LECTURE 20

Optimizing Python

THE NEED FOR SPEED

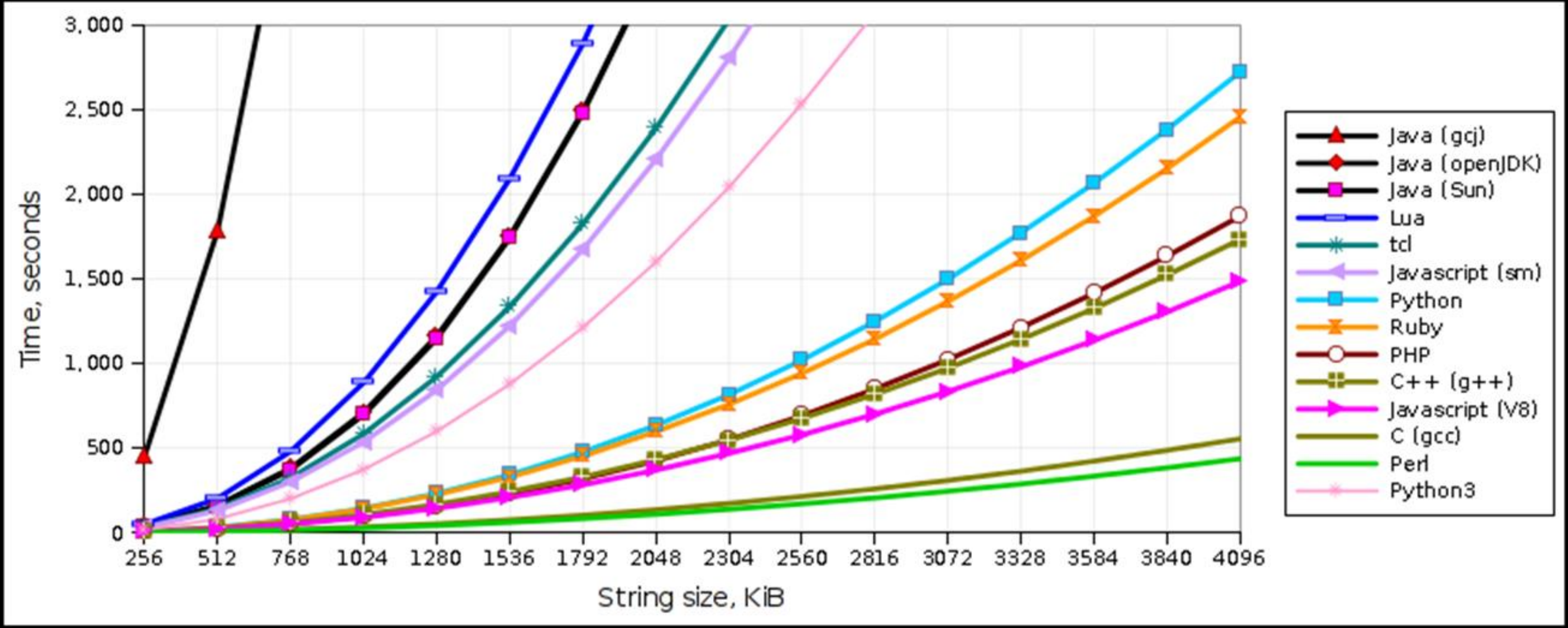
By now, hopefully I've shown that Python is an extremely versatile language that supports quick and easy development. However, a lot of the nice features that make it pleasant to develop with have a high cost behind the scenes.

As a result, one of Python's major drawbacks is its speed. Even for activities at which Python excels, like string manipulation, Python falls way behind in the category of "faster" languages.

For a particular String Manipulation Benchmark, the following time results were achieved for a 4096KB string size: 0:07:17 (Perl), 0:31:09 (PHP), 0:40:55 (Ruby), 0:45:20 (Python), 0:28:51 (C++), 0:09:15 (C).

STRING MANIPULATION BENCHMARK

Source: http://raid6.com.au/~onlyjob/posts/arena/



CYTHON AND NUMBA

So, what can be done? Aside from tiny improvements that can be made here-and-there within the code itself, we can also use compiling methods to speed up our code.

Two options are:

- Cython, an optimizing static compiler as well as a compiled language which generates Python modules that can be used by regular Python code.
- Numba, a Numpy-aware optimizing just-in-time compiler.

```
$ sudo apt-get install build-essential
$ sudo apt-get install llvm llvm-3.3 llvm-3.3-runtime
$ sudo sh -c "LLVM_CONFIG_PATH=/usr/bin/llvm-config-3.3 pip install llvmpy"
$ pip install cython
$ pip install numba
```

CYTHON

In the simplest of terms: Cython is Python with C data types.

Almost any piece of Python code is also valid Cython code, which the Cython compiler will convert into C code.

CYTHON

Cython code must, unlike Python, be compiled. This happens in two stages:

- A .pyx file is compiled by Cython to a .c file, containing the code of a Python extension module.
- The .c file is compiled by a C compiler to a .so file (or .pyd on Windows) which can be imported directly into a Python session.

HELLO, WORLD!

The basic steps to compiling a Cython extension are as follows:

1. In helloworld.pyx: `print "Hello, World!`

2. Create setup.py, your python "makefile".

```
from distutils.core import setup
from Cython.Build import cythonize
setup(
    ext_modules = cythonize("helloworld.pyx")
)
```

3. `$ python setup.py build_ext -inplace` → generates helloworld.so.

4.
```
>>> import helloworld
Hello, World!
```

HELLO, WORLD!

