

# Plotting in Python with matplotlib

---

Written by [Dave Pawlowski](#),

## Introduction

There are a variety of python modules that can be used to create figures and plots in python. One of the most widely used in the scientific community is matplotlib. matplotlib is a MATLAB-like in the way that it handles plotting. Typically, plot commands and figure configuration options (i.e. sizing, positioning, labels) occur on separate lines as opposed to on in a single function call. There is a lot of good documentation online for matplotlib. A good place to start, beyond what is discussed here, is [matplotlib.org](http://matplotlib.org).

I should mention that there are many many things that one can do with the tools in matplotlib. We are only going to be touching on the most basic of things here; enough to be able to create usable graphics for scientific analysis. That said, even the limited subset of tools discussed will be enough to produce *publication quality* graphics.

## 1 pyplot

We'll be using the functions that are part of the pyplot submodule with in matplotlib:

```
from matplotlib import pyplot
```

pyplot is a collection of functions designed to allow the user to do MATLAB style plotting. Each command is made to allow you to modify a figure. pyplot is considered to be *stateful*. This means that it keeps track of the current figure and plotting area that you are working on and that any functions that you call are directed to the current axes.

Once pyplot is imported, we play around a bit.

```
import numpy as np
x = np.linspace(0,99,100)
y = np.sin(x*np.pi/20.)
pyplot.plot(x,y)
```

Here we have created a line plot of a sine curve. However, python doesn't know what to do with the plot. It hasn't been told to actually show it to us yet. At this point we have two options:

1. Let python open up its own graphics window to display the plot, or
2. Save the plot to a file

If you are working on a remote computer, then I recommend option 2, as option 1 will require you to have x-windows forward properly setup and configured, and eventually, you probably will want to save the file anyway. For simplicity though, let's cover option 1 first.

### Option 1

To tell python to actually show us the plot, type

```
plt.show()
```

The show command does what it says, shows the plot in python's on graphics window. The plot should pop up, and you should be able to interact with the window using the mouse to do things like save the plot and such.

### Option 2

In general, unless trying to get a really quick view of a plot that you just made, I recommend just saving the thing to a file so that you have a copy that you can look back on later, or send to yourself or others. Additionally, if you are working on a remote computer, you can use some sort of secure copy program (i.e. scp, rsync, winscp, etc) to pull the file to a local machine. Saving a figure is pretty easy:

```
plt.savefig('myplot.eps')
```

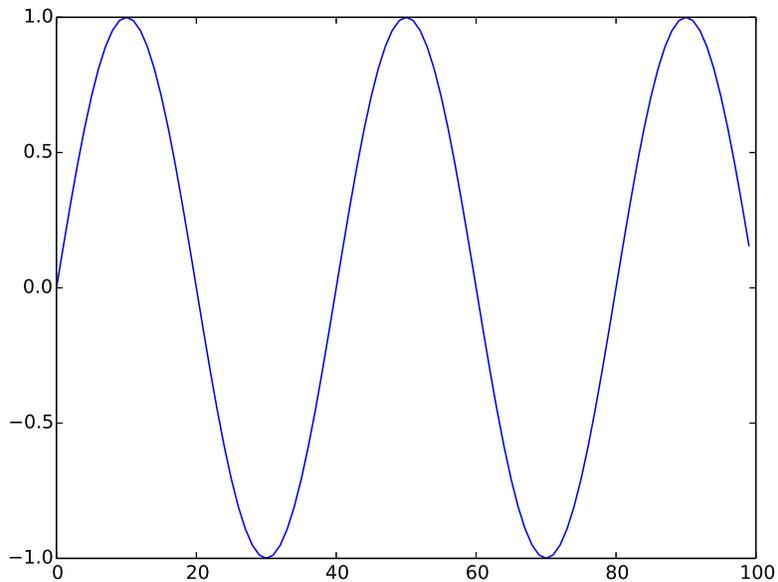
Entering this line should save your plot to a file called *myplot.eps*. `savefig` looks at the extension of the filename that you've entered and automatically formats the file accordingly (assuming the extension refers to a real image type). I am partial to postscript figures. The `.eps` extension will create an encapsulated postscript. If you create this type of file, you will need a program to view it. On Linux machines, there should be a utility called `ghostview` that allows you to do so that is already installed. The command that you would use to view the file from the Linux shell is:

```
% gv myplot.eps
```

