DATA VISUALIZATION

CS 3030: Python

Instructor: Damià Fuentes Escoté

University of Colorado
Colorado Springs

Next class – 25/4/18 – Quiz 5

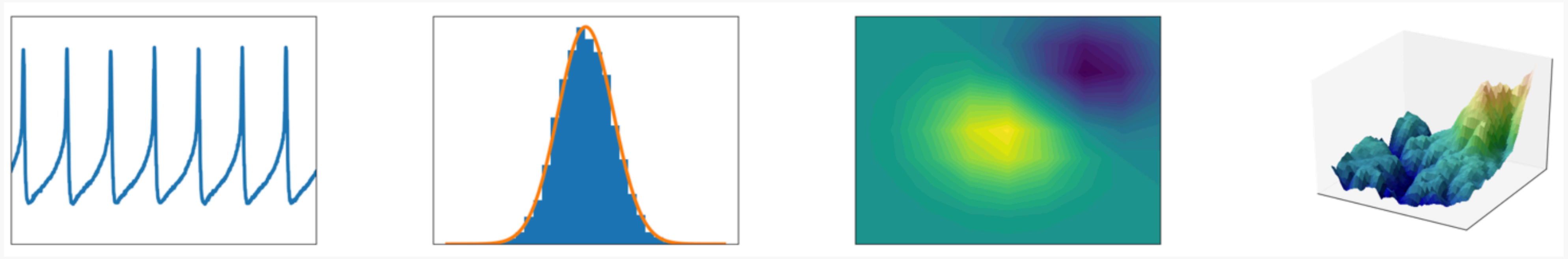
- Lectures 1 to 9
  - *From basic stuff like functions to iterators, etc*

Homework 9 late submissions

- Due to high demand and the amount of work that Homework 9 requires I will allow late submissions till next Tuesday April 29th (1 week more) with a penalty of 4 points.
- Those that submit it late will be graded but only up to **36 points instead of 40**.

Data visualization

- Data visualization is the discipline of trying to understand data by placing it in a visual context so that patterns, trends and correlations that might not otherwise be detected can be exposed.



https://towardsdatascience.com/introduction-to-data-visualization-in-python-89a54c97fbed

Data visualization

- Python offers multiple great graphing libraries that come packed with lots of different features. No matter if you want to create interactive, live or highly customized plots python has an excellent library for you.
- To get a little overview here are a few popular plotting libraries:
  - *Matplotlib: low level, provides lots of freedom*
  - *Pandas Visualization: easy to use interface, built on Matplotlib*
  - *Seaborn: high-level interface, great default styles*
  - *ggplot: based on R’s ggplot2, uses Grammar of Graphics*
  - *Plotly: can create interactive plots*

Data visualization

- Python offers multiple great graphing libraries that come packed with lots of different features. No matter if you want to create interactive, live or highly customized plots python has an excellent library for you.
- To get a little overview here are a few popular plotting libraries:
  - ***Matplotlib: low level, provides lots of freedom -> The one we'll learn***
  - *Pandas Visualization: easy to use interface, built on Matplotlib*
  - *Seaborn: high-level interface, great default styles*
  - *ggplot: based on R's ggplot2, uses Grammar of Graphics*
  - *Plotly: can create interactive plots*

MATPLOTLIB

Why matplotlib?

- Due to the growing interest in python the popularity of matplotlib is continually rising.
- The attractiveness of Matplotlib lies in the fact that it is widely considered to be a perfect alternative to MATLAB, if it is used in combination with Numpy and Scipy.
  - ***Whereas MATLAB is expensive and closed source, Matplotlib is free and open source code.***

https://www.python-course.eu/matplotlib.php

Why matplotlib?

- Matplotlib is its steep learning curve, which means that users usually make rapid progress after having started.
- The official website has to say the following about this:
  - *"matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc, with just a few lines of code."*

https://www.python-course.eu/matplotlib.php

Introduction

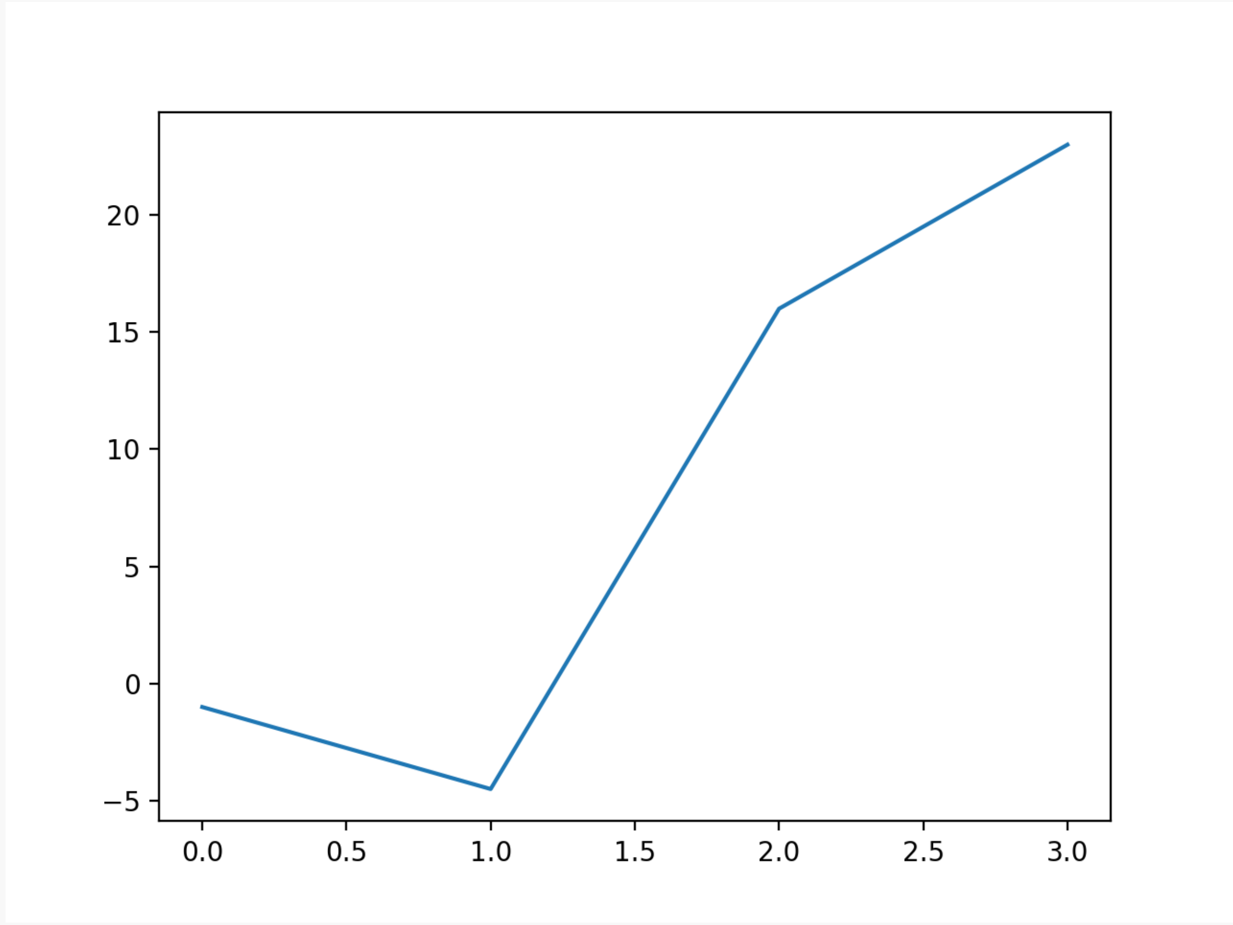
A First Example

- We will start with a simple graph, which is as simple as simple can be.
- A graph in matplotlib is a two- or three-dimensional drawing showing a relationship by means of points, a curve, or amongst others a series of bars.
- We have two axis:
  - *The **horizontal X-axis** is representing the **independent values***
  - *and the **vertical Y-axis** corresponds to the **depended values**.*

- It is common practice to rename **matplotlib.pyplot to plt**.

A First Example

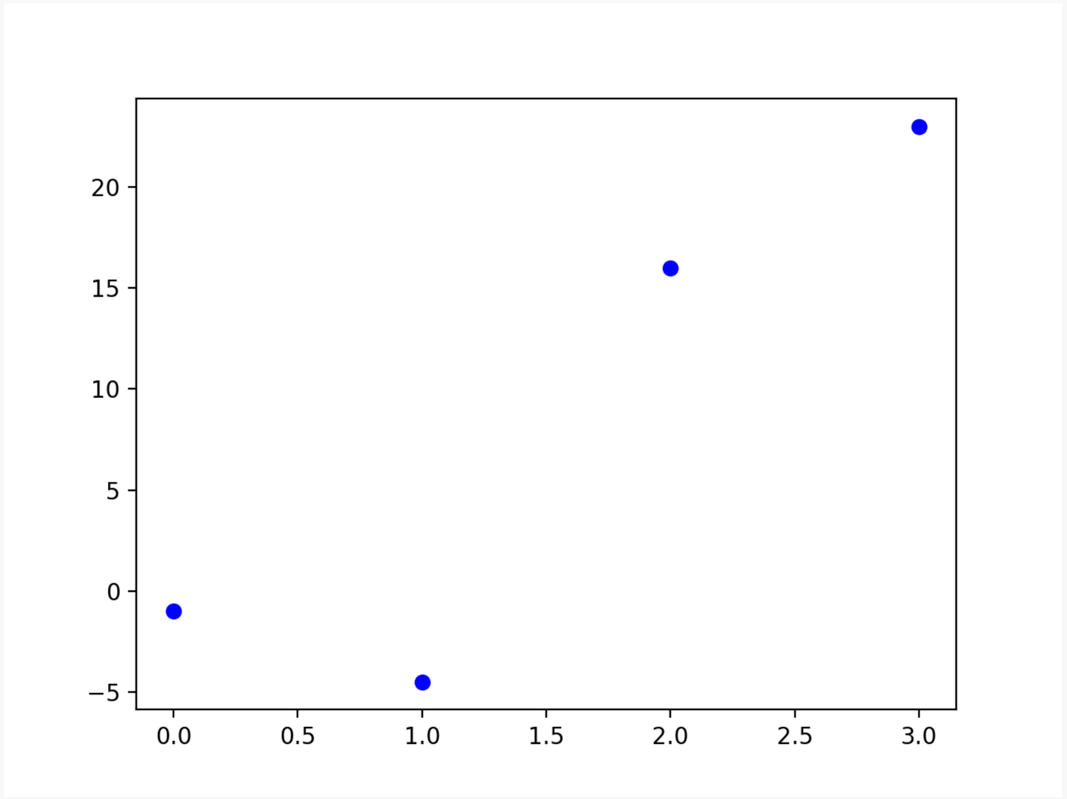
```python
import matplotlib.pyplot as plt

plt.plot([-1, -4.5, 16, 23])
plt.show()
```



A first example

- What we see in the previous slide is a **continuous graph**, even though we provided discrete data for the Y values. By adding a format string to the function call of plot, we can create a graph with **discrete values**, in our case blue circle markers. The format string defines the way how the discrete points have to be rendered.

```
plt.plot([-1, -4.5, 16, 23], "ob")
plt.show()
```



The format parameter of pyplot.plot

- We have used "ob" in our previous example as the format parameter.
- It consists of two characters.
  - *The first one defines the line style or the dicrete value style, i.e. the markers,*
  - *while the second one chooses a color for the graph.*
- The order of the two characters could have been reversed, i.e. we could have written it as "bo" as well.

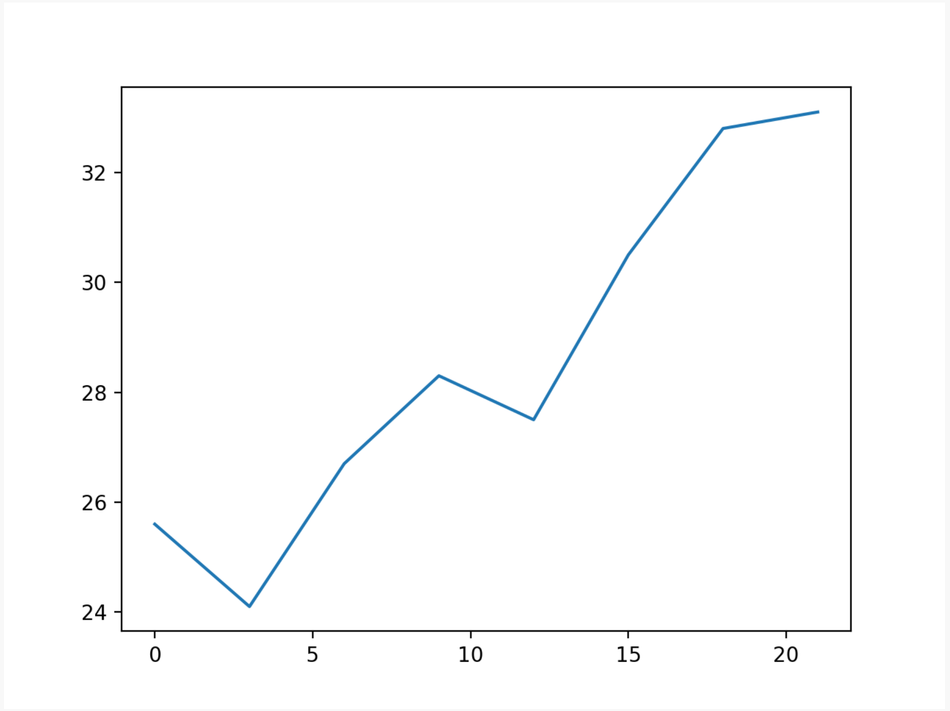
character	description
'-'	solid line style
'--'	dashed line style
'-.'	dash-dot line style
':'	dotted line style
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker

character	description
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

character	color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

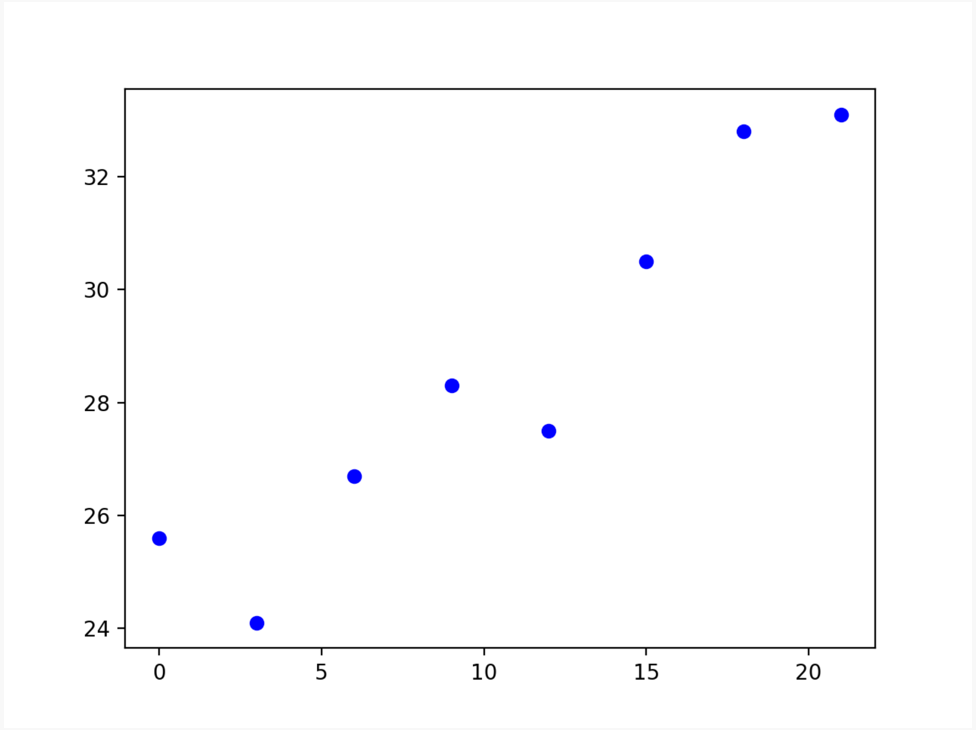
X values to the plot function

```
# our X values:
days = list(range(0, 22, 3))
# our Y values:
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
plt.plot(days, celsius_values)
plt.show()
```



