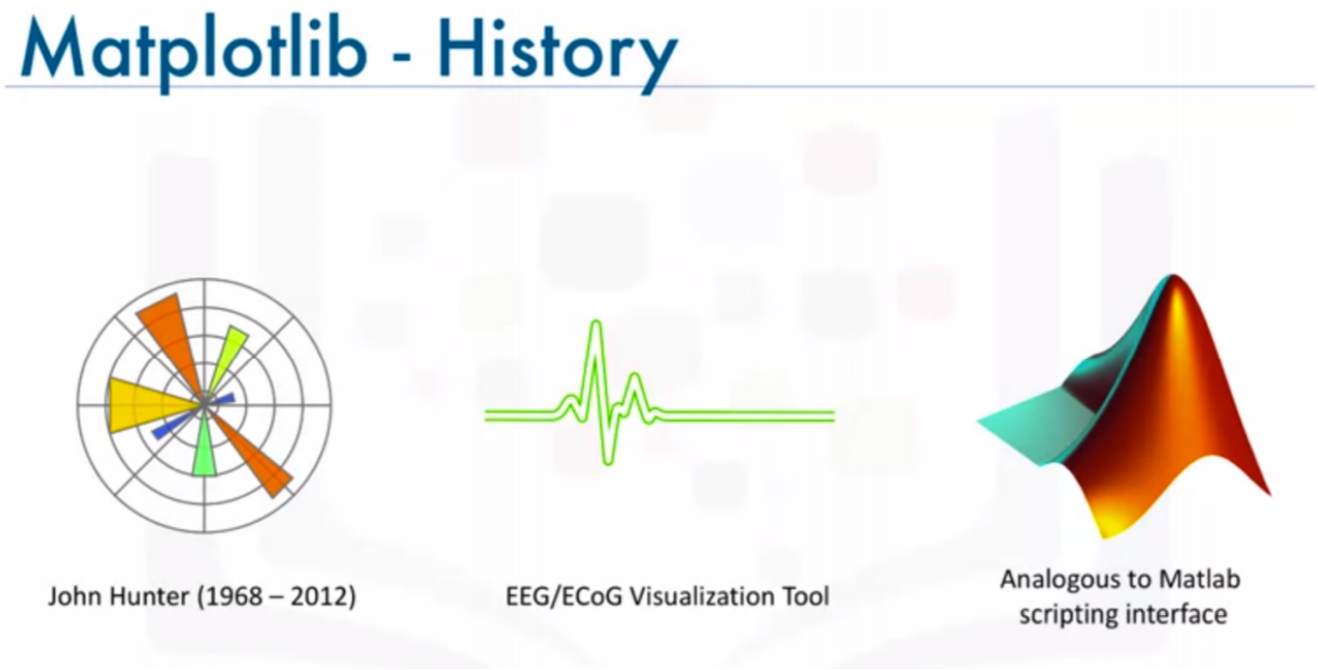
Good Morning Everyone

Matplotlib --> Data Visualization

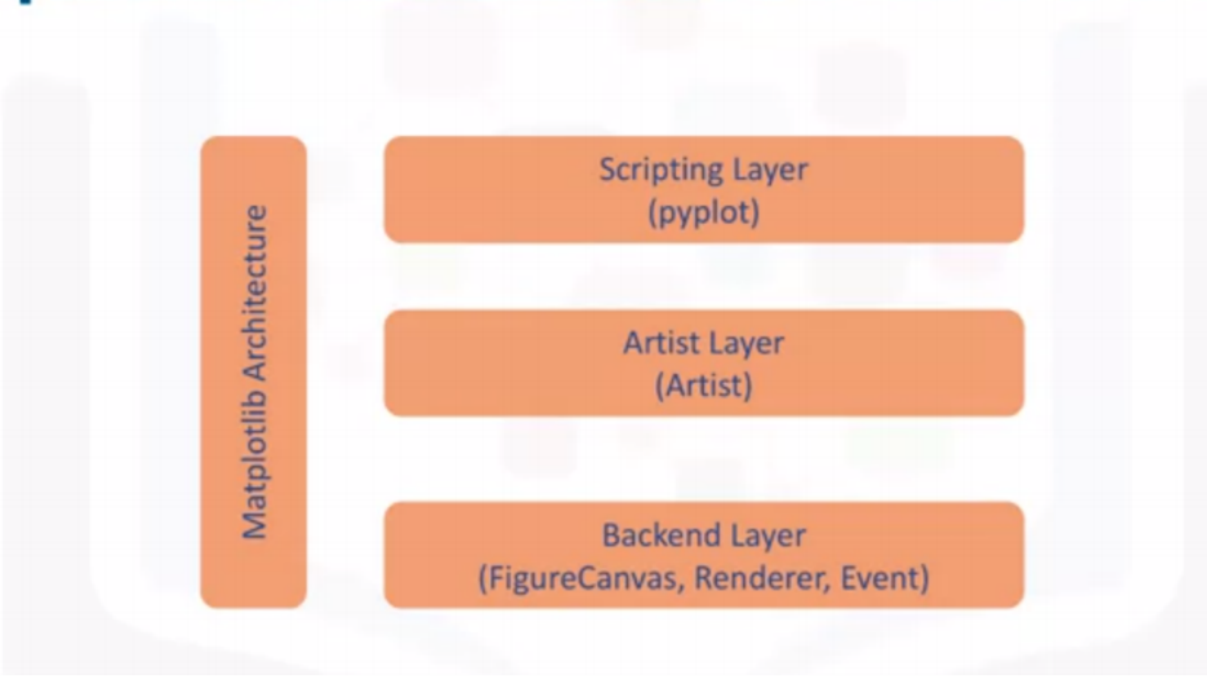
Seaborn

Plotly

Bokeh



Matplotlib Architecture



In [2]:

```
from matplotlib import pyplot as plt
import matplotlib.pyplot as plt
```

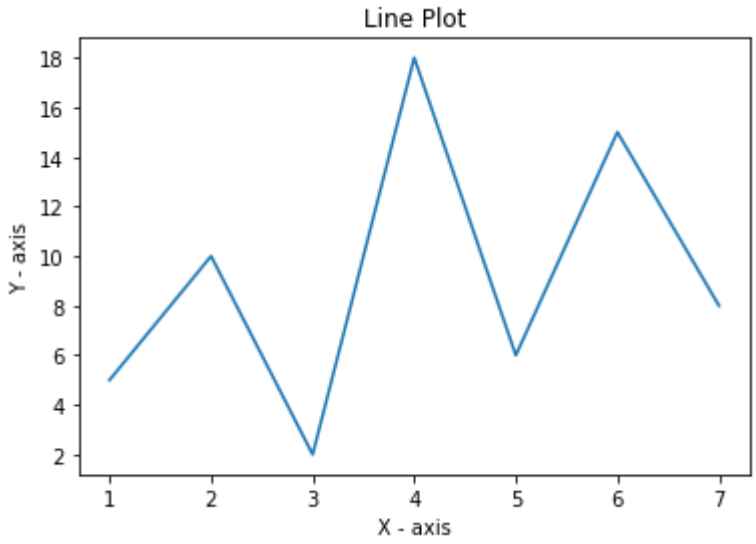
pip install matplotlib

conda install matplotlib

1. Line Plot
2. Scatter Plot
3. Bar Graph
4. Histgram
5. Pie chart
6. Box Plot
7. Stem Plot

In [8]:

```
x = [1, 2, 3, 4, 5, 6, 7]
y = [5, 10, 2, 18, 6, 15, 8]

plt.plot(x, y)
plt.xlabel("X - axis")
plt.ylabel("Y - axis")
plt.title('Line Plot')
plt.show()
```



