How to make Python perform like C

Ron Barak
RonPyConIsrael.comverse@xoxy.net

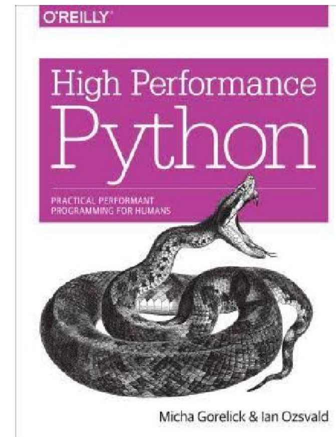
PyCon Israel 2016

Will display techniques and technologies

Will not go in depth into many of them

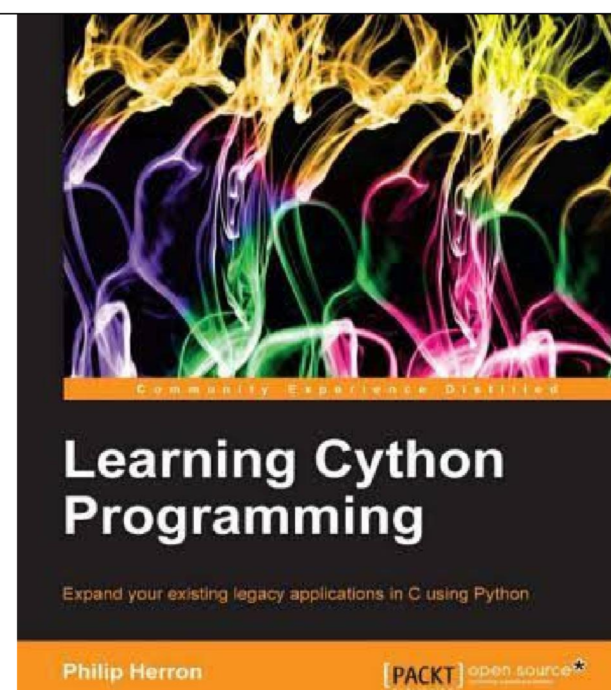
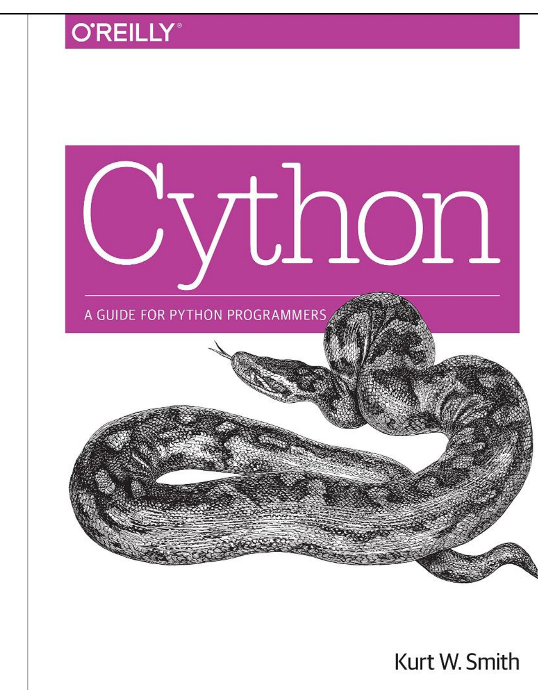
Examples and slides mainly from

Gorelick, Micha, and Ian Ozsvald. *High Performance Python*. 2014. ISBN: 978-1-449-36159-4



https://github.com/mynameisfiber/high_performance_python

Books on Cython

Herron, Philip. Learning Cython Programming. Birmingham: Packt Publishing Limited, 2013. ISBN 978-1-78328-079-7	Smith, Kurt W. Cython. 2015. ISBN: 978-1-491-90155-7
	

How to make code run faster?
Make it do less work.

First:
Use good algorithms
Reduce amount of data to process

Then:
Easiest way to execute fewer instructions
compile code down to machine code

Increase efficiency with specialized code
reduce number of instructions CPU executes.

Algorithmically.

Manually (JIT Versus AOT Compilers).

Use libraries written in other languages.

Python allows us

benefit from the speedups of other languages
(on some problems)

still maintain verbosity and flexibility

Note:

These techniques optimize CPU instructions *only*,
***not* I/O-bound processes (coupled to a CPU-bound problem).**

Compiling code for these (I/O) problems
may not provide any reasonable speedups.
We must rethink our solutions.
Employ parallelism?

