NEVER GET IN A BATTLE OF BITS WITHOUT AMMUNITION

Enrico Franchi
(enrico.franchi@gmail.com)
Python is not Slow...
It is “Differently fast”.
Some background...

- Memory usage is probably my main problem.
- Usually I find CPU-bound performance adequate (especially with NumPy derived stuff).
- I prefer algorithmic/architectural optimizations.
- Many issues are solved with PyPy.
- cPython is still "the standard".
- Python "Basic" types
  - Memory occupation
  - Implementation
- OO Design in scientific setting
- CPU Profiling
- Memory Profiling
High Level Languages

• Low-level vs. high-level

• High-level languages create abstractions, which is usually fine...

• Unless when it is not (!!)

• Then you have to understand quite a lot more about how your platform works

• What about abstraction leaks?

• Law of Leaky Abstractions
  (http://www.joelonsoftware.com/articles/LeakyAbstractions.html)

• Zen and the Art of Abstraction Maintenance (A. Martelli, OSCON'09)
Flat is Better than Nested

Object oriented programming leads towards "nested" structures.

What is the cost of all this?
Object Oriented Programming

- Mostly Stateful Programming
- Deals with Mutability with Encapsulation
- Which also helps with hiding the implementation details
  - Program to an Interface, not to an Implementation
  - Avoid returning "handles" to object internals
- Making Interfaces that provide
  - Computationally Efficient Operations
  - All the required operations
Example

- Points take a lot of memory
- I expect a point to behave more like a number (immutable)
- Identity!

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```
• Points take a lot of memory
• I expect a point to behave more like a number (immutable)
• Identity!

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

import collections
Point = collections.namedtuple('Point', 'x y')
```

```
In [24]: import collections
Point = collections.namedtuple('Point', 'x y')
Point(3, 4)
```

```
Out[24]: Point(x=3, y=4)
```
Example

Points take a lot of memory

I expect a point to behave more like a number (immutable)

Identity!

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

import collections
Point = collections.namedtuple('Point', 'x y')

In [24]: import collections
   ...: Point = collections.namedtuple('Point', 'x y')
   ...: Point(3, 4)

Out[24]: Point(x=3, y=4)

In [25]: sys.getsizeof(Point(3, 4))

Out[25]: 72
```

vs. 344 bytes
Example

- In triangle we are returning a handle.
- Few advantages over a list
- More memory
- Do something smarter!
- Make a copy
- Return a "view"
- Change the API!

```python
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y

class Triangle(object):
    def __init__(self, a1, a2, a3):
        self._vertices = [a1, a2, a3]

@property
def vertices(self):
    return self._vertices
```
• For the purposes of this discussion, we leave descriptor completely out

• Most (?) Python objects have a __dict__ attribute

• It is the first place where attributes are looked up

• It is the place where attributes are written

• A “normal” Python object at least occupies the space required for the __dict__ (and then some)

• Python objects defined in C normally have no __dict__ attribute

• Python objects whose class was defined with a __slots__ attribute do not have a __dict__ attribute (unless specifically requested)

• They still occupy some space (just for being there)
getsizeof(object, default) -> int

Return the size of object in bytes.

In [3]: import sys

In [10]: class Normal(object):
   ...:     pass
   ...:     n = Normal()
   ...:     sys.getsizeof(n)

Out[10]: 64

In [11]: class Slotted(object):
   ...:     __slots__ = ()
   ...:     s = Slotted()
   ...:     sys.getsizeof(s)

Out[11]: 16

In [12]: s.foo = 1  # AttributeError
If we require `__dict__`, we get it! (And we pay for it)

Instances of Subclasses of slotted classes, still have a `__dict__`!

