hqr2d5qfe

April 15, 2024

1 MATPLOTLIB

1.0.1 let's start by step by step in matplotlib as we have to sart from the begineer to advanced .

Installation

[1]:
```
!pip install matplotlib
```

```
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist-
packages (3.7.1)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.2.1)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (4.51.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (1.4.5)
Requirement already satisfied: numpy>=1.20 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (1.25.2)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (24.0)
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/dist-
packages (from matplotlib) (9.4.0)
Requirement already satisfied: pyparsing>=2.3.1 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.10/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-
packages (from python-dateutil>=2.7->matplotlib) (1.16.0)
```

Many of the students really don't know why the exclamation(!) is used while installing. The main reason behind this is that we are running it in an environment, so the "!" gives the command as a shell command. Basically, it instructs to pass the command to the operating system for execution.

Importing the Library

[2]:
```
import matplotlib.pyplot as plt
import pandas as pd
```

```
import numpy as np
```

We can't only use the single library Matplotlib; we also need Pandas and Numpy, as they are used for basic manipulation of the data.

pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

As the Matplotlib library is mostly related to graphs, So we will directly start with their

Let's see the variety of plots which are present in the Matplotlib

There are mainly 5 types of plots

- Pairwise data
- Statistical distributions
- Gridded data

Thier are also more two types but that are not very important at this stage

1.1 1.Pairwise data

Pairwise data isn't a standard term in data analysis, but it can have a few different interpretations depending on the context

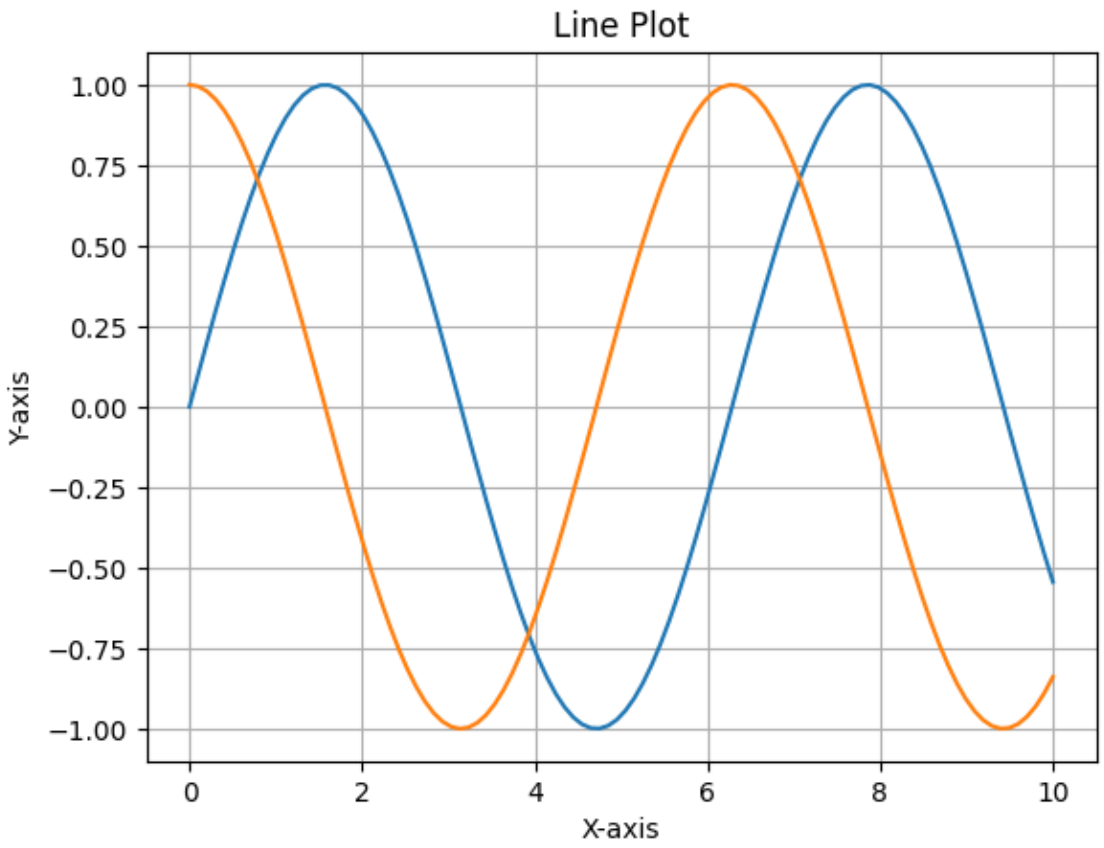
There many types of plots in pairwise :

- Line plot
- scatter plot
- bar plot
- stem plot
- fill_between plot
- stackplot plot
- stairs plot

1.1.1 Line plot

In Matplotlib, this is the simplest basic plotting function. The two lists or arrays, x and y, that represent the data points are connected by a line plot that is produced.

[4]:
```
x = np.linspace(0, 10, 100)
y1 = np.sin(x)
y2 = np.cos(x)
values = np.random.rand(10)

plt.plot(x, y1, label='Sine')
plt.plot(x, y2, label='Cosine')
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Plot')
plt.grid(True)
plt.show()
```



Let's first grasp a few terms in this example so that it becomes clearer from there.

plt.plot() is the function used to create a line plot.

These lines set the labels for the x-axis, y-axis, and the title of the plot

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Line Plot')

plt.grid(True) : This line adds a grid to the plot for better readability

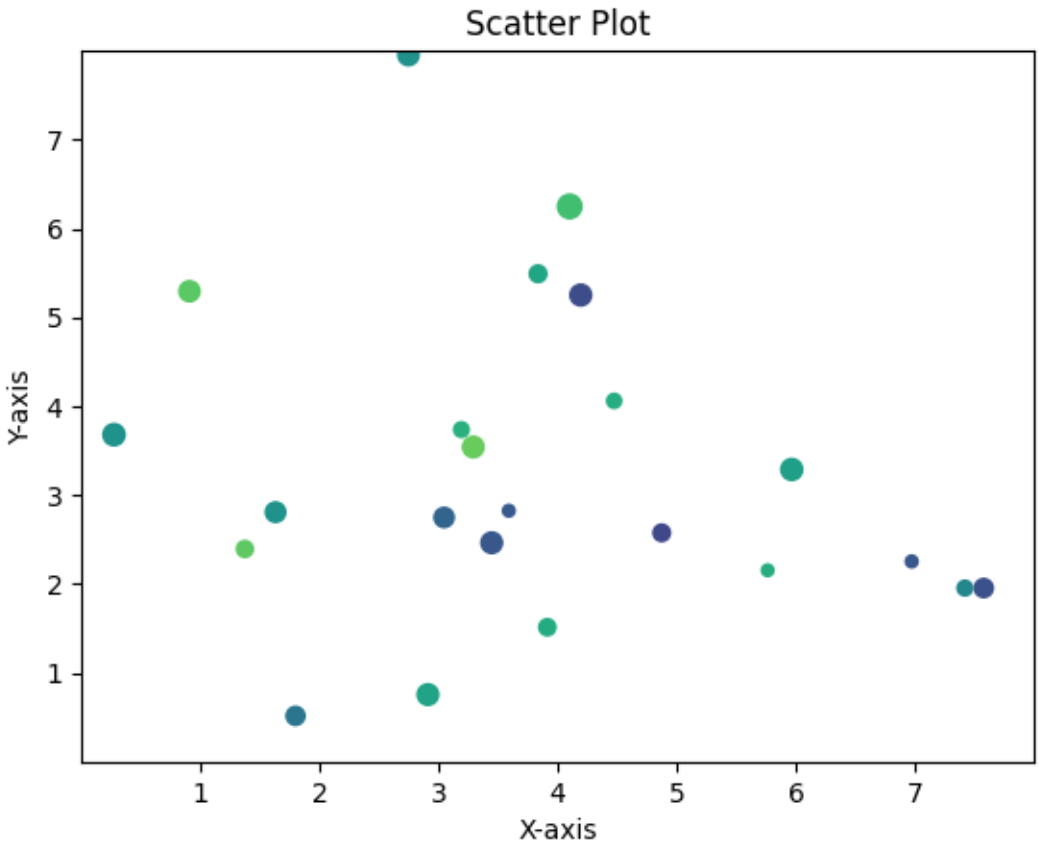
plt.show() : this line displays the plot on the screen.

1.1.2 Scatter Plot

plot, in which a marker—typically a circle—represents each data point in x and y on the graph. It is helpful in displaying the data distribution without necessarily highlighting a pattern.

