Efficient String Concatenation in Python

An assessment of the performance of several methods

Introduction

Building long strings in the Python progamming language can sometimes result in very slow running code. In this article I investigate the computational performance of various string concatenation methods.

In Python the string object is immutable - each time a string is assigned to a variable a new object is created in memory to represent the new value. This contrasts with languages like perl and basic, where a string variable can be modified in place. The common operation of constructing a long string out of several short segments is not very efficient in Python if you use the obvious approach of appending new segments to the end of the existing string. Each time you append to the end of a string, the Python interpreter must create a new string object and copy the contents of both the existing string and the appended string into it. As the strings you are manipulating become large this proces becomes increasingly slow.

What other methods are available and how does their performance compare? I decided to test several different approaches to constructing very long strings to see how much they vary in efficiency.

For this comparison I required a test problem that calls for the construction of very long strings out of a large number of segments. It should not do much other computation, so that the measured performance is dependent only on the string operation performance.

The test case I used is to concatenate a series of integers from zero up to some large number. One can easily vary this number to vary the size of the produced strings. For example the first 20 integers would give us a string like this:

```
012345678910111213141516171819
```

Although this particular test problem doesn't have any real world application that I can think of, it makes a good test case because it is easy to code and simple both conceptually and computationally. The component strings vary in both content and length, which prevents any possible interpreter or hardware optimizations that might rely on large strings of repeated bytes. I don't think the python interpreter actually does this, but it's a good principle to use a test that isn't susceptible to such situational optimizations.

Six Methods

These are the methods that I tested. Each of these six Python snippets computes the same output string.

Method 1: Naive appending

```
def method1():
  out_str = ''
  for num in xrange(loop_count):
    out_str += `num`
  return out_str
```

To me this is the most obvious approach to the problem. Use the concatenate operator (+=) to append each segment to the string. loop_count provides the number of strings to use. Using backticks (``) around num on the fourth line converts the integer value to a string. You can accomplish the same thing with the str() function, but that ended up being somewhat slower, so I stuck with the backticks for all my methods. As I mentioned, although this method is obvious it is not at all efficient. You can see in the results below that we ran a mere 3770 string operations per second. If you need to do lots of concatenations, this is not the right way to go about it.

Method 2: MutableString class

```
def method2():
  from UserString import MutableString
  out_str = MutableString()
  for num in xrange(loop_count):
    out_str += `num`
  return out_str
```

The python library includes a class called MutableString. According to the documentation the class is for educational purposes. One might think that an append operator on a mutable string would not reallocate or copy strings. In the test this method performed even worse than method 1. Examining the source code for UserString.py I found that the storage for MutableString is just a class member of type string and indeed MutableString doesn't even override the `__add__` method. Concatenations using this class aren't going to be any faster than normal immutable string operations, and indeed the extra overhead of interpreting the MutableString class methods make this approach a good deal slower.

Method 3: Character arrrays

```
def method3():
  from array import array
  char_array = array('c')
  for num in xrange(loop_count):
    char_array.fromstring(`num`)
  return char_array.tostring()
```

I almost didn't try this method at all but I had seen it suggested in a mail list, so I decided to give it a whirl. The idea is to use an array of characters to store the string. Arrays are mutable in python, so they can be modified in place without copying the existing array contents. In this case we're not interested in changing existing array elements. We just want to add new array elements at the end of the array. The `fromstring()` call appends the string character by character into the existing array.

Method 4: Build a list of strings, then join it

```
def method4():
  str_list = []
  for num in xrange(loop_count):
    str_list.append(`num`)
  return ''.join(str_list)
```

This approach is commonly suggested as a very pythonic way to do string concatenation. First a list is built containing each of the component strings, then in a single join operation a string is constructed conatining all of the list elements appended together.

There's a funny looking python idiom on the last line - we call the join method of the object identified by the empty string. Not too many languages will let you call methods on a string literal. If you find that offensive, you can write instead: `string.join(str_list, '')`

Method 5: Write to a pseudo file

```
def method5():
  from cStringIO import StringIO
  file_str = StringIO()
  for num in xrange(loop_count):
    file_str.write(`num`)

  return file_str.getvalue()
```

The cStringIO module provides a class called StringIO that works like a file, but is stored as a string. Obviously it's easy to append to a file - you simply write at the end of it and the same is true for this module. There is a similar module called just StringIO, but that's implemented in python whereas cStringIO is in C. It should be pretty speedy. Using this object we can build our string one write at a time and then collect the result using the `getvalue()` call.

Interestingly enough string objects in Java are immutable just like python. In java there is a class called StringBuffer. This is a bit more powerful than either the python StringIO or the array approach, because it supports inserting and removing sub-strings as well as appending them.

Method 6: List comprehensions

```
def method6():
  return ''.join([`num` for num in xrange(loop_count)])
```

This method is the shortest. I'll spoil the surprise and tell you it's also the fastest. It's extremely compact, and also pretty understandable. Create a list of numbers using a list comprehension and then join them all together. Couldn't be simpler than that. This is really just an abbreviated version of Method 4, and it consumes pretty much the same amount of memory. It's faster though because we don't have to call the `list.append()` function each time round the loop.

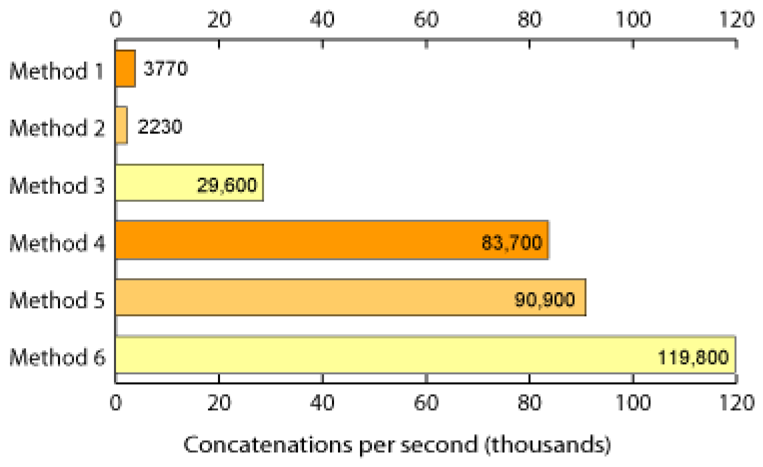
Results

I wanted to look at both the length of time taken to build the string and the amount of memory used by the Python interpreter during the computation. Although memory is cheap, there are a couple of reasons why it can be an important factor. The python program may be running on a system that imposes fixed resource limits. For example in a shared web hosting environment, the machine may be configured to limit the memory size of each process. Typically the kernel will kill a process whose allocated memory exceeds the quota. That would be annoying for a CGI script, and really unfortunate for a long-lived server process. So in those cases keeping memory use from expanding unpredictably is important. The other reason is that when you're dealing with very large strings, having the interpreter's memory allocation grow too large could cause the virtual memory system to start paging the process out to disk. Then performance will really go down hill. It doesn't matter if you find the fastest algorithm in the world - if it uses too much memory it will run slow as a dog. If we use an algorithm that uses less memory, the chances of paging are reduced and we will have more predictable performance.

I tried each method of the methods as a separate test using it's own python process.
I ran these tests using Python 2.2.1 on a 433MHz PII Celeron under FreeBSD 4.9.

Results: Twenty thousand integers

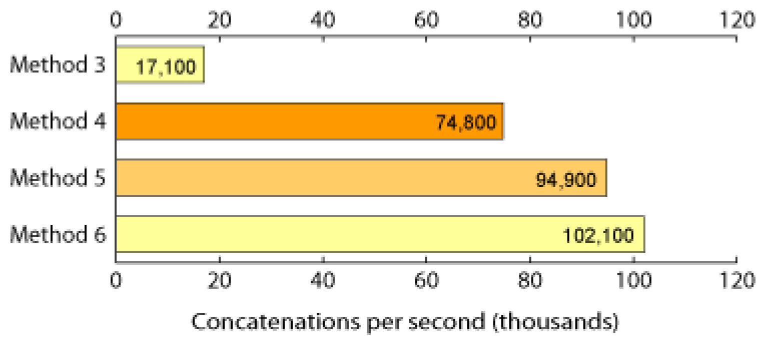
In the first test 20,000 integers were concatenated into a string 86kb long.



	Concatenations per second	Process size (kB)
Method 1	3770	2424
Method 2	2230	2424
Method 3	29,600	2452
Method 4	83,700	3028
Method 5	90,900	2536
Method 6	119,800	3000

Results: Five hundred thousand integers

Next I tried a run of each method using 500,000 integers concatenated into a string 2,821 kB long. This is a much more serious test and we start to see the size of the python interpreter process grow to accomodate the data structures used in the computation.



	Concatenations per second	Process size (kB)
Method 3	17,100	8,016
Method 4	74,800	22,872
Method 5	94,900	10,480
Method 6	102,100	22,844

I didn't even bother to try run this test to completion using Methods 1 and 2. They copy the entire source string on each append operation, so their performance will be O(n^2). It would take many minutes to concatenate a half million integers using these methods.

Comparing the results on this test to our previous graph, notice that the number of concatenations per second is lower for methods 3, 4 & 6. That's not too surprising - the string representation of each integer is a little longer in this test - usually five digits instead of four. In the first test Method 3 performed ten times better than our first two methods, but it didn't scale that well on the longer test. It is now doing less than 60% of its previous performance. It did however use less space than any of the other reasonable methods. Clearly python is doing a great job of storing the array efficiently and garbage collecting the temporary strings in this case.

The performance of Method 4 is more than twenty times better than naive appending in the 20,000 test and it does pretty well also on the 500,000 test. Interestingly method 5 did better in the longer test. Method 6 is still the overall winner, but Method 5 is now doing more concatenations per second and has almost caught up with Method 6. We can guess that if we went to an even longer running test, Method 5 would surpass Method 6.

Notice also the differences in process sizes. At the end of the computation for Method 6 the interpreter is using 22,844kB of memory, eight times the size of the string it is computing, whereas Methods 3 and 5 uses less than half that much.

Conclusions

I would use Method 6 in most real programs. It's fast and it's easy to understand. It does require that you be able to write a single expression that returns each of the values to append. Sometimes that's just not convenient to do - for example when there are several different chunks of code that are generating output. In those cases you can pick between Method 4 and Method 5.

Method 4 wins for flexibility. You can use all of the normal slice operations on your list of strings, for insertions, deletions and modifications. The performance for appending is pretty decent.

Method 5 wins out on efficiency. It's uses less memory than either of the list methods (4 & 6) and it's faster than even the list comprehension for very large numbers of operations (more than about 700,000). If you're doing a lot of string appending cStringIO is the way to go.

Measurement techniques

Measuring the time taken to execute each method was relatively easy. I used the Python library timing module to measure elapsed time. I didn't attempt to measure the CPU time used by the Python process as opposed to other processes running on the machine, but the machine was idle at the time, so I don't think this would make much difference.

Measuring memory used was a little trickier. Python doesn't currently provide a way to monitor the size of allocated objects, so I instead used the Unix 'ps' command to monitor the allocated process size. Since process size varies during execution I wanted to measure the maximum allocated memory. To do that I ran the 'ps' process right as the computation finishes. The call to `ps_stat()` was inserted immediately before the `method()` call returns, so as to measure the process size before the garbage collector starts destroying any objects local to that function. I ran code slightly modified from what you see above - the computation result was stored in a string variable whist `ps_stat()` ran. My implementation of `ps_stat()` uses split to separate the fields returned by ps and selects the virtual memory size by field number. The value 15 would probably need to be changed for different versions of ps.

The full code I used is available here.

```
from cStringIO import StringIO
import timing, commands, os
from sys import argv

# .....
# method definitions go here
# .....

def ps_stat():
        global process_size
        ps = commands.getoutput('ps -up ' + `pid`)
        process_size = ps.split()[15]

def call_method(num):
        global process_size
        timing.start()
        z = eval('method' + str(num))()
        timing.finish()
        print "method", num
        print "time", float(timing.micro()) / 1000000
        print "output size ", len(z) / 1024, "kb"
        print "process size", process_size, "kb"
        print

loop_count = 500000
pid = os.getpid()

call_method(argv[1])
```

xrange vs. range

I tried using range instead of xrange to pre-calculate the list of numbers. Somewhat surprisingly range ended up being slightly faster in every case. The difference was about 1% in the fastest methods. I don't think this is a good justification for using range in place of xrange in general though, since range does require more memory, and that's more likely to become a problem than the additional 1% time of traversing the xrange.

Armin Rigo has recently argued that xrange could be eliminated as a separate language construct if the interpreter were smart enough to return an object that uses the appropriate backing storage (iterator or list) depending on the context. I find this argument compelling from a language design perspective, although I have no idea how hard to implement such an optimization would be.

Future Work

I'd love to do a comparison of other programming languages on this same task. The obvious cases would be perl, php, Java and C. It would also be interesting to run a series of tests using increasing string sizes and graph those results.