Generators and Iterators

Building your own generators with `yield`

In [42]:
```
def counter(start, end):
    current = start
    while current < end:
        yield current
        current += 1
```

In [43]:
```
counter(1, 10)
```

Out[43]: <generator object counter at 0x1c61500>

In [44]:
```
x = counter(1,10)
x.next()
```

Out[44]: 1

In [45]:
```
x.next()
```

Out[45]: 2

In [46]:
```
x.next()
```

Out[46]: 3

In [47]:
```
x = counter(1,10)
list(x)
```

Out[47]: [1, 2, 3, 4, 5, 6, 7, 8, 9]

`yield` can also be used as a function, along with the `send()` method

In [48]:
```
def accumulator(start=0):
    current = start
    while True:
        current += yield(current)
```

In [49]:
```
x = accumulator()
x.next()
```

Out[49]: 0

In [50]:
```
x.send(1)
```

Out[50]: 1

In [51]:
```
x.send(1)
```

Out[51]: 2

In [52]:
```
x.send(10)
```

Out[52]: 12

The iterator protocol

What does `for x in sequence:` *really* do?

In [53]:
```
seq = range(4)
for x in seq: print x
```

```
0
1
2
3
```

In [54]:
```
iter_seq = iter(seq)
print iter_seq
```

```
<listiterator object at 0x1c95ad0>
```

In [55]:
```
iter_seq = iter(seq)
try:
    while True:
        x = iter_seq.next()
        print x
except StopIteration:
    pass
```

```
0
1
2
3
```

Generators are their own iterators:

In [56]:
```
print counter(0, 4)
print iter(counter(0, 4))
```

```
<generator object counter at 0x1c61780>
<generator object counter at 0x1c61780>
```

In [57]:
```
for item in counter(0, 4): print item
```

```
0
1
2
3
```

We can also define our own iterator classes (though generators are usually more readable):

In [58]:
```
class Counter(object):
    def __init__(self, start, end):
        self._start = start
        self._end = end
    def __iter__(self):
        return CounterIterator(self._start, self._end)

class CounterIterator(object):
    def __init__(self, start, end):
        self._cur = ctr._start
        self._end = ctr._end
    def next(self):
        result = self._cur
        self._cur += 1
        if result < self._end:
            return result
        else:
            raise StopIteration

ctr = Counter(0, 5)
print list(ctr)
```

```
[0, 1, 2, 3, 4]
```

Loop comprehensions

In [59]:
```
[ x*2 for x in range(4) ]
```

Out[59]:
```
[0, 2, 4, 6]
```

In [67]:
```
lst = [ ]
for x in range(4):
    lst.append(x*2)
lst
```

Out[67]:
```
[0, 2, 4, 6]
```

In [60]:
```
[ (x,y) for x in range(4) for y in range(4) ]
```

Out[60]:
```
[(0, 0),
 (0, 1),
 (0, 2),
 (0, 3),
 (1, 0),
 (1, 1),
 (1, 2),
 (1, 3),
 (2, 0),
 (2, 1),
 (2, 2),
 (2, 3),
 (3, 0),
 (3, 1),
 (3, 2),
 (3, 3)]
```

In [62]:
```
[ [ (r,c) for c in range(4) ]
  for r in range(4) ]
```

Out[62]:
```
[[(0, 0), (0, 1), (0, 2), (0, 3)],
 [(1, 0), (1, 1), (1, 2), (1, 3)],
 [(2, 0), (2, 1), (2, 2), (2, 3)],
 [(3, 0), (3, 1), (3, 2), (3, 3)]]
```

In [63]:
```
[ x for x in range(10) if x % 2 == 0 ]
```

Out[63]:
```
[0, 2, 4, 6, 8]
```

In [66]:
```
[ x * 4
  for x in range(10)
  if x % 2 == 0
  if x % 3 == 0 ]
```

Out[66]:
```
[0, 24]
```

Generator expressions

In [68]:
```
[ x for x in range(10) if x % 2 == 0 ]
```

Out[68]:
```
[0, 2, 4, 6, 8]
```

In [69]:
```
( x for x in range(10) if x % 2 == 0 )
```

Out[69]:
```
<generator object <genexpr> at 0x1c61730>
```

In [70]:
```
gen = ( x for x in range(10) if x % 2 == 0 )
```

In [71]:
```
gen.next()
```

Out[71]:
```
0
```

In [72]:
```
gen.next()
```

Out[72]:
```
2
```

In [79]:
```
list(gen)
```

Out[79]:
```
[6, 8]
```

Exercises

- Write a generator that will yield the nodes of a tree and their depth in post-order
- Write a loop that uses that generator to *print* the nodes of a tree in post-order