Unless their class also has a `__slots__`!
More on getsizeof (1)

- Return the size of an object in bytes.
- Built-in objects give correct results. Third party stuff depends.
- Only the object, not what it refers to!

```python
In [7]:
class Fatty(object):
    def __init__(self, sz):
        self.a = range(sz)

f = Fatty(10000)
print sys.getsizeof(f)
print sys.getsizeof(f.a)
print sum(map(sys.getsizeof, f.a))
```

64
80072
240000
**More on getsizeof (2)**

- **getsizeof** returns the value returned by **__sizeof__** + space used for reference counting

```python
In [8]:
class Munchausen(object):
    __slots__ = ()
    def __sizeof__(self):
        return 1000000

baron = Munchausen()
sys.getsizeof(baron)

Out[8]: 1000000
```
**GETSIZEOF (SLOTS VS. ATTRIBUTES)**

In [9]:
```python
class LotsOfSlots(object):
    __slots__ = ['a%d' % i for i in xrange(1000)]
los = LotsOfSlots()
sys.getsizeof(los)
```

Out[9]:
```
8048
```

In [14]:
```python
class LotsOfAttributes(object):
    def __init__(self):
        for i in xrange(1000):
            setattr(self, 'a%d' % i, None)

loa = LotsOfAttributes()
print sys.getsizeof(loa)
print sys.getsizeof(loa.__dict__)
```

```
64
49432
```

OK... AN OBJECT WITH 1000 ATTRIBUTES IS AN EXERCISE IN BAD DESIGN...
From the previous slides: 10000 ints = 240000 bytes

Yes: 1 Integer = 24 bytes

On a 64 bit Intel machine, a C "long" takes 64 BITS!!!

An integer in Python is a full-fledged object!

Here we do not consider the long datatypes
More Integers

- Although larger than C integers, Python integers are still relatively small entities (sizeof(size_t) + pointer + long)
- The cost of individually malloc'ing each integer used just once would be rather prohibitive (memory + cpu)
Puny Object!
Integer Storage

PyIntBlock

Unallocated

NULL

324

3672

next

objects

Allocated

free_list

PyInt_Type

ob_refcnt

ob_type

ob_ival

Small Integers

1

... ...

-5 ...

5 256

small_ints
• When an operation would result in a value not representable as a (Python) integer, a (Python) Long is returned instead

• A + B could “overflow”

• “Easy”: e.g., A + B < MAX_UNSIGNED_LONG (with A>0 & B>0)

```c
static PyObject *
int_add(PyIntObject *v, PyIntObject *w)
{
    register long a, b, x;
    CONVERT_TO_LONG(v, a);
    CONVERT_TO_LONG(w, b);
    /* casts in the line below avoid undefined behaviour on overflow */
    x = (long)((unsigned long)a + b);
    if ((x^a) >= 0 || (x^b) >= 0)
        return PyInt_FromLong(x);
    return PyLong_Type.tp_as_number->nb_add((PyObject *)v, (PyObject *)w);
}
```
... WHAT ENDS LONG!

- But some serious 🎶 can happen with multiplication (for example)
- From the docs: "Integer overflow checking for * is painful"
- Floating point arithmetic is used instead
  (No really, it makes sense; overflow checking is really a pain)
• There is a “floatBlock” linked list
• No “small floats” array (obviously)
• Operations are done according to the C platform semantics
• Each operation "converts" Python floats to C doubles, performs the operation and "converts" the result to a Python float

• \[x\times y \text{ for } x,y \text{ in } \text{izip}([...], [...])\]
• A Python list is implemented as an array of pointers to PyObject.

• Complexity is what we expect from a dynamic array.

