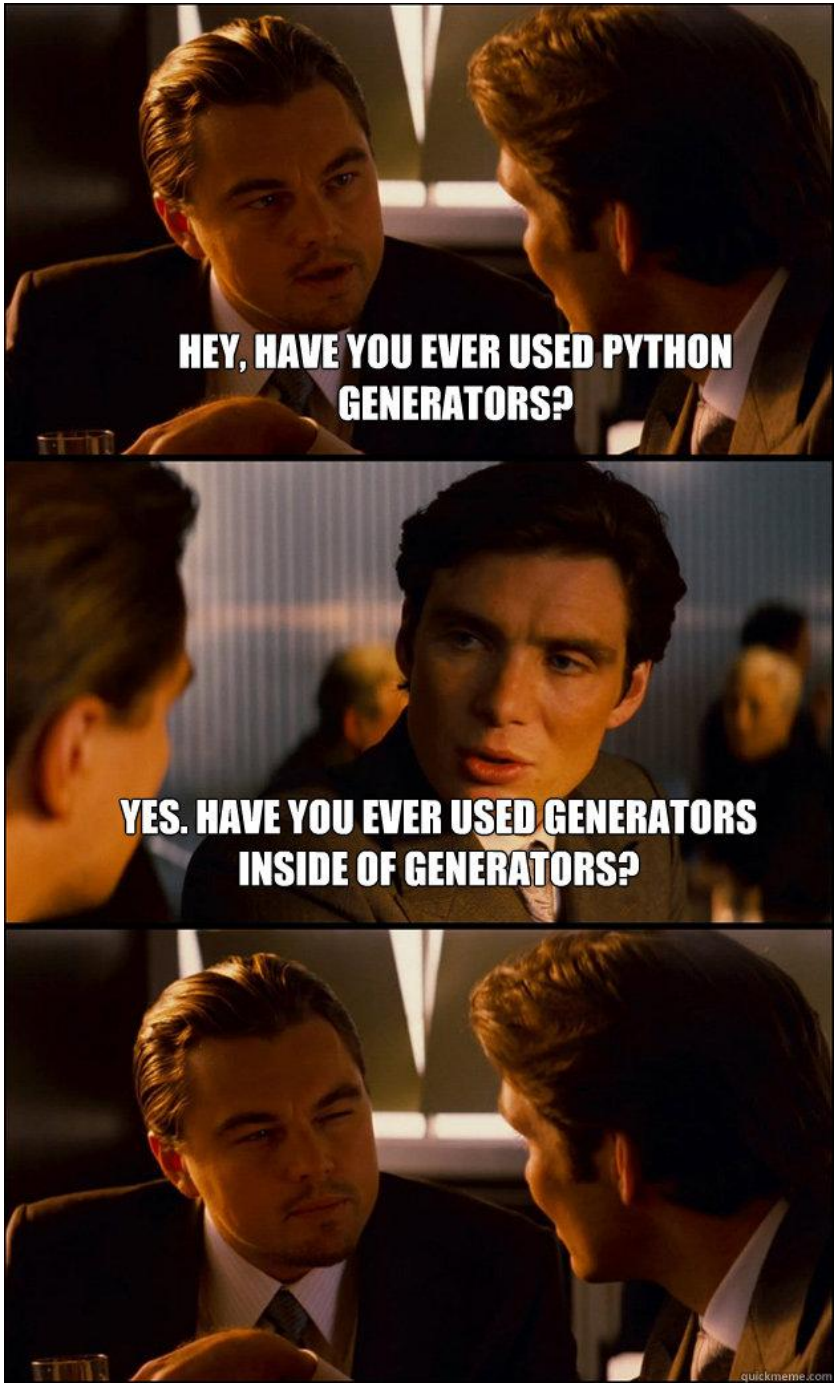
Python Generator Pipelines

Brett Langdon @brett_langdon


HEY, HAVE YOU EVER USED PYTHON GENERATORS?
YES. HAVE YOU EVER USED GENERATORS INSIDE OF GENERATORS?
quickmeme.com

Who Am I?

- Brett Langdon
- Software Engineer - Magnetic
- http://www.magnetic.com
- http://brett.is
- brett@blangdon.com
- @brett_langdon
- github.com/brettlangdon
- geekli.st/brett_langdon

Supporting Code

- github.com/PythonBuffalo/generator-pipelines.git

This talk is a rip off

- Python Generator Hacking
- David Beazley
- http://www.slideshare.net/dabeaz/python-generator-hacking
- Google: "python generator hacking"

What I Expect

- Basic Python syntax
- Nothing else

Summary

- Iterators in Python
- Generator Functions
- List Comprehension
- Generator Expressions
- Generator Pipeline
- Practical Use Case

Iterators

```
# this is a list
numbers = [1, 2, 3, 4, 5, 6]

# I can iterate the list like this
for number in numbers:
    print number
```

