Matplotlib Introduction.

Matplotlib is one of the most popular Python packages used for data visualization. It is a cross-platform library for making 2D plots from data in arrays. It provides an object-oriented API that helps in embedding plots in applications using Python GUI toolkits such as PyQt, WxPythonotTkinter. It can be used in Python and IPython shells, Jupyter notebook and web application servers also.

Matplotlib has a procedural interface named the Pylab, which is designed to resemble MATLAB, a proprietary programming language developed by MathWorks. Matplotlib along with NumPy can be considered as the open source equivalent of MATLAB.

Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002.

One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals. Matplotlib consists of several plots like line, bar, scatter, histogram etc.

Matplotlib - Pyplot API.

Types Of Plot.

1.bar plot.

2.barh plot.

3.box plot.

4.hist.

5.hist2d.

6.pie.

7.plot(line plot).

8.polar.

9.scatterplot.

10.stackplot.

11.stem.

12.step.

13.Quiver.

Image Functions.

1.Imread-Read an image from a file into an array.

2.Imsave-Save an array as in image file.

3.Imshow-Display an image on the axes.

Axis Functions.

1.Axes - Add axes to the figure.

2.Text - Add text to the axes.

3.Title - Set a title of the current axes.

4.Xlabel-Set the x axis label of the current axis.

5.Xlim- Get or set the x limits of the current axes.

6.Xscale-

7.Xticks- Get or set the x-limits of the current tick locations and labels.

8.Ylabel-Set the y axis label of the current axis.

9.Ylim- Get or set the y-limits of the current axes.

10.Yscale- Set the scaling of the y-axis.

11.Yticks- Get or set the y-limits of the current tick locations and labels.

Figure Functions.

1.Figtext -Add text to figure.

2.Figure - Creates a new figure.

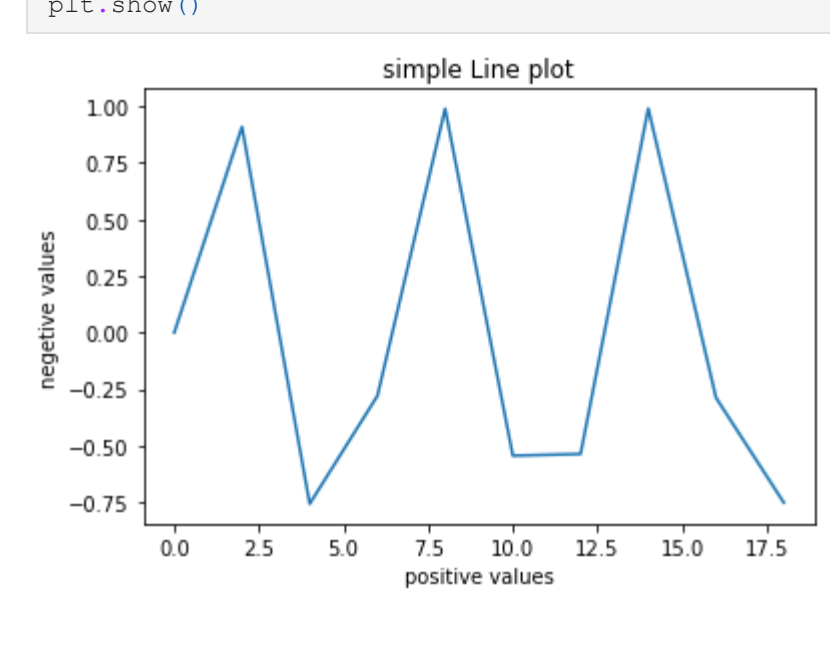
3.Show - Display a figure.

4.Savefig - Save the current figure.

5.Close - Close a figure window.

Matplotlib - Simple Plot.

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

x = np.arange(0,20,2)

y = np.sin(x)

plt.plot(x,y)

plt.xlabel('positive values')
plt.ylabel('negetive values')
plt.title('simple Line plot')
plt.show()
```



Matplotlib - Figure Class.

The matplotlib.figure module contains the Figure class. It is a top-level container for all plot elements. The Figure object is instantiated by calling the figure() function from the pyplot module.

fig = plt.figure()

The following table shows the additional parameters −

1.Figsize - (width,height) tuple in inches.

2.Dpi- Dots per inches.

3.Facecolor-Figure patch facecolor.

4.Edgecolor-Figure patch edge color.

5.Linewidth-Edge line width.

Matplotlib - Axes Class.

Axes object is the region of the image with the data space. A given figure can contain many Axes, but a given Axes object can only be in one Figure. The Axes contains two (or three in the case of 3D) Axis objects. The Axes class and its member functions are the primary entry point to working with the OO interface. m

Axes object is added to figure by calling the add_axes() method. It returns the axes object and adds an axes at position rect [left, bottom, width, height] where all quantities are in fractions of figure width and height.

Parameter :-

Following is the parameter for the Axes class −

rect − A 4-length sequence of [left, bottom, width, height] quantities.

ax=fig.add_axes([0,0,1,1])

The following member functions of axes class add different elements to plot −

ax.legend(handles, labels, loc)

Where labels is a sequence of strings and handles a sequence of Line2D or Patch instances. loc can be a string or an integer specifying the legend location.

Best = 0

upper right=1

upper left=2

lower left=3

lower right=4

Right =5

axes.plot()

This is the basic method of axes class that plots values of one array versus another as lines or markers. The plot() method can have an optional format string argument to specify color, style and size of line and marker.

Color Code:-

'b' Blue

'g' Green

'r' Red

'b' Blue

'c' Cyan

'm' Magenta

'y' Yellow

'k' Black

'b' Blue

'w' White

Marker codes:-

'.' Point marker

'o' Circle marker

'x' X marker

'D' Diamond marker

'H' Hexagon marker

's' Square marker

'+' Plus marker

Line Style :-

'-' Solid line

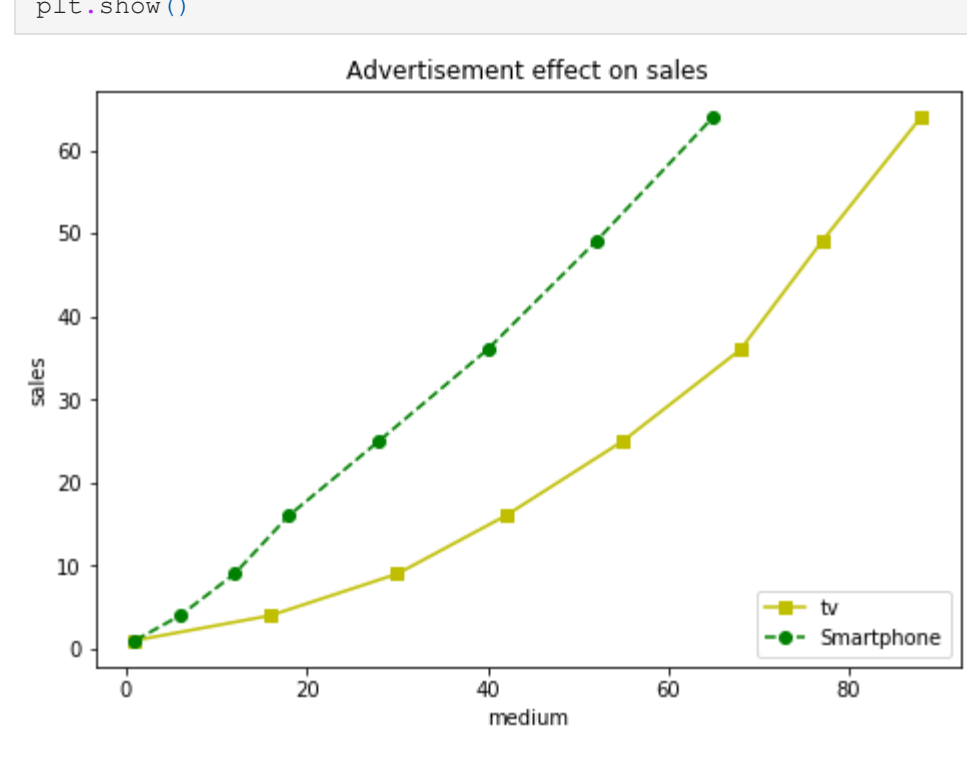
'—' Dashed line

'-.' Dash-dot line

':' Dotted line

'H' Hexagon marker

```python
import matplotlib.pyplot as plt
y = [1, 4, 9, 16, 25,36,49, 64]
x1 = [1, 16, 30, 42,55, 68, 77,88]
x2 = [1,6,12,18,28, 40, 52, 65]
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
l1 = ax.plot(x1,y,'ys-') # solid line with yellow colour and square marker
l2 = ax.plot(x2,y,'go--') # dash line with green colour and circle marker
ax.legend(labels = ('tv', 'Smartphone'), loc = 'lower right') # legend placed at lower right
ax.set_title("Advertisement effect on sales")
ax.set_xlabel('medium')
ax.set_ylabel('sales')
plt.show()
```



Matplotlib - Multiplots.

In this chapter, we will learn how to create multiple subplots on same canvas.

The subplot() function returns the axes object at a given grid position. The Call signature of this function is −

plt.subplot(subplot(nrows, ncols, index)

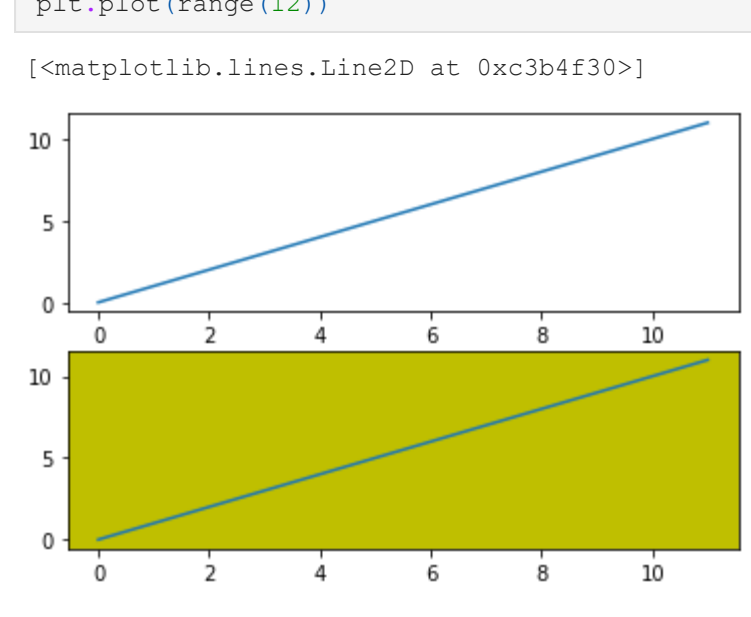
In the current figure, the function creates and returns an Axes object, at position index of a grid of nrows by ncolsaxes. Indexes go from 1 to nrows * ncols, incrementing in row-major order.If nrows, ncols and index are all less than 10. The indexes can also be given as single, concatenated, threedigitnumber.

For example, subplot(2, 3, 3) and subplot(233) both create an Axes at the top right corner of the current figure, occupying half of the figure height and a third of the figure width.

Creating a subplot will delete any pre-existing subplot that overlaps with it beyond sharing a boundary.

```python
import matplotlib.pyplot as plt
%matplotlib inline
# plot a line, implicitly creating a subplot(111)
plt.plot([1,2,3])
# now create a subplot which represents the top plot of a grid with 2 rows and 1 column.
#Since this subplot will overlap the first, the plot (and its axes) previously
#created, will be removed
plt.subplot(211)
plt.plot(range(12))
plt.subplot(212, facecolor='y') # creates 2nd subplot with yellow background
plt.plot(range(12))
```

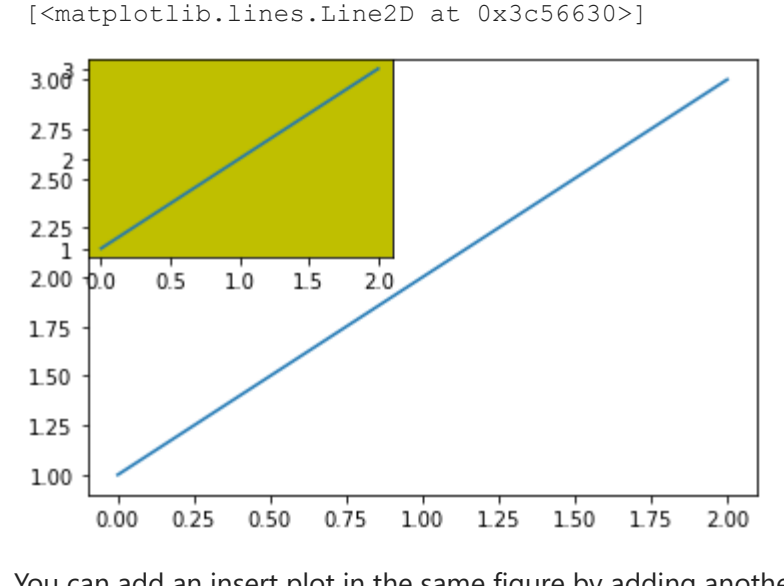
```
[<matplotlib.lines.Line2D at 0xc3b4f30>]
```



The add_subplot() function of the figure class will not overwrite the existing plot −

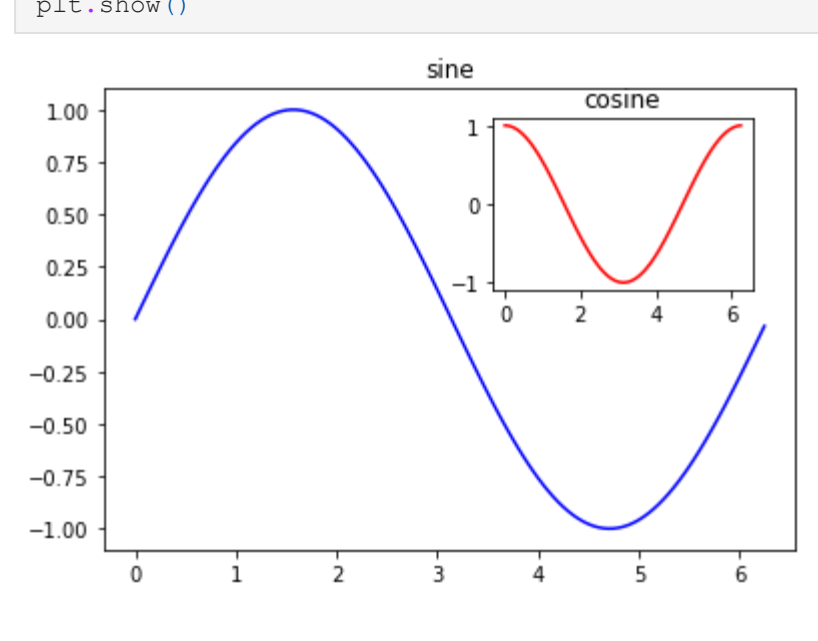
```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax1 = fig.add_subplot(111)
ax1.plot([1,2,3])
ax2 = fig.add_subplot(221, facecolor='y')
ax2.plot([1,2,3])
```

```
[<matplotlib.lines.Line2D at 0x3c56630>]
```



You can add an insert plot in the same figure by adding another axes object in the same figure canvas.

```python
import matplotlib.pyplot as plt
import numpy as np
import math
x = np.arange(0, math.pi*2, 0.05)
fig=plt.figure()
axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
axes2 = fig.add_axes([0.55, 0.55, 0.3, 0.3]) # inset axes
y = np.sin(x)
axes1.plot(x, y, 'b')
axes2.plot(x,np.cos(x),'r')
axes1.set_title('sine')
axes2.set_title("cosine")
plt.show()
```



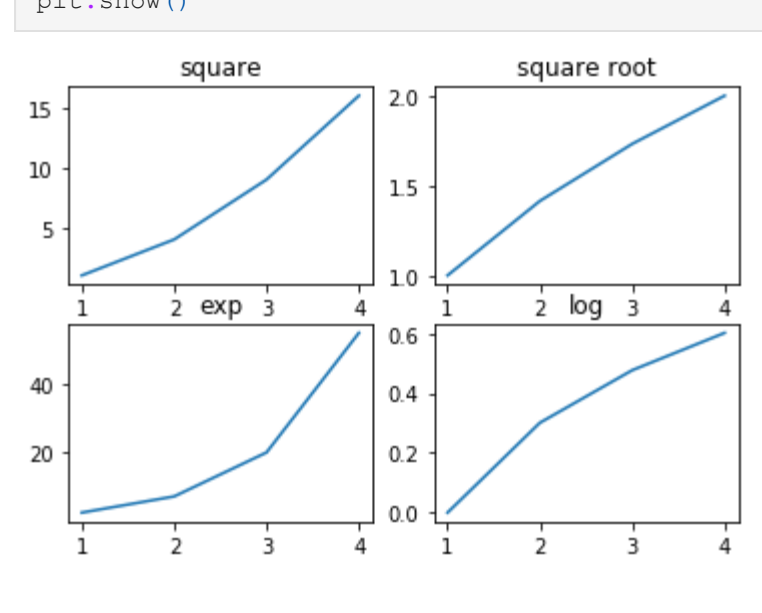
Matplotlib - Subplots() Function.

Matplotlib'spyplot API has a convenience function called subplots() which acts as a utility wrapper and helps in creating common layouts of subplots, including the enclosing figure object, in a single call.

Plt.subplots(nrows, ncols)

The two integer arguments to this function specify the number of rows and columns of the subplot grid. The function returns a figure object and a tuple containing axes objects equal to nrows*ncols. Each axes object is accessible by its index. Here we create a subplot of 2 rows by 2 columns and display 4 different plots in each subplot.

```python
import matplotlib.pyplot as plt
fig,ax =  plt.subplots(2,2)
import numpy as np
x = np.arange(1,5)
ax[0][0].plot(x,x*x)
ax[0][0].set_title('square')
ax[0][1].plot(x,np.sqrt(x))
ax[0][1].set_title('square root')
ax[1][0].plot(x,np.exp(x))
ax[1][0].set_title('exp')
ax[1][1].plot(x,np.log10(x))
ax[1][1].set_title('log')
plt.show()
```



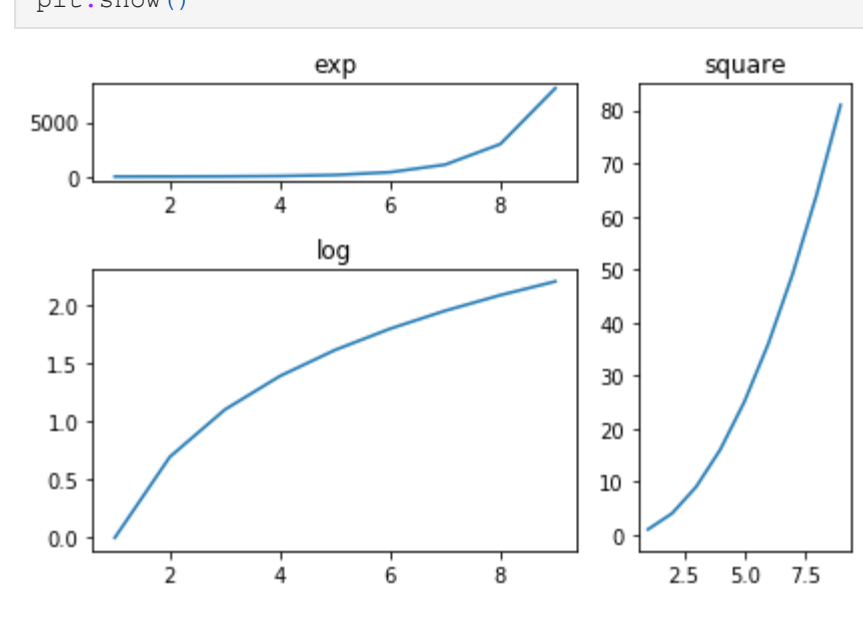
Matplotlib - Subplot2grid() Function

This function gives more flexibility in creating an axes object at a specific location of the grid. It also allows the axes object to be spanned across multiple rows or columns.

Plt.subplot2grid(shape, location, rowspan, colspan)

In the following example, a 3X3 grid of the figure object is filled with axes objects of varying sizes in row and column spans, each showing a different plot.

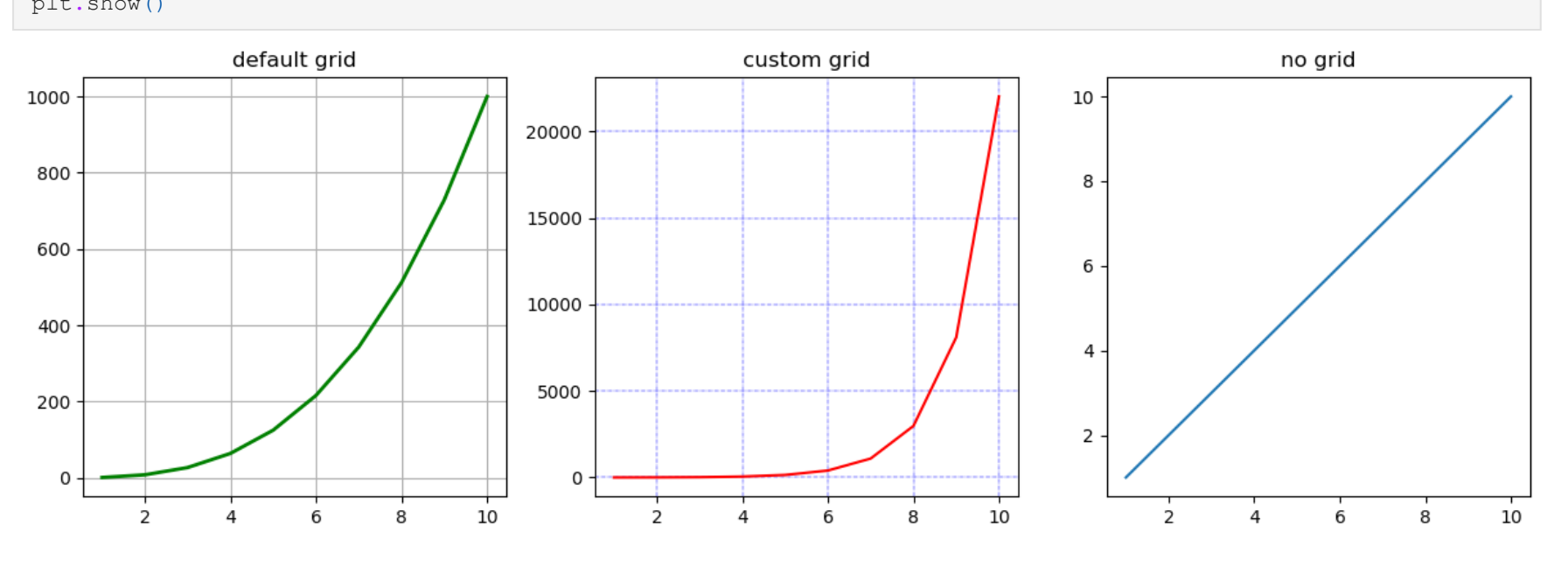
```python
import matplotlib.pyplot as plt
a1 = plt.subplot2grid((3,3),(0,0),colspan = 2)
a2 = plt.subplot2grid((3,3),(0,2), rowspan = 3)
a3 = plt.subplot2grid((3,3),(1,0),rowspan = 2, colspan = 2)
import numpy as np
x = np.arange(1,10)
a2.plot(x, x*x)
a2.set_title('square')
a1.plot(x, np.exp(x))
a1.set_title('exp')
a3.plot(x, np.log(x))
a3.set_title('log')
plt.tight_layout()
plt.show()
```



Matplotlib - Grids.

The grid() function of axes object sets visibility of grid inside the figure to on or off. You can also display major / minor (or both) ticks of the grid. Additionally color, linestyle and linewidth properties can be set in the grid() function.

```python
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots(1,3, figsize = (12,4),dpi=100)
x = np.arange(1,11)
ax[0].plot(x, x**3, 'g',lw=2)
ax[0].grid(True)
ax[0].set_title('default grid')
ax[1].plot(x, np.exp(x), 'r')
ax[1].grid(color='b', ls = '-.', lw = 0.35)
ax[1].set_title('custom grid')
ax[2].plot(x,x)
ax[2].set_title('no grid')
fig.tight_layout()
plt.show()
```



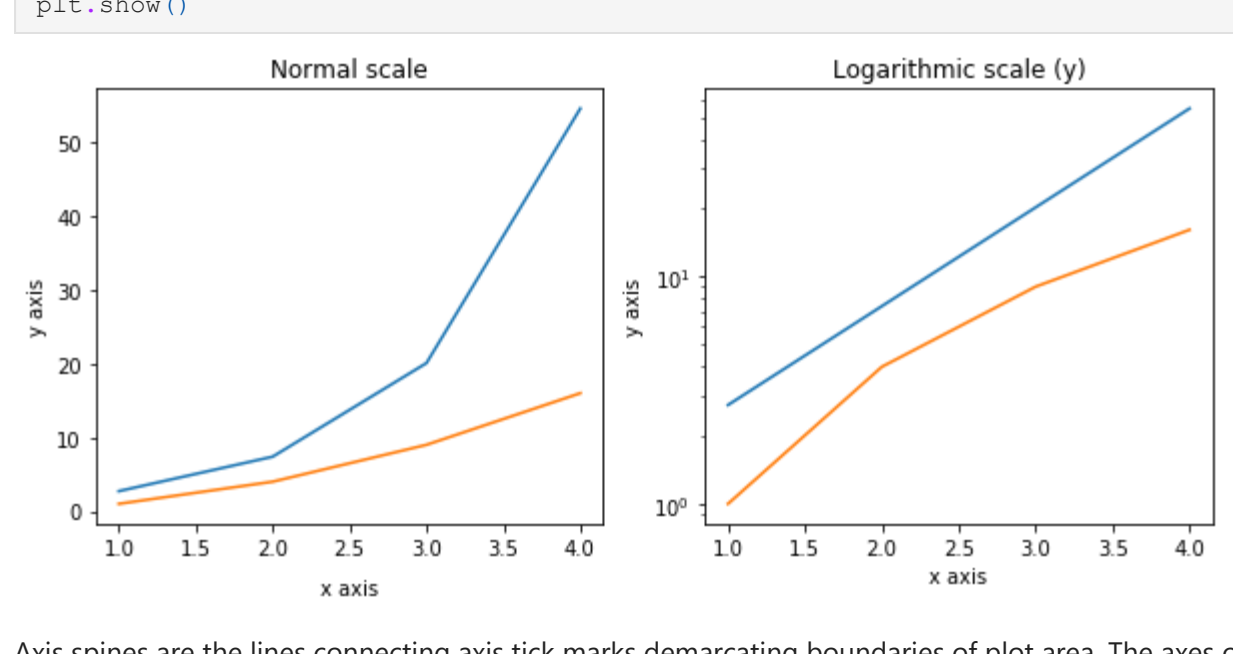
Matplotlib - Formatting Axes.

Sometimes, one or a few points are much larger than the bulk of data. In such a case, the scale of an axis needs to be set as logarithmic rather than the normal scale. This is the Logarithmic scale. In Matplotlib, it is possible by setting xscale or vscale property of axes object to 'log'.

It is also required sometimes to show some additional distance between axis numbers and axis label. The labelpad property of either axis (x or y or both) can be set to the desired value.

Both the above features are demonstrated with the help of the following example. The subplot on the right has a logarithmic scale and one on left has its x axis having label at more distance.

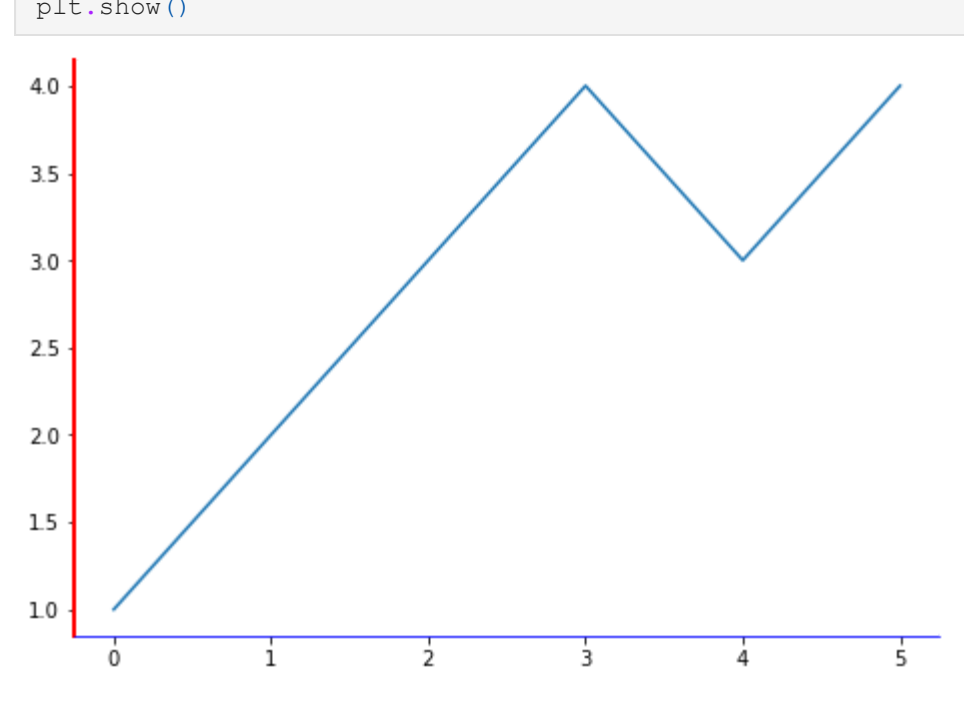
```python
import matplotlib.pyplot as plt
import numpy as np
fig, ax = plt.subplots(1, 2, figsize=(10,4))
x = np.arange(1,5)
ax[0].plot( x, np.exp(x))
ax[0].plot(x,x**2)
ax[0].set_title("Normal scale")
ax[1].plot (x, np.exp(x))
ax[1].plot(x, x**2)
ax[1].set_yscale("log")
ax[1].set_title("Logarithmic scale (y)")
ax[0].set_xlabel("x axis")
ax[0].set_ylabel("y axis")
ax[0].xaxis.labelpad = 10
ax[1].set_xlabel("x axis")
ax[1].set_ylabel("y axis")
plt.show()
```



Axis spines are the lines connecting axis tick marks demarcating boundaries of plot area. The axes object has spines located at top, bottom, left and right.

Each spine can be formatted by specifying color and width. Any edge can be made invisible if its color is set to none.

```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.spines['bottom'].set_color('blue')
ax.spines['left'].set_color('red')
ax.spines['left'].set_linewidth(2)
ax.spines['right'].set_color(None)
ax.spines['top'].set_color(None)
ax.plot([1,2,3,4,3,4])
plt.show()
```

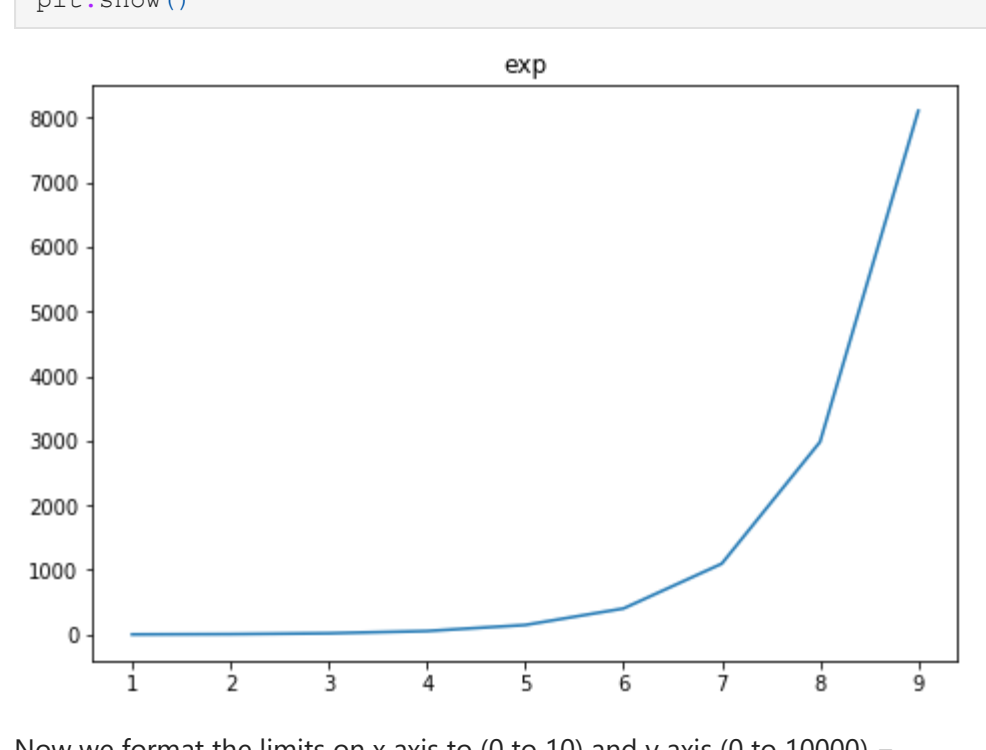


Matplotlib - Setting Limits.

Matplotlib automatically arrives at the minimum and maximum values of variables to be displayed along x, y (and z axis in case of 3D plot) axes of a plot. However, it is possible to set the limits explicitly by using set_xlim() and set_ylim() functions.

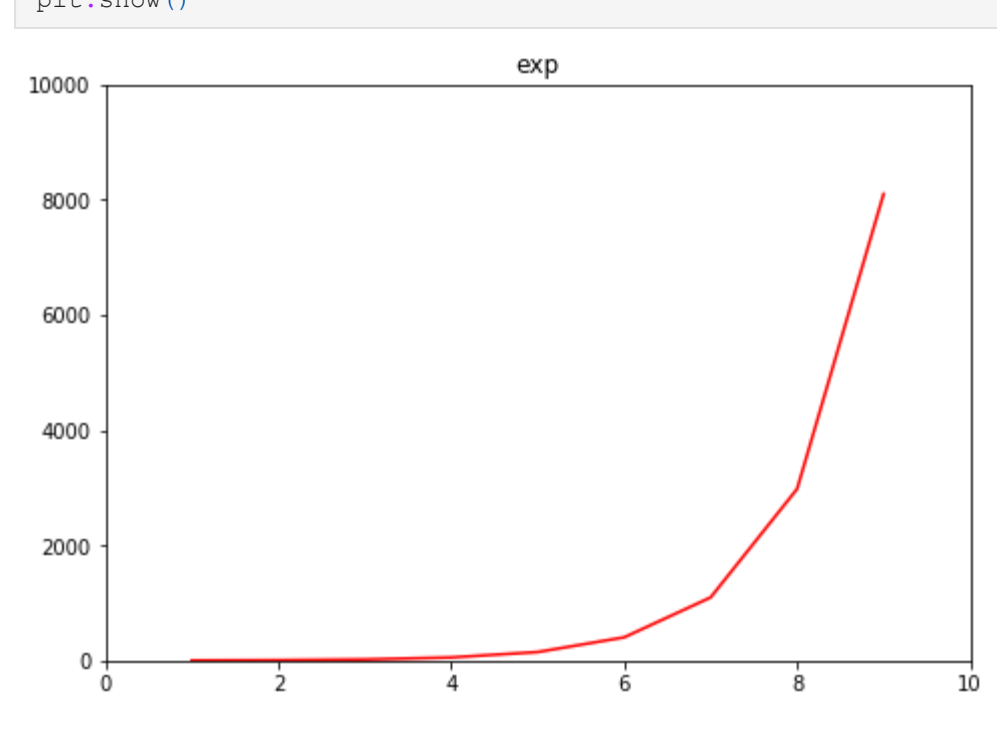
In the following plot, the autoscaled limits of x and y axes are shown −

```python
import matplotlib.pyplot as plt
fig = plt.figure()
a1 = fig.add_axes([0,0,1,1]) # l, b, w, h
import numpy as np
x = np.arange(1,10)
a1.plot(x, np.exp(x))
a1.set_title('exp')
plt.show()
```



Now we format the limits on x axis to (0 to 10) and y axis (0 to 10000) −

```python
import matplotlib.pyplot as plt
fig = plt.figure()
a1 = fig.add_axes([0,0,1,1])
import numpy as np
x = np.arange(1,10)
a1.plot(x, np.exp(x),'r')
a1.set_title('exp')
a1.set_ylim(0,10000)
a1.set_xlim(0,10)
plt.show()
```



Matplotlib - Setting Ticks and Tick Labels.

Ticks are the markers denoting data points on axes. Matplotlib has so far - in all our previous examples - automatically taken over the task of spacing points on the axis.Matplotlib's default tick locators and formatters are designed to be generally sufficient in many common situations. Position and labels of ticks can be explicitly mentioned to suit specific requirements.

The xticks() and yticks() function takes a list object as argument. The elements in the list denote the positions on corresponding action where ticks will be displayed.

ax.set_xticks([2,4,6,8,10])

This method will mark the data points at the given positions with ticks.

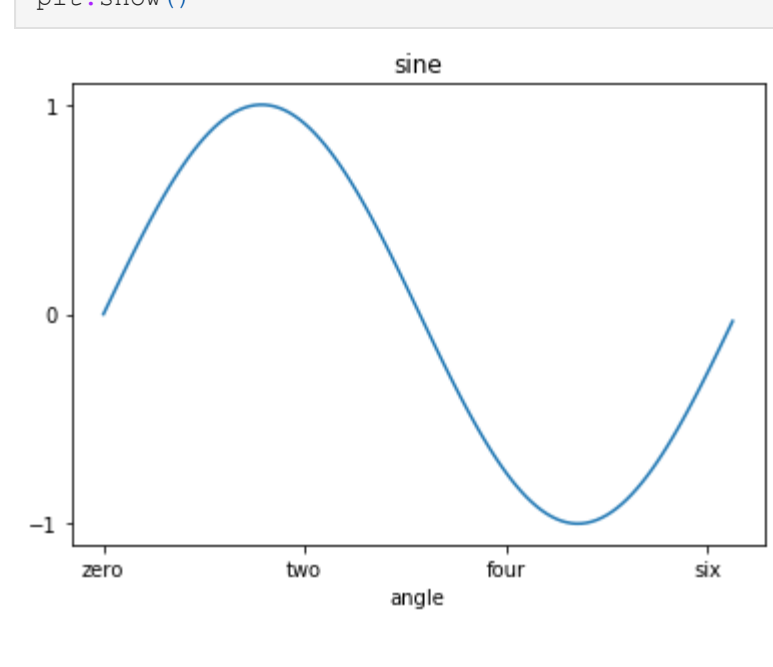
Similarly, labels corresponding to tick marks can be set by set_xlabels() and set_ylabels() functions respectively.

ax.set_xlabels(['two', 'four','six', 'eight', 'ten'])

This will display the text labels below the markers on the x axis.

Following example demonstrates the use of ticks and labels.

```python
import matplotlib.pyplot as plt
import numpy as np
import math
x = np.arange(0, math.pi*2, 0.05)
fig = plt.figure()
ax = fig.add_axes([0.1, 0.1, 0.8, 0.8]) # main axes
y = np.sin(x)
ax.plot(x, y)
ax.set_xlabel('angle')
ax.set_title('sine')
ax.set_xticks([0,2,4,6])
ax.set_xticklabels(['zero','two','four','six'])
ax.set_yticks([-1,0,1])
plt.show()
```

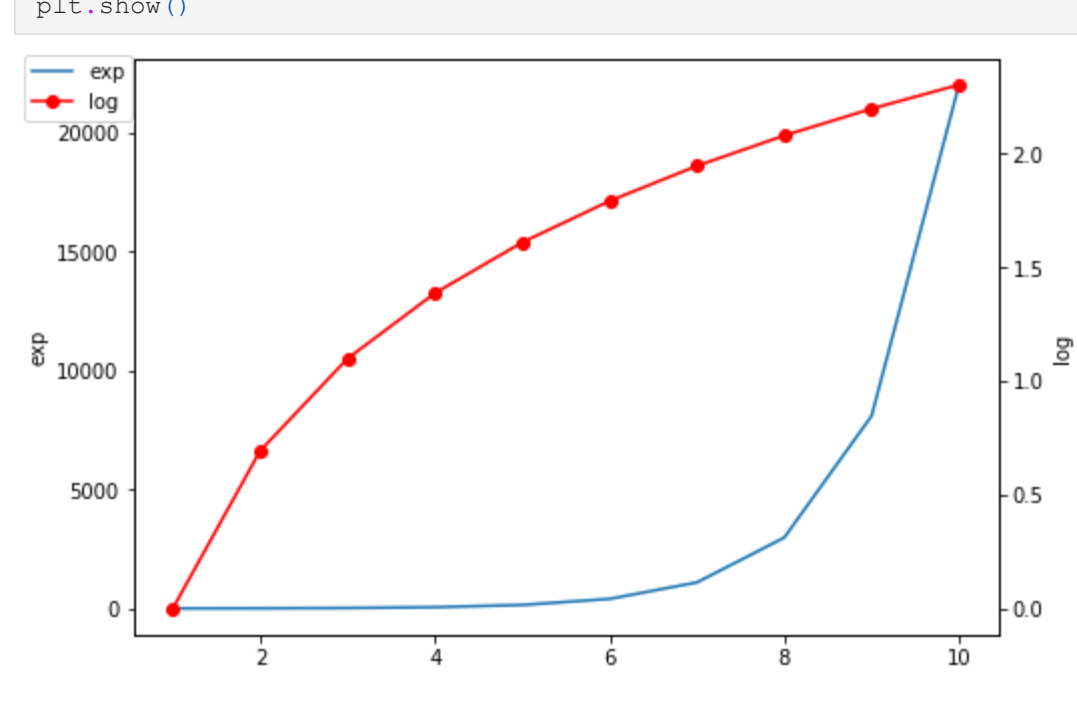


Matplotlib - Twin Axes.

It is considered useful to have dual x or y axes in a figure. Moreso, when plotting curves with different units together. Matplotlib supports this with the twinxand twiny functions.

In the following example, the plot has dual y axes, one showing exp(x) and the other showing log(x) −

```python
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
a1 = fig.add_axes([0,0,1,1])
x = np.arange(1,11)
a1.plot(x,np.exp(x))
a1.set_ylabel('exp')
a2 = a1.twinx()
a2.plot(x, np.log(x),'ro-')
a2.set_ylabel('log')
fig.legend(labels = ('exp','log'),loc='upper left')
plt.show()
```



Matplotlib - Bar Plot.

A bar chart or bar graph is a chart or graph that presents categorical data with rectangular bars with heights or lengths proportional to the values that they represent. The bars can be plotted vertically or horizontally.

A bar graph shows comparisons among discrete categories. One axis of the chart shows the specific categories being compared, and the other axis represents a measured value.

Matplotlib API provides the bar() function that can be used in the MATLAB style use as well as object oriented API. The signature of bar() function to be used with axes object is as follows −

ax.bar(x, height, width, bottom, align)

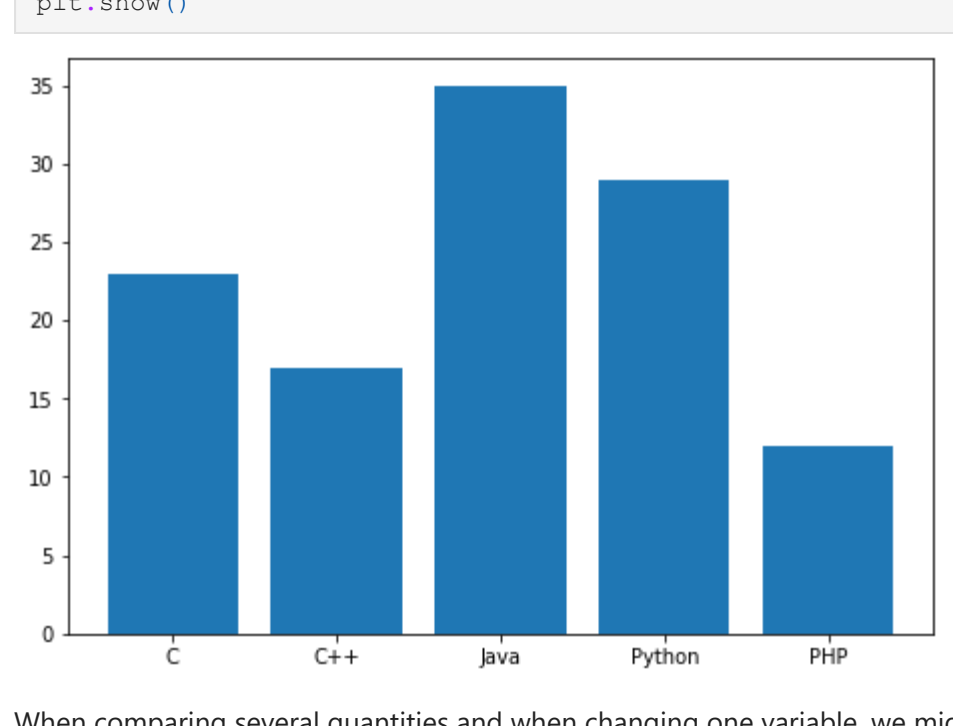
The function makes a bar plot with the bound rectangle of size (x −width = 2; x + width=2; bottom; bottom + height).

```
x-sequence of scalars representing the x coordinates of the bars. align controls if x is the bar center
(default) or left edge.

height-scalar or sequence of scalars representing the height(s) of the bars.

width-scalar or array-like, optional. the width(s) of the bars default 0.8.

bottom-scalar or array-like, optional. the y coordinate(s) of the bars default None.

align-{'center', 'edge'}, optional, default 'center'.
```

The function returns a Matplotlib container object with all bars.

Following is a simple example of the Matplotlib bar plot. It shows the number of students enrolled for various courses offered at an institute.

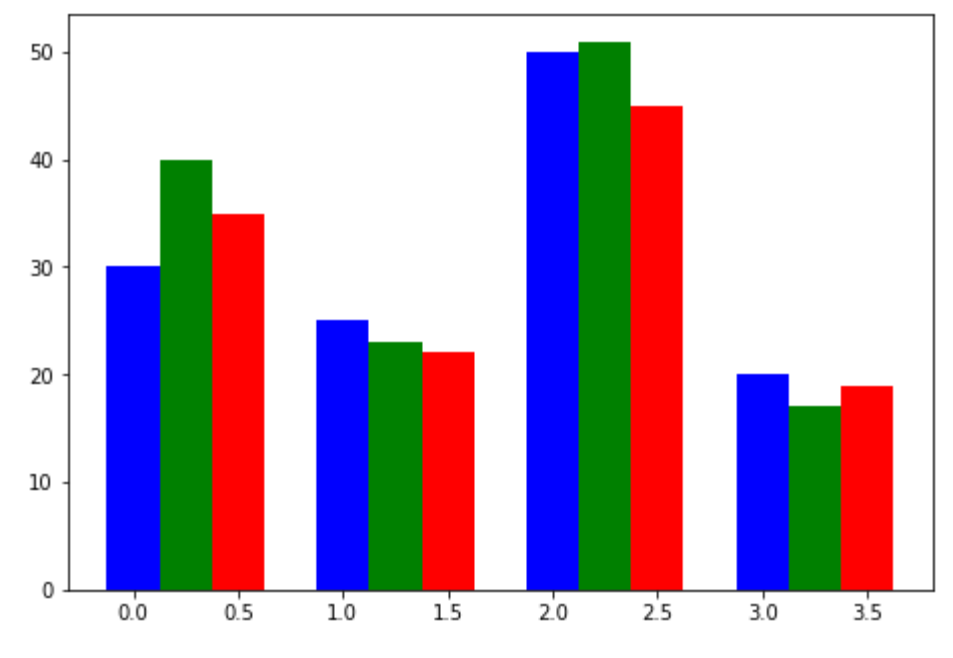
```python
import matplotlib.pyplot as plt
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
ax.bar(langs,students)
plt.show()
```



When comparing several quantities and when changing one variable, we might want a bar chart where we have bars of one color for one quantity value.

We can plot multiple bar charts by playing with the thickness and the positions of the bars. The data variable contains three series of four values. The following script will show three bar charts of four bars. The bars will have a thickness of 0.25 units. Each bar chart will be shifted 0.25 units from the previous one. The data object is a multidict containing number of students passed in three branches of an engineering college over the last four years.

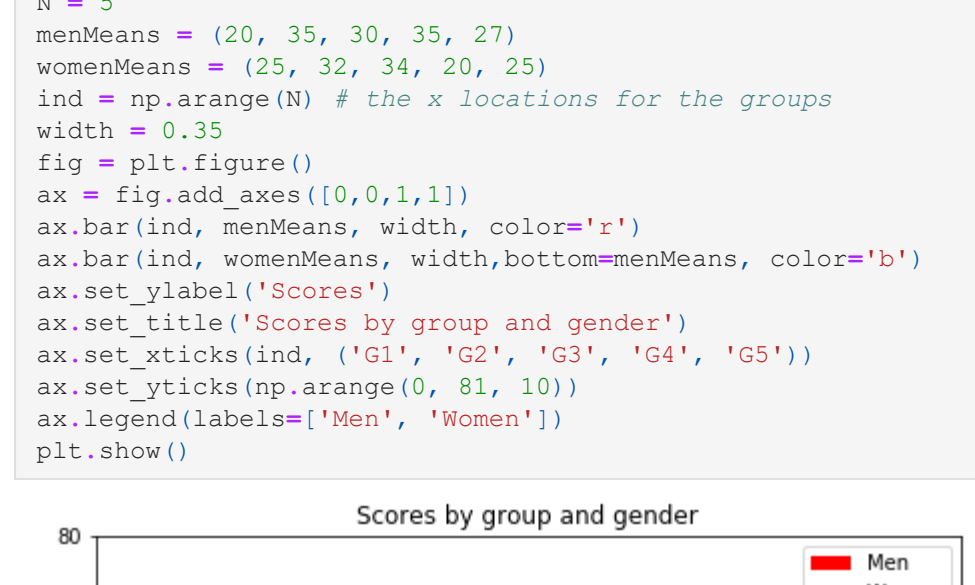
```python
import numpy as np
import matplotlib.pyplot as plt
data = [[30, 25, 50, 20],
[40, 23, 51, 17],
[35, 22, 45, 19]]
X = np.arange(4)
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(X + 0.00, data[0], color = 'b', width = 0.25)
ax.bar(X + 0.25, data[1], color = 'g', width = 0.25)
ax.bar(X + 0.50, data[2], color = 'r', width = 0.25)
plt.show()
```



The stacked bar chart stacks bars that represent different groups on top of each other. The height of the resulting bar shows the combined result of the groups.

The optional bottom parameter of the pyplot.bar() function allows you to specify a starting value for a bar. Instead of running from zero to a value, it will go from the bottom to the value. The first call to pyplot.bar() plots the blue bars. The second call to pyplot.bar() plots the red bars, with the bottom of the blue bars being at the top of the red bars.

```
In [42]: import numpy as np
import matplotlib.pyplot as plt
N = 5
menMeans = (20, 35, 30, 35, 27)
womenMeans = (25, 32, 34, 20, 25)
ind = np.arange(N) # the x locations for the groups
width = 0.35
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.bar(ind, menMeans, width, color='r')
ax.bar(ind, womenMeans, width,bottom=menMeans, color='b')
ax.set_ylabel('Scores')
ax.set_title('Scores by group and gender')
ax.set_xticks(ind, ('G1', 'G2', 'G3', 'G4', 'G5'))
ax.set_yticks(np.arange(0, 81, 10))
ax.legend(labels=['Men', 'Women'])
plt.show()
```



Matplotlib - Histogram.

A histogram is an accurate representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable. It is a kind of bar graph.

To construct a histogram, follow these steps –

Bin the range of values. Divide the entire range of values into a series of intervals. Count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable.

The matplotlib.pyplot.hist() function plots a histogram. It computes and draws the histogram of x.

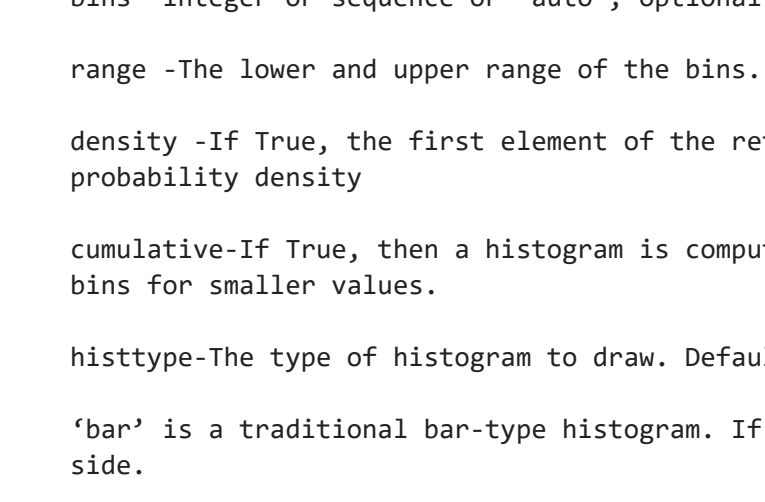
Parameters:-

The following table lists down the parameters for a histogram –

```
x-array or sequence of arrays

bins -integer or sequence or 'auto', optional

range -The lower and upper range of the bins.

density -If True, the first element of the return tuple will be the counts normalized to form a
probability density

cumulative-If True, then a histogram is computed where each bin gives the counts in that bin plus all
bins for smaller values.

histtype-The type of histogram to draw. Default is 'bar'

'bar' is a traditional bar-type histogram. If multiple data are given the bars are arranged side by
side.

'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.

'step' generates a lineplot that is by default unfilled.

'stepfilled' generates a lineplot that is by default filled.
```

Following example plots a histogram of marks obtained by students in a class. Four bins, 0-25, 26-50, 51-75, and 76-100 are defined. The Histogram shows number of students falling in this range

```
In [44]: from matplotlib import pyplot as plt
import numpy as np
fig,ax = plt.subplots(1,1)
a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27])
ax.hist(a, bins = [0,25,50,75,100])
ax.set_title("histogram of result")
ax.set_xticks([0,25,50,75,100])
ax.set_xlabel('marks')
ax.set_ylabel('no. of students')
plt.show()
```



Matplotlib - Pie Chart.

A Pie Chart can only display one series of data. Pie charts show the size of items (called wedge) in one data series, proportional to the sum of the items. The data points in a pie chart are shown as a percentage of the whole pie.

Matplotlib API has a pie() function that generates a pie diagram representing data in an array. The fractional area of each wedge is given by x/sum(x). If sum(x)< 1, then the values of x give the fractional area directly and the array will not be normalized. Theresulting pie will have an empty wedge of size 1 - sum(x).

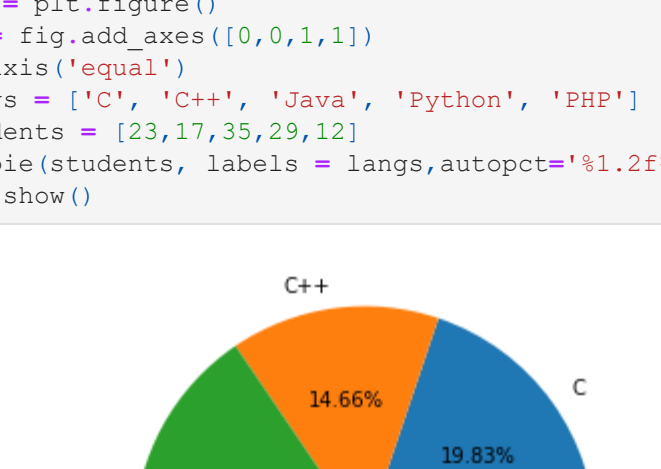
The pie chart looks best if the figure and axes are square, or the Axes aspect is equal.

Parameters:-

```
x-array-like. The wedge sizes.

labels-list. A sequence of strings providing the labels for each wedge.

Colors-A sequence of matplotlibcolorargs through which the pie chart will cycle. If None, will use the
colors in the currently active cycle.

Autopct-string, used to label the wedges with their numeric value. The label will be placed inside the
wedge. The format string will be fmt%pct.
```

Following code uses the pie() function to display the pie chart of the list of students enrolled for various computer language courses. The proportionate percentage is displayed inside the respective wedge with the help of autopct parameter which is set to %1.2f%.

```
In [45]: from matplotlib import pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
ax.axis('equal')
langs = ['C', 'C++', 'Java', 'Python', 'PHP']
students = [23,17,35,29,12]
ax.pie(students, labels = langs,autopct='%1.2f%%')
plt.show()
```

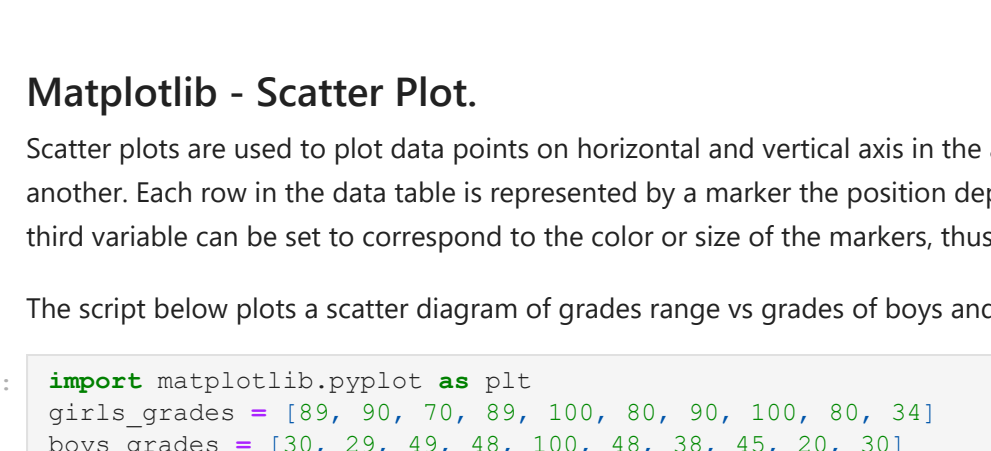


Matplotlib - Scatter Plot.

Scatter plots are used to plot data points on horizontal and vertical axis in the attempt to show how much one variable is affected by another. Each row in the data table is represented by a marker the position depends on its values in the columns set on the X and Y axes. A third variable can be set to correspond to the color or size of the markers, thus adding yet another dimension to the plot.

The script below plots a scatter diagram of grades range vs grades of boys and girls in two different colors.

