Python-Matplotlib tutorial

matplotlib is an excellent 2D (and 3D) python library that can be used to produce publication quality output from your data. The website https://matplotlib.org/ (https://matplotlib.org/) provides a complete resource for how to use matplotlib for your work. In particular if you click on an example plot in the gallery, https://matplotlib.org/gallery/index.html (https://matplotlib.org/gallery/index.html), the browser will display the code required to produce the plot. It is quite difficult to ask google "I would like my plot to look like this, and have these features, how do I do it?", however it is easy to browse through the gallery until you see the feature that you are interested in.

Unlike software like Excel in matplotlib you write code to determine the appearance of all aspects of your graph, you can recycle this code to easily create reproducable, consistent publication quality representations of your scientific data

Preparing the notebook for using matplotlib and numpy.

In [14]:

```
%matplotlib notebook
# this line is required for the plots to appear in the Jupyter cells, rather than launc
hing the matplotlib GUI

import matplotlib

import numpy as np

import matplotlib.pyplot as plt

# Let printing work the same in Python 2 and 3
from __future__ import print_function

# notice two underscores _ either side of future
```

Create some data for plotting examples.

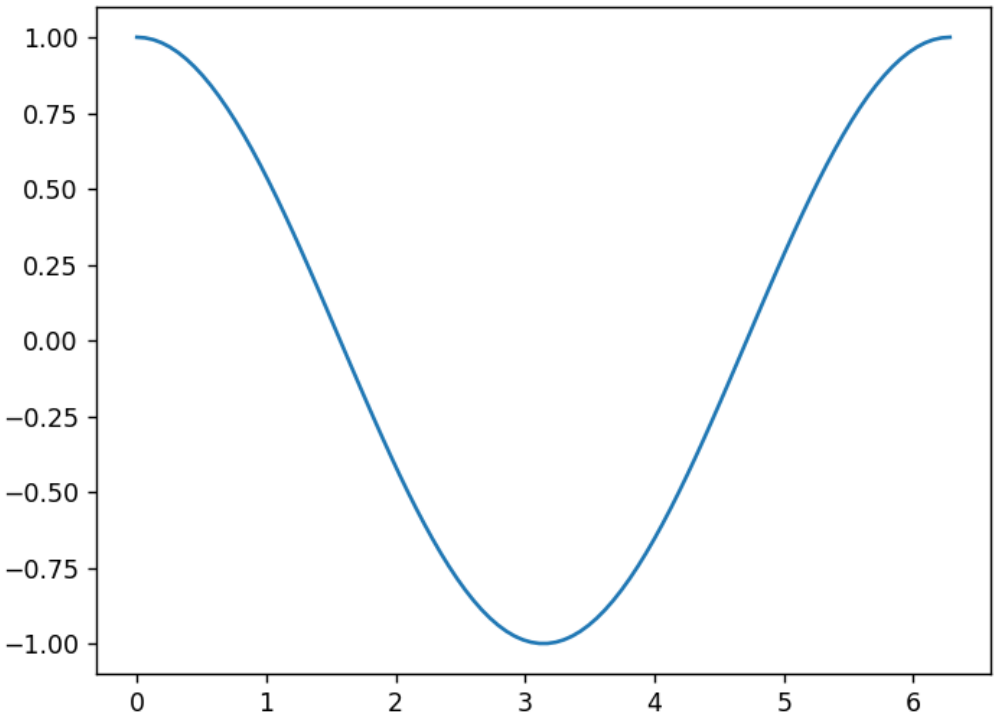
In [15]:

```
x=np.linspace(0,2*np.pi, 100)
y=np.cos(x)
```

Generate the basic matplotlib 2D plot, figure() creates the space into which the plot will be added.

