PYTHON FOR STRUCTURAL BIOINFORMATICS

Sophie COON
and
Michel SANNER
The Scripps Research Institute
La Jolla, California

SCHEDULE

- I - Fundamentals
  Code development strategies, Python
- II - PMV
  Fundamentals, main commands
- III - From building blocks to applications
  MolKit, DejaVu ViewerFramework, ...
  Putting it all together
  Writing a simple command
- Conclusion

I - Fundamentals

- Code development strategies
- Python primer
I - Fundamentals

- Code development strategies
  - The challenge
  - Traditional solution
  - Our solution

The challenge

- Visualization
- Docking Methods
- Protein Engineering
- Folding
- Ab Initio Methods
- Molecular Surfaces
- Modeling
- Electrostatic Calculations
- MM - MD
- Etc...
“Traditional” solution

- Electrostatic Calculations
- Visualization
- Protein Engineering
- Folding
- Docking Methods
- Sequence Analysis
- MM - MD
- Ab Initio Methods
- Etc...

Our solution

- High level language as a scripting environment
- Molecular Surfaces
- Molecules
- Data Base
- Electrostatic
- Delaunay
- Homology
- CSG
- 3D Viewer
- MM-MD
- New Method
- Your Method
- Writing an application

- High level coding
- Code re-use
- Extensible
Why Python?

Our language needed:
- Object-Oriented
- Advanced data structures
- Powerful data-parallel arrays
- Readability and modularity
- High-level
- Platform independence
- Interpreted

Not met by:
- Tcl, Perl, C, …
- Tcl, Perl
- Tcl
- Perl
- C, C++, Fortran
- C, C++, Java, …
- C, C++, Java, …

Python based molecular software
  http://www.cdl.ucsf.edu/chimera
- MMTK: CNRS - Institut de Biologie Structurale
  http://starship.python.net/crew/hinsen/mmtk.html
- Pymol: Delano Scientific.
  http://www.pymol.org
- PyDayLight: Daylight Chemical Information Systems, Inc.
  http://starship.python.net/crew/dalke/PyDaylight/
  http://www.ks.uiuc.edu/~jim/mdtools/
- ...

I - Fundamentals
- Python primer
  - Language characteristics
  - Basics
  - Standard library
  - Extending Python
  - Numeric extension
Language characteristics

- Interpreted, high level, object-oriented
- Flexible and extensible
- Introspection, self-documenting
- Platform independent
- Open-source
- Rapidly gaining acceptance

Basics

```python
>>> a = 2          # integer
>>> b = 7.5        # float
>>> c = 'hello'    # string
>>> d = "World!"   # string too
>>> # this is a comment
>>> print c+d
>>> print "sum :", a+b
```

List - mutable sequences

```python
>>> lst = [2, 1,'hello']
>>> lst.append(5)
>>> lst.remove(1)
>>> lst.insert(1,'Paul')
>>> lst.sort()
```

```plaintext
lst contains:
- [2, 1, 'hello']
- [2, 1, 'hello', 5]
- [2, 'hello', 5]
- [2, 'Paul', 'hello', 5]
- [2, 5, 'Paul', 'hello']
```
Tuple - immutable sequences

```python
tup = (1, 2, 'spam')  # automatic packing and unpacking
tup = 1, 2, 'spam'
(a, b, c) = tup
print a, b, c  # correct singleton syntax
(tup = (1,))
tup[0] = 6   # ERROR: cannot assign value in an immutable sequence!
```

Sequences indexing and slicing

```python
lst = [1, 'b', 6, 'a', 8, 'e']
(lst[2:4])
(lst[2:-2])
(lst[-4:-2])
```

Dictionary

```python
dict = {key1: value1, key2: value2, ...}
dict = { 'Marc': 25, 30: 'Jim', 5.7: 'joe' }
dict[5.7] = 'joe'
dict.items()  # Accessing the information
dict.values()  # ['Marc', 30, 'Jim', 5.7, 'joe']
dict.keys()  # (Marc, 25, 30, 'Jim', 5.7, 'joe')
dict.has_key('Marc')  # True
```
Control flow