If you are on a windows computer, you can download GSview for free online for all your postscript viewing needs. When using the `savefig` command, it is generally safe to use `.eps`, `.pdf`, and `.png` extensions. These should always work properly.

## 1.1 Plot Options

Once you view the plot in the example above, you should see a simple sine curve. However, our plot is missing several features, namely axis labels and a title. Adding these things is very straightforward. Remember that `matplotlib` is stateful! This means that options should be added before calling the `show` or `savefig` functions. Otherwise you won't see the changes.



```

from matplotlib import pyplot as p
import numpy as np
p.xlabel('Time (s)')
p.ylabel('Amplitude (m)')
p.title('Harmonic Oscillator')
p.plot(x,y)
p.savefig('myplot.eps')

```

The plot function can be used to plot multiple lines on a single plot.

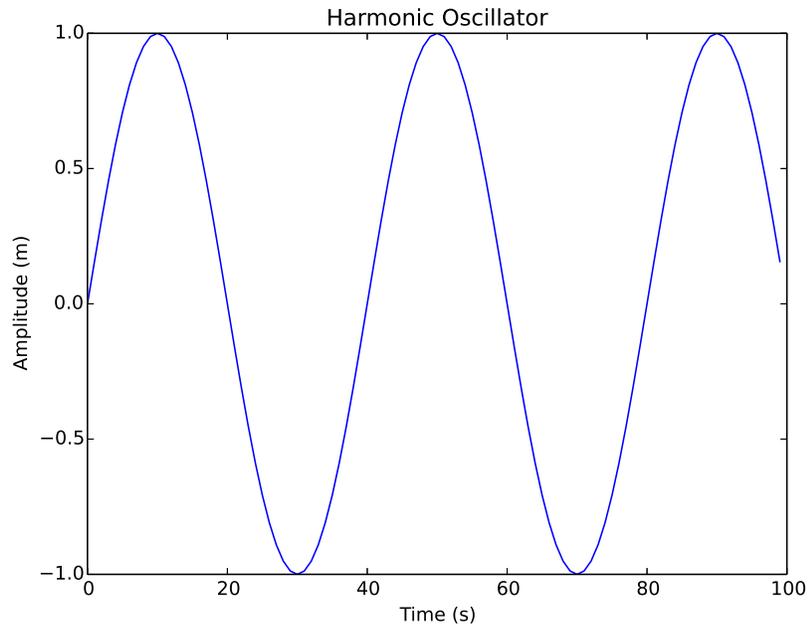
```

from matplotlib import pyplot as p
import numpy as np
x = np.linspace(0,99,100)

y1 = np.sin(x*np.pi/20.)
y2 = 0.5*np.cos(x*np.pi/10.)
y3 = 0.4*np.cos(x*np.pi/5.)
p.xlabel('Time (s)')
p.ylabel('Amplitude (m)')
p.title('Harmonic Oscillator')
p.plot(x,y1,x,y2,x,y3)
p.savefig('myplot.eps')

```

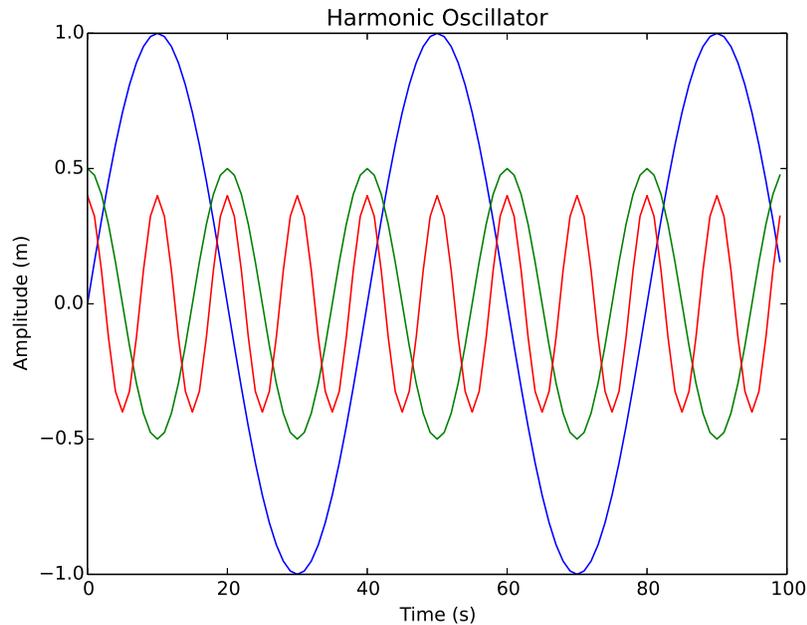
Notice that pyplot handles the colors for you, but you are free to change those, as well as use plot symbols.



```
p.plot(x,y1,'cs',x,y2,'r^',x,y3,'mo-')  
#cyan squares, red triangles, magenta circles with a line.  
p.savefig('myplot.eps')
```

For more information on colors and symbols, see [http://matplotlib.org/api/colors\\_api.html](http://matplotlib.org/api/colors_api.html) and [http://matplotlib.org/api/markers\\_api.html#module-matplotlib.markers](http://matplotlib.org/api/markers_api.html#module-matplotlib.markers).

It is also quite easy to make a figure that has multiple plots. To do this, we have to let pyplot know in advance what we are going to be doing. Remember, commands usually apply to the current axes. If we have multiple axes, then we need to tell pyplot which ones to deal with.



```

from matplotlib import pyplot as p
import numpy as np
x = np.linspace(0,99,100)

y1 = np.sin(x*np.pi/20.)
y2 = 0.5*np.cos(x*np.pi/10.)
y3 = 0.4*np.cos(x*np.pi/5.)

myfig = p.figure(1)
#this is actually optional since the figure is created by default.

p.subplot(311)
#We will have 3 plots in a vertical column. To do the same thing
#horizontally, we would use (131). First up, the first subplot.

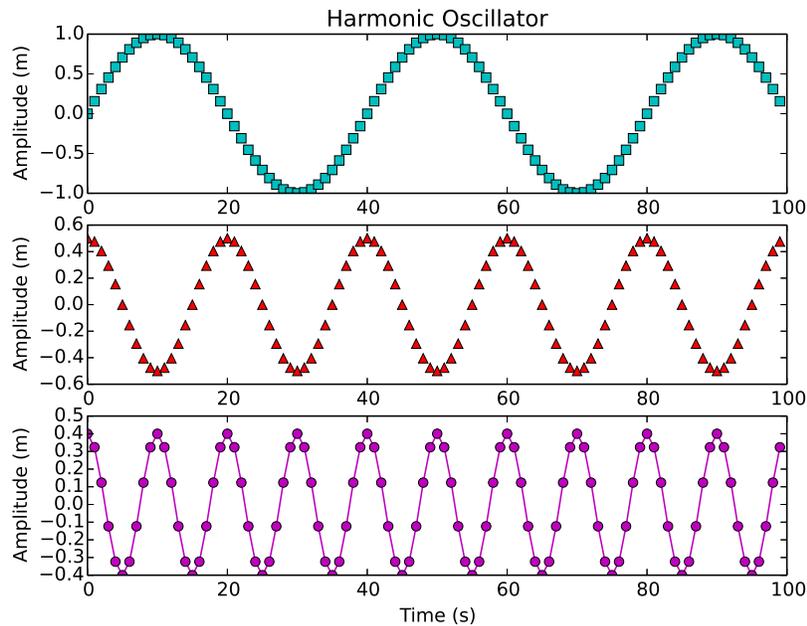
p.plot(x,y1,'cs')

#Note the labels. I don't need x-labels for the first two plots since they
#are all the same.
#Also, I only need to title the plot once.
p.ylabel('Amplitude (m)')
p.title('Harmonic Oscillator')

p.subplot(312) #Next, the second subplot in the first figure.
p.plot(x,y2,'r^')
p.ylabel('Amplitude (m)')

p.subplot(313) #Finally, the third.
p.plot(x,y3,'mo-')
p.xlabel('Time (s)')
p.ylabel('Amplitude (m)')
p.savefig('myplot.eps')

