# 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"

## Outline

- 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

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

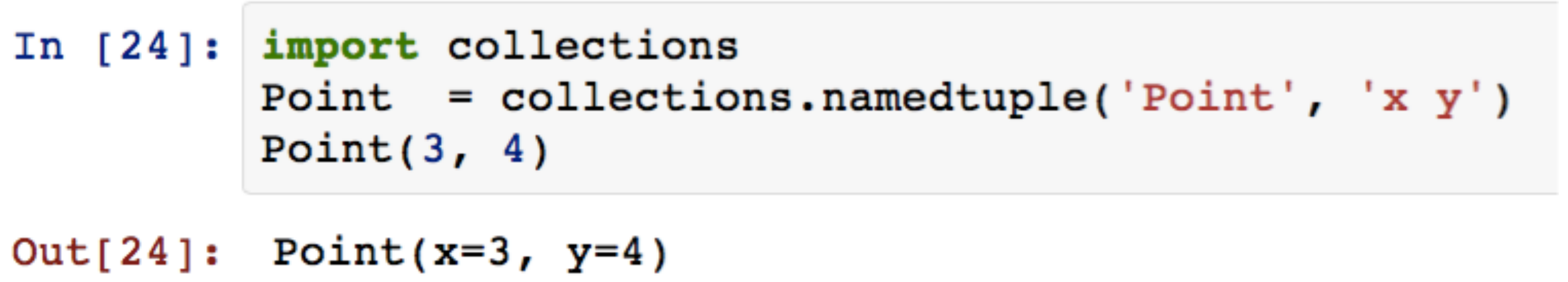
- Points take a lot of memory
- I expect a point to behave more like a number (immutable)
    - Identity!

## Example

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



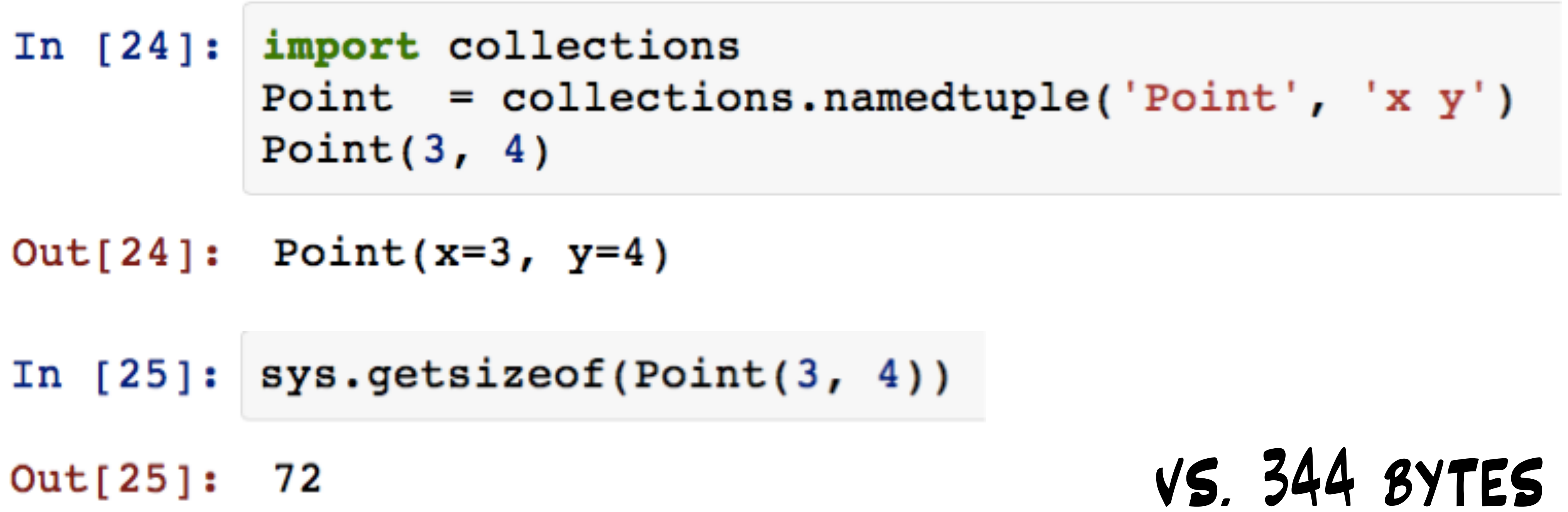
```
In [24]: import collections
         Point  = collections.namedtuple('Point', 'x y')
         Point(3, 4)

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

## Example

```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
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

```
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
```

- 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 Object Model

- 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)

## Starting to get the idea...

```
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
```

## More on Slots…

```
In [23]: class SlottedDict(object):
             __slot__ = ('__dict__', )
         sd = SlottedDict()
         sys.getsizeof(sd)

Out[23]: 64
```

If we require `__dict__`, we get it! (and we pay for it)

```
In [18]: class SubSlot(Slotted):
             pass
         ss = SubSlot()
         ss.a = 1
         sys.getsizeof(ss)

Out[18]: 64
```

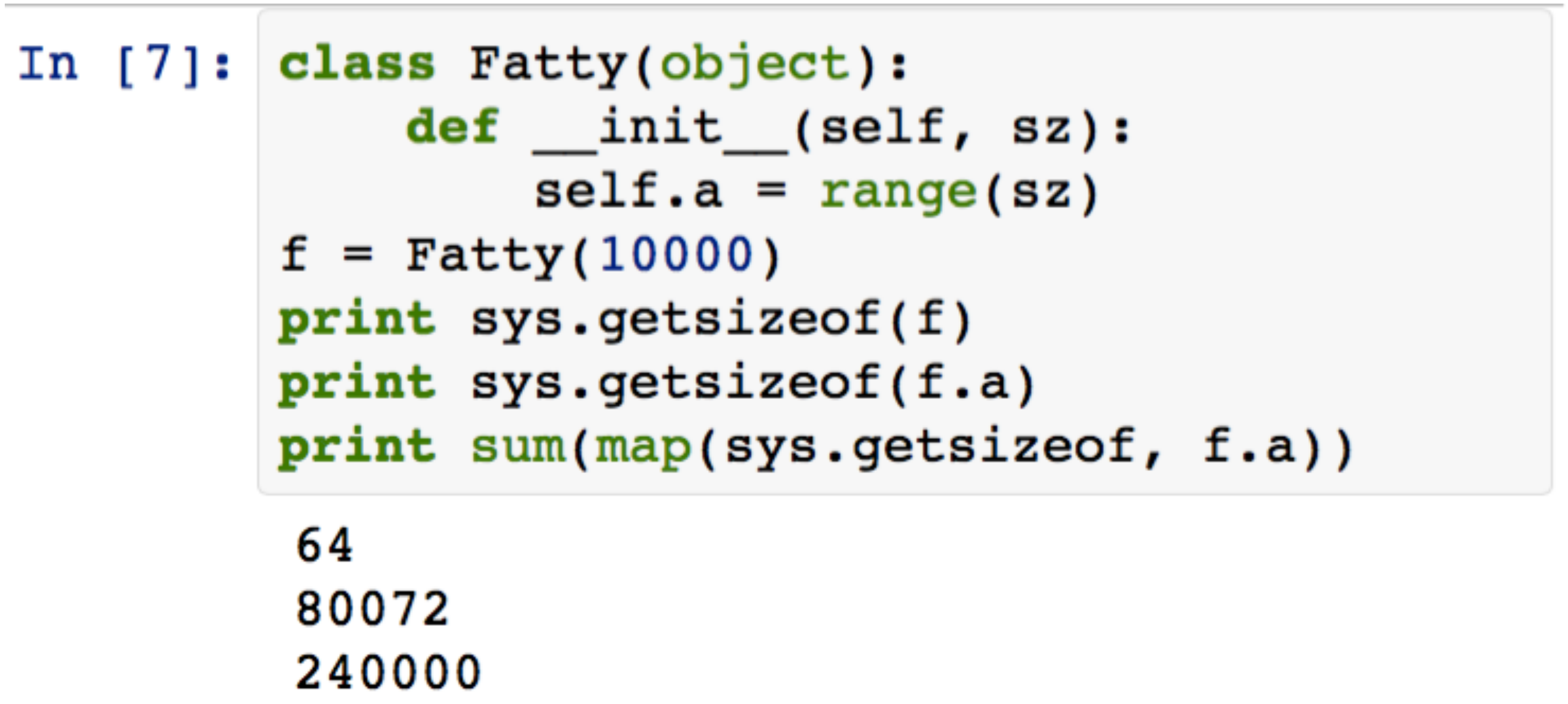
Instances of Subclasses of slotted classes, still have a `__dict__`!

```
In [21]: class SubSlot2(Slotted):
             __slots__ = ()
         ss2 = SubSlot2()
         sys.getsizeof(ss2)

Out[21]: 16
```

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!

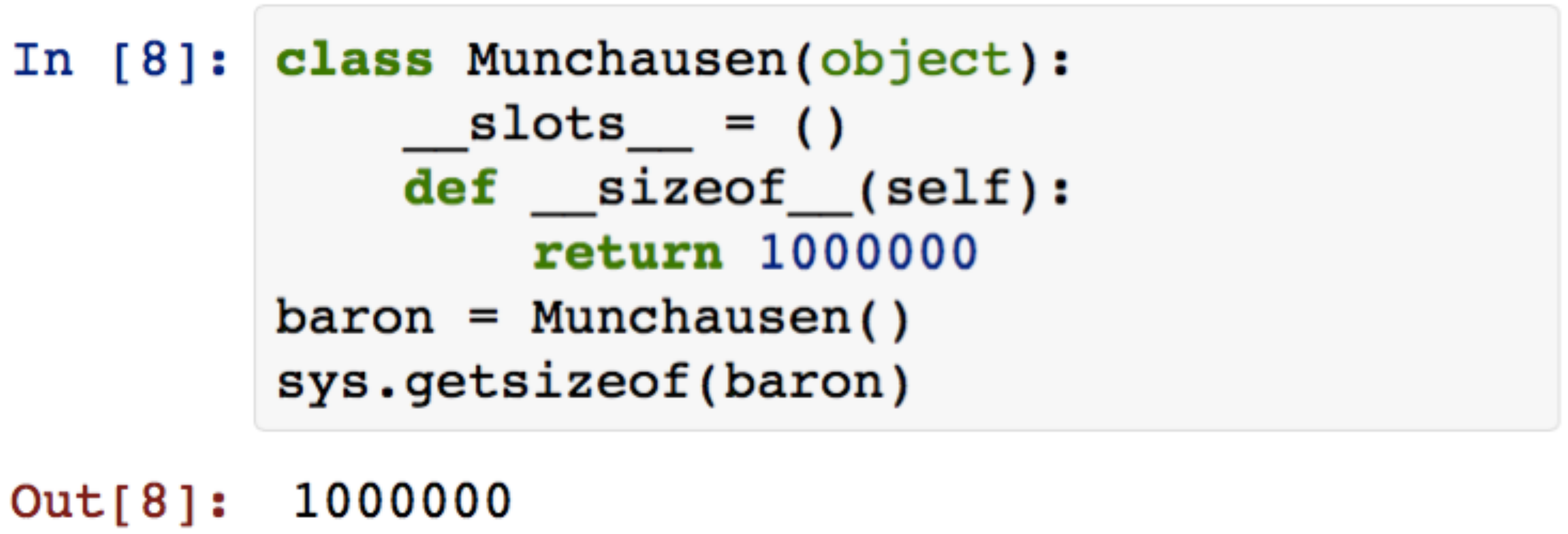


```
In [7]: class Fatty(object):
            def __init__(self, sz):
                self.a = range(sz)
        f = Fatty(10000)
        print sys.getsizeof(f)
        print sys.getsizeof(f.a)
        print sum(map(sys.getsizeof, f.a))
```

```
64
80072
240000
```

