Data Processing, Analysis and Visualization in Python

Matplotlib

1

Outline

- Introduction
- Line Plots
- Customizing Plots
- Scatter Plots
- Contour Plots
- Bar Plots
- Pie Charts
- Histograms

matplotlib

2

What is Matplotlib?

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

NumPy matplotlib IPython SymPy pandas

https://matplotlib.org/

3

What is Matplotlib?

- 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.
  - Version 0.1 released in 2003.
- Supports many operating systems, graphics backends, and output formats.
- Newer packages and more modern APIs exist
  - R's ggplot
  - Seaborn and others which are built on top of matplotlib
  - Web visualization toolkits based on D3js and HTML5 canvas

4

How do we display plots?

- There are three applicable contexts – using Matplotlib
  1. In a script
  2. In an IPython terminal
  3. In an IPython notebook

5

How do we display plots?

- There are three applicable contexts – using Matplotlib
  1. In a script
     - Use `plt.show()`, which starts an event loop, looks for all currently active figure objects, and opens one or more interactive windows that display your figure or figures.
     - The `plt.show()` command does a lot under the hood, as it must interact with your system's interactive graphical backend.
     - `plt.show()` command should be used *only once* per Python session and is most often seen at the very end of the script.
  2. In an IPython terminal
  3. In an IPython notebook

6

How do we display plots?

- There are three applicable contexts – using Matplotlib
  1. In a script
  2. In an IPython terminal
     - To enable interactive mode, use the `%matplotlib` magic command after starting ipython.
     - Any `plt` plot command will cause a figure window to open, and further commands can be run to update the plot.
     - Some changes will not draw automatically. To force an update, use `plt.draw()`. Using `plt.show()` is not required.
  3. In an IPython notebook

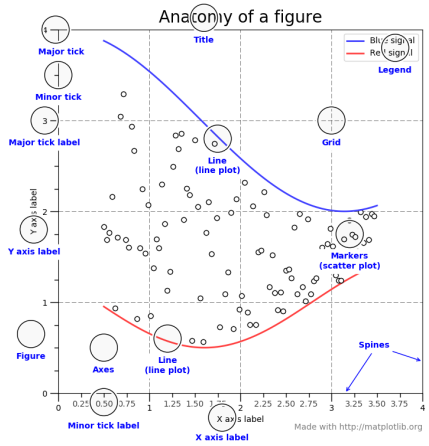
7

How do we display plots?

- There are three applicable contexts – using Matplotlib
  1. In a script
  2. In an IPython terminal
  3. In an IPython notebook
     - `%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
       - After running this command (only once per session), any cell within the notebook that creates a plot will embed a PNG image of the resulting graphic

8

Anatomy of a figure



9

Two interfaces in matplotlib

- Functional – a convenient MATLAB-style state-based interface
  - 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.
  - Convenient for simple plots. Problematic, for example, once a second panel is created, how can we go back and add something to the first?
- Object Oriented – more powerful
  - This interface is available for more complicated situations, and for when more control over the figure is required.
  - 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.

10

Importing matplotlib in functional interface

```
>>> import matplotlib as mpl
>>> mpl.__version__
'3.4.3'
>>> import matplotlib.pyplot as plt
```

- Understanding matplotlib's pyplot API is key to understanding how to work with plots:
  - ***matplotlib.pyplot.figure: Figure*** is the top-level container. It includes everything visualized in a plot including one or more ***Axes***.
  - ***matplotlib.pyplot.axes****: **Axes*** contain most of the elements in a plot**: *Axis, Tick, Line2D, Text,*** etc., and sets the coordinates. It is the area in which data is plotted. Axes include the x-Axis, y-Axis, and possibly a z-Axis, as well.

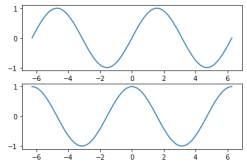
11

Functional interface

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

plt.figure()
plt.subplot(2, 1, 1) # (rows, columns, panel number)
x = np.linspace(-2*np.pi, 2*np.pi, 100)
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))
```

Library with functions acting on variables and functions



plt_sin_cos_1.py

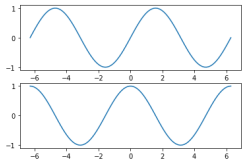
12

Object Oriented interface

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

fig, ax = plt.subplots(2)
x = np.linspace(-2*np.pi, 2*np.pi, 100)
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))
```

