Data Cleaning:

- Missing data [repalce,fillna,dropna]
- fill missing values using sklearn
- duplicate data

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
a = np.array([[1,2,np.nan,3,4],[10,22,34,67,89],[23,45,89,67,90],
              [np.nan,45,90,np.nan,90],[23,np.nan,90,67,89]])
```

In [3]:
```
a
```

Out[3]:
```
array([[ 1.,  2., nan,  3.,  4.],
       [10., 22., 34., 67., 89.],
       [23., 45., 89., 67., 90.],
       [nan, 45., 90., nan, 90.],
       [23., nan, 90., 67., 89.]])
```

In [4]:
```
a.shape
```

Out[4]:
```
(5, 5)
```

In [5]:
```
d = pd.DataFrame(a,columns = ["one","two","three","four","five"],index=["a",
d
```

Out[5]:

	one	two	three	four	five
a	1.0	2.0	NaN	3.0	4.0
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0
d	NaN	45.0	90.0	NaN	90.0
e	23.0	NaN	90.0	67.0	89.0

In [6]:
```
d.columns
```

Out[6]:
```
Index(['one', 'two', 'three', 'four', 'five'], dtype='object')
```

In [7]:
```
d.index
```

Out[7]:
```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [8]:
```
1 d.isnull()
```

Out[8]:

	one	two	three	four	five
a	False	False	True	False	False
b	False	False	False	False	False
c	False	False	False	False	False
d	True	False	False	True	False
e	False	True	False	False	False

In [9]:
```
1 d.isnull().sum()
```

Out[9]:
```
one      1
two      1
three    1
four     1
five     0
dtype: int64
```

In [10]:
```
1 # we can handle null values in two ways
2 # dropna
3 # fillna
```

In [11]:
```
1 d.dropna()
```

Out[11]:

	one	two	three	four	five
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0

In [12]:
```
1 d
```

Out[12]:

	one	two	three	four	five
a	1.0	2.0	NaN	3.0	4.0
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0
d	NaN	45.0	90.0	NaN	90.0
e	23.0	NaN	90.0	67.0	89.0

In [13]:
```
1 d.dropna(axis=1)
```

Out[13]:

	five
a	4.0
b	89.0
c	90.0
d	90.0
e	89.0

In [14]:
```
1 d.replace(np.nan,0)
```

Out[14]:

	one	two	three	four	five
a	1.0	2.0	0.0	3.0	4.0
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0
d	0.0	45.0	90.0	0.0	90.0
e	23.0	0.0	90.0	67.0	89.0

In [15]:
```
1 d
```

Out[15]:

	one	two	three	four	five
a	1.0	2.0	NaN	3.0	4.0
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0
d	NaN	45.0	90.0	NaN	90.0
e	23.0	NaN	90.0	67.0	89.0

In [16]:
```
1 d["one"] = d["one"].replace(np.nan,0)
2
```

In [17]:
```
1 d
```

Out[17]:

	one	two	three	four	five
a	1.0	2.0	NaN	3.0	4.0
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0
d	0.0	45.0	90.0	NaN	90.0
e	23.0	NaN	90.0	67.0	89.0

In [18]:
```
1 d["two"].mean()
```

Out[18]: 28.5

In [19]:
```
1 d["two"].replace(np.nan,d["two"].mean())
```

Out[19]:
```
a     2.0
b    22.0
c    45.0
d    45.0
e    28.5
Name: two, dtype: float64
```

In [20]:
```
1 d["two"] = d["two"].fillna(d["two"].mean())
```

In [21]:
```
1 d["three"].median()
```

Out[21]: 89.5

In [22]:
```
1 d["three"] = d["three"].fillna(d["three"].median())
```

In [23]:
```
1 d
```

Out[23]:

	one	two	three	four	five
a	1.0	2.0	89.5	3.0	4.0
b	10.0	22.0	34.0	67.0	89.0
c	23.0	45.0	89.0	67.0	90.0
d	0.0	45.0	90.0	NaN	90.0
e	23.0	28.5	90.0	67.0	89.0

In [24]:
```
1 s = pd.DataFrame({"lan":["english",np.nan,"telugu","english"],"alp":["a",np.
```

In [25]:
```
1 s
```

Out[25]:

	lan	alp
0	english	a
1	NaN	NaN
2	telugu	b
3	english	NaN

In [26]:
```
1 s["lan"].fillna(method="bfill")
```

Out[26]:
```
0    english
1     telugu
2     telugu
3    english
Name: lan, dtype: object
```

In [27]:
```
1 s["alp"].fillna(method="ffill")
```

Out[27]:
```
0    a
1    a
2    b
3    b
Name: alp, dtype: object
```

In [28]:
```
1 #s["alp"].isnull().sum()
```

In [29]:
```
1 d1 = pd.DataFrame({"sno":[1,2,2,3,3,4,5,6],"names":["a","b","b","c","c","d",
```

In [30]:
```
1 d1
```

Out[30]:

	sno	names
0	1	a
1	2	b
2	2	b
3	3	c
4	3	c
5	4	d
6	5	e
7	6	g

In [31]:
```
1 d1.duplicated()
```

Out[31]:
```
0    False
1    False
2     True
3    False
4     True
5    False
6    False
7    False
dtype: bool
```

In [32]:
```
1 d1[d1.duplicated()]
```

Out[32]:

	sno	names
2	2	b
4	3	c

In [33]:
```
1 d1["sno"]
```

Out[33]:
```
0    1
1    2
2    2
3    3
4    3
5    4
6    5
7    6
Name: sno, dtype: int64
```

In [34]:
```
1 d1.drop("sno",inplace = True,axis=1)
```

In [35]:
```
1 d1
```

Out[35]:

	names
0	a
1	b
2	b
3	c
4	c
5	d
6	e
7	g

In [36]:
```
1 d1["names"].dtype
```

Out[36]:
```
dtype('O')
```

Visuvalization

- Matplotlib
- Seaborn
- GGplot
- plotpy

Matplotlib

- 2D visuvalization

In [37]:
```
1 #pip install matplotlib
```

In [1]:
```
1 import matplotlib.pyplot as plt
```

In [39]:
```
1 print(dir(plt))
```

```
['Annotation', 'Arrow', 'Artist', 'AutoLocator', 'Axes', 'Button', 'Circle', 'F
igure', 'FigureCanvasBase', 'FixedFormatter', 'FixedLocator', 'FormatStrFormatt
er', 'Formatter', 'FuncFormatter', 'GridSpec', 'IndexLocator', 'Line2D', 'Linea
rLocator', 'Locator', 'LogFormatter', 'LogFormatterExponent', 'LogFormatterMath
text', 'LogLocator', 'MaxNLocator', 'MultipleLocator', 'Normalize', 'NullFormat
ter', 'NullLocator', 'Number', 'PolarAxes', 'Polygon', 'Rectangle', 'ScalarForm
atter', 'Slider', 'Subplot', 'SubplotTool', 'Text', 'TickHelper', 'Widget', '_I
NSTALL_FIG_OBSERVER', '_IP_REGISTERED', '__builtins__', '__cached__', '__doc_
_', '__file__', '__loader__', '__name__', '__package__', '__spec__', '_auto_dra
w_if_interactive', '_autogen_docstring', '_backend_mod', '_get_running_interact
ive_framework', '_interactive_bk', '_log', '_pylab_helpers', '_setp', '_setup_p
yplot_info_docstrings', '_show', '_string_to_bool', 'acorr', 'angle_spectrum',
'annotate', 'arrow', 'autoscale', 'autumn', 'axes', 'axhline', 'axhspan', 'axi
s', 'axvline', 'axvspan', 'bar', 'barbs', 'barh', 'bone', 'box', 'boxplot', 'br
oken_barh', 'cla', 'clabel', 'clf', 'clim', 'close', 'cm', 'cohere', 'colorba
r', 'colormaps', 'connect', 'contour', 'contourf', 'cool', 'copper', 'csd', 'cy
cler', 'dedent', 'delaxes', 'deprecated', 'disconnect', 'docstring', 'draw', 'd
raw_all', 'draw_if_interactive', 'errorbar', 'eventplot', 'figaspect', 'figimag
e', 'figlegend', 'fignum_exists', 'figtext', 'figure', 'fill', 'fill_between',
'fill_betweenx', 'findobj', 'flag', 'gca', 'gcf', 'gci', 'get', 'get_backend',
'get_cmap', 'get_current_fig_manager', 'get_figlabels', 'get_fignums', 'get_plo
t_commands', 'get_scale_docs', 'get_scale_names', 'getp', 'ginput', 'gray', 'gr
id', 'hexbin', 'hist', 'hist2d', 'hlines', 'hot', 'hsv', 'importlib', 'imread',
'imsave', 'imshow', 'inferno', 'inspect', 'install_repl_displayhook', 'interact
ive', 'ioff', 'ion', 'isinteractive', 'jet', 'legend', 'locator_params', 'loggi
ng', 'loglog', 'magma', 'magnitude_spectrum', 'margins', 'matplotlib', 'matsho
w', 'minorticks_off', 'minorticks_on', 'mlab', 'new_figure_manager', 'nipy_spec
tral', 'np', 'pause', 'pcolor', 'pcolormesh', 'phase_spectrum', 'pie', 'pink',
'plasma', 'plot', 'plot_date', 'plotfile', 'plotting', 'polar', 'prism', 'psd',
'pylab_setup', 'quiver', 'quiverkey', 'rc', 'rcParams', 'rcParamsDefault', 'rcP
aramsOrig', 'rc_context', 'rcdefaults', 'rcsetup', 're', 'register_cmap', 'rgri
ds', 'savefig', 'sca', 'scatter', 'sci', 'semilogx', 'semilogy', 'set_cmap', 's
etp', 'show', 'silent_list', 'specgram', 'spring', 'spy', 'stackplot', 'stem',
'step', 'streamplot', 'style', 'subplot', 'subplot2grid', 'subplot_tool', 'subp
lots', 'subplots_adjust', 'summer', 'suptitle', 'switch_backend', 'sys', 'tabl
e', 'text', 'thetagrids', 'tick_params', 'ticklabel_format', 'tight_layout', 't
ime', 'title', 'tricontour', 'tricontourf', 'tripcolor', 'triplot', 'twinx', 't
winy', 'uninstall_repl_displayhook', 'violinplot', 'viridis', 'vlines', 'waitfo
rbuttonpress', 'warn_deprecated', 'warnings', 'winter', 'xcorr', 'xkcd', 'xlabe
l', 'xlim', 'xscale', 'xticks', 'ylabel', 'ylim', 'yscale', 'yticks']
```

In [40]:
```
1 help(plt)
```

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

NAME
    matplotlib.pyplot

DESCRIPTION
    `matplotlib.pyplot` is a state-based interface to matplotlib. It provides
    a MATLAB-like way of plotting.

    pyplot is mainly intended for interactive plots and simple cases of program
matic
    plot generation::

        import numpy as np
        import matplotlib.pyplot as plt

        x = np.arange(0, 5, 0.1)
        y = np.sin(x)
        plt.plot(x, y)

    The object-oriented API is recommended for more complex plots.

FUNCTIONS
    acorr(x, *, data=None, **kwargs)
        Plot the autocorrelation of *x*.

        Parameters
        ----------

        x : sequence of scalar

        detrend : callable, optional, default: `mlab.detrend_none`
            *x* is detrended by the *detrend* callable. Default is no
            normalization.

        normed : bool, optional, default: True
            If ``True``, input vectors are normalised to unit length.

        usevlines : bool, optional, default: True
            If ``True``, `Axes.vlines` is used to plot the vertical lines from
            the origin to the acorr. Otherwise, `Axes.plot` is used.

        maxlags : int, optional, default: 10
            Number of lags to show. If ``None``, will return all
            ``2 * len(x) - 1`` lags.

        Returns
        -------
        lags : array (length ``2*maxlags+1``)
            lag vector.
        c : array  (length ``2*maxlags+1``)
            auto correlation vector.
        line : `.LineCollection` or `.Line2D`
            `.Artist` added to the axes of the correlation.
```

```
             `.LineCollection` if *usevlines* is True
             `.Line2D` if *usevlines* is False
        b : `.Line2D` or None
            Horizontal line at 0 if *usevlines* is True
            None *usevlines* is False

        Other Parameters
        ----------------
        linestyle : `.Line2D` property, optional, default: None
            Only used if usevlines is ``False``.

        marker : str, optional, default: 'o'

        Notes
        -----
        The cross correlation is performed with :func:`numpy.correlate` with
        ``mode = 2``.

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

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

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

    angle_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None,
*, data=None, **kwargs)
        Plot the angle spectrum.

        Call signature::

          angle_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
                         pad_to=None, sides='default', **kwargs)

        Compute the angle spectrum (wrapped phase spectrum) of *x*.
        Data is padded to a length of *pad_to* and the windowing function
        *window* is applied to the signal.

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data.

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*. To create window
```

```
            vectors see :func:`window_hanning`, :func:`window_none`,
            :func:`numpy.blackman`, :func:`numpy.hamming`,
            :func:`numpy.bartlett`, :func:`scipy.signal`,
            :func:`scipy.signal.get_window`, etc. The default is
            :func:`window_hanning`.  If a function is passed as the
            argument, it must take a data segment as an argument and
            return the windowed version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  While not increasing the actual resolution of
            the spectrum (the minimum distance between resolvable peaks),
            this can give more points in the plot, allowing for more
            detail. This corresponds to the *n* parameter in the call to fft().
            The default is None, which sets *pad_to* equal to the length of the
            input signal (i.e. no padding).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        Returns
        -------
        spectrum : 1-D array
            The values for the angle spectrum in radians (real valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *spectrum*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.

        Other Parameters
        ----------------
        **kwargs :
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

        See Also
        --------
        :func:`magnitude_spectrum`
            :func:`angle_spectrum` plots the magnitudes of the corresponding
            frequencies.

        :func:`phase_spectrum`
            :func:`phase_spectrum` plots the unwrapped version of this
            function.

        :func:`specgram`
            :func:`specgram` can plot the angle spectrum of segments within the
            signal in a colormap.

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

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
```

```
                following arguments are replaced by **data[<arg>]**:

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

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

    annotate(s, xy, *args, **kwargs)
        Annotate the point *xy* with text *s*.

        In the simplest form, the text is placed at *xy*.

        Optionally, the text can be displayed in another position *xytext*.
        An arrow pointing from the text to the annotated point *xy* can then
        be added by defining *arrowprops*.

        Parameters
        ----------
        s : str
            The text of the annotation.

        xy : (float, float)
            The point *(x,y)* to annotate.

        xytext : (float, float), optional
            The position *(x,y)* to place the text at.
            If *None*, defaults to *xy*.

        xycoords : str, `.Artist`, `.Transform`, callable or tuple, optional

            The coordinate system that *xy* is given in. The following types
            of values are supported:

            - One of the following strings:

              =================   ============================================
              Value               Description
              =================   ============================================
              'figure points'     Points from the lower left of the figure
              'figure pixels'     Pixels from the lower left of the figure
              'figure fraction'   Fraction of figure from lower left
              'axes points'       Points from lower left corner of axes
              'axes pixels'       Pixels from lower left corner of axes
              'axes fraction'     Fraction of axes from lower left
              'data'              Use the coordinate system of the object being
                                  annotated (default)
              'polar'             *(theta,r)* if not native 'data' coordinates
              =================   ============================================

            - An `.Artist`: *xy* is interpreted as a fraction of the artists
              `~matplotlib.transforms.Bbox`. E.g. *(0, 0)* would be the lower
              left corner of the bounding box and *(0.5, 1)* would be the
              center top of the bounding box.

            - A `.Transform` to transform *xy* to screen coordinates.
```

```
        - A function with one of the following signatures::

            def transform(renderer) -> Bbox
            def transform(renderer) -> Transform

          where *renderer* is a `.RendererBase` subclass.

          The result of the function is interpreted like the `.Artist` and
          `.Transform` cases above.

        - A tuple *(xcoords, ycoords)* specifying separate coordinate
          systems for *x* and *y*. *xcoords* and *ycoords* must each be
          of one of the above described types.

        See :ref:`plotting-guide-annotation` for more details.

        Defaults to 'data'.

    textcoords : str, `.Artist`, `.Transform`, callable or tuple, optional
        The coordinate system that *xytext* is given in.

        All *xycoords* values are valid as well as the following
        strings:

        =================   =========================================
        Value               Description
        =================   =========================================
        'offset points'     Offset (in points) from the *xy* value
        'offset pixels'     Offset (in pixels) from the *xy* value
        =================   =========================================

        Defaults to the value of *xycoords*, i.e. use the same coordinate
        system for annotation point and text position.

    arrowprops : dict, optional
        The properties used to draw a
        `~matplotlib.patches.FancyArrowPatch` arrow between the
        positions *xy* and *xytext*.

        If *arrowprops* does not contain the key 'arrowstyle' the
        allowed keys are:

        ==========   ======================================================
        Key          Description
        ==========   ======================================================
        width        The width of the arrow in points
        headwidth    The width of the base of the arrow head in points
        headlength   The length of the arrow head in points
        shrink       Fraction of total length to shrink from both ends
        ?            Any key to :class:`matplotlib.patches.FancyArrowPatch`
        ==========   ======================================================

        If *arrowprops* contains the key 'arrowstyle' the
        above keys are forbidden.  The allowed values of
        ``'arrowstyle'`` are:

        ============   =============================================
```

```
    Name           Attrs
    ============   ============================================
    ``'-'``        None
    ``'->'``       head_length=0.4,head_width=0.2
    ``'-['``       widthB=1.0,lengthB=0.2,angleB=None
    ``'|-|'``      widthA=1.0,widthB=1.0
    ``'-|>'``      head_length=0.4,head_width=0.2
    ``'<-'``       head_length=0.4,head_width=0.2
    ``'<->'``      head_length=0.4,head_width=0.2
    ``'<|-'``      head_length=0.4,head_width=0.2
    ``'<|-|>'``    head_length=0.4,head_width=0.2
    ``'fancy'``    head_length=0.4,head_width=0.4,tail_width=0.4
    ``'simple'``   head_length=0.5,head_width=0.5,tail_width=0.2
    ``'wedge'``    tail_width=0.3,shrink_factor=0.5
    ============   ============================================

    Valid keys for `~matplotlib.patches.FancyArrowPatch` are:

    ===============  ================================================
    Key              Description
    ===============  ================================================
    arrowstyle       the arrow style
    connectionstyle  the connection style
    relpos           default is (0.5, 0.5)
    patchA           default is bounding box of the text
    patchB           default is None
    shrinkA          default is 2 points
    shrinkB          default is 2 points
    mutation_scale   default is text size (in points)
    mutation_aspect  default is 1.
    ?                any key for :class:`matplotlib.patches.PathPatch`
    ===============  ================================================

    Defaults to None, i.e. no arrow is drawn.

annotation_clip : bool or None, optional
    Whether to draw the annotation when the annotation point *xy* is
    outside the axes area.

    - If *True*, the annotation will only be drawn when *xy* is
      within the axes.
    - If *False*, the annotation will always be drawn.
    - If *None*, the annotation will only be drawn when *xy* is
      within the axes and *xycoords* is 'data'.

    Defaults to *None*.

**kwargs
    Additional kwargs are passed to `~matplotlib.text.Text`.

Returns
-------
annotation : `.Annotation`

See Also
--------
:ref:`plotting-guide-annotation`.
```

```
    arrow(x, y, dx, dy, **kwargs)
        Add an arrow to the axes.

        This draws an arrow from ``(x, y)`` to ``(x+dx, y+dy)``.

        Parameters
        ----------
        x, y : float
            The x/y-coordinate of the arrow base.
        dx, dy : float
            The length of the arrow along x/y-direction.

        Returns
        -------
        arrow : `.FancyArrow`
            The created `.FancyArrow` object.

        Other Parameters
        ----------------
        **kwargs
            Optional kwargs (inherited from `.FancyArrow` patch) control the
            arrow construction and properties:

        Constructor arguments
          *width*: float (default: 0.001)
            width of full arrow tail

          *length_includes_head*: bool (default: False)
            True if head is to be counted in calculating the length.

          *head_width*: float or None (default: 3*width)
            total width of the full arrow head

          *head_length*: float or None (default: 1.5 * head_width)
            length of arrow head

          *shape*: ['full', 'left', 'right'] (default: 'full')
            draw the left-half, right-half, or full arrow

          *overhang*: float (default: 0)
            fraction that the arrow is swept back (0 overhang means
            triangular shape). Can be negative or greater than one.

          *head_starts_at_zero*: bool (default: False)
            if True, the head starts being drawn at coordinate 0
            instead of ending at coordinate 0.

        Other valid kwargs (inherited from :class:`Patch`) are:
          agg_filter: a filter function, which takes a (m, n, 3) float array an
    d a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: unknown
          capstyle: {'butt', 'round', 'projecting'}
          clip_box: `.Bbox`
          clip_on: bool
```

```
              clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
              color: color
              contains: callable
              edgecolor: color or None or 'auto'
              facecolor: color or None
              figure: `.Figure`
              fill: bool
              gid: str
              hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
              in_layout: bool
              joinstyle: {'miter', 'round', 'bevel'}
              label: object
              linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
              linewidth: float or None for default
              path_effects: `.AbstractPathEffect`
              picker: None or bool or float or callable
              rasterized: bool or None
              sketch_params: (scale: float, length: float, randomness: float)
              snap: bool or None
              transform: `.Transform`
              url: str
              visible: bool
              zorder: float

            Notes
            -----
            The resulting arrow is affected by the axes aspect ratio and limits.
            This may produce an arrow whose head is not square with its stem. To
            create an arrow whose head is square with its stem,
            use :meth:`annotate` for example:

            >>> ax.annotate("", xy=(0.5, 0.5), xytext=(0, 0),
            ...             arrowprops=dict(arrowstyle="->"))

        autoscale(enable=True, axis='both', tight=None)
            Autoscale the axis view to the data (toggle).

            Convenience method for simple axis view autoscaling.
            It turns autoscaling on or off, and then,
            if autoscaling for either axis is on, it performs
            the autoscaling on the specified axis or axes.

            Parameters
            ----------
            enable : bool or None, optional
                True (default) turns autoscaling on, False turns it off.
                None leaves the autoscaling state unchanged.

            axis : {'both', 'x', 'y'}, optional
                which axis to operate on; default is 'both'

            tight: bool or None, optional
                If True, set view limits to data limits;
                if False, let the locator and margins expand the view limits;
                if None, use tight scaling if the only artist is an image,
                otherwise treat *tight* as False.
```

```
                The *tight* setting is retained for future autoscaling
                until it is explicitly changed.

        autumn()
            Set the colormap to "autumn".

            This changes the default colormap as well as the colormap of the curren
t
            image if there is one. See ``help(colormaps)`` for more information.

        axes(arg=None, **kwargs)
            Add an axes to the current figure and make it the current axes.

            Call signatures::

                plt.axes()
                plt.axes(rect, projection=None, polar=False, **kwargs)
                plt.axes(ax)

            Parameters
            ----------
            arg : { None, 4-tuple, Axes }
                The exact behavior of this function depends on the type:

                - *None*: A new full window axes is added using
                  ``subplot(111, **kwargs)``
                - 4-tuple of floats *rect* = ``[left, bottom, width, height]``.
                  A new axes is added with dimensions *rect* in normalized
                  (0, 1) units using `~.Figure.add_axes` on the current figure.
                - `~.axes.Axes`: This is equivalent to `.pyplot.sca`.
                  It sets the current axes to *arg*. Note: This implicitly
                  changes the current figure to the parent of *arg*.

                  .. note:: The use of an `.axes.Axes` as an argument is deprecated
                            and will be removed in v3.0. Please use `.pyplot.sca`
                            instead.

            projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'pola
r', 'rectilinear', str}, optional
                The projection type of the `~.axes.Axes`. *str* is the name of
                a costum projection, see `~matplotlib.projections`. The default
                None results in a 'rectilinear' projection.

            polar : boolean, optional
                If True, equivalent to projection='polar'.

            sharex, sharey : `~.axes.Axes`, optional
                Share the x or y `~matplotlib.axis` with sharex and/or sharey.
                The axis will have the same limits, ticks, and scale as the axis
                of the shared axes.


            label : str
                A label for the returned axes.

            Other Parameters
            ----------------
```

```
        **kwargs
            This method also takes the keyword arguments for
            the returned axes class. The keyword arguments for the
            rectilinear axes class `~.axes.Axes` can be found in
            the following table but there might also be other keyword
            arguments if another projection is used, see the actual axes
            class.
              adjustable: {'box', 'datalim'}
          agg_filter: a filter function, which takes a (m, n, 3) float array an
d a dpi value, and returns a (m, n, 3) array
          alpha: float
          anchor: 2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
          animated: bool
          aspect: {'auto', 'equal'} or num
          autoscale_on: bool
          autoscalex_on: bool
          autoscaley_on: bool
          axes_locator: Callable[[Axes, Renderer], Bbox]
          axisbelow: bool or 'line'
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          contains: callable
          facecolor: color
          fc: color
          figure: `.Figure`
          frame_on: bool
          gid: str
          in_layout: bool
          label: object
          navigate: bool
          navigate_mode: unknown
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          position: [left, bottom, width, height] or `~matplotlib.transforms.Bb
ox`
          rasterization_zorder: float or None
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          title: str
          transform: `.Transform`
          url: str
          visible: bool
          xbound: unknown
          xlabel: str
          xlim: (left: float, right: float)
          xmargin: float greater than -0.5
          xscale: {"linear", "log", "symlog", "logit", ...}
          xticklabels: List[str]
          xticks: list
          ybound: unknown
          ylabel: str
          ylim: (bottom: float, top: float)
          ymargin: float greater than -0.5
          yscale: {"linear", "log", "symlog", "logit", ...}
```

```
      yticklabels: List[str]
      yticks: list
      zorder: float

    Returns
    -------
    axes : `~.axes.Axes` (or a subclass of `~.axes.Axes`)
        The returned axes class depends on the projection used. It is
        `~.axes.Axes` if rectilinear projection are used and
        `.projections.polar.PolarAxes` if polar projection
        are used.

    Notes
    -----
    If the figure already has a axes with key (*args*,
    *kwargs*) then it will simply make that axes current and
    return it.  This behavior is deprecated. Meanwhile, if you do
    not want this behavior (i.e., you want to force the creation of a
    new axes), you must use a unique set of args and kwargs.  The axes
    *label* attribute has been exposed for this purpose: if you want
    two axes that are otherwise identical to be added to the figure,
    make sure you give them unique labels.

    See Also
    --------
    .Figure.add_axes
    .pyplot.subplot
    .Figure.add_subplot
    .Figure.subplots
    .pyplot.subplots

    Examples
    --------
    ::

        #Creating a new full window axes
        plt.axes()

        #Creating a new axes with specified dimensions and some kwargs
        plt.axes((left, bottom, width, height), facecolor='w')

axhline(y=0, xmin=0, xmax=1, **kwargs)
    Add a horizontal line across the axis.

    Parameters
    ----------
    y : scalar, optional, default: 0
        y position in data coordinates of the horizontal line.

    xmin : scalar, optional, default: 0
        Should be between 0 and 1, 0 being the far left of the plot, 1 the
        far right of the plot.

    xmax : scalar, optional, default: 1
        Should be between 0 and 1, 0 being the far left of the plot, 1 the
        far right of the plot.
```

```
        Returns
        -------
        line : :class:`~matplotlib.lines.Line2D`

        Other Parameters
        ----------------
        **kwargs :
            Valid kwargs are :class:`~matplotlib.lines.Line2D` properties,
            with the exception of 'transform':

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

        See also
        --------
```

```
    hlines : Add horizontal lines in data coordinates.
    axhspan : Add a horizontal span (rectangle) across the axis.

    Examples
    --------

    * draw a thick red hline at 'y' = 0 that spans the xrange::

        >>> axhline(linewidth=4, color='r')

    * draw a default hline at 'y' = 1 that spans the xrange::

        >>> axhline(y=1)

    * draw a default hline at 'y' = .5 that spans the middle half of
      the xrange::

        >>> axhline(y=.5, xmin=0.25, xmax=0.75)

axhspan(ymin, ymax, xmin=0, xmax=1, **kwargs)
    Add a horizontal span (rectangle) across the axis.

    Draw a horizontal span (rectangle) from *ymin* to *ymax*.
    With the default values of *xmin* = 0 and *xmax* = 1, this
    always spans the xrange, regardless of the xlim settings, even
    if you change them, e.g., with the :meth:`set_xlim` command.
    That is, the horizontal extent is in axes coords: 0=left,
    0.5=middle, 1.0=right but the *y* location is in data
    coordinates.

    Parameters
    ----------
    ymin : float
           Lower limit of the horizontal span in data units.
    ymax : float
           Upper limit of the horizontal span in data units.
    xmin : float, optional, default: 0
           Lower limit of the vertical span in axes (relative
           0-1) units.
    xmax : float, optional, default: 1
           Upper limit of the vertical span in axes (relative
           0-1) units.

    Returns
    -------
    Polygon : `~matplotlib.patches.Polygon`

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

      agg_filter: a filter function, which takes a (m, n, 3) float array an
d a dpi value, and returns a (m, n, 3) array
      alpha: float or None
      animated: bool
      antialiased: unknown
      capstyle: {'butt', 'round', 'projecting'}
```

```
                clip_box: `.Bbox`
                clip_on: bool
                clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
                color: color
                contains: callable
                edgecolor: color or None or 'auto'
                facecolor: color or None
                figure: `.Figure`
                fill: bool
                gid: str
                hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
                in_layout: bool
                joinstyle: {'miter', 'round', 'bevel'}
                label: object
                linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
                linewidth: float or None for default
                path_effects: `.AbstractPathEffect`
                picker: None or bool or float or callable
                rasterized: bool or None
                sketch_params: (scale: float, length: float, randomness: float)
                snap: bool or None
                transform: `.Transform`
                url: str
                visible: bool
                zorder: float

            See Also
            --------
            axvspan : Add a vertical span across the axes.

        axis(*v, **kwargs)
            Convenience method to get or set some axis properties.

            Call signatures::

              xmin, xmax, ymin, ymax = axis()
              xmin, xmax, ymin, ymax = axis(xmin, xmax, ymin, ymax)
              xmin, xmax, ymin, ymax = axis(option)
              xmin, xmax, ymin, ymax = axis(**kwargs)

            Parameters
            ----------
            xmin, ymin, xmax, ymax : float, optional
                The axis limits to be set. Either none or all of the limits must
                be given.

            option : str
                Possible values:

                ======== ==========================================================
                Value    Description
                ======== ==========================================================
                'on'     Turn on axis lines and labels.
                'off'    Turn off axis lines and labels.
                'equal'  Set equal scaling (i.e., make circles circular) by
                         changing axis limits.
```

```
                'scaled' Set equal scaling (i.e., make circles circular) by
                         changing dimensions of the plot box.
                'tight'  Set limits just large enough to show all data.
                'auto'   Automatic scaling (fill plot box with data).
                'normal' Same as 'auto'; deprecated.
                'image'  'scaled' with axis limits equal to data limits.
                'square' Square plot; similar to 'scaled', but initially forcing
                         ``xmax-xmin = ymax-ymin``.
                ======== ============================================================

            emit : bool, optional, default *True*
                Whether observers are notified of the axis limit change.
                This option is passed on to `~.Axes.set_xlim` and
                `~.Axes.set_ylim`.

            Returns
            -------
            xmin, xmax, ymin, ymax : float
                The axis limits.

            See also
            --------
            matplotlib.axes.Axes.set_xlim
            matplotlib.axes.Axes.set_ylim

        axvline(x=0, ymin=0, ymax=1, **kwargs)
            Add a vertical line across the axes.

            Parameters
            ----------
            x : scalar, optional, default: 0
                x position in data coordinates of the vertical line.

            ymin : scalar, optional, default: 0
                Should be between 0 and 1, 0 being the bottom of the plot, 1 the
                top of the plot.

            ymax : scalar, optional, default: 1
                Should be between 0 and 1, 0 being the bottom of the plot, 1 the
                top of the plot.

            Returns
            -------
            line : :class:`~matplotlib.lines.Line2D`

            Other Parameters
            ----------------
            **kwargs :
                Valid kwargs are :class:`~matplotlib.lines.Line2D` properties,
                with the exception of 'transform':

                  agg_filter: a filter function, which takes a (m, n, 3) float arra
    y 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` | Non
e]
            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

        Examples
        --------
        * draw a thick red vline at *x* = 0 that spans the yrange::

            >>> axvline(linewidth=4, color='r')

        * draw a default vline at *x* = 1 that spans the yrange::

            >>> axvline(x=1)

        * draw a default vline at *x* = .5 that spans the middle half of
          the yrange::

            >>> axvline(x=.5, ymin=0.25, ymax=0.75)

        See also
        --------
        vlines : Add vertical lines in data coordinates.
```

```
        axvspan : Add a vertical span (rectangle) across the axis.

    axvspan(xmin, xmax, ymin=0, ymax=1, **kwargs)
        Add a vertical span (rectangle) across the axes.

        Draw a vertical span (rectangle) from `xmin` to `xmax`.  With
        the default values of `ymin` = 0 and `ymax` = 1. This always
        spans the yrange, regardless of the ylim settings, even if you
        change them, e.g., with the :meth:`set_ylim` command.  That is,
        the vertical extent is in axes coords: 0=bottom, 0.5=middle,
        1.0=top but the x location is in data coordinates.

        Parameters
        ----------
        xmin : scalar
            Number indicating the first X-axis coordinate of the vertical
            span rectangle in data units.
        xmax : scalar
            Number indicating the second X-axis coordinate of the vertical
            span rectangle in data units.
        ymin : scalar, optional
            Number indicating the first Y-axis coordinate of the vertical
            span rectangle in relative Y-axis units (0-1). Default to 0.
        ymax : scalar, optional
            Number indicating the second Y-axis coordinate of the vertical
            span rectangle in relative Y-axis units (0-1). Default to 1.

        Returns
        -------
        rectangle : matplotlib.patches.Polygon
            Vertical span (rectangle) from (xmin, ymin) to (xmax, ymax).

        Other Parameters
        ----------------
        **kwargs
            Optional parameters are properties of the class
            matplotlib.patches.Polygon.

        See Also
        --------
        axhspan : Add a horizontal span across the axes.

        Examples
        --------
        Draw a vertical, green, translucent rectangle from x = 1.25 to
        x = 1.55 that spans the yrange of the axes.

        >>> axvspan(1.25, 1.55, facecolor='g', alpha=0.5)

    bar(x, height, width=0.8, bottom=None, *, align='center', data=None, **kwar
gs)
        Make a bar plot.

        The bars are positioned at *x* with the given *align*\ment. Their
        dimensions are given by *width* and *height*. The vertical baseline
        is *bottom* (default 0).
```

```
Each of *x*, *height*, *width*, and *bottom* may either be a scalar
applying to all bars, or it may be a sequence of length N providing a
separate value for each bar.

Parameters
----------
x : sequence of scalars
    The x coordinates of the bars. See also *align* for the
    alignment of the bars to the coordinates.

height : scalar or sequence of scalars
    The height(s) of the bars.

width : scalar or array-like, optional
    The width(s) of the bars (default: 0.8).

bottom : scalar or array-like, optional
    The y coordinate(s) of the bars bases (default: 0).

align : {'center', 'edge'}, optional, default: 'center'
    Alignment of the bars to the *x* coordinates:

    - 'center': Center the base on the *x* positions.
    - 'edge': Align the left edges of the bars with the *x* positions.

    To align the bars on the right edge pass a negative *width* and
    ``align='edge'``.

Returns
-------
container : `.BarContainer`
    Container with all the bars and optionally errorbars.

Other Parameters
----------------
color : scalar or array-like, optional
    The colors of the bar faces.

edgecolor : scalar or array-like, optional
    The colors of the bar edges.

linewidth : scalar or array-like, optional
    Width of the bar edge(s). If 0, don't draw edges.

tick_label : string or array-like, optional
    The tick labels of the bars.
    Default: None (Use default numeric labels.)

xerr, yerr : scalar or array-like of shape(N,) or shape(2,N), optional
    If not *None*, add horizontal / vertical errorbars to the bar tips.
    The values are +/- sizes relative to the data:

    - scalar: symmetric +/- values for all bars
    - shape(N,): symmetric +/- values for each bar
    - shape(2,N): Separate - and + values for each bar. First row
        contains the lower errors, the second row contains the
        upper errors.
```

```
            - *None*: No errorbar. (Default)

            See :doc:`/gallery/statistics/errorbar_features`
            for an example on the usage of ``xerr`` and ``yerr``.

        ecolor : scalar or array-like, optional, default: 'black'
            The line color of the errorbars.

        capsize : scalar, optional
           The length of the error bar caps in points.
           Default: None, which will take the value from
           :rc:`errorbar.capsize`.

        error_kw : dict, optional
            Dictionary of kwargs to be passed to the `~.Axes.errorbar`
            method. Values of *ecolor* or *capsize* defined here take
            precedence over the independent kwargs.

        log : bool, optional, default: False
            If *True*, set the y-axis to be log scale.

        orientation : {'vertical',  'horizontal'}, optional
            *This is for internal use only.* Please use `barh` for
            horizontal bar plots. Default: 'vertical'.