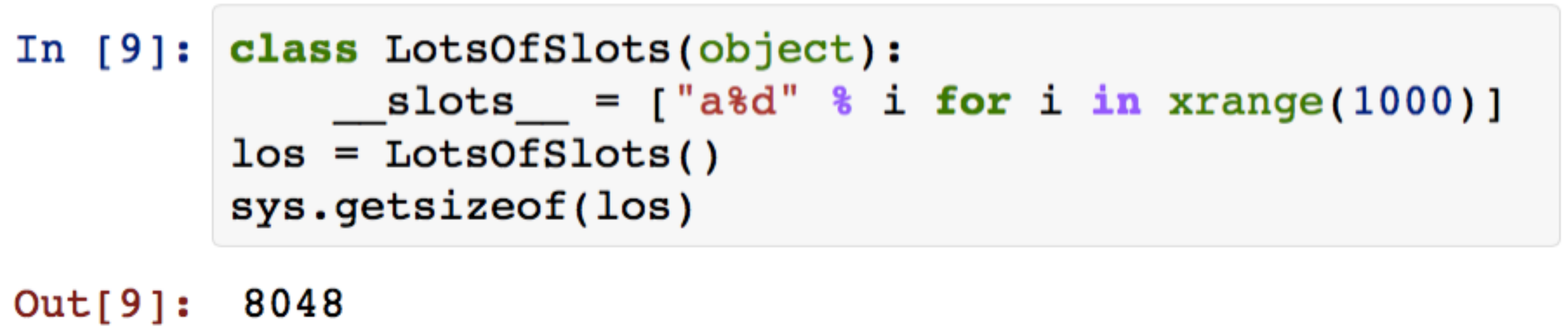
## More on getsizeof (2)

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

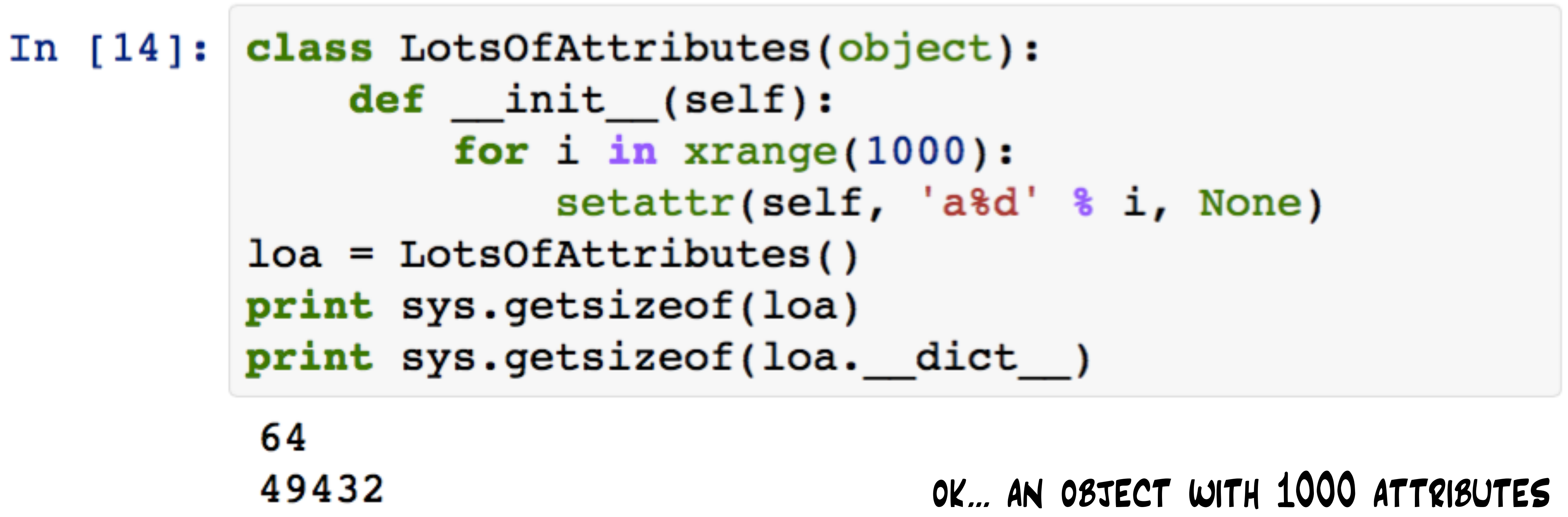


```
In [8]: class Munchausen(object):
            __slots__ = ()
            def __sizeof__(self):
                return 1000000
        baron = Munchausen()
        sys.getsizeof(baron)

Out[8]: 1000000
```

## getsizeof (slots vs. attributes)



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

Out[9]: 8048
```

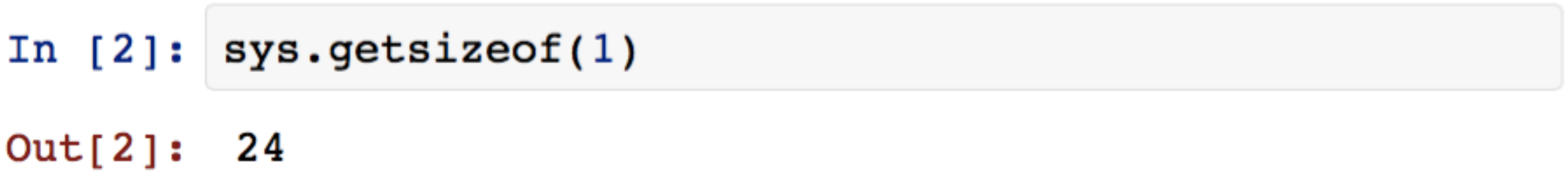


```
In [14]: class LotsOfAttributes(object):
             def __init__(self):
                 for i in xrange(1000):
                     setattr(self, 'a%d' % i, None)
         loa = LotsOfAttributes()
         print sys.getsizeof(loa)
         print sys.getsizeof(loa.__dict__)

         64
         49432
```

OK... an object with 1000 attributes is an exercise in bad design...

## Integers

- From the previous slides: 10000 ints = 240000 bytes
- Yes: 1 Integer = 24 bytes



```
In [2]: sys.getsizeof(1)

Out[2]:  24
```

- 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

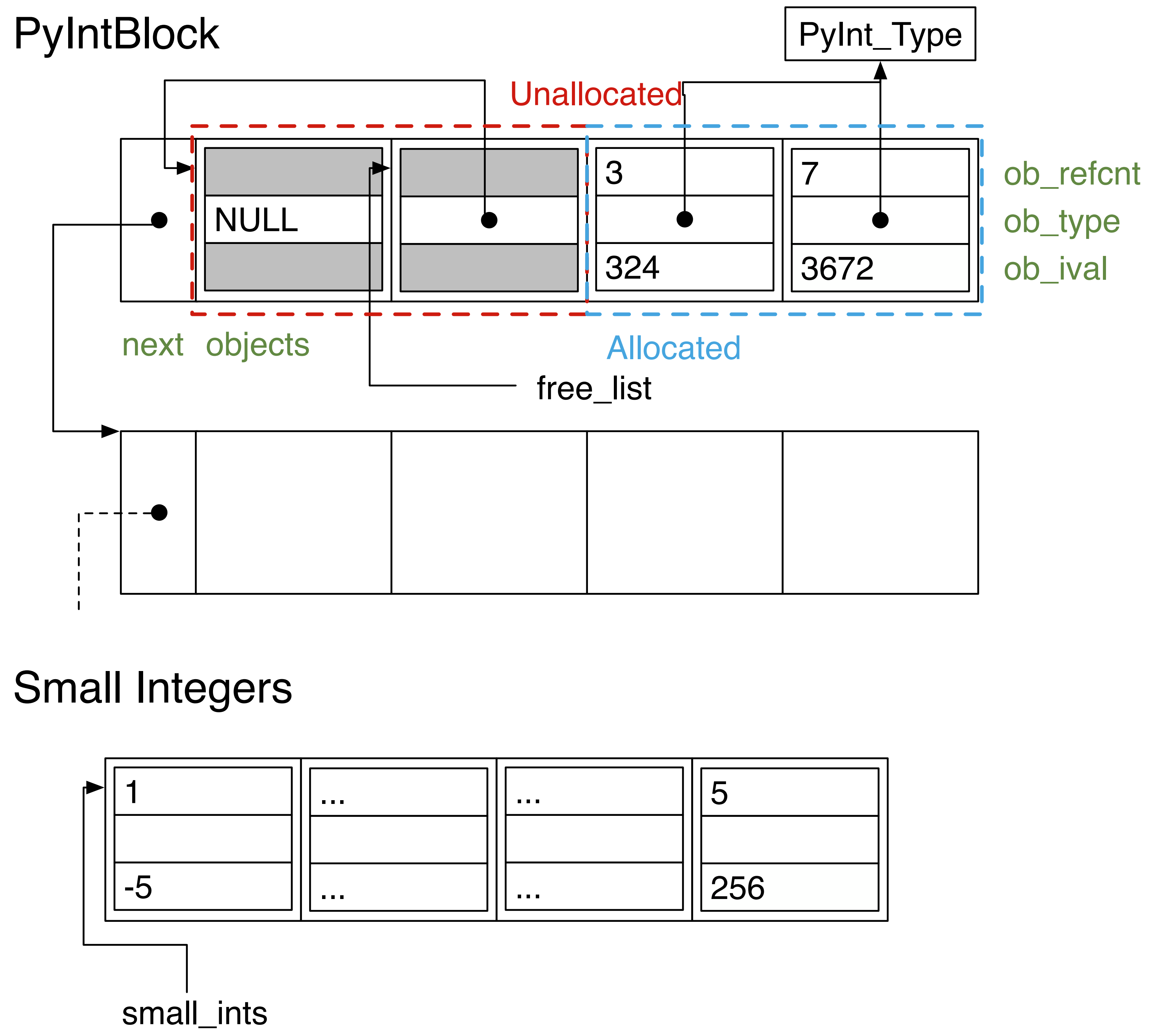
## More Integers

