Visualization with Matplotlib

Matplotlib is a multi-platform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.

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.

General Matplotlib Tips

Before we dive into the details of creating visualizations with Matplotlib, there are a few useful things you should know about using the package.

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:

```python
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:

```python
plt.style.use('classic')
```

Throughout this section, we will adjust this style as needed. Note that the stylesheets used here are supported as of Matplotlib version 1.5

show() or No show() - How to Display Your Plots

A visualization you can't see won't be of much use, but just how you view your Matplotlib plots depends on the context. The best use of Matplotlib differs depending on how you are using it; roughly, the three applicable contexts are using Matplotlib in a script, in an IPython terminal, or in an IPython notebook.

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.

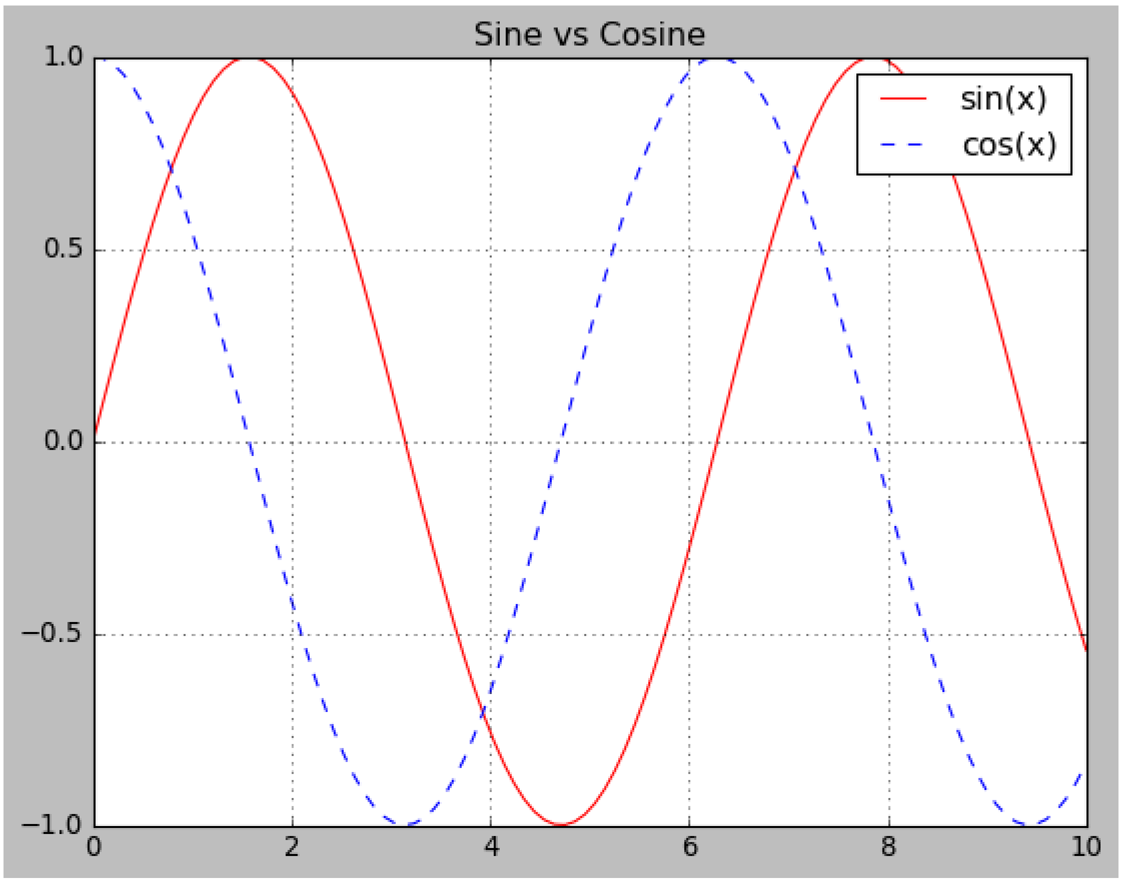
Plotting interactively within an IPython notebook can be done with the %matplotlib command, and works in a similar way to the IPython shell. In the IPython notebook, you also have the option of embedding graphics directly in the notebook, with two possible options:

- %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

For this notebook, we will generally opt for %matplotlib inline:

In [19…

```python
%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np

plt.style.use("classic")

# ToDo: create np-array ranging from 0 to 10 with 100 samples (Hint! 
x = np.linspace(0, 10, 100)
sin_x = np.sin(x)
cos_x = np.cos(x)
# print(x)
# print(sin_x)
# print(cos_x)