```
In [46]: import matplotlib.pyplot as plt
girls_grades = [89, 90, 70, 89, 100, 80, 90, 100, 80, 34]
boys_grades = [30, 29, 49, 48, 100, 48, 38, 45, 20, 30]
grades_range = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
fig=plt.figure()
ax=fig.add_axes([0,0,1,1])
ax.scatter(grades_range, girls_grades, color='r')
ax.scatter(grades_range, boys_grades, color='b')
ax.set_xlabel('Grades Range')
ax.set_ylabel('Grades Scored')
ax.set_title('scatter plot')
plt.show()
```



Matplotlib - Contour Plot.

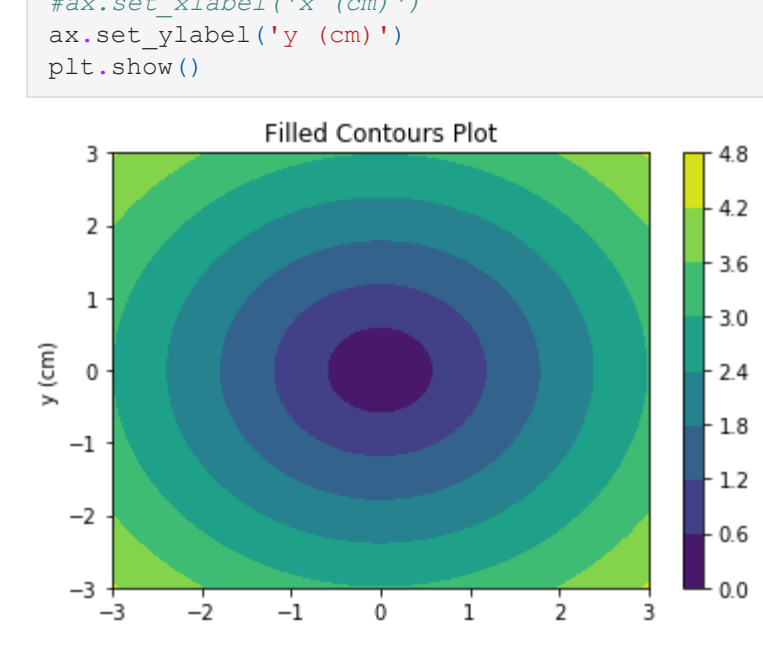
Contour plots (sometimes called Level Plots) are a way to show a three-dimensional surface on a two-dimensional plane. It graphs two predictor variables X Y on the y-axis and a response variable Z as contours. These contours are sometimes called the z-slices or the iso-response values.

A contour plot is appropriate if you want to see how alue Z changes as a function of two inputs X and Y, such that Z = f(X,Y). A contour line or isoline of a function of two variables is a curve along which the function has a constant value.

The independent variables x and y are usually restricted to a regular grid called meshgrid. The numpy.meshgrid creates a rectangular grid out of an array of x values and an array of y values.

Matplotlib API contains contour() and contourf() functions that draw contour lines and filled contours, respectively. Both functions need three parameters x,y and z.

```
In [49]: import numpy as np
import matplotlib.pyplot as plt
xlist = np.linspace(-3.0, 3.0, 100)
ylist = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(xlist, ylist)
Z = np.sqrt(X**2 + Y**2)
fig,ax=plt.subplots(1,1)
cp = ax.contourf(X, Y, Z)
fig.colorbar(cp) # Add a colorbar to a plot
ax.set_title('Filled Contours Plot')
#ax.set_xlabel('x (cm)')
ax.set_ylabel('y (cm)')
plt.show()
```



Matplotlib - Quiver Plot.

A quiver plot displays the velocity vectors as arrows with components (u,v) at the points (x,y).

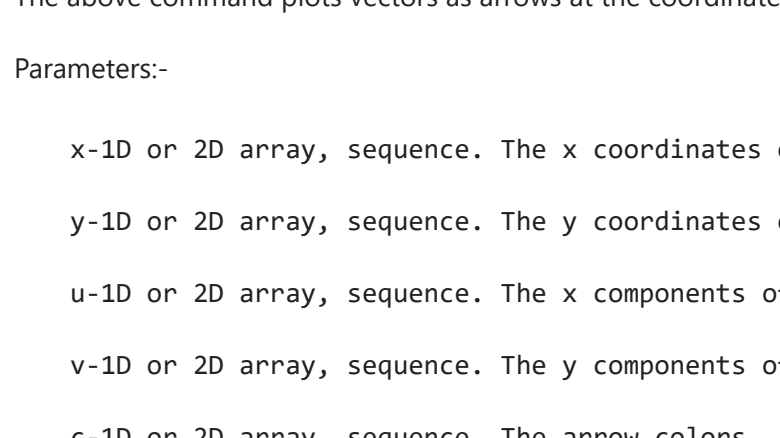
quiver(x,y,u,v)

The above command plots vectors as arrows at the coordinates specified in each corresponding pair of elements in x and y.

Parameters:-

```
x-1D or 2D array, sequence. The x coordinates of the arrow locations.

y-1D or 2D array, sequence. The y coordinates of the arrow locations.

u-1D or 2D array, sequence. The x components of the arrow vectors.

v-1D or 2D array, sequence. The y components of the arrow vectors.

c-1D or 2D array, sequence. The arrow colors.
```

```
In [50]: import matplotlib.pyplot as plt
import numpy as np
x,y = np.meshgrid(np.arange(-2, 2, .2), np.arange(-2, 2, .25))
z = x*np.exp(-x**2 - y**2)
v, u = np.gradient(z, .2, .2)
fig, ax = plt.subplots()
q = ax.quiver(x,y,u,v)
plt.show()
```



Matplotlib - Box Plot.

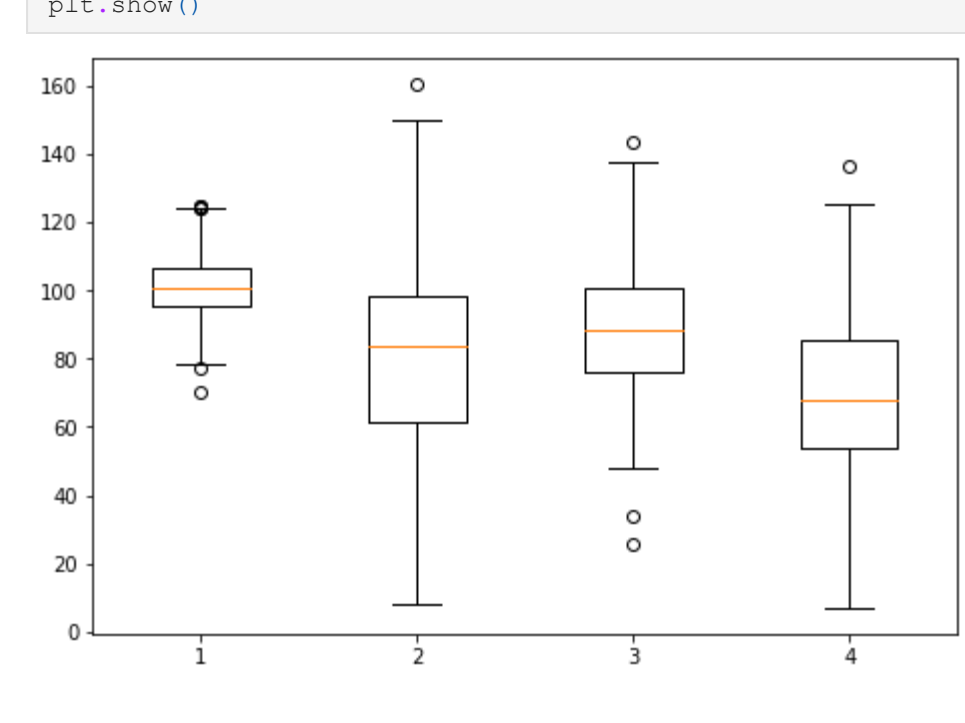
A box plot which is also known as a whisker plot displays a summary of a set of data containing the minimum, first quartile, median, third quartile, and maximum. In a box plot, we draw a box from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.

Let us create the data for the boxplots. We use the numpy.random.normal() function to create the fake data. It takes three arguments, mean and standard deviation of the normal distribution, and the number of values desired.

```
In [51]: np.random.seed(10)
collectn_1 = np.random.normal(100, 10, 200)
collectn_2 = np.random.normal(80, 30, 200)
collectn_3 = np.random.normal(90, 20, 200)
collectn_4 = np.random.normal(70, 25, 200)
```

The list of arrays that we created above is the only required input for creating the boxplot. Using the data_to_plot line of code, we can create the boxplot with the following code –

```
In [59]: collection= [collectn_1,collectn_2,collectn_3,collectn_4]
fig = plt.figure()
# Create an axes instance
ax = fig.add_axes([0,0,1,1])
# Create the boxplot
bp = ax.boxplot(collection)
plt.show()
```

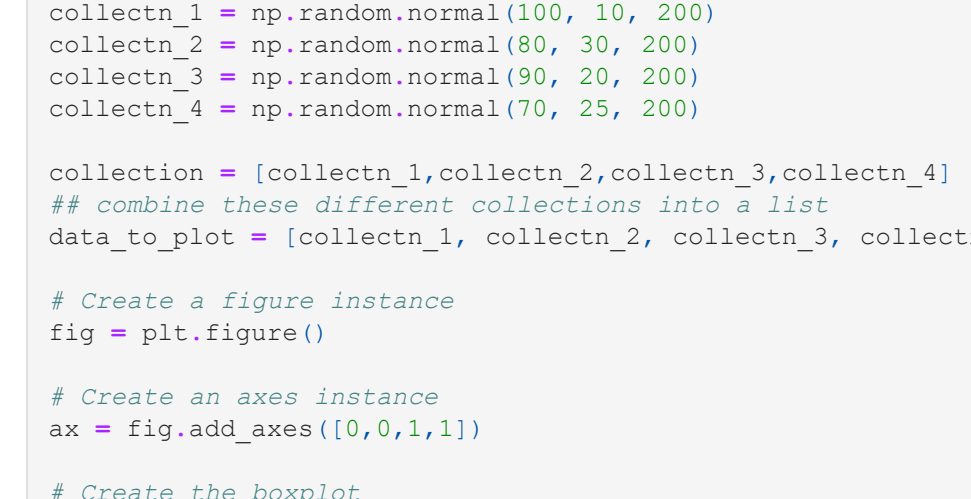


Matplotlib - Violin Plot.

Violin plots are similar to box plots, except that they also show the probability density of the data at different values. These plots include a marker for the median of the data and a box indicating the interquartile range, as in the standard box plots. Overlaid on this box plot is a kernel density estimation. Like box plots, violin plots are used to represent comparison of a variable distribution (or sample distribution) across different "categories".

A violin plot is more informative than a plain box plot. In fact while a box plot only shows summary statistics such as mean/median and interquartile ranges, the violin plot shows the full distribution of the data.

```
In [61]: import matplotlib.pyplot as plt

np.random.seed(10)
collectn_1 = np.random.normal(100, 10, 200)
collectn_2 = np.random.normal(80, 30, 200)
collectn_3 = np.random.normal(90, 20, 200)
collectn_4 = np.random.normal(70, 25, 200)

collection = [collectn_1,collectn_2,collectn_3,collectn_4]
## combine these different collections into a list
data_to_plot = [collectn_1, collectn_2, collectn_3, collectn_4]

# Create a figure instance
fig = plt.figure()

# Create an axes instance
ax = fig.add_axes([0,0,1,1])

# Create the boxplot
bp = ax.violinplot(collection)
plt.show()
```



Matplotlib By KGP.

Matplotlib is an amazing visualization library in Python for 2D plots of arrays. Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack. It was introduced by John Hunter in the year 2002.

One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals. Matplotlib consists of several plots like line, bar, scatter, histogram etc.

It usually import as matplotlib.pyplot as plt.

```
In [64]: import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
```

```
In [ ]: If we want to more about the API inclueded in the matplotlib that can find by using the help or dir.
```

```
In [65]: help(plt)
```

```
        change them, e.g., with the :meth:`set_ylim` command.  That is,
        the vertical extent is in axes coords: 0=bottom, 0.5=middle,
        1.0=top but the x location is in data coordinates.

        Parameters
        ----------
        xmin : scalar
            Number indicating the first X-axis coordinate of the vertical
            span rectangle in data units.
        xmax : scalar
            Number indicating the second X-axis coordinate of the vertical
            span rectangle in data units.
        ymin : scalar, optional
            Number indicating the first Y-axis coordinate of the vertical
            span rectangle in relative Y-axis units (0-1). Default to 0.
        ymax : scalar, optional
            Number indicating the second Y-axis coordinate of the vertical
            span rectangle in relative Y-axis units (0-1). Default to 1.

        Returns
        -------
        rectangle : matplotlib.patches.Polygon
            Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

        Other Parameters
        ----------------
        **kwargs
            Optional parameters are properties of the class
            matplotlib.patches.Polygon.

        See Also
        --------
        axhspan : Add a horizontal span across the axes.

        Examples
        --------
        Draw a vertical, green, translucent rectangle from x = 1.25 to
        x = 1.55 that spans the yrange of the axes.

        >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)

    bar(x, height, width=0.8, bottom=None, *, align='center', data=None, **kwargs)
        Make a bar plot.

        The bars are positioned at *x* with the given *align*\ment. Their
        dimensions are given by *width* and *height*. The vertical baseline
        is *bottom* (default 0).

        Each of *x*, *height*, *width*, and *bottom* may either be a scalar
        applying to all bars, or it may be a sequence of length N providing a
        separate value for each bar.

        Parameters
        ----------
        x : sequence of scalars
            The x coordinates of the bars. See also *align* for the
            alignment of the bars to the coordinates.

        height : scalar or sequence of scalars
            The height(s) of the bars.

        width : scalar or array-like, optional
            The width(s) of the bars (default: 0.8).

        bottom : scalar or array-like, optional
            The y coordinate(s) of the bars bases (default: 0).

        align : {'center', 'edge'}, optional, default: 'center'
            Alignment of the bars to the *x* coordinates:

            - 'center': Center the base on the *x* positions.
            - 'edge': Align the left edges of the bars with the *x* positions.

            To align the bars on the right edge pass a negative *width* and
            ``align='edge'``.

        Returns
        -------
        container : `.BarContainer`
            Container with all the bars and optionally errorbars.

        Other Parameters
        ----------------
        color : scalar or array-like, optional
            The colors of the bar faces.

        edgecolor : scalar or array-like, optional
            The colors of the bar edges.

        linewidth : scalar or array-like, optional
            Width of the bar edge(s). If 0, don't draw edges.

        tick_label : string or array-like, optional
            The tick labels of the bars.
            Default: None (Use default numeric labels.)

        xerr, yerr : scalar or array-like of shape(N,) or shape(2,N), optional
            If not *None*, add horizontal / vertical errorbars to the bar tips.
            The values are +/- sizes relative to the data:

            - scalar: symmetric +/- values for all bars
            - shape(N,): symmetric +/- values for each bar
            - shape(2,N): Separate - and + values for each bar. First row
                contains the lower errors, the second row contains the
                upper errors.
            - *None*: No errorbar. (Default)

            See :doc:`/gallery/statistics/errorbar_features`
            for an example on the usage of ``xerr`` and ``yerr``.

        ecolor : scalar or array-like, optional, default: 'black'
            The line color of the errorbars.

        capsize : scalar, optional
           The length of the error bar caps in points.
           Default: None, which will take the value from
           :rc:`errorbar.capsize`.

        error_kw : dict, optional
            Dictionary of kwargs to be passed to the `~.Axes.errorbar`
            method. Values of *ecolor* or *capsize* defined here take
            precedence over the independent kwargs.

        log : bool, optional, default: False
            If *True*, set the y-axis to be log scale.

        orientation : {'vertical',  'horizontal'}, optional
            *This is for internal use only.* Please use `barh` for
            horizontal bar plots. Default: 'vertical'.

        See also
        --------
        barh: Plot a horizontal bar plot.

        Notes
        -----
        The optional arguments *color*, *edgecolor*, *linewidth*,
        *xerr*, and *yerr* can be either scalars or sequences of
        length equal to the number of bars.  This enables you to use
        bar as the basis for stacked bar charts, or candlestick plots.
        Detail: *xerr* and *yerr* are passed directly to
        :meth:`errorbar`, so they can also have shape 2xN for
        independent specification of lower and upper errors.

        Other optional kwargs:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa: unknown
          capstyle: {'butt', 'round', 'projecting'}
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color: color
          contains: callable
          edgecolor or ec: color or None or 'auto'
          facecolor or fc: color or None
          figure: `.Figure`
          fill: bool
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float or None
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          visible: bool
          zorder: float

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    barbs(*args, data=None, **kw)
        Plot a 2D field of barbs.

        Call signature::

          barbs([X, Y], U, V, [C], **kw)

        Where *X*, *Y* define the barb locations, *U*, *V* define the barb
        directions, and *C* optionally sets the color.

        All arguments may be 1D or 2D. *U*, *V*, *C* may be masked arrays, but masked
        *X*, *Y* are not supported at present.

        Barbs are traditionally used in meteorology as a way to plot the speed
        and direction of wind observations, but can technically be used to
        plot any two dimensional vector quantity.  As opposed to arrows, which
        give vector magnitude by the length of the arrow, the barbs give more
        quantitative information about the vector magnitude by putting slanted
        lines or a triangle for various increments in magnitude, as show
        schematically below::

          :                   /\    \
          :                  /  \    \
          :                 /    \    \    \
          :                /      \    \    \
          :               ------------------------------


        The largest increment is given by a triangle (or "flag"). After those
        come full lines (barbs). The smallest increment is a half line.  There
        is only, of course, ever at most 1 half line.  If the magnitude is
        small and only needs a single half-line and no full lines or
        triangles, the half-line is offset from the end of the barb so that it
        can be easily distinguished from barbs with a single full line.  The
        magnitude for the barb shown above would nominally be 65, using the
        standard increments of 50, 10, and 5.

        See also https://en.wikipedia.org/wiki/Wind_barb.



        Parameters
        ----------
        X, Y : 1D or 2D array-like, optional
            The x and y coordinates of the barb locations. See *pivot* for how the
            barbs are drawn to the x, y positions.

            If not given, they will be generated as a uniform integer meshgrid based
            on the dimensions of *U* and *V*.

            If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D
            using ``X, Y = np.meshgrid(X, Y)``. In this case ``len(X)`` and ``len(Y)``
            must match the column and row dimensions of *U* and *V*.

        U, V : 1D or 2D array-like
            The x and y components of the barb shaft.

        C : 1D or 2D array-like, optional
            Numeric data that defines the barb colors by colormapping via *norm* and
            *cmap*.

            This does not support explicit colors. If you want to set colors directly,
            use *barbcolor* instead.

        length : float, default: 7
            Length of the barb in points; the other parts of the barb
            are scaled against this.

        pivot : {'tip', 'middle'} or float, default: 'tip'
            The part of the arrow that is anchored to the *X*, *Y* grid. The barb
            rotates about this point. This can also be a number, which shifts the
            start of the barb that many points away from grid point.

        barbcolor : color or color sequence
            Specifies the color of all parts of the barb except for the flags.  This
            parameter is analogous to the *edgecolor* parameter for polygons,
            which can be used instead. However this parameter will override
            facecolor.

        flagcolor : color or color sequence
            Specifies the color of any flags on the barb.  This parameter is
            analogous to the *facecolor* parameter for polygons, which can be
            used instead. However, this parameter will override facecolor.  If
            this is not set (and *C* has not either) then *flagcolor* will be
            set to match *barbcolor* so that the barb has a uniform color. If
            *C* has been set, *flagcolor* has no effect.

        sizes : dict, optional
            A dictionary of coefficients specifying the ratio of a given
            feature to the length of the barb. Only those values one wishes to
            override need to be included.  These features include:

            - 'spacing' - space between features (flags, full/half barbs)
            - 'height' - height (distance from shaft to top) of a flag or full barb
            - 'width' - width of a flag, twice the width of a full barb
            - 'emptybarb' - radius of the circle used for low magnitudes

        fill_empty : bool, default: False
            Whether the empty barbs (circles) that are drawn should be filled with
            the flag color.  If they are not filled, the center is transparent.

        rounding : bool, default: True
            Whether the vector magnitude should be rounded when allocating barb
            components.  If True, the magnitude is rounded to the nearest multiple
            of the half-barb increment.  If False, the magnitude is simply truncated
            to the next lowest multiple.

        barb_increments : dict, optional
            A dictionary of increments specifying values to associate with
            different parts of the barb. Only those values one wishes to
            override need to be included.

            - 'half' - half barbs (Default is 5)
            - 'full' - full barbs (Default is 10)
            - 'flag' - flags (default is 50)

        flip_barb : bool or array-like of bool, default: False
            Whether the lines and flags should point opposite to normal.
            Normal behavior is for the barbs and lines to point right (comes from wind
            barbs having these features point towards low pressure in the Northern
            Hemisphere).

            A single value is applied to all barbs. Individual barbs can be flipped by
            passing a bool array of the same size as *U* and *V*.

        Returns
        -------
        barbs : `~matplotlib.quiver.Barbs`

        Other Parameters
        ----------------
        **kwargs
            The barbs can further be customized using `.PolyCollection` keyword
            arguments:

              agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a
(m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa or antialiaseds: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods that have ``vmin`` and ``vmax`` kwa
rgs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          cmap: colormap or registered colormap name
          color: color or sequence of rgba tuples
          contains: callable
          edgecolor or ec or edgecolors: color or sequence of colors or 'face'
          facecolor or facecolors or fc: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or dashes or linestyles or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or linewidths or lw: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        Notes
        -----



        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    barh(y, width, height=0.8, left=None, *, align='center', **kwargs)
        Make a horizontal bar plot.

        The bars are positioned at *y* with the given *align*\ment. Their
        dimensions are given by *width* and *height*. The horizontal baseline
        is *left* (default 0).

        Each of *y*, *width*, *height*, and *left* may either be a scalar
        applying to all bars, or it may be a sequence of length N providing a
        separate value for each bar.

        Parameters
        ----------
        y : scalar or array-like
            The y coordinates of the bars. See also *align* for the
            alignment of the bars to the coordinates.

        width : scalar or array-like
            The width(s) of the bars.

        height : sequence of scalars, optional, default: 0.8
            The heights of the bars.

        left : sequence of scalars
            The x coordinates of the left sides of the bars (default: 0).

        align : {'center', 'edge'}, optional, default: 'center'
            Alignment of the base to the *y* coordinates*:

            - 'center': Center the bars on the *y* positions.
            - 'edge': Align the bottom edges of the bars with the *y*
              positions.

            To align the bars on the top edge pass a negative *height* and
            ``align='edge'``.

        Returns
        -------
        container : `.BarContainer`
            Container with all the bars and optionally errorbars.

        Other Parameters
        ----------------
        color : scalar or array-like, optional
            The colors of the bar faces.

        edgecolor : scalar or array-like, optional
            The colors of the bar edges.

        linewidth : scalar or array-like, optional
            Width of the bar edge(s). If 0, don't draw edges.

        tick_label : string or array-like, optional
            The tick labels of the bars.
            Default: None (Use default numeric labels.)

        xerr, yerr : scalar or array-like of shape(N,) or shape(2,N), optional
            If not ``None``, add horizontal / vertical errorbars to the
            bar tips. The values are +/- sizes relative to the data:

            - scalar: symmetric +/- values for all bars
            - shape(N,): symmetric +/- values for each bar
            - shape(2,N): Separate - and + values for each bar. First row
                contains the lower errors, the second row contains the
                upper errors.
            - *None*: No errorbar. (default)

            See :doc:`/gallery/statistics/errorbar_features`
            for an example on the usage of ``xerr`` and ``yerr``.

        ecolor : scalar or array-like, optional, default: 'black'
            The line color of the errorbars.

        capsize : scalar, optional
           The length of the error bar caps in points.
           Default: None, which will take the value from
           :rc:`errorbar.capsize`.

        error_kw : dict, optional
            Dictionary of kwargs to be passed to the `~.Axes.errorbar`
            method. Values of *ecolor* or *capsize* defined here take
            precedence over the independent kwargs.

        log : bool, optional, default: False
            If ``True``, set the x-axis to be log scale.

        See also
        --------
        bar: Plot a vertical bar plot.

        Notes
        -----
        The optional arguments *color*, *edgecolor*, *linewidth*,
        *xerr*, and *yerr* can be either scalars or sequences of
        length equal to the number of bars.  This enables you to use
        bar as the basis for stacked bar charts, or candlestick plots.
        Detail: *xerr* and *yerr* are passed directly to
        :meth:`errorbar`, so they can also have shape 2xN for
        independent specification of lower and upper errors.

        Other optional kwargs:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa: unknown
          capstyle: {'butt', 'round', 'projecting'}
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color: color
          contains: callable
          edgecolor or ec: color or None or 'auto'
          facecolor or fc: color or None
          figure: `.Figure`
          fill: bool
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float or None
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          visible: bool
          zorder: float

    bone()
        Set the colormap to "bone".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    box(on=None)
        Turn the axes box on or off on the current axes.

        Parameters
        ----------
        on : bool or None
            The new `~matplotlib.axes.Axes` box state. If ``None``, toggle
            the state.

        See Also
        --------
        :meth:`matplotlib.axes.Axes.set_frame_on`
        :meth:`matplotlib.axes.Axes.get_frame_on`

    boxplot(x, notch=None, sym=None, vert=None, whis=None, positions=None, widths=None, patch_artist=None, boot
strap=None, usermedians=None, conf_intervals=None, meanline=None, showmeans=None, showcaps=None, showbox=None,
showfliers=None, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=None, capprops=None,
whiskerprops=None, manage_ticks=True, autorange=False, zorder=None, *, data=None)
        Make a box and whisker plot.

        Make a box and whisker plot for each column of ``x`` or each
        vector in sequence ``x``.  The box extends from the lower to
        upper quartile values of the data, with a line at the median.
        The whiskers extend from the box to show the range of the
        data.  Flier points are those past the end of the whiskers.

        Parameters
        ----------
        x : Array or a sequence of vectors.
            The input data.

        notch : bool, optional (False)
            If `True`, will produce a notched box plot. Otherwise, a
            rectangular boxplot is produced. The notches represent the
            confidence interval (CI) around the median. See the entry
            for the ``bootstrap`` parameter for information regarding
            how the locations of the notches are computed.

            .. note::

                In cases where the values of the CI are less than the
                lower quartile or greater than the upper quartile, the
                notches will extend beyond the box, giving it a
                distinctive "flipped" appearance. This is expected
                behavior and consistent with other statistical
                visualization packages.

        sym : str, optional
            The default symbol for flier points. Enter an empty string
            ('') if you don't want to show fliers. If `None`, then the
            fliers default to 'b+'  If you want more control use the
            flierprops kwarg.

        vert : bool, optional (True)
            If `True` (default), makes the boxes vertical. If `False`,
            everything is drawn horizontally.

        whis : float, sequence, or string (default = 1.5)
            As a float, determines the reach of the whiskers to the beyond the
            first and third quartiles. In other words, where IQR is the
            interquartile range (`Q3-Q1`), the upper whisker will extend to
            last datum less than `Q3 + whis*IQR`). Similarly, the lower whisker
            will extend to the first datum greater than `Q1 - whis*IQR`.
            Beyond the whiskers, data
            are considered outliers and are plotted as individual
            points. Set this to an unreasonably high value to force the
            whiskers to show the min and max values. Alternatively, set
            this to an ascending sequence of percentile (e.g., [5, 95])
            to set the whiskers at specific percentiles of the data.
            Finally, ``whis`` can be the string ``'range'`` to force the
            whiskers to the min and max of the data.

        bootstrap : int, optional
            Specifies whether to bootstrap the confidence intervals
            around the median for notched boxplots. If ``bootstrap`` is
            None, no bootstrapping is performed, and notches are
            calculated using a Gaussian-based asymptotic approximation
            (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and
            Kendall and Stuart, 1967). Otherwise, bootstrap specifies
            the number of times to bootstrap the median to determine its
            95% confidence intervals. Values between 1000 and 10000 are
            recommended.

        usermedians : array-like, optional
            An array or sequence whose first dimension (or length) is
            compatible with ``x``. This overrides the medians computed
            by matplotlib for each element of ``usermedians`` that is not
            `None`. When an element of ``usermedians`` is None, the median
            will be computed by matplotlib as normal.

        conf_intervals : array-like, optional
            Array or sequence whose first dimension (or length) is
            compatible with ``x`` and whose second dimension is 2. When
            the an element of ``conf_intervals`` is not None, the
            notch locations computed by matplotlib are overridden
            (provided ``notch`` is `True`). When an element of
            ``conf_intervals`` is `None`, the notches are computed by the
            method specified by the other kwargs (e.g., ``bootstrap``).

        positions : array-like, optional
            Sets the positions of the boxes. The ticks and limits are
            automatically set to match the positions. Defaults to
            `range(1, N+1)` where N is the number of boxes to be drawn.

        widths : scalar or array-like
            Sets the width of each box either with a scalar or a
            sequence. The default is 0.5, or ``0.15*(distance between
            extreme positions)``, if that is smaller.

        patch_artist : bool, optional (False)
            If `False` produces boxes with the Line2D artist. Otherwise,
            boxes and drawn with Patch artists.

        labels : sequence, optional
            Labels for each dataset. Length must be compatible with
            dimensions of ``x``.

        manage_ticks : bool, optional (True)
            If True, the tick locations and labels will be adjusted to match
            the boxplot positions.

        autorange : bool, optional (False)
            When `True` and the data are distributed such that the 25th and
            75th percentiles are equal, ``whis`` is set to ``'range'`` such
            that the whisker ends are at the minimum and maximum of the data.

        meanline : bool, optional (False)
            If `True` (and ``showmeans`` is `True`), will try to render
            the mean as a line spanning the full width of the box
            according to ``meanprops`` (see below). Not recommended if
            ``shownotches`` is also True. Otherwise, means will be shown
            as points.

        zorder : scalar, optional (None)
            Sets the zorder of the boxplot.

        Other Parameters
        ----------------
        showcaps : bool, optional (True)
            Show the caps on the ends of whiskers.
        showbox : bool, optional (True)
            Show the central box.
        showfliers : bool, optional (True)
            Show the outliers beyond the caps.
        showmeans : bool, optional (False)
            Show the arithmetic means.
        capprops : dict, optional (None)
            Specifies the style of the caps.
        boxprops : dict, optional (None)
            Specifies the style of the box.
        whiskerprops : dict, optional (None)
            Specifies the style of the whiskers.
        flierprops : dict, optional (None)
            Specifies the style of the fliers.
        medianprops : dict, optional (None)
            Specifies the style of the median.
        meanprops : dict, optional (None)
            Specifies the style of the mean.

        Returns
        -------
        result : dict
          A dictionary mapping each component of the boxplot to a list
          of the :class:`matplotlib.lines.Line2D` instances
          created. That dictionary has the following keys (assuming
          vertical boxplots):

          - ``boxes``: the main body of the boxplot showing the
            quartiles and the median's confidence intervals if
            enabled.

          - ``medians``: horizontal lines at the median of each box.

          - ``whiskers``: the vertical lines extending to the most
            extreme, non-outlier data points.

          - ``caps``: the horizontal lines at the ends of the
            whiskers.

          - ``fliers``: points representing data that extend beyond
            the whiskers (fliers).

          - ``means``: points or lines representing the means.

        Notes
        -----



        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    broken_barh(xranges, yrange, *, data=None, **kwargs)
        Plot a horizontal sequence of rectangles.

        A rectangle is drawn for each element of *xranges*. All rectangles
        have the same vertical position and size defined by *yrange*.

        This is a convenience function for instantiating a
        `.BrokenBarHCollection`, adding it to the axes and autoscaling the
        view.

        Parameters
        ----------
        xranges : sequence of tuples (*xmin*, *xwidth*)
            The x-positions and extends of the rectangles. For each tuple
            (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* +
            *xwidth*.
        yranges : (*ymin*, *ymax*)
            The y-position and extend for all the rectangles.

        Other Parameters
        ----------------
        **kwargs : :class:`.BrokenBarHCollection` properties

            Each *kwarg* can be either a single argument applying to all
            rectangles, e.g.::

                facecolors='black'

            or a sequence of arguments over which is cycled, e.g.::

                facecolors=('black', 'blue')

            would create interleaving black and blue rectangles.

            Supported keywords:

              agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a
 (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa or antialiaseds: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods that have ``vmin`` and ``vmax`` kwa
rgs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          cmap: colormap or registered colormap name
          color: color or sequence of rgba tuples
          contains: callable
          edgecolor or ec or edgecolors: color or sequence of colors or 'face'
          facecolor or facecolors or fc: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or dashes or linestyles or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or linewidths or lw: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        Returns
        -------
        collection : A :class:`~.collections.BrokenBarHCollection`

        Notes
        -----



        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    cla()
        Clear the current axes.

    clabel(CS, *args, **kwargs)
        Label a contour plot.

        Call signature::

          clabel(cs, [levels,] **kwargs)

        Adds labels to line contours in *cs*, where *cs* is a
        :class:`~matplotlib.contour.ContourSet` object returned by
        ``contour()``.

        Parameters
        ----------
        cs : `.ContourSet`
            The ContourSet to label.

        levels : array-like, optional
            A list of level values, that should be labeled. The list must be
            a subset of ``cs.levels``. If not given, all levels are labeled.

        fontsize : string or float, optional
            Size in points or relative size e.g., 'smaller', 'x-large'.
            See `.Text.set_size` for accepted string values.

        colors : color-spec, optional
            The label colors:

            - If *None*, the color of each label matches the color of
              the corresponding contour.

            - If one string color, e.g., *colors* = 'r' or *colors* =
              'red', all labels will be plotted in this color.

            - If a tuple of matplotlib color args (string, float, rgb, etc),
              different labels will be plotted in different colors in the order
              specified.

        inline : bool, optional
            If ``True`` the underlying contour is removed where the label is
            placed. Default is ``True``.

        inline_spacing : float, optional
            Space in pixels to leave on each side of label when
            placing inline. Defaults to 5.

            This spacing will be exact for labels at locations where the
            contour is straight, less so for labels on curved contours.

        fmt : string or dict, optional
            A format string for the label. Default is '%1.3f'

            Alternatively, this can be a dictionary matching contour
            levels with arbitrary strings to use for each contour level
            (i.e., fmt[level]=string), or it can be any callable, such
            as a :class:`~matplotlib.ticker.Formatter` instance, that
            returns a string when called with a numeric contour level.

        manual : bool or iterable, optional
            If ``True``, contour labels will be placed manually using
            mouse clicks. Click the first button near a contour to
            add a label, click the second button (or potentially both
            mouse buttons at once) to finish adding labels. The third
            button can be used to remove the last label added, but
            only if labels are not inline. Alternatively, the keyboard
            can be used to select label locations (enter to end label
            placement, delete or backspace act like the third mouse button,
            and any other key will select a label location).

            *manual* can also be an iterable object of x,y tuples.
            Contour labels will be created as if mouse is clicked at each
            x,y positions.

        rightside_up : bool, optional
            If ``True``, label rotations will always be plus
            or minus 90 degrees from level. Default is ``True``.

        use_clabeltext : bool, optional
            If ``True``, `.ClabelText` class (instead of `.Text`) is used to
            create labels. `ClabelText` recalculates rotation angles
            of texts during the drawing time, therefore this can be used if
            aspect of the axes changes. Default is ``False``.

        Returns
        -------
        labels
            A list of `.Text` instances for the labels.

    clf()
        Clear the current figure.

    clim(vmin=None, vmax=None)
        Set the color limits of the current image.

        If either *vmin* or *vmax* is None, the image min/max respectively
        will be used for color scaling.

        If you want to set the clim of multiple images, use
        `~.ScalarMappable.set_clim` on every image, for example::

          for im in gca().get_images():
              im.set_clim(0, 0.5)

    close(fig=None)
        Close a figure window.

        Parameters
        ----------
        fig : None or int or str or `.Figure`
            The figure to close. There are a number of ways to specify this:

            - *None*: the current figure
            - `.Figure`: the given `.Figure` instance
            - ``int``: a figure number
            - ``str``: a figure name
            - 'all': all figures

    cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x0B070468>, window=<function window_h
anning at 0x0B05E9C0>, noverlap=0, pad_to=None, sides='default', scale_by_freq=None, *, data=None, **kwargs)
        Plot the coherence between *x* and *y*.

        Plot the coherence between *x* and *y*.  Coherence is the
        normalized cross spectral density:

        .. math::

          C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}

        Parameters
        ----------
        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*.  To create window vectors see
            `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
            `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
            default is `window_hanning`.  If a function is passed as the argument,
            it must take a data segment as an argument and return the windowed
            version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.
```

```
        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  This can be different from *NFFT*, which
            specifies the number of data points used.  While not increasing
            the actual resolution of the spectrum (the minimum distance between
            resolvable peaks), this can give more points in the plot,
            allowing for more detail. This corresponds to the *n* parameter
            in the call to fft(). The default is None, which sets *pad_to*
            equal to *NFFT*

        NFFT : int
            The number of data points used in each block for the FFT.
            A power 2 is most efficient.  The default value is 256.
            This should *NOT* be used to get zero padding, or the scaling of the
            result will be incorrect. Use *pad_to* for this instead.

        detrend : {'none', 'mean', 'linear'} or callable, default 'none'
            The function applied to each segment before fft-ing, designed to
            remove the mean or linear trend.  Unlike in MATLAB, where the
            *detrend* parameter is a vector, in Matplotlib is it a function.
            The :mod:`~matplotlib.mlab` module defines `.detrend_none`,
            `.detrend_mean`, and `.detrend_linear`, but you can use a custom
            function as well.  You can also use a string to choose one of the
            functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`.
            'linear' calls `.detrend_linear`.

        scale_by_freq : bool, optional
            Specifies whether the resulting density values should be scaled
            by the scaling frequency, which gives density in units of Hz^-1.
            This allows for integration over the returned frequency values.
            The default is True for MATLAB compatibility.

        noverlap : int
            The number of points of overlap between blocks.  The
            default value is 0 (no overlap).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.


        Returns
        -------
        Cxy : 1-D array
            The coherence vector.

        freqs : 1-D array
            The frequencies for the elements in *Cxy*.

        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m, n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        References
        ----------
        Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
        John Wiley & Sons (1986)

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    colorbar(mappable=None, cax=None, ax=None, **kw)
        Add a colorbar to a plot.

        Function signatures for the :mod:`~matplotlib.pyplot` interface; all
        but the first are also method signatures for the
        :meth:`~matplotlib.figure.Figure.colorbar` method::

          colorbar(**kwargs)
          colorbar(mappable, **kwargs)
          colorbar(mappable, cax=cax, **kwargs)
          colorbar(mappable, ax=ax, **kwargs)

        Parameters
        ----------
        mappable
            The `matplotlib.cm.ScalarMappable` (i.e., `~matplotlib.image.Image`,
            `~matplotlib.contour.ContourSet`, etc.) described by this colorbar.
            This argument is mandatory for the `.Figure.colorbar` method but optional
            for the `.pyplot.colorbar` function, which sets the default to the current
            image.

            Note that one can create a `ScalarMappable` "on-the-fly" to generate
            colorbars not attached to a previously drawn artist, e.g. ::

                fig.colorbar(cm.ScalarMappable(norm=norm, cmap=cmap), ax=ax)

        cax : :class:`~matplotlib.axes.Axes` object, optional
            Axes into which the colorbar will be drawn.

        ax : :class:`~matplotlib.axes.Axes`, list of Axes, optional
            Parent axes from which space for a new colorbar axes will be stolen.
            If a list of axes is given they will all be resized to make room for the
            colorbar axes.

        use_gridspec : bool, optional
            If *cax* is ``None``, a new *cax* is created as an instance of Axes.  If
            *ax* is an instance of Subplot and *use_gridspec* is ``True``, *cax* is
            created as an instance of Subplot using the :mod:`~.gridspec` module.

        Returns
        -------
        colorbar : `~matplotlib.colorbar.Colorbar`
            See also its base class, `~matplotlib.colorbar.ColorbarBase`.  Use
            `~.ColorbarBase.set_label` to label the colorbar.

        Notes
        -----
        Additional keyword arguments are of two kinds:

          axes properties:


            ============= ====================================================
            Property      Description
            ============= ====================================================
            *orientation* vertical or horizontal
            *fraction*    0.15; fraction of original axes to use for colorbar
            *pad*         0.05 if vertical, 0.15 if horizontal; fraction
                          of original axes between colorbar and new image axes
            *shrink*      1.0; fraction by which to multiply the size of the colorbar
            *aspect*      20; ratio of long to short dimensions
            *anchor*      (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal;
                          the anchor point of the colorbar axes
            *panchor*     (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal;
                          the anchor point of the colorbar parent axes. If
                          False, the parent axes' anchor will be unchanged
            ============= ====================================================


          colorbar properties:


            ============  ====================================================
            Property      Description
            ============  ====================================================
            *extend*      [ 'neither' | 'both' | 'min' | 'max' ]
                          If not 'neither', make pointed end(s) for out-of-
                          range values.  These are set for a given colormap
                          using the colormap set_under and set_over methods.
            *extendfrac*  [ *None* | 'auto' | length | lengths ]
                          If set to *None*, both the minimum and maximum
                          triangular colorbar extensions with have a length of
                          5% of the interior colorbar length (this is the
                          default setting). If set to 'auto', makes the
                          triangular colorbar extensions the same lengths as
                          the interior boxes (when *spacing* is set to
                          'uniform') or the same lengths as the respective
                          adjacent interior boxes (when *spacing* is set to
                          'proportional'). If a scalar, indicates the length
                          of both the minimum and maximum triangular colorbar
                          extensions as a fraction of the interior colorbar
                          length. A two-element sequence of fractions may also
                          be given, indicating the lengths of the minimum and
                          maximum colorbar extensions respectively as a
                          fraction of the interior colorbar length.
            *extendrect*  bool
                          If *False* the minimum and maximum colorbar extensions
                          will be triangular (the default). If *True* the
                          extensions will be rectangular.
            *spacing*     [ 'uniform' | 'proportional' ]
                          Uniform spacing gives each discrete color the same
                          space; proportional makes the space proportional to
                          the data interval.
            *ticks*       [ None | list of ticks | Locator object ]
                          If None, ticks are determined automatically from the
                          input.
            *format*      [ None | format string | Formatter object ]
                          If None, the
                          :class:`~matplotlib.ticker.ScalarFormatter` is used.
                          If a format string is given, e.g., '%.3f', that is
                          used. An alternative
                          :class:`~matplotlib.ticker.Formatter` object may be
                          given instead.
            *drawedges*   bool
                          Whether to draw lines at color boundaries.
            ============  ====================================================

            The following will probably be useful only in the context of
            indexed colors (that is, when the mappable has norm=NoNorm()),
            or other unusual circumstances.

            ============   ===================================================
            Property       Description
            ============   ===================================================
            *boundaries*   None or a sequence
            *values*       None or a sequence which must be of length 1 less
                           than the sequence of *boundaries*. For each region
                           delimited by adjacent entries in *boundaries*, the
                           color mapped to the corresponding value in values
                           will be used.
            ============   ===================================================



        If *mappable* is a :class:`~matplotlib.contours.ContourSet`, its *extend*
        kwarg is included automatically.

        The *shrink* kwarg provides a simple way to scale the colorbar with respect
        to the axes. Note that if *cax* is specified, it determines the size of the
        colorbar and *shrink* and *aspect* kwargs are ignored.

        For more precise control, you can manually specify the positions of
        the axes objects in which the mappable and the colorbar are drawn.  In
        this case, do not use any of the axes properties kwargs.

        It is known that some vector graphics viewers (svg and pdf) renders white gaps
        between segments of the colorbar.  This is due to bugs in the viewers, not
        Matplotlib.  As a workaround, the colorbar can be rendered with overlapping
        segments::

            cbar = colorbar()
            cbar.solids.set_edgecolor("face")
            draw()

        However this has negative consequences in other circumstances, e.g. with
        semi-transparent images (alpha < 1) and colorbar extensions; therefore, this
        workaround is not used by default (see issue #1188).

    colormaps()
        Matplotlib provides a number of colormaps, and others can be added using
        :func:`~matplotlib.cm.register_cmap`.  This function documents the built-in
        colormaps, and will also return a list of all registered colormaps if
        called.

        You can set the colormap for an image, pcolor, scatter, etc,
        using a keyword argument::

          imshow(X, cmap=cm.hot)

        or using the :func:`set_cmap` function::

          imshow(X)
          pyplot.set_cmap('hot')
          pyplot.set_cmap('jet')

        In interactive mode, :func:`set_cmap` will update the colormap post-hoc,
        allowing you to see which one works best for your data.

        All built-in colormaps can be reversed by appending ``_r``: For instance,
        ``gray_r`` is the reverse of ``gray``.

        There are several common color schemes used in visualization:

        Sequential schemes
          for unipolar data that progresses from low to high
        Diverging schemes
          for bipolar data that emphasizes positive or negative deviations from a
          central value
        Cyclic schemes
          for plotting values that wrap around at the endpoints, such as phase
          angle, wind direction, or time of day
        Qualitative schemes
          for nominal data that has no inherent ordering, where color is used
          only to distinguish categories

        Matplotlib ships with 4 perceptually uniform color maps which are
        the recommended color maps for sequential data:

          =========   ===================================================
          Colormap    Description
          =========   ===================================================
          inferno     perceptually uniform shades of black-red-yellow
          magma       perceptually uniform shades of black-red-white
          plasma      perceptually uniform shades of blue-red-yellow
          viridis     perceptually uniform shades of blue-green-yellow
          =========   ===================================================

        The following colormaps are based on the `ColorBrewer
        <http://colorbrewer2.org>`_ color specifications and designs developed by
        Cynthia Brewer:

        ColorBrewer Diverging (luminance is highest at the midpoint, and
        decreases towards differently-colored endpoints):

          ========  ===================================
          Colormap  Description
          ========  ===================================
          BrBG      brown, white, blue-green
          PiYG      pink, white, yellow-green
          PRGn      purple, white, green
          PuOr      orange, white, purple
          RdBu      red, white, blue
          RdGy      red, white, gray
          RdYlBu    red, yellow, blue
          RdYlGn    red, yellow, green
          Spectral  red, orange, yellow, green, blue
          ========  ===================================

        ColorBrewer Sequential (luminance decreases monotonically):

          ========  ====================================
          Colormap  Description
          ========  ====================================
          Blues     white to dark blue
          BuGn      white, light blue, dark green
          BuPu      white, light blue, dark purple
          GnBu      white, light green, dark blue
          Greens    white to dark green
          Greys     white to black (not linear)
          Oranges   white, orange, dark brown
          OrRd      white, orange, dark red
          PuBu      white, light purple, dark blue
          PuBuGn    white, light purple, dark green
          PuRd      white, light purple, dark red
          Purples   white to dark purple
          RdPu      white, pink, dark purple
          Reds      white to dark red
          YlGn      light yellow, dark green
          YlGnBu    light yellow, light green, dark blue
          YlOrBr    light yellow, orange, dark brown
          YlOrRd    light yellow, orange, dark red
          ========  ====================================

        ColorBrewer Qualitative:

        (For plotting nominal data, :class:`ListedColormap` is used,
        not :class:`LinearSegmentedColormap`.  Different sets of colors are
        recommended for different numbers of categories.)

        * Accent
        * Dark2
        * Paired
        * Pastel1
        * Pastel2
        * Set1
        * Set2
        * Set3

        A set of colormaps derived from those of the same name provided
        with Matlab are also included:

          =========   =======================================================
          Colormap    Description
          =========   =======================================================
          autumn      sequential linearly-increasing shades of red-orange-yellow
          bone        sequential increasing black-white color map with
                      a tinge of blue, to emulate X-ray film
          cool        linearly-decreasing shades of cyan-magenta
          copper      sequential increasing shades of black-copper
          flag        repetitive red-white-blue-black pattern (not cyclic at
                      endpoints)
          gray        sequential linearly-increasing black-to-white
                      grayscale
          hot         sequential black-red-yellow-white, to emulate blackbody
                      radiation from an object at increasing temperatures
          jet         a spectral map with dark endpoints, blue-cyan-yellow-red;
                      based on a fluid-jet simulation by NCSA [#]_
          pink        sequential increasing pastel black-pink-white, meant
                      for sepia tone colorization of photographs
          prism       repetitive red-yellow-green-blue-purple-...-green pattern
                      (not cyclic at endpoints)
          spring      linearly-increasing shades of magenta-yellow
          summer      sequential linearly-increasing shades of green-yellow
          winter      linearly-increasing shades of blue-green
          =========   =======================================================

        A set of palettes from the `Yorick scientific visualisation
        package <https://dhmunro.github.io/yorick-doc/>`_, an evolution of
        the GIST package, both by David H. Munro are included:

          ============  =======================================================
          Colormap      Description
          ============  =======================================================
          gist_earth    mapmaker's colors from dark blue deep ocean to green
                        lowlands to brown highlands to white mountains
          gist_heat     sequential increasing black-red-orange-white, to emulate
                        blackbody radiation from an iron bar as it grows hotter
          gist_ncar     pseudo-spectral black-blue-green-yellow-red-purple-white
                        colormap from National Center for Atmospheric
                        Research [#]_
          gist_rainbow  runs through the colors in spectral order from red to
                        violet at full saturation (like *hsv* but not cyclic)
          gist_stern    "Stern special" color table from Interactive Data
                        Language software
          ============  =======================================================

        A set of cyclic color maps:

          ================  =================================================
          Colormap          Description
          ================  =================================================
          hsv               red-yellow-green-cyan-blue-magenta-red, formed by
                            changing the hue component in the HSV color space
          twilight          perceptually uniform shades of
                            white-blue-black-red-white
          twilight_shifted  perceptually uniform shades of
                            black-blue-white-red-black
          ================  =================================================


        Other miscellaneous schemes:

          ============= =======================================================
          Colormap      Description
          ============= =======================================================
          afmhot        sequential black-orange-yellow-white blackbody
                        spectrum, commonly used in atomic force microscopy
          brg           blue-red-green
          bwr           diverging blue-white-red
          coolwarm      diverging blue-gray-red, meant to avoid issues with 3D
                        shading, color blindness, and ordering of colors [#]_
          CMRmap        "Default colormaps on color images often reproduce to
                        confusing grayscale images. The proposed colormap
                        maintains an aesthetically pleasing color image that
                        automatically reproduces to a monotonic grayscale with
                        discrete, quantifiable saturation levels." [#]_
          cubehelix     Unlike most other color schemes cubehelix was designed
                        by D.A. Green to be monotonically increasing in terms
                        of perceived brightness. Also, when printed on a black
                        and white postscript printer, the scheme results in a
                        greyscale with monotonically increasing brightness.
                        This color scheme is named cubehelix because the r,g,b
                        values produced can be visualised as a squashed helix
                        around the diagonal in the r,g,b color cube.
          gnuplot       gnuplot's traditional pm3d scheme
                        (black-blue-red-yellow)
          gnuplot2      sequential color printable as gray
                        (black-blue-violet-yellow-white)
          ocean         green-blue-white
          rainbow       spectral purple-blue-green-yellow-orange-red colormap
                        with diverging luminance
          seismic       diverging blue-white-red
          nipy_spectral black-purple-blue-green-yellow-red-white spectrum,
                        originally from the Neuroimaging in Python project
          terrain       mapmaker's colors, blue-green-yellow-brown-white,
                        originally from IGOR Pro
          ============= =======================================================

        The following colormaps are redundant and may be removed in future
        versions.  It's recommended to use the names in the descriptions
        instead, which produce identical output:

          =========  =======================================================
          Colormap   Description
          =========  =======================================================
          gist_gray  identical to *gray*
          gist_yarg  identical to *gray_r*
          binary     identical to *gray_r*
          =========  =======================================================

        .. rubric:: Footnotes

        .. [#] Rainbow colormaps, ``jet`` in particular, are considered a poor
          choice for scientific visualization by many researchers: `Rainbow Color
          Map (Still) Considered Harmful
          <http://ieeexplore.ieee.org/document/4118486/?arnumber=4118486>`_

        .. [#] Resembles "BkBlAqGrYeOrReViWh200" from NCAR Command
          Language. See `Color Table Gallery
          <https://www.ncl.ucar.edu/Document/Graphics/color_table_gallery.shtml>`_

        .. [#] See `Diverging Color Maps for Scientific Visualization
          <http://www.kennethmoreland.com/color-maps/>`_ by Kenneth Moreland.

        .. [#] See `A Color Map for Effective Black-and-White Rendering of
          Color-Scale Images
          <https://www.mathworks.com/matlabcentral/fileexchange/2662-cmrmap-m>`_
          by Carey Rappaport

    connect(s, func)
        Connect event with string *s* to *func*.  The signature of *func* is::

          def func(event)

        where event is a :class:`matplotlib.backend_bases.Event`.  The
        following events are recognized

        - 'button_press_event'
        - 'button_release_event'
        - 'draw_event'
        - 'key_press_event'
        - 'key_release_event'
        - 'motion_notify_event'
        - 'pick_event'
        - 'resize_event'
        - 'scroll_event'
        - 'figure_enter_event',
        - 'figure_leave_event',
        - 'axes_enter_event',
        - 'axes_leave_event'
        - 'close_event'

        For the location events (button and key press/release), if the
        mouse is over the axes, the variable ``event.inaxes`` will be
        set to the :class:`~matplotlib.axes.Axes` the event occurs is
        over, and additionally, the variables ``event.xdata`` and
        ``event.ydata`` will be defined.  This is the mouse location
        in data coords.  See
        :class:`~matplotlib.backend_bases.KeyEvent` and
        :class:`~matplotlib.backend_bases.MouseEvent` for more info.

        Return value is a connection id that can be used with
        :meth:`~matplotlib.backend_bases.Event.mpl_disconnect`.

        Examples
        --------
        Usage::

            def on_press(event):
                print('you pressed', event.button, event.xdata, event.ydata)

            cid = canvas.mpl_connect('button_press_event', on_press)

    contour(*args, data=None, **kwargs)
        Plot contours.

        Call signature::

            contour([X, Y,] Z, [levels], **kwargs)

        `.contour` and `.contourf` draw contour lines and filled contours,
        respectively.  Except as noted, function signatures and return values
        are the same for both versions.

        Parameters
        ----------
        X, Y : array-like, optional
            The coordinates of the values in *Z*.

            *X* and *Y* must both be 2-D with the same shape as *Z* (e.g.
            created via `numpy.meshgrid`), or they must both be 1-D such
            that ``len(X) == M`` is the number of columns in *Z* and
            ``len(Y) == N`` is the number of rows in *Z*.

            If not given, they are assumed to be integer indices, i.e.
            ``X = range(M)``, ``Y = range(N)``.

        Z : array-like(N, M)
            The height values over which the contour is drawn.

        levels : int or array-like, optional
            Determines the number and positions of the contour lines / regions.

            If an int *n*, use *n* data intervals; i.e. draw *n+1* contour
            lines. The level heights are automatically chosen.

            If array-like, draw contour lines at the specified levels.
            The values must be in increasing order.

        Returns
        -------
        c : `~.contour.QuadContourSet`

        Other Parameters
        ----------------
        corner_mask : bool, optional
            Enable/disable corner masking, which only has an effect if *Z* is
            a masked array.  If ``False``, any quad touching a masked point is
            masked out.  If ``True``, only the triangular corners of quads
            nearest those points are always masked out, other triangular
            corners comprising three unmasked points are contoured as usual.

            Defaults to :rc:`contour.corner_mask`, which defaults to ``True``.

        colors : color string or sequence of colors, optional
            The colors of the levels, i.e. the lines for `.contour` and the
            areas for `.contourf`.

            The sequence is cycled for the levels in ascending order. If the
            sequence is shorter than the number of levels, it's repeated.

            As a shortcut, single color strings may be used in place of
            one-element lists, i.e. ``'red'`` instead of ``['red']`` to color
            all levels with the same color. This shortcut does only work for
            color strings, not for other ways of specifying colors.

            By default (value *None*), the colormap specified by *cmap*
            will be used.

        alpha : float, optional
            The alpha blending value, between 0 (transparent) and 1 (opaque).

        cmap : str or `.Colormap`, optional
            A `.Colormap` instance or registered colormap name. The colormap
            maps the level values to colors.
            Defaults to :rc:`image.cmap`.

            If given, *colors* take precedence over *cmap*.

        norm : `~matplotlib.colors.Normalize`, optional
            If a colormap is used, the `.Normalize` instance scales the level
            values to the canonical colormap range [0, 1] for mapping to
            colors. If not given, the default linear scaling is used.

        vmin, vmax : float, optional
            If not *None*, either or both of these values will be supplied to
            the `.Normalize` instance, overriding the default color scaling
            based on *levels*.

        origin : {*None*, 'upper', 'lower', 'image'}, optional
            Determines the orientation and exact position of *Z* by specifying
            the position of ``Z[0, 0]``.  This is only relevant, if *X*, *Y*
            are not given.

            - *None*: ``Z[0, 0]`` is at X=0, Y=0 in the lower left corner.
            - 'lower': ``Z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
            - 'upper': ``Z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left
              corner.
            - 'image': Use the value from :rc:`image.origin`.

        extent : (x0, x1, y0, y1), optional
            If *origin* is not *None*, then *extent* is interpreted as in
            `.imshow`: it gives the outer pixel boundaries. In this case, the
            position of Z[0,0] is the center of the pixel, not a corner. If
            *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0],
            and (*x1*, *y1*) is the position of Z[-1,-1].

            This argument is ignored if *X* and *Y* are specified in the call
            to contour.

        locator : ticker.Locator subclass, optional
            The locator is used to determine the contour levels if they
            are not given explicitly via *levels*.
            Defaults to `~.ticker.MaxNLocator`.

        extend : {'neither', 'both', 'min', 'max'}, optional, default: 'neither'
            Determines the ``contourf``-coloring of values that are outside the
            *levels* range.

            If 'neither', values outside the *levels* range are not colored.
            If 'min', 'max' or 'both', color the values below, above or below
            and above the *levels* range.

            Values below ``min(levels)`` and above ``max(levels)`` are mapped
            to the under/over values of the `.Colormap`. Note, that most
            colormaps do not have dedicated colors for these by default, so
            that the over and under values are the edge values of the colormap.
            You may want to set these values explicitly using
            `.Colormap.set_under` and `.Colormap.set_over`.