        See also
        --------
        barh: Plot a horizontal bar plot.

        Notes
        -----
        The optional arguments *color*, *edgecolor*, *linewidth*,
        *xerr*, and *yerr* can be either scalars or sequences of
        length equal to the number of bars.  This enables you to use
        bar as the basis for stacked bar charts, or candlestick plots.
        Detail: *xerr* and *yerr* are passed directly to
        :meth:`errorbar`, so they can also have shape 2xN for
        independent specification of lower and upper errors.

        Other optional kwargs:

          agg_filter: a filter function, which takes a (m, n, 3) float array an
    d a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: unknown
          capstyle: {'butt', 'round', 'projecting'}
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
    e]
          color: color
          contains: callable
          edgecolor: color or None or 'auto'
          facecolor: color or None
          figure: `.Figure`
          fill: bool
```

```
              gid: str
              hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
              in_layout: bool
              joinstyle: {'miter', 'round', 'bevel'}
              label: object
              linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
              linewidth: float or None for default
              path_effects: `.AbstractPathEffect`
              picker: None or bool or float or callable
              rasterized: bool or None
              sketch_params: (scale: float, length: float, randomness: float)
              snap: bool or None
              transform: `.Transform`
              url: str
              visible: bool
              zorder: float

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

                * All arguments with the following names: 'bottom', 'color', 'ecolo
r', 'edgecolor', 'height', 'left', 'linewidth', 'tick_label', 'width', 'x', 'xe
rr', 'y', 'yerr'.
                * All positional arguments.

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

        barbs(*args, data=None, **kw)
            Plot a 2-D field of barbs.

            Call signatures::

              barb(U, V, **kw)
              barb(U, V, C, **kw)
              barb(X, Y, U, V, **kw)
              barb(X, Y, U, V, C, **kw)

            Arguments:

              *X*, *Y*:
                The x and y coordinates of the barb locations
                (default is head of barb; see *pivot* kwarg)

              *U*, *V*:
                Give the x and y components of the barb shaft

              *C*:
                An optional array used to map colors to the barbs

            All arguments may be 1-D or 2-D arrays or sequences. If *X* and *Y*
            are absent, they will be generated as a uniform grid.  If *U* and *V*
```

```
are 2-D arrays but *X* and *Y* are 1-D, and if ``len(X)`` and ``len(Y)`
`
match the column and row dimensions of *U*, then *X* and *Y* will be
expanded with :func:`numpy.meshgrid`.

*U*, *V*, *C* may be masked arrays, but masked *X*, *Y* are not
supported at present.

Keyword arguments:

  *length*:
    Length of the barb in points; the other parts of the barb
    are scaled against this.
    Default is 7.

  *pivot*: [ 'tip' | 'middle' | float ]
    The part of the arrow that is at the grid point; the arrow rotates
    about this point, hence the name *pivot*.  Default is 'tip'. Can
    also be a number, which shifts the start of the barb that many
    points from the origin.

  *barbcolor*: [ color | color sequence ]
    Specifies the color all parts of the barb except any flags.  This
    parameter is analogous to the *edgecolor* parameter for polygons,
    which can be used instead. However this parameter will override
    facecolor.

  *flagcolor*: [ color | color sequence ]
    Specifies the color of any flags on the barb.  This parameter is
    analogous to the *facecolor* parameter for polygons, which can be
    used instead. However this parameter will override facecolor.  If
    this is not set (and *C* has not either) then *flagcolor* will be
    set to match *barbcolor* so that the barb has a uniform color. If
    *C* has been set, *flagcolor* has no effect.

  *sizes*:
    A dictionary of coefficients specifying the ratio of a given
    feature to the length of the barb. Only those values one wishes to
    override need to be included.  These features include:

        - 'spacing' - space between features (flags, full/half barbs)

        - 'height' - height (distance from shaft to top) of a flag or
          full barb

        - 'width' - width of a flag, twice the width of a full barb

        - 'emptybarb' - radius of the circle used for low magnitudes

  *fill_empty*:
    A flag on whether the empty barbs (circles) that are drawn should
    be filled with the flag color.  If they are not filled, they will
    be drawn such that no color is applied to the center.  Default is
    False

  *rounding*:
    A flag to indicate whether the vector magnitude should be rounded
```

```
    when allocating barb components.  If True, the magnitude is
    rounded to the nearest multiple of the half-barb increment.  If
    False, the magnitude is simply truncated to the next lowest
    multiple.  Default is True

  *barb_increments*:
    A dictionary of increments specifying values to associate with
    different parts of the barb. Only those values one wishes to
    override need to be included.

        - 'half' - half barbs (Default is 5)

        - 'full' - full barbs (Default is 10)

        - 'flag' - flags (default is 50)

  *flip_barb*:
    Either a single boolean flag or an array of booleans.  Single
    boolean indicates whether the lines and flags should point
    opposite to normal for all barbs.  An array (which should be the
    same size as the other data arrays) indicates whether to flip for
    each individual barb.  Normal behavior is for the barbs and lines
    to point right (comes from wind barbs having these features point
    towards low pressure in the Northern Hemisphere.)  Default is
    False

Barbs are traditionally used in meteorology as a way to plot the speed
and direction of wind observations, but can technically be used to
plot any two dimensional vector quantity.  As opposed to arrows, which
give vector magnitude by the length of the arrow, the barbs give more
quantitative information about the vector magnitude by putting slanted
lines or a triangle for various increments in magnitude, as show
schematically below::

 :     /\    \\
 :    /  \    \\
 :   /    \    \    \\
 :  /      \    \    \\
 : ------------------------------

.. note the double \\ at the end of each line to make the figure
.. render correctly

The largest increment is given by a triangle (or "flag"). After those
come full lines (barbs). The smallest increment is a half line.  There
is only, of course, ever at most 1 half line.  If the magnitude is
small and only needs a single half-line and no full lines or
triangles, the half-line is offset from the end of the barb so that it
can be easily distinguished from barbs with a single full line.  The
magnitude for the barb shown above would nominally be 65, using the
standard increments of 50, 10, and 5.

linewidths and edgecolors can be used to customize the barb.
Additional :class:`~matplotlib.collections.PolyCollection` keyword
arguments:

  agg_filter: a filter function, which takes a (m, n, 3) float array an
```

```
d a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float


        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    barh(y, width, height=0.8, left=None, *, align='center', **kwargs)
        Make a horizontal bar plot.
```

```
The bars are positioned at *y* with the given *align*\ment. Their
dimensions are given by *width* and *height*. The horizontal baseline
is *left* (default 0).

Each of *y*, *width*, *height*, and *left* may either be a scalar
applying to all bars, or it may be a sequence of length N providing a
separate value for each bar.

Parameters
----------
y : scalar or array-like
    The y coordinates of the bars. See also *align* for the
    alignment of the bars to the coordinates.

width : scalar or array-like
    The width(s) of the bars.

height : sequence of scalars, optional, default: 0.8
    The heights of the bars.

left : sequence of scalars
    The x coordinates of the left sides of the bars (default: 0).

align : {'center', 'edge'}, optional, default: 'center'
    Alignment of the base to the *y* coordinates*:

    - 'center': Center the bars on the *y* positions.
    - 'edge': Align the bottom edges of the bars with the *y*
      positions.

    To align the bars on the top edge pass a negative *height* and
    ``align='edge'``.

Returns
-------
container : `.BarContainer`
    Container with all the bars and optionally errorbars.

Other Parameters
----------------
color : scalar or array-like, optional
    The colors of the bar faces.

edgecolor : scalar or array-like, optional
    The colors of the bar edges.

linewidth : scalar or array-like, optional
    Width of the bar edge(s). If 0, don't draw edges.

tick_label : string or array-like, optional
    The tick labels of the bars.
    Default: None (Use default numeric labels.)

xerr, yerr : scalar or array-like of shape(N,) or shape(2,N), optional
    If not ``None``, add horizontal / vertical errorbars to the
    bar tips. The values are +/- sizes relative to the data:
```

```
                - scalar: symmetric +/- values for all bars
                - shape(N,): symmetric +/- values for each bar
                - shape(2,N): Separate - and + values for each bar. First row
                    contains the lower errors, the second row contains the
                    upper errors.
                - *None*: No errorbar. (default)

                See :doc:`/gallery/statistics/errorbar_features`
                for an example on the usage of ``xerr`` and ``yerr``.

            ecolor : scalar or array-like, optional, default: 'black'
                The line color of the errorbars.

            capsize : scalar, optional
               The length of the error bar caps in points.
               Default: None, which will take the value from
               :rc:`errorbar.capsize`.

            error_kw : dict, optional
                Dictionary of kwargs to be passed to the `~.Axes.errorbar`
                method. Values of *ecolor* or *capsize* defined here take
                precedence over the independent kwargs.

            log : bool, optional, default: False
                If ``True``, set the x-axis to be log scale.

            See also
            --------
            bar: Plot a vertical bar plot.

            Notes
            -----
            The optional arguments *color*, *edgecolor*, *linewidth*,
            *xerr*, and *yerr* can be either scalars or sequences of
            length equal to the number of bars.  This enables you to use
            bar as the basis for stacked bar charts, or candlestick plots.
            Detail: *xerr* and *yerr* are passed directly to
            :meth:`errorbar`, so they can also have shape 2xN for
            independent specification of lower and upper errors.

            Other optional kwargs:

              agg_filter: a filter function, which takes a (m, n, 3) float array an
        d a dpi value, and returns a (m, n, 3) array
              alpha: float or None
              animated: bool
              antialiased: unknown
              capstyle: {'butt', 'round', 'projecting'}
              clip_box: `.Bbox`
              clip_on: bool
              clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
        e]
              color: color
              contains: callable
              edgecolor: color or None or 'auto'
              facecolor: color or None
              figure: `.Figure`
```

```
              fill: bool
              gid: str
              hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
              in_layout: bool
              joinstyle: {'miter', 'round', 'bevel'}
              label: object
              linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
              linewidth: float or None for default
              path_effects: `.AbstractPathEffect`
              picker: None or bool or float or callable
              rasterized: bool or None
              sketch_params: (scale: float, length: float, randomness: float)
              snap: bool or None
              transform: `.Transform`
              url: str
              visible: bool
              zorder: float

    bone()
        Set the colormap to "bone".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    box(on=None)
        Turn the axes box on or off on the current axes.

        Parameters
        ----------
        on : bool or None
            The new `~matplotlib.axes.Axes` box state. If ``None``, toggle
            the state.

        See Also
        --------
        :meth:`matplotlib.axes.Axes.set_frame_on`
        :meth:`matplotlib.axes.Axes.get_frame_on`

    boxplot(x, notch=None, sym=None, vert=None, whis=None, positions=None, widt
hs=None, patch_artist=None, bootstrap=None, usermedians=None, conf_intervals=No
ne, meanline=None, showmeans=None, showcaps=None, showbox=None, showfliers=Non
e, boxprops=None, labels=None, flierprops=None, medianprops=None, meanprops=Non
e, capprops=None, whiskerprops=None, manage_xticks=True, autorange=False, zorde
r=None, *, data=None)
        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_xticks : bool, optional (True)
    If the function should adjust the xlim and xtick locations.

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

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    broken_barh(xranges, yrange, *, data=None, **kwargs)
```

```
        Plot a horizontal sequence of rectangles.

        A rectangle is drawn for each element of *xranges*. All rectangles
        have the same vertical position and size defined by *yrange*.

        This is a convenience function for instantiating a
        `.BrokenBarHCollection`, adding it to the axes and autoscaling the
        view.

        Parameters
        ----------
        xranges : sequence of tuples (*xmin*, *xwidth*)
            The x-positions and extends of the rectangles. For each tuple
            (*xmin*, *xwidth*) a rectangle is drawn from *xmin* to *xmin* +
            *xwidth*.
        yranges : (*ymin*, *ymax*)
            The y-position and extend for all the rectangles.

        Other Parameters
        ----------------
        **kwargs : :class:`.BrokenBarHCollection` properties

            Each *kwarg* can be either a single argument applying to all
            rectangles, e.g.::

                facecolors='black'

            or a sequence of arguments over which is cycled, e.g.::

                facecolors=('black', 'blue')

            would create interleaving black and blue rectangles.

            Supported keywords:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y and a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
```

```
                joinstyle: {'miter', 'round', 'bevel'}
                label: object
                linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
                linewidth: float or sequence of floats
                norm: `.Normalize`
                offset_position: {'screen', 'data'}
                offsets: float or sequence of floats
                path_effects: `.AbstractPathEffect`
                picker: None or bool or float or callable
                pickradius: unknown
                rasterized: bool or None
                sketch_params: (scale: float, length: float, randomness: float)
                snap: bool or None
                transform: `.Transform`
                url: str
                urls: List[str] or None
                visible: bool
                zorder: float

            Returns
            -------
            collection : A :class:`~.collections.BrokenBarHCollection`

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

            .. note::
                In addition to the above described arguments, this function can tak
    e a
                **data** keyword argument. If such a **data** argument is given, th
    e
                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``).

        cla()
            Clear the current axes.

        clabel(CS, *args, **kwargs)
            Label a contour plot.

            Call signature::

              clabel(cs, [levels,] **kwargs)

            Adds labels to line contours in *cs*, where *cs* is a
            :class:`~matplotlib.contour.ContourSet` object returned by
            ``contour()``.

            Parameters
            ----------
            cs : `.ContourSet`
```

```
    The ContourSet to label.

levels : array-like, optional
    A list of level values, that should be labeled. The list must be
    a subset of ``cs.levels``. If not given, all levels are labeled.

fontsize : string or float, optional
    Size in points or relative size e.g., 'smaller', 'x-large'.
    See `.Text.set_size` for accepted string values.

colors : color-spec, optional
    The label colors:

    - If *None*, the color of each label matches the color of
      the corresponding contour.

    - If one string color, e.g., *colors* = 'r' or *colors* =
      'red', all labels will be plotted in this color.

    - If a tuple of matplotlib color args (string, float, rgb, etc),
      different labels will be plotted in different colors in the order
      specified.

inline : bool, optional
    If ``True`` the underlying contour is removed where the label is
    placed. Default is ``True``.

inline_spacing : float, optional
    Space in pixels to leave on each side of label when
    placing inline. Defaults to 5.

    This spacing will be exact for labels at locations where the
    contour is straight, less so for labels on curved contours.

fmt : string or dict, optional
    A format string for the label. Default is '%1.3f'

    Alternatively, this can be a dictionary matching contour
    levels with arbitrary strings to use for each contour level
    (i.e., fmt[level]=string), or it can be any callable, such
    as a :class:`~matplotlib.ticker.Formatter` instance, that
    returns a string when called with a numeric contour level.

manual : bool or iterable, optional
    If ``True``, contour labels will be placed manually using
    mouse clicks. Click the first button near a contour to
    add a label, click the second button (or potentially both
    mouse buttons at once) to finish adding labels. The third
    button can be used to remove the last label added, but
    only if labels are not inline. Alternatively, the keyboard
    can be used to select label locations (enter to end label
    placement, delete or backspace act like the third mouse button,
    and any other key will select a label location).

    *manual* can also be an iterable object of x,y tuples.
    Contour labels will be created as if mouse is clicked at each
    x,y positions.
```

```
            rightside_up : bool, optional
                If ``True``, label rotations will always be plus
                or minus 90 degrees from level. Default is ``True``.

            use_clabeltext : bool, optional
                If ``True``, `.ClabelText` class (instead of `.Text`) is used to
                create labels. `ClabelText` recalculates rotation angles
                of texts during the drawing time, therefore this can be used if
                aspect of the axes changes. Default is ``False``.

            Returns
            -------
            labels
                A list of `.Text` instances for the labels.

        clf()
            Clear the current figure.

        clim(vmin=None, vmax=None)
            Set the color limits of the current image.

            To apply clim to all axes images do::

              clim(0, 0.5)

            If either *vmin* or *vmax* is None, the image min/max respectively
            will be used for color scaling.

            If you want to set the clim of multiple images,
            use, for example::

              for im in gca().get_images():
                  im.set_clim(0, 0.05)

        close(fig=None)
            Close a figure window.

            Parameters
            ----------
            fig : None or int or str or `.Figure`
                The figure to close. There are a number of ways to specify this:

                - *None*: the current figure
                - `.Figure`: the given `.Figure` instance
                - ``int``: a figure number
                - ``str``: a figure name
                - 'all': all figures

        cohere(x, y, NFFT=256, Fs=2, Fc=0, detrend=<function detrend_none at 0x0000
    01FEE7E93F28>, window=<function window_hanning at 0x000001FEE7E65F28>, noverlap
    =0, pad_to=None, sides='default', scale_by_freq=None, *, data=None, **kwargs)
            Plot the coherence between *x* and *y*.

            Plot the coherence between *x* and *y*.  Coherence is the
            normalized cross spectral density:
```

```
        .. math::

          C_{xy} = \frac{|P_{xy}|^2}{P_{xx}P_{yy}}

        Parameters
        ----------
        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*. To create window
            vectors see :func:`window_hanning`, :func:`window_none`,
            :func:`numpy.blackman`, :func:`numpy.hamming`,
            :func:`numpy.bartlett`, :func:`scipy.signal`,
            :func:`scipy.signal.get_window`, etc. The default is
            :func:`window_hanning`.  If a function is passed as the
            argument, it must take a data segment as an argument and
            return the windowed version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  This can be different from *NFFT*, which
            specifies the number of data points used.  While not increasing
            the actual resolution of the spectrum (the minimum distance between
            resolvable peaks), this can give more points in the plot,
            allowing for more detail. This corresponds to the *n* parameter
            in the call to fft(). The default is None, which sets *pad_to*
            equal to *NFFT*

        NFFT : int
            The number of data points used in each block for the FFT.
            A power 2 is most efficient.  The default value is 256.
            This should *NOT* be used to get zero padding, or the scaling of th
e
            result will be incorrect. Use *pad_to* for this instead.

        detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable
            The function applied to each segment before fft-ing,
            designed to remove the mean or linear trend.  Unlike in
            MATLAB, where the *detrend* parameter is a vector, in
            matplotlib is it a function.  The :mod:`~matplotlib.mlab`
            module defines :func:`~matplotlib.mlab.detrend_none`,
            :func:`~matplotlib.mlab.detrend_mean`, and
            :func:`~matplotlib.mlab.detrend_linear`, but you can use
            a custom function as well.  You can also use a string to choose
            one of the functions.  'default', 'constant', and 'mean' call
            :func:`~matplotlib.mlab.detrend_mean`.  'linear' calls
            :func:`~matplotlib.mlab.detrend_linear`.  'none' calls
            :func:`~matplotlib.mlab.detrend_none`.
```

```
        scale_by_freq : bool, optional
            Specifies whether the resulting density values should be scaled
            by the scaling frequency, which gives density in units of Hz^-1.
            This allows for integration over the returned frequency values.
            The default is True for MATLAB compatibility.

        noverlap : int
            The number of points of overlap between blocks.  The
            default value is 0 (no overlap).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.


        Returns
        -------
        Cxy : 1-D array
            The coherence vector.

        freqs : 1-D array
            The frequencies for the elements in *Cxy*.

        Other Parameters
        ----------------
        **kwargs :
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

            References
            ----------
            Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
            John Wiley & Sons (1986)

            .. note::
                In addition to the above described arguments, this function can tak
    e a
                **data** keyword argument. If such a **data** argument is given, th
    e
                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``).

        colorbar(mappable=None, cax=None, ax=None, **kw)
            Add a colorbar to a plot.

            Function signatures for the :mod:`~matplotlib.pyplot` interface; all
            but the first are also method signatures for the
            :meth:`~matplotlib.figure.Figure.colorbar` method::

              colorbar(**kwargs)
              colorbar(mappable, **kwargs)
              colorbar(mappable, cax=cax, **kwargs)
              colorbar(mappable, ax=ax, **kwargs)

            Parameters
            ----------
            mappable :
                The :class:`~matplotlib.image.Image`,
                :class:`~matplotlib.contour.ContourSet`, etc. to
                which the colorbar applies; this argument is mandatory for the Figu
    re
```

```
            :meth:`~matplotlib.figure.Figure.colorbar` method but optional for
 the
            pyplot :func:`~matplotlib.pyplot.colorbar` function, which sets the
            default to the current image.

        cax : :class:`~matplotlib.axes.Axes` object, optional
            Axes into which the colorbar will be drawn.

        ax : :class:`~matplotlib.axes.Axes`, list of Axes, optional
            Parent axes from which space for a new colorbar axes will be stole
n.
            If a list of axes is given they will all be resized to make room fo
r the
            colorbar axes.

        use_gridspec : bool, optional
            If *cax* is ``None``, a new *cax* is created as an instance of
            Axes. If *ax* is an instance of Subplot and *use_gridspec* is ``Tru
e``,
            *cax* is created as an instance of Subplot using the
            grid_spec module.


        Returns
        -------
        colorbar : `~matplotlib.colorbar.Colorbar`
            See also its base class, `~matplotlib.colorbar.ColorbarBase`.  Use
            `~.ColorbarBase.set_label` to label the colorbar.

        Notes
        -----
        Additional keyword arguments are of two kinds:

          axes properties:


            ============= ====================================================
            Property      Description
            ============= ====================================================
            *orientation* vertical or horizontal
            *fraction*    0.15; fraction of original axes to use for colorbar
            *pad*         0.05 if vertical, 0.15 if horizontal; fraction
                          of original axes between colorbar and new image axes
            *shrink*      1.0; fraction by which to multiply the size of the co
lorbar
            *aspect*      20; ratio of long to short dimensions
            *anchor*      (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal;
                          the anchor point of the colorbar axes
            *panchor*     (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal;
                          the anchor point of the colorbar parent axes. If
                          False, the parent axes' anchor will be unchanged
            ============= ====================================================


          colorbar properties:
```

```
            ============  ==================================================
            Property      Description
            ============  ==================================================
            *extend*      [ 'neither' | 'both' | 'min' | 'max' ]
                          If not 'neither', make pointed end(s) for out-of-
                          range values.  These are set for a given colormap
                          using the colormap set_under and set_over methods.
            *extendfrac*  [ *None* | 'auto' | length | lengths ]
                          If set to *None*, both the minimum and maximum
                          triangular colorbar extensions with have a length of
                          5% of the interior colorbar length (this is the
                          default setting). If set to 'auto', makes the
                          triangular colorbar extensions the same lengths as
                          the interior boxes (when *spacing* is set to
                          'uniform') or the same lengths as the respective
                          adjacent interior boxes (when *spacing* is set to
                          'proportional'). If a scalar, indicates the length
                          of both the minimum and maximum triangular colorbar
                          extensions as a fraction of the interior colorbar
                          length. A two-element sequence of fractions may also
                          be given, indicating the lengths of the minimum and
                          maximum colorbar extensions respectively as a
                          fraction of the interior colorbar length.
            *extendrect*  bool
                          If *False* the minimum and maximum colorbar extension
            s
                          will be triangular (the default). If *True* the
                          extensions will be rectangular.
            *spacing*     [ 'uniform' | 'proportional' ]
                          Uniform spacing gives each discrete color the same
                          space; proportional makes the space proportional to
                          the data interval.
            *ticks*       [ None | list of ticks | Locator object ]
                          If None, ticks are determined automatically from the
                          input.
            *format*      [ None | format string | Formatter object ]
                          If None, the
                          :class:`~matplotlib.ticker.ScalarFormatter` is used.
                          If a format string is given, e.g., '%.3f', that is
                          used. An alternative
                          :class:`~matplotlib.ticker.Formatter` object may be
                          given instead.
            *drawedges*   bool
                          Whether to draw lines at color boundaries.
            ============  ==================================================

            The following will probably be useful only in the context of
            indexed colors (that is, when the mappable has norm=NoNorm()),
            or other unusual circumstances.

            ============   ==================================================
            Property       Description
            ============   ==================================================
            *boundaries*   None or a sequence
            *values*       None or a sequence which must be of length 1 less
                           than the sequence of *boundaries*. For each region
                           delimited by adjacent entries in *boundaries*, the
```

```
                                color mapped to the corresponding value in values
                                will be used.
                    ============   ==================================================


        If *mappable* is a :class:`~matplotlib.contours.ContourSet`, its *exten
d*
        kwarg is included automatically.

        The *shrink* kwarg provides a simple way to scale the colorbar with res
pect
        to the axes. Note that if *cax* is specified it determines the size of
 the
        colorbar and *shrink* and *aspect* kwargs are ignored.

        For more precise control, you can manually specify the positions of
        the axes objects in which the mappable and the colorbar are drawn.  In
        this case, do not use any of the axes properties kwargs.

        It is known that some vector graphics viewer (svg and pdf) renders whit
e gaps
        between segments of the colorbar. This is due to bugs in the viewers no
t
        matplotlib. As a workaround the colorbar can be rendered with overlappi
ng
        segments::

            cbar = colorbar()
            cbar.solids.set_edgecolor("face")
            draw()

        However this has negative consequences in other circumstances. Particul
arly
        with semi transparent images (alpha < 1) and colorbar extensions and is
not
        enabled by default see (issue #1188).

    colormaps()
        Matplotlib provides a number of colormaps, and others can be added usin
g
        :func:`~matplotlib.cm.register_cmap`.  This function documents the buil
t-in
        colormaps, and will also return a list of all registered colormaps if c
alled.

        You can set the colormap for an image, pcolor, scatter, etc,
        using a keyword argument::

          imshow(X, cmap=cm.hot)

        or using the :func:`set_cmap` function::

          imshow(X)
          pyplot.set_cmap('hot')
          pyplot.set_cmap('jet')
```

```
        In interactive mode, :func:`set_cmap` will update the colormap post-ho
c,
        allowing you to see which one works best for your data.

        All built-in colormaps can be reversed by appending ``_r``: For instanc
e,
        ``gray_r`` is the reverse of ``gray``.

        There are several common color schemes used in visualization:

        Sequential schemes
          for unipolar data that progresses from low to high
        Diverging schemes
          for bipolar data that emphasizes positive or negative deviations from
a
          central value
        Cyclic schemes
          for plotting values that wrap around at the endpoints, such as phase
          angle, wind direction, or time of day
        Qualitative schemes
          for nominal data that has no inherent ordering, where color is used
          only to distinguish categories

        Matplotlib ships with 4 perceptually uniform color maps which are
        the recommended color maps for sequential data:

          =========   ====================================================
          Colormap    Description
          =========   ====================================================
          inferno     perceptually uniform shades of black-red-yellow
          magma       perceptually uniform shades of black-red-white
          plasma      perceptually uniform shades of blue-red-yellow
          viridis     perceptually uniform shades of blue-green-yellow
          =========   ====================================================

        The following colormaps are based on the `ColorBrewer
        <http://colorbrewer2.org>`_ color specifications and designs developed
 by
        Cynthia Brewer:

        ColorBrewer Diverging (luminance is highest at the midpoint, and
        decreases towards differently-colored endpoints):

          ========  ===================================
          Colormap  Description
          ========  ===================================
          BrBG      brown, white, blue-green
          PiYG      pink, white, yellow-green
          PRGn      purple, white, green
          PuOr      orange, white, purple
          RdBu      red, white, blue
          RdGy      red, white, gray
          RdYlBu    red, yellow, blue
          RdYlGn    red, yellow, green
          Spectral  red, orange, yellow, green, blue
          ========  ===================================
```

```
ColorBrewer Sequential (luminance decreases monotonically):

  ========  ====================================
  Colormap  Description
  ========  ====================================
  Blues     white to dark blue
  BuGn      white, light blue, dark green
  BuPu      white, light blue, dark purple
  GnBu      white, light green, dark blue
  Greens    white to dark green
  Greys     white to black (not linear)
  Oranges   white, orange, dark brown
  OrRd      white, orange, dark red
  PuBu      white, light purple, dark blue
  PuBuGn    white, light purple, dark green
  PuRd      white, light purple, dark red
  Purples   white to dark purple
  RdPu      white, pink, dark purple
  Reds      white to dark red
  YlGn      light yellow, dark green
  YlGnBu    light yellow, light green, dark blue
  YlOrBr    light yellow, orange, dark brown
  YlOrRd    light yellow, orange, dark red
  ========  ====================================

ColorBrewer Qualitative:

(For plotting nominal data, :class:`ListedColormap` is used,
not :class:`LinearSegmentedColormap`.  Different sets of colors are
recommended for different numbers of categories.)

* Accent
* Dark2
* Paired
* Pastel1
* Pastel2
* Set1
* Set2
* Set3

A set of colormaps derived from those of the same name provided
with Matlab are also included:

  =========   ====================================================
  Colormap    Description
  =========   ====================================================
  autumn      sequential linearly-increasing shades of red-orange-yello
w
  bone        sequential increasing black-white color map with
              a tinge of blue, to emulate X-ray film
  cool        linearly-decreasing shades of cyan-magenta
  copper      sequential increasing shades of black-copper
  flag        repetitive red-white-blue-black pattern (not cyclic at
              endpoints)
  gray        sequential linearly-increasing black-to-white
              grayscale
  hot         sequential black-red-yellow-white, to emulate blackbody
```

```
                radiation from an object at increasing temperatures
  jet           a spectral map with dark endpoints, blue-cyan-yellow-red;
                based on a fluid-jet simulation by NCSA [#]_
  pink          sequential increasing pastel black-pink-white, meant
                for sepia tone colorization of photographs
  prism         repetitive red-yellow-green-blue-purple-...-green pattern
                (not cyclic at endpoints)
  spring        linearly-increasing shades of magenta-yellow
  summer        sequential linearly-increasing shades of green-yellow
  winter        linearly-increasing shades of blue-green
  =========   ====================================================

A set of palettes from the `Yorick scientific visualisation
package <https://dhmunro.github.io/yorick-doc/>`_, an evolution of
the GIST package, both by David H. Munro are included:

  ============  ============================================================
  Colormap      Description
  ============  ============================================================
  gist_earth    mapmaker's colors from dark blue deep ocean to green
                lowlands to brown highlands to white mountains
  gist_heat     sequential increasing black-red-orange-white, to emulat
e
                blackbody radiation from an iron bar as it grows hotter
  gist_ncar     pseudo-spectral black-blue-green-yellow-red-purple-whit
e
                colormap from National Center for Atmospheric
                Research [#]_
  gist_rainbow  runs through the colors in spectral order from red to
                violet at full saturation (like *hsv* but not cyclic)
  gist_stern    "Stern special" color table from Interactive Data
                Language software
  ============  ============================================================

A set of cyclic color maps:

  ================  ================================================
======
  Colormap          Description
  ================  ================================================
======
  hsv               red-yellow-green-cyan-blue-magenta-red, formed by c
hanging
                    the hue component in the HSV color space
  twilight          perceptually uniform shades of white-blue-black-red
-white
  twilight_shifted  perceptually uniform shades of black-blue-white-red
-black
  ================  ================================================
======


A set of miscellaneous schemes:

  ============ ============================================================
  Colormap      Description
  ============ ============================================================
```

```
    afmhot        sequential black-orange-yellow-white blackbody
                  spectrum, commonly used in atomic force microscopy
    brg           blue-red-green
    bwr           diverging blue-white-red
    coolwarm      diverging blue-gray-red, meant to avoid issues with 3D
                  shading, color blindness, and ordering of colors [#]_
    CMRmap        "Default colormaps on color images often reproduce to
                  confusing grayscale images. The proposed colormap
                  maintains an aesthetically pleasing color image that
                  automatically reproduces to a monotonic grayscale with
                  discrete, quantifiable saturation levels." [#]_
    cubehelix     Unlike most other color schemes cubehelix was designed
                  by D.A. Green to be monotonically increasing in terms
                  of perceived brightness. Also, when printed on a black
                  and white postscript printer, the scheme results in a
                  greyscale with monotonically increasing brightness.
                  This color scheme is named cubehelix because the r,g,b
                  values produced can be visualised as a squashed helix
                  around the diagonal in the r,g,b color cube.
    gnuplot       gnuplot's traditional pm3d scheme
                  (black-blue-red-yellow)
    gnuplot2      sequential color printable as gray
                  (black-blue-violet-yellow-white)
    ocean         green-blue-white
    rainbow       spectral purple-blue-green-yellow-orange-red colormap
                  with diverging luminance
    seismic       diverging blue-white-red
    nipy_spectral black-purple-blue-green-yellow-red-white spectrum,
                  originally from the Neuroimaging in Python project
    terrain       mapmaker's colors, blue-green-yellow-brown-white,
                  originally from IGOR Pro
    ============= ============================================================

  The following colormaps are redundant and may be removed in future
  versions.  It's recommended to use the names in the descriptions
  instead, which produce identical output:

    =========  ============================================================
    Colormap   Description
    =========  ============================================================
    gist_gray  identical to *gray*
    gist_yarg  identical to *gray_r*
    binary     identical to *gray_r*
    =========  ============================================================

  .. rubric:: Footnotes

  .. [#] Rainbow colormaps, ``jet`` in particular, are considered a poor
    choice for scientific visualization by many researchers: `Rainbow Col
or
    Map (Still) Considered Harmful
    <http://ieeexplore.ieee.org/document/4118486/?arnumber=4118486>`_

  .. [#] Resembles "BkBlAqGrYeOrReViWh200" from NCAR Command
    Language. See `Color Table Gallery
    <https://www.ncl.ucar.edu/Document/Graphics/color_table_gallery.shtml
>`_
```

```
            .. [#] See `Diverging Color Maps for Scientific Visualization
              <http://www.kennethmoreland.com/color-maps/>`_ by Kenneth Moreland.

            .. [#] See `A Color Map for Effective Black-and-White Rendering of
              Color-Scale Images
              <https://www.mathworks.com/matlabcentral/fileexchange/2662-cmrmap-m>`
_
              by Carey Rappaport

        connect(s, func)
            Connect event with string *s* to *func*.  The signature of *func* is::

              def func(event)

            where event is a :class:`matplotlib.backend_bases.Event`.  The
            following events are recognized

            - 'button_press_event'
            - 'button_release_event'
            - 'draw_event'
            - 'key_press_event'
            - 'key_release_event'
            - 'motion_notify_event'
            - 'pick_event'
            - 'resize_event'
            - 'scroll_event'
            - 'figure_enter_event',
            - 'figure_leave_event',
            - 'axes_enter_event',
            - 'axes_leave_event'
            - 'close_event'

            For the location events (button and key press/release), if the
            mouse is over the axes, the variable ``event.inaxes`` will be
            set to the :class:`~matplotlib.axes.Axes` the event occurs is
            over, and additionally, the variables ``event.xdata`` and
            ``event.ydata`` will be defined.  This is the mouse location
            in data coords.  See
            :class:`~matplotlib.backend_bases.KeyEvent` and
            :class:`~matplotlib.backend_bases.MouseEvent` for more info.

            Return value is a connection id that can be used with
            :meth:`~matplotlib.backend_bases.Event.mpl_disconnect`.

            Examples
            --------
            Usage::

                def on_press(event):
                    print('you pressed', event.button, event.xdata, event.ydata)

                cid = canvas.mpl_connect('button_press_event', on_press)

        contour(*args, data=None, **kwargs)
            Plot contours.
```

```
Call signature::

    contour([X, Y,] Z, [levels], **kwargs)

:func:`~matplotlib.pyplot.contour` and
:func:`~matplotlib.pyplot.contourf` draw contour lines and
filled contours, respectively.  Except as noted, function
signatures and return values are the same for both versions.


Parameters
----------
X, Y : array-like, optional
    The coordinates of the values in *Z*.

    *X* and *Y* must both be 2-D with the same shape as *Z* (e.g.
    created via :func:`numpy.meshgrid`), or they must both be 1-D such
    that ``len(X) == M`` is the number of columns in *Z* and
    ``len(Y) == N`` is the number of rows in *Z*.

    If not given, they are assumed to be integer indices, i.e.
    ``X = range(M)``, ``Y = range(N)``.

Z : array-like(N, M)
    The height values over which the contour is drawn.

levels : int or array-like, optional
    Determines the number and positions of the contour lines / regions.

    If an int *n*, use *n* data intervals; i.e. draw *n+1* contour
    lines. The level heights are automatically chosen.

    If array-like, draw contour lines at the specified levels.
    The values must be in increasing order.

Returns
-------
c : `~.contour.QuadContourSet`

Other Parameters
----------------
corner_mask : bool, optional
    Enable/disable corner masking, which only has an effect if *Z* is
    a masked array.  If ``False``, any quad touching a masked point is
    masked out.  If ``True``, only the triangular corners of quads
    nearest those points are always masked out, other triangular
    corners comprising three unmasked points are contoured as usual.

    Defaults to ``rcParams['contour.corner_mask']``, which defaults to
    ``True``.

colors : color string or sequence of colors, optional
    The colors of the levels, i.e. the lines for `.contour` and the
    areas for `.contourf`.

    The sequence is cycled for the levels in ascending order. If the
    sequence is shorter than the number of levels, it's repeated.
```

```
    As a shortcut, single color strings may be used in place of
    one-element lists, i.e. ``'red'`` instead of ``['red']`` to color
    all levels with the same color. This shortcut does only work for
    color strings, not for other ways of specifying colors.

