PYTHON
POUR LE
CALCUL SCIENTIFIQUE

Konrad HINSEN
Centre de Biophysique Moléculaire (Orléans)
et
Synchrotron Soleil (St Aubin)

A short history

1991: Python is published

1994: first scientific applications

1996: Numerical Python

.

.

.

2015: - lots of libraries

- three annual SciPy workshops
- several books available
- taught at many universities
- specialized companies

Some applications

Astronomy

ASTROLIB and PyFITS
(Space Telescope Science Institute)

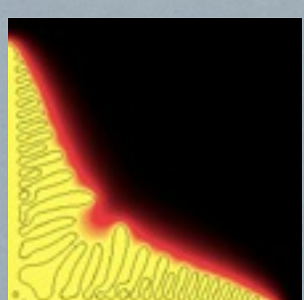
Neurology

Vision Egg
(International collaboration)

Bioinformatics

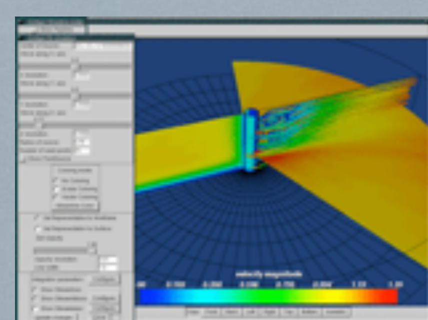
BioPython
(International collaboration)

Finite elements



FiPy
(NIST)

Visualization



MayaVi
(Prabhu Ramachandran)

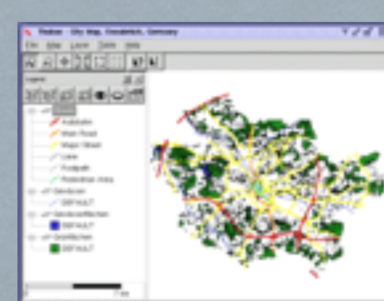
Statistics

Modular toolkit for Data Processing
(Humboldt-Universität)

Dynamical systems

SimPy
(International collaboration)

Geography



Thuban
(Intevation GmbH)

Mathematics

SAGE
(University of Washington)

Python 2 or Python 3?

There are *two* current versions of Python: 2.7 and 3.4

Python 3.x is not just an improvement or extension of Python 2.x It's a new series of versions that is not 100% compatible with the older ones, though very similar.

The two versions have coexisted for many years, and will continue to do so while library authors work on the migration. At this time, Python 2.x is still more useful for scientific applications, because many scientific libraries are not yet available for Python 3.

A presentation of the differences:

http://www.python.org/doc/essays/ppt/euro2008/Py3kEuro08.pdf

Application scenarios

Scripting language

- Read/write files
  - ➥ perl, awk, grep, vi, emacs, ...
- Data analysis and visualization
  - ➥ Matlab/Scilab/Octave, IDL, R
- Job management
  - ➥ sh/bash, csh
- System administration
  - ➥ sh/bash, csh, grep, awk, perl, ...

Advantages of Python:

☺ real programming language

☺ high-quality libraries

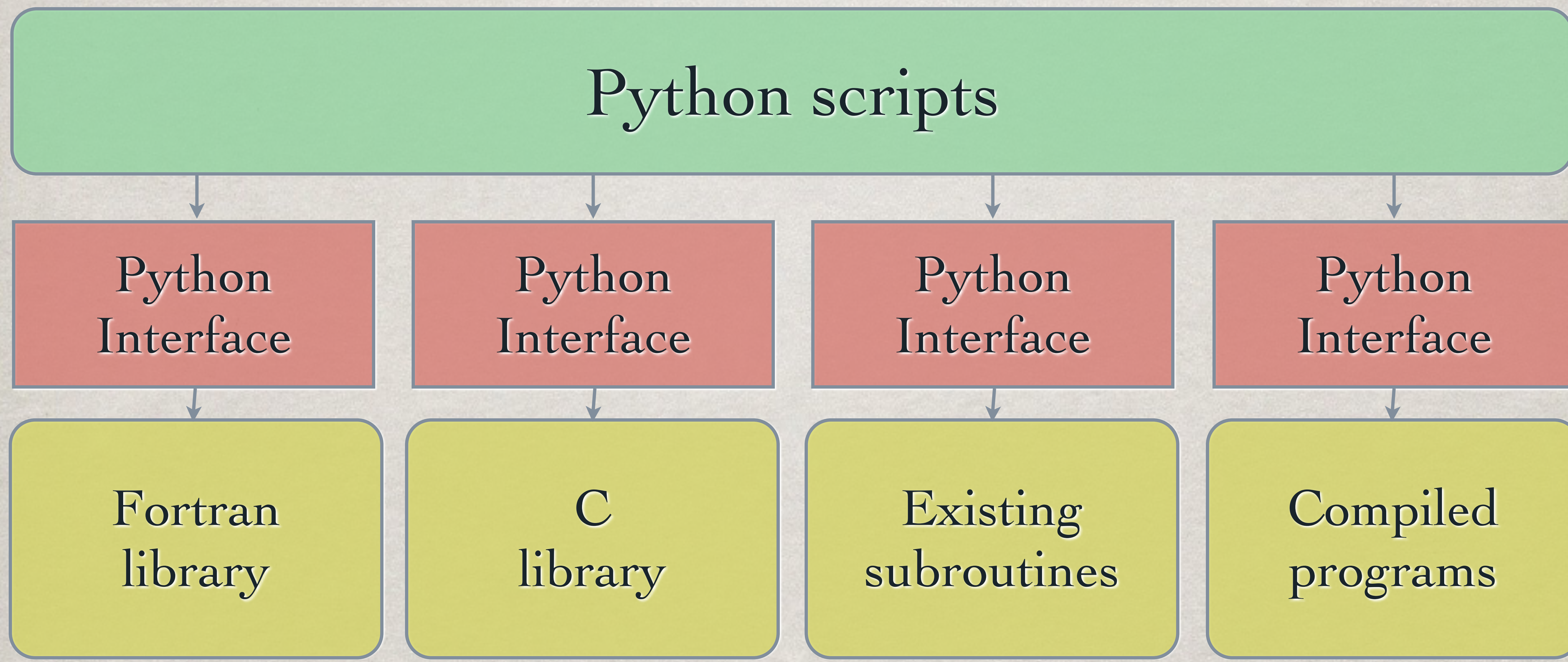
Explorative computing

- Data analysis
- Visualization
- ➥ Simple scripts and interactive work

Useful tools:

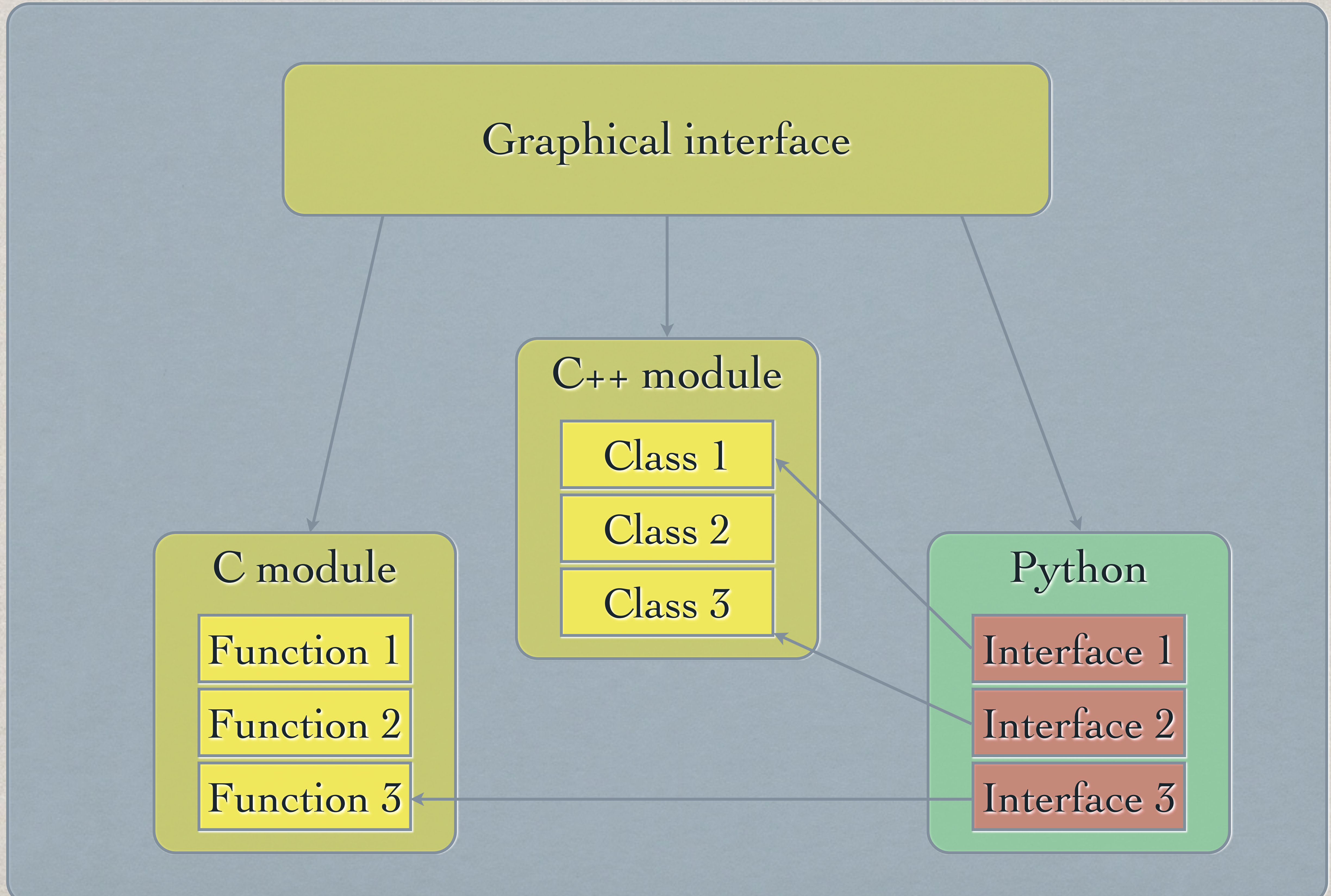
→ IPython and its notebook
→ Emacs + Python mode
→ matplotlib
→ VPython
→ Module pickle