Employing CPU and memory profiling

will start you thinking of

higher-level algorithmic optimizations.

https://www.huyng.com/posts/python-performance-analysis

Expected takeaways:

- Python code runs as lower-level code
- Difference between JIT compilers and AOT compilers
- Tasks where compiled Python performs faster than native Python
- Annotations speed up compiled Python code

Python has options:

Pure C/C++ compiling

- Cython
- Shed Skin
- Pythran

LLVM-based compiling

- Numba

Replacement virtual machine

- PyPy (includes built-in just-in-time (JIT) compiler)

Cython

- most commonly used for compiling to C
- both NumPy and normal Python code
- some knowledge of C required

Shed Skin

- automatic Python-to-C++ converter
- non-NumPy code

Numba

- compiler specialized for NumPy code

Pythran

- compiler for both NumPy and normal Python code

PyPy

- just-in-time compiler
- mainly non- NumPy code
- replaces normal Python (CPython) executable

Which is best for you?

Code adaptability
Team velocity

Each tool adds dependencies to tool-chain

If you use

- Python code, batteries-included libraries, no NumPy
    - Cython
    - Shed Skin
    - PyPy

- NumPy
    - Cython
    - Numba
    - Pythran

All support Python 2.7
Some support Python 3.2+

What Speed Gains to Expect?

If your problems may benefit from compiled approach – several orders of magnitude

Will benefit –

- **mathematical code**
- **loops repeating same operations**
- **loops creating temporary objects**

Will not –

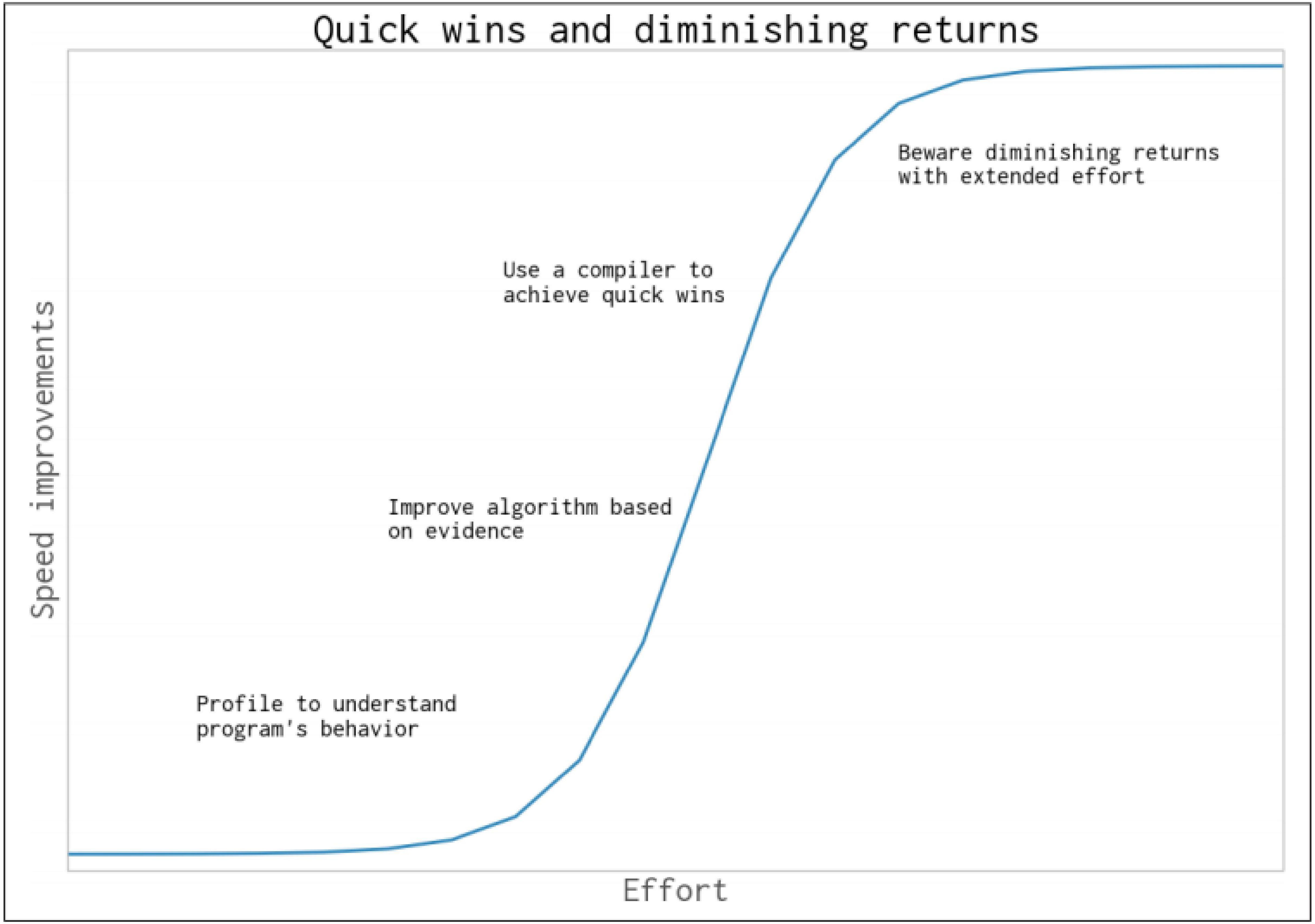
- **calls external libraries (e.g., regex, DB)**
- **I/O-bound**
- **uses vectorised NumPy**

