Python
Generators and Generator Expressions

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

September 25, 2012

Outline

Introducing generators

Description

- **Generators** are a simple and powerful tool for creating iterators.
- They are written like regular functions but use the `yield` statement whenever they want to return data.
- Each time `next()` is called, the generator resumes where it left-off (it remembers all the data values and which statement was last executed)

Example

```
def reverse_gen(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

Introducing generators

Example

```
def reverse_gen(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

Example

```
In [1]: for char in reverse_gen('abcd'):
   ...:     print char
   ...:
d
c
b
a
In [2]:
```

Introducing generators

Example

```
In [1]: def gen():
   ...:     print "gen: 1"
   ...:     yield 1
   ...:     print "gen: 2"
   ...:     yield 2
   ...:     return
In [2]: my_gen = gen()
In [3]: my_gen
Out[3]: <generator object gen at 0x86eda54>
In [4]: my_gen.next()
gen: 1
Out[4]: 1
In [5]: my_gen.next()
gen: 2
Out[5]: 2
In [6]: my_gen.next()
---------------------------------------------------------------------------
StopIteration Traceback (most recent call last)
```

Introducing generators

Example

```
In [1]: def gen():
   ...:     yield 1
   ...:     yield 2
   ...:     return
   ...:
In [2]: my_gen = gen()
In [3]: for val in my_gen:
   ...:     print val
   ...:
1
2
In [4]: for val in gen():
   ...:     print val
   ...:
1
2
In [5]:
```

Fibonacci

Example

```
# f_n = f_{n-1} + f_{n-2}
def fib():
    fn2 = 1 # "f_{n-2}"
    fn1 = 1 # "f_{n-1}"
    while True:
        (fn1, fn2, oldfn2) = (fn1+fn2, fn1, fn2)
        yield oldfn2

genexpr = (val for val in fib())
for val in genexpr: # print 1 1 2 3 5 8 13
    if val > 20:
        break
    else:
        print val,
```

Representing infinite sequences

Example

```
# a generator for 1, 2, ...
def infseq():
    i = 1
    while True:
        yield i
        i = i+1

# This series converges to PI
def pi_series():
    sum, i, j = 0, 1.0, 1
    while(1):
        sum = sum + j/i
        yield 4*sum
        i, j  = i + 2, j*-1
```

A convenient function that return the first *n* integers

Example

```
def pi_series():
    sum, i, j = 0, 1.0, 1
    while(1):
        sum = sum + j/i
        yield 4*sum
        i, j  = i + 2, j*-1

def firstn(g, n):
    for i in range(n):
        yield g.next()

print list(firstn(pi_series(), 8))
[4.0, 2.666666666666667, 3.466666666666668,
 2.895238095238095, 3.339682539682540,
 2.976046176046176, 3.283738483738484,
 3.017071817071818]
```

The Eratosthenes sieve

Example

```
def intsfrom(i):
    while True:
        yield i
        i = i + 1

def exclude_multiples(n, ints):
    for i in ints:
        if (i % n):
            yield i

def sieve(ints):
    while True:
        prime = ints.next()
        yield prime
        ints = exclude_multiples(prime, ints)
```

The Eratosthenes sieve

Example

```
In [1]: import erastothene

In [2]: def firstn(g, n):
   ...:     for i in range(n):
   ...:         yield g.next()
   ...:

In [3]: list(firstn(erastothene.sieve(erastothene.intsfrom(2)), 5))
Out[3]: [2, 3, 5, 7, 11]

In [4]: list(firstn(erastothene.sieve(erastothene.intsfrom(2)), 9))
Out[4]: [2, 3, 5, 7, 11, 13, 17, 19, 23]

In [5]:
```

Syracuse serie

Algorithm

- Take any natural number n.
- If n is even, divide it by 2 to get $n/2$,
- if n is odd multiply it by 3 and add 1 to obtain $3n + 1$.
- Repeat the process indefinitely.

Conjecture

No matter what number you start with, you will always eventually reach 1.

