Painless Application Development with 3D Slicer and Python

Basic Python and Index tricks

Demian Wassermann
LMI 09/2008
Python Basics
Python Basics

- Object oriented language
- Interpreted language
- Everything is an object
- References are the main way of passing parameters
- Duck typing
Python Basics

- Object oriented language
- Some functional capabilities
- Lambda expression
- Lists by comprehension
- Partial evaluation not directly supported
Python Basics

- Object oriented language
- Some functional capabilities
- Everything can be defined or redefined in execution time
Python basic Datatypes

- Literals (i.e. integers, floats, complex and characters)
- Tuples: fixed combinations of objects `t=(‘a’,2)`
- Lists: resizable combinations of objects `t=[‘a’,2]`
- Dictionaries: key/value pairs `t[‘a’]=2`
- Sets: non-iterable containers with a fast pertenence operation `t={‘a’,2}`

- Functions
- Classes
- Modules
Numpy basic Datatypes

- array datatype
- Multidimensional array
  - Operations are done in an element by element basis
- matrix datatype
  - Bidimensional array of elements
  - matrix semantics
Numpy: slicing

```python
>>> a[0, 3:5]
array([3, 4])

>>> a[4:, 4:]  
array([[44, 45],
       [54, 55]])

>>> a[:, 2]
array([2, 22, 52])

>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```

Slicing does not create copies of the array's contents.
### Numpy: fancy indexing

**INDEXING BY POSITION**

```python
>>> a = arange(0,80,10)

# fancy indexing
>>> y = a[[1, 2, -3]]
>>> print y
[10 20 50]

# using take
>>> y = take(a,[1,2,-3])
>>> print y
[10 20 50]
```

**INDEXING WITH BOOLEANS**

```python
>>> mask = array([0,1,1,0,0,1,0,0], dtype=bool)

# fancy indexing
>>> y = a[mask]
>>> print y
[10,20,50]

# using compress
>>> y = compress(mask, a)
>>> print y
[10,20,50]
```
Numpy: fancier indexing

>>> a[(0,1,2,3,4),(1,2,3,4,5)]
anarray([ 1, 12, 23, 34, 45])

>>> a[3:, [0, 2, 5]]
anarray([[30, 32, 35],
        [40, 42, 45],
        [50, 52, 55]])

>>> mask = array([1, 0, 1, 0, 0, 1],
dtype=bool)

>>> a[mask, 2]
anarray([2, 22, 52])

Unlike slicing, fancy indexing creates copies instead of views into original arrays.

[Jones, Oliphant]
Numpy broadcasting

Semantic of binary operations between arrays

\[\begin{align*}
4 \times 3 &+ 4 \times 3 = 4 \times 3 \\
4 \times 3 &+ 3 = 3 \\
4 \times 1 &+ 3 = 3 \\
\end{align*}\]
Full example: NCuts Nystrom

Get the most significative eigenvectors of a normalized matrix $W$, using only the submatrices $A$ and $B$

- Subsampling $W$ to get $A$
- Normalizing of the partial parts
- Eigenvalue decomposition of $A$
- Estimation of the Eigenvectors of $W$

$$W = D^{-\frac{1}{2}} WD^{-\frac{1}{2}}.$$
Full example: NCuts Nystrom

Subsampling D to get A and B
Full example: NCuts N"ystrom

Subsampling D to get A and B

#W is the initial matrix
#ratio is the relative size of A with respect to W

import numpy
from numpy import random

```python
import numpy
from numpy import random
```

$W$ is the initial matrix
$\text{ratio}$ is the relative size of $A$ with respect to $W$
Full example: NCuts Nystrom

Subsampling D to get A and B

#W is the initial matrix
#ratio is the relative size of A with respect to W

import numpy
from numpy import random

shuffled_indexes = numpy.arange(W.shape[0], dtype=int)
random.shuffle(shuffled_indexes)
Full example: NCuts Nyström

Subsampling D to get A and B

#W is the initial matrix
#ratio is the relative size of A with respect to W

import numpy
from numpy import random

shuffled_index = numpy.arange(W.shape[0], dtype=int)
random.shuffle(shuffled_index)

Na = int(numpy.round(elementQty*ratio))
Nb = elementQty-Na
a_index = shuffled_index[:Na]
b_index = shuffled_index[Na:]
**Full example:**
NCuts Nystrom

**Subsampling D to get A and B**

#W is the initial matrix
#ratio is the relative size of A with respect to W

