Northeastern University

DS 5010 – Lecture 3
Matplotlib

Roi Yehoshua

Agenda

Northeastern University

- Data visualization
- Plot types
- Matplotlib
- Seaborn

Data Visualization

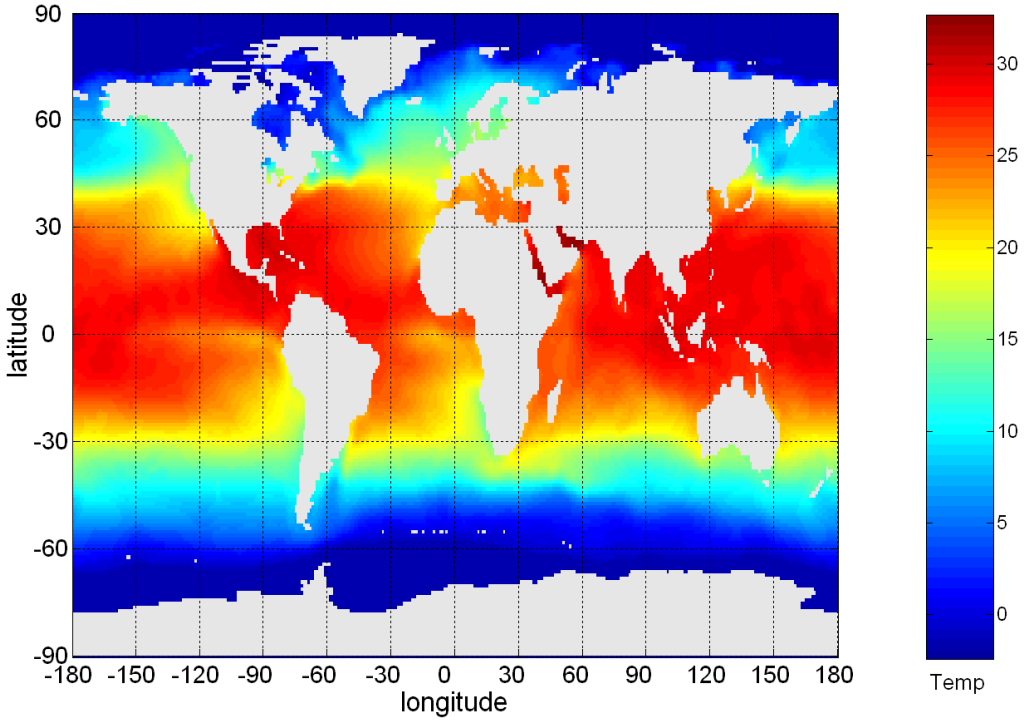
Northeastern University

- Visualization is the conversion of data into a visual or tabular format
  - **Exploratory visualization** is used to understand the data, discover patterns, and gain new insights, which then leads to deeper analysis
  - **Explanatory visualization** offers an explanation after exploration and analysis are done
- Motivations for visualization:
  - Humans can quickly absorb large amounts of visual information
  - Can detect general patterns and trends
  - Can detect outliers and unusual patterns
- Visualization goal: provide to the viewer the greatest number of ideas in the shortest time with the least ink in the smallest space

Example: Sea Surface Temperature

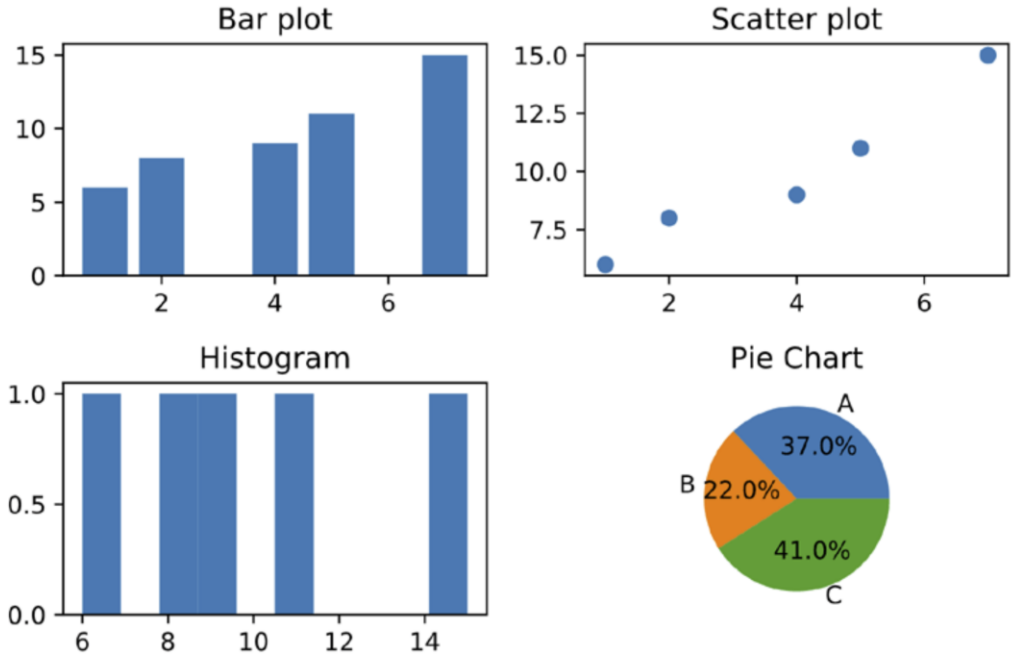
Northeastern University

- The following shows the Sea Surface Temperature (SST) for July 1982
- About 250,000 data points are summarized in a single figure



Plot Types

- **Line/scatter plots** are used to present a relationship between two variables
- **Histograms** are used to display numerical variable distributions
- **Bar charts** or **pie charts** are used to display categorical variable distributions



Matplotlib

Northeastern University

- Matplotlib is a Python library for plotting and data visualization
- Can be used to plot a wide range of graphs
- Designed to be as usable as MATLAB, with the ability to integrate Python code

- **pyplot** is the top-level module which provides simple functions for creating plots
- Typically imported under the alias **plt**

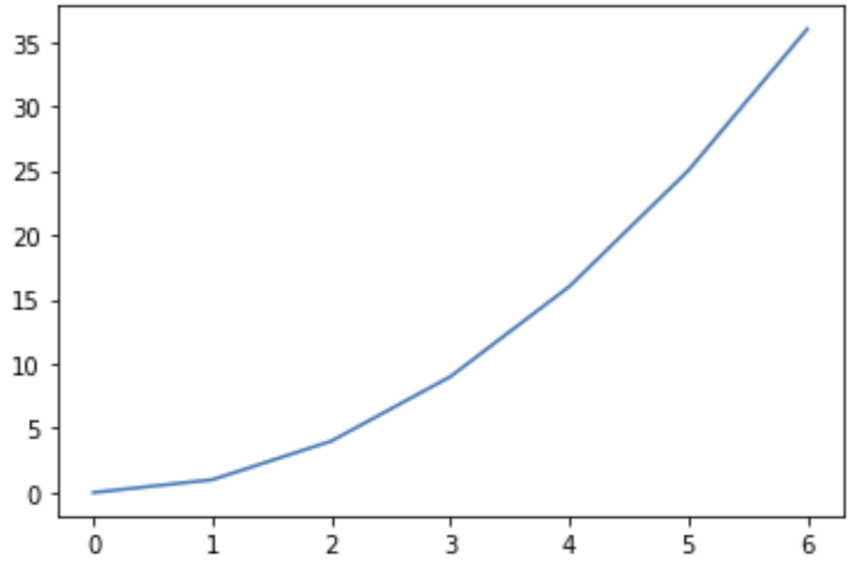
```
import matplotlib.pyplot as plt
```

Line Plots

Northeastern University

- A line plot is created by calling **plt.plot**(x, y, [fmt])
  - x and y are 1D arrays of the same length (can be lists, NumPy arrays, etc.)
  - fmt is an optional string that can be used to change properties of the graph (color, labels, etc.)

```
x = [0, 1, 2, 3, 4, 5, 6]
y = [num**2 for num in x]

plt.plot(x, y);
```



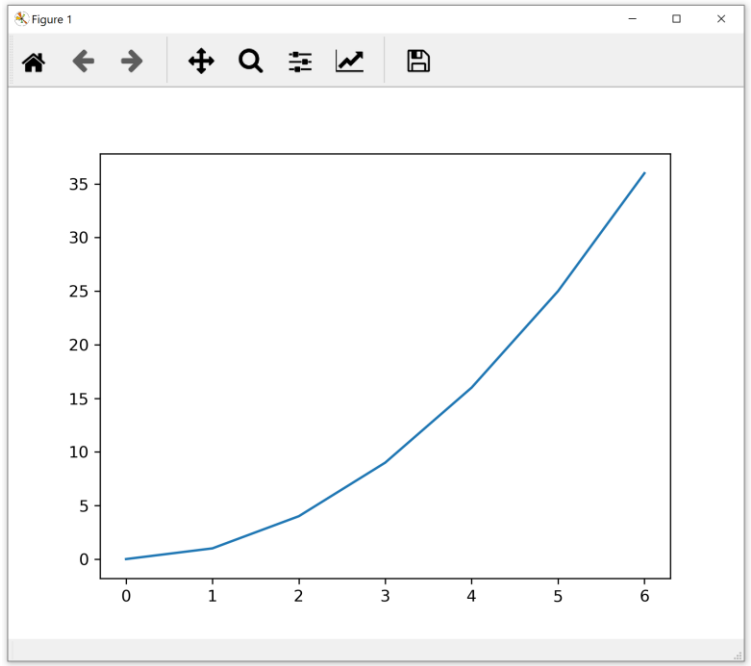
The ; at the end of the cell suppresses the output of the cell (otherwise the plot object itself would also be printed)

Plotting from a Script

Northeastern University

- To display a plot from within a script you need to call **plt.show()**
  - This will open a separate window with your figure displayed
  - The script is blocked until the figure's window is closed
  - Typically called at the last line of the script

```
import matplotlib.pyplot as plt

x = [0, 1, 2, 3, 4, 5, 6]
y = [num**2 for num in x]

plt.plot(x, y)
plt.show()
```

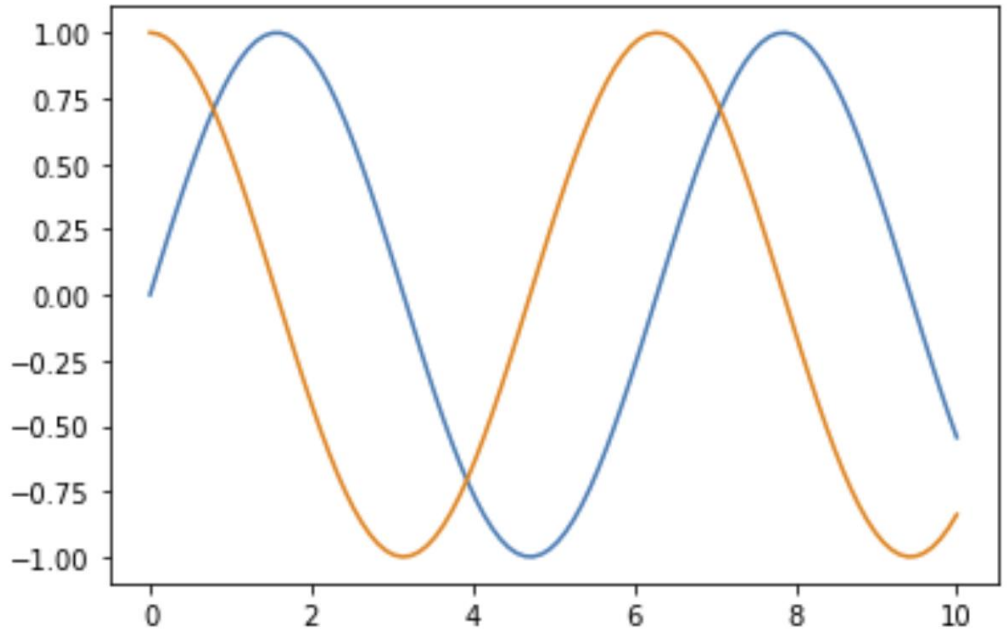


Displaying Multiple Plots

Northeastern University

- To create a single figure with multiple lines, simply call plot() multiple times:

