A SHORT INTRODUCTION TO PYTHON

Numpy, Scipy

Slides adapted from a presentation by Shai Fine
Essential Python Extensions

• The following packages extend Python with extra features
  • NumPy – Fast, multidimensional arrays
  • SciPy – Libraries of reliable, tested scientific functions

• Additional packages for Data Science (not covered today)
  • Wide range of learning algorithms (scikit-learn)
  • Tools for data manipulation (Pandas)
  • Plotting tools (Matplotlib)
  • Direct connection to R (rpy2)
PyLab

Sometimes the union of the 5 packages is called pylab
Helpful Sites

SCIPY DOCUMENTATION PAGE

http://www.scipy.org/Documentation

Numpy Example List With Doc

http://www.scipy.org/Numpy_Example_List_With_Doc

```python
numpy.apply_along_axis(func1d, axis, arr, *args)
```

Execute `func1d(arr[i], *args)` where `func1d` takes 1-D arrays and `arr` is an N-d array. `i` varies so as to apply the function along the given axis for each 1-d subarray in `arr`.

Example:

```python
>>> from numpy import *
>>> def myfunc(a):
...     return (a[0]+a[-1])/2
...     ...
>>> b = array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> apply_along_axis(myfunc, 0, b)  # apply myfunc to each column
array([4.5, 5.5])
>>> apply_along_axis(myfunc, 1, b)  # apply myfunc to each row
array([2.5, 5.5])
```
What is NumPy?

- NumPy is the fundamental package for scientific computing with Python

- NumPy provides a fast built-in object, `ndarray`, which is a multi-dimensional array of a homogeneous data-type that can be manipulated in a vectorized form
  - Numpy Offers Matlab-ish capabilities within Python

- NumPy can also be used as an efficient multi-dimensional container of generic data
  - This allows NumPy to seamlessly and speedily integrate with a wide variety of databases


- Chronology
  - Initially developed by Travis Oliphant
  - NumPy 1.0 released October, 2006
  - ~20K downloads/month from Sourceforge
    - Doesn’t count distributions that that include NumPy
  - NumPy is at the core of nearly every scientific Python
# Overview of NumPy

## N-Dimensional ARRAY (NDARRAY)
- A NumPy array is a homogeneous collection of "items" of the same "data-type" (dtype)
  - Can be 1-dim or N-dims
- Element of the array can be C-structure or simple data-type
- Fast algorithms on machine data-types (int, float, etc.)

## Universal Functions (UFUNC)
- Functions that operate element-by-element and return result
- Fast-loops registered for each fundamental data-type
  - \( \sin(x) = [\sin(x_i), i = 0 \ldots N] \)
  - \( x + y = [x_i + y_i, i = 0 \ldots N] \)
Arrays in Python

- Python doesn’t include a built-in multi-dimensional array

- Lists ok for storing small amounts of one-dimensional data

```python
>>> a = [1,3,5,7,9]
>>> print(a[2:4])
[5, 7]
>>> b = [[1, 3, 5, 7, 9], [2, 4, 6, 8, 10]]
>>> print(b[0])
[1, 3, 5, 7, 9]
>>> print(b[1][2:4])
[6, 8]
```

- But, can’t use directly with arithmetical operators (+, -, *, /, …)

```python
>>> a = [1,3,5,7,9]
>>> b = [3,5,6,7,9]
>>> c = a + b
>>> print c
[1, 3, 5, 7, 9, 3, 5, 6, 7, 9]
```

- Need efficient arrays with arithmetic and better multidimensional tools
Introducing NumPy Arrays

Simple Array Creation

```python
>>> a = array([0, 1, 2, 3])
>>> a
array([0, 1, 2, 3])
```

Checking the Type

```python
>>> type(a)
<type 'array'>
```

Numeric ‘Type’ of Elements

```python
>>> a.dtype
dtype('int32')
```

Number of Dimensions

```python
>>> a.ndim
1
```

Array Shape

```python
# shape returns a tuple
# listing the length of the
# array along each dimension.
>>> a.shape
(4,)
```

Array Size

```python
# size reports the entire
# number of elements in an
# array.
>>> a.size
4
```

```python
>>> size(a)
4
```
Introducing NumPy Arrays

**ARRAY COPY**

# create a copy of the array
>>> b = a.copy()

>>> b
array([0, 1, 2, 3])