Integration language

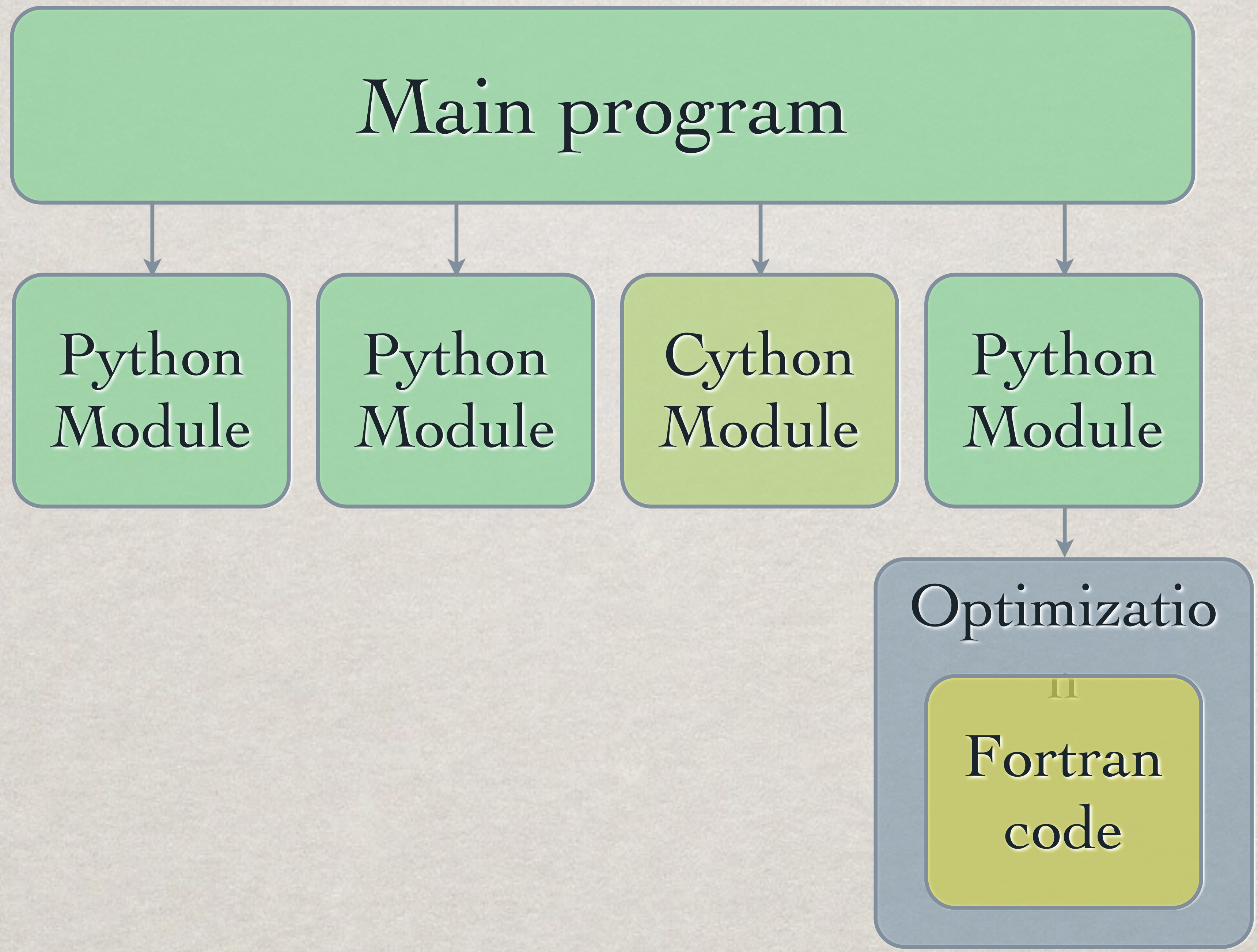


Tools: Cython, f2py, PyFort, swig, ctypes

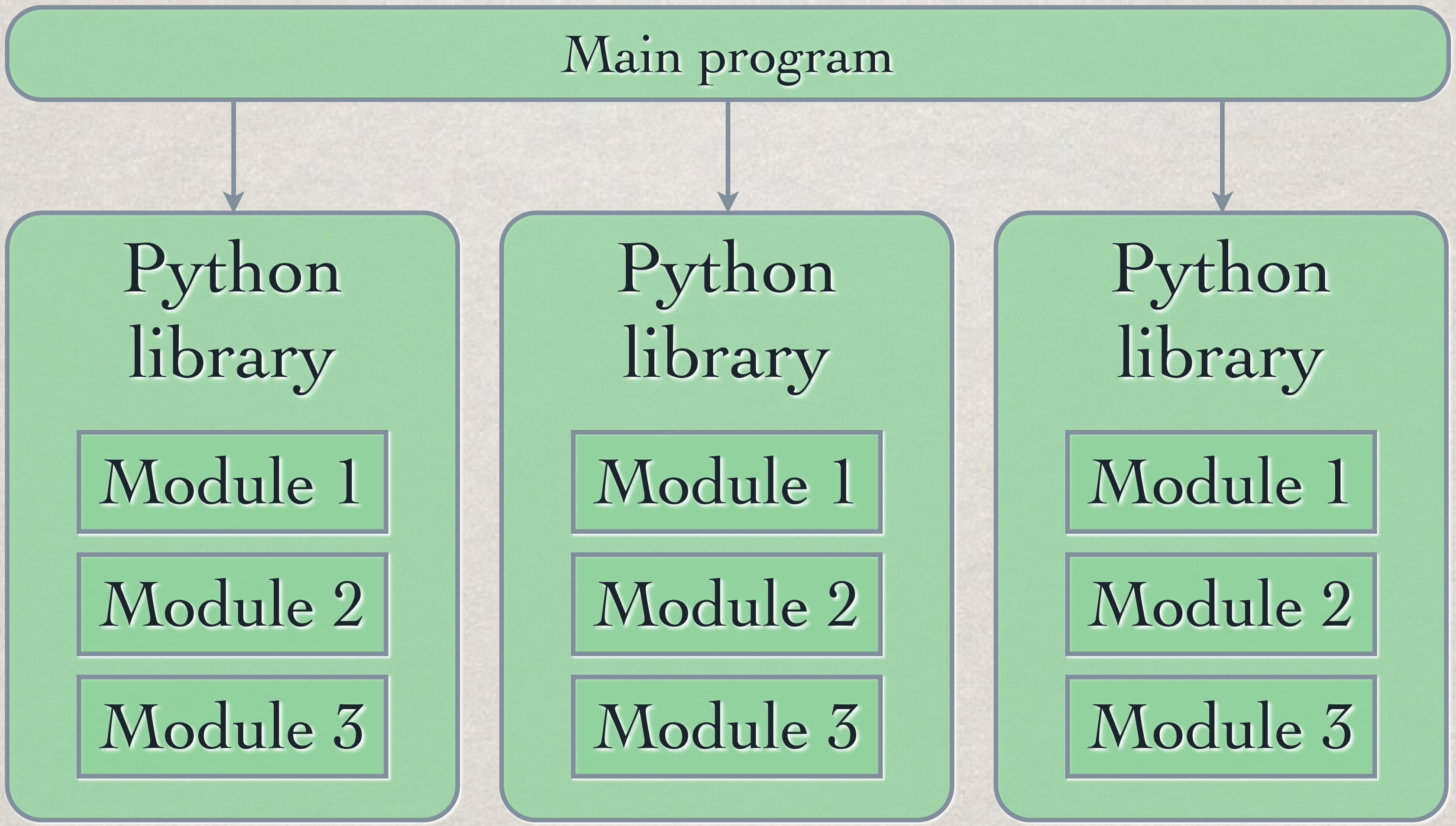
