Plotting and Numeric Tools: A Quick Survey

Special Topic in Computer Science
Python Programming

Introduction

- One of the strengths of python is its fantastic user base-in particular, the many useful modules that those users provide.
- In fact, this is exactly what open-source software is about-the sharing of effort for the good of all users.
- In this chapter, we take a brief look at the numpy and matplotlib modules

PLOTTING DATA WITH PYLAB

- In addition to drawing explicitly using tools like the turtle module, Python provides tools to plot data in both two and three dimensions.
- One of the most useful of those tools is the module pylab, which includes the plotting package matplotlib.
- Matplotlib is a plotting library, created in the image of the Matlab plotting library, for making publication quality figures.

MATPLOTLIB

- Matplotlib is a software package provided for creating graphs.
- matplotlib is *not* part of the standard Python distribution. It requires that you download some extra packages into your existing Python environment.
- Once downloaded and installed, matplotlib can be used in your Python code.
- The matplotlib website is http://matplotlib.sourceforge.net.

Getting matplotlib

- As is true for many open-software projects, matplotlib is hosted by SourceForge.
- SourceForge (http : //sourceforge.net) is the largest open-source collection on the Internet.
- As of September 2011, SourceForge reported hosting more 306,000 projects, with more than 2 million registered users.
- The matplotlib website is http://matplotlib.sourceforge.net.

Getting matplotlib (cont'd.)

- The simplest way to get matplotlib, and a lot of other tools not packaged with the standard Python distribution, is to get an *augmented* Python distribution that comes with all the extra tools preinstalled!
- In this way, you can not only get all of the standard Python distribution, and not only matplotlib, but also a host of other tools that you can explore as you wish.
- Two such distributions exist that can be downloaded off the Internet:
  - Enthought Python Distribution (EPD). The EPD (http:// www . enthought.com/products/epd.php) distribution contains approximately 100 extra modules in its distribution.
    - The download is free for educational use and requires a nominal fee otherwise. It is available for multiple platforms, including Windows, Macintosh, and Linux.
  - Python(x,y). Python(x,y) (http://code.google.com/p/ pythonxy) is a Python distribution focused on scientific and numeric computation, data visualization, and data analysis. It is specific to the Windows platform only.
- Installing one of these might save you a bit of time when it comes to adding modules later.

Getting matplotlib (cont'd.)

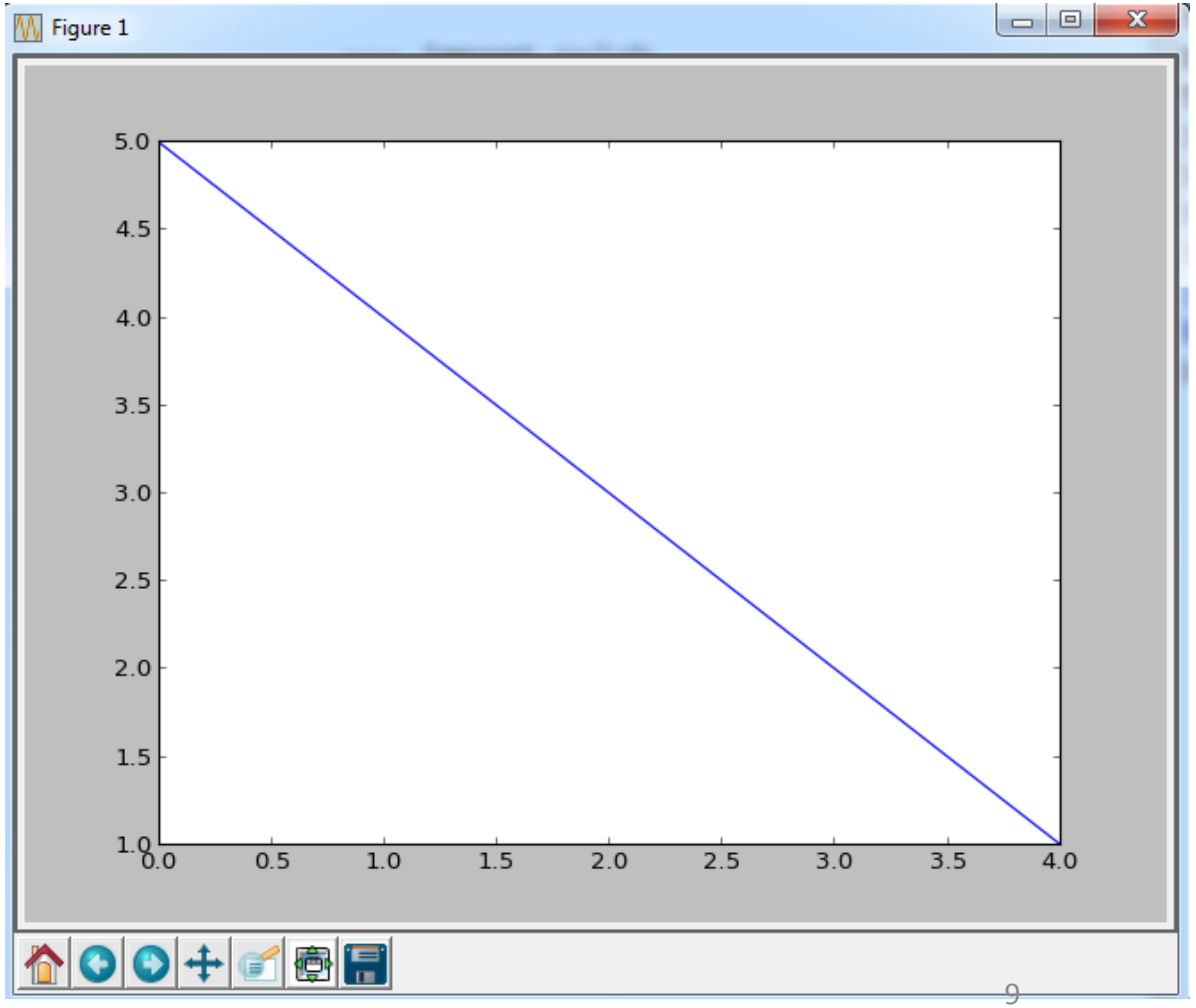
- The standard way to install any Python module is to begin with the standard Python distribution, then add modules as you need them by downloading the module from the Internet.
- One problem with this approach is called a *dependency.* Matplotlib requires that you download the Numeric Python (always shortened to numpy) package first.
- One other thing to worry about when downloading a Python module off the web is getting the right one, which means two things:
  - You need to get the correct version for your platform.
  - You need to get the correct Python version for the Python on your machine.

