Python Programming for Data Processing and Climate Analysis

Jules Kouatchou and Hamid Oloso
Jules.Kouatchou@nasa.gov and Amidu.o.Oloso@nasa.gov

Goddard Space Flight Center
Software System Support Office
Code 610.3

March 11, 2013
Training Objectives

We want to introduce:

- Basic concepts of Python programming
- Array manipulations
- Handling of files
- 2D visualization
- EOFs
Obtaining the Material

Slides for this session of the training are available from:

https://modelingguru.nasa.gov/docs/DOC-2322

You can obtain materials presented here on discover at

/discover/nobackup/jkouatch/pythontraininggsfc.tar.gz

After you untar the above file, you will obtain the directory

`pythontraininggsfc/` that contains:

- Examples/
- Slides/
We installed a Python distribution. To use it, you need to load the modules:

```
module load other/comp/gcc-4.5-sp1
module load lib/mkl-10.1.2.024
module load other/SIVO-PyD/spd_1.7.0_gcc-4.5-sp1
```
**Numbers:**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>1234, -24, 0</td>
</tr>
<tr>
<td>Unlimited precision integers</td>
<td>99999999999999L</td>
</tr>
<tr>
<td>Floating</td>
<td>1.23, 3.14e-10, 4E210, 4.0e+210</td>
</tr>
<tr>
<td>Oct and hex</td>
<td>0177, 0x9ff</td>
</tr>
<tr>
<td>Complex</td>
<td>3+4j, 3.0+4.0j, 3j</td>
</tr>
</tbody>
</table>
## Recall from the Last Session-2

### Built-in object types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>3.1415, 1234, 999L, 3+4j</td>
</tr>
<tr>
<td>Strings</td>
<td>'spam', &quot;guido's&quot;</td>
</tr>
<tr>
<td>Lists</td>
<td>[1, [2,'tree'], 4]</td>
</tr>
<tr>
<td>Dictionaries</td>
<td>'food':'spam', 'taste':'yum'</td>
</tr>
<tr>
<td>Tuples</td>
<td>(1,'spam', 4, 'U')</td>
</tr>
<tr>
<td>Files</td>
<td><code>text = open('eggs', 'r').read()</code></td>
</tr>
</tbody>
</table>
What Will be Covered Today

1. NumPy
   - Arrays
   - Array Indexing and Slicing
   - Loop over Array
   - Vectorization
   - Matrices
   - Matlab Users
   - Linear Algebra

2. SciPy
   - Interpolation
   - Optimization
   - Integration
   - Statistical Analysis
   - Fast Fourier Transform
   - Signal Processing
NumPy
Useful Links for NumPy

- **Tentative NumPy Tutorial**
  http://www.scipy.org/Tentative_NumPy_Tutorial

- **NumPy Reference**
  http://docs.scipy.org/doc/numpy/reference

- **NumPy for MATLAB Users**
  http://mathesaurus.sourceforge.net/matlab-numpy.html

- **NumPy for R (and S-Plus) Users**
  http://mathesaurus.sourceforge.net/r-numpy.html
What is NumPy?

- Efficient array computing in Python
- Creating arrays
- Indexing/slicing arrays
- Random numbers
- Linear algebra
- (The functionality is close to that of Matlab)
The critical thing to know is that Python for loops are slow! One should try to use array-operations as much as possible.

NumPy provides mathematical functions that operate on an entire array.
>>> from numpy import *
>>> n = 4
>>> a = zeros(n) # one-dim. array of length n
>>> print a # str(a), float (C double) is default type
[ 0. 0. 0. 0.]
>>> a # repr(a)
array([[ 0., 0., 0.],
       [ 0., 0., 0.]])
>>> p = q = 2
>>> a = zeros((p,q,3)) # p*q*3 three-dim. Array
>>> print a
[[[ 0. 0. 0.]
  [ 0. 0. 0.]]
 [[ 0. 0. 0.]
  [ 0. 0. 0.]]]

>>> a.shape # a’s dimension
(2, 2, 3)
Making Int, Float, Complex Arrays

```python
>>> a = zeros(3)
>>> print a.dtype # a’s data
Type float64
>>> a = zeros(3, int)
>>> print a
[0 0 0]
>>> print a.dtype
Int32
>>> a = zeros(3, float32) # single precision
>>> print a
[ 0. 0. 0.]
>>> print a.dtype
Float32
>>> a = zeros(3, complex)
>>> a
array([ 0.+0.j, 0.+0.j, 0.+0.j])
>>> a.dtype
dtype('complex128')
```
Array with Sequence of number

- `linspace(a, b, n)` generates `n` uniformly spaced coordinates, starting with `a` and ending with `b`

  ```python
generate x = linspace(-5, 5, 11)
generate print x
[-5. -4. -3. -2. -1. 0. 1. 2. 3. 4. 5.]
```

- A special compact syntax is available through the syntax

  ```python
generate a = r_-5:5:11j # same as linspace(-1, 1, 11)
generate print a
[-5. -4. -3. -2. -1. 0. 1. 2. 3. 4. 5.]
```

- `arange` works like `range (xrange)`

  ```python
generate x = arange(-5, 5, 1, float)
generate print x # upper limit 5 is not included!!
[-5. -4. -3. -2. -1. 0. 1. 2. 3. 4.]
```
Array Construct from a Python List

- array(list, [datatype]) generates an array from a list:

  >>> pl = [0, 1.2, 4, -9.1, 5, 8]
  >>> a = array(pl)

- The array elements are of the simplest possible type:

  >>> z = array([1, 2, 3])
  >>> print z              # int elements possible
  [1 2 3]
  >>> z = array([1, 2, 3], float)
  >>> print z
  [ 1.  2.  3.]

- A two-dim. array from two one-dim. lists:

  >>> x = [0, 0.5, 1]; y = [-6.1, -2, 1.2]       # Python lists
  >>> a = array([x, y]) # form array with x and y as rows
From ”Anything” to a NumPy Array

- Given an object a,

  ```python
  a = asarray(a)
  converts a to a NumPy array (if possible/necessary)
  ```

- Arrays can be ordered as in C (default) or Fortran:

  ```python
  a = asarray(a, order='Fortran')
  isfortran(a)  # returns True of a’s order is Fortran
  ```

- Use asarray to, e.g., allow flexible arguments in functions:

  ```python
  def myfunc(some_sequence, ...):
      a = asarray(some_sequence)
      # work with a as array

  myfunc([1,2,3], ...)
  myfunc((-1,1), ...)
  myfunc(zeros(10), ...)
  ```
Changing Array Dimension

```python
>>> a = array([0, 1.2, 4, -9.1, 5, 8])
>>> a.shape = (2,3) # turn a into a 2x3 matrix
>>> a.size
6
>>> a.shape = (a.size,) # turn a into a vector of length 6 again
>>> a.shape
(6,)
>>> a = a.reshape(2,3) # same effect as setting a.shape
>>> a.shape
(2, 3)
```
```python
>>> def myfunc(i, j):
...     return (i+1)*(j+4-i)
...

>>> # make 3x6 array where a[i,j] = myfunc(i,j):
>>> a = fromfunction(myfunc, (3,6))
>>> a
array([[ 4.,  5.,  6.,  7.,  8.,  9.],
       [ 6.,  8., 10., 12., 14., 16.],
       [ 6.,  9., 12., 15., 18., 21.]])
```
Basic Array Indexing

```python
a = linspace(-1, 1, 6)
a[-1] = a[0]  # set last element equal to first one
a[:] = 0  # set all elements of a equal to 0
a.fill(0)  # set all elements of a equal to 0

a.shape = (2,3)  # turn a into a 2x3 matrix
print a[0,1]  # print element (0,1)
a[i,j] = 10  # assignment to element (i,j)
a[i][j] = 10  # equivalent syntax (slower)
print a[:,k]  # print column with index k
print a[1,:]
print a[:,:] = 0  # set all elements of a equal to 0
```
More Advanced Array Indexing

```python
>>> a = linspace(0, 29, 30)
>>> a.shape = (5,6)
>>> a
array([[ 0.,  1.,  2.,  3.,  4.,  5.],
       [ 6.,  7.,  8.,  9., 10., 11.],
       [12., 13., 14., 15., 16., 17.],
       [18., 19., 20., 21., 22., 23.],
       [24., 25., 26., 27., 28., 29.]]

>>> a[1:3,:-1:2] # a[i,j] for i=1,2 and j=0,2,4
array([[ 6.,  8., 10.],
       [12., 14., 16.]])

>>> a[::3,2:-1:2] # a[i,j] for i=0,3 and j=2,4
array([[ 2.,  4.],
       [20., 22.]])

>>> i = slice(None, None, 3); j = slice(2, -1, 2)
>>> a[i,j]
array([[ 2.,  4.],
       [20., 22.]])
```
Array Slicing

SLICING WORKS MUCH LIKE STANDARD PYTHON SLICING

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

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

```python
>>> a[:, 2]
array([[2, 12, 22, 32, 42, 52]])
```

STRIDES ARE ALSO POSSIBLE

```python
>>> a[2::2, ::2]
array([[20, 22, 24],
       [40, 42, 44]])
```
Slices Refer the Array Data

- With a as list, a[:] makes a copy of the data
- With a as array, a[:] is a reference to the data

```python
>>> b = a[1,:]
>>> print a[1,1]
12.0
>>> b[1] = 2
>>> print a[1,1]
2.0 # change in b is reflected in a!
```

- Take a copy to avoid referencing via slices:

```python
>>> b = a[1,:].copy()
>>> print a[1,1]
12.0
>>> b[1] = 2
>>> print a[1,1]
12.0 # a is not affected by change in b
```
Integer Arrays as Indices

- An integer array or list can be used as (vectorized) index

```python
>>> a = linspace(1, 8, 8)
>>> aarray([ 1., 2., 3., 4., 5., 6., 7., 8.])
>>> a[[1,6,7]] = 10
>>> a
array([ 1., 10., 3., 4., 5., 6., 10., 10.])
>>> a[range(2,8,3)] = -2
>>> aarray([ 1., 10., -2., 4., 5., -2., 10., 10.])
>>> a[a < 0] # pick out the negative elements of a
array([-2., -2.])
>>> a[a < 0] = a.max()
>>> a
array([ 1., 10., 10., 4., 5., 10., 10., 10.])
```

- Such array indices are important for efficient vectorized code
Loop over Arrays-1

- Standard loop over each element:

```python
for i in xrange(a.shape[0]):
    for j in xrange(a.shape[1]):
        a[i,j] = (i+1)*(j+1)*(j+2)
        print 'a[%d,%d]=%g ' % (i,j,a[i,j]),
    print  # newline after each row
```

- A standard for loop iterates over the first index:

```python
>>> print a
[[ 2.  6. 12.]
 [ 4. 12. 24.]]
>>> for e in a:
...    print e
...
...    print e
...
[[ 2.  6. 12.]
 [ 4. 12. 24.]]
```
View array as one-dimensional and iterate over all elements:

```python
for e in a.flat:
    print e
```

For loop over all index tuples and values:

```python
>>> for index, value in ndenumerate(a):
...     print index, value
...  
(0, 0) 2.0
(0, 1) 6.0
(0, 2) 12.0
(1, 0) 4.0
(1, 1) 12.0
(1, 2) 24.0
```
Array Computations

Arithmetic operations can be used with arrays:

\[ b = 3a - 1 \]  # a is array, b becomes array

The above operation generates a temporary array:

\[
\begin{align*}
tb &= 3a \\
b &= tb - 1
\end{align*}
\]

As far as possible, we want to avoid the creation of temporary arrays to limit the memory usage and to decrease the computational time associated with array computations.
In-Place Array Arithmetics

- With in-place modifications of arrays, we can avoid temporary arrays (to some extent) to compute \( b = 3a - 1 \)

\[
\begin{align*}
    b &= a \\
    b &= a \\ 
    b &= 3 \quad \# \text{ or multiply} (b, 3, b) \\
    b &= 1 \quad \# \text{ or subtract} (b, 1, b)
\end{align*}
\]

- In-place operations:

\[
\begin{align*}
    a &= 3.0 \quad \# \text{ multiply a’s elements by 3} \\
    a &= 1.0 \quad \# \text{ subtract 1 from each element} \\
    a &= 3.0 \quad \# \text{ divide each element by 3} \\
    a &= 1.0 \quad \# \text{ add 1 to each element} \\
    a &= 2.0 \quad \# \text{ square all elements}
\end{align*}
\]
Timing Array Basic Operations

We want to perform the array operation

\[ b = 3a + 1 \]

in three different ways: (1) looping over the entries of the array, (2) using Numpy array operation, and (3) using in-place arithmetic.

<table>
<thead>
<tr>
<th>a.size</th>
<th>Loop</th>
<th>Numpy</th>
<th>In-Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>(10^7)</td>
<td>20.13</td>
<td>0.09</td>
<td>0.06</td>
</tr>
<tr>
<td>(10^8)</td>
<td>214.77</td>
<td>0.94</td>
<td>0.48</td>
</tr>
</tbody>
</table>
# let b be an array

c = \sin(b)

c = \arcsin(c)

c = \sinh(b)

# same functions for the cos and tan families

c = b^{2.5}  \# power function

c = \log(b)

c = \exp(b)

c = \sqrt{b}
Other Useful Array Operations

# a is an array
a.clip(min=3, max=12)  # clip elements
a.mean(); mean(a)      # mean value
a.var(); var(a)        # variance
a.std(); std(a)        # standard deviation
median(a)
cov(x,y)               # covariance
trapz(a)               # Trapezoidal integration
diff(a)                # finite differences (da/dx)

# more Matlab-like functions:
corrcoeff, cumprod, diag, eig, eye, fliplr,
flipud, max, min,prod, ptp, rot90, squeeze, sum,
svd, tri, tril, triu
```python
>>> a = zeros(4) + 3
>>> a
array([ 3., 3., 3., 3.])  # float data
>>> a.item(2)           # more efficient than a[2]
3.0
>>> a.itemset(3,-4.5)   # more efficient than a[3]=-4.5
>>> a
array([ 3. , 3. , 3. , -4.5])
>>> a.shape = (2,2)
>>> a
array([[ 3. , 3. ],
       [ 3. , -4.5]])
>>> a.ravel()           # from multi-dim to one-dim
array([ 3. , 3. , 3. , -4.5])
>>> a.ndim             # no of dimensions
2
>>> len(a.shape)       # no of dimensions
2
>>> rank(a)            # no of dimensions
2
>>> a.size             # total no of elements
4
>>> b = a.astype(int)  # change data type
>>> b
array([3, 3, 3, 3])
```
Complex Number Computing

```python
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error

>>> from numpy import sqrt
>>> sqrt(-1)
Warning: invalid value encountered in sqrt
nan
>>> from cmath import sqrt  # complex math functions
>>> sqrt(-1)
1j
>>> sqrt(4)  # cmath functions always return complex...
(2+0j)
>>> from numpy.lib.scimath import sqrt
>>> sqrt(4)
2.0  # real when possible
>>> sqrt(-1)
1j  # otherwise complex
```
# Goal: compute roots of a parabola, return real when possible, otherwise complex

def roots(a, b, c):
    # compute roots of $a \cdot x^2 + b \cdot x + c = 0$
    from numpy.lib.scimath import sqrt
    q = sqrt(b**2 - 4*a*c)  # q is real or complex
    r1 = (-b + q)/(2*a)
    r2 = (-b - q)/(2*a)
    return r1, r2

>> a = 1; b = 2; c = 100
>> roots(a, b, c)  # complex roots
((-1+9.94987437107j), (-1-9.94987437107j))

>> a = 1; b = 4; c = 1
>>> roots(a, b, c)  # real roots
(-0.267949192431, -3.73205080757)
Array Type and Data Type

```python
>>> import numpy
>>> a = numpy.zeros(5)
>>> type(a)
<type 'numpy.ndarray'>
>>> isinstance(a, ndarray)  # is a of type ndarray?
True
>>> a.dtype  # data (element) type object
dtype('float64')
>>> a.dtype.name
'float64'
>>> a.dtype.char  # character code
'd'
>>> a.dtype.itemsize  # no of bytes per array element
8
>>> b = zeros(6, float32)
>>> a.dtype == b.dtype  # do a and b have the same data type?
False
>>> c = zeros(2, float)
>>> a.dtype == c.dtype
True
```
Concept of Vectorization

- Loops over an array run slowly
- Vectorization = replace explicit loops by functions calls such that the whole loop is implemented in C (or Fortran)
- Explicit loops:
  
  ```python
  r = zeros(x.shape, x.dtype)
  for i in xrange(x.size):
      r[i] = sin(x[i])
  ```

- Vectorized version:
  
  ```python
  r = sin(x)
  ```

- Arithmetic expressions work for both scalars and arrays
- Many fundamental functions work for scalars and arrays
- Ex: $x^{**2} + \text{abs}(x)$ works for $x$ scalar or array
Vectorization using Functions

A mathematical function written for scalar arguments can (normally) take a array arguments:

```python
>>> def f(x):
...     return x**2 + sinh(x)*exp(-x) + 1
...     
...     # scalar argument:
>>> x = 2
>>> f(x)
5.4908421805556333

>>> # array argument:
>>> y = array([2, -1, 0, 1.5])
>>> f(y)
array([ 5.49084218, -1.19452805, 1. , 3.72510647])
```
Vectorization of Functions with if-tests

Consider a function with an if test:

```python
def somefunc(x):
    if x < 0:
        return 0
    else:
        return sin(x)
# or
def somefunc(x): return 0 if x < 0 else sin(x)
```

This function works with a scalar x but not an array.

Problem: x < 0 results in a boolean array, not a boolean value that can be used in the if test.

```python
>>> x = linspace(-1, 1, 3); print x[-1. 0. 1.]
>>> y = x < 0
>>> y
array([ True, False, False], dtype=bool)
>>> 'ok' if y else 'not ok'  # test of y in scalar boolean context
... ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```
Vectorization of Functions with if-tests

- Simplest remedy: call NumPy's vectorize function to allow array arguments to a function:

  ```python
  >>> somefuncv = vectorize(somefunc, otypes='d')
  >>> # test:
  >>> x = linspace(-1, 1, 3); print x[-1. 0. 1.]  
  >>> somefuncv(x)
  array([ 0.   , 0.   , 0.84147098])
  ```

  Note: The data type must be specified as a character

- The speed of somefuncv is unfortunately quite slow

- A better solution, using where:

  ```python
  def somefunc_NumPy2(x):
      x1 = zeros(x.size, float)
      x2 = sin(x)
      return where(x < 0, x1, x2)
  ```
General Vectorization with if-else Tests

```python
def f(x):  # scalar x
    if condition:
        x = <expression1>
    else:
        x = <expression2>
    return x

def f_vectorized(x):  # scalar or array x
    x1 = <expression1>
    x2 = <expression2>
    return where(condition, x1, x2)
```
Consider for instance a recursion scheme which arises from a one-dimensional diffusion equation.

Straightforward (slow) Python implementation:

```python
n = size(u)-1
for i in xrange(1,n,1):
    u_new[i] = beta*u[i-1] + (1-2*beta)*u[i] + beta*u[i+1]
```

Slices enable us to vectorize the expression:

```python
```
Matrix Objects-1

- NumPy has an array type, matrix, much like Matlab’s array type
  ```python
  >>> x1 = array([1, 2, 3], float)
  >>> x2 = matrix(x)  # or just mat(x)
  >>> x2
  matrix([[ 1., 2., 3.]])
  >>> x3 = mat(x).transpose()  # column vector
  >>> x3
  matrix([[ 1.],
           [ 2.],
           [ 3.]])
  >>> type(x3)
  <class 'numpy.core.defmatrix.matrix'>
  >>> isinstance(x3, matrix)
  True
  ```

- Only 1- and 2-dimensional arrays can be matrix
For matrix objects, the * operator means matrix-matrix or matrix-vector multiplication (not elementwise multiplication)

```python
>>> A = eye(3)  # identity matrix
>>> A = mat(A)  # turn array to matrix
>>> A
matrix([[ 1., 0., 0.],
         [ 0., 1., 0.],
         [ 0., 0., 1.]])
>>> y2 = x2*A  # vector-matrix product
>>> y2
matrix([[ 1., 2., 3.]])
>>> y3 = A*x3  # matrix-vector product
>>> y3
matrix([[ 1.],
         [ 2.],
         [ 3.]])
```
Example 1

```python
a = array([[1,2],[3,4]])
a
m = mat(a)
m
a[0]
m[0]
a*a
m*m
dot(a, a)
```
Example 2

```python
x = array([1, 0, 2, -1, 0, 0, 8])
indices = x.nonzero()
indices
x[indices]
indices = (x > -1).nonzero()
x[indices]
```
Example 3

```python
a = array([1, 2, 3])
a.prod()
prod(a)

b = array([[1, 2, 3], [4, 5, 6]])
b.prod(dtype=float)
b.prod(axis=0)
b.prod(axis=1)
```
Overview

- In NumPy, operation are elementwise by default
- There is a matrix type for linear algebra (subclass of array)
- Indexing start at 0 in NumPy
- Using Python with NumPy gives more programming power
- Function definition in Matlab have many restriction
- NumPy/SciPy is free but still widely used
- Matlab have lots of 'toolboxes' for specific task (lot less in NumPy/SciPy)
- There are many packages for plotting in Python that are as good as Matlab
## Matlab/NumPy Equivalence-1

<table>
<thead>
<tr>
<th>Matlab</th>
<th>NumPy</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = [1 2 3; 4 5 6]</code></td>
<td><code>a = array([[1.,2.,3.],[4.,5.,6.]])</code></td>
</tr>
<tr>
<td><code>a(end)</code></td>
<td><code>a[-1]</code></td>
</tr>
<tr>
<td><code>a(2,5)</code></td>
<td><code>a[1,4]</code></td>
</tr>
<tr>
<td><code>a(2,:)</code></td>
<td><code>a[1]</code> or <code>a[1,:]</code></td>
</tr>
<tr>
<td><code>a(1:5,:)</code></td>
<td><code>a[0:5]</code> or <code>a[:5]</code> or <code>a[0:5,:]</code></td>
</tr>
<tr>
<td><code>a(end-4:end,:)</code></td>
<td><code>a[-5:]</code></td>
</tr>
<tr>
<td><code>a(1:3,5:9)</code></td>
<td><code>a[0:3][:,4:9]</code></td>
</tr>
<tr>
<td><code>a(1:2:end,:)</code></td>
<td><code>a[::2,:]</code></td>
</tr>
<tr>
<td><code>a(end:-1:1,:)</code> or <code>flipud(a)</code></td>
<td><code>a[:::-1,:]</code></td>
</tr>
<tr>
<td><code>a.'</code></td>
<td><code>a.transpose()</code> or <code>a.T</code></td>
</tr>
<tr>
<td><code>a'</code></td>
<td><code>a.conj().transpose()</code> or <code>a.conj().T</code></td>
</tr>
<tr>
<td><code>a * b</code></td>
<td><code>dot(a,b)</code></td>
</tr>
<tr>
<td><code>a .* b</code></td>
<td><code>a * b</code></td>
</tr>
<tr>
<td><code>a./b</code></td>
<td><code>a/b</code></td>
</tr>
</tbody>
</table>
## Matlab/NumPy Equivalence-2

<table>
<thead>
<tr>
<th>Matlab</th>
<th>NumPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>diag(a)</td>
<td>g(a) or a.diagonal()</td>
</tr>
<tr>
<td>diag(a,0)</td>
<td>diag(a,0) or a.diagonal(0)</td>
</tr>
<tr>
<td>rand(3,4)</td>
<td>random.rand(3,4)</td>
</tr>
<tr>
<td>linspace(1,3,4)</td>
<td>inspace(1,3,4)</td>
</tr>
<tr>
<td>([x, y] = meshgrid(0 : 8, 0 : 5))</td>
<td>grid[0 : 9., 0 : 6.]</td>
</tr>
<tr>
<td>repmat(a, m, n)</td>
<td>tile(a, (m, n))</td>
</tr>
<tr>
<td>([a \ b])</td>
<td>concatenate((a,b),1) or hstack((a,b)) or c [a, b[a; b]]</td>
</tr>
<tr>
<td>([a; b])</td>
<td>concatenate((a,b)) or vstack((a,b)) or r [a, b]</td>
</tr>
<tr>
<td>max(max(a))</td>
<td>a.max()</td>
</tr>
<tr>
<td>max(a)</td>
<td>a.max(0)</td>
</tr>
<tr>
<td>max(a,[],2)</td>
<td>a.max(1)</td>
</tr>
<tr>
<td>max(a,b)</td>
<td>re(a&gt;b, a, b)</td>
</tr>
<tr>
<td>norm(v)</td>
<td>sqrt(dot(v,v)) or linalg.norm(v)</td>
</tr>
</tbody>
</table>
Matlab/NumPy Equivalence-3

<table>
<thead>
<tr>
<th>Matlab</th>
<th>NumPy</th>
</tr>
</thead>
<tbody>
<tr>
<td>inv(a)</td>
<td>linalg.inv(a)</td>
</tr>
<tr>
<td>pinv(a)</td>
<td>linalg.pinv(a)</td>
</tr>
<tr>
<td>a\b</td>
<td>linalg.solve(a,b)</td>
</tr>
<tr>
<td>b/a</td>
<td>Solve $a^T \times b^T = c^T$ instead</td>
</tr>
<tr>
<td>[U, S, V]=svd(a)</td>
<td>(U, S, V) = linalg.svd(a)</td>
</tr>
<tr>
<td>chol(a)</td>
<td>linalg.cholesky(a)</td>
</tr>
<tr>
<td>[V, D]=eig(a)</td>
<td>linalg.eig(a)</td>
</tr>
<tr>
<td>[Q, R, P]=qr(a,0)</td>
<td>Sci.linalg.qr(a)</td>
</tr>
<tr>
<td>[L, U, P]=lu(a)</td>
<td>Sci.linalg.lu(a) or Sci.linalg.lu_factor(a)</td>
</tr>
<tr>
<td>conjgrad</td>
<td>Sci.linalg.cg</td>
</tr>
<tr>
<td>fft(a)</td>
<td>fft(a)</td>
</tr>
<tr>
<td>ifft(a)</td>
<td>ifft(a)</td>
</tr>
<tr>
<td>sort(a)</td>
<td>sort(a) or a.sort()</td>
</tr>
<tr>
<td>sortrows(a,i)</td>
<td>a[arange(a[::,0], i)]</td>
</tr>
<tr>
<td>Matlab</td>
<td>NumPy</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>a.3</td>
<td>a * 3</td>
</tr>
<tr>
<td>find(a&gt;0.5)</td>
<td>where(a&gt;0.5)</td>
</tr>
<tr>
<td>a(a&lt;0.5)=0</td>
<td>a[a&lt;0.5]=0</td>
</tr>
<tr>
<td>a(:) = 3</td>
<td>a[:] = 3</td>
</tr>
<tr>
<td>y=x</td>
<td>y = x.copy()</td>
</tr>
<tr>
<td>y=x(2,:)</td>
<td>y = x[2,:].copy()</td>
</tr>
<tr>
<td>y=x(:)</td>
<td>y = x.flatten(1)</td>
</tr>
<tr>
<td>1:10</td>
<td>arange(1.,11.) or r[1.:11.]</td>
</tr>
<tr>
<td>0:9</td>
<td>arange(10.) or r[:10.]</td>
</tr>
<tr>
<td>zeros(3,4)</td>
<td>zeros((3,4))</td>
</tr>
<tr>
<td>zeros(3,4,5)</td>
<td>zeros((3,4,5))</td>
</tr>
<tr>
<td>ones(3,4)</td>
<td>ones((3,4))</td>
</tr>
<tr>
<td>eye(3)</td>
<td>eye(3)</td>
</tr>
</tbody>
</table>
Sample Matrix Multiplication

Given two $n \times n$ matrices $A$ and $B$, we want to compute:

$$C = A \times B$$

$A$ and $B$ have randomly generated entries.

Check the files:

matMultiPython.py # Multiply two matrices using do loops
matMultiNumpy.py  # Multiply two matrices using NumPy function
## Timing Results of the Matrix Multiplication

<table>
<thead>
<tr>
<th></th>
<th>n = 1000</th>
<th>n = 1200</th>
<th>n = 1500</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Python</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NumPy</td>
<td>8.14</td>
<td>14.04</td>
<td>28.15</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.023</td>
<td>0.048</td>
<td>0.057</td>
</tr>
<tr>
<td>gfortran (matmult)</td>
<td>0.604</td>
<td>1.212</td>
<td>3.000</td>
</tr>
<tr>
<td>gfortran with Blas</td>
<td>0.180</td>
<td>0.300</td>
<td>0.596</td>
</tr>
<tr>
<td>gcc</td>
<td>0.60</td>
<td>1.110</td>
<td>2.940</td>
</tr>
<tr>
<td>g++</td>
<td>1.351</td>
<td>2.382</td>
<td>4.928</td>
</tr>
</tbody>
</table>

For additional information, go to:

**Comparing Python, NumPy, Matlab, Fortran, etc.**

[https://modelingguru.nasa.gov/docs/D0C-1762](https://modelingguru.nasa.gov/docs/D0C-1762)
Random Numbers

- Drawing scalar random numbers:
  ```python
  import random
  random.seed(2198)  # control the seed
  print 'uniform random number on (0,1):', random.random()
  print 'uniform random number on (-1,1):', random.uniform(-1,1)
  print 'Normal(0,1) random number:', random.gauss(0,1)
  ```

- Vectorized drawing of random numbers (arrays):
  ```python
  from numpy import random
  random.seed(12)  # set seed
  u = random.random(n)  # n uniform numbers on (0,1)
  u = random.uniform(-1, 1, n)  # n uniform numbers on (-1,1)
  u = random.normal(m, s, n)  # n numbers from N(m,s)
  ```

- Note that both modules have the name random! A remedy:
  ```python
  import random as random_number  # rename random for scalars
  from numpy import *  # random is now numpy.random
  ```
Basic Linear Algebra

NumPy contains the `linalg` module for:

- Solving linear systems
- Computing the determinant of a matrix
- Computing the inverse of a matrix
- Computing eigenvalues and eigenvectors of a matrix
- Solving least-squares problems
- Computing the singular value decomposition of a matrix
- Computing the Cholesky decomposition of a matrix
numpy.linalg

cholesky(A)    # Cholesky decomposition
qr(a[,mode])   # Compute the qr factorization of a matrix
svd(a[,full_matrices,compute_uv]) # Singular Value Decomposition
eig(A)         # Compute the eigenvalues and right
eigvals(A)     # eigenvectors of a square array
det(A)         # Compute the determinant of a matrix
matrix_power(M,n) # Raise a square matrix to the (integer) power n
solve(A,b)     # Solve a linear matrix equation,
solve(A,b)     # or system of linear scalar equations.
inv(A)         # Compute the (multiplicative) inverse of a matrix
Sample Linear Algebra Session

```python
b = dot(A, x)  # matrix vector product
y = linalg.solve(A, b)  # solve A*y = b
if allclose(x, y, atol=1.0E-12, rtol=1.0E-12):
    print 'correct solution!'

d = linalg.det(A)
B = linalg.inv(A)

R = dot(A, B) - eye(n)  # residual
R_norm = linalg.norm(R)  # Frobenius norm of matrix R
print 'Residual R = A*A- inv - I:', R_norm

A_eigenvalues = linalg.eigvals(A)  # eigenvalues only
A_eigenvalues, A_eigenvectors = linalg.eig(A)

for e, v in zip(A_eigenvalues, A_eigenvectors):
    print 'eigenvalue %g has corresponding vector
```
We want to find the numerical solution of the 2D Laplace equation:

$$u_{xx} + u_{yy} = 0.$$  

We use the Jacobi iterative solver.
Finite Difference Schemes

We use two schemes:

\[ u_{i,j} = \frac{1}{4} (u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}) \]  \hspace{1cm} (1)

\[ u_{i,j} = \frac{1}{20} (4(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}) + 
                      u_{i-1,j-1} + u_{i-1,j+1} + u_{i+1,j-1} + u_{i+1,j+1}) \]  \hspace{1cm} (2)
## Timing Results with Scheme 1

<table>
<thead>
<tr>
<th></th>
<th>n = 50</th>
<th>n = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>22.818</td>
<td>350.912</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.2706</td>
<td>2.61626</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.5016</td>
<td>1.88999</td>
</tr>
<tr>
<td>f2py</td>
<td>0.0327</td>
<td>0.50701</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.0280</td>
<td>0.34800</td>
</tr>
</tbody>
</table>
## Timing Results with Scheme 2

<table>
<thead>
<tr>
<th></th>
<th>n = 50</th>
<th>n = 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td>32.771</td>
<td>547.349</td>
</tr>
<tr>
<td>NumPy</td>
<td>0.4109</td>
<td>4.24026</td>
</tr>
<tr>
<td>Matlab</td>
<td>0.4021</td>
<td>3.06696</td>
</tr>
<tr>
<td>f2py</td>
<td>0.0842</td>
<td>1.34566</td>
</tr>
<tr>
<td>Fortran</td>
<td>0.0400</td>
<td>0.58000</td>
</tr>
</tbody>
</table>
SciPy
Useful Links for SciPy

- **How to Think Like a Computer Scientist: Learning with Python**, 2nd Edition, Jeffrey Elkner, Allen B. Downey, and Chris Meyers
  

- **Dive Into Python: Python from Novice to Pro**, Mark Pilgrim
  
  [http://diveintopython.org](http://diveintopython.org)
What is SciPy?

- Collection of mathematical algorithms and convenience functions built on the Numeric extension for Python
- Adds significant power to the interactive Python session
- Can become a data-processing and system-prototyping environment
What SciPy Can Do

- **stats**: Statistical Functions
- **signal**: Signal Processing Tools
- **linalg**: Linear Algebra Tools
- **linsolve**: Linear Solvers
- **sparse**: Sparse Matrix
- **fftpack**: Discrete Fourier Transform Algorithms
- **ndimage**: n-dimensional Image Package
- **io**: Data Input and Output
- **integrate**: Integration Routines
- **interpolate**: Interpolation Tools
The SciPy library is built to work with NumPy arrays

Depends on NumPy for array manipulations
Loading SciPy

- Loading the SciPy module:
  ```python
  import scipy
  ```

- The following command will import all SciPy functions:
  ```python
  from scipy import *
  ```

- Help on SciPy:
  ```python
  scipy.info(scipy)
  help(scipy)
  ```
Two general interpolation facilities:

1. One class that performs 1D linear interpolation (interp1d)
2. Another (based on FITPACK) which provides 1D and 2D cubic-spline interpolations (splrep, splev, bisplrep, bisplev)
scipy.interpolate Syntax

```python
f = interp1d(x, y)  # 1D linear interpolation

tck = splrep(x, y, k=n)  # B-spline representation of 1-D
ynew = splev(xnew, tck, der=n)  # evaluate the value of the
# polynomial and its derivative

znew = bisplev(xnew, ynew, tck)  # evaluate the polynomial
# of the surface
```

J. Kouatchou and H. Oloso (SSSO)
NumPy and SciPy
March 11, 2013 68 / 100
1D Linear Interpolation

![Graph showing 1D linear interpolation with data points and a green line connecting them.](image)
1D Cubic Spline Interpolation

Cubic-spline interpolation

Integral estimation from spline

Derivative estimation from spline

Spline of parametrically-defined curve

- Linear
- Cubic Spline
- True
2D Cubic Spline Interpolation

Sparsely sampled function.

Interpolated function.
scipy.optimize

A collection of general-purpose optimization routines. We can mention:

- **fminbound**: Bounded minimization for scalar functions
- **fsolve**: Find the roots of a function
- **fmin**: Minimize a function using the downhill simplex algorithm
- **fixed_point**: Find the point where \( \text{func}(x) = x \)
- **leastsq**: Minimize the sum of squares of a set of equations
We want to plot a set of Bessel functions together with their maximum values.

```python
x = arange(0,10,0.01)

for k in arange(0.5,5.5):
    y = special.jv(k,x)
    plt.plot(x,y)

f = lambda x: -special.jv(k,x)

x_max = optimize.fminbound(f,0,6)

plt.plot([x_max], [special.jv(k,x_max)], 'ro')
```
Plots of Bessel Functions

Different Bessel functions and their local maxima
Least Square Approximation

\texttt{leastsq}(\texttt{efunc}, x0, \texttt{args}=(x, y))
Root Finding with fsolve

Assume that we want to solve the equations:

\[ x + 2 \cos(x) = 0 \]

\[ \begin{cases} 
  x_0 \cos(x_1) &= 4 \\
  x_0 x_1 - x_1 &= 5 
\end{cases} \]
Code with fsolve

```python
from scipy.optimize import fsolve

def func(x):
    return x + 2*scipy.cos(x)

def func2(x):
    out = [x[0]*scipy.cos(x[1]) - 4]
    out.append(x[1]*x[0] - x[1] -5)
    return out

x0 = fsolve(func, 0.3)
print(x0)

