Speeding Python
Python in HPC
TACC Training, Oct. 15, 2012

Presenters:
Andy R. Terrel, PhD
Texas Advanced Computing Center
University of Texas at Austin

Yaakoub El Khamra
Texas Advanced Computing Center
University of Texas at Austin

0.1 Interacting with the Tutorial Slides

This tutorial is an interactive worksheet designed to encourage you to try out the lessons during the demonstration. If you are looking at the pdf version, we encourage you to download the updated version (see previous slide) and try the interactive version.

To run the interactive version, you need a good Python environment including:

- IPython version >= 13.0
- Numpy version >= 1.5
- Scipy
- Matplotlib

Move to the directory containing the tarball and execute:

```
$ ipython notebook --pylab=inline
```

We heartily endorse the Anaconda distribution and the Free Enthought Python Distribution.

0.2 Presentation mode

The slide show mode is only supported by an IPython development branch version. To get it I recommend cloning from the official branch, adding Matthias Carreau's remote, fetching and using his branch slideshow_extension2. Here are the commands:

```
git clone git://github.com/ipython/ipython.git # Official clone
cd ipython
git remote add carreau git://github.com/Carreau/ipython.git # Matthias' branch
git fetch carreau # Fetch the branches
git checkout carreau/slideshow_extension2 # Checkout the slideshow extension
python setup.py develop # Install the development version
ipython notebook # Check out the slideshows.
```

1 How Slow is Python

Let's add one to a million number

```
lst = range(1000000) # A pure Python list
%timeit [i + 1 for i in lst] # A Python list comprehension (iteration happens in C but
    with PyObjects)
```

```
10 loops, best of 3: 79.2 ms per loop
```

1.1 Why is Python Slow?

Dynamic typing requires lots of metadata around variable.

- Python uses heavy frame objects during iteration

1.1.1 Solution:

- Make an object that has a single type and continuous storage.
- Implement common functionality into that object to iterate in C.

```
arr = arange(1000000) # A NumPy list of integers
%timeit arr + 1 # Use operator overloading for nice syntax, now iteration is in C with
    ints
```

```
100 loops, best of 3: 3.33 ms per loop
```

1.2 What makes NumPy so much faster?

- Data layout
- homogenous: every item takes up the same size block of memory
- single data-type objects
- powerful array scalar types
- universal function (ufuncs)
- function that operates on ndarrays in an element-by-element fashion
- vectorized wrapper for a function
- built-in functions are implemented in compiled C code

1.3 Numpy Data layout

- homogenous: every item takes up the same size block of memory
- single data-type objects
- powerful array scalar types

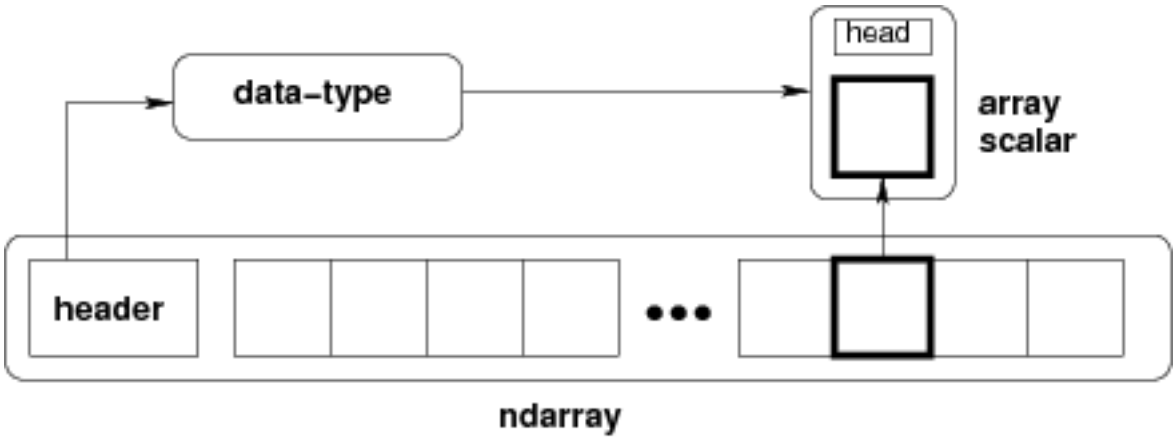


Figure 1: three fundamental

1.4 Numpy Universal Functions (ufuncs)

- function that operates on ndarrays in an element-by-element fashion
- vectorized wrapper for a function
- built-in functions are implemented in compiled C code

```
%timeit [sin(i)**2 for i in arr]
```

```
1 loops, best of 3: 5.65 s per loop
```

```
%timeit np.sin(arr)**2
```

```
10 loops, best of 3: 29.1 ms per loop
```