Getting matplotlib (cont'd.)

- To download NumPy, go to the NumPyweb site (http://numpy.scipy.org), select the Download link, and download the correct platform and version number for your computer.
- Before you go any further, it would be good to make sure that NumPy was installed properly.
- Open up Python (through IDLE or however you typically do it), and type the following into the shell:
- >>> import numpy

Getting matplotlib (cont'd.)

- To download matplotlib, go to the matplotlib website (http:*//* matplotlib.sourceforge.net) and, as with NumPy, download the proper matplotlib for your platform and Python version.
- *As* with NumPy, install matplodib.
- Try the following to see if your matplotlib installed properly:
  - >>> import pylab
  - >>> pylab.plot([5,4,3,2,1])
  - >>> pylab.show()



W0RKING WITH MATPLOTLIB

- There are some wonderful repositories of more complicated examples at the matplotlib website.
- In particular, the gallery (http://matplotlib. source forge.net/gallery. html) provides a number of beautiful examples of what you can do with matplotlib!

plot Command

- The plot command allows you to make two-dimensional line drawings.
- We first *import* pylab, then we can do plots of the following kinds:
  - pylab.plot (x_vals, y_vals): Assumes two lists of numeric values of equal length. Plot with default properties.
  - pylab.plot (y_vals): Creates *x* values in range of 0 to len (y_vals) -1.
  - pylab.plot (x_vals, y_vals, *'bo'):* Plots the *x* and *y* values, modifies the line type to blue circles .
  - pylab.plot(x_vals,y_vals, 'gx', x_vals2, y_vals2, 'y+'): Plots with two graphs in the same window. First plot is LValS vs. y_vals with green *x's,* second plot is x_vals2 vs. y_vals2 with yellow *+'s.*
- See the online documentation for all the variations on color and line styles.

First Plot and Using a List

- As with many things in Python, plotting requires that you use one of the collection data structures called the *list* data structure
- The values in list_ofints provide the values for the y-axis of the plot.
- The pylab. plot method provides the x-axis values as the sequence indices of the list.
- The first value in the list gets x-value 0, the second value gets x-value 1, and so on.

```
import pylab
list_of_ints = []
for counter in range(10):
    list_of_ints.append(counter*2)

print(list_of_ints)
print(len(list_of_ints))

# now plot the list
pylab.plot(list_of_ints)
pylab.show()
```

More Interesting Plot: A Sine Wave

- You can plot using all kinds of "markers" for each point in the graph.
- Each "formatting string" consists of a two-character string:
  - Color: The first is a lowercase letter for the color. The mappings are fairly obvious: "r" is red, "b" is blue, "g" is green. The only odd one is that "k" is black.
  - Marker: The second element is a single character indicating the type of marker. These are rather obscure-you simply have to learn them. For example: "0" is for circles, "." is for dots, "x" is for an x marker, and "+" is for a plus marker.

```
# plot a sine wave from 0 to 4pi

import math
import pylab

#initialize the two lists and the counting variable num. Note is a float
y_values = []
x_values = []
number = 0.0

#collect both number and the sine of number in a list
while number < math.pi * 4:
    y_values.append(math.sin(number))
    x_values.append(number)
    number += 0.1

#plot the x and y values as red circles
pylab.plot(x_values,y_values,'ro')
pylab.show()
```