[5]:
```
np.random.seed(3)
x = 4 + np.random.normal(0, 2, 24)
y = 4 + np.random.normal(0, 2, len(x))
```

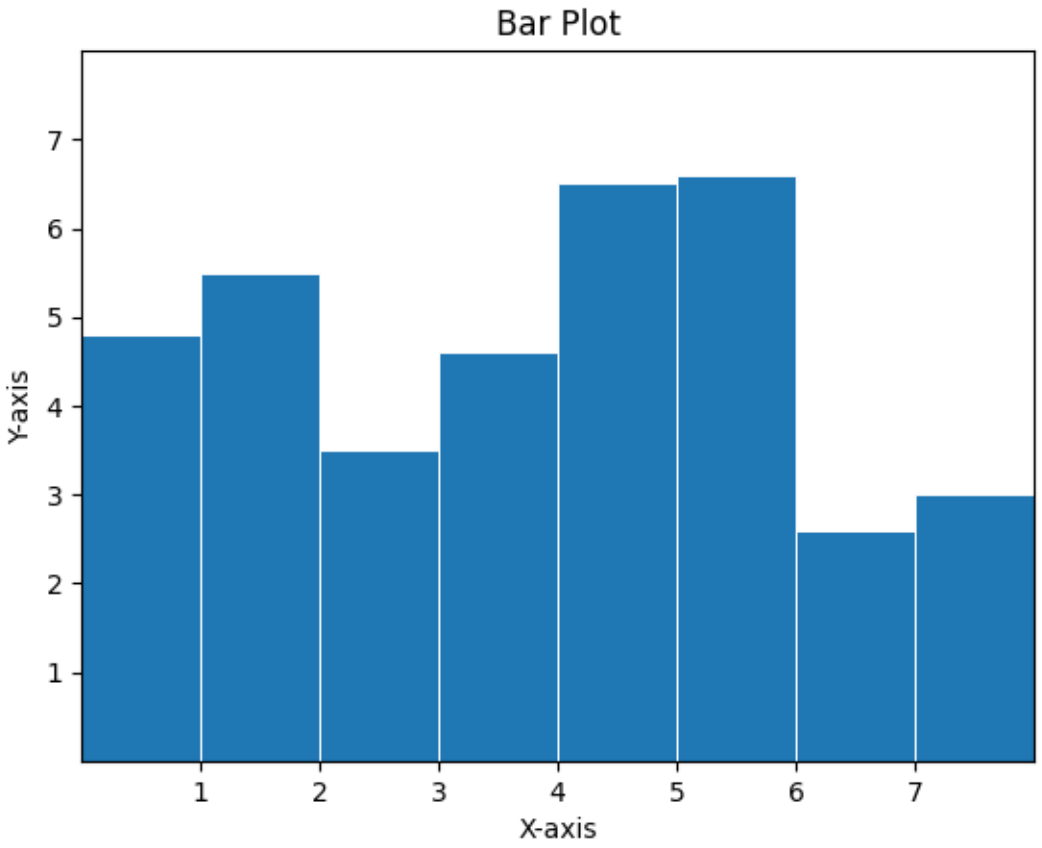
```
sizes = np.random.uniform(15, 80, len(x))
colors = np.random.uniform(15, 80, len(x))

fig, ax = plt.subplots()

ax.scatter(x, y, s=sizes, c=colors, vmin=0, vmax=100)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Scatter Plot')

plt.show()
```



1.1.3 Bar Plot

This feature produces a bar graph. It accepts two arguments: height, which is a list or array containing the bar heights, and x, which indicates the bar placements on the x-axis.

[6]:
```
x = 0.5 + np.arange(8)
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]
fig, ax = plt.subplots()
ax.bar(x, y, width=1, edgecolor="white", linewidth=0.7)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Bar Plot')
plt.show()
```

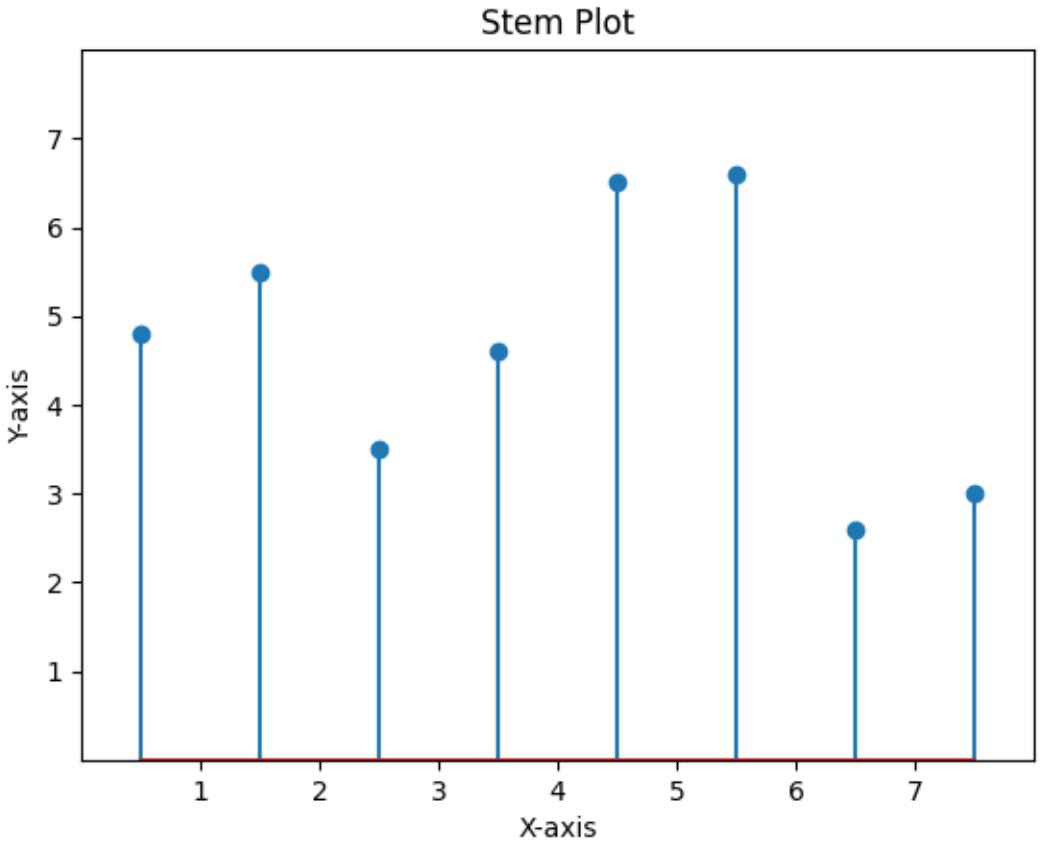


1.1.4 Stem Plot

This function creates a stem plot, which is a type of plot useful for visualizing discrete data. It displays vertical lines (stems) at the positions specified by x and horizontal lines at the corresponding values in y.

[7]:
```
x = 0.5 + np.arange(8)
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]
```

```
fig, ax = plt.subplots()

ax.stem(x, y)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Stem Plot')
plt.show()
```

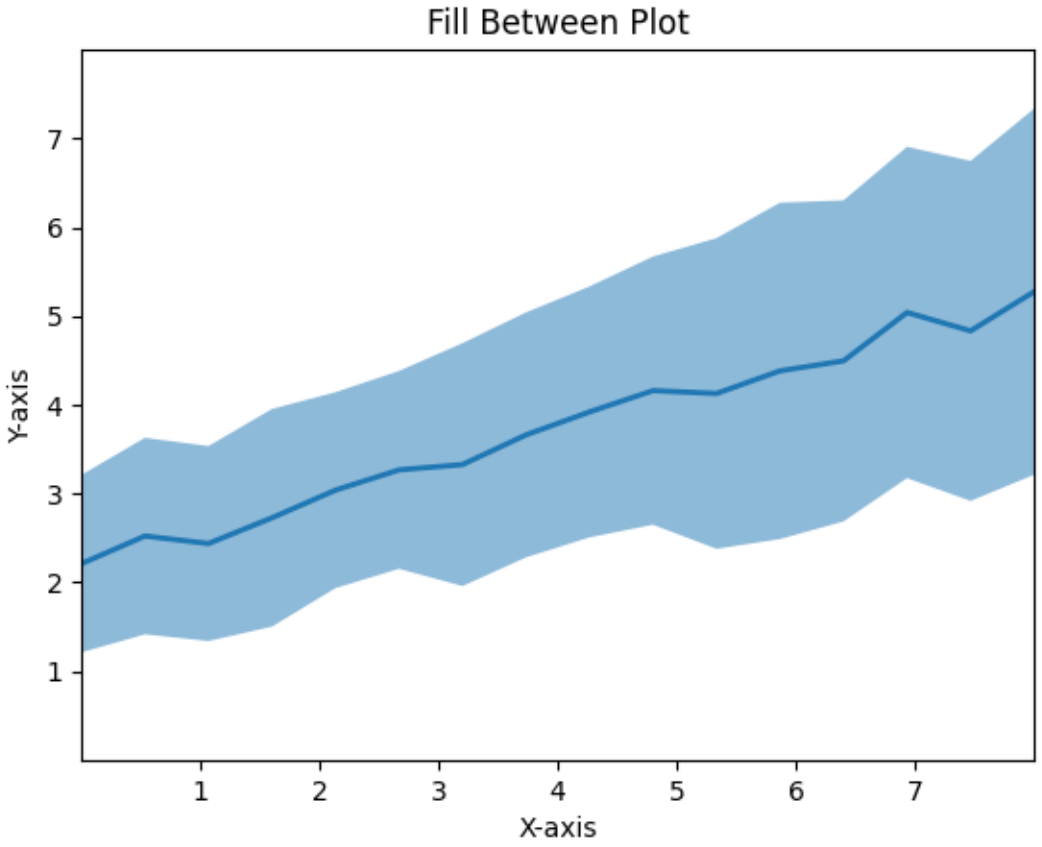


1.1.5 Fill between plot

This function fills the area between two curves defined by y1 and y2 data points (lists or arrays) on the x-axis specified by x. It's useful for visualizing areas under curves or representing distributions.

[8]:
```
np.random.seed(1)
x = np.linspace(0, 8, 16)
y1 = 3 + 4*x/8 + np.random.uniform(0.0, 0.5, len(x))
y2 = 1 + 2*x/8 + np.random.uniform(0.0, 0.5, len(x))
```

```
fig, ax = plt.subplots()
ax.fill_between(x, y1, y2, alpha=.5, linewidth=0)
ax.plot(x, (y1 + y2)/2, linewidth=2)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Fill Between Plot')
plt.show()
```

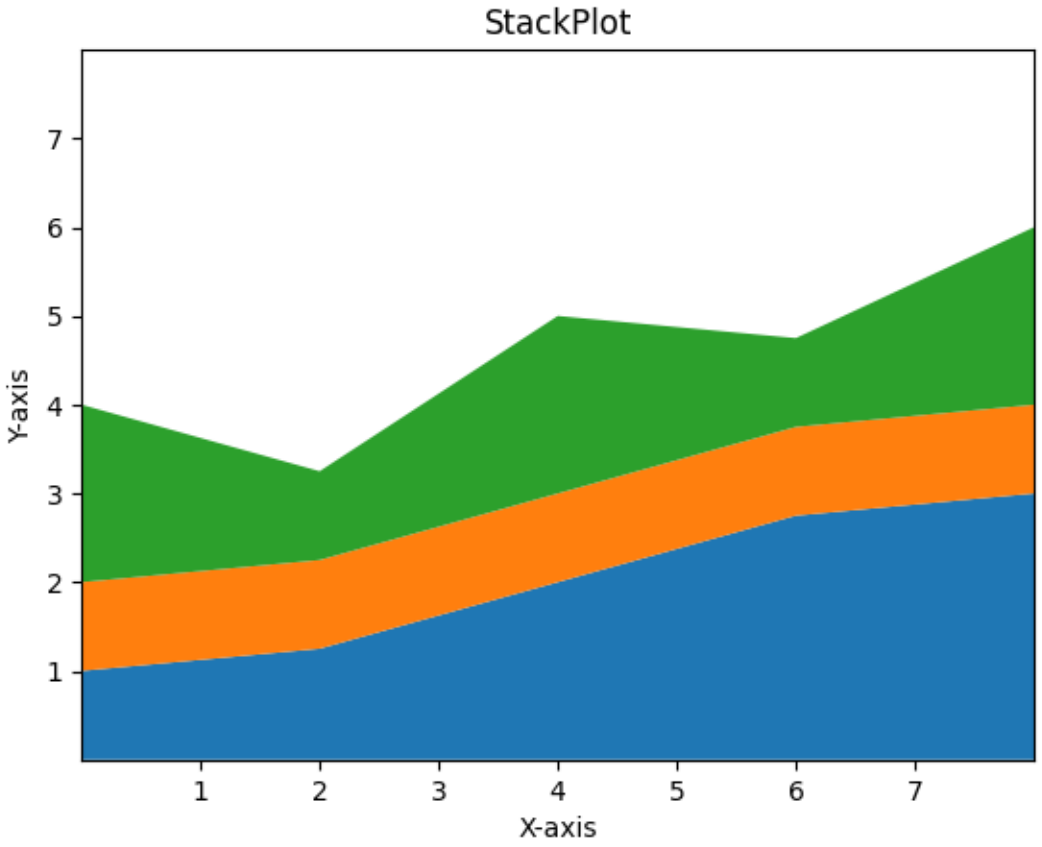


1.1.6 Stackplot

This function creates a stacked bar chart. It takes x for the positions on the x-axis and multiple lists or arrays within y for the heights of each stacked bar segment. Each list in y represents a layer on the bar at each x position.

[9]:
```
x = np.arange(0, 10, 2)
ay = [1, 1.25, 2, 2.75, 3]
by = [1, 1, 1, 1, 1]
cy = [2, 1, 2, 1, 2]
```

```
y = np.vstack([ay, by, cy])

fig, ax = plt.subplots()
ax.stackplot(x, y)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('StackPlot')
plt.show()
```

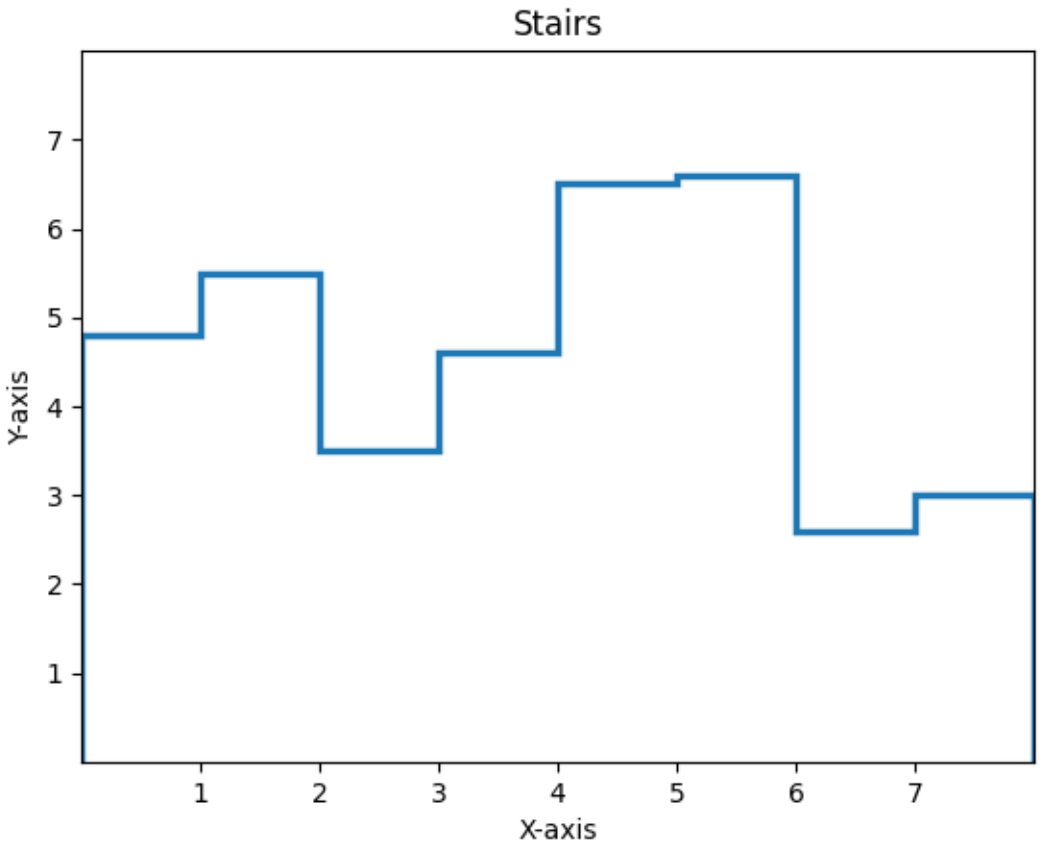


1.1.7 Stairs

This function creates a stairstep plot, which connects data points with horizontal and vertical line segments. It's useful for visualizing data that represents steps or changes over tim

[10]:
```
y = [4.8, 5.5, 3.5, 4.6, 6.5, 6.6, 2.6, 3.0]

fig, ax = plt.subplots()
ax.stairs(y, linewidth=2.5)
```

```
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Stairs')
plt.show()
```



1.2 Statistical Distributions

Plots representing the distribution of one or more variables in a dataset; several of these techniques additionally involve distribution computation.

There many types of plots in Statistical Distribution :

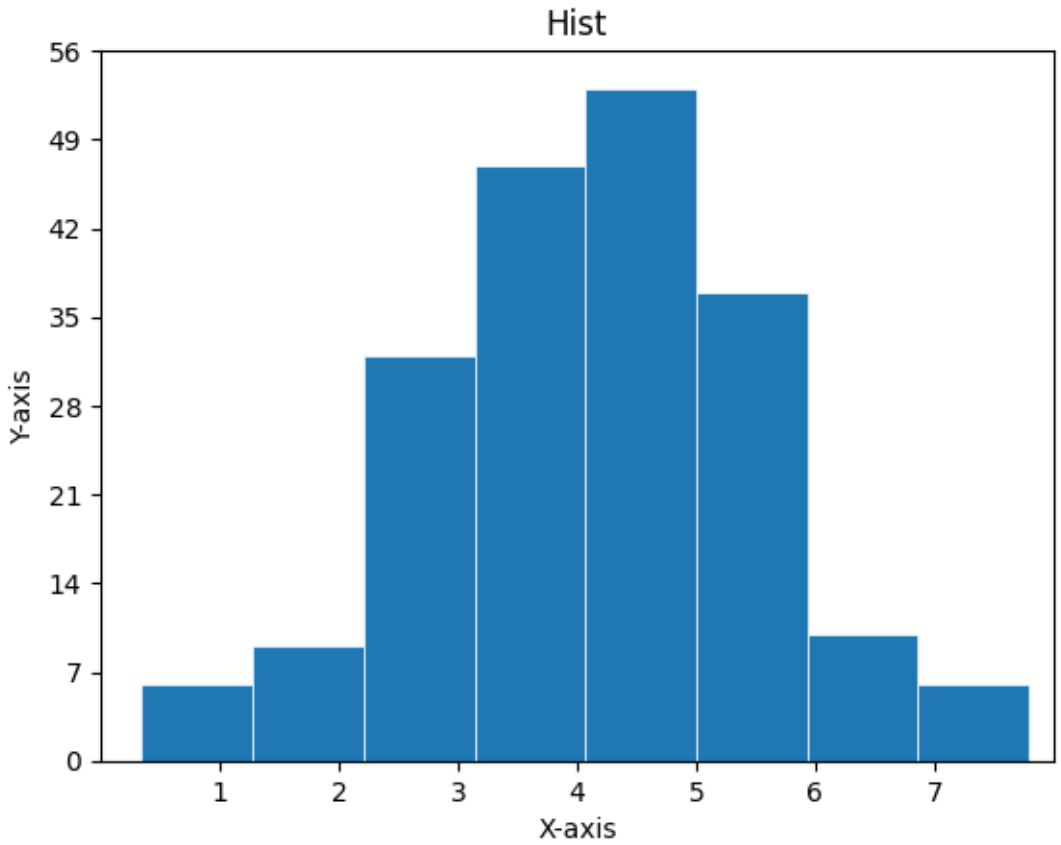
- Hist Plot
- Boxplot
- Errorbar
- Violinplot
- Hist2d
- Hexbin

- Pie
- Ecdf

1.2.1 Hist Plot

It visualizes the distribution of numerical data by dividing it into bins and showing the frequency of data points within each bin.

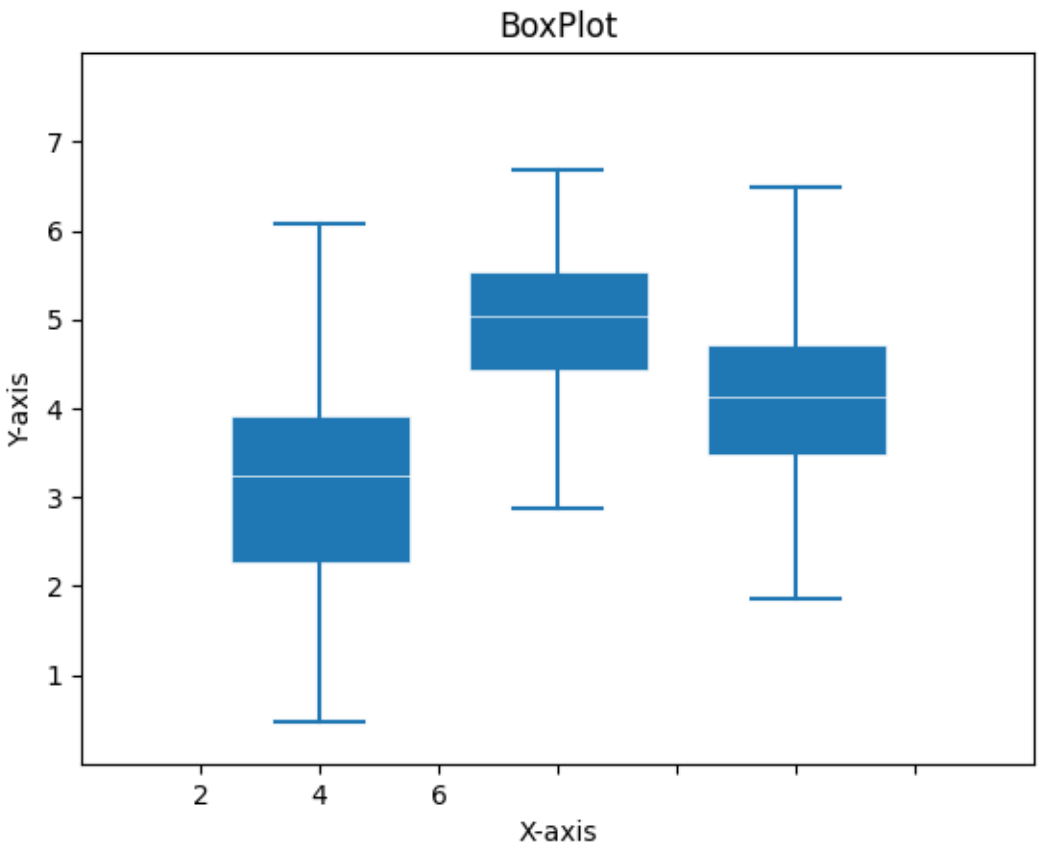
[11]:
```
np.random.seed(1)
x = 4 + np.random.normal(0, 1.5, 200)

fig, ax = plt.subplots()

ax.hist(x, bins=8, linewidth=0.5, edgecolor="white")

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 56), yticks=np.linspace(0, 56, 9))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Hist')
plt.show()
```



1.2.2 Boxplot

This function creates a box plot. It displays the distribution of data with five key statistics: minimum, first quartile, median, third quartile, and maximum. The box represents the interquartile range (IQR) between Q1 and Q3, with whiskers extending to the minimum and maximum values.

[13]:
```
np.random.seed(10)
D = np.random.normal((3, 5, 4), (1.25, 1.00, 1.25), (100, 3))


fig, ax = plt.subplots()
VP = ax.boxplot(D, positions=[2, 4, 6], widths=1.5, patch_artist=True,
                showmeans=False, showfliers=False,
                medianprops={"color": "white", "linewidth": 0.5},
                boxprops={"facecolor": "C0", "edgecolor": "white",
                          "linewidth": 0.5},
                whiskerprops={"color": "C0", "linewidth": 1.5},
                capprops={"color": "C0", "linewidth": 1.5})

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('BoxPlot')
plt.show()
```

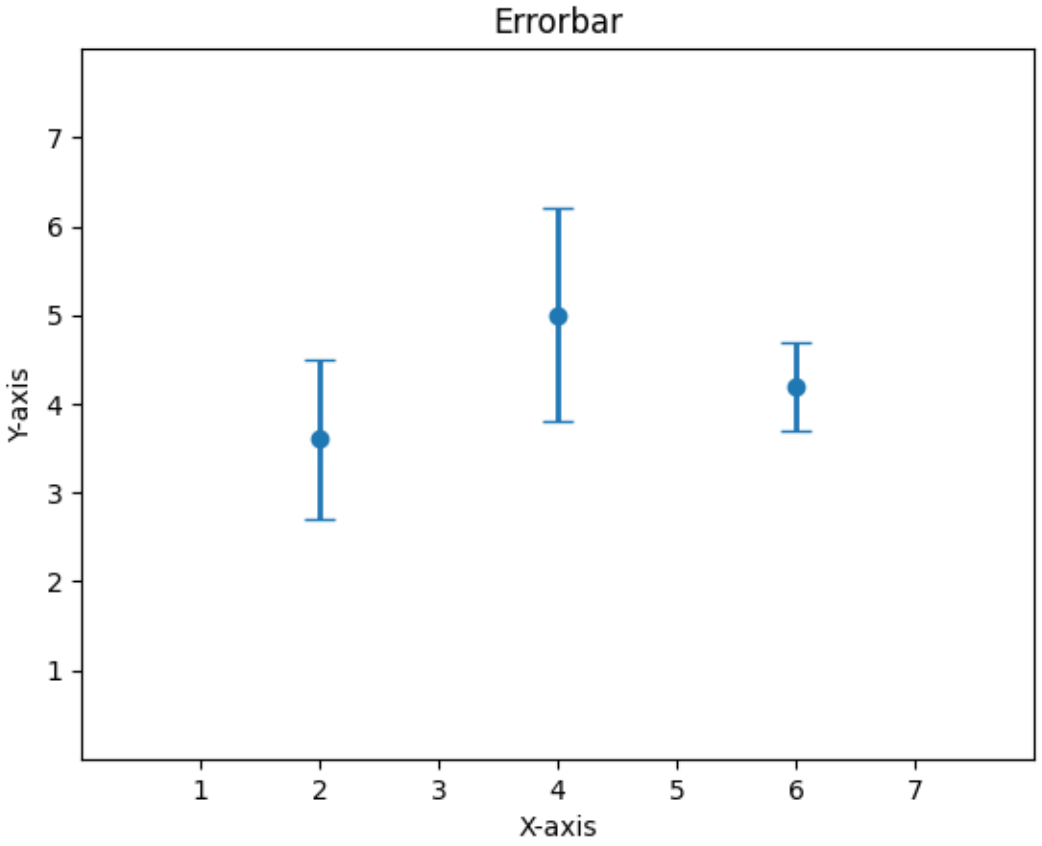


1.2.3 Errorbar

This function creates an error bar plot. It displays data points along with error bars that represent the variability (e.g., standard deviation) associated with each data point. You provide the x and y values, along with optional error values for both x and y (yerr and xerr).

[14]:
```
np.random.seed(1)
x = [2, 4, 6]
y = [3.6, 5, 4.2]
yerr = [0.9, 1.2, 0.5]

fig, ax = plt.subplots()

ax.errorbar(x, y, yerr, fmt='o', linewidth=2, capsize=6)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))

plt.xlabel('X-axis')
plt.ylabel('Y-axis')
```

```
plt.title('Errorbar')
plt.show()
```

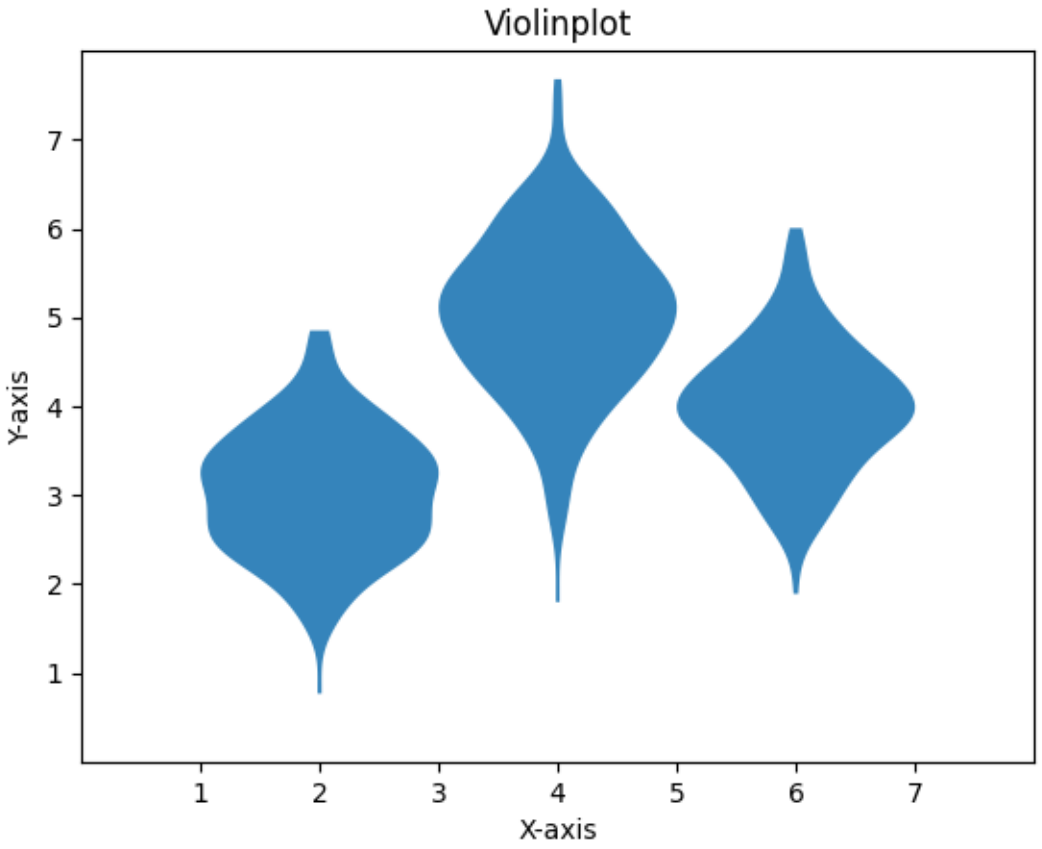


1.2.4 Violinplot

This function creates a violin plot. It combines features of a boxplot and a density plot, showing the distribution of data with a boxplot in the center and the density of data points on either side using violin shapes.

[15]:
```
np.random.seed(10)
D = np.random.normal((3, 5, 4), (0.75, 1.00, 0.75), (200, 3))


fig, ax = plt.subplots()

vp = ax.violinplot(D, [2, 4, 6], widths=2,
                   showmeans=False, showmedians=False, showextrema=False)
for body in vp['bodies']:
    body.set_alpha(0.9)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
```

```
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Violinplot')
plt.show()
```

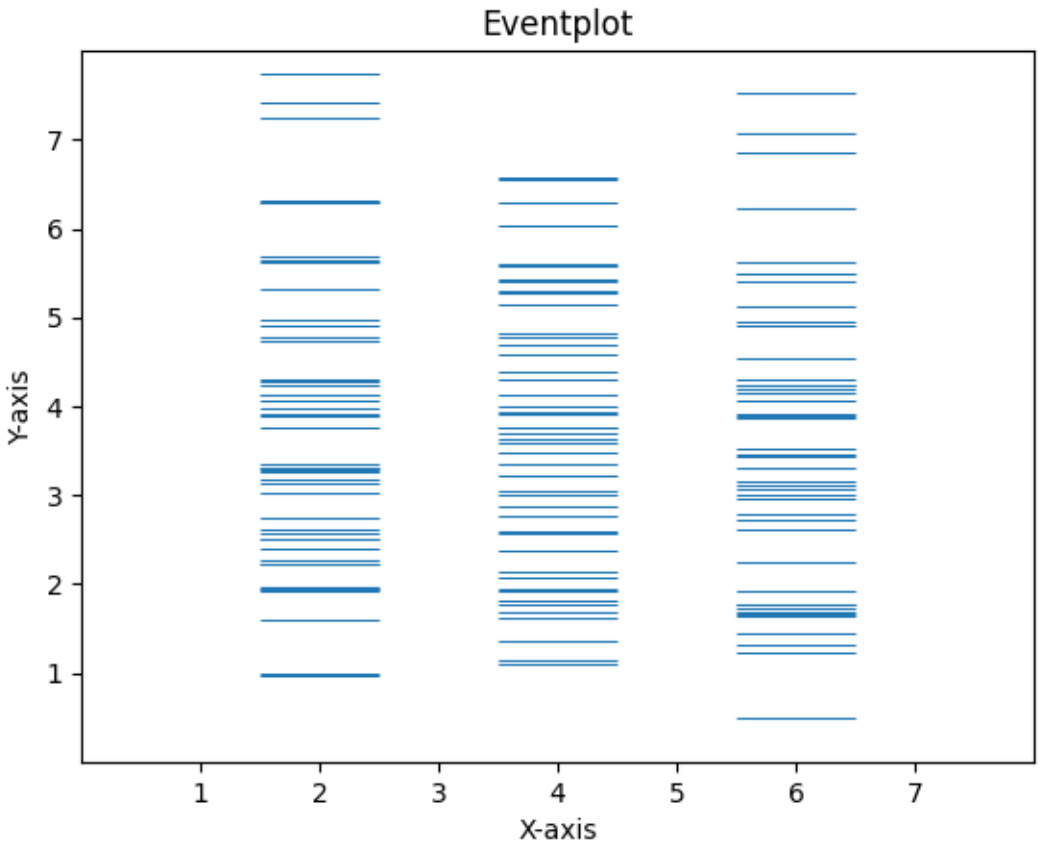


1.2.5 Eventplot

This function (depending on the library) might be used for event plots. These plots visualize categorical data points along a time axis, with markers representing events or occurrences.

[16]:
```
np.random.seed(1)
x = [2, 4, 6]
D = np.random.gamma(4, size=(3, 50))

fig, ax = plt.subplots()

ax.eventplot(D, orientation="vertical", lineoffsets=x, linewidth=0.75)
ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
       ylim=(0, 8), yticks=np.arange(1, 8))
```

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Eventplot')
plt.show()
```

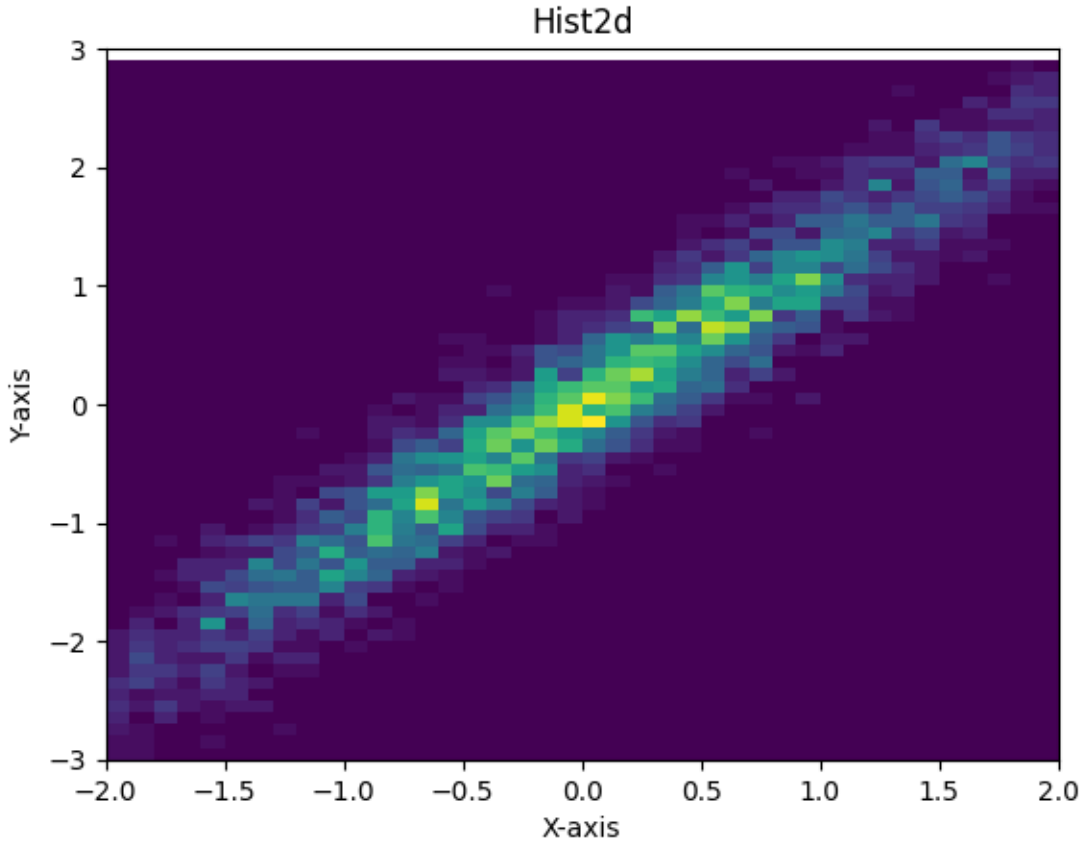


1.2.6 Hist2d

This function creates a 2D histogram. It visualizes the joint distribution of two variables by dividing their space into bins and showing the frequency of data points within each bin. This is useful for exploring relationships between two continuous variables.

[18]:
```
np.random.seed(1)
x = np.random.randn(5000)
y = 1.2 * x + np.random.randn(5000) / 3


fig, ax = plt.subplots()

ax.hist2d(x, y, bins=(np.arange(-3, 3, 0.1), np.arange(-3, 3, 0.1)))
```

```
ax.set(xlim=(-2, 2), ylim=(-3, 3))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Hist2d')

plt.show()
```

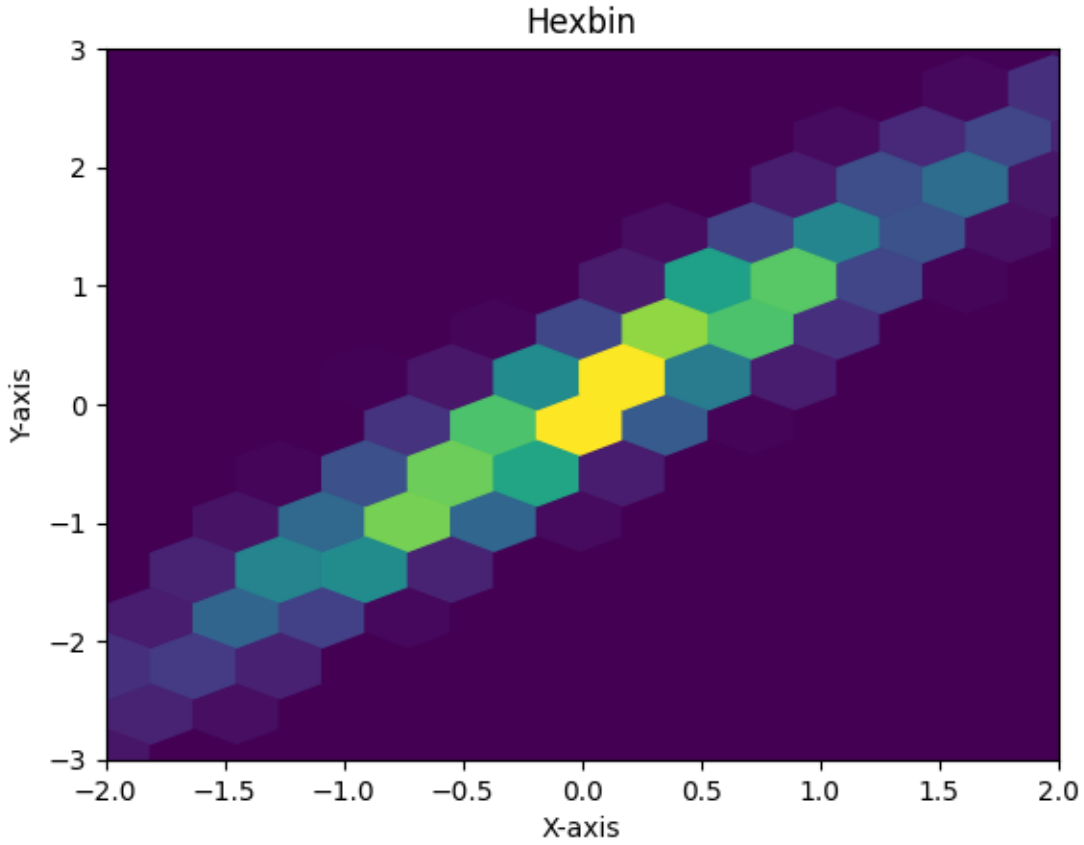


1.2.7 Hexbin

This function creates a hexagonal bin plot. Similar to hist2d, it visualizes the joint distribution of two variables but uses hexagons instead of squares for bins. This can sometimes be a more efficient way to represent dense data points.

[19]:
```
np.random.seed(1)
x = np.random.randn(5000)
y = 1.2 * x + np.random.randn(5000) / 3

fig, ax = plt.subplots()

ax.hexbin(x, y, gridsize=20)
```

```
ax.set(xlim=(-2, 2), ylim=(-3, 3))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Hexbin')
plt.show()
```

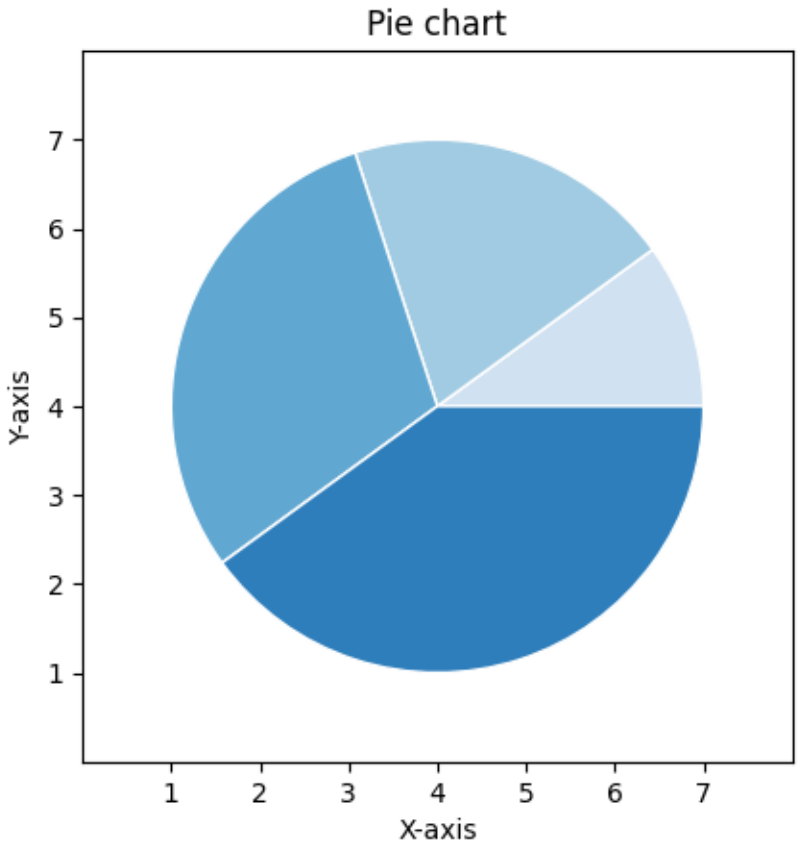


1.2.8 Pie

This function creates a pie chart. It represents a categorical data set where each slice of the pie corresponds to a category and its size reflects the proportion of data points in that category.

[21]:
```
x = [1, 2, 3, 4]
colors = plt.get_cmap('Blues')(np.linspace(0.2, 0.7, len(x)))

fig, ax = plt.subplots()
ax.pie(x, colors=colors, radius=3, center=(4, 4),
       wedgeprops={"linewidth": 1, "edgecolor": "white"}, frame=True)

ax.set(xlim=(0, 8), xticks=np.arange(1, 8),
```

```
       ylim=(0, 8), yticks=np.arange(1, 8))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Pie chart')
plt.show()
```

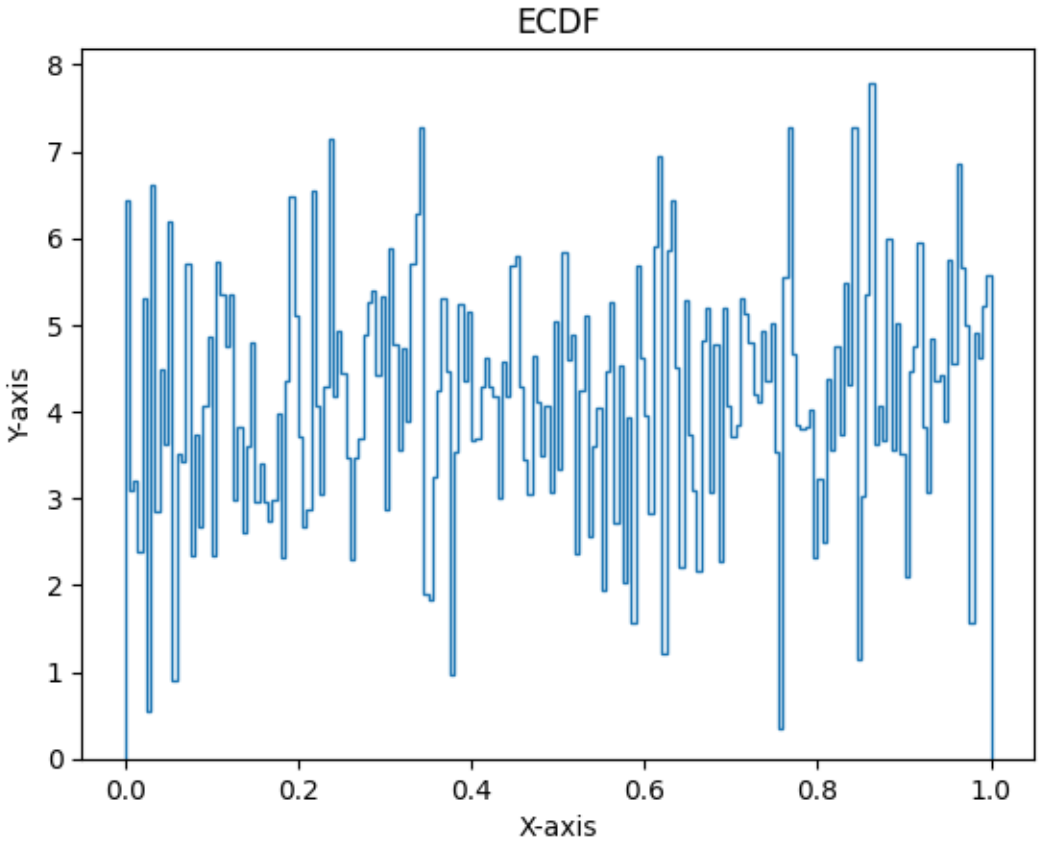


1.2.9 Ecdf

This function (depending on the library) might be used for the empirical cumulative distribution function (ECDF). It plots the probability that a data point in the set will be less than or equal to a certain value.

[26]:
```
np.random.seed(1)
x = 4 + np.random.normal(0, 1.5, 200)

fig, ax = plt.subplots()
ax.stairs(x, np.arange(len(x) + 1) / len(x))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('ECDF')
```

```
plt.show()
```



1.2.10 Now let's see some advance plots

1.3 Gridded data

Gridded data is a specific way of organizing and representing information on a two-dimensional surface

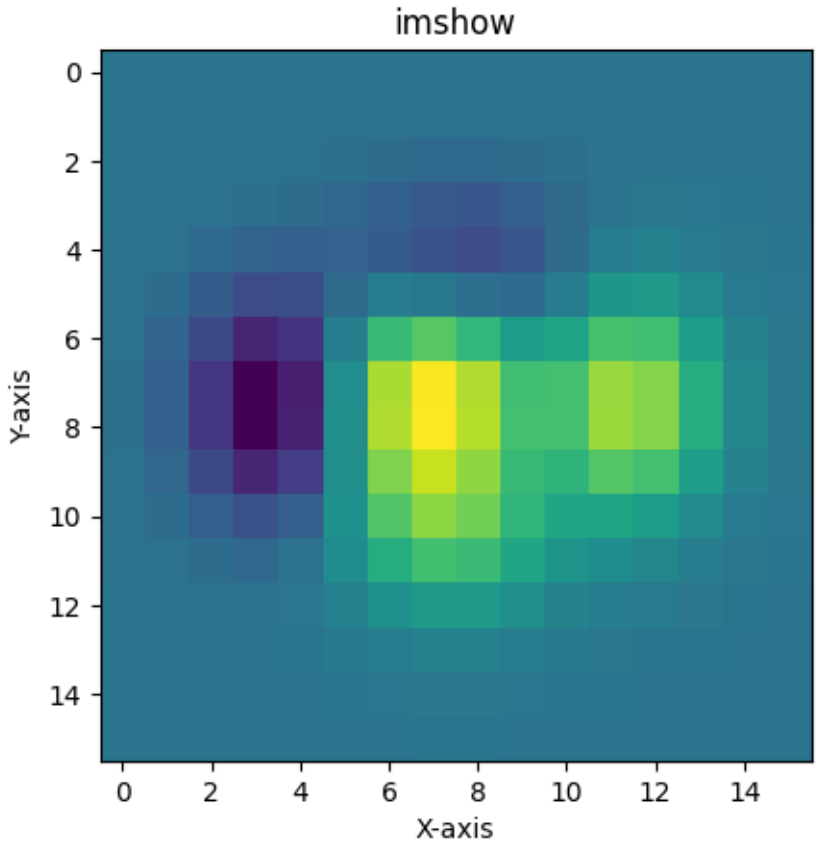
There many types of plots in Gridded data :

- imshow
- pcolormesh
- contour
- contourf
- barbs
- quiver
- streamplot

1.3.1 Imshow

This function, commonly used in Matplotlib, is used to display a 2D image. It assumes Z is a 2D array representing the image data, where each element corresponds to a pixel's intensity or color value.

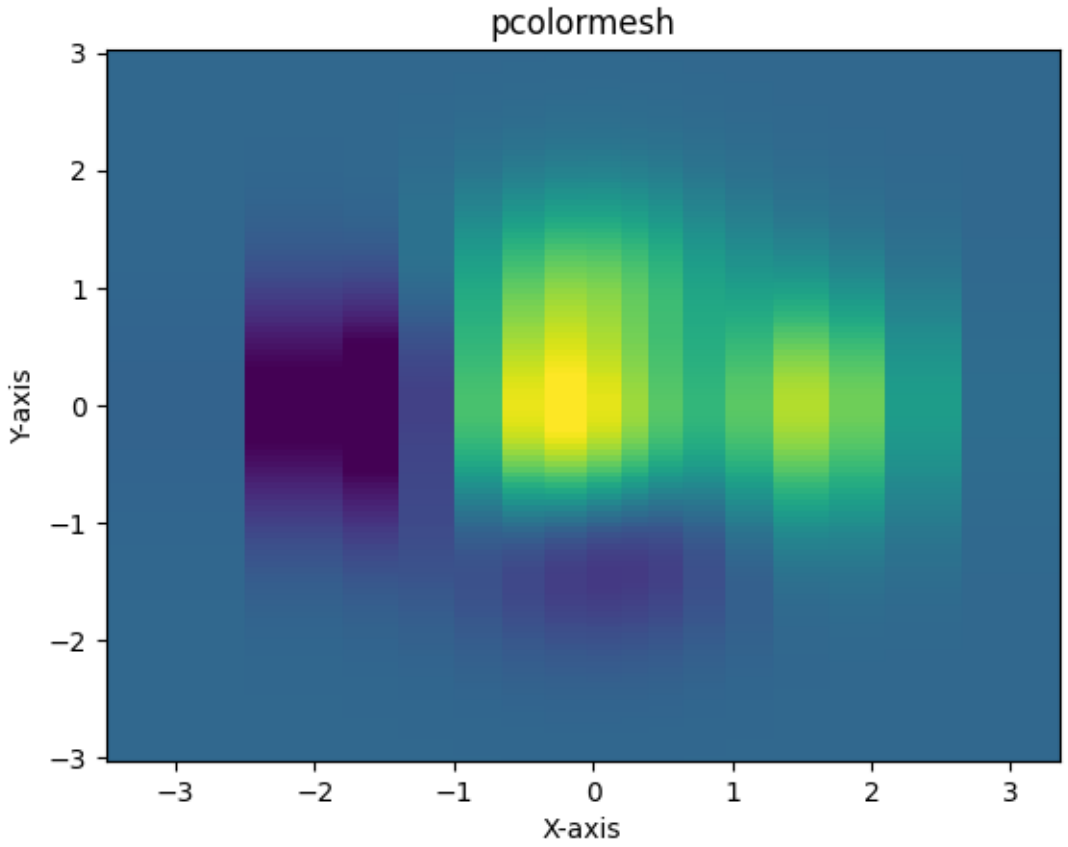
[40]:
```
X, Y = np.meshgrid(np.linspace(-3, 3, 16), np.linspace(-3, 3, 16))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

fig, ax = plt.subplots()

ax.imshow(Z)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('imshow')

plt.show()
```



1.3.2 Pcolormesh

This function creates a pseudocolor mesh plot. Similar to imshow, it visualizes a 2D data set using a colormap to represent values in Z. However, pcolormesh treats the data as if it's defined on cell centers rather than individual pixels, which can be useful for certain types of gridded data.

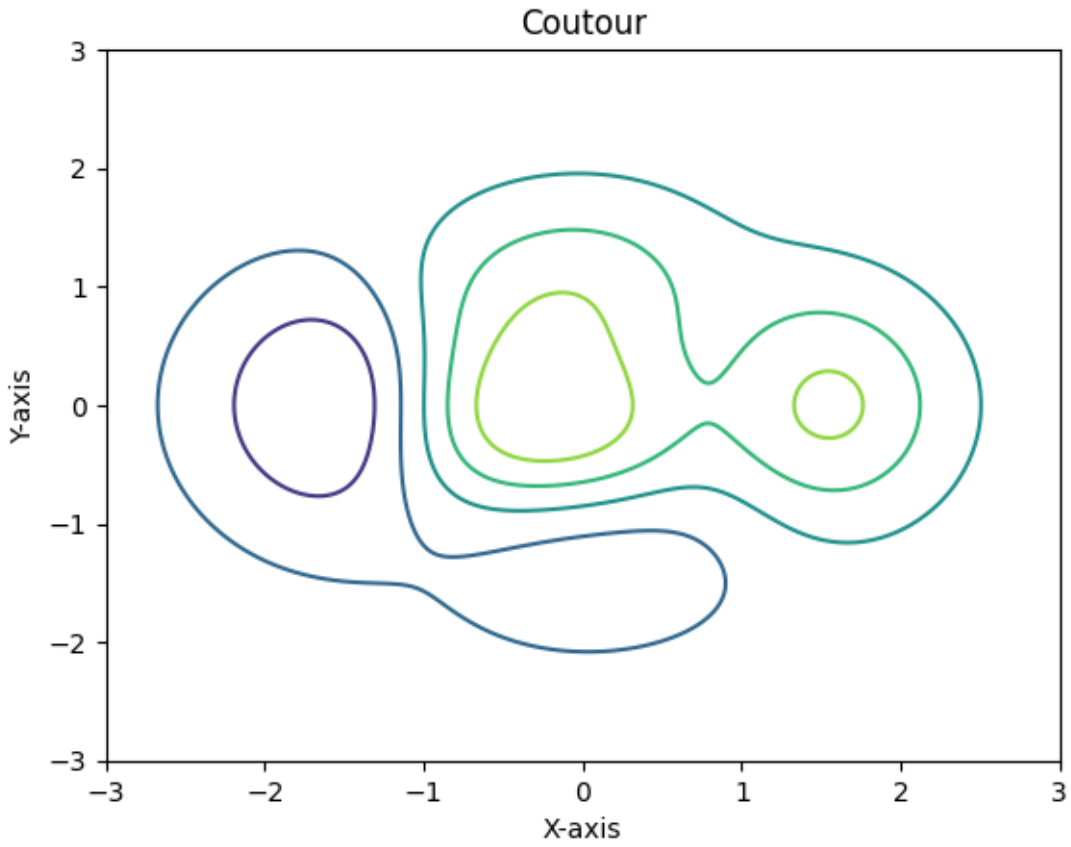
[41]:
```
x = [-3, -2, -1.6, -1.2, -.8, -.5, -.2, .1, .3, .5, .8, 1.1, 1.5, 1.9, 2.3, 3]
X, Y = np.meshgrid(x, np.linspace(-3, 3, 128))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)


fig, ax = plt.subplots()

ax.pcolormesh(X, Y, Z, vmin=-0.5, vmax=1.0)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('pcolormesh')

plt.show()
```



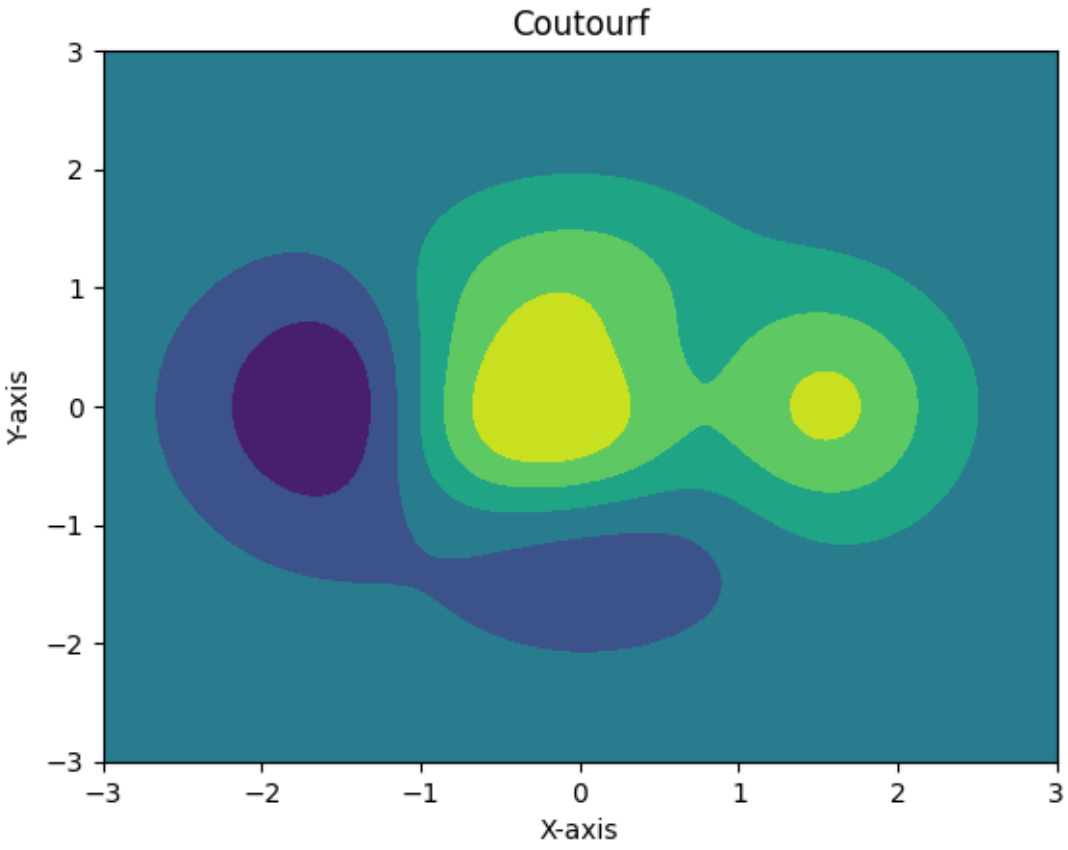
1.3.3 Contour and Contourf

These functions create contour plots. They visualize level sets of a 2D function (represented by Z) by drawing lines (contour) or filled regions (contourf) where the function value is constant. These are helpful for understanding the shape and variations of a function over a grid.

[42]:
```
X, Y = np.meshgrid(np.linspace(-3, 3, 256), np.linspace(-3, 3, 256))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)
levels = np.linspace(np.min(Z), np.max(Z), 7)

fig, ax = plt.subplots()

ax.contour(X, Y, Z, levels=levels)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Coutour')

plt.show()
```



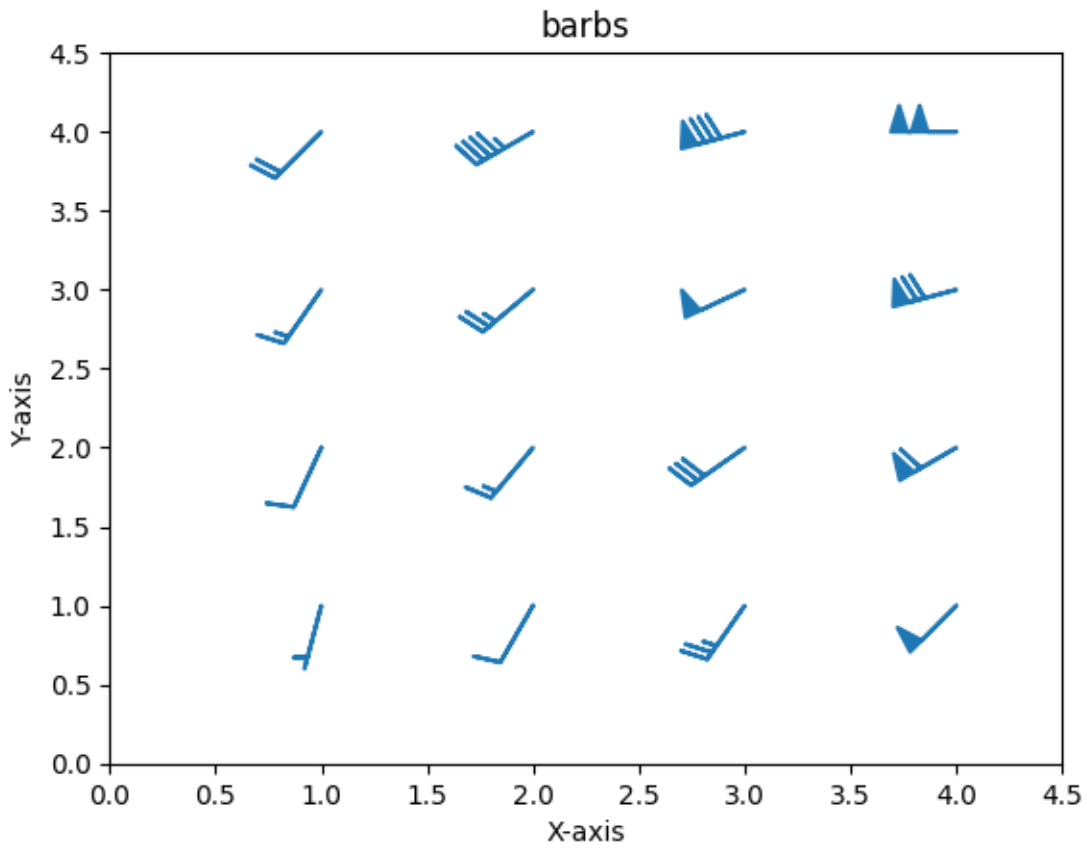
[43]:
```
X, Y = np.meshgrid(np.linspace(-3, 3, 256), np.linspace(-3, 3, 256))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)
levels = np.linspace(Z.min(), Z.max(), 7)


fig, ax = plt.subplots()

ax.contourf(X, Y, Z, levels=levels)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Coutourf')

plt.show()
```



1.3.4 Barbs

This function creates a barb plot. It displays arrows (barbs) at specific locations (X, Y) to represent vector data. The arrow direction indicates the direction of the vector (often velocity or wind), and the arrow length can represent the vector's magnitude.

[47]:
```
X, Y = np.meshgrid([1, 2, 3, 4], [1, 2, 3, 4])
angle = np.pi / 180 * np.array([[15., 30, 35, 45],
                                [25., 40, 55, 60],
                                [35., 50, 65, 75],
                                [45., 60, 75, 90]])
amplitude = np.array([[5, 10, 25, 50],
                      [10, 15, 30, 60],
                      [15, 26, 50, 70],
                      [20, 45, 80, 100]])
U = amplitude * np.sin(angle)
V = amplitude * np.cos(angle)


fig, ax = plt.subplots()

ax.barbs(X, Y, U, V, barbcolor='C0', flagcolor='C0', length=7, linewidth=1.5)

ax.set(xlim=(0, 4.5), ylim=(0, 4.5))
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('barbs')

plt.show()
```

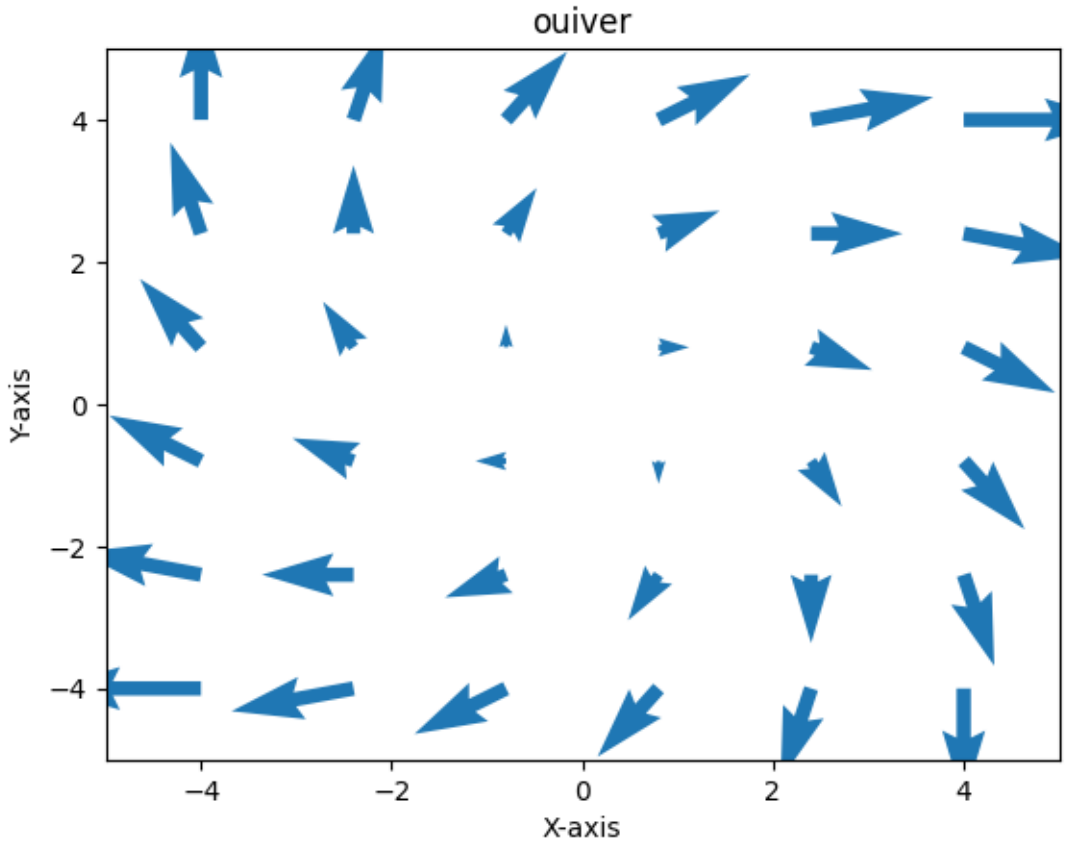


1.3.5 Quiver

Similar to barbs, this function creates a quiver plot. It displays arrows at specified locations (X, Y) to represent vector data, but with more control over arrow properties like head size, color, and scaling.

[48]:
```
x = np.linspace(-4, 4, 6)
y = np.linspace(-4, 4, 6)
X, Y = np.meshgrid(x, y)
U = X + Y
V = Y - X


fig, ax = plt.subplots()

ax.quiver(X, Y, U, V, color="C0", angles='xy',
          scale_units='xy', scale=5, width=.015)

ax.set(xlim=(-5, 5), ylim=(-5, 5))
plt.xlabel('X-axis')
```

```
plt.ylabel('Y-axis')
plt.title('ouiver')

plt.show()
```



1.3.6 Streamplot

This function creates a stream plot. It visualizes a 2D vector field (U, V) using arrows that are connected to form streamlines. These streamlines depict the flow or direction of the vector field.

[49]:
```
X, Y = np.meshgrid(np.linspace(-3, 3, 256), np.linspace(-3, 3, 256))
Z = (1 - X/2 + X**5 + Y**3) * np.exp(-X**2 - Y**2)

V = np.diff(Z[1:, :], axis=1)
U = -np.diff(Z[:, 1:], axis=0)

fig, ax = plt.subplots()

ax.streamplot(X[1:, 1:], Y[1:, 1:], U, V)
```

```
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('streamplot')

plt.show()
```