intro_Matplotlib

January 10, 2021

[2]:
```
# The following is to know when this notebook has been run and with which␣
↪python version.
import time, sys
print(time.ctime())
print(sys.version.split('|')[0])
```

```
Mon Oct 19 20:15:00 2020
3.7.6 (default, Jan  8 2020, 13:42:34)
[Clang 4.0.1 (tags/RELEASE_401/final)]
```

This is part of the Python lecture given by Christophe Morisset at IA-UNAM. More informations at: http://python-astro.blogspot.mx/

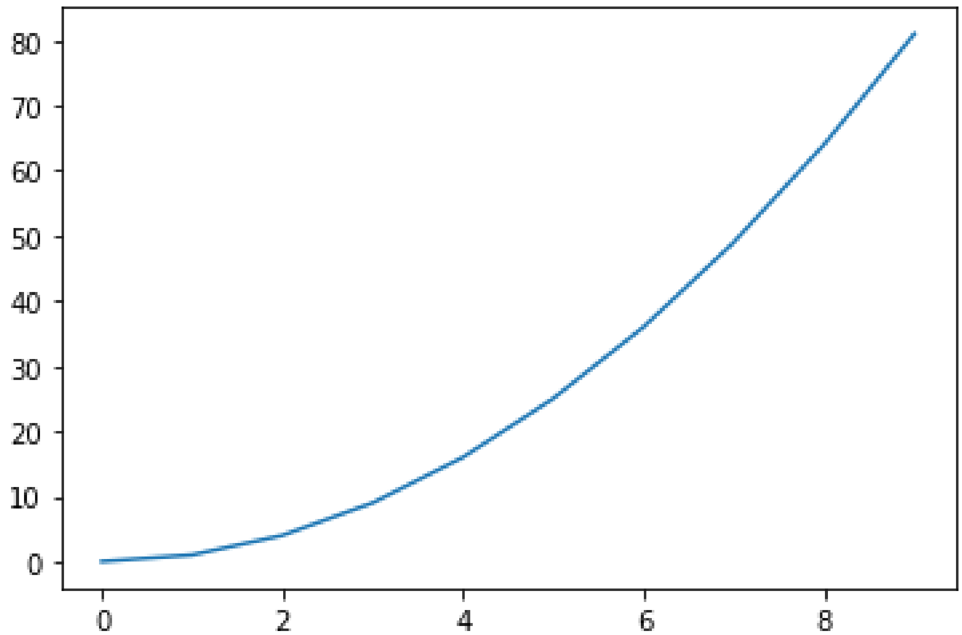
1 D: How to make plots, images, 3D, etc, using Matplotlib

[3]:
```
# this allows the plots to appear in the Notebook webpage:
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt # this is the plotting library
```

Very well done tutorials on the mail Matplotlib web page: http://matplotlib.org/

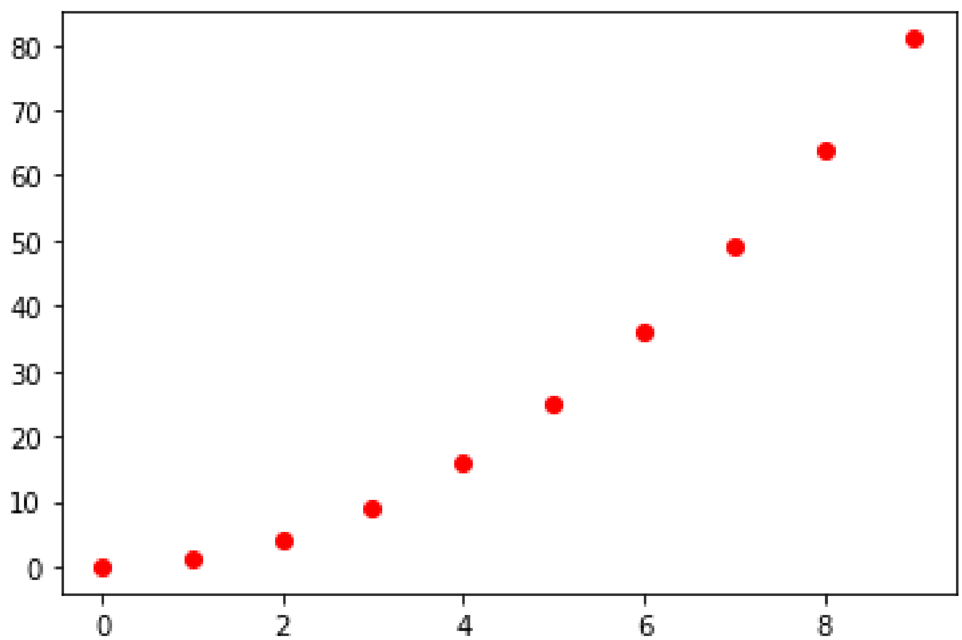
Simple plot In the following cell, we plot a function

[8]:
```
# Just to convince that things are easy:
x = np.arange(10) # define an array
plt.plot(x, x**2); # so quickly plotted... Notice the ";" at the end of the␣
↪line -> ;
```

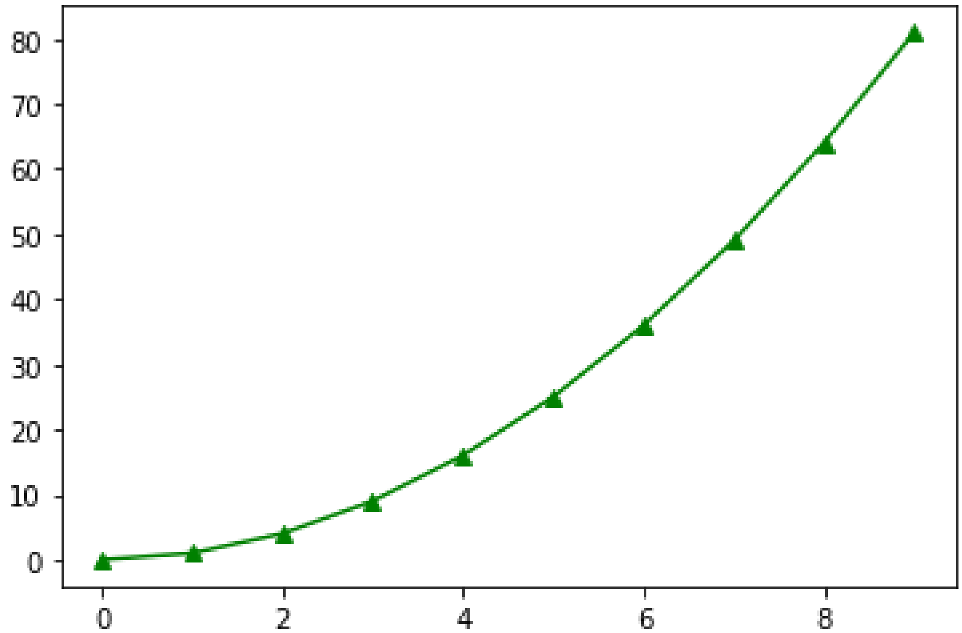


Controling colors and symbols

[12]:
```
plt.plot(x, x**2, 'or');
```



[22]:
```
plt.plot(x, x**2, c='green', marker='^', linestyle='-');
```

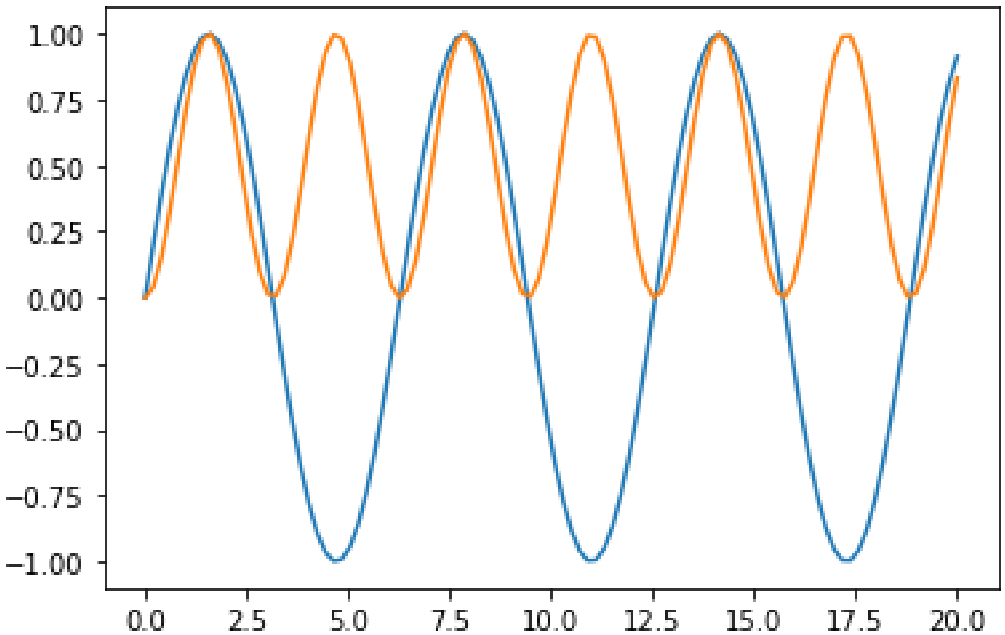


[24]:
```
# To illustrate the possibilities of the interactive window:
%matplotlib tk
plt.plot(x, x**2, '*b', linestyle='-') ;
```

[27]:
```
# Back to the inline graphics mode
%matplotlib inline
```

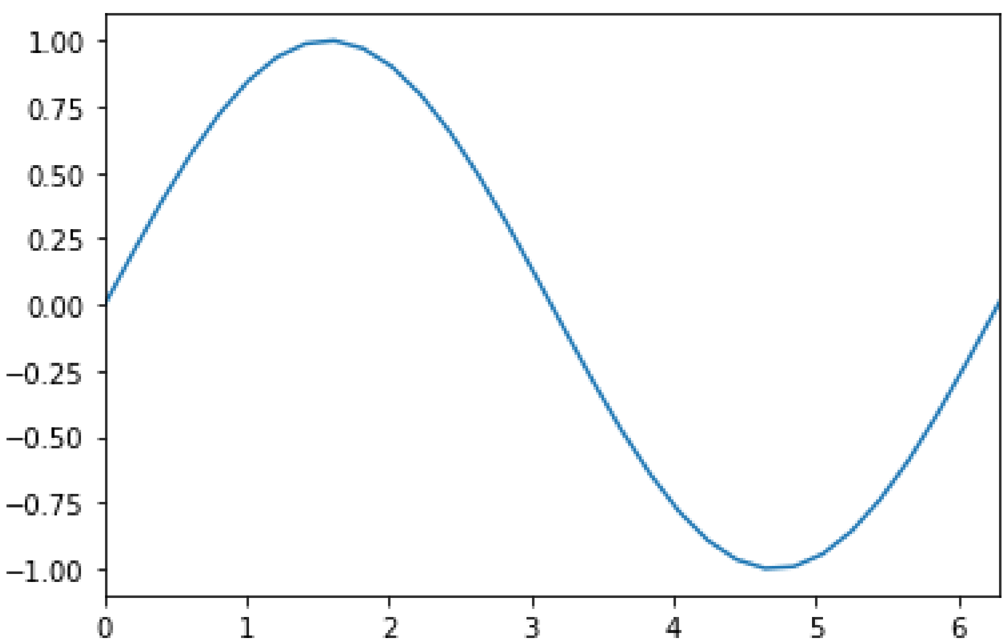
Overplot

[28]:
```
x = np.linspace(0, 20, 100)  # 100 evenly-spaced values from 0 to 20
y = np.sin(x)

plt.plot(x, y)
plt.plot(x, y**2); # overplot by default;
```



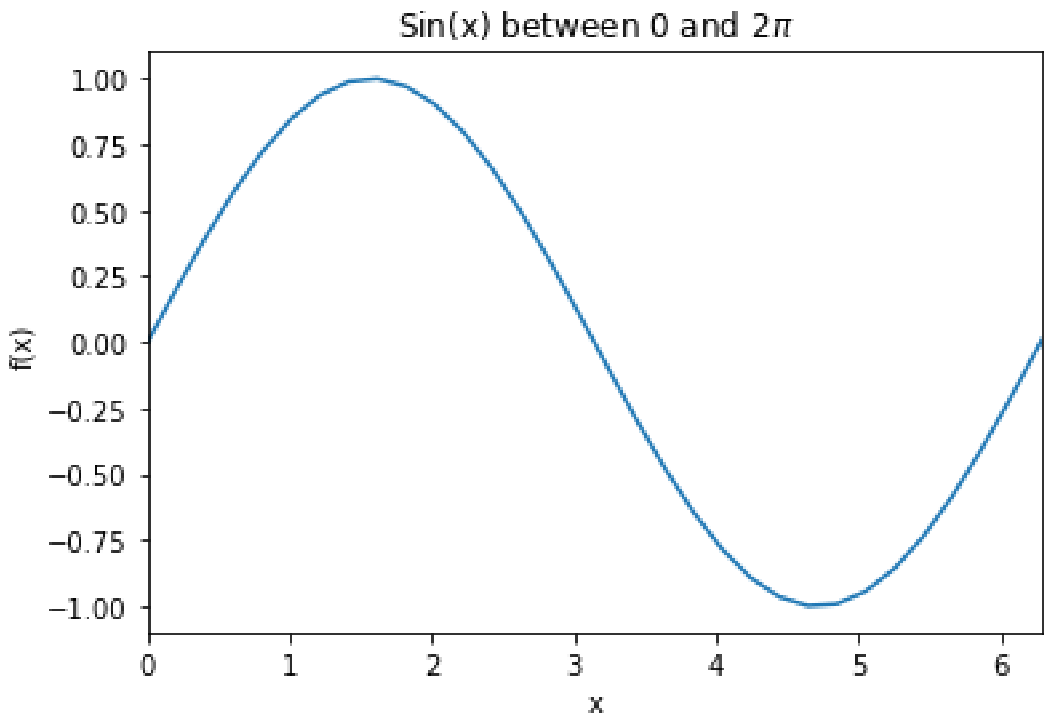
Fixing axes limits

[29]:
```
plt.plot(x, y)
plt.xlim(0., np.pi*2); # Take car, it's NOT plt.xlim = (1, 2), this would ERASE
 ↪the xlim method from plt!!!;
```



Labels, titles

[30]:
```
plt.plot(x, y)
plt.xlim((0., np.pi*2))
plt.title(r'Sin(x) between 0 and $2\pi$')
plt.xlabel('x')
plt.ylabel('f(x)');
```



plot method documentation

[31]:
```
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 and default to `range(len(y))`.

    Commonly, these parameters are 1D arrays.

    They can also be scalars, or two-dimensional (in that case, the
    columns represent separate data sets).

    These arguments cannot be passed as keywords.

fmt : str, optional
    A format string, e.g. 'ro' for red circles. See the *Notes*
    section for a full description of the format strings.

    Format strings are just an abbreviation for quickly setting
    basic line properties. All of these and more can also be
    controlled by keyword arguments.

    This argument cannot be passed as keyword.

data : indexable object, optional
    An object with labelled data. If given, provide the label names to
    plot in *x* and *y*.
```

```
        .. note::
            Technically there's a slight ambiguity in calls where the
            second label is a valid *fmt*. `plot('n', 'o', data=obj)`
            could be `plt(x, y)` or `plt(y, fmt)`. In such cases,
            the former interpretation is chosen, but a warning is issued.
            You may suppress the warning by adding an empty format string
            `plot('n', 'o', '', data=obj)`.

    Other Parameters
    ----------------
    scalex, scaley : bool, optional, default: True
        These parameters determined if the view limits are adapted to
        the data limits. The values are passed on to `autoscale_view`.

    **kwargs : `.Line2D` properties, optional
        *kwargs* are used to specify properties like a line label (for
        auto legends), linewidth, antialiasing, marker face color.
        Example::

        >>> plot([1,2,3], [1,2,3], 'go-', label='line 1', linewidth=2)
        >>> plot([1,2,3], [1,4,9], 'rs',  label='line 2')

        If you make multiple lines with one plot command, the kwargs
        apply to all those lines.

        Here is a list of available `.Line2D` properties:

      agg_filter: a filter function, which takes a (m, n, 3) float array and a
dpi value, and returns a (m, n, 3) array
      alpha: float
      animated: bool
      antialiased or aa: bool
      clip_box: `.Bbox`
      clip_on: bool
      clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
      color or c: color
      contains: callable
      dash_capstyle: {'butt', 'round', 'projecting'}
      dash_joinstyle: {'miter', 'round', 'bevel'}
      dashes: sequence of floats (on/off ink in points) or (None, None)
      drawstyle or ds: {'default', 'steps', 'steps-pre', 'steps-mid', 'steps-
post'}, default: 'default'
      figure: `.Figure`
      fillstyle: {'full', 'left', 'right', 'bottom', 'top', 'none'}
      gid: str
      in_layout: bool
      label: object
      linestyle or ls: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
```

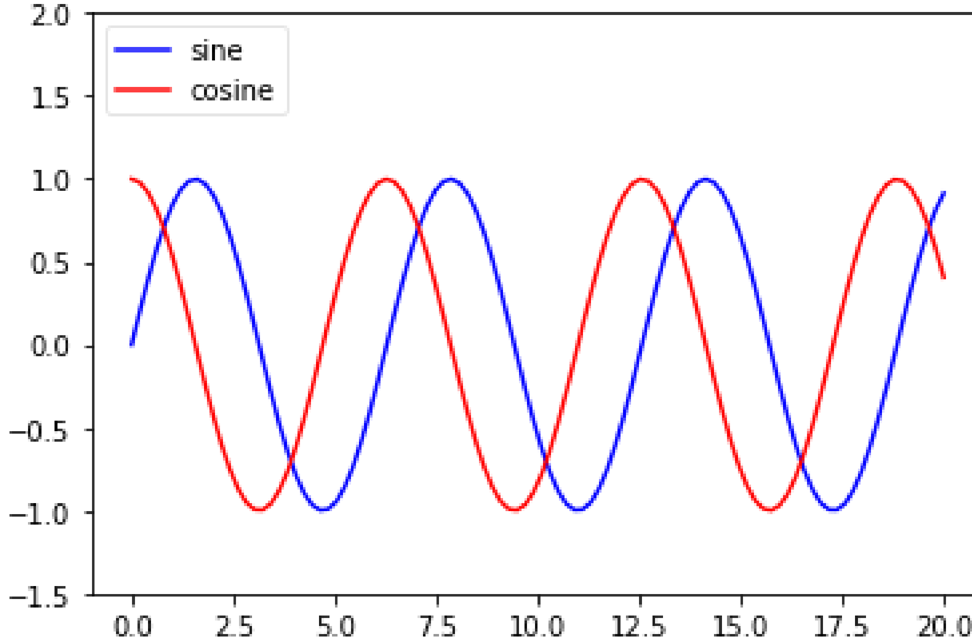
```
      linewidth or lw: float
      marker: marker style
      markeredgecolor or mec: color
      markeredgewidth or mew: float
      markerfacecolor or mfc: color
      markerfacecoloralt or mfcalt: color
      markersize or ms: float
      markevery: None or int or (int, int) or slice or List[int] or float or
(float, float)
      path_effects: `.AbstractPathEffect`
      picker: float or callable[[Artist, Event], Tuple[bool, dict]]
      pickradius: float
      rasterized: bool or None
      sketch_params: (scale: float, length: float, randomness: float)
      snap: bool or None
      solid_capstyle: {'butt', 'round', 'projecting'}
      solid_joinstyle: {'miter', 'round', 'bevel'}
      transform: `matplotlib.transforms.Transform`
      url: str
      visible: bool
      xdata: 1D array
      ydata: 1D array
      zorder: float

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

    See Also
    --------
    scatter : XY scatter plot with markers of varying size and/or color (
        sometimes also called bubble chart).

    Notes
    -----
    **Format Strings**

    A format string consists of a part for color, marker and line::

        fmt = '[marker][line][color]'

    Each of them is optional. If not provided, the value from the style
    cycle is used. Exception: If ``line`` is given, but no ``marker``,
    the data will be a line without markers.

    Other combinations such as ``[color][marker][line]`` are also
    supported, but note that their parsing may be ambiguous.
```

```
**Markers**

=============    ===============================
character        description
=============    ===============================
``'.'``          point marker
``','``          pixel marker
``'o'``          circle marker
``'v'``          triangle_down marker
``'^'``          triangle_up marker
``'<'``          triangle_left marker
``'>'``          triangle_right marker
``'1'``          tri_down marker
``'2'``          tri_up marker
``'3'``          tri_left marker
``'4'``          tri_right marker
``'s'``          square marker
``'p'``          pentagon marker
``'*'``          star marker
``'h'``          hexagon1 marker
``'H'``          hexagon2 marker
``'+'``          plus marker
``'x'``          x marker
``'D'``          diamond marker
``'d'``          thin_diamond marker
``'|'``          vline marker
``'_'``          hline marker
=============    ===============================

**Line Styles**

=============    ===============================
character        description
=============    ===============================
``'-'``          solid line style
``'--'``         dashed line style
``'-.'``         dash-dot line style
``':'``          dotted line style
=============    ===============================

Example format strings::

    'b'    # blue markers with default shape
    'or'   # red circles
    '-g'   # green solid line
    '--'   # dashed line with default color
    '^k:'  # black triangle_up markers connected by a dotted line
```

```
**Colors**

The supported color abbreviations are the single letter codes

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

and the ``'CN'`` colors that index into the default property cycle.

If the color is the only part of the format string, you can
additionally use any  `matplotlib.colors` spec, e.g. full names
(``'green'``) or hex strings (``'#008000'``).
```

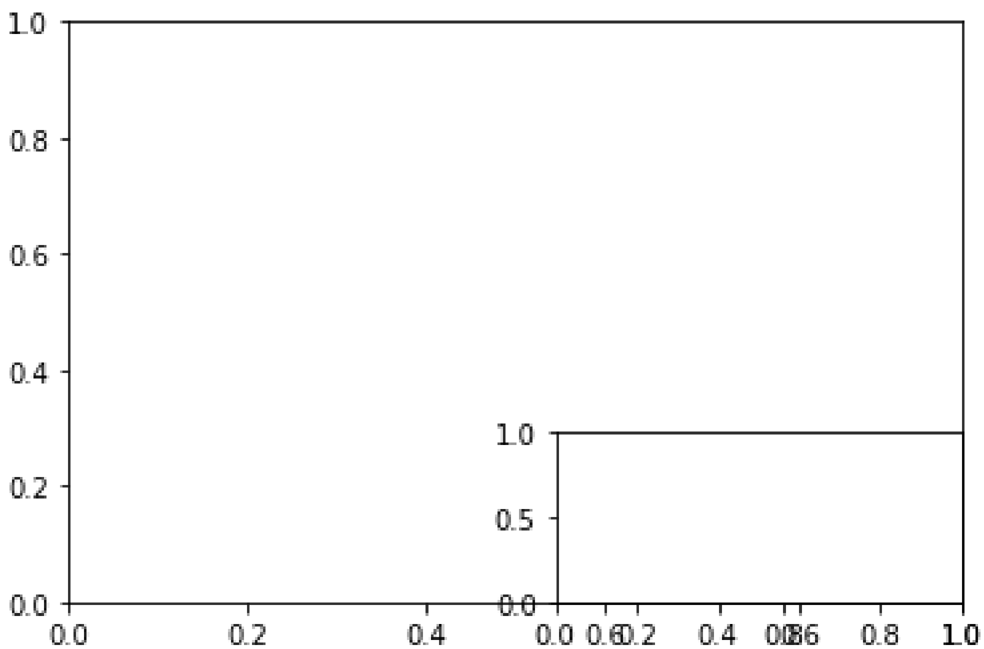
Legends

[42]:
```python
x = np.linspace(0, 20, 100)
y1 = np.sin(x)
y2 = np.cos(x)

plt.plot(x, y1, '-b', label='sine')
plt.plot(x, y2, '-r', label='cosine')
plt.legend(loc='upper left', fancybox=True, framealpha=0.5)
plt.ylim((-1.5, 2.0));
```