```



Calling `p.figure()` can be used if my program makes multiple figures. This enables the user to switch back and forth between them by calling the function with the appropriate figure number. In this case, we only had a single figure so it was unnecessary.

Finally, since we've made a reasonably complex figure, we may want to have a label.

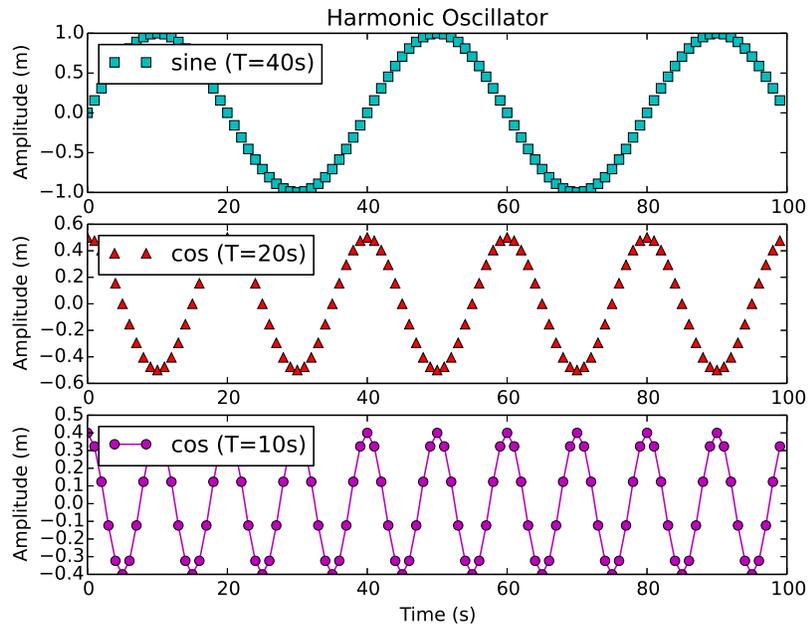
```
p.subplot(311)
p.plot(x,y1,'cs',label='sine (T=40s)')
p.legend(loc='upper left')

p.ylabel('Amplitude (m)')
p.title('Harmonic Oscillator')

p.subplot(312) #Next, the second subplot in the first figure.
p.plot(x,y2,'r^',label='cos (T=20s)')
p.ylabel('Amplitude (m)')
p.legend(loc='upper left')

p.subplot(313) #Finally, the third.
p.plot(x,y3,'mo-',label='cos (T=10s)')
p.xlabel('Time (s)')
p.ylabel('Amplitude (m)')
p.legend(loc='upper left')

p.savefig('myplot.eps')
```



In this example, we labeled each plot separately, so we had to call `p.legend()` 3 times. There are ways to make this a bit more professional, such as simply annotating each line, but they do require a bit more complexity. For now, this will suffice.

A huge bonus of pyplot is that it is capable of handling  $\LaTeX$ style typesetting, including math-mode, in its strings.