Data Visualization

Basics

- Use matplotlib.pyplot
  - Based on matlab
    - Allows
      - histograms
      - line plots
      - box-plots
      - scatter plots
      - hex-density plots

Basics

- Import numpy as np
- Import pandas as pd
- Import matplotlib.pyplot as plt

Basic Example

- Import an artificial time series

```
>>> ts1 = pd.read_csv('../Data/ts1.csv')
```

- Show it:

```
>>> ts1.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 2 columns):
time     5000 non-null int64
 TS      5000 non-null float64
dtypes: float64(1), int64(1)
memory usage: 78.2 KB
```

Basic Example

- Use head and tail

```
ts1.head()
ts1.tail()
```

- To make it more realistic, we need to make the index into one with actual dates
- Drop the column 'time'
  - We want to change the data frame, so we need to set `inplace` to `True`

Basic Example

```
>>> ts1.drop(columns=['time'], inplace=True)
>>> ts1.head()
            TS
0  1027.096129
1  1041.701344
2  1046.905793
3  1038.360279
4  1033.118933
```

- Create a **new** column with dates starting at January 1, 2001.
  - Use Bing to Google the name of the function:

```
>>> ts1['time'] = pd.date_range(start='1/1/2001',
periods = 5000)
```

Basic Example

- We still have an index, but now a new column

```
>>> ts1['time'] = pd.date_range(start='1/1/2001',
periods=5000)
>>> ts1.head()
             TS       time
0  1027.096129 2001-01-01
1  1041.701344 2001-01-02
2  1046.905793 2001-01-03
3  1038.360279 2001-01-04
4  1033.118933 2001-01-05
```

Basic Example

- Now we can re-index by setting the index

```
>>> ts1.set_index('time', inplace = True)

>>> ts1.info()
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 5000 entries, 2001-01-01 to 2014-09-09
Data columns (total 1 columns):
 TS    5000 non-null float64
dtypes: float64(1)
memory usage: 78.1 KB

>>> ts1.head()
                      TS
time
2001-01-01  1027.096129
2001-01-02  1041.701344
```

Basic Example

- If we try to only access the TS data, we run into a problem

```
>>> ts1.TS
Traceback (most recent call last):
  File "<pyshell#73>", line 1, in <module>
    ts1.TS
  File "/Library/Frameworks/Python.framework/
Versions/3.8/lib/python3.8/site-packages/pandas/
core/generic.py", line 5179, in __getattr__
    return object.__getattribute__(self, name)
AttributeError: 'DataFrame' object has no attribute
'TS'
```

Basic Example

- We can look at the columns of the data frame

```
>>> ts1.columns
Index([' TS'], dtype='object')
```

- And now we see the problem (cost me about an hour of my life)

```
>>> ts1.columns
Index([' TS'], dtype='object')
```

- The csv file has an additional white space after the comma

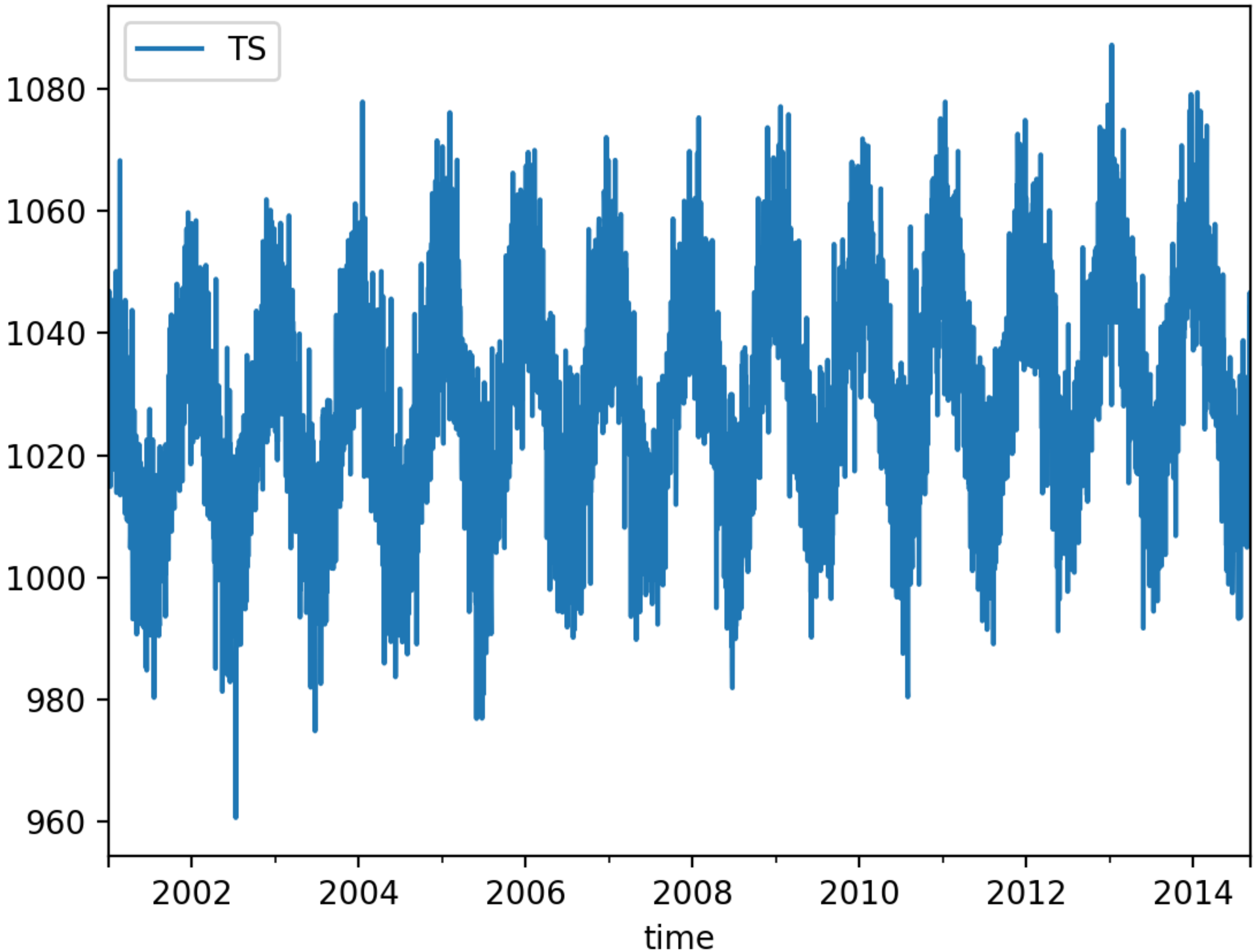
Basic Example

- We better rename that column

```
>>> ts1.rename(columns={' TS': 'TS'}, inplace = True)
>>> ts1.TS
time
2001-01-01    1027.096129
2001-01-02    1041.701344
2001-01-03    1046.905793
2001-01-04    1038.360279
2001-01-05    1033.118933
                  ...
2014-09-05    1019.451193
2014-09-06    1017.043391
2014-09-07    1046.658204
2014-09-08    1030.316278
2014-09-09    1044.078304
Name: TS, Length: 5000, dtype: float64
[5000 rows x 1 columns]
```

Basic Example

- We can now use the plotting component of Pandas




```
ts1.plot()
plt.show()
```

Basic Example

- We can also do a scatter graph
  - But this needs to be specialized because scatter graphs usually need two numeric values

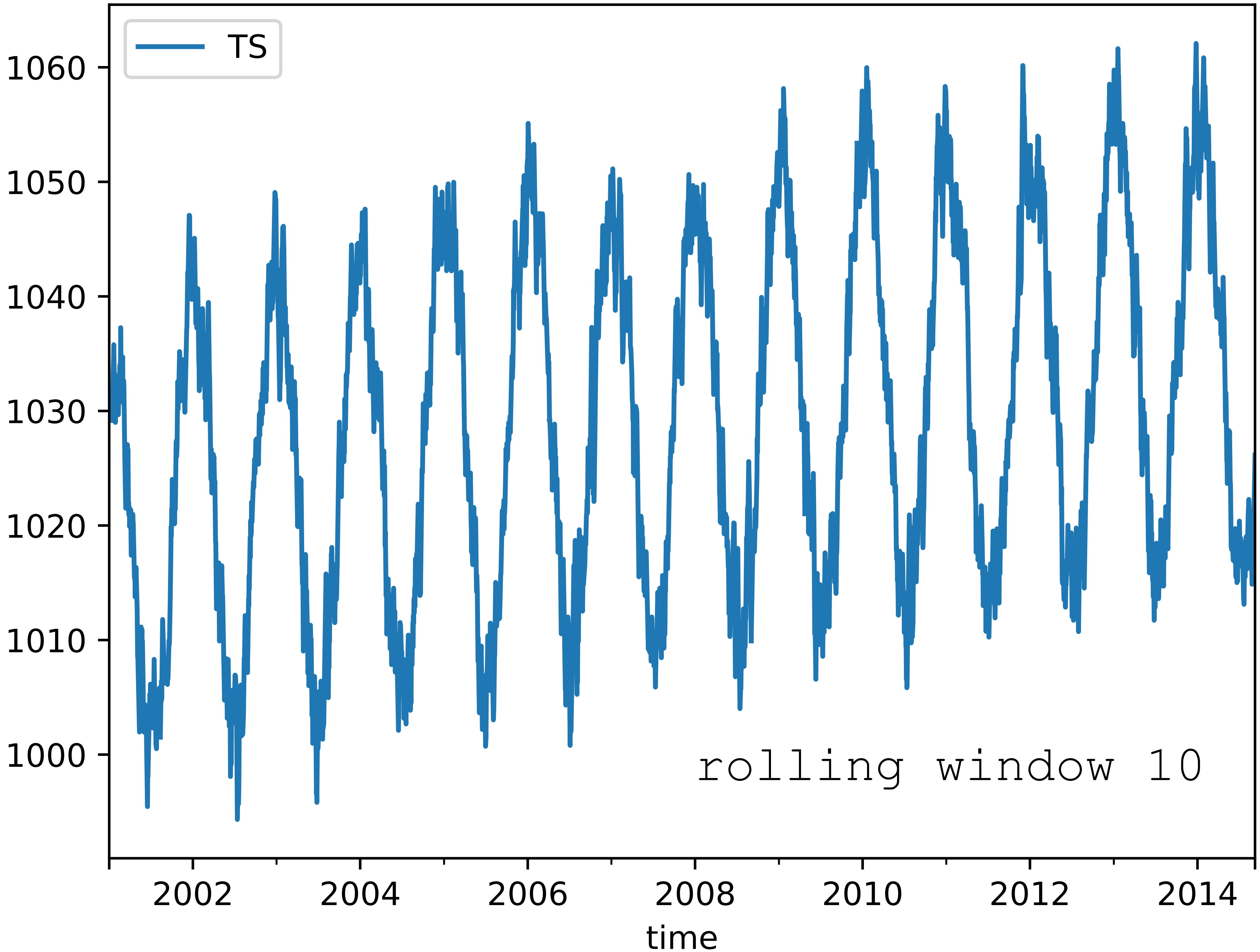
```
plt.plot_date(ts1.index, ts1.TS)
```

Basic Example

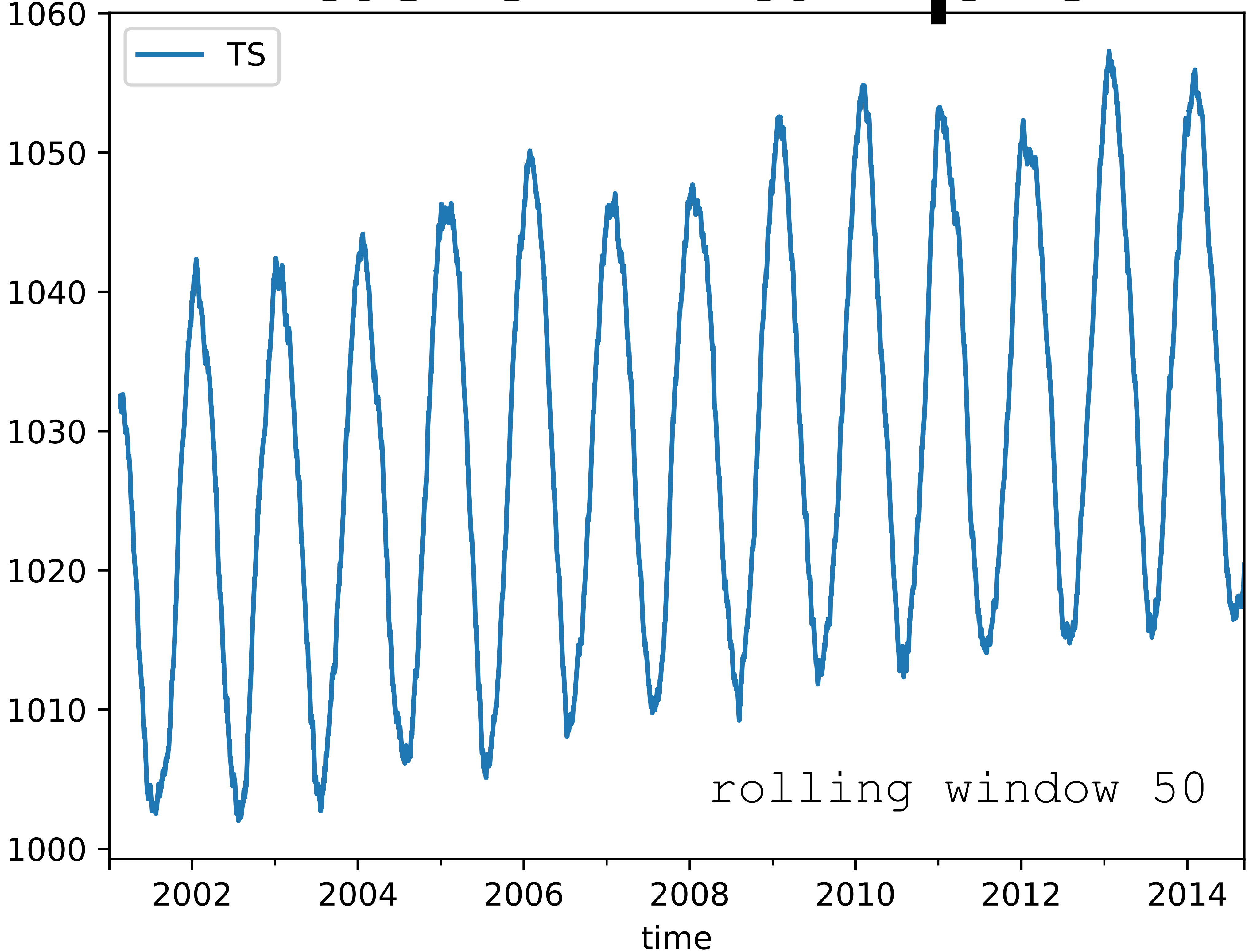
- Let's see whether we can make the line plot clearer
  - Use curve smoothing
    - Can use rolling, then mean, then plot

```
>>> ts1.rolling(10).mean().plot()
<matplotlib.axes._subplots.AxesSubplot object at
0x7fbb221e1be0>
>>> plt.show()
```

Basic Example



Basic Example

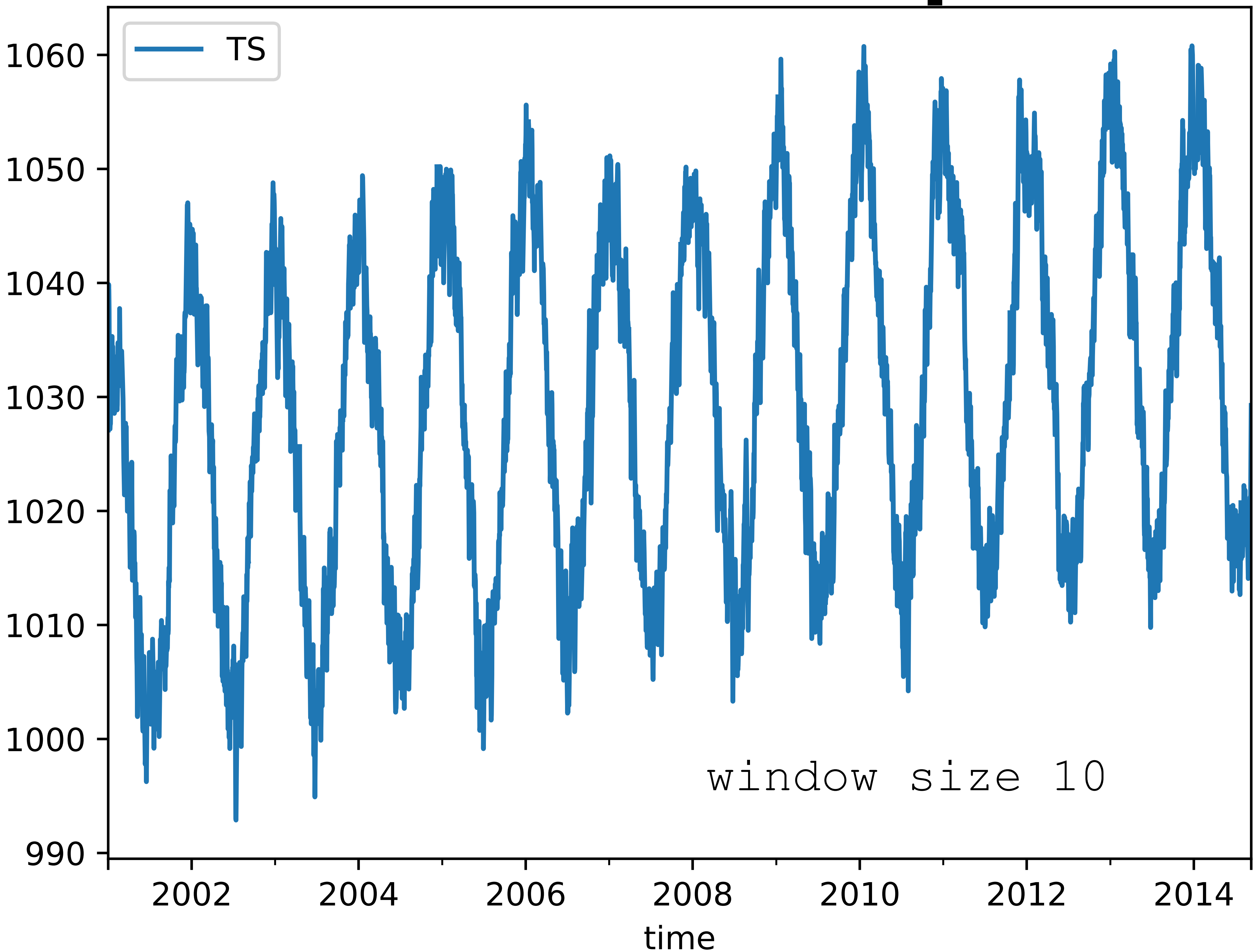


Basic Example

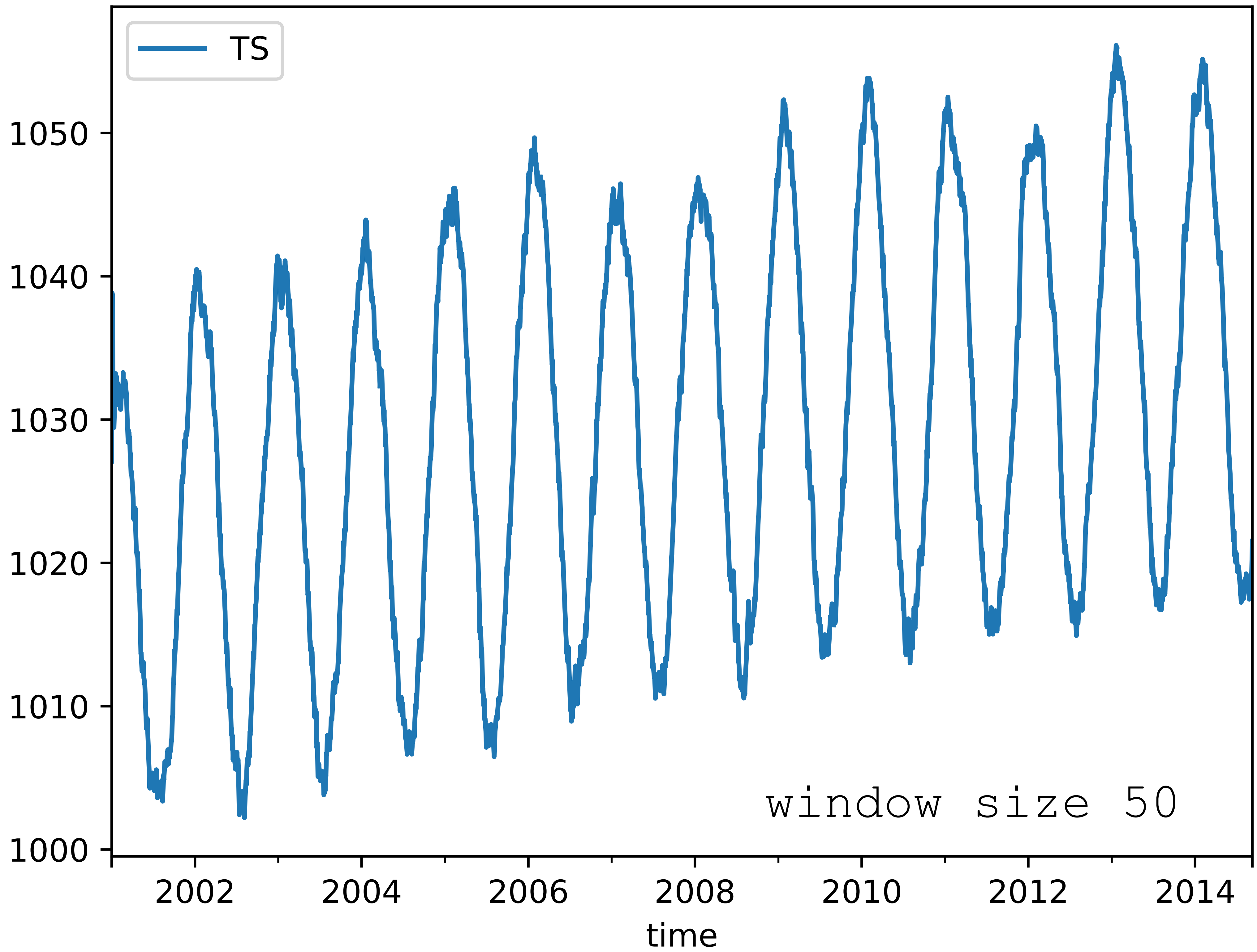
- While rolling takes all observations in the window at the same value, we can also use exponential weighted windows

```
ts1.ewm(span=10).mean().plot()
```

Basic Example



Basic Example



Fundamentals

- The basic plotting tool is still matplotlib
  - It can be wrapped by
    - Pandas
    - Seaborn ggplot
    - Holoview

Fundamentals

- Importing
  - Just as np from numpy and pd for pandas, we use traditional shortcuts

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

  - Usually only need the latter

Fundamentals

- We can pick style

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

- We can find styles with

```
plt.style.available
```

```
['seaborn-dark', 'seaborn-darkgrid', 'seaborn-ticks', 'fivethirtyeight',
'seaborn-whitegrid', 'classic', '_classic_test', 'fast', 'seaborn-talk',
'seaborn-dark-palette', 'seaborn-bright', 'seaborn-pastel', 'grayscale',
'seaborn-notebook', 'ggplot', 'seaborn-colorblind', 'seaborn-muted',
'seaborn', 'Solarize_Light2', 'seaborn-paper', 'bmh', 'tableau-colorblind10',
'seaborn-white', 'dark_background', 'seaborn-poster', 'seaborn-deep']
```

Fundamentals

- Plotting from a script
  - Use `plt.show( )`
    - Interacts with the system
      - Results are system dependent
      - plt.show( ) does a lot in the background
      - should only be run once in a script

Fundamentals

- Plotting from a notebook
    - Use `matplotlib inline`
    - Creates a new cell to embed any png created with plt.plot()

Fundamentals

- Saving figures to files
  - Use `fig.savefig(`address `)`
  - File format is inferred from file extension

Fundamentals

- Two interfaces:
  - MATLAB style interface
    - Best for relatively simple plots
    - Keeps track of all figure elements
  - Object oriented interface
    - Create figures and "axes"
    - Use method calls

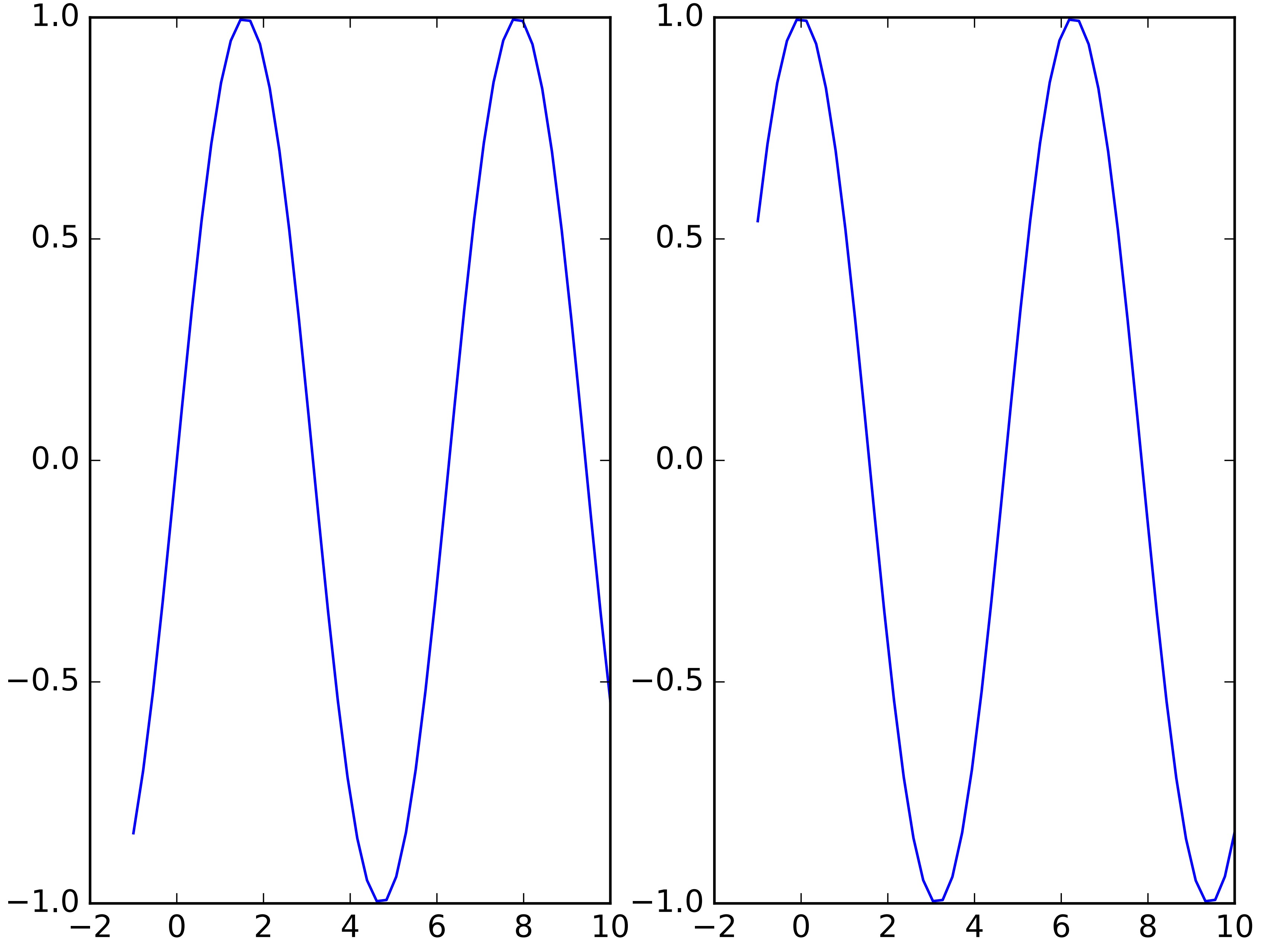
Fundamentals

- Example:
  - MATLAB interface

```
x = np.linspace(-1,10)
plt.figure( ) #create figure
plt.subplot(1,2,1)  #rows columns panel number
plt.plot(x, np.sin(x))

plt.subplot(1,2,2)
plt.plot(x, np.cos(x))
```

Fundamentals



Fundamentals

- OO interface

```
fig, ax = plt.subplots(ncols=2)
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))
```

- Figure : single container with potentially many axes
- Axes : bounding box with many elements
    - Axis, Tick, Line2D, Text, Polygon

Simple Line Plots

- Define a Figure and an Axes object
- Create an array of x values `[0., 0.01, 0.02, 0.03, 0.04, … ]`
- Create an array of y values

```
plt.style.use('seaborn-whitegrid')

fig = plt.figure
ax = plt.axes()
x = np.linspace(0,10, 1000)
ax.plot(x, (x**2+1)/(x**2+x+1))
```

Simple Line Plots

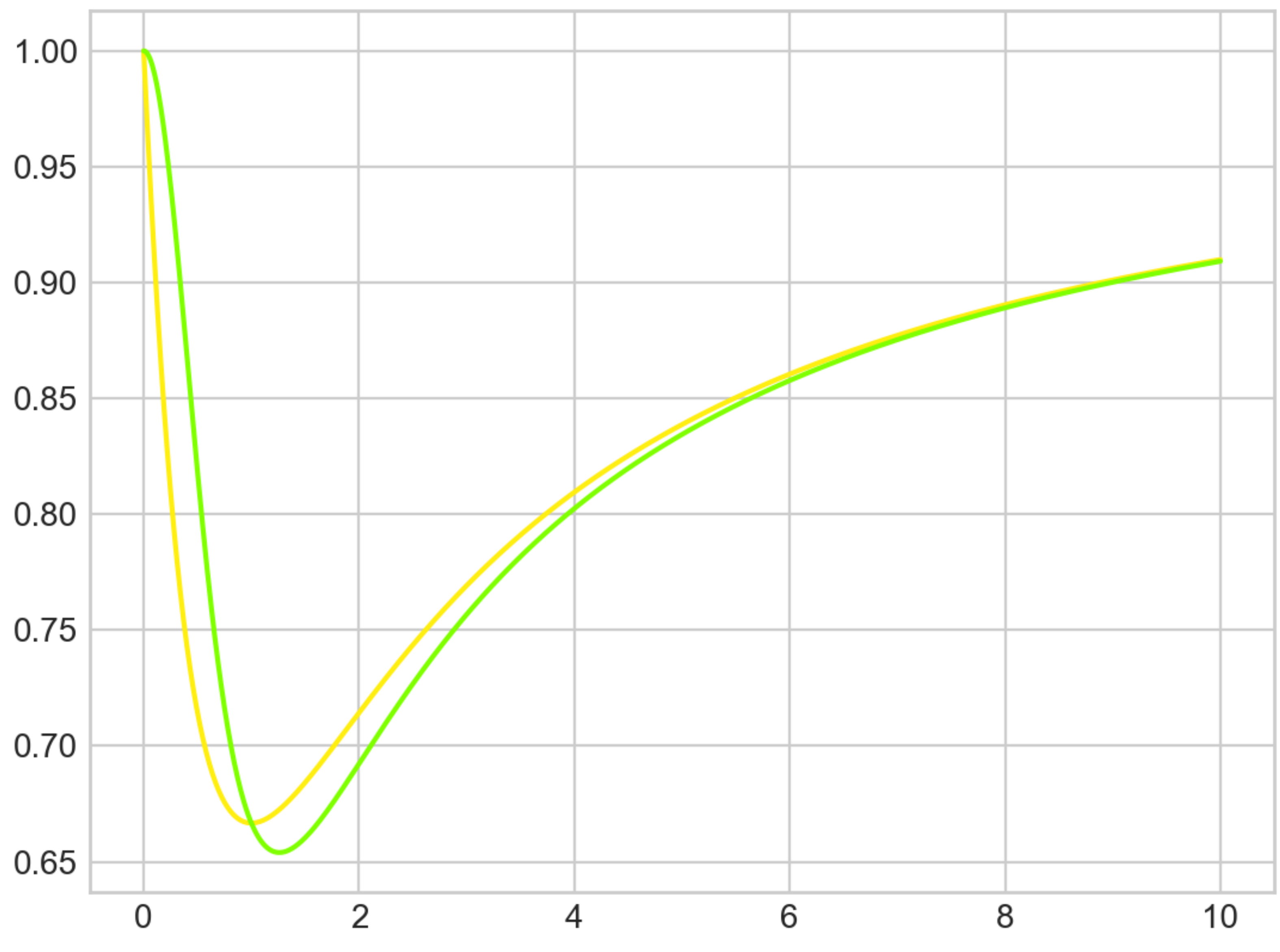
- Line colors
    - colors have
        - names,
        - abbreviations (rgbcmyk),
        - Grayscales between 0 and 1
        - Hexcodes (RRGGBB) between 00 and FF
        - RGB tuples with values between 0 and 1
        - HTML color names

Simple Line Plots

```
x = np.linspace(0,10, 1001)
ax.plot(x, (x**2+1)/(x**2+x+1), color = '#FFEE11')
```

Simple Line Plots

```
x = np.linspace(0,10, 1001)
ax.plot(x, (x**2+1)/(x**2+x+1), color = '#FFEE11')
ax.plot(x, (x**3+1)/(x**3+x**2+1), color = 'chartreuse')
```

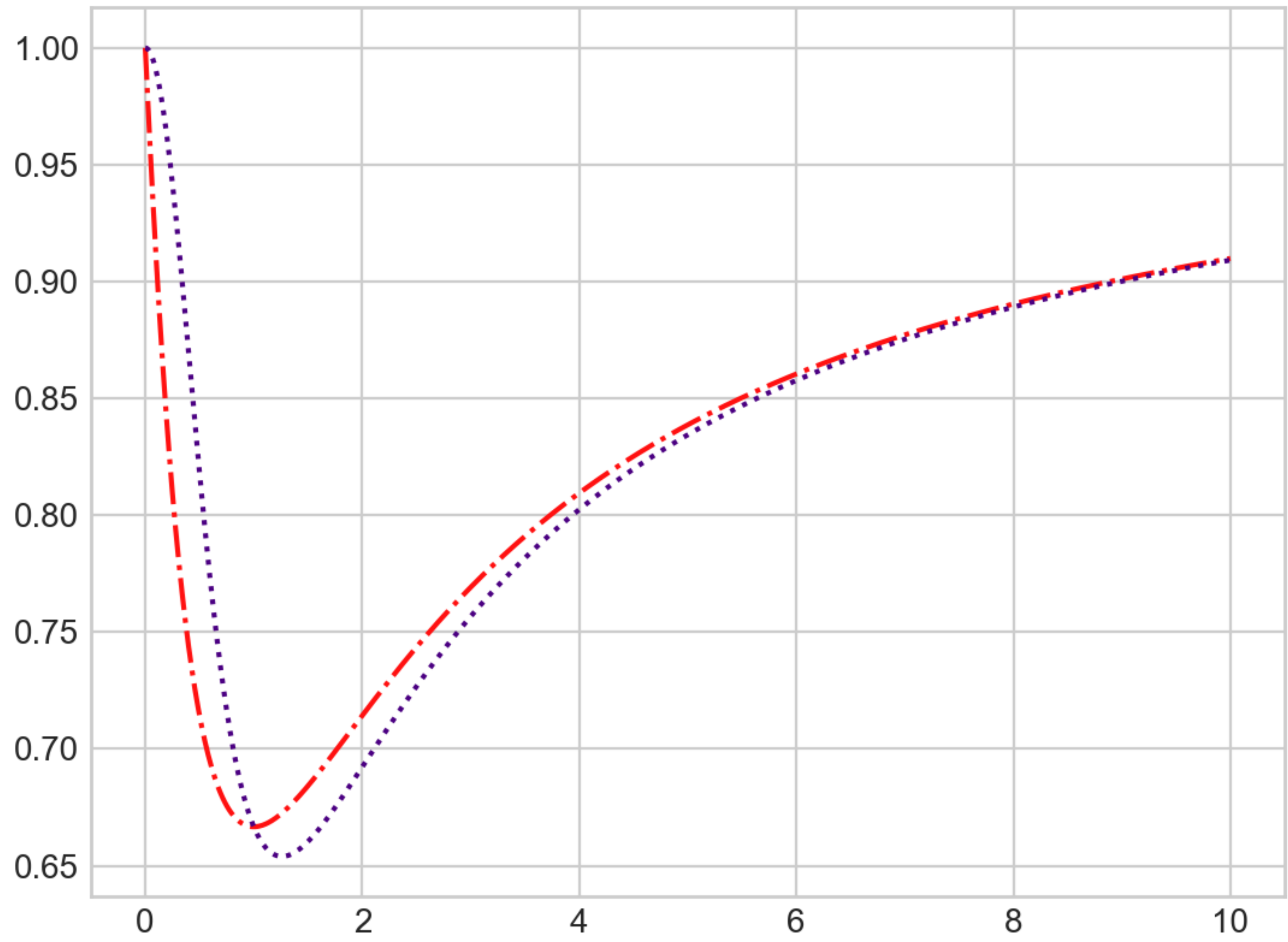


Simple Line Plots

- Line Styles
  - 'solid', 'dashed', 'dashdot' 'dotted'
- Abbreviated as
  - '-', '- -', '-.', ':'

Simple Line Plots

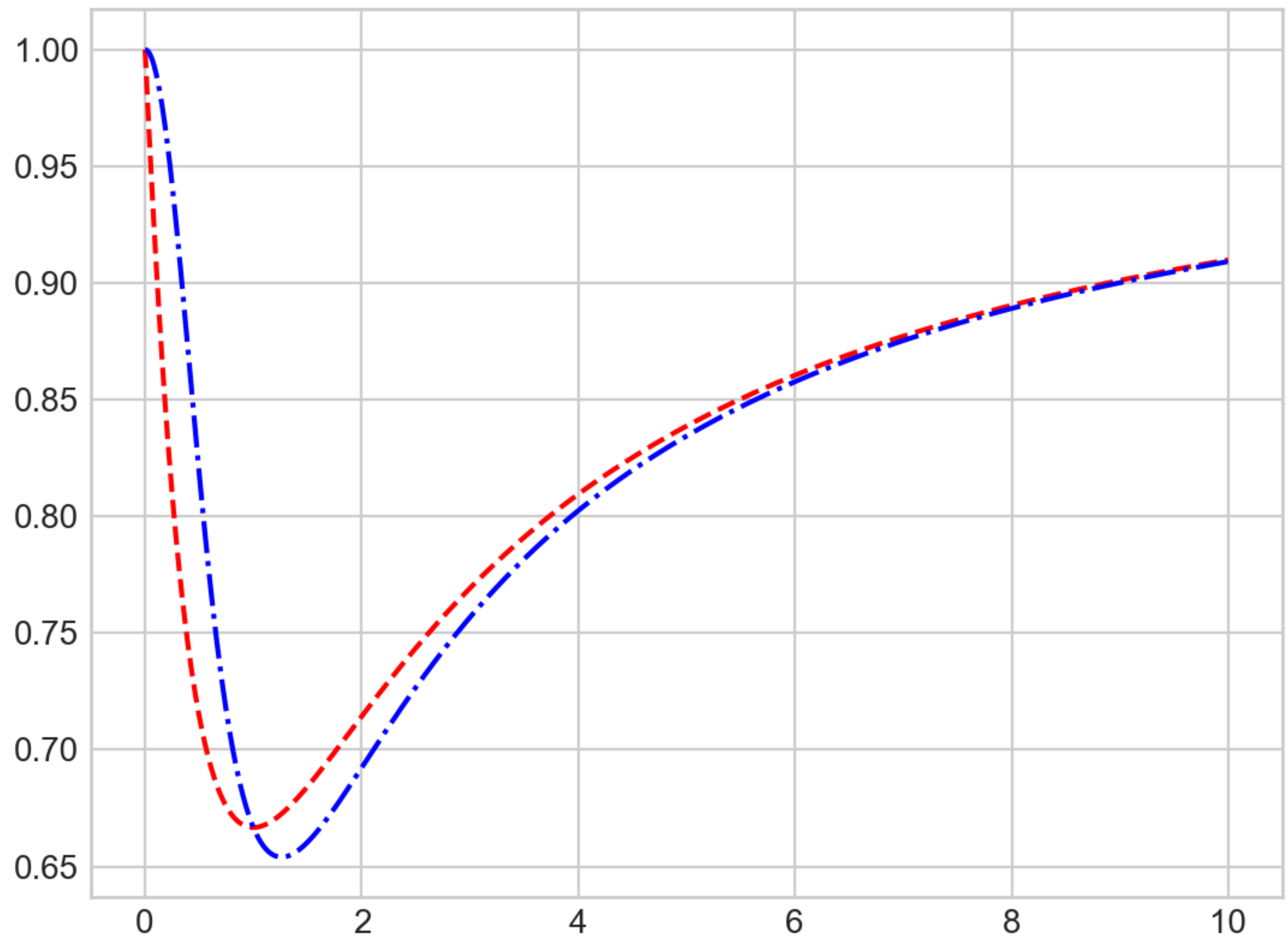
```
ax.plot(x, (x**2+1)/(x**2+x+1), color = '#FF1111', 
linestyle='dashdot')
ax.plot(x, (x**3+1)/(x**3+x**2+1), color = 'indigo', 
linestyle = ':')
```



Simple Line Plots

- These can also be combined

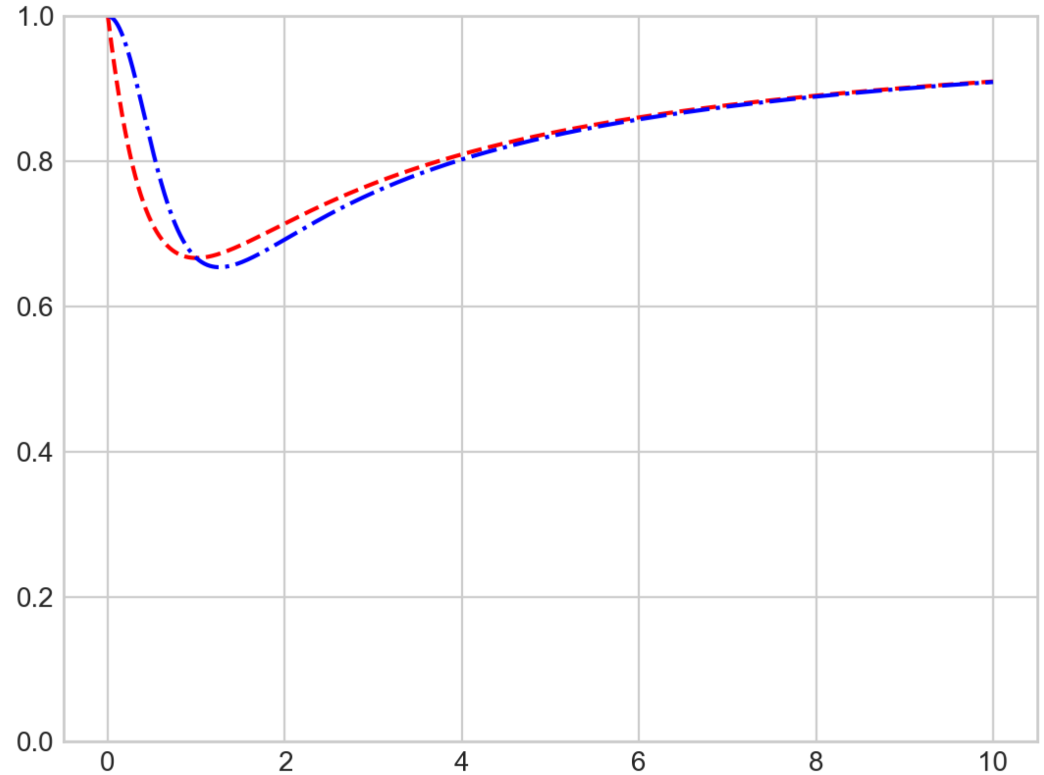
```
ax.plot(x, (x**2+1)/(x**2+x+1), 'r--')
ax.plot(x, (x**3+1)/(x**3+x**2+1), 'b-.')
```



Simple Line Plots

- Axes Limits for finer control
  - set xlim, ylim

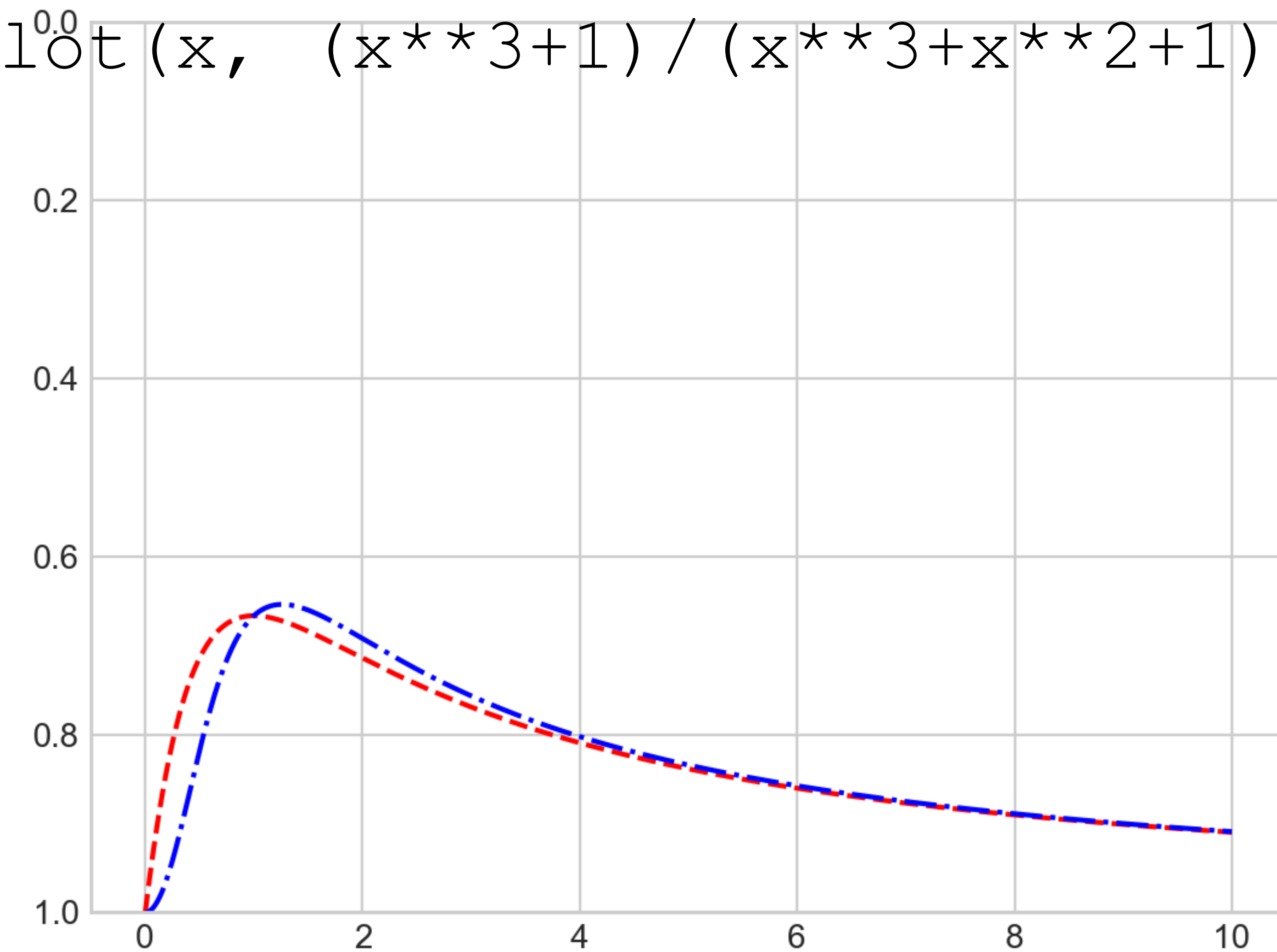
```
plt.ylim(0,1)
ax.plot(x, (x**2+1)/(x**2+x+1), 'r--')
ax.plot(x, (x**3+1)/(x**3+x**2+1), 'b-.')
```



Simple Line Plots

- You can even revert an axis

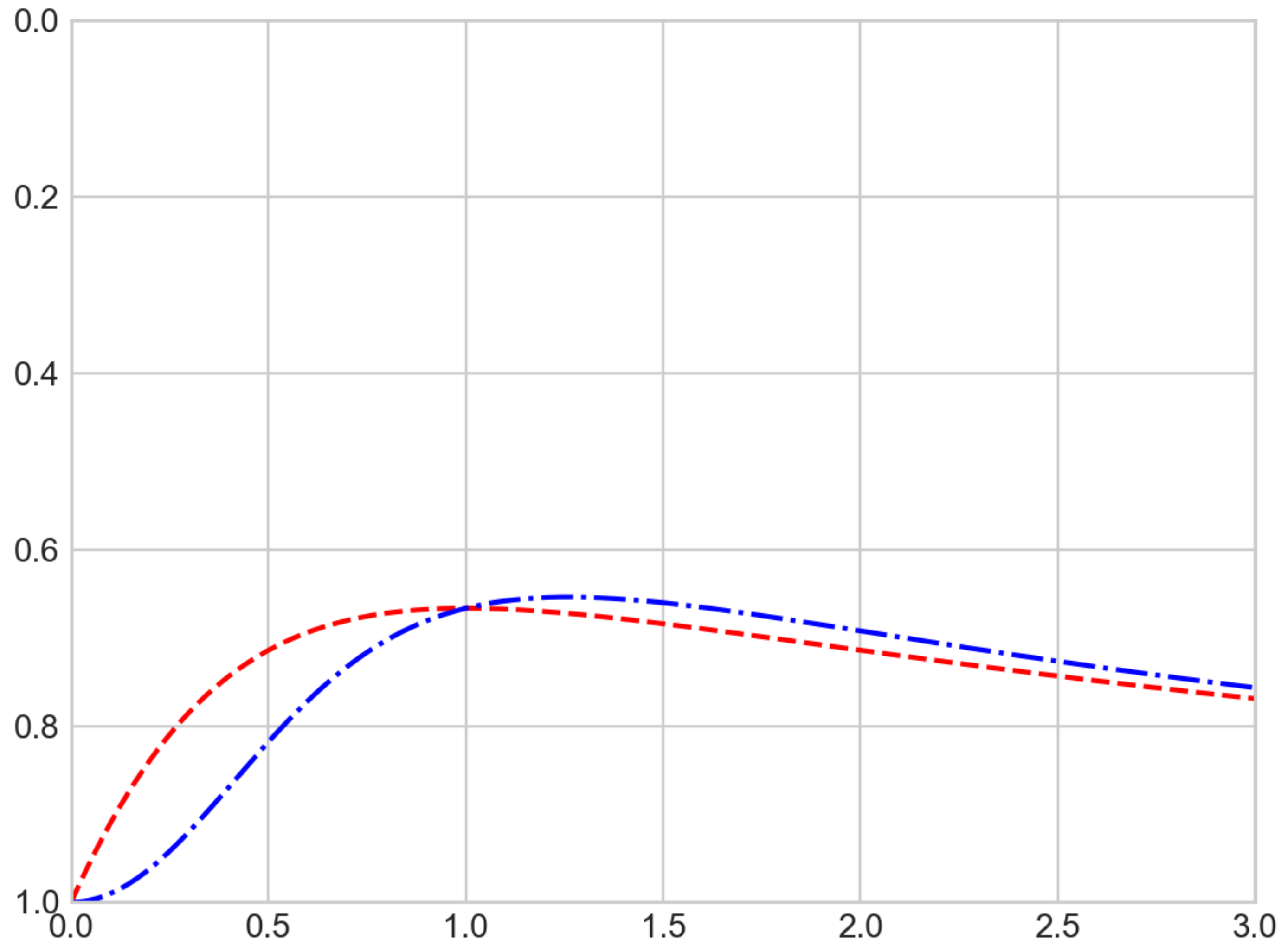
```
plt.ylim(1,0)
ax.plot(x, (x**2+1)/(x**2+x+1), 'r--')
ax.plot(x, (x**3+1)/(x**3+x**2+1), 'b-.')
```



Simple Line Plots

- You can set all axes with the confusingly named axis method

```
plt.axis([0,3,1,0])
```



Simple Line Plots

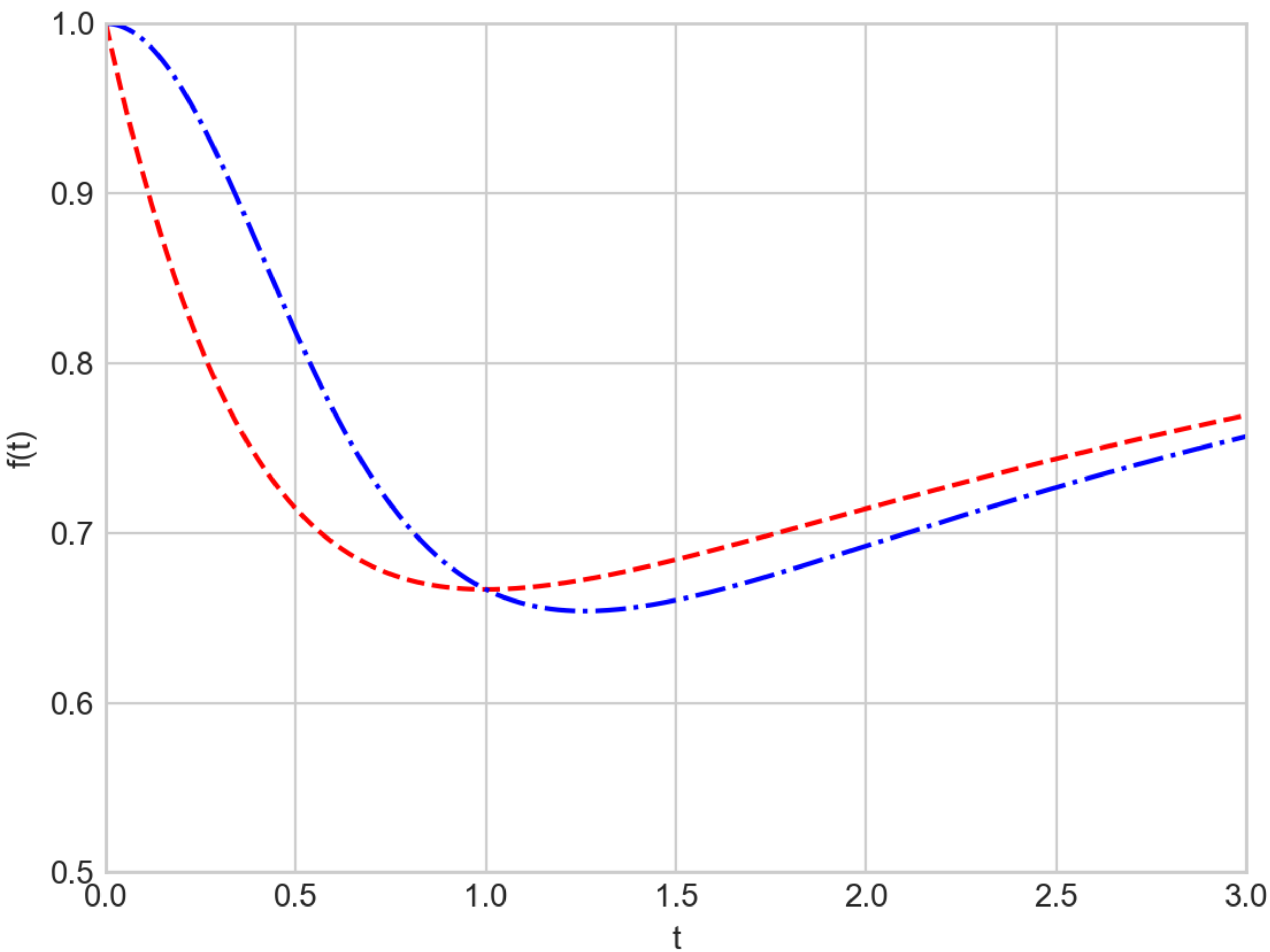
- plt.axis actually allow even more plot control
  - 'tight' to tighten bounds around current plot
  - ''equal' for equal aspect ratio

Simple Line Plots

- Plot axes can be labeled

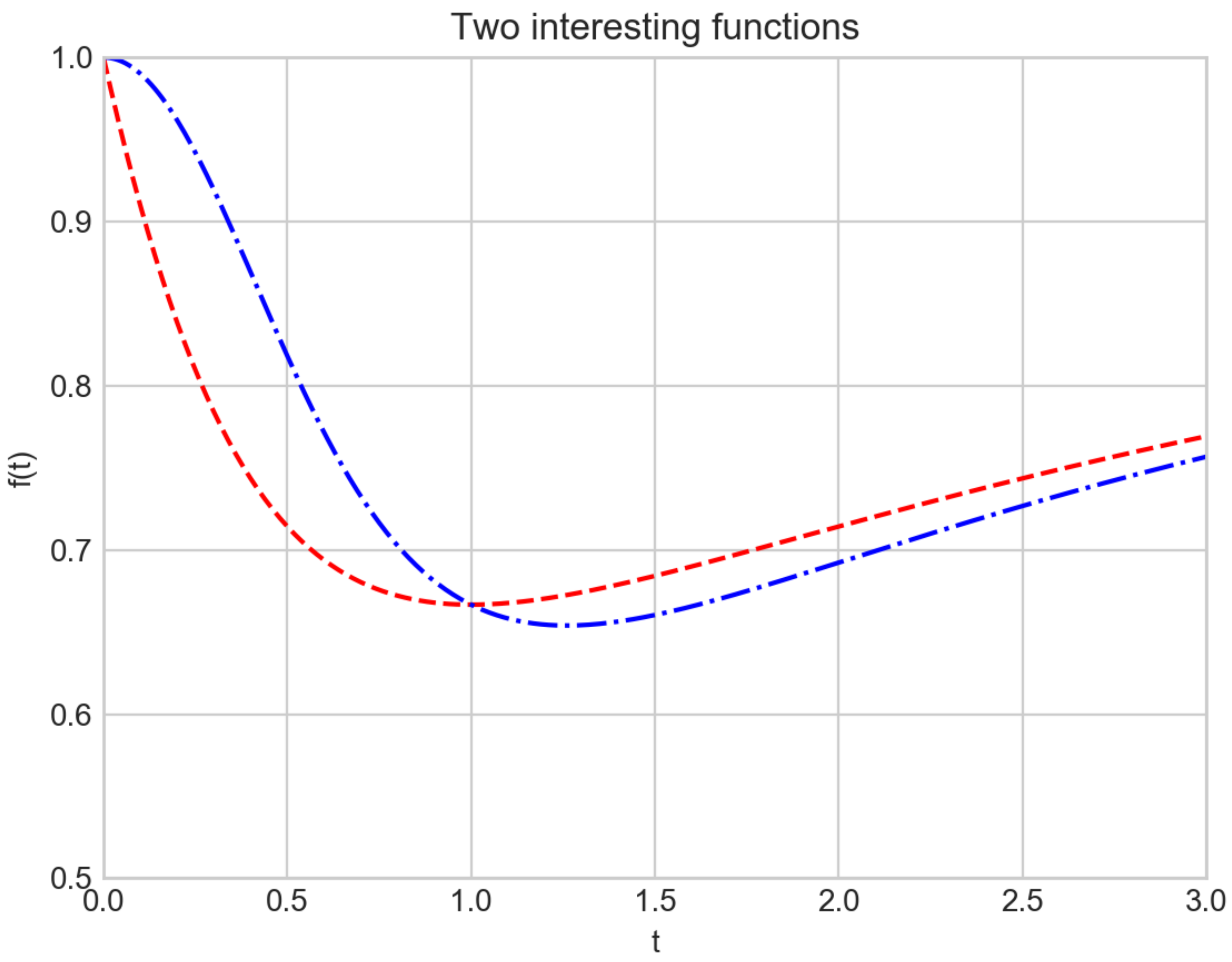
  - ```
    fig = plt.figure
    ax = plt.axes()
    x = np.linspace(0,10, 1001)
    plt.axis([0,3,0.5,1], 'tight')
    plt.xlabel('t')
    plt.ylabel('f(t)')
    ax.plot(x, (x**2+1)/(x**2+x+1), 'r--')
    ax.plot(x, (x**3+1)/(x**3+x**2+1), 'b-.')
    ```
    ```

Simple Line Plots



Simple Line Plots

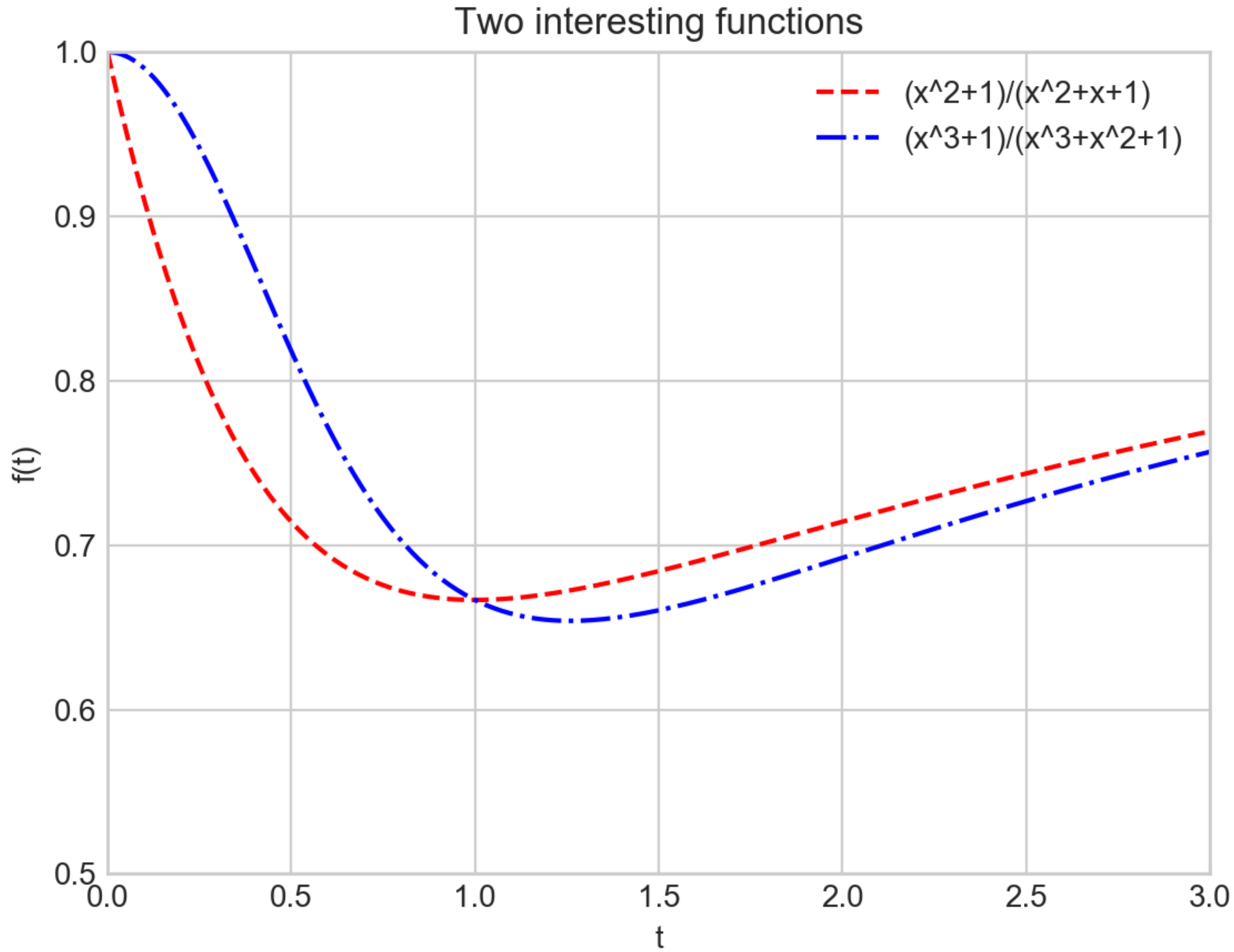
- We label a plot with plt.title



Simple Line Plots

- And we can provide a legend

```
ax.plot(x, (x**2+1)/(x**2+x+1), 'r--', label='(x^2+1)/(x^2+x+1)')
ax.plot(x, (x**3+1)/(x**3+x**2+1), 'b-.', label='(x^3+1)/(x^3+x^2+1)')
plt.legend()
```



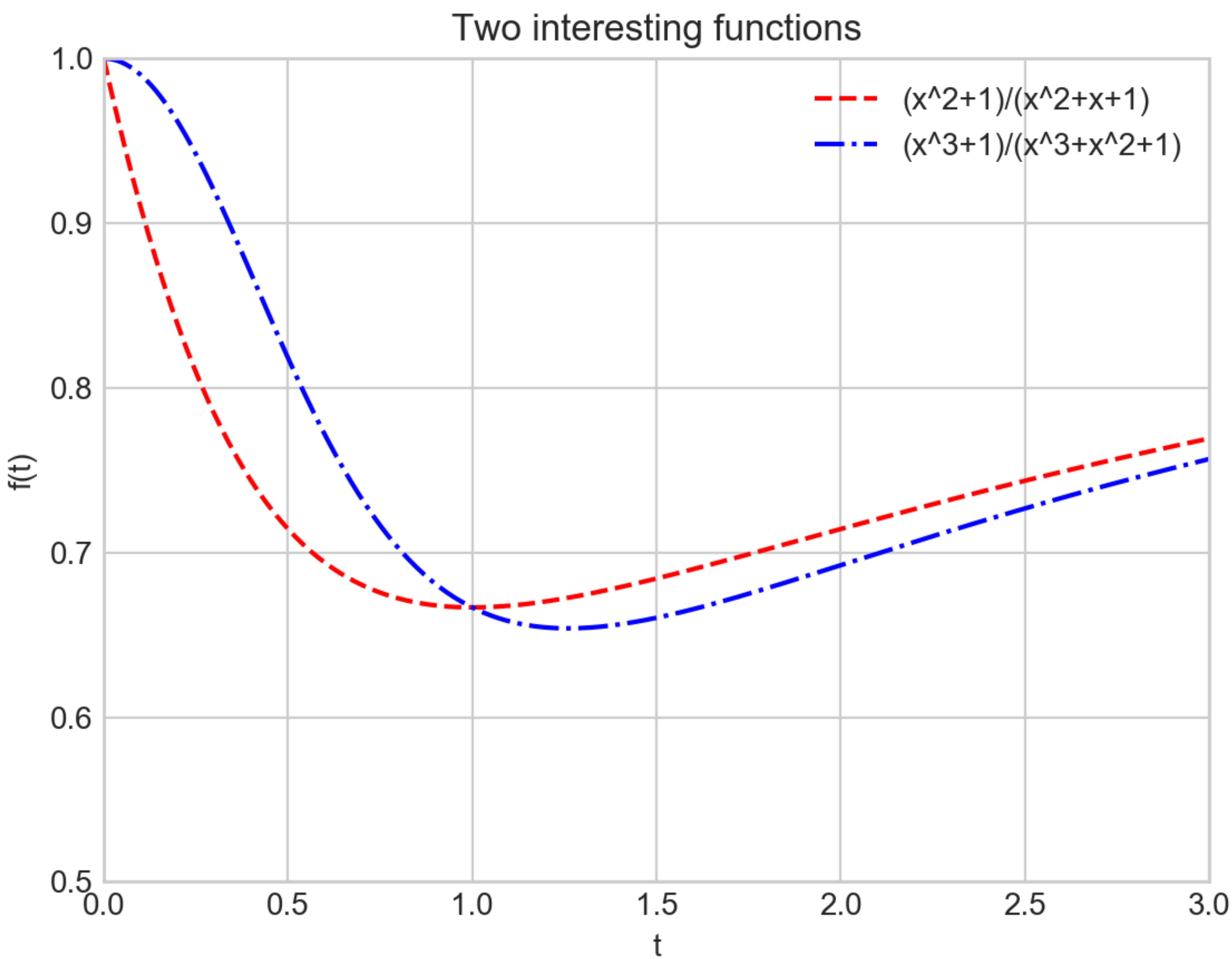
Simple Line Plots

- OO translation:
    - `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( )`
- or just use ax.set

Simple Line Plots

```
ax.plot(x, (x**2+1)/(x**2+x+1), 'r--', label='(x^2+1)/
(x^2+x+1)')
ax.plot(x, (x**3+1)/(x**3+x**2+1), 'b-.',
label='(x^3+1)/(x^3+x^2+1)')
ax.set(xlim=(0,1.5), ylim=(0.6, 1), xlabel='x',
ylabel='f(y)',
       title = 'Two functions')
```

Simple Line Plots

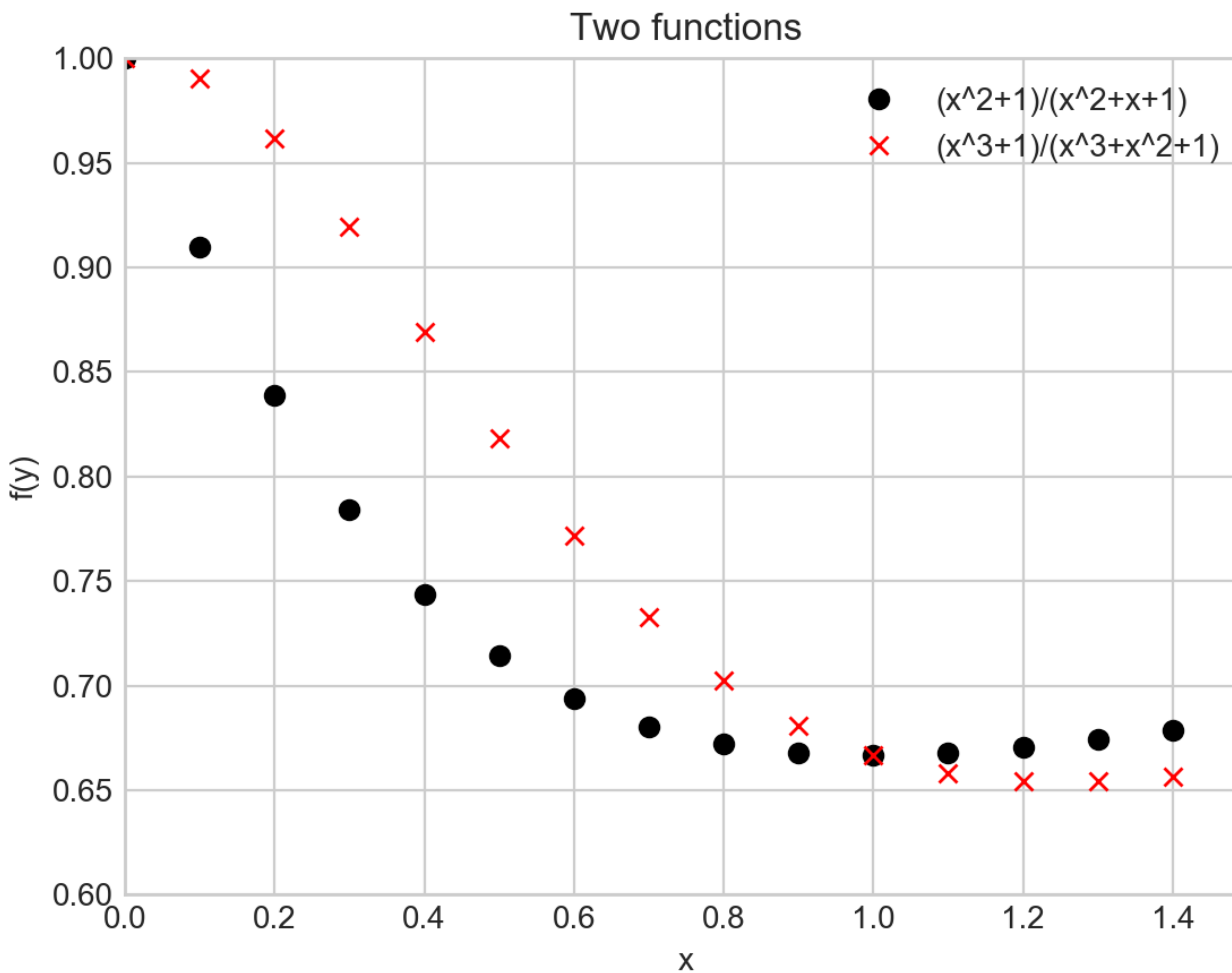


Simple Scatter Plots

- plt.plot / ax.plot can also produce scatter plots
- Just give it a marker

```
ax.plot(x, (x**2+1)/(x**2+x+1), 'o', color = 'black',
label='(x^2+1)/(x^2+x+1)')
ax.plot(x, (x**3+1)/(x**3+x**2+1), 'x', color = 'red',
label='(x^3+1)/(x^3+x^2+1)')
ax.set(xlim=(0,1.5), ylim=(0.6, 1), xlabel='x',
ylabel='f(y)', title = 'Two functions')
ax.legend()
```

Simple Scatter Plots



Simple Scatter Plots

- Additional argument represent the symbol
  - 'o', '.', 'x', '+', 'v', '^', '<', '>',
    - 's' square
    - 'd' diamond

Simple Scatter Plots

- More powerful: Use `plt.scatter`
    - Can control many more aspects

- Example:
    - Create 100 random pairs of x, y
    - Create random colors (between 0 and 1)
    - Create random sizes (between 0 and 1000)
    - Set alpha = 0.3 in order to make things transparent

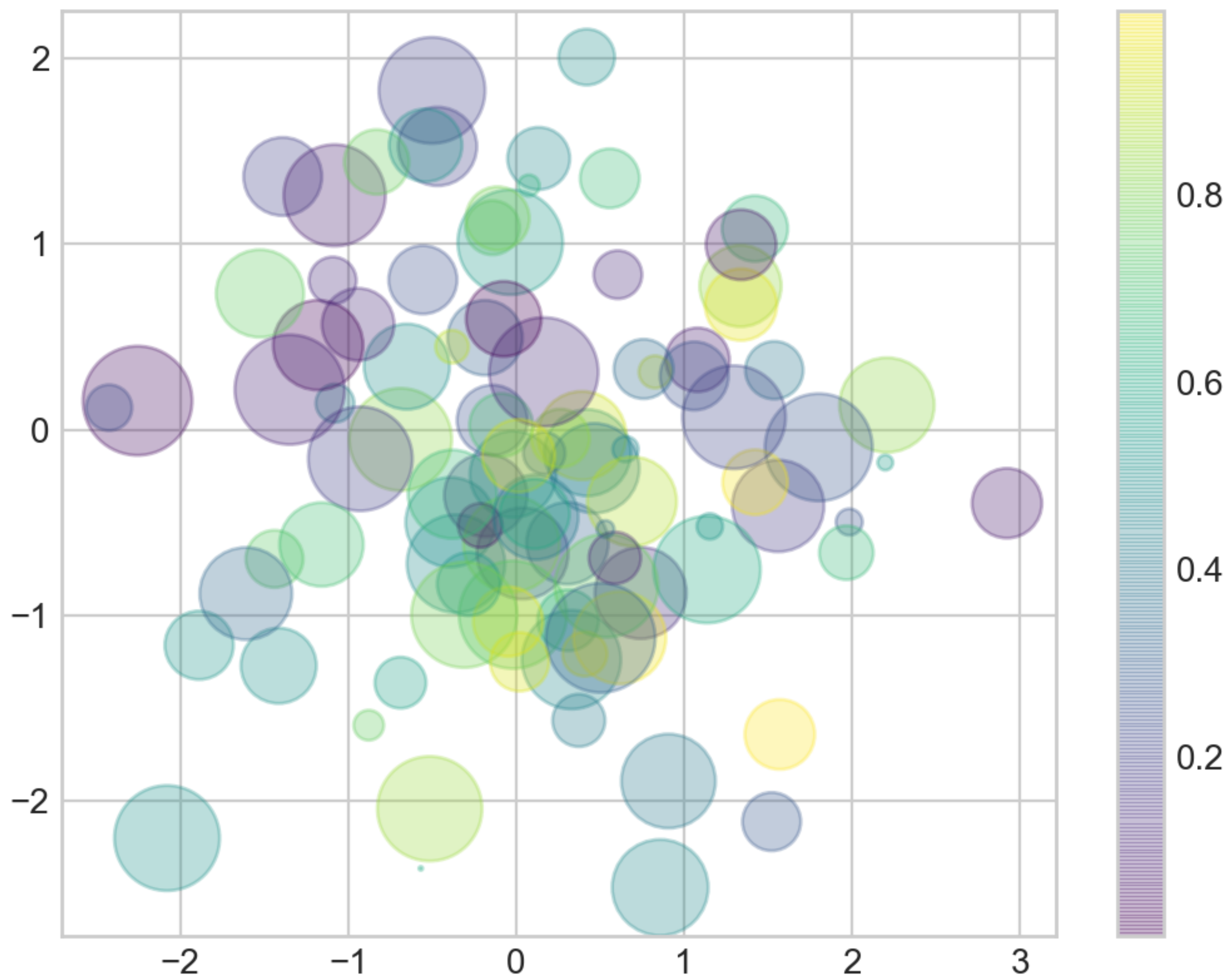
Simple Scatter Plots

```
np.random.seed(250620)
x = np.random.randn(100)
y = np.random.randn(100)
colors = np.random.rand(100)
sizes = 1000 * np.random.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha = 0.3,
            cmap = 'viridis')
plt.colorbar()  # show color scale

plt.show()
```

Simple Scatter Plots



Simple Scatter Plots

- Example: Iris data set (from sklearn.datasets)

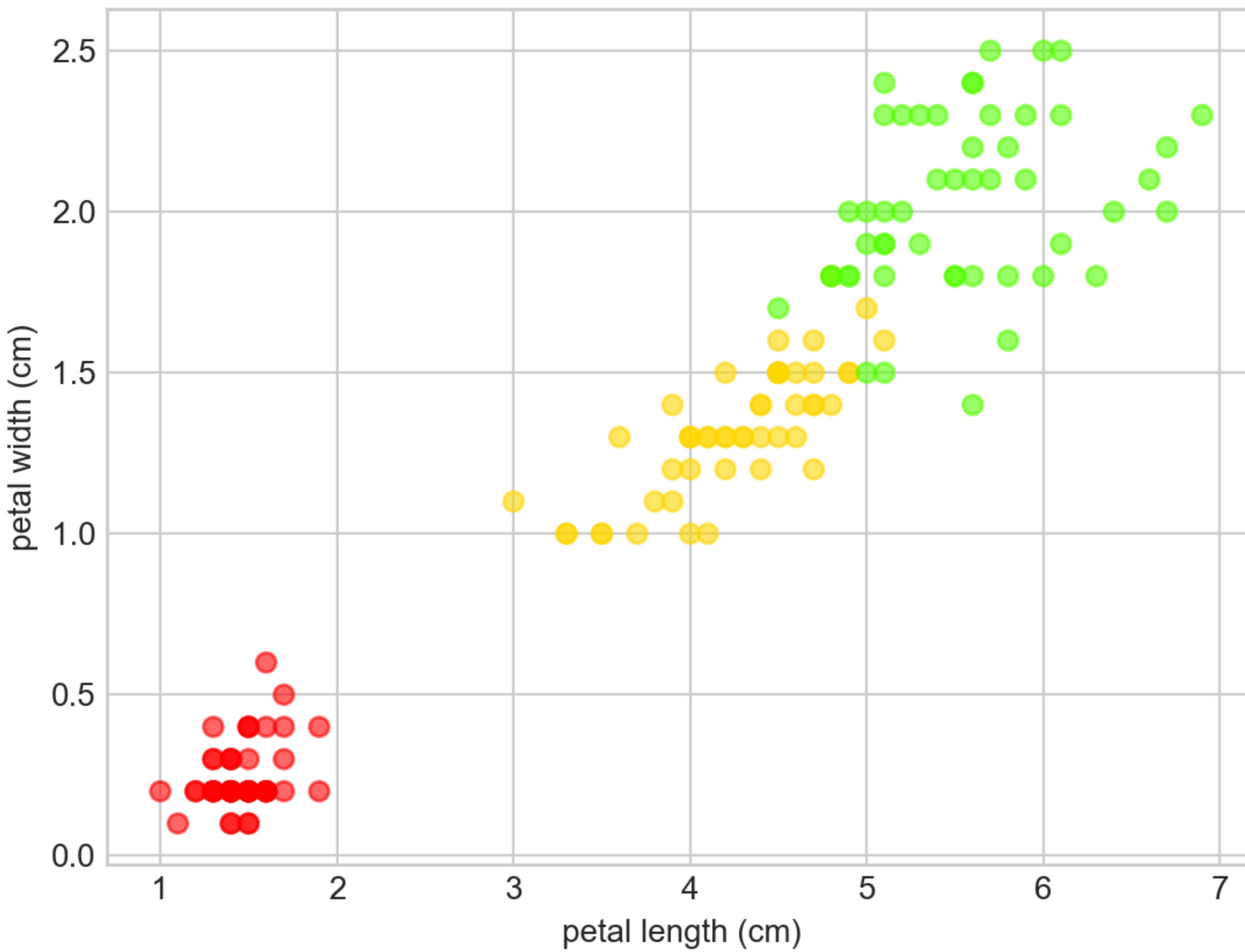
```
from sklearn.datasets import load_iris

iris = load_iris()
features = iris.data.T  #need to transpose

plt.scatter(features[2], features[3], alpha = 0.6,
            c=iris.target, cmap = 'prism')
plt.xlabel(iris.feature_names[2])
plt.ylabel(iris.feature_names[3])

plt.show()
```

Simple Scatter Plots



Simple Scatter Plots

- As datasets get larger, plot becomes more efficient than scatter

Error Bars

- "it's not science if there is no statistics" thomas schwarz, sj
  - "it's not statistics if there are no errorbars" thomas schwarz, sj
- When you have a data set, you should also have a confidence interval
  - This is displayed by an error bar
  - Use `plt.errorbar` with
    - x-values
    - y-values
    - confidence interval size
    - format code to control appearances (same as for lines and colors)

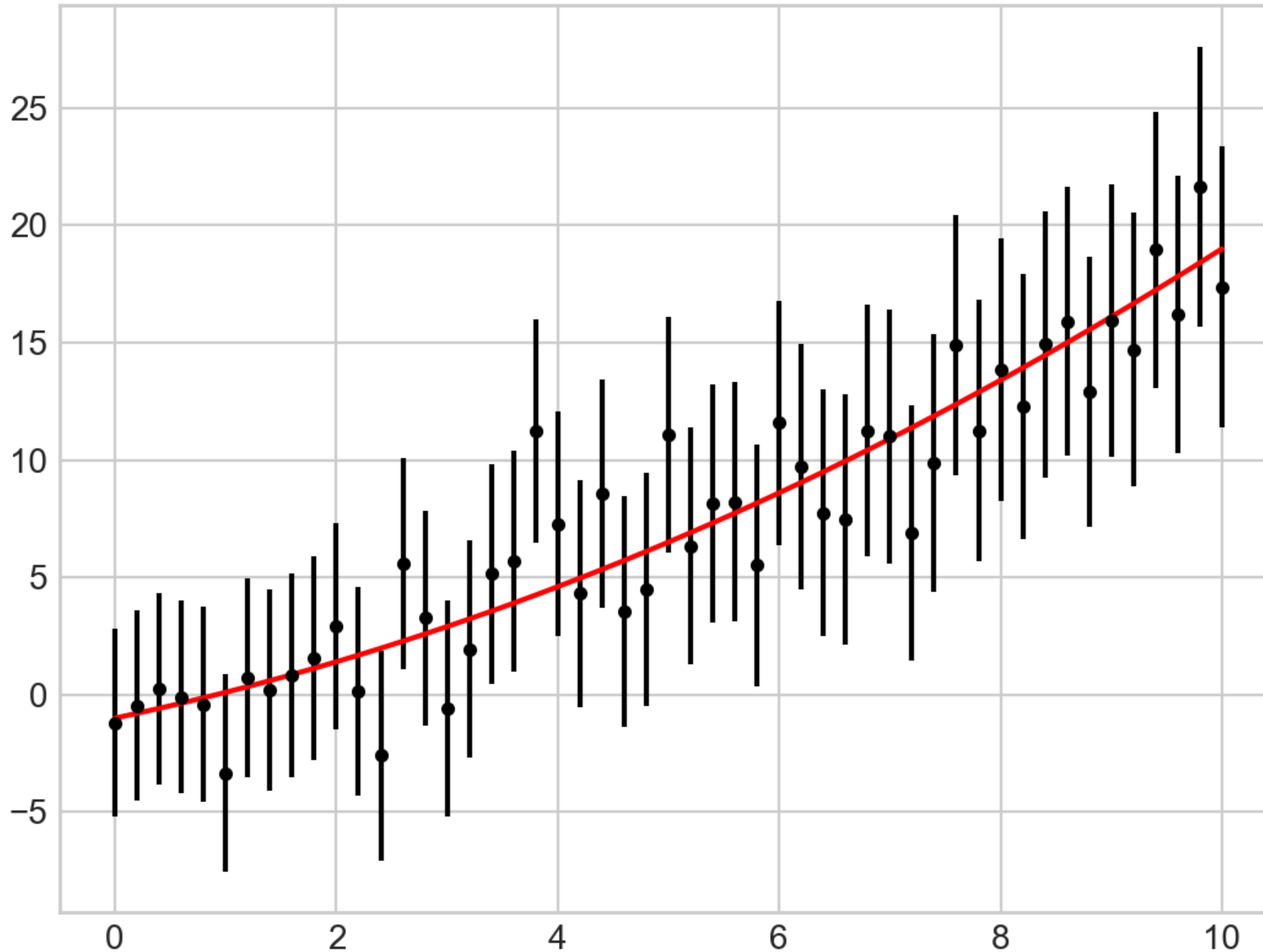
Error Bars

- Example:
  - Use a simple function
  - Use random.normal in order to generate y-values centered around the random function
  - Then draw error bars with $y \pm 2\sigma$

Error Bars

```
x = np.linspace(0,10,51)
y = np.random.normal(loc = x**2/10+x-1 , scale = 2+x/10)
plt.errorbar(x,y, yerr=2*(2+x/10), fmt='.k')
plt.plot(x, x**2/10+x-1, 'r-')
plt.show()
```

Error Bars



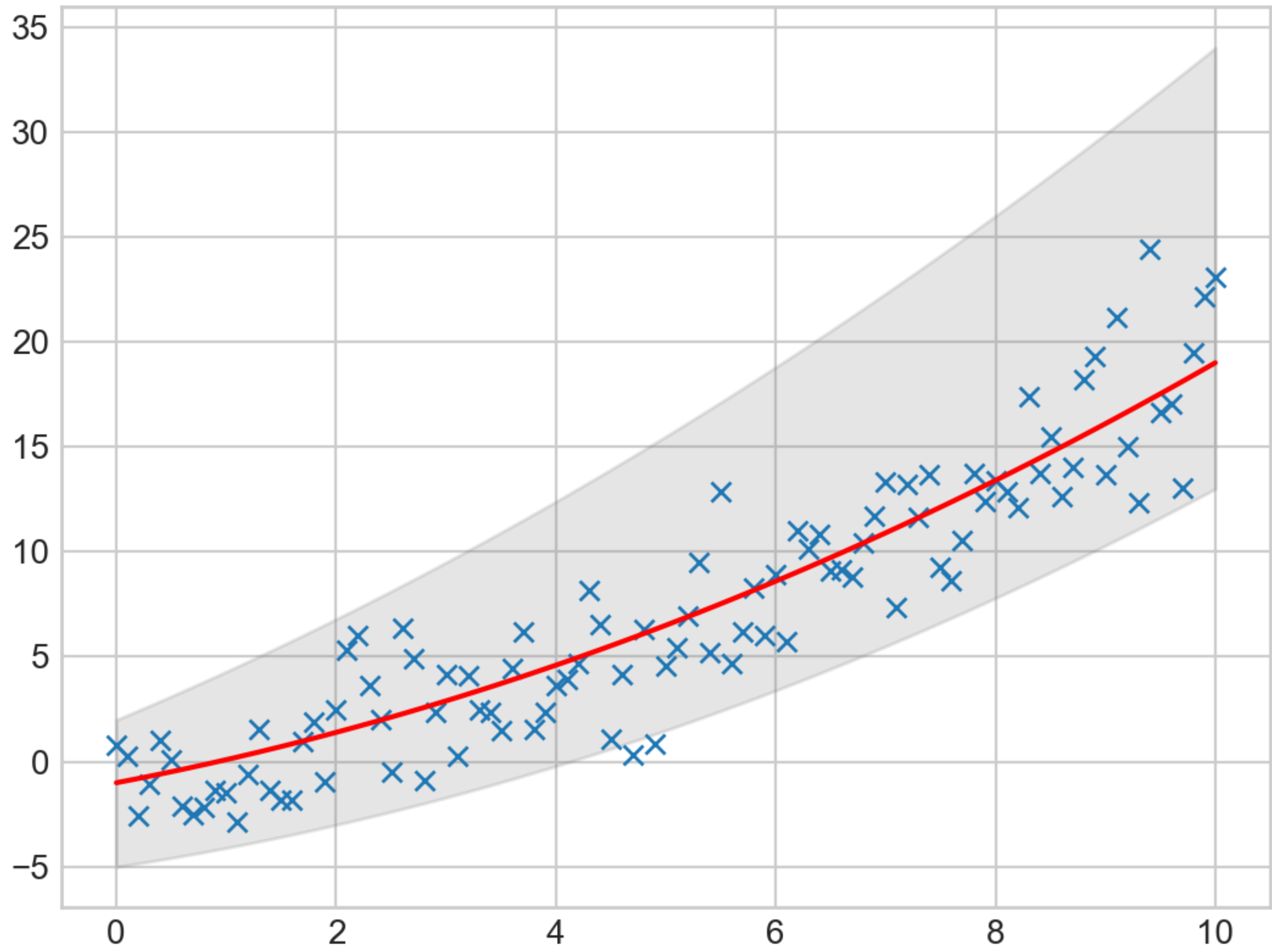
Error Bars

- Continuous errors
  - No real support in matplotlib
  - Can make it ourselves with filling between curves

```
x = np.linspace(0,10,101)
y = np.random.normal(loc = x**2/10+x-1 , scale = 2+x/10)
plt.plot(x,y, 'x')
plt.plot(x, x**2/10+x-1, 'r-')
plt.fill_between(x, x**2/10+x-1-2*(2+x/10),
x**2/10+2*(x-1+2+x/10), color = 'gray', alpha=0.2)

plt.show()
```

Error Bars



Contour / Density Plots

- As an example, use the following function

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

Contour / Density Plots

- Contour plot
  - Need to create a grid
    - Easiest with meshgrid

```
x = np.linspace(-5,5,101)
y = np.linspace(-5,5,101)
X,Y = np.meshgrid(x,y)
Z = f(X,Y)
```

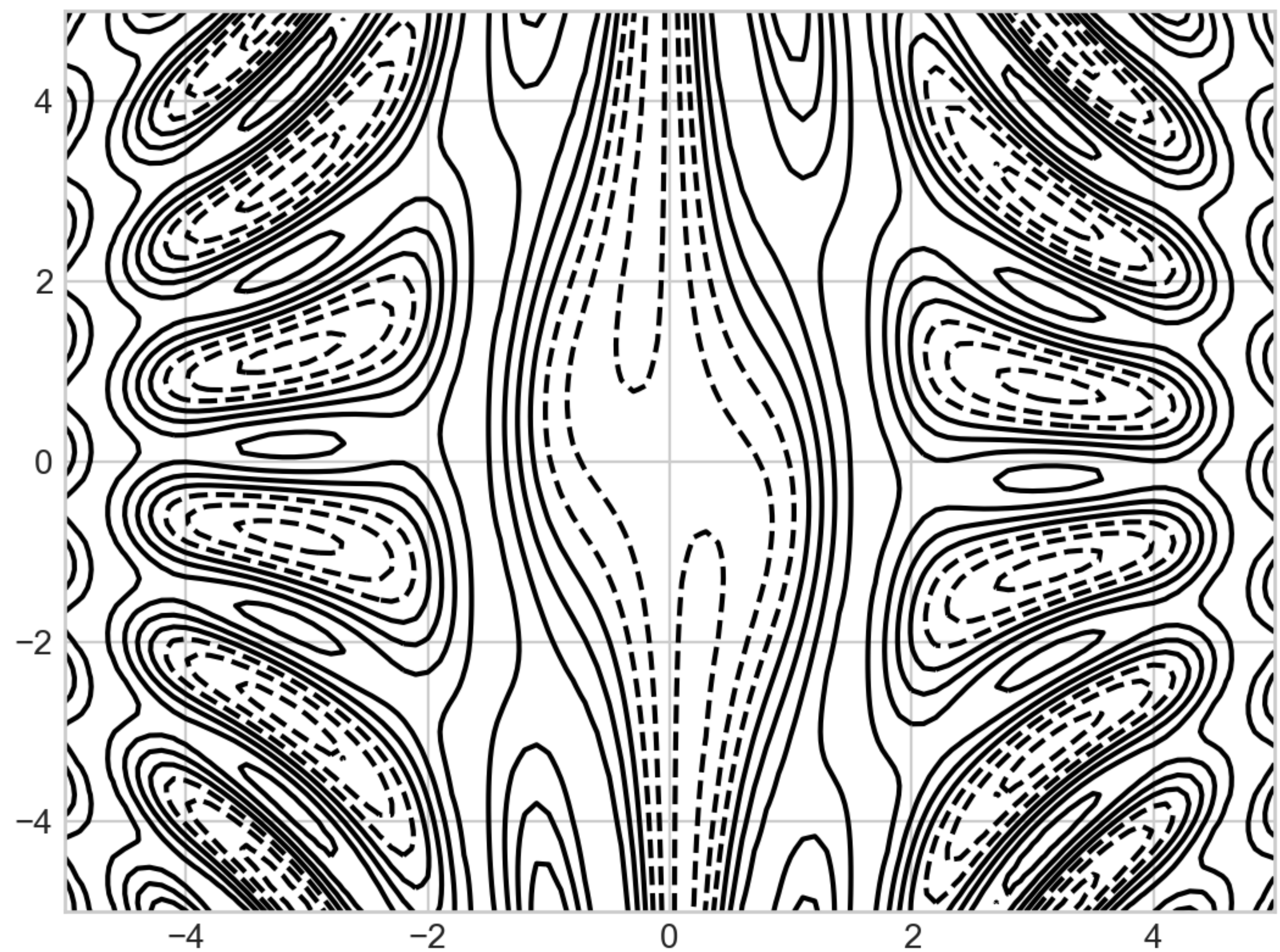
Contour / Density Plots

```
>>> X
array([[-5. , -4.9, -4.8, ...,  4.8,  4.9,  5. ],
       [-5. , -4.9, -4.8, ...,  4.8,  4.9,  5. ],
       [-5. , -4.9, -4.8, ...,  4.8,  4.9,  5. ],
       ...,
       [-5. , -4.9, -4.8, ...,  4.8,  4.9,  5. ],
       [-5. , -4.9, -4.8, ...,  4.8,  4.9,  5. ],
       [-5. , -4.9, -4.8, ...,  4.8,  4.9,  5. ]])
>>> Y
array([[-5. , -5. , -5. , ..., -5. , -5. , -5. ],
       [-4.9, -4.9, -4.9, ..., -4.9, -4.9, -4.9],
       [-4.8, -4.8, -4.8, ..., -4.8, -4.8, -4.8],
       ...,
       [ 4.8,  4.8,  4.8, ...,  4.8,  4.8,  4.8],
       [ 4.9,  4.9,  4.9, ...,  4.9,  4.9,  4.9],
       [ 5. ,  5. ,  5. , ...,  5. ,  5. ,  5. ]])
```

Contour / Density Plots

- Simple contour plot: dashed lines stand for neg. values

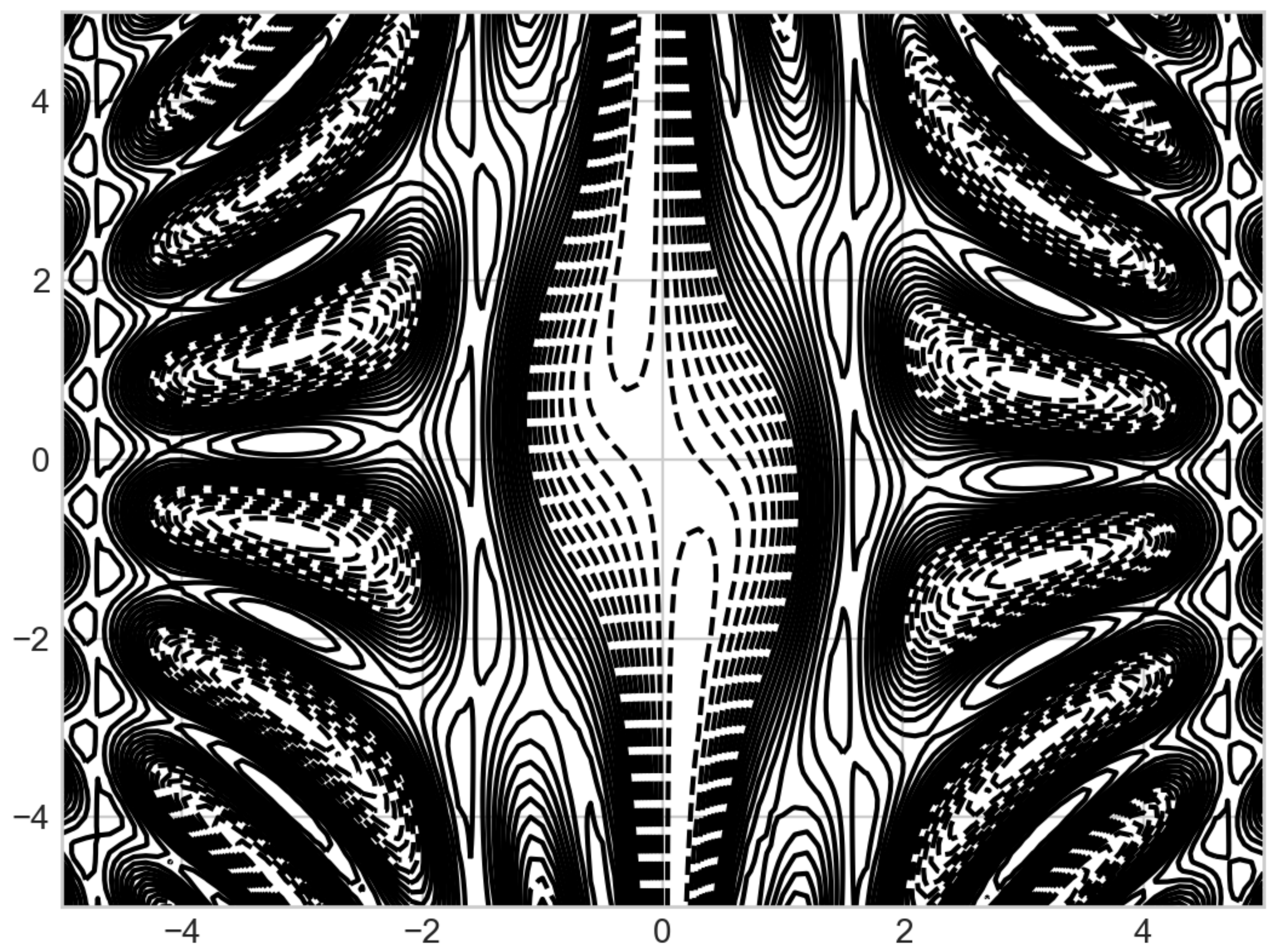
```
plt.contour(X,Y,Z, colors='black')
```



Contour / Density Plots

- Can specify the number of contour lines

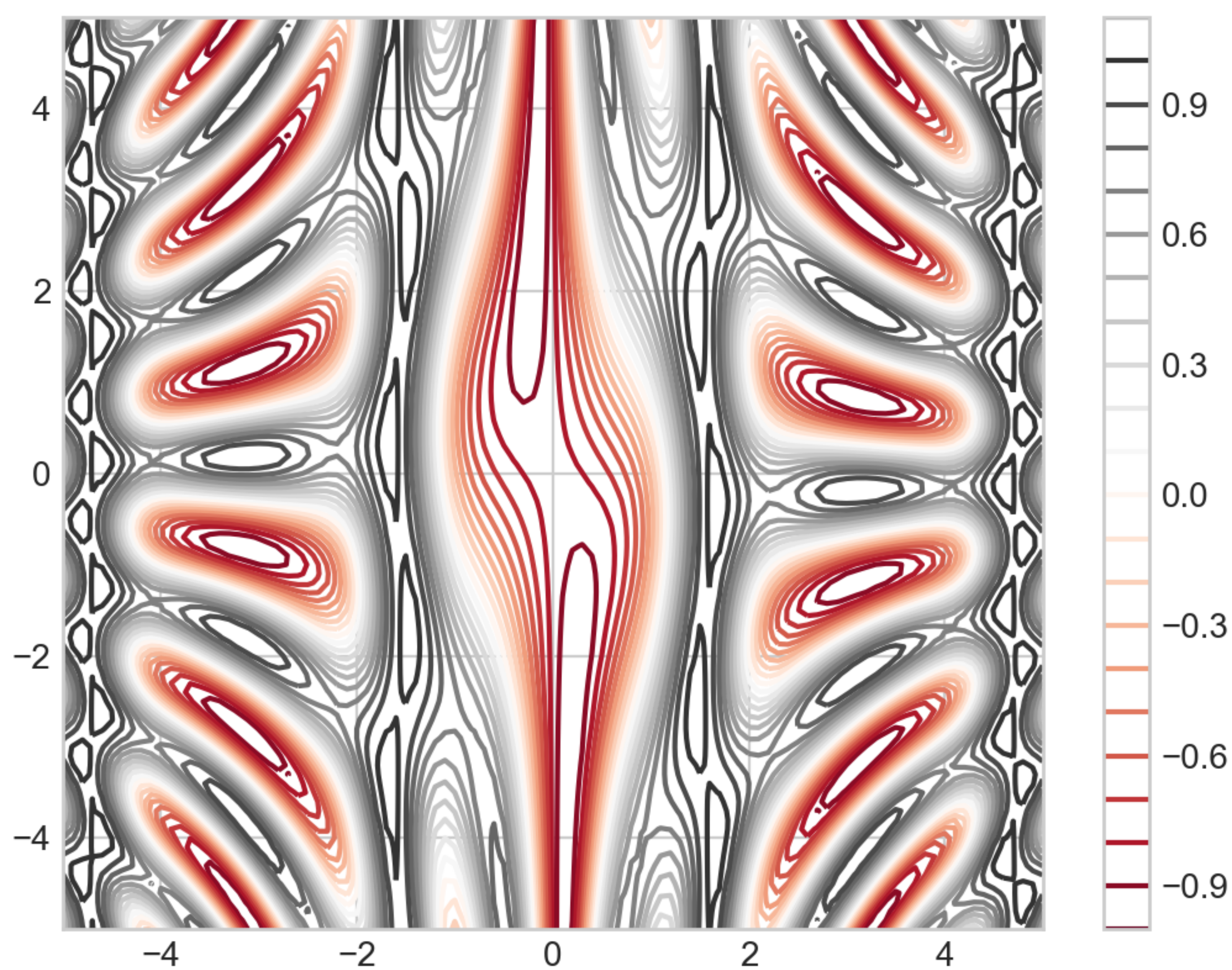
```
plt.contour(X,Y,Z, 20, colors='black')
```



Contour / Density Plots

- Can use a color map

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



Histograms

- Let's use the iris data set
  - A pre-processed version is in `sklearn.datasets`
    - Which means importing it

```
iris = load_iris()
features = iris.data.T
```

    - We better look at it first:

```
>>> features[:,:10]
array([[5.1, 4.9, 4.7, 4.6, 5. , 5.4, 4.6, 5. , 4.4, 4.9],
       [3.5, 3. , 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1],
       [1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5],
       [0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1]])
```

Histograms

- Let's create histograms for all four properties of an Iris set
  - We can define a simple figure with four different panels

```
fig, axs = plt.subplots(2,2,  squeeze = True)
```

  - We then load the axes elements

```
axs[0][0]            axs[0][1]


axs[1][0]            axs[1][1]
```

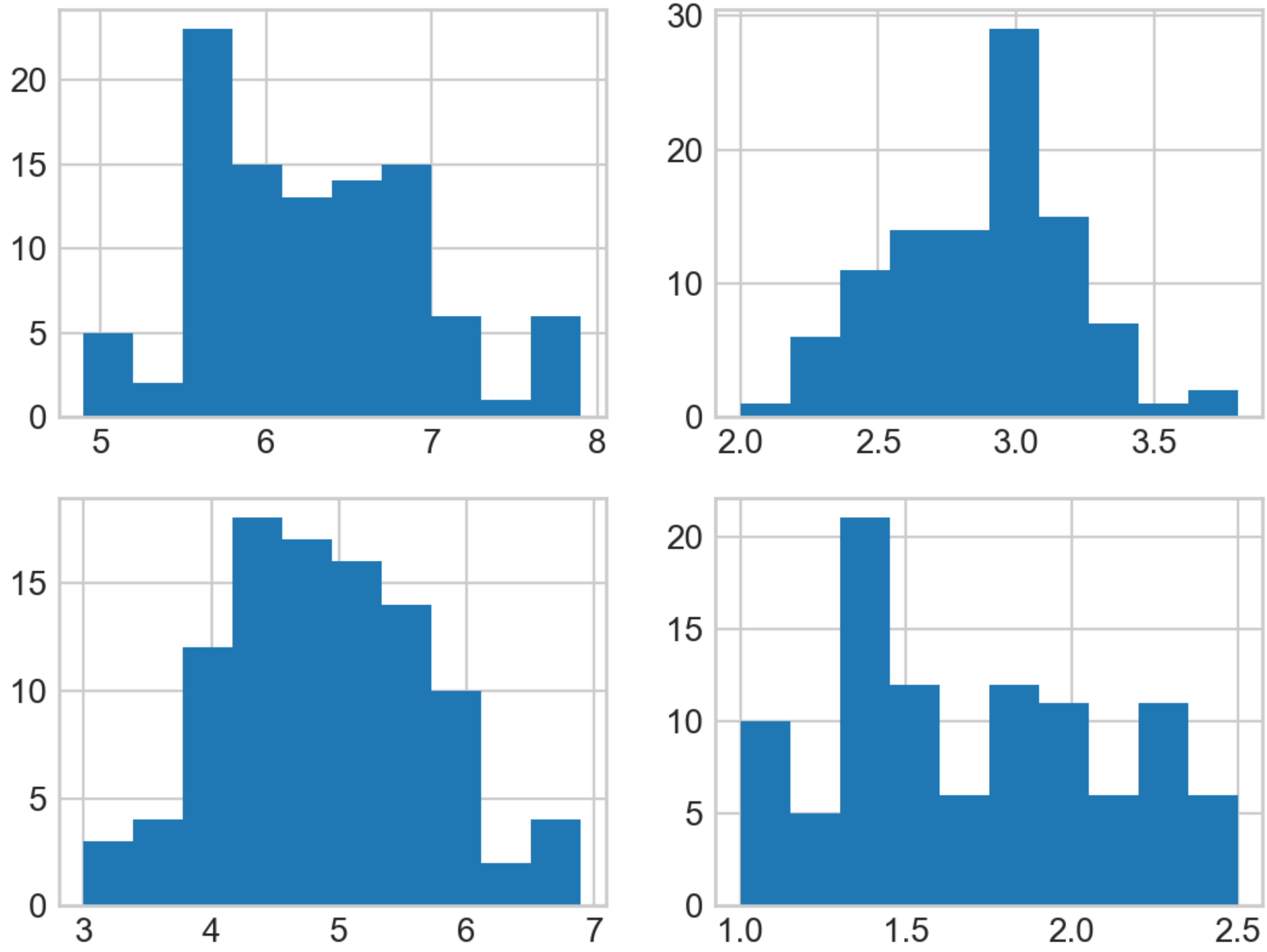
Histograms

- For a histogram, we use the hist method of an axes
  - Needs a np.array and a number of bins

```
axs[0][0].hist(features[0,50:150], bins = 10)
axs[0][1].hist(features[1,50:150], bins = 10)
axs[1][0].hist(features[2,50:150], bins = 10)
axs[1][1].hist(features[3,50:150], bins = 10)
```

Histograms

- Result so far:



Histograms

- The axs objects are actually a tuple
  - N, bins, patches
    - with N the count in each bin
    - bins the number in each bin
    - patches gives us access to the properties drawn

Histograms

- Here is how we get at them

```
N00,bins00,patches00 = axs[0][0].hist(features[0,50:150], bins=10)
N01,bins01,patches01 = axs[0][1].hist(features[1,50:150], bins=10)
N10,bins10,patches10 = axs[1][0].hist(features[2,50:150], bins=10)
N11,bins11,patches11 = axs[1][1].hist(features[3,50:150], bins=10)
```

- And we can see what is in them
    - N00, bins00

```
array([ 5.,  2., 23., 15., 13., 14., 15.,  6.,  1.,  6.])
array([4.9, 5.2, 5.5, 5.8, 6.1, 6.4, 6.7, 7. , 7.3, 7.6, 7.9])
```

Histograms

- The patches is a list of patch objects, one for each bin rectangle
- We can for example update the color
  - Use `patch.set_facecolor(color)`
    - Where color is a number between 0 and 256
    - Which we can select according to colormaps

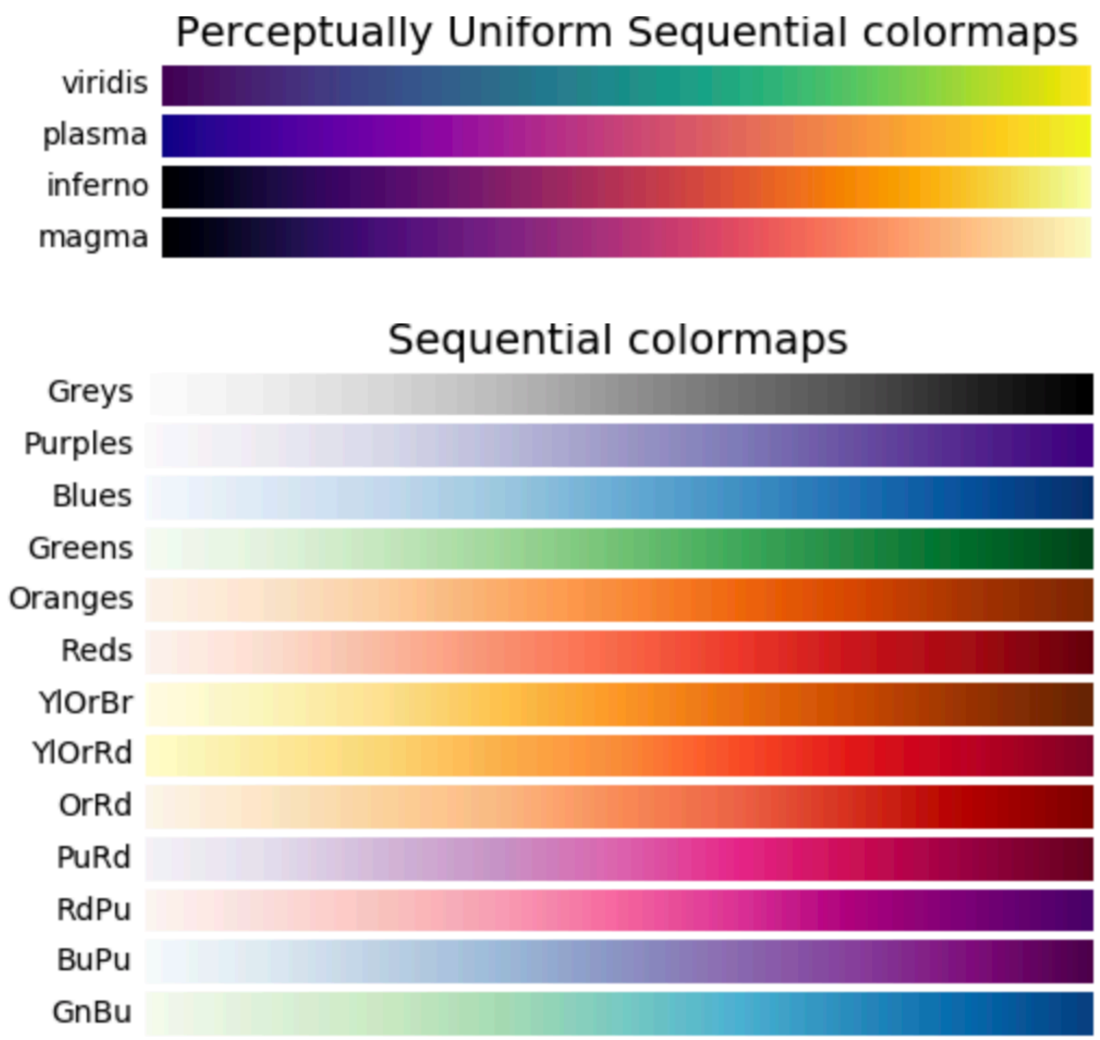
Colormaps

matplotlib

home | examples | gallery | pyplot | docs » Matplotlib Examples » color Examples »

color example code: colormaps_reference.py

(Source code)



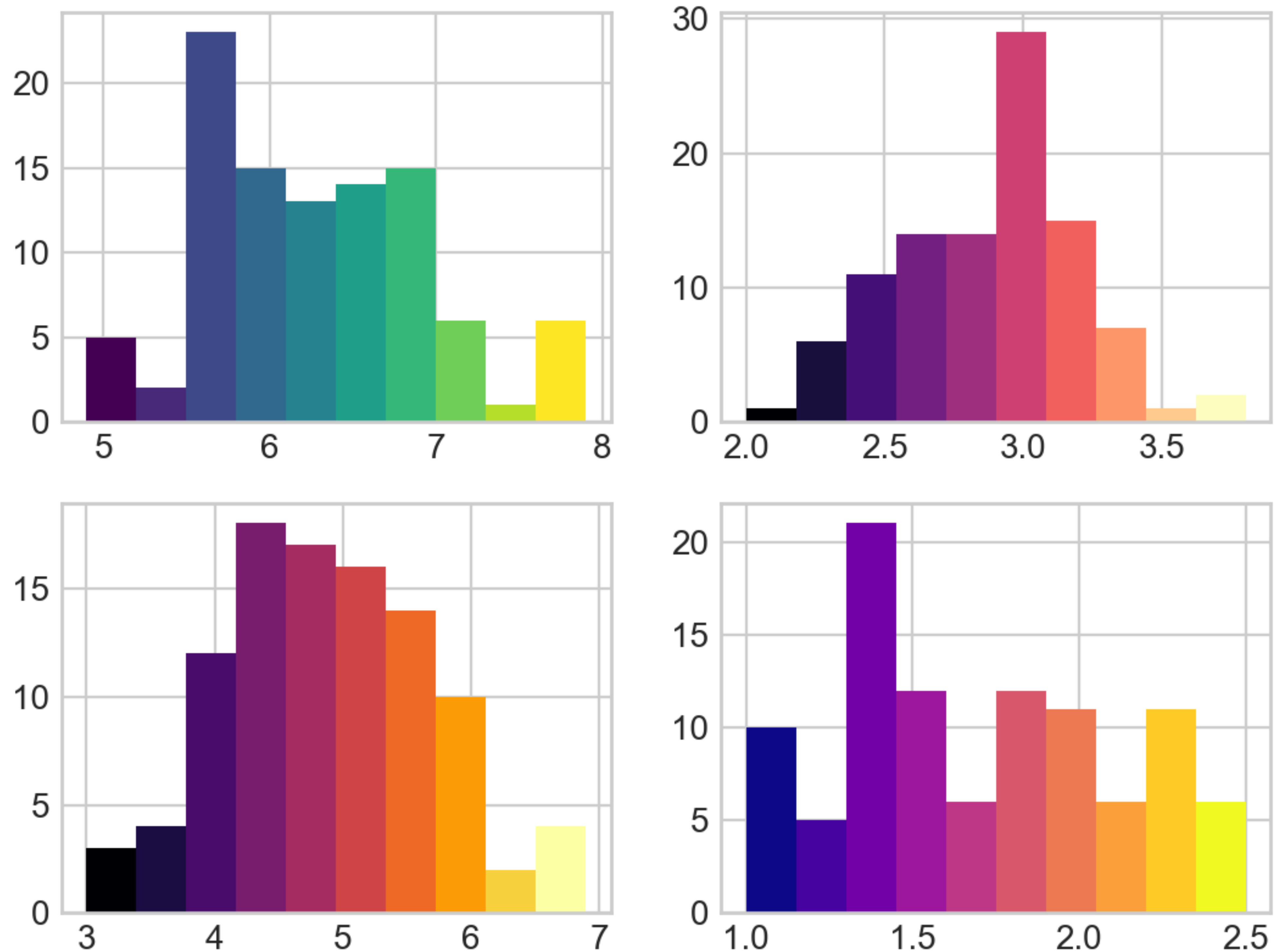
Histograms

- For each of the four histograms, we can use a different color map
  - We pick the number uniformly through 1 … 256

```
for i, patch in enumerate(patches00):
    color = plt.cm.viridis(i*256//9)
    patch.set_facecolor(color)

for i, patch in enumerate(patches01):
    color = plt.cm.magma(i*256//9)
    patch.set_facecolor(color)
```

Histograms



20
15
10
5
0
5
6
7
8
30
20
10
0
2.0
2.5
3.0
3.5
15
10
5
0
3
4
5
6
7
20
15
10
5
0
1.0
1.5
2.0
2.5

Python enumerate

- Remember that Python has some great list tools
    - `enumerate(a_list_or_sequence)`
    - will generate tuples of index and element

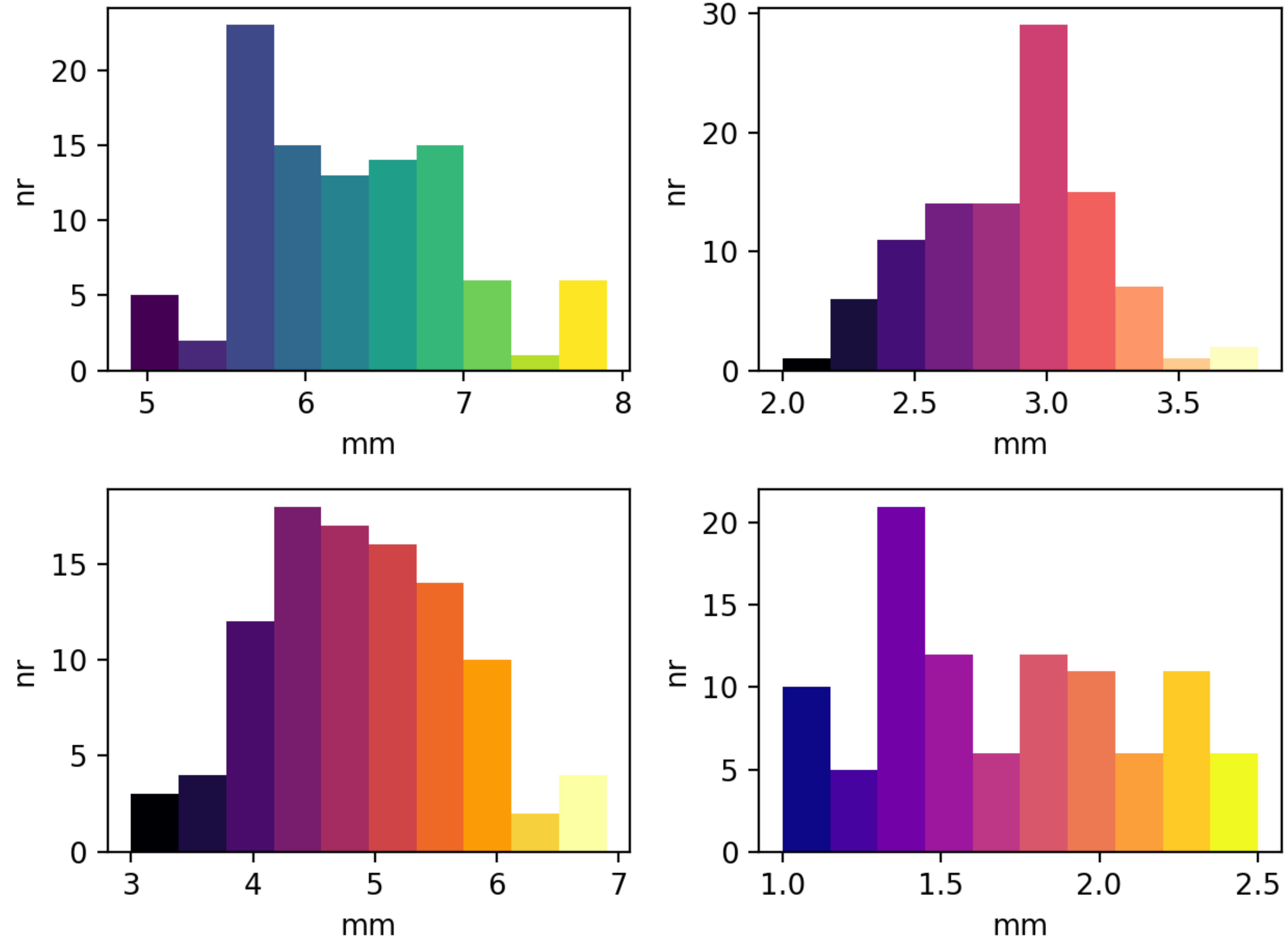
Histograms

- We should describe what we are displaying
  - First, let's add labels for the axes
    - Turns out that the rows are too close together
      - Search the web: can be adjusted with tight_layout

```
for ax in axs:
    for a in ax:
        a.set_xlabel('mm')
        a.set_ylabel('nr')

fig.tight_layout(pad=1.0)
```

Histograms



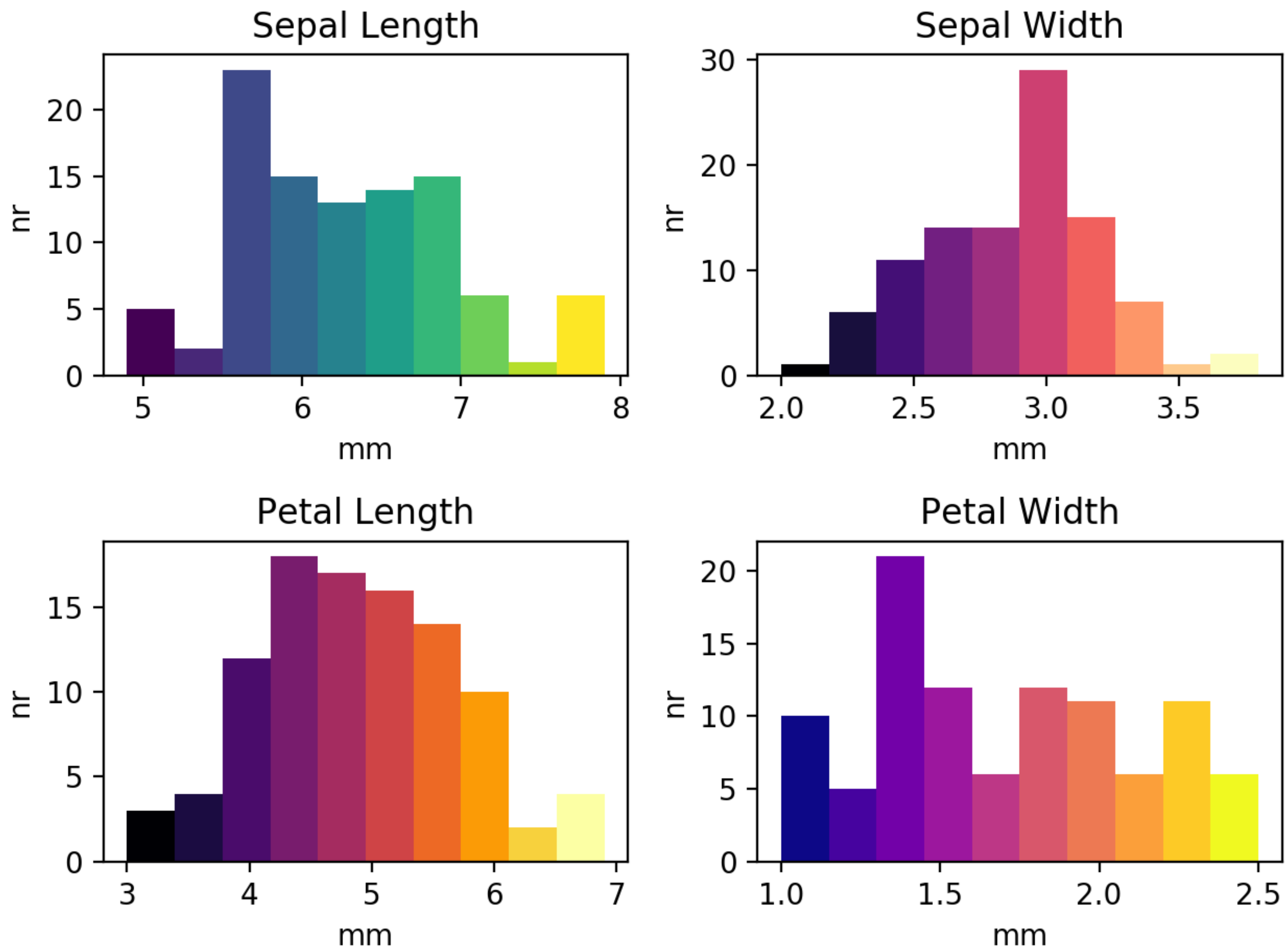
nr
mm
0
5
10
15
20
5
6
7
8
nr
mm
0
10
20
30
2.0
2.5
3.0
3.5
nr
mm
0
5
10
15
3
4
5
6
7
nr
mm
0
5
10
15
20
1.0
1.5
2.0
2.5

Histograms

```
for ax in axs:
    for a in ax:
        a.set_xlabel('mm')
        a.set_ylabel('nr')
axs[0][0].set_title('Sepal Length')
axs[0][1].set_title('Sepal Width')
axs[1][0].set_title('Petal Length')
axs[1][1].set_title('Petal Width')

fig.tight_layout(pad=1.0)
```

Histograms


Sepal Length
nr
mm
Sepal Width
nr
mm
Petal Length
nr
mm
Petal Width
nr
mm

2D-Histograms

- Two-dimensional histograms use color to show the numbers into a two dimensional bin

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

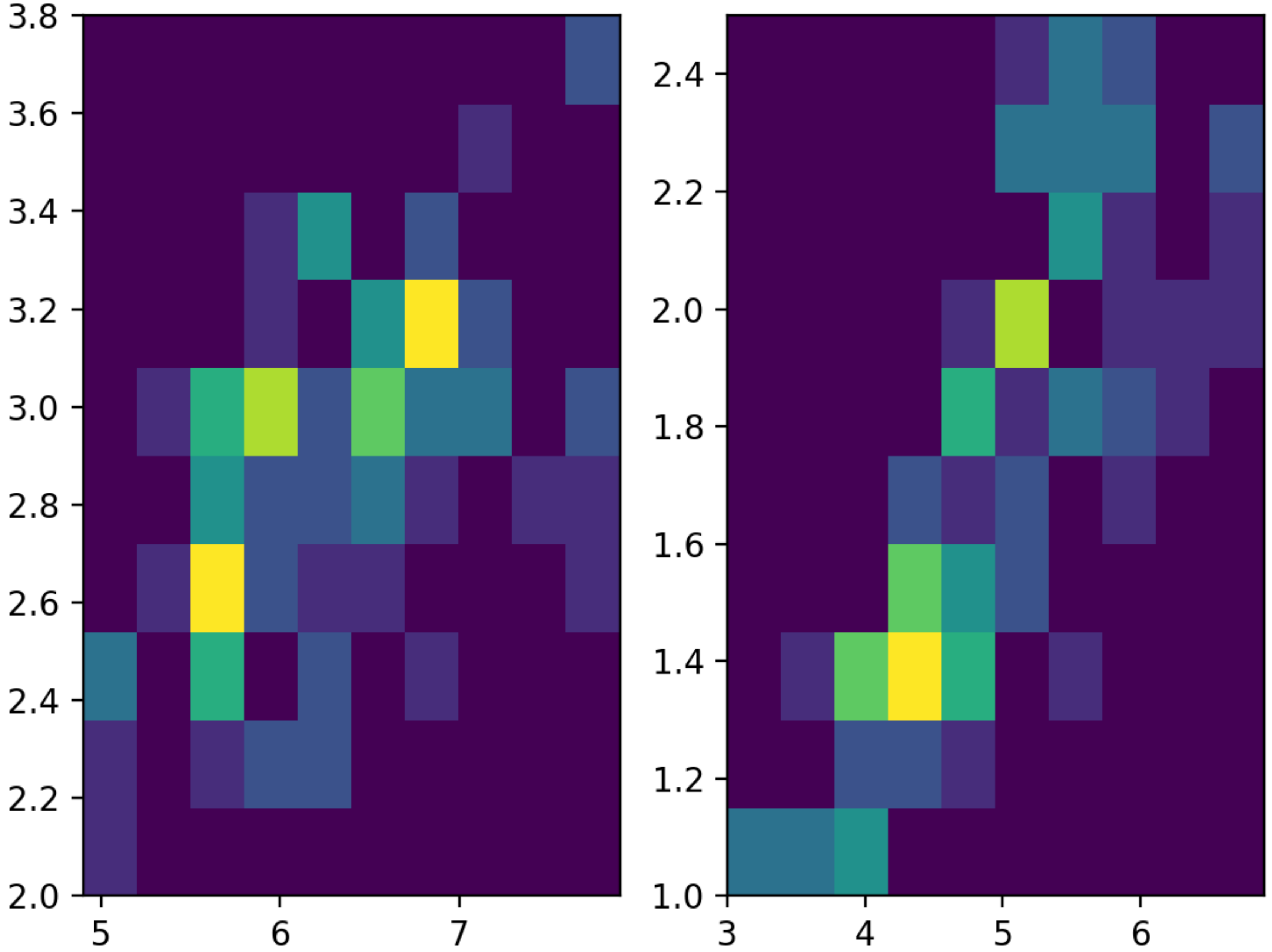

iris = load_iris()
features = iris.data.T

fig, axs = plt.subplots(1,2)
axs[0].hist2d(features[0,50:150],features[1,50:150])
axs[1].hist2d(features[2,50:150],features[3,50:150])

plt.show()
```

2D-Histograms

- The result is harder to read:



2D-Histograms

- We can adjust the number of bins on each side
- And adjust the color scheme
  - `from matplotlib import colors`
  - set color map to something easily interpretable

2D-Histograms

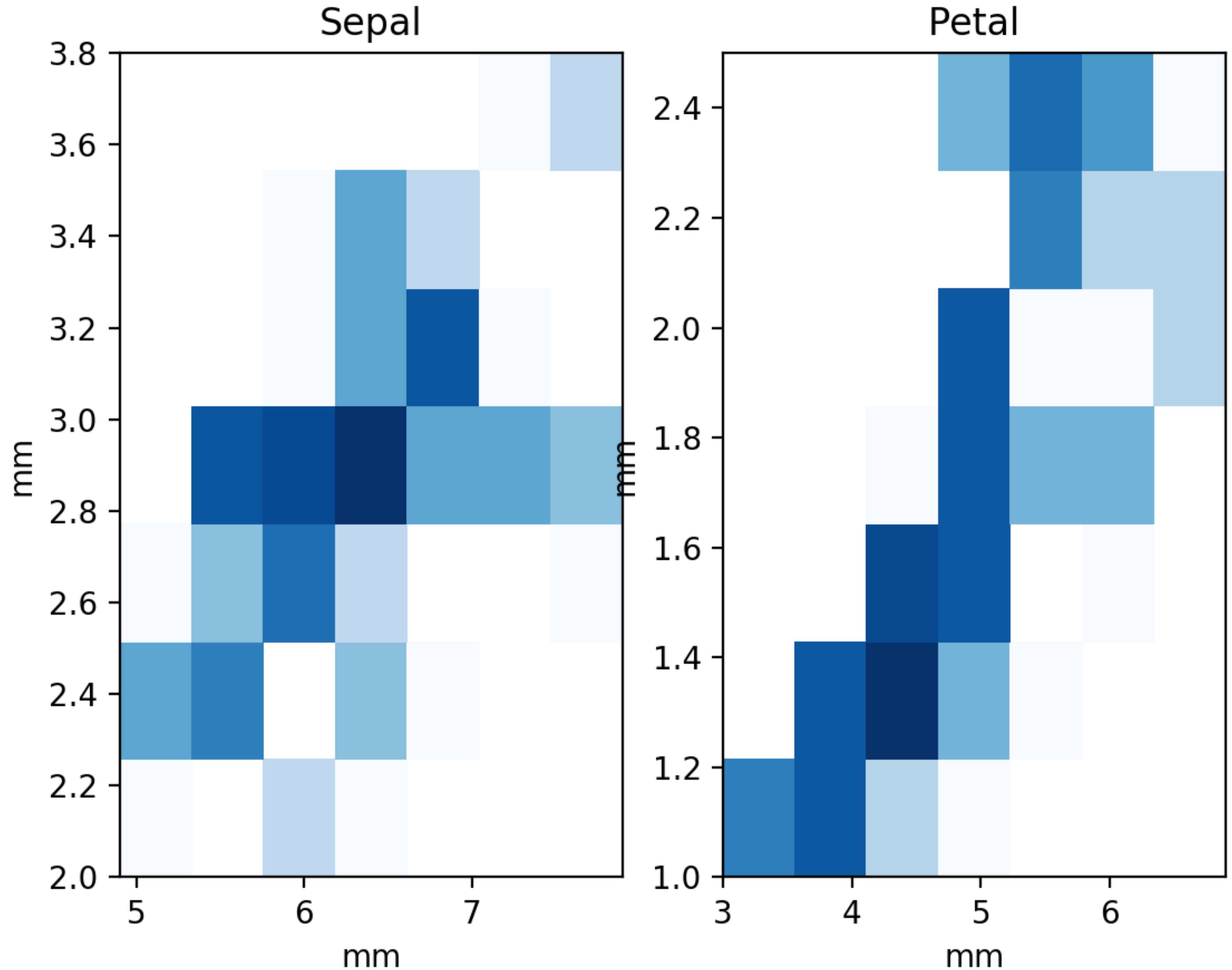
```
fig, axs = plt.subplots(1,2)
axs[0].hist2d(features[0,50:150],
              features[1,50:150],
              bins=7,
              norm=colors.LogNorm(),
              cmap = 'Blues')
axs[1].hist2d(features[2,50:150],
              features[3,50:150],
              bins=7,
              norm=colors.LogNorm(),
              cmap = 'Blues')
```

2D-Histograms

- Then add labels and titles

```
for i in range(2):
    axs[i].set_xlabel('mm')
    axs[i].set_ylabel('mm')
axs[0].set_title('Sepal')
axs[1].set_title('Petal')
```

2D-Histograms



2D-Histograms

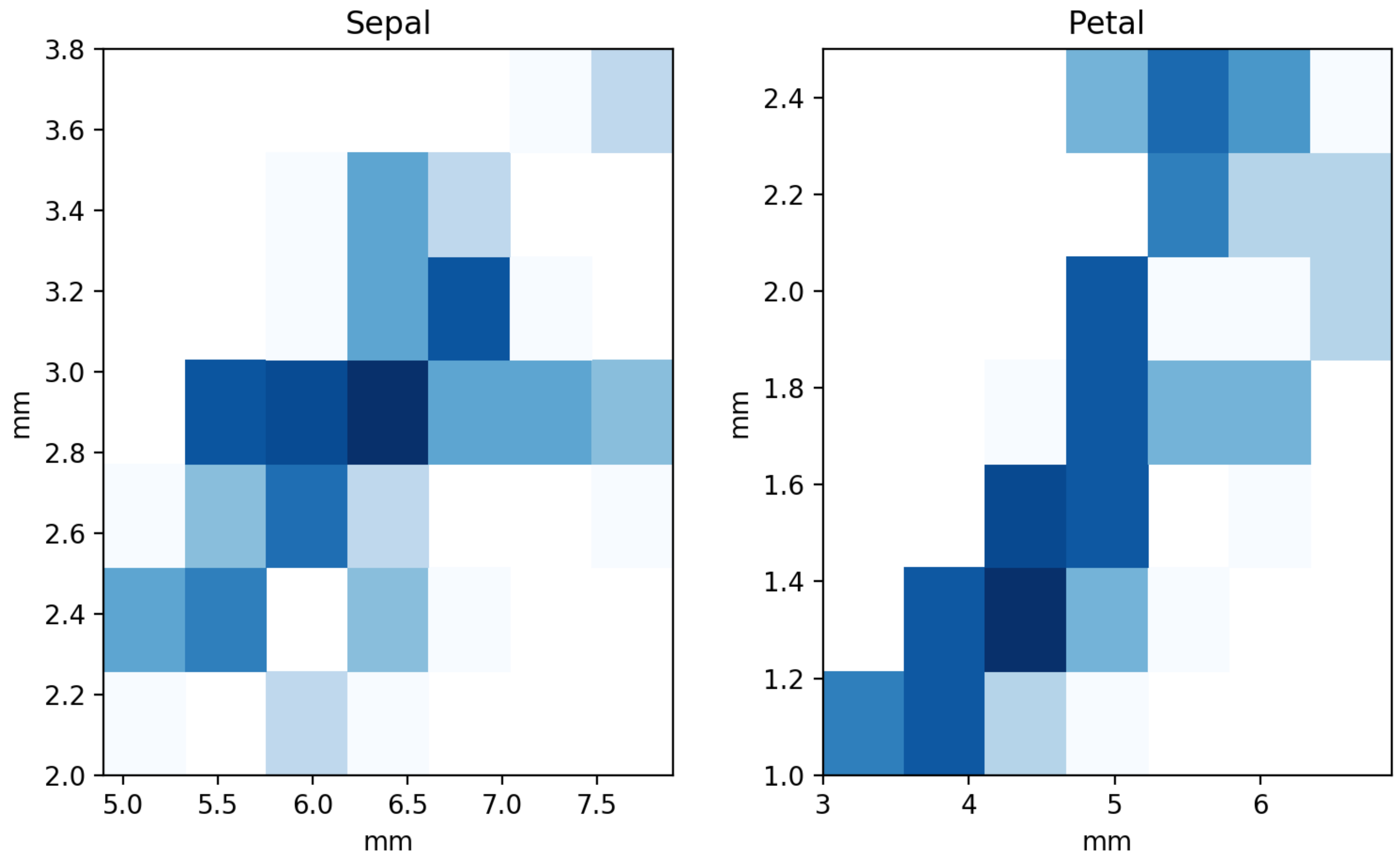
- Need to adjust padding

```
for a in axs:
    a.set_xlabel('mm')
    a.set_ylabel('mm')

axs[0].set_title('Sepal')
axs[1].set_title('Petal')

fig.tight_layout(pad=1.0)
```

2D-Histograms



Plot Legends

- Let's create a simple plot: compare arctan and the logistic functions
- Provide a simple legend
    - Give a label to the plot
    - call legend and the axes object

Plot Legends

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

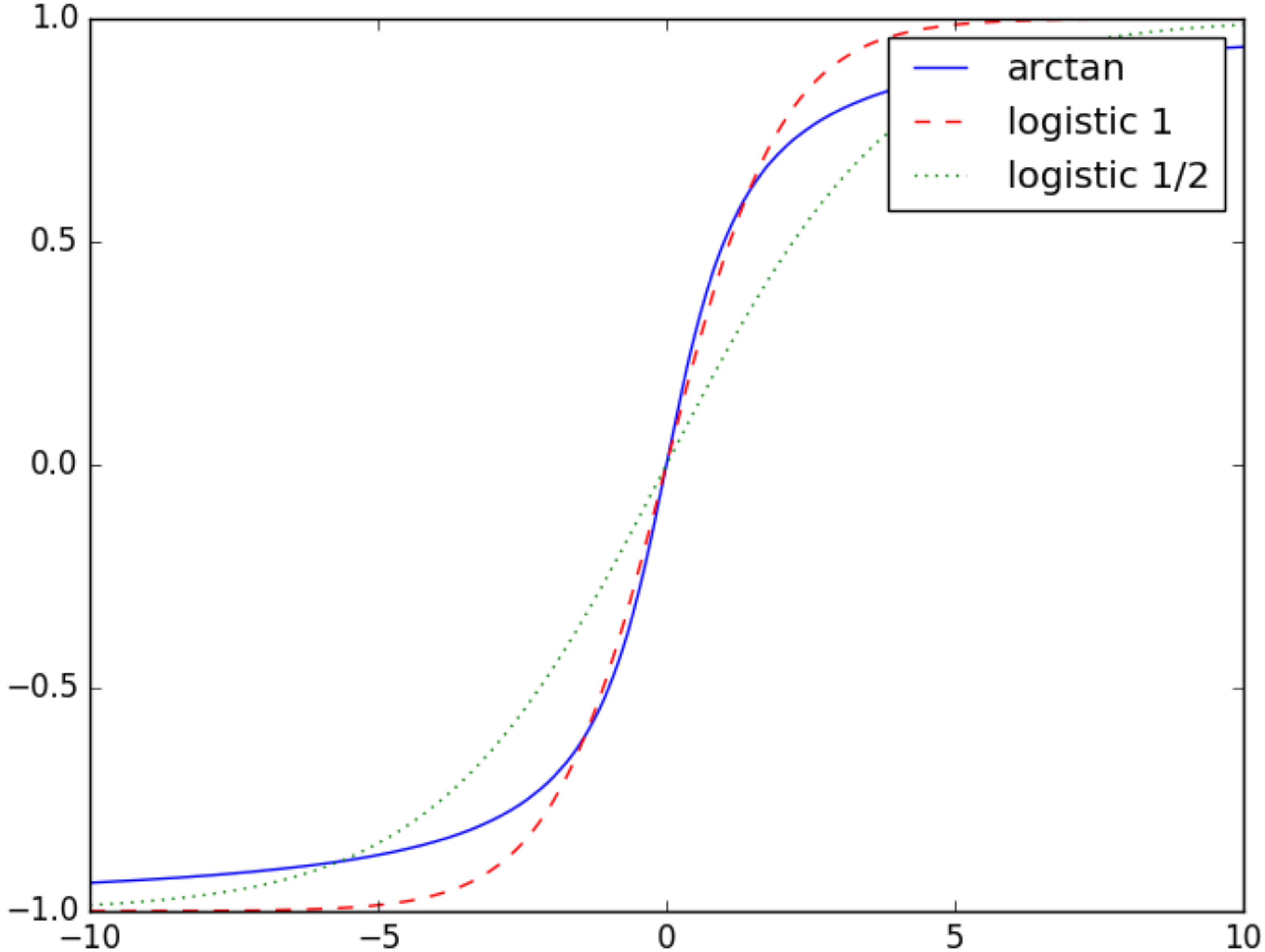
plt.style.use('classic')

x = np.linspace(-10,10,1001)
fig, ax = plt.subplots()
ax.plot(x, 2*np.arctan(x)/math.pi, 'b-', label='arctan')
ax.plot(x, 2/(np.exp(-x)+1)-1, 'r--', label='logistic 1')
ax.plot(x, 2/(np.exp(-x/2)+1)-1, 'g:', label='logistic 1/2')

ax.legend()

plt.show()
```

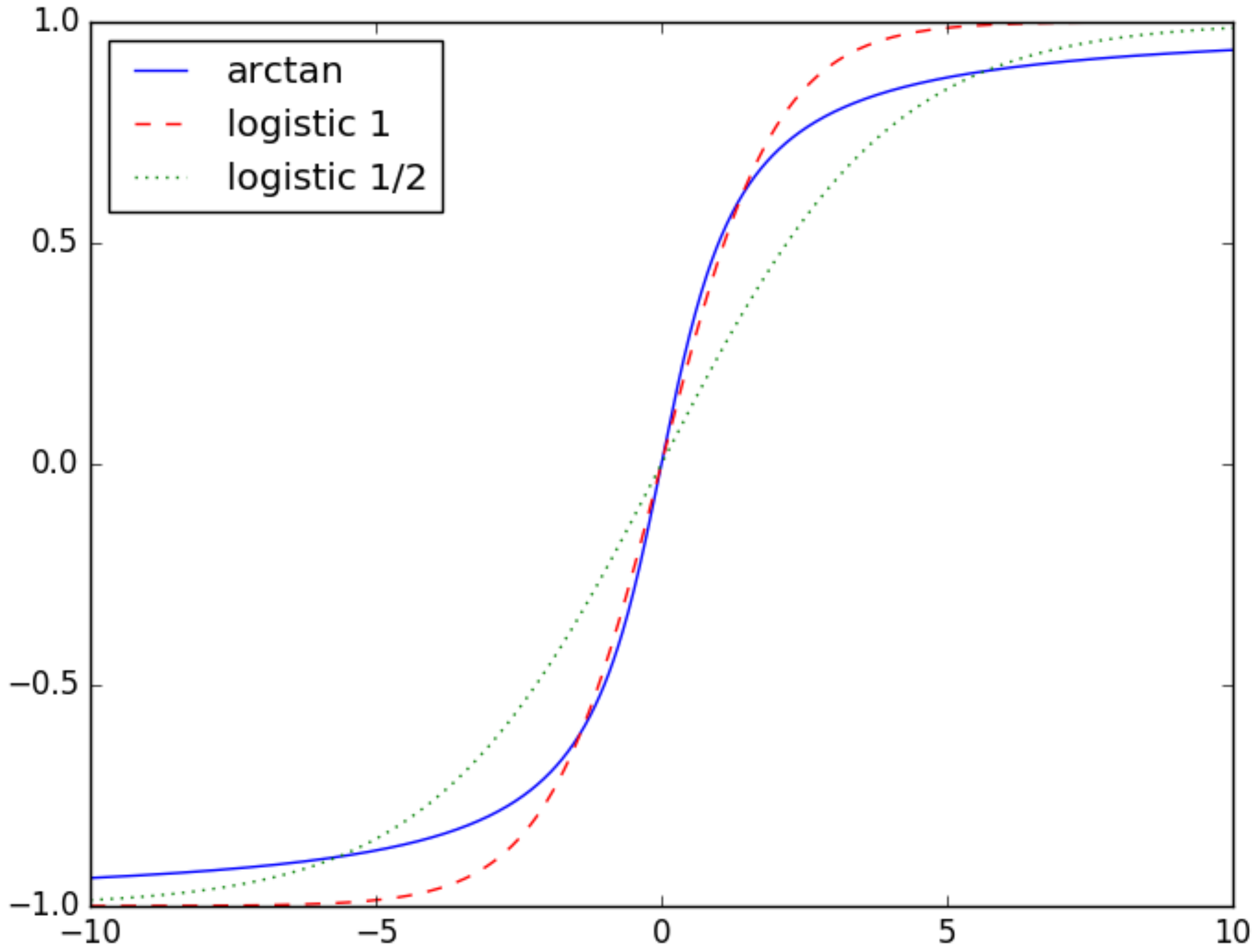
Plot Legends



Plot Legends

- Can specify placement of the legend

```
ax.legend(loc='upper left')
```



Plot Legends

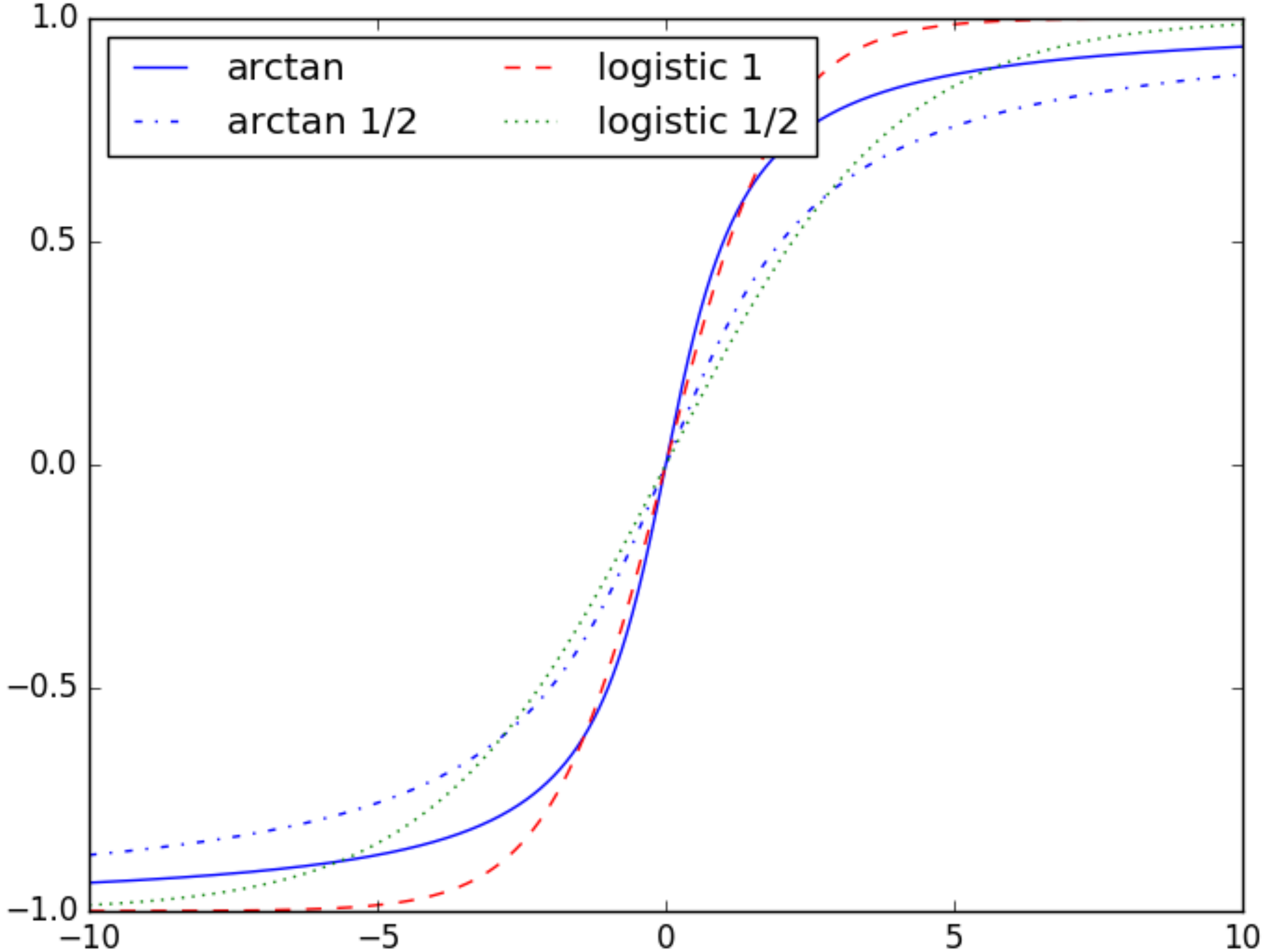
- Can specify the number of columns in the legend

```
x = np.linspace(-10,10,1001)
fig, ax = plt.subplots()
ax.plot(x, 2*np.arctan(x)/math.pi, 'b-', label='arctan')
ax.plot(x, 2*np.arctan(x/2)/math.pi, 'b-.', label='arctan 1/2')
ax.plot(x, 2/(np.exp(-x)+1)-1, 'r--', label='logistic 1')
ax.plot(x, 2/(np.exp(-x/2)+1)-1, 'g:', label='logistic 1/2')


ax.legend(loc='upper left', ncol = 2)

plt.show()
```

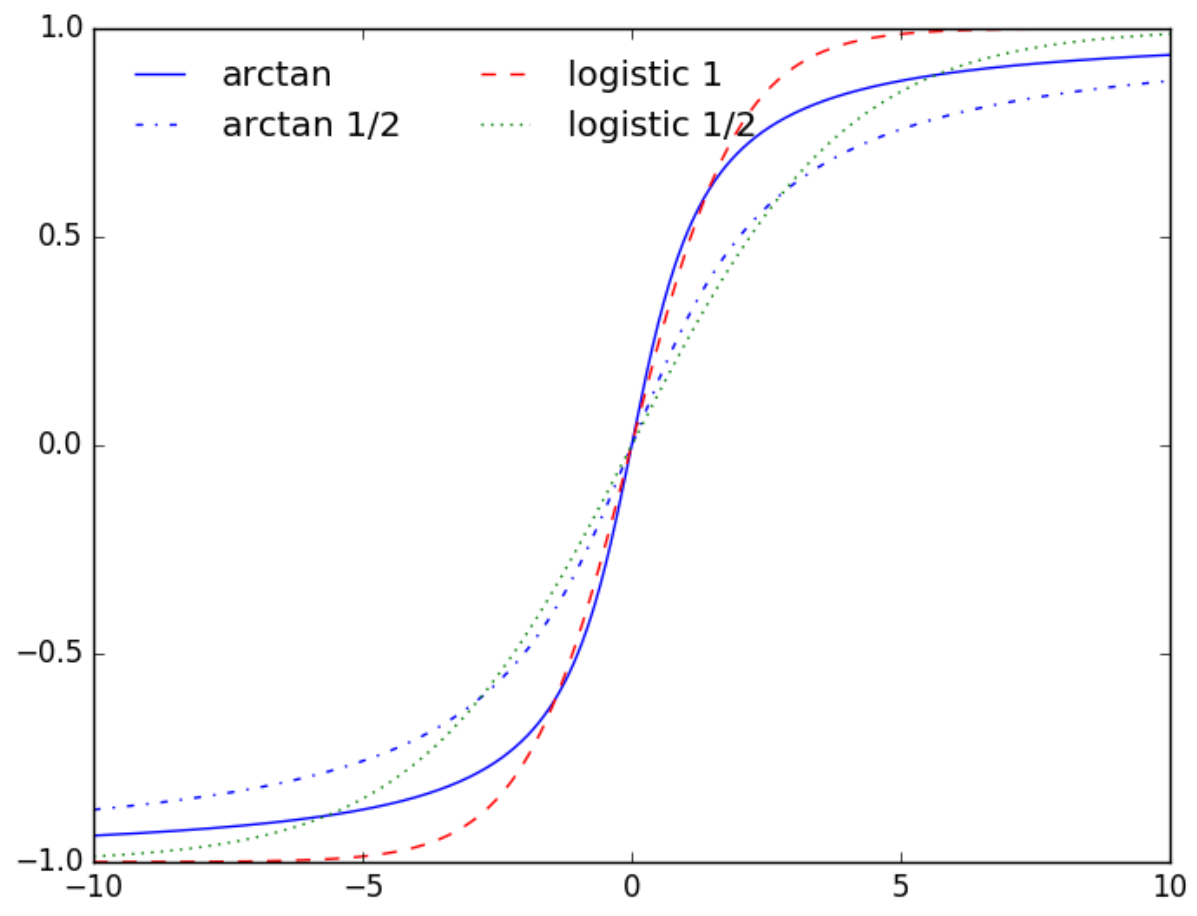
Plot Legends



Plot Legends

- Turn off the frame

```
ax.legend(loc='upper left',
          ncol = 2,
          frameon=False)
```



Plot Legends

- If we do not provide a label in a plot, then it will be ignored

Plot Legends

- Can use labels to provide additional information
  - Open california_cities.csv
  - Create a scatter plot based on latitude and longitude
  - Set the color to the population (decadic logarithm)
  - Set the size to the area of the city
  - Deploy a colorbar with a label (in LaTeX)

Plot Legends

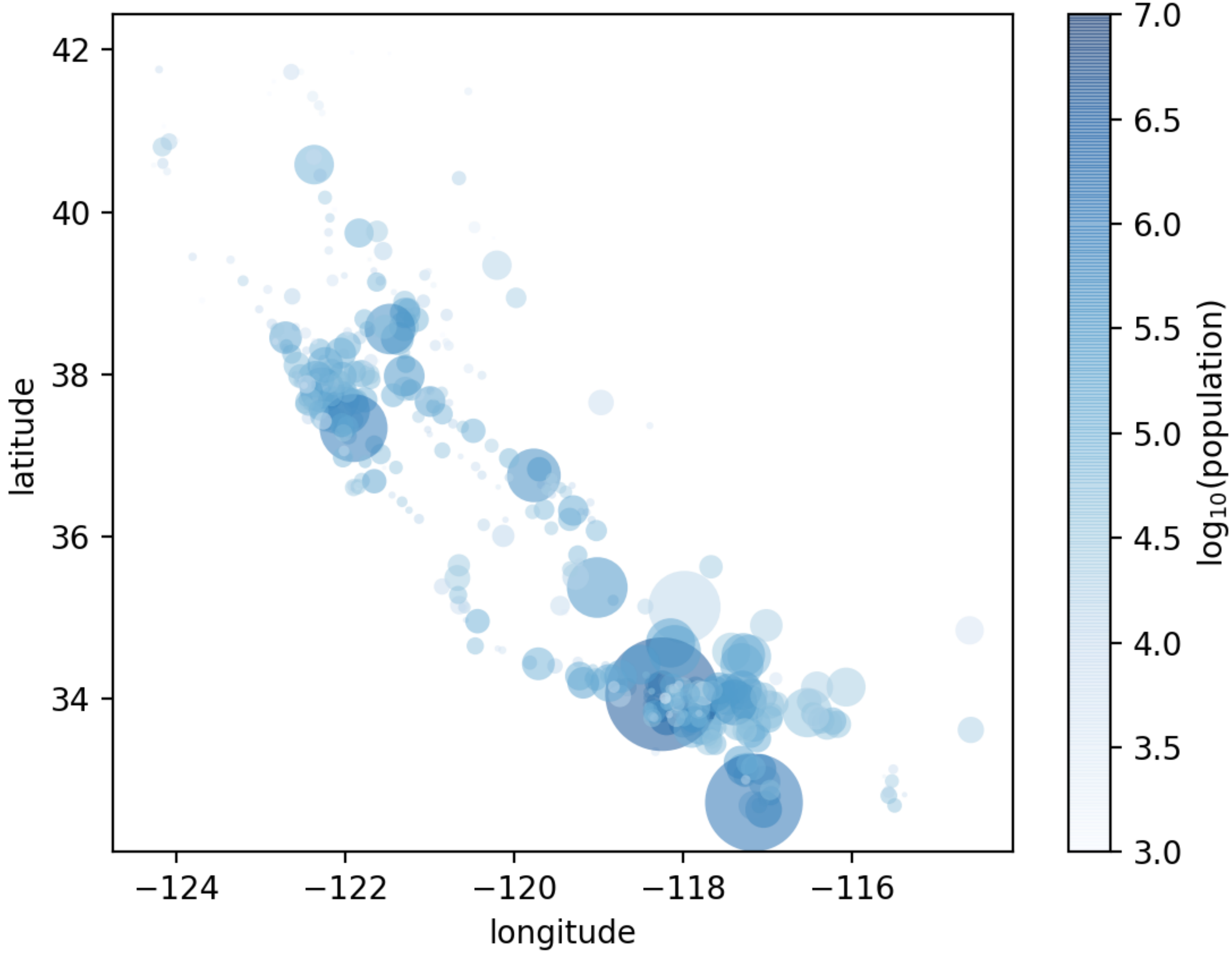
```
cities = pd.read_csv('california_cities.csv')

lat, lon = cities['latd'], cities['longd']
population, area = cities['population_total'],
cities['area_total_km2']

plt.scatter(lon, lat, label=None,
            c = np.log10(population), cmap='Blues',
            s = area, linewidth = 0, alpha = 0.5)

plt.axis(aspect='equal')
plt.xlabel('longitude')
plt.ylabel('latitude')
plt.colorbar(label='$\log_{10}$(population)')
plt.clim(3,7)  #color scaling
```

Plot Legends



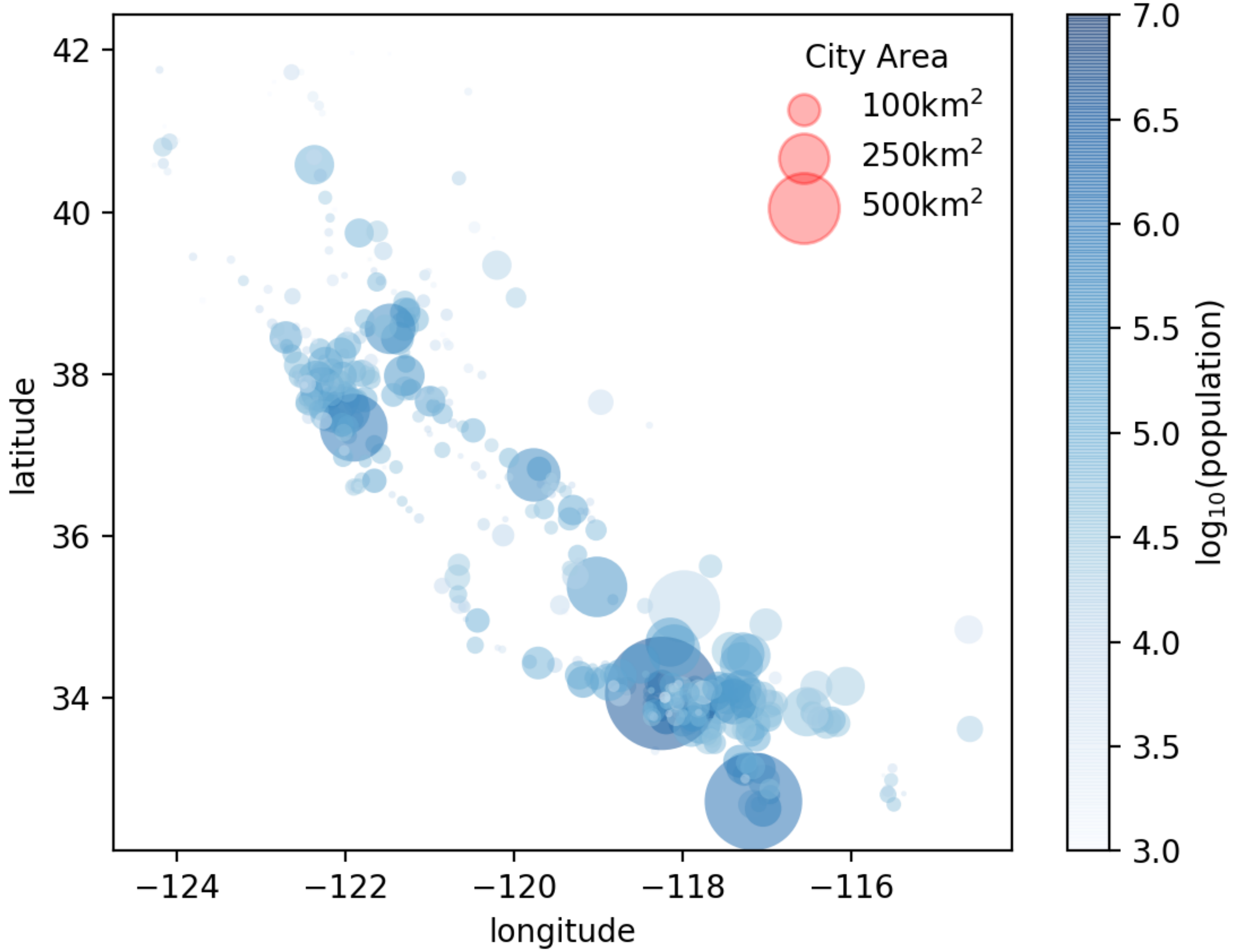
Plot Legends

- This lacks an explanation of the areas of the cities
  - In order to create a legend, we need to plot comparison cities with 'label'
  - But these cities do not have to exist

```
for area in [100, 250, 500]:
    plt.scatter([],[], c='r',
                alpha = 0.3, s=area,
                label = str(area)+'km$^2$')
plt.legend(frameon = False, title='City Area')
```

Plot Legends

- The label isn't so great

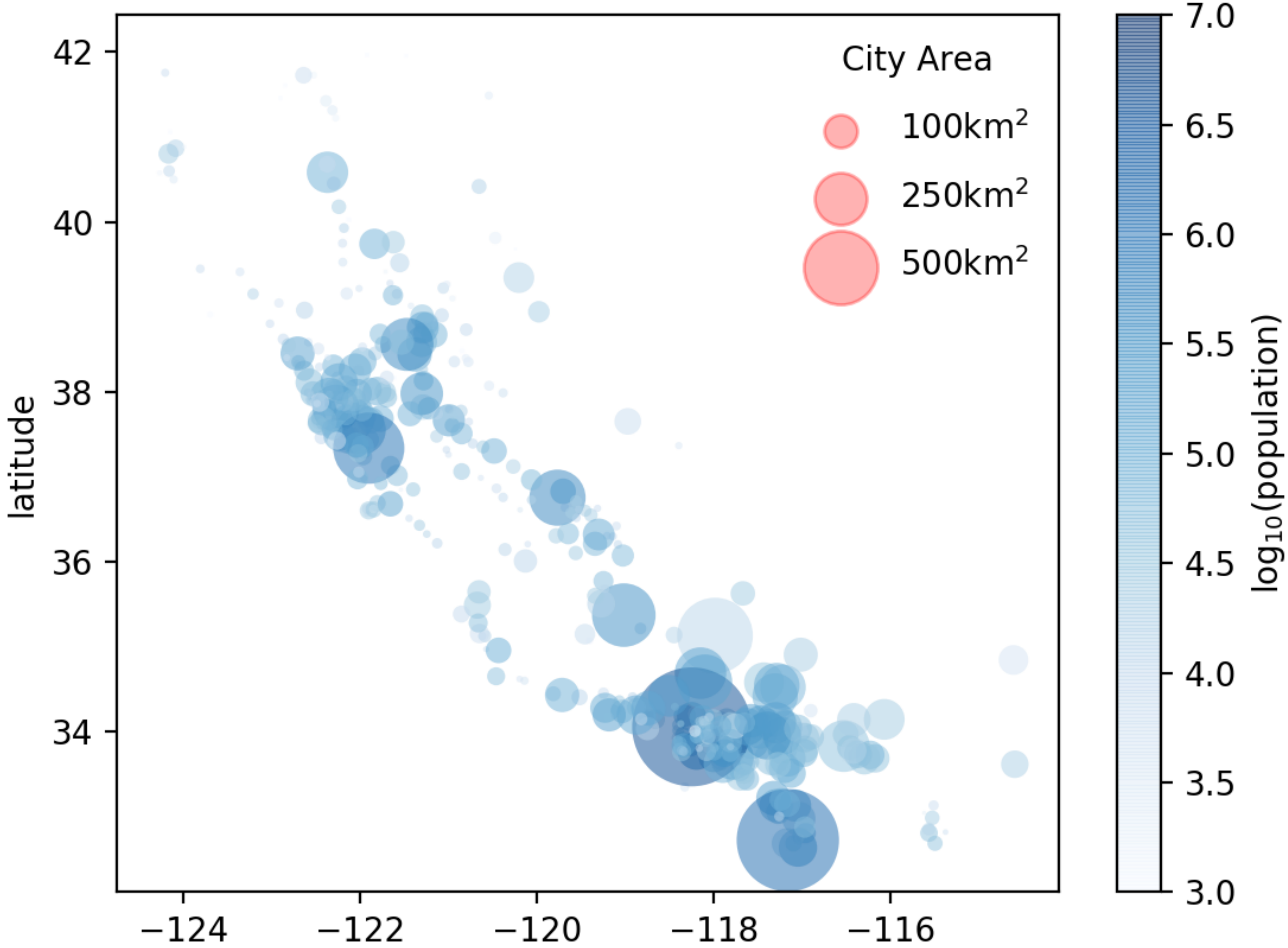


Plot Legends

- Use labelspacing to avoid overlap

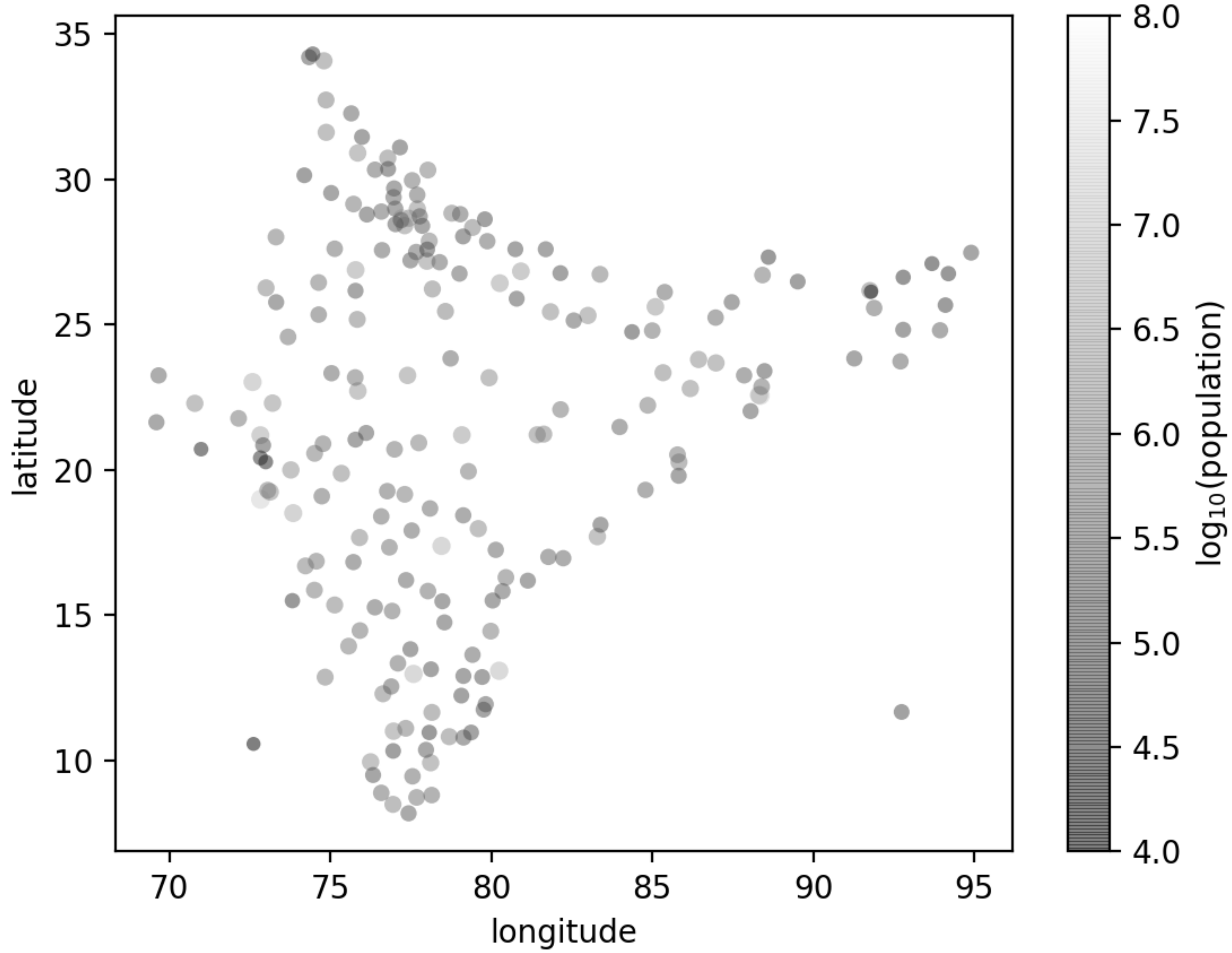
```
plt.legend(frameon = False,
           title='City Area',
           labelspacing=1)
```

Plot Legends



Homework

- Use the in.csv map to create a map of India's cities



Homework

- Restrict the map to cities of more than 2 million population
- Use a better color-scheme