Slides from INF3331 lectures – optimizing Python code

Ola Skavhaug, Joakim Sundnes and Hans Petter Langtangen

Dept. of Informatics, Univ. of Oslo

&

Simula Research Laboratory

August 2011

Python optimization

Contents

- Timing and profiling.
- Simple Python tricks.
- Vectorization and mixed-language programming.

Optimization of C, C++, and Fortran

- Compilers do a good job for C, C++, and Fortran.
- The type system makes agressive optimization possible.
- Examples: code inlining, loop unrolling, and memory prefetching.

Python optimization

- No compiler.
- No type declaration of variables.
- No inlining and no loop unrolling.
- Probably inefficient in Python:

```
def f(a, b):
    return a + b
```

Manual timing

- Use `time.time()`.
- Simple statements should be placed in a loop.
- Make sure constant machine load.
- Run the tests several times, choose the fastest.

The `timeit` module (1)

- Usage:

```
import timeit
timer =
timeit.Timer(stmt="a+=1",setup="a=0")
time = timer.timeit(number=10000) #or
times = timer.repeat(repeat=5,
number=10000)
```

The `timeit` module (2)

- Isolates the global namespace.
- Automatically wraps the code in a for–loop.
- Users can provide their own timer (callback).
- Time a user defined function:

```
timer = timeit.Timer(stmt=”myfunc()",
setup="from __main__ import my_func")
```

Profiling modules

- Prior to code optimization, hotspots and bottlenecks must be located.

  "First make it work. Then make it right. Then make it fast."

  - Kent Beck
- Two main modules: `profile` and `cProfile` (`hotshot` is no longer maintained).
- `profile` works for all Python versions.
- `cProfile` introduced in Python version 2.5.

The `profile` module (1)

- As a script: `profile.py script.py`
- As a module:

```
import profile
pr = profile.Profile()
res = pr.run("function()", "filename")
res.print_stats()
```

- Profile data saved to "filename" can be viewed with the `pstats` module.

The `profile` module (2)

- `profile.calibrate(number)` finds the profiling overhead.
- Remove profiling overhead:
  `pr = profile.Profile(bias=overhead)`
- Profile a single function call:

```
pr = profile.Profile()
pr.runcall(func, *args, **kwargs)
```

The `cProfile` module (recommended)

- Similar to `profile`, but mostly implemented in C.
- Smaller performance impact than `profile`.
- Useage:

```
import cProfile
cProfile.run('foo()', 'fooprof')
```

or to profile a script:

```
python -m cProfile my_script.py
```

The `pstats` module

- There are many ways to view profiling data.
- The module `pstats` provides the class `Stats` for creating profiling reports:

```
import pstats
data = pstats.Stats("fooprof")
data.print_stats()
```

- The method `sort_stats(key, *keys)` is used to sort future output.
- Common used keys: `'calls', 'cumulative', 'time'`.

Pure Python performance tips

- Place references to functions in the local namespace.

```
from math import *
def f(x):
    for i in xrange(len(x)):
        x[i] = sin(x[i]) # Slow
    return x

def g(x):
    loc_sin = sin # Local reference
    for i in xrange(len(x)):
        x[i] = loc_sin(x[i]) # Faster
    return x
```

- Reason: Local namespace is searched first.

More local references

- Local references to instance methods of global objects are even more important, as we need only one dictionary look–up to find the method instead of three (local, global, instance–dictionary).

```
class Dummy(object):
    def f(self): pass

d = Dummy()

def f():
    loc_f=d.f
    for i in xrange(10000): loc_f()
```

- Calling `loc_f()` instead of `d.f()` is 40% faster in this example.

Exceptions should never happen

- Use `if/else` instead of `try/except`
- Example:

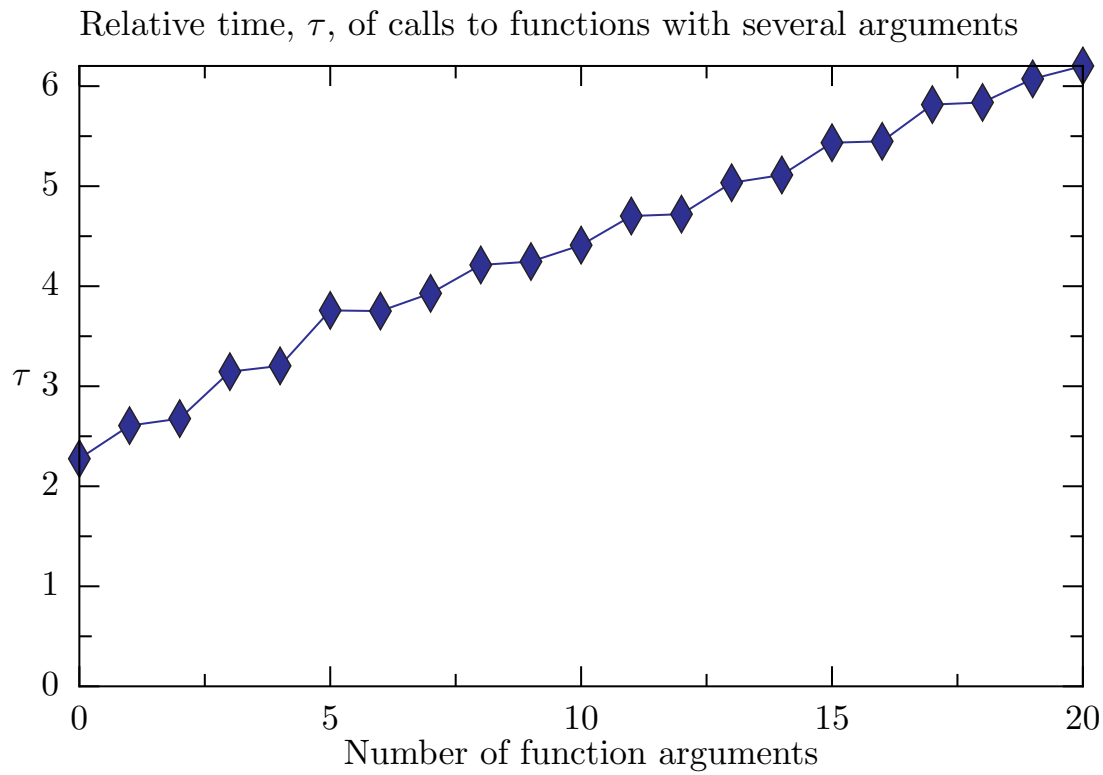
```
x = 0
try: 1.0/x
except: 0

if not (x==0): 1.0/x
else: 0
```

- `if/else` is more than 20 times faster.

Function calls

- The time of calling a function grows linearly with the number of arguments:



Numerical Python

- Vectorized computations are fast:

```
import numpy
x = numpy.arange(-1,1,0.01)
y = numpy.sin(x)
```

```
import math # Scalar functions
y = numpy.zeros(len(x))
for i in xrange(len(x)):
    y[i] = math.sin(x[i])
```

- The speedup is a factor of 20.

Resizing arrays

- The `resize` method of arrays is very slow.
- Increasing the array size by one in a loop is about 300-350 times slower than appending elements to a Python list.
- Best approach; allocate the memory once, and assign values later.

Conclusions

- Python scripts can often be heavily optimized.
- The results given here may vary on different architectures and Python versions
- Be extremely careful about the `from numpy import *`. For scalar arguments, functions from the `math` module are much faster than the corresponding `numpy` functions.
- Vectorized computations can achieve similar efficiency as optimized compiled language code.
- Time-critical operations that cannot be vectorized must be ported to a compiled language.