```python
import numpy
from numpy import random

shuffled_indexes = numpy.arange(W.shape[0], dtype=int)
random.shuffle(shuffled_indexes)

Na = int(numpy.round(elementQty*ratio))
Nb = elementQty-Na
a_indexes = shuffled_indexes[:Na]
b_indexes = shuffled_indexes[Na:]

A = numpy.asmatrix(D[a_indexes,:][:,a_indexes])
B = numpy.asmatrix(D[a_indexes,:][:,b_indexes])
```
Full example: NCuts Nýstrom

Normalizing A and B

\[ \mathcal{W} = D^{-\frac{1}{2}} W D^{-\frac{1}{2}} \]

\[ \hat{d} = \begin{bmatrix} a_r + b_r \\ b_c + B^T A^{-1} b_r \end{bmatrix} \]
Full example: NCuts Nýststrom

Normalizing A and B

\[ \hat{d} = \begin{bmatrix} a_r + b_r \\ b_c + B^T A^{-1} b_r \end{bmatrix} \]

\[ W = D^{-\frac{1}{2}} W D^{-\frac{1}{2}} \]

\[ a_r = A.\text{sum}(1) \]
\[ b_r = B.\text{sum}(1) \]
\[ b_c = B.T.\text{sum}(1) \]
Full example: NCuts Nüstrom

Normalizing A and B

```python
a_r = A.sum(1)
b_r = B.sum(1)
b_c = B.T.sum(1)

d = numpy.vstack((
    a_r + b_r,
    b_c + (B.T * linalg.inv(A)) * b_r
))

d_inv_sqr = 1./numpy.sqrt(d)

W = D^{-\frac{1}{2}}WD^{-\frac{1}{2}}

\hat{d} = \begin{bmatrix}
a_r + b_r \\
b_c + B^T A^{-1} b_r
\end{bmatrix}
```
Full example: NCuts Nystrom

Normalizing A and B

\[
\begin{align*}
a_r &= \text{A}_{\text{sum}}(1) \\
b_r &= \text{B}_{\text{sum}}(1) \\
b_c &= \text{B} \times \text{T}_{\text{sum}}(1) \\
d &= \text{numpy.vstack}((\text{a}_r + \text{b}_r, \\
& \quad \text{b}_c + (\text{B} \times \text{T} \times \text{linalg.inv(A)}) \times \text{b}_r)) \\
d_{\text{inv_sqr}} &= 1/\text{numpy.sqrt}(d) \\
A_n &= \text{numpy.multiply}((\text{d}_{\text{inv_sqr}}[:\text{Na}], \text{A} ),\(\text{d}_{\text{inv_sqr}}[:\text{Na}].T) \\
B_n &= \text{numpy.multiply}((\text{d}_{\text{inv_sqr}}[\text{Na}:], \text{B} ),\(\text{d}_{\text{inv_sqr}}[\text{Na}]) \\
W &= D^{-\frac{1}{2}} WD^{-\frac{1}{2}} \\
\hat{d} &= \begin{bmatrix} a_r + b_r \\
b_c + B^T A^{-1} b_r \end{bmatrix}
\end{align*}
\]
Equivalent syntax for the normalization

d_inv_sqrt = numpy.asarray(d_inv_sqrt)
A*n = d_inv_sqrt[:Na]*numpy.asarray(A)*d_inv_sqrt[:Na].T
#Warning here the d_inv_sqrt and A are arrays not matrices
**Full example: NCuts Nystrom**

Eigenvalue decomposition of $A$

$$
\Delta, U = \text{linalg.eig}(A) \\
\Delta_{\text{inv}} = \text{numpy.asmatrix}(\text{numpy.diag}(1./\Delta)) \\
U_{\text{bar}} = \text{numpy.vstack}((U, \\
\quad \text{Bn.T*U*Delta inv}) \\
))
$$

return $\Delta$, $U_{\text{bar}}[\text{numpy.argsort}($ shuffled_indexes $),:]$

$$
\bar{U} = \begin{bmatrix} 
U \\
\mathbf{B}^T \mathbf{U} \Lambda^{-1} 
\end{bmatrix}
$$
Time and energy to take a look at one more module?