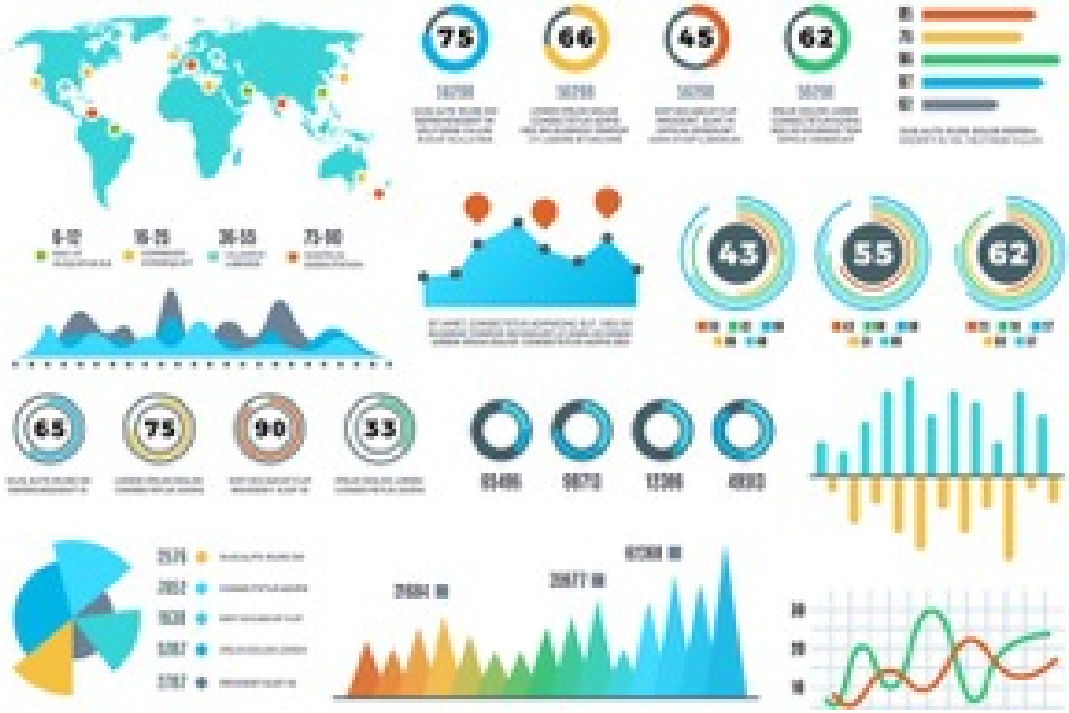
CHAPTER 3: DATA VISUALIZATION

Introduction to Data Science

Posts and Telecommunications Institute of Technology

Ha Noi, 2024

What is Data visualization?

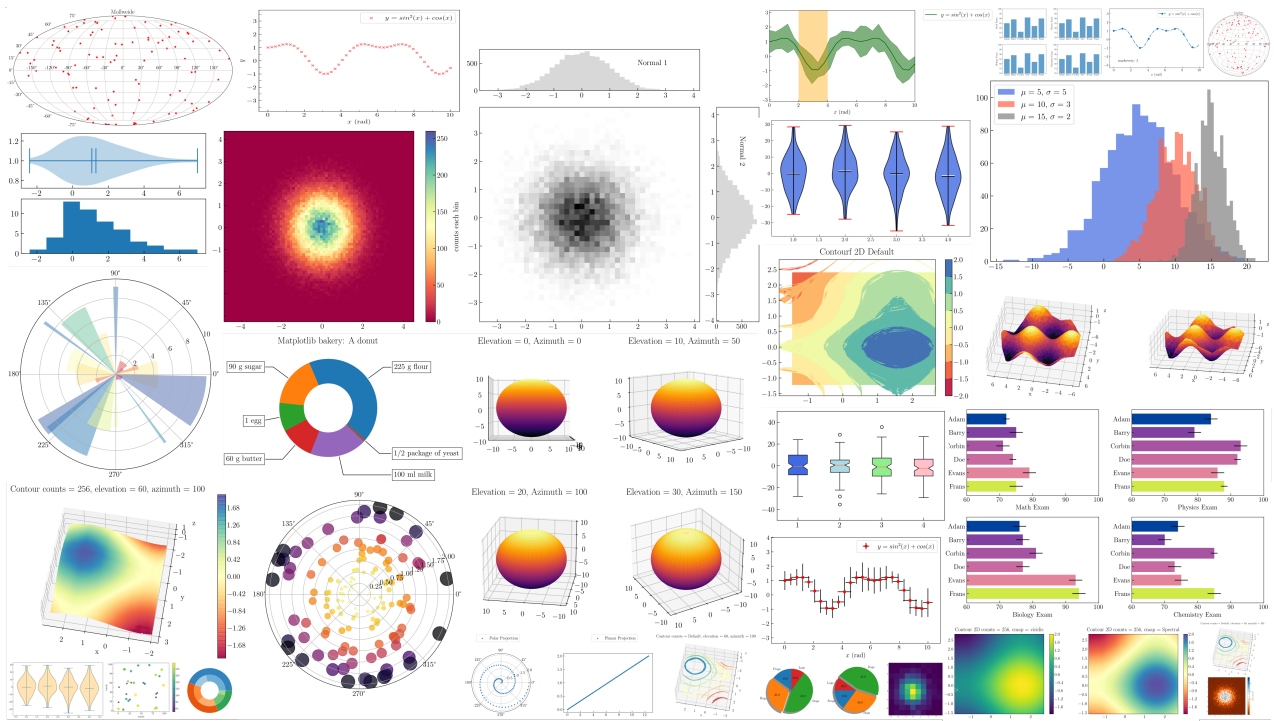


Data visualization is the representation of data through use of common graphics, such as charts, plots, infographics, and even animations. These visual displays of information communicate complex data relationships and data-driven insights in a way that is understand.

CONTENTS

Visualization with Matplotlib

- A multiplatform data visualization library built on NumPy arrays, and designed to work with the broader SciPy stack.
- Work well with multiple platforms and operating systems.
- A large number of users with popularity in the data science world use Python.



Visualization with Matplotlib

Importing matplotlib:

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

Setting Styles:

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

Plotting from a script:

```
# ------- 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()
```