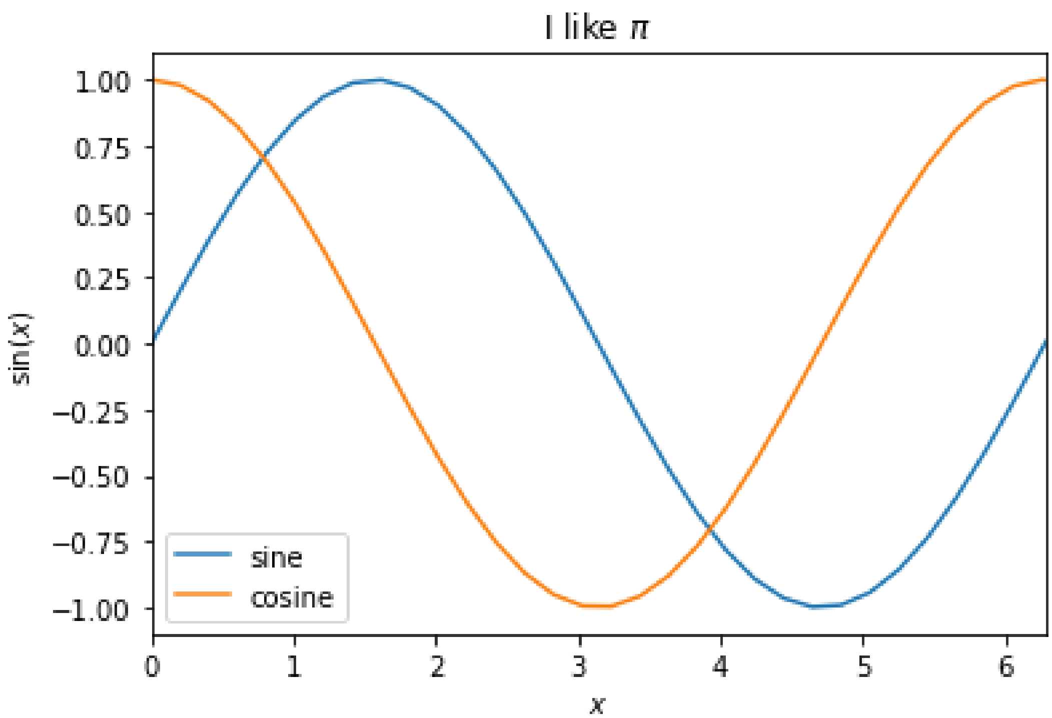


Object oriented way

[49]:
```
fig = plt.figure()  # a new figure window
ax = fig.add_subplot(1,1,1)  # specify (nrows, ncols, axnum)
ax2 = fig.add_subplot(3, 2, 6)  # specify (nrows, ncols, axnum)
# same as ax = fig.add_subplot()
```



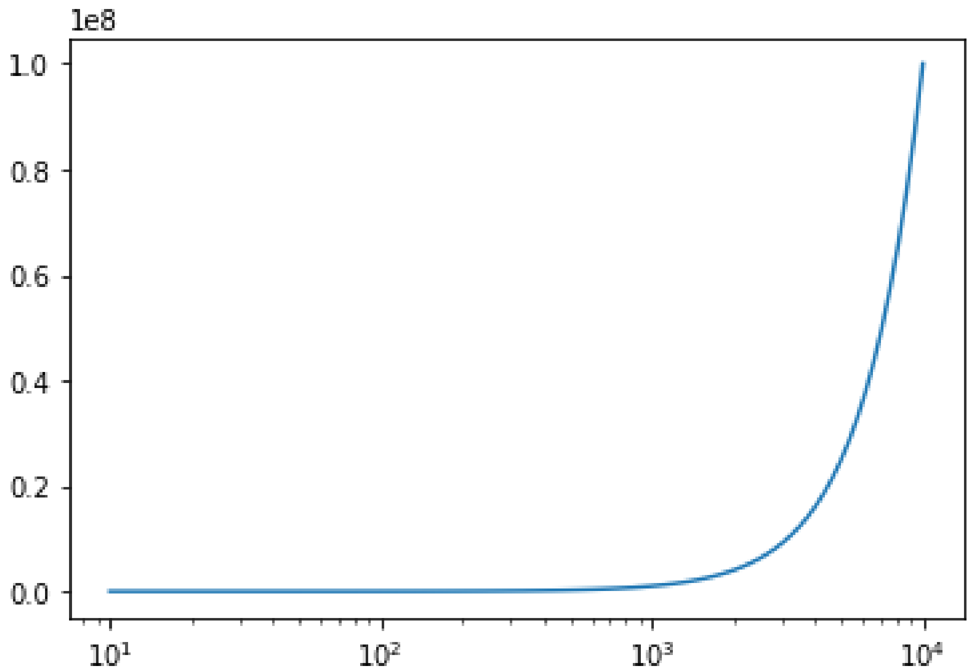
[50]:
```
fig, ax = plt.subplots()  # one command way
ax.plot(x, y1)
ax.plot(x, y2)
ax.set_xlim(0., 2*np.pi)
ax.legend(['sine', 'cosine'], loc='best') # If the legends are not already␣
↪defined in the plot call
ax.set_xlabel("$x$")
ax.set_ylabel("$\sin(x)$")
ax.set_title("I like $\pi$");
```



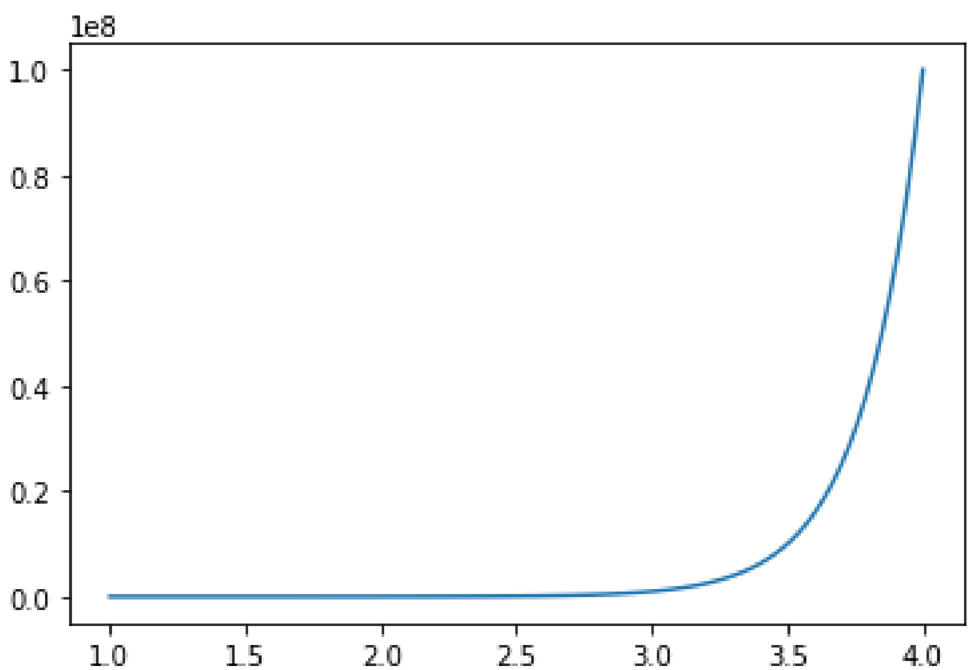
[52]:
```
# The following outputs a HUGE quantity of information! I comment it for now
#help(ax)
```

log plots

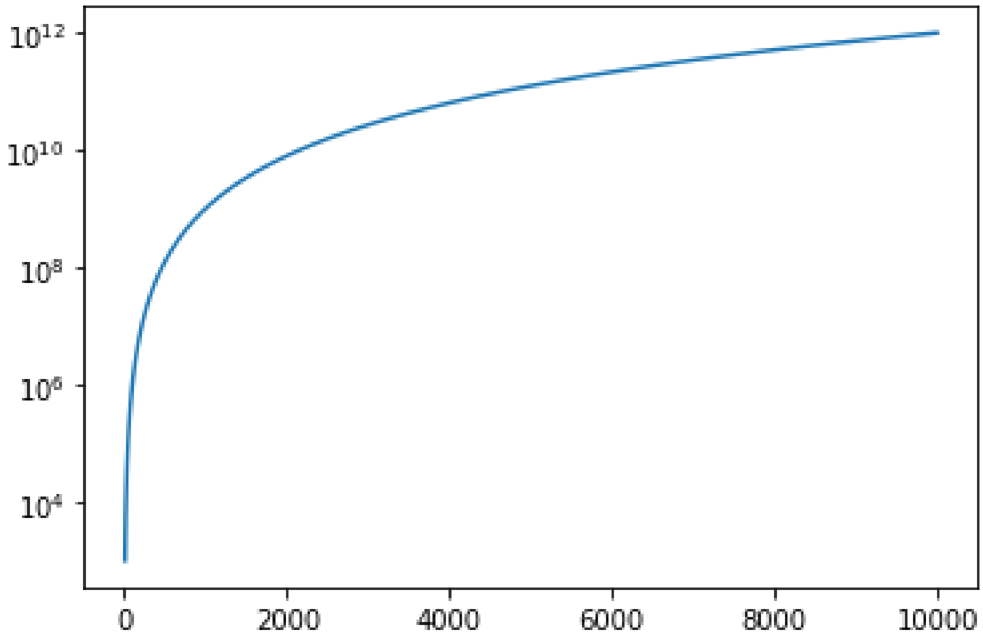
[55]:
```
xl = np.logspace(1, 4, 100)
fig, ax = plt.subplots()
ax.semilogx(xl, xl**2);
```



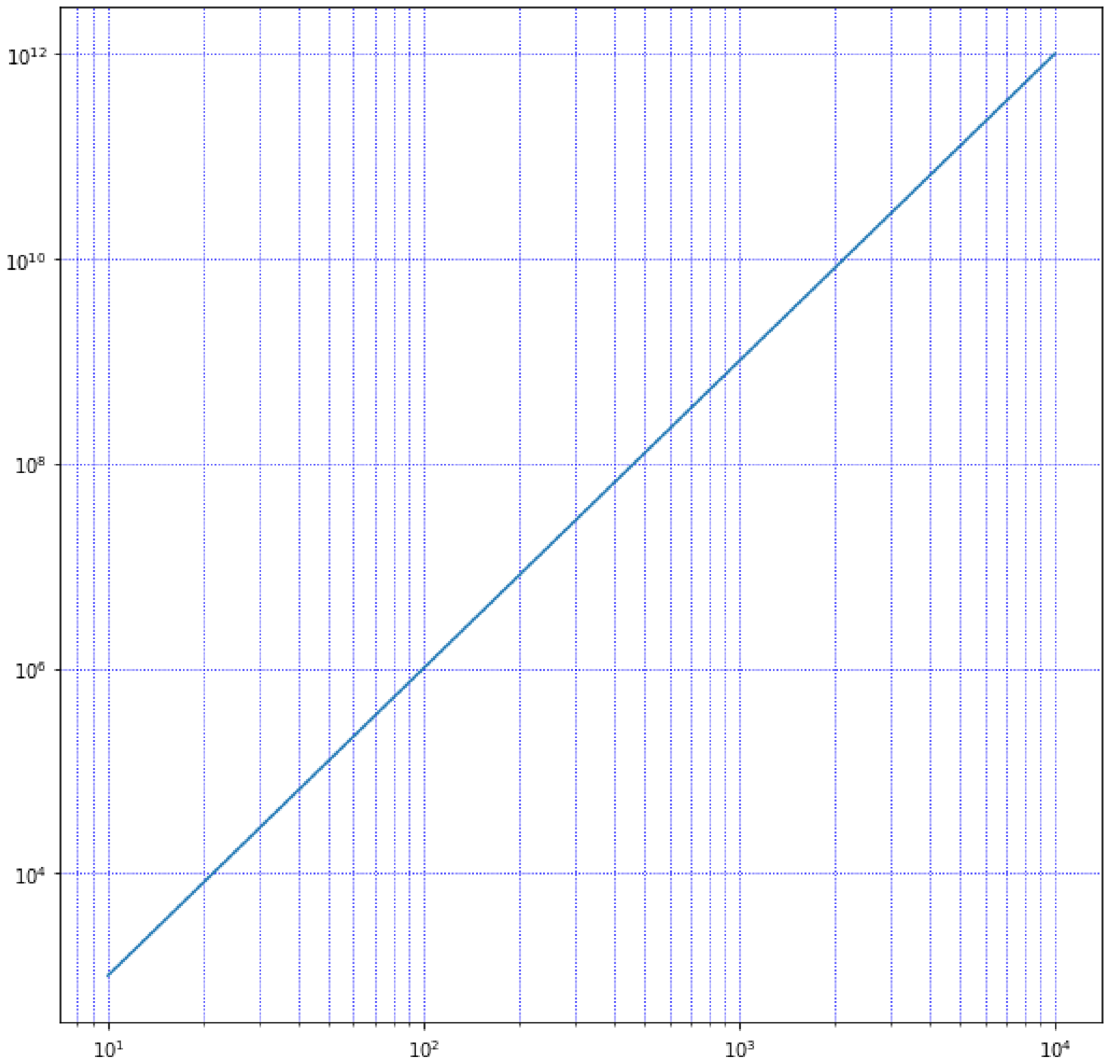
[56]:
```
fig, ax = plt.subplots()
ax.plot(np.log10(xl), xl**2);
```



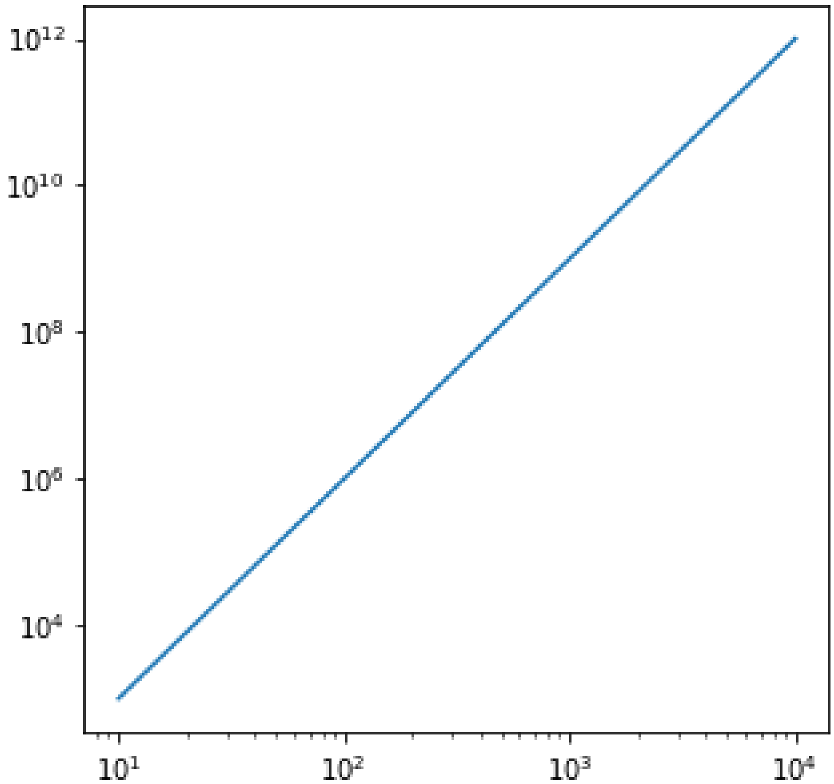
[57]:
```
fig, ax = plt.subplots()
ax.semilogy(xl, xl**3);
```



[58]:
```
fig, ax = plt.subplots(figsize=(10,10))
ax.loglog(xl, xl**3);
ax.grid(which="both",ls=":", c='blue')
```

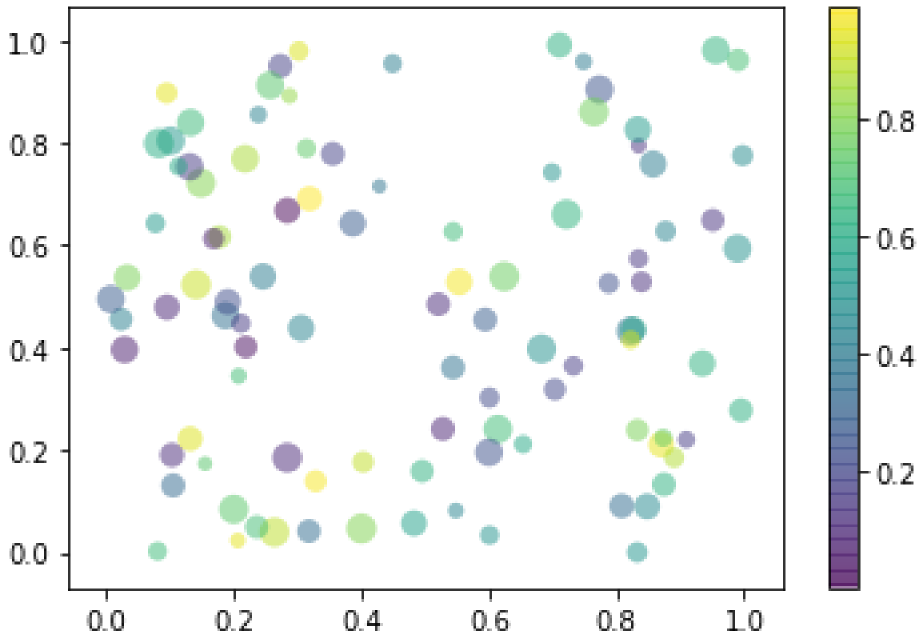


[62]:
```
# log axes can be defined after the plot
fig, ax = plt.subplots(figsize=(5,5))
ax.plot(xl, xl**3)
ax.set_yscale('log')
ax.set_xscale('log')
```



Scatter

[74]:
```
xr = np.random.rand(100)
yr = np.random.rand(100)
cr = np.random.rand(100)
sr = np.random.rand(100)

fig, ax = plt.subplots()
sc = ax.scatter(xr, yr, c=cr, s=30 + sr*100, edgecolor='none', alpha=0.5); #
 ↪Sizes and colors depend on valyues of other variables
fig.colorbar(sc);
```

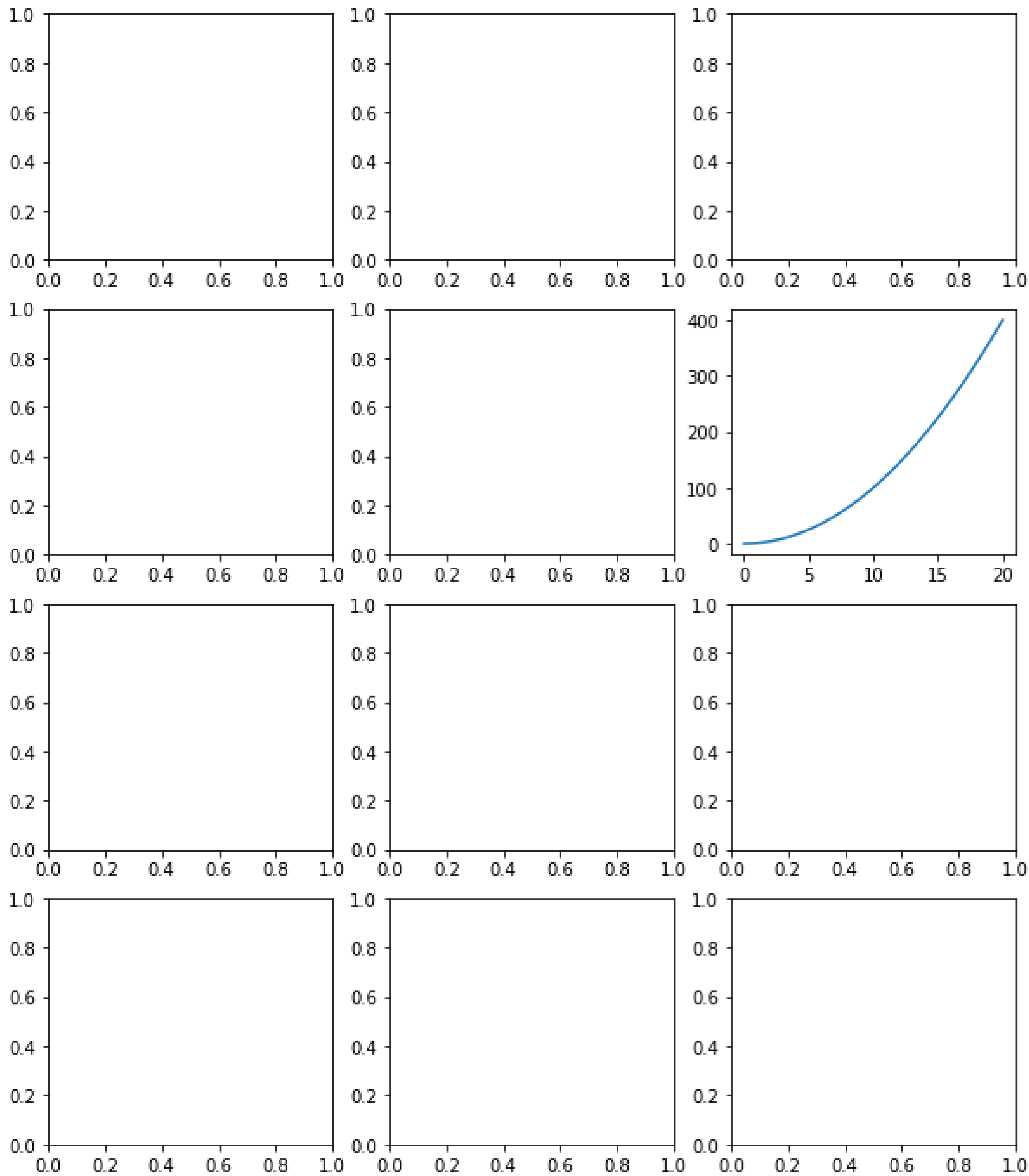


multiple plots

[96]:
```
fig, axes = plt.subplots(4, 3, figsize=(10,12))
print(axes.shape)
axes[1,2].plot(x,x**2)
```

