

An introduction to scientific programming with



**Session 5:**  
Extreme Python

# Efficient data crunching

## **PyTables**

- For creating, storing and analysing datasets
  - from simple, small tables to complex, huge datasets
  - standard HDF5 file format
  - incredibly fast – even faster with indexing

## **Numexpr**

- For highly efficient array operations

PyTables uses Numexpr, and various other speedup tools, seamlessly

See: <http://www.pytables.org/docs/LargeDataAnalysis.pdf>

# Using PyTables

See

<http://showmedo.com/videotutorials/video?name=1780000&fromSeriesID=178>

and

<http://showmedo.com/videotutorials/video?name=1780010&fromSeriesID=178>

# Testing performance

**timeit** – use in interpreter, script or command line

```
python -m timeit [-n N] [-r N] [-s S] [statement ...]
```

Options:

-s S, --setup=S

statement to be executed once initially (default pass)

-n N, --number=N

how many times to execute 'statement' (default: take ~0.2 sec total)

-r N, --repeat=N

how many times to repeat the timer (default 3)

# Testing performance

*# fastest way to calculate x\*\*5?*

```
$ python -m timeit -s 'from math import pow; x = 1.23' 'x*x*x*x*x'  
10000000 loops, best of 3: 0.161 usec per loop
```

```
$ python -m timeit -s 'from math import pow; x = 1.23' 'x**5'  
10000000 loops, best of 3: 0.111 usec per loop
```

```
$ python -m timeit -s 'from math import pow; x = 1.23' 'pow(x, 5)'  
1000000 loops, best of 3: 0.184 usec per loop
```

*# fastest minimisation routine?*

```
$ python -m timeit -s 'from math import cos; from scipy.optimize  
import fmin as fmin' 'fmin(cos, 3.0, disp=False)'  
1000 loops, best of 3: 772 usec per loop
```

```
$ python -m timeit -s 'from math import cos; from scipy.optimize  
import fmin_powell as fmin' 'fmin(cos, 2.5, disp=False)'  
1000 loops, best of 3: 873 usec per loop
```

```
$ python -m timeit -s 'from math import cos; from scipy.optimize  
import fmin_cg as fmin' 'fmin(cos, 3.0, disp=False)'  
1000 loops, best of 3: 221 usec per loop
```

# Mixing Python and C – fast and flexible

**Cython** is used for compiling Python-like code to machine-code

- supports a big subset of the Python language
  - conditions and loops run 2-8x faster, overall 30% faster for plain Python code
  - add types for speedups (hundreds of times)
  - easily use native libraries (C/C++/Fortran) directly
- 
- Cython code is turned into C code
    - uses the CPython API and runtime
- 
- Coding in Cython is like coding in Python and C at the same time!

# Cython

*Use cases:*

- Performance-critical code
  - which does not translate to array-based approach (numpy / pytables)
  - existing Python code → optimise critical parts
- Wrapping existing C/C++ libraries
  - particularly higher-level Pythonised wrapper
  - for one-to-one wrapping other tools might be better suited

# Cython

Cython code must be compiled.

Two stages:

- A `.pyx` file is compiled by Cython to a `.c` file, containing the code of a Python extension module
- The `.c` file is compiled by a C compiler
  - Generated C code can be built without Cython installed
  - Cython is a developer dependency, not a build-time dependency
  - Generated C code works with Python 2.3 through Python 3.1
  - The result is a `.so` file (or `.pyd` on Windows) which can be imported directly into a Python session

# Cython

Ways of building Cython code:

- Run cython command-line utility and compile the resulting C file
  - use favourite build tool
  - for cross-system operation you need to query Python for the C build options to use
- Use pyximport to importing Cython .pyx files as if they were .py files; building on the fly (recommended to start).
  - things get complicated if you must link to native libraries
  - larger projects tend to need a build phase anyway
- Write a distutils setup.py
  - standard way of distributing, building and installing Python modules

# Cython

- Cython supports most of normal Python
- Most standard Python code can be used directly with Cython
  - typical speedups of (very roughly) a factor of two
  - should not ever slow down code – safe to try
  - name file .pyx or use `pyimport = True`

```
>>> import pyximport
>>> pyximport.install()
>>> import mpyxmodule # converts and compiles on the fly

>>> pyximport.install(pyimport = True) # experimental
>>> import mpymodule # converts and compiles on the fly
                        # should fall back to Python if fails
```