```
#define PyObject_HEAD                              \
    Py_ssize_t ob_refcnt;                          \
    struct _typeobject *ob_type;

typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

- 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



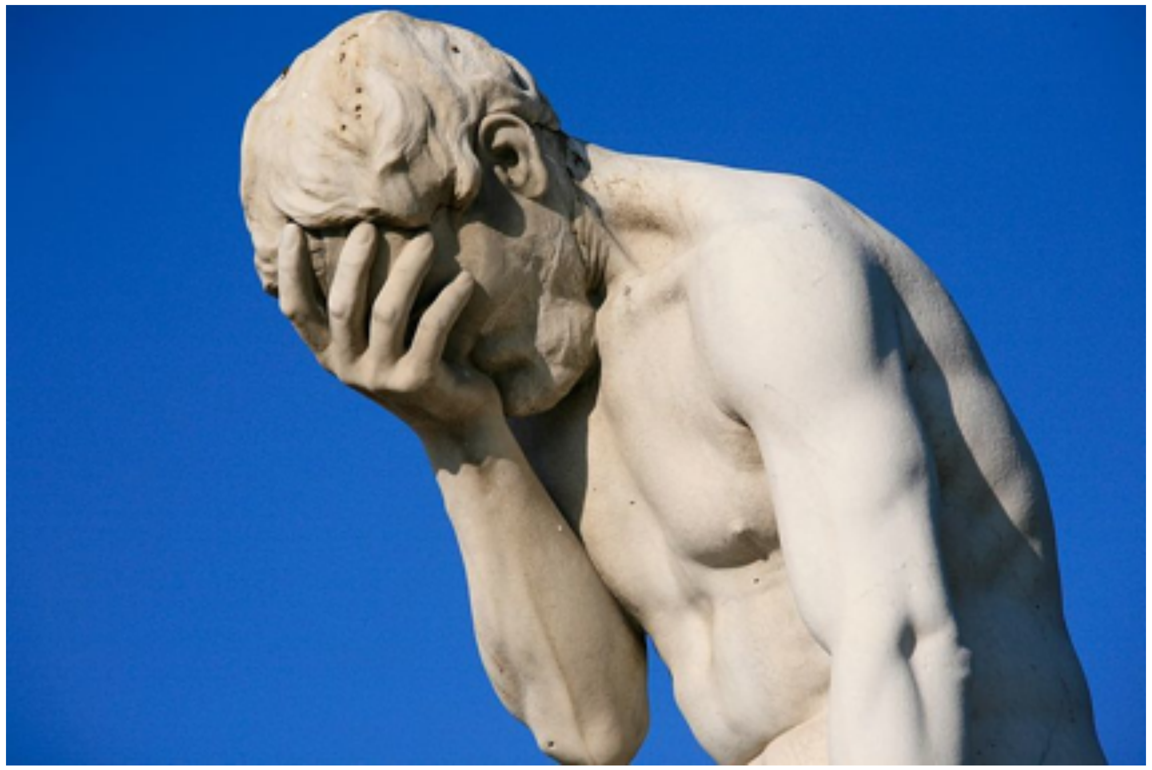
## All is well...

- 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)

```
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 ♫t♪ 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)



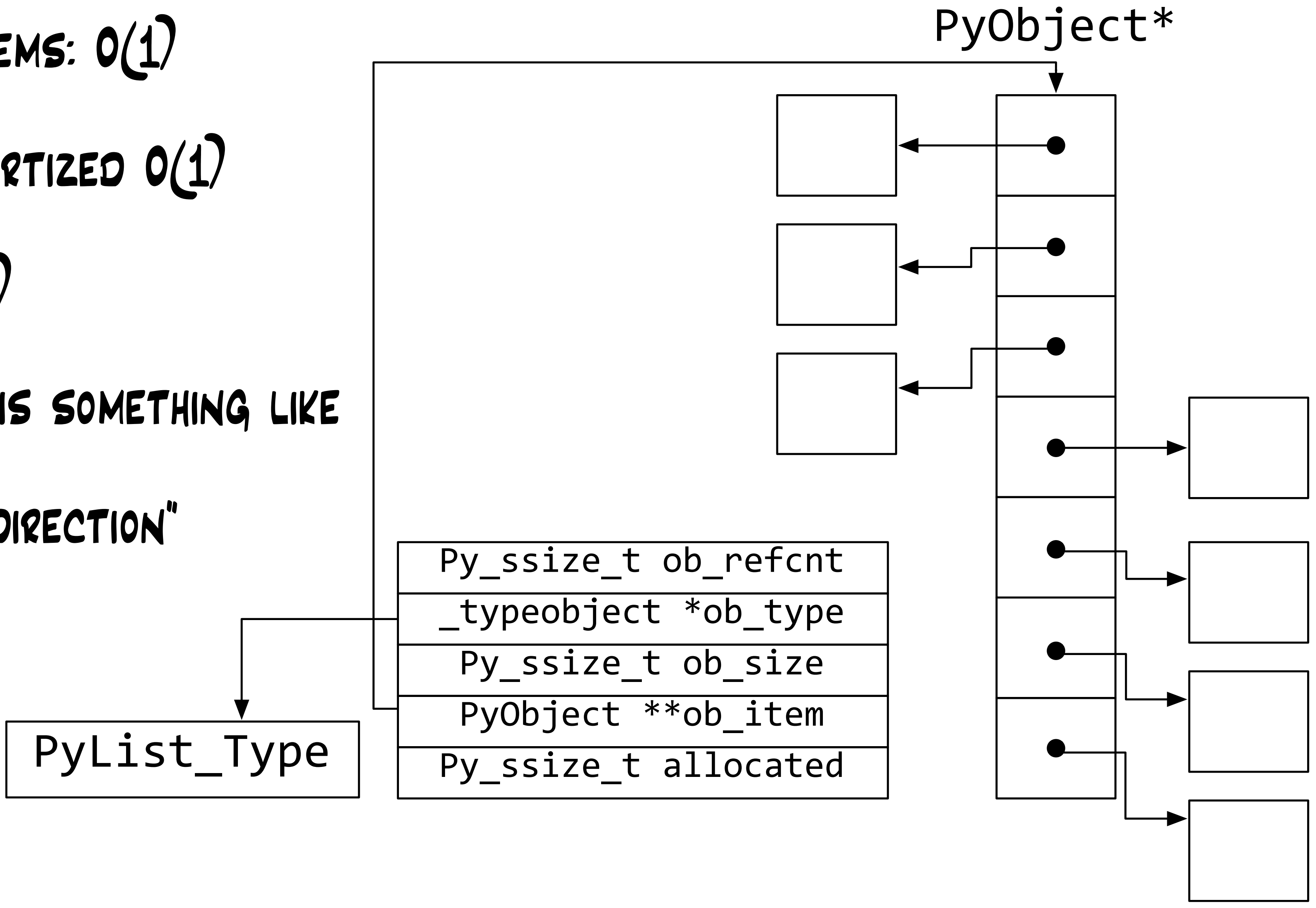
## Floats

- 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*y for x,y in izip([...], [...])]`

## Lists

- A Python list is implemented as an array of pointers to PyObjects
- 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

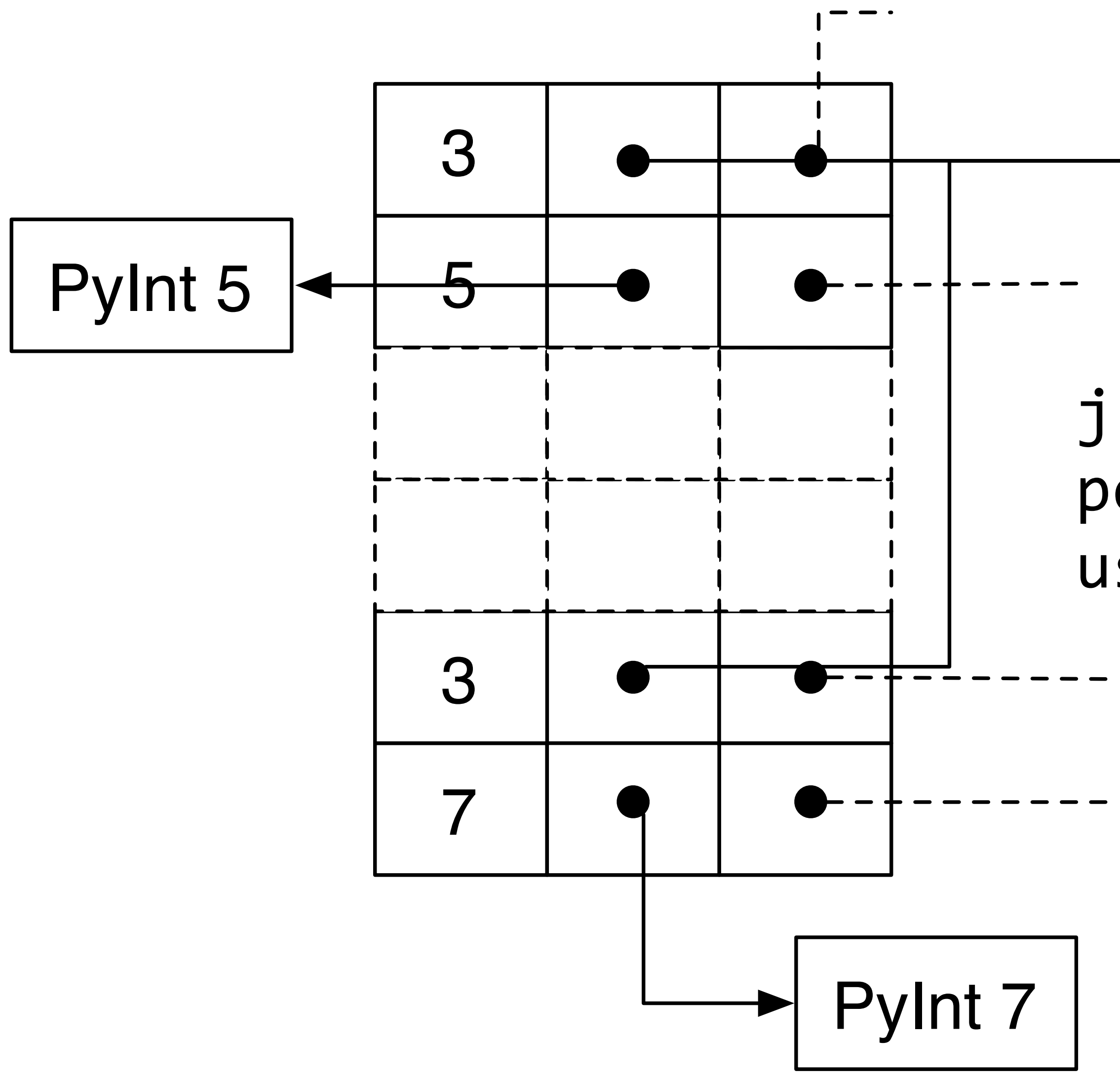
PyDictEntry

| |
|---|
| Py_ssize_t me_hash |
| PyObject* me_key |
| PyObject* me_value |

$$\frac{\text{sizeof(size\_t)} + 2 * \text{sizeof(pointer)}}{64 * 3 = 192 \text{ bits} = 24 \text{ bytes}}$$

| | |
|---|---|
| **Unused** | me_key = me_value = NULL |
| **Active** | me_value != NULL && me_key∉{NULL, dummy} |
| **Dummy** | me_key = dummy && me_value = NULL |

## Probe