1.5 Broadcasting Arrays

Computing a 3D grid of distances $R_{ijk} = \sqrt{(i^2 + j^2 + k^2)}$

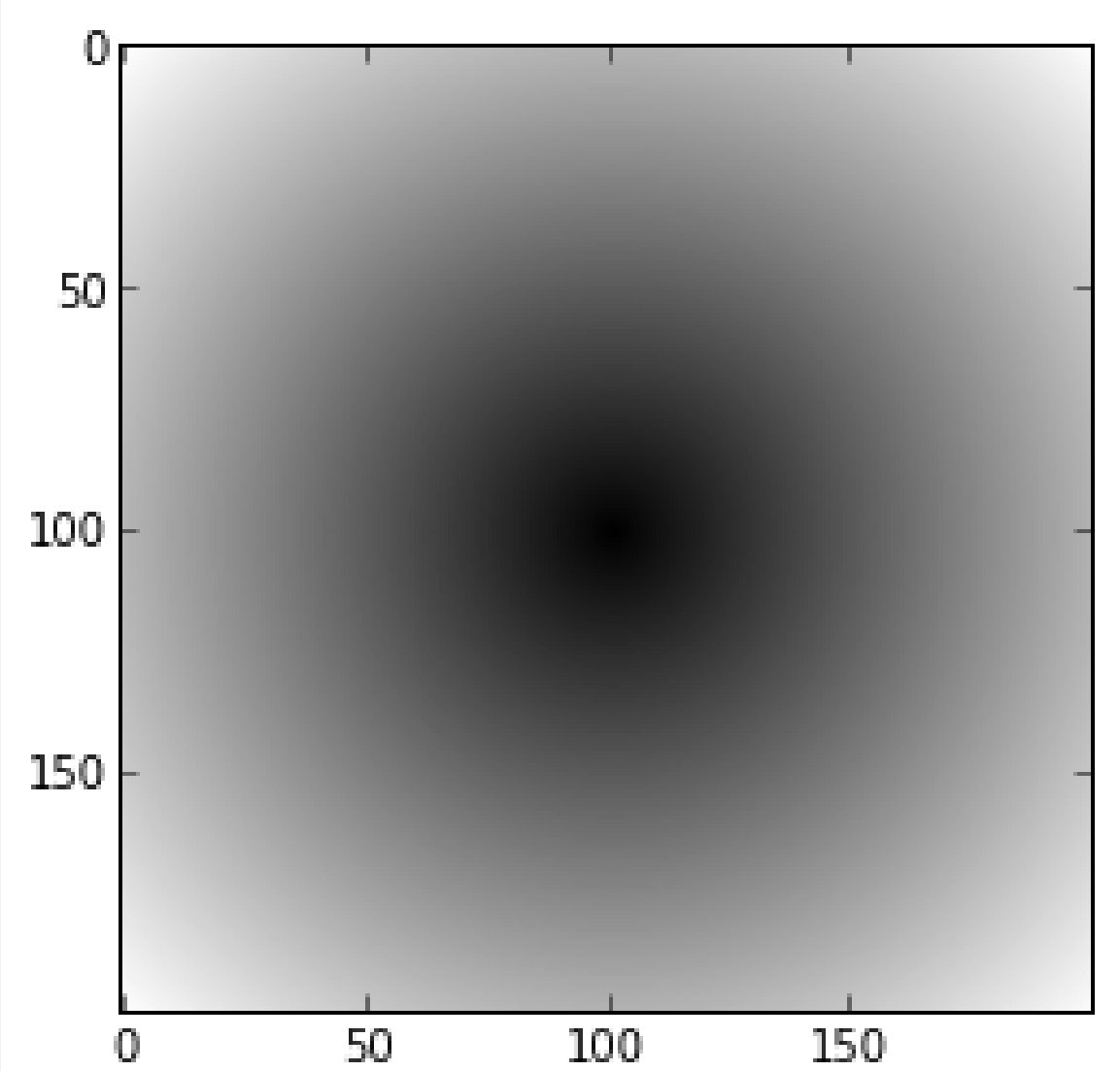
```
i, j, k = np.mgrid[-100:100, -100:100, -100:100]
print (i.shape, j.shape, k.shape)
```

```
((200, 200, 200), (200, 200, 200), (200, 200, 200))
```

```
R = np.sqrt(i**2 + j**2 + k**2)
R.shape
```

```
(200, 200, 200)
```

```
_ = imshow(R[100]/amax(R), cmap=cm.gray)
```


0
50
100
150
0
50
100
150

1.6 Broadcasting Arrays

But storing three dense arrays for a 3D grid is wasteful.

