Topic for the class:– EDA Graphical Technique
Unit _2 : Title-Digital data – an Imprint
Date & Time : 5.8.24 11.00 AM – 11.50 AM

Dr. Bhramaramba Ravi
Professor
Department of Computer Science and Engineering
GITAM School of Technology (GST)
Visakhapatnam – 530045
Email: **bravi@gitam.edu**

Unit2-syllabus

- **UNIT 2 Digital Data-An Imprint 9 hours, P - 2 hours** Type of data analytics (Descriptive, diagnostic, perspective, predictive, Prescriptive.) Exploratory Data Analysis (EDA), EDA-Quantitative Technique, EDA - Graphical Technique. Data Types for Plotting, Data Types and Plotting, Simple Line Plots, Simple Scatter Plots, Visualizing Errors, Density and Contour Plots, Histograms, Binnings, and Density, Customizing Plot Legends, Customizing Color bars, Multiple Subplots, Text and Annotation, Customizing Ticks.
- https://www.coursera.org/learn/data-visualization-r

Matplotlib

- Matplotlib is a multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack. It was conceived by John Hunter in 2002, originally as a patch to IPython for enabling interactive MATLAB-style plotting via gnuplot from the IPython command line.
- One of Matplotlib's most important features is its ability to play well with many operating systems and graphics backends. Matplotlib supports dozens of backends and output types, which means you can count on it to work regardless of which operating system you are using or which output format you wish. This cross-platform, everything-to-everyone approach has been one of the great strengths of Matplotlib. It has led to a large userbase, which in turn has led to an active developer base and Matplotlib's powerful tools and ubiquity within the scientific Python world.

Matplotlib

- In recent years, however, the interface and style of Matplotlib have begun to show their age. Newer tools like ggplot and ggvis in the R language, along with web visualization toolkits based on D3js and HTML5 canvas, often make Matplotlib feel clunky and old-fashioned.

General matplotlib tips

- Importing matplotlib
- Just as we use the np shorthand for NumPy and the pd shorthand for Pandas, we will use some standard shorthands for Matplotlib imports:
- In[1]: **import matplotlib as mpl**
- **import matplotlib.pyplot as plt**
- The plt interface is what we will use most often
- **Setting Styles**
- We will use the plt.style directive to choose appropriate aesthetic styles for our figures.
- Here we will set the classic style, which ensures that the plots we create use the
- classic Matplotlib style:
- In[2]: plt.style.use('classic')

Plotting from a script

- If you are using Matplotlib from within a script, the function plt.show() is your friend. plt.show() starts an event loop, looks for all currently active figure objects,
- and opens one or more interactive windows that display your figure or figures.
- So, for example, you may have a file called *myplot.py* containing the following:

```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

Plotting from a script

- You can then run this script from the command-line prompt, which will result in a window opening with your figure displayed:
- $ python myplot.py
- The plt.show() command does a lot under the hood, as it must interact with your system's interactive graphical backend. The details of this operation can vary greatly from system to system and even installation to installation, but Matplotlib does its best to hide all these details from you.
- One thing to be aware of the plt.show() command should be used only once per python session and Is most often seen at the very end of the script.
- Multiple show commands can lead to unpredictable backend-dependent behavior and should mostly be avoided.

Plotting from an IPython shell

- It can be very convenient to use Matplotlib interactively within an IPython shell
- . IPython is built to work well with Matplotlib if you specify Matplotlib mode. To enable this mode, you can use the %matplotlib magic command after starting ipython:
- In [1]: %**matplotlib**
- Using matplotlib backend: TkAgg
- In [2]: **import matplotlib.pyplot as plt**
- At this point, any plt plot command will cause a figure window to open, and further commands can be run to update the plot. Some changes (such as modifying properties of lines that are already drawn) will not draw automatically; to force an update,use plt.draw(). Using plt.show() in Matplotlib mode is not required.

Plotting from an Ipython notebook

- The IPython notebook is a browser-based interactive data analysis tool that can combine

  narrative, code, graphics, HTML elements, and much more into a single executable

  document .
- Plotting interactively within an IPython notebook can be done with the %matplotlib command, and works in a similar way to the IPython shell. In the IPython notebook,
- you also have the option of embedding graphics directly in the notebook, with two possible options:
- • %matplotlib notebook will lead to *interactive* plots embedded within the notebook
- • %matplotlib inline will lead to *static* images of your plot embedded in the notebook
- Here, we will generally opt for %matplotlib inline:
- In[3]: %matplotlib inline

Plotting from an IPython notebook contd.

- After you run this command (it needs to be done only once per kernel/session), any cell within the notebook that creates a plot will embed a PNG image of the resulting graphic (Figure 4-1):
- In[4]: **import numpy as np**
- x = np.linspace(0, 10, 100)
- fig = plt.figure()
- plt.plot(x, np.sin(x), '-')
- plt.plot(x, np.cos(x), '--');

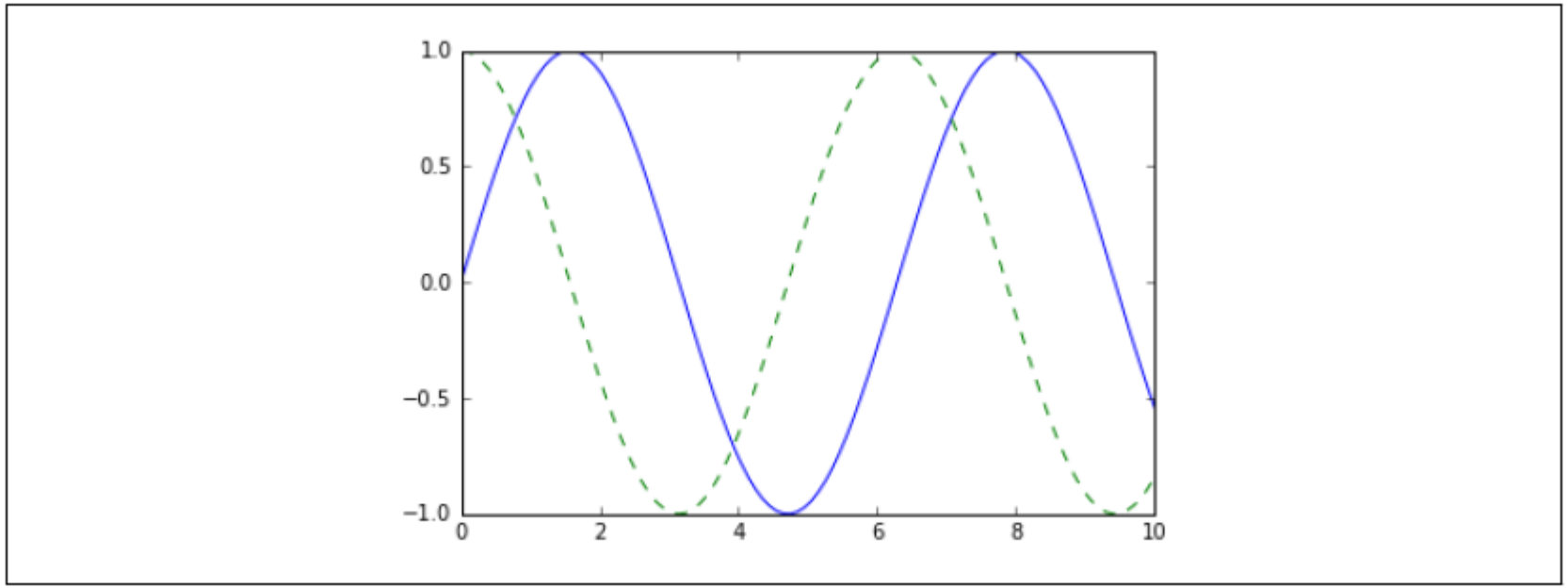

Figure 4-1. Basic plotting example

Saving figures to file

- One nice feature of Matplotlib is the ability to save figures in a wide variety of formats.