# Cython

- Big speedup from defining types of key variables
- Use native C-types (int, double, char \*, etc.)
- Use Python C-types (Py\_int\_t, Py\_float\_t, etc.)
- Use `cdef` to declare variable types
- Also use `cdef` to declare C-only functions (with return type)
  - can also use `cpdef` to declare functions which are automatically treated as C or Python depending on usage
- Don't forget function arguments (but note `cdef` not used here)

# Cython – primes example

- Efficient algorithm to find first  $N$  prime numbers

```
def primes(kmax):  
    p = []  
    k = 0  
    n = 2  
    while k < kmax:  
        i = 0  
        while i < k and n % p[i] != 0:  
            i = i + 1  
        if i == k:  
            k = k + 1  
            p.append(n)  
        n = n + 1  
    return p
```

**primes.py**

```
$ python -m timeit -s 'import primes as p' 'p.primes(100)'  
1000 loops, best of 3: 1.35 msec per loop
```

# Cython – primes example

**cprimes.pyx**

```
def primes(kmax):
    p = []
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0:
            i = i + 1
        if i == k:
            k = k + 1
            p.append(n)
        n = n + 1
    return p
```

```
$ python -m timeit -s 'import pyximport; pyximport.install
    (); import cprimes as p' 'p.primes(100)'
```

1000 loops, best of 3: 731 usec per loop

***1.8x speedup***

# Cython – primes example

**xprimes.pyx**

```
def primes(int kmax):      # declare types of parameters
    cdef int n, k, i      # declare types of variables
    cdef int p[1000]      # including arrays
    result = []           # can still use normal Python types
    if kmax > 1000:       # in this case need to hardcode limit
        kmax = 1000
    k = 0
    n = 2
    while k < kmax:
        i = 0
        while i < k and n % p[i] != 0: } contains only C-code
            i = i + 1
        if i == k:
            p[k] = n
            k = k + 1
            result.append(n)
        n = n + 1
    return result         # return Python object
```

40.8 usec per loop

**33x speedup**

# Cython – integration example

**integrate.py**

```
def f(x):  
    return 1/(x**3 + 2*x**2)  
  
def integrate_f(a, b, N):  
    s = 0  
    dx = float(b-a)/N  
    for i in range(N):  
        s += f(a+i*dx)  
    return s * dx
```

```
$ python -m timeit -s 'import integrate as integrate'  
    'integrate.integrate_f(1000000, 1, 100)'
```

1000 loops, best of 3: 75.5 usec per loop

# Cython – integration example

**cintegrate.pyx**

```
def f(x):  
    return 1/(x**3 + 2*x**2)  
  
def integrate_f(a, b, N):  
    s = 0  
    dx = float(b-a)/N  
    for i in range(N):  
        s += f(a+i*dx)  
    return s * dx
```

```
$ python -m timeit -s 'import pyximport; pyximport.install  
    (); import cintegrate as integrate' 'integrate.integrate_f  
    (1000000, 1, 100)'
```

10000 loops, best of 3: 44.2 usec per loop

***1.7x speedup***

# Cython – integration example

## xintegrate.pyx

```
cdef double f(double x):      # define return type of function
    return 1/(x**3 + 2*x**2)

def integrate_f(double a, double b, int N):
    cdef double s = 0          # define types of function
    cdef double dx = (b-a)/N   # arguments and variables
    cdef int i
    for i in range(N):
        s += f(a+i*dx)
    return s * dx
```

*but using scipy would be wiser...*

```
$ python -m timeit -s 'import pyximport; pyximport.install
    (); import xintegrate as integrate' 'integrate.integrate_f
    (1000000, 1, 100)'
```

100000 loops, best of 3: 2.43 usec per loop      **31x speedup**

# Cython and Numpy

- Cython provides a way to quickly access Numpy arrays with specified types and dimensionality
  - for implementing fast specific algorithms
- Also provides way to create generalized *Ufuncs*
- Can be useful, but often using functions provided by numpy, scipy, numexpr or pytables will be easier and faster

# Python web frameworks

## Django

a high-level Python Web framework that encourages rapid development and clean, pragmatic design.

and many others, e.g. Zope (massive), web2py (light), ...

- Give your scientific code a friendly face!
  - easy configuration
  - monitor progress
  - particularly for public code, cloud computing, HPC

An introduction to scientific programming with



**The End**