**CONVERSION TO LIST**

# convert a numpy array to a python list
>>> a.tolist()
[0, 1, 2, 3]

# For 1D arrays, list also works equivalently, but is slower
>>> list(a)
[0, 1, 2, 3]
Setting Array Elements

**ARRAY INDEXING**

```python
>>> a[0]
0
>>> a[0] = 10
>>> a
[10, 1, 2, 3]
```

**FILL**

# set all values in an array.
```python
>>> a.fill(0)
>>> a
[0, 0, 0, 0]
```

# This also works, but may
# be slower
```python
>>> a[:] = 1
>>> a
[1, 1, 1, 1]
```

**BEWARE OF TYPE COERSION**

```python
>>> a.dtype
dtype('int32')

# assigning a float to
# an int32 array will
# truncate decimal part
>>> a[0] = 10.6
>>> a
[10, 1, 2, 3]

# fill has the same behavior
>>> a.fill(-4.8)
>>> a
[-4, -4, -4, -4]
```
Multi-Dimensional Arrays (ndarray)

MULTI-DIMENSIONAL ARRAYS

>>> a = array([[ 0, 1, 2, 3],
              [10,11,12,13]])

>>> a
array([[ 0, 1, 2, 3],
        [10,11,12,13]])

(ROWS,COLUMNS)

>>> a.shape
(2, 4)

>>> shape(a)
(2, 4)

ELEMENT COUNT

>>> a.size
8

>>> size(a)
8

NUMBER OF DIMENSIONS

>>> a.ndims
2

GET/SET ELEMENTS

>>> a[1,3]
array([ 10, 11, 12, -1])

>>> a[1,3] = -1

>>> a
array([[ 0, 1, 2, 3],
        [10,11,12,-1]])

ADDRESS FIRST ROW USING SINGLE INDEX

>>> a[1]
array([10, 11, 12, -1])
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 Are References

Slices are references to memory in original array. Changing values in a slice also changes the original array.

```python
>>> a = array((0,1,2,3,4))

# create a slice containing only the last element of a
>>> b = a[2:4]
>>> b[0] = 10

# changing b changed a!
>>> a
array([ 1,  2, 10,  3,  4])
```
# Indexing by position

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

```python
>>> a[3:,0,2,5]
array([[30, 32, 35],
       [40, 42, 45]])
```

# Indexing with Booleans

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

```python
>>> a[mask,2]
array([2,22,52])
```

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

SUM FUNCTION

```python
>>> a = array([[1,2,3],
               [4,5,6]], float)
# Sum defaults to summing all
# *all* array values.
>>> sum(a)
21.

# supply the keyword axis to
# sum along the 0th axis.
>>> sum(a, axis=0)
array([5., 7., 9.])

# supply the keyword axis to
# sum along the last axis.
>>> sum(a, axis=-1)
array([6., 15.])
```

SUM ARRAY METHOD

```python
# The a.sum() defaults to
# summing *all* array values
>>> a.sum()
21.

# Supply an axis argument to
# sum along a specific axis.
>>> a.sum(axis=0)
array([5., 7., 9.])

# product along columns
>>> a.prod(axis=0)
array([4., 10., 18.])

# functional form
>>> prod(a, axis=0)
array([4., 10., 18.])
```
Min/Max

MIN
>>> a = array([2., 3., 0., 1.])
>>> a.min(axis=0)
0.
# use Numpy's amin() instead
# of Python's builtin min()
# for speed operations on
# multi-dimensional arrays.
>>> amin(a, axis=0)
0.

ARGMIN
# Find index of minimum value.
>>> a.argmin(axis=0)
2
# functional form
>>> argmin(a, axis=0)
2

MAX
>>> a = array([2., 1., 0., 3.])
>>> a.max(axis=0)
3.
# functional form
>>> amax(a, axis=0)
3.

ARGMAX
# Find index of maximum value.
>>> a.argmax(axis=0)
1
# functional form
>>> argmax(a, axis=0)
1
# Statistics Array Methods

## MEAN

```python
>>> a = array([[1,2,3],
              [4,5,6]], float)

# mean value of each column
>>> a.mean(axis=0)
array([ 2.5,  3.5,  4.5])
>>> mean(a, axis=0)
array([ 2.5,  3.5,  4.5])
>>> average(a, axis=0)
array([ 2.5,  3.5,  4.5])
```

# average can also calculate
# a weighted average
```python
>>> average(a, weights=[1,2],
          ..., axis=0)
array([ 3.,  4.,  5.])
```
Other Array Methods

CLIP

# Limit values to a range
>>> a = array([[1,2,3],
             [4,5,6]], float)

# Set values < 3 equal to 3.
# Set values > 5 equal to 5.
>>> a.clip(3,5)
array([[ 3.,  3.,  3.],
       [ 4.,  5.,  5.]])