- You can save a figure using the savefig() command. For example, to save the previous figure as a PNG file, you can run this:
- In[5]: fig.savefig('my_figure.png')
- We now have a file called *my_figure.png* in the current working directory:
- In[6]: !ls -lh my_figure.png
- -rw-r--r-- 1 jakevdp staff 16K Aug 11 10:59 my_figure.png
- To confirm that it contains what we think it contains, let's use the IPython Image object to display the contents of this file (Figure 4-2):
- In[7]: **from IPython.display import** Image
- Image('my_figure.png')

Saving figures to file

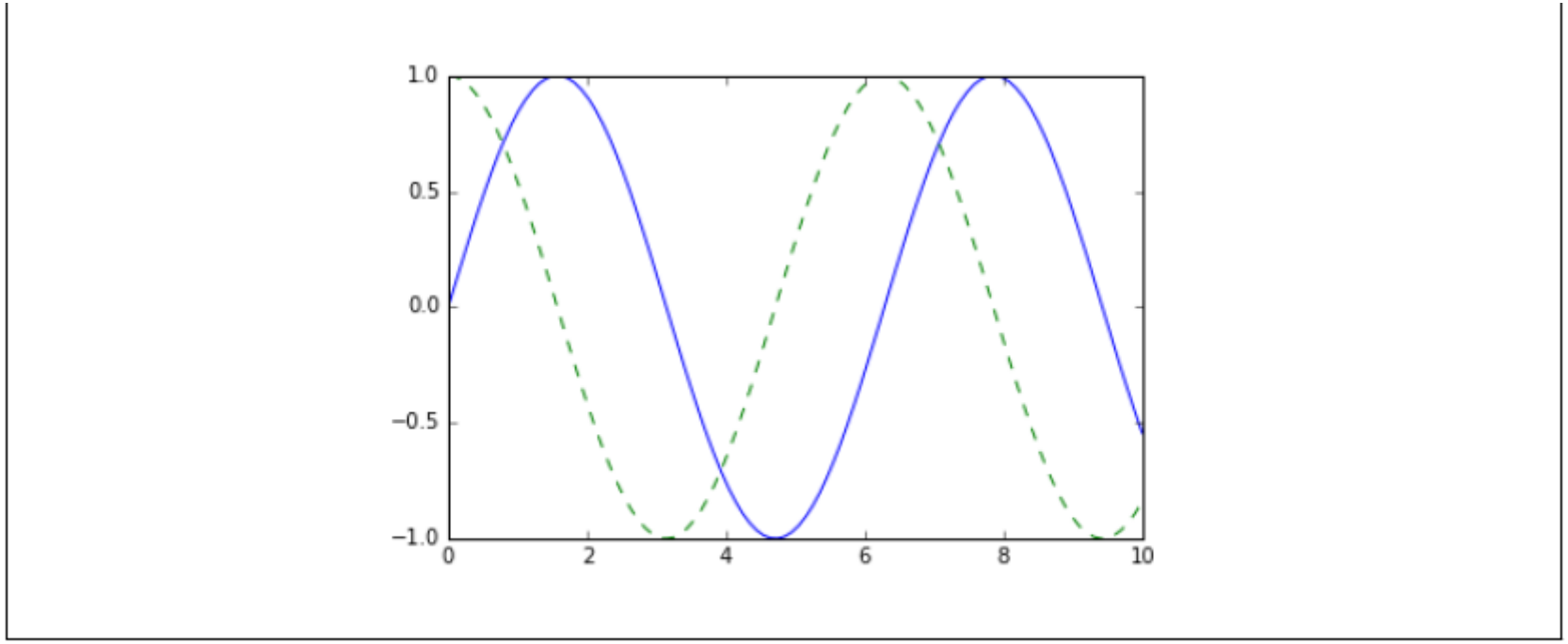

Figure 4-2. PNG rendering of the basic plot

Saving figures to file

- In savefig(), the file format is inferred from the extension of the given filename.
- Depending on what backends you have installed, many different file formats are available. You can find the list of supported file types for your system by using the following method of the figure canvas object:
- In[8]: fig.canvas.get_supported_filetypes()
- Out[8]: {'eps': 'Encapsulated Postscript',
- 'jpeg': 'Joint Photographic Experts Group',
- 'jpg': 'Joint Photographic Experts Group',
- 'pdf': 'Portable Document Format',
- 'pgf': 'PGF code for LaTeX',
- 'png': 'Portable Network Graphics',
- 'ps': 'Postscript',
- 'raw': 'Raw RGBA bitmap',
- 'rgba': 'Raw RGBA bitmap’,
- 'svg': 'Scalable Vector Graphics',
- 'svgz': 'Scalable Vector Graphics',
- 'tif': 'Tagged Image File Format',
- 'tiff': 'Tagged Image File Format'}
- Note that when saving your figure, it’s not necessary to use plt.show() or related commands discussed earlier.

Two Interfaces for the Price of One

- A potentially confusing feature of Matplotlib is its dual interfaces: a convenient MATLAB-style state-based interface, and a more powerful object-oriented interface.
- We'll quickly highlight the differences between the two here.
- MATLAB-style interface
- Matplotlib was originally written as a Python alternative for MATLAB users, and much of its syntax reflects that fact. The MATLAB-style tools are contained in the pyplot (plt) interface. For example, the following code will probably look quite familiar to MATLAB users (Figure 4-3):
- In[9]: plt.figure() *# create a plot figure*
- *# create the first of two panels and set current axis*
- plt.subplot(2, 1, 1) *# (rows, columns, panel number)*
- plt.plot(x, np.sin(x))
- *# create the second panel and set current axis*
- plt.subplot(2, 1, 2)
- plt.plot(x, np.cos(x));

Subplots using the MATLAB style interface

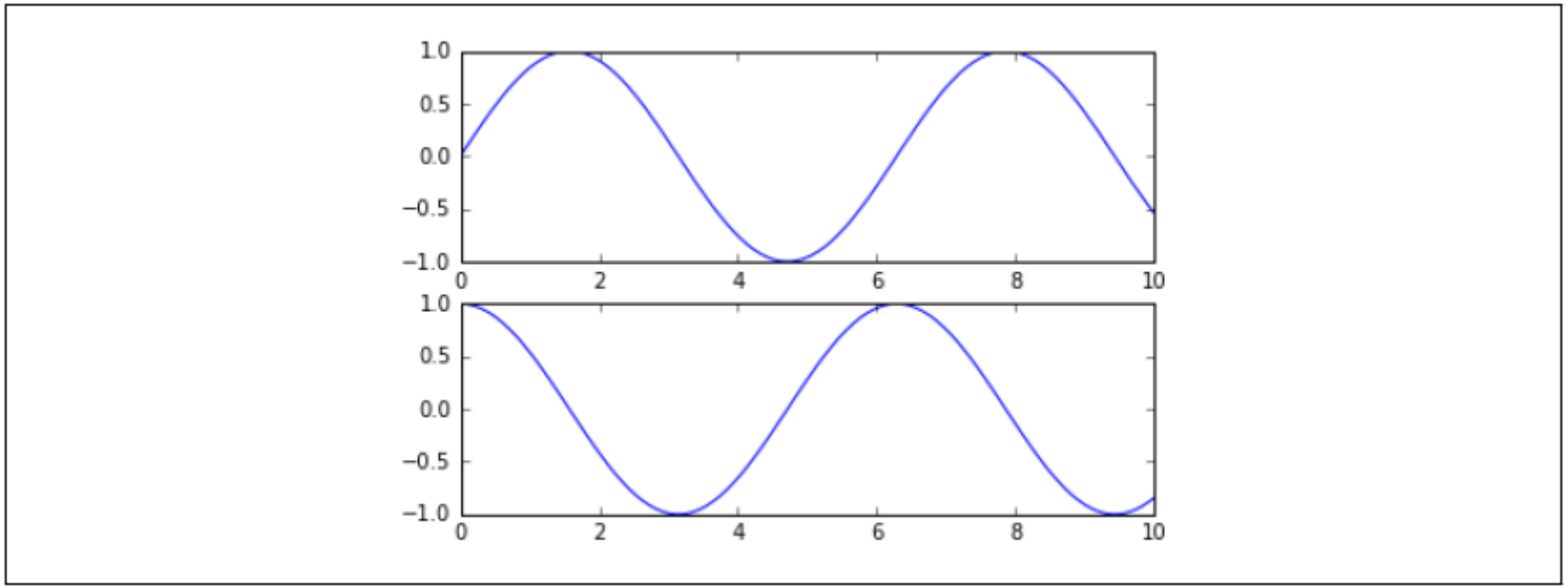

Figure 4-3. Subplots using the MATLAB-style interface

Subplots using the MATLAB style interface

- It's important to note that this interface is *stateful*: it keeps track of the "current" figure and axes, which are where all plt commands are applied. You can get a reference to these using the plt.gcf() (get current figure) and plt.gca() (get current axes)routines.
- While this stateful interface is fast and convenient for simple plots, it is easy to run into problems. For example, once the second panel is created, how can we go back and add something to the first? This is possible within the MATLAB-style interface, but a bit clunky. Fortunately, there is a better way.

Object-oriented interface

- The object-oriented interface is available for these more complicated situations, and for when you want more control over your figure. Rather than depending on some notion of an “active” figure or axes, in the object-oriented interface the plotting functions are *methods* of explicit Figure and Axes objects. To re-create the previous plot using this style of plotting, you might do the following (Figure 4-4):
- In[10]: *# First create a grid of plots*
- *# ax will be an array of two Axes objects*
- fig, ax = plt.subplots(2)
- *# Call plot() method on the appropriate object*
- ax[0].plot(x, np.sin(x))
- ax[1].plot(x, np.cos(x));
- For more simple plots, the choice of which style to use is largely a matter of preference, but the object-oriented approach can become a necessity as plots become more complicated.