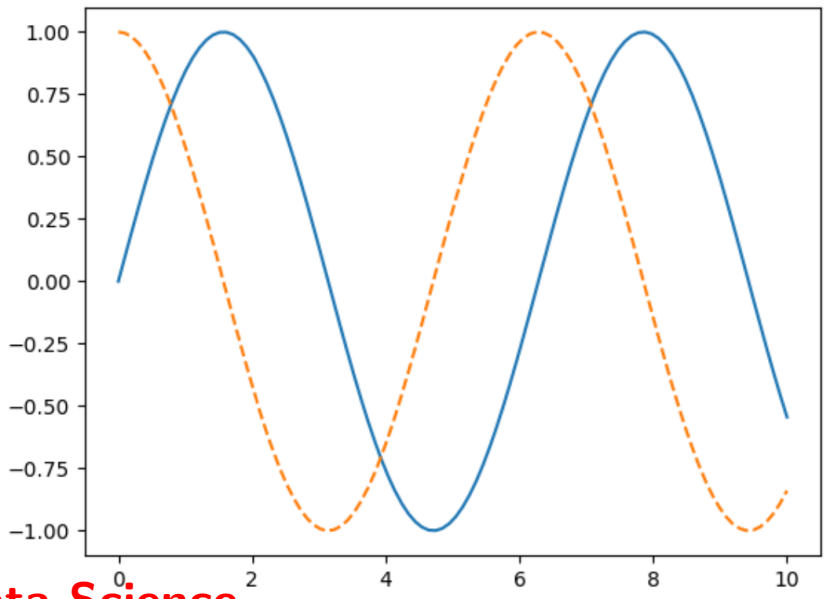

Visualization with Matplotlib

Saving Figures to File

```
fig.savefig('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:

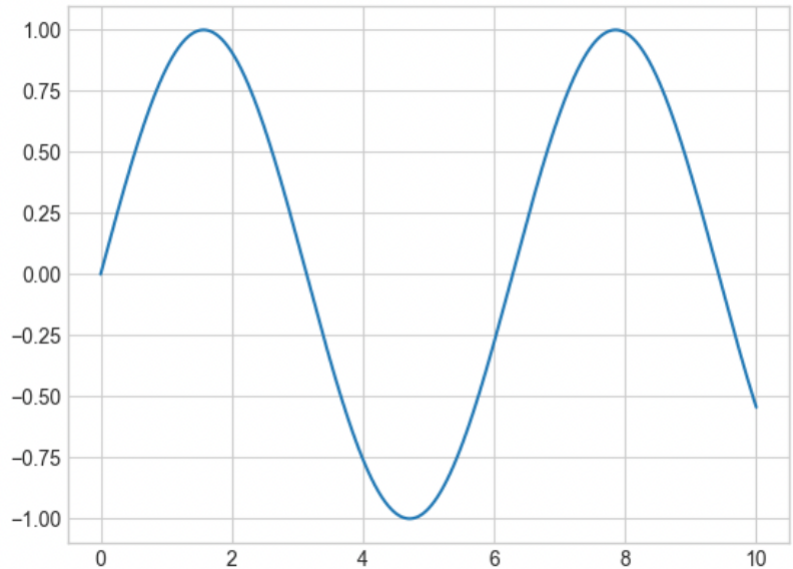
```
from IPython.display import Image
Image('my_figure.png')
```



Line Plots

Perhaps the simplest of all plots is the visualization of a single function $y = f(x)$:

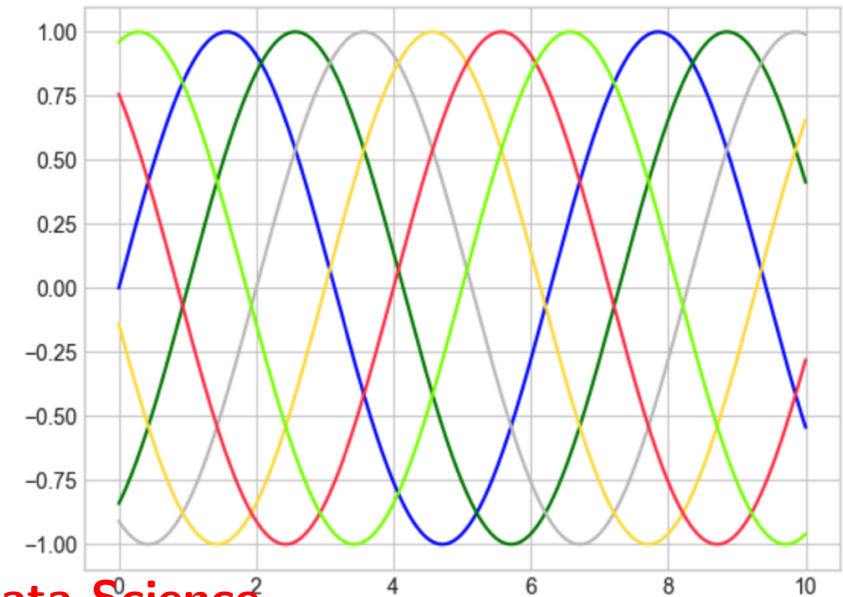
```
In[1]: %matplotlib inline
       import matplotlib.pyplot as plt
       plt.style.use('seaborn-whitegrid')
       import numpy as np
In[2]: fig = plt.figure()
       ax = plt.axes()
```



Line Plots

Adjusting the Plot: Line Colors and Styles - using the *color* keyword to adjust the color, which accepts a string argument representing virtually any imaginable color.

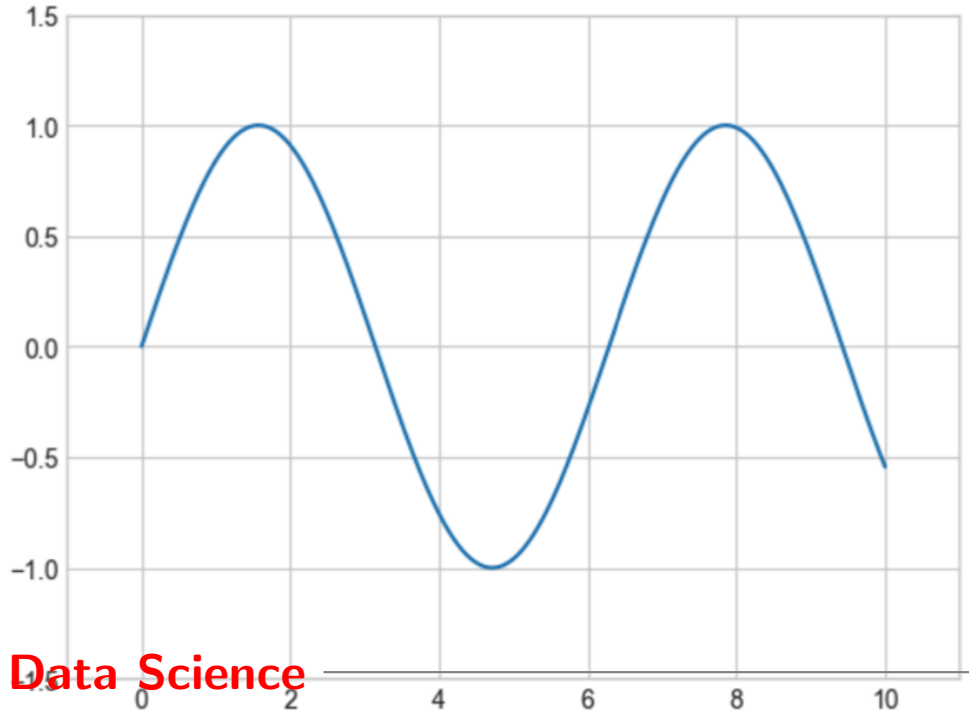
```
plt.plot(x, np.sin(x - 0), color='blue') #Mau cu the theo ten
plt.plot(x, np.sin(x - 1), color='g') #Ma mau bang viet tat (rgbcmyk)
plt.plot(x, np.sin(x - 2), color='0.75') #Thang xam - Grayscale tu 0 den 1
plt.plot(x, np.sin(x - 3), color='#FFDD44') #Ma Hex (RRGGBB tu 00 - FF)
