import antigravity

Why bother when all that is needed is an "import antigravity". Circumventing gravity can't possibly get any simpler. Here's to all things pythonic..!
A Python Code blog with insightful(hopefully) snippets illustrating its power and simplicity, along the long and treacherous path to finally conquering the language.
And while we're on it, do enjoy squeezing the juice out of Python[the one with the capital P]

Thursday, July 10, 2008

learning to yield

I was thinking of alternative titles like "yielding to yield" or even "learning to yield to yield". Might not *be* as redundant as they *seem*. Anyhow lets get down straight to the code..

Here are some really nice uses for Python's "yield" statement which generates results "lazily" : as and when required. In each of the following snippets the search is depth-first and the function yields a solution as soon as it sees one, like a normal DFS. Unlike the usual DFS however, it also capable of resuming from exactly where it left off in search of other solutions. This achieves two things: 1. The function, over time, analyzes the whole search space and finds *all* solutions. 2. It also saves the unecessary time and space overhead of a similar function that uses "return", which finds all solutions before returning control.

Generating Partitions

Generating integer partitions is a standard recursive problem. Given an integer n find the number of ways it can be expressed as the sum of integers in the range [1,n] (both ends inclusive)

```
def partitions(n, curtot = 0, cur = []) :
    if curtot == n :
        yield cur

    if cur : t = cur[-1]
    else  : t = 1

    for i in xrange(t, n - curtot + 1) :
        for j in partitions(n, curtot + i, cur + [i]) :
            yield j

for each in partitions(7) :
    print each
```

About Me

CHILLU

View my complete profile

Contact Me

quan...@gmail.com

yields(who would've thought yield could mean so many things) something like..

```
[1, 1, 1, 1, 1, 1, 1]
[1, 1, 1, 1, 1, 2]
[1, 1, 1, 1, 3]
[1, 1, 1, 2, 2]
[1, 1, 1, 4]
[1, 1, 2, 3]
[1, 1, 5]
[1, 2, 2, 2]
[1, 2, 4]
[1, 3, 3]
[1, 6]
[2, 2, 3]
[2, 5]
[3, 4]
[7]
```

Finding all paths in a directed graph from A to B

This kind of graph traversal seems ideal for a yield statement as it generates paths from A to B in a depth-first manner and yielding the result immediately as it finds one, only to resume from where it left off when required. The graph is represented as is suggested by Guido using lists and dictionaries. A sample graph looks like:

```
graph = {'A': ['B', 'C'],
'B': ['C', 'D'],
'C': ['D'],
'D': ['C'],
'E': ['F'],
'F': ['C']}
```

where node X is connected to Y (in that direction) iff "'Y' in graph['X']" evaluates to true. Code follows..

```
def find_all_paths(G, start, end, visited = []) :
    for i in G[start] :
        if i not in visited :
            if i == end :
                yield [start] + [i]
            else :
                for j in find_all_paths(G, i, end, visited + [st
art]) :
                    yield [start] + j

for each in find_all_paths(graph) :
    print each
```

Transgress programming the way Guido indented it

Subscribe and I'll ke

Posts

Comments

Label Cloud

Amarok BeautifulSoup binary search bisect cmus conky DFS dictionary discrete time convolution faster generators golomb sequence hashcash iterators lambda leoLyricsAPI list list comprehensions lyrics one liners optimize parallel python. swig psyco pyrex python quine quizzy's word challenge recursion reverse indent script sierpinski fractal solver speedup the-end yield

Outputs this:

```
['A', 'B', 'C', 'D']
['A', 'B', 'D']
['A', 'C', 'D']
```

This has the advantage of stop/resume over Guido's implementation which uses a return. And also does not require a separate function to find the smallest path. You can simply do:

```
min(find_all_paths(graph, 'A', 'D'), key = len)
```

using the relatively new generator expressions that follow similar semantics as list comprehensions but generate items lazily. Python 3000 is also moving in this direction(i.e becoming lazy) with many functions that earlier returned now being replaced with functions that yielded. The range function, for example, in Py3k will perform similar to xrange in previous versions.

Permutations

Arguably the simplest(read easily understood) implementation of generating permutations of a given string. This should get you started thinking on the power of generators.

```
def permute(l) :
    if len(l) == 1 :
        yield l
    else :
        for i in range(len(l)) :
            for j in permute(l[:i] + l[i + 1:]) :
                yield [l[i]] + j

test = ['import', 7**7, 'anti', 42, 'gravity']
for i in permute(test) :
    print i
```

I'm omitting the output here.

Combinations

This code is also equally self-documenting. It chooses r objects from a list in all possible ways.

```
def combinations(l, r) :
    if r == 1 :
        for i in l :
```

```
            yield [i]
    elif len(l) == r :
        yield l
    else :
        #choose l[0]
        for j in combinations(l[1:], r - 1 ) :
            yield j + [l[0]]
        #dont choose l[0]
        for j in combinations(l[1:], r) :
            yield j

test, r = ['import', 42, 7**7, 'anti', ['.', ':'], 'gravity'], 4
for i in combinations(test, r) :
    print i
```

What you should learn from these examples is the versatility of generators. This article only scratches the surface of generators. Python generators now support a send() method besides the next() method that for loops use implicitly. "yield" is now an expression that contains the value of the first argument in the send method. This makes generators co-routines(as against their sub-routine nature as illustrated here) by enabling information to travel both ways. If you find all this confusing here's the PEP that proposed this enhancement.

May thee learn to yield!

Posted by chillu at 12:22 PM

Labels: generators, iterators, python, script, yield

No comments:

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)