NumPy can broadcast across dimensions thus only working on the outer grids

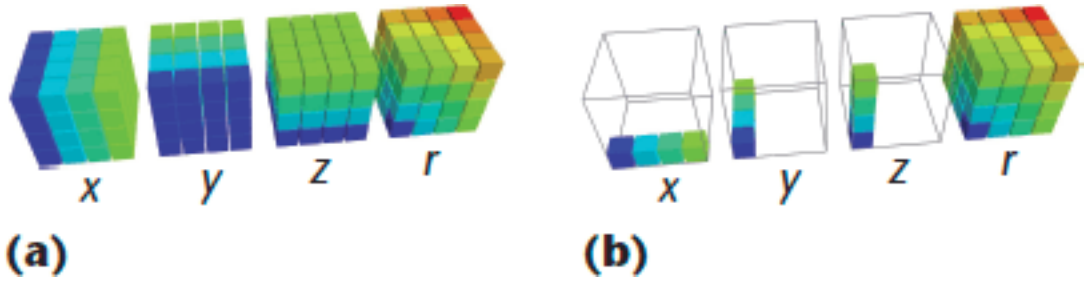
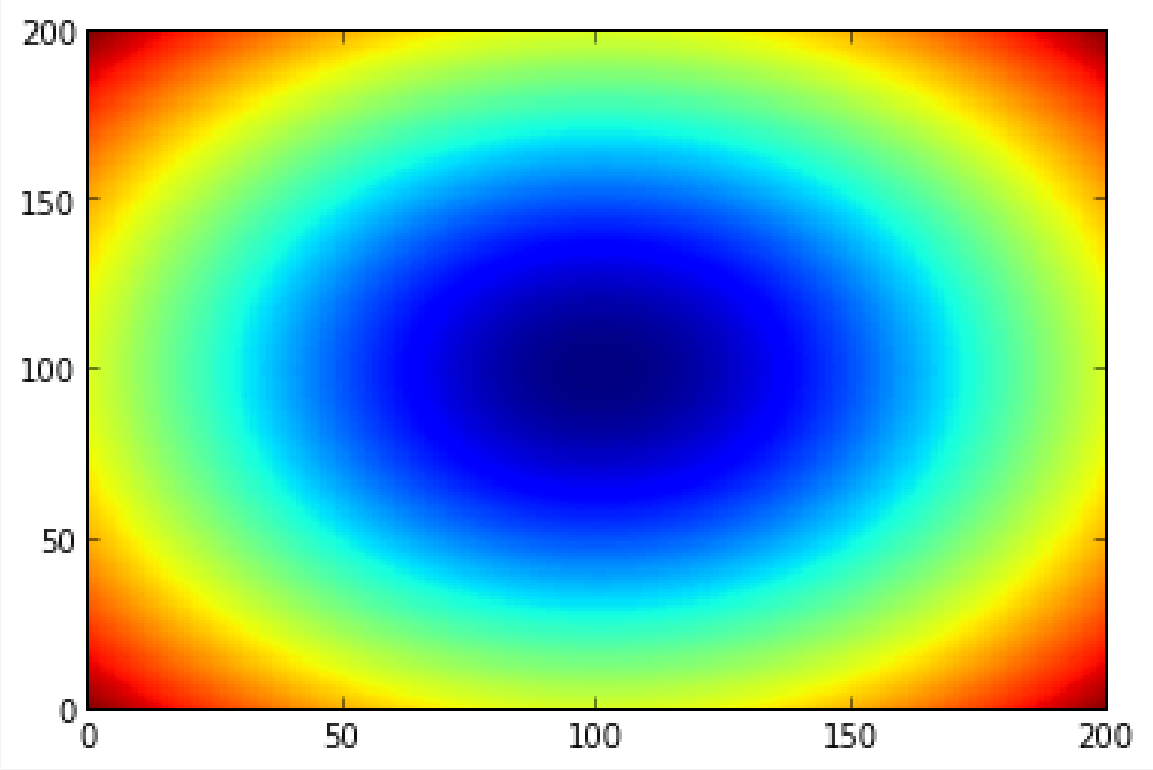


Figure 2: Numpy Broadcasting

```
i = arange(-100, 100).reshape(200, 1, 1)
j = reshape(i, (1, 200, 1))
k = reshape(i, (1, 1, 200))
R = np.sqrt(i**2 + j**2 + k**2)
```

```
_ = pcolor(R[-50]/amax(R))
```



```
i, j, k = ogrid[-100:100, -100:100, -100:100]
R = np.sqrt(i**2 + j**2 + k**2)
```

1.7 NumPy's View of Data

```
print array([1,2,3], dtype=float)
print arange(10).astype(float)
print np.int8(arange(10, dtype=float))
print np.dtype(int)
print np.issubdtype(np.int32, np.int)
print np.issubdtype(np.int, np.float)
```

```
[ 1.  2.  3.]
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
[0 1 2 3 4 5 6 7 8 9]
int64
True
False
```

1.8 Arrays of Structs in NumPy

```
x = np.zeros((2,),dtype=('i4,f4,a10'))
print x
```

```
[(0, 0.0, '') (0, 0.0, '')]
```

```
x[:] = [(1,2.,'Hello'),(2,3.,"World")]
print x
```

```
[(1, 2.0, 'Hello') (2, 3.0, 'World')]
```