```
j = (5*j) + 1 + perturb;
perturb >>= PERTURB_SHIFT;
use j % 2**i as the next table index;
```

## Dict

#define PyDict_MINSIZE 8

```
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!

```
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']}
```

Internal NetworkX representation
6 elements = 2.5 KB

```
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']}
```

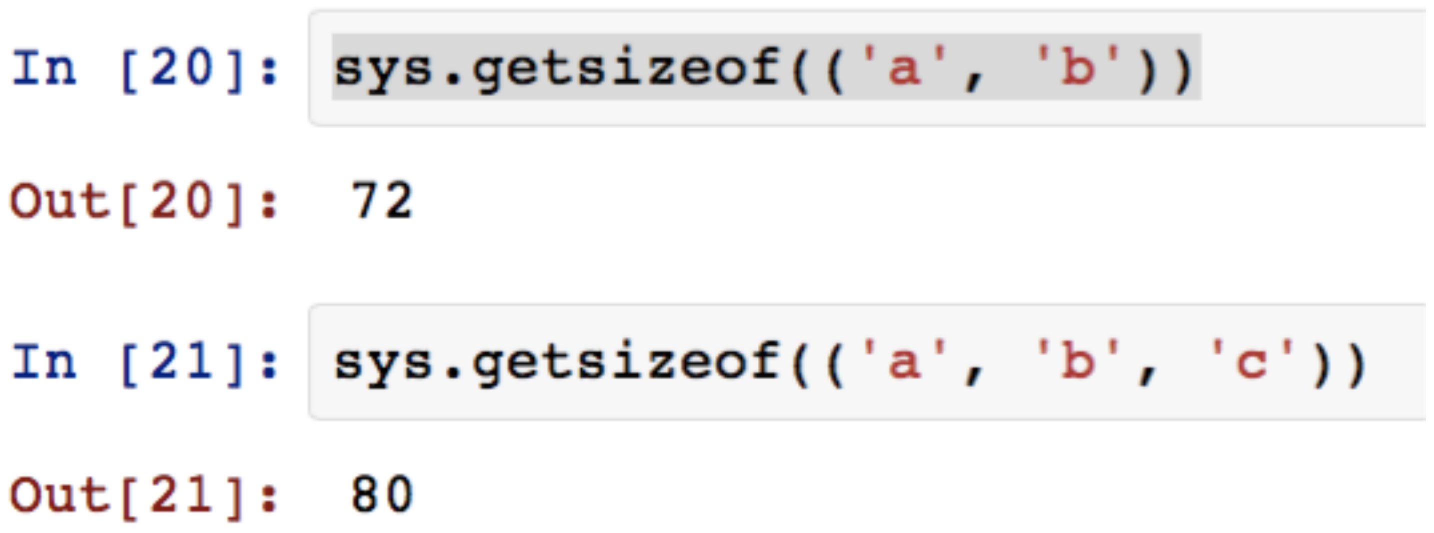
Internal NetworkX representation
6 elements = 2.5 KB

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

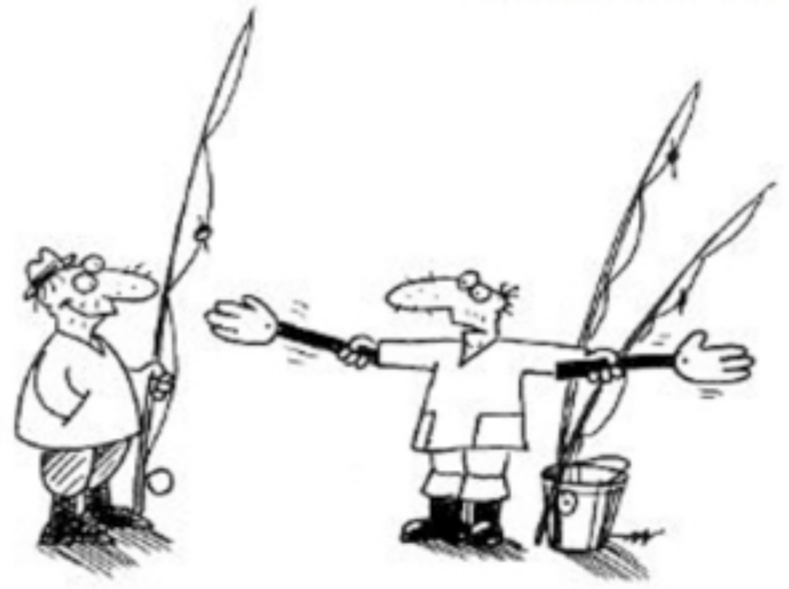
10^6 nodes with NX is 5 GB
as a sparse matrix it is < 2 MB

## 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

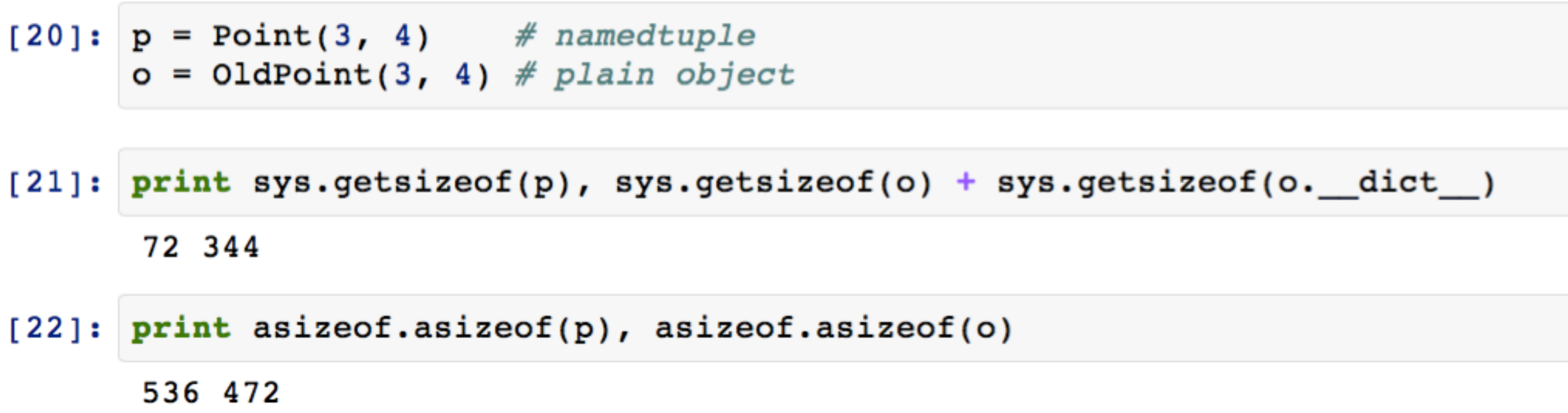


```
In [20]: sys.getsizeof(('a', 'b'))
Out[20]: 72

In [21]: sys.getsizeof(('a', 'b', 'c'))
Out[21]: 80
```



## How large is your object?

- getsizeof (already discussed)
- recursive (http://code.activestate.com/recipes/577504/)
- 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__

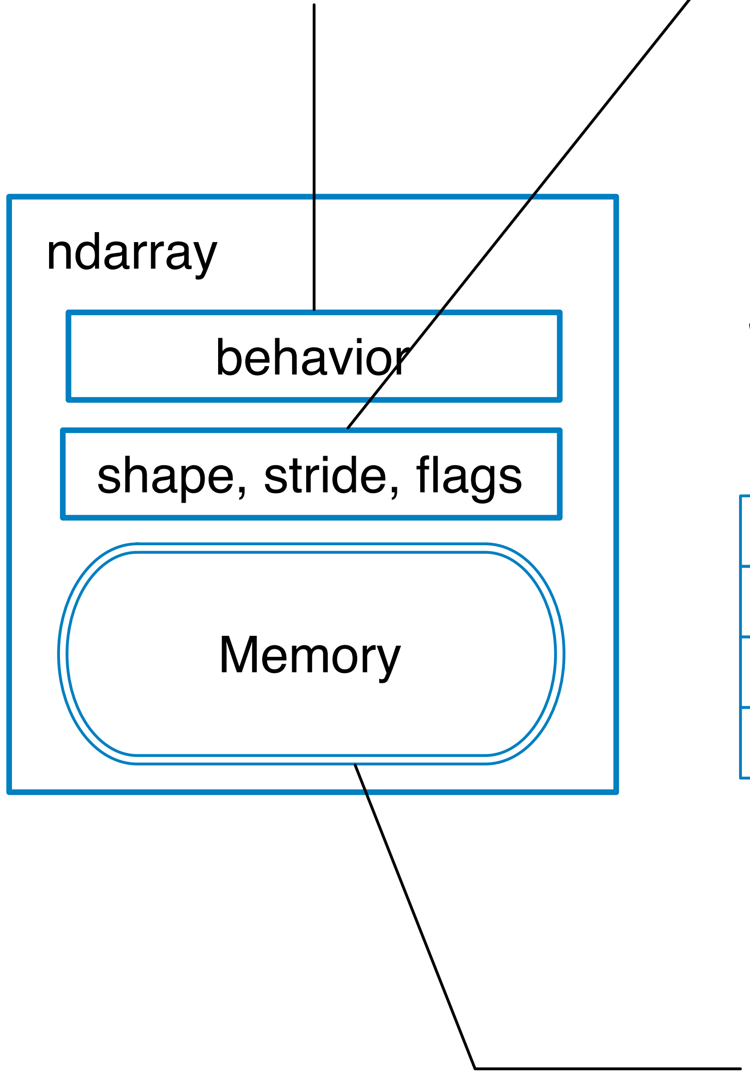
## Putting all together

- 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

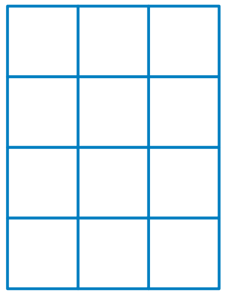
Is an object, so there is some behavior, e.g., the def. of __add__ and similar stuff