    By default (value *None*), the colormap specified by *cmap*
    will be used.

alpha : float, optional
    The alpha blending value, between 0 (transparent) and 1 (opaque).

cmap : str or `.Colormap`, optional
    A `.Colormap` instance or registered colormap name. The colormap
    maps the level values to colors.
    Defaults to :rc:`image.cmap`.

    If given, *colors* take precedence over *cmap*.

norm : `~matplotlib.colors.Normalize`, optional
    If a colormap is used, the `.Normalize` instance scales the level
    values to the canonical colormap range [0, 1] for mapping to
    colors. If not given, the default linear scaling is used.

vmin, vmax : float, optional
    If not *None*, either or both of these values will be supplied to
    the `.Normalize` instance, overriding the default color scaling
    based on *levels*.

origin : {*None*, 'upper', 'lower', 'image'}, optional
    Determines the orientation and exact position of *Z* by specifying
    the position of ``Z[0, 0]``.  This is only relevant, if *X*, *Y*
    are not given.

    - *None*: ``Z[0, 0]`` is at X=0, Y=0 in the lower left corner.
    - 'lower': ``Z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
    - 'upper': ``Z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left
      corner.
    - 'image': Use the value from :rc:`image.origin`. Note: The value
      *None* in the rcParam is currently handled as 'lower'.

extent : (x0, x1, y0, y1), optional
    If *origin* is not *None*, then *extent* is interpreted as
    in :func:`matplotlib.pyplot.imshow`: it gives the outer
    pixel boundaries. In this case, the position of Z[0,0]
    is the center of the pixel, not a corner. If *origin* is
    *None*, then (*x0*, *y0*) is the position of Z[0,0], and
    (*x1*, *y1*) is the position of Z[-1,-1].

    This keyword is not active if *X* and *Y* are specified in
    the call to contour.

locator : ticker.Locator subclass, optional
    The locator is used to determine the contour levels if they
    are not given explicitly via *levels*.
    Defaults to `~.ticker.MaxNLocator`.
```

```
            extend : {'neither', 'both', 'min', 'max'}, optional, default: 'neithe
r'
                Determines the ``contourf``-coloring of values that are outside the
                *levels* range.

                If 'neither', values outside the *levels* range are not colored.
                If 'min', 'max' or 'both', color the values below, above or below
                and above the *levels* range.

                Values below ``min(levels)`` and above ``max(levels)`` are mapped
                to the under/over values of the `.Colormap`. Note, that most
                colormaps do not have dedicated colors for these by default, so
                that the over and under values are the edge values of the colormap.
                You may want to set these values explicitly using
                `.Colormap.set_under` and `.Colormap.set_over`.

                .. note::

                    An exising `.QuadContourSet` does not get notified if
                    properties of its colormap are changed. Therefore, an explicit
                    call `.QuadContourSet.changed()` is needed after modifying the
                    colormap. The explicit call can be left out, if a colorbar is
                    assigned to the `.QuadContourSet` because it interally calls
                    `.QuadContourSet.changed()`.

                Example::

                    x = np.arange(1, 10)
                    y = x.reshape(-1, 1)
                    h = x * y

                    cs = plt.contourf(h, levels=[10, 30, 50],
                        colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
                    cs.cmap.set_over('red')
                    cs.cmap.set_under('blue')
                    cs.changed()

            xunits, yunits : registered units, optional
                Override axis units by specifying an instance of a
                :class:`matplotlib.units.ConversionInterface`.

            antialiased : bool, optinal
                Enable antialiasing, overriding the defaults.  For
                filled contours, the default is *True*.  For line contours,
                it is taken from :rc:`lines.antialiased`.

            Nchunk : int >= 0, optional
                If 0, no subdivision of the domain.  Specify a positive integer to
                divide the domain into subdomains of *nchunk* by *nchunk* quads.
                Chunking reduces the maximum length of polygons generated by the
                contouring algorithm which reduces the rendering workload passed
                on to the backend and also requires slightly less RAM.  It can
                however introduce rendering artifacts at chunk boundaries depending
                on the backend, the *antialiased* flag and value of *alpha*.

            linewidths : float or sequence of float, optional
                *Only applies to* `.contour`.
```

```
        The line width of the contour lines.

        If a number, all levels will be plotted with this linewidth.

        If a sequence, the levels in ascending order will be plotted with
        the linewidths in the order specified.

        Defaults to :rc:`lines.linewidth`.

    linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
        *Only applies to* `.contour`.

        If *linestyles* is *None*, the default is 'solid' unless the lines
        are monochrome.  In that case, negative contours will take their
        linestyle from :rc:`contour.negative_linestyle` setting.

        *linestyles* can also be an iterable of the above strings
        specifying a set of linestyles to be used. If this
        iterable is shorter than the number of contour levels
        it will be repeated as necessary.

    hatches : List[str], optional
        *Only applies to* `.contourf`.

        A list of cross hatch patterns to use on the filled areas.
        If None, no hatching will be added to the contour.
        Hatching is supported in the PostScript, PDF, SVG and Agg
        backends only.


    Notes
    -----
    1. :func:`~matplotlib.pyplot.contourf` differs from the MATLAB
       version in that it does not draw the polygon edges.
       To draw edges, add line contours with
       calls to :func:`~matplotlib.pyplot.contour`.

    2. contourf fills intervals that are closed at the top; that
       is, for boundaries *z1* and *z2*, the filled region is::

          z1 < Z <= z2

       There is one exception: if the lowest boundary coincides with
       the minimum value of the *Z* array, then that minimum value
       will be included in the lowest interval.

contourf(*args, data=None, **kwargs)
    Plot contours.

    Call signature::

        contour([X, Y,] Z, [levels], **kwargs)

    :func:`~matplotlib.pyplot.contour` and
    :func:`~matplotlib.pyplot.contourf` draw contour lines and
    filled contours, respectively.  Except as noted, function
```

```
signatures and return values are the same for both versions.


Parameters
----------
X, Y : array-like, optional
    The coordinates of the values in *Z*.

    *X* and *Y* must both be 2-D with the same shape as *Z* (e.g.
    created via :func:`numpy.meshgrid`), or they must both be 1-D such
    that ``len(X) == M`` is the number of columns in *Z* and
    ``len(Y) == N`` is the number of rows in *Z*.

    If not given, they are assumed to be integer indices, i.e.
    ``X = range(M)``, ``Y = range(N)``.

Z : array-like(N, M)
    The height values over which the contour is drawn.

levels : int or array-like, optional
    Determines the number and positions of the contour lines / regions.

    If an int *n*, use *n* data intervals; i.e. draw *n+1* contour
    lines. The level heights are automatically chosen.

    If array-like, draw contour lines at the specified levels.
    The values must be in increasing order.

Returns
-------
c : `~.contour.QuadContourSet`

Other Parameters
----------------
corner_mask : bool, optional
    Enable/disable corner masking, which only has an effect if *Z* is
    a masked array.  If ``False``, any quad touching a masked point is
    masked out.  If ``True``, only the triangular corners of quads
    nearest those points are always masked out, other triangular
    corners comprising three unmasked points are contoured as usual.

    Defaults to ``rcParams['contour.corner_mask']``, which defaults to
    ``True``.

colors : color string or sequence of colors, optional
    The colors of the levels, i.e. the lines for `.contour` and the
    areas for `.contourf`.

    The sequence is cycled for the levels in ascending order. If the
    sequence is shorter than the number of levels, it's repeated.

    As a shortcut, single color strings may be used in place of
    one-element lists, i.e. ``'red'`` instead of ``['red']`` to color
    all levels with the same color. This shortcut does only work for
    color strings, not for other ways of specifying colors.

    By default (value *None*), the colormap specified by *cmap*
```

```
        will be used.

    alpha : float, optional
        The alpha blending value, between 0 (transparent) and 1 (opaque).

    cmap : str or `.Colormap`, optional
        A `.Colormap` instance or registered colormap name. The colormap
        maps the level values to colors.
        Defaults to :rc:`image.cmap`.

        If given, *colors* take precedence over *cmap*.

    norm : `~matplotlib.colors.Normalize`, optional
        If a colormap is used, the `.Normalize` instance scales the level
        values to the canonical colormap range [0, 1] for mapping to
        colors. If not given, the default linear scaling is used.

    vmin, vmax : float, optional
        If not *None*, either or both of these values will be supplied to
        the `.Normalize` instance, overriding the default color scaling
        based on *levels*.

    origin : {*None*, 'upper', 'lower', 'image'}, optional
        Determines the orientation and exact position of *Z* by specifying
        the position of ``Z[0, 0]``.  This is only relevant, if *X*, *Y*
        are not given.

        - *None*: ``Z[0, 0]`` is at X=0, Y=0 in the lower left corner.
        - 'lower': ``Z[0, 0]`` is at X=0.5, Y=0.5 in the lower left corner.
        - 'upper': ``Z[0, 0]`` is at X=N+0.5, Y=0.5 in the upper left
          corner.
        - 'image': Use the value from :rc:`image.origin`. Note: The value
          *None* in the rcParam is currently handled as 'lower'.

    extent : (x0, x1, y0, y1), optional
        If *origin* is not *None*, then *extent* is interpreted as
        in :func:`matplotlib.pyplot.imshow`: it gives the outer
        pixel boundaries. In this case, the position of Z[0,0]
        is the center of the pixel, not a corner. If *origin* is
        *None*, then (*x0*, *y0*) is the position of Z[0,0], and
        (*x1*, *y1*) is the position of Z[-1,-1].

        This keyword is not active if *X* and *Y* are specified in
        the call to contour.

    locator : ticker.Locator subclass, optional
        The locator is used to determine the contour levels if they
        are not given explicitly via *levels*.
        Defaults to `~.ticker.MaxNLocator`.

    extend : {'neither', 'both', 'min', 'max'}, optional, default: 'neithe
r'
        Determines the ``contourf``-coloring of values that are outside the
        *levels* range.

        If 'neither', values outside the *levels* range are not colored.
        If 'min', 'max' or 'both', color the values below, above or below
```

```
    and above the *levels* range.

    Values below ``min(levels)`` and above ``max(levels)`` are mapped
    to the under/over values of the `.Colormap`. Note, that most
    colormaps do not have dedicated colors for these by default, so
    that the over and under values are the edge values of the colormap.
    You may want to set these values explicitly using
    `.Colormap.set_under` and `.Colormap.set_over`.

    .. note::

        An exising `.QuadContourSet` does not get notified if
        properties of its colormap are changed. Therefore, an explicit
        call `.QuadContourSet.changed()` is needed after modifying the
        colormap. The explicit call can be left out, if a colorbar is
        assigned to the `.QuadContourSet` because it interally calls
        `.QuadContourSet.changed()`.

    Example::

        x = np.arange(1, 10)
        y = x.reshape(-1, 1)
        h = x * y

        cs = plt.contourf(h, levels=[10, 30, 50],
            colors=['#808080', '#A0A0A0', '#C0C0C0'], extend='both')
        cs.cmap.set_over('red')
        cs.cmap.set_under('blue')
        cs.changed()

xunits, yunits : registered units, optional
    Override axis units by specifying an instance of a
    :class:`matplotlib.units.ConversionInterface`.

antialiased : bool, optinal
    Enable antialiasing, overriding the defaults.  For
    filled contours, the default is *True*.  For line contours,
    it is taken from :rc:`lines.antialiased`.

Nchunk : int >= 0, optional
    If 0, no subdivision of the domain.  Specify a positive integer to
    divide the domain into subdomains of *nchunk* by *nchunk* quads.
    Chunking reduces the maximum length of polygons generated by the
    contouring algorithm which reduces the rendering workload passed
    on to the backend and also requires slightly less RAM.  It can
    however introduce rendering artifacts at chunk boundaries depending
    on the backend, the *antialiased* flag and value of *alpha*.

linewidths : float or sequence of float, optional
    *Only applies to* `.contour`.

    The line width of the contour lines.

    If a number, all levels will be plotted with this linewidth.

    If a sequence, the levels in ascending order will be plotted with
    the linewidths in the order specified.
```

```
            Defaults to :rc:`lines.linewidth`.

        linestyles : {*None*, 'solid', 'dashed', 'dashdot', 'dotted'}, optional
            *Only applies to* `.contour`.

            If *linestyles* is *None*, the default is 'solid' unless the lines
            are monochrome.  In that case, negative contours will take their
            linestyle from :rc:`contour.negative_linestyle` setting.

            *linestyles* can also be an iterable of the above strings
            specifying a set of linestyles to be used. If this
            iterable is shorter than the number of contour levels
            it will be repeated as necessary.

        hatches : List[str], optional
            *Only applies to* `.contourf`.

            A list of cross hatch patterns to use on the filled areas.
            If None, no hatching will be added to the contour.
            Hatching is supported in the PostScript, PDF, SVG and Agg
            backends only.


        Notes
        -----
        1. :func:`~matplotlib.pyplot.contourf` differs from the MATLAB
           version in that it does not draw the polygon edges.
           To draw edges, add line contours with
           calls to :func:`~matplotlib.pyplot.contour`.

        2. contourf fills intervals that are closed at the top; that
           is, for boundaries *z1* and *z2*, the filled region is::

              z1 < Z <= z2

           There is one exception: if the lowest boundary coincides with
           the minimum value of the *Z* array, then that minimum value
           will be included in the lowest interval.

    cool()
        Set the colormap to "cool".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    copper()
        Set the colormap to "copper".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    csd(x, y, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=
None, pad_to=None, sides=None, scale_by_freq=None, return_line=None, *, data=No
ne, **kwargs)
```

```
Plot the cross-spectral density.

Call signature::

  csd(x, y, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
      window=mlab.window_hanning, noverlap=0, pad_to=None,
      sides='default', scale_by_freq=None, return_line=None, **kwargs)

The cross spectral density :math:`P_{xy}` by Welch's average
periodogram method.  The vectors *x* and *y* are divided into
*NFFT* length segments.  Each segment is detrended by function
*detrend* and windowed by function *window*.  *noverlap* gives
the length of the overlap between segments.  The product of
the direct FFTs of *x* and *y* are averaged over each segment
to compute :math:`P_{xy}`, with a scaling to correct for power
loss due to windowing.

If len(*x*) < *NFFT* or len(*y*) < *NFFT*, they will be zero
padded to *NFFT*.

Parameters
----------
x, y : 1-D arrays or sequences
    Arrays or sequences containing the data.

Fs : scalar
    The sampling frequency (samples per time unit).  It is used
    to calculate the Fourier frequencies, freqs, in cycles per time
    unit. The default value is 2.

window : callable or ndarray
    A function or a vector of length *NFFT*. To create window
    vectors see :func:`window_hanning`, :func:`window_none`,
    :func:`numpy.blackman`, :func:`numpy.hamming`,
    :func:`numpy.bartlett`, :func:`scipy.signal`,
    :func:`scipy.signal.get_window`, etc. The default is
    :func:`window_hanning`.  If a function is passed as the
    argument, it must take a data segment as an argument and
    return the windowed version of the segment.

sides : {'default', 'onesided', 'twosided'}
    Specifies which sides of the spectrum to return.  Default gives the
    default behavior, which returns one-sided for real data and both
    for complex data.  'onesided' forces the return of a one-sided
    spectrum, while 'twosided' forces two-sided.

pad_to : int
    The number of points to which the data segment is padded when
    performing the FFT.  This can be different from *NFFT*, which
    specifies the number of data points used.  While not increasing
    the actual resolution of the spectrum (the minimum distance between
    resolvable peaks), this can give more points in the plot,
    allowing for more detail. This corresponds to the *n* parameter
    in the call to fft(). The default is None, which sets *pad_to*
    equal to *NFFT*

NFFT : int
```

```
            The number of data points used in each block for the FFT.
            A power 2 is most efficient.  The default value is 256.
            This should *NOT* be used to get zero padding, or the scaling of th
e
            result will be incorrect. Use *pad_to* for this instead.

        detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable
            The function applied to each segment before fft-ing,
            designed to remove the mean or linear trend.  Unlike in
            MATLAB, where the *detrend* parameter is a vector, in
            matplotlib is it a function.  The :mod:`~matplotlib.mlab`
            module defines :func:`~matplotlib.mlab.detrend_none`,
            :func:`~matplotlib.mlab.detrend_mean`, and
            :func:`~matplotlib.mlab.detrend_linear`, but you can use
            a custom function as well.  You can also use a string to choose
            one of the functions.  'default', 'constant', and 'mean' call
            :func:`~matplotlib.mlab.detrend_mean`.  'linear' calls
            :func:`~matplotlib.mlab.detrend_linear`.  'none' calls
            :func:`~matplotlib.mlab.detrend_none`.

        scale_by_freq : bool, optional
            Specifies whether the resulting density values should be scaled
            by the scaling frequency, which gives density in units of Hz^-1.
            This allows for integration over the returned frequency values.
            The default is True for MATLAB compatibility.

        noverlap : int
            The number of points of overlap between segments.
            The default value is 0 (no overlap).

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        return_line : bool
            Whether to include the line object plotted in the returned values.
            Default is False.

        Returns
        -------
        Pxy : 1-D array
            The values for the cross spectrum `P_{xy}` before scaling
            (complex valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *Pxy*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.
            Only returned if *return_line* is True.

        Other Parameters
        ----------------
        **kwargs :
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
```

```
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

        See Also
        --------
        :func:`psd`
            :func:`psd` is the equivalent to setting y=x.

        Notes
        -----
        For plotting, the power is plotted as
        :math:`10\log_{10}(P_{xy})` for decibels, though `P_{xy}` itself
        is returned.
```

```
        References
        ----------
        Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
        John Wiley & Sons (1986)

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    delaxes(ax=None)
        Remove the `Axes` *ax* (defaulting to the current axes) from its figur
e.

        A KeyError is raised if the axes doesn't exist.

    disconnect(cid)
        Disconnect callback id cid

        Examples
        --------
        Usage::

            cid = canvas.mpl_connect('button_press_event', on_press)
            #...later
            canvas.mpl_disconnect(cid)

    draw()
        Redraw the current figure.

        This is used to update a figure that has been altered, but not
        automatically re-drawn.  If interactive mode is on (:func:`.ion()`), th
is
        should be only rarely needed, but there may be ways to modify the state
of
        a figure without marking it as `stale`.  Please report these cases as
        bugs.

        A more object-oriented alternative, given any
        :class:`~matplotlib.figure.Figure` instance, :attr:`fig`, that
        was created using a :mod:`~matplotlib.pyplot` function, is::

            fig.canvas.draw_idle()

    errorbar(x, y, yerr=None, xerr=None, fmt='', ecolor=None, elinewidth=None,
 capsize=None, barsabove=False, lolims=False, uplims=False, xlolims=False, xupl
ims=False, errorevery=1, capthick=None, *, data=None, **kwargs)
        Plot y versus x as lines and/or markers with attached errorbars.
```

```
*x*, *y* define the data locations, *xerr*, *yerr* define the errorbar
sizes. By default, this draws the data markers/lines as well the
errorbars. Use fmt='none' to draw errorbars without any data markers.

Parameters
----------
x, y : scalar or array-like
    The data positions.

xerr, yerr : scalar or array-like, shape(N,) or shape(2,N), optional
    The errorbar sizes:

    - scalar: Symmetric +/- values for all data points.
    - shape(N,): Symmetric +/-values for each data point.
    - shape(2,N): Separate - and + values for each bar. First row
        contains the lower errors, the second row contains the
        upper errors.
    - *None*: No errorbar.

    Note that all error arrays should have *positive* values.

    See :doc:`/gallery/statistics/errorbar_features`
    for an example on the usage of ``xerr`` and ``yerr``.

fmt : plot format string, optional, default: ''
    The format for the data points / data lines. See `.plot` for
    details.

    Use 'none' (case insensitive) to plot errorbars without any data
    markers.

ecolor : mpl color, optional, default: None
    A matplotlib color arg which gives the color the errorbar lines.
    If None, use the color of the line connecting the markers.

elinewidth : scalar, optional, default: None
    The linewidth of the errorbar lines. If None, the linewidth of
    the current style is used.

capsize : scalar, optional, default: None
    The length of the error bar caps in points. If None, it will take
    the value from :rc:`errorbar.capsize`.

capthick : scalar, optional, default: None
    An alias to the keyword argument *markeredgewidth* (a.k.a. *mew*).
    This setting is a more sensible name for the property that
    controls the thickness of the error bar cap in points. For
    backwards compatibility, if *mew* or *markeredgewidth* are given,
    then they will over-ride *capthick*. This may change in future
    releases.

barsabove : bool, optional, default: False
    If True, will plot the errorbars above the plot
    symbols. Default is below.

lolims, uplims, xlolims, xuplims : bool, optional, default: None
```

```
            These arguments can be used to indicate that a value gives only
            upper/lower limits. In that case a caret symbol is used to
            indicate this. *lims*-arguments may be of the same type as *xerr*
            and *yerr*.  To use limits with inverted axes, :meth:`set_xlim`
            or :meth:`set_ylim` must be called before :meth:`errorbar`.

        errorevery : positive integer, optional, default: 1
            Subsamples the errorbars. e.g., if errorevery=5, errorbars for
            every 5-th datapoint will be plotted. The data plot itself still
            shows all data points.

        Returns
        -------
        container : :class:`~.container.ErrorbarContainer`
            The container contains:

            - plotline: :class:`~matplotlib.lines.Line2D` instance of
              x, y plot markers and/or line.
            - caplines: A tuple of :class:`~matplotlib.lines.Line2D` instances
              of the error bar caps.
            - barlinecols: A tuple of
              :class:`~matplotlib.collections.LineCollection` with the
              horizontal and vertical error ranges.

        Other Parameters
        ----------------
        **kwargs :
            All other keyword arguments are passed on to the plot
            command for the markers. For example, this code makes big red
            squares with thick green edges::

                x,y,yerr = rand(3,10)
                errorbar(x, y, yerr, marker='s', mfc='red',
                         mec='green', ms=20, mew=4)

            where *mfc*, *mec*, *ms* and *mew* are aliases for the longer
            property names, *markerfacecolor*, *markeredgecolor*, *markersize*
            and *markeredgewidth*.

            Valid kwargs for the marker properties are `.Lines2D` properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
    y 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` | Non
    e]
      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

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

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

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

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

        eventplot(positions, orientation='horizontal', lineoffsets=1, linelengths=
    1, linewidths=None, colors=None, linestyles='solid', *, data=None, **kwargs)
            Plot identical parallel lines at the given positions.

            *positions* should be a 1D or 2D array-like object, with each row
            corresponding to a row or column of lines.

            This type of plot is commonly used in neuroscience for representing
            neural events, where it is usually called a spike raster, dot raster,
            or raster plot.
```

```
However, it is useful in any situation where you wish to show the
timing or position of multiple sets of discrete events, such as the
arrival times of people to a business on each day of the month or the
date of hurricanes each year of the last century.

Parameters
----------
positions : 1D or 2D array-like object
    Each value is an event. If *positions* is a 2D array-like, each
    row corresponds to a row or a column of lines (depending on the
    *orientation* parameter).

orientation : {'horizontal', 'vertical'}, optional
    Controls the direction of the event collections:

        - 'horizontal' : the lines are arranged horizontally in rows,
          and are vertical.
        - 'vertical' : the lines are arranged vertically in columns,
          and are horizontal.

lineoffsets : scalar or sequence of scalars, optional, default: 1
    The offset of the center of the lines from the origin, in the
    direction orthogonal to *orientation*.

linelengths : scalar or sequence of scalars, optional, default: 1
    The total height of the lines (i.e. the lines stretches from
    ``lineoffset - linelength/2`` to ``lineoffset + linelength/2``).

linewidths : scalar, scalar sequence or None, optional, default: None
    The line width(s) of the event lines, in points. If it is None,
    defaults to its rcParams setting.

colors : color, sequence of colors or None, optional, default: None
    The color(s) of the event lines. If it is None, defaults to its
    rcParams setting.

linestyles : str or tuple or a sequence of such values, optional
    Default is 'solid'. Valid strings are ['solid', 'dashed',
    'dashdot', 'dotted', '-', '--', '-.', ':']. Dash tuples
    should be of the form::

        (offset, onoffseq),

    where *onoffseq* is an even length tuple of on and off ink
    in points.

**kwargs : optional
    Other keyword arguments are line collection properties.  See
    :class:`~matplotlib.collections.LineCollection` for a list of
    the valid properties.

Returns
-------

list : A list of :class:`~.collections.EventCollection` objects.
    Contains the :class:`~.collections.EventCollection` that
    were added.
```

```
        Notes
        -----

        For *linelengths*, *linewidths*, *colors*, and *linestyles*, if only
        a single value is given, that value is applied to all lines.  If an
        array-like is given, it must have the same length as *positions*, and
        each value will be applied to the corresponding row of the array.

        Examples
        --------

        .. plot:: gallery/lines_bars_and_markers/eventplot_demo.py

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

            * All arguments with the following names: 'colors', 'linelengths',
 'lineoffsets', 'linestyles', 'linewidths', 'positions'.

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

    figimage(*args, **kwargs)
        Add a non-resampled image to the figure.

        The image is attached to the lower or upper left corner depending on
        *origin*.

        Parameters
        ----------
        X
            The image data. This is an array of one of the following shapes:

            - MxN: luminance (grayscale) values
            - MxNx3: RGB values
            - MxNx4: RGBA values

        xo, yo : int
            The *x*/*y* image offset in pixels.

        alpha : None or float
            The alpha blending value.

        norm : :class:`matplotlib.colors.Normalize`
            A :class:`.Normalize` instance to map the luminance to the
            interval [0, 1].

        cmap : str or :class:`matplotlib.colors.Colormap`
            The colormap to use. Default: :rc:`image.cmap`.

        vmin, vmax : scalar
```

```
        If *norm* is not given, these values set the data limits for the
        colormap.

    origin : {'upper', 'lower'}
        Indicates where the [0, 0] index of the array is in the upper left
        or lower left corner of the axes. Defaults to :rc:`image.origin`.

    resize : bool
        If *True*, resize the figure to match the given image size.

    Returns
    -------
    :class:`matplotlib.image.FigureImage`

    Other Parameters
    ----------------
    **kwargs
        Additional kwargs are `.Artist` kwargs passed on to `.FigureImage`.

    Notes
    -----
    figimage complements the axes image
    (:meth:`~matplotlib.axes.Axes.imshow`) which will be resampled
    to fit the current axes.  If you want a resampled image to
    fill the entire figure, you can define an
    :class:`~matplotlib.axes.Axes` with extent [0,0,1,1].


    Examples::

        f = plt.figure()
        nx = int(f.get_figwidth() * f.dpi)
        ny = int(f.get_figheight() * f.dpi)
        data = np.random.random((ny, nx))
        f.figimage(data)
        plt.show()

figlegend(*args, **kwargs)
    Place a legend in the figure.

    *labels*
      a sequence of strings

    *handles*
      a sequence of :class:`~matplotlib.lines.Line2D` or
      :class:`~matplotlib.patches.Patch` instances

    *loc*
      can be a string or an integer specifying the legend
      location

    A :class:`matplotlib.legend.Legend` instance is returned.

    Examples
    --------

    To make a legend from existing artists on every axes::
```

```
            figlegend()

        To make a legend for a list of lines and labels::

            figlegend( (line1, line2, line3),
                       ('label1', 'label2', 'label3'),
                       'upper right' )

        .. seealso::

           :func:`~matplotlib.pyplot.legend`

    fignum_exists(num)
        Return whether the figure with the given id exists.

    figtext(x, y, s, *args, **kwargs)
        Add text to figure.

        Parameters
        ----------
        x, y : float
            The position to place the text. By default, this is in figure
            coordinates, floats in [0, 1]. The coordinate system can be changed
            using the *transform* keyword.

        s : str
            The text string.

        fontdict : dictionary, optional, default: None
            A dictionary to override the default text properties. If fontdict
            is None, the defaults are determined by your rc parameters. A
            property in *kwargs* override the same property in fontdict.

        withdash : boolean, optional, default: False
            Creates a `~matplotlib.text.TextWithDash` instance instead of a
            `~matplotlib.text.Text` instance.

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.text.Text` properties
            Other miscellaneous text parameters.
              agg_filter: a filter function, which takes a (m, n, 3) float arra
y and a dpi value, and returns a (m, n, 3) array
          alpha: float
          animated: bool
          backgroundcolor: color
          bbox: dict with properties for `.patches.FancyBboxPatch`
          clip_box: `matplotlib.transforms.Bbox`
          clip_on: bool
          clip_path: { (`.path.Path`, `.transforms.Transform`), `.patches.Patch
`, None }
          color: color
          contains: callable
          figure: `.Figure`
          fontfamily: {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy',
 'monospace'}
```

```
          fontname: {FONTNAME, 'serif', 'sans-serif', 'cursive', 'fantasy', 'mo
nospace'}
          fontproperties: `.font_manager.FontProperties`
          fontsize: {size in points, 'xx-small', 'x-small', 'small', 'medium',
 'large', 'x-large', 'xx-large'}
          fontstretch: {a numeric value in range 0-1000, 'ultra-condensed', 'ex
tra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expa
nded', 'extra-expanded', 'ultra-expanded'}
          fontstyle: {'normal', 'italic', 'oblique'}
          fontvariant: {'normal', 'small-caps'}
          fontweight: {a numeric value in range 0-1000, 'ultralight', 'light',
 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'dem
i', 'bold', 'heavy', 'extra bold', 'black'}
          gid: str
          horizontalalignment: {'center', 'right', 'left'}
          in_layout: bool
          label: object
          linespacing: float (multiple of font size)
          multialignment: {'left', 'right', 'center'}
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          position: (float, float)
          rasterized: bool or None
          rotation: {angle in degrees, 'vertical', 'horizontal'}
          rotation_mode: {None, 'default', 'anchor'}
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          text: string or object castable to string (but ``None`` becomes ``''``
`)
          transform: `.Transform`
          url: str
          usetex: bool or None
          verticalalignment: {'center', 'top', 'bottom', 'baseline', 'center_ba
seline'}
          visible: bool
          wrap: bool
          x: float
          y: float
          zorder: float

        Returns
        -------
        text : `~.text.Text`

        See Also
        --------
        .Axes.text
        .pyplot.text

    figure(num=None, figsize=None, dpi=None, facecolor=None, edgecolor=None, fr
ameon=True, FigureClass=<class 'matplotlib.figure.Figure'>, clear=False, **kwar
gs)
        Create a new figure.

        Parameters
        ----------
```

```
        num : integer or string, optional, default: None
            If not provided, a new figure will be created, and the figure numbe
r
            will be incremented. The figure objects holds this number in a `num
ber`
            attribute.
            If num is provided, and a figure with this id already exists, make
            it active, and returns a reference to it. If this figure does not
            exists, create it and returns it.
            If num is a string, the window title will be set to this figure's
            `num`.

        figsize : (float, float), optional, default: None
            width, height in inches. If not provided, defaults to
            :rc:`figure.figsize` = ``[6.4, 4.8]``.

        dpi : integer, optional, default: None
            resolution of the figure. If not provided, defaults to
            :rc:`figure.dpi` = ``100``.

        facecolor :
            the background color. If not provided, defaults to
            :rc:`figure.facecolor` = ``'w'``.

        edgecolor :
            the border color. If not provided, defaults to
            :rc:`figure.edgecolor` = ``'w'``.

        frameon : bool, optional, default: True
            If False, suppress drawing the figure frame.

        FigureClass : subclass of `~matplotlib.figure.Figure`
            Optionally use a custom `.Figure` instance.

        clear : bool, optional, default: False
            If True and the figure already exists, then it is cleared.

        Returns
        -------
        figure : `~matplotlib.figure.Figure`
            The `.Figure` instance returned will also be passed to new_figure_m
anager
            in the backends, which allows to hook custom `.Figure` classes into
the
            pyplot interface. Additional kwargs will be passed to the `.Figure`
            init function.

        Notes
        -----
        If you are creating many figures, make sure you explicitly call
        :func:`.pyplot.close` on the figures you are not using, because this wi
ll
        enable pyplot to properly clean up the memory.

        `~matplotlib.rcParams` defines the default values, which can be modifie
d
        in the matplotlibrc file.
```

```
    fill(*args, data=None, **kwargs)
        Plot filled polygons.

        Parameters
        ----------
        args : sequence of x, y, [color]
            Each polygon is defined by the lists of *x* and *y* positions of
            its nodes, optionally followed by a *color* specifier. See
            :mod:`matplotlib.colors` for supported color specifiers. The
            standard color cycle is used for polygons without a color
            specifier.

            You can plot multiple polygons by providing multiple *x*, *y*,
            *[color]* groups.

            For example, each of the following is legal::

                ax.fill(x, y)                    # a polygon with default color
                ax.fill(x, y, "b")               # a blue polygon
                ax.fill(x, y, x2, y2)            # two polygons
                ax.fill(x, y, "b", x2, y2, "r")  # a blue and a red polygon

        Returns
        -------
        a list of :class:`~matplotlib.patches.Polygon`

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

        Notes
        -----
        Use :meth:`fill_between` if you would like to fill the region between
        two curves.

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    fill_between(x, y1, y2=0, where=None, interpolate=False, step=None, *, data
=None, **kwargs)
        Fill the area between two horizontal curves.

        The curves are defined by the points (*x*, *y1*) and (*x*, *y2*). This
        creates one or multiple polygons describing the filled area.

        You may exclude some horizontal sections from filling using *where*.
```

```
By default, the edges connect the given points directly. Use *step* if
the filling should be a step function, i.e. constant in between *x*.


Parameters
----------
x : array (length N)
    The x coordinates of the nodes defining the curves.

y1 : array (length N) or scalar
    The y coordinates of the nodes defining the first curve.

y2 : array (length N) or scalar, optional, default: 0
    The y coordinates of the nodes defining the second curve.

where : array of bool (length N), optional, default: None
    Define *where* to exclude some horizontal regions from being
    filled. The filled regions are defined by the coordinates
    ``x[where]``.  More precisely, fill between ``x[i]`` and ``x[i+1]``
    if ``where[i] and where[i+1]``.  Note that this definition implies
    that an isolated *True* value between two *False* values in
    *where* will not result in filling.  Both sides of the *True*
    position remain unfilled due to the adjacent *False* values.

interpolate : bool, optional
    This option is only relvant if *where* is used and the two curves
    are crossing each other.

    Semantically, *where* is often used for *y1* > *y2* or similar.
    By default, the nodes of the polygon defining the filled region
    will only be placed at the positions in the *x* array.  Such a
    polygon cannot describe the above semantics close to the
    intersection.  The x-sections containing the intersection are
    simply clipped.

    Setting *interpolate* to *True* will calculate the actual
    intersection point and extend the filled region up to this point.

step : {'pre', 'post', 'mid'}, optional
    Define *step* if the filling should be a step function,
    i.e. constant in between *x*. The value determines where the
    step will occur:

    - 'pre': The y value is continued constantly to the left from
      every *x* position, i.e. the interval ``(x[i-1], x[i]]`` has the
      value ``y[i]``.
    - 'post': The y value is continued constantly to the right from
      every *x* position, i.e. the interval ``[x[i], x[i+1])`` has the
      value ``y[i]``.
    - 'mid': Steps occur half-way between the *x* positions.

Other Parameters
----------------
**kwargs
    All other keyword arguments are passed on to `.PolyCollection`.
    They control the `.Polygon` properties:
```

```
              agg_filter: a filter function, which takes a (m, n, 3) float arra
y and a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        Returns
        -------
        `.PolyCollection`
            A `.PolyCollection` containing the plotted polygons.

        See Also
        --------
        fill_betweenx : Fill between two sets of x-values.

        Notes
        -----
        .. [notes section required to get data note injection right]

        .. note::
            In addition to the above described arguments, this function can tak
```

```
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'where', 'x', 'y1', 'y2'.

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

    fill_betweenx(y, x1, x2=0, where=None, step=None, interpolate=False, *, dat
a=None, **kwargs)
        Fill the area between two vertical curves.

        The curves are defined by the points (*x1*, *y*) and (*x2*, *y*). This
        creates one or multiple polygons describing the filled area.

        You may exclude some vertical sections from filling using *where*.

        By default, the edges connect the given points directly. Use *step* if
        the filling should be a step function, i.e. constant in between *y*.


        Parameters
        ----------
        y : array (length N)
            The y coordinates of the nodes defining the curves.

        x1 : array (length N) or scalar
            The x coordinates of the nodes defining the first curve.

        x2 : array (length N) or scalar, optional, default: 0
            The x coordinates of the nodes defining the second curve.

        where : array of bool (length N), optional, default: None
            Define *where* to exclude some vertical regions from being
            filled. The filled regions are defined by the coordinates
            ``y[where]``.  More precisely, fill between ``y[i]`` and ``y[i+1]``
            if ``where[i] and where[i+1]``.  Note that this definition implies
            that an isolated *True* value between two *False* values in
            *where* will not result in filling.  Both sides of the *True*
            position remain unfilled due to the adjacent *False* values.

        interpolate : bool, optional
            This option is only relvant if *where* is used and the two curves
            are crossing each other.