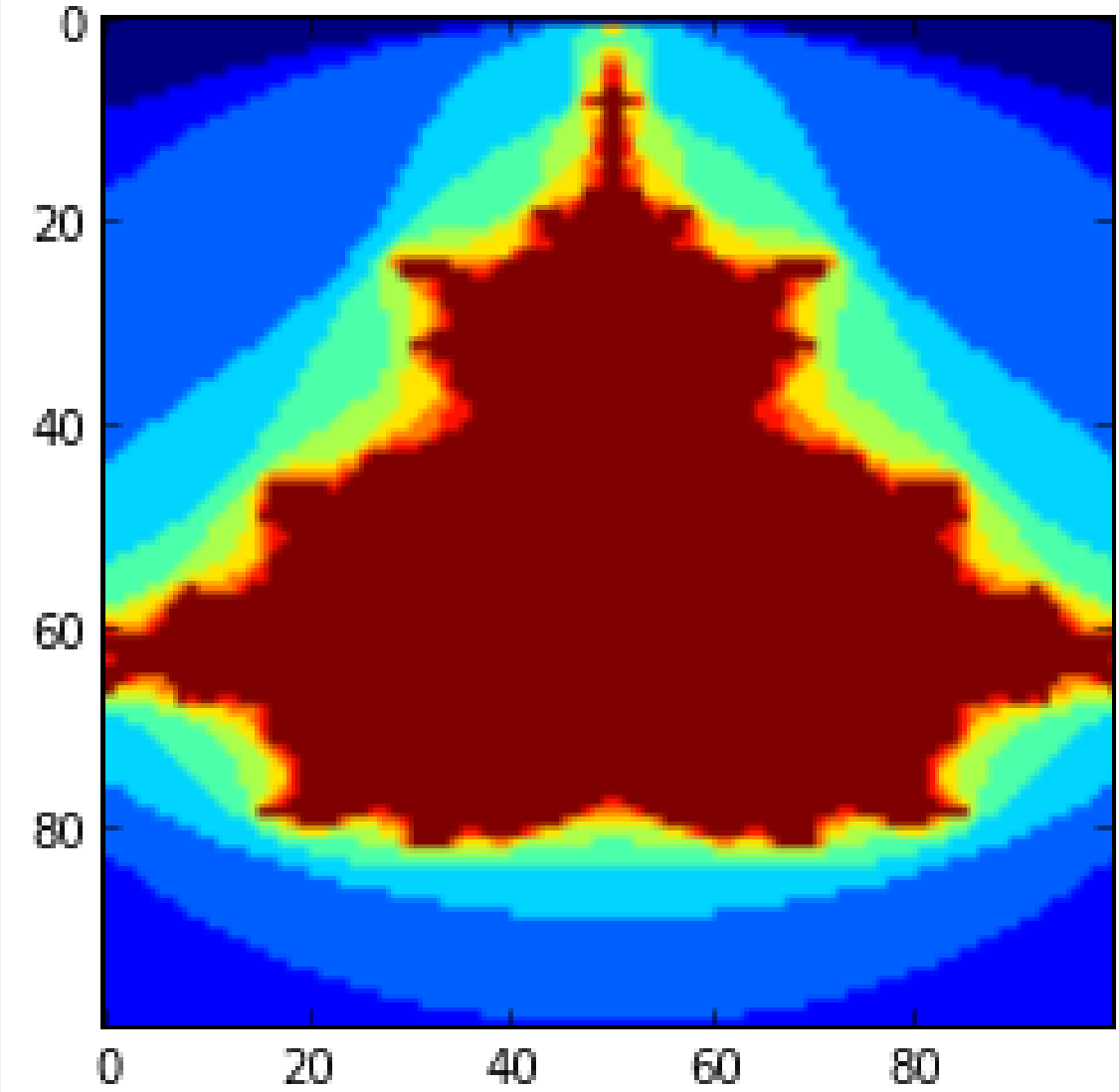
```
dt = dtype([('time', uint64),
            ('pos', [
               ('x', float),
               ('y', float)])])
x = np.array([
        (100, ( 0, 0.5)),
        (200, ( 0, 10.3)),
        (300, (5.5, 15.1))],
        dtype=dt)
print x['time']
print x[ x['time'] >= 200 ]
```

```
[100 200 300]
[(200L, (0.0, 10.3)) (300L, (5.5, 15.1))]
```