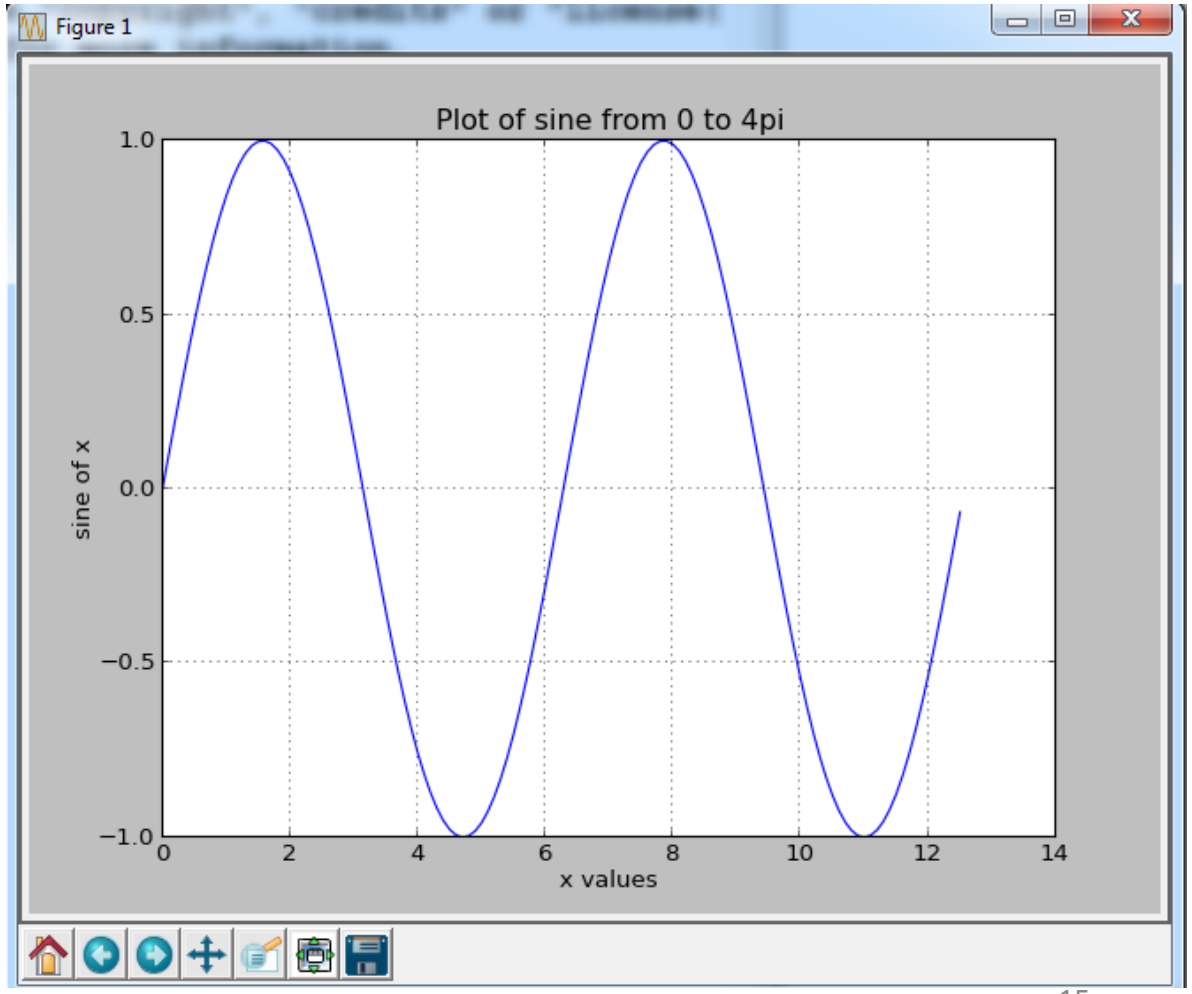
Plot Properties

- There are a number of properties you can attach to a plot. Here are some simple examples:
  - The x-axis can be labeled in your plot using the pylab.xlabel ('a_str') command. The provided string will be used as an x-axis label.
  - The y-axis can also be labeled in your plot using the pylab.ylabel (, *a_str'*) command. Again the string will be used and printed vertically along the y-axis.
  - The figure can be given a title (which shows up just above the plot) using the command pylab.title (' *a_str').*
  - The figure can have a grid superimposed on the plot area. The command is pylab.grid (*True)* .

Plot Properties (cont'd.)

```
import math
import pylab
import numpy

# use numpy arange to get an array of float values.
x_values = numpy.arange(0,math.pi*4,0.1)
y_values = [math.sin(x) for x in x_values]
pylab.plot(x_values,y_values)
pylab.xlabel('x values')
pylab.ylabel('sine of x')
pylab.title('Plot of sine from 0 to 4pi')
pylab.grid(True)
pylab.show()
```



Bar Graphs

- The bar command also takes a number of arguments.
  - The first are the tick locations, typically a NumPy *arange.*
  - The second is the list of values to plot.
  - The remaining arguments are optional, such as the width and the color arguments.