            Semantically, *where* is often used for *x1* > *x2* or similar.
            By default, the nodes of the polygon defining the filled region
            will only be placed at the positions in the *y* array.  Such a
            polygon cannot describe the above semantics close to the
            intersection.  The y-sections containing the intersecion are
            simply clipped.

            Setting *interpolate* to *True* will calculate the actual
            interscection point and extend the filled region up to this point.
```

```
        step : {'pre', 'post', 'mid'}, optional
            Define *step* if the filling should be a step function,
            i.e. constant in between *y*. The value determines where the
            step will occur:

            - 'pre': The y value is continued constantly to the left from
              every *x* position, i.e. the interval ``(x[i-1], x[i]]`` has the
              value ``y[i]``.
            - 'post': The y value is continued constantly to the right from
              every *x* position, i.e. the interval ``[x[i], x[i+1])`` has the
              value ``y[i]``.
            - 'mid': Steps occur half-way between the *x* positions.

        Other Parameters
        ----------------
        **kwargs
            All other keyword arguments are passed on to `.PolyCollection`.
            They control the `.Polygon` properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y and a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
```

```
            urls: List[str] or None
            visible: bool
            zorder: float

        Returns
        -------
        `.PolyCollection`
            A `.PolyCollection` containing the plotted polygons.

        See Also
        --------
        fill_between : Fill between two sets of y-values.

        Notes
        -----
        .. [notes section required to get data note injection right]

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

            * All arguments with the following names: 'where', 'x1', 'x2', 'y'.

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

    findobj(o=None, match=None, include_self=True)
        Find artist objects.

        Recursively find all :class:`~matplotlib.artist.Artist` instances
        contained in self.

        *match* can be

          - None: return all objects contained in artist.

          - function with signature ``boolean = match(artist)``
            used to filter matches

          - class instance: e.g., Line2D.  Only return artists of class type.

        If *include_self* is True (default), include self in the list to be
        checked for a match.

    flag()
        Set the colormap to "flag".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    gca(**kwargs)
        Get the current :class:`~matplotlib.axes.Axes` instance on the
```

```
        current figure matching the given keyword args, or create one.

        Examples
        --------
        To get the current polar axes on the current figure::

            plt.gca(projection='polar')

        If the current axes doesn't exist, or isn't a polar one, the appropriat
e
        axes will be created and then returned.

        See Also
        --------
        matplotlib.figure.Figure.gca : The figure's gca method.

    gcf()
        Get a reference to the current figure.

    gci()
        Get the current colorable artist.  Specifically, returns the
        current :class:`~matplotlib.cm.ScalarMappable` instance (image or
        patch collection), or *None* if no images or patch collections
        have been defined.  The commands :func:`~matplotlib.pyplot.imshow`
        and :func:`~matplotlib.pyplot.figimage` create
        :class:`~matplotlib.image.Image` instances, and the commands
        :func:`~matplotlib.pyplot.pcolor` and
        :func:`~matplotlib.pyplot.scatter` create
        :class:`~matplotlib.collections.Collection` instances.  The
        current image is an attribute of the current axes, or the nearest
        earlier axes in the current figure that contains an image.

    get_current_fig_manager()
        Return the figure manager of the active figure.

        If there is currently no active figure, a new one is created.

    get_figlabels()
        Return a list of existing figure labels.

    get_fignums()
        Return a list of existing figure numbers.

    get_plot_commands()
        Get a sorted list of all of the plotting commands.

    ginput(*args, **kwargs)
        Blocking call to interact with a figure.

        Wait until the user clicks *n* times on the figure, and return the
        coordinates of each click in a list.

        The buttons used for the various actions (adding points, removing
        points, terminating the inputs) can be overridden via the
        arguments *mouse_add*, *mouse_pop* and *mouse_stop*, that give
        the associated mouse button: 1 for left, 2 for middle, 3 for
        right.
```

```
        Parameters
        ----------
        n : int, optional, default: 1
            Number of mouse clicks to accumulate. If negative, accumulate
            clicks until the input is terminated manually.
        timeout : scalar, optional, default: 30
            Number of seconds to wait before timing out. If zero or negative
            will never timeout.
        show_clicks : bool, optional, default: False
            If True, show a red cross at the location of each click.
        mouse_add : int, one of (1, 2, 3), optional, default: 1 (left click)
            Mouse button used to add points.
        mouse_pop : int, one of (1, 2, 3), optional, default: 3 (right click)
            Mouse button used to remove the most recently added point.
        mouse_stop : int, one of (1, 2, 3), optional, default: 2 (middle click)
            Mouse button used to stop input.

        Returns
        -------
        points : list of tuples
            A list of the clicked (x, y) coordinates.

        Notes
        -----
        The keyboard can also be used to select points in case your mouse
        does not have one or more of the buttons.  The delete and backspace
        keys act like right clicking (i.e., remove last point), the enter key
        terminates input and any other key (not already used by the window
        manager) selects a point.

    gray()
        Set the colormap to "gray".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    grid(b=None, which='major', axis='both', **kwargs)
        Configure the grid lines.

        Parameters
        ----------
        b : bool or None
            Whether to show the grid lines. If any *kwargs* are supplied,
            it is assumed you want the grid on and *b* will be set to True.

            If *b* is *None* and there are no *kwargs*, this toggles the
            visibility of the lines.

        which : {'major', 'minor', 'both'}
            The grid lines to apply the changes on.

        axis : {'both', 'x', 'y'}
            The axis to apply the changes on.

        **kwargs : `.Line2D` properties
```

```
            Define the line properties of the grid, e.g.::

                grid(color='r', linestyle='-', linewidth=2)

            Valid *kwargs* are

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

        Notes
        -----
        The grid will be drawn according to the axes' zorder and not its own.

    hexbin(x, y, C=None, gridsize=100, bins=None, xscale='linear', yscale='line
ar', extent=None, cmap=None, norm=None, vmin=None, vmax=None, alpha=None, linew
```

```
idths=None, edgecolors='face', reduce_C_function=<function mean at 0x000001FEE5
4BE488>, mincnt=None, marginals=False, *, data=None, **kwargs)
        Make a hexagonal binning plot.

        Make a hexagonal binning plot of *x* versus *y*, where *x*,
        *y* are 1-D sequences of the same length, *N*. If *C* is *None*
        (the default), this is a histogram of the number of occurrences
        of the observations at (x[i],y[i]).

        If *C* is specified, it specifies values at the coordinate
        (x[i], y[i]). These values are accumulated for each hexagonal
        bin and then reduced according to *reduce_C_function*, which
        defaults to `numpy.mean`. (If *C* is specified, it must also
        be a 1-D sequence of the same length as *x* and *y*.)

        Parameters
        ----------
        x, y : array or masked array

        C : array or masked array, optional, default is *None*

        gridsize : int or (int, int), optional, default is 100
            The number of hexagons in the *x*-direction, default is
            100. The corresponding number of hexagons in the
            *y*-direction is chosen such that the hexagons are
            approximately regular. Alternatively, gridsize can be a
            tuple with two elements specifying the number of hexagons
            in the *x*-direction and the *y*-direction.

        bins : 'log' or int or sequence, optional, default is *None*
            If *None*, no binning is applied; the color of each hexagon
            directly corresponds to its count value.

            If 'log', use a logarithmic scale for the color
            map. Internally, :math:`log_{10}(i+1)` is used to
            determine the hexagon color.

            If an integer, divide the counts in the specified number
            of bins, and color the hexagons accordingly.

            If a sequence of values, the values of the lower bound of
            the bins to be used.

        xscale : {'linear', 'log'}, optional, default is 'linear'
            Use a linear or log10 scale on the horizontal axis.

        yscale : {'linear', 'log'}, optional, default is 'linear'
            Use a linear or log10 scale on the vertical axis.

        mincnt : int > 0, optional, default is *None*
            If not *None*, only display cells with more than *mincnt*
            number of points in the cell

        marginals : bool, optional, default is *False*
            if marginals is *True*, plot the marginal density as
            colormapped rectagles along the bottom of the x-axis and
            left of the y-axis
```

```
extent : scalar, optional, default is *None*
    The limits of the bins. The default assigns the limits
    based on *gridsize*, *x*, *y*, *xscale* and *yscale*.

    If *xscale* or *yscale* is set to 'log', the limits are
    expected to be the exponent for a power of 10. E.g. for
    x-limits of 1 and 50 in 'linear' scale and y-limits
    of 10 and 1000 in 'log' scale, enter (1, 50, 1, 3).

    Order of scalars is (left, right, bottom, top).

Other Parameters
----------------
cmap : object, optional, default is *None*
    a :class:`matplotlib.colors.Colormap` instance. If *None*,
    defaults to rc ``image.cmap``.

norm : object, optional, default is *None*
    :class:`matplotlib.colors.Normalize` instance is used to
    scale luminance data to 0,1.

vmin, vmax : scalar, optional, default is *None*
    *vmin* and *vmax* are used in conjunction with *norm* to
    normalize luminance data. If *None*, the min and max of the
    color array *C* are used.  Note if you pass a norm instance
    your settings for *vmin* and *vmax* will be ignored.

alpha : scalar between 0 and 1, optional, default is *None*
    the alpha value for the patches

linewidths : scalar, optional, default is *None*
    If *None*, defaults to 1.0.

edgecolors : {'face', 'none', *None*} or color, optional

    If 'face' (the default), draws the edges in the same color as the
    fill color.

    If 'none', no edge is drawn; this can sometimes lead to unsightly
    unpainted pixels between the hexagons.

    If *None*, draws outlines in the default color.

    If a matplotlib color arg, draws outlines in the specified color.

Returns
-------
polycollection
    A `.PolyCollection` instance; use `.PolyCollection.get_array` on
    this to get the counts in each hexagon.

    If *marginals* is *True*, horizontal
    bar and vertical bar (both PolyCollections) will be attached
    to the return collection as attributes *hbar* and *vbar*.

Notes
```

```
        -----
        The standard descriptions of all the
        :class:`~matplotlib.collections.Collection` parameters:

             agg_filter: a filter function, which takes a (m, n, 3) float arra
y and a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).
```

```
    hist(x, bins=None, range=None, density=None, weights=None, cumulative=Fals
e, bottom=None, histtype='bar', align='mid', orientation='vertical', rwidth=Non
e, log=False, color=None, label=None, stacked=False, normed=None, *, data=None,
**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 tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    hist2d(x, y, bins=10, range=None, normed=False, weights=None, cmin=None, cm
ax=None, *, data=None, **kwargs)
        Make a 2D histogram plot.

        Parameters
        ----------
```

```
x, y : array_like, shape (n, )
    Input values

bins : None or int or [int, int] or array_like or [array, array]

    The bin specification:

        - If int, the number of bins for the two dimensions
          (nx=ny=bins).

        - If ``[int, int]``, the number of bins in each dimension
          (nx, ny = bins).

        - If array_like, the bin edges for the two dimensions
          (x_edges=y_edges=bins).

        - If ``[array, array]``, the bin edges in each dimension
          (x_edges, y_edges = bins).

    The default value is 10.

range : array_like shape(2, 2), optional, default: None
     The leftmost and rightmost edges of the bins along each dimension
     (if not specified explicitly in the bins parameters): ``[[xmin,
     xmax], [ymin, ymax]]``. All values outside of this range will be
     considered outliers and not tallied in the histogram.

normed : bool, optional, default: False
     Normalize histogram.

weights : array_like, shape (n, ), optional, default: None
    An array of values w_i weighing each sample (x_i, y_i).

cmin : scalar, optional, default: None
     All bins that has count less than cmin will not be displayed and
     these count values in the return value count histogram will also
     be set to nan upon return

cmax : scalar, optional, default: None
     All bins that has count more than cmax will not be displayed (set
     to none before passing to imshow) and these count values in the
     return value count histogram will also be set to nan upon return

Returns
-------
h : 2D array
    The bi-dimensional histogram of samples x and y. Values in x are
    histogrammed along the first dimension and values in y are
    histogrammed along the second dimension.
xedges : 1D array
    The bin edges along the x axis.
yedges : 1D array
    The bin edges along the y axis.
image : `~.matplotlib.collections.QuadMesh`

Other Parameters
----------------
```

```
        cmap : Colormap or str, optional
            A `.colors.Colormap` instance.  If not set, use rc settings.

        norm : Normalize, optional
            A `.colors.Normalize` instance is used to
            scale luminance data to ``[0, 1]``. If not set, defaults to
            `.colors.Normalize()`.

        vmin/vmax : None or scalar, optional
            Arguments passed to the `~.colors.Normalize` instance.

        alpha : ``0 <= scalar <= 1`` or ``None``, optional
            The alpha blending value.

        See also
        --------
        hist : 1D histogram plotting

        Notes
        -----
        - Currently ``hist2d`` calculates it's own axis limits, and any limits
          previously set are ignored.
        - Rendering the histogram with a logarithmic color scale is
          accomplished by passing a `.colors.LogNorm` instance to the *norm*
          keyword argument. Likewise, power-law normalization (similar
          in effect to gamma correction) can be accomplished with
          `.colors.PowerNorm`.

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

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

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

    hlines(y, xmin, xmax, colors='k', linestyles='solid', label='', *, data=Non
e, **kwargs)
        Plot horizontal lines at each *y* from *xmin* to *xmax*.

        Parameters
        ----------
        y : scalar or sequence of scalar
            y-indexes where to plot the lines.

        xmin, xmax : scalar or 1D array_like
            Respective beginning and end of each line. If scalars are
            provided, all lines will have same length.

        colors : array_like of colors, optional, default: 'k'

        linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, optional
```

```
        label : string, optional, default: ''

        Returns
        -------
        lines : `~matplotlib.collections.LineCollection`

        Other Parameters
        ----------------
        **kwargs :  `~matplotlib.collections.LineCollection` properties.

        See also
        --------
        vlines : vertical lines
        axhline: horizontal line across the axes

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

            * All arguments with the following names: 'colors', 'xmax', 'xmin',
'y'.

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

    hot()
        Set the colormap to "hot".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    hsv()
        Set the colormap to "hsv".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    imread(fname, format=None)
        Read an image from a file into an array.

        Parameters
        ----------
        fname : str or file-like
            The image file to read. This can be a filename, a URL or a Python
            file-like object opened in read-binary mode.
        format : str, optional
            The image file format assumed for reading the data. If not
            given, the format is deduced from the filename.  If nothing can
            be deduced, PNG is tried.
```

```
        Returns
        -------
        imagedata : :class:`numpy.array`
            The image data. The returned array has shape

            - (M, N) for grayscale images.
            - (M, N, 3) for RGB images.
            - (M, N, 4) for RGBA images.

        Notes
        -----
        Matplotlib can only read PNGs natively. Further image formats are
        supported via the optional dependency on Pillow. Note, URL strings
        are not compatible with Pillow. Check the `Pillow documentation`_
        for more information.

        .. _Pillow documentation: http://pillow.readthedocs.io/en/latest/ (htt
    p://pillow.readthedocs.io/en/latest/)

    imsave(fname, arr, **kwargs)
        Save an array as in image file.

        The output formats available depend on the backend being used.

        Parameters
        ----------
        fname : str or file-like
            The filename or a Python file-like object to store the image in.
            The necessary output format is inferred from the filename extension
            but may be explicitly overwritten using *format*.
        arr : array-like
            The image data. The shape can be one of
            MxN (luminance), MxNx3 (RGB) or MxNx4 (RGBA).
        vmin, vmax : scalar, optional
            *vmin* and *vmax* set the color scaling for the image by fixing the
            values that map to the colormap color limits. If either *vmin*
            or *vmax* is None, that limit is determined from the *arr*
            min/max value.
        cmap : str or `~matplotlib.colors.Colormap`, optional
            A Colormap instance or registered colormap name. The colormap
            maps scalar data to colors. It is ignored for RGB(A) data.
            Defaults to :rc:`image.cmap` ('viridis').
        format : str, optional
            The file format, e.g. 'png', 'pdf', 'svg', ... . If not given, the
            format is deduced form the filename extension in *fname*.
            See `.Figure.savefig` for details.
        origin : {'upper', 'lower'}, optional
            Indicates whether the ``(0, 0)`` index of the array is in the upper
            left or lower left corner of the axes.  Defaults to :rc:`image.orig
    in`
            ('upper').
        dpi : int
            The DPI to store in the metadata of the file.  This does not affect
    the
            resolution of the output image.

    imshow(X, cmap=None, norm=None, aspect=None, interpolation=None, alpha=Non
```

```
e, vmin=None, vmax=None, origin=None, extent=None, shape=None, filternorm=1, fi
lterrad=4.0, imlim=None, resample=None, url=None, *, data=None, **kwargs)
        Display an image, i.e. data on a 2D regular raster.

        Parameters
        ----------
        X : array-like or PIL image
            The image data. Supported array shapes are:

            - (M, N): an image with scalar data. The data is visualized
              using a colormap.
            - (M, N, 3): an image with RGB values (float or uint8).
            - (M, N, 4): an image with RGBA values (float or uint8), i.e.
              including transparency.

            The first two dimensions (M, N) define the rows and columns of
            the image.

            The RGB(A) values should be in the range [0 .. 1] for floats or
            [0 .. 255] for integers.  Out-of-range values will be clipped to
            these bounds.

        cmap : str or `~matplotlib.colors.Colormap`, optional
            A Colormap instance or registered colormap name. The colormap
            maps scalar data to colors. It is ignored for RGB(A) data.
            Defaults to :rc:`image.cmap`.

        aspect : {'equal', 'auto'} or float, optional
            Controls the aspect ratio of the axes. The aspect is of particular
            relevance for images since it may distort the image, i.e. pixel
            will not be square.

            This parameter is a shortcut for explicitly calling
            `.Axes.set_aspect`. See there for further details.

            - 'equal': Ensures an aspect ratio of 1. Pixels will be square
              (unless pixel sizes are explicitly made non-square in data
              coordinates using *extent*).
            - 'auto': The axes is kept fixed and the aspect is adjusted so
              that the data fit in the axes. In general, this will result in
              non-square pixels.

            If not given, use :rc:`image.aspect` (default: 'equal').

        interpolation : str, optional
            The interpolation method used. If *None*
            :rc:`image.interpolation` is used, which defaults to 'nearest'.

            Supported values are 'none', 'nearest', 'bilinear', 'bicubic',
            'spline16', 'spline36', 'hanning', 'hamming', 'hermite', 'kaiser',
            'quadric', 'catrom', 'gaussian', 'bessel', 'mitchell', 'sinc',
            'lanczos'.

            If *interpolation* is 'none', then no interpolation is performed
            on the Agg, ps and pdf backends. Other backends will fall back to
            'nearest'.
```

```
        See
        :doc:`/gallery/images_contours_and_fields/interpolation_methods`
        for an overview of the supported interpolation methods.

        Some interpolation methods require an additional radius parameter,
        which can be set by *filterrad*. Additionally, the antigrain image
        resize filter is controlled by the parameter *filternorm*.

    norm : `~matplotlib.colors.Normalize`, optional
        If scalar data are used, the Normalize instance scales the
        data values to the canonical colormap range [0,1] for mapping
        to colors. By default, the data range is mapped to the
        colorbar range using linear scaling. This parameter is ignored for
        RGB(A) data.

    vmin, vmax : scalar, optional
        When using scalar data and no explicit *norm*, *vmin* and *vmax*
        define the data range that the colormap covers. By default,
        the colormap covers the complete value range of the supplied
        data. *vmin*, *vmax* are ignored if the *norm* parameter is used.

    alpha : scalar, optional
        The alpha blending value, between 0 (transparent) and 1 (opaque).
        This parameter is ignored for RGBA input data.

    origin : {'upper', 'lower'}, optional
        Place the [0,0] index of the array in the upper left or lower left
        corner of the axes. The convention 'upper' is typically used for
        matrices and images.
        If not given, :rc:`image.origin` is used, defaulting to 'upper'.

        Note that the vertical axes points upward for 'lower'
        but downward for 'upper'.

    extent : scalars (left, right, bottom, top), optional
        The bounding box in data coordinates that the image will fill.
        The image is stretched individually along x and y to fill the box.

        The default extent is determined by the following conditions.
        Pixels have unit size in data coordinates. Their centers are on
        integer coordinates, and their center coordinates range from 0 to
        columns-1 horizontally and from 0 to rows-1 vertically.

        Note that the direction of the vertical axis and thus the default
        values for top and bottom depend on *origin*:

        - For ``origin == 'upper'`` the default is
          ``(-0.5, numcols-0.5, numrows-0.5, -0.5)``.
        - For ``origin == 'lower'`` the default is
          ``(-0.5, numcols-0.5, -0.5, numrows-0.5)``.

        See the example :doc:`/tutorials/intermediate/imshow_extent` for a
        more detailed description.

    shape : scalars (columns, rows), optional, default: None
        For raw buffer images.
```

```
            filternorm : bool, optional, default: True
                A parameter for the antigrain image resize filter (see the
                antigrain documentation).  If *filternorm* is set, the filter
                normalizes integer values and corrects the rounding errors. It
                doesn't do anything with the source floating point values, it
                corrects only integers according to the rule of 1.0 which means
                that any sum of pixel weights must be equal to 1.0.  So, the
                filter function must produce a graph of the proper shape.

            filterrad : float > 0, optional, default: 4.0
                The filter radius for filters that have a radius parameter, i.e.
                when interpolation is one of: 'sinc', 'lanczos' or 'blackman'.

            resample : bool, optional
                When *True*, use a full resampling method.  When *False*, only
                resample when the output image is larger than the input image.

            url : str, optional
                Set the url of the created `.AxesImage`. See `.Artist.set_url`.

            Returns
            -------
            image : `~matplotlib.image.AxesImage`

            Other Parameters
            ----------------
            **kwargs : `~matplotlib.artist.Artist` properties
                These parameters are passed on to the constructor of the
                `.AxesImage` artist.

            See also
            --------
            matshow : Plot a matrix or an array as an image.

            Notes
            -----
            Unless *extent* is used, pixel centers will be located at integer
            coordinates. In other words: the origin will coincide with the center
            of pixel (0, 0).

            There are two common representations for RGB images with an alpha
            channel:

            -   Straight (unassociated) alpha: R, G, and B channels represent the
                color of the pixel, disregarding its opacity.
            -   Premultiplied (associated) alpha: R, G, and B channels represent
                the color of the pixel, adjusted for its opacity by multiplication.

            `~matplotlib.pyplot.imshow` expects RGB images adopting the straight
            (unassociated) alpha representation.

            .. note::
                In addition to the above described arguments, this function can tak
        e a
                **data** keyword argument. If such a **data** argument is given, th
        e
                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``).

    inferno()
        Set the colormap to "inferno".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    install_repl_displayhook()
        Install a repl display hook so that any stale figure are automatically
        redrawn when control is returned to the repl.

        This works both with IPython and with vanilla python shells.

    ioff()
        Turn the interactive mode off.

    ion()
        Turn the interactive mode on.

    isinteractive()
        Return the status of interactive mode.

    jet()
        Set the colormap to "jet".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    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` ('bes
t' 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`) o
r
            `figure.bbox` (if `.Figure.legend`).  This argument allows arbitrar
y
            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 calle
d.

            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 *
loc* at
            x, y.  For example, to put the legend's upper right-hand corner in
 the
            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',
    'large', '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

locator_params(axis='both', tight=None, **kwargs)
    Control behavior of tick locators.

    Parameters
    ----------
    axis : {'both', 'x', 'y'}, optional
        The axis on which to operate.

    tight : bool or None, optional
        Parameter passed to :meth:`autoscale_view`.
        Default is None, for no change.
```

```
        Other Parameters
        ----------------
        **kw :
            Remaining keyword arguments are passed to directly to the
            :meth:`~matplotlib.ticker.MaxNLocator.set_params` method.

        Typically one might want to reduce the maximum number
        of ticks and use tight bounds when plotting small
        subplots, for example::

            ax.locator_params(tight=True, nbins=4)

        Because the locator is involved in autoscaling,
        :meth:`autoscale_view` is called automatically after
        the parameters are changed.

        This presently works only for the
        :class:`~matplotlib.ticker.MaxNLocator` used
        by default on linear axes, but it may be generalized.

    loglog(*args, **kwargs)
        Make a plot with log scaling on both the x and y axis.

        Call signatures::

            loglog([x], y, [fmt], data=None, **kwargs)
            loglog([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

        This is just a thin wrapper around `.plot` which additionally changes
        both the x-axis and the y-axis to log scaling. All of the concepts and
        parameters of plot can be used here as well.

        The additional parameters *basex/y*, *subsx/y* and *nonposx/y* control
        the x/y-axis properties. They are just forwarded to `.Axes.set_xscale`
        and `.Axes.set_yscale`.

        Parameters
        ----------
        basex, basey : scalar, optional, default 10
            Base of the x/y logarithm.

        subsx, subsy : sequence, optional
            The location of the minor x/y ticks. If *None*, reasonable
            locations are automatically chosen depending on the number of
            decades in the plot.
            See `.Axes.set_xscale` / `.Axes.set_yscale` for details.

        nonposx, nonposy : {'mask', 'clip'}, optional, default 'mask'
            Non-positive values in x or y can be masked as invalid, or clipped
            to a very small positive number.

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

        Other Parameters
```

```
        ----------------
        **kwargs
            All parameters supported by `.plot`.

    magma()
        Set the colormap to "magma".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    magnitude_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=Non
e, scale=None, *, data=None, **kwargs)
        Plot the magnitude spectrum.

        Call signature::

          magnitude_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
                             pad_to=None, sides='default', **kwargs)

        Compute the magnitude spectrum of *x*.  Data is padded to a
        length of *pad_to* and the windowing function *window* is applied to
        the signal.

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data.

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*. To create window
            vectors see :func:`window_hanning`, :func:`window_none`,
            :func:`numpy.blackman`, :func:`numpy.hamming`,
            :func:`numpy.bartlett`, :func:`scipy.signal`,
            :func:`scipy.signal.get_window`, etc. The default is
            :func:`window_hanning`.  If a function is passed as the
            argument, it must take a data segment as an argument and
            return the windowed version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.

        pad_to : int
            The number of points to which the data segment is padded when
            performing the FFT.  While not increasing the actual resolution of
            the spectrum (the minimum distance between resolvable peaks),
            this can give more points in the plot, allowing for more
            detail. This corresponds to the *n* parameter in the call to fft().
            The default is None, which sets *pad_to* equal to the length of the
```

```
            input signal (i.e. no padding).

        scale : {'default', 'linear', 'dB'}
            The scaling of the values in the *spec*.  'linear' is no scaling.
            'dB' returns the values in dB scale, i.e., the dB amplitude
            (20 * log10). 'default' is 'linear'.

        Fc : int
            The center frequency of *x* (defaults to 0), which offsets
            the x extents of the plot to reflect the frequency range used
            when a signal is acquired and then filtered and downsampled to
            baseband.

        Returns
        -------
        spectrum : 1-D array
            The values for the magnitude spectrum before scaling (real valued).

        freqs : 1-D array
            The frequencies corresponding to the elements in *spectrum*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.

        Other Parameters
        ----------------
        **kwargs :
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

              See Also
              --------
              :func:`psd`
                  :func:`psd` plots the power spectral density.`.

              :func:`angle_spectrum`
                  :func:`angle_spectrum` plots the angles of the corresponding
                  frequencies.

              :func:`phase_spectrum`
                  :func:`phase_spectrum` plots the phase (unwrapped angle) of the
                  corresponding frequencies.

              :func:`specgram`
                  :func:`specgram` can plot the magnitude spectrum of segments within
                  the signal in a colormap.

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

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

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

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

          margins(*margins, x=None, y=None, tight=True)
              Set or retrieve autoscaling margins.

              The padding added to each limit of the axes is the *margin*
              times the data interval. All input parameters must be floats
```

```
        within the range [0, 1]. Passing both positional and keyword
        arguments is invalid and will raise a TypeError. If no
        arguments (positional or otherwise) are provided, the current
        margins will remain in place and simply be returned.

        Specifying any margin changes only the autoscaling; for example,
        if *xmargin* is not None, then *xmargin* times the X data
        interval will be added to each end of that interval before
        it is used in autoscaling.

        Parameters
        ----------
        args : float, optional
            If a single positional argument is provided, it specifies
            both margins of the x-axis and y-axis limits. If two
            positional arguments are provided, they will be interpreted
            as *xmargin*, *ymargin*. If setting the margin on a single
            axis is desired, use the keyword arguments described below.

        x, y : float, optional
            Specific margin values for the x-axis and y-axis,
            respectively. These cannot be used with positional
            arguments, but can be used individually to alter on e.g.,
            only the y-axis.

        tight : bool, default is True
            The *tight* parameter is passed to :meth:`autoscale_view`,
            which is executed after a margin is changed; the default
            here is *True*, on the assumption that when margins are
            specified, no additional padding to match tick marks is
            usually desired.  Set *tight* to *None* will preserve
            the previous setting.


        Returns
        -------
        xmargin, ymargin : float

        Notes
        -----
        If a previously used Axes method such as :meth:`pcolor` has set
        :attr:`use_sticky_edges` to `True`, only the limits not set by
        the "sticky artists" will be modified. To force all of the
        margins to be set, set :attr:`use_sticky_edges` to `False`
        before calling :meth:`margins`.

    matshow(A, fignum=None, **kwargs)
        Display an array as a matrix in a new figure window.

        The origin is set at the upper left hand corner and rows (first
        dimension of the array) are displayed horizontally.  The aspect
        ratio of the figure window is that of the array, unless this would
        make an excessively short or narrow figure.

        Tick labels for the xaxis are placed on top.

        Parameters
```

```
            ----------
            A : array-like(M, N)
                The matrix to be displayed.

            fignum : None or int or False
                If *None*, create a new figure window with automatic numbering.

                If a nonzero integer, draw into the figure with the given number
                (create it if it does not exist).

                If 0, use the current axes (or create one if it does not exist).

                .. note::

                   Because of how `.Axes.matshow` tries to set the figure aspect
                   ratio to be the one of the array, strange things may happen if y
ou
                   reuse an existing figure.

            Returns
            -------
            image : `~matplotlib.image.AxesImage`

            Other Parameters
            ----------------
            **kwargs : `~matplotlib.axes.Axes.imshow` arguments

        minorticks_off()
            Remove minor ticks from the axes.

        minorticks_on()
            Display minor ticks on the axes.

            Displaying minor ticks may reduce performance; you may turn them off
            using `minorticks_off()` if drawing speed is a problem.

        nipy_spectral()
            Set the colormap to "nipy_spectral".

            This changes the default colormap as well as the colormap of the curren
t
            image if there is one. See ``help(colormaps)`` for more information.

        pause(interval)
            Pause for *interval* seconds.

            If there is an active figure, it will be updated and displayed before t
he
            pause, and the GUI event loop (if any) will run during the pause.

            This can be used for crude animation.  For more complex animation, see
            :mod:`matplotlib.animation`.

            Notes
            -----
            This function is experimental; its behavior may be changed or extended
 in a
```

```
        future release.

    pcolor(*args, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, data=
None, **kwargs)
        Create a pseudocolor plot with a non-regular rectangular grid.

        Call signature::

            pcolor([X, Y,] C, **kwargs)

        *X* and *Y* can be used to specify the corners of the quadrilaterals.

        .. hint::

            ``pcolor()`` can be very slow for large arrays. In most
            cases you should use the similar but much faster
            `~.Axes.pcolormesh` instead. See there for a discussion of the
            differences.

        Parameters
        ----------
        C : array_like
            A scalar 2-D array. The values will be color-mapped.

        X, Y : array_like, optional
            The coordinates of the quadrilateral corners. The quadrilateral
            for ``C[i,j]`` has corners at::

                (X[i+1, j], Y[i+1, j])          (X[i+1, j+1], Y[i+1, j+1])
                                      +--------+
                                      | C[i,j] |
                                      +--------+
                    (X[i, j], Y[i, j])          (X[i, j+1], Y[i, j+1]),

            Note that the column index corresponds to the
            x-coordinate, and the row index corresponds to y. For
            details, see the :ref:`Notes <axes-pcolor-grid-orientation>`
            section below.

            The dimensions of *X* and *Y* should be one greater than those of
            *C*. Alternatively, *X*, *Y* and *C* may have equal dimensions, in
            which case the last row and column of *C* will be ignored.

            If *X* and/or *Y* are 1-D arrays or column vectors they will be
            expanded as needed into the appropriate 2-D arrays, making a
            rectangular grid.

        cmap : str or `~matplotlib.colors.Colormap`, optional
            A Colormap instance or registered colormap name. The colormap
            maps the *C* values to colors. Defaults to :rc:`image.cmap`.

        norm : `~matplotlib.colors.Normalize`, optional
            The Normalize instance scales the data values to the canonical
            colormap range [0, 1] for mapping to colors. By default, the data
            range is mapped to the colorbar range using linear scaling.

        vmin, vmax : scalar, optional, default: None
```

```
            The colorbar range. If *None*, suitable min/max values are
            automatically chosen by the `~.Normalize` instance (defaults to
            the respective min/max values of *C* in case of the default linear
            scaling).

        edgecolors : {'none', None, 'face', color, color sequence}, optional
            The color of the edges. Defaults to 'none'. Possible values:

            - 'none' or '': No edge.
            - *None*: :rc:`patch.edgecolor` will be used. Note that currently
              :rc:`patch.force_edgecolor` has to be True for this to work.
            - 'face': Use the adjacent face color.
            - An mpl color or sequence of colors will set the edge color.

            The singular form *edgecolor* works as an alias.

        alpha : scalar, optional, default: None
            The alpha blending value of the face color, between 0 (transparent)
            and 1 (opaque). Note: The edgecolor is currently not affected by
            this.

        snap : bool, optional, default: False
            Whether to snap the mesh to pixel boundaries.

        Returns
        -------
        collection : `matplotlib.collections.Collection`

        Other Parameters
        ----------------
        antialiaseds : bool, optional, default: False
            The default *antialiaseds* is False if the default
            *edgecolors*\ ="none" is used.  This eliminates artificial lines
            at patch boundaries, and works regardless of the value of alpha.
            If *edgecolors* is not "none", then the default *antialiaseds*
            is taken from :rc:`patch.antialiased`, which defaults to True.
            Stroking the edges may be preferred if *alpha* is 1, but will
            cause artifacts otherwise.

        **kwargs :
            Additionally, the following arguments are allowed. They are passed
            along to the `~matplotlib.collections.PolyCollection` constructor:

          agg_filter: a filter function, which takes a (m, n, 3) float array an
    d a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
    t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
    e]
          cmap: colormap or registered colormap name
```

```
  color: matplotlib color arg or sequence of rgba tuples
  contains: callable
  edgecolor: color or sequence of colors
  facecolor: color or sequence of colors
  figure: `.Figure`
  gid: str
  hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
  in_layout: bool
  joinstyle: {'miter', 'round', 'bevel'}
  label: object
  linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
  linewidth: float or sequence of floats
  norm: `.Normalize`
  offset_position: {'screen', 'data'}
  offsets: float or sequence of floats
  path_effects: `.AbstractPathEffect`
  picker: None or bool or float or callable
  pickradius: unknown
  rasterized: bool or None
  sketch_params: (scale: float, length: float, randomness: float)
  snap: bool or None
  transform: `.Transform`
  url: str
  urls: List[str] or None
  visible: bool
  zorder: float

See Also
--------
pcolormesh : for an explanation of the differences between
    pcolor and pcolormesh.
imshow : If *X* and *Y* are each equidistant, `~.Axes.imshow` can be a
    faster alternative.

Notes
-----

**Masked arrays**

*X*, *Y* and *C* may be masked arrays. If either ``C[i, j]``, or one
of the vertices surrounding ``C[i,j]`` (*X* or *Y* at
``[i, j], [i+1, j], [i, j+1], [i+1, j+1]``) is masked, nothing is
plotted.

.. _axes-pcolor-grid-orientation:

**Grid orientation**

The grid orientation follows the standard matrix convention: An array
*C* with shape (nrows, ncolumns) is plotted with the column number as
*X* and the row number as *Y*.

**Handling of pcolor() end-cases**

``pcolor()`` displays all columns of *C* if *X* and *Y* are not
specified, or if *X* and *Y* have one more column than *C*.
If *X* and *Y* have the same number of columns as *C* then the last
```

```
        column of *C* is dropped. Similarly for the rows.

        Note: This behavior is different from MATLAB's ``pcolor()``, which
        always discards the last row and column of *C*.

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    pcolormesh(*args, alpha=None, norm=None, cmap=None, vmin=None, vmax=None, s
hading='flat', antialiased=False, data=None, **kwargs)
        Create a pseudocolor plot with a non-regular rectangular grid.

        Call signature::

            pcolor([X, Y,] C, **kwargs)

        *X* and *Y* can be used to specify the corners of the quadrilaterals.

        .. note::

           ``pcolormesh()`` is similar to :func:`~Axes.pcolor`. It's much
           faster and preferred in most cases. For a detailed discussion on
           the differences see
           :ref:`Differences between pcolor() and pcolormesh()
           <differences-pcolor-pcolormesh>`.

        Parameters
        ----------
        C : array_like
            A scalar 2-D array. The values will be color-mapped.