  • Get/Set items: $O(1)$
  • Append: amortized $O(1)$
  • Insert: $O(N)$

• Memory usage is something like

  • "Lots of indirection"
Dictionaries

- **Dicts are essentially hash maps**
- **Definitely not a textbook implementation:**
  beautiful highly optimized implementation
- **No linked-lists (!!!)**
- **Open addressing**
- **It is probably the single most important structure of Python**
- **Used also as part of the implementation of other objects...**
- **Complexity is standard (performance is outstanding)**
- **Get/set amortized O(1)**
Dictionary Implementation

- DictEntry "holds" a key-value pair in the dictionary
- me_hash contains the hash of the key
- Very simple hashing functions for strings and integers (hash(int) = int)
- Relatively simple "probing" function

PyDictEntry

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Py_ssize_t me_hash</td>
<td>sizeof(size_t) + 2 * sizeof(pointer)</td>
</tr>
<tr>
<td>PyObject* me_key</td>
<td>64 * 3 = 192 bits = 24 bytes</td>
</tr>
<tr>
<td>PyObject* me_value</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>me_key = me_value = NULL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unused</td>
<td></td>
</tr>
<tr>
<td>Active</td>
<td>me_value != NULL &amp;&amp; me_key∉{NULL, dummy}</td>
</tr>
<tr>
<td>Dummy</td>
<td>me_key = dummy &amp;&amp; me_value = NULL</td>
</tr>
</tbody>
</table>
\[
\begin{align*}
\text{PyInt 5} & \quad \{5\} \\
\text{PyInt 3} & \quad \{3\} \\
\text{PyInt 7} & \quad \{7\}
\end{align*}
\]

\[
j = (5 \times j) + 1 + \text{perturb};
\]
\[
\text{perturb} >>= \text{PERTURB\_SHIFT};
\]
\[
\text{use } j \% 2^{\ast}i \text{ as the next table index};
\]
typedef struct __dictobject PyDictObject;
struct __dictobject {
    PyObject_HEAD
    Py_ssize_t ma_fill;    /* # Active + # Dummy */
    Py_ssize_t ma_used;    /* # Active */

    /* The table contains ma_mask + 1 slots, and that's a power of 2. */
    Py_ssize_t ma_mask;

    /* ma_table points to ma_smalltable for small tables, else to * additional malloc'ed memory. ma_table is never NULL! */
    PyDictEntry *ma_table;
    PyDictEntry *(*ma_lookup)
        (PyDictObject *mp, PyObject *key, long hash);
    PyDictEntry ma_smalltable[PyDict_MINSIZE];
};
Size of Dictionaries

- An empty dict takes 280 bytes on a 64 bit intel machine
- A lot of python objects do have dicts!

```python
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}
```
Size of Dictionaries

- An empty dict takes 280 bytes on a 64 bit Intel machine

- A lot of Python objects do have dicts!

graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}

graph = {'A': {'B': [], 'C': []},
         'B': {'C': [], 'D': []},
         'C': {'D': []},
         'D': {'C': []},
         'E': {'F': []},
         'F': {'C': []}}

Internal NetworkX representation
6 elements = 2.5 KB
Size of Dictionaries

- An empty dict takes 280 bytes on a 64 bit Intel machine
- A lot of Python objects do have dicts!

```python
graph = {'A': ['B', 'C'],
         'B': ['C', 'D'],
         'C': ['D'],
         'D': ['C'],
         'E': ['F'],
         'F': ['C']}
```

Internal NetworkX representation

6 elements = 2.5 KB

```python
graph = {'A': {'B': [], 'C': []},
         'B': {'C': [], 'D': []},
         'C': {'D': []},
         'D': {'C': []},
         'E': {'F': []},
         'F': {'C': []}}
```

10^6 nodes with NX is 5 GB

As a sparse matrix it is < 2 MB
Tuples

- "Immutable sequence"
- Not general purpose data structures (Wesley Chun: http://wescpy.blogspot.it/2012/05/tuples-arent-what-you-think-theyre-for.html)
- "Composite" dictionary keys
- "Individual entity" (maybe namedtuples are even better)
- Get data to and from functions
- Some memory overhead is there

```python
In [20]: import sys
    sys.getsizeof(('a', 'b'))
Out[20]: 72

In [21]: sys.getsizeof(('a', 'b', 'c'))
Out[21]: 80
```
How large is your object?