Will not run faster than hand-written C
Will not run much slower than C

- unless C programmer very good
- has intimate knowledge of target architecture

IBM speed comparison of C, Julia, Python, Numba, and Cython (LU Factorization)

https://www.ibm.com/developerworks/community/blogs/jfp/entry/A_Comparison_Of_C_Julia_Python_Numba_Cython_Scipy_and_BLAS_on_LU_Factorization


Quick wins and diminishing returns
Speed improvements
Effort
Profile to understand program's behavior
Improve algorithm based on evidence
Use a compiler to achieve quick wins
Beware diminishing returns with extended effort

Examples will demonstrate

- gains of 1-2 orders of magnitude - on single core
- OpenMP - on multiple cores

Compilers: JIT vs. AOT

compiling ahead of time

- best speedups
- most manual effort

compiling Just-in-time

- impressive speedups
- little (if any) manual intervention
- "cold start" problem

Cython, Shed Skin, Pythran - compiling ahead of time (AOT)
Numba, PyPy - compiling “just in time” (JIT)

Note: Cython is used by *NumPy, SciPy, scikit-learn* to compile parts of their libraries

Type Information Helps Code Run Faster

- Python is dynamically typed
- a variable can refer to an object of any type
- code may change the type of the object that is referred to

Result:
Python virtual machine may struggle to optimize how code should execute at the machine code level.

For instance:

`v` type is either a float or complex
both types may be valid in

- **different parts of same block**
- **or, at same place at different times**

```
$ python
Python 2.7.10 (default, Jun  1 2015, 18:17:45)
[GCC 4.9.2] on cygwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> v = -1.0
>>> print type(v), abs(v)
<type 'float'> 1.0
>>> v = 1-1j
>>> print type(v), abs(v)
<type 'complex'> 1.41421356237
>>> 
```

`abs()` works differently for the float or the complex variable

$$abs(c) = \sqrt{c.real^{2} + c.imag^{2}}$$

Cython

Cythonnn

- Introduces new language type (hybrid of Python and C)
- Team members without knowledge of C
  may struggle supporting this code

In practice - not a big problem:

Cython used only
in well-chosen
limited blocks of code

Cython

- converts type-annotated Python into a compiled extension module
- compiles a module using *`setup.py`* script
- type annotations are C-like (and Pure Python annotations: cython.declare)
- some automated annotation is possible
- extension module may be imported as a regular Python module using import
- Getting started is simple
- learning curve gets steeper with each additional level of complexity and optimization
- best for speeding up calculation-bound functions
- OpenMP support
- mature
- in wide use

OpenMP allows converting parallel problems into multiprocessing-aware modules that run on multiple CPUs on one machine.

Threads are hidden from your Python code (they operate via the generated C code).

http://docs.cython.org/

Example: compiling Python with Cython

https://github.com/mynameisfiber/high_performance_python

Three files:

- ***`cythonfn.pyx`*** **- CPU-bound function to be compiled in a `.pyx` file**
- ***`julia1.py`*** **- calling Python code (Julia code): will call the calculation function**
- ***`setup.py`*** **- instructing Cython to build the extension module**

cythonfn.pyx **(renamed from .py) contains Pure-Python code**

```
$ cat cythonfn.pyx
def calculate_z(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

julia1.py imports the compiled module into the main code

```
$ echo ; echo "..." ; head -3 julia1.py | tail -1 ; echo "..." ; head -10 julia1.py
| tail -1 ; echo "..." ; head -42 julia1.py | tail -6

...
import calculate
...
def calc_pure_python(desired_width, max_iterations):
...
    start_time = time.time()
    output = calculate.calculate_z(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print "Took", secs, "seconds"
```

setup.py **uses Cython to compile the `.pyx` file**

- **On *nix systems – to `.so`**
- **On Windows – to `.pyd` (DLL-like Python library)**
- **On Cygwin – to `.dll`**

```
$ cat setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

# for notes on compiler flags e.g. using
# export CFLAGS=-O2
# so gcc has -O2 passed (even though it doesn't make the code faster!)
# http://docs.python.org/install/index.html

setup(
    cmdclass={'build_ext': build_ext},
    ext_modules=[Extension("calculate", ["cythonfn.pyx"])]
)
```

When *`setup.py`* is run with `build_ext`,
Cython looks for *`cythonfn.pyx`* and builds *`calculate.so`*