- While:
  >>> b = 0
  >>> while b < 5:
  ...    print b
  0 1 2 3 4

- If:
  >>> q = 'Please Enter a number'
  >>> x = int( raw_input (q))
  >>> if x == 0:
  ...    print 'x equals 0'
  ...  elif x < 0:
  ...    print 'x is negative'
  ...  else:
  ...    print 'x is positive'

- For, range(), break, continue:
  >>> seq = range(-3, 4, 1)
  >>> print seq
  [-3, -2, -1, 0, 1, 2, 3]
  >>> for s in seq:
  ...    if s < 0:
  ...      continue
  ...    else:
  ...      print s
  0, 1, 2, 3

Functions and arguments

Positional arguments (required)

```
def func( a, b, n1=10, n2='hello' ):
```

Function name: `func`

Argument matching:
```
  a  b  n1  n2
```

- `func( 2, 'string', 3.14 )`
  - `2` string 3.14
  - `ERROR: missing argument`

- `func( 7.2,  'string', n2=15)`
  - `7.2` string 10 15
  - `ERROR: positional argument after named argument`

- `func( 'hello', 2 , 5, 'bye')`
  - `hello` 2 5 bye
  - `ERROR: positional argument after named argument`

- `func( n1 = 5 )`
  - `n1` 5
  - `ERROR: missing argument`

- `func( n1=12, 3.14 )`
  - `n1` 12
  - `ERROR: missing argument`

Functions - Arbitrary arguments

```
def func( *args, **kw )
```

Argument matching:
```
  args  kw
```

- `func( 2, 'string', n1 = 5, n2 = 'w' )`
  - `(2, 'string')` n1: 5, n2: 'w'

Combining the two argument passing methods
```
>>> def func(a, b, f=10, *args, **kw):
```
Classes - basics

class Rectangle:
def __init__(self, width, length):  # Constructor
    self.w = width  # instance attribute
    self.l = length  # instance attribute

# call the constructor to create an instance
rect1 = Rectangle(5, 6)
# create another instance
rect2 = Rectangle(4, 2)
# access instance’s attributes
print rect1.w, rect1.h                      5 6
print rect2.w, rect2.h                      4 2

Classes - methods

class Rectangle:
def __init__(self, width, length):  # Constructor
    self.w = width  # instance attribute
    self.l = length  # instance attribute
def area(self):  # method area
    return self.w * self.l

# call the constructor to create an instance
rect1 = Rectangle(5, 6)
rect2 = Rectangle(4, 2)
# calling a method
print rect1.area()                                        30
print rect1.area() + rect2.area()                 38

Classes - overwriting operators

class Rectangle:
def __init__(self, width, length):  # Constructor
    self.w = width  # instance attribute
    self.l = length  # instance attribute
def __add__(self, right):  # defines rect1+rect2
    def __mult__(self, right):  # defines rect1*rect2
    def __repr__(self):  # defines print(rect1)
def __call__(self, *args, **kw):  # define rect1('hello')
    etc ...

* def __repr__(self, *args, **kw) instead of def __repr__(self,right)
The file `MyModule.py` contains:

```python
def func1(b):
    print 'You called func1 with b = ', b
```

```python
>>> import MyModule
>>> dir(MyModule)
['__builtins__', '__doc__', '__file__', '__name__', 'func1']
>>> MyModule.func1(10)
You called func1 with b = 10
>>> from MyModule import func1
>>> func1(10)
You called func1 with b = 10
>>> from MyModule import *
>>> func1(10)
You called func1 with b = 10
```

### Packages - organizing modules

```python
>>> import sys
>>> print sys.path  # print the Python path
['dir1', 'dir2', 'dir3', ... ]
```

```
subdir1:
    __init__.py
    foo.py:
    def Func(a,b):
    ...
    class Object:
    ...

subdir2:
    data.txt
    bar.py
```

- `subdir1` is a package
- `subdir2` is NOT a package, `bar.py` CANNOT be imported

```python
from subdir1 import foo
from subdir1.foo import Func
from subdir1.foo import Object
```

### Standard libraries

- `sys`, `os`, `string`, `math`, `cgi`, `commands`, `shelve` etc.