# ToDo: Plot the x vs sin(x) using one dash line and plot to the same
plt.grid(True)
plt.plot(x, sin_x, "r-", label="sin(x)")
plt.plot(x, cos_x, "b--", label="cos(x)")
plt.title("Sine vs Cosine")
plt.legend()

plt.savefig("my_figure.png")
plt.show()
```



Output Figure:

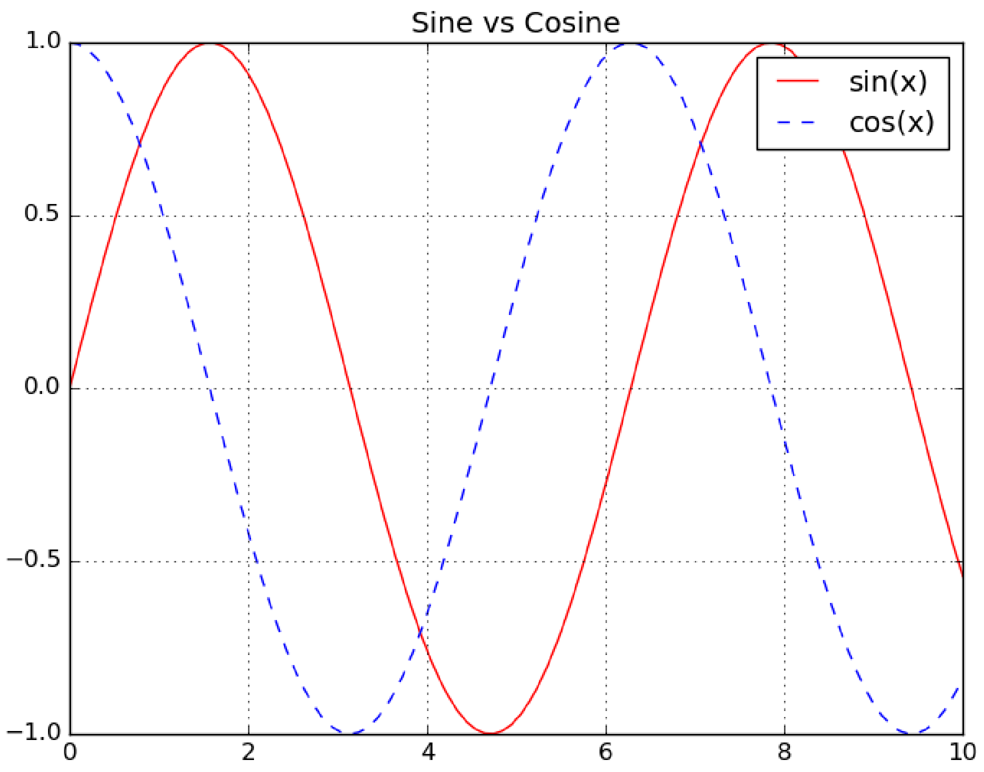
Saving Figures to File

One nice feature of Matplotlib is the ability to save figures in a wide variety of formats. Saving a figure can be done using the savefig() command. For example, to save the previous figure as a PNG file, you can run this:

In [ ]:
```
# ToDo: save the previous figure to the file called 'my_figure.png'
# done already in previous cell
```

In [20…
```
# just to test the image
# !ls -lh my_figure.png

from IPython.display import Image
Image('my_figure.png')
```

Out[20…



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. The list of supported file types can be found for your system by using the following method of the figure canvas object:

> Note that when saving your figure, it's not necessary to use plt.show() or related commands discussed earlier.

In [22…

```
# ToDo: Find out which filetypes are supported by your backend (Hint:
from PIL import Image

supported_formats = Image.registered_extensions()
print("Supported image file types:")
for key, value in supported_formats.items():
    print(f"{key}: {value}")
```

```
Supported image file types:
.png: PNG
.apng: PNG
.bmp: BMP
.dib: DIB
.gif: GIF
.jfif: JPEG
.jpe: JPEG
.jpg: JPEG
.jpeg: JPEG
.pbm: PPM
.pgm: PPM
.ppm: PPM
.pnm: PPM
.pfm: PPM
.blp: BLP
.bufr: BUFR
.cur: CUR
.pcx: PCX
.dcx: DCX
.dds: DDS
.ps: EPS
.eps: EPS
.fit: FITS
.fits: FITS
.fli: FLI
.flc: FLI
.ftc: FTEX
.ftu: FTEX
.gbr: GBR
.grib: GRIB
.h5: HDF5
.hdf: HDF5
.jp2: JPEG2000
.j2k: JPEG2000
.jpc: JPEG2000
.jpf: JPEG2000
.jpx: JPEG2000
.j2c: JPEG2000
.icns: ICNS
.ico: ICO
.im: IM
.iim: IPTC
.mpg: MPEG
.mpeg: MPEG
.tif: TIFF
.tiff: TIFF
.mpo: MPO
.msp: MSP
.palm: PALM
.pcd: PCD
.pdf: PDF
```