            .. note::

                An exising `.QuadContourSet` does not get notified if
                properties of its colormap are changed. Therefore, an explicit
                call `.QuadContourSet.changed()` is needed after modifying the
                colormap. The explicit call can be left out, if a colorbar is
                assigned to the `.QuadContourSet` because it internally calls
                `.QuadContourSet.changed()`.

            Example::

                x = np.arange(1, 10)
                y = x.reshape(-1, 1)
                h = x * y

                cs = plt.contourf(h, levels=[10, 30, 50],
                    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
                cs.cmap.set_over('red')
                cs.cmap.set_under('blue')
                cs.changed()

        xunits, yunits : registered units, optional
            Override axis units by specifying an instance of a
            :class:`matplotlib.units.ConversionInterface`.

        antialiased : bool, optional
            Enable antialiasing, overriding the defaults.  For
            filled contours, the default is *True*.  For line contours,
            it is taken from :rc:`lines.antialiased`.

        Nchunk : int >= 0, optional
            If 0, no subdivision of the domain.  Specify a positive integer to
            divide the domain into subdomains of *nchunk* by *nchunk* quads.
            Chunking reduces the maximum length of polygons generated by the
            contouring algorithm which reduces the rendering workload passed
            on to the backend and also requires slightly less RAM.  It can
            however introduce rendering artifacts at chunk boundaries depending
            on the backend, the *antialiased* flag and value of *alpha*.

        linewidths : float or sequence of float, optional
            *Only applies to* `.contour`.

            The line width of the contour lines.

            If a number, all levels will be plotted with this linewidth.

            If a sequence, the levels in ascending order will be plotted with
            the linewidths in the order specified.

            Defaults to :rc:`lines.linewidth`.

        linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
            *Only applies to* `.contour`.

            If *linestyles* is *None*, the default is 'solid' unless the lines
            are monochrome.  In that case, negative contours will take their
            linestyle from :rc:`contour.negative_linestyle` setting.

            *linestyles* can also be an iterable of the above strings
            specifying a set of linestyles to be used. If this
            iterable is shorter than the number of contour levels
            it will be repeated as necessary.

        hatches : List[str], optional
            *Only applies to* `.contourf`.

            A list of cross hatch patterns to use on the filled areas.
            If None, no hatching will be added to the contour.
            Hatching is supported in the PostScript, PDF, SVG and Agg
            backends only.

        Notes
        -----
        1. `.contourf` differs from the MATLAB version in that it does not draw
           the polygon edges. To draw edges, add line contours with calls to
           `.contour`.

        2. `.contourf` fills intervals that are closed at the top; that is, for
           boundaries *z1* and *z2*, the filled region is::

              z1 < Z <= z2

           except for the lowest interval, which is closed on both sides (i.e.
           it includes the lowest value).

    contourf(*args, data=None, **kwargs)
        Plot contours.

        Call signature::

            contour([X, Y,] Z, [levels], **kwargs)

        `.contour` and `.contourf` draw contour lines and filled contours,
        respectively.  Except as noted, function signatures and return values
        are the same for both versions.

        Parameters
        ----------
        X, Y : array-like, optional
            The coordinates of the values in *Z*.

            *X* and *Y* must both be 2-D with the same shape as *Z* (e.g.
            created via `numpy.meshgrid`), or they must both be 1-D such
            that ``len(X) == M`` is the number of columns in *Z* and
            ``len(Y) == N`` is the number of rows in *Z*.

            If not given, they are assumed to be integer indices, i.e.
            ``X = range(M)``, ``Y = range(N)``.

        Z : array-like(N, M)
            The height values over which the contour is drawn.

        levels : int or array-like, optional
            Determines the number and positions of the contour lines / regions.

            If an int *n*, use *n* data intervals; i.e. draw *n+1* contour
            lines. The level heights are automatically chosen.

            If array-like, draw contour lines at the specified levels.
            The values must be in increasing order.

        Returns
        -------
        c : `~.contour.QuadContourSet`

        Other Parameters
        ----------------
        corner_mask : bool, optional
            Enable/disable corner masking, which only has an effect if *Z* is
            a masked array.  If ``False``, any quad touching a masked point is
            masked out.  If ``True``, only the triangular corners of quads
            nearest those points are always masked out, other triangular
            corners comprising three unmasked points are contoured as usual.

            Defaults to :rc:`contour.corner_mask`, which defaults to ``True``.

        colors : color string or sequence of colors, optional
            The colors of the levels, i.e. the lines for `.contour` and the
            areas for `.contourf`.

            The sequence is cycled for the levels in ascending order. If the
            sequence is shorter than the number of levels, it's repeated.

            As a shortcut, single color strings may be used in place of
            one-element lists, i.e. ``'red'`` instead of ``['red']`` to color
            all levels with the same color. This shortcut does only work for
            color strings, not for other ways of specifying colors.

            By default (value *None*), the colormap specified by *cmap*
            will be used.

        alpha : float, optional
            The alpha blending value, between 0 (transparent) and 1 (opaque).

        cmap : str or `.Colormap`, optional
            A `.Colormap` instance or registered colormap name. The colormap
            maps the level values to colors.
            Defaults to :rc:`image.cmap`.

            If given, *colors* take precedence over *cmap*.

        norm : `~matplotlib.colors.Normalize`, optional
            If a colormap is used, the `.Normalize` instance scales the level
            values to the canonical colormap range [0, 1] for mapping to
            colors. If not given, the default linear scaling is used.

        vmin, vmax : float, optional
            If not *None*, either or both of these values will be supplied to
            the `.Normalize` instance, overriding the default color scaling
            based on *levels*.

        origin : {*None*, 'upper', 'lower', 'image'}, optional
            Determines the orientation and exact position of *Z* by specifying
            the position of ``Z[0, 0]``.  This is only relevant, if *X*, *Y*
            are not given.

            - *None*: ``Z[0, 0]`` is at X=0, Y=0 in the lower left corner.
            - 'lower': ``Z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
            - 'upper': ``Z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left
              corner.
            - 'image': Use the value from :rc:`image.origin`.

        extent : (x0, x1, y0, y1), optional
            If *origin* is not *None*, then *extent* is interpreted as in
            `.imshow`: it gives the outer pixel boundaries. In this case, the
            position of Z[0,0] is the center of the pixel, not a corner. If
            *origin* is *None*, then (*x0*, *y0*) is the position of Z[0,0],
            and (*x1*, *y1*) is the position of Z[-1,-1].

            This argument is ignored if *X* and *Y* are specified in the call
            to contour.

        locator : ticker.Locator subclass, optional
            The locator is used to determine the contour levels if they
            are not given explicitly via *levels*.
            Defaults to `~.ticker.MaxNLocator`.

        extend : {'neither', 'both', 'min', 'max'}, optional, default: 'neither'
            Determines the ``contourf``-coloring of values that are outside the
            *levels* range.

            If 'neither', values outside the *levels* range are not colored.
            If 'min', 'max' or 'both', color the values below, above or below
            and above the *levels* range.

            Values below ``min(levels)`` and above ``max(levels)`` are mapped
            to the under/over values of the `.Colormap`. Note, that most
            colormaps do not have dedicated colors for these by default, so
            that the over and under values are the edge values of the colormap.
            You may want to set these values explicitly using
            `.Colormap.set_under` and `.Colormap.set_over`.

            .. note::

                An exising `.QuadContourSet` does not get notified if
                properties of its colormap are changed. Therefore, an explicit
                call `.QuadContourSet.changed()` is needed after modifying the
                colormap. The explicit call can be left out, if a colorbar is
                assigned to the `.QuadContourSet` because it internally calls
                `.QuadContourSet.changed()`.

            Example::

                x = np.arange(1, 10)
                y = x.reshape(-1, 1)
                h = x * y

                cs = plt.contourf(h, levels=[10, 30, 50],
                    colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
                cs.cmap.set_over('red')
                cs.cmap.set_under('blue')
                cs.changed()

        xunits, yunits : registered units, optional
            Override axis units by specifying an instance of a
            :class:`matplotlib.units.ConversionInterface`.

        antialiased : bool, optional
            Enable antialiasing, overriding the defaults.  For
            filled contours, the default is *True*.  For line contours,
            it is taken from :rc:`lines.antialiased`.

        Nchunk : int >= 0, optional
            If 0, no subdivision of the domain.  Specify a positive integer to
            divide the domain into subdomains of *nchunk* by *nchunk* quads.
            Chunking reduces the maximum length of polygons generated by the
            contouring algorithm which reduces the rendering workload passed
            on to the backend and also requires slightly less RAM.  It can
            however introduce rendering artifacts at chunk boundaries depending
            on the backend, the *antialiased* flag and value of *alpha*.

        linewidths : float or sequence of float, optional
            *Only applies to* `.contour`.

            The line width of the contour lines.

            If a number, all levels will be plotted with this linewidth.

            If a sequence, the levels in ascending order will be plotted with
            the linewidths in the order specified.

            Defaults to :rc:`lines.linewidth`.

        linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
            *Only applies to* `.contour`.

            If *linestyles* is *None*, the default is 'solid' unless the lines
            are monochrome.  In that case, negative contours will take their
            linestyle from :rc:`contour.negative_linestyle` setting.

            *linestyles* can also be an iterable of the above strings
```

```
            specifying a set of linestyles to be used. If this
            iterable is shorter than the number of contour levels
            it will be repeated as necessary.

        hatches : List[str], optional
            *Only applies to* `.contourf`.

            A list of cross hatch patterns to use on the filled areas.
            If None, no hatching will be added to the contour.
            Hatching is supported in the PostScript, PDF, SVG and Agg
            backends only.

        Notes
        -----
        1. `.contourf` differs from the MATLAB version in that it does not draw
           the polygon edges. To draw edges, add line contours with calls to
           `.contour`.

        2. `.contourf` fills intervals that are closed at the top; that is, for
           boundaries *z1* and *z2*, the filled region is::

              z1 < Z <= z2

           except for the lowest interval, which is closed on both sides (i.e.
           it includes the lowest value).

    cool()
        Set the colormap to "cool".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    copper()
        Set the colormap to "copper".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    csd(x, y, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, s
cale_by_freq=None, return_line=None, *, data=None, **kwargs)
        Plot the cross-spectral density.

        The cross spectral density :math:`P_{xy}` by Welch's average
        periodogram method.  The vectors *x* and *y* are divided into
        *NFFT* length segments.  Each segment is detrended by function
        *detrend* and windowed by function *window*.  *noverlap* gives
        the length of the overlap between segments.  The product of
        the direct FFTs of *x* and *y* are averaged over each segment
        to compute :math:`P_{xy}`, with a scaling to correct for power
        loss due to windowing.

        If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero
        padded to *NFFT*.

        Parameters
        ----------
        x, y : 1-D arrays or sequences
            Arrays or sequences containing the data.

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*.  To create window vectors see
            `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
            `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
            default is `window_hanning`.  If a function is passed as the argument,
            it must take a data segment as an argument and return the windowed
            version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  This can be different from *NFFT*, which
            specifies the number of data points used.  While not increasing
            the actual resolution of the spectrum (the minimum distance between
            resolvable peaks), this can give more points in the plot,
            allowing for more detail. This corresponds to the *n* parameter
            in the call to fft(). The default is None, which sets *pad_to*
            equal to *NFFT*

        NFFT : int
            The number of data points used in each block for the FFT.
            A power 2 is most efficient.  The default value is 256.
            This should *NOT* be used to get zero padding, or the scaling of the
            result will be incorrect. Use *pad_to* for this instead.

        detrend : {'none', 'mean', 'linear'} or callable, default 'none'
            The function applied to each segment before fft-ing, designed to
            remove the mean or linear trend.  Unlike in MATLAB, where the
            *detrend* parameter is a vector, in Matplotlib is it a function.
            The :mod:`~matplotlib.mlab` module defines `.detrend_none`,
            `.detrend_mean`, and `.detrend_linear`, but you can use a custom
            function as well.  You can also use a string to choose one of the
            functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`.
            'linear' calls `.detrend_linear`.

        scale_by_freq : bool, optional
            Specifies whether the resulting density values should be scaled
            by the scaling frequency, which gives density in units of Hz^-1.
            This allows for integration over the returned frequency values.
            The default is True for MATLAB compatibility.

        noverlap : int
            The number of points of overlap between segments.
            The default value is 0 (no overlap).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        return_line : bool
            Whether to include the line object plotted in the returned values.
            Default is False.

        Returns
        -------
        Pxy : 1-D array
            The values for the cross spectrum `P_{xy}` before scaling
            (complex valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *Pxy*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.
            Only returned if *return_line* is True.

        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        See Also
        --------
        :func:`psd`
            :func:`psd` is the equivalent to setting y=x.

        Notes
        -----
        For plotting, the power is plotted as
        :math:`10\log_{10}(P_{xy})` for decibels, though `P_{xy}` itself
        is returned.

        References
        ----------
        Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
        John Wiley & Sons (1986)

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    delaxes(ax=None)
        Remove the `Axes` *ax* (defaulting to the current axes) from its figure.

        A KeyError is raised if the axes doesn't exist.

    disconnect(cid)
        Disconnect callback id cid

        Examples
        --------
        Usage::

            cid = canvas.mpl_connect('button_press_event', on_press)
            #...later
            canvas.mpl_disconnect(cid)

    draw()
        Redraw the current figure.

        This is used to update a figure that has been altered, but not
        automatically re-drawn.  If interactive mode is on (:func:`.ion()`), this
        should be only rarely needed, but there may be ways to modify the state of
        a figure without marking it as `stale`.  Please report these cases as
        bugs.

        A more object-oriented alternative, given any
        :class:`~matplotlib.figure.Figure` instance, :attr:`fig`, that
        was created using a :mod:`~matplotlib.pyplot` function, is::

            fig.canvas.draw_idle()

    errorbar(x, y, yerr=None, xerr=None, fmt='', ecolor=None, elinewidth=None, capsize=None, barsabove=False, l
olims=False, uplims=False, xlolims=False, xuplims=False, errorevery=1, capthick=None, *, data=None, **kwargs)
        Plot y versus x as lines and/or markers with attached errorbars.

        *x*, *y* define the data locations, *xerr*, *yerr* define the errorbar
        sizes. By default, this draws the data markers/lines as well the
        errorbars. Use fmt='none' to draw errorbars without any data markers.

        Parameters
        ----------
        x, y : scalar or array-like
            The data positions.

        xerr, yerr : scalar or array-like, shape(N,) or shape(2,N), optional
            The errorbar sizes:

            - scalar: Symmetric +/- values for all data points.
            - shape(N,): Symmetric +/-values for each data point.
            - shape(2,N): Separate - and + values for each bar. First row
                contains the lower errors, the second row contains the
                upper errors.
            - *None*: No errorbar.

            Note that all error arrays should have *positive* values.

            See :doc:`/gallery/statistics/errorbar_features`
            for an example on the usage of ``xerr`` and ``yerr``.

        fmt : plot format string, optional, default: ''
            The format for the data points / data lines. See `.plot` for
            details.

            Use 'none' (case insensitive) to plot errorbars without any data
            markers.

        ecolor : mpl color, optional, default: None
            A matplotlib color arg which gives the color the errorbar lines.
            If None, use the color of the line connecting the markers.

        elinewidth : scalar, optional, default: None
            The linewidth of the errorbar lines. If None, the linewidth of
            the current style is used.

        capsize : scalar, optional, default: None
            The length of the error bar caps in points. If None, it will take
            the value from :rc:`errorbar.capsize`.

        capthick : scalar, optional, default: None
            An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*).
            This setting is a more sensible name for the property that
            controls the thickness of the error bar cap in points. For
            backwards compatibility, if *mew* or *markeredgewidth* are given,
            then they will over-ride *capthick*. This may change in future
            releases.

        barsabove : bool, optional, default: False
            If True, will plot the errorbars above the plot
            symbols. Default is below.

        lolims, uplims, xlolims, xuplims : bool, optional, default: False
            These arguments can be used to indicate that a value gives only
            upper/lower limits. In that case a caret symbol is used to
            indicate this. *lims*-arguments may be of the same type as *xerr*
            and *yerr*.  To use limits with inverted axes, :meth:`set_xlim`
            or :meth:`set_ylim` must be called before :meth:`errorbar`.

        errorevery : positive integer, optional, default: 1
            Subsamples the errorbars. e.g., if errorevery=5, errorbars for
            every 5-th datapoint will be plotted. The data plot itself still
            shows all data points.

        Returns
        -------
        container : :class:`~.container.ErrorbarContainer`
            The container contains:

            - plotline: :class:`~matplotlib.lines.Line2D` instance of
              x, y plot markers and/or line.
            - caplines: A tuple of :class:`~matplotlib.lines.Line2D` instances
              of the error bar caps.
            - barlinecols: A tuple of
              :class:`~matplotlib.collections.LineCollection` with the
              horizontal and vertical error ranges.

        Other Parameters
        ----------------
        **kwargs
            All other keyword arguments are passed on to the plot
            command for the markers. For example, this code makes big red
            squares with thick green edges::

                x,y,yerr = rand(3,10)
                errorbar(x, y, yerr, marker='s', mfc='red',
                         mec='green', ms=20, mew=4)

            where *mfc*, *mec*, *ms* and *mew* are aliases for the longer
            property names, *markerfacecolor*, *markeredgecolor*, *markersize*
            and *markeredgewidth*.

            Valid kwargs for the marker properties are `.Lines2D` properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x', 'xerr', 'y', 'yerr'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    eventplot(positions, orientation='horizontal', lineoffsets=1, linelengths=1, linewidths=None, colors=None,
 linestyles='solid', *, data=None, **kwargs)
        Plot identical parallel lines at the given positions.

        *positions* should be a 1D or 2D array-like object, with each row
        corresponding to a row or column of lines.

        This type of plot is commonly used in neuroscience for representing
        neural events, where it is usually called a spike raster, dot raster,
        or raster plot.

        However, it is useful in any situation where you wish to show the
        timing or position of multiple sets of discrete events, such as the
        arrival times of people to a business on each day of the month or the
        date of hurricanes each year of the last century.

        Parameters
        ----------
        positions : 1D or 2D array-like object
            Each value is an event. If *positions* is a 2D array-like, each
            row corresponds to a row or a column of lines (depending on the
            *orientation* parameter).

        orientation : {'horizontal', 'vertical'}, optional
            Controls the direction of the event collections:

                - 'horizontal' : the lines are arranged horizontally in rows,
                  and are vertical.
                - 'vertical' : the lines are arranged vertically in columns,
                  and are horizontal.

        lineoffsets : scalar or sequence of scalars, optional, default: 1
            The offset of the center of the lines from the origin, in the
            direction orthogonal to *orientation*.

        linelengths : scalar or sequence of scalars, optional, default: 1
            The total height of the lines (i.e. the lines stretches from
            ``lineoffset - linelength/2`` to ``lineoffset + linelength/2``).

        linewidths : scalar, scalar sequence or None, optional, default: None
            The line width(s) of the event lines, in points. If it is None,
            defaults to its rcParams setting.

        colors : color, sequence of colors or None, optional, default: None
            The color(s) of the event lines. If it is None, defaults to its
            rcParams setting.

        linestyles : str or tuple or a sequence of such values, optional
            Default is 'solid'. Valid strings are ['solid', 'dashed',
            'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples
            should be of the form::

                (offset, onoffseq),

            where *onoffseq* is an even length tuple of on and off ink
            in points.

        **kwargs : optional
            Other keyword arguments are line collection properties.  See
            :class:`~matplotlib.collections.LineCollection` for a list of
            the valid properties.

        Returns
        -------

        list : A list of :class:`~.collections.EventCollection` objects.
            Contains the :class:`~.collections.EventCollection` that
            were added.

        Notes
        -----

        For *linelengths*, *linewidths*, *colors*, and *linestyles*, if only
        a single value is given, that value is applied to all lines.  If an
        array-like is given, it must have the same length as *positions*, and
        each value will be applied to the corresponding row of the array.

        Examples
        --------

        .. plot:: gallery/lines_bars_and_markers/eventplot_demo.py

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'colors', 'linelengths', 'lineoffsets', 'linestyles', 'li
newidths', 'positions'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    figimage(*args, **kwargs)
        Add a non-resampled image to the figure.

        The image is attached to the lower or upper left corner depending on
        *origin*.

        Parameters
        ----------
        X
            The image data. This is an array of one of the following shapes:

            - MxN: luminance (grayscale) values
            - MxNx3: RGB values
            - MxNx4: RGBA values

        xo, yo : int
            The *x*/*y* image offset in pixels.

        alpha : None or float
            The alpha blending value.

        norm : :class:`matplotlib.colors.Normalize`
            A :class:`.Normalize` instance to map the luminance to the
            interval [0, 1].

        cmap : str or :class:`matplotlib.colors.Colormap`
            The colormap to use. Default: :rc:`image.cmap`.

        vmin, vmax : scalar
            If *norm* is not given, these values set the data limits for the
            colormap.

        origin : {'upper', 'lower'}
            Indicates where the [0, 0] index of the array is in the upper left
            or lower left corner of the axes. Defaults to :rc:`image.origin`.

        resize : bool
            If *True*, resize the figure to match the given image size.

        Returns
        -------
        :class:`matplotlib.image.FigureImage`

        Other Parameters
        ----------------
        **kwargs
            Additional kwargs are `.Artist` kwargs passed on to `.FigureImage`.

        Notes
        -----
        figimage complements the axes image
        (:meth:`~matplotlib.axes.Axes.imshow`) which will be resampled
        to fit the current axes.  If you want a resampled image to
        fill the entire figure, you can define an
        :class:`~matplotlib.axes.Axes` with extent [0,0,1,1].


        Examples::

            f = plt.figure()
            nx = int(f.get_figwidth() * f.dpi)
            ny = int(f.get_figheight() * f.dpi)
            data = np.random.random((ny, nx))
            f.figimage(data)
            plt.show()

    figlegend(*args, **kwargs)
        Place a legend on the figure.

        To make a legend from existing artists on every axes::

          figlegend()

        To make a legend for a list of lines and labels::

          figlegend(
              (line1, line2, line3),
              ('label1', 'label2', 'label3'),
              loc='upper right')

        These can also be specified by keyword::

          figlegend(
              handles=(line1, line2, line3),
              labels=('label1', 'label2', 'label3'),
              loc='upper right')

        Parameters
        ----------

        handles : sequence of `.Artist`, optional
            A list of Artists (lines, patches) to be added to the legend.
            Use this together with *labels*, if you need full control on what
            is shown in the legend and the automatic mechanism described above
            is not sufficient.

            The length of handles and labels should be the same in this
            case. If they are not, they are truncated to the smaller length.

        labels : sequence of strings, optional
            A list of labels to show next to the artists.
            Use this together with *handles*, if you need full control on what
            is shown in the legend and the automatic mechanism described above
            is not sufficient.

        Other Parameters
        ----------------


        loc : str or pair of floats, default: :rc:`legend.loc` ('best' for axes, 'upper right' for figures)
            The location of the legend.

            The strings
            ``'upper left', 'upper right', 'lower left', 'lower right'``
            place the legend at the corresponding corner of the axes/figure.

            The strings
            ``'upper center', 'lower center', 'center left', 'center right'``
            place the legend at the center of the corresponding edge of the
            axes/figure.

            The string ``'center'`` places the legend at the center of the axes/figure.

            The string ``'best'`` places the legend at the location, among the nine
            locations defined so far, with the minimum overlap with other drawn
            artists.  This option can be quite slow for plots with large amounts of
            data; your plotting speed may benefit from providing a specific location.

            The location can also be a 2-tuple giving the coordinates of the lower-left
            corner of the legend in axes coordinates (in which case *bbox_to_anchor*
            will be ignored).

            For back-compatibility, ``'center right'`` (but no other location) can also
            be spelled ``'right'``, and each "string" locations can also be given as a
            numeric value:

                ===============   =============
                Location String   Location Code
                ===============   =============
                'best'            0
                'upper right'     1
                'upper left'      2
                'lower left'      3
                'lower right'     4
                'right'           5
                'center left'     6
                'center right'    7
                'lower center'    8
                'upper center'    9
                'center'          10
                ===============   =============

        bbox_to_anchor : `.BboxBase`, 2-tuple, or 4-tuple of floats
            Box that is used to position the legend in conjunction with *loc*.
            Defaults to `axes.bbox` (if called as a method to `.Axes.legend`) or
            `figure.bbox` (if `.Figure.legend`).  This argument allows arbitrary
            placement of the legend.

            Bbox coordinates are interpreted in the coordinate system given by
            `bbox_transform`, with the default transform
            Axes or Figure coordinates, depending on which ``legend`` is called.

            If a 4-tuple or `.BboxBase` is given, then it specifies the bbox
            ``(x, y, width, height)`` that the legend is placed in.
            To put the legend in the best location in the bottom right
            quadrant of the axes (or figure)::

                loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)

            A 2-tuple ``(x, y)`` places the corner of the legend specified by *loc* at
            x, y.  For example, to put the legend's upper right-hand corner in the
            center of the axes (or figure) the following keywords can be used::

                loc='upper right', bbox_to_anchor=(0.5, 0.5)

        ncol : integer
            The number of columns that the legend has. Default is 1.

        prop : None or :class:`matplotlib.font_manager.FontProperties` or dict
            The font properties of the legend. If None (default), the current
            :data:`matplotlib.rcParams` will be used.

        fontsize : int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}
            Controls the font size of the legend. If the value is numeric the
            size will be the absolute font size in points. String values are
            relative to the current default font size. This argument is only
            used if `prop` is not specified.

        numpoints : None or int
            The number of marker points in the legend when creating a legend
            entry for a `.Line2D` (line).
            Default is ``None``, which will take the value from
            :rc:`legend.numpoints`.

        scatterpoints : None or int
            The number of marker points in the legend when creating
            a legend entry for a `.PathCollection` (scatter plot).
            Default is ``None``, which will take the value from
            :rc:`legend.scatterpoints`.

        scatteryoffsets : iterable of floats
            The vertical offset (relative to the font size) for the markers
            created for a scatter plot legend entry. 0.0 is at the base the
            legend text, and 1.0 is at the top. To draw all markers at the
            same height, set to ``[0.5]``. Default is ``[0.375, 0.5, 0.3125]``.

        markerscale : None or int or float
            The relative size of legend markers compared with the originally
            drawn ones.
            Default is ``None``, which will take the value from
            :rc:`legend.markerscale`.

        markerfirst : bool
            If *True*, legend marker is placed to the left of the legend label.
            If *False*, legend marker is placed to the right of the legend
            label.
            Default is *True*.

        frameon : None or bool
            Control whether the legend should be drawn on a patch
            (frame).
            Default is ``None``, which will take the value from
            :rc:`legend.frameon`.

        fancybox : None or bool
            Control whether round edges should be enabled around the
            :class:`~matplotlib.patches.FancyBboxPatch` which makes up the
            legend's background.
            Default is ``None``, which will take the value from
            :rc:`legend.fancybox`.

        shadow : None or bool
            Control whether to draw a shadow behind the legend.
            Default is ``None``, which will take the value from
            :rc:`legend.shadow`.

        framealpha : None or float
            Control the alpha transparency of the legend's background.
            Default is ``None``, which will take the value from
            :rc:`legend.framealpha`.  If shadow is activated and
            *framealpha* is ``None``, the default value is ignored.

        facecolor : None or "inherit" or a color spec
            Control the legend's background color.
            Default is ``None``, which will take the value from
            :rc:`legend.facecolor`.  If ``"inherit"``, it will take
            :rc:`axes.facecolor`.

        edgecolor : None or "inherit" or a color spec
            Control the legend's background patch edge color.
            Default is ``None``, which will take the value from
            :rc:`legend.edgecolor` If ``"inherit"``, it will take
            :rc:`axes.edgecolor`.

        mode : {"expand", None}
            If `mode` is set to ``"expand"`` the legend will be horizontally
            expanded to fill the axes area (or `bbox_to_anchor` if defines
            the legend's size).

        bbox_transform : None or :class:`matplotlib.transforms.Transform`
            The transform for the bounding box (`bbox_to_anchor`). For a value
            of ``None`` (default) the Axes'
            :data:`~matplotlib.axes.Axes.transAxes` transform will be used.

        title : str or None
            The legend's title. Default is no title (``None``).

        title_fontsize: str or None
            The fontsize of the legend's title.  Default is the default fontsize.

        borderpad : float or None
            The fractional whitespace inside the legend border.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.borderpad`.

        labelspacing : float or None
            The vertical space between the legend entries.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.labelspacing`.

        handlelength : float or None
            The length of the legend handles.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.handlelength`.

        handletextpad : float or None
            The pad between the legend handle and text.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.handletextpad`.

        borderaxespad : float or None
            The pad between the axes and legend border.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.borderaxespad`.

        columnspacing : float or None
            The spacing between columns.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.columnspacing`.

        handler_map : dict or None
            The custom dictionary mapping instances or types to a legend
            handler. This `handler_map` updates the default handler map
            found at :func:`matplotlib.legend.Legend.get_legend_handler_map`.



        Returns
        -------
        :class:`matplotlib.legend.Legend` instance

        Notes
        -----
        Not all kinds of artist are supported by the legend command. See
        :doc:`/tutorials/intermediate/legend_guide` for details.

    fignum_exists(num)
        Return whether the figure with the given id exists.

    figtext(x, y, s, *args, **kwargs)
        Add text to figure.

        Parameters
        ----------
        x, y : float
            The position to place the text. By default, this is in figure
            coordinates, floats in [0, 1]. The coordinate system can be changed
            using the *transform* keyword.

        s : str
            The text string.

        fontdict : dictionary, optional, default: None
            A dictionary to override the default text properties. If fontdict
            is None, the defaults are determined by your rc parameters. A
            property in *kwargs* override the same property in fontdict.

        withdash : boolean, optional, default: False
            Creates a `~matplotlib.text.TextWithDash` instance instead of a
            `~matplotlib.text.Text` instance.

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.text.Text` properties
            Other miscellaneous text parameters.
              agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a
 (m, n, 3) array
          alpha: float
          animated: bool
          backgroundcolor: color
          bbox: dict with properties for `.patches.FancyBboxPatch`
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          figure: `.Figure`
          fontfamily or family: {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'monospace'}
          fontproperties or font_properties: `.font_manager.FontProperties`
          fontsize or size: {size in points, 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-
large'}
          fontstretch or stretch: {a numeric value in range 0-1000, 'ultra-condensed', 'extra-condensed', 'cond
ensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded', 'ultra-expanded'}
          fontstyle or style: {'normal', 'italic', 'oblique'}
          fontvariant or variant: {'normal', 'small-caps'}
          fontweight or weight: {a numeric value in range 0-1000, 'ultralight', 'light', 'normal', 'regular',
 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'}
          gid: str
          horizontalalignment or ha: {'center', 'right', 'left'}
          in_layout: bool
          label: object
          linespacing: float (multiple of font size)
          multialignment or ma: {'left', 'right', 'center'}
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          position: (float, float)
          rasterized: bool or None
          rotation: {angle in degrees, 'vertical', 'horizontal'}
          rotation_mode: {None, 'default', 'anchor'}
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          text: object
          transform: `.Transform`
          url: str
          usetex: bool or None
          verticalalignment or va: {'center', 'top', 'bottom', 'baseline', 'center_baseline'}
          visible: bool
          wrap: bool
          x: float
          y: float
          zorder: float

        Returns
        -------
        text : `~.text.Text`

        See Also
        --------
        .Axes.text
        .pyplot.text

    figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, frameon=True, FigureClass=<class
 'matplotlib.figure.Figure'>, clear=False, **kwargs)
        Create a new figure.

        Parameters
        ----------
        num : integer or string, optional, default: None
            If not provided, a new figure will be created, and the figure number
            will be incremented. The figure objects holds this number in a `number`
            attribute.
            If num is provided, and a figure with this id already exists, make
            it active, and returns a reference to it. If this figure does not
            exists, create it and returns it.
            If num is a string, the window title will be set to this figure's
            `num`.

        figsize : (float, float), optional, default: None
            width, height in inches. If not provided, defaults to
            :rc:`figure.figsize` = ``[6.4, 4.8]``.

        dpi : integer, optional, default: None
            resolution of the figure. If not provided, defaults to
            :rc:`figure.dpi` = ``100``.

        facecolor : color spec
            the background color. If not provided, defaults to
            :rc:`figure.facecolor` = ``'w'``.

        edgecolor : color spec
            the border color. If not provided, defaults to
            :rc:`figure.edgecolor` = ``'w'``.

        frameon : bool, optional, default: True
            If False, suppress drawing the figure frame.

        FigureClass : subclass of `~matplotlib.figure.Figure`
            Optionally use a custom `.Figure` instance.

        clear : bool, optional, default: False
            If True and the figure already exists, then it is cleared.

        Returns
        -------
        figure : `~matplotlib.figure.Figure`
            The `.Figure` instance returned will also be passed to
            new_figure_manager in the backends, which allows to hook custom
            `.Figure` classes into the pyplot interface. Additional kwargs will be
            passed to the `.Figure` init function.

        Notes
        -----
        If you are creating many figures, make sure you explicitly call
        :func:`.pyplot.close` on the figures you are not using, because this will
        enable pyplot to properly clean up the memory.

        `~matplotlib.rcParams` defines the default values, which can be modified
        in the matplotlibrc file.

    fill(*args, data=None, **kwargs)
```

```
            All bins that has count less than cmin will not be displayed and
            these count values in the return value count histogram will also
            be set to nan upon return

        cmax : scalar, optional, default: None
            All bins that has count more than cmax will not be displayed (set
            to none before passing to imshow) and these count values in the
            return value count histogram will also be set to nan upon return

        Returns
        -------
        h : 2D array
            The bi-dimensional histogram of samples x and y. Values in x are
            histogrammed along the first dimension and values in y are
            histogrammed along the second dimension.
        xedges : 1D array
            The bin edges along the x axis.
        yedges : 1D array
            The bin edges along the y axis.
        image : `~.matplotlib.collections.QuadMesh`

        Other Parameters
        ----------------
        cmap : Colormap or str, optional
            A `.colors.Colormap` instance.  If not set, use rc settings.

        norm : Normalize, optional
            A `.colors.Normalize` instance is used to
            scale luminance data to ``[0, 1]``. If not set, defaults to
            `.colors.Normalize()`.

        vmin/vmax : None or scalar, optional
            Arguments passed to the `~.colors.Normalize` instance.

        alpha : ``0 <= scalar <= 1`` or ``None``, optional
            The alpha blending value.

        See also
        --------
        hist : 1D histogram plotting

        Notes
        -----
        - Currently ``hist2d`` calculates it's own axis limits, and any limits
          previously set are ignored.
        - Rendering the histogram with a logarithmic color scale is
          accomplished by passing a `.colors.LogNorm` instance to the *norm*
          keyword argument. Likewise, power-law normalization (similar
          in effect to gamma correction) can be accomplished with
          `.colors.PowerNorm`.

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'weights', 'x', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    hlines(y, xmin, xmax, colors='k', linestyles='solid', label='', *, data=None, **kwargs)
        Plot horizontal lines at each *y* from *xmin* to *xmax*.

        Parameters
        ----------
        y : scalar or sequence of scalar
            y-indexes where to plot the lines.

        xmin, xmax : scalar or 1D array_like
            Respective beginning and end of each line. If scalars are
            provided, all lines will have same length.

        colors : array_like of colors, optional, default: 'k'

        linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, optional

        label : string, optional, default: ''

        Returns
        -------
        lines : `~matplotlib.collections.LineCollection`

        Other Parameters
        ----------------
        **kwargs :  `~matplotlib.collections.LineCollection` properties.

        See also
        --------
        vlines : vertical lines
        axhline: horizontal line across the axes

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'colors', 'xmax', 'xmin', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    hot()
        Set the colormap to "hot".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    hsv()
        Set the colormap to "hsv".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    imread(fname, format=None)
        Read an image from a file into an array.

        Parameters
        ----------
        fname : str or file-like
            The image file to read. This can be a filename, a URL or a Python
            file-like object opened in read-binary mode.
        format : str, optional
            The image file format assumed for reading the data. If not
            given, the format is deduced from the filename.  If nothing can
            be deduced, PNG is tried.

        Returns
        -------
        imagedata : :class:`numpy.array`
            The image data. The returned array has shape

            - (M, N) for grayscale images.
            - (M, N, 3) for RGB images.
            - (M, N, 4) for RGBA images.

        Notes
        -----
        Matplotlib can only read PNGs natively. Further image formats are
        supported via the optional dependency on Pillow. Note, URL strings
        are not compatible with Pillow. Check the `Pillow documentation`_
        for more information.

        .. _Pillow documentation: http://pillow.readthedocs.io/en/latest/

    imsave(fname, arr, **kwargs)
        Save an array as an image file.

        Parameters
        ----------
        fname : str or PathLike file-like
            A path or a Python file-like object to store the image in.
            If *format* is not set, then the output format is inferred from the
            extension of *fname*, if any, and from :rc:`savefig.format` otherwise.
            If *format* is set, it determines the output format.
        arr : array-like
            The image data. The shape can be one of
            MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).
        vmin, vmax : scalar, optional
            *vmin* and *vmax* set the color scaling for the image by fixing the
            values that map to the colormap color limits. If either *vmin*
            or *vmax* is None, that limit is determined from the *arr*
            min/max value.
        cmap : str or `~matplotlib.colors.Colormap`, optional
            A Colormap instance or registered colormap name. The colormap
            maps scalar data to colors. It is ignored for RGB(A) data.
            Defaults to :rc:`image.cmap` ('viridis').
        format : str, optional
            The file format, e.g. 'png', 'pdf', 'svg', ...  The behavior when this
            is unset is documented under *fname*.
        origin : {'upper', 'lower'}, optional
            Indicates whether the ``(0, 0)`` index of the array is in the upper
            left or lower left corner of the axes.  Defaults to :rc:`image.origin`
            ('upper').
        dpi : int
            The DPI to store in the metadata of the file.  This does not affect the
            resolution of the output image.

    imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=None, vmin=None, vmax=None, origin=N
one, extent=None, shape=<deprecated parameter>, filternorm=1, filterrad=4.0, imlim=<deprecated parameter>, resa
mple=None, url=None, *, data=None, **kwargs)
        Display an image, i.e. data on a 2D regular raster.

        Parameters
        ----------
        X : array-like or PIL image
            The image data. Supported array shapes are:

            - (M, N): an image with scalar data. The data is visualized
              using a colormap.
            - (M, N, 3): an image with RGB values (0-1 float or 0-255 int).
            - (M, N, 4): an image with RGBA values (0-1 float or 0-255 int),
              i.e. including transparency.

            The first two dimensions (M, N) define the rows and columns of
            the image.

            Out-of-range RGB(A) values are clipped.

        cmap : str or `~matplotlib.colors.Colormap`, optional
            The Colormap instance or registered colormap name used to map
            scalar data to colors. This parameter is ignored for RGB(A) data.
            Defaults to :rc:`image.cmap`.

        norm : `~matplotlib.colors.Normalize`, optional
            The `Normalize` instance used to scale scalar data to the [0, 1]
            range before mapping to colors using *cmap*. By default, a linear
            scaling mapping the lowest value to 0 and the highest to 1 is used.
            This parameter is ignored for RGB(A) data.

        aspect : {'equal', 'auto'} or float, optional
            Controls the aspect ratio of the axes. The aspect is of particular
            relevance for images since it may distort the image, i.e. pixel
            will not be square.

            This parameter is a shortcut for explicitly calling
            `.Axes.set_aspect`. See there for further details.

            - 'equal': Ensures an aspect ratio of 1. Pixels will be square
              (unless pixel sizes are explicitly made non-square in data
              coordinates using *extent*).
            - 'auto': The axes is kept fixed and the aspect is adjusted so
              that the data fit in the axes. In general, this will result in
              non-square pixels.

            If not given, use :rc:`image.aspect` (default: 'equal').

        interpolation : str, optional
            The interpolation method used. If *None*
            :rc:`image.interpolation` is used, which defaults to 'nearest'.

            Supported values are 'none', 'nearest', 'bilinear', 'bicubic',
            'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser',
            'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc',
            'lanczos'.

            If *interpolation* is 'none', then no interpolation is performed
            on the Agg, ps, pdf and svg backends. Other backends will fall back
            to 'nearest'. Note that most SVG renders perform interpolation at
            rendering and that the default interpolation method they implement
            may differ.

            See
            :doc:`/gallery/images_contours_and_fields/interpolation_methods`
            for an overview of the supported interpolation methods.

            Some interpolation methods require an additional radius parameter,
            which can be set by *filterrad*. Additionally, the antigrain image
            resize filter is controlled by the parameter *filternorm*.

        alpha : scalar, optional
            The alpha blending value, between 0 (transparent) and 1 (opaque).
            This parameter is ignored for RGBA input data.

        vmin, vmax : scalar, optional
            When using scalar data and no explicit *norm*, *vmin* and *vmax*
            define the data range that the colormap covers. By default,
            the colormap covers the complete value range of the supplied
            data. *vmin*, *vmax* are ignored if the *norm* parameter is used.

        origin : {'upper', 'lower'}, optional
            Place the [0,0] index of the array in the upper left or lower left
            corner of the axes. The convention 'upper' is typically used for
            matrices and images.
            If not given, :rc:`image.origin` is used, defaulting to 'upper'.

            Note that the vertical axes points upward for 'lower'
            but downward for 'upper'.

            See the :doc:`/tutorials/intermediate/imshow_extent` tutorial for
            examples and a more detailed description.

        extent : scalars (left, right, bottom, top), optional
            The bounding box in data coordinates that the image will fill.
            The image is stretched individually along x and y to fill the box.

            The default extent is determined by the following conditions.
            Pixels have unit size in data coordinates. Their centers are on
            integer coordinates, and their center coordinates range from 0 to
            columns-1 horizontally and from 0 to rows-1 vertically.

            Note that the direction of the vertical axis and thus the default
            values for top and bottom depend on *origin*:

            - For ``origin == 'upper'`` the default is
              ``(-0.5, numcols-0.5, numrows-0.5, -0.5)``.
            - For ``origin == 'lower'`` the default is
              ``(-0.5, numcols-0.5, -0.5, numrows-0.5)``.

            See the :doc:`/tutorials/intermediate/imshow_extent` tutorial for
            examples and a more detailed description.

        filternorm : bool, optional, default: True
            A parameter for the antigrain image resize filter (see the
            antigrain documentation).  If *filternorm* is set, the filter
            normalizes integer values and corrects the rounding errors. It
            doesn't do anything with the source floating point values, it
            corrects only integers according to the rule of 1.0 which means
            that any sum of pixel weights must be equal to 1.0.  So, the
            filter function must produce a graph of the proper shape.

        filterrad : float > 0, optional, default: 4.0
            The filter radius for filters that have a radius parameter, i.e.
            when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

        resample : bool, optional
            When *True*, use a full resampling method.  When *False*, only
            resample when the output image is larger than the input image.

        url : str, optional
            Set the url of the created `.AxesImage`. See `.Artist.set_url`.

        Returns
        -------
        image : `~matplotlib.image.AxesImage`

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.artist.Artist` properties
            These parameters are passed on to the constructor of the
            `.AxesImage` artist.

        See also
        --------
        matshow : Plot a matrix or an array as an image.

        Notes
        -----
        Unless *extent* is used, pixel centers will be located at integer
        coordinates. In other words: the origin will coincide with the center
        of pixel (0, 0).

        There are two common representations for RGB images with an alpha
        channel:

        -   Straight (unassociated) alpha: R, G, and B channels represent the
            color of the pixel, disregarding its opacity.
        -   Premultiplied (associated) alpha: R, G, and B channels represent
            the color of the pixel, adjusted for its opacity by multiplication.

        `~matplotlib.pyplot.imshow` expects RGB images adopting the straight
        (unassociated) alpha representation.

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    inferno()
        Set the colormap to "inferno".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    install_repl_displayhook()
        Install a repl display hook so that any stale figure are automatically
        redrawn when control is returned to the repl.

        This works both with IPython and with vanilla python shells.

    ioff()
        Turn the interactive mode off.

    ion()
        Turn the interactive mode on.

    isinteractive()
        Return the status of interactive mode.

    jet()
        Set the colormap to "jet".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    legend(*args, **kwargs)
        Place a legend on the axes.

        Call signatures::

            legend()
            legend(labels)
            legend(handles, labels)

        The call signatures correspond to three different ways how to use
        this method.

        **1. Automatic detection of elements to be shown in the legend**

        The elements to be added to the legend are automatically determined,
        when you do not pass in any extra arguments.

        In this case, the labels are taken from the artist. You can specify
        them either at artist creation or by calling the
        :meth:`~.Artist.set_label` method on the artist::

            line, = ax.plot([1, 2, 3], label='Inline label')
            ax.legend()

        or::

            line, = ax.plot([1, 2, 3])
            line.set_label('Label via method')
            ax.legend()

        Specific lines can be excluded from the automatic legend element
        selection by defining a label starting with an underscore.
        This is default for all artists, so calling `Axes.legend` without
        any arguments and without setting the labels manually will result in
        no legend being drawn.


        **2. Labeling existing plot elements**

        To make a legend for lines which already exist on the axes
        (via plot for instance), simply call this function with an iterable
        of strings, one for each legend item. For example::

            ax.plot([1, 2, 3])
            ax.legend(['A simple line'])

        Note: This way of using is discouraged, because the relation between
        plot elements and labels is only implicit by their order and can
        easily be mixed up.


        **3. Explicitly defining the elements in the legend**

        For full control of which artists have a legend entry, it is possible
        to pass an iterable of legend artists followed by an iterable of
        legend labels respectively::

            legend((line1, line2, line3), ('label1', 'label2', 'label3'))

        Parameters
        ----------

        handles : sequence of `.Artist`, optional
            A list of Artists (lines, patches) to be added to the legend.
            Use this together with *labels*, if you need full control on what
            is shown in the legend and the automatic mechanism described above
            is not sufficient.

            The length of handles and labels should be the same in this
            case. If they are not, they are truncated to the smaller length.

        labels : sequence of strings, optional
            A list of labels to show next to the artists.
            Use this together with *handles*, if you need full control on what
            is shown in the legend and the automatic mechanism described above
            is not sufficient.

        Other Parameters
        ----------------


        loc : str or pair of floats, default: :rc:`legend.loc` ('best' for axes, 'upper right' for figures)
            The location of the legend.

            The strings
            ``'upper left', 'upper right', 'lower left', 'lower right'``
            place the legend at the corresponding corner of the axes/figure.

            The strings
            ``'upper center', 'lower center', 'center left', 'center right'``
            place the legend at the center of the corresponding edge of the
            axes/figure.

            The string ``'center'`` places the legend at the center of the axes/figure.

            The string ``'best'`` places the legend at the location, among the nine
            locations defined so far, with the minimum overlap with other drawn
            artists.  This option can be quite slow for plots with large amounts of
            data; your plotting speed may benefit from providing a specific location.

            The location can also be a 2-tuple giving the coordinates of the lower-left
            corner of the legend in axes coordinates (in which case *bbox_to_anchor*
            will be ignored).

            For back-compatibility, ``'center right'`` (but no other location) can also
            be spelled ``'right'``, and each "string" locations can also be given as a
            numeric value:

                ===============   =============
                Location String   Location Code
                ===============   =============
                'best'            0
                'upper right'     1
                'upper left'      2
                'lower left'      3
                'lower right'     4
                'right'           5
                'center left'     6
                'center right'    7
                'lower center'    8
                'upper center'    9
                'center'          10
                ===============   =============

        bbox_to_anchor : `.BboxBase`, 2-tuple, or 4-tuple of floats
            Box that is used to position the legend in conjunction with *loc*.
            Defaults to `axes.bbox` (if called as a method to `.Axes.legend`) or
            `figure.bbox` (if `.Figure.legend`).  This argument allows arbitrary
            placement of the legend.

            Bbox coordinates are interpreted in the coordinate system given by
            `bbox_transform`, with the default transform
            Axes or Figure coordinates, depending on which ``legend`` is called.

            If a 4-tuple or `.BboxBase` is given, then it specifies the bbox
            ``(x, y, width, height)`` that the legend is placed in.
            To put the legend in the best location in the bottom right
            quadrant of the axes (or figure)::

                loc='best', bbox_to_anchor=(0.5, 0., 0.5, 0.5)

            A 2-tuple ``(x, y)`` places the corner of the legend specified by *loc* at
            x, y.  For example, to put the legend's upper right-hand corner in the
            center of the axes (or figure) the following keywords can be used::

                loc='upper right', bbox_to_anchor=(0.5, 0.5)

        ncol : integer
            The number of columns that the legend has. Default is 1.

        prop : None or :class:`matplotlib.font_manager.FontProperties` or dict
            The font properties of the legend. If None (default), the current
            :data:`matplotlib.rcParams` will be used.

        fontsize : int or float or {'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large'}
            Controls the font size of the legend. If the value is numeric the
            size will be the absolute font size in points. String values are
            relative to the current default font size. This argument is only
            used if `prop` is not specified.

        numpoints : None or int
            The number of marker points in the legend when creating a legend
            entry for a `.Line2D` (line).
            Default is ``None``, which will take the value from
            :rc:`legend.numpoints`.

        scatterpoints : None or int
            The number of marker points in the legend when creating
            a legend entry for a `.PathCollection` (scatter plot).
            Default is ``None``, which will take the value from
            :rc:`legend.scatterpoints`.

        scatteryoffsets : iterable of floats
            The vertical offset (relative to the font size) for the markers
            created for a scatter plot legend entry. 0.0 is at the base the
            legend text, and 1.0 is at the top. To draw all markers at the
            same height, set to ``[0.5]``. Default is ``[0.375, 0.5, 0.3125]``.

        markerscale : None or int or float
            The relative size of legend markers compared with the originally
            drawn ones.
            Default is ``None``, which will take the value from
            :rc:`legend.markerscale`.

        markerfirst : bool
            If *True*, legend marker is placed to the left of the legend label.
            If *False*, legend marker is placed to the right of the legend
            label.
            Default is *True*.

        frameon : None or bool
            Control whether the legend should be drawn on a patch
            (frame).
            Default is ``None``, which will take the value from
            :rc:`legend.frameon`.

        fancybox : None or bool
            Control whether round edges should be enabled around the
            :class:`~matplotlib.patches.FancyBboxPatch` which makes up the
            legend's background.
            Default is ``None``, which will take the value from
            :rc:`legend.fancybox`.

        shadow : None or bool
            Control whether to draw a shadow behind the legend.
            Default is ``None``, which will take the value from
            :rc:`legend.shadow`.

        framealpha : None or float
            Control the alpha transparency of the legend's background.
            Default is ``None``, which will take the value from
            :rc:`legend.framealpha`.  If shadow is activated and
            *framealpha* is ``None``, the default value is ignored.

        facecolor : None or "inherit" or a color spec
            Control the legend's background color.
            Default is ``None``, which will take the value from
            :rc:`legend.facecolor`.  If ``"inherit"``, it will take
            :rc:`axes.facecolor`.

        edgecolor : None or "inherit" or a color spec
            Control the legend's background patch edge color.
            Default is ``None``, which will take the value from
            :rc:`legend.edgecolor` If ``"inherit"``, it will take
            :rc:`axes.edgecolor`.

        mode : {"expand", None}
            If `mode` is set to ``"expand"`` the legend will be horizontally
            expanded to fill the axes area (or `bbox_to_anchor` if defines
            the legend's size).

        bbox_transform : None or :class:`matplotlib.transforms.Transform`
            The transform for the bounding box (`bbox_to_anchor`). For a value
            of ``None`` (default) the Axes'
            :data:`~matplotlib.axes.Axes.transAxes` transform will be used.

        title : str or None
            The legend's title. Default is no title (``None``).

        title_fontsize: str or None
            The fontsize of the legend's title.  Default is the default fontsize.

        borderpad : float or None
            The fractional whitespace inside the legend border.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.borderpad`.

        labelspacing : float or None
            The vertical space between the legend entries.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.labelspacing`.

        handlelength : float or None
            The length of the legend handles.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.handlelength`.

        handletextpad : float or None
            The pad between the legend handle and text.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.handletextpad`.

        borderaxespad : float or None
            The pad between the axes and legend border.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.borderaxespad`.

        columnspacing : float or None
            The spacing between columns.
            Measured in font-size units.
            Default is ``None``, which will take the value from
            :rc:`legend.columnspacing`.

        handler_map : dict or None
            The custom dictionary mapping instances or types to a legend
            handler. This `handler_map` updates the default handler map
            found at :func:`matplotlib.legend.Legend.get_legend_handler_map`.



        Returns
        -------

        :class:`matplotlib.legend.Legend` instance

        Notes
        -----

        Not all kinds of artist are supported by the legend command. See
        :doc:`/tutorials/intermediate/legend_guide` for details.

        Examples
        --------

        .. plot:: gallery/text_labels_and_annotations/legend.py

    locator_params(axis='both', tight=None, **kwargs)
        Control behavior of major tick locators.

        Because the locator is involved in autoscaling, `~.Axes.autoscale_view`
        is called automatically after the parameters are changed.

        Parameters
        ----------
        axis : {'both', 'x', 'y'}, optional
            The axis on which to operate.

        tight : bool or None, optional
            Parameter passed to `~.Axes.autoscale_view`.
            Default is None, for no change.

        Other Parameters
        ----------------
        **kwargs
            Remaining keyword arguments are passed to directly to the
            ``set_params()`` method of the locator. Supported keywords depend
            on the type of the locator. See for example
            `~.ticker.MaxNLocator.set_params` for the `.ticker.MaxNLocator`
            used by default for linear axes.

        Examples
        --------
        When plotting small subplots, one might want to reduce the maximum
        number of ticks and use tight bounds, for example::

            ax.locator_params(tight=True, nbins=4)

    loglog(*args, **kwargs)
        Make a plot with log scaling on both the x and y axis.

        Call signatures::

            loglog([x], y, [fmt], data=None, **kwargs)
            loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

        This is just a thin wrapper around `.plot` which additionally changes
        both the x-axis and the y-axis to log scaling. All of the concepts and
        parameters of plot can be used here as well.

        The additional parameters *basex/y*, *subsx/y* and *nonposx/y* control
        the x/y-axis properties. They are just forwarded to `.Axes.set_xscale`
        and `.Axes.set_yscale`.

        Parameters
        ----------
        basex, basey : scalar, optional, default 10
            Base of the x/y logarithm.

        subsx, subsy : sequence, optional
            The location of the minor x/y ticks. If *None*, reasonable
            locations are automatically chosen depending on the number of
            decades in the plot.
            See `.Axes.set_xscale` / `.Axes.set_yscale` for details.

        nonposx, nonposy : {'mask', 'clip'}, optional, default 'mask'
            Non-positive values in x or y can be masked as invalid, or clipped
            to a very small positive number.

        Returns
        -------
        lines
            A list of `~.Line2D` objects representing the plotted data.

        Other Parameters
        ----------------
        **kwargs
            All parameters supported by `.plot`.

    magma()
        Set the colormap to "magma".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    magnitude_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, scale=None, *, data=None, **k
wargs)
        Plot the magnitude spectrum.

        Compute the magnitude spectrum of *x*.  Data is padded to a
        length of *pad_to* and the windowing function *window* is applied to
        the signal.

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data.

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*.  To create window vectors see
            `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
            `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
            default is `window_hanning`.  If a function is passed as the argument,
            it must take a data segment as an argument and return the windowed
            version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  While not increasing the actual resolution of
            the spectrum (the minimum distance between resolvable peaks),
            this can give more points in the plot, allowing for more
            detail. This corresponds to the *n* parameter in the call to fft().
            The default is None, which sets *pad_to* equal to the length of the
            input signal (i.e. no padding).

        scale : {'default', 'linear', 'dB'}
            The scaling of the values in the *spec*.  'linear' is no scaling.
            'dB' returns the values in dB scale, i.e., the dB amplitude
            (20 * log10). 'default' is 'linear'.

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        Returns
        -------
        spectrum : 1-D array
            The values for the magnitude spectrum before scaling (real valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *spectrum*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.

        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        See Also
        --------
        :func:`psd`
            :func:`psd` plots the power spectral density.`.

        :func:`angle_spectrum`
            :func:`angle_spectrum` plots the angles of the corresponding
            frequencies.

        :func:`phase_spectrum`
            :func:`phase_spectrum` plots the phase (unwrapped angle) of the
            corresponding frequencies.

        :func:`specgram`
            :func:`specgram` can plot the magnitude spectrum of segments within
            the signal in a colormap.

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    margins(*margins, x=None, y=None, tight=True)
        Set or retrieve autoscaling margins.

        The padding added to each limit of the axes is the *margin*
        times the data interval. All input parameters must be floats
        within the range [0, 1]. Passing both positional and keyword
        arguments is invalid and will raise a TypeError. If no
        arguments (positional or otherwise) are provided, the current
        margins will remain in place and simply be returned.

        Specifying any margin changes only the autoscaling; for example,
        if *xmargin* is not None, then *xmargin* times the X data
        interval will be added to each end of that interval before
        it is used in autoscaling.

        Parameters
        ----------
        *margins : float, optional
            If a single positional argument is provided, it specifies
            both margins of the x-axis and y-axis limits. If two
            positional arguments are provided, they will be interpreted
            as *xmargin*, *ymargin*. If setting the margin on a single
            axis is desired, use the keyword arguments described below.

        x, y : float, optional
            Specific margin values for the x-axis and y-axis,
            respectively. These cannot be used with positional
            arguments, but can be used individually to alter on e.g.,
            only the y-axis.

        tight : bool or None, default is True
            The *tight* parameter is passed to :meth:`autoscale_view`,
            which is executed after a margin is changed; the default
            here is *True*, on the assumption that when margins are
            specified, no additional padding to match tick marks is
            usually desired.  Set *tight* to *None* will preserve
            the previous setting.

        Returns
        -------
        xmargin, ymargin : float

        Notes
        -----
        If a previously used Axes method such as :meth:`pcolor` has set
        :attr:`use_sticky_edges` to `True`, only the limits not set by
        the "sticky artists" will be modified. To force all of the
        margins to be set, set :attr:`use_sticky_edges` to `False`
        before calling :meth:`margins`.

    matshow(A, fignum=None, **kwargs)
        Display an array as a matrix in a new figure window.

