Math 246 Unit 5: Plotting graphs, with matplotlib

Brenton LeMesurier

September 24, 2015, revised and expanded after today's lab

Note: this was updated after the lab, with answers to some questions that came up. Also, I have added more information about the command for getting figures to appear in separate windows rather than inline: the "tk" option should work for both Mac OS and Windows.

Introduction: Matplotlib and Pyplot

Numerical data is often presented with graphs, and the tools we use for this come from the module `matplotlib.pyplot` which is part of the Python *package* `matplotlib`. (A package is essentially a collection of modules.)

Sources on Matplotlib

Matplotlib is a huge collection of graphics tools, of which we see just a few here. For more information, the home site for Matplotlib is

http://matplotlib.org (http://matplotlib.org)

and the section on pyplot is at

http://matplotlib.org/1.3.1/api/pyplot_api.html (http://matplotlib.org/1.3.1/api/pyplot_api.html)

However, another site that I find easier as an introduction is

http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html (http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html)

In fact, that whole site

http://scipy-lectures.github.io (http://scipy-lectures.github.io)

is quite useful.

Choosing where the graphs appear

First, we request that the graphs produced by `matplotlib.pyplot` appear "inline"; that is, within this notebook window:

In [1]:

```
%matplotlib inline
```

This is an IPython *magic command* - you can read more about them at

https://ipython.org/ipython-doc/dev/interactive/magics.html (https://ipython.org/ipython-doc/dev/interactive/magics.html)

Alternatively, one could have graphs appear in separate windows, which is useful when you want to save them to files, or zoom and pan around the image. That is requested with

```
%matplotlib tk
```

(As far as I know this works for Windows and Linux as well as Mac OS; let me know!)

We need some `numpy` stuff to create arrays of numbers to plot:

In [2]:

```
from numpy import linspace, sin, cos, pi
```

and for now, just the one main `matplotlib` graphics function, `plot`

In [3]:

```
from matplotlib.pyplot import plot
```

Producing arrays of "x" values with the Numpy function `linspace`

To plot the graph of a function, we first need a collection of values for the abscissa (horizontal axis). The function `linspace` gives an array containing a specified number of equally spaced values over a specified interval, so that

In [4]:

```
tenvalues = linspace(1., 7., 10)
```

gives ten equally spaced values ranging from 1 to 7:

In [5]:

```
print("array tenvalues is", tenvalues)
```

```
array tenvalues is [ 1.          1.66666667  2.33333333  3.
3.66666667  4.33333333
  5.          5.66666667  6.33333333  7.        ]
```

Not quite what you expected? To get values with ten *intervals* in between them, you need 11 values:

In [6]:

```
tenintervals = linspace(1., 7., 11)
print("array tenintervals is", tenintervals)
```

```
array tenintervals is [ 1.   1.6  2.2  2.8  3.4  4.   4.6  5.2  5.8
6.4  7. ]
```

Basic graphs with `plot`

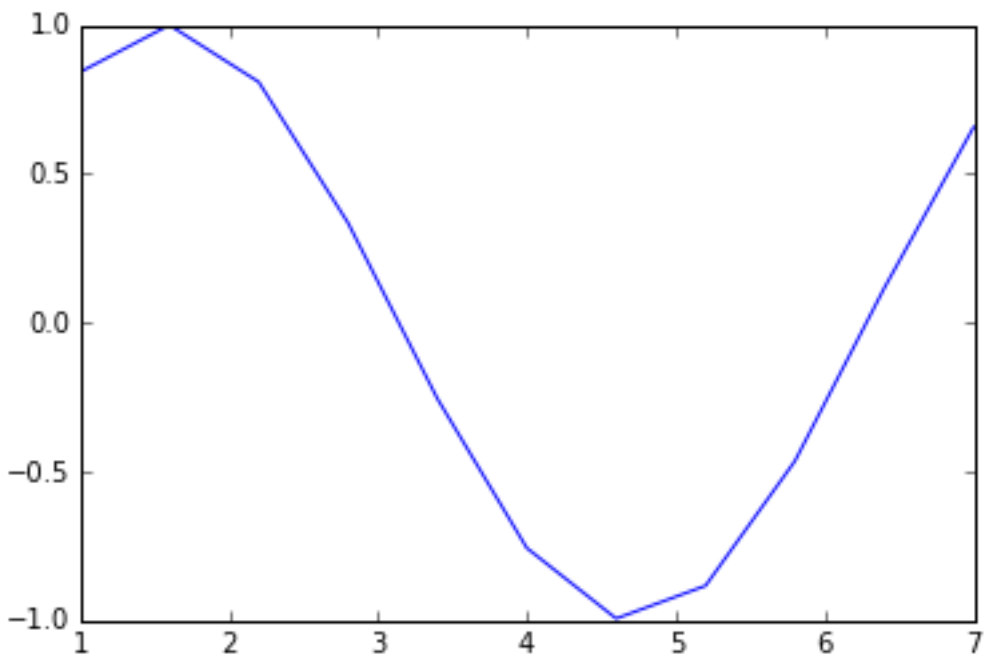
We could use these 11 values to graph a function, but the result is a bit rough, because the given points are joined with straight line segments:

In [7]:

```
plot(tenintervals, sin(tenintervals))
```

Out[7]:

```
[<matplotlib.lines.Line2D at 0x106928630>]
```



Here we see the default behavior of joining the given points with straight lines.

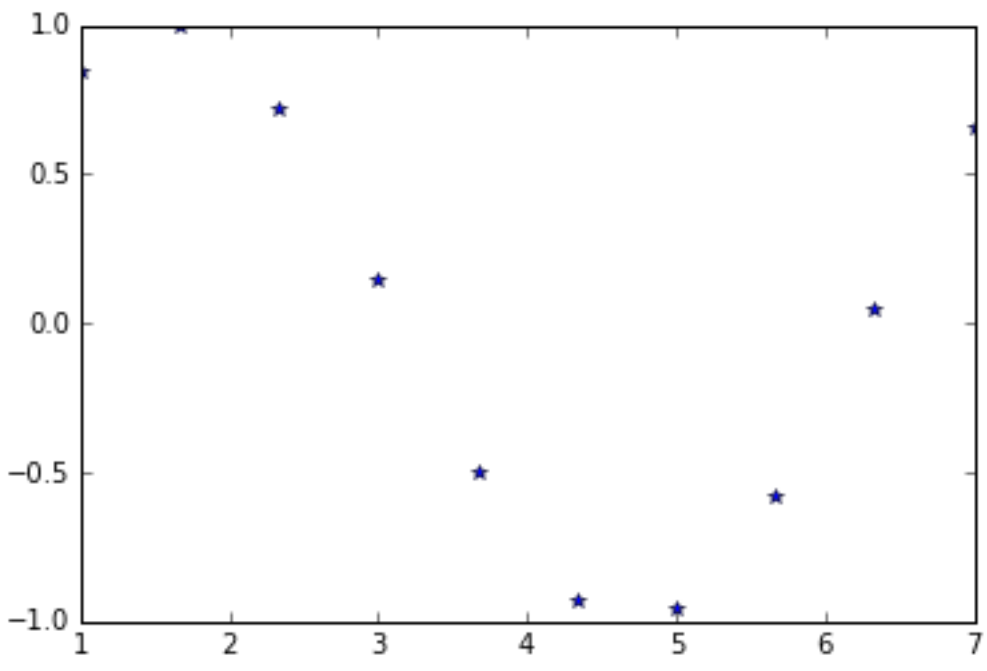
For discrete data it might be better to display each point wiht a marker, unconnected. This is done by adding a third argument, a text string specifying a marker, such as a star:

In [8]:

```
plot(tenvalues, sin(tenvalues), '*')
```

Out[8]:

```
[<matplotlib.lines.Line2D at 0x106b27b38>]
```



Smoother graphs

It turns out that 50 points is often a good choice for a smooth-looking curve, so the function `linspace` has this as a *default input parameter*: you can omit that third input value, and get 50 points.

Let's use this to plot some trig. functions.

In [9]:

```
x = linspace(-pi, pi)
print(x)
```

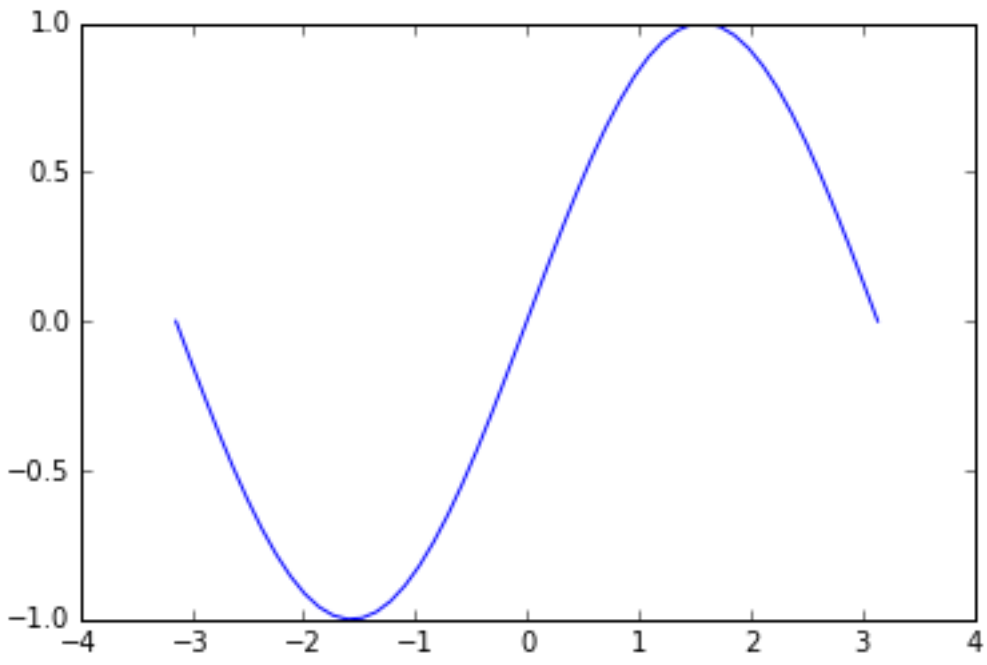
```
[-3.14159265 -3.01336438 -2.88513611 -2.75690784 -2.62867957 -2.5004
513
 -2.37222302 -2.24399475 -2.11576648 -1.98753821 -1.85930994 -1.7310
8167
 -1.60285339 -1.47462512 -1.34639685 -1.21816858 -1.08994031 -0.9617
1204
 -0.83348377 -0.70525549 -0.57702722 -0.44879895 -0.32057068 -0.1923
4241
 -0.06411414  0.06411414  0.19234241  0.32057068  0.44879895  0.5770
2722
  0.70525549  0.83348377  0.96171204  1.08994031  1.21816858  1.3463
9685
  1.47462512  1.60285339  1.73108167  1.85930994  1.98753821  2.1157
6648
  2.24399475  2.37222302  2.5004513   2.62867957  2.75690784  2.8851
3611
  3.01336438  3.14159265]
```

In [10]:

```
plot(x, sin(x))
```

Out[10]:

```
[<matplotlib.lines.Line2D at 0x106bf3278>]
```

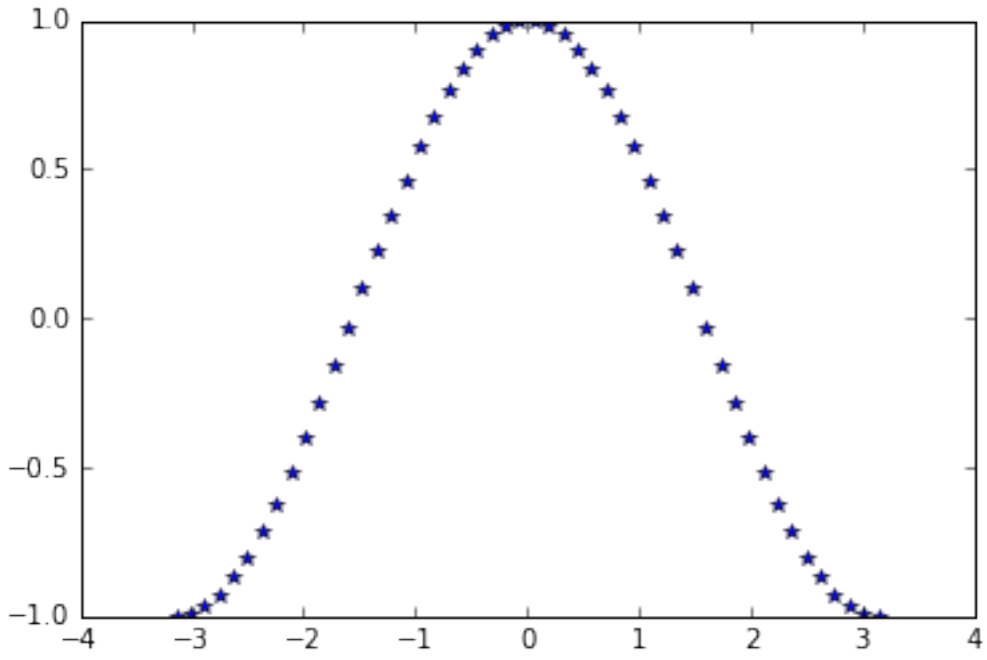


In [11]:

```
plot(x, cos(x), '*')
```

Out[11]:

```
[<matplotlib.lines.Line2D at 0x106cb29e8>]
```



Multiple curves on a single figure

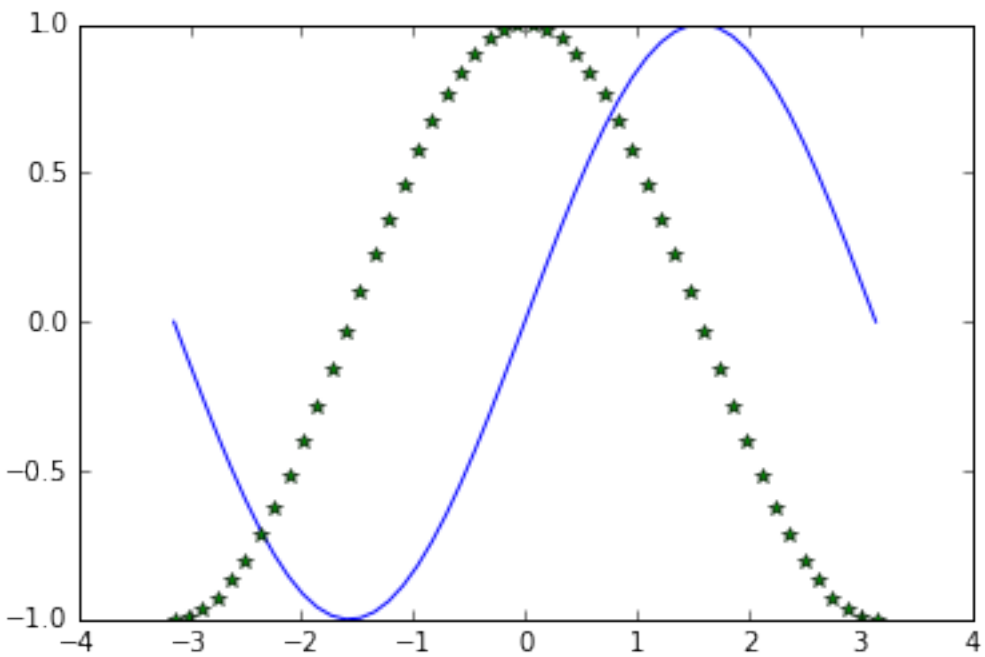
With inline graphs in an IPython notebook, each separate cell produces a new figure. To combine curves on a single graph, several commands can be put in a single cell:

In [12]:

```
plot(x, sin(x))
plot(x, cos(x), '*')
```

Out[12]:

```
[<matplotlib.lines.Line2D at 0x106cc5160>]
```



(Note: this does not work the same way when plotting in an external figure window, as with the above "tk" option.)

Two curves with a single `plot` command

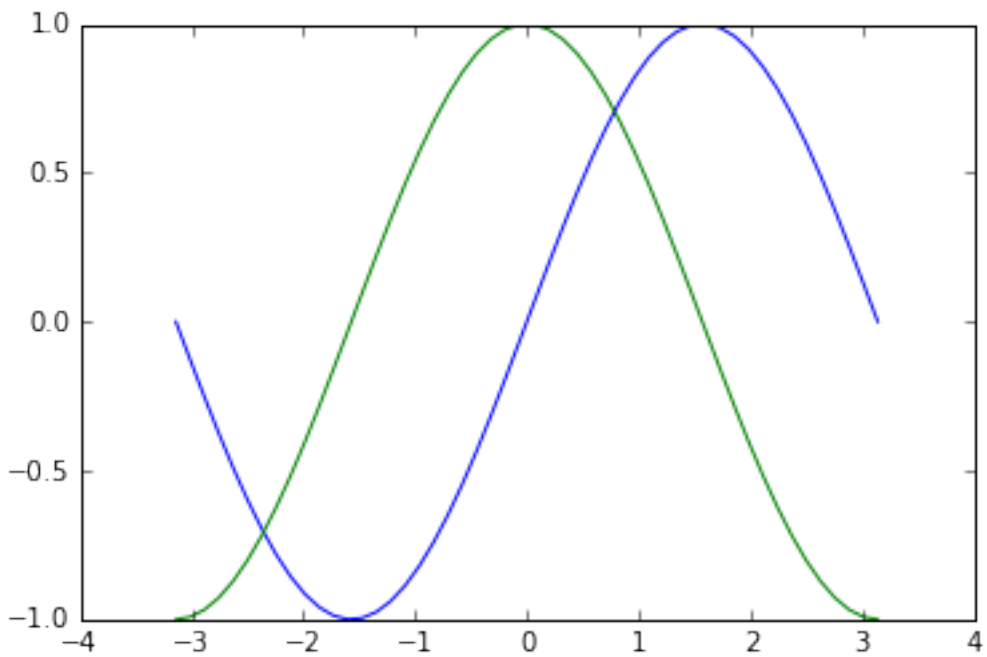
Alternatively, several curves can be specified in a single `plot` command (which also works with separate figure windows.)

In [13]:

```
plot(x, sin(x), x, cos(x))
```

Out[13]:

```
[<matplotlib.lines.Line2D at 0x106dd2f28>,
 <matplotlib.lines.Line2D at 0x106dd82b0>]
```



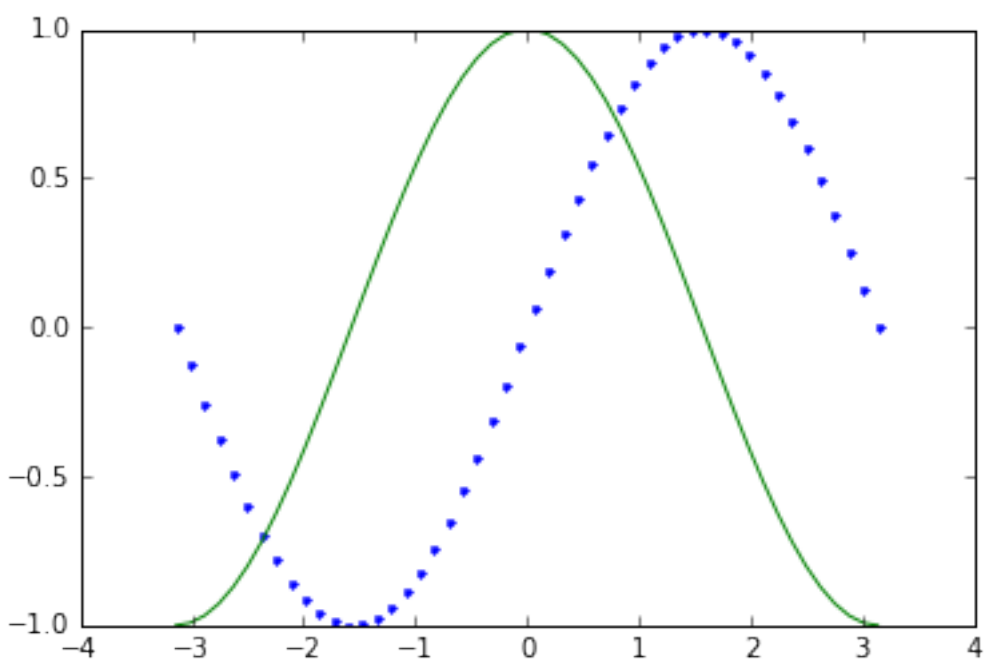
Even with multiple curves in a single plot command, markers can be specified on some, none or all: Matplotlib uses the difference between an array and a text string to recognize which arguments specify markers instead of data.

In [14]:

```
plot(x, sin(x), '.', x, cos(x))
```

Out[14]:

```
[<matplotlib.lines.Line2D at 0x107019e48>,
 <matplotlib.lines.Line2D at 0x1070211d0>]
```



(Note that a dot is another choice of marker.)

Multiple curves with a single `plot` command

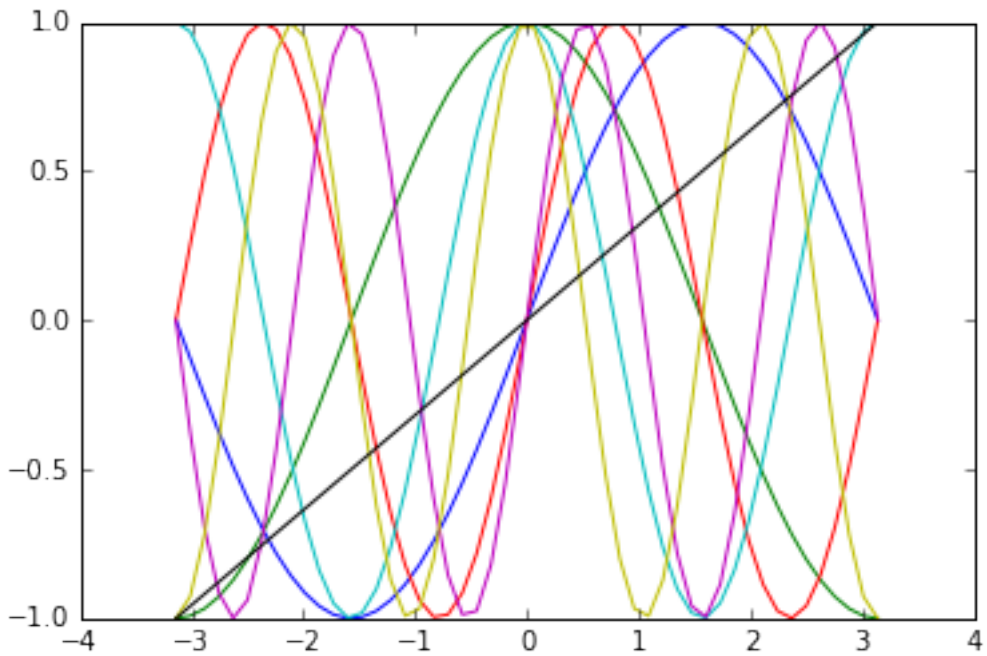
There can be any number of curves in a single `plot` command:

In [15]:

```
plot(x, sin(x), x, cos(x), x, sin(2*x), x, cos(2*x), x, sin(3*x), x, cos(3*x), x
, x/pi)
```

Out[15]:

```
[<matplotlib.lines.Line2D at 0x1070e7ac8>,
 <matplotlib.lines.Line2D at 0x1070e7e10>,
 <matplotlib.lines.Line2D at 0x1070ec668>,
 <matplotlib.lines.Line2D at 0x1070ece10>,
 <matplotlib.lines.Line2D at 0x1070f25f8>,
 <matplotlib.lines.Line2D at 0x1070f2da0>,
 <matplotlib.lines.Line2D at 0x1070f7588>]
```



Note the color sequence: it is blue, green, red, cyan, magenta, yellow, black. After that, it repeats – but you probably don't want more than seven curves on one graph.

Plotting sequences

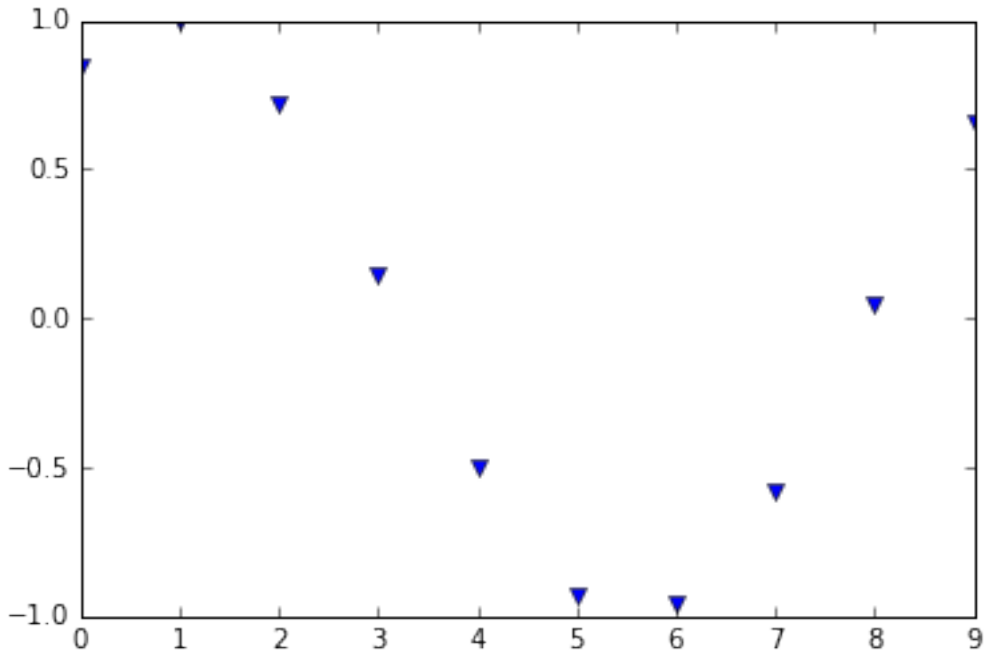
A curve can also be specified by a single array of numbers: these are taken as the values of a sequence, indexed Pythonically from zero, and plotted as the ordinates (vertical values). And we see a new marker option:

In [16]:

```
plot(sin(tenvalues), 'v')
```

Out[16]:

```
[<matplotlib.lines.Line2D at 0x107154e80>]
```



When working with Python files or in the IPython command window (as used within Spyder), one can control whether each new `plot` command produces a new figure or adds to the previous one, with function `hold`.

In [17]:

```
from matplotlib.pyplot import hold
```

Try the following in an IPython command window, not here!

First, get `hold` as above with

```
from matplotlib.pyplot import hold
```

Next, ensure that holding is off:

```
hold(False)
```

and try plotting several curves, like

```
plot(x, sin(x))
plot(x, cos(x))
```

Finally, turn holding on, and repeat:

```
hold(True)
plot(x, sin(2*x))
plot(x, cos(2*x))
```

Decorating the Curves

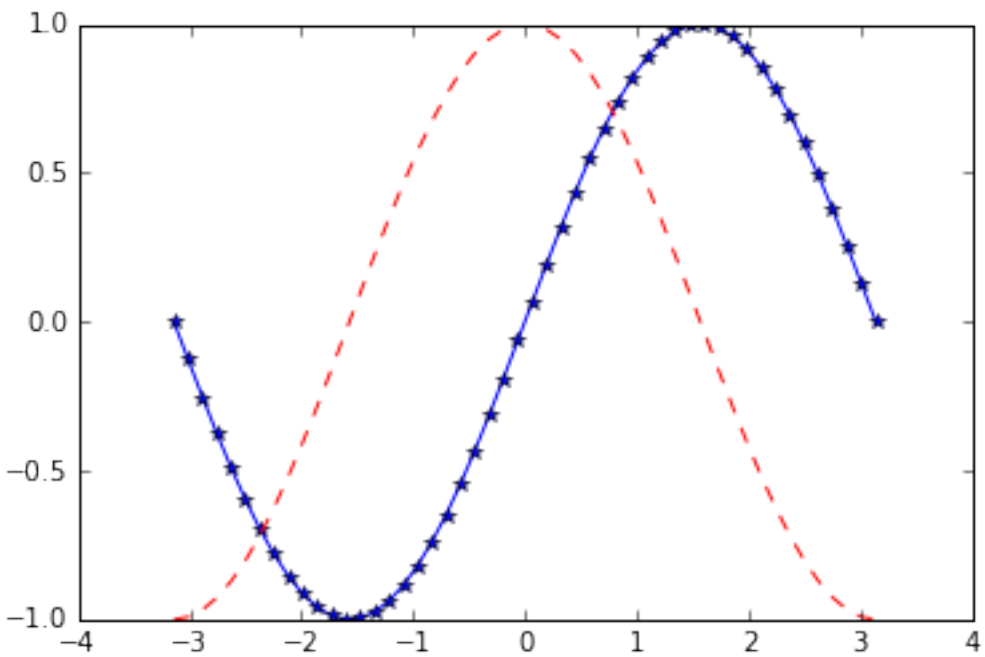
Curves can be decorated in different ways. We have seen how to use markers at the given points instead of joining them with a solid curve and controlling the color. Other options are to specify a color, to specify various line styles like dashed instead of solid, and to have both marker and lines. As seen above, this can be controlled by an optional text string argument after the arrays of data for a curve:

In [18]:

```
plot(x, sin(x), '*-')
plot(x, cos(x), 'r--')
```

Out[18]:

```
[<matplotlib.lines.Line2D at 0x10712f160>]
```



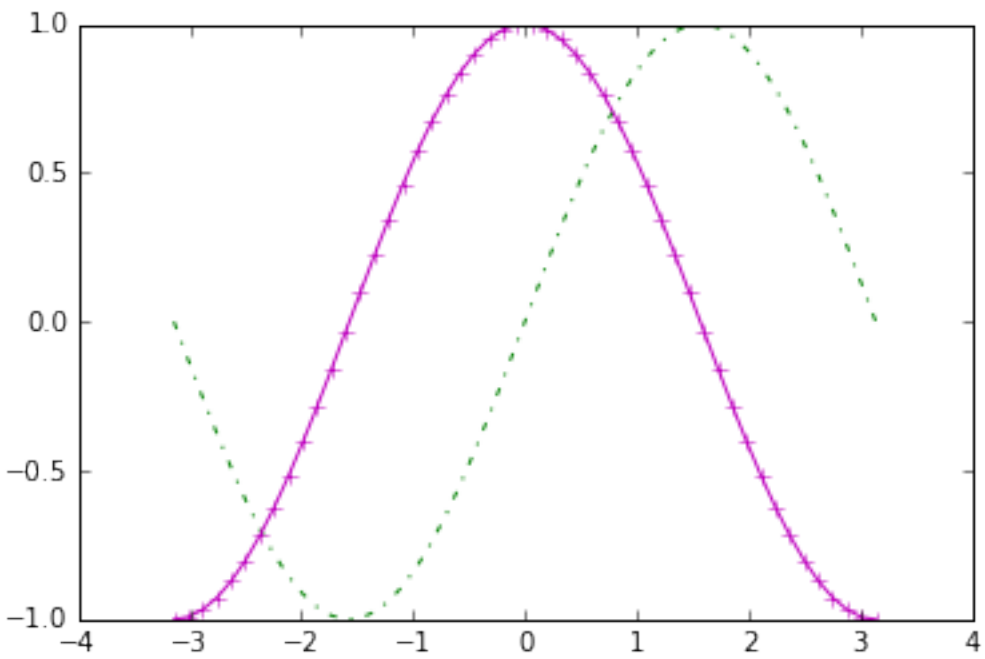
These three part curve specifications can be combined: in the following, `plot` knows that there are two curves each specified by three arguments, not three curves each specified by just an "x-y" pair:

In [19]:

```
plot(x, sin(x), 'g-.', x, cos(x), 'm+-')
```

Out[19]:

```
[<matplotlib.lines.Line2D at 0x1072d7cc0>,
 <matplotlib.lines.Line2D at 0x10733d048>]
```



Exercise: Explore ways to refine your figures

There are many commands for refining the appearance of a figure after its initial creation with `plot`. Experiment yourself with the commands `title`, `xlabel`, `ylabel`, `grid`, and `legend`.

Using the functions mentioned above, produce a refined version of the above sine and cosine graph, with:

- a title at the top
- labels on both axes
- a legend identifying the two curves
- a "graph paper" background, to make it easier to judge details like where the functions have zero values.

Then work out how to save this to a file (probably in format PNG), and turn that in through the Dropbox in OAKS.

Explore other features, like zooming and panning: remember that this must be done with the graphs appearing in a separste window, not inline.

Getting help from the documentation

For some of these, you will probably need to read up. For simple things, there is a function `help`, which is best used in the IPython interactive input window (within Spyder for example), but I will illustrate it here.

In [20]:

```
help(hold)
```

```
Help on function hold in module matplotlib.pyplot:

hold(b=None)
    Set the hold state.  If *b* is None (default), toggle the
    hold state, else set the hold state to boolean value *b*::

      hold()      # toggle hold
      hold(True)  # hold is on
      hold(False) # hold is off

    When *hold* is *True*, subsequent plot commands will be added to
    the current axes.  When *hold* is *False*, the current axes and
    figure will be cleared on the next plot command.
```

The jargon used in `help` can be confusing at first, but there are other online sources that are more readable and better illustrated, like http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html (http://scipy-lectures.github.io/intro/matplotlib/matplotlib.html) mentioned above.

However, that does not cover everything; the official pyplot documentation at http://matplotlib.org/1.3.1/api/pyplot_api.html (http://matplotlib.org/1.3.1/api/pyplot_api.html) is more complete: explore its search feature.

P. S. A shortcut revealed: pylab

So far I have encourage you to use explcit specific import commands, because this is good practice when developing larger programs. However, for quck interactive work in the IPython command window and IPython notebooks, there is a useful shortcut: the IPython magic command

```
%pylab
```

adds everything from Numpy and the main parts of Matplotlib, including all the items imported above. (This creates the so-called **pylab** environment: that name combines "Python" with "Matlab", as its goal is to produce an environment very similar to Matlab.)

Note that this is a command for the IPython interactive system command, not a Python language command, so it must be used either in a IPython notebook or in the IPython command window (within Spyder), not in a Python ".py" file.