Numerical and Scientific Packages

Numeric and scientific applications

As you might expect, there are a number of third-party packages available for numerical and scientific computing that extend Python's basic math module.

These include:

- NumPy/SciPy – numerical and scientific function libraries.
- Numba – Python compiler that support JIT compilation.
- ALGLIB – numerical analysis library.
- Pandas – high-performance data structures and data analysis tools.
- PyGSL – Python interface for GNU Scientific Library.
- ScientificPython – collection of scientific computing modules.

Scipy and friends

By far, the most commonly used packages are those in the SciPy stack. We will focus on these in this class. These packages include:

- NumPy
- SciPy
- Matplotlib – plotting library.
- IPython – interactive computing.
- Pandas – data analysis library.
- SymPy – symbolic computation library.

Numpy

Let’s start with NumPy. Among other things, NumPy contains:

- A powerful N-dimensional array object.
- Sophisticated (broadcasting/universal) functions.
- Tools for integrating C/C++ and Fortran code.
- Useful linear algebra, Fourier transform, and random number capabilities.

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Numpy

The key to NumPy is the ndarray object, an *n*-dimensional array of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size. Modifying the size means creating a new array.
- NumPy arrays must be of the same data type, but this can include Python objects.
- More efficient mathematical operations than built-in sequence types.

Numpy Datatypes

To begin, NumPy supports a wider variety of data types than are built-in to the Python language by default. They are defined by the numpy.dtype class and include:

- intc (same as a C integer) and intp (used for indexing)
- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- float16, float32, float64
- complex64, complex128
- bool_, int_, float_, complex_ are shorthand for defaults.

These can be used as functions to cast literals or sequence types, as well as arguments to numpy functions that accept the dtype keyword argument.

Numpy Datatypes

- Some examples:

```
>>> import numpy as np
>>> x = np.float32(1.0)
>>> x
1.0
>>> y = np.int_([1,2,4])
>>> y
array([1, 2, 4])
>>> z = np.arange(3,
dtype=np.uint8)
>>> z
array([0, 1, 2], dtype=uint8)
>>> z.dtype
dtype('uint8')
```

Numpy Arrays

There are a couple of mechanisms for creating arrays in NumPy:

- Conversion from other Python structures (e.g., lists, tuples).
- Built-in NumPy array creation (e.g., arange, ones, zeros, etc.).
- Reading arrays from disk, either from standard or custom formats (e.g. reading in from a CSV file).
- and others …

Numpy arrays

- In general, any numerical data that is stored in an array-like container can be converted to an ndarray through use of the array() function. The most obvious examples are sequence types like lists and tuples.

```
>>> x = np.array([2,3,1,0])
>>> x = np.array([2, 3, 1, 0])
>>> x = np.array([[1,2.0],[0,0],(1+1j,3.)])
>>> x = np.array([[ 1.+0.j, 2.+0.j], [ 0.+0.j, 0.+0.j],
[ 1.+1.j, 3.+0.j]])
```

Numpy Arrays

There are a couple of built-in NumPy functions which will create arrays from scratch.

- zeros(shape) -- creates an array filled with 0 values with the specified shape. The default dtype is float64.

```
>>> np.zeros((2, 3))
array([[ 0., 0., 0.], [ 0., 0., 0.]])
```

- ones(shape) -- creates an array filled with 1 values.
- arange() -- creates arrays with regularly incrementing values.

```
>>> np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> np.arange(2, 10, dtype=np.float)
array([ 2., 3., 4., 5., 6., 7., 8., 9.])
>>> np.arange(2, 3, 0.1)
array([ 2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8,
2.9])
```

Numpy Arrays

- linspace() -- creates arrays with a specified number of elements, and spaced equally between the specified beginning and end values.

```
>>> np.linspace(1., 4., 6)
array([ 1. , 1.6, 2.2, 2.8, 3.4,
4. ])
```

```
>>> np.random.random((2,3))
array([[ 0.75688597, 0.41759916,
0.35007419],
       [ 0.77164187, 0.05869089,
0.98792864]])
```

- random.random(shape) – creates arrays with random floats over the interval [0,1).

Numpy Arrays

- Printing an array can be done with the print statement.

```
>>> import numpy as np
>>> a = np.arange(3)
>>> print a
[0 1 2]
>>> a
array([0, 1, 2])
>>> b =
np.arange(9).reshape(3,3)
>>> print b
[[0 1 2]
 [3 4 5]
 [6 7 8]]
>>> c =
np.arange(8).reshape(2,2,2)
>>> print c
[[[0 1]
  [2 3]]

 [[4 5]
  [6 7]]]
```

Indexing

- Single-dimension indexing is accomplished as usual.

```
>>> x = np.arange(10)
>>> x[2]
2
>>> x[-2]
8
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

- Multi-dimensional arrays support multi-dimensional indexing.