- `getsizeof` (already discussed)
- `pympler.asizeof`
- `pip install pympler`

```
[20]: p = Point(3, 4)  # namedtuple
    o = OldPoint(3, 4) # plain object

[21]: print sys.getsizeof(p), sys.getsizeof(o) + sys.getsizeof(o.__dict__)
    72 344

[22]: print asizeof.asizeof(p), asizeof.asizeof(o)
    536 472
```

`namedtuples “lazily” create __dict__`
Putting all together

- Python “core” structures are very useful and powerful
- Built for ease of use + *some* performance constraints (super fast dicts)
- Each “nested” structure forces some indirection
  - More memory overhead
  - “Pointer chains”
    - Less memory locality
    - Just more operations
Numpy Array

An n-dimensional array has property such as its shape or the data-type of the elements contains

\[(i_0, \ldots, i_{n-1}) \rightarrow I\]

Shape: \((d_0, \ldots, d_{n-1})\)

An n-dimensional array references some (usually contiguous memory area)

N-dimensional arrays are homogeneous

Is an object, so there is some behavior, e.g., the def. of \_\_add\_\_ and similar stuff

An n-dimensional array has shape, stride, flags
Numpy, copies & views

b = a.view()
b.flags.writeable = False
b[0,0] = 0 # RuntimeError
Profiling vs. Benchmarking

Profiling

Benchmarking
Profiling vs. Benchmarking

Profiling

Dynamic Program Analysis to measure space, frequency or duration of function calls or instructions

Benchmarking

Running a set of programs to assess their relative performance
Profiling vs. Benchmarking

Profiling

Dynamic Program Analysis to measure space, frequency or duration of function calls or instructions

Where is the problem?

Is there a problem?

Running a set of programs to assess their relative performance

Benchmarking
Benchmarking execution

- "Manual solutions"
- `timeit` module
- `ipython`

```python
In [9]: @timeit map(lambda x: x + 1, range(10000))
100 loops, best of 3: 4.07 ms per loop

In [10]: def f(x):
       return x + 1

In [11]: @timeit lst = range(10000)
   map(f, lst)
100 loops, best of 3: 3.16 ms per loop
```
Profiling Execution

- Profiling often requires "instrumenting" interpreter/code, etc.
- Profiling does not reliably measure "performance"

```python
import cProfile
import re

cProfile.run('re.compile("foo|bar")')  # to stdout
cProfile.run('re.compile("foo|bar")', 'restats')

python -m cProfile [-o output_file] [-s sort_order] myscript.py

In [9]: %prun re.compile("foo|bar")
```

- With pstat you read back the profiling log and examine it

```python
In [1]: import pstats

In [2]: p = pstats.Stats('standard.prof')
```
514425388 function calls (514424680 primitive calls) in 691.147 seconds

