Optimizing Python

pypy, cython, ctypes & more

Blaine Booher
bgbooher@gmail.com
Presented at CincyPy
2.19.12

Introduction

I'm a computer engineer who works with lots of different devices, platforms, and languages.

I use python primarily for my graduate research, but have used it professionally as well.

I've used python for: scientific applications, USB drivers, genetic algorithm research, desktop applications, etc.

How to make python faster?

- Code base: https://github.com/booherbg/Python-Optimizations
- Python is a fun language to use, but it is slow
- There are lots of ways to optimize your code, each with +/-
- Some common ways to "easily" make your code faster:
  - Change your code
    - Profiling to identify bottlenecking
    - Removing lambda functions
    - Unrolling loops, etc.
  - Change your interpreter
    - PyPy (Just-In-Time interpreter, written in python)
    - Unladen Swallow (Developed by Google)
    - Jython (Python-to-Java bytecode)
    - berp (Python-to-Haskell => native executable)
    - Shedskin (Python-to-C++)
  - Change your extension
    - Cython (Ports python into or C/C++, wraps into python extension)
    - Ctypes (Write in C, interface using ctypes)
    - Swig (Mix C and Python for fast modules, interfaces written in C/C++)
    - Use numpy for math & linear algebra
    - Use 'psyco' for python on 32-bit systems

What are we going to cover today?

- Not code optimizations (another day)
- What are the pros and cons of using python?
- And how does this affect the way we use it?
- What is Psyco?
- What is numpy?
- What is PyPy?
- What is Psyco?
- What is Cython?
- What is Ctypes?
- What is Swig?
- What is Unladen Swallow?
- What is Jython?
- What is Berp?

A quick note about python

The neat thing about python is how easy it is to use.

There are some things about python that are great for quick development but make things fuzzy if you want to do some strange things with the implementation.

For example, dynamic typing.

RPython is a small subset of the language with some strict definitions. Once RPython is implemented, you can skip the standard python interpreter and use all kinds of other neat tricks to push the boundaries of what you can do with python.

Jython

Jython is a project that takes python code and translates it into Java bytecode that can be run with the Java Virtual Machine.

The idea is that you can then interface with native Java classes, run it on Java platforms, etc. You should also get the performance of Java as well.

I've never used Jython but I hear good things about it.
There is also a Django-Jython branch. This lets you deploy a Django application to a Java platform.

http://www.jython.org/

Berp

Berp is another esoteric interpreter that has recently been created by a creative hacker.

Berp takes python and converts it into haskell

With haskell, you can run it through the haskell compiler and get a native application. This is extremely experimental but shows what you can do when you implement a minimal python runtime.

I'm not a haskell guy so I can't get all of the packages up and running correctly. Warning: This is a time vampire.

Python 3 only. https://github.com/bjpop/berp/wiki

Shedskin

Shedskin is another small project who aims to take restricted python and port it into C++ that can be compiled.

Not too useful for large projects, but it's a fun one to play with.

Not really recommended unless you're just having fun...

http://code.google.com/p/shedskin/

Unladen Swallow

Unladen Swallow is an interpreter created by Google to help increase python's performance.

It is not highly maintained now that PyPy has taken off. But it still beats out Cpython and is fun to play around with.

It utilizes LLVM and Clang to speed up the python interpreter

http://code.google.com/p/unladen-swallow/

Psyco

Psyco is a simple and incredibly easy way to make your programs fast... but only if you're on a 32-bit operating system.

Simply install the pysco module and do this:

```
import psyco
psyco.profile()
```

Behind the scenes, psyco analyzes your program and pre-compiles routines that seem worth it.

It is no longer maintained and only works on 32-bit systems :(
http://psyco.sourceforge.net

PyPy

PyPy is an interpreter for python.
http://pypy.org
Mission statement
We aim to provide:

- a common translation and support framework for producing implementations of dynamic languages, emphasizing a clean separation between language specification and implementation aspects. We call this the RPython toolchain.
- a compliant, flexible and fast implementation of the Python Language which uses the above toolchain to enable new advanced high-level features without having to encode the low-level details.

What is numpy?

- Numpy is a python module for all kinds of awesome mathematical and scientific applications
- It's a big project but SO EASY to install on most platforms
- Native support for sparse matrices
- Linear Algebra, Statistics, all kinds of fun stuff
- Plotting & Graphical tools (with matplotlib)
- Allows for efficient usage of large matrices
- Very well supported, fast
- A lot of the library is compiled for fast operation

Numpy Pros & Cons

Pros

- Compiled on the backend
- Usable by lots of platforms
- Incredibly large library of functions
- Great support

Cons

- Can't be used with pypy
- Rather large; likely not suitable for small systems
- Needs more research to see if one should distribute

PyPy

- PyPy is a predecessor to psyco
- It is a fascinating project. A python interpreter written in python
- It also supports a Just-In-Time (JIT) compiler
- This means that it dynamically runs faster the longer a program is executed, especially with loops and repetition
- Written in python means that the developers can write the implementation of python with python, allowing for lots of strange hacks. It's python turtles all the way down.
- The pypy interpreter can "bootstrap" itself, running its own python interpreter with itself. Aka Black Magic Voodoo.