Embedded language



Main language

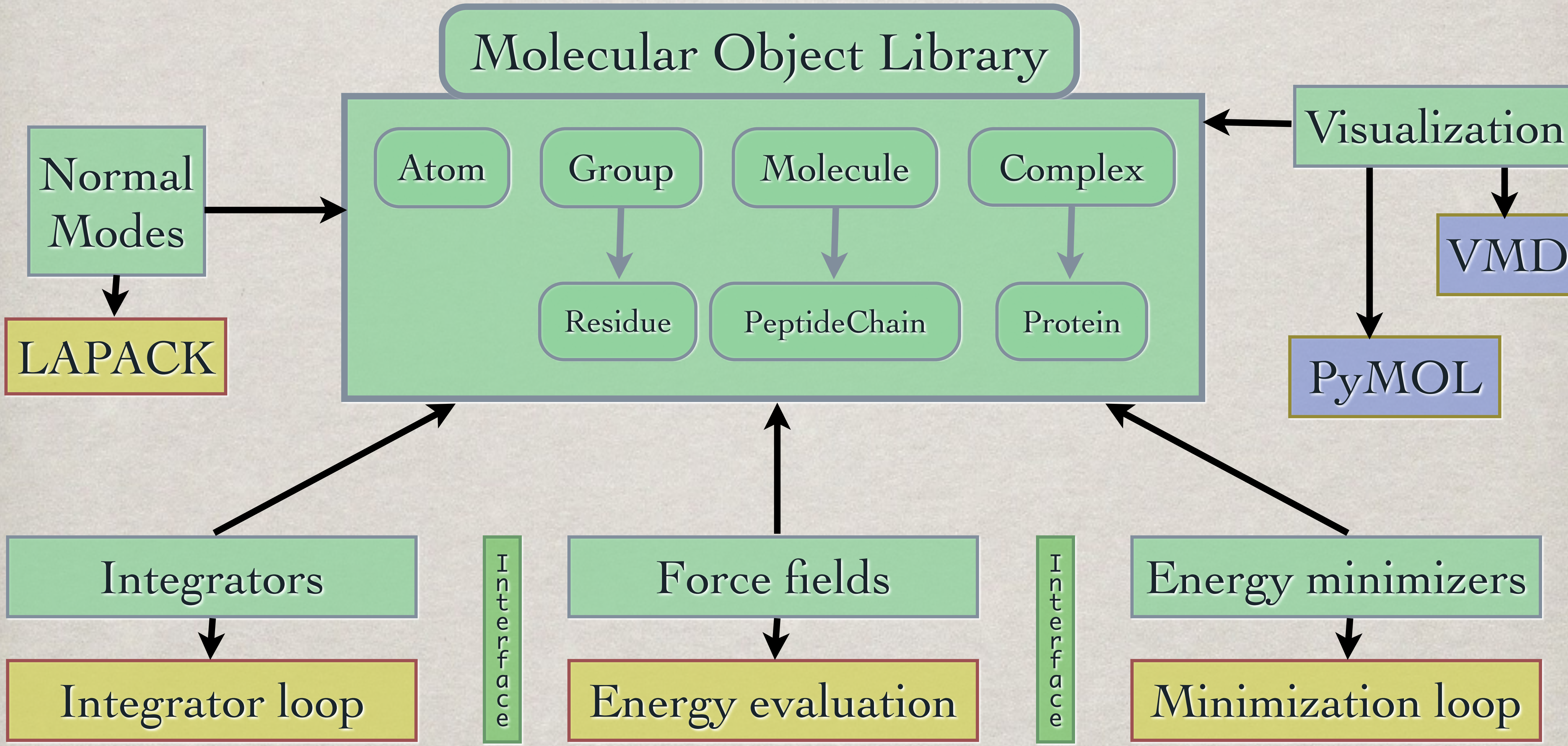


Think Libraries!



A library is more useful than routines hidden in a big program!