        The origin is set at the upper left hand corner and rows (first
```

```
        dimension of the array) are displayed horizontally.  The aspect
        ratio of the figure window is that of the array, unless this would
        make an excessively short or narrow figure.

        Tick labels for the xaxis are placed on top.

        Parameters
        ----------
        A : array-like(M, N)
            The matrix to be displayed.

        fignum : None or int or False
            If *None*, create a new figure window with automatic numbering.

            If a nonzero integer, draw into the figure with the given number
            (create it if it does not exist).

            If 0, use the current axes (or create one if it does not exist).

            .. note::

               Because of how `.Axes.matshow` tries to set the figure aspect
               ratio to be the one of the array, strange things may happen if you
               reuse an existing figure.

        Returns
        -------
        image : `~matplotlib.image.AxesImage`

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.axes.Axes.imshow` arguments

    minorticks_off()
        Remove minor ticks from the axes.

    minorticks_on()
        Display minor ticks on the axes.

        Displaying minor ticks may reduce performance; you may turn them off
        using `minorticks_off()` if drawing speed is a problem.

    nipy_spectral()
        Set the colormap to "nipy_spectral".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    pause(interval)
        Pause for *interval* seconds.

        If there is an active figure, it will be updated and displayed before the
        pause, and the GUI event loop (if any) will run during the pause.

        This can be used for crude animation.  For more complex animation, see
        :mod:`matplotlib.animation`.

        Notes
        -----
        This function is experimental; its behavior may be changed or extended in a
        future release.

    pcolor(*args, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, data=None, **kwargs)
        Create a pseudocolor plot with a non-regular rectangular grid.

        Call signature::

            pcolor([X, Y,] C, **kwargs)

        *X* and *Y* can be used to specify the corners of the quadrilaterals.

        .. hint::

            ``pcolor()`` can be very slow for large arrays. In most
            cases you should use the similar but much faster
            `~.Axes.pcolormesh` instead. See there for a discussion of the
            differences.

        Parameters
        ----------
        C : array_like
            A scalar 2-D array. The values will be color-mapped.

        X, Y : array_like, optional
            The coordinates of the quadrilateral corners. The quadrilateral
            for ``C[i,j]`` has corners at::

                (X[i+1, j], Y[i+1, j])           (X[i+1, j+1], Y[i+1, j+1])
                                      +--------+
                                      | C[i,j] |
                                      +--------+
                    (X[i, j], Y[i, j])           (X[i, j+1], Y[i, j+1]),

            Note that the column index corresponds to the
            x-coordinate, and the row index corresponds to y. For
            details, see the :ref:`Notes <axes-pcolor-grid-orientation>`
            section below.

            The dimensions of *X* and *Y* should be one greater than those of
            *C*. Alternatively, *X*, *Y* and *C* may have equal dimensions, in
            which case the last row and column of *C* will be ignored.

            If *X* and/or *Y* are 1-D arrays or column vectors they will be
            expanded as needed into the appropriate 2-D arrays, making a
            rectangular grid.

        cmap : str or `~matplotlib.colors.Colormap`, optional
            A Colormap instance or registered colormap name. The colormap
            maps the *C* values to colors. Defaults to :rc:`image.cmap`.

        norm : `~matplotlib.colors.Normalize`, optional
            The Normalize instance scales the data values to the canonical
            colormap range [0, 1] for mapping to colors. By default, the data
            range is mapped to the colorbar range using linear scaling.

        vmin, vmax : scalar, optional, default: None
            The colorbar range. If *None*, suitable min/max values are
            automatically chosen by the `~.Normalize` instance (defaults to
            the respective min/max values of *C* in case of the default linear
            scaling).

        edgecolors : {'none', None, 'face', color, color sequence}, optional
            The color of the edges. Defaults to 'none'. Possible values:

            - 'none' or '': No edge.
            - *None*: :rc:`patch.edgecolor` will be used. Note that currently
              :rc:`patch.force_edgecolor` has to be True for this to work.
            - 'face': Use the adjacent face color.
            - An mpl color or sequence of colors will set the edge color.

            The singular form *edgecolor* works as an alias.

        alpha : scalar, optional, default: None
            The alpha blending value of the face color, between 0 (transparent)
            and 1 (opaque). Note: The edgecolor is currently not affected by
            this.

        snap : bool, optional, default: False
            Whether to snap the mesh to pixel boundaries.

        Returns
        -------
        collection : `matplotlib.collections.Collection`

        Other Parameters
        ----------------
        antialiaseds : bool, optional, default: False
            The default *antialiaseds* is False if the default
            *edgecolors*\ ="none" is used.  This eliminates artificial lines
            at patch boundaries, and works regardless of the value of alpha.
            If *edgecolors* is not "none", then the default *antialiaseds*
            is taken from :rc:`patch.antialiased`, which defaults to True.
            Stroking the edges may be preferred if *alpha* is 1, but will
            cause artifacts otherwise.

        **kwargs
            Additionally, the following arguments are allowed. They are passed
            along to the `~matplotlib.collections.PolyCollection` constructor:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa or antialiaseds: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods that have ``vmin`` and ``vmax`` kwa
rgs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          cmap: colormap or registered colormap name
          color: color or sequence of rgba tuples
          contains: callable
          edgecolor or ec or edgecolors: color or sequence of colors or 'face'
          facecolor or facecolors or fc: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or dashes or linestyles or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or linewidths or lw: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        See Also
        --------
        pcolormesh : for an explanation of the differences between
            pcolor and pcolormesh.
        imshow : If *X* and *Y* are each equidistant, `~.Axes.imshow` can be a
            faster alternative.

        Notes
        -----

        **Masked arrays**

        *X*, *Y* and *C* may be masked arrays. If either ``C[i, j]``, or one
        of the vertices surrounding ``C[i,j]`` (*X* or *Y* at
        ``[i, j], [i+1, j], [i, j+1], [i+1, j+1]``) is masked, nothing is
        plotted.

        .. _axes-pcolor-grid-orientation:

        **Grid orientation**

        The grid orientation follows the standard matrix convention: An array
        *C* with shape (nrows, ncolumns) is plotted with the column number as
        *X* and the row number as *Y*.

        **Handling of pcolor() end-cases**

        ``pcolor()`` displays all columns of *C* if *X* and *Y* are not
        specified, or if *X* and *Y* have one more column than *C*.
        If *X* and *Y* have the same number of columns as *C* then the last
        column of *C* is dropped. Similarly for the rows.

        Note: This behavior is different from MATLAB's ``pcolor()``, which
        always discards the last row and column of *C*.

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    pcolormesh(*args, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, shading='flat', antialiased=Fals
e, data=None, **kwargs)
        Create a pseudocolor plot with a non-regular rectangular grid.

        Call signature::

            pcolor([X, Y,] C, **kwargs)

        *X* and *Y* can be used to specify the corners of the quadrilaterals.

        .. note::

           ``pcolormesh()`` is similar to :func:`~Axes.pcolor`. It's much
           faster and preferred in most cases. For a detailed discussion on
           the differences see
           :ref:`Differences between pcolor() and pcolormesh()
           <differences-pcolor-pcolormesh>`.

        Parameters
        ----------
        C : array_like
            A scalar 2-D array. The values will be color-mapped.

        X, Y : array_like, optional
            The coordinates of the quadrilateral corners. The quadrilateral
            for ``C[i,j]`` has corners at::

                (X[i+1, j], Y[i+1, j])           (X[i+1, j+1], Y[i+1, j+1])
                                      +--------+
                                      | C[i,j] |
                                      +--------+
                    (X[i, j], Y[i, j])           (X[i, j+1], Y[i, j+1]),

            Note that the column index corresponds to the
            x-coordinate, and the row index corresponds to y. For
            details, see the :ref:`Notes <axes-pcolormesh-grid-orientation>`
            section below.

            The dimensions of *X* and *Y* should be one greater than those of
            *C*. Alternatively, *X*, *Y* and *C* may have equal dimensions, in
            which case the last row and column of *C* will be ignored.

            If *X* and/or *Y* are 1-D arrays or column vectors they will be
            expanded as needed into the appropriate 2-D arrays, making a
            rectangular grid.

        cmap : str or `~matplotlib.colors.Colormap`, optional
            A Colormap instance or registered colormap name. The colormap
            maps the *C* values to colors. Defaults to :rc:`image.cmap`.

        norm : `~matplotlib.colors.Normalize`, optional
            The Normalize instance scales the data values to the canonical
            colormap range [0, 1] for mapping to colors. By default, the data
            range is mapped to the colorbar range using linear scaling.

        vmin, vmax : scalar, optional, default: None
            The colorbar range. If *None*, suitable min/max values are
            automatically chosen by the `~.Normalize` instance (defaults to
            the respective min/max values of *C* in case of the default linear
            scaling).

        edgecolors : {'none', None, 'face', color, color sequence}, optional
            The color of the edges. Defaults to 'none'. Possible values:

            - 'none' or '': No edge.
            - *None*: :rc:`patch.edgecolor` will be used. Note that currently
              :rc:`patch.force_edgecolor` has to be True for this to work.
            - 'face': Use the adjacent face color.
            - An mpl color or sequence of colors will set the edge color.

            The singular form *edgecolor* works as an alias.

        alpha : scalar, optional, default: None
            The alpha blending value, between 0 (transparent) and 1 (opaque).

        shading : {'flat', 'gouraud'}, optional
            The fill style, Possible values:

            - 'flat': A solid color is used for each quad. The color of the
              quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by
              ``C[i,j]``.
            - 'gouraud': Each quad will be Gouraud shaded: The color of the
              corners (i', j') are given by ``C[i',j']``. The color values of
              the area in between is interpolated from the corner values.
              When Gouraud shading is used, *edgecolors* is ignored.

        snap : bool, optional, default: False
            Whether to snap the mesh to pixel boundaries.

        Returns
        -------
        mesh : `matplotlib.collections.QuadMesh`

        Other Parameters
        ----------------
        **kwargs
            Additionally, the following arguments are allowed. They are passed
            along to the `~matplotlib.collections.QuadMesh` constructor:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa or antialiaseds: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods that have ``vmin`` and ``vmax`` kwa
rgs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          cmap: colormap or registered colormap name
          color: color or sequence of rgba tuples
          contains: callable
          edgecolor or ec or edgecolors: color or sequence of colors or 'face'
          facecolor or facecolors or fc: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or dashes or linestyles or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or linewidths or lw: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float


        See Also
        --------
        pcolor : An alternative implementation with slightly different
            features. For a detailed discussion on the differences see
            :ref:`Differences between pcolor() and pcolormesh()
            <differences-pcolor-pcolormesh>`.
        imshow : If *X* and *Y* are each equidistant, `~.Axes.imshow` can be a
            faster alternative.

        Notes
        -----

        **Masked arrays**

        *C* may be a masked array. If ``C[i, j]`` is masked, the corresponding
        quadrilateral will be transparent. Masking of *X* and *Y* is not
        supported. Use `~.Axes.pcolor` if you need this functionality.

        .. _axes-pcolormesh-grid-orientation:

        **Grid orientation**

        The grid orientation follows the standard matrix convention: An array
        *C* with shape (nrows, ncolumns) is plotted with the column number as
        *X* and the row number as *Y*.

        .. _differences-pcolor-pcolormesh:

        **Differences between pcolor() and pcolormesh()**

        Both methods are used to create a pseudocolor plot of a 2-D array
        using quadrilaterals.

        The main difference lies in the created object and internal data
        handling:
        While `~.Axes.pcolor` returns a `.PolyCollection`, `~.Axes.pcolormesh`
        returns a `.QuadMesh`. The latter is more specialized for the given
        purpose and thus is faster. It should almost always be preferred.

        There is also a slight difference in the handling of masked arrays.
        Both `~.Axes.pcolor` and `~.Axes.pcolormesh` support masked arrays
        for *C*. However, only `~.Axes.pcolor` supports masked arrays for *X*
        and *Y*. The reason lies in the internal handling of the masked values.
        `~.Axes.pcolor` leaves out the respective polygons from the
        PolyCollection. `~.Axes.pcolormesh` sets the facecolor of the masked
        elements to transparent. You can see the difference when using
        edgecolors. While all edges are drawn irrespective of masking in a
        QuadMesh, the edge between two adjacent masked quadrilaterals in
        `~.Axes.pcolor` is not drawn as the corresponding polygons do not
        exist in the PolyCollection.

        Another difference is the support of Gouraud shading in
        `~.Axes.pcolormesh`, which is not available with `~.Axes.pcolor`.

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All positional and all keyword arguments.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    phase_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None, *, data=None, **kwargs)
        Plot the phase spectrum.

        Compute the phase spectrum (unwrapped angle spectrum) of *x*.
        Data is padded to a length of *pad_to* and the windowing function
        *window* is applied to the signal.

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*.  To create window vectors see
            `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
            `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
            default is `window_hanning`.  If a function is passed as the argument,
            it must take a data segment as an argument and return the windowed
            version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  While not increasing the actual resolution of
            the spectrum (the minimum distance between resolvable peaks),
            this can give more points in the plot, allowing for more
            detail. This corresponds to the *n* parameter in the call to fft().
            The default is None, which sets *pad_to* equal to the length of the
            input signal (i.e. no padding).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        Returns
        -------
        spectrum : 1-D array
            The values for the phase spectrum in radians (real valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *spectrum*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.

        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        See Also
        --------
        :func:`magnitude_spectrum`
            :func:`magnitude_spectrum` plots the magnitudes of the
            corresponding frequencies.

        :func:`angle_spectrum`
            :func:`angle_spectrum` plots the wrapped version of this function.

        :func:`specgram`
            :func:`specgram` can plot the phase spectrum of segments within the
            signal in a colormap.

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.6, shadow=False, labeldistance=
1.1, startangle=None, radius=None, counterclock=True, wedgeprops=None, textprops=None, center=(0, 0), frame=Fal
se, rotatelabels=False, *, data=None)
        Plot a pie chart.

        Make a pie chart of array *x*.  The fractional area of each wedge is
        given by ``x/sum(x)``.  If ``sum(x) < 1``, then the values of *x* give
        the fractional area directly and the array will not be normalized. The
        resulting pie will have an empty wedge of size ``1 - sum(x)``.

        The wedges are plotted counterclockwise, by default starting from the
        x-axis.

        Parameters
        ----------
        x : array-like
            The wedge sizes.

        explode : array-like, optional, default: None
            If not *None*, is a ``len(x)`` array which specifies the fraction
            of the radius with which to offset each wedge.

        labels : list, optional, default: None
            A sequence of strings providing the labels for each wedge

        colors : array-like, optional, default: None
            A sequence of matplotlib color args through which the pie chart
            will cycle.  If *None*, will use the colors in the currently
            active cycle.

        autopct : None (default), string, or function, optional
            If not *None*, is a string or function used to label the wedges
            with their numeric value.  The label will be placed inside the
            wedge.  If it is a format string, the label will be ``fmt%pct``.
            If it is a function, it will be called.

        pctdistance : float, optional, default: 0.6
            The ratio between the center of each pie slice and the start of
            the text generated by *autopct*.  Ignored if *autopct* is *None*.

        shadow : bool, optional, default: False
            Draw a shadow beneath the pie.

        labeldistance : float or None, optional, default: 1.1
            The radial distance at which the pie labels are drawn.
            If set to ``None``, label are not drawn, but are stored for use in
            ``legend()``

        startangle : float, optional, default: None
            If not *None*, rotates the start of the pie chart by *angle*
            degrees counterclockwise from the x-axis.

        radius : float, optional, default: None
            The radius of the pie, if *radius* is *None* it will be set to 1.

        counterclock : bool, optional, default: True
            Specify fractions direction, clockwise or counterclockwise.

        wedgeprops : dict, optional, default: None
            Dict of arguments passed to the wedge objects making the pie.
            For example, you can pass in ``wedgeprops = {'linewidth': 3}``
            to set the width of the wedge border lines equal to 3.
            For more details, look at the doc/arguments of the wedge object.
            By default ``clip_on=False``.

        textprops : dict, optional, default: None
            Dict of arguments to pass to the text objects.

        center :  list of float, optional, default: (0, 0)
            Center position of the chart. Takes value (0, 0) or is a sequence
            of 2 scalars.

        frame : bool, optional, default: False
            Plot axes frame with the chart if true.

        rotatelabels : bool, optional, default: False
            Rotate each label to the angle of the corresponding slice if true.

        Returns
        -------
        patches : list
            A sequence of :class:`matplotlib.patches.Wedge` instances

        texts : list
            A list of the label :class:`matplotlib.text.Text` instances.

        autotexts : list
            A list of :class:`~matplotlib.text.Text` instances for the numeric
            labels. This will only be returned if the parameter *autopct* is
            not *None*.

        Notes
        -----
        The pie chart will probably look best if the figure and axes are
        square, or the Axes aspect is equal.
        This method sets the aspect ratio of the axis to "equal".
        The axes aspect ratio can be controlled with `Axes.set_aspect`.

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'colors', 'explode', 'labels', 'x'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    pink()
        Set the colormap to "pink".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    plasma()
        Set the colormap to "plasma".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    plot(*args, scalex=True, scaley=True, data=None, **kwargs)
        Plot y versus x as lines and/or markers.

        Call signatures::

            plot([x], y, [fmt], *, data=None, **kwargs)
            plot([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

        The coordinates of the points or line nodes are given by *x*, *y*.

        The optional parameter *fmt* is a convenient way for defining basic
        formatting like color, marker and linestyle. It's a shortcut string
        notation described in the *Notes* section below.

        >>> plot(x, y)        # plot x and y using default line style and color
        >>> plot(x, y, 'bo')  # plot x and y using blue circle markers
        >>> plot(y)           # plot y using x as index array 0..N-1
        >>> plot(y, 'r+')     # ditto, but with red plusses

        You can use `.Line2D` properties as keyword arguments for more
        control on the appearance. Line properties and *fmt* can be mixed.
        The following two calls yield identical results:

        >>> plot(x, y, 'go--', linewidth=2, markersize=12)
        >>> plot(x, y, color='green', marker='o', linestyle='dashed',
        ...      linewidth=2, markersize=12)

        When conflicting with *fmt*, keyword arguments take precedence.


        **Plotting labelled data**

        There's a convenient way for plotting objects with labelled data (i.e.
        data that can be accessed by index ``obj['y']``). Instead of giving
        the data in *x* and *y*, you can provide the object in the *data*
        parameter and just give the labels for *x* and *y*::

        >>> plot('xlabel', 'ylabel', data=obj)

        All indexable objects are supported. This could e.g. be a `dict`, a
        `pandas.DataFame` or a structured numpy array.


        **Plotting multiple sets of data**

        There are various ways to plot multiple sets of data.

        - The most straight forward way is just to call `plot` multiple times.
          Example:

          >>> plot(x1, y1, 'bo')
          >>> plot(x2, y2, 'go')

        - Alternatively, if your data is already a 2d array, you can pass it
          directly to *x*, *y*. A separate data set will be drawn for every
          column.

          Example: an array ``a`` where the first column represents the *x*
          values and the other columns are the *y* columns::

          >>> plot(a[0], a[1:])

        - The third way is to specify multiple sets of *[x]*, *y*, *[fmt]*
          groups::

          >>> plot(x1, y1, 'g^', x2, y2, 'g-')

          In this case, any additional keyword argument applies to all
          datasets. Also this syntax cannot be combined with the *data*
          parameter.

        By default, each line is assigned a different style specified by a
        'style cycle'. The *fmt* and line property parameters are only
        necessary if you want explicit deviations from these defaults.
        Alternatively, you can also change the style cycle using the
        'axes.prop_cycle' rcParam.


        Parameters
        ----------
        x, y : array-like or scalar
            The horizontal / vertical coordinates of the data points.
            *x* values are optional and default to `range(len(y))`.

            Commonly, these parameters are 1D arrays.

            They can also be scalars, or two-dimensional (in that case, the
            columns represent separate data sets).

            These arguments cannot be passed as keywords.

        fmt : str, optional
            A format string, e.g. 'ro' for red circles. See the *Notes*
            section for a full description of the format strings.

            Format strings are just an abbreviation for quickly setting
            basic line properties. All of these and more can also be
            controlled by keyword arguments.

            This argument cannot be passed as keyword.

        data : indexable object, optional
            An object with labelled data. If given, provide the label names to
            plot in *x* and *y*.

            .. note::
                Technically there's a slight ambiguity in calls where the
                second label is a valid *fmt*. `plot('n', 'o', data=obj)`
                could be `plt(x, y)` or `plt(y, fmt)`. In such cases,
                the former interpretation is chosen, but a warning is issued.
                You may suppress the warning by adding an empty format string
                `plot('n', 'o', '', data=obj)`.

        Other Parameters
        ----------------
        scalex, scaley : bool, optional, default: True
            These parameters determined if the view limits are adapted to
            the data limits. The values are passed on to `autoscale_view`.

        **kwargs : `.Line2D` properties, optional
            *kwargs* are used to specify properties like a line label (for
            auto legends), linewidth, antialiasing, marker face color.
            Example::

            >>> plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
            >>> plot([1,2,3], [1,4,9], 'rs',  label='line 2')

            If you make multiple lines with one plot command, the kwargs
            apply to all those lines.

            Here is a list of available `.Line2D` properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        Returns
        -------
        lines
            A list of `.Line2D` objects representing the plotted data.

        See Also
        --------
        scatter : XY scatter plot with markers of varying size and/or color (
            sometimes also called bubble chart).

        Notes
        -----
        **Format Strings**

        A format string consists of a part for color, marker and line::

            fmt = '[marker][line][color]'

        Each of them is optional. If not provided, the value from the style
        cycle is used. Exception: If ``line`` is given, but no ``marker``,
        the data will be a line without markers.

        Other combinations such as ``[color][marker][line]`` are also
        supported, but note that their parsing may be ambiguous.

        **Markers**

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

        **Line Styles**

        =============    ===============================
        character        description
        =============    ===============================
        ``'-'``          solid line style
        ``'--'``         dashed line style
        ``'-.'``         dash-dot line style
        ``':'``          dotted line style
        =============    ===============================

        Example format strings::

            'b'    # blue markers with default shape
            'or'   # red circles
            '-g'   # green solid line
            '--'   # dashed line with default color
            '^k:'  # black triangle_up markers connected by a dotted line

        **Colors**

        The supported color abbreviations are the single letter codes

        =============    ===============================
        character        color
        =============    ===============================
        ``'b'``          blue
        ``'g'``          green
        ``'r'``          red
        ``'c'``          cyan
```

```
        ``'m'``          magenta
        ``'y'``          yellow
        ``'k'``          black
        ``'w'``          white
        =============    ===============================

        and the ``'CN'`` colors that index into the default property cycle.

        If the color is the only part of the format string, you can
        additionally use any  `matplotlib.colors` spec, e.g. full names
        (``'green'``) or hex strings (``'#008000'``).

    plot_date(x, y, fmt='o', tz=None, xdate=True, ydate=False, *, data=None, **kwargs)
        Plot data that contains dates.

        Similar to `.plot`, this plots *y* vs. *x* as lines or markers.
        However, the axis labels are formatted as dates depending on *xdate*
        and *ydate*.

        Parameters
        ----------
        x, y : array-like
            The coordinates of the data points. If *xdate* or *ydate* is
            *True*, the respective values *x* or *y* are interpreted as
            :ref:`Matplotlib dates <date-format>`.

        fmt : str, optional
            The plot format string. For details, see the corresponding
            parameter in `.plot`.

        tz : [ *None* | timezone string | :class:`tzinfo` instance]
            The time zone to use in labeling dates. If *None*, defaults to
            rcParam ``timezone``.

        xdate : bool, optional, default: True
            If *True*, the *x*-axis will be interpreted as Matplotlib dates.

        ydate : bool, optional, default: False
            If *True*, the *y*-axis will be interpreted as Matplotlib dates.


        Returns
        -------
        lines
            A list of `~.Line2D` objects representing the plotted data.


        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float


        See Also
        --------
        matplotlib.dates : Helper functions on dates.
        matplotlib.dates.date2num : Convert dates to num.
        matplotlib.dates.num2date : Convert num to dates.
        matplotlib.dates.drange : Create an equally spaced sequence of dates.


        Notes
        -----
        If you are using custom date tickers and formatters, it may be
        necessary to set the formatters/locators after the call to
        `.plot_date`. `.plot_date` will set the default tick locator to
        `.AutoDateLocator` (if the tick locator is not already set to a
        `.DateLocator` instance) and the default tick formatter to
        `.AutoDateFormatter` (if the tick formatter is not already set to a
        `.DateFormatter` instance).

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    plotfile(fname, cols=(0,), plotfuncs=None, comments='#', skiprows=0, checkrows=5, delimiter=',', names=Non
e, subplots=True, newfig=True, **kwargs)
        Plot the data in a file.

        *cols* is a sequence of column identifiers to plot.  An identifier
        is either an int or a string.  If it is an int, it indicates the
        column number.  If it is a string, it indicates the column header.
        matplotlib will make column headers lower case, replace spaces with
        underscores, and remove all illegal characters; so ``'Adj Close*'``
        will have name ``'adj_close'``.

        - If len(*cols*) == 1, only that column will be plotted on the *y* axis.

        - If len(*cols*) > 1, the first element will be an identifier for
          data for the *x* axis and the remaining elements will be the
          column indexes for multiple subplots if *subplots* is *True*
          (the default), or for lines in a single subplot if *subplots*
          is *False*.

        *plotfuncs*, if not *None*, is a dictionary mapping identifier to
        an :class:`~matplotlib.axes.Axes` plotting function as a string.
        Default is 'plot', other choices are 'semilogy', 'fill', 'bar',
        etc.  You must use the same type of identifier in the *cols*
        vector as you use in the *plotfuncs* dictionary, e.g., integer
        column numbers in both or column names in both. If *subplots*
        is *False*, then including any function such as 'semilogy'
        that changes the axis scaling will set the scaling for all
        columns.

        - *comments*: the character used to indicate the start of a comment
          in the file, or *None* to switch off the removal of comments

        - *skiprows*: is the number of rows from the top to skip

        - *checkrows*: is the number of rows to check to validate the column
          data type.  When set to zero all rows are validated.

        - *delimiter*: is the character(s) separating row items

        - *names*: if not None, is a list of header names.  In this case, no
          header will be read from the file

        If *newfig* is *True*, the plot always will be made in a new figure;
        if *False*, it will be made in the current figure if one exists,
        else in a new figure.

        kwargs are passed on to plotting functions.

        Example usage::

          # plot the 2nd and 4th column against the 1st in two subplots
          plotfile(fname, (0,1,3))

          # plot using column names; specify an alternate plot type for volume
          plotfile(fname, ('date', 'volume', 'adj_close'),
                                        plotfuncs={'volume': 'semilogy'})

        Note: plotfile is intended as a convenience for quickly plotting
        data from flat files; it is not intended as an alternative
        interface to general plotting with pyplot or matplotlib.

    plotting()
        ============================ =========================================================================
==================================================
        Function                     Description

        ============================ =========================================================================
==================================================
        `acorr`                      Plot the autocorrelation of *x*.

        `angle_spectrum`             Plot the angle spectrum.

        `annotate`                   Annotate the point *xy* with text *text*.

        `arrow`                      Add an arrow to the axes.

        `autoscale`                  Autoscale the axis view to the data (toggle).

        `axes`                       Add an axes to the current figure and make it the current axes.

        `axhline`                    Add a horizontal line across the axis.

        `axhspan`                    Add a horizontal span (rectangle) across the axis.

        `axis`                       Convenience method to get or set some axis properties.

        `axvline`                    Add a vertical line across the axes.

        `axvspan`                    Add a vertical span (rectangle) across the axes.

        `bar`                        Make a bar plot.

        `barbs`                      Plot a 2D field of barbs.

        `barh`                       Make a horizontal bar plot.

        `box`                        Turn the axes box on or off on the current axes.

        `boxplot`                    Make a box and whisker plot.

        `broken_barh`                Plot a horizontal sequence of rectangles.

        `cla`                        Clear the current axes.

        `clabel`                     Label a contour plot.

        `clf`                        Clear the current figure.

        `clim`                       Set the color limits of the current image.

        `close`                      Close a figure window.

        `cohere`                     Plot the coherence between *x* and *y*.

        `colorbar`                   Add a colorbar to a plot.

        `contour`                    Plot contours.

        `contourf`                   Plot contours.

        `csd`                        Plot the cross-spectral density.

        `delaxes`                    Remove the `Axes` *ax* (defaulting to the current axes) from its figure.

        `draw`                       Redraw the current figure.

        `errorbar`                   Plot y versus x as lines and/or markers with attached errorbars.

        `eventplot`                  Plot identical parallel lines at the given positions.

        `figimage`                   Add a non-resampled image to the figure.

        `figlegend`                  Place a legend on the figure.

        `fignum_exists`              Return whether the figure with the given id exists.

        `figtext`                    Add text to figure.

        `figure`                     Create a new figure.

        `fill`                       Plot filled polygons.

        `fill_between`               Fill the area between two horizontal curves.

        `fill_betweenx`              Fill the area between two vertical curves.

        `findobj`                    Find artist objects.

        `gca`                        Get the current :class:`~matplotlib.axes.Axes` instance on the current fig
ure matching the given keyword args, or create one.
        `gcf`                        Get the current figure.

        `gci`                        Get the current colorable artist.

        `get_figlabels`              Return a list of existing figure labels.

        `get_fignums`                Return a list of existing figure numbers.

        `grid`                       Configure the grid lines.

        `hexbin`                     Make a hexagonal binning plot.

        `hist`                       Plot a histogram.

        `hist2d`                     Make a 2D histogram plot.

        `hlines`                     Plot horizontal lines at each *y* from *xmin* to *xmax*.

        `imread`                     Read an image from a file into an array.

        `imsave`                     Save an array as an image file.

        `imshow`                     Display an image, i.e.

        `install_repl_displayhook`   Install a repl display hook so that any stale figure are automatically red
rawn when control is returned to the repl.
        `ioff`                       Turn the interactive mode off.

        `ion`                        Turn the interactive mode on.

        `isinteractive`              Return the status of interactive mode.

        `legend`                     Place a legend on the axes.

        `locator_params`             Control behavior of major tick locators.

        `loglog`                     Make a plot with log scaling on both the x and y axis.

        `magnitude_spectrum`         Plot the magnitude spectrum.

        `margins`                    Set or retrieve autoscaling margins.

        `matshow`                    Display an array as a matrix in a new figure window.

        `minorticks_off`             Remove minor ticks from the axes.

        `minorticks_on`              Display minor ticks on the axes.

        `pause`                      Pause for *interval* seconds.

        `pcolor`                     Create a pseudocolor plot with a non-regular rectangular grid.

        `pcolormesh`                 Create a pseudocolor plot with a non-regular rectangular grid.

        `phase_spectrum`             Plot the phase spectrum.

        `pie`                        Plot a pie chart.

        `plot`                       Plot y versus x as lines and/or markers.

        `plot_date`                  Plot data that contains dates.

        `plotfile`                   Plot the data in a file.

        `polar`                      Make a polar plot.

        `psd`                        Plot the power spectral density.

        `quiver`                     Plot a 2D field of arrows.

        `quiverkey`                  Add a key to a quiver plot.

        `rc`                         Set the current rc params.

        `rc_context`                 Return a context manager for managing rc settings.

        `rcdefaults`                 Restore the rc params from Matplotlib's internal default style.

        `rgrids`                     Get or set the radial gridlines on the current polar plot.

        `savefig`                    Save the current figure.

        `sca`                        Set the current Axes instance to *ax*.

        `scatter`                    A scatter plot of *y* vs *x* with varying marker size and/or color.

        `sci`                        Set the current image.

        `semilogx`                   Make a plot with log scaling on the x axis.

        `semilogy`                   Make a plot with log scaling on the y axis.

        `set_cmap`                   Set the default colormap.

        `setp`                       Set a property on an artist object.

        `show`                       Display a figure.

        `specgram`                   Plot a spectrogram.

        `spy`                        Plot the sparsity pattern of a 2D array.

        `stackplot`                  Draw a stacked area plot.

        `stem`                       Create a stem plot.

        `step`                       Make a step plot.

        `streamplot`                 Draw streamlines of a vector flow.

        `subplot`                    Add a subplot to the current figure.

        `subplot2grid`               Create an axis at specific location inside a regular grid.

        `subplot_tool`               Launch a subplot tool window for a figure.

        `subplots`                   Create a figure and a set of subplots.

        `subplots_adjust`            Tune the subplot layout.

        `suptitle`                   Add a centered title to the figure.

        `switch_backend`             Close all open figures and set the Matplotlib backend.

        `table`                      Add a table to an `~.axes.Axes`.

        `text`                       Add text to the axes.

        `thetagrids`                 Get or set the theta gridlines on the current polar plot.

        `tick_params`                Change the appearance of ticks, tick labels, and gridlines.

        `ticklabel_format`           Change the `~matplotlib.ticker.ScalarFormatter` used by default for linear
 axes.
        `tight_layout`               Automatically adjust subplot parameters to give specified padding.

        `title`                      Set a title for the axes.

        `tricontour`                 Draw contours on an unstructured triangular grid.

        `tricontourf`                Draw contours on an unstructured triangular grid.

        `tripcolor`                  Create a pseudocolor plot of an unstructured triangular grid.

        `triplot`                    Draw a unstructured triangular grid as lines and/or markers.

        `twinx`                      Make and return a second axes that shares the *x*-axis.

        `twiny`                      Make and return a second axes that shares the *y*-axis.

        `uninstall_repl_displayhook` Uninstall the matplotlib display hook.

        `violinplot`                 Make a violin plot.

        `vlines`                     Plot vertical lines.

        `xcorr`                      Plot the cross correlation between *x* and *y*.

        `xkcd`                       Turn on `xkcd <https://xkcd.com/>`_ sketch-style drawing mode.

        `xlabel`                     Set the label for the x-axis.

        `xlim`                       Get or set the x limits of the current axes.

        `xscale`                     Set the x-axis scale.

        `xticks`                     Get or set the current tick locations and labels of the x-axis.

        `ylabel`                     Set the label for the y-axis.

        `ylim`                       Get or set the y-limits of the current axes.

        `yscale`                     Set the y-axis scale.

        `yticks`                     Get or set the current tick locations and labels of the y-axis.

        ============================ =========================================================================
==================================================

    polar(*args, **kwargs)
        Make a polar plot.

        call signature::

          polar(theta, r, **kwargs)

        Multiple *theta*, *r* arguments are supported, with format
        strings, as in :func:`~matplotlib.pyplot.plot`.

    prism()
        Set the colormap to "prism".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    psd(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, pad_to=None, sides=None, scal
e_by_freq=None, return_line=None, *, data=None, **kwargs)
        Plot the power spectral density.

        The power spectral density :math:`P_{xx}` by Welch's average
        periodogram method.  The vector *x* is divided into *NFFT* length
        segments.  Each segment is detrended by function *detrend* and
        windowed by function *window*.  *noverlap* gives the length of
        the overlap between segments.  The :math:`|\mathrm{fft}(i)|^2`
        of each segment :math:`i` are averaged to compute :math:`P_{xx}`,
        with a scaling to correct for power loss due to windowing.

        If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*.  To create window vectors see
            `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
            `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
            default is `window_hanning`.  If a function is passed as the argument,
            it must take a data segment as an argument and return the windowed
            version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  This can be different from *NFFT*, which
            specifies the number of data points used.  While not increasing
            the actual resolution of the spectrum (the minimum distance between
            resolvable peaks), this can give more points in the plot,
            allowing for more detail. This corresponds to the *n* parameter
            in the call to fft(). The default is None, which sets *pad_to*
            equal to *NFFT*

        NFFT : int
            The number of data points used in each block for the FFT.
            A power 2 is most efficient.  The default value is 256.
            This should *NOT* be used to get zero padding, or the scaling of the
            result will be incorrect. Use *pad_to* for this instead.

        detrend : {'none', 'mean', 'linear'} or callable, default 'none'
            The function applied to each segment before fft-ing, designed to
            remove the mean or linear trend.  Unlike in MATLAB, where the
            *detrend* parameter is a vector, in Matplotlib is it a function.
            The :mod:`~matplotlib.mlab` module defines `.detrend_none`,
            `.detrend_mean`, and `.detrend_linear`, but you can use a custom
            function as well.  You can also use a string to choose one of the
            functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`.
            'linear' calls `.detrend_linear`.

        scale_by_freq : bool, optional
            Specifies whether the resulting density values should be scaled
            by the scaling frequency, which gives density in units of Hz^-1.
            This allows for integration over the returned frequency values.
            The default is True for MATLAB compatibility.

        noverlap : int
            The number of points of overlap between segments.
            The default value is 0 (no overlap).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        return_line : bool
            Whether to include the line object plotted in the returned values.
            Default is False.

        Returns
        -------
        Pxx : 1-D array
            The values for the power spectrum `P_{xx}` before scaling
            (real valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *Pxx*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.
            Only returned if *return_line* is True.

        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          antialiased or aa: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          color or c: color
          contains: callable
          dash_capstyle: {'butt', 'round', 'projecting'}
          dash_joinstyle: {'miter', 'round', 'bevel'}
          dashes: sequence of floats (on/off ink in points) or (None, None)
          drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
          figure: `.Figure`
          fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
          gid: str
          in_layout: bool
          label: object
          linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or lw: float
          marker: marker style
          markeredgecolor or mec: color
          markeredgewidth or mew: float
          markerfacecolor or mfc: color
          markerfacecoloralt or mfcalt: color
          markersize or ms: float
          markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
          path_effects: `.AbstractPathEffect`
          picker: float or callable[[Artist, Event], Tuple[bool, dict]]
          pickradius: float
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          solid_capstyle: {'butt', 'round', 'projecting'}
          solid_joinstyle: {'miter', 'round', 'bevel'}
          transform: `matplotlib.transforms.Transform`
          url: str
          visible: bool
          xdata: 1D array
          ydata: 1D array
          zorder: float

        See Also
        --------
        :func:`specgram`
            :func:`specgram` differs in the default overlap; in not returning
            the mean of the segment periodograms; in returning the times of the
            segments; and in plotting a colormap instead of a line.

        :func:`magnitude_spectrum`
            :func:`magnitude_spectrum` plots the magnitude spectrum.

        :func:`csd`
            :func:`csd` plots the spectral density between two signals.

        Notes
        -----
        For plotting, the power is plotted as
        :math:`10\log_{10}(P_{xx})` for decibels, though *Pxx* itself
        is returned.

        References
        ----------
        Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
        John Wiley & Sons (1986)

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    quiver(*args, data=None, **kw)
        Plot a 2D field of arrows.

        Call signature::

          quiver([X, Y], U, V, [C], **kw)

        Where *X*, *Y* define the arrow locations, *U*, *V* define the arrow
        directions, and *C* optionally sets the color.

        **Arrow size**

        The default settings auto-scales the length of the arrows to a reasonable size.
        To change this behavior see the *scale* and *scale_units* parameters.

        **Arrow shape**

        The defaults give a slightly swept-back arrow; to make the head a
        triangle, make *headaxislength* the same as *headlength*. To make the
        arrow more pointed, reduce *headwidth* or increase *headlength* and
        *headaxislength*. To make the head smaller relative to the shaft,
        scale down all the head parameters. You will probably do best to leave
        minshaft alone.

        **Arrow outline**

        *linewidths* and *edgecolors* can be used to customize the arrow
        outlines.

        Parameters
        ----------
        X, Y : 1D or 2D array-like, optional
            The x and y coordinates of the arrow locations.

            If not given, they will be generated as a uniform integer meshgrid based
            on the dimensions of *U* and *V*.

            If *X* and *Y* are 1D but *U*, *V* are 2D, *X*, *Y* are expanded to 2D
            using ``X, Y = np.meshgrid(X, Y)``. In this case ``len(X)`` and ``len(Y)``
            must match the column and row dimensions of *U* and *V*.

        U, V : 1D or 2D array-like
            The x and y direction components of the arrow vectors.

        C : 1D or 2D array-like, optional
            Numeric data that defines the arrow colors by colormapping via *norm* and
            *cmap*.

            This does not support explicit colors. If you want to set colors directly,
            use *color* instead.

        units : {'width', 'height', 'dots', 'inches', 'x', 'y' 'xy'}, default: 'width'
            The arrow dimensions (except for *length*) are measured in multiples of
            this unit.

            The following values are supported:

            - 'width', 'height': The width or height of the axis.
            - 'dots', 'inches': Pixels or inches based on the figure dpi.
            - 'x', 'y', 'xy': *X*, *Y* or :math:`\sqrt{X^2 + Y^2}` in data units.

            The arrows scale differently depending on the units.  For
            'x' or 'y', the arrows get larger as one zooms in; for other
            units, the arrow size is independent of the zoom state.  For
            'width or 'height', the arrow size increases with the width and
            height of the axes, respectively, when the window is resized;
            for 'dots' or 'inches', resizing does not change the arrows.

        angles : {'uv', 'xy'} or array-like, optional, default: 'uv'
            Method for determining the angle of the arrows.

            - 'uv': The arrow axis aspect ratio is 1 so that
              if *U* == *V* the orientation of the arrow on the plot is 45 degrees
              counter-clockwise from the horizontal axis (positive to the right).

              Use this if the arrows symbolize a quantity that is not based on
              *X*, *Y* data coordinates.

            - 'xy': Arrows point from (x,y) to (x+u, y+v).
              Use this for plotting a gradient field, for example.

            - Alternatively, arbitrary angles may be specified explicitly as an array
              of values in degrees, counter-clockwise from the horizontal axis.

              In this case *U*, *V* is only used to determine the length of the
              arrows.

            Note: inverting a data axis will correspondingly invert the
            arrows only with ``angles='xy'``.

        scale : float, optional
            Number of data units per arrow length unit, e.g., m/s per plot width; a
            smaller scale parameter makes the arrow longer. Default is *None*.

            If *None*, a simple autoscaling algorithm is used, based on the average
            vector length and the number of vectors. The arrow length unit is given by
            the *scale_units* parameter.

        scale_units : {'width', 'height', 'dots', 'inches', 'x', 'y', 'xy'}, optional
            If the *scale* kwarg is *None*, the arrow length unit. Default is *None*.

            e.g. *scale_units* is 'inches', *scale* is 2.0, and
            ``(u,v) = (1,0)``, then the vector will be 0.5 inches long.

            If *scale_units* is 'width' or 'height', then the vector will be half the
            width/height of the axes.

            If *scale_units* is 'x' then the vector will be 0.5 x-axis
            units. To plot vectors in the x-y plane, with u and v having
            the same units as x and y, use
            ``angles='xy', scale_units='xy', scale=1``.

        width : float, optional
            Shaft width in arrow units; default depends on choice of units,
            above, and number of vectors; a typical starting value is about
            0.005 times the width of the plot.

        headwidth : float, optional, default: 3
            Head width as multiple of shaft width.

        headlength : float, optional, default: 5
            Head length as multiple of shaft width.

        headaxislength : float, optional, default: 4.5
            Head length at shaft intersection.

        minshaft : float, optional, default: 1
            Length below which arrow scales, in units of head length. Do not
            set this to less than 1, or small arrows will look terrible!

        minlength : float, optional, default: 1
            Minimum length as a multiple of shaft width; if an arrow length
            is less than this, plot a dot (hexagon) of this diameter instead.

        pivot : {'tail', 'mid', 'middle', 'tip'}, optional, default: 'tail'
            The part of the arrow that is anchored to the *X*, *Y* grid. The arrow
            rotates about this point.

            'mid' is a synonym for 'middle'.

        color : color or color sequence, optional
            Explicit color(s) for the arrows. If *C* has been set, *color* has no
            effect.

            This is a synonym for the `~.PolyCollection` *facecolor* parameter.

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.collections.PolyCollection` properties, optional
            All other keyword arguments are passed on to `.PolyCollection`:

              agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a
 (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased or aa or antialiaseds: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods that have ``vmin`` and ``vmax`` kwa
rgs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          cmap: colormap or registered colormap name
          color: color or sequence of rgba tuples
          contains: callable
          edgecolor or ec or edgecolors: color or sequence of colors or 'face'
          facecolor or facecolors or fc: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle or dashes or linestyles or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth or linewidths or lw: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        See Also
        --------
        quiverkey : Add a key to a quiver plot.

    quiverkey(Q, X, Y, U, label, **kw)
        Add a key to a quiver plot.

        Call signature::

          quiverkey(Q, X, Y, U, label, **kw)

        Arguments:

          *Q*:
            The Quiver instance returned by a call to quiver.

          *X*, *Y*:
            The location of the key; additional explanation follows.

          *U*:
            The length of the key

          *label*:
            A string with the length and units of the key

        Keyword arguments:

          *angle* = 0
            The angle of the key arrow. Measured in degrees anti-clockwise from the
            x-axis.

          *coordinates* = [ 'axes' | 'figure' | 'data' | 'inches' ]
            Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are
            normalized coordinate systems with 0,0 in the lower left and 1,1
            in the upper right; 'data' are the axes data coordinates (used for
            the locations of the vectors in the quiver plot itself); 'inches'
            is position in the figure in inches, with 0,0 at the lower left
            corner.

          *color*:
            overrides face and edge colors from *Q*.

          *labelpos* = [ 'N' | 'S' | 'E' | 'W' ]
            Position the label above, below, to the right, to the left of the
            arrow, respectively.

          *labelsep*:
            Distance in inches between the arrow and the label.  Default is
            0.1

          *labelcolor*:
            defaults to default :class:`~matplotlib.text.Text` color.

          *fontproperties*:
            A dictionary with keyword arguments accepted by the
            :class:`~matplotlib.font_manager.FontProperties` initializer:
            *family*, *style*, *variant*, *size*, *weight*

        Any additional keyword arguments are used to override vector
        properties taken from *Q*.

        The positioning of the key depends on *X*, *Y*, *coordinates*, and
        *labelpos*.  If *labelpos* is 'N' or 'S', *X*, *Y* give the position
        of the middle of the key arrow.  If *labelpos* is 'E', *X*, *Y*
        positions the head, and if *labelpos* is 'W', *X*, *Y* positions the
        tail; in either of these two cases, *X*, *Y* is somewhere in the
        middle of the arrow+label key object.

    rc(group, **kwargs)
        Set the current rc params.  *group* is the grouping for the rc, e.g.,
        for ``lines.linewidth`` the group is ``lines``, for
        ``axes.facecolor``, the group is ``axes``, and so on.  Group may
        also be a list or tuple of group names, e.g., (*xtick*, *ytick*).
        *kwargs* is a dictionary attribute name/value pairs, e.g.,::

          rc('lines', linewidth=2, color='r')

        sets the current rc params and is equivalent to::

          rcParams['lines.linewidth'] = 2
          rcParams['lines.color'] = 'r'

        The following aliases are available to save typing for interactive
        users:

        =====   =================
        Alias   Property
        =====   =================
        'lw'    'linewidth'
        'ls'    'linestyle'
        'c'     'color'
        'fc'    'facecolor'
        'ec'    'edgecolor'
        'mew'   'markeredgewidth'
        'aa'    'antialiased'
        =====   =================

        Thus you could abbreviate the above rc command as::

              rc('lines', lw=2, c='r')


        Note you can use python's kwargs dictionary facility to store
        dictionaries of default parameters.  e.g., you can customize the
        font rc as follows::

          font = {'family' : 'monospace',
                  'weight' : 'bold',
                  'size'   : 'larger'}

          rc('font', **font)  # pass in the font dict as kwargs

        This enables you to easily switch between several configurations.  Use
        ``matplotlib.style.use('default')`` or :func:`~matplotlib.rcdefaults` to
        restore the default rc params after changes.

    rc_context(rc=None, fname=None)
        Return a context manager for managing rc settings.
```

```
    This allows one to do::

        with mpl.rc_context(fname='screen.rc'):
            plt.plot(x, a)
            with mpl.rc_context(fname='print.rc'):
                plt.plot(x, b)
            plt.plot(x, c)

    The 'a' vs 'x' and 'c' vs 'x' plots would have settings from
    'screen.rc', while the 'b' vs 'x' plot would have settings from
    'print.rc'.

    A dictionary can also be passed to the context manager::

        with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
            plt.plot(x, a)

    The 'rc' dictionary takes precedence over the settings loaded from
    'fname'.  Passing a dictionary only is also valid. For example a
    common usage is::

        with mpl.rc_context(rc={'interactive': False}):
            fig, ax = plt.subplots()
            ax.plot(range(3), range(3))
            fig.savefig('A.png', format='png')
            plt.close(fig)

rcdefaults()
    Restore the rc params from Matplotlib's internal default style.

    Style-blacklisted rc params (defined in
    `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

    See Also
    --------
    rc_file_defaults
        Restore the rc params from the rc file originally loaded by Matplotlib.
    matplotlib.style.use :
        Use a specific style file.  Call ``style.use('default')`` to restore
        the default style.

rgrids(*args, **kwargs)
    Get or set the radial gridlines on the current polar plot.

    Call signatures::

     lines, labels = rgrids()
     lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None, **kwargs)

    When called with no arguments, `.rgrids` simply returns the tuple
    (*lines*, *labels*). When called with arguments, the labels will
    appear at the specified radial distances and angle.

    Parameters
    ----------
    radii : tuple with floats
        The radii for the radial gridlines

    labels : tuple with strings or None
        The labels to use at each radial gridline. The
        `matplotlib.ticker.ScalarFormatter` will be used if None.

    angle : float
        The angular position of the radius labels in degrees.

    fmt : str or None
        Format string used in `matplotlib.ticker.FormatStrFormatter`.
        For example '%f'.

    Returns
    -------
    lines, labels : list of `.lines.Line2D`, list of `.text.Text`
        *lines* are the radial gridlines and *labels* are the tick labels.

    Other Parameters
    ----------------
    **kwargs
        *kwargs* are optional `~.Text` properties for the labels.

    Examples
    --------
    ::

      # set the locations of the radial gridlines
      lines, labels = rgrids( (0.25, 0.5, 1.0) )

      # set the locations and labels of the radial gridlines
      lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' ))

    See Also
    --------
    .pyplot.thetagrids
    .projections.polar.PolarAxes.set_rgrids
    .Axis.get_gridlines
    .Axis.get_ticklabels

savefig(*args, **kwargs)
    Save the current figure.

    Call signature::

      savefig(fname, dpi=None, facecolor='w', edgecolor='w',
              orientation='portrait', papertype=None, format=None,
              transparent=False, bbox_inches=None, pad_inches=0.1,
              frameon=None, metadata=None)

    The output formats available depend on the backend being used.

    Parameters
    ----------

    fname : str or PathLike or file-like object
        A path, or a Python file-like object, or
        possibly some backend-dependent object such as
        `matplotlib.backends.backend_pdf.PdfPages`.

        If *format* is not set, then the output format is inferred from
        the extension of *fname*, if any, and from :rc:`savefig.format`
        otherwise.  If *format* is set, it determines the output format.

        Hence, if *fname* is not a path or has no extension, remember to
        specify *format* to ensure that the correct backend is used.

    Other Parameters
    ----------------

    dpi : [ *None* | scalar > 0 | 'figure' ]
        The resolution in dots per inch.  If *None*, defaults to
        :rc:`savefig.dpi`.  If 'figure', uses the figure's dpi value.

    quality : [ *None* | 1 <= scalar <= 100 ]
        The image quality, on a scale from 1 (worst) to 95 (best).
        Applicable only if *format* is jpg or jpeg, ignored otherwise.
        If *None*, defaults to :rc:`savefig.jpeg_quality` (95 by default).
        Values above 95 should be avoided; 100 completely disables the
        JPEG quantization stage.

    optimize : bool
        If *True*, indicates that the JPEG encoder should make an extra
        pass over the image in order to select optimal encoder settings.
        Applicable only if *format* is jpg or jpeg, ignored otherwise.
        Is *False* by default.

    progressive : bool
        If *True*, indicates that this image should be stored as a
        progressive JPEG file. Applicable only if *format* is jpg or
        jpeg, ignored otherwise. Is *False* by default.

    facecolor : color spec or None, optional
        The facecolor of the figure; if *None*, defaults to
        :rc:`savefig.facecolor`.

    edgecolor : color spec or None, optional
        The edgecolor of the figure; if *None*, defaults to
        :rc:`savefig.edgecolor`

    orientation : {'landscape', 'portrait'}
        Currently only supported by the postscript backend.

    papertype : str
        One of 'letter', 'legal', 'executive', 'ledger', 'a0' through
        'a10', 'b0' through 'b10'. Only supported for postscript
        output.

    format : str
        The file format, e.g. 'png', 'pdf', 'svg', ... The behavior when
        this is unset is documented under *fname*.

    transparent : bool
        If *True*, the axes patches will all be transparent; the
        figure patch will also be transparent unless facecolor
        and/or edgecolor are specified via kwargs.
        This is useful, for example, for displaying
        a plot on top of a colored background on a web page.  The
        transparency of these patches will be restored to their
        original values upon exit of this function.

    bbox_inches : str or `~matplotlib.transforms.Bbox`, optional
        Bbox in inches. Only the given portion of the figure is
        saved. If 'tight', try to figure out the tight bbox of
        the figure. If None, use savefig.bbox

    pad_inches : scalar, optional
        Amount of padding around the figure when bbox_inches is
        'tight'. If None, use savefig.pad_inches

    bbox_extra_artists : list of `~matplotlib.artist.Artist`, optional
        A list of extra artists that will be considered when the
        tight bbox is calculated.

    metadata : dict, optional
        Key/value pairs to store in the image metadata. The supported keys
        and defaults depend on the image format and backend:

        - 'png' with Agg backend: See the parameter ``metadata`` of
          `~.FigureCanvasAgg.print_png`.
        - 'pdf' with pdf backend: See the parameter ``metadata`` of
          `~.backend_pdf.PdfPages`.
        - 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

    pil_kwargs : dict, optional
        Additional keyword arguments that are passed to `PIL.Image.save`
        when saving the figure.  Only applicable for formats that are saved
        using Pillow, i.e. JPEG, TIFF, and (if the keyword is set to a
        non-None value) PNG.

sca(ax)
    Set the current Axes instance to *ax*.

    The current Figure is updated to the parent of *ax*.

scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidt
hs=None, verts=None, edgecolors=None, *, plotnonfinite=False, data=None, **kwargs)
    A scatter plot of *y* vs *x* with varying marker size and/or color.

    Parameters
    ----------
    x, y : array_like, shape (n, )
        The data positions.

    s : scalar or array_like, shape (n, ), optional
        The marker size in points**2.
        Default is ``rcParams['lines.markersize'] ** 2``.

    c : color, sequence, or sequence of color, optional
        The marker color. Possible values:

        - A single color format string.
        - A sequence of color specifications of length n.
        - A sequence of n numbers to be mapped to colors using *cmap* and
          *norm*.
        - A 2-D array in which the rows are RGB or RGBA.

        Note that *c* should not be a single numeric RGB or RGBA sequence
        because that is indistinguishable from an array of values to be
        colormapped. If you want to specify the same RGB or RGBA value for
        all points, use a 2-D array with a single row.  Otherwise, value-
        matching will have precedence in case of a size matching with *x*
        and *y*.

        Defaults to ``None``. In that case the marker color is determined
        by the value of ``color``, ``facecolor`` or ``facecolors``. In case
        those are not specified or ``None``, the marker color is determined
        by the next color of the ``Axes``' current "shape and fill" color
        cycle. This cycle defaults to :rc:`axes.prop_cycle`.

    marker : `~matplotlib.markers.MarkerStyle`, optional
        The marker style. *marker* can be either an instance of the class
        or the text shorthand for a particular marker.
        Defaults to ``None``, in which case it takes the value of
        :rc:`scatter.marker` = 'o'.
        See `~matplotlib.markers` for more information about marker styles.

    cmap : `~matplotlib.colors.Colormap`, optional, default: None
        A `.Colormap` instance or registered colormap name. *cmap* is only
        used if *c* is an array of floats. If ``None``, defaults to rc
        ``image.cmap``.

    norm : `~matplotlib.colors.Normalize`, optional, default: None
        A `.Normalize` instance is used to scale luminance data to 0, 1.
        *norm* is only used if *c* is an array of floats. If *None*, use
        the default `.colors.Normalize`.

    vmin, vmax : scalar, optional, default: None
        *vmin* and *vmax* are used in conjunction with *norm* to normalize
        luminance data. If None, the respective min and max of the color
        array is used. *vmin* and *vmax* are ignored if you pass a *norm*
        instance.

    alpha : scalar, optional, default: None
        The alpha blending value, between 0 (transparent) and 1 (opaque).

    linewidths : scalar or array_like, optional, default: None
        The linewidth of the marker edges. Note: The default *edgecolors*
        is 'face'. You may want to change this as well.
        If *None*, defaults to rcParams ``lines.linewidth``.

    edgecolors : {'face', 'none', *None*} or color or sequence of color, optional.
        The edge color of the marker. Possible values:

        - 'face': The edge color will always be the same as the face color.
        - 'none': No patch boundary will be drawn.
        - A Matplotlib color or sequence of color.

        Defaults to ``None``, in which case it takes the value of
        :rc:`scatter.edgecolors` = 'face'.

        For non-filled markers, the *edgecolors* kwarg is ignored and
        forced to 'face' internally.

    plotnonfinite : boolean, optional, default: False
        Set to plot points with nonfinite *c*, in conjunction with
        `~matplotlib.colors.Colormap.set_bad`.

    Returns
    -------
    paths : `~matplotlib.collections.PathCollection`

    Other Parameters
    ----------------
    **kwargs : `~matplotlib.collections.Collection` properties

    See Also
    --------
    plot : To plot scatter plots when markers are identical in size and
        color.

    Notes
    -----

    * The `.plot` function will be faster for scatterplots where markers
      don't vary in size or color.

    * Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which
      case all masks will be combined and only unmasked points will be
      plotted.

    * Fundamentally, scatter works with 1-D arrays; *x*, *y*, *s*, and *c*
      may be input as 2-D arrays, but within scatter they will be
      flattened. The exception is *c*, which will be flattened only if its
      size matches the size of *x* and *y*.

    .. note::
        In addition to the above described arguments, this function can take a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All arguments with the following names: 'c', 'color', 'edgecolors', 'facecolor', 'facecolors', 'l
inewidths', 's', 'x', 'y'.