Example:

```python
>>> import sys
>>> path = sys.path
>>> print path[3]
C:\Program Files\Python\lib\plat-win']
```
Extending Python

Add functionality to Python
Gaining access to legacy code

Extending Python

Implement

NEW MODULE

Wrap

LEGACY
C, C++

FORTRAN

Platform independent
Portability

Platform dependent
speed

Wrapping C code

```c
#include "MyLib.h"

int fact(int n) {
    if (n<=1) return 1;
    else return n*fact(n-1);
}
```

```python
PyObject *fact_w(PyObject *obj)
{
    int n = PyGetInt(obj);
    int res = fact(n);
    PyObject *Pyres = PyMakeInt(res);
    return Pyres;
}
```

```python
>>> import MyLib
>>> a = 5
>>> MyLib.fact(a)
```

MyLib.h

extern int fact(int n)

MyLib.a

/* Compute factorial of n */
int fact(int n) {
    if (n<=1) return 1;
    else return n*fact(n-1);
}

MyLibModule.so
## Numeric extension

Efficient storage and manipulation of large arrays of data.

### Concepts

- In C:
  - PyArrayObject
  - void *data
  - int *shape
  - char typecode

- In Python:
  - `import Numeric`
  - `ar = Numeric.array([(1,5,9), (-2,3,4,26), (1,2,3,7)])`
  - `print ar.shape` → (3,4)
  - `print ar.typecode()` → 'l'
  - `print ar.iscontiguous()` → '1'

### Reshaping

- `B = Numeric.array([[1,2,3],[4,5,6]])`
- `Numeric.reshape(B,(2,3))`
  - `shape (2,3)` → (3,2)
  - `1 2 3
    4 5 6`

- `Numeric.reshape(B,(1,3))`
  - `1 2 3
    4 5 6`
  - `shape (2,3)` → (1,6) same as (6)
Indexing & Slicing

>>> B = Numeric.array([[1,2,3],[4,5,6],[7,8,9]])

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

B[1, 0] element

B[0, :] row

B[1:3, 1] column

* B[1,0] instead of B[0,1]

Element-wise operations

Element-wise operation at C speed!

\[
\begin{bmatrix}
1 & 2 \\
3 & 4 \\
\end{bmatrix} +
\begin{bmatrix}
5 & 6 \\
7 & 8 \\
\end{bmatrix} =
\begin{bmatrix}
6 & 8 \\
10 & 12 \\
\end{bmatrix}
\]

binary operators (bop): +, -, /, %, power ...
unary operators (uop): sin, cos, sqrt ...

Universal functions.

The most common Numeric universal functions:
take, transpose, repeat, choose, ravel,
nonzero, where, compress, diagonal, trace,
searchsorted, sort, argsort, argmax, fromstring,
dot, matrixmultiply, clip, indices, swapaxes,
concatenate, innerproduct, array_repr, array_str,
resize, diagonal, repeat, convolve, where, identity,
sum, cumsum, product, cumproduct, alltrue,
sometrue....
II - PMV

- Fundamentals
- Commands
- Specialized extension: ADT

II - PMV

- Fundamentals
  - Application design features

PMV : Python Molecule Viewer

DEMO
Application design features

- Dynamic loading of commands
- Python shell for scripting
- Dual interaction mode (GUI/Shell)
- Lightweight commands: Macros
- Command logging
- Dynamic commands (introspection)
- User-preferences / customization