Molecular Modelling Toolkit



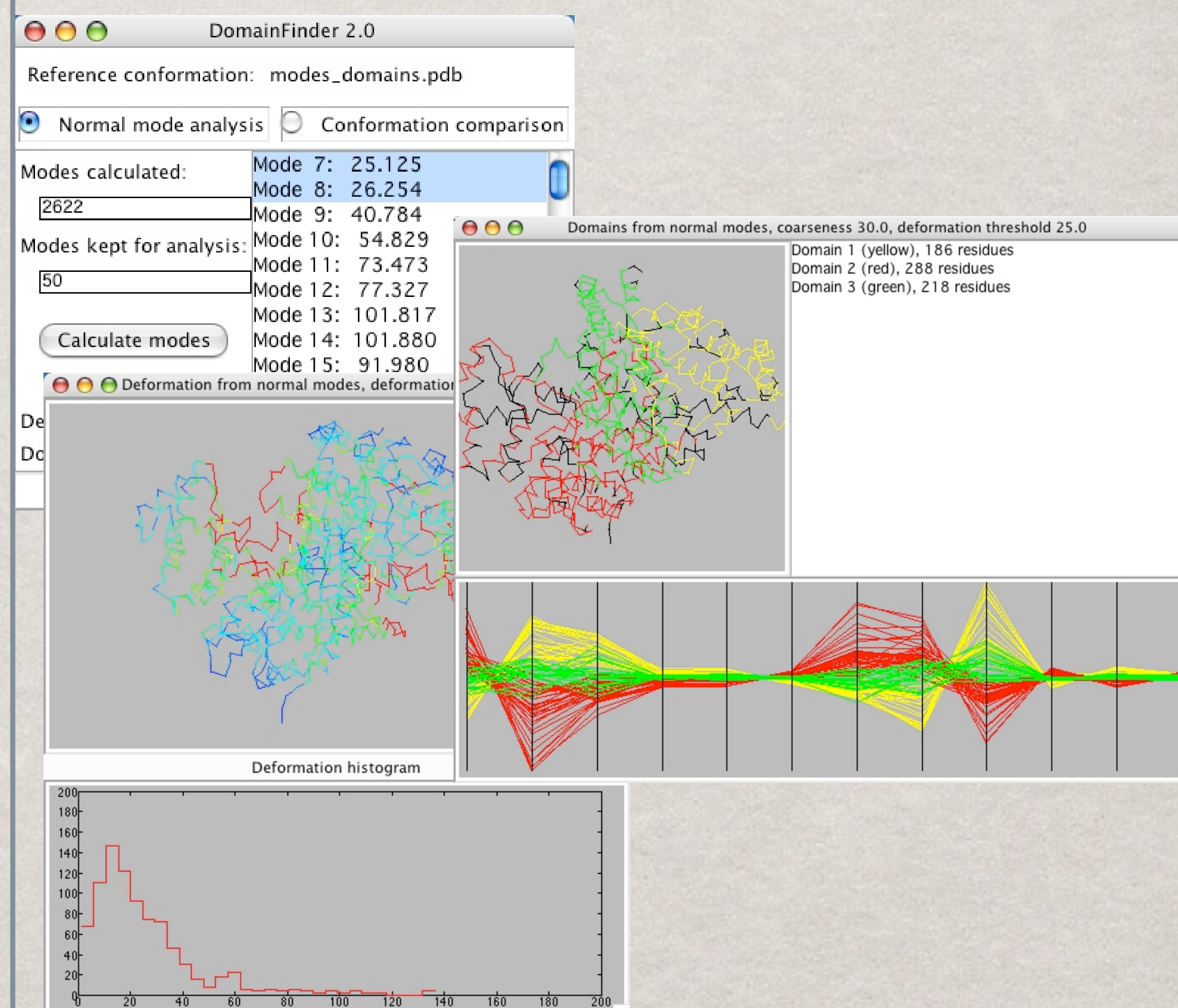
Using MMTK...

Scripts	Graphical interfaces

Scripts

```
# Standard normal mode calculation.
#

from MMTK import *
from MMTK.Proteins import Protein
from MMTK.ForceFields import Amber99ForceField
from MMTK.NormalModes import VibrationalModes
from MMTK.Minimization import ConjugateGradientMinimizer
from MMTK.Trajectory import StandardLogOutput
from MMTK.Visualization import view

# Construct system
universe = InfiniteUniverse(Amber99ForceField())
universe.protein = Protein('bala1')

# Minimize
minimizer = ConjugateGradientMinimizer(universe,
                          actions=[StandardLogOutput(50)])
minimizer(convergence = 1.e-3, steps = 10000)

# Calculate normal modes
modes = VibrationalModes(universe)

# Show animation of the first non-trivial mode
view(modes[6])
```

Graphical interfaces



NumPy

NumPy

Basic functionality for scientific computing:

- Multidimensional arrays
- Arithmetic and mathematical functions on arrays
- Linear algebra (LAPACK)
- Fourier transforms (FFTPACK)
- Random numbers

➥ Efficient implementation that makes handling large data possible using pure Python code.

Documentation: http://numpy.scipy.org/

Arrays

- multidimensional rectangular data container
- all elements have the same type
- compact data layout, compatible with C/Fortran
- efficient operations
- arithmetic
- flexible indexing

Why arrays?

Arrays are the most "natural" data structure for many types of scientific data:

- Matrices
- Time series
- Images
- Functions sampled on a grid
- Tables of data
- ... many more ...

Python lists can handle this, right?

Why arrays?

Python lists are nice, but...

- They are slow to process
- They use a lot of memory
- For tables, matrices, or volumetric data, you need lists of lists of lists... which becomes messy to program.

```
from random import random
from operator import add
import numpy as np
n = 1000000
l1 = [random() for i in range(n)]
l2 = [random() for i in range(n)]
a1 = np.array(l1)
a2 = np.array(l2)

%timeit l3 = map(add, l1, l2)
10 loops, best of 3: 147 ms per loop

%timeit a3 = a1+a2
100 loops, best of 3: 8 ms per loop
```

Bytes per element in a list of floats: 32
Bytes per element in an array of floats: 8

Never forget:

```
import numpy as np
```

(I won't repeat this on every slide!)