```
.pxr: PIXAR
.psd: PSD
.qoi: QOI
.bw: SGI
.rgb: SGI
.rgba: SGI
.sgi: SGI
.ras: SUN
.tga: TGA
.icb: TGA
.vda: TGA
.vst: TGA
.webp: WEBP
.wmf: WMF
.emf: WMF
.xbm: XBM
.xpm: XPM
```

Two Interfaces for plotting

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:

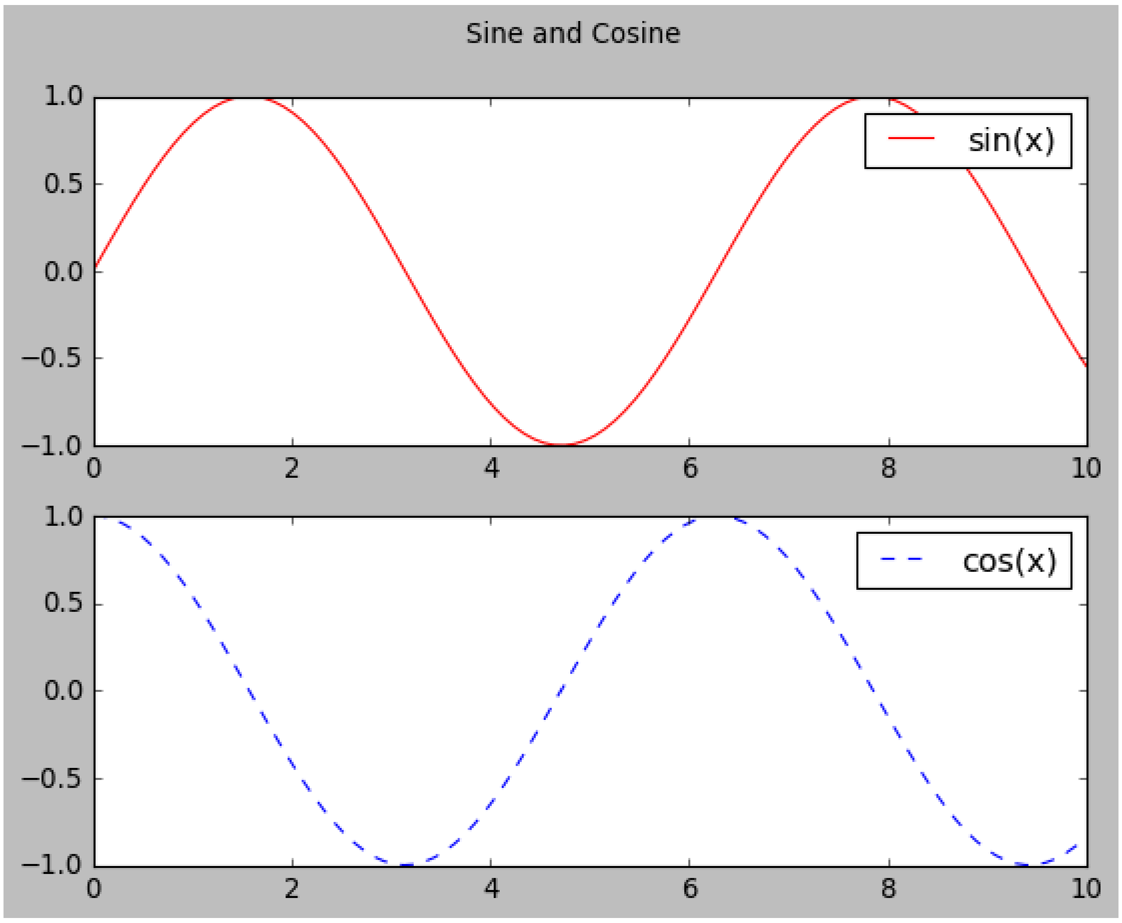
It is important to note that this interface is stateful: it keeps track of the "current" figure and axes, which are where all plt commands are applied. You can get a reference to these using the plt.gcf() (get current figure) and plt.gca() (get current axes) routines.

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.

In [39…

```
# ToDo: Create two subplots: 1st plots the sin(x) and the 2nd plots t
# Hint: Use matlab stype interface, which is the pyplot

# make sure variables still stored
#print(sin_x)
#print(cos_x)
```

```
# make a new plot
import matplotlib.pyplot as plt2

plt2.figure(figsize=(8,6))
plt2.subplot(211)
plt2.plot(x, sin_x, "r-", label="sin(x)")
plt2.legend()
plt2.subplot(212)
plt2.plot(x, cos_x, "b--", label="cos(x)")
plt2.suptitle("Sine and Cosine")
plt2.legend()
plt2.show()



# create the first of two panels and set current axis

# create the second panel and set current axis
```



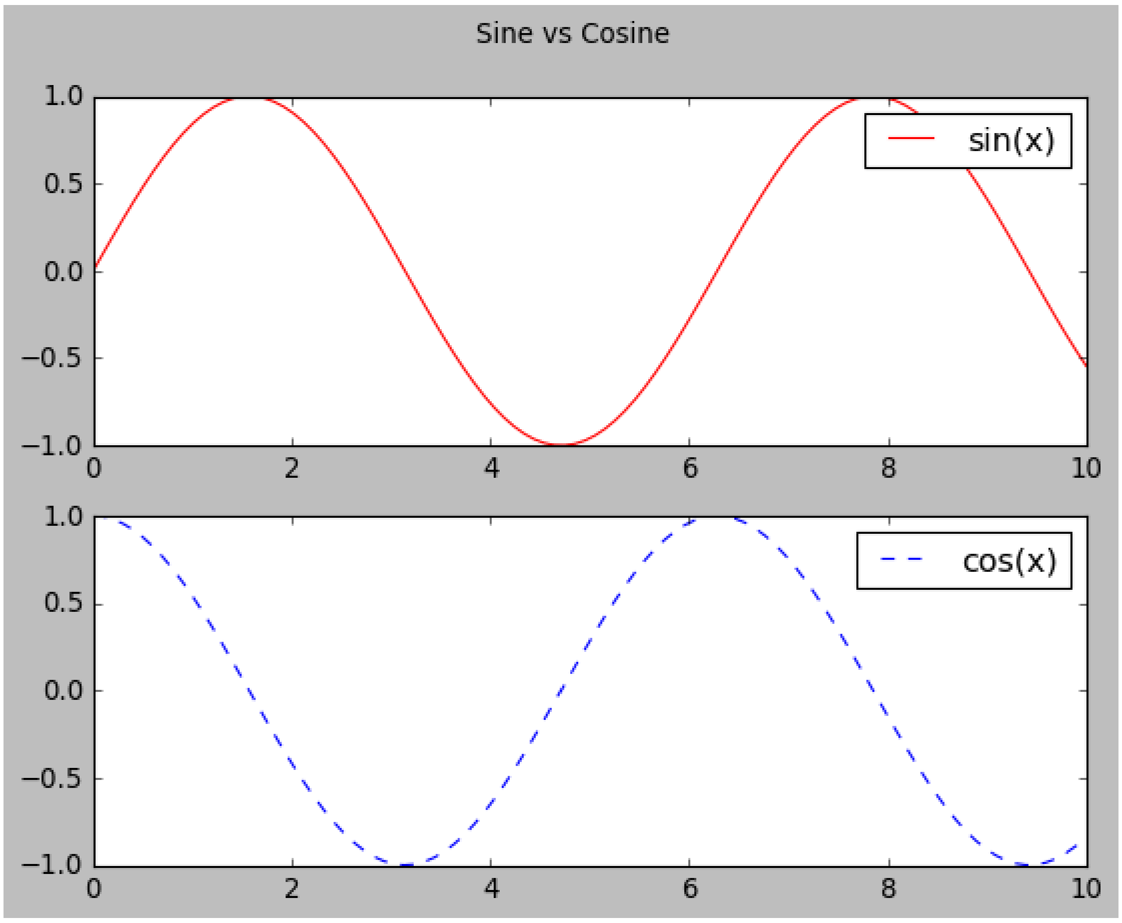
Output Figure:

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:

In [44…

