

# Python Optimization

By Imri Goldberg  
[www.algorithm.co.il](http://www.algorithm.co.il)  
[plnr.com](http://plnr.com)

# Introduction - the cliches

---

- ▶ Python is slow...
- ▶ ...but is fast enough!
  
- ▶ Usually we don't care about speed, until we do.
- ▶ Don't do premature optimization
- ▶ There are only two hard things in computer science:
  - ▶ cache invalidation and naming things
  - ▶ *-Phil Karlton*



# The basics

---

- ▶ Profile first, ask questions later.

For example:

```
import profile

def main():
    print 'running'

if __name__ == '__main__':
    profile.run('main()', 'out.profile')
```

- ▶ Then later:

```
import pstats

p = pstats.Stats(stats_filename)
p = p.strip_dirs()
p = p.sort_stats('cumulative')
p.print_stats()
```



# The basics, cont.

---

- ▶ Create a benchmark
- ▶ Unit tests as benchmarks may be fine, as long as they are representative
- ▶ Multiple benchmarks are a good idea:
  - ▶ A short one that runs in a few seconds
  - ▶ A longer one that runs for a longer time but is more representative
- ▶ Use the following heuristic until happy:
  1. Profile
  2. Where do we spend too much time?
  3. Optimize wasteful part
  4. GOTO 1.
- ▶ Be sure to have unit-tests, to make sure your code is still correct!



# Low hanging fruit

---

## ▶ Pysco:

```
import pysco  
pysco.full()
```

## ▶ Prefer faster libraries

- ▶ numpy

- ▶ lxml

- ▶ ...



# The C issue

---

- ▶ The saying goes:  
"You can always rewrite the slow parts in C".
- ▶ I've done that very very very rarely.
- ▶ I prefer to keep my code in an easily refactor-able form.
- ▶ Still, yes, you can do that.
- ▶ There are many modules to help you. I didn't use them...
  - ▶ Still, check out pyrex, swig, etc...
- ▶ ctypes is also very easy to use.



# The small time eater

---

- ▶ When we sort according to cumulative time, we get functions sorted according to the time they and all their children in the callgraph took.
- ▶ We would prefer a list that looks like a topological sort of the callgraph, meaning:
- ▶ If a function takes more time - it's higher in the call chain.
- ▶ When you get a low-level function jumping up - that's a place where you can optimize!



# Small time eater - example

---

- ▶ I ran a profiler on a unit-test of mine "for the lulz":

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
      1   0.000   0.000   71.502   71.502  profile:0(nose.main(argv = sys.argv))
    559/1   0.003   0.000   71.502   71.502  <string>:1(<module>)
      1   0.000   0.000   71.502   71.502  core.py:182(__init__)
      1   0.000   0.000   71.497   71.497  unittest.py:752(__init__)
[... snipped long list ... ]
    4261   0.139   0.000   13.647   0.003  main.py:1181(__init__)
   1473   0.199   0.000   13.141   0.009  main.py:1236(_create)
      3   0.004   0.001   12.560   4.187  joiner.py:44(add_csv_to_db)
      3   0.243   0.081   12.498   4.166  joiner.py:89(add_data_to_db)
    4261   0.162   0.000   12.229   0.003  main.py:912(_init)
   69663   0.958   0.000   10.673   0.000  main.py:875(get)
   46168   0.418   0.000    9.913   0.000  dbconnection.py:605(sqlrepr)
146724/46177  1.251   0.000    9.154   0.000  converters.py:190(sqlrepr)
   1473   0.111   0.000    8.924   0.006  main.py:1271(_SO_finishCreate)
    723   0.160   0.000    8.587   0.012  combiner.py:13(handle_group)
  729930   5.514   0.000    8.362   0.000  main.py:939(_SO_loadValue)
   3814   0.818   0.000    7.884   0.002  dbconnection.py:529(_SO_selectOneAlt)
237360 2.489 0.000 7.806 0.000 decimal.py:1404(__float__)
[... snipped long list ... ]
```



# Small time eater - example

---

- ▶ from decimal.py:

```
def __float__(self):  
    """Float representation."""  
    return float(str(self))
```

- ▶ No wonder it's slow...
- ▶ Sometimes, these time eaters can get very high in the list.
- ▶ Only then they are worth the trouble. In this case, this function takes 10% of total time. A 50% reduction in its running time will yield a reduction of 5% of the total time.
- ▶ This might be worth it depending on situation.
- ▶ I found "small time eaters" in many real-life problems.



# Various techniques

---

- ▶ Python dicts are fast, use that to your advantage.
- ▶ Loop avoidance:
- ▶ Use sets. Instead of :

```
for x in a:  
    for y in b:  
        if x == y:  
            yield (x,y)
```

- ▶ Write:

```
return set(a) & set(b)
```

- ▶ Sometimes you can use built-in loops. Some built-in loops exist in the re module, and the struct module. For example, this:

```
struct.pack('B'*len(data), *data)
```

- ▶ Is faster than this:

```
".join(chr(x) for x in data)
```

---



# Avoid unnecessary recursion

---

- ▶ This is a bit rare in real-life programs, but I encountered ones similar to it.

```
def fib(n):  
    if n == 0 or n == 1:  
        return 1  
    return fib(n-1) + fib(n-2)
```

- ▶ **Use caching:**

```
def fib2(n):  
    if n == 0 or n == 1:  
        return 1  
    if n in fib2._cache:  
        return fib2._cache[n]  
    result = fib2(n-1) + fib2(n-2)  
    fib2._cache[n] = result  
    return result
```

```
fib2._cache = {}
```

- ▶ **Don't forget to avoid memory leaks!**
- 



# Algorithm

---

- ▶ If you have no cheap shots, it's time to reconsider your design and/or algorithm.
- ▶ At the least estimate your complexity, and lookup existing algorithms that solve the problem.
- ▶ If that fails, write one yourself...
- ▶ If that fails, use an oracle, they have lower complexity requirements :)



## More tips

---

- ▶ Avoid blocking IO in time critical code.
- ▶ In many cases, I didn't need the "full blown" computation, and an estimate was "good enough".
- ▶ When dealing with very large inputs, write your codes in "generator form". Cache only where necessary. That way you can also control memory requirements.
- ▶ Sometimes that means writing output to disk as soon as possible.



# Further Reading

---

- ▶ <http://wiki.python.org/moin/PythonSpeed/PerformanceTips>
- ▶ <http://www.algorithm.co.il/blogs/index.php/programming/python/10-python-optimization-tips-and-issues/>
- ▶ <http://www.dabeaz.com/generators/index.html>