Array creation

- np.array([[1, 2], np.array([3, 4])])

```
array([[1, 2],
       [3, 4]])
```

- np.zeros((2, 3), dtype=np.float)

```
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

- np.arange(0, 10, 2)

```
array([0, 2, 4, 6, 8])
```

- np.arange(0., 0.5, 0.1)

```
array([ 0. ,  0.1,  0.2,  0.3,  0.4])
```

Watch out for round-off problems!

You may prefer 0.1*np.arange(5)

Optional dtype=... everywhere:

```
dtype=np.int
dtype=np.int16
dtype=np.float32
```

...

Array creation

- np.linspace(0., 1., 6)

```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

- np.eye(3)

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

- np.diag([1., 2., 3.])

```
array([[ 1.,  0.,  0.],
       [ 0.,  2.,  0.],
       [ 0.,  0.,  3.]])
```

Indexing

a = np.arange(6)

```
array([0, 1, 2, 3, 4, 5])
```

- a[2]

  ```
  2
  ```

- a[2:4]

  ```
  array([2, 3])
  ```

- a[1:-1]

  ```
  array([1, 2, 3, 4])
  ```

- a[:4]

  ```
  array([0, 1, 2, 3])
  ```

- a[1:4:2]

  ```
  array([1, 3])
  ```

- a[::-1]

  ```
  array([5, 4, 3, 2, 1, 0])
  ```

This works exactly like for lists!

Indexing

a = np.array([[1, 2], [3, 4]])

```
array([[1, 2],
       [3, 4]])
```

- a[1, 0]

  ```
  3
  ```

- a[1, :]   a[1]

  ```
  array([3, 4])
  ```

- a[:, 1]

  ```
  array([2, 4])
  ```

- a[:, :, np.newaxis]

  ```
  array([[[1],
          [2]],

         [[3],
          [4]]])
  ```

Arithmetic

a = np.array([[1, 2], [3, 4]])   a.shape = (2, 2)

```
array([[1, 2],
       [3, 4]])
```

- a + a

```
array([[2, 4],
       [6, 8]])
```

- a + 1

```
array([[2, 3],
       [4, 5]])
```

- a + np.array([10, 20])   array([10, 20]).shape = (2,)

```
array([[11, 22],
       [13, 24]])
```

- a + np.array([[10], [20]])   array([[10], [20]]).shape = (2, 1)

```
array([[11, 12],
       [23, 24]])
```

Broadcasting rules

c = a + b with a.shape==(2, 3, 1) and b.shape==(3, 2)

1) len(a.shape) > len(b.shape)
   ➥ b → b[newaxis, :, :], b.shape → (1, 3, 2)

2) Compare a.shape and b.shape element by element:
   - a.shape[i] == b.shape[i]: easy
   - a.shape[i] == 1: repeat a b.shape[i] times
   - b.shape[i] == 1: repeat b a.shape[i] times
   - otherwise : error

3) Calculate the sum element by element

4) c.shape == (2, 3, 2)

Structural operations

a = (1 + np.arange(4))**2

```
array([ 1,  4,  9, 16])
```

- np.take(a, [2, 2, 0, 1]) or a.take([2, 2, 0, 1])

```
array([9, 9, 1, 4])
```

- np.where(a >= 2, a, -1)

```
array([-1,  4,  9, 16])
```

- np.reshape(a, (2, 2)) or a.reshape((2, 2))

```
array([[ 1,  4],
       [ 9, 16]])
```

- np.resize(a, (3, 5)) or a.resize((3, 5))

```
array([[ 1,  4,  9, 16,  1],
       [ 4,  9, 16,  1,  4],
       [ 9, 16,  1,  4,  9]])
```

- np.repeat(a, [2, 0, 2, 1]) or a.repeat([2, 0, 2, 1])

```
array([ 1,  1,  9,  9, 16])
```

Array structure

a = np.arange(1, 7)