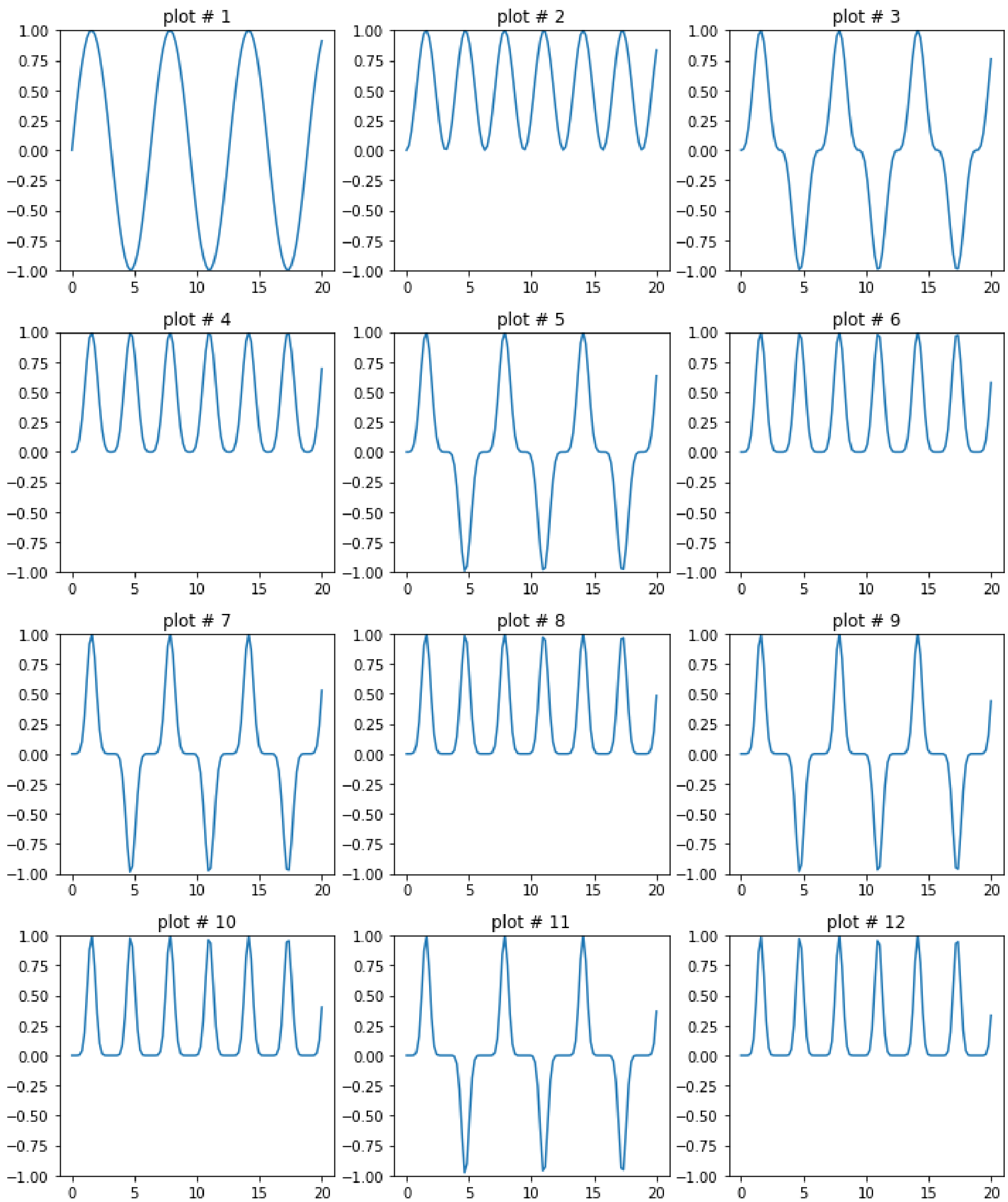
```
(4, 3)
```

[96]: [<matplotlib.lines.Line2D at 0x7f9bd58fc950>]



[82]:
```
x = np.linspace(0, 20, 100)
y1 = np.sin(x)

fig, axes = plt.subplots(4, 3, figsize=(10,12))
for i, ax in enumerate(axes.ravel()): # axes is a 2D array.. Need to ravel it␣
 ↪to run over every ax
    ax.set_title('plot # {}'.format(i+1))
    ax.plot(x, y1**(i+1))
```

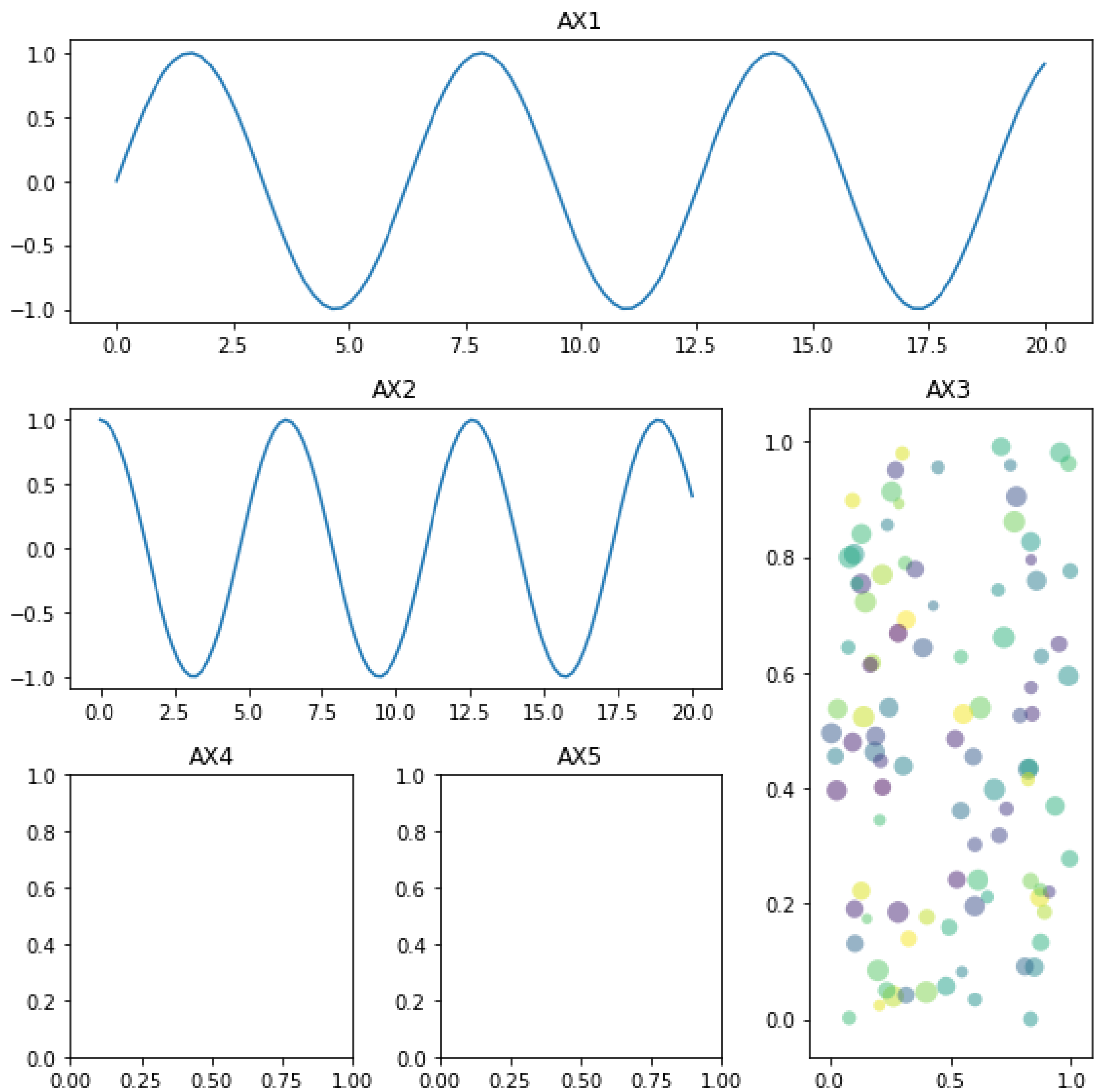
```
    ax.set_ylim((-1, 1))
fig.tight_layout() # Better output
```



[87]:

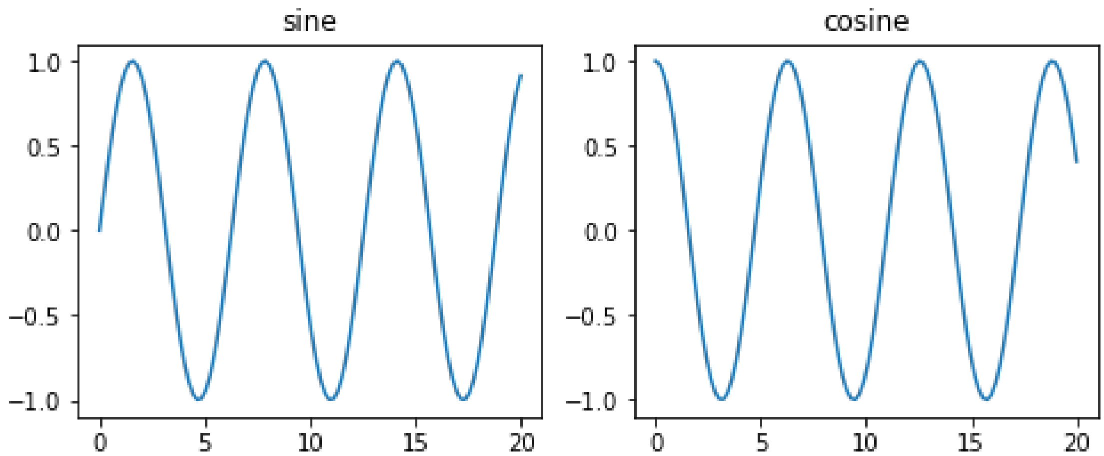
```
fig = plt.figure(figsize=(8, 8))
gs = plt.GridSpec(3, 3)
ax1 = fig.add_subplot(gs[0, :])
ax2 = fig.add_subplot(gs[1, :2])
```

```python
ax3 = fig.add_subplot(gs[1:, 2])
ax4 = fig.add_subplot(gs[2, 0]) # 3rd row, 1rst column
ax5 = fig.add_subplot(gs[2, 1])
ax1.plot(x, y1)
ax1.set_title('AX1')
ax2.plot(x, y2)
ax2.set_title('AX2')
ax3.scatter(xr, yr, c=cr, s=30+sr*100, edgecolor='none', alpha=0.5)
ax3.set_title('AX3')
ax4.set_title('AX4')
ax5.set_title('AX5')
fig.tight_layout();
#etc...
```

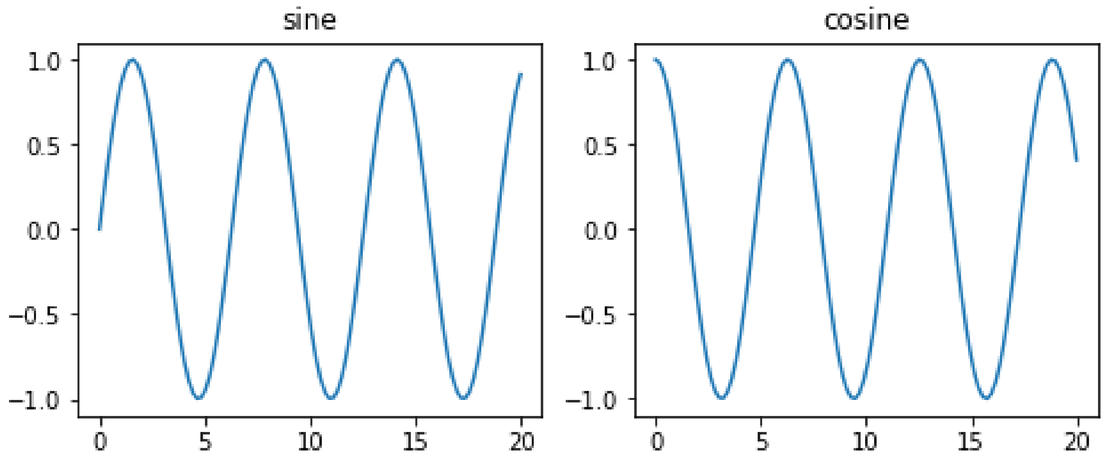


Order of the commands

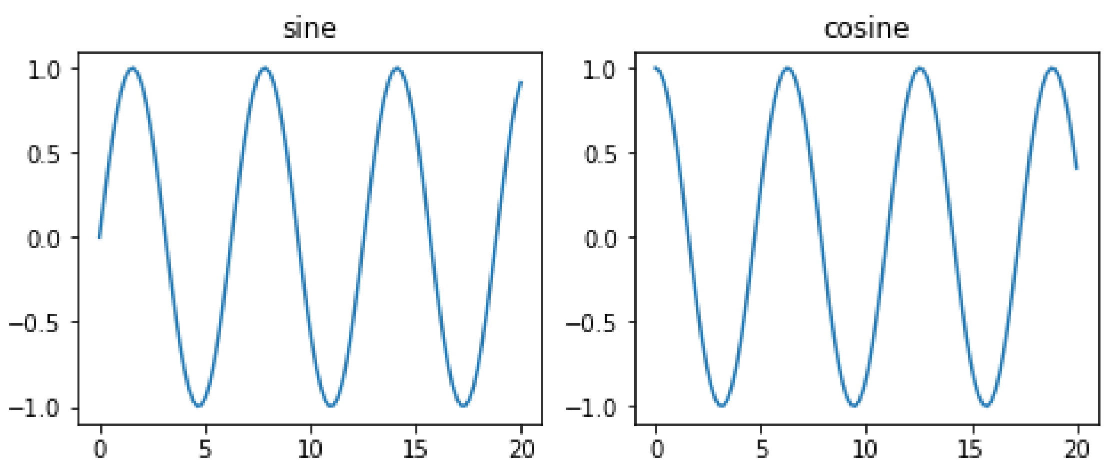
[97]:
```
fig1 = plt.figure(figsize=(8, 3))

ax1 = fig1.add_subplot(1, 2, 1)
ax1.plot(x, np.sin(x))
ax1.set_title('sine')

ax2 = fig1.add_subplot(1, 2, 2)
ax2.plot(x, np.cos(x))
ax2.set_title('cosine');
```



[99]:
```
fig1 = plt.figure(figsize=(8, 3))
ax1 = fig1.add_subplot(1, 2, 1)
ax2 = fig1.add_subplot(1, 2, 2)

ax1.plot(x, np.sin(x))
ax2.plot(x, np.cos(x))

ax1.set_title('sine') # you can go back to change ax1 and ax2 after plotting
ax2.set_title('cosine'); # They both are objects containing method to apply on␣
↪them;
```



[103]:
```
fig1, (ax1, ax2) = plt.subplots(1, 2, figsize=(8, 3))
# fig, axes = plt.subplots(1, 2, figsize=(8, 3))
# ax1 = axes[0]
# ax2 = axes[1]

ax1.plot(x, np.sin(x))
ax2.plot(x, np.cos(x))

ax1.set_title('sine') # you can go back to change ax1 and ax2 after plotting
ax2.set_title('cosine'); # They both are objects containing method to apply on␣
 ↪them
```

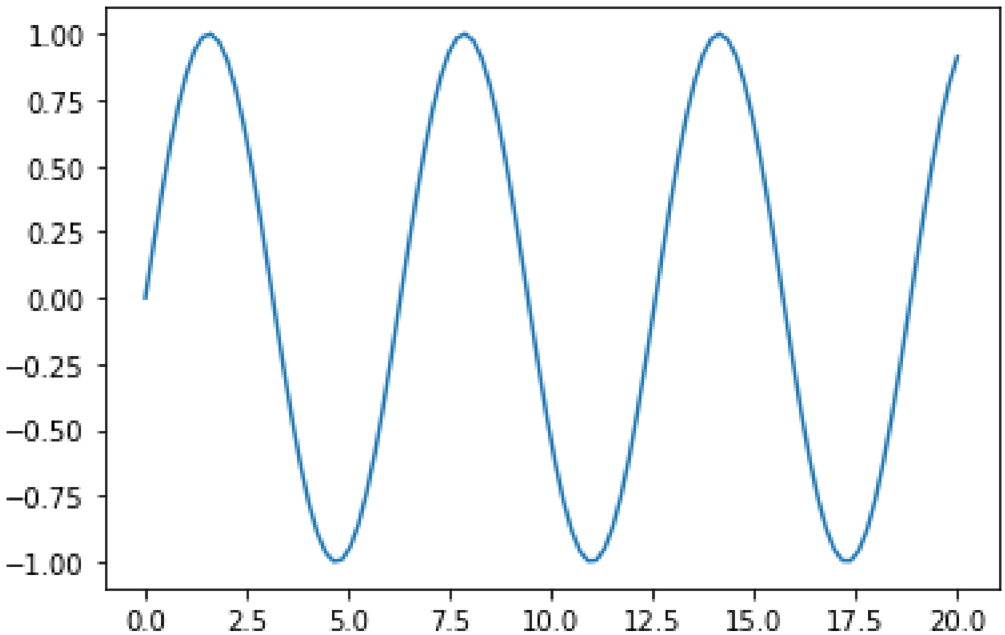


Everything is object

[106]:

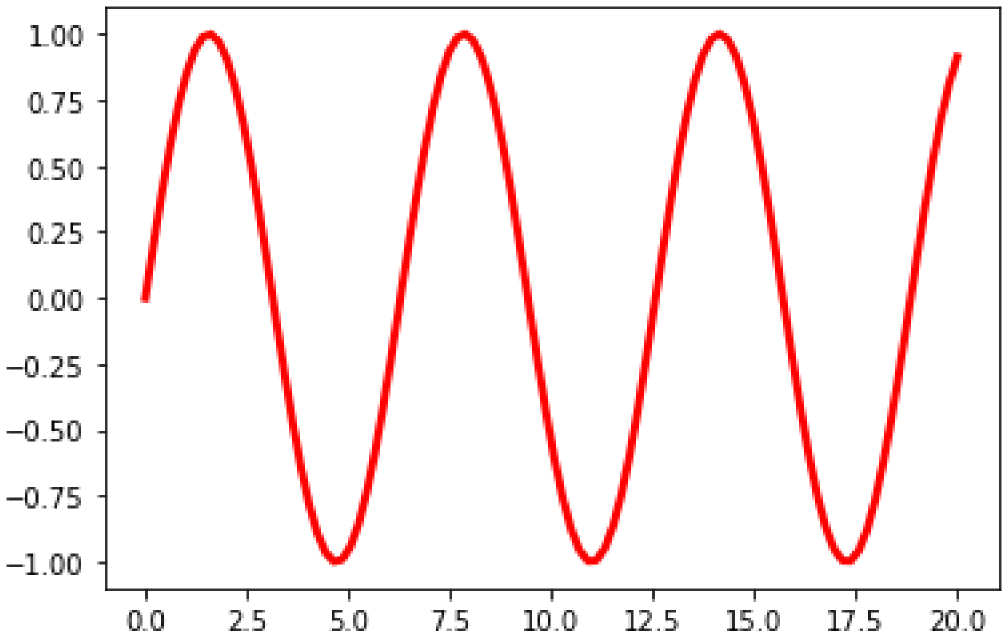
```
fig, ax = plt.subplots()
lines = ax.plot(x, np.sin(x))
print(type(lines))
print(len(lines))
```

```
<class 'list'>
1
```



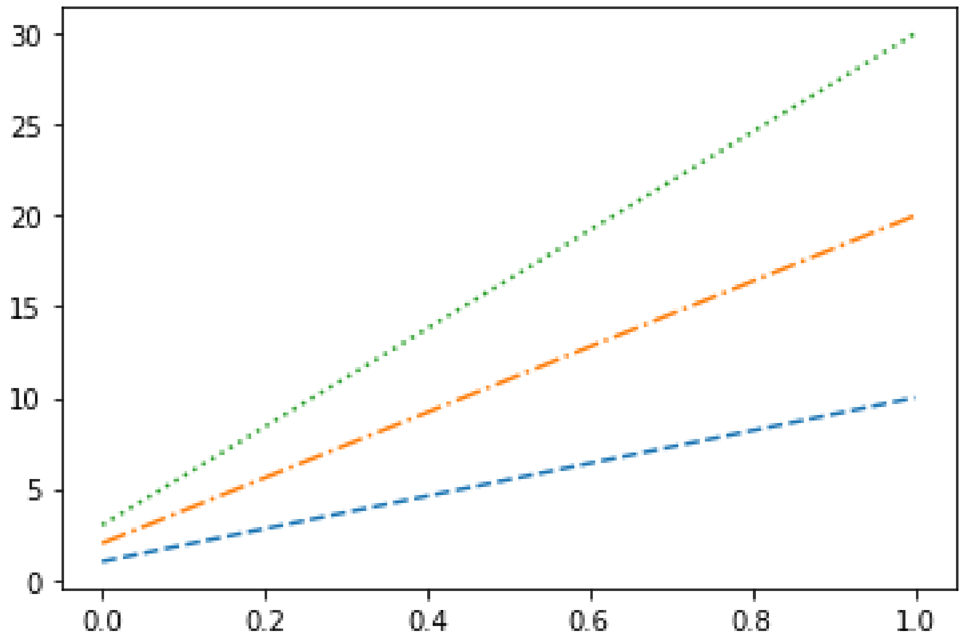
[111]:

```
fig, ax = plt.subplots()
lines = ax.plot(x, np.sin(x))
line = lines[0]
#help(line) # HUGE quantity of information
line.set_color('red')
line.set_linewidth(3)
fig.canvas.draw() # this is not necessary in notebook, but in scripts it is.
```

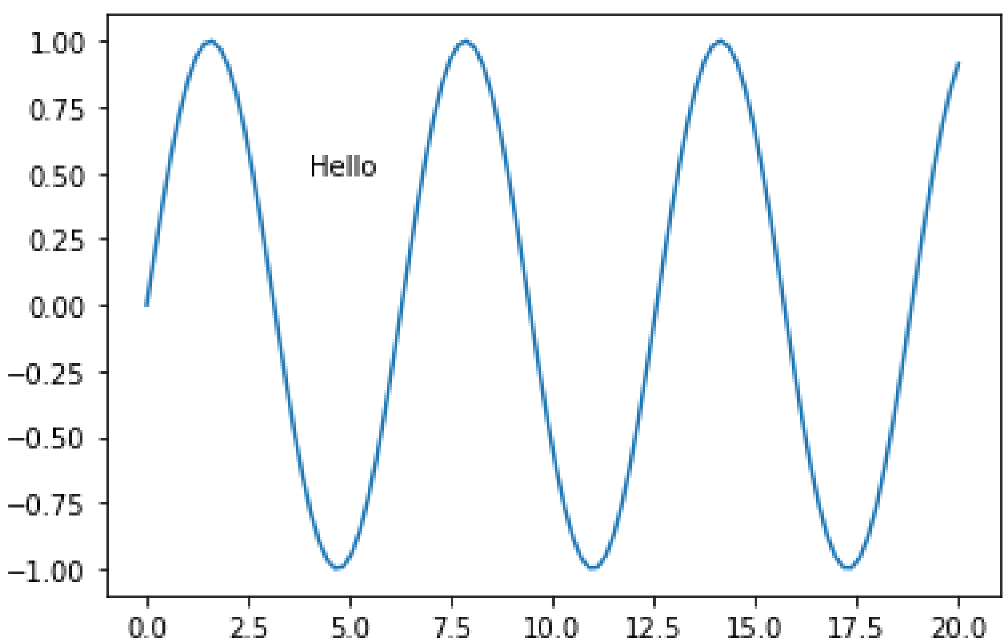


[142]:
```
fig, ax = plt.subplots()
lines = ax.plot([[1,2,3],[10,20,30]])
print(lines)

line_styles = ('--', '-.', ':')

for i, line in enumerate(lines):
    color = 'C{}'.format(i)
    lines[i].set_linestyle(line_styles[i])
    lines[i].set_color(color)
```