In short, this means that PyPy can run your python applications faster. Sometimes much, much faster.

PyPy: Pros and Cons

Pros

- You use it just like cpython. $pypy myprogram.py
- It's fast in lots of cases
- In some cases it can use less memory than cpython
- It's a great resource for finding out about the guts of python
- It can cause dynamic speedup
- You can donate to the developers to create new features

Cons

- Currently it works best on "pure python" applications
- ==> No compiled modules (such as numpy; in the works)
- In fringe cases it may have memory issues on some platforms

Ctypes

Ctypes is a python module that provides a mapping layer between compiled libraries that are not python aware, and the python interpreter

Let's say you have a .dll or .so file that has some functionality you want to use.

Ctypes sets up the interfaces and tells python *how* to use the dynamic library, modify data types to fit into the functions, etc.

Works great with C and Fortran compiled libraries, needs some hacks to work with C++ objects

Ctypes - Example

Here is my amazing routine, written in C

```
// gcc -fPIC -shared myfunc.c -o myfunc.so
int mean(int *vector, int length) {
   int i;
   int sum=0;
   for (i=0; i<length; i++) {
      sum = sum + vector[i];
   }
  return sum/length;
```

This is a pure C module that can be compiled directly to a shared library, native executable.

ctypes (con't)

```
import ctypes

# Load the compiled shared object
so = ctypes.CDLL("./fib.so")

# Set up the interfaces (integers)
so.fib.argtypes=(ctypes.c_int)
so.fib.restype = (ctypes.c_int)

# Create a python mapping
def fib(n):
   return so.fib(n)
```

Ctypes Pros & Cons

Pros
- Built into python
- Can use any kind of compiled library, including fortran
- Makes using existing system libraries possible
- If you're a C hacker you can do magic with it
- You can use *any* compiled library... C, C++, Fortran, Assembler, BrainF!ck, etc.

Cons
- Requires the use of a compiler
- Syntax is a bit unwieldy, especially w/ pointers
- Requires use of C runtime so that you handle memory leaks appropriately

Cython

So, what is cython?

Cython is a project that allows you to write a python module that is translated into a compiled extension.

You write it in pure python (at first), then cython ports into into C or C++, and wraps the python extension framework around it.

In short, it's kind of like ctypes except you write your module in python, not C. And it is a "true" python extension.

Cython (con't)

Pros:

- Makes writing very fast extensions easy
- Write extensions in python, port to C or C++
- Helps to understand *what* python does in the background
- You can end up with a python *and* compiled version of the same code, great for using apps on platforms without compilers

Cons

- Modules now depend on python-dev for extension compilation (because you need python headers to build these "python extensions", as opposed to ctypes where you use a compiled module with no knowledge of python)
- Need a compiler on the system, unless you distribute binaries

Cython example

First, write your module in pure python...

```
#fib_cython.pyx
def fib(n):
    """Calculate Fib lots of times up to N"""
    k = 0
    while k < 1000000:
        a = 0
        b = 1
        while b < n:
            t = b
            b = a + b
            a = t
        k = k+1
    return b

if __name__ == '__main__':
    print fib(1234567890)
```

Cython, con't

Now we have fib.so, a regular compiled python extension.

```
# import the true python extension...
import fib_cython
fib_cython.fib(100)
```

Cython, con't

But wait. There's more!

We can give cython hints about what kinds of datatypes our fib program needs.

```
def fib(int n):
    """Calculate Fib lots of times up to N"""
    cdef int k
    cdef int a
    cdef int b
    cdef int t
    k = 0
    while k < 1000000:
        a = 0
        b = 1
        while b < n:
            t = b
            b = a + b
            a = t
        k = k+1
    return b

#if __name__ == '__main__':
#    print fib(1234567890)
```

Cython (con't)

Our loop is now really fast because cython knows what each data type is and compiles it appropriately into the native C type. It's a beautiful thing, really.

Here let's look at what the difference is when you optimize cython vs letting it use native python objects.

(... opens up gedit)

Now, for some benchmarks

I've written the fibonocci algorithm in both C and Python.