plt.plot(x, np.sin(x - 4), color=(1.0,0.2,0.3)) #RGB, gia tri giua 0 va 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); #Ten mau HTML duoc ho tro
```



Line Plots

Adjusting the Plot: Axes Limits - to adjust axis limits is to use the *plt.xlim()* and *plt.ylim()* methods

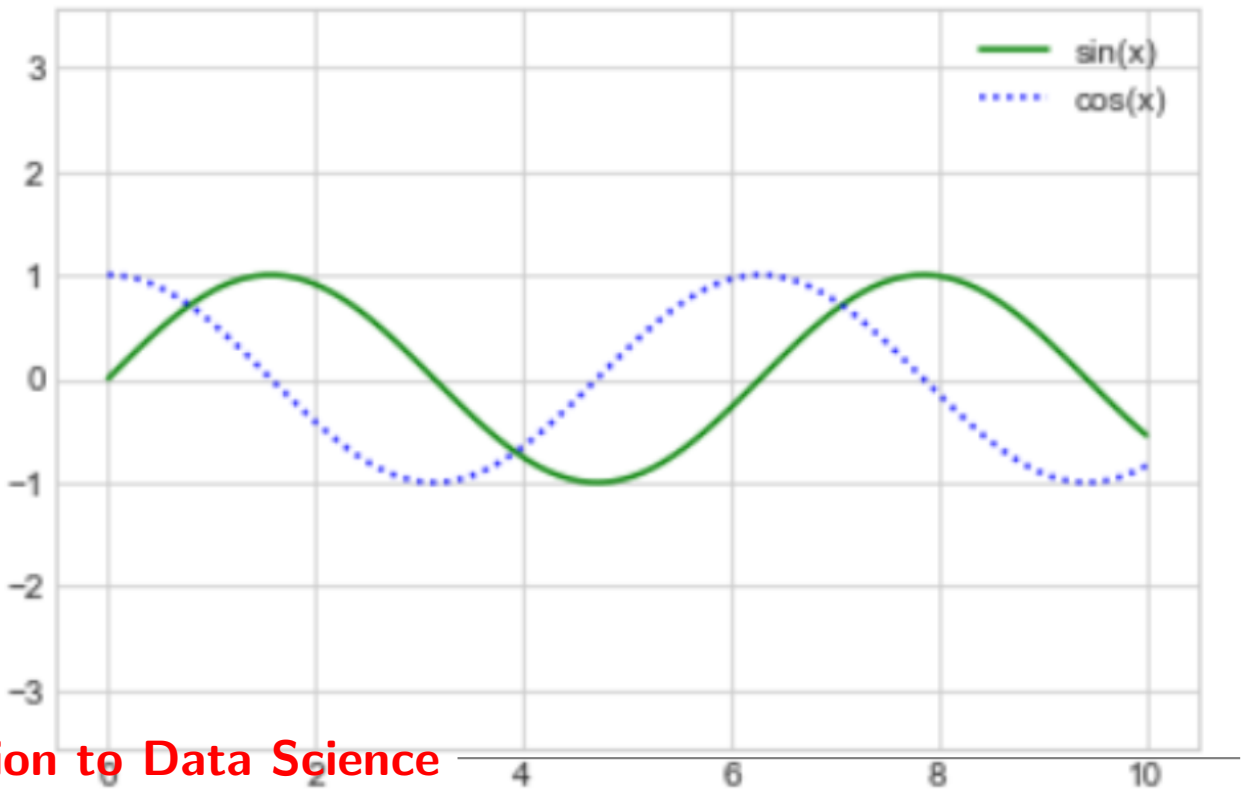
```
plt.plot(x, np.sin(x))
plt.xlim(-1, 11)
plt.ylim(-1.5, 1.5);
```



Line Plots

Labeling Plots:

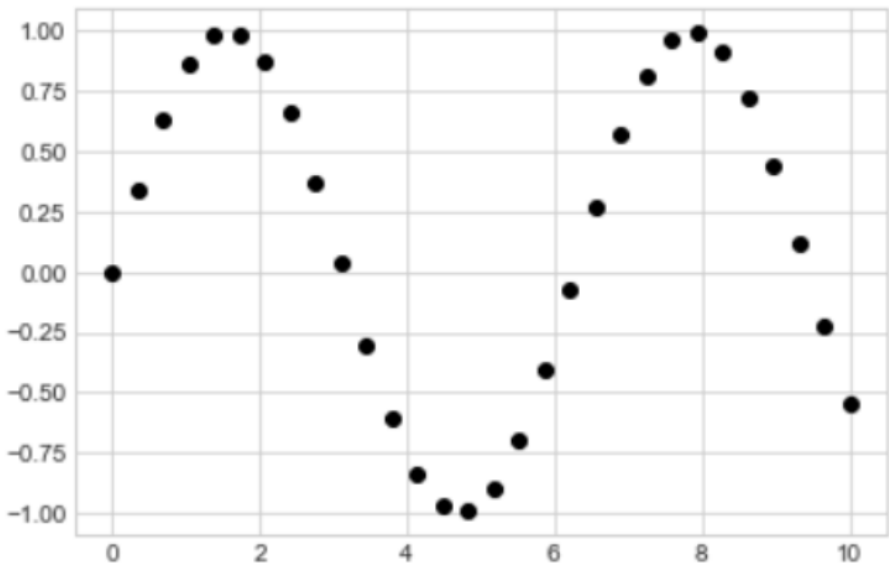
```
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
plt.legend();
```



Scatter Plots

The points are represented individually with a dot, circle, or other shape by *plt.plot/ax.plot*:

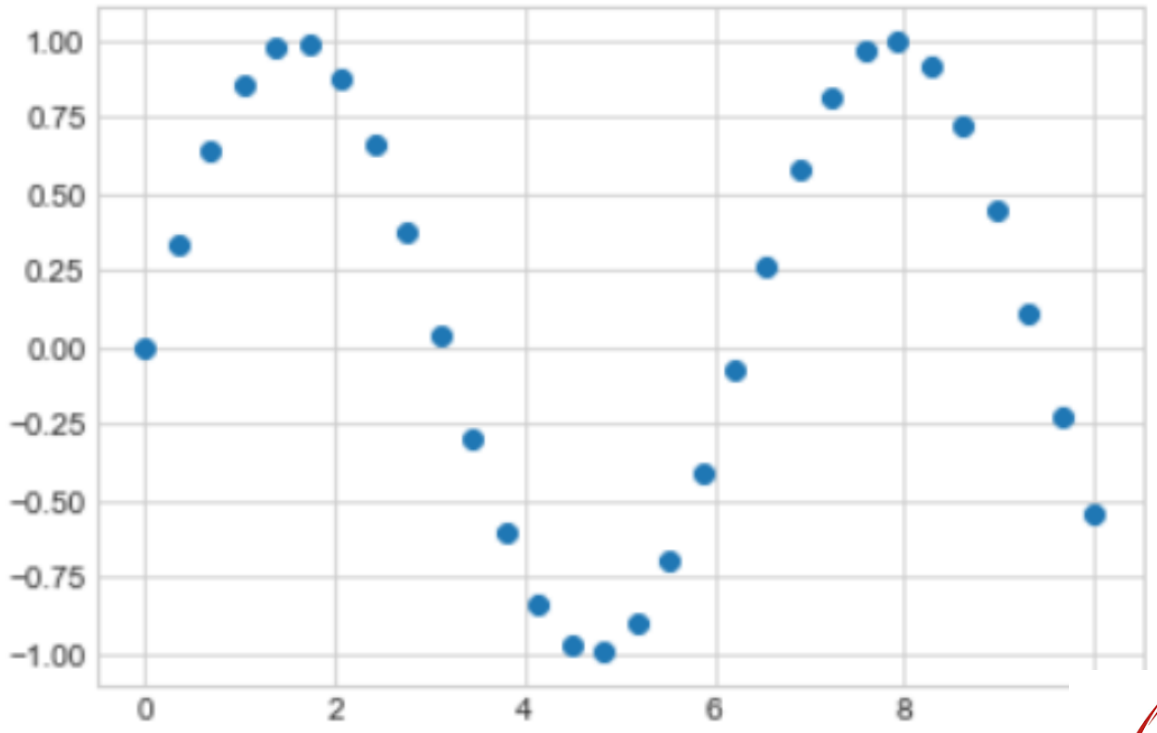
```
plt.plot(x, np.sin(x), 'o', color='black');
```



Scatter Plots

Scatter Plots with *plt.scatter*

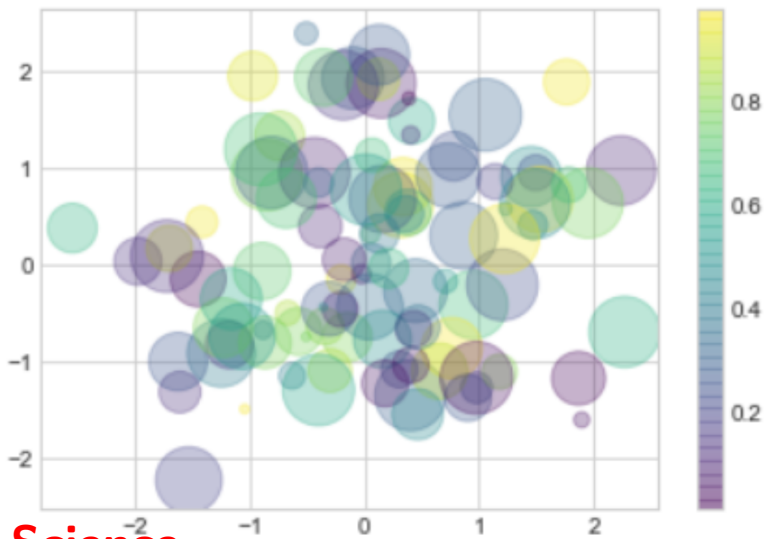
```
plt.scatter(x, y, marker='o');
```



Scatter Plots

Changing size, color, and transparency in scatter points

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
plt.colorbar(); # Hien thi thang mau
```