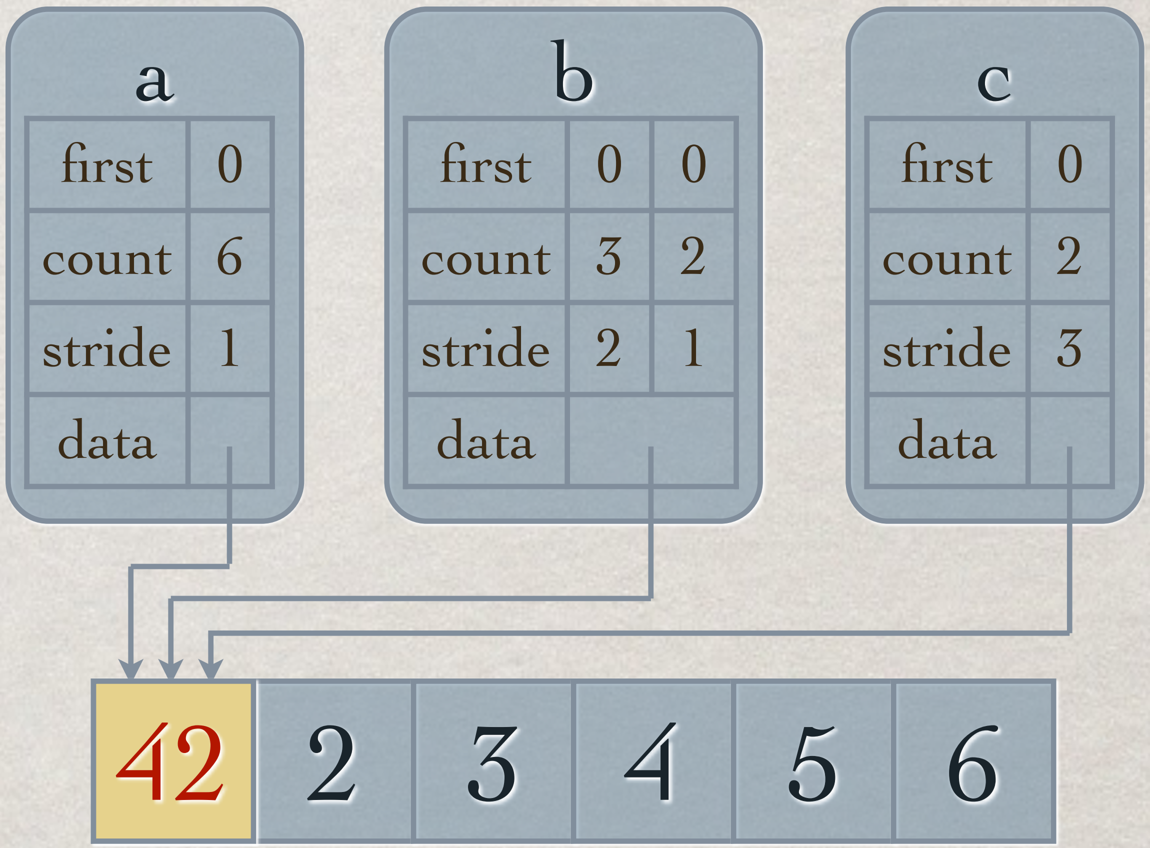
```
array([1, 2, 3, 4, 5, 6])
```

b = np.reshape(a, (3, 2))

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

c = a[::3]

```
array([1, 4])
```



Watch out:

a[0] = 42
print c

```
array([42, 4])
```

Views

A view is a new array (i.e. a new Python object) that references that storage space of the array from which it was created.

If you modify array elements in the original array or in the view, they also change on the other side!

The big question: which operations return views, and which fresh arrays with independent storage areas?

Rule of thumb: An operation creates a view if this is possible for all its allowed arguments. Otherwise it returns a fresh array.

So... how do you find out if an array is a view on another arrays storage space? Check the attribute `base`.

```
a = np.arange(10)        assert a.base is None
b = a[::2]               assert b.base is a
```

Mathematical functions

arccos, arcsin, arctan, arctan2, ceil, cos, cosh, exp, fabs, floor, fmod, hypot, log, log10, sin, sinh, sqrt, tan, tanh

Constants : π, e

Three sources:

- Module `math` : only for real arguments
- Module `cmath` : real and complex
- Module `numpy` :

  real, complex, **arrays**, and more..

Always use module `numpy` **!**

Array programming

- Array operations are fast, Python loops are slow.
  (array operation = everything from module `numpy`)
- Top priority: avoid loops
- It's better to do the work three times with array operations than once with a loop.
- This does require a change of habits.
- This does require some experience.
- NumPy's array operations are designed to make this possible.

Get started with today's exercises!

Array programming strategy

- Identify the kind of operation you want to do (applying a function, filtering, rearranging, ...)
- Go through the list of array operations and check if they do that kind of operation
- Use a mixture of thinking and trying out to get the job done.
- There is often more than one way to do it.

Further reading

Hans-Petter Langtangen
Python Scripting for Computational Science
3rd edition, Springer, 2009

Computing in Science and Engineering
Special Issue "Python: Batteries included"
May/June 2007

Computing in Science and Engineering
Special Issue "Scientific Python"
March/April 2011