        X, Y : array_like, optional
            The coordinates of the quadrilateral corners. The quadrilateral
            for ``C[i,j]`` has corners at::

                (X[i+1, j], Y[i+1, j])          (X[i+1, j+1], Y[i+1, j+1])
                                      +--------+
                                      | C[i,j] |
                                      +--------+
                    (X[i, j], Y[i, j])          (X[i, j+1], Y[i, j+1]),

            Note that the column index corresponds to the
            x-coordinate, and the row index corresponds to y. For
            details, see the :ref:`Notes <axes-pcolormesh-grid-orientation>`
            section below.

            The dimensions of *X* and *Y* should be one greater than those of
```

```
    *C*. Alternatively, *X*, *Y* and *C* may have equal dimensions, in
    which case the last row and column of *C* will be ignored.

    If *X* and/or *Y* are 1-D arrays or column vectors they will be
    expanded as needed into the appropriate 2-D arrays, making a
    rectangular grid.

cmap : str or `~matplotlib.colors.Colormap`, optional
    A Colormap instance or registered colormap name. The colormap
    maps the *C* values to colors. Defaults to :rc:`image.cmap`.

norm : `~matplotlib.colors.Normalize`, optional
    The Normalize instance scales the data values to the canonical
    colormap range [0, 1] for mapping to colors. By default, the data
    range is mapped to the colorbar range using linear scaling.

vmin, vmax : scalar, optional, default: None
    The colorbar range. If *None*, suitable min/max values are
    automatically chosen by the `~.Normalize` instance (defaults to
    the respective min/max values of *C* in case of the default linear
    scaling).

edgecolors : {'none', None, 'face', color, color sequence}, optional
    The color of the edges. Defaults to 'none'. Possible values:

    - 'none' or '': No edge.
    - *None*: :rc:`patch.edgecolor` will be used. Note that currently
      :rc:`patch.force_edgecolor` has to be True for this to work.
    - 'face': Use the adjacent face color.
    - An mpl color or sequence of colors will set the edge color.

    The singular form *edgecolor* works as an alias.

alpha : scalar, optional, default: None
    The alpha blending value, between 0 (transparent) and 1 (opaque).

shading : {'flat', 'gouraud'}, optional
    The fill style, Possible values:

    - 'flat': A solid color is used for each quad. The color of the
      quad (i, j), (i+1, j), (i, j+1), (i+1, j+1) is given by
      ``C[i,j]``.
    - 'gouraud': Each quad will be Gouraud shaded: The color of the
      corners (i', j') are given by ``C[i',j']``. The color values of
      the area in between is interpolated from the corner values.
      When Gouraud shading is used, *edgecolors* is ignored.

snap : bool, optional, default: False
    Whether to snap the mesh to pixel boundaries.

Returns
-------
mesh : `matplotlib.collections.QuadMesh`

Other Parameters
----------------
**kwargs
```

```
            Additionally, the following arguments are allowed. They are passed
            along to the `~matplotlib.collections.QuadMesh` constructor:

          agg_filter: a filter function, which takes a (m, n, 3) float array an
d a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float


        See Also
        --------
        pcolor : An alternative implementation with slightly different
            features. For a detailed discussion on the differences see
            :ref:`Differences between pcolor() and pcolormesh()
            <differences-pcolor-pcolormesh>`.
        imshow : If *X* and *Y* are each equidistant, `~.Axes.imshow` can be a
            faster alternative.

        Notes
        -----
```

```
            **Masked arrays**

            *C* may be a masked array. If ``C[i, j]`` is masked, the corresponding
            quadrilateral will be transparent. Masking of *X* and *Y* is not
            supported. Use `~.Axes.pcolor` if you need this functionality.

            .. _axes-pcolormesh-grid-orientation:

            **Grid orientation**

            The grid orientation follows the standard matrix convention: An array
            *C* with shape (nrows, ncolumns) is plotted with the column number as
            *X* and the row number as *Y*.

            .. _differences-pcolor-pcolormesh:

            **Differences between pcolor() and pcolormesh()**

            Both methods are used to create a pseudocolor plot of a 2-D array
            using quadrilaterals.

            The main difference lies in the created object and internal data
            handling:
            While `~.Axes.pcolor` returns a `.PolyCollection`, `~.Axes.pcolormesh`
            returns a `.QuadMesh`. The latter is more specialized for the given
            purpose and thus is faster. It should almost always be preferred.

            There is also a slight difference in the handling of masked arrays.
            Both `~.Axes.pcolor` and `~.Axes.pcolormesh` support masked arrays
            for *C*. However, only `~.Axes.pcolor` supports masked arrays for *X*
            and *Y*. The reason lies in the internal handling of the masked values.
            `~.Axes.pcolor` leaves out the respective polygons from the
            PolyCollection. `~.Axes.pcolormesh` sets the facecolor of the masked
            elements to transparent. You can see the difference when using
            edgecolors. While all edges are drawn irrespective of masking in a
            QuadMesh, the edge between two adjacent masked quadrilaterals in
            `~.Axes.pcolor` is not drawn as the corresponding polygons do not
            exist in the PolyCollection.

            Another difference is the support of Gouraud shading in
            `~.Axes.pcolormesh`, which is not available with `~.Axes.pcolor`.

            .. note::
                In addition to the above described arguments, this function can tak
e a
                **data** keyword argument. If such a **data** argument is given, th
e
                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``).

    phase_spectrum(x, Fs=None, Fc=None, window=None, pad_to=None, sides=None,
 *, data=None, **kwargs)
```

```
Plot the phase spectrum.

Call signature::

  phase_spectrum(x, Fs=2, Fc=0,  window=mlab.window_hanning,
                 pad_to=None, sides='default', **kwargs)

Compute the phase spectrum (unwrapped angle spectrum) of *x*.
Data is padded to a length of *pad_to* and the windowing function
*window* is applied to the signal.

Parameters
----------
x : 1-D array or sequence
    Array or sequence containing the data

Fs : scalar
    The sampling frequency (samples per time unit).  It is used
    to calculate the Fourier frequencies, freqs, in cycles per time
    unit. The default value is 2.

window : callable or ndarray
    A function or a vector of length *NFFT*. To create window
    vectors see :func:`window_hanning`, :func:`window_none`,
    :func:`numpy.blackman`, :func:`numpy.hamming`,
    :func:`numpy.bartlett`, :func:`scipy.signal`,
    :func:`scipy.signal.get_window`, etc. The default is
    :func:`window_hanning`.  If a function is passed as the
    argument, it must take a data segment as an argument and
    return the windowed version of the segment.

sides : {'default', 'onesided', 'twosided'}
    Specifies which sides of the spectrum to return.  Default gives the
    default behavior, which returns one-sided for real data and both
    for complex data.  'onesided' forces the return of a one-sided
    spectrum, while 'twosided' forces two-sided.

pad_to : int
    The number of points to which the data segment is padded when
    performing the FFT.  While not increasing the actual resolution of
    the spectrum (the minimum distance between resolvable peaks),
    this can give more points in the plot, allowing for more
    detail. This corresponds to the *n* parameter in the call to fft().
    The default is None, which sets *pad_to* equal to the length of the
    input signal (i.e. no padding).

Fc : int
    The center frequency of *x* (defaults to 0), which offsets
    the x extents of the plot to reflect the frequency range used
    when a signal is acquired and then filtered and downsampled to
    baseband.

Returns
-------
spectrum : 1-D array
    The values for the phase spectrum in radians (real valued).
```

```
        freqs : 1-D array
            The frequencies corresponding to the elements in *spectrum*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.

        Other Parameters
        ----------------
        **kwargs :
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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
```

```
        See Also
        --------
        :func:`magnitude_spectrum`
            :func:`magnitude_spectrum` plots the magnitudes of the
            corresponding frequencies.

        :func:`angle_spectrum`
            :func:`angle_spectrum` plots the wrapped version of this function.

        :func:`specgram`
            :func:`specgram` can plot the phase spectrum of segments within the
            signal in a colormap.

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

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

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

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

    pie(x, explode=None, labels=None, colors=None, autopct=None, pctdistance=0.
6, shadow=False, labeldistance=1.1, startangle=None, radius=None, counterclock=
True, wedgeprops=None, textprops=None, center=(0, 0), frame=False, rotatelabels
=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 tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            following arguments are replaced by **data[<arg>]**:

            * All arguments with the following names: 'colors', 'explode', 'lab
els', 'x'.

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

    pink()
        Set the colormap to "pink".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    plasma()
        Set the colormap to "plasma".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    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 arra
y 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` | Non
e]
          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 tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    plot_date(x, y, fmt='o', tz=None, xdate=True, ydate=False, *, data=None, **
kwargs)
        Plot data that contains dates.

        Similar to `.plot`, this plots *y* vs. *x* as lines or markers.
        However, the axis labels are formatted as dates depending on *xdate*
        and *ydate*.

        Parameters
        ----------
        x, y : array-like
            The coordinates of the data points. If *xdate* or *ydate* is
            *True*, the respective values *x* or *y* are interpreted as
            :ref:`Matplotlib dates <date-format>`.

        fmt : str, optional
            The plot format string. For details, see the corresponding
            parameter in `.plot`.

        tz : [ *None* | timezone string | :class:`tzinfo` instance]
```

```
            The time zone to use in labeling dates. If *None*, defaults to
            rcParam ``timezone``.

        xdate : bool, optional, default: True
            If *True*, the *x*-axis will be interpreted as Matplotlib dates.

        ydate : bool, optional, default: False
            If *True*, the *y*-axis will be interpreted as Matplotlib dates.


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


        Other Parameters
        ----------------
        **kwargs
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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


        See Also
        --------
        matplotlib.dates : Helper functions on dates.
        matplotlib.dates.date2num : Convert dates to num.
        matplotlib.dates.num2date : Convert num to dates.
        matplotlib.dates.drange : Create an equally spaced sequence of dates.


        Notes
        -----
        If you are using custom date tickers and formatters, it may be
        necessary to set the formatters/locators after the call to
        `.plot_date`. `.plot_date` will set the default tick locator to
        `.AutoDateLocator` (if the tick locator is not already set to a
        `.DateLocator` instance) and the default tick formatter to
        `.AutoDateFormatter` (if the tick formatter is not already set to a
        `.DateFormatter` instance).

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    plotfile(fname, cols=(0,), plotfuncs=None, comments='#', skiprows=0, checkr
ows=5, delimiter=',', names=None, subplots=True, newfig=True, **kwargs)
        Plot the data in a file.

        *cols* is a sequence of column identifiers to plot.  An identifier
        is either an int or a string.  If it is an int, it indicates the
        column number.  If it is a string, it indicates the column header.
        matplotlib will make column headers lower case, replace spaces with
        underscores, and remove all illegal characters; so ``'Adj Close*'``
        will have name ``'adj_close'``.

        - If len(*cols*) == 1, only that column will be plotted on the *y* axi
s.

        - If len(*cols*) > 1, the first element will be an identifier for
```

```
          data for the *x* axis and the remaining elements will be the
          column indexes for multiple subplots if *subplots* is *True*
          (the default), or for lines in a single subplot if *subplots*
          is *False*.

        *plotfuncs*, if not *None*, is a dictionary mapping identifier to
        an :class:`~matplotlib.axes.Axes` plotting function as a string.
        Default is 'plot', other choices are 'semilogy', 'fill', 'bar',
        etc.  You must use the same type of identifier in the *cols*
        vector as you use in the *plotfuncs* dictionary, e.g., integer
        column numbers in both or column names in both. If *subplots*
        is *False*, then including any function such as 'semilogy'
        that changes the axis scaling will set the scaling for all
        columns.

        *comments*, *skiprows*, *checkrows*, *delimiter*, and *names*
        are all passed on to :func:`matplotlib.mlab.csv2rec` to
        load the data into a record array.

        If *newfig* is *True*, the plot always will be made in a new figure;
        if *False*, it will be made in the current figure if one exists,
        else in a new figure.

        kwargs are passed on to plotting functions.

        Example usage::

          # plot the 2nd and 4th column against the 1st in two subplots
          plotfile(fname, (0,1,3))

          # plot using column names; specify an alternate plot type for volume
          plotfile(fname, ('date', 'volume', 'adj_close'),
                                       plotfuncs={'volume': 'semilogy'})

        Note: plotfile is intended as a convenience for quickly plotting
        data from flat files; it is not intended as an alternative
        interface to general plotting with pyplot or matplotlib.

    plotting()
        ============================ ========================================
=============================================================================
====
        Function                     Description
        ============================ ========================================
=============================================================================
====
        `acorr`                      Plot the autocorrelation of *x*.
        `angle_spectrum`             Plot the angle spectrum.
        `annotate`                   Annotate the point *xy* with text *s*.
        `arrow`                      Add an arrow to the axes.
        `autoscale`                  Autoscale the axis view to the data (toggl
e).
        `axes`                       Add an axes to the current figure and make
it the current axes.
        `axhline`                    Add a horizontal line across the axis.
        `axhspan`                    Add a horizontal span (rectangle) across t
he axis.
```

```
        `axis`                      Convenience method to get or set some axis
properties.
        `axvline`                   Add a vertical line across the axes.
        `axvspan`                   Add a vertical span (rectangle) across the
axes.
        `bar`                       Make a bar plot.
        `barbs`                     Plot a 2-D field of barbs.
        `barh`                      Make a horizontal bar plot.
        `box`                       Turn the axes box on or off on the current
axes.
        `boxplot`                   Make a box and whisker plot.
        `broken_barh`               Plot a horizontal sequence of rectangles.
        `cla`                       Clear the current axes.
        `clabel`                    Label a contour plot.
        `clf`                       Clear the current figure.
        `clim`                      Set the color limits of the current image.
        `close`                     Close a figure window.
        `cohere`                    Plot the coherence between *x* and *y*.
        `colorbar`                  Add a colorbar to a plot.
        `contour`                   Plot contours.
        `contourf`                  Plot contours.
        `csd`                       Plot the cross-spectral density.
        `delaxes`                   Remove the `Axes` *ax* (defaulting to the
 current axes) from its figure.
        `draw`                      Redraw the current figure.
        `errorbar`                  Plot y versus x as lines and/or markers wi
th attached errorbars.
        `eventplot`                 Plot identical parallel lines at the given
positions.
        `figimage`                  Add a non-resampled image to the figure.
        `figlegend`                 Place a legend in the figure.
        `fignum_exists`             Return whether the figure with the given i
d exists.
        `figtext`                   Add text to figure.
        `figure`                    Create a new figure.
        `fill`                      Plot filled polygons.
        `fill_between`              Fill the area between two horizontal curve
s.
        `fill_betweenx`             Fill the area between two vertical curves.
        `findobj`                   Find artist objects.
        `gca`                       Get the current :class:`~matplotlib.axes.A
xes` instance on the current figure matching the given keyword args, or create
 one.
        `gcf`                       Get a reference to the current figure.
        `gci`                       Get the current colorable artist.
        `get_figlabels`             Return a list of existing figure labels.
        `get_fignums`               Return a list of existing figure numbers.
        `grid`                      Configure the grid lines.
        `hexbin`                    Make a hexagonal binning plot.
        `hist`                      Plot a histogram.
        `hist2d`                    Make a 2D histogram plot.
        `hlines`                    Plot horizontal lines at each *y* from *xm
in* to *xmax*.
        `imread`                    Read an image from a file into an array.
        `imsave`                    Save an array as in image file.
        `imshow`                    Display an image, i.e.
        `install_repl_displayhook`  Install a repl display hook so that any st
```

```
ale figure are automatically redrawn when control is returned to the repl.
        `ioff`                      Turn the interactive mode off.
        `ion`                       Turn the interactive mode on.
        `isinteractive`             Return the status of interactive mode.
        `legend`                    Place a legend on the axes.
        `locator_params`            Control behavior of tick locators.
        `loglog`                    Make a plot with log scaling on both the x
and y axis.
        `magnitude_spectrum`        Plot the magnitude spectrum.
        `margins`                   Set or retrieve autoscaling margins.
        `matshow`                   Display an array as a matrix in a new figu
re window.
        `minorticks_off`            Remove minor ticks from the axes.
        `minorticks_on`             Display minor ticks on the axes.
        `pause`                     Pause for *interval* seconds.
        `pcolor`                    Create a pseudocolor plot with a non-regul
ar rectangular grid.
        `pcolormesh`                Create a pseudocolor plot with a non-regul
ar rectangular grid.
        `phase_spectrum`            Plot the phase spectrum.
        `pie`                       Plot a pie chart.
        `plot`                      Plot y versus x as lines and/or markers.
        `plot_date`                 Plot data that contains dates.
        `plotfile`                  Plot the data in a file.
        `polar`                     Make a polar plot.
        `psd`                       Plot the power spectral density.
        `quiver`                    Plot a 2-D field of arrows.
        `quiverkey`                 Add a key to a quiver plot.
        `rc`                        Set the current rc params.
        `rc_context`                Return a context manager for managing rc s
ettings.
        `rcdefaults`                Restore the rc params from Matplotlib's in
ternal default style.
        `rgrids`                    Get or set the radial gridlines on the cur
rent polar plot.
        `savefig`                   Save the current figure.
        `sca`                       Set the current Axes instance to *ax*.
        `scatter`                   A scatter plot of *y* vs *x* with varying
 marker size and/or color.
        `sci`                       Set the current image.
        `semilogx`                  Make a plot with log scaling on the x axi
s.
        `semilogy`                  Make a plot with log scaling on the y axi
s.
        `set_cmap`                  Set the default colormap.
        `setp`                      Set a property on an artist object.
        `show`                      Display a figure.
        `specgram`                  Plot a spectrogram.
        `spy`                       Plot the sparsity pattern of a 2D array.
        `stackplot`                 Draw a stacked area plot.
        `stem`                      Create a stem plot.
        `step`                      Make a step plot.
        `streamplot`                Draw streamlines of a vector flow.
        `subplot`                   Add a subplot to the current figure.
        `subplot2grid`              Create an axis at specific location inside
a regular grid.
        `subplot_tool`              Launch a subplot tool window for a figure.
```

```
        `subplots`                    Create a figure and a set of subplots.
        `subplots_adjust`             Tune the subplot layout.
        `suptitle`                    Add a centered title to the figure.
        `switch_backend`              Close all open figures and set the Matplot
lib backend.
        `table`                       Add a table to the current axes.
        `text`                        Add text to the axes.
        `thetagrids`                  Get or set the theta gridlines on the curr
ent polar plot.
        `tick_params`                 Change the appearance of ticks, tick label
s, and gridlines.
        `ticklabel_format`            Change the `~matplotlib.ticker.ScalarForma
tter` used by default for linear axes.
        `tight_layout`                Automatically adjust subplot parameters to
give specified padding.
        `title`                       Set a title for the axes.
        `tricontour`                  Draw contours on an unstructured triangula
r grid.
        `tricontourf`                 Draw contours on an unstructured triangula
r grid.
        `tripcolor`                   Create a pseudocolor plot of an unstructur
ed triangular grid.
        `triplot`                     Draw a unstructured triangular grid as lin
es and/or markers.
        `twinx`                       Make a second axes that shares the *x*-axi
s.
        `twiny`                       Make a second axes that shares the *y*-axi
s.
        `uninstall_repl_displayhook` Uninstall the matplotlib display hook.
        `violinplot`                  Make a violin plot.
        `vlines`                      Plot vertical lines.
        `xcorr`                       Plot the cross correlation between *x* and
*y*.
        `xkcd`                        Turn on `xkcd <https://xkcd.com/>`_ sketch
-style drawing mode.
        `xlabel`                      Set the label for the x-axis.
        `xlim`                        Get or set the x limits of the current axe
s.
        `xscale`                      Set the x-axis scale.
        `xticks`                      Get or set the current tick locations and
 labels of the x-axis.
        `ylabel`                      Set the label for the y-axis.
        `ylim`                        Get or set the y-limits of the current axe
s.
        `yscale`                      Set the y-axis scale.
        `yticks`                      Get or set the current tick locations and
 labels of the y-axis.
        ============================ ========================================
================================================================================
====

    polar(*args, **kwargs)
        Make a polar plot.

        call signature::

          polar(theta, r, **kwargs)
```

```
        Multiple *theta*, *r* arguments are supported, with format
        strings, as in :func:`~matplotlib.pyplot.plot`.

    prism()
        Set the colormap to "prism".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    psd(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverlap=Non
e, pad_to=None, sides=None, scale_by_freq=None, return_line=None, *, data=None,
**kwargs)
        Plot the power spectral density.

        Call signature::

          psd(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
              window=mlab.window_hanning, noverlap=0, pad_to=None,
              sides='default', scale_by_freq=None, return_line=None, **kwargs)

        The power spectral density :math:`P_{xx}` by Welch's average
        periodogram method.  The vector *x* is divided into *NFFT* length
        segments.  Each segment is detrended by function *detrend* and
        windowed by function *window*.  *noverlap* gives the length of
        the overlap between segments.  The :math:`|\mathrm{fft}(i)|^2`
        of each segment :math:`i` are averaged to compute :math:`P_{xx}`,
        with a scaling to correct for power loss due to windowing.

        If len(*x*) < *NFFT*, it will be zero padded to *NFFT*.

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*. To create window
            vectors see :func:`window_hanning`, :func:`window_none`,
            :func:`numpy.blackman`, :func:`numpy.hamming`,
            :func:`numpy.bartlett`, :func:`scipy.signal`,
            :func:`scipy.signal.get_window`, etc. The default is
            :func:`window_hanning`.  If a function is passed as the
            argument, it must take a data segment as an argument and
            return the windowed version of the segment.

        sides : {'default', 'onesided', 'twosided'}
            Specifies which sides of the spectrum to return.  Default gives the
            default behavior, which returns one-sided for real data and both
            for complex data.  'onesided' forces the return of a one-sided
            spectrum, while 'twosided' forces two-sided.
```

```
                pad_to : int
                    The number of points to which the data segment is padded when
                    performing the FFT.  This can be different from *NFFT*, which
                    specifies the number of data points used.  While not increasing
                    the actual resolution of the spectrum (the minimum distance between
                    resolvable peaks), this can give more points in the plot,
                    allowing for more detail. This corresponds to the *n* parameter
                    in the call to fft(). The default is None, which sets *pad_to*
                    equal to *NFFT*

                NFFT : int
                    The number of data points used in each block for the FFT.
                    A power 2 is most efficient.  The default value is 256.
                    This should *NOT* be used to get zero padding, or the scaling of th
e
                    result will be incorrect. Use *pad_to* for this instead.

                detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable
                    The function applied to each segment before fft-ing,
                    designed to remove the mean or linear trend.  Unlike in
                    MATLAB, where the *detrend* parameter is a vector, in
                    matplotlib is it a function.  The :mod:`~matplotlib.mlab`
                    module defines :func:`~matplotlib.mlab.detrend_none`,
                    :func:`~matplotlib.mlab.detrend_mean`, and
                    :func:`~matplotlib.mlab.detrend_linear`, but you can use
                    a custom function as well.  You can also use a string to choose
                    one of the functions.  'default', 'constant', and 'mean' call
                    :func:`~matplotlib.mlab.detrend_mean`.  'linear' calls
                    :func:`~matplotlib.mlab.detrend_linear`.  'none' calls
                    :func:`~matplotlib.mlab.detrend_none`.

                scale_by_freq : bool, optional
                    Specifies whether the resulting density values should be scaled
                    by the scaling frequency, which gives density in units of Hz^-1.
                    This allows for integration over the returned frequency values.
                    The default is True for MATLAB compatibility.

                noverlap : int
                    The number of points of overlap between segments.
                    The default value is 0 (no overlap).

                Fc : int
                    The center frequency of *x* (defaults to 0), which offsets
                    the x extents of the plot to reflect the frequency range used
                    when a signal is acquired and then filtered and downsampled to
                    baseband.

                return_line : bool
                    Whether to include the line object plotted in the returned values.
                    Default is False.

                Returns
                -------
                Pxx : 1-D array
                    The values for the power spectrum `P_{xx}` before scaling
                    (real valued).
```

```
        freqs : 1-D array
            The frequencies corresponding to the elements in *Pxx*.

        line : a :class:`~matplotlib.lines.Line2D` instance
            The line created by this function.
            Only returned if *return_line* is True.

        Other Parameters
        ----------------
        **kwargs :
            Keyword arguments control the :class:`~matplotlib.lines.Line2D`
            properties:

              agg_filter: a filter function, which takes a (m, n, 3) float arra
y 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` | Non
e]
          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

            See Also
            --------
            :func:`specgram`
                :func:`specgram` differs in the default overlap; in not returning
                the mean of the segment periodograms; in returning the times of the
                segments; and in plotting a colormap instead of a line.

            :func:`magnitude_spectrum`
                :func:`magnitude_spectrum` plots the magnitude spectrum.

            :func:`csd`
                :func:`csd` plots the spectral density between two signals.

            Notes
            -----
            For plotting, the power is plotted as
            :math:`10\log_{10}(P_{xx})` for decibels, though *Pxx* itself
            is returned.

            References
            ----------
            Bendat & Piersol -- Random Data: Analysis and Measurement Procedures,
            John Wiley & Sons (1986)

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

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

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

        quiver(*args, data=None, **kw)
            Plot a 2-D field of arrows.

            Call signatures::

              quiver(U, V, **kw)
              quiver(U, V, C, **kw)
              quiver(X, Y, U, V, **kw)
              quiver(X, Y, U, V, C, **kw)

            *U* and *V* are the arrow data, *X* and *Y* set the location of the
            arrows, and *C* sets the color of the arrows. These arguments may be 1-
    D or
            2-D arrays or sequences.

            If *X* and *Y* are absent, they will be generated as a uniform grid.
            If *U* and *V* are 2-D arrays and *X* and *Y* are 1-D, and if ``len(X)`
    ` and
```

```
        ``len(Y)`` match the column and row dimensions of *U*, then *X* and *Y*
will be
        expanded with :func:`numpy.meshgrid`.

        The default settings auto-scales the length of the arrows to a reasonab
le size.
        To change this behavior see the *scale* and *scale_units* kwargs.

        The defaults give a slightly swept-back arrow; to make the head a
        triangle, make *headaxislength* the same as *headlength*. To make the
        arrow more pointed, reduce *headwidth* or increase *headlength* and
        *headaxislength*. To make the head smaller relative to the shaft,
        scale down all the head parameters. You will probably do best to leave
        minshaft alone.

        *linewidths* and *edgecolors* can be used to customize the arrow
        outlines.

        Parameters
        ----------
        X : 1D or 2D array, sequence, optional
            The x coordinates of the arrow locations
        Y : 1D or 2D array, sequence, optional
            The y coordinates of the arrow locations
        U : 1D or 2D array or masked array, sequence
            The x components of the arrow vectors
        V : 1D or 2D array or masked array, sequence
            The y components of the arrow vectors
        C : 1D or 2D array, sequence, optional
            The arrow colors
        units : [ 'width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'xy' ]
            The arrow dimensions (except for *length*) are measured in multiple
s of
            this unit.

            'width' or 'height': the width or height of the axis

            'dots' or 'inches': pixels or inches, based on the figure dpi

            'x', 'y', or 'xy': respectively *X*, *Y*, or :math:`\sqrt{X^2 + Y^
2}`
            in data units

            The arrows scale differently depending on the units.  For
            'x' or 'y', the arrows get larger as one zooms in; for other
            units, the arrow size is independent of the zoom state.  For
            'width or 'height', the arrow size increases with the width and
            height of the axes, respectively, when the window is resized;
            for 'dots' or 'inches', resizing does not change the arrows.
        angles : [ 'uv' | 'xy' ], array, optional
            Method for determining the angle of the arrows. Default is 'uv'.

            'uv': the arrow axis aspect ratio is 1 so that
            if *U*==*V* the orientation of the arrow on the plot is 45 degrees
            counter-clockwise from the horizontal axis (positive to the right).

            'xy': arrows point from (x,y) to (x+u, y+v).
```

```
            Use this for plotting a gradient field, for example.

            Alternatively, arbitrary angles may be specified as an array
            of values in degrees, counter-clockwise from the horizontal axis.

            Note: inverting a data axis will correspondingly invert the
            arrows only with ``angles='xy'``.
        scale : None, float, optional
            Number of data units per arrow length unit, e.g., m/s per plot widt
h; a
            smaller scale parameter makes the arrow longer. Default is *None*.

            If *None*, a simple autoscaling algorithm is used, based on the ave
rage
            vector length and the number of vectors. The arrow length unit is g
iven by
            the *scale_units* parameter
        scale_units : [ 'width' | 'height' | 'dots' | 'inches' | 'x' | 'y' | 'x
y' ], None, optional
            If the *scale* kwarg is *None*, the arrow length unit. Default is *
None*.

            e.g. *scale_units* is 'inches', *scale* is 2.0, and
            ``(u,v) = (1,0)``, then the vector will be 0.5 inches long.

            If *scale_units* is 'width'/'height', then the vector will be half
 the
            width/height of the axes.

            If *scale_units* is 'x' then the vector will be 0.5 x-axis
            units. To plot vectors in the x-y plane, with u and v having
            the same units as x and y, use
            ``angles='xy', scale_units='xy', scale=1``.
        width : scalar, optional
            Shaft width in arrow units; default depends on choice of units,
            above, and number of vectors; a typical starting value is about
            0.005 times the width of the plot.
        headwidth : scalar, optional
            Head width as multiple of shaft width, default is 3
        headlength : scalar, optional
            Head length as multiple of shaft width, default is 5
        headaxislength : scalar, optional
            Head length at shaft intersection, default is 4.5
        minshaft : scalar, optional
            Length below which arrow scales, in units of head length. Do not
            set this to less than 1, or small arrows will look terrible!
            Default is 1
        minlength : scalar, optional
            Minimum length as a multiple of shaft width; if an arrow length
            is less than this, plot a dot (hexagon) of this diameter instead.
            Default is 1.
        pivot : [ 'tail' | 'mid' | 'middle' | 'tip' ], optional
            The part of the arrow that is at the grid point; the arrow rotates
            about this point, hence the name *pivot*.
        color : [ color | color sequence ], optional
            This is a synonym for the
            :class:`~matplotlib.collections.PolyCollection` facecolor kwarg.
```

```
            If *C* has been set, *color* has no effect.

        Notes
        -----
        Additional :class:`~matplotlib.collections.PolyCollection`
        keyword arguments:

          agg_filter: a filter function, which takes a (m, n, 3) float array an
d a dpi value, and returns a (m, n, 3) array
          alpha: float or None
          animated: bool
          antialiased: bool or sequence of bools
          array: ndarray
          capstyle: {'butt', 'round', 'projecting'}
          clim: a length 2 sequence of floats; may be overridden in methods tha
t have ``vmin`` and ``vmax`` kwargs.
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          cmap: colormap or registered colormap name
          color: matplotlib color arg or sequence of rgba tuples
          contains: callable
          edgecolor: color or sequence of colors
          facecolor: color or sequence of colors
          figure: `.Figure`
          gid: str
          hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
          in_layout: bool
          joinstyle: {'miter', 'round', 'bevel'}
          label: object
          linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
          linewidth: float or sequence of floats
          norm: `.Normalize`
          offset_position: {'screen', 'data'}
          offsets: float or sequence of floats
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          pickradius: unknown
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          urls: List[str] or None
          visible: bool
          zorder: float

        See Also
        --------
        quiverkey : Add a key to a quiver plot

    quiverkey(Q, X, Y, U, label, **kw)
        Add a key to a quiver plot.

        Call signature::
```

```
    quiverkey(Q, X, Y, U, label, **kw)

  Arguments:

    *Q*:
      The Quiver instance returned by a call to quiver.

    *X*, *Y*:
      The location of the key; additional explanation follows.

    *U*:
      The length of the key

    *label*:
      A string with the length and units of the key

  Keyword arguments:

    *angle* = 0
      The angle of the key arrow. Measured in degrees anti-clockwise from
the
      x-axis.

    *coordinates* = [ 'axes' | 'figure' | 'data' | 'inches' ]
      Coordinate system and units for *X*, *Y*: 'axes' and 'figure' are
      normalized coordinate systems with 0,0 in the lower left and 1,1
      in the upper right; 'data' are the axes data coordinates (used for
      the locations of the vectors in the quiver plot itself); 'inches'
      is position in the figure in inches, with 0,0 at the lower left
      corner.

    *color*:
      overrides face and edge colors from *Q*.

    *labelpos* = [ 'N' | 'S' | 'E' | 'W' ]
      Position the label above, below, to the right, to the left of the
      arrow, respectively.

    *labelsep*:
      Distance in inches between the arrow and the label.  Default is
      0.1

    *labelcolor*:
      defaults to default :class:`~matplotlib.text.Text` color.

    *fontproperties*:
      A dictionary with keyword arguments accepted by the
      :class:`~matplotlib.font_manager.FontProperties` initializer:
      *family*, *style*, *variant*, *size*, *weight*

  Any additional keyword arguments are used to override vector
  properties taken from *Q*.

  The positioning of the key depends on *X*, *Y*, *coordinates*, and
  *labelpos*.  If *labelpos* is 'N' or 'S', *X*, *Y* give the position
  of the middle of the key arrow.  If *labelpos* is 'E', *X*, *Y*
  positions the head, and if *labelpos* is 'W', *X*, *Y* positions the
```

```
    tail; in either of these two cases, *X*, *Y* is somewhere in the
    middle of the arrow+label key object.

rc(group, **kwargs)
    Set the current rc params.  *group* is the grouping for the rc, e.g.,
    for ``lines.linewidth`` the group is ``lines``, for
    ``axes.facecolor``, the group is ``axes``, and so on.  Group may
    also be a list or tuple of group names, e.g., (*xtick*, *ytick*).
    *kwargs* is a dictionary attribute name/value pairs, e.g.,::

      rc('lines', linewidth=2, color='r')

    sets the current rc params and is equivalent to::

      rcParams['lines.linewidth'] = 2
      rcParams['lines.color'] = 'r'

    The following aliases are available to save typing for interactive
    users:

    =====   =================
    Alias   Property
    =====   =================
    'lw'    'linewidth'
    'ls'    'linestyle'
    'c'     'color'
    'fc'    'facecolor'
    'ec'    'edgecolor'
    'mew'   'markeredgewidth'
    'aa'    'antialiased'
    =====   =================

    Thus you could abbreviate the above rc command as::

          rc('lines', lw=2, c='r')


    Note you can use python's kwargs dictionary facility to store
    dictionaries of default parameters.  e.g., you can customize the
    font rc as follows::

      font = {'family' : 'monospace',
              'weight' : 'bold',
              'size'   : 'larger'}

      rc('font', **font)  # pass in the font dict as kwargs

    This enables you to easily switch between several configurations.  Use
    ``matplotlib.style.use('default')`` or :func:`~matplotlib.rcdefaults` t
o
    restore the default rc params after changes.

rc_context(rc=None, fname=None)
    Return a context manager for managing rc settings.

    This allows one to do::
```

```
                with mpl.rc_context(fname='screen.rc'):
                    plt.plot(x, a)
                    with mpl.rc_context(fname='print.rc'):
                        plt.plot(x, b)
                    plt.plot(x, c)

            The 'a' vs 'x' and 'c' vs 'x' plots would have settings from
            'screen.rc', while the 'b' vs 'x' plot would have settings from
            'print.rc'.

            A dictionary can also be passed to the context manager::

                with mpl.rc_context(rc={'text.usetex': True}, fname='screen.rc'):
                    plt.plot(x, a)

            The 'rc' dictionary takes precedence over the settings loaded from
            'fname'.  Passing a dictionary only is also valid. For example a
            common usage is::

                with mpl.rc_context(rc={'interactive': False}):
                    fig, ax = plt.subplots()
                    ax.plot(range(3), range(3))
                    fig.savefig('A.png', format='png')
                    plt.close(fig)

        rcdefaults()
            Restore the rc params from Matplotlib's internal default style.

            Style-blacklisted rc params (defined in
            `matplotlib.style.core.STYLE_BLACKLIST`) are not updated.

            See Also
            --------
            rc_file_defaults :
                Restore the rc params from the rc file originally loaded by Matplot
    lib.
            matplotlib.style.use :
                Use a specific style file.  Call ``style.use('default')`` to restor
    e
                the default style.

        rgrids(*args, **kwargs)
            Get or set the radial gridlines on the current polar plot.

            Call signatures::

             lines, labels = rgrids()
             lines, labels = rgrids(radii, labels=None, angle=22.5, fmt=None, **kwa
    rgs)

            When called with no arguments, `.rgrids` simply returns the tuple
            (*lines*, *labels*). When called with arguments, the labels will
            appear at the specified radial distances and angle.

            Parameters
            ----------
            radii : tuple with floats
```

```
        The radii for the radial gridlines

    labels : tuple with strings or None
        The labels to use at each radial gridline. The
        `matplotlib.ticker.ScalarFormatter` will be used if None.

    angle : float
        The angular position of the radius labels in degrees.

    fmt : str or None
        Format string used in `matplotlib.ticker.FormatStrFormatter`.
        For example '%f'.

    Returns
    -------
    lines, labels : list of `.lines.Line2D`, list of `.text.Text`
        *lines* are the radial gridlines and *labels* are the tick labels.