        Objects passed as **data** must support item access (``data[<arg>]``) and
        membership test (``<arg> in data``).

sci(im)
    Set the current image.

    This image will be the target of colormap functions like
    `~.pyplot.viridis`, and other functions such as `~.pyplot.clim`.  The
    current image is an attribute of the current axes.

semilogx(*args, **kwargs)
    Make a plot with log scaling on the x axis.

    Call signatures::

        semilogx([x], y, [fmt], data=None, **kwargs)
        semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    This is just a thin wrapper around `.plot` which additionally changes
    the x-axis to log scaling. All of the concepts and parameters of plot
    can be used here as well.

    The additional parameters *basex*, *subsx* and *nonposx* control the
    x-axis properties. They are just forwarded to `.Axes.set_xscale`.

    Parameters
    ----------
    basex : scalar, optional, default 10
        Base of the x logarithm.

    subsx : array_like, optional
        The location of the minor xticks. If *None*, reasonable locations
        are automatically chosen depending on the number of decades in the
        plot. See `.Axes.set_xscale` for details.

    nonposx : {'mask', 'clip'}, optional, default 'mask'
        Non-positive values in x can be masked as invalid, or clipped to a
        very small positive number.

    Returns
    -------
    lines
        A list of `~.Line2D` objects representing the plotted data.

    Other Parameters
    ----------------
    **kwargs
        All parameters supported by `.plot`.

semilogy(*args, **kwargs)
    Make a plot with log scaling on the y axis.

    Call signatures::

        semilogy([x], y, [fmt], data=None, **kwargs)
        semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    This is just a thin wrapper around `.plot` which additionally changes
    the y-axis to log scaling. All of the concepts and parameters of plot
    can be used here as well.

    The additional parameters *basey*, *subsy* and *nonposy* control the
    y-axis properties. They are just forwarded to `.Axes.set_yscale`.

    Parameters
    ----------
    basey : scalar, optional, default 10
        Base of the y logarithm.

    subsy : array_like, optional
        The location of the minor yticks. If *None*, reasonable locations
        are automatically chosen depending on the number of decades in the
        plot. See `.Axes.set_yscale` for details.

    nonposy : {'mask', 'clip'}, optional, default 'mask'
        Non-positive values in y can be masked as invalid, or clipped to a
        very small positive number.

    Returns
    -------
    lines
        A list of `~.Line2D` objects representing the plotted data.

    Other Parameters
    ----------------
    **kwargs
        All parameters supported by `.plot`.

set_cmap(cmap)
    Set the default colormap.  Applies to the current image if any.
    See help(colormaps) for more information.

    *cmap* must be a :class:`~matplotlib.colors.Colormap` instance, or
    the name of a registered colormap.

    See :func:`matplotlib.cm.register_cmap` and
    :func:`matplotlib.cm.get_cmap`.

setp(obj, *args, **kwargs)
    Set a property on an artist object.

    matplotlib supports the use of :func:`setp` ("set property") and
    :func:`getp` to set and get object properties, as well as to do
    introspection on the object.  For example, to set the linestyle of a
    line to be dashed, you can do::

      >>> line, = plot([1,2,3])
      >>> setp(line, linestyle='--')

    If you want to know the valid types of arguments, you can provide
    the name of the property you want to set without a value::

      >>> setp(line, 'linestyle')
          linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]

    If you want to see all the properties that can be set, and their
    possible values, you can do::

      >>> setp(line)
          ... long output listing omitted

    You may specify another output file to `setp` if `sys.stdout` is not
    acceptable for some reason using the `file` keyword-only argument::

      >>> with fopen('output.log') as f:
      >>>     setp(line, file=f)

    :func:`setp` operates on a single instance or a iterable of
    instances. If you are in query mode introspecting the possible
    values, only the first instance in the sequence is used. When
    actually setting values, all the instances will be set.  e.g.,
    suppose you have a list of two lines, the following will make both
    lines thicker and red::

      >>> x = arange(0,1.0,0.01)
      >>> y1 = sin(2*pi*x)
      >>> y2 = sin(4*pi*x)
      >>> lines = plot(x, y1, x, y2)
      >>> setp(lines, linewidth=2, color='r')

    :func:`setp` works with the MATLAB style string/value pairs or
    with python kwargs.  For example, the following are equivalent::

      >>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
      >>> setp(lines, linewidth=2, color='r')        # python style

show(*args, **kw)
    Display a figure.

    When running in ipython with its pylab mode, display all
    figures and return to the ipython prompt.

    In non-interactive mode, display all figures and block until
    the figures have been closed; in interactive mode it has no
    effect unless figures were created prior to a change from
    non-interactive to interactive mode (not recommended).  In
    that case it displays the figures but does not block.

    A single experimental keyword argument, *block*, may be
    set to True or False to override the blocking behavior
    described above.

specgram(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=None, cmap=None, xextent=None,
 pad_to=None, sides=None, scale_by_freq=None, mode=None, scale=None, vmin=None, vmax=None, *, data=None, **kwar
gs)
    Plot a spectrogram.

    Compute and plot a spectrogram of data in *x*.  Data are split into
    *NFFT* length segments and the spectrum of each section is
    computed.  The windowing function *window* is applied to each
    segment, and the amount of overlap of each segment is
    specified with *noverlap*. The spectrogram is plotted as a colormap
    (using imshow).

    Parameters
    ----------
    x : 1-D array or sequence
        Array or sequence containing the data.

    Fs : scalar
        The sampling frequency (samples per time unit).  It is used
        to calculate the Fourier frequencies, freqs, in cycles per time
        unit. The default value is 2.

    window : callable or ndarray
        A function or a vector of length *NFFT*.  To create window vectors see
        `window_hanning`, `window_none`, `numpy.blackman`, `numpy.hamming`,
        `numpy.bartlett`, `scipy.signal`, `scipy.signal.get_window`, etc.  The
        default is `window_hanning`.  If a function is passed as the argument,
        it must take a data segment as an argument and return the windowed
        version of the segment.

    sides : {'default', 'onesided', 'twosided'}
        Specifies which sides of the spectrum to return.  Default gives the
        default behavior, which returns one-sided for real data and both
        for complex data.  'onesided' forces the return of a one-sided
        spectrum, while 'twosided' forces two-sided.

    pad_to : int
        The number of points to which the data segment is padded when
        performing the FFT.  This can be different from *NFFT*, which
        specifies the number of data points used.  While not increasing
        the actual resolution of the spectrum (the minimum distance between
        resolvable peaks), this can give more points in the plot,
        allowing for more detail. This corresponds to the *n* parameter
        in the call to fft(). The default is None, which sets *pad_to*
        equal to *NFFT*

    NFFT : int
        The number of data points used in each block for the FFT.
        A power 2 is most efficient.  The default value is 256.
        This should *NOT* be used to get zero padding, or the scaling of the
        result will be incorrect. Use *pad_to* for this instead.

    detrend : {'none', 'mean', 'linear'} or callable, default 'none'
        The function applied to each segment before fft-ing, designed to
        remove the mean or linear trend.  Unlike in MATLAB, where the
        *detrend* parameter is a vector, in Matplotlib is it a function.
        The :mod:`~matplotlib.mlab` module defines `.detrend_none`,
        `.detrend_mean`, and `.detrend_linear`, but you can use a custom
        function as well.  You can also use a string to choose one of the
        functions: 'none' calls `.detrend_none`. 'mean' calls `.detrend_mean`.
        'linear' calls `.detrend_linear`.

    scale_by_freq : bool, optional
        Specifies whether the resulting density values should be scaled
        by the scaling frequency, which gives density in units of Hz^-1.
        This allows for integration over the returned frequency values.
        The default is True for MATLAB compatibility.

    mode : {'default', 'psd', 'magnitude', 'angle', 'phase'}
        What sort of spectrum to use.  Default is 'psd', which takes the
        power spectral density.  'magnitude' returns the magnitude
        spectrum.  'angle' returns the phase spectrum without unwrapping.
        'phase' returns the phase spectrum with unwrapping.

    noverlap : int
        The number of points of overlap between blocks.  The
        default value is 128.

    scale : {'default', 'linear', 'dB'}
        The scaling of the values in the *spec*.  'linear' is no scaling.
        'dB' returns the values in dB scale.  When *mode* is 'psd',
        this is dB power (10 * log10).  Otherwise this is dB amplitude
        (20 * log10). 'default' is 'dB' if *mode* is 'psd' or
        'magnitude' and 'linear' otherwise.  This must be 'linear'
        if *mode* is 'angle' or 'phase'.

    Fc : int
        The center frequency of *x* (defaults to 0), which offsets
        the x extents of the plot to reflect the frequency range used
        when a signal is acquired and then filtered and downsampled to
        baseband.

    cmap
        A :class:`matplotlib.colors.Colormap` instance; if *None*, use
        default determined by rc

    xextent : *None* or (xmin, xmax)
        The image extent along the x-axis. The default sets *xmin* to the
        left border of the first bin (*spectrum* column) and *xmax* to the
        right border of the last bin. Note that for *noverlap>0* the width
        of the bins is smaller than those of the segments.

    **kwargs
        Additional kwargs are passed on to imshow which makes the
        specgram image.

    Returns
    -------
    spectrum : 2-D array
        Columns are the periodograms of successive segments.

    freqs : 1-D array
        The frequencies corresponding to the rows in *spectrum*.

    t : 1-D array
        The times corresponding to midpoints of segments (i.e., the columns
        in *spectrum*).

    im : instance of class :class:`~matplotlib.image.AxesImage`
        The image created by imshow containing the spectrogram

    See Also
    --------
    :func:`psd`
        :func:`psd` differs in the default overlap; in returning the mean
        of the segment periodograms; in not returning times; and in
        generating a line plot instead of colormap.

    :func:`magnitude_spectrum`
        A single spectrum, similar to having a single segment when *mode*
        is 'magnitude'. Plots a line instead of a colormap.

    :func:`angle_spectrum`
        A single spectrum, similar to having a single segment when *mode*
        is 'angle'. Plots a line instead of a colormap.

    :func:`phase_spectrum`
        A single spectrum, similar to having a single segment when *mode*
        is 'phase'. Plots a line instead of a colormap.

    Notes
    -----
    The parameters *detrend* and *scale_by_freq* do only apply when *mode*
    is set to 'psd'.

    .. note::
        In addition to the above described arguments, this function can take a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All arguments with the following names: 'x'.

        Objects passed as **data** must support item access (``data[<arg>]``) and
        membership test (``<arg> in data``).

spring()
    Set the colormap to "spring".

    This changes the default colormap as well as the colormap of the current
    image if there is one. See ``help(colormaps)`` for more information.

spy(Z, precision=0, marker=None, markersize=None, aspect='equal', origin='upper', **kwargs)
    Plot the sparsity pattern of a 2D array.

    This visualizes the non-zero values of the array.

    Two plotting styles are available: image and marker. Both
    are available for full arrays, but only the marker style
    works for `scipy.sparse.spmatrix` instances.

    **Image style**

    If *marker* and *markersize* are *None*, `~.Axes.imshow` is used. Any
    extra remaining kwargs are passed to this method.

    **Marker style**

    If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are
    *None*, a `~matplotlib.lines.Line2D` object will be returned with
    the value of marker determining the marker type, and any
    remaining kwargs passed to `~.Axes.plot`.

    Parameters
    ----------
    Z : array-like (M, N)
        The array to be plotted.

    precision : float or 'present', optional, default: 0
        If *precision* is 0, any non-zero value will be plotted. Otherwise,
        values of :math:`|Z| > precision` will be plotted.

        For :class:`scipy.sparse.spmatrix` instances, you can also
        pass 'present'. In this case any value present in the array
        will be plotted, even if it is identically zero.

    origin : {'upper', 'lower'}, optional
        Place the [0,0] index of the array in the upper left or lower left
        corner of the axes. The convention 'upper' is typically used for
        matrices and images.
        If not given, :rc:`image.origin` is used, defaulting to 'upper'.


    aspect : {'equal', 'auto', None} or float, optional
        Controls the aspect ratio of the axes. The aspect is of particular
        relevance for images since it may distort the image, i.e. pixel
        will not be square.

        This parameter is a shortcut for explicitly calling
        `.Axes.set_aspect`. See there for further details.

        - 'equal': Ensures an aspect ratio of 1. Pixels will be square.
        - 'auto': The axes is kept fixed and the aspect is adjusted so
          that the data fit in the axes. In general, this will result in
          non-square pixels.
        - *None*: Use :rc:`image.aspect` (default: 'equal').

        Default: 'equal'

    Returns
    -------
    ret : `~matplotlib.image.AxesImage` or `.Line2D`
        The return type depends on the plotting style (see above).

    Other Parameters
    ----------------
    **kwargs
        The supported additional parameters depend on the plotting style.

        For the image style, you can pass the following additional
        parameters of `~.Axes.imshow`:

        - *cmap*
        - *alpha*
        - *url*
        - any `.Artist` properties (passed on to the `.AxesImage`)

        For the marker style, you can pass any `.Line2D` property except
        for *linestyle*:

      agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
      alpha: float
      animated: bool
      antialiased or aa: bool
      clip_box: `.Bbox`
      clip_on: bool
      clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
      color or c: color
      contains: callable
      dash_capstyle: {'butt', 'round', 'projecting'}
      dash_joinstyle: {'miter', 'round', 'bevel'}
      dashes: sequence of floats (on/off ink in points) or (None, None)
      drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-post'}, default: 'default'
      figure: `.Figure`
      fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
      gid: str
      in_layout: bool
      label: object
      linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
      linewidth or lw: float
      marker: marker style
      markeredgecolor or mec: color
      markeredgewidth or mew: float
      markerfacecolor or mfc: color
      markerfacecoloralt or mfcalt: color
      markersize or ms: float
      markevery: None or int or (int, int) or slice or List[int] or float or (float, float)
      path_effects: `.AbstractPathEffect`
      picker: float or callable[[Artist, Event], Tuple[bool, dict]]
      pickradius: float
      rasterized: bool or None
      sketch_params: (scale: float, length: float, randomness: float)
      snap: bool or None
      solid_capstyle: {'butt', 'round', 'projecting'}
      solid_joinstyle: {'miter', 'round', 'bevel'}
      transform: `matplotlib.transforms.Transform`
      url: str
      visible: bool
      xdata: 1D array
      ydata: 1D array
      zorder: float

stackplot(x, *args, labels=(), colors=None, baseline='zero', data=None, **kwargs)
    Draw a stacked area plot.

    Parameters
    ----------
    x : 1d array of dimension N

    y : 2d array (dimension MxN), or sequence of 1d arrays (each dimension 1xN)

        The data is assumed to be unstacked. Each of the following
        calls is legal::

            stackplot(x, y)               # where y is MxN
            stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1xNm

    baseline : {'zero', 'sym', 'wiggle', 'weighted_wiggle'}
        Method used to calculate the baseline:

        - ``'zero'``: Constant zero baseline, i.e. a simple stacked plot.
        - ``'sym'``:  Symmetric around zero and is sometimes called
          'ThemeRiver'.
        - ``'wiggle'``: Minimizes the sum of the squared slopes.
        - ``'weighted_wiggle'``: Does the same but weights to account for
          size of each layer. It is also called 'Streamgraph'-layout. More
          details can be found at http://leebyron.com/streamgraph/.

    labels : Length N sequence of strings
        Labels to assign to each data series.

    colors : Length N sequence of colors
        A list or tuple of colors. These will be cycled through and used to
        colour the stacked areas.

    **kwargs
        All other keyword arguments are passed to `Axes.fill_between()`.


    Returns
    -------
    list : list of `.PolyCollection`
        A list of `.PolyCollection` instances, one for each element in the
        stacked area plot.

    Notes
    -----


    .. note::
        In addition to the above described arguments, this function can take a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All positional and all keyword arguments.

        Objects passed as **data** must support item access (``data[<arg>]``) and
        membership test (``<arg> in data``).

stem(*args, linefmt=None, markerfmt=None, basefmt=None, bottom=0, label=None, use_line_collection=False, da
ta=None)
    Create a stem plot.

    A stem plot plots vertical lines at each *x* location from the baseline
    to *y*, and places a marker there.

    Call signature::

      stem([x,] y, linefmt=None, markerfmt=None, basefmt=None)

    The x-positions are optional. The formats may be provided either as
    positional or as keyword-arguments.

    Parameters
    ----------
    x : array-like, optional
        The x-positions of the stems. Default: (0, 1, ..., len(y) - 1).

    y : array-like
        The y-values of the stem heads.

    linefmt : str, optional
        A string defining the properties of the vertical lines. Usually,
        this will be a color or a color and a linestyle:

        =========  =============
        Character  Line Style
        =========  =============
        ``'-'``    solid line
        ``'--'``   dashed line
        ``'-.'``   dash-dot line
        ``':'``    dotted line
        =========  =============

        Default: 'C0-', i.e. solid line with the first color of the color
        cycle.

        Note: While it is technically possible to specify valid formats
        other than color or color and linestyle (e.g. 'rx' or '-.'), this
        is beyond the intention of the method and will most likely not
        result in a reasonable reasonable plot.

    markerfmt : str, optional
        A string defining the properties of the markers at the stem heads.
        Default: 'C0o', i.e. filled circles with the first color of the
        color cycle.

    basefmt : str, optional
        A format string defining the properties of the baseline.

        Default: 'C3-' ('C2-' in classic mode).

    bottom : float, optional, default: 0
        The y-position of the baseline.

    label : str, optional, default: None
        The label to use for the stems in legends.

    use_line_collection : bool, optional, default: False
        If ``True``, store and plot the stem lines as a
        `~.collections.LineCollection` instead of individual lines. This
        significantly increases performance, and will become the default
        option in Matplotlib 3.3. If ``False``, defaults to the old
        behavior of using a list of `.Line2D` objects.


    Returns
    -------
    container : :class:`~matplotlib.container.StemContainer`
        The container may be treated like a tuple
        (*markerline*, *stemlines*, *baseline*)


    Notes
    -----

    .. seealso::
        The MATLAB function
        `stem <http://www.mathworks.com/help/techdoc/ref/stem.html>`_
        which inspired this method.

    .. note::
        In addition to the above described arguments, this function can take a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

        * All positional and all keyword arguments.

        Objects passed as **data** must support item access (``data[<arg>]``) and
        membership test (``<arg> in data``).

step(x, y, *args, where='pre', data=None, **kwargs)
    Make a step plot.

    Call signatures::

        step(x, y, [fmt], *, data=None, where='pre', **kwargs)
        step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)

    This is just a thin wrapper around `.plot` which changes some
    formatting options. Most of the concepts and parameters of plot can be
    used here as well.

    Parameters
    ----------
    x : array_like
        1-D sequence of x positions. It is assumed, but not checked, that
        it is uniformly increasing.

    y : array_like
        1-D sequence of y levels.

    fmt : str, optional
        A format string, e.g. 'g' for a green line. See `.plot` for a more
        detailed description.

        Note: While full format strings are accepted, it is recommended to
        only specify the color. Line styles are currently ignored (use
        the keyword argument *linestyle* instead). Markers are accepted
        and plotted on the given positions, however, this is a rarely
        needed feature for step plots.

    data : indexable object, optional
        An object with labelled data. If given, provide the label names to
        plot in *x* and *y*.

    where : {'pre', 'post', 'mid'}, optional, default 'pre'
        Define where the steps should be placed:
```

```
            - 'pre': The y value is continued constantly to the left from
              every *x* position, i.e. the interval ``(x[i-1], x[i]]`` has the
              value ``y[i]``.
            - 'post': The y value is continued constantly to the right from
              every *x* position, i.e. the interval ``[x[i], x[i+1])`` has the
              value ``y[i]``.
            - 'mid': Steps occur half-way between the *x* positions.

        Returns
        -------
        lines
            A list of `.Line2D` objects representing the plotted data.

        Other Parameters
        ----------------
        **kwargs
            Additional parameters are the same as those for `.plot`.

        Notes
        -----
        .. [notes section required to get data note injection right]

    streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, norm=None, arrowsize=1, arrowstyle
='-|>', minlength=0.1, transform=None, zorder=None, start_points=None, maxlength=4.0, integration_direction='bo
th', *, data=None)
        Draw streamlines of a vector flow.

        Parameters
        ----------
        x, y : 1D arrays
            An evenly spaced grid.
        u, v : 2D arrays
            *x* and *y*-velocities. The number of rows and columns must match
            the length of *y* and *x*, respectively.
        density : float or (float, float)
            Controls the closeness of streamlines. When ``density = 1``, the domain
            is divided into a 30x30 grid. *density* linearly scales this grid.
            Each cell in the grid can have, at most, one traversing streamline.
            For different densities in each direction, use a tuple
            (density_x, density_y).
        linewidth : float or 2D array
            The width of the stream lines. With a 2D array the line width can be
            varied across the grid. The array must have the same shape as *u*
            and *v*.
        color : matplotlib color code, or 2D array
            The streamline color. If given an array, its values are converted to
            colors using *cmap* and *norm*.  The array must have the same shape
            as *u* and *v*.
        cmap : `~matplotlib.colors.Colormap`
            Colormap used to plot streamlines and arrows. This is only used if
            *color* is an array.
        norm : `~matplotlib.colors.Normalize`
            Normalize object used to scale luminance data to 0, 1. If ``None``,
            stretch (min, max) to (0, 1). This is only used if *color* is an array.
        arrowsize : float
            Scaling factor for the arrow size.
        arrowstyle : str
            Arrow style specification.
            See `~matplotlib.patches.FancyArrowPatch`.
        minlength : float
            Minimum length of streamline in axes coordinates.
        start_points : Nx2 array
            Coordinates of starting points for the streamlines in data coordinates
            (the same coordinates as the *x* and *y* arrays).
        zorder : int
            The zorder of the stream lines and arrows.
            Artists with lower zorder values are drawn first.
        maxlength : float
            Maximum length of streamline in axes coordinates.
        integration_direction : {'forward', 'backward', 'both'}
            Integrate the streamline in forward, backward or both directions.
            default is ``'both'``.

        Returns
        -------
        stream_container : StreamplotSet
            Container object with attributes

            - ``lines``: `.LineCollection` of streamlines

            - ``arrows``: `.PatchCollection` containing `.FancyArrowPatch`
              objects representing the arrows half-way along stream lines.

            This container will probably change in the future to allow changes
            to the colormap, alpha, etc. for both lines and arrows, but these
            changes should be backward compatible.

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'start_points', 'u', 'v', 'x', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    subplot(*args, **kwargs)
        Add a subplot to the current figure.

        Wrapper of `.Figure.add_subplot` with a difference in behavior
        explained in the notes section.

        Call signatures::

           subplot(nrows, ncols, index, **kwargs)
           subplot(pos, **kwargs)
           subplot(ax)

        Parameters
        ----------
        *args
            Either a 3-digit integer or three separate integers
            describing the position of the subplot. If the three
            integers are *nrows*, *ncols*, and *index* in order, the
            subplot will take the *index* position on a grid with *nrows*
            rows and *ncols* columns. *index* starts at 1 in the upper left
            corner and increases to the right.

            *pos* is a three digit integer, where the first digit is the
            number of rows, the second the number of columns, and the third
            the index of the subplot. i.e. fig.add_subplot(235) is the same as
            fig.add_subplot(2, 3, 5). Note that all integers must be less than
            10 for this form to work.

        projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'polar', 'rectilinear', str}, optional
            The projection type of the subplot (`~.axes.Axes`). *str* is the name
            of a costum projection, see `~matplotlib.projections`. The default
            None results in a 'rectilinear' projection.

        polar : boolean, optional
            If True, equivalent to projection='polar'.

        sharex, sharey : `~.axes.Axes`, optional
            Share the x or y `~matplotlib.axis` with sharex and/or sharey. The
            axis will have the same limits, ticks, and scale as the axis of the
            shared axes.

        label : str
            A label for the returned axes.

        Other Parameters
        ----------------
        **kwargs
            This method also takes the keyword arguments for
            the returned axes base class. The keyword arguments for the
            rectilinear base class `~.axes.Axes` can be found in
            the following table but there might also be other keyword
            arguments if another projection is used.
              adjustable: {'box', 'datalim'}
          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          anchor: 2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
          animated: bool
          aspect: {'auto', 'equal'} or num
          autoscale_on: bool
          autoscalex_on: bool
          autoscaley_on: bool
          axes_locator: Callable[[Axes, Renderer], Bbox]
          axisbelow: bool or 'line'
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          contains: callable
          facecolor: color
          fc: color
          figure: `.Figure`
          frame_on: bool
          gid: str
          in_layout: bool
          label: object
          navigate: bool
          navigate_mode: unknown
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          position: [left, bottom, width, height] or `~matplotlib.transforms.Bbox`
          rasterization_zorder: float or None
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          title: str
          transform: `.Transform`
          url: str
          visible: bool
          xbound: unknown
          xlabel: str
          xlim: (left: float, right: float)
          xmargin: float greater than -0.5
          xscale: {"linear", "log", "symlog", "logit", ...}
          xticklabels: List[str]
          xticks: list
          ybound: unknown
          ylabel: str
          ylim: (bottom: float, top: float)
          ymargin: float greater than -0.5
          yscale: {"linear", "log", "symlog", "logit", ...}
          yticklabels: List[str]
          yticks: list
          zorder: float

        Returns
        -------
        axes : an `.axes.SubplotBase` subclass of `~.axes.Axes` (or a subclass     of `~.axes.Axes`)

            The axes of the subplot. The returned axes base class depends on
            the projection used. It is `~.axes.Axes` if rectilinear projection
            are used and `.projections.polar.PolarAxes` if polar projection
            are used. The returned axes is then a subplot subclass of the
            base class.

        Notes
        -----
        Creating a subplot will delete any pre-existing subplot that overlaps
        with it beyond sharing a boundary::

            import matplotlib.pyplot as plt
            # plot a line, implicitly creating a subplot(111)
            plt.plot([1,2,3])
            # now create a subplot which represents the top plot of a grid
            # with 2 rows and 1 column. Since this subplot will overlap the
            # first, the plot (and its axes) previously created, will be removed
            plt.subplot(211)

        If you do not want this behavior, use the `.Figure.add_subplot` method
        or the `.pyplot.axes` function instead.

        If the figure already has a subplot with key (*args*,
        *kwargs*) then it will simply make that subplot current and
        return it.  This behavior is deprecated. Meanwhile, if you do
        not want this behavior (i.e., you want to force the creation of a
        new subplot), you must use a unique set of args and kwargs.  The axes
        *label* attribute has been exposed for this purpose: if you want
        two subplots that are otherwise identical to be added to the figure,
        make sure you give them unique labels.

        In rare circumstances, `.add_subplot` may be called with a single
        argument, a subplot axes instance already created in the
        present figure but not in the figure's list of axes.

        See Also
        --------
        .Figure.add_subplot
        .pyplot.subplots
        .pyplot.axes
        .Figure.subplots

        Examples
        --------
        ::

            plt.subplot(221)

            # equivalent but more general
            ax1=plt.subplot(2, 2, 1)

            # add a subplot with no frame
            ax2=plt.subplot(222, frameon=False)

            # add a polar subplot
            plt.subplot(223, projection='polar')

            # add a red subplot that shares the x-axis with ax1
            plt.subplot(224, sharex=ax1, facecolor='red')

            # delete ax2 from the figure
            plt.delaxes(ax2)

            # add ax2 to the figure again
            plt.subplot(ax2)

    subplot2grid(shape, loc, rowspan=1, colspan=1, fig=None, **kwargs)
        Create an axis at specific location inside a regular grid.

        Parameters
        ----------
        shape : sequence of 2 ints
            Shape of grid in which to place axis.
            First entry is number of rows, second entry is number of columns.

        loc : sequence of 2 ints
            Location to place axis within grid.
            First entry is row number, second entry is column number.

        rowspan : int
            Number of rows for the axis to span to the right.

        colspan : int
            Number of columns for the axis to span downwards.

        fig : `Figure`, optional
            Figure to place axis in. Defaults to current figure.

        **kwargs
            Additional keyword arguments are handed to `add_subplot`.


        Notes
        -----
        The following call ::

            subplot2grid(shape, loc, rowspan=1, colspan=1)

        is identical to ::

            gridspec=GridSpec(shape[0], shape[1])
            subplotspec=gridspec.new_subplotspec(loc, rowspan, colspan)
            subplot(subplotspec)

    subplot_tool(targetfig=None)
        Launch a subplot tool window for a figure.

        A :class:`matplotlib.widgets.SubplotTool` instance is returned.

    subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplot_kw=None, gridspec_kw=None, **f
ig_kw)
        Create a figure and a set of subplots.

        This utility wrapper makes it convenient to create common layouts of
        subplots, including the enclosing figure object, in a single call.

        Parameters
        ----------
        nrows, ncols : int, optional, default: 1
            Number of rows/columns of the subplot grid.

        sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
            Controls sharing of properties among x (`sharex`) or y (`sharey`)
            axes:

                - True or 'all': x- or y-axis will be shared among all
                  subplots.
                - False or 'none': each subplot x- or y-axis will be
                  independent.
                - 'row': each subplot row will share an x- or y-axis.
                - 'col': each subplot column will share an x- or y-axis.

            When subplots have a shared x-axis along a column, only the x tick
            labels of the bottom subplot are created. Similarly, when subplots
            have a shared y-axis along a row, only the y tick labels of the first
            column subplot are created. To later turn other subplots' ticklabels
            on, use `~matplotlib.axes.Axes.tick_params`.

        squeeze : bool, optional, default: True
            - If True, extra dimensions are squeezed out from the returned
              array of `~matplotlib.axes.Axes`:

                - if only one subplot is constructed (nrows=ncols=1), the
                  resulting single Axes object is returned as a scalar.
                - for Nx1 or 1xM subplots, the returned object is a 1D numpy
                  object array of Axes objects.
                - for NxM, subplots with N>1 and M>1 are returned as a 2D array.

            - If False, no squeezing at all is done: the returned Axes object is
              always a 2D array containing Axes instances, even if it ends up
              being 1x1.

        num : integer or string, optional, default: None
            A `.pyplot.figure` keyword that sets the figure number or label.

        subplot_kw : dict, optional
            Dict with keywords passed to the
            `~matplotlib.figure.Figure.add_subplot` call used to create each
            subplot.

        gridspec_kw : dict, optional
            Dict with keywords passed to the `~matplotlib.gridspec.GridSpec`
            constructor used to create the grid the subplots are placed on.

        **fig_kw
            All additional keyword arguments are passed to the
            `.pyplot.figure` call.

        Returns
        -------
        fig : `~.figure.Figure`

        ax : `.axes.Axes` object or array of Axes objects.
            *ax* can be either a single `~matplotlib.axes.Axes` object or an
            array of Axes objects if more than one subplot was created.  The
            dimensions of the resulting array can be controlled with the squeeze
            keyword, see above.

        Examples
        --------
        ::

            # First create some toy data:
            x = np.linspace(0, 2*np.pi, 400)
            y = np.sin(x**2)

            # Creates just a figure and only one subplot
            fig, ax = plt.subplots()
            ax.plot(x, y)
            ax.set_title('Simple plot')

            # Creates two subplots and unpacks the output array immediately
            f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
            ax1.plot(x, y)
            ax1.set_title('Sharing Y axis')
            ax2.scatter(x, y)

            # Creates four polar axes, and accesses them through the returned array
            fig, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
            axes[0, 0].plot(x, y)
            axes[1, 1].scatter(x, y)

            # Share a X axis with each column of subplots
            plt.subplots(2, 2, sharex='col')

            # Share a Y axis with each row of subplots
            plt.subplots(2, 2, sharey='row')

            # Share both X and Y axes with all subplots
            plt.subplots(2, 2, sharex='all', sharey='all')

            # Note that this is the same as
            plt.subplots(2, 2, sharex=True, sharey=True)

            # Creates figure number 10 with a single subplot
            # and clears it if it already exists.
            fig, ax=plt.subplots(num=10, clear=True)

        See Also
        --------
        .pyplot.figure
        .pyplot.subplot
        .pyplot.axes
        .Figure.subplots
        .Figure.add_subplot

    subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None, hspace=None)
        Tune the subplot layout.

        The parameter meanings (and suggested defaults) are::

          left = 0.125  # the left side of the subplots of the figure
          right = 0.9   # the right side of the subplots of the figure
          bottom = 0.1  # the bottom of the subplots of the figure
          top = 0.9     # the top of the subplots of the figure
          wspace = 0.2  # the amount of width reserved for space between subplots,
                        # expressed as a fraction of the average axis width
          hspace = 0.2  # the amount of height reserved for space between subplots,
                        # expressed as a fraction of the average axis height

        The actual defaults are controlled by the rc file

    summer()
        Set the colormap to "summer".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    suptitle(t, **kwargs)
        Add a centered title to the figure.

        Parameters
        ----------
        t : str
            The title text.

        x : float, default 0.5
            The x location of the text in figure coordinates.

        y : float, default 0.98
            The y location of the text in figure coordinates.

        horizontalalignment, ha : {'center', 'left', right'}, default: 'center'
            The horizontal alignment of the text relative to (*x*, *y*).

        verticalalignment, va : {'top', 'center', 'bottom', 'baseline'}, default: 'top'
            The vertical alignment of the text relative to (*x*, *y*).

        fontsize, size : default: :rc:`figure.titlesize`
            The font size of the text. See `.Text.set_size` for possible
            values.

        fontweight, weight : default: :rc:`figure.titleweight`
            The font weight of the text. See `.Text.set_weight` for possible
            values.


        Returns
        -------
            text
                The `.Text` instance of the title.


        Other Parameters
        ----------------
        fontproperties : None or dict, optional
            A dict of font properties. If *fontproperties* is given the
            default values for font size and weight are taken from the
            `FontProperties` defaults. :rc:`figure.titlesize` and
            :rc:`figure.titleweight` are ignored in this case.

        **kwargs
            Additional kwargs are :class:`matplotlib.text.Text` properties.


        Examples
        --------

        >>> fig.suptitle('This is the figure title', fontsize=12)

    switch_backend(newbackend)
        Close all open figures and set the Matplotlib backend.

        The argument is case-insensitive.  Switching to an interactive backend is
        possible only if no event loop for another interactive backend has started.
        Switching to and from non-interactive backends is always possible.

        Parameters
        ----------
        newbackend : str
            The name of the backend to use.

    table(cellText=None, cellColours=None, cellLoc='right', colWidths=None, rowLabels=None, rowColours=None, ro
wLoc='left', colLabels=None, colColours=None, colLoc='center', loc='bottom', bbox=None, edges='closed', **kwarg
s)
        Add a table to an `~.axes.Axes`.

        At least one of *cellText* or *cellColours* must be specified. These
        parameters must be 2D lists, in which the outer lists define the rows and
        the inner list define the column values per row. Each row must have the
        same number of elements.

        The table can optionally have row and column headers, which are configured
        using *rowLabels*, *rowColours*, *rowLoc* and *colLabels*, *colColours*,
        *colLoc* respectively.

        For finer grained control over tables, use the `.Table` class and add it to
        the axes with `.Axes.add_table`.

        Parameters
        ----------
        cellText : 2D list of str, optional
            The texts to place into the table cells.

            *Note*: Line breaks in the strings are currently not accounted for and
            will result in the text exceeding the cell boundaries.

        cellColours : 2D list of matplotlib color specs, optional
            The background colors of the cells.

        cellLoc : {'left', 'center', 'right'}, default: 'right'
            The alignment of the text within the cells.

        colWidths : list of float, optional
            The column widths in units of the axes. If not given, all columns will
            have a width of *1 / ncols*.

        rowLabels : list of str, optional
            The text of the row header cells.

        rowColours : list of matplotlib color specs, optional
            The colors of the row header cells.

        rowLoc : {'left', 'center', 'right'}, optional, default: 'left'
            The text alignment of the row header cells.

        colLabels : list of str, optional
            The text of the column header cells.

        colColours : list of matplotlib color specs, optional
            The colors of the column header cells.

        rowLoc : {'left', 'center', 'right'}, optional, default: 'left'
            The text alignment of the column header cells.

        loc : str, optional
            The position of the cell with respect to *ax*. This must be one of
            the `~.Table.codes`.

        bbox : `.Bbox`, optional
            A bounding box to draw the table into. If this is not *None*, this
            overrides *loc*.

        edges : substring of 'BRTL' or {'open', 'closed', 'horizontal', 'vertical'}
            The cell edges to be drawn with a line. See also
            `~.CustomCell.visible_edges`.

        Other Parameters
        ----------------
        **kwargs
            `.Table` properties.

          agg_filter: a filter function, which takes a (m, n, 3) float array and a dpi value, and returns a (m,
 n, 3) array
          alpha: float
          animated: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
          contains: callable
          figure: `.Figure`
          fontsize: float
          gid: str
          in_layout: bool
          label: object
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          visible: bool
          zorder: float

        Returns
        -------
        table : `~matplotlib.table.Table`
            The created table.

    text(x, y, s, fontdict=None, withdash=<deprecated parameter>, **kwargs)
        Add text to the axes.

        Add the text *s* to the axes at location *x*, *y* in data coordinates.

        Parameters
        ----------
        x, y : scalars
            The position to place the text. By default, this is in data
            coordinates. The coordinate system can be changed using the
            *transform* parameter.

        s : str
            The text.

        fontdict : dictionary, optional, default: None
            A dictionary to override the default text properties. If fontdict
            is None, the defaults are determined by your rc parameters.

        withdash : boolean, optional, default: False
            Creates a `~matplotlib.text.TextWithDash` instance instead of a
            `~matplotlib.text.Text` instance.

        Returns
        -------
        text : `.Text`
            The created `.Text` instance.

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.text.Text` properties.
            Other miscellaneous text parameters.

        Examples
        --------
        Individual keyword arguments can be used to override any given
        parameter::

            >>> text(x, y, s, fontsize=12)

        The default transform specifies that text is in data coords,
        alternatively, you can specify text in axis coords (0,0 is
        lower-left and 1,1 is upper-right).  The example below places
        text in the center of the axes::

            >>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
            ...      verticalalignment='center', transform=ax.transAxes)

        You can put a rectangular box around the text instance (e.g., to
        set a background color) by using the keyword `bbox`.  `bbox` is
        a dictionary of `~matplotlib.patches.Rectangle`
        properties.  For example::

            >>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))

    thetagrids(*args, **kwargs)
        Get or set the theta gridlines on the current polar plot.

        Call signatures::

         lines, labels = thetagrids()
         lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)

        When called with no arguments, `.thetagrids` simply returns the tuple
        (*lines*, *labels*). When called with arguments, the labels will
        appear at the specified angles.

        Parameters
        ----------
        angles : tuple with floats, degrees
            The angles of the theta gridlines.

        labels : tuple with strings or None
            The labels to use at each radial gridline. The
            `.projections.polar.ThetaFormatter` will be used if None.

        fmt : str or None
            Format string used in `matplotlib.ticker.FormatStrFormatter`.
            For example '%f'. Note that the angle in radians will be used.

        Returns
        -------
        lines, labels : list of `.lines.Line2D`, list of `.text.Text`
            *lines* are the theta gridlines and *labels* are the tick labels.

        Other Parameters
        ----------------
        **kwargs
            *kwargs* are optional `~.Text` properties for the labels.

        Examples
        --------
        ::

          # set the locations of the angular gridlines
          lines, labels = thetagrids( range(45,360,90) )

          # set the locations and labels of the angular gridlines
          lines, labels = thetagrids( range(45,360,90), ('NE', 'NW', 'SW','SE') )

        See Also
        --------
        .pyplot.rgrids
        .projections.polar.PolarAxes.set_thetagrids
        .Axis.get_gridlines
        .Axis.get_ticklabels

    tick_params(axis='both', **kwargs)
        Change the appearance of ticks, tick labels, and gridlines.

        Parameters
        ----------
        axis : {'x', 'y', 'both'}, optional
            Which axis to apply the parameters to.

        Other Parameters
        ----------------

        axis : {'x', 'y', 'both'}
            Axis on which to operate; default is 'both'.

        reset : bool
            If *True*, set all parameters to defaults
            before processing other keyword arguments.  Default is
            *False*.

        which : {'major', 'minor', 'both'}
            Default is 'major'; apply arguments to *which* ticks.

        direction : {'in', 'out', 'inout'}
            Puts ticks inside the axes, outside the axes, or both.

        length : float
            Tick length in points.

        width : float
            Tick width in points.

        color : color
            Tick color; accepts any mpl color spec.

        pad : float
            Distance in points between tick and label.

        labelsize : float or str
            Tick label font size in points or as a string (e.g., 'large').

        labelcolor : color
            Tick label color; mpl color spec.

        colors : color
            Changes the tick color and the label color to the same value:
            mpl color spec.

        zorder : float
            Tick and label zorder.

        bottom, top, left, right : bool
            Whether to draw the respective ticks.

        labelbottom, labeltop, labelleft, labelright : bool
            Whether to draw the respective tick labels.

        labelrotation : float
            Tick label rotation

        grid_color : color
            Changes the gridline color to the given mpl color spec.

        grid_alpha : float
            Transparency of gridlines: 0 (transparent) to 1 (opaque).

        grid_linewidth : float
            Width of gridlines in points.

        grid_linestyle : string
            Any valid `~matplotlib.lines.Line2D` line style spec.

        Examples
        --------

        Usage ::

            ax.tick_params(direction='out', length=6, width=2, colors='r',
                           grid_color='r', grid_alpha=0.5)

        This will make all major ticks be red, pointing out of the box,
        and with dimensions 6 points by 2 points.  Tick labels will
        also be red.  Gridlines will be red and translucent.

    ticklabel_format(*, axis='both', style='', scilimits=None, useOffset=None, useLocale=None, useMathText=Non
e)
        Change the `~matplotlib.ticker.ScalarFormatter` used by
        default for linear axes.

        Optional keyword arguments:

          ==============   =========================================
          Keyword          Description
          ==============   =========================================
          *axis*           [ 'x' | 'y' | 'both' ]
          *style*          [ 'sci' (or 'scientific') | 'plain' ]
                           plain turns off scientific notation
          *scilimits*      (m, n), pair of integers; if *style*
                           is 'sci', scientific notation will
                           be used for numbers outside the range
                           10\ :sup:`m` to 10\ :sup:`n`.
                           Use (0,0) to include all numbers.
                           Use (m,m) where m <> 0 to fix the order
                           of magnitude to 10\ :sup:`m`.
          *useOffset*      [ bool | offset ]; if True,
                           the offset will be calculated as needed;
                           if False, no offset will be used; if a
                           numeric offset is specified, it will be
                           used.
          *useLocale*      If True, format the number according to
                           the current locale.  This affects things
                           such as the character used for the
                           decimal separator.  If False, use
                           C-style (English) formatting.  The
                           default setting is controlled by the
                           axes.formatter.use_locale rcparam.
          *useMathText*    If True, render the offset and scientific
                           notation in mathtext
          ==============   =========================================

        Only the major ticks are affected.
        If the method is called when the `~matplotlib.ticker.ScalarFormatter`
        is not the `~matplotlib.ticker.Formatter` being used, an
        `AttributeError` will be raised.

    tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None)
        Automatically adjust subplot parameters to give specified padding.

        Parameters
        ----------
        pad : float
            Padding between the figure edge and the edges of subplots,
            as a fraction of the font size.
        h_pad, w_pad : float, optional
            Padding (height/width) between edges of adjacent subplots,
            as a fraction of the font size.  Defaults to *pad*.
        rect : tuple (left, bottom, right, top), optional
            A rectangle (left, bottom, right, top) in the normalized
            figure coordinate that the whole subplots area (including
            labels) will fit into. Default is (0, 0, 1, 1).

    title(label, fontdict=None, loc='center', pad=None, **kwargs)
        Set a title for the axes.

        Set one of the three available axes titles. The available titles
        are positioned above the axes in the center, flush with the left
        edge, and flush with the right edge.

        Parameters
        ----------
        label : str
            Text to use for the title

        fontdict : dict
            A dictionary controlling the appearance of the title text,
            the default `fontdict` is::

               {'fontsize': rcParams['axes.titlesize'],
                'fontweight' : rcParams['axes.titleweight'],
                'verticalalignment': 'baseline',
                'horizontalalignment': loc}

        loc : {'center', 'left', 'right'}, str, optional
            Which title to set, defaults to 'center'

        pad : float
            The offset of the title from the top of the axes, in points.
            Default is ``None`` to use rcParams['axes.titlepad'].

        Returns
        -------
        text : :class:`~matplotlib.text.Text`
            The matplotlib text instance representing the title

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.text.Text` properties
            Other keyword arguments are text properties, see
            :class:`~matplotlib.text.Text` for a list of valid text
            properties.

    tricontour(*args, **kwargs)
        Draw contours on an unstructured triangular grid.

        `.tricontour` and `.tricontourf` draw contour lines and filled contours,
        respectively.  Except as noted, function signatures and return values are
        the same for both versions.

        The triangulation can be specified in one of two ways; either ::

            tricontour(triangulation, ...)

        where *triangulation* is a `matplotlib.tri.Triangulation` object, or ::

            tricontour(x, y, ...)
            tricontour(x, y, triangles, ...)
            tricontour(x, y, triangles=triangles, ...)
            tricontour(x, y, mask=mask, ...)
            tricontour(x, y, triangles, mask=mask, ...)

        in which case a `.Triangulation` object will be created.  See that class'
        docstring for an explanation of these cases.

        The remaining arguments may be::

            tricontour(..., Z)

        where *Z* is the array of values to contour, one per point in the
        triangulation.  The level values are chosen automatically.

        ::

            tricontour(..., Z, N)

        contour up to *N+1* automatically chosen contour levels (*N* intervals).

        ::

            tricontour(..., Z, V)

        draw contour lines at the values specified in sequence *V*,
        which must be in increasing order.

        ::

            tricontourf(..., Z, V)

        fill the (len(*V*)-1) regions between the values in *V*,
        which must be in increasing order.

        ::

            tricontour(Z, **kwargs)
```

```
        Use keyword args to control colors, linewidth, origin, cmap ... see
        below for more details.

        `.tricontour(...)` returns a `~matplotlib.contour.TriContourSet` object.

        Optional keyword arguments:

            *colors*: [ *None* | string | (mpl_colors) ]
            If *None*, the colormap specified by cmap will be used.

            If a string, like 'r' or 'red', all levels will be plotted in this
            color.

            If a tuple of matplotlib color args (string, float, rgb, etc),
            different levels will be plotted in different colors in the order
            specified.

            *alpha*: float
            The alpha blending value

            *cmap*: [ *None* | Colormap ]
            A cm :class:`~matplotlib.colors.Colormap` instance or
            *None*. If *cmap* is *None* and *colors* is *None*, a
            default Colormap is used.

            *norm*: [ *None* | Normalize ]
            A :class:`matplotlib.colors.Normalize` instance for
            scaling data values to colors. If *norm* is *None* and
            *colors* is *None*, the default linear scaling is used.

            *levels* [level0, level1, ..., leveln]
            A list of floating point numbers indicating the level
            curves to draw, in increasing order; e.g., to draw just
            the zero contour pass ``levels=[0]``

            *origin*: [ *None* | 'upper' | 'lower' | 'image' ]
            If *None*, the first value of *Z* will correspond to the
            lower left corner, location (0,0). If 'image', the rc
            value for ``image.origin`` will be used.

            This keyword is not active if *X* and *Y* are specified in
            the call to contour.

            *extent*: [ *None* | (x0,x1,y0,y1) ]

            If *origin* is not *None*, then *extent* is interpreted as
            in :func:`matplotlib.pyplot.imshow`: it gives the outer
            pixel boundaries. In this case, the position of Z[0,0]
            is the center of the pixel, not a corner. If *origin* is
            *None*, then (*x0*, *y0*) is the position of Z[0,0], and
            (*x1*, *y1*) is the position of Z[-1,-1].

            This keyword is not active if *X* and *Y* are specified in
            the call to contour.

            *locator*: [ *None* | ticker.Locator subclass ]
            If *locator* is None, the default
            :class:`~matplotlib.ticker.MaxNLocator` is used. The
            locator is used to determine the contour levels if they
            are not given explicitly via the *V* argument.

            *extend*: [ 'neither' | 'both' | 'min' | 'max' ]
            Unless this is 'neither', contour levels are automatically
            added to one or both ends of the range so that all data
            are included. These added ranges are then mapped to the
            special colormap values which default to the ends of the
            colormap range, but can be set via
            :meth:`matplotlib.colors.Colormap.set_under` and
            :meth:`matplotlib.colors.Colormap.set_over` methods.

            *xunits*, *yunits*: [ *None* | registered units ]
            Override axis units by specifying an instance of a
            :class:`matplotlib.units.ConversionInterface`.

        tricontour-only keyword arguments:

            *linewidths*: [ *None* | number | tuple of numbers ]
            If *linewidths* is *None*, defaults to rc:`lines.linewidth`.

            If a number, all levels will be plotted with this linewidth.

            If a tuple, different levels will be plotted with different
            linewidths in the order specified

            *linestyles*: [ *None* | 'solid' | 'dashed' | 'dashdot' | 'dotted' ]
            If *linestyles* is *None*, the 'solid' is used.

            *linestyles* can also be an iterable of the above strings
            specifying a set of linestyles to be used. If this
            iterable is shorter than the number of contour levels
            it will be repeated as necessary.

            If contour is using a monochrome colormap and the contour
            level is less than 0, then the linestyle specified
            in :rc:`contour.negative_linestyle` will be used.

        tricontourf-only keyword arguments:

            *antialiased*: bool
            enable antialiasing

        Note: `.tricontourf` fills intervals that are closed at the top; that is,
        for boundaries *z1* and *z2*, the filled region is::

            z1 < Z <= z2

        except for the lowest interval, which is closed on both sides (i.e. it
        includes the lowest value).

    tricontourf(*args, **kwargs)
        Draw contours on an unstructured triangular grid.

        `.tricontour` and `.tricontourf` draw contour lines and filled contours,
        respectively.  Except as noted, function signatures and return values are
        the same for both versions.

        The triangulation can be specified in one of two ways; either ::

            tricontour(triangulation, ...)

        where *triangulation* is a `matplotlib.tri.Triangulation` object, or ::

            tricontour(x, y, ...)
            tricontour(x, y, triangles, ...)
            tricontour(x, y, triangles=triangles, ...)
            tricontour(x, y, mask=mask, ...)
            tricontour(x, y, triangles, mask=mask, ...)

        in which case a `.Triangulation` object will be created.  See that class'
        docstring for an explanation of these cases.

        The remaining arguments may be::

            tricontour(..., Z)

        where *Z* is the array of values to contour, one per point in the
        triangulation.  The level values are chosen automatically.

        ::

            tricontour(..., Z, N)

        contour up to *N+1* automatically chosen contour levels (*N* intervals).

        ::

            tricontour(..., Z, V)

        draw contour lines at the values specified in sequence *V*,
        which must be in increasing order.

        ::

            tricontourf(..., Z, V)

        fill the (len(*V*)-1) regions between the values in *V*,
        which must be in increasing order.

        ::

            tricontour(Z, **kwargs)

        Use keyword args to control colors, linewidth, origin, cmap ... see
        below for more details.

        `.tricontour(...)` returns a `~matplotlib.contour.TriContourSet` object.

        Optional keyword arguments:

            *colors*: [ *None* | string | (mpl_colors) ]
            If *None*, the colormap specified by cmap will be used.

            If a string, like 'r' or 'red', all levels will be plotted in this
            color.

            If a tuple of matplotlib color args (string, float, rgb, etc),
            different levels will be plotted in different colors in the order
            specified.

            *alpha*: float
            The alpha blending value

            *cmap*: [ *None* | Colormap ]
            A cm :class:`~matplotlib.colors.Colormap` instance or
            *None*. If *cmap* is *None* and *colors* is *None*, a
            default Colormap is used.

            *norm*: [ *None* | Normalize ]
            A :class:`matplotlib.colors.Normalize` instance for
            scaling data values to colors. If *norm* is *None* and
            *colors* is *None*, the default linear scaling is used.

            *levels* [level0, level1, ..., leveln]
            A list of floating point numbers indicating the level
            curves to draw, in increasing order; e.g., to draw just
            the zero contour pass ``levels=[0]``

            *origin*: [ *None* | 'upper' | 'lower' | 'image' ]
            If *None*, the first value of *Z* will correspond to the
            lower left corner, location (0,0). If 'image', the rc
            value for ``image.origin`` will be used.

            This keyword is not active if *X* and *Y* are specified in
            the call to contour.

            *extent*: [ *None* | (x0,x1,y0,y1) ]

            If *origin* is not *None*, then *extent* is interpreted as
            in :func:`matplotlib.pyplot.imshow`: it gives the outer
            pixel boundaries. In this case, the position of Z[0,0]
            is the center of the pixel, not a corner. If *origin* is
            *None*, then (*x0*, *y0*) is the position of Z[0,0], and
            (*x1*, *y1*) is the position of Z[-1,-1].

            This keyword is not active if *X* and *Y* are specified in
            the call to contour.

            *locator*: [ *None* | ticker.Locator subclass ]
            If *locator* is None, the default
            :class:`~matplotlib.ticker.MaxNLocator` is used. The
            locator is used to determine the contour levels if they
            are not given explicitly via the *V* argument.

            *extend*: [ 'neither' | 'both' | 'min' | 'max' ]
            Unless this is 'neither', contour levels are automatically
            added to one or both ends of the range so that all data
            are included. These added ranges are then mapped to the
            special colormap values which default to the ends of the
            colormap range, but can be set via
            :meth:`matplotlib.colors.Colormap.set_under` and
            :meth:`matplotlib.colors.Colormap.set_over` methods.

            *xunits*, *yunits*: [ *None* | registered units ]
            Override axis units by specifying an instance of a
            :class:`matplotlib.units.ConversionInterface`.

        tricontour-only keyword arguments:

            *linewidths*: [ *None* | number | tuple of numbers ]
            If *linewidths* is *None*, defaults to rc:`lines.linewidth`.

            If a number, all levels will be plotted with this linewidth.

            If a tuple, different levels will be plotted with different
            linewidths in the order specified

            *linestyles*: [ *None* | 'solid' | 'dashed' | 'dashdot' | 'dotted' ]
            If *linestyles* is *None*, the 'solid' is used.

            *linestyles* can also be an iterable of the above strings
            specifying a set of linestyles to be used. If this
            iterable is shorter than the number of contour levels
            it will be repeated as necessary.

            If contour is using a monochrome colormap and the contour
            level is less than 0, then the linestyle specified
            in :rc:`contour.negative_linestyle` will be used.

        tricontourf-only keyword arguments:

            *antialiased*: bool
            enable antialiasing

        Note: `.tricontourf` fills intervals that are closed at the top; that is,
        for boundaries *z1* and *z2*, the filled region is::

            z1 < Z <= z2

        except for the lowest interval, which is closed on both sides (i.e. it
        includes the lowest value).

    tripcolor(*args, alpha=1.0, norm=None, cmap=None, vmin=None, vmax=None, shading='flat', facecolors=None, **
kwargs)
        Create a pseudocolor plot of an unstructured triangular grid.

        The triangulation can be specified in one of two ways; either::

          tripcolor(triangulation, ...)

        where triangulation is a :class:`matplotlib.tri.Triangulation`
        object, or

        ::

          tripcolor(x, y, ...)
          tripcolor(x, y, triangles, ...)
          tripcolor(x, y, triangles=triangles, ...)
          tripcolor(x, y, mask=mask, ...)
          tripcolor(x, y, triangles, mask=mask, ...)

        in which case a Triangulation object will be created.  See
        :class:`~matplotlib.tri.Triangulation` for a explanation of these
        possibilities.

        The next argument must be *C*, the array of color values, either
        one per point in the triangulation if color values are defined at
        points, or one per triangle in the triangulation if color values
        are defined at triangles. If there are the same number of points
        and triangles in the triangulation it is assumed that color
        values are defined at points; to force the use of color values at
        triangles use the kwarg ``facecolors=C`` instead of just ``C``.

        *shading* may be 'flat' (the default) or 'gouraud'. If *shading*
        is 'flat' and C values are defined at points, the color values
        used for each triangle are from the mean C of the triangle's
        three points. If *shading* is 'gouraud' then color values must be
        defined at points.

        The remaining kwargs are the same as for
        :meth:`~matplotlib.axes.Axes.pcolor`.

    triplot(*args, **kwargs)
        Draw a unstructured triangular grid as lines and/or markers.

        The triangulation to plot can be specified in one of two ways;
        either::

          triplot(triangulation, ...)

        where triangulation is a :class:`matplotlib.tri.Triangulation`
        object, or

        ::

          triplot(x, y, ...)
          triplot(x, y, triangles, ...)
          triplot(x, y, triangles=triangles, ...)
          triplot(x, y, mask=mask, ...)
          triplot(x, y, triangles, mask=mask, ...)

        in which case a Triangulation object will be created.  See
        :class:`~matplotlib.tri.Triangulation` for a explanation of these
        possibilities.

        The remaining args and kwargs are the same as for
        :meth:`~matplotlib.axes.Axes.plot`.

        Return a list of 2 :class:`~matplotlib.lines.Line2D` containing
        respectively:

            - the lines plotted for triangles edges
            - the markers plotted for triangles nodes

    twinx(ax=None)
        Make and return a second axes that shares the *x*-axis.  The new axes will
        overlay *ax* (or the current axes if *ax* is *None*), and its ticks will be
        on the right.

        Examples
        --------
        :doc:`/gallery/subplots_axes_and_figures/two_scales`

    twiny(ax=None)
        Make and return a second axes that shares the *y*-axis.  The new axes will
        overlay *ax* (or the current axes if *ax* is *None*), and its ticks will be
        on the top.

        Examples
        --------
        :doc:`/gallery/subplots_axes_and_figures/two_scales`

    uninstall_repl_displayhook()
        Uninstall the matplotlib display hook.

        .. warning

           Need IPython >= 2 for this to work.  For IPython < 2 will raise a
           ``NotImplementedError``

        .. warning

           If you are using vanilla python and have installed another
           display hook this will reset ``sys.displayhook`` to what ever
           function was there when matplotlib installed it's displayhook,
           possibly discarding your changes.

    violinplot(dataset, positions=None, vert=True, widths=0.5, showmeans=False, showextrema=True, showmedians=F
alse, points=100, bw_method=None, *, data=None)
        Make a violin plot.

