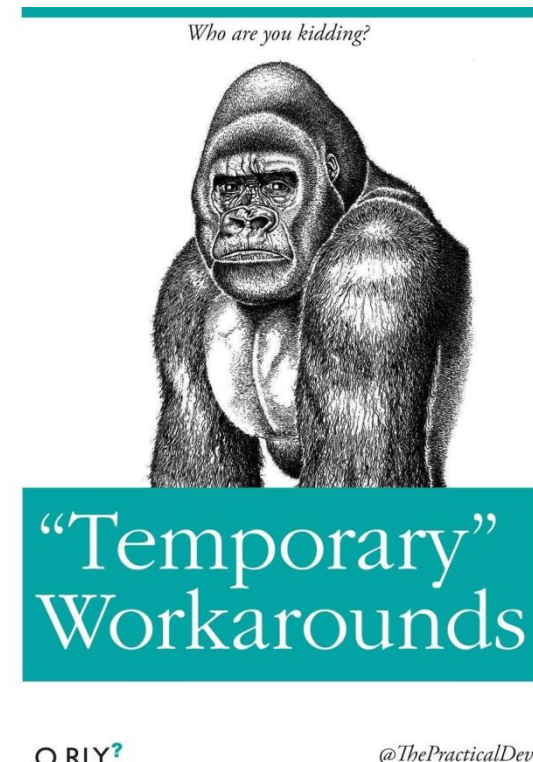
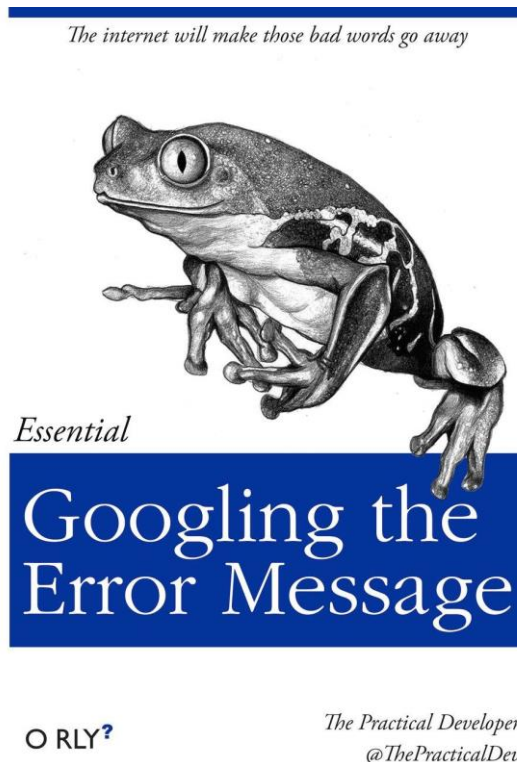
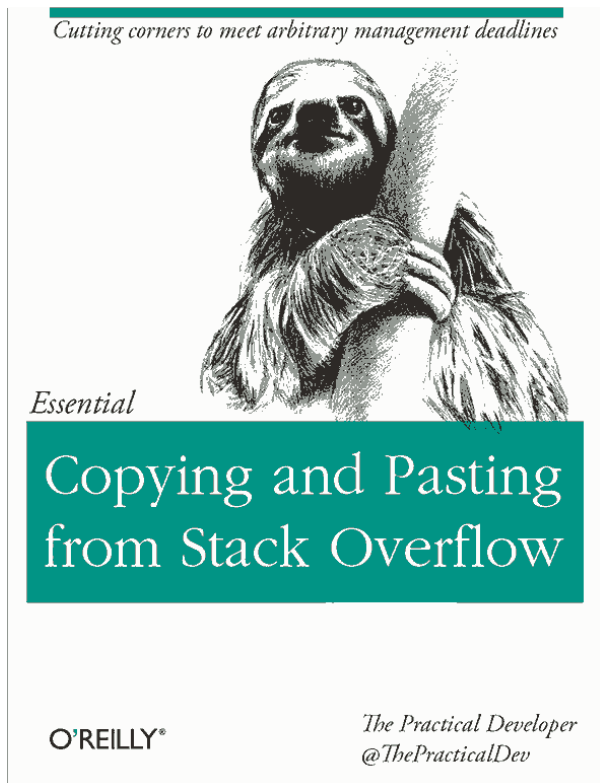


Practical Python Programming

Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
October 19, 2016

Today's training is for something like...



Outline

- **Basic Python Overview**
- **Introducing Python Modules:**
 - ❖ Numpy
 - ❖ Matplotlib
 - ❖ Scipy
- **Examples**
 - Calculate derivative of a function
 - Calculate k nearest neighbor

Overview of Basic Python

- Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language.
- It was created by Guido van Rossum during 1985-1990. Like Perl, Python source code is also available under the GNU General Public License (GPL).

Advantage of using Python

➤ Python is:

- Interpreted:
 - Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
- Interactive:
 - You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
- Object-Oriented:
 - Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
- Beginner's Language:
 - Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to browsers to games.

First “Hello World” in Python

- **Compared with other language:**
 - Fast for writing testing and developing code
 - Dynamically typed, high level, interpreted

- **First Hello World example:**

```
#!/usr/bin/env python  
print "Hello World!"
```

- **Explanation of the first line shebang (also called a hashbang, hashpling, pound bang, or crunchbang) refers to the characters "#!":**
 - Technically, in Python, this is just a comment line, can omit if run with the python command in terminal
 - In order to run the python script, we need to tell the shell three things:
 - That the file is a script
 - Which interpreter we want to execute the script
 - The path of said interpreter

Run Hello World in Script Mode

➤ On Philip (Other clusters similar):

```
[fchen14@philip1 python]$ qsub -I -l nodes=1:ppn=1 -q single
```

```
-----  
Concluding PBS prologue script - 07-Oct-2016 10:09:22  
-----
```

```
[fchen14@philip001 ~]$ module av python
```

```
-----/usr/local/packages/Modules/modulefiles/apps -----
```

```
python/2.7.10-anaconda python/2.7.7/GCC-4.9.0
```

```
[fchen14@philip001 ~]$ module load python/2.7.10-anaconda
```

```
2) mpich/3.1.4/INTEL-15.0.3 4) gcc/4.9.0
```

```
[fchen14@philip001 ~]$ which python
```

```
/usr/local/packages/python/2.7.10-anaconda/bin/python
```

```
[fchen14@philip001 ~]$ cd /home/fchen14/python/
```

```
# use python to execute the script
```

```
[fchen14@philip001 python]$ python hello.py
```

```
Hello World!
```

```
# or directly run the script
```

```
[fchen14@philip001 python]$ chmod +x hello.py
```

```
[fchen14@philip001 python]$ ./hello.py
```

```
Hello World!
```

Or Run Interactively

➤ **On Philip (Similar on other clusters):**

```
[fchen14@philip001 python]$ python
Python 2.7.10 |Anaconda 2.3.0 (64-bit)| (default, May 28 2015, 17:02:03)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux2
...
>>> print "Hello World!"
Hello World!
```

➤ **Easier for todays demonstration:**

```
[fchen14@philip001 ~]$ ipython
Python 2.7.10 |Anaconda 2.3.0 (64-bit)| (default, May 28 2015, 17:02:03)
Type "copyright", "credits" or "license" for more information.
...
In [1]: %pylab
Using matplotlib backend: Qt4Agg
Populating the interactive namespace from numpy and matplotlib
In [2]:
```


Directly Run into List/Array

```
#!/usr/bin/env python
# generate array from 0-4
a = list(range(5))
print a
# len(a)=5
for idx in range(len(a)):
    a[idx] += 5
print a
```

```
[fchen14@philip001 python]$ ./loop_array.py
[0, 1, 2, 3, 4]
[5, 6, 7, 8, 9]
```

Python Tuples

- A Python tuple is a sequence of immutable Python objects. Creating a tuple is as simple as putting different comma-separated values.

```
#!/usr/bin/env python
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
# The empty tuple is written as two parentheses containing nothing
tup1 = ();
# To write a tuple containing a single value you have to include a comma,
tup1 = (50,);
# Accessing Values in Tuples
print "tup1[0]: ", tup1[0]
print "tup2[1:5]: ", tup2[1:5]
# Updating Tuples, create a new tuple as follows
tup3 = tup1 + tup2;
print tup3
# delete tuple elements
del tup3;
print "After deleting tup3 : "
print tup3
```

Practical Python Programming

Introducing Numpy

Numpy Overview

- NumPy (Numeric Python) is the fundamental package for scientific computing in Python.
- It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices)
- An assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
- In short , NumPy package provides basic routines for manipulating large arrays and matrices of numeric data.

Basic Array Operations

- Simple array math using np.array
- Note that NumPy array starts its index from **0**, end at **N-1** (C-style)

```
# To avoid module name collision inside package context
```

```
>>> import numpy as np
```

```
>>> a = np.array([1,2,3])
```

```
>>> b = np.array([4,5,6])
```

```
>>> a+b
```

```
array([5, 7, 9])
```

```
>>> a*b
```

```
array([ 4, 10, 18])
```

```
>>> a ** b
```

```
array([ 1, 32, 729])
```

Setting Array Element Values

```
>>> a[0]
1
>>> a[0]=11
>>> a
array([11,  2,  3,  4])
>>> a.fill(0) # set all values in the array with 0
>>> a[:]=1 # why we need to use [:]?
>>> a
array([1, 1, 1, 1])
>>> a.dtype # note that a is still int64 type !
dtype('int64')
>>> a[0]=10.6 # decimal parts are truncated, be careful!
>>> a
array([10,  1,  1,  1])
>>> a.fill(-3.7) # fill() will have the same behavior
>>> a
array([-3, -3, -3, -3])
```

Numpy Array Properties (1)

```
>>> a = np.array([0,1,2,3]) # create a from a list
# create evenly spaced values within [start, stop)
>>> a = np.arange(1,5)
>>> a
array([1, 2, 3, 4])
>>> type(a)
<type 'numpy.ndarray'>
>>> a.dtype
dtype('int64')
# Length of one array element in bytes
>>> a.itemsize
8
```

Numpy Array Properties (2)

```
# shape returns a tuple listing the length of the array
# along each dimension.
>>> a.shape # or np.shape(a)
>>> a.size  # or np.size(a), return the total number of elements
4
# return the number of bytes used by the data portion of the array
>>> a.nbytes
32
# return the number of dimensions of the array
>>> a.ndim
1
```


Numpy Array Creation Functions (1)

Nearly identical to Python's range(). Creates an array of values in the range [start,stop) with the specified step value. Allows non-integer values for start, stop, and step. Default dtype is derived from the start, stop, and step values.

```
>>> np.arange(4)
array([0, 1, 2, 3])
>>> np.arange(0, 2*np.pi, np.pi/4)
array([ 0.,  0.78539816,  1.57079633,  2.35619449,  3.14159265,  3.92699082,  4.71238898,  5.49778714])
>>> np.arange(1.5,2.1,0.3)
array([ 1.5,  1.8,  2.1])
# ONES, ZEROS
# ones(shape, dtype=float64)
# zeros(shape, dtype=float64)
# shape is a number or sequence
```

specifying the dimensions of the # array. If dtype is not specified, # it defaults to float64.

```
>>> a=np.ones((2,3))
>>> a
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> a.dtype
dtype('float64')
>>> a=np.zeros(3)
>>> a
array([ 0.,  0.,  0.])
>>> a.dtype
dtype('float64')
```