Application design features (cont'd)

- Load multiple molecules
- Hierarchical representation of molecules
- Create/Select homogeneous sets
- Current selection concept
- Commands apply to current selection
- Interactive commands

II - PMV

Commands

- Create, delete, write molecules
- Selection
- Basic representations and coloring
- Advanced representations
- Editing molecules
- ...
Create, delete, write molecules

- Create:
  - PDBReader
  - Mol2Reader
  - PDBQReader
  - PDBQSReader
  - PQRReader
  - GeneralReader

- Delete
  - deleteMol

- Write
  - PDBWriter

Selection

- From string
- By picking
  - molecule, chain, residue, atom
- By distance
- Displayed lines or cpk
- Invert selection
- On chain
- ...

Basic representations and coloring

- Color geometries by:
  - atom type
  - residue type
  - shapely, Rasmol, N to C
  - chains
  - molecules
  - properties
  - secondary structure type
  - ...

- Display by:
  - lines
  - cpk
  - sticks and balls
  - ...

- Label:
  - by properties
Advanced representations

- MSMS molecular surface
  - compute & display:
    - MSMSMol
    - MSMSSel
- CA trace:
  - compute
  - extrude
  - display
- Spline:
  - compute
  - display

Secondary Structure:

- get SS information:
  - from file
  - from stride
  - extrude
    - default (rectangle, circle)
    - circle
    - rectangle
    - ellipse
    - ...
  - display

Editing molecules

- AIDE Module:
  (pyBabel reimplementation of some of the Babel v1.6 functionalities)
  - atom type assignment
  - gasteiger charges calculation
  - atom type conversion
  - rings detection
  - bond order assignment
  - aromaticity detection
  - hydrogen atoms addition

II - PMV

- Specialized extension: ADT
  - AutoDock fundamentals
  - AutoDock ToolKit (ADT)
AutoDock fundamentals
Automated docking of a flexible ligand to macromolecules using affinity grids

AutoDock ToolKit (ADT)
- AutoTors: ligand preparation
- AutoGpf: grid definition
- AutoDpf: docking parameters definition
- AutoStart: job launching and monitoring
- AutoAnalyze: docking results analysis

DEMODEMO

III - From Building Blocks to applications
- MolKit
- DejaVu
- ViewerFramework
- Putting it all together
- Writing a simple command

* Snake image related to AutoDock
III - From Building Blocks to applications

- **MolKit**
  - Hierarchical data-structure
  - TreeNode and TreeNodeSets
  - Derived classes
  - Parsers
  - Examples

**Hierarchical data-structure**

```
TreeNode
  .parent
  .top
  .children
  .elementType
  .adopt(child)

TreeNodeSet(ListSet)
  [TreeNode, TreeNode, ...]
```

**Hierarchical structure (cont'd)**

- building trees by adoption
  ```python
  parent = TreeNode()
  child = TreeNode()
  parent.adopt(child)
  ```
- TreeNodeSet slicing and indexing
  ```python
  node.children[5]
  node.children[10:25]
  node.children[0].children[-1]
  ```
- multi-level hierarchy
  ```python
  node.children[0].children[1]
  ```
- dynamic adding of new members
  ```python
  node = TreeNode()
  node.newMember = myvalue
  ```
- shortcut to access children’s members
  ```python
  print node.children.name
  ```
  ```python
  [ 'NoName', 'NoName', 'NoName', ... ]
  ```
TreeNode, TreeNodeSet

- TreeNodeSet:
  - Boolean operation
  - uniq()
  - split()
  - sort()
  - NodesFromName()
  - findChildrenOfType()
  - findParentOfType()
  - ...

- TreeNode:
  - adopt() / remove()
  - full_name()
  - NodeFromName()
  - split() / merge()
  - getParentOfType()
  - findType()
  - compare()
  - assignUniqIndex()
  - isAbove() / isBelow()
  - ...

