Python Generators

Reuven M. Lerner, PhD
reuven@lerner.co.il

Generators

- A generator is a function that returns a value to the user
- When we execute the function again, it continues from where we left off
- Generators allow us to be lazy, delaying evaluation and generation of a value until it is needed

Example

```
def abc():
  yield "a"
  yield "b"
  yield "c"
g = abc()
```

- The generator is g, what abc() returns

What is g?

```
>>> g = abc()

>>> g

<generator object abc at 0x1004a76e0>

>>> type(g)

<type 'generator'>
```

What do we do with g?

```
>>> for i in abc():
...   print i
...
a
b
c
```

What do we do with g?

```
>>> for i in g:
...   print i
...
a
b
c
```

Generator vs. iterator

- Hey, that looks like an iterator!
- But why do we need generators?
- Because they’re easy to work with, and fairly efficient

yield vs. return

```
def abc():
  yield "a"
  yield "b"
  yield "c"
```

- yield returns, while remembering its state!
- The next iteration, it does the next yield

Independent state

```
>>> g1 = abc()
>>> g2 = abc()
>>> g1.next()
'a'
>>> g1.next()
'b'
>>> g2.next()
'a'
```

Generator functions

- Returns a new generator when invoked
- Invoking "next" on the generator runs the function through the first "yield"
- Subsequent runs go through the next "yield"
- When the generator function exits, the generator raises StopIteration

When use generators?

- If you need a sequence of computed values
- Only one at a time
- It costs time or memory to do all of the calculations at once
- ... try a generator!

Get all results

- If your generator produces a finite iteration, then you can get all of its results with list() or tuple():

```
>>> list(abc())
['a', 'b', 'c']

>>> tuple(abc())
('a', 'b', 'c')
```

Fibonacci

```
def fib():
  first = 0
  yield first
  second = 1
  yield second
  while True:
    first, second = second, first+second
    yield second
```

Don’t do this!

```
g = fib()

print list(g)
```

return

- You almost certainly don't want to use "return" in a generator function
- If you try to return a value, Python won't let the generator function even compile.
- If you "return" without a value, it raises StopIteration

Parameters

- You may define parameters to your generator function, just like any other function
- The parameter is then a local variable, just like all other local variables,

Other ideas

- Processing filenames
- Handling XML or CSV data
- Results from a database query
- Retrieving data from the network

Don’t raise StopIteration!

- Raising StopIteration in a generator function is a bad idea
  - As of Python 3.5, it’ll start to give you problems
  - As of Python 3.6, you’ll get error messages
- You should return from a generator function, not raise StopIteration

Pipelines

- David Beazley suggests that we use generators like Unix pipes
- You can stack a bunch of generators together, putting your input in one end and getting transformed data on the other

When it hits the end...

- When a generator gets to the end, it raises StopIteration
- And then... it's basically useless. You cannot reset it to the beginning

Generator expressions

- We have already seen list comprehensions:

```
[ x**2 for x in range(10) if x%2 ]
```

- Python also supports generator expressions:

```
( x**2 for x in range(10) if x%2 )
```

Generator expressions

```
>>> type([x**2 for x
          in range(10) if x%2 ])
<type 'list'>


>>> type( x**2 for x
          in range(10) if x%2 )
<type 'generator'>
```

Why?

- Just as list comprehensions let you easily create lists, generator expressions let you easily create generators.
- Instead of writing a function, you can do it all in one line

Comprehensions

```
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> (x*x for x in range(10))
<generator object <genexpr> at 0x10073b690>

>>> {x*x for x in range(10)}
set([0, 1, 4, 81, 64, 9, 16, 49, 25, 36])

>>> {x:x*x for x in range(10)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
8: 64, 9: 81}
```

Using generators

- If you are going to iterate over a potentially large number of items,
- And if you don't want to create an iterator class,
- Then use a generator function — or even better, a generator expression!

Examples

- Write a generator expression that returns all files (i.e., not directories, and not links) from a directory.
- Write a generator expression that returns the lines from a file.
- Write a generator expression that returns the number of times each letter appears in the line.
- Hook them together, and you have a generator pipeline!

Coroutines

- If you put your “yield” call on the right side of assignment, the code that runs can send a value back to the generator!
- Instead of invoking the generator with "next", you can use "send", which takes a parameter.
- The parameter is passed into the generator function, as if it were the result of the call to yield.
- This is known as a coroutine

Simple coroutine

```
def grep(pattern):
    print "Looking for %s" % pattern
    while True:
        line = (yield)
        if pattern in line:
            print "Found '%s' in '%s':" % (pattern, line)
        else:
            print "Did *NOT* find '%s' in '%s'" % (pattern, line)


# Example use
if __name__ == '__main__':
    g = grep("python")
    g.next()
    g.send("Yeah, but no, but yeah, but no")
    g.send("A series of tubes")
    g.send("python generators rock!")
```

Initialize a coroutine

- We always need to invoke "next" on a coroutine, before we can use "send"
- This can get annoying and repetitive
- So hey, let's write a decorator to take care of it for us!

Our decorator

```
def coroutine(func):
    def start(*args, **kwargs):
        g = func(*args, **kwargs)
        g.next()
        return g
    return start
```

Using our decorator

```
@coroutine

def receiver():

    print("Ready to receive")

    while True:

        n = (yield)

        print("Got %s" % n)



r = receiver()

r.send("hello, world")
```