```python
# ToDo: plot the same figure using Object-oriented interface
# First create a grid of plots
# ax will be an array of two Axes objects


# Call plot() method on the appropriate object

import matplotlib.pyplot as plt3

class Compare_Two_Values_Plotter:
    def __init__(self, x_1: np.array, y1: np.array, y2: np.array, plt
        self.x = x
        self.y1 = y1
        self.y2 = y2
        # plt3.subplots contains both the figure and the axis
        # 2, 1 means 2 rows, 1 column of subplots
        # figsize defines how large image
        self.fig, self.axs = plt3.subplots(2, 1, figsize=(8, 6))
        self.label1 = label1
        self.label2 = label2
        self.suptitle = suptitle

    def plot(self):
        self.axs[0].plot(self.x, self.y1, "r-", label=self.label1)
        self.axs[0].legend()
        self.axs[1].plot(self.x, self.y2, "b--", label=self.label2)
        self.axs[1].legend()
        self.fig.suptitle(self.suptitle)
        plt3.show()

plotter = Compare_Two_Values_Plotter(x, sin_x, cos_x, plt3, "sin(x)",
plotter.plot()
```



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. Throughout this chapter, we will switch between the MATLAB-style and object-oriented interfaces, depending on what is most convenient. In most cases, the difference is as small as switching plt.plot() to ax.plot(), but there are a few gotchas that we will highlight as they come up in the following sections.

Adjusting the Plot: Line Colors and Styles

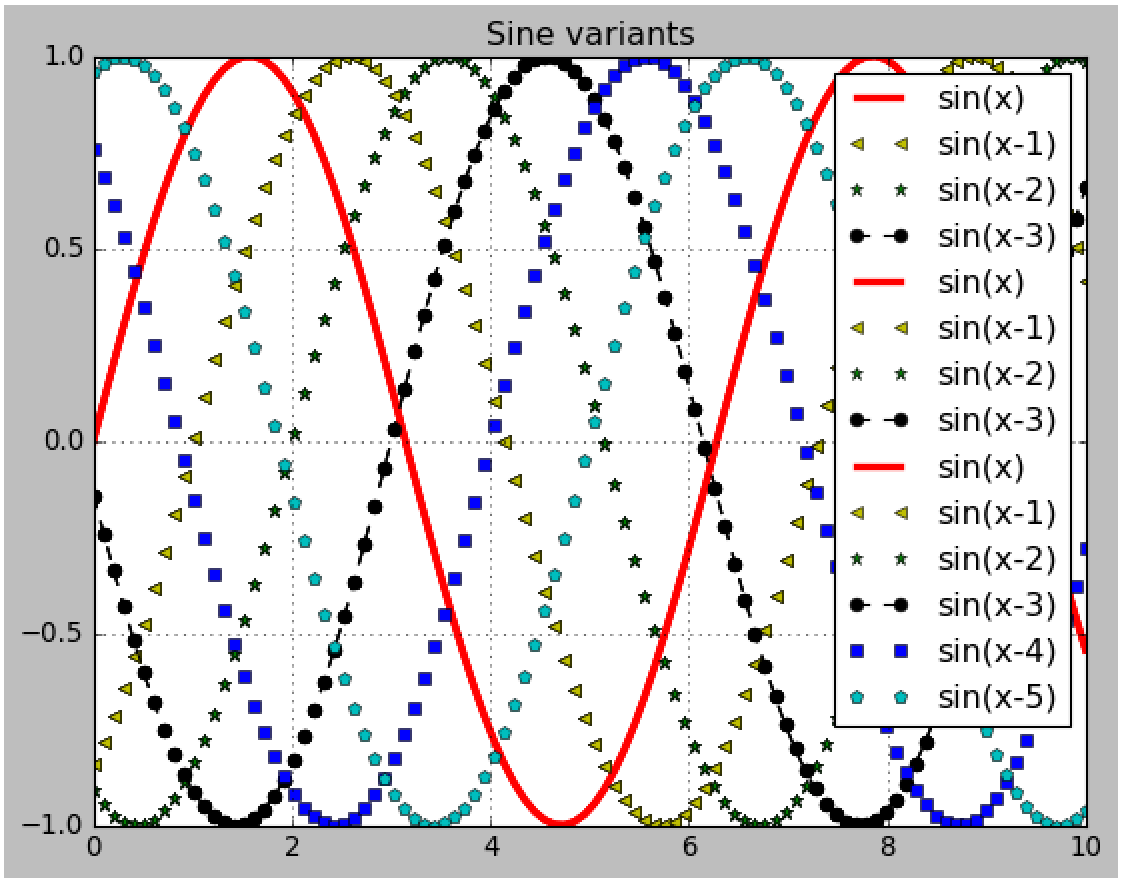
The first adjustment you might wish to make to a plot is to control the line colors and styles. The plt.plot() function takes additional arguments that can be used to specify these. To adjust the color, you can use the color keyword, which accepts a string argument representing virtually any imaginable color. The color can be specified in a variety of ways:

> If no color is specified, Matplotlib will automatically cycle through a set of default colors for multiple lines.

In [64…