    Other Parameters
    ----------------
    **kwargs
        *kwargs* are optional `~.Text` properties for the labels.

    Examples
    --------
    ::

      # set the locations of the radial gridlines
      lines, labels = rgrids( (0.25, 0.5, 1.0) )

      # set the locations and labels of the radial gridlines
      lines, labels = rgrids( (0.25, 0.5, 1.0), ('Tom', 'Dick', 'Harry' ))

    See Also
    --------
    .pyplot.thetagrids
    .projections.polar.PolarAxes.set_rgrids
    .Axis.get_gridlines
    .Axis.get_ticklabels

savefig(*args, **kwargs)
    Save the current figure.

    Call signature::

      savefig(fname, dpi=None, facecolor='w', edgecolor='w',
              orientation='portrait', papertype=None, format=None,
              transparent=False, bbox_inches=None, pad_inches=0.1,
              frameon=None, metadata=None)

    The output formats available depend on the backend being used.

    Parameters
    ----------

    fname : str or file-like object
        A string containing a path to a filename, or a Python
```

```
    file-like object, or possibly some backend-dependent object
    such as :class:`~matplotlib.backends.backend_pdf.PdfPages`.

    If *format* is *None* and *fname* is a string, the output
    format is deduced from the extension of the filename. If
    the filename has no extension, :rc:`savefig.format` is used.

    If *fname* is not a string, remember to specify *format* to
    ensure that the correct backend is used.

Other Parameters
----------------

dpi : [ *None* | scalar > 0 | 'figure' ]
    The resolution in dots per inch.  If *None*, defaults to
    :rc:`savefig.dpi`.  If 'figure', uses the figure's dpi value.

quality : [ *None* | 1 <= scalar <= 100 ]
    The image quality, on a scale from 1 (worst) to 95 (best).
    Applicable only if *format* is jpg or jpeg, ignored otherwise.
    If *None*, defaults to :rc:`savefig.jpeg_quality` (95 by default).
    Values above 95 should be avoided; 100 completely disables the
    JPEG quantization stage.

facecolor : color spec or None, optional
    The facecolor of the figure; if *None*, defaults to
    :rc:`savefig.facecolor`.

edgecolor : color spec or None, optional
    The edgecolor of the figure; if *None*, defaults to
    :rc:`savefig.edgecolor`

orientation : {'landscape', 'portrait'}
    Currently only supported by the postscript backend.

papertype : str
    One of 'letter', 'legal', 'executive', 'ledger', 'a0' through
    'a10', 'b0' through 'b10'. Only supported for postscript
    output.

format : str
    One of the file extensions supported by the active
    backend.  Most backends support png, pdf, ps, eps and svg.

transparent : bool
    If *True*, the axes patches will all be transparent; the
    figure patch will also be transparent unless facecolor
    and/or edgecolor are specified via kwargs.
    This is useful, for example, for displaying
    a plot on top of a colored background on a web page.  The
    transparency of these patches will be restored to their
    original values upon exit of this function.

frameon : bool
    If *True*, the figure patch will be colored, if *False*, the
    figure background will be transparent.  If not provided, the
    rcParam 'savefig.frameon' will be used.
```

```
            bbox_inches : str or `~matplotlib.transforms.Bbox`, optional
                Bbox in inches. Only the given portion of the figure is
                saved. If 'tight', try to figure out the tight bbox of
                the figure. If None, use savefig.bbox

            pad_inches : scalar, optional
                Amount of padding around the figure when bbox_inches is
                'tight'. If None, use savefig.pad_inches

            bbox_extra_artists : list of `~matplotlib.artist.Artist`, optional
                A list of extra artists that will be considered when the
                tight bbox is calculated.

            metadata : dict, optional
                Key/value pairs to store in the image metadata. The supported keys
                and defaults depend on the image format and backend:

                - 'png' with Agg backend: See the parameter ``metadata`` of
                  `~.FigureCanvasAgg.print_png`.
                - 'pdf' with pdf backend: See the parameter ``metadata`` of
                  `~.backend_pdf.PdfPages`.
                - 'eps' and 'ps' with PS backend: Only 'Creator' is supported.

        sca(ax)
            Set the current Axes instance to *ax*.

            The current Figure is updated to the parent of *ax*.

        scatter(x, y, s=None, c=None, marker=None, cmap=None, norm=None, vmin=None,
    vmax=None, alpha=None, linewidths=None, verts=None, edgecolors=None, *, data=No
    ne, **kwargs)
            A scatter plot of *y* vs *x* with varying marker size and/or color.

            Parameters
            ----------
            x, y : array_like, shape (n, )
                The data positions.

            s : scalar or array_like, shape (n, ), optional
                The marker size in points**2.
                Default is ``rcParams['lines.markersize'] ** 2``.

            c : color, sequence, or sequence of color, optional
                The marker color. Possible values:

                - A single color format string.
                - A sequence of color specifications of length n.
                - A sequence of n numbers to be mapped to colors using *cmap* and
                  *norm*.
                - A 2-D array in which the rows are RGB or RGBA.

                Note that *c* should not be a single numeric RGB or RGBA sequence
                because that is indistinguishable from an array of values to be
                colormapped. If you want to specify the same RGB or RGBA value for
                all points, use a 2-D array with a single row.  Otherwise, value-
                matching will have precedence in case of a size matching with *x*
```

```
    and *y*.

    Defaults to ``None``. In that case the marker color is determined
    by the value of ``color``, ``facecolor`` or ``facecolors``. In case
    those are not specified or ``None``, the marker color is determined
    by the next color of the ``Axes``' current "shape and fill" color
    cycle. This cycle defaults to :rc:`axes.prop_cycle`.

marker : `~matplotlib.markers.MarkerStyle`, optional
    The marker style. *marker* can be either an instance of the class
    or the text shorthand for a particular marker.
    Defaults to ``None``, in which case it takes the value of
    :rc:`scatter.marker` = 'o'.
    See `~matplotlib.markers` for more information about marker styles.

cmap : `~matplotlib.colors.Colormap`, optional, default: None
    A `.Colormap` instance or registered colormap name. *cmap* is only
    used if *c* is an array of floats. If ``None``, defaults to rc
    ``image.cmap``.

norm : `~matplotlib.colors.Normalize`, optional, default: None
    A `.Normalize` instance is used to scale luminance data to 0, 1.
    *norm* is only used if *c* is an array of floats. If *None*, use
    the default `.colors.Normalize`.

vmin, vmax : scalar, optional, default: None
    *vmin* and *vmax* are used in conjunction with *norm* to normalize
    luminance data. If None, the respective min and max of the color
    array is used. *vmin* and *vmax* are ignored if you pass a *norm*
    instance.

alpha : scalar, optional, default: None
    The alpha blending value, between 0 (transparent) and 1 (opaque).

linewidths : scalar or array_like, optional, default: None
    The linewidth of the marker edges. Note: The default *edgecolors*
    is 'face'. You may want to change this as well.
    If *None*, defaults to rcParams ``lines.linewidth``.

edgecolors : color or sequence of color, optional, default: 'face'
    The edge color of the marker. Possible values:

    - 'face': The edge color will always be the same as the face color.
    - 'none': No patch boundary will be drawn.
    - A matplotib color.

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

Returns
-------
paths : `~matplotlib.collections.PathCollection`

Other Parameters
----------------
**kwargs : `~matplotlib.collections.Collection` properties
```

```
        See Also
        --------
        plot : To plot scatter plots when markers are identical in size and
            color.

        Notes
        -----

        * The `.plot` function will be faster for scatterplots where markers
          don't vary in size or color.

        * Any or all of *x*, *y*, *s*, and *c* may be masked arrays, in which
          case all masks will be combined and only unmasked points will be
          plotted.

        * Fundamentally, scatter works with 1-D arrays; *x*, *y*, *s*, and *c*
          may be input as 2-D arrays, but within scatter they will be
          flattened. The exception is *c*, which will be flattened only if its
          size matches the size of *x* and *y*.

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

            * All arguments with the following names: 'c', 'color', 'edgecolor
s', 'facecolor', 'facecolors', 'linewidths', 's', 'x', 'y'.

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

    sci(im)
        Set the current image.

        This image will be the target of colormap functions like
        `~.pyplot.viridis`, and other functions such as `~.pyplot.clim`.  The
        current image is an attribute of the current axes.

    semilogx(*args, **kwargs)
        Make a plot with log scaling on the x axis.

        Call signatures::

            semilogx([x], y, [fmt], data=None, **kwargs)
            semilogx([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

        This is just a thin wrapper around `.plot` which additionally changes
        the x-axis to log scaling. All of the concepts and parameters of plot
        can be used here as well.

        The additional parameters *basex*, *subsx* and *nonposx* control the
        x-axis properties. They are just forwarded to `.Axes.set_xscale`.

        Parameters
```

```
    ----------
    basex : scalar, optional, default 10
        Base of the x logarithm.

    subsx : array_like, optional
        The location of the minor xticks. If *None*, reasonable locations
        are automatically chosen depending on the number of decades in the
        plot. See `.Axes.set_xscale` for details.

    nonposx : {'mask', 'clip'}, optional, default 'mask'
        Non-positive values in x can be masked as invalid, or clipped to a
        very small positive number.

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

    Other Parameters
    ----------------
    **kwargs
        All parameters supported by `.plot`.

semilogy(*args, **kwargs)
    Make a plot with log scaling on the y axis.

    Call signatures::

        semilogy([x], y, [fmt], data=None, **kwargs)
        semilogy([x], y, [fmt], [x2], y2, [fmt2], ..., **kwargs)

    This is just a thin wrapper around `.plot` which additionally changes
    the y-axis to log scaling. All of the concepts and parameters of plot
    can be used here as well.

    The additional parameters *basey*, *subsy* and *nonposy* control the
    y-axis properties. They are just forwarded to `.Axes.set_yscale`.

    Parameters
    ----------
    basey : scalar, optional, default 10
        Base of the y logarithm.

    subsy : array_like, optional
        The location of the minor yticks. If *None*, reasonable locations
        are automatically chosen depending on the number of decades in the
        plot. See `.Axes.set_yscale` for details.

    nonposy : {'mask', 'clip'}, optional, default 'mask'
        Non-positive values in y can be masked as invalid, or clipped to a
        very small positive number.

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

```
    Other Parameters
    ----------------
    **kwargs
        All parameters supported by `.plot`.

set_cmap(cmap)
    Set the default colormap.  Applies to the current image if any.
    See help(colormaps) for more information.

    *cmap* must be a :class:`~matplotlib.colors.Colormap` instance, or
    the name of a registered colormap.

    See :func:`matplotlib.cm.register_cmap` and
    :func:`matplotlib.cm.get_cmap`.

setp(obj, *args, **kwargs)
    Set a property on an artist object.

    matplotlib supports the use of :func:`setp` ("set property") and
    :func:`getp` to set and get object properties, as well as to do
    introspection on the object.  For example, to set the linestyle of a
    line to be dashed, you can do::

      >>> line, = plot([1,2,3])
      >>> setp(line, linestyle='--')

    If you want to know the valid types of arguments, you can provide
    the name of the property you want to set without a value::

      >>> setp(line, 'linestyle')
          linestyle: [ '-' | '--' | '-.' | ':' | 'steps' | 'None' ]

    If you want to see all the properties that can be set, and their
    possible values, you can do::

      >>> setp(line)
          ... long output listing omitted

    You may specify another output file to `setp` if `sys.stdout` is not
    acceptable for some reason using the `file` keyword-only argument::

      >>> with fopen('output.log') as f:
      >>>     setp(line, file=f)

    :func:`setp` operates on a single instance or a iterable of
    instances. If you are in query mode introspecting the possible
    values, only the first instance in the sequence is used. When
    actually setting values, all the instances will be set.  e.g.,
    suppose you have a list of two lines, the following will make both
    lines thicker and red::

      >>> x = arange(0,1.0,0.01)
      >>> y1 = sin(2*pi*x)
      >>> y2 = sin(4*pi*x)
      >>> lines = plot(x, y1, x, y2)
      >>> setp(lines, linewidth=2, color='r')
```

```
        :func:`setp` works with the MATLAB style string/value pairs or
        with python kwargs.  For example, the following are equivalent::

          >>> setp(lines, 'linewidth', 2, 'color', 'r')  # MATLAB style
          >>> setp(lines, linewidth=2, color='r')        # python style

    show(*args, **kw)
        Display a figure.
        When running in ipython with its pylab mode, display all
        figures and return to the ipython prompt.

        In non-interactive mode, display all figures and block until
        the figures have been closed; in interactive mode it has no
        effect unless figures were created prior to a change from
        non-interactive to interactive mode (not recommended).  In
        that case it displays the figures but does not block.

        A single experimental keyword argument, *block*, may be
        set to True or False to override the blocking behavior
        described above.

    specgram(x, NFFT=None, Fs=None, Fc=None, detrend=None, window=None, noverla
p=None, cmap=None, xextent=None, pad_to=None, sides=None, scale_by_freq=None, m
ode=None, scale=None, vmin=None, vmax=None, *, data=None, **kwargs)
        Plot a spectrogram.

        Call signature::

          specgram(x, NFFT=256, Fs=2, Fc=0, detrend=mlab.detrend_none,
                   window=mlab.window_hanning, noverlap=128,
                   cmap=None, xextent=None, pad_to=None, sides='default',
                   scale_by_freq=None, mode='default', scale='default',
                   **kwargs)

        Compute and plot a spectrogram of data in *x*.  Data are split into
        *NFFT* length segments and the spectrum of each section is
        computed.  The windowing function *window* is applied to each
        segment, and the amount of overlap of each segment is
        specified with *noverlap*. The spectrogram is plotted as a colormap
        (using imshow).

        Parameters
        ----------
        x : 1-D array or sequence
            Array or sequence containing the data.

        Fs : scalar
            The sampling frequency (samples per time unit).  It is used
            to calculate the Fourier frequencies, freqs, in cycles per time
            unit. The default value is 2.

        window : callable or ndarray
            A function or a vector of length *NFFT*. To create window
            vectors see :func:`window_hanning`, :func:`window_none`,
            :func:`numpy.blackman`, :func:`numpy.hamming`,
            :func:`numpy.bartlett`, :func:`scipy.signal`,
            :func:`scipy.signal.get_window`, etc. The default is
```

```
        :func:`window_hanning`.  If a function is passed as the
        argument, it must take a data segment as an argument and
        return the windowed version of the segment.

    sides : {'default', 'onesided', 'twosided'}
        Specifies which sides of the spectrum to return.  Default gives the
        default behavior, which returns one-sided for real data and both
        for complex data.  'onesided' forces the return of a one-sided
        spectrum, while 'twosided' forces two-sided.

    pad_to : int
        The number of points to which the data segment is padded when
        performing the FFT.  This can be different from *NFFT*, which
        specifies the number of data points used.  While not increasing
        the actual resolution of the spectrum (the minimum distance between
        resolvable peaks), this can give more points in the plot,
        allowing for more detail. This corresponds to the *n* parameter
        in the call to fft(). The default is None, which sets *pad_to*
        equal to *NFFT*

    NFFT : int
        The number of data points used in each block for the FFT.
        A power 2 is most efficient.  The default value is 256.
        This should *NOT* be used to get zero padding, or the scaling of th
e
        result will be incorrect. Use *pad_to* for this instead.

    detrend : {'default', 'constant', 'mean', 'linear', 'none'} or callable
        The function applied to each segment before fft-ing,
        designed to remove the mean or linear trend.  Unlike in
        MATLAB, where the *detrend* parameter is a vector, in
        matplotlib is it a function.  The :mod:`~matplotlib.mlab`
        module defines :func:`~matplotlib.mlab.detrend_none`,
        :func:`~matplotlib.mlab.detrend_mean`, and
        :func:`~matplotlib.mlab.detrend_linear`, but you can use
        a custom function as well.  You can also use a string to choose
        one of the functions.  'default', 'constant', and 'mean' call
        :func:`~matplotlib.mlab.detrend_mean`.  'linear' calls
        :func:`~matplotlib.mlab.detrend_linear`.  'none' calls
        :func:`~matplotlib.mlab.detrend_none`.

    scale_by_freq : bool, optional
        Specifies whether the resulting density values should be scaled
        by the scaling frequency, which gives density in units of Hz^-1.
        This allows for integration over the returned frequency values.
        The default is True for MATLAB compatibility.

    mode : {'default', 'psd', 'magnitude', 'angle', 'phase'}
        What sort of spectrum to use.  Default is 'psd', which takes
        the power spectral density.  'complex' returns the complex-valued
        frequency spectrum.  'magnitude' returns the magnitude spectrum.
        'angle' returns the phase spectrum without unwrapping.  'phase'
        returns the phase spectrum with unwrapping.

    noverlap : int
        The number of points of overlap between blocks.  The
        default value is 128.
```

```
scale : {'default', 'linear', 'dB'}
    The scaling of the values in the *spec*.  'linear' is no scaling.
    'dB' returns the values in dB scale.  When *mode* is 'psd',
    this is dB power (10 * log10).  Otherwise this is dB amplitude
    (20 * log10). 'default' is 'dB' if *mode* is 'psd' or
    'magnitude' and 'linear' otherwise.  This must be 'linear'
    if *mode* is 'angle' or 'phase'.

Fc : int
    The center frequency of *x* (defaults to 0), which offsets
    the x extents of the plot to reflect the frequency range used
    when a signal is acquired and then filtered and downsampled to
    baseband.

cmap :
    A :class:`matplotlib.colors.Colormap` instance; if *None*, use
    default determined by rc

xextent : *None* or (xmin, xmax)
    The image extent along the x-axis. The default sets *xmin* to the
    left border of the first bin (*spectrum* column) and *xmax* to the
    right border of the last bin. Note that for *noverlap>0* the width
    of the bins is smaller than those of the segments.

**kwargs :
    Additional kwargs are passed on to imshow which makes the
    specgram image.

Returns
-------
spectrum : 2-D array
    Columns are the periodograms of successive segments.

freqs : 1-D array
    The frequencies corresponding to the rows in *spectrum*.

t : 1-D array
    The times corresponding to midpoints of segments (i.e., the columns
    in *spectrum*).

im : instance of class :class:`~matplotlib.image.AxesImage`
    The image created by imshow containing the spectrogram

See Also
--------
:func:`psd`
    :func:`psd` differs in the default overlap; in returning the mean
    of the segment periodograms; in not returning times; and in
    generating a line plot instead of colormap.

:func:`magnitude_spectrum`
    A single spectrum, similar to having a single segment when *mode*
    is 'magnitude'. Plots a line instead of a colormap.

:func:`angle_spectrum`
    A single spectrum, similar to having a single segment when *mode*
```

```
            is 'angle'. Plots a line instead of a colormap.

        :func:`phase_spectrum`
            A single spectrum, similar to having a single segment when *mode*
            is 'phase'. Plots a line instead of a colormap.

        Notes
        -----
        The parameters *detrend* and *scale_by_freq* do only apply when *mode*
        is set to 'psd'.

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

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

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

    spring()
        Set the colormap to "spring".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    spy(Z, precision=0, marker=None, markersize=None, aspect='equal', origin='u
pper', **kwargs)
        Plot the sparsity pattern of a 2D array.

        This visualizes the non-zero values of the array.

        Two plotting styles are available: image and marker. Both
        are available for full arrays, but only the marker style
        works for `scipy.sparse.spmatrix` instances.

        **Image style**

        If *marker* and *markersize* are *None*, `~.Axes.imshow` is used. Any
        extra remaining kwargs are passed to this method.

        **Marker style**

        If *Z* is a `scipy.sparse.spmatrix` or *marker* or *markersize* are
        *None*, a `~matplotlib.lines.Line2D` object will be returned with
        the value of marker determining the marker type, and any
        remaining kwargs passed to `~.Axes.plot`.

        Parameters
        ----------
        Z : array-like (M, N)
            The array to be plotted.
```

```
            precision : float or 'present', optional, default: 0
                If *precision* is 0, any non-zero value will be plotted. Otherwise,
                values of :math:`|Z| > precision` will be plotted.

                For :class:`scipy.sparse.spmatrix` instances, you can also
                pass 'present'. In this case any value present in the array
                will be plotted, even if it is identically zero.

            origin : {'upper', 'lower'}, optional
                Place the [0,0] index of the array in the upper left or lower left
                corner of the axes. The convention 'upper' is typically used for
                matrices and images.
                If not given, :rc:`image.origin` is used, defaulting to 'upper'.


            aspect : {'equal', 'auto', None} or float, optional
                Controls the aspect ratio of the axes. The aspect is of particular
                relevance for images since it may distort the image, i.e. pixel
                will not be square.

                This parameter is a shortcut for explicitly calling
                `.Axes.set_aspect`. See there for further details.

                - 'equal': Ensures an aspect ratio of 1. Pixels will be square.
                - 'auto': The axes is kept fixed and the aspect is adjusted so
                  that the data fit in the axes. In general, this will result in
                  non-square pixels.
                - *None*: Use :rc:`image.aspect` (default: 'equal').

                Default: 'equal'

            Returns
            -------
            ret : `~matplotlib.image.AxesImage` or `.Line2D`
                The return type depends on the plotting style (see above).

            Other Parameters
            ----------------
            **kwargs
                The supported additional parameters depend on the plotting style.

                For the image style, you can pass the following additional
                parameters of `~.Axes.imshow`:

                - *cmap*
                - *alpha*
                - *url*
                - any `.Artist` properties (passed on to the `.AxesImage`)

                For the marker style, you can pass any `.Line2D` property except
                for *linestyle*:

                  agg_filter: a filter function, which takes a (m, n, 3) float arra
        y 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` | Non
e]
            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

    stackplot(x, *args, data=None, **kwargs)
        Draw a stacked area plot.

        Parameters
        ----------
        x : 1d array of dimension N

        y : 2d array (dimension MxN), or sequence of 1d arrays (each dimension
 1xN)

            The data is assumed to be unstacked. Each of the following
            calls is legal::

                stackplot(x, y)               # where y is MxN
                stackplot(x, y1, y2, y3, y4)  # where y1, y2, y3, y4, are all 1
xNm
```

```
        baseline : {'zero', 'sym', 'wiggle', 'weighted_wiggle'}
            Method used to calculate the baseline:

            - ``'zero'``: Constant zero baseline, i.e. a simple stacked plot.
            - ``'sym'``:  Symmetric around zero and is sometimes called
              'ThemeRiver'.
            - ``'wiggle'``: Minimizes the sum of the squared slopes.
            - ``'weighted_wiggle'``: Does the same but weights to account for
              size of each layer. It is also called 'Streamgraph'-layout. More
              details can be found at http://leebyron.com/streamgraph/. (htt
    p://leebyron.com/streamgraph/.)

        labels : Length N sequence of strings
            Labels to assign to each data series.

        colors : Length N sequence of colors
            A list or tuple of colors. These will be cycled through and used to
            colour the stacked areas.

        **kwargs :
            All other keyword arguments are passed to `Axes.fill_between()`.


        Returns
        -------
        list : list of `.PolyCollection`
            A list of `.PolyCollection` instances, one for each element in the
            stacked area plot.

    stem(*args, linefmt=None, markerfmt=None, basefmt=None, bottom=0, label=Non
    e, data=None)
        Create a stem plot.

        A stem plot plots vertical lines at each *x* location from the baseline
        to *y*, and places a marker there.

        Call signature::

          stem([x,] y, linefmt=None, markerfmt=None, basefmt=None)

        The x-positions are optional. The formats may be provided either as
        positional or as keyword-arguments.

        Parameters
        ----------
        x : array-like, optional
            The x-positions of the stems. Default: (0, 1, ..., len(y) - 1).

        y : array-like
            The y-values of the stem heads.

        linefmt : str, optional
            A string defining the properties of the vertical lines. Usually,
            this will be a color or a color and a linestyle:

            =========  =============
```

```
    Character  Line Style
    =========  ==========
    ``'-'``    solid line
    ``'--'``   dashed line
    ``'-.'``   dash-dot line
    ``':'``    dotted line
    =========  ==========

    Default: 'C0-', i.e. solid line with the first color of the color
    cycle.

    Note: While it is technically possible to specify valid formats
    other than color or color and linestyle (e.g. 'rx' or '-.'), this
    is beyond the intention of the method and will most likely not
    result in a reasonable reasonable plot.

markerfmt : str, optional
    A string defining the properties of the markers at the stem heads.
    Default: 'C0o', i.e. filled circles with the first color of the
    color cycle.

basefmt : str, optional
    A format string defining the properties of the baseline.

    Default: 'C3-' ('C2-' in classic mode).

bottom : float, optional, default: 0
    The y-position of the baseline.

label : str, optional, default: None
    The label to use for the stems in legends.


Returns
-------
container : :class:`~matplotlib.container.StemContainer`
    The container may be treated like a tuple
    (*markerline*, *stemlines*, *baseline*)


Notes
-----

.. seealso::
    The MATLAB function
    `stem <http://www.mathworks.com/help/techdoc/ref/stem.html>`_
    which inspired this method.

.. note::
    In addition to the above described arguments, this function can tak
e a
    **data** keyword argument. If such a **data** argument is given, th
e
    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``).

        step(x, y, *args, where='pre', data=None, **kwargs)
            Make a step plot.

            Call signatures::

                step(x, y, [fmt], *, data=None, where='pre', **kwargs)
                step(x, y, [fmt], x2, y2, [fmt2], ..., *, where='pre', **kwargs)

            This is just a thin wrapper around `.plot` which changes some
            formatting options. Most of the concepts and parameters of plot can be
            used here as well.

            Parameters
            ----------
            x : array_like
                1-D sequence of x positions. It is assumed, but not checked, that
                it is uniformly increasing.

            y : array_like
                1-D sequence of y levels.

            fmt : str, optional
                A format string, e.g. 'g' for a green line. See `.plot` for a more
                detailed description.

                Note: While full format strings are accepted, it is recommended to
                only specify the color. Line styles are currently ignored (use
                the keyword argument *linestyle* instead). Markers are accepted
                and plotted on the given positions, however, this is a rarely
                needed feature for step plots.

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

            where : {'pre', 'post', 'mid'}, optional, default 'pre'
                Define where the steps should be placed:

                - 'pre': The y value is continued constantly to the left from
                  every *x* position, i.e. the interval ``(x[i-1], x[i]]`` has the
                  value ``y[i]``.
                - 'post': The y value is continued constantly to the right from
                  every *x* position, i.e. the interval ``[x[i], x[i+1])`` has the
                  value ``y[i]``.
                - 'mid': Steps occur half-way between the *x* positions.

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

            Other Parameters
            ----------------
```

```
        **kwargs
            Additional parameters are the same as those for `.plot`.

        Notes
        -----
        .. [notes section required to get data note injection right]

        .. note::
            In addition to the above described arguments, this function can tak
e a
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    streamplot(x, y, u, v, density=1, linewidth=None, color=None, cmap=None, no
rm=None, arrowsize=1, arrowstyle='-|>', minlength=0.1, transform=None, zorder=N
one, start_points=None, maxlength=4.0, integration_direction='both', *, data=No
ne)
        Draw streamlines of a vector flow.

        *x*, *y* : 1d arrays
            an *evenly spaced* grid.
        *u*, *v* : 2d arrays
            x and y-velocities. Number of rows should match length of y, and
            the number of columns should match x.
        *density* : float or 2-tuple
            Controls the closeness of streamlines. When `density = 1`, the doma
in
            is divided into a 30x30 grid---*density* linearly scales this grid.
            Each cell in the grid can have, at most, one traversing streamline.
            For different densities in each direction, use [density_x, density_
y].
        *linewidth* : numeric or 2d array
            vary linewidth when given a 2d array with the same shape as velocit
ies.
        *color* : matplotlib color code, or 2d array
            Streamline color. When given an array with the same shape as
            velocities, *color* values are converted to colors using *cmap*.
        *cmap* : :class:`~matplotlib.colors.Colormap`
            Colormap used to plot streamlines and arrows. Only necessary when u
sing
            an array input for *color*.
        *norm* : :class:`~matplotlib.colors.Normalize`
            Normalize object used to scale luminance data to 0, 1. If None, str
etch
            (min, max) to (0, 1). Only necessary when *color* is an array.
        *arrowsize* : float
            Factor scale arrow size.
        *arrowstyle* : str
            Arrow style specification.
            See :class:`~matplotlib.patches.FancyArrowPatch`.
```

```
            *minlength* : float
                Minimum length of streamline in axes coordinates.
            *start_points*: Nx2 array
                Coordinates of starting points for the streamlines.
                In data coordinates, the same as the ``x`` and ``y`` arrays.
            *zorder* : int
                any number
            *maxlength* : float
                Maximum length of streamline in axes coordinates.
            *integration_direction* : ['forward', 'backward', 'both']
                Integrate the streamline in forward, backward or both directions.

            Returns:

                *stream_container* : StreamplotSet
                    Container object with attributes

                        - lines: `matplotlib.collections.LineCollection` of streaml
    ines

                        - arrows: collection of `matplotlib.patches.FancyArrowPatch
    `
                          objects representing arrows half-way along stream
                          lines.

                    This container will probably change in the future to allow chan
    ges
                    to the colormap, alpha, etc. for both lines and arrows, but the
    se
                    changes should be backward compatible.

        subplot(*args, **kwargs)
            Add a subplot to the current figure.

            Wrapper of `.Figure.add_subplot` with a difference in behavior
            explained in the notes section.

            Call signatures::

               subplot(nrows, ncols, index, **kwargs)
               subplot(pos, **kwargs)
               subplot(ax)

            Parameters
            ----------
            *args
                Either a 3-digit integer or three separate integers
                describing the position of the subplot. If the three
                integers are *nrows*, *ncols*, and *index* in order, the
                subplot will take the *index* position on a grid with *nrows*
                rows and *ncols* columns. *index* starts at 1 in the upper left
                corner and increases to the right.

                *pos* is a three digit integer, where the first digit is the
                number of rows, the second the number of columns, and the third
                the index of the subplot. i.e. fig.add_subplot(235) is the same as
                fig.add_subplot(2, 3, 5). Note that all integers must be less than
```

```
            10 for this form to work.

        projection : {None, 'aitoff', 'hammer', 'lambert', 'mollweide', 'pola
r', 'rectilinear', str}, optional
            The projection type of the subplot (`~.axes.Axes`). *str* is the na
me
            of a costum projection, see `~matplotlib.projections`. The default
            None results in a 'rectilinear' projection.

        polar : boolean, optional
            If True, equivalent to projection='polar'.

        sharex, sharey : `~.axes.Axes`, optional
            Share the x or y `~matplotlib.axis` with sharex and/or sharey. The
            axis will have the same limits, ticks, and scale as the axis of the
            shared axes.

        label : str
            A label for the returned axes.

        Other Parameters
        ----------------
        **kwargs
            This method also takes the keyword arguments for
            the returned axes base class. The keyword arguments for the
            rectilinear base class `~.axes.Axes` can be found in
            the following table but there might also be other keyword
            arguments if another projection is used.
              adjustable: {'box', 'datalim'}
          agg_filter: a filter function, which takes a (m, n, 3) float array an
d a dpi value, and returns a (m, n, 3) array
          alpha: float
          anchor: 2-tuple of floats or {'C', 'SW', 'S', 'SE', ...}
          animated: bool
          aspect: {'auto', 'equal'} or num
          autoscale_on: bool
          autoscalex_on: bool
          autoscaley_on: bool
          axes_locator: Callable[[Axes, Renderer], Bbox]
          axisbelow: bool or 'line'
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          contains: callable
          facecolor: color
          fc: color
          figure: `.Figure`
          frame_on: bool
          gid: str
          in_layout: bool
          label: object
          navigate: bool
          navigate_mode: unknown
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          position: [left, bottom, width, height] or `~matplotlib.transforms.Bb
```

```
ox`
          rasterization_zorder: float or None
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          title: str
          transform: `.Transform`
          url: str
          visible: bool
          xbound: unknown
          xlabel: str
          xlim: (left: float, right: float)
          xmargin: float greater than -0.5
          xscale: {"linear", "log", "symlog", "logit", ...}
          xticklabels: List[str]
          xticks: list
          ybound: unknown
          ylabel: str
          ylim: (bottom: float, top: float)
          ymargin: float greater than -0.5
          yscale: {"linear", "log", "symlog", "logit", ...}
          yticklabels: List[str]
          yticks: list
          zorder: float

        Returns
        -------
        axes : an `.axes.SubplotBase` subclass of `~.axes.Axes` (or a subclass
of `~.axes.Axes`)

            The axes of the subplot. The returned axes base class depends on
            the projection used. It is `~.axes.Axes` if rectilinear projection
            are used and `.projections.polar.PolarAxes` if polar projection
            are used. The returned axes is then a subplot subclass of the
            base class.

        Notes
        -----
        Creating a subplot will delete any pre-existing subplot that overlaps
        with it beyond sharing a boundary::

            import matplotlib.pyplot as plt
            # plot a line, implicitly creating a subplot(111)
            plt.plot([1,2,3])
            # now create a subplot which represents the top plot of a grid
            # with 2 rows and 1 column. Since this subplot will overlap the
            # first, the plot (and its axes) previously created, will be remove
d
            plt.subplot(211)

        If you do not want this behavior, use the `.Figure.add_subplot` method
        or the `.pyplot.axes` function instead.

        If the figure already has a subplot with key (*args*,
        *kwargs*) then it will simply make that subplot current and
        return it.  This behavior is deprecated. Meanwhile, if you do
        not want this behavior (i.e., you want to force the creation of a
```

```
    new suplot), you must use a unique set of args and kwargs.  The axes
    *label* attribute has been exposed for this purpose: if you want
    two subplots that are otherwise identical to be added to the figure,
    make sure you give them unique labels.

    In rare circumstances, `.add_subplot` may be called with a single
    argument, a subplot axes instance already created in the
    present figure but not in the figure's list of axes.

    See Also
    --------
    .Figure.add_subplot
    .pyplot.subplots
    .pyplot.axes
    .Figure.subplots

    Examples
    --------
    ::

        plt.subplot(221)

        # equivalent but more general
        ax1=plt.subplot(2, 2, 1)

        # add a subplot with no frame
        ax2=plt.subplot(222, frameon=False)

        # add a polar subplot
        plt.subplot(223, projection='polar')

        # add a red subplot that shares the x-axis with ax1
        plt.subplot(224, sharex=ax1, facecolor='red')

        #delete ax2 from the figure
        plt.delaxes(ax2)

        #add ax2 to the figure again
        plt.subplot(ax2)

subplot2grid(shape, loc, rowspan=1, colspan=1, fig=None, **kwargs)
    Create an axis at specific location inside a regular grid.

    Parameters
    ----------
    shape : sequence of 2 ints
        Shape of grid in which to place axis.
        First entry is number of rows, second entry is number of columns.

    loc : sequence of 2 ints
        Location to place axis within grid.
        First entry is row number, second entry is column number.

    rowspan : int
        Number of rows for the axis to span to the right.

    colspan : int
```

```
                Number of columns for the axis to span downwards.

            fig : `Figure`, optional
                Figure to place axis in. Defaults to current figure.

            **kwargs
                Additional keyword arguments are handed to `add_subplot`.


            Notes
            -----
            The following call ::

                subplot2grid(shape, loc, rowspan=1, colspan=1)

            is identical to ::

                gridspec=GridSpec(shape[0], shape[1])
                subplotspec=gridspec.new_subplotspec(loc, rowspan, colspan)
                subplot(subplotspec)

        subplot_tool(targetfig=None)
            Launch a subplot tool window for a figure.

            A :class:`matplotlib.widgets.SubplotTool` instance is returned.

        subplots(nrows=1, ncols=1, sharex=False, sharey=False, squeeze=True, subplo
    t_kw=None, gridspec_kw=None, **fig_kw)
            Create a figure and a set of subplots.

            This utility wrapper makes it convenient to create common layouts of
            subplots, including the enclosing figure object, in a single call.

            Parameters
            ----------
            nrows, ncols : int, optional, default: 1
                Number of rows/columns of the subplot grid.

            sharex, sharey : bool or {'none', 'all', 'row', 'col'}, default: False
                Controls sharing of properties among x (`sharex`) or y (`sharey`)
                axes:

                    - True or 'all': x- or y-axis will be shared among all
                      subplots.
                    - False or 'none': each subplot x- or y-axis will be
                      independent.
                    - 'row': each subplot row will share an x- or y-axis.
                    - 'col': each subplot column will share an x- or y-axis.

                When subplots have a shared x-axis along a column, only the x tick
                labels of the bottom subplot are created. Similarly, when subplots
                have a shared y-axis along a row, only the y tick labels of the fir
    st
                column subplot are created. To later turn other subplots' ticklabel
    s
                on, use `~matplotlib.axes.Axes.tick_params`.
```

```
        squeeze : bool, optional, default: True
            - If True, extra dimensions are squeezed out from the returned
              array of `~matplotlib.axes.Axes`:

                - if only one subplot is constructed (nrows=ncols=1), the
                  resulting single Axes object is returned as a scalar.
                - for Nx1 or 1xM subplots, the returned object is a 1D numpy
                  object array of Axes objects.
                - for NxM, subplots with N>1 and M>1 are returned as a 2D arra
y.

            - If False, no squeezing at all is done: the returned Axes object i
s
              always a 2D array containing Axes instances, even if it ends up
              being 1x1.

        num : integer or string, optional, default: None
            A `.pyplot.figure` keyword that sets the figure number or label.

        subplot_kw : dict, optional
            Dict with keywords passed to the
            `~matplotlib.figure.Figure.add_subplot` call used to create each
            subplot.

        gridspec_kw : dict, optional
            Dict with keywords passed to the `~matplotlib.gridspec.GridSpec`
            constructor used to create the grid the subplots are placed on.

        **fig_kw :
            All additional keyword arguments are passed to the
            `.pyplot.figure` call.

        Returns
        -------
        fig : `~.figure.Figure`

        ax : `.axes.Axes` object or array of Axes objects.
            *ax* can be either a single `~matplotlib.axes.Axes` object or an
            array of Axes objects if more than one subplot was created.  The
            dimensions of the resulting array can be controlled with the squeez
e
            keyword, see above.

        Examples
        --------
        ::

            #First create some toy data:
            x = np.linspace(0, 2*np.pi, 400)
            y = np.sin(x**2)

            #Creates just a figure and only one subplot
            fig, ax = plt.subplots()
            ax.plot(x, y)
            ax.set_title('Simple plot')

            #Creates two subplots and unpacks the output array immediately
```

```
                f, (ax1, ax2) = plt.subplots(1, 2, sharey=True)
                ax1.plot(x, y)
                ax1.set_title('Sharing Y axis')
                ax2.scatter(x, y)

                #Creates four polar axes, and accesses them through the returned ar
ray
                fig, axes = plt.subplots(2, 2, subplot_kw=dict(polar=True))
                axes[0, 0].plot(x, y)
                axes[1, 1].scatter(x, y)

                #Share a X axis with each column of subplots
                plt.subplots(2, 2, sharex='col')

                #Share a Y axis with each row of subplots
                plt.subplots(2, 2, sharey='row')

                #Share both X and Y axes with all subplots
                plt.subplots(2, 2, sharex='all', sharey='all')

                #Note that this is the same as
                plt.subplots(2, 2, sharex=True, sharey=True)

                #Creates figure number 10 with a single subplot
                #and clears it if it already exists.
                fig, ax=plt.subplots(num=10, clear=True)

            See Also
            --------
            .pyplot.figure
            .pyplot.subplot
            .pyplot.axes
            .Figure.subplots
            .Figure.add_subplot

        subplots_adjust(left=None, bottom=None, right=None, top=None, wspace=None,
 hspace=None)
            Tune the subplot layout.

            The parameter meanings (and suggested defaults) are::

              left  = 0.125  # the left side of the subplots of the figure
              right = 0.9    # the right side of the subplots of the figure
              bottom = 0.1   # the bottom of the subplots of the figure
              top = 0.9      # the top of the subplots of the figure
              wspace = 0.2   # the amount of width reserved for space between subpl
ots,
                             # expressed as a fraction of the average axis width
              hspace = 0.2   # the amount of height reserved for space between subp
lots,
                             # expressed as a fraction of the average axis height

            The actual defaults are controlled by the rc file

        summer()
            Set the colormap to "summer".
```

```
        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    suptitle(t, **kwargs)
        Add a centered title to the figure.

        Parameters
        ----------
        t : str
            The title text.

        x : float, default 0.5
            The x location of the text in figure coordinates.

        y : float, default 0.98
            The y location of the text in figure coordinates.

        horizontalalignment, ha : {'center', 'left', right'}, default: 'center'
            The horizontal alignment of the text relative to (*x*, *y*).

        verticalalignment, va : {'top', 'center', 'bottom', 'baseline'}, defaul
t: 'top'
            The vertical alignment of the text relative to (*x*, *y*).

        fontsize, size : default: :rc:`figure.titlesize`
            The font size of the text. See `.Text.set_size` for possible
            values.

        fontweight, weight : default: :rc:`figure.titleweight`
            The font weight of the text. See `.Text.set_weight` for possible
            values.


        Returns
        -------
            text
                The `.Text` instance of the title.


        Other Parameters
        ----------------
        fontproperties : None or dict, optional
            A dict of font properties. If *fontproperties* is given the
            default values for font size and weight are taken from the
            `FontProperties` defaults. :rc:`figure.titlesize` and
            :rc:`figure.titleweight` are ignored in this case.

        **kwargs
            Additional kwargs are :class:`matplotlib.text.Text` properties.


        Examples
        --------

        >>> fig.suptitle('This is the figure title', fontsize=12)
```

```
    switch_backend(newbackend)
        Close all open figures and set the Matplotlib backend.

        The argument is case-insensitive.  Switching to an interactive backend
 is
        possible only if no event loop for another interactive backend has star
ted.
        Switching to and from non-interactive backends is always possible.

        Parameters
        ----------
        newbackend : str
            The name of the backend to use.

    table(**kwargs)
        Add a table to the current axes.

        Call signature::

          table(cellText=None, cellColours=None,
                cellLoc='right', colWidths=None,
                rowLabels=None, rowColours=None, rowLoc='left',
                colLabels=None, colColours=None, colLoc='center',
                loc='bottom', bbox=None)

        Returns a :class:`matplotlib.table.Table` instance. Either `cellText`
        or `cellColours` must be provided. For finer grained control over
        tables, use the :class:`~matplotlib.table.Table` class and add it to
        the axes with :meth:`~matplotlib.axes.Axes.add_table`.

        Thanks to John Gill for providing the class and table.

        kwargs control the :class:`~matplotlib.table.Table`
        properties:

          agg_filter: a filter function, which takes a (m, n, 3) float array an
d a dpi value, and returns a (m, n, 3) array
          alpha: float
          animated: bool
          clip_box: `.Bbox`
          clip_on: bool
          clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | Non
e]
          contains: callable
          figure: `.Figure`
          fontsize: float
          gid: str
          in_layout: bool
          label: object
          path_effects: `.AbstractPathEffect`
          picker: None or bool or float or callable
          rasterized: bool or None
          sketch_params: (scale: float, length: float, randomness: float)
          snap: bool or None
          transform: `.Transform`
          url: str
          visible: bool
```

```
      zorder: float

text(x, y, s, fontdict=None, withdash=False, **kwargs)
    Add text to the axes.

    Add the text *s* to the axes at location *x*, *y* in data coordinates.

    Parameters
    ----------
    x, y : scalars
        The position to place the text. By default, this is in data
        coordinates. The coordinate system can be changed using the
        *transform* parameter.

    s : str
        The text.

    fontdict : dictionary, optional, default: None
        A dictionary to override the default text properties. If fontdict
        is None, the defaults are determined by your rc parameters.

    withdash : boolean, optional, default: False
        Creates a `~matplotlib.text.TextWithDash` instance instead of a
        `~matplotlib.text.Text` instance.

    Returns
    -------
    text : `.Text`
        The created `.Text` instance.

    Other Parameters
    ----------------
    **kwargs : `~matplotlib.text.Text` properties.
        Other miscellaneous text parameters.

    Examples
    --------
    Individual keyword arguments can be used to override any given
    parameter::

        >>> text(x, y, s, fontsize=12)

    The default transform specifies that text is in data coords,
    alternatively, you can specify text in axis coords (0,0 is
    lower-left and 1,1 is upper-right).  The example below places
    text in the center of the axes::

        >>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
        ...      verticalalignment='center', transform=ax.transAxes)

    You can put a rectangular box around the text instance (e.g., to
    set a background color) by using the keyword `bbox`.  `bbox` is
    a dictionary of `~matplotlib.patches.Rectangle`
    properties.  For example::

        >>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