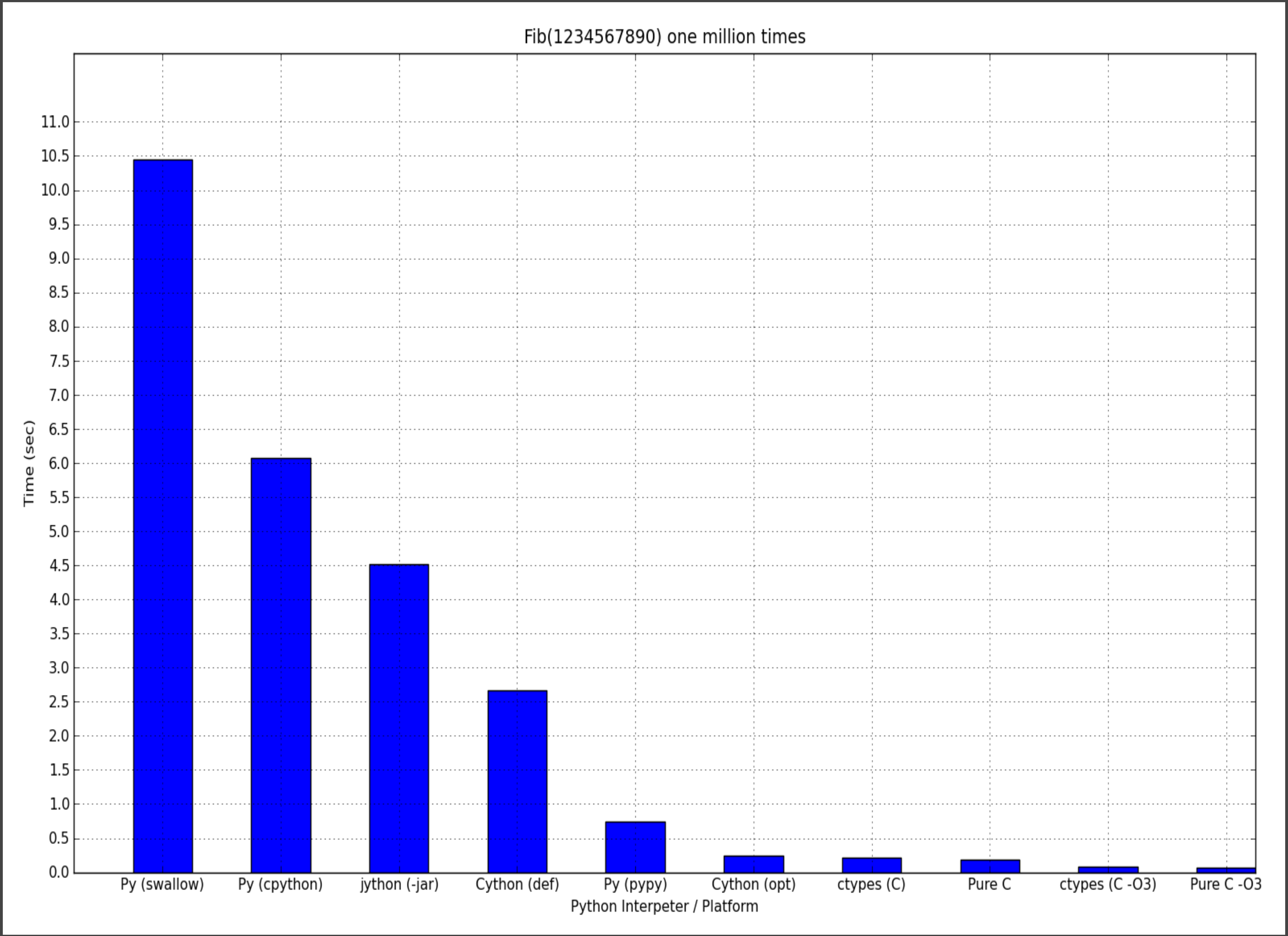
We can now use cython, python, pypy, unladen swallow, compiled C, ctypes, and jython to see how they all stack up.

For the entire software suite, benchmark tool, etc. check out the code base @
https://github.com/booherbg/Python-Optimizations

Final Benchmarks

Here are some final benchmarks for comparison.
Calculate the 1234567890th Fibonacci number one million times. Average of 5 runs.

Benchmark Name	Time Taken
Pure Python (swallow)	10.448 sec
Pure Python (cpython)	6.074 sec
jython (-jar)	4.514 sec
Cython (default)	2.667 sec
Pure Python (pypy)	0.742 sec
Cython (optimized)	0.239 sec
ctypes (C)	0.208 sec
Pure C (Compiled)	0.188 sec
ctypes (C -O3)	0.087 sec
Pure C (Compiled w/ -O3)	0.066 sec


Fib(1234567890) one million times
Time (sec)
11.0
10.5
10.0
9.5
9.0
8.5
8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Py (swallow)
Py (cpython)
jython (-jar)
Cython (def)
Py (pypy)
Cython (opt)
ctypes (C)
Pure C
ctypes (C -O3)
Pure C -O3
Python Interpeter / Platform

Bonus Feature: PyEvolve Genetic Algorithm

Let's look at a genetic algorithm using PyPy vs CPython vs CPython + Cython

I have a cellular automata framework that I've written both in C++ (and interfacing with Cython) and Pure Python (for benchmarking and using with PyPy)

The PyEvolve framework is pure python; PyPy loves it. BUT I can't use my Cython extension with PyPy because it's compiled.

Results

Show it.

In Conclusion...

The main ways I solve optimization:

- ctypes if I have an easy to use, pure C library
- cython if I wish to have a full blown python extension, or if I wish to prototype in python first... or if you have a C++ class
- pypy if I have all pure python and just want to run it quicker
- psyco if I want something simple but only have 32-bit
- If I have a lazy sunday afternoon, try berp, shedskin, unladen swallow
- If you're into java, try jython