Histograms

- A histogram is a graph that shows the number of elements within a range of values.
- The range, called a *bin,* is used uniformly for all the data in the histogram. Thus, the x-axis indicates the various bins and the y-axis the number of elements in each *bin.*
- The command for a histogram is pylab.hist.
- There are a number of arguments that can be provided, but the two most important are the list of values and the number of bins.

Histograms (cont'd.)

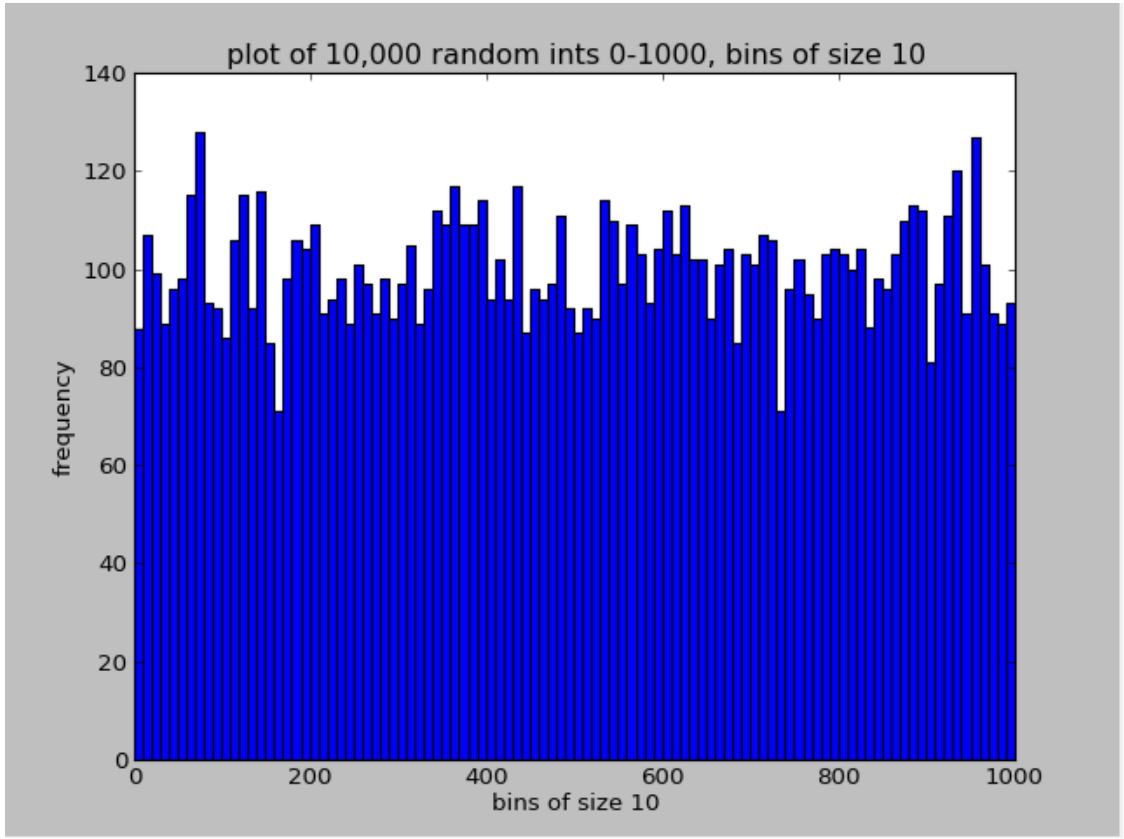
- For example, the program generates 10,000 values, randomly distributed between 0 and 1000.
- We allow for 100 bins (thus each bin is of size 10, 100 bins for a range from 0-1000).
- Note that the distribution is fairly, but not exactly, equal in all bins.

Histograms (cont'd.)

```
import pylab
import random

x_values = [random.randint(0,1000) for x in range(10000)]

pylab.hist(x_values,100)
pylab.xlabel('bins of size 10')
pylab.ylabel('frequency')
pylab.title('plot of 10,000 random ints 0-1000, bins of size 10')
pylab.show()
```



Pie Charts

- Pie charts show the percentage of the whole that each value in the list occupies.
- The command is pylab. pie(values, ...) , and plots each element in the values list as a "slice" whose size is proportional to that element's percentage of the whole (the sum of all the values).
- It takes a number of optional arguments, but two very useful ones are the following:
  - colors= (...) : a list of strings, each a single letter indicating a color. This is the progression of colors in the pie chart. The length of this list must be the same as the number of values in the chart.
  - labels= (. ..) : a list of strings, each one a label for its corresponding pie value. It also must have the same number of entries as the number of values being plotted.