TreeNode and TreeNodeSet specialization

MoleculeSet
- Molecule
  - Chain
    - Residue
      - Atom

Parsers

from MolKit pdbParser import PdbParser
parser = PdbParser('1crn.pdb')
mols = parser.parse()
Examples

```python
>>> from MolKit import Read
>>> molecules = Read('./1crn.pdb')
>>> mol = molecules[0]  # Read returns a ProteinSet
>>> print mol.chains.residues.name
>>> print mol.chains.residues.atoms[20:85].full_name()
>>> from MolKit.molecule import Atom
>>> allAtoms = mol.findType(Atom)
>>> set1 = allAtoms.get(lambda x: x.temperatureFactor > 20)
>>> allResidues = allAtoms.parent.uniq()
>>> import Numeric
>>> for r in allResidues:
...     coords = r.atoms.coords
...     r.geomCenter = Numeric.sum(coords) / len(coords)
```
Demo Code

```python
>>> from DejaVu import Viewer
>>> vi = Viewer()
>>> from DejaVu.Spheres import Spheres
>>> centers = [[0.0,0.0],[3.0,0.0],[0.0,3.0]]
>>> s = Spheres(sph, centers = centers)
>>> s.Set(quality=10)
>>> vi.AddObject(s)
```

Features

- OpenGL Lighting and Material model
- Object hierarchy with transformation and rendering properties inheritance
- Arbitrary clipping planes
- Material editor
- DepthCueing (fog), global anti-aliasing
- glScissors/magic lens
- Multi-level picking
- Extensible set of geometries

Geometries

<table>
<thead>
<tr>
<th>Geom</th>
<th>IndexedGeoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyLine</td>
<td>IndexedPolyLines</td>
</tr>
<tr>
<td>Points</td>
<td>IndexedPolygons</td>
</tr>
<tr>
<td>Spheres</td>
<td>Triangle_Strip</td>
</tr>
<tr>
<td>Labels</td>
<td>Quad_Strip</td>
</tr>
<tr>
<td>Arc3D...</td>
<td>Cylinders ...</td>
</tr>
</tbody>
</table>
DejaVu and MolKit

```python
>>> from MolKit import Read
>>> molecules = Read('/tsri/pdb/struct/1crn.pdb')
>>> mol = molecules[0]  # Read returns a ProteinSet
>>> coords = allAtoms.coords
>>> radii = allAtoms.radius
>>> sph = Spheres('sph', centers = coords, radii = radii, quality=10)
>>> vi.AddObject(sph)
```

III - From Building Blocks to applications

- ViewerFramework
  - Overview
  - Design features
  - Implementation
Overview

- Numeric
- PyOpenGL
- Tkinter
- DejaVu
- Idle
- ViewerFramework

Design features

- Dynamic loading of commands
- Python shell for scripting
- Dual interaction mode (GUI/Shell)
- Support for command:
  - development, logging, GUI, dependencies
- Lightweight commands: Macros
- Dynamic commands (introspection)
- Extensible set of commands

Implementation

- ViewerFramework
- VFGUI
- VFCommandsGUI
  - VPCommand
  - Objects
    - GeomContainer
    - UserPreference
  - addObject(...)  
  - removeObject(...)  
  - addCommand(...)  
  - askFileOpen(...)  
  - askFileSave(...)  
  - getUserInput(...)
III - From Building Blocks to applications

- Putting it all together:
  - Specializing ViewerFramework
  - MolKit in PMV
  - DejaVu in PMV

Specializing ViewerFramework

Specializing ViewerFramework (cont'd)
PMV Architecture

MolKit in PMV

```python
>>> print mv.Mols
<MoleculeSet instance> holding 2 Protein

>>> mv.Mols[0]
<Protein instance> 1crn with 1 MolKit.protein.Chain

>>> from MolKit.protein import Residue
>>> residues = mv.Mols.findType(Residue)
>>> residues
<ResidueSet instance> holding 154 Residue

>>> residues.myIndex = range(len(residues))
>>> residues[1:10].myIndex
[1,2,3,4,5,6,7,8,9]
```