In [6]:

```
help(plt.plot)
```

```
Help on function plot in module matplotlib.pyplot:

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. If not given, they default to
    ``[0, ..., N-1]``.

    Commonly, these parameters are arrays of length N. However,
    scalars are supported as well (equivalent to an array with
    constant value).

    The parameters can also be 2-dimensional. Then, the columns
    represent separate data sets.

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.

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: bool
      clip_box: `.Bbox`
      clip_on: bool
      clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
      color: 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: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-pos
t'}
      figure: `.Figure`
      fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
      gid: str
      in_layout: bool
      label: object
      linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
      linewidth: float
      marker: unknown
      markeredgecolor: color
      markeredgewidth: float
      markerfacecolor: color
      markerfacecoloralt: color
      markersize: float
      markevery: unknown
      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 = '[color][marker][line]'

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.

Colors

The following color abbreviations are supported:

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

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'``).

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
        'ro'   # red circles
        'g-'   # green solid line
        '--'   # dashed line with default color
        'k^:' # black triangle_up markers connected by a dotted line

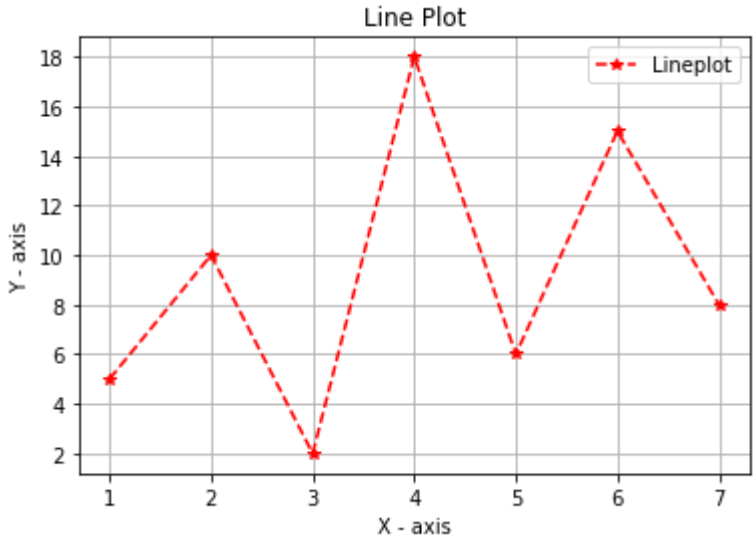
    .. note::
        In addition to the above described arguments, this function can take
a
        data keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

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

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

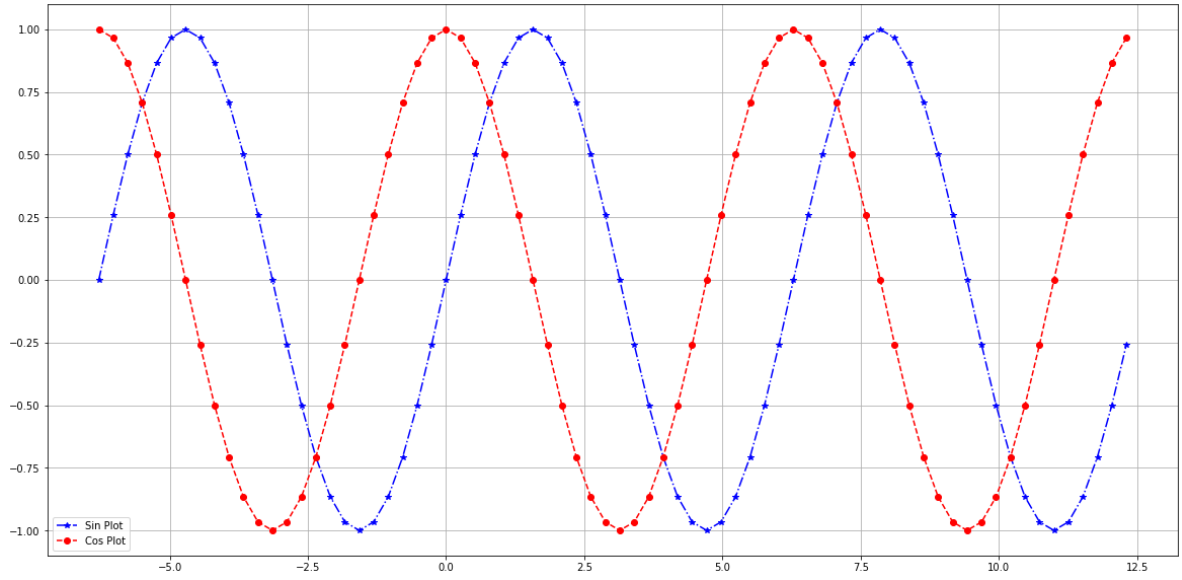
In [17]:

```
x = [1, 2, 3, 4, 5, 6, 7]
y = [5, 10, 2, 18, 6, 15, 8]

plt.plot(x, y, linestyle = '--', linewidth = 1.5, color = 'r', marker = '*', label = 'Linep
plt.xlabel("X - axis")
plt.ylabel("Y - axis")
plt.title('Line Plot')
plt.legend()
plt.grid()
plt.show()
```



In [27]:

```
import numpy as np

x = np.radians(np.arange(-360, 720, 15))
siny = np.sin(x)
cosy = np.cos(x)
plt.figure(figsize=(20, 10)) # w, h
plt.plot(x, siny, linestyle = '-.', linewidth = 1.5, color = 'b', marker = '*', label = 'Si
plt.plot(x, cosy, linestyle = '--', linewidth = 1.5, color = 'r', marker = 'o', label = 'Co
plt.legend()
plt.grid()
plt.show()
```



In [25]:

```
help(plt.legend)
```

```
Help on function legend in module matplotlib.pyplot:

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.set_label('Label via method')
        line, = ax.plot([1, 2, 3])
        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 : int or string or pair of floats, default: :rc:`legend.loc` ('best'
for axes, 'upper right' for figures)
        The location of the legend. Possible codes are:

            ===============   =============
            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
            ===============   =============


        Alternatively can be a 2-tuple giving ``x, y`` of the lower-left
        corner of the legend in axes coordinates (in which case
        ``bbox_to_anchor`` will be ignored).

        The 'best' option can be quite slow for plots with large amounts
        of data. Your plotting speed may benefit from providing a specific
        location.

    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.
```

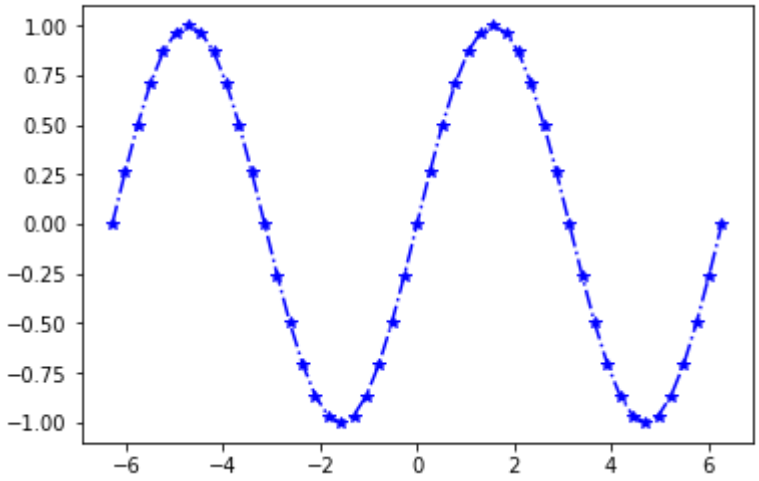
```
        Bbox coordinates are interpreted in the coordinate system given by
        `bbox_transform`, with the default transform
        Axes or Figure coordinates, depending on which ``legend`` is called.

        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 *l
oc* at
        x, y.  For example, to put the legend's upper right-hand corner in t
he
        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', 'l
arge', '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
```

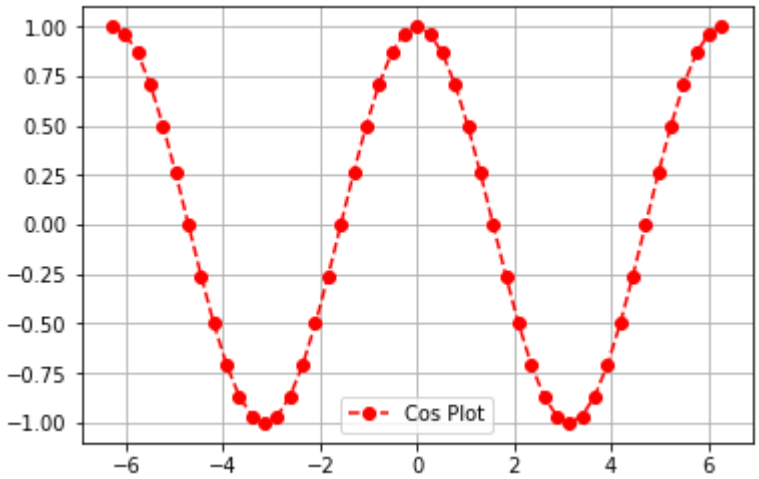
```
        label.
        Default is *True*.

