# Generator Tricks For Systems Programmers

(Version 3)

David Beazley
http://www.dabeaz.com

Originally Presented at PyCon 2008
Updated: October, 2018

# Introduction 3.0

This tutorial was originally presented at PyCon'2008 (Chicago). Although the original tutorial was written for Python 2.5, the underlying concepts remain current. This revised version of the tutorial has been updated to Python 3.7. Enjoy!

-- Dave Beazley (October 2018)

If you like this tutorial, come to Chicago and take an advanced programming class!

https://www.dabeaz.com/courses.html

# Support Files

- Files used in this tutorial are available here:

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

- Go there to follow along with the examples

# An Introduction

- Generators are cool!
- But what are they?
- And what are they good for?
- That's what this tutorial is about

# Our Goal

- Explore 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

# 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.
- Let's have a discussion

# Part 1

## Introduction to Iterators and Generators

# 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, end=' ')
...
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 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
>>>
```

# Iterating over a String

- If you iterate over a string, you get characters

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

# Iterating over a File

- If you iterate over a file you get lines

```
>>> for line in open("real.txt"):
...     print(line, end='')
...
        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 operations consume an "iterable" object
- Reductions:

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

- Constructors

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

- Various operators

```
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() is said to be "iterable."

# Supporting Iteration

- User-defined objects can support iteration
- Example: Counting down...

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

- To do this, you have to make the object implement __iter__() and __next__()

# Supporting Iteration

- Sample implementation

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

class 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
```

# Iteration Example

- Example use:

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

- Instead of returning a value, you generate 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 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

# Generator Functions

- The function only executes on __next__()

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


- 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 a much 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 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)

# 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, end=' ')
...
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 condition)
```

- What it means

```
for i in s:
    if condition:
       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

```
bytes_sent = 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 bytes_sent != '-':
   bytes_sent = int(bytes_sent)
```

# A Non-Generator Soln

- Just use a simple for-loop

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

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



Example File: nongenlog.py

# A Generator Solution

- Let's use some generator expressions

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

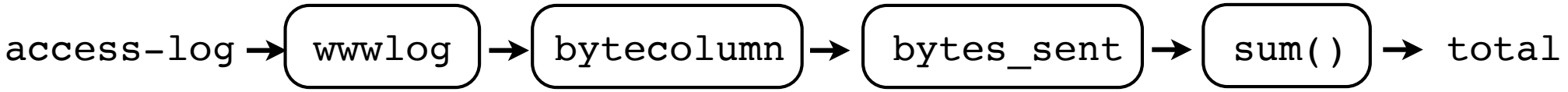
- Whoa! That's different!
  - Less code
  - A completely different programming style



Example File: genlog.py

# Generators as a Pipeline

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




- Each step is defined by iteration/generation

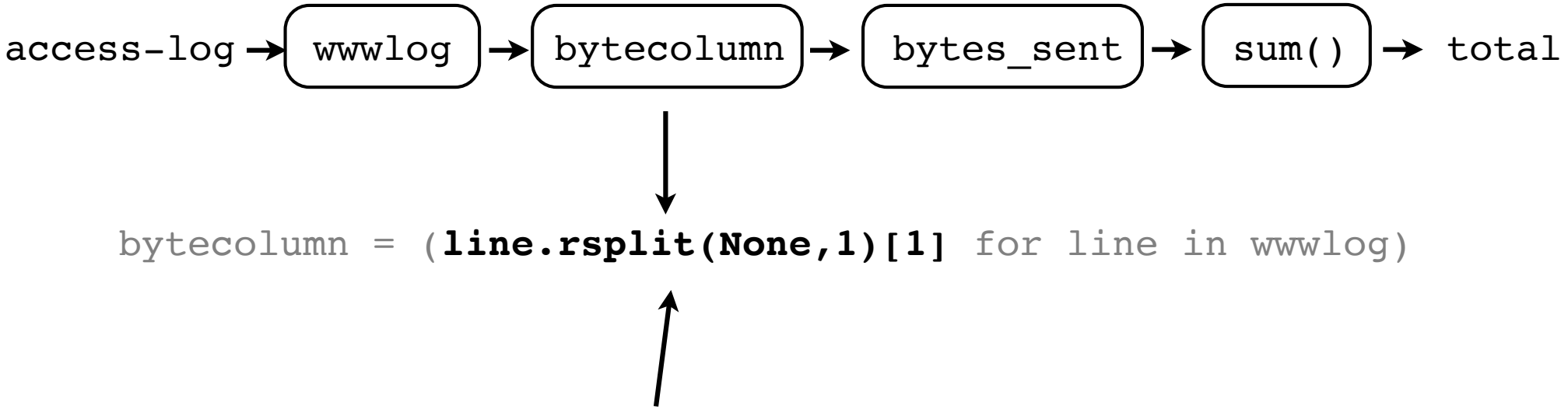
```
with open("access-log") as wwwlog:
    bytecolumn = (line.rsplit(None,1)[1] for line in wwwlog)
    bytes_sent = (int(x) for x in bytecolumn if x != '-')
    print("Total", sum(bytes_sent))
```



Example File: genlog.py

# Being Declarative

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



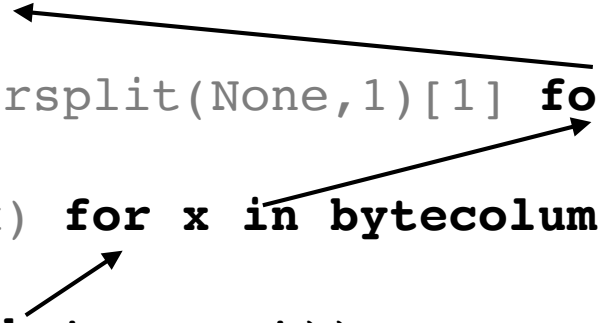
# Being Declarative

- 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
- It's 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

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



- The calculation is being driven by the last step
- The sum() function is consuming values being pulled 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
```

(Note: Use the script 'python3 makebig.py 2000' to create this file).

# Performance Contest

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

Time

18.6

---

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

Time

16.7

# Commentary

- Not only was it not slow, it was 10% 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

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

Time

16.7

---

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

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

Time

70.5

# 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
```

# Path.rglob()

- A useful way to search the filesystem

```
from pathlib import Path

for filename in Path('/').rglob('*.py'):
    print(filename)
```

- Guess what?  It uses generators!

```
>>> from pathlib import Path
>>> Path('/').rglob('*.py')
<generator object Path.rglob at 0x10e3e0b88>
>>>
```

- So, you could build processing pipelines from it



Example File: genfind.py

# A File Opener

- Open a sequence of paths

```
import gzip, bz2
def gen_open(paths):
    for path in paths:
        if path.suffix == '.gz':
            yield gzip.open(path, 'rt')
        elif path.suffix == '.bz2':
            yield bz2.open(path, 'rt')
        else:
            yield open(path, 'rt')
```

- This is interesting.... it takes a sequence of paths as input and yields a sequence of open file objects



Example File: genopen.py

# cat

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

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

OR

```
def gen_cat(sources):
    for src in sources:
        yield from src
```

- Example:

```
lognames = Path('/usr/www').rglob("access-log*")
logfiles = gen_open(lognames)
loglines = gen_cat(logfiles)
```



Example File: gencat.py

# Aside: yield from

- 'yield from' can be used to delegate iteration

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

def countup(stop):
    n = 1
    while n < stop:
        yield n
        n += 1
```

```
def up_and_down(n):
    yield from countup(n)
    yield from countdown(n)
```

```
>>> for x in up_and_down(3):
...     print(x)
...
1
2
3
2
1
>>>
```

# grep

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

```
import re

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

- Example:

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



Example File: gengrep.py

# Example

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

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

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

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



Example File: bytesgen.py

# 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
```

*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')
```



Example File: retuple.py

# 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

# 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'}
```



Example File: redict.py

# 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)
```



Example File: fieldmap.py

# 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

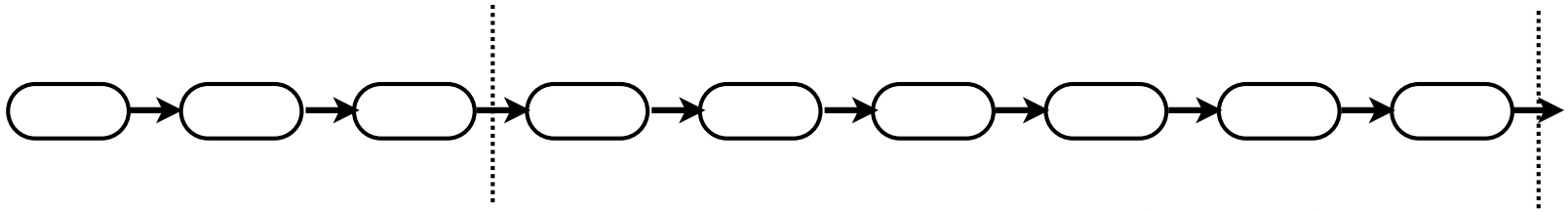
```
from pathlib import Path

lognames = Path('www').rglob('access-log*')
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)
```



Example File: fieldmap.py

# Getting Organized

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



generate lines from a set of files in a directory

Parse a sequence of lines from Apache server logs into a sequence of dictionaries

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

# Packaging

- Example : multiple pipeline stages inside a function

```
from pathlib import Path

def lines_from_dir(filepat, dirname):
    names = Path(dirname).rglob(filepat)
    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



Example File: linesdir.py

# 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
```



Example File: apachelog.py

# 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



Example File: apachelog.py

# 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
- Is it simpler to have a bunch of components that all operate on dictionaries or to have components that require inputs/outputs to be different kinds of user-defined instances?

# A Query Language

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

```
stat404 = { 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'])
```



Example Files: query404.py, largefiles.py

# 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 = { 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')
```

```
Example Files: largest.py, hosts.py, downloads.py
```

# A Query Language

- Find out who has been hitting robots.txt

```
addrs = { 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)
```



Example File: robots.py

# Some Thoughts

- I like the idea of using generator expressions as a pipeline query language
- You can write simple filters, extract data, etc.
- If 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
import os

def follow(thefile):
    thefile.seek(0, os.SEEK_END) # End-of-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 File: follow.py

# Example

- Using our follow function

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

for line in loglines:
    print(line, end='')
```

- 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'])
```



Example File: realtime404.py

# 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

# 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))
```

# A Thought

- There is no rule that says you have to generate pipeline data from a file.
- Or that the input data has to be a string
- Or that it has to be turned into a dictionary
- Remember: All Python objects are "first-class"
- Which means that all objects are fair-game for use in a generator pipeline

# 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(b"Hello World\n")
    c.close()
```



Example File: genreceive.py

# 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)
```



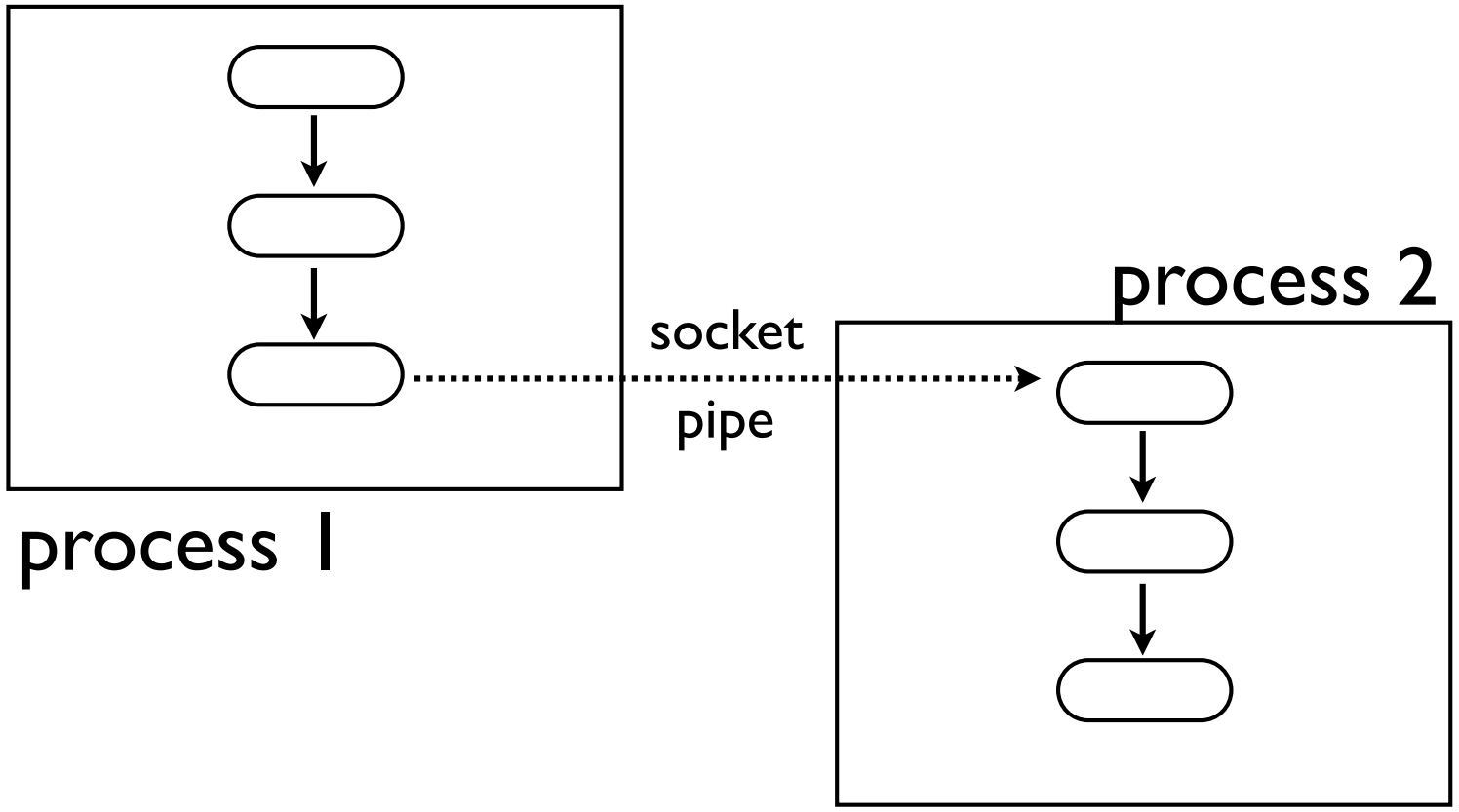
Example File: genmessages.py

# Part 7

## Extending the Pipeline

# Multiple Processes

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



# Pickler/Unpickler

- Turn a generated sequence into pickled objects

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

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

- Now, attach these to a pipe or socket



Example File: genpickle.py

# 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('rb')):
        yield item
    c.close()
```



Example Files: sendto.py, receivefrom.py

# Example Use

- Example: Read log lines and parse into records

```
# producer.py

lines = follow(open("access-log"))
log   = apache_log(lines)
sendto(log,("",15000))
```

- Example: Pick up the log on another machine

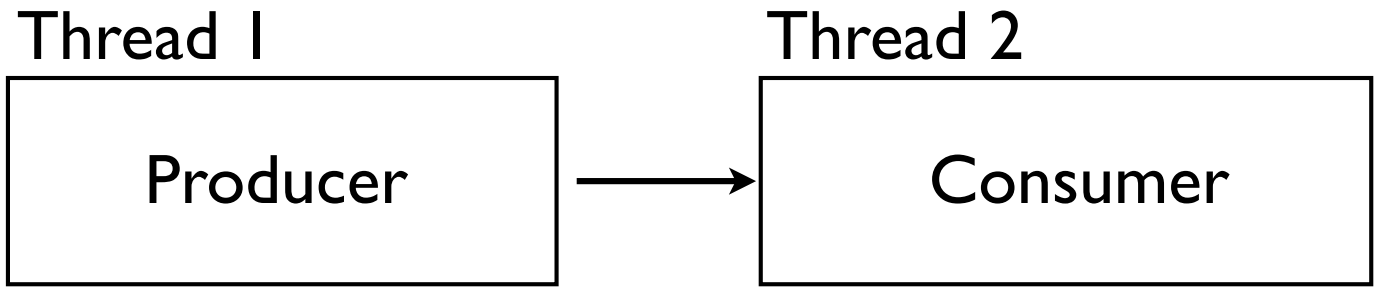
```
# consumer.py
for r in receivefrom(("",15000)):
    print(r)
```



Example Files: sendto.py, receivefrom.py

# Generators and Threads

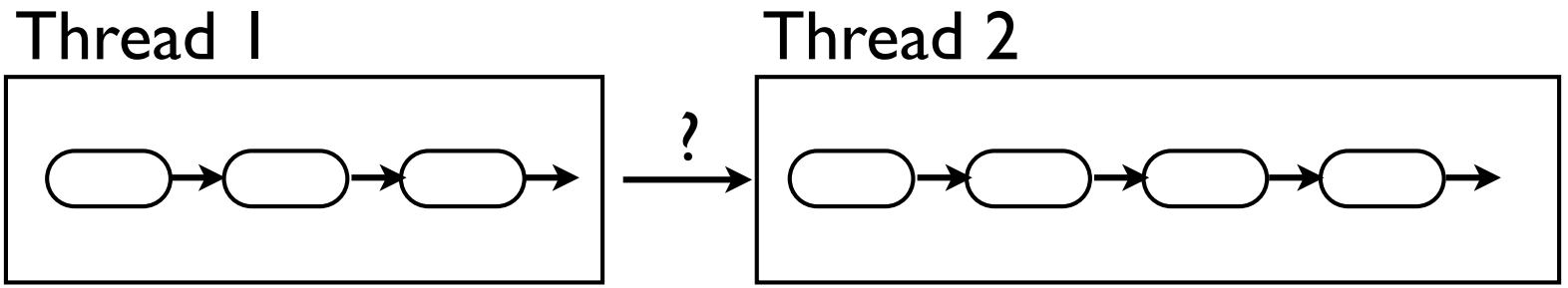
- Processing pipelines sometimes come up in the context of thread programming
- Producer/consumer problems




- Question: Can generator pipelines be integrated with thread programming?

# Multiple Threads

- For example, can a generator pipeline span multiple threads?




- Yes, if you connect them with a Queue object

# Generators and Queues

- Feed a generated sequence into a queue

```
# genqueue.py
def sendto_queue(source, thequeue):
    for item in source:
        thequeue.put(item)
    thequeue.put(StopIteration)
```

- Generate items received on a queue

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

- Note: Using StopIteration as a sentinel



Example File: genqueue.py

# Thread Example

- Here is a consumer function

```
# A consumer.   Prints out 404 records.
def print_r404(log_q):
    log = genfrom_queue(log_q)
    r404 = (r for r in log if r['status'] == 404)
    for r in r404:
        print(r['host'],r['datetime'],r['request'])
```

- This function will be launched in its own thread
- Using a Queue object as the input source

# Thread Example

- Launching the consumer

```
import threading, queue
log_q = queue.Queue()
r404_thr = threading.Thread(target=print_r404,
                            args=(log_q,))
r404_thr.start()
```

- Code that feeds the consumer

```
lines = follow(open("access-log"))
log   = apache_log(lines)
sendto_queue(log, log_q)
```

# Part 8

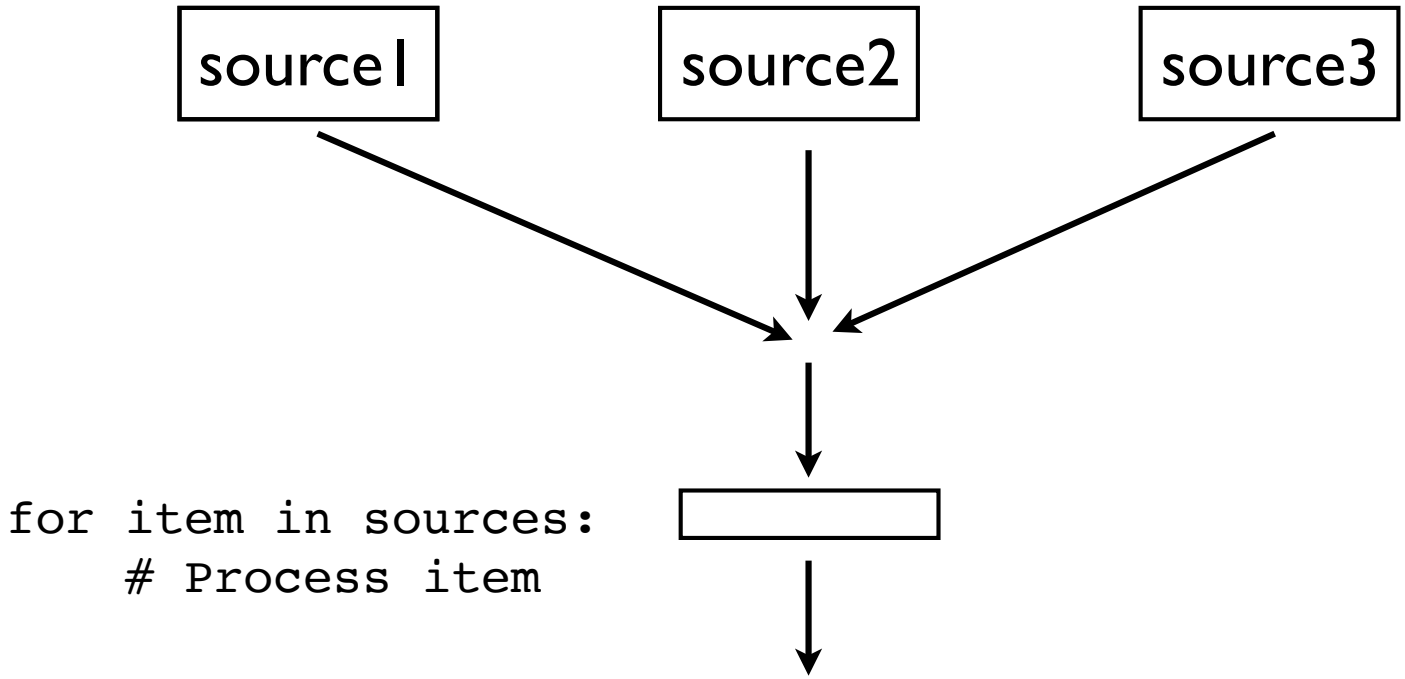
## Advanced Data Routing

# The Story So Far

- You can use generators to set up pipelines
- You can extend the pipeline over the network
- You can extend it between threads
- However, it's still just a pipeline (there is one input and one output).
- Can you do more than that?

# Multiple Sources

- Can a processing pipeline be fed by multiple sources---for example, multiple generators?



# Concatenation

- Concatenate one source after another (reprise)

```
def gen_cat(sources):
    for src in sources:
        yield from src
```

- This generates one big sequence
- Consumes each generator one at a time
- But only works if generators terminate
- So, you wouldn't use this for real-time streams

# 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 any input ends

# Multiplexing

- Feed a pipeline from multiple generators in real-time--producing values as they arrive
- Example use

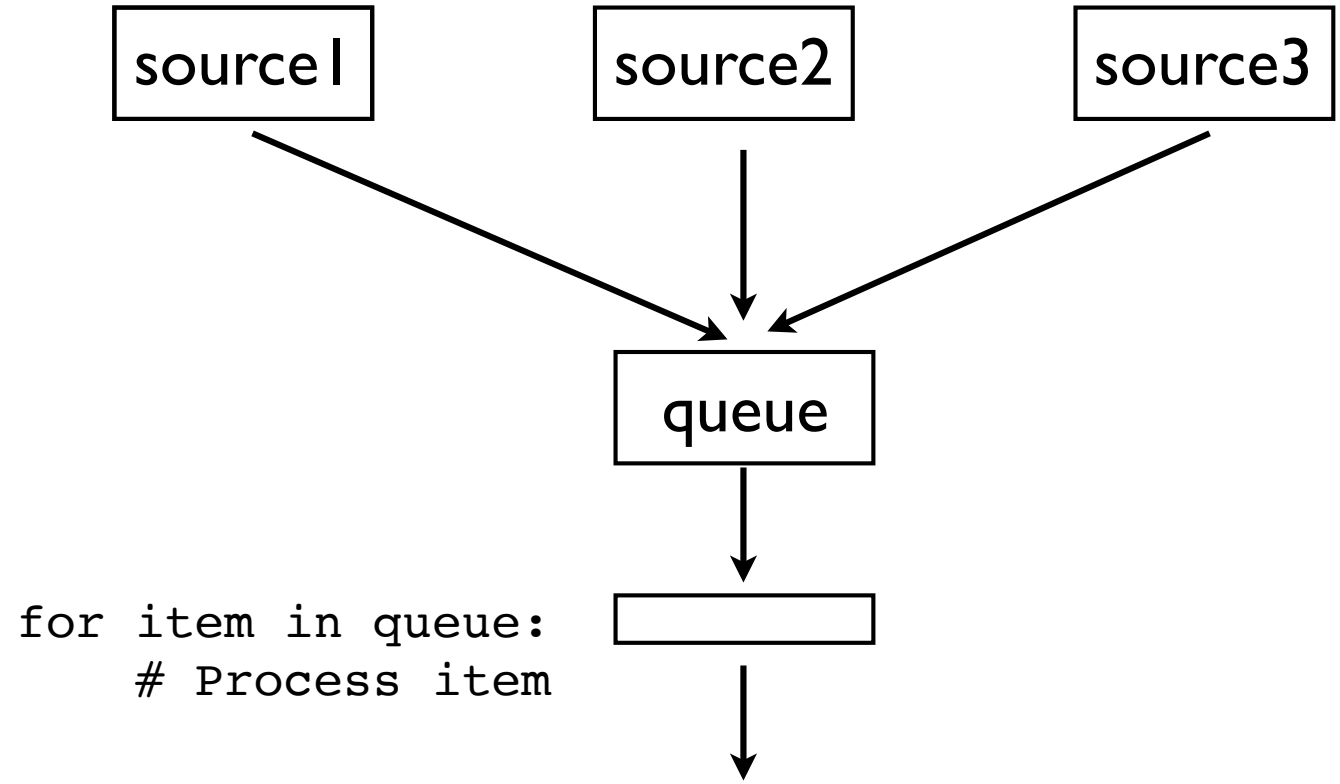
```
log1 = follow(open("foo/access-log"))
log2 = follow(open("bar/access-log"))

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

- There is no way to poll a generator
- And only one for-loop executes at a time

# Multiplexing

- You can multiplex if you use threads and you use the tools we've developed so far

- Idea :



# Multiplexing

```
# genmulti.py

import threading, queue
from genqueue import genfrom_queue
from gencat import gen_cat

def multiplex(sources):
    in_q = queue.Queue()
    consumers = []
    for src in sources:
        thr = threading.Thread(target=sendto_queue,
                               args=(src, in_q))
        thr.start()
        consumers.append(genfrom_queue(in_q))
    return gen_cat(consumers)
```

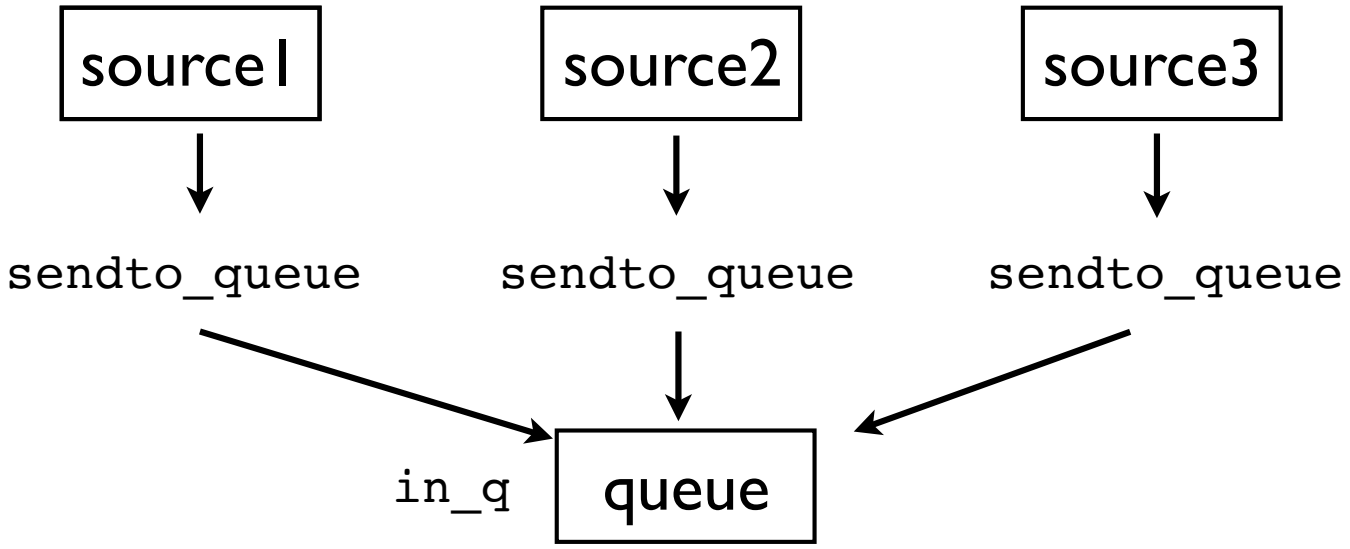
- Note: This is the trickiest example so far...



Example File: genmulti.py

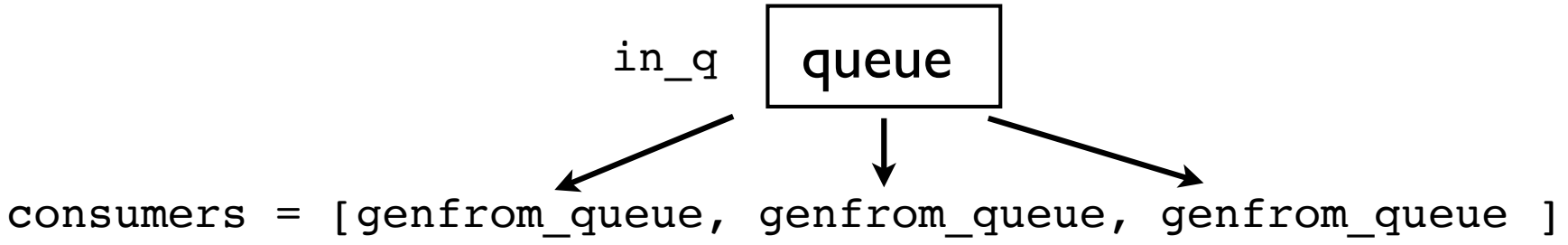
# Multiplexing

- Each input source is wrapped by a thread which runs the generator and dumps the items into a shared queue



# Multiplexing

- For each source, we create a consumer of queue data



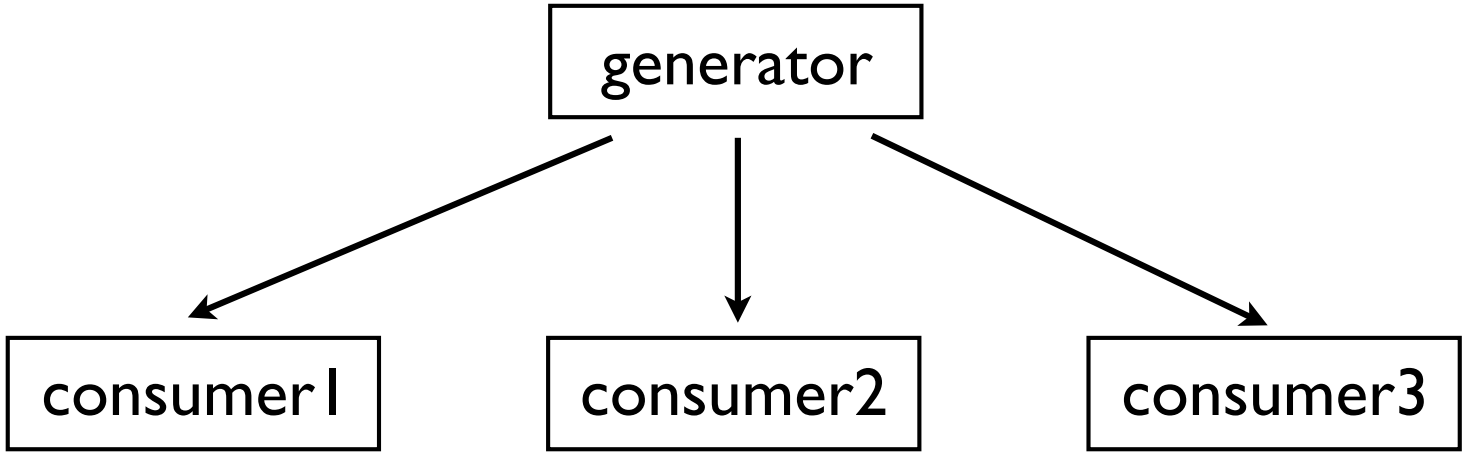

- Now, just concatenate the consumers together

```
get_cat(consumers)
```

- Each time a producer terminates, we move to the next consumer (until there are no more)

# Broadcasting

- Can you broadcast to multiple consumers?



# Broadcasting

- Consume a generator and send to consumers

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

- It works, but now the control-flow is unusual
- The broadcast loop is what runs the program
- Consumers run by having items sent to them

# 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])
```

# 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 across the network



Example File: netsend.py

# Network Consumer

- Example Usage:

```
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...



Example File: netsend.py

# Commentary

- Once you start broadcasting, consumers can't follow the same programming model as before
- Only one for-loop can run the pipeline.
- However, you can feed an existing pipeline if you're willing to run it in a different thread or in a different process

# Consumer Thread

- Example: Routing items to a separate thread

```
import queue, threading
from genqueue import genfrom_queue

class ConsumerThread(threading.Thread):
    def __init__(self,target):
         threading. Thread.__init__(self)
         self.setDaemon(True)
         self.in_q   = queue.Queue()
         self.target = target
    def send(self,item):
         self.in_q.put(item)
    def run(self):
        self.target(genfrom_queue(self.in_q))
```



Example File: thrsend.py

# 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
```



Example File: thrsend.py

# Part 9

## 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(range(100)):
    print(x)
```

# Debug Tracing

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

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

- This can easily be placed around any generator

```
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

```
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

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

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



Example File: storelast.py

# Shutting Down

- Generators can be shut down using .close()

```
import time
def follow(thefile):
    thefile.seek(0, os.SEEK_END) # End of 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, end='')
    if i == 10:
        lines.close()
```



Example File: genshutdown.py

# Shutting Down

- In the generator, GeneratorExit is raised

```
import time
def follow(thefile):
    thefile.seek(0, os.SEEK_END)
    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, os.SEEK_END)
    while True:
        try:
            line = thefile.readline()
            if not line:
                time.sleep(0.1)    # Sleep briefly
                continue
            yield line
        except GeneratorExit:       # Note: inside while
            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, end='')
```

# Shutdown and Threads

- Separate threads can not call .close()
- Output:

```
Exception in thread Thread-1:
Traceback (most recent call last):
  ...
  File "genfollow.py", line 31, in sleep_and_close
    lines.close()
ValueError: generator already executing
```

- Similarly, don't call .close() from signal handlers

# 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, os.SEEK_END)
    while True:
        if shutdown and shutdown.is_set():
            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, end='')
```

# Part 10

## Parsing and Printing

# Incremental Parsing

- Generators are a useful way to incrementally parse almost any kind of data

```
# 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 function sweeps through a file and generates a sequence of unpacked records

# Incremental Parsing

- Example:

```
from genrecord import gen_records

f = open("stockdata.bin","rb")
for name, shares, price in gen_records("<8sif",f):
     # Process data
     ...
```

- Tip : Look at xml.etree.ElementTree.iterparse for a neat way to incrementally process large XML documents using generators

# yield as print

- Generator functions can use yield like a print statement
- Example:

```
def print_count(n):
    yield "Hello World\n"
    yield "\n"
    yield "Look at me count to %d\n" % n
    for i in range(n):
        yield "   %d\n" % i
    yield "I'm done!\n"
```

- This is useful if you're producing I/O output, but you want flexibility in how it gets handled

# yield as print

- Examples of processing the output stream:

```
# Generate the output
out = print_count(10)

# Turn it into one big string
out_str = "".join(out)

# Write it to a file
f = open("out.txt","w")
for chunk in out:
    f.write(chunk)

# Send it across a network socket
for chunk in out:
    s.sendall(chunk)
```

# yield as print

- This technique of producing output leaves the exact output method unspecified
- So, the code is not hardwired to use files, sockets, or any other specific kind of output
- There is an interesting code-reuse element
- One use of this : WSGI applications

# Part 11

## Co-routines

# The Final Frontier

- Generators can also 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 File: recvcount.py

# Example Use

- Using a receiver

```
>>> r = recv_count()
>>> r.send(None)
>>> 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!
>>>
```

Note: must call .send(None) here

# 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 .send(None) 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.send(None)
```

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

# @consumer decorator

- The initialization can be handled via decoration

```
def consumer(func):
    def start(*args,**kwargs):
        c = func(*args,**kwargs)
        c.send(None)
        return c
    return start
```

- Example:

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

Example File: consumer.py

# @consumer decorator

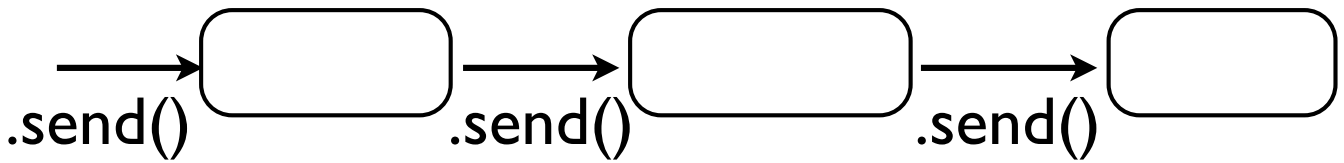
- Using the decorated version

```
>>> 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 .send(None) 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()])
```



Example File: logcoroutine.py

# 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
- But 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(b"Hello World\n")

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

- A generator version

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

# Pitfalls

- Springing this programming style on the uninitiated might cause their head to explode
- Error handling is 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

  Web: http://www.dabeaz.com

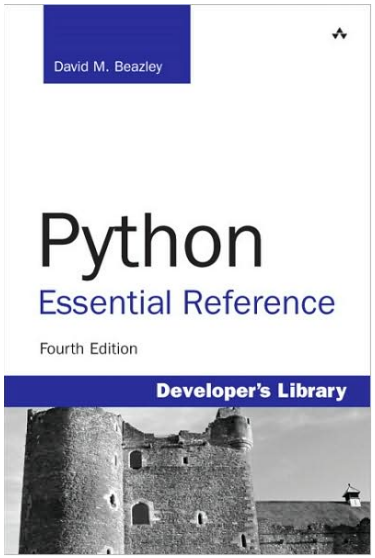
  Twitter: @dabeaz

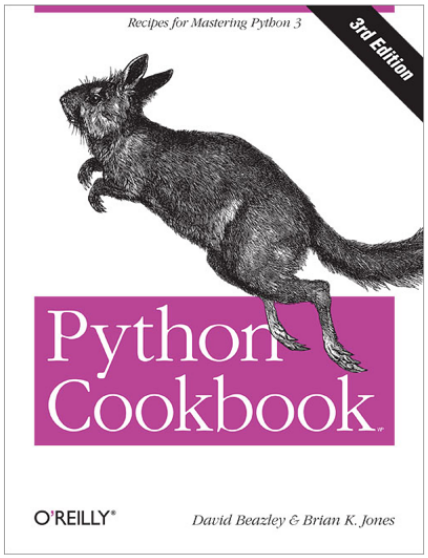
# Take a Course!

Go beyond the notes and self-study! Since 2007, David Beazley has taught in-person computer science and programming courses for developers who want to take their skills to the next level.  Expand your thinking.

**http://www.dabeaz.com/courses.html**

# Books/Video





Python Programming Language

★★★★★ 127 reviews

by David Beazley

Publisher: Addison-Wesley Professional

*Release Date: August 2016*

ISBN: 9780134217314

**https://www.safaribooksonline.com**