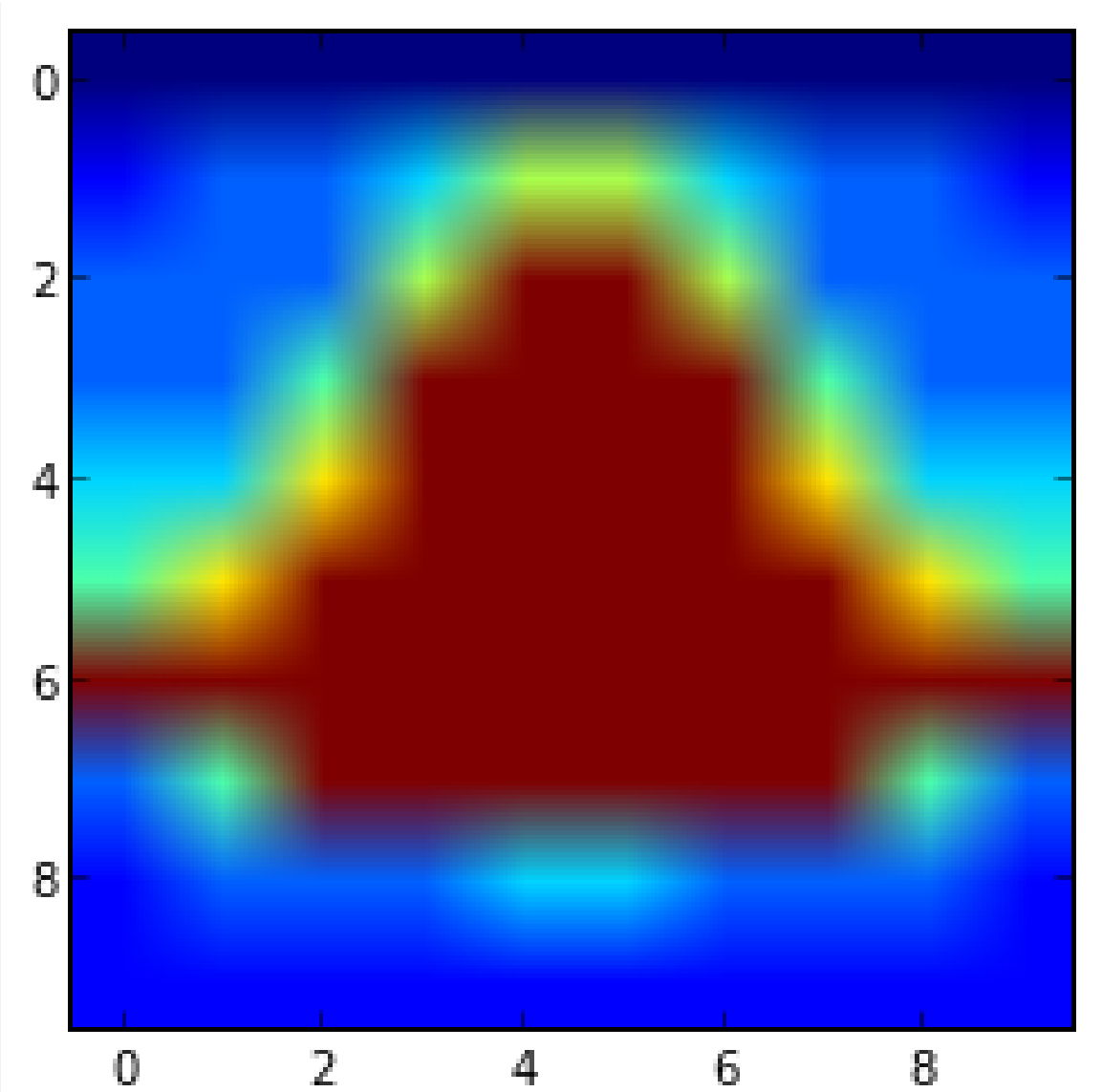
2 Profiling Python

Profiling is an important way to find the bottlenecks in a program. Python has several different profilers, here we show the most common one that is part of the standard library.

```
def mandelbrot (x, y, maxit):
    c = x + y*1.j
    z = 0. + 0.j
    it = 0
    while abs(z) < 2 and it < maxit:
        z = z**2 + c
        it += 1
    return float(it)/maxit

def draw_mandelbrot(num_x, num_y):
    results = []
    for x in linspace(-2.0, 1.0, num_x):
        results.append([])
        for y in linspace(-1.0, 1.0, num_y):
            results[-1].append(mandelbrot(x, y, 10))
    imshow(results)

draw_mandelbrot(100,100)
```




```
import profile
profile.run("draw_mandelbrot(10, 10)", "dm.stats")
```




```
import pstats
p = pstats.Stats('dm.stats')
p.strip_dirs().sort_stats(-1).print_stats(10)
```

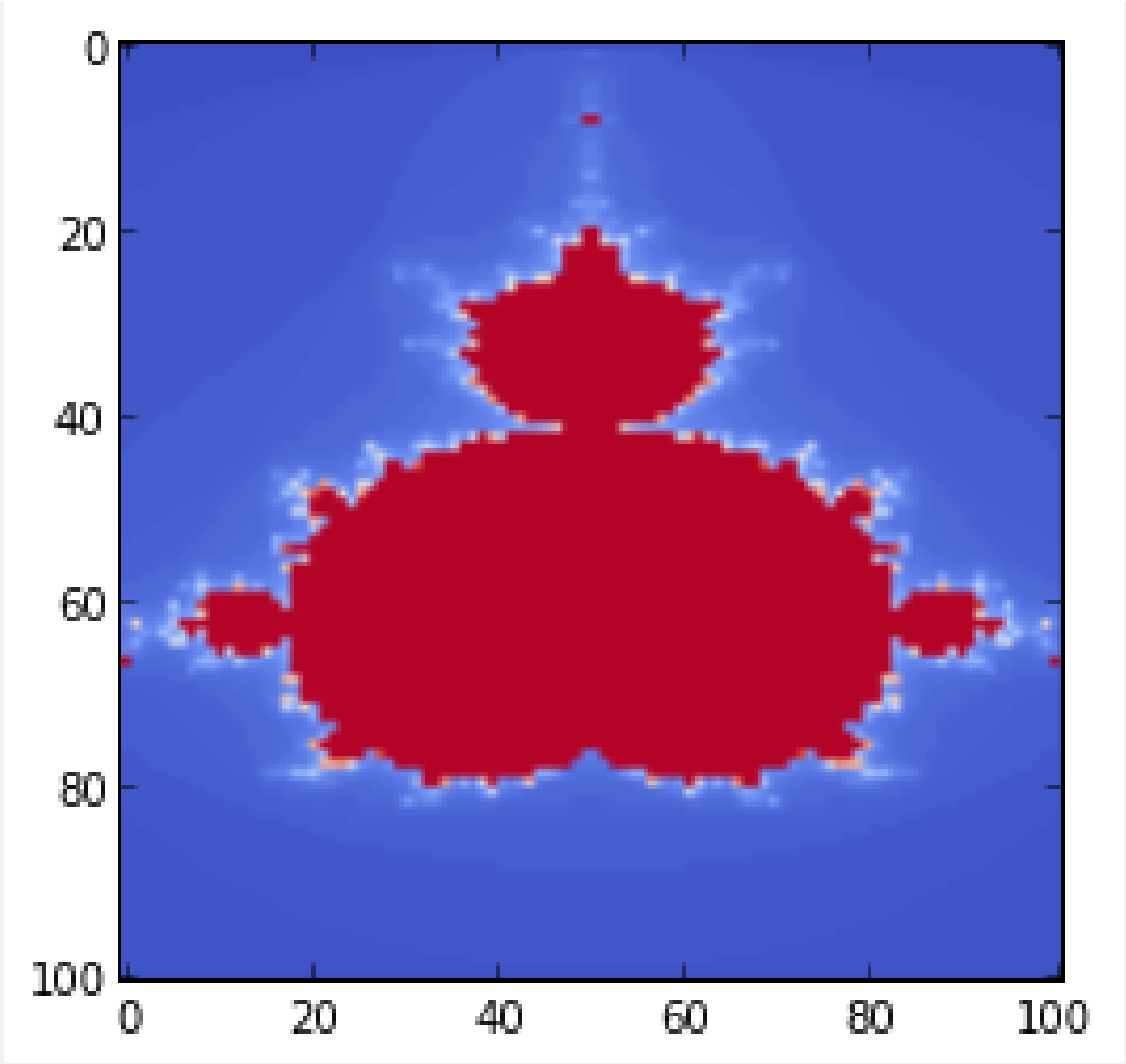
```
Sat Oct 13 22:47:21 2012    dm.stats

         34798 function calls (34263 primitive calls) in 0.352 seconds

   Ordered by: standard name
   List reduced from 418 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      654    0.003    0.000    0.003    0.000 :0(abs)
```

```
       2    0.000    0.000    0.000    0.000 :0(accumulate)
     585    0.008    0.000    0.008    0.000 :0(all)
       1    0.000    0.000    0.000    0.000 :0(any)
     249    0.001    0.000    0.001    0.000 :0(append)
      13    0.000    0.000    0.000    0.000 :0(arange)
    1298    0.018    0.000    0.018    0.000 :0(array)
      80    0.000    0.000    0.000    0.000 :0(callable)
       1    0.000    0.000    0.000    0.000 :0(can_cast)
      40    0.001    0.000    0.001    0.000 :0(concatenate)

<pstats.Stats instance at 0x1114f05f0>
```