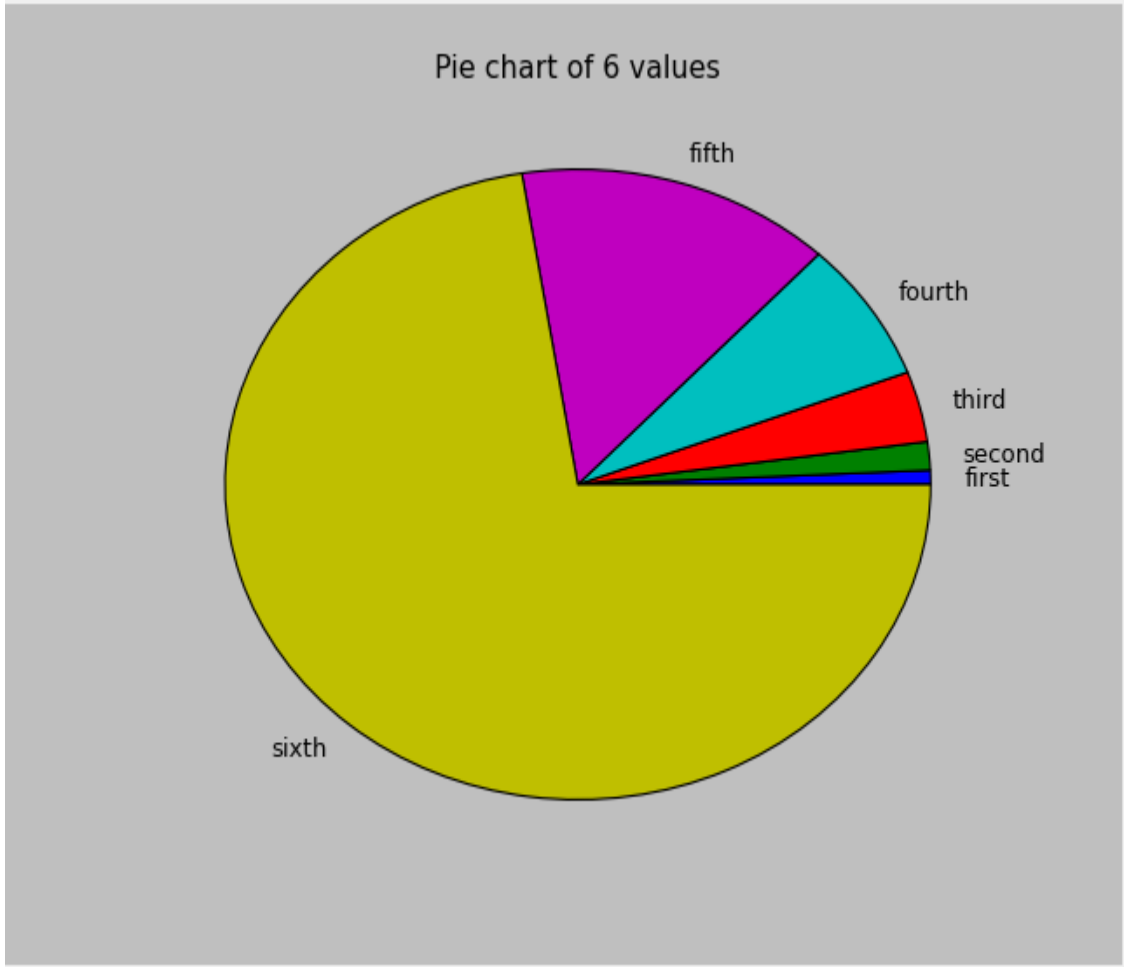
Pie Charts (cont'd.)

```
import pylab

values = [10,20, 50,100,200,1000]
pieLabels = ['first','second','third','fourth','fifth','sixth']

# these are the default colors. You get these if you do not provide any
colorLst = ('b', 'g', 'r', 'c', 'm', 'y', 'k', 'w')

pylab.pie(values,labels=pieLabels,colors=colorLst)
pylab.title('Pie chart of 6 values')
pylab.show()
```



NUMERIC PYTHON (NUMPY)

- Numeric Python, or NumPy, is the base module for scientific, mathematical, and engineering computing in Python.
- It implements a special *array* data type that is efficient, can be manipulated easily, and interfaces with other programming languages (C, C++, Fortran, and others).

Arrays Are Not Lists

- The basic unit of NumPy is the array. An array is a sequence, similar to a list, but it differs in two important ways:
  - Arrays can consist only of numbers.
  - The *type* of the numbers must be the same throughout the entire array.
- Arrays can be indexed, sliced, and iterated through, just like lists, but only with numbers and only with the same type of number (integer or floats) throughout the entire array.

Creating a NumPy Array

- An array is created by one of two general methods:
- • Using the array constructor. The array constructor takes a *single* sequence of same type, numeric elements and creates an array-for example aray = numpy.array ([10,20,30]).
  - NumPy provides a separate range function called arange (). Like *range,* it takes three potential arguments (the begin value, the end value, and the increment value), but it differs in that it can also work with floats. For example: aray = arange (0, math.pi * 4, 0.1) generates the sequence from 0 to $4 * pi$ by increments of 0.1.
  - Two special array constructors are zeros and ones. They create arrays of all 0s or all 1s. The argument can be a type of *n* values, and an n-dimensional array will be created.