Object-oriented interface

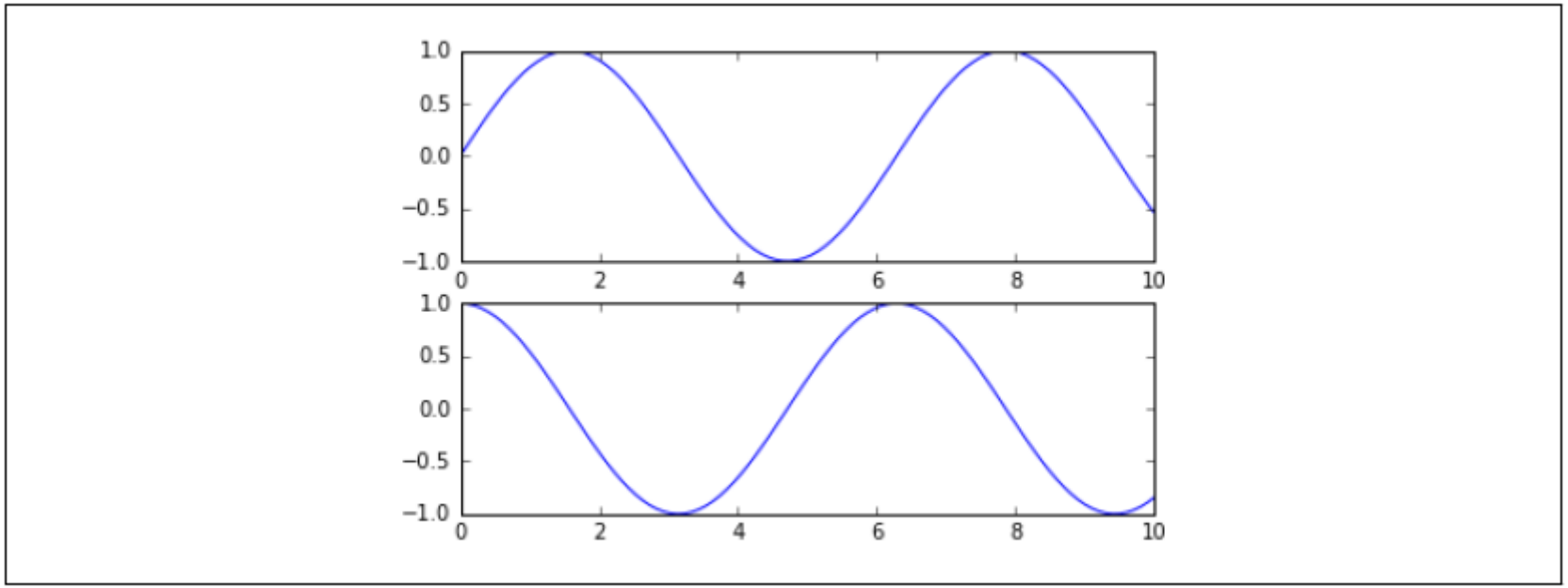

Figure 4-4. Subplots using the object-oriented interface

Simple line plots

- Perhaps the simplest of all plots is the visualization of a single function $y = f\,x$. Here we will take a first look at creating a simple plot of this type.
- As with all the following sections, we'll start by setting up the notebook for plotting and importing the functions
- we will use:
- In[1]: %matplotlib inline
- **import matplotlib.pyplot as plt**
- plt.style.use('seaborn-whitegrid')
- **import numpy as np**
- For all Matplotlib plots, we start by creating a figure and an axes. In their simplest form, a figure and axes can be created as follows (Figure 4-5):
- In[2]: fig = plt.figure()
- ax = plt.axes()

Fig 4.5 an empty gridded axes

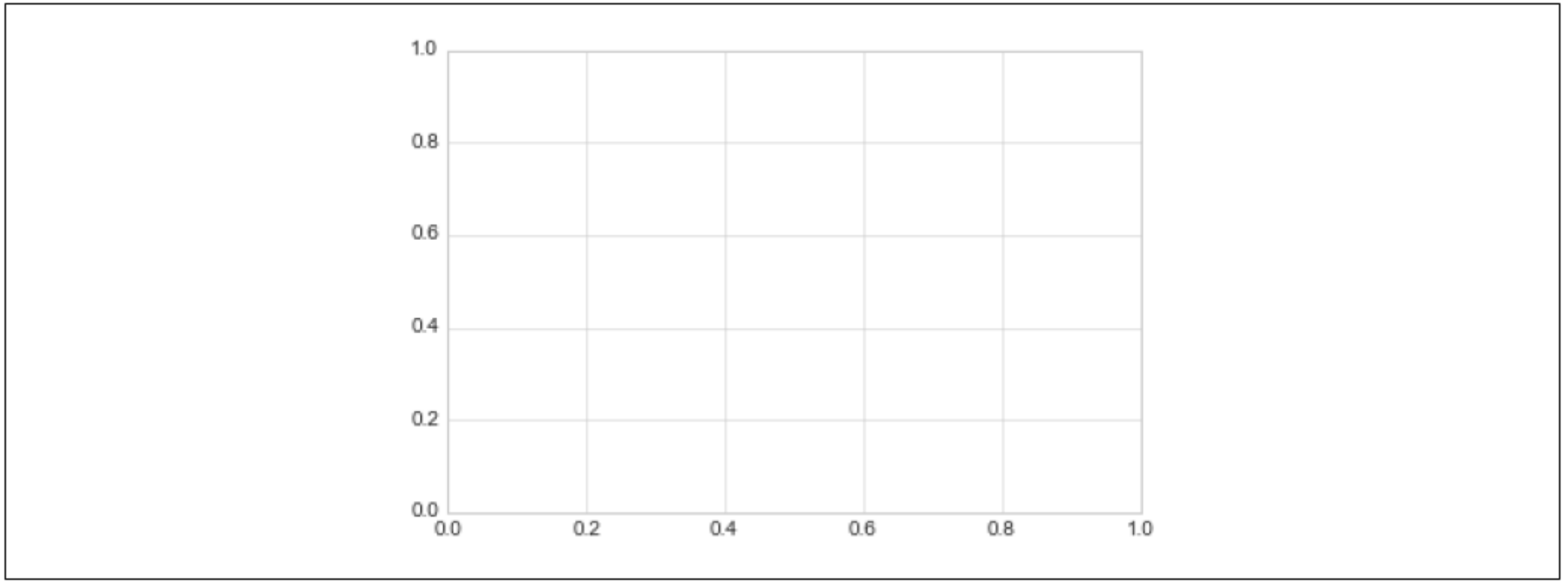

Figure 4-5. An empty gridded axes

Simple line plots

- In Matplotlib, the *figure* (an instance of the class plt.Figure) can be thought of as a single container that contains all the objects representing axes, graphics, text, and labels.
- The *axes* (an instance of the class plt.Axes) is what we see above: a bounding box with ticks and labels, which will eventually contain the plot elements that make up our visualization.
- fig to refer to a figure instance, and ax to refer to an axes instance or group of axes
- instances.
- Once we have created an axes, we can use the ax.plot function to plot some data.
- Let's start with a simple sinusoid (Figure 4-6):
- In[3]: fig = plt.figure()
- ax = plt.axes()
- x = np.linspace(0, 10, 1000)
- ax.plot(x, np.sin(x));

Single line plots

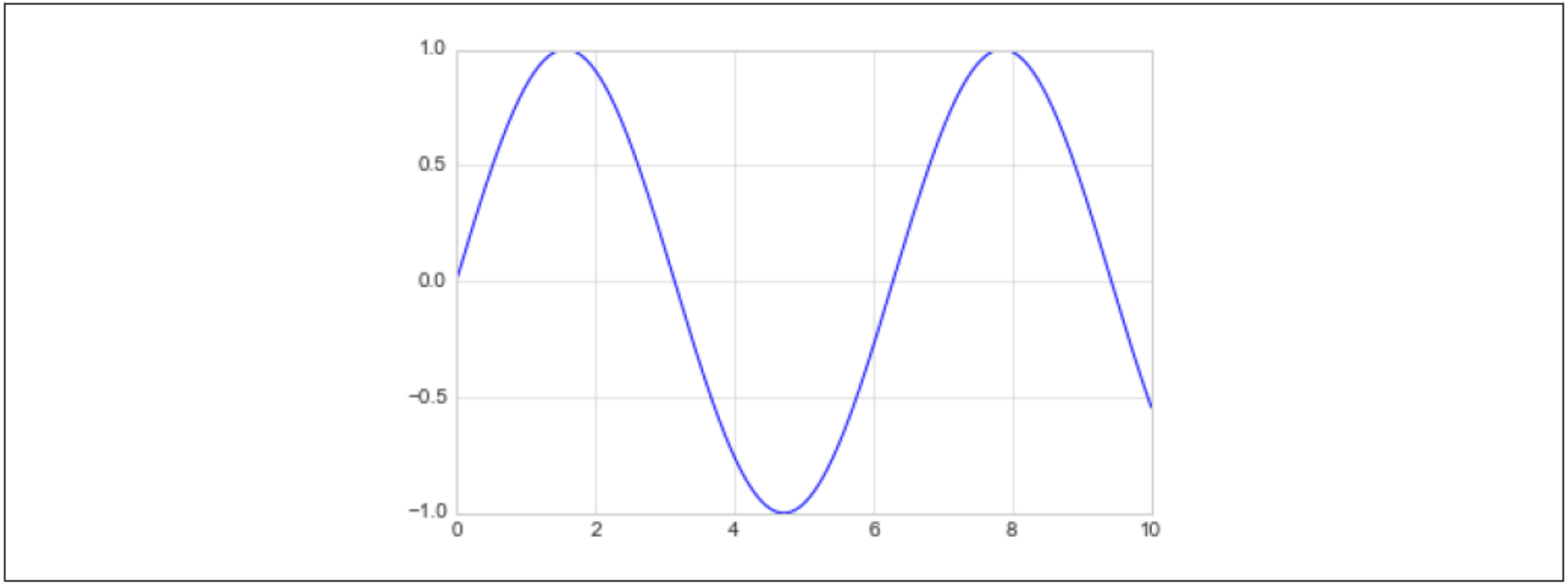

Figure 4-6. A simple sinusoid

Simple line plots