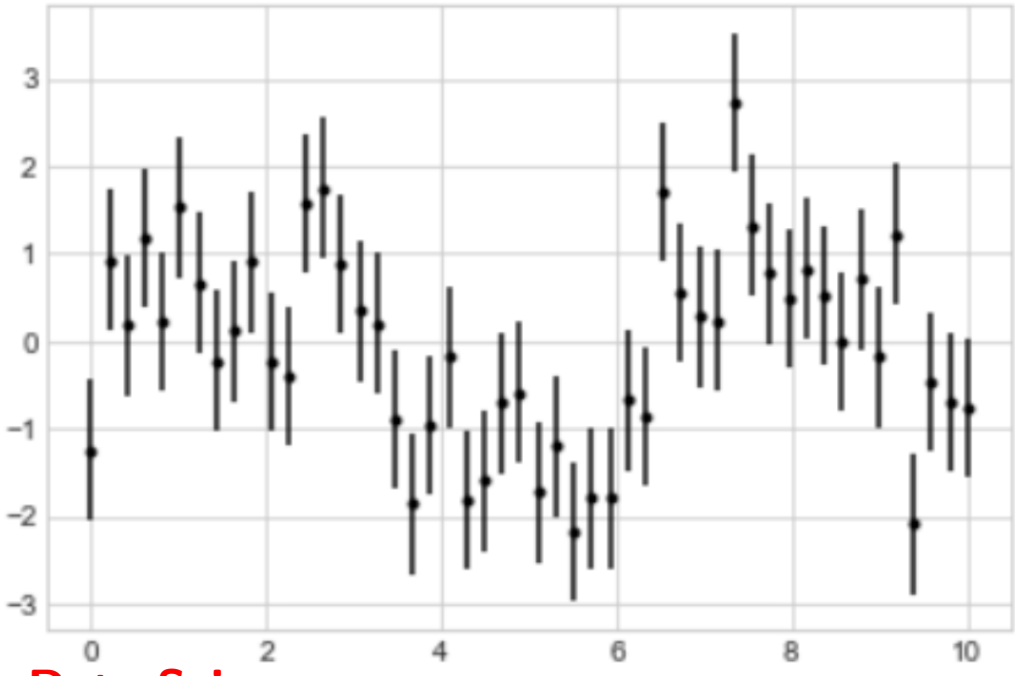


Visualizing Errors

In visualization of data and results, showing these errors effectively can make a plot convey much more complete information.

Basic Errorbars:

```
plt.errorbar(x, y, yerr=dy, fmt='.k');
```



Visualizing Errors

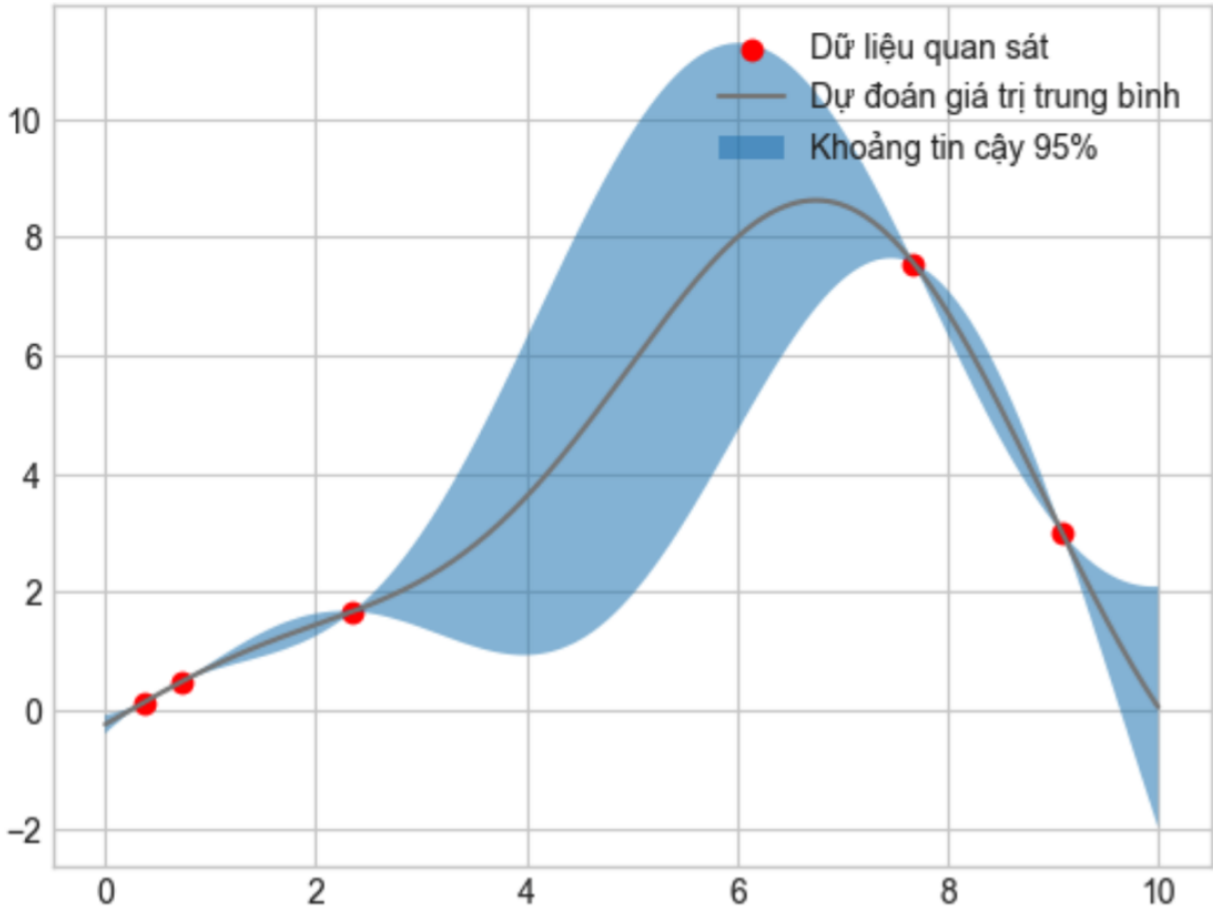
Continuous Errors: combine primitives like *plt.plot* and *plt.fill_between* for a useful result to show errorbars on continuous quantities.

A simple *Gaussian process regression* (GPR), using the Scikit-Learn API as a continuous domain error measure as follows:

```
from sklearn.gaussian_process import GaussianProcessRegressor as GPR
from sklearn.gaussian_process.kernels import RBF

# Khoi tao gia tri cac diem du lieu
model = lambda x: x * np.sin(x)
xdata = np.linspace(0, 10, 1_000).reshape(-1, 1)
ydata = model(xdata)

# Tinh toan hoi quy Gaussian
rng = np.random.RandomState(1)
dtrain = rng.choice(np.arange(ydata.size), size=5)
X_train, y_train = xdata[dtrain], ydata[dtrain]

kernel = 1 * RBF(length_scale=1.0, length_scale_bounds=(1e-2, 1e2))
gp = GPR(kernel=kernel, n_restarts_optimizer=9)
gp.fit(X_train, y_train)
mean, std = gp.predict(X, return_std=True)
```

Visualizing Errors

Continuous Errors:



Density and Contour Plots

Sometimes it is useful to display three-dimensional data in two dimensions using contours or color-coded regions.

–> Matplotlib functions that can be helpful for this task:

- *plt.contour* for contour plots
- *plt.contourf* for filled contour plots
- *plt.imshow* for showing images.

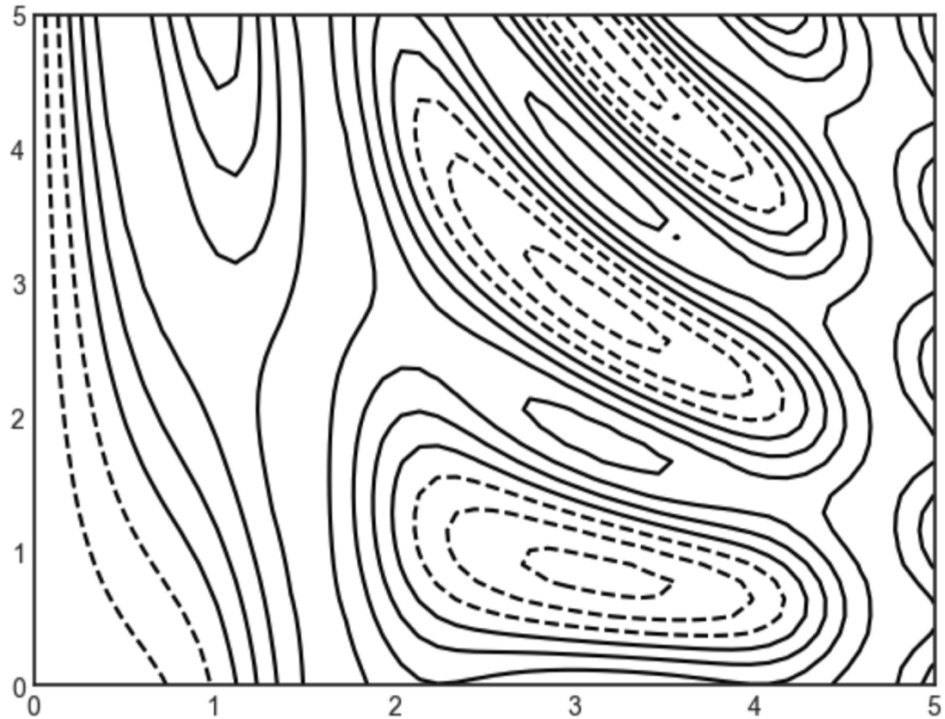
Visualizing a Three-Dimensional Function using a function $z = f(x, y)$