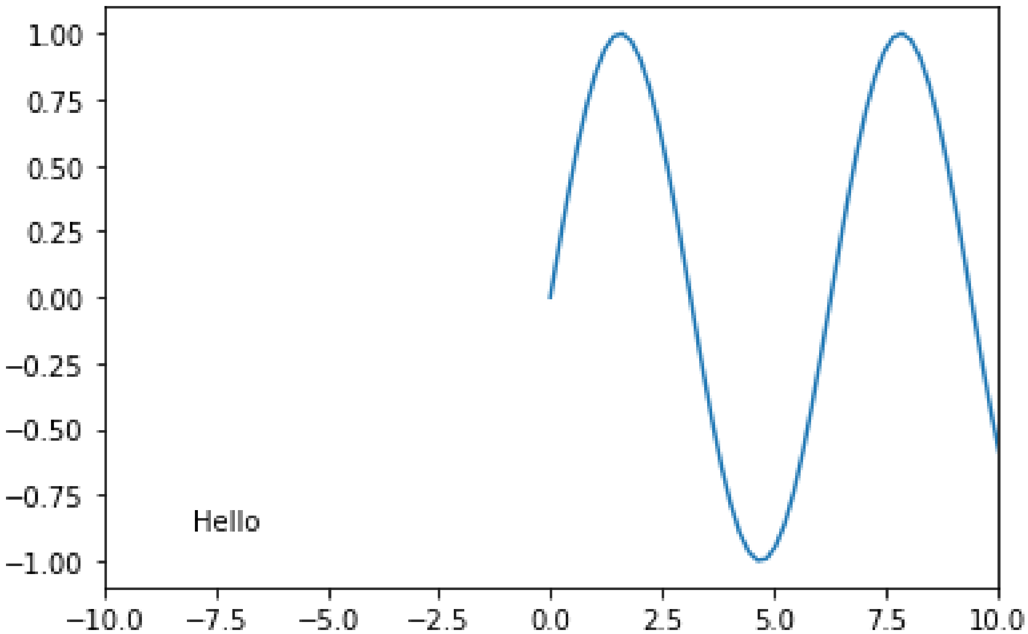
```
[<matplotlib.lines.Line2D object at 0x7f9bd4f85fd0>, <matplotlib.lines.Line2D
object at 0x7f9bd4b0da90>, <matplotlib.lines.Line2D object at 0x7f9bd4b0d550>]
```



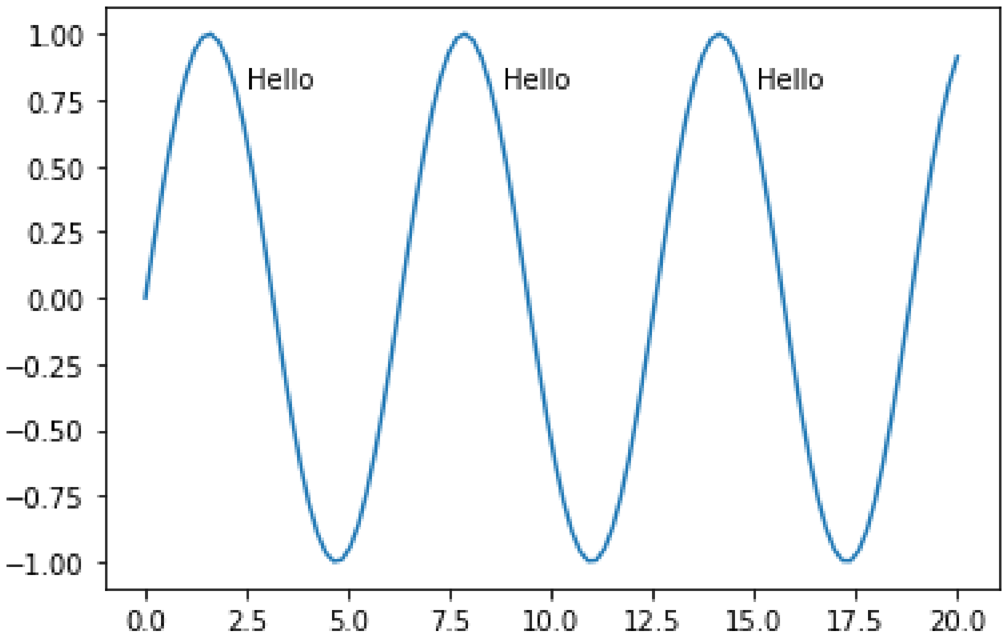
[113]:
```
fig, ax = plt.subplots()
lines = ax.plot(x, np.sin(x))
ax.text(4, 0.5, "Hello");
```



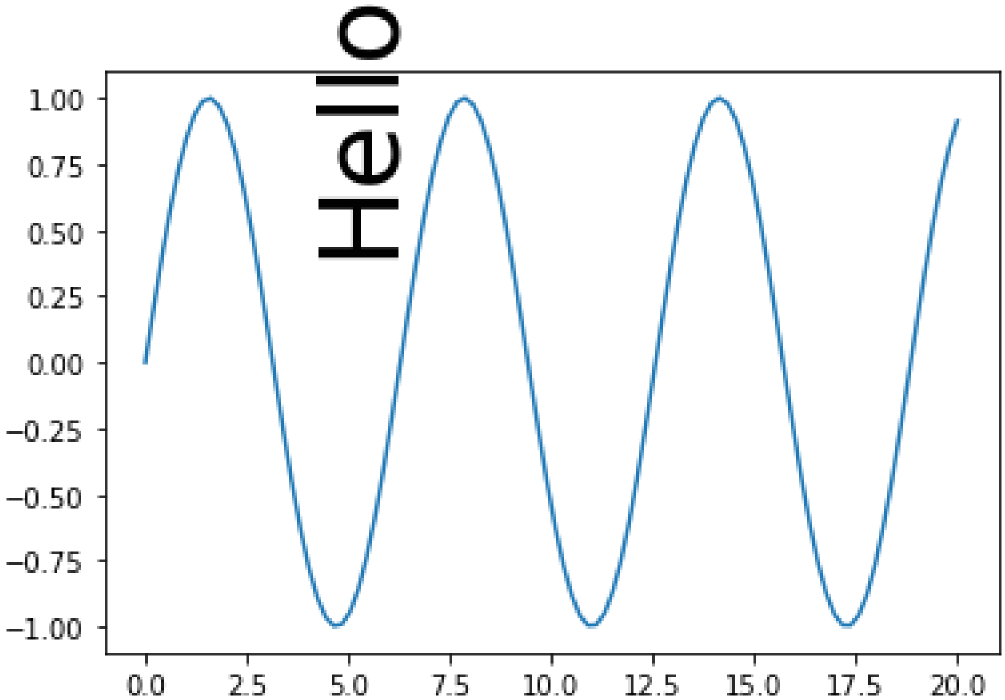
[118]:
```
fig, ax = plt.subplots()
lines = ax.plot(x, np.sin(x))
ax.set_xlim(-10,10)
ax.text(0.1, 0.1, "Hello", transform=ax.transAxes);
```



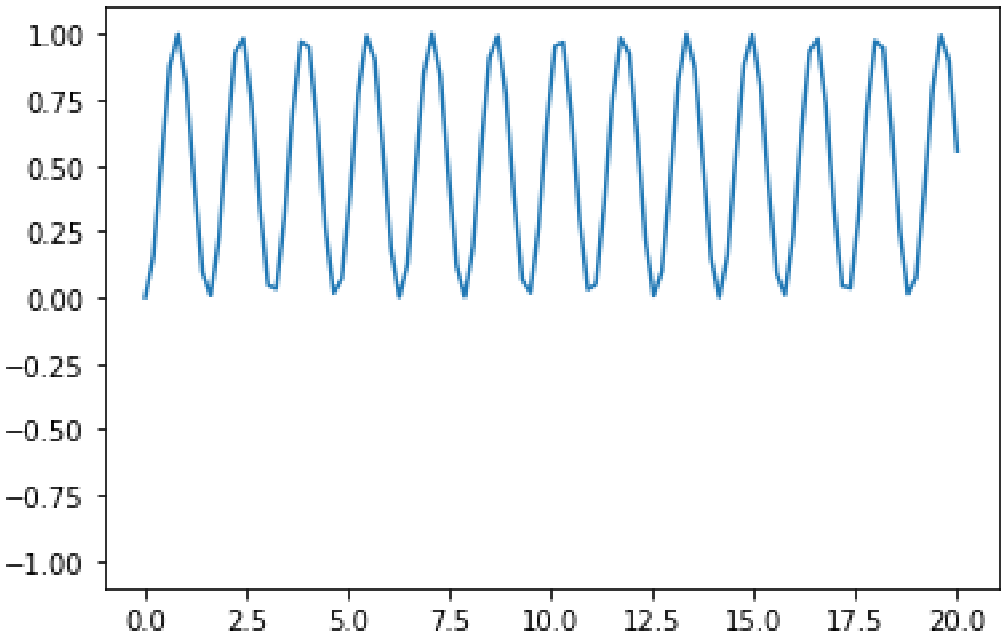
[119]:
```
fig, ax = plt.subplots()
lines = ax.plot(x, np.sin(x))
for x_text in 2.5+2*np.pi*(np.arange(3)):
    ax.text(x_text, 0.8, "Hello")
```



[120]:
```
fig, ax = plt.subplots()
lines = ax.plot(x, np.sin(x))
txt = ax.text(4, 0.5, "Hello")
txt.set_rotation(90)
txt.set_size(40)
```



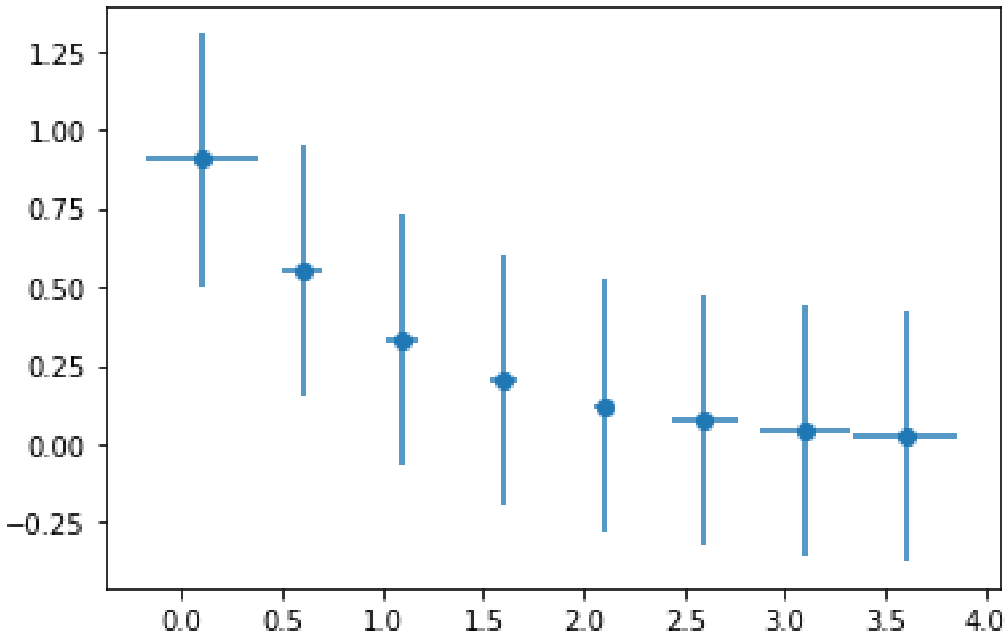
[129]:
```
x = np.linspace(0, 20, 100)
fig, ax = plt.subplots()
ax.set_ylim((-1.1,1.1))
lines = ax.plot(x, np.sin(x)) # !!! data will change latter
lines[0].set_ydata(np.sin(2 * x)**2) # Change the data themselve!!!
```



Error bars

[144]:
```
x = np.arange(0.1, 4, 0.5)
y = np.exp(-x)
xerr = np.random.rand(len(x))*0.3
fig, ax = plt.subplots()
eb = ax.errorbar(x, y, xerr=xerr, yerr=0.4, fmt='o');
print(type(eb))
```

```
<class 'matplotlib.container.ErrorbarContainer'>
```



Sharing axes

[146]:
```
%matplotlib tk
fig, axes = plt.subplots(2, sharex=True)
axes[0].plot(x, y)
axes[0].set_title('Sharing X axis')
axes[1].scatter(x, y);
```

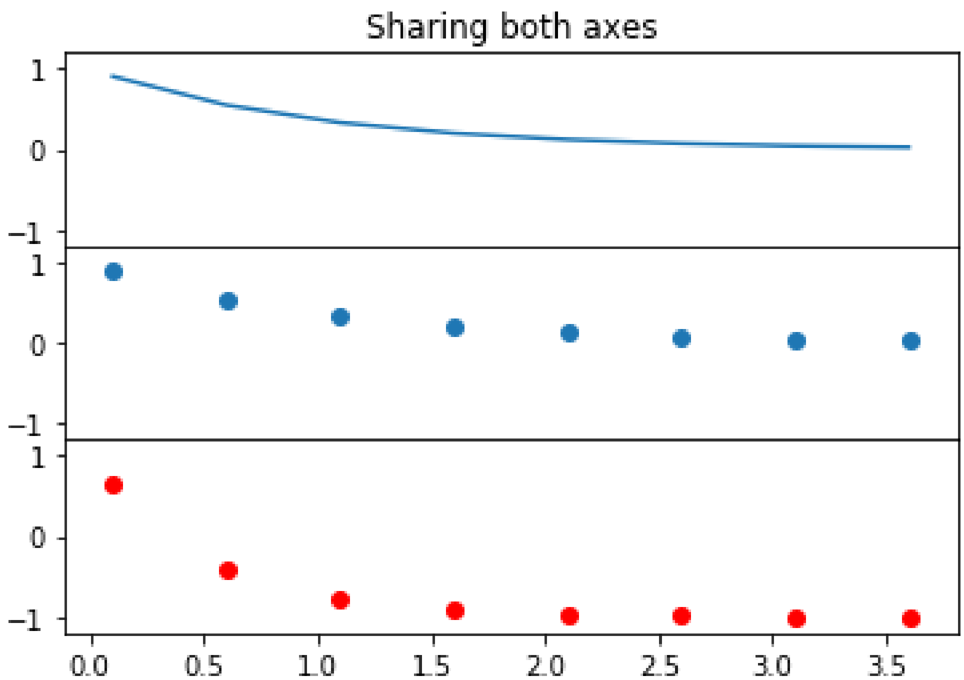
[148]:
```
f, (ax1, ax2) = plt.subplots(1, 2, sharey=True) # Unpacking the axes
ax1.plot(x, y)
f.suptitle('Main TITLE')
ax1.set_title('Sharing Y axis')
ax2.scatter(x, y);
```

[149]:
```
fig, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)
ax1.plot(x, y)
ax1.set_title('Sharing both axes')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')
# Fine-tune figure; make subplots close to each other
fig.subplots_adjust(hspace=0)
```

[151]:
```
%matplotlib inline
```

[159]:
```
fig, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)
ax1.plot(x, y)
```

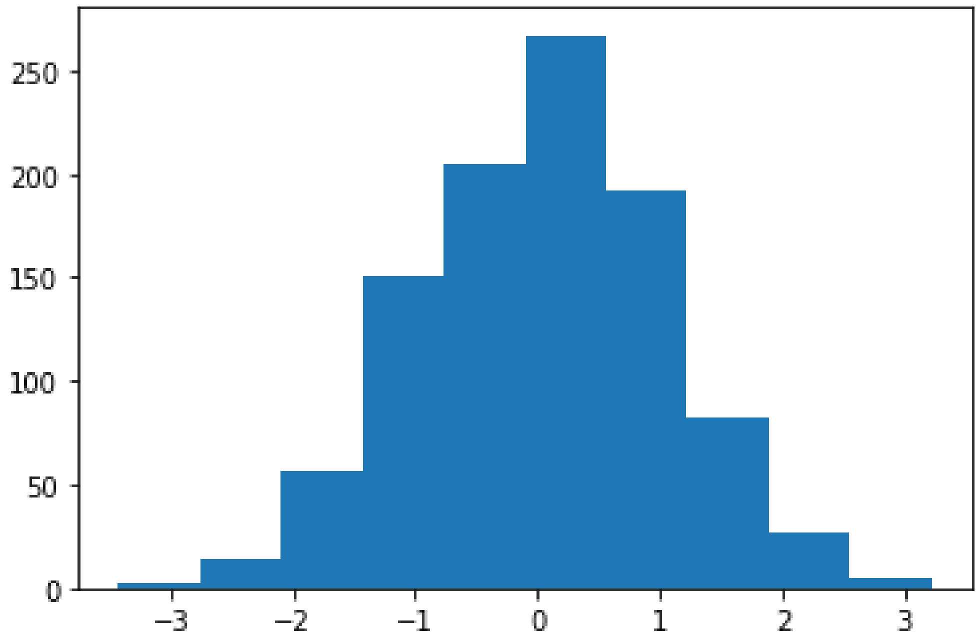
```
ax1.set_title('Sharing both axes')
ax2.scatter(x, y)
ax3.scatter(x, 2 * y ** 2 - 1, color='r')
# Fine-tune figure; make subplots close to each other
fig.subplots_adjust(hspace=0)
for ax in (ax1, ax2, ax3):
    ax.set_ylim(-1.2, 1.2)
```



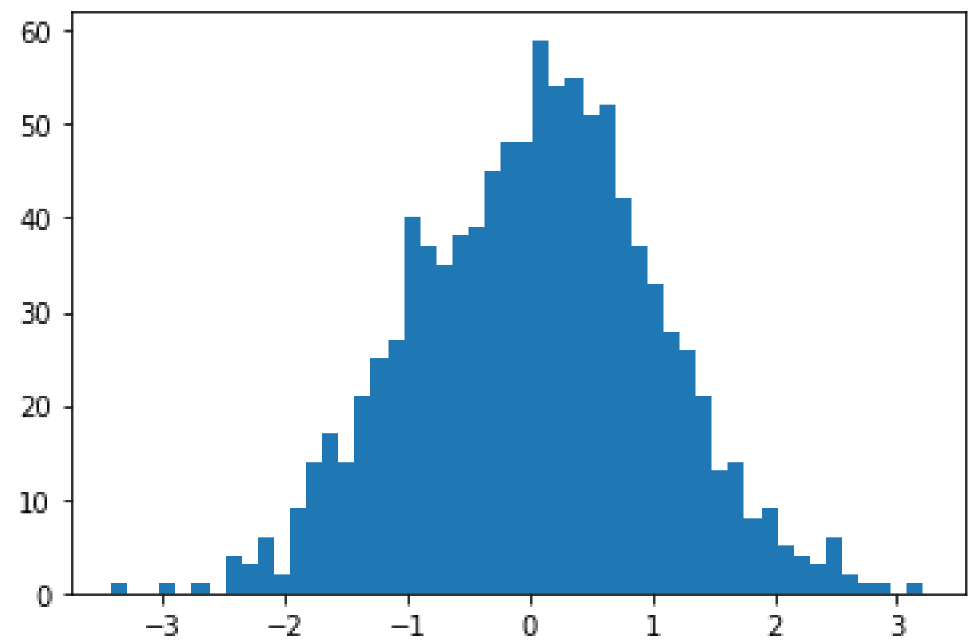
Histograms

[160]:
```
x = np.random.normal(size=1000)
fig, ax = plt.subplots()
H = ax.hist(x)
print(H)
```

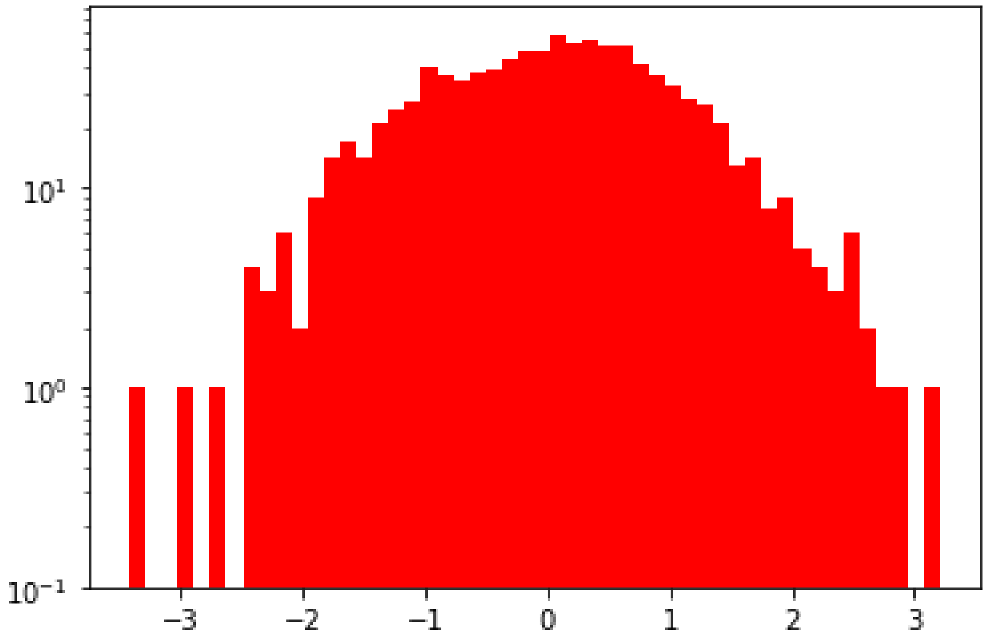
```
(array([  2.,  14.,  56., 150., 205., 267., 192.,  82.,  27.,   5.]),
array([-3.42280519, -2.75887292, -2.09494064, -1.43100837, -0.7670761 ,
       -0.10314383,  0.56078844,  1.22472072,  1.88865299,  2.55258526,
        3.21651753]), <a list of 10 Patch objects>)
```



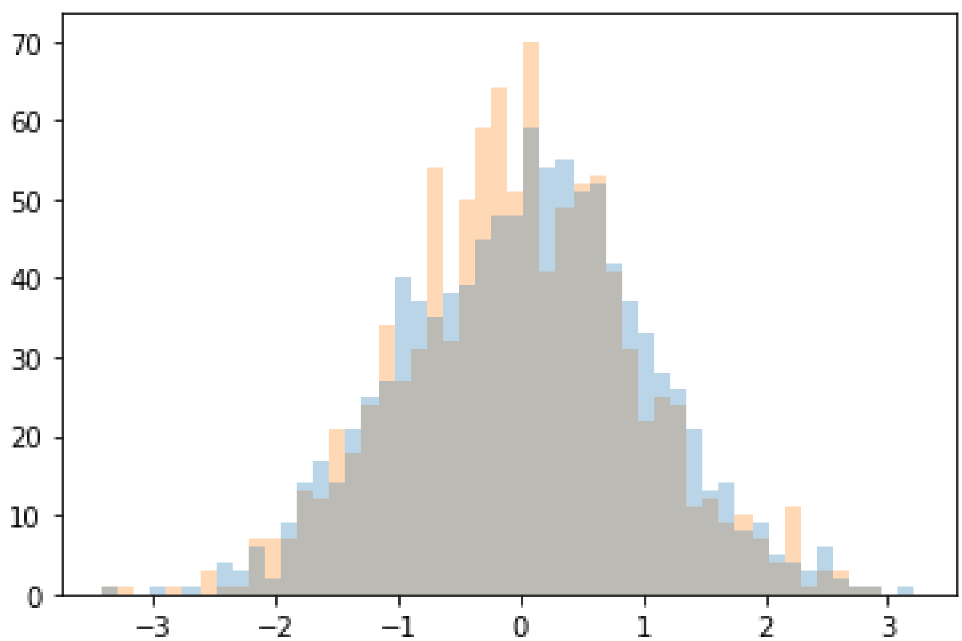
[163]:
```
fig, ax = plt.subplots()
H = ax.hist(x, bins=50)
```



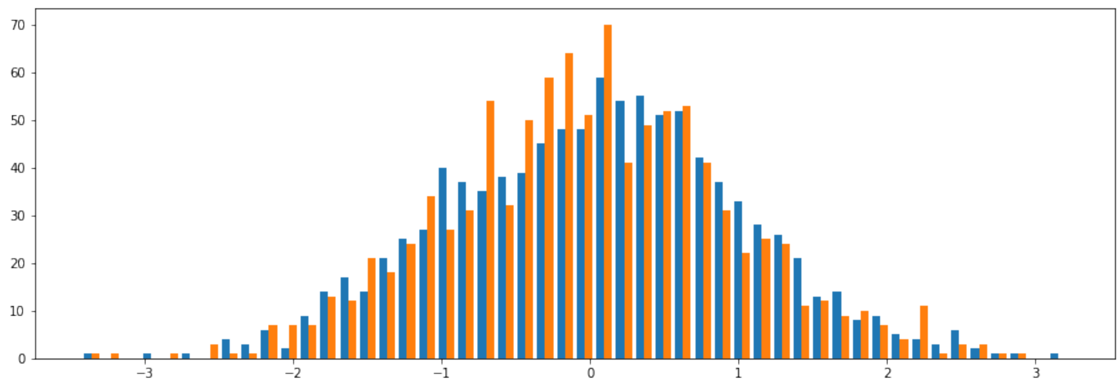
[164]:
```
fig, ax = plt.subplots()
H = ax.hist(x, bins=50, histtype='stepfilled', log=True, color='r')
```



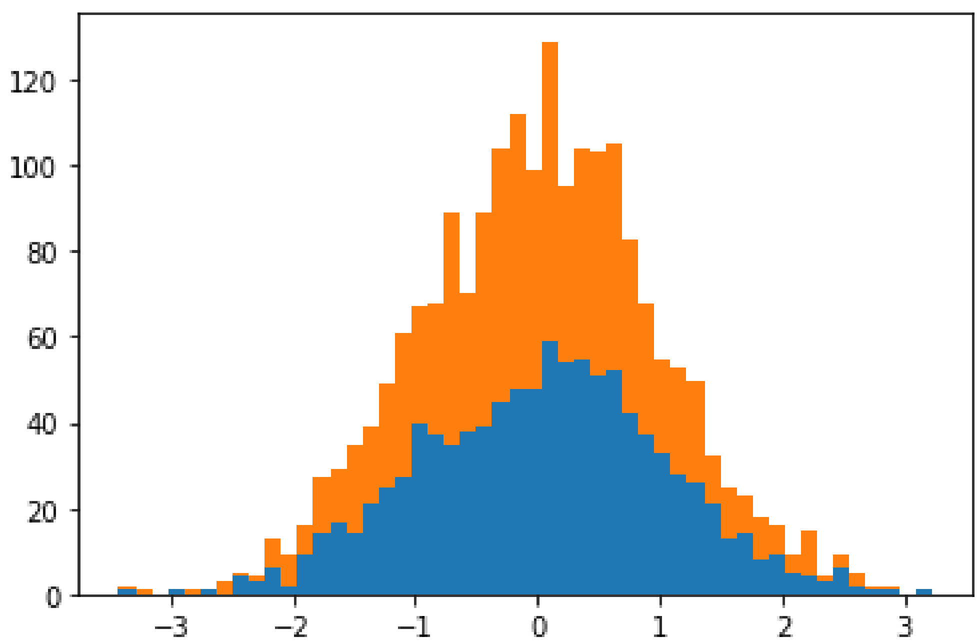
[168]:
```
x2 = np.random.normal(size=1000)
fig, ax = plt.subplots()
H = ax.hist((x, x2), bins=50, alpha=.3, histtype='stepfilled')
```



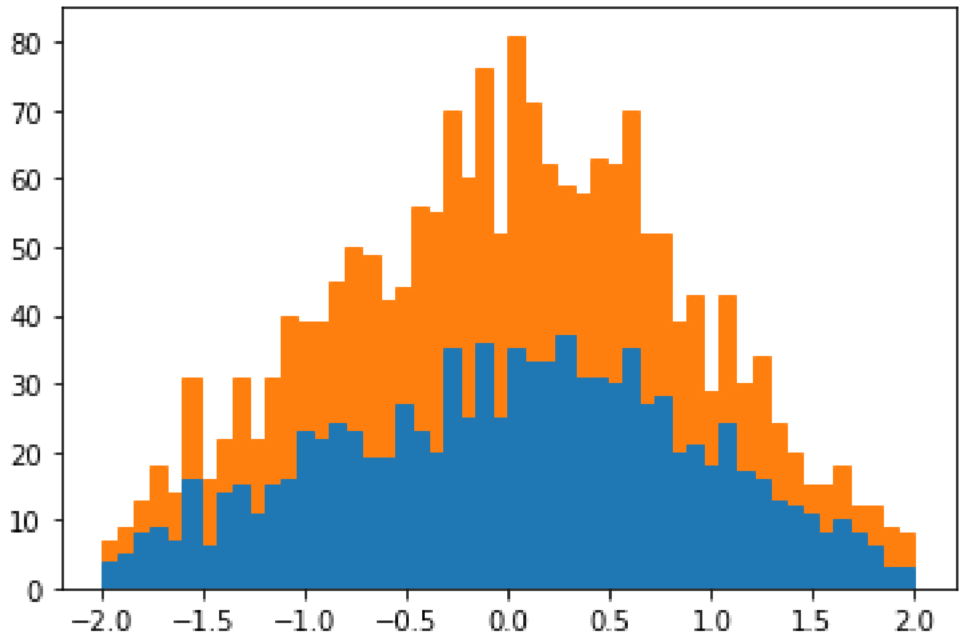
[169]:
```
fig, ax = plt.subplots(figsize=(15,5))
H = ax.hist((x, x2), bins=50, histtype='bar')
```



[170]:
```
fig, ax = plt.subplots()
H = ax.hist((x, x2), bins=50, histtype='bar', stacked=True)
```

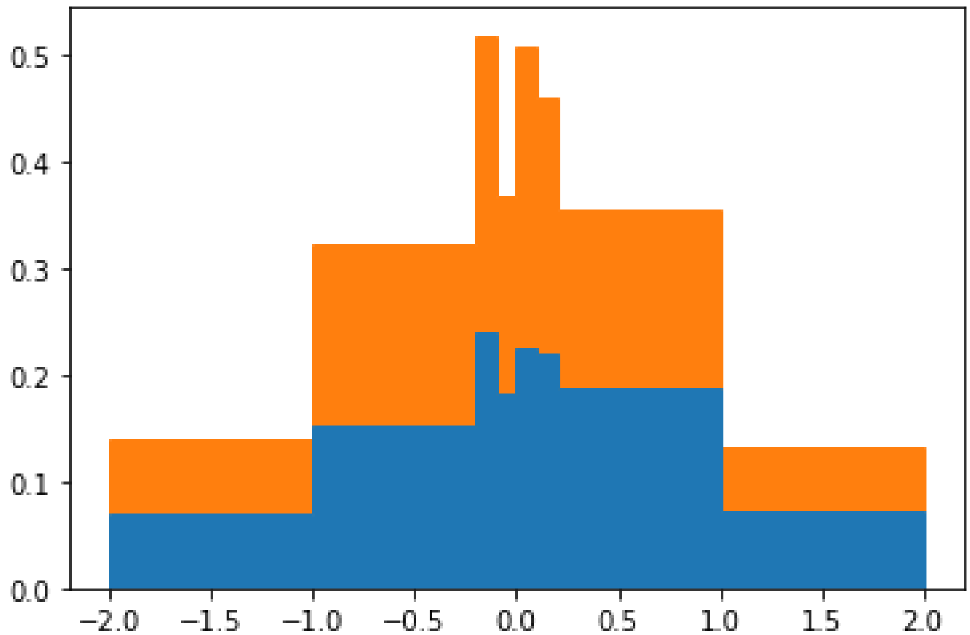


[171]:
```
fig, ax = plt.subplots()
H = ax.hist((x, x2), bins=50, range=(-2, 2), histtype='step', stacked=True,␣
 ↪fill=True)
```



[172]:
```
fig, ax = plt.subplots()
H = ax.hist((x, x2), bins=(-2, -1, -0.2, -0.1, 0., 0.1, 0.2, 1, 2), range=(-2,␣
 ↪2),
            histtype='step', stacked=True, fill=True, normed=True)
```

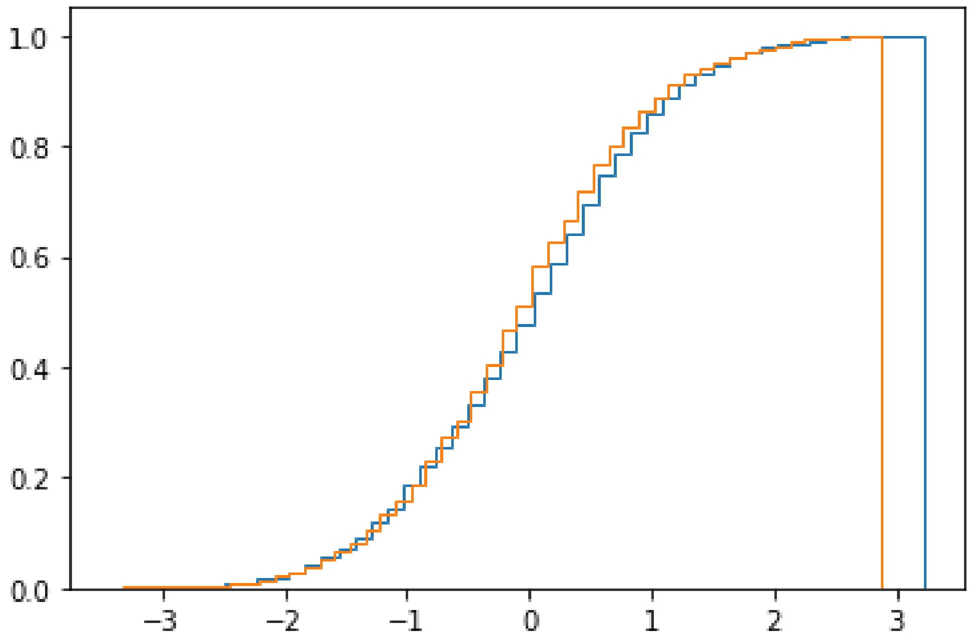
```
/Users/christophemorisset/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:3: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1.
Use 'density' instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
```



[173]:

```
fig, ax = plt.subplots()
H = ax.hist(x, bins=50, histtype='step', cumulative=True, normed=True)
H2 = ax.hist(x2, bins=50, histtype='step', cumulative=True, normed=True)
```

```
/Users/christophemorisset/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:2: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1.
Use 'density' instead.

/Users/christophemorisset/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:3: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1.
Use 'density' instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
```



boxplots

[174]:
```
help(ax.boxplot)
```

```
Help on method boxplot in module matplotlib.axes._axes:

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

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

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

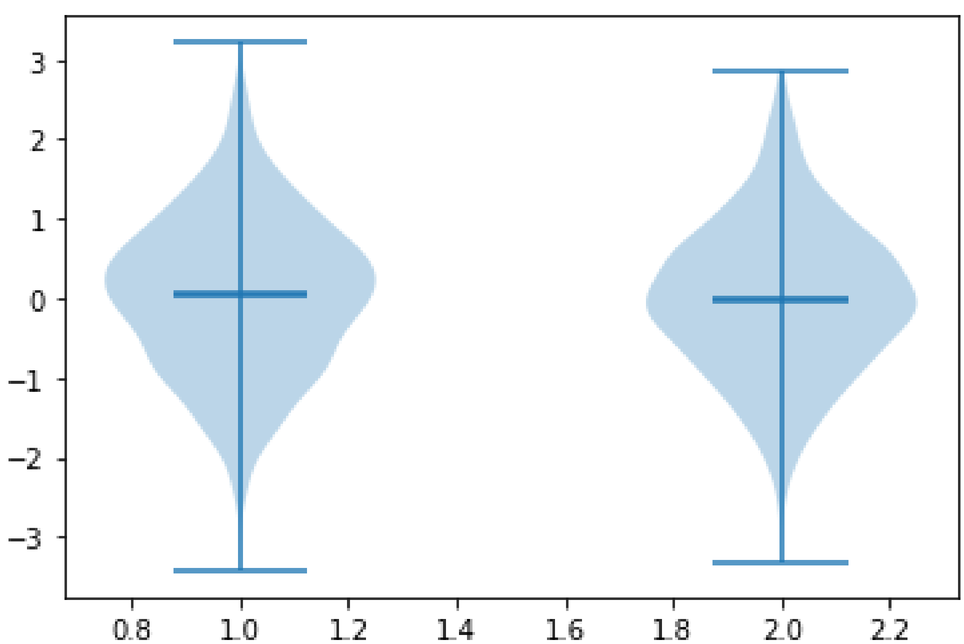
```
notch : bool, optional (False)
    If `True`, will produce a notched box plot. Otherwise, a
    rectangular boxplot is produced. The notches represent the
    confidence interval (CI) around the median. See the entry
    for the ``bootstrap`` parameter for information regarding
    how the locations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

    .. note::

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

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

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

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

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

```
    the number of times to bootstrap the median to determine its
    95% confidence intervals. Values between 1000 and 10000 are
    recommended.

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

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

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

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

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

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

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

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

meanline : bool, optional (False)
```

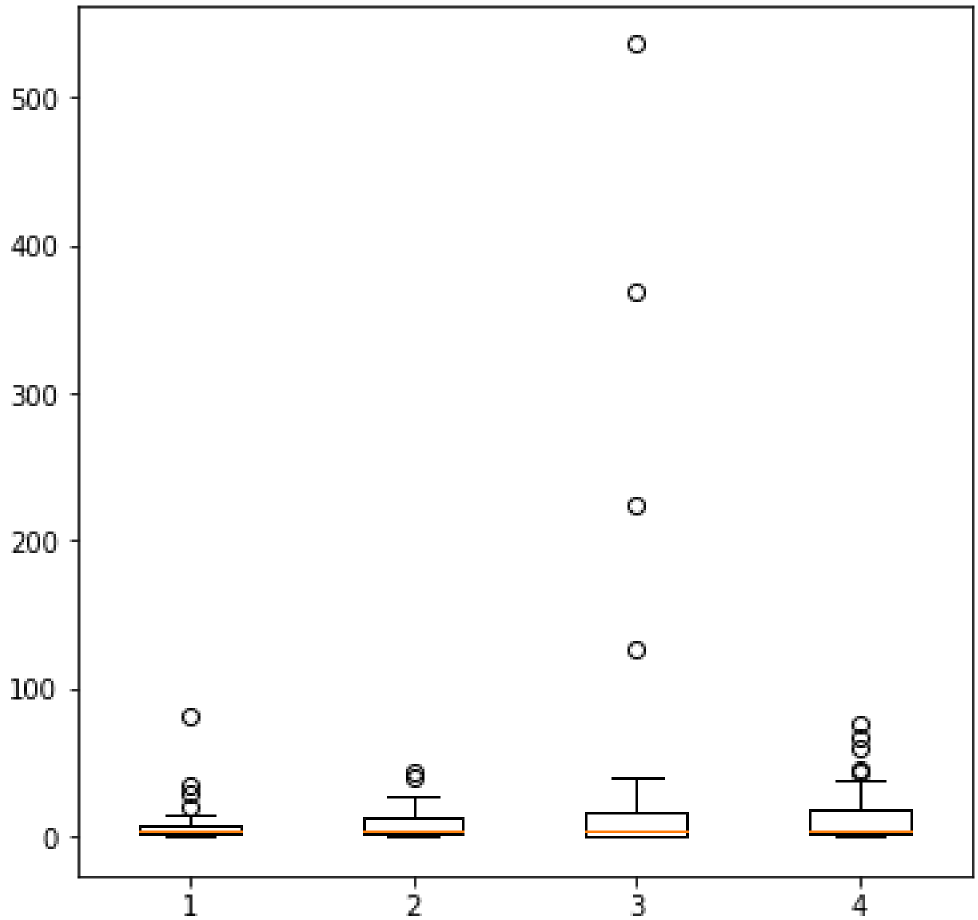
```
    If `True` (and ``showmeans`` is `True`), will try to render
    the mean as a line spanning the full width of the box
    according to ``meanprops`` (see below). Not recommended if
    ``shownotches`` is also True. Otherwise, means will be shown
    as points.

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

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

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

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

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

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

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

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

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

Notes
-----



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

    * All positional and all keyword arguments.

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

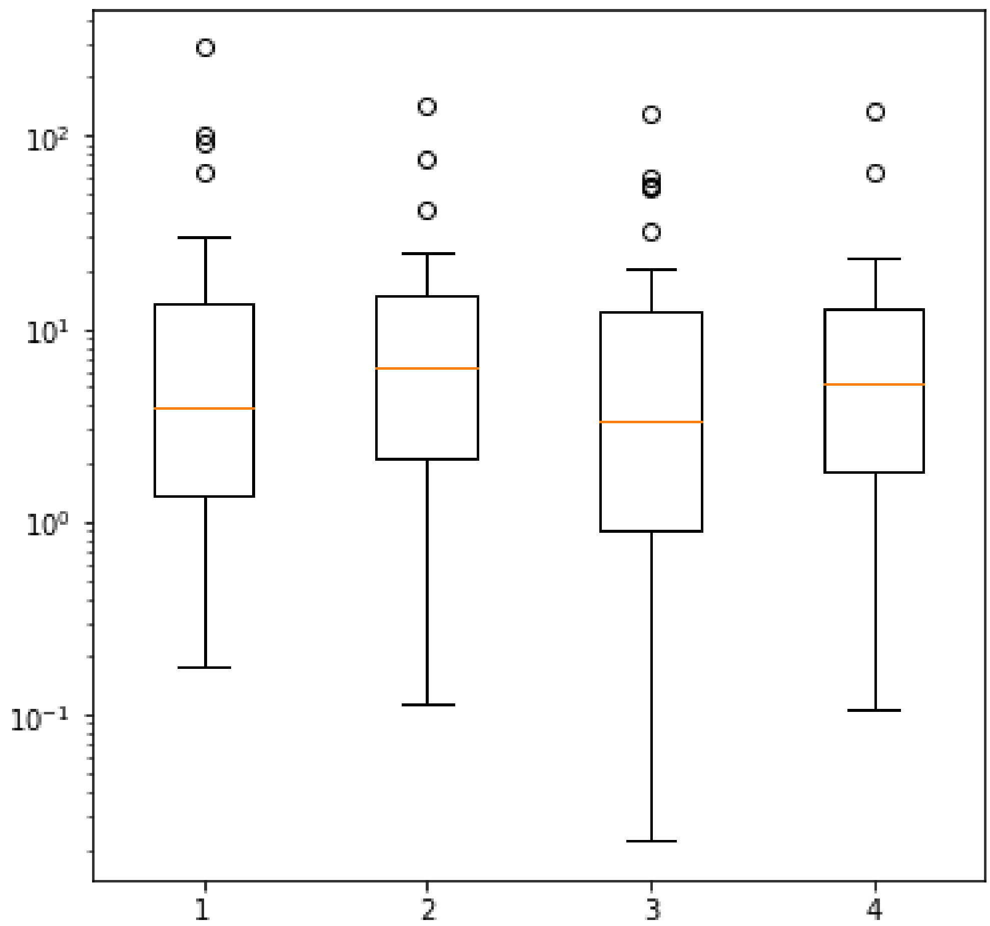
[175]:
```python
fig, ax = plt.subplots()
bp = ax.violinplot((x, x2), showmeans=True,showmedians=True)
```



[176]:
```
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)

fig, ax = plt.subplots(figsize=(6,6))
bp = ax.boxplot(data) # Nothing to see !
```



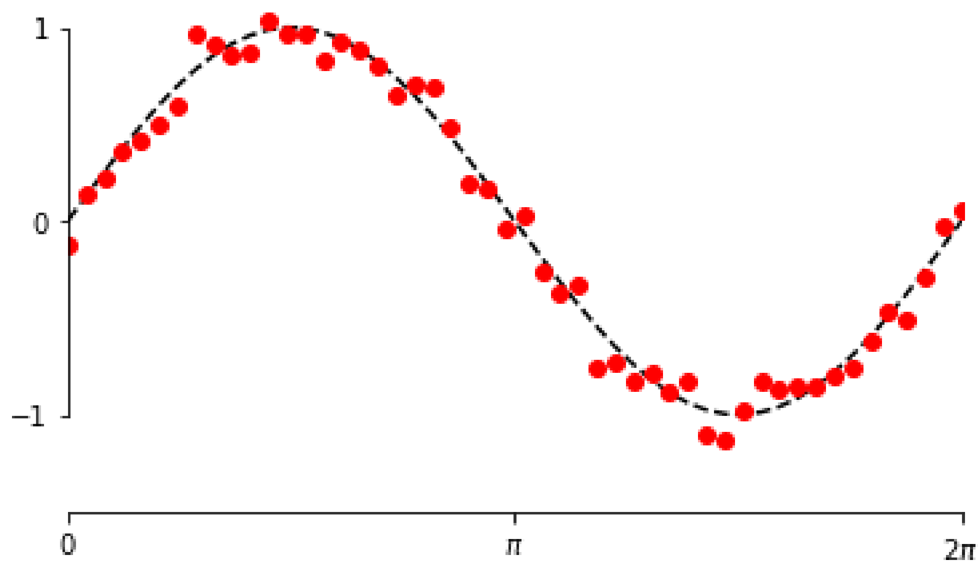
[177]:
```
data = np.random.lognormal(size=(37, 4), mean=1.5, sigma=1.75)

fig, ax = plt.subplots(figsize=(6,6))
bp = ax.boxplot(data)
ax.set_yscale('log')
```



Ticks, axes and spines

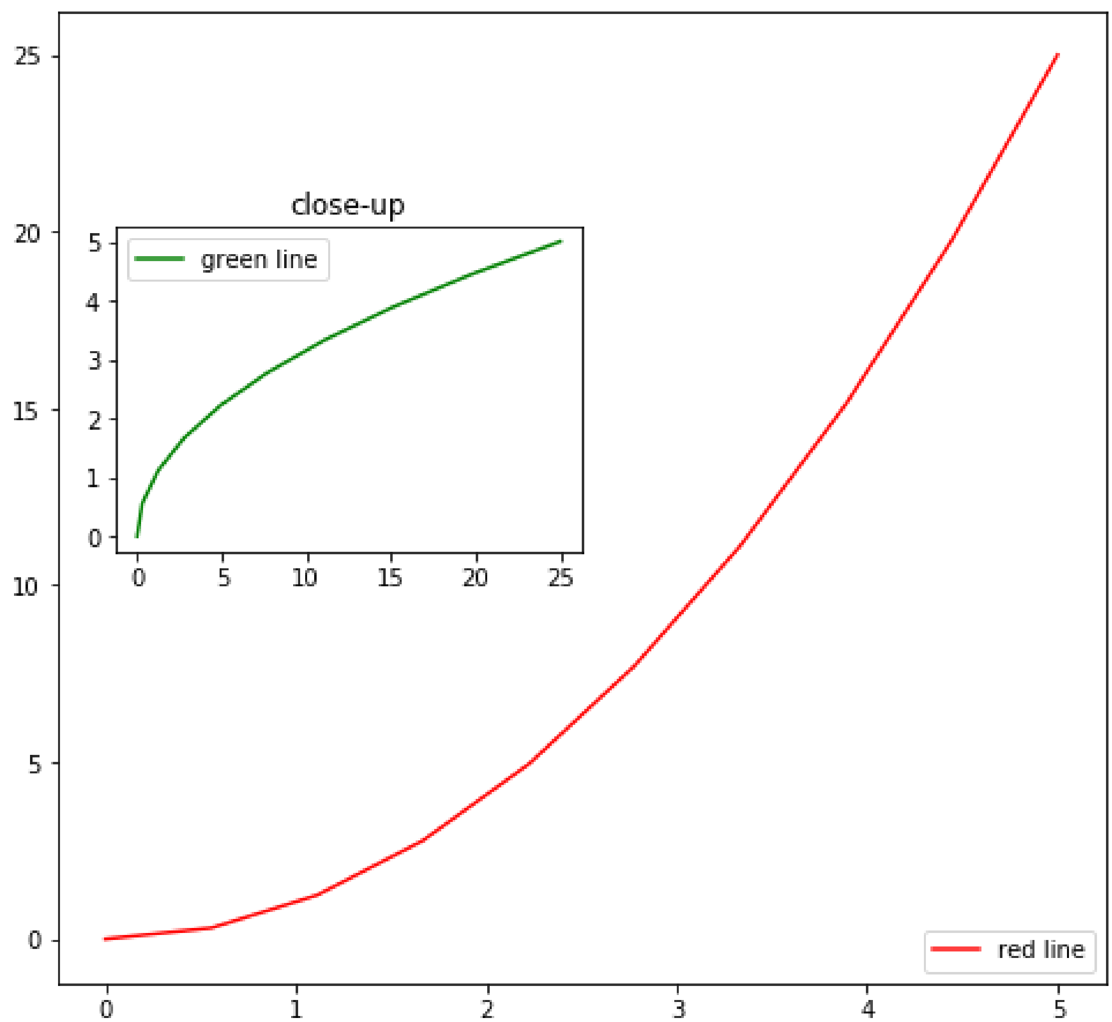
[178]:
```
x = np.linspace(0, 2*np.pi, 50)
y = np.sin(x)
y2 = y + 0.1 * np.random.normal(size=x.shape) # add noise to the data

fig, ax = plt.subplots()
ax.plot(x, y, 'k--')
ax.plot(x, y2, 'ro')

# set ticks and tick labels
ax.set_xlim((0, 2*np.pi))
ax.set_xticks([0, np.pi, 2*np.pi])
ax.set_xticklabels(['0', '$\pi$','2$\pi$'])
ax.set_ylim((-1.5, 1.5))
ax.set_yticks([-1, 0, 1])

# Only draw spine between the y-ticks
```

```
ax.spines['left'].set_bounds(-1, 1)
# Hide the right and top spines
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
# Only show ticks on the left and bottom spines
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
```



A plot inside a plot

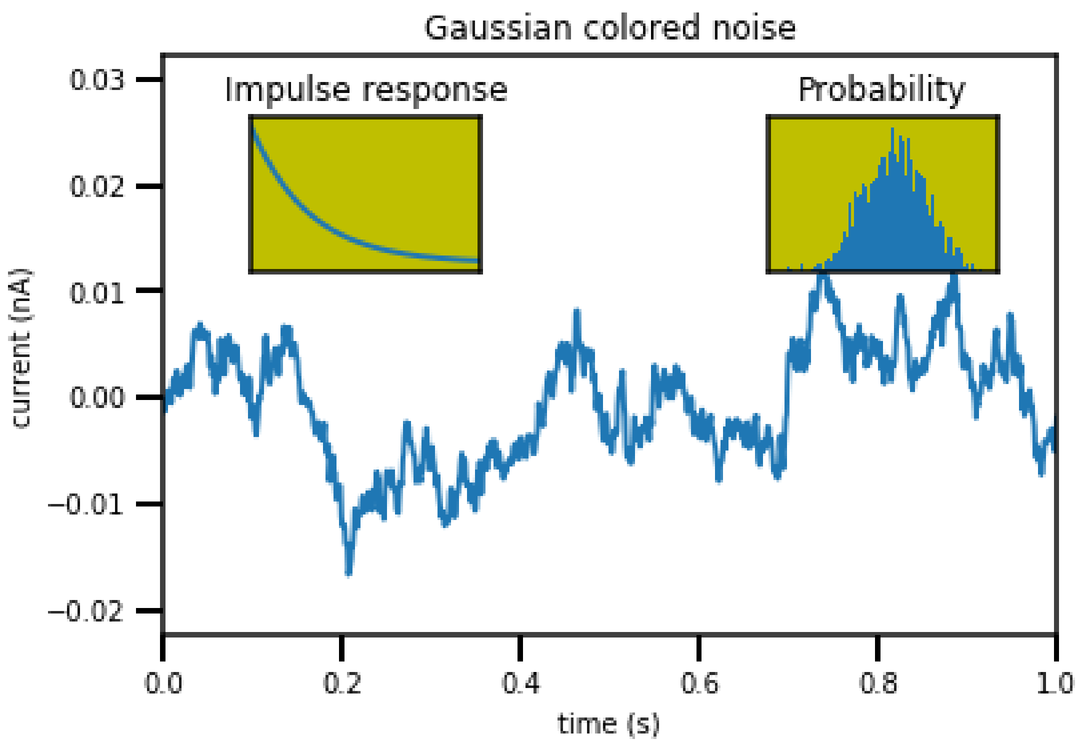
[179]:
```
x = np.linspace(0, 5, 10)
y = x ** 2

fig = plt.figure(figsize=(7,6.5))

ax1 = fig.add_axes([0.1, 0.1, 0.9, 0.9]) # main axes
ax2 = fig.add_axes([0.15, 0.5, 0.4, 0.3]) # inset axes

# main figure
ax1.plot(x, y, 'r', label='red line')
ax1.legend(loc=4)
# insert
ax2.plot(y, x, 'g', label = 'green line')
ax2.set_title('close-up')
ax2.legend(loc='best');
```



[221]:
```
# The classical way

# create some data to use for the plot
dt = 0.001
t = np.arange(0.0, 10.0, dt)
r = np.exp(-t[:1000]/0.05)  # impulse response
x = np.random.randn(len(t))
s = np.convolve(x,r)[:len(x)]*dt  # colored noise

# the main axes is subplot(111) by default
plt.plot(t, s)
plt.axis([0, 1, 1.1*np.amin(s), 2*np.amax(s) ])
plt.xlabel('time (s)')
plt.ylabel('current (nA)')
plt.title('Gaussian colored noise')
```

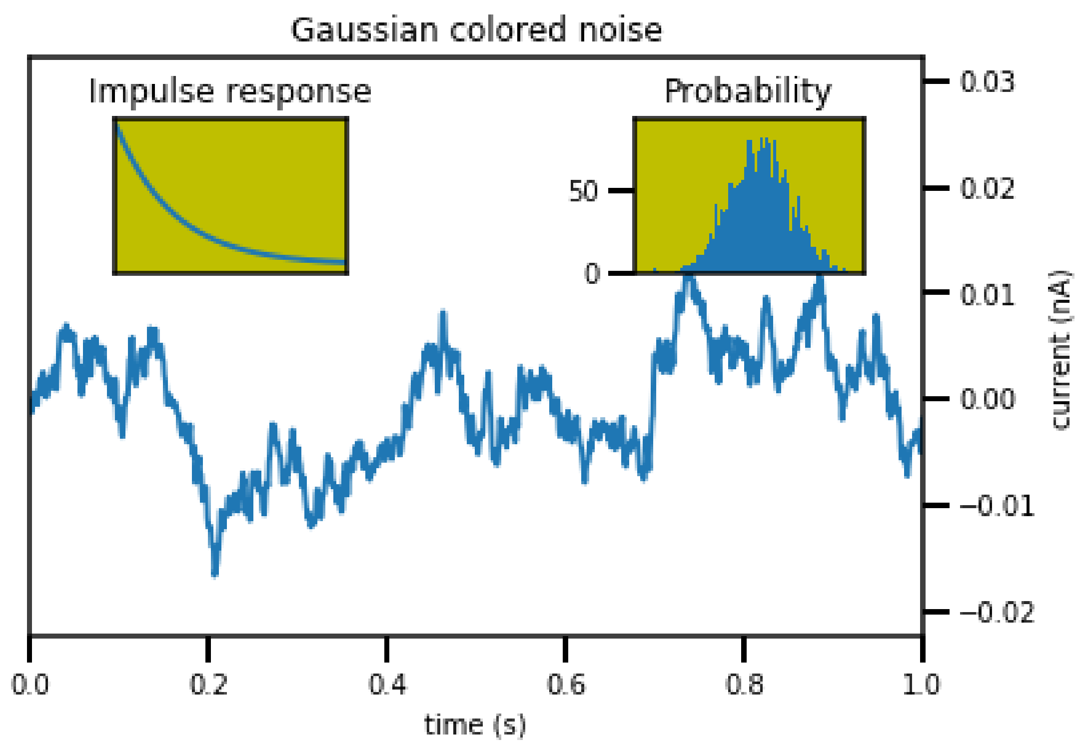
```
# this is an inset axes over the main axes
a = plt.axes([.65, .6, .2, .2], facecolor='y')
n, bins, patches = plt.hist(s, 400, normed=1)
plt.title('Probability')
plt.setp(a, xticks=[], yticks=[])

# this is another inset axes over the main axes
b = plt.axes([0.2, 0.6, .2, .2], facecolor='y')
plt.plot(t[:len(r)], r)
plt.title('Impulse response')
plt.setp(b, xlim=(0,.2), xticks=[], yticks=[]);
```

```
/Users/christophemorisset/anaconda3/lib/python3.7/site-
packages/ipykernel_launcher.py:19: MatplotlibDeprecationWarning:
The 'normed' kwarg was deprecated in Matplotlib 2.1 and will be removed in 3.1.
Use 'density' instead.
```



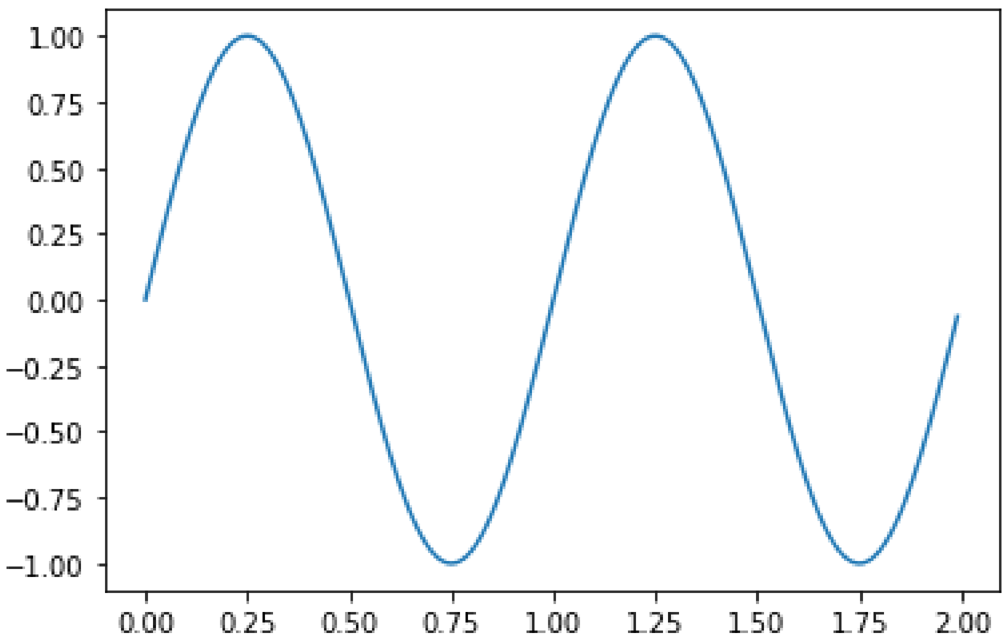
[228]:
```
# The Object oriented way

# the main axes is subplot(111) by default
fig, ax = plt.subplots()
ax.plot(t, s)
```

```python
ax.axis([0, 1, 1.1*np.amin(s), 2*np.amax(s) ])
# The previous command is equivalent to:
#ax.set_xlim((0., 1))
#ax.set_ylim((1.1*np.amin(s), 2*np.amax(s)))
ax.set_xlabel('time (s)')
ax.set_ylabel('current (nA)')
ax.set_title('Gaussian colored noise')
ax.yaxis.tick_right()
ax.yaxis.set_label_position("right")

# this is an inset axes over the main axes
ax2 = plt.axes([.65, .6, .2, .2], facecolor='y')
n, bins, patches = ax2.hist(s, 400, density=True)
ax2.set_title('Probability')
ax2.xaxis.set_ticks([])
#ax2.yaxis.set_ticks([])

# this is another inset axes over the main axes
ax3 = plt.axes([0.2, 0.6, .2, .2], facecolor='y')
ax3.plot(t[:len(r)], r)
ax3.set_title('Impulse response')
ax3.set_xlim((0., .2))
ax3.xaxis.set_ticks([])
ax3.yaxis.set_ticks([]);
```



Play with all the objects of a plot

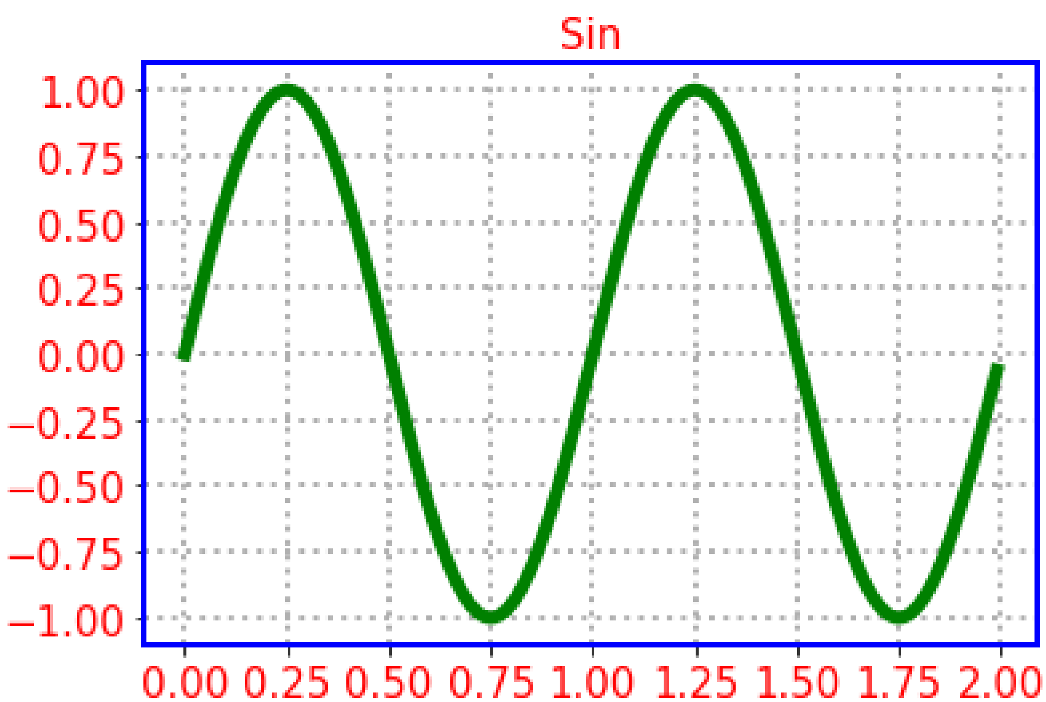
[182]:
```
# Define some data
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots()

# Plot the data and keep the data-line into an object
ax.plot(t, s);
```



[185]:
```
# Define some data
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots()

# Plot the data and keep the data-line into an object
datalines = ax.plot(t, s)



# Plot grids on the figure
ax.grid(True)
tit = ax.set_title('Sin')

# Put all the lines and labels into lists of objects
```

```
ticklines = iter(ax.spines.values())
gridlines = ax.get_xgridlines()
gridlines.extend( ax.get_ygridlines() )
labels = ax.get_xticklabels()
labels.extend( ax.get_yticklabels() )
labels.append(tit)# Loop on the lists of lines to change properties
print(labels)

for line in ticklines:
    line.set_linewidth(2)
    line.set_color('blue')

for line in datalines:
    line.set_linewidth(5)
    line.set_color('green')

for line in gridlines:
    line.set_linestyle(':')
    line.set_linewidth(2)

# loop on the labels to change properties
for label in labels:
    label.set_color('r')
    label.set_fontsize(15)
```

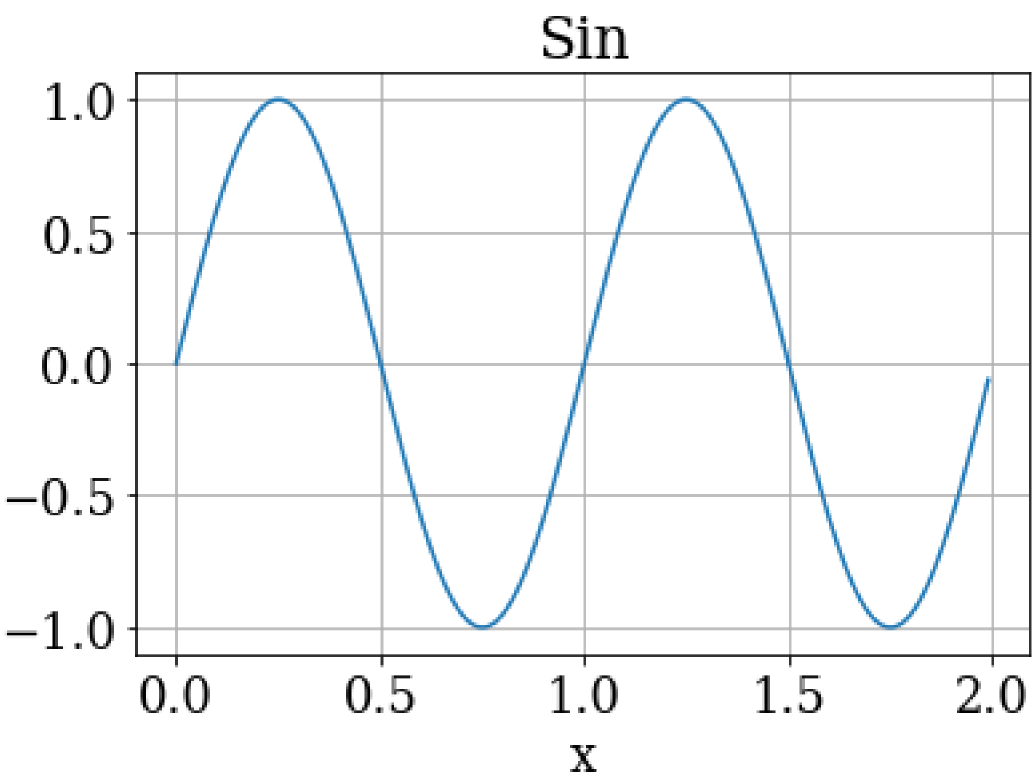
```
<a list of 23 Text xticklabel objects>
```



Changing font etc for all the plots:

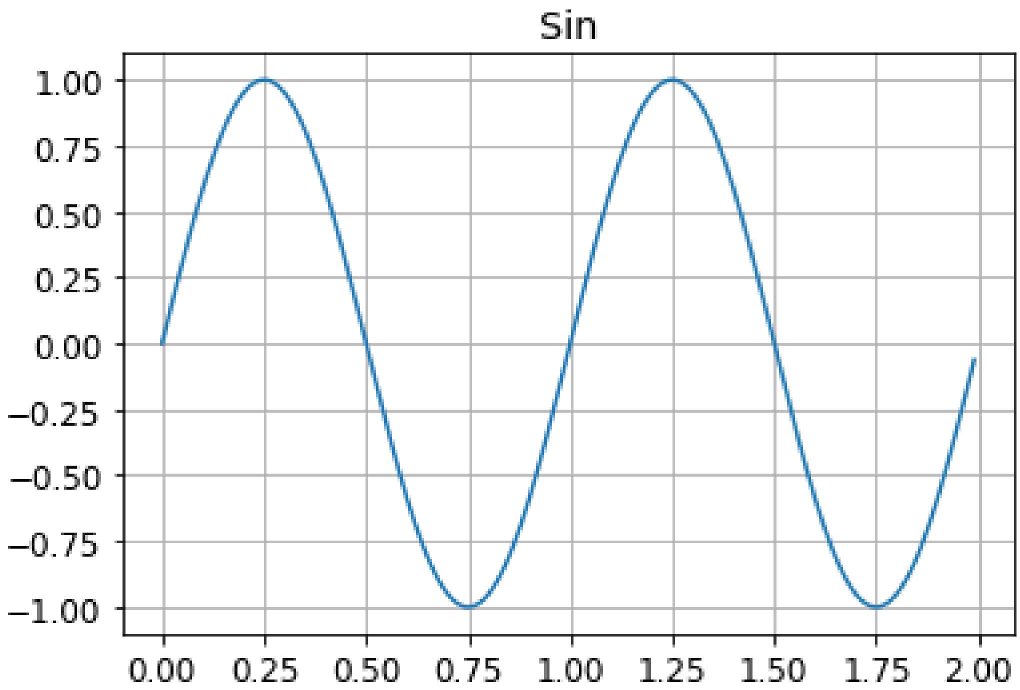
[194]:
```
import matplotlib
matplotlib.rcParams.update({'font.size': 18, 'font.family': 'serif'})
```

[195]:
```
# Define some data
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots()