Objects with methods acting on object



plt_sin_cos_2.py

13

Line plots

- Show the change in value of an attribute with respect to a x-variable
- Can be used to visually compare the values of several related attributes

```
fig = plt.figure()
ax = plt.axes()
x = np.linspace(0, 10, 1000)
ax.plot(x, np.sin(x));
# Alternatively
plt.plot(x, np.sin(x))
```

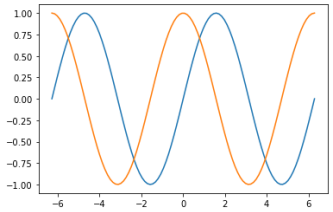
plt_sin_cos_3.py

14

Line plots

- Show the change in value of an attribute with respect to a x-variable
- Can be used to visually compare the values of several related attributes

```
ax.plot(x, np.sin(x))
ax.plot(x, np.cos(x))
# Alternatively
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
```



plt_sin_cos_3.py

15

Defining elements of a plot

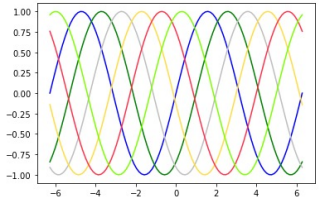
- Plot elements add context to a plot, so the plot effectively conveys meaning to its viewers
  - You set line colors, styles, and widths to differentiate between different line objects
  - You set axes limits to make sure that your chart fits your data
  - You set axes tick marks and plot grids to make it easier and faster for the viewers to interpret your chart
  - You create a legend to label each line object
  - You can use subplots to visually compare changes in data values under different conditions, like different seasons, different locations, or in different years

16

Line color

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)

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 to 1
plt.plot(x, np.sin(x - 5), color='chartreuse'); # HTML color names
```



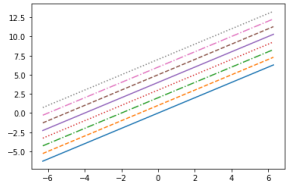
plt_sin_cos_4.py

17

Line style

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)

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
```



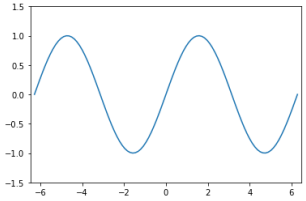
plt_sin_cos_5.py

18

Axes limits

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)

plt.plot(x, np.sin(x))
plt.xlim(-2*np.pi - 0.2, 2*np.pi + 0.2)
plt.ylim(-1.5, 1.5)
# Alternatively
#plt.axis([-2*np.pi - 0.2, 2*np.pi + 0.2, -1.5, 1.5]);
```



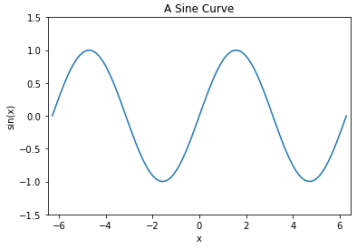
plt_sin_cos_6.py

19

Labeling plots

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)

plt.plot(x, np.sin(x))
plt.xlim(-2*np.pi - 0.2, 2*np.pi + 0.2)
plt.ylim(-1.5, 1.5)
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)")
```



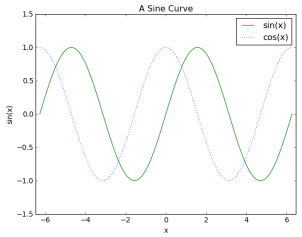
plt_sin_cos_6.py

20

Legends

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)


plt.xlim(-2*np.pi - 0.2, 2*np.pi + 0.2)
plt.ylim(-1.5, 1.5)
# Add titles & labels
plt.title("A Sine Curve")
plt.xlabel("x")
plt.ylabel("sin(x)")
# Add legends
plt.plot(x, np.sin(x), '-g', label='sin(x)')
plt.plot(x, np.cos(x), ':b', label='cos(x)')
# one unit in x is equal to one unit in y
plt.legend()
```



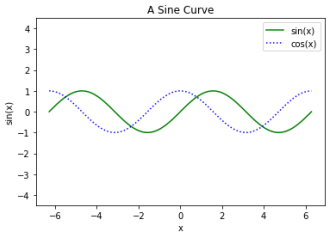
plt_sin_cos_7.py

21

plt functions to ax methods translation

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)
```

- plt.plot() → ax.plot()
- plt.legend() → ax.legend()
- but
- 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()

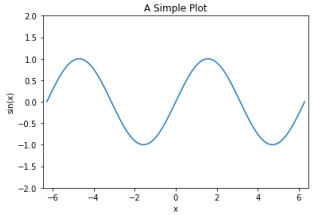


22

Set all properties at once

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)


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')
```



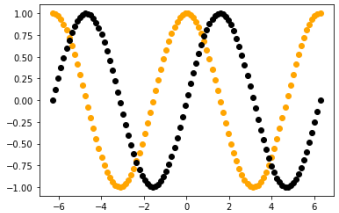
plt_sin_cos_8.py

23

Scatter plot using scatter

```
x = np.linspace(-2*np.pi, 2*np.pi, 100)


y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1, 'o', color='black')
plt.scatter(x, y2, marker='o', color='orange')
```



plt_sin_cos_9.py

24

Scatter plot using scatter

- 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, fill color, edge color, etc.) can be individually controlled or mapped to data.

