Charming Python: Iterators and simple generators

New constructs in Python 2.2

David Mertz September 01, 2001

Python 2.2 introduces a new construct accompanied by a new keyword. The construct is generators; the keyword is yield. Generators make possible several new, powerful, and expressive programming idioms, but are also a little bit hard to get one's mind around at first glance. In this article, David provides a gentle introduction to generators, and also to the related topic of iterators.

Welcome to the world of exotic flow control. With Python 2.2 (now in its third alpha release -- see Related topics later in this article), programmers will get some new options for making programs tick that were not available -- or at least not as convenient -- in earlier Python versions.

While what Python 2.2 gives us is not quite as mind-melting as the full continuations and microthreads that are possible in Stackless Python, generators and iterators do something a bit different from traditional functions and classes.

Let's consider iterators first, since they are simpler to understand. Basically, an *iterator* is an object that has a `.next()` method. Well, that's not quite true; but it's close. Actually, most iterator contexts want an object that will generate an iterator when the new `iter()` built-in function is applied to it. To have a user-defined class (that has the requisite `.next()` method) return an iterator, you need to have an `__iter__()` method return `self`. The examples will make this all clear. An iterator's `.next()` method might decide to raise a `StopIteration` exception if the iteration has a logical termination.

A generator is a little more complicated and general. But the most typical use of generators will be for defining iterators; so some of the subtlety is not always worth worrying about. A *generator* is a function that remembers the point in the function body where it last returned. Calling a generator function a second (or nth) time jumps into the middle of the function, with all local variables intact from the last invocation.

In some ways, a generator is like the closures which were discussed in previous installments of this column discussing functional programming (see Related topics). Like a closure, a generator "remembers" the state of its data. But a generator goes a bit further than a closure: a generator also "remembers" its position within flow-control constructs (which, in imperative programming, is

something more than just data values). Continuations are still more general since they let you jump arbitrarily between execution frames, rather than returning always to the immediate caller's context (as a generator does).

Fortunately, using a generator is much less work than understanding all the conceptual issues of program flow and state. In fact, after very little practice, generators seem as obvious as ordinary functions.

Taking a random walk

Let's consider a fairly simple problem that we can solve in several ways -- both new and old. Suppose we want a stream of positive random numbers less than one that obey a backward-looking constraint. Specifically, we want each successive number to be at least 0.4 more or less than the last one. Moreover, the stream itself is not infinite, but rather ends after a random number of steps. For the examples, we will simply end the stream when a number less than 0.1 is produced. The constraints described are a bit like one might find in a "random walk" algorithm, with the end condition resembling a "statisficing" or "local minimum" result -- but certainly the requirements are simpler than most real-world ones.

In Python 2.1 or earlier, we have a few approaches to solving our problem. One approach is to simply produce and return a list of numbers in the stream. This might look like:

```
import random
def randomwalk_list():
    last, rand = 1, random.random() # init candidate elements
    nums = []                       # empty list
    while rand > 0.1:                # threshhold terminator
        if abs(last-rand) >= 0.4:    # accept the number
            last = rand
            nums.append(rand)       # add latest candidate to nums
        else:
            print '*',               # display the rejection
        rand = random.random()      # new candidate
    nums.append(rand)               # add the final small element
return nums
```

Utilizing this function is as simple as:

```
for num in randomwalk_list():
    print num,
```

There are a few notable limitations to the above approach. The specific example is exceedingly unlikely to produce huge lists; but just by making the threshhold terminator more stringent, we could create arbitrarily large streams (of random exact size, but of anticipatable order-of-magnitude). At a certain point, memory and performance issues can make this approach undesirable and unnecessary. This same concern got `xrange()` and `xreadlines()` added to Python in earlier versions. More significantly, many streams depend on external events, and yet should be processed as each element is available. For example, a stream might listen to a port, or wait for user inputs. Trying to create a complete list out of the stream is simply not an option in these cases.

One trick available in Python 2.1 and earlier is to use a "static" function-local variable to remember things about the last invocation of a function. Obviously, global variables could do the same job, but they cause the familiar problems with pollution of the global namespace, and allow mistakes due to non-locality. You might be surprised here if you are unfamiliar with the trick--Python does not have an "official" static scoping declaration. However, if named parameters are given mutable default values, the parameters can act as persistent memories of previous invocations. Lists, specifically, are handy mutable objects that can conveniently even hold multiple values.

Using a "static" approach, we can write a function like:

```
import random
def randomwalk_static(last=[1]):# init the "static" var(s)
    rand = random.random()          # init a candidate value
    if last[0] < 0.1:                # threshhold terminator
        return None                  # end-of-stream flag
    while abs(last[0]-rand) < 0.4:   # look for usable candidate
        print '*',                   # display the rejection
        rand = random.random()      # new candidate
    last[0] = rand                  # update the "static" var
return rand
```

This function is quite memory friendly. All it needs to remember is one previous value, and all it returns is a single number (not a big list of them). And a function similar to this could return successive values that depend (partly or wholly) on external events. On the down side, utilizing this function is somewhat less concise, and considerably less elegant:

```
num = randomwalk_static()
while num is not None:
    print num,
    num = randomwalk_static()
```

New ways of walking

"Under the hood", Python 2.2 sequences are all iterators. The familiar Python idiom `for elem in lst:` now actually asks `lst` to produce an iterator. The `for` loop then repeatedly calls the `.next()` method of this iterator until it encounters a `StopIteration` exception. Luckily, Python programmers do not need to know what is happening here, since all the familiar built-in types produce their iterators automatically. In fact, now dictionaries have the methods `.iterkeys()`, `.iteritems()`, and `.itervalues()` to produce iterators; the first is what gets used in the new idiom `for key in dct:`. Likewise, the new idiom `for line in file:` is supported via an iterator that calls `.readline()`.

But given what is actually happening within the Python interpreter, it becomes obvious to use custom classes that produce their own iterators rather than exclusively use the iterators of built-in types. A custom class that enables both the direct usage of `randomwalk_list()` and the element-at-a-time parsimony of `randomwalk_static` is straightforward:

```
import random
class  randomwalk_iter:
def  __init__(self):
    self.last = 1               # init the prior value
    self.rand = random.random() # init a candidate value
def __iter__(self):
    return self                 # simplest iterator creation
def next(self):
    if self.rand < 0.1:         # threshhold terminator
        raise StopIteration     # end of iteration
    else:                       # look for usable candidate
        while abs(self.last-self.rand) < 0.4:
                print '*',           # display the rejection
                self.rand = random.random() # new candidate
            self.last = self.rand   # update prior value
            return self.rand
```

Use of this custom iterator looks exactly the same as for a true list generated by a function:

```
for num in randomwalk_iter():
    print num,
```

In fact, even the idiom `if elem in iterator` is supported, which lazily only tries as many elements of the iterator as are needed to determine the truth value (if it winds up false, it needs to try all the elements, of course).

Leaving a trail of crumbs

The above approaches are fine for the problem at hand. But none of them scale very well to the case where a routine creates a large number of local variables along the way, and winds its way into a nest of loops and conditionals. If an iterator class or a function with static (or global) variables depends on multiple data states, two problems come up. One is the mundane matter of creating multiple instance attributes or static list elements to hold each of the data values. The far more important problem is figuring out how to get back to exactly the relevant part of the flow logic that corresponds to the data states. It is awfully easy to forget about the interaction and codependence of different data.

Generators simply bypass the whole problem. A generator "returns" with the new keyword `yield`, but "remembers" the exact point of execution where it returned. Next time the generator is called, it picks up where it left before -- both in terms of function flow and in terms of variable values.

