in source:
        print item
        yield item
```

- This can easily be placed around any generator

```python
lines = follow(open("access-log"))
log   = trace(apache_log(lines))

r404  = trace(r for r in log if r['status'] == 404)
```

- Note: Might consider logging module for this

Recording the Last Item

- Store the last item generated in the generator

```python
class storelast(object):
    def __init__(self,source):
        self.source = source
    def next(self):
        item = self.source.next()
        self.last = item
        return item
    def __iter__(self):
        return self
```

- This can be easily wrapped around a generator

```python
lines = storelast(follow(open("access-log")))
log   = apache_log(lines)

for r in log:
    print r
    print lines.last
```

Shutting Down

- Generators can be shut down using .close()

```
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
```

- Example:

```
lines = follow(open("access-log"))
for i,line in enumerate(lines):
    print line,
    if i == 10: lines.close()
```

Shutting Down

- In the generator, GeneratorExit is raised

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

- This allows for resource cleanup (if needed)

Ignoring Shutdown

- Question: Can you ignore GeneratorExit?

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

- Answer: No. You'll get a RuntimeError

Shutdown and Threads

- Question : Can a thread shutdown a generator running in a different thread?

```
lines = follow(open("foo/test.log"))

def sleep_and_close(s):
    time.sleep(s)
    lines.close()
threading.Thread(target=sleep_and_close,args=(30,)).start()

for line in lines:
    print line,
```

Shutdown and Threads

- Separate threads <u>can not</u> call .close()

- Output:

```
Exception in thread Thread-1:
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/2.5/
lib/python2.5/threading.py", line 460, in __bootstrap
    self.run()
  File "/Library/Frameworks/Python.framework/Versions/2.5/
lib/python2.5/threading.py", line 440, in run
    self.__target(*self.__args, **self.__kwargs)
  File "genfollow.py", line 31, in sleep_and_close
    lines.close()
ValueError: generator already executing
```

Shutdown and Signals

- Can you shutdown a generator with a signal?

```
import signal
def sigusr1(signo,frame):
    print "Closing it down"
    lines.close()

signal.signal(signal.SIGUSR1,sigusr1)

lines = follow(open("access-log"))
for line in lines:
    print line,
```

- From the command line

```
% kill -USR1 pid
```

Shutdown and Signals

- This also fails:

```
Traceback (most recent call last):
  File "genfollow.py", line 35, in <module>
    for line in lines:
  File "genfollow.py", line 8, in follow
    time.sleep(0.1)
  File "genfollow.py", line 30, in sigusr1
    lines.close()
ValueError: generator already executing
```

- Sigh.

Shutdown

- The only way to externally shutdown a generator would be to instrument with a flag or some kind of check

```
def follow(thefile,shutdown=None):
    thefile.seek(0,2)
    while True:
        if shutdown and shutdown.isSet(): break
        line = thefile.readline()
        if not line:
            time.sleep(0.1)
            continue
        yield line
```

Shutdown

- Example:

```
import threading,signal

shutdown = threading.Event()
def sigusr1(signo,frame):
    print "Closing it down"
    shutdown.set()
signal.signal(signal.SIGUSR1,sigusr1)

lines = follow(open("access-log"),shutdown)
for line in lines:
    print line,
```

Part 9

Co-routines

The Final Frontier

- In Python 2.5, generators picked up the ability to receive values using .send()

```
def recv_count():
    try:
        while True:
            n = (yield)      # Yield expression
            print "T-minus", n
    except GeneratorExit:
        print "Kaboom!"
```

- Think of this function as receiving values rather than generating them

Example Use

- Using a receiver

```
>>> r = recv_count()
>>> r.next()  ◄·············································  Note: must call .next() here
>>> for i in range(5,0,-1):
...       r.send(i)
...
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>> r.close()
Kaboom!
>>>
```

Co-routines

- This form of a generator is a "co-routine"

- Also sometimes called a "reverse-generator"

- Python books (mine included) do a pretty poor job of explaining how co-routines are supposed to be used

- I like to think of them as "receivers" or "consumer". They receive values sent to them.

Setting up a Coroutine

- To get a co-routine to run properly, you have to ping it with a .next() operation first

```
def recv_count():
    try:
        while True:
            n = (yield)      # Yield expression
            print "T-minus", n
    except GeneratorExit:
        print "Kaboom!"
```

- Example:

```
r = recv_count()
r.next()
```

- This advances it to the first yield--where it will receive its first value

@consumer decorator

- The .next() bit can be handled via decoration

```python
def consumer(func):
    def start(*args,**kwargs):
        c = func(*args,**kwargs)
        c.next()
        return c
    return start
```

- Example:

```python
@consumer
def recv_count():
    try:
        while True:
            n = (yield)      # Yield expression
            print "T-minus", n
    except GeneratorExit:
        print "Kaboom!"
```
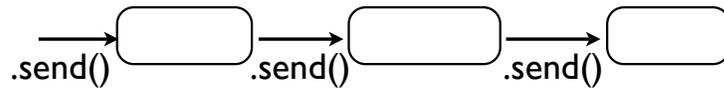

@consumer decorator

- Using the decorated version

```python
>>> r = recv_count()
>>> for i in range(5,0,-1):
...     r.send(i)
...
T-minus 5
T-minus 4
T-minus 3
T-minus 2
T-minus 1
>>> r.close()
Kaboom!
>>>
```

- Don't need the extra .next() step here

Coroutine Pipelines

- Co-routines also set up a processing pipeline

- Instead of being defining by iteration, it's defining by pushing values into the pipeline using .send()

- We already saw some of this with broadcasting

Broadcasting (Reprise)

- Consume a generator and send items to a set of consumers

```
def broadcast(source, consumers):
    for item in source:
        for c in consumers:
            c.send(item)
```

- Notice that send() operation there

- The consumers could be co-routines

Example

```
@consumer
def find_404():
    while True:
         r = (yield)
         if r['status'] == 404:
             print r['status'],r['datetime'],r['request']

@consumer
def bytes_transferred():
    total = 0
    while True:
         r = (yield)
         total += r['bytes']
         print "Total bytes", total

lines = follow(open("access-log"))
log   = apache_log(lines)
broadcast(log,[find_404(),bytes_transferred()])
```

Discussion

- In last example, multiple consumers

- However, there were no threads

- Further exploration along these lines can take you into co-operative multitasking, concurrent programming without using threads

- That's an entirely different tutorial!

Wrap Up

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

- Can create components that plugged into the pipeline as reusable pieces

- Can extend the pipeline idea in many directions (networking, threads, co-routines)

Code Reuse

- I like the way that code gets reused with generators

- Small components that just process a data stream

- Personally, I think this is much easier than what you commonly see with OO patterns

Example

- SocketServer Module (Strategy Pattern)

```
import SocketServer
class HelloHandler(SocketServer.BaseRequestHandler):
    def handle(self):
        self.request.sendall("Hello World\n")

serv = SocketServer.TCPServer(("",8000),HelloHandler)
serv.serve_forever()
```

- My generator version

```
for c,a in receive_connections(("",8000)):
    c.send("Hello World\n")
    c.close()
```

Pitfalls

- I don't think many programmers really understand generators yet

- Springing this on the uninitiated might cause their head to explode

- Error handling is really tricky because you have lots of components chained together

- Need to pay careful attention to debugging, reliability, and other issues.

Thanks!

- I hope you got some new ideas from this class

- Please feel free to contact me

http://www.dabeaz.com