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

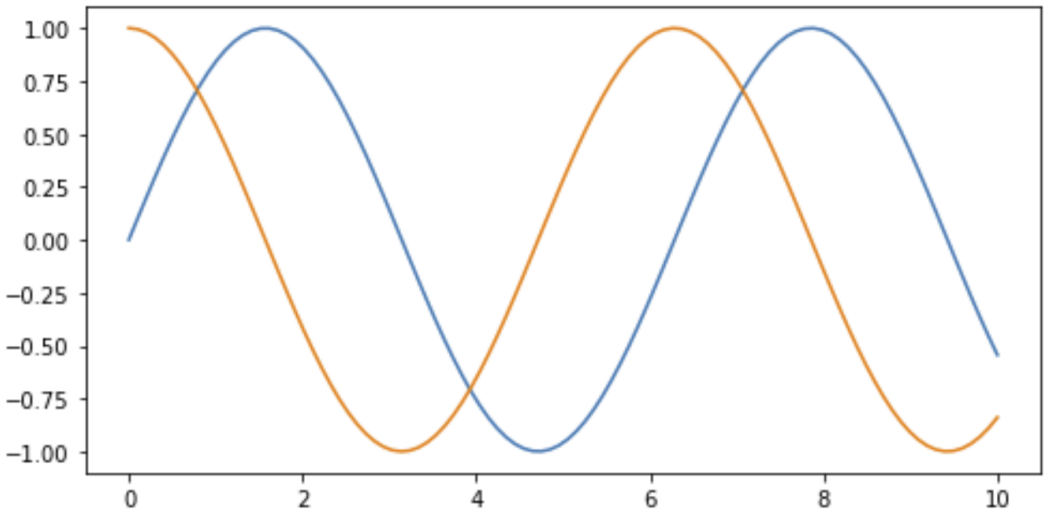


Changing the Figure Size

Northeastern University

- You can change the figure size by calling the function **plt.figure**(*figsize*)
  - *figsize* is a tuple of (width, height) in inches (default is (6.4, 4.8))

```
x = np.linspace(0, 10, 100)

plt.figure(figsize=(8, 4))
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x));
```

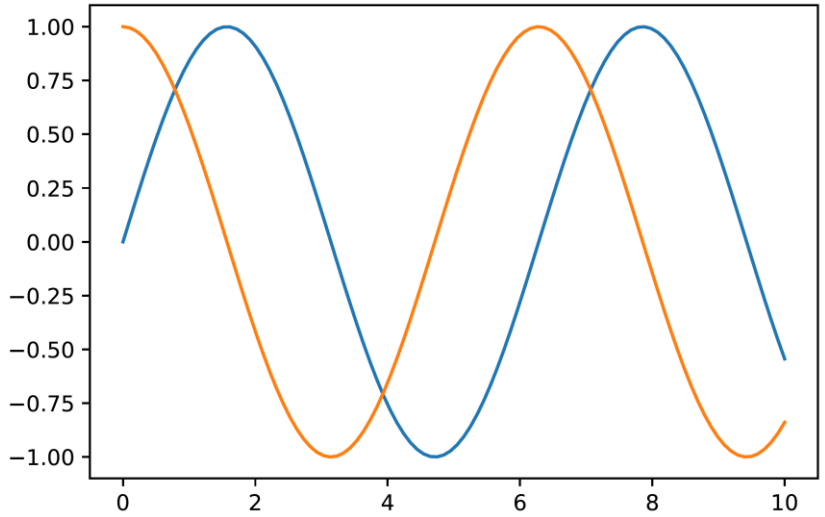


Saving a Figure

Northeastern University

- You can save the current figure to a file by calling the function **plt.savefig()**
- Matplotlib supports various file formats such as png, pdf, svg

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))

plt.savefig('line_plot.pdf')
```

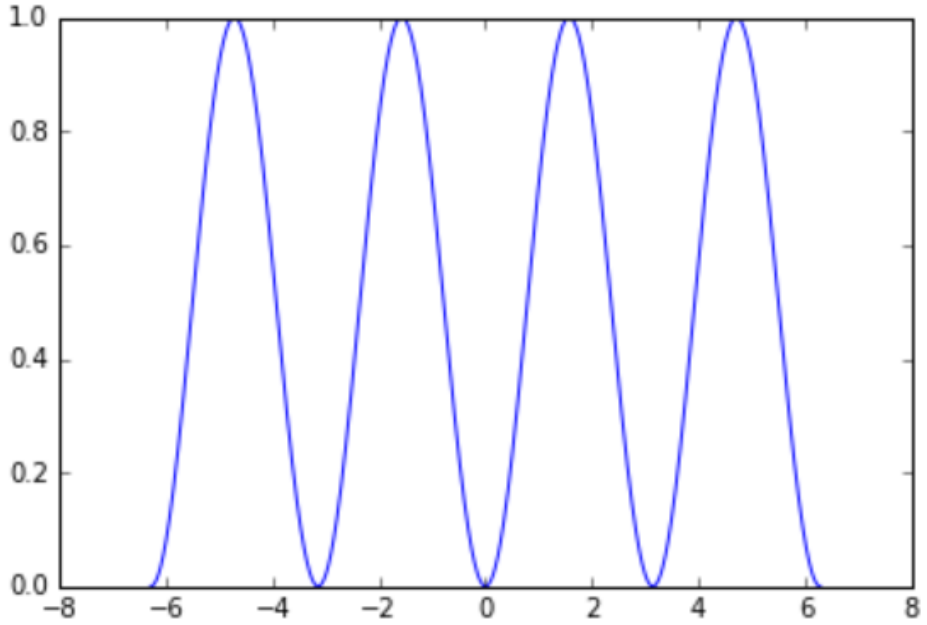


The plot in the PDF file

Class Exercise

Northeastern University

- Plot the function $f(x) = \sin^2 x$ for 1,000 points across the range $-2\pi \leq x \leq 2\pi$



Two Matplotlib Interfaces

Northeastern University

There are two primary ways to work with matplotlib:

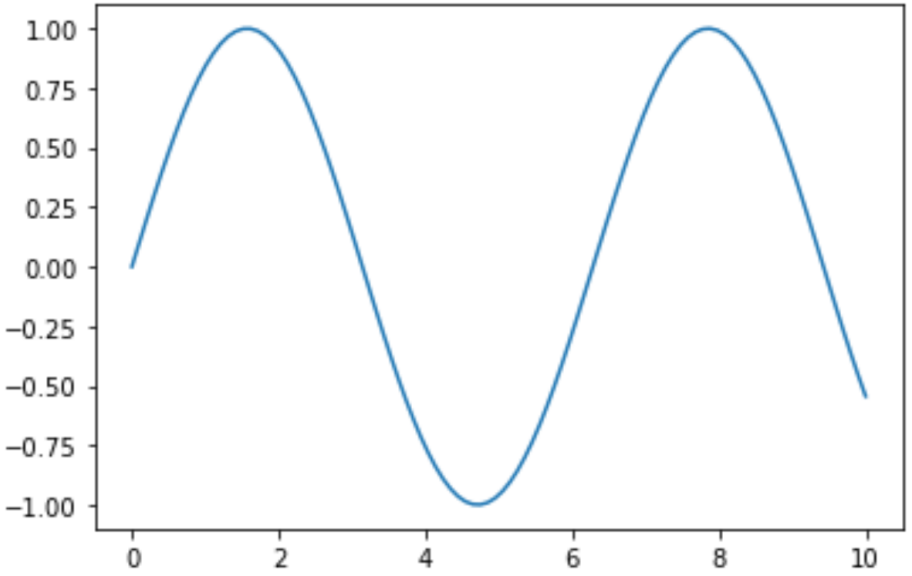
- A convenient procedural, MATLAB-style interface using **matplotlib.pyplot**
  - There is only one active figure and one axes at each time point
  - Each pyplot function makes some change to that figure, e.g., plot some lines
  - The state of the figure is preserved across function calls
- A more powerful object-oriented interface
  - Allows more advanced and customizable use
  - The plotting functions are **methods** of the Figure and Axes objects

Object-Oriented Interface

Northeastern University

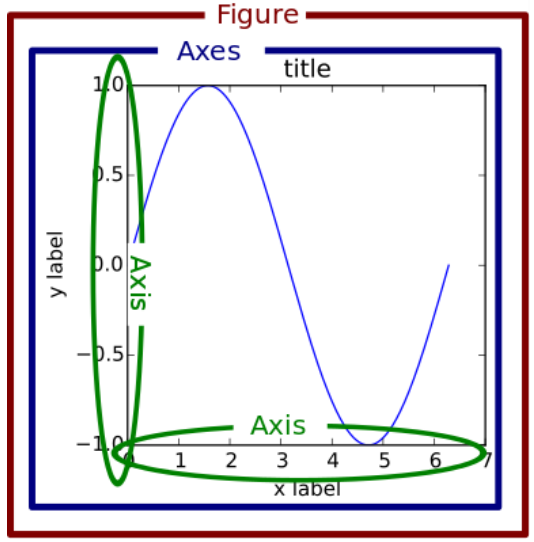
- The object-oriented interface gives you more control over your figure

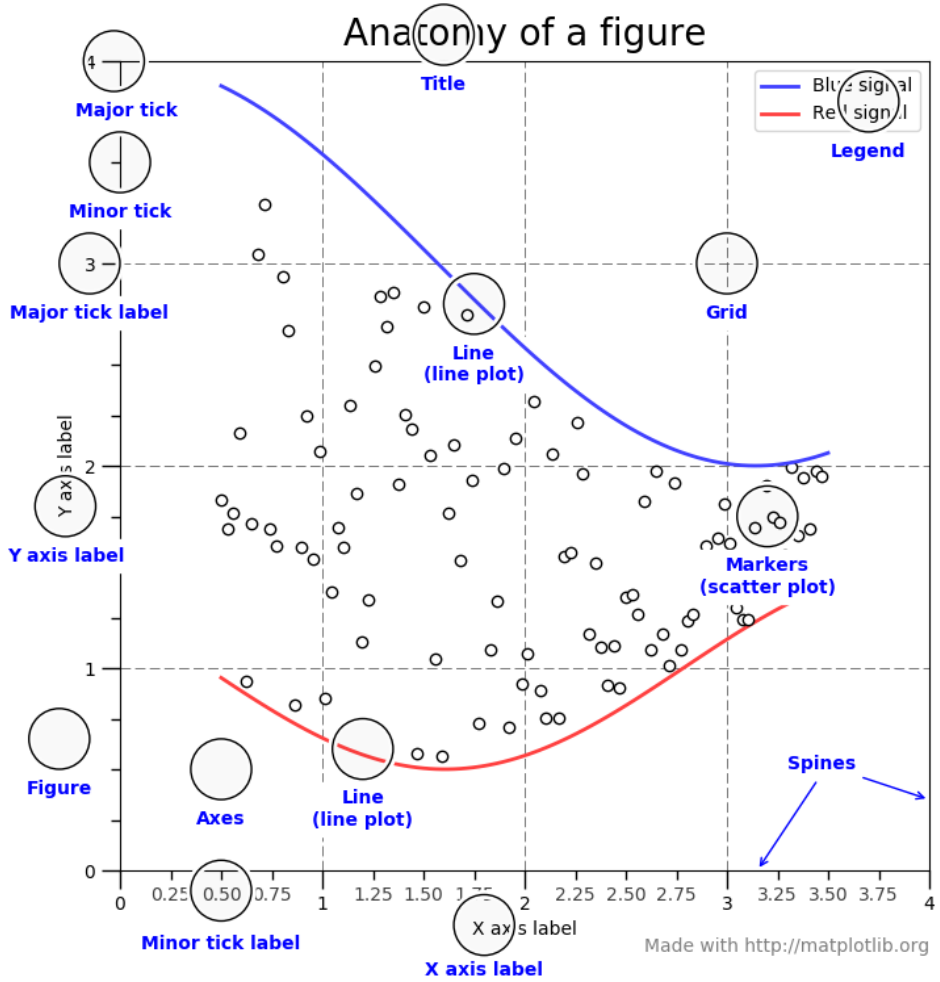
```
# Create a figure and a set of subplots
fig, ax = plt.subplots(1)

x = np.linspace(0, 10, 100)

# Call plot() method on the axes object
ax.plot(x, np.sin(x));
```



In most cases, the difference is as small as switching from plt.plot() to ax.plot()

The Anatomy of a Figure



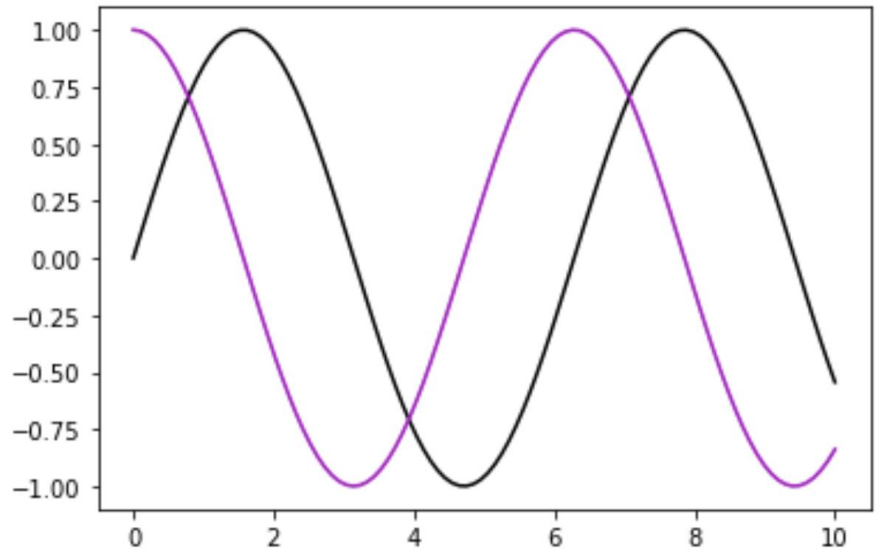


Changing the Line Color

Northeastern University

- You can specify the line's color by passing the argument **color** or **c** to plt.plot()
- Matplotlib supports different color formats:
  - one of the characters {'b', 'g', 'r', 'c', 'm', 'y', 'k', 'w'}, which are short-hand notations for blue, green, red, cyan, magenta, yellow, black, and white
  - HTML color names, e.g., 'chatreuse'
  - RGB tuple of float values, e.g., (0.1, 0.2, 0.5)

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), color='k')
plt.plot(x, np.cos(x), color='m');
```

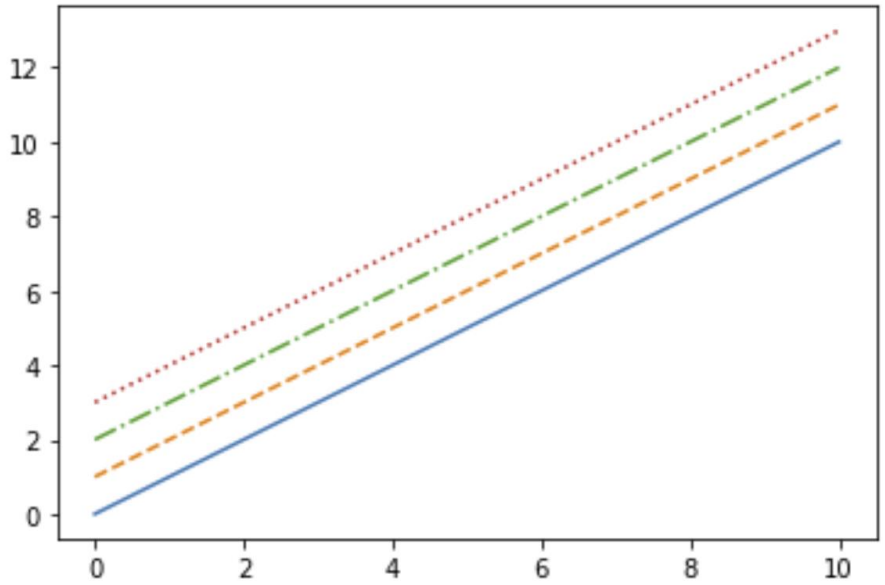


Changing the Line Style

Northeastern University

- You can specify the line's style by passing the argument **linestyle** or **ls** to plt.plot()
- Matplotlib supports four line styles: - solid, -- dashed, -. dashdot, : dotted

```
x = np.array([0, 10])
plt.plot(x, x, linestyle='-')
plt.plot(x, x + 1, linestyle='--')
plt.plot(x, x + 2, linestyle='-.')
plt.plot(x, x + 3, linestyle=':');
```

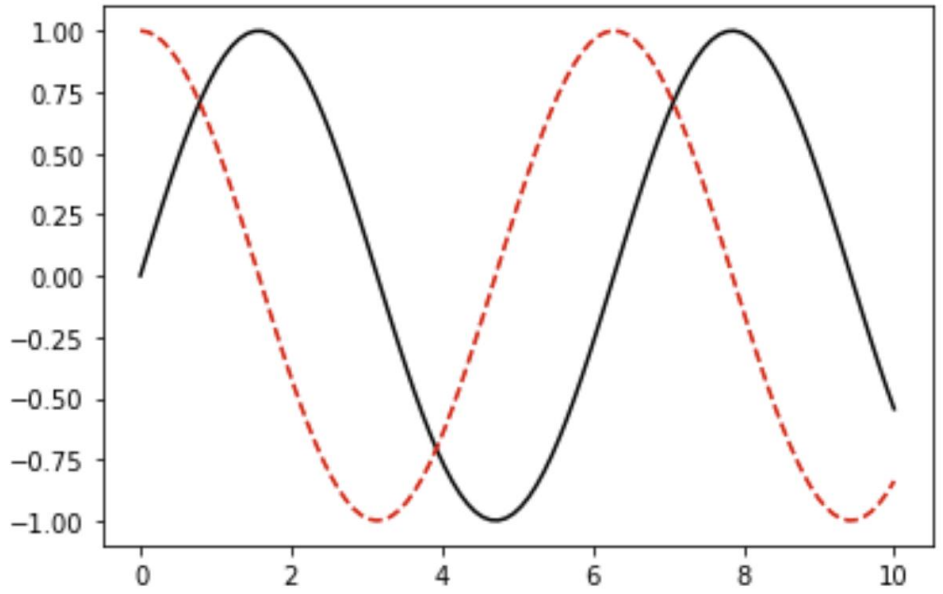


Combining Line Color and Style

Northeastern University

- You can combine linestyle and color codes together and pass them as the third argument of plt.plot():

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), 'k')     # solid black
plt.plot(x, np.cos(x), '--r');  # dashed red
```

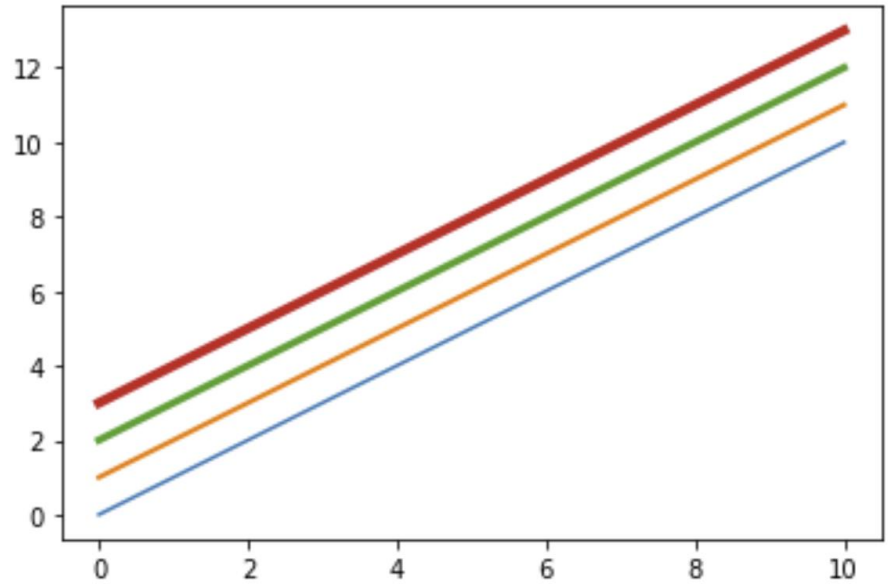


Changing Line Width

Northeastern University

- You can specify the line's width in pixels by passing the argument **linewidth** or **lw**
  - The default width is 1 pixel

```
x = np.array([0, 10])
plt.plot(x, x)
plt.plot(x, x + 1, lw=2)
plt.plot(x, x + 2, lw=3)
plt.plot(x, x + 3, lw=4);
```



Markers

Northeastern University

- You can display markers on the points of the graph by passing the type of symbol to use as the third argument of plt.plot()
- Common markers:

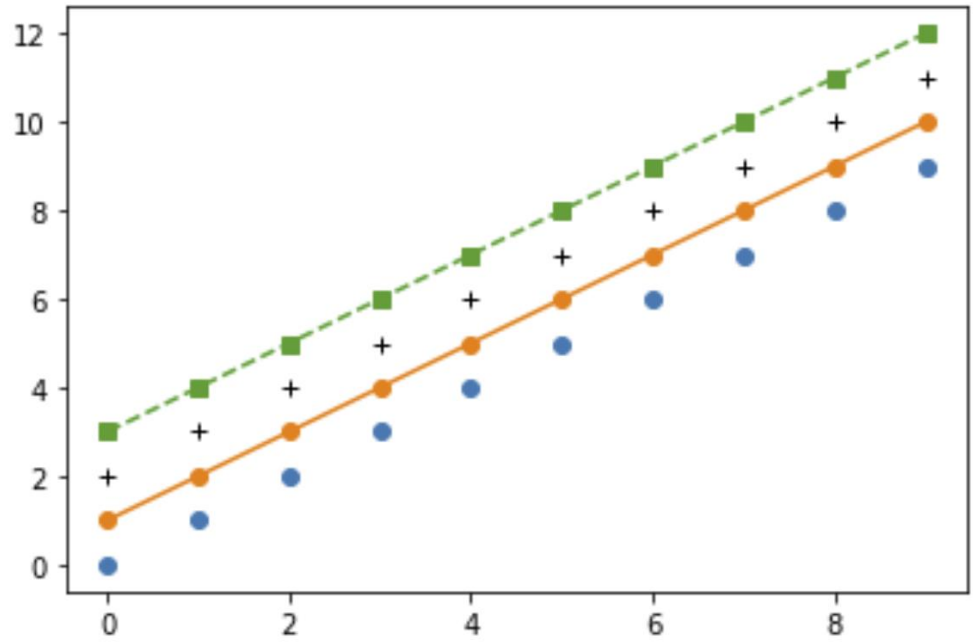
Code	Marker	Description
.	·	point
o	○	circle
+	+	plus
x	×	x
D	◇	diamond
v	▽	downward triangle
^	△	upward triangle
s	□	square
*	★	star

- If you don’t specify a line style, no line will be drawn (just the markers)
- Full list of markers: https://matplotlib.org/3.3.0/api/markers_api.html

Markers

Northeastern University

```
x = np.arange(0, 10, 1)
plt.plot(x, x, 'o')         # circles
plt.plot(x, x + 1, '-o')    # solid line + circles
plt.plot(x, x + 2, 'k+')    # black +
plt.plot(x, x + 3, '--s'); # dashed line + squares
```

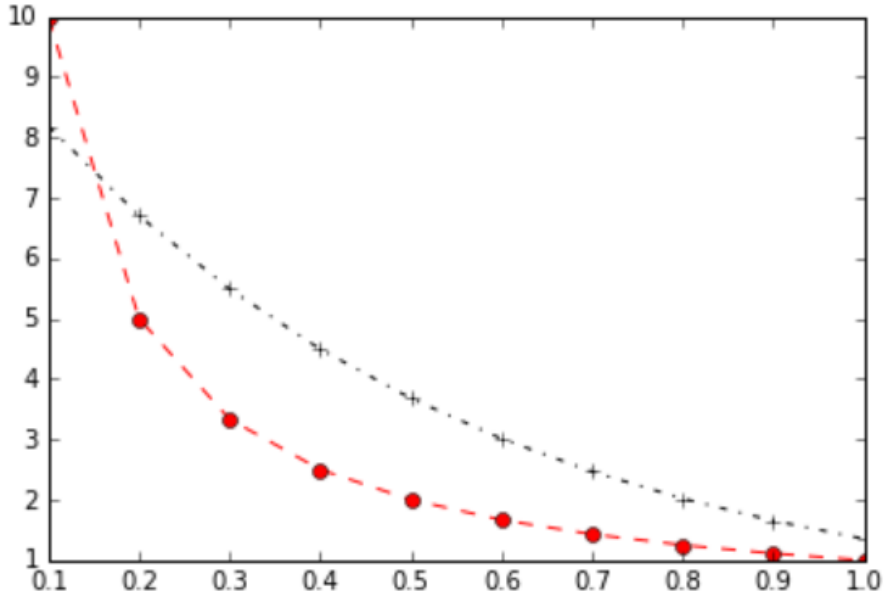


Markers

Northeastern University

- Multiple lines can be plotted using a sequence of x, y, format arguments:

```
x = np.linspace(0.1, 1, 10)
y1 = 1 / x
y2 = 10 * np.exp(-2 * x)

plt.plot(x, y1, 'r--o', x, y2, 'k-.+');
```

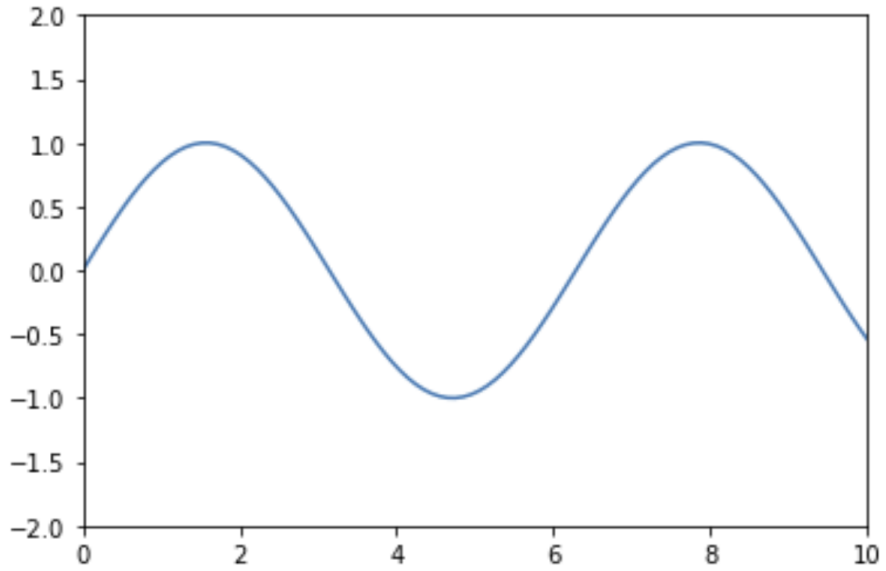


Axes Limits

Northeastern University

- Matplotlib does a decent job of choosing default axes limits for your plot
- However, you can adjust axis limits by using the **plt.xlim()** and **plt.ylim()** methods:

```
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))

plt.xlim(0, 10)
plt.ylim(-2, 2);
```

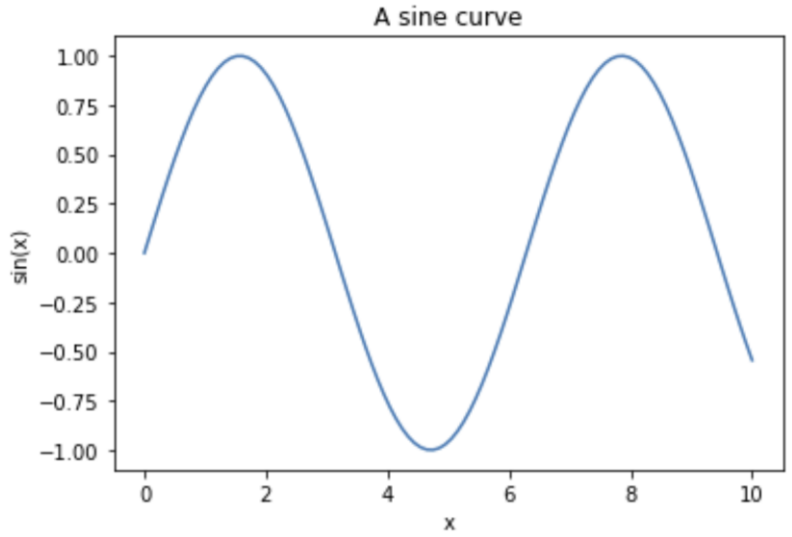


Labeling Plots

Northeastern University

- A plot can be given a title above the axes by passing a string to the **title()** function
- Similarly, the methods **xlabel()** and **ylabel()** control the labeling of the *x*- and *y*-axes
- The position, size, and style of these labels can be adjusted using optional arguments

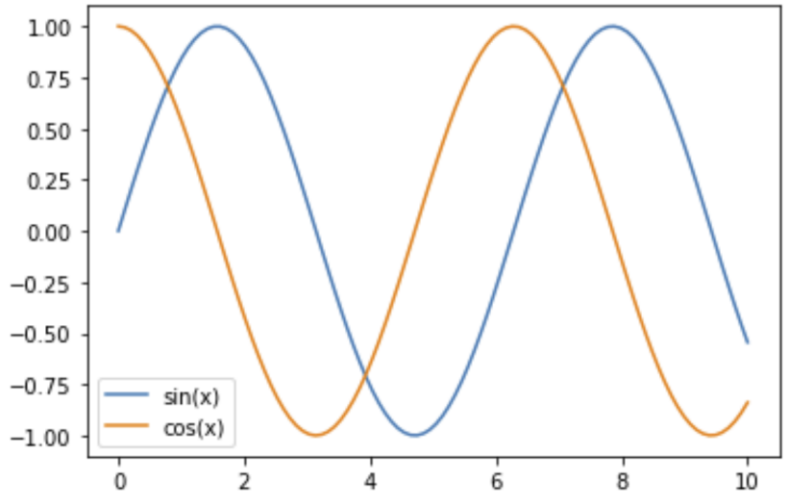
```
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))

plt.title('A sine curve')
plt.xlabel('x')
plt.ylabel('sin(x)');
```



Legend

- When showing multiple lines, it is useful to create a legend with labels for each line
- You provide a label for each line by passing the **label** argument to plt.plot()
- The labels appear on the plot when you call the function **plt.legend()**

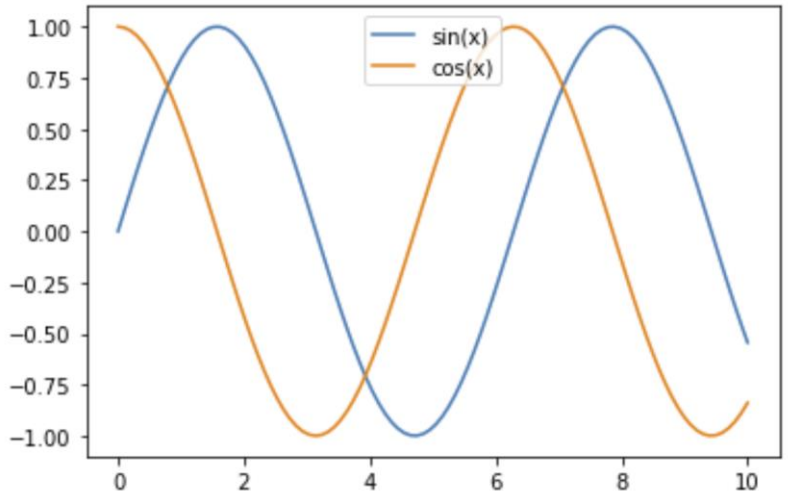
```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')

plt.legend();
```



Legend

- You can control the location of the legend using the argument loc
- The location options are listed in https://matplotlib.org/3.2.2/api/_as_gen/matplotlib.pyplot.legend.html

```
x = np.linspace(0, 10, 100)

plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')

plt.legend(loc='upper center');
```



Font Properties

Northeastern University

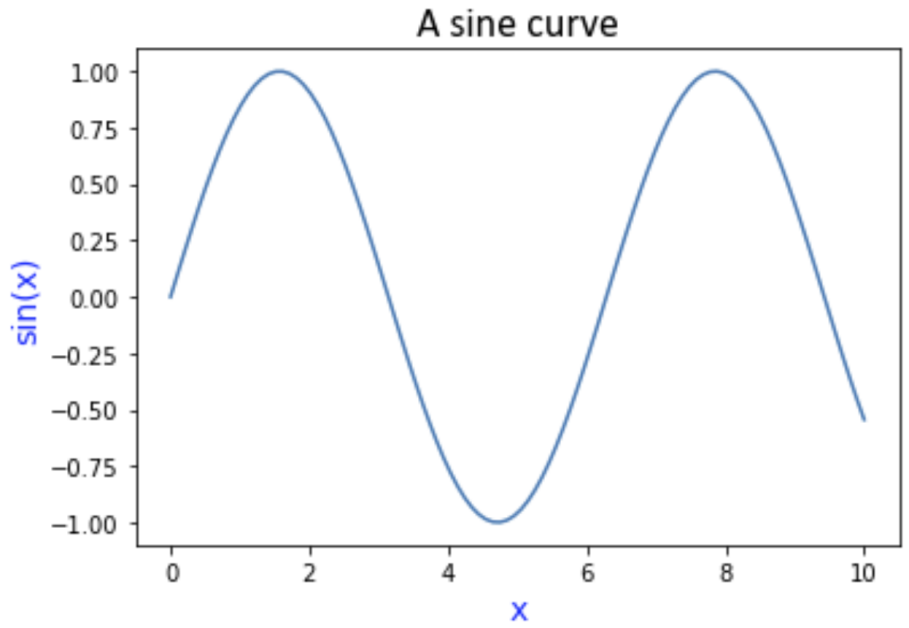
- The font of the text elements of a plot (titles, legend, axis labels, etc.) can be customized with the arguments given in the following table:

Argument	Description
fontsize	The size of the font in points (e.g., 12)
fontname	The font name (e.g., 'Courier', 'Arial')
family	The font family (e.g., 'sans-serif', 'cursive', 'monospace')
fontweight	The font weight (e.g., 'normal', 'bold')
fontstyle	The font style (e.g., 'normal', 'italic')
color	Any Matplotlib color specifier (e.g., 'r', '#ff00ff')

Font Properties

Northeastern University

```
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))

plt.title('A sine curve', fontsize=18, fontname='Calibri')
plt.xlabel('x', fontsize=14, color='b')
plt.ylabel('sin(x)', fontsize=14, color='b');
```

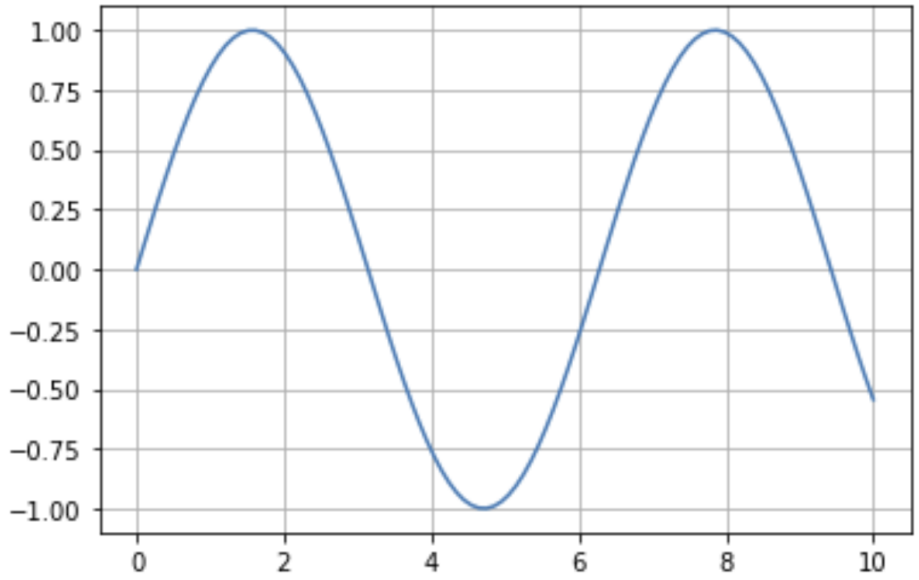


Gridlines

Northeastern University

- You can show gridlines across the plot to aid with locating numerical values of data points by calling **plt.grid()**

```
x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))

plt.grid();
```

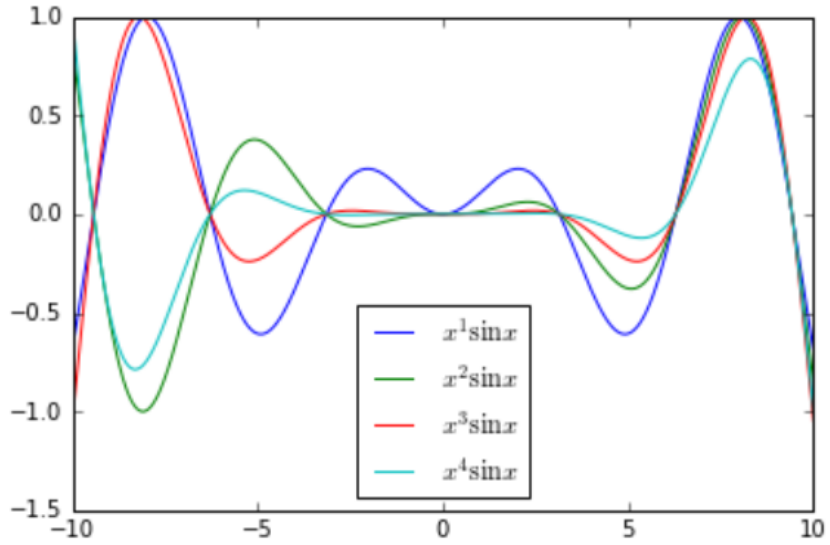


Using LaTeX

Northeastern University

- You can use LaTeX markup in pyplot
  - Use raw strings (r'xxx') to avoid escaping any characters followed by LaTeX backslashes