- Alternatively, we can use the pylab interface and let the figure and axes be created for us in the background (Figure 4-7]
- In[4]: plt.plot(x, np.sin(x));

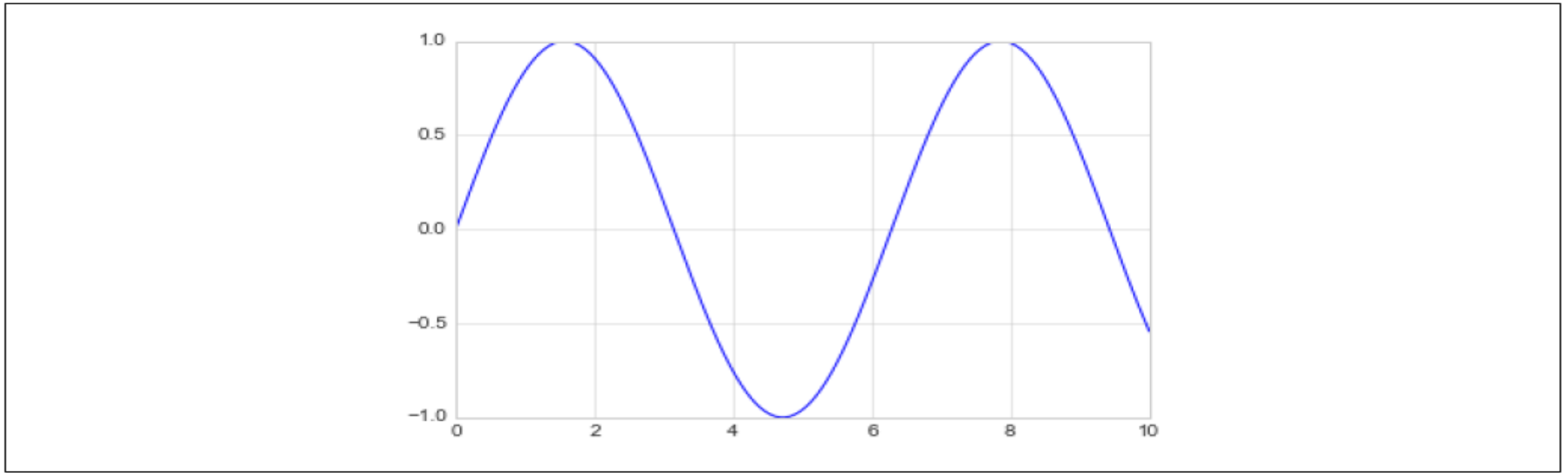

Figure 4-7. A simple sinusoid via the object-oriented interface

Single line Plots

- If we want to create a single figure with multiple lines, we can simply call the plot function multiple times (Figure 4-8):
- In[5]: plt.plot(x, np.sin(x))
- plt.plot(x, np.cos(x));

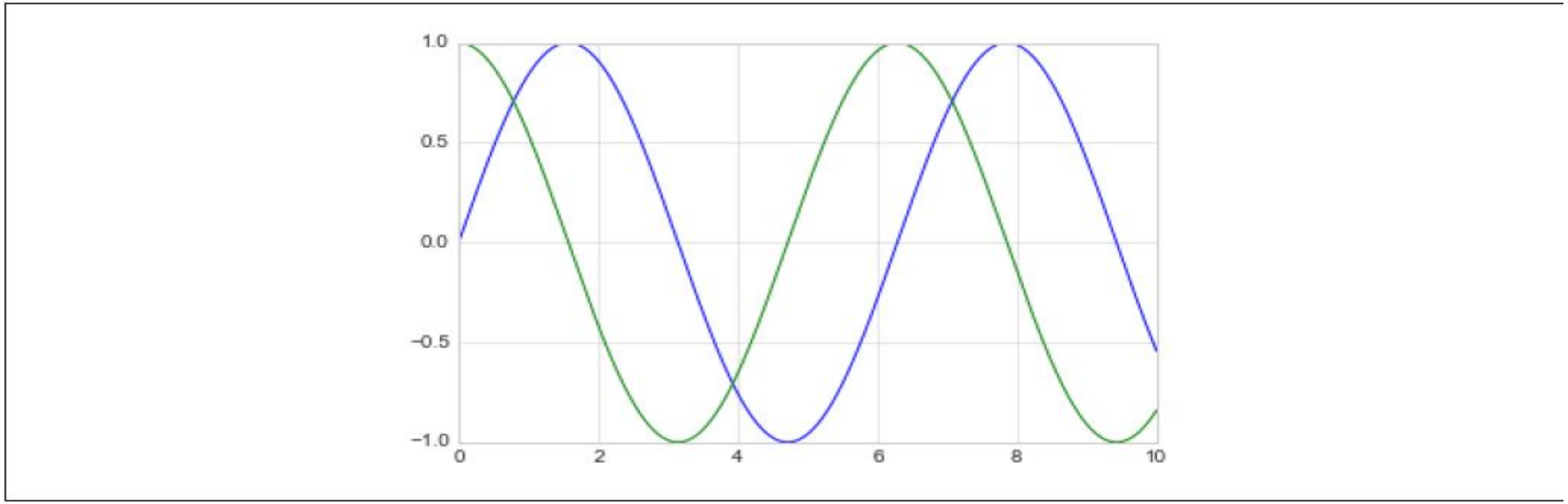

Figure 4-8. Over-plotting multiple lines

Adjusting the plot: line colors and styles

- The first adjustment you might wish to make to a plot is to control the line colors and
- styles. The plt.plot() function takes additional arguments that can be used to specify
- these. To adjust the color, you can use the color keyword, which accepts a string
- argument representing virtually any imaginable color. The color can be specified in a
- variety of ways (Figure 4-9):
- In[6]:
- plt.plot(x, np.sin(x - 0), color='blue') *# specify color by name*
- plt.plot(x, np.sin(x - 1), color='g') *# short color code (rgbcmyk)*
- plt.plot(x, np.sin(x - 2), color='0.75') *# Grayscale between 0 and 1*
- plt.plot(x, np.sin(x - 3), color='#FFDD44') *# Hex code (RRGGBB from 00 to FF)*
- plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) *# RGB tuple, values 0 and 1*
- plt.plot(x, np.sin(x - 5), color='chartreuse'); *# all HTML color names supported*

Adjusting the plot: line colors and styles

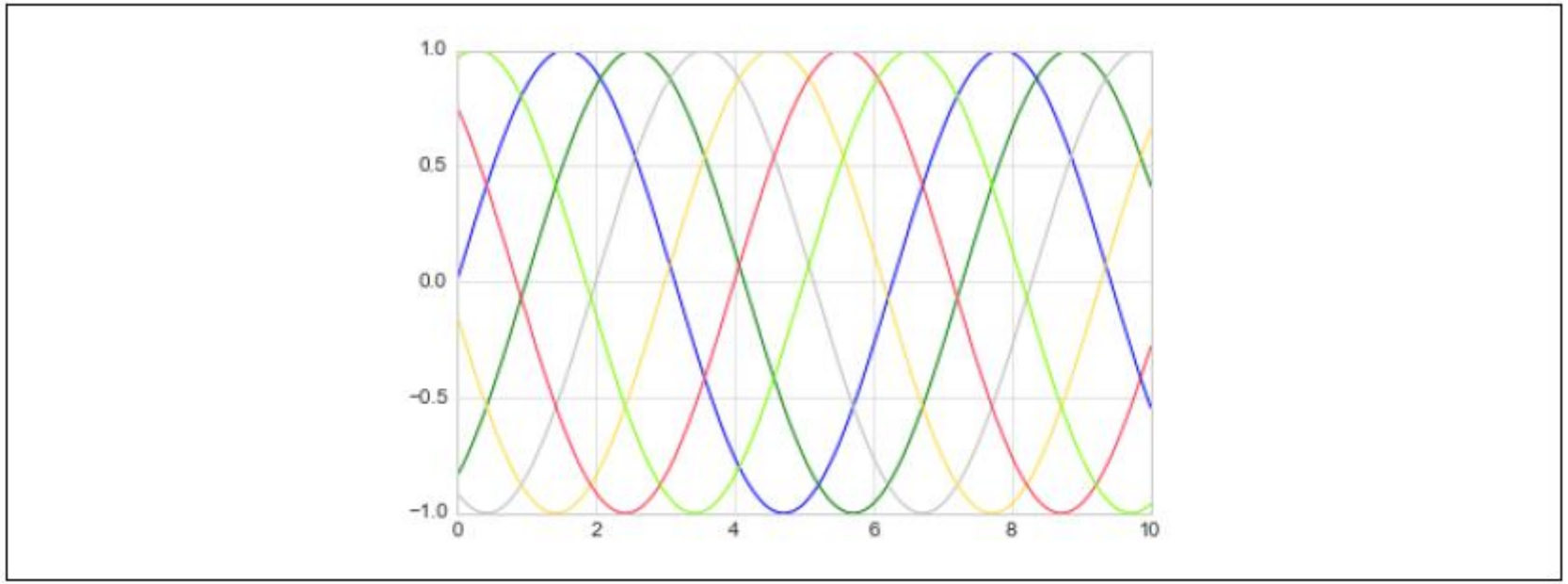

Figure 4-9. Controlling the color of plot elements

Adjusting the plot: Line colors and styles contd.

- If no color is specified, Matplotlib will automatically cycle through a set of default
- colors for multiple lines.
- Similarly, you can adjust the line style using the linestyle keyword (Figure 4-10):
- In[7]: plt.plot(x, x + 0, linestyle='solid')
- plt.plot(x, x + 1, linestyle='dashed')
- plt.plot(x, x + 2, linestyle='dashdot')
- plt.plot(x, x + 3, linestyle='dotted');
- *# For short, you can use the following codes:*
- plt.plot(x, x + 4, linestyle='-') *# solid*
- plt.plot(x, x + 5, linestyle='--') *# dashed*
- plt.plot(x, x + 6, linestyle='-.') *# dashdot*
- plt.plot(x, x + 7, linestyle=':'); *# dotted*

Adjusting the plot:line colors and styles contd.

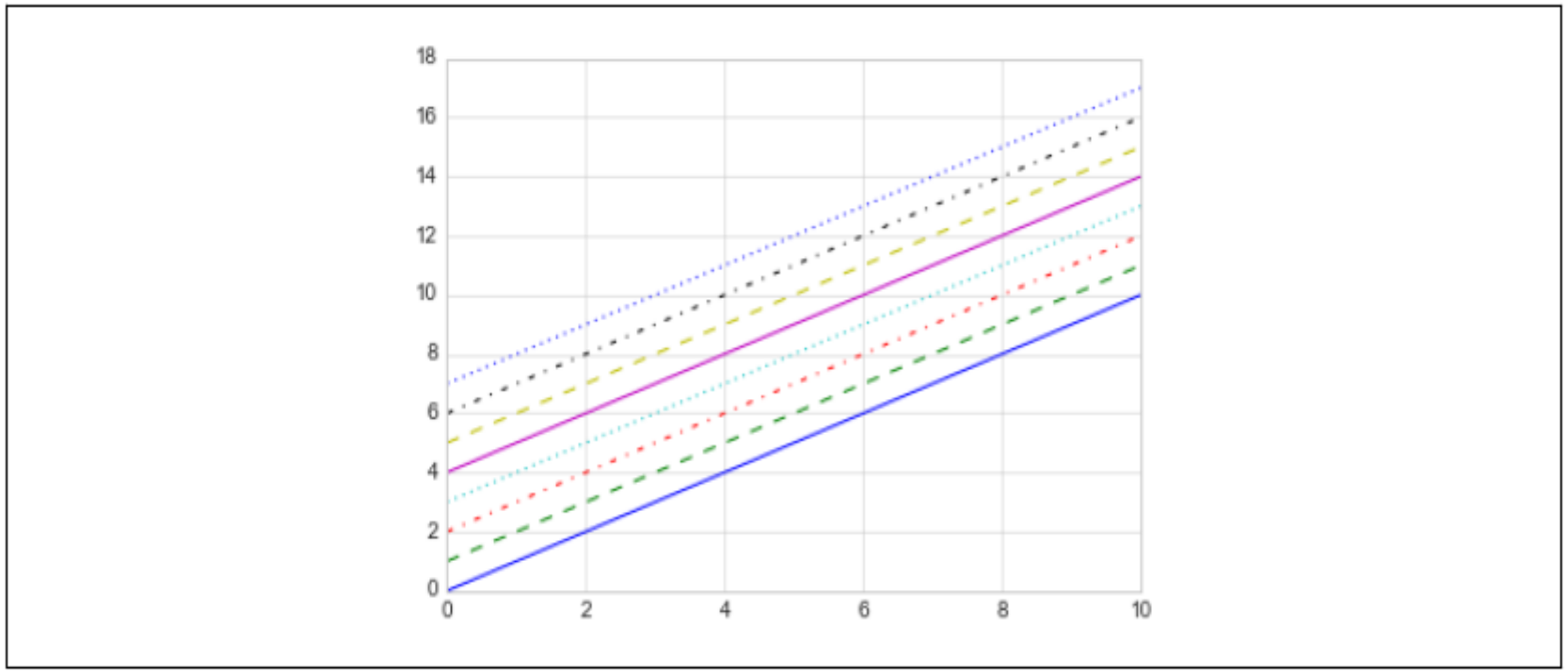

Figure 4-10. Example of various line styles

Adjusting the plot:line colors and styles contd.

- If you would like to be extremely terse, these linestyle and color codes can be combined nto a single nonkeyword argument to the plt.plot() function (Figure 4-11):
- In[8]: plt.plot(x, x + 0, '-g') *# solid greer*
- plt.plot(x, x + 1, '--c') *# dashed cyan*
- plt.plot(x, x + 2, '-.k') *# dashdot black*
- plt.plot(x, x + 3, ':r'); *# dotted red*

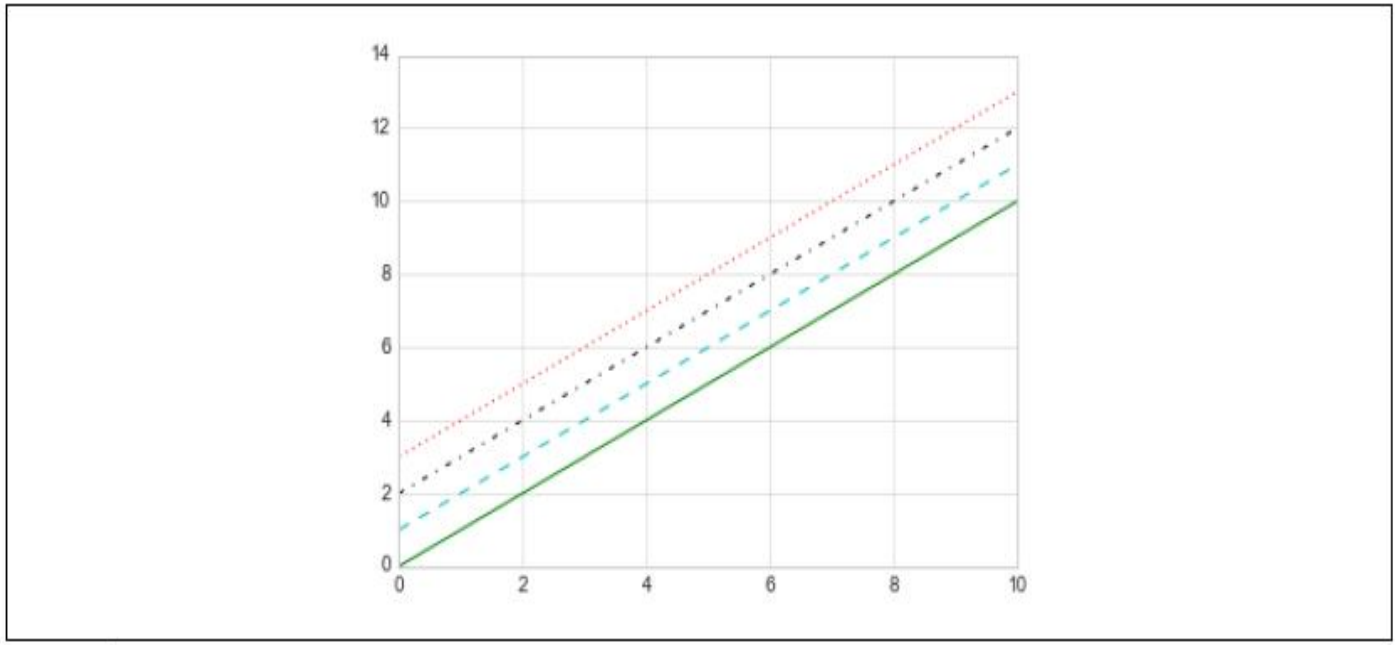

Figure 4-11. Controlling colors and styles with the shorthand syntax

Adjusting the plot: line colors and styles contd.

- These single-character color codes reflect the standard abbreviations in the RGB (Red/Green/Blue) and CMYK (Cyan/Magenta/Yellow/blacK) color systems, commonly used for digital color graphics.
- There are many other keyword arguments that can be used to fine-tune the appearance of the plot.

Adjusting the Plot: Axes Limits

- Matplotlib does a decent job of choosing default axes limits for your plot, but sometimes
- it's nice to have finer control. The most basic way to adjust axis limits is to use the plt.xlim() and plt.ylim() methods (Figure 4-12):
- In[9]: plt.plot(x, np.sin(x
- plt.xlim(-1, 11)
- plt.ylim(-1.5, 1.5);

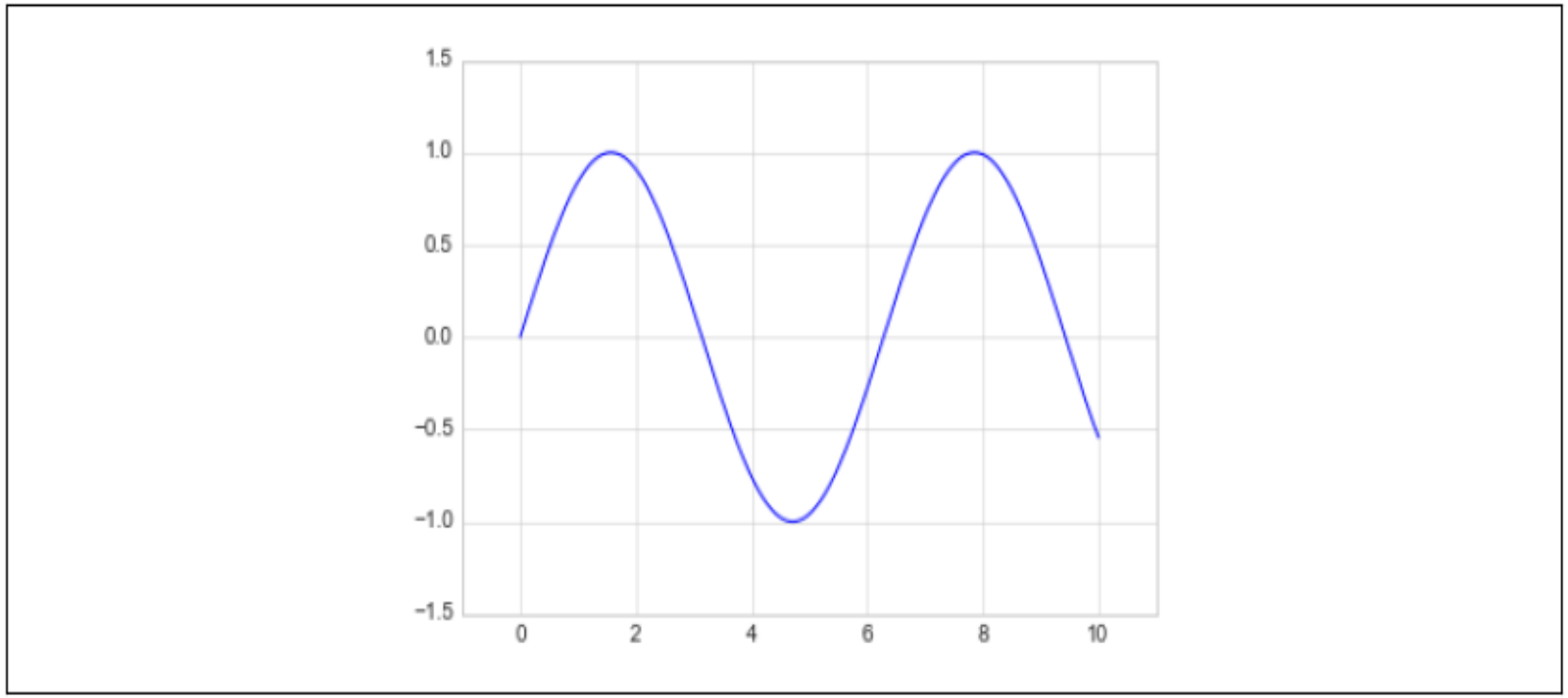

Figure 4-12. Example of setting axis limits

Adjusting the plot: Axes limits

- If for some reason you'd like either axis to be displayed in reverse, you can simply reverse the order of the arguments (Figure 4-13):
- In[10]: plt.plot(x, np.sin(x))
- plt.xlim(10, 0)
- plt.ylim(1.2, -1.2);

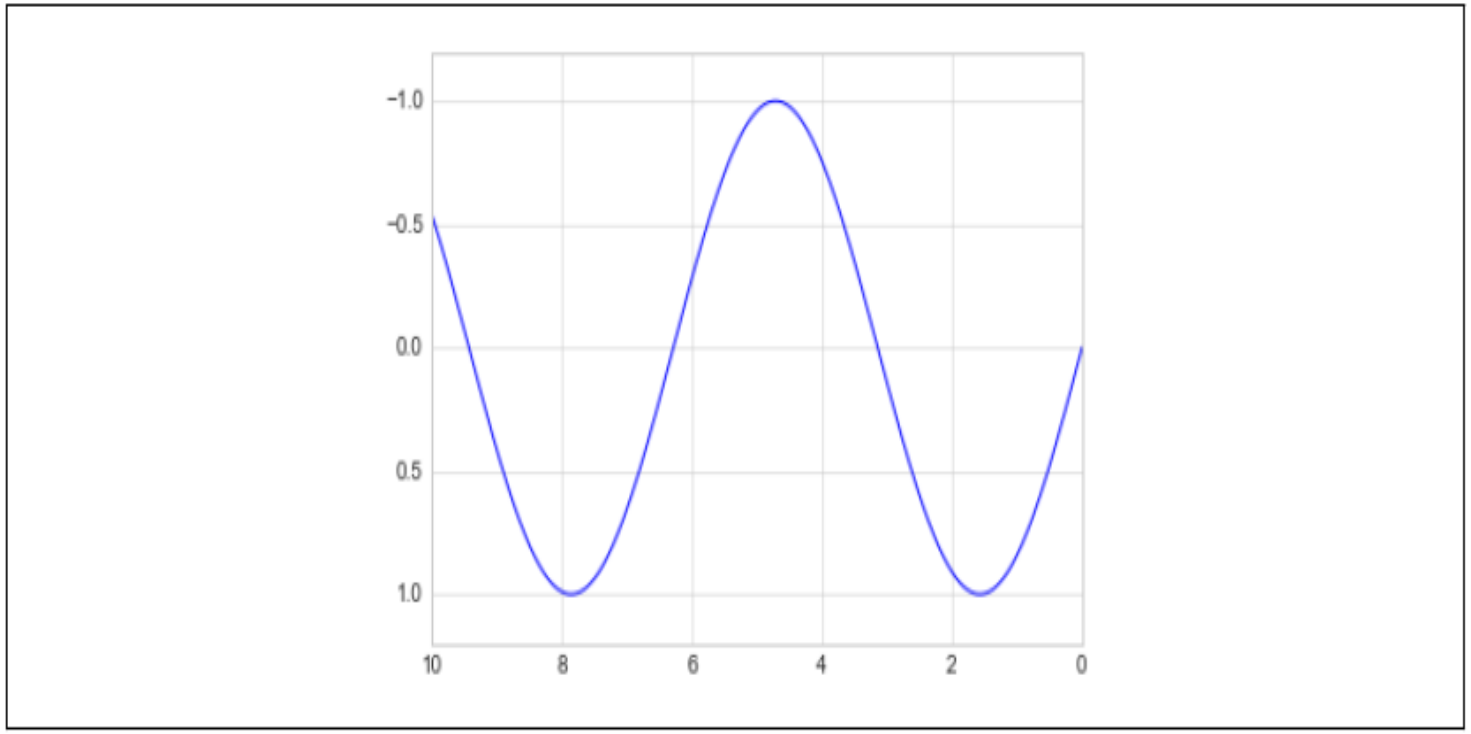

Figure 4-13. Example of reversing the y-axis

Adjusting the plot: Axes limits

- A useful related method is plt.axis() .The plt.axis() method allows you to set the x and y limits with a single call, by passing a list that specifies [xmin, xmax, ymin, ymax] (Figure 4-14):