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

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

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

Density and Contour Plots

Visualizing three-dimensional data with contours

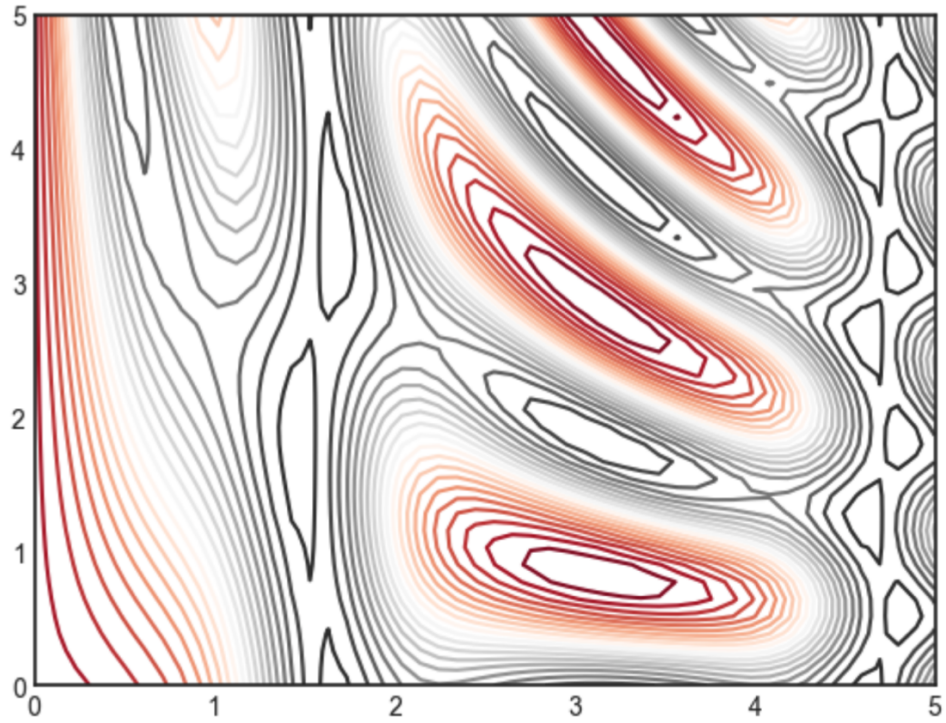
```
plt.contour(X, Y, Z, colors='black');
```



Density and Contour Plots

Visualizing three-dimensional data with colored contours

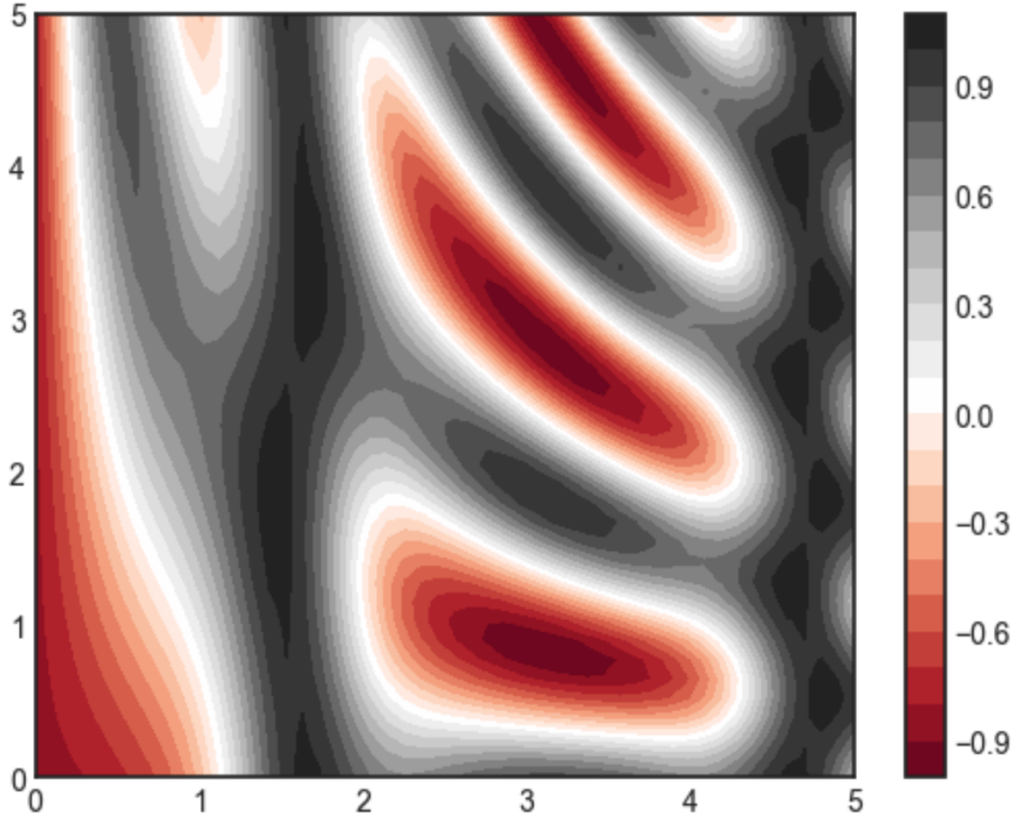
```
plt.contour(X, Y, Z, 20, cmap='RdGy');
```



Density and Contour Plots

Visualizing three-dimensional data with filled contours

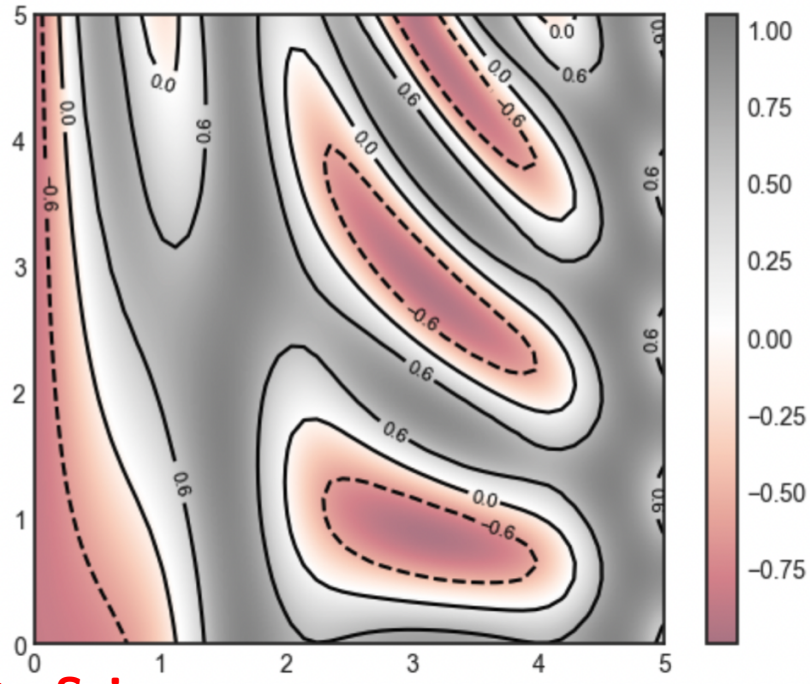
```
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar();
```



Density and Contour Plots

Labeled contours on top of an image:

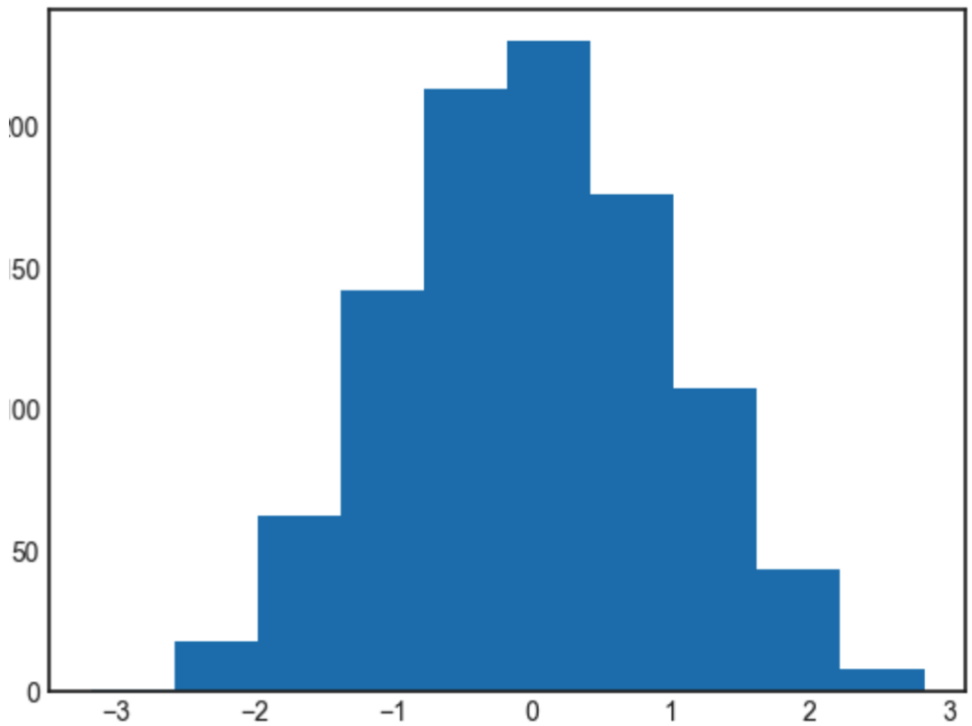
```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)

plt.imshow(Z, extent=[0, 5, 0, 5], origin='lower',cmap='RdGy',
               alpha=0.5, interpolation="bicubic")
plt.colorbar();
```



Histograms, Binnings, and Density

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

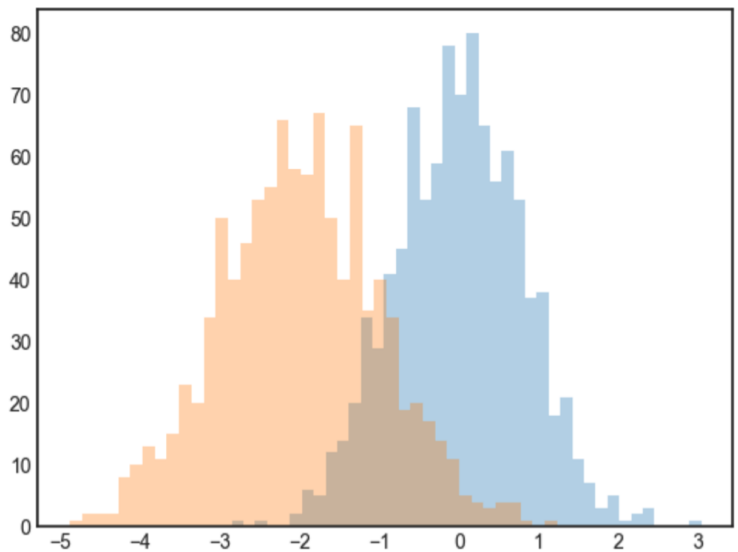
```
data = np.random.randn(1000)
plt.hist(data);
```



Histograms, Binnings, and Density

Combination of *histtype='stepfilled'* along with some transparency alpha to be very useful when comparing histograms of several distributions:

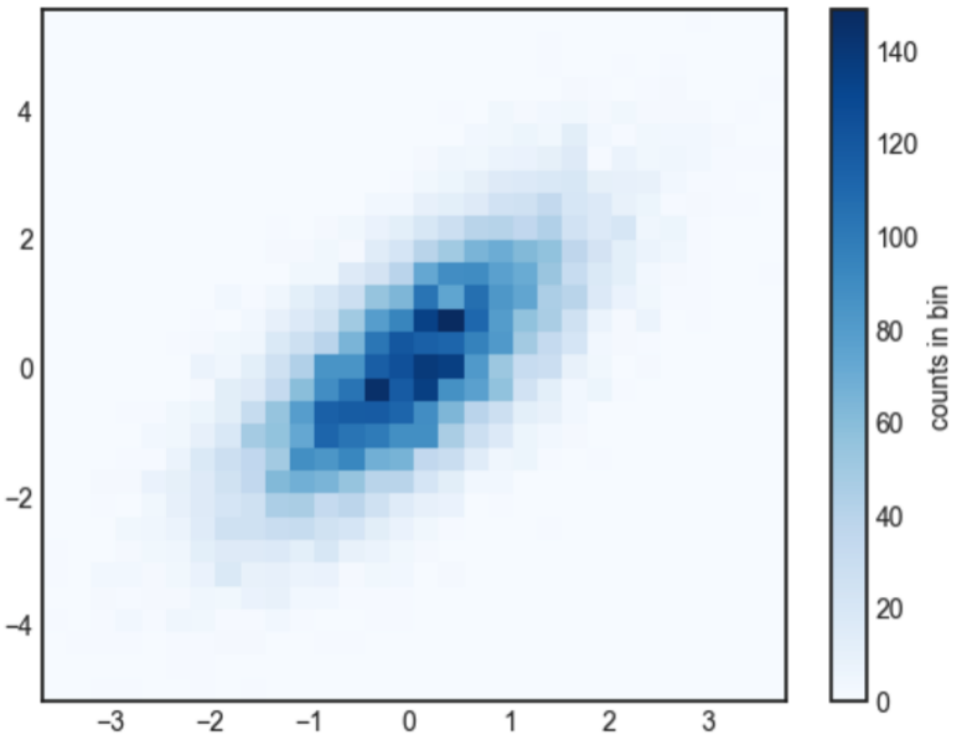
```
x1 = np.random.normal(0, 0.8, 1000)
x2 = np.random.normal(-2, 1, 1000)
kwargs = dict(histtype='stepfilled', alpha=0.3, bins=40)
plt.hist(x1, **kwargs)
plt.hist(x2, **kwargs)
```



Histograms, Binnings, and Density

Two-Dimensional Histograms and Binnings: *plt.hist2d*

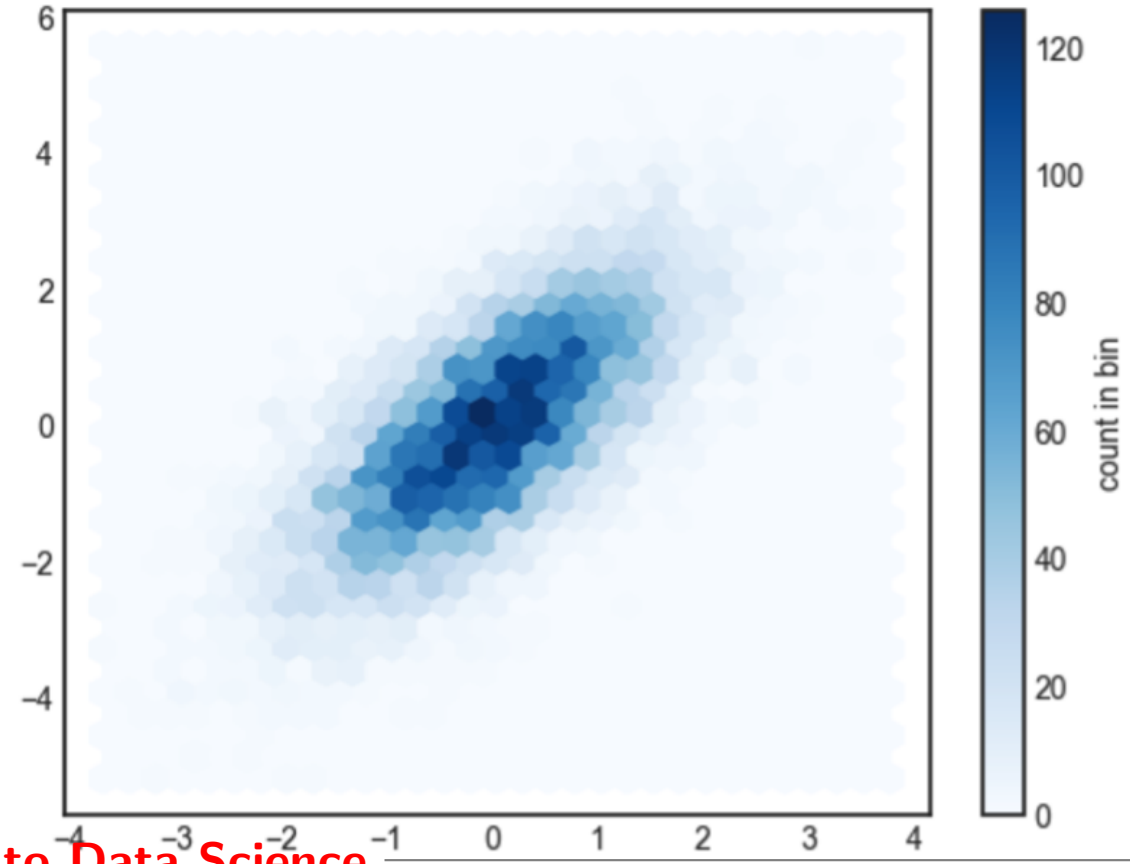
```
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```



Histograms, Binnings, and Density

Hexagonal binnings: *plt.hexbin*

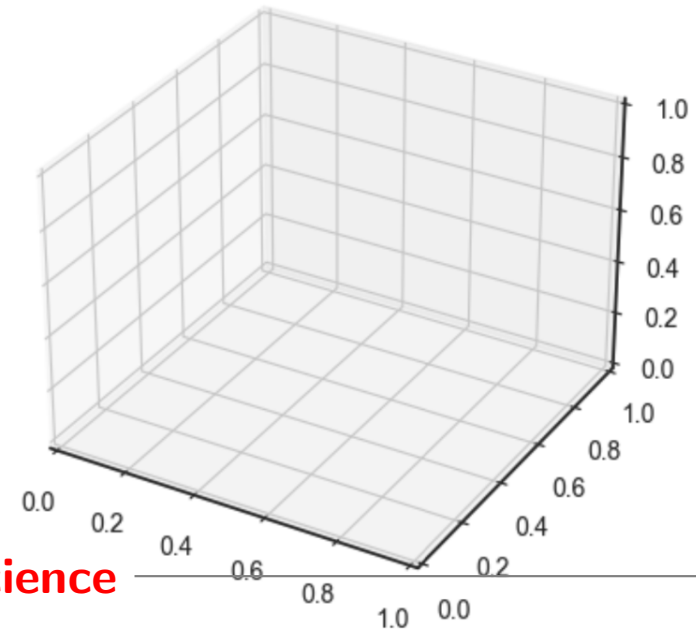
```
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```



Three-Dimensional Plotting

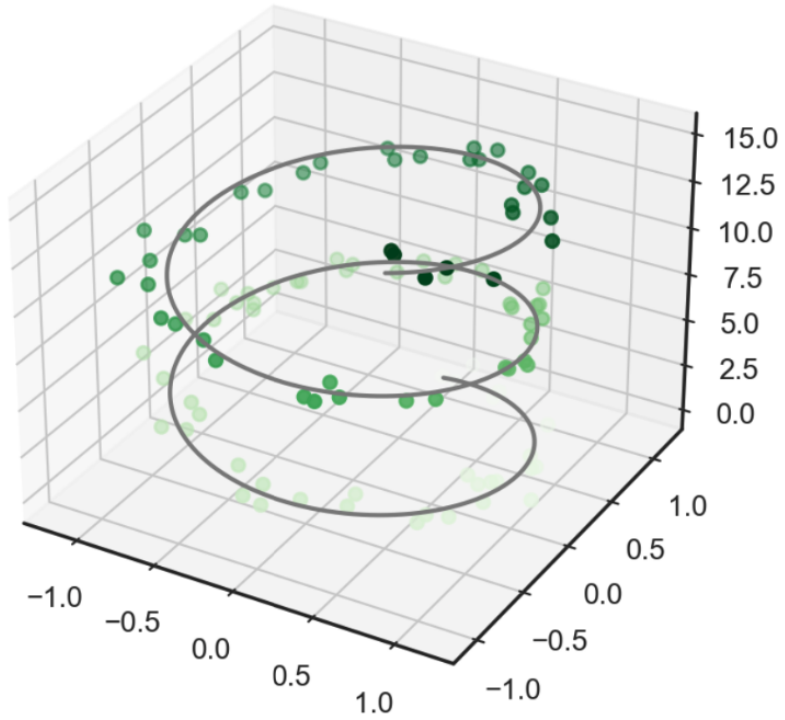
Three-dimensional plots by importing the mplot3d toolkit, included with the main Matplotlib installation:

```
from mpl_toolkits import mplot3d

import numpy as np
import matplotlib.pyplot as plt

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



Three-Dimensional Plotting

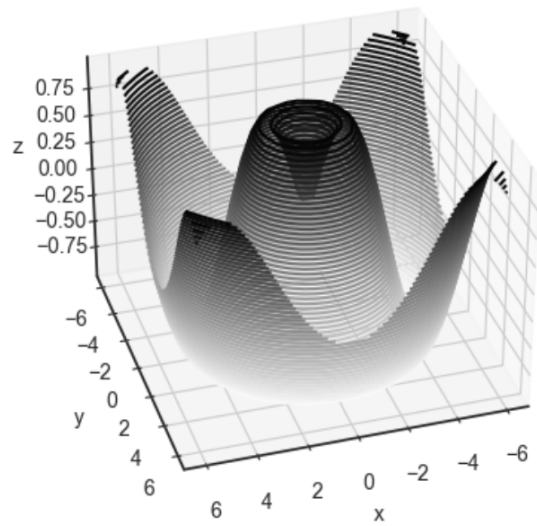
Points and lines in three dimensions



Three-Dimensional Plotting

Three-Dimensional Contour Plots:

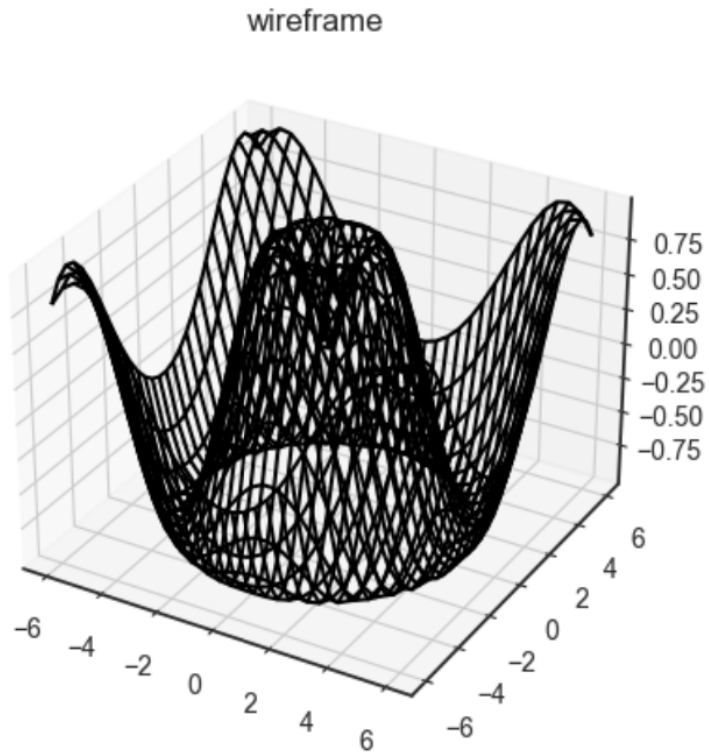
```
def f(x, y):
    return np.sin(np.sqrt(x ** 2 + y ** 2))
x = np.linspace(-6, 6, 30)
y = np.linspace(-6, 6, 30)
X, Y = np.meshgrid(x, y)
Z=f(X,Y)