```
x = np.linspace(-10, 10, 1000)
for n in range(1, 5):
    y = x**n * np.sin(x)
    y /= max(y)  # scale to a maximum of 1
    plt.plot(x, y, label=fr'$x^{n}\sin x$')
plt.legend(loc='lower center');
```

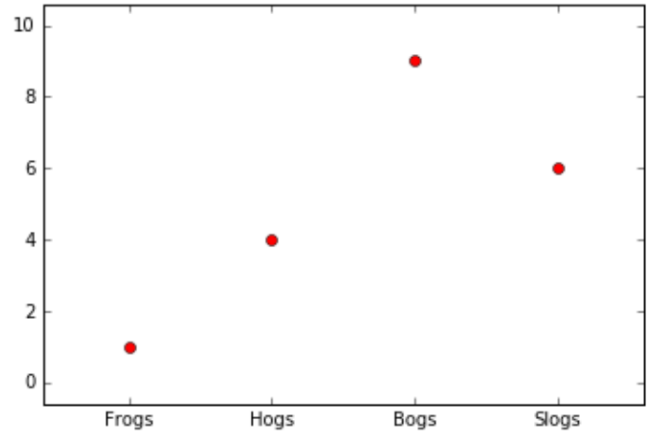


Customizing the Ticks

Northeastern University

- Matplotlib does its best to label representative values (*tick marks)* on each axis
- However, sometimes you might want to customize their frequency or labels
- To that end, you can use the methods **plt.xticks**(locs, labels) or **plt.yticks**(locs, labels)
  - These methods get a list of positions for the tick labels, and a list of the label strings (optional)
  - To remove the tick marks altogether, set them to an empty list, e.g., xticks([])

```
x = [1, 2, 3, 4]
y = [1, 4, 9, 6]
labels = ['Frogs', 'Hogs', 'Bogs', 'Slogs']

plt.plot(x, y, 'ro')
plt.xticks(x, labels)

# Pad margins so that markers don't get clipped by the axes
plt.margins(0.2);
```

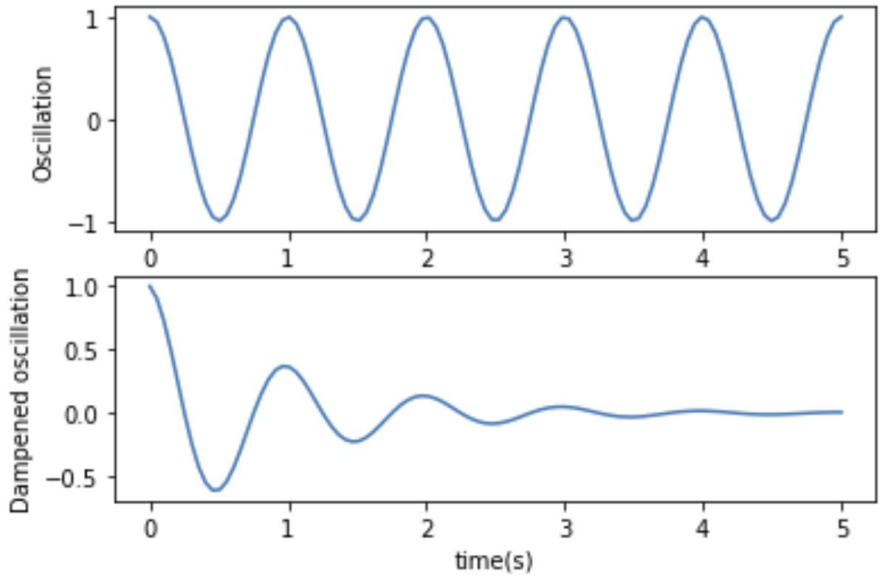


Multiple Subplots

Northeastern University

- You can define multiple subplots in the same figure
- **plt.subplot(numrows, numscols, fignum)** adds a new subplot to the current figure
  - the arguments are the number of rows, number of columns and the subplot number
  - fignum ranges from 1 to numrows * numcols
  - The commas between the arguments are optional if numrows * numcols < 10

```
x = np.linspace(0, 5, 100)
y1 = np.cos(2 * np.pi * x)
y2 = np.cos(2 * np.pi * x) * np.exp(-x)

plt.subplot(211)  # 2 rows, 1 column, fignum #1
plt.plot(x, y1)
plt.ylabel('Oscillation')

plt.subplot(212)  # 2 rows, 1 column, fignum #2
plt.plot(x, y2)
plt.xlabel('time(s)')
plt.ylabel('Dampened oscillation');
```

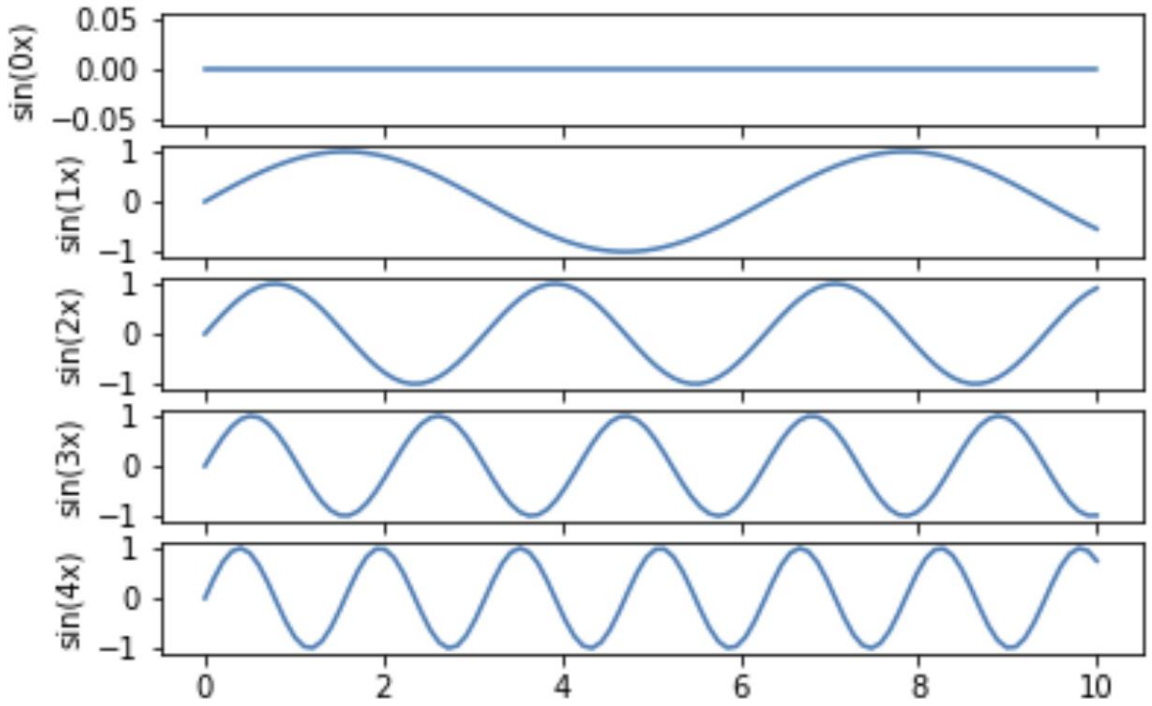


Multiple Subplots

Northeastern University

- As an alternative method, **plt.subplots(numrows, numscols)** (note the s at the end) creates a grid of subplots and returns their axes
- The axes objects can be used to perform an operation on all the subplots

```
x = np.linspace(0, 10, 100)

fig, axes = plt.subplots(5, 1)

for i in range(5):
    axes[i].plot(x, np.sin(i * x))
    axes[i].set_ylabel(f'sin({i}x)')
```

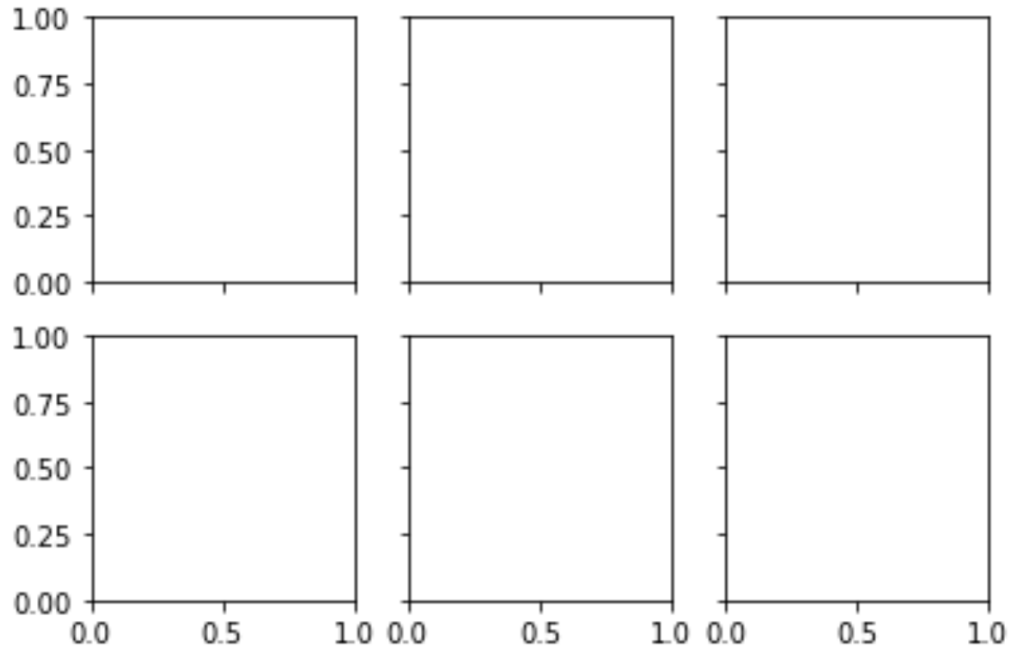


Multiple Subplots

Northeastern University

- You can use **sharex** and **sharey** parameters to share the tick labels among the subplots

```
fig, axes = plt.subplots(2, 3, sharex='col', sharey='row');
```

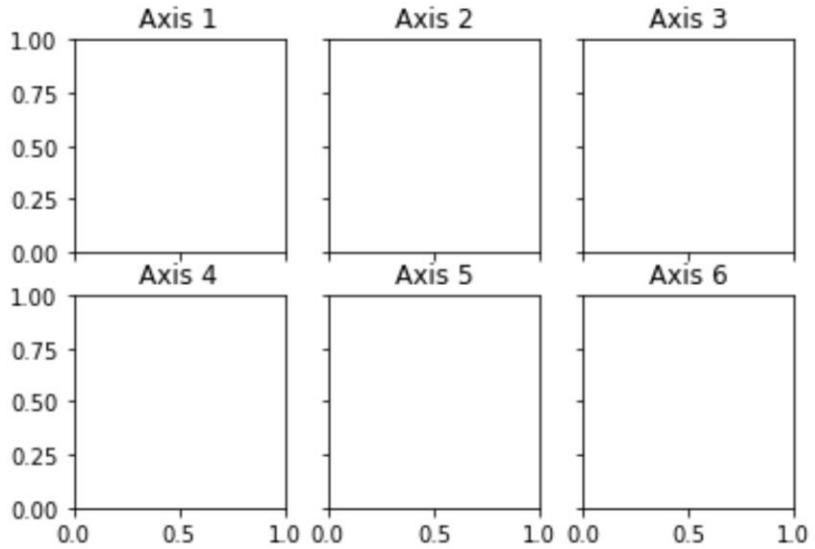


Multiple Subplots

Northeastern University

- When stacking in two directions, the returned axes is a 2D NumPy array
- You can iterate over all the subplots in the 2D grid by using **axes.flat**:

```
fig, axes = plt.subplots(2, 3, sharex='col', sharey='row')

i = 1

for ax in axes.flat:
    ax.set_title('Axis ' + str(i))
    i += 1
```

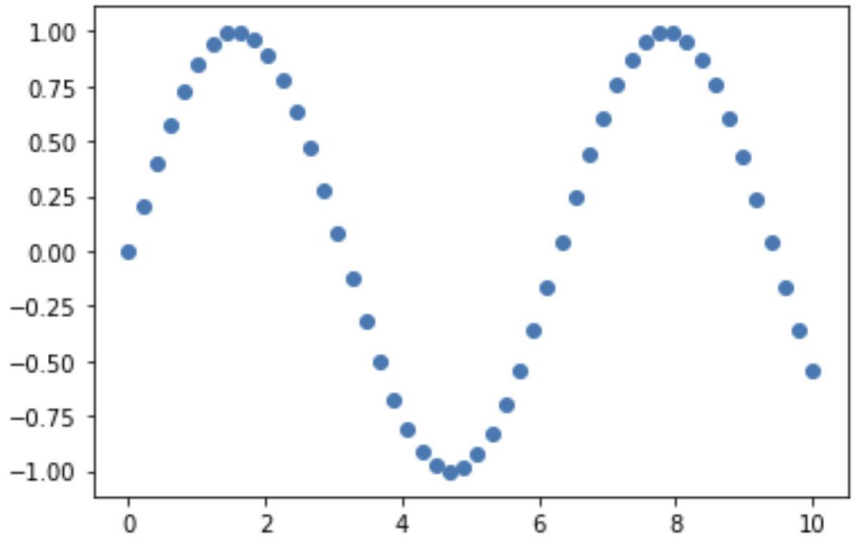


Scatter Plots

Northeastern University

- A scatter plot draws points without lines connecting them
- Created by calling **plt.scatter()**
- A scatter plot allows you to specify a different color and size for each point individually, which is not possible to do in a line plot

```
x = np.linspace(0, 10, 50)
plt.scatter(x, np.sin(x));
```

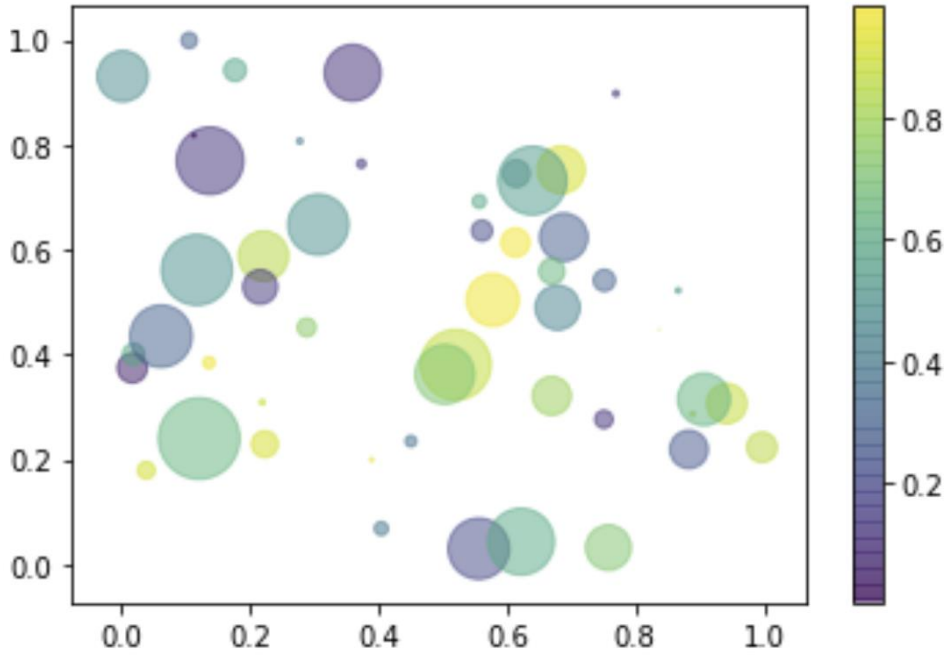


Scatter Plots with plt.scatter()

Northeastern University

- The colors and sizes of the points can be specified using the **c** and **s** arguments
- The sizes are specified in points**2 (1 pt = 1/72 inch ~ 0.035 cm)
- A scatter plot of 50 random points with random colors and sizes:

```
n = 50  # number of points
x = np.random.rand(n)
y = np.random.rand(n)
sizes = (30 * np.random.rand(n)) ** 2
colors = np.random.rand(n)

plt.scatter(x, y, s=sizes, c=colors, alpha=0.5)
plt.colorbar();   # show the color scale
```



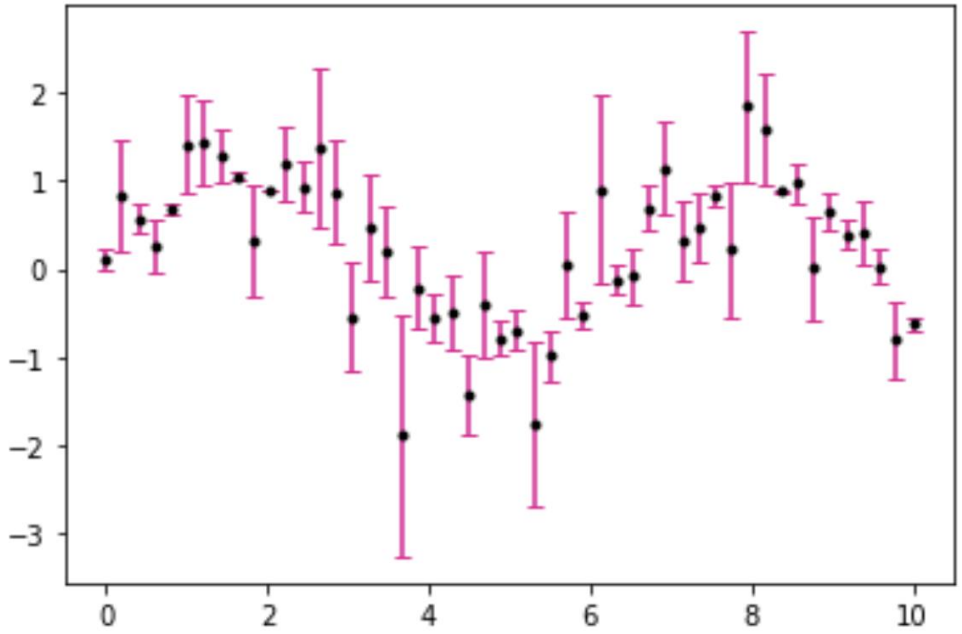
Error Bars

- **plt.errorbar()** generates a plotted line with error bars
- The errors in the x and y coordinates are specified by the arguments **xerr** and **yerr**

```
n = 50
x = np.linspace(0, 10, n)
y = np.sin(x)
dy = 0.5 * np.random.randn(n)

plt.errorbar(x, y + dy, yerr=dy, fmt='.k',
             ecolor='deeppink', capsize=3);
```

Color of the error bars

Style of the line plot

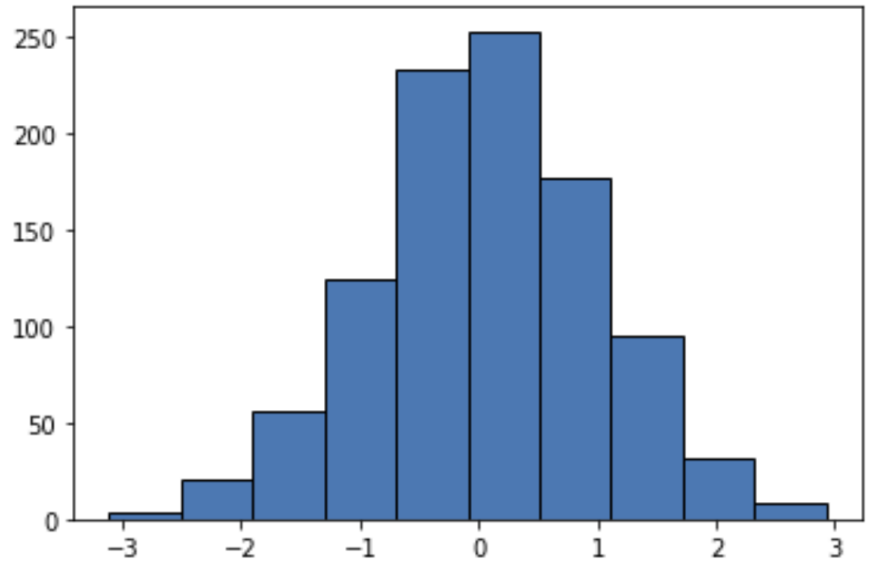


Histograms

Northeastern University

- A histogram displays the distribution of a continuous numerical variable
- **plt.hist**(*x*, *bins=None*) plots a histogram of the values in the array *x*
  - *bins* defines the number of equal-width bins in the range (default is 10)
  - Specify an *edgecolor* if you want to see outlines around each bar

```
x = np.random.randn(1000)
plt.hist(x, edgecolor='k');
```

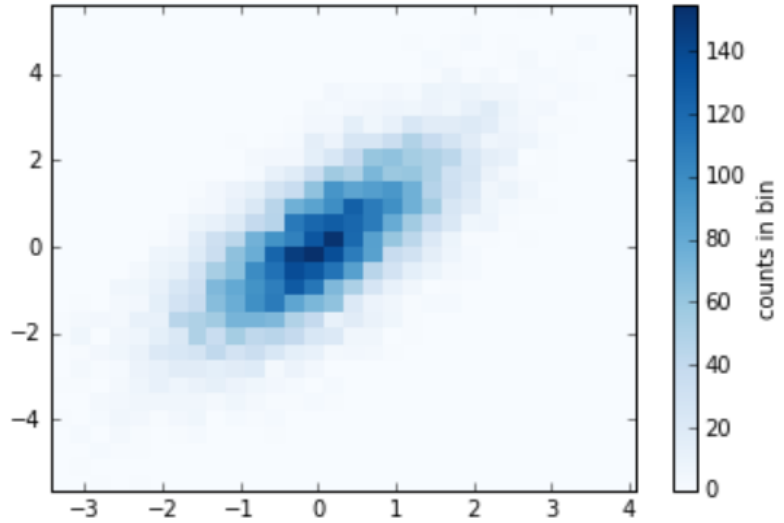


Two-Dimensional Histograms

- To plot a 2D histogram use the **plt.hist2d()** function
- This is useful for displaying multivariate distribution such as a multivariate Guassian

```
mean = [0, 0]
cov = [[1, 1], [1, 2]]
x, y = np.random.multivariate_normal(mean, cov, 10000).T
```

```
plt.hist2d(x, y, bins=30, cmap='Blues')
plt.colorbar(label='counts in bin');
```

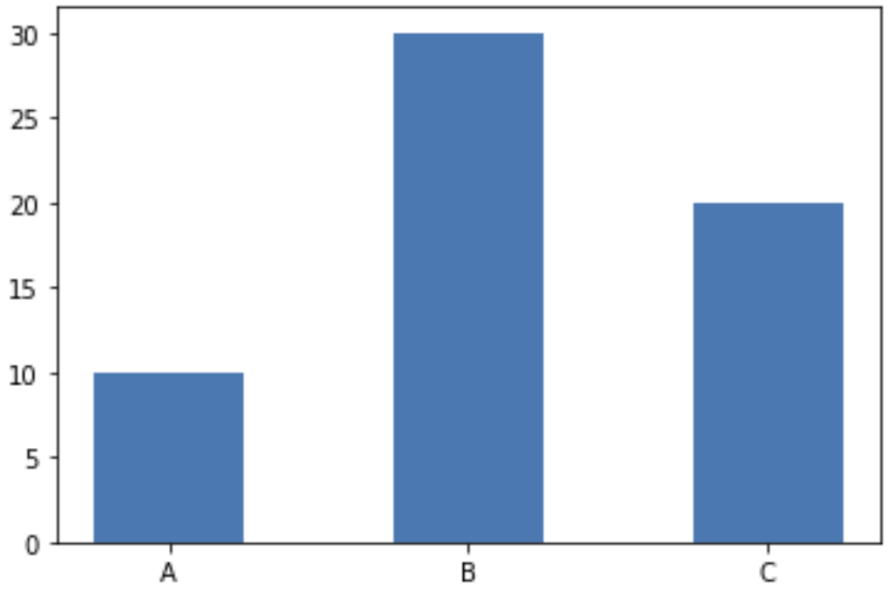


Bar Charts

Northeastern University

- Bar charts are used to compare categories of data
- **plt.bar**(*x*, *height*, *width=0.8*) creates a bar plot
  - *x* defines the x coordinates of the bars
  - *height* defines the heights of the bars
  - *width* defines the widths of the bars (default: 0.8)

```
labels = ['A', 'B', 'C']
data = [10, 30, 20]

x = np.arange(len(labels))  # the label locations
plt.bar(x, data, width=0.5)

plt.xticks(ticks=x, labels=labels);
```

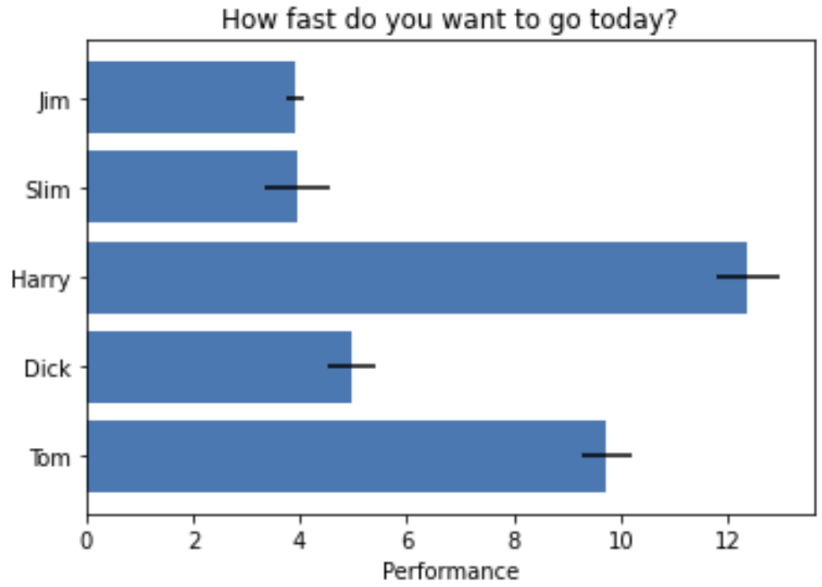


Horizontal Bar Charts

Northeastern University

- **plt.barh**(*y*, *width*, *height=0.8*) creates a horizontal bar plot
  - *y* defines the y coordinates of the bars
  - *width* defines the width of the bars
  - *height* defines the height of the bars (default: 0.8)

```
people = ('Tom', 'Dick', 'Harry', 'Slim', 'Jim')
y_pos = np.arange(len(people))
performance = 3 + 10 * np.random.rand(len(people))
error = np.random.rand(len(people))

plt.barh(y_pos, performance, xerr=error)
plt.yticks(y_pos, people)
plt.xlabel('Performance')
plt.title('How fast do you want to go today?');
```

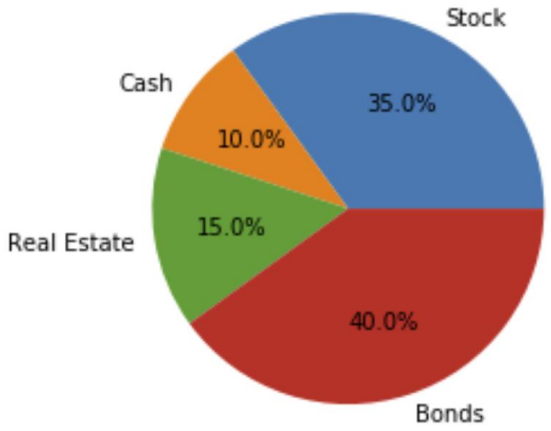


Pie Charts

Northeastern University

- **plt.pie**(*x*, *labels=None, autopct=None*) creates a pie chart of the values in *x*
  - The values in *x* are normalized by their sum
  - *labels* provide the labels for each wedge
  - *autopct* is a string used the label the wedges of the form *fmt%pct*

```
labels = ['Stock', 'Cash', 'Real Estate', 'Bonds']
x = [35, 10, 15, 40]

plt.pie(x, labels=labels, autopct='%.1f%%');
```

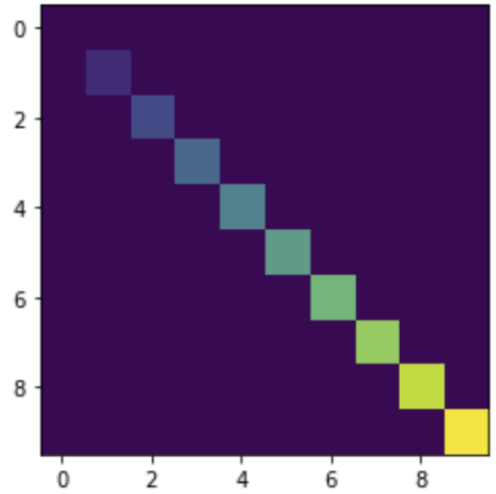


Heatmaps

Northeastern University

- A heatmap shows a magnitude of a phenomenon using colors in 2D
  - Each pixel in the map is determined by the corresponding value in the data
- **plt.imshow**(*x*, *cmap=None*) creates a heatmap of the data in *x*
  - If x has a shape (M, N), the data is visualized using a color map
  - If x has a shape (M, N, 3), then x is treated as an image with RGB values (0-1 float or 0-255 int)

```
a = np.diag(np.arange(10))
plt.imshow(a);
```

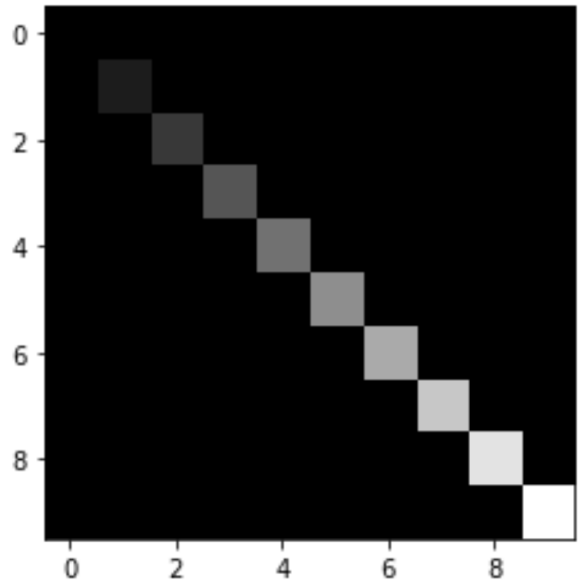


Color Maps

Northeastern University

- A colormap can be specified using a string or a ColorMap object from plt.cm

```
a = np.diag(np.arange(10))
plt.imshow(a, cmap='gray');
```



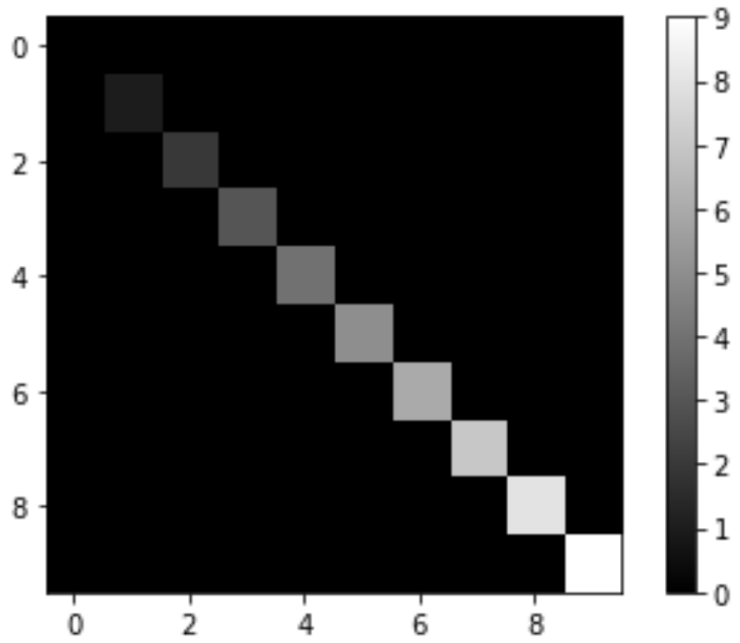
- Matplotlib has a wide range of colormaps available
  - A full list of color maps: https://matplotlib.org/3.1.1/gallery/color/colormap_reference.html

Color Bars

Northeastern University

- A colorbar provides a key for the meaning of colors in a plot
- You can display a colorbar by calling the **plt.colorbar()** function:

```
a = np.diag(np.arange(10))
plt.imshow(a, cmap='gray')
plt.colorbar();
```



Heatmaps

Northeastern University

- Heatmaps can also be used to display 3D data
- For example, consider the following function z = f(x, y):

```
def f(x, y):
    return np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

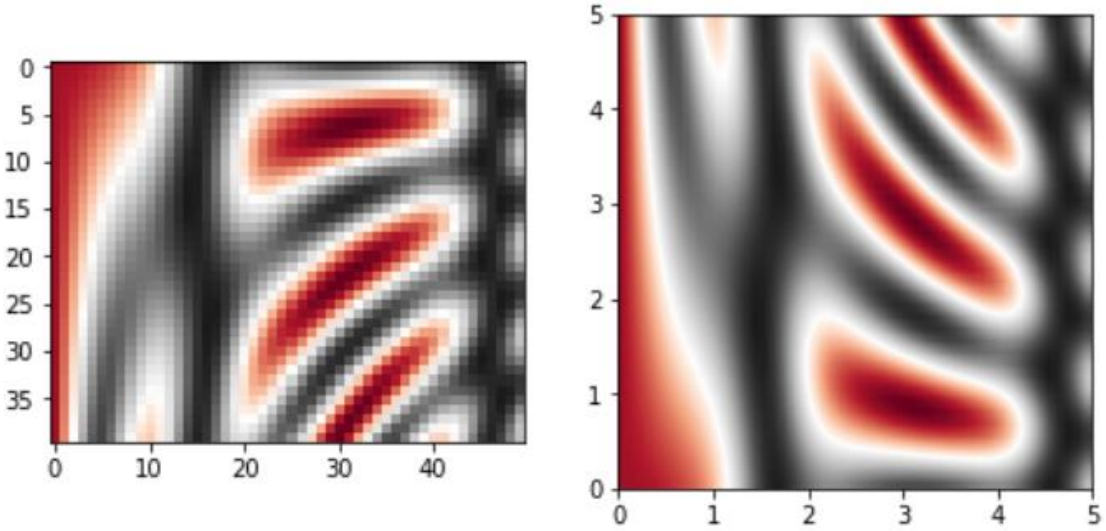
- First, we prepare a grid of x and y values for the plot by using **np.meshgrid()**, which builds 2D grids from one-dimensional arrays:

```
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 40)

X, Y = np.meshgrid(x, y)
Z = f(X, Y)
```

Heatmaps

Northeastern University

```
fig, axes = plt.subplots(1, 2, figsize=(8, 6))

axes[0].imshow(Z, cmap='RdGy')
axes[1].imshow(Z, extent=[0, 5, 0, 5], origin='lower',
               cmap='RdGy', interpolation='bilinear');
```



Plotting Images

Northeastern University

- plt.imshow() can also be used to display real images
- The function **image.imread()** can be used to read an image from a file into a 3D array with the shape (M, N, 3)

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
```

```
im = mpimg.imread('bostonfall.jpg')
```

```
plt.imshow(im);
```

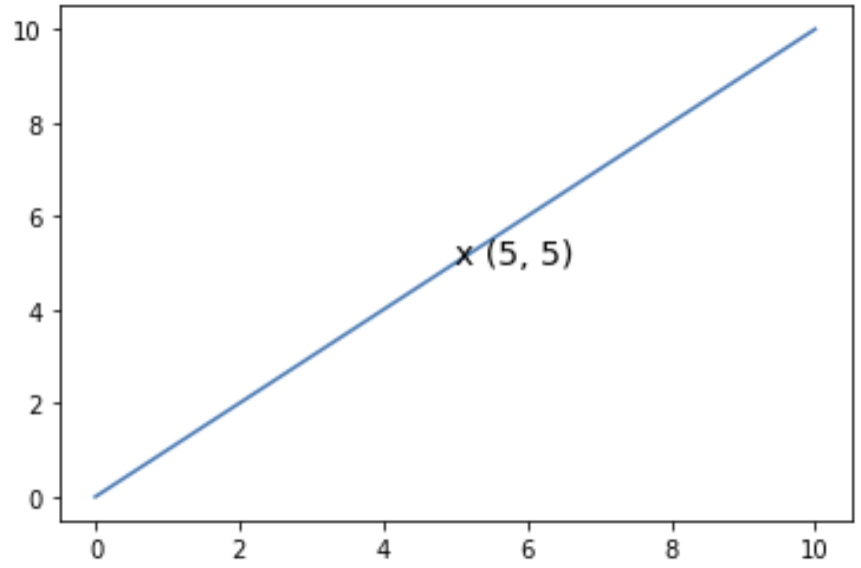
- To remove the axes you can call plt.axis('off')

Text and Annotations

Northeastern University

- You can add different kinds of annotation to your plots
- **plt.text**(*x*, *y*, *s*) adds a text string *s* at position (*x*, *y*) to the figure

```
x = [0, 10]
plt.plot(x, x)

plt.text(5, 5, 'x (5, 5)', size=14);
```



Arrows and Annotation

Northeastern University

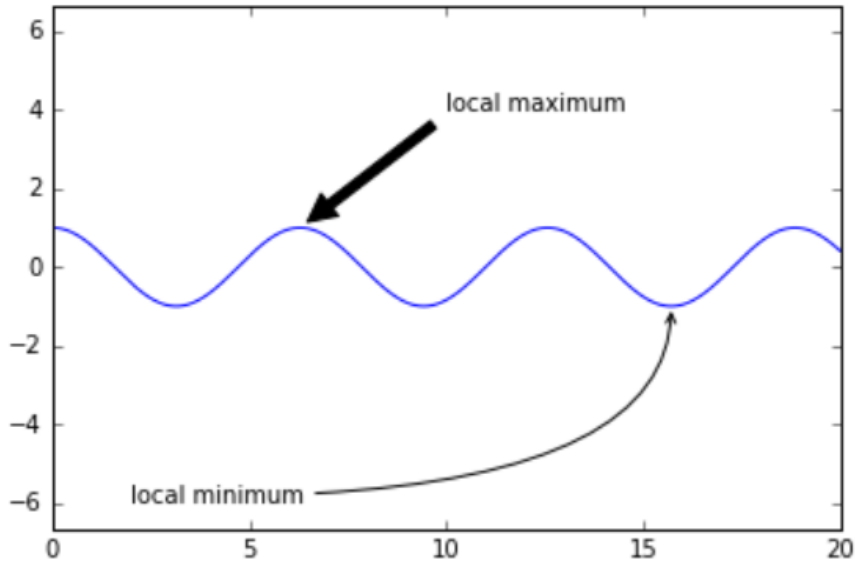
- Along with tick marks and text, another useful annotation mark is the simple arrow
- The **plt.annotate()** method is similar to plt.text(), but draws an arrow from the text to a specified point in the plot. The important arguments to plt.annotate() are:

Argument	Description
s	the string to output as a text label
xy	a tuple, (x,y) giving the coordinates of the position to annotate (i.e., where the arrow points to)
xytext	a tuple, (x,y) giving the coordinates of the text label (i.e., where the arrow points from)
xycoords	an optional string determining the type of coordinates referred to by the argument xy: – 'data': data coordinates, the default – 'axes fraction': fractional coordinates of the axes – 'figure fraction': fractional coordinates of the figure size
textcoords	as for xycoords, an optional string determining the type of coordinates referred to by xytext. An additional value is permitted for this string: 'offset points' specifies that the tuple xytext is an offset in points from the xy position.
arrowprops	if present, determines the properties and style of the arrow drawn between xytext and xy

Arrows and Annotation

Northeastern University

```
x = np.linspace(0, 20, 1000)
plt.plot(x, np.cos(x))
plt.axis('equal')

plt.annotate('local maximum', xy=(2 * np.pi, 1), xytext=(10, 4),
             arrowprops=dict(facecolor='black', shrink=0.05))
plt.annotate('local minimum', xy=(5 * np.pi, -1), xytext=(2, -6),
             arrowprops=dict(arrowstyle='->',
                             connectionstyle='angle3,angleA=0,angleB=-90'));
```



3D Plotting

Northeastern University

- Matplotlib was initially designed with only two-dimensional plotting in mind
- Around the time of the 1.0 release, some 3D plotting utilities were built on top of Matplotlib's 2D display
- Three-dimensional plots are enabled by importing the mplot3dtoolkit, included with the main Matplotlib installation:

```
from mpl_toolkits import mplot3d
```

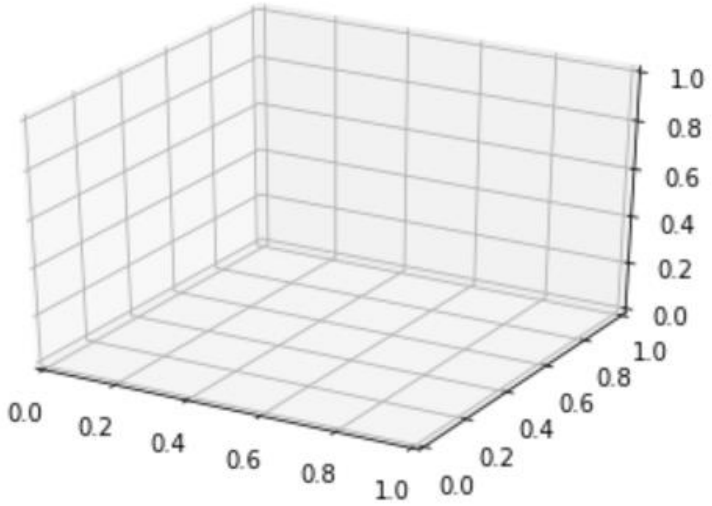
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt
```

- Once this submodule is imported, a three-dimensional axes can be created by passing the keyword projection='3d' to any of the normal axes creation routines

3D Plotting

Northeastern University

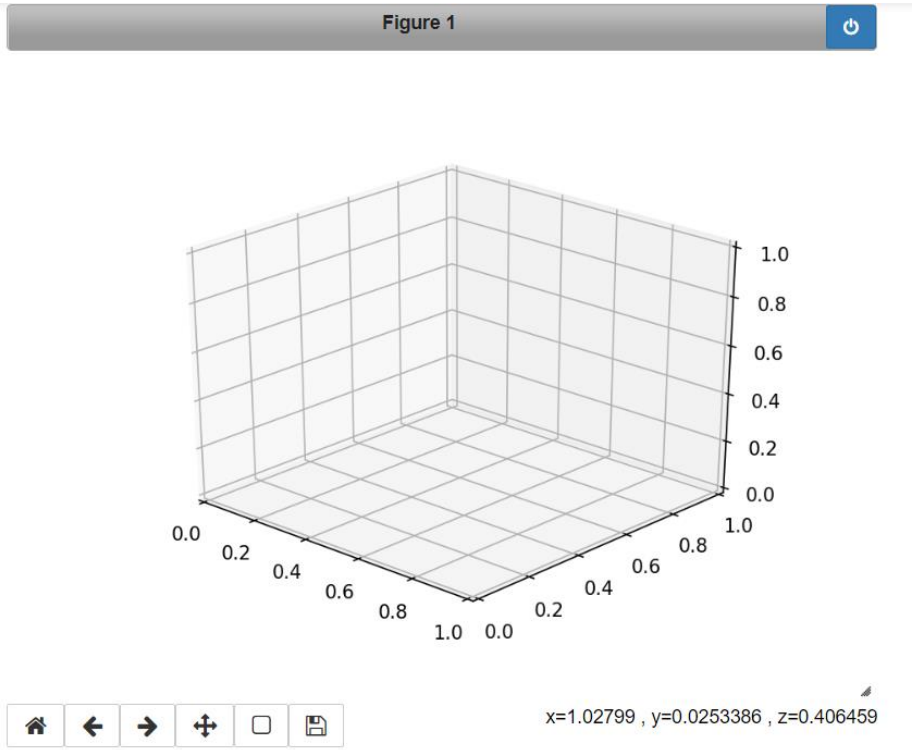
```
fig = plt.figure()
ax = plt.axes(projection='3d')
```



- With this 3D axes enabled, we can now plot a variety of 3D plot types

3D Plotting

- 3D plotting is one of the functionalities that benefits immensely from viewing figures interactively rather than statically in the notebook
- To use interactive figures, run **%matplotlib notebook** instead of %matplotlib inline

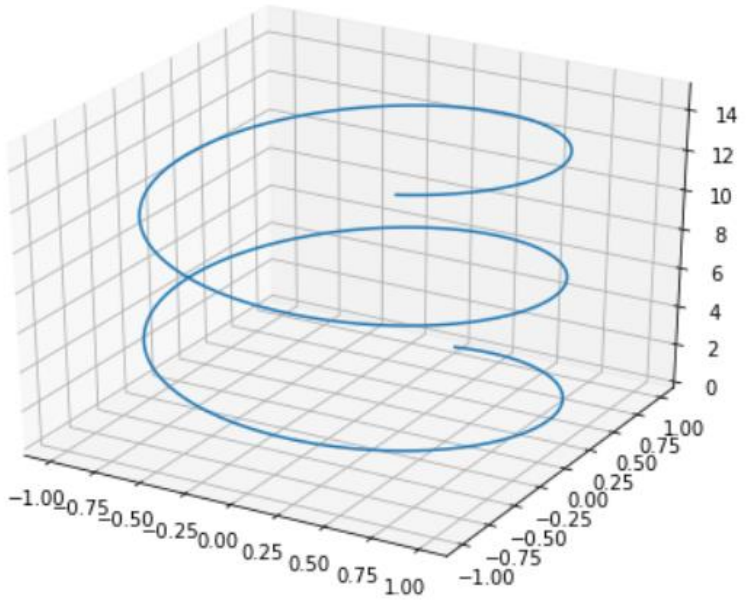


3D Points and Lines

Northeastern University

- The most basic 3D plot is a line or scatter plot created from sets of (x, y, z) triples
- These can be created using the **ax.plot3D()** and **ax.scatter3D()** functions
- The call signature for these is nearly identical to that of their 2D counterparts
- For example, we'll plot a trigonometric spiral:

```
fig = plt.figure(figsize=(8, 6))
ax = plt.axes(projection='3d')

# Data for a 3D line
zline = np.linspace(0, 15, 1000)
xline = np.sin(zline)
yline = np.cos(zline)
ax.plot3D(xline, yline, zline);
```

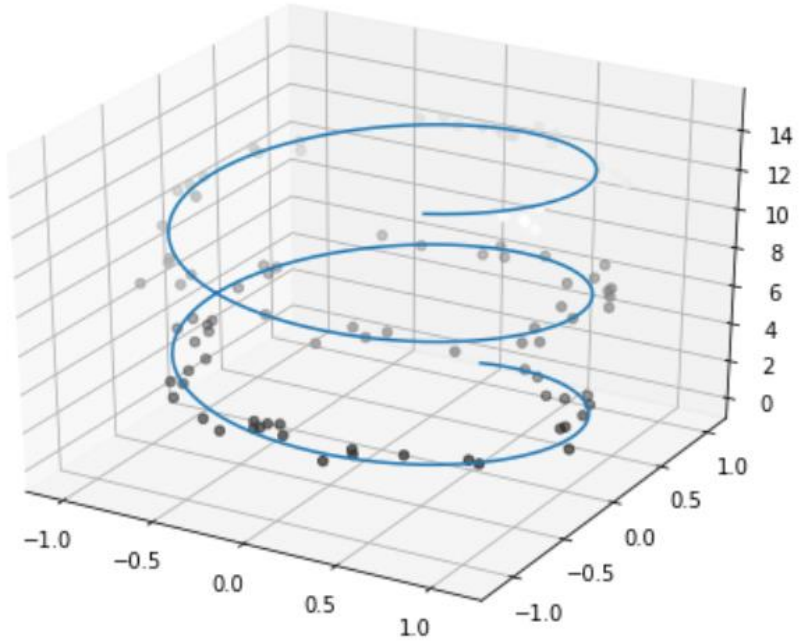


3D Points and Lines

Northeastern University

- Now let’s add a scatter plot of some points drawn randomly near the line:

```
# Data for 3D scattered points
zdata = 15 * np.random.random(100)
xdata = np.sin(zdata) + 0.1 * np.random.randn(100)
ydata = np.cos(zdata) + 0.1 * np.random.randn(100)
ax.scatter3D(xdata, ydata, zdata, c=zdata, cmap='gray')
fig
```



Seaborn

Northeastern University

- Matplotlib has proven to be an incredibly useful and popular visualization tool, but it has its own shortcomings:
  - Matplotlib's defaults are not exactly the best choices. It was based off of MATLAB circa 1999, and this often shows.
  - Matplotlib's API is relatively low level. Doing sophisticated statistical visualization is possible, but often requires a *lot* of boilerplate code.
- Seaborn provides an API on top of Matplotlib that offers additional choices for plot styles, and high-level functions for common statistical plot types

Seaborn

Northeastern University

- For example, **sns.pairlot()** plots multiple pairwise bivariate distributions in a dataset

```
import seaborn as sns
```

```
iris = sns.load_dataset('iris')
sns.pairplot(iris);
```