Numpy Array Creation Functions (2)

```
# Generate an n by n identity
# array. The default dtype is
# float64.
```

```
>>> a = np.identity(4)
>>> a
array([[ 1.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.],
       [ 0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  1.]])
```

```
>>> a.dtype
dtype('float64')
>>> np.identity(4, dtype=int)
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

```
# empty(shape, dtype=float64,
# order='C')
```

```
>>> a = np.empty(2)
>>> a
array([ 0.,  0.])
# fill array with 5.0
>>> a.fill(5.0)
>>> a
array([ 5.,  5.])
# alternative approach
# (slightly slower)
>>> a[:] = 4.0
>>> a
array([ 4.,  4.]])
```

Numpy Array Creation Functions (3)

```
# Generate N evenly spaced elements between (and including)  
# start and stop values.
```

```
>>> np.linspace(0,1,5)  
array([ 0. ,  0.25,  0.5 ,  0.75,  1.  ])
```

```
# Generate N evenly spaced elements on a log scale between  
# base**start and base**stop (default base=10).
```

```
>>> np.logspace(0,1,5)  
array([ 1., 1.77827941, 3.16227766,  5.62341325, 10.] )
```

Array from/to ASCII files

➤ **Useful tool for generating array from txt file**

- loadtxt
- genfromtxt

➤ **Consider the following example:**

```
# data.txt
Index
Brain Weight
Body Weight
#here is the training set
1      3.385      44.500 abjkh
2      0.480      33.38  bc_00asdk
...
#here is the cross validation set
6      27.660     115.000 rk
7      14.830      98.200 fff
...
9      4.190      58.000 kij
```

Using loadtxt and genfromtxt

```
>>> a= np.loadtxt('data.txt',skiprows=16,usecols={0,1,2},dtype=None,comments="#")
```

```
>>> a
```

```
array([[ 1.    ,  3.385,  44.5  ],
       [ 2.    ,  0.48 ,  33.38 ],
       [ 3.    ,  1.35 ,   8.1  ],
       [ 4.    , 465.    , 423.   ],
       [ 5.    ,  36.33 , 119.5  ],
       [ 6.    ,  27.66 ,  115.   ],
       [ 7.    ,  14.83 ,   98.2  ],
       [ 8.    ,   1.04 ,    5.5  ],
       [ 9.    ,   4.19 ,   58.   ]])
```

np.genfromtxt can guess the actual type of your columns by using dtype=None

```
>>> a= np.genfromtxt('data.txt',skip_header=16,dtype=None)
```

```
>>> a
```

```
array([(1, 3.385, 44.5, 'abjkh'), (2, 0.48, 33.38, 'bc_00asdk'),
       (3, 1.35, 8.1, 'fb'), (4, 465.0, 423.0, 'cer'),
       (5, 36.33, 119.5, 'rg'), (6, 27.66, 115.0, 'rk'),
       (7, 14.83, 98.2, 'fff'), (8, 1.04, 5.5, 'zxs'),
       (9, 4.19, 58.0, 'kij')],
      dtype=[('f0', '<i8'), ('f1', '<f8'), ('f2', '<f8'), ('f3', 'S9')])
```

Reshaping arrays

```
>>> a = np.arange(6)
>>> a
array([0, 1, 2, 3, 4, 5])
>>> a.shape
(6,)
>>> a.shape = (2,3) # reshape array to 2x3
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
>>> a.reshape(3,2) # reshape array to 3x2
array([[0, 1],
       [2, 3],
       [4, 5]])
>>> a.reshape(2,5) # cannot change the number of elements in the array
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
>>> a.reshape(2,-1) # numpy determines the last dimension
```

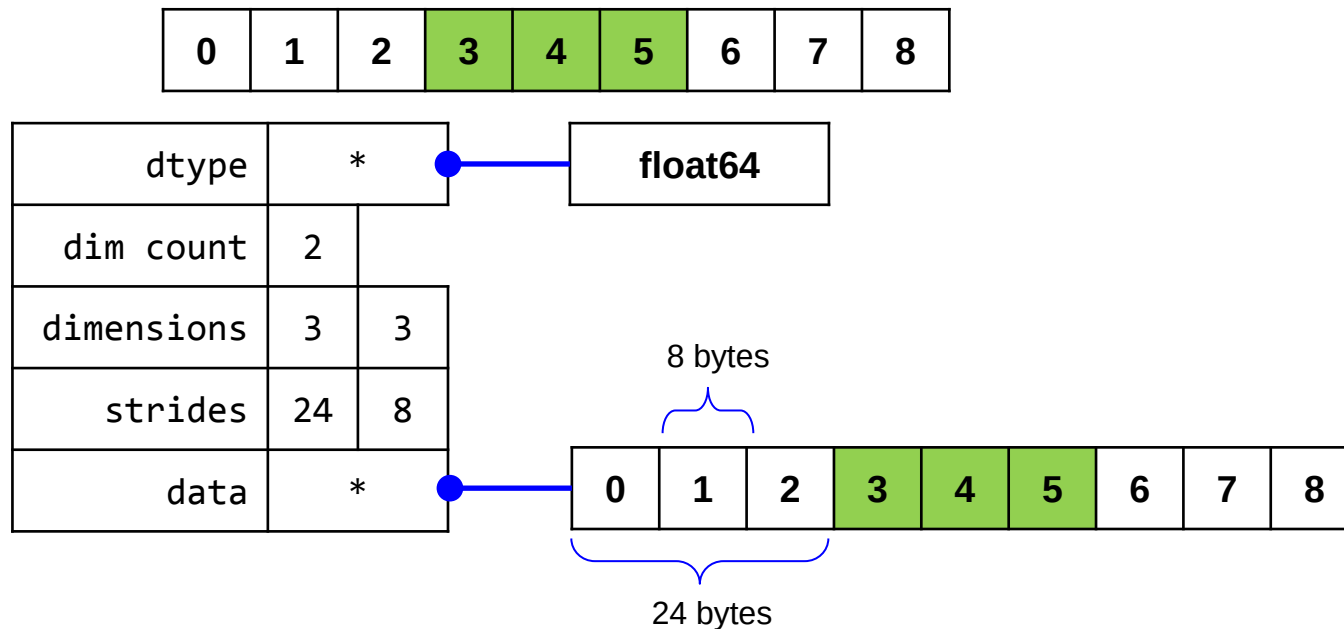
Numpy Array Data Structure

➤ Numpy view of 2D array

```
>>> a=arange(9).reshape(3,-1)
>>> a.strides
(24, 8)
>>> a.ndim
2
```

0	1	2
4	5	6
7	8	9

➤ Memory block of the 2D array



Flattening Multi-dimensional Arrays

```
# Note the difference between
# a.flatten() and a.flat
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
# a.flatten() converts a
# multidimensional array into
# a 1-D array. The new array is a
# copy of the original data.
>>> b = a.flatten()
>>> b
array([1, 2, 3, 4, 5, 6])
>>> b[0] = 7
>>> b
array([7, 2, 3, 4, 5, 6])
>>> a
array([[1, 2, 3],
       [4, 5, 6]])
```

```
# a.flat is an attribute that
# returns an iterator object that
# accesses the data in the multi-
# dimensional array data as a 1-D
# array. It references the original
# memory.
>>> a.flat
<numpy.flatiter object at 0x1421c40>
>>> a.flat[:]
array([1, 2, 3, 4, 5, 6])
>>> b = a.flat
>>> b[0] = 7
>>> a
array([[7, 2, 3],
       [4, 5, 6]])
```


(Un)raveling Multi-dimensional Arrays

```
>>> a
array([[7, 2, 3],
       [4, 5, 6]])
# ravel() is the same as flatten
# but returns a reference of the
# array if possible
>>> b = a.ravel()
>>> b
array([7, 2, 3, 4, 5, 6])
>>> b[0] = 13
>>> b
array([13, 2, 3, 4, 5, 6])
>>> a
array([[13, 2, 3],
       [4, 5, 6]])
```

```
>>> at = a.transpose()
>>> at
array([[13, 4],
       [ 2, 5],
       [ 3, 6]])
>>> b = at.ravel()
>>> b
array([13, 4, 2, 5, 3, 6])
>>> b[0]=19
>>> b
array([19, 4, 2, 5, 3, 6])
>>> a
array([[13, 2, 3],
       [4, 5, 6]])
```

Practical Python Programming

Basic Usage of Matplotlib

Introduction

- Matplotlib is probably the single most used Python package for 2D-graphics. (<http://matplotlib.org/>)
- It provides both a very quick way to visualize data from Python and *publication-quality* figures in many formats.
- Provides Matlab/Mathematica-like functionality.

matplotlib

Simple plot

- Draw the cosine and sine functions on the same plot.

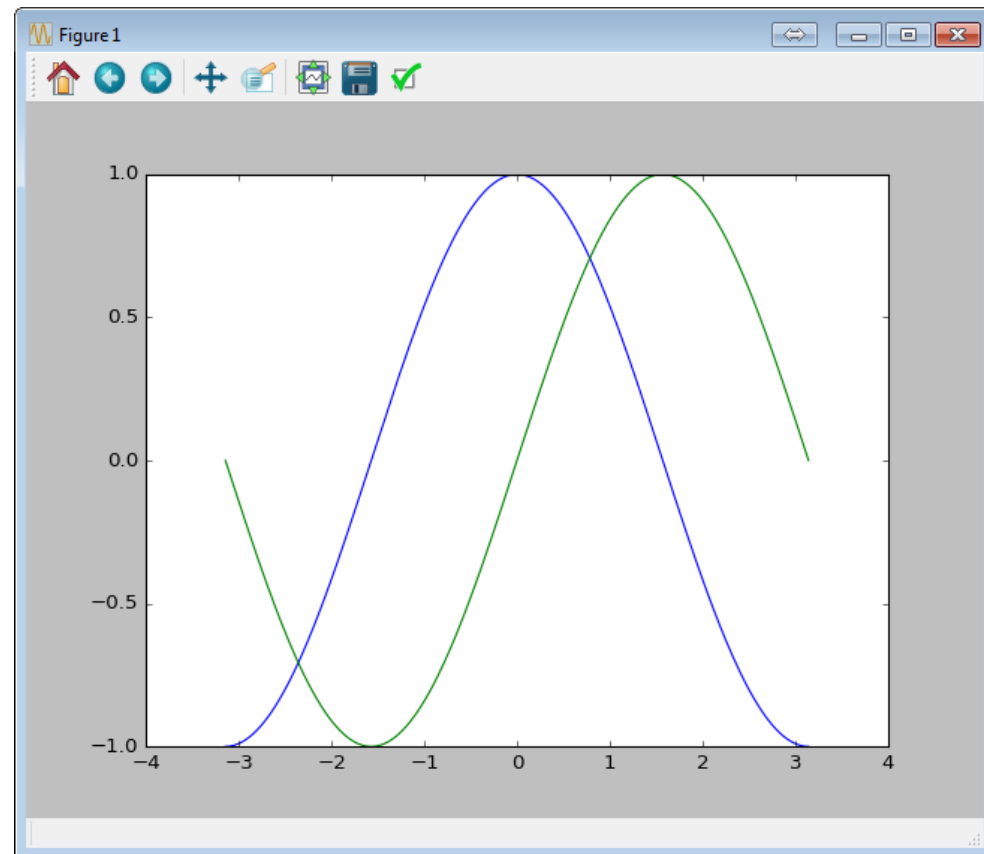
```
import numpy as np