```
    thetagrids(*args, **kwargs)
        Get or set the theta gridlines on the current polar plot.

        Call signatures::

         lines, labels = thetagrids()
         lines, labels = thetagrids(angles, labels=None, fmt=None, **kwargs)

        When called with no arguments, `.thetagrids` simply returns the tuple
        (*lines*, *labels*). When called with arguments, the labels will
        appear at the specified angles.

        Parameters
        ----------
        angles : tuple with floats, degrees
            The angles of the theta gridlines.

        labels : tuple with strings or None
            The labels to use at each radial gridline. The
            `.projections.polar.ThetaFormatter` will be used if None.

        fmt : str or None
            Format string used in `matplotlib.ticker.FormatStrFormatter`.
            For example '%f'. Note that the angle in radians will be used.

        Returns
        -------
        lines, labels : list of `.lines.Line2D`, list of `.text.Text`
            *lines* are the theta gridlines and *labels* are the tick labels.

        Other Parameters
        ----------------
        **kwargs
            *kwargs* are optional `~.Text` properties for the labels.

        Examples
        --------
        ::

          # set the locations of the angular gridlines
          lines, labels = thetagrids( range(45,360,90) )

          # set the locations and labels of the angular gridlines
          lines, labels = thetagrids( range(45,360,90), ('NE', 'NW', 'SW','SE')
)

        See Also
        --------
        .pyplot.rgrids
        .projections.polar.PolarAxes.set_thetagrids
        .Axis.get_gridlines
        .Axis.get_ticklabels

    tick_params(axis='both', **kwargs)
        Change the appearance of ticks, tick labels, and gridlines.

        Parameters
```

```
----------
axis : {'x', 'y', 'both'}, optional
    Which axis to apply the parameters to.

Other Parameters
----------------

axis : {'x', 'y', 'both'}
    Axis on which to operate; default is 'both'.

reset : bool
    If *True*, set all parameters to defaults
    before processing other keyword arguments.  Default is
    *False*.

which : {'major', 'minor', 'both'}
    Default is 'major'; apply arguments to *which* ticks.

direction : {'in', 'out', 'inout'}
    Puts ticks inside the axes, outside the axes, or both.

length : float
    Tick length in points.

width : float
    Tick width in points.

color : color
    Tick color; accepts any mpl color spec.

pad : float
    Distance in points between tick and label.

labelsize : float or str
    Tick label font size in points or as a string (e.g., 'large').

labelcolor : color
    Tick label color; mpl color spec.

colors : color
    Changes the tick color and the label color to the same value:
    mpl color spec.

zorder : float
    Tick and label zorder.

bottom, top, left, right : bool
    Whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright : bool
    Whether to draw the respective tick labels.

labelrotation : float
    Tick label rotation

grid_color : color
    Changes the gridline color to the given mpl color spec.
```

```
        grid_alpha : float
            Transparency of gridlines: 0 (transparent) to 1 (opaque).

        grid_linewidth : float
            Width of gridlines in points.

        grid_linestyle : string
            Any valid :class:`~matplotlib.lines.Line2D` line style spec.

        Examples
        --------

        Usage ::

            ax.tick_params(direction='out', length=6, width=2, colors='r',
                           grid_color='r', grid_alpha=0.5)

        This will make all major ticks be red, pointing out of the box,
        and with dimensions 6 points by 2 points.  Tick labels will
        also be red.  Gridlines will be red and translucent.

    ticklabel_format(*, axis='both', style='', scilimits=None, useOffset=None,
useLocale=None, useMathText=None)
        Change the `~matplotlib.ticker.ScalarFormatter` used by
        default for linear axes.

        Optional keyword arguments:

          ==============   =========================================
          Keyword          Description
          ==============   =========================================
          *axis*           [ 'x' | 'y' | 'both' ]
          *style*          [ 'sci' (or 'scientific') | 'plain' ]
                           plain turns off scientific notation
          *scilimits*      (m, n), pair of integers; if *style*
                           is 'sci', scientific notation will
                           be used for numbers outside the range
                           10\ :sup:`m` to 10\ :sup:`n`.
                           Use (0,0) to include all numbers.
                           Use (m,m) where m <> 0 to fix the order
                           of magnitude to 10\ :sup:`m`.
          *useOffset*      [ bool | offset ]; if True,
                           the offset will be calculated as needed;
                           if False, no offset will be used; if a
                           numeric offset is specified, it will be
                           used.
          *useLocale*      If True, format the number according to
                           the current locale.  This affects things
                           such as the character used for the
                           decimal separator.  If False, use
                           C-style (English) formatting.  The
                           default setting is controlled by the
                           axes.formatter.use_locale rcparam.
          *useMathText*    If True, render the offset and scientific
                           notation in mathtext
          ==============   =========================================
```

```
    Only the major ticks are affected.
    If the method is called when the
    :class:`~matplotlib.ticker.ScalarFormatter` is not the
    :class:`~matplotlib.ticker.Formatter` being used, an
    :exc:`AttributeError` will be raised.

tight_layout(pad=1.08, h_pad=None, w_pad=None, rect=None)
    Automatically adjust subplot parameters to give specified padding.

    Parameters
    ----------
    pad : float
        Padding between the figure edge and the edges of subplots,
        as a fraction of the font size.
    h_pad, w_pad : float, optional
        Padding (height/width) between edges of adjacent subplots,
        as a fraction of the font size.  Defaults to *pad*.
    rect : tuple (left, bottom, right, top), optional
        A rectangle (left, bottom, right, top) in the normalized
        figure coordinate that the whole subplots area (including
        labels) will fit into. Default is (0, 0, 1, 1).

title(label, fontdict=None, loc='center', pad=None, **kwargs)
    Set a title for the axes.

    Set one of the three available axes titles. The available titles
    are positioned above the axes in the center, flush with the left
    edge, and flush with the right edge.

    Parameters
    ----------
    label : str
        Text to use for the title

    fontdict : dict
        A dictionary controlling the appearance of the title text,
        the default `fontdict` is::

           {'fontsize': rcParams['axes.titlesize'],
            'fontweight' : rcParams['axes.titleweight'],
            'verticalalignment': 'baseline',
            'horizontalalignment': loc}

    loc : {'center', 'left', 'right'}, str, optional
        Which title to set, defaults to 'center'

    pad : float
        The offset of the title from the top of the axes, in points.
        Default is ``None`` to use rcParams['axes.titlepad'].

    Returns
    -------
    text : :class:`~matplotlib.text.Text`
        The matplotlib text instance representing the title

    Other Parameters
```

```
    ----------------
    **kwargs : `~matplotlib.text.Text` properties
        Other keyword arguments are text properties, see
        :class:`~matplotlib.text.Text` for a list of valid text
        properties.

tricontour(*args, **kwargs)
    Draw contours on an unstructured triangular grid.
    :func:`~matplotlib.pyplot.tricontour` and
    :func:`~matplotlib.pyplot.tricontourf` draw contour lines and
    filled contours, respectively.  Except as noted, function
    signatures and return values are the same for both versions.

    The triangulation can be specified in one of two ways; either::

        tricontour(triangulation, ...)

    where triangulation is a :class:`matplotlib.tri.Triangulation`
    object, or

    ::

        tricontour(x, y, ...)
        tricontour(x, y, triangles, ...)
        tricontour(x, y, triangles=triangles, ...)
        tricontour(x, y, mask=mask, ...)
        tricontour(x, y, triangles, mask=mask, ...)

    in which case a Triangulation object will be created.  See
    :class:`~matplotlib.tri.Triangulation` for a explanation of
    these possibilities.

    The remaining arguments may be::

        tricontour(..., Z)

    where *Z* is the array of values to contour, one per point
    in the triangulation.  The level values are chosen
    automatically.

    ::

        tricontour(..., Z, N)

    contour up to *N+1* automatically chosen contour levels
    (*N* intervals).

    ::

        tricontour(..., Z, V)

    draw contour lines at the values specified in sequence *V*,
    which must be in increasing order.

    ::

        tricontourf(..., Z, V)
```

```
fill the (len(*V*)-1) regions between the values in *V*,
which must be in increasing order.

::

    tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see
below for more details.

``C = tricontour(...)`` returns a
:class:`~matplotlib.contour.TriContourSet` object.

Optional keyword arguments:

    *colors*: [ *None* | string | (mpl_colors) ]
    If *None*, the colormap specified by cmap will be used.

    If a string, like 'r' or 'red', all levels will be plotted in this
    color.

    If a tuple of matplotlib color args (string, float, rgb, etc),
    different levels will be plotted in different colors in the order
    specified.

    *alpha*: float
    The alpha blending value

    *cmap*: [ *None* | Colormap ]
    A cm :class:`~matplotlib.colors.Colormap` instance or
    *None*. If *cmap* is *None* and *colors* is *None*, a
    default Colormap is used.

    *norm*: [ *None* | Normalize ]
    A :class:`matplotlib.colors.Normalize` instance for
    scaling data values to colors. If *norm* is *None* and
    *colors* is *None*, the default linear scaling is used.

    *levels* [level0, level1, ..., leveln]
    A list of floating point numbers indicating the level
    curves to draw, in increasing order; e.g., to draw just
    the zero contour pass ``levels=[0]``

    *origin*: [ *None* | 'upper' | 'lower' | 'image' ]
    If *None*, the first value of *Z* will correspond to the
    lower left corner, location (0,0). If 'image', the rc
    value for ``image.origin`` will be used.

    This keyword is not active if *X* and *Y* are specified in
    the call to contour.

    *extent*: [ *None* | (x0,x1,y0,y1) ]

    If *origin* is not *None*, then *extent* is interpreted as
    in :func:`matplotlib.pyplot.imshow`: it gives the outer
    pixel boundaries. In this case, the position of Z[0,0]
```

```
    is the center of the pixel, not a corner. If *origin* is
    *None*, then (*x0*, *y0*) is the position of Z[0,0], and
    (*x1*, *y1*) is the position of Z[-1,-1].

    This keyword is not active if *X* and *Y* are specified in
    the call to contour.

    *locator*: [ *None* | ticker.Locator subclass ]
    If *locator* is None, the default
    :class:`~matplotlib.ticker.MaxNLocator` is used. The
    locator is used to determine the contour levels if they
    are not given explicitly via the *V* argument.

    *extend*: [ 'neither' | 'both' | 'min' | 'max' ]
    Unless this is 'neither', contour levels are automatically
    added to one or both ends of the range so that all data
    are included. These added ranges are then mapped to the
    special colormap values which default to the ends of the
    colormap range, but can be set via
    :meth:`matplotlib.colors.Colormap.set_under` and
    :meth:`matplotlib.colors.Colormap.set_over` methods.

    *xunits*, *yunits*: [ *None* | registered units ]
    Override axis units by specifying an instance of a
    :class:`matplotlib.units.ConversionInterface`.


tricontour-only keyword arguments:

    *linewidths*: [ *None* | number | tuple of numbers ]
    If *linewidths* is *None*, defaults to rc:`lines.linewidth`.

    If a number, all levels will be plotted with this linewidth.

    If a tuple, different levels will be plotted with different
    linewidths in the order specified

    *linestyles*: [ *None* | 'solid' | 'dashed' | 'dashdot' | 'dotted'
]
    If *linestyles* is *None*, the 'solid' is used.

    *linestyles* can also be an iterable of the above strings
    specifying a set of linestyles to be used. If this
    iterable is shorter than the number of contour levels
    it will be repeated as necessary.

    If contour is using a monochrome colormap and the contour
    level is less than 0, then the linestyle specified
    in :rc:`contour.negative_linestyle` will be used.

tricontourf-only keyword arguments:

    *antialiased*: bool
    enable antialiasing

Note: tricontourf fills intervals that are closed at the top; that
is, for boundaries *z1* and *z2*, the filled region is::
```

```
            z1 < z <= z2

        There is one exception: if the lowest boundary coincides with
        the minimum value of the *z* array, then that minimum value
        will be included in the lowest interval.

    tricontourf(*args, **kwargs)
        Draw contours on an unstructured triangular grid.
        :func:`~matplotlib.pyplot.tricontour` and
        :func:`~matplotlib.pyplot.tricontourf` draw contour lines and
        filled contours, respectively.  Except as noted, function
        signatures and return values are the same for both versions.

        The triangulation can be specified in one of two ways; either::

            tricontour(triangulation, ...)

        where triangulation is a :class:`matplotlib.tri.Triangulation`
        object, or

        ::

            tricontour(x, y, ...)
            tricontour(x, y, triangles, ...)
            tricontour(x, y, triangles=triangles, ...)
            tricontour(x, y, mask=mask, ...)
            tricontour(x, y, triangles, mask=mask, ...)

        in which case a Triangulation object will be created.  See
        :class:`~matplotlib.tri.Triangulation` for a explanation of
        these possibilities.

        The remaining arguments may be::

            tricontour(..., Z)

        where *Z* is the array of values to contour, one per point
        in the triangulation.  The level values are chosen
        automatically.

        ::

            tricontour(..., Z, N)

        contour up to *N+1* automatically chosen contour levels
        (*N* intervals).

        ::

            tricontour(..., Z, V)

        draw contour lines at the values specified in sequence *V*,
        which must be in increasing order.

        ::
```

```
    tricontourf(..., Z, V)

fill the (len(*V*)-1) regions between the values in *V*,
which must be in increasing order.

::

    tricontour(Z, **kwargs)

Use keyword args to control colors, linewidth, origin, cmap ... see
below for more details.

``C = tricontour(...)`` returns a
:class:`~matplotlib.contour.TriContourSet` object.

Optional keyword arguments:

    *colors*: [ *None* | string | (mpl_colors) ]
    If *None*, the colormap specified by cmap will be used.

    If a string, like 'r' or 'red', all levels will be plotted in this
    color.

    If a tuple of matplotlib color args (string, float, rgb, etc),
    different levels will be plotted in different colors in the order
    specified.

    *alpha*: float
    The alpha blending value

    *cmap*: [ *None* | Colormap ]
    A cm :class:`~matplotlib.colors.Colormap` instance or
    *None*. If *cmap* is *None* and *colors* is *None*, a
    default Colormap is used.

    *norm*: [ *None* | Normalize ]
    A :class:`matplotlib.colors.Normalize` instance for
    scaling data values to colors. If *norm* is *None* and
    *colors* is *None*, the default linear scaling is used.

    *levels* [level0, level1, ..., leveln]
    A list of floating point numbers indicating the level
    curves to draw, in increasing order; e.g., to draw just
    the zero contour pass ``levels=[0]``

    *origin*: [ *None* | 'upper' | 'lower' | 'image' ]
    If *None*, the first value of *Z* will correspond to the
    lower left corner, location (0,0). If 'image', the rc
    value for ``image.origin`` will be used.

    This keyword is not active if *X* and *Y* are specified in
    the call to contour.

    *extent*: [ *None* | (x0,x1,y0,y1) ]

    If *origin* is not *None*, then *extent* is interpreted as
    in :func:`matplotlib.pyplot.imshow`: it gives the outer
```

```
        pixel boundaries. In this case, the position of Z[0,0]
        is the center of the pixel, not a corner. If *origin* is
        *None*, then (*x0*, *y0*) is the position of Z[0,0], and
        (*x1*, *y1*) is the position of Z[-1,-1].

        This keyword is not active if *X* and *Y* are specified in
        the call to contour.

        *locator*: [ *None* | ticker.Locator subclass ]
        If *locator* is None, the default
        :class:`~matplotlib.ticker.MaxNLocator` is used. The
        locator is used to determine the contour levels if they
        are not given explicitly via the *V* argument.

        *extend*: [ 'neither' | 'both' | 'min' | 'max' ]
        Unless this is 'neither', contour levels are automatically
        added to one or both ends of the range so that all data
        are included. These added ranges are then mapped to the
        special colormap values which default to the ends of the
        colormap range, but can be set via
        :meth:`matplotlib.colors.Colormap.set_under` and
        :meth:`matplotlib.colors.Colormap.set_over` methods.

        *xunits*, *yunits*: [ *None* | registered units ]
        Override axis units by specifying an instance of a
        :class:`matplotlib.units.ConversionInterface`.


    tricontour-only keyword arguments:

        *linewidths*: [ *None* | number | tuple of numbers ]
        If *linewidths* is *None*, defaults to rc:`lines.linewidth`.

        If a number, all levels will be plotted with this linewidth.

        If a tuple, different levels will be plotted with different
        linewidths in the order specified

        *linestyles*: [ *None* | 'solid' | 'dashed' | 'dashdot' | 'dotted'
]
        If *linestyles* is *None*, the 'solid' is used.

        *linestyles* can also be an iterable of the above strings
        specifying a set of linestyles to be used. If this
        iterable is shorter than the number of contour levels
        it will be repeated as necessary.

        If contour is using a monochrome colormap and the contour
        level is less than 0, then the linestyle specified
        in :rc:`contour.negative_linestyle` will be used.

    tricontourf-only keyword arguments:

        *antialiased*: bool
        enable antialiasing

    Note: tricontourf fills intervals that are closed at the top; that
```

```
    is, for boundaries *z1* and *z2*, the filled region is::

        z1 < z <= z2

    There is one exception: if the lowest boundary coincides with
    the minimum value of the *z* array, then that minimum value
    will be included in the lowest interval.

tripcolor(*args, **kwargs)
    Create a pseudocolor plot of an unstructured triangular grid.

    The triangulation can be specified in one of two ways; either::

      tripcolor(triangulation, ...)

    where triangulation is a :class:`matplotlib.tri.Triangulation`
    object, or

    ::

      tripcolor(x, y, ...)
      tripcolor(x, y, triangles, ...)
      tripcolor(x, y, triangles=triangles, ...)
      tripcolor(x, y, mask=mask, ...)
      tripcolor(x, y, triangles, mask=mask, ...)

    in which case a Triangulation object will be created.  See
    :class:`~matplotlib.tri.Triangulation` for a explanation of these
    possibilities.

    The next argument must be *C*, the array of color values, either
    one per point in the triangulation if color values are defined at
    points, or one per triangle in the triangulation if color values
    are defined at triangles. If there are the same number of points
    and triangles in the triangulation it is assumed that color
    values are defined at points; to force the use of color values at
    triangles use the kwarg ``facecolors=C`` instead of just ``C``.

    *shading* may be 'flat' (the default) or 'gouraud'. If *shading*
    is 'flat' and C values are defined at points, the color values
    used for each triangle are from the mean C of the triangle's
    three points. If *shading* is 'gouraud' then color values must be
    defined at points.

    The remaining kwargs are the same as for
    :meth:`~matplotlib.axes.Axes.pcolor`.

triplot(*args, **kwargs)
    Draw a unstructured triangular grid as lines and/or markers.

    The triangulation to plot can be specified in one of two ways;
    either::

      triplot(triangulation, ...)

    where triangulation is a :class:`matplotlib.tri.Triangulation`
    object, or
```

```
        ::

          triplot(x, y, ...)
          triplot(x, y, triangles, ...)
          triplot(x, y, triangles=triangles, ...)
          triplot(x, y, mask=mask, ...)
          triplot(x, y, triangles, mask=mask, ...)

        in which case a Triangulation object will be created.  See
        :class:`~matplotlib.tri.Triangulation` for a explanation of these
        possibilities.

        The remaining args and kwargs are the same as for
        :meth:`~matplotlib.axes.Axes.plot`.

        Return a list of 2 :class:`~matplotlib.lines.Line2D` containing
        respectively:

            - the lines plotted for triangles edges
            - the markers plotted for triangles nodes

    twinx(ax=None)
        Make a second axes that shares the *x*-axis.  The new axes will
        overlay *ax* (or the current axes if *ax* is *None*).  The ticks
        for *ax2* will be placed on the right, and the *ax2* instance is
        returned.

        .. seealso::

           :doc:`/gallery/subplots_axes_and_figures/two_scales`

    twiny(ax=None)
        Make a second axes that shares the *y*-axis.  The new axis will
        overlay *ax* (or the current axes if *ax* is *None*).  The ticks
        for *ax2* will be placed on the top, and the *ax2* instance is
        returned.

    uninstall_repl_displayhook()
        Uninstall the matplotlib display hook.

        .. warning

           Need IPython >= 2 for this to work.  For IPython < 2 will raise a
           ``NotImplementedError``

        .. warning

           If you are using vanilla python and have installed another
           display hook this will reset ``sys.displayhook`` to what ever
           function was there when matplotlib installed it's displayhook,
           possibly discarding your changes.

    violinplot(dataset, positions=None, vert=True, widths=0.5, showmeans=False,
showextrema=True, showmedians=False, points=100, bw_method=None, *, data=None)
        Make a violin plot.
```

```
Make a violin plot for each column of *dataset* or each vector in
sequence *dataset*.  Each filled area extends to represent the
entire data range, with optional lines at the mean, the median,
the minimum, and the maximum.

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

positions : array-like, default = [1, 2, ..., n]
  Sets the positions of the violins. The ticks and limits are
  automatically set to match the positions.

vert : bool, default = True.
  If true, creates a vertical violin plot.
  Otherwise, creates a horizontal violin plot.

widths : array-like, default = 0.5
  Either a scalar or a vector that sets the maximal width of
  each violin. The default is 0.5, which uses about half of the
  available horizontal space.

showmeans : bool, default = False
  If `True`, will toggle rendering of the means.

showextrema : bool, default = True
  If `True`, will toggle rendering of the extrema.

showmedians : bool, default = False
  If `True`, will toggle rendering of the medians.

points : scalar, default = 100
  Defines the number of points to evaluate each of the
  gaussian kernel density estimations at.

bw_method : str, scalar or callable, optional
  The method used to calculate the estimator bandwidth.  This can be
  'scott', 'silverman', a scalar constant or a callable.  If a
  scalar, this will be used directly as `kde.factor`.  If a
  callable, it should take a `GaussianKDE` instance as its only
  parameter and return a scalar. If None (default), 'scott' is used.

Returns
-------

result : dict
  A dictionary mapping each component of the violinplot to a
  list of the corresponding collection instances created. The
  dictionary has the following keys:

    - ``bodies``: A list of the
      :class:`matplotlib.collections.PolyCollection` instances
      containing the filled area of each violin.

    - ``cmeans``: A
      :class:`matplotlib.collections.LineCollection` instance
```

```
                created to identify the mean values of each of the
                violin's distribution.

            - ``cmins``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the bottom of each violin's
              distribution.

            - ``cmaxes``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the top of each violin's
              distribution.

            - ``cbars``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the centers of each violin's
              distribution.

            - ``cmedians``: A
              :class:`matplotlib.collections.LineCollection` instance
              created to identify the median values of each of the
              violin's distribution.

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

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

            * All arguments with the following names: 'dataset'.

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

    viridis()
        Set the colormap to "viridis".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    vlines(x, ymin, ymax, colors='k', linestyles='solid', label='', *, data=Non
e, **kwargs)
        Plot vertical lines.

        Plot vertical lines at each *x* from *ymin* to *ymax*.

        Parameters
        ----------
        x : scalar or 1D array_like
            x-indexes where to plot the lines.
```

```
        ymin, ymax : scalar or 1D array_like
            Respective beginning and end of each line. If scalars are
            provided, all lines will have same length.

        colors : array_like of colors, optional, default: 'k'

        linestyles : {'solid', 'dashed', 'dashdot', 'dotted'}, optional

        label : string, optional, default: ''

        Returns
        -------
        lines : `~matplotlib.collections.LineCollection`

        Other Parameters
        ----------------
        **kwargs : `~matplotlib.collections.LineCollection` properties.

        See also
        --------
        hlines : horizontal lines
        axvline: vertical line across the axes

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

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

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

    waitforbuttonpress(*args, **kwargs)
        Blocking call to interact with the figure.

        This will return True is a key was pressed, False if a mouse
        button was pressed and None if *timeout* was reached without
        either being pressed.

        If *timeout* is negative, does not timeout.

    winter()
        Set the colormap to "winter".

        This changes the default colormap as well as the colormap of the curren
t
        image if there is one. See ``help(colormaps)`` for more information.

    xcorr(x, y, normed=True, detrend=<function detrend_none at 0x000001FEE7E93F
28>, usevlines=True, maxlags=10, *, data=None, **kwargs)
        Plot the cross correlation between *x* and *y*.
```

```
        The correlation with lag k is defined as
        :math:`\sum_n x[n+k] \cdot y^*[n]`, where :math:`y^*` is the complex
        conjugate of :math:`y`.

        Parameters
        ----------
        x : sequence of scalars of length n

        y : sequence of scalars of length n

        detrend : callable, optional, default: `mlab.detrend_none`
            *x* is detrended by the *detrend* callable. Default is no
            normalization.

        normed : bool, optional, default: True
            If ``True``, input vectors are normalised to unit length.

        usevlines : bool, optional, default: True
            If ``True``, `Axes.vlines` is used to plot the vertical lines from
            the origin to the acorr. Otherwise, `Axes.plot` is used.

        maxlags : int, optional
            Number of lags to show. If None, will return all ``2 * len(x) - 1``
            lags. Default is 10.

        Returns
        -------
        lags : array (length ``2*maxlags+1``)
            lag vector.
        c : array  (length ``2*maxlags+1``)
            auto correlation vector.
        line : `.LineCollection` or `.Line2D`
            `.Artist` added to the axes of the correlation

             `.LineCollection` if *usevlines* is True
             `.Line2D` if *usevlines* is False
        b : `.Line2D` or None
            Horizontal line at 0 if *usevlines* is True
            None *usevlines* is False

        Other Parameters
        ----------------
        linestyle : `.Line2D` property, optional
            Only used if usevlines is ``False``.

        marker : string, optional
            Default is 'o'.

        Notes
        -----
        The cross correlation is performed with :func:`numpy.correlate` with
        ``mode = 2``.

        .. note::
            In addition to the above described arguments, this function can tak
e a
```

```
            **data** keyword argument. If such a **data** argument is given, th
e
            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``).

    xkcd(scale=1, length=100, randomness=2)
        Turn on `xkcd <https://xkcd.com/>`_ sketch-style drawing mode.
        This will only have effect on things drawn after this function is
        called.

        For best results, the "Humor Sans" font should be installed: it is
        not included with matplotlib.

        Parameters
        ----------
        scale : float, optional
            The amplitude of the wiggle perpendicular to the source line.
        length : float, optional
            The length of the wiggle along the line.
        randomness : float, optional
            The scale factor by which the length is shrunken or expanded.

        Notes
        -----
        This function works by a number of rcParams, so it will probably
        override others you have set before.

        If you want the effects of this function to be temporary, it can
        be used as a context manager, for example::

            with plt.xkcd():
                # This figure will be in XKCD-style
                fig1 = plt.figure()
                # ...

            # This figure will be in regular style
            fig2 = plt.figure()

    xlabel(xlabel, fontdict=None, labelpad=None, **kwargs)
        Set the label for the x-axis.

        Parameters
        ----------
        xlabel : str
            The label text.

        labelpad : scalar, optional, default: None
            Spacing in points between the label and the x-axis.

        Other Parameters
        ----------------
        **kwargs : `.Text` properties
```

```
            `.Text` properties control the appearance of the label.

        See also
        --------
        text : for information on how override and the optional args work

    xlim(*args, **kwargs)
        Get or set the x limits of the current axes.

        Call signatures::

            left, right = xlim()  # return the current xlim
            xlim((left, right))   # set the xlim to left, right
            xlim(left, right)     # set the xlim to left, right

        If you do not specify args, you can pass *left* or *right* as kwargs,
        i.e.::

            xlim(right=3)  # adjust the right leaving left unchanged
            xlim(left=1)  # adjust the left leaving right unchanged

        Setting limits turns autoscaling off for the x-axis.

        Returns
        -------
        left, right
            A tuple of the new x-axis limits.

        Notes
        -----
        Calling this function with no arguments (e.g. ``xlim()``) is the pyplot
        equivalent of calling `~.Axes.get_xlim` on the current axes.
        Calling this function with arguments is the pyplot equivalent of callin
g
        `~.Axes.set_xlim` on the current axes. All arguments are passed though.

    xscale(value, **kwargs)
        Set the x-axis scale.

        Parameters
        ----------
        value : {"linear", "log", "symlog", "logit", ...}
            The axis scale type to apply.

        **kwargs
            Different keyword arguments are accepted, depending on the scale.
            See the respective class keyword arguments:

            - `matplotlib.scale.LinearScale`
            - `matplotlib.scale.LogScale`
            - `matplotlib.scale.SymmetricalLogScale`
            - `matplotlib.scale.LogitScale`


        Notes
        -----
        By default, Matplotlib supports the above mentioned scales.
```

```
        Additionally, custom scales may be registered using
        `matplotlib.scale.register_scale`. These scales can then also
        be used here.

    xticks(ticks=None, labels=None, **kwargs)
        Get or set the current tick locations and labels of the x-axis.

        Call signatures::

            locs, labels = xticks()           # Get locations and labels

            xticks(ticks, [labels], **kwargs)  # Set locations and labels

        Parameters
        ----------
        ticks : array_like
            A list of positions at which ticks should be placed. You can pass a
n
            empty list to disable xticks.

        labels : array_like, optional
            A list of explicit labels to place at the given *locs*.

        **kwargs
            :class:`.Text` properties can be used to control the appearance of
            the labels.

        Returns
        -------
        locs
            An array of label locations.
        labels
            A list of `.Text` objects.

        Notes
        -----
        Calling this function with no arguments (e.g. ``xticks()``) is the pypl
ot
        equivalent of calling `~.Axes.get_xticks` and `~.Axes.get_xticklabels`
 on
        the current axes.
        Calling this function with arguments is the pyplot equivalent of callin
g
        `~.Axes.set_xticks` and `~.Axes.set_xticklabels` on the current axes.

