**UNIT-2:**

**Displaying Statistical Datasets**

# Import standard packages

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

import scipy.stats as stats

import seaborn as sns

# NumPy in Python

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays.

It is the fundamental package for scientific computing with Python.

#### Arrays in Numpy

Array in Numpy is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy, number of dimensions of the array is called rank of the array.A tuple of integers giving the size of the array along each dimension is known as shape of the array. An array class in Numpy is called as **ndarray**. Elements in Numpy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.

import numpy as np

# Creating a rank 1 Array

arr = np.array([1, 2, 3])

print("Array with Rank 1: \n",arr)

# Creating a rank 2 Array

arr = np.array([[1, 2, 3],

[4, 5, 6]])

print("Array with Rank 2: \n", arr)

**output:**

Array with Rank 1:

[1 2 3]

Array with Rank 2:

[[1 2 3]

[4 5 6]]

**numpy.arange()**

The Numpy arange function (sometimes called np.arange) is a tool for creating numeric sequences in Python.

**Syntax:**

**np.arange( start , stop, step, dtype)**

**start :** [optional] start of interval range. By default start = 0

**stop :** end of interval range

**step :** [optional] step size of interval. By default step size = 1,

**dtype :** type of output array

**Example:**

import numpy as np

x=np.arange(6)

print(x)

**Output :** [0 1 2 3 4 5]

x=np.arange(2,10)

print(x)

**Output :** [2 3 4 5 6 7 8 9]

x=np.arange(2,20,3)

print(x)

**Output:** [ 2 5 8 11 14 17]

x=np.arange(6).reshape(2,3)

print(x)

**Output:**

[[0 1 2]

[3 4 5]]

**numpy.random.randn():**

It creates an array of specified shape and fills it with random values as per **standard normal** distribution.

**syntax: numpy.random.randn(d0, d1, …, dn)**

**Parameters :**

**d0, d1, ..., dn :** [int, optional]Dimension of the returned array we require,

If no argument is given a single Python float is returned.

**Return :**

Array of defined shape, filled with random floating-point samples from

the standard normal distribution.

**Example:**

import numpy as np

x = np.random.randn(5)

print(x)

**output:**

[ 0.25355723 -0.87898335 -1.19984135 0.88814022 -1.90335088]

**Example:**

import numpy as np

x = np.random.randn(3, 4)

print(x)

**output:**

[[-2.40545012 -2.02718914 -1.17641764 1.91650802]

[-2.03143066 -0.25081453 2.0769773 -0.85822511]

[ 0.56468827 -1.44469774 -0.15541205 0.66830447]]

**Example:**

import numpy as np

x = np.random.randn(2, 3, 3)

print(x)

**output:**

[[[-0.61346234 -0.59524987 0.70510714]

[ 0.70383491 1.04870392 -1.08468406]

[-0.88573309 2.73599984 -1.22141965]]

[[ 0.63960706 -0.10799721 0.7946028 ]

[ 0.9124436 1.09047532 0.16501142]

[ 2.58185541 0.25913597 -0.88391828]]]

## ****MatPlotLib****

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.

Matplotlib consists of several plots like line, bar, scatter, histogram etc.

Plots help to understand trends, patterns, and to make correlations. They’re typically instruments for reasoning about quantitative information.

Advantages of Matplotlib:

1. It could be used on any operating system via its array of backend.
2. It has a familiar interface: one similar to MatLab it had a coherent vision: to do 2D graphics, and do them well.
3. Great control of every element in a figure, including figure size and DPI.
4. High-quality output in many formats, including PNG, PDF, SVG, EPS, and PGF.

**Importing matplotlib:**

from matplotlib import pyplot as plt

or

import matplotlib.pyplot as plt

**pyplot()** is the most important function in matplotlib library, which is used to plot 2D data.

Plt.show() function is used to display a figure.

**Example:**

from matplotlib import pyplot as plt

# x-axis values

x = [5, 2, 9, 4, 7]

# Y-axis values

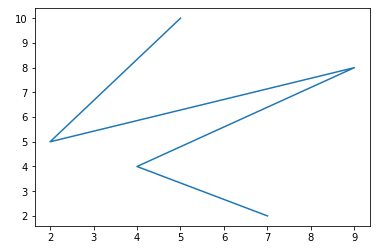
y = [10, 5, 8, 4, 2]

# Function to plot

plt.plot(x,y)

# function to show the plot

plt.show()



Instead of the linear graph, the values can be displayed discretely by adding a format string to the **plot()** function. Following formatting characters can be used.

|  |  |
| --- | --- |
| **Sr.No.** | **Character & Description** |
| 1 | **'-'**  Solid line style |
| 2 | **'--'**  Dashed line style |
| 3 | **'-.'**  Dash-dot line style |
| 4 | **':'**  Dotted line style |
| 5 | **'.'**  Point marker |
| 6 | **','**  Pixel marker |
| 7 | **'o'**  Circle marker |
| 8 | **'v'**  Triangle\_down marker |
| 9 | **'^'**  Triangle\_up marker |
| 10 | **'<'**  Triangle\_left marker |
| 11 | **'>'**  Triangle\_right marker |
| 12 | **'1'**  Tri\_down marker |
| 13 | **'2'**  Tri\_up marker |
| 14 | **'3'**  Tri\_left marker |
| 15 | **'4'**  Tri\_right marker |
| 16 | **'s'**  Square marker |
| 17 | **'p'**  Pentagon marker |
| 18 | **'\*'**  Star marker |
| 19 | **'h'**  Hexagon1 marker |
| 20 | **'H'**  Hexagon2 marker |
| 21 | **'+'**  Plus marker |
| 22 | **'x'**  X marker |
| 23 | **'D'**  Diamond marker |
| 24 | **'d'**  Thin\_diamond marker |
| 25 | **'|'**  Vline marker |
| 26 | **'\_'**  Hline marker |

The following color abbreviations are also defined.

|  |  |
| --- | --- |
| **Character** | **Color** |
| 'b' | Blue |
| 'g' | Green |
| 'r' | Red |
| 'c' | Cyan |
| 'm' | Magenta |
| 'y' | Yellow |
| 'k' | Black |
| 'w' | White |

To display the circles representing points, instead of the line in the above example, use **“ob”** as the format string in plot() function.