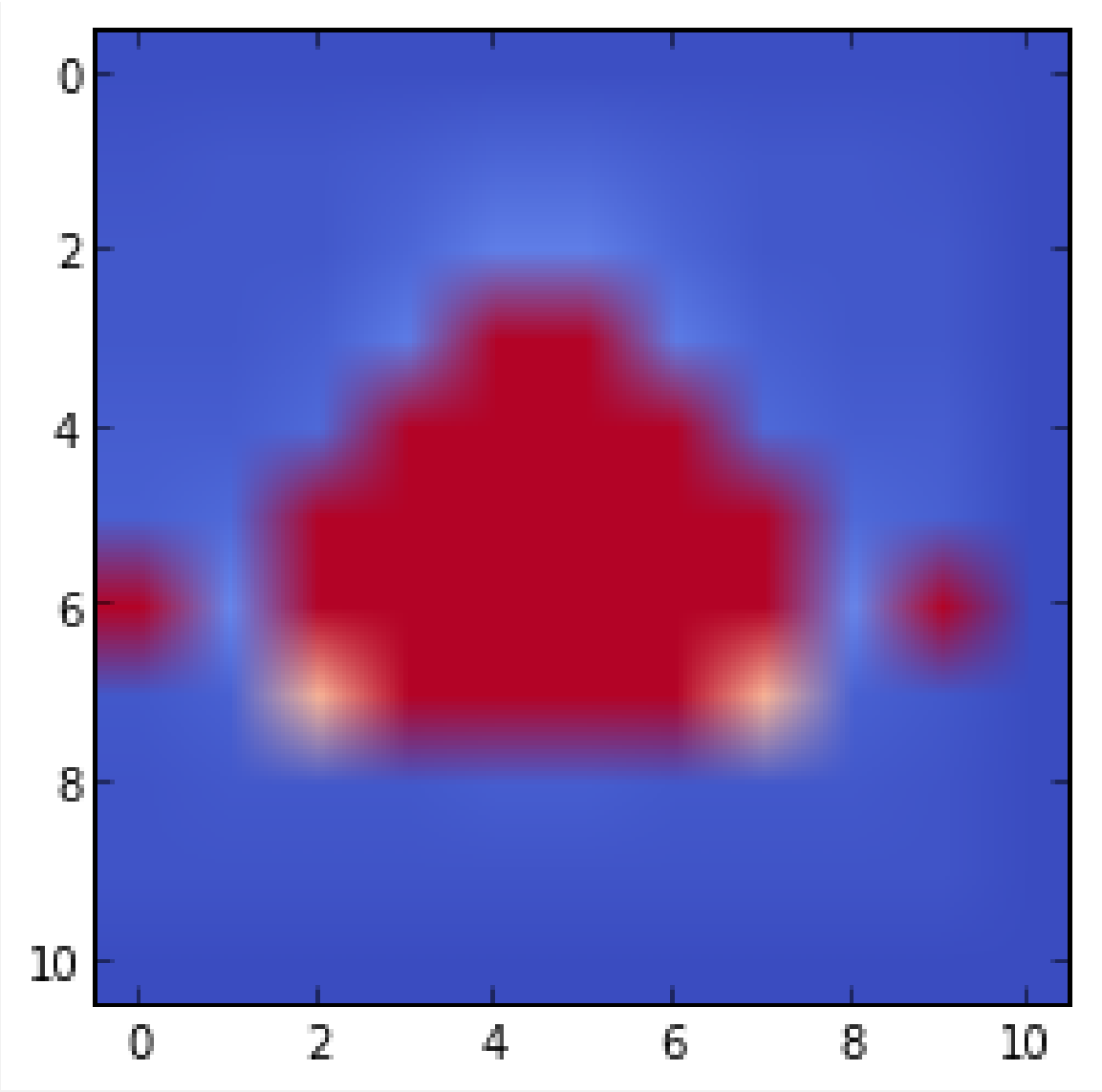
```
p.strip_dirs().sort_stats('time').print_stats(10)
```

```
Sat Oct 13 22:47:21 2012    dm.stats

         34798 function calls (34263 primitive calls) in 0.352 seconds

   Ordered by: internal time
   List reduced from 418 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      585    0.024    0.000    0.055    0.000 path.py:86(__init__)
     1298    0.018    0.000    0.018    0.000 :0(array)
      120    0.018    0.000    0.167    0.001 lines.py:128(__init__)
      540    0.013    0.000    0.117    0.000 markers.py:115(_recache)
     2544    0.012    0.000    0.012    0.000 __init__.py:662(__getitem__)
     1145    0.011    0.000    0.018    0.000 transforms.py:80(__init__)
     2101    0.009    0.000    0.009    0.000 :0(isinstance)
      100    0.009    0.000    0.011    0.000 <ipython-input-15-036d85cb27e4>:1(mandelbrot)
      933    0.008    0.000    0.022    0.000 transforms.py:1321(__init__)
      983    0.008    0.000    0.016    0.000 numeric.py:167(asarray)

<pstats.Stats instance at 0x1114f05f0>
```

```
def draw_mandelbrot2(num_x, num_y):
    results = np.zeros((num_x+1, num_y+1), dtype=float)
    for i,x in enumerate(linspace(-2.0, 1.0, num_x)):
        for j,y in enumerate(linspace(-1.0, 1.0, num_y)):
            results[i, j] = mandelbrot(x, y, 100)
    imshow(results, cmap=cm.coolwarm)
draw_mandelbrot2(100,100)
```




```
profile.run("draw_mandelbrot2(10, 10)", "dm2.stats")
```




```
p = pstats.Stats('dm2.stats')
p.strip_dirs().sort_stats(-1).print_stats(10)
```

```
Sat Oct 13 22:47:24 2012    dm2.stats

         36976 function calls (36441 primitive calls) in 0.363 seconds

   Ordered by: standard name
   List reduced from 418 to 10 due to restriction <10>

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     2944    0.013    0.000    0.013    0.000 :0(abs)
        2    0.000    0.000    0.000    0.000 :0(accumulate)
      585    0.007    0.000    0.007    0.000 :0(all)
```

```
        1    0.000    0.000    0.000    0.000 :0(any)
      139    0.000    0.000    0.000    0.000 :0(append)
       13    0.000    0.000    0.000    0.000 :0(arange)
     1298    0.016    0.000    0.016    0.000 :0(array)
       80    0.000    0.000    0.000    0.000 :0(callable)
        1    0.000    0.000    0.000    0.000 :0(can_cast)
       40    0.001    0.000    0.001    0.000 :0(concatenate)

<pstats.Stats instance at 0x1293fb5a8>
```

3 Compiling to C

C is faster, and Python is easier to write. We want both!

