Range vs. xrange

Help Center

In the "Timing" activity, we saw `xrange` consistently outperforming `range` in IDLE based on this code. What is the cause of this behavior. It all comes down to the implementation of these functions. We can gain a clue by looking at the output of just these functions alone:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> xrange(5)
xrange(5)
>>>
```

We can see by this output that the result of calling `range` is a list while the result of calling `xrange` is some mysterious `xrange` object. Here in lies the problem with `range`. When writing code like:

```
for i in range(n):
    ....
```

Desktop Python first constructs a list of length *n* and then begins iterating over it. The issue is that creating and then iterating over large lists can be slow; especially when *n* becomes large. The `for` statement in Python doesn't have to iterate over just lists, but any iterable object. `xrange` is such an object and, without going into detail, we state that iterating over objects like `xrange` is measurably faster than iterating over lists. (You might experiment with this timing code in CodeSkulptor and see if you find a difference in running time between `range` and `xrange`.)

It is for this reason that we see different performance characteristics when using different techniques for examining all the nodes in a graph using our dictionary based representation. Note how invoking the `keys` method creates a list just like `range`:

```
>>> g = { 0: set([1]), 1:set([2]), 2:set([0]) }
>>> g.keys()
[0, 1, 2]
```

We can avoid creating this list during iteration by instead looping over the nodes in a graph in the following manner:

```
>>> for n in g:
        print n


0
1
2
```

```
>>>
```

While both are correct, the second technique can save massive amounts of time.

Created Sun 24 Aug 2014 12:41 AM BST

Last Modified Sun 24 Aug 2014 6:01 AM BST