Iterators, Continued

```
numbers = [1, 2, 3, 4, 5, 6]
i_numbers = iter(numbers)
print i_numbers
# <listiterator object at 0x102478050>
print i_numbers.next
# <method-wrapper 'next' of listiterator object at
0x10e5fd610>
print i_numbers.next()
# 1
for number in i_numbers:
    print number
# prints 2 -> 6
print i_numbers.next()
```

Iterators, StopIteration

```
numbers = [1, 2, 3, 4, 5,
6]
i_numbers = iter
(numbers)
while True:
   try:
      print i_numbers.
next()
   except StopIteration:
      break
```

Iterators, Non-Lists

```
mapping = {'a': 5, 'b': 10, 'c':
12}
for key in mapping:
   print key
# prints 'a' 'b' 'c'
name = 'Brett'
for char in name:
   print char
# prints 'B', 'r', 'e', 't', 't'
names = ('Brett', 'Jim', 'Dan')
for name in names:
   print name
# prints 'Brett', 'Jim', 'Dan'
```

Generator Functions

- PEP 255 - http://www.python.org/dev/peps/pep-0255/

```
def my_generator():
    yield 'Hello Generators'
gen = my_generator()
print gen
# <generator object my_generator at 0x10f7f8320>
gen.next()
# 'Hello Generators'
gen.next()
# throws StopIteration
```

Generator, Multi. Yield

```
def my_generator():
    yield 'Hello'
    yield 'Generators'
for word in
my_generator():
    print word
# prints 'Hello' 'Generators'
```

Generator, Multi. Yield

```
def my_generator():
    print 'gen: Hello'
    yield 'Hello'
    print 'gen: Generators'
    yield 'Generators'
gen = my_generator()
print gen.next()
# gen: Hello
# Hello
print gen.next()
# gen: Generators
# Generators
```

Generator, Parameters

```
def lower_words(words):
    for word in words:
        yield word.lower()

words = ['HERE', 'ARE', 'SOME', 'ANGRY', 'WORDS']
for word in lower_words(words):
    print word
```

Así... ¿Por qué?

- Build functions that act as iterators
- Only yield when data is needed
  - Reduces memory
- Don't need to know what we are yielding

Generator vs Iterator

```
fp = open('./my-badass.file')
for line in fp.readlines():
   print line

fp = open('./my-badass.file')
for line in fp:
   print line
```

List Comprehension

- PEP 202 - http://www.python.org/dev/peps/pep-0202/

```
words = ['LOOK', 'MA', 'I', 'AM', 'SHOUTING']
lowered = []
for word in words:
   lowered.append(word.lower())
for word in lowered:
   print word
# List Comprehension
words = ['LOOK', 'MA', 'I', 'AM', 'SHOUTING']
lowered = [word.lower() for word in words]
for word in lowered:
  print word
```

List Comp, Continued

```
words = ['here', 'be', 'words', 'here', 'be', 'words']
unique= set([word for word in words])

numbers = [1, 2, 6, 2, 0, 3, 4, 5, 5, 7, 9]
even = [number in numbers if number % 2 == 0]
```

Generator Expression

- PEP 289 - http://www.python.org/dev/peps/pep-0289/

```
words = ['LOOK', 'MA', 'I', 'AM', 'SHOUTING']
lowered = (word.lower() for word in words)
print lowered
# <generator object <genexpr> at 0x1098d6460>
print lowered.next()
for word in lowered:
   print word
try:
   lowered.next()
except StopIteration:
   print 'none left to iterate'
```

Gen Exp, Continued

```
words = ['here', 'be', 'words', 'here', 'be', 'words']
lowered = set(word for word in words)
numbers = [1, 2, 6, 2, 0, 3, 4, 5, 5, 7, 9]
even = list(number in numbers if number % 2 == 0)
```

Generator Pipeline

```
def first_generator(numbers):
    for number in numbers:
        print 'First: %s' % number
        yield number
def second_generator(numbers):
    for number in numbers:
        print 'Second: %s' % number
        yield number
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
numbers_pipeline = second_generator(first_generator(numbers))
for number in numbers_pipeline:
    print 'Final: %s' % number
```

Gen Pipeline, Results

First: 1
Second: 1
Final: 1
First: 2
Second: 2
Final: 2
First: 3
Second: 3
Final: 3
First: 4
Second: 4
Final: 4
...