# X is now a numpy array with 256 values ranging [-pi, pi]
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
# C is the cosine (256 values) and S is the sine (256 values).
C,S = np.cos(X), np.sin(X)
```

Using default settings to plot

- Plot the sine and cosine arrays using the default settings

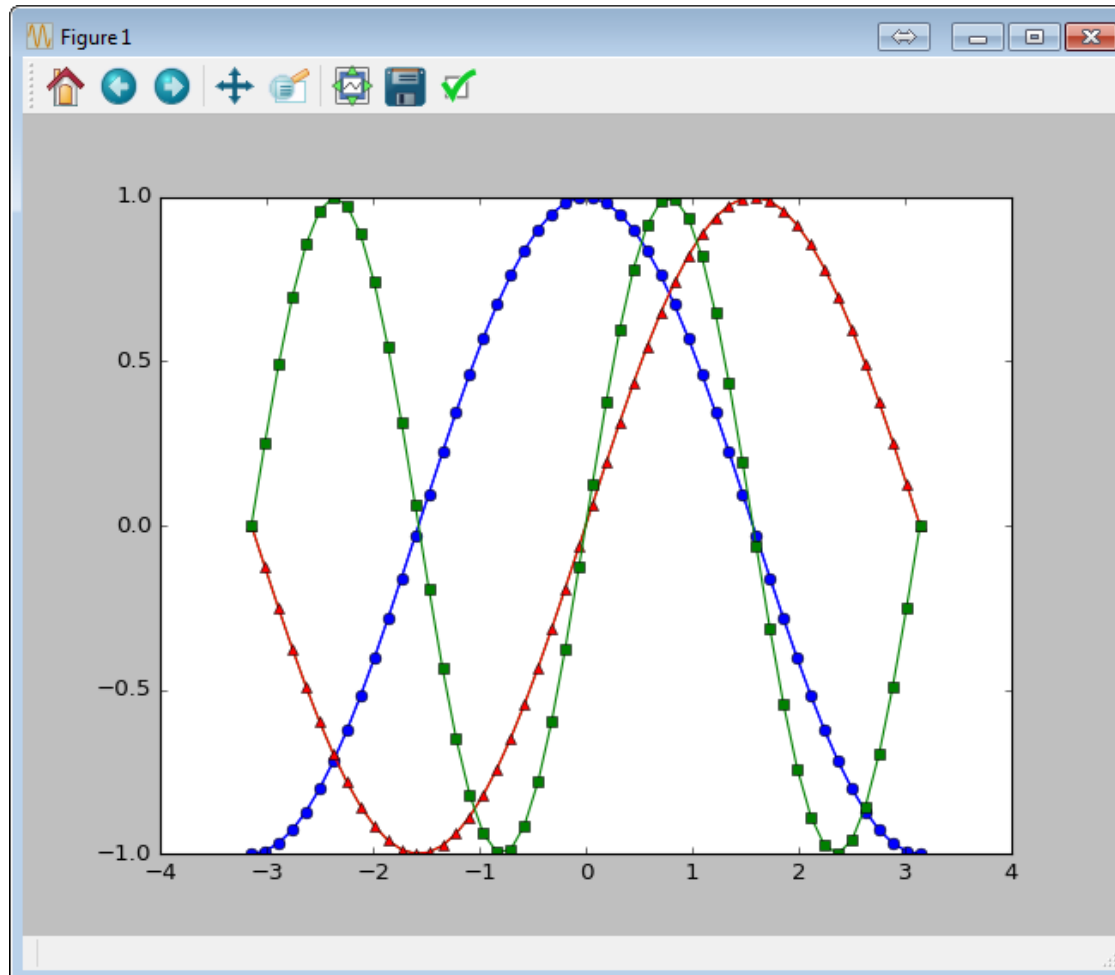
```
import numpy as np
import matplotlib.pyplot as plt
X = np.linspace(-np.pi, np.pi, 256, endpoint=True)
C,S = np.cos(X), np.sin(X)
# plt.plot(X,C)
# plt.plot(X,S)
plt.plot(X,C,X,S)
plt.show()
```



Line Formatting and Multiple Plot Groups

plot multiple groups with different line styles

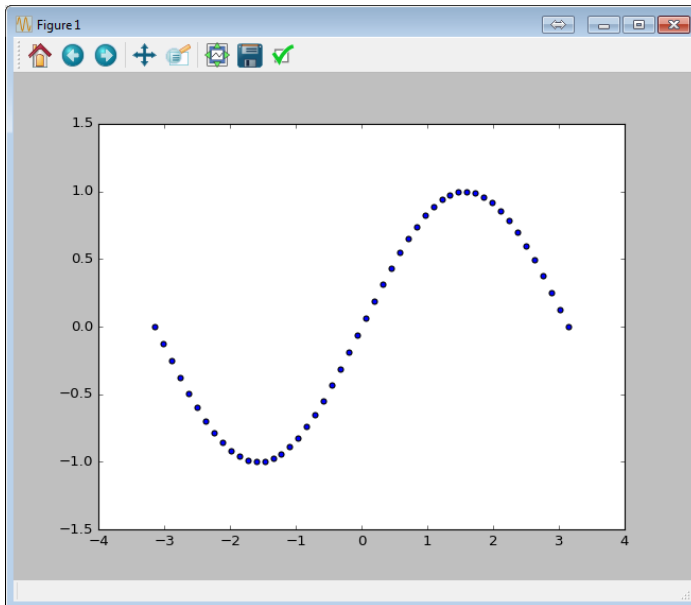
```
plt.plot(X,C,'b-o',X,S,'r-^',X,np.sin(2*X),'g-s')
```



Scatter Plots

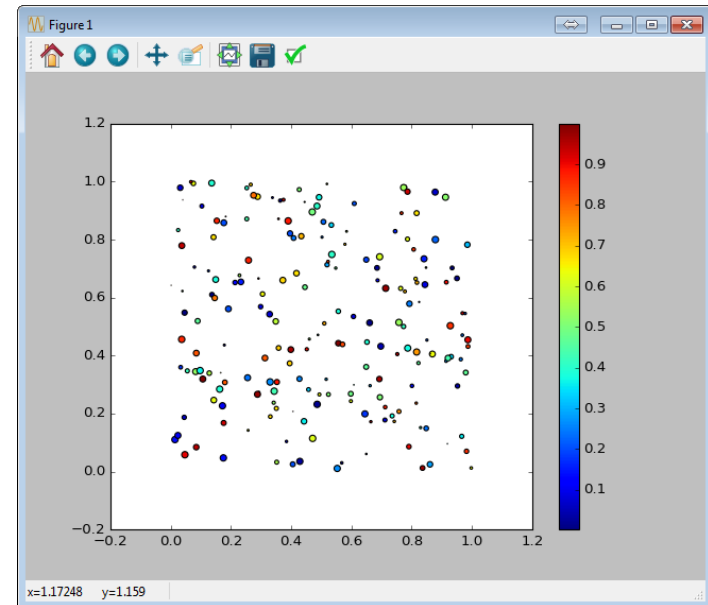
➤ Simple Scatter Plot

```
plt.scatter(X,S)
plt.show()
```



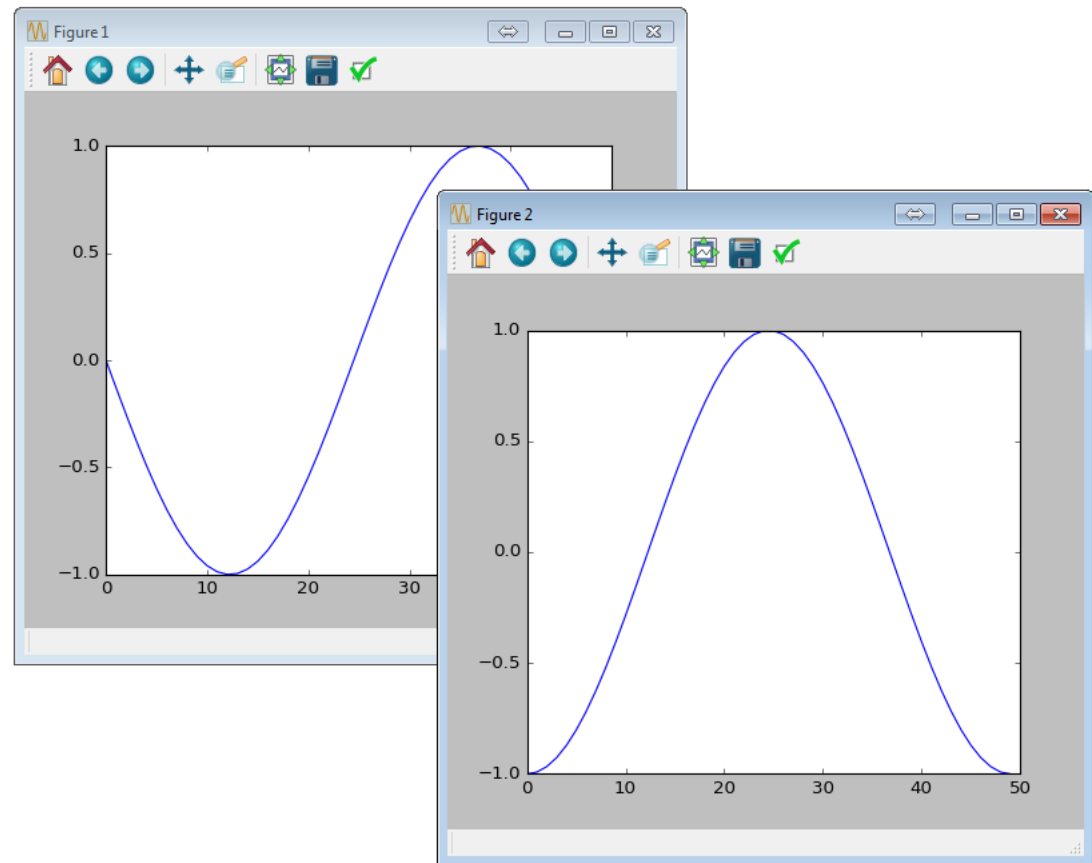
➤ Scatter Plot with Colormap