```
import matplotlib.pyplot as plt
import numpy as np
import time as t

cm =['viridis', 'plasma', 'inferno', 'magma', 'cividis']
rng = np.random.RandomState(seed=int(t.time()))
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=cm[rng.randint(4)])
plt.colorbar() # show color scale
```

plt_scatter.py

25

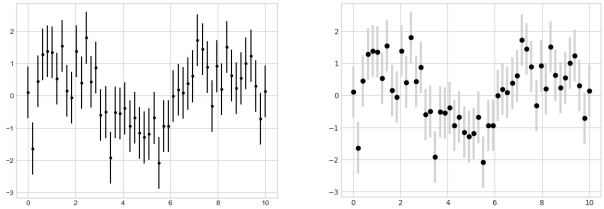
Plot or Scatter?

- `plt.scatter()` is more powerful and has the capability to render a different size and/or color for each point
- `plt.plot()` is more efficient for datasets with larger than a few thousand points, as the appearance of the points is the same for the entire dataset

26

Plotting Error Bars

```
# Plotting Error Bars
>>> plt.style.use('seaborn-whitegrid')
>>> x = np.linspace(0, 10, 50)
>>> dy = 0.8
>>> y = np.sin(x) + dy * np.random.randn(50)
>>> plt.errorbar(x, y, yerr=dy, fmt='.k')
```



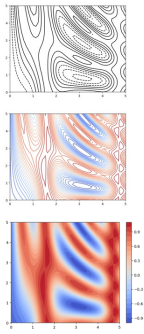
27

Contour Plots

```
>>> # Contour plots
>>> 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) # Create 2D grid
>>> Z = f(X, Y)
>>> plt.contour(X, Y, Z, colors='black')
>>> plt.show()
>>> plt.contour(X, Y, Z, 20, cmap='coolwarm')
>>> plt.show()
>>> plt.contourf(X, Y, Z, 20, cmap='coolwarm')
>>> plt.colorbar()
>>> plt.show()
```



28

Contour Plots Exercise

- Write a script which draws a contour plot (of your choice) for the following function:

$$f(x,y) = \cos\left(\frac{x}{2}\right) + \sin\left(\frac{y}{4}\right)$$

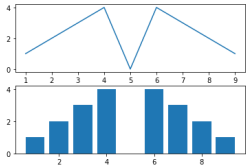
- for the range

- x: $x \in (0, 50, 2)$
- y: $y \in (0, 50, 3)$

29

Bar plots

- Represents data attribute values within a particular data category by using bars of different heights
- Bar charts represent observation counts within categories



```
import matplotlib.pyplot as plt

plt.figure()
x = range(1,10)
y = [1,2,3,4,0,4,3,2,1]
plt.subplot(2, 1, 1) # (rows, columns, panel number)
plt.plot(x,y)
plt.subplot(2, 1, 2) # (rows, columns, panel number)
plt.bar(x, y)
```

plt_bar.py

30

Bar plots

```
plt.figure()
x = range(1,10)
# Index 4 and 6 demonstrate overlapping cases
x1 = [1, 3, 4, 5, 6, 7, 9]
y1 = [4, 7, 2, 4, 7, 8, 3]
x2 = [2, 4, 6, 8, 10]
y2 = [5, 6, 2, 6, 2]
plt.bar(x1, y1, label='Blue Bar', color='b')
plt.bar(x2, y2, label='Green Bar', color='g')
plt.xlabel("bar number")
plt.ylabel("bar height")
plt.title("Bar Chart")
plt.legend()
```

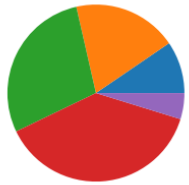
plt_bar.py

31

Pie charts

- Represents data attribute values using a circle and slices that comprise it
- A whole and entire set of categorical data is represented by the complete circle, and the proportions of observations that fall into the different categories are represented by proportionate pie slices

```
x = [1, 2, 3, 4, 0.5]
plt.pie(x)
plt.show()
```



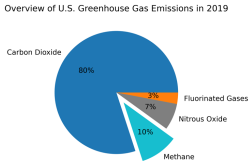
plt_pie.py

32

Pie charts

```
import matplotlib.pyplot as plt

fig = plt.figure()
x = [80, 10, 7, 3] # Raw data
labels = ['Carbon Dioxide', 'Methane', 'Nitrous Oxide',
          'Fluorinated Gases']
colors = ['tab:blue', 'tab:cyan', 'tab:gray', 'tab:orange']
explode = [0, 0.2, 0, 0]
fig, ax = plt.subplots()
ax.pie(x, labels = labels, colors = colors, autopct='%.0f%%',
explode = explode)
ax.set_title('Overview of U.S. Greenhouse Gas Emissions in 2019')
fig.savefig('greenhouse.png', dpi=600)
```



plt_pie.py

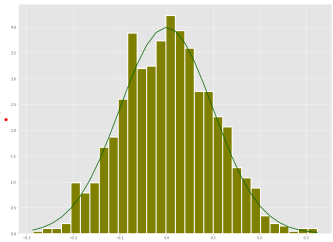
https://www.epa.gov/ghgemissions/overview-greenhouse-gases

33

Histograms

```
import matplotlib.pyplot as plt
import numpy as np
import matplotlib.style  as st

mu, sigma = 0, 0.1 # mean and st. dev.
s = np.random.normal(mu, sigma, 1000)
nbins = 30
st.use('ggplot') # return grid
plt.figure(figsize = (16,12))
count, bins, ignored = plt.hist(s, nbins, density=True, color='olive')
plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
         np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
         linewidth=2, color='darkgreen')
```

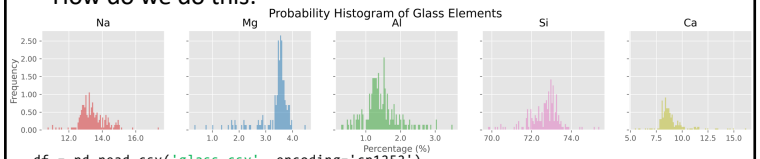


plt_hist.py

34

Histograms

- How do we do this:




```
df = pd.read_csv('glass.csv', encoding='cp1252')
…
elements = ['Na', 'Mg', 'Al', 'Si', 'Ca']
fig, axes = plt.subplots(1, 5, figsize=(17.5,2.5), dpi=600, sharex=False, sharey=True)
colors = ['tab:red', 'tab:blue', 'tab:green', 'tab:pink', 'tab:olive']
for i in range(len(elements)):
    na_data = df[elements[i]].values
    axes[i].hist(na_data, alpha=0.5, bins=100, density=True, stacked=True,
                 label=elements[i], color=colors[i])
    axes[i].set_title(elements[i])
…
axes[0].set_ylabel("Frequency"); axes[2].set_xlabel("Percentage (%)")
```

plt_hist_periodic_table.py

35

Pandas and matplotlib – other plot types

- Plotting methods allow for a handful of plot styles other than the default line plot. These methods can be provided as the kind keyword argument to plot(), and include:
- 'bar' or 'barh' for bar plots
- 'hist' for histogram
- 'box' for boxplot
- 'kde' or 'density' for density plots
- 'area' for area plots
- 'scatter' for scatter plots
- 'hexbin' for hexagonal bin plots
- 'pie' for pie plots

37

Plotting with missing data

- Pandas tries to be pragmatic about plotting DataFrames or Series that contain missing data.
- Missing values are dropped, left out, or filled depending on the plot type.
- If any of these defaults are not what you want, or you want to be explicit about how missing values are handled, consider fillna() or dropna() before plotting.

38

Plotting with missing data

Plot Type	NaN Handling
Line	Leave gaps at NaNs
Line (stacked)	Fill 0's
Bar	Fill 0's
Scatter	Drop NaNs
Histogram	Drop NaNs (column-wise)
Box	Drop NaNs (column-wise)
Area	Fill 0's
KDE	Drop NaNs (column-wise)
Hexbin	Drop NaNs
Pie	Fill 0's

39

Matplotlib drawbacks

- Prior to version 2.0, 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.
- Matplotlib predated Pandas by more than a decade, and thus is not designed for use with Pandas DataFrames.
  - In order to visualize data from a Pandas DataFrame, you must extract each Series and often concatenate them together into the right format.

It would be nicer to have a plotting library that can intelligently use the DataFrame labels in a plot…

40

41