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



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

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

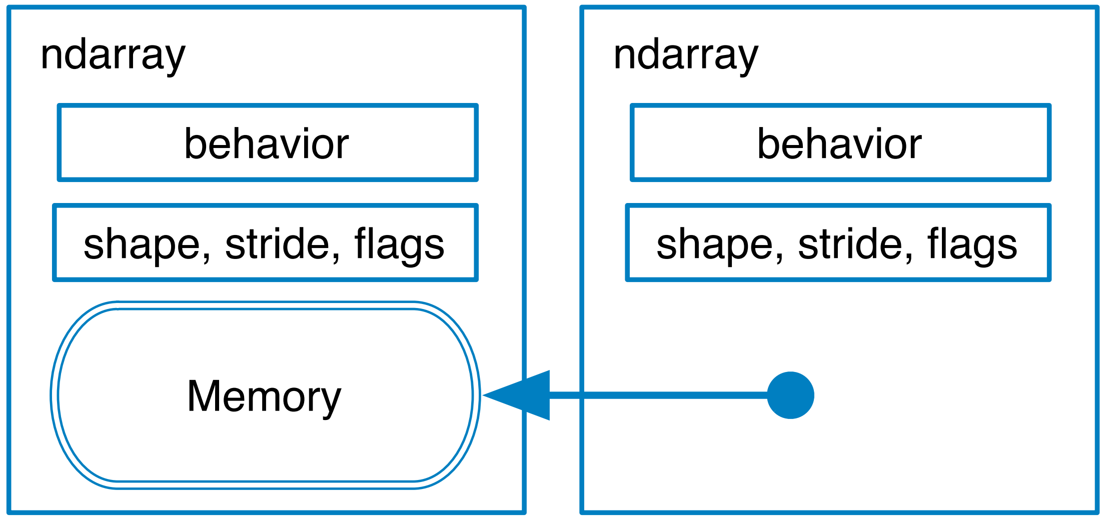


4x3

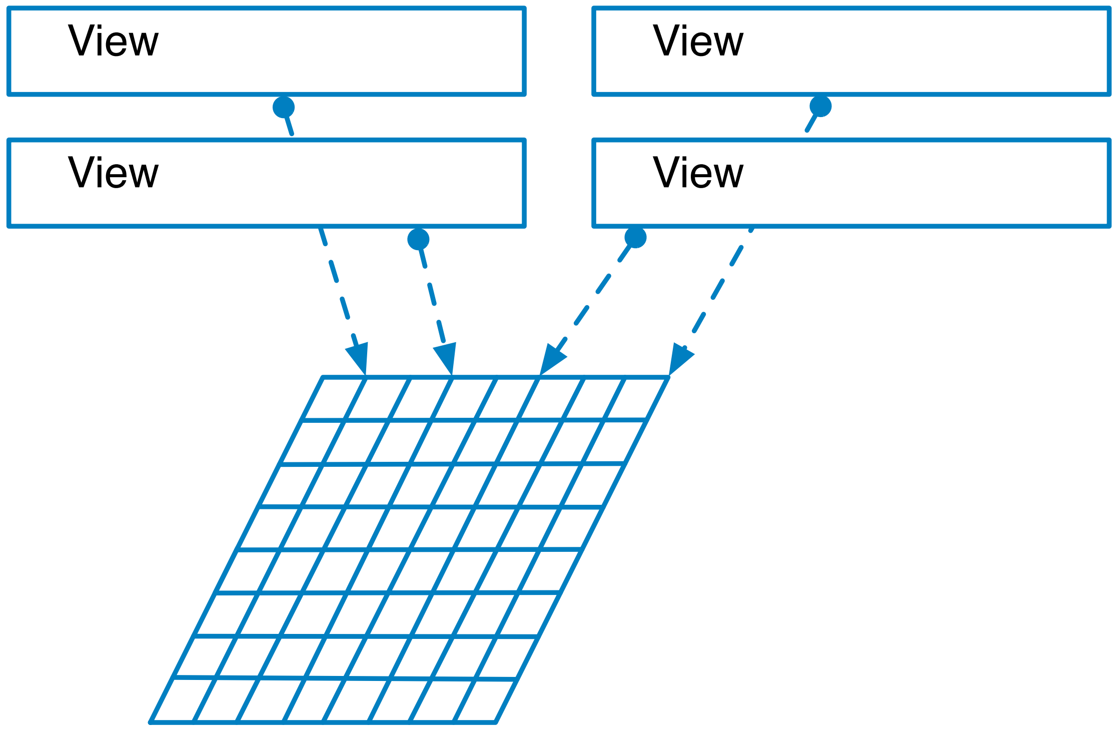
An n-dimensional array references some (usually contiguous memory area)

N-dimensional arrays are homogeneous

## 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

Running a set of programs to assess their relative performance

### Benchmarking

## Profiling vs. Benchmarking

Profiling

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

where is the problem?

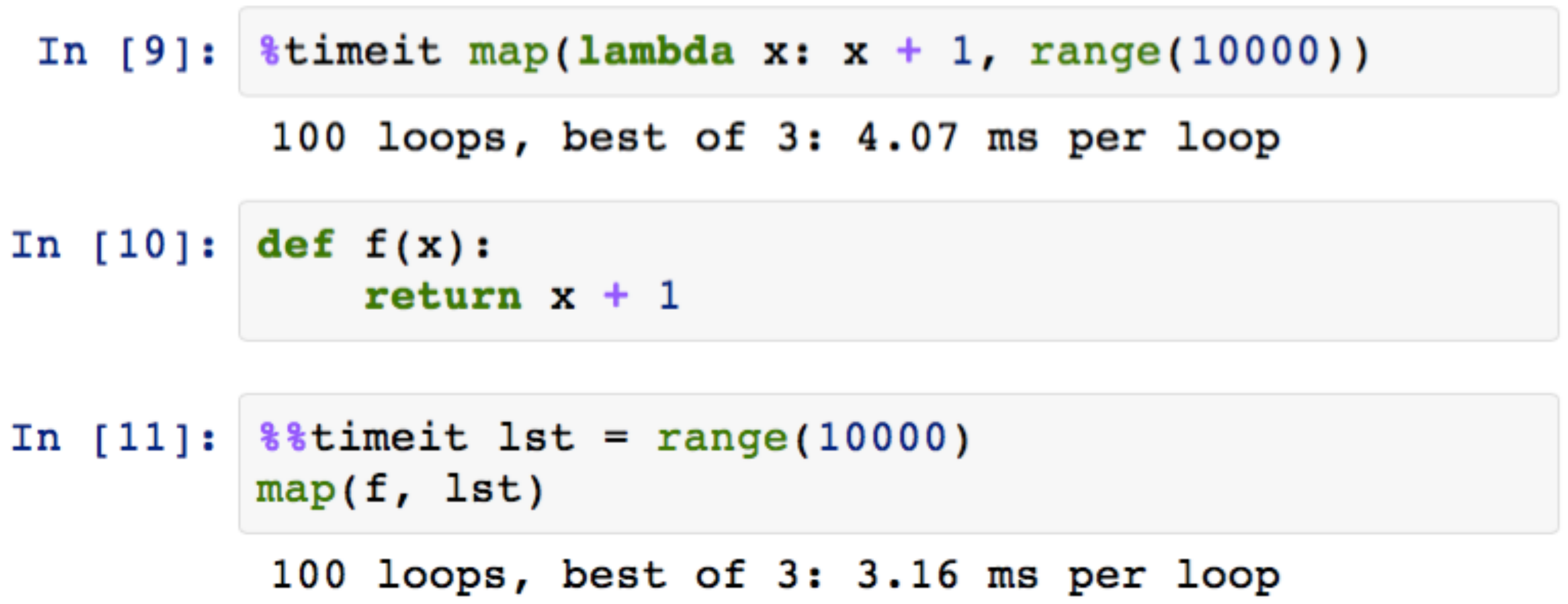
Is there a problem?

Running a set of programs to assess their relative performance

Benchmarking

## Benchmarking execution

- "manual solutions"
- timeit module
- ipython



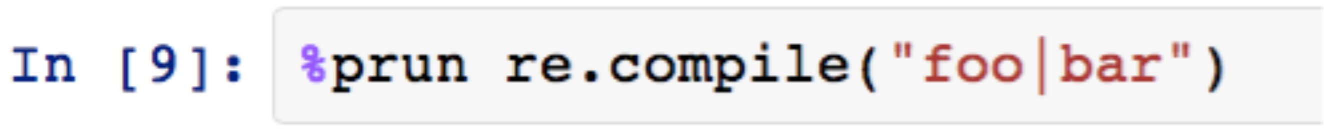
```
In [9]: %timeit map(lambda x: x + 1, range(10000))

        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"

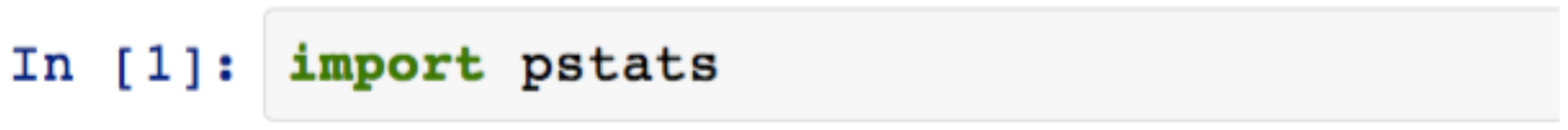
```
import cProfile
import re
cProfile.run('re.compile("foo|bar")') # to stdout
cProfile.run('re.compile("foo|bar")', 'restats')
```

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

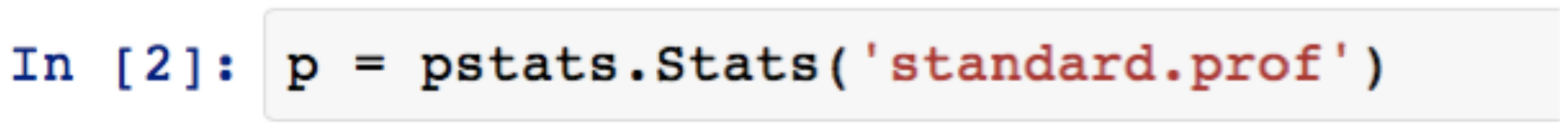