```
x = np.random.rand(200)
y = np.random.rand(200)
size = np.random.rand(200)*30
color = np.random.rand(200)
plt.scatter(x, y, size, color)
plt.colorbar()
```



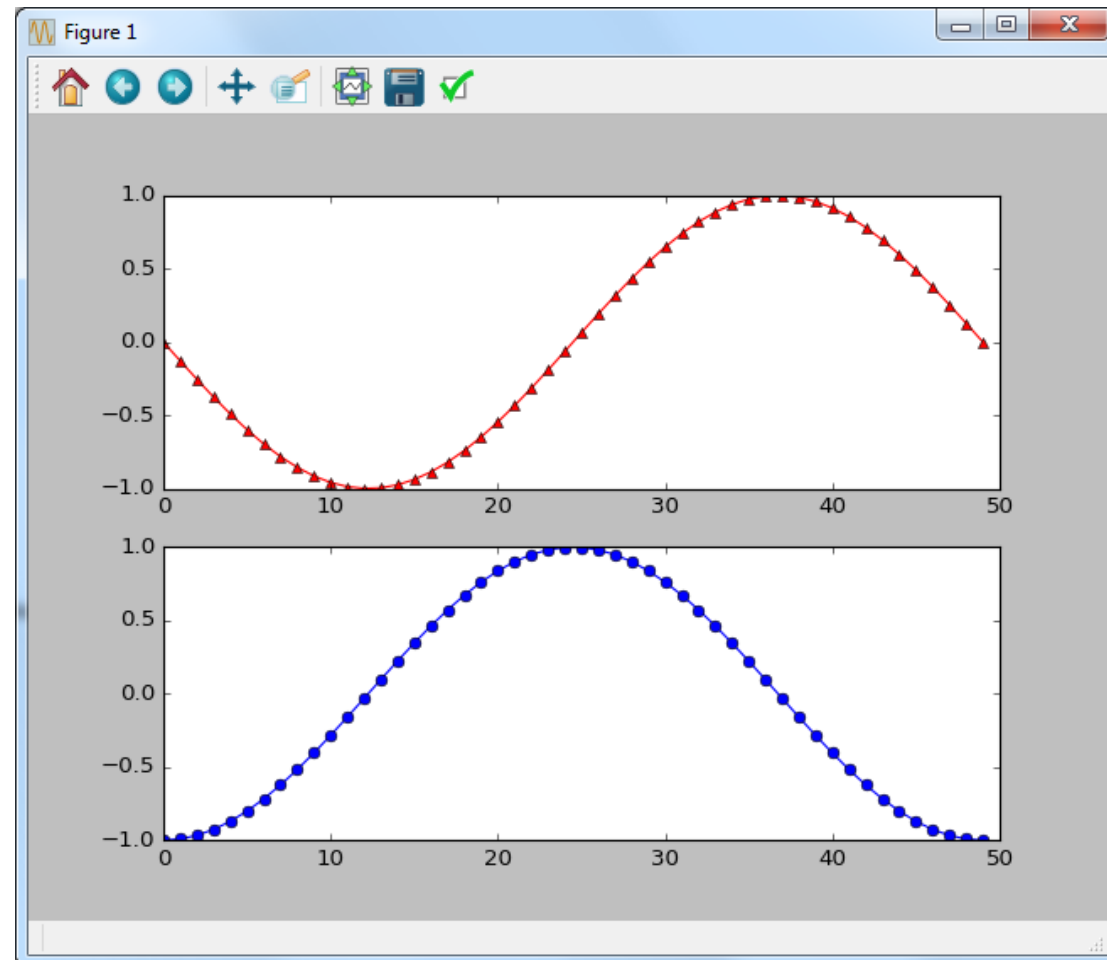
Multiple Figures

```
X = np.linspace(-np.pi, np.pi, 50, endpoint=True)
C, S = np.cos(X), np.sin(X)
# create a figure
plt.figure()
plt.plot(S)
# create a new figure
plt.figure()
plt.plot(C)
plt.show()
```



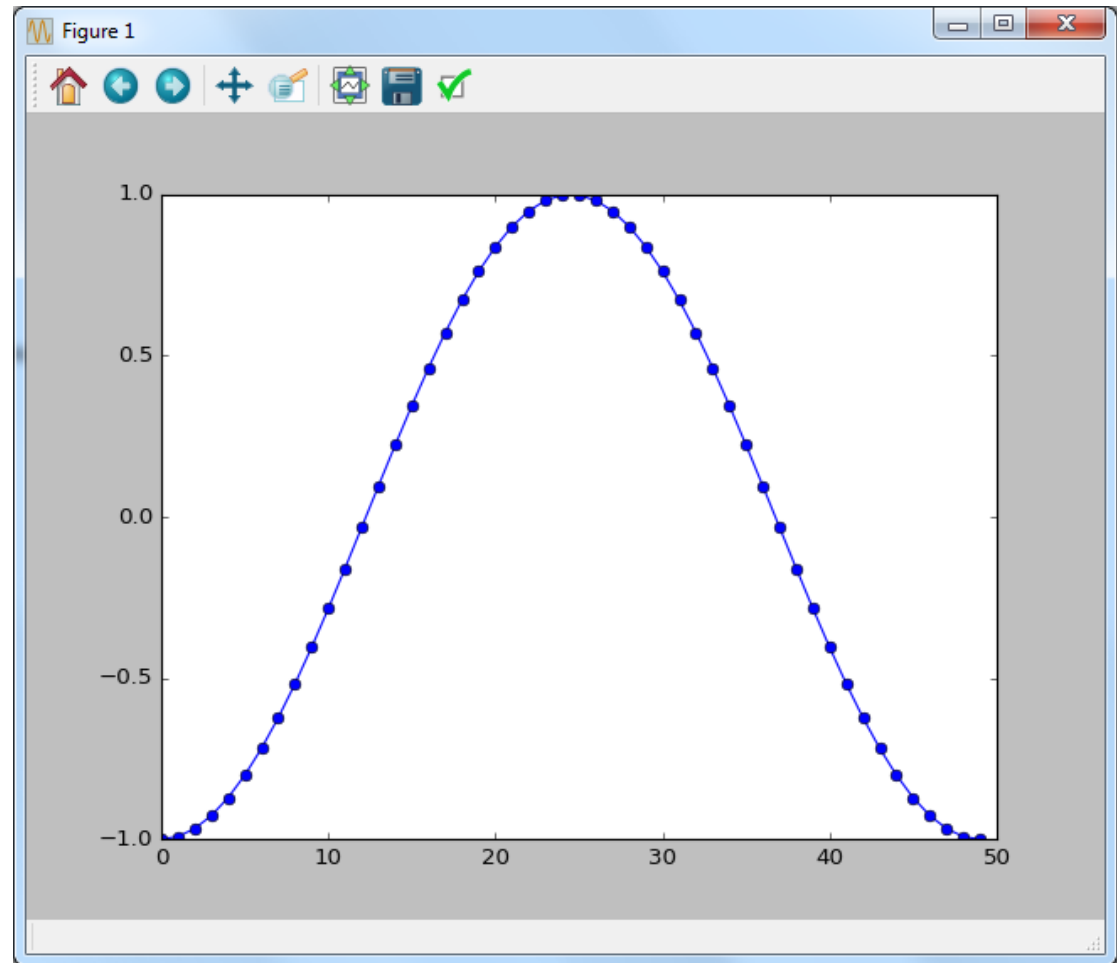
Multiple Plots Using subplot

```
# divide the plotting area in 2 rows and 1 column(s)
# subplot(rows, columns, active_plot)
plt.subplot(2, 1, 1)
plt.plot(S, 'r-^')
# create a new figure
plt.subplot(2, 1, 2)
plt.plot(C, 'b-o')
plt.show()
```



Erase the Previous Curves

```
plt.plot(S, 'r-^')  
# whether to keep the old plot use hold(True/False)  
plt.hold(False)  
plt.plot(C, 'b-o')  
plt.show()
```



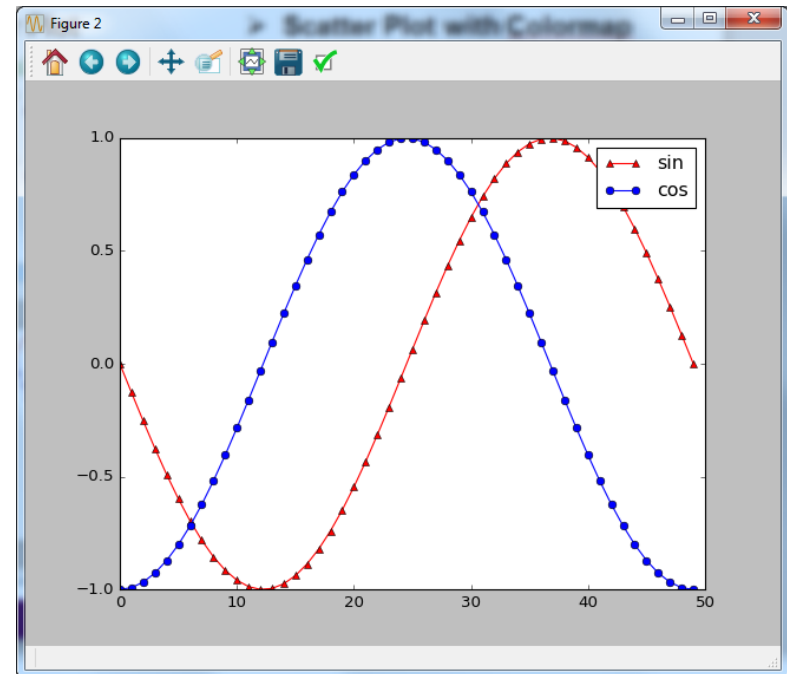
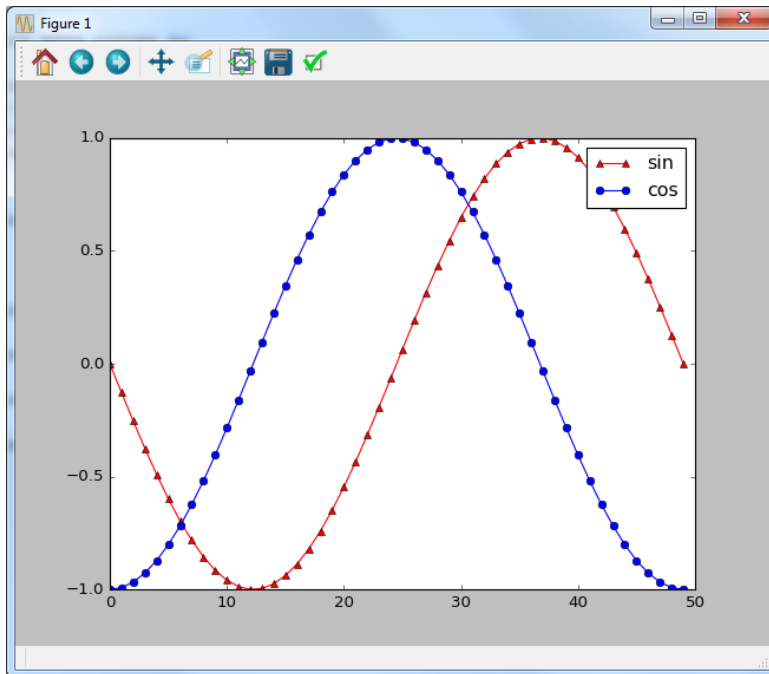
Adding Legend to Plot

➤ Legend labels with plot

```
# Add labels in plot command
plt.plot(S, 'r-^', label='sin')
plt.plot(C, 'b-o', label='cos')
plt.legend()
```

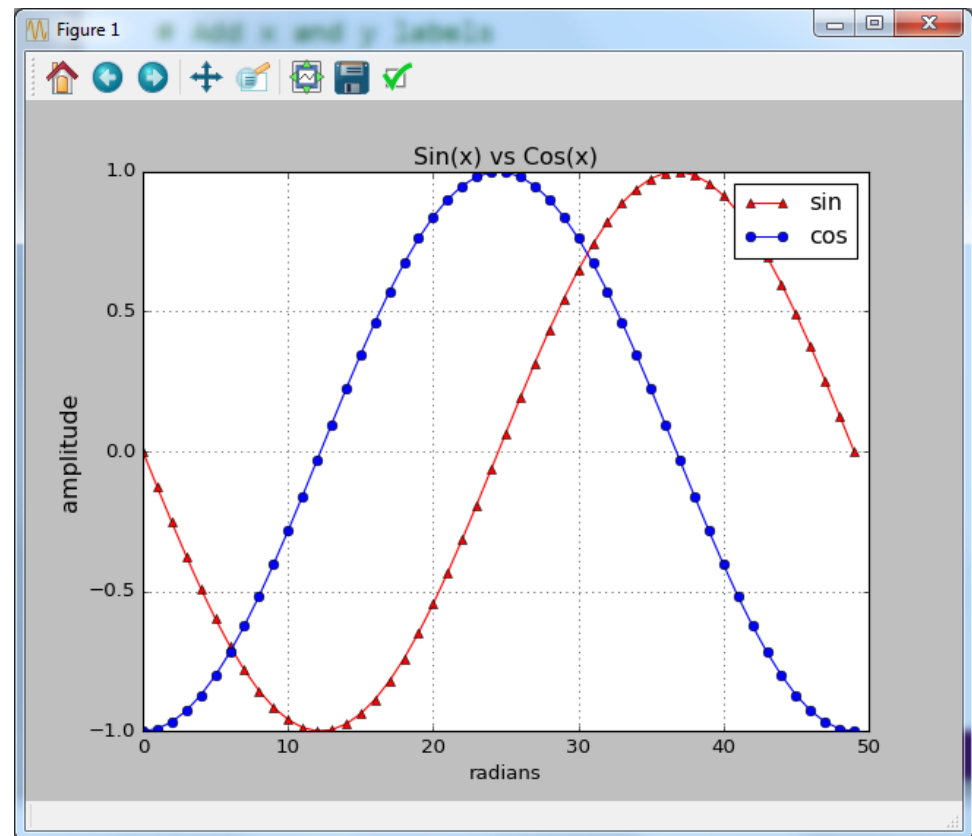
➤ Label with plt.legend