Ordered by: cumulative time
List reduced from 1251 to 9 due to restriction <9>

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.397</td>
<td>2.397</td>
<td>691.149</td>
<td>691.149</td>
<td>blogsim/execution.py:77(main)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>688.746</td>
<td>688.746</td>
<td>blogsim/execution.py:38(simulate)</td>
</tr>
<tr>
<td>1</td>
<td>0.000</td>
<td>0.000</td>
<td>686.806</td>
<td>686.806</td>
<td>blogsim/simulation.py:42(run)</td>
</tr>
<tr>
<td>1</td>
<td>0.630</td>
<td>0.630</td>
<td>666.737</td>
<td>666.737</td>
<td>blogsim/engine.py:98(run)</td>
</tr>
<tr>
<td>374636</td>
<td>53.619</td>
<td>0.000</td>
<td>659.533</td>
<td>0.002</td>
<td>blogsim/engine.py:58(read_post)</td>
</tr>
<tr>
<td>41416852</td>
<td>63.272</td>
<td>0.000</td>
<td>587.674</td>
<td>0.000</td>
<td>blogsim/engine.py:42(get_messages)</td>
</tr>
<tr>
<td>36416971</td>
<td>45.691</td>
<td>0.000</td>
<td>458.925</td>
<td>0.000</td>
<td>blogsim/purepy_backend.py:58(mark_could_have_read_if_active)</td>
</tr>
<tr>
<td>41416852</td>
<td>201.541</td>
<td>0.000</td>
<td>426.443</td>
<td>0.000</td>
<td>blogsim/purepy_backend.py:34(_mark_read_aux)</td>
</tr>
<tr>
<td>124251573</td>
<td>169.862</td>
<td>0.000</td>
<td>169.862</td>
<td>0.000</td>
<td>{method 'extend' of 'list' object}</td>
</tr>
</tbody>
</table>
```
p.sort_stats('time').print_stats(10)

Wed Jul  3 11:45:23 2013    standard.prof

514425388 function calls (514424680 primitive calls) in 691.147 seconds

Ordered by: internal time
List reduced from 1251 to 10 due to restriction <10>

<table>
<thead>
<tr>
<th>ncalls</th>
<th>tottime</th>
<th>percall</th>
<th>cumtime</th>
<th>percall</th>
<th>filename:lineno(function)</th>
</tr>
</thead>
<tbody>
<tr>
<td>41416852</td>
<td>201.541</td>
<td>0.000</td>
<td>426.443</td>
<td>0.000</td>
<td>blogsim/purepy_backend.py:34(_mark_read_aux)</td>
</tr>
<tr>
<td>124251573</td>
<td>169.862</td>
<td>0.000</td>
<td>169.862</td>
<td>0.000</td>
<td>{method 'extend' of 'list' objects}</td>
</tr>
<tr>
<td>41416852</td>
<td>63.272</td>
<td>0.000</td>
<td>587.674</td>
<td>0.000</td>
<td>blogsim/engine.py:42(get_messages)</td>
</tr>
<tr>
<td>374636</td>
<td>53.619</td>
<td>0.000</td>
<td>659.533</td>
<td>0.002</td>
<td>blogsim/engine.py:58(read_post)</td>
</tr>
<tr>
<td>36416971</td>
<td>45.691</td>
<td>0.000</td>
<td>458.925</td>
<td>0.000</td>
<td>blogsim/purepy_backend.py:58(mark_could_have_read)</td>
</tr>
<tr>
<td>41416852</td>
<td>38.875</td>
<td>0.000</td>
<td>38.875</td>
<td>0.000</td>
<td>{_bisect.bisect_right}</td>
</tr>
<tr>
<td>36416971</td>
<td>30.477</td>
<td>0.000</td>
<td>39.130</td>
<td>0.000</td>
<td>blogsim/purepy_backend.py:51(last_activation_of)</td>
</tr>
<tr>
<td>78270800</td>
<td>18.458</td>
<td>0.000</td>
<td>18.458</td>
<td>0.000</td>
<td>{method 'setdefault' of 'dict' objects}</td>
</tr>
<tr>
<td>46859708</td>
<td>10.162</td>
<td>0.000</td>
<td>10.162</td>
<td>0.000</td>
<td>blogsim/timeline.py:4(now)</td>
</tr>
<tr>
<td>374636</td>
<td>8.892</td>
<td>0.000</td>
<td>11.426</td>
<td>0.000</td>
<td>blogsim/engine.py:94(neighbors)</td>
</tr>
</tbody>
</table>
```
• If you really want to optimize a procedure, you need to know where time is spent inside the procedure!

• line_profiler (search the cheeseshop) does that!

```python
@profile
def slow_function(a, b, c):
    ...
```

• Run the program as usual

• Read the results with

```bash
$ python -m line_profiler script_to_profile.py.lprof
```
In [8]:
   : lst = range(100)
   : random.shuffle(lst)

In [9]:
   : %load_ext line_profiler

In [11]:
   : %lprun -f sort_numbers sort_numbers(lst)

In [12]:
   : %lprun -f insertion_sort.sort_numbers insertion_sort.sort_numbers(lst)
Timer unit: 1e-06 s
File: insertion_sort.py
Function: sort_numbers at line 1
Total time: 0.011399 s

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
</table>
| 1      | 100  | 535   | 5.3     | 4.7     | def sort_numbers(s):
| 2      | 99   | 124   | 1.3     | 1.1     | for i in range(1, len(s)):
| 3      | 99   | 120   | 1.2     | 1.1     | val = s[i]
| 4      | 99   | 120   | 1.2     | 1.1     | j = i - 1
| 5      | 2608 | 4110  | 1.6     | 36.1    | while (j >= 0) and (s[j] >
| 6      | 2509 | 3378  | 1.3     | 29.6    | s[j+1] = s[j]
| 7      | 2509 | 2971  | 1.2     | 26.1    | j = j - 1
| 8      | 99   | 161   | 1.6     | 1.4     | s[j+1] = val |
In [8]:
    
    lst = range(100)
    random.shuffle(lst)

In [9]:
    %load_ext line_profiler

In [11]:
    %lprun -f sort_numbers sort_numbers(lst)

In [25]:
    %lprun -f insertion_sort.sort_numbers insertion_sort.sort_numbers(lst)
Timer unit: 1e-06 s

File: insertion_sort.py
Function: sort_numbers at line 1
Total time: 143.9 s

<table>
<thead>
<tr>
<th>Line #</th>
<th>Hits</th>
<th>Time</th>
<th>Per Hit</th>
<th>% Time</th>
<th>Line Contents</th>
</tr>
</thead>
</table>
| 1      | 10000  | 22512 | 2.3     | 0.0     | def sort_numbers(s):
| 2      |        |       |         |         |               |
| 3      | 9999   | 24376 | 2.4     | 0.0     | for i in range(1, len(s)):
| 4      | 9999   | 19258 | 1.9     | 0.0     | val = s[i]
| 5      | 25289518 | 48885095 | 1.9   | 34.0     | j = i - 1
| 6      | 25279519 | 50397861 | 2.0   | 35.0     | while (j >= 0) and (s[j] > val):
| 7      | 25279519 | 44531577 | 1.8   | 30.9     | s[j+1] = s[j]
| 8      | 9999   | 19185 | 1.9     | 0.0     | j = j - 1
|        |        |       |         |         | s[j+1] = val |
Memory Profiling

- No "standard" memory profiler

- Several options with slightly different focus
  - ObjGraph: creates a graph of references/backreferences
  - Meliae: "two step process" like cProfile & pstat
  - PyMpler (especially memory leaks)
instance
<DocTest index.txt from docs/index.txt:0>

```python
>>> x = []
>>> y = [x, [x], dict(x=x)]
>>> import objgraph
>>> objgraph.show_refs([y], filename='sample-graph.png')
```

```python
objgraph.show_backrefs([x], filename='sample-backref-graph.png')
```
We would need xdot to navigate the graph interactively!
```python
>>> om = loader.load('my.dump')

>>> om.summarize()
Total 5078730 objects, 290 types,
Total size = 367.4MiB (385233882 bytes)

<table>
<thead>
<tr>
<th>Index</th>
<th>Count</th>
<th>%</th>
<th>Size</th>
<th>% Cum</th>
<th>Max</th>
<th>Kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2375950</td>
<td>46</td>
<td>224148214</td>
<td>58</td>
<td>4194313</td>
<td>str</td>
</tr>
<tr>
<td>1</td>
<td>63209</td>
<td>1</td>
<td>77855404</td>
<td>20</td>
<td>3145868</td>
<td>dict</td>
</tr>
<tr>
<td>2</td>
<td>1647097</td>
<td>32</td>
<td>29645488</td>
<td>7</td>
<td>20</td>
<td>dict</td>
</tr>
<tr>
<td>3</td>
<td>374259</td>
<td>7</td>
<td>14852532</td>
<td>3</td>
<td>304</td>
<td>tuple</td>
</tr>
<tr>
<td>4</td>
<td>138464</td>
<td>2</td>
<td>12387988</td>
<td>3</td>
<td>536</td>
<td>unicode</td>
</tr>
</tbody>
</table>
```

...
How to get faster?

- Pypy
- For floating point computation Numpy! (even with pypy)
- Pandas: "R" in Python
- weave (C++ embedded!)

```python
def prod7(m, v):
    nrows, ncolumns = m.shape
    res = np.zeros(nrows)
    code = r""
    for (int i=0; i<nrows; i++) {
        for (int j=0; j<ncolumns; j++) {
            res(i) += m(i, j)*v(j); }
    }
    ""
    err = weave.inline(code,
                       ['nrows', 'ncolumns', 'res', 'm', 'v'],
                       type_converters=weave.converters.blitz, compiler='gcc')
```
How to get faster?

- Pypy
- For floating point computation Numpy! (even with pypy)
- Pandas: "R" in Python
- Weave (C++ embedded!)

Numpy is still faster!

```python
def prod7(m, v):
    nrows, ncolumns = m.shape
    res = np.zeros(nrows)
    code = r""
    for (int i=0; i<nrows; i++) {
        for (int j=0; j<ncolumns; j++) {
            res(i) += m(i, j)*v(j);
        }
    }
    ""

    err = weave.inline(code,
                       ['nrows', 'ncolumns', 'res', 'm', 'v'],
                       type_converters=weave.converters.blitz, compiler='gcc')
```

```
In [5]: %timeit np.dot(m, v)
100 loops, best of 3: 2.19 ms per loop

In [6]: %timeit prod7(m, v)
10 loops, best of 3: 19 ms per loop
```
### Performance Hints

Running matrix-vector products with a 2000 × 2000 dense matrix and `numpy` arrays gave the following relative timings:

<table>
<thead>
<tr>
<th>method</th>
<th>function name</th>
<th>CPU time</th>
</tr>
</thead>
<tbody>
<tr>
<td>pure Python loops</td>
<td>prod1</td>
<td>490</td>
</tr>
<tr>
<td>map/reduce</td>
<td>prod2</td>
<td>454</td>
</tr>
<tr>
<td>map/reduce</td>
<td>prod3</td>
<td>209</td>
</tr>
<tr>
<td>Psyco</td>
<td>prod6</td>
<td>327</td>
</tr>
<tr>
<td>Fortran</td>
<td>prod4</td>
<td>2.9</td>
</tr>
<tr>
<td>Fortran, cache-friendly loops</td>
<td>prod5</td>
<td>1.0</td>
</tr>
<tr>
<td>Weave</td>
<td>prod7</td>
<td>1.6</td>
</tr>
<tr>
<td>NumPy</td>
<td>dot</td>
<td>1.0</td>
</tr>
</tbody>
</table>

All these results were obtained with double precision array elements.
Cython

- Creates a compiled python module
- Cython -> C -> native code
- Interoperates with numpy
- To some extent, with C++
- Allows to declare buffers
- Type annotations
- "Objects" with methods that are not Python accessible

```python
def naive_convolve(np.ndarray[np.DTYPE_t, ndim=2] f not None,
                    np.ndarray[np.DTYPE_t, ndim=2] g not None):
```
from monary import Monary
import numpy

with Monary("127.0.0.1") as monary:
    arrays = monary.query(
        "mydb",                       # database name
        "collection",                # collection name
        {},                          # query spec
        ["x1", "x2", "x3", "x4", "x5"], # field names (in Mongo record)
        ["float64"] * 5              # Monary field types (see below)
    )

    for array in arrays:
        print numpy.mean(array)       # prove that we did something...

• PyMongo Insert -- EC2: 102 s -- Mac: 76 s
• PyMongo Query -- EC2: 85 s -- Mac: 88 s
• Monary Query -- EC2: 5.4 s -- Mac: 3.8 s
Thanks For Your Kind Attention

Never get in a Battle of Bits without Ammunition

Enrico Franchi (enrico.franchi@gmail.com)