One does not directly *write* a generator in Python 2.2+. Instead, one writes a function that, when called, returns a generator. This might seem odd, but "function factories" are a familiar feature of Python, and "generator factories" are an obvious conceptual extension of this. What makes a function a generator factory in Python 2.2+ is the presence of one or more `yield` statements somewhere in its body. If `yield` occurs, `return` must only occur without any accompanying return value. A better choice, however, is to arrange the function bodies so that execution just "falls off the end" after all the `yield`s are accomplished. But if a `return` is encountered, it causes the produced generator to raise a `StopIteration` exception rather than yield further values.

In my opinion, the choice of syntax for generator factories was somewhat poor. A `yield` statement can occur well into the body of a function, and you might be unable to determine that a function

is destined to act as a generator factory anywhere within the first N lines of a function. The same thing could, of course, be true of a function factory -- but being a function factory doesn't change the actual *syntax* of a function body (and a function body is allowed to sometimes return a plain value; albeit probably not out of good design). To my mind, a new keyword -- such as `generator` in place of `def` -- would have been a better choice.

Quibbles over syntax aside, generators have the good manners to automatically act as iterators when called on to do so. Nothing like the `.__iter__()` method of classes is needed here. Every `yield` encountered becomes a return value for generator's `.next()` method. Let's look at the simplest generator to make things clear:

```
>>> from __future__ import generators
>>> def gen():
        yield 1

>>> g = gen()
>>> g.next()
1
>>> g.next()
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in ?
    g.next()
StopIteration
```

Let's put a generator to work in our sample problem:

```
from __future__ import generators   # only needed for Python 2.2
import random
def  randomwalk_generator():
    last, rand = 1, random.random() # initialize candidate elements
    while rand > 0.1:                # threshhold terminator
        print '*',                   # display the rejection
        if abs(last-rand) >= 0.4:    # accept the number
            last = rand             # update prior value
            yield rand              # return AT THIS POINT
        rand = random.random()      # new candidate
    yield rand                      # return the final small element
```

The simplicity of this definition is appealing. You can utilize the generator either manually or as an iterator. In the manual case, the generator can be passed around a program, and called wherever and whenever needed (which is quite flexible). A simple example of the manual case is:

```
gen = randomwalk_generator()
try:
    while 1: print gen.next(),
except StopIteration:
    pass
```

Most frequently, however, you are likely to use a generator as an iterator, which is even more concise (and again looks just like an old-fashioned sequence):

```
for num in randomwalk_generator():
    print_short(num)
```

Yielding

It will take a little while for Python programmers to become familiar with the ins-and-outs of generators. The added power of such a simple construct is surprising at first; and even quite accomplished programmers (like the Python developers themselves) will continue to discover subtle new techniques using generators for some time, I predict.

To close, let me present one more generator example that comes from the `test_generators.py` module distributed with Python 2.2. Suppose you have a tree object, and want to search its leaves in left-to-right order. Using state-monitoring variables, getting a class or function just right is difficult. Using generators makes it almost laughably easy:

```
>>>> # A recursive generator that generates Tree leaves in in-order.
>>> def inorder(t):
...     if t:
...         for x in inorder(t.left):
...             yield x
...         yield t.label
...         for x in inorder(t.right):
...             yield x
```

Related topics

- Read the previous installments of *Charming Python*.
- Regarding the last several Python versions, Andrew Kuchling has written his usual excellent introduction to the changes in Python 2.2; read *What's New in Python 2.2*.
- Read the definitive word on Simple Generators in the Python Enhancement Proposal, PEP255.
- The real dirt on Iterators is in PEP234.
- The code demonstated in this column installment can be found in a single source file.
- Read related *developerWorks* articles by David Mertz:
  - Functional programming in Python, Part 1
  - Functional programming in Python, Part 2
  - Functional programming in Python, Part 3
- Browse more Linux resources on *developerWorks*.
- Browse more Open source resources on *developerWorks*.