ROUND

# Round values in an array.
# Numpy rounds to even, so 1.5 and 2.5 both round to 2.
>>> a = array([1.35, 2.5, 1.5])
>>> a.round()
array([ 1.,  2.,  2.])

# Round to first decimal place.
>>> a.round(decimals=1)
array([ 1.4,  2.5,  1.5])

POINT TO POINT

# Calculate max - min for
# array along columns
>>> a.ptp(axis=0)
array([ 3.0,  3.0,  3.0])
# max - min for entire array.
>>> a.ptp(axis=None)
5.0
Universal Functions (ufunc)

- ufuncs are objects that rapidly evaluate a function element-by-element over an array.
- Core piece is a 1-d loop written in C that performs the operation over the largest dimension of the array.
- For 1-d arrays it is equivalent to but much faster than list comprehension.

```python
>>> type(np.exp)
<type 'numpy.ufunc'>
>>> x = array([1,2,3,4,5])
>>> print np.exp(x)
[2.71828, 7.38905, 20.08553, 54.59815, 148.41315]
>>> print [math.exp(val) for val in x]
[2.71828, 7.38905, 20.08553, 54.59815, 148.41315]
```

# note: values reformatted to fit slide
Vectorizing Functions

**Example**

```python
# special.sinc already available
# This is just for show.
def sinc(x):
    if x == 0.0:
        return 1.0
    else:
        w = pi*x
        return sin(w) / w
```

```python
>>> sinc([1.3,1.5])
TypeError: can't multiply sequence to non-int
>>> x = r_[-5:5:100j]
>>> y = vsinc(x)
>>> plot(x, y)
```

**SOLUTION**

```python
>>> from numpy import vectorize
>>> vsinc = vectorize(sinc)
>>> vsinc([1.3,1.5])
array([-0.1981, -0.2122])
```
Mathematic Binary Operators *element by element*

- $a + b \rightarrow \text{add}(a,b)$
- $a - b \rightarrow \text{subtract}(a,b)$
- $a \% b \rightarrow \text{remainder}(a,b)$
- $a \times b \rightarrow \text{multiply}(a,b)$
- $a / b \rightarrow \text{divide}(a,b)$
- $a ** b \rightarrow \text{power}(a,b)$

**Multiply by a Scalar**

```python
>>> a = array((1,2))
>>> a*3.
array([3., 6.])
```

**Element by Element Addition**

```python
>>> a = array([1,2])
>>> b = array([3,4])
>>> a + b
array([4, 6])
```

**Addition Using an Operator Function**

```python
>>> add(a,b)
array([4, 6])
```

**In Place Operation**

```python
# Overwrite contents of a.
# Saves array creation overhead
>>> add(a,b,a) # a += b
array([4, 6])
>>> a
array([4, 6])
```
## Comparison and Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td><code>==</code></td>
</tr>
<tr>
<td>Greater Equal</td>
<td><code>&gt;=</code></td>
</tr>
<tr>
<td>Logical And</td>
<td></td>
</tr>
<tr>
<td>Logical Not</td>
<td></td>
</tr>
<tr>
<td>Not Equal</td>
<td><code>!=</code></td>
</tr>
<tr>
<td>Less</td>
<td><code>&lt;</code></td>
</tr>
<tr>
<td>Logical Or</td>
<td></td>
</tr>
<tr>
<td>Logical Xor</td>
<td></td>
</tr>
</tbody>
</table>

### 2D Example

```python
>>> a = array(((1,2,3,4),(2,3,4,5)))
>>> b = array(((1,2,5,4),(1,3,4,5)))
>>> a == b
array([[True, True, False, True],
       [False, True, True, True]])

# functional equivalent
>>> equal(a,b)
array([[True, True, False, True],
       [False, True, True, True]])
```
# Bitwise Operators

Work only on Integer arrays

<table>
<thead>
<tr>
<th>bitwise_and (&amp;)</th>
<th>invert (~)</th>
<th>right_shift(a, shifts)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitwise_or (</td>
<td></td>
<td>)</td>
</tr>
</tbody>
</table>

## Bitwise Examples

```python
>>> a = array((1, 2, 4, 8))
>>> b = array((16, 32, 64, 128))
>>> bitwise_or(a, b)
array([17, 34, 68, 136])
```

```python
# bit inversion
>>> a = array((1, 2, 3, 4), uint8)
>>> invert(a)
array([254, 253, 252, 251], dtype=uint8)
```

```python
# left shift operation
>>> left_shift(a, 3)
array([ 8, 16, 24, 32], dtype=uint8)
```
Matrix

- For two dimensional arrays NumPy defined a special matrix class in module `matrix`
  - Objects are created either with `matrix()` or `mat()` or converted from an array with method `asmatrix()`

```python
>>> import numpy
>>> m = numpy.mat([[1,2],[3,4]])
# or
>>> a = numpy.array([[1,2],[3,4]])
>>> m = numpy.mat(a)
# or
>>> a = numpy.array([[1,2],[3,4]])
>>> m = numpy.asmatrix(a)
```

- Note that the statement `m = mat(a)` creates a copy of array 'a', whereas, method `m = asmatrix(a)` returns a new reference to the same data
Broadcasting

- Multiple inputs must be “broadcasted” to the same shape
  - All arrays are promoted to the same number of dimensions
  - All dimensions of length 1 are expanded as needed

- The *trailing* axes of both arrays must either be 1 or have the same size for broadcasting to occur
Matrix Objects

**STRING CONSTRUCTION**

```python
>>> from numpy import mat
>>> a = mat('[1,3,5;2,5,1;2,3,6]')
>>> a
matrix([[1, 3, 5],
        [2, 5, 1],
        [2, 3, 6]])
```

**TRANSPOSE ATTRIBUTE**

```python
>>> a.T
matrix([[1, 2, 2],
        [3, 5, 3],
        [5, 1, 6]])
```

**INVERTED ATTRIBUTE**

```python
>>> a.I
matrix([[-1.1739, 0.1304, 0.956],
        [0.4347, 0.1739, -0.391],
        [0.1739, -0.130, 0.0434]])
```

**DIAGONAL**

```python
>>> a.diagonal()
matrix([[1, 5, 6]])
>>> a.diagonal(-1)
matrix([[3, 1]])
```

**SOLVE**

```python
>>> b = mat('10;8;3')
>>> a.I*b
matrix([[-7.82608696],
        [4.56521739],
        [0.82608696]])
```

```python
>>> from scipy import linalg
>>> linalg.solve(a,b)
matrix([[-7.82608696],
        [4.56521739],
        [0.82608696]])
```
Matrix vs. Array

• Operator *, dot(), and multiply():
  • Array – '*' means element-wise multiplication; dot() is used for matrix mul.
  • Matrix – '*' means matrix multiplication; multiply() is used for element-wise mul.

• Handling of vectors (rank-1 arrays)
  • Array – the vector shapes 1xN, Nx1 are different things. Operations like A[:,1] return a rank-1 of shape N, not a rank-2 of shape Nx1. Transpose a rank-1 array does nothing.
  • Matrix – rank-1 arrays are always upgraded to 1xN or Nx1 matrices (row or column vectors). A[:,1] returns a rank-2 matrix of shape Nx1.

• Handling of higher-rank arrays (rank > 2)
  • Array objects can have rank > 2
  • Matrix objects always have exactly rank 2

• Convenience attributes
  • Array has a .T attribute, which returns the transpose of the data
  • Matrix has .T, .H, .I, and .A attributes, which return the conjugate transpose, inverse, and asarray() of the matrix, respectively.

• Convenience constructor
  • Array constructor takes (nested) Python sequences as initializers
  • Matrix constructor additionally takes a convenient string initializer
Example – Array and Matrix Calc.

```python
>>> A = np.array([[n+m*10 for n in range(5)] for m in range(5)])
>>> v1 = arange(0, 5)
>>> A
array([[  0,  1,  2,  3,  4],
       [ 10, 11, 12, 13, 14],
       [ 20, 21, 22, 23, 24],
       [ 30, 31, 32, 33, 34],
       [ 40, 41, 42, 43, 44]])
>>> v1
array([0, 1, 2, 3, 4])
>>> np.dot(A, A)
array([[ 300,  310,  320,  330,  340],
       [ 1300, 1360, 1420, 1480, 1540],
       [ 2300, 2410, 2520, 2630, 2740],
       [ 3300, 3460, 3620, 3780, 3940],
       [ 4300, 4510, 4720, 4930, 5140]])
>>> np.dot(A, v1)
array([ 30, 130, 230, 330, 430])
>>> np.dot(v1, v1)
30
```
Examples – Array and Matrix Calc.

# Alternatively, we can cast the array objects to the type matrix. This # changes the behavior of the standard arithmetic operators +, -, * to # use matrix algebra.

```python
>>> M = np.matrix(A)
>>> v = np.matrix(v1).T
>>> v
matrix([[0],
    [1],
    [2],
    [3],
    [4]]))
>>> M*v
matrix([[30],
    [130],
    [230],
    [330],
    [430]])
>>> v.T * v   # inner product
matrix([[30]])
```
Concluding Remarks

• Using arrays wisely
  • Array operations are implemented in C or Fortran
    • Optimized algorithms - i.e. fast!
  • Python loops (i.e. for i in a:…) are much slower
    • Prefer array operations over loops, especially when speed important
    • Also produces shorter code, often more readable

• Matrix or Array, which one to use?
  • Short answer – Use Array
    • They are the standard vector/matrix/tensor type of NumPy. Many NumPy functions return arrays, not matrices
    • There is a clear distinction between element-wise and linear algebra operations
    • You can have standard vectors or row/column vectors if you like
    • The main disadvantage of using the array type is that you will have to use \texttt{dot()} instead of ‘*’ matrix multiplication

• NumPy for Matlab Users
  • \url{http://wiki.scipy.org/NumPy_for_Matlab_Users}
SCIPY
Scientific Python
SciPy Overview

- Available at [www.scipy.org](http://www.scipy.org)

CURRENT PACKAGES

- Special Functions (scipy.special)
- Signal Processing (scipy.signal)
- Image Processing (scipy.ndimage)
- Fourier Transforms (scipy.fftpack)
- Optimization (scipy.optimize)
- Numerical Integration (scipy.integrate)
- Linear Algebra (scipy.linalg)
- Input/Output (scipy.io)
- Statistics (scipy.stats)
- Fast Execution (scipy.weave)
- Clustering Algorithms (scipy.cluster)
- Sparse Matrices (scipy.sparse)
- Interpolation (scipy.interpolate)
- More (e.g. scipy.odr, scipy.maxentropy)
Image Processing

# The famous lena image is packaged with scipy
>>> from scipy import lena, signal
>>> lena = lena().astype(float32)
>>> imshow(lena, cmap=cm.gray)

# Blurring using a median filter
>>> fl = signal.medfilt2d(lena, [15,15])
>>> imshow(fl, cmap=cm.gray)
# Edge detection using Sobel filter

```python
>>> from scipy.ndimage.filters import sobel
>>> imshow(lena)
>>> edges = sobel(lena)
>>> imshow(edges)
```

**NOISY IMAGE**

**FILTERED IMAGE**
Statistics

scipy.stats --- CONTINUOUS DISTRIBUTIONS

over 80 continuous distributions!

METHODS

dfd
cdf
rvs
ppf
stats
10 standard discrete distributions (plus any arbitrary finite RV)

METHODS

pdf  cdf  rvs  ppf  stats
# Sample normal dist. 100 times.

```python
>>> samp = stats.norm.rvs(size=100)
```

```plaintext
>>> x = r_-5:5:100j
# Calculate probability dist.
>>> pdf = stats.norm.pdf(x)
```

# Calculate cummulative Dist.
```plaintext
>>> cdf = stats.norm.cdf(x)
```

# Calculate Percent Point Function
```plaintext
>>> ppf = stats.norm.ppf(x)
```
Statistics

**scipy.stats --- Basic Statistical Calculations on Data**

- `numpy.mean`, `numpy.std`, `numpy.var`, `numpy.cov`
- `stats.skew`, `stats.kurtosis`, `stats.moment`

**scipy.stats.bayes_mvs --- Bayesian mean, variance, and std.**

```python
# Create “frozen” Gamma distribution with a=2.5
>>> grv = stats.gamma(2.5)
>>> grv.stats()  # Theoretical mean and variance
(array(2.5), array(2.5))
# Estimate mean, variance, and std with 95% confidence
>>> vals = grv.rvs(size=100)
>>> stats.bayes_mvs(vals, alpha=0.95)
((2.52887906081, 2.19560839724, 2.86214972438),
 (2.87924964268, 2.17476164549, 3.8070215789),
 (1.69246760584, 1.47470730841, 1.95115903475))
# (expected value and confidence interval for each of
# mean, variance, and standard-deviation)
```
# Sample normal dist. 100 times.

```python
>>> rv1 = stats.norm()
>>> rv2 = stats.norm(2.0, 0.8)
>>> samp = [rv1.rvs(size=100), rv2.rvs(size=100)]
```

# Kernel estimate (smoothed histogram)

```python
>>> apdf = stats.kde.gaussian_kde(samp)
>>> x = linspace(-3, 6, 200)
>>> plot(x, apdf(x), 'r')
```

# Histogram

```python
>>> hist(x, bins=25, normed=True)
```
Linear Algebra

scipy.linalg --- FAST LINEAR ALGEBRA

• Uses ATLAS if available --- very fast

• Low-level access to BLAS and LAPACK routines in modules linalg.fblas, and linalg.flapack (FORTRAN order)

• High level matrix routines
  • Linear Algebra Basics: inv, solve, det, norm, lstsq, pinv
  • Decompositions: eig, lu, svd, orth, cholesky, qr, schur
  • Matrix Functions: expm, logm, sqrtm, cosm, coshm, funm (general matrix functions)
Linear Algebra

**LU FACTORIZATION**

```python
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# time consuming factorization
>>> lu, piv = linalg.lu_factor(a)

# fast solve for 1 or more
# right hand sides.
>>> b = array([10,8,3])
>>> linalg.lu_solve((lu, piv), b)
array([-7.82608696, 4.56521739, 0.82608696])
```

**EIGEN VALUES AND VECTORS**

```python
>>> from scipy import linalg
>>> a = array([[1,3,5],
...            [2,5,1],
...            [2,3,6]])
# compute eigen values/vectors
>>> vals, vecs = linalg.eig(a)
# print eigen values
>>> vals
array([9.39895873+0.j, -0.73379338+0.j, 3.33483465+0.j])
# eigen vectors are in columns
# print first eigen vector
>>> vecs[:,:,0]
array([-0.57028326, -0.41979215, -0.70608183])
# norm of vector should be 1.0
>>> linalg.norm(vecs[:,:,0])
1.0
```
Optimization

scipy.optimize --- unconstrained minimization and root finding

• Unconstrained Optimization
  
  *fmin* (Nelder-Mead simplex), *fmin_powell* (Powell’s method), *fmin_bfgs* (BFGS quasi-Newton method), *fmin_ncg* (Newton conjugate gradient), *leastsq* (Levenberg-Marquardt), *anneal* (simulated annealing global minimizer), *brute* (brute force global minimizer), *brent* (excellent 1-D minimizer), *golden*, *bracket*

• Constrained Optimization
  
  *fmin_l_bfgs_b*, *fmin_tnc* (truncated newton code), *fmin_cobyla* (constrained optimization by linear approximation), *fminbound* (interval constrained 1-d minimizer)

• Root finding
  
  *fsolve* (using MINPACK), *brentq*, *brenth*, *ridder*, *newton*, *bisect*, *fixed_point* (fixed point equation solver)
Optimization

EXAMPLE: Non-linear least-squares data fitting

# fit data-points to a curve
# demo/data_fitting/datafit.py
>>> from numpy.random import randn
>>> from numpy import exp, sin, pi
>>> from numpy import linspace
>>> from scipy.optimize import leastsq

>>> def func(x, A, a, f, phi):
    return A*exp(-a*sin(f*x+pi/4))

>>> def errfunc(params, x, data):
    return func(x, *params) - data

>>> ptrue = [3, 2, 1, pi/4]
>>> x = linspace(0, 2*pi, 25)
>>> true = func(x, *ptrue)
>>> noisy = true + 0.3*randn(len(x))
>>> p0 = [1, 1, 1, 1]
>>> pmin, ier = leastsq(errfunc, p0, args=(x, noisy))
>>> pmin
array([ 3.1705,  1.9501,  1.0206,  0.7034])
THANK YOU