In [16]:

```
plt.figure()
plt.plot(x,y)
```

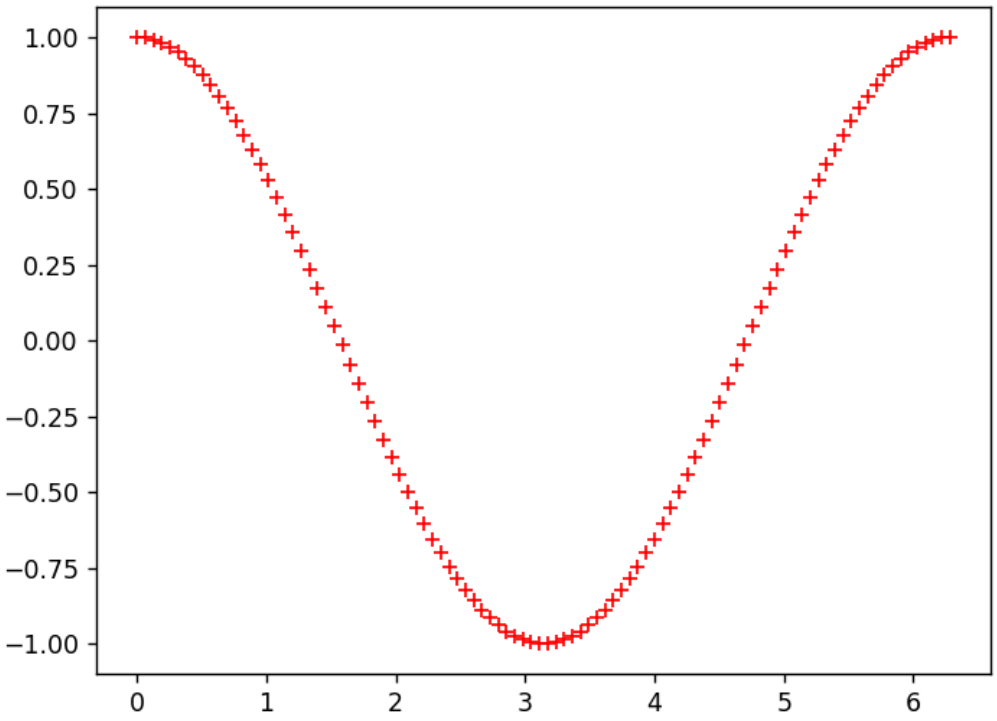


Out[16]:

```
[<matplotlib.lines.Line2D at 0x1f3db51c8d0>]
```

In [17]:

```
plt.figure()
plt.plot(x,y,'r+') # change the line style to red plusses highlights that we are dealin
g with a discrete set of points
```



Out[17]:

```
[<matplotlib.lines.Line2D at 0x1f3db94e5c0>]
```

Within matplotlib.pyplot there are too many functions to describe here:

In [18]:

```
print(dir(plt)) # matplotlib.pyplot is an extensive package
```

```
['Annotation', 'Arrow', 'Artist', 'AutoLocator', 'Axes', 'Button', 'Circl
e', 'Figure', 'FigureCanvasBase', 'FixedFormatter', 'FixedLocator', 'Forma
tStrFormatter', 'Formatter', 'FuncFormatter', 'GridSpec', 'IndexLocator',
'Line2D', 'LinearLocator', 'Locator', 'LogFormatter', 'LogFormatterExponen
t', 'LogFormatterMathtext', 'LogLocator', 'MaxNLocator', 'MultipleLocato
r', 'Normalize', 'NullFormatter', 'NullLocator', 'PolarAxes', 'Polygon',
'Rectangle', 'ScalarFormatter', 'Slider', 'Subplot', 'SubplotTool', 'Tex
t', 'TickHelper', 'Widget', '_INSTALL_FIG_OBSERVER', '_IP_REGISTERED', '__
builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name_
_', '__package__', '__spec__', '_auto_draw_if_interactive', '_autogen_docs
tring', '_backend_mod', '_backend_selection', '_hold_msg', '_imread', '_im
save', '_interactive_bk', '_pylab_helpers', '_setp', '_setup_pyplot_info_d
ocstrings', '_show', '_string_to_bool', 'absolute_import', 'acorr', 'angle
_spectrum', 'annotate', 'arrow', 'autoscale', 'autumn', 'axes', 'axhline',
'axhspan', 'axis', 'axvline', 'axvspan', 'bar', 'barbs', 'barh', 'bone',
'box', 'boxplot', 'broken_barh', 'cla', 'clabel', 'clf', 'clim', 'close',
'cm', 'cohere', 'colorbar', 'colormaps', 'colors', 'connect', 'contour',
'contourf', 'cool', 'copper', 'csd', 'cycler', 'dedent', 'delaxes', 'depre
cated', 'disconnect', 'division', 'docstring', 'draw', 'draw_all', 'draw_i
f_interactive', 'errorbar', 'eventplot', 'figaspect', 'figimage', 'figlege
nd', 'fignum_exists', 'figtext', 'figure', 'fill', 'fill_between', 'fill_b
etweenx', 'findobj', 'flag', 'gca', 'gcf', 'gci', 'get', 'get_backend', 'g
et_cmap', 'get_current_fig_manager', 'get_figlabels', 'get_fignums', 'get_
plot_commands', 'get_scale_docs', 'get_scale_names', 'getp', 'ginput', 'gr
ay', 'grid', 'hexbin', 'hist', 'hist2d', 'hlines', 'hold', 'hot', 'hsv',
'imread', 'imsave', 'imshow', 'inferno', 'install_repl_displayhook', 'inte
ractive', 'ioff', 'ion', 'is_numlike', 'ishold', 'isinteractive', 'jet',
'legend', 'locator_params', 'loglog', 'magma', 'magnitude_spectrum', 'marg
ins', 'matplotlib', 'matshow', 'minorticks_off', 'minorticks_on', 'mlab',
'new_figure_manager', 'nipy_spectral', 'np', 'over', 'pause', 'pcolor', 'p
colormesh', 'phase_spectrum', 'pie', 'pink', 'plasma', 'plot', 'plot_dat
e', 'plotfile', 'plotting', 'polar', 'print_function', 'prism', 'psd', 'py
lab_setup', 'quiver', 'quiverkey', 'rc', 'rcParams', 'rcParamsDefault', 'r
c_context', 'rcdefaults', 'register_cmap', 'rgrids', 'savefig', 'sca', 'sc
atter', 'sci', 'semilogx', 'semilogy', 'set_cmap', 'setp', 'show', 'silent
_list', 'six', 'specgram', 'spectral', 'spring', 'spy', 'stackplot', 'ste
m', 'step', 'streamplot', 'style', 'subplot', 'subplot2grid', 'subplot_too
l', 'subplots', 'subplots_adjust', 'summer', 'suptitle', 'switch_backend',
'sys', 'table', 'text', 'thetagrids', 'tick_params', 'ticklabel_format',
'tight_layout', 'time', 'title', 'tricontour', 'tricontourf', 'tripcolor',
'triplot', 'twinx', 'twiny', 'unicode_literals', 'uninstall_repl_displayho
ok', 'violinplot', 'viridis', 'vlines', 'waitforbuttonpress', 'warn_deprec
ated', 'warnings', 'winter', 'xcorr', 'xkcd', 'xlabel', 'xlim', 'xscale',
'xticks', 'ylabel', 'ylim', 'yscale', 'yticks']
```

In [19]:

```
help(plt.plot) # the plot docstring gives a detailed set of instructions on the usasge
```