```
$ python setup.py build_ext --inplace
running build_ext
cythoning cythonfn.pyx to cythonfn.c
building 'calculate' extension
creating build
creating build/temp.cygwin-2.5.0-i686-2.7
gcc -fno-strict-aliasing -ggdb -O2 -pipe -Wimplicit-function-declaration -fdebug-pref
ix-map=/usr/src/ports/python/python-2.7.10-1.i686/build=/usr/src/debug/python-2.7.10-
1 -fdebug-prefix-map=/usr/src/ports/python/python-2.7.10-1.i686/src/Python-2.7.10=/us
r/src/debug/python-2.7.10-1 -DNDEBUG -g -fwrapv -O3 -Wall -Wstrict-prototypes -I/usr/
include/python2.7 -c cythonfn.c -o build/temp.cygwin-2.5.0-i686-2.7/cythonfn.o
gcc -shared -Wl,--enable-auto-image-base -L. build/temp.cygwin-2.5.0-i686-2.7/cythonf
n.o -L. -L/usr/lib -lpython2.7 -o /home/Administrator/python/pycon/code/high_performa
nce_python/07_compiling/cython/lists/1/calculate.dll
```

`--inplace` instructs Cython to build the compiled module in the current directory (not in *`./build`*)

When build is complete, an intermediate *`cythonfn.c`* and `calculate.so/calculate.dll/calculate.pyd` are created

Note: re-run this step if you change the *`.pyx`* or *`setup.py`* files

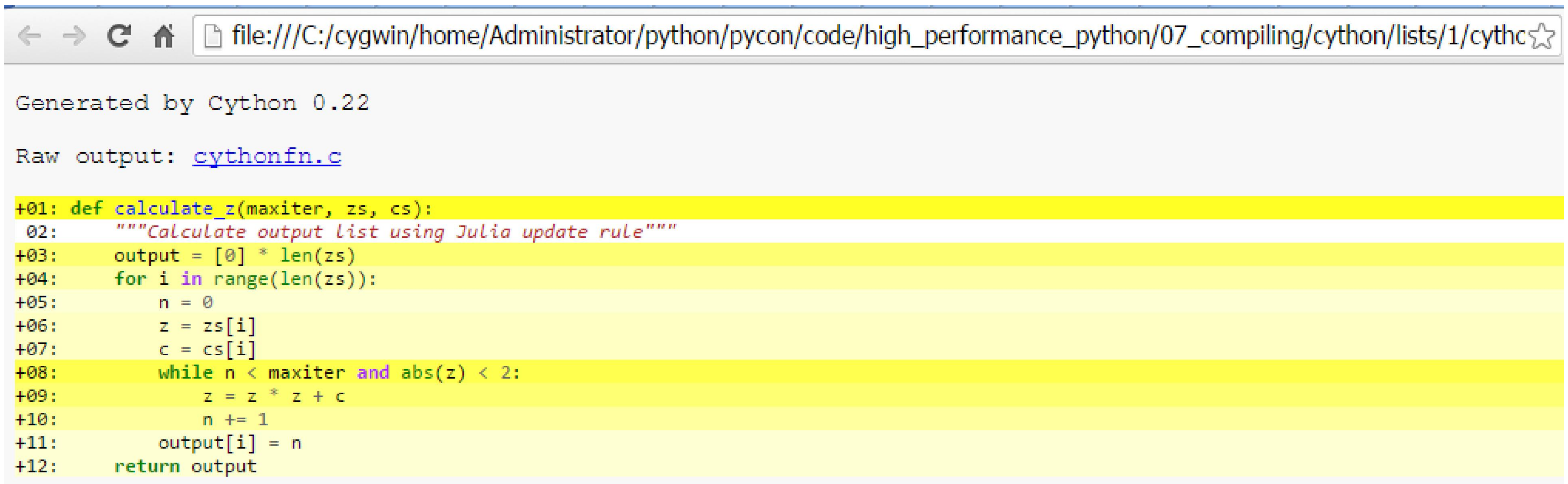
On my machine

- **pure Python code runs in ~26 seconds**
- **Cython code runs in ~16 seconds**

Nice gain for very little effort.

`for` *loops* and *mathematical operations* above could be speeded up

Cython has an annotation option that produces an HTML file (*`cythonfn.html`*):
`cython -a cythonfn.pyx`



darker yellow: more calls into the Python virtual machine (expensive in loops)
white: more C code (non-Python)

when clicked, lines show the generated C code

file:///C:/cygwin/home/Administrator/python/pycon/code/high_performance_python/07_compiling/cython/lists/1/cytho