Alternatively, for typical modules that don’t require any extra C libraries, there is the pyximport method.

```
>>> import pyximport
>>> pyximport.install()
>>> import helloworld
Hello, World!
```

STATIC TYPING

One of the main advantages of using Cython is to enforce static typing. By default, Python is obviously dynamically-typed but for performance-critical code, this may be undesirable.

Using static typing allows the Cython compiler to generate simpler, faster C code.

The use of static typing, however, is not “pythonic” and results in less-readable code so you are encouraged to only use static typing when the performance improvements justify it.

BASICS OF CYTHON

The cdef statement is used to declare C variables, as well as C struct, union and enum types.

```
cdef int i, j, k
cdef float f, g[42], *h

cdef struct Node:
    int id
    float size

cdef union Data:
    char *str_data
    float *fl_data

cdef enum Color:
    red, blue, green
```

```
cdef:
    int i, j, k
    float f, g[42], *h

    struct Node:
        int id
        float size

    union Data:
        char *str_data
        float *fl_data

    enum Color:
        red, blue, green
```

BASICS OF CYTHON

Cython supports all built-in C types as well as the special Cython types `bint`, used for C boolean values (int with 0/non-0 values for False/True), and `Py_ssize_t`, for (signed) sizes of Python containers.

Also, the Python types list, dict, tuple, etc. may be used for static typing, as well as any user defined extension types.

STATIC TYPING

Consider the following purely Python code.

```
def f(x):
    return x**2-x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

STATIC TYPING

Consider the following purely Python code.

```
def f(x):
    return x**2-x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

print(timeit.timeit("integrate_f(0.0,
5.0,10000000)", setup="from cydemo import
integrate_f", number=1))
```

Using Python's timeit module, the call integrate_f(0, 5, 100000000) took about 4.198 seconds.

By just compiling with Cython, the call took about 2.137 seconds.

STATIC TYPING

A Cythonic version of this code might look like this:

```
>>> import pyximport
>>> pyximport.install()
>>> import cydemo3
0.663282871246
```

Pure Python code took about 4.198 seconds.
By just compiling with Cython, the call took about 2.137 seconds.
By performing some static typing, the call took about .663 seconds.

```
def f(double x):
    return x**2-x

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit code here
```

CYTHON FUNCTIONS

Python functions are defined using the def statement, as usual. They take Python objects as parameters and return Python objects.

C functions are defined using the cdef statement. They take either Python objects or C values as parameters, and can return either Python objects or C values.

```
def spam(int i, char *s):
    ...

cdef int eggs(unsigned long l, float f):
    ...
```

Behind the scenes:

```
def spam(python_i, python_s):
    cdef int i = python_i
    cdef char* s = python_s
    ...
```

CYTHON FUNCTIONS

Within a Cython module, Python functions and C functions can call each other freely, but only Python functions can be called from outside the module by interpreted Python code. So, any functions that you want to “export” from your Cython module must be declared as Python functions using def.

There is also a hybrid function, called cpdef. A cpdef function can be called from anywhere, but uses the faster C calling conventions when being called from other Cython code.

TYPING FUNCTIONS

When using Cython, Python function calls are extra expensive because one might need to convert to and from Python objects to do the call. We can create some more speedup just by typing our functions.

```
>>> import pyximport
>>> pyximport.install()
>>> import cydemo4
0.0377948284149
```

```
cdef double f(double x):
    return x**2-x

def integrate_f(double a, double b, int N):
    cdef int i
    cdef double s, dx
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit code here
```

SOME RESULTS

- cydemo: pure Python implementation.
- cydemo2: pure Python compiled with Cython.
- cydemo3: static typing.
- cydemo4: static typing and function typing.

module	**N = 10000000**	**N = 100000000**
cydemo	4.198	41.69
cydemo2	2.137	22.74
cydemo3	.663	5.90
cydemo4	.0377	0.382

CYTHON

There is obviously a lot more to Cython but just knowing how to do some static typing and function typing is enough to gain some serious improvements in speed. If you're interested, check here for the Cython documentation.

NUMBA

Numba provides a Just-In-Time compiler for Python code. Just-in-time compilation refers to the process of compiling during execution rather than before-hand. It uses the LLVM infrastructure to compile Python code into machine code.

Central to the use of Numba is the numba.jit decorator.

```
>>> import numbademo
3.98660914803 # N = 10000000
```

numbademo.py

```
@numba.jit
def f(x):
    return x**2-x

def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit function
```

NUMBA JIT DECORATOR

You can also specify the signature of the function. Otherwise Numba will generate separate compiled code for every possible type.

```
>>> import numbademo
0.0191540718079 # N = 10000000
```

```
import numba
from numba import float64, int32

@numba.jit
def f(x):
    return x**2-x

@numba.jit(float64(float64, float64, int32))
def integrate_f(a, b, N):
    s = 0
    dx = (b-a)/N
    for i in range(N):
        s += f(a+i*dx)
    return s * dx

# timeit function
```

SOME RESULTS

- cydemo: pure Python implementation.
- cydemo2: pure Python compiled with Cython.
- cydemo3: static typing.
- cydemo4: static typing and function typing.
- numbademo: jit-compiled functions.

module	**N = 10000000**	**N = 100000000**
cydemo	4.198	41.69
cydemo2	2.137	22.74
cydemo3	.663	5.90
cydemo4	.0377	0.382
numbademo	.0192	0.191

Bottom-line: When it really matters, use Cython or Numba to improve your code's speed. This is not quite a magic wand – these methods increase your dependencies, reduce your readability, and complicate your development.

For simple examples like these, Cython and Numba are not too painful to add in, but they may be a headache for more complicated modules.

See also Dropbox's Pyston, a JIT compiler for Python.