```
In [9]: %prun re.compile("foo|bar")
```

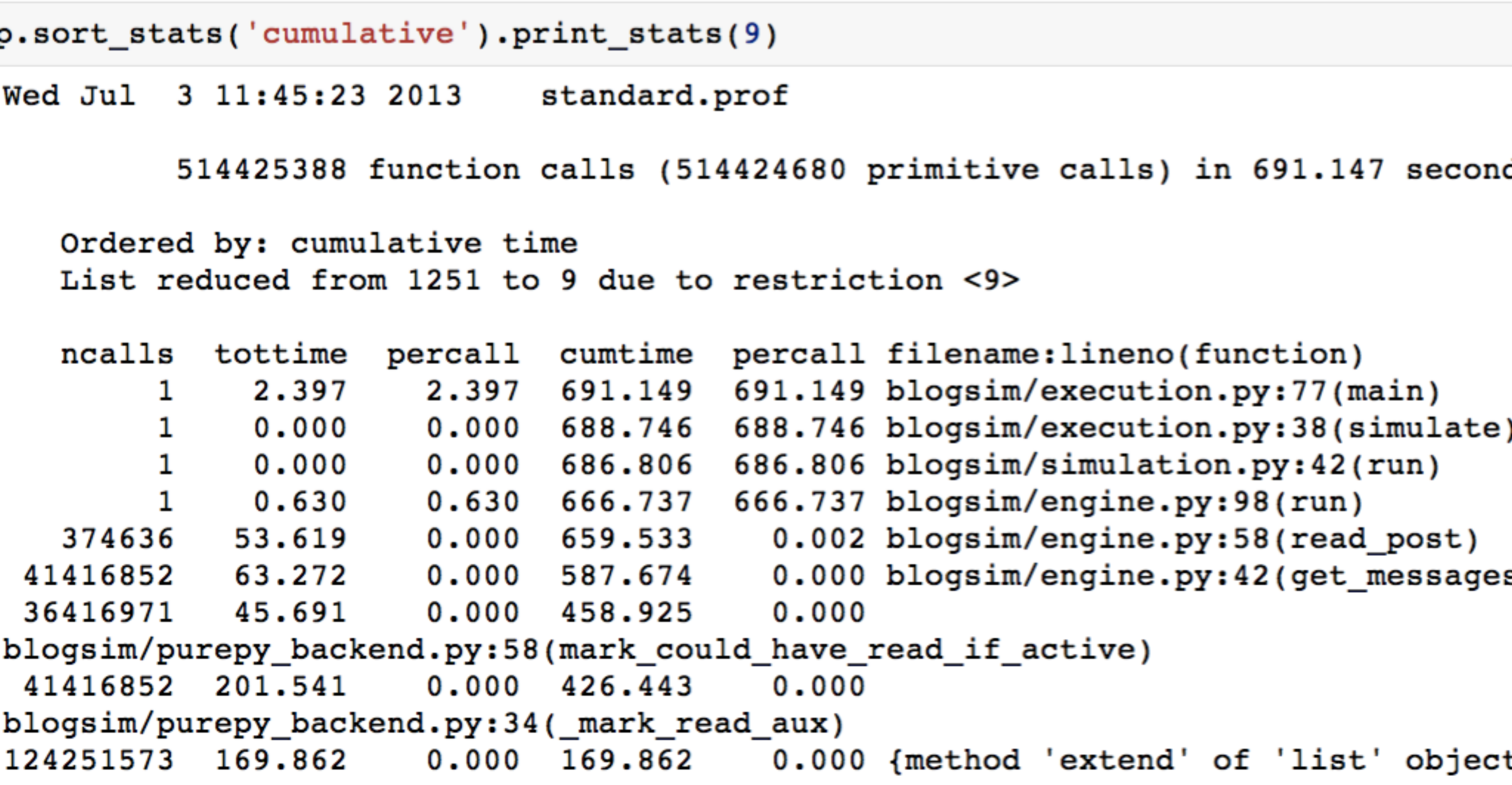
- WITH PSTAT YOU READ BACK THE PROFILING LOG AND EXAMINE IT



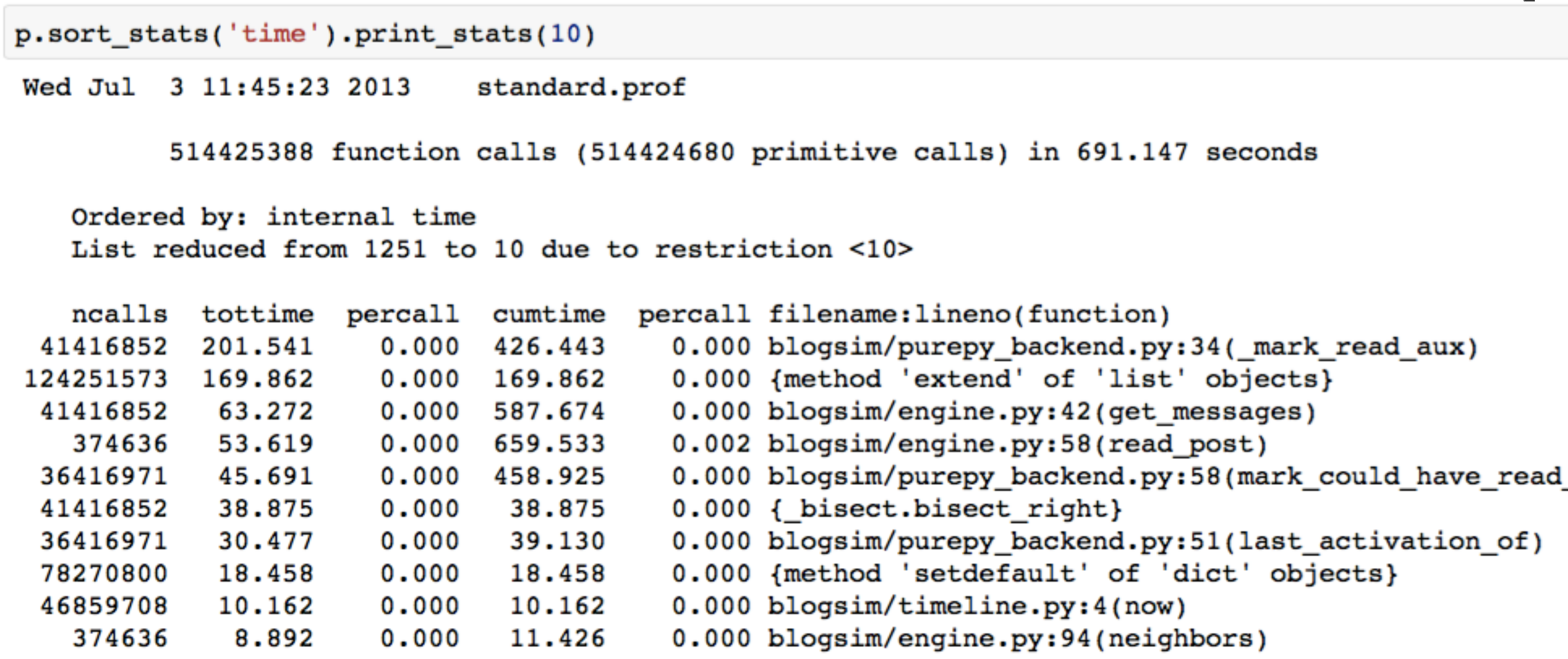
```
In [1]: import pstats
```



```
In [2]: p = pstats.Stats('standard.prof')
```



```
p.sort_stats('cumulative').print_stats(9)

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

         514425388 function calls (514424680 primitive calls) in 691.147 second

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

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



```
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>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 41416852  201.541    0.000  426.443    0.000 blogsim/purepy_backend.py:34(_mark_read_aux)
124251573  169.862    0.000  169.862    0.000 {method 'extend' of 'list' objects}
 41416852   63.272    0.000  587.674    0.000 blogsim/engine.py:42(get_messages)
   374636   53.619    0.000  659.533    0.002 blogsim/engine.py:58(read_post)
 36416971   45.691    0.000  458.925    0.000 blogsim/purepy_backend.py:58(mark_could_have_read_
 41416852   38.875    0.000   38.875    0.000 {_bisect.bisect_right}
 36416971   30.477    0.000   39.130    0.000 blogsim/purepy_backend.py:51(last_activation_of)
 78270800   18.458    0.000   18.458    0.000 {method 'setdefault' of 'dict' objects}
 46859708   10.162    0.000   10.162    0.000 blogsim/timeline.py:4(now)
   374636    8.892    0.000   11.426    0.000 blogsim/engine.py:94(neighbors)
```

## LINE_PROFILER

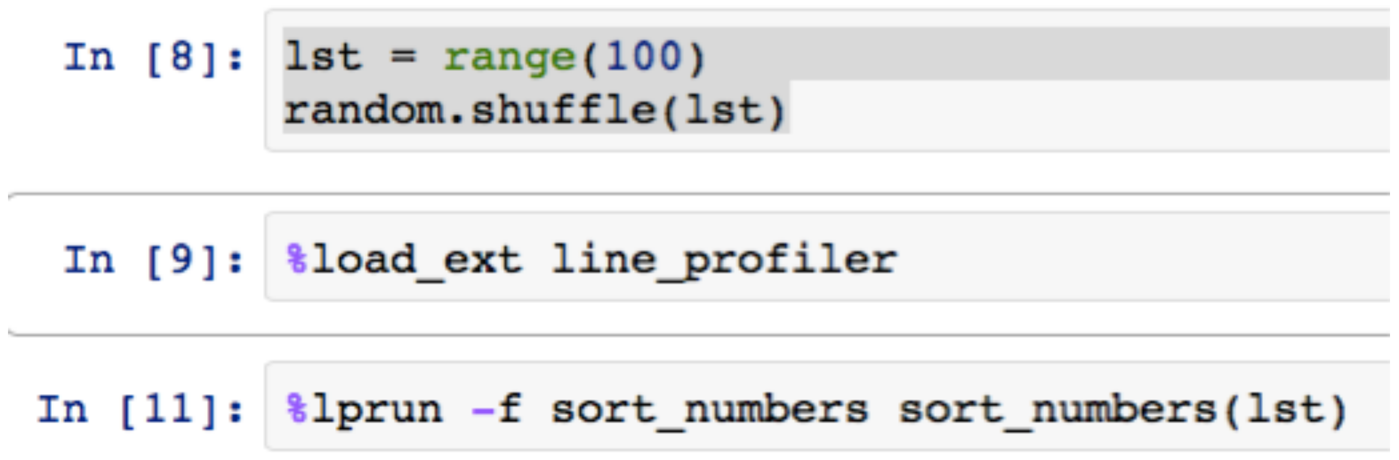
- If you really want to optimize a procedure, you need to know where time is spent inside the procedure!
- line_profiler (search the cheeseshop) does that!

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

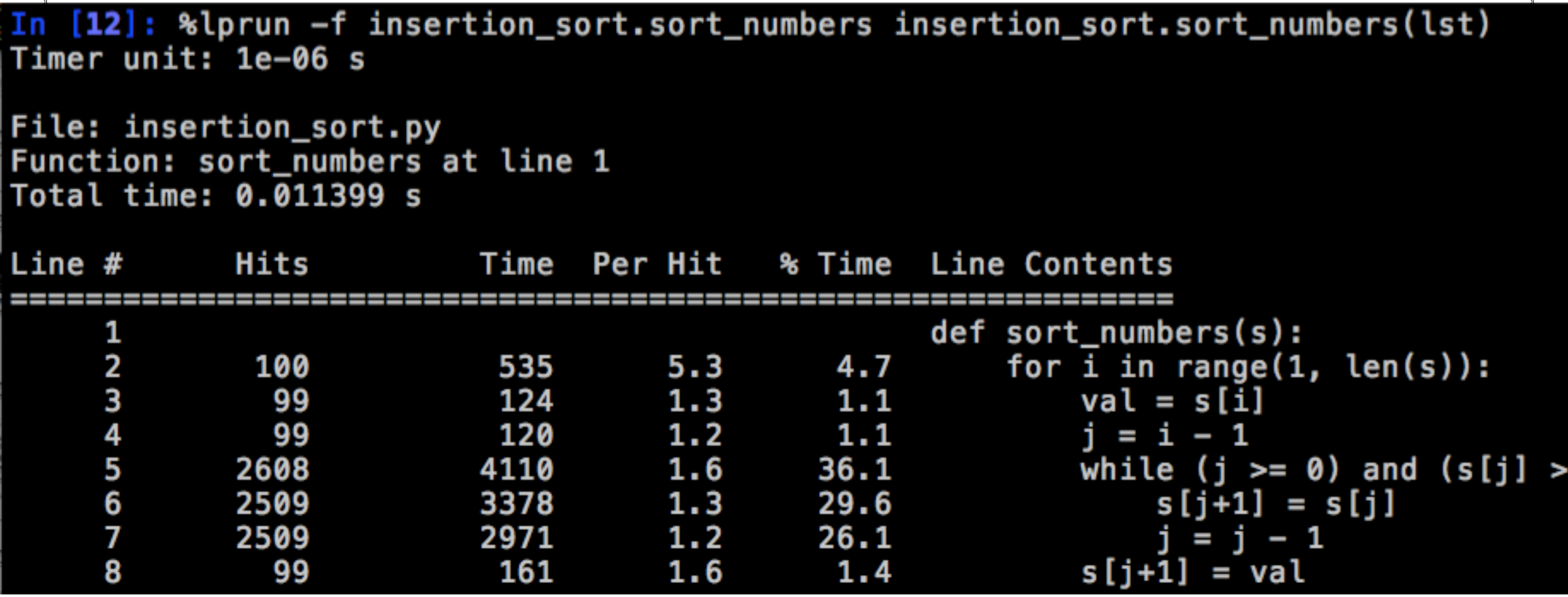
- Run the program as usual
- Read the results with

```
$ python -m line_profiler script_to_profile.py.lprof
```