- In[11]: plt.plot(x, np.sin(x)
- plt.axis([-1, 11, -1.5, 1.5]);

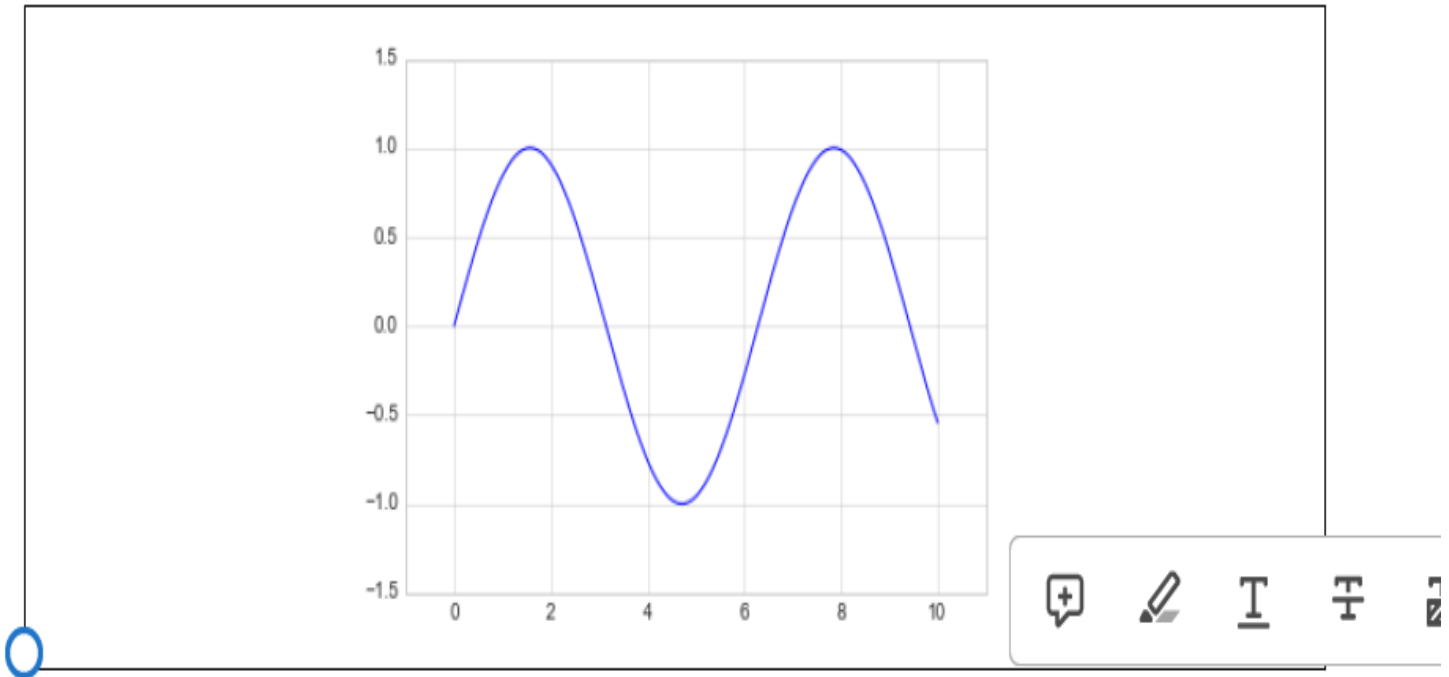

Figure 4-14. Setting the axis limits with plt.axis

Adjusting the Plot: Axes limits

- The plt.axis() method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot (Figure 4-15):
- In[12]: plt.plot(x, np.sin(x))
- plt.axis('tight');

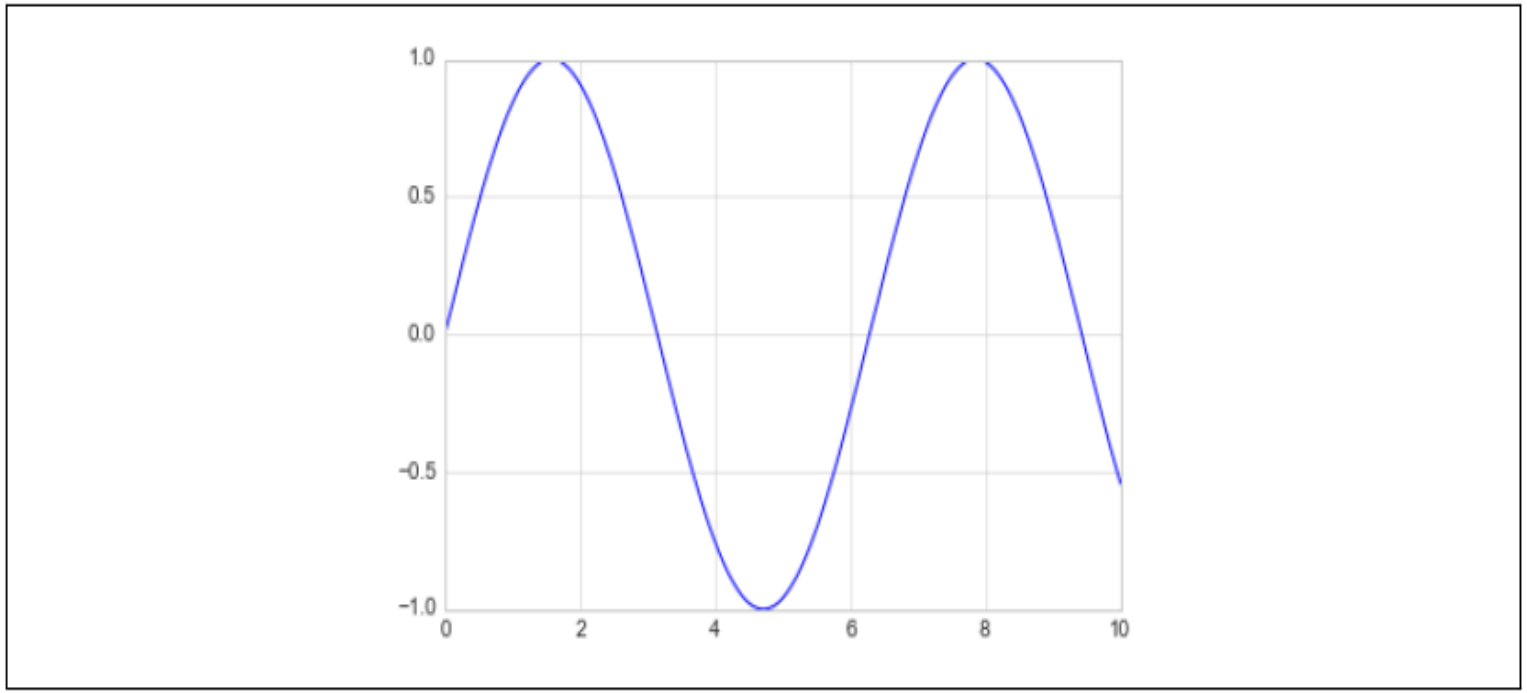

Figure 4-15. Example of a "tight" layout

Adjusting the plot: Axes limits

- It allows even higher-level specifications, such as ensuring an equal aspect ratio so that on your screen, one unit in x is equal to one unit in y (Figure 4-16):
- In[13]: plt.plot(x, np.sin(x))
- plt.axis('equal');

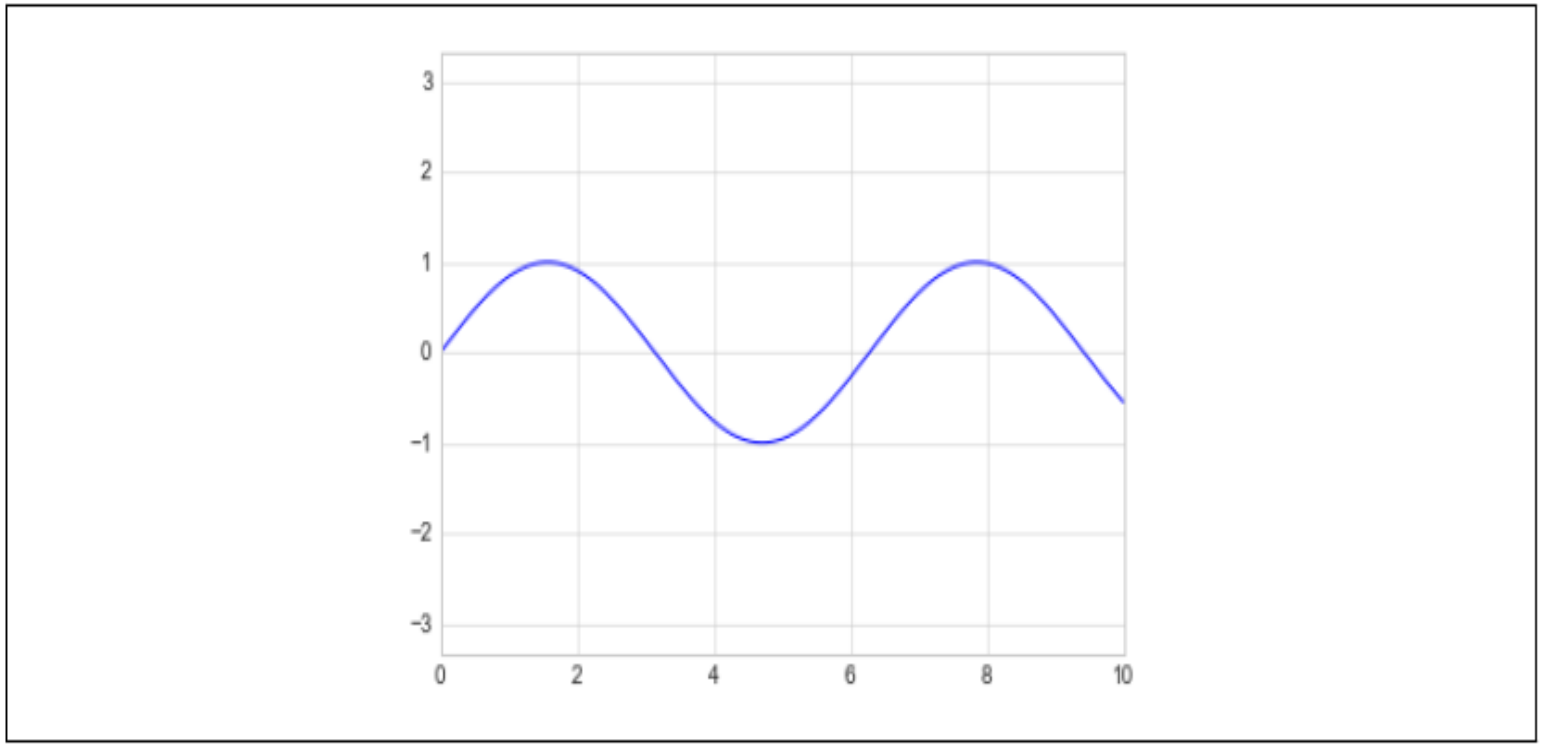

Figure 4-16. Example of an "equal" layout, with units matched to the output resolution

Labeling Plots

- we'll briefly look at the labeling of plots: titles, axis labels, and simple legends.