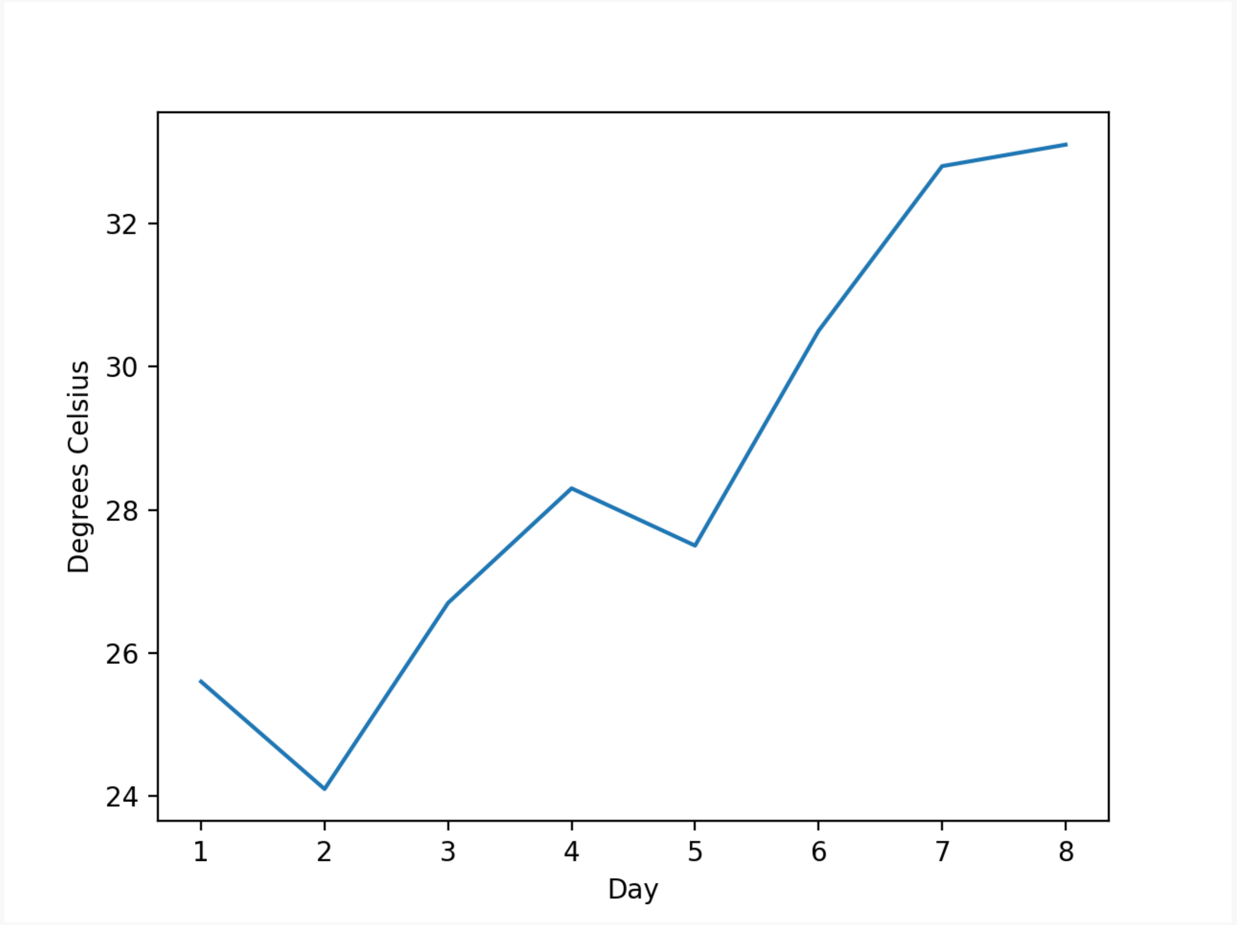
X values to the plot function

```
# our X values:
days = list(range(0, 22, 3))
# our Y values:
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
plt.plot(days, celsius_values, "ob")
plt.show()
```



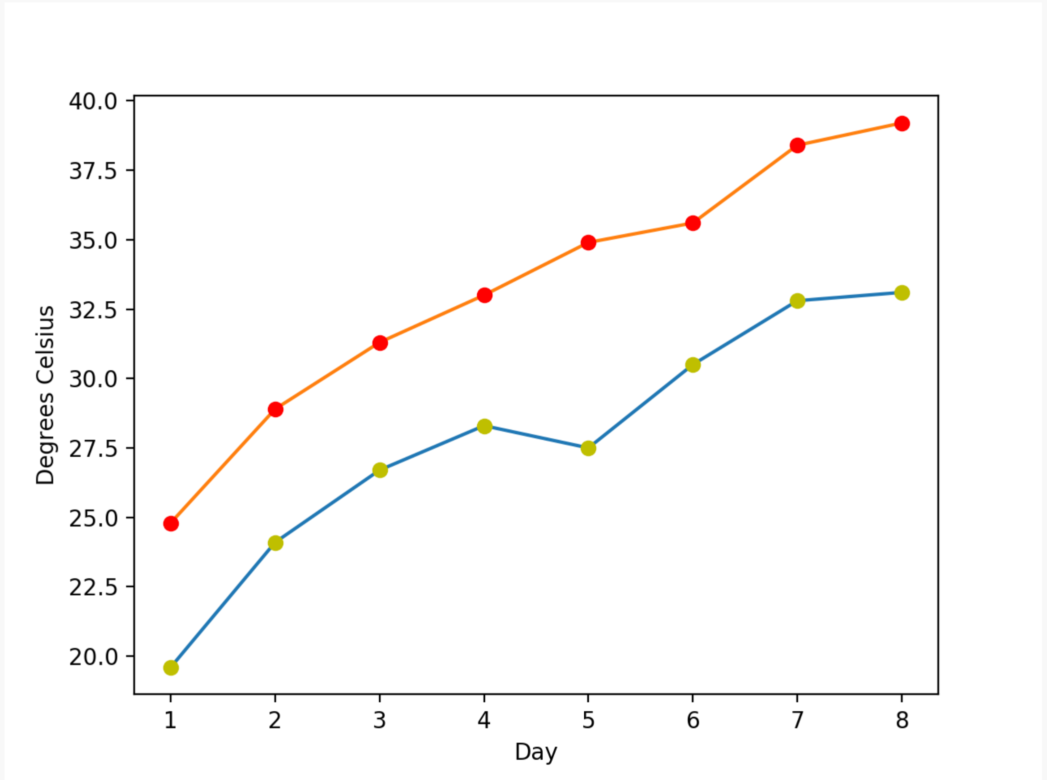
Labels on Axes

```
days = list(range(1,9))
celsius_values = [25.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8,
33.1]
plt.plot(days, celsius_values)
plt.xlabel('Day’)
plt.ylabel('Degrees Celsius’)
plt.show()
```



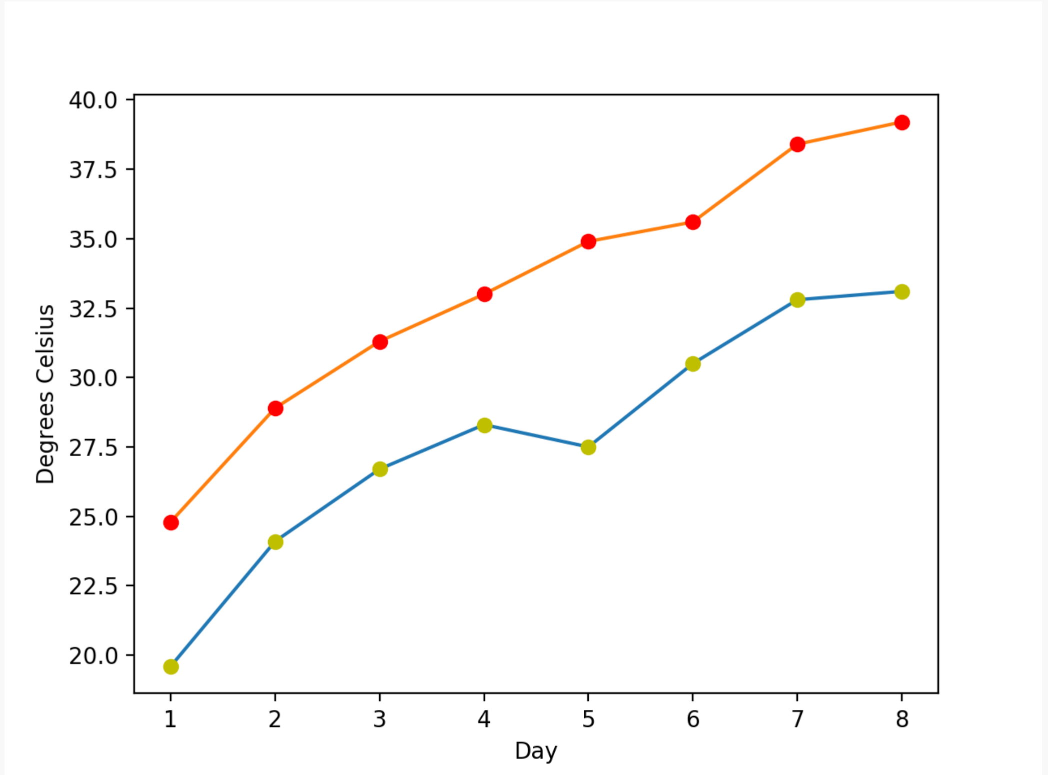
Arbitrary number of x and y

```
import matplotlib.pyplot as plt
days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]
plt.xlabel('Day’)
plt.ylabel('Degrees Celsius’)
plt.plot(days, celsius_min,
         days, celsius_min, "oy",
         days, celsius_max,
         days, celsius_max, "or")
plt.show()
```



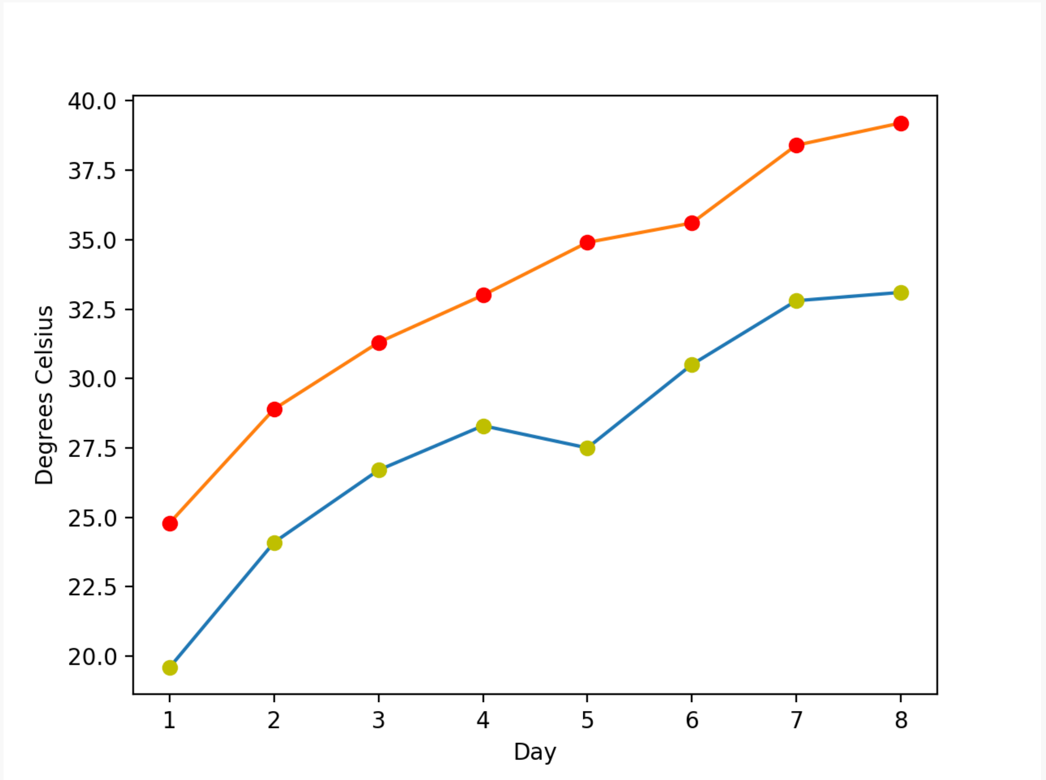
Arbitrary number of x and y

```
import matplotlib.pyplot as plt
days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]
plt.xlabel('Day’)
plt.ylabel('Degrees Celsius’)
plt.plot(days, celsius_min)
plt.plot(days, celsius_min, "oy")
plt.plot(days, celsius_max)
plt.plot(days, celsius_max, "or")
plt.show()
```



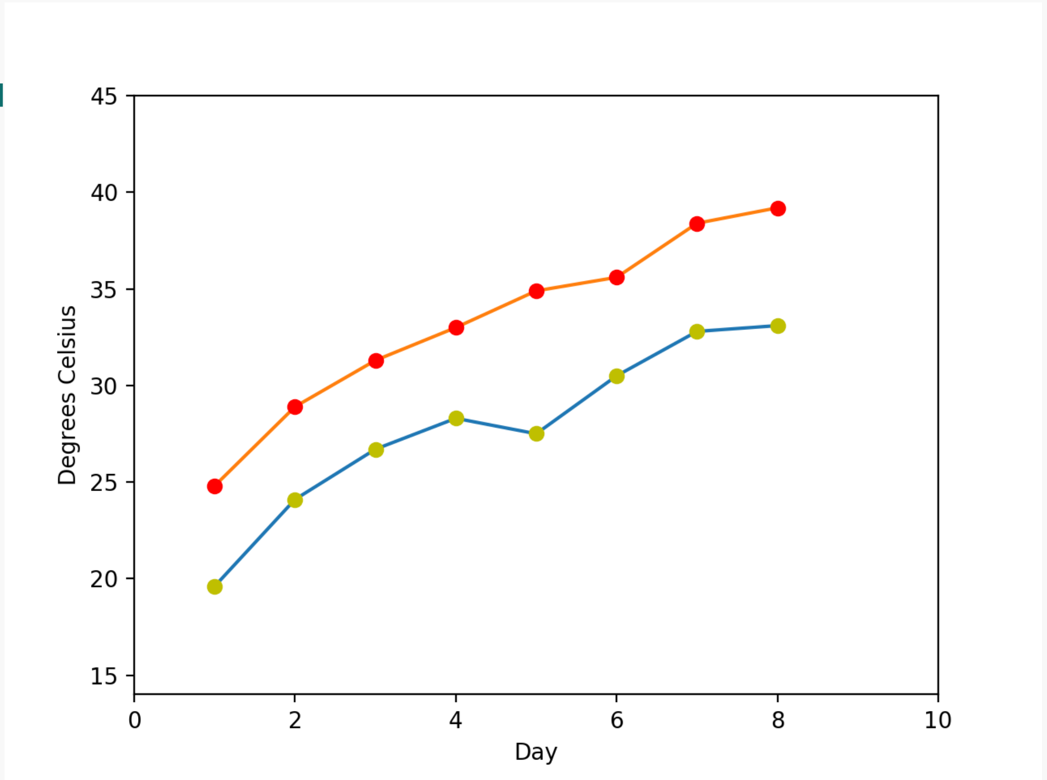
Checking and Defining the Range of Axes

```
print("The current limits for the axes are:")
print(plt.axis())
# (0.6499999999999999, 8.35, 18.62, 40.18)
```