    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 fontsiz
e.

    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
```

In [23]:

```
import numpy as np

x = np.radians(np.arange(-360, 361, 15))
siny = np.sin(x)
cosy = np.cos(x)

plt.plot(x, siny, linestyle = '-.', linewidth = 1.5, color = 'b', marker = '*', label = 'Si
plt.show()
plt.plot(x, cosy, linestyle = '--', linewidth = 1.5, color = 'r', marker = 'o', label = 'Co
plt.legend()
plt.grid()
plt.show()
```





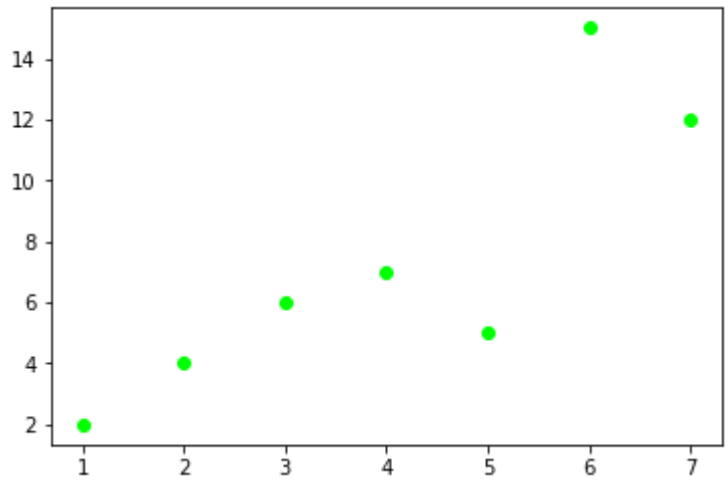
Scatter Plot

In [37]:

```
x = [1, 2, 3, 4, 5, 6, 7]
y = [2, 4, 6, 7, 5, 15, 12]


plt.scatter(x, y, color = '#00ff00') # rrggbb
#plt.plot(x, y)
```

Out[37]:

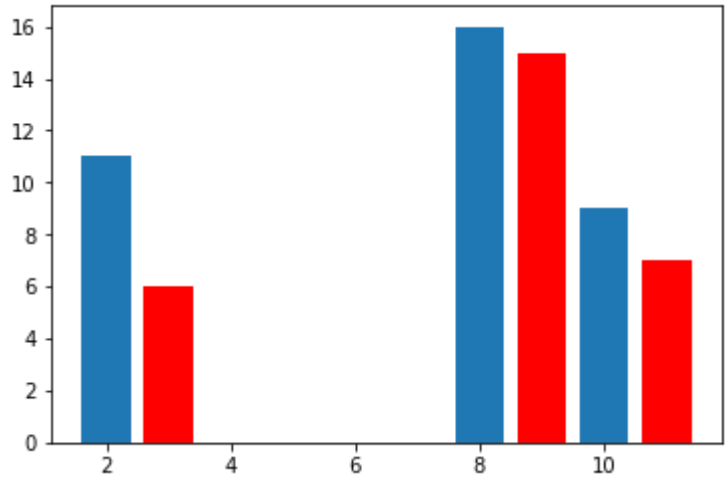
```
<matplotlib.collections.PathCollection at 0x1a8e42db048>
```



Bar Graph

In [39]:

```
x = [2,8,10]
y = [11,16,9]

x2 = [3,9,11]
y2 = [6,15,7]


plt.bar(x, y)
plt.bar(x2, y2, color = "r")

plt.show()
```



Histogarm

In [42]:

```
np.random.randint(1, 100)
```

Out[42]:

```
74
```

In [43]:

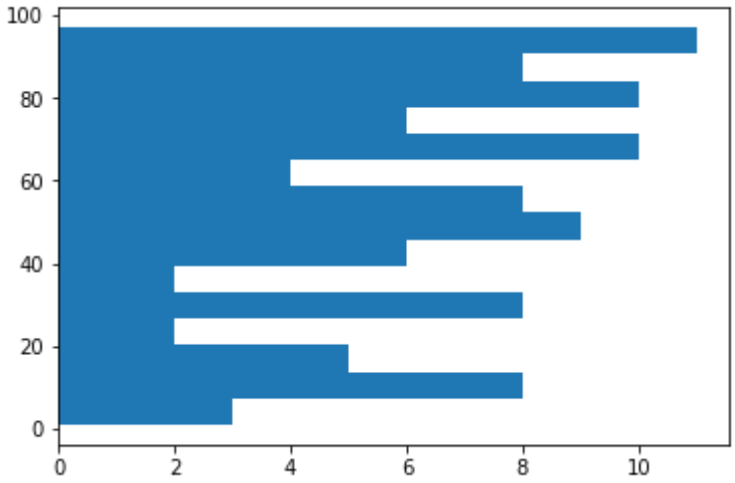
```
marks = [np.random.randint(1, 100) for i in range(100)]
print(marks)
```

```
[90, 57, 95, 67, 42, 96, 31, 48, 68, 65, 16, 83, 83, 89, 67, 97, 62, 77, 9,
44, 63, 29, 42, 12, 31, 15, 30, 20, 10, 53, 65, 46, 44, 88, 1, 12, 47, 15, 4
6, 21, 95, 57, 69, 87, 72, 74, 8, 93, 78, 53, 12, 96, 62, 16, 82, 57, 82, 9
2, 30, 51, 79, 32, 74, 89, 82, 94, 66, 47, 90, 3, 61, 38, 85, 66, 43, 56, 7
7, 94, 73, 49, 56, 11, 6, 79, 58, 8, 95, 28, 33, 31, 47, 97, 47, 81, 68, 66,
23, 86, 78, 42]
```

In [53]:

```
calc = plt.hist(marks, bins = 15, orientation='horizontal')
print(calc)
plt.show()
```

```
(array([ 3.,  8.,  5.,  2.,  8.,  2.,  6.,  9.,  8.,  4., 10.,  6., 10.,
        8., 11.]), array([ 1. ,  7.4, 13.8, 20.2, 26.6, 33. , 39.4, 45.8, 5
2.2, 58.6, 65. ,
       71.4, 77.8, 84.2, 90.6, 97. ]), <a list of 15 Patch objects>)
```



In [3]:

```
help(plt.hist)
```

```
Help on function hist in module matplotlib.pyplot:

hist(x, bins=None, range=None, density=None, weights=None, cumulative=False,
bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=Non
e, log=False, color=None, label=None, stacked=False, normed=None, *, data=No
ne, **kwargs)
    Plot a histogram.

    Compute and draw the histogram of *x*. The return value is a
    tuple (*n*, *bins*, *patches*) or ([*n0*, *n1*, ...], *bins*,
    [*patches0*, *patches1*,...]) if the input contains multiple
    data.

    Multiple data can be provided via *x* as a list of datasets
    of potentially different length ([*x0*, *x1*, ...]), or as
    a 2-D ndarray in which each column is a dataset.  Note that
    the ndarray form is transposed relative to the list form.

    Masked arrays are not supported at present.

    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`.

        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`.

        The default is taken from :rc:`hist.bins`.