        Make a violin plot for each column of *dataset* or each vector in
        sequence *dataset*.  Each filled area extends to represent the
        entire data range, with optional lines at the mean, the median,
        the minimum, and the maximum.

        Parameters
        ----------
        dataset : Array or a sequence of vectors.
          The input data.

        positions : array-like, default = [1, 2, ..., n]
          Sets the positions of the violins. The ticks and limits are
          automatically set to match the positions.

        vert : bool, default = True.
          If true, creates a vertical violin plot.
          Otherwise, creates a horizontal violin plot.

        widths : array-like, default = 0.5
          Either a scalar or a vector that sets the maximal width of
          each violin. The default is 0.5, which uses about half of the
          available horizontal space.

        showmeans : bool, default = False
          If `True`, will toggle rendering of the means.

        showextrema : bool, default = True
          If `True`, will toggle rendering of the extrema.

        showmedians : bool, default = False
          If `True`, will toggle rendering of the medians.

        points : scalar, default = 100
          Defines the number of points to evaluate each of the
          gaussian kernel density estimations at.

        bw_method : str, scalar or callable, optional
          The method used to calculate the estimator bandwidth.  This can be
          'scott', 'silverman', a scalar constant or a callable.  If a
          scalar, this will be used directly as `kde.factor`.  If a
          callable, it should take a `GaussianKDE` instance as its only
          parameter and return a scalar. If None (default), 'scott' is used.

        Returns
        -------

        result : dict
          A dictionary mapping each component of the violinplot to a
          list of the corresponding collection instances created. The
          dictionary has the following keys:

            - ``bodies``: A list of the
              :class:`matplotlib.collections.PolyCollection` instances
              containing the filled area of each violin.

            - ``cmeans``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the mean values of each of the
              violin's distribution.

            - ``cmins``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the bottom of each violin's
              distribution.

            - ``cmaxes``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the top of each violin's
              distribution.

            - ``cbars``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the centers of each violin's
              distribution.

            - ``cmedians``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the median values of each of the
              violin's distribution.

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'dataset'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    viridis()
        Set the colormap to "viridis".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    vlines(x, ymin, ymax, colors='k', linestyles='solid', label='', *, data=None, **kwargs)
        Plot vertical lines.

        Plot vertical lines at each *x* from *ymin* to *ymax*.

        Parameters
        ----------
        x : scalar or 1D array_like
            x-indexes where to plot the lines.

        ymin, ymax : scalar or 1D array_like
            Respective beginning and end of each line. If scalars are
            provided, all lines will have same length.

        colors : array_like of colors, optional, default: 'k'

        linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, optional

        label : string, optional, default: ''

        Returns
        -------
        lines : `~matplotlib.collections.LineCollection`

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.collections.LineCollection` properties.

        See also
        --------
        hlines : horizontal lines
        axvline: vertical line across the axes

        Notes
        -----


        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'colors', 'x', 'ymax', 'ymin'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    waitforbuttonpress(*args, **kwargs)
        Blocking call to interact with the figure.

        This will return True is a key was pressed, False if a mouse
        button was pressed and None if *timeout* was reached without
        either being pressed.

        If *timeout* is negative, does not timeout.

    winter()
        Set the colormap to "winter".

        This changes the default colormap as well as the colormap of the current
        image if there is one. See ``help(colormaps)`` for more information.

    xcorr(x, y, normed=True, detrend=<function detrend_none at 0x0B070468>, usevlines=True, maxlags=10, *, data
=None, **kwargs)
        Plot the cross correlation between *x* and *y*.

        The correlation with lag k is defined as
        :math:`\sum_n x[n+k] \cdot y^*[n]`, where :math:`y^*` is the complex
        conjugate of :math:`y`.

        Parameters
        ----------
        x : array-like of length n

        y : array-like of length n

        detrend : callable, optional, default: `mlab.detrend_none`
            *x* and *y* are detrended by the *detrend* callable. This must be a
            function ``x = detrend(x)`` accepting and returning an
            `numpy.array`. Default is no normalization.

        normed : bool, optional, default: True
            If ``True``, input vectors are normalised to unit length.

        usevlines : bool, optional, default: True
            Determines the plot style.

            If ``True``, vertical lines are plotted from 0 to the xcorr value
            using `Axes.vlines`. Additionally, a horizontal line is plotted
            at y=0 using `Axes.axhline`.

            If ``False``, markers are plotted at the xcorr values using
            `Axes.plot`.

        maxlags : int, optional, default: 10
            Number of lags to show. If None, will return all ``2 * len(x) - 1``
            lags.

        Returns
        -------
        lags : array (length ``2*maxlags+1``)
            The lag vector.
        c : array  (length ``2*maxlags+1``)
            The auto correlation vector.
        line : `.LineCollection` or `.Line2D`
            `.Artist` added to the axes of the correlation:

            - `.LineCollection` if *usevlines* is True.
            - `.Line2D` if *usevlines* is False.
        b : `.Line2D` or None
            Horizontal line at 0 if *usevlines* is True
            None *usevlines* is False.

        Other Parameters
        ----------------
        linestyle : `.Line2D` property, optional
            The linestyle for plotting the data points.
            Only used if *usevlines* is ``False``.

        marker : str, optional, default: 'o'
            The marker for plotting the data points.
            Only used if *usevlines* is ``False``.

        Notes
        -----
        The cross correlation is performed with :func:`numpy.correlate` with
        ``mode = "full"``.

        .. note::
            In addition to the above described arguments, this function can take a
            **data** keyword argument. If such a **data** argument is given, the
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'x', 'y'.

            Objects passed as **data** must support item access (``data[<arg>]``) and
            membership test (``<arg> in data``).

    xkcd(scale=1, length=100, randomness=2)
        Turn on `xkcd <https://xkcd.com/>`_ sketch-style drawing mode.
        This will only have effect on things drawn after this function is
        called.

        For best results, the "Humor Sans" font should be installed: it is
        not included with matplotlib.

        Parameters
        ----------
        scale : float, optional
            The amplitude of the wiggle perpendicular to the source line.
        length : float, optional
            The length of the wiggle along the line.
        randomness : float, optional
            The scale factor by which the length is shrunken or expanded.

        Notes
        -----
        This function works by a number of rcParams, so it will probably
        override others you have set before.

        If you want the effects of this function to be temporary, it can
        be used as a context manager, for example::

            with plt.xkcd():
                # This figure will be in XKCD-style
                fig1 = plt.figure()
                # ...

            # This figure will be in regular style
            fig2 = plt.figure()

    xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)
        Set the label for the x-axis.

        Parameters
        ----------
        xlabel : str
            The label text.

        labelpad : scalar, optional, default: None
            Spacing in points from the axes bounding box including ticks
            and tick labels.

        Other Parameters
        ----------------
        **kwargs : `.Text` properties
            `.Text` properties control the appearance of the label.

        See also
        --------
        text : for information on how override and the optional args work

    xlim(*args, **kwargs)
        Get or set the x limits of the current axes.

        Call signatures::

            left, right = xlim()  # return the current xlim
            xlim((left, right))   # set the xlim to left, right
            xlim(left, right)     # set the xlim to left, right

        If you do not specify args, you can pass *left* or *right* as kwargs,
        i.e.::

            xlim(right=3)  # adjust the right leaving left unchanged
            xlim(left=1)  # adjust the left leaving right unchanged

        Setting limits turns autoscaling off for the x-axis.

        Returns
        -------
        left, right
            A tuple of the new x-axis limits.

        Notes
        -----
        Calling this function with no arguments (e.g. ``xlim()``) is the pyplot
        equivalent of calling `~.Axes.get_xlim` on the current axes.
        Calling this function with arguments is the pyplot equivalent of calling
        `~.Axes.set_xlim` on the current axes. All arguments are passed though.

    xscale(value, **kwargs)
        Set the x-axis scale.

        Parameters
        ----------
        value : {"linear", "log", "symlog", "logit", ...}
            The axis scale type to apply.

        **kwargs
            Different keyword arguments are accepted, depending on the scale.
            See the respective class keyword arguments:

            - `matplotlib.scale.LinearScale`
            - `matplotlib.scale.LogScale`
            - `matplotlib.scale.SymmetricalLogScale`
            - `matplotlib.scale.LogitScale`


        Notes
        -----
        By default, Matplotlib supports the above mentioned scales.
        Additionally, custom scales may be registered using
        `matplotlib.scale.register_scale`. These scales can then also
        be used here.

    xticks(ticks=None, labels=None, **kwargs)
        Get or set the current tick locations and labels of the x-axis.

        Call signatures::

            locs, labels = xticks()            # Get locations and labels
            xticks(ticks, [labels], **kwargs)  # Set locations and labels

        Parameters
        ----------
        ticks : array_like
            A list of positions at which ticks should be placed. You can pass an
            empty list to disable xticks.

        labels : array_like, optional
            A list of explicit labels to place at the given *locs*.

        **kwargs
            :class:`.Text` properties can be used to control the appearance of
            the labels.

        Returns
        -------
        locs
            An array of label locations.
        labels
            A list of `.Text` objects.

        Notes
        -----
        Calling this function with no arguments (e.g. ``xticks()``) is the pyplot
        equivalent of calling `~.Axes.get_xticks` and `~.Axes.get_xticklabels` on
        the current axes.
        Calling this function with arguments is the pyplot equivalent of calling
        `~.Axes.set_xticks` and `~.Axes.set_xticklabels` on the current axes.

        Examples
        --------
        Get the current locations and labels:

            >>> locs, labels = xticks()

        Set label locations:

            >>> xticks(np.arange(0, 1, step=0.2))

        Set text labels:

            >>> xticks(np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))

        Set text labels and properties:

            >>> xticks(np.arange(12), calendar.month_name[1:13], rotation=20)

        Disable xticks:

            >>> xticks([])

    ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)
        Set the label for the y-axis.

        Parameters
        ----------
        ylabel : str
            The label text.

        labelpad : scalar, optional, default: None
            Spacing in points from the axes bounding box including ticks
            and tick labels.

        Other Parameters
        ----------------
        **kwargs : `.Text` properties
            `.Text` properties control the appearance of the label.

        See also
        --------
        text : for information on how override and the optional args work

    ylim(*args, **kwargs)
        Get or set the y-limits of the current axes.

        Call signatures::

            bottom, top = ylim()  # return the current ylim
            ylim((bottom, top))   # set the ylim to bottom, top
            ylim(bottom, top)     # set the ylim to bottom, top

        If you do not specify args, you can alternatively pass *bottom* or
        *top* as kwargs, i.e.::

            ylim(top=3)  # adjust the top leaving bottom unchanged
            ylim(bottom=1)  # adjust the bottom leaving top unchanged

        Setting limits turns autoscaling off for the y-axis.

        Returns
        -------
        bottom, top
            A tuple of the new y-axis limits.

        Notes
        -----
        Calling this function with no arguments (e.g. ``ylim()``) is the pyplot
        equivalent of calling `~.Axes.get_ylim` on the current axes.
        Calling this function with arguments is the pyplot equivalent of calling
        `~.Axes.set_ylim` on the current axes. All arguments are passed though.

    yscale(value, **kwargs)
        Set the y-axis scale.

        Parameters
        ----------
        value : {"linear", "log", "symlog", "logit", ...}
            The axis scale type to apply.

        **kwargs
            Different keyword arguments are accepted, depending on the scale.
            See the respective class keyword arguments:

            - `matplotlib.scale.LinearScale`
            - `matplotlib.scale.LogScale`
            - `matplotlib.scale.SymmetricalLogScale`
            - `matplotlib.scale.LogitScale`


        Notes
        -----
        By default, Matplotlib supports the above mentioned scales.
        Additionally, custom scales may be registered using
        `matplotlib.scale.register_scale`. These scales can then also
        be used here.

    yticks(ticks=None, labels=None, **kwargs)
        Get or set the current tick locations and labels of the y-axis.

        Call signatures::

            locs, labels = yticks()            # Get locations and labels
            yticks(ticks, [labels], **kwargs)  # Set locations and labels

        Parameters
        ----------
        ticks : array_like
            A list of positions at which ticks should be placed. You can pass an
            empty list to disable yticks.

        labels : array_like, optional
            A list of explicit labels to place at the given *locs*.

        **kwargs
            :class:`.Text` properties can be used to control the appearance of
            the labels.

        Returns
        -------
        locs
            An array of label locations.
        labels
            A list of `.Text` objects.

        Notes
        -----
        Calling this function with no arguments (e.g. ``yticks()``) is the pyplot
        equivalent of calling `~.Axes.get_yticks` and `~.Axes.get_yticklabels` on
        the current axes.
        Calling this function with arguments is the pyplot equivalent of calling
        `~.Axes.set_yticks` and `~.Axes.set_yticklabels` on the current axes.

        Examples
        --------
        Get the current locations and labels:

            >>> locs, labels = yticks()

        Set label locations:

            >>> yticks(np.arange(0, 1, step=0.2))

        Set text labels:

            >>> yticks(np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))

        Set text labels and properties:

            >>> yticks(np.arange(12), calendar.month_name[1:13], rotation=45)

        Disable yticks:

            >>> yticks([])

DATA
    rcParams = RcParams({'_internal.classic_mode': False,
         ...nor.widt...
    rcParamsDefault = RcParams({'_internal.classic_mode': False,
         ...n...
    rcParamsOrig = RcParams({'_internal.classic_mode': False,
         ...nor....

FILE
    c:\program files (x86)\python37-32\lib\site-packages\matplotlib\pyplot.py
```

```
c:\program files (x86)\python37-32\lib\pydoc.py:1053: MatplotlibDeprecationWarning:
The examples.directory rcparam was deprecated in Matplotlib 3.0 and will be removed in 3.2. In the future, exam
ples will be found relative to the 'datapath' directory.
  return cram(stripid(repr(x)), self.maxother)
c:\program files (x86)\python37-32\lib\pydoc.py:1053: MatplotlibDeprecationWarning:
The examples.directory rcparam was deprecated in Matplotlib 3.0 and will be removed in 3.2. In the future, exam
ples will be found relative to the 'datapath' directory.
  return cram(stripid(repr(x)), self.maxother)
c:\program files (x86)\python37-32\lib\pydoc.py:1053: MatplotlibDeprecationWarning:
The examples.directory rcparam was deprecated in Matplotlib 3.0 and will be removed in 3.2. In the future, exam
ples will be found relative to the 'datapath' directory.
  return cram(stripid(repr(x)), self.maxother)
```

In [66]:

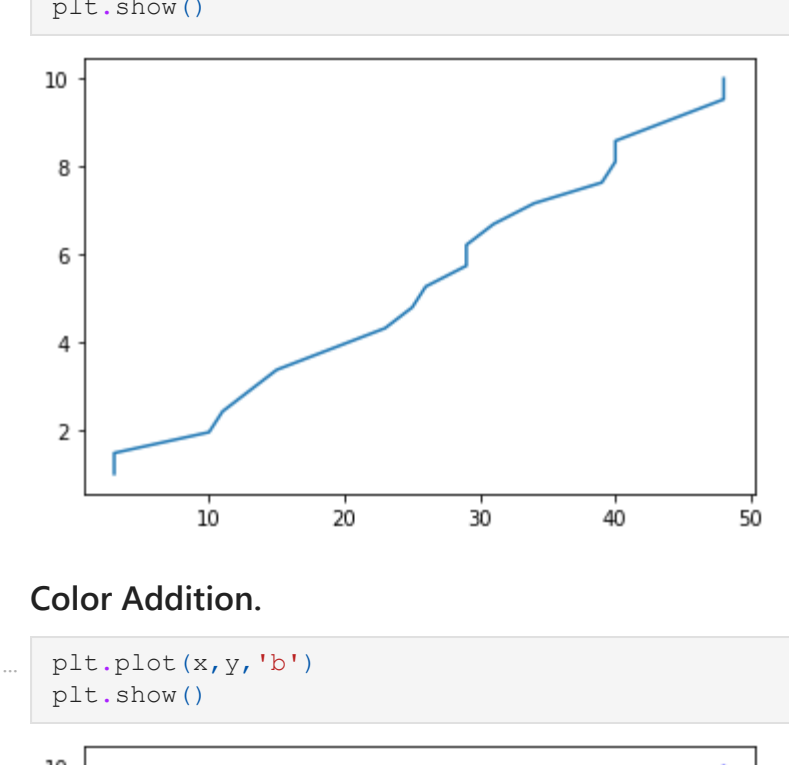
```
dir(plt)
```

```
Out[66]: ['Annotation',
 'Arrow',
 'Artist',
 'AutoLocator',
 'Axes',
 'Button',
 'Circle',
 'Figure',
 'FigureCanvasBase',
 'FixedFormatter',
 'FixedLocator',
 'FormatStrFormatter',
 'Formatter',
 'FuncFormatter',
 'GridSpec',
 'IndexLocator',
 'Line2D',
 'LinearLocator',
 'Locator',
 'LogFormatter',
 'LogFormatterExponent',
 'LogFormatterMathtext',
 'LogLocator',
 'MaxNLocator',
 'MultipleLocator',
 'Normalize',
 'NullFormatter',
 'NullLocator',
 'Number',
 'PolarAxes',
 'Polygon',
 'Rectangle',
 'ScalarFormatter',
 'Slider',
 'Subplot',
 'SubplotTool',
 'Text',
 'TickHelper',
 'Widget',
 '_INSTALL_FIG_OBSERVER',
 '_IP_REGISTERED',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 '_auto_draw_if_interactive',
 '_backend_mod',
 '_get_running_interactive_framewo
 '_interactive_bk',
 '_log',
 '_pylab_helpers',
 '_setp',
 '_setup_pyplot_info_docstrings',
 '_show',
 'acorr',
 'angle_spectrum',
 'annotate',
 'arrow',
 'autoscale',
 'autumn',
 'axes',
 'axhline',
 'axhspan',
 'axis',
 'axvline',
 'axvspan',
 'bar',
 'barbs',
 'barh',
 'bone',
 'box',
 'boxplot',
 'broken_barh',
 'cbook',
 'cla',
 'clabel',
 'clf',
 'clim',
 'close',
 'cm',
 'cohere',
 'colorbar',
 'colormaps',
 'connect',
 'contour',
 'contourf',
 'cool',
 'copper',
 'csd',
 'cycler',
 'dedent',
 'delaxes',
 'deprecated',
 'disconnect',
 'docstring',
 'draw',
 'draw_all',
 'draw_if_interactive',
 'errorbar',
 'eventplot',
 'figaspect',
 'figimage',
 'figlegend',
 'fignum_exists',
 'figtext',
 'figure',
 'fill',
 'fill_between',
 'fill_betweenx',
 'findobj',
 'flag',
 'functools',
 'gca',
 'gcf',
 'gci',
 'get',
 'get_backend',
 'get_cmap',
 'get_current_fig_manager',
 'get_figlabels',
 'get_fignums',
 'get_plot_commands',
 'get_scale_docs',
 'get_scale_names',
 'getp',
 'ginput',
 'gray',
 'grid',
 'hexbin',
 'hist',
 'hist2d',
 'hlines',
 'hot',
 'hsv',
 'importlib',
 'imread',
 'imsave',
 'imshow',
 'inferno',
 'inspect',
 'install_repl_displayhook',
 'interactive',
 'ioff',
 'ion',
 'isinteractive',
 'jet',
 'legend',
 'locator_params',
 'logging',
 'loglog',
 'magma',
 'magnitude_spectrum',
 'margins',
 'matplotlib',
 'matshow',
 'minorticks_off',
 'minorticks_on',
 'mlab',
 'new_figure_manager',
 'nipy_spectral',
 'np',
 'pause',
 'pcolor',
 'pcolormesh',
 'phase_spectrum',
 'pie',
 'pink',
 'plasma',
 'plot',
 'plot_date',
 'plotfile',
 'plotting',
 'polar',
 'prism',
 'psd',
 'pylab_setup',
 'quiver',
 'quiverkey',
 'rc',
 'rcParams',
 'rcParamsDefault',
 'rcParamsOrig',
 'rc_context',
 'rcdefaults',
 'rcsetup',
 're',
 'register_cmap',
 'rgrids',
 'savefig',
 'sca',
 'scatter',
 'sci',
 'semilogx',
 'semilogy',
 'set_cmap',
 'set_loglevel',
 'setp',
 'show',
 'silent_list',
 'specgram',
 'spring',
 'spy',
 'stackplot',
 'stem',
 'step',
 'streamplot',
 'style',
 'subplot',
 'subplot2grid',
 'subplot_tool',
 'subplots',
 'subplots_adjust',
 'summer',
 'suptitle',
 'switch_backend',
 'sys',
 'table',
 'text',
 'thetagrids',
 'tick_params',
 'ticklabel_format',
 'tight_layout',
 'time',
 'title',
 'tricontour',
 'tricontourf',
 'tripcolor',
 'triplot',
 'twinx',
 'twiny',
 'uninstall_repl_displayhook',
 'violinplot',
 'viridis',
 'vlines',
 'waitforbuttonpress',
 'warn_deprecated',
 'winter',
 'xcorr',
 'xkcd',
 'xlabel',
 'xlim',
 'xscale',
 'xticks',
 'ylabel',
 'ylim',
 'yscale',
 'yticks']
```

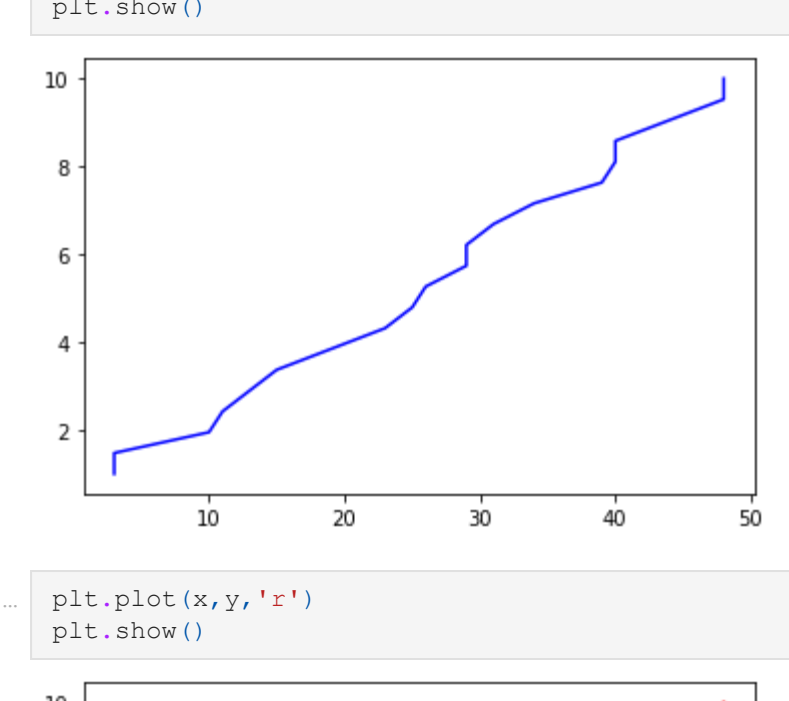
Let's Take example with the help of the arrays.

In [104]:
```
x = np.random.randint(1,50,20)
y=  np.linspace(1,10,20)
```

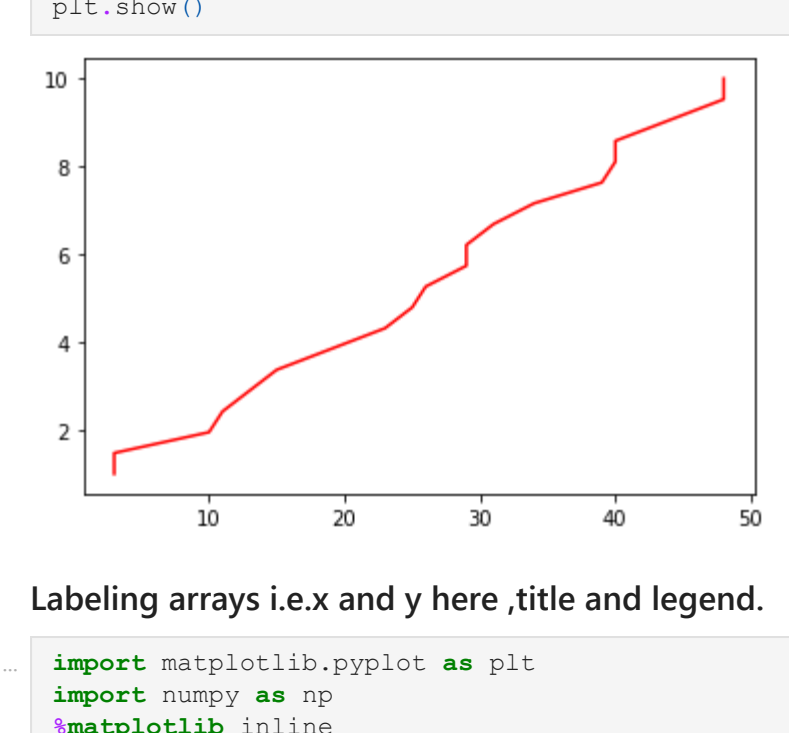
In [105]:
```
plt.plot(x)
plt.show()
```



In [106]:
```
x= np.sort(x)
plt.plot(x)
plt.show()
```

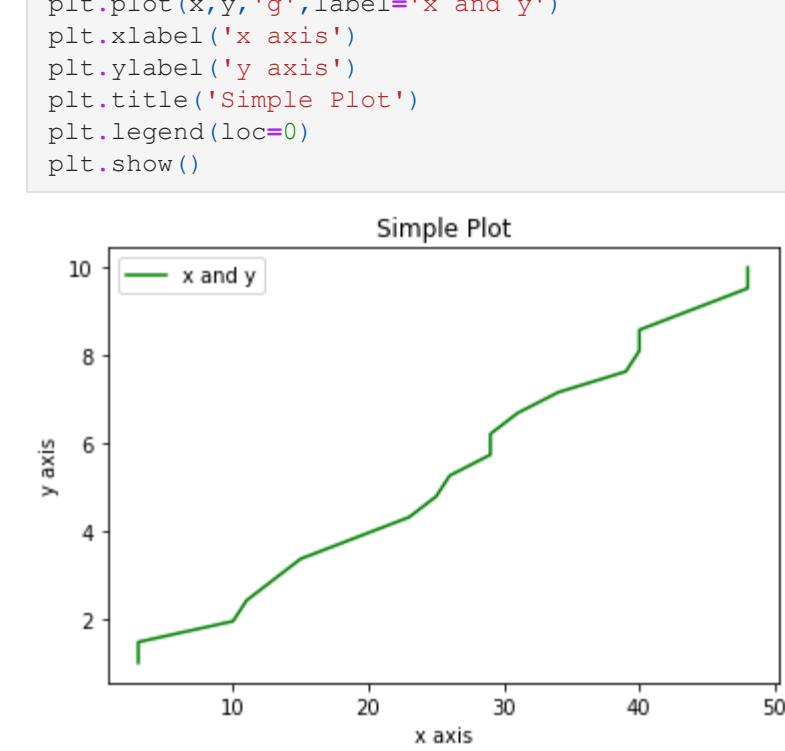


In [110]:
```
plt.plot(x,y)
plt.show()
```

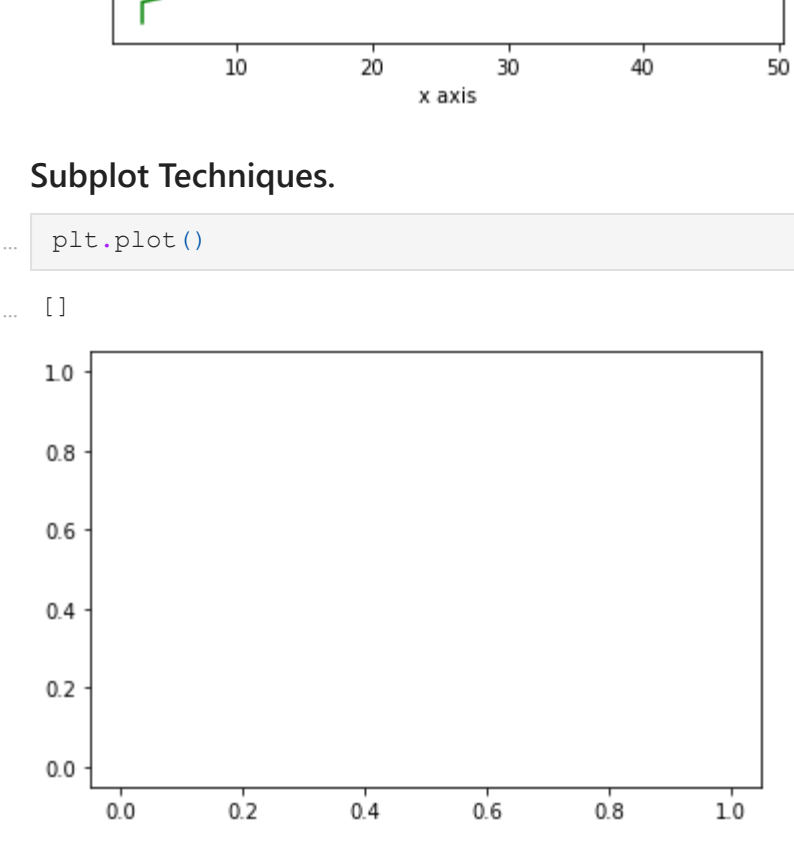


Color Addition.

In [109]:
```
plt.plot(x,y,'b')
plt.show()
```

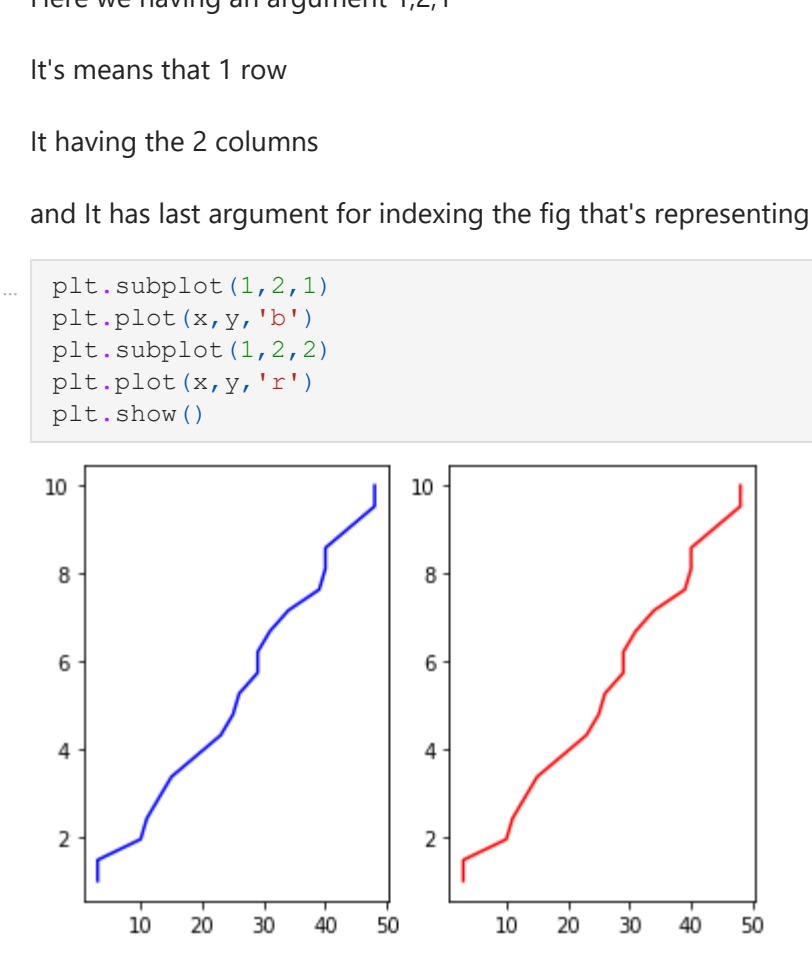


In [112]:
```
plt.plot(x,y,'r')
plt.show()
```



Labeling arrays i.e.x and y here ,title and legend.

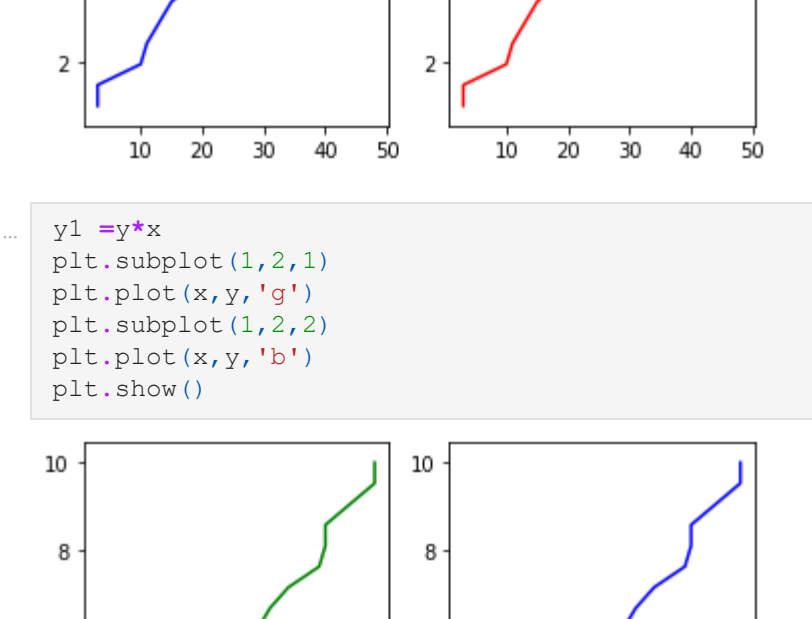
In [117]:
```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
plt.plot(x,y,'g',label='x and y')
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title('Simple Plot')
plt.legend(loc=0)
plt.show()
```



Subplot Techniques.

In [118]:
```
plt.plot()
```

Out[118]: []



Subplot.

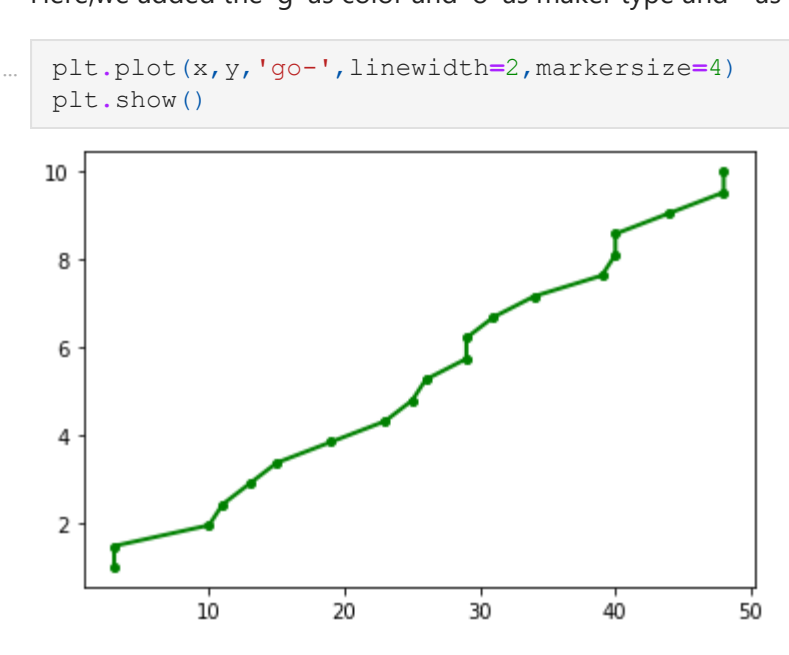
Here we having an argument 1,2,1

It's means that 1 row

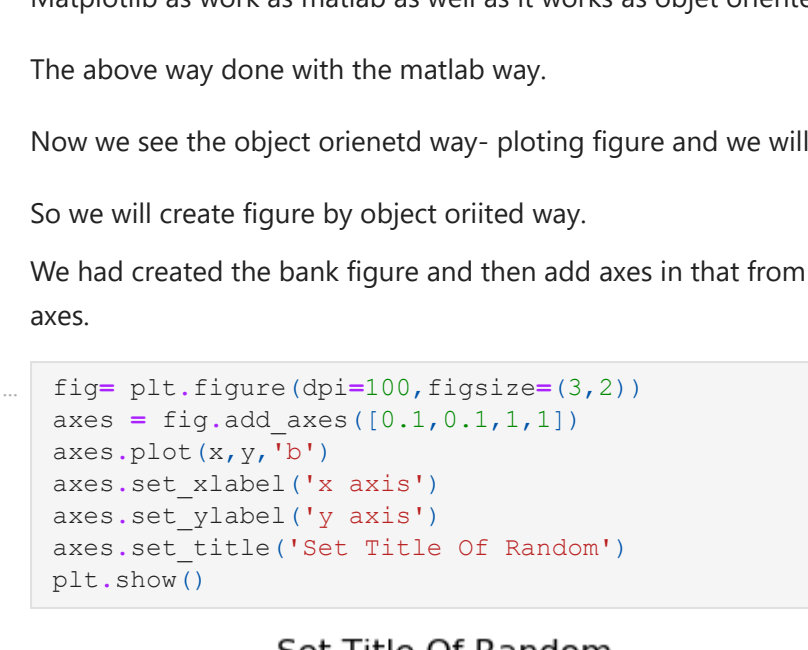
It having the 2 columns

and It has last argument for indexing the fig that's representing how would be the first among the two figures.

In [124]:
```
plt.subplot(1,2,1)
plt.plot(x,y,'b')
plt.subplot(1,2,2)
plt.plot(x,y,'r')
plt.show()
```



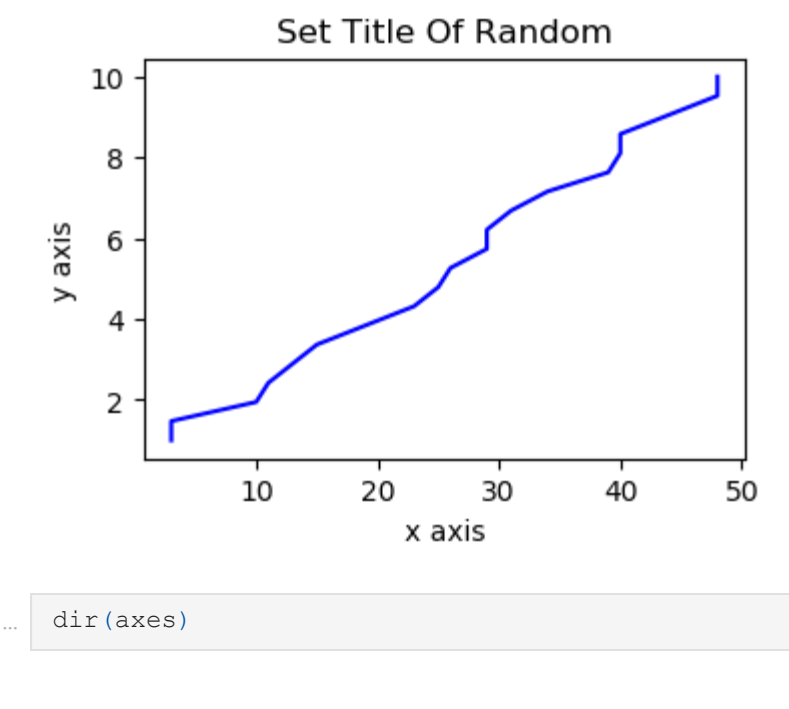
In [130]:
```
y1 =y*x
plt.subplot(1,2,1)
plt.plot(x,y,'g')
plt.subplot(1,2,2)
plt.plot(x,y,'b')
plt.show()
```



Plot with Linewidth,marker,linestyle,Markersize.

Here,we added the 'g' as color and 'o' as maker type and - as linestyle afterwards we had given the markersize ,linewidth parameters.

In [131]:
```
plt.plot(x,y,'go-',linewidth=2,markersize=4)
plt.show()
```



Diffrence Between the Matplotlib & Matlab.

Matplotlib as work as matlab as well as it works as objet oriented language or method.its called to figure object.

The above way done with the matlab way.

Now we see the object orienetd way- ploting figure and we will get the control of statics of that fig.

So we will create figure by object oriited way.

We had created the bank figure and then add axes in that from the four corners. The dimensions [left, bottom, width, height] of the new axes.

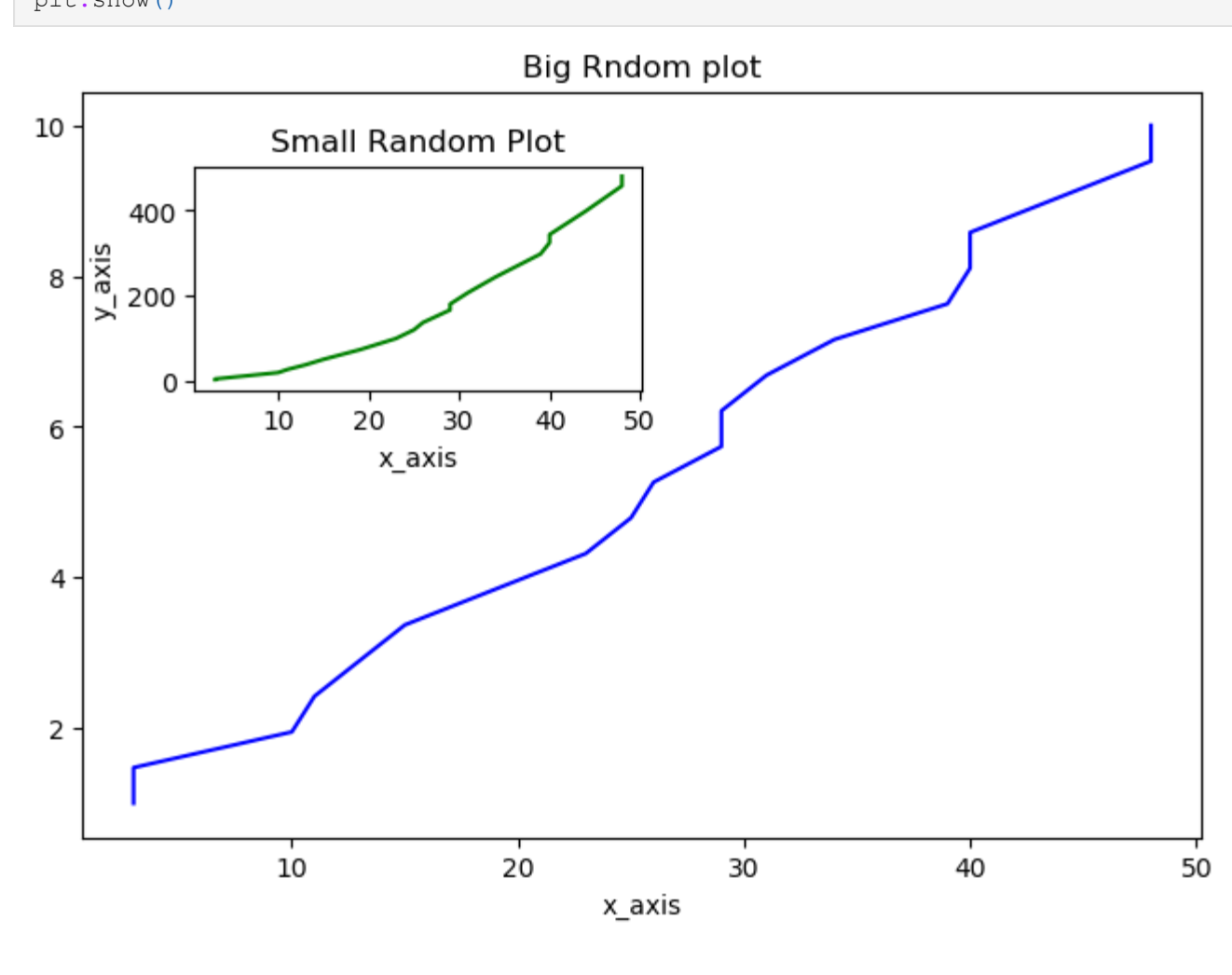
In [143]:
```
fig= plt.figure(dpi=100,figsize=(3,2))
axes = fig.add_axes([0.1,0.1,1,1])
axes.plot(x,y,'b')
axes.set_xlabel('x axis')
axes.set_ylabel('y axis')
axes.set_title('Set Title Of Random')
plt.show()
```


In [144]:
```
dir(axes)
```

```
Out[144… ['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__eq__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__getstate__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__setstate__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 '_add_text',
 '_adjustable',
 '_agg_filter',
 '_alpha',
 '_anchor',
 '_animated',
 '_aspect',
 '_autoscaleXon',
 '_autoscaleYon',
 '_autotitlepos',
 '_axes',
 '_axes_locator',
 '_axisbelow',
 '_clipon',
 '_clippath',
 '_connected',
 '_contains',
 '_convert_dx',
 '_current_image',
 '_facecolor',
 '_frameon',
 '_gci',
 '_gen_axes_patch',
 '_gen_axes_spines',
 '_get_axis_list',
 '_get_clipping_extent_bbox',
 '_get_lines',
 '_get_patches_for_fill',
 '_get_view',
 '_gid',
 '_gridOn',
 '_in_layout',
 '_init_axis',
 '_label',
 '_layoutbox',
 '_left_title',
 '_make_twin_axes',
 '_mouseover',
 '_mouseover_set',
 '_navigate',
 '_navigate_mode',
 '_oid',
 '_on_units_changed',
 '_originalPosition',
 '_parse_scatter_color_args',
 '_path_effects',
 '_pcolorargs',
 '_picker',
 '_position',
 '_poslayoutbox',
 '_process_unit_info',
 '_prop_order',
 '_propobservers',
 '_quiver_units',
 '_rasterization_zorder',
 '_rasterized',
 '_remove_legend',
 '_remove_method',
 '_right_title',
 '_sci',
 '_set_artist_props',
 '_set_gc_clip',
 '_set_lim_and_transforms',
 '_set_position',
 '_set_title_offset_trans',
 '_set_view',
 '_set_view_from_bbox',
 '_shared_x_axes',
 '_shared_y_axes',
 '_sharex',
 '_sharey',
 '_sketch',
 '_snap',
 '_stale',
 '_sticky_edges',
 '_tight',
 '_transform',
 '_transformSet',
 '_twinned_axes',
 '_update_image_limits',
 '_update_line_limits',
 '_update_patch_limits',
 '_update_title_position',
 '_update_transScale',
 '_url',
 '_use_sticky_edges',
 '_validate_converted_limits',
 '_visible',
 '_xaxis_transform',
 '_xcid',
 '_xmargin',
 '_yaxis_transform',
 '_ycid',
 '_ymargin',
 'acorr',
 'add_artist',
 'add_callback',
 'add_child_axes',
 'add_collection',
 'add_container',
 'add_image',
 'add_line',
 'add_patch',
 'add_table',
 'aname',
 'angle_spectrum',
 'annotate',
 'apply_aspect',
 'arrow',
 'artists',
 'autoscale',
 'autoscale_view',
 'axes',
 'axhline',
 'axhspan',
 'axis',
 'axison',
 'axvline',
 'axvspan',
 'bar',
 'barbs',
 'barh',
 'bbox',
 'boxplot',
 'broken_barh',
 'bxp',
 'callbacks',
 'can_pan',
 'can_zoom',
 'child_axes',
 'cla',
 'clabel',
 'clear',
 'clipbox',
 'cohere',
 'collections',
 'containers',
 'contains',
 'contains_point',
 'contour',
 'contourf',
 'convert_xunits',
 'convert_yunits',
 'csd',
 'dataLim',
 'drag_pan',
 'draw',
 'draw_artist',
 'end_pan',
 'errorbar',
 'eventplot',
 'eventson',
 'figure',
 'fill',
 'fill_between',
 'fill_betweenx',
 'findobj',
 'fmt_xdata',
 'fmt_ydata',
 'format_coord',
 'format_cursor_data',
 'format_xdata',
 'format_ydata',
 'get_adjustable',
 'get_agg_filter',
 'get_alpha',
 'get_anchor',
 'get_animated',
 'get_aspect',
 'get_autoscale_on',
 'get_autoscalex_on',
 'get_autoscaley_on',
 'get_axes_locator',
 'get_axisbelow',
 'get_children',
 'get_clip_box',
 'get_clip_on',
 'get_clip_path',
 'get_contains',
 'get_cursor_data',
 'get_data_ratio',
 'get_data_ratio_log',
 'get_default_bbox_extra_artists',
 'get_facecolor',
 'get_fc',
 'get_figure',
 'get_frame_on',
 'get_gid',
 'get_images',
 'get_in_layout',
 'get_label',
 'get_legend',
 'get_legend_handles_labels',
 'get_lines',
 'get_navigate',
 'get_navigate_mode',
 'get_path_effects',
 'get_picker',
 'get_position',
 'get_rasterization_zorder',
 'get_rasterized',
 'get_renderer_cache',
 'get_shared_x_axes',
 'get_shared_y_axes',
 'get_sketch_params',
 'get_snap',
 'get_tightbbox',
 'get_title',
 'get_transform',
 'get_transformed_clip_path_and_affine',
 'get_url',
 'get_visible',
 'get_window_extent',
 'get_xaxis',
 'get_xaxis_text1_transform',
 'get_xaxis_text2_transform',
 'get_xaxis_transform',
 'get_xbound',
 'get_xgridlines',
 'get_xlabel',
 'get_xlim',
 'get_xmajorticklabels',
 'get_xminorticklabels',
 'get_xscale',
 'get_xticklabels',
 'get_xticklines',
 'get_xticks',
 'get_yaxis',
 'get_yaxis_text1_transform',
 'get_yaxis_text2_transform',
 'get_yaxis_transform',
 'get_ybound',
 'get_ygridlines',
 'get_ylabel',
 'get_ylim',
 'get_ymajorticklabels',
 'get_yminorticklabels',
 'get_yscale',
 'get_yticklabels',
 'get_yticklines',
 'get_yticks',
 'get_zorder',
 'grid',
 'has_data',
 'have_units',
 'hexbin',
 'hist',
 'hist2d',
 'hlines',
 'ignore_existing_data_limits',
 'images',
 'imshow',
 'in_axes',
 'indicate_inset',
 'indicate_inset_zoom',
 'inset_axes',
 'invert_xaxis',
 'invert_yaxis',
 'is_transform_set',
 'legend',
 'legend_',
 'lines',
 'locator_params',
 'loglog',
 'magnitude_spectrum',
 'margins',
 'matshow',
 'minorticks_off',
 'minorticks_on',
 'mouseover',
 'mouseover_set',
 'name',
 'patch',
 'patches',
 'pchanged',
 'pcolor',
 'pcolorfast',
 'pcolormesh',
 'phase_spectrum',
 'pick',
 'pickable',
 'pie',
 'plot',
 'plot_date',
 'properties',
 'psd',
 'quiver',
 'quiverkey',
 'redraw_in_frame',
 'relim',
 'remove',
 'remove_callback',
 'reset_position',
 'scatter',
 'secondary_xaxis',
 'secondary_yaxis',
 'semilogx',
 'semilogy',
 'set',
 'set_adjustable',
 'set_agg_filter',
 'set_alpha',
 'set_anchor',
 'set_animated',
 'set_aspect',
 'set_autoscale_on',
 'set_autoscalex_on',
 'set_autoscaley_on',
 'set_axes_locator',
 'set_axis_off',
 'set_axis_on',
 'set_axisbelow',
 'set_clip_box',
 'set_clip_on',
 'set_clip_path',
 'set_contains',
 'set_facecolor',
 'set_fc',
 'set_figure',
 'set_frame_on',
 'set_gid',
 'set_in_layout',
 'set_label',
 'set_navigate',
 'set_navigate_mode',
 'set_path_effects',
 'set_picker',
 'set_position',
 'set_prop_cycle',
 'set_rasterization_zorder',
 'set_rasterized',
 'set_sketch_params',
 'set_snap',
 'set_title',
 'set_transform',
 'set_url',
 'set_visible',
 'set_xbound',
 'set_xlabel',
 'set_xlim',
 'set_xmargin',
 'set_xscale',
 'set_xticklabels',
 'set_xticks',
 'set_ybound',
 'set_ylabel',
 'set_ylim',
 'set_ymargin',
 'set_yscale',
 'set_yticklabels',
 'set_yticks',
 'set_zorder',
 'specgram',
 'spines',
 'spy',
 'stackplot',
 'stale',
 'stale_callback',
 'start_pan',
 'stem',
 'step',
 'sticky_edges',
 'streamplot',
 'table',
 'tables',
 'text',
 'texts',
 'tick_params',
 'ticklabel_format',
 'title',
 'titleOffsetTrans',
 'transAxes',
 'transData',
 'transLimits',
 'transScale',
 'tricontour',
 'tricontourf',
 'tripcolor',
 'triplot',
 'twinx',
 'twiny',
 'update',
 'update_datalim',
 'update_datalim_bounds',
 'update_from',
 'use_sticky_edges',
 'viewLim',
 'violin',
 'violinplot',
 'vlines',
 'xaxis',
 'xaxis_date',
 'xaxis_inverted',
 'xcorr',
 'yaxis',
 'yaxis_date',
 'yaxis_inverted',
 'zorder']
```

Plot inside Plot .

```
In [149… fig = plt.figure(dpi=100)
ax1= fig.add_axes([0,0,1,1])
ax2 = fig.add_axes([0.1,0.6,0.4,0.3])
ax1.plot(x,y,'b')
ax1.set_xlabel('x_axis')
ax2.set_ylabel('y_axis')

ax2.plot(x,y1,'g')
ax2.set_xlabel('x_axis')
ax2.set_ylabel('y_axis')
ax1.set_title('Big Rndom plot')
ax2.set_title('Small Random Plot')
plt.show()
```



Subplots Technique.

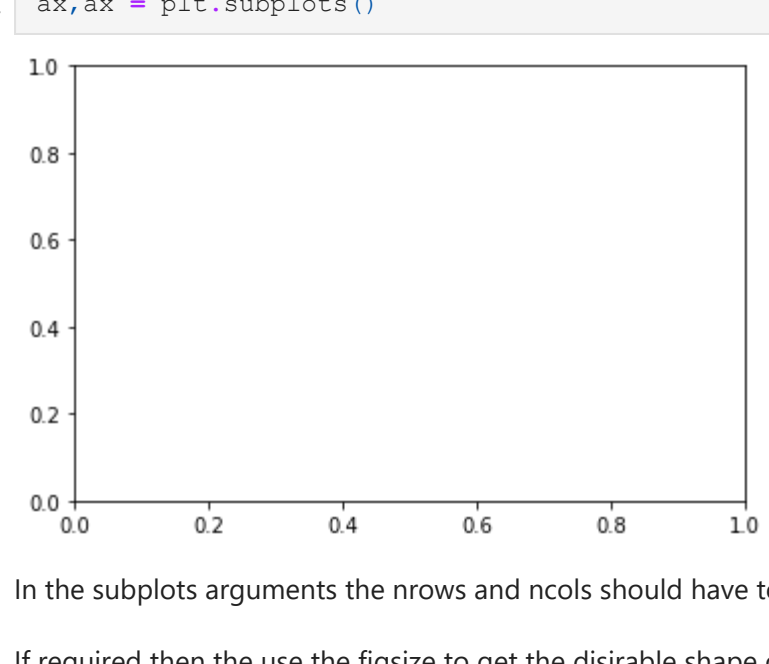
plt.subplots

If we see the last time subplot that was not object oriented and also it wasn't gives the full control over the data.

If we required the 10 times the subplots that time we have write the same command for the 10 times and it becomes tedious.

plt.sublpots() --------gives us the outcomes in two ways that are the axis and figure.

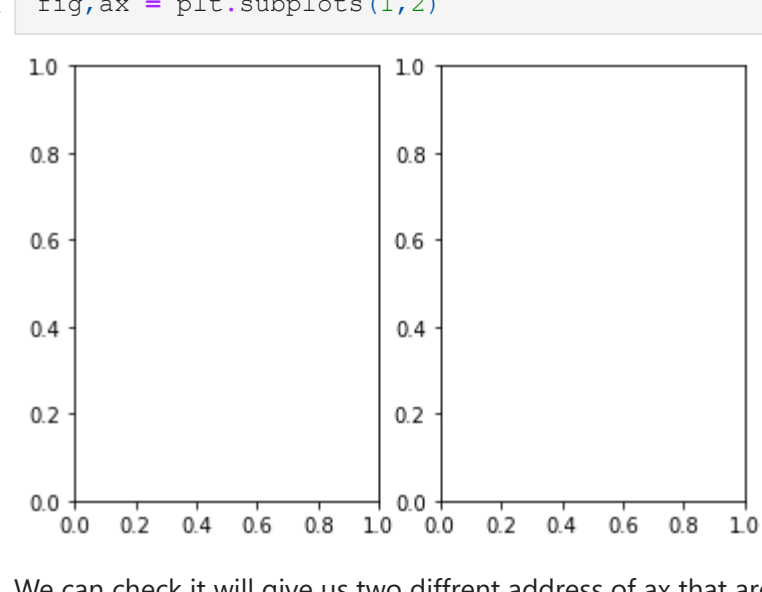
```
In [153… ax,ax = plt.subplots()
```



In the subplots arguments the nrows and ncols should have to insert to get the numbers of rows and columns of the subfigure.

If required then the use the figsize to get the disirable shape of the figure.

```
In [157… fig,ax = plt.subplots(1,2)
```



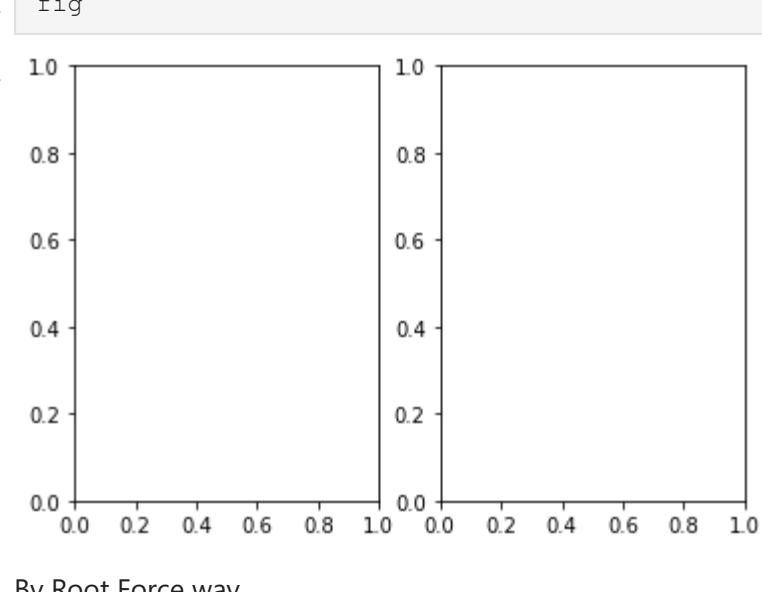
We can check it will give us two diffrent address of ax that are the for the two figures respectively.