x02 = fsolve(func2, [1, 1])
print(x02)
```
Minimization Problem

Assume that we want to minimize the function:

\[ f(x) = \sum_{i=1}^{N-1} 100(x_i - x_{i-1}^2)^2 + (1 - x_{i-1})^2 \]
from scipy.optimize import fmin

def rosen(x):
    """The Rosenbrock function""
    return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + \
                (1-x[:-1])**2.0)

x0 = [1.3, 0.7, 0.8, 1.9, 1.2]
xopt = fmin(rosen, x0, xtol=1e-8)
from scipy.optimize import fixed_point

def func(x, c1, c2):
    return sqrt(c1/(x+c2))

c1 = array([10, 12.])
c2 = array([3, 5.])
print fixed_point(func, [1.2, 1.3], args=(c1,c2))
scipy.integrate

quad : General purpose integration.
dblquad : General purpose double integration.
tplquad : General purpose triple integration.
fixed_quad : Integrate using Gaussian quadrature of order n.
quadrature : Integrate with given tolerance using Gaussian quadrature.
romberg : Integrate func using Romberg integration.
trapz : Use trapezoidal rule to compute integral from samples.
cumtrapz : Use trapezoidal rule to cumulatively compute integral.
simps : Use Simpson’s rule to compute integral from samples.
romb : Use Romberg Integration to compute integral from \((2^{**k} + 1)\) evenly-spaced samples.
odeint : General integration of ordinary differential equations.
ode : Integrate ODE using VODE and ZVODE routines.
Example of ODE

Assume that we want to solve the equation:

\[ x''(t) + \mu x'(t)(x^2(t) - 1) + x(t) = 0 \]

It can be transformed into:

\[
\begin{align*}
    x' &= y \\
    y' &= -x + \mu y(1 - x^2)
\end{align*}
\]
import matplotlib.pyplot as plt
import scipy
from scipy import integrate

def f_1(y,t):
    return [y[1], -y[0] - 10*y[1]*(y[0]**2 - 1)]

def j_1(y,t):
    return [[0, 1.0], [-2.0*10*y[0]*y[1] - 1.0, -10*(y[0]*y[0] - 1.0)]]

x = scipy.arange(0, 100, .1)

y = integrate.odeint(f_1, [1, 0], x, Dfun=j_1)

p = [((x[i], y[i][0])) for i in range(len(x))]

plt.plot(p)
plt.show()
scipy.stats

Provides tools for statistical analysis:

- More than 84 continuous distributions.
- More than 12 discrete distributions
- Tools for manipulating them:
  - Statistical functions
  - Statistical tests
  - Statistical models
Syntax

probability density function
   \texttt{generic.pdf(x,<shape(s)>,loc=0,scale=1)}

cumulative density function
   \texttt{generic.cdf(x,<shape(s)>,loc=0,scale=1)}

percent point function (inverse of cdf --- percentiles)
   \texttt{generic.ppf(q,<shape(s)>,loc=0,scale=1)}

random variates
   \texttt{generic.rvs(<shape(s)>,loc=0,scale=1,size=1)}

mean('m'), variance('v'), skew('s'), and/or kurtosis('k')
   \texttt{generic.stats(<shape(s)>,loc=0,scale=1,moments='mv')}
Sample Code: Distribution

```python
from scipy import stats

x = np.linspace(-3.0, 3.0, 15)
q = np.linspace(0.0, 1.0, 15)

stats.norm.pdf(x, loc=0.0, scale=1.0)
stats.norm.cdf(x, loc=0.0, scale=1.0)
stats.norm.ppf(q, loc=0.0, scale=1.0)
stats.norm.stats(loc=0.0, scale=1.0)
stats.norm.rvs(loc=0.0, scale=1.0, size=15)
```
Plots of Distributions

PDF

CDF
Summary Statistics

```python
x = stats.norm.rvs(size=1000)
x.mean(); np.mean(x)
x.std(); np.std(x)
x.var(); np.var(x)
np.median(x)
stats.mode(stats.geom.rvs(0.1, size=1000))
```
Examples with scipy.stats

- Linear Regression: linearRegression.py
- Example of distribution: distributionExample.py
- Computation of mean, std: statEstimatorsSample.py
Sample Linear Regression Plot
Discrete Fourier Transform Algorithms

- `fft`, `ifft`, `fft2`, `ifft2`, `fftn`, `ifftn`
- `fftshift`, `ifftshift`, `fftfreq`
```python
import matplotlib.pyplot as plt
from scipy import *
from scipy import *
from scipy.fftpack import fftshift, fftfreq

x = r_[0:1:100j]
y = 2*sin(2*pi*10*x) + 3*cos(2*pi*20*x)

Y02 = fft(y, 1024)
w = fftfreq(1024)
plt.plot(w, abs(Y02))
```
Sample FFT

FFT Demo: 1024 point FFT

FFT Demo: 100 point FFT

Magnitude

Frequency

Magnitude

Frequency
Sample Band-Pass Filtering

(A) Original Signal

(B) Electrical Noise Sources (3 Sine Waves)

(C) Electrical Noise (3 sine waves added together)

(D) Static (random noise)

(E) Signal + Static

(F) Recording (Signal + Static + Electrical Noise)

(G) FFT of Recording

(H) Low-Pass FFT

(I) Inverse FFT

(J) Signal vs. iFFT

(K) Normalized Signal vs. iFFT

(L) Difference / Error
scipy.signal

Provides functions for:

- Convolution
- B-Splines
- Filtering & Filter Design
- Linear Systems
- Window Functions
- Wavelets
Sample Convolution Code

```python
from scipy import *
from scipy import signal

n = 64
x = linspace(0, n-1, n)

y01 = hamming(n)
y02 = hanning(n)

z01 = signal.convolve(y01, y02, mode='same')
```
Sample Convolution Plot

![Convolution Plots](Convolution_Plots.png)
Windowing Function

Functions to generate the following types of windows:

- `boxcar(M, sym = 1)`  # M-point boxcar window
- `triang(M, sym = 1)`
- `blackman(M, sym = 1)`
- `hamming(M, sym = 1)`
- `kaiser(M, beta, sym = 1)`  # Return a Kaiser window of length M
  # with shape parameter beta
- `gaussian(M, std, sym = 1)`  # Return a Gaussian window of length M
  # with standard-deviation std
Sample Convolution Plot

Windowing Functions

- hamming
- hanning
- kaiser


Sandro Tosi, Matplotlib for Python Developers, 2009.