- Titles and axis labels are the simplest such labels—there are methods that can be used to quickly set them (Figure 4-17):
- In[14]: plt.plot(x, np.sin(x))
- plt.title("A Sine Curve")
- plt.xlabel("x")
- plt.ylabel("sin(x)");

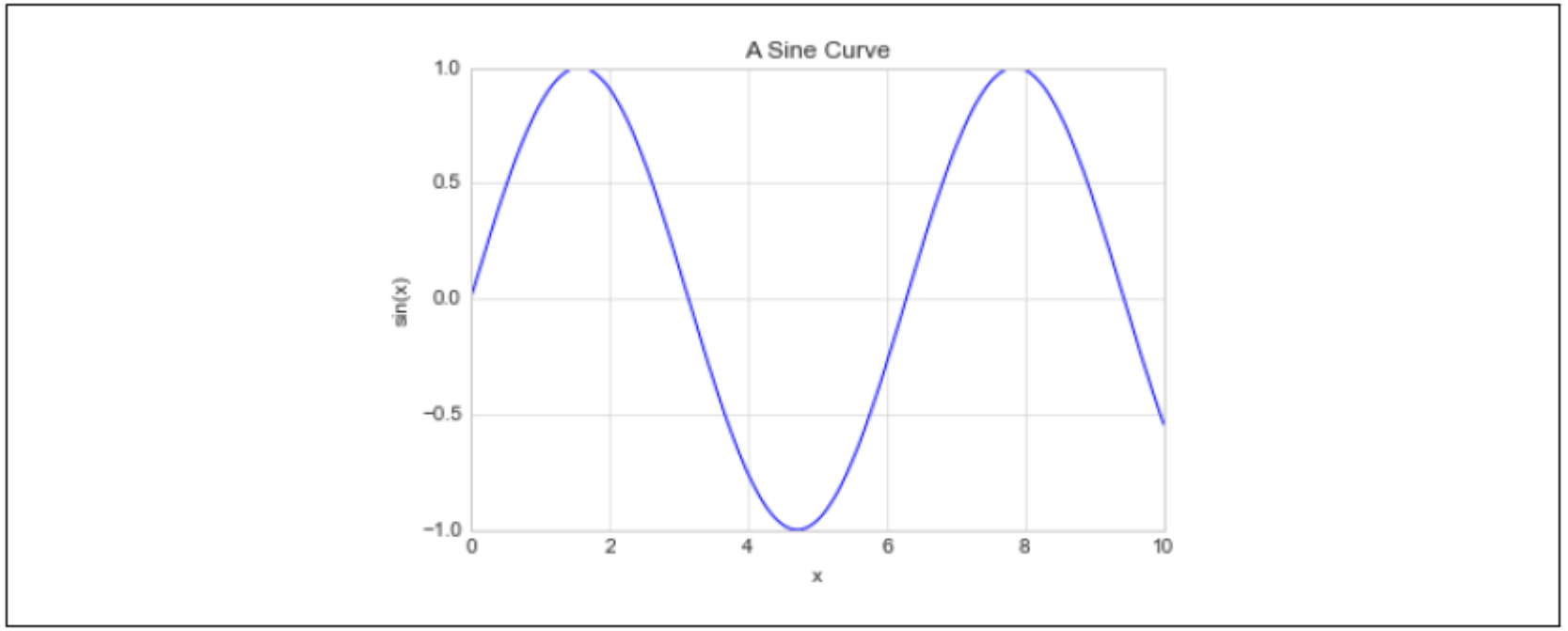

Figure 4-17. Examples of axis labels and title

Labeling Plots contd.

- You can adjust the position, size, and style of these labels using optional arguments to the function. For more information, see the Matplotlib documentation and the docstrings of each of these functions.
- When multiple lines are being shown within a single axes, it can be useful to create a plot legend that labels each line type.
- Again, Matplotlib has a built-in way of quickly creating such a legend. It is done via the (you guessed it) plt.legend() method.
- Though there are several valid ways of using this, I find it is easier to specify the label of each line using the label keyword of the plot function (Figure 4-18):
- In[15]: plt.plot(x, np.sin(x), '-g', label='sin(x)')
- plt.plot(x, np.cos(x), ':b', label='cos(x)')
- plt.axis('equal')
- plt.legend();

Labeling plots contd.

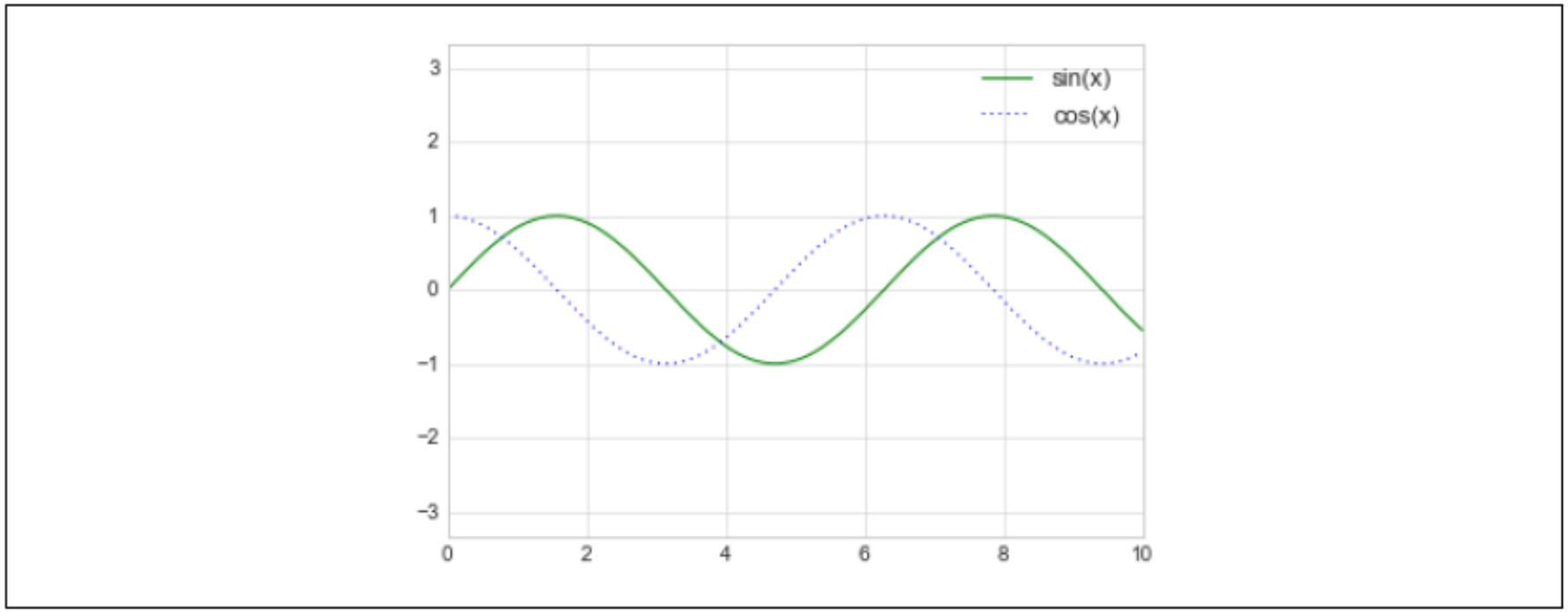

Figure 4-18. Plot legend example

Labeling plots contd.

- The plt.legend() function keeps track of the line style and color, and matches these with the correct label.

Matplotlib Gotchas

- While most plt functions translate directly to ax methods (such as plt.plot() →
- ax.plot(), plt.legend() → ax.legend(), etc.), this is not the case for all commands.
- In particular, functions to set limits, labels, and titles are slightly modified.
- For transitioning between MATLAB-style functions and object-oriented methods, make the following changes:
- • plt.xlabel() → ax.set_xlabel()
- • plt.ylabel() → ax.set_ylabel()
- • plt.xlim() → ax.set_xlim()
- • plt.ylim() → ax.set_ylim()
- • plt.title() → ax.set_title()
- In the object-oriented interface to plotting, rather than calling these functions individually, it is often more convenient to use the ax.set() method to set all these properties at once (Figure 4-19):

Matplotlib gotchas contd.

- In[16]: ax = plt.axes()
- ax.plot(x, np.sin(x))
- ax.set(xlim=(0, 10), ylim=(-2, 2),
- xlabel='x', ylabel='sin(x)',
- title='A Simple Plot');

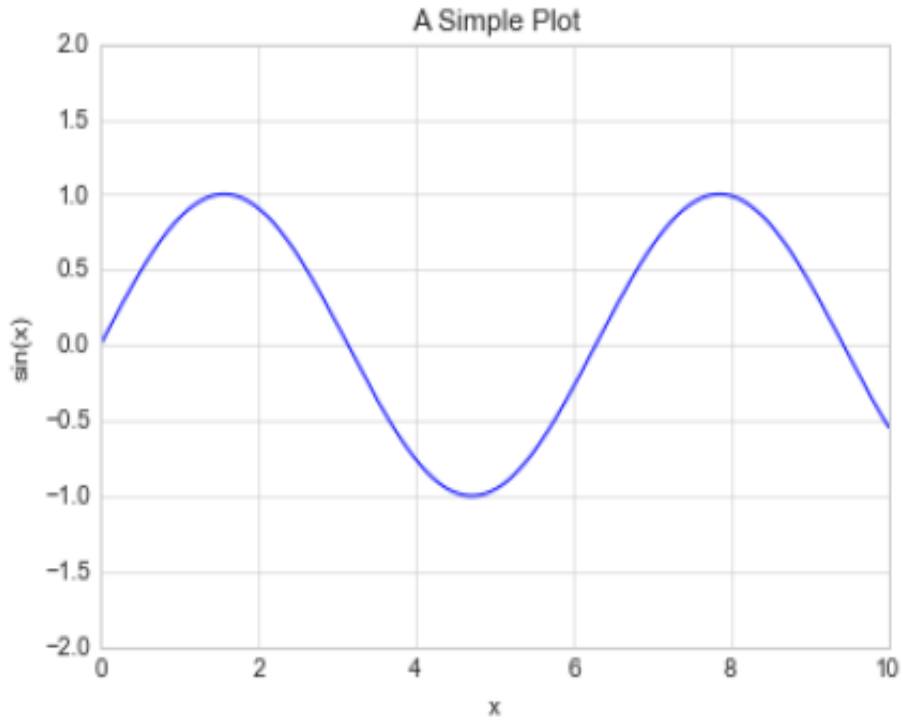

Figure 4-19. Example of using ax.set to set multiple properties at once

THANK YOU