### Example

import numpy as np

from matplotlib import pyplot as plt

x = np.arange(1,11)

y = 2 \* x + 5

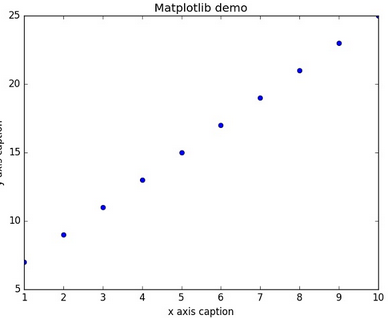
plt.title("Matplotlib demo")

plt.xlabel("x axis caption")

plt.ylabel("y axis caption")

plt.plot(x,y,"ob")

plt.show()



## Sine Wave Plot

The following script produces the **sine wave plot** using matplotlib.

### Example

import numpy as np

import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on a sine curve

x = np.arange(0, 3 \* np.pi, 0.1)

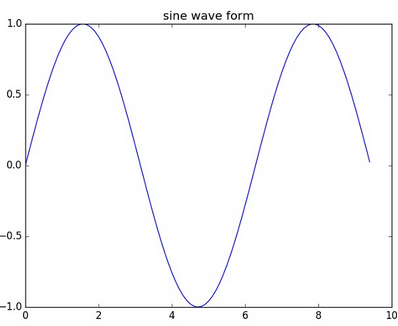
y = np.sin(x)

plt.title("sine wave form")

# Plot the points using matplotlib

plt.plot(x, y)

plt.show()



## subplot()

The subplot() function allows you to plot different things in the same figure. In the following script, **sine** and **cosine values** are plotted.

### Example

import numpy as np

import matplotlib.pyplot as plt

# Compute the x and y coordinates for points on sine and cosine curves

x = np.arange(0, 3 \* np.pi, 0.1)

y\_sin = np.sin(x)

y\_cos = np.cos(x)

# Set up a subplot grid that has height 2 and width 1,

# and set the first such subplot as active.

plt.subplot(2, 1, 1)

# Make the first plot

plt.plot(x, y\_sin)

plt.title('Sine')

# Set the second subplot as active, and make the second plot.

plt.subplot(2, 1, 2)

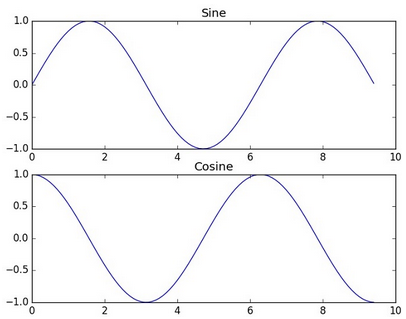
plt.plot(x, y\_cos)

plt.title('Cosine')

# Show the figure.

plt.show()

The above code should produce the following output −



## bar()

The **pyplot submodule** provides **bar()** function to generate bar graphs. The following example produces the bar graph of two sets of **x** and **y** arrays.

### Example

from matplotlib import pyplot as plt

x = [5,8,10]

y = [12,16,6]

x2 = [6,9,11]

y2 = [6,15,7]

plt.bar(x, y, align = 'center')

plt.bar(x2, y2, color = 'g', align = 'center')

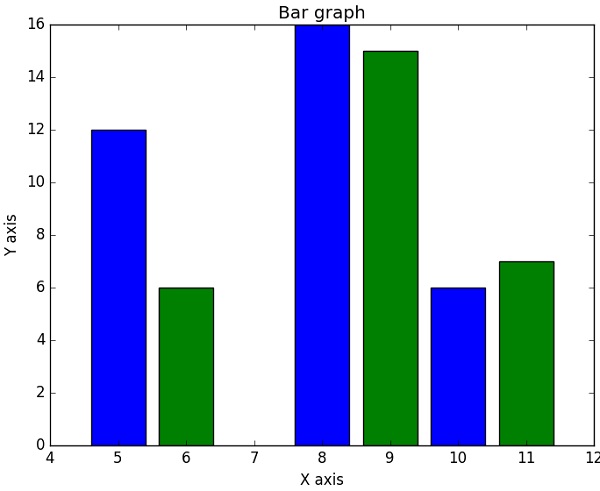
plt.title('Bar graph')

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.show()

This code should produce the following output −



***Univariate Data***

**Univariate** means "one variable" (one type of data)

### Example: Travel Time (minutes): 15, 29, 8, 42, 35, 21, 18, 42, 26

The variable is **Travel Time**

### Example: Puppy Weights

You weigh the pups and get these results:

**2.5, 3.5, 3.3, 3.1, 2.6, 3.6, 2.4**

The variable is **Puppy Weight**

We can do lots of things with univariate data:

* Find a central value using [mean](https://www.mathsisfun.com/mean.html), [median](https://www.mathsisfun.com/median.html) and [mode](https://www.mathsisfun.com/mode.html)
* Find how spread out it is using [range](https://www.mathsisfun.com/data/range.html), [quartiles](https://www.mathsisfun.com/data/quartiles.html) and [standard deviation](https://www.mathsisfun.com/data/standard-deviation.html)
* Make plots like [Bar Graphs](https://www.mathsisfun.com/data/bar-graphs.html), [Pie Charts](https://www.mathsisfun.com/data/pie-charts.html) and [Histograms](https://www.mathsisfun.com/data/histograms.html)

**Bivariate** means "two variables", in other words there are two types of data

With bivariate data we have **two** sets of related data we want to **compare**:

### Example: Sales vs Temperature

An ice cream shop keeps track of how much ice cream they sell versus the temperature on that day.

So with bivariate data we are interested in **comparing** the two sets of data and finding any **relationships**.

We can use Tables, [Scatter Plots](https://www.mathsisfun.com/data/scatter-xy-plots.html), [Correlation](https://www.mathsisfun.com/data/correlation.html), Line of Best Fit, and plain old common sense.

The following examples all have the same format. Only the “Plot-command” line changes.

# Import standard packages

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

import scipy.stats as stats

import seaborn as sns

# Generate the data

x = np.random.randn(500)

# Plot-command start ---------------------

plt.plot(x, '.')

# Plot-command end -----------------------

# Show plot

plt.show()

import numpy as np

from matplotlib import pyplot as plt

1. **Scatter Plots**

A **scatter plot** is a two-dimensional data visualization that uses dots to represent the values obtained for two different variables - one plotted along the x-axis and the other plotted along the y-axis

Scatter plots are used when you want to show the relationship between two variables. Scatter plots are sometimes called correlation plots because they show how two variables are [correlated](https://en.wikipedia.org/wiki/Correlation_and_dependence).

matplotlib.pyplot.scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, \*, data=None, \*\*kwargs)

**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 [rcParams["axes.prop\_cycle"]](https://matplotlib.org/tutorials/introductory/customizing.html#matplotlib-rcparams).