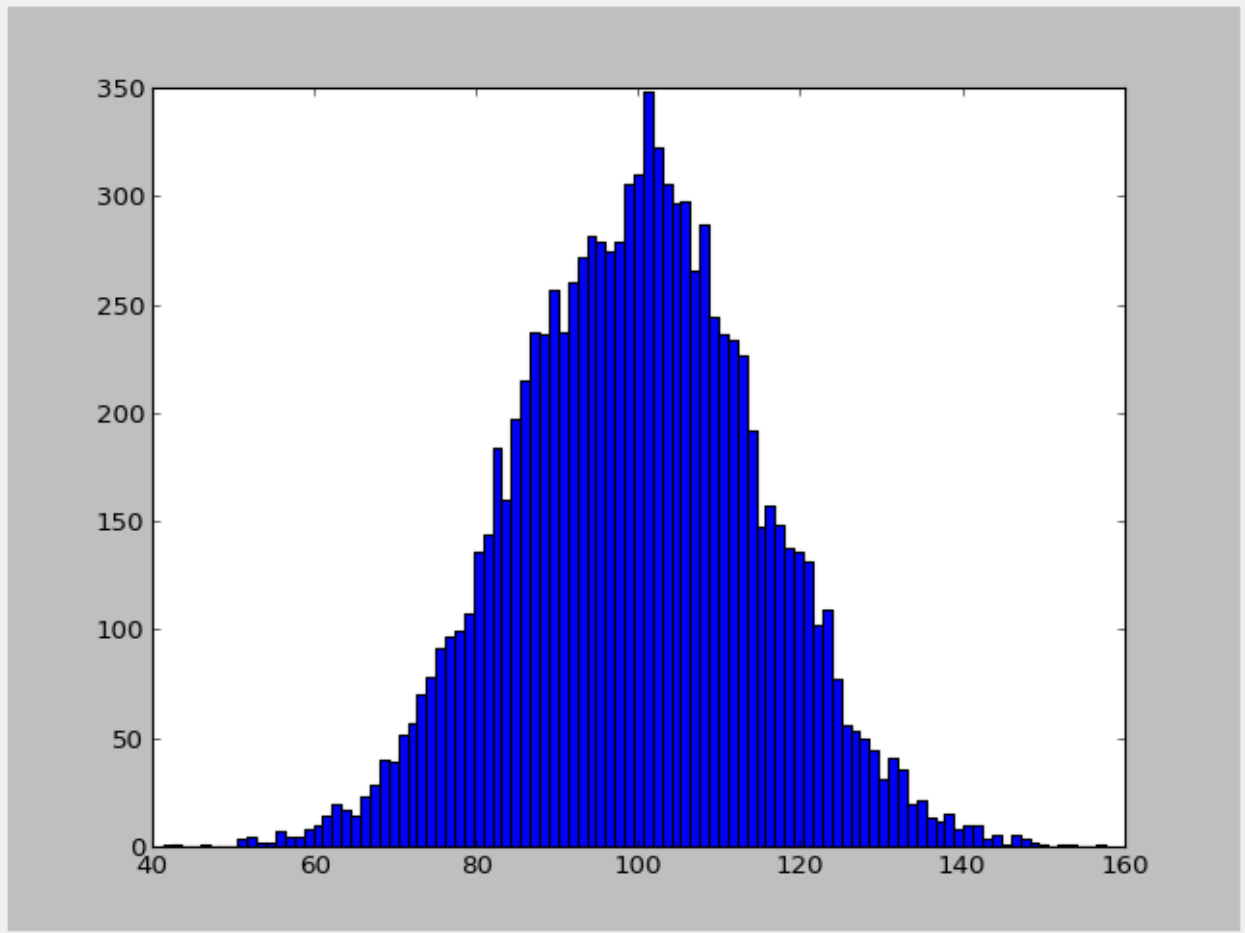
Manipulating Arrays

- Arrays can be manipulated in ways that lists cannot. Because the type of an array is fixed at its creation (a float or integer array), you can perform mathematical operations on the entire array using an approach called *broadcasting.*
- By broadcasting, NumPy means that a single operation is "broadcast" across the entire array.
- For example:
  - my _array * 4 multiplies each element of the NumPy array ny_array by 4. The other operations of +, -, / would produce similar results.
  - my _array + my _array adds each element of my _ar ray together, creating a third array of values. The two NumPy arrays must be of the same "shape," which for us really means of the same dimension (length).

Manipulating Arrays (cont'd.)

```
import numpy
import pylab

vals = numpy.ones((10000))
vals = vals * 100
# standard distribution, 10,000 elements, mu=0, std=1
sigma = numpy.random.standard_normal(10000)
sigma = sigma*15
#generate a standard distribution, mu=100, std-15
vals = vals + sigma
pylab.hist(vals,100)
pylab.show()
```



NumPy Arrays

- matplotlib relies on another Python module called numpy, short for "numeric Python."
- The numpy module provides, in a single module, the following capabilities:
  - A new data type, the *array* object
  - Support for foating-point data types
  - Support functions for foating-point values
- The array data type and its associated methods and functions are particularly useful.
- In fact, matplotlib works not with lists but numpy arrays. If you provide a list as an argument, matplotlib converts it to an array.

NumPy Arrays (cont’d.)

- numpy array can contain only the same data type, by default a foating-point number.
- It is type-restricted so that foating-point operations can be done more efficiently on the array.
- There are a number of ways to make an array. The array constructor can take a list object and convert it to an array. The elements of the list must consist only of numbers.
- The resulting array object will consist of numbers all of the same type.
- If mixed types exist (floats and ints), then all numbers will be converted to floats.
- When printed, an array has the string "array" printed as part of the object.
- One can append onto the end of an existing array using the numpy function append.

Arrays and range

- Very much like the *range* function, the numpy arange function generates a range of values.
- The difference is that the values are Boating-point values instead of integers.
- The arange takes three arguments: the begin value *(float),* the end value *(float),* and an increment *(floa*t).

```
>>> import numpy
>>> my_array = numpy.array([1,2,3,4])
>>> my_array
array([1, 2, 3, 4])                    # all integers
>>> numpy.append(my_array,50.0)
array([  1.,   2.,   3.,   4.,  50.])  # append a float, now all floats
>>> new_array = numpy.arange(0,2,0.1)  # new array using arange
>>> new_array
array([ 0.,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1.,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9])
```

Arrays and range (cont'd.)

- One of the best features of using arrays is the ability to perform operations between two arrays using standard arithmetic operators.
- This means that the operators are overloaded for arrays. For example, if you were to multiply an array by a floating-point number, then *every* element of the array would be multiplied by that number, yielding a new array.
- The same will occur if you apply a function to an array. *Every* element of the array has the function applied, yielding a new array.

Arrays and range (cont'd.)

```
>>> import numpy
>>> my_array = numpy.arange(0,6.3,0.1)
>>> my_array
array([0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
       1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
       2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
       3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
       4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
       5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2])
>>> new_array = my_array * 2
>>> new_array
array([  0. ,   0.2,   0.4,   0.6,   0.8,   1. ,   1.2,   1.4,   1.6,
         1.8,   2. ,   2.2,   2.4,   2.6,   2.8,   3. ,   3.2,   3.4,
         3.6,   3.8,   4. ,   4.2,   4.4,   4.6,   4.8,   5. ,   5.2,
         5.4,   5.6,   5.8,   6. ,   6.2,   6.4,   6.6,   6.8,   7. ,
         7.2,   7.4,   7.6,   7.8,   8. ,   8.2,   8.4,   8.6,   8.8,
         9. ,   9.2,   9.4,   9.6,   9.8,  10. ,  10.2,  10.4,  10.6,
        10.8,  11. ,  11.2,  11.4,  11.6,  11.8,  12. ,  12.2,  12.4])
```

Arrays and range (cont'd.)

```
>>> new_array = numpy.sin(my_array)
>>> new_array
array([ 0.        ,  0.09983342,  0.19866933,  0.29552021,  0.38941834,
        0.47942554,  0.56464247,  0.64421769,  0.71735609,  0.78332691,
        0.84147098,  0.89120736,  0.93203909,  0.96355819,  0.98544973,
        0.99749499,  0.9995736 ,  0.99166481,  0.97384763,  0.94630009,
        0.90929743,  0.86320937,  0.8084964 ,  0.74570521,  0.67546318,
        0.59847214,  0.51550137,  0.42737988,  0.33498815,  0.23924933,
        0.14112001,  0.04158066, -0.05837414, -0.15774569, -0.2555411 ,
       -0.35078323, -0.44252044, -0.52983614, -0.61185789, -0.68776616,
       -0.7568025 , -0.81827711, -0.87157577, -0.91616594, -0.95160207,
       -0.97753012, -0.993691  , -0.99992326, -0.99616461, -0.98245261,
       -0.95892427, -0.92581468, -0.88345466, -0.83226744, -0.77276449,
       -0.70554033, -0.63126664, -0.55068554, -0.46460218, -0.37387666,
       -0.2794155 , -0.1821625 , -0.0830894 ])
```

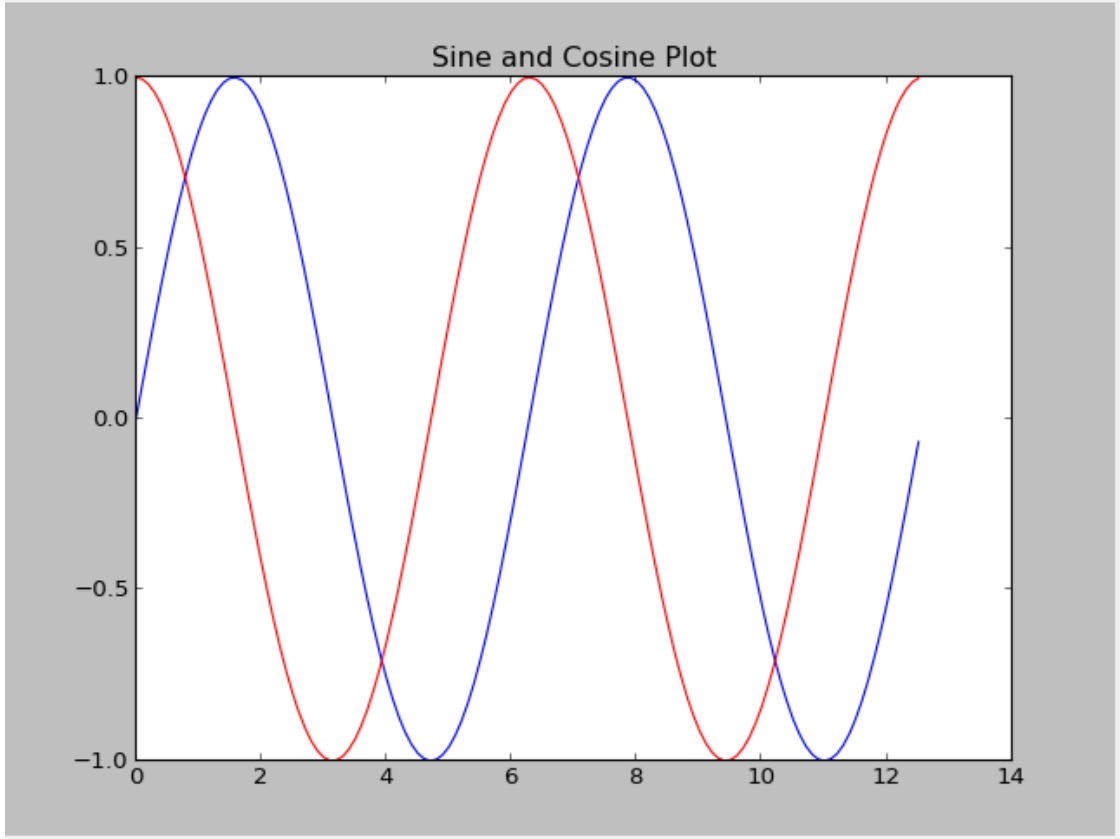
Arrays and range (cont'd.)

Notice that we can place as many plots as we like in the figure before we show it.

```
import numpy
import pylab

# Generate lists of points for both sine and cosine
x_values = numpy.arange(0, 4*numpy.pi, 0.1)
y1_values = numpy.sin(x_values)
y2_values = numpy.cos(x_values)

# Plot two curves on the same graph
pylab.title('Sine and Cosine Plot')
pylab.plot(x_values,y1_values,'b')
pylab.plot(x_values,y2_values, 'r')
pylab.show()
```



BAR GRAPH OF WORD FREQUENCY

- Having constructed the dictionary of word-frequency pairs, it would be interesting to plot the words versus their frequency to visualize the results.
- Matplotlib provides a bar command to plot bar charts, so that is a natural choice.
- There are three general steps to make the bar chart in this case.
  - The first is to set up the x-axis to use the words as the labels.
  - The second is to get the dictionary data gathered together in the appropriate format for plotting.
  - The third is to actually make the plot.
- Matplotlib provides a command to orient and label each x-axis entry. These entries are often called "ticks," so the commands to label them are called xticks and yticks.
- If you do not provide such a command, matplotlib will label the ticks with the appropriate number.
- However, you have the option to say what each tick label should be and where it should be located.

BAR GRAPH OF WORD FREQUENCY (cont'd.)

- We set a width bar_width for the width of our bars in the graph.
- We create a list with numbers from 0 to the size of our key 1ist.
- We use the numPy function arange, similar to range, to generate the list. This will be the number of xticks on our graph.
- The xticks command takes three arguments.
  - The first is the location of the tick being placed. We are using a numPy array to indicate the position of each tick, and to each of those ticks we add the tick position to the bar_width divided by 2. Because it is a numPy array, the addition-division is done to each element of the xVals array.
  - The second argument is the list of labels, the key list.
  - The third is an option that specifies to rotate each label 45 degrees.

BAR GRAPH OF WORD FREQUENCY (cont'd.)

- The bar command takes four arguments here.
  - The first is the xtick locations. The labels were set previously by the xticks command.
  - The second is the list of values to plot, the value_list.
  - The last two are plot options, the bar width and the bar color. The show command then draws the plot.

BAR GRAPH OF WORD FREQUENCY (cont'd.)

```
import numpy
import pylab
def bar_graph(word_count_dict):
    '''bar graph of word-frequency, xaxis labeled with words'''
    # collect key and value list for plotting
    word_list = []
    for key,val in word_count_dict.items():
        word_list.append((key,val))
    word_list.sort()
    key_list = [key for key,val in word_list]
    value_list = [val for key,val in word_list]
    # get ticks as the keys/words
    bar_width=0.5
    x_values = numpy.arange(len(key_list))
    pylab.xticks(x_values+bar_width/2.0,key_list,rotation=45)
    # create the bar graph
    pylab.bar(x_values,value_list,width=bar_width,color='r')
    pylab.show()
```