Generators functions allow you to declare a function that behaves like an iterator, i.e. it can be used in a for loop.

Simplified Code

The simplification of code is a result of generator function and generator expression support provided by Python.

To illustrate this, we will compare different implementations that implement a function, "firstn", that represents the first *n* non-negative integers, where *n* is a really big number, and assume (for the sake of the examples in this section) that each integer take up a lot of space, say 10 megabytes each.

Note: Please note that in real life, integers do not take up that much space, unless they are really, really, really, big integers. For instance you can represent a 309 digit number with 128 bytes (add some overhead, it will still be less than 150 bytes).

First, let us consider the simple example of building a list and returning it.

Toggle line numbers

```
# Build and return a list
def firstn(n):
```

```python
    num, nums = 0, []
    while num < n:
        nums.append(num)
        num += 1
    return nums

sum_of_first_n = sum(firstn(1000000))
```

The code is quite simple and straightforward, but its builds the full list in memory. This is clearly not acceptable in our case, because we cannot afford to keep all *n* "10 megabyte" integers in memory.

So, we resort to the generator pattern. The following implements generator as an iterable object.

Toggle line numbers

```python
# Using the generator pattern (an iterable)
class firstn(object):
    def __init__(self, n):
        self.n = n
        self.num, self.nums = 0, []

    def __iter__(self):
        return self

    # Python 3 compatibility
    def __next__(self):
        return self.next()

    def next(self):
        if self.num < self.n:
            cur, self.num = self.num, self.num+1
            return cur
        else:
            raise StopIteration()

sum_of_first_n = sum(firstn(1000000))
```

This will perform as we expect, but we have the following issues:

- there is a lot of boilerplate
- the logic has to be expressed in a somewhat convoluted way

Furthermore, this is a pattern that we will use over and over for many similar constructs. Imagine writing all that just to get an iterator.

Python provides generator functions as a convenient shortcut to building iterators. Lets us rewrite the above iterator as a generator function:

Toggle line numbers

```
# a generator that yields items instead of returning a list
def firstn(n):
    num = 0
    while num < n:
        yield num
        num += 1

sum_of_first_n = sum(firstn(1000000))
```

Note that the expression of the number generation logic is clear and natural. It is very similar to the implementation that built a list in memory, but has the memory usage characteristic of the iterator implementation.

Note: the above code is perfectly acceptable for expository purposes, but remember that in Python 2 ***firstn()*** *is equivalent to the built-in* ***xrange()*** *function, and in Python 3* ***range()*** *is a generator. The built-ins will always be much faster. SH*

Generator expressions provide an additional shortcut to build generators out of expressions similar to that of list comprehensions.

In fact, we can turn a list comprehension into a generator expression by replacing the square brackets ("[ ]") with parentheses. Alternately, we can think of list comprehensions as generator expressions wrapped in a list constructor.

Consider the following example:

Toggle line numbers

```
# list comprehension
doubles = [2 * n for n in range(50)]

# same as the list comprehension above
doubles = list(2 * n for n in range(50))
```

Notice how a list comprehension looks essentially like a generator expression passed to a list constructor.

By allowing generator expressions, we don't have to write a generator function if we do not need the list. If only list comprehensions were available, and we needed to lazily build a set of items to be processed, we will have to write a generator function.

This also means that we can use the same syntax we have been using for list comprehensions to build generators.

Keep in mind that generators are a special type of iterator, and that containers like `list` and `set` are also iterables. The uniform way in which all of these are handled, adds greatly to the simplification of code.

Improved Performance

The performance improvement from the use of generators is the result of the lazy (on demand) generation of values, which translates to lower memory usage. Furthermore, we do not need to wait until all the elements have been generated before we start to use them. This is similar to the benefits provided by iterators, but the generator makes building iterators easy.

This can be illustrated by comparing the `range` and `xrange` built-ins of Python 2.x.

Both `range` and `xrange` represent a range of numbers, and have the same function signature, but `range` returns a `list` while `xrange` returns a generator (at least in concept; the implementation may differ).

Say, we had to compute the sum of the first *n*, say 1,000,000, non-negative numbers.

Toggle line numbers

```
# Note: Python 2.x only
# using a non-generator
sum_of_first_n = sum(range(1000000))

# using a generator
sum_of_first_n = sum(xrange(1000000))
```

Note that both lines are identical in form, but the one using `range` is much more expensive.

When we use `range` we build a 1,000,000 element list in memory and then find its sum. This is a waste, considering that we use these 1,000,000 elements just to compute the sum.

This waste becomes more pronounced as the number of elements (our *n*) becomes larger, the size of our elements become larger, or both.

On the other hand, when we use `xrange`, we do not incur the cost of building a 1,000,000 element list in memory. The generator created by xrange will generate each number, which `sum` will consume to accumulate the sum.

In the case of the "range" function, using it as an iterable is the dominant use-case, and this is reflected in Python 3.x, which makes the `range` built-in return a generator instead of a list.

Note: a generator will provide performance benefits only if we do not intend to use that set of generated values more than once.

Consider the following example:

Toggle line numbers

```
# Note: Python 2.x only
s = sum(xrange(1000000))
p = product(xrange(1000000))
```

Imagine that making a integer is a very expensive process. In the above code, we just performed the same expensive process twice. In cases like this, building a list in memory might be worth it (see example below):

Toggle line numbers

```
# Note: Python 2.x only
nums = list(xrange(1000000))
s = sum(nums)
p = product(nums)
```

However, a generator might still be the only way, if the storage of these generated objects in memory is not practical, and it might be worth to pay the price of duplicated expensive computations.

Examples

For example, the RangeGenerator can be used to iterate over a large number of values, without creating a massive list (like range would)

Toggle line numbers

```
#the for loop will generate each i (i.e. 1,2,3,4,5, ...), add it to total,  and throw it away
#before the next i is generated.  This is opposed to iterating through range(...), which creates
#a potentially massive list and then iterates through it.
total = 0
for i in irange(1000000):
    total += i
```

Generators can be composed. Here we create a generator on the squares of consecutive integers.

Toggle line numbers

```
#square is a generator
square = (i*i for i in irange(1000000))
#add the squares
total = 0
for i in square:
    total += i
```

Here, we compose a square generator with the takewhile generator, to generate squares less than 100

Toggle line numbers

```
#add squares less than 100
```

```
square = (i*i for i in count())
bounded_squares = takewhile(lambda x : x< 100, square)
total = 0
for i in bounded_squares:
    total += i
```

to be written: Generators made from classes?

Links

- PEP-255: Simple Iterators -- the original
- Iterators and Simple Generators
- combinatorial functions in itertools
- Python Generator Tricks -- various infinite sequences, recursions, ...
- "weightless threads" -- simulating threads using generators
- XML processing -- yes, using generators
- C2:GeneratorsAreNotCoroutines -- particulars on generators, coroutines, and continuations

See also: Iterator

Discussion

I once saw MikeOrr demonstrate Before and After examples. But, I forget how they worked.

Can someone demonstrate here?

He did something like: Show how a normal list operation could be written to use generators. Something like:

Toggle line numbers

```
def double(L):
    return [x*2 for x in L]

eggs = double([1, 2, 3, 4, 5])
```

...he showed how that, or something like that, could be rewritten using iterators, generators.

It's been a while since I've seen it, I may be getting this all wrong.

-- LionKimbro 2005-04-02 19:12:19

Toggle line numbers

```
# explicitly write a generator function
def double(L):
    for x in L:
        yield x*2

# eggs will be a generator
eggs = double([1, 2, 3, 4, 5])

# the above is equivalent to ("generator comprehension"?)
eggs = (x*2 for x in [1, 2, 3, 4, 5])

# need to do this if you need a list
eggs = list(double([1, 2, 3, 4, 5]))

# the above is equivalent to (list comprehension)
eggs = [x*2 for x in [1, 2, 3, 4, 5]]
```

For the above example, a generator comprehension or list comprehension is sufficient unless you need to apply that in many places.

Also, a generator function will be cleaner and more clear, if the generated expressions are more complex, involve multiple steps, or depend on additional temporary state.

Consider the following example:

Toggle line numbers

```
def unique(iterable, key=lambda x: x):
    seen = set()
    for elem, ekey in ((e, key(e)) for e in iterable):
        if ekey not in seen:
            yield elem
            seen.add(ekey)
```

Here, the temporary keys collector, *seen*, is a temporary storage that will just be more clutter in the location where this generator will be used.

Even if we were to use this only once, it is worth writing a function (for the sake of clarity; remember that Python allows nested functions).

Generators (last edited 2014-01-18 11:13:27 by kekschaot)