Syracuse

Example

```
def syracuse(n):
    while True:
        n = (3*n + 1) if n % 2 else (n / 2)
        yield n

def firstn(g, n):
    for i in range(n):
        yield g.next()
```

Syracuse

Example

```
In [1]: from syracuse import syracuse, firstn

In [2]: print list(firstn(syracuse(20), 10))
[10, 5, 16, 8, 4, 2, 1, 4, 2, 1]

In [3]: import itertools as it

In [4]: print list(it.takewhile(lambda x : x != 1, syracuse(20)))
[10, 5, 16, 8, 4, 2]

In [5]: print list(it.takewhile(lambda x : x != 1, syracuse(25)))
[76, 38, 19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40,
 20, 10, 5, 16, 8, 4, 2]

In [6]:
```

Recursive generators

Example

```
def gen1():
    a = gen2()
    for i in a:
        yield i
    yield 'gen1'
def gen2():
    a = gen3()
    for i in a:
        yield i
    yield 'gen2'
def gen3():
    for i in xrange(3):
        yield i
    yield 'gen3'
for i in gen1():
    print i
```

Recursive generators

Example

```
barbalala:src krtek> python recursive_generators.py
0
1
2
gen3
gen2
gen1
barbalala:src krtek>
```

Tree traversal

Example

```
class BinTree(object):
    def __init__(self, data, left = None, right = None):
        self.data = data
        self.left = left
        self.right = right

    def __str__(self):
        fmt = '(%d%s%s)'
        return  fmt % (self.data,
                       ', ' + str(self.left) if self.left is not None else '',
                       ', ' + str(self.right) if self.right is not None else '')

def inorder_traversal(tree):
    if tree is not None:
        for t in inorder_traversal(tree.left):
            yield t
        yield tree
        for t in inorder_traversal(tree.right):
            yield t
```

Tree traversal

Example

```
from tree import BinTree, inorder_traversal

t = BinTree(1,
            BinTree(2, BinTree(3), BinTree(4)),
            BinTree(5, BinTree(6), BinTree(7)))

# (1, (2, (3), (4)), (5, (6), (7)))
print t

# 3, 2, 4, 1, 6, 5, 7
print ', '.join(str(node.data) for node in inorder_traversal(t)
```

range vs. xrange

Python code

```
def xrange(start, stop=None, step=1):
    if stop is None:
        start, stop = 0, start
    while step > 0 and start < stop or step < 0 and start > stop:
        yield start
        start += step
```

Chain of generators

Example

```
def change_value():
    for step in range(1, 10):
        yield step

def create_box(steps):
    for step in steps:
        yield (' ').join( str(step) for i in range(2 * step - 1))

def create_pyramid(height):
    boxes = create_box(change_value())
    for step, box in enumerate(boxes):
        if step == height:
            break
        else:
            l = (height-step) * 2
            print ('').join(' ' for i in range(l)) + box
```

Chain of generators

Example

```
In [1]: import pyramid

In [2]: pyramid.create_pyramid(9)
                    1
                  2 2 2
                3 3 3 3 3
              4 4 4 4 4 4 4
            5 5 5 5 5 5 5 5 5
          6 6 6 6 6 6 6 6 6 6 6
        7 7 7 7 7 7 7 7 7 7 7 7 7
      8 8 8 8 8 8 8 8 8 8 8 8 8 8 8
    9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9

In [3]:
```

Power sets

Example

```
def powerset(seq):
    """
    Returns all the subsets of this set.
    """
    if len(seq) <= 1:
        yield seq
        if len(seq) == 1:
            yield []
    else:
        for item in powerset(seq[1:]):
            yield [seq[0]] + item
            yield item
```

Power sets

Example

```
In [1]: import powerset

In [2]: for i in range(4):
   ...:     print range(i), ':', list(powerset.powerset(range(i)))
   ...:
[] : [[]]
[0] : [[0], []]
[0, 1] : [[0, 1], [1], [0], []]
[0, 1, 2] : [[0, 1, 2], [1, 2], [0, 2], [2], [0, 1], [1], [0], []]

In [3]:
```

A generator that follows lines written to a real-time log file

Example

```
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

if __name__ == '__main__':
    logfile = open("access-log")
    for line in follow(logfile):
        print line,
```

Using generators to set up a simple processing pipeline

Example

```
def grep(pattern, lines):
    for line in lines:
        if pattern in line:
            yield line

if __name__ == '__main__':
    from follow import follow

    # Set up a processing pipe : tail -f | grep python
    logfile  = open("access-log")
    loglines = follow(logfile)
    pylines  = grep("python", loglines)

    # Pull results out of the processing pipeline
    for line in pylines:
        print line,
```

Generators and classes

Example

```
class EvenNumbers(object):
    def __init__(self, n = 0):
        self.n = n if n % 2 == 0 else n + 1
    def __call__(self):
        while True:
            yield self.n
            self.n = self.n + 2
g = EvenNumbers(5) # <__main__.EvenNumbers object at 0x10220a590>
print g            # <generator object __call__ at 0x1021f2cd0>
print g()          # 6 8 10 12 14 16
for x in g():
    print x,
    if x > 15:
        import sys
        sys.exit(0)
```