# Plot the data and keep the data-line into an object
datalines = ax.plot(t, s)
# Plot grids on the figure
ax.grid(True)
ax.set_xlabel('x')
tit = ax.set_title('Sin')
```



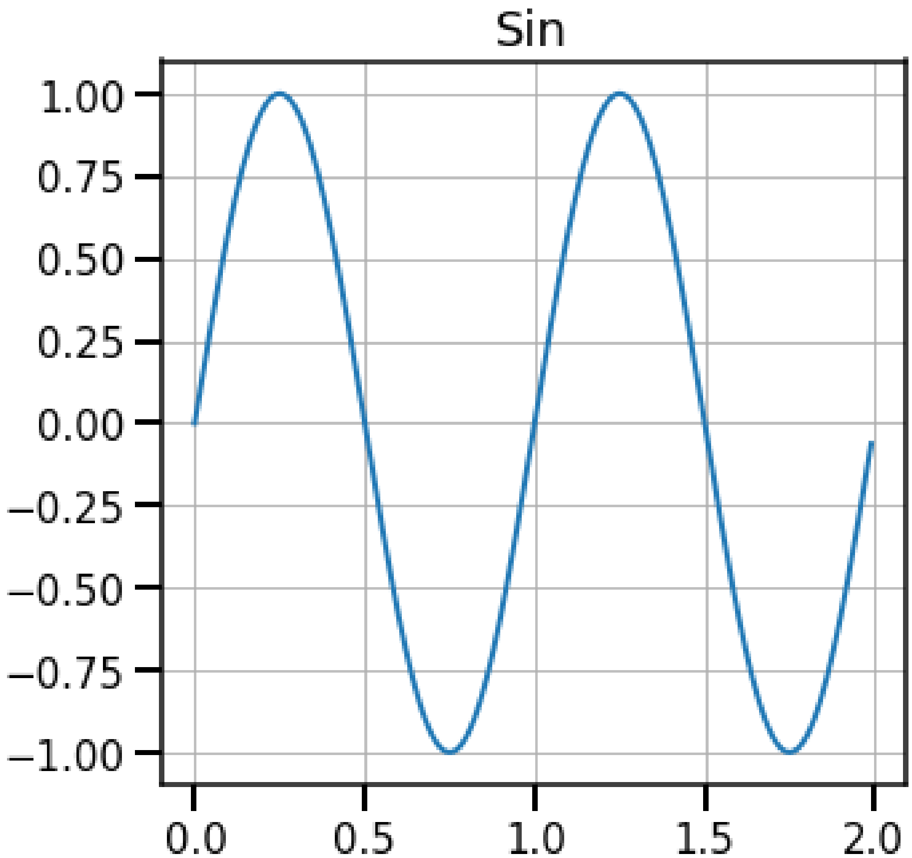
[196]:
```
# Back to default values
matplotlib.rcParams.update({'font.size': 12, 'font.family': 'sans'})
```

[197]:
```
# Define some data
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots()

# Plot the data and keep the data-line into an object
datalines = ax.plot(t, s)
# Plot grids on the figure
ax.grid(True)
tit = ax.set_title('Sin')
```



[198]:
```
matplotlib.rc('axes', linewidth=1.5)
matplotlib.rc('lines', linewidth=2)
matplotlib.rc('font', size=15)
matplotlib.rc('xtick.major', width=2, size=10)
matplotlib.rc('xtick.minor', width=2, size=5)
matplotlib.rc('ytick.major', width=2, size=10)
matplotlib.rc('ytick.minor', width=2, size=5)

# Define some data
t = np.arange(0.0, 2.0, 0.01)
s = np.sin(2*np.pi*t)
fig, ax = plt.subplots(figsize=(5,5))
```

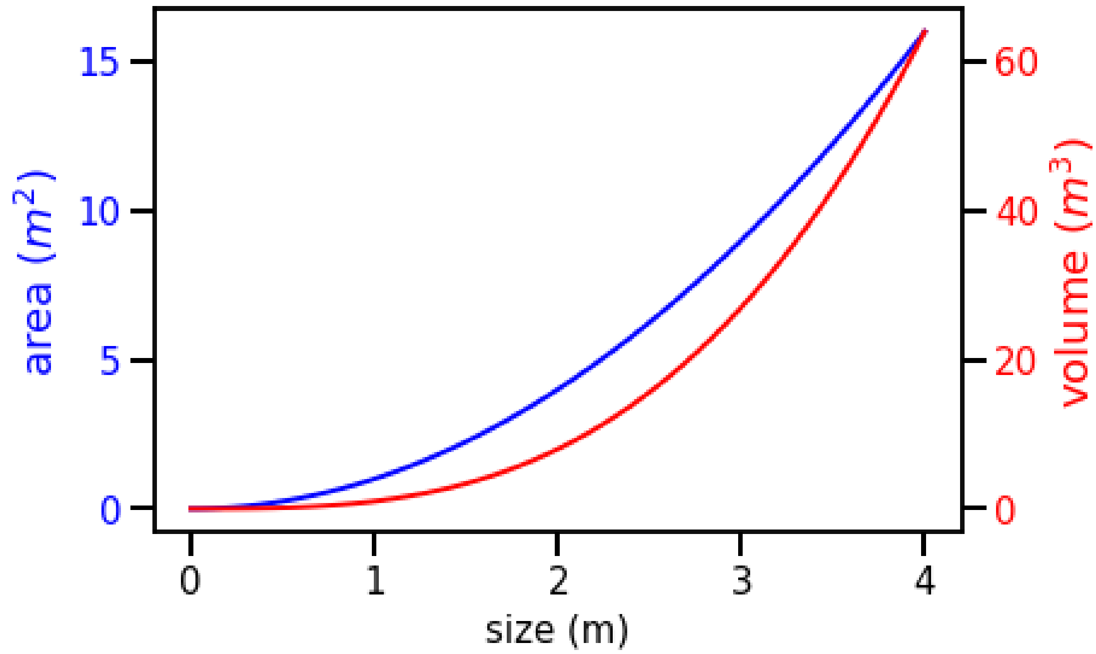
```python
# Plot the data and keep the data-line into an object
datalines = ax.plot(t, s)
# Plot grids on the figure
ax.grid(True)
tit = ax.set_title('Sin')
```



Twin axes

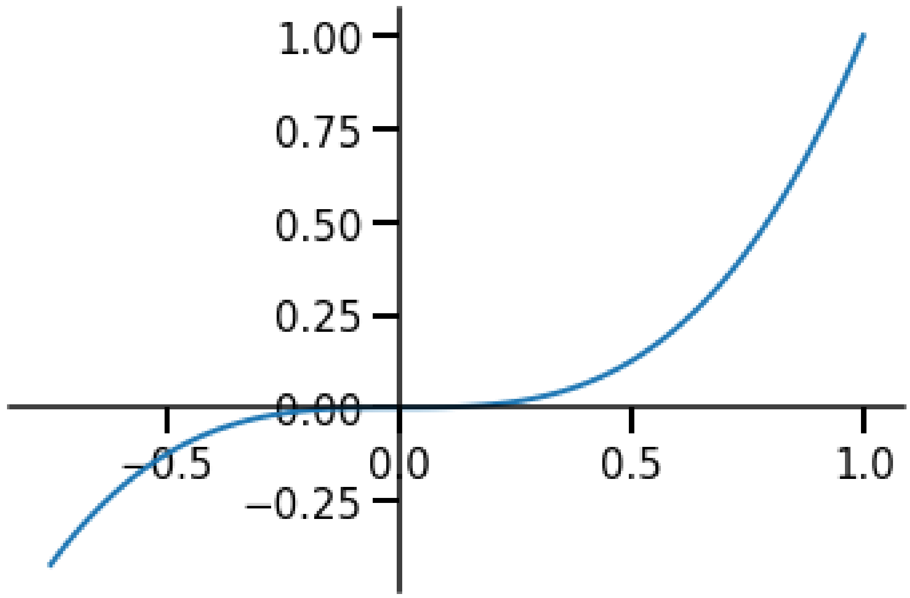
[199]:
```python
x = np.linspace(0, 4, 100)
fig, ax1 = plt.subplots()

ax1.plot(x, x**2, lw=2, color="blue")
ax1.set_xlabel('size (m)')
ax1.set_ylabel(r"area $(m^2)$", fontsize=18, color="blue")
for label in ax1.get_yticklabels():
    label.set_color("blue")

ax2 = ax1.twinx()
ax2.plot(x, x**3, lw=2, color="red")
ax2.set_ylabel(r"volume $(m^3)$", fontsize=18, color="red")
```

```
for label in ax2.get_yticklabels():
    label.set_color("red")
```



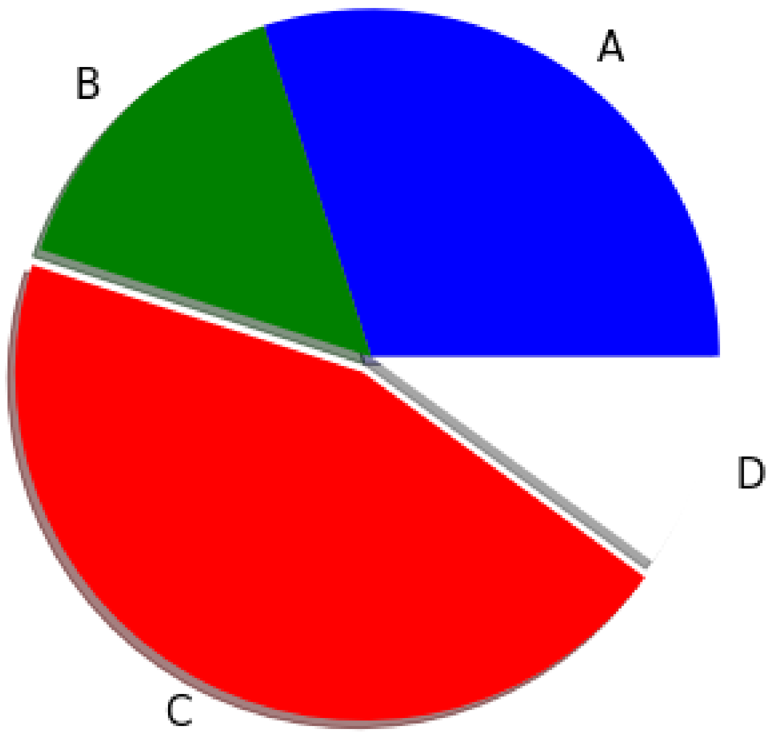
Axis crossing at 0

[200]:
```
fig, ax = plt.subplots()

ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

ax.xaxis.set_ticks_position('bottom')
ax.spines['bottom'].set_position(('data',0)) # set position of x spine to x=0

ax.yaxis.set_ticks_position('left')
ax.spines['left'].set_position(('data',0))   # set position of y spine to y=0

xx = np.linspace(-0.75, 1., 100)
ax.plot(xx, xx**3);
```



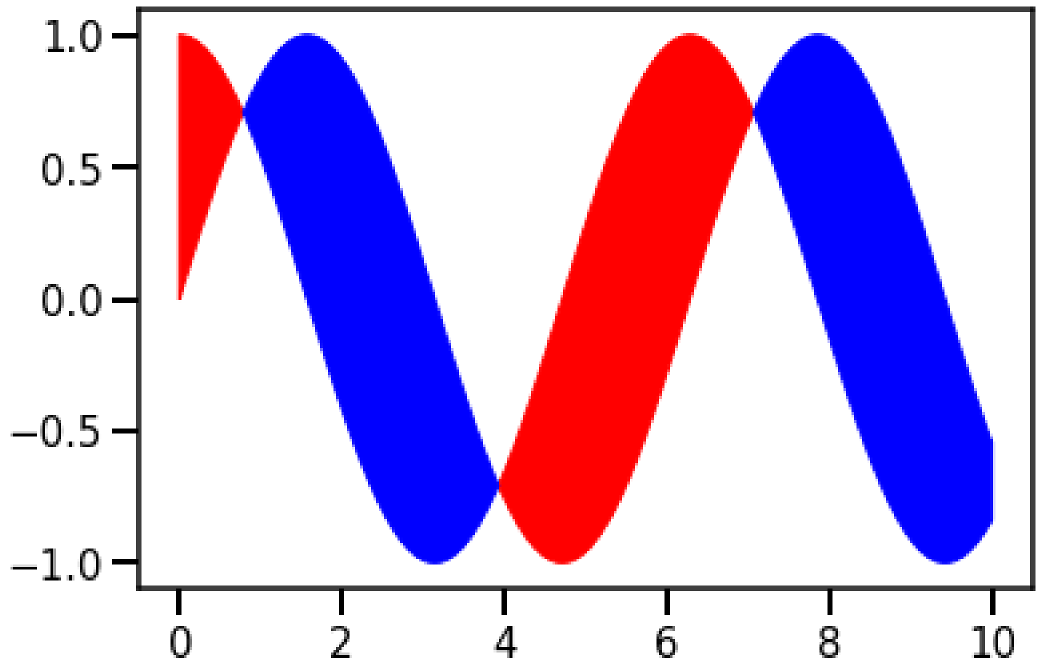
Pie plots

[201]:
```
fracs = [30, 15, 45, 10]
colors = ['b', 'g', 'r', 'w']

fig, ax = plt.subplots(figsize=(6, 6))  # make the plot square
pie = ax.pie(fracs, colors=colors, explode=(0, 0, 0.05, 0), shadow=True,
             labels=['A', 'B', 'C', 'D'])
```



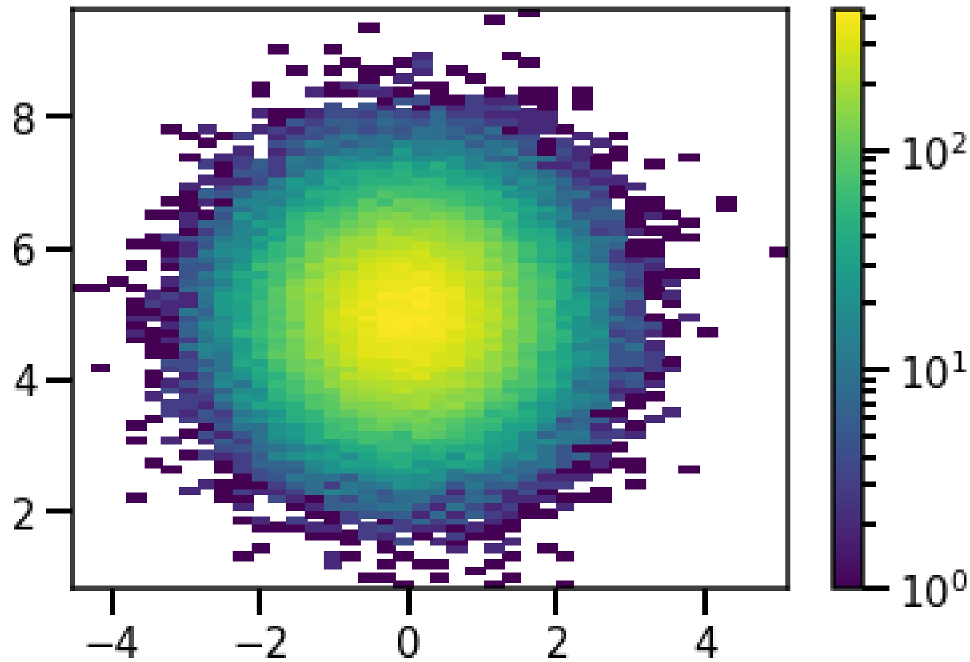
Filled regions

[202]:
```
x = np.linspace(0, 10, 1000)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax = plt.subplots()
ax.fill_between(x, y1, y2, where=(y1 < y2), color='red')
ax.fill_between(x, y1, y2, where=(y1 > y2), color='blue');
```



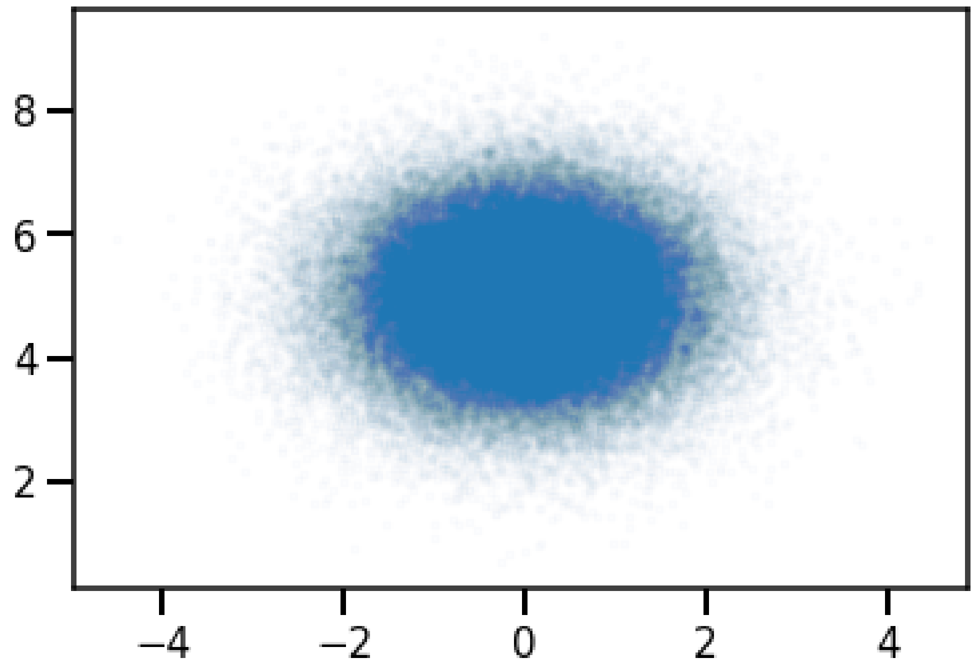
2D-histograms and hexagon plots

[203]:
```
from matplotlib.colors import LogNorm

#normal distribution center at x=0 and y=5
x = np.random.randn(100000)
y = np.random.randn(100000)+5

fig, ax = plt.subplots()
counts, xedges, yedges, Image = ax.hist2d(x, y, bins=(40, 80), norm=LogNorm())
cb = fig.colorbar(Image)
```



[204]:
```
from matplotlib.colors import LogNorm

#normal distribution center at x=0 and y=5
x = np.random.randn(100000)
y = np.random.randn(100000)+5

fig, ax = plt.subplots()
ax.scatter(x,y, alpha=0.01, marker='.')
```

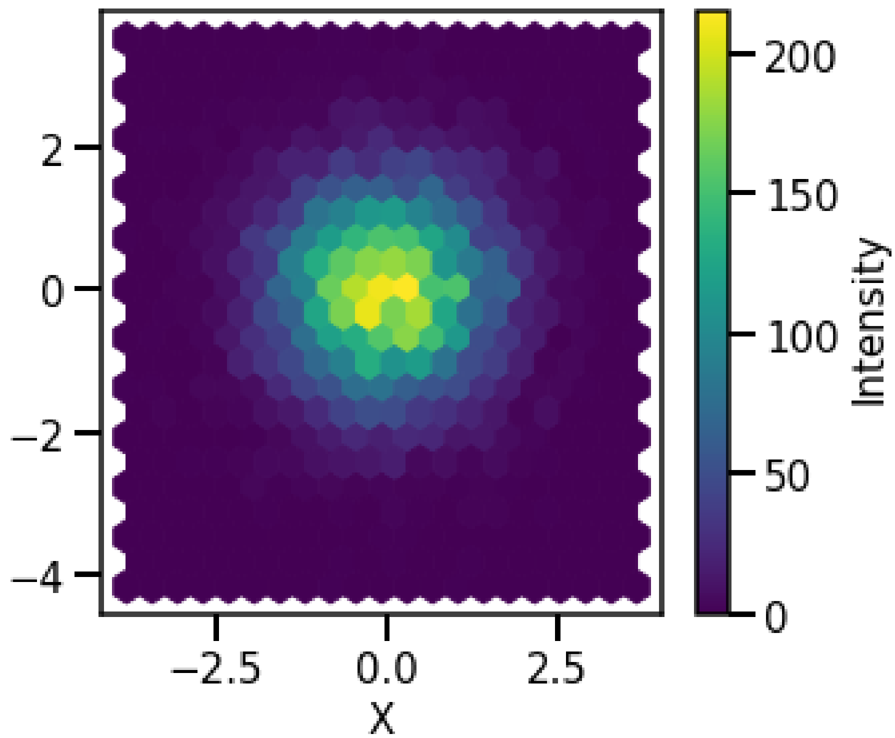
[204]: <matplotlib.collections.PathCollection at 0x7f9bb78e7850>

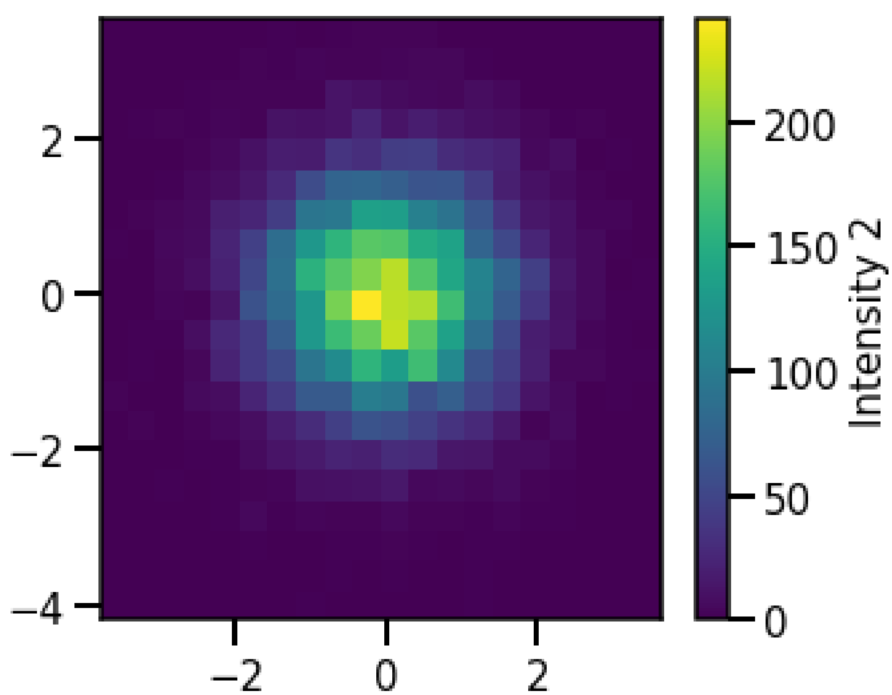


[205]:

```
x, y = np.random.normal(size=(2, 10000))

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(5, 8))

im = ax1.hexbin(x, y, gridsize=20)
ax1.set_xlabel('X')
cb = fig.colorbar(im, ax=ax1)
cb.set_label('Intensity')

H = ax2.hist2d(x, y, bins=20)
print(H[0].shape)
cb = fig.colorbar(H[3], ax=ax2)
cb.set_label('Intensity 2')
fig.tight_layout()
```

```
(20, 20)
```

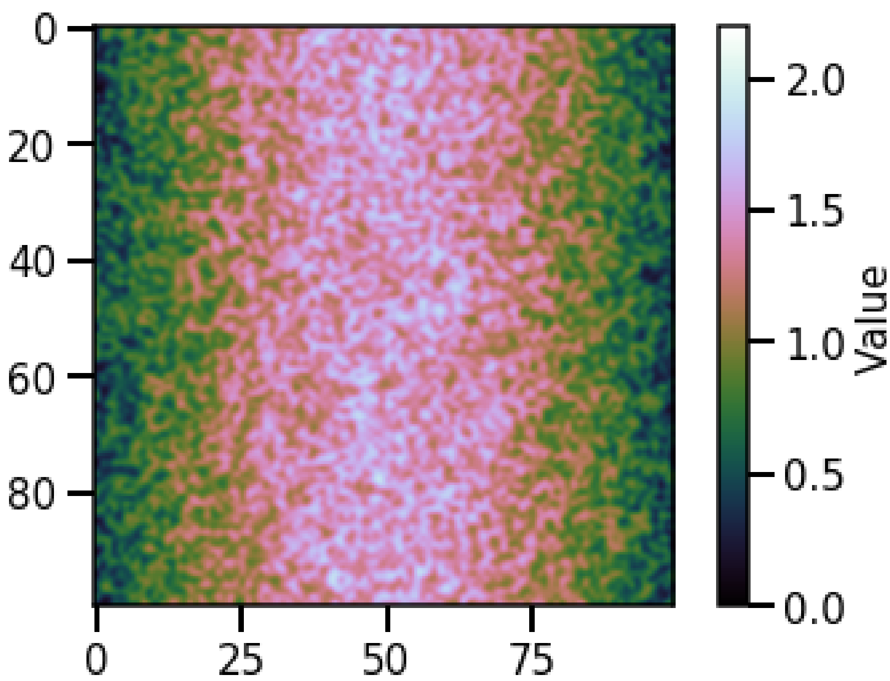




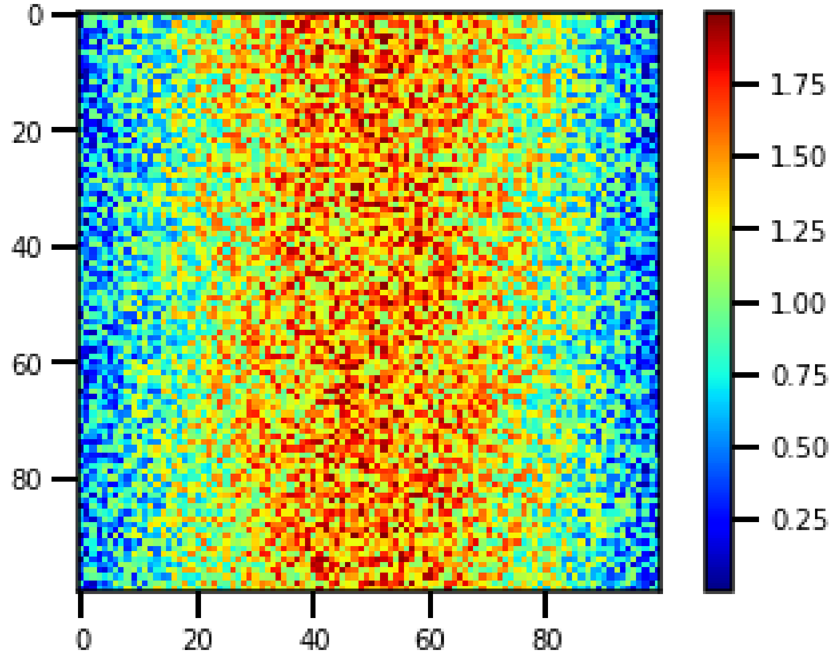
2D data sets and Images

[206]:
```
N = 100
I = np.random.random((N, N))
I += np.sin(np.linspace(0, np.pi, N))
print(I.shape)
fig, ax = plt.subplots()
im = ax.imshow(I, cmap=plt.cm.cubehelix, vmin=0, vmax=2.2,␣
↪interpolation='bicubic') # draw the image
cb = fig.colorbar(im) # put the colorbar
cb.set_label('Value')
```

```
(100, 100)
```



[219]:
```
fig, ax = plt.subplots()
im = ax.imshow(I, cmap=plt.cm.jet, interpolation='none') # draw the image, no␣
↪interpolations, raw data


ax = plt.gca()
fig = plt.gcf()
cb = fig.colorbar(im, ax=ax) # put the colorbar
```



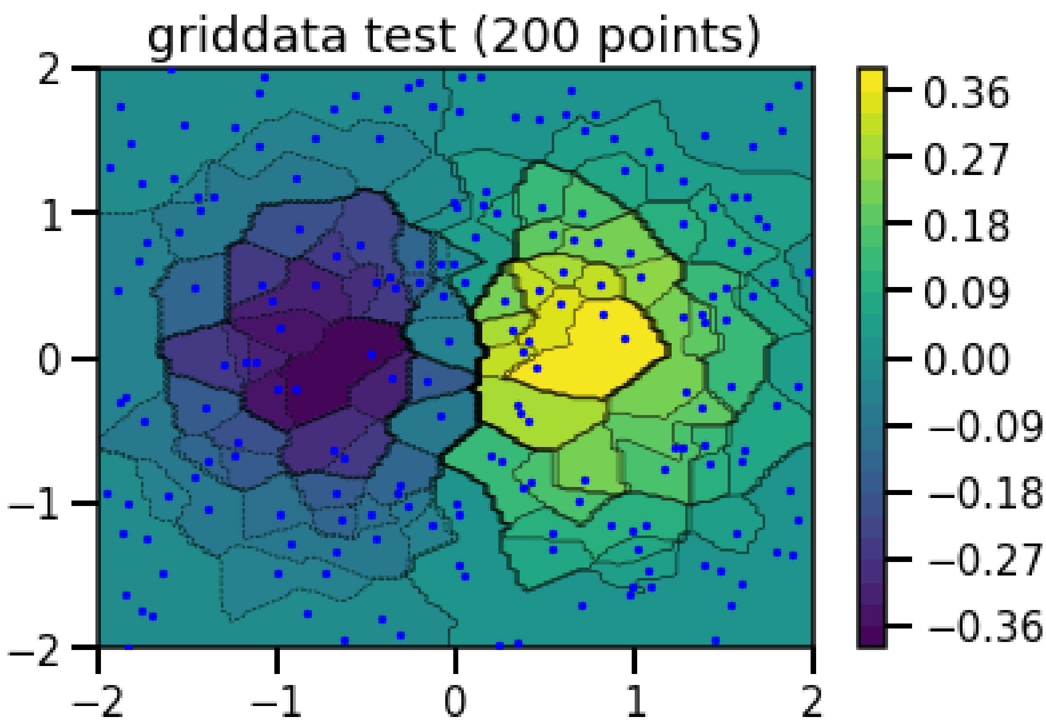

```
[ ]: help(plt.imshow)
```

Contour

```
[208]: from scipy.interpolate import griddata

# make up data.
#npts = int(raw_input('enter # of random points to plot:'))
npts = 200
points = np.random.rand(npts, 2) * 4 - 2
x = points[:,0]
y = points[:,1]
#print(points)
def func(x,y):
    #return x*(1-x)*np.cos(4*np.pi*x) * np.sin(4*np.pi*y**2)**2
    return x * np.exp(-x**2 - y**2)
print(points.shape)
values = func(points[:,0], points[:,1])
# define grid.
xi, yi = np.mgrid[-2.1:2.1:100j, -2.1:2.1:200j]
# grid the data.
zi = griddata(points, values, (xi, yi), method='nearest') # Linear does not␣
↪work???

# contour the gridded data, plotting dots at the nonuniform data points.
```

```
fig, ax = plt.subplots()
ax.contour(xi, yi, zi, 25, linewidths=0.5, colors='k')
CF = ax.contourf(xi, yi, zi, 25,
                 vmax=abs(zi).max(), vmin=-abs(zi).max())
cb = fig.colorbar(CF)  # draw colorbar
# plot data points.
ax.scatter(x, y, marker='o', c='b', s=5, zorder=10)
ax.set_xlim(-2, 2)
ax.set_ylim(-2, 2)
ax.set_title('griddata test (%d points)' % npts);
```

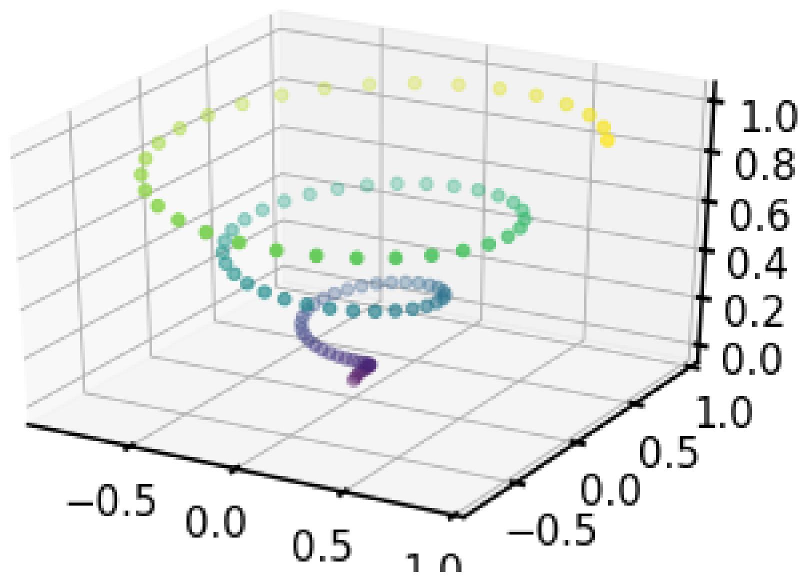
```
(200, 2)
```



3D scatter plots

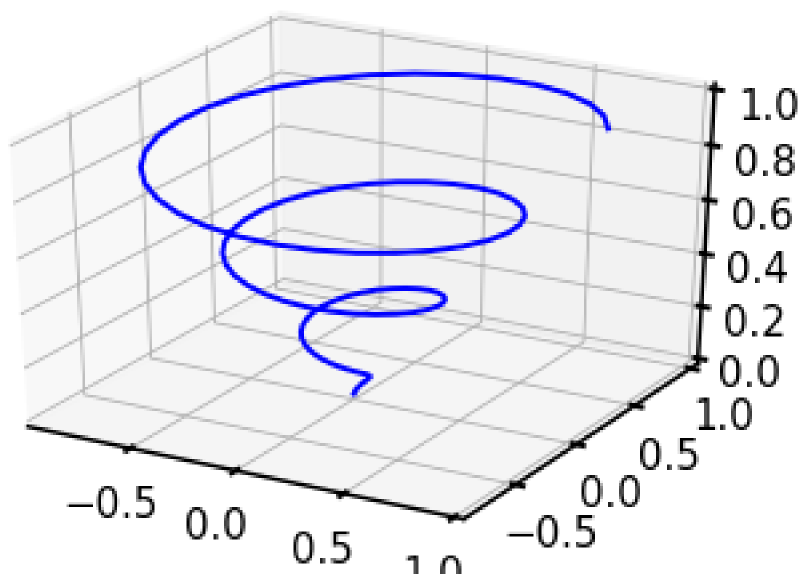
[209]:
```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = plt.axes(projection='3d')

z = np.linspace(0, 1, 100)
x = z * np.sin(20 * z)
y = z * np.cos(20 * z)

c = np.sqrt(x**2 + y**2)
```

```
ax.scatter(x, y, z, c=c);
#ax.set_zscale('log')
```

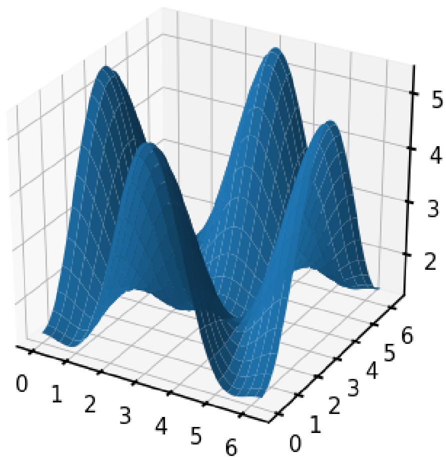


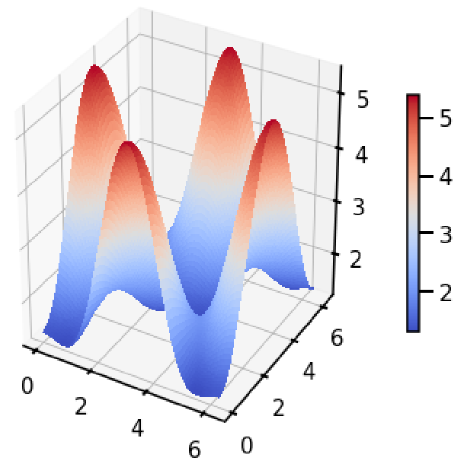
[210]:
```
fig = plt.figure()
ax = plt.axes(projection='3d')

ax.plot(x, y, z, '-b')
```

[210]: [<mpl_toolkits.mplot3d.art3d.Line3D at 0x7f9bba854290>]

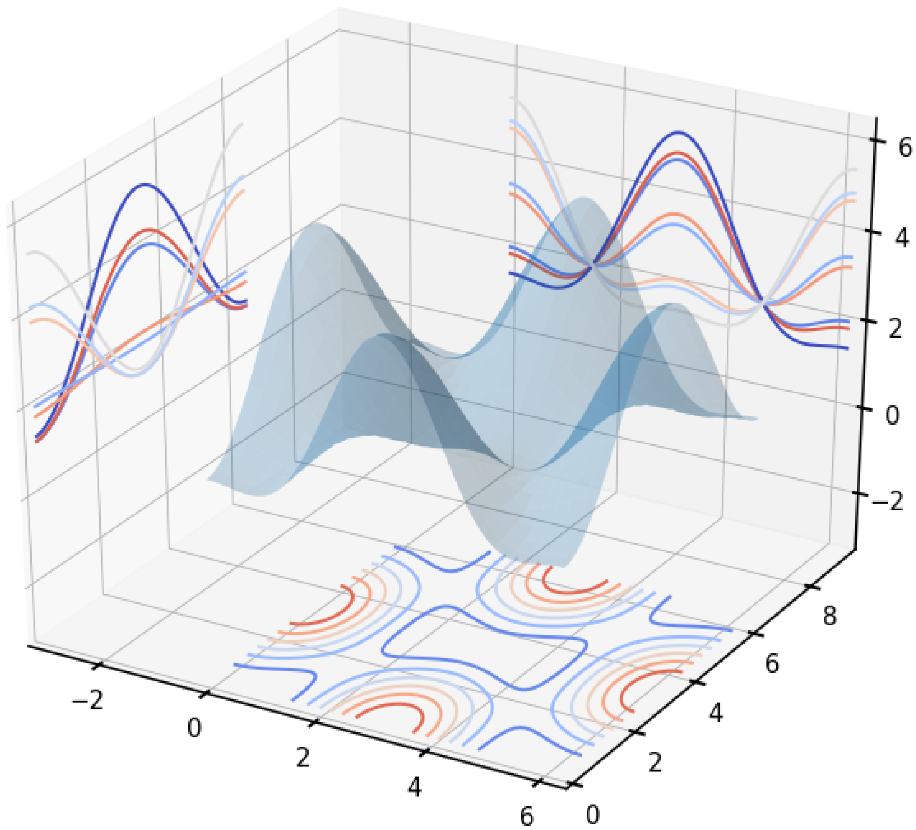


[211]:
```
alpha = 0.7
phi_ext = 2 * np.pi * 0.5

def flux_qubit_potential(phi_m, phi_p):
    return 2 + alpha - 2 * np.cos(phi_p)*np.cos(phi_m) - alpha * np.cos(phi_ext␣
↪- 2*phi_p)

phi_m = np.linspace(0, 2*np.pi, 100)
phi_p = np.linspace(0, 2*np.pi, 100)
X,Y = np.meshgrid(phi_p, phi_m)
Z = flux_qubit_potential(X, Y).T
fig = plt.figure(figsize=(14,6))

# `ax` is a 3D-aware axis instance, because of the projection='3d' keyword␣
↪argument to add_subplot
ax = fig.add_subplot(1, 2, 1, projection='3d')

p = ax.plot_surface(X, Y, Z, rstride=4, cstride=4, linewidth=0)

# surface_plot with color grading and color bar
ax = fig.add_subplot(1, 2, 2, projection='3d')
p = ax.plot_surface(X, Y, Z, rstride=1, cstride=1, cmap=plt.cm.coolwarm,␣
↪linewidth=0,
                    antialiased=False)
cb = fig.colorbar(p, shrink=0.5)
```





[212]:
```
pi = np.pi
fig = plt.figure(figsize=(12,10))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-pi, cmap=plt.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-pi, cmap=plt.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*pi, cmap=plt.cm.coolwarm)

ax.set_xlim3d(-pi, 2*pi);
ax.set_ylim3d(0, 3*pi);
ax.set_zlim3d(-pi, 2*pi);
```

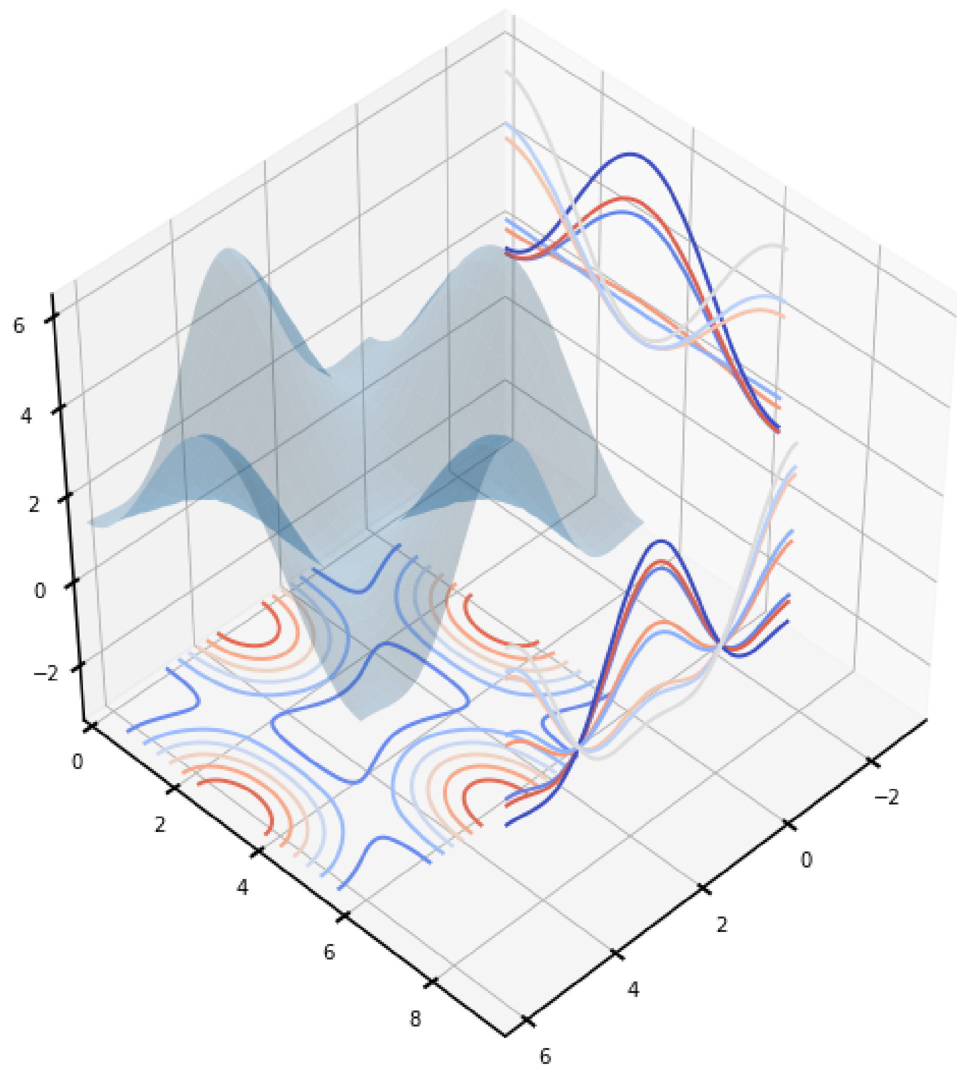


[216]:
```
# Interactive rotating the plot
%matplotlib tk
fig = plt.figure(figsize=(12,10))

ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-pi, cmap=plt.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-pi, cmap=plt.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*pi, cmap=plt.cm.coolwarm)

ax.set_xlim3d(-pi, 2*pi);
ax.set_ylim3d(0, 3*pi);
ax.set_zlim3d(-pi, 2*pi);
```

Saving plots

[217]:
```
%matplotlib inline
fig = plt.figure()
```

```
ax = fig.add_subplot(1,1,1, projection='3d')

ax.plot_surface(X, Y, Z, rstride=4, cstride=4, alpha=0.25)
cset = ax.contour(X, Y, Z, zdir='z', offset=-pi, cmap=plt.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='x', offset=-pi, cmap=plt.cm.coolwarm)
cset = ax.contour(X, Y, Z, zdir='y', offset=3*pi, cmap=plt.cm.coolwarm)
ax.view_init(45, 45)
ax.set_xlim3d(-pi, 2*pi);
ax.set_ylim3d(0, 3*pi);
ax.set_zlim3d(-pi, 2*pi);
fig.set_size_inches(10,10)
fig.savefig('Fig1.pdf')
```



[ ]:
```
ls *pdf
```

Other tutorials: http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html

Access and clear the current figure and axe

[ ]:
```
fig, ax = plt.subplots()
print(plt.gca() is ax) # You can get the  current axes with gca
print(plt.gcf() is fig) # The same for the current axes.
# But it's preferable to store them in a variable when creating
plt.clf() # clear the current figure
plt.cla() # clear the current axes
fig.clf() # clear a given figure
ax.cla(); # clear a given axes
```

What's happen when not in a Notebook? plt.show() and plt.ion() commands We are here in a Notebook, but most of the time, you will execute programs from a script or using the command line in a terminal.

When using plot, scatter or any other plotting tool, the figure will not appear when typing the command, you need to send the *plt.show()* command to pop-up it (or them if you did more than one figure). And you will loose the interactivity with the command line! You will recover it once the figure windows are closed.

The way to change this behaviour is to call the *plt.ion()* command (interactive On).

If you are working within an ipython session created with the –pylab option, it is done by default.