        Examples
        --------
        Get the current locations and labels:

            >>> locs, labels = xticks()

        Set label locations:

            >>> xticks(np.arange(0, 1, step=0.2))

        Set text labels:
```

```
            >>> xticks(np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))

        Set text labels and properties:

            >>> xticks(np.arange(12), calendar.month_name[1:13], rotation=20)

        Disable xticks:

            >>> xticks([])

    ylabel(ylabel, fontdict=None, labelpad=None, **kwargs)
        Set the label for the y-axis.

        Parameters
        ----------
        ylabel : str
            The label text.

        labelpad : scalar, optional, default: None
            Spacing in points between the label and the y-axis.

        Other Parameters
        ----------------
        **kwargs : `.Text` properties
            `.Text` properties control the appearance of the label.

        See also
        --------
        text : for information on how override and the optional args work

    ylim(*args, **kwargs)
        Get or set the y-limits of the current axes.

        Call signatures::

            bottom, top = ylim()  # return the current ylim
            ylim((bottom, top))   # set the ylim to bottom, top
            ylim(bottom, top)     # set the ylim to bottom, top

        If you do not specify args, you can alternatively pass *bottom* or
        *top* as kwargs, i.e.::

            ylim(top=3)  # adjust the top leaving bottom unchanged
            ylim(bottom=1)  # adjust the bottom leaving top unchanged

        Setting limits turns autoscaling off for the y-axis.

        Returns
        -------
        bottom, top
            A tuple of the new y-axis limits.

        Notes
        -----
        Calling this function with no arguments (e.g. ``ylim()``) is the pyplot
        equivalent of calling `~.Axes.get_ylim` on the current axes.
        Calling this function with arguments is the pyplot equivalent of callin
```

```
g
        `~.Axes.set_ylim` on the current axes. All arguments are passed though.

    yscale(value, **kwargs)
        Set the y-axis scale.

        Parameters
        ----------
        value : {"linear", "log", "symlog", "logit", ...}
            The axis scale type to apply.

        **kwargs
            Different keyword arguments are accepted, depending on the scale.
            See the respective class keyword arguments:

            - `matplotlib.scale.LinearScale`
            - `matplotlib.scale.LogScale`
            - `matplotlib.scale.SymmetricalLogScale`
            - `matplotlib.scale.LogitScale`


        Notes
        -----
        By default, Matplotlib supports the above mentioned scales.
        Additionally, custom scales may be registered using
        `matplotlib.scale.register_scale`. These scales can then also
        be used here.

    yticks(ticks=None, labels=None, **kwargs)
        Get or set the current tick locations and labels of the y-axis.

        Call signatures::

            locs, labels = yticks()           # Get locations and labels

            yticks(ticks, [labels], **kwargs)  # Set locations and labels

        Parameters
        ----------
        ticks : array_like
            A list of positions at which ticks should be placed. You can pass a
n
            empty list to disable yticks.

        labels : array_like, optional
            A list of explicit labels to place at the given *locs*.

        **kwargs
            :class:`.Text` properties can be used to control the appearance of
            the labels.

        Returns
        -------
        locs
            An array of label locations.
        labels
            A list of `.Text` objects.
```

```
        Notes
        -----
        Calling this function with no arguments (e.g. ``yticks()``) is the pypl
ot
        equivalent of calling `~.Axes.get_yticks` and `~.Axes.get_yticklabels`
 on
        the current axes.
        Calling this function with arguments is the pyplot equivalent of callin
g
        `~.Axes.set_yticks` and `~.Axes.set_yticklabels` on the current axes.

        Examples
        --------
        Get the current locations and labels:

            >>> locs, labels = yticks()

        Set label locations:

            >>> yticks(np.arange(0, 1, step=0.2))

        Set text labels:

            >>> yticks(np.arange(5), ('Tom', 'Dick', 'Harry', 'Sally', 'Sue'))

        Set text labels and properties:

            >>> yticks(np.arange(12), calendar.month_name[1:13], rotation=45)

        Disable yticks:

            >>> yticks([])

DATA
    rcParams = RcParams({'_internal.classic_mode': False,
         ...nor.widt...
    rcParamsDefault = RcParams({'_internal.classic_mode': False,
         ...n...
    rcParamsOrig = RcParams({'_internal.classic_mode': False,
         ...nor....

FILE
    c:\users\alekhya\anaconda3\lib\site-packages\matplotlib\pyplot.py
```

```
C:\Users\Alekhya\Anaconda3\lib\site-packages\matplotlib\__init__.py:886: Matplo
tlibDeprecationWarning:
examples.directory is deprecated; in the future, examples will be found relativ
e to the 'datapath' directory.
  "found relative to the 'datapath' directory.".format(key))
C:\Users\Alekhya\Anaconda3\lib\site-packages\matplotlib\__init__.py:886: Matplo
tlibDeprecationWarning:
examples.directory is deprecated; in the future, examples will be found relativ
e to the 'datapath' directory.
```

```
  "found relative to the 'datapath' directory.".format(key))
C:\Users\Alekhya\Anaconda3\lib\site-packages\matplotlib\__init__.py:886: Matplo
tlibDeprecationWarning:
examples.directory is deprecated; in the future, examples will be found relativ
e to the 'datapath' directory.
  "found relative to the 'datapath' directory.".format(key))
```

- Line plot
- Box Plot
- Bar plot
- pie chart

In [41]:
```
1 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 an
d 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-post'}
      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

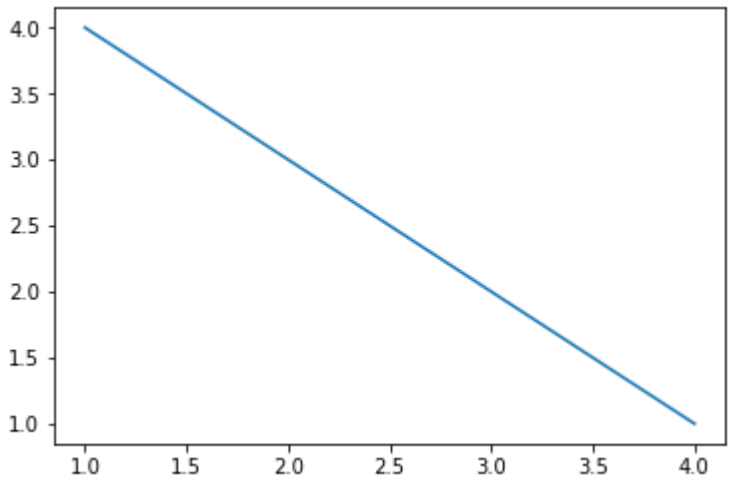
    .. note::
        In addition to the above described arguments, this function can take a
        **data** keyword argument. If such a **data** argument is given, the
        following arguments are replaced by **data[<arg>]**:

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

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

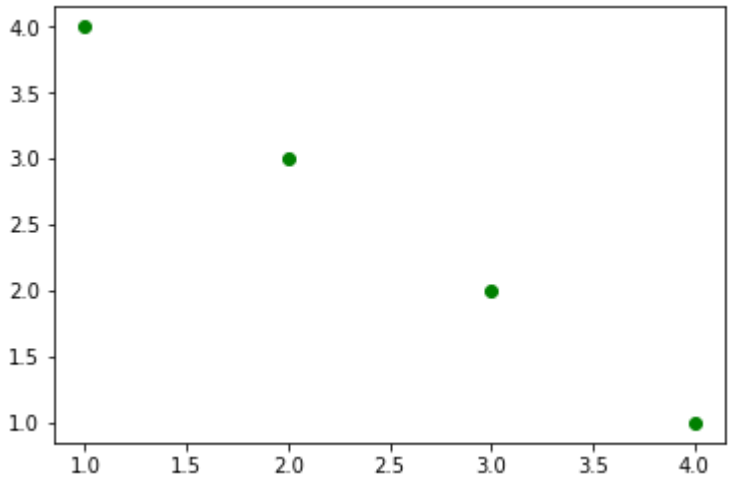
In [2]:
```
1 x = [1,2,3,4]
2 y = [4,3,2,1]
3 plt.plot(x,y)
```

Out[2]: [<matplotlib.lines.Line2D at 0x2242d2394e0>]



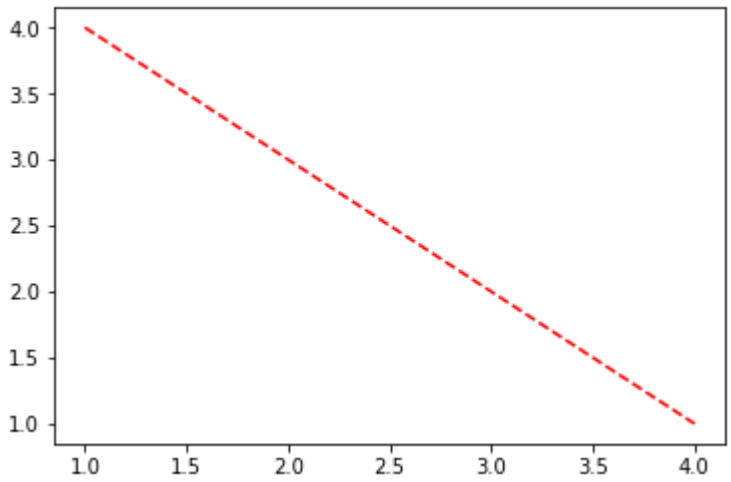
In [3]:
```
1 plt.plot(x,y,"go")
```

Out[3]: [<matplotlib.lines.Line2D at 0x2242d2de208>]



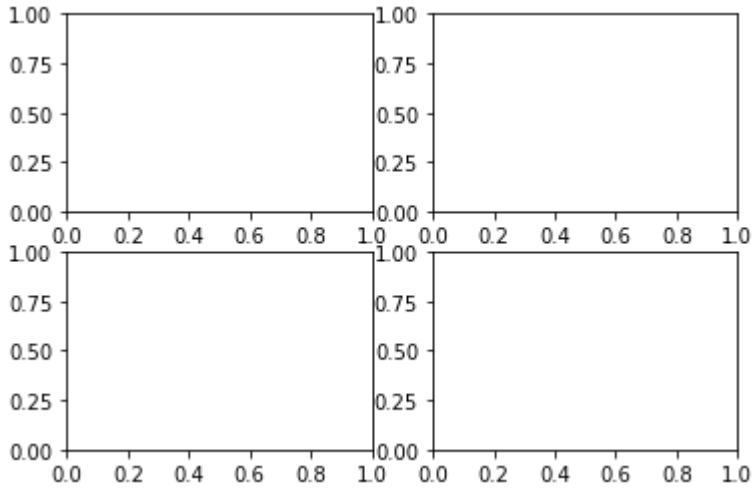
In [4]:
```
1 plt.plot(x,y,"r--")
```

Out[4]: [<matplotlib.lines.Line2D at 0x2242d343f28>]



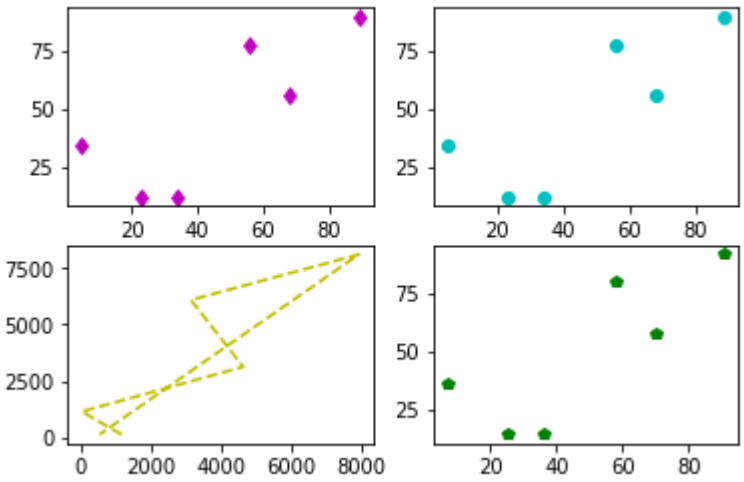
- subplots

In [8]:
```
1 plt.subplots(2,2)
2 plt.show()
```



In [17]:
```
import numpy as np
x = np.array([34,5,68,56,89,23])
y = np.array([12,34,56,78,90,12])
plt.subplot(2,2,1) # rows,columns,position
plt.plot(x,y,"md")
plt.subplot(2,2,2)
plt.plot(x,y,"co")
plt.subplot(2,2,3)
plt.plot(x**2,y**2,"y--")
plt.subplot(2,2,4)
plt.plot(x+2,y+2,"gp")
```

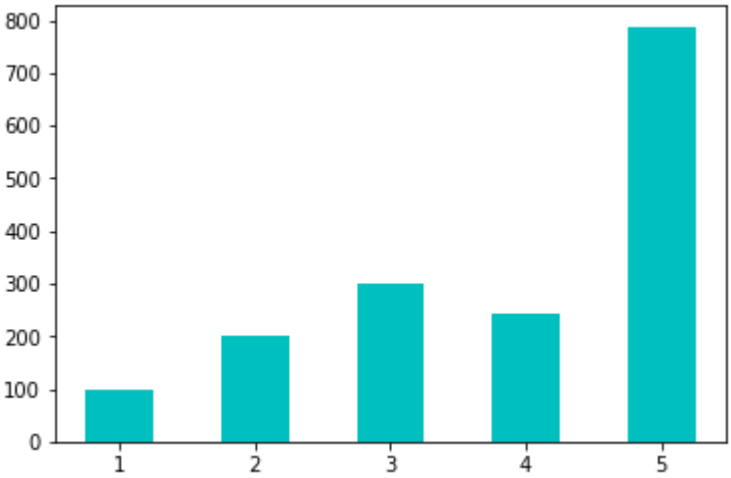
Out[17]:
```
[<matplotlib.lines.Line2D at 0x2242dabc3c8>]
```



Bar charts

In [20]:
```
plt.bar([1,2,3,4,5],[100,200,300,244,788],color = "c",width=0.5)
```

Out[20]:
```
<BarContainer object of 5 artists>
```



In [21]:
```
1 help(plt.bar)
```

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

bar(x, height, width=0.8, bottom=None, *, align='center', data=None, **kwargs)
    Make a bar plot.

    The bars are positioned at *x* with the given *align*\ment. Their
    dimensions are given by *width* and *height*. The vertical baseline
    is *bottom* (default 0).

    Each of *x*, *height*, *width*, and *bottom* may either be a scalar
    applying to all bars, or it may be a sequence of length N providing a
    separate value for each bar.

    Parameters
    ----------
    x : sequence of scalars
        The x coordinates of the bars. See also *align* for the
        alignment of the bars to the coordinates.

    height : scalar or sequence of scalars
        The height(s) of the bars.

    width : scalar or array-like, optional
        The width(s) of the bars (default: 0.8).

    bottom : scalar or array-like, optional
        The y coordinate(s) of the bars bases (default: 0).

    align : {'center', 'edge'}, optional, default: 'center'
        Alignment of the bars to the *x* coordinates:

        - 'center': Center the base on the *x* positions.
        - 'edge': Align the left edges of the bars with the *x* positions.

        To align the bars on the right edge pass a negative *width* and
        ``align='edge'``.

    Returns
    -------
    container : `.BarContainer`
        Container with all the bars and optionally errorbars.

    Other Parameters
    ----------------
    color : scalar or array-like, optional
        The colors of the bar faces.

    edgecolor : scalar or array-like, optional
        The colors of the bar edges.

    linewidth : scalar or array-like, optional
        Width of the bar edge(s). If 0, don't draw edges.

    tick_label : string or array-like, optional
```

```
        The tick labels of the bars.
        Default: None (Use default numeric labels.)

    xerr, yerr : scalar or array-like of shape(N,) or shape(2,N), optional
        If not *None*, add horizontal / vertical errorbars to the bar tips.
        The values are +/- sizes relative to the data:

        - scalar: symmetric +/- values for all bars
        - shape(N,): symmetric +/- values for each bar
        - shape(2,N): Separate - and + values for each bar. First row
            contains the lower errors, the second row contains the
            upper errors.
        - *None*: No errorbar. (Default)

        See :doc:`/gallery/statistics/errorbar_features`
        for an example on the usage of ``xerr`` and ``yerr``.

    ecolor : scalar or array-like, optional, default: 'black'
        The line color of the errorbars.

    capsize : scalar, optional
       The length of the error bar caps in points.
       Default: None, which will take the value from
       :rc:`errorbar.capsize`.

    error_kw : dict, optional
        Dictionary of kwargs to be passed to the `~.Axes.errorbar`
        method. Values of *ecolor* or *capsize* defined here take
        precedence over the independent kwargs.

    log : bool, optional, default: False
        If *True*, set the y-axis to be log scale.

    orientation : {'vertical',  'horizontal'}, optional
        *This is for internal use only.* Please use `barh` for
        horizontal bar plots. Default: 'vertical'.

    See also
    --------
    barh: Plot a horizontal bar plot.

    Notes
    -----
    The optional arguments *color*, *edgecolor*, *linewidth*,
    *xerr*, and *yerr* can be either scalars or sequences of
    length equal to the number of bars.  This enables you to use
    bar as the basis for stacked bar charts, or candlestick plots.
    Detail: *xerr* and *yerr* are passed directly to
    :meth:`errorbar`, so they can also have shape 2xN for
    independent specification of lower and upper errors.

    Other optional kwargs:

      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 or None
      animated: bool
```

```
        antialiased: unknown
        capstyle: {'butt', 'round', 'projecting'}
        clip_box: `.Bbox`
        clip_on: bool
        clip_path: [(`~matplotlib.path.Path`, `.Transform`) | `.Patch` | None]
        color: color
        contains: callable
        edgecolor: color or None or 'auto'
        facecolor: color or None
        figure: `.Figure`
        fill: bool
        gid: str
        hatch: {'/', '\\', '|', '-', '+', 'x', 'o', 'O', '.', '*'}
        in_layout: bool
        joinstyle: {'miter', 'round', 'bevel'}
        label: object
        linestyle: {'-', '--', '-.', ':', '', (offset, on-off-seq), ...}
        linewidth: float or None for default
        path_effects: `.AbstractPathEffect`
        picker: None or bool or float or callable
        rasterized: bool or None
        sketch_params: (scale: float, length: float, randomness: float)
        snap: bool or None
        transform: `.Transform`
        url: str
        visible: bool
        zorder: float

    .. 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>]**:

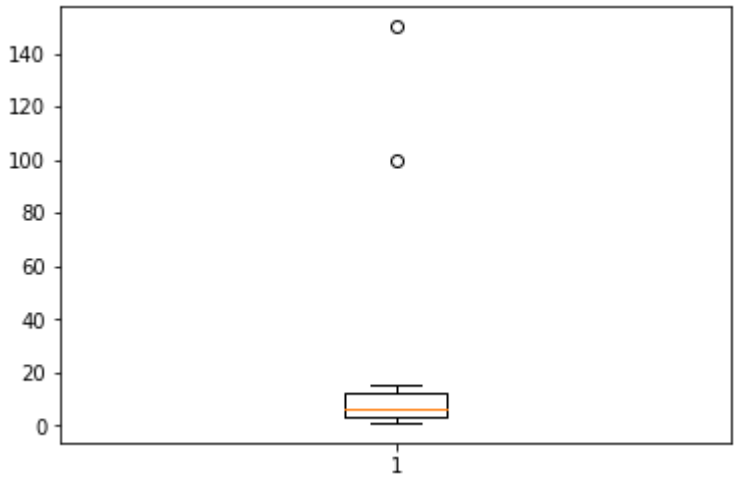
        * All arguments with the following names: 'bottom', 'color', 'ecolor',
'edgecolor', 'height', 'left', 'linewidth', 'tick_label', 'width', 'x', 'xerr',
'y', 'yerr'.
        * All positional arguments.

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

Box plot/whisker plot

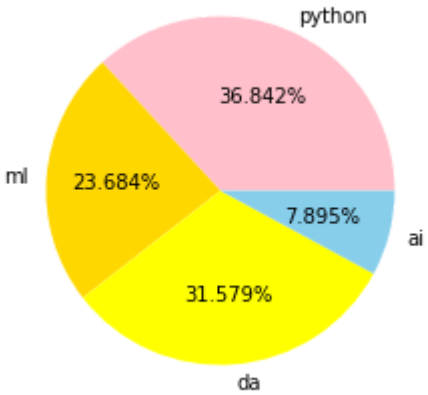
- to find the outliers

In [24]:
```
x = [1,2,3,4,5,6,7,10,15,100,150]
plt.boxplot(x)
plt.show()
```

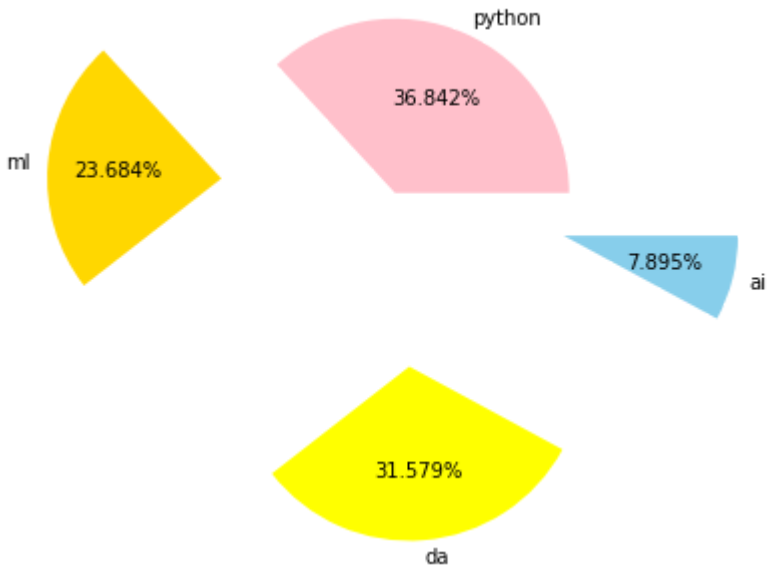


Piechart

In [31]:
```
a = ["python","ml","da","ai"]
b = [70,45,60,15]
c = ["pink","gold","yellow","skyblue"]
plt.pie(b,labels=a,autopct = '%1.3f%%',colors = c)
plt.show()
```



In [38]:
```
a = ["python","ml","da","ai"]
b = [70,45,60,15]
c = ["pink","gold","yellow","skyblue"]
e = (0,1,1,1)
plt.pie(b,labels=a,autopct = '%1.3f%%',colors = c,explode=e)
plt.show()
```



Seaborn

In [39]:
```
import seaborn as sns
```

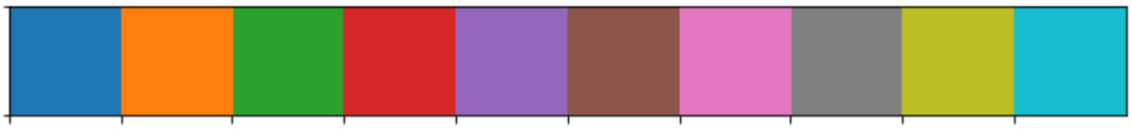
In [40]:
```
1 print(dir(sns))
```

```
['FacetGrid', 'JointGrid', 'PairGrid', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '_
_version__', '_orig_rc_params', 'algorithms', 'axes_style', 'axisgrid', 'barplo
t', 'blend_palette', 'boxenplot', 'boxplot', 'categorical', 'catplot', 'choose_
colorbrewer_palette', 'choose_cubehelix_palette', 'choose_dark_palette', 'choos
e_diverging_palette', 'choose_light_palette', 'clustermap', 'cm', 'color_palett
e', 'colors', 'countplot', 'crayon_palette', 'crayons', 'cubehelix_palette', 'd
ark_palette', 'desaturate', 'despine', 'distplot', 'distributions', 'diverging_
palette', 'dogplot', 'external', 'factorplot', 'get_dataset_names', 'heatmap',
'hls_palette', 'husl_palette', 'jointplot', 'kdeplot', 'light_palette', 'linepl
ot', 'lmplot', 'load_dataset', 'lvplot', 'matrix', 'miscplot', 'mpl', 'mpl_pale
tte', 'pairplot', 'palettes', 'palplot', 'plotting_context', 'pointplot', 'rcmo
d', 'regplot', 'regression', 'relational', 'relplot', 'reset_defaults', 'reset_
orig', 'residplot', 'rugplot', 'saturate', 'scatterplot', 'set', 'set_color_cod
es', 'set_context', 'set_hls_values', 'set_palette', 'set_style', 'stripplot',
'swarmplot', 'timeseries', 'tsplot', 'utils', 'violinplot', 'widgets', 'xkcd_pa
lette', 'xkcd_rgb']
```

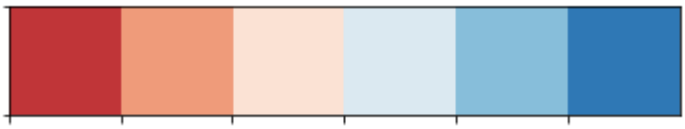
In [41]:
```
1 sns.color_palette()
```

Out[41]:
```
[(0.12156862745098039, 0.4666666666666667, 0.7058823529411765),
 (1.0, 0.4980392156862745, 0.054901960784313725),
 (0.17254901960784313, 0.6274509803921569, 0.17254901960784313),
 (0.8392156862745098, 0.15294117647058825, 0.1568627450980392),
 (0.5803921568627451, 0.403921568627451, 0.7411764705882353),
 (0.5490196078431373, 0.33725490196078434, 0.29411764705882354),
 (0.8901960784313725, 0.4666666666666667, 0.7607843137254902),
 (0.4980392156862745, 0.4980392156862745, 0.4980392156862745),
 (0.7372549019607844, 0.7411764705882353, 0.13333333333333333),
 (0.09019607843137255, 0.7450980392156863, 0.8117647058823529)]
```

In [42]:
```
1 sns.palplot(sns.color_palette())
```



In [47]:
```
1 sns.palplot(sns.color_palette("RdBu"))
```



In [49]:
```
1 sns.palplot(sns.color_palette("RdBu",15))
```

In [50]:
```
1 sns.palplot(sns.color_palette("Rdbu"))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\Anaconda3\lib\site-packages\seaborn\palettes.py in color_palette(palette, n
_colors, desat)
    231                 # Perhaps a named matplotlib colormap?
--> 232                 palette = mpl_palette(palette, n_colors)
    233             except ValueError:

~\Anaconda3\lib\site-packages\seaborn\palettes.py in mpl_palette(name, n_colo
rs)
    455     else:
--> 456         cmap = mpl.cm.get_cmap(name)
    457         if cmap is None:

~\Anaconda3\lib\site-packages\matplotlib\cm.py in get_cmap(name, lut)
    181             "Colormap %s is not recognized. Possible values are: %s"
--> 182             % (name, ', '.join(sorted(cmap_d))))
    183

ValueError: Colormap Rdbu is not recognized. Possible values are: Accent, Acc
ent_r, Blues, Blues_r, BrBG, BrBG_r, BuGn, BuGn_r, BuPu, BuPu_r, CMRmap, CMRm
ap_r, Dark2, Dark2_r, GnBu, GnBu_r, Greens, Greens_r, Greys, Greys_r, OrRd, O
rRd_r, Oranges, Oranges_r, PRGn, PRGn_r, Paired, Paired_r, Pastel1, Pastel1_
r, Pastel2, Pastel2_r, PiYG, PiYG_r, PuBu, PuBuGn, PuBuGn_r, PuBu_r, PuOr, Pu
Or_r, PuRd, PuRd_r, Purples, Purples_r, RdBu, RdBu_r, RdGy, RdGy_r, RdPu, RdP
u_r, RdYlBu, RdYlBu_r, RdYlGn, RdYlGn_r, Reds, Reds_r, Set1, Set1_r, Set2, Se
t2_r, Set3, Set3_r, Spectral, Spectral_r, Wistia, Wistia_r, YlGn, YlGnBu, YlG
nBu_r, YlGn_r, YlOrBr, YlOrBr_r, YlOrRd, YlOrRd_r, afmhot, afmhot_r, autumn,
 autumn_r, binary, binary_r, bone, bone_r, brg, brg_r, bwr, bwr_r, cividis, c
ividis_r, cool, cool_r, coolwarm, coolwarm_r, copper, copper_r, cubehelix, cu
behelix_r, flag, flag_r, gist_earth, gist_earth_r, gist_gray, gist_gray_r, gi
st_heat, gist_heat_r, gist_ncar, gist_ncar_r, gist_rainbow, gist_rainbow_r, g
ist_stern, gist_stern_r, gist_yarg, gist_yarg_r, gnuplot, gnuplot2, gnuplot2_
r, gnuplot_r, gray, gray_r, hot, hot_r, hsv, hsv_r, icefire, icefire_r, infer
no, inferno_r, jet, jet_r, magma, magma_r, mako, mako_r, nipy_spectral, nipy_
spectral_r, ocean, ocean_r, pink, pink_r, plasma, plasma_r, prism, prism_r, r
ainbow, rainbow_r, rocket, rocket_r, seismic, seismic_r, spring, spring_r, su
mmer, summer_r, tab10, tab10_r, tab20, tab20_r, tab20b, tab20b_r, tab20c, tab
20c_r, terrain, terrain_r, twilight, twilight_r, twilight_shifted, twilight_s
hifted_r, viridis, viridis_r, vlag, vlag_r, winter, winter_r

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-50-7d2c6edaec31> in <module>
----> 1 sns.palplot(sns.color_palette("Rdbu"))

~\Anaconda3\lib\site-packages\seaborn\palettes.py in color_palette(palette, n
_colors, desat)
    232                 palette = mpl_palette(palette, n_colors)
    233             except ValueError:
--> 234                 raise ValueError("%s is not a valid palette name" % p
alette)
    235
    236     if desat is not None:
```

```
250     if desat is not None:

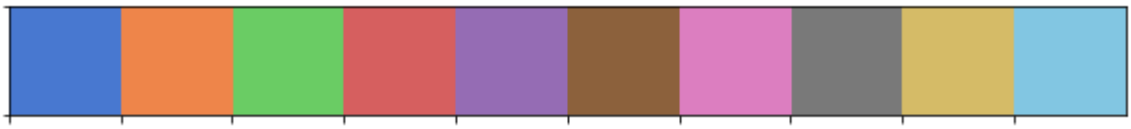
ValueError: Rdbu is not a valid palette name
```

In [53]:
```
1 sns.palplot(sns.color_palette("RdPu_r",n_colors=20))
```

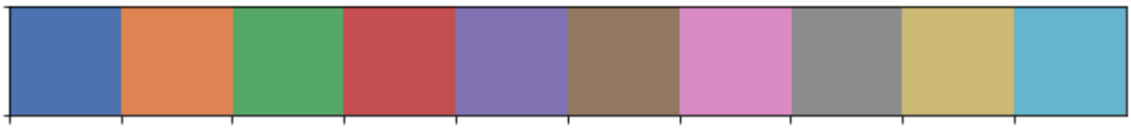
Seaborn is having 6 variations of its default color palette

- deep
- muted
- dark
- bright
- pastel
- colorblind

In [55]:
```
1 sns.palplot(sns.color_palette("muted"))
```

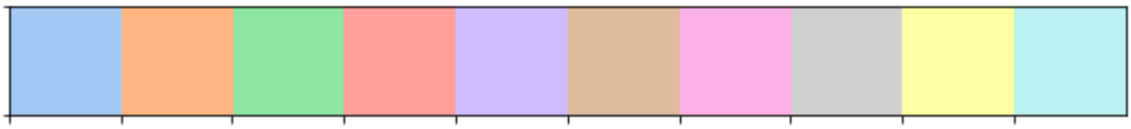


In [56]:
```
1 sns.palplot(sns.color_palette("deep"))
```



In [57]:
```
1 sns.palplot(sns.color_palette("bright"))
```

In [58]:
```
1 sns.palplot(sns.color_palette("pastel"))
```



In [59]:
```
1 sns.get_dataset_names()
```

```
C:\Users\Alekhya\Anaconda3\lib\site-packages\seaborn\utils.py:376: UserWarning:
No parser was explicitly specified, so I'm using the best available HTML parser
for this system ("lxml"). This usually isn't a problem, but if you run this cod
e on another system, or in a different virtual environment, it may use a differ
ent parser and behave differently.

The code that caused this warning is on line 376 of the file C:\Users\Alekhya\A
naconda3\lib\site-packages\seaborn\utils.py. To get rid of this warning, pass t
he additional argument 'features="lxml"' to the BeautifulSoup constructor.

  gh_list = BeautifulSoup(http)
```

Out[59]:
```
['anagrams',
 'anscombe',
 'attention',
 'brain_networks',
 'car_crashes',
 'diamonds',
 'dots',
 'exercise',
 'flights',
 'fmri',
 'gammas',
 'geyser',
 'iris',
 'mpg',
 'penguins',
 'planets',
 'tips',
 'titanic']
```

In [65]:
```
1 df = sns.load_dataset("iris")
2 df.head()
```

Out[65]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [69]:
```
1 df.shape
```

Out[69]:
```
(150, 5)
```

In [67]:
```
1 df["species"].value_counts()
```

Out[67]:
```
setosa        50
versicolor    50
virginica     50
Name: species, dtype: int64
```

In [63]:
```
1 df1 = sns.load_dataset("titanic")
2 df1.head()
```

Out[63]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN

In [70]:
```
1 df1["sex"].value_counts()
```

Out[70]:
```
male      577
female    314
Name: sex, dtype: int64
```

In [72]:
```
1 df.head()
```

Out[72]:

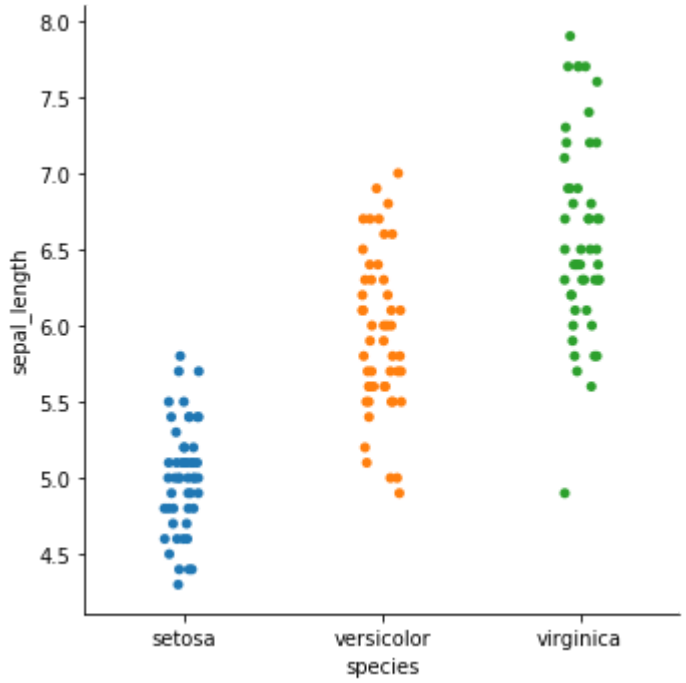
	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

In [73]:
```
1 df.columns
```

Out[73]:
```
Index(['sepal_length', 'sepal_width', 'petal_length', 'petal_width',
       'species'],
      dtype='object')
```

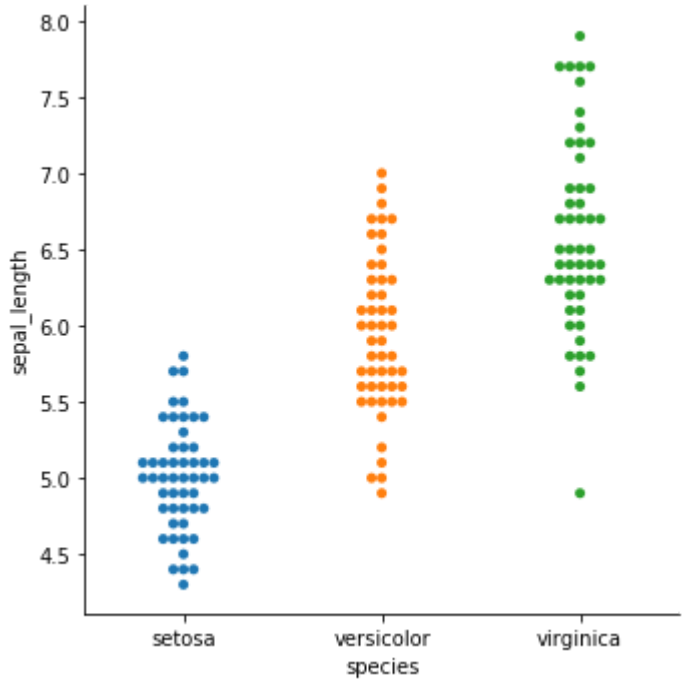
In [74]:
```
## Strip plot
sns.catplot(x ="species",y ="sepal_length",data=df)
```

Out[74]: <seaborn.axisgrid.FacetGrid at 0x22430dc9400>



In [75]:
```
## Swarm plot
# adjust the points automatically
# prevents from overlapping
sns.catplot(x ="species",y ="sepal_length",data=df,kind="swarm")
```

Out[75]: <seaborn.axisgrid.FacetGrid at 0x22431050828>



In [ ]:
```
1
```