```
# Add labels via list in legend.
plt.plot(S, 'r-^', C, 'b-o')
plt.legend(['sin','cos'])
```



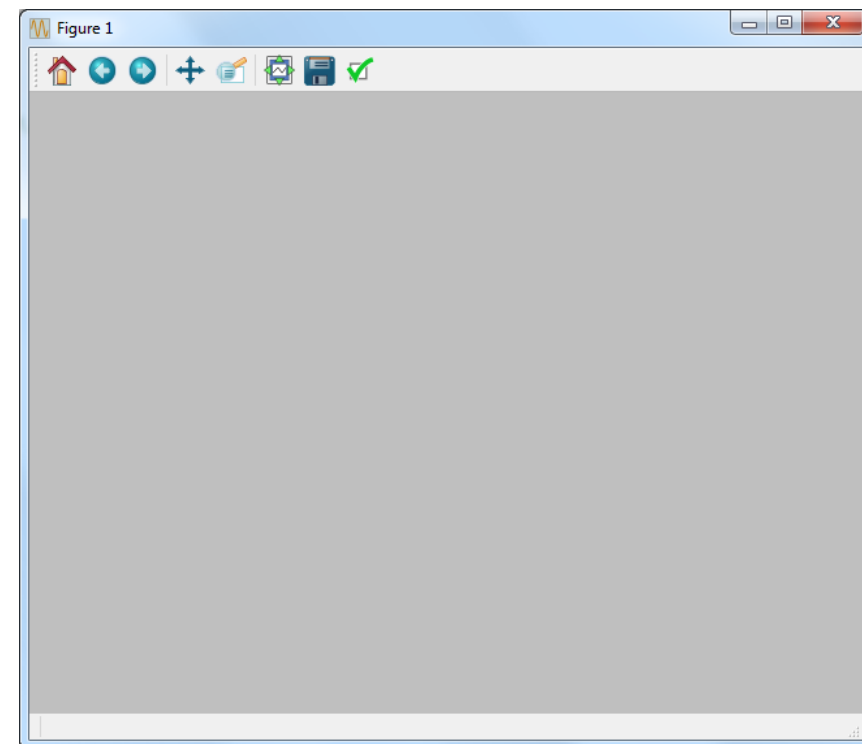
Adding Titles and Grid

```
# Add x and y labels  
plt.xlabel('radians')  
# Keywords set text properties.  
plt.ylabel('amplitude', fontsize='large')  
# Add title and show grid  
plt.title('Sin(x) vs Cos(x)')  
plt.grid()  
plt.show()
```

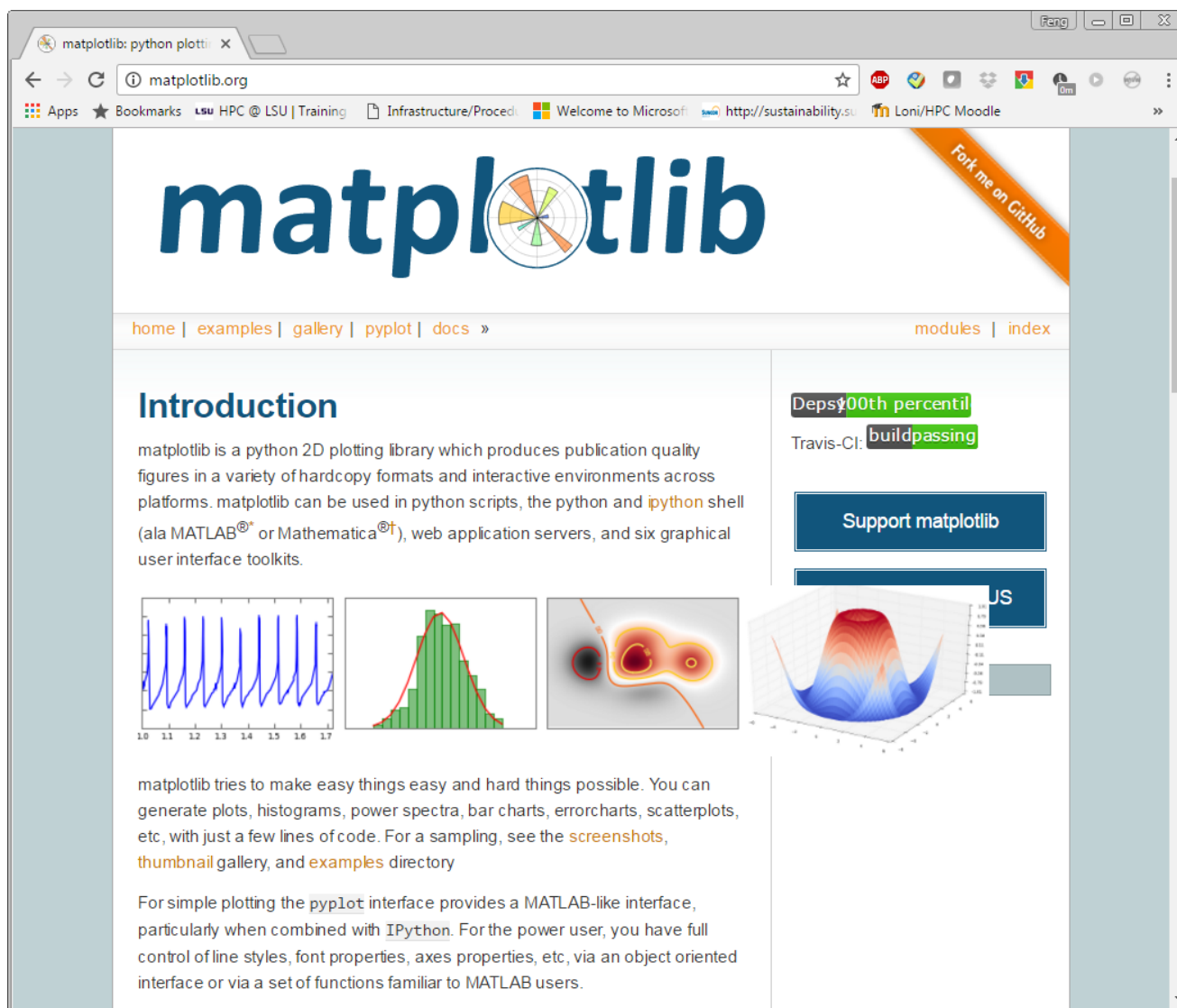


Clearing and Closing Plots

```
# Plot some curves command
plt.plot(S, 'r-^', label='sin')
plt.plot(C, 'b-o', label='cos')
# clf will clear the current plot (figure).
plt.clf()
plt.show()
# close() will close the currently
# active plot window.
plt.close()
# close('all') closes all the plot
# windows.
plt.close('all')
```



Visit Matplotlib Website for More



The screenshot shows the Matplotlib website homepage. At the top, there's a navigation bar with links: [home](#), [examples](#), [gallery](#), [pyplot](#), [docs](#), [modules](#), and [index](#). The main heading is "matplotlib" with a circular logo. Below the heading, there's an "Introduction" section. The text describes Matplotlib as a Python 2D plotting library that produces publication-quality figures in various formats and interactive environments. It mentions that Matplotlib can be used in Python scripts, the Python and IPython shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits. To the right of the introduction, there are badges for "Deps100th percentil" and "Travis-CI: buildpassing", and a button that says "Support matplotlib". Below the text, there are four small plots: a line plot with many peaks, a histogram with a red curve, a contour plot, and a 3D surface plot. At the bottom, there's more text about the pyplot interface and IPython.

matplotlib

[home](#) | [examples](#) | [gallery](#) | [pyplot](#) | [docs](#) » [modules](#) | [index](#)

Introduction

matplotlib is a python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. matplotlib can be used in python scripts, the python and [ipython](#) shell (ala MATLAB® or Mathematica®), web application servers, and six graphical user interface toolkits.

matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code. For a sampling, see the [screenshots](#), [thumbnail gallery](#), and [examples](#) directory

For simple plotting the [pyplot](#) interface provides a MATLAB-like interface, particularly when combined with [IPython](#). For the power user, you have full control of line styles, font properties, axes properties, etc, via an object oriented interface or via a set of functions familiar to MATLAB users.

Support matplotlib

Deps100th percentil
Travis-CI: buildpassing

Four Tools in Numpy

- **Removing loops using NumPy**
 - 1) Ufunc (Universal Function)
 - 2) Aggregation
 - 3) Broadcasting
 - 4) Slicing, masking and fancy indexing

Numpy's Universal Functions

- Numpy's universal function (or ufunc for short) is a function that operates on ndarrays in an element-by-element fashion
- Ufunc is a “vectorized” wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs.
- Many of the built-in functions are implemented in compiled C code.
 - They can be much faster than the code on the Python level

Ufunc: Math Functions on Numpy Arrays

```
>>> x = np.arange(5.)
>>> x
array([ 0.,  1.,  2.,  3.,  4.])
>>> c = np.pi
>>> x *= c
array([ 0.          ,  3.14159265,  6.28318531,  9.42477796,
 12.56637061])
>>> y = np.sin(x)
>>> y
array([ 0.00000000e+00,  1.22464680e-16, -2.44929360e-16,
        3.67394040e-16, -4.89858720e-16])
>>> import math
>>> y = math.sin(x) # must use np.sin to perform array math
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: only length-1 arrays can be converted to Python scalars
```

Ufunc: Many ufuncs available

- **Arithmetic Operators:** `+` `-` `*` `/` `//` `%` `**`
- **Bitwise Operators:** `&` `|` `~` `^` `>>` `<<`
- **Comparison Oper's:** `<` `>` `<=` `>=` `==` `!=`
- **Trig Family:** `np.sin`, `np.cos`, `np.tan` ...
- **Exponential Family:** `np.exp`, `np.log`, `np.log10` ...
- **Special Functions:** `scipy.special.*`
- ... and many, many more.

Aggregation Functions

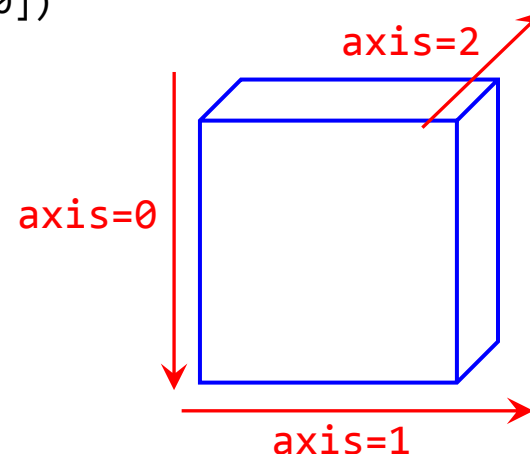
- **Aggregations are functions which summarize the values in an array (e.g. min, max, sum, mean, etc.)**
- **Numpy aggregations are much faster than Python built-in functions**

Numpy Aggregation - Array Calculation