```
In [158… ax

Out[158… array([<matplotlib.axes._subplots.AxesSubplot object at 0x11B6E3D0>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x108BD050>],
      dtype=object)
```

It showing the result of the caonvas with the two figures.

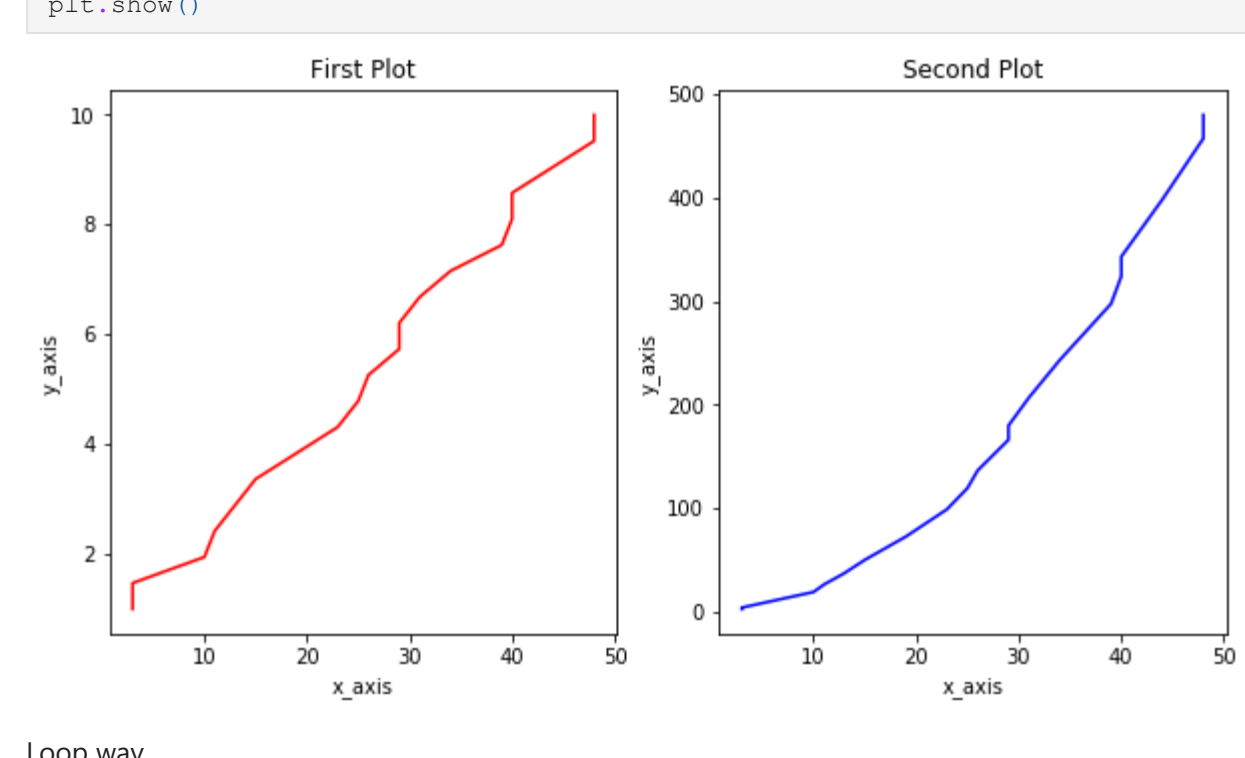
We can call the individual figure with the help of the that is ax[0] and ax[1].

```
In [159… fig

Out[159…
```



By Root Force way.

```
In [162… fig,ax = plt.subplots(1,2,figsize=(10,5))
ax[0].plot(x,y,'r')
ax[0].set_xlabel('x_axis')
ax[0].set_ylabel('y_axis')
ax[0].set_title('First Plot')
ax[1].plot(x,y1,'b')
ax[1].set_xlabel('x_axis')
ax[1].set_ylabel('y_axis')
ax[1].set_title('Second Plot')
plt.show()
```

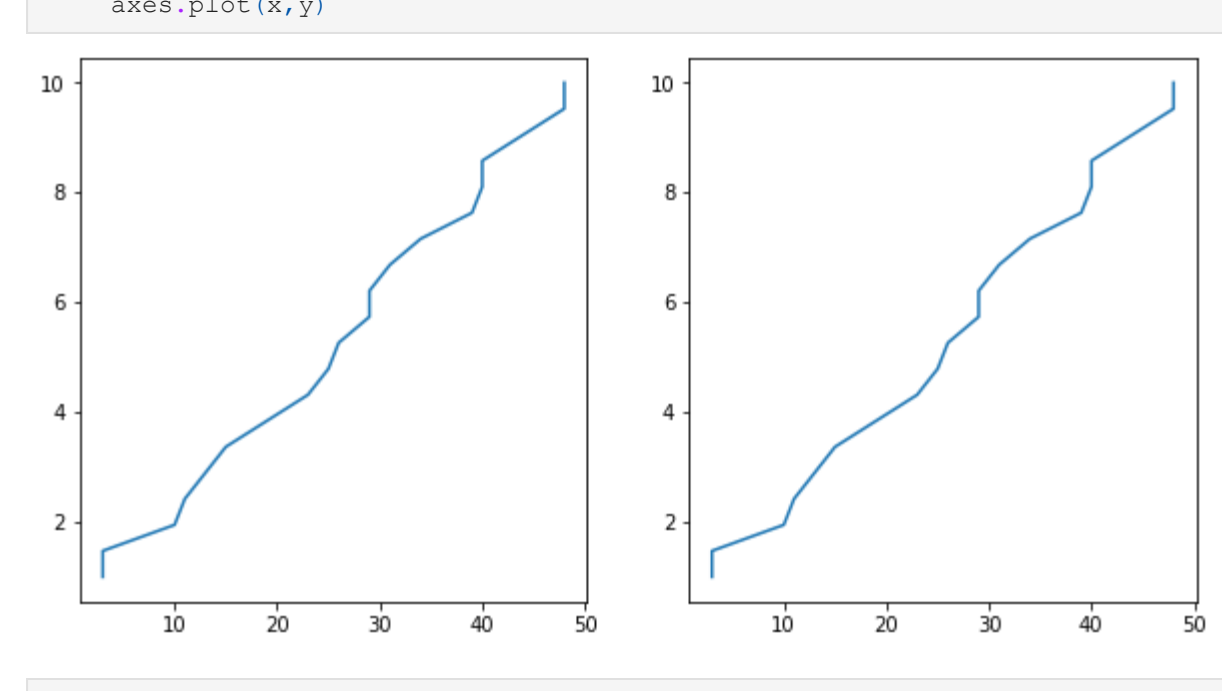


Loop way,

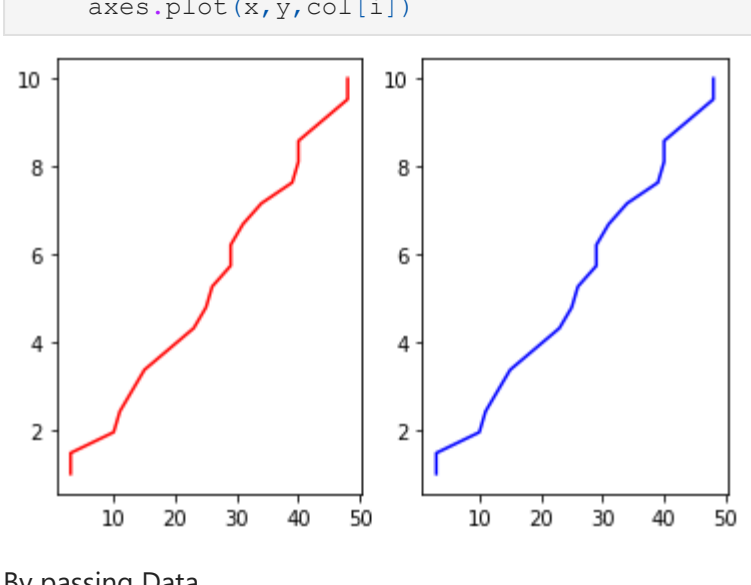
Here if we tried to get the new colour that time we both colour will change.

That's why we need to do something for that.

```
In [165… fig,ax = plt.subplots(1,2,figsize=(10,5))

for axes in ax:
    axes.plot(x,y)
```

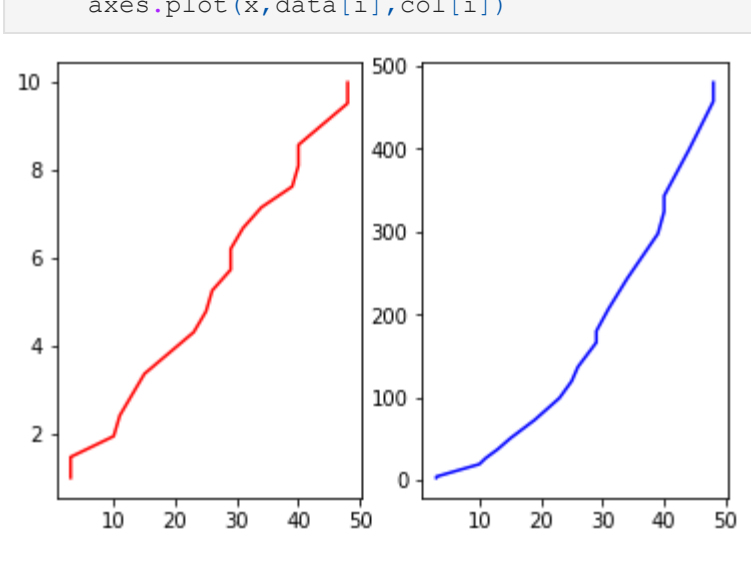


```
In [179… import matplotlib.pyplot as plt

col = ['r','b']

fig,ax=plt.subplots(1,2)

for i,axes in enumerate(ax):
    axes.plot(x,y,col[i])
```



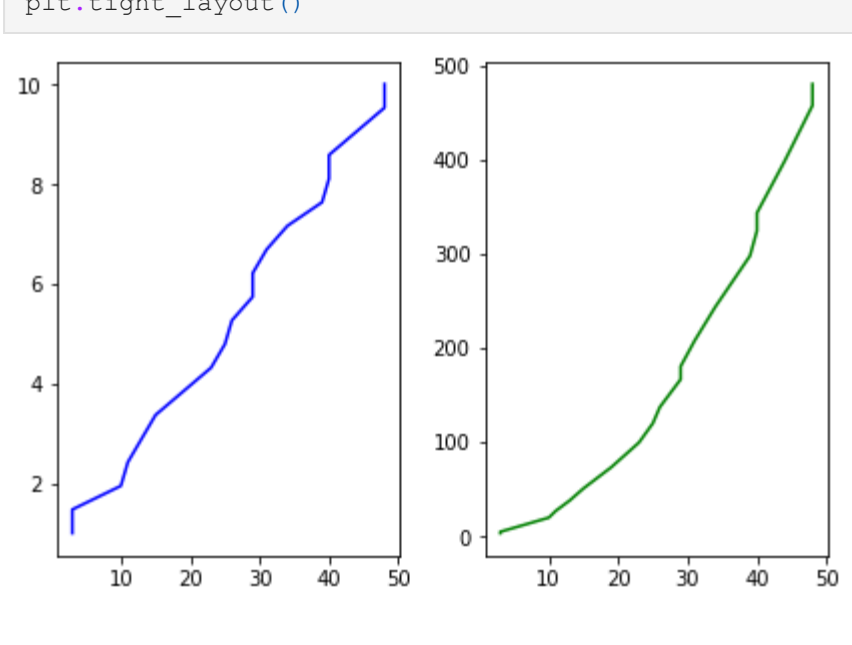
By passing Data,

```
In [180… col=['r','b']
data =[y,y1]

fig,ax = plt.subplots(1,2)

for i,axes in enumerate(ax):
    axes.plot(x,data[i],col[i])
```



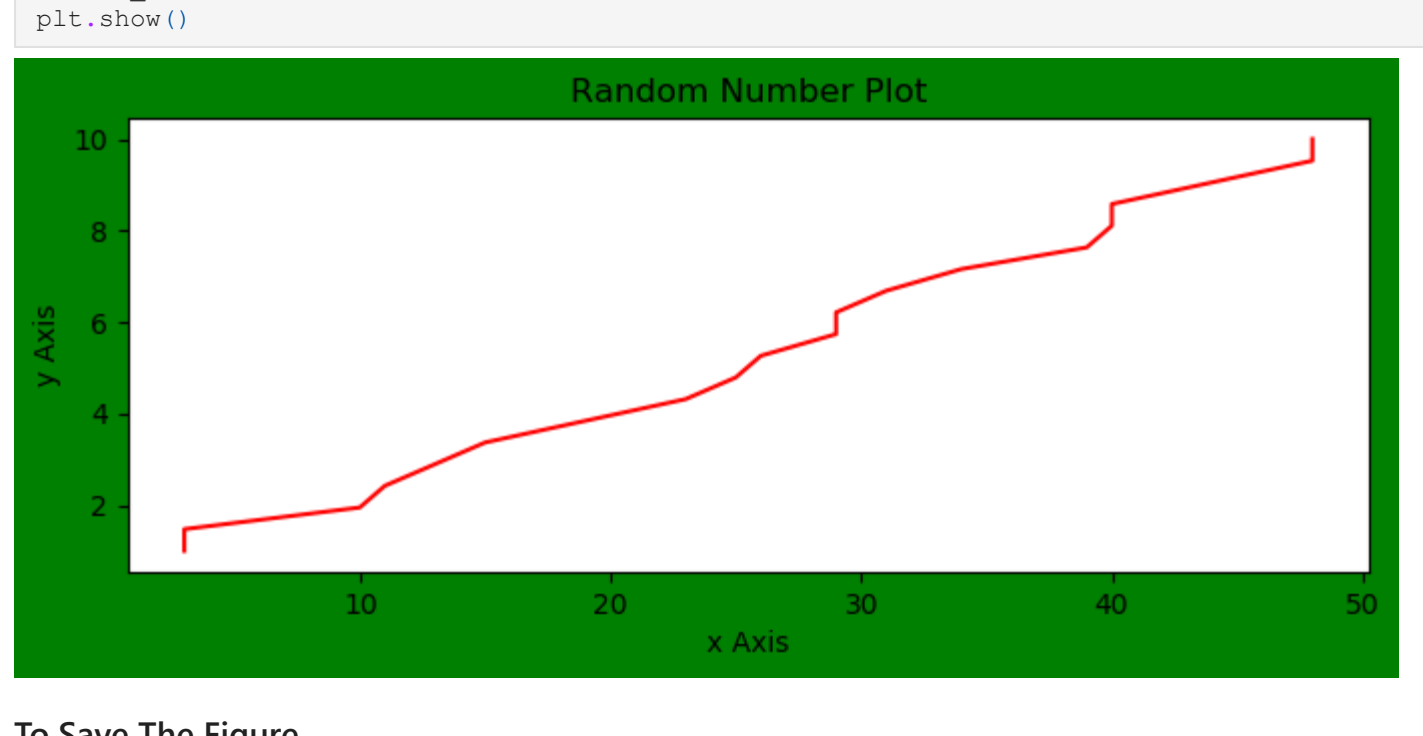
To Avoid the Overlapping of two difftent plot or figure by using tight_layout().

```
In [182… col = ['b','g']
data = [y,y1]

fig,ax = plt.subplots(1,2)

for i,axes in enumerate(ax):
    axes.plot(x,data[i],col[i])
plt.tight_layout()
```



Use the figuresize,facecolor and dpi (dots per inch).

```
In [196… fig,ax = plt.subplots(figsize=(8,3),dpi=100,facecolor='g')
ax.plot(x,y,'r')
ax.set_xlabel('x Axis')
ax.set_ylabel('y Axis')
ax.set_title('Random Number Plot')
plt.show()
```

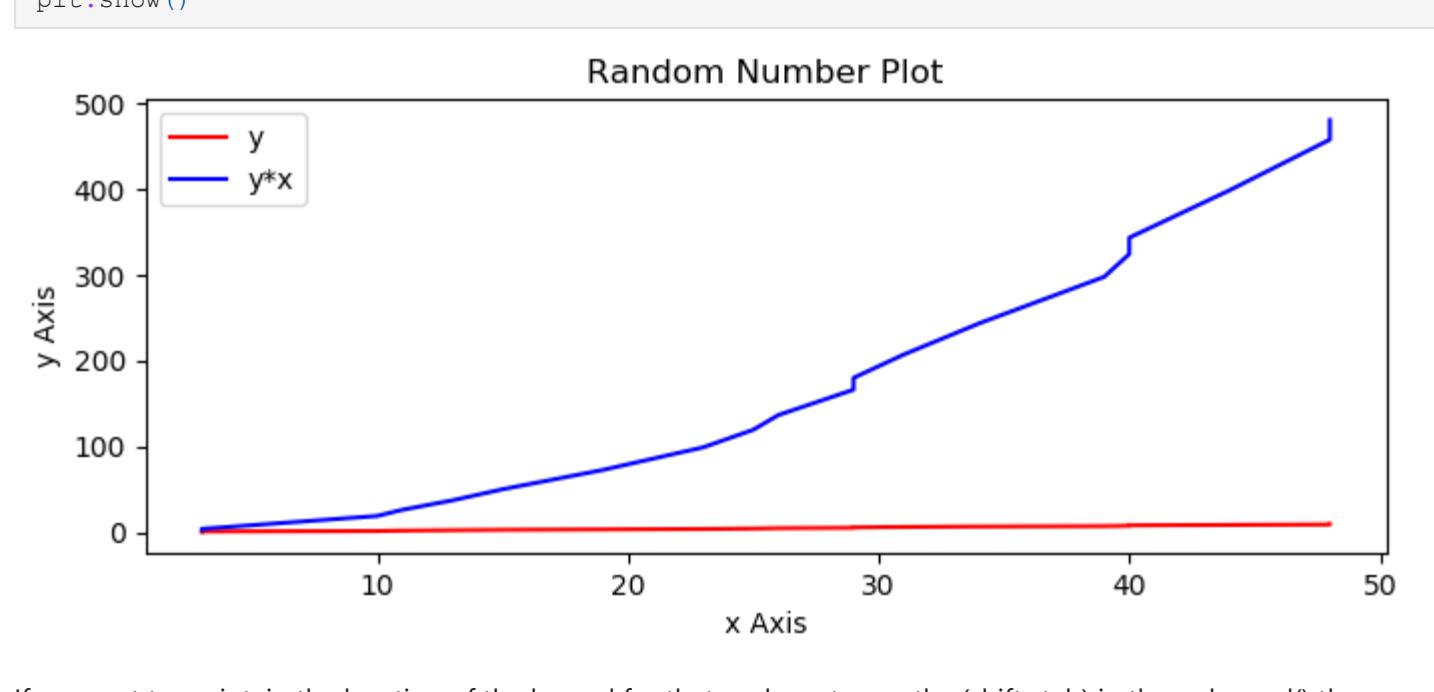


To Save The Figure.

```
In [197… fig.savefig('random plot.png')
```

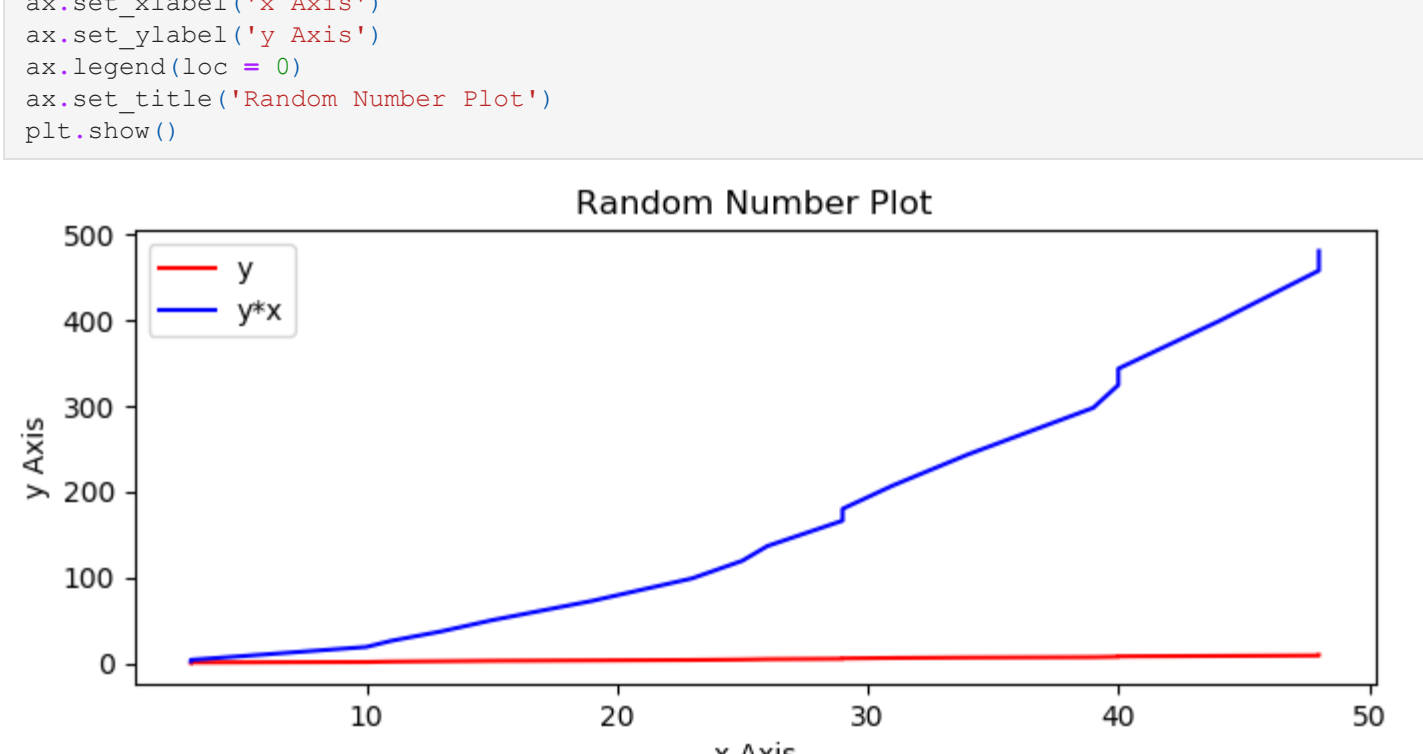
Legend.

```
In [199… fig,ax= plt.subplots(figsize= (8,3),dpi = 100)
ax.plot(x,y,'r',label= 'y')#we can use in this way
ax.plot(x,y1,'b',label='y*x') #or in this way also
ax.set_xlabel('x Axis')
ax.set_ylabel('y Axis')
ax.legend()
ax.set_title('Random Number Plot')
plt.show()
```



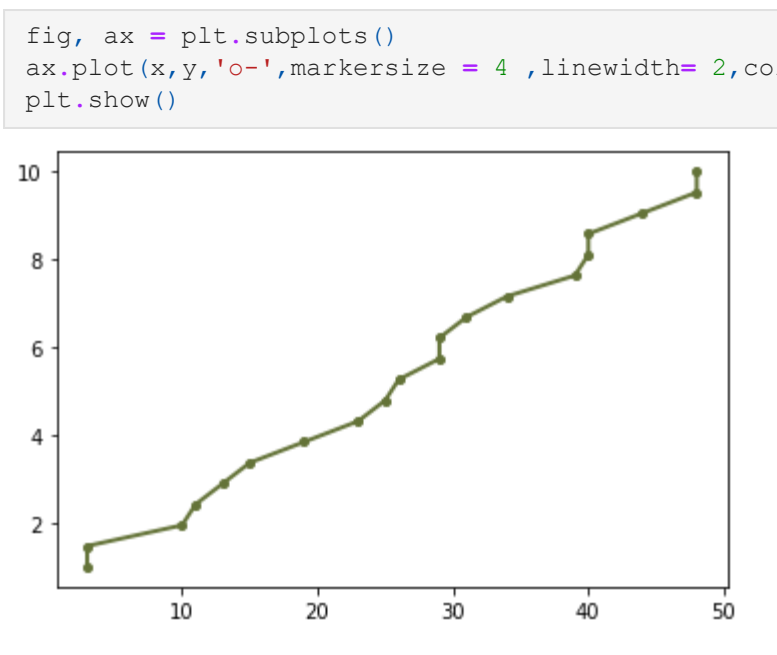
If we want to maintain the location of the legend for that we have to use the (shift+tab) in the ax.legend() then we will find the best place to maintain this legend in plot.

```
In [201… fig,ax= plt.subplots(figsize= (8,3),dpi = 100)
ax.plot(x,y,'r',label= 'y')#we can use in this way
ax.plot(x,y1,'b',label='y*x') #or in this way also
ax.set_xlabel('x Axis')
ax.set_ylabel('y Axis')
ax.legend(loc = 0)
ax.set_title('Random Number Plot')
plt.show()
```



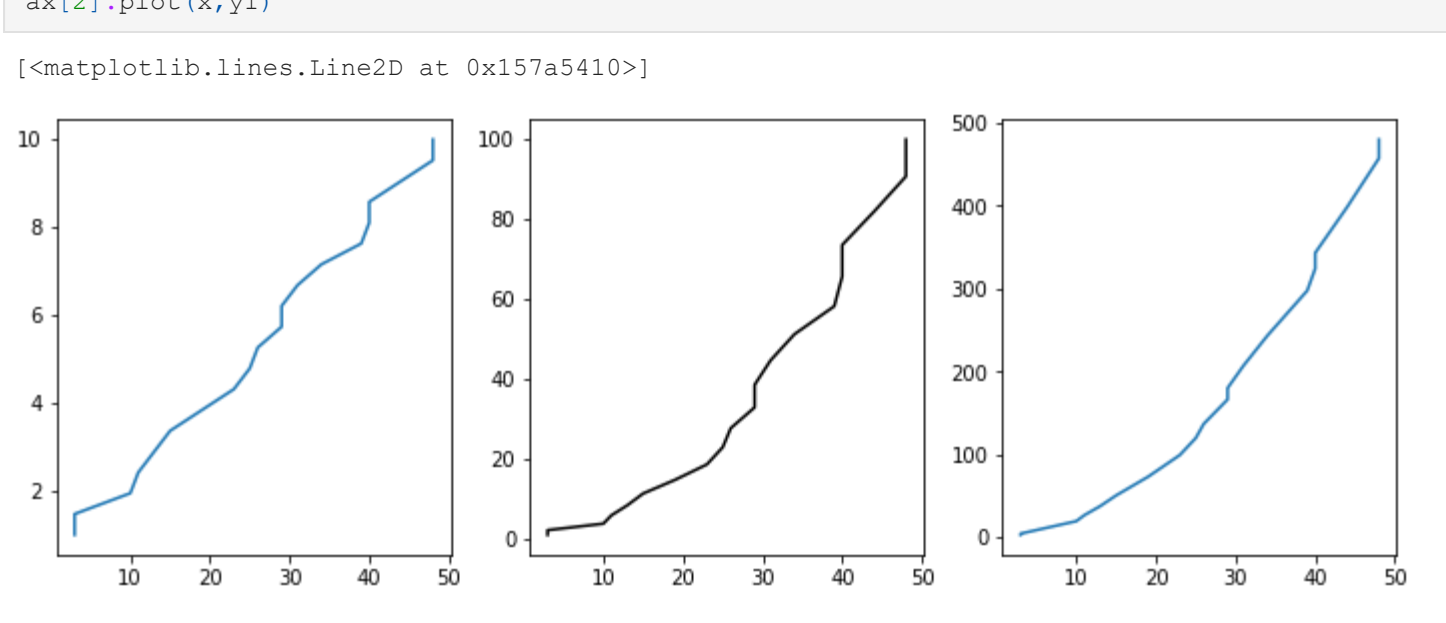
Hex Color

```
In [202… fig, ax = plt.subplots()
ax.plot(x,y,'o-',markersize = 4 ,linewidth= 2,color ='#647338')
plt.show()
```



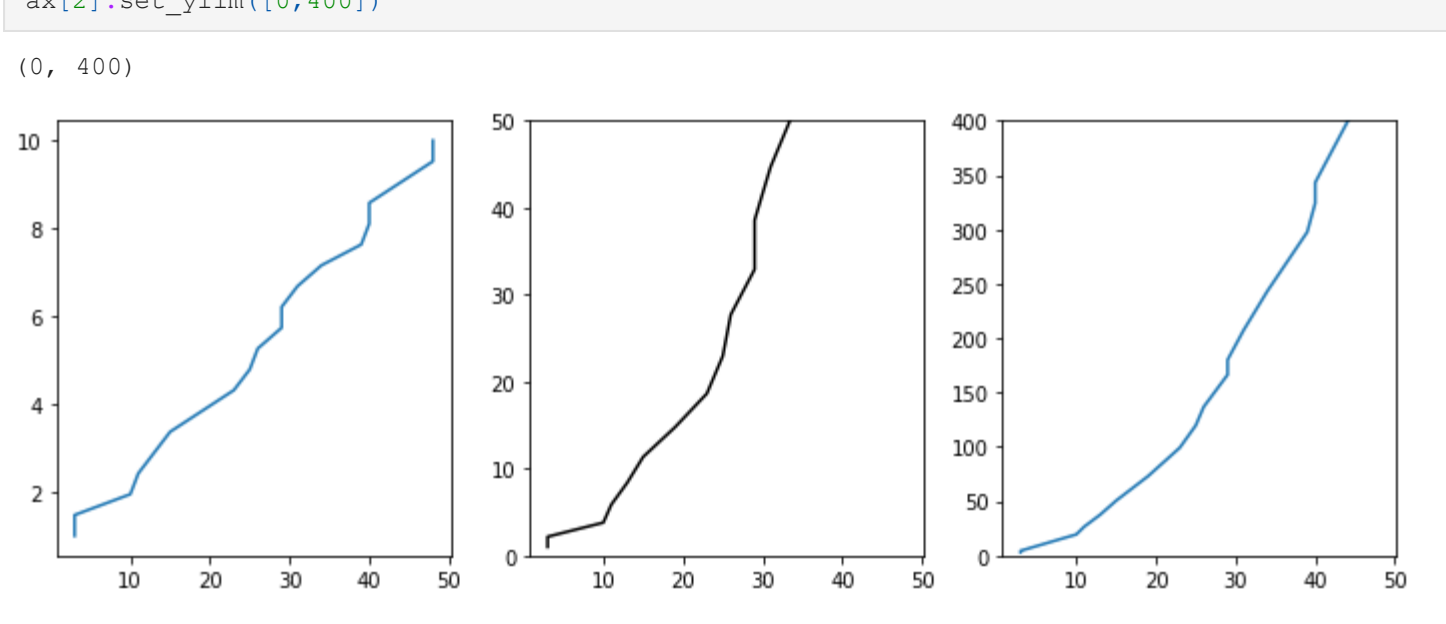
Plot-Range By Putting the limits on the axis.

```
In [207… fig,ax = plt.subplots(1,3,figsize=(12,4))

ax[0].plot(x,y)
ax[1].plot(x,y**2,'k')
ax[2].plot(x,y1)

Out[207… [<matplotlib.lines.Line2D at 0x157a5410>]
```



Put The Limit.

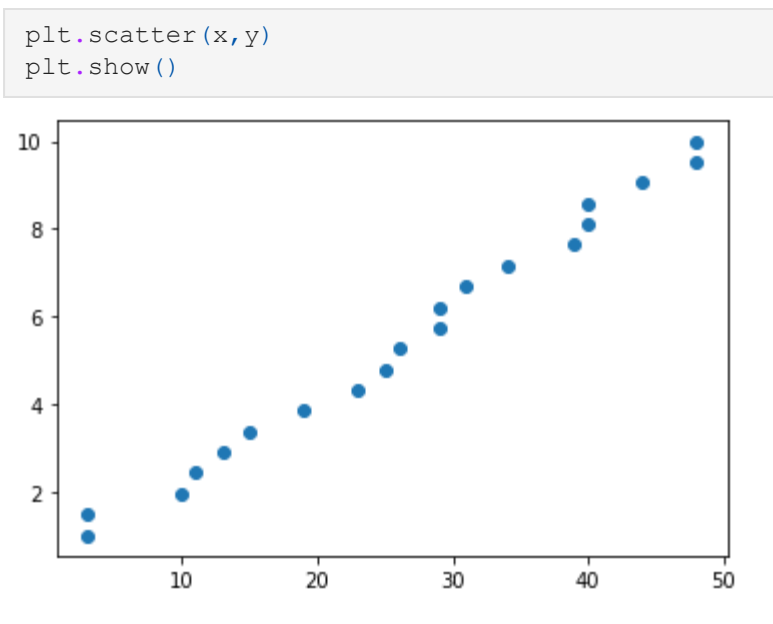
```
In [208… fig,ax = plt.subplots(1,3,figsize=(12,4))

ax[0].plot(x,y)
ax[1].plot(x,y**2,'k')


ax[1].set_ylim([0,50])
ax[2].plot(x,y1)
ax[2].set_ylim([0,400])

Out[208… (0, 400)
```



Other Types of the Plots.

1.Scatterplot.

```
In [209… plt.scatter(x,y)
plt.show()
```

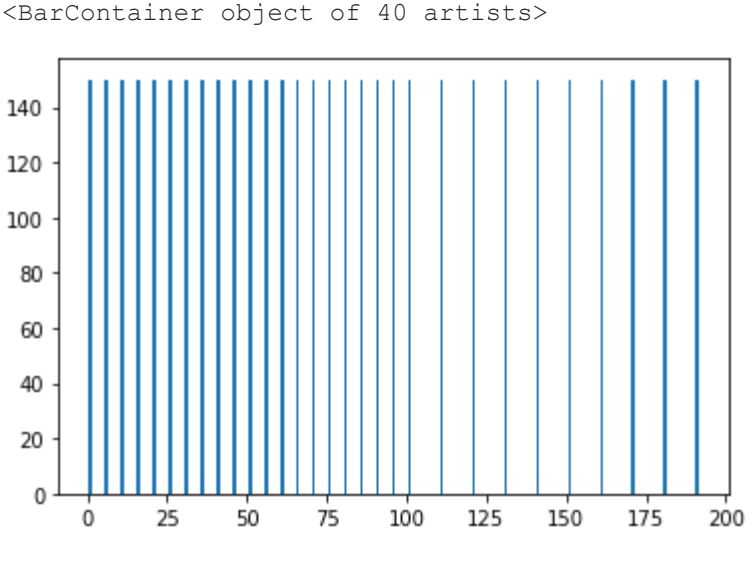


2.Barplot.

```
In [214… p = list(range(1,100,5))
p2 =list(range(1,200,10))
```

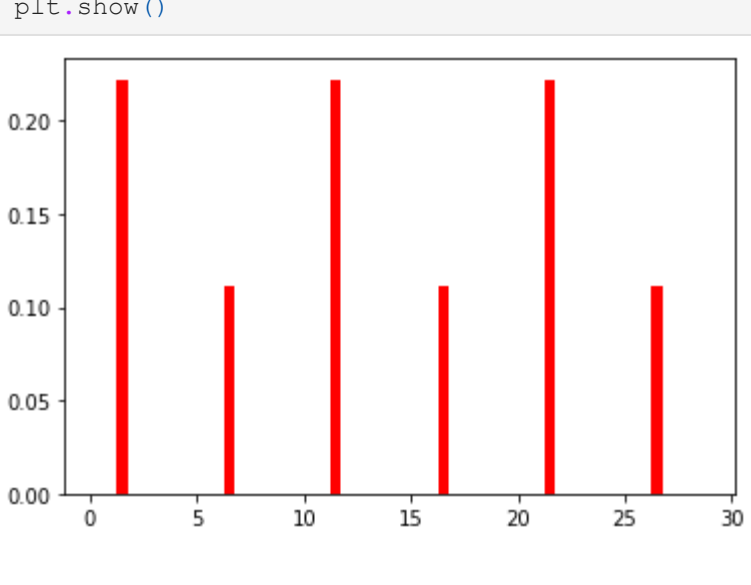
```
In [217… data = p+p2

plt.bar(data,  height=150,width=0.9)
```

```
Out[217…] <BarContainer object of 40 artists>
```



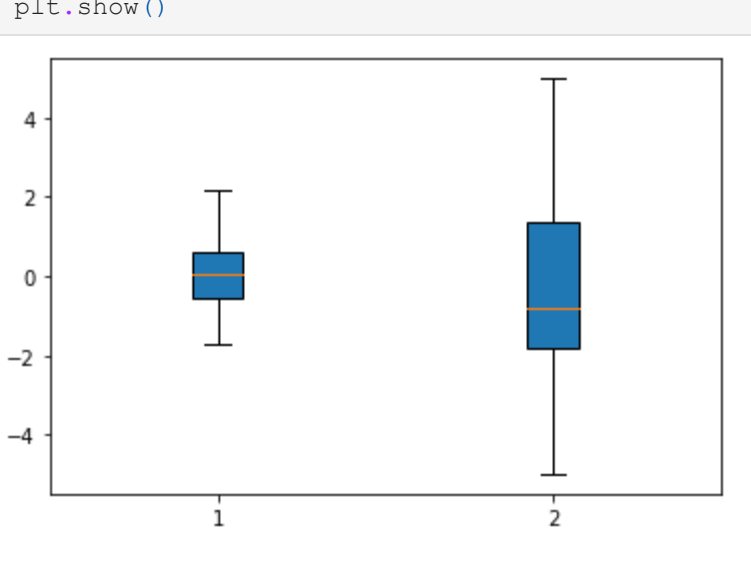
Histogram Plot.

```
In [225…] plt.hist(data,bins=range(0,30),density=True,rwidth=0.5,color='r')
plt.show()
```



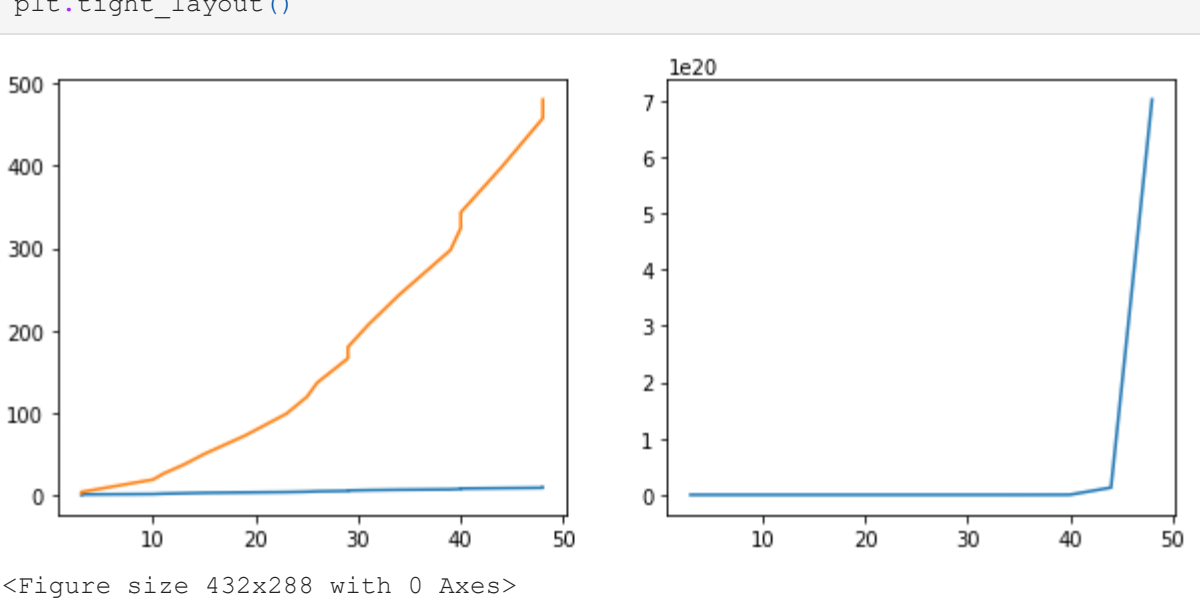
Box Plot.

```
In [231…] data =[np.random.normal(0,std,50) for std in range(1,3)]
plt.boxplot(data,vert=True,patch_artist=True)
plt.show()
```



Log_Axis Plot.

```
In [233…] fig,ax = plt.subplots(1,2,figsize=(10,4))
ax[0].plot(x,y,x,y1)
ax[1].plot(x,np.exp(x))
plt.show()
plt.tight_layout()
```

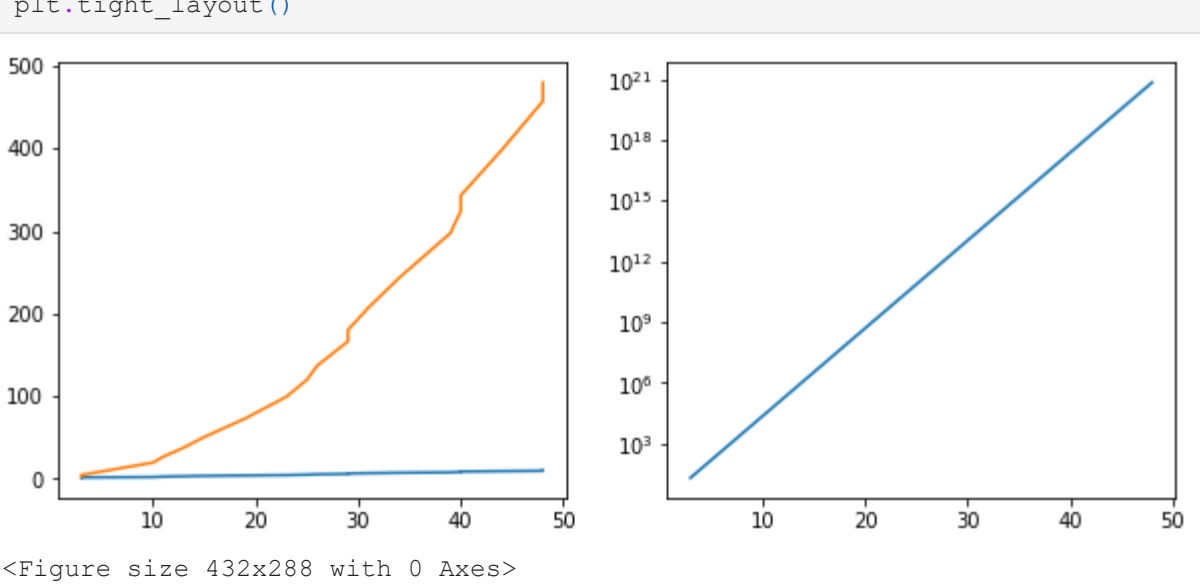


```
<Figure size 432x288 with 0 Axes>
```

Set_Scale

If we check that(y =f(exp) is based on the exponential scale so how we can change the from exponential scale to log sacle of this graph.

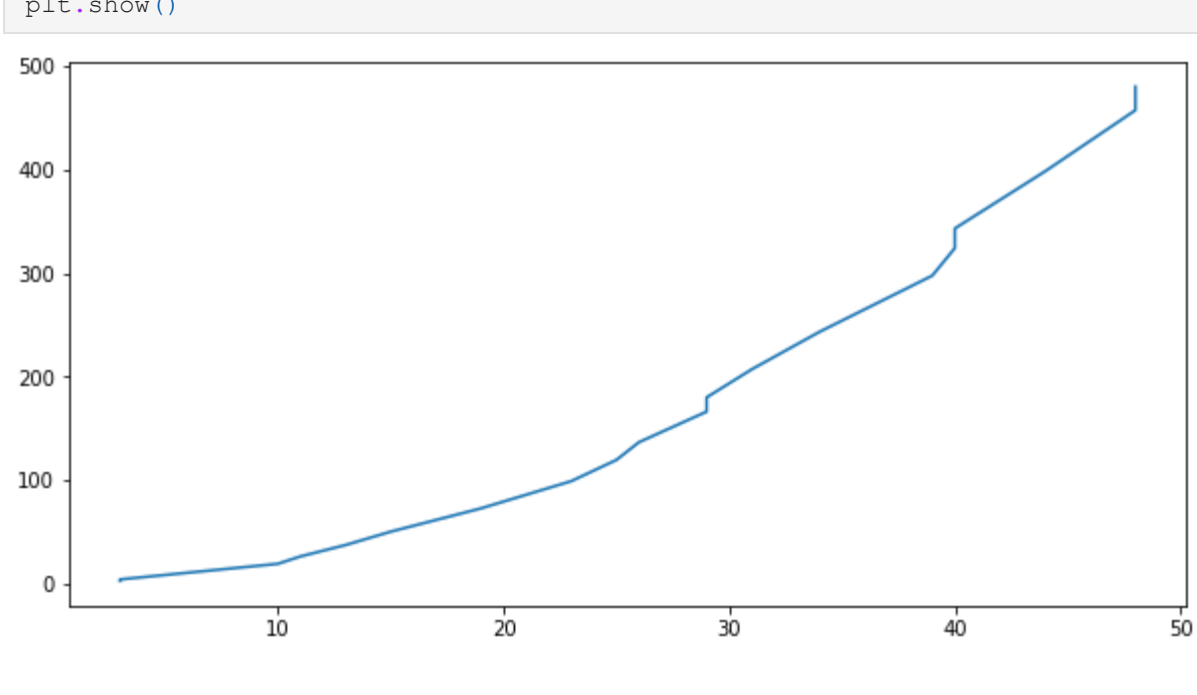
```
In [234…] fig,ax = plt.subplots(1,2,figsize=(10,4))
ax[0].plot(x,y,x,y1)
ax[1].plot(x,np.exp(x))
ax[1].set_yscale('log')
plt.show()
plt.tight_layout()
```



```
<Figure size 432x288 with 0 Axes>
```

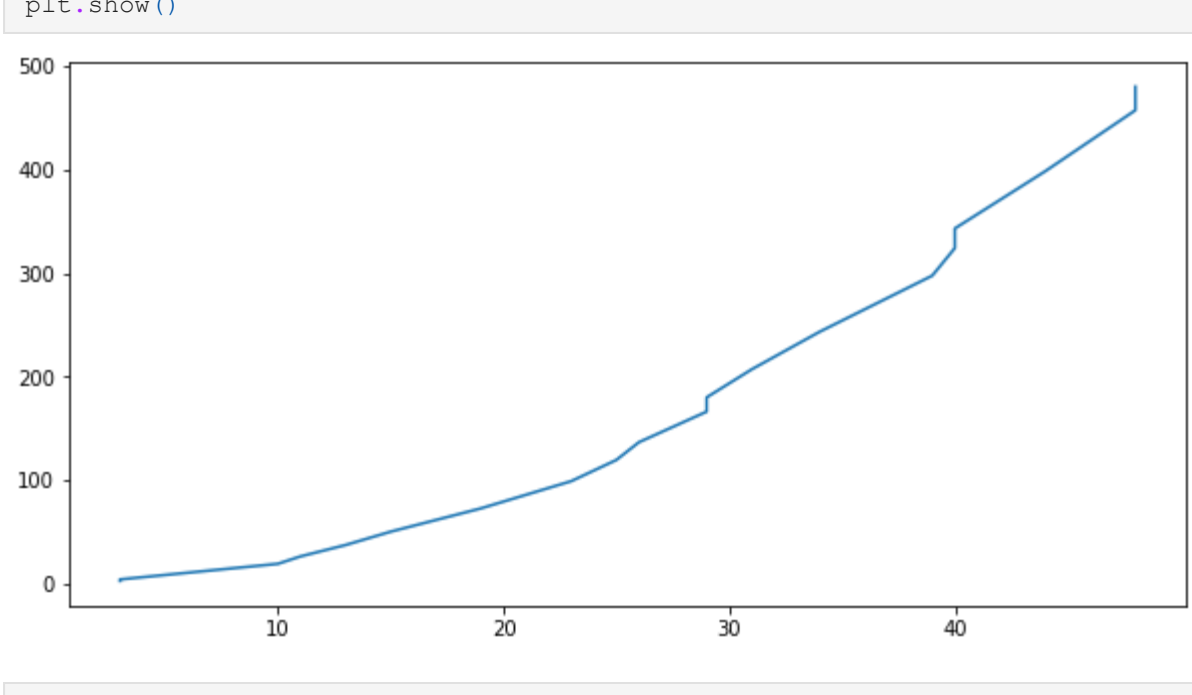
Ticks,labeling of ticks and evrything about the ticks.

```
In [235… fig,ax=plt.subplots(figsize=(10,5))
ax.plot(x,y1)
plt.show()
```

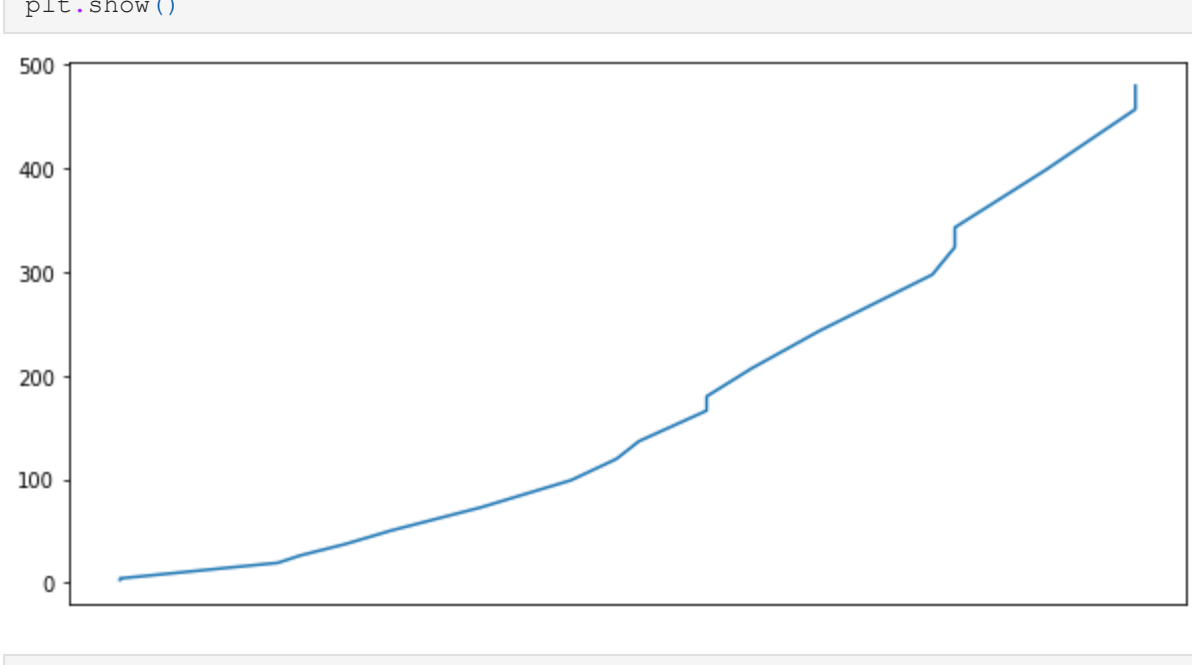


Set_ticks

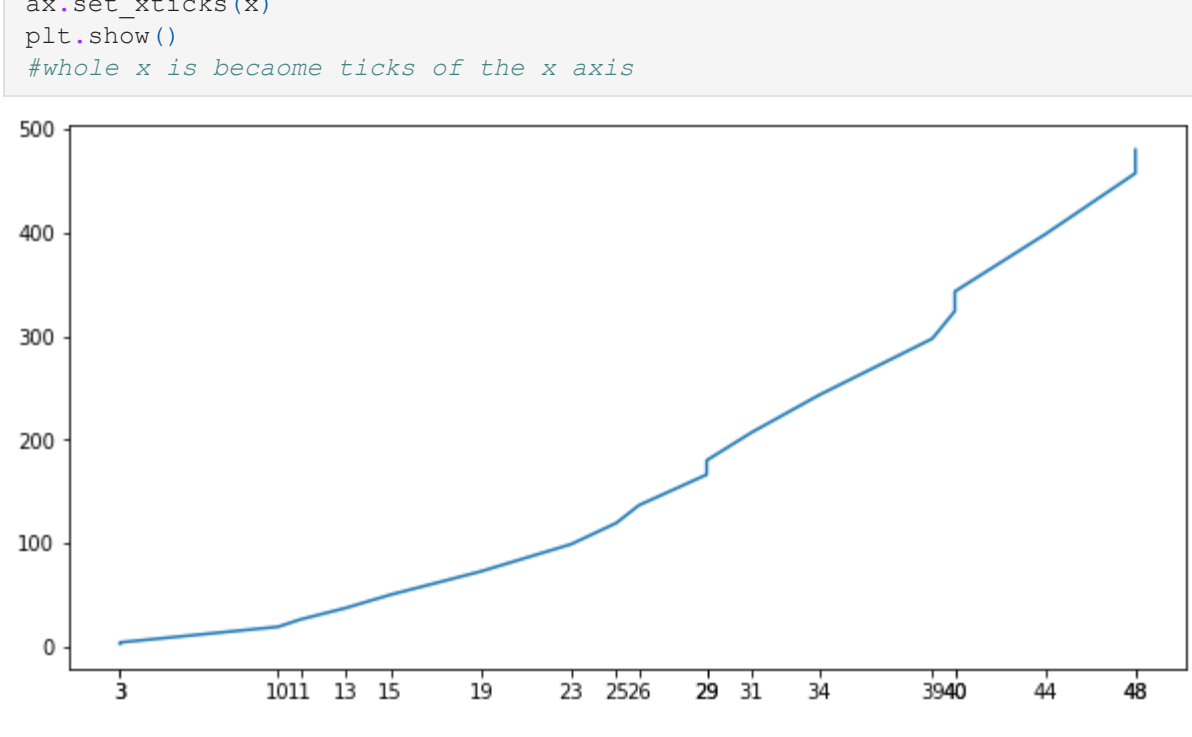
```
In [237… fig,ax = plt.subplots(figsize=(10,5))
ax.plot(x,y1)
ax.set_xticks([10,20,30,40])
plt.show()
```



```
In [238… fig,ax = plt.subplots(figsize=(10,5))
ax.plot(x,y1)
ax.set_xticks([])
plt.show()
```

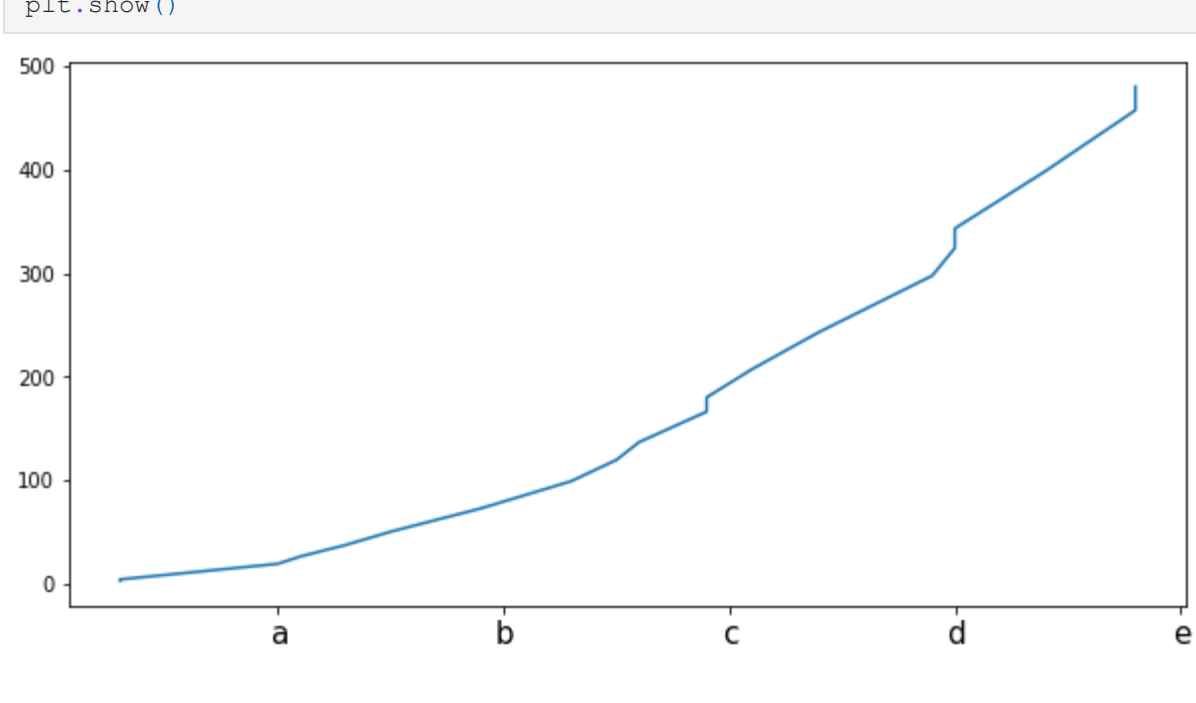


```
In [239… fig,ax = plt.subplots(figsize = (10,5))
ax.plot(x,y1)
ax.set_xticks(x)
plt.show()
#whole x is becaome ticks of the x axis
```



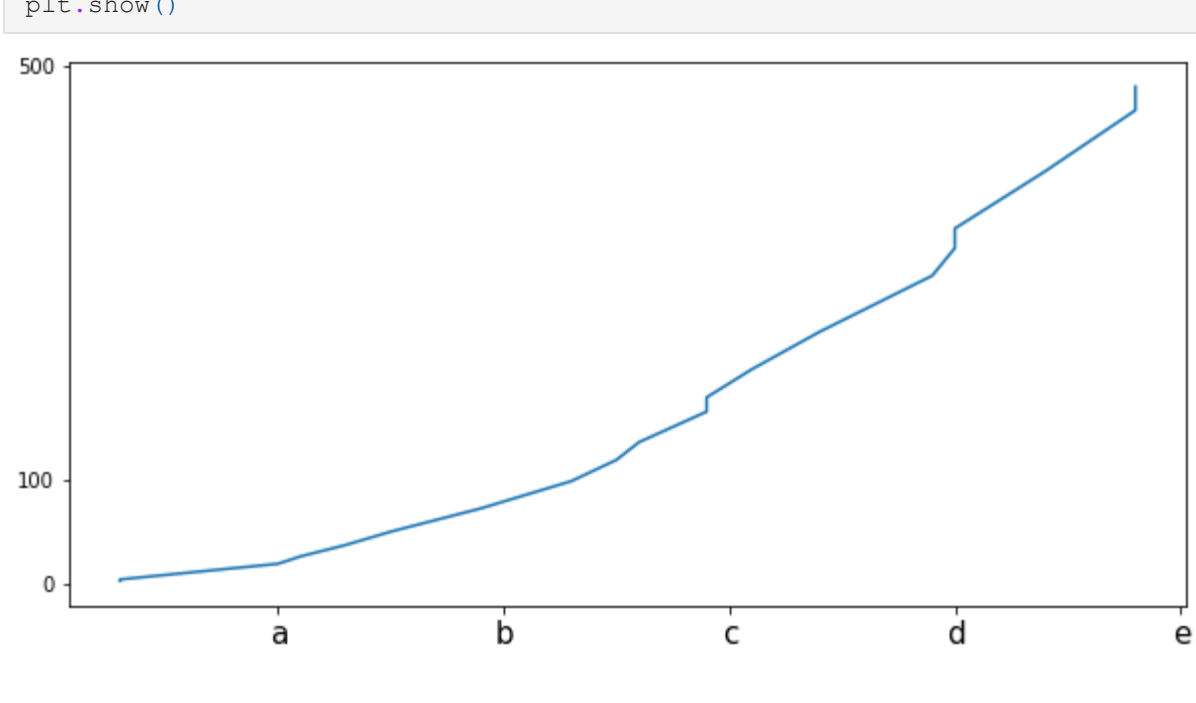
Set The Ticks Label .