ax = plt.axes(projection='3d')
ax.contour3D(X, Y, Z, 50, cmap='binary')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z');
```



Three-Dimensional Plotting

Wireframes and Surface Plots:

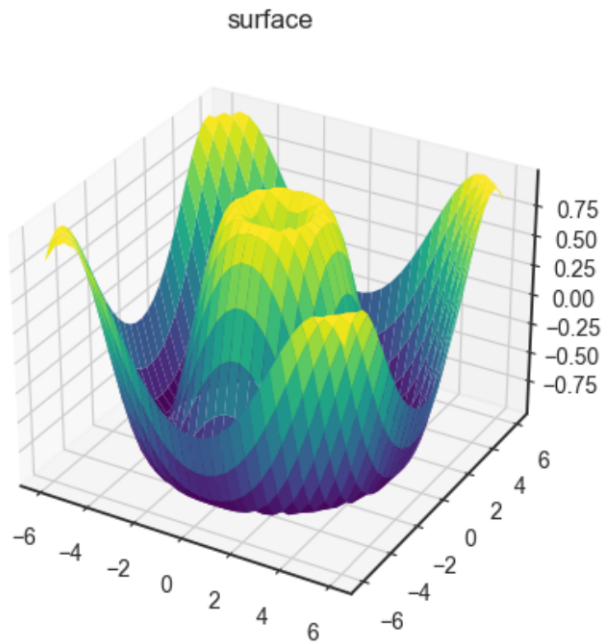
```
ax = plt.axes(projection='3d')
ax.plot_wireframe(X, Y, Z, color='black')
ax.set_title('wireframe');
```



Three-Dimensional Plotting

A three-dimensional surface plot:

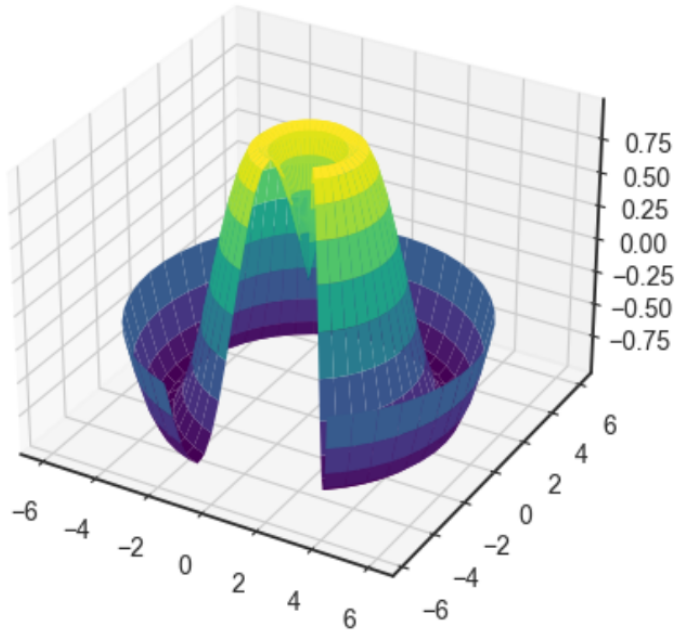
```
ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none')
ax.set_title('surface');
```



Three-Dimensional Plotting

A polar surface plot:

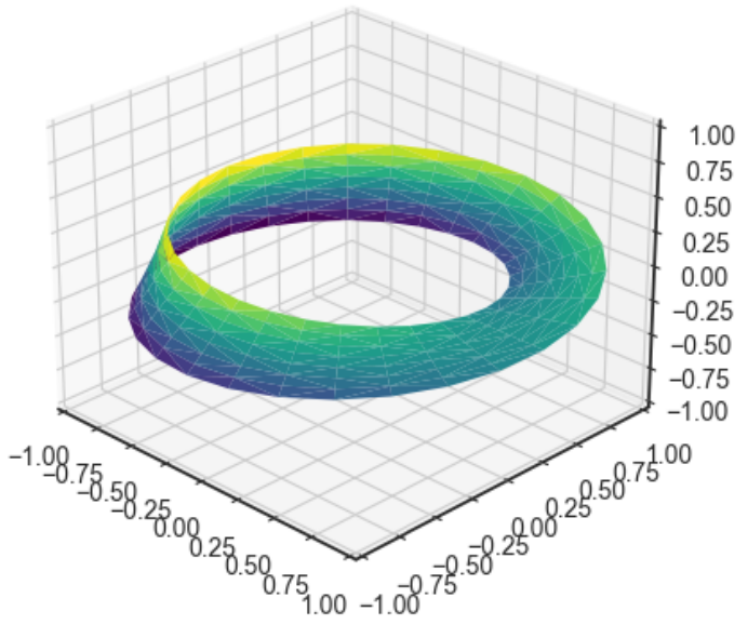
```
r = np.linspace(0, 6, 20)
theta = np.linspace(-0.9 * np.pi, 0.8 * np.pi, 40)
r, theta = np.meshgrid(r, theta)

X = r * np.sin(theta)
Y = r * np.cos(theta)
Z=f(X,Y)

ax = plt.axes(projection='3d')
ax.plot_surface(X, Y, Z, rstride=1, cstride=1,
                cmap='viridis', edgecolor='none');
```



Three-Dimensional Plotting

Example: Visualizing a Mobius strip

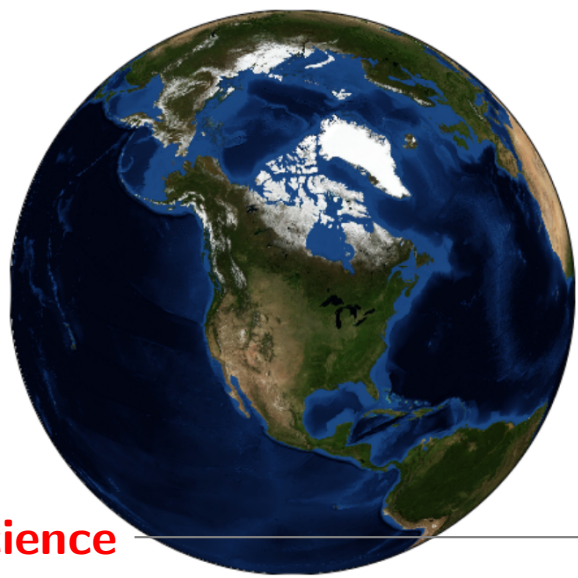
```
from matplotlib.tri import Triangulation
tri = Triangulation(np.ravel(w), np.ravel(theta))

ax = plt.axes(projection='3d')
ax.plot_trisurf(x, y, z, triangles=tri.triangles,
                cmap='viridis', linewidths=0.2);

ax.set_xlim(-1, 1);
ax.set_ylim(-1, 1);
ax.set_zlim(-1, 1);
```



Geographic Data with Basemapa

Matplotlib's main tool for visualization of geographic data is the *Basemap* toolkit, which is one of several Matplotlib toolkits located in the *mpl_toolkits* domain.

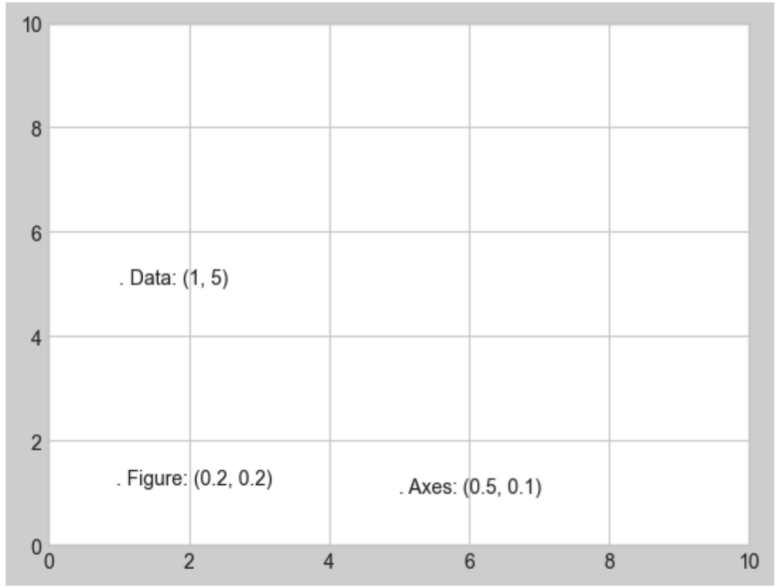
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig = plt.figure(figsize=(8, 8))
m = Basemap(projection='ortho', resolution=None, lat_0=50, lon_0=-100)
m.bluemarble(scale=0.5);
```



Text and Annotation

Text and annotation are essential. The most basic types of annotations to use are the labels of the coordinate axes and the title of the image.

```
ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure);
```




```
import matplotlib.pyplot as plt
import matplotlib as mpl
plt.style.use('seaborn-whitegrid')
import numpy as np
import pandas as pd
```