```
>>> a=np.arange(6).reshape(2,-1)
>>> a
array([[0, 1, 2],
       [3, 4, 5]])
# by default a.sum() adds up all values
>>> a.sum()
15
# same result, functional form
>>> np.sum(a)
15
# note this is not numpy's sum!
>>> sum(a)
array([3, 5, 7])
# not numpy's sum either!
>>> sum(a,axis=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sum() takes no keyword
arguments
```

```
# sum along different axis
>>> np.sum(a,axis=0)
array([3, 5, 7])
>>> np.sum(a,axis=1)
array([ 3, 12])
>>> np.sum(a,axis=-1)
array([ 3, 12])
# product along different axis
>>> np.prod(a,axis=0)
array([ 0,  4, 10])
>>> a.prod(axis=1)
array([ 0, 60])
```

The **axes** of an array describe the order of indexing into the array, e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.



Numpy Aggregation – Statistical Methods

```
>>> np.set_printoptions(precision=4)
# generate 2x3 random float array
>>> a=np.random.random(6).reshape(2,3)
>>> a
array([[ 0.7639,  0.6408,  0.9969],
       [ 0.5546,  0.5764,  0.1712]])
>>> a.mean(axis=0)
array([ 0.6592,  0.6086,  0.5841])
>>> a.mean()
0.61730865425015347
>>> np.mean(a)
0.61730865425015347
# average can use weights
>>> np.average(a,weights=[1,2,3],axis=1)
array([ 0.8394,  0.3702])
# standard deviation
>>> a.std(axis=0)
array([ 0.1046,  0.0322,  0.4129])
```

```
# variance
>>> np.var(a, axis=1)
array([ 0.0218,  0.0346])
>>> a.min()
0.17118969968007625
>>> np.max(a)
0.99691892655137737
# find index of the minimum
>>> a.argmin(axis=0)
array([1, 1, 1])
>>> np.argmax(a,axis=1)
array([2, 1])
# this will return flattened index
>>> np.argmin(a)
5
>>> a.argmax()
2
```

Numpy's Aggregation - Summary

➤ **All have the same call style.**

- `np.min()` `np.max()` `np.sum()` `np.prod()`
- `np.argsort()`
- `np.mean()` `np.std()` `np.var()` `np.any()`
- `np.all()` `np.median()` `np.percentile()`
- `np.argmin()` `np.argmax()` . . .
- `np.nanmin()` `np.nanmax()` `np.nansum()`. . .

Array Broadcasting

- **Broadcasting is a set of rules by which ufuncs operate on arrays of different sizes and/or dimensions.**
- **Broadcasting allows NumPy arrays of different dimensionality to be combined in the same expression.**
- **Arrays with smaller dimension are broadcasted to match the larger arrays, without copying data.**

Broadcasting Rules

1. If array shapes differ, left-pad the smaller shape with 1s
2. If any dimension does not match, broadcast the dimension with size=1
3. If neither non-matching dimension is 1, raise an error.

`np.arange(3) + 5`

0	1	2
---	---	---

+

5

=

0	1	2
---	---	---

+

5	5	5
---	---	---

=

5	6	7
---	---	---

`np.ones((3,3)) + np.arange(3)`

1	1	1
1	1	1
1	1	1

+

0	1	2
---	---	---

=

1	1	1
1	1	1
1	1	1

+

0	1	2
0	1	2
0	1	2

=

1	2	3
1	2	3
1	2	3

`np.arange(3).reshape(3,1) + np.arange(3)`

0
1
2

+

0	1	2
---	---	---

=

0	0	0
1	1	1
2	2	2

+

0	1	2
0	1	2
0	1	2

=

0	1	2
1	2	3
2	3	4

Broadcasting Rules – 1D array

`np.arange(3) + 5`



1. If array shapes differ, left-pad the smaller shape with 1s

- 1) `shape=(3)` `shape=()`
- 2) `shape=(3)` `shape=(1)`
- 3) `shape=(3)` `shape=(3)`
- 4) **final** `shape=(3)`

2. If any dimension does not match, broadcast the dimension with size=1

3. If neither non-matching dimension is 1, raise an error.



Broadcasting Rules – 2D array (1)

`np.ones((3,3)) + np.arange(3)`

1	1	1		0	1	2		1	2	3
1	1	1	+	0	1	2	=	1	2	3
1	1	1		0	1	2		1	2	3

- 1) shape=(3,3) shape=(3,)
- 2) shape=(3,3) shape=(1,3)
- 3) shape=(3,3) shape=(3,3)
- final shape=(3,3)

1. If array shapes differ, left-pad the smaller shape with 1s
2. If any dimension does not match, broadcast the dimension with size=1
3. If neither non-matching dimension is 1, raise an error.

Broadcasting Rules – 2D array (2)

`np.arange(3).reshape(3,1) + np.arange(3)`

0	0	0		0	1	2		0	1	2
1	1	1	+	0	1	2	=	1	2	3
2	2	2		0	1	2		2	3	4

- 1) shape=(3,1) shape=(3)
- 2) shape=(3,1) shape=(1,3)
- 3) shape=(3,3) shape=(3,3)
- final shape=(3,3)

1. If array shapes differ, left-pad the smaller shape with 1s
2. If any dimension does not match, broadcast the dimension with size=1
3. If neither non-matching dimension is 1, raise an error.

Broadcasting Rules – Error

- The trailing axes of either arrays must be 1 or both must have the same size for broadcasting to occur. Otherwise, a **"ValueError: operands could not be broadcast together with shapes"** exception is thrown.

```
>>> a=np.arange(6).reshape(3,2)
```

```
>>> a
```

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

```
>>> b=np.arange(3)
```

```
>>> b
```

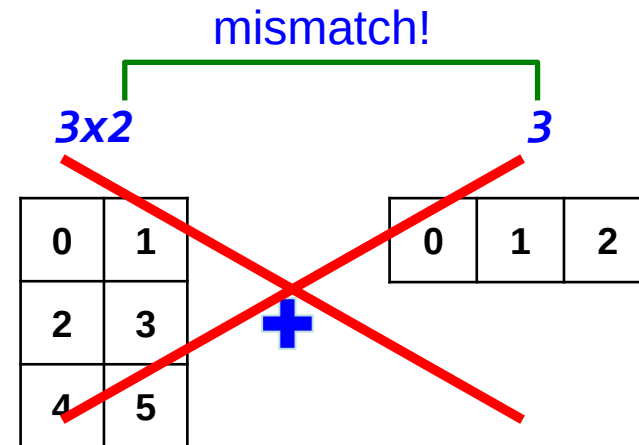
```
array([0, 1, 2])
```

```
>>> a+b
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ValueError: operands could not be broadcast together with shapes (3,2) (3,)
```



Slicing, Masking and Fancy Indexing

- See next few slides...

Array Slicing (1)

- `arr[lower:upper:step]`
- Extracts a portion of a sequence by specifying a lower and upper bound. The lower-bound element is included, but the upper-bound element is not included. Mathematically: `[lower, upper)`. The step value specifies the stride between elements.

```
# indices:           0  1  2  3  4
# negative indices:-5 -4 -3 -2 -1

>>> a = np.array([10,11,12,13,14])
# The following slicing results are the same
>>> a[1:3]
array([11, 12])
>>> a[1:-2]
array([11, 12])
>>> a[-4:3]
array([11, 12])
```

Array Slicing (2)

- **Omitting Indices:** omitted boundaries are assumed to be the beginning or end of the list, compare the following results

```
>>> a[:3] # first 3 elements
```

```
array([10, 11, 12])
```

```
>>> a[-2:] # last 2 elements
```

```
array([13, 14])
```

```
>>> a[1:] # from 1st element to the last
```

```
array([11, 12, 13, 14])
```

```
>>> a[:-1] # from 1st to the second to last
```

```
array([10, 11, 12, 13])
```

```
>>> a[:] # entire array
```

```
array([10, 11, 12, 13, 14])
```

```
>>> a[::2] # from 1st, every other element (even indices)
```

```
array([10, 12, 14])
```

```
>>> a[1::2] # from 2nd, every other element (odd indices)
```

```
array([11, 13])
```

Multidimensional Arrays

- A few 2D operations similar to the 1D operations shown above

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

```
>>> a
```

```
array([[ 0.,  1.,  2.,  3.],  
       [10., 11., 12., 13.]])
```

```
>>> a.shape # shape = (rows, columns)  
(2, 4)
```

```
>>> a.size # total elements in the array  
8
```

```
>>> a.ndim # number of dimensions  
2
```

```
>>> a[1,3] # reference a 2D array element  
13
```

```
>>> a[1,3] = -1 # set value of an array element
```

```
>>> a[1] # address second row using a single index  
array([10., 11., 12., -1.])
```


2D Array Slicing

```
>>> a = np.arange(1,26)
>>> a = a.reshape(5,5) # generate the 2D array
>>> a[0,3:5]
array([4, 5])
>>> a[0,3:4]
array([4])
>>> a[4:,4:]
array([[25]])
>>> a[3:,3:]
array([[19, 20],
       [24, 25]])
>>> a[:,2]
array([ 3,  8, 13, 18, 23])
>>> a[2::2,::2]
array([[11, 13, 15],
       [21, 23, 25]])
```

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

Slices Are References

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

```
>>> a = np.arange(5)
>>> a
array([0, 1, 2, 3, 4])
>>> b = a[2:4]
>>> b
array([2, 3])
>>> b[0]=7
>>> a
array([0, 1, 7, 3, 4])
```

Masking

```
>>> a=np.arange(10)
```

```
>>> a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

a	0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---	---

```
# creation of mask using ufunc
```

```
>>> mask=np.abs(a-5)>2
```

mask	1	1	1	0	0	0	0	0	1	1
------	---	---	---	---	---	---	---	---	---	---

```
>>> mask
```

```
array([ True,  True,  True, False, False, False, False, False,  True,
        True], dtype=bool)
```

```
>>> a[mask]
```

mask	0	1	0	1						
------	---	---	---	---	--	--	--	--	--	--

```
array([0, 1, 2, 8, 9])
```

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

```
# manual creation of mask
```

```
>>> mask
```

```
array([False,  True, False,  True], dtype=bool)
```

```
>>> a[mask]
```

```
array([1, 3])
```

Masking and where

```
>>> a=np.arange(8)**2
>>> a
array([ 0,  1,  4,  9, 16, 25, 36, 49])
>>> mask=np.abs(a-9)>5
>>> mask
array([ True,  True, False, False,  True,  True,  True,  True],
      dtype=bool)
# find the locations in array where expression is true
>>> np.where(mask)
(array([0, 1, 4, 5, 6, 7]),)
>>> loc=np.where(mask)
>>> a[loc]
array([ 0,  1, 16, 25, 36, 49])
```

Masking in 2D

```
>>> a=np.arange(25).reshape(5,5)+10
>>> a
array([[10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34]])
>>> mask=np.array([0,1,1,0,1],dtype=bool)
>>> a[mask]    # on rows, same as a[mask,:]
array([[15, 16, 17, 18, 19],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34]])
>>> a[:,mask] # on columns
array([[11, 12, 14],
       [16, 17, 19],
       [21, 22, 24],
       [26, 27, 29],
       [31, 32, 34]])
```

a[:,mask]				
0	1	1	0	1

a[mask]	0	10	11	12	13	14
	1	15	16	17	18	19
	1	20	21	22	23	24
	0	25	26	27	28	29
	1	30	31	32	33	34