DejaVu in PMV

```python
>>> # access to DejaVu features from the pyShell
>>> vi = mv.GUI.VIEWER
>>> camera = vi.cameras[0]
>>> camera.Set(color=(1.,1.,1.))
>>> vi.Redraw()

>>> # show the Viewer's original GUI
>>> vi.GUI.root.deiconify()

>>> # hide the Viewer's original GUI
>>> vi.GUI.root.withdraw()
```
III - From Building Blocks to applications

- Writing a simple command
  - MVCommand overview
  - Subclassing MVCommand
  - Loading the command

### MVCommand overview

```python
class MyCommand(MVCommand):
    def guiCallback(self, *args, **kw):
        apply(self.doitWrapper, args, kw)
    def doit(self, *args, **kw):
        pass
    def doitWrapper(self, *args, **kw):
        self.beforeDoit()
        self.vf.tryto(apply(self.doit, args, kw))
        self.afterDoit()
```

### Subclassing MVCommand

```python
class MyReader(MVCommand):
    def guiCallback(self):
        fTypes = [('PDB file','*.pdb'), ('PDBQ file','*.pdbq'),
                  ('MOL2 file','*.mol2')]
        filename = self.vf.askFileOpen(types=fTypes,
                                       title='Choose molecule file')
        mol = self.doitWrapper(filename, log=1)
        return mol
    def __call__(self, filename = None, log=1, redraw=0):
        if filename is None: self.guiCallbak()
        else: self.doitWrapper(filename, log=log, redraw = redraw)
    def doit(self, filename):
        from MolKit import Read
        molecules = Read(filename)
        self.vf.AddMolecule(molecules)
```
**Loading the command**

```python
mv.addCommand( MyCommand(),
               'myCommand',
               MyCommandGUI )
```

An instance of a `Command` describes the GUI associated with a command (radiobutton, checkbutton, menu entry ...)

**Example**

```python
>>> from Pmv.myCmd import MyReader
>>> from ViewerFramework.VFCommand import CommandGUI

# Create a menu entry GUI for the command MyReader
>>> MyReaderGUI = CommandGUI()

# Add the GUI to the menuRoot in the File menu
>>> MyReaderGUI.addMenuCommand('menuRoot', 'File',
                               'Read File ...', index=0)

# Add the command with its associated GUI, and name to PMV
>>> mv.addCommand(MyReader(), 'myread', MyReaderGUI)
```

**Conclusion**

- Validity of the approach
- Python
- Availability
- Future directions
Validity of the approach

- Set of components
  - extensible
  - inter-operable
  - re-usable
  - short development cycle
- User base expanding beyond our lab.
- Components re-use outside the field of structural biology

Python

- Appropriate language for this approach
  - modularity, extensibility, dynamic loading, object-oriented, virtually on any platform, many extensions from third party
- Rapidly growing community of programmers using Python for biological applications
- Short comings
  - reference counting, distribution mechanism, no strong typing

Availability

- Modularity enables fine grain licensing schemes (a la carte)
- Core modules are freely available
- Online Download site:
  http://www.scripps.edu/~sanner/Python
Future directions

- Add support for editing molecular structures (i.e. mutations, deletion, addition)
- Interface with MMTK, Tinker, ....
- Enhance documentation and tutorials
- Setup a CVS server for programmers wanting to help!
- Too many to list ....

Acknowledgments

- Christian Carrillo, Kevin Chan
- Ruth Huey, Fariba Fana
- Vincenzo Tchinke, Greg Paris
- MGL at TSRI
- Pat Walters, Matt Stahl
- Don Bashford
- Guido van Rossum & Python community