```
Generated by Cython 0.22

Raw output: cythonfn.c

+01: def calculate_z(maxiter, zs, cs):
 02:     """Calculate output list using Julia update rule"""
+03:     output = [0] * len(zs)
+04:     for i in range(len(zs)):
+05:         n = 0
+06:         z = zs[i]
+07:         c = cs[i]
+08:         while n < maxiter and abs(z) < 2:
+09:             z = z * z + c
+10:             n += 1
+11:         output[i] = n
+12:     return output
```

Most interesting *yellow* lines: 4 and 8 (from profiling we know line 8 is executed ~30 million times).
Almost as *yellow*: 9-11. Inside the loop, so interesting also.

From profiling we know that lines 6 and 7 are called only about a million times: less interesting.
(Also, as list objects they would benefit being replaced with NumPy arrays)

***Profiling:* see line_profiler** https://pypi.python.org/pypi/line_profiler/

To improve execution time we declare object types

- Inside loops
- dereferencing list and array items
- math

Adding some primitive types using the `cdef` syntax (at top of function):

- `int` for a signed integer
- `unsigned int` for positive integers
- `double complex` for double-precision complex numbers

```
$ cat cythonfn.pyx
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Cython annotations in the *.pyx* file are non-Python code

- Cannot use interactive Python interpreter
- Coding cycle as in C: code-compile-run-debug

Adding *C annotations* above results in

file:///C:/cygwin/home/Administrator/python/pycon/code/high_performance_python/07_compiling/cython/lists/2/cythc

Generated by Cython 0.22

Raw output: [cythonfn.c]()

```
+01: def calculate_z(int maxiter, zs, cs):
 02:     """Calculate output list using Julia update rule"""
 03:     cdef unsigned int i, n
 04:     cdef double complex z, c
+05:     output = [0] * len(zs)
+06:     for i in range(len(zs)):
+07:         n = 0
+08:         z = zs[i]
+09:         c = cs[i]
+10:         while n < maxiter and abs(z) < 2:
+11:             z = z * z + c
+12:             n += 1
+13:         output[i] = n
+14:     return output
```

Lines 11-12, inside the loop, are white!
The *~30 million times* line 10 is still yellow.

With these changes, Cython code runs in ~15 seconds.

If we had also changed

```
def calculate_z(int maxiter, zs, cs):
```
to
```
def calculate_z(int maxiter, list zs, list cs):
```

The time would drop to ~13 seconds.

Line 10 performs `abs()` on a complex number. Namely

$$\sqrt{c.real^2 + c.imag^2} < \sqrt{4}$$

Which we can modify to

$$c.real^2 + c.imag^2 < 4$$

So, the code becomes:

```
$ cat cythonfn.pyx
def calculate_z(int maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, n
    cdef double complex z, c
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

Since line 10 performs 30 million times, we can expect good dividends when eliminating calls to Python virtual machine for the `abs()`.

Indeed, now the code takes *0.26 seconds*.

Namely, on my machine

- **pure Python code runs in ~26 seconds**
- **Cython code runs in ~0.26 seconds**

Two orders of magnitude:
now, that's a gain!

`pyximport` may be used in place of `import`

in case your module doesn't need extra C libraries,
or a special build setup,
`pyximport` could import `.pyx` files *directly* (no need for *`setup.py`*):

```
>>> import pyximport; pyximport.install()
>>> import helloworld
Hello World
```

Cython has more tricks (one more we could use on this code is eliminate bound checking – when dereferencing list items – with `#cython: boundscheck=False`**).**

Cython and NumPy: lists and arrays

List objects have overhead for each dereference (list objects may reside anywhere in memory). Array objects store primitive type in contiguous addresses.

Python has (1D) `array` module (for primitive type like integers, floating-point numbers, characters, and Unicode strings).

NumPy's `numpy.array` module support multidimensional storage (and additional primitive types, like complex numbers).

When iterating over array objects –
compiler can calculate addresses directly
(using C offsets. no need to consult Python virtual machine).

Parallelizing with OpenMP on One Machine

OpenMP (Open Multi-Processing): a cross-platform API that supports parallel execution and memory sharing for C, C++, and Fortran.

**Speed may be gained,
for certain (embarrassingly) parallel problems,
by employing OpenMP C++ extensions: utilize multiple cores.**

In Cython, OpenMP is added with

- **`prange` (parallel range) operator**
- **adding `-fopenmp` compiler directive to *`setup.py`*.**

Releasing GIL in Cython

Work in a `prange` loop can be performed in parallel because we disable the global interpreter lock (GIL).

How?

In code section

`with nogil:` is a block where GIL is disabled;
inside this block we use `prange` to enable an OpenMP parallel `for` loop to independently calculate each `i`.