```
# ToDo: plot 6 sin-curves (from (x) to (x -5)) using different color (

import matplotlib.pyplot as plt4
plt4.style.use("classic")
```

```
# ToDo: Plot the x vs sin(x) using one dash line and plot to the same
plt4.grid(True)
plt4.plot(x, sin_x, "r-", label="sin(x)", linewidth=3)
plt4.plot(x, np.sin(x-1), "y<", label="sin(x-1)")
plt4.plot(x, np.sin(x-2), "g*", label="sin(x-2)")
plt4.plot(x, np.sin(x-3), "ko--", label="sin(x-3)")
plt4.plot(x, np.sin(x-4), "bs", label="sin(x-4)")
plt4.plot(x, np.sin(x-5), "cp", label="sin(x-5)")
plt4.title("Sine variants")
plt4.legend()

plt4.show()
```



Output Figure:

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:

A useful related method is plt.axis() (note here the potential confusion between axes with an e, and axis with an i). The plt.axis() method allows you to set the x and y limits with a single call, by passing a list which specifies [xmin, xmax, ymin, ymax]:

The plt.axis() method goes even beyond this, allowing you to do things like automatically tighten the bounds around the current plot:

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:

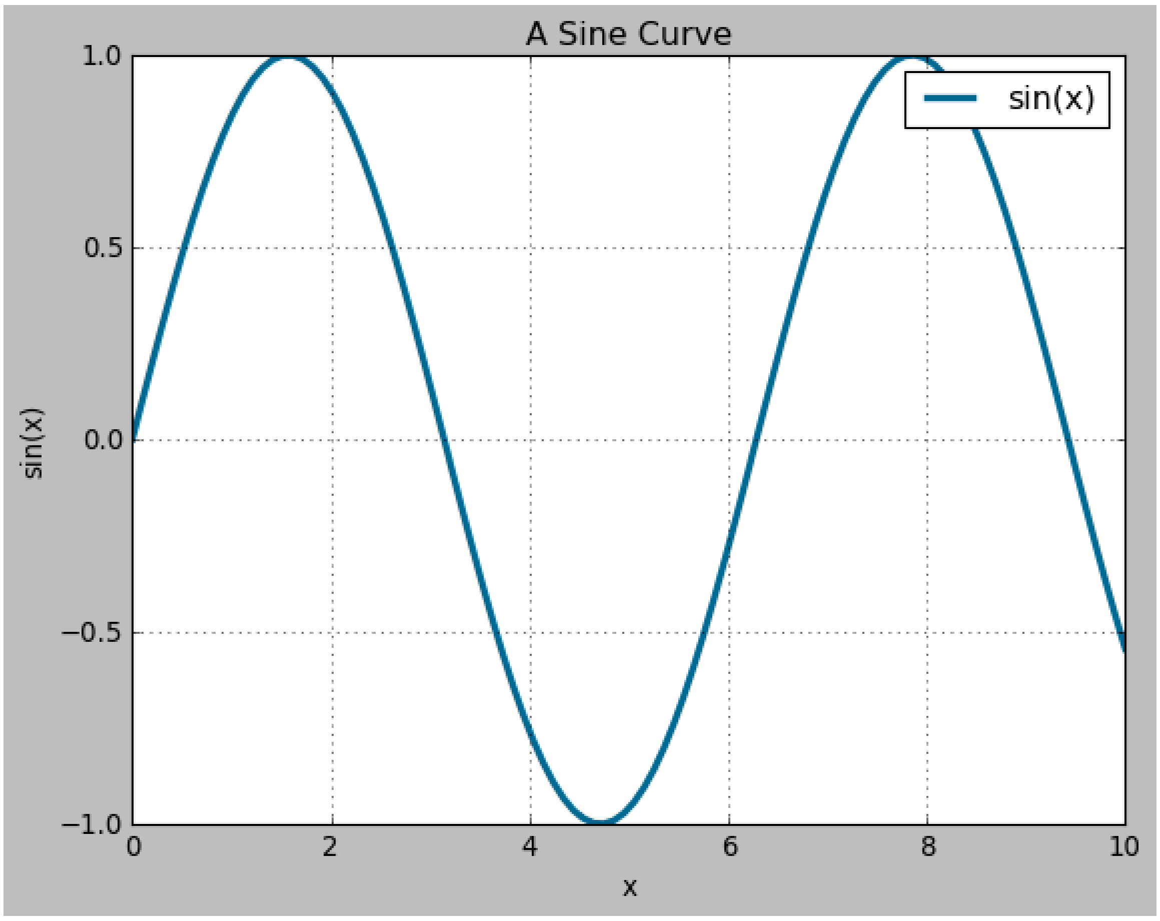
Labeling Plots

As the last piece of this section, 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:

In [66…

```
# ToDo: plot x vs sin(x) and
# - set title of the plot to "A Sine curve"
# - set the x axel label to x
# - set the y axel label to sin(x)

import matplotlib.pyplot as plt5
plt5.style.use("classic")

# ToDo: Plot the x vs sin(x) using one dash line and plot to the same
plt5.grid(True)
plt5.plot(x, sin_x, "#006994", label="sin(x)", linewidth=3)
plt5.title("A Sine Curve")
plt5.ylabel("sin(x)")
plt5.xlabel("x")
plt5.legend()

plt5.show()
```



Output Figure:

Simple Scatter Plots

Another commonly used plot type is the simple scatter plot, a close cousin of the line plot. Instead of points being joined by line segments, here the points are represented individually with a dot, circle, or other shape. We'll start by setting up the notebook for plotting and importing the functions we will use:

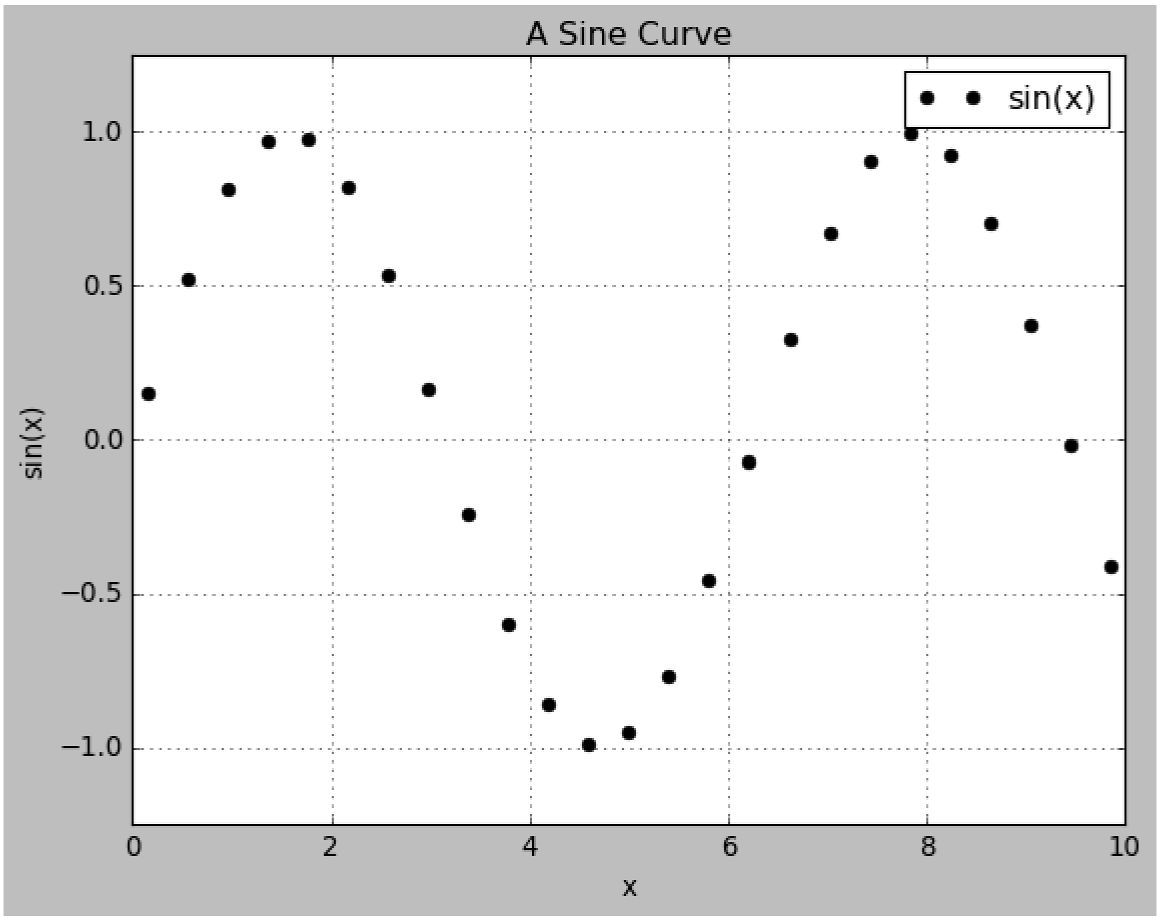
Scatter Plots with plt.plot

In the previous section we looked at plt.plot/ax.plot to produce line plots. It turns out that this same function can produce scatter plots as well:

The third argument in the function call is a character that represents the type of symbol used for the plotting. Just as you can specify options such as '-', '--' to control the line style, the marker style has its own set of short string codes. The full list of available symbols can be seen in the documentation of plt.plot, or in Matplotlib's online documentation. Most of the possibilities are fairly intuitive, and we'll show a number of the more common ones here:

In [84…

```
# ToDo: Plot sin curve using just dots (use plt.plot() function)
# Hint: set x axel values from 0 to 10 using 30 items

import matplotlib.pyplot as plt6
plt6.style.use("classic")

# ToDo: Plot the x vs sin(x) using one dash line and plot to the same
# to show 100 data pairs as 30 data pairs, i will combine every 4 dat

x_sample = [np.mean(x[i:i+4]) for i in range(0, len(x), 4)]
sin_x_sample = [np.mean(sin_x[i:i+4]) for i in range (0, len(sin_x),

plt6.grid(True)
plt6.plot(x_sample, sin_x_sample, "ko", label="sin(x)")
plt6.title("A Sine Curve")
plt6.ylabel("sin(x)")
plt6.xlabel("x")
plt6.legend()

plt6.xlim(0,10)
plt6.ylim(-1.25, 1.25)

plt6.show()
```



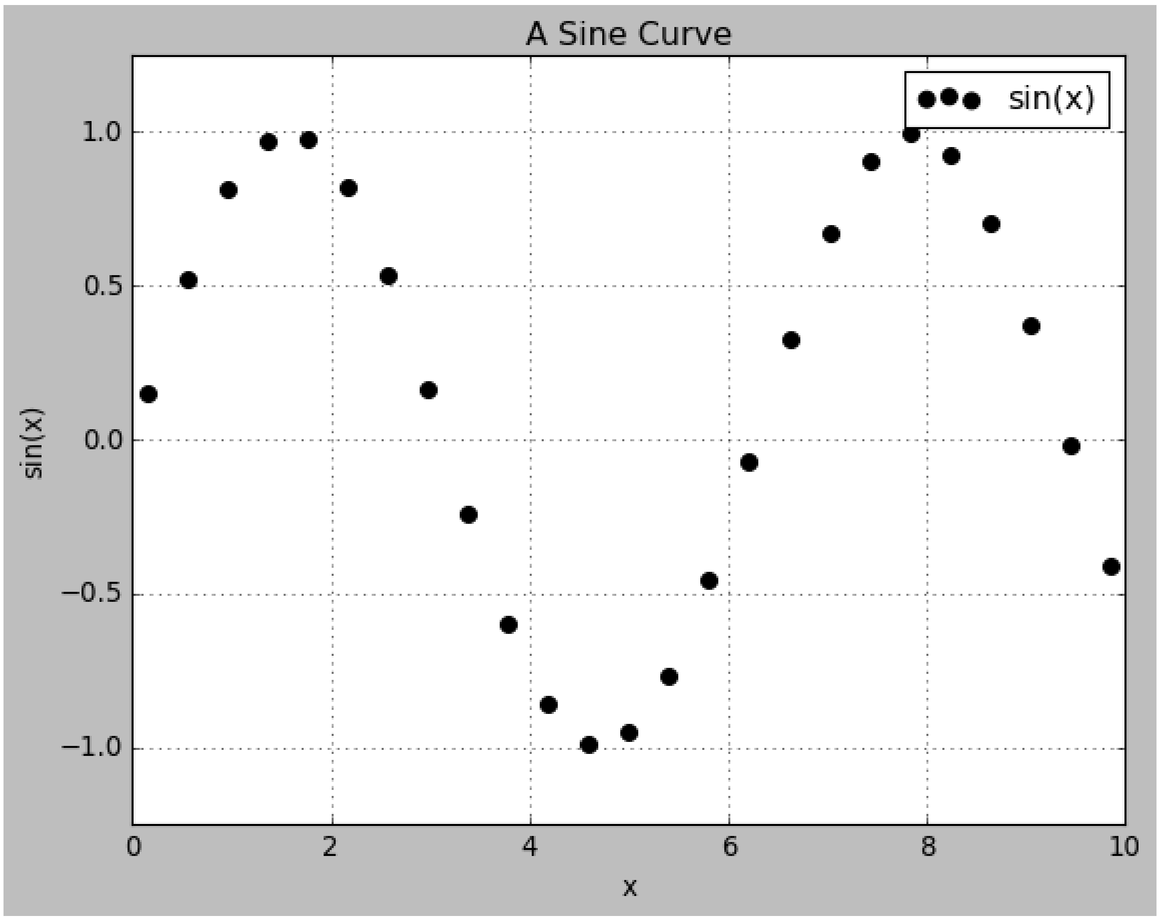
Output Figure:

Scatter Plots with plt.scatter

A second, more powerful method of creating scatter plots is the plt.scatter function, which can be used very similarly to the plt.plot function:

In [85…

```
# ToDo: Plot the same figure as above, but use the scatter function i

# ToDo: Plot sin curve using just dots (use plt.plot() function)
# Hint: set x axel values from 0 to 10 using 30 items

import matplotlib.pyplot as plt7
plt7.style.use("classic")

# ToDo: Plot the x vs sin(x) using one dash line and plot to the same
# to show 100 data pairs as 30 data pairs, i will combine every 4 dat

x_sample = [np.mean(x[i:i+4]) for i in range(0, len(x), 4)]
sin_x_sample = [np.mean(sin_x[i:i+4]) for i in range (0, len(sin_x),

plt7.grid(True)
plt7.scatter(x_sample, sin_x_sample, c="k", s=50, label="sin(x)")
plt7.title("A Sine Curve")
plt7.ylabel("sin(x)")
plt7.xlabel("x")
plt7.legend()
plt7.xlim(0,10)
plt7.ylim(-1.25, 1.25)

plt7.show()
```



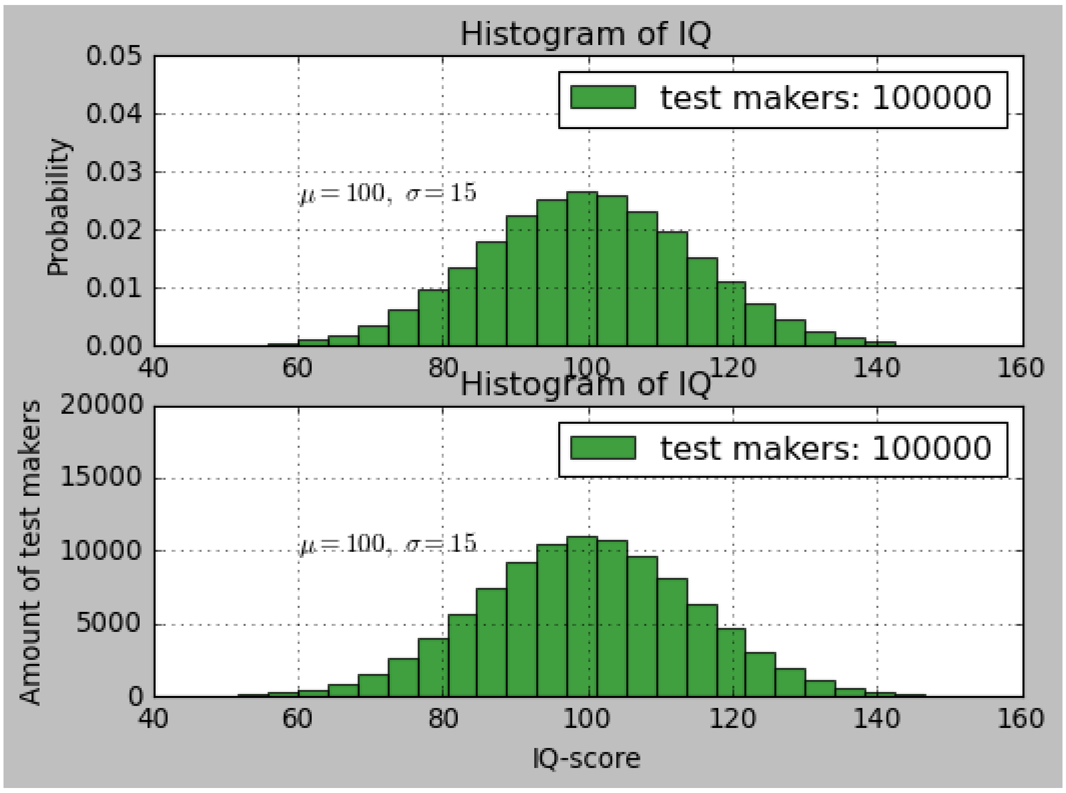
The primary difference of plt.scatter from plt.plot is that it can be used to create scatter plots where the properties of each individual point (size, face color, edge color, etc.) can be individually controlled or mapped to data.

Histograms, Binnings, and Density

A simple histogram can be a great first step in understanding a dataset.

In [10…

```
# ToDo: generate 10000 random samples and plot the histogram of the d
# Note! the histogram changes at every run

# learn to make a histogram
# make 2x IQ-score histograms, first probability, below amount of tes
# other showing probability, other amount of test makers

import matplotlib.pyplot as plt8

MOOOO, sigma, bins, how_many_tested = 100, 15, 30, 100000
x = MOOOO + sigma * np.random.randn(how_many_tested)


plt8.figure(figsize=(7, 5))

# the hist function has next parameters:
# x = data for histogram
# bins = the amount of pilars in histogram
# density = True means showing probability rather than amounts
#           most meaningful in normalized data
# facebolor = color of pilars
# alpha=0.75 = pilars show through 25%
plt8.subplot(211)
plt8.style.use("classic")
n, bins, patches = plt8.hist(
    x,
    bins,
    density=True,
    facecolor="g",
    alpha=0.75,
    label=f"test makers: {how_many_tested}",
)
plt8.ylabel("Probability")
plt8.title("Histogram of IQ")
plt8.text(60, 0.025, r"$\mu=100,\ \sigma=15$")
plt8.axis([40, 160, 0, 0.05])
plt8.grid(True)
plt8.legend()

plt8.subplot(212)
n, bins, patches = plt8.hist(
    x,
    bins,
    density=False,
    facecolor="g",
    alpha=0.75,
    label=f"test makers: {how_many_tested}",
)
plt8.xlabel("IQ-score")
plt8.ylabel("Amount of test makers")
plt8.title("Histogram of IQ")
plt8.text(60, how_many_tested / 5 / 2, r"$\mu=100,\ \sigma=15$")
plt8.axis([40, 160, 0, how_many_tested / 5])
plt8.grid(True)
plt8.legend()
plt8.show()
```



Output Figure:

```
In [ ]:
```