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

plot(*args, **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 colo
r
    >>> 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 arr
ay and a dpi value, and returns a (m, n, 3) array
      alpha: float (0.0 transparent through 1.0 opaque)
      animated: bool
      antialiased or aa: bool
      clip_box: a `.Bbox` instance
      clip_on: bool
      clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
      color or c: any matplotlib color
      contains: a callable function
      dash_capstyle: ['butt' | 'round' | 'projecting']
      dash_joinstyle: ['miter' | 'round' | 'bevel']
      dashes: sequence of on/off ink in points
      drawstyle: ['default' | 'steps' | 'steps-pre' | 'steps-mid' | 'steps
-post']
      figure: a `.Figure` instance
      fillstyle: ['full' | 'left' | 'right' | 'bottom' | 'top' | 'none']
      gid: an id string
      label: object
      linestyle or ls: ['solid' | 'dashed', 'dashdot', 'dotted' | (offset,
on-off-dash-seq) | ``'-'`` | ``'--'`` | ``'-.'`` | ``':'`` | ``'None'`` |
``' '`` | ``''``]
      linewidth or lw: float value in points
      marker: :mod:`A valid marker style <matplotlib.markers>`
      markeredgecolor or mec: any matplotlib color
      markeredgewidth or mew: float value in points
      markerfacecolor or mfc: any matplotlib color
      markerfacecoloralt or mfcalt: any matplotlib color
      markersize or ms: float
      markevery: [None | int | length-2 tuple of int | slice | list/array
of int | float | length-2 tuple of float]
      path_effects: `.AbstractPathEffect`
      picker: float distance in points or callable pick function ``fn(arti
st, event)``
      pickradius: float distance in points
      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: a :class:`matplotlib.transforms.Transform` instance
      url: a url string
      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 variing 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 ta
ke a
        **data** keyword argument. If such a **data** argument is given, t
he
        following arguments are replaced by **data[<arg>]**:

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

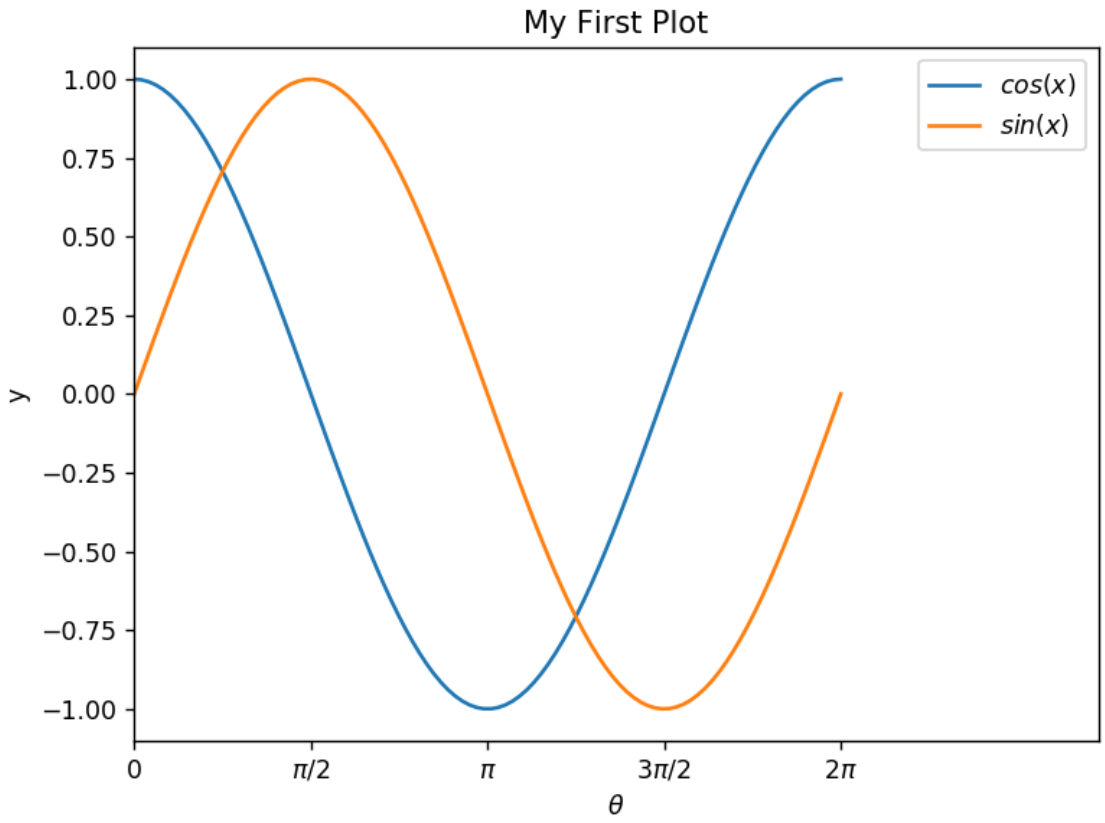
plot(*args, **kwargs) refers to the functions arguments and keyword arguments. The order of the arguments in a python function determines how the argument is passed into the function i.e plot(x,y) will have x as the x-axis, plot(y,x) will have y as the x-axis. The kwargs can come in any order as they are recognised by the keyword i.e. label='my experimental data'.

Returning to our plot:

The following code begins to show how much control you can have over the appearance of the plot, in particular note that LaTex math symbols have been used to label the xticks, and the ticks have been moved to user defined positions.

In [20]:

```
z=np.sin(x)
plt.figure()
plt.plot(x,y, label=r'$cos(x)$')
plt.plot(x,z, label=r'$sin(x)$')# I have not specified the colour, but matplotlib will
 increment
#through a range as new plots are addded.
plt.legend(loc=1) # places the legend (created from the plot labels) in the upper-right
plt.title('My First Plot')
plt.xlabel(r'$\theta$') # the r tells python to read all characters, otherwise it would
not read the \
plt.ylabel('y')
xmin,xmax=plt.xlim() # returns the current limits
plt.xlim(0,xmax*1.3) # sets new limits, makes some space on the right for the legend
plt.xticks((0,np.pi/2,np.pi,3*np.pi/2,2*np.pi),('0','$\pi/2$','$\pi$','$3\pi/2$','$2\pi
$')) # Move the tick labels and use
#Latex commands for the labels
plt.tight_layout() #Ensures nothing overlaps
plt.show() # this is not needed in the notebook but is required from your code.
```



Loading data from a file to an array, np.genfromtxt('fname',...)

Data downloaded from
https://climate.nasa.gov/system/internal_resources/details/original/647_Global_Temperature_Data_File.txt
(https://climate.nasa.gov/system/internal_resources/details/original/647_Global_Temperature_Data_File.txt)

In [21]:

```
climate=np.genfromtxt("https://climate.nasa.gov/system/internal_resources/details/original/647_Global_Temperature_Data_File.txt", skip_header=5, usecols = (0,1)) # data downloaded from the NASA climate change website.
```

In [22]:

```
np.shape(climate) # I want the first two columns in this array
#print(climate)
```

Out[22]:

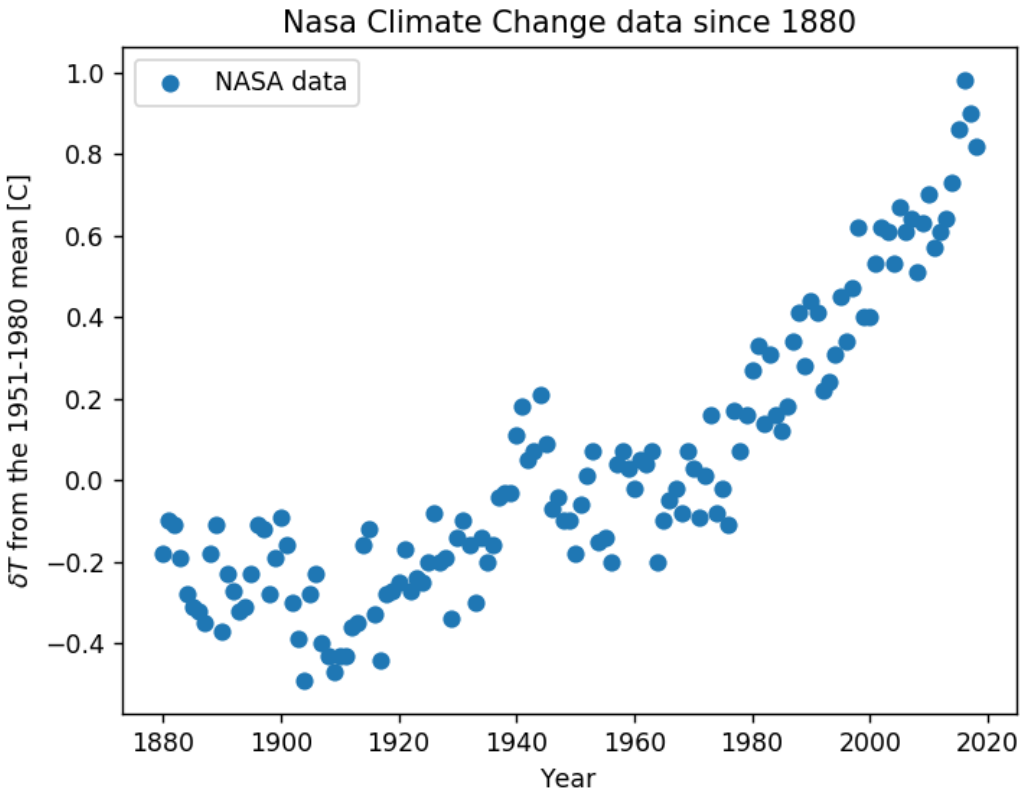
```
(139, 2)
```

In [23]:

```
year,tempchange=climate.transpose()[0],climate.transpose()[1]
# by sepearting the variables with a comma we can assign both in a single line
```

In [24]:

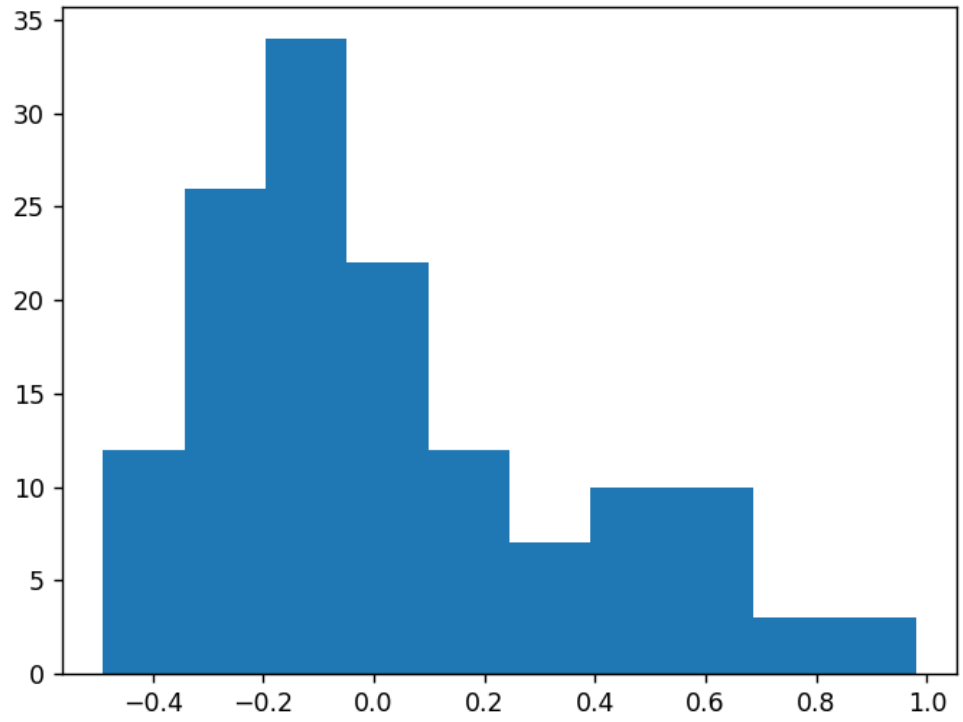
```
plt.figure()
plt.scatter(year,tempchange, label='NASA data')
plt.title('Nasa Climate Change data since 1880')
plt.xlabel('Year')
plt.ylabel('$\delta T$ from the 1951-1980 mean [C]')
plt.legend()
plt.show()
```



As a quick look at how simple it can be to analyse your data with python the following histogram can be generated with a single additional line of code.