```
$ cat cython_np.pyx
from cython.parallel import prange
import numpy as np
cimport numpy as np

def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs):
    """Calculate output list using Julia update rule"""
    cdef unsigned int i, length
    cdef double complex z, c
    cdef int[:] output = np.empty(len(zs), dtype=np.int32)
    length = len(zs)
    with nogil:
        for i in prange(length, schedule="guided"):
            z = zs[i]
            c = cs[i]
            output[i] = 0
            while output[i] < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
                z = z * z + c
                output[i] += 1
    return output
```

To suspend Gil for the whole function

```
def calculate_z(int maxiter, double complex[:] zs, double complex[:] cs) nogil:
```

A variant on the loop parallelizing is to release the GIL explicitly:

```
for i in prange(length, schedule="guided", nogil=True):
```

Note:

When in `nogil` section, call regular Python objects (e.g., lists) at your peril!
You should call only primitive objects and *`memoryview`* objects (supporting the *`buffer`* interface/protocol).
Cython will *not* stop you accessing Python objects.

When compiling *cython_np.pyx*, we ask the C compiler to enable OpenMP by using -fopenmp as argument during compilation and linking, as in this *setup.py:*

```
$ cat setup.py
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

# for notes on compiler flags e.g. using
# export CFLAGS=-O2
# so gcc has -O2 passed (even though it doesn't make the code faster!)
# http://docs.python.org/install/index.html

setup(
    cmdclass={'build_ext': build_ext},
    ext_modules=[Extension("calculate", ["cython_np.pyx"], extra_compile_args=[
                           '-fopenmp'], extra_link_args=['-fopenmp'])]
)
```

prange

`cython.parallel`.**`prange`**(*[start,] stop[, step][, nogil=False][, schedule=None[, chunksize=None]][, num_threads=None]*)

> This function can be used for parallel loops. OpenMP automatically starts a thread pool and distributes the work according to the schedule used. `step` must not be 0. This function can only be used with the GIL released. If `nogil` is true, the loop will be wrapped in a nogil section.

With Cython's `prange`, we can choose different scheduling approaches.

Static: workload is distributed evenly across the available CPUs.
With this scheduling, some parallel calculations may end before others (then their threads will be idle).
dynamic and guided schedules: Cython allocates work in smaller chunks, so work is more evenly spread among CPUs.

Speed Statistics from the web -

Traveling Salesman Problem Times

https://www.stavros.io/posts/optimizing-python-with-cython/

cities	Python	Cython
17	20 sec	3.45 sec
48	69 sec	3.92 sec
100	217 sec	4.88 sec
2500	many hours	868 sec

44x speedup with the 100 cities dataset.

Note: looking at the code, there's room for improvement, so after algorithm optimizing - the speedup may be less.

Bioinformatics Problem Times

https://www.stavros.io/posts/speeding-up-python-code-with-shedskin/

Python	Shed Skin
4841.94 sec	103.30 sec

x47 speedup.

Shed Skin

Shed Skin

- experimental Python-to-C++ compiler (Python 2.4–2.7)
- uses type inference: *automatically* annotate each variable (static) type
- user provides example how to call function –
  Shed Skin figures the rest (with right kind of data)
- annotated code translated into C++ compiled with a standard compilers (e.g., g++)
- Shed Skin builds standalone executables – normal import into regular Python code
- More than 75 working examples (mainly math, pure Python)
- uses Boehm mark-sweep garbage collection

Pro

- no need to explicitly specify types by hand

Cons

- analyzer needs to infer types for *every* variable
- (currently) limited to automatically converting into C few thousands of lines of Python
- uses external implementations of standard libraries (~25, e.g. `re`, `random`)

- Similar benefits to PyPy
- automatically added type annotations – interesting
- Shed Skin's generated C may be easier to read than Cython’s generated C

http://github.com/shedskin/shedskin
http://shedskin.readthedocs.org/en/latest/documentation.html

Example: Building an Extension Shed Skin Module

https://github.com/mynameisfiber/high_performance_python

We will build an extension module,
which can be imported as in previous Cython example
(also possible to compile a module into standalone executable).

Following code is normal Python code: *not* annotated.

We add `__main__` with example arguments - so Shed Skin can infer types passed into `calculate_z`, and thus infer rest of types (function that calls another function will save adding second to `__main__` stanza).

```
$ cat shedskinfn.py
def calculate_z(maxiter, zs, cs):
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        while n < maxiter and abs(z) < 2:
            z = z * z + c
            n += 1
        output[i] = n
    return output


if __name__ == "__main__":
    # make a trivial example using the correct types to enable type inference
    # call the function so ShedSkin can analyze the types
    output = calculate_z(1, [0j], [0j])
```

No manual annotation, so - module may be imported before and after compilation (normal Python debugging possible).

Compiling the external module

```
$ shedskin -e shedskinfn.py
*** SHED SKIN Python-to-C++ Compiler 0.9.4 ***
Copyright 2005-2011 Mark Dufour; License GNU GPL version 3 (See LICENSE)

[analyzing types..]
*********************************100%
[generating c++ code..]
[elapsed time: 2.21 seconds]
```

Compiling `shedskinfn.py` creates files:

- `shedskinfn.cpp` - C++ source
- `shedskinfn.hpp` - C++ header
- `Makefile`

- `shedskinfn.ss.py` - annotation

Created C++

