

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" software
but back in the Good Old Days,
when the term "software" sounded funny
and Real Computers were made out of drums and vacuum tubes
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 generate a series of values (using the yield statement)

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 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,
...
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 be 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)

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

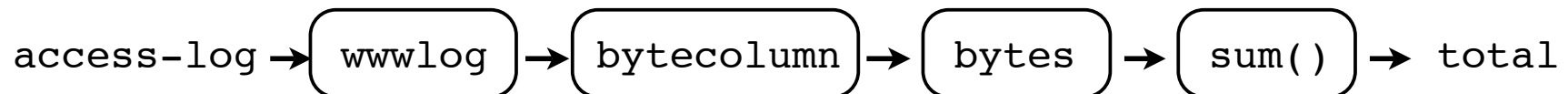
```
wwwlog      = open("access-log")
bytecolumn  = (line.rsplit(None,1)[1] for line in wwwlog)
bytes       = (int(x) for x in bytecolumn if x != '-')

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



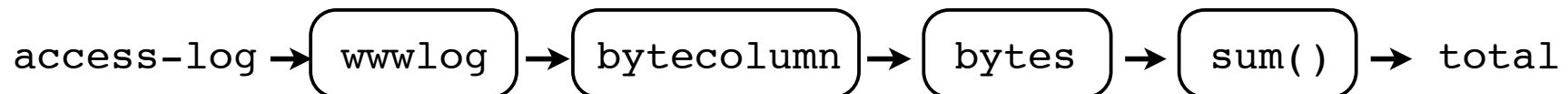
- Each step is defined by iteration/generation

```
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 will be applied to the entire input stream



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

This operation gets applied to
every line of the log file

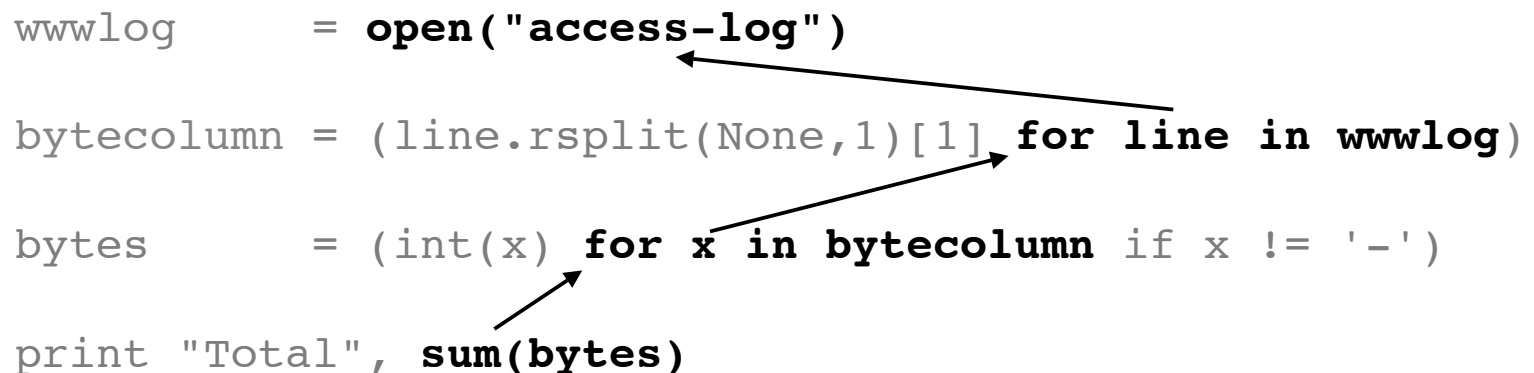
Being Declarative

- 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
```

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

Time

37.33

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

Programming Problem

You have hundreds of web server logs scattered across various directories. In addition, 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/")
```

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