Gen Pipeline, Filtering

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
# add three
numbers_pipeline = (num + 3 for num in numbers)
# even only
numbers_pipeline = (num for num in numbers_pipeline
                    if num % 2 == 0)
# times three
numbers_pipeline = (num * 3 for num in numbers_pipeline)
for number in numbers_pipeline:
   print number
```

Real Life Use Case

- Read file in TSV format
- Remove all empty lines
- Lowercase the line
- Turn line into a dict
- Filter some lines out

Use Case, FilterPipeline

```
class FilterPipeline(object):
    def __init__(self):
        self._filters = []
    def add_filter(self, filter):
        self._filters.append(filter)
    def read(self, iterable):
        pipeline = iterable
        for filter in self._filters:
            pipeline = filter(pipeline)
        for element in pipeline:
            yield element
```

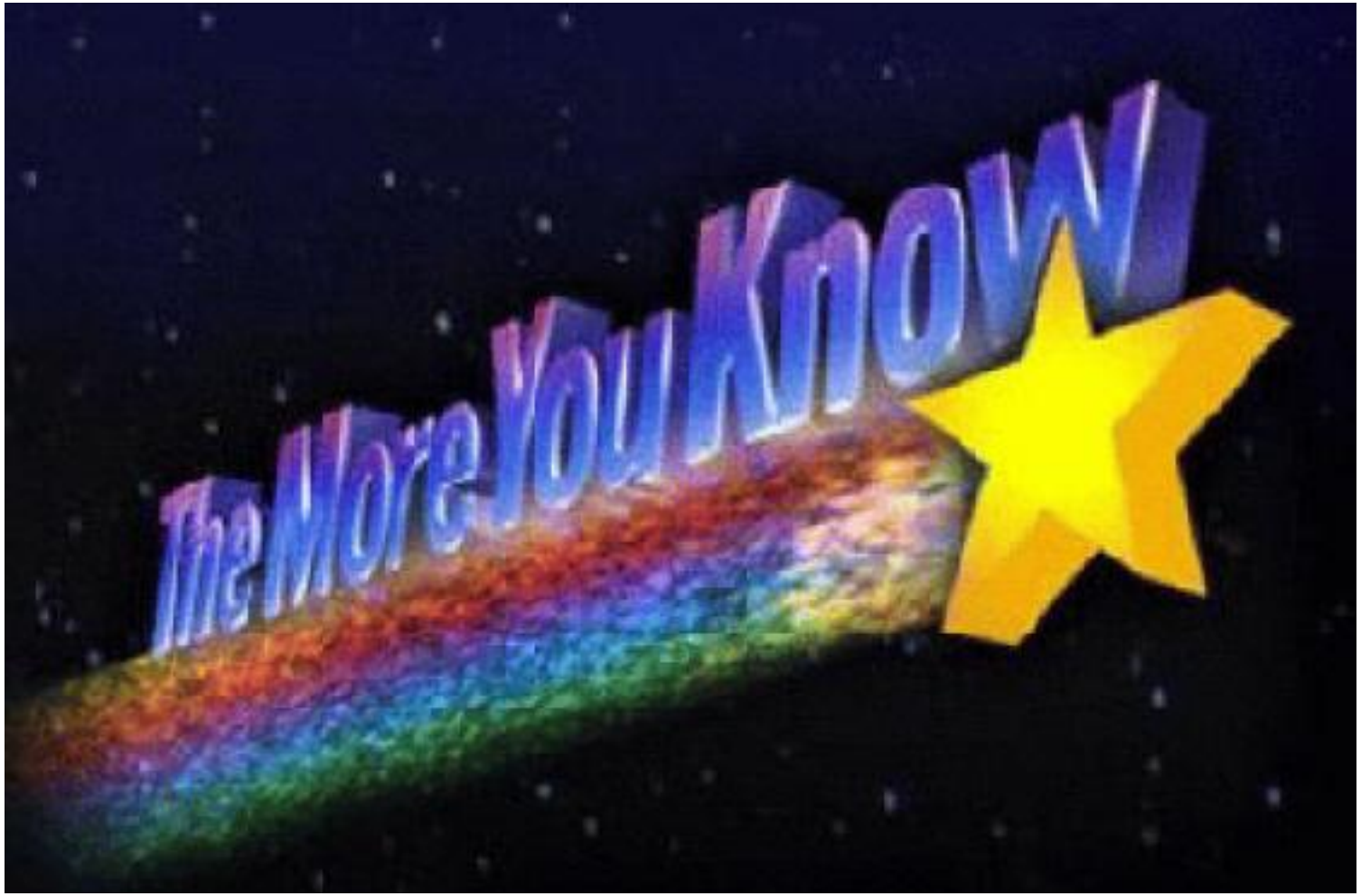
Use Case, Filters

```
def strip_lines(lines):
    for line in lines:
        yield line.strip()
def remove_empty(lines):
    for line in lines:
        if line:
            yield line
def to_lower(lines):
    for line in lines:
        yield line.lower()
```

```
def to_dict(lines):
    for line in lines:
        parts = line.split('\t')
        yield dict(part.split('=', 1)
                   for part in parts)
def remove_pastrami_subject(lines):
    for line in lines:
        if 'pastrami' not in line['subject']:
            yield line
```

Use Case, Usage

```
pipeline = FilterPipeline()
pipeline.add_filter(strip_lines)
pipeline.add_filter(remove_empty)
pipeline.add_filter(to_lower)
pipeline.add_filter(to_dict)
pipeline.add_filter(remove_pastrami_subject)
fp = open('./test.tsv')
for line in pipeline.read(fp):
    print line
```


The More You Know