```
$ head -20 shedskinfn.cpp
#include "builtin.hpp"
#include "shedskinfn.hpp"

namespace __shedskinfn__ {

str *const_0;


list<__ss_int> *output;
str *__name__;



list<__ss_int> *calculate_z(__ss_int maxiter, list<complex> *zs, list<complex> *
cs) {
    /**
    Calculate output list using Julia update rule
    */
    __ss_bool __2, __3;
    __ss_int __0, __1, i, n;
    complex c, z;
```

Automatic annotation file created when we use `"-a --ann"` or `"-e --extmod"`:

```
$ cat shedskinfn.ss.py
def calculate_z(maxiter, zs, cs):        # maxiter: [int], zs: [list(complex)],
cs: [list(complex)]
    """Calculate output list using Julia update rule"""
    output = [0] * len(zs)               # [list(int)]
    for i in range(len(zs)):             # [__iter(int)]
        n = 0                            # [int]
        z = zs[i]                        # [complex]
        c = cs[i]                        # [complex]
        while n < maxiter and abs(z) < 2: # [int]
            z = z * z + c                # [complex]
            n += 1                       # [int]
        output[i] = n                    # [int]
    return output                        # [list(int)]


if __name__ == "__main__":               # []
    # make a trivial example using the correct types to enable type inference
    # call the function so ShedSkin can analyze the types
    output = calculate_z(1, [0j], [0j])  # [list(int)]
```

Running `make` will generate *`shedskinfn.so/shedskinfn.pyd/shedskinfn.dll`*

Importing the function

```
$ echo ; echo "..." ; head -3 julia1.py | tail -1 ; echo "..." ; head -10 julia1
.py  | tail -1 ; echo "..." ; head -42 julia1.py | tail -6

...
import shedskinfn
...
def calc_pure_python(desired_width, max_iterations):
...
    start_time = time.time()
    output = shedskinfn.calculate_z(max_iterations, zs, cs)
    end_time = time.time()
    secs = end_time - start_time
    print "Took", secs, "seconds"
```

on my machine

- pure Python code runs in ~26 seconds
- Shed Skin code runs in ~0.77 seconds

A very respectable gain in speed for almost no work!

If we do the `abs()` function expansion as we did with the Cython example

Shed Skin code runs in ~0.32 seconds (only a bit slower than Cython).

Execution time difference between Shed Skin and Cython is caused by 2,000,000 complex numbers (list objects) copied into `calculate_z` in the Shed Skin environment, and 1,000,000 integers copied out again.

However, in Shed Skin, no programmer's time is needed to annotate variables.

Note on running times:
We get similar times (Cython, PyPy, and Shed Skin) for this code.
This does not mean that times would be similar for *all* codes.
Time your code to get relative performances.

Numba

Numba

A just-in-time compiler (from Continuum Analytics) specializing in NumPy.

Pros

- doesn't require a pre-compilation pass: when you run it on new code it compiles each annotated function for your hardware
- you only have to add a decorator marking which functions to focus on
- runs on all standard NumPy code
- Supports OpenMP
- Can use GPUs

Cons

- compiles at runtime via the LLVM compiler (not via g++ or gcc as previous cases)
- LLVM has many dependencies: best to install Numba via Continuum's Anaconda distribution
- young project: API still changes between versions

Example: Numba compilation

`@jit` decorator is added to the core Julia function.
This is all we need to do!

```
@jit()
def calculate_z_serial_purepython(maxiter, zs, cs, output):
```

With that change, the gains in speed are impressive (~0.34 seconds vs. ~75 seconds). If we run same code a second time, it would run even faster (same as Cython), as the JIT compiler has nothing to do (pypy has same warm-up issue, as it's likewise a JIT compiler).

Numba also has the `@vectorize` decorator, and has several introspection capabilities, like

```
>>> print(numba.typeof(zs))
array(complex128, 1d, C))q
```

or `inspect_types`, where the inferred compiled code type information is reviewed (so you can change the code to help Numba accurately determine more type inference opportunities.

Pythran

Pythran

A Python-to-C++ compiler for a subset of Python that includes partial NumPy support.
Similar to Numba and Cython:

- **Requires function type annotation (as comments)**
- **Pythran adds**
  - **more type annotation**
  - **code specialization including**
    - **vectorization possibilities**
    - **OpenMP-based parallelization (e.g., when using `map()`)**

Python 2.7 only.

- **Supports large subset of Python (including Exceptions, generators, named parameters)**
- **Pythran finds parallelization opportunities (`map()`), and converts to parallel code – automatically**
- **(like Cython) you can mark parallel blocks manually with pragma `omp`**
- **Compiles normal Python and NumPy to fast C++ (even faster than Cython)**
- **Young project**

https://github.com/serge-sans-paille/pythran
https://serge-sans-paille.github.io/talks/pythran-2014-07-15.html

Example: compiling with Pythran

Pythran creates *Python-compatible code:* directives are Python comments.
Advantage in debugging: just delete the `.so/.dll/.pyd`.

```
$ cat calculate.py
import numpy as np

# pythran export calculate_z(int, complex[], complex[], int[])
def calculate_z(maxiter, zs, cs, output):
    """Calculate output list using Julia update rule"""
    # omp parallel for schedule(guided)
    for i in range(len(zs)):
        n = 0
        z = zs[i]
        c = cs[i]
        # while n < maxiter and abs(z) < 2:
        while n < maxiter and (z.real * z.real + z.imag * z.imag) < 4:
            z = z * z + c
            n += 1
        output[i] = n
    return output
```

When we add the `omp` pragma, run time is similar to Cython.

PyPy

PyPy

- A replacement JIT of CPython compiler: supports all batteries-included modules
- Python 2.7, experimental Python 3.2+
- Almost Cython speedup without any work
- No built-in support for NumPy (see current numpypy status at `http://buildbot.pypy.org/numpy-status/latest.html`).

`julia1.py` **code runs without any modifications**

- pure Python code runs in ~26 seconds
- Cython code runs in ~0.26 seconds
- PyPy code runs in ~0.33 seconds

As with Numba, if same code runs again *in the same session*, subsequent runs don't need to compile and are faster.

Notes:

- **PyPy supports all built-in modules**
- **If your problem may be parallelized (with just batteries included modules), you can use** `multiprocessing` **module**
- **Importing pure Python modules likely to work (list:** https://bitbucket.org/pypy/compatibility/wiki/Home)
- **Importing Python C extension modules probably won't work**
- **Garbage collection:**
  - **CPython – reference counting**
  - **PyPy – mark-and-sweep**
  - **Caveats:**
    - **e.g., flushing of files: use context manager (**`with` **directive)**
- **GIL used. *STM* project tries to remove GIL dependency** (http://doc.pypy.org/en/latest/stm.html).
- **Profiling:**
  - **Jitviewer** (https://bitbucket.org/pypy/jitviewer)
  - **Logparser** (https://github.com/MichaelBlume/pypy/blob/master/pypy/tool/logparser.py)

Comparisons

	Cython	Shed Skin	Numba	Pythran	PyPy
Mature project	Yes	Yes			Yes
Used widely	Yes				
NumPy support	Yes		Yes	Yes	Numpypy
Not breaking Python code		Yes	Yes	Yes	Yes
C knowledge needed	Yes				
OpenMP support	Yes		Yes	Yes	Yes

- *Cython* **probably offers the best results for the widest set of problems. Requires more effort in using and debugging due to its use of mix of Python with C annotations**
- *PyPy* **is a strong option if not using NumPy or other hard-to-port C extensions**
- *Numba* **may offer quick wins for little effort. likewise has limitations which might stop it working well on your code. Also - relatively young project**
- *Shed Skin* **may be useful if you want to compile to C and you're not using NumPy or other external libraries**

Note: for light numeric requirements, Cython's buffer interface accepts `array.array` matrices—this is an easy way to pass a block of data to Cython for fast numeric processing without having to add NumPy as project dependency.

Addenda

time was limited

no religious wars

not all alternatives were discussed

No discussion of -

- **Weave (`scipy.weave`), `NumExpr`,**
  - **as Numba gives us**
    - **same benefits**
    - **interface may be easier**

- **Parakeet**
  - **similar to Numba (JIT)**
  - **only supported data types: NumPy arrays, scalars, tuples, and slices**
  - **Numba supports also pure Python**

- **Theano**
  - **higher-level language for expressing mathematical operators on multidimensional arrays**
  - **tight integration with NumPy**
  - **compiles code to C using CPUs and GPUs**

- **Psyco**
  - **extension module to speedup any Python code**
  - **JIT techniques**
  - **unmaintained and dead**

- **Nikita**
  - **good replacement for CPython**
  - **compiles every construct that CPython 2.6, 2.7, 3.2, 3.3 and 3.4 offers**
  - **translates Python code into C++ code**

Other Upcoming Projects: http://compilers.pydata.org

Profiling to Find Bottlenecks

Why profiling? To identify –

- **speed bottlenecks**
- **too much RAM usage**
- **too much disk I/O or network I/O**

with -

- **print `time.time()` in strategic places. (or use Unix `time`)**
- **cProfile**
- **line_profiler**
- **heapy tracks all objects inside Python's memory (good for memory leaks)**
- **For long-running systems, use dowser: allows live objects introspection (web browser interface)**
- **memory_profiler for RAM usage**
- **examine Python bytecode with dis**