```
>>> x.shape = (2,5) # now x is 2-dimensional
>>> x[1,3]
8
>>> x[1,-1]
9
```

$$\begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

Indexing

- Using fewer dimensions to index will result in a subarray.

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

- This means that `x[i, j] == x[i][j]` but the second method is less efficient.

Indexing

- Slicing is possible just as it is for typical Python sequences.

```
>>> x = np.arange(10)
>>> x[2:5]
array([2, 3, 4])
>>> x[:-7]
array([0, 1, 2])
>>> x[1:7:2]
array([1, 3, 5])
>>> y = np.arange(35).reshape(5,7)
>>> y[1:5:2,::3]
array([[ 7, 10, 13], [21, 24, 27]])
```

Array operations

```
>>> a = np.arange(5)
>>> b = np.arange(5)
>>> a+b
array([0, 2, 4, 6, 8])
>>> a-b
array([0, 0, 0, 0, 0])
>>> a**2
array([ 0,  1,  4,  9, 16])
>>> a>3
array([False, False, False,
False,  True], dtype=bool)
>>> 10*np.sin(a)
array([ 0.,
8.41470985, 9.09297427, 1.41120008, -
7.56802495])
>>> a*b
array([ 0,  1,  4,  9, 16])
```

- Basic operations apply element-wise. The result is a new array with the resultant elements.

  Operations like *= and += will modify the existing array.

Array operations

- Since multiplication is done element-wise, you need to specifically perform a dot product to perform matrix multiplication.

```
>>> a = np.zeros(4).reshape(2,2)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> a[0,0] = 1
>>> a[1,1] = 1
>>> b = np.arange(4).reshape(2,2)
>>> b
array([[0, 1],
       [2, 3]])
>>> a*b
array([[ 0.,  0.],
       [ 0.,  3.]])
>>> np.dot(a,b)
array([[ 0.,  1.],
       [ 2.,  3.]])
```

Array operations

- There are also some built-in methods of ndarray objects.

  Universal functions which may also be applied include exp, sqrt, add, sin, cos, etc…

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.68166391, 0.98943098,
0.69361582],
       [ 0.78888081, 0.62197125,
0.40517936]])
>>> a.sum()
4.1807421388722164
>>> a.min()
0.4051793610379143
>>> a.max(axis=0)
array([ 0.78888081, 0.98943098,
0.69361582])
>>> a.min(axis=1)
array([ 0.68166391, 0.40517936])
```

Array operations

- An array shape can be manipulated by a number of methods.

  resize(size) will modify an array in place.

  reshape(size) will return a copy of the array with a new shape.

```
>>> a =
np.floor(10*np.random.random((3,4)))
>>> print a
[[ 9.  8.  7.  9.]
 [ 7.  5.  9.  7.]
 [ 8.  2.  7.  5.]]
>>> a.shape
(3, 4)
>>> a.ravel()
array([ 9., 8., 7., 9., 7., 5., 9., 7., 8.,
2., 7., 5.]) >>> a.shape = (6,2)
>>> print a
[[ 9.  8.]
 [ 7.  9.]
 [ 7.  5.]
 [ 9.  7.]
 [ 8.  2.]
 [ 7.  5.]]
>>> a.transpose()
array([[ 9., 7., 7., 9., 8., 7.],
       [ 8., 9., 5., 7., 2., 5.]])
```

Linear algebra

- One of the most common reasons for using the NumPy package is its linear algebra module.

```
>>> from numpy import *
>>> from numpy.linalg import *
>>> a = array([[1.0, 2.0], [3.0,
4.0]])
>>> print a
[[ 1.  2.]
 [ 3.  4.]]
>>> a.transpose()
array([[ 1.,  3.],
       [ 2.,  4.]])
>>> inv(a) # inverse
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Linear algebra

```
>>> u = eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[ 1., 0.],
       [ 0., 1.]])
>>> j = array([[0.0, -1.0], [1.0, 0.0]])
>>> dot(j, j) # matrix product
array([[-1., 0.],
       [ 0., -1.]])
>>> trace(u) # trace
2.0
>>> y = array([[5.], [7.]])
>>> solve(a, y) # solve linear matrix equation
array([[-3.],
       [ 4.]])
>>> eig(j) # get eigenvalues/eigenvectors of matrix
(array([ 0.+1.j, 0.-1.j]),
 array([[ 0.70710678+0.j, 0.70710678+0.j],
        [ 0.00000000-0.70710678j,
0.00000000+0.70710678j]]))
```

Matrices

- There is also a matrix class which inherits from the ndarray class.

  There are some slight differences but matrices are very similar to general arrays.

  In NumPy’s own words, the question of whether to use arrays or matrices comes down to the short answer of “use arrays”.

```
>>> A = matrix('1.0 2.0; 3.0
4.0')
>>> A
[[ 1. 2.]
 [ 3. 4.]]
>>> type(A)
<class
'numpy.matrixlib.defmatrix.matri
x'> >>> A.T # transpose
[[ 1. 3.]
 [ 2. 4.]]
>>> X = matrix('5.0 7.0')
>>> Y = X.T
>>> print A*Y # matrix
multiplication
[[19.]
 [43.]]
>>> print A.I # inverse
[[-2. 1. ]
 [ 1.5 -0.5]]
>>> solve(A, Y) # solving linear
equation
matrix([[-3.], [ 4.]])
```

NumPY docs

- There is a very nice table of NumPy equivalent operations for MATLAB users. However, even if you do not know MATLAB, this is a pretty handy overview of NumPy functionality.

SciPy

- Now we move on to SciPy. In it’s own words:

  SciPy is a collection of mathematical algorithms and convenience functions built on the Numpy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy an interactive Python session becomes a data-processing and system-prototyping environment rivaling sytems such as MATLAB, IDL, Octave, R-Lab, and SciLab.

- Basically, SciPy contains various tools and functions for solving common problems in scientific computing.

SciPY

SciPy’s functionality is implemented in a number of specific sub-modules. These include:

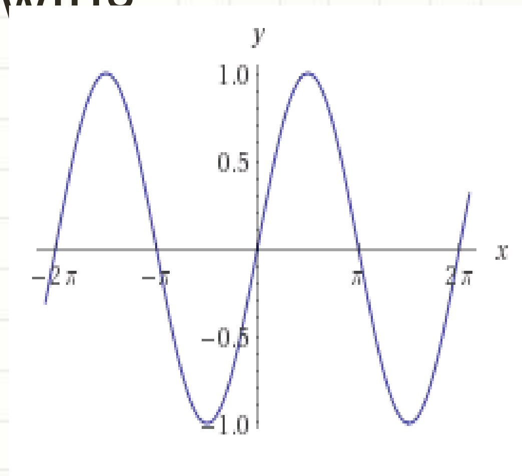
Special mathematical functions (scipy.special) -- airy, elliptic, bessel, etc.
Integration (scipy.integrate)
Optimization (scipy.optimize)
Interpolation (scipy.interpolate)
Fourier Transforms (scipy.fftpack)
Signal Processing (scipy.signal)
Linear Algebra (scipy.linalg)
Compressed Sparse Graph Routines (scipy.sparse.csgraph)
Spatial data structures and algorithms (scipy.spatial)
Statistics (scipy.stats)
Multidimensional image processing (scipy.ndimage)
Data IO (scipy.io)
Weave (scipy.weave)
and more!

scipy

- We can't possibly tour all of the SciPy library and, even if we did, it might be a little boring. So let's just look at some example modules with SciPy to see how it can be used in a Python program.

  Let's start with a simple little integration example. Say we wanted to compute the following:

$$\int_a^b \sin x \, dx$$



- Obviously, the first place we should look is scipy.integrate!

Scipy.integrate

- Methods for Integrating Functions given a function object:

`quad` -- General purpose integration.
`dblquad` -- General purpose double integration.
`tplquad` -- General purpose triple integration.
`fixed_quad` -- Integrate func(x) using Gaussian quadrature of order n.
`quadrature` -- Integrate with given tolerance using Gaussian quadrature.
`romberg` -- Integrate func using Romberg integration.

- Methods for Integrating Functions given a fixed set of samples:

`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.

Scipy.integrate

We have a function object – np.sin defines the sin function for us. We can compute the definite integral from $x = 0$ to $x = \pi$ using the quad function.

```
>>> result = scipy.integrate.quad(np.sin, 0, np.pi)
>>> print result
(2.0, 2.220446049250313e-14) # 2 with a very small error
margin!
>>> result = scipy.integrate.quad(np.sin, -np.inf,
+np.inf)
>>> print result
(0.0, 0.0) # Integral does not converge
```

Scipy.integrate

Let’s say that we don’t have a function object, we only have some (x,y) samples that “define” our function.

We can estimate the integral using the trapezoidal rule.

```
>>> sample_x = np.linspace(0, np.pi, 1000)
>>> sample_y = np.sin(sample_x) # Creating
1,000 samples
>>> result = scipy.integrate.trapz(sample_y,
sample_x)
>>> print result
1.99999835177
>>> sample_x = np.linspace(0, np.pi, 1000000)
>>> sample_y = np.sin(sample_x) # Creating
1,000,000 samples
>>> result = scipy.integrate.trapz(sample_y,
sample_x)
>>> print result
2.0
```

Connect Activity

Question:

The process of systematically applying techniques to evaluate data is known as ?

Data Analysis:

- What is it?
  - Apply logical techniques to
  - Describe, condense, recap and evaluate Data and
  - Illustrate Information

- Goals of Data Analysis:
  1. Discover useful information
  2. Provide insights
  3. Suggest conclusions
  4. Support Decision Making

What is pandas ?

- ***Pandas*** is Python package for **data analysis**.
- It Provides built-in data structures which simplify the manipulation and analysis of data sets.
- Pandas is easy to use and powerful, but “with great power comes great responsibility”
- We cannot teach you all things Pandas, we must focus on how it works, so you can figure out the rest on your own.
- http://pandas.pydata.org/pandas-docs/stable/

Watch Me Code 1

Pandas Basics

- Series
- DataFrame
- Creating a DataFrame from a dict
- Select columns, Select rows with Boolean indexing

Libraries - Pandas

- **A popular library for importing and managing datasets in Python for Machine Learning is ‘pandas’.**

Keyword to import a library

Keyword to refer to library by an alias (shortcut) name

```
import pandas as pd
```

Used for:

- **Data Analysis**
- **Data Manipulation**
- **Data Visualization**

PyData.org : high-performance, easy-to-use data structures and data analysis tools for the Python programming language.

Pandas – Indexed Arrays

- **Pandas are used to build indexed arrays (1D) and matrices (2D), where columns and rows are labeled (named) and can be accessed via the labels (names).**

raw data

1	**2**	**3**	**4**
4	5	6	7
8	9	10	11

Columns (features) index

Row (samples) index

	x1	**x2**	**x3**	**x4**
one	**1**	**2**	**3**	**4**
two	4	5	6	7
three	8	9	10	11

Panda Indexed Matrix

Pandas – Series and Data Frames

- Pandas Indexed Arrays are referred to as **Series** (1D) and **Data Frames** (2D).

- **Series** is a 1D labeled (indexed) array and can hold any data type, and mix of data types.

Series   Raw data   Column Index Labels

```
s = pd.Series( data, index=[ 'x1', 'x2', 'x3', 'x4' ] )
```

- **Data Frame** is a 2D labeled (indexed) matrix and can hold any data type, and mix of data types.

Data Frame   Row Index Labels   Column Index Labels

```
df = pd.DataFrame( data, index=['one', 'two'], columns=[ 'x1', 'x2', 'x3', 'x4' ] )
```

Pandas: Essential Concepts

- A **Series** is a named Python list (dict with list as value).
  ```
  { ‘grades’ : [50,90,100,45] }
  ```

- A **DataFrame** is a dictionary of Series (dict of series):
  ```
  {  { ‘names’ :
  [‘bob’,’ken’,’art’,’joe’]}
     { ‘grades’ : [50,90,100,45] }
   }
  ```

Check Yourself: Series or DataFrame?

Match the code to the result. One result is a Series, the other a DataFrame

1. df['Quarter']
2. df[['Quarter']]

	Quarter	Sold
0	Q1	100
1	Q2	120
2	Q3	90
3	Q4	150

Check Yourself: Boolean Index

Which rows are included in this Boolean index?

```
df[ df['Sold'] < 110 ]
```

	Quarter	Sold
0	Q1	100
1	Q2	120
2	Q3	90
3	Q4	150

Pandas – Selecting

- **Selecting One Column**

Selects column labeled x1 for all rows

```
x1 = df[ ‘x1’ ]
```

1	
4	
8	

- **Selecting Multiple Columns**

Note: df[‘x1’:’x3’] this python syntax does not work!

Selects columns labeled x1 and x3 for all rows

```
x1 = df[ [ ‘x1’, ‘x3’ ] ]
```

1	**3**
4	6
8	10

Selects columns labeled x1 through x3 for all rows

```
x1 = df.ix[ :, ‘x1’:’x3’ ]
```

Slicing function

rows (all)

columns

1	**2**	**3**
4	5	6
8	9	10

And many more functions: merge, concat, stack, …

Libraries - Matplotlib

- **A popular library for plotting and visualizing data in Python**

Keyword to import a library

Keyword to refer to library by an alias (shortcut) name

```
import matplotlib.pyplot as plt
```

Used for:

- **Plots**
- **Histograms**
- **Bar Charts**
- **Scatter Plots**
- **etc**

matplotlib.org: Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms.

Matplotlib - Plot

- **The function plot plots a 2D graph.**

Function to plot

X values to plot

Y values to plot

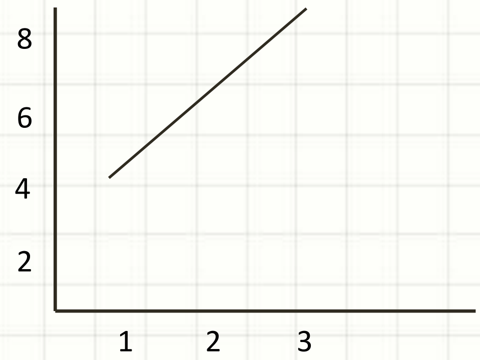
```
plt.plot( x, y )
```

- **Example:**

X

Y

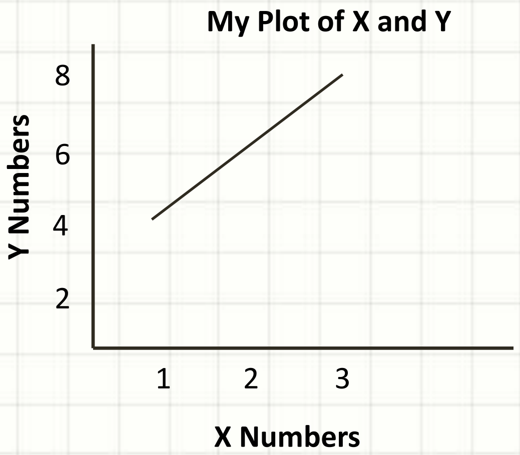
```
plt.plot( [ 1, 2, 3 ], [ 4, 6, 8 ] )  # Draws plot in the background
plt.show()                            # Displays the plot
```



Matplotlib – Plot Labels

- **Add Labels for X and Y Axis and Plot Title (caption)**

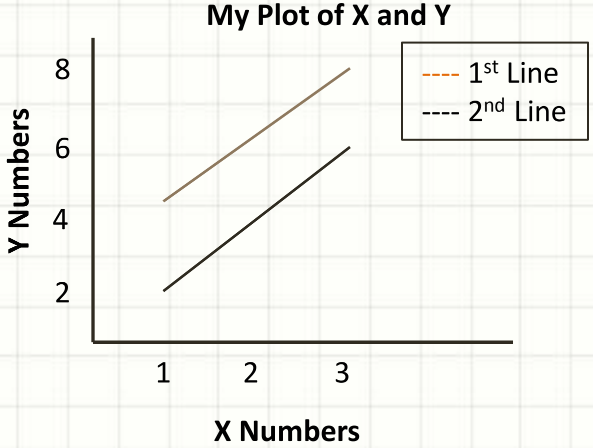
```
plt.plot( [ 1, 2, 3 ], [ 4, 6, 8 ] )
plt.xlabel( “X Numbers” )            # Label on the X-axis
plt.ylabel( “Y Numbers” )            # Label on the Y-axis
plt.title( “My Plot of X and Y”)     # Title for the Plot
plt.show()
```



Matplotlib – Multiple Plots and Legend

- **You can add multiple plots in a Graph**

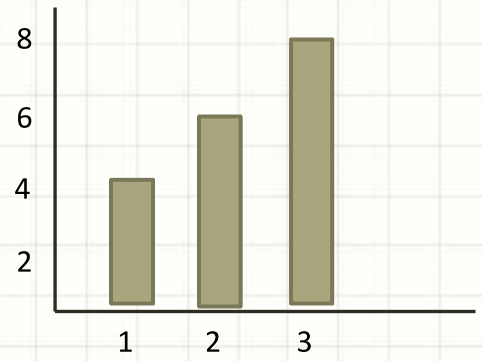
```
plt.plot( [ 1, 2, 3 ], [ 4, 6, 8 ], label=‘ 1st Line’ )      # Plot for 1st Line
plt.plot( [ 1, 2, 3 ], [ 2, 4, 6 ], label=‘2nd Line’ )       # Plot for 2nd Line
plt.xlabel( “X Numbers” )
plt.ylabel( “Y Numbers” )
plt.title( “My Plot of X and Y”)
plt.legend()                                                 # Show Legend for the plots
plt.show()
```



Matplotlib – Bar Chart

- **The function bar plots a bar graph.**

```
plt.plot( [ 1, 2, 3 ], [ 4, 6, 8 ] )  # Plot for 1st Line
plt.bar()                              # Draw a bar chart
plt.show()
```



And many more functions: hist, scatter, …