    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
```

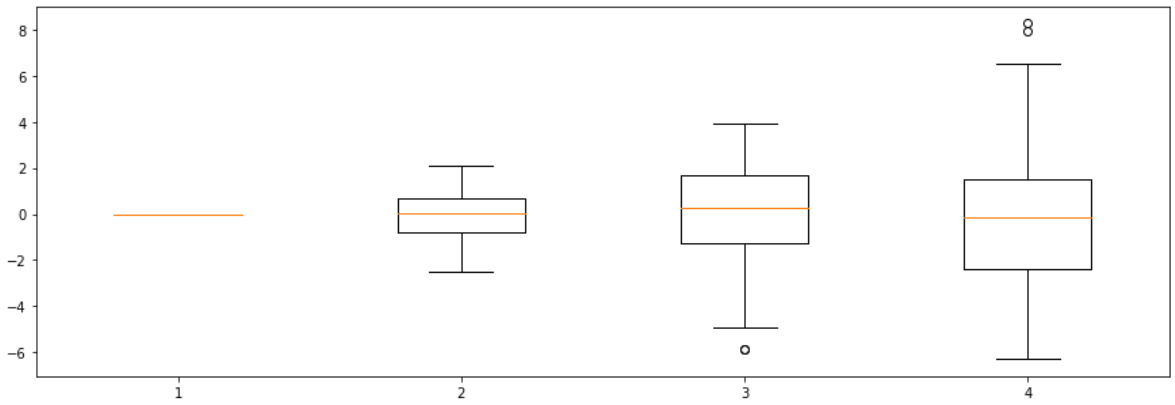
```
      unfilled.

    - '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', `~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.

    Other Parameters
    ----------------
    **kwargs : `~matplotlib.patches.Patch` properties

    See also
    --------
    hist2d : 2D histograms

    Notes
    -----
    .. [Notes section required for data comment. See #10189.]

    .. 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'.

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

Boxplot

In [65]:

```
plt.figure(figsize=(15, 5))
box = [np.random.normal(0, i, 100) for i in range(0, 4)]
#print(box)
plt.boxplot(box)

plt.show()
```



In [56]:

```
box.mean()
```

Out[56]:

```
-0.09931151218690468
```

In [58]:

```
box.max(), box.min()
```

Out[58]:

```
(2.5482484666449503, -2.511322731977096)
```

In [59]:

```
np.percentile(box, 75), np.percentile(box, 25)
```

Out[59]:

```
(0.5300684508500852, -0.8516094079892039)
```

Pie Chart

In [67]:

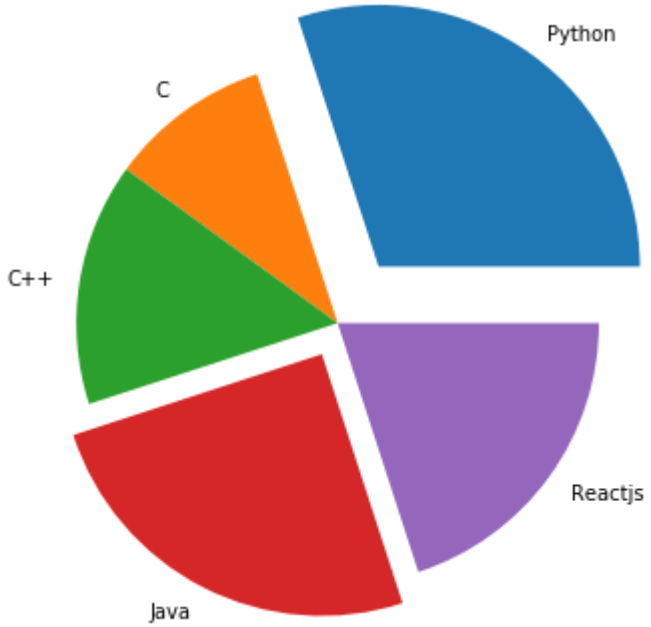
```
programming = ['Python', 'C', 'C++', 'Java', 'Reactjs']
pop = [30, 10, 15, 25, 20]
sum(pop)
```

Out[67]:

```
100
```

In [76]:

```
share = [0.4, 0.0, 0.0, 0.2, 0.0]
plt.pie(pop, labels = programming, explode=share, radius=1.5)
plt.show()
```



In [70]:

```
help(plt.pie)
```

```
Help on function pie in module matplotlib.pyplot:

pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.
6, shadow=False, labeldistance=1.1, startangle=None, radius=None, counterclo
ck=True, wedgeprops=None, textprops=None, center=(0, 0), frame=False, rotate
labels=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, optional, default: 1.1
        The radial distance at which the pie labels are drawn

    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', 'labe
ls', 'x'.

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