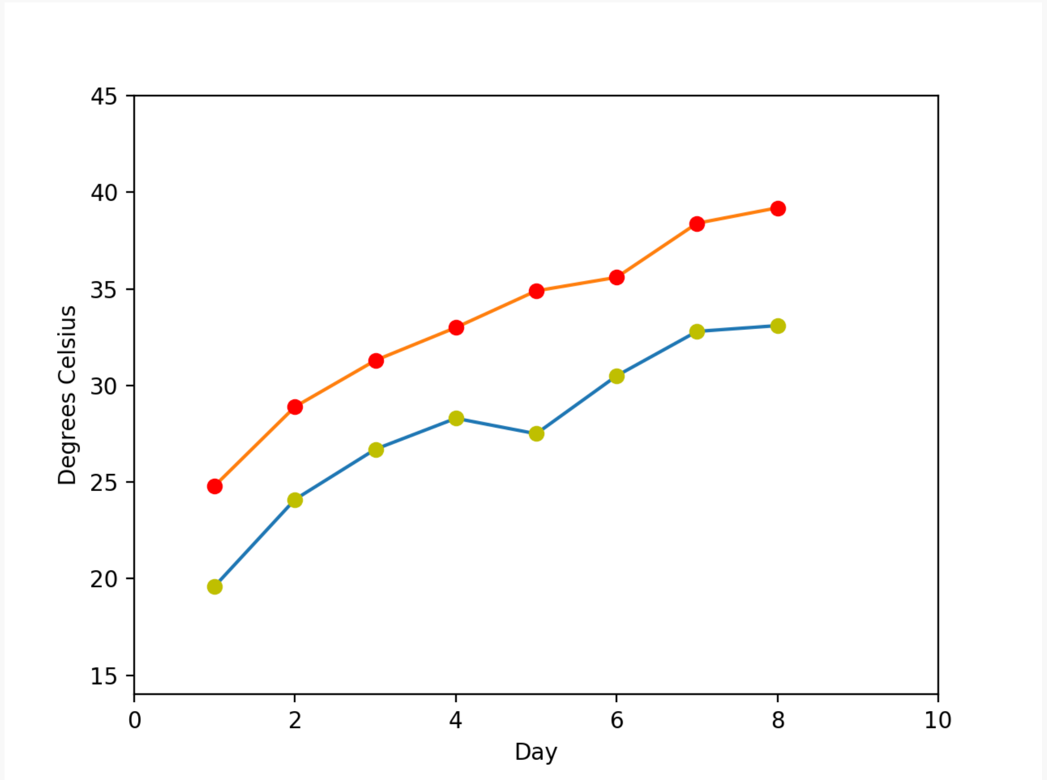
Checking and Defining the Range of Axes

```
print("The current limits for the axes are:")
print(plt.axis())
# (0.6499999999999999, 8.35, 18.62, 40.18)
print("We set the axes to the following valu
xmin, xmax, ymin, ymax = 0, 10, 14, 45
print(xmin, xmax, ymin, ymax)
# 0 10 14 45
plt.axis([xmin, xmax, ymin, ymax])
plt.show()
```



Checking and Defining the Range of Axes

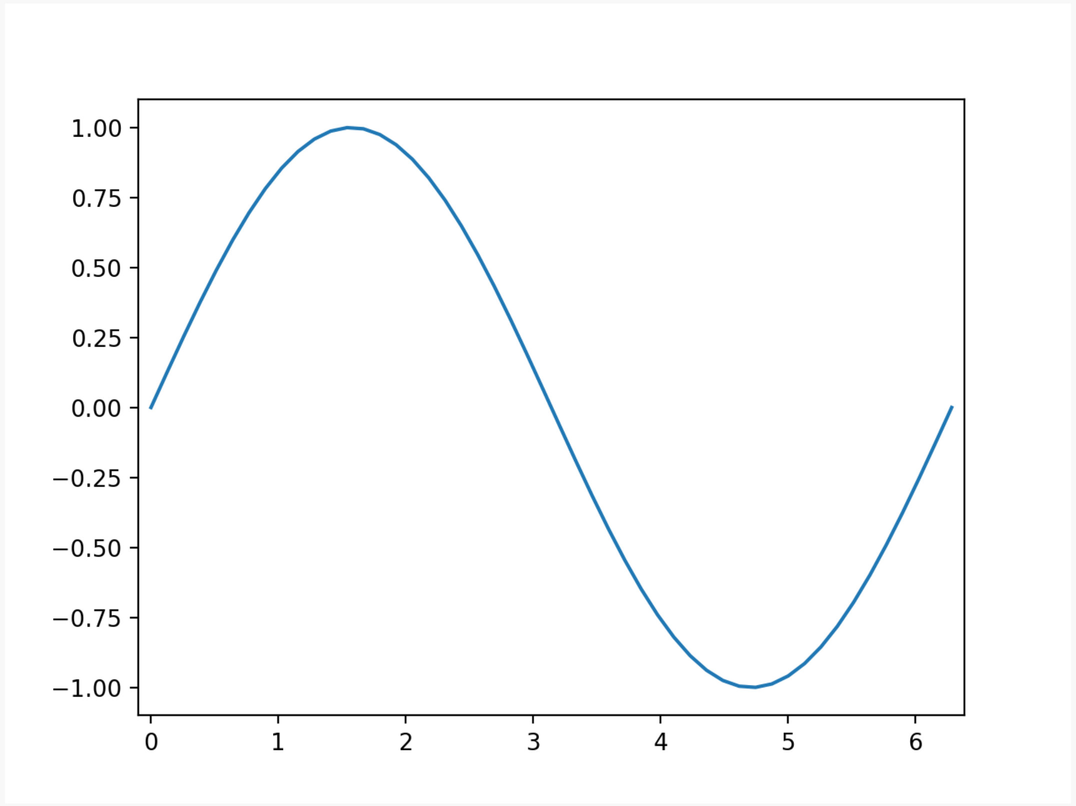
```
days = list(range(1,9))
celsius_min = [19.6, 24.1, 26.7, 28.3, 27.5, 30.5, 32.8, 33.1]
celsius_max = [24.8, 28.9, 31.3, 33.0, 34.9, 35.6, 38.4, 39.2]
plt.xlabel('Day’)
plt.ylabel('Degrees Celsius’)
plt.plot(days, celsius_min,
         days, celsius_min, "oy",
         days, celsius_max,
         days, celsius_max, "or")
plt.axis([0, 10, 18, 41])
plt.show()
```



"linspace" to Define X Values

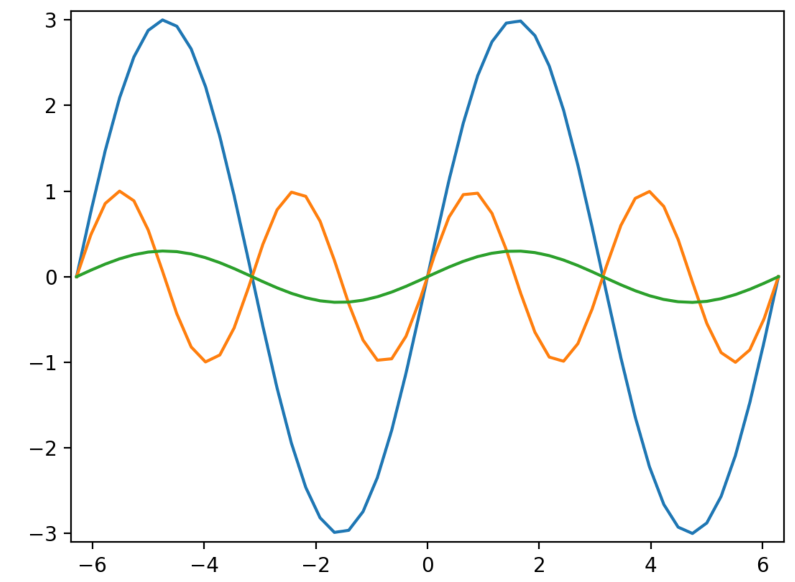
- Numpy.linspace can be used to create evenly spaced numbers over a specified interval.

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

X = np.linspace(0, 2 * np.pi, 50,
                endpoint=True)
F = np.sin(X)
plt.plot(X,F)
startx, endx = -0.1, 2*np.pi + 0.1
starty, endy = -1.1, 1.1
plt.axis([startx, endx, starty, endy])
plt.show()
```



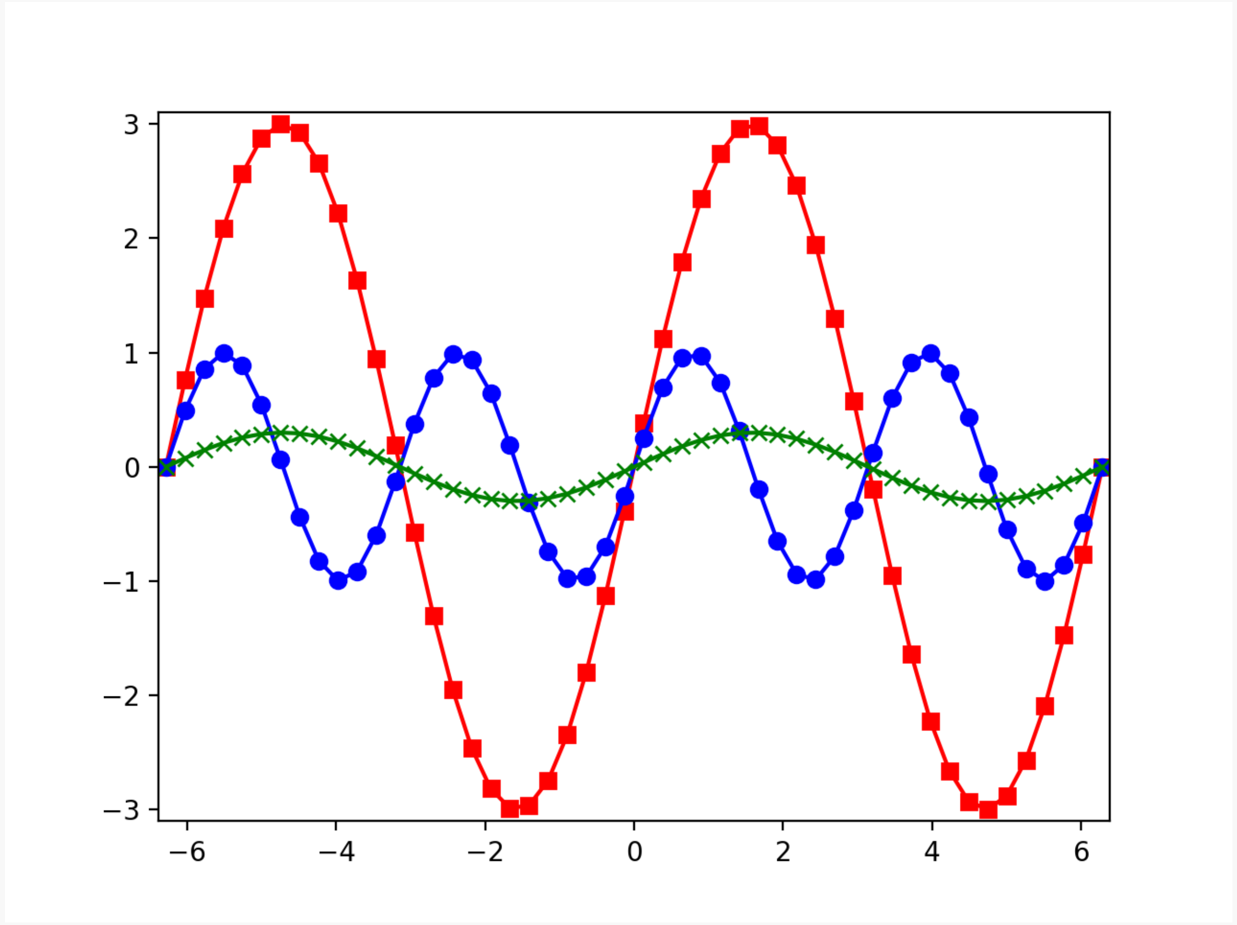
"linspace" to Define X Values

```
X = np.linspace(-2 * np.pi, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
startx, endx = -2 * np.pi - 0.1, 2*np.pi + 0.1
starty, endy = -3.1, 3.1
plt.axis([startx, endx, starty, endy])
plt.plot(X,F1)
plt.plot(X,F2)
plt.plot(X,F3)
plt.show()
```



More plots with discrete points…

```
plt.plot(X, F1, 'r’)
plt.plot(X, F1, 'rs’)
plt.plot(X, F2, 'b’)
plt.plot(X, F2, 'bo’)
plt.plot(X, F3, 'g’)
plt.plot(X, F3, 'gx’)
plt.show()
```

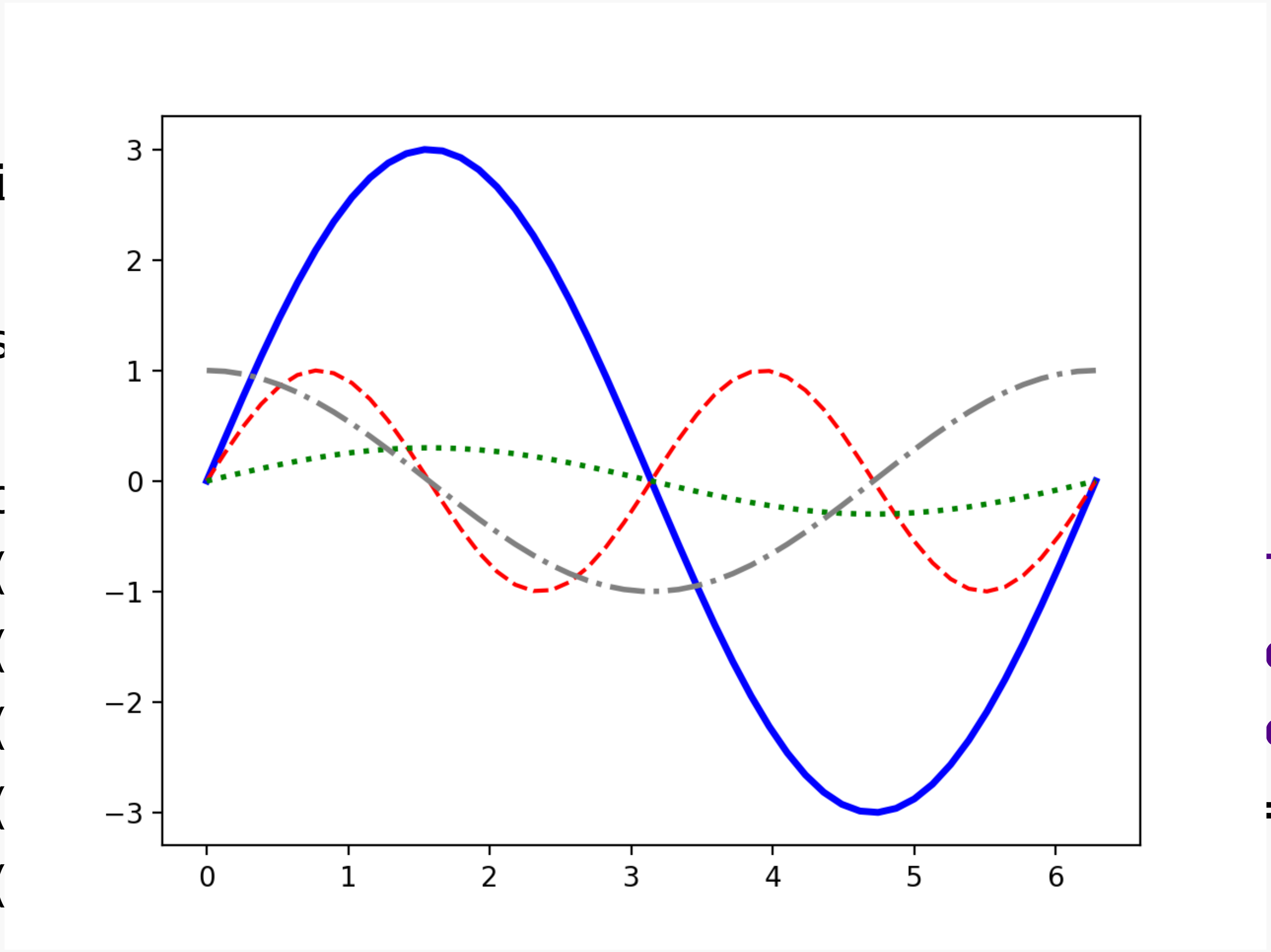


Changing the Line Style

```
X = np.linspace(0, 2 * np.pi, 50, endpoint=True)
F1 = 3 * np.sin(X)
F2 = np.sin(2*X)
F3 = 0.3 * np.sin(X)
F4 = np.cos(X)
plt.plot(X, F1, color="blue", linewidth=2.5, linestyle="-")
plt.plot(X, F2, color="red", linewidth=1.5, linestyle="--")
plt.plot(X, F3, color="green", linewidth=2, linestyle=":")
plt.plot(X, F4, color="grey", linewidth=2, linestyle="-.")
plt.show()
```

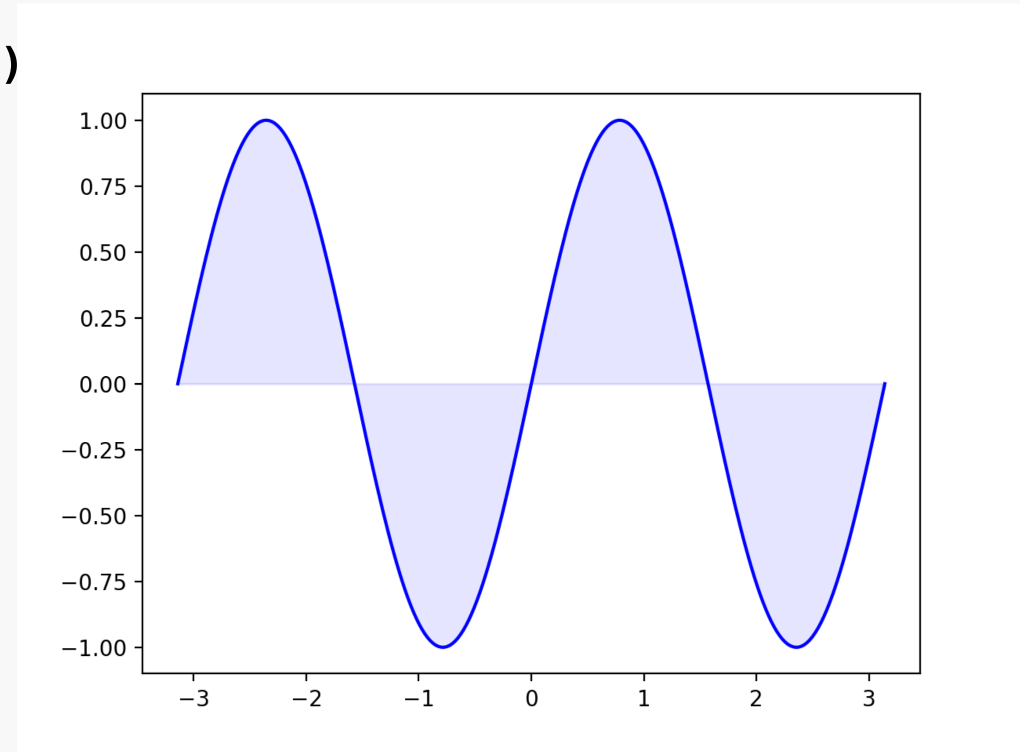
Changing the Line Style

```
X = np.li
F1 = 3 *
F2 = np.s
F3 = 0.3
F4 = np.c
plt.plot(                                         le="-")
plt.plot(                                         e="--")
plt.plot(                                         e=":")
plt.plot(                                         ="-.")
plt.show(
```



Shading Regions with fill_between()

```
X = np.linspace(-np.pi,np.pi,256,endpoint=True)
Y = np.sin(2*X)
plt.plot (X, Y, color='blue', alpha=1.00)
plt.fill_between(X, 0, Y, color='blue', alpha=.1)
plt.show()
```

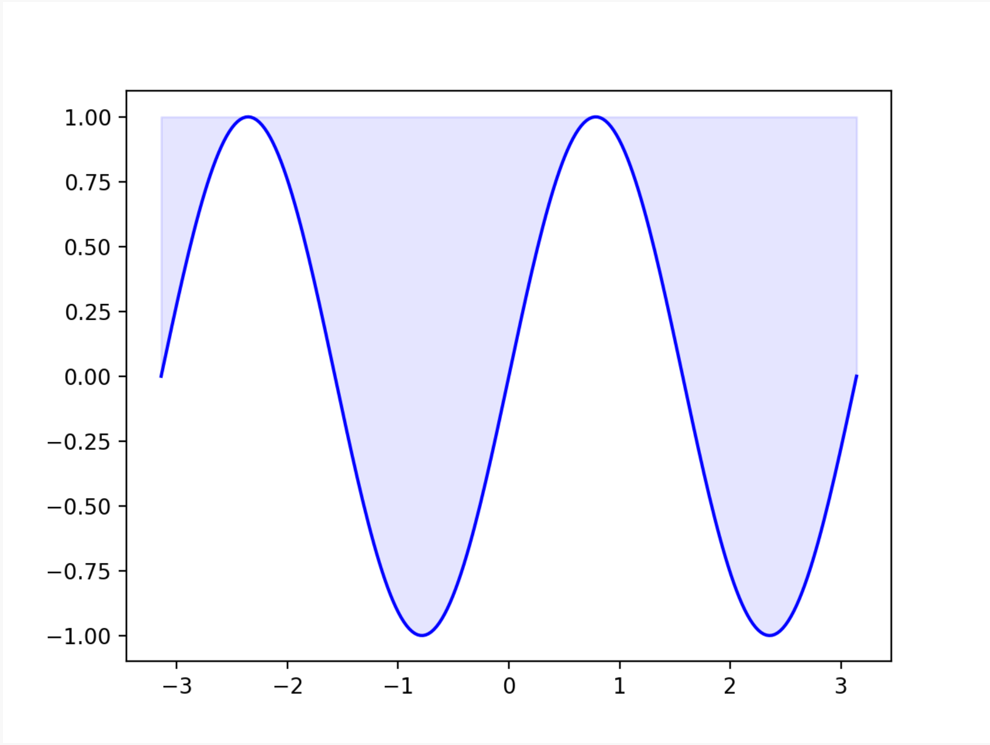


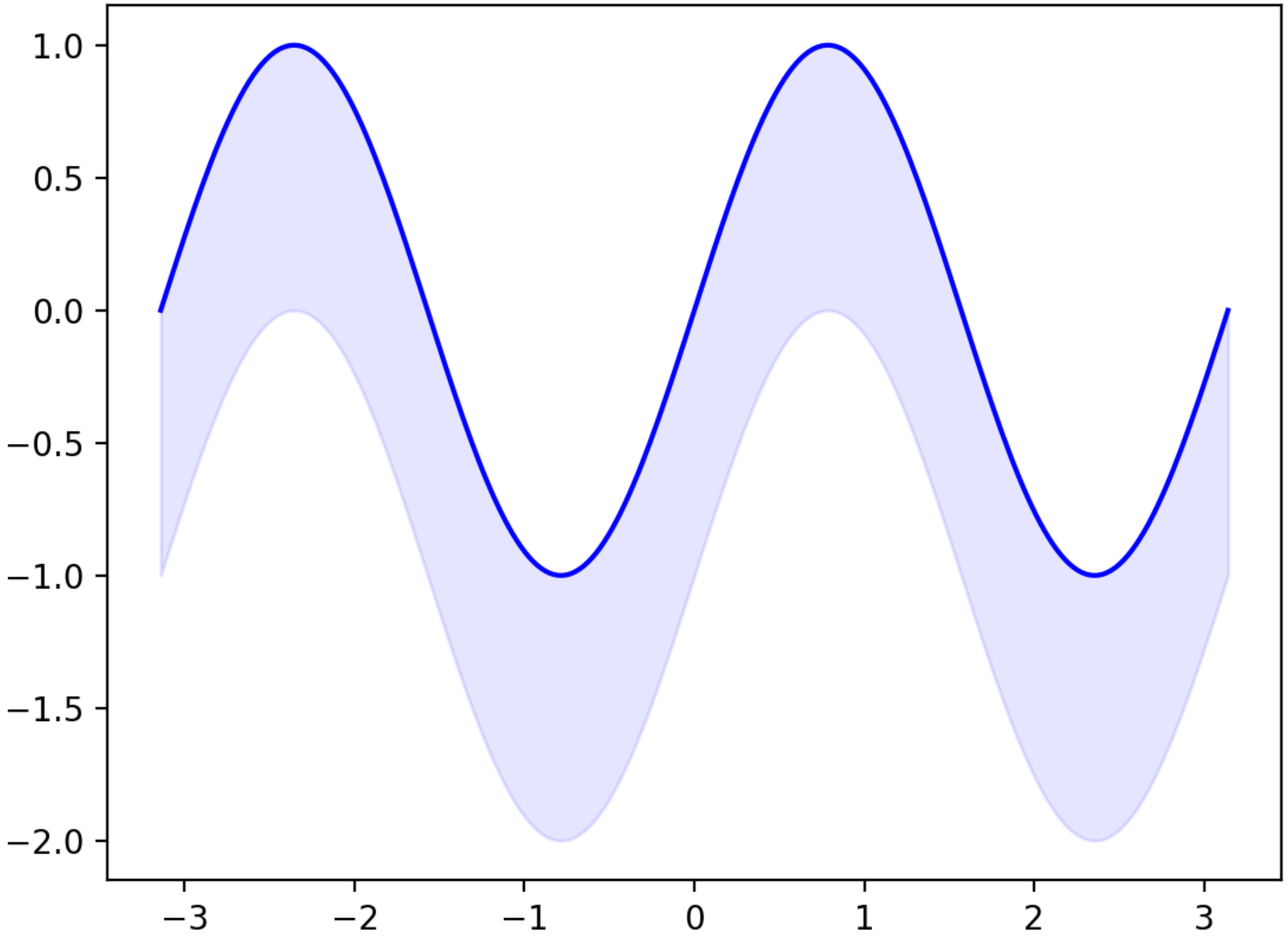
Shading Regions with fill_between()

- fill_between(x, y1, y2=0, where=None, interpolate=False, **kwargs)

x	An N-length array of the x data
y1	An N-length array (or scalar) of the y data
y2	An N-length array (or scalar) of the y data
where	If None, default to fill between everywhere. If not None, it is an N-length numpy boolean array and the fill will only happen over the regions where where==True.
interpolate	If True, interpolate between the two lines to find the precise point of intersection. Otherwise, the start and end points of the filled region will only occur on explicit values in the x array.
kwargs	Keyword args passed on to the PolyCollection

```
X = np.linspace(-np.pi,np.pi,256,endpoint=True)
Y = np.sin(2*X)
plt.plot (X, Y, color='blue', alpha=1.00)
plt.fill_between(X, Y, 1, color='blue', alpha=.1)
plt.show()
```



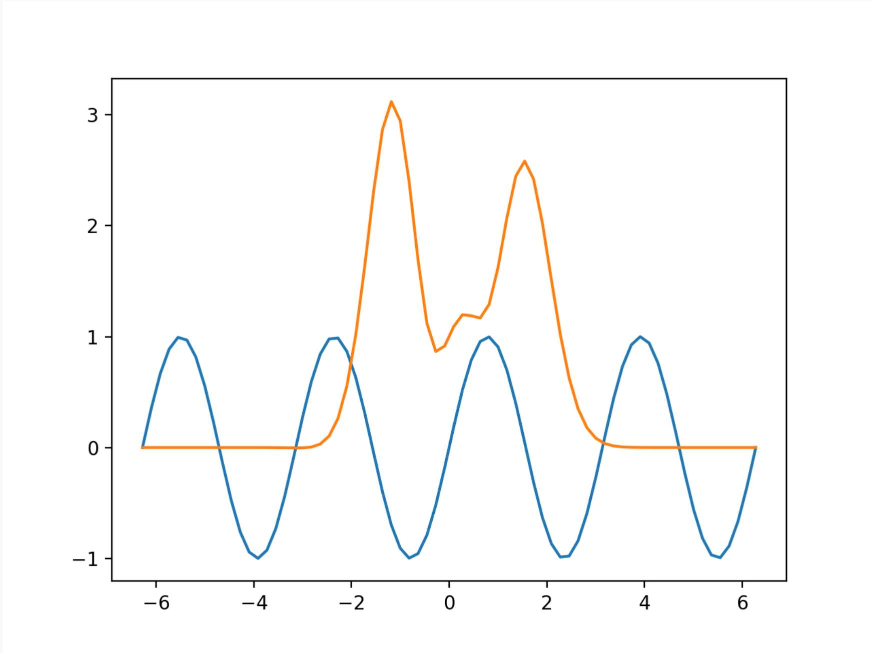

1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−3
−2
−1
0
1
2
3

Spines and ticks

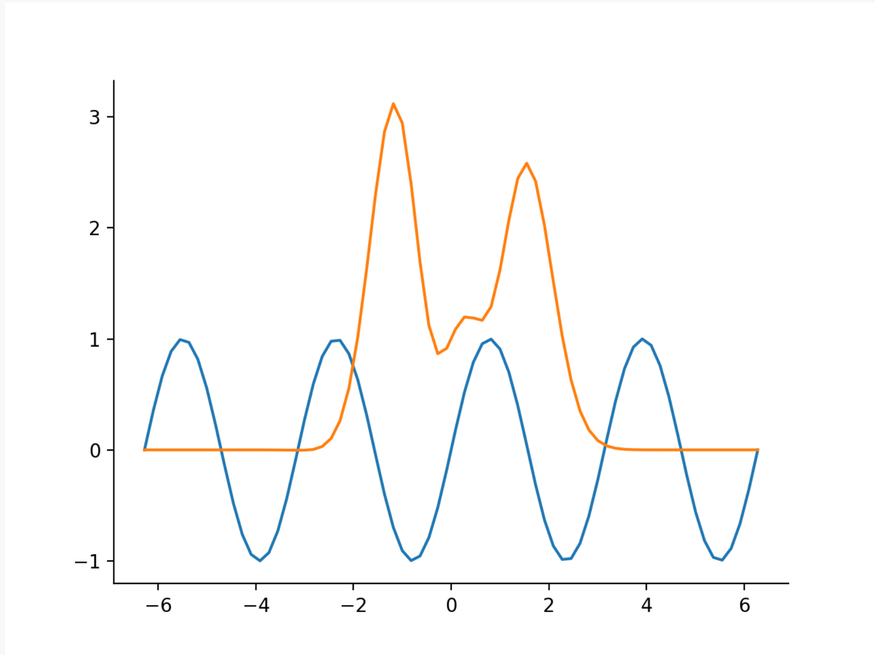
Moving the Border Lines and Polishing up the Axes Notations

- Spines in matplotlib are the lines connecting the axis tick marks and noting the boundaries of the data area.
- **We will demonstrate in the following that the spines can be placed at arbitrary positions.**
- gca function returns the current Axes instance on the current figure.

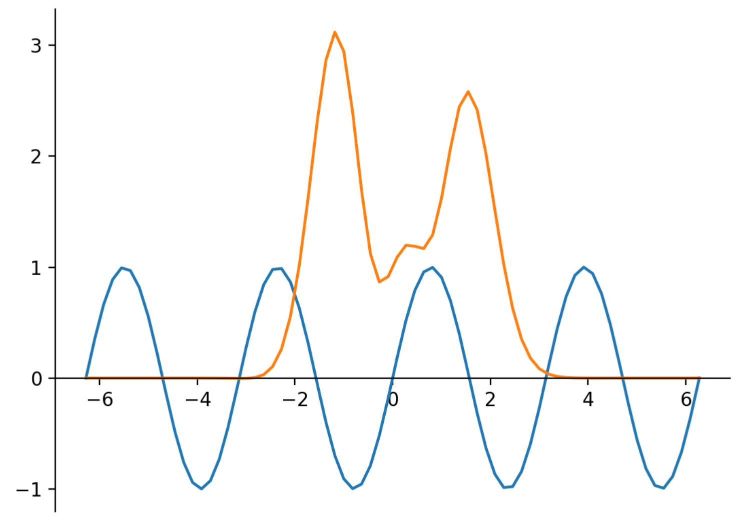
```
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(2* X)
F2 = (2*X**5 + 4*X**4 - 4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(-X**2)
plt.plot(X, F1)
plt.plot(X, F2)
plt.show()
```



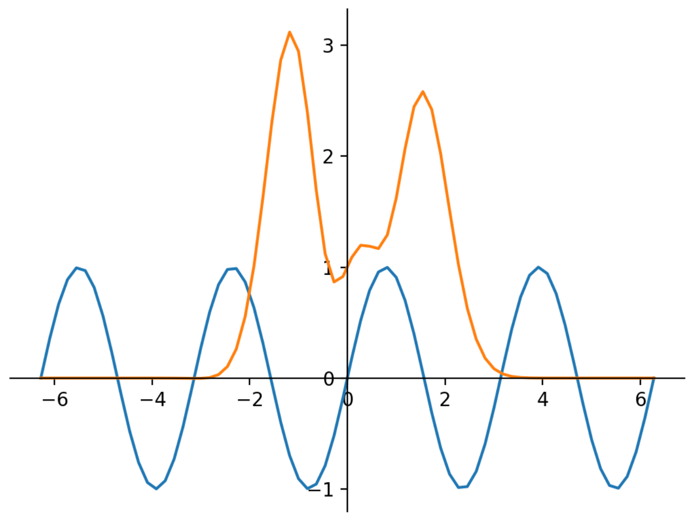
```
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(2* X)
F2 = (2*X**5 + 4*X**4 - 4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(-X**2)
# get the current axes, creating them if necessary:
ax = plt.gca()
# making the top and right spine invisible:
ax.spines['top'].set_color('none’)
ax.spines['right'].set_color('none')
plt.plot(X, F1)
plt.plot(X, F2)
plt.show()
```



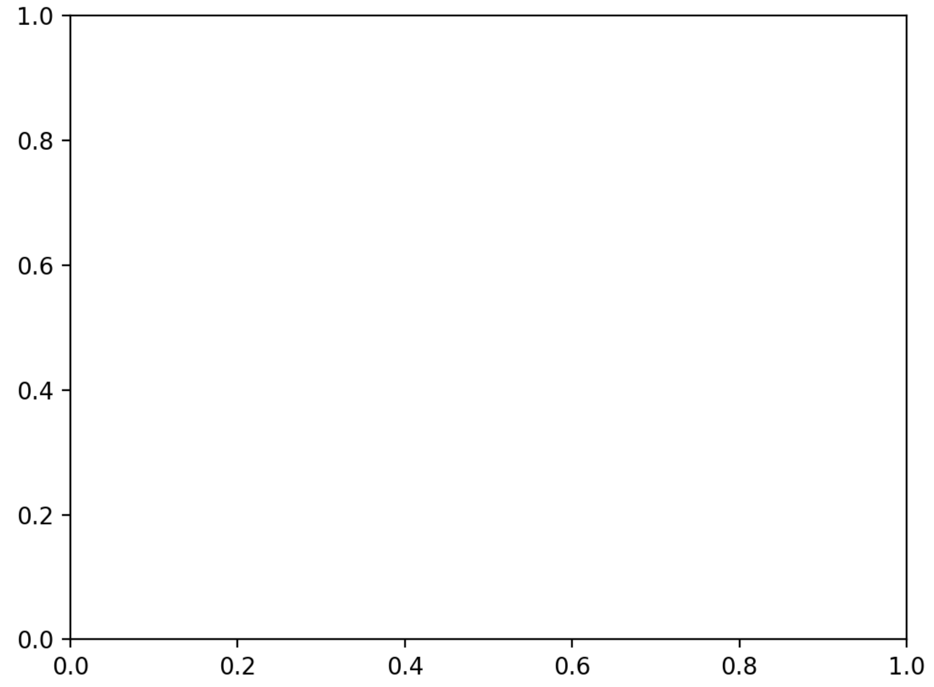
```
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(2* X)
F2 = (2*X**5 + 4*X**4 - 4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(-X**2)
# get the current axes, creating them if necessary:
ax = plt.gca()
# making the top and right spine invisible:
ax.spines['top'].set_color('none’)
ax.spines['right'].set_color('none’)
# moving bottom spine up to y=0 position:
ax.xaxis.set_ticks_position('bottom’)
ax.spines['bottom'].set_position(('data',0))
plt.plot(X, F1)
plt.plot(X, F2)
plt.show()
```



```
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(2* X)
F2 = (2*X**5 + 4*X**4 - 4.8*X**3 + 1.2*X**2 + X + 1)*np.exp(-X**2)
# get the current axes, creating them if necessary:
ax = plt.gca()
# making the top and right spine invisible:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
# moving bottom spine up to y=0 position:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
# moving left spine to the right to position x == 0:
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
plt.plot(X, F1)
plt.plot(X, F2)
plt.show()
```

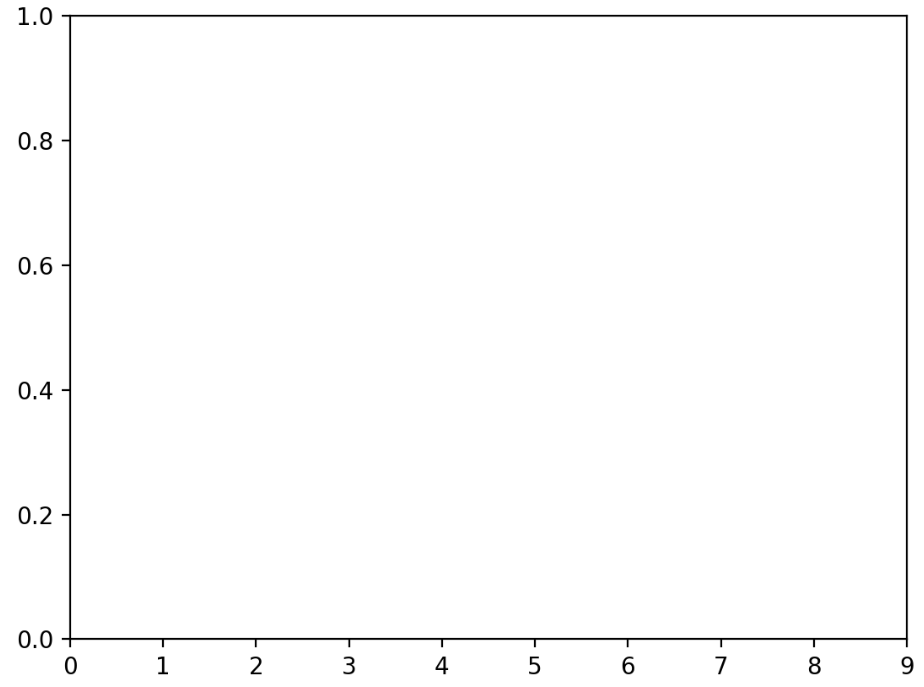


Customizing Ticks



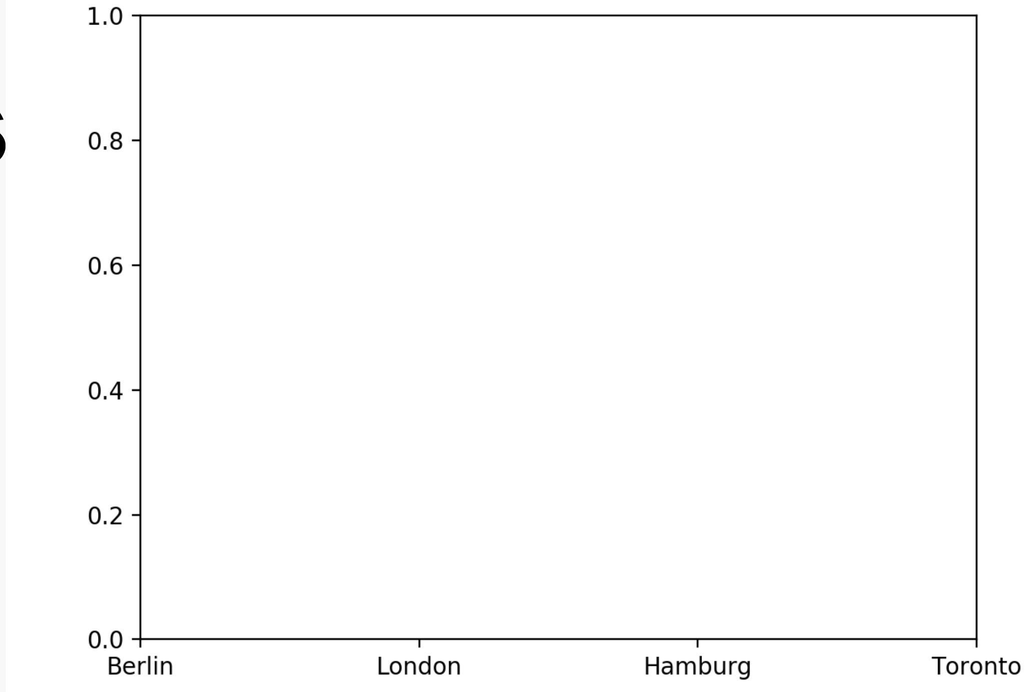
```
locs, labels = plt.xticks()
print(locs, labels)
locs, labels = plt.yticks()
print(locs, labels)
plt.show()
# [0.  0.2 0.4 0.6 0.8 1. ] <a list of 6 Text xticklabel objects>
# [0.  0.2 0.4 0.6 0.8 1. ] <a list of 6 Text yticklabel objects>
```

Customizing Ticks



```
plt.xticks( np.arange(10) )
locs, labels = plt.xticks()
print(locs, labels)
plt.show()
# [0 1 2 3 4 5 6 7 8 9] <a list of 10 Text xticklabel objects>
```

Customizing Ticks

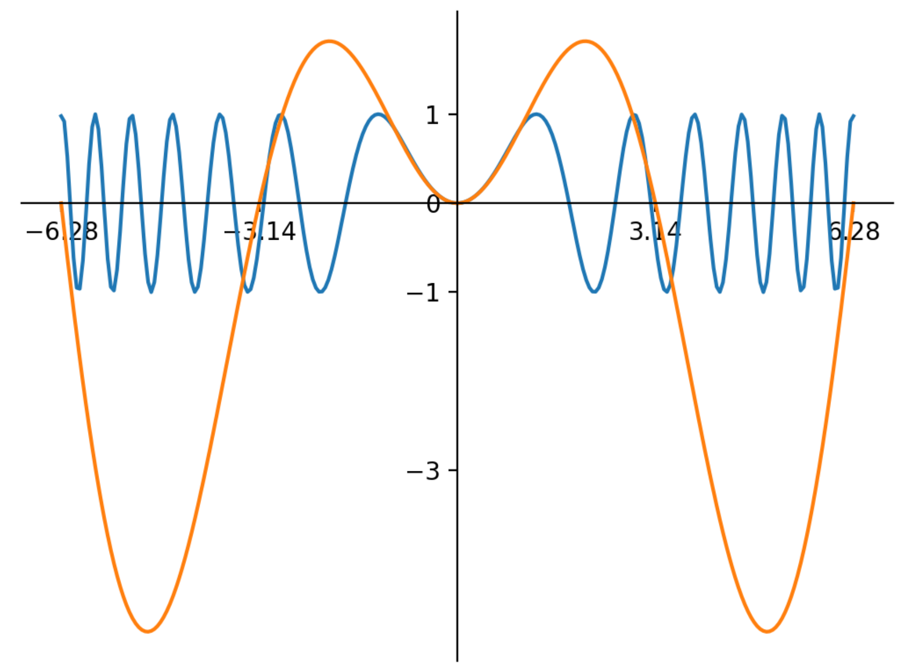


```
plt.xticks( np.arange(4),
            ('Berlin', 'London', 'Hamburg', 'Toronto’) )
plt.show()
```

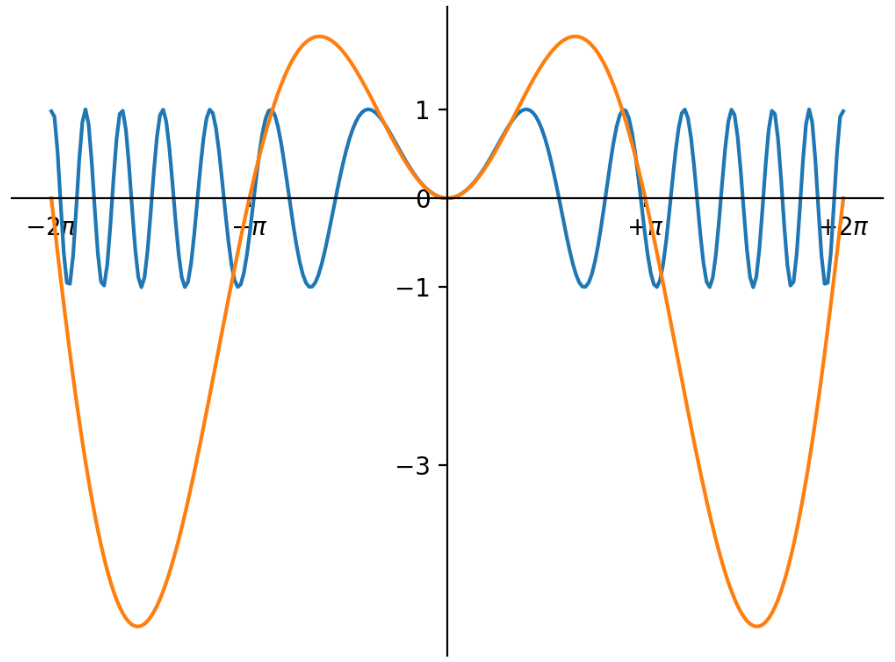
Example

- When working with trigonometric functions, most people might consider factors of Pi to be more appropriate for the X axis than the integer labels.

```
X = np.linspace(-2 * np.pi, 2 * np.pi, 256, endpoint=True)
F1 = np.sin(X**2)
F2 = X * np.sin(X)
# get the current axes, creating them if necessary:
ax = plt.gca()
# making the top and right spine invisible:
ax.spines['top'].set_color('none')
ax.spines['right'].set_color('none')
# moving bottom spine up to y=0 position:
ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0))
# moving left spine to position x == 0:
ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))
plt.xticks([-6.28, -3.14, 3.14, 6.28])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1)
plt.plot(X, F2)
plt.show()
```

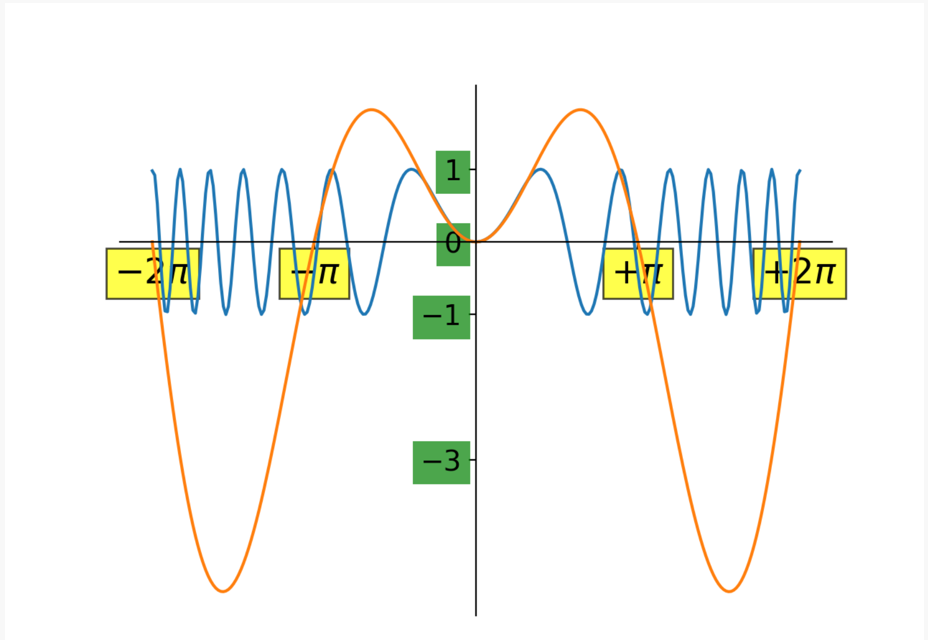


```
X = np.linspace(-2 * np.pi, 2 * np.pi, 256, endpoint=True)
F1 = np.sin(X**2)
F2 = X * np.sin(X)
# get the current axes, creating them if necessary:
ax = plt.gca()
# making the top and right spine invisible:
ax.spines['top'].set_color('none’)
ax.spines['right'].set_color('none’)
# moving bottom spine up to y=0 position:
ax.xaxis.set_ticks_position('bottom’)
ax.spines['bottom'].set_position(('data',0))
# moving left spine to position x == 0:
ax.yaxis.set_ticks_position('left’)
ax.spines['left'].set_position(('data',0))
plt.xticks([-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$’,
         r'$+\pi$', r'$+2\pi$'])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1)
plt.plot(X, F2)
plt.show()
```



Adjusting the tick labels

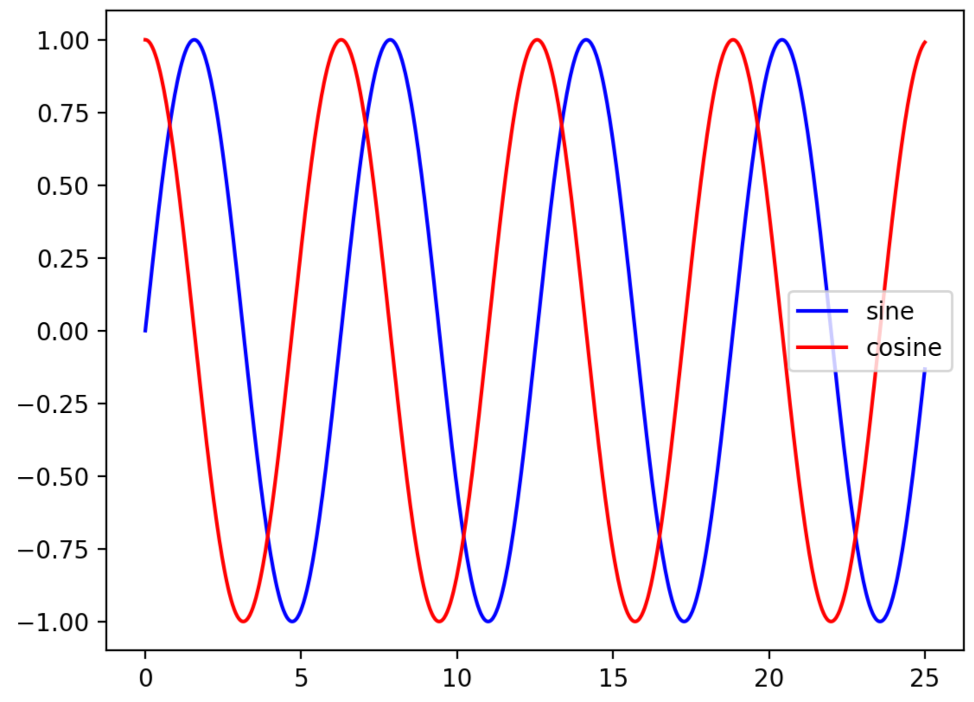
```
for xtick in ax.get_xticklabels():
    xtick.set_fontsize(18)
    xtick.set_bbox(dict(facecolor='yellow', edgecolor='black', alpha=0.7 ))
for ytick in ax.get_yticklabels():
    ytick.set_fontsize(14)
    ytick.set_bbox(dict(facecolor='green', edgecolor='None', alpha=0.7 ))
```



Legend and title

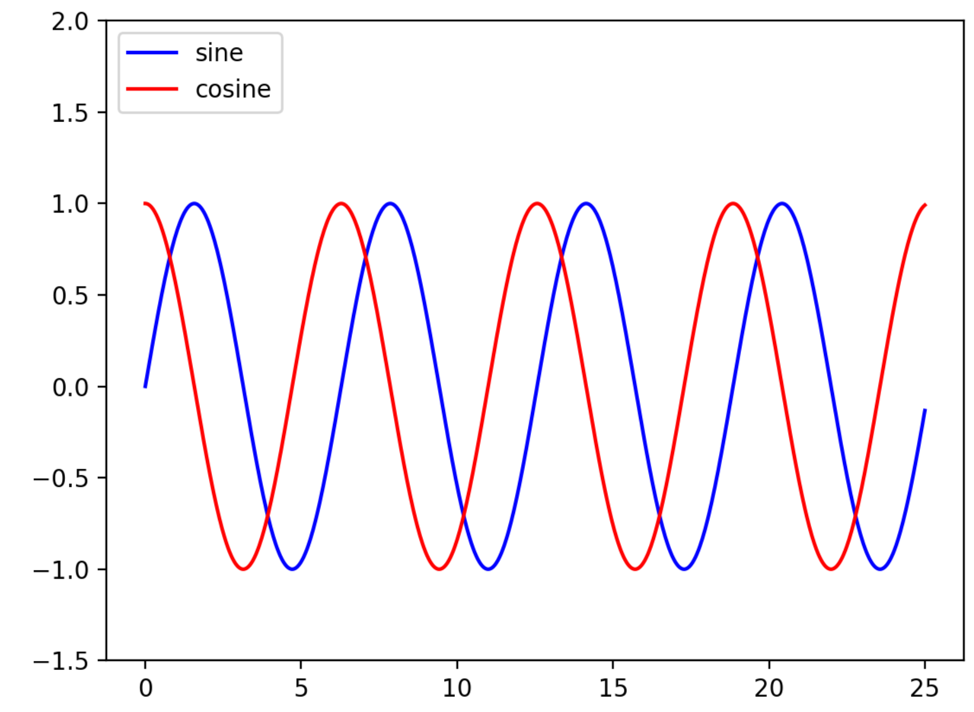
Legend

```
x = np.linspace(0, 25, 1000)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1, '-b', label='sine’)
plt.plot(x, y2, '-r', label='cosine’)
plt.legend()
plt.show()
```



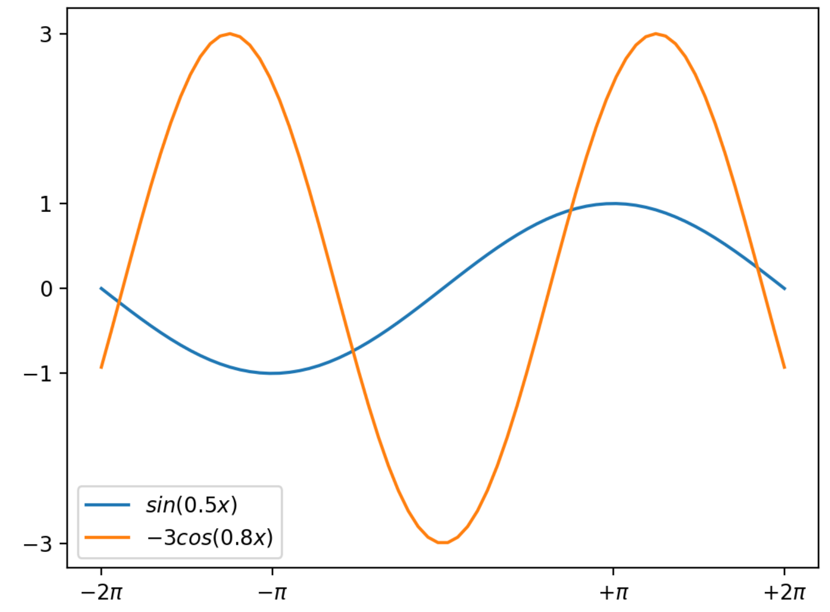
Legend

```
x = np.linspace(0, 25, 1000)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1, '-b', label='sine’)
plt.plot(x, y2, '-r', label='cosine’)
plt.legend(loc='upper left’) # upper/lower and left/right
plt.ylim(-1.5, 2) # To make some space
plt.show()
```



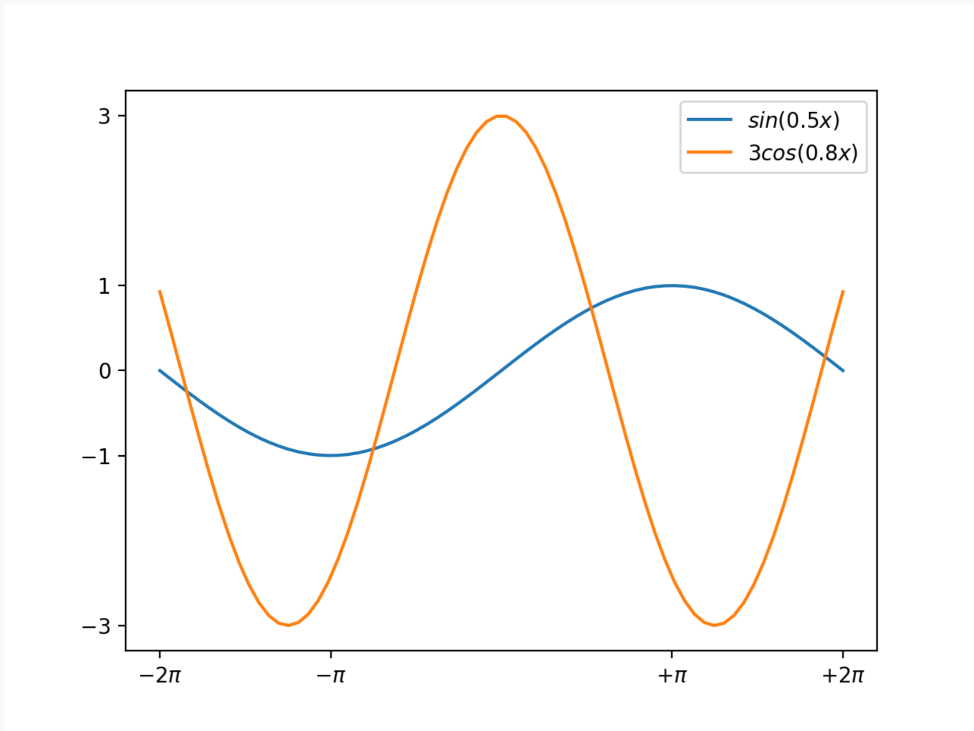
Legend – loc=‘best’

```
X = np.linspace(−2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = −3 * np.cos(0.8*X)
plt.xticks( [−6.28, −3.14, 3.14, 6.28],
        [r'$−2\pi$', r'$−\pi$’,
         r'$+\pi$', r'$+2\pi$’])
plt.yticks([−3, −1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$−3 cos(0.8x)$")
plt.legend(loc='best’)
plt.show()
```



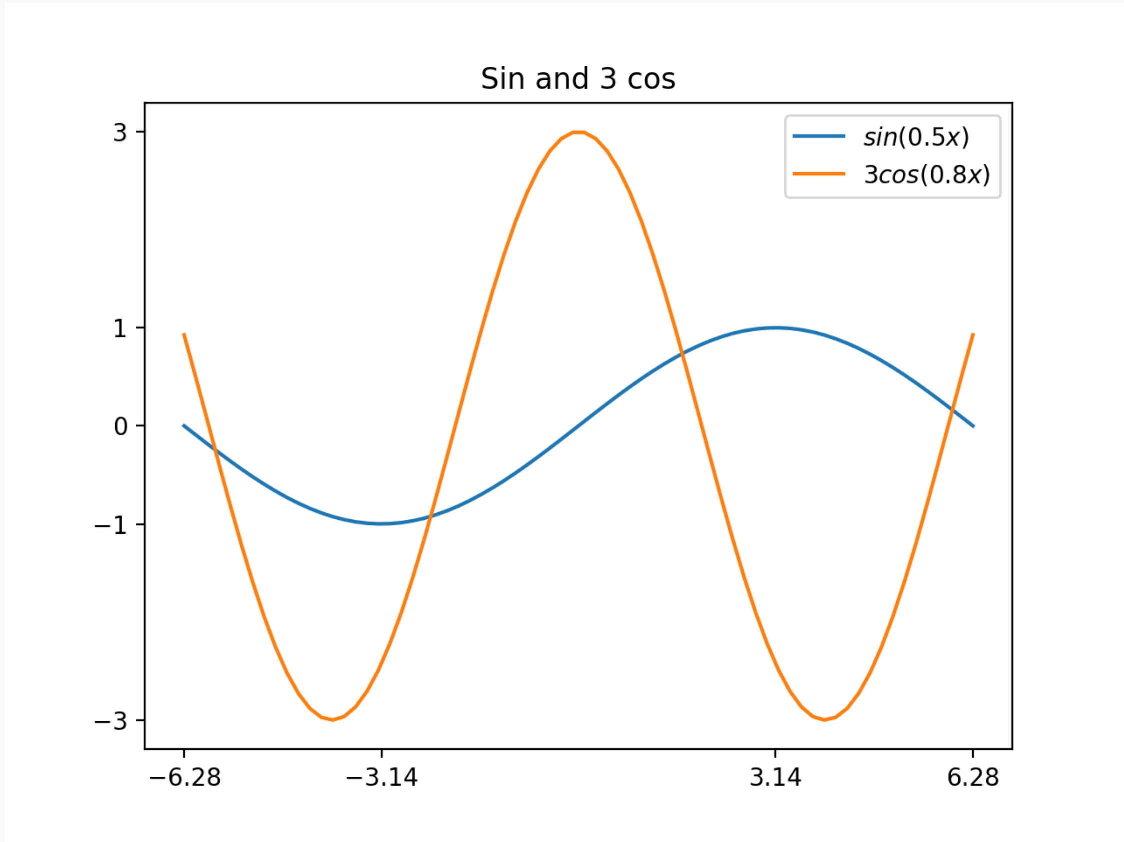
Legend – loc=‘best’

```
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = 3 * np.cos(0.8*X)
plt.xticks( [-6.28, -3.14, 3.14, 6.28],
        [r'$-2\pi$', r'$-\pi$’,
         r'$+\pi$’,r'$+2\pi$’])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$3 cos(0.8x)$")
plt.legend(loc='best’)
plt.show()
```



Title

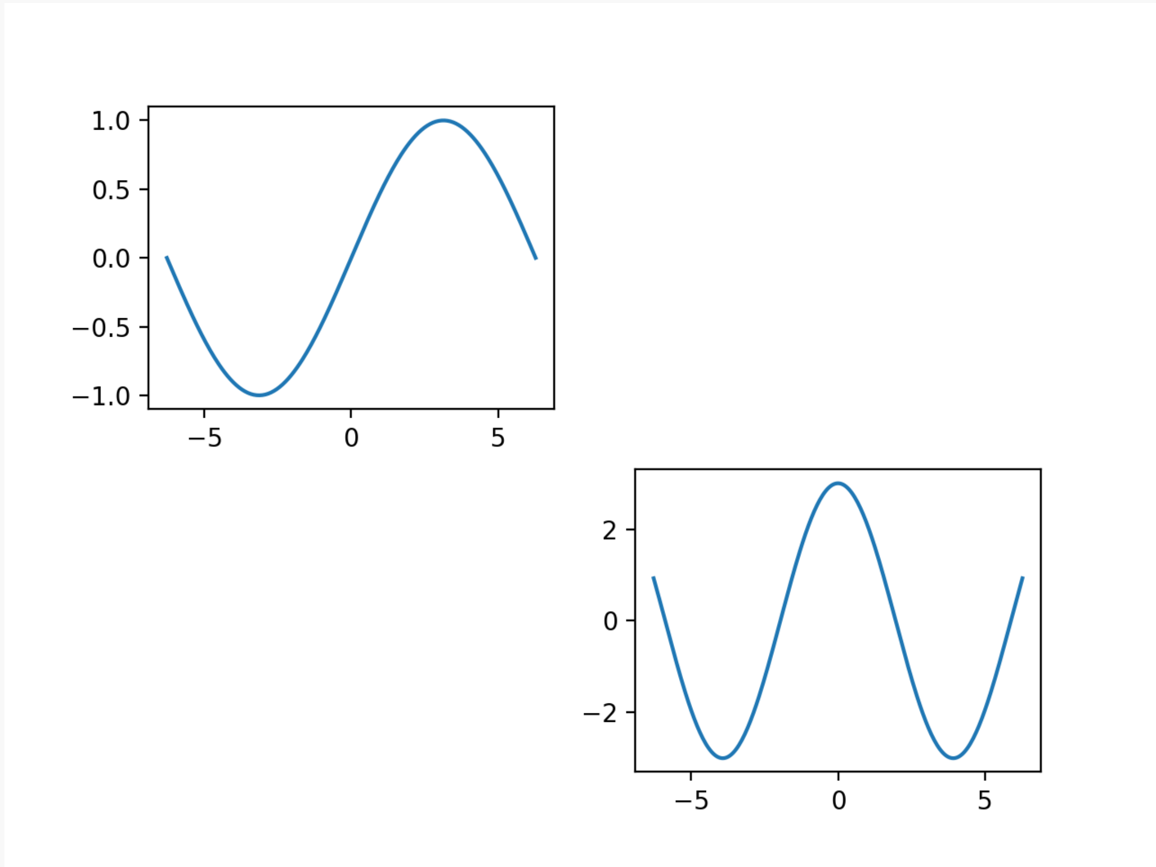
```
X = np.linspace(-2 * np.pi, 2 * np.pi, 70, endpoint=True)
F1 = np.sin(0.5*X)
F2 = 3 * np.cos(0.8*X)
plt.xticks( [-6.28, -3.14, 3.14, 6.28])
plt.yticks([-3, -1, 0, +1, 3])
plt.plot(X, F1, label="$sin(0.5x)$")
plt.plot(X, F2, label="$3 cos(0.8x)$")
plt.legend(loc='best’)
plt.title("Sin and 3 cos")
plt.show()
```



Multiple Plots, Grid Lines, Saving Figures and Loading Data From Files

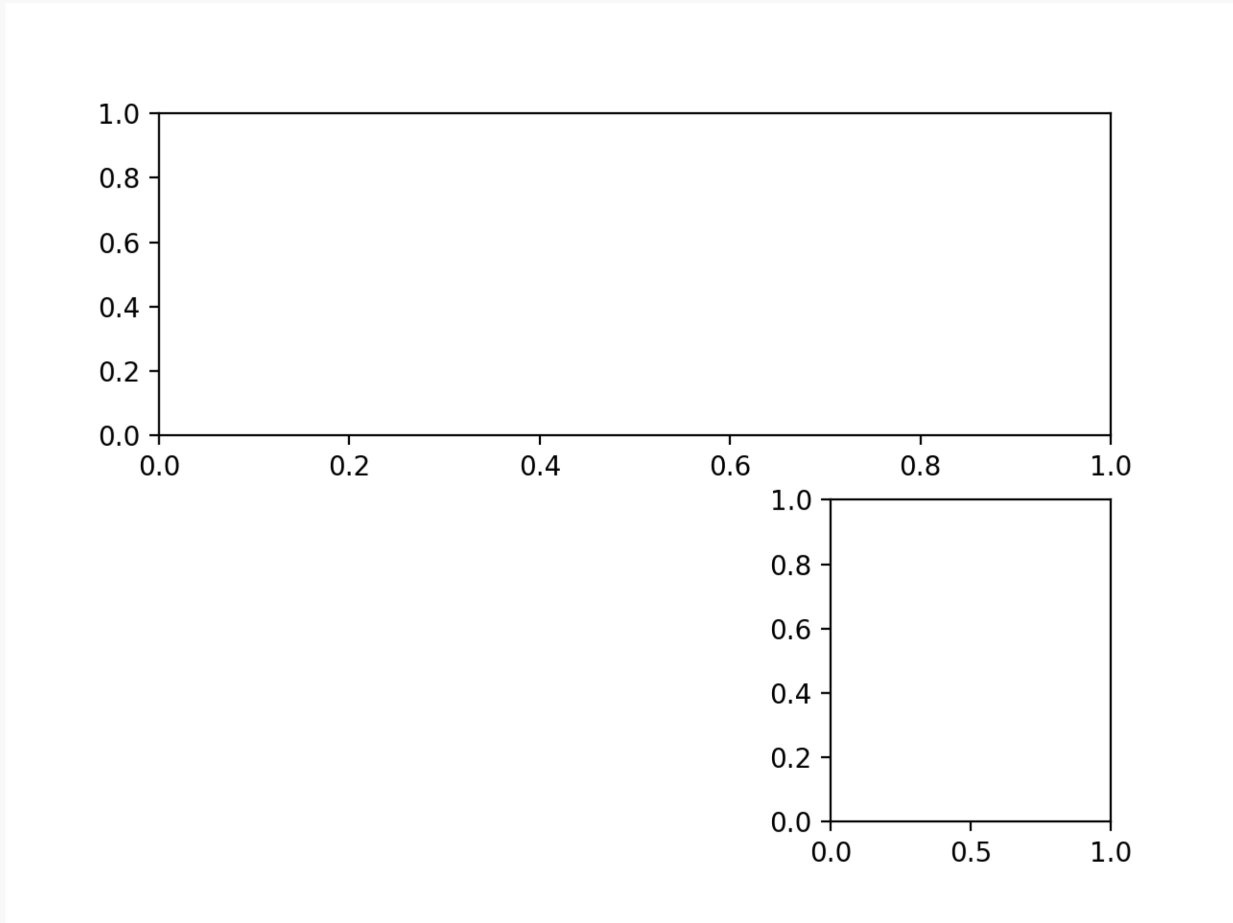
Subplots

```
X = np.linspace(-2 * np.pi, 2 * np.pi, 256, endpoint=True)
F1 = np.sin(0.5*X)
F2 = 3 * np.cos(0.8*X)
plt.subplot(2,2,1)
plt.plot(X, F1)
plt.subplot(2,2,4)
plt.plot(X, F2)
plt.show()
```

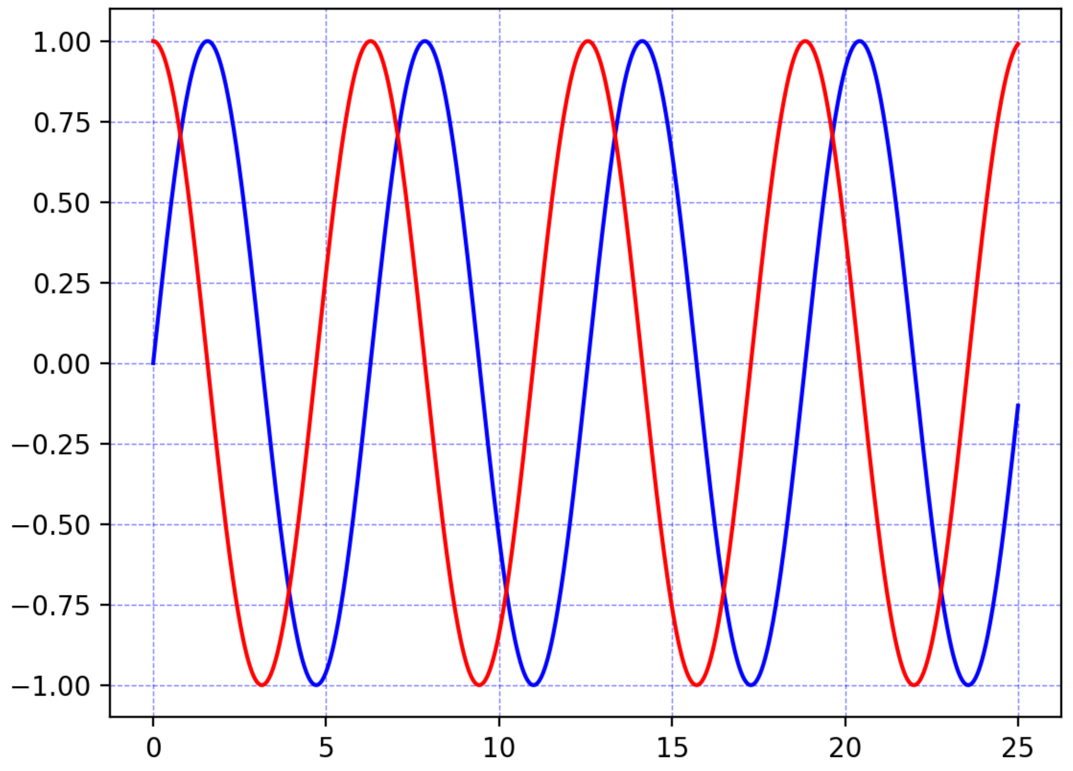


Subplots

```
plt.subplot(2,1,1)
plt.subplot(2,3,6)
plt.show()
```



Grid Lines



```
x = np.linspace(0, 25, 1000)
y1 = np.sin(x)
y2 = np.cos(x)
plt.plot(x, y1, 'b')
plt.plot(x, y2, 'r')
plt.grid(color='b', alpha=0.5, linestyle='dashed', linewidth=0.5)
plt.show()
```

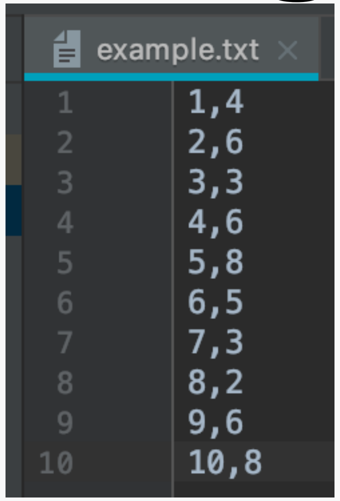
Saving figures

- Output can be generated in the formats PNG, JPG, EPS, SVG, PGF and PDF.
- It is possible to optionally specify the DPI (Dots per Inch).

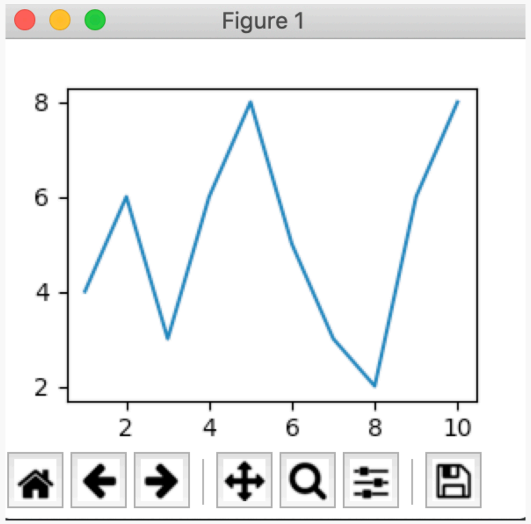
```
plt.savefig("filename.png”, dpi=200)
```

Loading data from files using numpy

```
import numpy as np

x, y = np.loadtxt('example.txt', delimiter=',', unpack=True)

plt.plot(x, y)
```

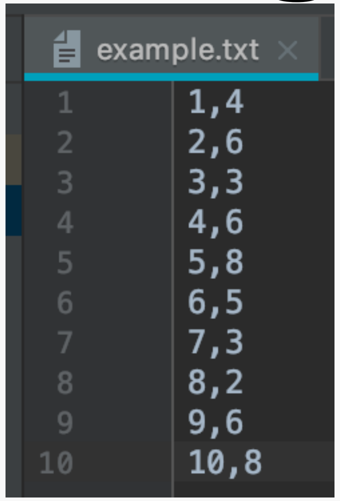

example.txt

```
1,4
2,6
3,3
4,6
5,8
6,5
7,3
8,2
9,6
10,8
```

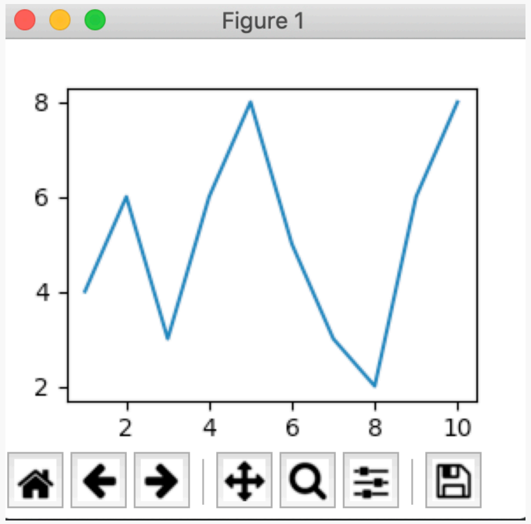


Loading data from files using csv

```
import matplotlib.pyplot as plt
import csv


x = []
y = []

with open('example.txt', 'r') as csvfile:
    plots = csv.reader(csvfile, delimiter=',')
    for row in plots:
        x.append(int(row[0]))
        y.append(int(row[1]))

plt.plot(x, y)
```


example.txt

```
1,4
2,6
3,3
4,6
5,8
6,5
7,3
8,2
9,6
10,8
```



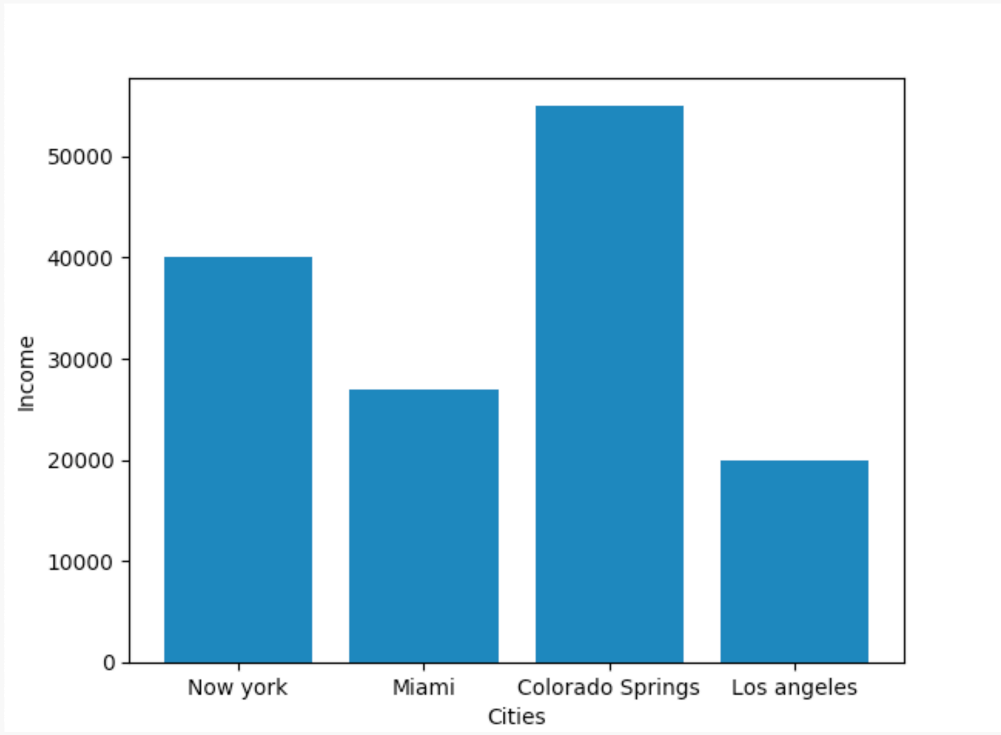
Damià Fuentes

Other types of plots

Bar chart

- A graph drawn using rectangular bars to show how large each value is.

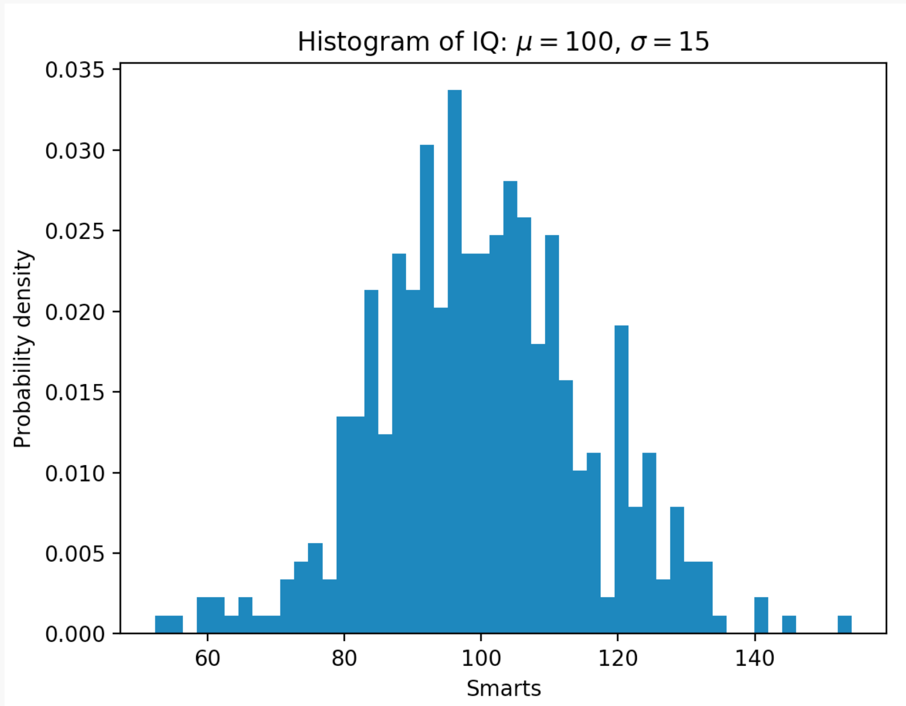
```
x = ['Now york', 'Miami', 'Colorado Springs', 'Los angeles’]
y = [40000, 27000, 55000, 20000]
plt.bar(x, y)
plt.show()
```



Histogram

- A graphical display where the data is grouped into ranges (such as ”20 to 39", ”40 to 69", etc), and then plotted as bars. Similar to a Bar Graph, but in a Histogram each bar is for a range of data

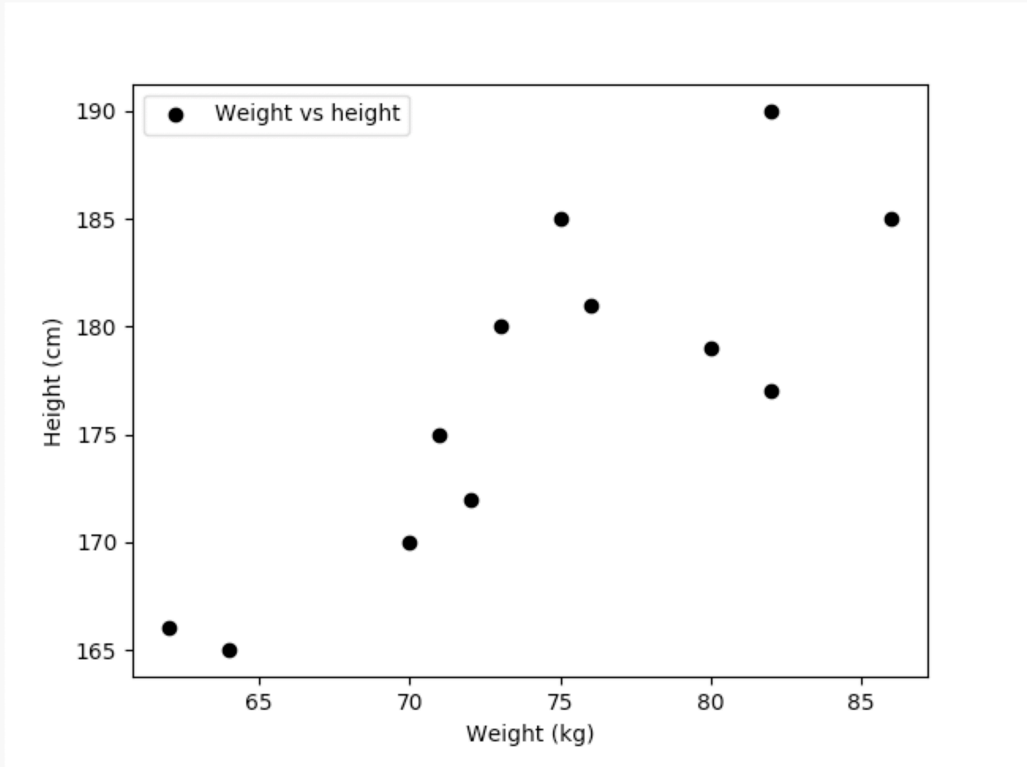
```
mu = 100  # mean of distribution
sigma = 15  # standard deviation of distribution
x = mu + sigma * np.random.randn(500)
# num_bins = [0, 80, 100, 120, 200] # For specific bins
num_bins = 50  # For default bins
# the histogram of the data, density = 1 for normalized form
plt.hist(x, num_bins, density=1)
plt.xlabel('Smarts’)
plt.ylabel('Probability density’)
plt.title(r'Histogram of IQ: $\mu=100$, $\sigma=15$’)
plt.show()
```



Scatter plot

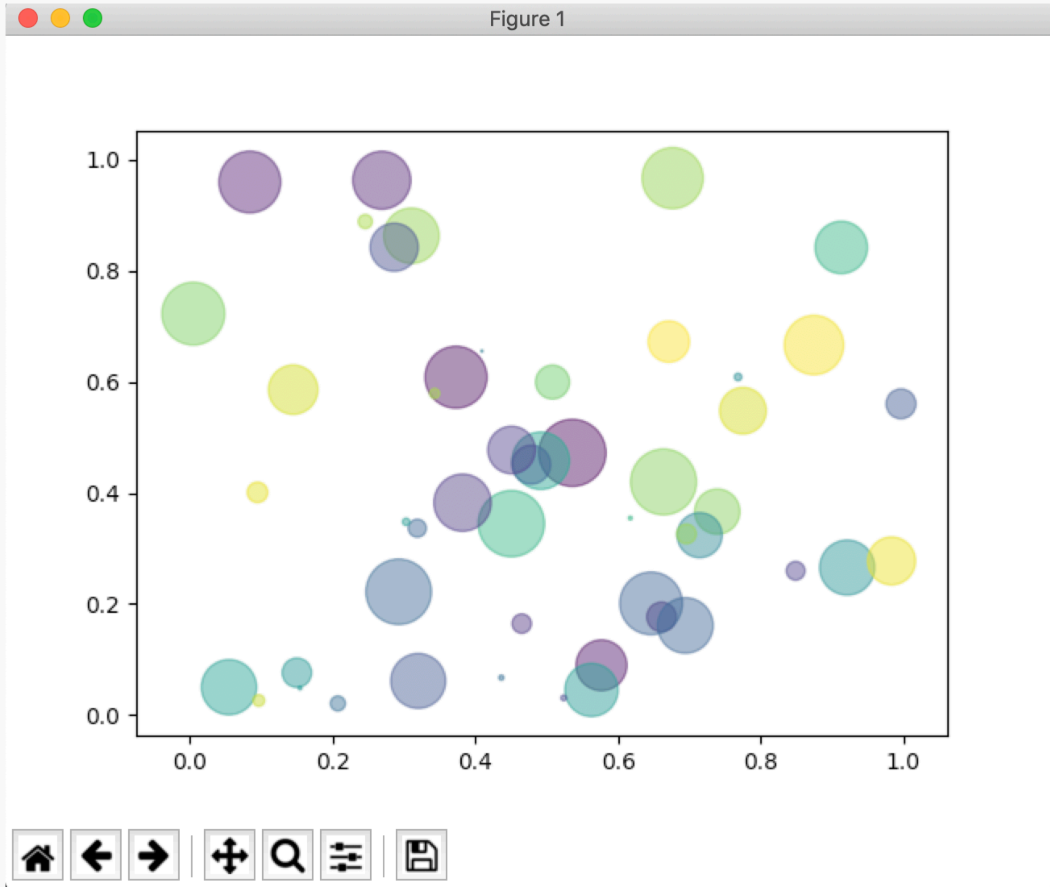
- A graph of plotted points that show the relationship between two sets of data.

```
a = np.array([[64, 165],
              [70, 170],
              [73, 180],
              [80, 179],
              [82, 190],
              [75, 185],
              [62, 166],
              [72, 172],
              [71, 175],
              [82, 177],
              [86, 185],
              [76, 181]])
plt.scatter(a[:, 0], a[:, 1],
            label='Weight vs height', marker='o', s=10)
plt.legend()
plt.show()
```



More complex scatter plot

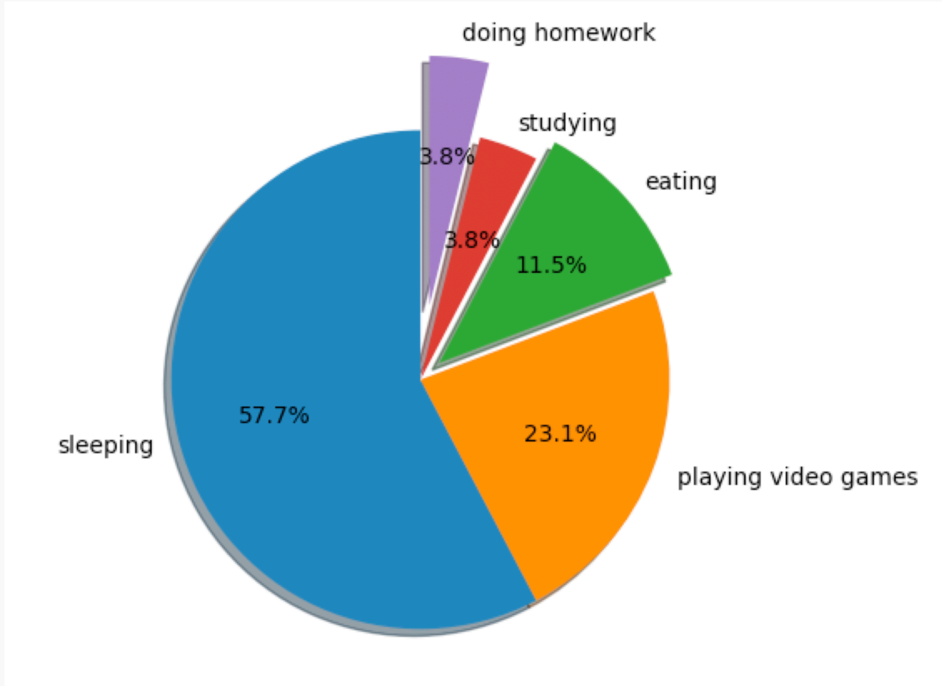
```
N = 50
x = np.random.rand(N)
y = np.random.rand(N)
colors = np.random.rand(N)
area = (30 * np.random.rand(N)) ** 2
plt.scatter(x, y, s=area, c=colors, alpha=0.5)
plt.show()
```



Pie charts

- A Pie Chart (or Pie Graph) is a special chart that uses "pie slices" to show relative sizes of data. The chart is divided into sectors, where each sector shows the relative size of each value.

```
slices = [15, 6, 3, 1, 1]
activities = ['sleeping', 'playing video games',
              'eating', 'studying',
              'doing homework']
explode = (0.0, 0.0, 0.1, 0.0, 0.3)
plt.pie(slices, labels=activities, explode=explode,
        autopct='%1.1f%%', shadow=True, startangle=90)
plt.show()
```


doing homework
studying
eating
sleeping
playing video games