**marker** : [MarkerStyle](https://matplotlib.org/api/_as_gen/matplotlib.markers.MarkerStyle.html#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 [rcParams["scatter.marker"]](https://matplotlib.org/tutorials/introductory/customizing.html#matplotlib-rcparams) = 'o'. See [markers](https://matplotlib.org/api/markers_api.html#module-matplotlib.markers) for more information about marker styles.

**cmap** : [Colormap](https://matplotlib.org/api/_as_gen/matplotlib.colors.Colormap.html#matplotlib.colors.Colormap), optional, default: None

A [Colormap](https://matplotlib.org/api/_as_gen/matplotlib.colors.Colormap.html#matplotlib.colors.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** : [Normalize](https://matplotlib.org/api/_as_gen/matplotlib.colors.Normalize.html#matplotlib.colors.Normalize), optional, default: None

A [Normalize](https://matplotlib.org/api/_as_gen/matplotlib.colors.Normalize.html#matplotlib.colors.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](https://matplotlib.org/api/_as_gen/matplotlib.colors.Normalize.html#matplotlib.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** : color or sequence of color, optional, default: 'face'

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 matplotib color.

For non-filled markers, the *edgecolors* kwarg is ignored and forced to 'face' internally.

**Returns**: **paths** : [PathCollection](https://matplotlib.org/api/collections_api.html#matplotlib.collections.PathCollection)

For example this scatter plot shows the height and weight of a fictitious set of children.

import matplotlib.pyplot as plt

weight1=[63.3,57,64.3,63,71,61.8,62.9,65.6,64.8,63.1,68.3,69.7,65.4,66.3,60.7]

height1=[156.3,100.7,114.8,156.3,237.1,123.9,151.8,164.7,105.4,136.1,175.2,137.4,164.2,151,124.3]

plt.scatter(weight1,height1,c='b',marker='o')

plt.xlabel('weight', fontsize=16)

plt.ylabel('height', fontsize=16)

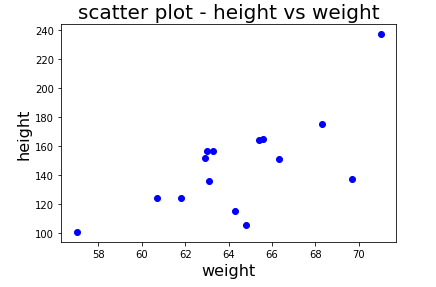
plt.title('scatter plot - height vs weight',fontsize=20)

plt.show()

**Explanation:**

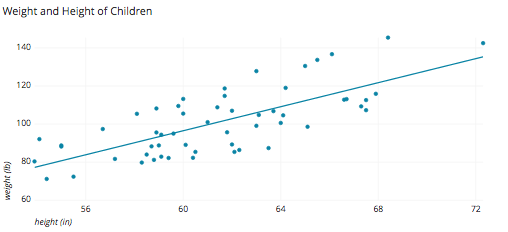
**Line 1:** Imports the pyplot function of matplotlib library in the name of plt.  
**Line 3 and Line 4:** Inputs the arrays to the variables named weight1 and height1.  
**Line 6:** scatter function which takes takes x axis (weight1) as first argument, y axis (height1) as second argument, colour is chosen as blue in third argument and marker=’o’ denotes the type of plot, Which is dot in our case.  
**Line 7 and Line 8:** x label and y label with desired font size is created.  
**Line 9 and Line 10:** Mentions the Chart Title with font size and scatter plot is shown.

**Output:**



Different values for markers and their representation is shown below.

|  |  |
| --- | --- |
| Marker | meaning |
| . | point |
| , | pixel |
| o | circle |
| v | triangle\_down |
| ^ | triangle\_up |
| 8 | octagon |
| s | sqaure |
| p | pentagon |
| \* | star |
| h | hexagon |
| + | plus |
| D | diamond |



Each dot represents one child with his or her height measured along the x-axis and weight measured along the y-axis.

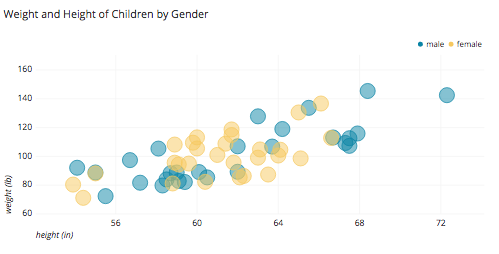
## When to Use Scatter Plots

In the height and weight example, the chart wasn’t just a simple log of the height and weight of a set of children, but it also visualized the relationship between height and weight - namely that weight increases as height increases. Notice that the relationship isn’t perfect, some taller children weight less than some shorter children, but the general trend is pretty strong and we can see that weight is correlated with height.

Several advanced visualization tools allow for more complex visualizations.

Often scatter plots will include a trend line to help make the relationship more clear.

Additionally, the size, shape or color of the dot could represents a third (or even fourth variable). For example, this chart shows the height and weight data but adds in the information of the gender of the child as the color of the dot.



Other:

A scatter plot can be used either when one continuous variable that is under the control of the experimenter and the other depends on it or when both continuous variables are independent.

If a [parameter](https://en.wikipedia.org/wiki/Parameter) exists that is systematically incremented and/or decremented by the other, it is called the *control parameter* or [independent variable](https://en.wikipedia.org/wiki/Independent_variable) and is customarily plotted along the horizontal axis. The measured or [dependent variable](https://en.wikipedia.org/wiki/Dependent_variable) is customarily plotted along the vertical axis. If no dependent variable exists, either type of variable can be plotted on either axis and a scatter plot will illustrate only the degree of [correlation](https://en.wikipedia.org/wiki/Correlation) (not [causation](https://en.wikipedia.org/wiki/Causality)) between two variables.

A scatter plot can suggest various kinds of correlations between variables with a certain [confidence interval](https://en.wikipedia.org/wiki/Confidence_interval). For example, weight and height, weight would be on y axis and height would be on the x axis. Correlations may be positive (rising), negative (falling), or null (uncorrelated). If the pattern of dots slopes from lower left to upper right, it indicates a positive [correlation](https://en.wikipedia.org/wiki/Correlation) between the variables being studied. If the pattern of dots slopes from upper left to lower right, it indicates a negative correlation. A line of [best fit](https://en.wikipedia.org/wiki/Curve_fitting) (alternatively called 'trendline') can be drawn in order to study the relationship between the variables. An equation for the correlation between the variables can be determined by established best-fit procedures. For a linear correlation, the best-fit procedure is known as [linear regression](https://en.wikipedia.org/wiki/Linear_regression) and is guaranteed to generate a correct solution in a finite time. No universal best-fit procedure is guaranteed to generate a correct solution for arbitrary relationships. A scatter plot is also very useful when we wish to see how two comparable data sets agree to show nonlinear relationships between variables. The ability to do this can be enhanced by adding a smooth line such as [LOESS](https://en.wikipedia.org/wiki/Local_regression).[[5]](https://en.wikipedia.org/wiki/Scatter_plot#cite_note-5) Furthermore, if the data are represented by a mixture model of simple relationships, these relationships will be visually evident as superimposed patterns.

The scatter diagram is one of the [seven basic tools](https://en.wikipedia.org/wiki/Seven_Basic_Tools_of_Quality) of [quality control](https://en.wikipedia.org/wiki/Quality_control).[[6]](https://en.wikipedia.org/wiki/Scatter_plot#cite_note-6)

## scatterplot matrices

For a set of data variables (dimensions) X1, X2, ... , Xk, the scatter plot matrix shows all the pairwise scatter plots of the variables on a single view with multiple scatterplots in a matrix format. For k variables, the scatterplot matrix will contain k rows and k columns. A plot located on the intersection of i-th row and j-th column is a plot of variables Xi versus Xj.[[8]](https://en.wikipedia.org/wiki/Scatter_plot#cite_note-8) This means that each row and column is one dimension, and each cell plots a scatterplot of two dimensions.

A **generalized scatterplot matrix**[[9]](https://en.wikipedia.org/wiki/Scatter_plot" \l "cite_note-9) offers a range of displays of paired combinations of categorical and quantitative variables. A [mosaic plot](https://en.wikipedia.org/wiki/Mosaic_plot), [fluctuation diagram](https://en.wikipedia.org/w/index.php?title=Fluctuation_diagram&action=edit&redlink=1), or faceted [bar chart](https://en.wikipedia.org/wiki/Bar_chart) may be used to display two categorical variables. Other plots are used for one categorical and one quantitative variables.

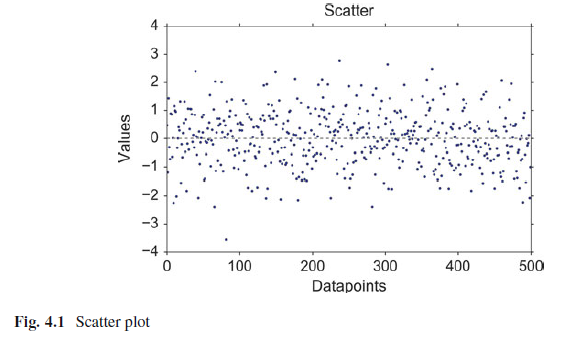
This is the simplest way to represent univariate data: just plot each individual datapoint. The corresponding plot-command is either

plt.plot(x, '.')

or, equivalently,

plt.scatter(np.arange(len(x), x))

Note: In cases where we only have few discrete values on the *x*-axis (e.g.,*Group1, Group2, Group3*), it may be helpful to spread overlapping data points slightly (also referred to as *“adding jitter”)* to show each data point. An examplecan be found at <http://stanford.edu/~mwaskom/software/seaborn/generated/>seaborn.stripplot.html)



1. **Histograms**

A histogram is an accurate graphical representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable (quantitative variable) and was first introduced by Karl Pearson. It is a kind of bar graph. To construct a histogram, the first step is to “bin” the range of values — that is, divide the entire range of values into a series of intervals — and then count how many values fall into each interval. The bins are usually specified as consecutive, non-overlapping intervals of a variable. The bins (intervals) must be adjacent, and are often (but are not required to be) of equal size.

Basically, histograms are used to represent data given in form of some groups. X-axis is about bin ranges where Y-axis talks about frequency.

A histogram is an accurate graphical representation of the distribution of numerical data. It is an estimate of the probability distribution of a continuous variable (quantitative variable) and was first introduced by Karl Pearson.

A histogram is a display of statistical information that uses rectangles to show the frequency of data items in successive numerical intervals of equal size. In the most common form of histogram, the [independent variable](https://whatis.techtarget.com/definition/independent-variable) is plotted along the horizontal axis and the [dependent variable](https://whatis.techtarget.com/definition/dependent-variable) is plotted along the vertical axis. The data appears as colored or shaded rectangles of variable area.

**Syntax:**

matplotlib.pyplot.hist(x, bins=None, range=None, density=None, weights=None, cumulative=False, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=None, log=False, color=None, label=None, stacked=False, normed=None, \*, data=None, \*\*kwargs)

**Parameters:**

**x** : (n,) array or sequence of (n,) arrays

Input values, this takes either a single array or a sequence of arrays which are not required to be of the same length.

**bins** : int or sequence or str, optional

If an integer is given, bins + 1 bin edges are calculated and returned, consistent with [numpy.histogram](https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html#numpy.histogram).

If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

All but the last (righthand-most) bin is half-open. In other words, if bins is:

[1, 2, 3, 4]

then the first bin is [1, 2) (including 1, but excluding 2) and the second [2, 3). The last bin, however, is [3, 4], which *includes* 4.

Unequally spaced bins are supported if *bins* is a sequence.

With Numpy 1.11 or newer, you can alternatively provide a string describing a binning strategy, such as 'auto', 'sturges', 'fd', 'doane', 'scott', 'rice', 'sturges' or 'sqrt', see [numpy.histogram](https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html#numpy.histogram).

The default is taken from [rcParams["hist.bins"]](https://matplotlib.org/tutorials/introductory/customizing.html#matplotlib-rcparams).

**bins** : int or sequence or str, optional

If an integer is given, bins + 1 bin edges are calculated and returned, consistent with [numpy.histogram](https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html#numpy.histogram).

If bins is a sequence, gives bin edges, including left edge of first bin and right edge of last bin. In this case, bins is returned unmodified.

All but the last (righthand-most) bin is half-open. In other words, if bins is:

[1, 2, 3, 4]

then the first bin is [1, 2) (including 1, but excluding 2) and the second [2, 3). The last bin, however, is [3, 4], which *includes* 4.

Unequally spaced bins are supported if *bins* is a sequence.

With Numpy 1.11 or newer, you can alternatively provide a string describing a binning strategy, such as 'auto', 'sturges', 'fd', 'doane', 'scott', 'rice', 'sturges' or 'sqrt', see [numpy.histogram](https://docs.scipy.org/doc/numpy/reference/generated/numpy.histogram.html#numpy.histogram).

The default is taken from [rcParams["hist.bins"]](https://matplotlib.org/tutorials/introductory/customizing.html#matplotlib-rcparams).

**range** : tuple or None, optional

The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, *range* is (x.min(), x.max()). Range has no effect if *bins* is a sequence.

If *bins* is a sequence or *range* is specified, autoscaling is based on the specified bin range instead of the range of x.

Default is None

**density** : bool, optional

If True, the first element of the return tuple will be the counts normalized to form a probability density, i.e., the area (or integral) under the histogram will sum to 1. This is achieved by dividing the count by the number of observations times the bin width and not dividing by the total number of observations. If *stacked* is also True, the sum of the histograms is normalized to 1.

Default is None for both *normed* and *density*. If either is set, then that value will be used. If neither are set, then the args will be treated as False.

If both *density* and *normed* are set an error is raised.

**weights** : (n, ) array\_like or None, optional

An array of weights, of the same shape as *x*. Each value in *x* only contributes its associated weight towards the bin count (instead of 1). If *normed* or *density* is True, the weights are normalized, so that the integral of the density over the range remains 1.

Default is None

**cumulative** : bool, optional

If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If *normed* or *density* is also True then the histogram is normalized such that the last bin equals 1. If *cumulative* evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if *normed* and/or *density* is also True, then the histogram is normalized such that the first bin equals 1.

Default is False

**bottom** : array\_like, scalar, or None

Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

Default is None

**histtype** : {'bar', 'barstacked', 'step', 'stepfilled'}, optional

The type of histogram to draw.

* '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.

Default is 'bar'

**align** : {'left', 'mid', 'right'}, optional

Controls how the histogram is plotted.

* 'left': bars are centered on the left bin edges.
* 'mid': bars are centered between the bin edges.
* 'right': bars are centered on the right bin edges.

Default is 'mid'

**orientation** : {'horizontal', 'vertical'}, optional

If 'horizontal', [barh](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.barh.html#matplotlib.pyplot.barh) will be used for bar-type histograms and the *bottom* kwarg will be the left edges.

**rwidth** : scalar or None, optional

The relative width of the bars as a fraction of the bin width. If None, automatically compute the width.

Ignored if *histtype* is 'step' or 'stepfilled'.

Default is None

**log** : bool, optional

If True, the histogram axis will be set to a log scale. If *log* is True and *x* is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned.

Default is False

**color** : color or array\_like of colors or None, optional

Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

Default is None

**label** : str or None, optional

String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

default is None

**stacked** : bool, optional

If True, multiple data are stacked on top of each other If False multiple data are arranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'

Default is False

**normed** : bool, optional

Deprecated; use the density keyword argument instead.

**Returns:**

**n** : array or list of arrays

The values of the histogram bins. See *normed* or *density* and *weights* for a description of the possible semantics. If input *x* is an array, then this is an array of length *nbins*. If input is a sequence of arrays [data1, data2,..], then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

**bins** : array

The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

**patches** : list or list of lists

Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

**Example:**

from matplotlib import pyplot as plt

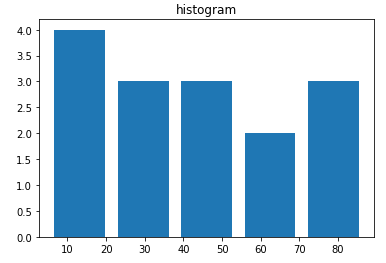
import numpy as np

a = np.array([22,87,5,43,56,73,55,54,11,20,51,5,79,31,27])

plt.hist(a, bins = 5, rwidth=0.8) #

plt.title("histogram")

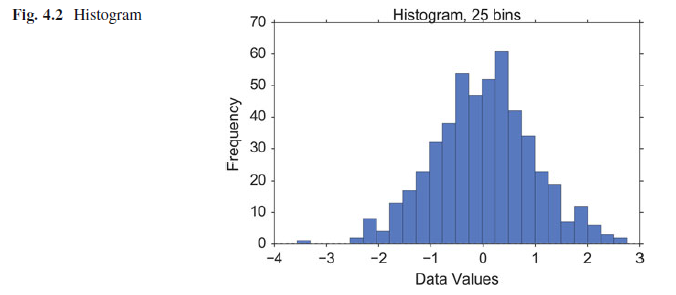
plt.show()



A histogram is a great tool for quickly assessing a [probability distribution](https://en.wikipedia.org/wiki/Probability_distribution) that is intuitively understood by almost any audience. Python offers a handful of different options for building and plotting histograms. Most people know a histogram by its graphical representation, which is similar to a bar graph:

*Histograms* provide a first good overview of the distribution of your data. If you divide by the overall number of data points, you get a *relative frequency histogram*;and if you just connect the top center points of each bin, you obtain a *relativefrequency polygon*.

plt.hist(x, bins=25)



1. **Kernel-Density-Estimation (KDE) Plots**

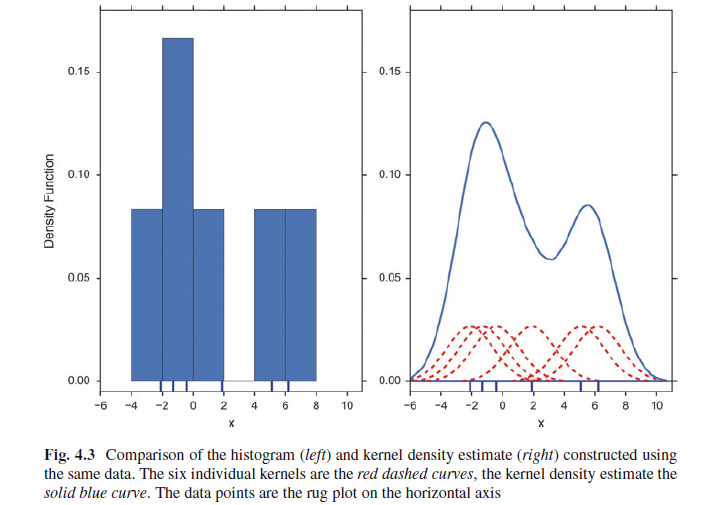
In [statistics](https://en.wikipedia.org/wiki/Statistics), **kernel density estimation** (**KDE**) is a [non-parametric](https://en.wikipedia.org/wiki/Non-parametric_statistics) way to [estimate](https://en.wikipedia.org/wiki/Density_estimation) the [probability density function](https://en.wikipedia.org/wiki/Probability_density_function) of a [random variable](https://en.wikipedia.org/wiki/Random_variable). Kernel density estimation is a fundamental data smoothing problem where inferences about the [population](https://en.wikipedia.org/wiki/Statistical_population) are made, based on a finite data [sample](https://en.wikipedia.org/wiki/Statistical_sample).

Histograms have the disadvantage that they are discontinuous, and that their shapecritically depends on the chosen bin-width. In order to obtain smooth *probabilitydensities*, i.e., curves describing the likelihood of finding an event in any giveninterval, the technique of *Kernel Density Estimation* (KDE) can be used. Therebya normal distribution is typically used for the kernel. The width of this kernelfunction determines the amount of smoothing. To see how this works, we comparethe construction of histogram and kernel density estimators, using the following sixdata points:

x = [-2.1, -1.3, -0.4, 1.9, 5.1, 6.2].

For the histogram, first the horizontal axis is divided into subintervals or binswhich cover the range of the data. In Fig. 4.3, left, we have six bins each of width 2.Whenever a data point falls inside this interval, we place a box of height 1/12. Ifmore than one data point falls inside the same bin, we stack the boxes on top ofeach other.

For the kernel density estimate, we place a normal kernel with variance 2.25(indicated by the red dashed lines in Fig. 4.3, right) on each of the data points *xi*.The kernels are summed to make the kernel density estimate (solid blue curve). Thesmoothness of the kernel density estimate is evident. Compared to the discreteness

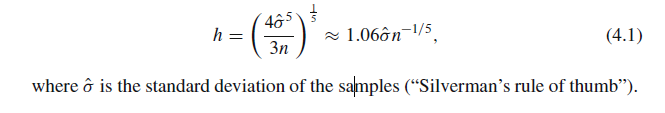


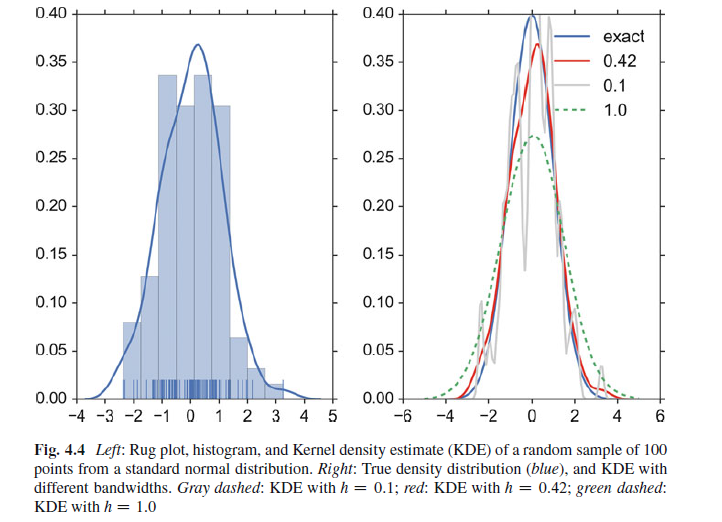
density for continuous random variables.

sns.kdeplot(x)

The bandwidth of the kernel is the parameter which determines how much wesmooth out the contribution from each event. To illustrate its effect, we take asimulated random sample from the standard normal distribution, plotted as theblue spikes in the *rug plot* on the horizontal axis in Fig. 4.4, left. (A *rug plot* isa plot where every data entry is visualized by a small vertical tick.) The right plotshows the true density in blue. (A normal density with mean 0 and variance 1.) Incomparison, the gray curve is undersmoothed since it contains too many spuriousdata artifacts arising from using a bandwidth *h* D 0:1 which is too small. The greendashed curve is oversmoothed since using the bandwidth *h* D 1 obscures much ofthe underlying structure. The red curve with a bandwidth of *h* D 0:42 is consideredto be optimally smoothed since its density estimate is close to the true density.

It can be shown that under certain conditions the optimal choice for *h* is



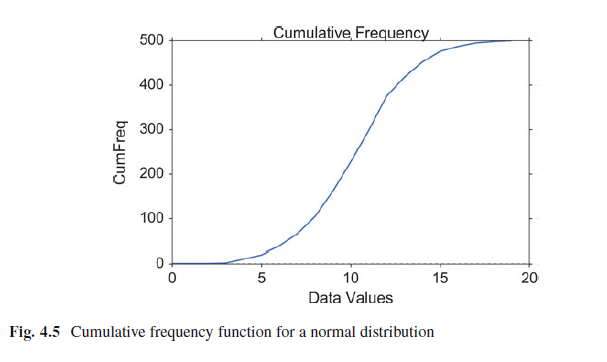


**d) Cumulative Frequencies**

A *cumulative frequency* curve indicates the number (or percent) of data with lessthan a given value. This curve is very useful for statistical analysis, for examplewhen we want to know the data range containing 95% of all the values. Cumulativefrequencies are also useful for comparing the distribution of values in two or moredifferent groups of individuals.

When you use percentage points, the cumulative frequency presentation has theadditional advantage that it is bounded: 

plt.plot(stats.cumfreq(x,numbins)[0])



**e) Error-Bars**

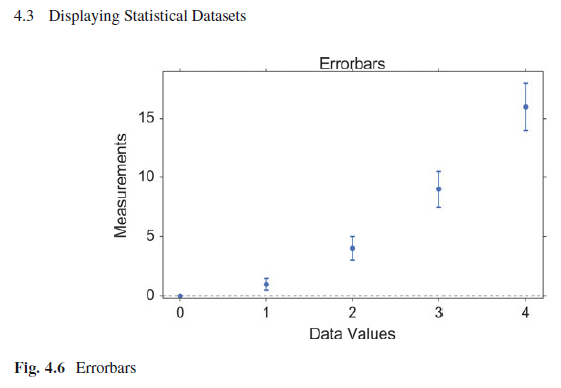
*Error-bars* are a common way to show mean value and variability when comparing measurement values. Note that it always has to be stated explicitly if the error-bars correspond to the *standard deviation* or to the *standard error* of the data. Using *standard errors* has a nice feature: When error bars for the standard errors fortwo groups overlap, one can be sure the difference between the two means is notstatistically significant (*p* > 0:05). However, the opposite is not always true!

index = np.arange(5)

y = index\*\*2

errorBar = index/2 # just for demonstration

plt.errorbar(index,y, yerr=errorBar, fmt='o',capsize=5, capthick=3)



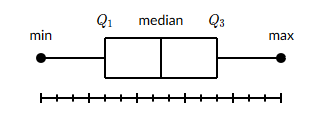
**f) Box Plots**

**Box plots** (also called **box-and-whisker plots** or **box-whisker plots**) give a good graphical image of the concentration of the data. They also show how far the extreme values are from most of the data.

Boxplots are frequently used in scientific publications to indicate values in two or more groups.

A box plot is constructed from five values: the minimum value, the first quartile, the median (second quartile), the third quartile, and the maximum value.

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.



Example:

Consider, again, this dataset.

25, 28, 29 29, 30, 34, 35, 35, 37, 38

**Step 1:** Order the data from smallest to largest.

25, 28, 29 29, 30, 34, 35, 35, 37, 38

**Step 2:** Find the median.

(30+34)/2​=32

The median (Q2) = 32.

**Step 3:**

The first quartile is the median of the data points to the left of the median. Q1​=29

The third quartile is the median of the data points to the right of the median. Q3​=35

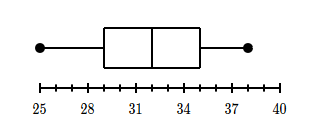
**step 4:**

Complete the five-number summary by finding the min and the max.

The min is the smallest data point, which is 25

The max is the largest data point, which is 38

The five-number summary is 25, 29, 32, 35, 38



Care has to be taken with the whiskers, as different conventions exist for them. The most common form is that the lower whisker indicates the lowest value still within 1.5 \* *interquartile-range* (IQR) of the lower quartile, and the upper whisker the highest value still within 1.5 \* IQR of the upper quartile. Outliers (outside the whiskers) are plotted separately. Another convention is to have the whiskers indicate the full data range.

There are a number of tests to check for outliers. The method suggested by Tukey,for example, is to check for data which lie more than 1.5 \* IQR above or below thefirst/third quartile (see Sect. 6.1.2).

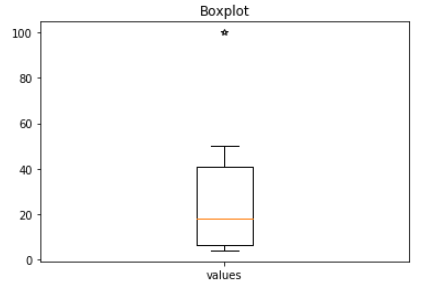
Example:

x = [5,7,16,20,38,4,50,100]

plt.boxplot(x, sym='\*',labels=["values"])

plt.title("Boxplot")

plt.show()



Example-2:

import matplotlib.pyplot as plt

value1 = [82,76,24,40,67,62,75,78,71,32,98,89,78,67,72]

value2 = [62,5,91,25,36,32,96,95,3,90,95,32,27,55,100]

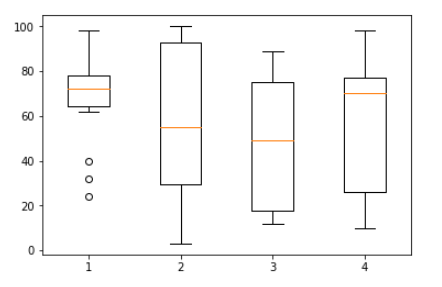
value3 = [23,89,12,78,72,89,25,69,68,86,19,49,15,16,16]

value4 = [59,73,70,16,81,61,88,98,10,87,29,72,16,23,72]

box\_plot\_data=[value1,value2,value3,value4]

plt.boxplot(box\_plot\_data)

plt.show()



Example-3

import matplotlib.pyplot as plt

value1 = [82,76,24,40,67,62,75,78,71,32,98,89,78,67,72]

value2 = [62,5,91,25,36,32,96,95,3,90,95,32,27,55,100]

value3 = [23,89,12,78,72,89,25,69,68,86,19,49,15,16,16]

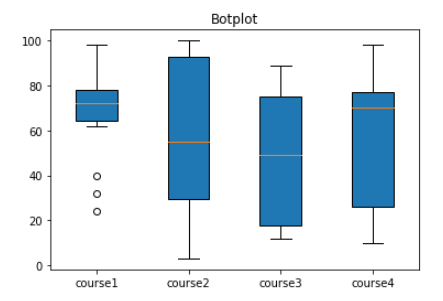
value4 = [59,73,70,16,81,61,88,98,10,87,29,72,16,23,72]

box\_plot\_data=[value1,value2,value3,value4]

plt.boxplot(box\_plot\_data,patch\_artist=True,labels=['course1','course2','course3','course4'])

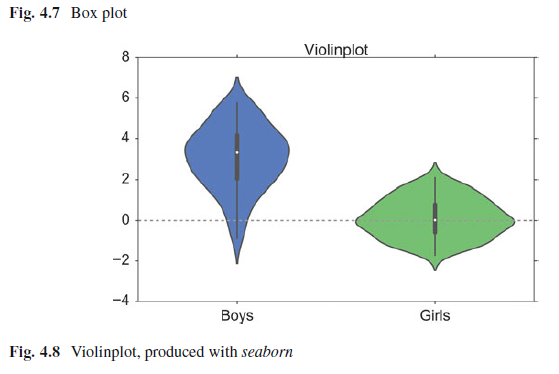
plt.title("Botplot")

plt.show()



boxplot() function takes the data array to be plotted as input in first argument, second argument notch=**‘True’** creates the notch format of the box plot**.** Third argument **patch\_artist=True,** fills the boxplot with color and fourth argument takes the label to be plotted.

Boxplots can be combined with KDE-plots to produce the so-called *violin plots*,where the vertical axis is the same as for the box-plot, but in addition a KDE-plot isshown symmetrically along the horizontal direction (Fig. 4.8).



# Generate the data

nd = stats.norm

data = nd.rvs(size=(100))

nd2 = stats.norm(loc = 3, scale = 1.5)

data2 = nd2.rvs(size=(100))

# Use pandas and the seaborn package

# for the violin plot

df = pd.DataFrame({'Girls':data, 'Boys':data2})

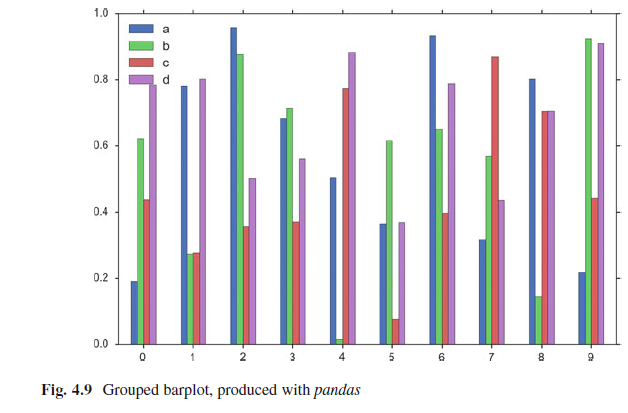
sns.violinplot(df)

**g) Grouped Bar Charts**

For some applications the plotting abilities of *pandas* can facilitate the generation of useful graphs, e.g., for grouped bar plots (Figs. 4.9, 4.10, 4.11, and 4.12):

df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c', 'd'])

df.plot(kind='bar', grid=False)

****

**h) Pie Charts**

A [circle graph/pie chart](https://www150.statcan.gc.ca/n1/edu/power-pouvoir/glossary-glossaire/5214842-eng.htm#piechart) is a way of summarizing a set of [categorical](https://www150.statcan.gc.ca/n1/edu/power-pouvoir/glossary-glossaire/5214842-eng.htm#categorical) data or displaying the different values of a given variable (e.g., percentage distribution). This type of chart is a circle divided into a series of segments(arcs). Each segment represents a particular category. The area of each segment is the same proportion of a circle as the category is of the total data set.

In a pie chart, the arc(segment) length of each slice (and consequently its central angle and area), is proportional to the quantity it represents.

In order to make a pie chart, you must have a list of [**categorical variables**](https://www.statisticshowto.datasciencecentral.com/what-is-a-categorical-variable/) (descriptions of your categories) as well as [**numeric variables**](https://www.statisticshowto.datasciencecentral.com/what-are-quantitative-variables-and-quantitative-data/).

Pie charts are good to show proportional data of different categories and figures are usually in percentages here.

**Example:**

import matplotlib.pyplot as plt

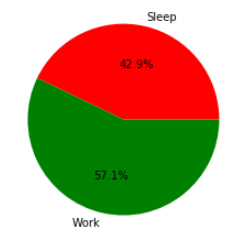
slices\_hours = [6, 8]

activities = ['Sleep', 'Work']

colors = ['r', 'g']

plt.pie(slices\_hours, labels=activities, colors=colors, autopct='%.1f%%')

plt.show()

**

*Example:*

*import matplotlib.pyplot as plt*

*# Data to plot*

*labels = 'Python', 'C++', 'Ruby', 'Java'*

*sizes = [215, 130, 245, 210]*

*colors = ['gold', 'yellowgreen', 'blue', 'orange']*

*explode = (0.1, 0, 0, 0) # explode 1st slice*

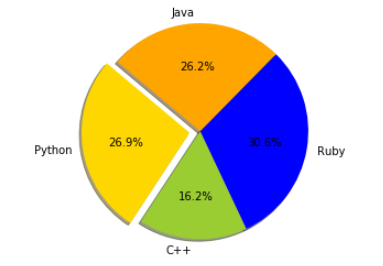
*# Plot*

*plt.pie(sizes, explode=explode, labels=labels, colors=colors,*

*autopct='%1.1f%%', shadow=True, startangle=140)*

*plt.axis('equal')*

*plt.show()*

**

*Pie charts* can be generated with a number of different options, e.g. import seaborn as sns

import matplotlib.pyplot as plt

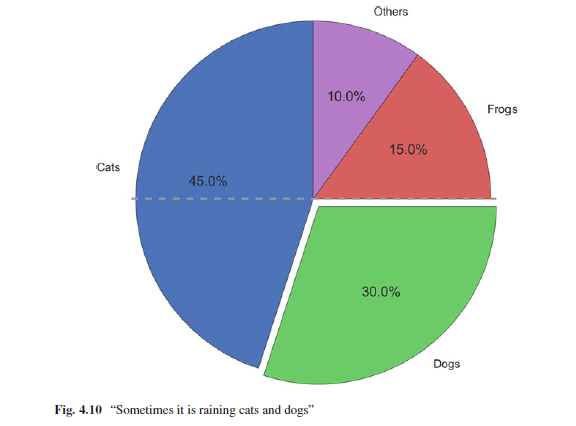
txtLabels = 'Cats', 'Dogs', 'Frogs', 'Others'

fractions = [45, 30, 15, 10]

offsets =(0, 0.05, 0, 0)

plt.pie(fractions, explode=offsets, labels=txtLabels, autopct='%1.1f%%', shadow=True, startangle=90, colors=sns.color\_palette('muted') )

plt.axis('equal')



***4.3.2 Bivariate and Multivariate Plots***

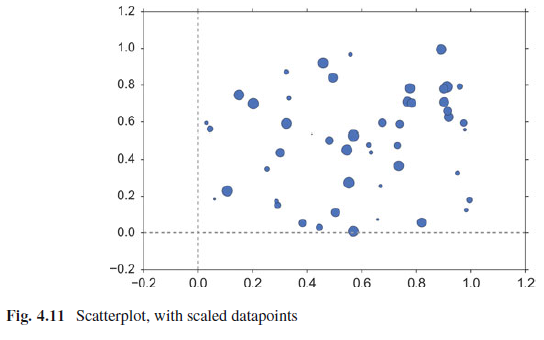
**a) Bivariate Scatter Plots**

Simple scatter plots are trivial. But *pandas* also makes fancy scatter plots easy:

df2 = pd.DataFrame(np.random.rand(50, 4),

columns=['a', 'b', 'c', 'd'])

df2.plot(kind='scatter', x='a', y='b', s=df['c']\*300);



1. **3D Plots**

3D plots in *matplotlib* are a bit awkward, because separate modules have to beimported, and axes for 3D plots have to be explicitly declared. However, once theaxis is correctly defined, the rest is straightforward. Here are two examples:

# imports specific to the plots in this example

import numpy as np

from matplotlib import cm

from mpl\_toolkits.mplot3d.axes3d import get\_test\_data

# Twice as wide as it is tall.

fig = plt.figure(figsize=plt.figaspect(0.5))

#---- First subplot

# Note that the declaration "projection='3d'"

# is required for 3d plots!

ax = fig.add\_subplot(1, 2, 1, projection='3d')

# Generate the grid

X = np.arange(-5, 5, 0.1)

Y = np.arange(-5, 5, 0.1)

X, Y = np.meshgrid(X, Y)

# Generate the surface data

R = np.sqrt(X\*\*2 + Y\*\*2)

Z = np.sin(R)

# Plot the surface

surf = ax.plot\_surface(X, Y, Z, rstride=1, cstride=1,

cmap=cm.GnBu, linewidth=0, antialiased=False)

ax.set\_zlim3d(-1.01, 1.01)

fig.colorbar(surf, shrink=0.5, aspect=10)

#---- Second subplot

ax = fig.add\_subplot(1, 2, 2, projection='3d')

X, Y, Z = get\_test\_data(0.05)

ax.plot\_wireframe(X, Y, Z, rstride=10, cstride=10)

outfile = '3dGraph.png'

plt.savefig(outfile, dpi=200)

print('Image saved to {0}'.format(outfile))

plt.show()