In [25]:

```
plt.figure()
plt.hist(tempchange)
```



Out[25]:

```
(array([12., 26., 34., 22., 12.,  7., 10., 10.,  3.,  3.]),
 array([-0.49 , -0.343, -0.196, -0.049,  0.098,  0.245,  0.392,  0.539,
         0.686,  0.833,  0.98 ]),
 <a list of 10 Patch objects>)
```

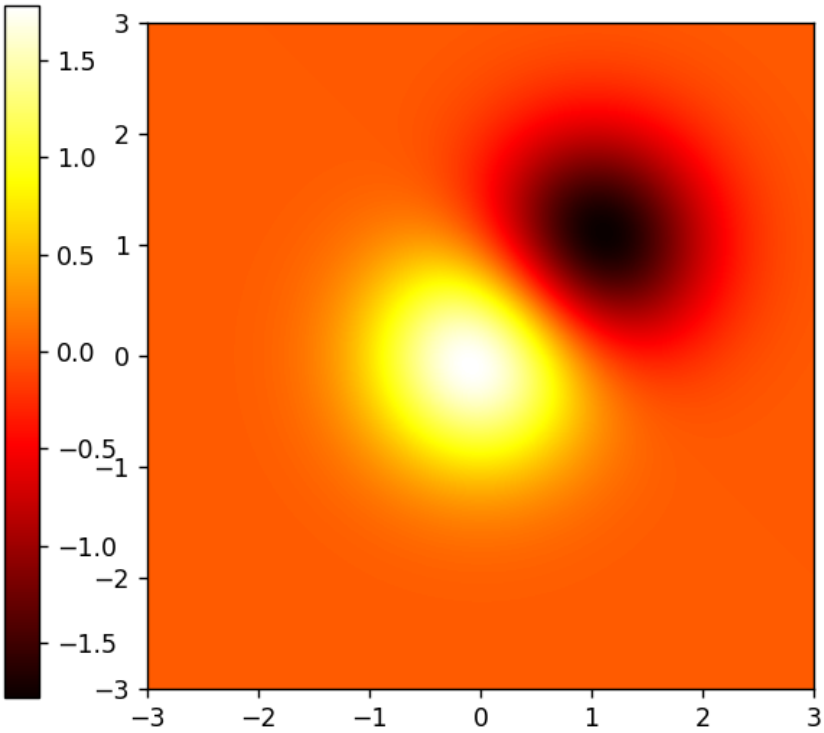
This notebook forms the basic introduction to plotting in python with matplotlib, next term we will expand on this with further topics:

- Adding multiple plots to a figure (subplots)
- Exploring different types of graph
  - Imshow, Axes3D, plotting histograms
- Animate plots

In [28]:

```
import numpy as np
import matplotlib.cm as cm # import the colour map
import matplotlib.pyplot as plt
#Then generate some data to plot
delta = 0.025
x = y = np.arange(-3.0, 3.0, delta)
X, Y = np.meshgrid(x, y) # mesh grid generates a 2D grid of points that links the x and
y data
Z1 = np.exp(-X**2 - Y**2)
Z2 = np.exp(-(X - 1)**2 - (Y - 1)**2)
Z = (Z1 - Z2) * 2
# Now to make the plot
fig, ax = plt.subplots() # the , allows these two commands to go on the same line, foll
owed by the imshow() command
im = ax.imshow(Z, interpolation='bilinear', cmap=cm.hot,origin='lower', extent=[-3, 3,
-3, 3],vmax=abs(Z).max(), vmin=-abs(Z).max())
cbaxes = fig.add_axes([0.1, 0.1, 0.03, 0.8])  # This is the position for the colorbar
cb = plt.colorbar(im, cax = cbaxes)

plt.show()
```



In [ ]:

In [ ]: