Python Patterns - An Optimization Anecdote

The other day, a friend asked me a seemingly simple question: what's the best way to convert a list of integers into a string, presuming that the integers are ASCII values. For instance, the list [97, 98, 99] should be converted to the string 'abc'. Let's assume we want to write a function to do this.

The first version I came up with was totally straightforward:

```
def f1(list):
    string = ""
    for item in list:
        string = string + chr(item)
    return string
```

That can't be the fastest way to do it, said my friend. How about this one:

```
def f2(list):
    return reduce(lambda string, item: string + chr(item), list, "")
```

This version performs exactly the same set of string operations as the first one, but gets rid of the for loop overhead in favor of the faster, implied loop of the reduce() function.

Sure, I replied, but it does so at the cost of a function call (the lambda function) per list item. I betcha it's slower, since function call overhead in Python is bigger than for loop overhead.

(OK, so I had already done the comparisons. f2() took 60% more time than f1(). So there :-)

Hmm, said my friend. I need this to be faster. OK, I said, how about this version:

```
def f3(list):
    string = ""
    for character in map(chr, list):
        string = string + character
    return string
```

To both our surprise, f3() clocked *twice* as fast as f1()! The reason that this surprised us was twofold: first, it uses more storage (the result of map(chr, list) is another list of the same length); second, it contains two loops instead of one (the one implied by the map() function, and the for loop).

Of course, space versus time is a well-known trade-off, so the first one shouldn't have surprised us. However, how come two loops are faster than one? Two reasons.

First, in f1(), the built-in function chr() is looked up on every iteration, while in f3() it is only looked up once (as the argument to map()). This look-up is relatively expensive, I told my friend, since Python's dynamic scope rules mean that it is first looked up (unsuccessfully) in the current module's global dictionary, and then in the dictionary of built-in function (where it is found). Worse, unsuccessful dictionary lookups are (on average) a bit slower than successful ones, because of the way the hash chaining works.

The second reason why f3() is faster than f1() is that the call to chr(item), as executed by the bytecode interpreter, is probably a bit slower than when executed by the map() function - the bytecode interpreter must execute three bytecode instructions for each call (load 'chr', load 'item', call), while the map() function does it all in C.

This led us to consider a compromise, which wouldn't waste extra space, but which would speed up the lookup for the chr() function:

```
def f4(list):
    string = ""
    lchr = chr
```

```
        for item in list:
            string = string + lchr(item)
        return string
```

As expected, f4() was slower than f3(), but only by 25%; it was about 40% faster than f1() still. This is because local variable lookups are *much* faster than global or built-in variable lookups: the Python "compiler" optimizes most function bodies so that for local variables, no dictionary lookup is necessary, but a simple array indexing operation is sufficient. The relative speed of f4() compared to f1() and f3() suggests that both reasons why f3() is faster contribute, but that the first reason (fewer lookups) is a bit more important. (To get more precise data on this, we would have to instrument the interpreter.)

Still, our best version, f3(), was only twice as fast as the most straightforward version, f1(). Could we do better?

I was worried that the quadratic behavior of the algorithm was killing us. So far, we had been using a list of 256 integers as test data, since that was what my friend needed the function for. But what if it were applied to a list of two thousand characters? We'd be concatenating longer and longer strings, one character at a time. It is easy to see that, apart from overhead, to create a list of length N in this way, there are 1 + 2 + 3 + ... + (N-1) characters to be copied in total, or (N-1)*(N-2), or N**2 - 3N + 2. In addition to this, there are N string allocation operations, but for sufficiently large N, the term N**2 will take over. Indeed, for a list that's 8 times as long (2048 items), these functions all take much more than 8 times as long; close to 16 times as long, in fact. I didn't dare try a list of 64 times as long.

There's a general technique to avoid quadratic behavior in algorithms like this. I coded it as follows for strings of exactly 256 items:

```
def f5(list):
    string = ""
    for i in range(0, 256, 16): # 0, 16, 32, 48, 64, ...
        s = ""
        for character in map(chr, list[i:i+16]):
            s = s + character
        string = string + s
    return string
```

Unfortunately, for a list of 256 items, this version ran a bit slower (though within 20%) of f3(). Since writing a general version would only slow it down more, we didn't bother to pursue this path any further (except that we also compared it with a variant that didn't use map(), which of course was slower again).

Finally, I tried a radically different approach: use *only* implied loops. Notice that the whole operation can be described as follows: apply chr() to each list item; then concatenate the resulting characters. We were already using an implied loop for the first part: map(). Fortunately, there are some string concatenation functions in the string module that are implemented in C. In particular, string.joinfields(list_of_strings, delimiter) concatenates a list of strings, placing a delimiter of choice between each two strings. Nothing stops us from concatenating a list of characters (which are just strings of length one in Python), using the empty string as delimiter. Lo and behold:

```
import string
def f6(list):
    return string.joinfields(map(chr, list), "")
```

This function ran four to five times as fast as our fastest contender, f3(). Moreover, it doesn't have the quadratic behavior of the other versions.

And The Winner Is...

The next day, I remembered an odd corner of Python: the array module. This happens to have an operation to create an array of 1-byte wide integers from a list of Python integers, and every array can be written to a file or converted to a string as a binary data structure. Here's our function implemented using these operations:

```
import array
def f7(list):
```

```
        return array.array('b', list).tostring()
```

This is about three times as fast as f6(), or 12 to 15 times as fast as f3()! it also uses less intermediate storage - it only allocates 2 objects of N bytes (plus fixed overhead), while f6() begins by allocating a list of N items, which usually costs 4N bytes (8N bytes on a 64-bit machine) - assuming the character objects are shared with similar objects elsewhere in the program (like small integers, Python caches strings of length one in most cases).

Stop, said my friend, before you get into negative times - this is fast enough for my program. I agreed, though I had wanted to try one more approach: write the whole function in C. This could have minimal storage requirements (it would allocate a string of length N right away) and save a few instructions in the C code that I knew were there in the array module, because of its genericity (it supports integer widths of 1, 2, and 4 bytes). However, it wouldn't be able to avoid having to extract the items from the list one at a time, and to extract the C integer from them, both of which are fairly costly operations in the Python-C API, so I expected at most modest speed up compared to f7(). Given the effort of writing and testing an extension (compared to whipping up those Python one-liners), as well as the dependency on a non-standard Python extension, I decided not to pursue this option...

Conclusion

If you feel the need for speed, go for built-in functions - you can't beat a loop written in C. Check the library manual for a built-in function that does what you want. If there isn't one, here are some guidelines for loop optimization:

- *Rule number one:* only optimize when there is a proven speed bottleneck. Only optimize the innermost loop. (This rule is independent of Python, but it doesn't hurt repeating it, since it can save a lot of work. :-)
- Small is beautiful. Given Python's hefty charges for bytecode instructions and variable look-up, it rarely pays off to add extra tests to save a little bit of work.
- Use intrinsic operations. An implied loop in map() is faster than an explicit for loop; a while loop with an explicit loop counter is even slower.
- Avoid calling functions written in Python in your inner loop. This includes lambdas. In-lining the inner loop can save a lot of time.
- Local variables are faster than globals; if you use a global constant in a loop, copy it to a local variable before the loop. And in Python, function names (global or built-in) are also global constants!
- Try to use map(), filter() or reduce() to replace an explicit for loop, but only if you can use a built-in function: map with a built-in function beats for loop, but a for loop with in-line code beats map with a lambda function!
- Check your algorithms for quadratic behavior. But notice that a more complex algorithm only pays off for large N - for small N, the complexity doesn't pay off. In our case, 256 turned out to be small enough that the simpler version was still a tad faster. Your mileage may vary - this is worth investigating.
- And last but not least: collect data. Python's excellent profile module can quickly show the bottleneck in your code. if you're considering different versions of an algorithm, test it in a tight loop using the time.clock() function.

By the way, here's the timing function that I used. it calls a function f n*10 times with argument a, and prints the function name followed by the time it took, rounded to milliseconds. The 10 repeated calls are done to minimize the loop overhead of the timing function itself. You could go even further and make 100 calls... Also note that the expression range(n) is calculated outside the timing brackets - another trick to minimize the overhead caused by the timing function. If you are worried about this overhead, you can calibrate it by calling the timing function with a do-nothing function.

```
    import time
    def timing(f, n, a):
        print f.__name__,
        r = range(n)
        t1 = time.clock()
        for i in r:
            f(a); f(a); f(a); f(a); f(a); f(a); f(a); f(a); f(a); f(a)
```

```
        t2 = time.clock()
        print round(t2-t1, 3)
```

Epilogue

A few days later, my friend was back with the question: how do you do the reverse operation? I.e. create a list of integer ASCII values from a string. Oh no, here we go again, it flashed through my mind...

But this time, it was relatively painless. There are two candidates, the obvious:

```
    def g1(string):
        return map(ord, string)
```

and the somewhat less obvious:

```
    import array
    def g2(string):
        return array.array('b', string).tolist()
```

Timing these reveals that g2() is about five times as fast as g1(). There's a catch though: g2() returns integers in the range -128..127, while g1() returns integers in the range 0..255. If you need the positive integers, g1() is going to be faster than anything postprocessing you could do on the result from g2().

Sample Code

- timing f1() through f7()
- timing g1() and g2()