3.1 Cython

- a programming language based on Python
- uses extra syntax allowing for optional static type declarations
- source code gets translated into optimized C/C++ code and compiled as Python extension modules

3.2 Using Cython in IPython

In IPython we can make any cell call out to Cython via the cell magic
First load the extension

```
%load_ext cythonmagic
```

Now use `%%cython` at the begining of a code cell to call out to Cython.

```
%%cython
def f_cython(int i):
    return i**4 + 3*i**2 + 10
```

Now use Cython function in code:

```
f_cython(100)
```

```
100030010
```

3.3 How much faster is Cython?

The more you are able to provide type information the better the compile. For example f without type information:

```
def f_python(i):
    return i**4 + 3*i**2 + 10
```

```
%timeit f_python(100)
```

```
1000000 loops, best of 3: 409 ns per loop
```

```
%timeit f_cython(100)
```

```
10000000 loops, best of 3: 114 ns per loop
```

3.4 Declaring Cython variables for C level

If you use a variable or function only at the Cython level you can keep it in C via `cdef`:

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

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

```
%timeit integrate_f(1.0, 2.0, 1000)
```

```
10000 loops, best of 3: 36.6 us per loop
```

The pure Python version:

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

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

```
%timeit integrate_f(1.0, 2.0, 1000)
```

```
1000 loops, best of 3: 498 us per loop
```

3.5 Using NumPy with Cython

You can also use fast accessors to NumPy arrays from Cython:

```
%%cython
import numpy as np
# "cimport" is used to import special compile-time information
# about the numpy module (this is stored in a file numpy.pxd which is
# currently part of the Cython distribution).
cimport numpy as np
# We now need to fix a datatype for our arrays. I've used the variable
# DTYPE for this, which is assigned to the usual NumPy runtime
# type info object.
DTYPE = np.int
# "ctypedef" assigns a corresponding compile-time type to DTYPE_t. For
# every type in the numpy module there's a corresponding compile-time
# type with a _t-suffix.
ctypedef np.int_t DTYPE_t
# "def" can type its arguments but not have a return type. The type of the
# arguments for a "def" function is checked at run-time when entering the
# function.
#
# The arrays f, g and h is typed as "np.ndarray" instances. The only effect
# this has is to a) insert checks that the function arguments really are
# NumPy arrays, and b) make some attribute access like f.shape[0] much
# more efficient. (In this example this doesn't matter though.)
def naive_convolve(np.ndarray f, np.ndarray g):
    if g.shape[0] % 2 != 1 or g.shape[1] % 2 != 1:
        raise ValueError("Only odd dimensions on filter supported")
    assert f.dtype == DTYPE and g.dtype == DTYPE
    # The "cdef" keyword is also used within functions to type variables. It
    # can only be used at the top indendation level (there are non-trivial
    # problems with allowing them in other places, though we'd love to see
    # good and thought out proposals for it).
    #
    # For the indices, the "int" type is used. This corresponds to a C int,
    # other C types (like "unsigned int") could have been used instead.
    # Purists could use "Py_ssize_t" which is the proper Python type for
    # array indices.
    cdef int vmax = f.shape[0]
    cdef int wmax = f.shape[1]
    cdef int smax = g.shape[0]
    cdef int tmax = g.shape[1]
    cdef int smid = smax // 2
    cdef int tmid = tmax // 2
    cdef int xmax = vmax + 2*smid
    cdef int ymax = wmax + 2*tmid
    cdef np.ndarray h = np.zeros([xmax, ymax], dtype=DTYPE)
    cdef int x, y, s, t, v, w
    # It is very important to type ALL your variables. You do not get any
    # warnings if not, only much slower code (they are implicitly typed as
    # Python objects).
```

```
    cdef int s_from, s_to, t_from, t_to
    # For the value variable, we want to use the same data type as is
    # stored in the array, so we use "DTYPE_t" as defined above.
    # NB! An important side-effect of this is that if "value" overflows its
    # datatype size, it will simply wrap around like in C, rather than raise
    # an error like in Python.
    cdef DTYPE_t value
    for x in range(xmax):
        for y in range(ymax):
            s_from = max(smid - x, -smid)
            s_to = min((xmax - x) - smid, smid + 1)
            t_from = max(tmid - y, -tmid)
            t_to = min((ymax - y) - tmid, tmid + 1)
            value = 0
            for s in range(s_from, s_to):
                for t in range(t_from, t_to):
                    v = x - smid + s
                    w = y - tmid + t
                    value += g[smid - s, tmid - t] * f[v, w]
            h[x, y] = value
    return h
```

```
N=100
f = np.arange(N*N, dtype=np.int).reshape((N,N))
g = np.arange(81, dtype=np.int).reshape((9, 9))
%timeit -n2 -r3 naive_convolve(f, g)
```

```
2 loops, best of 3: 1.62 s per loop
```