```
In [241… fig,ax = plt.subplots(figsize = (10,5))
ax.plot(x,y1)
ax.set_xticks([10,20,30,40,50])
ax.set_xticklabels([r'a',r'b',r'c',r'd',r'e'],fontsize = 15)
plt.show()
```



Here we have set the y ticks labels.

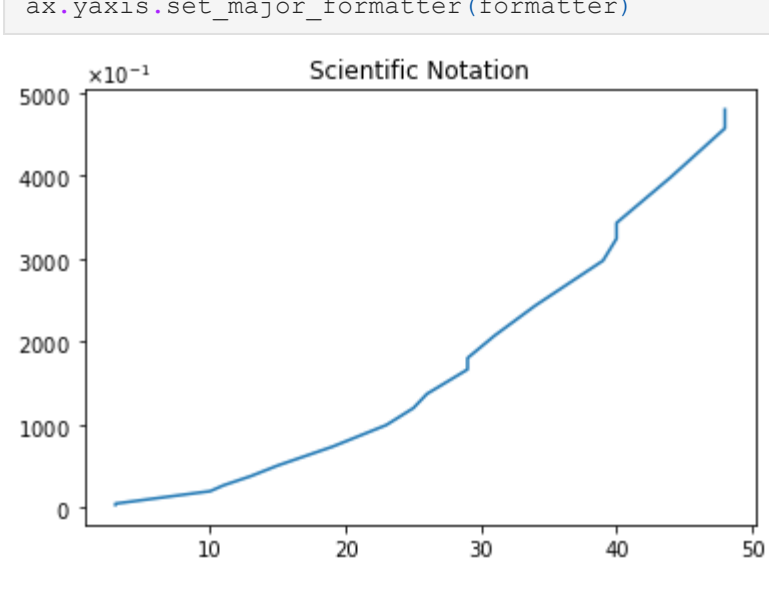
```
In [243… fig,ax = plt.subplots(figsize = (10,5))
ax.plot(x,y1)
ax.set_xticks([10,20,30,40,50])
ax.set_xticklabels([r'a',r'b',r'c',r'd',r'e'],fontsize = 15)
ax.set_yticks([0,100,500])
plt.show()
```



Scientific Notation.

```
In [ ]: from matplotlib import ticker
```

```
In [244… from matplotlib import ticker
fig,ax = plt.subplots()
ax.plot(x,y1)
ax.set_title('Scientific Notation')

formatter =ticker.ScalarFormatter(useMathText= True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,-1))
ax.yaxis.set_major_formatter(formatter)
```



Real Time application in data Analytics Of Matplotlib.

Data Visualization is an important part of business activities as organizations nowadays collect a huge amount of data. Sensors all over the world are collecting climate data, user data through clicks, car data for prediction of steering wheels etc. All of these data collected hold key insights for businesses and visualizations make these insights easy to interpret.

Why are visualizations important?

Visualizations are the easiest way to analyze and absorb information. Visuals help to easily understand the complex problem. They help in identifying patterns, relationships, and outliers in data. It helps in understanding business problems better and quickly. It helps to build a compelling story based on visuals. Insights gathered from the visuals help in building strategies for businesses. It is also a precursor to many high-level data analysis for Exploratory Data Analysis(EDA) and Machine Learning(ML). Human beings are visual creatures. Countless studies show how our brain is wired for the visual, and processes everything faster when it is through the eye.

```
In [3]: import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import pandas as pd
import os
```

```
In [8]: data = pd.read_csv('C:/Users/Microsoft/Desktop/pandas/ml-data-files/foods.csv',sep=',')
data.head()
```

Out[8]:

	First Name	Gender	City	Frequency	Item	Spend
0	Wanda	Female	Stamford	Weekly	Burger	15.66
1	Eric	Male	Stamford	Daily	Chalupa	10.56
2	Charles	Male	New York	Never	Sushi	42.14
3	Anna	Female	Philadelphia	Once	Ice Cream	11.01
4	Deborah	Female	Philadelphia	Daily	Chalupa	23.49

Histogram.

A histogram takes in a series of data and divides the data into a number of bins. It then plots the frequency data points in each bin (i.e. the interval of points). It is useful in understanding the count of data ranges.

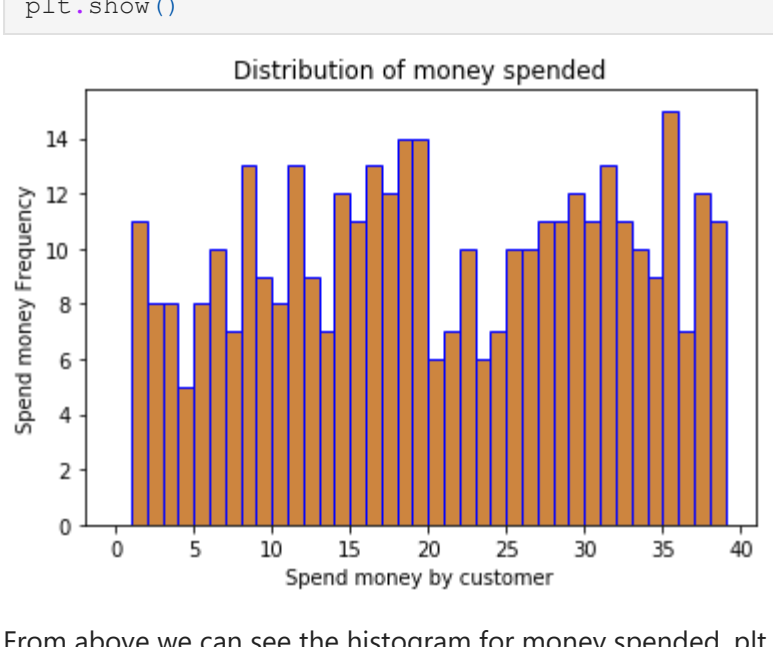
When to use:

We should use histogram when we need the count of the variable in a plot.

eg: Number of particular games sold in a store.

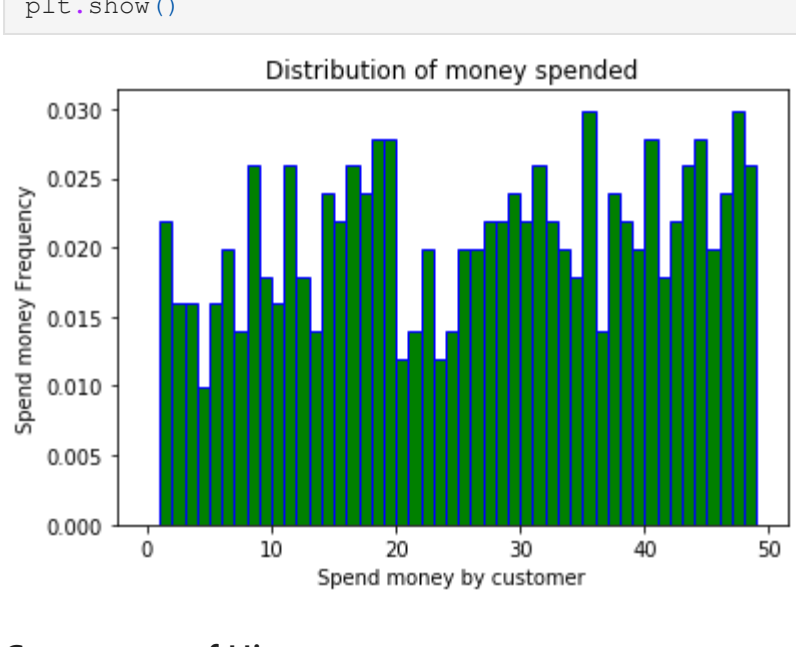
In this graph we are getting idea that money on the x axis and y is showing it's frequency.

```
In [15]: plt.hist(data['Spend'],facecolor ='peru',edgecolor='blue',bins=range(0,40))
plt.xlabel('Spend money by customer')
plt.ylabel('Spend money Frequency')
plt.title('Distribution of money spended')
plt.show()
```



From above we can see the histogram for money spended. plt.hist() takes the first argument as numeric data in the horizontal axis i.e Money spended.bins=range(0,50) is used to create bins from 0 to 50 between values of visitors in money spended.

```
In [21]: plt.hist(data['Spend'],facecolor ='green',edgecolor='blue',bins=range(0,50),density=True)
plt.xlabel('Spend money by customer')
plt.ylabel('Spend money Frequency')
plt.title('Distribution of money spended')
plt.show()
```



Componets of Histogram.

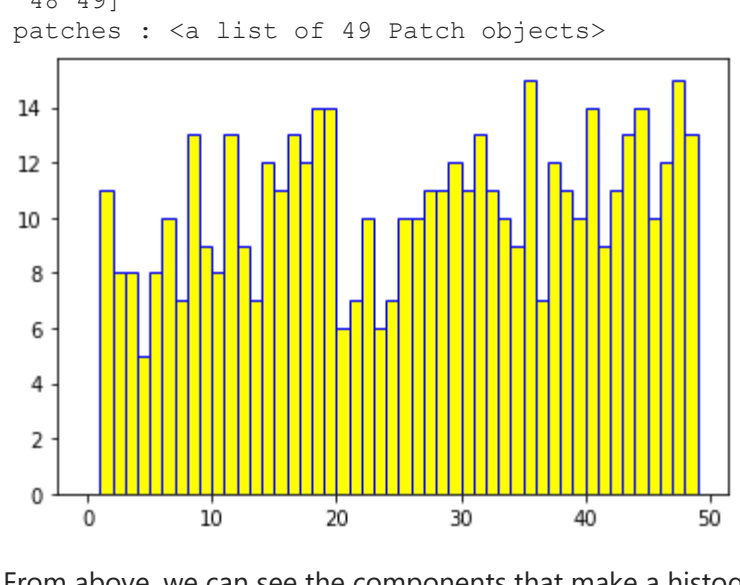
n= contains frequency of each bin.

bins = It represents middle values of each bins.

pathces = path of the object for the rectangular shape representing each bar.

```
In [23]: n,bins,pathces =plt.hist(data['Spend'],facecolor ='yellow',edgecolor='blue',bins=range(0,50))


print('n :',n)
print('bins :',bins)
print('patches :',pathces)
```

```
n : [ 0. 11.  8.  8.  5.  8. 10.  7. 13.  9.  8. 13.  9.  7. 12. 11. 13. 12.
 14. 14.  6.  7. 10.  6.  7. 10. 10. 11. 11. 12. 11. 13. 11. 10.  9. 15.
  7. 12. 11. 10. 14.  9. 11. 13. 14. 10. 12. 15. 13.]
bins : [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49]
patches : <a list of 49 Patch objects>
```

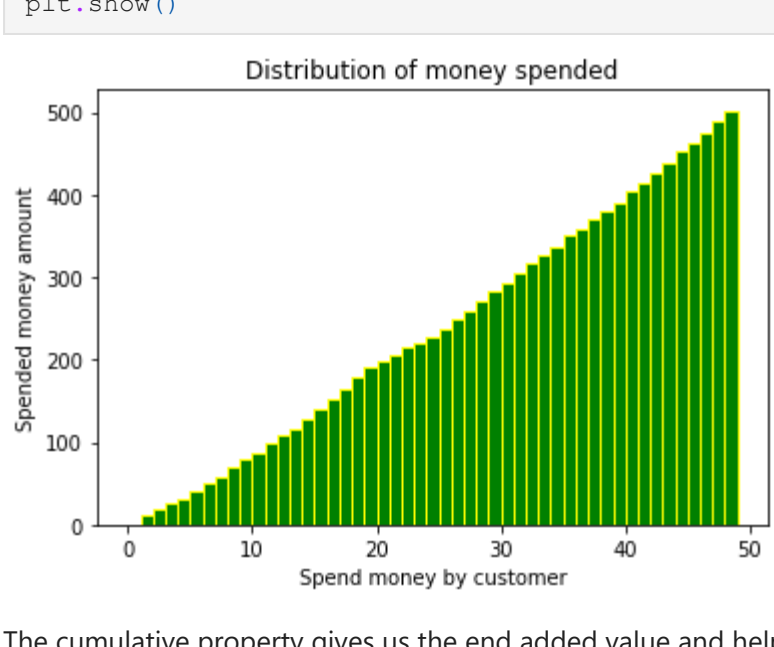


From above, we can see the components that make a histogram, n as the max values in each bin of histogram i.e 0.,11., and so on.

Cumulative Property.

If True then the histogram computed where each bin gives the countes in that bin plus all bins for the samll values.The bins gives the total numbetr of datapoints.

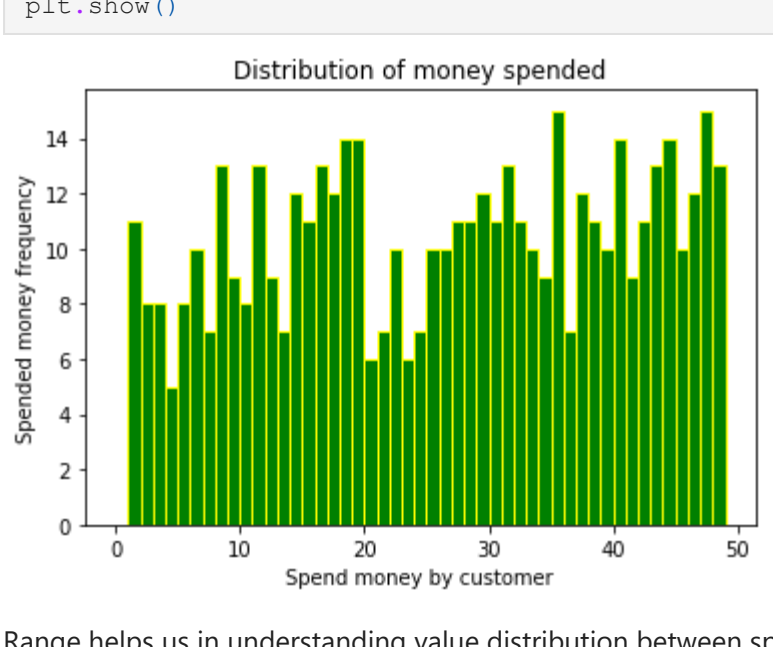
```
In [28]: plt.hist(data['Spend'],facecolor ='green',edgecolor='yellow',bins=range(0,50),cumulative=True)
plt.xlabel('Spend money by customer')
plt.ylabel('Spended money amount')
plt.title('Distribution of money spended')
plt.show()
```



The cumulative property gives us the end added value and helps us understand the increase in value at each bin.

Check the histogram to the range of the values.

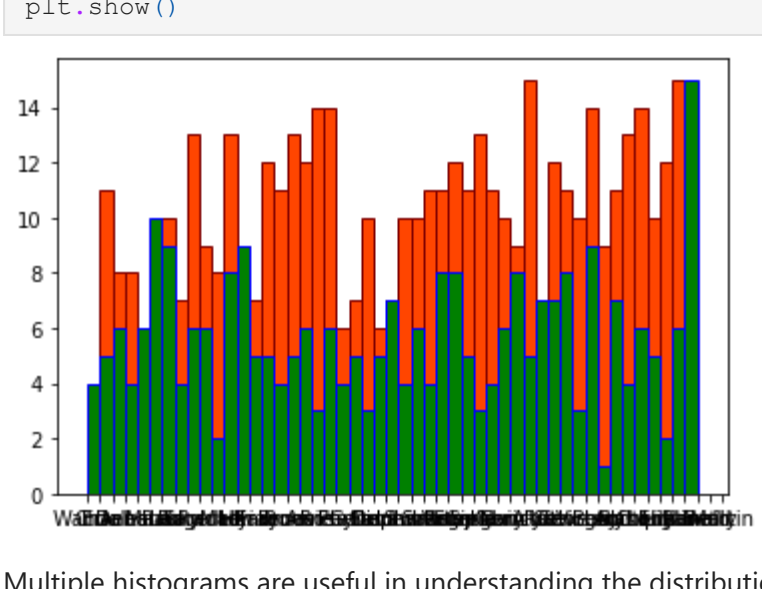
```
In [30]: plt.hist(data['Spend'],facecolor ='green',edgecolor='yellow',bins=range(0,50),range=(10,100))
plt.xlabel('Spend money by customer')
plt.ylabel('Spended money frequency')
plt.title('Distribution of money spended')
plt.show()
```



Range helps us in understanding value distribution between specified values.

Multiple Histogram.

```
In [39]: plt.hist(data['Spend'],facecolor ='orangered',edgecolor='maroon',bins=range(0,50))


plt.hist(data['First Name'],facecolor ='green',edgecolor='blue',bins=range(0,50))

plt.show()
```



Multiple histograms are useful in understanding the distribution between 2 entity variables. We can see that according to the name we got the spended money.

Pie Chart.

It is a circular plot which is divided into slices to illustrate numerical proportion. The slice of a pie chart is to show the proportion of parts out of a whole.

When to use:

Pie chart should be used seldom used as It is difficult to compare sections of the chart. Bar plot is used instead as comparing sections is easy.

eg: Market share in Films.

Note: Pie Charts is not a good chart to illustrate information.

Pie Chart.

Pie shows the distribution of data based on proportion of pie occupied.

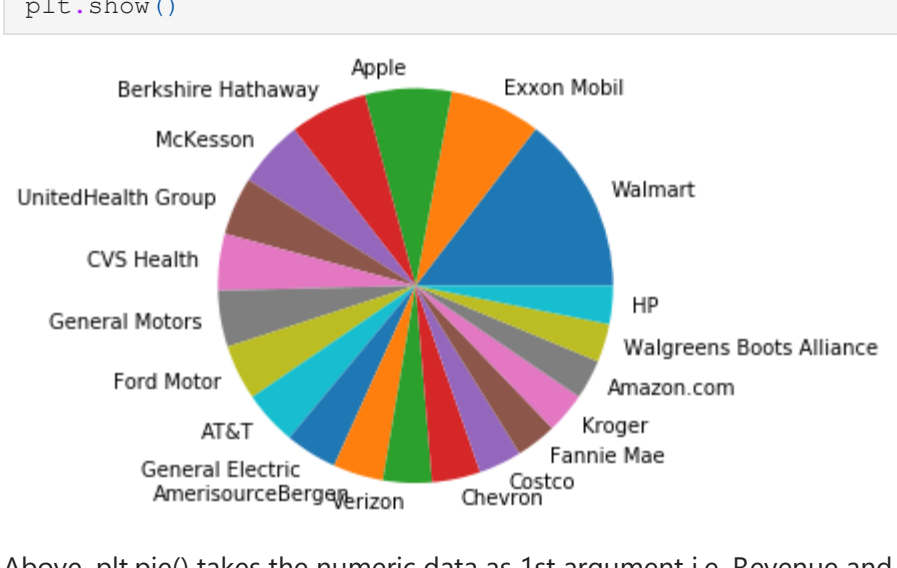
```
In [61]: data = pd.read_csv('C:/Users/microsoft/Desktop/pandas/ml-data-files/fortune1000.csv',index_col='Rank',nrows=20)
```

```
In [62]: data.head()
```

Out[62]:

Rank	Company	Sector	Industry	Location	Revenue	Profits	Employees
1	Walmart	Retailing	General Merchandisers	Bentonville, AR	482130	14694	2300000
2	Exxon Mobil	Energy	Petroleum Refining	Irving, TX	246204	16150	75600
3	Apple	Technology	Computers, Office Equipment	Cupertino, CA	233715	53394	110000
4	Berkshire Hathaway	Financials	Insurance: Property and Casualty (Stock)	Omaha, NE	210821	24083	331000
5	McKesson	Health Care	Wholesalers: Health Care	San Francisco, CA	181241	1476	70400

```
In [63]: plt.pie(data['Revenue'],labels=data['Company'])
plt.axis('equal')
plt.show()
```



Above, plt.pie() takes the numeric data as 1st argument i.e. Revenue and labels to display as second argument i.e Company . Ultimately, it shows the distribution of data in proportion to the pie.

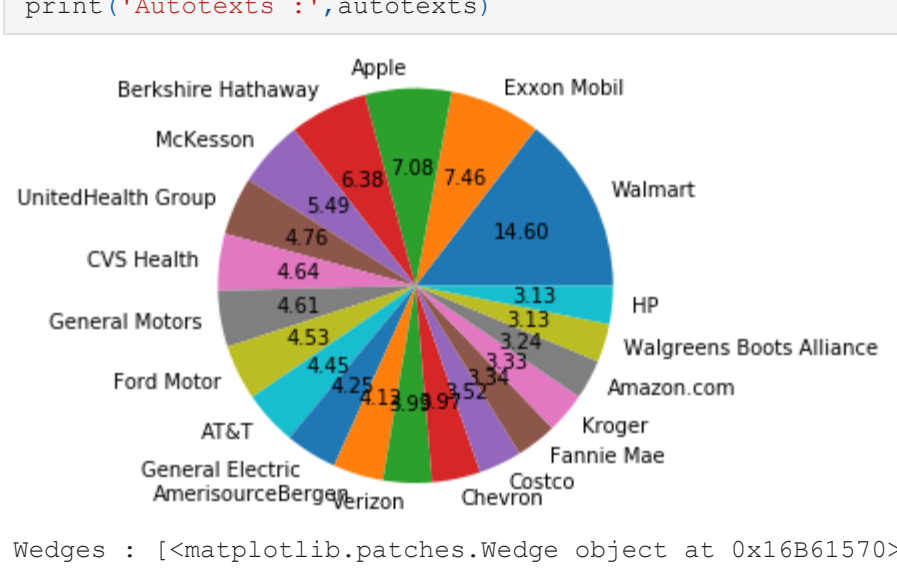
Pie Chart Customizations.

Wedges=A list parch objects representing each wedges.

Text= Use or text object represeting the labels.

Autotexts=The list of text objects for the numeric values -this only avalible if the values for the pie chart is not none.

```
In [67]: wedges,texts,autotexts= plt.pie(data['Revenue'],labels=data['Company'],autopct='%.2f')
plt.axis('equal')
plt.show()

print('Wedges :',wedges)
print('Texts :', texts)
print('Autotexts :',autotexts)
```



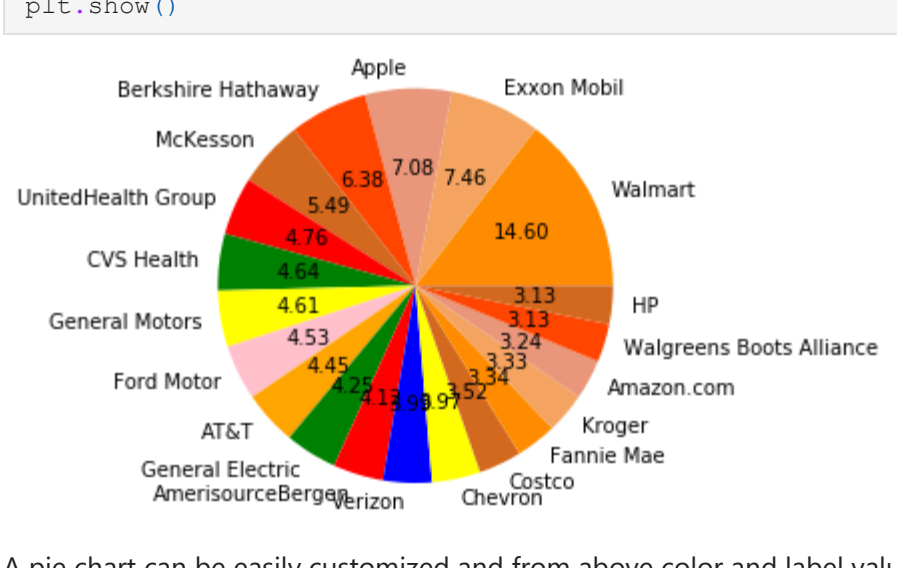
```
Wedges : [<matplotlib.patches.Wedge object at 0x16B61570>, <matplotlib.patches.Wedge object at 0x16B61850>, <matplotlib.patches.Wedge object at 0x16B61C50>, <matplotlib.patches.Wedge object at 0x16B80070>, <matplotlib.patches.Wedge object at 0x16B80570>, <matplotlib.patches.Wedge object at 0x16B809B0>, <matplotlib.patches.Wedge object at 0x16B80C90>, <matplotlib.patches.Wedge object at 0x16B8B090>, <matplotlib.patches.Wedge object at 0x16B80450>, <matplotlib.patches.Wedge object at 0x16B80890>, <matplotlib.patches.Wedge object at 0x16B54070>, <matplotlib.patches.Wedge object at 0x16B8BF50>, <matplotlib.patches.Wedge object at 0x16B9A350>, <matplotlib.patches.Wedge object at 0x16B9A730>, <matplotlib.patches.Wedge object at 0x16B9AB10>, <matplotlib.patches.Wedge object at 0x16B9AEF0>, <matplotlib.patches.Wedge object at 0x16BA52F0>, <matplotlib.patches.Wedge object at 0x16BA56D0>, <matplotlib.patches.Wedge object at 0x16BA5AB0>, <matplotlib.patches.Wedge object at 0x16BA5E90>]
Texts : [Text(0.9863216957661801, 0.4870005261402981, 'Walmart'), Text(0.44784475095269555, 1.0047064641198036, 'Exxon Mobil'), Text(-0.04093680015185061, 1.099237998976258, 'Apple'), Text(-0.4884416204337722, 0.9856088389559171, 'Berkshire Hathaway'), Text(-0.8139917295129246, 0.7398766547773743, 'McKesson'), Text(-1.0062395261778498, 0.4443894867763821, 'UnitedHealth Group'), Text(-1.0920100520155085, 0.1323406449171478, 'CVS Health'), Text(-1.0841149826913201, -0.1862651451673092, 'General Motors'), Text(-0.9869458488457811, -0.4857343836358312, 'Ford Motor'), Text(-0.8128534304298823, -0.7411270470690201, 'AT&T'), Text(-0.582743042484351, -0.932957955342084, 'General Electric'), Text(-0.3202759772995448, -1.0523418163148424, 'AmerisourceBergen'), Text(-0.04497183060515863, -1.0990803130126665, 'Verizon'), Text(0.22827447410084875, -1.0760533278950357, 'Chevron'), Text(0.47281652197711777, -0.9931991424409617, 'Costco'), Text(0.6742783224034855, -0.8691080162653785, 'Fannie Mae'), Text(0.8402562953174939, -0.7099079927563294, 'Kroger'), Text(0.9678412237281707, -0.5227651151830592, 'Amazon.com'), Text(1.0524689318835598, -0.3198580113423435, 'Walgreens Boots Alliance'), Text(1.0946876243850052, -0.107976872608508, 'HP')]
Autotexts : [Text(0.5379936522360982, 0.2656366506219807, '14.60'), Text(0.2442789550651066, 0.548021707701711, '7.46'), Text(-0.02232916371919124, 0.5995843630779588, '7.08'), Text(-0.2664227020547848, 0.537604821248682, '6.38'), Text(-0.4439954888252316, 0.4035690844240223, '5.49'), Text(-0.5488579233697363, 0.2423942655143902, '4.76'), Text(-0.5956418465539136, 0.0721858063184425, '4.64'), Text(-0.5913354451043564, -0.10159917009125956, '4.61'), Text(-0.538334099370426, -0.2649460274377261, '4.53'), Text(-0.4433745984162994, -0.4042511168558291, '4.45'), Text(-0.31785984135510054, -0.5088861574593185, '4.25'), Text(-0.1746959876179335, -0.5740046270808231, '4.12'), Text(-0.02453008942099561, -0.5994983525523635, '3.99'), Text(0.12451334950955385, -0.5869381788518375, '3.97'), Text(0.25789992107842785, -0.541744986785979, '3.52'), Text(0.3677881758564466, -0.4740589179629365, '3.34'), Text(0.4583216156277239, -0.3872225415034524, '3.33'), Text(0.5279133947608203, -0.28514460828166865, '3.24'), Text(0.5740739628455781, -0.17446800618673278, '3.13'), Text(0.5971023405736391, -0.05889647596827708, '3.13')]
```

From above we can the components that make a pie chart and it returns wedge object, text in labels and so on.

Pie Chart Custmizations.

We set the value for color and set the autopct properties.This latter sets the format for the values to be displayed.

```
In [75]: color = ['darkorange','sandybrown','darksalmon','orangered','chocolate','red','green','yellow','pink','orange',

wedges,texts,autotexts= plt.pie(data['Revenue'],labels=data['Company'],autopct='%.2f',colors=color)
plt.axis('equal')
plt.show()
```



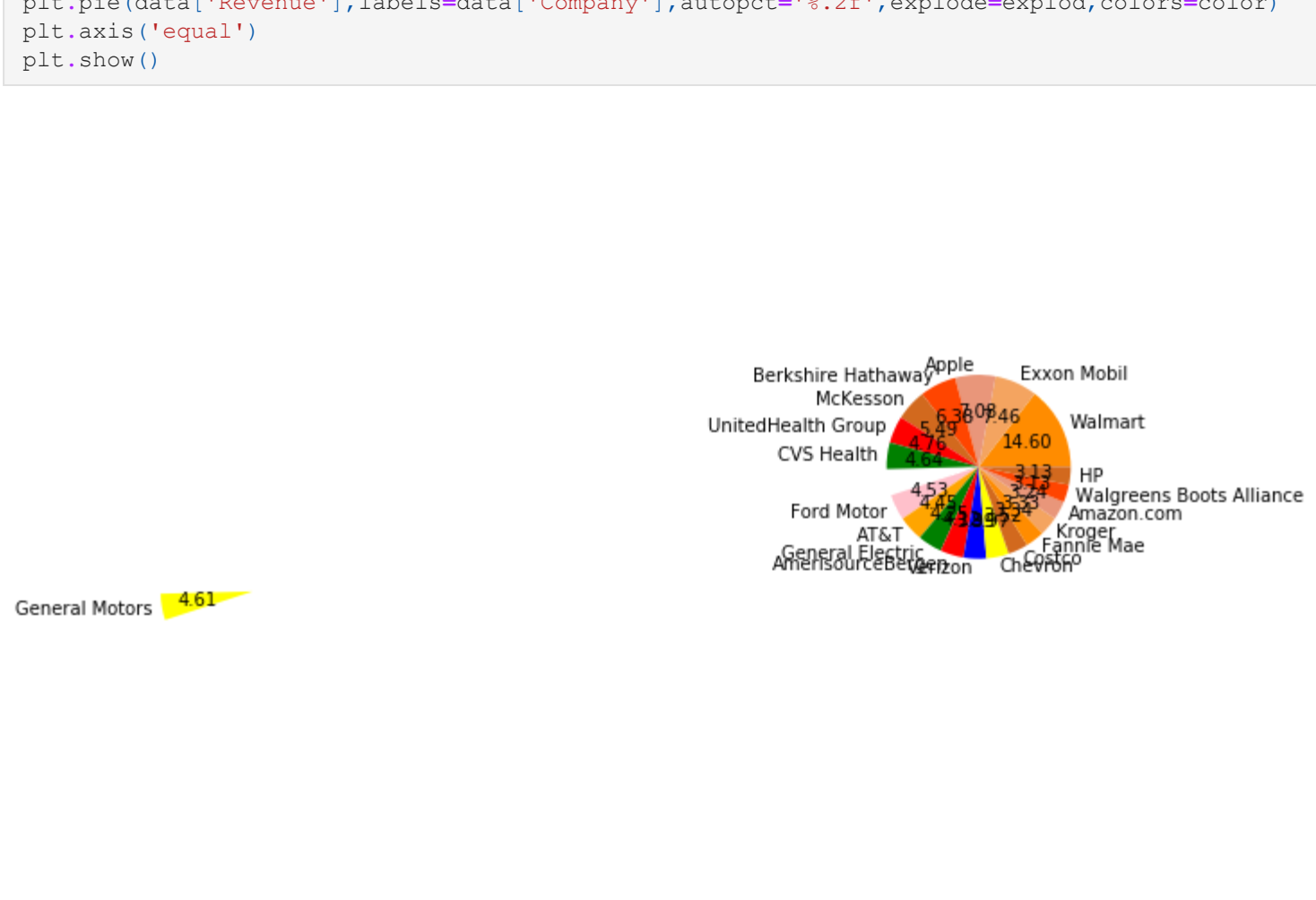
A pie chart can be easily customized and from above color and label values are formatted.

The explod Property.

To highlight the particular widges of the pie chart ,we can exploid to seperate it from the rest of the chart.

The values for 'explod' represents the fraction of the radius with which to offset each wedge.

```
In [90]: plt.figure(figsize=(10,8))
explod = [0,0,0,0,0,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0]

plt.pie(data['Revenue'],labels=data['Company'],autopct='%.2f',explode=explod,colors=color)
plt.axis('equal')
plt.show()
```



Explod lengnth should be equal to the len(x i.e.data['Revenue']).

From above explode is used to separate out points from the pie. Similar to a pizza piece being cut.

Time Series by line plot.

Time series is a line plot and it is basically connecting data points with a straight line. It is useful in understanding the trend over time. It can explain the correlation between points by the trend. An upward trend means positive correlation and downward trend means a negative correlation. It mostly used in forecasting, monitoring models.

When to use:

Time Series should be used when single or multiple variables are to be plotted over time.

eg: Stock Market Analysis of Companies, Weather Forecasting.

```
In [102… data = pd.read_csv('C:/Users/Microsoft/Desktop/pandas/ml-data-files/forestfires2.csv',nrows=20)
```

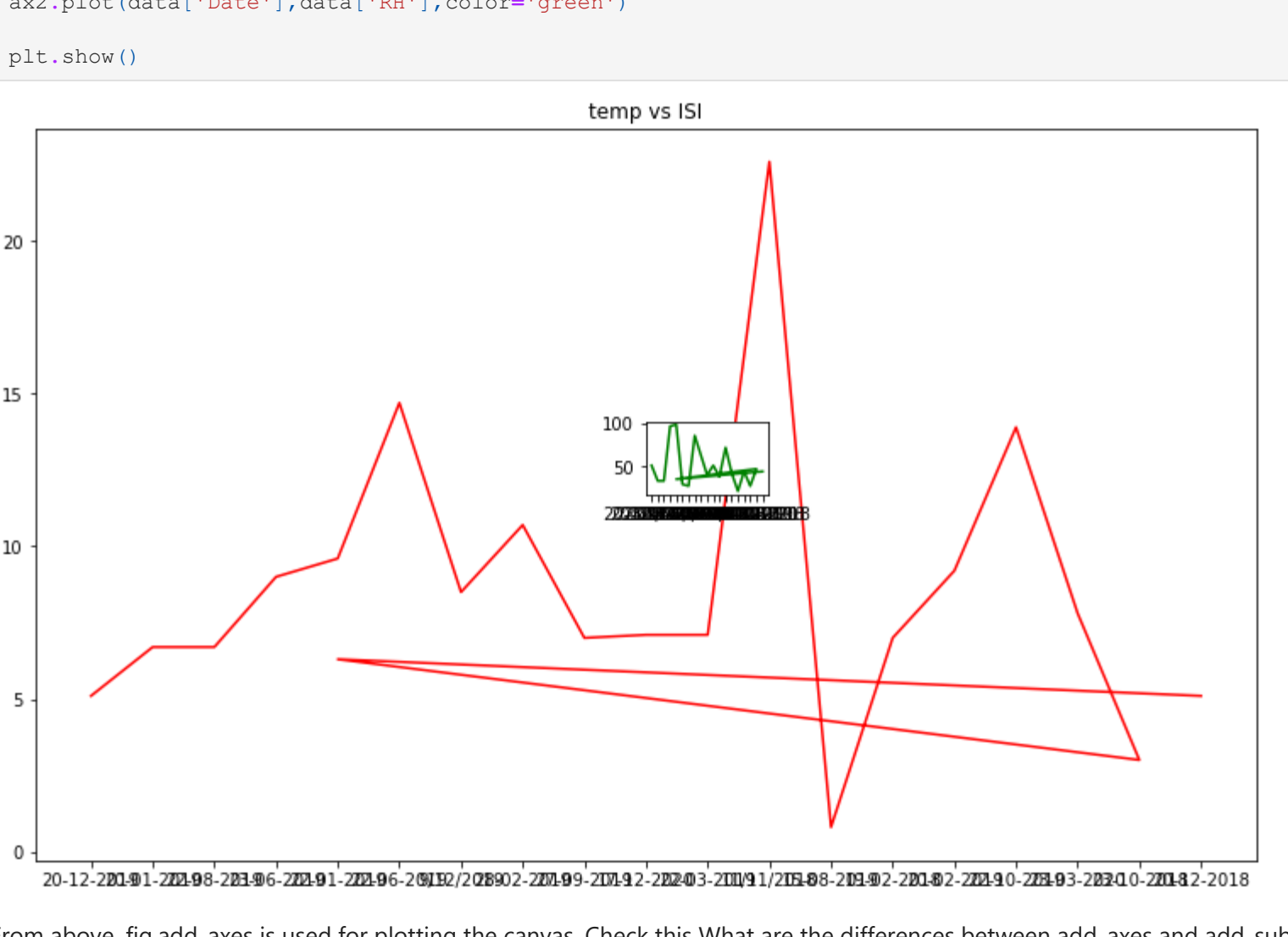
```
In [103… data.head()
```

Out[103…

	Date	Y	month	day	FFMC	DMC	DC	ISI	temp	RH	wind	rain	area
0	20-12-2019	5	mar	fri	86.2	26.2	94.3	5.1	8.2	51	6.7	0.0	0
1	20-01-2019	4	oct	tue	90.6	35.4	669.1	6.7	18.0	33	0.9	0.0	0
2	22-08-2019	4	oct	sat	90.6	43.7	686.9	6.7	14.6	33	1.3	0.0	0
3	23-06-2019	6	mar	fri	91.7	33.3	77.5	9.0	8.3	97	4.0	0.2	0
4	22-01-2019	6	mar	sun	89.3	51.3	102.2	9.6	11.4	99	1.8	0.0	0

Compare the temp,ISI,RH side by side

```
In [113… fig= plt.figure(figsize=(10,6))

ax1 = fig.add_axes([0,0,1,1])
ax2 = fig.add_axes([0.5,0.5,0.1,0.1])

ax1.set_title('temp vs ISI')

ax1.plot(data['Date'],data['ISI'],color='red')

ax2.plot(data['Date'],data['RH'],color='green')

plt.show()
```



From above, fig.add_axes is used for plotting the canvas. Check this What are the differences between add_axes and add_subplot?

to understand axes and subplots. plt.plot() takes the 1st argument as numeric data i.e Date and 2nd argument is to numeric ISI data.ISI is considered as ax1 which is the outer figure and on ax2 RH is considered for plotting which is inset.

Boxplot and Violinplot.

Boxplot gives a nice summary of the data. It helps in understanding our distribution better.

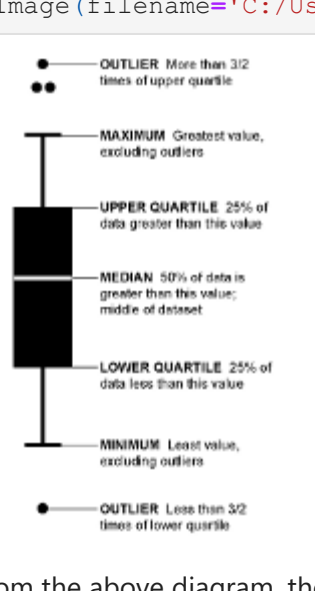
When to use:

It should be used when we require to use the overall statistical information on the distribution of the data. It can be used to detect outliers in the data.

eg: Credit Score of Customer. We can get the max, min and much more information about the mark.

```
In [115… from IPython.display import Image
Image(filename='C:/Users/Microsoft/Downloads/photo44.png')
```

Out[115…



From the above diagram, the line that divides the box into 2 parts represents the median of the data. The end of the box shows the upper quartile(75%)and the start of the box represents the lower quartile(25%). Upper Quartile is also called 3rd quartile and similarly, Lower Quartile is also called as 1st quartile. The region between lower quartile and the upper quartile is called as Inter Quartile Range(IQR) and it is used to approximate the 50% spread in the middle data(75–25=50%).

The maximum is the highest value in data, similarly minimum is the lowest value in data, it is also called as caps. The points outside the boxes and between the maximum and maximum are called as whiskers, they show the range of values in data. The extreme points are outliers to the data. A commonly used rule is that a value is an outlier if it's less than lower quartile-1.5 *IQR or high than the upper quartile + 1.5* IQR.

```
In [117… data = pd.read_csv('C:/Users/Microsoft/Desktop/pandas/ml-data-files/titanic.csv')
data.head()
```

Out[117…

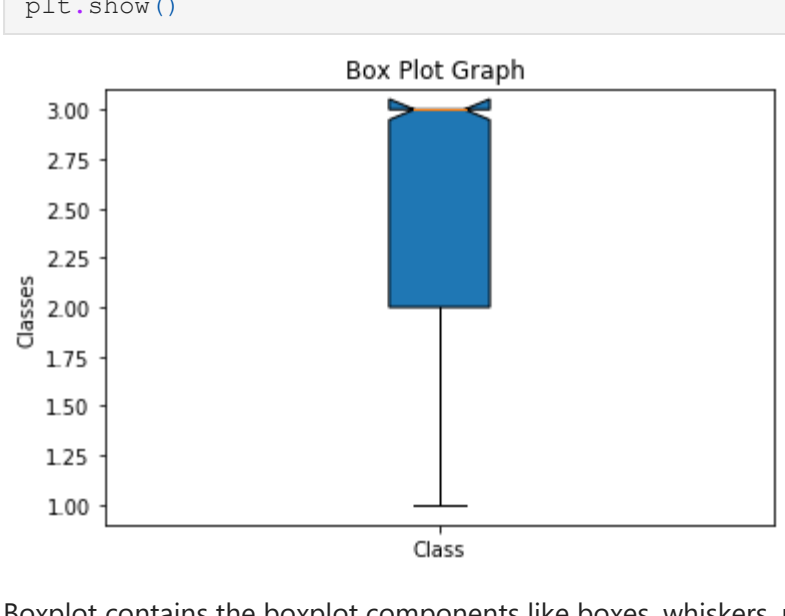
	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

```
In [118… print(data.describe())
```

```
       PassengerId    Survived      Pclass         Age       SibSp  \
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

In [153…

```python
plt.boxplot(data['Pclass'],notch=True, patch_artist=True)
plt.title('Box Plot Graph ')
plt.ylabel('Classes')
plt.xticks([1],['Class'])
plt.show()
```



Boxplot contains the boxplot components like boxes, whiskers, medians, caps. Seaborn another plotting library makes it easier to build custom plots than matplotlib. patch_artist makes the customization possible. notchmakes the median look more prominent.

A caveat of using boxplot is the number of observations in the unique value is not defined, Jitter Plot in Seaborn can overcome this caveat or Violinplot is also useful.

Violin plot.

Violin plot is a better chart than boxplot as it gives a much broader understanding of the distribution. It resembles a violin and dense areas point the more distribution of data otherwise hidden by box plots.

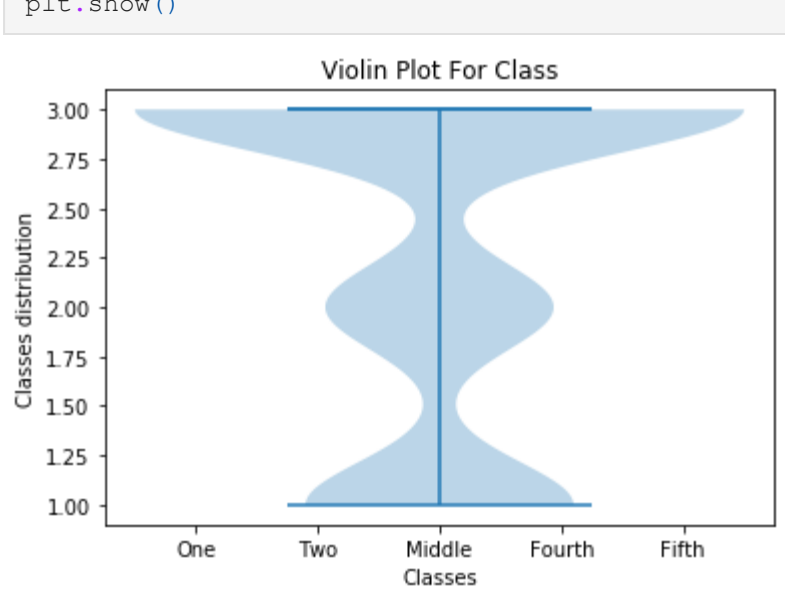
When to use:

Its an extension to boxplot. It should be used when we require a better intuitive understanding of data.

It is same as boxplot they can show the density of the data points around the particular values with their widths.

In [155…

```python
vp = plt.violinplot(data['Pclass'],showmedians=True)
plt.xticks([0.8,0.9,1.0,1.1,1.2],['One','Two','Middle','Fourth','Fifth'])
plt.title('Violin Plot For Class')
plt.ylabel('Classes distribution')
plt.xlabel('Classes')
plt.show()
```



The density of points in the middle seems more as classes tend to score around average mostly..

Stack Plot and Stem Plot.

Stack Plot.

Stack plot visualizes data in stacks and shows the distribution of data over time.

When to use: It is used for checking multiple variable area plots in a single plot.

Eg: It is useful in understanding the change of distribution in multiple variables over an interval.

In [178…

```python
data = pd.read_csv('C:/Users/Microsoft/Desktop/pandas/ml-data-files/foods22.csv',nrows=25)
data.head()
```

Out[178…

	Year	Gender	City	Frequency	Item	Spend	People	Maintainace
0	2015	Female	Stamford	Weekly	Burger	15.66	23.0	654.0
1	2016	Male	Stamford	Daily	Chalupa	10.56	23.0	44.0
2	2017	Male	New York	Never	Sushi	42.14	43.0	33.0
3	2018	Female	Philadelphia	Once	Ice Cream	11.01	565.0	787.0
4	2019	Female	Philadelphia	Daily	Chalupa	23.49	56.0	44.0

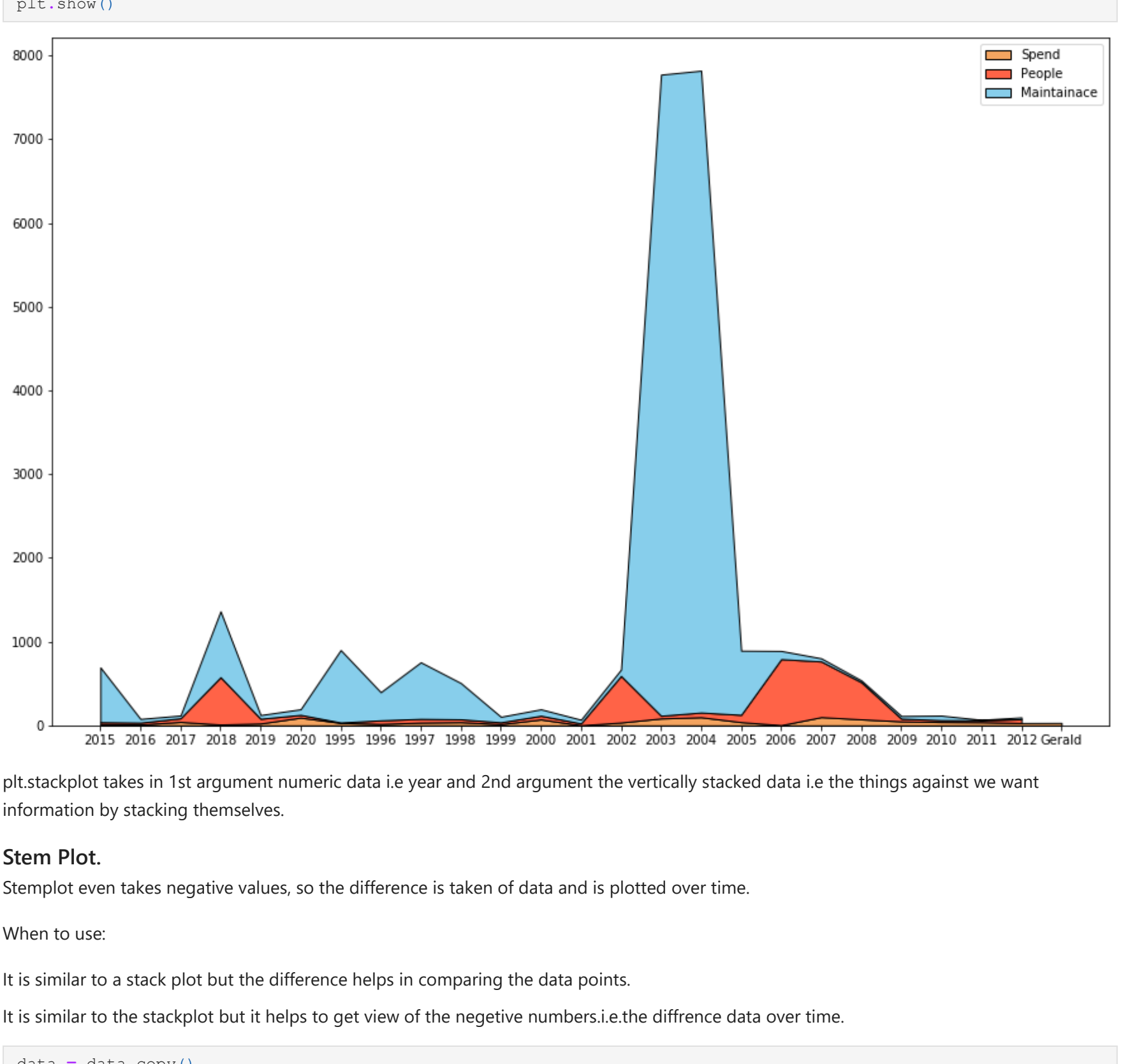
In [183…

```python
x = data['Year']

y= np.vstack([data['Spend'],data['People'],data['Maintainace']])
```

Stacks Plots.

In [196…

```python
#labels to each stacks
plt.figure(figsize =(15,10))

label= ['Spend','People','Maintainace']

#color to each data.plot.area()

color= ['sandybrown','tomato','skyblue']

plt.stackplot(x,y,labels=label,colors=color,edgecolor = 'black')

plt.legend()
plt.show()
```



plt.stackplot takes in 1st argument numeric data i.e year and 2nd argument the vertically stacked data i.e the things against we want information by stacking themselves.

Stem Plot.

Stemplot even takes negative values, so the difference is taken of data and is plotted over time.

When to use:

It is similar to a stack plot but the difference helps in comparing the data points.

It is similar to the stackplot but it helps to get view of the negetive numbers.i.e.the diffrence data over time.

In [199…

```python
data = data.copy()
```

In [200…

```python
data.head()
```

Out[200…

	Date	New York	Los Angeles	Miami
0	1/1/16	985	122	499
1	1/2/16	738	788	534
2	1/3/16	14	20	933
3	1/4/16	730	904	885
4	1/5/16	114	71	253

In [201…

```python
data[['New York','Los Angeles','Miami']]=data[['New York','Los Angeles','Miami']].diff()

data.head()
```

Out[201…

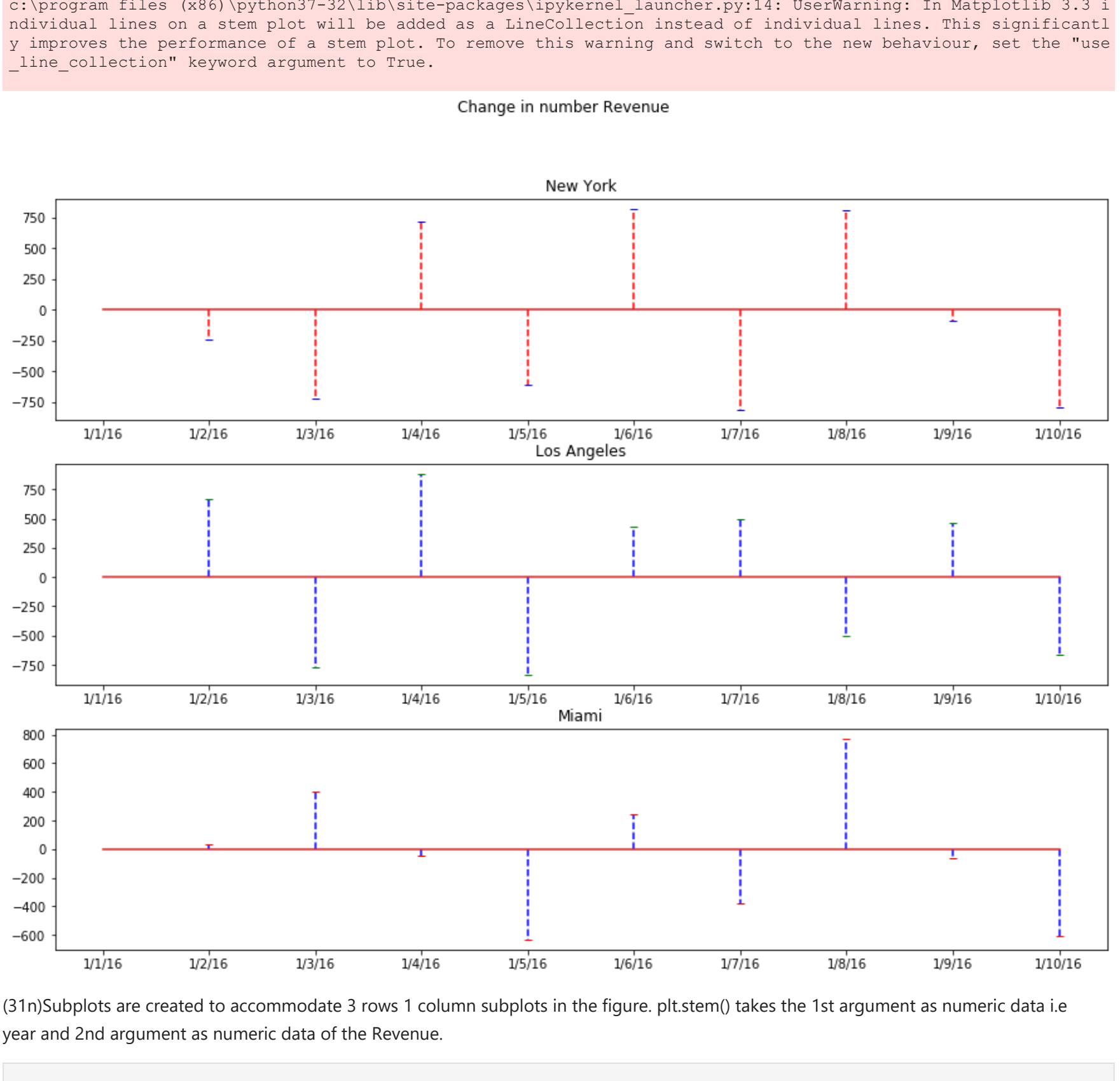
	Date	New York	Los Angeles	Miami
0	1/1/16	NaN	NaN	NaN
1	1/2/16	-247.0	666.0	35.0
2	1/3/16	-724.0	-768.0	399.0
3	1/4/16	716.0	884.0	-48.0
4	1/5/16	-616.0	-833.0	-632.0

diff() is used to find the difference between previous data and is stored in another copy of the data. The first data point is NaN (Not a Number) as it doesn't contain any previous data for calculating the difference.

In [209…

```python
plt.figure(figsize =(15,11))

plt.suptitle('Change in number Revenue',y=0.98)

plt.subplot(3,1,1)
plt.stem(data['Date'],data['New York'],markerfmt='b_',linefmt='r--',basefmt='C3-')
plt.title('New York')

plt.subplot(3,1,2)
plt.stem(data['Date'],data['Los Angeles'],markerfmt='g_',linefmt='b--',basefmt='C3-')
plt.title('Los Angeles')

plt.subplot(3,1,3)
plt.stem(data['Date'],data['Miami'],markerfmt='r_',linefmt='b--',basefmt='C3-')
plt.title('Miami')
plt.show()
```

```
c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:6: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection instead of individual lines. This significantly improves the performance of a stem plot. To remove this warning and switch to the new behaviour, set the "use_line_collection" keyword argument to True.

c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:10: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection instead of individual lines. This significantly improves the performance of a stem plot. To remove this warning and switch to the new behaviour, set the "use_line_collection" keyword argument to True.
  # Remove the CWD from sys.path while we load stuff.
c:\program files (x86)\python37-32\lib\site-packages\ipykernel_launcher.py:14: UserWarning: In Matplotlib 3.3 individual lines on a stem plot will be added as a LineCollection instead of individual lines. This significantly improves the performance of a stem plot. To remove this warning and switch to the new behaviour, set the "use_line_collection" keyword argument to True.
```



(31n)Subplots are created to accommodate 3 rows 1 column subplots in the figure. plt.stem() takes the 1st argument as numeric data i.e year and 2nd argument as numeric data of the Revenue.

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: