Python Performance

Parity Testing

```
from timeit import Timer


t1 = Timer("for i in xrange(100): i % 2")
t2 = Timer("for i in xrange(100): i & 1")
# The "not" tests show what happens when interpreting
# the result as a boolean
t3 = Timer("for i in xrange(100): not i % 2")
t4 = Timer("for i in xrange(100): not i & 1")

print "Checking for odd parity with `mod`:\t%.4f" % t1.timeit()
print "Checking for odd parity with `and`:\t%.4f" % t2.timeit()
print "Checking for even parity with `mod`:\t%.4f" % t3.timeit()
print "Checking for even parity with `and`:\t%.4f" % t4.timeit()
```

MacPython 2.7.2

```
Checking for odd parity with `mod`:    6.5617
Checking for odd parity with `and`: 5.3778
Checking for even parity with `mod`:    8.4417
Checking for even parity with `and`:    7.4086
```

PyPy 1.6.0 (with GCC 4.0.1; Python 2.7.2)

```
Checking for odd parity with `mod`:    0.2556
Checking for odd parity with `and`: 0.2312
Checking for even parity with `mod`:    1.7576
Checking for even parity with `and`:    0.6614
```

The results for odd parity were murky. Sometimes mod was slightly faster; sometimes bitwise-and was faster. There was no question with the even parity, however: The bitwise-and operator played much more nicely with the `not` operator than did the mod operator.

Jython 2.5.2

```
Checking for odd parity with `mod`:    3.4480
Checking for odd parity with `and`: 1.9380
Checking for even parity with `mod`:    3.6050
Checking for even parity with `and`:    2.0440
```

Tuple Unpacking

```
    from timeit import Timer

    index1 = Timer("x = tpl[0]", "tpl = (5,)")
    unpack1 = Timer("x, = tpl", "tpl = (5,)")
    index2 = Timer("x = tpl[1]", "tpl = (5, 6)")
    unpack2 = Timer("y, x = tpl", "tpl = (5, 6)")
    index3 = Timer("x = tpl[2]", "tpl = (5, 6, 7)")
    unpack3 = Timer("y, y, x = tpl", "tpl = (5, 6, 7)")
    index4 = Timer("x = tpl[3]", "tpl = (5, 6, 7, 8)")
    unpack4 = Timer("y, y, y, x = tpl", "tpl = (5, 6, 7, 8)")

    list_index2 = Timer("[tpl[1] for tpl in tuples]", "tuples = [(i, i * i) for i in xrange(100)]")
    list_unpack2 = Timer("[y for x, y in tuples]", "tuples = [(i, i * i) for i in xrange(100)]")
    list_map2 = Timer("map(itemgetter(1), tuples)", "tuples = [(i, i * i) for i in xrange(100)]; from operator import
itemgetter")

    times = 100000000

    print "Indexing vs. unpacking a 1-tuple:\t%.4f\t%.4f" % (index1.timeit(number=times),
unpack1.timeit(number=times))
    print "Indexing vs. unpacking a 2-tuple:\t%.4f\t%.4f" % (index2.timeit(number=times),
unpack2.timeit(number=times))
    print "Indexing vs. unpacking a 3-tuple:\t%.4f\t%.4f" % (index3.timeit(number=times),
unpack3.timeit(number=times))
    print "Indexing vs. unpacking a 4-tuple:\t%.4f\t%.4f" % (index4.timeit(number=times),
unpack4.timeit(number=times))
    print "Indexing vs. unpacking a list of 2-tuples:\t%.4f\t%.4f" % (list_index2.timeit(), list_unpack2.timeit())
    print "map() and itemgetter() (just for kicks):\t%.4f" % (list_map2.timeit())
```

MacPython 2.7.2

```
Indexing vs. unpacking a 1-tuple:  5.0712  3.3939
Indexing vs. unpacking a 2-tuple:   5.7888  6.2801
Indexing vs. unpacking a 3-tuple:   6.1820  7.5976
Indexing vs. unpacking a 4-tuple:   7.1802  7.8219
Indexing vs. unpacking a list of 2-tuples:  8.6561  8.3513
map() and itemgetter() (just for kicks):    9.1651
```

Unpacking is slightly faster for a tuple of a single item. This happens more often than you might thing; consider, for example, `struct.unpack(">H")`, which returns a tuple. Thus, use `val, = struct.unpack(">H")` in these situations instead of `val = struct.unpack(">H")[0]`. That said, use with care, since tuple unpacking is also slightly more unreadable than indexing, and so it does not seem that tuple unpacking causes a bottleneck for our software... yet. As the tuple grows, however, indexing is always faster. Also, as one might have suspected, `itemgetter` works more slowly than a list comprehension.

PyPy 1.6.0 (with GCC 4.0.1; Python 2.7.2)

```
Indexing vs. unpacking a 1-tuple:  0.2268  0.2279
Indexing vs. unpacking a 2-tuple:   0.2301  0.2302
Indexing vs. unpacking a 3-tuple:   0.2335  0.2320
Indexing vs. unpacking a 4-tuple:   0.2332  0.2344
Indexing vs. unpacking a list of 2-tuples:  1.2610  1.2698
map() and itemgetter() (just for kicks):    5.4586
```

There is no clear difference in pypy; both the indexing and unpacking operations seem to vary constantly. (I tested informally using a 250 item tuple. My test with 1000 slowed down unpacking considerably, but I suspect the bottleneck was with the source code parser, not the operation itself.) It is clear that map and itemgetter are significantly slower for pypy, however.

Jython 2.5.2

```
Indexing vs. unpacking a 1-tuple:  0.6510  1.1520
Indexing vs. unpacking a 2-tuple:   0.9610  0.7800
Indexing vs. unpacking a 3-tuple:   0.8930  0.8330
Indexing vs. unpacking a 4-tuple:   1.0250  0.8070
Indexing vs. unpacking a list of 2-tuples:  36.4800 40.4600
map() and itemgetter() (just for kicks):    11.0170
```

The Jython results varied from run to run, but it looks like unpacking was almost always faster. It also looks like Jython does not handle list comprehensions very well. Now you know.

Powers of Two

```
from timeit import Timer

# Use 62 to prevent slowdown from long ints
t1 = Timer("for i in xrange(62): 1 << i")
t2 = Timer("for i in xrange(62): 2 ** i")

times = 1000000

print "Bit-shifting vs. Exponentation:\t%.4f\t%.4f" % (t1.timeit(number=times), t2.timeit(number=times))
```

MacPython 2.7.2

```
Bit-shifting vs. Exponentation:    3.8654  8.3995
```

Bit-shifting wins by a longshot.

PyPy 1.6.0 (with GCC 4.0.1; Python 2.7.2)

```
Bit-shifting vs. Exponentation:    0.2184  2.0279
```

Again, bit-shifting wins by a longshot.

Jython 2.5.2

```
Bit-shifting vs. Exponentation:    2.6870  17.0960
```

Don't use Jython.

Revision #2
Created 23 March 2023 14:31:02 by Nick Dobson
Updated 24 March 2023 13:46:57 by Nick Dobson