Generators and classes

Example

```
class EvenNumbers(object):
    def __init__(self, n = 0):
        self.n = n if n % 2 == 0 else n + 1
    def __iter__(self):
        return self()
    def __call__(self):
        while True:
            yield self.n
            self.n = self.n + 2
g = EvenNumbers(5) # <__main__.EvenNumbers object at 0x105dd6610>
print g            # <generator object __call__ at 0x105dbebe0>
print iter(g)      # 6 8 10 12 14 16
for x in EvenNumbers(5):
    print x,
    if x > 15:
        import sys
        sys.exit(0)
```

Generators and classes

Example

```
import random
import time

def sensor_temp(init_temp = 0):
    temp = init_temp
    while True:
        yield temp
        p = random.uniform(0, 1)
        temp = (temp - 1) if p <= .05 else (temp if p <= .95 else (temp + 1))

class HighTempDetector(object):
    def __init__(self, sensor, high_temp = 50):
        self.sensor = sensor
        self.high_temp = high_temp
    def __iter__(self):
        return self()
    def __call__(self):
        for temp in self.sensor:
            if temp >= self.high_temp:
                yield time.clock(), temp
```

Introducing generator expressions

Description

- Generator expressions extend naturally from list comprehensions.
- Instead of building a list with values, they return a generator that yields after processing each item.
- Generator expressions are much more memory efficient by performing lazy evaluation.

```
# List comprehension
[expr for item_var in iterable if cond_expr]

# Generator expression
(expr for item_var in iterable if cond_expr)
```

yield and generator expression

Using yield

```
def generator(x, y):
    for i in xrange(x):
        for j in xrange(y):
            yield(i, j)
```

Using generator expression

```
def generator(x, y):
    return ((i, j) for i in xrange(x) for j in xrange(y))
```

Let's sum the squares of the numbers up to 100

Using a loop

```
def sum_of_square():
    total = 0
    for num in range(1, 101):
        total += num * num
    return total
```

Using a list comprehension

```
total = sum([num * num for num in range(1, 101)])
```

Using a generator expression

```
total = sum(num * num for num in xrange(1, 101))
```

Billions

Why do I need generator expressions?

- If we were summing the squares of several billion integers, we'd run out of memory with list comprehensions,

```
total = sum([num * num for num in range(1, 10000001)])
```

- but generator expressions have no problem.

```
total = sum(num * num for num in range(1, 10000001))
```

- This does take time, though!

Dealing with large files: Refactoring example

Example

```
def longest_line_in_file(finename):
    f = open(filename, 'r')
    longest_line = 0
    while True:
        line_len = len(f.readline().strip())
        if not line_len:
            break
        if line_len > longest_line:
            longest_line = line_len
    f.close()
    return longest_line
```

Dealing with large files: Refactoring example

Example

```
def longest_line_in_file(finename):
    f = open(filename, 'r')
    all_lines = f.readlines()
    f.close()
    longest_line = 0
    for line in all_lines:
        line_len = len(line.strip())
        if line_len > longest_line:
            longest_line = line_len
    return longest_line
```

Dealing with large files: Refactoring example

Example

```
def longest_line_in_file(finename):
    f = open(filename, 'r')
    all_lines = [line.strip() for line in f.readlines()]
    f.close()
    longest_line = 0
    for line in all_lines:
        line_len = len(line)
        if line_len > longest_line:
            longest_line = line_len
    return longest_line
```

Dealing with large files: Refactoring example

Example

```
def longest_line_in_file(finename):
    f = open(filename, 'r')
    all_len = [len(line.strip()) for line in f]
    f.close()
    return max(all_len)
```

Dealing with large files: Refactoring example

Example

```
def longest_line_in_file(finename):
    f = open(filename, 'r')
    longest_line = max(len(line.strip()) for line in f)
    f.close()
    return longest_line
```

Python 2.5 adds a simple way to pass values into a generator

Desciption

- In 2.5, `yield` is now an expression, returning a value that can be assigned to a variable or otherwise operated on
- I recommend that you always put parentheses around a yield expression when you're doing something with the returned value.
- Values are sent into a generator by calling its `send(value)` method. The generator's code is then resumed and the `yield` expression returns the specified value.
- If the regular `next()` method is called, the `yield` returns `None`.

Passing values into a generator

Example

```
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

Passing values into a generator

Example

```
In [1]: from simplecounter import counter
In [2]: it = counter(10)
In [3]: print it.next()
0
In [4]: print it.next()
1
In [5]: print it.send(8)
8
In [6]: print it.next()
9
In [7]: print it.next()
---------------------------------------------------------------------------
StopIteration                       Traceback (most recent call last)
/Users/.../gexpressions/src/<ipython-input-7-28de134fde97> in <module>()
In [8]:
```