## LINE_PROFILER

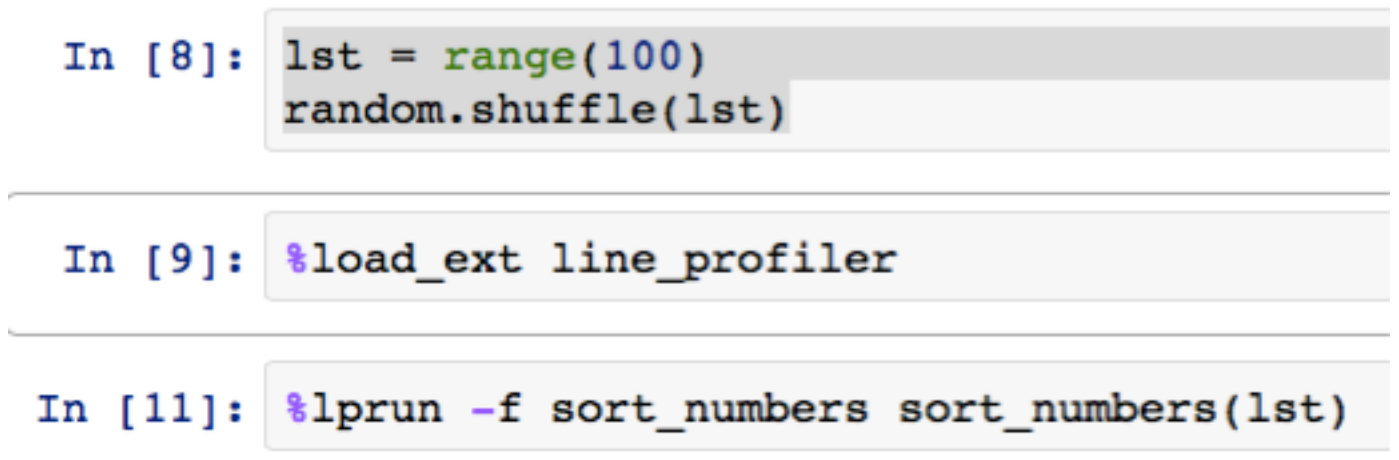


```
In [8]:  lst = range(100)
         random.shuffle(lst)

In [9]:  %load_ext line_profiler

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

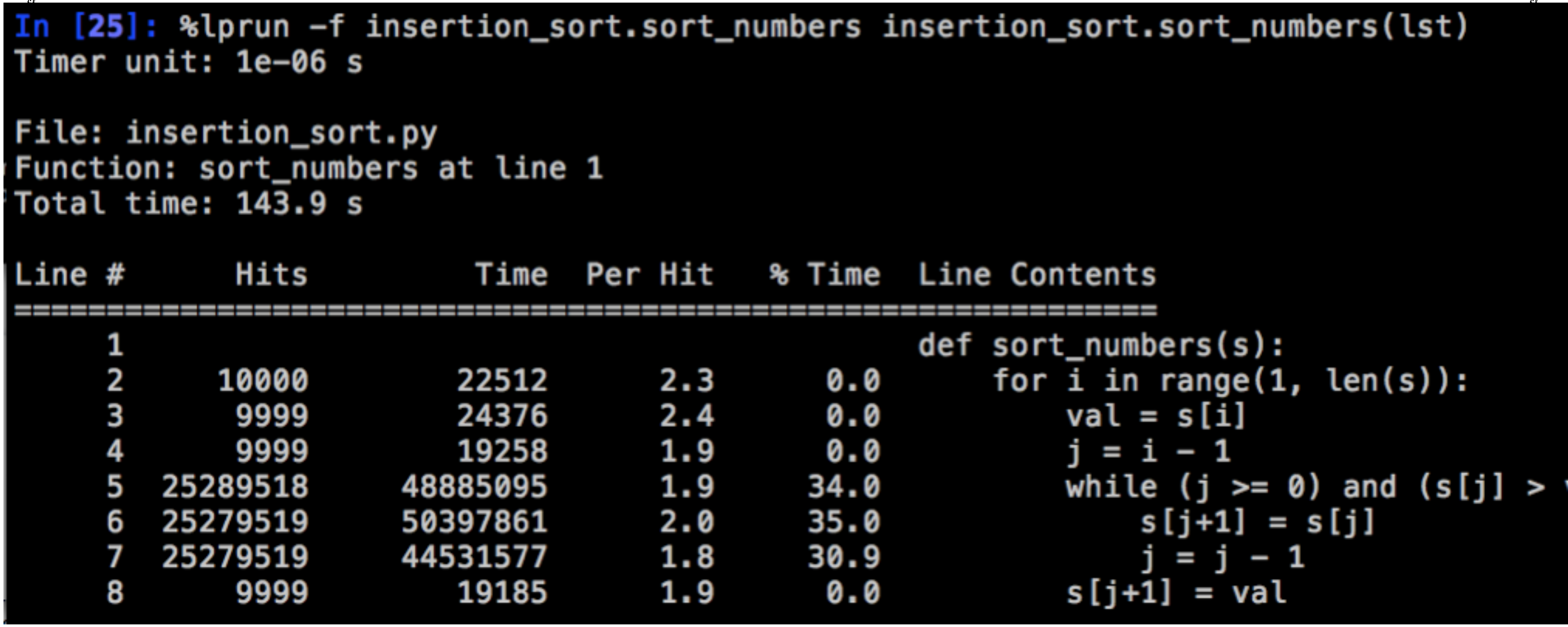


```
In [12]: %lprun -f insertion_sort.sort_numbers insertion_sort.sort_numbers(lst)
Timer unit: 1e-06 s

File: insertion_sort.py
Function: sort_numbers at line 1
Total time: 0.011399 s

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sort_numbers(s):
     2       100          535      5.3      4.7      for i in range(1, len(s)):
     3        99          124      1.3      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
```

## LINE_PROFILER



```
In [8]:  lst = range(100)
         random.shuffle(lst)

In [9]:  %load_ext line_profiler

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

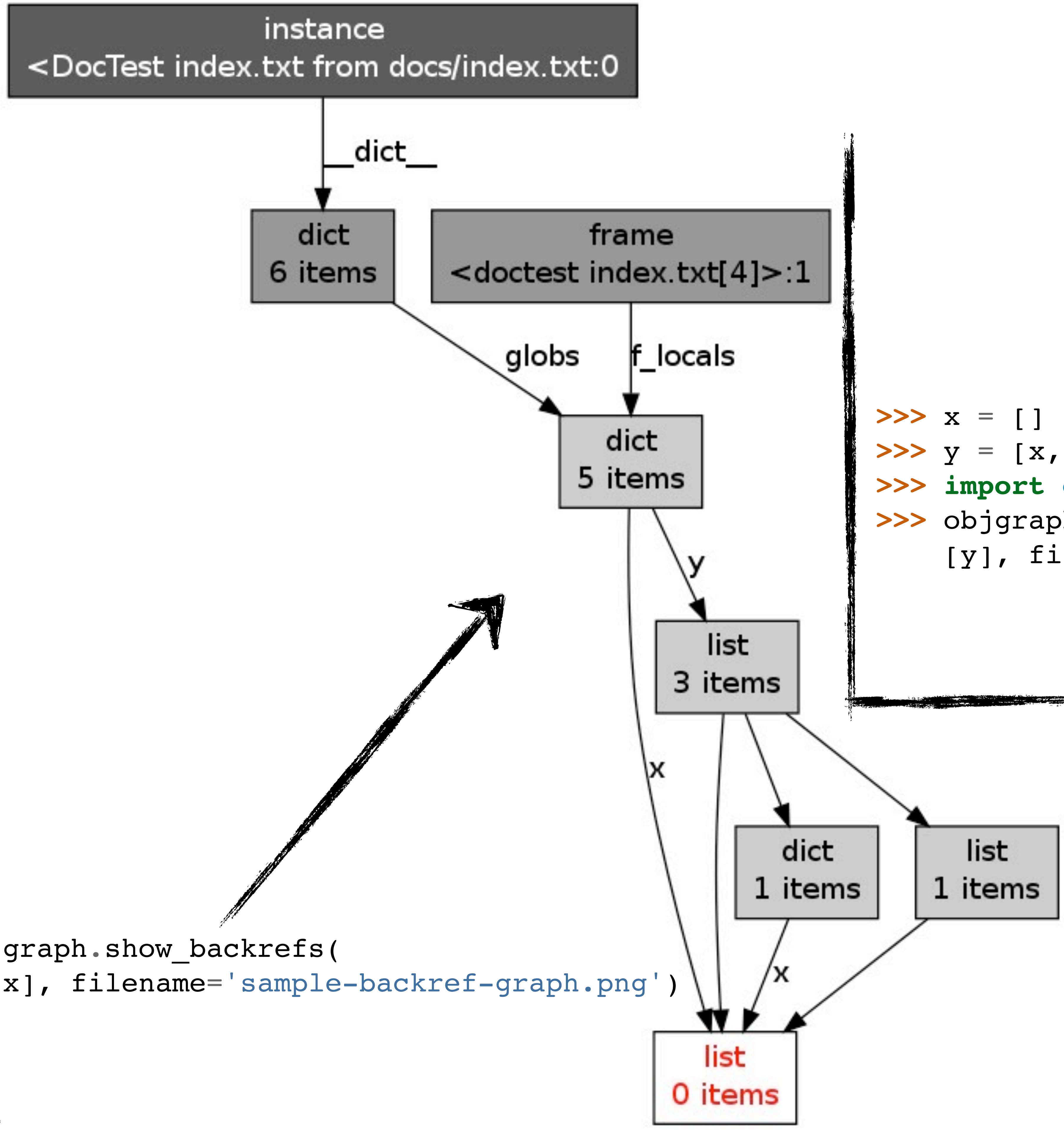


```
In [25]: %lprun -f insertion_sort.sort_numbers insertion_sort.sort_numbers(lst)
Timer unit: 1e-06 s

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

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

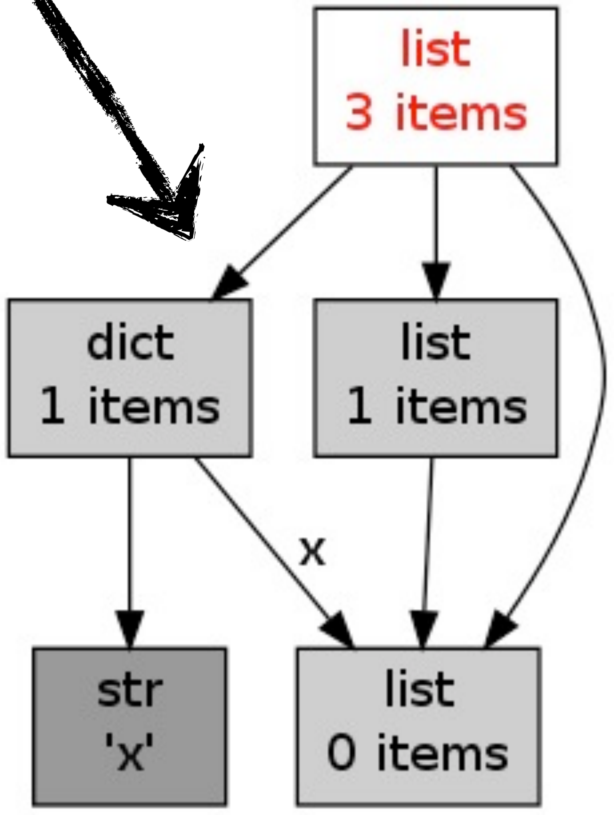
## Memory Profiling

- 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)

## OBJGRAPH

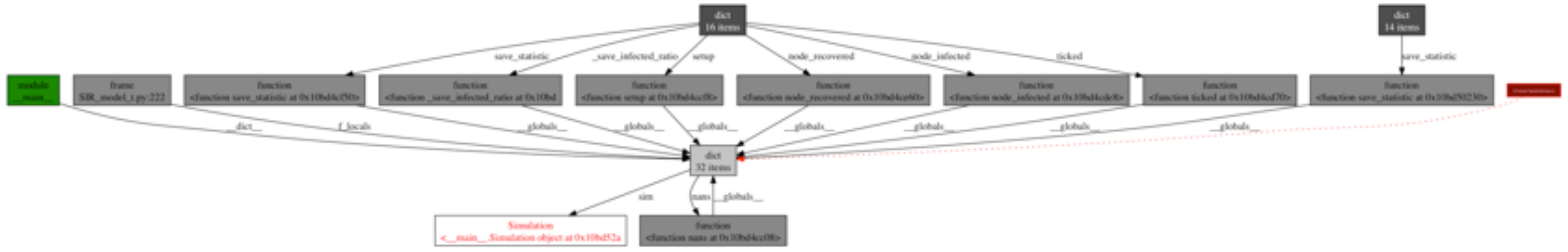


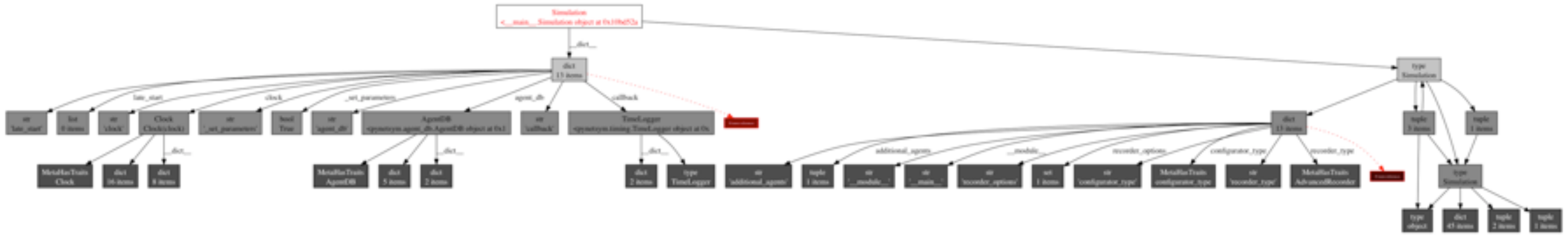
```
>>> x = []
>>> y = [x, [x], dict(x=x)]
>>> import objgraph
>>> objgraph.show_refs(
    [y], filename='sample-graph.png')
```



