Python in HPC

Supercomputing 2012

Presenters:

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

Travis Oliphant, PhD
Continuum Analytics

Aron Ahmadia, PhD
Supercomputing Laboratory
King Abdullah University of Science and Technoglogy

Updated Tutorial

These presentation materials are being continuously updated as we refine and improve our demonstrats. To get the latest version of this tutorial you can:

1) Download a zip or tar ball from the github SC2012 tag:

```
wget -
-no-check-certificate https://github.com/aterrel/HPCPythonSC2012/zipball/SC2012
wget -
-no-check-certificate https://github.com/aterrel/HPCPythonSC2012/tarball/SC2012
```

2) Checkout from git

```
git clone https://github.com/aterrel/HPCPythonSC2012.git
```

3) View the html version on nbviewer:
http://nbviewer.ipython.org/urls/raw.github.com/aterrel/HPCPythonSC2012/master/02_Speeding_Python.ipynb

4) As a last resort, head to https://github.com/aterrel/HPCPythonSC2012 for updated instructions (see the README at the bottom of the page).

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 Free Enthought Python Distribution.

How Slow is Python

Let's add one to a million number

In [1]:
```
lst = range(1000000) # A pure Python list
%timeit [i + 1 for i in lst] # A Python list comprehension (iteration happe
```

```
1 loops, best of 3: 208 ms per loop
```

Why is Python Slow?

Dynamic typing requires lots of metadata around variable.

- Python uses heavy frame objects during iteration

Solution:

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

In [2]:
```
arr = arange(1000000) # A NumPy list of integers
%timeit arr + 1 # Use operator overloading for nice syntax, now iteration i
```

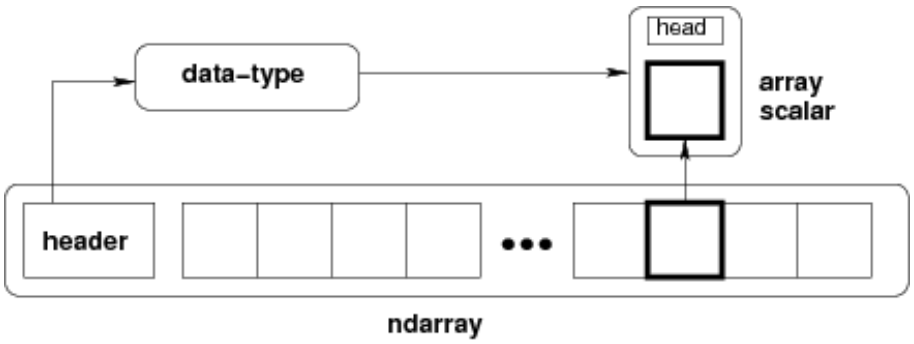
```
100 loops, best of 3: 6.64 ms per loop
```

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

NumPy Data layout

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



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

```
In [3]: %timeit [sin(i)**2 for i in arr]
```

```
1 loops, best of 3: 11.6 s per loop
```

```
In [6]: %timeit np.sin(arr)**2
```

```
10 loops, best of 3: 44.8 ms per loop
```

Other NumPy features to be aware of

- Reshaping

In [10]:
```
arr2 = arr.reshape((10,100000))
print(arr2)
```

```
[[     0      1      2 ...,  99997  99998  99999]
 [100000 100001 100002 ..., 199997 199998 199999]
 [200000 200001 200002 ..., 299997 299998 299999]
 ...,
 [700000 700001 700002 ..., 799997 799998 799999]
 [800000 800001 800002 ..., 899997 899998 899999]
 [900000 900001 900002 ..., 999997 999998 999999]]
```

- Memory Views

In [13]:
```
arr2.view?
```

- Index Slicing

In [14]:
```
x = np.arange(0, 20, 2); y = x**2
((y[1:] - y[:-1]) / (x[1:] - x[:-1])) # dy/dx
```

Out[14]:
```
array([ 2,  6, 10, 14, 18, 22, 26, 30, 34])
```

In [15]:
```
((y[2:] - y[:-2])/(x[2:] - x[:-2])) # d^2y/dx^2 via center differencing
```

Out[15]:
```
array([ 4,  8, 12, 16, 20, 24, 28, 32])
```

- Fancy Indexing

In [18]:
```
evens = arr[arr%2 == 0]
print(evens)
```

```
[     0      2      4 ..., 999994 999996 999998]
```

Compiling to C

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

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

Using Cython in IPython

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

First load the extension

In [19]:
```
%load_ext cythonmagic
```

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

In [27]:
```
%%cython
def f_cython(int i):
    return i**4 + 3*i**2 + 10
```

Now use Cython function in code:

In [28]:
```
f(100)
```

Out[28]:
```
100030010
```

How much faster is Cython?

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

In [24]:
```
%%cython
def f_slow(i):
    return i**4 + 3*i**2 + 10
```

In [25]:
```
%timeit f_slow(100)
```

```
1000000 loops, best of 3: 248 ns per loop
```

In [29]:
```
%timeit f_cython(100)
```

```
10000000 loops, best of 3: 117 ns per loop
```

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`:

In [30]:
```
%%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
```

In [32]:
```
%timeit integrate_f(1.0, 2.0, 1000)
```

```
10000 loops, best of 3: 37.4 us per loop
```

The pure Python version:

In [33]:
```
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
```

In [34]:
```
%timeit integrate_f(1.0, 2.0, 1000)
```

```
1000 loops, best of 3: 530 us per loop
```

Using NumPy with Cython

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

In [35]:
```
%%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
```

In [36]:
```
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: 1.52 s per loop
```