Fancy Indexing - 1D

- NumPy offers more indexing facilities than regular Python sequences.
- In addition to indexing by integers and slices, arrays can be indexed by arrays of integers and arrays of Booleans (as seen before).

```
>>> a=np.arange(8)**2
>>> a
array([ 0,  1,  4,  9, 16, 25, 36, 49])
# indexing by position
>>> i=np.array([1,3,5,1])
>>> a[i]
array([ 1,  9, 25,  1])
>>> b=(np.arange(6)**2).reshape(2,-1)
>>> b
array([[ 0,  1,  4],
       [ 9, 16, 25]])
>>> i=[0,1,0]
>>> j=[0,2,1]
>>> b[i,j] # indexing 2D array by position
array([ 0, 25,  1])
```

Fancy Indexing - 2D

```
>>> b=(np.arange(12)**2).reshape(3,-1)
```

```
>>> b
```

```
array([[ 0,  1,  4,  9],
       [16, 25, 36, 49],
       [64, 81, 100, 121]])
```

```
>>> i=[0,2,1]
```

```
>>> j=[0,2,3]
```

```
# indexing 2D array
```

```
>>> b[i,j]
```

```
array([ 0, 100,  49])
```

```
# note the shape of the resulting array
```

```
>>> i=[[0,2],[2,1]]
```

```
>>> j=[[0,3],[3,1]]
```

```
# When an array of indices is used,
```

```
# the result has the same shape as the indices;
```

```
>>> b[i,j]
```

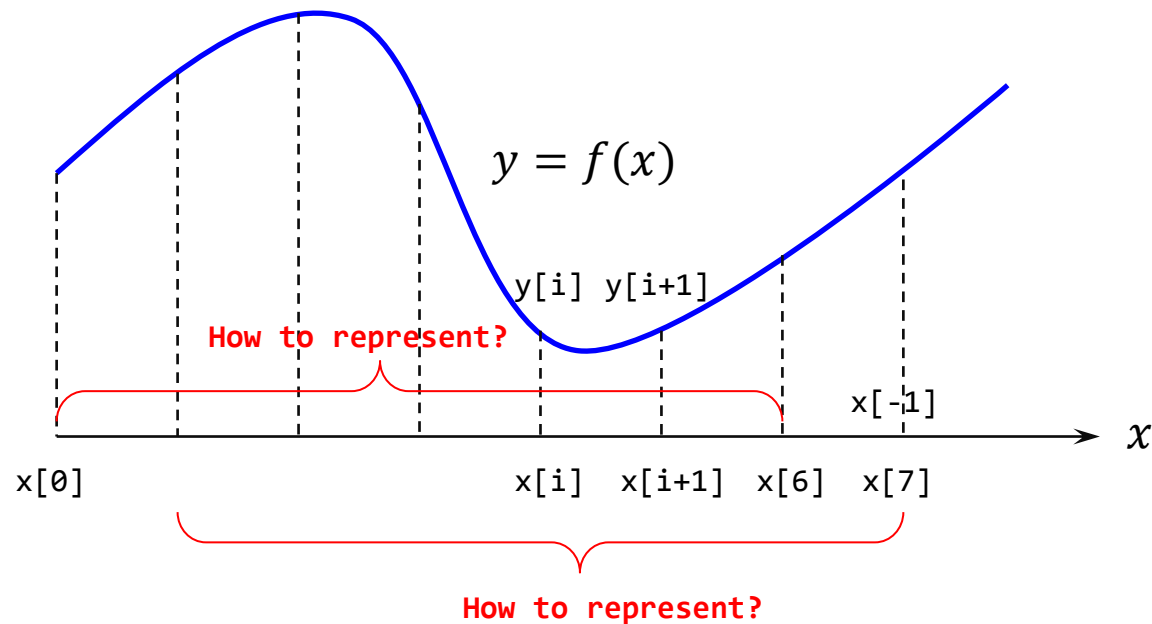
```
array([[ 0, 121],
       [121,  25]])
```

idx	0	1	2	3
0	0	1	4	9
1	16	25	36	49
2	64	81	100	121

Practical Python Programming

Calculate Derivative/Integration

Problem Description



➤ Numerical Derivative and Integration:

$$y' = \frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}$$

$$\int_a^b f(x) dx = \sum_{i=1}^N \frac{1}{2} (y_i + y_{i+1}) \cdot \Delta x$$

➤ How to get a vector of Δy and Δx ?

Calculate Derivative - Solution

➤ Using Numpy slicing:

```
import numpy as np
import matplotlib.pyplot as plt

# calculate the sin() function on evenly spaced data.
x = np.linspace(0, 2*np.pi, 101)
y = np.sin(x)

# use slicing to get dy and dx
dy = y[1:] - y[:-1]
dx = x[1:] - x[:-1]

dy_dx = dy/dx
```

Calculate Integration - Solution

```
import numpy as np
import matplotlib.pyplot as plt

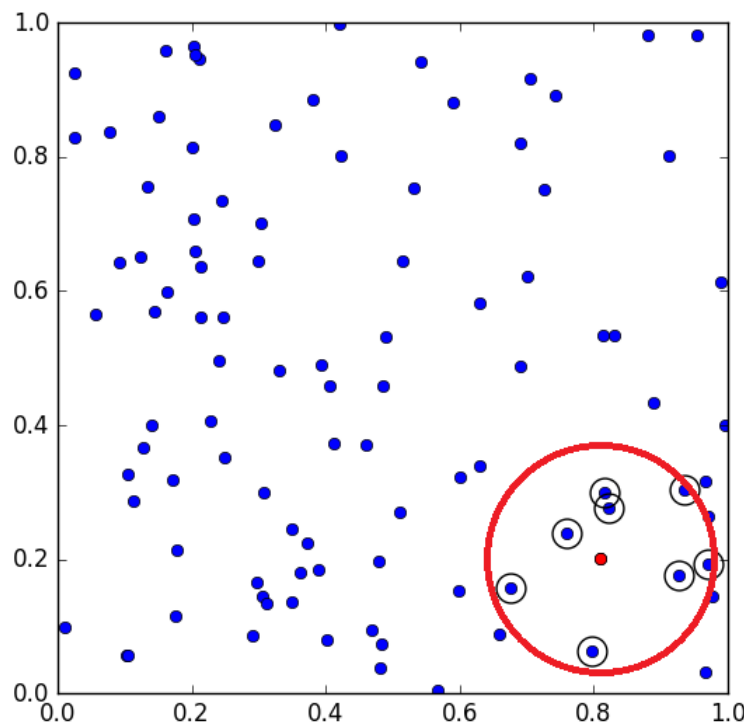
# calculate the sin() function on evenly spaced data.
x = np.linspace(0, 2*np.pi, 101)
y = np.sin(x)

# use slicing to get dy and dx
cx = 0.5*(x[1:]+x[:-1])
dy_by_2=0.5*(y[1:]+y[:-1])

# note the trapezoid rule with cumsum
area = np.cumsum(dx*dy_by_2)
analytical = -np.cos(x) + np.cos(0)
```

Example using Numpy: Nearest Neighbors

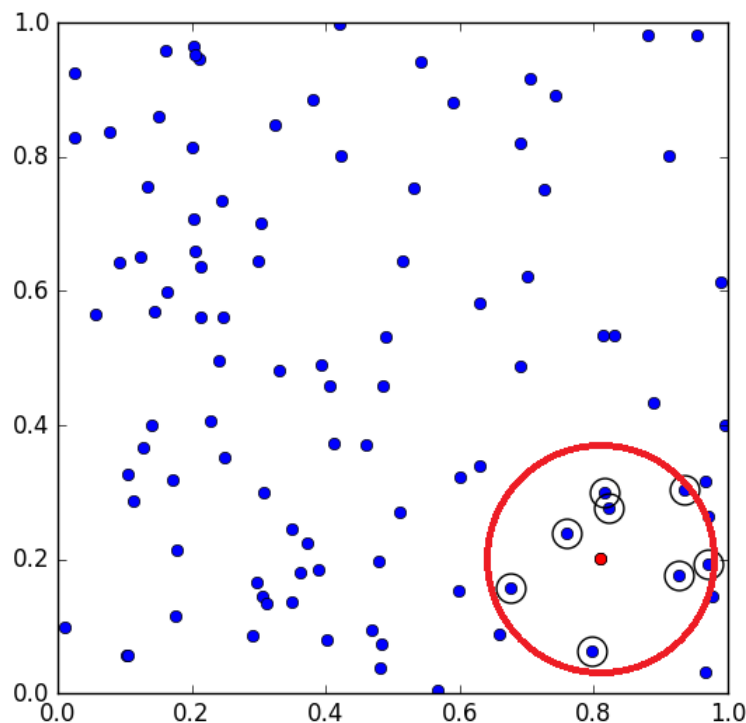
- A k-nearest neighbor search identifies the top k nearest neighbors to a query.
- The problem is: given a dataset D of vectors in a d -dimensional space and a query point x in the same space, find the closest point in D to x .
 - Molecular Dynamics
 - K-means clustering



Nearest Neighbors - Naive Implementation

➤ Using for loops...?

$$d_{i,j}^2 = (x_i - x_j)^2 + (y_i - y_j)^2$$



Nearest Neighbors - Better Implementation

```
# A better implementation
N=100
dim=2
# generate N random points in 2D
X = np.random.random((N,dim))
```

Nearest Neighbors - Pairwise differences

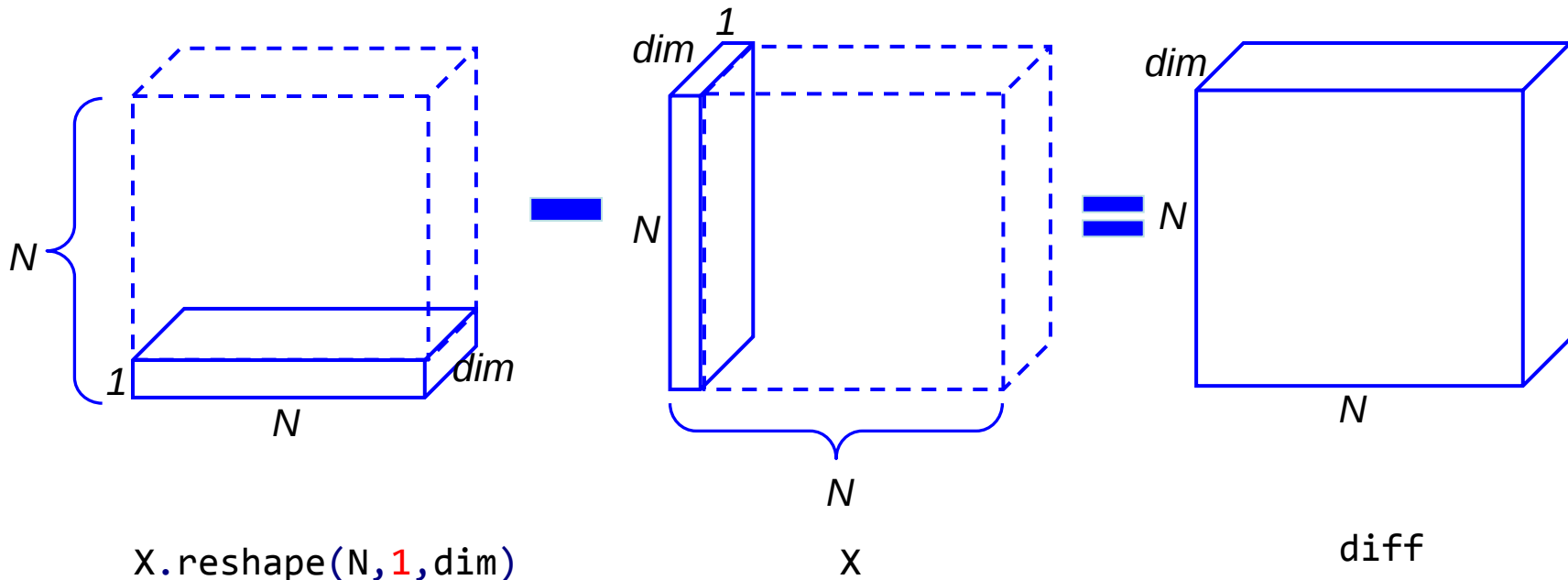
find pairwise difference using broadcasting

```
diff = X.reshape(N,1,dim)-X
```