```
objgraph.show_backrefs(
  [x], filename='sample-backref-graph.png')
```

## (more objgraph)





We would need xdot to navigate the graph interactively!

## Meliae

```
>>> om = loader.load('my.dump')

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

Index   Count   %      Size   % Cum     Max Kind
    0 2375950  46 224148214  58  58 4194313 str
    1   63209   1  77855404  20  78 3145868 dict
    2 1647097  32  29645488   7  86      20
bzrlib._static_tuple_c.StaticTuple
    3  374259   7  14852532   3  89     304 tuple
    4  138464   2  12387988   3  93     536 unicode
 ...
```

## How to get faster?

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

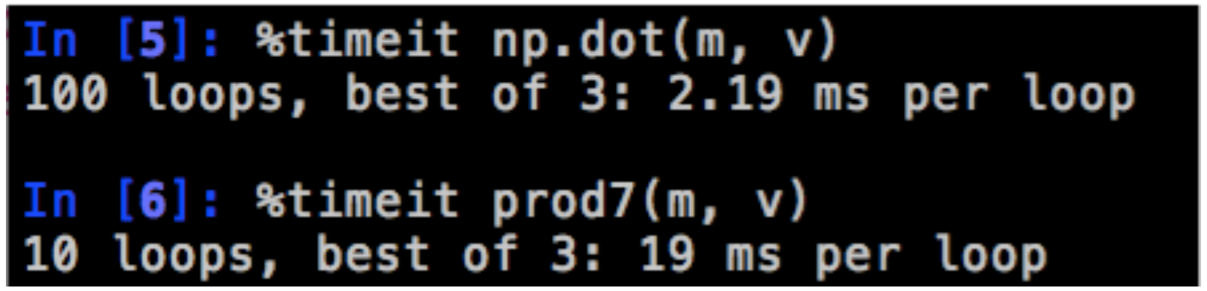
```
def prod7(m, v):
    nrows, ncolumns = m.shape
    res = np.zeros(nrows)
    code = r"""
for (int i=0; i<nrows; i++) {
for (int j=0; j<ncolumns; j++) {
res(i) += m(i, j)*v(j); }
} """
    err = weave.inline(code,
        ['nrows', 'ncolumns', 'res', 'm', 'v'],
        type_converters=weave.converters.blitz,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

## How to get faster?

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

```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')
```

Numpy is still faster!

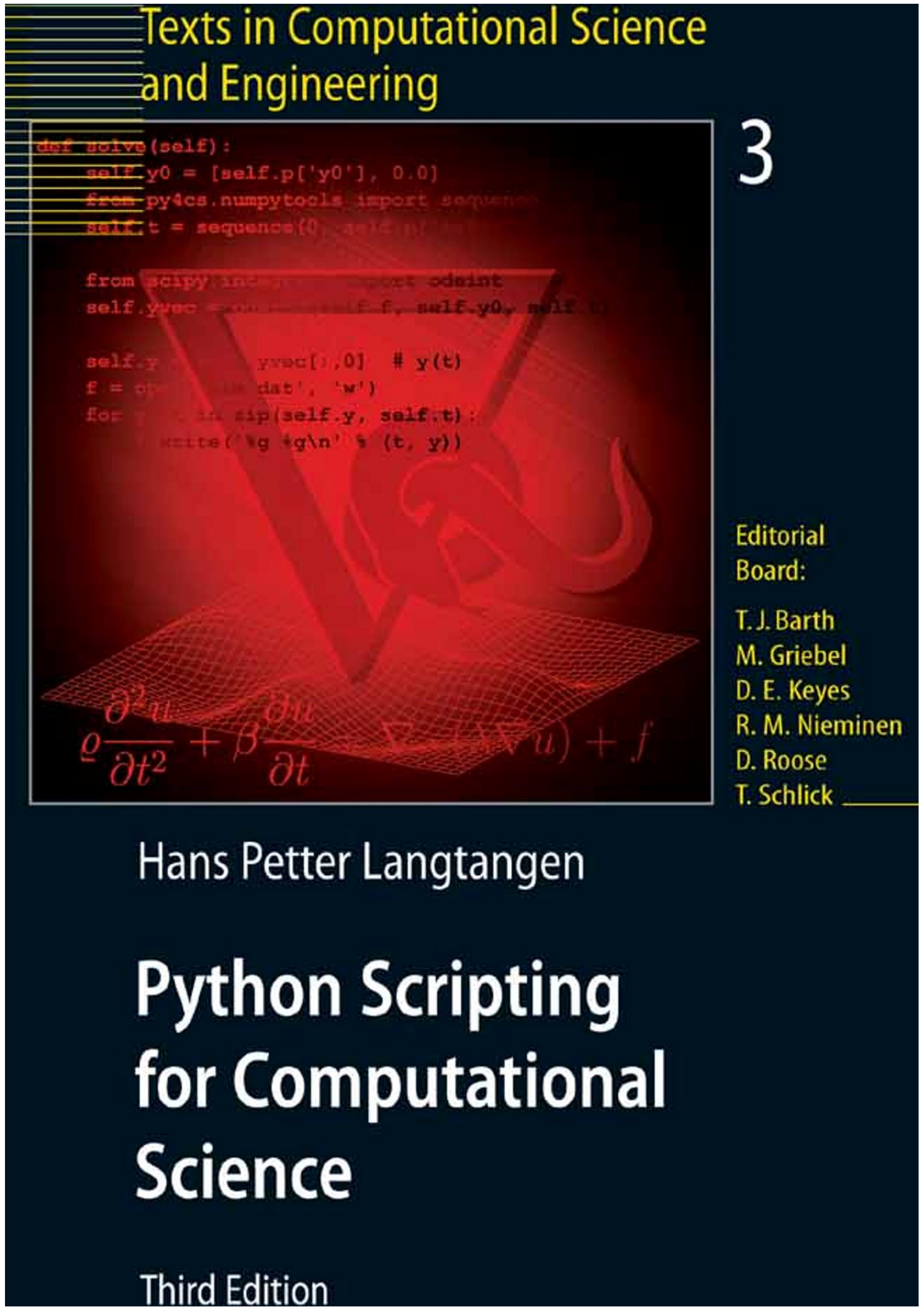


```
In [5]: %timeit np.dot(m, v)
100 loops, best of 3: 2.19 ms per loop

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

## Performance Hints

| method | function name | CPU time |
|---|---|---|
| pure Python loops | `prod1` | 490 |
| `map/reduce` | `prod2` | 454 |
| `map/reduce` | `prod3` | 209 |
| Psyco | `prod6` | 327 |
| Fortran | `prod4` | 2.9 |
| Fortran, cache-friendly loops | `prod5` | 1.0 |
| Weave | `prod7` | 1.6 |
| NumPy | `dot` | 1.0 |

2K x 2K dense matrix multiplied with 2K Array



## Cython

- Creates a compiled python module
- Cython -> C -> native code
- interoperates with numpy

```
def naive_convolve(np.ndarray[DTYPE_t, ndim=2] f not None,
                   np.ndarray[DTYPE_t, ndim=2] g not None):
```

- to some extent, with C++
- allows to declare buffers
- Type annotations
- "Objects" with methods that are not Python accessible

## Mongo + Numpy = Monary

```
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:                     # prove that we did something...
    print numpy.mean(array)
```

- 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)