Efficiency of List Operations

MARCH 30TH, 2012
Some ways of modifying lists are faster than others...

Consider this code snippet:

```python
L = []
for i in range(100000):
    L.insert(0, i)
```

This constructs a list of one hundred thousand integers: 99999, 99998, 99997, ..., 3, 2, 1, 0.

How does this compare in speed to the other ways one can do this in Python?
Other ways of doing the same thing...

```
L = []
for i in range(100000-1, 0, -1):
    L.append(i)
```

```
L = []
for i in range(100000):
    L = [i] + L
```
Here is a puzzle

- When I ran these different ways and measured the running time, here is what I got (in seconds):
  
  0.031,  5.063,  34.55.

Can you match the running times with the code snippets?

- The medium-speed code is more than 150 times slower than the fastest code. The slowest code is more than 1000 times slower than the fastest code!
• Suppose we execute $L = [12, 15, 11, 4]$.

• A block of memory is allocated and the items 12, 15, 11, and 4 are stored consecutively at the beginning of this block.

• This allows efficient access to all elements of the list. The location of $L[i]$ in memory is simply $i + \text{starting location of } L$.

• This guarantees that every element in the list, no matter what its index is, can be accessed equally quickly. This kind of access is called *random access*.
Consequences of this implementation

- **append** is fast. Consider `L.append(e)`. The length of `L` is known and hence the location of the first empty slot following `L` is also known. The element `e` is stored in this slot.

- Notice that the running time of the **append** operation is *independent* of the size of `L`. `append` takes the same amount of time, no matter how large `L` is.

- We say that the running time of `append` is *constant*. This does not mean that it is the same across different machines.
Consequences of this implementation

- `insert` and `remove` can be slow because these might cause a large portion of the list to “shift.”
- For example, `L.insert(0, e)` causes every element in the list to move one slot. This creates a “hole” at the beginning of the list for element `e`.
- This also means that insert operations towards the end of the list are cheaper than those at the beginning of the list.
- In the **worst case** insert takes time that is proportional to the length of L.
- In other words, insert is said to take linear time in the worst case.
Analyzing the code snippets

```
L = []
for i in range(n-1, 0, -1):
    L.append(i)
```

• Assume that append takes time $c$, a constant that has nothing to do with $n$.
• Since the for-loop executes $n$ times, the running time of this code snippet is $cn$.
• Since $c$ is a constant this is a linear function in $n$. 
Analyzing the code snippets

```python
L = []
for i in range(n):
    L.insert(0, i)
```

• After the for-loop has executed $i$ times, we have a list of length $i$. We know that `insert` takes time $c \times i$ on this list.
• Therefore the total running time is
  $$c \times (1 + 2 + 3 + ... + n-1) = c \times n \times (n - 1)/2.$$  
• Since $c$ is a constant this is a quadratic function in $n$. 