(N,1,dim) (N,dim)

(N,1,dim) (1,N,dim)

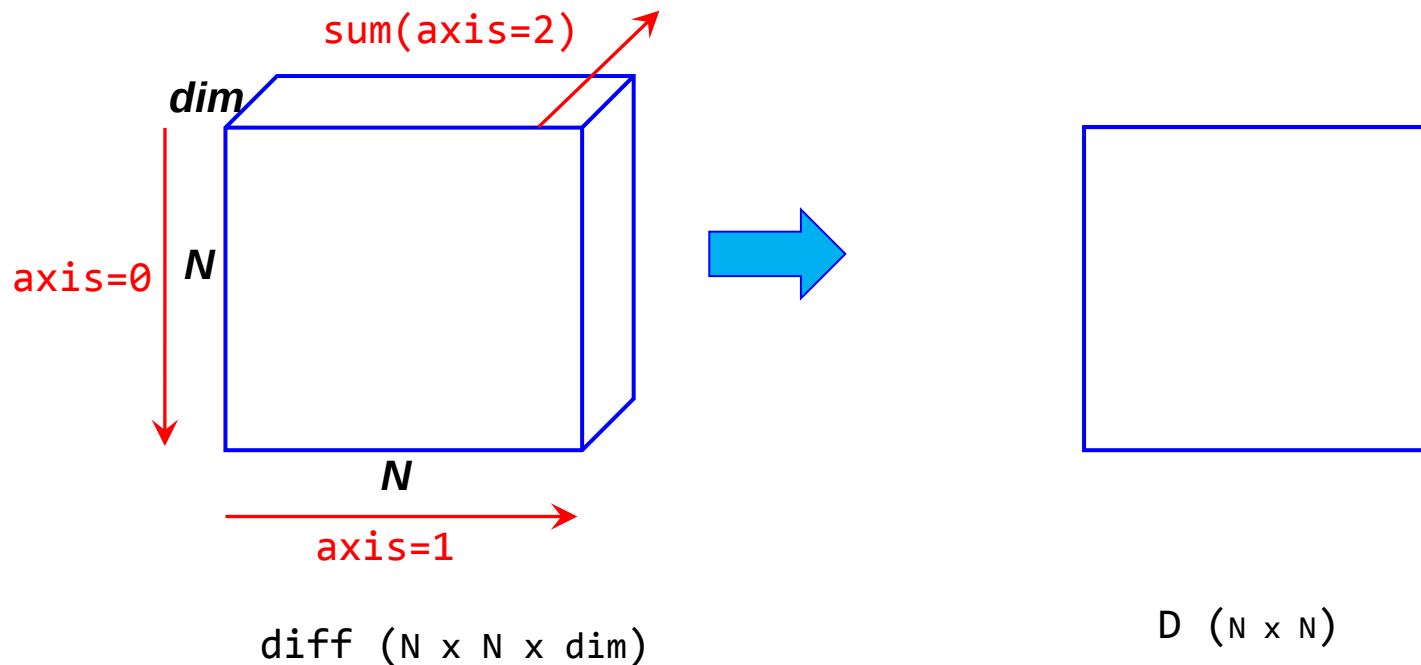
(N,N,dim) (N,N,dim)



Nearest Neighbors - argsum

Calculate the sum of diff using the aggregate function

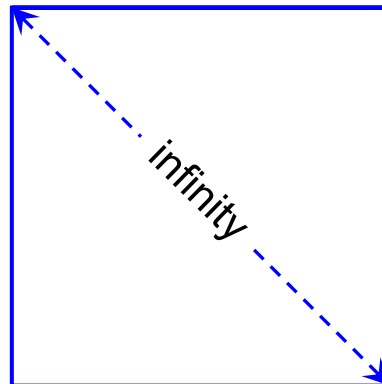
```
D = (diff**2).sum(axis=2)
```



Nearest Neighbors - Diagonal Values

- In order to avoid the self-self neighbors, set the diagonal values to infinity using numpy's pre-defined values

```
# set diagonal to infinity to skip self-neighbors  
i = np.arange(N)  
# using advanced (fancy) indexing  
D[i,i]=np.inf
```



$D (N \times N)$

Nearest Neighbors - Obtain the Indices

- An example of the D matrix (N=5)

```
[[          inf  0.06963122  0.44504047  0.04605534  0.36092231]
 [ 0.06963122          inf  0.23486059  0.06682903  0.31504998]
 [ 0.44504047  0.23486059          inf  0.23614707  0.12082747]
 [ 0.04605534  0.06682903  0.23614707          inf  0.14981131]
 [ 0.36092231  0.31504998  0.12082747  0.14981131          inf]]
```

- For $p=1$, the nearest neighbor is 3

Nearest Neighbors – k nearest neighbor

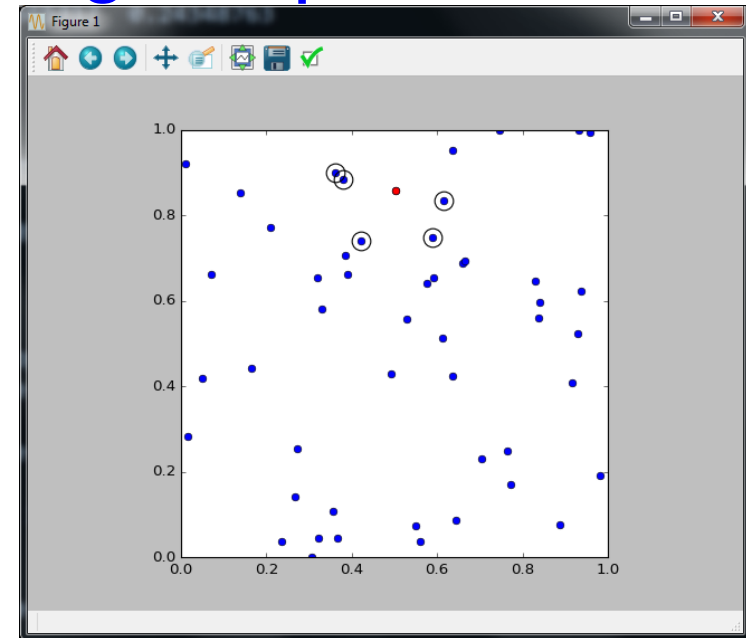
➤ **How to get the k nearest neighbors?**

this will return all the nearest neighbors matrix (nnm)

```
nnm = D.argsort(axis=1)
```

Verify Results using Matplotlib

```
# plot all the random points
plt.plot(X[:,0],X[:,1],'ob')
# plot pth point in red
p = N/2
plt.plot(X[p,0],X[p,1],'or')
k = 5
```



```
# plot k neighbors in circles
plt.plot(X[nm[p,:k],0],X[nm[p,:k],1],'o',markerfacecolor='None',marker
size=15,markeredgewidth=1)
# equalize the x and y scale
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

Practical Python Programming

Introducing Scipy

Numerical Methods with Scipy

- **Scipy package (SCientific PYthon) provides a multitude of numerical algorithms built on Numpy data structures**
- **Organized into subpackages covering different scientific computing areas**
- **A data-processing and prototyping environment almost rivaling MATLAB**

Major modules from scipy

➤ Available sub-packages include:

- constants: physical constants and conversion factors
- cluster: hierarchical clustering, vector quantization, K-means
- integrate: numerical integration routines
- interpolate: interpolation tools
- io: data input and output
- linalg: linear algebra routines
- ndimage: various functions for multi-dimensional image processing
- optimize: optimization algorithms including linear programming
- signal: signal processing tools
- sparse: sparse matrix and related algorithms
- spatial: KD-trees, nearest neighbors, distance functions
- special: special functions
- stats: statistical functions
- weave: tool for writing C/C++ code as Python multiline strings

Scipy Example: Integration

$$\int_1^3 x^2 dx = \frac{1}{3} x^3 \Big|_1^3$$

```
#!/usr/bin/env python
import scipy.integrate as integrate
import scipy.special as special
result_integ, err = integrate.quad(lambda x: x**2, 1, 3)
result_real = 1./3.*(3.**3-1**3)

print "result_real=", result_real
print "result_integ=", result_integ
```

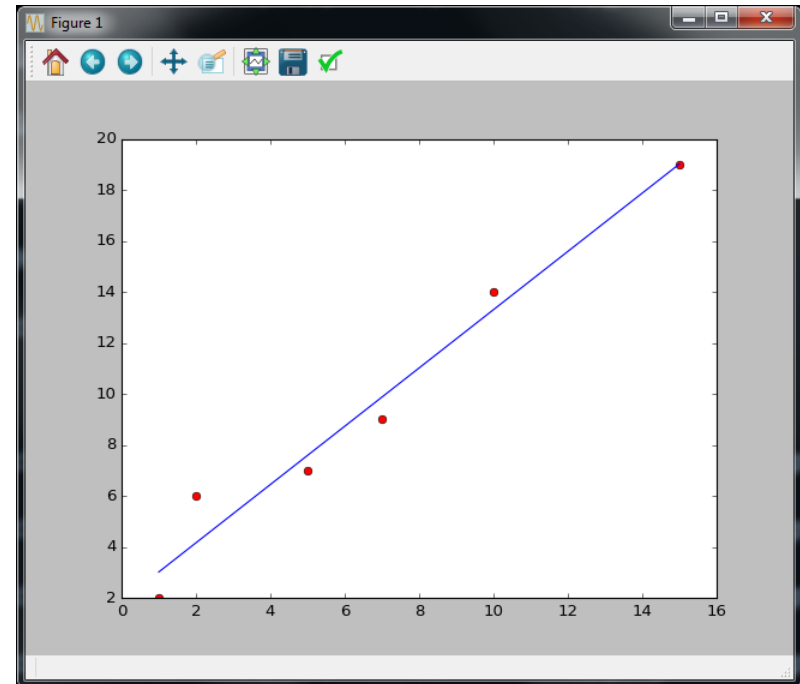

Scipy Example: Regression

```
#!/usr/bin/env python
```

```
from scipy import stats  
import numpy as np  
import matplotlib.pyplot as plt
```

```
x = np.array([1, 2, 5, 7, 10, 15])  
y = np.array([2, 6, 7, 9, 14, 19])  
slope, intercept, r_value, p_value, std_err = stats.linregress(x,y)
```

```
plt.plot(x,y,'or')  
yh = x*slope + intercept  
plt.plot(x, yh, '-b')  
plt.show()
```



Future Trainings

- **Next week training: Performance Analysis of Matlab Code**
 - Wednesdays October 26, 2016, Frey Computing Service Center 307
- **Programming/Parallel Programming workshops**
 - Usually in summer
- **Keep an eye on our webpage: www.hpc.lsu.edu**